

# Statistics and Machine Learning Toolbox™

## User's Guide



# MATLAB®

R2017b

 MathWorks®

# How to Contact MathWorks



Latest news: [www.mathworks.com](http://www.mathworks.com)  
Sales and services: [www.mathworks.com/sales\\_and\\_services](http://www.mathworks.com/sales_and_services)  
User community: [www.mathworks.com/matlabcentral](http://www.mathworks.com/matlabcentral)  
Technical support: [www.mathworks.com/support/contact\\_us](http://www.mathworks.com/support/contact_us)



Phone: 508-647-7000



The MathWorks, Inc.  
3 Apple Hill Drive  
Natick, MA 01760-2098

*Statistics and Machine Learning Toolbox™ User's Guide*

© COPYRIGHT 1993–2017 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

**FEDERAL ACQUISITION:** This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

## **Trademarks**

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See [www.mathworks.com/trademarks](http://www.mathworks.com/trademarks) for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

## **Patents**

MathWorks products are protected by one or more U.S. patents. Please see [www.mathworks.com/patents](http://www.mathworks.com/patents) for more information.

## Revision History

September 1993	First printing	Version 1.0
March 1996	Second printing	Version 2.0
January 1997	Third printing	Version 2.11
November 2000	Fourth printing	Revised for Version 3.0 (Release 12)
May 2001	Fifth printing	Minor revisions
July 2002	Sixth printing	Revised for Version 4.0 (Release 13)
February 2003	Online only	Revised for Version 4.1 (Release 13.0.1)
June 2004	Seventh printing	Revised for Version 5.0 (Release 14)
October 2004	Online only	Revised for Version 5.0.1 (Release 14SP1)
March 2005	Online only	Revised for Version 5.0.2 (Release 14SP2)
September 2005	Online only	Revised for Version 5.1 (Release 14SP3)
March 2006	Online only	Revised for Version 5.2 (Release 2006a)
September 2006	Online only	Revised for Version 5.3 (Release 2006b)
March 2007	Eighth printing	Revised for Version 6.0 (Release 2007a)
September 2007	Ninth printing	Revised for Version 6.1 (Release 2007b)
March 2008	Online only	Revised for Version 6.2 (Release 2008a)
October 2008	Online only	Revised for Version 7.0 (Release 2008b)
March 2009	Online only	Revised for Version 7.1 (Release 2009a)
September 2009	Online only	Revised for Version 7.2 (Release 2009b)
March 2010	Online only	Revised for Version 7.3 (Release 2010a)
September 2010	Online only	Revised for Version 7.4 (Release 2010b)
April 2011	Online only	Revised for Version 7.5 (Release 2011a)
September 2011	Online only	Revised for Version 7.6 (Release 2011b)
March 2012	Online only	Revised for Version 8.0 (Release 2012a)
September 2012	Online only	Revised for Version 8.1 (Release 2012b)
March 2013	Online only	Revised for Version 8.2 (Release 2013a)
September 2013	Online only	Revised for Version 8.3 (Release 2013b)
March 2014	Online only	Revised for Version 9.0 (Release 2014a)
October 2014	Online only	Revised for Version 9.1 (Release 2014b)
March 2015	Online only	Revised for Version 10.0 (Release 2015a)
September 2015	Online only	Revised for Version 10.1 (Release 2015b)
March 2016	Online only	Revised for Version 10.2 (Release 2016a)
September 2016	Online only	Revised for Version 11 (Release 2016b)
March 2017	Online only	Revised for Version 11.1 (Release 2017a)
September 2017	Online only	Revised for Version 11.2 (Release 2017b)



## Getting Started

1

### Statistics and Machine Learning Toolbox Product

<b>Description</b> .....	1-2
Key Features .....	1-2
<b>Supported Data Types</b> .....	1-3
<b>Statistics and Machine Learning Toolbox Functions with         gpuArray Arguments</b> .....	1-5

## Organizing Data

2

<b>Other MATLAB Functions Supporting Nominal and Ordinal         Arrays</b> .....	2-3
<b>Create Nominal and Ordinal Arrays</b> .....	2-4
Create Nominal Arrays .....	2-4
Create Ordinal Arrays .....	2-6
<b>Change Category Labels</b> .....	2-10
<b>Reorder Category Levels</b> .....	2-12
Reorder Category Levels in Ordinal Arrays .....	2-12
Reorder Category Levels in Nominal Arrays .....	2-14
<b>Categorize Numeric Data</b> .....	2-18
<b>Merge Category Levels</b> .....	2-22

<b>Add and Drop Category Levels</b> .....	<b>2-25</b>
<b>Plot Data Grouped by Category</b> .....	<b>2-29</b>
<b>Test Differences Between Category Means</b> .....	<b>2-34</b>
<b>Summary Statistics Grouped by Category</b> .....	<b>2-44</b>
<b>Sort Ordinal Arrays</b> .....	<b>2-46</b>
<b>Categorical Arrays</b> .....	<b>2-48</b>
What Are Categorical Arrays? .....	<b>2-48</b>
Categorical Array Conversion .....	<b>2-48</b>
<b>Advantages of Using Categorical Arrays</b> .....	<b>2-50</b>
Manipulate Category Levels .....	<b>2-50</b>
Analysis Using Categorical Arrays .....	<b>2-50</b>
Reduce Memory Requirements .....	<b>2-51</b>
<b>Index and Search Using Categorical Arrays</b> .....	<b>2-53</b>
Index By Category .....	<b>2-53</b>
Common Indexing and Searching Methods .....	<b>2-53</b>
<b>Grouping Variables</b> .....	<b>2-59</b>
What Are Grouping Variables? .....	<b>2-59</b>
Group Definition .....	<b>2-60</b>
Analysis Using Grouping Variables .....	<b>2-60</b>
Missing Group Values .....	<b>2-61</b>
<b>Dummy Indicator Variables</b> .....	<b>2-62</b>
What Are Dummy Variables? .....	<b>2-62</b>
Creating Dummy Variables .....	<b>2-63</b>
<b>Regression with Categorical Covariates</b> .....	<b>2-65</b>
<b>Create a Dataset Array from Workspace Variables</b> .....	<b>2-70</b>
Create a Dataset Array from a Numeric Array .....	<b>2-70</b>
Create Dataset Array from Heterogeneous Workspace Variables .....	<b>2-73</b>
<b>Create a Dataset Array from a File</b> .....	<b>2-77</b>
Create a Dataset Array from a Tab-Delimited Text File .....	<b>2-77</b>

Create a Dataset Array from a Comma-Separated Text File .....	2-80
Create a Dataset Array from an Excel File .....	2-82
<b>Add and Delete Observations .....</b>	<b>2-85</b>
<b>Add and Delete Variables .....</b>	<b>2-89</b>
<b>Access Data in Dataset Array Variables .....</b>	<b>2-93</b>
<b>Select Subsets of Observations .....</b>	<b>2-99</b>
<b>Sort Observations in Dataset Arrays .....</b>	<b>2-103</b>
<b>Merge Dataset Arrays .....</b>	<b>2-107</b>
<b>Stack or Unstack Dataset Arrays .....</b>	<b>2-112</b>
<b>Calculations on Dataset Arrays .....</b>	<b>2-117</b>
<b>Export Dataset Arrays .....</b>	<b>2-121</b>
<b>Clean Messy and Missing Data .....</b>	<b>2-123</b>
<b>Dataset Arrays in the Variables Editor .....</b>	<b>2-128</b>
Open Dataset Arrays in the Variables Editor .....	2-128
Modify Variable and Observation Names .....	2-129
Reorder or Delete Variables .....	2-131
Add New Data .....	2-133
Sort Observations .....	2-135
Select a Subset of Data .....	2-136
Create Plots .....	2-139
<b>Dataset Arrays .....</b>	<b>2-142</b>
What Are Dataset Arrays? .....	2-142
Dataset Array Conversion .....	2-142
Dataset Array Properties .....	2-143
<b>Index and Search Dataset Arrays .....</b>	<b>2-145</b>
Ways To Index and Search .....	2-145
Examples .....	2-145

**3**

<b>Introduction to Descriptive Statistics</b> .....	3-2
<b>Measures of Central Tendency</b> .....	3-3
Measures of Central Tendency .....	3-3
<b>Measures of Dispersion</b> .....	3-5
Compare Measures of Dispersion .....	3-6
<b>Quantiles and Percentiles</b> .....	3-7
<b>Exploratory Analysis of Data</b> .....	3-11
<b>Resampling Statistics</b> .....	3-17
Bootstrap Resampling .....	3-17
Jackknife Resampling .....	3-20
Parallel Computing Support for Resampling Methods .....	3-21
<b>Data with Missing Values</b> .....	3-22
Working with Data with Missing Values .....	3-22

**Statistical Visualization**

**4**

<b>Introduction to Statistical Visualization</b> .....	4-2
<b>Create Scatter Plots Using Grouped Data</b> .....	4-3
<b>Box Plots</b> .....	4-7
Compare Grouped Data Using Box Plots .....	4-7
<b>Distribution Plots</b> .....	4-10
Overview .....	4-10
Normal Probability Plots .....	4-10
Quantile-Quantile Plots .....	4-13
Cumulative Distribution Plots .....	4-15
Other Probability Plots .....	4-17



<b>Working with Probability Distributions</b> .....	5-3
Types of Probability Distributions .....	5-3
Probability Distribution Objects .....	5-4
Probability Distribution Functions .....	5-8
Probability Distribution Apps and User Interfaces .....	5-12
<b>Supported Distributions</b> .....	5-17
Continuous Distributions (Data) .....	5-18
Continuous Distributions (Statistics) .....	5-22
Discrete Distributions .....	5-23
Multivariate Distributions .....	5-25
Nonparametric Distributions .....	5-26
Flexible Distribution Families .....	5-26
<b>Maximum Likelihood Estimation</b> .....	5-27
<b>Negative Loglikelihood Functions</b> .....	5-30
<b>Random Number Generation</b> .....	5-34
<b>Nonparametric and Empirical Probability Distributions</b> ..	5-37
Overview .....	5-37
Kernel Distribution .....	5-37
Empirical Cumulative Distribution Function .....	5-39
Piecewise Linear Distribution .....	5-41
Pareto Tails .....	5-42
Triangular Distribution .....	5-43
<b>Fit Kernel Distribution Object to Data</b> .....	5-46
<b>Fit Kernel Distribution Using ksdensity</b> .....	5-51
<b>Fit Distributions to Grouped Data Using ksdensity</b> .....	5-54
<b>Fit a Nonparametric Distribution with Pareto Tails</b> .....	5-57
<b>Generate Random Numbers Using the Triangular Distribution</b> .....	5-62

<b>Explore the Probability Distribution Function UI</b> . . . . .	<b>5-67</b>
<b>Model Data Using the Distribution Fitter App</b> . . . . .	<b>5-70</b>
Explore Probability Distributions Interactively . . . . .	<b>5-70</b>
Create and Manage Data Sets . . . . .	<b>5-71</b>
Create a New Fit . . . . .	<b>5-76</b>
Display Results . . . . .	<b>5-81</b>
Manage Fits . . . . .	<b>5-83</b>
Evaluate Fits . . . . .	<b>5-85</b>
Exclude Data . . . . .	<b>5-89</b>
Save and Load Sessions . . . . .	<b>5-94</b>
Generate a File to Fit and Plot Distributions . . . . .	<b>5-94</b>
<b>Fit a Distribution Using the Distribution Fitter App</b> . . . . .	<b>5-97</b>
Step 1: Load Sample Data . . . . .	<b>5-97</b>
Step 2: Import Data . . . . .	<b>5-97</b>
Step 3: Create a New Fit . . . . .	<b>5-99</b>
Step 4: Create and Manage Additional Fits . . . . .	<b>5-105</b>
<b>Custom Distributions Using the Distribution Fitter App</b> . . . . .	<b>5-109</b>
Opening the Distribution Fitter App . . . . .	<b>5-109</b>
Defining Custom Distributions . . . . .	<b>5-110</b>
Importing Custom Distributions . . . . .	<b>5-110</b>
<b>Explore the Random Number Generation UI</b> . . . . .	<b>5-111</b>
<b>Compare Multiple Distribution Fits</b> . . . . .	<b>5-114</b>
<b>Fit Probability Distribution Objects to Grouped Data</b> . . . . .	<b>5-121</b>
<b>Multinomial Probability Distribution Objects</b> . . . . .	<b>5-125</b>
<b>Multinomial Probability Distribution Functions</b> . . . . .	<b>5-129</b>
<b>Generate Random Numbers Using Uniform Distribution     Inversion</b> . . . . .	<b>5-132</b>
<b>Represent Cauchy Distribution Using t Location-Scale</b> . . . . .	<b>5-136</b>
<b>Generate Cauchy Random Numbers Using Student's t</b> . . . . .	<b>5-140</b>
<b>Generate Correlated Data Using Rank Correlation</b> . . . . .	<b>5-142</b>

<b>Gaussian Mixture Models</b> . . . . .	<b>5-148</b>
<b>Create a Gaussian Mixture Model</b> . . . . .	<b>5-150</b>
<b>Fit a Gaussian Mixture Model to Data</b> . . . . .	<b>5-155</b>
<b>Simulate Data from a Gaussian Mixture Model</b> . . . . .	<b>5-160</b>
<b>Copulas: Generate Correlated Samples</b> . . . . .	<b>5-164</b>
Determining Dependence Between Simulation Inputs . . . . .	<b>5-164</b>
Constructing Dependent Bivariate Distributions . . . . .	<b>5-168</b>
Using Rank Correlation Coefficients . . . . .	<b>5-173</b>
Using Bivariate Copulas . . . . .	<b>5-176</b>
Higher Dimension Copulas . . . . .	<b>5-185</b>
Archimedean Copulas . . . . .	<b>5-187</b>
Simulating Dependent Multivariate Data Using Copulas . . . . .	<b>5-189</b>
Fitting Copulas to Data . . . . .	<b>5-194</b>

## Gaussian Processes

# 6

<b>Gaussian Process Regression Models</b> . . . . .	<b>6-2</b>
<b>Kernel (Covariance) Function Options</b> . . . . .	<b>6-6</b>
<b>Exact GPR Method</b> . . . . .	<b>6-11</b>
Parameter Estimation . . . . .	<b>6-11</b>
Prediction . . . . .	<b>6-13</b>
Computational Complexity of Exact Parameter Estimation and Prediction . . . . .	<b>6-16</b>
<b>Subset of Data Approximation for GPR Models</b> . . . . .	<b>6-18</b>
<b>Subset of Regressors Approximation for GPR Models</b> . . . . .	<b>6-20</b>
Approximating the Kernel Function . . . . .	<b>6-20</b>
Parameter Estimation . . . . .	<b>6-21</b>
Prediction . . . . .	<b>6-23</b>
Predictive Variance Problem . . . . .	<b>6-24</b>

<b>Fully Independent Conditional Approximation for GPR</b>	
<b>Models</b> . . . . .	<b>6-26</b>
Approximating the Kernel Function . . . . .	<b>6-26</b>
Parameter Estimation . . . . .	<b>6-27</b>
Prediction . . . . .	<b>6-28</b>
<b>Block Coordinate Descent Approximation for GPR</b>	
<b>Models</b> . . . . .	<b>6-30</b>
Fit GPR Models Using BCD Approximation . . . . .	<b>6-31</b>

## Random Number Generation

# 7

<b>Generating Random Data</b> . . . . .	<b>7-2</b>
<b>Random Number Generation Functions</b> . . . . .	<b>7-3</b>
<b>Common Pseudorandom Number Generation Methods</b> . . . . .	<b>7-5</b>
Direct Methods . . . . .	<b>7-5</b>
Inversion Methods . . . . .	<b>7-7</b>
Acceptance-Rejection Methods . . . . .	<b>7-10</b>
<b>Representing Sampling Distributions Using Markov Chain Samplers</b> . . . . .	<b>7-14</b>
Using the Metropolis-Hastings Algorithm . . . . .	<b>7-14</b>
Using Slice Sampling . . . . .	<b>7-15</b>
Using Hamiltonian Monte Carlo . . . . .	<b>7-15</b>
<b>Generating Quasi-Random Numbers</b> . . . . .	<b>7-18</b>
Quasi-Random Sequences . . . . .	<b>7-18</b>
Quasi-Random Point Sets . . . . .	<b>7-19</b>
Quasi-Random Streams . . . . .	<b>7-26</b>
<b>Generating Data Using Flexible Families of Distributions</b> . . . . .	<b>7-28</b>
Pearson and Johnson Systems . . . . .	<b>7-28</b>
Generating Data Using the Pearson System . . . . .	<b>7-29</b>
Generating Data Using the Johnson System . . . . .	<b>7-31</b>
<b>Bayesian Linear Regression Using Hamiltonian Monte Carlo</b> . . . . .	<b>7-35</b>

**8**

<b>Introduction to Hypothesis Tests</b> . . . . .	8-2
<b>Hypothesis Test Terminology</b> . . . . .	8-3
<b>Hypothesis Test Assumptions</b> . . . . .	8-5
<b>Hypothesis Testing</b> . . . . .	8-7
<b>Available Hypothesis Tests</b> . . . . .	8-14

**Analysis of Variance**

**9**

<b>Introduction to Analysis of Variance</b> . . . . .	9-2
<b>One-Way ANOVA</b> . . . . .	9-3
Introduction to One-Way ANOVA . . . . .	9-3
Prepare Data for One-Way ANOVA . . . . .	9-4
Perform One-Way ANOVA . . . . .	9-6
Mathematical Details . . . . .	9-12
<b>Two-Way ANOVA</b> . . . . .	9-15
Introduction to Two-Way ANOVA . . . . .	9-15
Prepare Data for Balanced Two-Way ANOVA . . . . .	9-17
Perform Two-Way ANOVA . . . . .	9-18
Mathematical Details . . . . .	9-22
<b>Multiple Comparisons</b> . . . . .	9-26
Introduction . . . . .	9-26
Multiple Comparisons Using One-Way ANOVA . . . . .	9-27
Multiple Comparisons for Three-Way ANOVA . . . . .	9-29
Multiple Comparison Procedures . . . . .	9-32
<b>N-Way ANOVA</b> . . . . .	9-36
Introduction to N-Way ANOVA . . . . .	9-36
Prepare Data for N-Way ANOVA . . . . .	9-38

Perform N-Way ANOVA . . . . .	9-39
<b>ANOVA with Random Effects</b> . . . . .	9-49
<b>Other ANOVA Models</b> . . . . .	9-58
<b>Analysis of Covariance</b> . . . . .	9-60
Introduction to Analysis of Covariance . . . . .	9-60
Analysis of Covariance Tool . . . . .	9-60
Confidence Bounds . . . . .	9-64
Multiple Comparisons . . . . .	9-66
<b>Nonparametric Methods</b> . . . . .	9-69
Introduction to Nonparametric Methods . . . . .	9-69
Kruskal-Wallis Test . . . . .	9-69
Friedman's Test . . . . .	9-70
<b>MANOVA</b> . . . . .	9-72
Introduction to MANOVA . . . . .	9-72
ANOVA with Multiple Responses . . . . .	9-72
<b>Model Specification for Repeated Measures Models</b> . . . . .	9-79
Wilkinson Notation . . . . .	9-79
<b>Compound Symmetry Assumption and Epsilon Corrections</b> . . . . .	9-81
<b>Mauchly's Test of Sphericity</b> . . . . .	9-83
<b>Multivariate Analysis of Variance for Repeated Measures</b> . . . . .	9-85

## Bayesian Optimization

# 10

<b>Bayesian Optimization Algorithm</b> . . . . .	10-2
Algorithm Outline . . . . .	10-2
Gaussian Process Regression for Fitting the Model . . . . .	10-3
Acquisition Function Types . . . . .	10-4
Acquisition Function Maximization . . . . .	10-7

<b>Parallel Bayesian Optimization</b> .....	<b>10-9</b>
Optimize in Parallel .....	10-9
Parallel Bayesian Algorithm .....	10-9
Settings for Best Parallel Performance .....	10-10
Differences in Parallel Bayesian Optimization Output .....	10-12
<b>Bayesian Optimization Plot Functions</b> .....	<b>10-14</b>
Built-In Plot Functions .....	10-14
Custom Plot Function Syntax .....	10-15
Create a Custom Plot Function .....	10-16
<b>Bayesian Optimization Output Functions</b> .....	<b>10-23</b>
What Is a Bayesian Optimization Output Function? .....	10-23
Built-In Output Functions .....	10-23
Custom Output Functions .....	10-23
Bayesian Optimization Output Function .....	10-24
<b>Bayesian Optimization Workflow</b> .....	<b>10-30</b>
What Is Bayesian Optimization? .....	10-30
Two Ways to Perform Bayesian Optimization .....	10-30
Bayesian Optimization Using a Fit Function .....	10-31
Bayesian Optimization Using bayesopt .....	10-32
Bayesian Optimization Characteristics .....	10-32
Parameters Available for Fit Functions .....	10-34
Hyperparameter Optimization Options for Fit Functions ..	10-36
<b>Variables for a Bayesian Optimization</b> .....	<b>10-39</b>
Syntax for Creating Optimization Variables .....	10-39
Variables for Optimization Examples .....	10-40
<b>Bayesian Optimization Objective Functions</b> .....	<b>10-42</b>
Objective Function Syntax .....	10-42
Objective Function Example .....	10-42
Objective Function Errors .....	10-43
<b>Constraints in Bayesian Optimization</b> .....	<b>10-44</b>
Bounds .....	10-44
Deterministic Constraints — XConstraintFcn .....	10-44
Conditional Constraints — ConditionalVariableFcn .....	10-45
Coupled Constraints .....	10-47
Bayesian Optimization with Coupled Constraints .....	10-48

<b>Optimize a Cross-Validated SVM Classifier Using Bayesian Optimization</b> .....	<b>10-53</b>
<b>Optimize an SVM Classifier Fit Using Bayesian Optimization</b> .....	<b>10-66</b>
<b>Optimize a Boosted Regression Ensemble</b> .....	<b>10-75</b>

## Parametric Regression Analysis

# 11

<b>Parametric Regression Analysis</b> .....	<b>11-3</b>
What Is Parametric Regression? .....	<b>11-3</b>
Choose a Regression Function .....	<b>11-3</b>
Update Legacy Code with New Fitting Methods .....	<b>11-4</b>
<b>What Are Linear Regression Models?</b> .....	<b>11-8</b>
<b>Linear Regression</b> .....	<b>11-11</b>
Prepare Data .....	<b>11-11</b>
Choose a Fitting Method .....	<b>11-13</b>
Choose a Model or Range of Models .....	<b>11-14</b>
Fit Model to Data .....	<b>11-19</b>
Examine Quality and Adjust the Fitted Model .....	<b>11-20</b>
Predict or Simulate Responses to New Data .....	<b>11-38</b>
Share Fitted Models .....	<b>11-41</b>
<b>Linear Regression Workflow</b> .....	<b>11-43</b>
<b>Regression Using Dataset Arrays</b> .....	<b>11-50</b>
<b>Regression Using Tables</b> .....	<b>11-53</b>
<b>Linear Regression with Interaction Effects</b> .....	<b>11-56</b>
<b>Interpret Linear Regression Results</b> .....	<b>11-64</b>
<b>Cook's Distance</b> .....	<b>11-71</b>
Purpose .....	<b>11-71</b>
Definition .....	<b>11-71</b>



How To . . . . .	11-72
Determine Outliers Using Cook's Distance . . . . .	11-72
<b>Coefficient Standard Errors and Confidence Intervals . . .</b>	<b>11-75</b>
Coefficient Covariance and Standard Errors . . . . .	11-75
Compute Coefficient Covariance and Standard Errors . . . .	11-75
Coefficient Confidence Intervals . . . . .	11-76
Compute Coefficient Confidence Intervals . . . . .	11-77
<b>Coefficient of Determination (R-Squared) . . . . .</b>	<b>11-79</b>
Purpose . . . . .	11-79
Definition . . . . .	11-79
How To . . . . .	11-79
Display Coefficient of Determination . . . . .	11-80
<b>Delete-1 Statistics . . . . .</b>	<b>11-82</b>
Delete-1 Change in Covariance (covratio) . . . . .	11-82
Determine Influential Observations Using CovRatio . . . . .	11-83
Delete-1 Scaled Difference in Coefficient Estimates (Dfbetas) . . . . .	11-85
Determine Observations Influential on Coefficients Using Dfbetas . . . . .	11-86
Delete-1 Scaled Change in Fitted Values (Dffits) . . . . .	11-86
Determine Observations Influential on Fitted Response Using Dffits . . . . .	11-87
Delete-1 Variance (S2_i) . . . . .	11-89
Compute and Examine Delete-1 Variance Values . . . . .	11-90
<b>Durbin-Watson Test . . . . .</b>	<b>11-93</b>
Purpose . . . . .	11-93
Definition . . . . .	11-93
How To . . . . .	11-93
Test for Autocorrelation Among Residuals . . . . .	11-93
<b>F-statistic and t-statistic . . . . .</b>	<b>11-95</b>
F-statistic . . . . .	11-95
Assess Fit of Model Using F-statistic . . . . .	11-96
t-statistic . . . . .	11-98
Assess Significance of Regression Coefficients Using t- statistic . . . . .	11-99
<b>Hat Matrix and Leverage . . . . .</b>	<b>11-102</b>
Hat Matrix . . . . .	11-102

Leverage . . . . .	11-103
Determine High Leverage Observations . . . . .	11-104
<b>Residuals . . . . .</b>	<b>11-106</b>
Purpose . . . . .	11-106
Definition . . . . .	11-106
How To . . . . .	11-107
Assess Model Assumptions Using Residuals . . . . .	11-107
<b>Summary of Output and Diagnostic Statistics . . . . .</b>	<b>11-115</b>
<b>Wilkinson Notation . . . . .</b>	<b>11-117</b>
Overview . . . . .	11-117
Formula Specification . . . . .	11-118
Linear Model Examples . . . . .	11-121
Linear Mixed-Effects Model Examples . . . . .	11-122
Generalized Linear Model Examples . . . . .	11-124
Generalized Linear Mixed-Effects Model Examples . . . . .	11-125
Repeated Measures Model Examples . . . . .	11-126
<b>Stepwise Regression . . . . .</b>	<b>11-127</b>
Stepwise Regression to Select Appropriate Models . . . . .	11-127
Compare large and small stepwise models . . . . .	11-127
<b>Robust Regression — Reduce Outlier Effects . . . . .</b>	<b>11-133</b>
What Is Robust Regression? . . . . .	11-133
Robust Regression versus Standard Least-Squares Fit . . . . .	11-133
<b>Introduction to Ridge Regression . . . . .</b>	<b>11-136</b>
<b>Ridge Regression . . . . .</b>	<b>11-137</b>
<b>Lasso and Elastic Net . . . . .</b>	<b>11-141</b>
What Are Lasso and Elastic Net? . . . . .	11-141
Lasso and Elastic Net Details . . . . .	11-141
References . . . . .	11-143
<b>Wide Data via Lasso and Parallel Computing . . . . .</b>	<b>11-144</b>
<b>Lasso Regularization . . . . .</b>	<b>11-149</b>
<b>Lasso and Elastic Net with Cross-Validation . . . . .</b>	<b>11-153</b>

<b>Partial Least Squares</b> .....	11-157
Introduction to Partial Least Squares .....	11-157
Partial Least Squares .....	11-158
<b>Linear Mixed-Effects Models</b> .....	11-162
<b>Prepare Data for Linear Mixed-Effects Models</b> .....	11-166
Tables and Dataset Arrays .....	11-166
Design Matrices .....	11-168
Relation of Matrix Form to Tables and Dataset Arrays ...	11-170
<b>Relationship Between Formula and Design Matrices</b> ....	11-172
Formula .....	11-172
Design Matrices for Fixed and Random Effects .....	11-174
Grouping Variables .....	11-176
<b>Estimating Parameters in Linear Mixed-Effects Models</b> .	11-179
Maximum Likelihood (ML) .....	11-179
Restricted Maximum Likelihood (REML) .....	11-181
<b>Linear Mixed-Effects Model Workflow</b> .....	11-183
<b>Fit Mixed-Effects Spline Regression</b> .....	11-197

## Generalized Linear Models

# 12

<b>Multinomial Models for Nominal Responses</b> .....	12-2
<b>Multinomial Models for Ordinal Responses</b> .....	12-5
<b>Hierarchical Multinomial Models</b> .....	12-9
<b>Generalized Linear Models</b> .....	12-12
What Are Generalized Linear Models? .....	12-12
Prepare Data .....	12-13
Choose Generalized Linear Model and Link Function .....	12-15
Choose Fitting Method and Model .....	12-18
Fit Model to Data .....	12-23
Examine Quality and Adjust the Fitted Model .....	12-24

Predict or Simulate Responses to New Data . . . . .	12-35
Share Fitted Models . . . . .	12-39
<b>Generalized Linear Model Workflow . . . . .</b>	<b>12-41</b>
<b>Lasso Regularization of Generalized Linear Models . . . . .</b>	<b>12-46</b>
What is Generalized Linear Model Lasso Regularization? . . . . .	12-46
Generalized Linear Model Lasso and Elastic Net . . . . .	12-46
References . . . . .	12-48
<b>Regularize Poisson Regression . . . . .</b>	<b>12-49</b>
<b>Regularize Logistic Regression . . . . .</b>	<b>12-53</b>
<b>Regularize Wide Data in Parallel . . . . .</b>	<b>12-61</b>
<b>Generalized Linear Mixed-Effects Models . . . . .</b>	<b>12-67</b>
What Are Generalized Linear Mixed-Effects Models? . . . . .	12-67
GLME Model Equations . . . . .	12-67
Prepare Data for Model Fitting . . . . .	12-69
Choose a Distribution Type for the Model . . . . .	12-69
Choose a Link Function for the Model . . . . .	12-70
Specify the Model Formula . . . . .	12-71
Display the Model . . . . .	12-74
Work with the Model . . . . .	12-76
<b>Fit a Generalized Linear Mixed-Effects Model . . . . .</b>	<b>12-79</b>

## Nonlinear Regression

# 13

<b>Nonlinear Regression . . . . .</b>	<b>13-2</b>
What Are Parametric Nonlinear Regression Models? . . . . .	13-2
Prepare Data . . . . .	13-3
Represent the Nonlinear Model . . . . .	13-4
Choose Initial Vector beta0 . . . . .	13-6
Fit Nonlinear Model to Data . . . . .	13-7
Examine Quality and Adjust the Fitted Nonlinear Model . . . . .	13-7
Predict or Simulate Responses Using a Nonlinear Model . . . . .	13-10

<b>Nonlinear Regression Workflow</b> .....	13-15
<b>Mixed-Effects Models</b> .....	13-22
Introduction to Mixed-Effects Models .....	13-22
Mixed-Effects Model Hierarchy .....	13-23
Specifying Mixed-Effects Models .....	13-24
Specifying Covariate Models .....	13-26
Choosing nlmeFit or nlmeFitsa .....	13-28
Using Output Functions with Mixed-Effects Models .....	13-31
<b>Examining Residuals for Model Verification</b> .....	13-37
<b>Mixed-Effects Models Using nlmeFit and nlmeFitsa</b> .....	13-43

## Survival Analysis

# 14

<b>What Is Survival Analysis?</b> .....	14-2
Introduction .....	14-2
Censoring .....	14-2
Data .....	14-3
Survivor Function .....	14-4
Hazard Function .....	14-6
<b>Kaplan-Meier Method</b> .....	14-10
<b>Hazard and Survivor Functions for Different Groups</b> ....	14-17
<b>Survivor Functions for Two Groups</b> .....	14-24
<b>Cox Proportional Hazards Model</b> .....	14-29
Introduction .....	14-29
Hazard Ratio .....	14-29
Extension of Cox Proportional Hazards Model .....	14-30
Partial Likelihood Function .....	14-31
Partial Likelihood Function for Tied Events .....	14-32
Frequency or Weights of Observations .....	14-33
<b>Cox Proportional Hazards Model for Censored Data</b> ....	14-35

<b>Cox Proportional Hazards Model with Time-Dependent Covariates</b> .....	<b>14-41</b>
--	--------------

## Multivariate Methods

# 15

<b>Introduction to Multivariate Methods</b> .....	<b>15-3</b>
<b>Multivariate Linear Regression</b> .....	<b>15-4</b>
Multivariate Linear Regression Model .....	15-4
Solving Multivariate Regression Problems .....	15-4
<b>Estimation of Multivariate Regression Models</b> .....	<b>15-7</b>
Least Squares Estimation .....	15-7
Maximum Likelihood Estimation .....	15-10
Missing Response Data .....	15-13
<b>Set Up Multivariate Regression Problems</b> .....	<b>15-16</b>
Response Matrix .....	15-16
Design Matrices .....	15-21
Common Multivariate Regression Problems .....	15-22
<b>Multivariate General Linear Model</b> .....	<b>15-29</b>
<b>Fixed Effects Panel Model with Concurrent Correlation</b> ..	<b>15-35</b>
<b>Longitudinal Analysis</b> .....	<b>15-43</b>
<b>Multidimensional Scaling</b> .....	<b>15-50</b>
<b>Nonclassical and Nonmetric Multidimensional Scaling</b> ...	<b>15-51</b>
Nonclassical Multidimensional Scaling .....	15-51
Nonmetric Multidimensional Scaling .....	15-53
<b>Classical Multidimensional Scaling</b> .....	<b>15-57</b>
<b>Example: Multidimensional Scaling</b> .....	<b>15-60</b>
<b>Procrustes Analysis</b> .....	<b>15-63</b>
Compare Landmark Data .....	15-63

Data Input .....	15-63
Preprocess Data for Accurate Results .....	15-64
<b>Compare Handwritten Shapes Using Procrustes</b>	
<b>Analysis</b> .....	15-65
<b>Feature Selection</b> .....	15-73
Introduction to Feature Selection .....	15-73
Sequential Feature Selection .....	15-73
<b>Select Subset of Features with Comparative Predictive</b>	
<b>Power</b> .....	15-75
<b>Feature Transformation</b> .....	15-78
<b>Nonnegative Matrix Factorization</b> .....	15-79
<b>Perform Nonnegative Matrix Factorization</b> .....	15-80
<b>Principal Component Analysis (PCA)</b> .....	15-83
<b>Analyze Quality of Life in U.S. Cities Using PCA</b> .....	15-85
<b>Factor Analysis</b> .....	15-97
<b>Analyze Stock Prices Using Factor Analysis</b> .....	15-98
<b>Robust Feature Selection Using NCA for Regression</b> ....	15-107
<b>Neighborhood Component Analysis (NCA) Feature</b>	
<b>Selection</b> .....	15-126
NCA Feature Selection for Classification .....	15-126
NCA Feature Selection for Regression .....	15-129
Impact of Standardization .....	15-130
Choosing the Regularization Parameter Value .....	15-131
<b>t-SNE</b> .....	15-132
What Is t-SNE? .....	15-132
t-SNE Algorithm .....	15-133
Barnes-Hut Variation of t-SNE .....	15-136
Characteristics of t-SNE .....	15-136

<b>t-SNE Output Function</b> .....	<b>15-140</b>
t-SNE Output Function Description .....	15-140
tsne optimValues Structure .....	15-141
t-SNE Custom Output Function .....	15-141
<b>Visualize High-Dimensional Data Using t-SNE</b> .....	<b>15-144</b>
<b>tsne Settings</b> .....	<b>15-149</b>
<b>Feature Extraction</b> .....	<b>15-163</b>
What Is Feature Extraction? .....	15-163
Sparse Filtering Algorithm .....	15-163
Reconstruction ICA Algorithm .....	15-166
<b>Feature Extraction Workflow</b> .....	<b>15-169</b>
<b>Extract Mixed Signals</b> .....	<b>15-205</b>

## Cluster Analysis

# 16

<b>Introduction to Cluster Analysis</b> .....	<b>16-2</b>
<b>Hierarchical Clustering</b> .....	<b>16-3</b>
Introduction to Hierarchical Clustering .....	16-3
Algorithm Description .....	16-3
Similarity Measures .....	16-4
Linkages .....	16-6
Dendrograms .....	16-8
Verify the Cluster Tree .....	16-9
Create Clusters .....	16-16
<b>k-Means Clustering</b> .....	<b>16-21</b>
Introduction to k-Means Clustering .....	16-21
Create Clusters and Determine Separation .....	16-21
Determine the Correct Number of Clusters .....	16-23
Avoid Local Minima .....	16-26
<b>Clustering Using Gaussian Mixture Models</b> .....	<b>16-28</b>



<b>Cluster Data from Mixture of Gaussian Distributions . . . .</b>	<b>16-37</b>
<b>Cluster Gaussian Mixture Data Using Soft Clustering . . . .</b>	<b>16-47</b>
<b>Tune Gaussian Mixture Models . . . . .</b>	<b>16-55</b>

## Parametric Classification

17

<b>Parametric Classification . . . . .</b>	<b>17-2</b>
<b>Performance Curves . . . . .</b>	<b>17-3</b>
Introduction to Performance Curves . . . . .	17-3
What are ROC Curves? . . . . .	17-3
Evaluate Classifier Performance Using perfcurve . . . . .	17-3

## Nonparametric Supervised Learning

18

<b>Supervised Learning Workflow and Algorithms . . . . .</b>	<b>18-2</b>
What is Supervised Learning? . . . . .	18-2
Steps in Supervised Learning . . . . .	18-3
Characteristics of Classification Algorithms . . . . .	18-7
<b>Visualize Decision Surfaces of Different Classifiers . . . . .</b>	<b>18-11</b>
<b>Classification Using Nearest Neighbors . . . . .</b>	<b>18-16</b>
Pairwise Distance Metrics . . . . .	18-16
k-Nearest Neighbor Search and Radius Search . . . . .	18-18
Classify Query Data . . . . .	18-24
Find Nearest Neighbors Using a Custom Distance Metric . . . . .	18-32
K-Nearest Neighbor Classification for Supervised Learning . . . . .	18-36
Construct KNN Classifier . . . . .	18-36
Examine Quality of KNN Classifier . . . . .	18-37
Predict Classification Using KNN Classifier . . . . .	18-38
Modify KNN Classifier . . . . .	18-39

<b>Framework for Ensemble Learning</b> .....	18-41
Prepare the Predictor Data .....	18-42
Prepare the Response Data .....	18-42
Choose an Applicable Ensemble Aggregation Method .....	18-43
Set the Number of Ensemble Members .....	18-46
Prepare the Weak Learners .....	18-47
Call fitcensemble or fitrensemble .....	18-49
<b>Ensemble Algorithms</b> .....	18-51
AdaBoostM1 .....	18-51
AdaBoostM2 .....	18-53
Bag .....	18-54
GentleBoost .....	18-58
LogitBoost .....	18-58
LPBoost .....	18-60
LSBoost .....	18-61
RobustBoost .....	18-61
RUSBoost .....	18-62
Subspace .....	18-63
TotalBoost .....	18-64
<b>Train Classification Ensemble</b> .....	18-67
<b>Train Regression Ensemble</b> .....	18-70
<b>Select Predictors for Random Forests</b> .....	18-73
<b>Test Ensemble Quality</b> .....	18-81
Test Ensemble Quality .....	18-81
<b>Ensemble Regularization</b> .....	18-88
Regularize a Regression Ensemble .....	18-89
<b>Classification with Imbalanced Data</b> .....	18-101
<b>Handle Imbalanced Data or Unequal Misclassification Costs</b> <b>in Classification Ensembles</b> .....	18-107
Train Ensemble With Unequal Classification Costs .....	18-108
<b>Classification with Many Categorical Levels</b> .....	18-115
<b>Surrogate Splits</b> .....	18-121

<b>LPBoost and TotalBoost for Small Ensembles</b> . . . . .	<b>18-124</b>
<b>Tune RobustBoost</b> . . . . .	<b>18-131</b>
<b>Random Subspace Classification</b> . . . . .	<b>18-135</b>
<b>Bootstrap Aggregation (Bagging) of Regression Trees</b> . . .	<b>18-141</b>
Regression of Insurance Risk Rating for Car Imports Using TreeBagger . . . . .	<b>18-141</b>
<b>Bootstrap Aggregation (Bagging) of Classification Trees</b>	<b>18-154</b>
Classifying Radar Returns for Ionosphere Data Using TreeBagger . . . . .	<b>18-154</b>
<b>Detect Outliers Using Quantile Regression</b> . . . . .	<b>18-170</b>
<b>Conditional Quantile Estimation Using Kernel Smoothing</b> . . . . .	<b>18-177</b>
<b>Tune Random Forest Using Quantile Error and Bayesian Optimization</b> . . . . .	<b>18-181</b>
<b>Support Vector Machines for Binary Classification</b> . . . . .	<b>18-186</b>
Understanding Support Vector Machines . . . . .	<b>18-186</b>
Using Support Vector Machines . . . . .	<b>18-192</b>
Train SVM Classifiers Using a Gaussian Kernel . . . . .	<b>18-195</b>
Train SVM Classifier Using Custom Kernel . . . . .	<b>18-199</b>
Optimize a Cross-Validated SVM Classifier Using Bayesian Optimization . . . . .	<b>18-205</b>
Plot Posterior Probability Regions for SVM Classification Models . . . . .	<b>18-218</b>
Analyze Images Using Linear Support Vector Machines . .	<b>18-221</b>
<b>Bibliography</b> . . . . .	<b>18-228</b>

## Decision Trees

<b>Decision Trees</b> . . . . .	<b>19-2</b>
Train Classification Tree . . . . .	<b>19-2</b>

Train Regression Tree . . . . .	19-3
<b>View Decision Tree . . . . .</b>	<b>19-5</b>
<b>Growing Decision Trees . . . . .</b>	<b>19-9</b>
<b>Prediction Using Classification and Regression Trees . . . .</b>	<b>19-11</b>
<b>Predict Out-of-Sample Responses of Subtrees . . . . .</b>	<b>19-13</b>
<b>Improving Classification Trees and Regression Trees . . . .</b>	<b>19-17</b>
Examining Resubstitution Error . . . . .	19-17
Cross Validation . . . . .	19-18
Choose Split Predictor Selection Technique . . . . .	19-19
Control Depth or “Leafiness” . . . . .	19-21
Pruning . . . . .	19-26
<b>Alternative: classregtree . . . . .</b>	<b>19-32</b>
Train Classification Trees Using classregtree . . . . .	19-32
Train Regression Trees Using classregtree . . . . .	19-37
<b>Splitting Categorical Predictors . . . . .</b>	<b>19-42</b>
Challenges in Splitting Multilevel Predictors . . . . .	19-42
Pull Left By Purity . . . . .	19-43
Principal Component-Based Partitioning . . . . .	19-43
One Versus All By Class . . . . .	19-43

## Discriminant Analysis

20

<b>Discriminant Analysis Classification . . . . .</b>	<b>20-2</b>
Create Discriminant Analysis Classifiers . . . . .	20-2
<b>Creating Discriminant Analysis Model . . . . .</b>	<b>20-5</b>
Weighted Observations . . . . .	20-5
<b>Prediction Using Discriminant Analysis Models . . . . .</b>	<b>20-8</b>
Posterior Probability . . . . .	20-8
Prior Probability . . . . .	20-9
Cost . . . . .	20-9

<b>Create and Visualize Discriminant Analysis Classifier . . .</b>	<b>20-11</b>
<b>Improving Discriminant Analysis Models . . . . .</b>	<b>20-20</b>
Deal with Singular Data . . . . .	20-20
Choose a Discriminant Type . . . . .	20-21
Examine the Resubstitution Error and Confusion Matrix . .	20-22
Cross Validation . . . . .	20-23
Change Costs and Priors . . . . .	20-24
<b>Regularize a Discriminant Analysis Classifier . . . . .</b>	<b>20-28</b>
<b>Examine the Gaussian Mixture Assumption . . . . .</b>	<b>20-36</b>
Bartlett Test of Equal Covariance Matrices for Linear Discriminant Analysis . . . . .	20-36
Q-Q Plot . . . . .	20-37
Mardia Kurtosis Test of Multivariate Normality . . . . .	20-41

## Naive Bayes

# 21

<b>Naive Bayes Classification . . . . .</b>	<b>21-2</b>
Supported Distributions . . . . .	21-2
<b>Plot Posterior Classification Probabilities . . . . .</b>	<b>21-7</b>

## Classification and Regression for High-Dimensional Data

# 22

<b>Alternatives for Lower-Dimensional Data . . . . .</b>	<b>22-2</b>
--	-------------

<b>Machine Learning in MATLAB</b> .....	23-2
What Is Machine Learning? .....	23-2
Selecting the Right Algorithm .....	23-3
Train Classification Models in Classification Learner App ..	23-7
Train Regression Models in Regression Learner App .....	23-8
Train Neural Networks for Deep Learning .....	23-9
<b>Train Classification Models in Classification Learner</b>	
<b>App</b> .....	23-11
What Is Supervised Machine Learning? .....	23-11
Automated Classifier Training .....	23-12
Manual Classifier Training .....	23-14
Parallel Classifier Training .....	23-15
Compare and Improve Classification Models .....	23-16
<b>Select Data and Validation for Classification Problem</b> ...	23-20
Select Data from the Workspace .....	23-20
Import Data from File .....	23-21
Example Data for Classification .....	23-21
Choose Validation Scheme .....	23-23
<b>Choose Classifier Options</b> .....	23-26
Choose a Classifier Type .....	23-26
Decision Trees .....	23-30
Discriminant Analysis .....	23-34
Logistic Regression .....	23-35
Support Vector Machines .....	23-35
Nearest Neighbor Classifiers .....	23-39
Ensemble Classifiers .....	23-42
<b>Feature Selection and Feature Transformation Using</b>	
<b>Classification Learner App</b> .....	23-47
Investigate Features in the Scatter Plot .....	23-47
Select Features to Include .....	23-49
Transform Features with PCA in Classification Learner ...	23-50
Investigate Features in the Parallel Coordinates Plot .....	23-50
<b>Assess Classifier Performance in Classification Learner</b> ...	23-54
Check Performance in the History List .....	23-54
Plot Classifier Results .....	23-54

Check Performance Per Class in the Confusion Matrix . . . .	23-56
Check the ROC Curve . . . . .	23-59
<b>Export Classification Model to Predict New Data . . . . .</b>	<b>23-61</b>
Export the Model to the Workspace to Make Predictions for New Data . . . . .	23-61
Make Predictions for New Data . . . . .	23-62
Generate MATLAB Code to Train the Model with New Data . . . . .	23-63
Generate C Code for Prediction . . . . .	23-64
<b>Train Decision Trees Using Classification Learner App . .</b>	<b>23-67</b>
<b>Train Discriminant Analysis Classifiers Using Classification Learner App . . . . .</b>	<b>23-80</b>
<b>Train Logistic Regression Classifiers Using Classification Learner App . . . . .</b>	<b>23-84</b>
<b>Train Support Vector Machines Using Classification Learner App . . . . .</b>	<b>23-88</b>
<b>Train Nearest Neighbor Classifiers Using Classification Learner App . . . . .</b>	<b>23-92</b>
<b>Train Ensemble Classifiers Using Classification Learner App . . . . .</b>	<b>23-96</b>

## Regression Learner

<b>Train Regression Models in Regression Learner App . . . .</b>	<b>24-2</b>
What Is Supervised Machine Learning? . . . . .	24-2
Automated Regression Model Training . . . . .	24-3
Manual Regression Model Training . . . . .	24-4
Parallel Regression Model Training . . . . .	24-6
Compare and Improve Regression Models . . . . .	24-6
<b>Select Data and Validation for Regression Problem . . . . .</b>	<b>24-10</b>
Select Data from Workspace . . . . .	24-10

Import Data from File . . . . .	24-11
Example Data for Regression . . . . .	24-11
Choose Validation Scheme . . . . .	24-12
<b>Choose Regression Model Options . . . . .</b>	<b>24-15</b>
Choose Regression Model Type . . . . .	24-15
Linear Regression Models . . . . .	24-18
Regression Trees . . . . .	24-21
Support Vector Machines . . . . .	24-23
Gaussian Process Regression Models . . . . .	24-26
Ensembles of Trees . . . . .	24-29
<b>Feature Selection and Feature Transformation Using</b>	
<b>Regression Learner App . . . . .</b>	<b>24-32</b>
Investigate Features in the Response Plot . . . . .	24-32
Select Features to Include . . . . .	24-33
Transform Features with PCA in Regression Learner . . . . .	24-34
<b>Assess Model Performance in Regression Learner App . . . . .</b>	<b>24-36</b>
Check Performance in History List . . . . .	24-36
View Model Statistics in Current Model Window . . . . .	24-37
Explore Data and Results in Response Plot . . . . .	24-38
Plot Predicted vs. Actual Response . . . . .	24-40
Evaluate Model Using Residuals Plot . . . . .	24-41
<b>Export Regression Model to Predict New Data . . . . .</b>	<b>24-44</b>
Export Model to Workspace . . . . .	24-44
Make Predictions for New Data . . . . .	24-45
Generate MATLAB Code to Train Model with New Data . . . . .	24-45
<b>Train Regression Trees Using Regression Learner App . . . . .</b>	<b>24-48</b>

## Support Vector Machines

25

<b>Understanding Support Vector Machine Regression . . . . .</b>	<b>25-2</b>
Mathematical Formulation of SVM Regression . . . . .	25-2
Solving the SVM Regression Optimization Problem . . . . .	25-6



<b>Markov Chains</b> .....	26-2
<b>Hidden Markov Models (HMM)</b> .....	26-4
Introduction to Hidden Markov Models (HMM) .....	26-4
Analyzing Hidden Markov Models .....	26-6

<b>Design of Experiments</b> .....	27-2
<b>Full Factorial Designs</b> .....	27-3
Multilevel Designs .....	27-3
Two-Level Designs .....	27-3
<b>Fractional Factorial Designs</b> .....	27-5
Introduction to Fractional Factorial Designs .....	27-5
Plackett-Burman Designs .....	27-5
General Fractional Designs .....	27-6
<b>Response Surface Designs</b> .....	27-9
Introduction to Response Surface Designs .....	27-9
Central Composite Designs .....	27-9
Box-Behnken Designs .....	27-12
<b>D-Optimal Designs</b> .....	27-15
Introduction to D-Optimal Designs .....	27-15
Generate D-Optimal Designs .....	27-16
Augment D-Optimal Designs .....	27-18
Specify Fixed Covariate Factors .....	27-19
Specify Categorical Factors .....	27-20
Specify Candidate Sets .....	27-21
<b>Improve an Engine Cooling Fan Using Design for Six Sigma Techniques</b> .....	27-24

## Statistical Process Control

28

<b>Introduction to Statistical Process Control</b> .....	28-2
<b>Control Charts</b> .....	28-3
<b>Capability Studies</b> .....	28-7

## Tall Arrays

29

<b>Tall Array Support, Usage Notes, and Limitations</b> .....	29-2
Descriptive Statistics and Visualization .....	29-2
Probability Distributions .....	29-5
Cluster Analysis .....	29-6
Regression .....	29-7
Classification .....	29-16
Dimensionality Reduction .....	29-28
<b>Logistic Regression with Tall Arrays</b> .....	29-30

## Parallel Statistics

30

<b>Quick Start Parallel Computing for Statistics and Machine Learning Toolbox</b> .....	30-2
What Is Parallel Statistics Functionality? .....	30-2
How To Compute in Parallel .....	30-4
<b>Use Parallel Processing for Regression TreeBagger Workflow</b> .....	30-6
<b>Concepts of Parallel Computing in Statistics and Machine Learning Toolbox</b> .....	30-8
Subtleties in Parallel Computing .....	30-8

Vocabulary for Parallel Computation . . . . .	30-8
<b>When to Run Statistical Functions in Parallel . . . . .</b>	<b>30-9</b>
Why Run in Parallel? . . . . .	30-9
Factors Affecting Speed . . . . .	30-9
Factors Affecting Results . . . . .	30-10
<b>Working with parfor . . . . .</b>	<b>30-11</b>
How Statistical Functions Use parfor . . . . .	30-11
Characteristics of parfor . . . . .	30-12
<b>Reproducibility in Parallel Statistical Computations . . . . .</b>	<b>30-14</b>
Issues and Considerations in Reproducing Parallel Computations . . . . .	30-14
Running Reproducible Parallel Computations . . . . .	30-14
Parallel Statistical Computation Using Random Numbers . . . . .	30-15
<b>Implement Jackknife Using Parallel Computing . . . . .</b>	<b>30-19</b>
<b>Implement Cross-Validation Using Parallel Computing . . . . .</b>	<b>30-21</b>
Simple Parallel Cross Validation . . . . .	30-21
Reproducible Parallel Cross Validation . . . . .	30-22
<b>Implement Bootstrap Using Parallel Computing . . . . .</b>	<b>30-24</b>
Bootstrap in Serial and Parallel . . . . .	30-24
Reproducible Parallel Bootstrap . . . . .	30-26

## Code Generation

<b>Code Generation Workflows . . . . .</b>	<b>31-2</b>
General Code Generation Support and Limitations . . . . .	31-2
Code Generation Workflow . . . . .	31-2
Code Generation Workflow for Prediction . . . . .	31-5
<b>C Code Generation and Classification Learner . . . . .</b>	<b>31-8</b>
<b>Predict Class Labels Using MATLAB Function Block . . . . .</b>	<b>31-18</b>
<b>Code Generation Support, Usage Notes, and Limitations . . . . .</b>	<b>31-24</b>

Specify Variable-Sized Arguments for Code Generation . . .	31-32
System Objects for Classification and Code Generation . . .	31-36
Code Generation Workflow for Exhaustive Searcher . . . . .	31-47
Predict Class Labels Using Stateflow . . . . .	31-52

## Functions — Alphabetical List

32

### Sample Data Sets

A

Sample Data Sets . . . . .	A-2
----------------------------	-----

### Distribution Reference

B

<b>Bernoulli Distribution</b> . . . . .	B-2
Overview . . . . .	B-2
Parameters . . . . .	B-2
Probability Mass Function . . . . .	B-2
Mean and Variance . . . . .	B-2
Relationship to Other Distributions . . . . .	B-3
<b>Beta Distribution</b> . . . . .	B-4
Overview . . . . .	B-4
Parameters . . . . .	B-4
Probability Density Function . . . . .	B-5
Cumulative Distribution Function . . . . .	B-7
Example . . . . .	B-7

<b>Binomial Distribution</b> .....	<b>B-9</b>
Overview .....	B-9
Parameters .....	B-9
Probability Density Function .....	B-9
Mean and Variance .....	B-10
Relationship to Other Distributions .....	B-10
Example .....	B-10
<b>Birnbaum-Saunders Distribution</b> .....	<b>B-13</b>
Definition .....	B-13
Background .....	B-13
Parameters .....	B-14
<b>Burr Type XII Distribution</b> .....	<b>B-15</b>
Definition .....	B-15
Background .....	B-15
Parameters .....	B-16
Fit a Burr Distribution and Draw the cdf .....	B-17
Compare Lognormal and Burr pdfs .....	B-20
Burr pdf for Various Parameters .....	B-21
Survival and Hazard Functions of Burr Distribution .....	B-23
Divergence of Parameter Estimates .....	B-25
<b>Chi-Square Distribution</b> .....	<b>B-28</b>
Overview .....	B-28
Parameters .....	B-28
Probability Density Function .....	B-28
Cumulative Distribution Function .....	B-28
Descriptive Statistics .....	B-29
Relationship to Other Distributions .....	B-29
Examples .....	B-29
<b>Copulas</b> .....	<b>B-32</b>
<b>Custom Distributions</b> .....	<b>B-33</b>
<b>Exponential Distribution</b> .....	<b>B-34</b>
Definition .....	B-34
Background .....	B-34
Parameters .....	B-34
Examples .....	B-35

<b>Extreme Value Distribution</b> .....	B-38
Definition .....	B-38
Background .....	B-38
Parameters .....	B-40
Examples .....	B-41
<b>F Distribution</b> .....	B-44
Definition .....	B-44
Background .....	B-44
Examples .....	B-45
<b>Gamma Distribution</b> .....	B-47
Definition .....	B-47
Background .....	B-47
Parameters .....	B-47
Examples .....	B-49
<b>Gaussian Distribution</b> .....	B-51
<b>Gaussian Mixture Distributions</b> .....	B-52
<b>Generalized Extreme Value Distribution</b> .....	B-53
Definition .....	B-53
Background .....	B-53
Parameters .....	B-54
Examples .....	B-55
<b>Generalized Pareto Distribution</b> .....	B-58
Definition .....	B-58
Background .....	B-58
Parameters .....	B-59
Examples .....	B-61
<b>Geometric Distribution</b> .....	B-64
Overview .....	B-64
Parameters .....	B-64
Probability Distribution Function .....	B-64
Cumulative Distribution Function .....	B-67
Mean and Variance .....	B-69
Example .....	B-70
<b>Half-Normal Distribution</b> .....	B-73
Overview .....	B-73

Parameters . . . . .	B-73
Probability Density Function . . . . .	B-74
Cumulative Distribution Function . . . . .	B-75
Descriptive Statistics . . . . .	B-77
Relationship to Other Distributions . . . . .	B-78
<b>Hypergeometric Distribution . . . . .</b>	<b>B-79</b>
Definition . . . . .	B-79
Background . . . . .	B-79
Examples . . . . .	B-80
<b>Inverse Gaussian Distribution . . . . .</b>	<b>B-82</b>
Definition . . . . .	B-82
Background . . . . .	B-82
Parameters . . . . .	B-82
<b>Inverse Wishart Distribution . . . . .</b>	<b>B-83</b>
Definition . . . . .	B-83
Background . . . . .	B-83
Example . . . . .	B-83
<b>Johnson System . . . . .</b>	<b>B-85</b>
<b>Kernel Distribution . . . . .</b>	<b>B-86</b>
Overview . . . . .	B-86
Kernel Density Estimator . . . . .	B-86
Kernel Smoothing Function . . . . .	B-86
Bandwidth . . . . .	B-92
<b>Logistic Distribution . . . . .</b>	<b>B-96</b>
Overview . . . . .	B-96
Parameters . . . . .	B-96
Probability Density Function . . . . .	B-96
Relationship to Other Distributions . . . . .	B-96
<b>Loglogistic Distribution . . . . .</b>	<b>B-98</b>
Overview . . . . .	B-98
Parameters . . . . .	B-98
Probability Density Function . . . . .	B-98
Relationship to Other Distributions . . . . .	B-99
<b>Lognormal Distribution . . . . .</b>	<b>B-100</b>
Overview . . . . .	B-100

Parameters . . . . .	B-100
Probability Density Function . . . . .	B-100
Descriptive Statistics . . . . .	B-101
Relationship to Other Distributions . . . . .	B-101
Examples . . . . .	B-101
<b>Multinomial Distribution . . . . .</b>	<b>B-103</b>
Overview . . . . .	B-103
Parameter . . . . .	B-103
Probability Density Function . . . . .	B-103
Descriptive Statistics . . . . .	B-104
Relationship to Other Distributions . . . . .	B-104
<b>Multivariate Gaussian Distribution . . . . .</b>	<b>B-105</b>
<b>Multivariate Normal Distribution . . . . .</b>	<b>B-106</b>
Definition . . . . .	B-106
Background . . . . .	B-106
Examples . . . . .	B-107
<b>Multivariate t Distribution . . . . .</b>	<b>B-112</b>
Definition . . . . .	B-112
Background . . . . .	B-112
Example . . . . .	B-113
<b>Nakagami Distribution . . . . .</b>	<b>B-118</b>
Definition . . . . .	B-118
Background . . . . .	B-118
Parameters . . . . .	B-118
<b>Negative Binomial Distribution . . . . .</b>	<b>B-120</b>
Definition . . . . .	B-120
Background . . . . .	B-120
Parameters . . . . .	B-121
Example . . . . .	B-123
<b>Noncentral Chi-Square Distribution . . . . .</b>	<b>B-125</b>
Definition . . . . .	B-125
Background . . . . .	B-125
Examples . . . . .	B-126
<b>Noncentral F Distribution . . . . .</b>	<b>B-128</b>
Definition . . . . .	B-128



Background .....	B-128
Examples .....	B-129
<b>Noncentral t Distribution</b> .....	<b>B-131</b>
Definition .....	B-131
Background .....	B-131
Examples .....	B-132
<b>Normal Distribution</b> .....	<b>B-134</b>
Definition .....	B-134
Background .....	B-134
Parameters .....	B-134
Examples .....	B-136
<b>Pareto Distribution</b> .....	<b>B-139</b>
<b>Pearson System</b> .....	<b>B-140</b>
<b>Piecewise Linear Distribution</b> .....	<b>B-141</b>
Overview .....	B-141
Parameters .....	B-141
Cumulative Distribution Function .....	B-141
Relationship to Other Distributions .....	B-141
<b>Poisson Distribution</b> .....	<b>B-143</b>
Definition .....	B-143
Background .....	B-143
Parameters .....	B-143
Examples .....	B-144
<b>Rayleigh Distribution</b> .....	<b>B-146</b>
Definition .....	B-146
Background .....	B-146
Parameters .....	B-147
Examples .....	B-147
<b>Rician Distribution</b> .....	<b>B-149</b>
Definition .....	B-149
Background .....	B-149
Parameters .....	B-149
<b>Stable Distribution</b> .....	<b>B-151</b>
Overview .....	B-151

Parameters . . . . .	B-152
Probability Density Function . . . . .	B-152
Cumulative Distribution Function . . . . .	B-156
Descriptive Statistics . . . . .	B-160
Relationship to Other Distributions . . . . .	B-161
<b>Student's t Distribution . . . . .</b>	<b>B-166</b>
Overview . . . . .	B-166
Parameters . . . . .	B-166
Probability Density Function . . . . .	B-166
Cumulative Distribution Function . . . . .	B-169
Mean and Variance . . . . .	B-171
Example . . . . .	B-172
<b>t Location-Scale Distribution . . . . .</b>	<b>B-175</b>
Overview . . . . .	B-175
Parameters . . . . .	B-175
Probability Density Function . . . . .	B-175
Cumulative Distribution Function . . . . .	B-176
Descriptive Statistics . . . . .	B-176
Relationship to Other Distributions . . . . .	B-177
<b>Triangular Distribution . . . . .</b>	<b>B-178</b>
Overview . . . . .	B-178
Parameters . . . . .	B-178
Probability Density Function . . . . .	B-179
Cumulative Distribution Function . . . . .	B-180
Descriptive Statistics . . . . .	B-182
<b>Uniform Distribution (Continuous) . . . . .</b>	<b>B-184</b>
Overview . . . . .	B-184
Parameters . . . . .	B-184
Probability Density Function . . . . .	B-184
Cumulative Distribution Function . . . . .	B-186
Descriptive Statistics . . . . .	B-188
Relationship to Other Distributions . . . . .	B-189
<b>Uniform Distribution (Discrete) . . . . .</b>	<b>B-190</b>
Definition . . . . .	B-190
Background . . . . .	B-190
Examples . . . . .	B-190

<b>Weibull Distribution</b> .....	<b>B-193</b>
Definition .....	<b>B-193</b>
Background .....	<b>B-193</b>
Plot the Weibull Distribution Hazard Function .....	<b>B-193</b>
Estimate Parameters of the Weibull Distribution .....	<b>B-194</b>
Estimate Parameters of a Three-Parameter Weibull Distribution .....	<b>B-195</b>
<b>Wishart Distribution</b> .....	<b>B-198</b>
Overview .....	<b>B-198</b>
Parameters .....	<b>B-198</b>
Probability Density Function .....	<b>B-198</b>
Example .....	<b>B-199</b>

## Bibliography

### C

<b>Bibliography</b> .....	<b>C-2</b>
---------------------------	------------



# Getting Started

---

- “Statistics and Machine Learning Toolbox Product Description” on page 1-2
- “Supported Data Types” on page 1-3
- “Statistics and Machine Learning Toolbox Functions with gpuArray Arguments” on page 1-5

# Statistics and Machine Learning Toolbox Product Description

## Analyze and model data using statistics and machine learning

Statistics and Machine Learning Toolbox provides functions and apps to describe, analyze, and model data. You can use descriptive statistics and plots for exploratory data analysis, fit probability distributions to data, generate random numbers for Monte Carlo simulations, and perform hypothesis tests. Regression and classification algorithms let you draw inferences from data and build predictive models.

For multidimensional data analysis, Statistics and Machine Learning Toolbox provides feature selection, stepwise regression, principal component analysis (PCA), regularization, and other dimensionality reduction methods that let you identify variables or features that impact your model.

The toolbox provides supervised and unsupervised machine learning algorithms, including support vector machines (SVMs), boosted and bagged decision trees,  $k$ -nearest neighbor,  $k$ -means,  $k$ -medoids, hierarchical clustering, Gaussian mixture models, and hidden Markov models. Many of the statistics and machine learning algorithms can be used for computations on data sets that are too big to be stored in memory.

## Key Features

- Regression techniques, including linear, generalized linear, nonlinear, robust, regularized, ANOVA, repeated measures, and mixed-effects models
- Big data algorithms for dimension reduction, descriptive statistics,  $k$ -means clustering, linear regression, logistic regression, and discriminant analysis
- Univariate and multivariate probability distributions, random and quasi-random number generators, and Markov chain samplers
- Hypothesis tests for distributions, dispersion, and location, and design of experiments (DOE) techniques for optimal, factorial, and response surface designs
- Classification Learner app and algorithms for supervised machine learning, including support vector machines (SVMs), boosted and bagged decision trees,  $k$ -nearest neighbor, Naïve Bayes, discriminant analysis, and Gaussian process regression
- Unsupervised machine learning algorithms, including  $k$ -means,  $k$ -medoids, hierarchical clustering, Gaussian mixtures, and hidden Markov models
- Bayesian optimization for tuning machine learning algorithms by searching for optimal hyperparameters

## Supported Data Types

Statistics and Machine Learning Toolbox supports the following data types for input arguments:

- Numeric scalars, vectors, matrices, or arrays having single- or double-precision entries. These data forms have data type `single` or `double`. Examples include response variables, predictor variables, and numeric values.
- Cell arrays of character vectors; character, logical, or categorical arrays; or numeric vectors for categorical variables representing grouping data. These data forms have data types `cellstr`, `char`, `logical`, `categorical`, and `single` or `double`, respectively. An example is an array of class labels in machine learning.
- You can also use nominal or ordinal arrays for categorical data. However, the nominal and ordinal data types might be removed in a future release. To work with nominal or ordinal categorical data, use the `categorical` data type instead.
- You can use signed or unsigned integers, e.g., `int8` or `uint8`. However:
  - Estimation functions might not support signed or unsigned integer data types for nongrouping data.
  - If you recast a `single` or `double` numeric vector containing NaN values to a signed or unsigned integer, then the software converts the NaN elements to 0.
- Some functions support tabular arrays for heterogeneous data (for details, see “Tables” (MATLAB)). The `table` data type contains variables of any of the data types previously listed. An example is mixed categorical and numerical predictor data for regression analysis.
  - For some functions, you can also use `dataset` arrays for heterogeneous data. However, the `dataset` data type might be removed in a future release. To work with heterogeneous data, use the `table` data type if the estimation function supports it.
  - Functions that do not support the `table` data type support sample data of type `single` or `double`, e.g., matrices.
- Some functions accept `gpuArray` input arguments so that they execute on the GPU. For the full list of Statistics and Machine Learning Toolbox functions that accept `gpuArrays`, see “Statistics and Machine Learning Toolbox Functions with `gpuArray` Arguments” on page 1-5.

- Some functions accept `tall` array input arguments to work with large data sets. For the full list of Statistics and Machine Learning Toolbox functions that accept tall arrays, see “Tall Array Support, Usage Notes, and Limitations” on page 29-2.

Statistics and Machine Learning Toolbox does not support the following data types:

- Complex numbers.
- Custom numeric data types, e.g., a variable that is double precision and an object.
- Signed or unsigned numeric integers for nongrouping data, e.g., `uint8` and `int16`.
- Sparse matrices, i.e., matrix `A` such that `issparse(A)` returns 1. To use data that is of data type `sparse`, recast the data to a matrix using `full`.

---

**Note** If you specify data of an unsupported type, then the software might return an error or unexpected results.

---



## Statistics and Machine Learning Toolbox Functions with gpuArray Arguments

The following functions are enhanced to accept gpuArray input arguments so that they execute on the GPU:

betacdf	evlike	geocdf	mad	pdist	tpdf
betainv	evrnd	geoinvgeome	margin	pdist2	tstat
betapdf	evstat	an	moment	poisscdf	ttest
betastat	expcdf	geopdf	mvncdf	poissinv	ttest2
binofit	expfit	geornd	mvnpdf	poisspdf	trimmean
binoinv	expinv	geostat	mvnrnd	poisstat,	unidcdf
binopdf	explike	harmmean	nanmax	prctile	unidinv
binornd	exppdf	iqr	nanmean	predict	unidpdf
binostat	exprnd	kmeans	nanmedian	quantile	unidrnd
chi2cdf	expstat	knnsearch	nanmin	range	unidstat
chi2gof	fcdf	kurtosis	nanstd	raylcdf	unifinv
chi2inv	finv	logncdf	nansum	raylfit	unifrnd
chi2stat	fitcsvm	lognfit	nanvar	raylinv	unifstat
cholcov	fstat	logninv	normcdf	raylpdf	vartest
corrcov	gamcdf	lognlike	normfit	raylrnd	vartest2
evcdf	gaminv	lognpdf	norminv	raylstat	zscore
evpdf	gamlike	lognrnd	normlike	resume	ztest
evinv	gampdf	lognstat	normpdf	skewness	
	gamstat	loss	normrnd	tcdf	
			normstat	tin	



# Organizing Data

---

- “Other MATLAB Functions Supporting Nominal and Ordinal Arrays” on page 2-3
- “Create Nominal and Ordinal Arrays” on page 2-4
- “Change Category Labels” on page 2-10
- “Reorder Category Levels” on page 2-12
- “Categorize Numeric Data” on page 2-18
- “Merge Category Levels” on page 2-22
- “Add and Drop Category Levels” on page 2-25
- “Plot Data Grouped by Category” on page 2-29
- “Test Differences Between Category Means” on page 2-34
- “Summary Statistics Grouped by Category” on page 2-44
- “Sort Ordinal Arrays” on page 2-46
- “Categorical Arrays” on page 2-48
- “Advantages of Using Categorical Arrays” on page 2-50
- “Index and Search Using Categorical Arrays” on page 2-53
- “Grouping Variables” on page 2-59
- “Dummy Indicator Variables” on page 2-62
- “Regression with Categorical Covariates” on page 2-65
- “Create a Dataset Array from Workspace Variables” on page 2-70
- “Create a Dataset Array from a File” on page 2-77
- “Add and Delete Observations” on page 2-85
- “Add and Delete Variables” on page 2-89
- “Access Data in Dataset Array Variables” on page 2-93
- “Select Subsets of Observations” on page 2-99
- “Sort Observations in Dataset Arrays” on page 2-103
- “Merge Dataset Arrays” on page 2-107
- “Stack or Unstack Dataset Arrays” on page 2-112

- “Calculations on Dataset Arrays” on page 2-117
- “Export Dataset Arrays” on page 2-121
- “Clean Messy and Missing Data” on page 2-123
- “Dataset Arrays in the Variables Editor” on page 2-128
- “Dataset Arrays” on page 2-142
- “Index and Search Dataset Arrays” on page 2-145

## Other MATLAB Functions Supporting Nominal and Ordinal Arrays

Notable functions that operate on nominal and ordinal arrays are listed in `nominal` and `ordinal`. In addition to these, many other functions in MATLAB® operate on nominal and ordinal arrays in much the same way that they operate on other arrays. A few functions might exhibit special behavior when operating on categorical arrays:

- If multiple input arguments are categorical arrays, the function often requires that they have the same set of categories, including order if ordinal.
- Relational functions, such as `max` and `gt`, require that the input arrays be ordinal.

The following table lists MATLAB functions that operate on nominal and ordinal arrays in addition to other arrays.

<code>size</code>	<code>isequal</code>	<code>intersect</code>	<code>histogram</code>	<code>double</code>
<code>length</code>	<code>isequaln</code>	<code>ismember</code>	<code>pietimes</code>	<code>single</code>
<code>ndims</code>		<code>setdiff</code>		<code>int8</code>
<code>numel</code>	<code>eq</code>	<code>setxor</code>	<code>sort</code>	<code>int16</code>
	<code>ne</code>	<code>unique</code>	<code>sortrows</code>	<code>int32</code>
<code>isrow</code>	<code>lt</code>	<code>union</code>	<code>issorted</code>	<code>int64</code>
<code>iscolumn</code>	<code>le</code>			<code>uint8</code>
	<code>ge</code>		<code>permute</code>	<code>uint16</code>
<code>cat</code>	<code>gt</code>		<code>reshape</code>	<code>uint32</code>
<code>horzcat</code>			<code>transpose</code>	<code>uint64</code>
<code>vertcat</code>	<code>min</code>		<code>ctranspose</code>	<code>char</code>
	<code>max</code>			<code>cellstr</code>
	<code>median</code>			
	<code>mode</code>			

### See Also

`nominal` | `ordinal`

# Create Nominal and Ordinal Arrays

In this section...
“Create Nominal Arrays” on page 2-4
“Create Ordinal Arrays” on page 2-6

## Create Nominal Arrays

This example shows how to create nominal arrays using `nominal`.

### Load sample data.

The variable `species` is a 150-by-1 cell array of character vectors containing the species name for each observation. The unique species types are `setosa`, `versicolor`, and `virginica`.

```
load fisheriris
unique(species)
```

```
ans =
```

```
3x1 cell array

    {'setosa'    }
    {'versicolor'}
    {'virginica' }
```

### Create a nominal array.

Convert `species` to a nominal array using the categories occurring in the data.

```
speciesNom = nominal(species);
class(speciesNom)
```

```
ans =
```

```
'nominal'
```

**Explore category levels.**

The nominal array, `speciesNom`, has three levels corresponding to the three unique species. The levels of a nominal array are the set of possible values that its elements can take.

```
getlevels(speciesNom)

ans =

    1x3 nominal array

    setosa    versicolor    virginica
```

A nominal array can have more levels than actually appear in the data. For example, a nominal array named `AllSizes` might have levels `small`, `medium`, and `large`, but you might only have observations that are `medium` and `large` in your data. To see which levels of a nominal array are actually present in the data, use `unique`, for instance, `unique(AllSizes)`.

**Explore category labels.**

Each level has a label. By default, nominal labels the category levels with the values occurring in the data. For `speciesNom`, these labels are the species types.

```
getlabels(speciesNom)

ans =

    1x3 cell array

    {'setosa'}    {'versicolor'}    {'virginica'}
```

**Specify your own category labels.**

You can specify your own labels for each category level. You can specify labels when you create the nominal array.

```
speciesNom2 = nominal(species, {'seto', 'vers', 'virg'});
getlabels(speciesNom2)
```

```
ans =  
  
    1x3 cell array  
  
    {'seto'}    {'vers'}    {'virg'}
```

You can also change category labels on an existing nominal array using `setlabels`

### Verify new category labels.

Verify that the new labels correspond to the original labels in `speciesNom`.

```
isequal(speciesNom=='setosa',speciesNom2=='seto')
```

```
ans =  
  
    logical  
  
     1
```

The logical value 1 indicates that the two labels, 'setosa' and 'seto', correspond to the same observations.

## Create Ordinal Arrays

This example shows how to create ordinal arrays using `ordinal`.

### Load sample data.

```
AllSizes = {'medium','large','small','small','medium',...  
            'large','medium','small'};
```

The created variable, `AllSizes`, is a cell array of character vectors containing size measurements on eight objects.

### Create an ordinal array.

Create an ordinal array with category levels and labels corresponding to the values in the cell array (the default levels and labels).



```
sizeOrd = ordinal(AllSizes);
getlevels(sizeOrd)

ans =

    1x3 ordinal array

    large      medium      small
```

### Explore category labels.

By default, `ordinal` uses the original character vectors as category labels. The default order of the categories is ascending alphabetical order.

```
getlabels(sizeOrd)

ans =

    1x3 cell array

    {'large'}    {'medium'}    {'small'}
```

### Add additional categories.

Suppose that you want to include additional levels for the ordinal array, `xsmall` and `xlarge`, even though they do not occur in the original data. To specify additional levels, use the third input argument to `ordinal`.

```
sizeOrd2 = ordinal(AllSizes, {}, ...
                  {'xsmall', 'small', 'medium', 'large', 'xlarge'});
getlevels(sizeOrd2)

ans =

    1x5 ordinal array

    xsmall      small      medium      large      xlarge
```

### Explore category labels.

To see which levels are actually present in the data, use `unique`.

```
unique(sizeOrd2)

ans =

    1x3 ordinal array

    small    medium    large
```

### Specify the category order.

Convert `AllSizes` to an ordinal array with categories `small < medium < large`. Generally, an ordinal array is distinct from a nominal array because there is a natural ordering for levels of an ordinal array. You can use the third input argument to `ordinal` to specify the ascending order of the levels. Here, the order of the levels is smallest to largest.

```
sizeOrd = ordinal(AllSizes, {}, {'small', 'medium', 'large'});
getlevels(sizeOrd)

ans =

    1x3 ordinal array

    small    medium    large
```

The second input argument for `ordinal` is a list of labels for the category levels. When you use braces `{}` for the level labels, `ordinal` uses the labels specified in the third input argument (the labels come from the levels present in the data if only one input argument is used).

### Compare elements.

Verify that the first object (with size `medium`) is smaller than the second object (with size `large`).

```
sizeOrd(1) < sizeOrd(2)
```

```
ans =  
  logical  
  1
```

The logical value 1 indicates that the inequality holds.

## See Also

`getlabels` | `getlevels` | `nominal` | `ordinal`

## Related Examples

- “Change Category Labels” on page 2-10
- “Reorder Category Levels” on page 2-12
- “Merge Category Levels” on page 2-22
- “Index and Search Using Categorical Arrays” on page 2-53

## More About

- “Categorical Arrays” on page 2-48
- “Advantages of Using Categorical Arrays” on page 2-50

### Change Category Labels

This example shows how to change the labels for category levels in categorical arrays using `setlabels`. You also have the option to specify labels when creating a categorical array.

#### Load sample data.

The variable `Cylinders` contains the number of cylinders in 100 sample cars.

```
load carsmall
unique(Cylinders)
```

```
ans =
```

```
4
6
8
```

The sample has 4-, 6-, and 8-cylinder cars.

#### Create an ordinal array.

Convert `Cylinders` to a nominal array with the default category labels (taken from the values in the data).

```
cyl = ordinal(Cylinders);
getlabels(cyl)
```

```
ans =
```

```
1x3 cell array
    {'4'}    {'6'}    {'8'}
```

`ordinal` created labels using the integer values in `Cylinders`, but you should provide labels for numeric data.

### Change category labels.

Relabel the categories in `cyl` to Four, Six, and Eight.

```
cyl = setlabels(cyl , {'Four', 'Six', 'Eight'});  
getlabels(cyl)
```

```
ans =
```

```
    1x3 cell array  
  
    {'Four'}    {'Six'}    {'Eight'}
```

Alternatively, you can specify category labels when you create a nominal or ordinal array using the second input argument, for example by specifying `ordinal(Cylinders, {'Four', 'Six', 'Eight'})`.

## See Also

`getlabels` | `nominal` | `ordinal` | `setlabels`

## Related Examples

- “Reorder Category Levels” on page 2-12
- “Add and Drop Category Levels” on page 2-25
- “Index and Search Using Categorical Arrays” on page 2-53

## More About

- “Categorical Arrays” on page 2-48
- “Advantages of Using Categorical Arrays” on page 2-50

# Reorder Category Levels

<b>In this section...</b>
“Reorder Category Levels in Ordinal Arrays” on page 2-12
“Reorder Category Levels in Nominal Arrays” on page 2-14

## Reorder Category Levels in Ordinal Arrays

This example shows how to reorder the category levels in an ordinal array using `reorderlevels`.

### Load sample data.

```
AllSizes = {'medium', 'large', 'small', 'small', 'medium', ...  
           'large', 'medium', 'small'};
```

The created variable, `AllSizes`, is a cell array of character vectors containing size measurements on eight objects.

### Create an ordinal array.

Convert `AllSizes` to an ordinal array without specifying the order of the category levels.

```
size = ordinal (AllSizes);  
getlevels (size)
```

```
ans =
```

```
    1x3 ordinal array  
  
    large    medium    small
```

By default, the categories are ordered by their labels in ascending alphabetical order, `large < medium < small`.

### Compare elements.

Check whether or not the first object (which has size `medium`) is smaller than the second object (which has size `large`).

```
size(1) < size(2)
```

```
ans =
```

```
logical
```

```
0
```

The logical value 0 indicates that the medium object is not smaller than the large object.

### Reorder category levels.

Reorder the category levels so that `small < medium < large`.

```
size = reorderlevels(size, {'small', 'medium', 'large'});  
getlevels(size)
```

```
ans =
```

```
1x3 ordinal array
```

```
small    medium    large
```

### Compare elements.

Verify that the first object is now smaller than the second object.

```
size(1) < size(2)
```

```
ans =
```

```
logical
```

```
1
```

The logical value 1 indicates that the expected inequality now holds.

### Reorder Category Levels in Nominal Arrays

This example shows how to reorder the category levels in nominal arrays using `reorderlevels`. By definition, nominal array categories have no natural ordering. However, you might want to change the order of levels for display or analysis purposes. For example, when fitting a regression model with categorical covariates, `fitlm` uses the first level of a nominal independent variable as the reference category.

#### Load sample data.

The dataset array, `hospital`, contains variables measured on 100 sample patients. The variable `Weight` contains the weight of each patient. The variable `Sex` is a nominal variable containing the gender, `Male` or `Female`, for each patient.

```
load hospital
getlevels(hospital.Sex)
```

```
ans =
    1x2 nominal array
    Female      Male
```

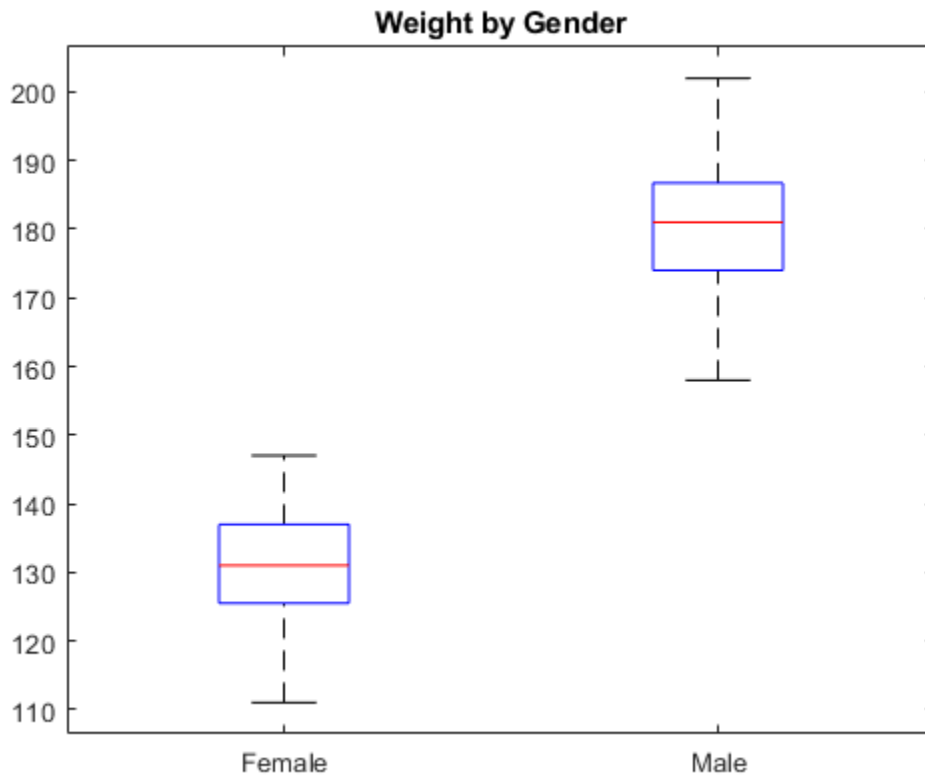
By default, the order of the nominal categories is in ascending alphabetical order of the labels.

#### Plot data grouped by category level.

Draw box plots of weight, grouped by gender.

```
figure
boxplot(hospital.Weight,hospital.Sex)
title('Weight by Gender')
```





The box plots appear in the same alphabetical order returned by `getlevels`.

**Change the category order.**

Change the order of the category levels.

```
hospital.Sex = reorderlevels(hospital.Sex, {'Male', 'Female'});  
getlevels(hospital.Sex)
```

```
ans =
```

```
1x2 nominal array
```

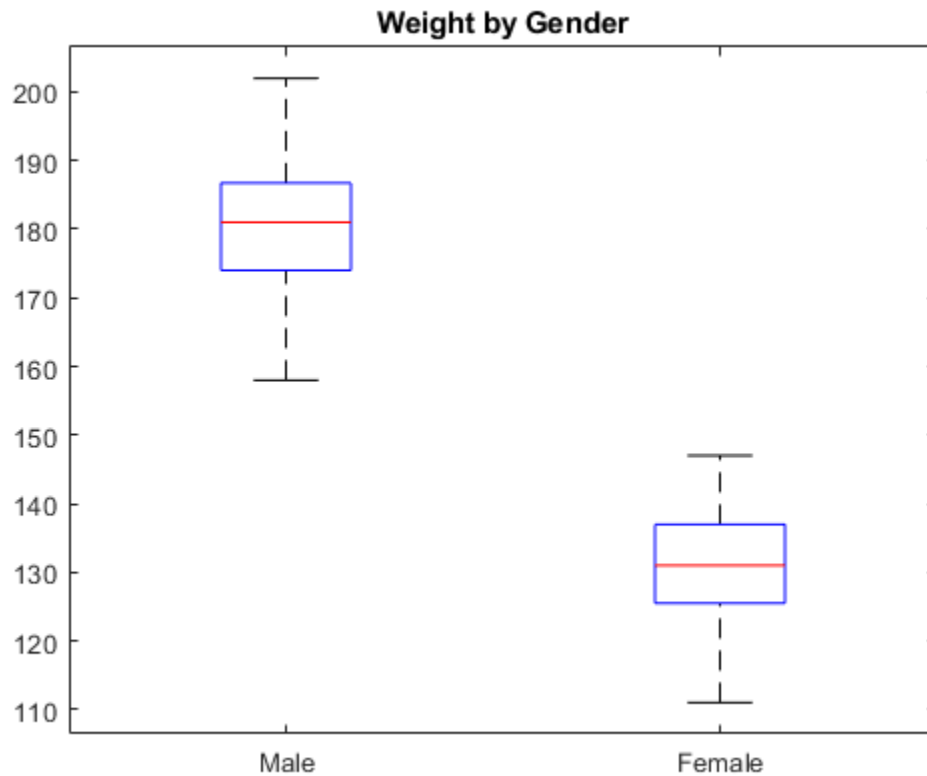
Male      Female

The levels are in the newly specified order.

**Plot data in new order.**

Draw box plots of weight by gender.

```
figure  
boxplot(hospital.Weight,hospital.Sex)  
title('Weight by Gender')
```



The order of the box plots corresponds to the new level order.

## See Also

`fitlm` | `getlevels` | `nominal` | `ordinal` | `reorderlevels`

## Related Examples

- “Change Category Labels” on page 2-10
- “Merge Category Levels” on page 2-22
- “Add and Drop Category Levels” on page 2-25
- “Index and Search Using Categorical Arrays” on page 2-53

## More About

- “Categorical Arrays” on page 2-48
- “Advantages of Using Categorical Arrays” on page 2-50

# Categorize Numeric Data

This example shows how to categorize numeric data into a categorical ordinal array using `ordinal`. This is useful for discretizing continuous data.

### Load sample data.

The dataset array, `hospital`, contains variables measured on a sample of patients. Compute the minimum, median, and maximum of the variable `Age`.

```
load hospital
quantile(hospital.Age, [0, .5, 1])
```

```
ans =
    25    39    50
```

The patient ages range from 25 to 50.

### Convert a numeric array to an ordinal array.

Group patients into the age categories `Under 30`, `30-39`, `Over 40`.

```
hospital.AgeCat = ordinal(hospital.Age, {'Under 30', '30-39', 'Over 40'}, ...
    [], [25, 30, 40, 50]);
getlevels(hospital.AgeCat)
```

```
ans =
    1x3 ordinal array
    Under 30    30-39    Over 40
```

The last input argument to `ordinal` has the endpoints for the categories. The first category begins at age 25, the second at age 30, and so on. The last category contains ages 40 and above, so begins at 40 and ends at 50 (the maximum age in the data set). To specify three categories, you must specify four endpoints (the last endpoint is the upper bound of the last category).

**Explore categories.**

Display the age and age category for the second patient.

```
dataset({hospital.Age(2), 'Age'}, ...
        {hospital.AgeCat(2), 'AgeCategory'})
```

```
ans =
```

```
   Age   AgeCategory
   43     Over 40
```

When you discretize a numeric array into categories, the categorical array loses all information about the actual numeric values. In this example, `AgeCat` is not numeric, and you cannot recover the raw data values from it.

**Categorize a numeric array into quartiles.**

The variable `Weight` has weight measurements for the sample patients. Categorize the patient weights into four categories, by quartile.

```
p = 0:.25:1;
breaks = quantile(hospital.Weight,p);
hospital.WeightQ = ordinal(hospital.Weight,{'Q1','Q2','Q3','Q4'},...
                           [],breaks);
getlevels(hospital.WeightQ)
```

```
ans =
```

```
   1x4 ordinal array
      Q1      Q2      Q3      Q4
```

**Explore categories.**

Display the weight and weight quartile for the second patient.

```
dataset({hospital.Weight(2), 'Weight'}, ...
        {hospital.WeightQ(2), 'WeightQuartile'})
```

```
ans =
```

```
Weight      WeightQuartile
163         Q3
```

### Summary statistics grouped by category levels.

Compute the mean systolic and diastolic blood pressure for each age and weight category.

```
grpstats(hospital, {'AgeCat', 'WeightQ'}, 'mean', 'DataVars', 'BloodPressure')
```

```
ans =
```

	AgeCat	WeightQ	GroupCount	mean_BloodPressure	
Under 30_Q1	Under 30	Q1	6	123.17	79.667
Under 30_Q2	Under 30	Q2	3	120.33	79.667
Under 30_Q3	Under 30	Q3	2	127.5	86.5
Under 30_Q4	Under 30	Q4	4	122	78
30-39_Q1	30-39	Q1	12	121.75	81.75
30-39_Q2	30-39	Q2	9	119.56	82.556
30-39_Q3	30-39	Q3	9	121	83.222
30-39_Q4	30-39	Q4	11	125.55	87.273
Over 40_Q1	Over 40	Q1	7	122.14	84.714
Over 40_Q2	Over 40	Q2	13	123.38	79.385
Over 40_Q3	Over 40	Q3	14	123.07	84.643
Over 40_Q4	Over 40	Q4	10	124.6	85.1

The variable `BloodPressure` is a matrix with two columns. The first column is systolic blood pressure, and the second column is diastolic blood pressure. The group in the sample with the highest mean diastolic blood pressure, 87.273, is aged 30–39 and in the highest weight quartile, 30–39\_Q4.

## See Also

`grpstats` | `ordinal`

## Related Examples

- “Create Nominal and Ordinal Arrays” on page 2-4
- “Merge Category Levels” on page 2-22

- “Plot Data Grouped by Category” on page 2-29
- “Index and Search Using Categorical Arrays” on page 2-53

### **More About**

- “Categorical Arrays” on page 2-48
- “Advantages of Using Categorical Arrays” on page 2-50

### Merge Category Levels

This example shows how to merge categories in a categorical array using `mergelevels`. This is useful for collapsing categories with few observations.

#### Load sample data.

```
load carsmall
```

#### Create a nominal array.

The variable `Origin` is a character array containing the country of origin for 100 sample cars. Convert `Origin` to a nominal array.

```
Origin = nominal(Origin);  
getlevels(Origin)
```

```
ans =
```

```
1x6 nominal array
```

```
France      Germany      Italy      Japan      Sweden      USA
```

There are six unique countries of origin in the data.

#### Tabulate category counts.

Explore the elements of the categorical array.

```
tabulate(Origin)
```

Value	Count	Percent
France	4	4.00%
Germany	9	9.00%
Italy	1	1.00%
Japan	15	15.00%
Sweden	2	2.00%
USA	69	69.00%

There are relatively few observations in each European country.



**Merge categories.**

Merge the categories France, Germany, Italy, and Sweden into one category called Europe.

```
Origin = mergelevels(Origin, {'France', 'Germany', 'Italy', 'Sweden'}, ...
                    'Europe');
getlevels(Origin)
```

```
ans =
    1x3 nominal array
    Europe      Japan      USA
```

The variable Origin now has only three category levels.

**Tabulate category counts.**

Explore the elements of the merged categories.

```
tabulate(Origin)
    Value      Count      Percent
    Europe      16      16.00%
    Japan       15      15.00%
    USA         69      69.00%
```

The category Europe has the 16% of observations that were previously distributed across four countries.

**See Also**

`mergelevels` | `nominal`

**Related Examples**

- “Create Nominal and Ordinal Arrays” on page 2-4
- “Add and Drop Category Levels” on page 2-25
- “Index and Search Using Categorical Arrays” on page 2-53

### **More About**

- “Categorical Arrays” on page 2-48
- “Advantages of Using Categorical Arrays” on page 2-50

## Add and Drop Category Levels

This example shows how to add and drop levels from a categorical array.

### Load sample data.

```
load('examgrades')
```

The array `grades` contains exam scores from 0 to 100 on five exams for a sample of 120 students.

### Create an ordinal array.

Assign letter grades to each student for each test using these categories.

Grade Range	Letter Grade
100	A+
90–99	A
80–89	B
70–79	C
60–69	D

```
letter = ordinal(grades, {'D', 'C', 'B', 'A', 'A+'}, [], ...
                  [60, 70, 80, 90, 100, 100]);
getlevels(letter)
```

```
ans =
```

```
      D      C      B      A      A+
```

There are five grade categories, in the specified order  $D < C < B < A < A+$ .

### Check for undefined categories.

Check whether or not there are any exam scores that do not fall into the five letter categories.

```
any(isundefined(letter))
```

```
ans =
```

```
      1      0      1      1      0
```

Recall that there are five exam scores for each student. The previous command returns a logical value for each of the five exams, indicating whether there are any scores that are `<undefined>`. There are scores for the first, third, and fourth exams that are `<undefined>`, that is, missing a category level.

### Identify elements in undefined categories.

You can find the exam scores that do not have a letter grade using the `isundefined` logical condition.

```
grades(isundefined(letter))
```

```
ans =
```

```
55  
59  
58  
59  
54  
57  
56  
59  
59  
50  
59  
52
```

The exam scores that are in the 50s do not have a letter grade.

### Add a new category.

Put all scores that are `<undefined>` into a new category labeled `D-`.

```
letter(isundefined(letter)) = 'D-';  
getlevels(letter)
```

```
Warning: Categorical level 'D-' being added.  
> In categorical.subsasgn at 55
```

```
ans =
```

```
D      C      B      A      A+     D-
```

The ordinal variable, `letter`, has a new category added to the end.

**Reorder category levels.**

Reorder the categories so that D- < D.

```
letter = reorderlevels(letter, {'D-', 'D', 'C', 'B', 'A', 'A+'});
getlevels(letter)
```

```
ans =
```

```
    D-     D     C     B     A     A+
```

**Compare elements.**

Now that all exam scores have a letter grade, count how many students received a higher letter grade on the second test than on the first test.

```
sum(letter(:,2) > letter(:,1))
```

```
ans =
```

```
    32
```

Thirty-two students improved their letter grade between the first two exams.

**Explore categories.**

Count the number of A+ scores in each of the five exams.

```
sum(letter=='A+')
```

```
ans =
```

```
    0     0     0     0     0
```

There are no A+ scores on any of the five exams.

**Drop a category.**

Drop the category A+ from the ordinal variable, letter.

```
letter = droplevels(letter, 'A+');
getlevels(letter)
```

```
ans =
```

```
    D-     D     C     B     A
```

Category A+ is no longer in the ordinal variable, `letter`.

### See Also

`droplevels` | `ordinal` | `reorderlevels`

### Related Examples

- “Create Nominal and Ordinal Arrays” on page 2-4
- “Reorder Category Levels” on page 2-12
- “Merge Category Levels” on page 2-22
- “Index and Search Using Categorical Arrays” on page 2-53

### More About

- “Categorical Arrays” on page 2-48
- “Advantages of Using Categorical Arrays” on page 2-50

## Plot Data Grouped by Category

This example shows how to plot data grouped by the levels of a categorical variable.

### Load sample data.

```
load carsmall
```

The variable `Acceleration` contains acceleration measurements on 100 sample cars. The variable `Origin` is a character array containing the country of origin for each car.

### Create a nominal array.

Convert `Origin` to a nominal array.

```
Origin = nominal(Origin);  
getlevels(Origin)
```

```
ans =
```

```
1x6 nominal array
```

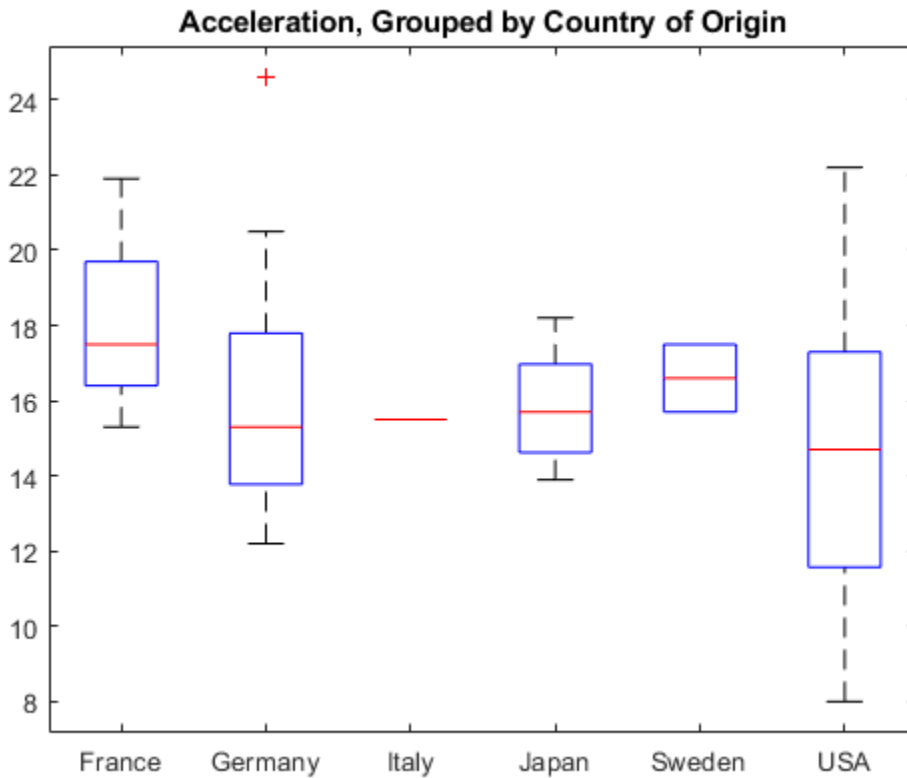
```
France      Germany      Italy      Japan      Sweden      USA
```

There are six unique countries of origin in the sample. By default, `nominal` orders the countries in ascending alphabetical order.

### Plot data grouped by category.

Draw box plots for `Acceleration`, grouped by `Origin`.

```
figure  
boxplot(Acceleration, Origin)  
title('Acceleration, Grouped by Country of Origin')
```



The box plots appear in the same order as the categorical levels (use `reorderlevels` to change the order of the categories).

Few observations have Italy as the country of origin.

#### **Tabulate category counts.**

Tabulate the number of sample cars from each country.

```
tabulate(Origin)
```

Value	Count	Percent
France	4	4.00%
Germany	9	9.00%



Italy	1	1.00%
Japan	15	15.00%
Sweden	2	2.00%
USA	69	69.00%

Only one car is made in Italy.

### Drop a category.

Delete the Italian car from the sample.

```
Acceleration2 = Acceleration(Origin~='Italy');
Origin2 = Origin(Origin~='Italy');
getlevels(Origin2)
```

```
ans =
```

```
1x6 nominal array
```

```
France      Germany      Italy      Japan      Sweden      USA
```

Even though the car from Italy is no longer in the sample, the nominal variable, `Origin2`, still has the category `Italy`. Note that this is intentional—the levels of a categorical array do not necessarily coincide with the values.

### Drop a category level.

Use `droplevels` to remove the Italy category.

```
Origin2 = droplevels(Origin2, 'Italy');
tabulate(Origin2)
```

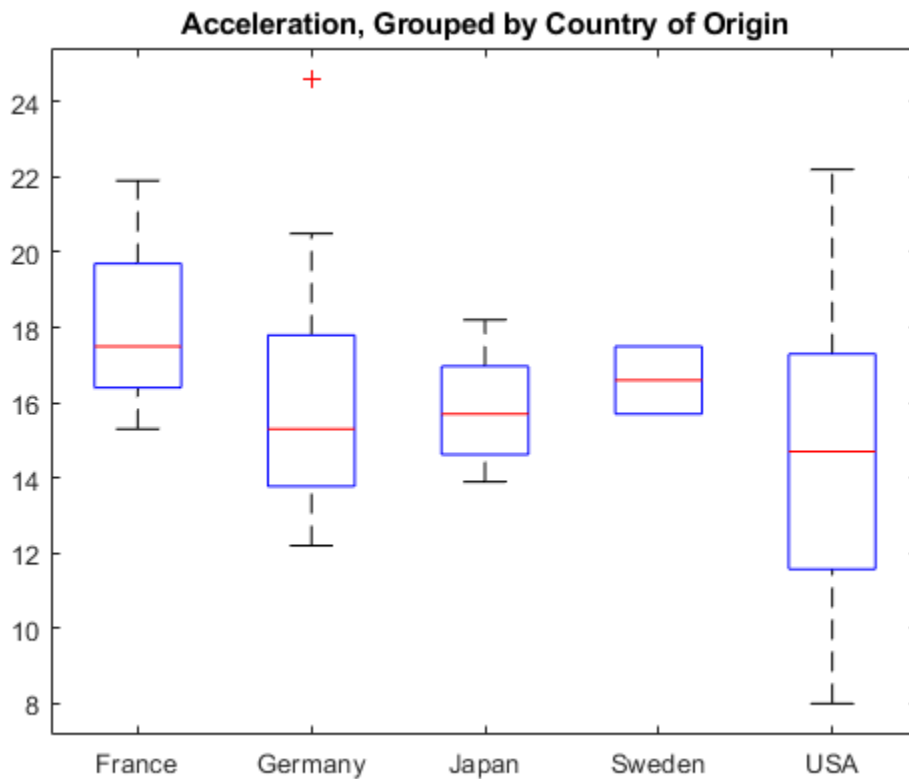
Value	Count	Percent
France	4	4.04%
Germany	9	9.09%
Japan	15	15.15%
Sweden	2	2.02%
USA	69	69.70%

The `Italy` category is no longer in the nominal array, `Origin2`.

**Plot data grouped by category.**

Draw box plots of `Acceleration2`, grouped by `Origin2`.

```
figure
boxplot(Acceleration2,Origin2)
title('Acceleration, Grouped by Country of Origin')
```



The plot no longer includes the car from Italy.

### See Also

`boxplot` | `droplevels` | `nominal` | `reorderlevels`

## **Related Examples**

- “Test Differences Between Category Means” on page 2-34
- “Summary Statistics Grouped by Category” on page 2-44
- “Regression with Categorical Covariates” on page 2-65

## **More About**

- “Categorical Arrays” on page 2-48
- “Advantages of Using Categorical Arrays” on page 2-50
- “Grouping Variables” on page 2-59

# Test Differences Between Category Means

This example shows how to test for significant differences between category (group) means using a *t*-test, two-way ANOVA (analysis of variance), and ANOCOVA (analysis of covariance) analysis.

The goal is determining if the expected miles per gallon for a car depends on the decade in which it was manufactured, or the location where it was manufactured.

### Load sample data.

```
load('carsmall')
unique(Model_Year)

ans =

    70
    76
    82
```

The variable MPG has miles per gallon measurements on a sample of 100 cars. The variables Model\_Year and Origin contain the model year and country of origin for each car.

The first factor of interest is the decade of manufacture. There are three manufacturing years in the data.

### Create a factor for the decade of manufacture.

Create an ordinal array named Decade by merging the observations from years 70 and 76 into a category labeled 1970s, and putting the observations from 82 into a category labeled 1980s.

```
Decade = ordinal(Model_Year, {'1970s', '1980s'}, [], [70 77 82]);
getlevels(Decade)

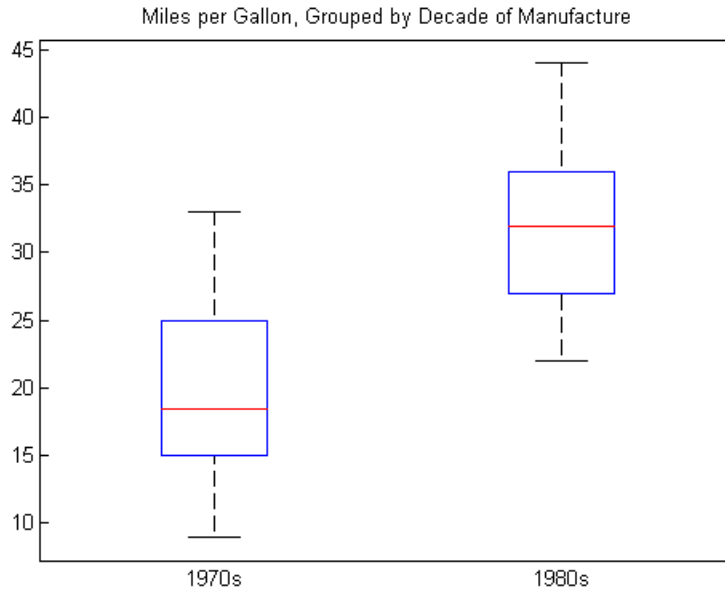
ans =

    1970s    1980s
```

### Plot data grouped by category.

Draw a box plot of miles per gallon, grouped by the decade of manufacture.

```
figure()
boxplot(MPG, Decade)
title('Miles per Gallon, Grouped by Decade of Manufacture')
```



The box plot suggests that miles per gallon is higher in cars manufactured during the 1980s compared to the 1970s.

### Compute summary statistics.

Compute the mean and variance of miles per gallon for each decade.

```
[xbar, s2, grp] = grpstats(MPG, Decade, {'mean', 'var', 'gname'})
```

```
xbar =
```

```
19.7857
31.7097
```

```
s2 =
```

```
35.1429
29.0796
```

```
grp =
    '1970s'
    '1980s'
```

This output shows that the mean miles per gallon in the 1980s was 31.71, compared to 19.79 in the 1970s. The variances in the two groups are similar.

### Conduct a two-sample *t*-test for equal group means.

Conduct a two-sample *t*-test, assuming equal variances, to test for a significant difference between the group means. The hypothesis is

$$H_0 : \mu_{70} = \mu_{80}$$

$$H_A : \mu_{70} \neq \mu_{80}.$$

```
MPG70 = MPG(Decade=='1970s');
MPG80 = MPG(Decade=='1980s');
[h,p] = ttest2(MPG70,MPG80)
```

```
h =
    1
```

```
p =
    3.4809e-15
```

The logical value 1 indicates the null hypothesis is rejected at the default 0.05 significance level. The *p*-value for the test is very small. There is sufficient evidence that the mean miles per gallon in the 1980s differs from the mean miles per gallon in the 1970s.

### Create a factor for the location of manufacture.

The second factor of interest is the location of manufacture. First, convert `Origin` to a nominal array.

```
Location = nominal(Origin);
tabulate(Location)
```

```
tabulate(Location)
  Value      Count      Percent
  France         4       4.00%
  Germany        9       9.00%
  Italy           1       1.00%
  Japan         15      15.00%
  Sweden         2       2.00%
  USA           69      69.00%
```

There are six different countries of manufacture. The European countries have relatively few observations.

### Merge categories.

Combine the categories France, Germany, Italy, and Sweden into a new category named Europe.

```
Location = mergelevels(Location, {'France', 'Germany', 'Italy', 'Sweden'}, ...
                          'Europe');
tabulate(Location)
```

```
  Value      Count      Percent
  Japan         15      15.00%
  USA           69      69.00%
  Europe        16      16.00%
```

### Compute summary statistics.

Compute the mean miles per gallon, grouped by the location of manufacture.

```
[xbar, grp] = grpstats(MPG, Location, {'mean', 'gname'})
```

```
xbar =
```

```
31.8000
21.1328
26.6667
```

```
grp =
```

```
 'Japan'
```

```
'USA'  
'Europe'
```

This result shows that average miles per gallon is lowest for the sample of cars manufactured in the U.S.

### Conduct two-way ANOVA.

Conduct a two-way ANOVA to test for differences in expected miles per gallon between factor levels for `Decade` and `Location`.

The statistical model is

$$MPG_{ij} = \mu + \alpha_i + \beta_j + \varepsilon_{ij}, \quad i = 1, 2; j = 1, 2, 3,$$

where  $MPG_{ij}$  is the response, miles per gallon, for cars made in decade  $i$  at location  $j$ . The treatment effects for the first factor, decade of manufacture, are the  $\alpha_i$  terms (constrained to sum to zero). The treatment effects for the second factor, location of manufacture, are the  $\beta_j$  terms (constrained to sum to zero). The  $\varepsilon_{ij}$  are uncorrelated, normally distributed noise terms.

The hypotheses to test are equality of decade effects,

$$H_0 : \alpha_1 = \alpha_2 = 0$$

$$H_A : \text{at least one } \alpha_i \neq 0,$$

and equality of location effects,

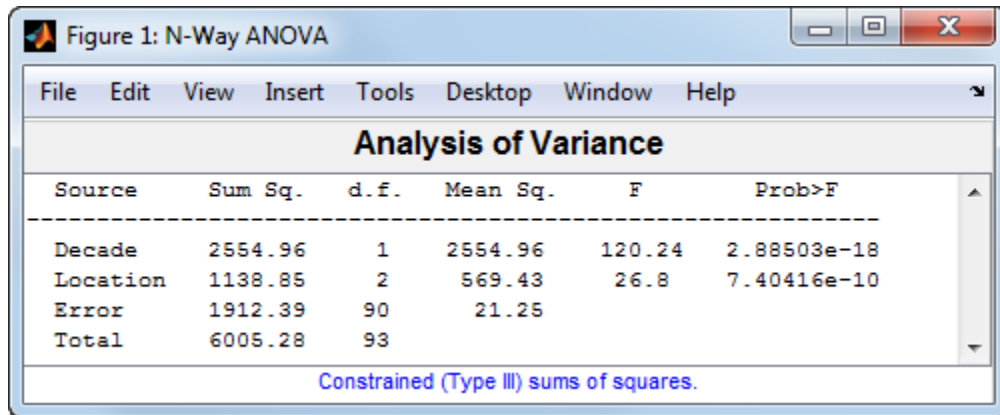
$$H_0 : \beta_1 = \beta_2 = \beta_3 = 0$$

$$H_A : \text{at least one } \beta_j \neq 0.$$

You can conduct a multiple-factor ANOVA using `anovan`.

```
anovan(MPG, {Decade, Location}, 'varnames', {'Decade', 'Location'});
```





This output shows the results of the two-way ANOVA. The p-value for testing the equality of decade effects is  $2.88503e-18$ , so the null hypothesis is rejected at the 0.05 significance level. The p-value for testing the equality of location effects is  $7.40416e-10$ , so this null hypothesis is also rejected.

### Conduct ANOCOVA analysis.

A potential confounder in this analysis is car weight. Cars with greater weight are expected to have lower gas mileage. Include the variable `Weight` as a continuous covariate in the ANOVA; that is, conduct an ANOCOVA analysis.

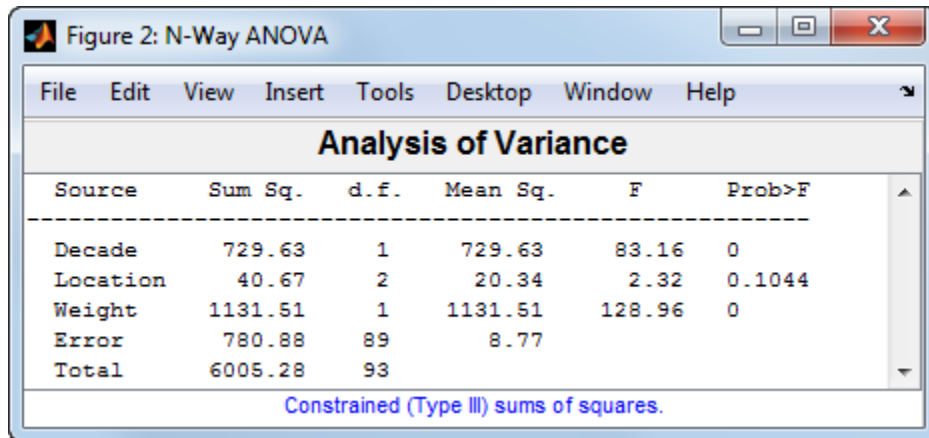
Assuming parallel lines, the statistical model is

$$MPG_{ijk} = \mu + \alpha_i + \beta_j + \gamma Weight_{ijk} + \varepsilon_{ijk}, \quad i = 1, 2; \quad j = 1, 2, 3; \quad k = 1, \dots, 100.$$

The difference between this model and the two-way ANOVA model is the inclusion of the continuous predictor,  $Weight_{ijk}$ , the weight for the  $k$ th car, which was made in the  $i$ th decade and in the  $j$ th location. The slope parameter is  $\gamma$ .

Add the continuous covariate as a third group in the second `anovan` input argument. Use the name-value pair argument `Continuous` to specify that `Weight` (the third group) is continuous.

```
anovan(MPG, {Decade, Location, Weight}, 'Continuous', 3, ...
    'varnames', {'Decade', 'Location', 'Weight'});
```



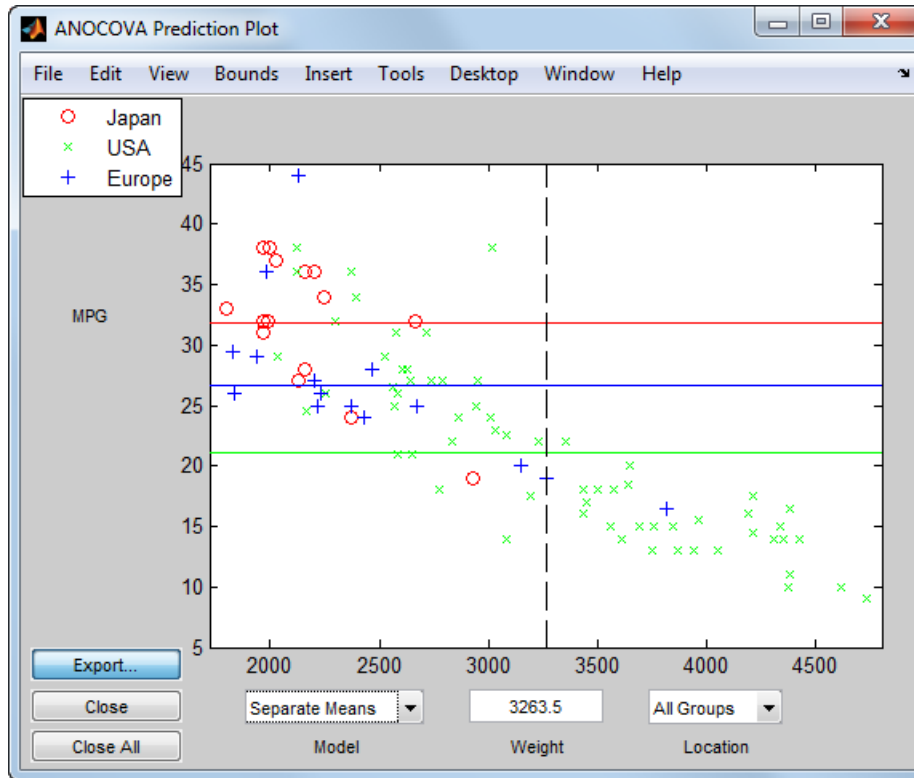
This output shows that when car weight is considered, there is insufficient evidence of a manufacturing location effect (p-value = 0.1044).

### Use interactive tool.

You can use the interactive `aocool` to explore this result.

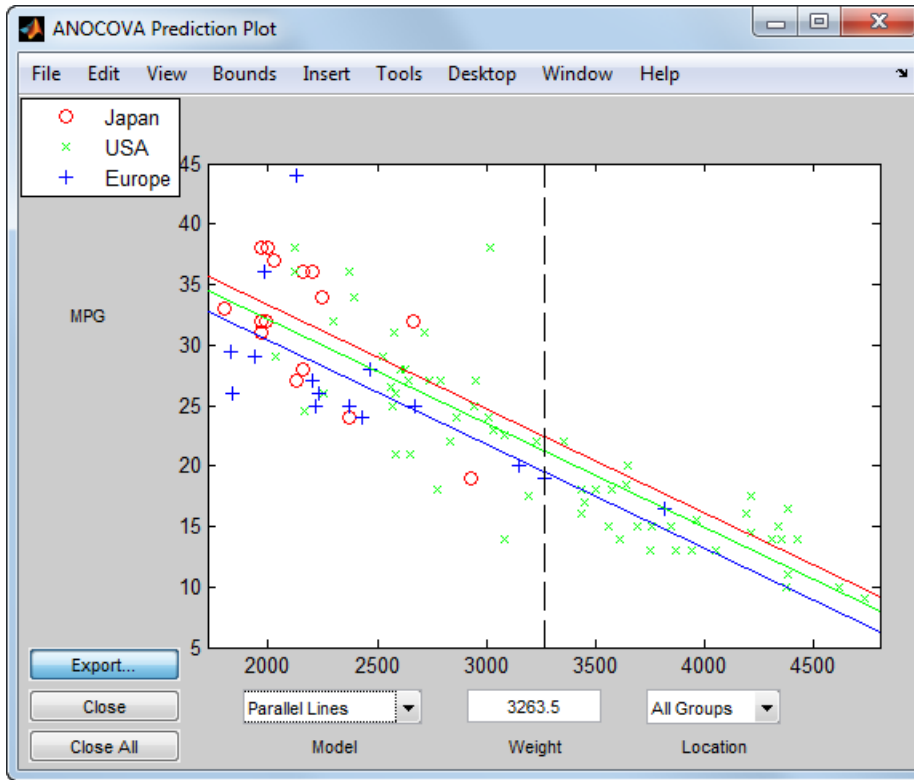
```
aocool(Weight,MPG,Location);
```

This command opens three dialog boxes. In the ANOCOVA Prediction Plot dialog box, select the **Separate Means** model.



This output shows that when you do not include *Weight* in the model, there are fairly large differences in the expected miles per gallon among the three manufacturing locations. Note that here the model does not adjust for the decade of manufacturing.

Now, select the **Parallel Lines** model.



When you include weight in the model, the difference in expected miles per gallon among the three manufacturing locations is much smaller.

## See Also

[anovan](#) | [aoctool](#) | [boxplot](#) | [grpstats](#) | [nominal](#) | [ordinal](#) | [ttest2](#)

## Related Examples

- “Plot Data Grouped by Category” on page 2-29
- “Summary Statistics Grouped by Category” on page 2-44
- “Regression with Categorical Covariates” on page 2-65

## More About

- “Categorical Arrays” on page 2-48
- “Advantages of Using Categorical Arrays” on page 2-50
- “Grouping Variables” on page 2-59

## Summary Statistics Grouped by Category

This example shows how to compute summary statistics grouped by levels of a categorical variable. You can compute group summary statistics for a numeric array or a dataset array using `grpstats`.

### Load sample data.

```
load hospital
```

The dataset array, `hospital`, has 7 variables (columns) and 100 observations (rows).

### Compute summary statistics by category.

The variable `Sex` is a nominal array with two levels, `Male` and `Female`. Compute the minimum and maximum weights for each gender.

```
stats = grpstats(hospital, 'Sex', {'min', 'max'}, 'DataVars', 'Weight')
```

```
stats =
```

	Sex	GroupCount	min_Weight	max_Weight
Female	Female	53	111	147
Male	Male	47	158	202

The dataset array, `stats`, has observations corresponding to the levels of the variable `Sex`. The variable `min_Weight` contains the minimum weight for each group, and the variable `max_Weight` contains the maximum weight for each group.

### Compute summary statistics by multiple categories.

The variable `Smoker` is a logical array with value 1 for smokers and value 0 for nonsmokers. Compute the minimum and maximum weights for each gender and smoking combination.

```
stats = grpstats(hospital, {'Sex', 'Smoker'}, {'min', 'max'}, ...  
                'DataVars', 'Weight')
```

```
stats =
```

---

	Sex	Smoker	GroupCount	min_Weight	max_Weight
Female_0	Female	false	40	111	147
Female_1	Female	true	13	115	146
Male_0	Male	false	26	158	194
Male_1	Male	true	21	164	202

The dataset array, `stats`, has an observation row for each combination of levels of `Sex` and `Smoker` in the original data.

## See Also

`dataset` | `grpstats` | `nominal`

## Related Examples

- “Plot Data Grouped by Category” on page 2-29
- “Test Differences Between Category Means” on page 2-34
- “Calculations on Dataset Arrays” on page 2-117

## More About

- “Grouping Variables” on page 2-59
- “Categorical Arrays” on page 2-48
- “Dataset Arrays” on page 2-142

### Sort Ordinal Arrays

This example shows how to determine sorting order for ordinal arrays.

#### Load sample data.

```
AllSizes = {'medium', 'large', 'small', 'small', 'medium', ...  
           'large', 'medium', 'small'};
```

The created variable, `AllSizes`, is a cell array of character vectors containing size measurements on eight objects.

#### Create an ordinal array.

Convert `AllSizes` to an ordinal array with levels `small < medium < large`.

```
AllSizes = ordinal(AllSizes, {}, {'small', 'medium', 'large'});  
getlevels(AllSizes)
```

```
ans =  
  
    1x3 ordinal array  
  
    small    medium    large
```

#### Sort the ordinal array.

When you sort ordinal arrays, the sorted observations are in the same order as the category levels.

```
sizeSort = sort(AllSizes);  
sizeSort(:)
```

```
ans =  
  
    8x1 ordinal array  
  
    small  
    small  
    small  
    medium
```



```
medium  
medium  
large  
large
```

The sorted ordinal array, `sizeSort`, contains the observations ordered from small to large.

## See Also

`ordinal`

## Related Examples

- “Reorder Category Levels” on page 2-12
- “Add and Drop Category Levels” on page 2-25

## More About

- “Categorical Arrays” on page 2-48
- “Advantages of Using Categorical Arrays” on page 2-50

# Categorical Arrays

---

**Note** The `nominal` and `ordinal` array data types might be removed in a future release. To represent ordered and unordered discrete, nonnumeric data, use the MATLAB `categorical` data type instead.

---

In this section...
“What Are Categorical Arrays?” on page 2-48
“Categorical Array Conversion” on page 2-48

## What Are Categorical Arrays?

Categorical arrays are Statistics and Machine Learning Toolbox data types for storing categorical values. Categorical arrays store data that have a finite set of discrete levels, which might or might not have a natural order. There are two types of categorical arrays:

- `ordinal` arrays store categorical values with ordered levels. For example, an ordinal variable might have levels {small, medium, large}.
- `nominal` arrays store categorical values with unordered levels. For example, a nominal variable might have levels {red, blue, green}.

In experimental design, these variables are often called factors, with ordered or unordered factor levels.

Categorical arrays are convenient and memory efficient containers for storing categorical variables. In addition to storing information about which category each observation belongs to, categorical arrays store descriptive metadata including category labels and order.

Categorical arrays have associated methods that streamline common tasks such as merging categories, adding or dropping levels, and changing level labels.

## Categorical Array Conversion

You can easily convert to and from categorical arrays. To create a nominal or ordinal array, use `nominal` or `ordinal`, respectively. You can convert these data types to categorical arrays:

- Numeric array
- Logical array
- Character array
- Cell array of character vectors

## See Also

`nominal` | `ordinal`

## Related Examples

- “Create Nominal and Ordinal Arrays” on page 2-4
- “Summary Statistics Grouped by Category” on page 2-44
- “Plot Data Grouped by Category” on page 2-29
- “Index and Search Using Categorical Arrays” on page 2-53

## More About

- “Advantages of Using Categorical Arrays” on page 2-50
- “Grouping Variables” on page 2-59

## Advantages of Using Categorical Arrays

---

**Note** The `nominal` and `ordinal` array data types might be removed in a future release. To represent ordered and unordered discrete, nonnumeric data, use the MATLAB `categorical` data type instead.

---

### In this section...

“Manipulate Category Levels” on page 2-50

“Analysis Using Categorical Arrays” on page 2-50

“Reduce Memory Requirements” on page 2-51

## Manipulate Category Levels

When working with categorical variables and their levels, you’ll encounter some typical challenges. This table summarizes the functions you can use with categorical arrays to manipulate category levels. For additional functions, type methods `nominal` or methods `ordinal` at the command line, or see the `nominal` and `ordinal` reference pages.

Task	Function
Add new category levels	<code>addlevels</code>
Drop category levels	<code>droplevels</code>
Combine category levels	<code>mergelevels</code>
Reorder category levels	<code>reorderlevels</code>
Count the number of observations in each category	<code>levelcounts</code>
Change the label or name of category levels	<code>setlabels</code>
Create an interaction factor	<code>times</code>
Find observations that are not in a defined category	<code>isundefined</code>

## Analysis Using Categorical Arrays

You can use categorical arrays in a variety of statistical analyses. For example, you might want to compute descriptive statistics for data grouped by the category levels,

conduct statistical tests on differences between category means, or perform regression analysis using categorical predictors.

Statistics and Machine Learning Toolbox functions that accept a grouping variable as an input argument accept categorical arrays. This includes descriptive functions such as:

- `grpstats`
- `gscatter`
- `boxplot`
- `gplotmatrix`

You can also use categorical arrays as input arguments to analysis functions and methods based on models, such as:

- `anovan`
- `fitlm`
- `fitglm`
- `fitnlm`

When you use a categorical array as a predictor in these functions, the fitting function automatically recognizes the categorical predictor, and constructs appropriate dummy indicator variables for analysis. Alternatively, you can construct your own dummy indicator variables using `dummyvar`.

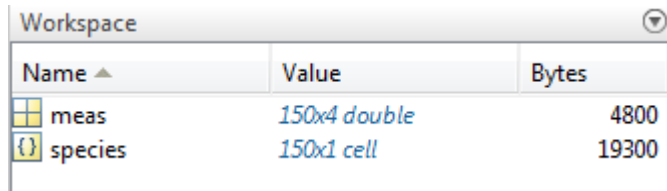
## Reduce Memory Requirements

The levels of categorical variables are often defined as text, which can be costly to store and manipulate in a cell array of character vectors or `char` array. Categorical arrays separately store category membership and category labels, greatly reducing the amount of memory required to store the variable.

For example, load some sample data:

```
load('fisheriris')
```

The variable `species` is a cell array of character vectors requiring 19,300 bytes of memory.

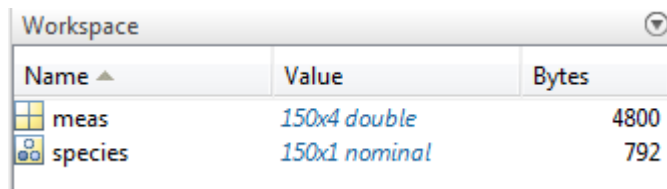


The screenshot shows the MATLAB Workspace window with two variables:

Name ▲	Value	Bytes
meas	150x4 double	4800
species	150x1 cell	19300

Convert `species` to a nominal array:

```
species = nominal(species);
```



The screenshot shows the MATLAB Workspace window after the conversion. The `species` variable is now a nominal array:

Name ▲	Value	Bytes
meas	150x4 double	4800
species	150x1 nominal	792

There is a 95% reduction in memory required to store the variable.

## See Also

[nominal](#) | [ordinal](#)

## Related Examples

- “Create Nominal and Ordinal Arrays” on page 2-4
- “Test Differences Between Category Means” on page 2-34
- “Regression with Categorical Covariates” on page 2-65
- “Index and Search Using Categorical Arrays” on page 2-53

## More About

- “Categorical Arrays” on page 2-48
- “Grouping Variables” on page 2-59
- “Dummy Indicator Variables” on page 2-62

## Index and Search Using Categorical Arrays

---

**Note** The `nominal` and `ordinal` array data types might be removed in a future release. To represent ordered and unordered discrete, nonnumeric data, use the MATLAB `categorical` data type instead.

---

### Index By Category

It is often useful to index and search data by its category, or group. If you store categories as labels inside a cell array of character vectors or `char` array, it can be difficult to index and search the categories. When using categorical arrays, you can easily:

- **Index elements from particular categories.** For both nominal and ordinal arrays, you can use the logical operators `==` and `~=` to index the observations that are in, or not in, a particular category. For ordinal arrays, which have an encoded order, you can also use inequalities, `>`, `>=`, `<`, and `<=`, to find observations in categories above or below a particular category.
- **Search for members of a category.** In addition to the logical operator `==`, you can use `ismember` to find observations in a particular group.
- **Find elements that are not in a defined category.** Categorical arrays indicate which elements do not belong to a defined category by `<undefined>`. You can use `isundefined` to find observations missing a category.
- **Delete observations that are in a particular category.** You can use logical operators to include or exclude observations from particular categories. Even if you remove all observations from a category, the category level remains defined unless you remove it using `droplevels`.

### Common Indexing and Searching Methods

This example shows several common indexing and searching methods.

Load the sample data.

```
load carsmall;
```

Convert the `char` array, `Origin`, to a nominal array. This variable contains the country of origin, or manufacture, for each sample car.

```
Origin = nominal(Origin);
```

Search for observations in a category. Determine if there are any cars in the sample that were manufactured in Canada.

```
any(Origin=='Canada')
```

```
ans =
```

```
logical
```

```
0
```

There are no sample cars manufactured in Canada.

List the countries that are levels of Origin.

```
getlevels(Origin)
```

```
ans =
```

```
1x6 nominal array
```

```
France
```

```
Germany
```

```
Italy
```

```
Japan
```

```
Sweden
```

```
USA
```

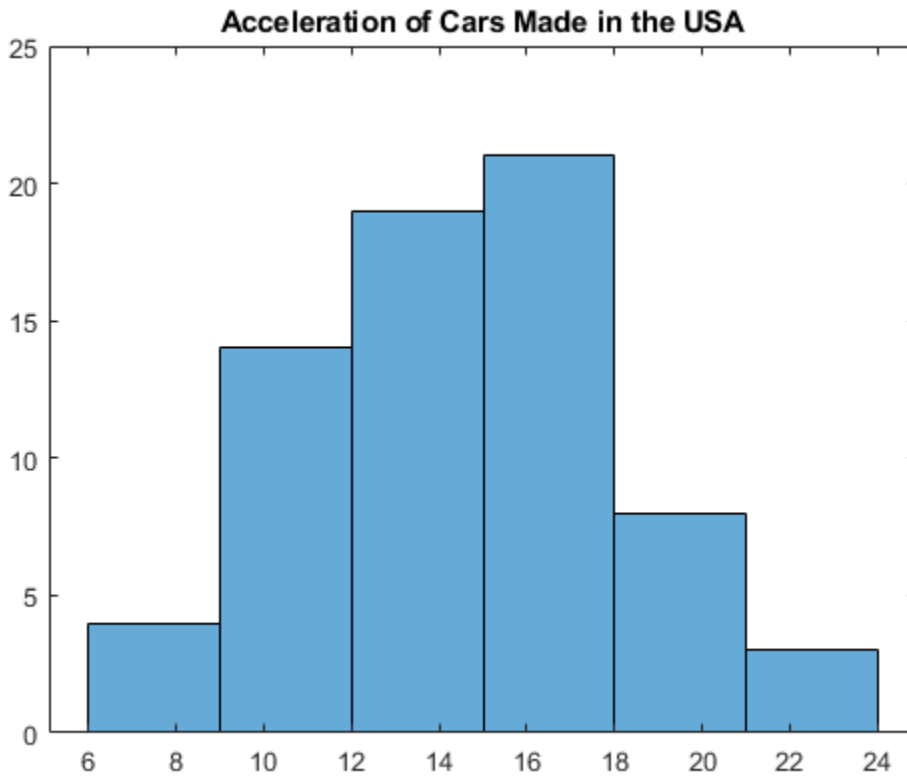
Index elements that are in a particular category. Plot a histogram of the acceleration measurements for cars made in the U.S.

```
figure();
```

```
histogram(Acceleration(Origin=='USA'))
```

```
title('Acceleration of Cars Made in the USA')
```





Delete observations that are in a particular category. Delete all cars made in Sweden from `Origin`.

```
Origin = Origin(Origin~='Sweden');  
any(ismember(Origin, 'Sweden'))
```

```
ans =
```

```
logical
```

```
0
```

The cars made in Sweden are deleted from `Origin`, but Sweden is still a level of `Origin`.

```
getlevels(Origin)
```

```
ans =
```

```
1x6 nominal array
```

```
France    Germany    Italy    Japan    Sweden    USA
```

Remove Sweden from the levels of `Origin`.

```
Origin = droplevels(Origin, 'Sweden');
```

```
getlevels(Origin)
```

```
ans =
```

```
1x5 nominal array
```

```
France    Germany    Italy    Japan    USA
```

Check for observations not in a defined category. Get the indices for the cars made in France.

```
ix = find(Origin=='France')
```

```
ix =
```

```
11  
27  
39  
61
```

There are four cars from France. Remove France from the levels of `Origin`.

```
Origin = droplevels(Origin, 'France');
```

This returns a warning indicating that you are dropping a category level that has elements in it. These observations are no longer in a defined category, indicated by `undefined`.

```
Origin(ix)

ans =

  4x1 nominal array

   <undefined>
   <undefined>
   <undefined>
   <undefined>
```

You can use `isundefined` to search for observations with an undefined category.

```
find(isundefined(Origin))

ans =

  11
  27
  39
  61
```

These indices correspond to the observations that were in category `France`, before that category was dropped from `Origin`.

## See Also

`droplevels` | `nominal` | `ordinal`

## Related Examples

- “Create Nominal and Ordinal Arrays” on page 2-4
- “Reorder Category Levels” on page 2-12
- “Merge Category Levels” on page 2-22
- “Add and Drop Category Levels” on page 2-25

### **More About**

- “Categorical Arrays” on page 2-48
- “Advantages of Using Categorical Arrays” on page 2-50

## Grouping Variables

In this section...
“What Are Grouping Variables?” on page 2-59
“Group Definition” on page 2-60
“Analysis Using Grouping Variables” on page 2-60
“Missing Group Values” on page 2-61

### What Are Grouping Variables?

Grouping variables are utility variables used to group, or categorize, observations. Grouping variables are useful for summarizing or visualizing data by group. A grouping variable can be any of these data types:

- Numeric vector
- Logical vector
- Character array
- Cell array of character vectors
- Categorical vector

A grouping variable must have the same number of observations (rows) as the table, dataset array, or numeric array you are grouping. Observations that have the same grouping variable value belong to the same group.

For example, the following variables comprise the same groups. Each grouping variable divides five observations into two groups. The first group contains the first and fourth observations. The other three observations are in the second group.

Data Type	Grouping Variable
Numeric vector	[1 2 2 1 2]
Logical vector	[0 1 1 0 1]
Cell array of character vectors	{ 'Male', 'Female', 'Female', 'Male', 'Female' ' }
Categorical vector	Male Female Female Male Female

Grouping variables with labels gives each group a meaningful name. A categorical array is an efficient and flexible choice of grouping variable.

## Group Definition

Typically, there are as many groups as unique values in the grouping variable. However, categorical arrays can have levels that are not represented in the data. The groups and the order of the groups depend on the data type of the grouping variable. Suppose  $G$  is a grouping variable.

- If  $G$  is a numeric or logical vector, then the groups correspond to the distinct values in  $G$ , in the sorted order of the unique values.
- If  $G$  is a character array or cell array of character vectors, then the groups correspond to the distinct elements in  $G$ , in the order of their first appearance.
- If  $G$  is a categorical vector, then the groups correspond to the unique category levels in  $G$ , in the order returned by `getlevels`.

Some functions, such as `grpstats`, accept multiple grouping variables specified as a cell array of grouping variables, for example,  $\{G1, G2, G3\}$ . In this case, the groups are defined by the unique combinations of values in the grouping variables. The order is decided first by the order of the first grouping variable, then by the order of the second grouping variable, and so on.

## Analysis Using Grouping Variables

This table lists common tasks you might want to perform using grouping variables.

Grouping Task	Function Accepting Grouping Variable
Draw side-by-side boxplots for data in different groups.	<code>boxplot</code>
Draw a scatter plot with markers colored by group.	<code>gscatter</code>
Draw a scatter plot matrix with markers colored by group.	<code>gplotmatrix</code>
Compute summary statistics by group.	<code>grpstats</code>
Test for differences between group means.	<code>anovan</code>
Create an index vector from a grouping variable.	<code>grp2idx</code>

## Missing Group Values

Grouping variables can have missing values provided you include a valid indicator.

Grouping Variable Data Type	Missing Value Indicator
Numeric vector	NaN
Logical vector	(Cannot be missing)
Character array	Row of spaces
Cell array of character vectors	' '
Categorical vector	<undefined>

## See Also

`nominal` | `ordinal`

## Related Examples

- “Plot Data Grouped by Category” on page 2-29
- “Summary Statistics Grouped by Category” on page 2-44

## More About

- “Categorical Arrays” on page 2-48
- “Advantages of Using Categorical Arrays” on page 2-50
- `nominal`
- `ordinal`

## Dummy Indicator Variables

### In this section...

“What Are Dummy Variables?” on page 2-62

“Creating Dummy Variables” on page 2-63

### What Are Dummy Variables?

When performing regression analysis, it is common to include both continuous and categorical (quantitative and qualitative) predictor variables. When including a categorical independent variable, it is important not to input the variable as a numeric array. Numeric arrays have both order and magnitude. A categorical variable might have order (for example, an ordinal variable), but it does not have magnitude. Using a numeric array implies a known “distance” between the categories.

The appropriate way to include categorical predictors is as dummy indicator variables. An indicator variable has values 0 and 1. A categorical variable with  $c$  categories can be represented by  $c - 1$  indicator variables.

For example, suppose you have a categorical variable with levels {Small, Medium, Large}. You can represent this variable using two dummy variables, as shown in this figure.

	X <sub>1</sub>	X <sub>2</sub>	
Small	0	0	Reference Group
Medium	1	0	
Large	0	1	

In this example,  $X_1$  is a dummy variable that has value 1 for the Medium group, and 0 otherwise.  $X_2$  is a dummy variable that has value 1 for the Large group, and 0 otherwise.



Together, these two variables represent the three categories. Observations in the `Small` group have 0s for both dummy variables.

The category represented by all 0s is the reference group. When you include the dummy variables in a regression model, the coefficients of the dummy variables are interpreted with respect to the reference group.

## Creating Dummy Variables

### Automatic Creation of Dummy Variables

Most classification and regression fitting functions accept categorical predictors.

- If the predictor data is in a table, the function assumes that a variable is categorical if it contains logical values, categorical values, or a cell array of character vectors.
- If the predictor data is a matrix, the function assumes all predictors are continuous. To identify any categorical predictors, use the `'CategoricalPredictors'` or `'CategoricalVars'` name-value pair argument.

For parametric regression fitting functions such as `fitlm` and `fitglm`, if there are  $c$  unique levels in the categorical array, then the fitting function estimates  $c - 1$  coefficients for the categorical predictor.

### Manual Creation of Dummy Variables

If you prefer to create your own dummy variable design matrix, use `dummyvar`. This function accepts a numeric or categorical column vector, and returns a matrix of indicator variables. The dummy variable design matrix has a column for every group, and a row for every observation.

For example,

```
gender = nominal({'Male'; 'Female'; 'Female'; 'Male'; 'Female'});
dv = dummyvar(gender)
```

```
dv =
```

```

0     1
1     0
1     0
0     1
1     0
```

There are five rows corresponding to the number of rows in `gender`, and two columns for the unique groups, `Female` and `Male`. Column order corresponds to the order of the levels in `gender`. For nominal arrays, the default order is ascending alphabetical.

To use these dummy variables in a regression model, you must either delete a column (to create a reference group), or fit a regression model with no intercept term. For the `gender` example, only one dummy variable is needed to represent two genders. Notice what happens if you add an intercept term to the complete design matrix, `dv`.

```
X = [ones(5,1) dv]
```

```
X =
```

```
    1     0     1
    1     1     0
    1     1     0
    1     0     1
    1     1     0
```

```
rank(X)
```

```
ans =
```

```
    2
```

The design matrix with an intercept term is not of full rank, and is not invertible. Because of this linear dependence, use only  $c - 1$  indicator variables to represent a categorical variable with  $c$  categories in a regression model with an intercept term.

## See Also

`dummyvar` | `fitglm` | `fitlm`

## Related Examples

- “Regression with Categorical Covariates” on page 2-65
- “Test Differences Between Category Means” on page 2-34

## Regression with Categorical Covariates

This example shows how to perform a regression with categorical covariates using categorical arrays and `fitlm`.

### Load sample data.

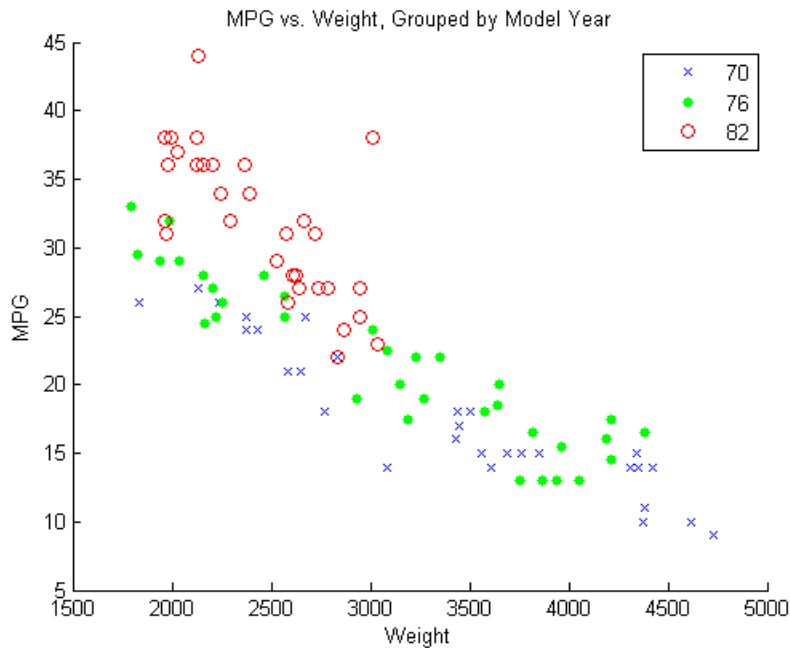
```
load('carsmall')
```

The variable `MPG` contains measurements on the miles per gallon of 100 sample cars. The model year of each car is in the variable `Model_Year`, and `Weight` contains the weight of each car.

### Plot grouped data.

Draw a scatter plot of `MPG` against `Weight`, grouped by model year.

```
figure()
gscatter(Weight,MPG,Model_Year,'bgr','x.o')
title('MPG vs. Weight, Grouped by Model Year')
```



The grouping variable, `Model_Year`, has three unique values, 70, 76, and 82, corresponding to model years 1970, 1976, and 1982.

### Create table and nominal arrays.

Create a table that contains the variables `MPG`, `Weight`, and `Model_Year`. Convert the variable `Model_Year` to a nominal array.

```
cars = table(MPG,Weight,Model_Year);
cars.Model_Year = nominal(cars.Model_Year);
```

### Fit a regression model.

Fit a regression model using `fitlm` with `MPG` as the dependent variable, and `Weight` and `Model_Year` as the independent variables. Because `Model_Year` is a categorical covariate with three levels, it should enter the model as two indicator variables.

The scatter plot suggests that the slope of `MPG` against `Weight` might differ for each model year. To assess this, include weight-year interaction terms.

The proposed model is

$$E(MPG) = \beta_0 + \beta_1 Weight + \beta_2 I[1976] + \beta_3 I[1982] + \beta_4 Weight \times I[1976] + \beta_5 Weight \times I[1982],$$

where  $I[1976]$  and  $I[1982]$  are dummy variables indicating the model years 1976 and 1982, respectively.  $I[1976]$  takes the value 1 if model year is 1976 and takes the value 0 if it is not.  $I[1982]$  takes the value 1 if model year is 1982 and takes the value 0 if it is not. In this model, 1970 is the reference year.

```
fit = fitlm(cars, 'MPG~Weight*Model_Year')
```

```
fit =
```

Linear regression model:

```
MPG ~ 1 + Weight*Model_Year
```

Estimated Coefficients:

	Estimate	SE
	-----	-----
(Intercept)	37.399	2.1466
Weight	-0.0058437	0.00061765
Model_Year_76	4.6903	2.8538
Model_Year_82	21.051	4.157

```

Weight:Model_Year_76    -0.00082009    0.00085468
Weight:Model_Year_82    -0.0050551     0.0015636
    
```

```

                                tStat      pValue
                                -----      -
(Intercept)                    17.423    2.8607e-30
Weight                          -9.4612   4.6077e-15
Model_Year_76                   1.6435    0.10384
Model_Year_82                   5.0641   2.2364e-06
Weight:Model_Year_76            -0.95953  0.33992
Weight:Model_Year_82            -3.2329  0.0017256
    
```

Number of observations: 94, Error degrees of freedom: 88  
 Root Mean Squared Error: 2.79  
 R-squared: 0.886, Adjusted R-Squared 0.88  
 F-statistic vs. constant model: 137, p-value = 5.79e-40

The regression output shows:

- `fitlm` recognizes `Model_Year` as a nominal variable, and constructs the required indicator (dummy) variables. By default, the first level, 70, is the reference group (use `reorderlevels` to change the reference group).
- The model specification, `MPG~Weight*Model_Year`, specifies the first-order terms for `Weight` and `Model_Year`, and all interactions.
- The model  $R^2 = 0.886$ , meaning the variation in miles per gallon is reduced by 88.6% when you consider weight, model year, and their interactions.
- The fitted model is

$$\hat{MPG} = 37.4 - 0.006Weight + 4.7I[1976] + 21.1I[1982] - 0.0008Weight \times I[1976] - 0.005Weight \times I[1982]$$

Thus, the estimated regression equations for the model years are as follows.

Model Year	Predicted MPG Against Weight
1970	$\hat{MPG} = 37.4 - 0.006Weight$
1976	$\hat{MPG} = (37.4 + 4.7) - (0.006 + 0.0008)Weight$
1982	$\hat{MPG} = (37.4 + 21.1) - (0.006 + 0.005)Weight$

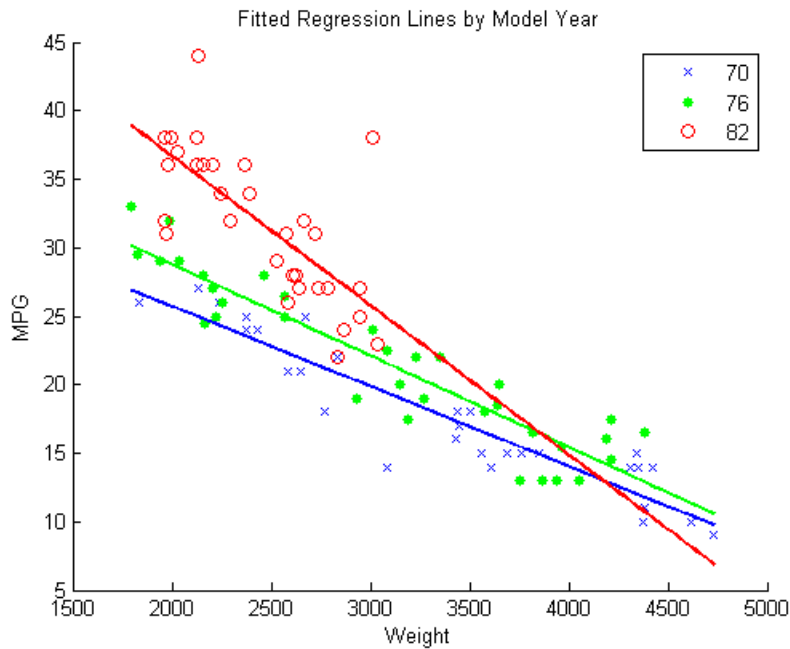
The relationship between MPG and Weight has an increasingly negative slope as the model year increases.

**Plot fitted regression lines.**

Plot the data and fitted regression lines.

```
w = linspace(min(Weight),max(Weight));

figure()
gscatter(Weight,MPG,Model_Year,'bgr','x.o')
line(w,feval(fit,w,'70'),'Color','b','LineWidth',2)
line(w,feval(fit,w,'76'),'Color','g','LineWidth',2)
line(w,feval(fit,w,'82'),'Color','r','LineWidth',2)
title('Fitted Regression Lines by Model Year')
```



**Test for different slopes.**

Test for significant differences between the slopes. This is equivalent to testing the hypothesis

$$H_0 : \beta_4 = \beta_5 = 0$$

$$H_A : \beta_i \neq 0 \text{ for at least one } i.$$

```
anova(fit)
```

```
ans =
```

	SumSq	DF	MeanSq	F	pValue
Weight	2050.2	1	2050.2	263.87	3.2055e-28
Model_Year	807.69	2	403.84	51.976	1.2494e-15
Weight:Model_Year	81.219	2	40.609	5.2266	0.0071637
Error	683.74	88	7.7698		

This output shows that the  $p$ -value for the test is 0.0072 (from the interaction row, `Weight:Model_Year`), so the null hypothesis is rejected at the 0.05 significance level. The value of the test statistic is 5.2266. The numerator degrees of freedom for the test is 2, which is the number of coefficients in the null hypothesis.

There is sufficient evidence that the slopes are not equal for all three model years.

## See Also

`dataset` | `fitlm` | `nominal` | `reorderlevels`

## Related Examples

- “Plot Data Grouped by Category” on page 2-29
- “Test Differences Between Category Means” on page 2-34
- “Summary Statistics Grouped by Category” on page 2-44

## More About

- “Advantages of Using Categorical Arrays” on page 2-50
- “Grouping Variables” on page 2-59
- “Dummy Indicator Variables” on page 2-62

# Create a Dataset Array from Workspace Variables

---

**Note** The dataset data type might be removed in a future release. To work with heterogeneous data, use the MATLAB `table` data type instead. See MATLAB `table` documentation for more information.

---

### In this section...

“Create a Dataset Array from a Numeric Array” on page 2-70

“Create Dataset Array from Heterogeneous Workspace Variables” on page 2-73

## Create a Dataset Array from a Numeric Array

This example shows how to create a dataset array from a numeric array existing in the MATLAB® workspace.

### Load sample data.

```
load fisheriris
```

Two variables load into the workspace: `meas`, a 150-by-4 numeric array, and `species`, a 150-by-1 cell array of species labels.

### Create a dataset array.

Use `mat2dataset` to convert the numeric array, `meas`, into a dataset array.

```
ds = mat2dataset(meas);  
ds(1:10, :)
```

```
ans =
```

```
    meas1    meas2    meas3    meas4  
    5.1      3.5      1.4      0.2  
    4.9      3        1.4      0.2  
    4.7      3.2      1.3      0.2  
    4.6      3.1      1.5      0.2  
     5      3.6      1.4      0.2  
    5.4      3.9      1.7      0.4
```



```
4.6      3.4      1.4      0.3
 5       3.4      1.5      0.2
4.4      2.9      1.4      0.2
4.9      3.1      1.5      0.1
```

The array, `meas`, has four columns, so the dataset array, `ds`, has four variables. The default variable names are the array name, `meas`, with column numbers appended.

You can specify your own variable or observation names using the name-value pair arguments `VarNames` and `ObsNames`, respectively.

If you use `dataset` to convert a numeric array to a dataset array, by default, the resulting dataset array has one variable that is an array instead of separate variables for each column.

### Examine the dataset array.

Return the size of the dataset array, `ds`.

```
size(ds)
```

```
ans =
```

```
150    4
```

The dataset array, `ds`, is the same size as the numeric array, `meas`. Variable names and observation names do not factor into the size of a dataset array.

### Explore dataset array metadata.

Return the metadata properties of the dataset array, `ds`.

```
ds.Properties
```

```
ans =
```

```
struct with fields:
    Description: ''
    VarDescription: {}
```

```
Units: {}
DimNames: {'Observations' 'Variables'}
UserData: []
ObsNames: {}
VarNames: {'meas1' 'meas2' 'meas3' 'meas4'}
```

You can also access the properties individually. For example, you can retrieve the variable names using `ds.Properties.VarNames`.

### Access data in a dataset array variable.

You can use variable names with dot indexing to access the data in a dataset array. For example, find the minimum value in the first variable, `meas1`.

```
min(ds.meas1)
```

```
ans =
```

```
4.3000
```

### Change variable names.

The four variables in `ds` are actually measurements of sepal length, sepal width, petal length, and petal width. Modify the variable names to be more descriptive.

```
ds.Properties.VarNames = {'SLength', 'SWidth', 'PLength', 'PWidth'};
```

### Add description.

you can add a description for the dataset array.

```
ds.Properties.Description = 'Fisher iris data';
ds.Properties
```

```
ans =
```

```
struct with fields:
    Description: 'Fisher iris data'
    VarDescription: {}
    Units: {}
```

```

DimNames: {'Observations' 'Variables'}
UserData: []
ObsNames: {}
VarNames: {'SLength' 'SWidth' 'PLength' 'PWidth'}

```

The dataset array properties are updated with the new variable names and description.

### Add a variable to the dataset array.

The variable `species` is a cell array of species labels. Add `species` to the dataset array, `ds`, as a nominal array named `Species`. Display the first five observations in the dataset array.

```

ds.Species = nominal(species);
ds(1:5,:)

```

```
ans =
```

SLength	SWidth	PLength	PWidth	Species
5.1	3.5	1.4	0.2	setosa
4.9	3	1.4	0.2	setosa
4.7	3.2	1.3	0.2	setosa
4.6	3.1	1.5	0.2	setosa
5	3.6	1.4	0.2	setosa

The dataset array, `ds`, now has the fifth variable, `Species`.

## Create Dataset Array from Heterogeneous Workspace Variables

This example shows how to create a dataset array from heterogeneous variables existing in the MATLAB® workspace.

### Load sample data.

```
load carsmall
```

### Create a dataset array.

Create a dataset array from a subset of the workspace variables.

```
ds = dataset(Origin,Acceleration,Cylinders,MPG);  
ds.Properties.VarNames(:)
```

```
ans =  
  
4x1 cell array  
  
{'Origin'      }  
{'Acceleration'}  
{'Cylinders'   }  
{'MPG'         }
```

When creating the dataset array, you do not need to enter variable names. `dataset` automatically uses the name of each workspace variable.

Notice that the dataset array, `ds`, contains a collection of variables with heterogeneous data types. `Origin` is a character array, and the other variables are numeric.

### Examine a dataset array.

Display the first five observations in the dataset array.

```
ds(1:5,:)

ans =  
  
    Origin    Acceleration    Cylinders    MPG  
    USA         12             8           18  
    USA     11.5             8           15  
    USA         11             8           18  
    USA         12             8           16  
    USA     10.5             8           17
```

### Apply a function to a dataset array.

Use `datasetfun` to return the data type of each variable in `ds`.

```
varclass = datasetfun(@class,ds,'UniformOutput',false);  
varclass(:)
```

```
ans =
```

```
4x1 cell array

    {'char'  }
    {'double'}
    {'double'}
    {'double'}
```

You can get additional information about the variables using `summary(ds)`.

### Modify a dataset array.

`Cylinders` is a numeric variable that has values 4, 6, and 8 for the number of cylinders. Convert `Cylinders` to a nominal array with levels `four`, `six`, and `eight`.

Display the country of origin and number of cylinders for the first 15 cars.

```
ds.Cylinders = nominal(ds.Cylinders,{'four','six','eight'});
ds(1:15,{'Origin','Cylinders'})
```

```
ans =
```

Origin	Cylinders
USA	eight
USA	eight
USA	eight
USA	eight
USA	eight
USA	eight
USA	eight
USA	eight
USA	eight
USA	eight
USA	eight
France	four
USA	eight
USA	eight
USA	eight
USA	eight

The variable `Cylinders` has a new data type.

### See Also

`dataset` | `datasetfun` | `mat2dataset` | `nominal`

### Related Examples

- “Create a Dataset Array from a File” on page 2-77
- “Export Dataset Arrays” on page 2-121
- “Dataset Arrays in the Variables Editor” on page 2-128
- “Index and Search Dataset Arrays” on page 2-145

### More About

- “Dataset Arrays” on page 2-142

## Create a Dataset Array from a File

**Note** The dataset data type might be removed in a future release. To work with heterogeneous data, use the MATLAB `table` data type instead. See MATLAB `table` documentation for more information.

### In this section...

“Create a Dataset Array from a Tab-Delimited Text File” on page 2-77

“Create a Dataset Array from a Comma-Separated Text File” on page 2-80

“Create a Dataset Array from an Excel File” on page 2-82

### Create a Dataset Array from a Tab-Delimited Text File

This example shows how to create a dataset array from the contents of a tab-delimited text file.

#### Create a dataset array using default settings.

Navigate to the folder containing sample data. Import the text file `hospitalSmall.txt` as a dataset array using the default settings.

```
cd(matlabroot)
cd('help/toolbox/stats/examples')
ds = dataset('File','hospitalSmall.txt')

ds =

    name          sex    age    wgt    smoke
    'SMITH'        'm'    38     176    1
    'JOHNSON'      'm'    43     163    0
    'WILLIAMS'     'f'    38     131    0
    'JONES'        'f'    40     133    0
    'BROWN'        'f'    49     119    0
    'DAVIS'        'f'    46     142    0
    'MILLER'       'f'    33     142    1
    'WILSON'       'm'    40     180    0
    'MOORE'        'm'    28     183    0
    'TAYLOR'       'f'    31     132    0
    'ANDERSON'    'f'    45     128    0
```

```
'THOMAS'      'f'      42      137      0
'JACKSON'     'm'      25      174      0
'WHITE'       'm'      39      202      1
```

By default, `dataset` uses the first row of the text file for variable names. If the first row does not contain variable names, you can specify the optional name-value pair argument `'ReadVarNames', false` to change the default behavior.

The dataset array contains heterogeneous variables. The variables `id`, `name`, and `sex` are cell arrays of character vectors, and the other variables are numeric.

### Summarize the dataset array.

You can see the data type and other descriptive statistics for each variable by using `summary` to summarize the dataset array.

```
summary(ds)

name: [14x1 cell array of character vectors]

sex: [14x1 cell array of character vectors]

age: [14x1 double]

      min      1st quartile      median      3rd quartile      max
      25       33             39.5         43             49

wgt: [14x1 double]

      min      1st quartile      median      3rd quartile      max
     119      132             142         176             202

smoke: [14x1 double]

      min      1st quartile      median      3rd quartile      max
       0       0             0           0             1
```

### Import observation names.

Import the text file again, this time specifying that the first column contains observation names.

```
ds = dataset('File', 'hospitalSmall.txt', 'ReadObsNames', true)
```



```
ds =

      sex      age      wgt      smoke
SMITH      'm'      38      176      1
JOHNSON    'm'      43      163      0
WILLIAMS   'f'      38      131      0
JONES      'f'      40      133      0
BROWN      'f'      49      119      0
DAVIS      'f'      46      142      0
MILLER     'f'      33      142      1
WILSON     'm'      40      180      0
MOORE      'm'      28      183      0
TAYLOR     'f'      31      132      0
ANDERSON   'f'      45      128      0
THOMAS     'f'      42      137      0
JACKSON    'm'      25      174      0
WHITE      'm'      39      202      1
```

The elements of the first column in the text file, last names, are now observation names. Observation names and row names are dataset array properties. You can always add or change the observation names of an existing dataset array by modifying the property `ObsNames`.

### Change dataset array properties.

By default, the `DimNames` property of the dataset array has name as the descriptor of the observation (row) dimension. `dataset` got this name from the first row of the first column in the text file.

Change the first element of `DimNames` to `LastName`.

```
ds.Properties.DimNames{1} = 'LastName';
ds.Properties

ans =

      Description: ''
      VarDescription: {}
              Units: {}
      DimNames: {'LastName' 'Variables'}
      UserData: []
      ObsNames: {14x1 cell}
      VarNames: {'sex' 'age' 'wgt' 'smoke'}
```

### Index into dataset array.

You can use observation names to index into a dataset array. For example, return the data for the patient with last name BROWN.

```
ds('BROWN', :)  
  
ans =  
  
          sex      age      wgt      smoke  
BROWN    'f'      49      119      0
```

Note that observation names must be unique.

### Create a Dataset Array from a Comma-Separated Text File

This example shows how to create a dataset array from the contents of a comma-separated text file.

#### Create a dataset array.

Navigate to the folder containing sample data. Import the file `hospitalSmall.csv` as a dataset array, specifying the comma-delimited format.

```
cd(matlabroot)  
cd('help/toolbox/stats/examples')  
ds = dataset('File', 'hospitalSmall.csv', 'Delimiter', ',')  
  
ds =  
  
          id          name          sex      age      wgt      smoke  
'YPL-320'      'SMITH'      'm'      38      176      1  
'GLI-532'      'JOHNSON'    'm'      43      163      0  
'PNI-258'      'WILLIAMS'   'f'      38      131      0  
'MIJ-579'      'JONES'      'f'      40      133      0  
'XLK-030'      'BROWN'      'f'      49      119      0  
'TFP-518'      'DAVIS'      'f'      46      142      0  
'LPD-746'      'MILLER'     'f'      33      142      1  
'ATA-945'      'WILSON'     'm'      40      180      0  
'VNL-702'      'MOORE'      'm'      28      183      0  
'LQW-768'      'TAYLOR'     'f'      31      132      0  
'QFY-472'      'ANDERSON'   'f'      45      128      0  
'UJG-627'      'THOMAS'     'f'      42      137      0
```

```
'XUE-826'      'JACKSON'      'm'      25      174      0
'TRW-072'      'WHITE'        'm'      39      202      1
```

By default, `dataset` uses the first row in the text file as variable names.

### Add observation names.

Use the unique identifiers in the variable `id` as observation names. Then, delete the variable `id` from the dataset array.

```
ds.Properties.ObsNames = ds.id;
ds.id = []
```

```
ds =
```

	name	sex	age	wgt	smoke
YPL-320	'SMITH'	'm'	38	176	1
GLI-532	'JOHNSON'	'm'	43	163	0
PNI-258	'WILLIAMS'	'f'	38	131	0
MIJ-579	'JONES'	'f'	40	133	0
XLK-030	'BROWN'	'f'	49	119	0
TFP-518	'DAVIS'	'f'	46	142	0
LPD-746	'MILLER'	'f'	33	142	1
ATA-945	'WILSON'	'm'	40	180	0
VNL-702	'MOORE'	'm'	28	183	0
LQW-768	'TAYLOR'	'f'	31	132	0
QFY-472	'ANDERSON'	'f'	45	128	0
UJG-627	'THOMAS'	'f'	42	137	0
XUE-826	'JACKSON'	'm'	25	174	0
TRW-072	'WHITE'	'm'	39	202	1

### Delete observations.

Delete any patients with the last name BROWN. You can use `strcmp` to match 'BROWN' with the elements of the variable containing last names, `name`.

```
toDelete = strcmp(ds.name, 'BROWN');
ds(toDelete,:) = []
```

```
ds =
```

	name	sex	age	wgt	smoke
YPL-320	'SMITH'	'm'	38	176	1
GLI-532	'JOHNSON'	'm'	43	163	0
PNI-258	'WILLIAMS'	'f'	38	131	0

MIJ-579	'JONES'	'f'	40	133	0
TFP-518	'DAVIS'	'f'	46	142	0
LPD-746	'MILLER'	'f'	33	142	1
ATA-945	'WILSON'	'm'	40	180	0
VNL-702	'MOORE'	'm'	28	183	0
LQW-768	'TAYLOR'	'f'	31	132	0
QFY-472	'ANDERSON'	'f'	45	128	0
UJG-627	'THOMAS'	'f'	42	137	0
XUE-826	'JACKSON'	'm'	25	174	0
TRW-072	'WHITE'	'm'	39	202	1

One patient having last name BROWN is deleted from the dataset array.

### Return size of dataset array.

The array now has 13 observations.

```
size(ds)
ans =
    13     5
```

Note that the row and column corresponding to variable and observation names, respectively, are not included in the size of a dataset array.

## Create a Dataset Array from an Excel File

This example shows how to create a dataset array from the contents of an Excel® spreadsheet file.

### Create a dataset array.

Navigate to the folder containing sample data. Import the data from the first worksheet in the file `hospitalSmall.xlsx`, specifying that the data file is an Excel spreadsheet.

```
cd(matlabroot)
cd('help/toolbox/stats/examples')
ds = dataset('XLSFile','hospitalSmall.xlsx')

ds =

    id          name          sex      age      wgt      smoke
    'YPL-320'   'SMITH'      'm'      38      176      1
```

'GLI-532'	'JOHNSON'	'm'	43	163	0
'PNI-258'	'WILLIAMS'	'f'	38	131	0
'MIJ-579'	'JONES'	'f'	40	133	0
'XLK-030'	'BROWN'	'f'	49	119	0
'TFP-518'	'DAVIS'	'f'	46	142	0
'LPD-746'	'MILLER'	'f'	33	142	1
'ATA-945'	'WILSON'	'm'	40	180	0
'VNL-702'	'MOORE'	'm'	28	183	0
'LQW-768'	'TAYLOR'	'f'	31	132	0
'QFY-472'	'ANDERSON'	'f'	45	128	0
'UJG-627'	'THOMAS'	'f'	42	137	0
'XUE-826'	'JACKSON'	'm'	25	174	0
'TRW-072'	'WHITE'	'm'	39	202	1

By default, `dataset` creates variable names using the contents of the first row in the spreadsheet.

### Specify which worksheet to import.

Import the data from the second worksheet into a new dataset array.

```
ds2 = dataset('XLSFile', 'hospitalSmall.xlsx', 'Sheet', 2)
```

```
ds2 =
```

id	name	sex	age	wgt	smoke
'TRW-072'	'WHITE'	'm'	39	202	1
'ELG-976'	'HARRIS'	'f'	36	129	0
'KOQ-996'	'MARTIN'	'm'	48	181	1
'YUZ-646'	'THOMPSON'	'm'	32	191	1
'XBR-291'	'GARCIA'	'f'	27	131	1
'KPW-846'	'MARTINEZ'	'm'	37	179	0
'XBA-581'	'ROBINSON'	'm'	50	172	0
'BKD-785'	'CLARK'	'f'	48	133	0

## See Also

`dataset` | `summary`

## Related Examples

- “Create a Dataset Array from Workspace Variables” on page 2-70

- “Clean Messy and Missing Data” on page 2-123
- “Export Dataset Arrays” on page 2-121
- “Dataset Arrays in the Variables Editor” on page 2-128
- “Index and Search Dataset Arrays” on page 2-145

### **More About**

- “Dataset Arrays” on page 2-142

## Add and Delete Observations

This example shows how to add and delete observations in a dataset array. You can also edit dataset arrays using the Variables editor.

### Load sample data.

Navigate to the folder containing sample data. Import the data from the first worksheet in `hospitalSmall.xlsx` into a dataset array.

```
cd(matlabroot)
cd('help/toolbox/stats/examples')
ds = dataset('XLSFile','hospitalSmall.xlsx');
size(ds)

ans =

    14     6
```

The dataset array, `ds`, has 14 observations (rows) and 6 variables (columns).

### Add observations by concatenation.

The second worksheet in `hospitalSmall.xlsx` has additional patient data. Append the observations in this spreadsheet to the end of `ds`.

```
ds2 = dataset('XLSFile','hospitalSmall.xlsx','Sheet',2);
dsNew = [ds;ds2];
size(dsNew)

ans =

    22     6
```

The dataset array `dsNew` has 22 observations. In order to vertically concatenate two dataset arrays, both arrays must have the same number of variables, with the same variable names.

### Add observations from a cell array.

If you want to append new observations stored in a cell array, first convert the cell array to a dataset array, and then concatenate the dataset arrays.

```
cellObs = {'id','name','sex','age','wgt','smoke';
           'YQR-965','BAKER','M',36,160,0;
```

```
        'LFG-497', 'WALL' , 'F', 28, 125, 1;  
        'KSD-003', 'REED' , 'M', 32, 187, 0};  
dsNew = [dsNew; cell2dataset(cellObs)];  
size(dsNew)  
  
ans =  
  
    25     6
```

### Add observations from a structure.

You can also append new observations stored in a structure. Convert the structure to a dataset array, and then concatenate the dataset arrays.

```
structObs(1,1).id = 'GHK-842';  
structObs(1,1).name = 'GEORGE';  
structObs(1,1).sex = 'M';  
structObs(1,1).age = 45;  
structObs(1,1).wgt = 182;  
structObs(1,1).smoke = 1;  
  
structObs(2,1).id = 'QRH-308';  
structObs(2,1).name = 'BAILEY';  
structObs(2,1).sex = 'F';  
structObs(2,1).age = 29;  
structObs(2,1).wgt = 120;  
structObs(2,1).smoke = 0;  
  
dsNew = [dsNew; struct2dataset(structObs)];  
size(dsNew)  
  
ans =  
  
    27     6
```

### Delete duplicate observations.

Use `unique` to delete any observations in a dataset array that are duplicated.

```
dsNew = unique(dsNew);  
size(dsNew)  
  
ans =  
  
    21     6
```



One duplicated observation is deleted.

### Delete observations by observation number.

Delete observations 18, 20, and 21 from the dataset array.

```
dsNew([18,20,21],:) = [];
size(dsNew)
```

```
ans =
    18     6
```

The dataset array has only 18 observations now.

### Delete observations by observation name.

First, specify the variable of identifiers, `id`, as observation names. Then, delete the variable `id` from `dsNew`. You can use the observation name to index observations.

```
dsNew.Properties.ObsNames = dsNew.id;
dsNew.id = [];
dsNew('KOQ-996',:) = [];
size(dsNew)
```

```
ans =
    17     5
```

The dataset array now has one less observation and one less variable.

### Search for observations to delete.

You can also search for observations in the dataset array. For example, delete observations for any patients with the last name `WILLIAMS`.

```
toDelete = strcmp(dsNew.name, 'WILLIAMS');
dsNew(toDelete,:) = [];
size(dsNew)
```

```
ans =
    16     5
```

The dataset array now has one less observation.

### See Also

`cell2dataset` | `dataset` | `struct2dataset`

### Related Examples

- “Add and Delete Variables” on page 2-89
- “Select Subsets of Observations” on page 2-99
- “Dataset Arrays in the Variables Editor” on page 2-128
- “Index and Search Dataset Arrays” on page 2-145

### More About

- “Dataset Arrays” on page 2-142

## Add and Delete Variables

This example shows how to add and delete variables in a dataset array. You can also edit dataset arrays using the Variables editor.

### Load sample data.

Navigate to the folder containing sample data.

```
cd(matlabroot)
cd('help/toolbox/stats/examples')
```

Import the data from the first worksheet in `hospitalSmall.xlsx` into a dataset array.

```
ds = dataset('XLSFile', 'hospitalSmall.xlsx');
size(ds)
```

```
ans =
     14     6
```

The dataset array, `ds`, has 14 observations (rows) and 6 variables (columns).

### Add variables by concatenating dataset arrays.

The worksheet `Heights` in `hospitalSmall.xlsx` has heights for the patients on the first worksheet. Concatenate the data in this spreadsheet with `ds`.

```
ds2 = dataset('XLSFile', 'hospitalSmall.xlsx', 'Sheet', 'Heights');
ds = [ds ds2];
size(ds)
```

```
ans =
     14     7
```

The dataset array now has seven variables. You can only horizontally concatenate dataset arrays with observations in the same position, or with the same observation names.

```
ds.Properties.VarNames{end}
ans =
hgt
```

The name of the last variable in `ds` is `hgt`, which dataset read from the first row of the imported spreadsheet.

### Delete variables by variable name.

First, specify the unique identifiers in the variable `id` as observation names. Then, delete the variable `id` from the dataset array.

```
ds.Properties.ObsNames = ds.id;
ds.id = [];
size(ds)

ans =

    14     6
```

The dataset array now has six variables. List the variable names.

```
ds.Properties.VarNames(:)

ans =

    'name'
    'sex'
    'age'
    'wgt'
    'smoke'
    'hgt'
```

There is no longer a variable called `id`.

### Add a new variable by name.

Add a new variable, `bmi`—which contains the body mass index (BMI) for each patient—to the dataset array. BMI is a function of height and weight. Display the last name, gender, and BMI for each patient.

```
ds.bmi = ds.wgt*703./ds.hgt.^2;
ds(:, {'name', 'sex', 'bmi'})

ans =

    name          sex          bmi
    YPL-320      'SMITH'      'm'          24.544
    GLI-532      'JOHNSON'    'm'          24.068
```

PNI-258	'WILLIAMS'	'f'	23.958
MIJ-579	'JONES'	'f'	25.127
XLK-030	'BROWN'	'f'	21.078
TFP-518	'DAVIS'	'f'	27.729
LPD-746	'MILLER'	'f'	26.828
ATA-945	'WILSON'	'm'	24.41
VNL-702	'MOORE'	'm'	27.822
LQW-768	'TAYLOR'	'f'	22.655
QFY-472	'ANDERSON'	'f'	23.409
UJG-627	'THOMAS'	'f'	25.883
XUE-826	'JACKSON'	'm'	24.265
TRW-072	'WHITE'	'm'	29.827

The operators `./` and `.^` in the calculation of BMI indicate element-wise division and exponentiation, respectively.

### Delete variables by variable number.

Delete the variable `wgt`, the fourth variable in the dataset array.

```
ds(:,4) = [];
ds.Properties.VarNames(:)

ans =

    'name'
    'sex'
    'age'
    'smoke'
    'hgt'
    'bmi'
```

The variable `wgt` is deleted from the dataset array.

## See Also

`dataset`

## Related Examples

- “Add and Delete Observations” on page 2-85
- “Merge Dataset Arrays” on page 2-107

- “Calculations on Dataset Arrays” on page 2-117
- “Dataset Arrays in the Variables Editor” on page 2-128
- “Index and Search Dataset Arrays” on page 2-145

### **More About**

- “Dataset Arrays” on page 2-142

## Access Data in Dataset Array Variables

This example shows how to work with dataset array variables and their data.

### Access variables by name.

You can access variable data, or select a subset of variables, by using variable (column) names and dot indexing. Load a sample dataset array. Display the names of the variables in `hospital`.

```
load hospital
hospital.Properties.VarNames(:)
```

```
ans =
```

```
7x1 cell array

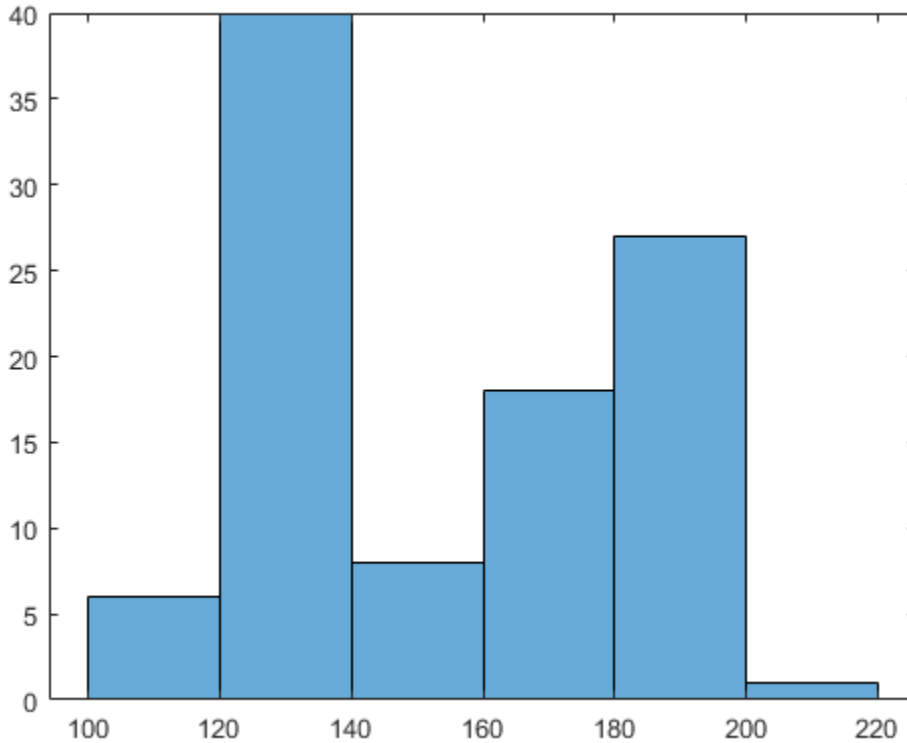
    {'LastName'   }
    {'Sex'        }
    {'Age'        }
    {'Weight'     }
    {'Smoker'     }
    {'BloodPressure'}
    {'Trials'     }
```

The dataset array has 7 variables (columns) and 100 observations (rows). You can double-click `hospital` in the Workspace window to view the dataset array in the Variables editor.

### Plot histogram.

Plot a histogram of the data in the variable `Weight`.

```
figure
histogram(hospital.Weight)
```



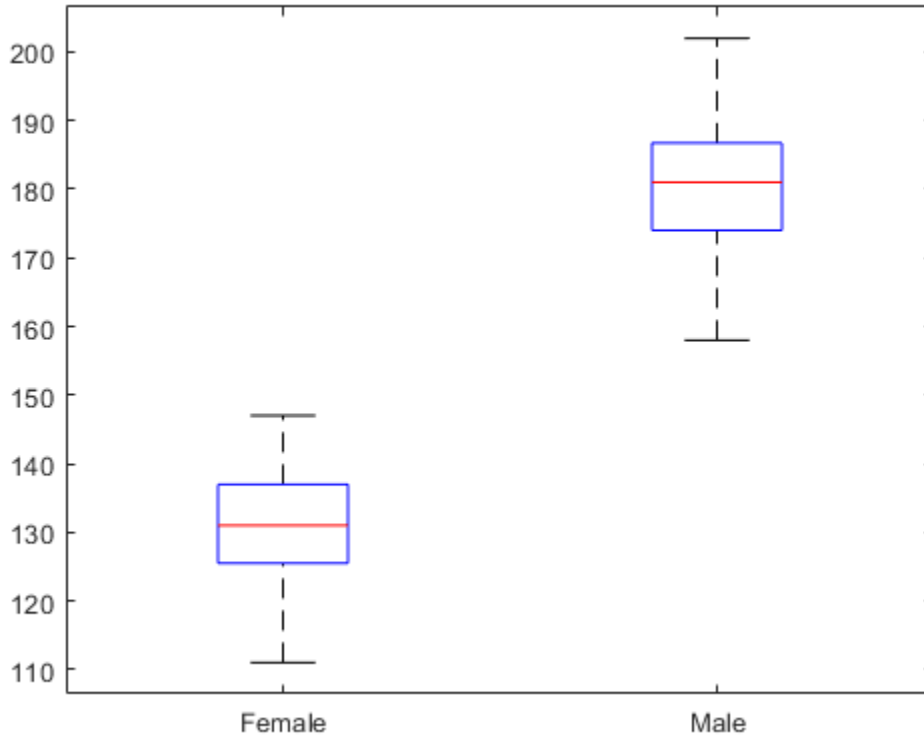
The histogram shows that the weight distribution is bimodal.

**Plot data grouped by category.**

Draw box plots of `Weight` grouped by the values in `Sex` (Male and Female). That is, use the variable `Sex` as a grouping variable.

```
figure  
boxplot(hospital.Weight, hospital.Sex)
```





The box plot suggests that gender accounts for the bimodality in weight.

### Select a subset of variables.

Create a new dataset array with only the variables `LastName`, `Sex`, and `Weight`. You can access the variables by name or column number.

```
ds1 = hospital(:, {'LastName', 'Sex', 'Weight'});
ds2 = hospital(:, [1, 2, 4]);
```

The dataset arrays `ds1` and `ds2` are equivalent. Use parentheses ( ) when indexing dataset arrays to preserve the data type; that is, to create a dataset array from a subset of a dataset array. You can also use the Variables editor to create a new dataset array from a subset of variables and observations.

### Convert the variable data type.

Convert the data type of the variable `Smoker` from logical to nominal with labels `No` and `Yes`.

```
hospital.Smoker = nominal(hospital.Smoker, {'No', 'Yes'});  
class(hospital.Smoker)
```

```
ans =  
  
    'nominal'
```

### Explore data.

Display the first 10 elements of `Smoker`.

```
hospital.Smoker(1:10)
```

```
ans =  
  
    10x1 nominal array  
  
    Yes  
    No  
    No  
    No  
    No  
    No  
    Yes  
    No  
    No  
    No
```

If you want to change the level labels in a nominal array, use `setlabels`.

### Add variables.

The variable `BloodPressure` is a 100-by-2 array. The first column corresponds to systolic blood pressure, and the second column to diastolic blood pressure. Separate this array into two new variables, `SysPressure` and `DiaPressure`.

```

hospital.SysPressure = hospital.BloodPressure(:,1);
hospital.DiaPressure = hospital.BloodPressure(:,2);
hospital.Properties.VarNames(:)

```

```
ans =
```

```

9x1 cell array

    {'LastName'    }
    {'Sex'         }
    {'Age'         }
    {'Weight'      }
    {'Smoker'      }
    {'BloodPressure'}
    {'Trials'      }
    {'SysPressure' }
    {'DiaPressure' }

```

The dataset array, `hospital`, has two new variables.

### Search for variables by name.

Use `regexp` to find variables in `hospital` with 'Pressure' in their name. Create a new dataset array containing only these variables.

```

bp = regexp(hospital.Properties.VarNames, 'Pressure');
bpIdx = cellfun(@isempty, bp);
bpData = hospital(:, ~bpIdx);
bpData.Properties.VarNames(:)

```

```
ans =
```

```

3x1 cell array

    {'BloodPressure'}
    {'SysPressure'   }
    {'DiaPressure'   }

```

The new dataset array, `bpData`, contains only the blood pressure variables.

### Delete variables.

Delete the variable `BloodPressure` from the dataset array, `hospital`.

```
hospital.BloodPressure = [];  
hospital.Properties.VarNames(:)
```

```
ans =
```

```
8x1 cell array  
  
    {'LastName' }  
    {'Sex'      }  
    {'Age'     }  
    {'Weight'  }  
    {'Smoker'  }  
    {'Trials'  }  
    {'SysPressure'}  
    {'DiaPressure'}
```

The variable `BloodPressure` is no longer in the dataset array.

## See Also

`dataset`

### Related Examples

- “Add and Delete Variables” on page 2-89
- “Calculations on Dataset Arrays” on page 2-117
- “Dataset Arrays in the Variables Editor” on page 2-128
- “Index and Search Dataset Arrays” on page 2-145

### More About

- “Dataset Arrays” on page 2-142
- “Grouping Variables” on page 2-59

## Select Subsets of Observations

This example shows how to select an observation or subset of observations from a dataset array.

### Load sample data.

Load the sample dataset array, `hospital`. Dataset arrays can have observation (row) names. This array has observation names corresponding to unique patient identifiers.

```
load hospital
hospital.Properties.ObsNames(1:10)
```

```
ans =
```

```
10x1 cell array
```

```
{'YPL-320'}
{'GLI-532'}
{'PNI-258'}
{'MIJ-579'}
{'XLK-030'}
{'TFP-518'}
{'LPD-746'}
{'ATA-945'}
{'VNL-702'}
{'LQW-768'}
```

These are the first 10 observation names.

### Index an observation by name.

You can use the observation names to index into the dataset array. For example, extract the last name, sex, and age for the patient with identifier `XLK-030`.

```
hospital('XLK-030', {'LastName', 'Sex', 'Age'})
```

```
ans =
```

	LastName	Sex	Age
XLK-030	'BROWN'	Female	49

### Index a subset of observations by number.

Create a new dataset array containing the first 50 patients.

```
ds50 = hospital(1:50, :);  
size(ds50)
```

```
ans =  
  
    50     7
```

### Search observations using a logical condition.

Create a new dataset array containing only male patients. To find the male patients, use a logical condition to search the variable containing gender information.

```
dsMale = hospital(hospital.Sex=='Male', :);  
dsMale(1:10, {'LastName', 'Sex'})
```

```
ans =  
  
    LastName    Sex  
    YPL-320    'SMITH'    Male  
    GLI-532    'JOHNSON'  Male  
    ATA-945    'WILSON'   Male  
    VNL-702    'MOORE'    Male  
    XUE-826    'JACKSON'  Male  
    TRW-072    'WHITE'    Male  
    KOQ-996    'MARTIN'   Male  
    YUZ-646    'THOMPSON' Male  
    KPW-846    'MARTINEZ' Male  
    XBA-581    'ROBINSON' Male
```

### Search observations using multiple conditions.

You can use multiple conditions to search the dataset array. For example, create a new dataset array containing only female patients older than 40.

```
dsFemale = hospital(hospital.Sex=='Female' & hospital.Age > 40, :);  
dsFemale(1:10, {'LastName', 'Sex', 'Age'})
```

```
ans =
```

	LastName	Sex	Age
XLK-030	'BROWN'	Female	49
TFP-518	'DAVIS'	Female	46
QFY-472	'ANDERSON'	Female	45
UJG-627	'THOMAS'	Female	42
BKD-785	'CLARK'	Female	48
VWL-936	'LEWIS'	Female	41
AAX-056	'LEE'	Female	44
AFK-336	'WRIGHT'	Female	45
KKL-155	'ADAMS'	Female	48
RBA-579	'SANCHEZ'	Female	44

### Select a random subset of observations.

Create a new dataset array containing a random subset of 20 patients from the dataset array `hospital`.

```
rng('default') % For reproducibility
dsRandom = hospital(randsample(length(hospital),20),:);
dsRandom.Properties.ObsNames
```

```
ans =
```

20x1 cell array

```
{'DAU-529'}
{'AGR-528'}
{'RBO-332'}
{'QOO-305'}
{'RVS-253'}
{'QEQ-082'}
{'EHE-616'}
{'HVR-372'}
{'KOQ-996'}
{'REV-997'}
{'PUE-347'}
{'LQW-768'}
{'YLN-495'}
{'HJQ-495'}
{'ELG-976'}
```

```
{ 'XUE-826' }  
{ 'MEZ-469' }  
{ 'UDS-151' }  
{ 'MIJ-579' }  
{ 'DGC-290' }
```

### Delete observations by name.

Delete the data for the patient with observation name HVR-372.

```
hospital('HVR-372', :) = [];  
size(hospital)
```

```
ans =  
    99     7
```

The dataset array has one less observation.

## See Also

dataset

### Related Examples

- “Add and Delete Observations” on page 2-85
- “Clean Messy and Missing Data” on page 2-123
- “Dataset Arrays in the Variables Editor” on page 2-128
- “Sort Observations in Dataset Arrays” on page 2-103
- “Index and Search Dataset Arrays” on page 2-145

### More About

- “Dataset Arrays” on page 2-142



## Sort Observations in Dataset Arrays

This example shows how to sort observations (rows) in a dataset array using the command line. You can also sort rows using the Variables editor.

### Sort observations in ascending order.

Load the sample dataset array, `hospital`. Sort the observations by the values in `Age`, in ascending order.

```
load hospital
dsAgeUp = sortrows(hospital, 'Age');
dsAgeUp(1:10, {'LastName', 'Age'})
```

ans =

	LastName	Age
XUE-826	'JACKSON'	25
FZR-250	'HALL'	25
PUE-347	'YOUNG'	25
LIM-480	'HILL'	25
SCQ-914	'JAMES'	25
REV-997	'ALEXANDER'	25
XBR-291	'GARCIA'	27
VNL-702	'MOORE'	28
DTT-578	'WALKER'	28
XAX-646	'COOPER'	28

The youngest patients are age 25.

### Sort observations in descending order.

Sort the observations by `Age` in descending order.

```
dsAgeDown = sortrows(hospital, 'Age', 'descend');
dsAgeDown(1:10, {'LastName', 'Age'})
```

ans =

	LastName	Age
XBA-581	'ROBINSON'	50

DAU-529	'REED'	50
XLK-030	'BROWN'	49
FLJ-908	'STEWART'	49
GGU-691	'HUGHES'	49
MEZ-469	'GRIFFIN'	49
KOQ-996	'MARTIN'	48
BKD-785	'CLARK'	48
KKL-155	'ADAMS'	48
NSK-403	'RAMIREZ'	48

The oldest patients are age 50.

### Sort observations by the values of two variables.

Sort the observations in `hospital` by `Age`, and then by `LastName`.

```
dsName = sortrows(hospital, {'Age', 'LastName'});  
dsName(1:10, {'LastName', 'Age'})
```

ans =

	LastName	Age
REV-997	'ALEXANDER'	25
FZR-250	'HALL'	25
LIM-480	'HILL'	25
XUE-826	'JACKSON'	25
SCQ-914	'JAMES'	25
PUE-347	'YOUNG'	25
XBR-291	'GARCIA'	27
XAX-646	'COOPER'	28
QEQ-082	'COX'	28
NSU-424	'JENKINS'	28

Now the names are sorted alphabetically within increasing age groups.

### Sort observations in mixed order.

Sort the observations in `hospital` by `Age` in an increasing order, and then by `Weight` in a decreasing order.

```
dsWeight = sortrows(hospital, {'Age', 'Weight'}, {'ascend', 'descend'});  
dsWeight(1:10, {'LastName', 'Age', 'Weight'})
```

```
ans =
```

	LastName	Age	Weight
FZR-250	'HALL'	25	189
SCQ-914	'JAMES'	25	186
XUE-826	'JACKSON'	25	174
REV-997	'ALEXANDER'	25	171
LIM-480	'HILL'	25	138
PUE-347	'YOUNG'	25	114
XBR-291	'GARCIA'	27	131
NSU-424	'JENKINS'	28	189
VNL-702	'MOORE'	28	183
XAX-646	'COOPER'	28	127

This shows that the maximum weight among patients that are age 25 is 189 lbs.

### Sort observations by observation name.

Sort the observations in `hospital` by the observation names.

```
dsObs = sortrows(hospital, 'obsnames');
dsObs(1:10, {'LastName', 'Age'})
```

```
ans =
```

	LastName	Age
AAX-056	'LEE'	44
AFB-271	'PEREZ'	44
AFK-336	'WRIGHT'	45
AGR-528	'SIMMONS'	45
ATA-945	'WILSON'	40
BEZ-311	'DIAZ'	45
BKD-785	'CLARK'	48
DAU-529	'REED'	50
DGC-290	'BUTLER'	38
DTT-578	'WALKER'	28

The observations are sorted by observation name in ascending alphabetical order.

### See Also

dataset | sortrows

### Related Examples

- “Select Subsets of Observations” on page 2-99
- “Stack or Unstack Dataset Arrays” on page 2-112
- “Dataset Arrays in the Variables Editor” on page 2-128
- “Index and Search Dataset Arrays” on page 2-145

### More About

- “Dataset Arrays” on page 2-142

## Merge Dataset Arrays

This example shows how to merge dataset arrays using `join`.

### Load sample data.

Navigate to a folder containing sample data. Import the data from the first worksheet in `hospitalSmall.xlsx` into a dataset array, then keep only a few of the variables.

```
cd(matlabroot)
cd('help/toolbox/stats/examples')
ds1 = dataset('XLSFile','hospitalSmall.xlsx');
ds1 = ds1(:,{'id','name','sex','age'})

ds1 =

    id          name          sex          age
    'YPL-320'    'SMITH'          'm'          38
    'GLI-532'    'JOHNSON'        'm'          43
    'PNI-258'    'WILLIAMS'       'f'          38
    'MIJ-579'    'JONES'          'f'          40
    'XLK-030'    'BROWN'          'f'          49
    'TFP-518'    'DAVIS'          'f'          46
    'LPD-746'    'MILLER'         'f'          33
    'ATA-945'    'WILSON'         'm'          40
    'VNL-702'    'MOORE'          'm'          28
    'LQW-768'    'TAYLOR'         'f'          31
    'QFY-472'    'ANDERSON'       'f'          45
    'UJG-627'    'THOMAS'         'f'          42
    'XUE-826'    'JACKSON'        'm'          25
    'TRW-072'    'WHITE'          'm'          39
```

The dataset array, `ds1`, has 14 observations (rows) and 4 variables (columns).

Import the data from the worksheet `Heights2` in `hospitalSmall.xlsx`.

```
ds2 = dataset('XLSFile','hospitalSmall.xlsx','Sheet','Heights2')

ds2 =

    id          hgt
    'LPD-746'    61
    'PNI-258'    62
    'XUE-826'    71
```

```
'ATA-945'      72
'XLK-030'      63
```

ds2 has height measurements for a subset of five individuals from the first dataset array, ds1.

### Merge only the matching subset of observations.

Use `join` to merge the two dataset arrays, ds1 and ds2, keeping only the subset of observations that are in ds2.

```
JoinSmall = join(ds2, ds1)
```

```
JoinSmall =
```

```
   id      hgt  name      sex  age
'LPD-746'   61  'MILLER'  'f'  33
'PNI-258'   62  'WILLIAMS' 'f'  38
'XUE-826'   71  'JACKSON'  'm'  25
'ATA-945'   72  'WILSON'  'm'  40
'XLK-030'   63  'BROWN'   'f'  49
```

In `JoinSmall`, the variable `id` only appears once. This is because it is the key variable—the variable that links observations between the two dataset arrays—and has the same variable name in both ds1 and ds2.

### Include incomplete observations in the merge.

Merge ds1 and ds2 keeping all observations in the larger ds1.

```
joinAll = join(ds2, ds1, 'type', 'rightouter', 'mergekeys', true)
```

```
joinAll =
```

```
   id      hgt  name      sex  age
'ATA-945'   72  'WILSON'  'm'  40
'GLI-532'   NaN  'JOHNSON' 'm'  43
'LPD-746'   61  'MILLER'  'f'  33
'LQW-768'   NaN  'TAYLOR'  'f'  31
'MIJ-579'   NaN  'JONES'   'f'  40
'PNI-258'   62  'WILLIAMS' 'f'  38
'QFY-472'   NaN  'ANDERSON' 'f'  45
'TFP-518'   NaN  'DAVIS'   'f'  46
'TRW-072'   NaN  'WHITE'   'm'  39
```

'UJG-627'	NaN	'THOMAS'	'f'	42
'VNL-702'	NaN	'MOORE'	'm'	28
'XLK-030'	63	'BROWN'	'f'	49
'XUE-826'	71	'JACKSON'	'm'	25
'YPL-320'	NaN	'SMITH'	'm'	38

Each observation in `ds1` without corresponding height measurements in `ds2` has height value `NaN`. Also, because there is no `id` value in `ds2` for each observation in `ds1`, you need to merge the keys using the option `'MergeKeys', true`. This merges the key variable, `id`.

### Merge dataset arrays with different key variable names.

When using `join`, it is not necessary for the key variable to have the same name in the dataset arrays to be merged. Import the data from the worksheet named `Heights3` in `hospitalSmall.xlsx`.

```
ds3 = dataset('XLSFile', 'hospitalSmall.xlsx', 'Sheet', 'Heights3')
```

```
ds3 =
```

identifier	hgt
'GLI-532'	69
'QFY-472'	62
'MIJ-579'	61
'VNL-702'	68
'XLK-030'	63
'LPD-746'	61
'TFP-518'	60
'YPL-320'	71
'ATA-945'	72
'LQW-768'	64
'PNI-258'	62
'UJG-627'	61
'XUE-826'	71
'TRW-072'	69

`ds3` has height measurements for each observation in `ds1`. This dataset array has the same patient identifiers as `ds1`, but they are under the variable name `identifier`, instead of `id` (and in a different order).

### Specify key variable.

You can easily change the variable name of the key variable in `ds3` by setting `ds3.Properties.VarNames` or using the Variables editor, but it is not required to perform a merge. Instead, you can specify the name of the key variable in each dataset array using `LeftKeys` and `RightKeys`.

```
joinDiff = join(ds3,ds1, 'LeftKeys', 'identifier', 'RightKeys', 'id')
```

```
joinDiff =
```

identifier	hgt	name	sex	age
'GLI-532'	69	'JOHNSON'	'm'	43
'QFY-472'	62	'ANDERSON'	'f'	45
'MIJ-579'	61	'JONES'	'f'	40
'VNL-702'	68	'MOORE'	'm'	28
'XLK-030'	63	'BROWN'	'f'	49
'LPD-746'	61	'MILLER'	'f'	33
'TFP-518'	60	'DAVIS'	'f'	46
'YPL-320'	71	'SMITH'	'm'	38
'ATA-945'	72	'WILSON'	'm'	40
'LQW-768'	64	'TAYLOR'	'f'	31
'PNI-258'	62	'WILLIAMS'	'f'	38
'UJG-627'	61	'THOMAS'	'f'	42
'XUE-826'	71	'JACKSON'	'm'	25
'TRW-072'	69	'WHITE'	'm'	39

The merged dataset array, `joinDiff`, has the same key variable order and name as the first dataset array input to `join`, `ds3`.

## See Also

[dataset](#) | [join](#)

## Related Examples

- “Add and Delete Variables” on page 2-89
- “Stack or Unstack Dataset Arrays” on page 2-112
- “Dataset Arrays in the Variables Editor” on page 2-128
- “Index and Search Dataset Arrays” on page 2-145



## More About

- “Dataset Arrays” on page 2-142

## Stack or Unstack Dataset Arrays

This example shows how to reformat dataset arrays using `stack` and `unstack`.

### Load sample data.

Navigate to the folder containing sample data. Import the data from the comma-separated text file `testScores.csv`.

```
cd(matlabroot)
cd('help/toolbox/stats/examples')
ds = dataset('File','testScores.csv','Delimiter','(',')')

ds =

    LastName          Sex          Test1    Test2    Test3    Test4
    'HOWARD'         'male'        90      87      93      92
    'WARD'           'male'        87      85      83      90
    'TORRES'         'male'        86      85      88      86
    'PETERSON'       'female'      75      80      72      77
    'GRAY'           'female'      89      86      87      90
    'RAMIREZ'        'female'      96      92      98      95
    'JAMES'          'male'        78      75      77      77
    'WATSON'         'female'      91      94      92      90
    'BROOKS'        'female'      86      83      85      89
    'KELLY'          'male'        79      76      82      80
```

Each of the 10 students has 4 test scores.

### Perform calculations on dataset array.

With the data in this format, you can, for example, calculate the average test score for each student. The test scores are in columns 3 to 6.

```
ds.TestAve = mean(double(ds(:,3:6)),2);
ds(:,{'LastName','Sex','TestAve'})

ans =

    LastName          Sex          TestAve
    'HOWARD'         'male'        90.5
    'WARD'           'male'        86.25
    'TORRES'         'male'        86.25
    'PETERSON'       'female'      76
```

'GRAY'	'female'	88
'RAMIREZ'	'female'	95.25
'JAMES'	'male'	76.75
'WATSON'	'female'	91.75
'BROOKS'	'female'	85.75
'KELLY'	'male'	79.25

A new variable with average test scores is added to the dataset array, ds.

### Reformat the dataset array.

Stack the test score variables into a new variable, Scores.

```
dsNew = stack(ds, {'Test1', 'Test2', 'Test3', 'Test4'}, ...
              'newDataVarName', 'Scores')
```

dsNew =

LastName	Sex	TestAve	Scores_Indicator	Scores
'HOWARD'	'male'	90.5	Test1	90
'HOWARD'	'male'	90.5	Test2	87
'HOWARD'	'male'	90.5	Test3	93
'HOWARD'	'male'	90.5	Test4	92
'WARD'	'male'	86.25	Test1	87
'WARD'	'male'	86.25	Test2	85
'WARD'	'male'	86.25	Test3	83
'WARD'	'male'	86.25	Test4	90
'TORRES'	'male'	86.25	Test1	86
'TORRES'	'male'	86.25	Test2	85
'TORRES'	'male'	86.25	Test3	88
'TORRES'	'male'	86.25	Test4	86
'PETERSON'	'female'	76	Test1	75
'PETERSON'	'female'	76	Test2	80
'PETERSON'	'female'	76	Test3	72
'PETERSON'	'female'	76	Test4	77
'GRAY'	'female'	88	Test1	89
'GRAY'	'female'	88	Test2	86
'GRAY'	'female'	88	Test3	87
'GRAY'	'female'	88	Test4	90
'RAMIREZ'	'female'	95.25	Test1	96
'RAMIREZ'	'female'	95.25	Test2	92
'RAMIREZ'	'female'	95.25	Test3	98
'RAMIREZ'	'female'	95.25	Test4	95
'JAMES'	'male'	76.75	Test1	78
'JAMES'	'male'	76.75	Test2	75

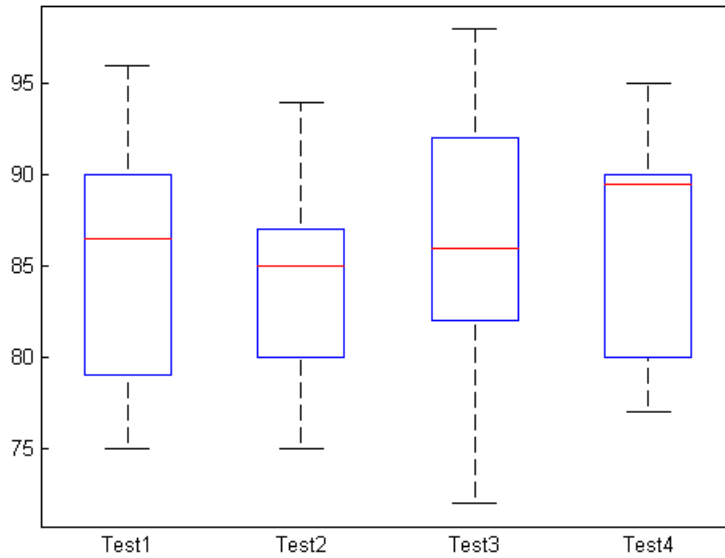
' JAMES '	' male '	76.75	Test3	77
' JAMES '	' male '	76.75	Test4	77
' WATSON '	' female '	91.75	Test1	91
' WATSON '	' female '	91.75	Test2	94
' WATSON '	' female '	91.75	Test3	92
' WATSON '	' female '	91.75	Test4	90
' BROOKS '	' female '	85.75	Test1	86
' BROOKS '	' female '	85.75	Test2	83
' BROOKS '	' female '	85.75	Test3	85
' BROOKS '	' female '	85.75	Test4	89
' KELLY '	' male '	79.25	Test1	79
' KELLY '	' male '	79.25	Test2	76
' KELLY '	' male '	79.25	Test3	82
' KELLY '	' male '	79.25	Test4	80

The original test variable names, `Test1`, `Test2`, `Test3`, and `Test4`, appear as levels in the combined test scores indicator variable, `Scores_Indicator`.

### Plot data grouped by category.

With the data in this format, you can use `Scores_Indicator` as a grouping variable, and draw box plots of test scores grouped by test.

```
figure()  
boxplot(dsNew.Scores, dsNew.Scores_Indicator)
```



**Revert the dataset array to the original format.**

Reformat `dsNew` back into its original format.

```
dsOrig = unstack(dsNew, 'Scores', 'Scores_Indicator');
dsOrig(:, {'LastName', 'Test1', 'Test2', 'Test3', 'Test4'})
```

`ans =`

LastName	Test1	Test2	Test3	Test4
'HOWARD'	90	87	93	92
'WARD'	87	85	83	90
'TORRES'	86	85	88	86
'PETERSON'	75	80	72	77
'GRAY'	89	86	87	90
'RAMIREZ'	96	92	98	95
'JAMES'	78	75	77	77
'WATSON'	91	94	92	90
'BROOKS'	86	83	85	89
'KELLY'	79	76	82	80

The dataset array is back in wide format. `unstack` reassigns the levels of the indicator variable, `Scores_Indicator`, as variable names in the unstacked dataset array.

### See Also

`dataset` | `double` | `stack` | `unstack`

### Related Examples

- “Access Data in Dataset Array Variables” on page 2-93
- “Calculations on Dataset Arrays” on page 2-117
- “Index and Search Dataset Arrays” on page 2-145

### More About

- “Dataset Arrays” on page 2-142
- “Grouping Variables” on page 2-59

## Calculations on Dataset Arrays

This example shows how to perform calculations on dataset arrays.

### Load sample data.

Navigate to the folder containing sample data. Import the data from the comma-separated text file `testScores.csv`.

```
cd(matlabroot)
cd('help/toolbox/stats/examples')
ds = dataset('File','testScores.csv','Delimiter','(',')')

ds =

    LastName          Sex          Test1    Test2    Test3    Test4
    'HOWARD'         'male'       90       87       93       92
    'WARD'           'male'       87       85       83       90
    'TORRES'         'male'       86       85       88       86
    'PETERSON'       'female'     75       80       72       77
    'GRAY'           'female'     89       86       87       90
    'RAMIREZ'        'female'     96       92       98       95
    'JAMES'          'male'       78       75       77       77
    'WATSON'         'female'     91       94       92       90
    'BROOKS'        'female'     86       83       85       89
    'KELLY'         'male'       79       76       82       80
```

There are 4 test scores for each of 10 students, in wide format.

### Average dataset array variables.

Compute the average (mean) test score for each student in the dataset array, and store it in a new variable, `TestAvg`. Test scores are in columns 3 to 6.

Use `double` to convert the specified dataset array variables into a numeric array. Then, calculate the mean across the second dimension (across columns) to get the test average for each student.

```
ds.TestAvg = mean(double(ds(:,3:6)),2);
ds(:,{'LastName','TestAvg'})

ans =

    LastName          TestAvg
```

```
'HOWARD'          90.5
'WARD'            86.25
'TORRES'          86.25
'PETERSON'        76
'GRAY'            88
'RAMIREZ'         95.25
'JAMES'           76.75
'WATSON'          91.75
'BROOKS'          85.75
'KELLY'           79.25
```

### Summarize the dataset array using a grouping variable.

Compute the mean and maximum average test scores for each gender.

```
stats = grpstats(ds, 'Sex', {'mean', 'max'}, 'DataVars', 'TestAvg')

stats =

    male      Sex      GroupCount  mean_TestAvg  max_TestAvg
    female   'female'      5             83.8          90.5
    female   'female'      5             87.35         95.25
```

This returns a new dataset array containing the specified summary statistics for each level of the grouping variable, Sex.

### Replace data values.

The denominator for each test score is 100. Convert the test score denominator to 25.

```
scores = double(ds(:,3:6));
newScores = scores*25/100;
ds = replacedata(ds,newScores,3:6)

ds =

    LastName      Sex      Test1  Test2  Test3  Test4  TestAvg
    'HOWARD'      'male'    22.5  21.75  23.25   23    90.5
    'WARD'        'male'    21.75  21.25  20.75  22.5  86.25
    'TORRES'      'male'    21.5   21.25  22      21.5  86.25
    'PETERSON'    'female'  18.75  20     18     19.25  76
    'GRAY'        'female'  22.25  21.5   21.75  22.5   88
    'RAMIREZ'     'female'   24     23     24.5   23.75  95.25
    'JAMES'       'male'    19.5   18.75  19.25  19.25  76.75
    'WATSON'      'female'  22.75  23.5   23     22.5   91.75
```



```

'BROOKS'      'female'      21.5    20.75    21.25    22.25    85.75
'KELLY'       'male'        19.75    19        20.5     20        79.25

```

The first two lines of code extract the test data and perform the desired calculation. Then, `replacedata` inserts the new test scores back into the dataset array.

The variable of test score averages, `TestAvg`, is now the final score for each student.

### Change variable name.

Change the variable name to `Final`.

```

ds.Properties.VarNames{end} = 'Final';
ds

```

```

ds =

```

LastName	Sex	Test1	Test2	Test3	Test4	Final
'HOWARD'	'male'	22.5	21.75	23.25	23	90.5
'WARD'	'male'	21.75	21.25	20.75	22.5	86.25
'TORRES'	'male'	21.5	21.25	22	21.5	86.25
'PETERSON'	'female'	18.75	20	18	19.25	76
'GRAY'	'female'	22.25	21.5	21.75	22.5	88
'RAMIREZ'	'female'	24	23	24.5	23.75	95.25
'JAMES'	'male'	19.5	18.75	19.25	19.25	76.75
'WATSON'	'female'	22.75	23.5	23	22.5	91.75
'BROOKS'	'female'	21.5	20.75	21.25	22.25	85.75
'KELLY'	'male'	19.75	19	20.5	20	79.25

## See Also

`dataset` | `double` | `grpstats` | `replacedata`

## Related Examples

- “Stack or Unstack Dataset Arrays” on page 2-112
- “Access Data in Dataset Array Variables” on page 2-93
- “Select Subsets of Observations” on page 2-99
- “Index and Search Dataset Arrays” on page 2-145

## **More About**

- “Dataset Arrays” on page 2-142

## Export Dataset Arrays

This example shows how to export a dataset array from the MATLAB workspace to a text or spreadsheet file.

### Load sample data.

```
load('hospital')
```

The dataset array has 100 observations and 7 variables.

### Export to a text file.

Export the dataset array, `hospital`, to a text file named `hospital.txt`. By default, `export` writes to a tab-delimited text file with the same name as the dataset array, appended by `.txt`.

```
export(hospital)
```

This creates the file `hospital.txt` in the current working folder, if it does not previously exist. If the file already exists in the current working folder, `export` overwrites the existing file.

By default, variable names are in the first line of the text file. Observation names, if present, are in the first column.

### Export without variable names.

Export `hospital` with variable names suppressed to a text file named `NoLabels.txt`.

```
export(hospital, 'File', 'NoLabels.txt', 'WriteVarNames', false)
```

There are no variable names in the first line of the created text file, `NoLabels.txt`.

### Export to a comma-delimited format.

Export `hospital` to a comma-delimited text file, `hospital.csv`.

```
export(hospital, 'File', 'hospital.csv', 'Delimiter', ',')
```

### Export to an Excel spreadsheet.

Export `hospital` to an Excel spreadsheet named `hospital.xlsx`.

```
export(hospital, 'XLSFile', 'hospital.xlsx')
```

By default, the first row of `hospital.xlsx` has variable names, and the first column has observation names.

### See Also

`dataset` | `export`

### Related Examples

- “Create a Dataset Array from Workspace Variables” on page 2-70
- “Create a Dataset Array from a File” on page 2-77

### More About

- “Dataset Arrays” on page 2-142

## Clean Messy and Missing Data

This example shows how to find, clean, and delete observations with missing data in a dataset array.

### Load sample data.

Navigate to the folder containing sample data. Import the data from the spreadsheet `messy.xlsx`.

```
cd(matlabroot)
cd('help/toolbox/stats/examples')
messyData = dataset('XLSFile','messy.xlsx')
```

```
messyData =
```

var1	var2	var3	var4	var5
'afe1'	'3'	'yes'	'3'	3
'egh3'	'.'	'no'	'7'	7
'wth4'	'3'	'yes'	'3'	3
'atn2'	'23'	'no'	'23'	23
'arg1'	'5'	'yes'	'5'	5
'jre3'	'34.6'	'yes'	'34.6'	34.6
'wen9'	'234'	'yes'	'234'	234
'ple2'	'2'	'no'	'2'	2
'dbo8'	'5'	'no'	'5'	5
'oii4'	'5'	'yes'	'5'	5
'wnk3'	'245'	'yes'	'245'	245
'abk6'	'563'	''	'563'	563
'pnj5'	'463'	'no'	'463'	463
'wnn3'	'6'	'no'	'6'	6
'oks9'	'23'	'yes'	'23'	23
'wba3'	''	'yes'	'NaN'	14
'pkn4'	'2'	'no'	'2'	2
'adw3'	'22'	'no'	'22'	22
'poj2'	'-99'	'yes'	'-99'	-99
'bas8'	'23'	'no'	'23'	23
'gry5'	'NA'	'yes'	'NaN'	21

When you import data from a spreadsheet, `dataset` reads any variables with nonnumeric elements as a cell array of character vectors. This is why the variable `var2` is a cell array of character vectors. When importing data from a text file, you have more flexibility to specify which nonnumeric expressions to treat as missing using the option `TreatAsEmpty`.

There are many different missing data indicators in `messy.xlsx`, such as:

- Empty cells
- A period (.)
- NA
- NaN
- -99

### Find observations with missing values.

Display the subset of observations that have at least one missing value using `ismissing`.

```
ix = ismissing(messyData, 'NumericTreatAsMissing', -99, ...
              'StringTreatAsMissing', {'NaN', '.', 'NA'});
messyData(any(ix, 2), :)

ans =

    var1    var2    var3    var4    var5
    'egh3'    '.'    'no'    '7'    7
    'abk6'    '563'    ''    '563'    563
    'wba3'    ''    'yes'    'NaN'    14
    'poj2'    '-99'    'yes'    '-99'    -99
    'gry5'    'NA'    'yes'    'NaN'    21
```

By default, `ismissing` recognizes the following missing value indicators:

- NaN for numeric arrays
- '' for character arrays
- <undefined> for categorical arrays

Use the `NumericTreatAsMissing` and `StringTreatAsMissing` options to specify other values to treat as missing.

### Convert character arrays to double arrays.

You can convert the `char` variables that should be numeric using `str2double`.

```
messyData.var2 = str2double(messyData.var2);
messyData.var4 = str2double(messyData.var4)
```

```

messyData =

    var1      var2      var3      var4      var5
    'afe1'         3      'yes'         3         3
    'egh3'        NaN      'no'          7         7
    'wth4'         3      'yes'         3         3
    'atn2'        23      'no'        23         23
    'arg1'         5      'yes'         5         5
    'jre3'       34.6      'yes'       34.6       34.6
    'wen9'       234      'yes'       234        234
    'ple2'         2      'no'          2         2
    'dbo8'         5      'no'          5         5
    'oii4'         5      'yes'         5         5
    'wnk3'       245      'yes'       245        245
    'abk6'       563      ''           563        563
    'pnj5'       463      'no'        463        463
    'wnn3'         6      'no'          6         6
    'oks9'        23      'yes'        23         23
    'wba3'        NaN      'yes'        NaN         14
    'pkn4'         2      'no'          2         2
    'adw3'        22      'no'        22         22
    'poj2'       -99      'yes'       -99        -99
    'bas8'        23      'no'         23         23
    'gry5'        NaN      'yes'        NaN         21

```

Now, `var2` and `var4` are numeric arrays. During the conversion, `str2double` replaces the nonnumeric elements of the variables `var2` and `var4` with the value `NaN`. However, there are no changes to the numeric missing value indicator, `-99`.

When applying the same function to many dataset array variables, it can sometimes be more convenient to use `datasetfun`. For example, to convert both `var2` and `var4` to numeric arrays simultaneously, you can use:

```

messyData(:, [2, 4]) = datasetfun(@str2double, messyData, 'DataVars', [2, 4], ...
    'DatasetOutput', true);

```

### Replace missing value indicators.

Clean the data so that the missing values indicated by the code `-99` have the standard MATLAB numeric missing value indicator, `NaN`.

```

messyData = replaceWithMissing(messyData, 'NumericValues', -99)

messyData =

```

var1	var2	var3	var4	var5
'afe1'	3	'yes'	3	3
'egh3'	NaN	'no'	7	7
'wth4'	3	'yes'	3	3
'atn2'	23	'no'	23	23
'arg1'	5	'yes'	5	5
'jre3'	34.6	'yes'	34.6	34.6
'wen9'	234	'yes'	234	234
'ple2'	2	'no'	2	2
'dbo8'	5	'no'	5	5
'oii4'	5	'yes'	5	5
'wnk3'	245	'yes'	245	245
'abk6'	563	''	563	563
'pnj5'	463	'no'	463	463
'wnn3'	6	'no'	6	6
'oks9'	23	'yes'	23	23
'wba3'	NaN	'yes'	NaN	14
'pkn4'	2	'no'	2	2
'adw3'	22	'no'	22	22
'poj2'	NaN	'yes'	NaN	NaN
'bas8'	23	'no'	23	23
'gry5'	NaN	'yes'	NaN	21

### Create a dataset array with complete observations.

Create a new dataset array that contains only the complete observations—those without missing data.

```
ix = ismissing(messyData);  
completeData = messyData(~any(ix,2),:)  
  
completeData =
```

var1	var2	var3	var4	var5
'afe1'	3	'yes'	3	3
'wth4'	3	'yes'	3	3
'atn2'	23	'no'	23	23
'arg1'	5	'yes'	5	5
'jre3'	34.6	'yes'	34.6	34.6
'wen9'	234	'yes'	234	234
'ple2'	2	'no'	2	2
'dbo8'	5	'no'	5	5
'oii4'	5	'yes'	5	5
'wnk3'	245	'yes'	245	245
'pnj5'	463	'no'	463	463



---

'wnn3'	6	'no'	6	6
'oks9'	23	'yes'	23	23
'pkn4'	2	'no'	2	2
'adw3'	22	'no'	22	22
'bas8'	23	'no'	23	23

## See Also

`dataset` | `ismissing` | `replaceWithMissing`

## Related Examples

- “Select Subsets of Observations” on page 2-99
- “Calculations on Dataset Arrays” on page 2-117
- “Index and Search Dataset Arrays” on page 2-145

## More About

- “Dataset Arrays” on page 2-142

## Dataset Arrays in the Variables Editor

---

**Note** The `dataset` data type might be removed in a future release. To work with heterogeneous data, use the MATLAB `table` data type instead. See MATLAB `table` documentation for more information.

---

### In this section...

“Open Dataset Arrays in the Variables Editor” on page 2-128

“Modify Variable and Observation Names” on page 2-129

“Reorder or Delete Variables” on page 2-131

“Add New Data” on page 2-133

“Sort Observations” on page 2-135

“Select a Subset of Data” on page 2-136

“Create Plots” on page 2-139

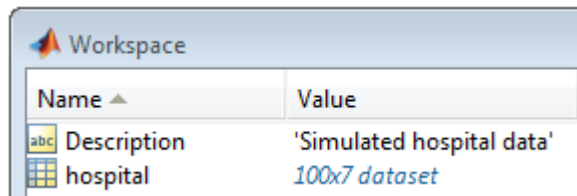
## Open Dataset Arrays in the Variables Editor

The MATLAB Variables editor provides a convenient interface for viewing, modifying, and plotting dataset arrays.

First, load the sample data set, `hospital`.

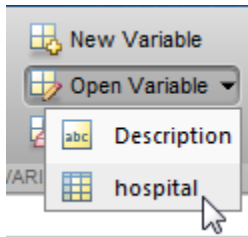
```
load hospital
```

The dataset array, `hospital`, is created in the MATLAB workspace.



The dataset array has 100 observations and 7 variables.

To open `hospital` in the Variables editor, click **Open Variable**, and select `hospital`.



The Variables editor opens, displaying the contents of the dataset array (only the first 10 observations are shown here).

	1 LastName	2 Sex	3 Age	4 Weight	5 Smoker	6 BloodPressure	7 Trials
1	YPL-320 'SMITH'	Male	38	176	1	124	93 18
2	GLI-532 'JOHNSON'	Male	43	163	0	109	77 [11,13,22]
3	PNI-258 'WILLIAMS'	Female	38	131	0	125	83 []
4	MIJ-579 'JONES'	Female	40	133	0	117	75 [6,12]
5	XLK-030 'BROWN'	Female	49	119	0	122	80 [14,23]
6	TFP-518 'DAVIS'	Female	46	142	0	121	70 19
7	LPD-746 'MILLER'	Female	33	142	1	130	88 13
8	ATA-945 'WILSON'	Male	40	180	0	115	82 []
9	VNL-702 'MOORE'	Male	28	183	0	115	78 2
10	LQW-768 'TAYLOR'	Female	31	132	0	118	86 11

In the Variables editor, you can see the names of the seven variables along the top row, and the observations names down the first column.

## Modify Variable and Observation Names

You can modify variable and observation names by double-clicking a name, and then typing new text.

hospital ✕

100x7 [dataset](#)

	1 LastName	2 Sex
1 YPL-320	'SMITH'	Male
2 GLI-532	'JOHNSON'	Male
3 PNI-258	'WILLIAMS'	Female
4 MIJ-579	'JONES'	Female
5 XLK-030	'BROWN'	Female

hospital ✕

100x7 [dataset](#)

	1 LastName	2 Gender
1 YPL-320	'SMITH'	Male
2 GLI-532	'JOHNSON'	Male
3 PNI-258	'WILLIAMS'	Female
4 MIJ-579	'JONES'	Female
5 XLK-030	'BROWN'	Female

All changes made in the Variables editor are also sent to the command line.

Command Window

```
>> hospital.Properties.VarNames{2} = 'Gender';
fx >>
```

The sixth variable in the data set, `BloodPressure`, is a numeric array with two columns. The first column shows systolic blood pressure, and the second column shows diastolic blood pressure. Click the arrow that appears on the right side of the variable name cell to see the units and description of the variable. You can type directly in the units and description fields to modify the text. The variable data type and size are shown under the variable description.

6	7	8	9
BloodPressure	Trials		
124	Ascending		
109	Descending		
125	UNITS		
117	mm Hg		
122	DESCRIPTION		
121	Systolic/Diastolic		
130			
115	100x2 double		
115			

## Reorder or Delete Variables

You can reorder variables in a dataset array using the Variables editor. Hover over the left side of a variable name cell until a four-headed arrow appears.

4	5
Weight	Smoker
176	1
163	0
131	0
133	0
119	0
142	0

After the arrow appears, click and drag the variable column to a new location.

4	5	5	6
Weight	Smoke	Smoker	BloodPressure
176	1	1	124
163	0	0	109
131	0	0	125
133	0	0	117
119	0	0	122
142	0	0	121
142	1	1	130
180	0	0	115
183	0	0	115
132	0	0	118
128	0	0	114
137	0	0	115
174	0	0	127

5	6	7
BloodPressure	Smoker	Trials
124	93	1 18
109	77	0 [11,13,22]
125	83	0 []
117	75	0 [6,12]
122	80	0 [14,23]
121	70	0 19
130	88	1 17

The command for the variable reordering appears in the command line.

```
Command Window
>> hospital = hospital(:, [1:4 6 5 end]);
fx >>
```

You can delete a variable in the Variables editor by selecting the variable column, right-clicking, and selecting **Delete Column Variable(s)**.

3	4	5	6	7
Age	Weight			
38				
43				
38				
40				
49				
46				
33				
40				
28				
31				
45				

Cut	Ctrl+X
Copy	Ctrl+C
Paste	Ctrl+V
Paste as New Dataset Column Variable(s) to the left	
Paste as New Dataset Column Variable(s) to the right	
Delete Dataset Column Variable(s)	
Sort Ascending	
Sort Descending	
New Workspace Variable from Selection	

The command for the variable deletion appears in the command line.

```
Command Window
>> hospital(:, 'Weight') = [];
fx >>
```

## Add New Data

You can enter new data values directly into the Variables editor. For example, you can add a new patient observation to the `hospital` data set. To enter a new last name, add a character vector to the end of the variable `LastName`.

100	ZZB-405	'HAYES'	Male	48	114	86
101		'JONES'				
102						

The variable `Gender` is a nominal array. The levels of the categorical variable appear in a drop-down list when you double-click a cell in the `Gender` column. You can choose one of the levels previously used, or create a new level by selecting **New Item**.

100	ZB-405	'HAYES'	Male	48
101	Obs101	'JONES'	<undef...>	0
102			Female	
103			Male	
104			<undefined>	
105			New item	

You can continue to add data for the remaining variables.

To change the observation name, click the observation name and type the new name.

100	ZB-405	'HAYES'	Male	48
101	Obs101	'JONES'	Female	45
102				

The commands for entering the new data appear at the command line.

```
Command Window
>> hospital.LastName{101} = 'JONES';
Warning: Observations with default values added to dataset
variables.
> In dataset.subsasgn at 584
>> hospital.Sex(101) = 'Female';
>> hospital.Age(101) = 45;
>> hospital.BloodPressure(101,2) = 85;
>> hospital.BloodPressure(101,1) = 120;
>> hospital.Properties.ObsNames{101} = 'QPO-187';
fx >>
```

Notice the warning that appears after the first assignment. When you enter the first piece of data in the new observation row—here, the last name—default values are assigned to all other variables. Default assignments are:

- 0 for numeric variables
- <undefined> for categorical variables
- [] for cell arrays



You can also copy and paste data from one dataset array to another using the Variables editor.

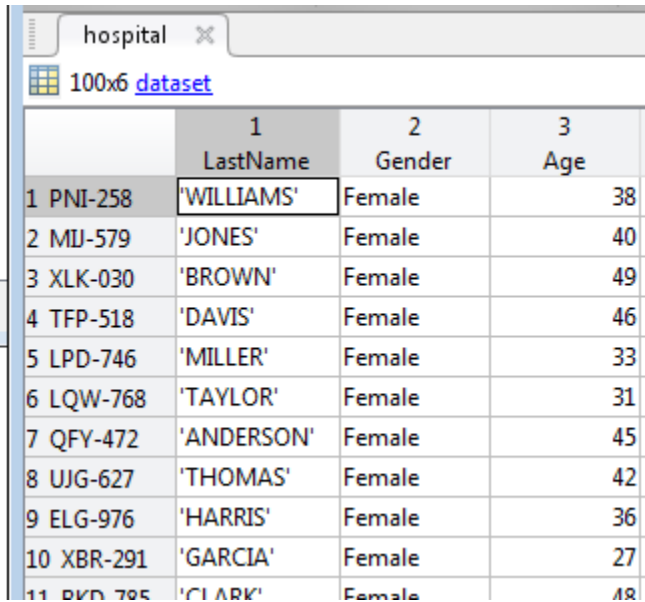
## Sort Observations

You can use the Variables editor to sort dataset array observations by the values of one or more variables. To sort by gender, for example, select the variable `Gender`. Then click **Sort**, and choose to sort rows by ascending or descending values of the selected variable.

The screenshot shows the Variables Editor interface with the 'VIEW' tab selected. The 'Sort' button is highlighted, and its dropdown menu is open, showing two options: 'Sort Ascending by Selected Columns' and 'Sort Descending by Selected Columns'. The 'Gender' column in the dataset is selected. Below the interface, a table displays the data for the 'hospital' dataset, sorted by the 'Gender' column.

	1	2	3	4	5	6	
	LastName	Gender	Age	BloodPressure	Smoker	Trials	
1	YPL-320	'SMITH'	Male	38	124	93	1 18
2	GLI-532	'JOHNSON'	Male	43	109	77	0 [11,13,22]
3	PNI-258	'WILLIAMS'	Female	38	125	83	0 []
4	MJ-579	'JONES'	Female	40	117	75	0 [6,12]
5	XLK-030	'BROWN'	Female	49	122	80	0 [14,23]

When sorting by variables that are cell arrays of character vectors or of nominal data type, observations are sorted alphabetically. For ordinal variables, rows are sorted by the ordering of the levels. For example, when the observations of `hospital` are sorted by the values in `Gender`, the females are grouped together, followed by the males.



	1 LastName	2 Gender	3 Age
1 PNI-258	'WILLIAMS'	Female	38
2 MJJ-579	'JONES'	Female	40
3 XLK-030	'BROWN'	Female	49
4 TFP-518	'DAVIS'	Female	46
5 LPD-746	'MILLER'	Female	33
6 LQW-768	'TAYLOR'	Female	31
7 QFY-472	'ANDERSON'	Female	45
8 UJG-627	'THOMAS'	Female	42
9 ELG-976	'HARRIS'	Female	36
10 XBR-291	'GARCIA'	Female	27
11 BVD-785	'CLARK'	Female	48

To sort by the values of multiple variables, press **Ctrl** while you select multiple variables.

When you use the Variables editor to sort rows, it is the same as calling `sortrows`. You can see this at the command line after executing the sorting.

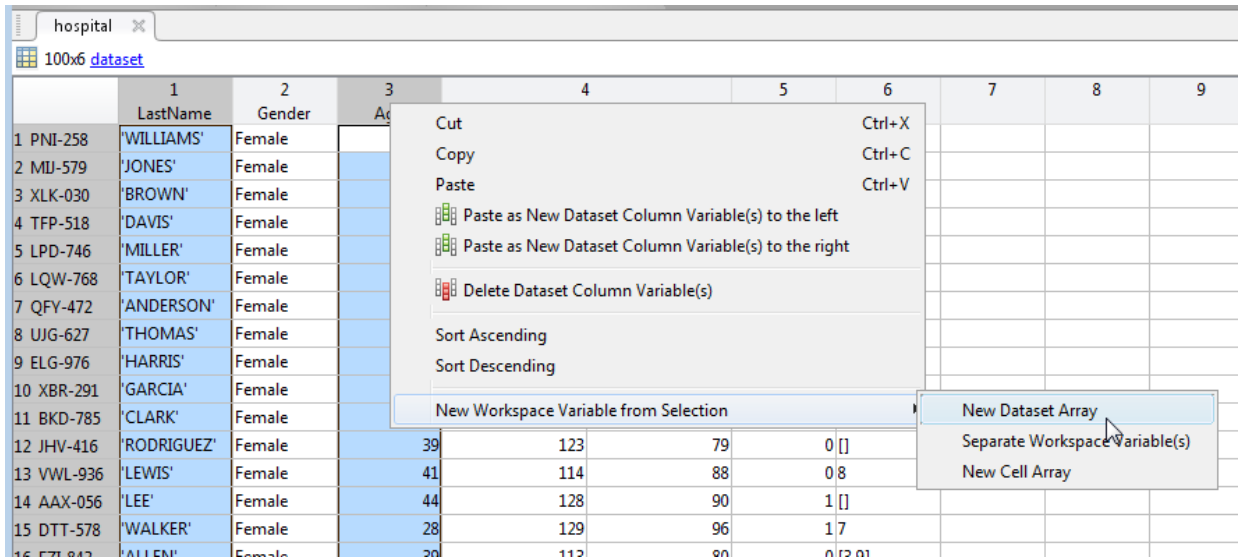
Command Window

```
>> hospital = sortrows(hospital, 'Gender', 'ascend');  
fx >>
```

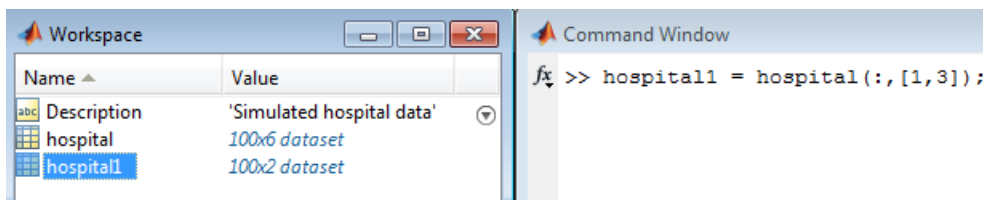
### Select a Subset of Data

You can select a subset of data from a dataset array in the Variables editor, and create a new dataset array from the selection. For example, to create a dataset array containing only the variables `LastName` and `Age`:

- 1 Hold **Ctrl** while you click the variables `LastName` and `Age`.
- 2 Right-click, and select **New Workspace Variable from Selection > New Dataset Array**.



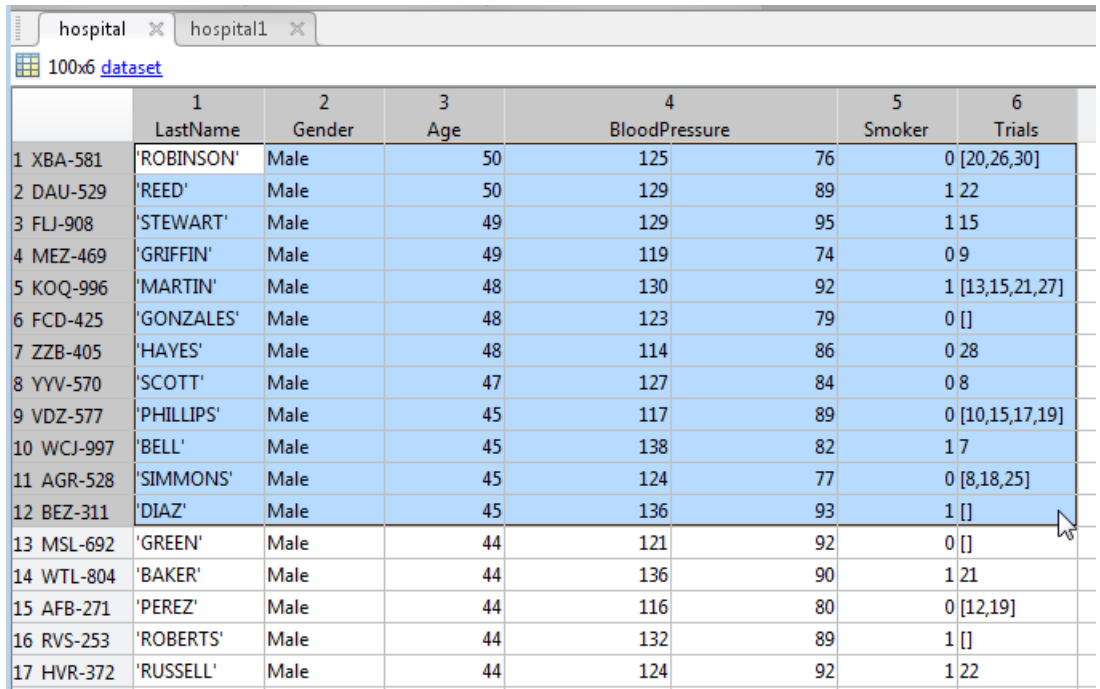
The new dataset array appears in the Workspace window with the name `hospital1`. The Command Window shows the commands that execute the selection.



You can use the same steps to select any subset of data. To select observations according to some logical condition, you can use a combination of sorting and selecting. For example, to create a new dataset array containing only males aged 45 and older:

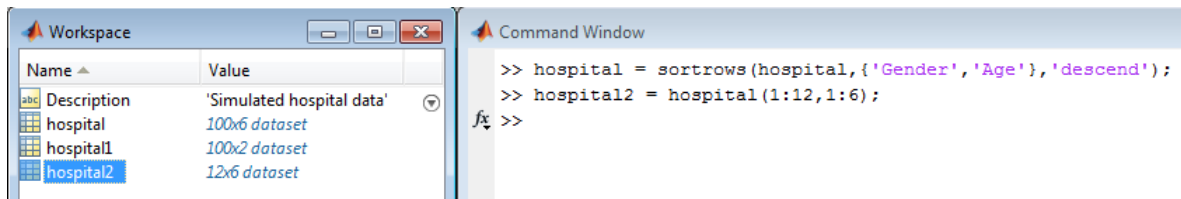
- 1 Sort the observations of `hospital` by the values in `Gender` and `Age`, descending.
- 2 Select the male observations with age 45 and older.

## 2 Organizing Data

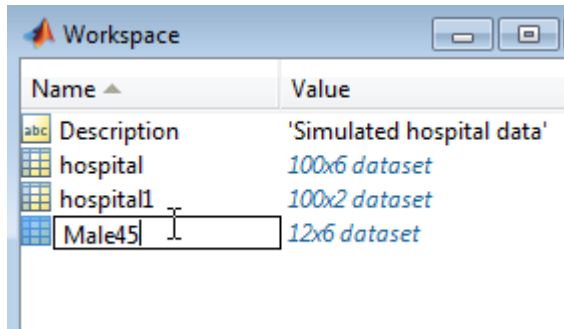


	1	2	3	4	5	6	
	LastName	Gender	Age	BloodPressure	Smoker	Trials	
1	XBA-581	'ROBINSON'	Male	50	125	76	0 [20,26,30]
2	DAU-529	'REED'	Male	50	129	89	1 22
3	FLJ-908	'STEWART'	Male	49	129	95	1 15
4	MEZ-469	'GRIFFIN'	Male	49	119	74	0 9
5	KOQ-996	'MARTIN'	Male	48	130	92	1 [13,15,21,27]
6	FCD-425	'GONZALES'	Male	48	123	79	0 []
7	ZZB-405	'HAYES'	Male	48	114	86	0 28
8	YYV-570	'SCOTT'	Male	47	127	84	0 8
9	VDZ-577	'PHILLIPS'	Male	45	117	89	0 [10,15,17,19]
10	WCJ-997	'BELL'	Male	45	138	82	1 7
11	AGR-528	'SIMMONS'	Male	45	124	77	0 [8,18,25]
12	BEZ-311	'DIAZ'	Male	45	136	93	1 []
13	MSL-692	'GREEN'	Male	44	121	92	0 []
14	WTL-804	'BAKER'	Male	44	136	90	1 21
15	AFB-271	'PEREZ'	Male	44	116	80	0 [12,19]
16	RVS-253	'ROBERTS'	Male	44	132	89	1 []
17	HVR-372	'RUSSELL'	Male	44	124	92	1 22

- 3 Right-click, and select **New Workspace Variables from Selection > New Dataset Array**. The new dataset array, `hospital2`, is created in the Workspace window.



- 4 You can rename the dataset array in the Workspace window.



## Create Plots

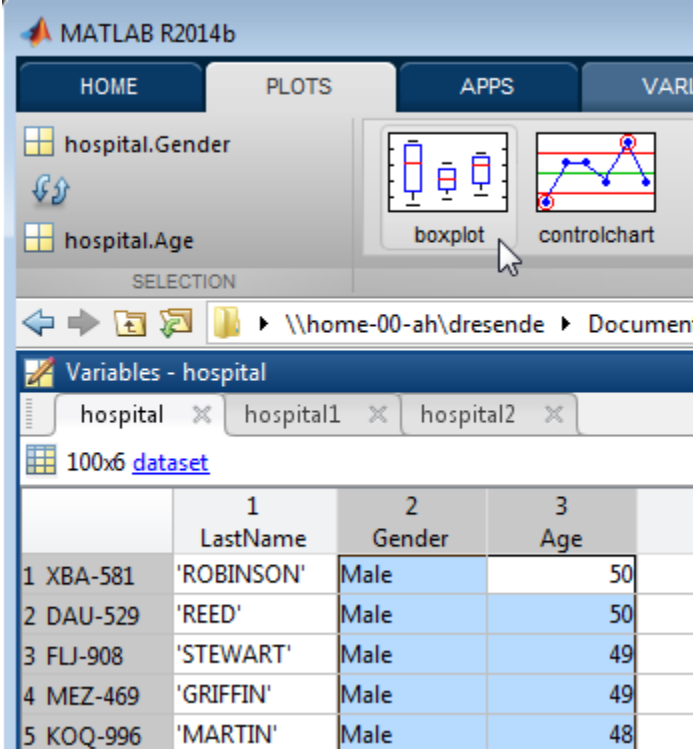
You can plot data from a dataset array using plotting options in the Variables editor. Available plot choices depend on the data types of variables to be plotted.

For example, if you select the variable `Age`, you can see in the **Plots** tab some plotting options that are appropriate for a univariate, numeric variable.

Variables - hospital

	1	2	3	4	5
	LastName	Gender	Age	BloodPressure	Smc
1 XBA-581	'ROBINSON'	Male	50	125	76
2 DAU-529	'REED'	Male	50	129	89
3 FLJ-908	'STEWART'	Male	49	129	95
4 MEZ-469	'GRIFFIN'	Male	49	119	74
5 KOQ-996	'MARTIN'	Male	48	130	92
6 FCD-425	'GONZALES'	Male	48	123	79

Sometimes, there are plot options for multiple variables, depending on their data types. For example, if you select both `Age` and `Gender`, you can draw box plots of age, grouped by gender.



The image shows the MATLAB R2014b interface. The 'PLOTS' tab is active, displaying two plot options: 'boxplot' and 'controlchart'. The 'boxplot' option is selected, indicated by a mouse cursor. Below the plot options is a 'Variables - hospital' window showing a dataset with columns for 'LastName', 'Gender', and 'Age'. The dataset contains 5 rows of data.

	1	2	3	
	LastName	Gender	Age	
1	XBA-581	'ROBINSON'	Male	50
2	DAU-529	'REED'	Male	50
3	FLJ-908	'STEWART'	Male	49
4	MEZ-469	'GRIFFIN'	Male	49
5	KOQ-996	'MARTIN'	Male	48

## See Also

[dataset](#) | [sortrows](#)

## Related Examples

- “Add and Delete Observations” on page 2-85
- “Add and Delete Variables” on page 2-89
- “Access Data in Dataset Array Variables” on page 2-93
- “Select Subsets of Observations” on page 2-99

- “Sort Observations in Dataset Arrays” on page 2-103

## **More About**

- “Dataset Arrays” on page 2-142

## Dataset Arrays

---

**Note** The `dataset` data type might be removed in a future release. To work with heterogeneous data, use the MATLAB `table` data type instead. See MATLAB `table` documentation for more information.

---

In this section...
“What Are Dataset Arrays?” on page 2-142
“Dataset Array Conversion” on page 2-142
“Dataset Array Properties” on page 2-143

### What Are Dataset Arrays?

Statistics and Machine Learning Toolbox has dataset arrays for storing variables with heterogeneous data types. For example, you can combine numeric data, logical data, cell arrays of character vectors, and categorical arrays in one dataset array variable.

Within a dataset array, each variable (column) must be one homogeneous data type, but the different variables can be of heterogeneous data types. A dataset array is usually interpreted as a set of variables measured on many units of observation. That is, each row in a dataset array corresponds to an observation, and each column to a variable. In this sense, a dataset array organizes data like a typical spreadsheet.

Dataset arrays are a unique data type, with a corresponding set of valid operations. Even if a dataset array contains only numeric variables, you cannot operate on the dataset array like a numeric variable. The valid operations for dataset arrays are the methods of the `dataset` class.

### Dataset Array Conversion

You can create a dataset array by combining variables that exist in the MATLAB workspace, or directly importing data from a file, such as a text file or spreadsheet. This table summarizes the functions you can use to create dataset arrays.

Data Source	Conversion to Dataset Array
Data from a file	<code>dataset</code>



Data Source	Conversion to Dataset Array
Heterogeneous collection of workspace variables	<code>dataset</code>
Numeric array	<code>mat2dataset</code>
Cell array	<code>cell2dataset</code>
Structure array	<code>struct2dataset</code>
Table	<code>table2dataset</code>

You can export dataset arrays to text or spreadsheet files using `export`. To convert a dataset array to a cell array or structure array, use `dataset2cell` or `dataset2struct`. To convert a dataset array to a table, use `dataset2table`.

## Dataset Array Properties

In addition to storing data in a dataset array, you can store metadata such as:

- Variable and observation names
- Data descriptions
- Units of measurement
- Variable descriptions

This information is stored as dataset array properties. For a dataset array named `ds`, you can view the dataset array metadata by entering `ds.Properties` at the command line. You can access a specific property, such as variable names—property `VarNames`—using `ds.Properties.VarNames`. You can both retrieve and modify property values using this syntax.

Variable and observation names are included in the display of a dataset array. Variable names display across the top row, and observation names, if present, appear in the first column. Note that variable and observation names do not affect the size of a dataset array.

## See Also

`cell2dataset` | `dataset` | `dataset2cell` | `dataset2struct` | `dataset2table` | `export` | `mat2dataset` | `struct2dataset` | `table2dataset`

### Related Examples

- “Create a Dataset Array from Workspace Variables” on page 2-70
- “Create a Dataset Array from a File” on page 2-77
- “Export Dataset Arrays” on page 2-121
- “Dataset Arrays in the Variables Editor” on page 2-128
- “Index and Search Dataset Arrays” on page 2-145

## Index and Search Dataset Arrays

---

**Note** The `dataset` data type might be removed in a future release. To work with heterogeneous data, use the MATLAB `table` data type instead. See MATLAB `table` documentation for more information.

---

### Ways To Index and Search

There are many ways to index into dataset arrays. For example, for a dataset array, `ds`, you can:

- Use `()` to create a new dataset array from a subset of `ds`. For example, `ds1 = ds(1:5, :)` creates a new dataset array, `ds1`, consisting of the first five rows of `ds`. Metadata, including variable and observation names, transfers to the new dataset array.
- Use variable names with dot notation to index individual variables in a dataset array. For example, `ds.Height` indexes the variable named `Height`.
- Use observation names to index individual observations in a dataset array. For example, `ds('Obs1', :)` gives data for the observation named `Obs1`.
- Use observation or variable numbers. For example, `ds(:, [1, 3, 5])` gives the data in the first, third, and fifth variables (columns) of `ds`.
- Use logical indexing to search for observations in `ds` that satisfy a logical condition. For example, `ds(ds.Gender=='Male', :)` gives the observations in `ds` where the variable named `Gender`, a nominal array, has the value `Male`.
- Use `ismissing` to find missing data in the dataset array.

### Examples

#### Common Indexing and Searching Methods

This example shows several indexing and searching methods for categorical arrays.

Load the sample data.

```
load hospital;  
size(hospital)
```

```
ans =  
    100     7
```

The dataset array has 100 observations and 7 variables.

Index a variable by name. Return the minimum age in the dataset array.

```
min(hospital.Age)
```

```
ans =  
    25
```

Delete the variable `Trials`.

```
hospital.Trials = [];  
size(hospital)
```

```
ans =  
    100     6
```

Index an observation by name. Display measurements on the first five variables for the observation named `PUE-347`.

```
hospital('PUE-347',1:5)
```

```
ans =  
    PUE-347    LastName    Sex    Age    Weight    Smoker  
    PUE-347    'YOUNG'    Female    25    114    false
```

Index variables by number. Create a new dataset array containing the first four variables of `hospital`.

```
dsNew = hospital(:,1:4);  
dsNew.Properties.VarNames(:)
```

```
ans =

4x1 cell array

    {'LastName'}
    {'Sex'      }
    {'Age'      }
    {'Weight'   }
```

Index observations by number. Delete the last 10 observations.

```
hospital(end-9:end,:) = [];
size(hospital)
```

```
ans =

    90     6
```

Search for observations by logical condition. Create a new dataset array containing only females who smoke.

```
dsFS = hospital(hospital.Sex=='Female' & hospital.Smoker==true,:);
dsFS(:, {'LastName', 'Sex', 'Smoker'})
```

```
ans =

      LastName      Sex      Smoker
LPD-746  'MILLER'    Female    true
XBR-291  'GARCIA'    Female    true
AAX-056  'LEE'        Female    true
DTT-578  'WALKER'    Female    true
AFK-336  'WRIGHT'    Female    true
RBA-579  'SANCHEZ'    Female    true
HAK-381  'MORRIS'    Female    true
NSK-403  'RAMIREZ'    Female    true
ILS-109  'WATSON'    Female    true
JDR-456  'SANDERS'    Female    true
HWZ-321  'PATTEKSON'  Female    true
GGU-691  'HUGHES'    Female    true
```

```
WUS-105    'FLORES'           Female    true
```

### See Also

dataset

### Related Examples

- “Access Data in Dataset Array Variables” on page 2-93
- “Select Subsets of Observations” on page 2-99

### More About

- “Dataset Arrays” on page 2-142

# Descriptive Statistics

---

- “Introduction to Descriptive Statistics” on page 3-2
- “Measures of Central Tendency” on page 3-3
- “Measures of Dispersion” on page 3-5
- “Quantiles and Percentiles” on page 3-7
- “Exploratory Analysis of Data” on page 3-11
- “Resampling Statistics” on page 3-17
- “Data with Missing Values” on page 3-22

## Introduction to Descriptive Statistics

You may need to summarize large, complex data sets—both numerically and visually—to convey their essence to the data analyst and to allow for further processing.



## Measures of Central Tendency

Measures of central tendency locate a distribution of data along an appropriate scale.

The following table lists the functions that calculate the measures of central tendency.

Function Name	Description
geomean	Geometric mean
harmmean	Harmonic mean
mean	Arithmetic average
median	50th percentile
mode	Most frequent value
trimmean	Trimmed mean

The average is a simple and popular estimate of location. If the data sample comes from a normal distribution, then the sample mean is also optimal (minimum variance unbiased estimator (MVUE) of  $\mu$ ).

Unfortunately, outliers, data entry errors, or glitches exist in almost all real data. The sample mean is sensitive to these problems. One bad data value can move the average away from the center of the rest of the data by an arbitrarily large distance.

The median and trimmed mean are two measures that are resistant (robust) to outliers. The median is the 50th percentile of the sample, which will only change slightly if you add a large perturbation to any value. The idea behind the trimmed mean is to ignore a small percentage of the highest and lowest values of a sample when determining the center of the sample.

The geometric mean and harmonic mean, like the average, are not robust to outliers. They are useful when the sample is distributed lognormal or heavily skewed.

## Measures of Central Tendency

This example shows how to compute and compare measures of location for sample data that contains one outlier.

Generate sample data that contains one outlier.

```
x = [ones(1,6),100]
```

```
x =  
    1    1    1    1    1    1  100
```

Compute the geometric mean, harmonic mean, mean, median, and trimmed mean for the sample data.

```
locate = [geomean(x) harmmean(x) mean(x) median(x) ...  
          trimmean(x,25)]
```

```
locate =  
    1.9307    1.1647    15.1429    1.0000    1.0000
```

The mean (`mean`) is far from any data value because of the influence of the outlier. The geometric mean (`geomean`) and the harmonic mean (`harmmean`) are influenced by the outlier, but not as significantly. The median (`median`) and trimmed mean (`trimmean`) ignore the outlier value and describe the location of the rest of the data values.

## See Also

### Related Examples

- “Exploratory Analysis of Data” on page 3-11

## Measures of Dispersion

The purpose of measures of dispersion is to find out how spread out the data values are on the number line. Another term for these statistics is measures of spread.

The table gives the function names and descriptions.

Function Name	Description
iqr	Interquartile range
mad	Mean absolute deviation
moment	Central moment of all orders
range	Range
std	Standard deviation
var	Variance

The range (the difference between the maximum and minimum values) is the simplest measure of spread. But if there is an outlier in the data, it will be the minimum or maximum value. Thus, the range is not robust to outliers.

The standard deviation and the variance are popular measures of spread that are optimal for normally distributed samples. The sample variance is the minimum variance unbiased estimator (MVUE) of the normal parameter  $\sigma^2$ . The standard deviation is the square root of the variance and has the desirable property of being in the same units as the data. That is, if the data is in meters, the standard deviation is in meters as well. The variance is in meters<sup>2</sup>, which is more difficult to interpret.

Neither the standard deviation nor the variance is robust to outliers. A data value that is separate from the body of the data can increase the value of the statistics by an arbitrarily large amount.

The mean absolute deviation (MAD) is also sensitive to outliers. But the MAD does not move quite as much as the standard deviation or variance in response to bad data.

The interquartile range (IQR) is the difference between the 75th and 25th percentile of the data. Since only the middle 50% of the data affects this measure, it is robust to outliers.

## Compare Measures of Dispersion

This example shows how to compute and compare measures of dispersion for sample data that contains one outlier.

Generate sample data that contains one outlier value.

```
x = [ones(1,6),100]
```

```
x =
```

```
     1     1     1     1     1     1    100
```

Compute the interquartile range, mean absolute deviation, range, and standard deviation of the sample data.

```
stats = [iqr(x),mad(x),range(x),std(x)]
```

```
stats =
```

```
     0    24.2449    99.0000    37.4185
```

The interquartile range (`iqr`) is the difference between the 75th and 25th percentile of the sample data, and is robust to outliers. The range (`range`) is the difference between the maximum and minimum values in the data, and is strongly influenced by the presence of an outlier.

Both the mean absolute deviation (`mad`) and the standard deviation (`std`) are sensitive to outliers. However, the mean absolute deviation is less sensitive than the standard deviation.

## See Also

### Related Examples

- “Exploratory Analysis of Data” on page 3-11

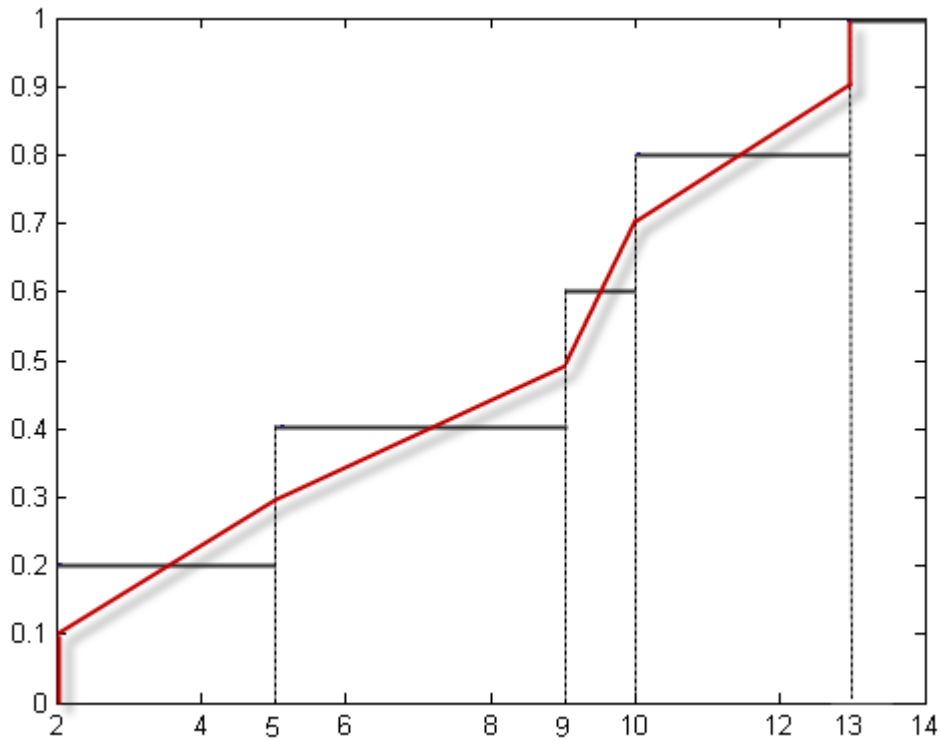
## Quantiles and Percentiles

This section explains how the Statistics and Machine Learning Toolbox functions `quantile` and `prctile` compute quantiles and percentiles.

The `prctile` function calculates the percentiles in a similar way as `quantile` calculates quantiles. The following steps in the computation of quantiles are also true for percentiles, given the fact that, for the same data sample, the quantile at the value  $Q$  is the same as the percentile at the value  $P = 100*Q$ .

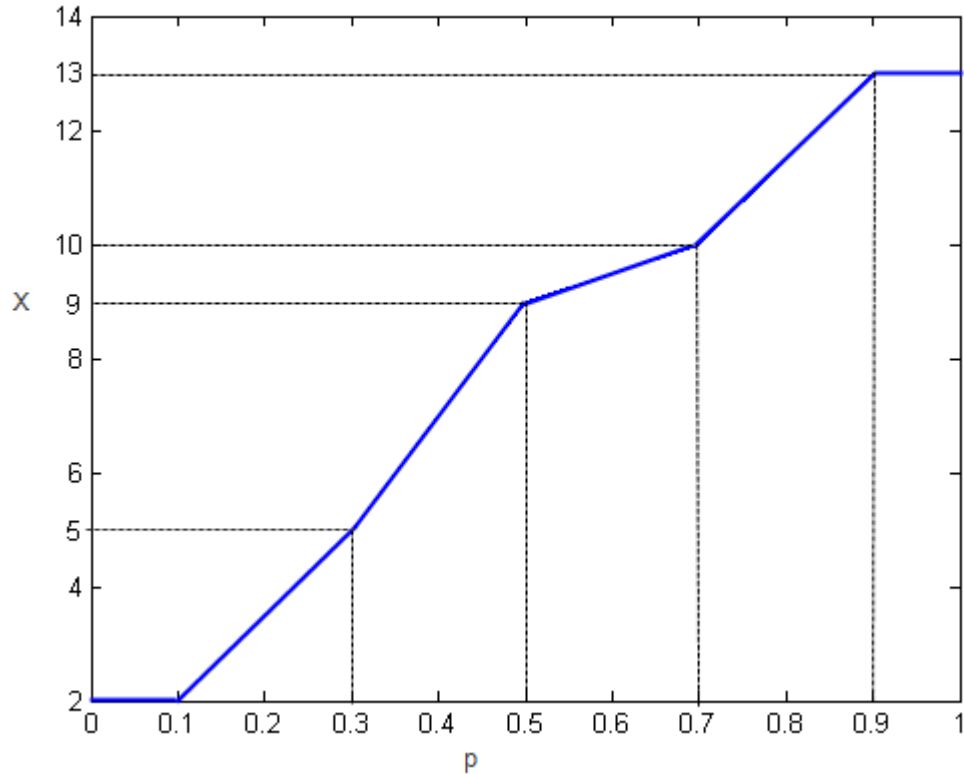
- 1 `quantile` initially assigns the sorted values in  $x$  to the  $(0.5/n)$ ,  $(1.5/n)$ , ...,  $([n - 0.5]/n)$  quantiles. For example:
  - For a data vector of six elements such as {6, 3, 2, 10, 8, 1}, the sorted elements {1, 2, 3, 6, 8, 10} respectively correspond to the  $(0.5/6)$ ,  $(1.5/6)$ ,  $(2.5/6)$ ,  $(3.5/6)$ ,  $(4.5/6)$ , and  $(5.5/6)$  quantiles.
  - For a data vector of five elements such as {2, 10, 5, 9, 13}, the sorted elements {2, 5, 9, 10, 13} respectively correspond to the 0.1, 0.3, 0.5, 0.7, and 0.9 quantiles.

The following figure illustrates this approach for data vector  $X = \{2, 10, 5, 9, 13\}$ . The first observation corresponds to the cumulative probability  $1/5 = 0.2$ , the second observation corresponds to the cumulative probability  $2/5 = 0.4$ , and so on. The step function in this figure shows these cumulative probabilities. `quantile` instead places the observations in midpoints, such that the first corresponds to  $0.5/5 = 0.1$ , the second corresponds to  $1.5/5 = 0.3$ , and so on, and then connects these midpoints. The red lines in the following figure connect the midpoints.



### Assigning Observations to Quantiles

By switching the axes, as the next figure, you can see the values of the variable  $X$  that correspond to the  $p$  quantiles.



### Quantiles of $X$

- 2 quantile finds any quantiles between the data values using linear interpolation.

*Linear interpolation* uses linear polynomials to approximate a function  $f(x)$  and construct new data points within the range of a known set of data points.

Algebraically, given the data points  $(x_1, y_1)$  and  $(x_2, y_2)$ , where  $y_1 = f(x_1)$  and  $y_2 = f(x_2)$ , linear interpolation finds  $y = f(x)$  for a given  $x$  between  $x_1$  and  $x_2$  as follows:

$$y = f(x) = y_1 + \frac{(x - x_1)}{(x_2 - x_1)}(y_2 - y_1).$$

Similarly, if the  $1.5/n$  quantile is  $y_{1.5/n}$  and the  $2.5/n$  quantile is  $y_{2.5/n}$ , then linear interpolation finds the  $2.3/n$  quantile  $y_{2.3/n}$  as

$$y_{\frac{2.3}{n}} = y_{\frac{1.5}{n}} + \left( \frac{\frac{2.3}{n} - \frac{1.5}{n}}{\frac{2.5}{n} - \frac{1.5}{n}} \right) \left( y_{\frac{2.5}{n}} - y_{\frac{1.5}{n}} \right).$$

- 3** `quantile` assigns the first and last values of  $X$  to the quantiles for probabilities less than  $(0.5/n)$  and greater than  $([n-0.5]/n)$ , respectively.

## References

- [1] Langford, E. “Quartiles in Elementary Statistics”, *Journal of Statistics Education*. Vol. 14, No. 3, 2006.

## See Also

`median` | `prctile` | `quantile`

## Related Examples

- “Exploratory Analysis of Data” on page 3-11



## Exploratory Analysis of Data

This example shows how to explore the distribution of data using descriptive statistics.

### Generate sample data.

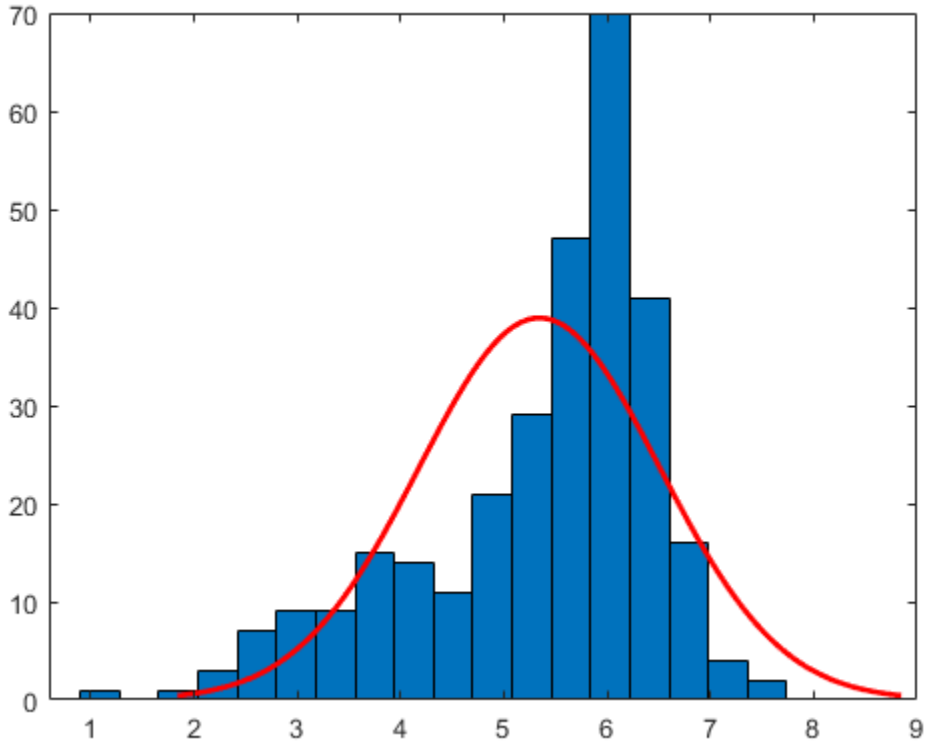
Generate a vector containing randomly-generated sample data.

```
rng default % For reproducibility
x = [normrnd(4,1,1,100),normrnd(6,0.5,1,200)];
```

### Plot a histogram.

Plot a histogram of the sample data with a normal density fit. This provides a visual comparison of the sample data and a normal distribution fitted to the data.

```
histfit(x)
```

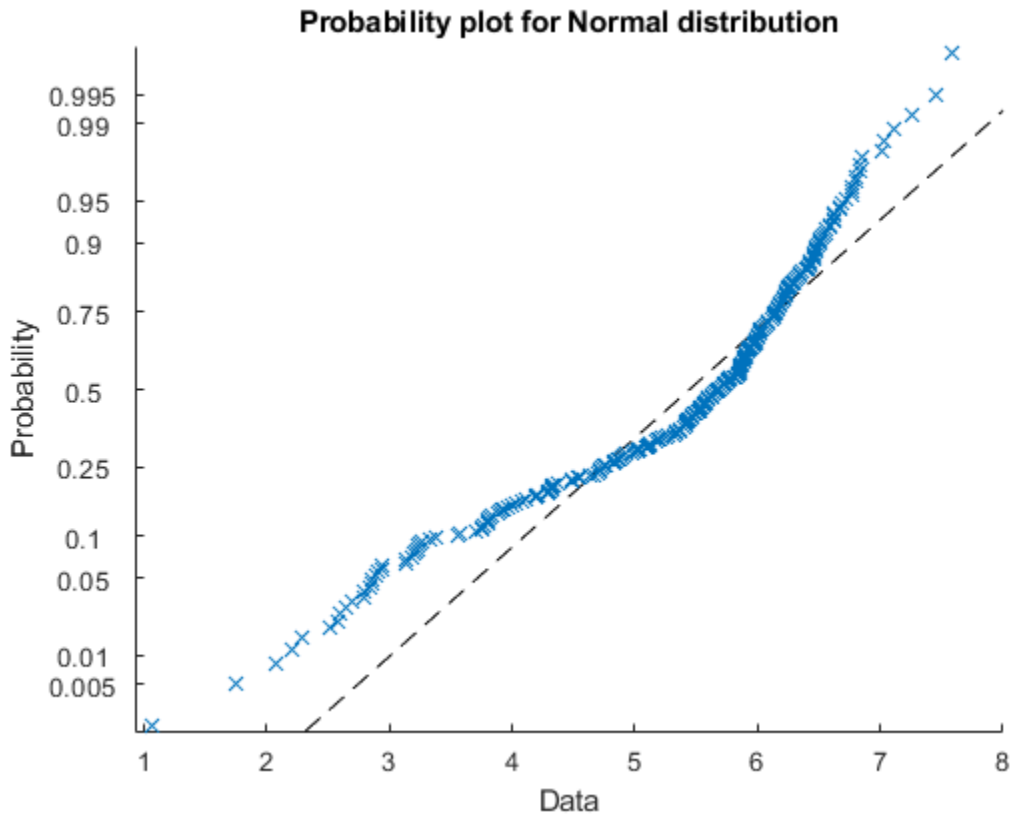


The distribution of the data appears to be left skewed. A normal distribution does not look like a good fit for this sample data.

#### **Obtain a normal probability plot.**

Obtain a normal probability plot. This plot provides another way to visually compare the sample data to a normal distribution fitted to the data.

```
probplot('normal', x)
```



The probability plot also shows the deviation of data from normality.

**Compute the quantiles.**

Compute the quantiles of the sample data.

```
p = 0:0.25:1;
y = quantile(x,p);
z = [p;y]
```

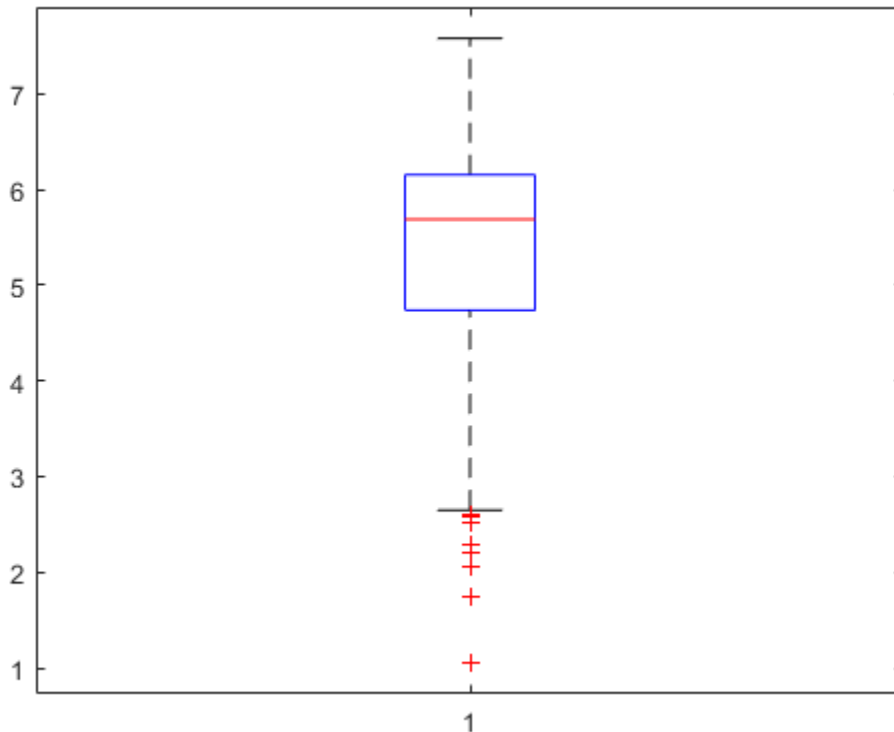
z =

```
0      0.2500      0.5000      0.7500      1.0000
```

1.0557    4.7375    5.6872    6.1526    7.5784

Create a box plot to visualize the statistics.

```
boxplot(x)
```



The box plot shows the 0.25, 0.5, and 0.75 quantiles. The long lower tail and plus signs show the lack of symmetry in the sample data values.

**Compute descriptive statistics.**

Compute the mean and median of the data.

```
y = [mean(x), median(x)]
```

```
y =  
    5.3438    5.6872
```

The mean and median values seem close to each other, but a mean smaller than the median usually indicates that the data is left skewed.

Compute the skewness and kurtosis of the data.

```
y = [skewness(x), kurtosis(x)]  
  
y =  
    -1.0417    3.5895
```

A negative skewness value means the data is left skewed. The data has a larger peakedness than a normal distribution because the kurtosis value is greater than 3.

**Compute z-scores.**

Identify possible outliers by computing the z-scores and finding the values that are greater than 3 or less than -3.

```
Z = zscore(x);  
find(abs(Z)>3);
```

Based on the z-scores, the 3rd and 35th observations might be outliers.

## See Also

`boxplot` | `histfit` | `kurtosis` | `mean` | `median` | `prctile` | `quantile` | `skewness`

## More About

- “Box Plots” on page 4-7
- “Measures of Central Tendency” on page 3-3
- “Measures of Dispersion” on page 3-5

- “Quantiles and Percentiles” on page 3-7

## Resampling Statistics

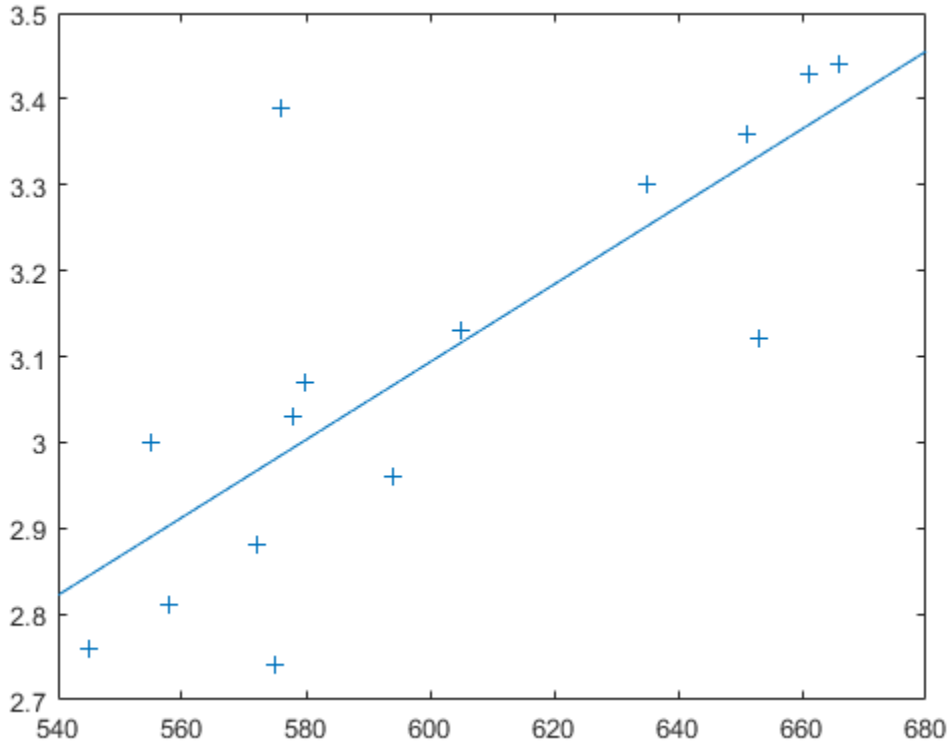
In this section...
“Bootstrap Resampling” on page 3-17
“Jackknife Resampling” on page 3-20
“Parallel Computing Support for Resampling Methods” on page 3-21

### Bootstrap Resampling

The bootstrap procedure involves choosing random samples with replacement from a data set and analyzing each sample the same way. Sampling with replacement means that each observation is selected separately at random from the original dataset. So a particular data point from the original data set could appear multiple times in a given bootstrap sample. The number of elements in each bootstrap sample equals the number of elements in the original data set. The range of sample estimates you obtain enables you to establish the uncertainty of the quantity you are estimating.

This example from Efron and Tibshirani compares Law School Admission Test (LSAT) scores and subsequent law school grade point average (GPA) for a sample of 15 law schools.

```
load lawdata
plot(lsat, gpa, '+')
lsline
```



The least-squares fit line indicates that higher LSAT scores go with higher law school GPAs. But how certain is this conclusion? The plot provides some intuition, but nothing quantitative.

You can calculate the correlation coefficient of the variables using the `|corr|` function.

```
rhohat = corr(lsat,gpa)
```

```
rhohat =
```

```
0.7764
```



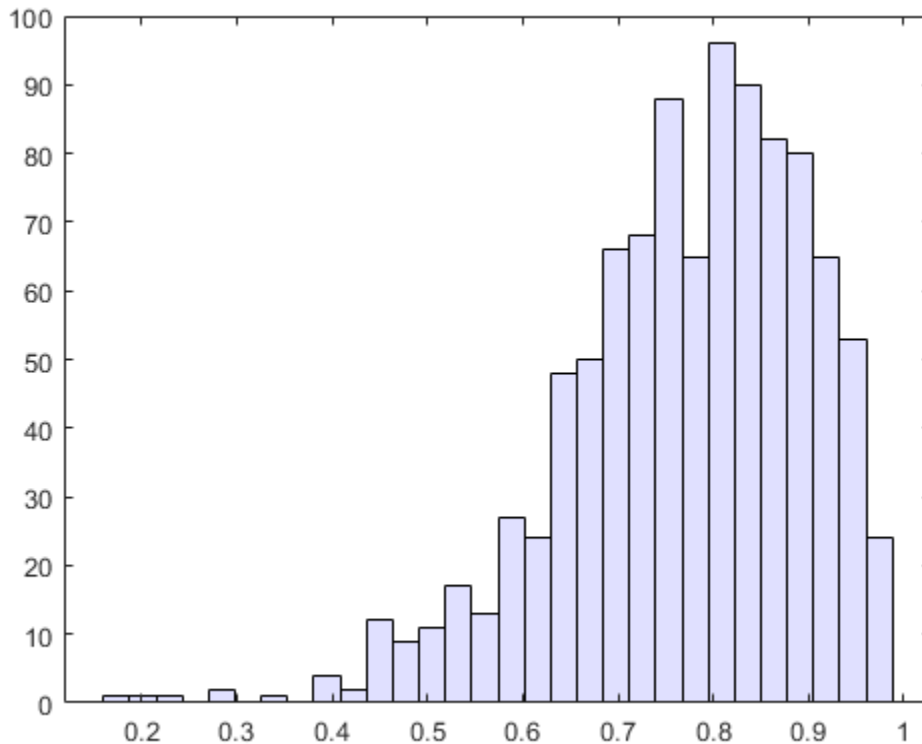
Now you have a number describing the positive connection between LSAT and GPA; though it may seem large, you still do not know if it is statistically significant.

Using the `bootstrap` function you can resample the `lsat` and `gpa` vectors as many times as you like and consider the variation in the resulting correlation coefficients.

```
rng default % For reproducibility
rhos1000 = bootstrap(1000, 'corr', lsat, gpa);
```

This resamples the `lsat` and `gpa` vectors 1000 times and computes the `corr` function on each sample. You can then plot the result in a histogram.

```
histogram(rhos1000, 30, 'FaceColor', [.8 .8 1])
```



Nearly all the estimates lie on the interval [0.4 1.0].

It is often desirable to construct a confidence interval for a parameter estimate in statistical inferences. Using the `bootci` function, you can use bootstrapping to obtain a confidence interval for the `lsat` and `gpa` data.

```
ci = bootci(5000,@corr,lsat,gpa)
```

```
ci =  
  
    0.3319  
    0.9427
```

Therefore, a 95% confidence interval for the correlation coefficient between LSAT and GPA is [0.33 0.94]. This is strong quantitative evidence that LSAT and subsequent GPA are positively correlated. Moreover, this evidence does not require any strong assumptions about the probability distribution of the correlation coefficient.

Although the `bootci` function computes the Bias Corrected and accelerated (BCa) interval as the default type, it is also able to compute various other types of bootstrap confidence intervals, such as the studentized bootstrap confidence interval.

## Jackknife Resampling

Similar to the bootstrap is the jackknife, which uses resampling to estimate the bias of a sample statistic. Sometimes it is also used to estimate standard error of the sample statistic. The jackknife is implemented by the Statistics and Machine Learning Toolbox™ function `jackknife`.

The jackknife resamples systematically, rather than at random as the bootstrap does. For a sample with  $n$  points, the jackknife computes sample statistics on  $n$  separate samples of size  $n-1$ . Each sample is the original data with a single observation omitted.

In the bootstrap example, you measured the uncertainty in estimating the correlation coefficient. You can use the jackknife to estimate the bias, which is the tendency of the sample correlation to over-estimate or under-estimate the true, unknown correlation. First compute the sample correlation on the data.

```
load lawdata  
rho_hat = corr(lsat,gpa)
```

```
rhohat =  
    0.7764
```

Next compute the correlations for jackknife samples, and compute their mean.

```
rng default; % For reproducibility  
jackrho = jackknife(@corr,lsat,gpa);  
meanrho = mean(jackrho)
```

```
meanrho =  
    0.7759
```

Now compute an estimate of the bias.

```
n = length(lsat);  
biasrho = (n-1) * (meanrho-rhohat)
```

```
biasrho =  
   -0.0065
```

The sample correlation probably underestimates the true correlation by about this amount.

## Parallel Computing Support for Resampling Methods

For information on computing resampling statistics in parallel, see Parallel Computing Toolbox™.

## Data with Missing Values

Many data sets have one or more missing values. It is convenient to code missing values as NaN (Not a Number) to preserve the structure of data sets across multiple variables and observations.

Normal MATLAB arithmetic operations yield NaN values when operands are NaN. Removing the NaN values would destroy the matrix structure. Removing the rows containing the NaN values would discard data. Statistics and Machine Learning Toolbox functions in the following table remove NaN values only for the purposes of computation.

Function	Description
nancov	Covariance matrix, ignoring NaN values
nanmax	Maximum, ignoring NaN values
nanmean	Mean, ignoring NaN values
nanmedian	Median, ignoring NaN values
nanmin	Minimum, ignoring NaN values
nanstd	Standard deviation, ignoring NaN values
nansum	Sum, ignoring NaN values
nanvar	Variance, ignoring NaN values

Other Statistics and Machine Learning Toolbox functions also ignore NaN values. These include `iqr`, `kurtosis`, `mad`, `prctile`, `range`, `skewness`, and `trimmean`.

### Working with Data with Missing Values

Create a 3-by-3 matrix of sample data. Remove two data values by replacing them with NaN.

```
X = magic(3);  
X([1 5]) = [NaN NaN]
```

X =

```
NaN     1     6  
  3   NaN     7
```

```
4     9     2
```

Compute the sum of for each column of the sample data matrix using the `sum` function.

```
s1 = sum(X)
```

```
s1 =
```

```
NaN    NaN    15
```

If a column contains a `NaN` value, then the `sum` function will return `NaN` as the sum of the data in that column.

For comparison, compute the sum for each column of the sample data matrix using the `nansum` function.

```
s2 = nansum(X)
```

```
s2 =
```

```
7     10    15
```

If a column contains a `NaN` value, then the `nansum` function ignores the `NaN` value and returns the sum of the remaining values in the column.



# Statistical Visualization

---

- “Introduction to Statistical Visualization” on page 4-2
- “Create Scatter Plots Using Grouped Data” on page 4-3
- “Box Plots” on page 4-7
- “Distribution Plots” on page 4-10

# Introduction to Statistical Visualization

Statistics and Machine Learning Toolbox data visualization functions add to the extensive graphics capabilities already in MATLAB.

- Scatter plots on page 4-3 are a basic visualization tool for multivariate data. They are used to identify relationships among variables. Grouped versions of these plots use different plotting symbols to indicate group membership. The `gname` function is used to label points on these plots with a text label or an observation number.
- Box plots on page 4-7 display a five-number summary of a set of data: the median, the two ends of the interquartile range (the box), and two extreme values (the whiskers) above and below the box. Because they show less detail than histograms, box plots are most useful for side-by-side comparisons of two distributions.
- Distribution plots on page 4-10 help you identify an appropriate distribution family for your data. They include normal and Weibull probability plots, quantile-quantile plots, and empirical cumulative distribution plots.

Advanced Statistics and Machine Learning Toolbox visualization functions are available for specialized statistical analyses.



## Create Scatter Plots Using Grouped Data

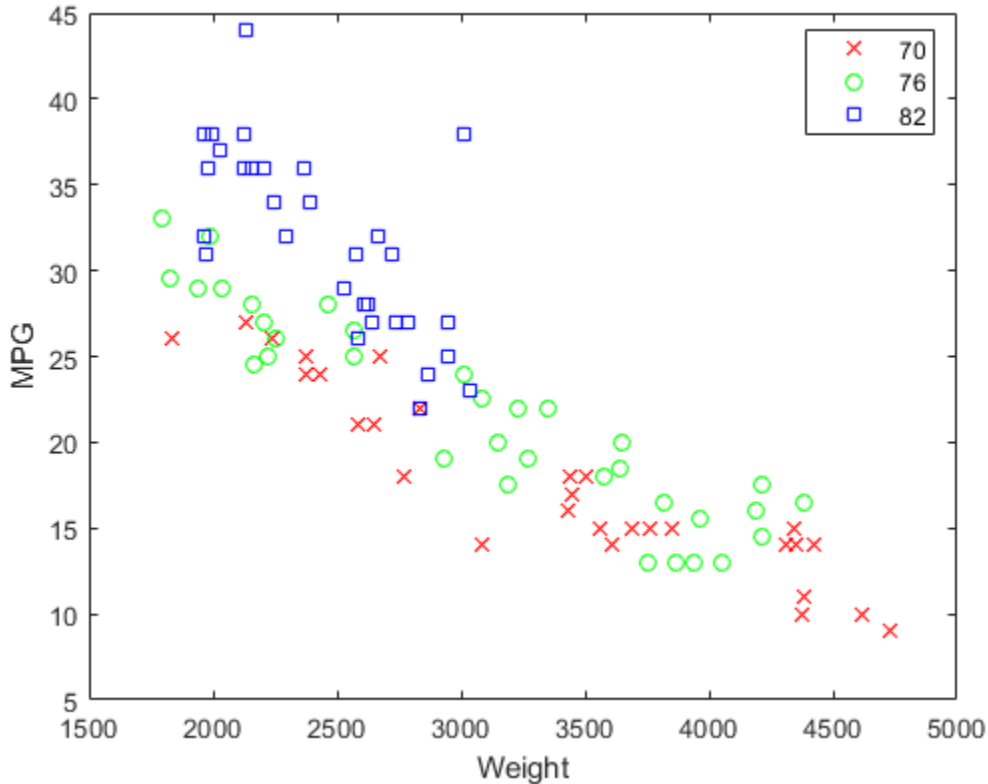
This example shows how to create scatter plots using grouped sample data.

A scatter plot is a simple plot of one variable against another. The MATLAB® functions `plot` and `scatter` produce scatter plots. The MATLAB function `plotmatrix` can produce a matrix of such plots showing the relationship between several pairs of variables.

Statistics and Machine Learning Toolbox™ functions `gscatter` and `gplotmatrix` produce grouped versions of these plots. These are useful for determining whether the values of two variables or the relationship between those variables is the same in each group.

Suppose you want to examine the weight and mileage of cars from three different model years.

```
load carsmall
gscatter(Weight,MPG,Model_Year,'','xos')
```

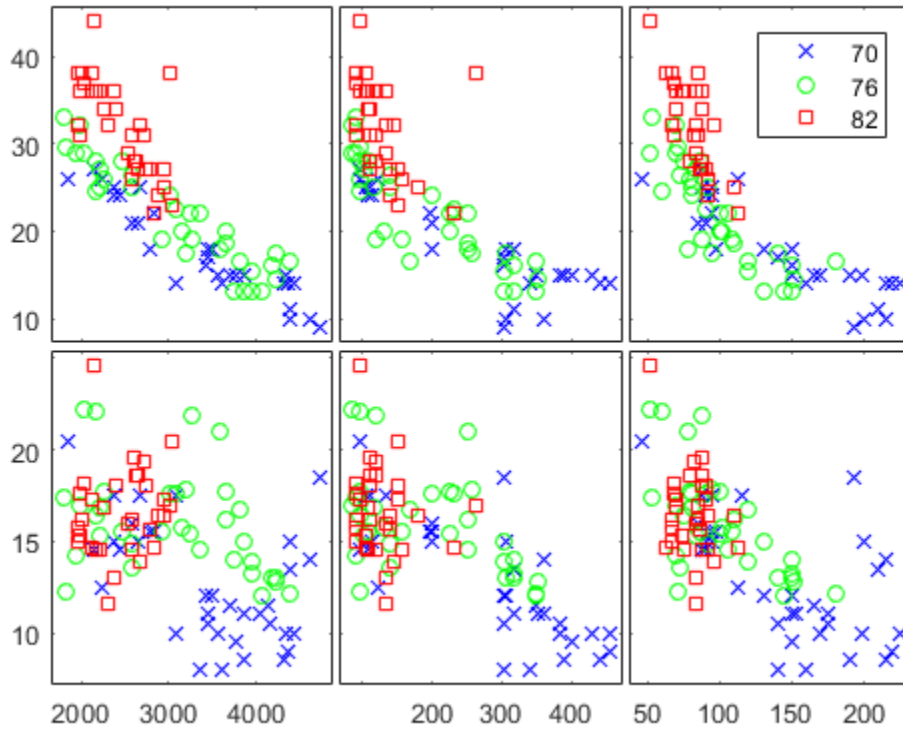


This shows that not only is there a strong relationship between the weight of a car and its mileage, but also that newer cars tend to be lighter and have better gas mileage than older cars.

The default arguments for `gscatter` produce a scatter plot with the different groups shown with the same symbol but different colors. The last two arguments above request that all groups be shown in default colors and with different symbols.

The `carsmall` data set contains other variables that describe different aspects of cars. You can examine several of them in a single display by creating a grouped plot matrix.

```
xvars = [Weight Displacement Horsepower];
yvars = [MPG Acceleration];
gplotmatrix(xvars,yvars,Model_Year, '', 'xos')
```



The upper right subplot displays MPG against Horsepower, and shows that over the years the horsepower of the cars has decreased but the gas mileage has improved.

The `gplotmatrix` function can also graph all pairs from a single list of variables, along with histograms for each variable. See MANOVA.

## See Also

`gplotmatrix` | `gscatter`

## **More About**

- “Grouping Variables” on page 2-59

## Box Plots

Box plots provide a visualization of summary statistics for sample data and contain the following features:

- The tops and bottoms of each “box” are the 25th and 75th percentiles of the samples, respectively. The distances between the tops and bottoms are the interquartile ranges. You can compute the value of the interquartile range using `iqr`.
- The line in the middle of each box is the sample median. If the median is not centered in the box, it shows sample skewness. You can compute the value of the median using the `median` function.
- The whiskers are lines extending above and below each box. Whiskers are drawn from the ends of the interquartile ranges to the furthest observations within the whisker length (the *adjacent values*).
- Observations beyond the whisker length are marked as outliers. By default, an outlier is a value that is more than 1.5 times the interquartile range away from the top or bottom of the box, but this value can be adjusted with additional input arguments. Outliers are displayed with a red + sign.
- Notches display the variability of the median between samples. The width of a notch is computed so that box plots whose notches do not overlap (as above) have different medians at the 5% significance level. The significance level is based on a normal distribution assumption, but comparisons of medians are reasonably robust for other distributions. Comparing box-plot medians is like a visual hypothesis test, analogous to the *t* test used for means.

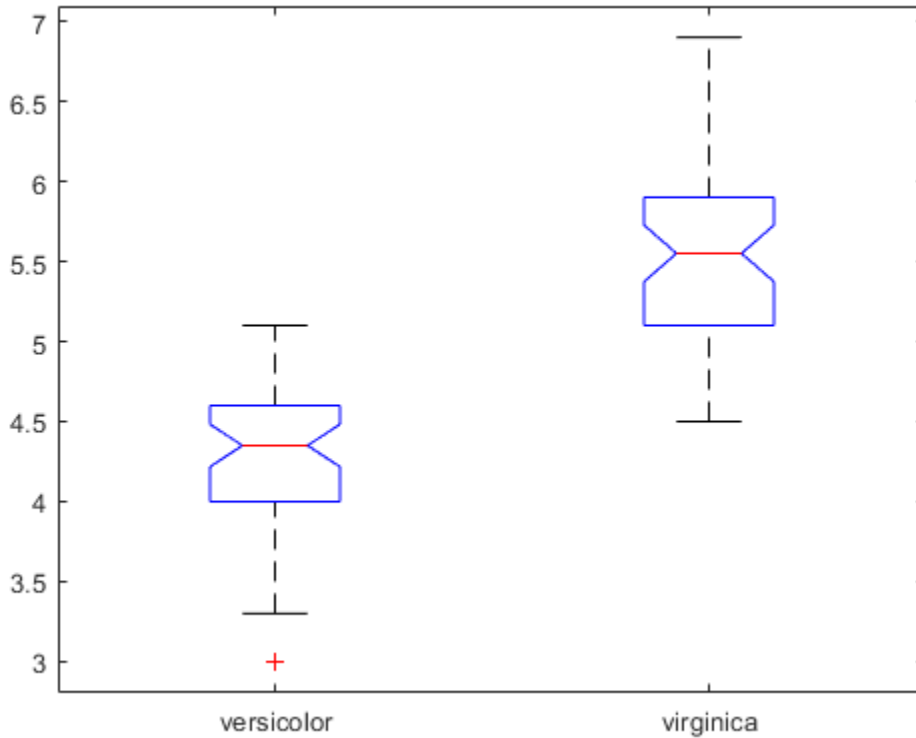
### Compare Grouped Data Using Box Plots

Load the Fisher iris sample data. The data contains length and width measurements from the sepals and petals of three species of iris flowers. Store the petal length data for the versicolor irises as `s1`, and the petal length data for the virginica irises as `s2`.

```
load fisheriris
s1 = meas(51:100, 3);
s2 = meas(101:150, 3);
```

Create a box plot using the sample data. Include a notch on the plot and label each box with the name of the iris species it represents.

```
figure
boxplot([s1 s2], 'notch', 'on', ...
        'labels', {'versicolor', 'virginica'})
```



The notches of the two box plots do not overlap, which indicates that the median petal length of the versicolor and virginica irises are significantly different at the 5% significance level.

The median line in the versicolor plot does not appear to be centered inside the box, which indicates that the sample is slightly skewed. Additionally, the versicolor data contains one outlier value, while the virginica data does not contain any outliers.

## See Also

`boxplot` | `iqr` | `median`

## Related Examples

- “Exploratory Analysis of Data” on page 3-11

## More About

- “Measures of Central Tendency” on page 3-3
- “Measures of Dispersion” on page 3-5
- “Quantiles and Percentiles” on page 3-7

## Distribution Plots

In this section...
“Overview” on page 4-10
“Normal Probability Plots” on page 4-10
“Quantile-Quantile Plots” on page 4-13
“Cumulative Distribution Plots” on page 4-15
“Other Probability Plots” on page 4-17

### Overview

*Distribution plots* visually assess the distribution of sample data by comparing the empirical distribution of the data with the theoretical values expected from a specified distribution. Use distribution plots in addition to more formal hypothesis tests to determine whether the sample data comes from a specified distribution.

Statistics and Machine Learning Toolbox offers several distribution plot options:

- *Normal probability plots* assess whether sample data comes from a normal distribution. For syntax options, see `normplot`.
- *Quantile-quantile (q-q) plots* assess whether two sets of sample data come from the same distribution family, and is robust with respect to differences in location and scale. For syntax options, see `qqplot`.
- *Cumulative distribution plots* display the empirical cumulative distribution function (cdf) of the sample data for visual comparison to the theoretical cdf of a specified distribution. For syntax options, see `cdfplot`, `ecdf`, and `stairs`.

You can create distribution plots for distributions other than normal, or explore the distribution of censored data, using `probplot`.

### Normal Probability Plots

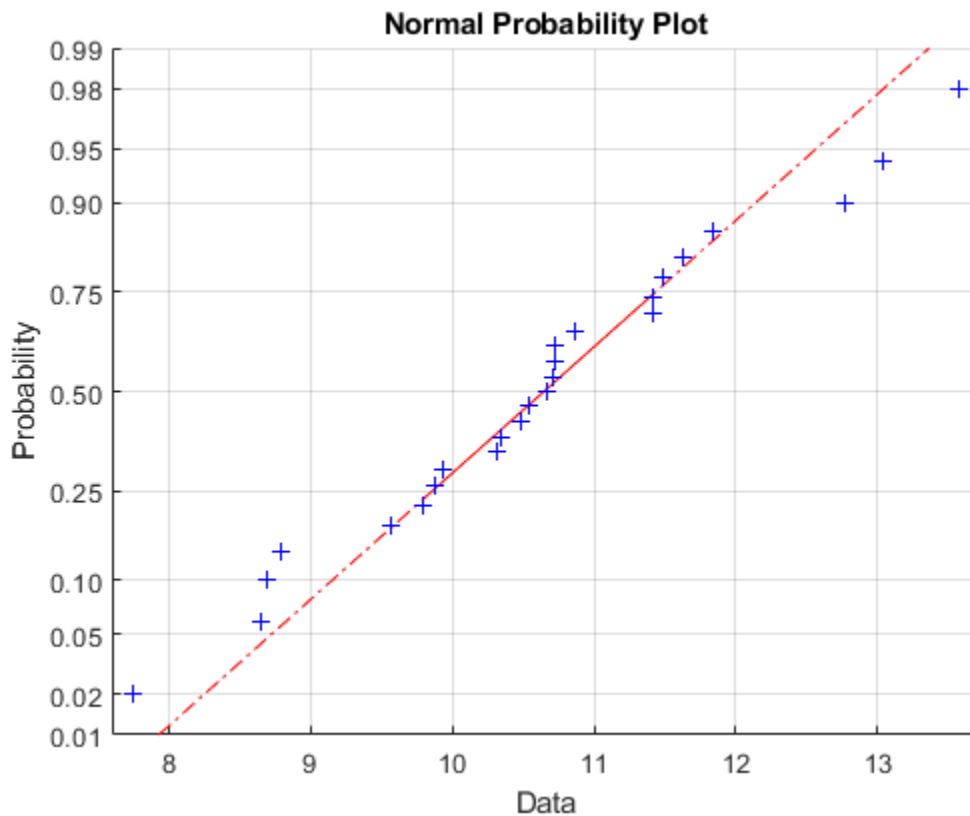
Normal probability plots are used to assess whether data comes from a normal distribution. Many statistical procedures make the assumption that an underlying distribution is normal, so normal probability plots can provide some assurance that the assumption is justified, or else provide a warning of problems with the assumption. An



analysis of normality typically combines normal probability plots with hypothesis tests for normality.

This example generates a data sample of 25 random numbers from a normal distribution with  $\mu = 10$  and  $\sigma = 1$ , and creates a normal probability plot of the data.

```
rng default; % For reproducibility
x = normrnd(10,1,25,1);
normplot(x)
```

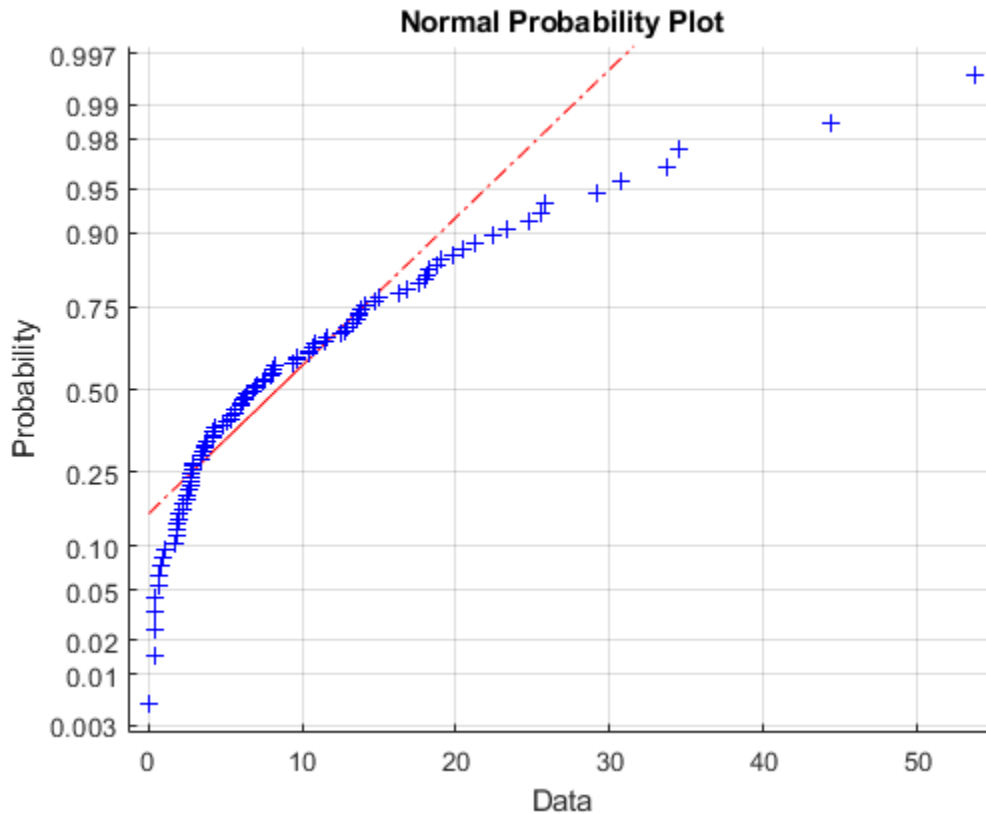


The plus signs plot the empirical probability versus the data value for each point in the data. A solid line connects the 25th and 75th percentiles in the data, and a dashed line extends it to the ends of the data. The  $y$ -axis values are probabilities from zero to one, but the scale is not linear. The distance between tick marks on the  $y$ -axis matches the

distance between the quantiles of a normal distribution. The quantiles are close together near the median (probability = 0.5) and stretch out symmetrically as you move away from the median.

In a normal probability plot, if all the data points fall near the line, an assumption of normality is reasonable. Otherwise, the points will curve away from the line, and an assumption of normality is not justified. For example, the following generates a data sample of 100 random numbers from an exponential distribution with  $\mu = 10$ , and creates a normal probability plot of the data.

```
x = exprnd(10,100,1);  
normplot(x)
```



The plot is strong evidence that the underlying distribution is not normal.

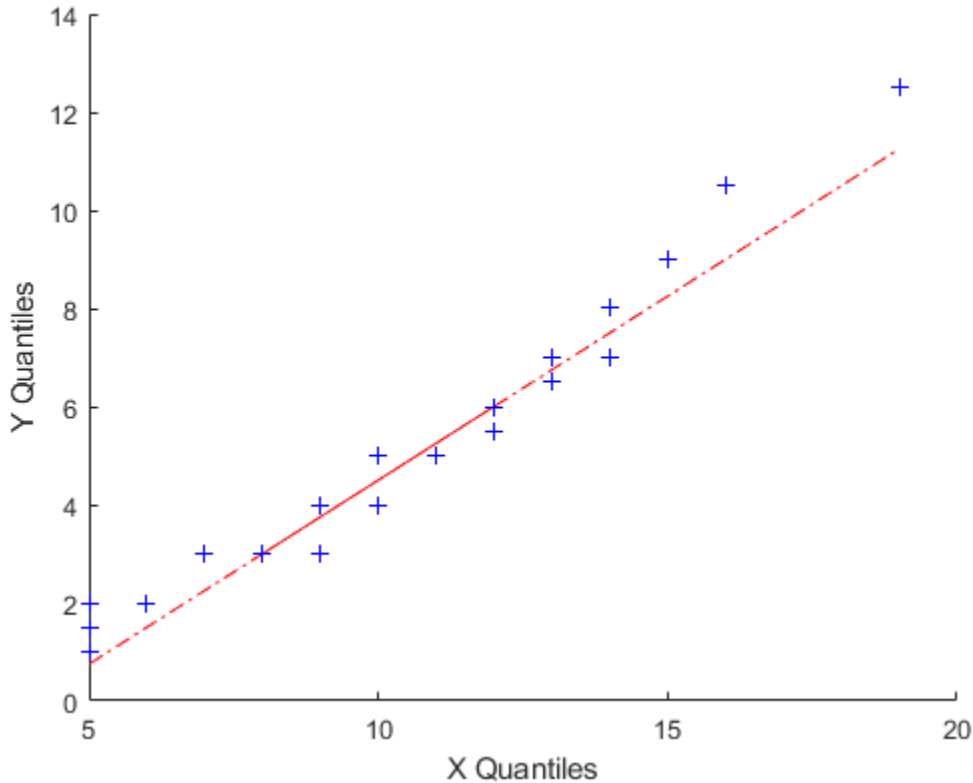
## Quantile-Quantile Plots

Quantile-quantile plots are used to determine whether two samples come from the same distribution family. They are scatter plots of quantiles computed from each sample, with a line drawn between the first and third quartiles. If the data falls near the line, it is reasonable to assume that the two samples come from the same distribution. The method is robust with respect to changes in the location and scale of either distribution.

To create a quantile-quantile plot, use the `qqplot` function.

The following example generates two data samples containing random numbers from Poisson distributions with different parameter values, and creates a quantile-quantile plot. The data in `x` is from a Poisson distribution with `lambda = 10`, and the data in `y` is from a Poisson distribution with `lambda = 5`.

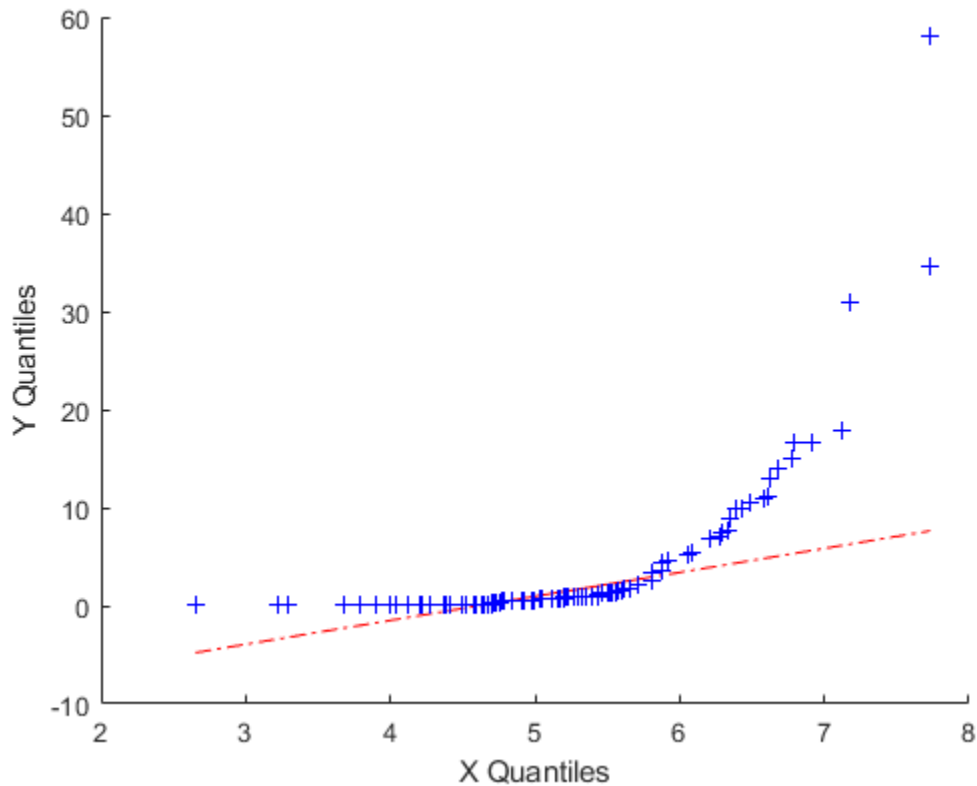
```
x = poissrnd(10,50,1);  
y = poissrnd(5,100,1);  
qqplot(x,y);
```



Even though the parameters and sample sizes are different, the approximate linear relationship suggests that the two samples may come from the same distribution family. As with normal probability plots, hypothesis tests can provide additional justification for such an assumption. For statistical procedures that depend on the two samples coming from the same distribution, however, a linear quantile-quantile plot is often sufficient.

The following example shows what happens when the underlying distributions are not the same. Here,  $x$  contains 100 random numbers generated from a normal distribution with  $\mu = 5$  and  $\sigma = 1$ , while  $y$  contains 100 random numbers generated from a Weibull distribution with  $A = 2$  and  $B = 0.5$ .

```
x = normrnd(5,1,100,1);  
y = wblrnd(2,0.5,100,1);  
qqplot(x,y);
```



These samples clearly are not from the same distribution family.

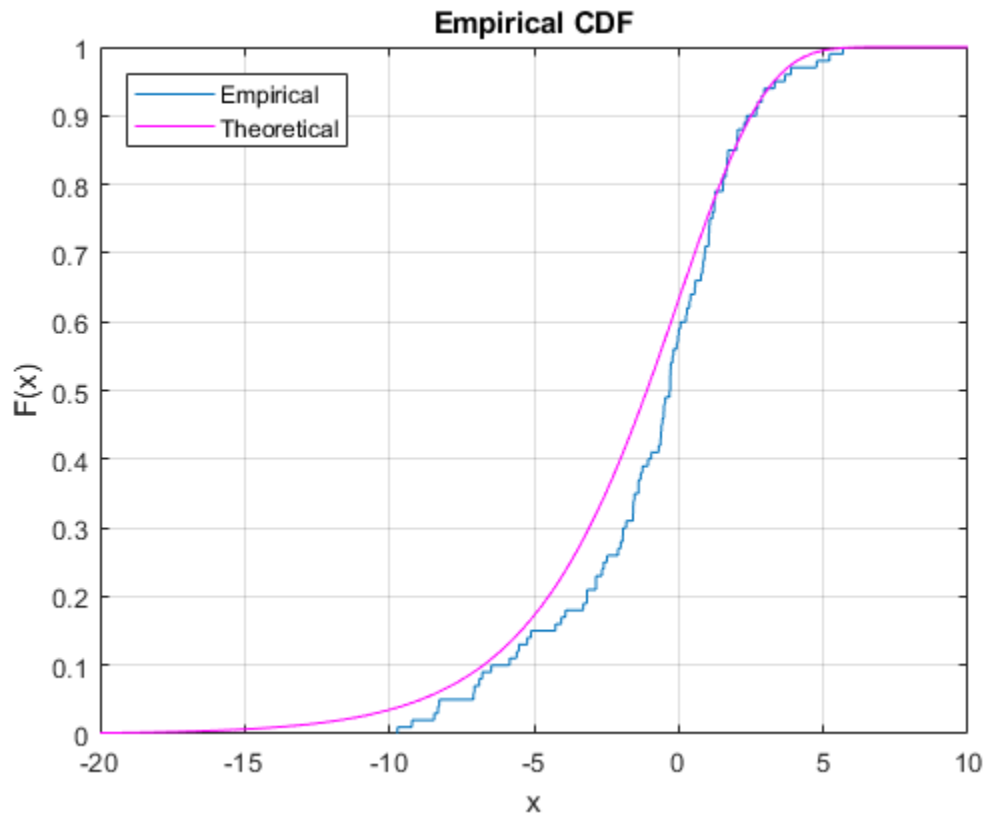
## Cumulative Distribution Plots

An empirical cumulative distribution function (cdf) plot shows the proportion of data less than each  $x$  value, as a function of  $x$ . The scale on the  $y$ -axis is linear; in particular, it is not scaled to any particular distribution. Empirical cdf plots are used to compare data cdfs to cdfs for particular distributions.

To create an empirical cdf plot, use the `cdfplot` function (or `ecdf` and `stairs`).

The following example compares the empirical cdf for a sample from an extreme value distribution with a plot of the cdf for the sampling distribution. In practice, the sampling distribution would be unknown, and would be chosen to match the empirical cdf.

```
y = evrnd(0,3,100,1);  
cdfplot(y)  
hold on  
x = -20:0.1:10;  
f = evcdf(x,0,3);  
plot(x,f,'m')  
legend('Empirical','Theoretical','Location','NW')
```



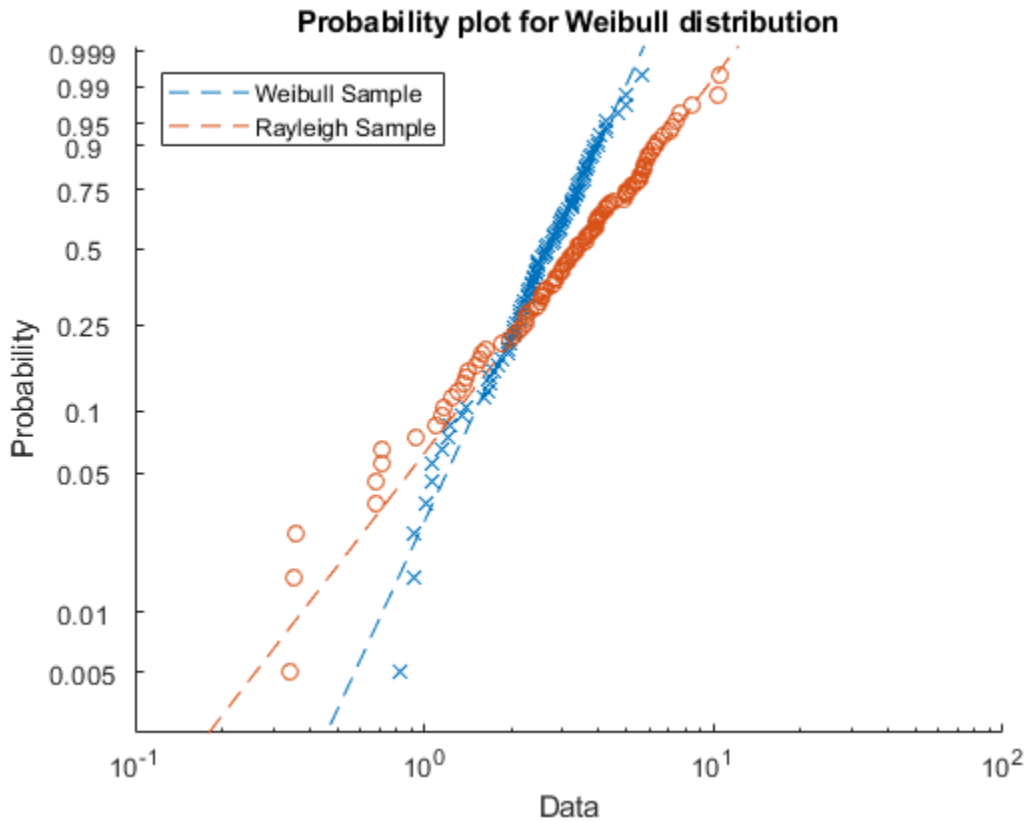
## Other Probability Plots

A probability plot, like the normal probability plot, is just an empirical cdf plot scaled to a particular distribution. The  $y$ -axis values are probabilities from zero to one, but the scale is not linear. The distance between tick marks is the distance between quantiles of the distribution. In the plot, a line is drawn between the first and third quartiles in the data. If the data falls near the line, it is reasonable to choose the distribution as a model for the data.

To create probability plots for different distributions, use the `probplot` function.

The following example assesses two samples, one from a Weibull distribution with  $A = 3$  and  $B = 3$ , and one from a Rayleigh distribution with  $B = 3$ , to see if either distribution may have come from a Weibull population.

```
x1 = wblrnd(3,3,100,1);  
x2 = raylrnd(3,100,1);  
probplot('weibull',[x1 x2])  
legend('Weibull Sample','Rayleigh Sample','Location','NW')
```



The plot gives justification for modeling the first sample with a Weibull distribution; much less so for the second sample.

A distribution analysis typically combines probability plots with hypothesis tests for a particular distribution.



# Probability Distributions

---

- “Working with Probability Distributions” on page 5-3
- “Supported Distributions” on page 5-17
- “Maximum Likelihood Estimation” on page 5-27
- “Negative Loglikelihood Functions” on page 5-30
- “Random Number Generation” on page 5-34
- “Nonparametric and Empirical Probability Distributions” on page 5-37
- “Fit Kernel Distribution Object to Data” on page 5-46
- “Fit Kernel Distribution Using `ksdensity`” on page 5-51
- “Fit Distributions to Grouped Data Using `ksdensity`” on page 5-54
- “Fit a Nonparametric Distribution with Pareto Tails” on page 5-57
- “Generate Random Numbers Using the Triangular Distribution” on page 5-62
- “Explore the Probability Distribution Function UI” on page 5-67
- “Model Data Using the Distribution Fitter App” on page 5-70
- “Fit a Distribution Using the Distribution Fitter App” on page 5-97
- “Custom Distributions Using the Distribution Fitter App” on page 5-109
- “Explore the Random Number Generation UI” on page 5-111
- “Compare Multiple Distribution Fits” on page 5-114
- “Fit Probability Distribution Objects to Grouped Data” on page 5-121
- “Multinomial Probability Distribution Objects” on page 5-125
- “Multinomial Probability Distribution Functions” on page 5-129
- “Generate Random Numbers Using Uniform Distribution Inversion” on page 5-132
- “Represent Cauchy Distribution Using  $t$  Location-Scale” on page 5-136
- “Generate Cauchy Random Numbers Using Student's  $t$ ” on page 5-140
- “Generate Correlated Data Using Rank Correlation” on page 5-142
- “Gaussian Mixture Models” on page 5-148
- “Create a Gaussian Mixture Model” on page 5-150

- “Fit a Gaussian Mixture Model to Data” on page 5-155
- “Simulate Data from a Gaussian Mixture Model” on page 5-160
- “Copulas: Generate Correlated Samples” on page 5-164

## Working with Probability Distributions

In this section...
“Types of Probability Distributions” on page 5-3
“Probability Distribution Objects” on page 5-4
“Probability Distribution Functions” on page 5-8
“Probability Distribution Apps and User Interfaces” on page 5-12

### Types of Probability Distributions

Probability distributions are theoretical distributions based on assumptions about a source population. The distributions assign probability to the event that a random variable has a specific, discrete value, or falls within a specified range of continuous values.

Statistics and Machine Learning Toolbox offers several ways to work with probability distributions.

- Use “Probability Distribution Objects” on page 5-4 to fit a probability distribution object to sample data, or to create a probability distribution object with specified parameter values.
- Use “Probability Distribution Functions” on page 5-8 to work with data input from matrices, tables, and dataset arrays.
- Use “Probability Distribution Apps and User Interfaces” on page 5-12 to interactively fit, explore, and generate random numbers from probability distributions. Available apps and user interfaces include:
  - The **Distribution Fitter** app
  - The **Probability Distribution Function** user interface
  - The Random Number Generation user interface (`randtool`)

For a list of distributions supported by Statistics and Machine Learning Toolbox, see “Supported Distributions” on page 5-17.

## Probability Distribution Objects

Probability distribution objects allow you to fit a probability distribution to sample data, or define a distribution by specifying parameter values. You can then perform a variety of analyses on the distribution object.

### Create Probability Distribution Objects

Estimate probability distribution parameters from sample data by fitting a probability distribution object to the data using `fitdist`. You can fit a single specified parametric or nonparametric distribution to the sample data. You can also fit multiple distributions of the same type to the sample data based on grouping variables. For most distributions, `fitdist` uses maximum likelihood estimation (MLE) to estimate the distribution parameters from the sample data. For more information and additional syntax options, see `fitdist`.

Alternatively, you can create a probability distribution object with specified parameter values using `makedist`.

### Work with Probability Distribution Objects

Once you create a probability distribution object, you can use object functions to:

- Compute confidence intervals for the distribution parameters (`paramci`).
- Compute summary statistics, including mean (`mean`), median (`median`), interquartile range (`iqr`), variance (`var`), and standard deviation (`std`).
- Evaluate the probability density function (`pdf`).
- Evaluate the cumulative distribution function (`cdf`) or the inverse cumulative distribution function (`icdf`).
- Compute the negative log likelihood (`negloglik`) and profile likelihood function (`proflik`) for the distribution.
- Generate random numbers from the distribution (`random`).
- Truncate the distribution to specified lower and upper limits (`truncate`).

### Save a Probability Distribution Object

To save your probability distribution object to a `.MAT` file:

- In the toolbar, click **Save Workspace**. This option saves all of the variables in your workspace, including any probability distribution objects.
- In the workspace browser, right-click the probability distribution object and select **Save as**. This option saves only the selected probability distribution object, not the other variables in your workspace.

Alternatively, you can save a probability distribution object directly from the command line by using the `save` function. `save` enables you to choose a file name and specify the probability distribution object you want to save. If you do not specify an object (or other variable), MATLAB saves all of the variables in your workspace, including any probability distribution objects, to the specified file name. For more information and additional syntax options, see `save`.

### Example

This example shows how to use probability distribution objects to perform a multistep analysis on a fitted distribution.

The following analysis illustrates how to:

- Fit a probability distribution object to sample data that contains 120 students' exam grades, using `fitdist`.
- Compute the mean of the exam grades, using `mean`.
- Plot a histogram of the exam grade data, overlaid with a plot of the pdf of the fitted distribution, using `plot` and `pdf`.
- Compute the boundary for the top 10 percent of student grades, using `icdf`.
- Save the fitted probability distribution object, using `save`.

Load the sample data.

```
load examgrades
```

The sample data contains a 120-by-5 matrix of students' exam grades. The exams are scored on a scale of 0 to 100.

Create a vector containing the first column of students' exam grade data.

```
x = grades(:,1);
```

Fit a normal distribution to the sample data by using `fitdist` to create a probability distribution object.

```
pd = fitdist(x, 'Normal')

pd =

NormalDistribution

Normal distribution
    mu = 75.0083    [73.4321, 76.5846]
    sigma = 8.7202    [7.7391, 9.98843]
```

`fitdist` returns a probability distribution object, `pd`, of the type `NormalDistribution`. This object contains the estimated parameter values, `mu` and `sigma`, for the fitted normal distribution.

Compute the mean of the students' exam grades using the fitted distribution object, `pd`.

```
m = mean(pd)

m =

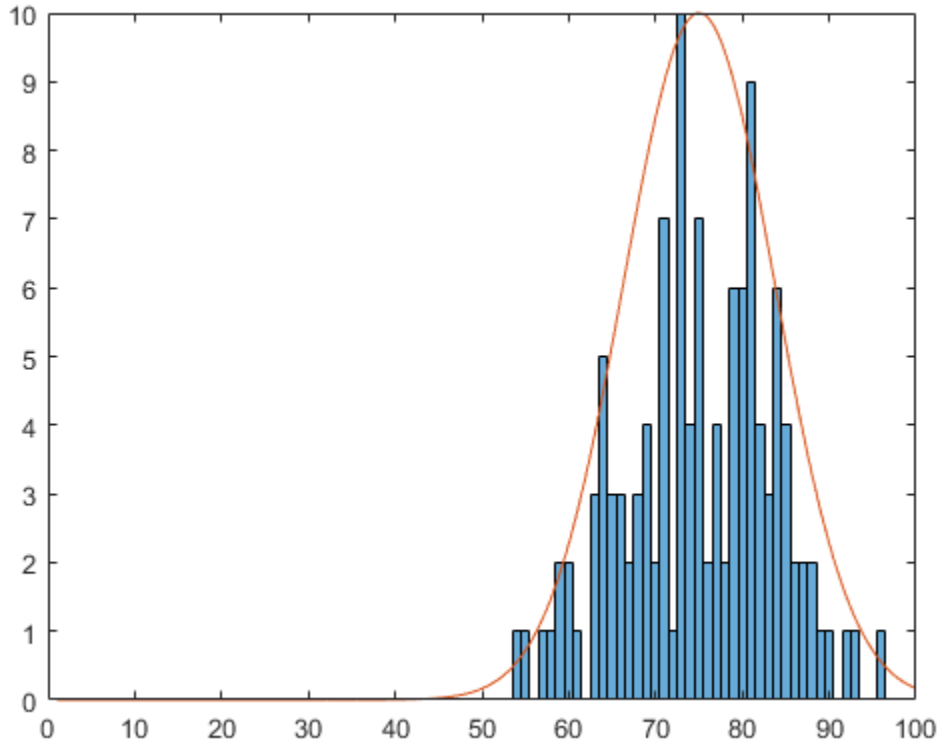
    75.0083
```

The mean of the exam grades is equal to the `mu` parameter estimated by `fitdist`.

Plot a histogram of the exam grades. Overlay a scaled plot of the fitted pdf to visually compare the fitted normal distribution with the actual exam grades.

```
x_pdf = [1:0.1:100];
y = pdf(pd, x_pdf);

figure
histogram(x)
hold on
scale = 10/max(y);
plot((x_pdf), (y.*scale))
hold off
```



The pdf of the fitted distribution follows the same shape as the histogram of the exam grades.

Use the inverse cumulative distribution function (`icdf`) to determine the boundary for the upper 10 percent of student exam grades. This boundary is equivalent to the value at which the cdf of the probability distribution is equal to 0.9. In other words, 90 percent of the exam grades are less than or equal to this boundary value.

```
A = icdf(pd,0.9)
```

```
A =
```

86.1837

Based on the fitted distribution, 10 percent of students received an exam grade greater than 86.1837. Equivalently, 90 percent of students received an exam grade less than or equal to 86.1837.

Save the fitted probability distribution, `pd`, as a file named `myobject.mat`.

```
save myobject.mat pd
```

### Probability Distribution Functions

You can also work with probability distributions using command-line functions. Command-line functions let you further explore parametric and nonparametric distributions, fit relevant models to your data, and generate random data from a specified distribution. For a list of supported probability distributions, see “Supported Distributions” on page 5-17.

Probability distribution functions are useful for generating random numbers and computing summary statistics inside a loop or script, or passing a cdf or pdf as a function handle (MATLAB) to another function. You can also use functions if your desired distribution is not available as a probability distribution object.

#### Examples

This example shows how to use the probability distribution function `normcdf` as a function handle in the chi-square goodness of fit test (`chi2gof`).

This example tests the null hypothesis that the sample data contained in the input vector, `x`, comes from a normal distribution with parameters  $\mu$  and  $\sigma$  equal to the mean (`mean`) and standard deviation (`std`) of the sample data, respectively.

```
rng default
x = normrnd(50,5,100,1);
h = chi2gof(x,'cdf',{@normcdf,mean(x),std(x)})

h =

     0
```



The returned result `h = 0` indicates that `chi2gof` does not reject the null hypothesis at the default 5% significance level.

This next example illustrates how to use probability distribution functions as a function handle in the slice sampler (`slicesample`). The example uses `normpdf` to generate a random sample of 2,000 values from a standard normal distribution, and plots a histogram of the resulting values.

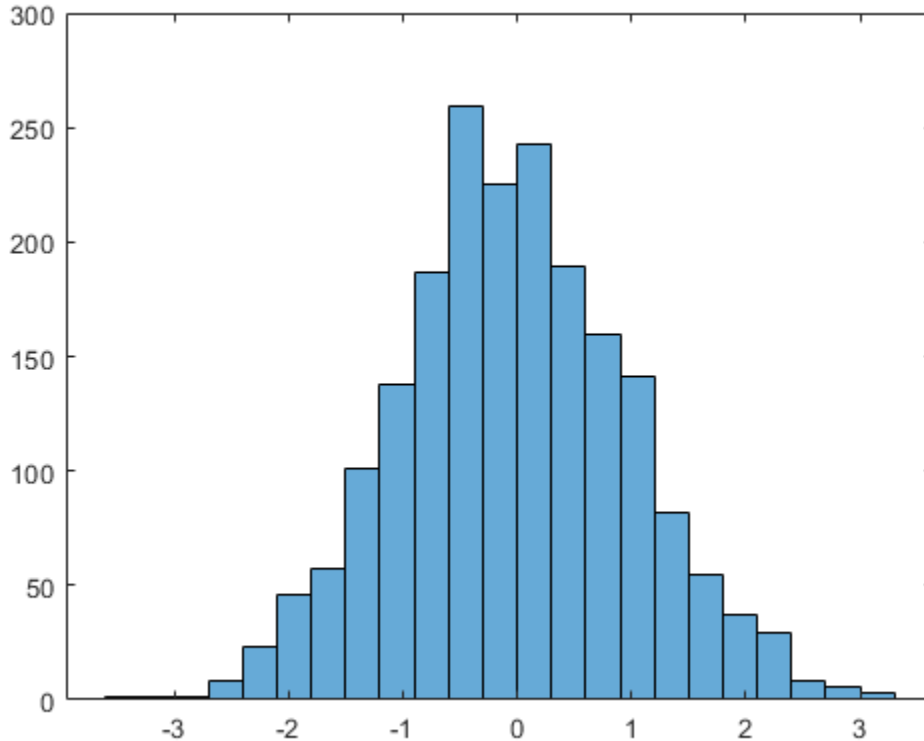
```
rng default
x = slicesample(1,2000,'pdf',@normpdf,'thin',5,'burnin',1000);
h = histogram(x)
```

```
h =
```

```
Histogram with properties:
```

```
    Data: [2000x1 double]
  Values: [1x23 double]
  NumBins: 23
 BinEdges: [1x24 double]
 BinWidth: 0.3000
BinLimits: [-3.6000 3.3000]
Normalization: 'count'
  FaceColor: 'auto'
  EdgeColor: [0 0 0]
```

```
Use GET to show all properties
```



The histogram shows that, when using `normpdf`, the resulting random sample has a standard normal distribution.

If you pass the probability distribution function for the exponential distribution pdf (`exppdf`) as a function handle instead of `normpdf`, then `slicesample` generates the 2,000 random samples from an exponential distribution with a default parameter value of  $\mu$  equal to 1.

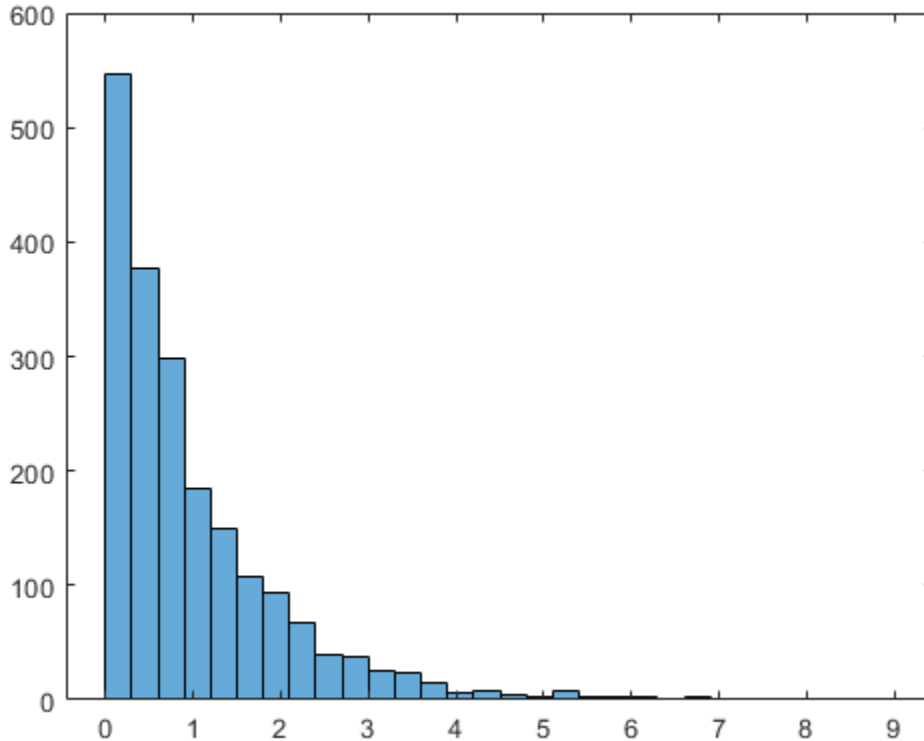
```
rng default
x = slicesample(1,2000,'pdf',@exppdf,'thin',5,'burnin',1000);
h = histogram(x)
```

```
h =
```

Histogram with properties:

```
    Data: [2000x1 double]
  Values: [1x30 double]
  NumBins: 30
  BinEdges: [1x31 double]
  BinWidth: 0.3000
  BinLimits: [0 9.0000]
Normalization: 'count'
  FaceColor: 'auto'
  EdgeColor: [0 0 0]
```

Use GET to show all properties



The histogram shows that the resulting random sample when using `exppdf` has an exponential distribution.

### Probability Distribution Apps and User Interfaces

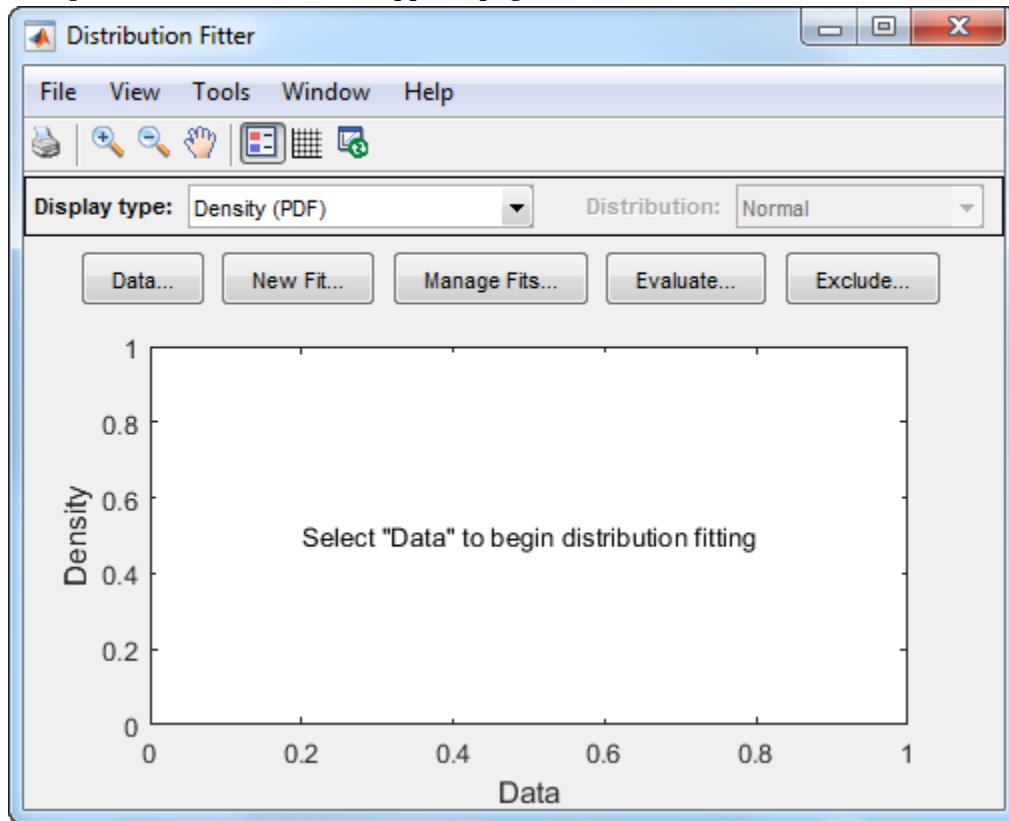
Apps and user interfaces provide an interactive approach to working with parametric and nonparametric probability distributions.

#### Distribution Fitter App

The **Distribution Fitter** app allows you to interactively fit a probability distribution to your data. You can display different types of plots, compute confidence bounds, and evaluate the fit of the data. You can also exclude data from the fit. You can save the data,

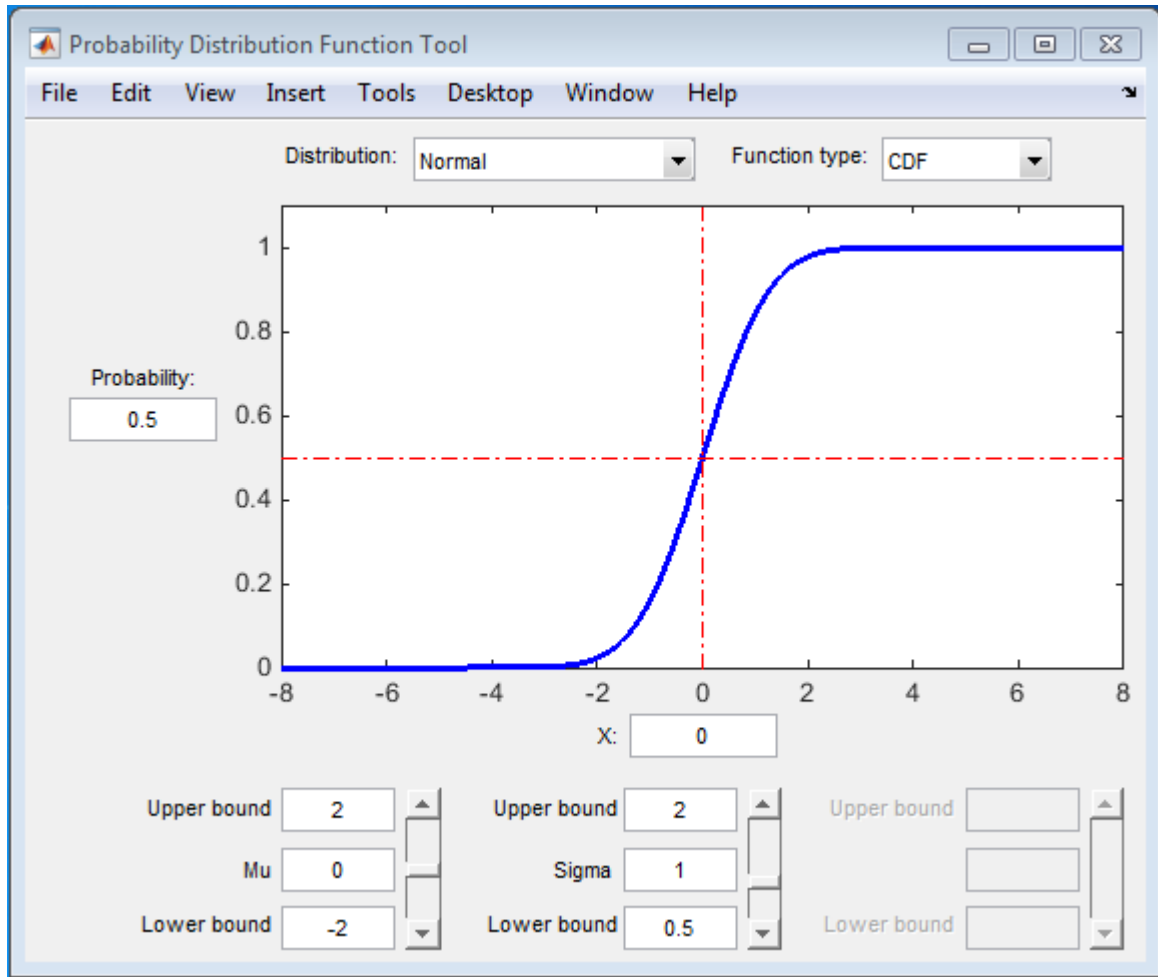
and export the fit to your workspace as a probability distribution object to perform further analysis.

Load the Distribution Fitter app from the Apps tab, or by entering `distributionFitter` in the command window. For more information, see “Model Data Using the Distribution Fitter App” on page 5-70.



### Probability Distribution Function Tool

The **Probability Distribution Function** user interface visually explores probability distributions. You can load the Probability Distribution Function user interface by entering `disttool` in the command window.

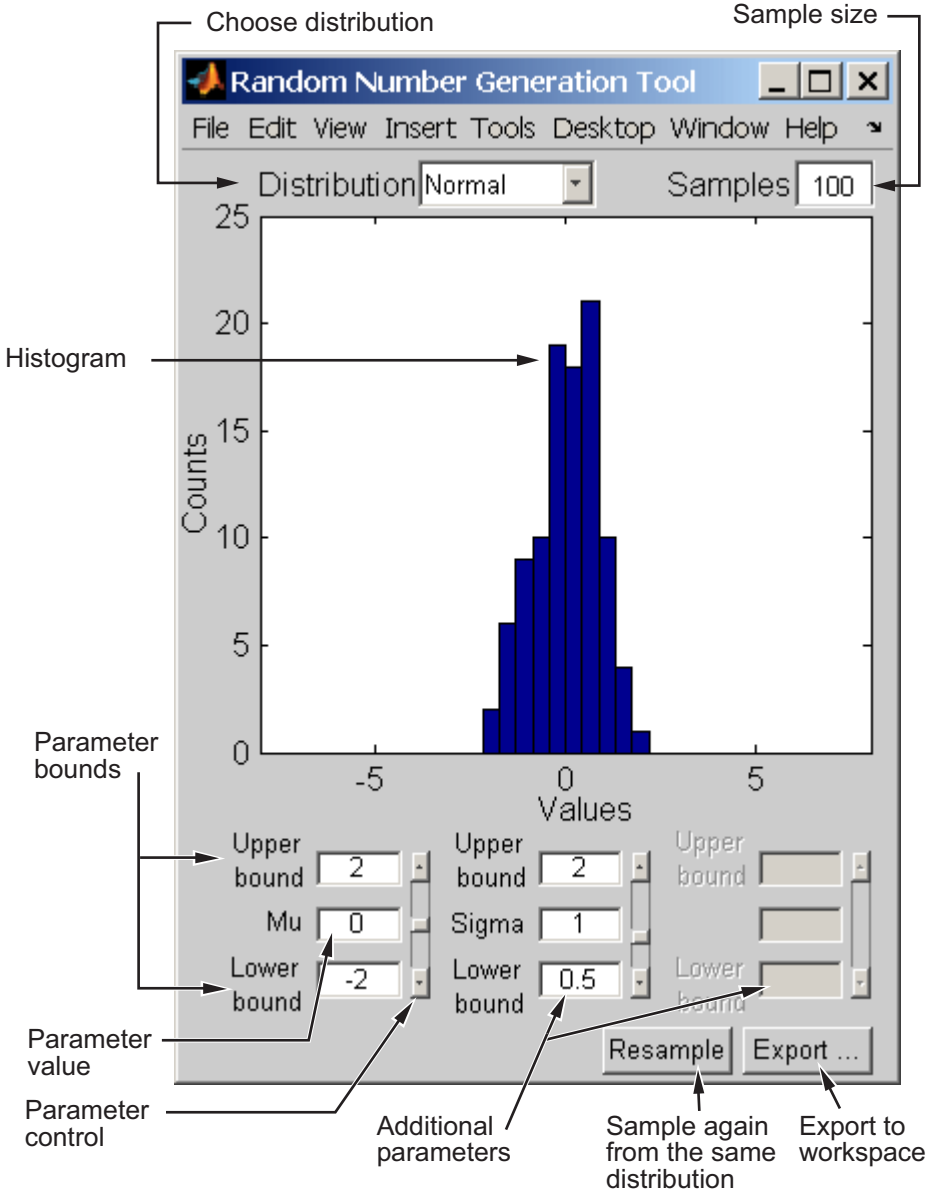


### Random Number Generation Tool

The Random Number Generation user interface generates random data from a specified distribution and exports the results to your workspace. You can use this tool to explore the effects of changing parameters and sample size on the distributions.

The Random Number Generation user interface allows you to set parameter values for the distribution and change their lower and upper bounds; draw another sample from the same distribution, using the same size and parameters; and export the current random

sample to your workspace for use in further analysis. A dialog box enables you to provide a name for the sample.



## See Also

**Distribution Fitter** | **Probability Distribution Function** | `fitdist` | `makedist` | `randtool`

## More About

- “Supported Distributions” on page 5-17



## Supported Distributions

In this section...
“Continuous Distributions (Data)” on page 5-18
“Continuous Distributions (Statistics)” on page 5-22
“Discrete Distributions” on page 5-23
“Multivariate Distributions” on page 5-25
“Nonparametric Distributions” on page 5-26
“Flexible Distribution Families” on page 5-26

Statistics and Machine Learning Toolbox supports more than 30 probability distributions, including parametric, nonparametric, continuous, and discrete distributions.

The toolbox provides several ways to work with probability distributions.

- Use *probability distribution objects* to fit a probability distribution object to sample data, or to create a probability distribution object with specified parameter values. The Using Objects page for each distribution provides information about the object’s properties and the functions you can use to work with the object.
- Use *probability distribution functions* to work with data input from matrices, tables, and dataset arrays. Some of the supported distributions have distribution-specific functions. These functions use the following abbreviations:
  - pdf — Probability density functions
  - cdf — Cumulative distribution functions
  - inv — Inverse cumulative distribution functions
  - stat — Distribution statistics functions
  - fit — Distribution Fitter functions
  - like — Negative log-likelihood functions
  - rnd — Random number generators

You can also use the following generic functions to work with most of the distributions:

- pdf — Probability density function

- `cdf` — Cumulative distribution function
- `icdf` — Inverse cumulative distribution function
- `mle` — Distribution fitting function
- `random` — Random number generating function
- Use *probability distribution apps and user interfaces* to interactively fit, explore, and generate random numbers from probability distributions. Available apps and user interfaces include:
  - The **Distribution Fitter** app, to interactively fit a distribution to sample data, and export a probability distribution object to the workspace.
  - The **Probability Distribution Function** user interface, to visually explore the effect on the pdf and cdf of changing the distribution parameter values.
  - The Random Number Generation user interface (`randtool`), to interactively generate random numbers from a probability distribution with specified parameter values and export them to the workspace.

For more information on the different ways to work with probability distributions, see “Working with Probability Distributions” on page 5-3.

### Continuous Distributions (Data)

Distribution	Using Objects	Legacy Functions	Apps and UIs
Beta on page B-4	<code>BetaDistribution</code>	<code>betapdf</code> <code>betacdf</code> <code>betainv</code> <code>betastat</code> <code>betafit</code> <code>betalike</code> <code>betarnd</code>	<b>Distribution Fitter</b> <b>Probability Distribution Function</b> <code>randtool</code>
Birnbaum-Saunders on page B-13	<code>BirnbaumSaundersDistribution</code>	<code>pdf</code> <code>cdf</code> <code>icdf</code> <code>mle</code> <code>random</code>	<b>Distribution Fitter</b>

Distribution	Using Objects	Legacy Functions	Apps and UIs
Burr Type XII on page B-15	BurrDistribution	pdf cdf icdf mle random	<b>Distribution Fitter</b> <b>Probability Distribution Function</b> randtool
Exponential on page B-34	ExponentialDistribution	exppdf expcdf expinv expstat expfit explike	<b>Distribution Fitter</b> <b>Probability Distribution Function</b> randtool
Extreme value on page B-38	ExtremeValueDistribution	evpdf evcdf evinv evstat evfit evlike evrnd	<b>Distribution Fitter</b> <b>Probability Distribution Function</b> randtool
Gamma on page B-47	GammaDistribution	gampdf gamcdf gaminv gamstat gamfit gamlike gamrnd	<b>Distribution Fitter</b> <b>Probability Distribution Function</b> randtool
Generalized extreme value on page B-53	GeneralizedExtremeValueDistribution	gevpdf gevcdf gevinv gevstat gevfit gevlike gevrnd	<b>Distribution Fitter</b> <b>Probability Distribution Function</b> randtool

Distribution	Using Objects	Legacy Functions	Apps and Uls
Generalized Pareto on page B-58	GeneralizedParetoDistribution	gppdf gpcdf gpinv gpstat gpfit gplike gprnd	<b>Distribution Fitter</b> <b>Probability Distribution Function</b> randtool
“Half-Normal Distribution” on page B-73	HalfNormalDistribution	pdf cdf icdf mle random	<b>Distribution Fitter</b> <b>Probability Distribution Function</b> randtool
Inverse Gaussian on page B-82	InverseGaussianDistribution	pdf cdf icdf mle random	<b>Distribution Fitter</b>
Logistic on page B-96	LogisticDistribution	pdf cdf icdf mle random	<b>Distribution Fitter</b>
Loglogistic on page B-98	LoglogisticDistribution	pdf cdf icdf mle random	<b>Distribution Fitter</b>
Lognormal on page B-100	LognormalDistribution	lognpdf logncdf logninv lognstat lognfit lognlike lognrnd	<b>Distribution Fitter</b> <b>Probability Distribution Function</b> randtool

Distribution	Using Objects	Legacy Functions	Apps and Uls
Nakagami on page B-118	NakagamiDistribution	pdf cdf icdf mle random	<b>Distribution Fitter</b>
Normal (Gaussian) on page B-134	NormalDistribution	normpdf normcdf norminv normstat normfit normlike normrnd	<b>Distribution Fitter Probability Distribution Function randtool</b>
Piecewise linear on page B-141	PiecewiseLinearDistribution	pdf cdf icdf random	
Rayleigh on page B-146	RayleighDistribution	raylpdf raylcdf raylinv raylstat raylfit raylrnd	<b>Distribution Fitter Probability Distribution Function randtool</b>
Rician on page B-149	RicianDistribution	pdf cdf icdf mle random	<b>Distribution Fitter</b>
Stable on page B-151	StableDistribution	pdf cdf icdf mle random	<b>Distribution Fitter Probability Distribution Function randtool</b>
Triangular on page B-178	TriangularDistribution		

Distribution	Using Objects	Legacy Functions	Apps and UIs
Uniform (continuous) on page B-184	UniformDistribution	unifpdf unifcdf unifinv unifstat unifit unifrnd	<b>Probability Distribution Function</b> randtool
Weibull on page B-193	WeibullDistribution	wblpdf wblcdf wblinv wblstat wblfit wbllike wblrnd	<b>Distribution Fitter</b> <b>Probability Distribution Function</b> randtool

### Continuous Distributions (Statistics)

Distribution	Using Objects	Legacy Functions	Apps and UIs
Chi-square on page B-28		chi2pdf chi2cdf chi2inv chi2stat chi2rnd	<b>Probability Distribution Function</b> randtool
<i>F on page B-44</i>		fpdf fcdf finv fstat frnd	<b>Probability Distribution Function</b> randtool
Noncentral chi-square on page B-125		ncx2pdf ncx2cdf ncx2inv ncx2stat ncx2rnd	<b>Probability Distribution Function</b> randtool

Distribution	Using Objects	Legacy Functions	Apps and UIs
Noncentral $F$ on page B-128		ncfpdf ncfcdf ncfinv ncfstat ncfrnd	<b>Probability Distribution Function</b> randtool
Noncentral $t$ on page B-131		nctpdf nctcdf nctinv nctstat nctrnd	<b>Probability Distribution Function</b> randtool
Student's $t$ on page B-166		tpdf tcdf tinv tstat trnd	<b>Probability Distribution Function</b> randtool
$t$ location- scale on page B-175	tLocationScaleDist ribution	pdf cdf icdf mle random	<b>Distribution Fitter</b>

### Discrete Distributions

Distribution	Using Objects	Legacy Functions	Apps/UIs
Binomial on page B-9	BinomialDistributi on	binopdf binocdf binoinv binostat binofit binornd	<b>Probability Distribution Function</b> randtool
Bernoulli on page B-2		mle	

Distribution	Using Objects	Legacy Functions	Apps/UIs
Geometric on page B-64		geopdf geocdf geoinv geostat mle geornd	<b>Probability Distribution Function</b> randtool
Hypergeometric on page B-79		hygepdf hygecdf hygeinv hygestat hygernd	<b>Probability Distribution Function</b> randtool
Multinomial on page B-103	MultinomialDistribution	mnpdf mnrnd	
Negative binomial on page B-120	NegativeBinomialDistribution	nbinspdf nbincdf nbininv nbinstat nbinfit nbinrnd	distributionFitter <b>Probability Distribution Function</b> randtool
Poisson on page B-143	PoissonDistribution	poisspdf poisscdf poissinv poisstat poissfit poissrnd	distributionFitter <b>Probability Distribution Function</b> randtool
Uniform (discrete) on page B-190		unidpdf unidcdf unidinv unidstat mle unidrnd	<b>Probability Distribution Function</b> randtool



## Multivariate Distributions

Distribution	Object	Legacy Functions	Apps/UI
Gaussian copula on page 5-164		copulapdf copulacdf copulastat copulafit copularnd	
Gaussian mixture on page B-52	gmdistribution	pdf cdf fit random	
<i>t</i> copula on page 5-164		copulapdf copulacdf copulastat copulafit copularnd	
Clayton copula on page 5-164		copulapdf copulacdf copulastat copulafit copularnd	
Frank copula on page 5-164		copulapdf copulacdf copulastat copulafit copularnd	
Gumbel copula on page 5-164		copulapdf copulacdf copulastat copulafit copularnd	
Inverse Wishart on page B-83		iwishrnd	
Multivariate normal on page B-106		mvnpdf mvncdf mvnrnd	

Distribution	Object	Legacy Functions	Apps/UI
Multivariate $t$ on page B-112		mvtpdf mvtcdf mvtrnd	
Wishart on page B-198		wishrnd	

### Nonparametric Distributions

Distribution	Using Objects	Legacy Functions	Apps/Uls
Nonparametric on page B-86 (kernel)	KernelDistribution	ksdensity	<b>Distribution Fitter</b>
Pareto	paretotails		

### Flexible Distribution Families

Distribution	Using Objects	Legacy Functions	Apps/Uls
Pearson system on page B-140		pearsrnd	
Johnson system on page B-85		johnsrnd	

## See Also

### More About

- “Working with Probability Distributions” on page 5-3
- “Nonparametric and Empirical Probability Distributions” on page 5-37

## Maximum Likelihood Estimation

The Statistics and Machine Learning Toolbox function `mle` is a convenient front end to the individual distribution fitting functions, and more. The function computes maximum likelihood estimates (MLEs) for distributions beyond those for which Statistics and Machine Learning Toolbox software provides specific pdf functions.

For some pdfs, MLEs can be given in closed form and computed directly. For other pdfs, a search for the maximum likelihood must be employed. The search can be controlled with an `options` input argument, created using the `statset` function. For efficient searches, it is important to choose a reasonable distribution model and set appropriate convergence tolerances.

MLEs can be heavily biased, especially for small samples. As sample size increases, however, MLEs become unbiased minimum variance estimators with approximate normal distributions. This is used to compute confidence bounds for the estimates.

For example, consider the following distribution of means from repeated random samples of an exponential distribution:

```
mu = 1; % Population parameter
n = 1e3; % Sample size
ns = 1e4; % Number of samples

rng default % For reproducibility
samples = exprnd(mu,n,ns); % Population samples
means = mean(samples); % Sample means
```

The Central Limit Theorem says that the means will be approximately normally distributed, regardless of the distribution of the data in the samples. The `normfit` function can be used to find the normal distribution on page B-134 that best fits the means:

```
[muhat,sigmahat,muci,sigmaci] = normfit(means)
```

```
muhat =

    1.0000

sigmahat =
```

```
0.0315

muci =

    0.9994
    1.0006

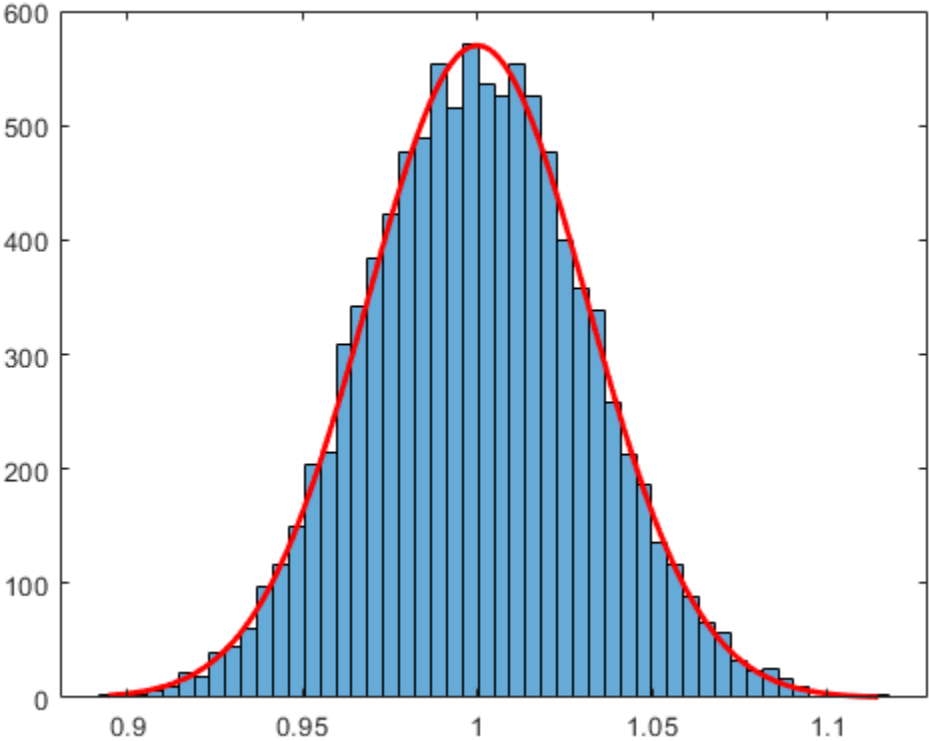
sigmaci =

    0.0311
    0.0319
```

The function returns MLEs for the mean and standard deviation and their 95% confidence intervals.

To visualize the distribution of sample means together with the fitted normal distribution, you must scale the fitted pdf, with area = 1, to the area of the histogram being used to display the means:

```
numbins = 50;
histogram(means,numbins)
hold on
[bincounts,binpositions] = hist(means,numbins);
binwidth = binpositions(2) - binpositions(1);
histarea = binwidth*sum(bincounts);
x = binpositions(1):0.001:binpositions(end);
y = normpdf(x,muhat,sigmahat);
plot(x,histarea*y,'r','LineWidth',2)
```



## Negative Loglikelihood Functions

Negative loglikelihood functions for supported Statistics and Machine Learning Toolbox distributions all end with `like`, as in `explike`. Each function represents a parametric family of distributions. Input arguments are lists of parameter values specifying a particular member of the distribution family followed by an array of data. Functions return the negative log-likelihood of the parameters, given the data.

Negative log-likelihood functions are used as objective functions in search algorithms such as the one implemented by the MATLAB function `fminsearch`. Additional search algorithms are implemented by Optimization Toolbox™ functions and Global Optimization Toolbox functions.

When used to compute maximum likelihood estimates (MLEs), negative log-likelihood functions allow you to choose a search algorithm and exercise low-level control over algorithm execution. By contrast, the functions discussed in “Maximum Likelihood Estimation” on page 5-27 use preset algorithms with options limited to those set by the `statset` function.

Likelihoods are conditional probability densities. A parametric family of distributions is specified by its pdf  $f(x,a)$ , where  $x$  and  $a$  represent the variables and parameters, respectively. When  $a$  is fixed, the pdf is used to compute the density at  $x$ ,  $f(x|a)$ . When  $x$  is fixed, the pdf is used to compute the *likelihood* of the parameters  $a$ ,  $f(a|x)$ . The joint likelihood of the parameters over an independent random sample  $X$  is

$$L(a) = \prod_{x \in X} f(a|x)$$

Given  $X$ , MLEs maximize  $L(a)$  over all possible  $a$ .

In numerical algorithms, the log-likelihood function,  $\log(L(a))$ , is (equivalently) optimized. The logarithm transforms the product of potentially small likelihoods into a sum of logs, which is easier to distinguish from 0 in computation. For convenience, Statistics and Machine Learning Toolbox negative log-likelihood functions return the *negative* of this sum, since the optimization algorithms to which the values are passed typically search for minima rather than maxima.

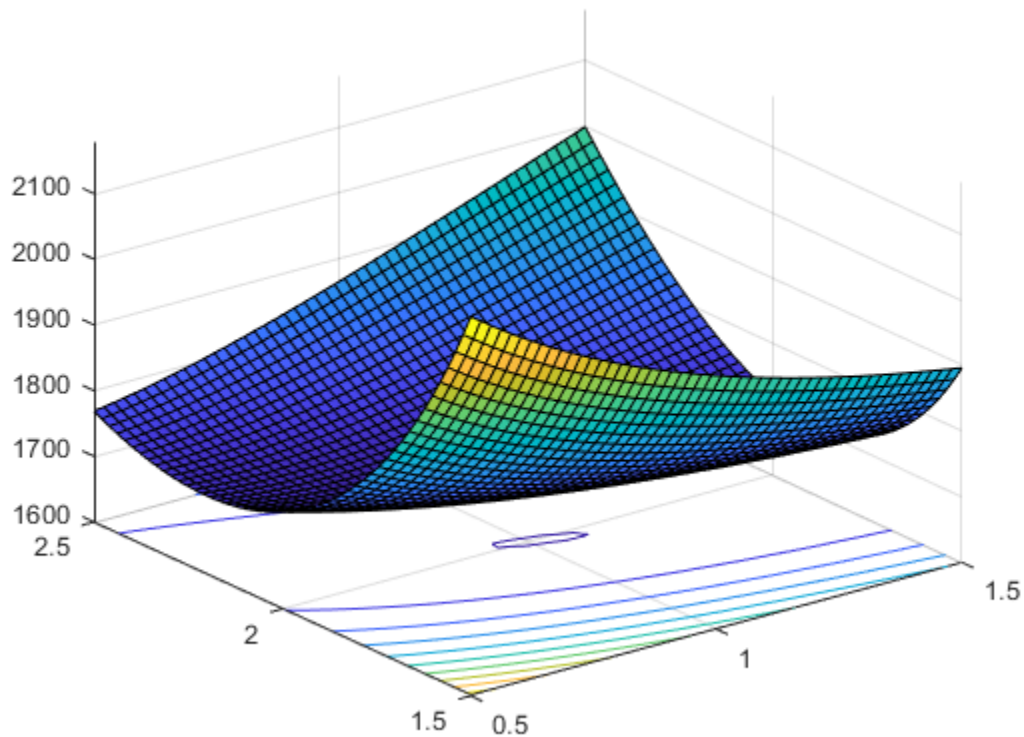
For example, use `gamrnd` to generate a random sample from a specific gamma distribution on page B-47:

```
rng default; % for reproducibility
a = [1,2];
X = gamrnd(a(1),a(2),1e3,1);
```

Given X, the `gamlike` function can be used to visualize the likelihood surface in the neighborhood of a:

```
mesh = 50;
delta = 0.5;
a1 = linspace(a(1)-delta,a(1)+delta,mesh);
a2 = linspace(a(2)-delta,a(2)+delta,mesh);
logL = zeros(mesh); % Preallocate memory
for i = 1:mesh
    for j = 1:mesh
        logL(i,j) = gamlike([a1(i),a2(j)],X);
    end
end

[A1,A2] = meshgrid(a1,a2);
surf(A1,A2,logL)
```



The MATLAB function `fminsearch` is used to search for the minimum of the likelihood surface:

```
LL = @(u) gamlike([u(1),u(2)],X); % Likelihood given X
MLEs = fminsearch(LL,[1,2])
```

```
MLEs =
```

```
0.9980    2.0172
```

These can be compared to the MLEs returned by the `gamfit` function, which uses a combination search and solve algorithm:



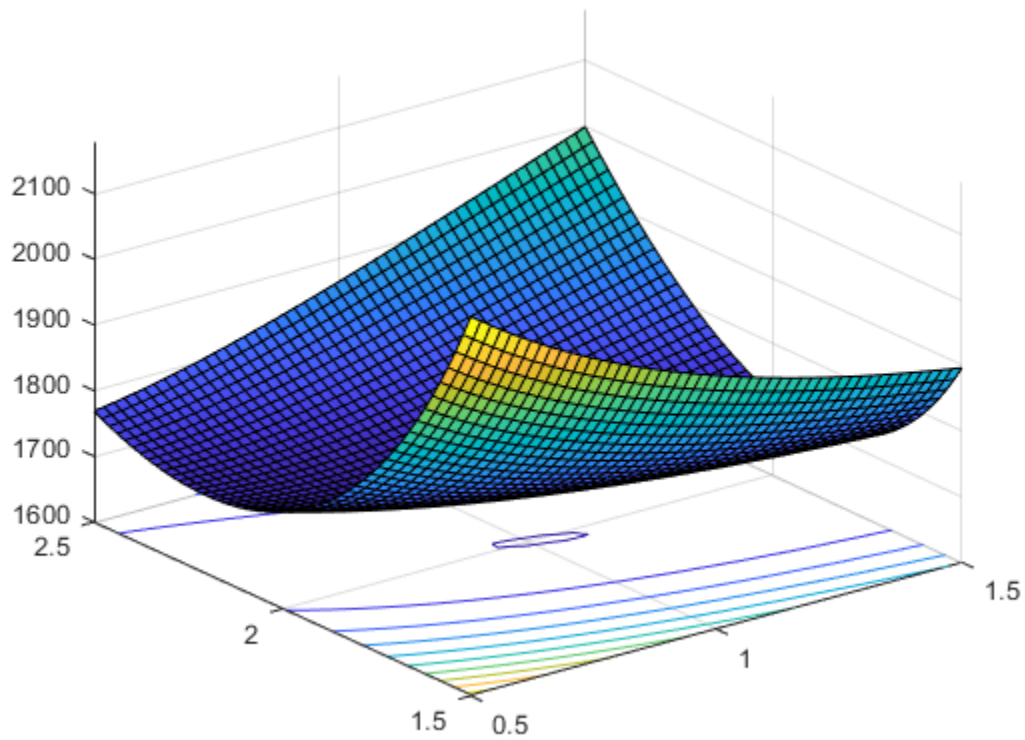
```
ahat = gamfit(X)
```

```
ahat =
```

```
    0.9980    2.0172
```

The MLEs can be added to the surface plot (rotated to show the minimum):

```
hold on  
plot3(MLES(1),MLES(2),LL(MLES),...  
      'ro','MarkerSize',5,...  
      'MarkerFaceColor','r')
```



## Random Number Generation

Statistics and Machine Learning Toolbox supports the generation of random numbers from various distributions. Each RNG represents a parametric family of distributions. RNGs return random numbers from the specified distribution in an array of the specified dimensions.

Other random number generation functions which do not support specific distributions include:

- `cvpartition`
- `datasample`
- `hmmgenerate`
- `lhsdesign`
- `lhsnorm`
- `mhsample`
- `random`
- `randsample`
- `slicesample`

RNGs in Statistics and Machine Learning Toolbox software depend on MATLAB's default random number stream via the `rand` and `randn` functions, each RNG uses one of the techniques discussed in “Common Pseudorandom Number Generation Methods” on page 7-5 to generate random numbers from a given distribution.

By controlling the default random number stream and its state, you can control how the RNGs in Statistics and Machine Learning Toolbox software generate random values. For example, to reproduce the same sequence of values from an RNG, you can save and restore the default stream's state, or reset the default stream. For details on managing the default random number stream, see “Managing the Global Stream” (MATLAB).

MATLAB initializes the default random number stream to the same state each time it starts up. Thus, RNGs in Statistics and Machine Learning Toolbox software will generate the same sequence of values for each MATLAB session unless you modify that state at startup. One simple way to do that is to add commands to `startup.m` such as

```
rng shuffle
```

that initialize MATLAB's default random number stream to a different state for each session.

The following table lists the dependencies of Statistics and Machine Learning Toolbox RNGs on the MATLAB base RNGs `rand`, `randi`, and/or `randn`.

<b>RNG</b>	<b>MATLAB Base RNG</b>
<code>betarnd</code>	<code>rand</code> , <code>randn</code>
<code>binornd</code>	<code>rand</code>
<code>chi2rnd</code>	<code>rand</code> , <code>randn</code>
<code>evrnd</code>	<code>rand</code>
<code>exprnd</code>	<code>rand</code>
<code>datasample</code>	<code>rand</code> , <code>randi</code> , <code>randperm</code>
<code>frnd</code>	<code>rand</code> , <code>randn</code>
<code>gamrnd</code>	<code>rand</code> , <code>randn</code>
<code>geornd</code>	<code>rand</code>
<code>gevrnd</code>	<code>rand</code>
<code>gprnd</code>	<code>rand</code>
<code>hygernd</code>	<code>rand</code>
<code>iwishrnd</code>	<code>rand</code> , <code>randn</code>
<code>johnsrnd</code>	<code>randn</code>
<code>lhsdesign</code>	<code>rand</code>
<code>lhsnorm</code>	<code>rand</code>
<code>lognrnd</code>	<code>randn</code>
<code>mhsample</code>	<code>rand</code> or <code>randn</code> , depending on the RNG given for the proposal distribution
<code>mvnrnd</code>	<code>randn</code>
<code>mvtrnd</code>	<code>rand</code> , <code>randn</code>
<code>nbinrnd</code>	<code>rand</code> , <code>randn</code>
<code>ncfrnd</code>	<code>rand</code> , <code>randn</code>
<code>nctrnd</code>	<code>rand</code> , <code>randn</code>

<b>RNG</b>	<b>MATLAB Base RNG</b>
ncx2rnd	randn
normrnd	randn
pearsrnd	rand or randn, depending on the distribution type
poissrnd	rand, randn
random	rand or randn, depending on the specified distribution
randsample	rand
raylrnd	randn
slicesample	rand
trnd	rand, randn
unidrnd	rand
unifrnd	rand
wblrnd	rand
wishrnd	rand, randn

# Nonparametric and Empirical Probability Distributions

**In this section...**

“Overview” on page 5-37

“Kernel Distribution” on page 5-37

“Empirical Cumulative Distribution Function” on page 5-39

“Piecewise Linear Distribution” on page 5-41

“Pareto Tails” on page 5-42

“Triangular Distribution” on page 5-43

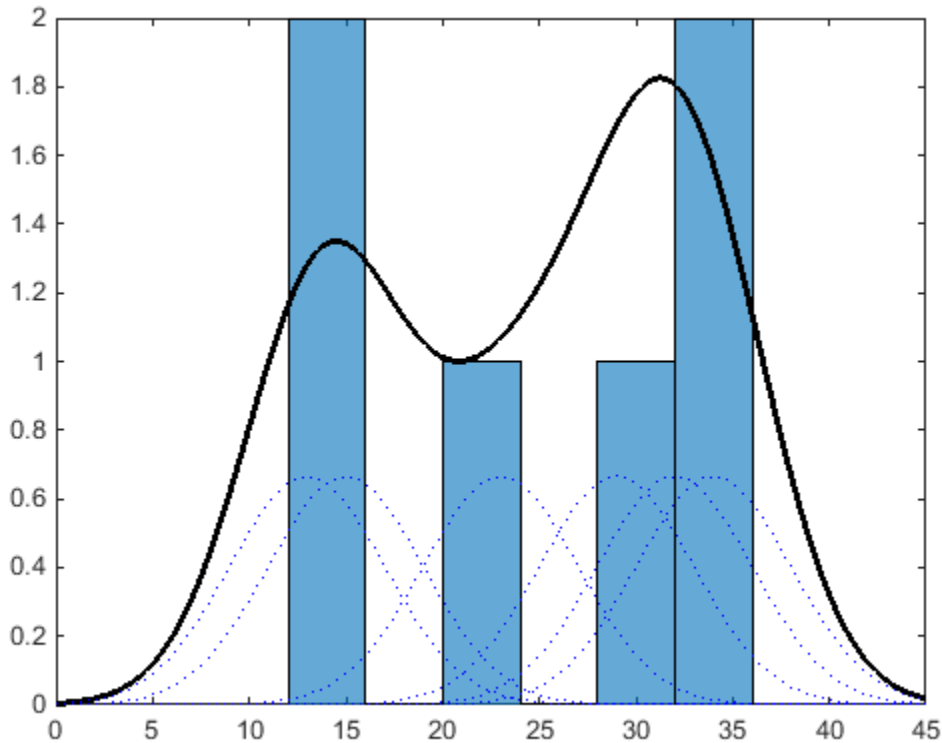
## Overview

In some situations, you cannot accurately describe a data sample using a parametric distribution. Instead, the probability density function (pdf) or cumulative distribution function (cdf) must be estimated from the data. Statistics and Machine Learning Toolbox provides several options for estimating the pdf or cdf from sample data.

## Kernel Distribution

A kernel distribution on page B-86 produces a nonparametric probability density estimate that adapts itself to the data, rather than selecting a density with a particular parametric form and estimating the parameters. This distribution is defined by a kernel density estimator, a smoothing function that determines the shape of the curve used to generate the pdf, and a bandwidth value that controls the smoothness of the resulting density curve.

Similar to a histogram, the kernel distribution builds a function to represent the probability distribution using the sample data. But unlike a histogram, which places the values into discrete bins, a kernel distribution sums the component smoothing functions for each data value to produce a smooth, continuous probability curve. The following plot shows a visual comparison of a histogram and a kernel distribution generated from the same sample data.



A histogram represents the probability distribution by establishing bins and placing each data value in the appropriate bin. Because of this bin count approach, the histogram produces a discrete probability density function. This might be unsuitable for certain applications, such as generating random numbers from a fitted distribution.

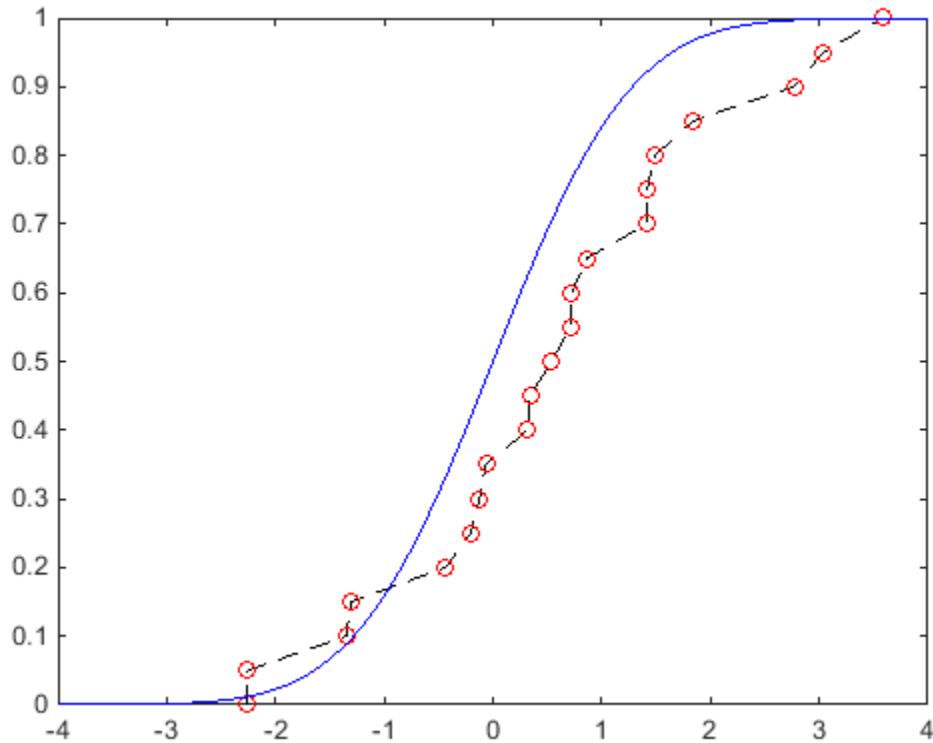
Alternatively, the kernel distribution builds the probability density function (pdf) by creating an individual probability density curve for each data value, then summing the smooth curves. This approach creates one smooth, continuous probability density function for the data set.

For more general information about kernel distributions, see “Kernel Distribution” on page B-86. For information on how to work with a kernel distribution, see `Using KernelDistribution Objects` and `ksdensity`.

## Empirical Cumulative Distribution Function

An empirical cumulative distribution function (ecdf) estimates the cdf of a random variable by assigning equal probability to each observation in a sample. Because of this approach, the ecdf is a discrete cumulative distribution function that creates an exact match between the ecdf and the distribution of the sample data.

The following plot shows a visual comparison of the ecdf of 20 random numbers generated from a standard normal distribution, and the theoretical cdf of a standard normal distribution. The circles indicate the value of the ecdf calculated at each sample data point. The dashed line that passes through each circle visually represents the ecdf, although the ecdf is not a continuous function. The solid line shows the theoretical cdf of the standard normal distribution from which the random numbers in the sample data were drawn.



The ecdf is similar in shape to the theoretical cdf, although it is not an exact match. Instead, the ecdf is an exact match to the sample data. The ecdf is a discrete function, and is not smooth, especially in the tails where data might be sparse. You can smooth the distribution with Pareto tails on page 5-42, using the `paretotails` function.

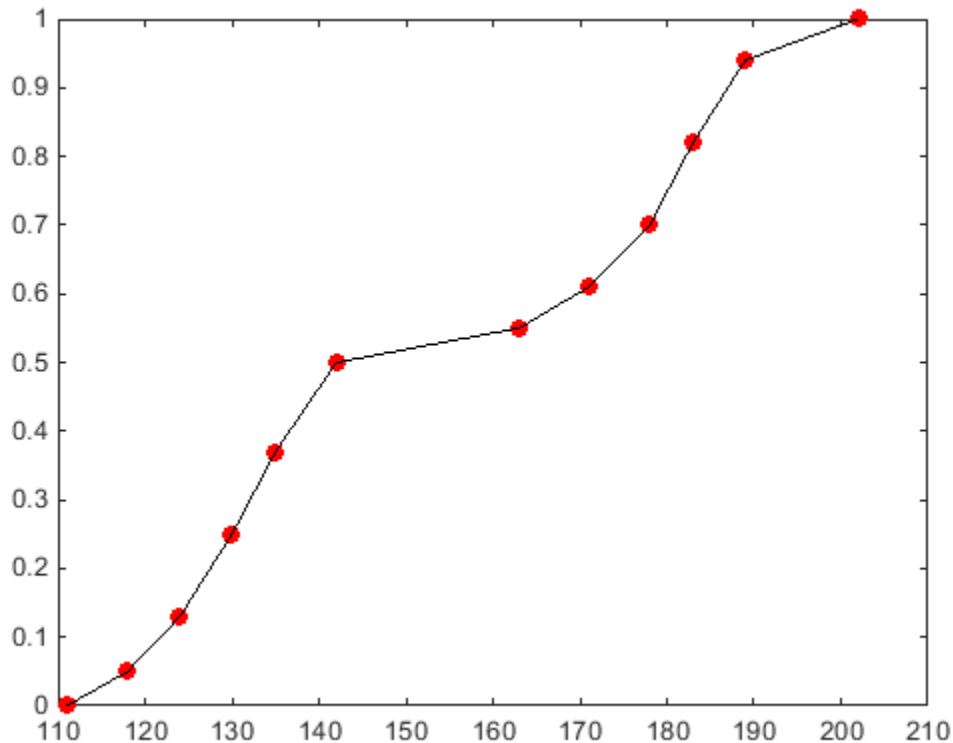
For more information and additional syntax options, see `ecdf`. To construct a continuous function based on cdf values computed from sample data, see “Piecewise Linear Distribution” on page 5-41.



## Piecewise Linear Distribution

A piecewise linear distribution on page B-141 estimates an overall cdf for the sample data by computing the cdf value at each individual point, and then linearly connecting these values to form a continuous curve.

The following plot shows the cdf for a piecewise linear distribution based on a sample of hospital patients' weight measurements. The circles represent each individual data point (weight measurement). The black line that passes through each data point represents the piecewise linear distribution cdf for the sample data.



A piecewise linear distribution linearly connects the cdf values calculated at each sample data point to form a continuous curve. By contrast, an empirical cumulative distribution function on page 5-39 constructed using the `ecdf` function produces a discrete cdf. For

example, random numbers generated from the ecdf can only include  $x$  values contained in the original sample data. Random numbers generated from a piecewise linear distribution can include any  $x$  value between the lower and upper boundaries of the sample data.

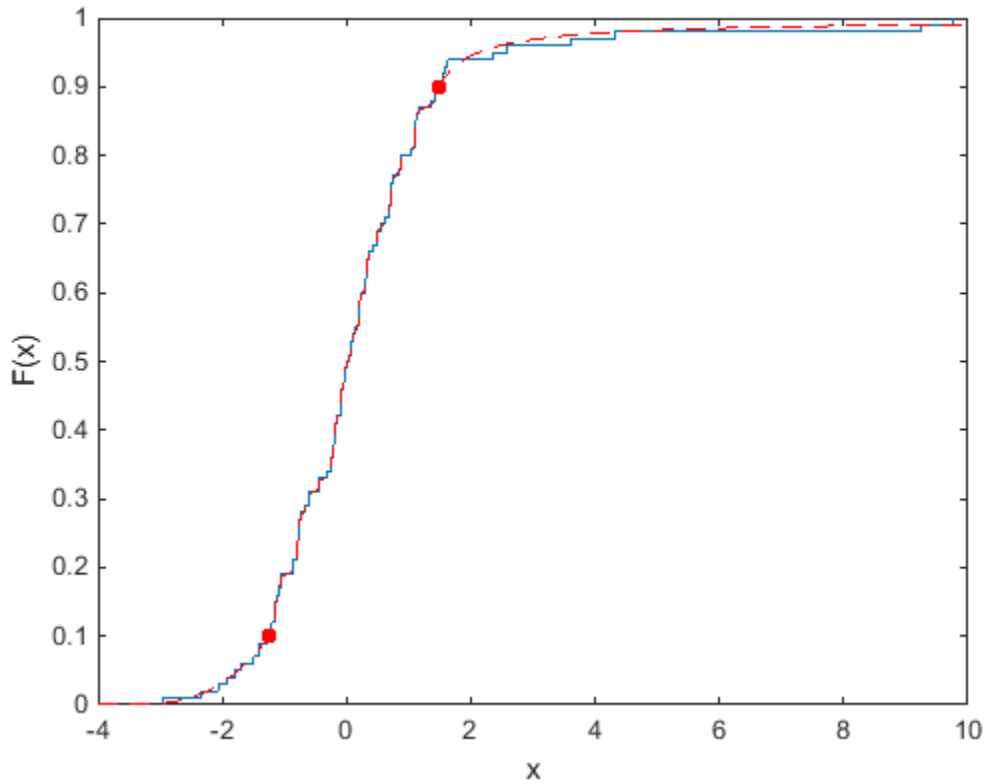
Because the piecewise linear distribution cdf is constructed from the values contained in the sample data, the resulting curve is often not smooth, especially in the tails where data might be sparse. You can smooth the distribution with Pareto tails on page 5-42, using the `paretotails` function.

For information on how to work with a piecewise linear distribution, see [Using PiecewiseLinearDistribution Objects](#).

### Pareto Tails

Pareto tails use a piecewise approach to improve the fit of a nonparametric cdf or pdf by smoothing the tails of the distribution. You can fit a kernel distribution on page 5-37, empirical cdf on page 5-39, or piecewise linear distribution on page 5-41 to the middle data values, then fit generalized Pareto distribution on page B-58 curves to the tails. This technique is especially useful when the sample data is sparse in the tails.

The following plot shows the empirical cdf (ecdf) of a data sample containing 20 random numbers. The solid line represents the ecdf, and the dashed line represents the empirical cdf with Pareto tails fit to the lower and upper 10 percent of the data. The circles denote the boundaries for the lower and upper 10 percent of the data.

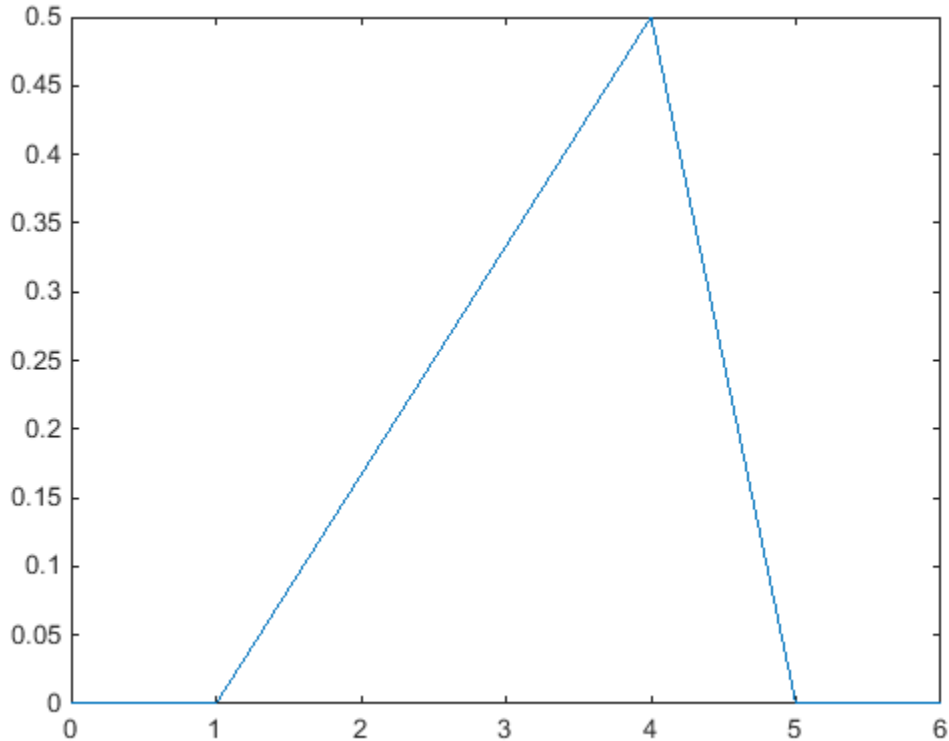


Fitting Pareto tails to the lower and upper 10 percent of the sample data makes the cdf smoother in the tails, where the data is sparse. For more information on working with Pareto tails, see `paretotails`.

## Triangular Distribution

A “Triangular Distribution” on page B-178 provides a simplistic representation of the probability distribution when limited sample data is available. This continuous distribution is parameterized by a lower limit, peak location, and upper limit. These points are linearly connected to estimate the pdf of the sample data. You can use the mean, median, or mode of the data as the peak location.

The following plot shows the triangular distribution pdf of a random sample of 10 integers from 0 to 5. The lower limit is the smallest integer in the sample data, and the upper limit is the largest integer. The peak for this plot is at the mode, or most frequently-occurring value, in the sample data.



Business applications such as simulation and project management sometimes use a triangular distribution to create models when limited sample data exists. For more information, see “Triangular Distribution” on page B-178.

### See Also

`ecdf` | `ksdensity` | `paretotails`

## Related Examples

- “Fit a Nonparametric Distribution with Pareto Tails” on page 5-57

## More About

- “Kernel Distribution” on page B-86
- “Piecewise Linear Distribution” on page B-141
- “Triangular Distribution” on page B-178

## Fit Kernel Distribution Object to Data

This example shows how to fit a kernel probability distribution object to sample data.

### Step 1. Load sample data.

Load the sample data.

```
load carsmall;
```

This data contains miles per gallon (MPG) measurements for different makes and models of cars, grouped by country of origin (Origin), model year (Year), and other vehicle characteristics.

### Step 2. Fit a kernel distribution object.

Use `fitdist` to fit a kernel probability distribution object to the miles per gallon (MPG) data for all makes of cars.

```
pd = fitdist(MPG, 'Kernel')
```

```
pd =
```

```
KernelDistribution

Kernel = normal
Bandwidth = 4.11428
Support = unbounded
```

This creates a `prob.KernelDistribution` object. By default, `fitdist` uses a normal kernel smoothing function and chooses an optimal bandwidth for estimating normal densities, unless you specify otherwise. You can access information about the fit and perform further calculations using the related object functions.

### Step 3. Compute descriptive statistics.

Compute the mean, median, and standard deviation of the fitted kernel distribution.

```
m = mean(pd)
```

```
m =
```

```
23.7181

med = median(pd)

med =

23.4841

s = std(pd)

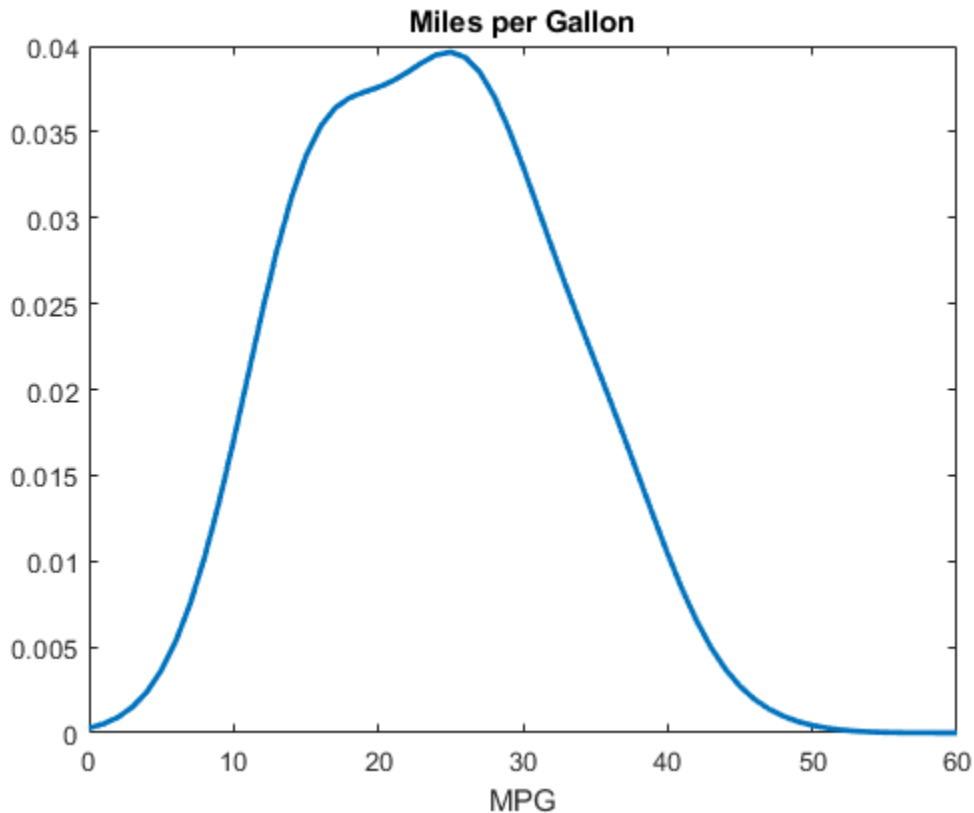
s =

8.9896
```

**Step 4. Compute and plot the pdf.**

Compute and plot the pdf of the fitted kernel distribution.

```
figure
x = 0:1:60;
y = pdf(pd,x);
plot(x,y,'LineWidth',2)
title('Miles per Gallon')
xlabel('MPG')
```



The plot shows the pdf of the kernel distribution fit to the MPG data across all makes of cars. The distribution is smooth and fairly symmetrical, although it is slightly skewed with a heavier right tail.

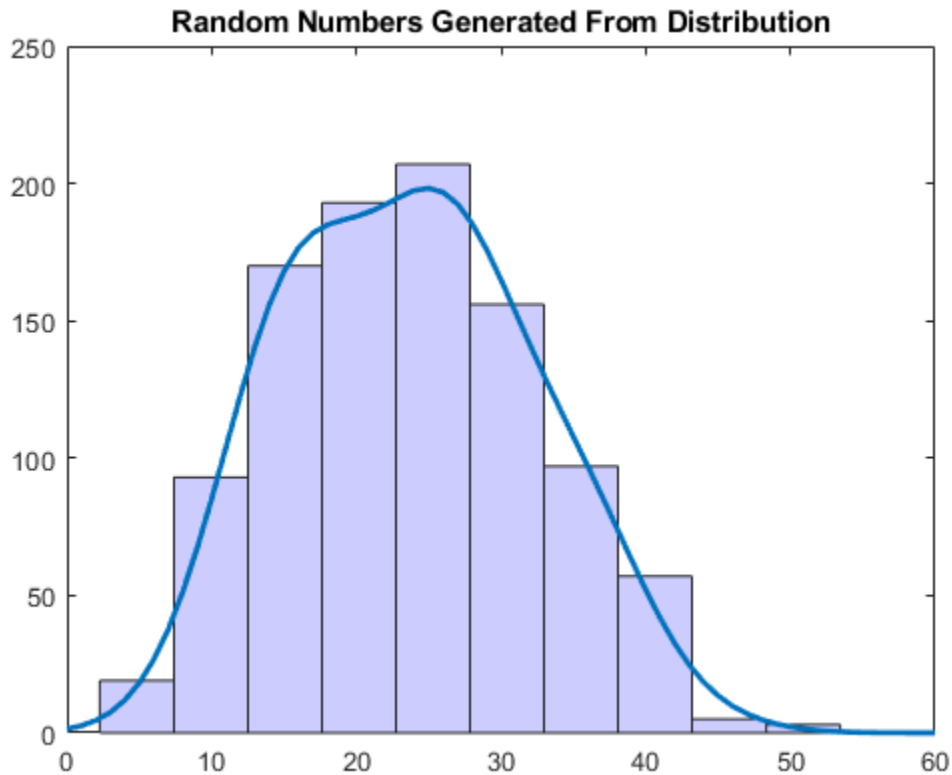
#### Step 5. Generate random numbers.

Generate a vector of random numbers from the fitted kernel distribution.

```
rng('default') % For reproducibility
r = random(pd,1000,1);
figure
hist(r);
set(get(gca,'Children'),'FaceColor',[.8 .8 1]);
hold on
```



```
y = y*5000; % Scale pdf to overlay on histogram
plot(x,y,'LineWidth',2)
title('Random Numbers Generated From Distribution')
hold off
```



The histogram has a similar shape to the pdf plot because the random numbers generate from the nonparametric kernel distribution fit to the sample data.

## See Also

`fitdist`

## **Related Examples**

- “Fit Kernel Distribution Using ksdensity” on page 5-51

## **More About**

- “Kernel Distribution” on page B-86

## Fit Kernel Distribution Using `ksdensity`

This example shows how to generate a kernel probability density estimate from sample data using the `ksdensity` function.

### Step 1. Load sample data.

Load the sample data.

```
load carsmall;
```

This data contains miles per gallon (MPG) measurements for different makes and models of cars, grouped by country of origin (`Origin`), model year (`Year`), and other vehicle characteristics.

### Step 2. Generate a kernel probability density estimate.

Use `ksdensity` to generate a kernel probability density estimate for the miles per gallon (MPG) data.

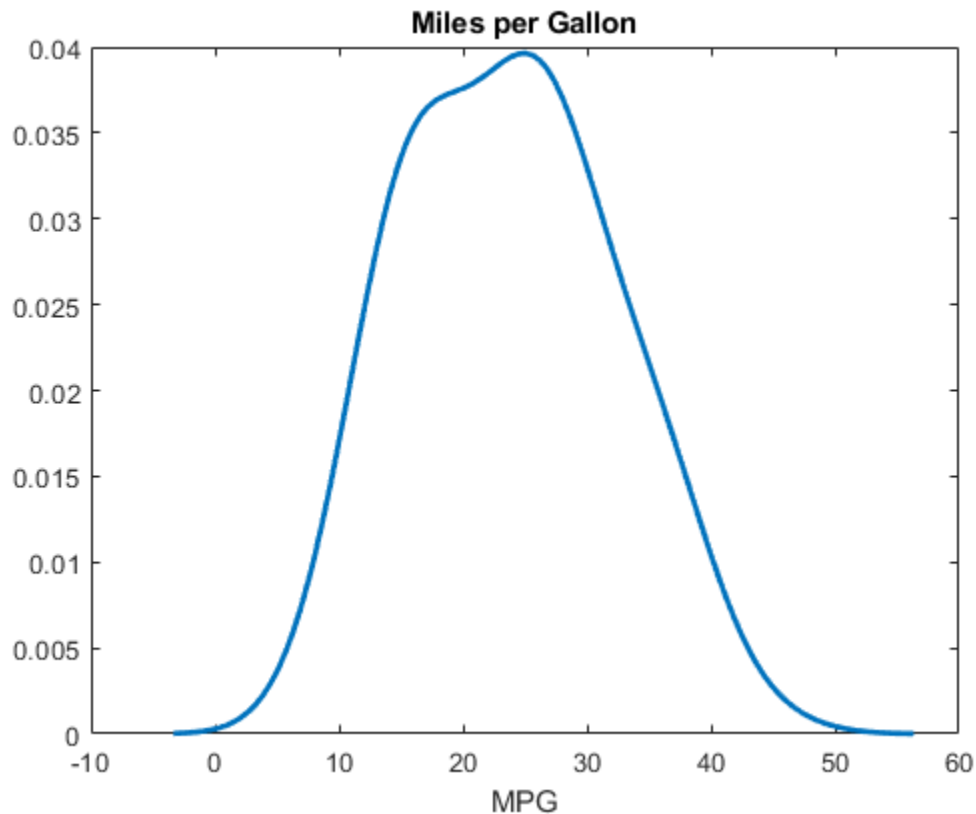
```
[f,xi] = ksdensity(MPG);
```

By default, `ksdensity` uses a normal kernel smoothing function and chooses an optimal bandwidth for estimating normal densities, unless you specify otherwise.

### Step 3. Plot the kernel probability density estimate.

Plot the kernel probability density estimate to visualize the MPG distribution.

```
plot(xi,f,'LineWidth',2)  
title('Miles per Gallon')  
xlabel('MPG')
```



The plot shows the pdf of the kernel distribution fit to the MPG data across all makes of cars. The distribution is smooth and fairly symmetrical, although it is slightly skewed with a heavier right tail.

### See Also

`ksdensity`

### Related Examples

- “Fit Kernel Distribution Object to Data” on page 5-46

## More About

- “Kernel Distribution” on page B-86

## Fit Distributions to Grouped Data Using `ksdensity`

This example shows how to fit kernel distributions to grouped sample data using the `ksdensity` function.

### Step 1. Load sample data.

Load the sample data.

```
load carsmall;
```

The data contains miles per gallon (MPG) measurements for different makes and models of cars, grouped by country of origin (`Origin`), model year (`Model_Year`), and other vehicle characteristics.

### Step 2. Group sample data by origin.

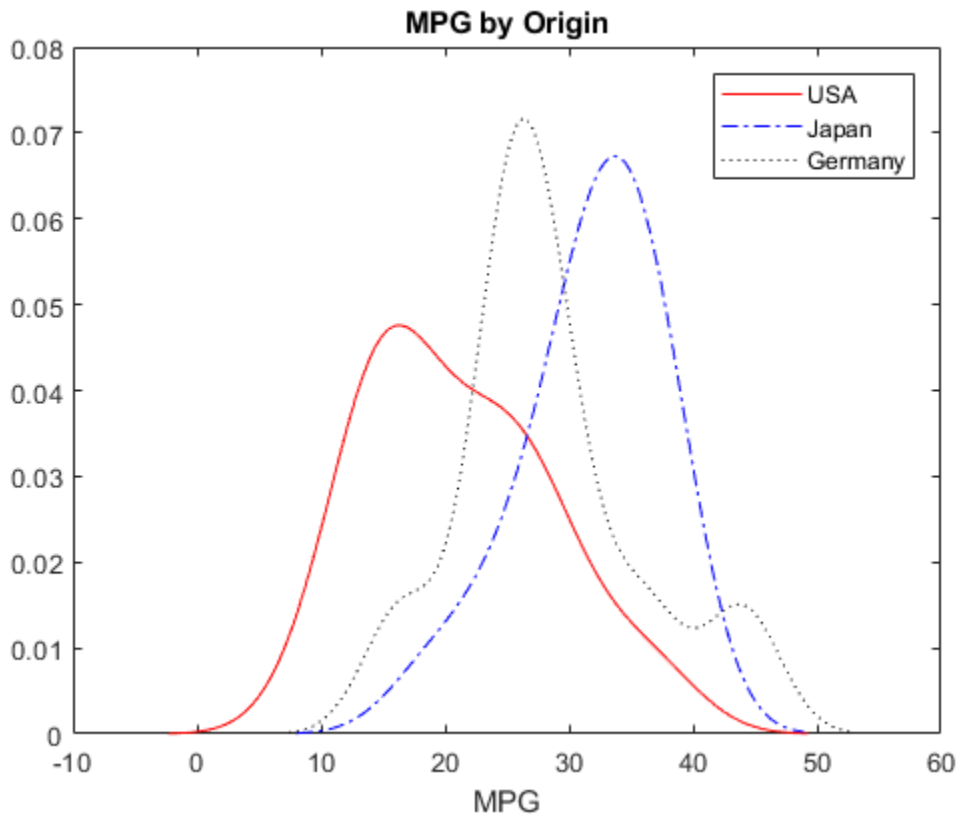
Group the MPG data by origin (`Origin`) for cars made in the USA, Japan, and Germany.

```
Origin = nominal(Origin);  
MPG_USA = MPG(Origin=='USA');  
MPG_Japan = MPG(Origin=='Japan');  
MPG_Germany = MPG(Origin=='Germany');
```

### Step 3. Compute and plot the pdf.

Compute and plot the pdf for each group.

```
[fi,xi] = ksdensity(MPG_USA);  
plot(xi,fi,'r-')  
hold on  
  
[fj,xj] = ksdensity(MPG_Japan);  
plot(xj,fj,'b-.')  
  
[fk,xk] = ksdensity(MPG_Germany);  
plot(xk,fk,'k:')  
  
legend('USA','Japan','Germany')  
title('MPG by Origin')  
xlabel('MPG')  
hold off
```



The plot shows how miles per gallon (MPG) performance differs by country of origin (Origin). Using this data, the USA has the widest distribution, and its peak is at the lowest MPG value of the three origins. Japan has the most regular distribution with a slightly heavier left tail, and its peak is at the highest MPG value of the three origins. The peak for Germany is between the USA and Japan, and the second bump near 44 miles per gallon suggests that there might be multiple modes in the data.

## See Also

makedist

### Related Examples

- “Fit Kernel Distribution Using `ksdensity`” on page 5-51
- “Fit Probability Distribution Objects to Grouped Data” on page 5-121

### More About

- “Kernel Distribution” on page B-86
- “Grouping Variables” on page 2-59



## Fit a Nonparametric Distribution with Pareto Tails

This example shows how to fit a nonparametric probability distribution to sample data using Pareto tails to smooth the distribution in the tails.

### Step 1. Generate sample data.

Generate sample data that contains more outliers than expected from a standard normal distribution.

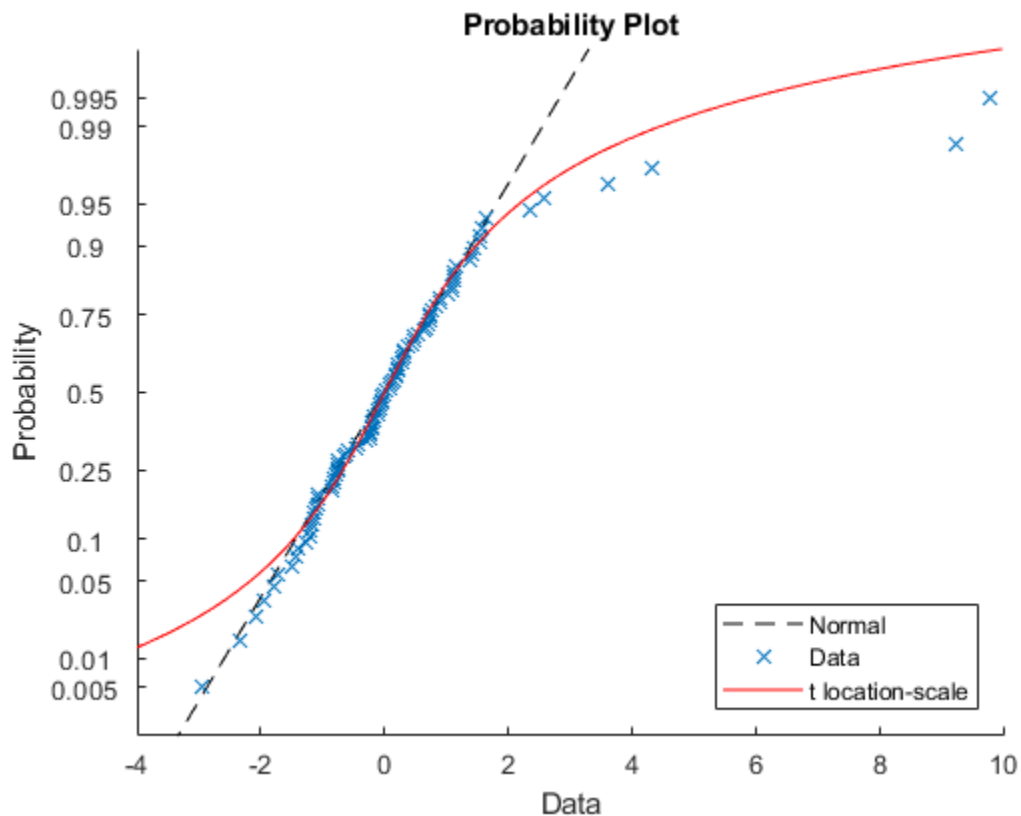
```
rng('default') % For reproducibility
left_tail = -exprnd(1,10,1);
right_tail = exprnd(5,10,1);
center = randn(80,1);
data = [left_tail;center;right_tail];
```

The data contains 80% values from a standard normal distribution, 10% from an exponential distribution with a mean of 5, and 10% from an exponential distribution with mean of -1. The data contains random numbers from an exponential distribution. Compared to a standard normal distribution, the exponential values are more likely to be outliers, especially in the upper tail.

### Step 2. Fit probability distributions to the data.

Fit a normal distribution and a  $t$  location-scale distribution to the data, and plot for a visual comparison.

```
probplot(data);
p = fitdist(data,'tlocation-scale');
h = probplot(gca,p);
set(h,'color','r','linestyle','-');
title('Probability Plot')
legend('Normal','Data','t location-scale','Location','SE')
```

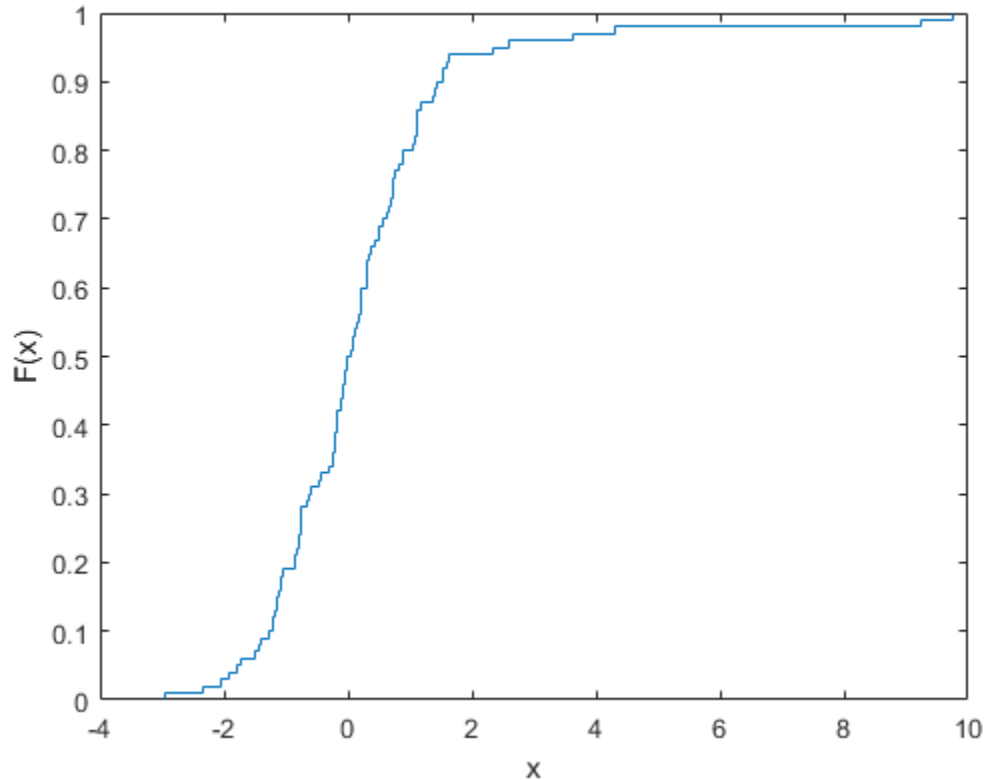


Both distributions appear to fit reasonably well in the center, but neither the normal distribution nor the  $t$  location-scale distribution fit the tails very well.

### Step 3. Generate an empirical distribution.

To obtain a better fit, use `ecdf` to generate an empirical cdf based on the sample data.

```
figure  
ecdf(data)
```



The empirical distribution provides a perfect fit, but the outliers make the tails very discrete. Random samples generated from this distribution using the inversion method might include, for example, values near 4.33 and 9.25, but no values in between.

#### Step 4. Fit a distribution using Pareto tails.

Use `paretotails` to generate an empirical cdf for the middle 80% of the data and fit generalized Pareto distributions to the lower and upper 10%.

```
pfit = paretotails(data,0.1,0.9)
```

```
pfit =
```

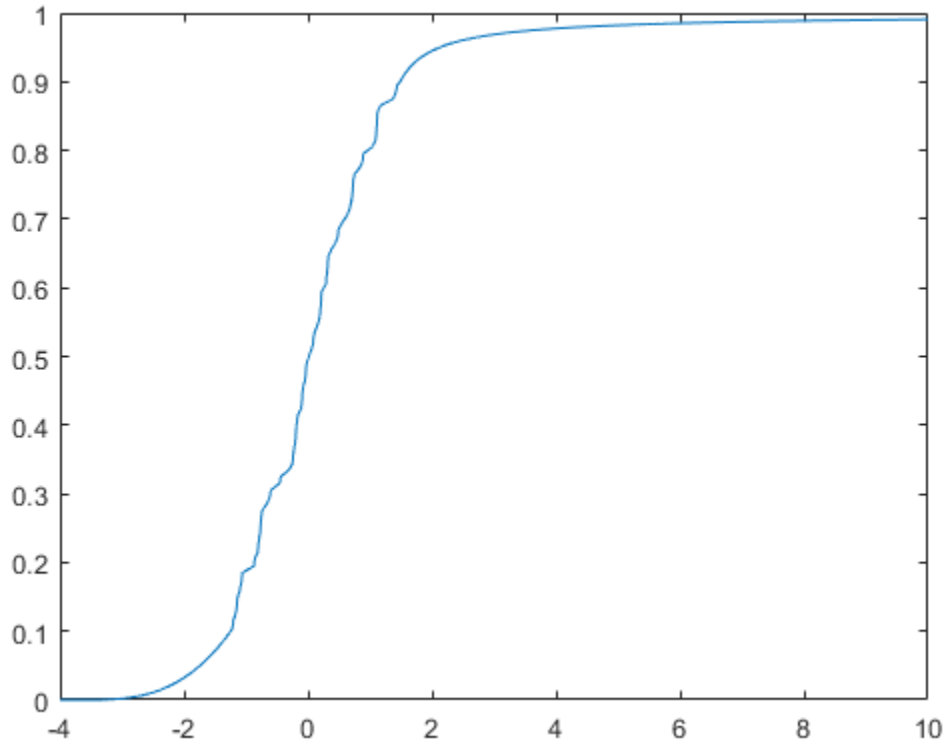
```
Piecewise distribution with 3 segments
  -Inf < x < -1.24623    (0 < p < 0.1): lower tail, GPD(-0.334156,0.798745)
 -1.24623 < x < 1.48551 (0.1 < p < 0.9): interpolated empirical cdf
  1.48551 < x < Inf     (0.9 < p < 1): upper tail, GPD(1.23681,0.581868)
```

To obtain a better fit, `paretotails` fits a distribution by piecing together an `ecdf` or kernel distribution in the center of the sample, and smooth generalized Pareto distributions (GPDs) in the tails. The `paretotails` function creates a `paretotails` probability distribution object. You can access information about the fit and perform further calculations on the object using the methods of the `paretotails` class. For example, you can evaluate the `cdf` or generate random numbers from the distribution.

### Step 5. Compute and plot the `cdf`.

Compute and plot the `cdf` of the fitted `paretotails` distribution.

```
x = -4:0.01:10;
plot(x, cdf(pfit, x))
```



The `paretotails` cdf closely fits the data but is smoother in the tails than the ecdf generated in Step 3.

## Generate Random Numbers Using the Triangular Distribution

This example shows how to create a triangular probability distribution object based on sample data, and generate random numbers for use in a simulation.

### Step 1. Input sample data.

Input the data vector `time`, which contains the observed length of time (in seconds) that 10 different cars stopped at a highway tollbooth.

```
time = [6 14 8 7 16 8 23 6 7 15];
```

The data shows that, while most cars stopped for 6 to 16 seconds, one outlier stopped for 23 seconds.

### Step 2. Estimate distribution parameters.

Estimate the triangular distribution parameters from the sample data.

```
lower = min(time);  
peak = median(time);  
upper = max(time);
```

A triangular distribution provides a simplistic representation of the probability distribution when sample data is limited. Estimate the lower and upper boundaries of the distribution by finding the minimum and maximum values of the sample data. For the peak parameter, the median might provide a better estimate of the mode than the mean, since the data includes an outlier.

### Step 3. Create a probability distribution object.

Create a triangular probability distribution object using the estimated parameter values.

```
pd = makedist('Triangular','a',lower,'b',peak,'c',upper)
```

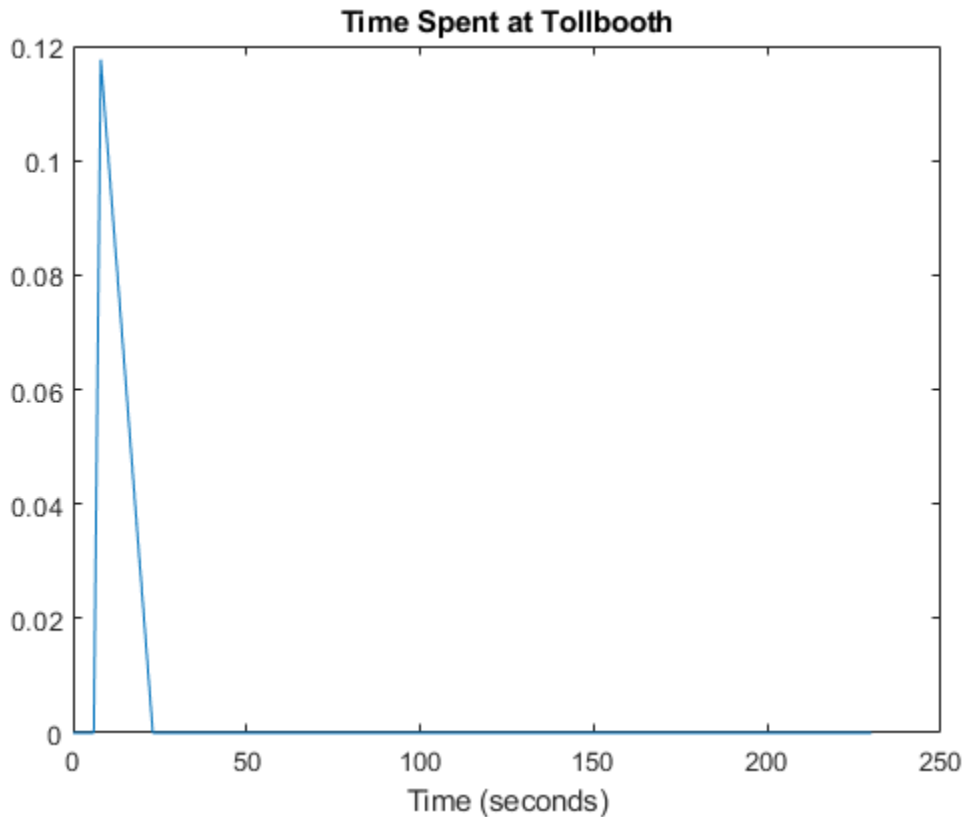
```
pd =
```

```
    TriangularDistribution
```

```
A = 6, B = 8, C = 23
```

Compute and plot the pdf of the triangular distribution.

```
x = 0:.1:230;  
y = pdf(pd,x);  
plot(x,y)  
title('Time Spent at Tollbooth')  
xlabel('Time (seconds)')
```



The plot shows that this triangular distribution is skewed to the right. However, since the estimated peak value is the sample median, the distribution should be symmetrical about the peak. Because of its skew, this model might, for example, generate random numbers that seem unusually high when compared to the initial sample data.

### Step 4. Generate random numbers.

Generate random numbers from this distribution to simulate future traffic flow through the tollbooth.

```
rng('default'); % For reproducibility
r = random(pd,10,1)
```

```
r =
    16.1265
    18.0987
     8.0796
    18.3001
    13.3176
     7.8211
     9.4360
    12.2508
    19.7082
    20.0078
```

The returned values in `r` are the time in seconds that the next 10 simulated cars spend at the tollbooth. These values seem high compared to the values in the original data vector `time` because the outlier skewed the distribution to the right. Using the second-highest value as the upper limit parameter might mitigate the effects of the outlier and generate a set of random numbers more similar to the initial sample data.

### Step 5. Revise estimated parameters.

Estimate the upper boundary of the distribution using the second largest value in the sample data.

```
sort_time = sort(time,'descend');
secondLargest = sort_time(2);
```

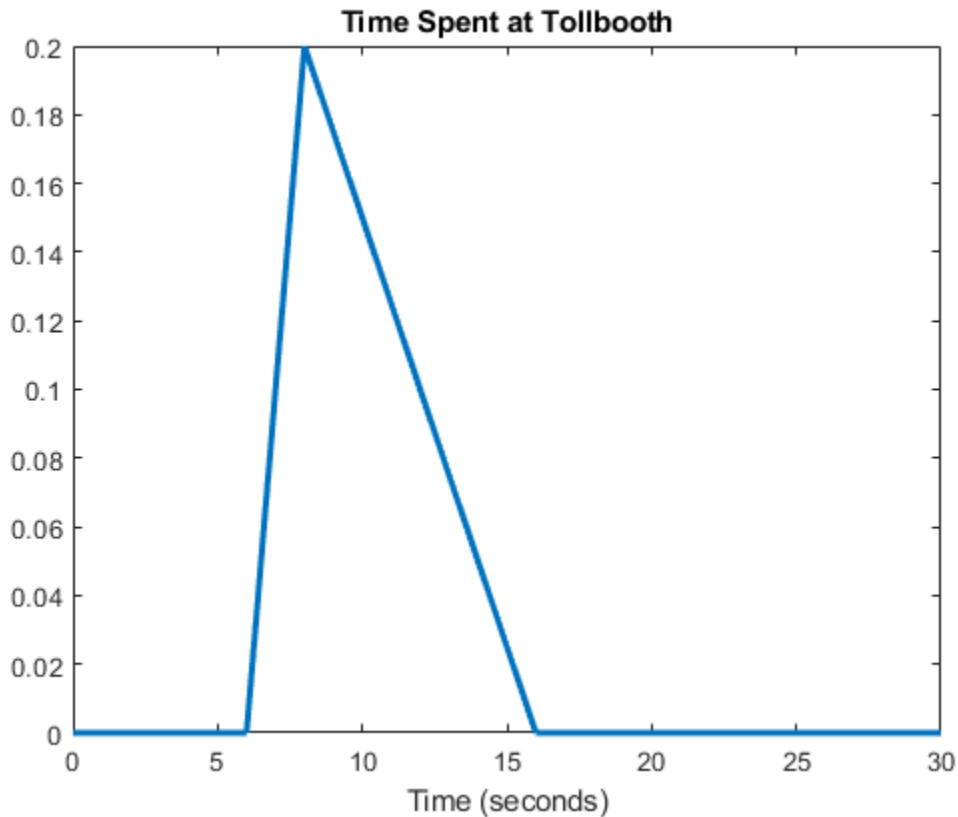
### Step 6. Create a new distribution object and plot the pdf.

Create a new triangular probability distribution object using the revised estimated parameters, and plot its pdf.

```
figure
pd2 = makedist('Triangular','a',lower,'b',peak,'c',secondLargest);
```



```
y2 = pdf(pd2,x);  
plot(x,y2,'LineWidth',2)  
title('Time Spent at Tollbooth')  
xlabel('Time (seconds)')  
xlim([0 30])
```



The plot shows that this triangular distribution is still slightly skewed to the right. However, it is much more symmetrical about the peak than the distribution that used the maximum sample data value to estimate the upper limit.

#### **Step 7. Generate new random numbers.**

Generate new random numbers from the revised distribution.

```
rng('default'); % For reproducibility
r2 = random(pd2,10,1)
```

```
r2 =
    12.1501
    13.2547
     7.5937
    13.3675
    10.5768
     7.3967
     8.4026
     9.9792
    14.1562
    14.3240
```

These new values more closely resemble those in the original data vector `time`. They are also closer to the sample median than the random numbers generated by the distribution that used the outlier to estimate its upper limit. This example does not remove the outlier from the sample data when computing the median. Other options for parameter estimation include removing outliers from the sample data altogether, or using the mean or mode of the sample data as the peak value.

### See Also

`makedist` | `pdf` | `random`

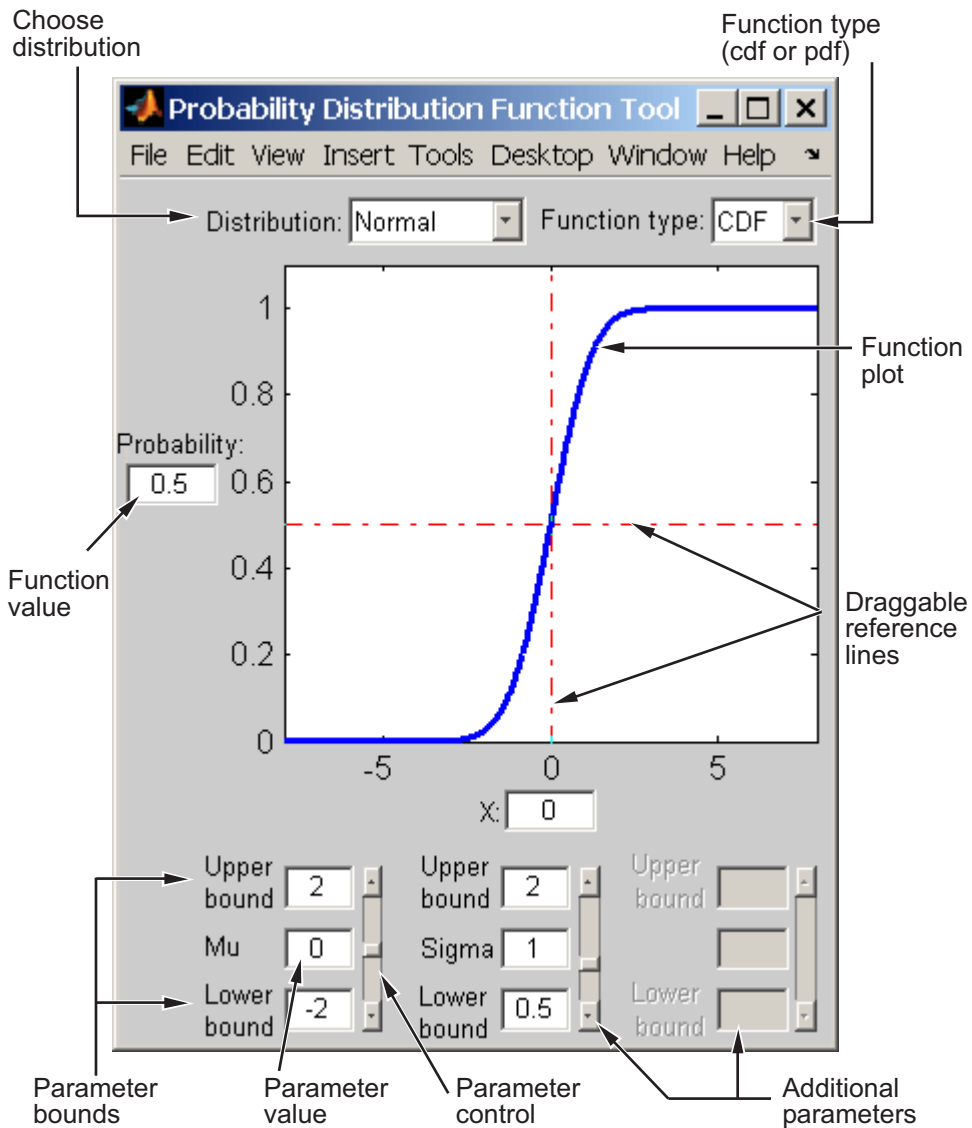
### More About

- “Triangular Distribution” on page B-178

## Explore the Probability Distribution Function UI

The Probability Distribution Function user interface (UI) interactively displays the influence of parameter changes on the shapes of the pdfs and cdfs of supported Statistics and Machine Learning Toolbox distributions.

Run the user interface by typing `disttool` at the command line.



Start by selecting a distribution. Then choose the function type: probability density function (pdf) or cumulative distribution function (cdf).

After the plot appears, you can

- Calculate a new function value by
  - Typing a new  $x$  value in the text box on the  $x$ -axis
  - Dragging the vertical reference line.
  - Clicking in the figure where you want the line to be.

The new function value appears in the text box to the left of the plot.

- For cdf plots, find critical values corresponding to a specific probability by typing the desired probability in the text box on the  $y$ -axis or by dragging the horizontal reference line.
- Use the controls at the bottom of the window to set parameter values for the distribution and to change their upper and lower bounds.

## Model Data Using the Distribution Fitter App

The Distribution Fitter app provides a visual, interactive approach to fitting univariate distributions to data.

### In this section...

“Explore Probability Distributions Interactively” on page 5-70

“Create and Manage Data Sets” on page 5-71

“Create a New Fit” on page 5-76

“Display Results” on page 5-81

“Manage Fits” on page 5-83

“Evaluate Fits” on page 5-85

“Exclude Data” on page 5-89

“Save and Load Sessions” on page 5-94

“Generate a File to Fit and Plot Distributions” on page 5-94

### Explore Probability Distributions Interactively

You can use the Distribution Fitter app to interactively fit probability distributions to data imported from the MATLAB workspace. You can choose from 22 built-in probability distributions, or create your own custom distribution. The app displays the fitted distribution over plots of the empirical distributions, including pdf, cdf, probability plots, and survivor functions. You can export the fit data, including fitted parameter values, to the workspace for further analysis.

#### Distribution Fitter App Workflow

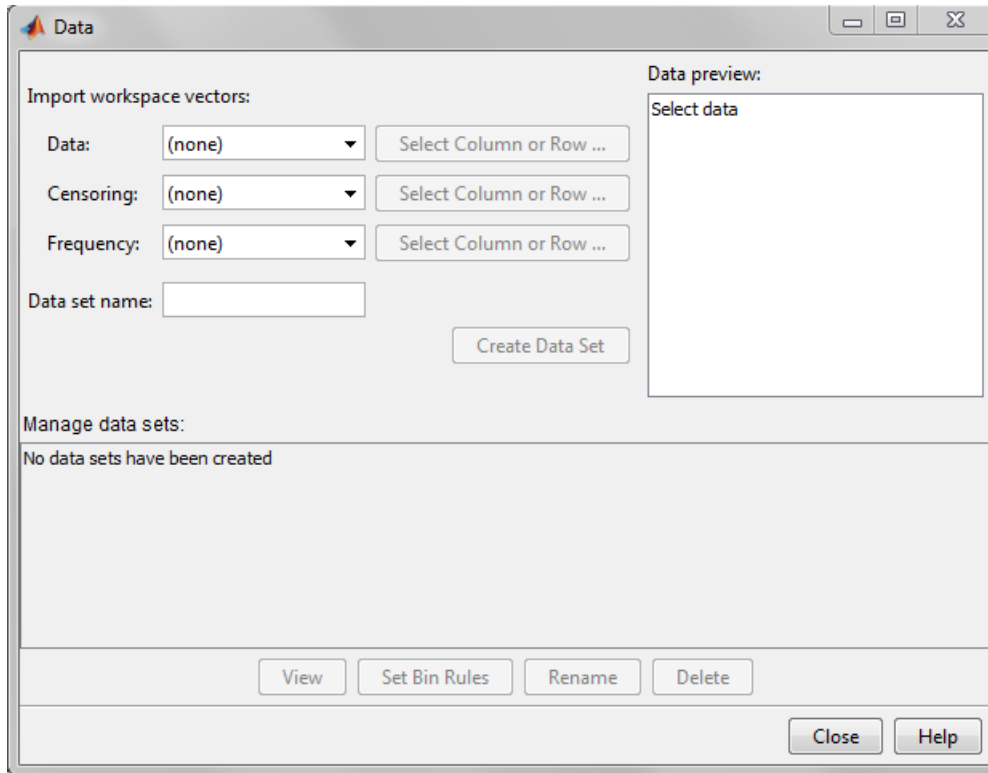
To fit a probability distribution to your sample data:

- 1 On the MATLAB Toolstrip, click the Apps tab. In the Math, Statistics and Optimization group, open the Distribution Fitter app. Alternatively, at the command prompt, enter `distributionFitter`.
- 2 Import your sample data, or create a data vector directly in the app. You can also manage your data sets and choose which one to fit. See “Create and Manage Data Sets” on page 5-71.

- 3 Create a new fit for your data. See “Create a New Fit” on page 5-76.
- 4 Display the results of the fit. You can choose to display the density (pdf), cumulative probability (cdf), quantile (inverse cdf), probability plot (choose one of several distributions), survivor function, and cumulative hazard. See “Display Results” on page 5-81.
- 5 You can create additional fits, and manage multiple fits from within the app. See “Manage Fits” on page 5-83.
- 6 Evaluate probability functions for the fit. You can choose to evaluate the density (pdf), cumulative probability (cdf), quantile (inverse cdf), survivor function, and cumulative hazard. See “Evaluate Fits” on page 5-85.
- 7 Improve the fit by excluding certain data. You can specify bounds for the data to exclude, or you can exclude data graphically using a plot of the values in the sample data. See “Exclude Data” on page 5-89.
- 8 Save your current Distribution Fitter app session so you can open it later. See “Save and Load Sessions” on page 5-94.

## Create and Manage Data Sets

To open the Data dialog box, click the **Data** button in the Distribution Fitter app.



## Import Data

Create a data set by importing a vector from the MATLAB workspace using the **Import workspace vectors** pane.

- **Data** — In the **Data** field, the drop-down list contains the names of all matrices and vectors, other than 1-by-1 matrices (scalars) in the MATLAB workspace. Select the array containing the data that you want to fit. The actual data you import must be a vector. If you select a matrix in the **Data** field, the first column of the matrix is imported by default. To select a different column or row of the matrix, click **Select Column or Row**. The matrix displays in the Variables editor. You can select a row or column by highlighting it.

Alternatively, you can enter any valid MATLAB expression in the **Data** field.



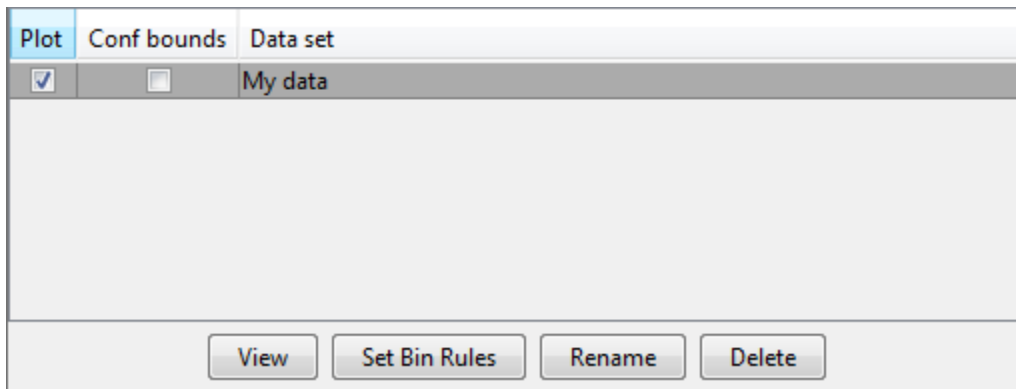
When you select a vector in the **Data** field, a histogram of the data appears in the **Data preview** pane.

- **Censoring** — If some of the points in the data set are censored, enter a Boolean vector of the same size as the data vector, specifying the censored entries of the data. A 1 in the censoring vector specifies that the corresponding entry of the data vector is censored. A 0 specifies that the entry is not censored. If you enter a matrix, you can select a column or row by clicking **Select Column or Row**. If you do not have censored data, leave the **Censoring** field blank.
- **Frequency** — Enter a vector of positive integers of the same size as the data vector to specify the frequency of the corresponding entries of the data vector. For example, a value of 7 in the 15th entry of frequency vector specifies that there are 7 data points corresponding to the value in the 15th entry of the data vector. If all entries of the data vector have frequency 1, leave the **Frequency** field blank.
- **Data set name** — Enter a name for the data set that you import from the workspace, such as `My data`.

After you have entered the information in the preceding fields, click **Create Data Set** to create the data set `My data`.

## Manage Data Sets

View and manage the data sets that you create using the **Manage data sets** pane. When you create a data set, its name appears in the **Data sets** list. The following figure shows the **Manage data sets** pane after creating the data set `My data`.



For each data set in the **Data sets** list, you can:

- Select the **Plot** check box to display a plot of the data in the main Distribution Fitter app window. When you create a new data set, **Plot** is selected by default. Clearing the **Plot** check box removes the data from the plot in the main window. You can specify the type of plot displayed in the **Display type** field in the main window.
- If **Plot** is selected, you can also select **Bounds** to display confidence interval bounds for the plot in the main window. These bounds are pointwise confidence bounds around the empirical estimates of these functions. The bounds are displayed only when you set **Display Type** in the main window to one of the following:
  - Cumulative probability (CDF)
  - Survivor function
  - Cumulative hazard

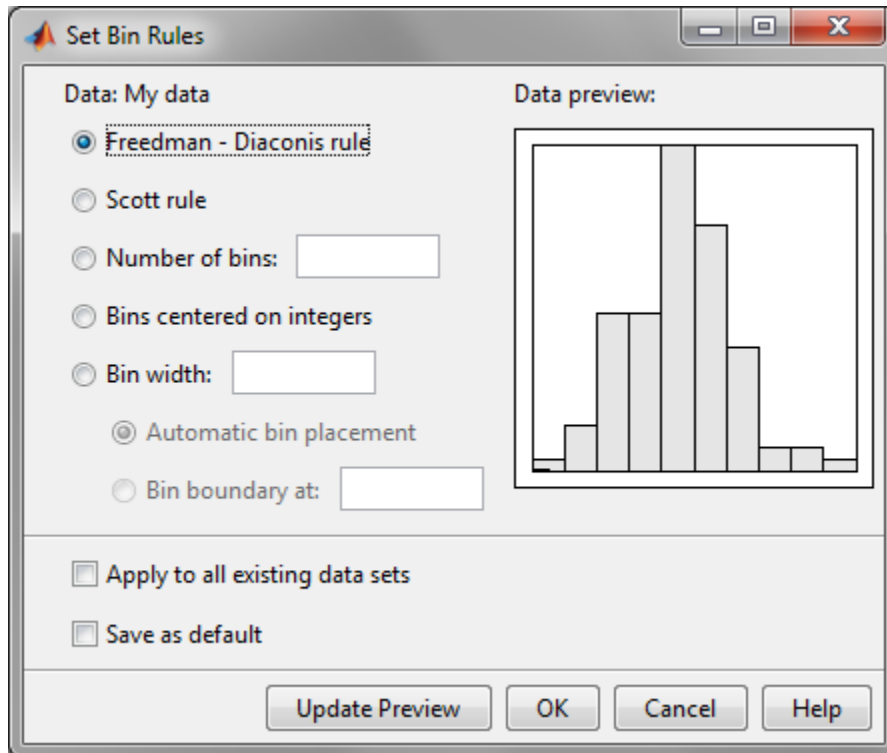
The Distribution Fitter app cannot display confidence bounds on density (PDF), quantile (inverse CDF), or probability plots. Clearing the **Bounds** check box removes the confidence bounds from the plot in the main window.

When you select a data set from the list, you can access the following buttons:

- **View** — Display the data in a table in a new window.
- **Set Bin Rules** — Defines the histogram bins used in a density (PDF) plot.
- **Rename** — Rename the data set.
- **Delete** — Delete the data set.

### Set Bin Rules

To set bin rules for the histogram of a data set, click **Set Bin Rules** to open the **Set Bin Width Rules** dialog box.



You can select from the following rules:

- **Freedman-Diaconis rule** — Algorithm that chooses bin widths and locations automatically, based on the sample size and the spread of the data. This rule, which is the default, is suitable for many kinds of data.
- **Scott rule** — Algorithm intended for data that are approximately normal. The algorithm chooses bin widths and locations automatically.
- **Number of bins** — Enter the number of bins. All bins have equal widths.
- **Bins centered on integers** — Specifies bins centered on integers.
- **Bin width** — Enter the width of each bin. If you select this option, you can also select:
  - **Automatic bin placement** — Place the edges of the bins at integer multiples of the **Bin width**.

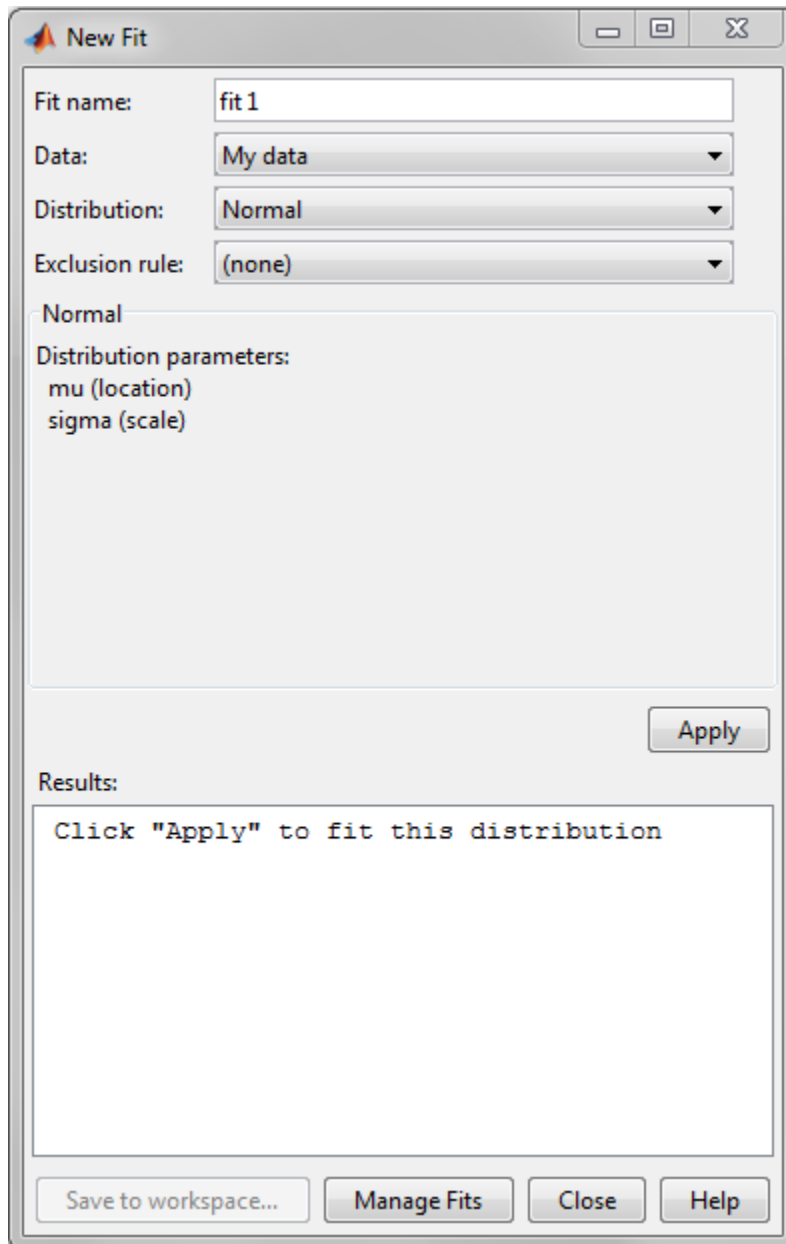
- **Bin boundary at** — Enter a scalar to specify the boundaries of the bins. The boundary of each bin is equal to this scalar plus an integer multiple of the **Bin width**.

You can also:

- **Apply to all existing data sets** — Apply the rule to all data sets. Otherwise, the rule is applied only to the data set currently selected in the Data dialog box.
- **Save as default** — Apply the current rule to any new data sets that you create. You can set default bin width rules by selecting **Set Default Bin Rules** from the **Tools** menu in the main window.

### Create a New Fit

Click the **New Fit** button at the top of the main window to open the New Fit dialog box. If you created the data set `My_data`, it appears in the **Data** field.



Field Name	Description
Fit Name	Enter a name for the fit.
Data	Select the data set to which you want to fit a distribution from the drop-down list.
Distribution	<p>Select the type of distribution to fit from the <b>Distribution</b> drop-down list.</p> <p>Only the distributions that apply to the values of the selected data set appear in the <b>Distribution</b> field. For example, when the data include values that are zero or negative, positive distributions are not displayed .</p> <p>You can specify either a parametric or a nonparametric distribution. When you select a parametric distribution from the drop-down list, a description of its parameters appears. Distribution Fitter estimates these parameters to fit the distribution to the data set. If you select the binomial distribution or the generalized extreme value distribution, you must specify a fixed value for one of the parameters. The pane contains a text field into which you can specify that parameter.</p> <p>When you select <code>Nonparametric fit</code>, options for the fit appear in the pane, as described in “Further Options for Nonparametric Fits” on page 5-80.</p>
Exclusion rule	Specify a rule to exclude some data. Create an exclusion rule by clicking <b>Exclude</b> in the Distribution Fitter app. For more information, see “Exclude Data” on page 5-89.

### Apply the New Fit

Click **Apply** to fit the distribution. For a parametric fit, the **Results** pane displays the values of the estimated parameters. For a nonparametric fit, the **Results** pane displays information about the fit.

When you click **Apply**, the Distribution Fitter app displays a plot of the distribution and the corresponding data.

---

**Note** When you click **Apply**, the title of the dialog box changes to Edit Fit. You can now make changes to the fit you just created and click **Apply** again to save them. After

closing the Edit Fit dialog box, you can reopen it from the Fit Manager dialog box at any time to edit the fit.

---

After applying the fit, you can save the information to the workspace using probability distribution objects by clicking **Save to workspace**.

### Available Distributions

All of the distributions available in the Distribution Fitter app are supported elsewhere in Statistics and Machine Learning Toolbox software. You can use the `fitdist` function to fit any of the distributions supported by the app. Many distributions also have dedicated fitting functions. These functions compute the majority of the fits in the Distribution Fitter app, and are referenced in the following list. Other fits are computed using functions internal to the Distribution Fitter app.

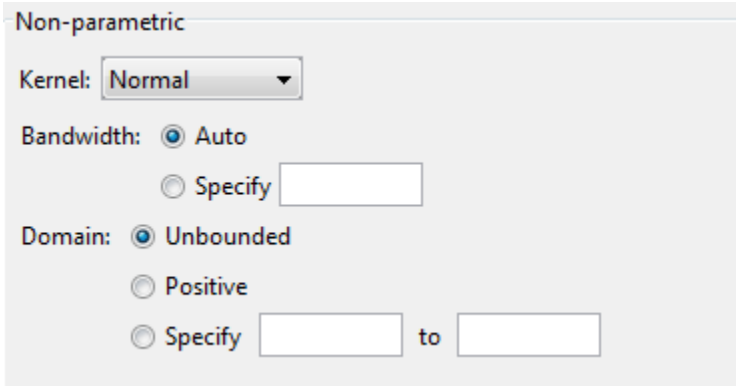
Not all of the distributions listed are available for all data sets. The Distribution Fitter app determines the extent of the data (nonnegative, unit interval, etc.) and displays appropriate distributions in the **Distribution** drop-down list. Distribution data ranges are given parenthetically in the following list.

- Beta on page B-4 (unit interval values) distribution, fit using the function `betafit`.
- Binomial on page B-9 (nonnegative integer values) distribution, fit using the function `binopdf`.
- Birnbaum-Saunders on page B-13 (positive values) distribution.
- Burr Type XII on page B-15 (positive values) distribution.
- Exponential on page B-34 (nonnegative values) distribution, fit using the function `expfit`.
- Extreme value on page B-38 (all values) distribution, fit using the function `evfit`.
- Gamma on page B-47 (positive values) distribution, fit using the function `gamfit`.
- Generalized extreme value on page B-53 (all values) distribution, fit using the function `gevfit`.
- Generalized Pareto on page B-58 (all values) distribution, fit using the function `gpf`.
- Inverse Gaussian on page B-82 (positive values) distribution.
- Logistic on page B-96 (all values) distribution.
- Loglogistic on page B-98 (positive values) distribution.

- Lognormal on page B-100 (positive values) distribution, fit using the function `lognfit`.
- Nakagami on page B-118 (positive values) distribution.
- Negative binomial on page B-120 (nonnegative integer values) distribution, fit using the function `nbinpdf`.
- Nonparametric on page B-86 (all values) distribution, fit using the function `ksdensity`.
- Normal on page B-134 (all values) distribution, fit using the function `normfit`.
- Poisson on page B-143 (nonnegative integer values) distribution, fit using the function `poisspdf`.
- Rayleigh on page B-146 (positive values) distribution using the function `raylfit`.
- Rician on page B-149 (positive values) distribution.
- $t$  location-scale on page B-175 (all values) distribution.
- Weibull on page B-193 (positive values) distribution using the function `wblfit`.

### Further Options for Nonparametric Fits

When you select `Non-parametric` in the **Distribution** field, a set of options appears in the **Non-parametric** pane, as shown in the following figure.



The image shows a software interface for non-parametric fits. It features a title bar labeled "Non-parametric". Below the title bar, there are three main sections of options:

- Kernel:** A dropdown menu currently set to "Normal".
- Bandwidth:** Two radio button options: "Auto" (which is selected) and "Specify" (with an empty text input field next to it).
- Domain:** Three radio button options: "Unbounded" (selected), "Positive", and "Specify" (with two empty text input fields and the word "to" between them).

The options for nonparametric distributions are:

- **Kernel** — Type of kernel function to use.
  - Normal



- Box
- Triangle
- Epanechnikov
- **Bandwidth** — The bandwidth of the kernel smoothing window. Select **Auto** for a default value that is optimal for estimating normal densities. After you click **Apply**, this value appears in the **Fit results** pane. Select **Specify** and enter a smaller value to reveal features such as multiple modes or a larger value to make the fit smoother.
- **Domain** — The allowed  $x$ -values for the density.
  - **Unbounded** — The density extends over the whole real line.
  - **Positive** — The density is restricted to positive values.
  - **Specify** — Enter lower and upper bounds for the domain of the density.




When you select **Positive** or **Specify**, the nonparametric fit has zero probability outside the specified domain.

## Display Results

The Distribution Fitter app window displays plots of:

- The data sets for which you select **Plot** in the Data dialog box.
- The fits for which you select **Plot** in the Fit Manager dialog box.
- Confidence bounds for:
  - The data sets for which you select **Bounds** in the Data dialog box.
  - The fits for which you select **Bounds** in the Fit Manager dialog box.

Adjust the plot display using the buttons at the top of the tool:

-  — Toggle the legend on (default) or off.
-  — Toggle grid lines on or off (default).
-  — Restore default axes limits.

The following fields are available.

### Display Type

Specify the type of plot to display using the **Display Type** field in the main app window. Each type corresponds to a probability function, for example, a probability density function. You can choose from the following display types:

- `Density (PDF)` — Display a probability density function (PDF) plot for the fitted distribution. The main window displays data sets using a probability histogram, in which the height of each rectangle is the fraction of data points that lie in the bin divided by the width of the bin. This makes the sum of the areas of the rectangles equal to 1.
- `Cumulative probability (CDF)` — Display a cumulative probability plot of the data. The main window displays data sets using a cumulative probability step function. The height of each step is the cumulative sum of the heights of the rectangles in the probability histogram.
- `Quantile (inverse CDF)` — Display a quantile (inverse CDF) plot.
- `Probability plot` — Display a probability plot of the data. Specify the type of distribution used to construct the probability plot in the **Distribution** field. This field is only available when you select `Probability plot`. The choices for the distribution are:
  - `Exponential`
  - `Extreme value`
  - `Logistic`
  - `Log-Logistic`
  - `Lognormal`
  - `Normal`
  - `Rayleigh`
  - `Weibull`

You can also create a probability plot against a parametric fit that you create in the **New Fit** pane. When you create these fits, they are added at the bottom of the **Distribution** drop-down list.

- `Survivor function` — Display survivor function plot of the data.
- `Cumulative hazard` — Display cumulative hazard plot of the data.

---

**Note** If the plotted data includes 0 or negative values, some distributions are unavailable.

---

### Confidence Bounds

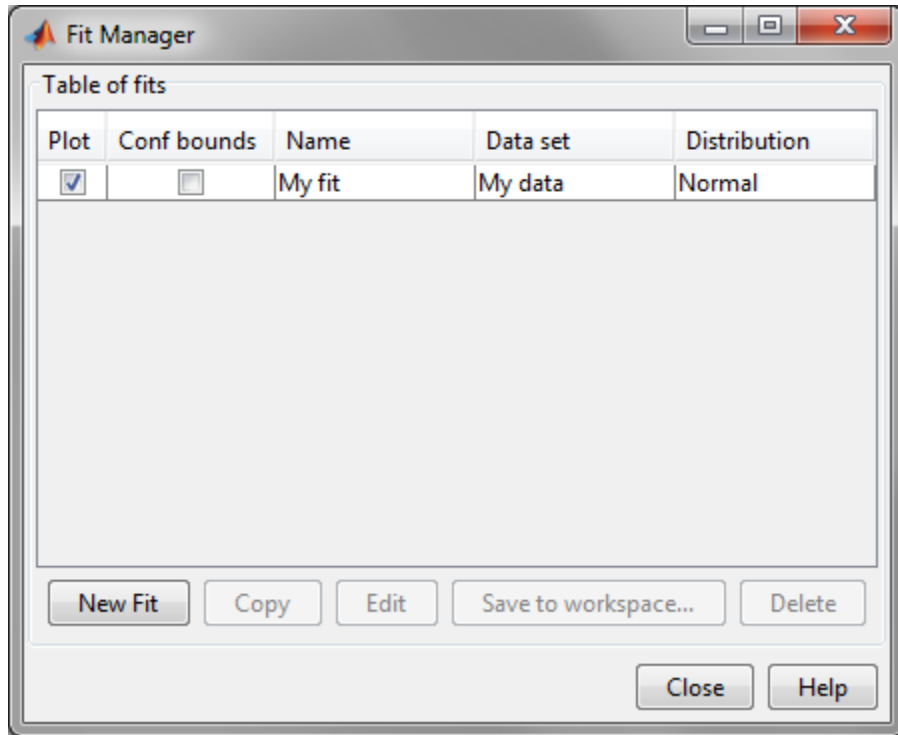
You can display confidence bounds for data sets and fits when you set **Display Type** to Cumulative probability (CDF), Survivor function, Cumulative hazard, or, for fits only, Quantile (inverse CDF).

- To display bounds for a data set, select **Bounds** next to the data set in the **Data sets** pane of the Data dialog box.
- To display bounds for a fit, select **Bounds** next to the fit in the Fit Manager dialog box. Confidence bounds are not available for all fit types.

To set the confidence level for the bounds, select Confidence Level from the **View** menu in the main window and choose from the options.

### Manage Fits

Click the **Manage Fits** button to open the **Fit Manager** dialog box.



The **Table of fits** displays a list of the fits that you create, with the following options:

- **Plot** — Displays a plot of the fit in the main window of the Distribution Fitter app. When you create a new fit, **Plot** is selected by default. Clearing the **Plot** check box removes the fit from the plot in the main window.
- **Bounds** — If you select **Plot**, you can also select **Bounds** to display confidence bounds in the plot. The bounds are displayed when you set **Display Type** in the main window to one of the following:
  - Cumulative probability (CDF)
  - Quantile (inverse CDF)
  - Survivor function
  - Cumulative hazard

The Distribution Fitter app cannot display confidence bounds on density (PDF) or probability plots. Bounds are not supported for nonparametric fits and some parametric fits.

Clearing the **Bounds** check box removes the confidence intervals from the plot in the main window.

When you select a fit in the **Table of fits**, the following buttons are enabled below the table:

- **New Fit** — Open a New Fit window.
- **Copy** — Create a copy of the selected fit.
- **Edit** — Open an Edit Fit dialog box, to edit the fit.

---

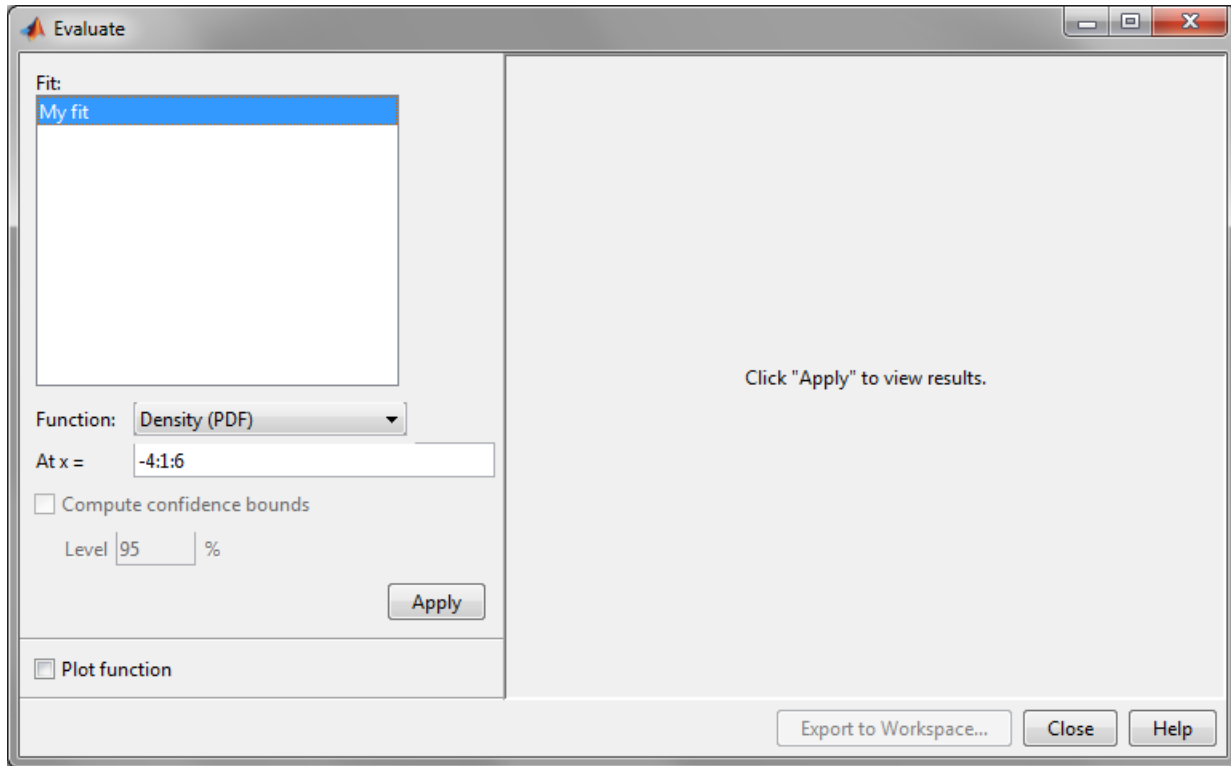
**Note** You can edit only the currently selected fit in the Edit Fit dialog box. To edit a different fit, select it in the **Table of fits** and click **Edit** to open another Edit Fit dialog box.

---

- **Save to workspace** — Save the selected fit as a distribution object.
- **Delete** — Delete the selected fit.

## Evaluate Fits

Use the **Evaluate** dialog box to evaluate your fitted distribution at any data points you choose. To open the dialog box, click the **Evaluate** button.



In the **Evaluate** dialog box, choose from the following items:

- **Fit** pane — Display the names of existing fits. Select one or more fits that you want to evaluate. Using your platform specific functionality, you can select multiple fits.
- **Function** — Select the type of probability function that you want to evaluate for the fit. The available functions are:
  - Density (PDF) — Computes a probability density function.
  - Cumulative probability (CDF) — Computes a cumulative probability function.
  - Quantile (inverse CDF) — Computes a quantile (inverse CDF) function.
  - Survivor function — Computes a survivor function.
  - Cumulative hazard — Computes a cumulative hazard function.

- Hazard rate — Computes the hazard rate.
- **At  $\mathbf{x} =$**  — Enter a vector of points or the name of a workspace variable containing a vector of points at which you want to evaluate the distribution function. If you change **Function** to `Quantile` (inverse CDF), the field name changes to **At  $\mathbf{p} =$** , and you enter a vector of probability values.
- **Compute confidence bounds** — Select this box to compute confidence bounds for the selected fits. The check box is enabled only if you set **Function** to one of the following:
  - Cumulative probability (CDF)
  - Quantile (inverse CDF)
  - Survivor function
  - Cumulative hazard

The Distribution Fitter app cannot compute confidence bounds for nonparametric fits and for some parametric fits. In these cases, it returns NaN for the bounds.

- **Level** — Set the level for the confidence bounds.
- **Plot function** — Select this box to display a plot of the distribution function, evaluated at the points you enter in the **At  $\mathbf{x} =$**  field, in a new window.

---

**Note** The settings for **Compute confidence bounds**, **Level**, and **Plot function** do not affect the plots that are displayed in the main window of the Distribution Fitter app. The settings apply only to plots you create by clicking **Plot function** in the Evaluate window.

---

To apply these evaluation settings to the selected fit, click **Apply**. The following figure shows the results of evaluating the cumulative density function for the fit **My fit**, at the points in the vector `-4:1:6`.

X	My fit		
	F(X)	LB	UB
-4	0.00268	0.00064	0.0094
-3	0.01498	0.00568	0.03518
-2	0.05984	0.03225	0.10321
-1	0.1732	0.11948	0.24022
0	0.37179	0.29863	0.44992
1	0.61309	0.53479	0.68703
2	0.81644	0.74822	0.87178
3	0.93529	0.89025	0.96446
4	0.98345	0.96201	0.99356
5	0.99698	0.98969	0.99925
6	0.99961	0.99782	0.99995

The columns of the table to the right of the **Fit** pane display the following values:

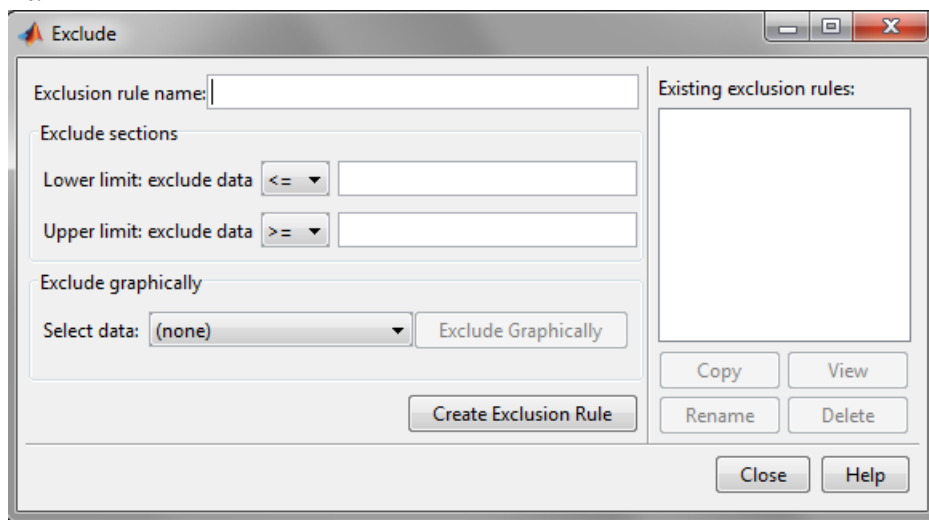
- X — The entries of the vector that you enter in **At x =** field.
- F(X)— The corresponding values of the CDF at the entries of X.
- LB — The lower bounds for the confidence interval, if you select **Compute confidence bounds**.
- UB — The upper bounds for the confidence interval, if you select **Compute confidence bounds**.



To save the data displayed in the table to a matrix in the MATLAB workspace, click **Export to Workspace**.

## Exclude Data

To exclude values from fit, open the **Exclude** window by clicking the **Exclude** button. In the **Exclude** window, you can create rules for excluding specified data values. When you create a new fit in the **New Fit** window, you can use these rules to exclude data from the fit.

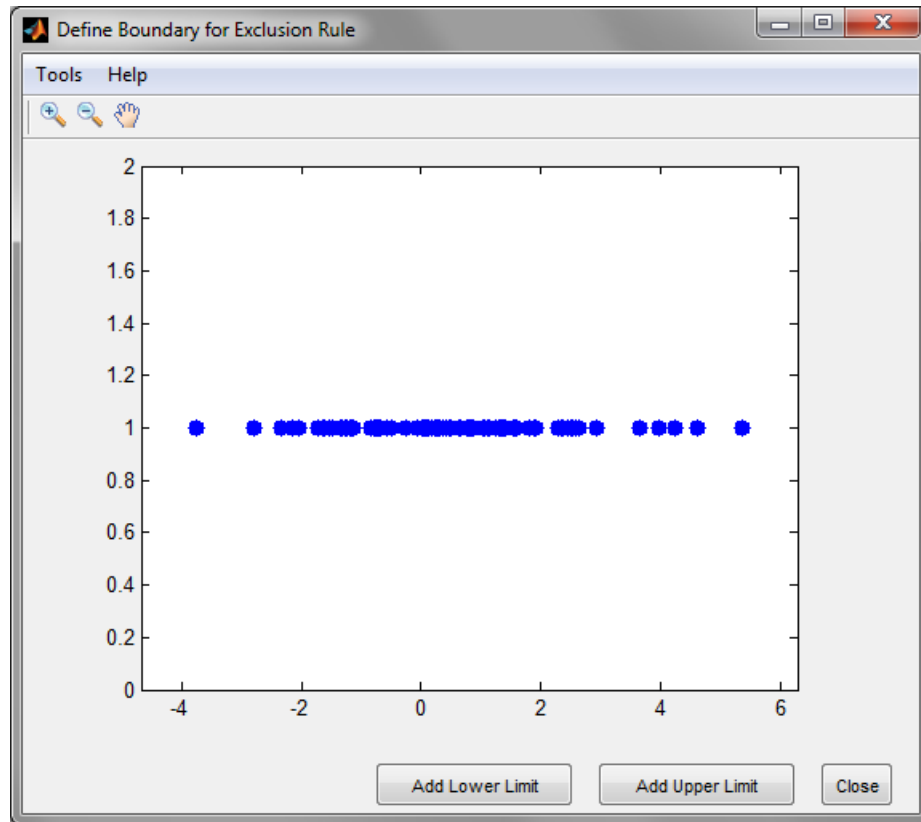


To create an exclusion rule:

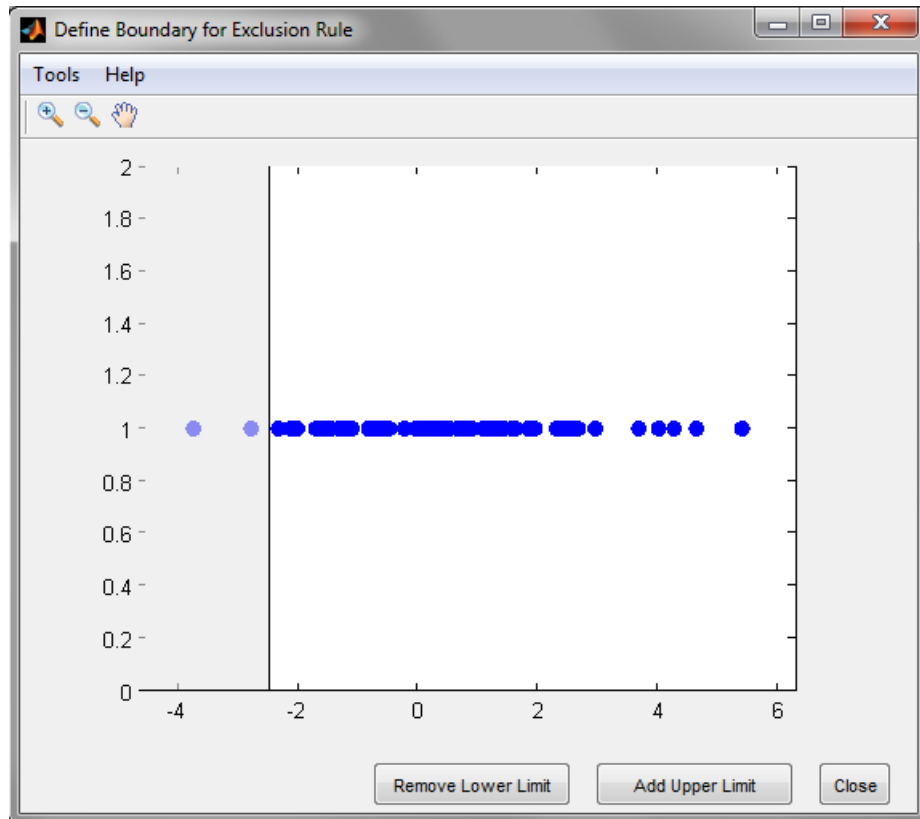
- 1 **Exclusion Rule Name**— Enter a name for the exclusion rule.
- 2 **Exclude Sections**— Specify bounds for the excluded data:
  - In the **Lower limit: exclude data** drop-down list, select  $\leq$  or  $<$  and enter a scalar value in the field to the right. Depending on which operator you select, the app excludes from the fit any data values that are less than or equal to the scalar value, or less than the scalar value, respectively.
  - In the **Upper limit: exclude data** drop-down list, select  $\geq$  or  $>$  and enter a scalar value in the field to the right. Depending on which operator you select, the app excludes from the fit any data values that are greater than or equal to the scalar value, or greater than the scalar value, respectively.

OR

Click the **Exclude Graphically** button to define the exclusion rule by displaying a plot of the values in a data set and selecting the bounds for the excluded data. For example, if you created the data set *My data* as described in *Create and Manage Data Sets*, select it from the drop-down list next to **Exclude graphically**, and then click the **Exclude graphically** button. The app displays the values in *My data* in a new window.



To set a lower limit for the boundary of the excluded region, click **Add Lower Limit**. The app displays a vertical line on the left side of the plot window. Move the line to the point you where you want the lower limit, as shown in the following figure.



Move the vertical line to change the value displayed in the **Lower limit: exclude data** field in the **Exclude** window.

Exclude sections

Lower limit: exclude data  $\leq$  -2.5

Upper limit: exclude data  $\geq$

The value displayed corresponds to the  $x$ -coordinate of the vertical line.

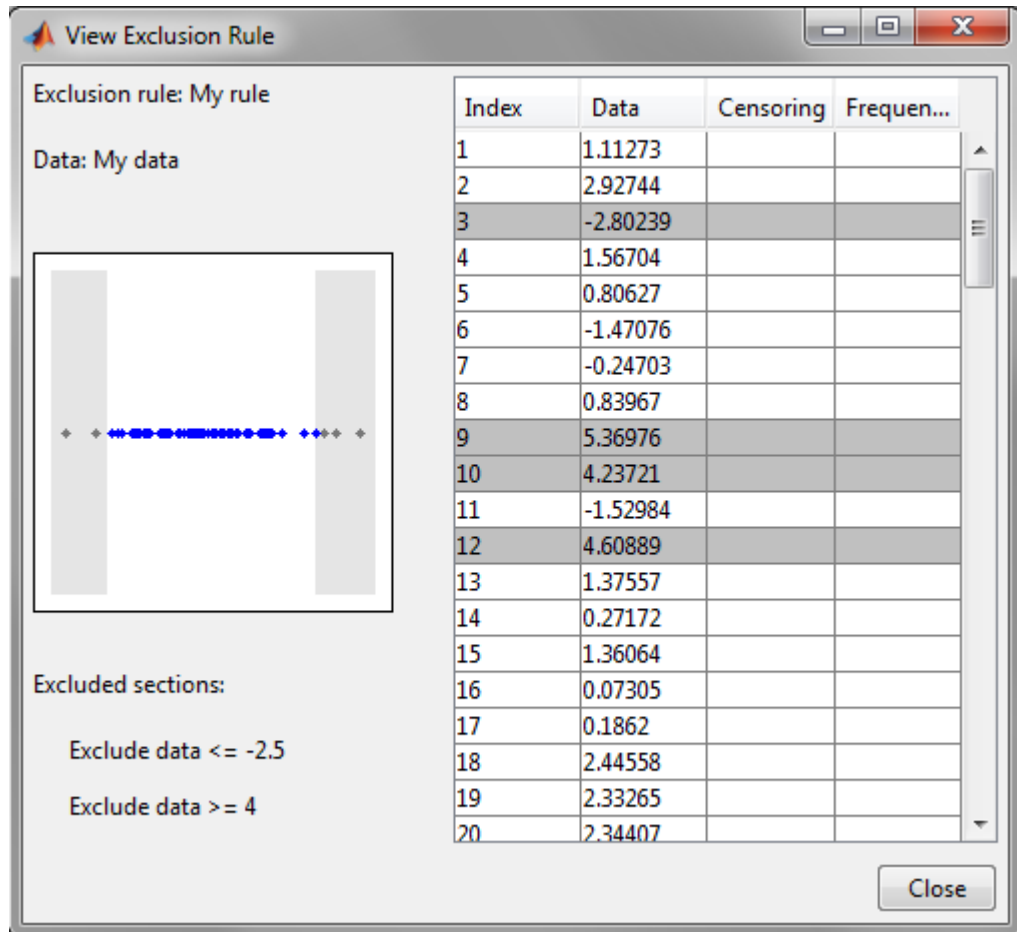
Similarly, you can set the upper limit for the boundary of the excluded region by clicking **Add Upper Limit**, and then moving the vertical line that appears at the

right side of the plot window. After setting the lower and upper limits, click **Close** and return to the Exclude window.

- 3 Create Exclusion Rule**—Once you have set the lower and upper limits for the boundary of the excluded data, click **Create Exclusion Rule** to create the new rule. The name of the new rule appears in the **Existing exclusion rules** pane.

Selecting an exclusion rule in the **Existing exclusion rules** pane enables the following buttons:

- **Copy** — Creates a copy of the rule, which you can then modify. To save the modified rule under a different name, click **Create Exclusion Rule**.
- **View** — Opens a new window in which you can see the data points excluded by the rule. The following figure shows a typical example.



The shaded areas in the plot graphically display which data points are excluded. The table to the right lists all data points. The shaded rows indicate excluded points:

- **Rename** — Rename the rule.
- **Delete** — Delete the rule.

After you define an exclusion rule, you can use it when you fit a distribution to your data. The rule does not exclude points from the display of the data set.

## Save and Load Sessions

Save your work in the current session, and then load it in a subsequent session, so that you can continue working where you left off.

### Save a Session

To save the current session, from the **File** menu in the main window, select `Save Session`. A dialog box opens and prompts you to enter a file name, for example `my_session.dfit`. Click **Save** to save the following items created in the current session:

- Data sets
- Fits
- Exclusion rules
- Plot settings
- Bin width rules

### Load a Session

To load a previously saved session, from the **File** menu in the main window, select `Load Session`. Enter the name of a previously saved session. Click **Open** to restore the information from the saved session to the current session.

## Generate a File to Fit and Plot Distributions

Use the `Generate Code` option in the **File** to create a file that:

- Fits the distributions in the current session to any data vector in the MATLAB workspace.
- Plots the data and the fits.

After you end the current session, you can use the file to create plots in a standard MATLAB figure window, without reopening the `Distribution Fitter` app.

As an example, if you created the fit described in “Create a New Fit” on page 5-76, do the following steps:

- 1 From the **File** menu, select `Generate Code`.

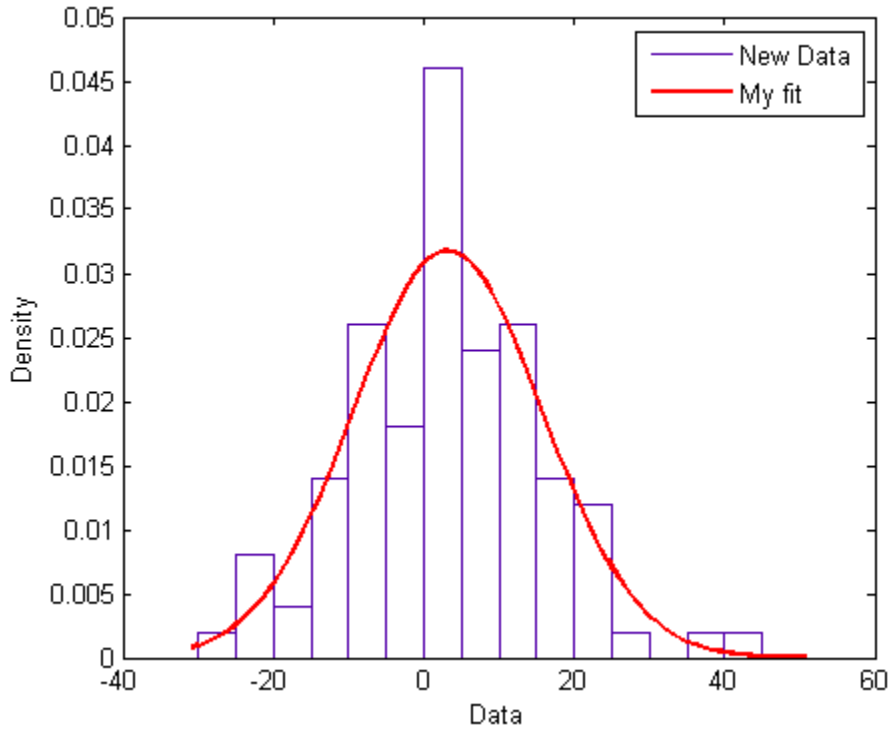
- 2 In the MATLAB Editor window, choose **File > Save as**. Save the file as `normal_fit.m` in a folder on the MATLAB path.

You can then apply the function `normal_fit` to any vector of data in the MATLAB workspace. For example, the following commands:

```
new_data = normrnd(4.1, 12.5, 100, 1);  
newfit = normal_fit(new_data)  
legend('New Data', 'My fit')
```

generate `newfit`, a fitted normal distribution of the data. The commands also generate a plot of the data and the fit.

```
newfit =  
  
normal distribution  
  
mu = 3.19148  
sigma = 12.5631
```



---

**Note** By default, the file labels the data in the legend using the same name as the data set in the Distribution Fitter app. You can change the label using the `legend` command, as illustrated by the preceding example.

---



## Fit a Distribution Using the Distribution Fitter App

This example shows how you can use the Distribution Fitter app to interactively fit a probability distribution to data.

### In this section...

“Step 1: Load Sample Data” on page 5-97

“Step 2: Import Data” on page 5-97

“Step 3: Create a New Fit” on page 5-99

“Step 4: Create and Manage Additional Fits” on page 5-105

### Step 1: Load Sample Data

Load the sample data.

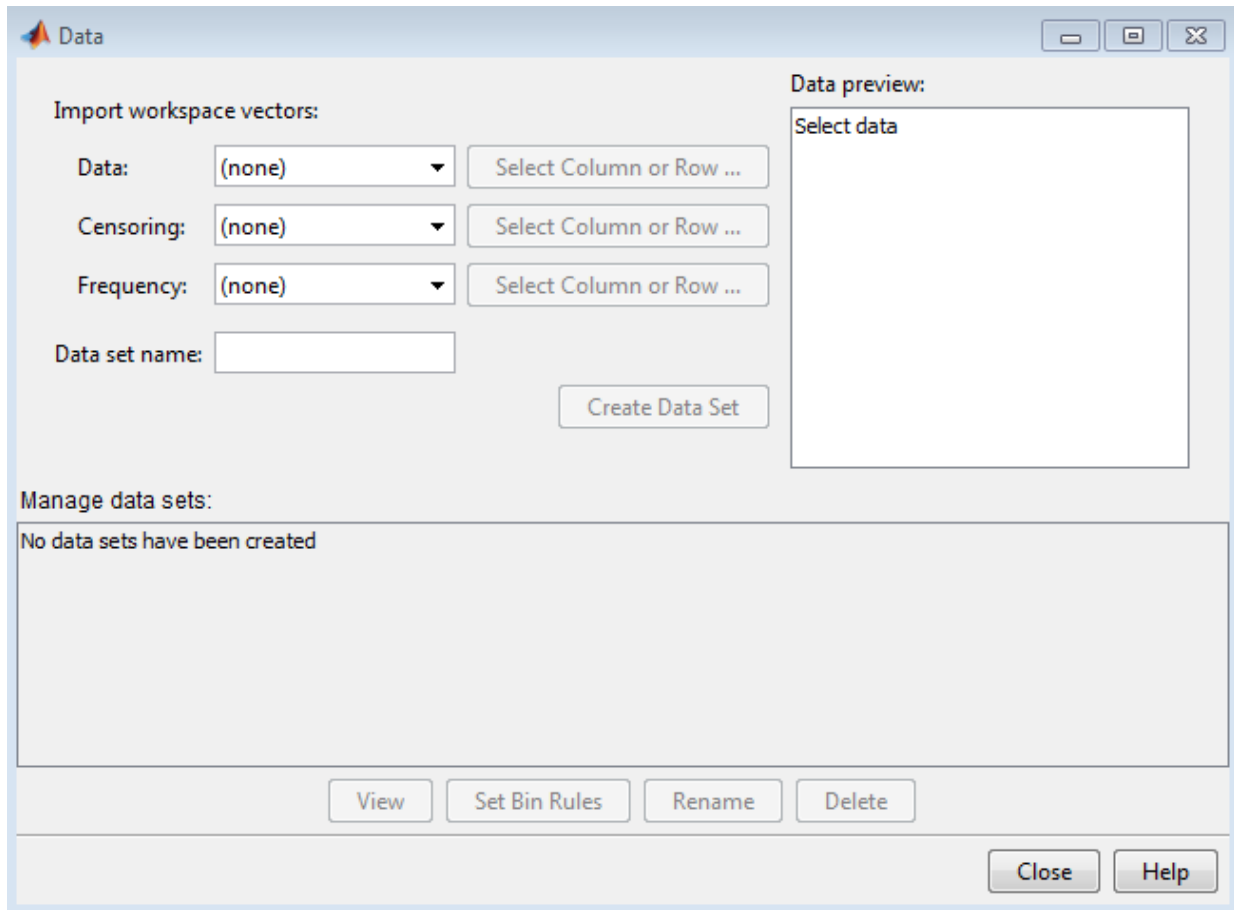
```
load carsmall
```

### Step 2: Import Data

Open the Distribution Fitter tool.

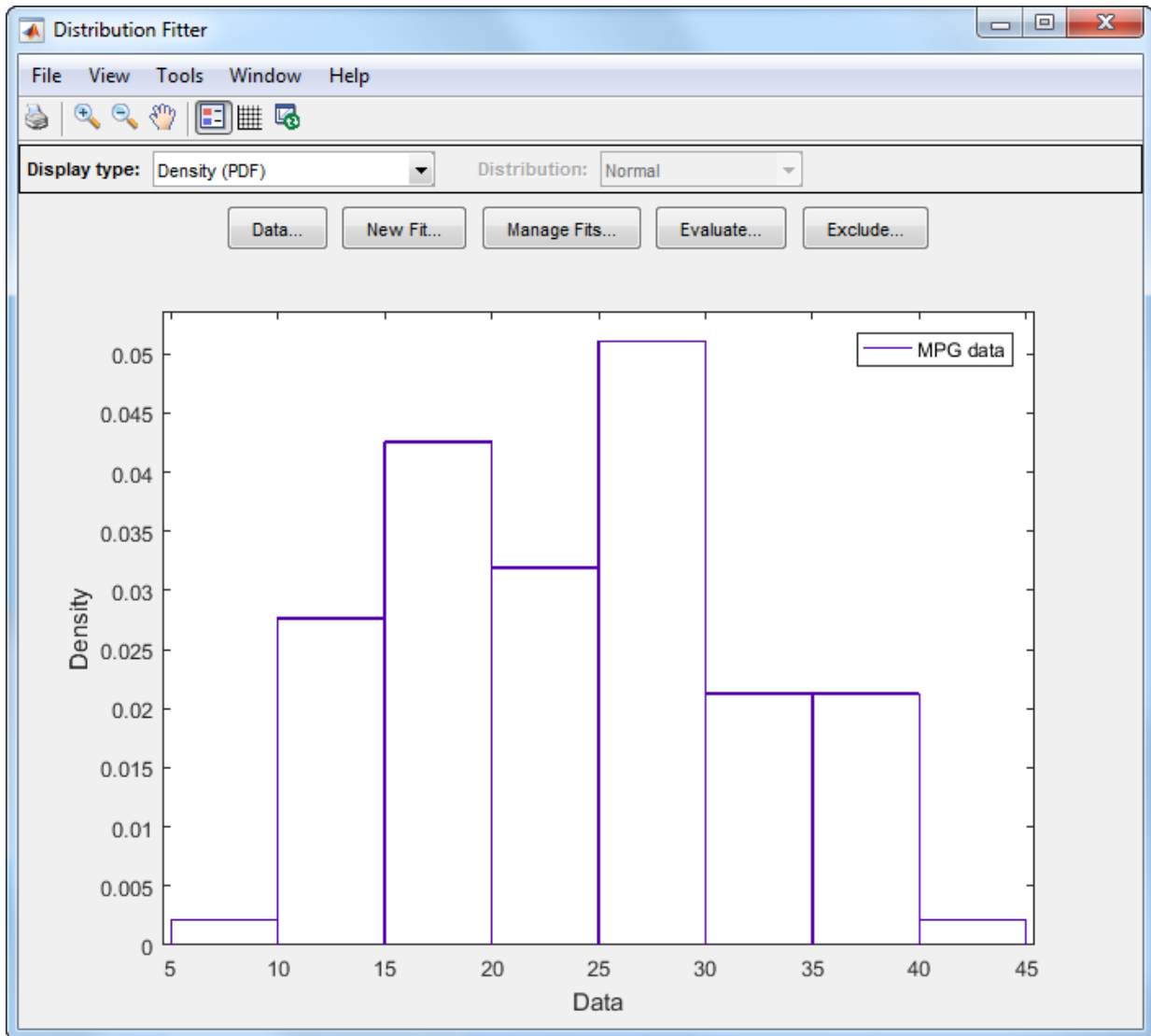
```
distributionFitter
```

To import the vector `MPG` into the Distribution Fitter app, click the **Data** button. The **Data** dialog box opens.



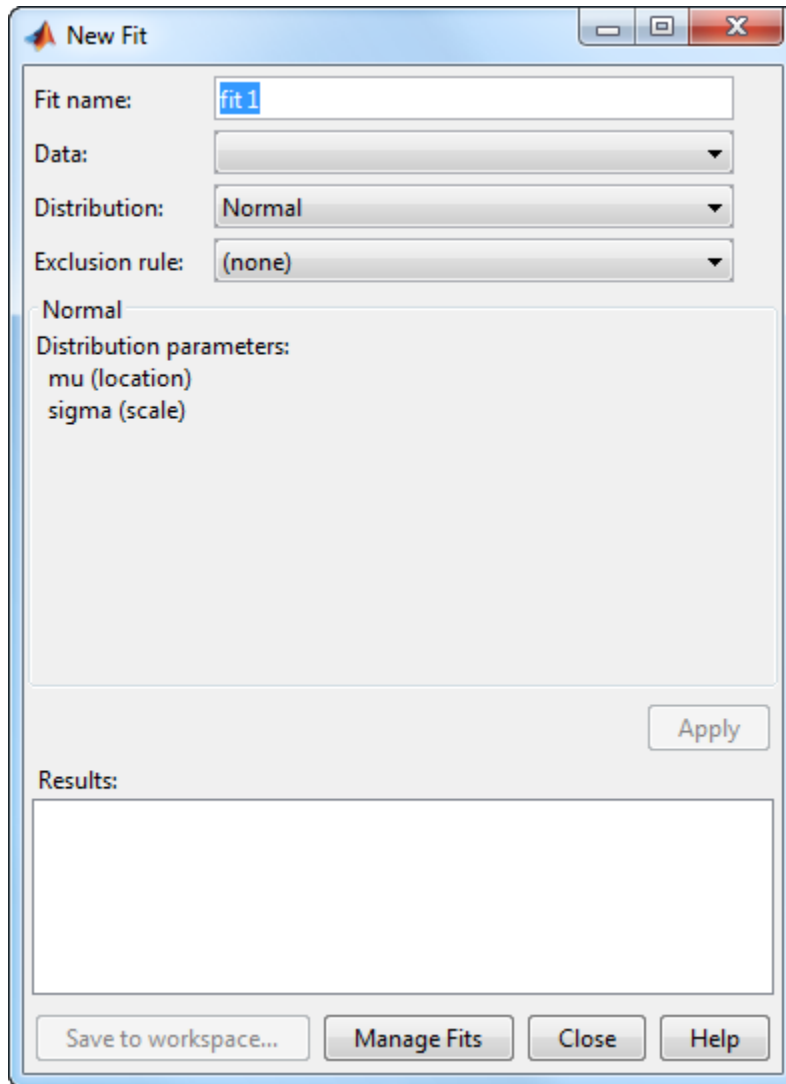
The **Data** field displays all numeric arrays in the MATLAB workspace. From the drop-down list, select `MPG`. A histogram of the selected data appears in the **Data preview** pane.

In the **Data set name** field, type a name for the data set, such as `MPG data`, and click **Create Data Set**. The main window of the Distribution Fitter app now displays a larger version of the histogram in the **Data preview** pane.



### Step 3: Create a New Fit

To fit a distribution to the data, in the main window of the Distribution Fitter app, click **New Fit**.

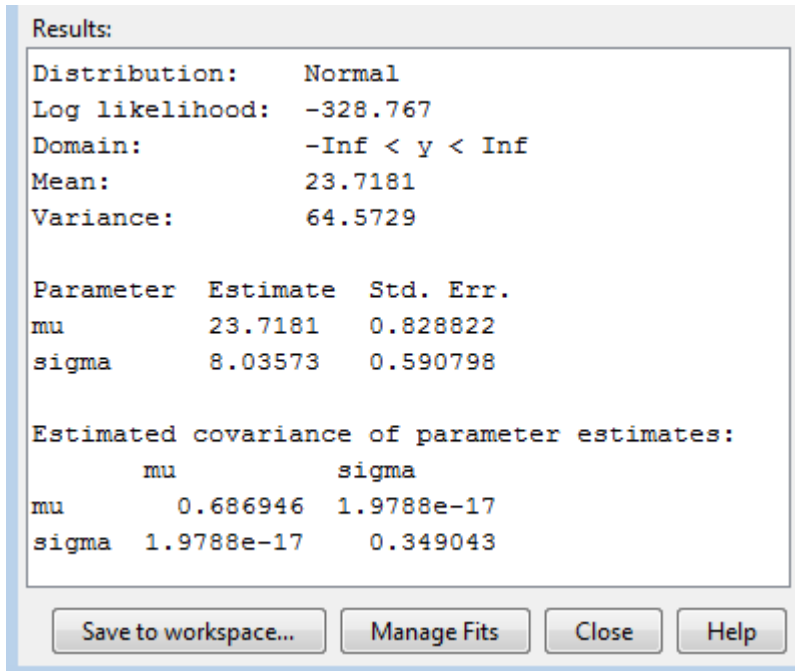


To fit a normal distribution to My data:

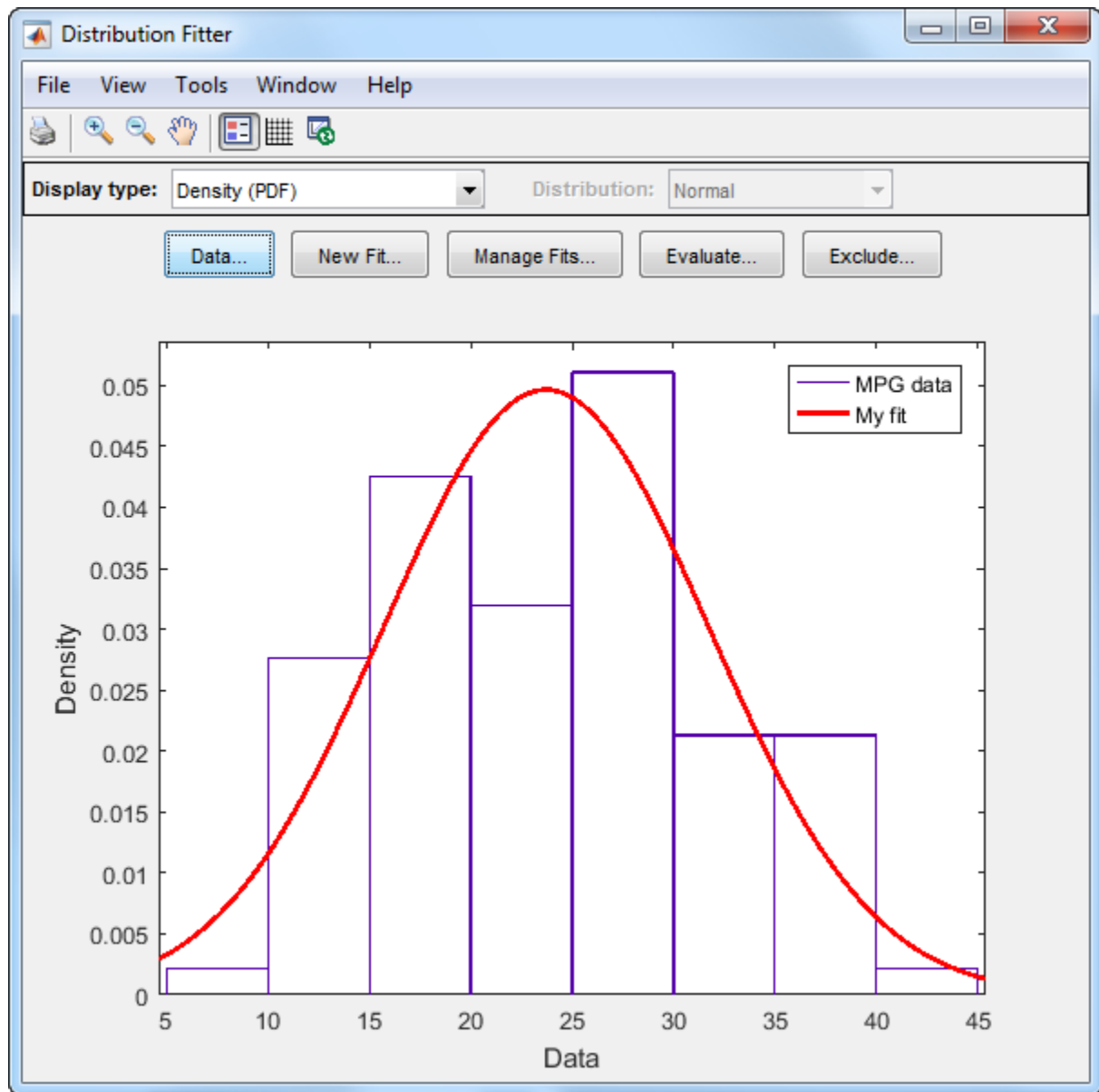
- 1 In the **Fit name** field, enter a name for the fit, such as My fit.
- 2 From the drop-down list in the **Data** field, select MPG data.

- 3 Confirm that `Normal` is selected from the drop-down menu in the **Distribution** field.
- 4 Click **Apply**.

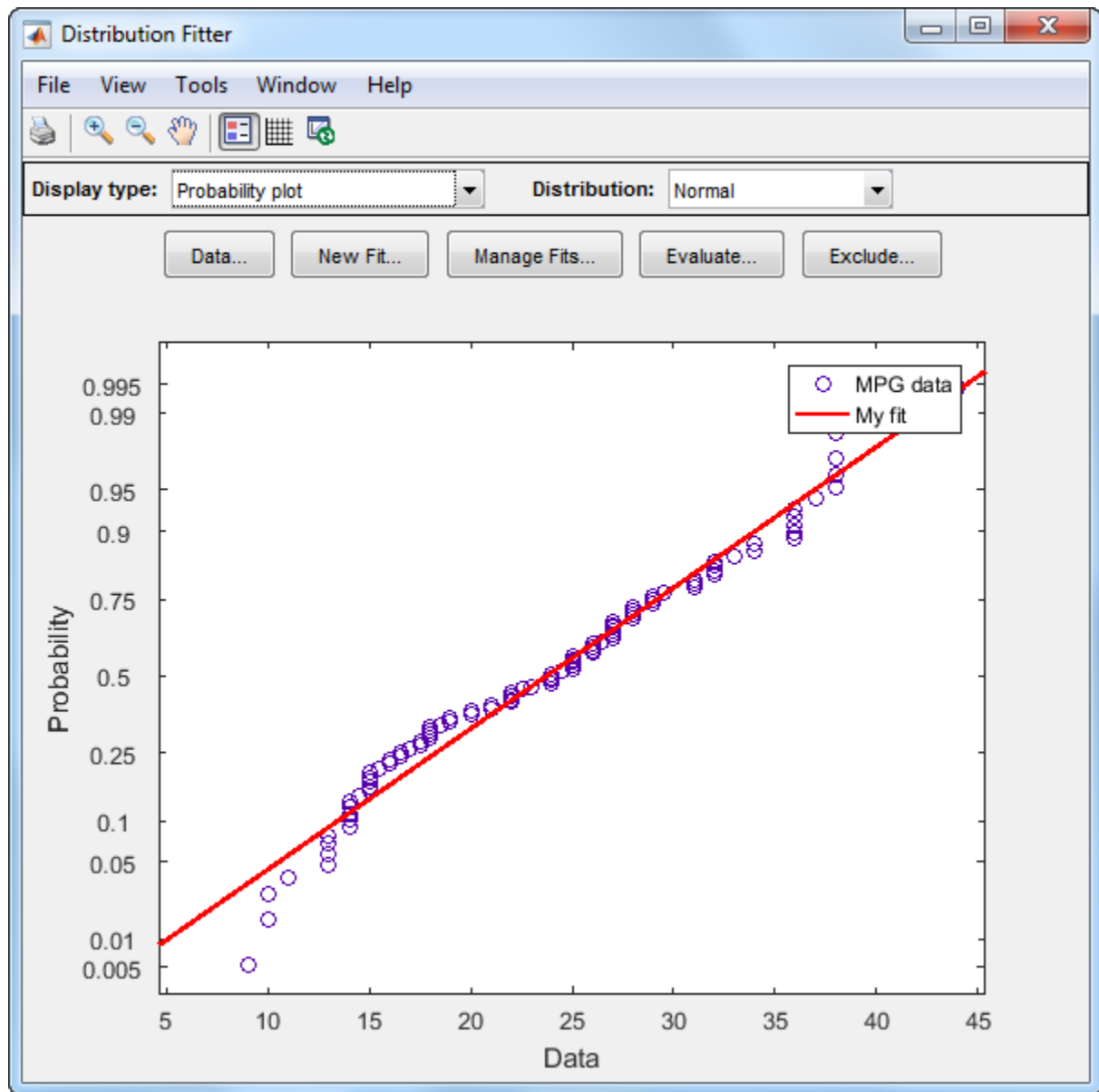
The **Results** pane displays the mean and standard deviation of the normal distribution that best fits `MPG` data.



The Distribution Fitter app main window displays a plot of the normal distribution with this mean and standard deviation.



Based on the plot, a normal distribution does not appear to provide a good fit for the MPG data. To obtain a better evaluation, select Probability plot from the **Display type** drop-down list. Confirm that the **Distribution** drop-down list is set to Normal. The main window displays the following figure.





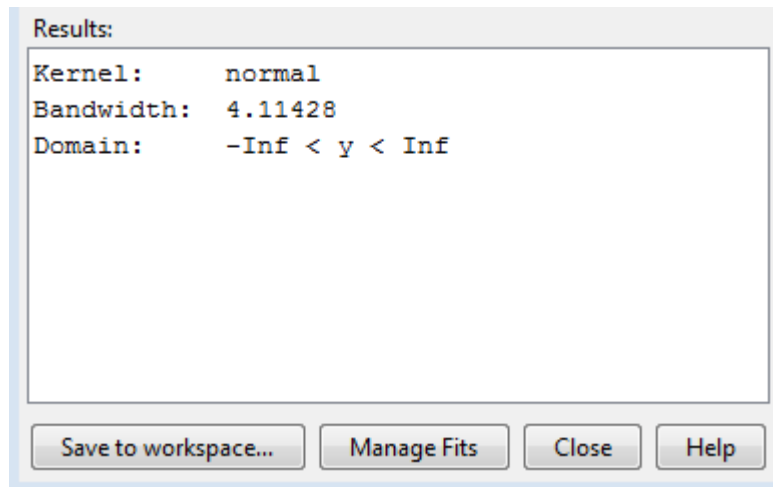
The normal probability plot shows that the data deviates from normal, especially in the tails.

## Step 4: Create and Manage Additional Fits

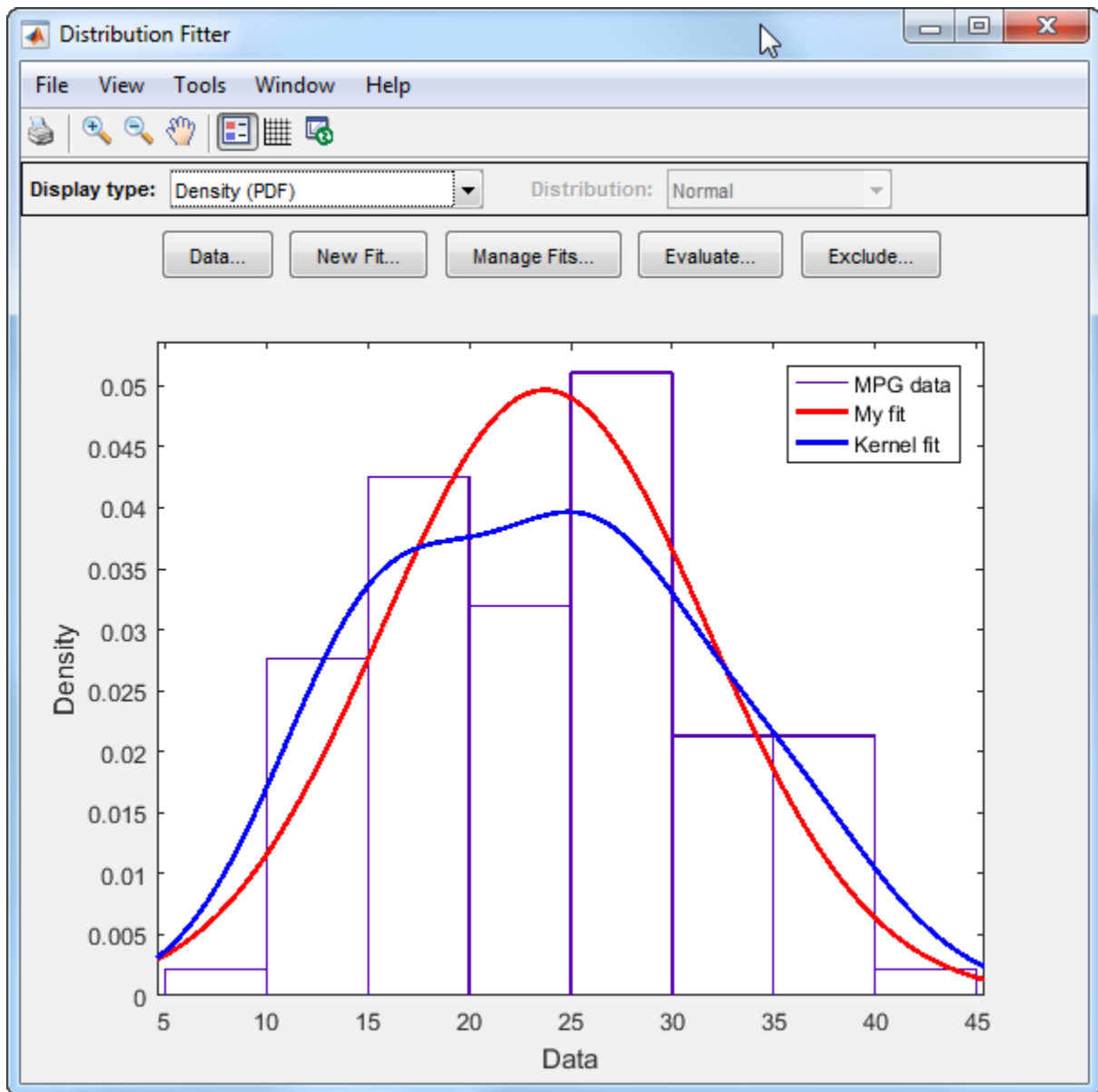
The MPG data pdf indicates that the data has two peaks. Try fitting a nonparametric kernel distribution to obtain a better fit for this data.

- 1 Click **Manage Fits**. In the dialog box, click **New Fit**.
- 2 In the **Fit name** field, enter a name for the fit, such as `Kernel fit`.
- 3 From the drop-down list in the **Data** field, select `MPG data`.
- 4 From the drop-down list in the **Distribution** field, select **Non-parametric**. This enables several options in the **Non-parametric** pane, including **Kernel**, **Bandwidth**, and **Domain**. For now, accept the default value to apply a normal kernel shape and automatically determine the kernel bandwidth (using **Auto**). For more information about nonparametric kernel distributions, see “Kernel Distribution” on page B-86.
- 5 Click **Apply**.

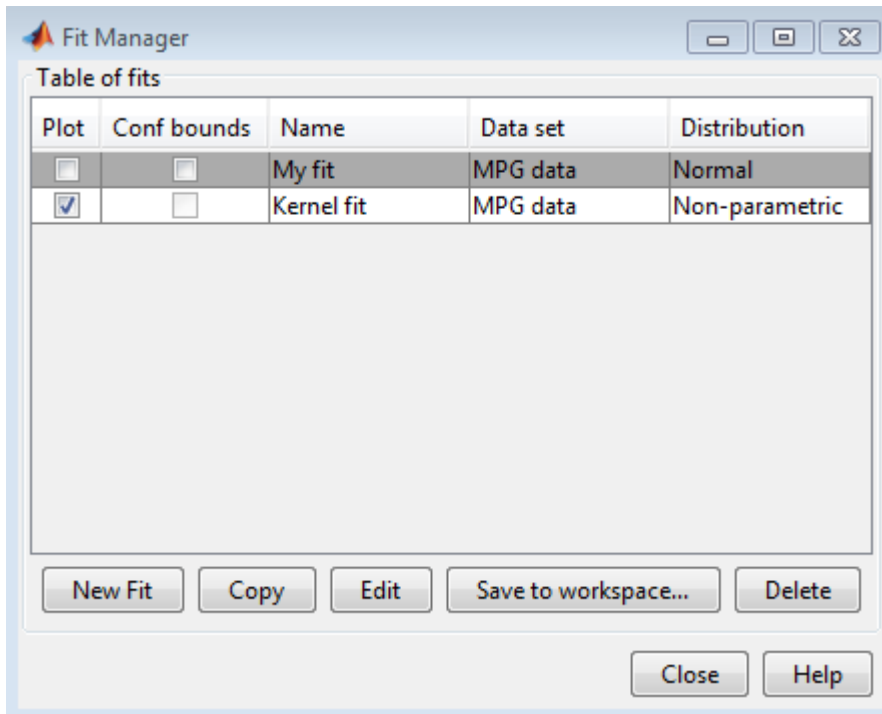
The **Results** pane displays the kernel type, bandwidth, and domain of the nonparametric distribution fit to MPG data.



The main window displays plots of the original MPG data with the normal distribution and nonparametric kernel distribution overlaid. To visually compare these two fits, select `Density (PDF)` from the **Display type** drop-down list.



To include only the nonparametric kernel fit line (`Kernel fit`) on the plot, click `Manage Fits`. In the **Table of fits** pane, locate the row for the normal distribution fit (`My fit`) and clear the box in the **Plot** column.



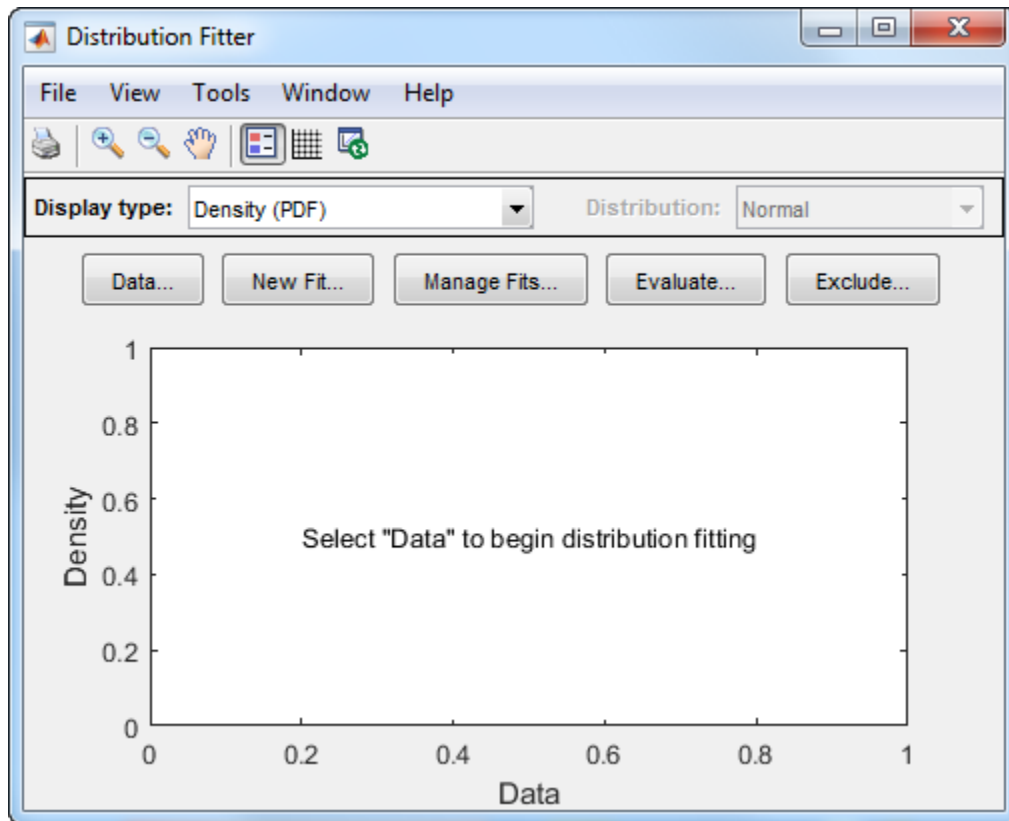
## Custom Distributions Using the Distribution Fitter App

You can use the Distribution Fitter app to fit distributions not supported by the Statistics and Machine Learning Toolbox by defining a custom distribution.

### Opening the Distribution Fitter App

To open the Distribution Fitter app, enter the command

```
distributionFitter
```



Alternatively, click Distribution Fitter on the Apps tab.

### Defining Custom Distributions

To define a custom distribution, select `Define Custom Distribution` from the **File** menu. This opens a file template in the MATLAB editor. You then edit this file so that it computes the distribution you want.

The template includes example code that computes the Laplace distribution. Follow the instructions in the template to define your own custom distribution.

To save your custom distribution, create a directory called `+prob` on your path. Save the file in this directory using a name that matches your distribution name. If you save the template in a folder on the MATLAB path, under its default name `dfittooldists.m`, the Distribution Fitter app reads it in automatically when you start the tool. You can also save the template under a different name, such as `Laplace.m`, and then import the custom distribution as described in the following section.

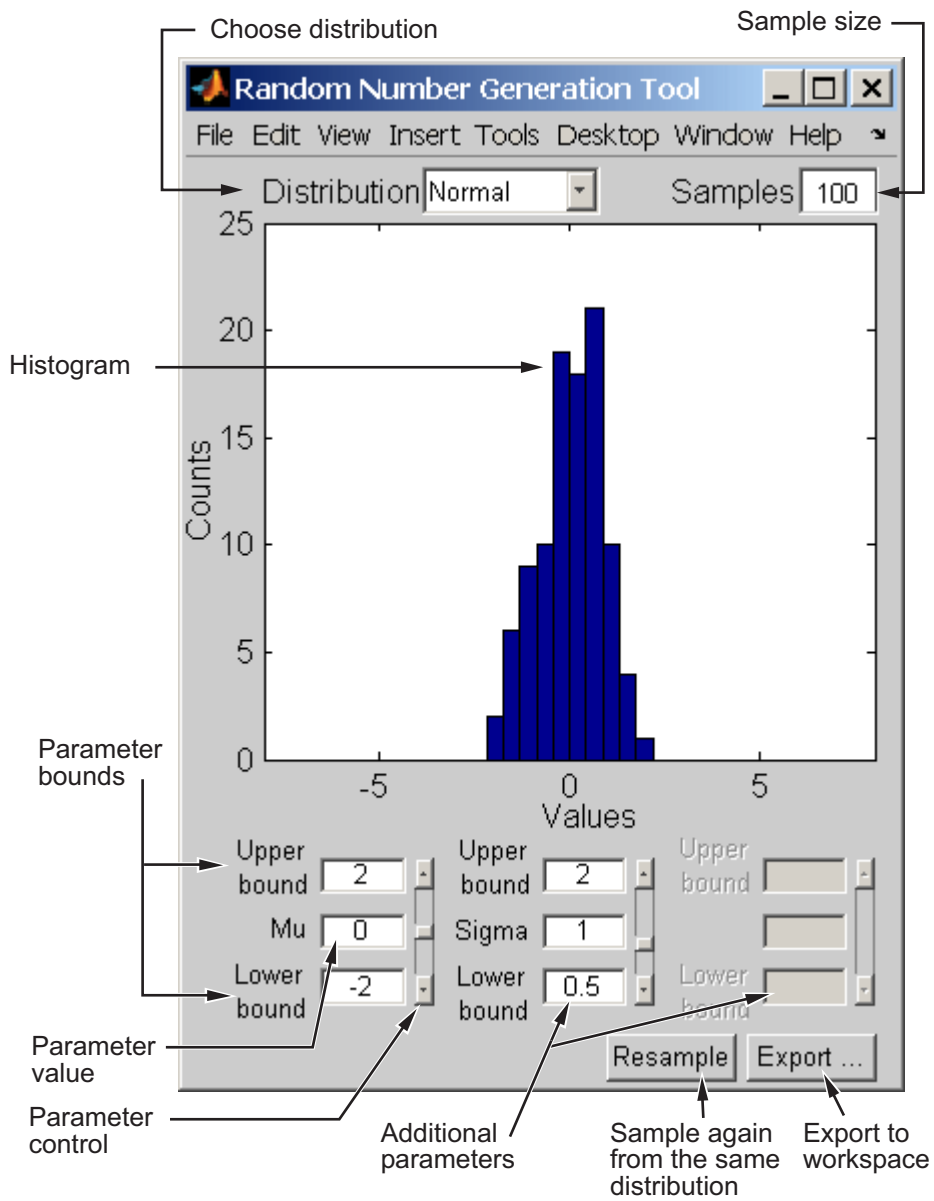
### Importing Custom Distributions

To import a custom distribution, select `Import Custom Distributions` from the **File** menu. This opens a dialog box in which you can select the file that defines the distribution. For example, if you created the file `Laplace.m`, as described in the preceding section, the `New Parametric Distribution List` dialog that launches when you select `Import Custom Distributions` now includes `Laplace`. In addition, the **Distribution** field of the `New Fit` window also contains the option `Laplace`.

## Explore the Random Number Generation UI

The Random Number Generation user interface (UI) generates random samples from specified probability distributions, and displays the samples as histograms. Use the interface to explore the effects of changing parameters and sample size on the distributions.

Run the user interface by typing `randtool` at the command line.



Start by selecting a distribution, then enter the desired sample size.



You can also

- Use the controls at the bottom of the window to set parameter values for the distribution and to change their upper and lower bounds.
- Draw another sample from the same distribution, with the same size and parameters.
- Export the current sample to your workspace. A dialog box enables you to provide a name for the sample.

## Compare Multiple Distribution Fits

This example shows how to fit multiple probability distribution objects to the same set of sample data, and obtain a visual comparison of how well each distribution fits the data.

### Step 1. Load sample data.

Load the sample data.

```
load carsmall;
```

This data contains miles per gallon (MPG) measurements for different makes and models of cars, grouped by country of origin (`Origin`), model year (`Model_Year`), and other vehicle characteristics.

### Step 2. Create a nominal array.

Transform `Origin` into a nominal array and remove the Italian car from the sample data.

```
Origin = nominal(Origin);  
MPG2 = MPG(Origin~='Italy');  
Origin2 = Origin(Origin~='Italy');  
Origin2 = droplevels(Origin2, 'Italy');
```

Since there is only one Italian car, `fitdist` cannot fit a distribution to that group. Removing the Italian car from the sample data prevents `fitdist` from producing an error.

### Step 3. Fit multiple distributions by group.

Use `fitdist` to fit Weibull, normal, logistic, and kernel distributions to each country of origin group in the MPG data.

```
[WeiByOrig,Country] = fitdist(MPG2, 'weibull', 'by', Origin2);  
[NormByOrig,Country] = fitdist(MPG2, 'normal', 'by', Origin2);  
[LogByOrig,Country] = fitdist(MPG2, 'logistic', 'by', Origin2);  
[KerByOrig,Country] = fitdist(MPG2, 'kernel', 'by', Origin2);
```

```
WeiByOrig
```

```
WeiByOrig =
```

```

1x5 cell array

Columns 1 through 2

    {1x1 prob.WeibullDistribution}    {1x1 prob.WeibullDistribution}

Columns 3 through 4

    {1x1 prob.WeibullDistribution}    {1x1 prob.WeibullDistribution}

Column 5

    {1x1 prob.WeibullDistribution}

Country

Country =

5x1 cell array

    {'France' }
    {'Germany'}
    {'Japan'   }
    {'Sweden' }
    {'USA'    }

```

Each country group now has four distribution objects associated with it. For example, the cell array `WeiByOrig` contains five Weibull distribution objects, one for each country represented in the sample data. Likewise, the cell array `NormByOrig` contains five normal distribution objects, and so on. Each object contains properties that hold information about the data, distribution, and parameters. The array `Country` lists the country of origin for each group in the same order as the distribution objects are stored in the cell arrays.

#### Step 4. Compute the pdf for each distribution.

Extract the four probability distribution objects for USA and compute the pdf for each distribution. As shown in Step 3, USA is in position 5 in each cell array.

```

WeiUSA = WeiByOrig{5};
NormUSA = NormByOrig{5};

```

```
LogUSA = LogByOrig{5};
KerUSA = KerByOrig{5};

x = 0:1:50;
pdf_Wei = pdf(WeiUSA,x);
pdf_Norm = pdf(NormUSA,x);
pdf_Log = pdf(LogUSA,x);
pdf_Ker = pdf(KerUSA,x);
```

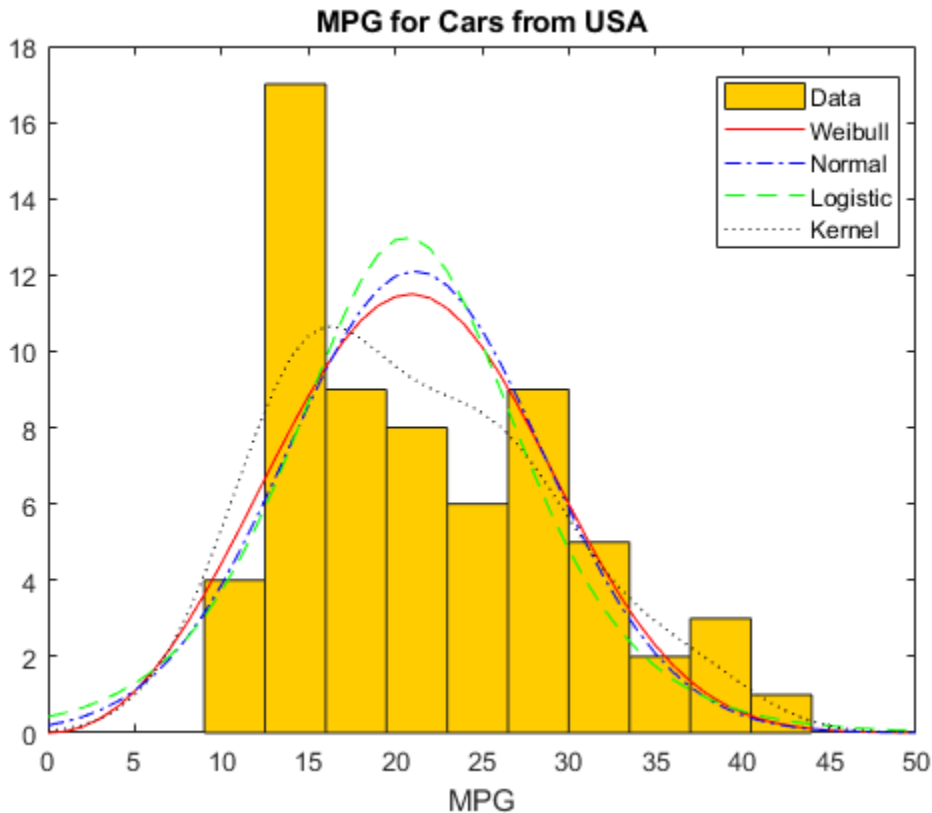
### Step 5. Plot the pdf for each distribution.

Plot the pdf for each distribution fit to the USA data, superimposed on a histogram of the sample data. Scale the density by the histogram area for easier display. Create a histogram of the USA sample data

```
data = MPG(Origin2=='USA');
figure
[n,y] = hist(data,10);
b = bar(y,n,'hist');
set(b,'FaceColor',[1,0.8,0]);

% Scale the density by the histogram area for easier display
area = sum(n)*(y(2)-y(1));
time = 0:50;
pdfWei = pdf(WeiUSA,time);
pdfNorm = pdf(NormUSA,time);
pdfLog = pdf(LogUSA,time);
pdfKer = pdf(KerUSA,time);

% Plot the pdf of each fitted distribution
line(x,pdfWei*area,'LineStyle','-','Color','r')
hold on
line(x,pdfNorm*area,'LineStyle','-.','Color','b')
line(x,pdfLog*area,'LineStyle','--','Color','g')
line(x,pdfKer*area,'LineStyle',':','Color','k')
legend('Data','Weibull','Normal','Logistic','Kernel','Location','Best')
title('MPG for Cars from USA')
xlabel('MPG')
hold off
```



Superimposing the pdf plots over a histogram of the sample data provides a visual comparison of how well each type of distribution fits the data. Only the nonparametric kernel distribution `KerUSA` comes close to revealing the two modes in the original data.

#### Step 6. Further group USA data by year.

To investigate the two modes revealed in Step 5, group the MPG data by both country of origin (`Origin`) and model year (`Model_Year`), and use `fitdist` to fit kernel distributions to each group.

```
[KerByYearOrig,Names] = fitdist(MPG,'Kernel','By',{Origin Model_Year});
```

Each unique combination of origin and model year now has a kernel distribution object associated with it.

**Step 7. Compute the pdf for each group.**

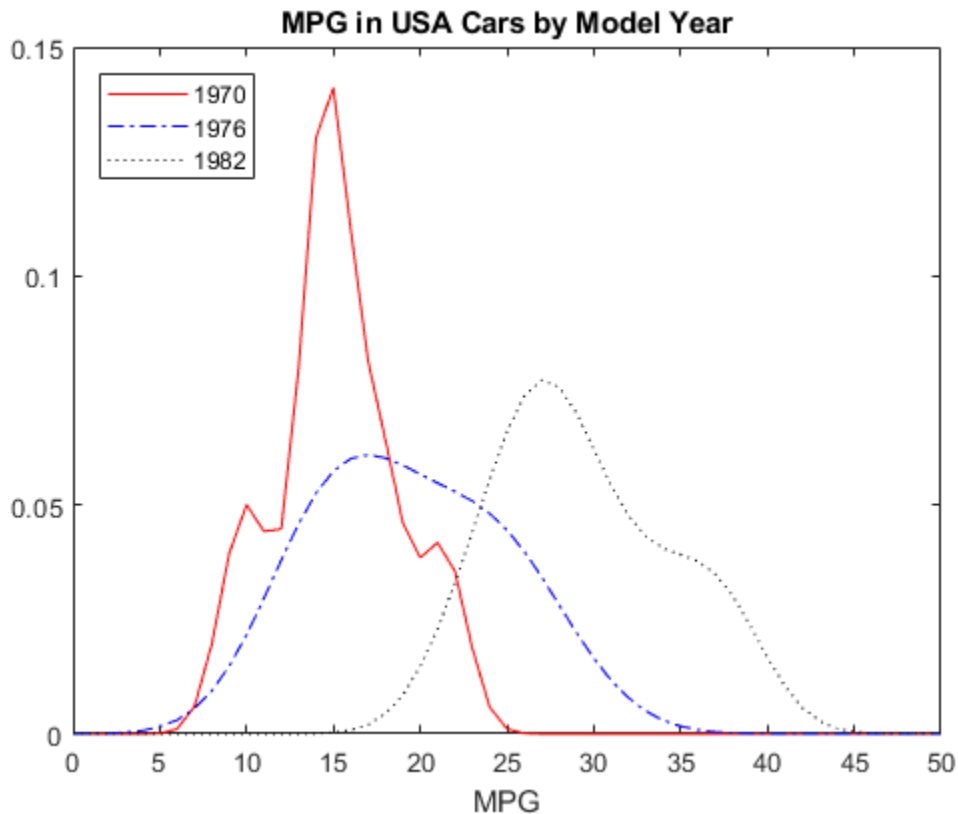
Extract the three probability distributions for each USA model year, which are in positions 12, 13, and 14 in the cell array `KerByYearOrig`. Compute each pdf.

```
USA70 = KerByYearOrig{12};  
USA76 = KerByYearOrig{13};  
USA82 = KerByYearOrig{14};  
  
pdf70 = pdf(USA70,x);  
pdf76 = pdf(USA76,x);  
pdf82 = pdf(USA82,x);
```

**Step 8. Plot pdf for each group.**

Plot the pdf for each group on the same figure.

```
figure  
plot(x,pdf70,'r-')  
hold on  
plot(x,pdf76,'b-.')  
plot(x,pdf82,'k:')  
legend({'1970','1976','1982'},'Location','NW')  
title('MPG in USA Cars by Model Year')  
xlabel('MPG')  
hold off
```



When further grouped by model year, the pdf plots reveal two distinct peaks in the MPG data for cars made in the USA — one for the model year 1970, and one for the model year 1982. This explains why the smooth curve produced by the kernel distribution for the combined USA miles per gallon data shows two peaks instead of one.

## See Also

`fitdist`

## Related Examples

- “Fit Probability Distribution Objects to Grouped Data” on page 5-121

## **More About**

- “Grouping Variables” on page 2-59



## Fit Probability Distribution Objects to Grouped Data

This example shows how to fit probability distribution objects to grouped sample data, and create a plot to visually compare the pdf of each group.

### Step 1. Load sample data.

Load the sample data.

```
load carsmall;
```

The data contains miles per gallon (MPG) measurements for different makes and models of cars, grouped by country of origin (`Origin`), model year (`Model_Year`), and other vehicle characteristics.

### Step 2. Create a nominal array.

Transform `Origin` into a nominal array and remove the Italian car from the sample data. Since there is only one Italian car, `fitdist` cannot fit a distribution to that group. Removing the Italian car from the sample data prevents `fitdist` from returning an error.

```
Origin = nominal(Origin);
MPG2 = MPG(Origin~='Italy');
Origin2 = Origin(Origin~='Italy');
Origin2 = droplevels(Origin2, 'Italy');
```

### Step 3. Fit kernel distributions to each group.

Use `fitdist` to fit kernel distributions to each country of origin group in the MPG data.

```
[KerByOrig, Country] = fitdist(MPG2, 'Kernel', 'by', Origin2)
```

```
KerByOrig =
```

```
1x5 cell array
```

```
Columns 1 through 2
```

```
    {1x1 prob.KernelDistribution}    {1x1 prob.KernelDistribution}
```

```
Columns 3 through 4
```

```
{1x1 prob.KernelDistribution}    {1x1 prob.KernelDistribution}

Column 5

{1x1 prob.KernelDistribution}

Country =

5x1 cell array

    {'France' }
    {'Germany'}
    {'Japan'   }
    {'Sweden' }
    {'USA'    }
```

The cell array `KerByOrig` contains five kernel distribution objects, one for each country represented in the sample data. Each object contains properties that hold information about the data, the distribution, and the parameters. The array `Country` lists the country of origin for each group in the same order as the distribution objects are stored in `KerByOrig`.

### Step 4. Compute the pdf for each group.

Extract the probability distribution objects for Germany, Japan, and USA. Use the positions of each country in `KerByOrig` shown in Step 3, which indicates that Germany is the second country, Japan is the third country, and USA is the fifth country. Compute the pdf for each group.

```
Germany = KerByOrig{2};
Japan = KerByOrig{3};
USA = KerByOrig{5};

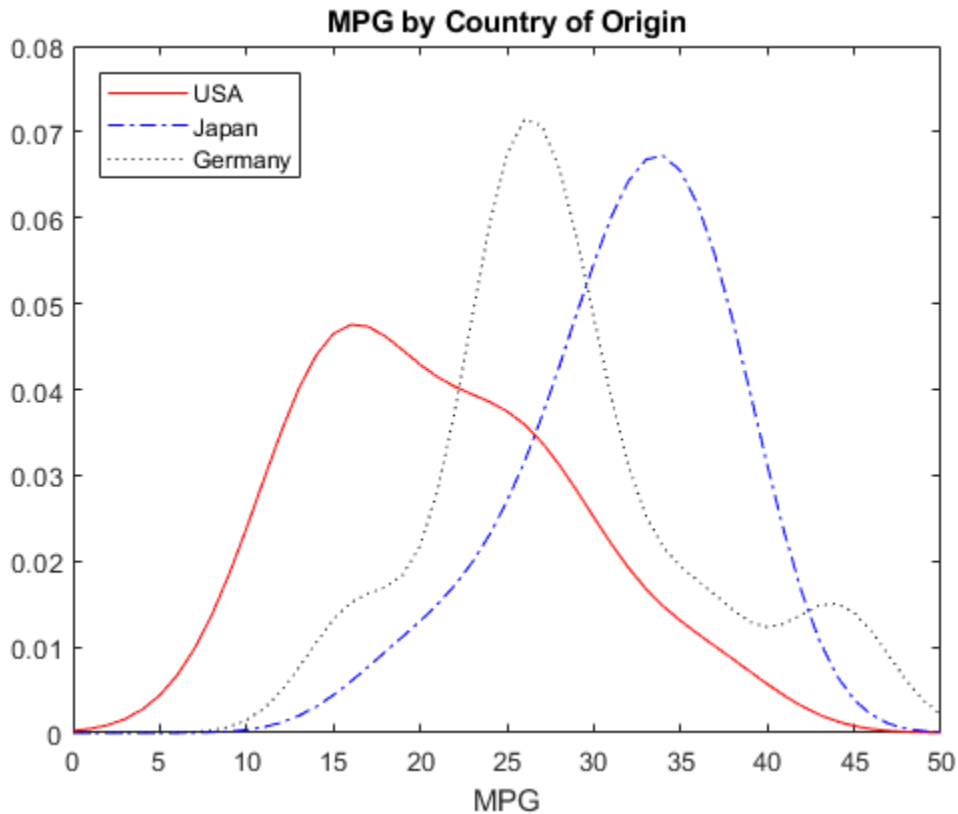
x = 0:1:50;

USA_pdf = pdf(USA,x);
Japan_pdf = pdf(Japan,x);
Germany_pdf = pdf(Germany,x);
```

**Step 5. Plot the pdf for each group.**

Plot the pdf for each group on the same figure.

```
plot(x,USA_pdf,'r-')
hold on
plot(x,Japan_pdf,'b-.')
plot(x,Germany_pdf,'k:')
legend({'USA','Japan','Germany'},'Location','NW')
title('MPG by Country of Origin')
xlabel('MPG')
```



The resulting plot shows how miles per gallon (MPG) performance differs by country of origin (Origin). Using this data, the USA has the widest distribution, and its peak is at

the lowest MPG value of the three origins. Japan has the most regular distribution with a slightly heavier left tail, and its peak is at the highest MPG value of the three origins. The peak for Germany is between the USA and Japan, and the second bump near 44 miles per gallon suggests that there might be multiple modes in the data.

### See Also

`fitdist` | pdf

### Related Examples

- “Fit Distributions to Grouped Data Using `ksdensity`” on page 5-54

### More About

- “Kernel Distribution” on page B-86
- “Grouping Variables” on page 2-59

## Multinomial Probability Distribution Objects

This example shows how to generate random numbers, compute and plot the pdf, and compute descriptive statistics of a multinomial distribution using probability distribution objects.

### Step 1. Define the distribution parameters.

Create a vector `p` containing the probability of each outcome. Outcome 1 has a probability of 1/2, outcome 2 has a probability of 1/3, and outcome 3 has a probability of 1/6. The number of trials `n` in each experiment is 5, and the number of repetitions `reps` of the experiment is 8.

```
p = [1/2 1/3 1/6];  
n = 5;  
reps = 8;
```

### Step 2. Create a multinomial probability distribution object.

Create a multinomial probability distribution object using the specified value `p` for the `Probabilities` parameter.

```
pd = makedist('Multinomial','Probabilities',p)
```

```
pd =  
  
MultinomialDistribution  
  
Probabilities:  
0.5000    0.3333    0.1667
```

### Step 3. Generate one random number.

Generate one random number from the multinomial distribution, which is the outcome of a single trial.

```
rng('default') % For reproducibility  
r = random(pd)
```

```
r =
```

2

This trial resulted in outcome 2.

#### Step 4. Generate a matrix of random numbers.

You can also generate a matrix of random numbers from the multinomial distribution, which reports the results of multiple experiments that each contain multiple trials. Generate a matrix that contains the outcomes of an experiment with  $n = 5$  trials and  $\text{reps} = 8$  repetitions.

```
r = random(pd, reps, n)
```

```
r =
```

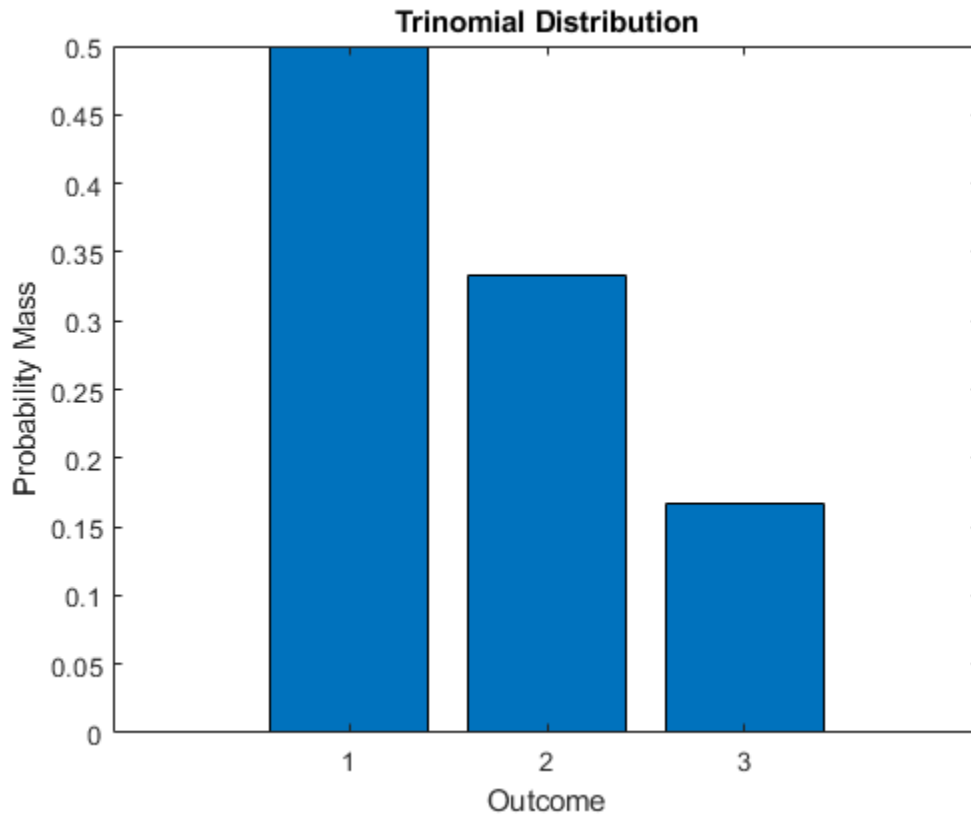
```
 3     3     3     2     1
 1     1     2     2     1
 3     3     3     1     2
 2     3     2     2     2
 1     1     1     1     1
 1     2     3     2     3
 2     1     3     1     1
 3     1     2     1     1
```

Each element in the resulting matrix is the outcome of one trial. The columns correspond to the five trials in each experiment, and the rows correspond to the eight experiments. For example, in the first experiment (corresponding to the first row), one of the five trials resulted in outcome 1, one of the five trials resulted in outcome 2, and three of the five trials resulted in outcome 3.

#### Step 5. Compute and plot the pdf.

Compute the pdf of the distribution.

```
x = 1:3;
y = pdf(pd, x);
bar(x, y)
xlabel('Outcome')
ylabel('Probability Mass')
title('Trinomial Distribution')
```



The plot shows the probability mass for each  $k$  possible outcome. For this distribution, the pdf value for any  $x$  other than 1, 2, or 3 is 0.

**Step 6. Compute descriptive statistics.**

Compute the mean, median, and standard deviation of the distribution.

```
m = mean(pd)
```

```
m =
```

```
1.6667
```

```
med = median(pd)
```

```
med =
```

```
1
```

```
s = std(pd)
```

```
s =
```

```
0.7454
```



## Multinomial Probability Distribution Functions

This example shows how to generate random numbers and compute and plot the pdf of a multinomial distribution using probability distribution functions.

### Step 1. Define the distribution parameters.

Create a vector `p` containing the probability of each outcome. Outcome 1 has a probability of 1/2, outcome 2 has a probability of 1/3, and outcome 3 has a probability of 1/6. The number of trials in each experiment `n` is 5, and the number of repetitions of the experiment `reps` is 8.

```
p = [1/2 1/3 1/6];
n = 5;
reps = 8;
```

### Step 2. Generate one random number.

Generate one random number from the multinomial distribution, which is the outcome of a single trial.

```
rng('default') % For reproducibility
r = mnrnd(1,p,1)
```

```
r =
     0     1     0
```

The returned vector `r` contains three elements, which show the counts for each possible outcome. This single trial resulted in outcome 2.

### Step 3. Generate a matrix of random numbers.

You can also generate a matrix of random numbers from the multinomial distribution, which reports the results of multiple experiments that each contain multiple trials. Generate a matrix that contains the outcomes of an experiment with `n = 5` trials and `reps = 8` repetitions.

```
r = mnrnd(n,p,reps)
```

```
r =
```

```
1     1     3
3     2     0
1     1     3
0     4     1
5     0     0
1     2     2
3     1     1
3     1     1
```

Each row in the resulting matrix contains counts for each of the  $k$  multinomial bins. For example, in the first experiment (corresponding to the first row), one of the five trials resulted in outcome 1, one of the five trials resulted in outcome 2, and three of the five trials resulted in outcome 3.

#### Step 4. Compute the pdf.

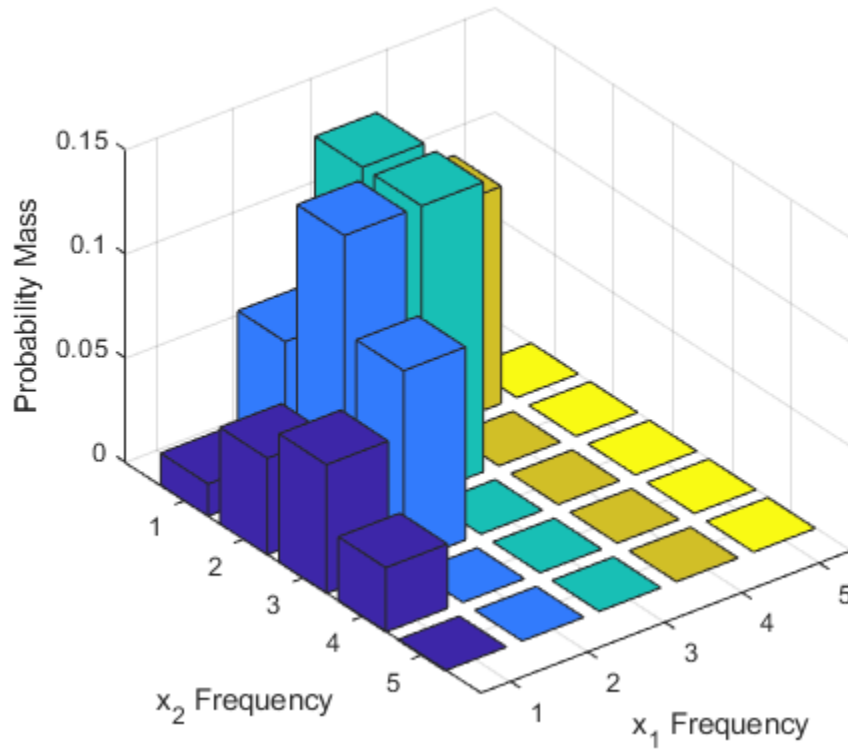
Since multinomial functions work with bin counts, create a multidimensional array of all possible outcome combinations, and compute the pdf using `mnpdf`.

```
count1 = 1:n;
count2 = 1:n;
[x1,x2] = meshgrid(count1,count2);
x3 = n-(x1+x2);
y = mnpdf([x1(:),x2(:),x3(:)], repmat(p, (n)^2, 1));
```

#### Step 5. Plot the pdf.

Create a 3-D bar graph to visualize the pdf for each combination of outcome frequencies.

```
y = reshape(y,n,n);
bar3(y)
set(gca, 'XTickLabel', 1:n);
set(gca, 'YTickLabel', 1:n);
xlabel('x_1 Frequency')
ylabel('x_2 Frequency')
zlabel('Probability Mass')
```



The plot shows the probability mass for each possible combination of outcomes. It does not show  $x_3$ , which is determined by the constraint  $x_1 + x_2 + x_3 = n$ .

## Generate Random Numbers Using Uniform Distribution Inversion

This example shows how to generate random numbers using the uniform distribution inversion method. This is useful for distributions when it is possible to compute the inverse cumulative distribution function, but there is no support for sampling from the distribution directly.

### Step 1. Generate random numbers from the standard uniform distribution.

Use `rand` to generate 1000 random numbers from the uniform distribution on the interval (0,1).

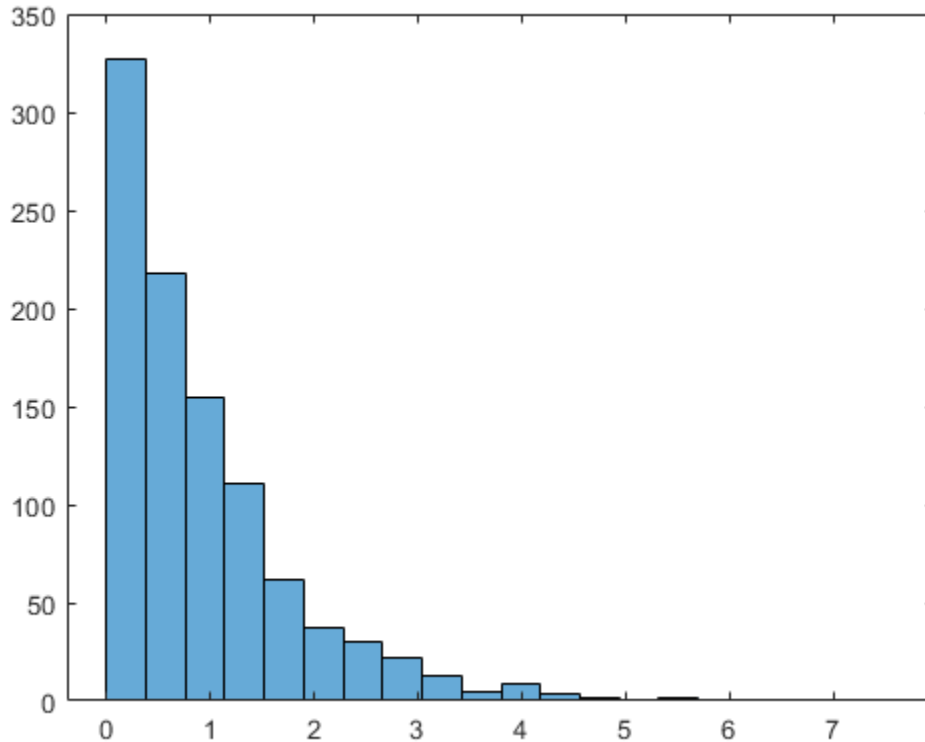
```
rng('default') % For reproducibility
u = rand(1000,1);
```

The inversion method relies on the principle that continuous cumulative distribution functions (cdfs) range uniformly over the open interval (0,1). If  $u$  is a uniform random number on (0,1), then  $x = F^{-1}(u)$  generates a random number  $x$  from any continuous distribution with the specified cdf  $F$ .

### Step 2. Generate random numbers from the Weibull distribution.

Use the inverse cumulative distribution function to generate the random numbers from a Weibull distribution with parameters  $A = 1$  and  $B = 1$  that correspond to the probabilities in `u`. Plot the results.

```
x = wblinv(u,1,1);
histogram(x,20);
```

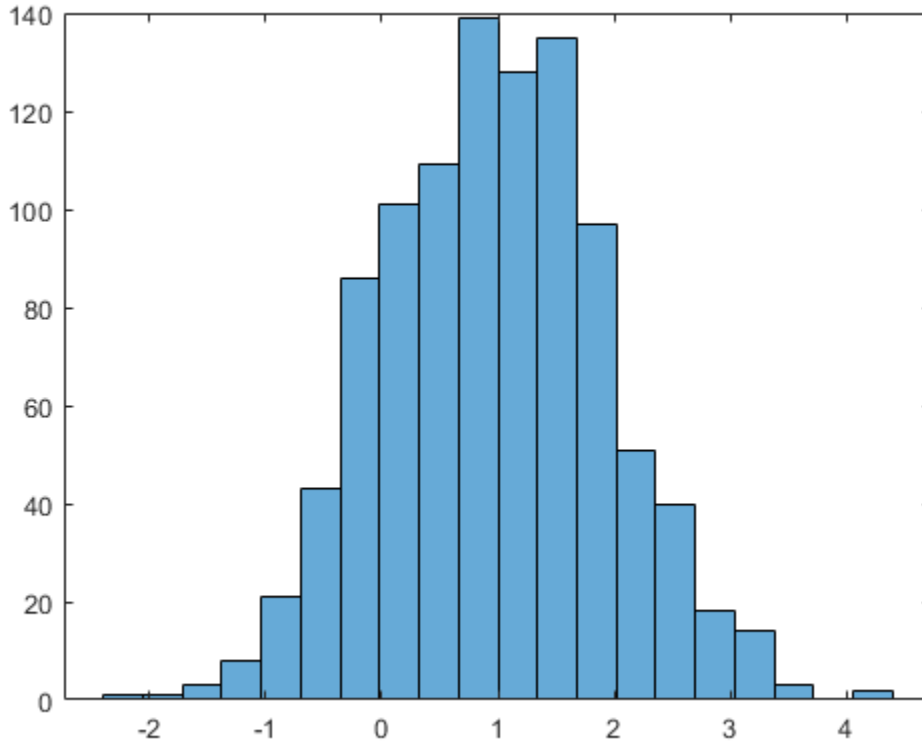


The histogram shows that the random numbers generated using the Weibull inverse cdf function `wblinv` have a Weibull distribution.

### Step 3. Generate random numbers from the standard normal distribution.

The same values in `u` can generate random numbers from any distribution, for example the standard normal, by following the same procedure using the inverse cdf of the desired distribution.

```
figure
x_norm = norminv(u,1,1);
histogram(x_norm,20)
```



The histogram shows that, by using the standard normal inverse cdf `norminv`, the random numbers generated from `u` now have a standard normal distribution.

### See Also

`hist` | `norminv` | `rand` | `wblinv`

### More About

- “Uniform Distribution (Continuous)” on page B-184
- “Weibull Distribution” on page B-193

- “Normal Distribution” on page B-134

## Represent Cauchy Distribution Using t Location-Scale

This example shows how to use the t location-scale probability distribution object to work with a Cauchy distribution with nonstandard parameter values.

### Step 1. Create a probability distribution object.

Create a `t` location-scale probability distribution object with degrees of freedom `nu = 1`. Specify `mu = 3` to set the location parameter equal to 3, and `sigma = 1` to set the scale parameter equal to 1.

```
pd = makedist('tLocationScale', 'mu', 3, 'sigma', 1, 'nu', 1)
```

```
pd =
```

```
tLocationScaleDistribution  
  
t Location-Scale distribution  
  mu = 3  
  sigma = 1  
  nu = 1
```

### Step 2. Compute descriptive statistics.

Use object functions to compute descriptive statistics for the Cauchy distribution.

```
med = median(pd)
```

```
med =
```

```
3
```

```
r = iqr(pd)
```

```
r =
```

```
2
```

```
m = mean(pd)
```



```
m =  
    NaN
```

```
s = std(pd)
```

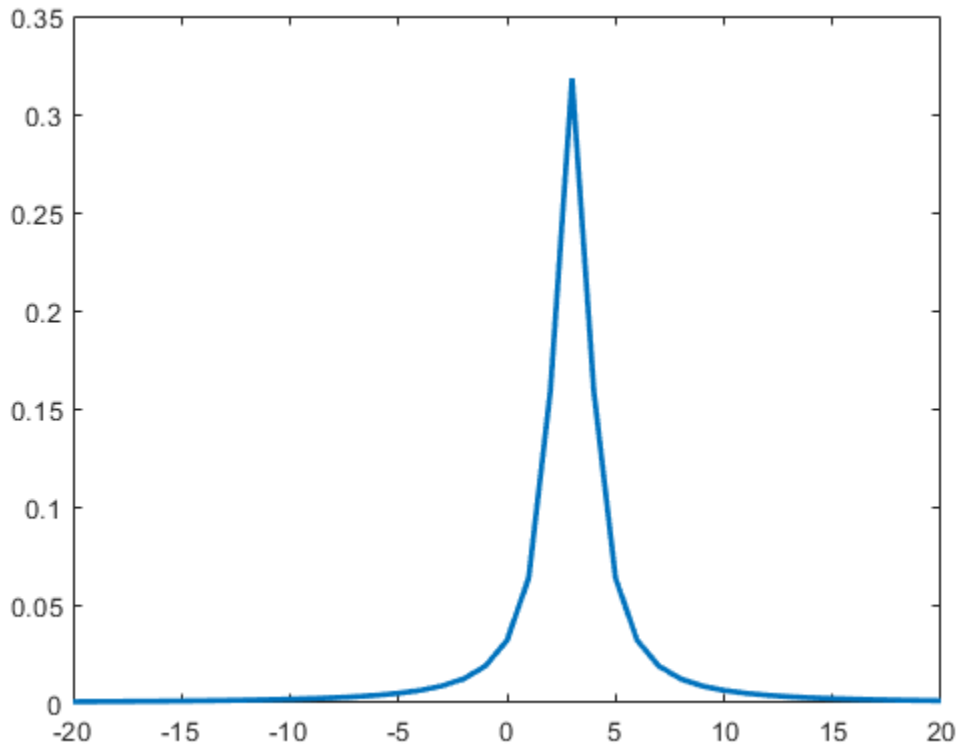
```
s =  
    Inf
```

The median of the Cauchy distribution is equal to its location parameter, and the interquartile range is equal to two times its scale parameter. Its mean and standard deviation are undefined.

### **Step 3. Compute and plot the pdf.**

Compute and plot the pdf of the Cauchy distribution.

```
x = -20:1:20;  
y = pdf(pd,x);  
plot(x,y,'LineWidth',2)
```



The peak of the pdf is centered at the location parameter  $\mu = 3$ .

**Step 4. Generate a vector of Cauchy random numbers.**

Generate a column vector containing 10 random numbers from the Cauchy distribution using the `random` function for the `t` location-scale probability distribution object.

```
rng('default'); % For reproducibility  
r = random(pd,10,1)
```

```
r =
```

```
3.2678
```

```
4.6547
2.0604
4.7322
3.1810
1.6649
1.8471
4.2466
5.4647
8.8874
```

### Step 5. Generate a matrix of Cauchy random numbers.

Generate a 5-by-5 matrix of Cauchy random numbers.

```
r = random(pd, 5, 5)
```

```
r =
```

```
2.2867    2.9692   -1.7003    5.5949    1.9806
2.7421    2.7180    3.2210    2.4233    3.1394
3.5966    3.9806    1.0182    6.4180    5.1367
5.4791   15.6472    0.7558    2.8908    5.9031
1.6863    4.0985    2.9934   13.9506    4.8792
```

## See Also

`makedist`

## Related Examples

- “Generate Cauchy Random Numbers Using Student's *t*” on page 5-140

## More About

- “*t* Location-Scale Distribution” on page B-175

## Generate Cauchy Random Numbers Using Student's t

This example shows how to use the Student's t distribution to generate random numbers from a standard Cauchy distribution.

### Step 1. Generate a vector of random numbers.

Generate a column vector containing 10 random numbers from a standard Cauchy distribution, which has a location parameter  $\mu = 0$  and scale parameter  $\sigma = 1$ . Use `trnd` with degrees of freedom  $\nu = 1$ .

```
rng('default'); % For reproducibility
r = trnd(1,10,1)
```

r =

```
    0.2678
    1.6547
   -0.9396
    1.7322
    0.1810
   -1.3351
   -1.1529
    1.2466
    2.4647
    5.8874
```

### Step 2. Generate a matrix of random numbers.

Generate a 5-by-5 matrix of random numbers from a standard Cauchy distribution.

```
r = trnd(1,5,5)
```

r =

```
   -0.7133   -0.0308   -4.7003    2.5949   -1.0194
   -0.2579   -0.2820    0.2210   -0.5767    0.1394
    0.5966    0.9806   -1.9818    3.4180    2.1367
    2.4791   12.6472   -2.2442   -0.1092    2.9031
```

-1.3137    1.0985    -0.0066    10.9506    1.8792

## See Also

trnd

## Related Examples

- “Represent Cauchy Distribution Using t Location-Scale” on page 5-136

## More About

- “Student's t Distribution” on page B-166

## Generate Correlated Data Using Rank Correlation

This example shows how to use a copula and rank correlation to generate correlated data from probability distributions that do not have an inverse cdf function available, such as the Pearson flexible distribution family.

### Step 1. Generate Pearson random numbers.

Generate 1000 random numbers from two different Pearson distributions, using the `pearsrnd` function. The first distribution has the parameter values  $\mu$  equal to 0,  $\sigma$  equal to 1, skew equal to 1, and kurtosis equal to 4. The second distribution has the parameter values  $\mu$  equal to 0,  $\sigma$  equal to 1, skew equal to 0.75, and kurtosis equal to 3.

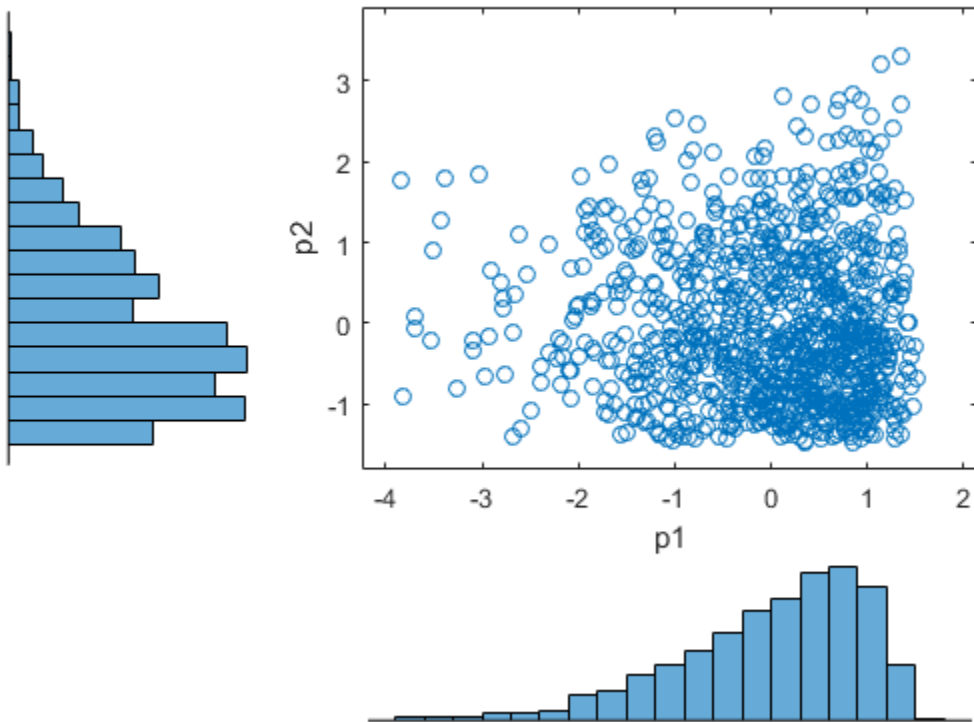
```
rng default % For reproducibility
p1 = pearsrnd(0,1,-1,4,1000,1);
p2 = pearsrnd(0,1,0.75,3,1000,1);
```

At this stage, `p1` and `p2` are independent samples from their respective Pearson distributions, and are uncorrelated.

### Step 2. Plot the Pearson random numbers.

Create a `scatterhist` plot to visualize the Pearson random numbers.

```
figure
scatterhist(p1,p2)
```

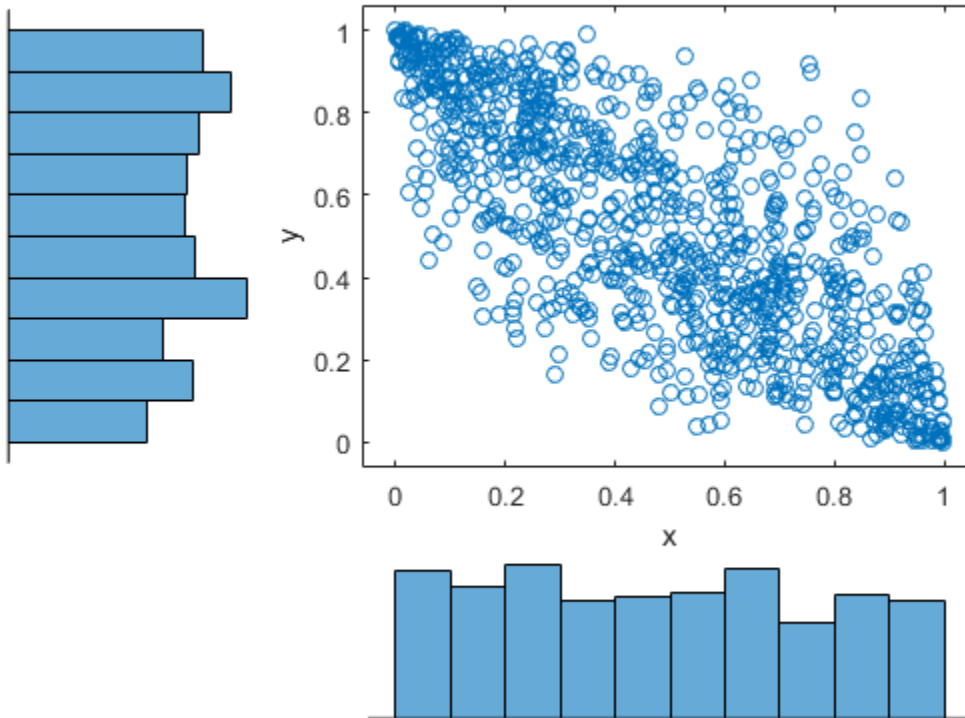


The histograms show the marginal distributions for  $p1$  and  $p2$ . The scatterplot shows the joint distribution for  $p1$  and  $p2$ . The lack of pattern to the scatterplot shows that  $p1$  and  $p2$  are independent.

### Step 3. Generate random numbers using a Gaussian copula.

Use `copularnd` to generate 1000 correlated random numbers with a correlation coefficient equal to  $-0.8$ , using a Gaussian copula. Create a scatterhist plot to visualize the random numbers generated from the copula.

```
u = copularnd('Gaussian',-0.8,1000);
figure
scatterhist(u(:,1),u(:,2))
```



The histograms show that the data in each column of the copula have a marginal uniform distribution. The scatterplot shows that the data in the two columns are negatively correlated.

#### Step 4. Sort the copula random numbers.

Using Spearman's rank correlation, transform the two independent Pearson samples into correlated data.

Use the `sort` function to sort the copula random numbers from smallest to largest, and to return a vector of indices describing the rearranged order of the numbers.

```
[s1,i1] = sort(u(:,1));
[s2,i2] = sort(u(:,2));
```



`s1` and `s2` contain the numbers from the first and second columns of the copula, `u`, sorted in order from smallest to largest. `i1` and `i2` are index vectors that describe the rearranged order of the elements into `s1` and `s2`. For example, if the first value in the sorted vector `s1` is the third value in the original unsorted vector, then the first value in the index vector `i1` is 3.

**Step 5. Transform the Pearson samples using Spearman's rank correlation.**

Create two vectors of zeros, `x1` and `x2`, that are the same size as the sorted copula vectors, `s1` and `s2`. Sort the values in `p1` and `p2` from smallest to largest. Place the values into `x1` and `x2`, in the same order as the indices `i1` and `i2` generated by sorting the copula random numbers.

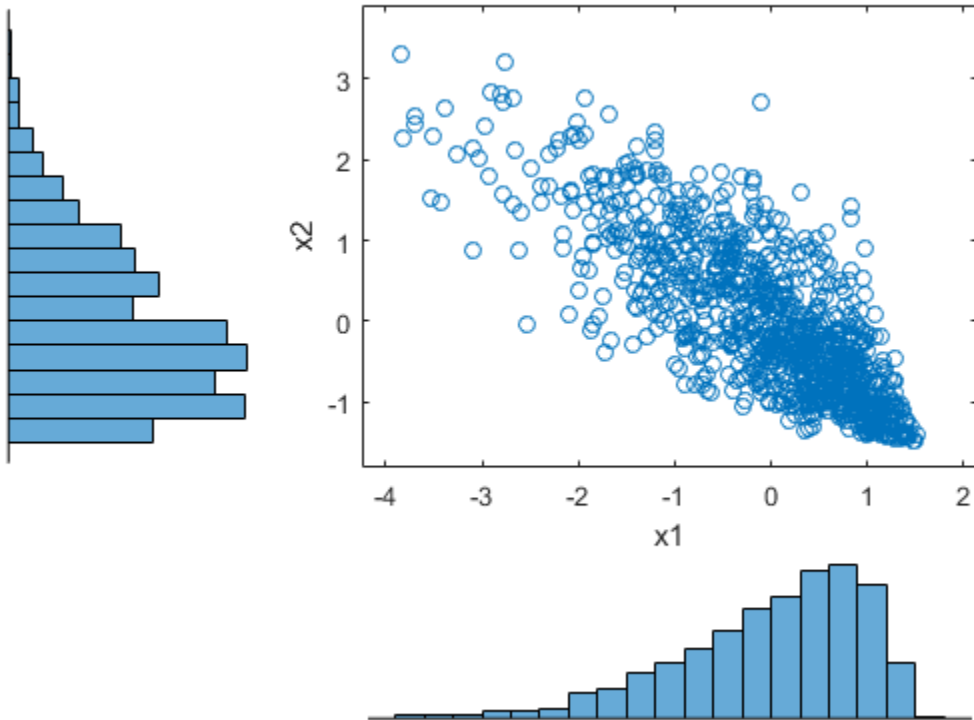
```
x1 = zeros(size(s1));
x2 = zeros(size(s2));

x1(i1) = sort(p1);
x2(i2) = sort(p2);
```

**Step 6. Plot the correlated Pearson random numbers.**

Create a `scatterhist` plot to visualize the correlated Pearson data.

```
figure
scatterhist(x1,x2)
```



The histograms show the marginal Pearson distributions for each column of data. The scatterplot shows the joint distribution of  $p_1$  and  $p_2$ , and indicates that the data are now negatively correlated.

### Step 7. Confirm Spearman rank correlation coefficient values.

Confirm that the Spearman rank correlation coefficient is the same for the copula random numbers and the correlated Pearson random numbers.

```
copula_corr = corr(u, 'Type', 'spearman')
pearson_corr = corr([x1,x2], 'Type', 'spearman')
```

```
copula_corr =
```

```
1.0000 -0.7858
-0.7858 1.0000
```

```
pearson_corr =
```

```
1.0000 -0.7858
-0.7858 1.0000
```

The Spearman rank correlation is the same for the copula and the Pearson random numbers.

## See Also

`copularnd` | `corr` | `sort`

## More About

- “Copulas: Generate Correlated Samples” on page 5-164

## Gaussian Mixture Models

Gaussian mixture models (GMM) are composed of  $k$  multivariate normal density components, where  $k$  is a positive integer. Each component has a  $d$ -dimensional mean ( $d$  is a positive integer),  $d$ -by- $d$  covariance matrix, and a mixing proportion. Mixing proportion  $j$  determines the proportion of the population composed by component  $j$ ,  $j = 1, \dots, k$ .

You can fit a GMM using the Statistics and Machine Learning Toolbox function `fitgmdist` by specifying  $k$  and by supplying  $X$ , an  $n$ -by- $d$  matrix of data. The columns of  $X$  correspond to predictors, features, or attributes, and the rows correspond to observations or examples. By default, `fitgmdist` fits full covariance matrices that are different among components (or unshared).

`fitgmdist` fits GMMs to data using the iterative Expectation-Maximization (EM) algorithm. Using initial values for component means, covariance matrices, and mixing proportions, the EM algorithm proceeds using these steps.

- 1 For each observation, the algorithm computes posterior probabilities of component memberships. You can think of the result as an  $n$ -by- $k$  matrix, where element  $(i,j)$  contains the posterior probability that observation  $i$  is from component  $j$ . This is the E-step of the EM algorithm.
- 2 Using the component-membership posterior probabilities as weights, the algorithm estimates the component means, covariance matrices, and mixing proportions by applying maximum likelihood. This is the M-step of the EM algorithm.

The algorithm iterates over these steps until convergence. The likelihood surface is complex, and the algorithm might converge to a local optimum. Also, the resulting local optimum might depend on the initial conditions. `fitgmdist` has several options for choosing initial conditions, including random component assignments for the observations and the  $k$ -means ++ algorithm.

`fitgmdist` returns a fitted `gmdistribution` model object. The object contains properties that store the estimation results, which include the estimated parameters, convergence information, and information criteria (Akaike and Bayesian information criteria). You can use dot notation to access the properties.

Once you have a fitted GMM, you can cluster query data using it. Clustering using GMM is sometimes considered a soft clustering method. The posterior probabilities for each point indicate that each data point has some probability of belonging to each cluster. For

more information on clustering with GMM, see “Clustering Using Gaussian Mixture Models” on page 16-28.

## See Also

`cluster` | `fitgmdist` | `gmdistribution`

## Related Examples

- “Create a Gaussian Mixture Model” on page 5-150
- “Fit a Gaussian Mixture Model to Data” on page 5-155
- “Simulate Data from a Gaussian Mixture Model” on page 5-160

## More About

- “Clustering Using Gaussian Mixture Models” on page 16-28

## Create a Gaussian Mixture Model

This example shows how to create a known, or fully specified, Gaussian mixture model (GMM) object using `gmdistribution` and by specifying component means, covariances, and mixture proportions. To create a GMM object by fitting data to a GMM, see “Fit a Gaussian Mixture Model to Data” on page 5-155.

Specify the component means, covariances, and mixing proportions for a two-component mixture of bivariate Gaussian distributions.

```
Mu = [1 2;-3 -5]; % Means
Sigma = cat(3,[2 0;0 .5],[1 0;0 1]); % Covariances
P = ones(1,2)/2; % Mixing proportions
```

The rows of `Mu` correspond to the component mean vectors, and the pages of `Sigma` correspond to the component covariance matrices.

Create the GMM object using `gmdistribution`.

```
gm = gmdistribution(Mu,Sigma,P);
```

Display properties of the GMM.

```
properties = properties(gm)
```

```
properties =
    15x1 cell array
    {'NumVariables'      }
    {'DistributionName'  }
    {'NumComponents'    }
    {'ComponentProportion' }
    {'SharedCovariance' }
    {'NumIterations'    }
    {'RegularizationValue' }
    {'NegativeLogLikelihood'}
    {'CovarianceType'   }
    {'mu'               }
    {'Sigma'            }
    {'AIC'              }
    {'BIC'              }
    {'Converged'       }
```

```
{'ProbabilityTolerance' }
```

For a description of the properties, see `gmdistribution`. To access the value of a property, use dot notation. For example, access the dimensions of the GMM.

```
dimension = gm.NDimensions
```

```
dimension =
```

```
2
```

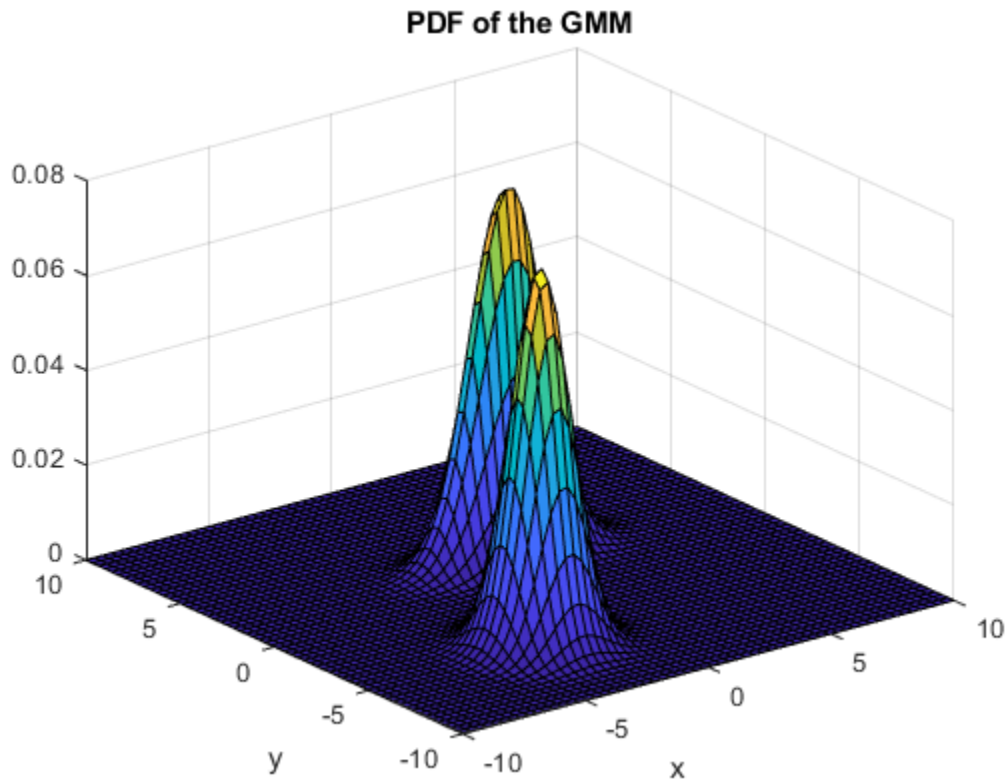
Visualize the pdf of the GMM using `gmdistribution.pdf` and the MATLAB® function `ezsurf`.

```
gmPDF = @(x,y)pdf(gm,[x y]);
```

```
figure;
```

```
ezsurf(gmPDF,[-10 10],[-10 10])
```

```
title('PDF of the GMM');
```

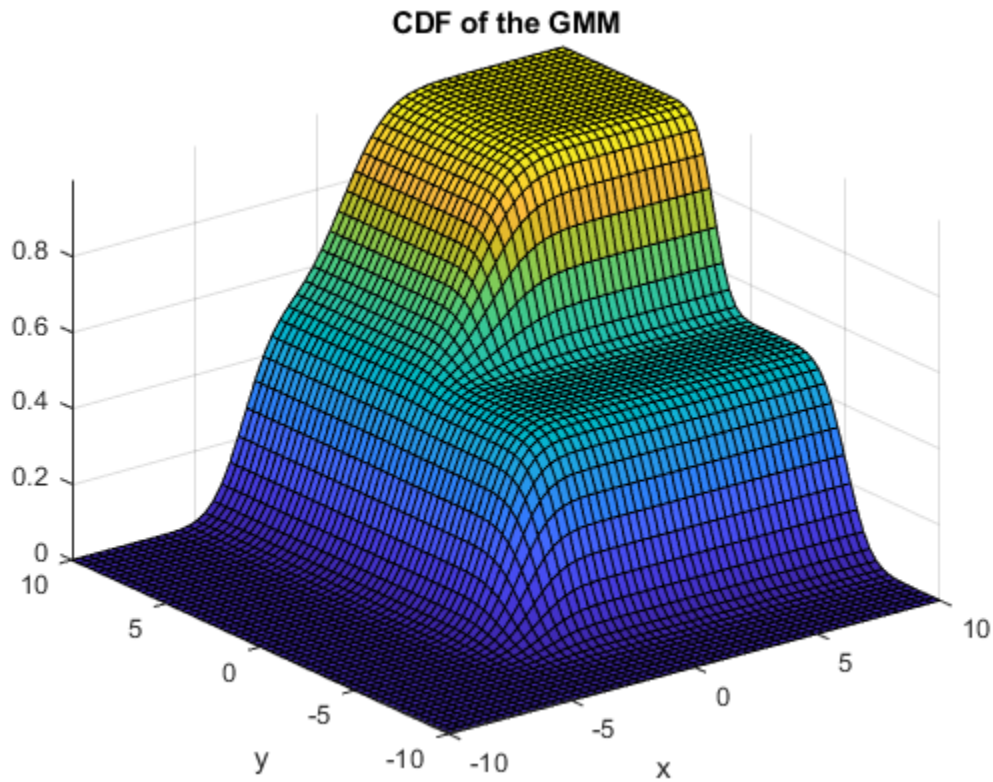


Visualize the cdf of the GMM using `gmdistribution.cdf` and `ezsurf`.

```
gmCDF = @(x,y)cdf(gm,[x y]);
```

```
figure  
ezsurf(@(x,y)cdf(gm,[x y]),[-10 10],[-10 10])  
title('CDF of the GMM');
```





## See Also

`fitgmdist` | `gmdistribution`

## Related Examples

- “Fit a Gaussian Mixture Model to Data” on page 5-155
- “Simulate Data from a Gaussian Mixture Model” on page 5-160

## **More About**

- “Gaussian Mixture Models” on page 5-148
- “Clustering Using Gaussian Mixture Models” on page 16-28

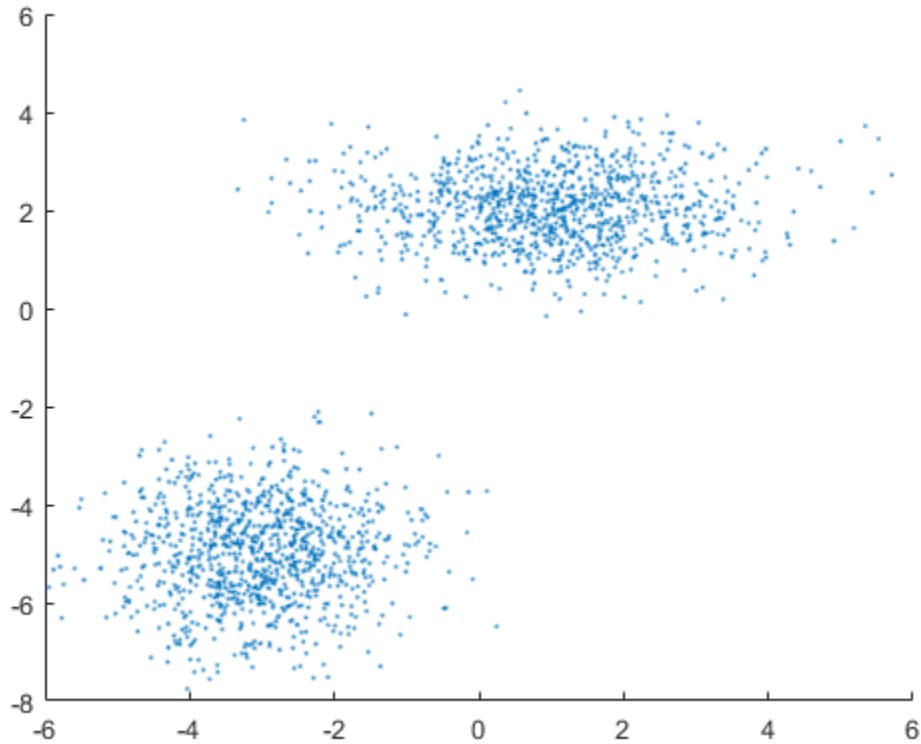
## Fit a Gaussian Mixture Model to Data

This example shows how to simulate data from a multivariate normal distribution, and then fit a Gaussian mixture model (GMM) to the data using `fitgmdist`. To create a known, or fully specified, GMM object, see “Create a Gaussian Mixture Model” on page 5-150.

`fitgmdist` requires a matrix of data ( $X$ ) and the number of components in the GMM ( $k$ ). To create a useful GMM, you must choose  $k$  carefully. Too few components fails to model the data accurately (i.e., underfitting to the data). Too many components leads to an over-fit model with singular covariance matrices.

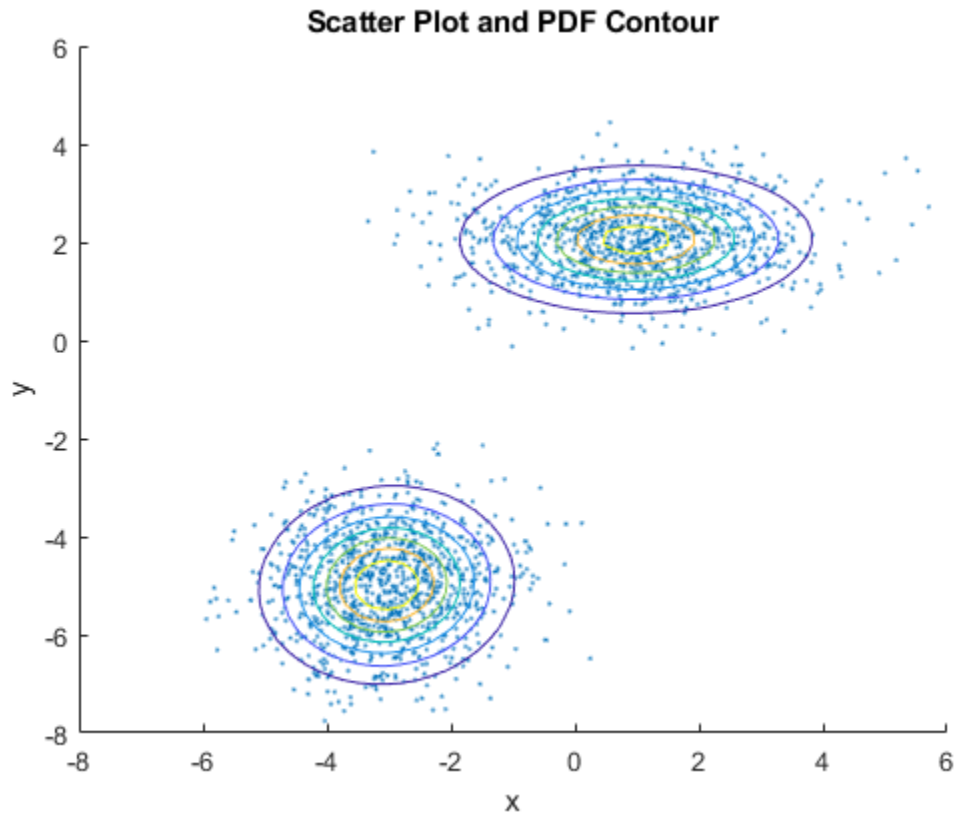
Simulate data from a mixture of two bivariate Gaussian distributions using `mvnrnd`.

```
MU1 = [1 2];  
SIGMA1 = [2 0; 0 .5];  
MU2 = [-3 -5];  
SIGMA2 = [1 0; 0 1];  
  
rng(1); % For reproducibility  
X = [mvnrnd(MU1, SIGMA1, 1000);  
     mvnrnd(MU2, SIGMA2, 1000)];  
  
figure;  
scatter(X(:,1), X(:,2), 10, 'r')
```



Fit a two-component GMM. Plot the pdf of the fitted GMM.

```
options = statset('Display','final');  
gm = fitgmdist(X,2,'Options',options);  
gmPDF = @(x,y)pdf(gm,[x y]);  
  
hold on  
h = ezcontour(gmPDF,[-8 6],[-8 6]);  
title('Scatter Plot and PDF Contour')  
hold off  
  
5 iterations, log-likelihood = -7105.71
```



Display the estimates for  $\mu$ ,  $\sigma$ , and mixture proportions

```
ComponentMeans = gm.mu  
ComponentCovariances = gm.Sigma  
MixtureProportions = gm.PComponents
```

```
ComponentMeans =
```

```
    -3.0377    -4.9859  
     0.9812     2.0563
```

```
ComponentCovariances(:, :, 1) =
```

```
1.0132    0.0482
0.0482    0.9796
```

```
ComponentCovariances(:,:,2) =
```

```
1.9919    0.0127
0.0127    0.5533
```

```
MixtureProportions =
```

```
0.5000    0.5000
```

Fit four models to the data, each with an increasing number of components.

```
AIC = zeros(1,4);
gm = cell(1,4);
for k = 1:4
    gm{k} = fitgmdist(X,k);
    AIC(k) = gm{k}.AIC;
end
```

Display the number of components that minimizes the AIC.

```
[minAIC,numComponents] = min(AIC);
numComponents
```

```
numComponents =
```

```
2
```

The two-component model minimizes the AIC.

Display the two-component GMM.

```
gm2 = gm{numComponents}
```

```
gm2 =
```

```
Gaussian mixture distribution with 2 components in 2 dimensions
Component 1:
Mixing proportion: 0.500000
Mean:    -3.0377    -4.9859

Component 2:
Mixing proportion: 0.500000
Mean:     0.9812     2.0563
```

Both the Akaike and Bayes information are negative log-likelihoods for the data with penalty terms for the number of estimated parameters. You can use them to determine an appropriate number of components for a model when the number of components is unspecified.

## See Also

`fitgmdist` | `gmdistribution` | `mvnrnd` | `random`

## Related Examples

- “Create a Gaussian Mixture Model” on page 5-150
- “Simulate Data from a Gaussian Mixture Model” on page 5-160

## More About

- “Gaussian Mixture Models” on page 5-148
- “Clustering Using Gaussian Mixture Models” on page 16-28

## Simulate Data from a Gaussian Mixture Model

This example shows how to simulate data from a Gaussian mixture model (GMM) using a fully specified `gmdistribution` object and `gmdistribution.random`.

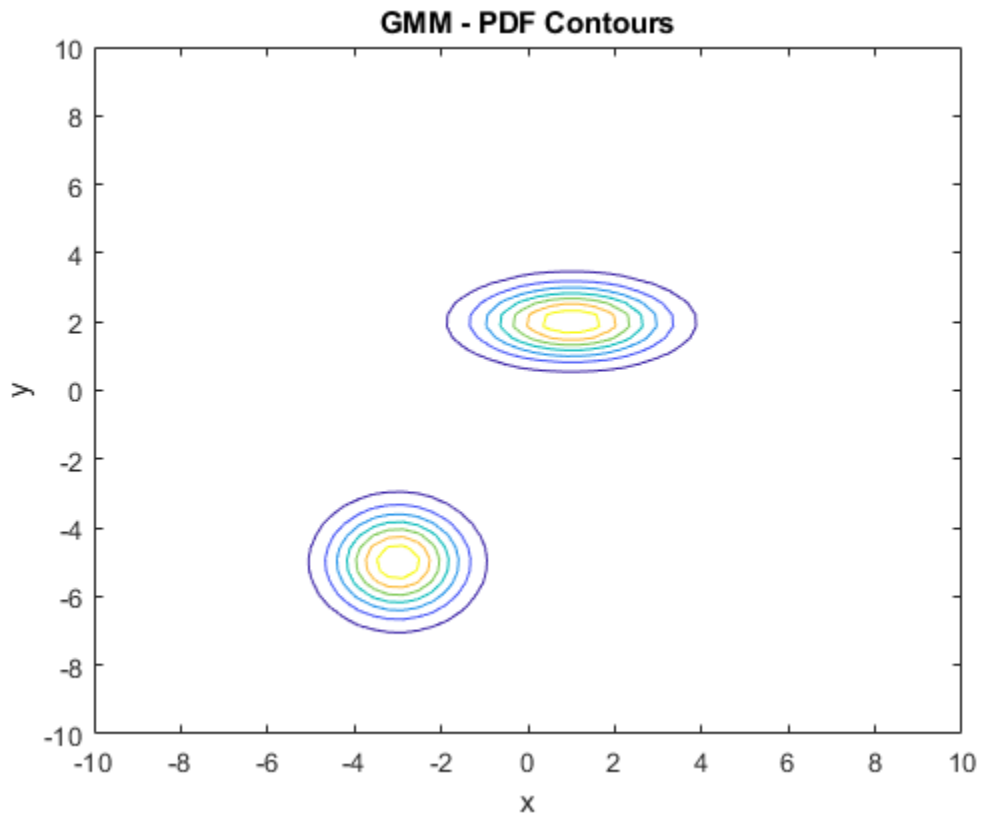
Create a known, two-component GMM object.

```
Mu = [1 2; -3 -5];  
Sigma = cat(3, [2 0; 0 .5], [1 0; 0 1]);  
P = ones(1,2)/2;  
gm = gmdistribution(Mu, Sigma, P);
```

Plot the contour of the pdf of the GMM.

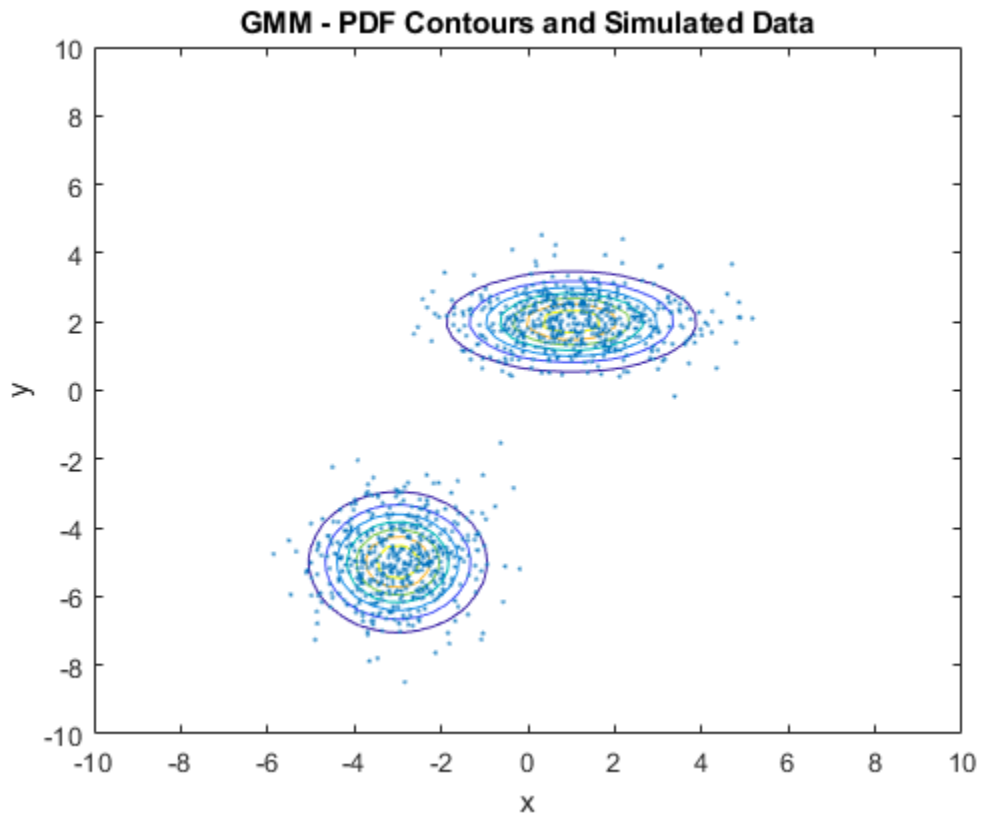
```
gmPDF = @(x,y)pdf(gm, [x y]);  
  
figure;  
ezcontour(gmPDF, [-10 10], [-10 10]);  
hold on  
title('GMM - PDF Contours');
```





Generate 1000 random variates from the GMM. Plot the variates with the pdf contours.

```
X = random(gm,1000);  
scatter(X(:,1),X(:,2),10, '.')  
title('GMM - PDF Contours and Simulated Data');
```



## See Also

`fitgmdist` | `gmdistribution` | `mvnrnd` | `random`

## Related Examples

- “Create a Gaussian Mixture Model” on page 5-150
- “Fit a Gaussian Mixture Model to Data” on page 5-155

## More About

- “Gaussian Mixture Models” on page 5-148
- “Clustering Using Gaussian Mixture Models” on page 16-28

## Copulas: Generate Correlated Samples

In this section...
“Determining Dependence Between Simulation Inputs” on page 5-164
“Constructing Dependent Bivariate Distributions” on page 5-168
“Using Rank Correlation Coefficients” on page 5-173
“Using Bivariate Copulas” on page 5-176
“Higher Dimension Copulas” on page 5-185
“Archimedean Copulas” on page 5-187
“Simulating Dependent Multivariate Data Using Copulas” on page 5-189
“Fitting Copulas to Data” on page 5-194

Copulas are functions that describe dependencies among variables, and provide a way to create distributions that model correlated multivariate data. Using a copula, you can construct a multivariate distribution by specifying marginal univariate distributions, and then choose a copula to provide a correlation structure between variables. Bivariate distributions, as well as distributions in higher dimensions, are possible.

### Determining Dependence Between Simulation Inputs

One of the design decisions for a Monte Carlo simulation is a choice of probability distributions for the random inputs. Selecting a distribution for each individual variable is often straightforward, but deciding what dependencies should exist between the inputs may not be. Ideally, input data to a simulation should reflect what you know about dependence among the real quantities you are modeling. However, there may be little or no information on which to base any dependence in the simulation. In such cases, it is useful to experiment with different possibilities in order to determine the model's sensitivity.

It can be difficult to generate random inputs with dependence when they have distributions that are not from a standard multivariate distribution. Further, some of the standard multivariate distributions can model only limited types of dependence. It is always possible to make the inputs independent, and while that is a simple choice, it is not always sensible and can lead to the wrong conclusions.

For example, a Monte-Carlo simulation of financial risk could have two random inputs that represent different sources of insurance losses. You could model these inputs as

lognormal random variables. A reasonable question to ask is how dependence between these two inputs affects the results of the simulation. Indeed, you might know from real data that the same random conditions affect both sources; ignoring that in the simulation could lead to the wrong conclusions.

### Generate and Exponentiate Normal Random Variables

The `lognrnd` function simulates independent lognormal random variables. In the following example, the `mvnrnd` function generates  $n$  pairs of independent normal random variables, and then exponentiates them. Notice that the covariance matrix used here is diagonal.

```
n = 1000;

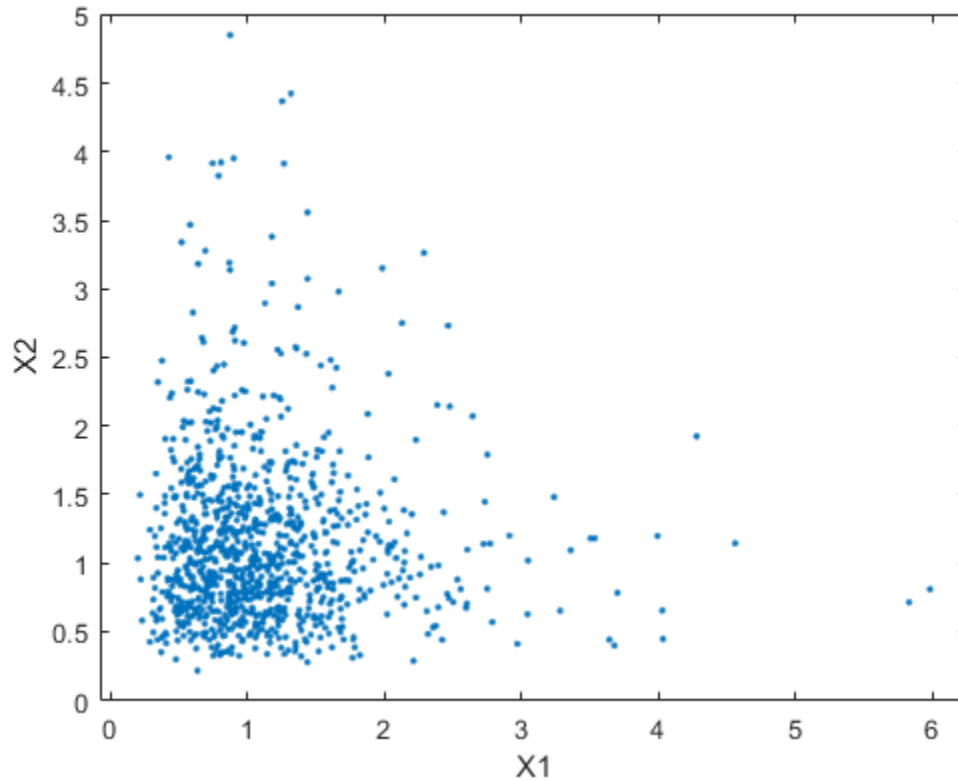
sigma = .5;
SigmaInd = sigma.^2 .* [1 0; 0 1]

rng('default'); % For reproducibility
ZInd = mvnrnd([0 0],SigmaInd,n);
XInd = exp(ZInd);

plot(XInd(:,1),XInd(:,2),'.')
axis([0 5 0 5])
axis equal
xlabel('X1')
ylabel('X2')
```

```
SigmaInd =

    0.2500    0
         0    0.2500
```



Dependent bivariate lognormal random variables are also easy to generate using a covariance matrix with nonzero off-diagonal terms.

```
rho = .7;

SigmaDep = sigma.^2 .* [1 rho; rho 1]

ZDep = mvnrnd([0 0],SigmaDep,n);
XDep = exp(ZDep);

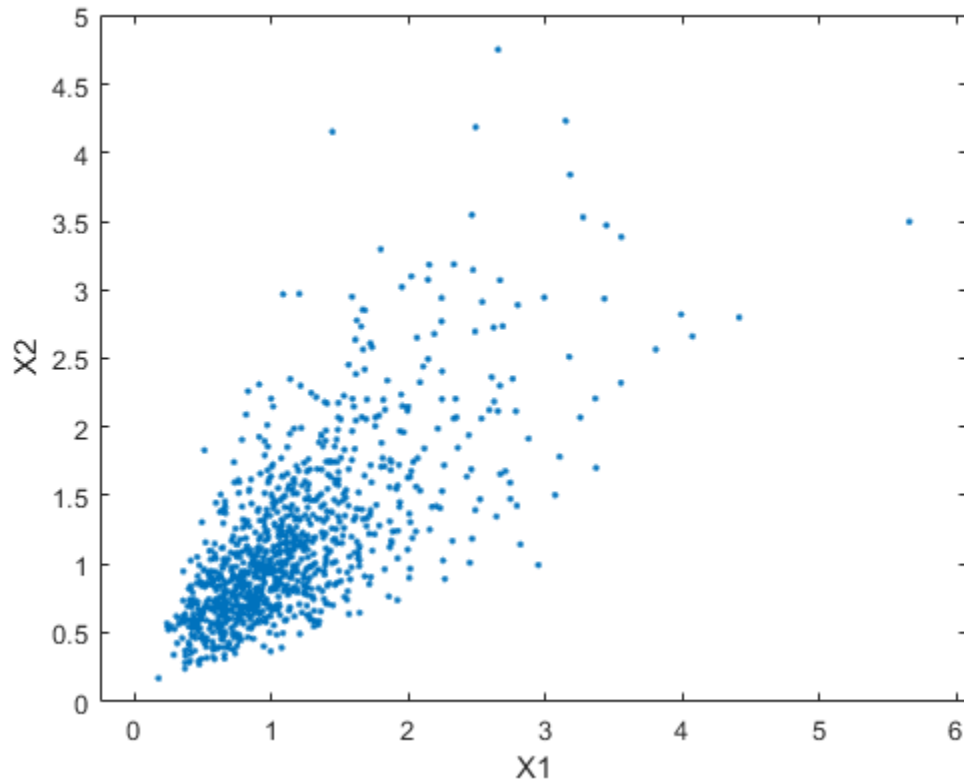
SigmaDep =

    0.2500    0.1750
```

0.1750    0.2500

A second scatter plot demonstrates the difference between these two bivariate distributions.

```
plot(XDep(:,1),XDep(:,2),'.')
axis([0 5 0 5])
axis equal
xlabel('X1')
ylabel('X2')
```



It is clear that there is a tendency in the second data set for large values of X1 to be associated with large values of X2, and similarly for small values. The correlation

parameter  $\rho$  of the underlying bivariate normal determines this dependence. The conclusions drawn from the simulation could well depend on whether you generate  $X_1$  and  $X_2$  with dependence. The bivariate lognormal distribution is a simple solution in this case; it easily generalizes to higher dimensions in cases where the marginal distributions are different lognormals.

Other multivariate distributions also exist. For example, the multivariate  $t$  and the Dirichlet distributions simulate dependent  $t$  and beta random variables, respectively. But the list of simple multivariate distributions is not long, and they only apply in cases where the marginals are all in the same family (or even the exact same distributions). This can be a serious limitation in many situations.

## Constructing Dependent Bivariate Distributions

Although the construction discussed in the previous section creates a bivariate lognormal that is simple, it serves to illustrate a method that is more generally applicable.

- 1 Generate pairs of values from a bivariate normal distribution. There is statistical dependence between these two variables, and each has a normal marginal distribution.
- 2 Apply a transformation (the exponential function) separately to each variable, changing the marginal distributions into lognormals. The transformed variables still have a statistical dependence.

If a suitable transformation can be found, this method can be generalized to create dependent bivariate random vectors with other marginal distributions. In fact, a general method of constructing such a transformation does exist, although it is not as simple as exponentiation alone.

By definition, applying the normal cumulative distribution function (cdf), denoted here by  $\Phi$ , to a standard normal random variable results in a random variable that is uniform on the interval  $[0,1]$ . To see this, if  $Z$  has a standard normal distribution, then the cdf of  $U = \Phi(Z)$  is

$$\Pr\{U \leq u\} = \Pr\{\Phi(Z) \leq u\} = \Pr\{Z \leq \Phi^{-1}(u)\} = u$$

and that is the cdf of a  $\text{Unif}(0,1)$  random variable. Histograms of some simulated normal and transformed values demonstrate that fact:

```
n = 1000;  
rng default % for reproducibility
```

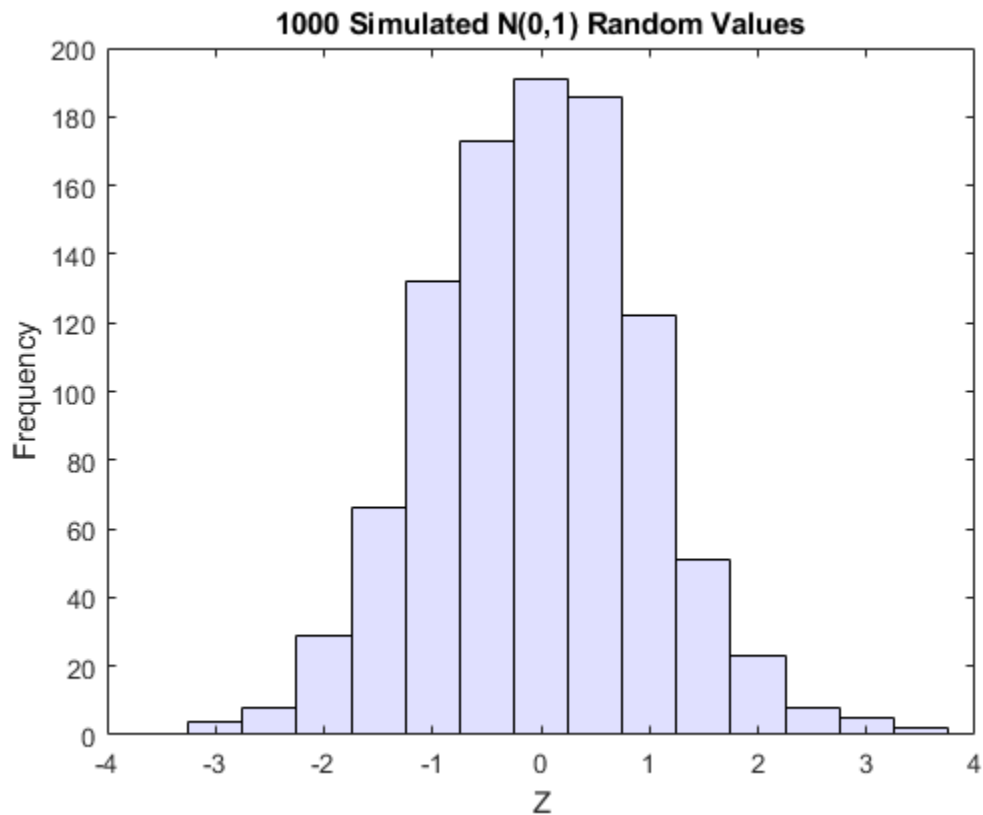


```

z = normrnd(0,1,n,1); % generate standard normal data

histogram(z,-3.75:.5:3.75,'FaceColor',[.8 .8 1]) % plot the histogram of data
xlim([-4 4])
title('1000 Simulated N(0,1) Random Values')
xlabel('Z')
ylabel('Frequency')

```

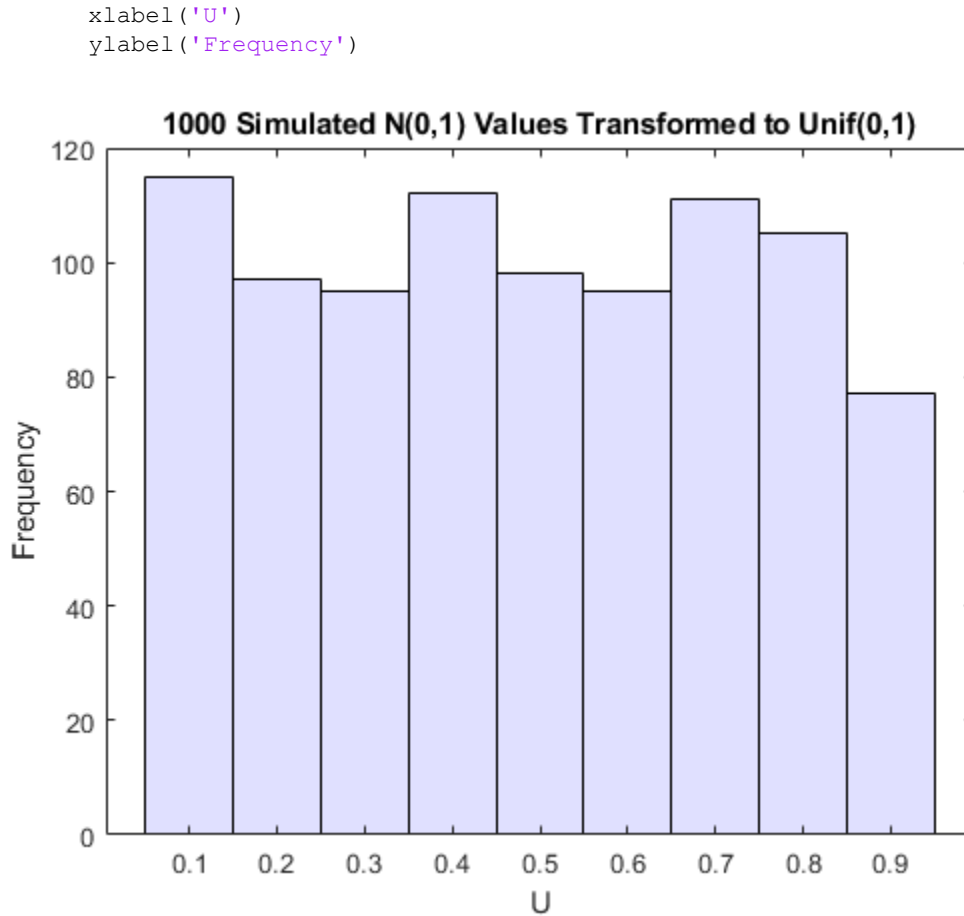


```

u = normcdf(z); % compute the cdf values of the sample data

figure
histogram(u,.05:.1:.95,'FaceColor',[.8 .8 1]) % plot the histogram of the cdf values
title('1000 Simulated N(0,1) Values Transformed to Unif(0,1)')

```



Borrowing from the theory of univariate random number generation, applying the inverse cdf of any distribution,  $F$ , to a  $\text{Unif}(0,1)$  random variable results in a random variable whose distribution is exactly  $F$  (see “Inversion Methods” on page 7-7). The proof is essentially the opposite of the preceding proof for the forward case. Another histogram illustrates the transformation to a gamma distribution:

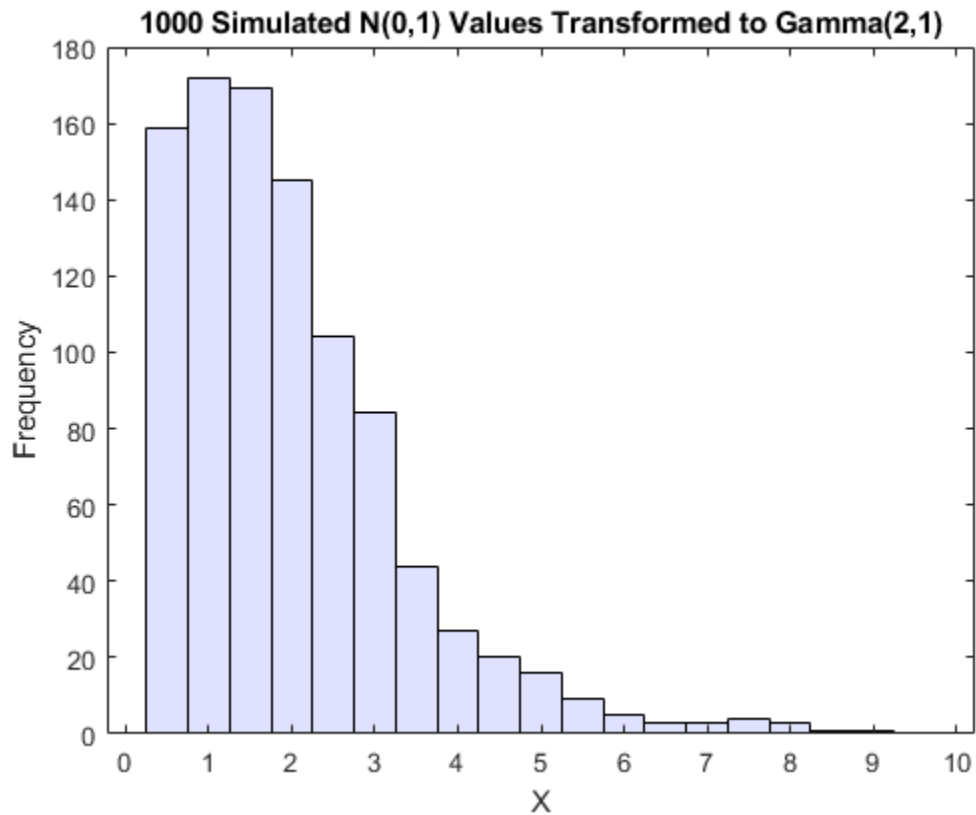
```

x = gaminv(u,2,1); % transform to gamma values

figure
histogram(x,.25:.5:9.75,'FaceColor',[.8 .8 1]) % plot the histogram of gamma values
title('1000 Simulated N(0,1) Values Transformed to Gamma(2,1)')

```

```
xlabel('X')  
ylabel('Frequency')
```



You can apply this two-step transformation to each variable of a standard bivariate normal, creating dependent random variables with arbitrary marginal distributions. Because the transformation works on each component separately, the two resulting random variables need not even have the same marginal distributions. The transformation is defined as:

$$Z = [Z_1, Z_2] \sim N\left([0, 0], \begin{bmatrix} 1 & \rho \\ \rho & 1 \end{bmatrix}\right)$$

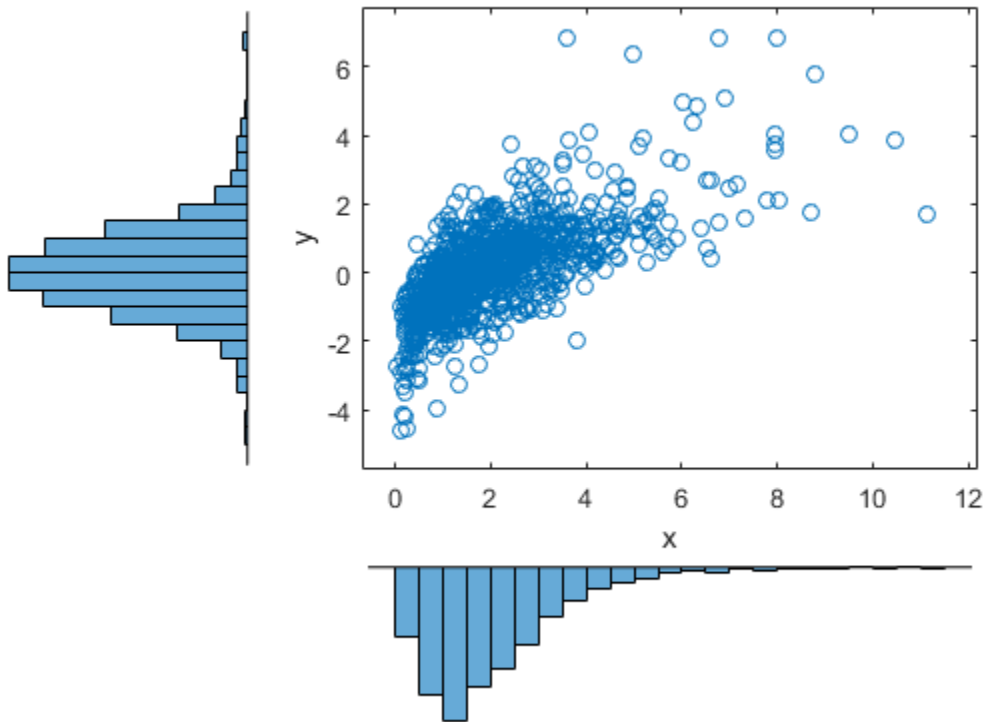
$$U = [\Phi(Z_1), \Phi(Z_2)]$$

$$X = [G_1(U_1), G_2(U_2)]$$

where  $G_1$  and  $G_2$  are inverse cdfs of two possibly different distributions. For example, the following generates random vectors from a bivariate distribution with  $t_5$  and  $\text{Gamma}(2, 1)$  marginals:

```
n = 1000; rho = .7;
Z = mvnrnd([0 0], [1 rho; rho 1], n);
U = normcdf(Z);
X = [gaminv(U(:,1), 2, 1) tinv(U(:,2), 5)];

% draw the scatter plot of data with histograms
figure
scatterhist(X(:,1), X(:,2), 'Direction', 'out')
```



This plot has histograms alongside a scatter plot to show both the marginal distributions, and the dependence.

## Using Rank Correlation Coefficients

The correlation parameter,  $\rho$ , of the underlying bivariate normal determines the dependence between  $X_1$  and  $X_2$  in this construction. However, the linear correlation of  $X_1$  and  $X_2$  is not  $\rho$ . For example, in the original lognormal case, a closed form for that correlation is:

$$\text{cor}(X_1, X_2) = \frac{e^{\rho\sigma^2} - 1}{e^{\sigma^2} - 1}$$

which is strictly less than  $\rho$ , unless  $\rho$  is exactly 1. In more general cases such as the Gamma/ $t$  construction, the linear correlation between  $X_1$  and  $X_2$  is difficult or impossible to express in terms of  $\rho$ , but simulations show that the same effect happens.

That is because the linear correlation coefficient expresses the linear dependence between random variables, and when nonlinear transformations are applied to those random variables, linear correlation is not preserved. Instead, a rank correlation coefficient, such as Kendall's  $\tau$  or Spearman's  $\rho$ , is more appropriate.

Roughly speaking, these rank correlations measure the degree to which large or small values of one random variable associate with large or small values of another. However, unlike the linear correlation coefficient, they measure the association only in terms of ranks. As a consequence, the rank correlation is preserved under any monotonic transformation. In particular, the transformation method just described preserves the rank correlation. Therefore, knowing the rank correlation of the bivariate normal  $Z$  exactly determines the rank correlation of the final transformed random variables,  $X$ . While the linear correlation coefficient,  $\rho$ , is still needed to parameterize the underlying bivariate normal, Kendall's  $\tau$  or Spearman's  $\rho$  are more useful in describing the dependence between random variables, because they are invariant to the choice of marginal distribution.

For the bivariate normal, there is a simple one-to-one mapping between Kendall's  $\tau$  or Spearman's  $\rho$ , and the linear correlation coefficient  $\rho$ :

$$\tau = \frac{2}{\pi} \arcsin(\rho) \quad \text{or} \quad \rho = \sin\left(\tau \frac{\pi}{2}\right)$$
$$\rho_s = \frac{6}{\pi} \arcsin\left(\frac{\rho}{2}\right) \quad \text{or} \quad \rho = 2 \sin\left(\rho_s \frac{\pi}{6}\right)$$

The following plot shows the relationship.

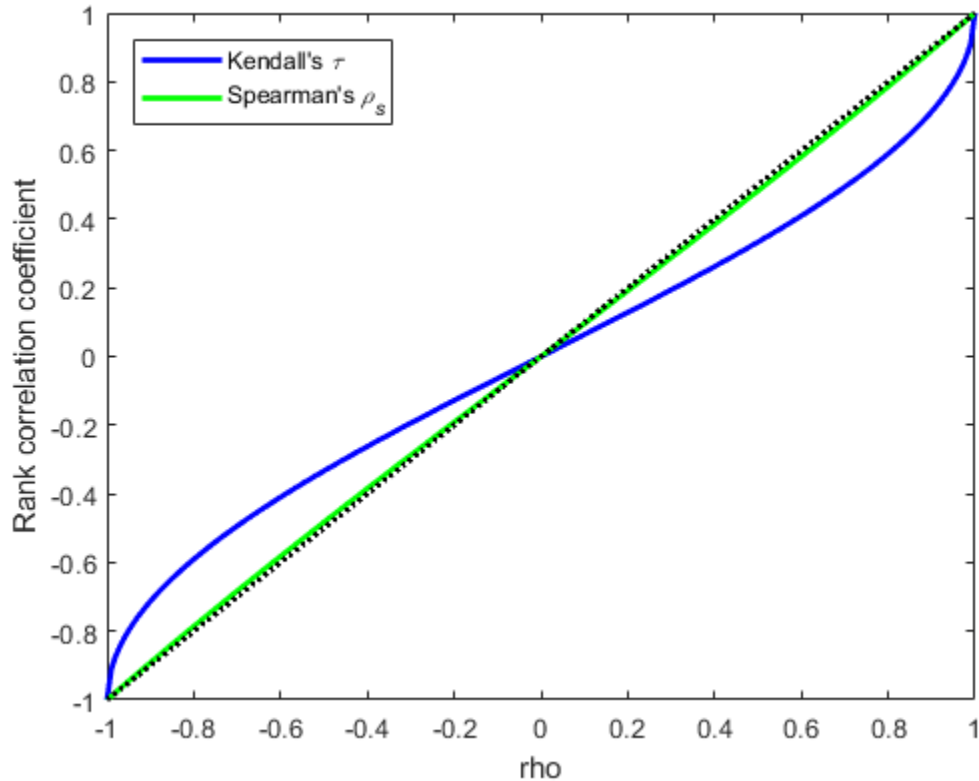
```
rho = -1:.01:1;
tau = 2.*asin(rho)./pi;
rho_s = 6.*asin(rho./2)./pi;

plot(rho,tau,'b-','LineWidth',2)
hold on
plot(rho,rho_s,'g-','LineWidth',2)
plot([-1 1],[-1 1],'k:','LineWidth',2)
axis([-1 1 -1 1])
xlabel('rho')
ylabel('Rank correlation coefficient')
```

```

legend('Kendall's  $\tau$ ', ...
       'Spearman's  $\rho_s$ ', ...
       'location', 'NW')

```



Thus, it is easy to create the desired rank correlation between  $X_1$  and  $X_2$ , regardless of their marginal distributions, by choosing the correct  $\rho$  parameter value for the linear correlation between  $Z_1$  and  $Z_2$ .

For the multivariate normal distribution, Spearman's rank correlation is almost identical to the linear correlation. However, this is not true once you transform to the final random variables.

## Using Bivariate Copulas

The first step of the construction described in the previous section defines what is known as a bivariate Gaussian copula. A copula is a multivariate probability distribution, where each random variable has a uniform marginal distribution on the unit interval  $[0,1]$ . These variables may be completely independent, deterministically related (e.g.,  $U_2 = U_1$ ), or anything in between. Because of the possibility for dependence among variables, you can use a copula to construct a new multivariate distribution for dependent variables. By transforming each of the variables in the copula separately using the inversion method, possibly using different cdfs, the resulting distribution can have arbitrary marginal distributions. Such multivariate distributions are often useful in simulations, when you know that the different random inputs are not independent of each other.

Statistics and Machine Learning Toolbox functions compute:

- Probability density functions (`copulapdf`) and the cumulative distribution functions (`copulacdf`) for Gaussian copulas
- Rank correlations from linear correlations (`copulastat`) and vice versa (`copulaparam`)
- Random vectors (`copularnd`)
- Parameters for copulas fit to data (`copulafit`)

For example, use the `copularnd` function to create scatter plots of random values from a bivariate Gaussian copula for various levels of  $\rho$ , to illustrate the range of different dependence structures. The family of bivariate Gaussian copulas is parameterized by the linear correlation matrix:

$$P = \begin{pmatrix} 1 & \rho \\ \rho & 1 \end{pmatrix}$$

$U_1$  and  $U_2$  approach linear dependence as  $\rho$  approaches  $\pm 1$ , and approach complete independence as  $\rho$  approaches zero:

```
n = 500;

rng('default') % for reproducibility
U = copularnd('Gaussian',[1 .8; .8 1],n);
subplot(2,2,1)
plot(U(:,1),U(:,2),'.')

```

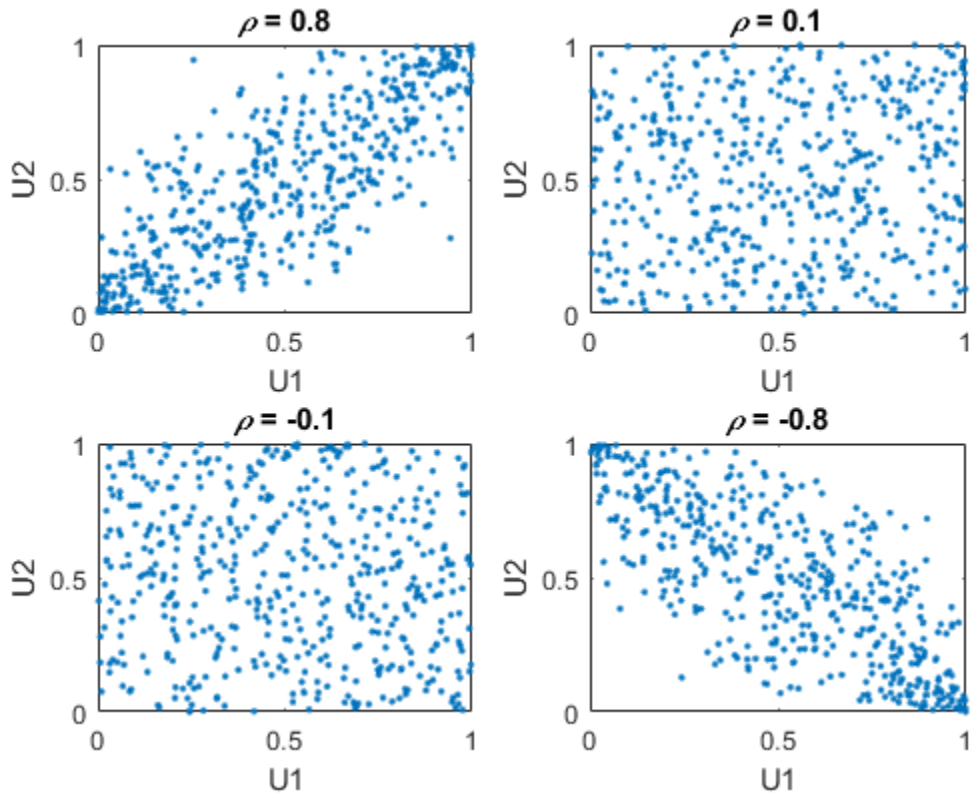


```
title('\it\rho = 0.8')
xlabel('U1')
ylabel('U2')

U = copularnd('Gaussian',[1 .1; .1 1],n);
subplot(2,2,2)
plot(U(:,1),U(:,2),'.')
title('\it\rho = 0.1')
xlabel('U1')
ylabel('U2')

U = copularnd('Gaussian',[1 -.1; -.1 1],n);
subplot(2,2,3)
plot(U(:,1),U(:,2),'.')
title('\it\rho = -0.1')
xlabel('U1')
ylabel('U2')

U = copularnd('Gaussian',[1 -.8; -.8 1],n);
subplot(2,2,4)
plot(U(:,1),U(:,2),'.')
title('\it\rho = -0.8')
xlabel('U1')
ylabel('U2')
```



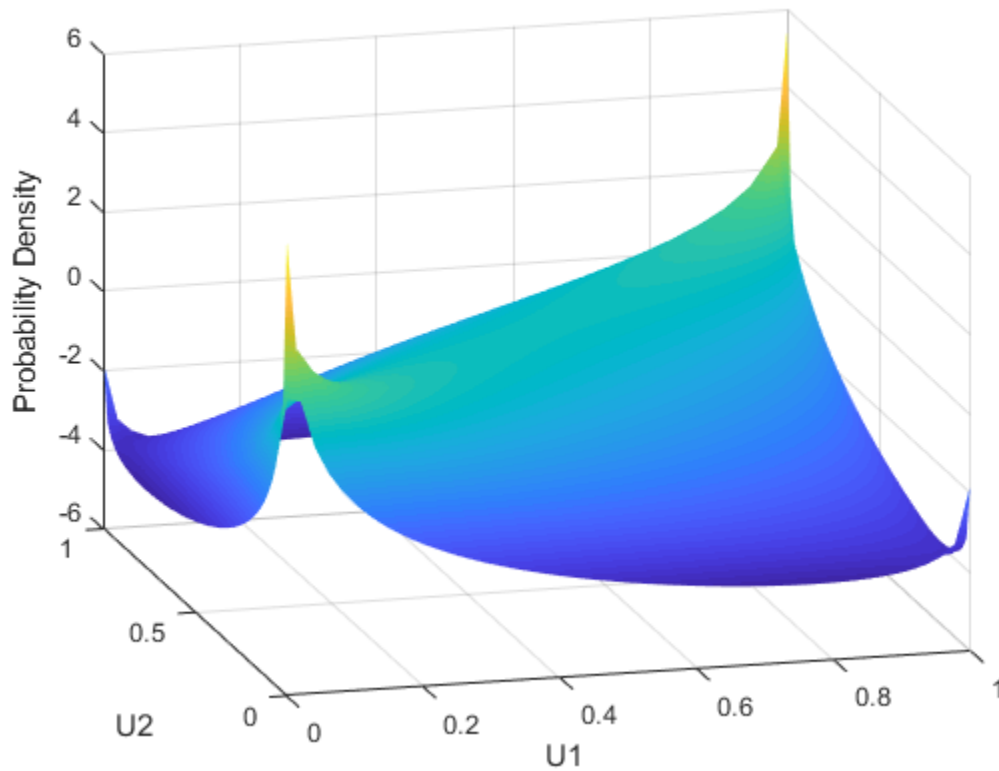
The dependence between  $U_1$  and  $U_2$  is completely separate from the marginal distributions of  $X_1 = G(U_1)$  and  $X_2 = G(U_2)$ .  $X_1$  and  $X_2$  can be given any marginal distributions, and still have the same rank correlation. This is one of the main appeals of copulas—they allow this separate specification of dependence and marginal distribution. You can also compute the pdf (`copulapdf`) and the cdf (`copulacdf`) for a copula. For example, these plots show the pdf and cdf for  $\rho = .8$ :

```
u1 = linspace(1e-3,1-1e-3,50);
u2 = linspace(1e-3,1-1e-3,50);
[U1,U2] = meshgrid(u1,u2);
Rho = [1 .8; .8 1];
f = copulapdf('t',[U1(:) U2(:)],Rho,5);
f = reshape(f,size(U1));
```

```

figure()
surf(u1,u2,log(f),'FaceColor','interp','EdgeColor','none')
view([-15,20])
xlabel('U1')
ylabel('U2')
zlabel('Probability Density')

```

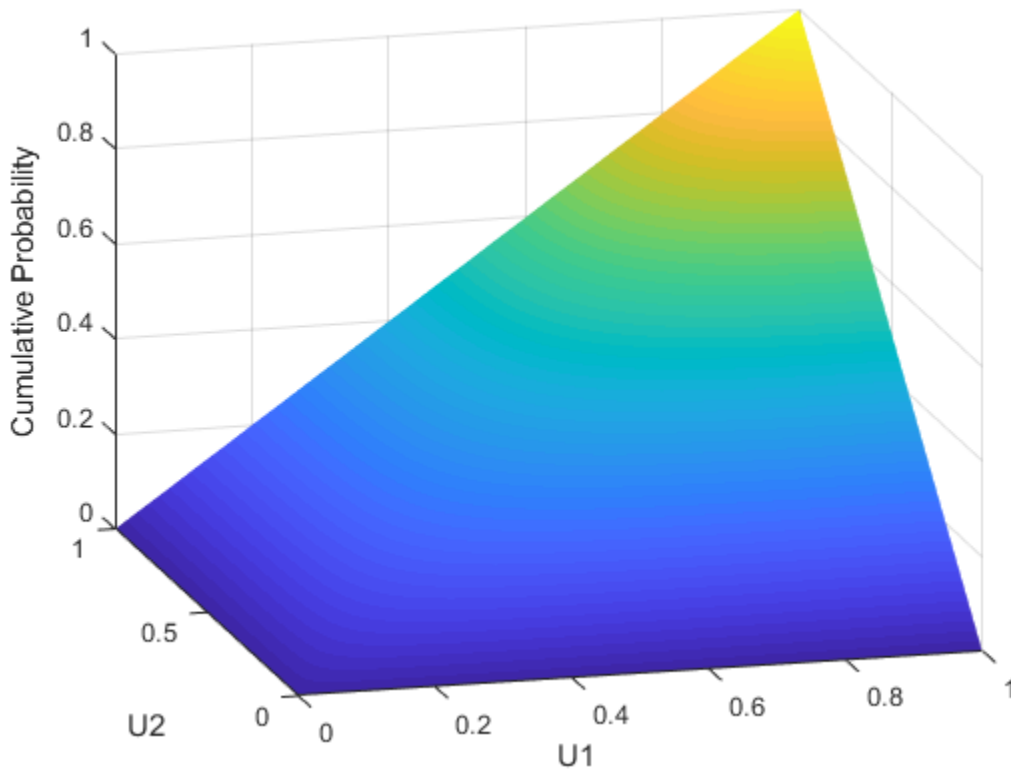


```

u1 = linspace(1e-3,1-1e-3,50);
u2 = linspace(1e-3,1-1e-3,50);
[U1,U2] = meshgrid(u1,u2);
F = copulacdf('t',[U1(:) U2(:)],Rho,5);
F = reshape(F,size(U1));

```

```
figure()
surf(u1,u2,F,'FaceColor','interp','EdgeColor','none')
view([-15,20])
xlabel('U1')
ylabel('U2')
zlabel('Cumulative Probability')
```



A different family of copulas can be constructed by starting from a bivariate  $t$  distribution and transforming using the corresponding  $t$  cdf. The bivariate  $t$  distribution is parameterized with  $P$ , the linear correlation matrix, and  $\nu$ , the degrees of freedom. Thus, for example, you can speak of a  $t_1$  or a  $t_5$  copula, based on the multivariate  $t$  with one and five degrees of freedom, respectively.

Just as for Gaussian copulas, Statistics and Machine Learning Toolbox functions for  $t$  copulas compute:

- Probability density functions (`copulapdf`) and the cumulative distribution functions (`copulacdf`) for Gaussian copulas
- Rank correlations from linear correlations (`copulastat`) and vice versa (`copulaparam`)
- Random vectors (`copularnd`)
- Parameters for copulas fit to data (`copulafit`)

For example, use the `copularnd` function to create scatter plots of random values from a bivariate  $t_1$  copula for various levels of  $\rho$ , to illustrate the range of different dependence structures:

```
n = 500;
nu = 1;

rng('default') % for reproducibility
U = copularnd('t',[1 .8; .8 1],nu,n);
subplot(2,2,1)
plot(U(:,1),U(:,2),'.')
title('{\it\rho} = 0.8')
xlabel('U1')
ylabel('U2')

U = copularnd('t',[1 .1; .1 1],nu,n);
subplot(2,2,2)
plot(U(:,1),U(:,2),'.')
title('{\it\rho} = 0.1')
xlabel('U1')
ylabel('U2')

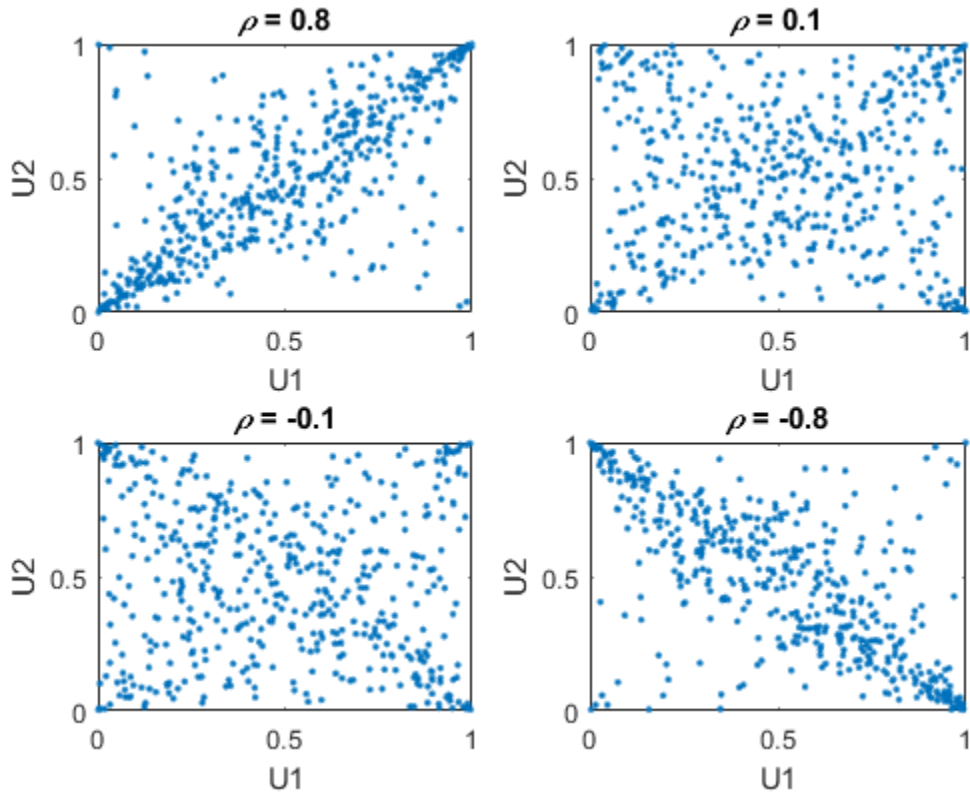
U = copularnd('t',[1 -.1; -.1 1],nu,n);
subplot(2,2,3)
plot(U(:,1),U(:,2),'.')
title('{\it\rho} = -0.1')
xlabel('U1')
ylabel('U2')

U = copularnd('t',[1 -.8; -.8 1],nu, n);
subplot(2,2,4)
plot(U(:,1),U(:,2),'.')
```

```

title('\rho = -0.8')
xlabel('U1')
ylabel('U2')

```



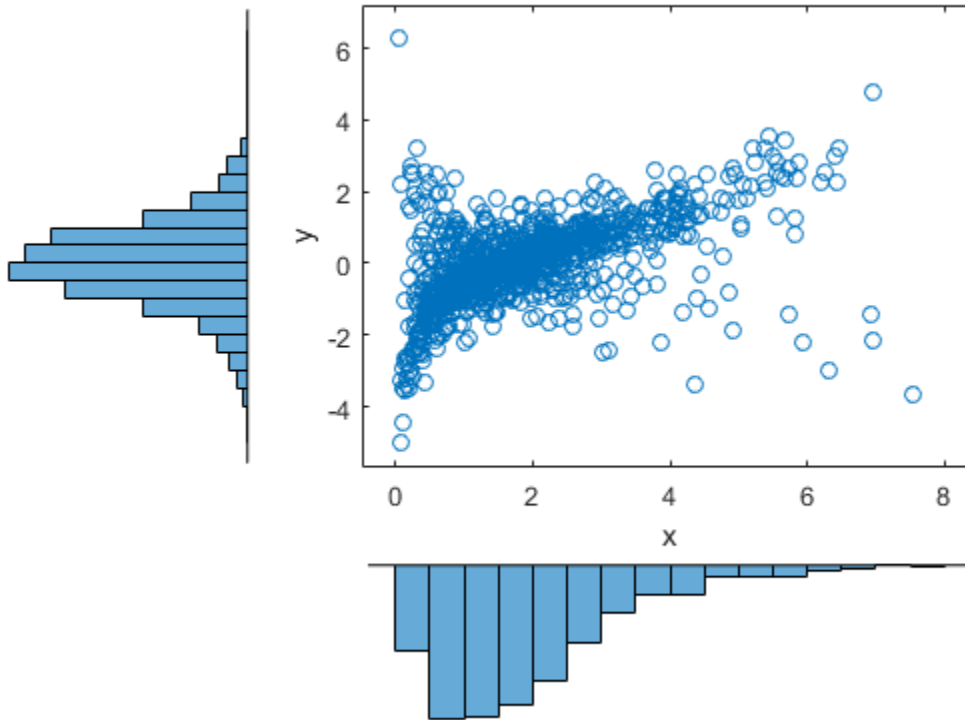
A  $t$  copula has uniform marginal distributions for  $U_1$  and  $U_2$ , just as a Gaussian copula does. The rank correlation  $\tau$  or  $\rho_s$  between components in a  $t$  copula is also the same function of  $\rho$  as for a Gaussian. However, as these plots demonstrate, a  $t_1$  copula differs quite a bit from a Gaussian copula, even when their components have the same rank correlation. The difference is in their dependence structure. Not surprisingly, as the degrees of freedom parameter  $\nu$  is made larger, a  $t_\nu$  copula approaches the corresponding Gaussian copula.

As with a Gaussian copula, any marginal distributions can be imposed over a  $t$  copula. For example, using a  $t$  copula with 1 degree of freedom, you can again generate random vectors from a bivariate distribution with Gamma(2,1) and  $t_5$  marginals using `copularnd`:

```
n = 1000;
rho = .7;
nu = 1;

rng('default') % for reproducibility
U = copularnd('t',[1 rho; rho 1],nu,n);
X = [gaminv(U(:,1),2,1) tinv(U(:,2),5)];

figure()
scatterhist(X(:,1),X(:,2),'Direction','out')
```



Compared to the bivariate Gamma/ $t$  distribution constructed earlier, which was based on a Gaussian copula, the distribution constructed here, based on a  $t_1$  copula, has the same marginal distributions and the same rank correlation between variables but a very different dependence structure. This illustrates the fact that multivariate distributions are not uniquely defined by their marginal distributions, or by their correlations. The choice of a particular copula in an application may be based on actual observed data, or different copulas may be used as a way of determining the sensitivity of simulation results to the input distribution.



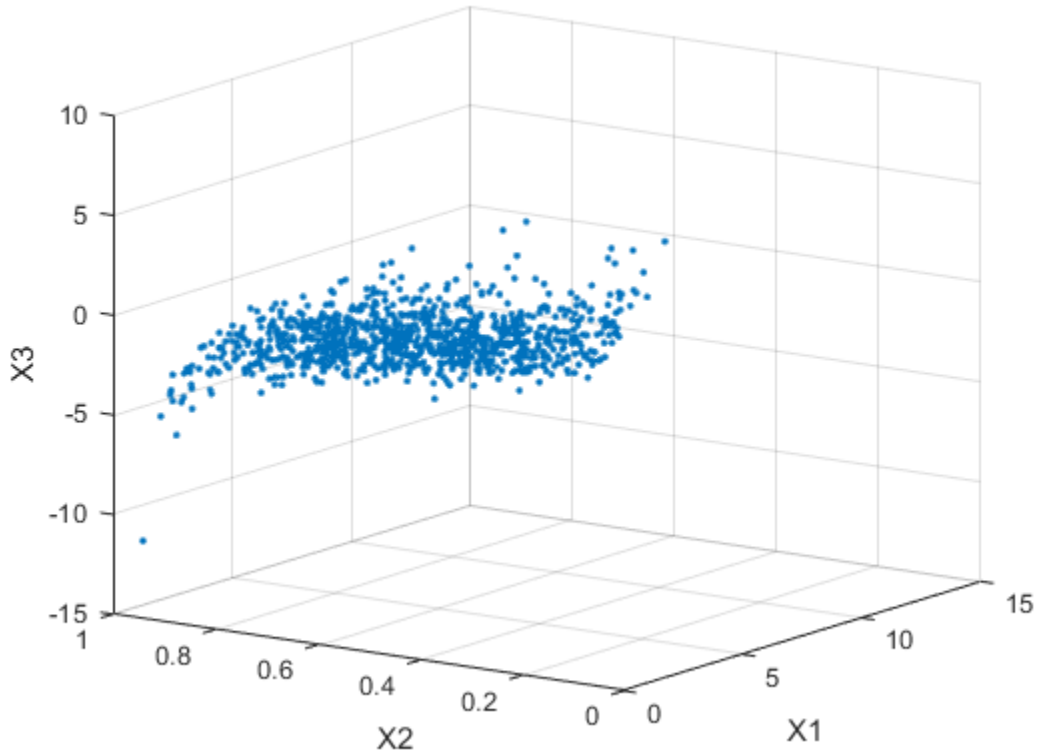
## Higher Dimension Copulas

The Gaussian and  $t$  copulas are known as elliptical copulas. It is easy to generalize elliptical copulas to a higher number of dimensions. For example, simulate data from a trivariate distribution with Gamma(2,1), Beta(2,2), and  $t_5$  marginals using a Gaussian copula and `copularnd`, as follows:

```
n = 1000;
Rho = [1 .4 .2; .4 1 -.8; .2 -.8 1];
rng('default') % for reproducibility
U = copularnd('Gaussian',Rho,n);
X = [gaminv(U(:,1),2,1) betainv(U(:,2),2,2) tinv(U(:,3),5)];
```

Plot the data.

```
subplot(1,1,1)
plot3(X(:,1),X(:,2),X(:,3),'.')
grid on
view([-55, 15])
xlabel('X1')
ylabel('X2')
zlabel('X3')
```



Notice that the relationship between the linear correlation parameter  $\rho$  and, for example, Kendall's  $\tau$ , holds for each entry in the correlation matrix  $P$  used here. You can verify that the sample rank correlations of the data are approximately equal to the theoretical values:

```
tauTheoretical = 2.*asin(Rho)./pi
```

```
tauTheoretical =
```

1.0000	0.2620	0.1282
0.2620	1.0000	-0.5903
0.1282	-0.5903	1.0000

```
tauSample = corr(X, 'type', 'Kendall')
```

```
tauSample =
```

```
    1.0000    0.2581    0.1414
    0.2581    1.0000   -0.5790
    0.1414   -0.5790    1.0000
```

## Archimedean Copulas

Statistics and Machine Learning Toolbox functions are available for three bivariate Archimedean copula families:

- Clayton copulas
- Frank copulas
- Gumbel copulas

These are one-parameter families that are defined directly in terms of their cdfs, rather than being defined constructively using a standard multivariate distribution.

To compare these three Archimedean copulas to the Gaussian and  $t$  bivariate copulas, first use the `copulastat` function to find the rank correlation for a Gaussian or  $t$  copula with linear correlation parameter of 0.8, and then use the `copulaparam` function to find the Clayton copula parameter that corresponds to that rank correlation:

```
tau = copulastat('Gaussian', .8, 'type', 'kendall')
```

```
tau =
```

```
    0.5903
```

```
alpha = copulaparam('Clayton', tau, 'type', 'kendall')
```

```
alpha =
```

```
    2.8820
```

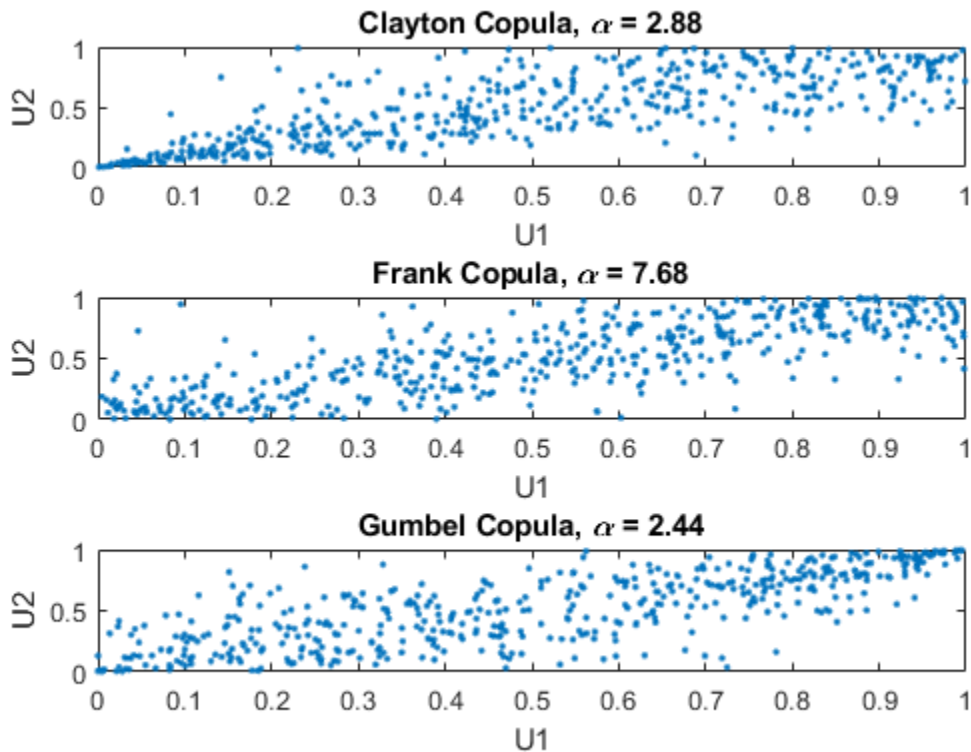
Finally, plot a random sample from the Clayton copula with `copularnd`. Repeat the same procedure for the Frank and Gumbel copulas:

```
n = 500;

U = copularnd('Clayton',alpha,n);
subplot(3,1,1)
plot(U(:,1),U(:,2),'.');
title(['Clayton Copula, {\it\alpha} = ',sprintf('%0.2f',alpha)])
xlabel('U1')
ylabel('U2')

alpha = copulaparam('Frank',tau,'type','kendall');
U = copularnd('Frank',alpha,n);
subplot(3,1,2)
plot(U(:,1),U(:,2),'.');
title(['Frank Copula, {\it\alpha} = ',sprintf('%0.2f',alpha)])
xlabel('U1')
ylabel('U2')

alpha = copulaparam('Gumbel',tau,'type','kendall');
U = copularnd('Gumbel',alpha,n);
subplot(3,1,3)
plot(U(:,1),U(:,2),'.');
title(['Gumbel Copula, {\it\alpha} = ',sprintf('%0.2f',alpha)])
xlabel('U1')
ylabel('U2')
```



## Simulating Dependent Multivariate Data Using Copulas

To simulate dependent multivariate data using a copula, you must specify each of the following:

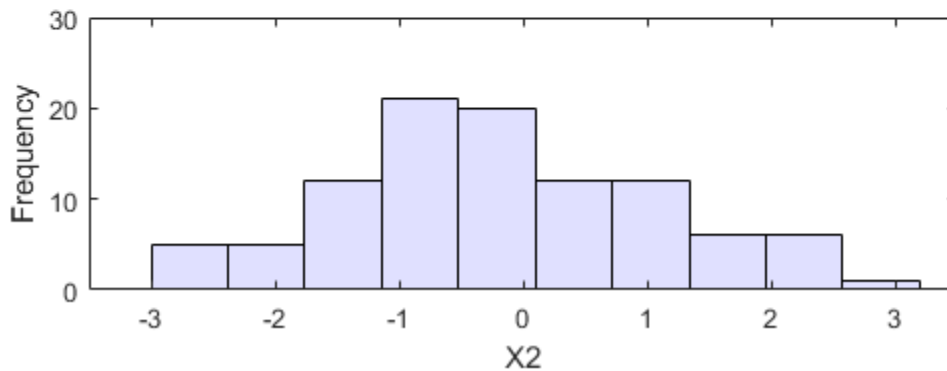
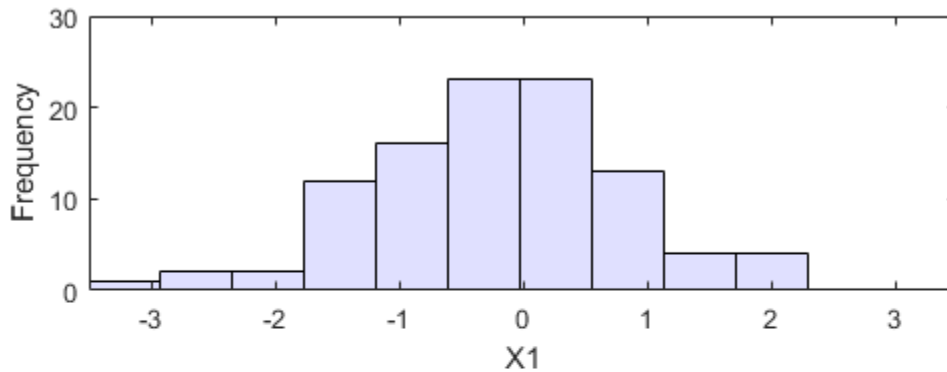
- The copula family (and any shape parameters)
- The rank correlations among variables
- Marginal distributions for each variable

Suppose you have return data for two stocks and want to run a Monte Carlo simulation with inputs that follow the same distributions as the data:

```
load stockreturns
nobs = size(stocks,1);

subplot(2,1,1)
histogram(stocks(:,1),10,'FaceColor',[.8 .8 1])
xlim([-3.5 3.5])
xlabel('X1')
ylabel('Frequency')

subplot(2,1,2)
histogram(stocks(:,2),10,'FaceColor',[.8 .8 1])
xlim([-3.5 3.5])
xlabel('X2')
ylabel('Frequency')
```



You could fit a parametric model separately to each dataset, and use those estimates as the marginal distributions. However, a parametric model may not be sufficiently flexible. Instead, you can use a nonparametric model to transform to the marginal distributions. All that is needed is a way to compute the inverse cdf for the nonparametric model.

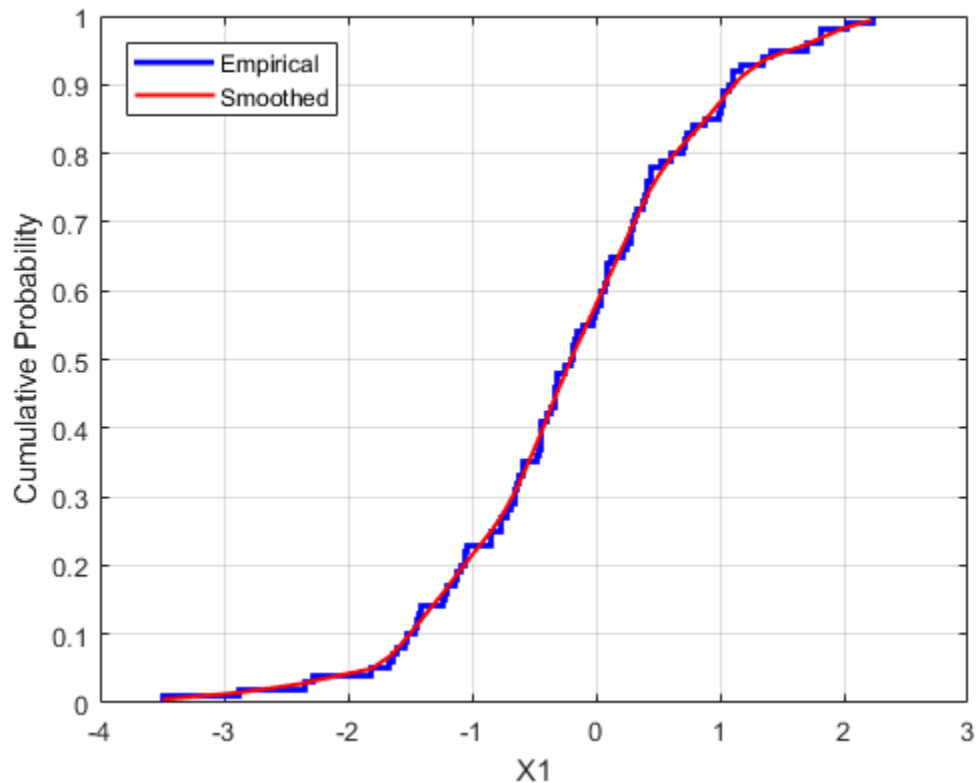
The simplest nonparametric model is the empirical cdf, as computed by the `ecdf` function. For a discrete marginal distribution, this is appropriate. However, for a continuous distribution, use a model that is smoother than the step function computed by `ecdf`. One way to do that is to estimate the empirical cdf and interpolate between the midpoints of the steps with a piecewise linear function. Another way is to use kernel smoothing with `ksdensity`. For example, compare the empirical cdf to a kernel smoothed cdf estimate for the first variable:

```
[Fi,xi] = ecdf(stocks(:,1));

figure()
stairs(xi,Fi,'b','LineWidth',2)
hold on

Fi_sm = ksdensity(stocks(:,1),xi,'function','cdf','width',.15);

plot(xi,Fi_sm,'r-','LineWidth',1.5)
xlabel('X1')
ylabel('Cumulative Probability')
legend('Empirical','Smoothed','Location','NW')
grid on
```



For the simulation, experiment with different copulas and correlations. Here, you will use a bivariate  $t$  copula with a fairly small degrees of freedom parameter. For the correlation parameter, you can compute the rank correlation of the data.

```
nu = 5;
tau = corr(stocks(:,1),stocks(:,2),'type','kendall')

tau =

    0.5180
```

Find the corresponding linear correlation parameter for the  $t$  copula using `copulaparam`.



```
rho = copulaparam('t', tau, nu, 'type','kendall')

rho =

    0.7268
```

Next, use `copularnd` to generate random values from the  $t$  copula and transform using the nonparametric inverse cdfs. The `ksdensity` function allows you to make a kernel estimate of distribution and evaluate the inverse cdf at the copula points all in one step:

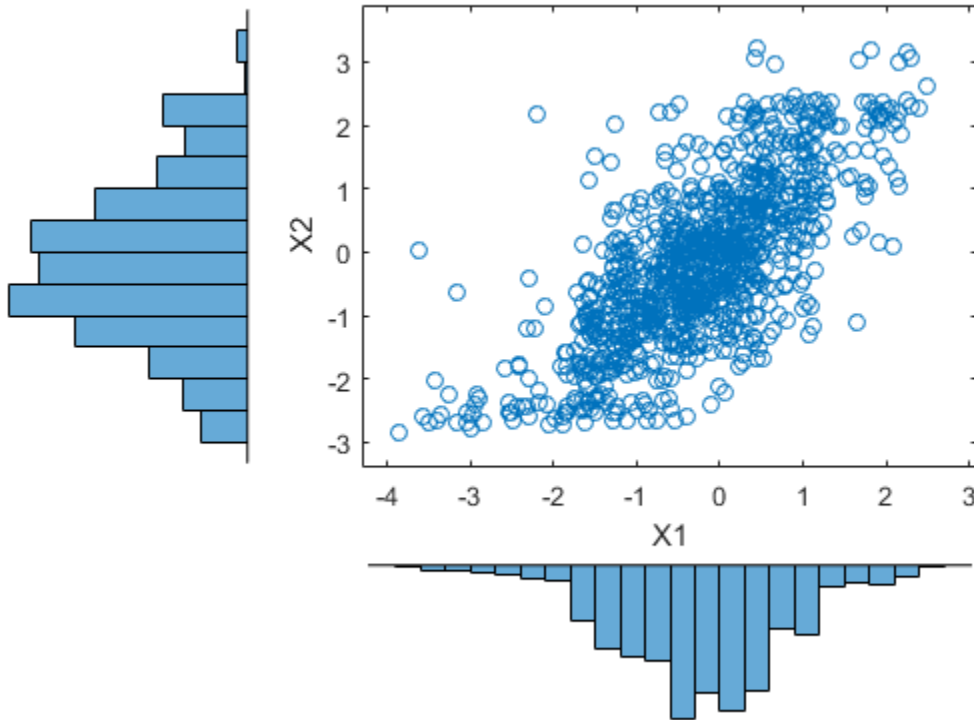
```
n = 1000;
U = copularnd('t',[1 rho; rho 1],nu,n);

X1 = ksdensity(stocks(:,1),U(:,1),...
    'function','icdf','width',.15);
X2 = ksdensity(stocks(:,2),U(:,2),...
    'function','icdf','width',.15);
```

Alternatively, when you have a large amount of data or need to simulate more than one set of values, it may be more efficient to compute the inverse cdf over a grid of values in the interval (0,1) and use interpolation to evaluate it at the copula points:

```
p = linspace(0.00001,0.99999,1000);
G1 = ksdensity(stocks(:,1),p,'function','icdf','width',0.15);
X1 = interp1(p,G1,U(:,1),'spline');
G2 = ksdensity(stocks(:,2),p,'function','icdf','width',0.15);
X2 = interp1(p,G2,U(:,2),'spline');

scatterhist(X1,X2,'Direction','out')
```



The marginal histograms of the simulated data are a smoothed version of the histograms for the original data. The amount of smoothing is controlled by the bandwidth input to `ksdensity`.

## Fitting Copulas to Data

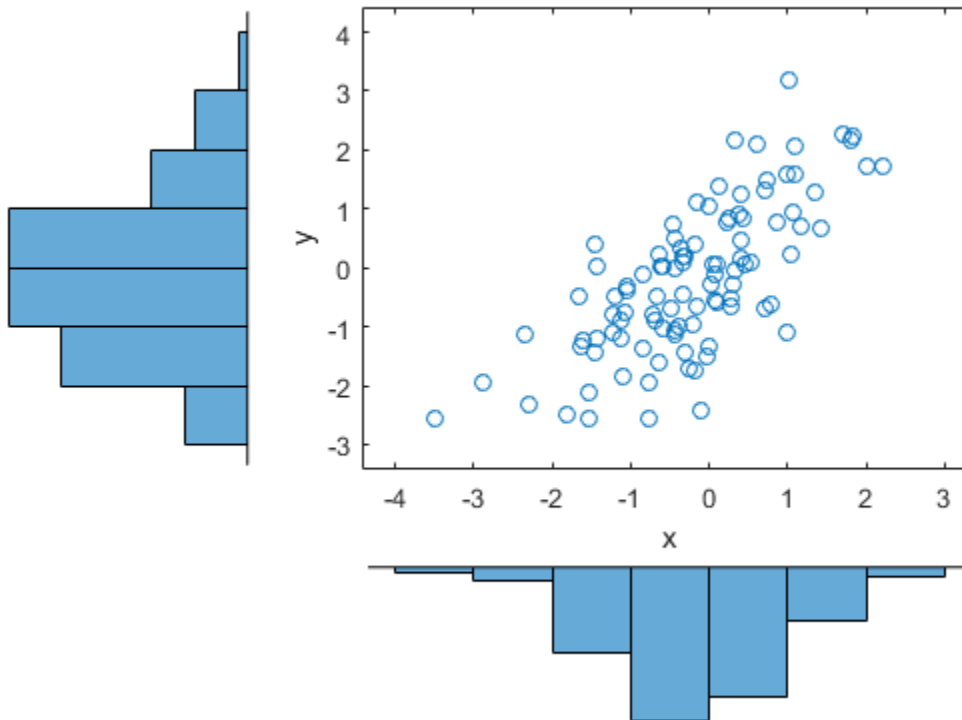
This example shows how to use `copulafit` to calibrate copulas with data. To generate data  $X_{sim}$  with a distribution "just like" (in terms of marginal distributions and correlations) the distribution of data in the matrix  $X$ , you need to fit marginal distributions to the columns of  $X$ , use appropriate cdf functions to transform  $X$  to  $U$ , so that  $U$  has values between 0 and 1, use `copulafit` to fit a copula to  $U$ , generate new

data  $U_{sim}$  from the copula, and use appropriate inverse cdf functions to transform  $U_{sim}$  to  $X_{sim}$ .

Load and plot the simulated stock return data.

```
load stockreturns
x = stocks(:,1);
y = stocks(:,2);

scatterhist(x,y,'Direction','out')
```



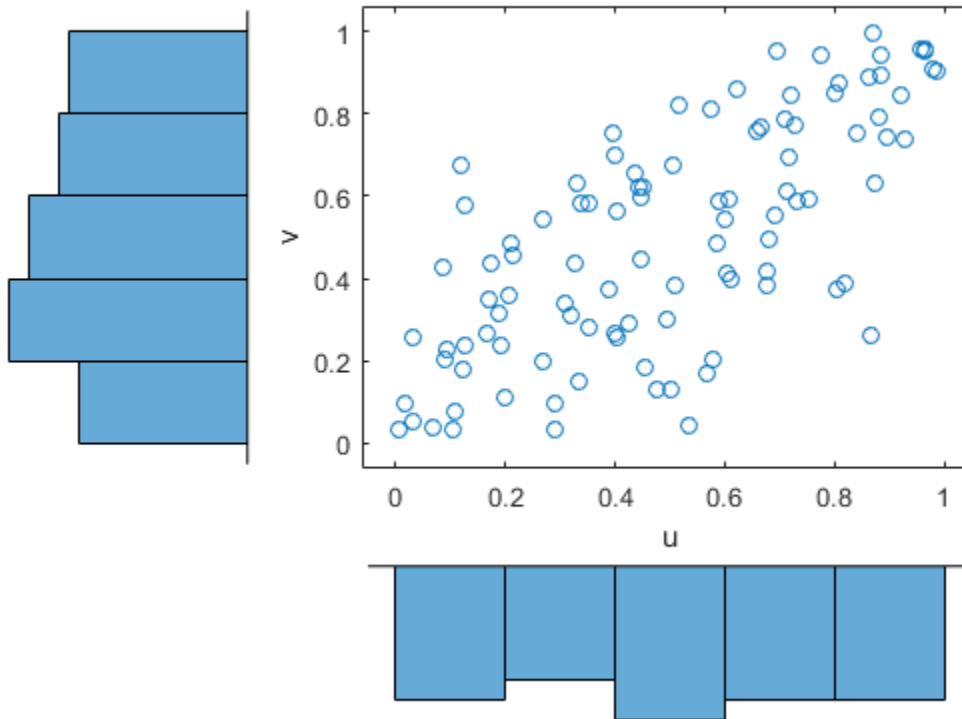
Transform the data to the copula scale (unit square) using a kernel estimator of the cumulative distribution function.

```

u = ksdensity(x,x,'function','cdf');
v = ksdensity(y,y,'function','cdf');

scatterhist(u,v,'Direction','out')
xlabel('u')
ylabel('v')

```



Fit a  $t$  copula.

```
[Rho,nu] = copulafit('t',[u v],'Method','ApproximateML')
```

Rho =

```
1.0000    0.7220
```

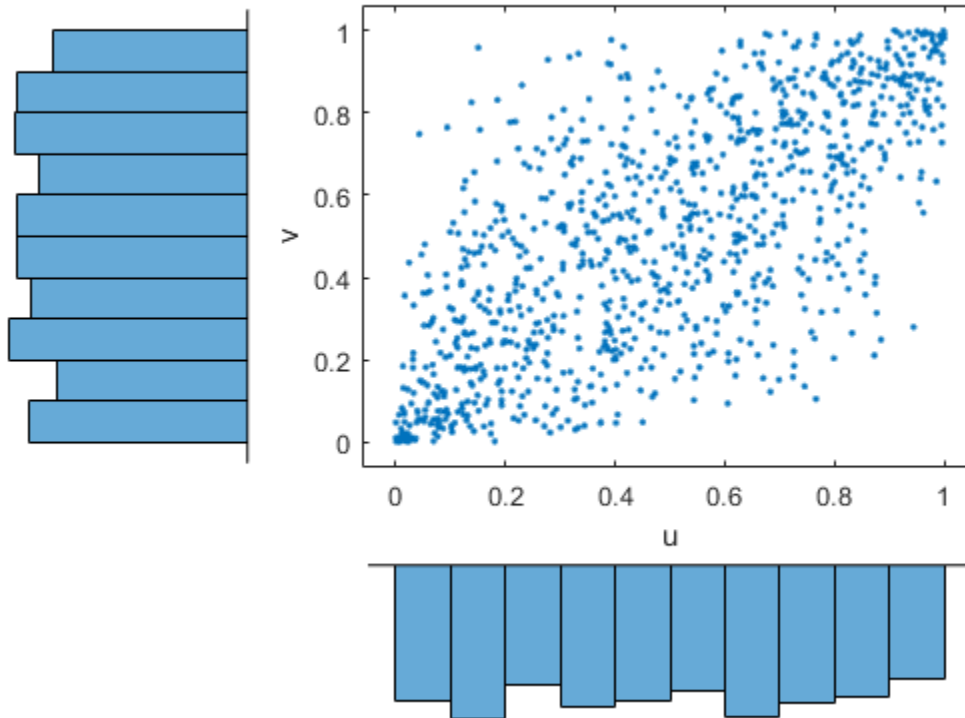
```
0.7220    1.0000
```

```
nu =
```

```
3.1800e+06
```

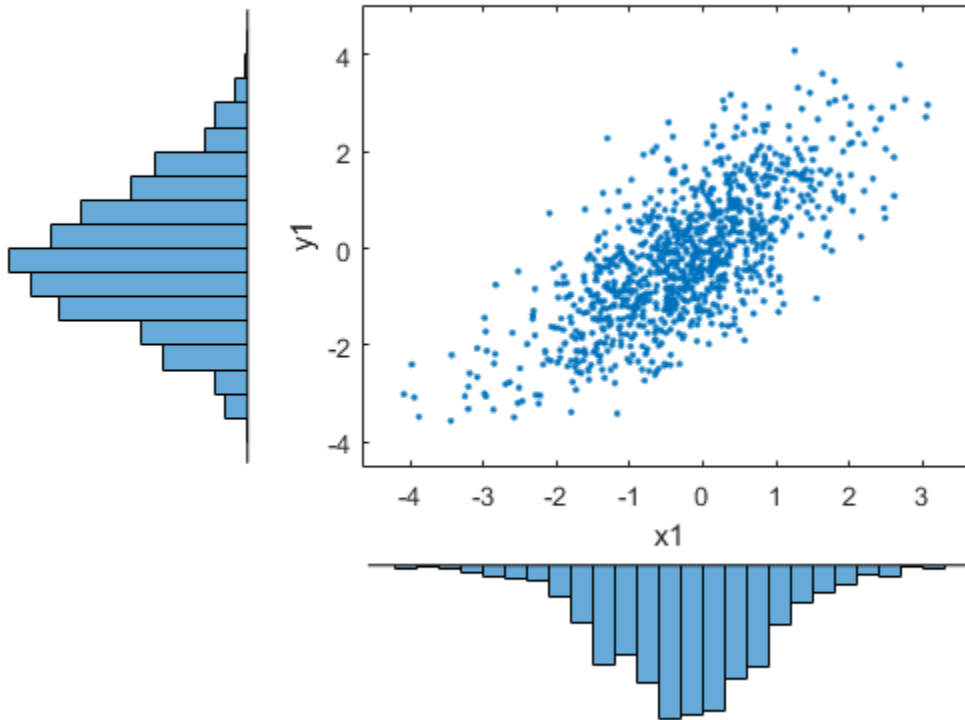
Generate a random sample from the  $t$  copula.

```
r = copularnd('t',Rho,nu,1000);  
u1 = r(:,1);  
v1 = r(:,2);  
  
scatterhist(u1,v1,'Direction','out')  
xlabel('u')  
ylabel('v')  
set(get(gca,'children'),'marker','.')
```



Transform the random sample back to the original scale of the data.

```
x1 = ksdensity(x,u1,'function','icdf');  
y1 = ksdensity(y,v1,'function','icdf');  
  
scatterhist(x1,y1,'Direction','out')  
set(get(gca,'children'),'marker','.')
```



As the example illustrates, copulas integrate naturally with other distribution fitting functions.





# Gaussian Processes

---

- “Gaussian Process Regression Models” on page 6-2
- “Kernel (Covariance) Function Options” on page 6-6
- “Exact GPR Method” on page 6-11
- “Subset of Data Approximation for GPR Models” on page 6-18
- “Subset of Regressors Approximation for GPR Models” on page 6-20
- “Fully Independent Conditional Approximation for GPR Models” on page 6-26
- “Block Coordinate Descent Approximation for GPR Models” on page 6-30

## Gaussian Process Regression Models

Gaussian process regression (GPR) models are nonparametric kernel-based probabilistic models. You can train a GPR model using the `fitrgp` function.

Consider the training set  $\{(\mathbf{x}_i, y_i); i = 1, 2, \dots, n\}$ , where  $\mathbf{x}_i \in \mathbb{R}^d$  and  $y_i \in \mathbb{R}$ , drawn from an unknown distribution. A GPR model addresses the question of predicting the value of a response variable  $y_{new}$ , given the new input vector  $\mathbf{x}_{new}$ , and the training data. A linear regression model is of the form

$$y = \mathbf{x}^T \boldsymbol{\beta} + \varepsilon,$$

where  $\varepsilon \sim N(0, \sigma^2)$ . The error variance  $\sigma^2$  and the coefficients  $\boldsymbol{\beta}$  are estimated from the data. A GPR model explains the response by introducing latent variables,

$f(\mathbf{x}_i)$ ,  $i = 1, 2, \dots, n$ , from a Gaussian process (GP), and explicit basis functions,  $h$ . The covariance function of the latent variables captures the smoothness of the response and basis functions project the inputs  $\mathbf{x}$  into a  $p$ -dimensional feature space.

A GP is a set of random variables, such that any finite number of them have a joint

Gaussian distribution. If  $\{f(\mathbf{x}), \mathbf{x} \in \mathbb{R}^d\}$  is a GP, then given  $n$  observations  $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n$ , the joint distribution of the random variables  $f(\mathbf{x}_1), f(\mathbf{x}_2), \dots, f(\mathbf{x}_n)$  is Gaussian. A GP is defined by its mean function  $m(\mathbf{x})$  and covariance function,  $k(\mathbf{x}, \mathbf{x}')$ . That is, if

$\{f(\mathbf{x}), \mathbf{x} \in \mathbb{R}^d\}$  is a Gaussian process, then  $E(f(\mathbf{x})) = m(\mathbf{x})$  and

$$\text{Cov}[f(\mathbf{x}), f(\mathbf{x}')] = E[\{f(\mathbf{x}) - m(\mathbf{x})\}\{f(\mathbf{x}') - m(\mathbf{x}')\}] = k(\mathbf{x}, \mathbf{x}').$$

Now consider the following model.

$$\mathbf{h}(\mathbf{x})^T \boldsymbol{\beta} + f(\mathbf{x}),$$

where  $f(\mathbf{x}) \sim GP(0, k(\mathbf{x}, \mathbf{x}'))$ , that is  $f(x)$  are from a zero mean GP with covariance

function,  $k(\mathbf{x}, \mathbf{x}')$ .  $h(x)$  are a set of basis functions that transform the original feature vector  $x$  in  $\mathbb{R}^d$  into a new feature vector  $h(x)$  in  $\mathbb{R}^p$ .  $\boldsymbol{\beta}$  is a  $p$ -by-1 vector of basis function

coefficients. This model represents a GPR model. An instance of response  $y$  can be modeled as

$$P(y_i | f(\mathbf{x}_i), \mathbf{x}_i) \sim N(y_i | \mathbf{h}(\mathbf{x}_i)^T \boldsymbol{\beta} + f(\mathbf{x}_i), \sigma^2)$$

Hence, a GPR model is a probabilistic model. There is a latent variable  $f(\mathbf{x}_i)$  introduced for each observation  $\mathbf{x}_i$ , which makes the GPR model nonparametric. In vector form, this model is equivalent to

$$P(\mathbf{y} | \mathbf{f}, \mathbf{X}) \sim N(\mathbf{y} | \mathbf{H}\boldsymbol{\beta} + \mathbf{f}, \sigma^2 \mathbf{I}),$$

where

$$\mathbf{X} = \begin{pmatrix} \mathbf{x}_1^T \\ \mathbf{x}_2^T \\ \vdots \\ \mathbf{x}_n^T \end{pmatrix}, \quad \mathbf{y} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}, \quad \mathbf{H} = \begin{pmatrix} \mathbf{h}(\mathbf{x}_1^T) \\ \mathbf{h}(\mathbf{x}_2^T) \\ \vdots \\ \mathbf{h}(\mathbf{x}_n^T) \end{pmatrix}, \quad \mathbf{f} = \begin{pmatrix} f(\mathbf{x}_1) \\ f(\mathbf{x}_2) \\ \vdots \\ f(\mathbf{x}_n) \end{pmatrix}.$$

The joint distribution of latent variables  $f(\mathbf{x}_1), f(\mathbf{x}_2), \dots, f(\mathbf{x}_n)$  in the GPR model is as follows:

$$P(\mathbf{f} | \mathbf{X}) \sim N(\mathbf{f} | \mathbf{0}, \mathbf{K}(\mathbf{X}, \mathbf{X})),$$

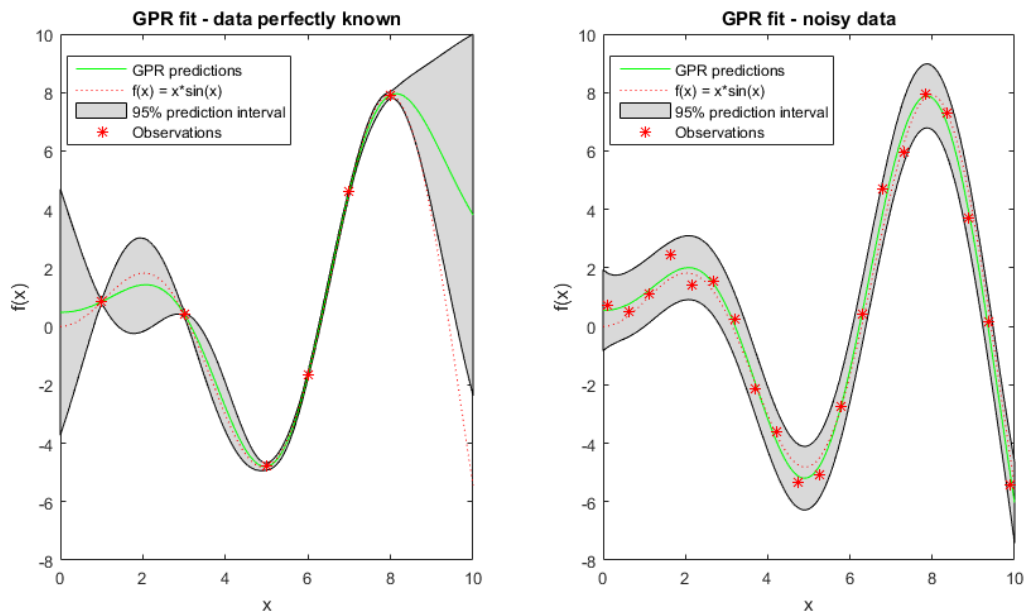
close to a linear regression model, where  $\mathbf{K}(\mathbf{X}, \mathbf{X})$  looks as follows:

$$\mathbf{K}(\mathbf{X}, \mathbf{X}) = \begin{pmatrix} k(\mathbf{x}_1, \mathbf{x}_1) & k(\mathbf{x}_1, \mathbf{x}_2) & \cdots & k(\mathbf{x}_1, \mathbf{x}_n) \\ k(\mathbf{x}_2, \mathbf{x}_1) & k(\mathbf{x}_2, \mathbf{x}_2) & \cdots & k(\mathbf{x}_2, \mathbf{x}_n) \\ \vdots & \vdots & \ddots & \vdots \\ k(\mathbf{x}_n, \mathbf{x}_1) & k(\mathbf{x}_n, \mathbf{x}_2) & \cdots & k(\mathbf{x}_n, \mathbf{x}_n) \end{pmatrix}.$$

The covariance function  $k(\mathbf{x}, \mathbf{x}')$  is usually parameterized by a set of kernel parameters or hyperparameters,  $\boldsymbol{\theta}$ . Often  $k(\mathbf{x}, \mathbf{x}')$  is written as  $k(\mathbf{x}, \mathbf{x}' | \boldsymbol{\theta})$  to explicitly indicate the dependence on  $\boldsymbol{\theta}$ .

`fitrgp` estimates the basis function coefficients,  $\beta$ , the noise variance,  $\sigma^2$ , and the hyperparameters,  $\theta$ , of the kernel function from the data while training the GPR model. You can specify the basis function, the kernel (covariance) function, and the initial values for the parameters.

Because a GPR model is probabilistic, it is possible to compute the prediction intervals using the trained model (see `predict` and `resubPredict`). Consider some data observed from the function  $g(x) = x \cdot \sin(x)$ , and assume that they are noise free. The subplot on the left in the following figure illustrates the observations, the GPR fit, and the actual function. It is more realistic that the observed values are not the exact function values, but a noisy realization of them. The subplot on the right illustrates this case. When observations are noise free (as in the subplot on the left), the GPR fit crosses the observations, and the standard deviation of the predicted response is zero. Hence, you do not see prediction intervals around these values.



You can also compute the regression error using the trained GPR model (see `loss` and `resubLoss`).

## References

- [1] Rasmussen, C. E. and C. K. I. Williams. *Gaussian Processes for Machine Learning*. MIT Press. Cambridge, Massachusetts, 2006.

## See Also

RegressionGP | fitrgp

## More About

- “Exact GPR Method” on page 6-11
- “Subset of Data Approximation for GPR Models” on page 6-18
- “Subset of Regressors Approximation for GPR Models” on page 6-20
- “Fully Independent Conditional Approximation for GPR Models” on page 6-26
- “Block Coordinate Descent Approximation for GPR Models” on page 6-30

## Kernel (Covariance) Function Options

In supervised learning, it is expected that the points with similar predictor values  $\mathbf{x}_i$ , naturally have close response (target) values  $y_i$ . In Gaussian processes, the covariance function expresses this similarity [1]. It specifies the covariance between the two latent variables  $f(\mathbf{x}_i)$  and  $f(\mathbf{x}_j)$ , where both  $\mathbf{x}_i$  and  $\mathbf{x}_j$  are  $d$ -by-1 vectors. In other words, it determines how the response at one point  $\mathbf{x}_i$  is affected by responses at other points  $\mathbf{x}_j$ ,  $i \neq j$ ,  $i = 1, 2, \dots, n$ . The covariance function  $k(\mathbf{x}_i, \mathbf{x}_j)$  can be defined by various kernel functions. It can be parameterized in terms of the kernel parameters in vector  $\boldsymbol{\theta}$ . Hence, it is possible to express the covariance function as  $k(\mathbf{x}_i, \mathbf{x}_j | \boldsymbol{\theta})$ .

For many standard kernel functions, the kernel parameters are based on the signal standard deviation  $\sigma_f$  and the characteristic length scale  $\sigma_l$ . The characteristic length scales briefly define how far apart the input values  $\mathbf{x}_i$  can be for the response values to become uncorrelated. Both  $\sigma_l$  and  $\sigma_f$  need to be greater than 0, and this can be enforced by the unconstrained parametrization vector  $\boldsymbol{\theta}$ , such that  $\theta_1 = \log \sigma_l$ ,  $\theta_2 = \log \sigma_f$ .

The built-in kernel (covariance) functions *with same length scale for each predictor* are:

- **Squared Exponential Kernel**

This is one of the most commonly used covariance functions and is the default option for `fitrgp`. The squared exponential kernel function is defined as

$$k(\mathbf{x}_i, \mathbf{x}_j | \boldsymbol{\theta}) = \sigma_f^2 \exp \left[ -\frac{1}{2} \frac{(\mathbf{x}_i - \mathbf{x}_j)^T (\mathbf{x}_i - \mathbf{x}_j)}{\sigma_l^2} \right].$$

where  $\sigma_l$  is the characteristic length scale, and  $\sigma_f$  is the signal standard deviation.

- **Exponential Kernel**

You can specify the exponential kernel function using the 'KernelFunction', 'exponential' name-value pair argument. This covariance function is defined by

$$k(\mathbf{x}_i, \mathbf{x}_j | \boldsymbol{\theta}) = \sigma_f^2 \exp\left(-\frac{r}{\sigma_l}\right),$$

where  $\sigma_l$  is the characteristic length scale and

$$r = \sqrt{(\mathbf{x}_i - \mathbf{x}_j)^T (\mathbf{x}_i - \mathbf{x}_j)}$$

is the Euclidean distance between  $\mathbf{x}_i$  and  $\mathbf{x}_j$ .

- **Matern 3/2**

You can specify the Matern 3/2 kernel function using the 'KernelFunction', 'matern32' name-value pair argument. This covariance function is defined by

$$k(\mathbf{x}_i, \mathbf{x}_j | \boldsymbol{\theta}) = \sigma_f^2 \left(1 + \frac{\sqrt{3}r}{\sigma_l}\right) \exp\left(-\frac{\sqrt{3}r}{\sigma_l}\right),$$

where

$$r = \sqrt{(\mathbf{x}_i - \mathbf{x}_j)^T (\mathbf{x}_i - \mathbf{x}_j)}$$

is the Euclidean distance between  $\mathbf{x}_i$  and  $\mathbf{x}_j$ .

- **Matern 5/2**

You can specify the Matern 5/2 kernel function using the 'KernelFunction', 'matern52' name-value pair argument. The Matern 5/2 covariance function is defined as

$$k(\mathbf{x}_i, \mathbf{x}_j) = \sigma_f^2 \left(1 + \frac{\sqrt{5}r}{\sigma_l} + \frac{5r^2}{3\sigma_l^2}\right) \exp\left(-\frac{\sqrt{5}r}{\sigma_l}\right),$$

where

$$r = \sqrt{(\mathbf{x}_i - \mathbf{x}_j)^T (\mathbf{x}_i - \mathbf{x}_j)}$$

is the Euclidean distance between  $\mathbf{x}_i$  and  $\mathbf{x}_j$ .

- **Rational Quadratic Kernel**

You can specify the rational quadratic kernel function using the 'KernelFunction', 'rationalquadratic' name-value pair argument. This covariance function is defined by

$$k(\mathbf{x}_i, \mathbf{x}_j | \boldsymbol{\theta}) = \sigma_f^2 \left( 1 + \frac{r^2}{2\alpha\sigma_l^2} \right)^{-\alpha},$$

where  $\sigma_l$  is the characteristic length scale,  $\alpha$  is a positive-valued scale-mixture parameter, and

$$r = \sqrt{(\mathbf{x}_i - \mathbf{x}_j)^T (\mathbf{x}_i - \mathbf{x}_j)}$$

is the Euclidean distance between  $\mathbf{x}_i$  and  $\mathbf{x}_j$ .

It is possible to use a separate length scale  $\sigma_m$  for each predictor  $m$ ,  $m = 1, 2, \dots, d$ . The built-in kernel (covariance) functions with a separate length scale for each predictor implement automatic relevance determination (ARD) [2]. The unconstrained parametrization  $\boldsymbol{\theta}$  in this case is

$$\theta_m = \log \sigma_m, \quad \text{for } m = 1, 2, \dots, d$$

$$\theta_{d+1} = \log \sigma_f.$$

The built-in kernel (covariance) functions *with separate length scale for each predictor* are:

- **ARD Squared Exponential Kernel**

You can specify this kernel function using the 'KernelFunction', 'ardsquaredexponential' name-value pair argument. This covariance function is the squared exponential kernel function, with a separate length scale for each predictor. It is defined as

$$k(x_i, x_j | \boldsymbol{\theta}) = \sigma_f^2 \exp \left[ -\frac{1}{2} \sum_{m=1}^d \frac{(x_{im} - x_{jm})^2}{\sigma_m^2} \right].$$

- **ARD Exponential Kernel**



You can specify this kernel function using the 'KernelFunction', 'ardexponential' name-value pair argument. This covariance function is the exponential kernel function, with a separate length scale for each predictor. It is defined as

$$k(\mathbf{x}_i, \mathbf{x}_j | \boldsymbol{\theta}) = \sigma_f^2 \exp(-r),$$

where

$$r = \sqrt{\sum_{m=1}^d \frac{(x_{im} - x_{jm})^2}{\sigma_m^2}}.$$

- **ARD Matern 3/2**

You can specify this kernel function using the 'KernelFunction', 'ardmatern32' name-value pair argument. This covariance function is the Matern 3/2 kernel function, with a different length scale for each predictor. It is defined as

$$k(\mathbf{x}_i, \mathbf{x}_j | \boldsymbol{\theta}) = \sigma_f^2 (1 + \sqrt{3} r) \exp(-\sqrt{3} r),$$

where

$$r = \sqrt{\sum_{m=1}^d \frac{(x_{im} - x_{jm})^2}{\sigma_m^2}}.$$

- **ARD Matern 5/2**

You can specify this kernel function using the 'KernelFunction', 'ardmatern52' name-value pair argument. This covariance function is the Matern 5/2 kernel function, with a different length scale for each predictor. It is defined as

$$k(\mathbf{x}_i, \mathbf{x}_j | \boldsymbol{\theta}) = \sigma_f^2 \left( 1 + \sqrt{5} r + \frac{5}{3} r^2 \right) \exp(-\sqrt{5} r),$$

where

$$r = \sqrt{\sum_{m=1}^d \frac{(x_{im} - x_{jm})^2}{\sigma_m^2}}.$$

- **ARD Rational Quadratic Kernel**

You can specify this kernel function using the 'KernelFunction', 'ardrationalquadratic' name-value pair argument. This

covariance function is the rational quadratic kernel function, with a separate length scale for each predictor. It is defined as

$$k(\mathbf{x}_i, \mathbf{x}_j | \boldsymbol{\theta}) = \sigma_f^2 \left( 1 + \frac{1}{2\alpha} \sum_{m=1}^d \frac{(x_{im} - x_{jm})^2}{\sigma_m^2} \right)^{-\alpha}.$$

You can specify the kernel function using the `KernelFunction` name-value pair argument in a call to `fitrgp`. You can either specify one of the built-in kernel parameter options, or specify a custom function. When providing the initial kernel parameter values for a built-in kernel function, input the initial values for signal standard deviation and the characteristic length scale(s) as a numeric vector. When providing the initial kernel parameter values for a custom kernel function, input the initial values the unconstrained parametrization vector  $\boldsymbol{\theta}$ . `fitrgp` uses analytical derivatives to estimate parameters when using a built-in kernel function, whereas when using a custom kernel function it uses numerical derivatives.

## References

- [1] Rasmussen, C. E. and C. K. I. Williams. *Gaussian Processes for Machine Learning*. MIT Press. Cambridge, Massachusetts, 2006.
- [2] Neal, R. M. *Bayesian Learning for Neural Networks*. Springer, New York. Lecture Notes in Statistics, 118, 1996.

## See Also

RegressionGP | `fitrgp`

## Related Examples

- “Use Separate Length Scales for Predictors” on page 32-2330
- “Fit GPR Model Using Custom Kernel Function” on page 32-2343
- “Impact of Specifying Initial Kernel Parameter Values” on page 32-2328

## Exact GPR Method

### In this section...

“Parameter Estimation” on page 6-11

“Prediction” on page 6-13

“Computational Complexity of Exact Parameter Estimation and Prediction” on page 6-16

An instance of response  $y$  from a Gaussian process regression (GPR) model on page 6-2 can be modeled as

$$P(y_i | f(\mathbf{x}_i), \mathbf{x}_i) \sim N\left(y_i \mid \mathbf{h}(\mathbf{x}_i)^T \boldsymbol{\beta} + f(\mathbf{x}_i), \sigma^2\right)$$

Hence, making predictions for new data from a GPR model requires:

- Knowledge of the coefficient vector,  $\boldsymbol{\beta}$ , of fixed basis functions
- Ability to evaluate the covariance function  $k(\mathbf{x}, \mathbf{x}' | \boldsymbol{\theta})$  for arbitrary  $\mathbf{x}$  and  $\mathbf{x}'$ , given the kernel parameters or hyperparameters,  $\boldsymbol{\theta}$ .
- Knowledge of the noise variance  $\sigma^2$  that appears in the density  $P(y_i | f(\mathbf{x}_i), \mathbf{x}_i)$

That is, one needs first to estimate  $\boldsymbol{\beta}$ ,  $\boldsymbol{\theta}$ , and  $\sigma^2$  from the data  $(\mathbf{X}, \mathbf{y})$ .

### Parameter Estimation

One approach for estimating the parameters  $\boldsymbol{\beta}$ ,  $\boldsymbol{\theta}$ , and  $\sigma^2$  of a GPR model is by maximizing the likelihood  $P(\mathbf{y} | \mathbf{X})$  as a function of  $\boldsymbol{\beta}$ ,  $\boldsymbol{\theta}$ , and  $\sigma^2$  [1]. That is, if  $\hat{\boldsymbol{\beta}}$ ,  $\hat{\boldsymbol{\theta}}$ , and  $\hat{\sigma}^2$  are the estimates of  $\boldsymbol{\beta}$ ,  $\boldsymbol{\theta}$ , and  $\sigma^2$ , respectively, then:

$$\hat{\boldsymbol{\beta}}, \hat{\boldsymbol{\theta}}, \hat{\sigma}^2 = \arg \max_{\boldsymbol{\beta}, \boldsymbol{\theta}, \sigma^2} \log P(\mathbf{y} | \mathbf{X}, \boldsymbol{\beta}, \boldsymbol{\theta}, \sigma^2).$$

Because

$$P(\mathbf{y}|\mathbf{X}) = P(\mathbf{y}|\mathbf{X}, \boldsymbol{\beta}, \boldsymbol{\theta}, \sigma^2) = \mathcal{N}(\mathbf{y} | \mathbf{H}\boldsymbol{\beta}, K(\mathbf{X}, \mathbf{X}|\boldsymbol{\theta}) + \sigma^2 \mathbf{I}_n),$$

the marginal log likelihood function is as follows:

$$\begin{aligned} \log P(\mathbf{y}|\mathbf{X}, \boldsymbol{\beta}, \boldsymbol{\theta}, \sigma^2) &= -\frac{1}{2}(\mathbf{y} - \mathbf{H}\boldsymbol{\beta})^T [K(\mathbf{X}, \mathbf{X}|\boldsymbol{\theta}) + \sigma^2 \mathbf{I}_n]^{-1} (\mathbf{y} - \mathbf{H}\boldsymbol{\beta}) \\ &\quad - \frac{n}{2} \log 2\pi - \frac{1}{2} \log |K(\mathbf{X}, \mathbf{X}|\boldsymbol{\theta}) + \sigma^2 \mathbf{I}_n|. \end{aligned}$$

where  $\mathbf{H}$  is the vector of explicit basis functions, and  $K(\mathbf{X}, \mathbf{X}|\boldsymbol{\theta})$  is the covariance function matrix (for more information, see “Gaussian Process Regression Models” on page 6-2).

To estimate the parameters, the software first computes  $\hat{\boldsymbol{\beta}}(\boldsymbol{\theta}, \sigma^2)$ , which maximizes the log likelihood function with respect to  $\boldsymbol{\beta}$  for given  $\boldsymbol{\theta}$  and  $\sigma^2$ . It then uses this estimate to compute the  $\boldsymbol{\beta}$ -profiled likelihood:

$$\log \left\{ P(\mathbf{y}|\mathbf{X}, \hat{\boldsymbol{\beta}}(\boldsymbol{\theta}, \sigma^2), \boldsymbol{\theta}, \sigma^2) \right\}.$$

The estimate of  $\boldsymbol{\beta}$  for given  $\boldsymbol{\theta}$ , and  $\sigma^2$  is

$$\hat{\boldsymbol{\beta}}(\boldsymbol{\theta}, \sigma^2) = \left[ \mathbf{H}^T [K(\mathbf{X}, \mathbf{X}|\boldsymbol{\theta}) + \sigma^2 \mathbf{I}_n]^{-1} \mathbf{H} \right]^{-1} \mathbf{H}^T [K(\mathbf{X}, \mathbf{X}|\boldsymbol{\theta}) + \sigma^2 \mathbf{I}_n]^{-1} \mathbf{y}.$$

Then, the  $\boldsymbol{\beta}$ -profiled log likelihood is given by

$$\begin{aligned} \log P(\mathbf{y}|\mathbf{X}, \hat{\boldsymbol{\beta}}(\boldsymbol{\theta}, \sigma^2), \boldsymbol{\theta}, \sigma^2) &= -\frac{1}{2}(\mathbf{y} - \mathbf{H}\hat{\boldsymbol{\beta}}(\boldsymbol{\theta}, \sigma^2))^T [K(\mathbf{X}, \mathbf{X}|\boldsymbol{\theta}) + \sigma^2 \mathbf{I}_n]^{-1} (\mathbf{y} - \mathbf{H}\hat{\boldsymbol{\beta}}(\boldsymbol{\theta}, \sigma^2)) \\ &\quad - \frac{n}{2} \log 2\pi - \frac{1}{2} \log |K(\mathbf{X}, \mathbf{X}|\boldsymbol{\theta}) + \sigma^2 \mathbf{I}_n| \end{aligned}$$

The software then maximizes the  $\boldsymbol{\beta}$ -profiled log-likelihood over  $\boldsymbol{\theta}$ , and  $\sigma^2$  to find their estimates.

## Prediction

Making probabilistic predictions from a GPR model with known parameters requires the density  $P(y_{new} | \mathbf{y}, \mathbf{X}, \mathbf{x}_{new})$ . Using the definition of conditional probabilities, one can write:

$$P(y_{new} | \mathbf{y}, \mathbf{X}, \mathbf{x}_{new}) = \frac{P(y_{new}, \mathbf{y} | \mathbf{X}, \mathbf{x}_{new})}{P(\mathbf{y} | \mathbf{X}, \mathbf{x}_{new})}.$$

To find the joint density in the numerator, it is necessary to introduce the latent variables  $f_{new}$  and  $\mathbf{f}$  corresponding to  $y_{new}$ , and  $\mathbf{y}$ , respectively. Then, it is possible to use the joint distribution for  $y_{new}$ ,  $\mathbf{y}$ ,  $f_{new}$ , and  $\mathbf{f}$  to compute  $P(y_{new}, \mathbf{y} | \mathbf{X}, \mathbf{x}_{new})$ :

$$\begin{aligned} P(y_{new}, \mathbf{y} | \mathbf{X}, \mathbf{x}_{new}) &= \iint P(y_{new}, \mathbf{y}, f_{new}, \mathbf{f} | \mathbf{X}, \mathbf{x}_{new}) d\mathbf{f} df_{new} \\ &= \iint P(y_{new}, \mathbf{y} | f_{new}, \mathbf{f}, \mathbf{X}, \mathbf{x}_{new}) P(f_{new}, \mathbf{f} | \mathbf{X}, \mathbf{x}_{new}) d\mathbf{f} df_{new}. \end{aligned}$$

Gaussian process models assume that each response  $y_i$  only depends on the corresponding latent variable  $f_i$  and the feature vector  $\mathbf{x}_i$ . Writing

$P(y_{new}, \mathbf{y} | f_{new}, \mathbf{f}, \mathbf{X}, \mathbf{x}_{new})$  as a product of conditional densities and based on this assumption produces:

$$P(y_{new}, \mathbf{y} | f_{new}, \mathbf{f}, \mathbf{X}, \mathbf{x}_{new}) = P(y_{new} | f_{new}, \mathbf{x}_{new}) \prod_{i=1}^n P(y_i | f(\mathbf{x}_i), \mathbf{x}_i).$$

After integrating with respect to  $y_{new}$ , the result only depends on  $\mathbf{f}$  and  $\mathbf{X}$ :

$$P(\mathbf{y} | \mathbf{f}, \mathbf{X}) = \prod_{i=1}^n P(y_i | f_i, \mathbf{x}_i) = \prod_{i=1}^n \mathcal{N}(y_i | \mathbf{h}(\mathbf{x}_i)^T \boldsymbol{\beta} + f_i, \sigma^2).$$

Hence,

$$P(y_{new}, \mathbf{y} | f_{new}, \mathbf{f}, \mathbf{X}, \mathbf{x}_{new}) = P(y_{new} | f_{new}, \mathbf{x}_{new}) P(\mathbf{y} | \mathbf{f}, \mathbf{X}).$$

Again using the definition of conditional probabilities,

$$P(f_{new}, \mathbf{f} | \mathbf{X}, \mathbf{x}_{new}) = P(f_{new} | \mathbf{f}, \mathbf{X}, \mathbf{x}_{new}) * P(\mathbf{f} | \mathbf{X}, \mathbf{x}_{new}),$$

it is possible to write  $P(y_{new}, \mathbf{y} | \mathbf{X}, \mathbf{x}_{new})$  as follows:

$$P(y_{new}, \mathbf{y} | \mathbf{X}, \mathbf{x}_{new}) = \int \int P(y_{new} | f_{new}, \mathbf{x}_{new}) P(\mathbf{y} | \mathbf{f}, \mathbf{X}) P(f_{new} | \mathbf{f}, \mathbf{X}, \mathbf{x}_{new}) P(\mathbf{f} | \mathbf{X}, \mathbf{x}_{new}) d\mathbf{f} df_{new}.$$

Using the facts that

$$P(\mathbf{f} | \mathbf{X}, \mathbf{x}_{new}) = P(\mathbf{f} | \mathbf{X})$$

and

$$P(\mathbf{y} | \mathbf{f}, \mathbf{X}) P(\mathbf{f} | \mathbf{X}) = P(\mathbf{y}, \mathbf{f} | \mathbf{X}) = P(\mathbf{f} | \mathbf{y}, \mathbf{X}) P(\mathbf{y} | \mathbf{X}),$$

one can rewrite  $P(y_{new}, \mathbf{y} | \mathbf{X}, \mathbf{x}_{new})$  as follows:

$$P(y_{new}, \mathbf{y} | \mathbf{X}, \mathbf{x}_{new}) = P(\mathbf{y} | \mathbf{X}) \int \int P(y_{new} | f_{new}, \mathbf{x}_{new}) P(\mathbf{f} | \mathbf{y}, \mathbf{X}) P(f_{new} | \mathbf{f}, \mathbf{X}, \mathbf{x}_{new}) d\mathbf{f} df_{new}.$$

It is also possible to show that

$$P(\mathbf{y} | \mathbf{X}, \mathbf{x}_{new}) = P(\mathbf{y} | \mathbf{X}).$$

Hence, the required density  $P(y_{new} | \mathbf{y}, \mathbf{X}, \mathbf{x}_{new})$  is:

$$\begin{aligned} P(y_{new} | \mathbf{y}, \mathbf{X}, \mathbf{x}_{new}) &= \frac{P(y_{new}, \mathbf{y} | \mathbf{X}, \mathbf{x}_{new})}{P(\mathbf{y} | \mathbf{X}, \mathbf{x}_{new})} = \frac{P(y_{new}, \mathbf{y} | \mathbf{X}, \mathbf{x}_{new})}{P(\mathbf{y} | \mathbf{X})} \\ &= \int \int \underbrace{P(y_{new} | f_{new}, \mathbf{x}_{new})}_{(1)} \underbrace{P(\mathbf{f} | \mathbf{y}, \mathbf{X})}_{(2)} \underbrace{P(f_{new} | \mathbf{f}, \mathbf{X}, \mathbf{x}_{new})}_{(3)} d\mathbf{f} df_{new}. \end{aligned}$$

It can be shown that

$$(1) \quad P(y_{new} | f_{new}, \mathbf{x}_{new}) = \mathcal{N}(y_{new} | \mathbf{h}(\mathbf{x}_{new})^T \boldsymbol{\beta} + f_{new}, \sigma_{new}^2)$$

$$(2) \quad P(\mathbf{f} | \mathbf{y}, \mathbf{X}) = \mathcal{N} \left( \mathbf{f} \left| \frac{1}{\sigma^2} \left( \frac{\mathbf{I}_n}{\sigma^2} + \mathbf{K}(\mathbf{X}, \mathbf{X}) \right)^{-1} (\mathbf{y} - \mathbf{H}\boldsymbol{\beta}), \left( \frac{\mathbf{I}_n}{\sigma^2} + \mathbf{K}(\mathbf{X}, \mathbf{X}) \right)^{-1} \right. \right)$$

$$(3) \quad P(f_{new} | \mathbf{f}, \mathbf{X}, \mathbf{x}_{new}) = \mathcal{N} \left( f_{new} \middle| \mathbf{K}(\mathbf{x}_{new}^T, \mathbf{X}) \mathbf{K}(\mathbf{X}, \mathbf{X})^{-1} \mathbf{f}, \Delta \right),$$

where  $\Delta = k(\mathbf{x}_{new}, \mathbf{x}_{new}) - \mathbf{K}(\mathbf{x}_{new}^T, \mathbf{X}) \mathbf{K}(\mathbf{X}, \mathbf{X})^{-1} \mathbf{K}(\mathbf{X}, \mathbf{x}_{new}^T)$ .

After the integration and required algebra, the density of the new response  $y_{new}$  at a new point  $\mathbf{x}_{new}$ , given  $\mathbf{y}$ ,  $\mathbf{X}$  is found as

$$P(y_{new} | \mathbf{y}, \mathbf{X}, \mathbf{x}_{new}) = \mathcal{N} \left( y_{new} \middle| \mathbf{h}(\mathbf{x}_{new})^T \boldsymbol{\beta} + \mu, \sigma_{new}^2 + \Sigma \right),$$

where

$$\mu = \mathbf{K}(\mathbf{x}_{new}^T, \mathbf{X}) \underbrace{\left( \mathbf{K}(\mathbf{X}, \mathbf{X}) + \sigma^2 \mathbf{I}_n \right)^{-1}}_{\boldsymbol{\alpha}} (\mathbf{y} - \mathbf{H}\boldsymbol{\beta})$$

and

$$\Sigma = k(\mathbf{x}_{new}, \mathbf{x}_{new}) - \mathbf{K}(\mathbf{x}_{new}^T, \mathbf{X}) \left( \mathbf{K}(\mathbf{X}, \mathbf{X}) + \sigma^2 \mathbf{I}_n \right)^{-1} \mathbf{K}(\mathbf{X}, \mathbf{x}_{new}^T).$$

The expected value of prediction  $y_{new}$  at a new point  $\mathbf{x}_{new}$  given  $\mathbf{y}$ ,  $\mathbf{X}$ , and parameters  $\boldsymbol{\beta}$ ,  $\boldsymbol{\theta}$ , and  $\sigma^2$  is

$$\begin{aligned} E(y_{new} | \mathbf{y}, \mathbf{X}, \mathbf{x}_{new}, \boldsymbol{\beta}, \boldsymbol{\theta}, \sigma^2) &= \mathbf{h}(\mathbf{x}_{new})^T \boldsymbol{\beta} + \mathbf{K}(\mathbf{x}_{new}^T, \mathbf{X} | \boldsymbol{\theta}) \boldsymbol{\alpha} \\ &= \mathbf{h}(\mathbf{x}_{new})^T \boldsymbol{\beta} + \sum_{i=1}^n \alpha_i k(\mathbf{x}_{new}, \mathbf{x}_i | \boldsymbol{\theta}), \end{aligned}$$

where

$$\boldsymbol{\alpha} = \left( \mathbf{K}(\mathbf{X}, \mathbf{X} | \boldsymbol{\theta}) + \sigma^2 \mathbf{I}_n \right)^{-1} (\mathbf{y} - \mathbf{H}\boldsymbol{\beta}).$$

## Computational Complexity of Exact Parameter Estimation and Prediction

Training a GPR model with the exact method (when `FitMethod` is 'Exact') requires the inversion of an  $n$ -by- $n$  kernel matrix  $\mathbf{K}(\mathbf{X}, \mathbf{X})$ . The memory requirement for this step scales as  $O(n^2)$  since  $\mathbf{K}(\mathbf{X}, \mathbf{X})$  must be stored in memory. One evaluation of  $\log P(y|\mathbf{X})$  scales as  $O(n^3)$ . Therefore, the computational complexity is  $O(k*n^3)$ , where  $k$  is the number of function evaluations needed for maximization and  $n$  is the number of observations.

Making predictions on new data involves the computation of  $\hat{\alpha}$ . If prediction intervals are desired, this step could also involve the computation and storage of the Cholesky factor of  $(\mathbf{K}(\mathbf{X}, \mathbf{X}) + \sigma^2 \mathbf{I}_n)$  for later use. The computational complexity of this step using the direct computation of  $\hat{\alpha}$  is  $O(n^3)$  and the memory requirement is  $O(n^2)$ .

Hence, for large  $n$ , estimation of parameters or computing predictions can be very expensive. The approximation methods usually involve rearranging the computation so as to avoid the inversion of an  $n$ -by- $n$  matrix. For the available approximation methods, please see the related links at the bottom of the page.

## References

- [1] Rasmussen, C. E. and C. K. I. Williams. *Gaussian Processes for Machine Learning*. MIT Press. Cambridge, Massachusetts, 2006.

## See Also

`fitrgp` | `predict`

## More About

- “Gaussian Process Regression Models” on page 6-2
- “Subset of Data Approximation for GPR Models” on page 6-18



- “Subset of Regressors Approximation for GPR Models” on page 6-20
- “Fully Independent Conditional Approximation for GPR Models” on page 6-26
- “Block Coordinate Descent Approximation for GPR Models” on page 6-30

## Subset of Data Approximation for GPR Models

Training a GPR model with the exact method (when `FitMethod` is 'Exact') requires the inversion of an  $n$ -by- $n$  matrix. Therefore, the computational complexity is  $O(k*n^3)$ ,

where  $k$  is the number of function evaluations required for estimating  $\beta$ ,  $\theta$ , and  $\sigma^2$ , and  $n$  is the number of observations. For large  $n$ , estimation of parameters or computing predictions can be very expensive.

One simple way to solve the computational complexity problem with large data sets is to select  $m < n$  observations out of  $n$  and then apply exact GPR model on page 6-11 to these  $m$  points to estimate  $\beta$ ,  $\theta$ , and  $\sigma^2$  while ignoring the other  $(n - m)$  points. This smaller subset is known as the *active set*. And this approximation method is called the Subset of Data (SD) method.

The computational complexity when using SD method is  $O(km^3)$ , where  $k$  is the number of function evaluations and  $m$  is the active set size. The storage requirements are  $O(m^2)$

since only a part of the full kernel matrix  $K(\mathbf{X}, \mathbf{X}|\theta)$  needs to be stored in memory.

You can specify the SD method for parameter estimation by using the 'FitMethod', 'sd' name-value pair argument in the call to `fitrgp`. To specify the SD method for prediction, use the 'PredictMethod', 'sd' name-value pair argument.

For estimating parameters using the exact GPR model, see parameter estimation using the exact GPR method on page 6-11. For making predictions using the exact GPR model, see prediction using the exact GPR method on page 6-13.

### See Also

`fitrgp` | `predict`

### More About

- “Gaussian Process Regression Models” on page 6-2
- “Exact GPR Method” on page 6-11
- “Subset of Regressors Approximation for GPR Models” on page 6-20
- “Fully Independent Conditional Approximation for GPR Models” on page 6-26

- “Block Coordinate Descent Approximation for GPR Models” on page 6-30

## Subset of Regressors Approximation for GPR Models

### In this section...

“Approximating the Kernel Function” on page 6-20

“Parameter Estimation” on page 6-21

“Prediction” on page 6-23

“Predictive Variance Problem” on page 6-24

The subset of regressors (SR) approximation method consists of replacing the kernel function  $k(\mathbf{x}, \mathbf{x}_r | \boldsymbol{\theta})$  in the exact GPR method on page 6-11 by its approximation

$\hat{k}_{SR}(\mathbf{x}, \mathbf{x}_r | \boldsymbol{\theta}, \mathbf{A})$ , given the active set  $\mathbf{A} \subset \mathbf{N} = \{1, 2, \dots, n\}$ . You can specify the SR method for parameter estimation by using the 'FitMethod', 'sr' name-value pair argument in the call to `fitrgp`. For prediction using SR, you can use the 'PredictMethod', 'sr' name-value pair argument in the call to `fitrgp`.

### Approximating the Kernel Function

For the exact GPR model on page 6-13, the expected prediction in GPR depends on the

set of  $\mathbf{N}$  functions  $S_{\mathbf{N}} = \{k(\mathbf{x}, \mathbf{x}_i | \boldsymbol{\theta}), i = 1, 2, \dots, n\}$ , where  $\mathbf{N} = \{1, 2, \dots, n\}$  is the set of indices of all observations, and  $n$  is the total number of observations. The idea is to

approximate the span of these functions by a smaller set of functions,  $S_{\mathbf{A}}$ , where

$\mathbf{A} \subset \mathbf{N} = \{1, 2, \dots, n\}$  is the subset of indices of points selected to be in the active set.

Consider  $S_{\mathbf{A}} = \{k(\mathbf{x}, \mathbf{x}_j | \boldsymbol{\theta}), j \in \mathbf{A}\}$ . The aim is to approximate the elements of  $S_{\mathbf{N}}$  as linear combinations of the elements of  $S_{\mathbf{A}}$ .

Suppose the approximation to  $k(\mathbf{x}, \mathbf{x}_r | \boldsymbol{\theta})$  using the functions in  $S_{\mathbf{A}}$  is as follows:

$$\hat{k}(\mathbf{x}, \mathbf{x}_r | \boldsymbol{\theta}) = \sum_{j \in \mathbf{A}} c_{j_r} k(\mathbf{x}, \mathbf{x}_j | \boldsymbol{\theta}),$$

where  $c_{jr} \in \mathbb{R}$  are the coefficients of the linear combination for approximating  $k(\mathbf{x}, \mathbf{x}_r | \boldsymbol{\theta})$ . Suppose  $\mathbf{C}$  is the matrix that contains all the coefficients  $c_{jr}$ . Then,  $\mathbf{C}$ , is a  $|\mathbb{A}| \times n$  matrix such that  $\mathbf{C}(j, r) = c_{jr}$ . The software finds the best approximation to the elements of  $\mathbb{S}_N$  using the active set  $\mathbb{A} \subset \mathbb{N} = \{1, 2, \dots, n\}$  by minimizing the error function

$$E(\mathbb{A}, \mathbf{C}) = \sum_{r=1}^n \left\| k(\mathbf{x}, \mathbf{x}_r | \boldsymbol{\theta}) - k(\mathbf{x}, \mathbf{x}_r | \boldsymbol{\theta}) \right\|_{\mathbb{H}}^2,$$

where  $\mathbb{H}$  is the Reproducing Kernel Hilbert Spaces (RKHS) associated with the kernel function  $k$  [1], [2].

The coefficient matrix that minimizes  $E(\mathbb{A}, \mathbf{C})$  is

$$\hat{\mathbf{C}}_{\mathbb{A}} = \mathbf{K}(\mathbf{X}_{\mathbb{A}}, \mathbf{X}_{\mathbb{A}} | \boldsymbol{\theta})^{-1} \mathbf{K}(\mathbf{X}_{\mathbb{A}}, \mathbf{X} | \boldsymbol{\theta}),$$

and an approximation to the kernel function using the elements in the active set

$\mathbb{A} \subset \mathbb{N} = \{1, 2, \dots, n\}$  is

$$\hat{k}(\mathbf{x}, \mathbf{x}_r | \boldsymbol{\theta}) = \sum_{j \in \mathbb{A}} c_{jr} k(\mathbf{x}, \mathbf{x}_j | \boldsymbol{\theta}) = \mathbf{K}(\mathbf{x}^T, \mathbf{X}_{\mathbb{A}} | \boldsymbol{\theta}) \mathbf{C}(:, r).$$

The SR approximation to the kernel function using the active set  $\mathbb{A} \subset \mathbb{N} = \{1, 2, \dots, n\}$  is defined as:

$$\hat{k}_{SR}(\mathbf{x}, \mathbf{x}_r | \boldsymbol{\theta}, \mathbb{A}) = \mathbf{K}(\mathbf{x}^T, \mathbf{X}_{\mathbb{A}} | \boldsymbol{\theta}) \hat{\mathbf{C}}_{\mathbb{A}}(:, r) = \mathbf{K}(\mathbf{x}^T, \mathbf{X}_{\mathbb{A}} | \boldsymbol{\theta}) \mathbf{K}(\mathbf{X}_{\mathbb{A}}, \mathbf{X}_{\mathbb{A}} | \boldsymbol{\theta})^{-1} \mathbf{K}(\mathbf{X}_{\mathbb{A}}, \mathbf{x}_r^T | \boldsymbol{\theta})$$

and the SR approximation to  $K(\mathbf{X}, \mathbf{X} | \boldsymbol{\theta})$  is:

$$\hat{\mathbf{K}}_{SR}(\mathbf{X}, \mathbf{X} | \boldsymbol{\theta}, \mathbb{A}) = \mathbf{K}(\mathbf{X}, \mathbf{X}_{\mathbb{A}} | \boldsymbol{\theta}) \mathbf{K}(\mathbf{X}_{\mathbb{A}}, \mathbf{X}_{\mathbb{A}} | \boldsymbol{\theta})^{-1} \mathbf{K}(\mathbf{X}_{\mathbb{A}}, \mathbf{X} | \boldsymbol{\theta}).$$

## Parameter Estimation

Replacing  $K(\mathbf{X}, \mathbf{X} | \boldsymbol{\theta})$  by  $\hat{\mathbf{K}}_{SR}(\mathbf{X}, \mathbf{X} | \boldsymbol{\theta}, \mathbb{A})$  in the marginal log likelihood function produces its SR approximation:

$$\begin{aligned} \log P_{SR}(\mathbf{y} | \mathbf{X}, \boldsymbol{\beta}, \boldsymbol{\theta}, \sigma^2, \mathbb{A}) &= -\frac{1}{2}(\mathbf{y} - \mathbf{H}\boldsymbol{\beta})^T \left[ \widehat{\mathbf{K}}_{SR}(\mathbf{X}, \mathbf{X} | \boldsymbol{\theta}, \mathbb{A}) + \sigma^2 \mathbf{I}_n \right]^{-1} (\mathbf{y} - \mathbf{H}\boldsymbol{\beta}) \\ &\quad - \frac{N}{2} \log 2\pi - \frac{1}{2} \log \left| \widehat{\mathbf{K}}_{SR}(\mathbf{X}, \mathbf{X} | \boldsymbol{\theta}, \mathbb{A}) + \sigma^2 \mathbf{I}_n \right| \end{aligned}$$

As in the exact method on page 6-11, the software estimates the parameters by first computing  $\hat{\boldsymbol{\beta}}(\boldsymbol{\theta}, \sigma^2)$ , the optimal estimate of  $\boldsymbol{\beta}$ , given  $\boldsymbol{\theta}$  and  $\sigma^2$ . Then it estimates  $\boldsymbol{\theta}$ , and  $\sigma^2$  using the  $\boldsymbol{\beta}$ -profiled marginal log likelihood. The SR estimate to  $\boldsymbol{\beta}$  for given  $\boldsymbol{\theta}$ , and  $\sigma^2$  is:

$$\hat{\boldsymbol{\beta}}_{SR}(\boldsymbol{\theta}, \sigma^2, \mathbb{A}) = \left[ \underbrace{\mathbf{H}^T \left[ \widehat{\mathbf{K}}_{SR}(\mathbf{X}, \mathbf{X} | \boldsymbol{\theta}, \mathbb{A}) + \sigma^2 \mathbf{I}_n \right]^{-1} \mathbf{H}}_* \right]^{-1} \underbrace{\mathbf{H}^T \left[ \widehat{\mathbf{K}}_{SR}(\mathbf{X}, \mathbf{X} | \boldsymbol{\theta}, \mathbb{A}) + \sigma^2 \mathbf{I}_n \right]^{-1} \mathbf{y}}_{**}$$

where

$$\begin{aligned} \left[ \widehat{\mathbf{K}}_{SR}(\mathbf{X}, \mathbf{X} | \boldsymbol{\theta}, \mathbb{A}) + \sigma^2 \mathbf{I}_n \right]^{-1} &= \frac{\mathbf{I}_N}{\sigma^2} - \frac{\mathbf{K}(\mathbf{X}, \mathbf{X}_\mathbb{A} | \boldsymbol{\theta})}{\sigma^2} \mathbf{A}_\mathbb{A}^{-1} \frac{\mathbf{K}(\mathbf{X}_\mathbb{A}, \mathbf{X} | \boldsymbol{\theta})}{\sigma^2}, \\ \mathbf{A}_\mathbb{A} &= \mathbf{K}(\mathbf{X}_\mathbb{A}, \mathbf{X}_\mathbb{A} | \boldsymbol{\theta}) + \frac{\mathbf{K}(\mathbf{X}_\mathbb{A}, \mathbf{X} | \boldsymbol{\theta}) \mathbf{K}(\mathbf{X}, \mathbf{X}_\mathbb{A} | \boldsymbol{\theta})}{\sigma^2}, \\ * &= \frac{\mathbf{H}^T \mathbf{H}}{\sigma^2} - \frac{\mathbf{H}^T \mathbf{K}(\mathbf{X}, \mathbf{X}_\mathbb{A} | \boldsymbol{\theta})}{\sigma^2} \mathbf{A}_\mathbb{A}^{-1} \frac{\mathbf{K}(\mathbf{X}_\mathbb{A}, \mathbf{X} | \boldsymbol{\theta}) \mathbf{H}}{\sigma^2}, \\ ** &= \frac{\mathbf{H}^T \mathbf{y}}{\sigma^2} - \frac{\mathbf{H}^T \mathbf{K}(\mathbf{X}, \mathbf{X}_\mathbb{A} | \boldsymbol{\theta})}{\sigma^2} \mathbf{A}_\mathbb{A}^{-1} \frac{\mathbf{K}(\mathbf{X}_\mathbb{A}, \mathbf{X} | \boldsymbol{\theta}) \mathbf{y}}{\sigma^2}. \end{aligned}$$

And the SR approximation to the  $\boldsymbol{\beta}$ -profiled marginal log likelihood is:

$$\begin{aligned} \log P_{SR}(\mathbf{y} | \mathbf{X}, \hat{\boldsymbol{\beta}}_{SR}(\boldsymbol{\theta}, \sigma^2, \mathbb{A}), \boldsymbol{\theta}, \sigma^2, \mathbb{A}) &= \\ &= -\frac{1}{2}(\mathbf{y} - \mathbf{H}\hat{\boldsymbol{\beta}}_{SR}(\boldsymbol{\theta}, \sigma^2, \mathbb{A}))^T \left[ \widehat{\mathbf{K}}_{SR}(\mathbf{X}, \mathbf{X} | \boldsymbol{\theta}, \mathbb{A}) + \sigma^2 \mathbf{I}_n \right]^{-1} (\mathbf{y} - \mathbf{H}\hat{\boldsymbol{\beta}}_{SR}(\boldsymbol{\theta}, \sigma^2, \mathbb{A})) \\ &\quad - \frac{N}{2} \log 2\pi - \frac{1}{2} \log \left| \widehat{\mathbf{K}}_{SR}(\mathbf{X}, \mathbf{X} | \boldsymbol{\theta}, \mathbb{A}) + \sigma^2 \mathbf{I}_n \right|. \end{aligned}$$

## Prediction

The SR approximation to the distribution of  $y_{new}$  given  $\mathbf{y}$ ,  $\mathbf{X}$ ,  $\mathbf{x}_{new}$  is

$$P(y_{new} | \mathbf{y}, \mathbf{X}, \mathbf{x}_{new}) = \mathcal{N} \left( y_{new} \middle| \mathbf{h}(\mathbf{x}_{new})^T \boldsymbol{\beta} + \boldsymbol{\mu}_{SR}, \sigma_{new}^2 + \boldsymbol{\Sigma}_{SR} \right),$$

where  $\boldsymbol{\mu}_{SR}$  and  $\boldsymbol{\Sigma}_{SR}$  are the SR approximations to  $\boldsymbol{\mu}$  and  $\boldsymbol{\Sigma}$  shown in prediction using the exact GPR method on page 6-13.

$\boldsymbol{\mu}_{SR}$  and  $\boldsymbol{\Sigma}_{SR}$  are obtained by replacing  $k(\mathbf{x}, \mathbf{x}_r | \boldsymbol{\theta})$  by its SR approximation

$\hat{k}_{SR}(\mathbf{x}, \mathbf{x}_r | \boldsymbol{\theta}, \mathbb{A})$  in  $\boldsymbol{\mu}$  and  $\boldsymbol{\Sigma}$ , respectively.

That is,

$$\boldsymbol{\mu}_{SR} = \underbrace{\widehat{\mathbf{K}}_{SR}(\mathbf{x}_{new}^T, \mathbf{X} | \boldsymbol{\theta}, \mathbb{A})}_{(1)} \underbrace{\left( \widehat{\mathbf{K}}_{SR}(\mathbf{X}, \mathbf{X} | \boldsymbol{\theta}, \mathbb{A}) + \sigma^2 \mathbf{I}_N \right)^{-1}}_{(2)} (\mathbf{y} - \mathbf{H}\boldsymbol{\beta}).$$

Since

$$(1) = \mathbf{K}(\mathbf{x}_{new}^T, \mathbf{X}_A | \boldsymbol{\theta}) \mathbf{K}(\mathbf{X}_A, \mathbf{X}_A | \boldsymbol{\theta})^{-1} \mathbf{K}(\mathbf{X}_A, \mathbf{X} | \boldsymbol{\theta}),$$

$$(2) = \frac{\mathbf{I}_N}{\sigma^2} - \frac{\mathbf{K}(\mathbf{X}, \mathbf{X}_A | \boldsymbol{\theta})}{\sigma^2} \left[ \mathbf{K}(\mathbf{X}_A, \mathbf{X}_A | \boldsymbol{\theta}) + \frac{\mathbf{K}(\mathbf{X}_A, \mathbf{X} | \boldsymbol{\theta}) \mathbf{K}(\mathbf{X}, \mathbf{X}_A | \boldsymbol{\theta})}{\sigma^2} \right]^{-1} \frac{\mathbf{K}(\mathbf{X}_A, \mathbf{X} | \boldsymbol{\theta})}{\sigma^2},$$

and from the fact that  $\mathbf{I}_N - \mathbf{B}(\mathbf{A} + \mathbf{B})^{-1} = \mathbf{A}(\mathbf{A} + \mathbf{B})^{-1}$ ,  $\boldsymbol{\mu}_{SR}$  can be written as

$$\boldsymbol{\mu}_{SR} = \mathbf{K}(\mathbf{x}_{new}^T, \mathbf{X}_A | \boldsymbol{\theta}) \left[ \mathbf{K}(\mathbf{X}_A, \mathbf{X}_A | \boldsymbol{\theta}) + \frac{\mathbf{K}(\mathbf{X}_A, \mathbf{X} | \boldsymbol{\theta}) \mathbf{K}(\mathbf{X}, \mathbf{X}_A | \boldsymbol{\theta})}{\sigma^2} \right]^{-1} \frac{\mathbf{K}(\mathbf{X}_A, \mathbf{X} | \boldsymbol{\theta})}{\sigma^2} (\mathbf{y} - \mathbf{H}\boldsymbol{\beta}).$$

Similarly,  $\boldsymbol{\Sigma}_{SR}$  is derived as follows:

$$\boldsymbol{\Sigma}_{SR} = \underbrace{\hat{k}_{SR}(\mathbf{x}_{new}, \mathbf{x}_{new} | \boldsymbol{\theta}, \mathbb{A})}_{*} - \underbrace{\widehat{\mathbf{K}}_{SR}(\mathbf{x}_{new}^T, \mathbf{X} | \boldsymbol{\theta}, \mathbb{A})}_{**} \underbrace{\left( \widehat{\mathbf{K}}_{SR}(\mathbf{X}, \mathbf{X} | \boldsymbol{\theta}, \mathbb{A}) + \sigma^2 \mathbf{I}_N \right)^{-1}}_{***} \underbrace{\widehat{\mathbf{K}}_{SR}(\mathbf{X}, \mathbf{x}_{new}^T | \boldsymbol{\theta}, \mathbb{A})}_{****}.$$

Because

$$* = \mathbf{K}(\mathbf{x}_{new}^T, \mathbf{X}_A | \boldsymbol{\theta}) \mathbf{K}(\mathbf{X}_A, \mathbf{X}_A | \boldsymbol{\theta})^{-1} \mathbf{K}(\mathbf{X}_A, \mathbf{x}_{new}^T | \boldsymbol{\theta}),$$

$$** = \mathbf{K}(\mathbf{x}_{new}^T, \mathbf{X}_A | \boldsymbol{\theta}) \mathbf{K}(\mathbf{X}_A, \mathbf{X}_A | \boldsymbol{\theta})^{-1} \mathbf{K}(\mathbf{X}_A, \mathbf{X} | \boldsymbol{\theta}),$$

\*\*\* = (2) in the equation of  $\mu_{SR}$ ,

$$**** = \mathbf{K}(\mathbf{X}, \mathbf{X}_A | \boldsymbol{\theta}) \mathbf{K}(\mathbf{X}_A, \mathbf{X}_A | \boldsymbol{\theta})^{-1} \mathbf{K}(\mathbf{X}_A, \mathbf{x}_{new}^T | \boldsymbol{\theta}),$$

$\Sigma_{SR}$  is found as follows:

$$\Sigma_{SR} = \mathbf{K}(\mathbf{x}_{new}^T, \mathbf{X}_A | \boldsymbol{\theta}) \left[ \mathbf{K}(\mathbf{X}_A, \mathbf{X}_A | \boldsymbol{\theta}) + \frac{\mathbf{K}(\mathbf{X}_A, \mathbf{X} | \boldsymbol{\theta}) \mathbf{K}(\mathbf{X}, \mathbf{X}_A | \boldsymbol{\theta})}{\sigma^2} \right]^{-1} \mathbf{K}(\mathbf{X}_A, \mathbf{x}_{new}^T | \boldsymbol{\theta}).$$

## Predictive Variance Problem

One of the disadvantages of the SR method is that it can give unreasonably small predictive variances when making predictions in a region far away from the chosen

active set  $\mathbb{A} \subset \mathbb{N} = \{1, 2, \dots, n\}$ . Consider making a prediction at a new point  $\mathbf{x}_{new}$  that is

far away from the training set  $\mathbf{X}$ . In other words, assume that  $\mathbf{K}(\mathbf{x}_{new}^T, \mathbf{X} | \boldsymbol{\theta}) \approx \mathbf{0}$ .

For exact GPR, the posterior distribution of  $f_{new}$  given  $\mathbf{y}$ ,  $\mathbf{X}$  and  $\mathbf{x}_{new}$  would be

Normal with mean  $\boldsymbol{\mu} = \mathbf{0}$  and variance  $\boldsymbol{\Sigma} = k(\mathbf{x}_{new}, \mathbf{x}_{new} | \boldsymbol{\theta})$ . This value is correct in the

sense that, if  $\mathbf{x}_{new}$  is far from  $\mathbf{X}$ , then the data  $(\mathbf{X}, \mathbf{y})$  does not supply any new

information about  $f_{new}$  and so the posterior distribution of  $f_{new}$  given  $\mathbf{y}$ ,  $\mathbf{X}$ , and  $\mathbf{x}_{new}$

should reduce to the prior distribution  $f_{new}$  given  $\mathbf{x}_{new}$ , which is a Normal distribution

with mean 0 and variance  $k(\mathbf{x}_{new}, \mathbf{x}_{new} | \boldsymbol{\theta})$ .



For the SR approximation, if  $\mathbf{x}_{new}$  is far away from  $\mathbf{X}$  (and hence also far away from  $\mathbf{X}_A$ ), then  $\boldsymbol{\mu}_{SR} = 0$  and  $\boldsymbol{\Sigma}_{SR} = 0$ . Thus in this extreme case,  $\boldsymbol{\mu}_{SR}$  agrees with  $\boldsymbol{\mu}$  from exact GPR, but  $\boldsymbol{\Sigma}_{SR}$  is unreasonably small compared to  $\boldsymbol{\Sigma}$  from exact GPR.

The fully independent conditional approximation method on page 6-26 can help avoid this problem.

## References

- [1] Rasmussen, C. E. and C. K. I. Williams. *Gaussian Processes for Machine Learning*. MIT Press. Cambridge, Massachusetts, 2006.
- [2] Smola, A. J. and B. Schölkopf. *Sparse greedy matrix approximation for machine learning*. In Proceedings of the Seventeenth International Conference on Machine Learning, 2000.

## See Also

`fitrgp` | `predict`

## More About

- “Gaussian Process Regression Models” on page 6-2
- “Exact GPR Method” on page 6-11
- “Subset of Data Approximation for GPR Models” on page 6-18
- “Fully Independent Conditional Approximation for GPR Models” on page 6-26
- “Block Coordinate Descent Approximation for GPR Models” on page 6-30

## Fully Independent Conditional Approximation for GPR Models

### In this section...

“Approximating the Kernel Function” on page 6-26

“Parameter Estimation” on page 6-27

“Prediction” on page 6-28

The fully independent conditional (FIC) approximation[1] is a way of systematically approximating the true GPR kernel function in a way that avoids the predictive variance problem of the SR approximation on page 6-24 while still maintaining a valid Gaussian process. You can specify the FIC method for parameter estimation by using the 'FitMethod', 'fic' name-value pair argument in the call to `fitrgp`. For prediction using FIC, you can use the 'PredictMethod', 'fic' name-value pair argument in the call to `fitrgp`.

### Approximating the Kernel Function

The FIC approximation to  $k(\mathbf{x}_i, \mathbf{x}_j | \boldsymbol{\theta})$  for active set  $\mathbb{A} \subset \mathbb{N} = \{1, 2, \dots, n\}$  is given by:

$$\hat{k}_{FIC}(\mathbf{x}_i, \mathbf{x}_j | \boldsymbol{\theta}, \mathbb{A}) = \hat{k}_{SR}(\mathbf{x}_i, \mathbf{x}_j | \boldsymbol{\theta}, \mathbb{A}) + \delta_{ij} \left( k(\mathbf{x}_i, \mathbf{x}_j | \boldsymbol{\theta}) - \hat{k}_{SR}(\mathbf{x}_i, \mathbf{x}_j | \boldsymbol{\theta}, \mathbb{A}) \right),$$

$$\delta_{ij} = \begin{cases} 1, & \text{if } i = j, \\ 0 & \text{if } i \neq j. \end{cases}$$

That is, the FIC approximation is equal to the SR approximation if  $i \neq j$ . For  $i = j$ , the software uses the exact kernel value rather than an approximation. Define an  $n$ -by- $n$

diagonal matrix  $\boldsymbol{\Omega}(\mathbf{X} | \boldsymbol{\theta}, \mathbb{A})$  as follows:

$$\begin{aligned} [\boldsymbol{\Omega}(\mathbf{X} | \boldsymbol{\theta}, \mathbb{A})]_{ij} &= \delta_{ij} \left( k(\mathbf{x}_i, \mathbf{x}_j | \boldsymbol{\theta}) - k_{SR}(\mathbf{x}_i, \mathbf{x}_j | \boldsymbol{\theta}, \mathbb{A}) \right) \\ &= \begin{cases} k(\mathbf{x}_i, \mathbf{x}_j | \boldsymbol{\theta}) - k_{SR}(\mathbf{x}_i, \mathbf{x}_j | \boldsymbol{\theta}, \mathbb{A}) & \text{if } i = j, \\ 0 & \text{if } i \neq j. \end{cases} \end{aligned}$$

The FIC approximation to  $\mathbf{K}(\mathbf{X}, \mathbf{X} | \boldsymbol{\theta})$  is then given by:

$$\begin{aligned}\widehat{\mathbf{K}}_{FIC}(\mathbf{X}, \mathbf{X} | \boldsymbol{\theta}, \mathbb{A}) &= \widehat{\mathbf{K}}_{SR}(\mathbf{X}, \mathbf{X} | \boldsymbol{\theta}, \mathbb{A}) + \boldsymbol{\Omega}(\mathbf{X} | \boldsymbol{\theta}, \mathbb{A}) \\ &= \mathbf{K}(\mathbf{X}, \mathbf{X}_A | \boldsymbol{\theta}) \mathbf{K}(\mathbf{X}_A, \mathbf{X}_A | \boldsymbol{\theta})^{-1} \mathbf{K}(\mathbf{X}_A, \mathbf{X} | \boldsymbol{\theta}) + \boldsymbol{\Omega}(\mathbf{X} | \boldsymbol{\theta}, \mathbb{A}).\end{aligned}$$

## Parameter Estimation

Replacing  $\mathbf{K}(\mathbf{X}, \mathbf{X} | \boldsymbol{\theta})$  by  $\widehat{\mathbf{K}}_{FIC}(\mathbf{X}, \mathbf{X} | \boldsymbol{\theta}, \mathbb{A})$  in the marginal log likelihood function produces its FIC approximation:

$$\begin{aligned}\log P_{FIC}(\mathbf{y} | \mathbf{X}, \boldsymbol{\beta}, \boldsymbol{\theta}, \sigma^2, \mathbb{A}) &= -\frac{1}{2}(\mathbf{y} - \mathbf{H}\boldsymbol{\beta})^T \left[ \widehat{\mathbf{K}}_{FIC}(\mathbf{X}, \mathbf{X} | \boldsymbol{\theta}, \mathbb{A}) + \sigma^2 \mathbf{I}_n \right]^{-1} (\mathbf{y} - \mathbf{H}\boldsymbol{\beta}) \\ &\quad - \frac{N}{2} \log 2\pi - \frac{1}{2} \log \left| \widehat{\mathbf{K}}_{FIC}(\mathbf{X}, \mathbf{X} | \boldsymbol{\theta}, \mathbb{A}) + \sigma^2 \mathbf{I}_n \right|.\end{aligned}$$

As in the exact method on page 6-11, the software estimates the parameters by first

computing  $\widehat{\boldsymbol{\beta}}(\boldsymbol{\theta}, \sigma^2)$ , the optimal estimate of  $\boldsymbol{\beta}$ , given  $\boldsymbol{\theta}$  and  $\sigma^2$ . Then it estimates  $\boldsymbol{\theta}$ , and  $\sigma^2$  using the  $\boldsymbol{\beta}$ -profiled marginal log likelihood. The FIC estimate to  $\boldsymbol{\beta}$  for given  $\boldsymbol{\theta}$ , and  $\sigma^2$  is

$$\widehat{\boldsymbol{\beta}}_{FIC}(\boldsymbol{\theta}, \sigma^2, \mathbb{A}) = \left[ \underbrace{\mathbf{H}^T \left( \widehat{\mathbf{K}}_{FIC}(\mathbf{X}, \mathbf{X} | \boldsymbol{\theta}, \mathbb{A}) + \sigma^2 \mathbf{I}_n \right)^{-1} \mathbf{H}}_* \right]^{-1} \underbrace{\mathbf{H}^T \left( \widehat{\mathbf{K}}_{FIC}(\mathbf{X}, \mathbf{X} | \boldsymbol{\theta}, \mathbb{A}) + \sigma^2 \mathbf{I}_n \right)^{-1} \mathbf{y}}_{**},$$

$$* = \mathbf{H}^T \boldsymbol{\Lambda}(\boldsymbol{\theta}, \sigma^2, \mathbb{A})^{-1} \mathbf{H} - \mathbf{H}^T \boldsymbol{\Lambda}(\boldsymbol{\theta}, \sigma^2, \mathbb{A})^{-1} \mathbf{K}(\mathbf{X}, \mathbf{X}_A | \boldsymbol{\theta}) \mathbf{B}_A^{-1} \mathbf{K}(\mathbf{X}_A, \mathbf{X} | \boldsymbol{\theta}) \boldsymbol{\Lambda}(\boldsymbol{\theta}, \sigma^2, \mathbb{A})^{-1} \mathbf{H},$$

$$** = \mathbf{H}^T \boldsymbol{\Lambda}(\boldsymbol{\theta}, \sigma^2, \mathbb{A})^{-1} \mathbf{y} - \mathbf{H}^T \boldsymbol{\Lambda}(\boldsymbol{\theta}, \sigma^2, \mathbb{A})^{-1} \mathbf{K}(\mathbf{X}, \mathbf{X}_A | \boldsymbol{\theta}) \mathbf{B}_A^{-1} \mathbf{K}(\mathbf{X}_A, \mathbf{X} | \boldsymbol{\theta}) \boldsymbol{\Lambda}(\boldsymbol{\theta}, \sigma^2, \mathbb{A})^{-1} \mathbf{y},$$

$$\mathbf{B}_A = \mathbf{K}(\mathbf{X}_A, \mathbf{X}_A | \boldsymbol{\theta}) + \mathbf{K}(\mathbf{X}_A, \mathbf{X} | \boldsymbol{\theta}) \boldsymbol{\Lambda}(\boldsymbol{\theta}, \sigma^2, \mathbb{A})^{-1} \mathbf{K}(\mathbf{X}, \mathbf{X}_A | \boldsymbol{\theta}),$$

$$\boldsymbol{\Lambda}(\boldsymbol{\theta}, \sigma^2, \mathbb{A}) = \boldsymbol{\Omega}(\mathbf{X} | \boldsymbol{\theta}, \mathbb{A}) + \sigma^2 \mathbf{I}_n.$$

Using  $\widehat{\boldsymbol{\beta}}_{FIC}(\boldsymbol{\theta}, \sigma^2, \mathbb{A})$ , the  $\boldsymbol{\beta}$ -profiled marginal log likelihood for FIC approximation is:

$$\begin{aligned} \log P_{FIC}(\mathbf{y} | \mathbf{X}, \hat{\boldsymbol{\beta}}_{FIC}(\boldsymbol{\theta}, \sigma^2, \mathbb{A}), \boldsymbol{\theta}, \sigma^2, \mathbb{A}) = \\ -\frac{1}{2}(\mathbf{y} - \mathbf{H}\hat{\boldsymbol{\beta}}_{FIC}(\boldsymbol{\theta}, \sigma^2, \mathbb{A}))^T (\widehat{\mathbf{K}}_{FIC}(\mathbf{X}, \mathbf{X} | \boldsymbol{\theta}, \mathbb{A}) + \sigma^2 \mathbf{I}_N)^{-1} (\mathbf{y} - \mathbf{H}\hat{\boldsymbol{\beta}}_{FIC}(\boldsymbol{\theta}, \sigma^2, \mathbb{A})) \\ -\frac{N}{2} \log 2\pi - \frac{1}{2} \log |\widehat{\mathbf{K}}_{FIC}(\mathbf{X}, \mathbf{X} | \boldsymbol{\theta}, \mathbb{A}) + \sigma^2 \mathbf{I}_N|, \end{aligned}$$

where

$$\begin{aligned} & (\widehat{\mathbf{K}}_{FIC}(\mathbf{X}, \mathbf{X} | \boldsymbol{\theta}, \mathbb{A}) + \sigma^2 \mathbf{I}_N)^{-1} \\ &= \boldsymbol{\Lambda}(\boldsymbol{\theta}, \sigma^2, \mathbb{A})^{-1} - \boldsymbol{\Lambda}(\boldsymbol{\theta}, \sigma^2, \mathbb{A})^{-1} \mathbf{K}(\mathbf{X}, \mathbf{X}_A | \boldsymbol{\theta}) \mathbf{B}_A^{-1} \mathbf{K}(\mathbf{X}_A, \mathbf{X} | \boldsymbol{\theta}) \boldsymbol{\Lambda}(\boldsymbol{\theta}, \sigma^2, \mathbb{A})^{-1}, \\ \log |\widehat{\mathbf{K}}_{FIC}(\mathbf{X}, \mathbf{X} | \boldsymbol{\theta}, \mathbb{A}) + \sigma^2 \mathbf{I}_N| &= \log |\boldsymbol{\Lambda}(\boldsymbol{\theta}, \sigma^2, \mathbb{A})| + \log |\mathbf{B}_A| - \log |\mathbf{K}(\mathbf{X}_A, \mathbf{X}_A | \boldsymbol{\theta})|. \end{aligned}$$

## Prediction

The FIC approximation to the distribution of  $y_{new}$  given  $\mathbf{y}$ ,  $\mathbf{X}$ ,  $\mathbf{x}_{new}$  is

$$P(y_{new} | \mathbf{y}, \mathbf{X}, \mathbf{x}_{new}) = \mathcal{N}\left(y_{new} \mid \mathbf{h}(\mathbf{x}_{new})^T \boldsymbol{\beta} + \boldsymbol{\mu}_{FIC}, \sigma_{new}^2 + \boldsymbol{\Sigma}_{FIC}\right),$$

where  $\boldsymbol{\mu}_{FIC}$  and  $\boldsymbol{\Sigma}_{FIC}$  are the FIC approximations to  $\boldsymbol{\mu}$  and  $\boldsymbol{\Sigma}$  given in prediction using exact GPR method on page 6-13. As in the SR case,  $\boldsymbol{\mu}_{FIC}$  and  $\boldsymbol{\Sigma}_{FIC}$  are obtained by replacing all occurrences of the true kernel with its FIC approximation. The final forms of  $\boldsymbol{\mu}_{FIC}$  and  $\boldsymbol{\Sigma}_{FIC}$  are as follows:

$$\begin{aligned} \boldsymbol{\mu}_{FIC} &= \mathbf{K}(\mathbf{x}_{new}^T, \mathbf{X}_A | \boldsymbol{\theta}) \mathbf{B}_A^{-1} \mathbf{K}(\mathbf{X}_A, \mathbf{X} | \boldsymbol{\theta}) \boldsymbol{\Lambda}(\boldsymbol{\theta}, \sigma^2, \mathbb{A})^{-1} (\mathbf{y} - \mathbf{H}\boldsymbol{\beta}), \\ \boldsymbol{\Sigma}_{FIC} &= k(\mathbf{x}_{new}, \mathbf{x}_{new} | \boldsymbol{\theta}) - \mathbf{K}(\mathbf{x}_{new}^T, \mathbf{X}_A | \boldsymbol{\theta}) \mathbf{K}(\mathbf{X}_A, \mathbf{X}_A | \boldsymbol{\theta})^{-1} \mathbf{K}(\mathbf{X}_A, \mathbf{x}_{new}^T | \boldsymbol{\theta}) \\ &\quad + \mathbf{K}(\mathbf{x}_{new}^T, \mathbf{X}_A | \boldsymbol{\theta}) \mathbf{B}_A^{-1} \mathbf{K}(\mathbf{X}_A, \mathbf{x}_{new}^T | \boldsymbol{\theta}), \end{aligned}$$

where

$$\mathbf{B}_A = \mathbf{K}(\mathbf{X}_A, \mathbf{X}_A | \boldsymbol{\theta}) + \mathbf{K}(\mathbf{X}_A, \mathbf{X} | \boldsymbol{\theta}) \boldsymbol{\Lambda}(\boldsymbol{\theta}, \sigma^2, A)^{-1} \mathbf{K}(\mathbf{X}, \mathbf{X}_A | \boldsymbol{\theta}),$$
$$\boldsymbol{\Lambda}(\boldsymbol{\theta}, \sigma^2, A) = \boldsymbol{\Omega}(\mathbf{X} | \boldsymbol{\theta}, A) + \sigma^2 \mathbf{I}_n.$$

## References

- [1] Candela, J. Q. *A Unifying View of Sparse Approximate Gaussian Process Regression*. Journal of Machine Learning Research. Vol 6, pp. 1939–1959, 2005.

## See Also

`fitrgp` | `predict`

## More About

- “Gaussian Process Regression Models” on page 6-2
- “Exact GPR Method” on page 6-11
- “Subset of Data Approximation for GPR Models” on page 6-18
- “Subset of Regressors Approximation for GPR Models” on page 6-20
- “Block Coordinate Descent Approximation for GPR Models” on page 6-30

## Block Coordinate Descent Approximation for GPR Models

For a large number of observations, using the exact method for parameter estimation and making predictions on new data can be expensive (see “Exact GPR Method” on page 6-11). One of the approximation methods that help deal with this issue for prediction is the Block Coordinate Descent (BCD) method. You can make predictions using the BCD method by first specifying the predict method using the 'PredictMethod', 'bcd' name-value pair argument in the call to `fitrgp`, and then using the `predict` function.

The idea of the BCD method is to compute

$$\hat{\alpha} = \left( \mathbf{K}(\mathbf{X}, \mathbf{X}) + \sigma^2 \mathbf{I}_N \right)^{-1} (\mathbf{y} - \mathbf{H}\beta)$$

in a different way than the exact method. BCD estimates  $\hat{\alpha}$  by solving the following optimization problem:

$$\hat{\alpha} = \arg \min_{\alpha} f(\alpha)$$

where

$$f(\alpha) = \frac{1}{2} \alpha^T \left[ \mathbf{K}(\mathbf{X}, \mathbf{X}) + \sigma^2 \mathbf{I}_N \right] \alpha - \alpha^T (\mathbf{y} - \mathbf{H}\beta).$$

`fitrgp` uses block coordinate descent (BCD) to solve the above optimization problem.

For users familiar with support vector machines (SVMs), sequential minimal optimization (SMO) and ISDA (iterative single data algorithm) used to fit SVMs are special cases of BCD. `fitrgp` performs two steps for each BCD update - block selection and block update. In the block selection phase, `fitrgp` uses a combination of random and greedy strategies for selecting the set of  $\alpha$  coefficients to optimize next. In the block update phase, the selected  $\alpha$  coefficients are optimized while keeping the other  $\alpha$  coefficients fixed to their previous values. These two steps are repeated until

convergence. BCD does not store the  $n * n$  matrix  $\mathbf{K}(\mathbf{X}, \mathbf{X})$  in memory but it just

computes chunks of the  $\mathbf{K}(\mathbf{X}, \mathbf{X})$  matrix as needed. As a result BCD is more memory efficient compared to 'PredictMethod', 'exact'.

Practical experience indicates that it is not necessary to solve the optimization problem for computing  $\alpha$  to very high precision. For example, it is reasonable to stop the

iterations when  $||\nabla f(\alpha)||$  drops below  $\eta ||\nabla f(\alpha_0)||$ , where  $\alpha_0$  is the initial value of  $\alpha$

and  $\eta$  is a small constant (say  $\eta = 10^{-3}$ ). The results from 'PredictMethod', 'exact' and 'PredictMethod', 'bcd' are equivalent except BCD computes  $\alpha$  in a different way. Hence, 'PredictMethod', 'bcd' can be thought of as a memory efficient way of doing 'PredictMethod', 'exact'. BCD is often faster than 'PredictMethod', 'exact' for large  $n$ . However, you cannot compute the prediction intervals using the 'PredictMethod', 'bcd' option. This is because computing prediction intervals requires solving a problem like the one solved for computing  $\alpha$  for every test point, which again is expensive.

## Fit GPR Models Using BCD Approximation

This example shows fitting a Gaussian Process Regression (GPR) model to data with a large number of observations, using the Block Coordinate Descent (BCD) Approximation.

### Small n - PredictMethod 'exact' and 'bcd' Produce the Same Results

Generate a small sample dataset.

```
rng(0, 'twister');
n = 1000;
X = linspace(0,1,n)';
X = [X,X.^2];
y = 1 + X*[1;2] + sin(20*X*[1;-2])./(X(:,1)+1) + 0.2*randn(n,1);
```

Create a GPR model using 'FitMethod', 'exact' and 'PredictMethod', 'exact'.

```
gpr = fitrgp(X,y,'KernelFunction','squaredexponential',...
    'FitMethod','exact','PredictMethod','exact');
```

Create another GPR model using 'FitMethod', 'exact' and 'PredictMethod', 'bcd'.

```
gprbcd = fitrgp(X,y,'KernelFunction','squaredexponential',...
    'FitMethod','exact','PredictMethod','bcd','BlockSize',200);
```

'PredictMethod', 'exact' and 'PredictMethod', 'bcd' should be equivalent. They compute the same  $\hat{\alpha}$  but just in different ways. You can also see the iterations by using the 'Verbose', 1 name-value pair argument in the call to fitrgp.

Compute the fitted values using the two methods and compare them:

```
yypred = resubPredict(gpr);  
ypredbcd = resubPredict(gprbcd);  
  
max(abs(yypred-ypredbcd))
```

```
ans =
```

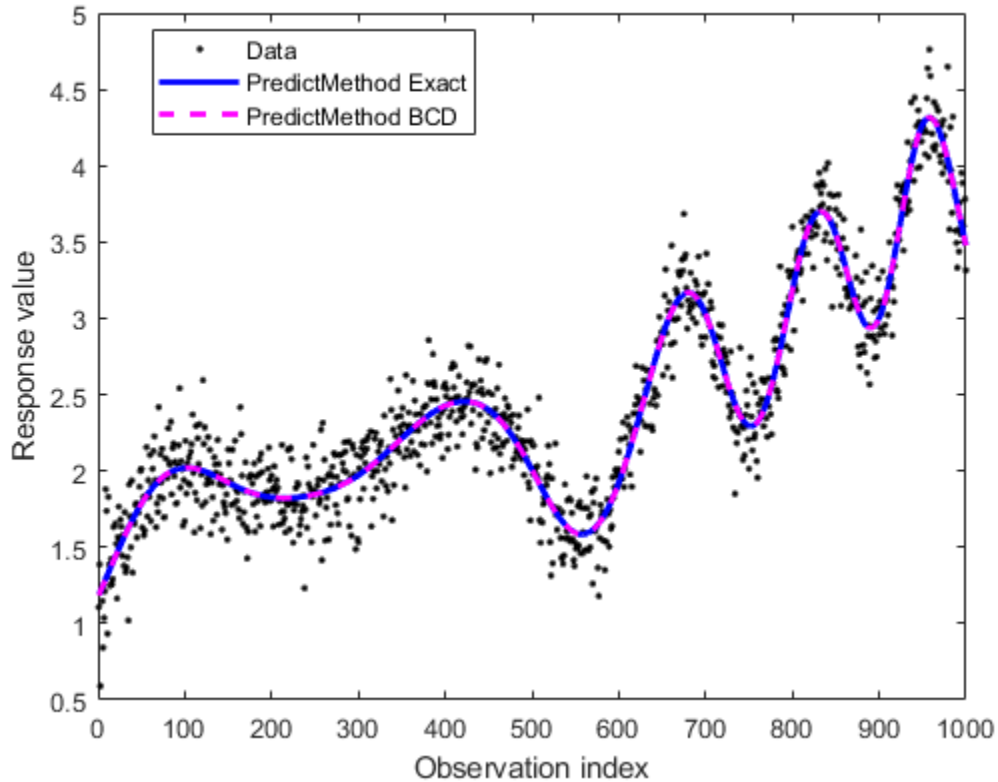
```
5.8853e-04
```

The fitted values are close to each other.

Now, plot the data along with fitted values for 'PredictMethod', 'exact' and 'PredictMethod', 'bcd'.

```
figure;  
plot(y, 'k. ');  
hold on;  
plot(yypred, 'b-', 'LineWidth', 2);  
plot(ypredbcd, 'm--', 'LineWidth', 2);  
legend('Data', 'PredictMethod Exact', 'PredictMethod BCD', 'Location', 'Best');  
xlabel('Observation index');  
ylabel('Response value');
```





It can be seen that 'PredictMethod', 'exact' and 'PredictMethod', 'bcd' produce nearly identical fits.

#### Large n - Use 'FitMethod','sd' and 'PredictMethod','bcd'

Generate a bigger dataset similar to the previous one.

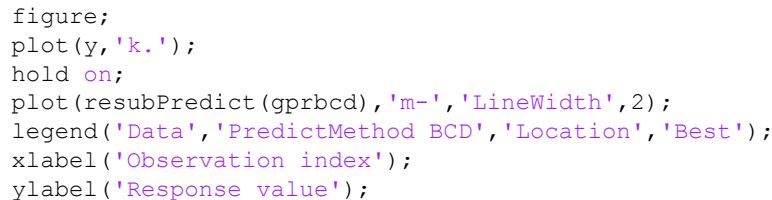
```
rng(0, 'twister');
n = 50000;
X = linspace(0,1,n)';
X = [X,X.^2];
y = 1 + X*[1;2] + sin(20*X*[1;-2])./(X(:,1)+1) + 0.2*randn(n,1);
```

For  $n = 50000$ , the matrix  $K(X, X)$  would be 50000-by-50000. Storing  $K(X, X)$  in memory would require around 20 GB of RAM. To fit a GPR model to this dataset, use 'FitMethod', 'sd' with a random subset of  $m = 2000$  points. The 'ActiveSetSize' name-value pair argument in the call to `fitrgp` specifies the active set size  $m$ . For computing  $\alpha$  use 'PredictMethod', 'bcd' with a 'BlockSize' of 5000. The 'BlockSize' name-value pair argument in `fitrgp` specifies the number of elements of the  $\alpha$  vector that the software optimizes at each iteration. A 'BlockSize' of 5000 assumes that the computer you use can store a 5000-by-5000 matrix in memory.

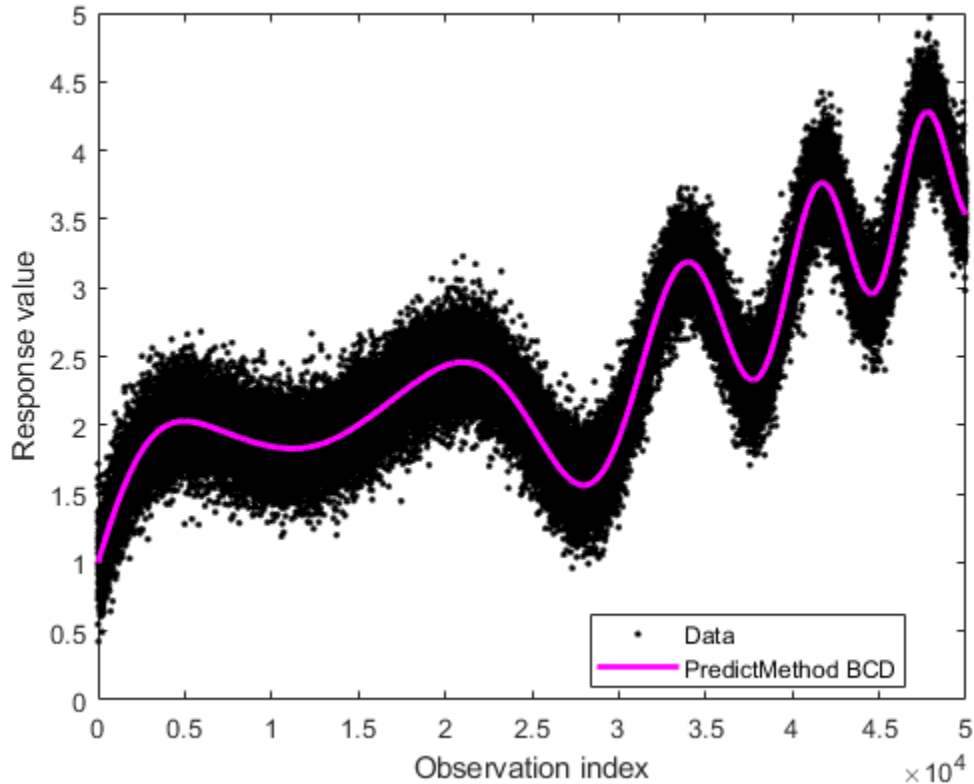
```
gprbcd = fitrgp(X,y,'KernelFunction','squareexponential',...  
              'FitMethod','sd','ActiveSetSize',2000,'PredictMethod','bcd','BlockSize',5000);
```

Plot the data and the fitted values.

```
figure;  
plot(y,'k.');
```



```
hold on;  
plot(resubPredict(gprbcd),'m-','LineWidth',2);  
legend('Data','PredictMethod BCD','Location','Best');  
xlabel('Observation index');  
ylabel('Response value');
```



The plot is similar to the one for smaller  $n$ .

## References

- [1] Grippo, L. and M. Sciandrone. *On the convergence of the block nonlinear gauss-seifel method under convex constraints*. Operations Research Letters. Vol. 26, pp. 127–136, 2000.
- [2] Bo, L. and C. Sminchisescu. *Greed block coordinate descent for large scale Gaussian process regression*. In Proceedings of the Twenty Fourth Conference on Uncertainty in Artificial Intelligence (UAI2008): <http://arxiv.org/abs/1206.3238>, 2012.

## See Also

`fitrgp` | `predict`

## More About

- “Gaussian Process Regression Models” on page 6-2
- “Exact GPR Method” on page 6-11
- “Subset of Data Approximation for GPR Models” on page 6-18
- “Subset of Regressors Approximation for GPR Models” on page 6-20
- “Fully Independent Conditional Approximation for GPR Models” on page 6-26

# Random Number Generation

---

- “Generating Random Data” on page 7-2
- “Random Number Generation Functions” on page 7-3
- “Common Pseudorandom Number Generation Methods” on page 7-5
- “Representing Sampling Distributions Using Markov Chain Samplers” on page 7-14
- “Generating Quasi-Random Numbers” on page 7-18
- “Generating Data Using Flexible Families of Distributions” on page 7-28
- “Bayesian Linear Regression Using Hamiltonian Monte Carlo” on page 7-35

### Generating Random Data

Pseudorandom numbers are generated by deterministic algorithms. They are "random" in the sense that, on average, they pass statistical tests regarding their distribution and correlation. They differ from true random numbers in that they are generated by an algorithm, rather than a truly random process.

Random number generators (RNGs) like those in MATLAB are algorithms for generating pseudorandom numbers with a specified distribution.

For more information on the GUI for generating random numbers from supported distributions, see "Explore the Random Number Generation UI" on page 5-111.

## Random Number Generation Functions

The following table lists the supported distributions and their respective random number generation functions.

Distribution	Random Number Generation Function
Beta on page B-4	betarnd, random, randtool
Binomial on page B-9	binornd, random, randtool
Chi-square on page B-28	chi2rnd, random, randtool
Clayton copula on page 5-164	copularnd
Exponential on page B-34	exprnd, random, randtool
Extreme value on page B-38	evrnd, random, randtool
<i>F</i> on page B-44	frnd, random, randtool
Frank copula on page 5-164	copularnd
Gamma on page B-47	gamrnd, randg, random, randtool
Gaussian copula on page 5-164	copularnd
Gaussian mixture on page B-52	random
Generalized extreme value on page B-53	gevrnd, random, randtool
Generalized Pareto on page B-58	gprnd, random, randtool
Geometric on page B-64	geornd, random, randtool
Gumbel copula on page 5-164	copularnd
Hypergeometric on page B-79	hygernd, random
Inverse Wishart on page B-83	iwishrnd

Distribution	Random Number Generation Function
Johnson system on page B-85	johnsrnd
Lognormal on page B-100	lognrnd, random, randtool
Multinomial on page B-103	mnrnd
Multivariate normal on page B-106	mvnrnd
Multivariate $t$ on page B-112	mvtrnd
Negative binomial on page B-120	nbinrnd, random, randtool
Noncentral chi-square on page B-125	ncx2rnd, random, randtool
Noncentral $F$ on page B-128	ncfrnd, random, randtool
Noncentral $t$ on page B-131	nctrnd, random, randtool
Normal (Gaussian) on page B-134	normrnd, randn, random, randtool
Pearson system on page B-140	pearsrnd
Piecewise on page B-141	random
Poisson on page B-143	poissrnd, random, randtool
Rayleigh on page B-146	raylrnd, random, randtool
Student's $t$ on page B-166	trnd, random, randtool
$t$ copula on page 5-164	copularnd
Uniform (continuous) on page B-184	unifrnd, rand, random
Uniform (discrete) on page B-190	unidrnd, random, randtool
Weibull on page B-193	wblrnd, random
Wishart on page B-198	wishrnd



## Common Pseudorandom Number Generation Methods

### In this section...

“Direct Methods” on page 7-5

“Inversion Methods” on page 7-7

“Acceptance-Rejection Methods” on page 7-10

Methods for generating pseudorandom numbers usually start with uniform random numbers, like the MATLAB `rand` function produces. The methods described in this section detail how to produce random numbers from other distributions.

### Direct Methods

Direct methods directly use the definition of the distribution.

For example, consider binomial on page B-9 random numbers. A binomial random number is the number of heads in  $N$  tosses of a coin with probability  $p$  of a heads on any single toss. If you generate  $N$  uniform random numbers on the interval  $(0,1)$  and count the number less than  $p$ , then the count is a binomial random number with parameters  $N$  and  $p$ .

This function is a simple implementation of a binomial RNG using the direct approach:

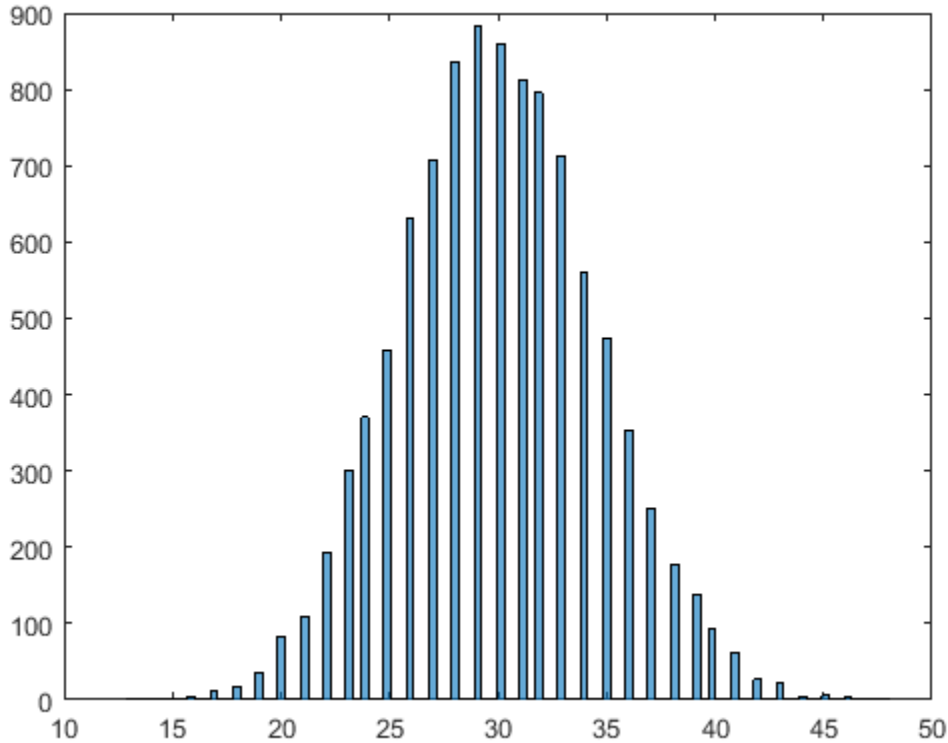
```
function X = directbinornd(N,p,m,n)

X = zeros(m,n); % Preallocate memory
for i = 1:m*n
    u = rand(N,1);
    X(i) = sum(u < p);
end

end
```

For example:

```
X = directbinornd(100,0.3,1e4,1);
histogram(X,101)
```



The Statistics and Machine Learning Toolbox function `binornd` uses a modified direct method, based on the definition of a binomial random variable as the sum of Bernoulli on page B-2 random variables.

You can easily convert the previous method to a random number generator for the Poisson distribution on page B-143 with parameter  $\lambda$ . The Poisson distribution is the limiting case of the binomial distribution as  $N$  approaches infinity,  $p$  approaches zero, and  $Np$  is held fixed at  $\lambda$ . To generate Poisson random numbers, create a version of the previous generator that inputs  $\lambda$  rather than  $N$  and  $p$ , and internally sets  $N$  to some large number and  $p$  to  $\lambda/N$ .

The Statistics and Machine Learning Toolbox function `poissrnd` actually uses two direct methods:

- A waiting time method for small values of  $\lambda$
- A method due to Ahrens and Dieter for larger values of  $\lambda$

## Inversion Methods

Inversion methods are based on the observation that continuous cumulative distribution functions (cdfs) range uniformly over the interval (0,1). If  $u$  is a uniform random number on (0,1), then using  $X = F^{-1}(U)$  generates a random number  $X$  from a continuous distribution with specified cdf  $F$ .

For example, the following code generates random numbers from a specific exponential distribution on page B-34 using the inverse cdf and the MATLAB uniform random number generator `rand`:

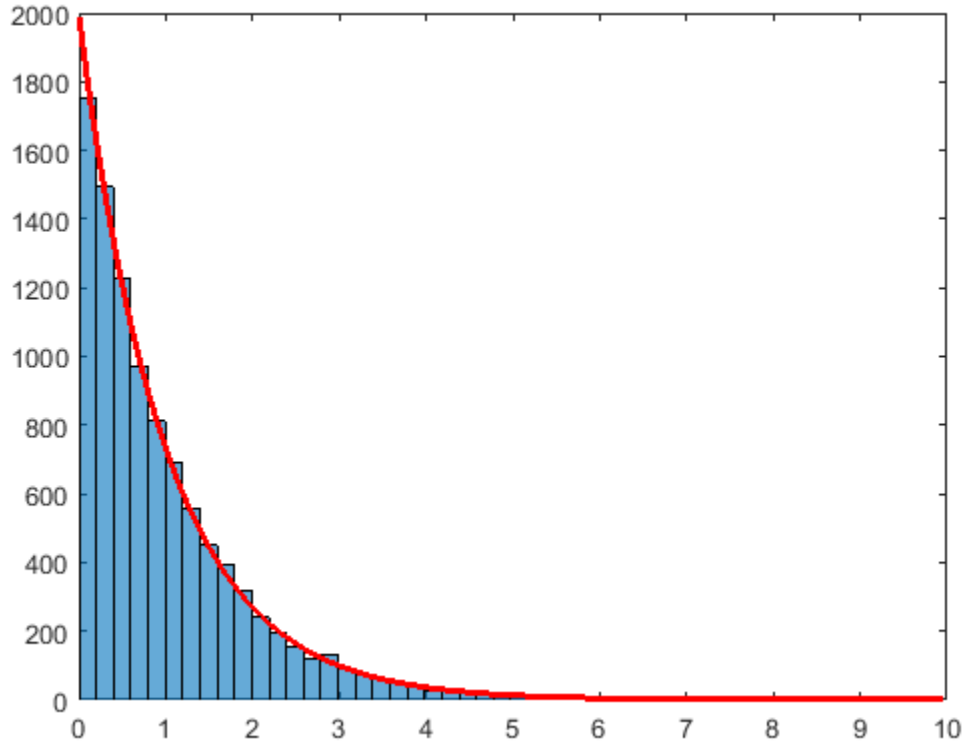
```
mu = 1;
X = expinv(rand(1e4,1),mu);
```

Compare the distribution of the generated random numbers to the pdf of the specified exponential by scaling the pdf to the area of the histogram used to display the distribution:

```
numbins = 50;
h = histogram(X,numbins)
hold on

histarea = h.BinWidth*sum(h.Values);

x = h.BinEdges(1):0.001:h.BinEdges(end);
y = exppdf(x,mu);
plot(x,histarea*y,'r','LineWidth',2)
```



Inversion methods also work for discrete distributions. To generate a random number  $X$  from a discrete distribution with probability mass vector  $P(X=x_i) = p_i$  where  $x_0 < x_1 < x_2 < \dots$ , generate a uniform random number  $u$  on  $(0,1)$  and then set  $X = x_i$  if  $F(x_{i-1}) < u < F(x_i)$ .

For example, the following function implements an inversion method for a discrete distribution with probability mass vector  $p$ :

```
function X = discreteinvrnd(p,m,n)

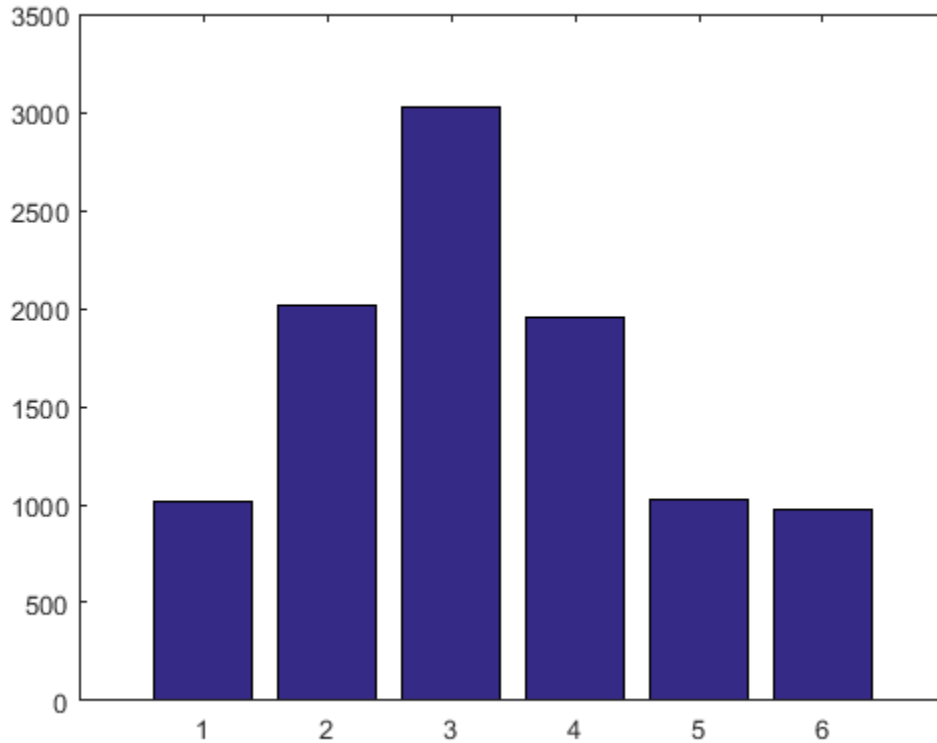
X = zeros(m,n); % Preallocate memory
for i = 1:m*n
    u = rand;
    I = find(u < cumsum(p));
    X(i) = min(I);
end
```

```
end
```

```
end
```

Use the function to generate random numbers from any discrete distribution:

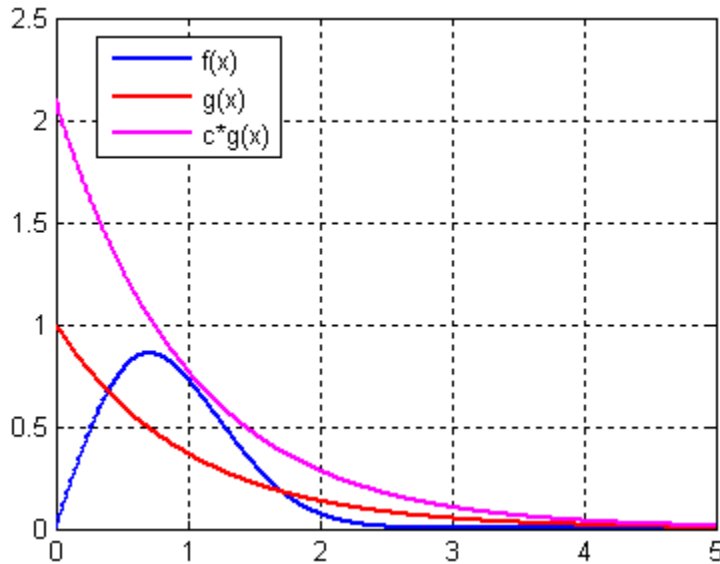
```
p = [0.1 0.2 0.3 0.2 0.1 0.1]; % Probability mass vector  
X = discreteinvrnd(p,1e4,1);  
h = histogram(X,length(p));  
bar(1:length(p),h.Values)
```



## Acceptance-Rejection Methods

The functional form of some distributions makes it difficult or time-consuming to generate random numbers using direct or inversion methods. Acceptance-rejection methods provide an alternative in these cases.

Acceptance-rejection methods begin with uniform random numbers, but require an additional random number generator. If your goal is to generate a random number from a continuous distribution with pdf  $f$ , acceptance-rejection methods first generate a random number from a continuous distribution with pdf  $g$  satisfying  $f(x) \leq cg(x)$  for some  $c$  and all  $x$ .



A continuous acceptance-rejection RNG proceeds as follows:

- 1 Chooses a density  $g$ .
- 2 Finds a constant  $c$  such that  $f(x)/g(x) \leq c$  for all  $x$ .
- 3 Generates a uniform random number  $u$ .
- 4 Generates a random number  $v$  from  $g$ .
- 5 If  $cu \leq f(v)/g(v)$ , accepts and returns  $v$ .
- 6 Otherwise, rejects  $v$  and goes to step 3.

For efficiency, a “cheap” method is necessary for generating random numbers from  $g$ , and the scalar  $c$  should be small. The expected number of iterations to produce a single random number is  $c$ .

The following function implements an acceptance-rejection method for generating random numbers from pdf  $f$ , given  $f$ ,  $g$ , the RNG `grnd` for  $g$ , and the constant  $c$ :

```
function X = accrejrnd(f,g,grnd,c,m,n)

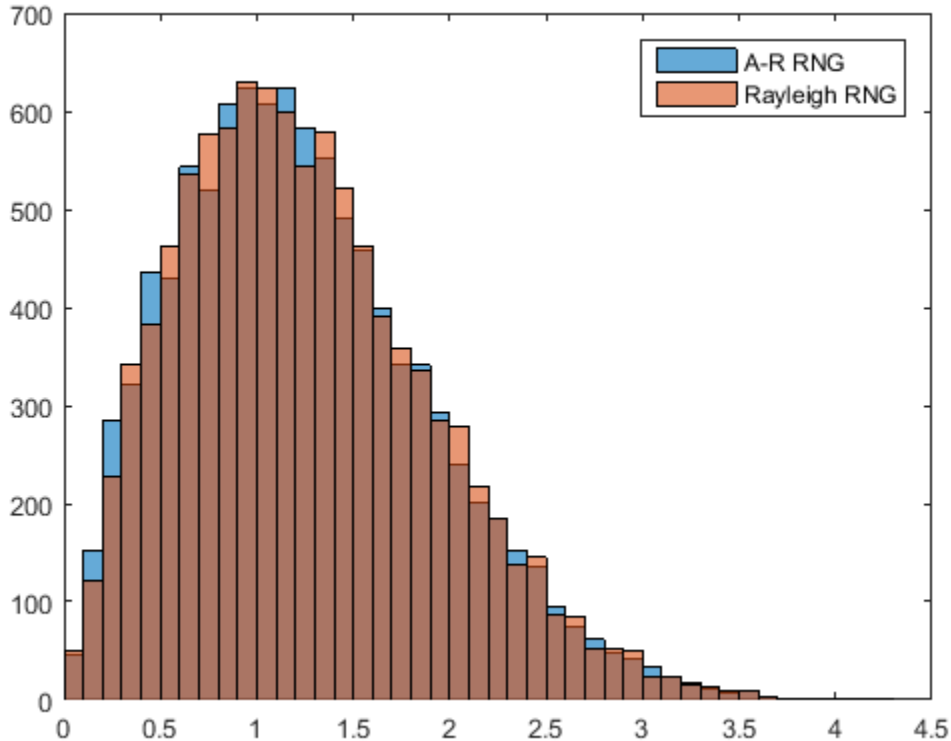
X = zeros(m,n); % Preallocate memory
for i = 1:m*n
    accept = false;
    while accept == false
        u = rand();
        v = grnd();
        if c*u <= f(v)/g(v)
            X(i) = v;
            accept = true;
        end
    end
end
end
```

For example, the function  $f(x) = xe^{-x^2/2}$  satisfies the conditions for a pdf on  $[0,\infty)$  (nonnegative and integrates to 1). The exponential pdf with mean 1,  $f(x) = e^{-x}$ , dominates  $g$  for  $c$  greater than about 2.2. Thus, you can use `rand` and `exprnd` to generate random numbers from  $f$ :

```
f = @(x)x.*exp(-(x.^2)/2);
g = @(x)exp(-x);
grnd = @()exprnd(1);
X = accrejrnd(f,g,grnd,2.2,1e4,1);
```

The pdf  $f$  is actually a Rayleigh distribution on page B-146 with shape parameter 1. This example compares the distribution of random numbers generated by the acceptance-rejection method with those generated by `raylrnd`:

```
Y = raylrnd(1,1e4,1);
histogram(X)
hold on
histogram(Y)
legend('A-R RNG','Rayleigh RNG')
```



The Statistics and Machine Learning Toolbox function `raylrnd` uses a transformation method, expressing a Rayleigh random variable in terms of a chi-square on page B-28 random variable, which you compute using `randn`.

Acceptance-rejection methods also work for discrete distributions. In this case, the goal is to generate random numbers from a distribution with probability mass  $P_p(X = i) = p_i$ , assuming that you have a method for generating random numbers from a distribution with probability mass  $P_q(X = i) = q_i$ . The RNG proceeds as follows:

- 1 Chooses a density  $P_q$ .
- 2 Finds a constant  $c$  such that  $p_i/q_i \leq c$  for all  $i$ .
- 3 Generates a uniform random number  $u$ .



- 4 Generates a random number  $v$  from  $P_q$ .
- 5 If  $cu \leq p/q_v$ , accepts and returns  $v$ .
- 6 Otherwise, rejects  $v$  and goes to step 3.

## Representing Sampling Distributions Using Markov Chain Samplers

### In this section...

“Using the Metropolis-Hastings Algorithm” on page 7-14

“Using Slice Sampling” on page 7-15

“Using Hamiltonian Monte Carlo” on page 7-15

For more complex probability distributions, you might need more advanced methods for generating samples than the methods described in “Common Pseudorandom Number Generation Methods” on page 7-5. Such distributions arise, for example, in Bayesian data analysis and in the large combinatorial problems of Markov chain Monte Carlo (MCMC) simulations. An alternative is to construct a Markov chain with a stationary distribution equal to the target sampling distribution, using the states of the chain to generate random numbers after an initial burn-in period in which the state distribution converges to the target.

### Using the Metropolis-Hastings Algorithm

The Metropolis-Hastings algorithm draws samples from a distribution that is only known up to a constant. Random numbers are generated from a distribution with a probability density function that is equal to or proportional to a proposal function.

To generate random numbers:

- 1 Assume an initial value  $x(t)$ .
- 2 Draw a sample,  $y(t)$ , from a proposal distribution  $q(y | x(t))$ .
- 3 Accept  $y(t)$  as the next sample  $x(t + 1)$  with probability  $r(x(t), y(t))$ , and keep  $x(t)$  as the next sample  $x(t + 1)$  with probability  $1 - r(x(t), y(t))$ , where:

$$r(x, y) = \min \left\{ \frac{f(y) q(x | y)}{f(x) q(y | x)}, 1 \right\}$$

- 4 Increment  $t \rightarrow t + 1$ , and repeat steps 2 and 3 until you get the desired number of samples.

Generate random numbers using the Metropolis-Hastings method with the `mhsample` function. To produce quality samples efficiently with the Metropolis-Hastings algorithm,

it is crucial to select a good proposal distribution. If it is difficult to find an efficient proposal distribution, use slice sampling (`slicesample`) or Hamiltonian Monte Carlo (`hmcSampler`) instead.

## Using Slice Sampling

In instances where it is difficult to find an efficient Metropolis-Hastings proposal distribution, the slice sampling algorithm does not require an explicit specification. The slice sampling algorithm draws samples from the region under the density function using a sequence of vertical and horizontal steps. First, it selects a height at random from 0 to the density function  $f(x)$ . Then, it selects a new  $x$  value at random by sampling from the horizontal “slice” of the density above the selected height. A similar slice sampling algorithm is used for a multivariate distribution.

If a function  $f(x)$  proportional to the density function is given, then do the following to generate random numbers:

- 1 Assume an initial value  $x(t)$  within the domain of  $f(x)$ .
- 2 Draw a real value  $y$  uniformly from  $(0, f(x(t)))$ , thereby defining a horizontal “slice” as  $S = \{x: y < f(x)\}$ .
- 3 Find an interval  $I = (L, R)$  around  $x(t)$  that contains all, or much of the “slice”  $S$ .
- 4 Draw the new point  $x(t + 1)$  within this interval.
- 5 Increment  $t \rightarrow t + 1$  and repeat steps 2 through 4 until you get the desired number of samples.

Slice sampling can generate random numbers from a distribution with an arbitrary form of the density function, provided that an efficient numerical procedure is available to find the interval  $I = (L, R)$ , which is the “slice” of the density.

Generate random numbers using the slice sampling method with the `slicesample` function.

## Using Hamiltonian Monte Carlo

Metropolis-Hastings and slice sampling can produce MCMC chains that mix slowly and take a long time to converge to the stationary distribution, especially in medium-dimensional and high-dimensional problems. Use the gradient-based Hamiltonian Monte Carlo (HMC) sampler to speed up sampling in these situations.

To use HMC sampling, you must specify  $\log f(x)$  (up to an additive constant) and its gradient. You can use a numerical gradient, but this leads to slower sampling. All sampling variables must be unconstrained, meaning that  $\log f(x)$  and its gradient are well-defined for all real  $x$ . To sample constrained variables, transform these variables into unconstrained ones before using the HMC sampler.

The HMC sampling algorithm introduces a random “momentum vector”  $z$  and defines a joint density of  $z$  and the “position vector”  $x$  as  $P(x,z) = f(x)g(z)$ . The goal is to sample from this joint distribution and then to ignore the values of  $z$  — the marginal distribution of  $x$  has the desired density  $f(x)$ .

The HMC algorithm assigns a Gaussian density with covariance matrix  $M$  (the “mass matrix”) to  $z$ :

$$g(z) \propto \exp\left(-\frac{1}{2}z^T M^{-1}z\right)$$

Then, it defines an “energy function” as

$$E(x,z) = -\log f(x) + \frac{1}{2}z^T M^{-1}z = U(x) + K(z)$$

with  $U(x) = -\log f(x)$  the “potential energy” and  $K(z) = z^T M^{-1}z/2$  the “kinetic energy”. The joint density is given by  $P(x,z) \propto \exp\{-E(x,z)\}$ .

To generate random samples, the HMC algorithm:

- 1 Assumes an initial value  $x$  of the position vector.
- 2 Generates a sample of the momentum vector:  $z \sim g(z)$ .
- 3 Evolves the state  $(x, z)$  for some amount of fictitious time  $\tau$  to a new state  $(x', z')$  using the “equations of motion”:

$$\frac{dz}{d\tau} = -\frac{\partial U}{\partial z}$$

$$\frac{dx}{d\tau} = \frac{\partial K}{\partial x}$$

If the equations of motion could be solved exactly, the energy (and hence the density) would remain constant:  $E(x,z) = E(x',z')$ . In practice, the equations of motions must be solved numerically (usually using so-called leapfrog integration) and the energy is not conserved.

- 4 Accepts  $x'$  as the next sample with probability  $p_{\text{acc}} = \min(1, \exp\{E(x,z) - E(x',z')\})$ , and keeps  $x$  as the next sample with probability  $1 - p_{\text{acc}}$ .

**5** Repeats steps 2 through 4 until it has generated the desired number of samples.

To use HMC sampling, create a sampler using the `hmcSampler` function. After creating a sampler, you can compute MAP (maximum-a-posteriori) point estimates, tune the sampler, draw samples, and check convergence diagnostics. For an example of this workflow, see “Bayesian Linear Regression Using Hamiltonian Monte Carlo” on page 7-35.

## See Also

### Functions

`hmcSampler` | `mhsample` | `slicesample`

## Generating Quasi-Random Numbers

<b>In this section...</b>
“Quasi-Random Sequences” on page 7-18
“Quasi-Random Point Sets” on page 7-19
“Quasi-Random Streams” on page 7-26

### Quasi-Random Sequences

Quasi-random number generators (QRNGs) produce highly uniform samples of the unit hypercube. QRNGs minimize the discrepancy between the distribution of generated points and a distribution with equal proportions of points in each sub-cube of a uniform partition of the hypercube. As a result, QRNGs systematically fill the “holes” in any initial segment of the generated quasi-random sequence.

Unlike the pseudorandom sequences described in “Common Pseudorandom Number Generation Methods” on page 7-5, quasi-random sequences fail many statistical tests for randomness. Approximating true randomness, however, is not their goal. Quasi-random sequences seek to fill space uniformly, and to do so in such a way that initial segments approximate this behavior up to a specified density.

QRNG applications include:

- **Quasi-Monte Carlo (QMC) integration.** Monte Carlo techniques are often used to evaluate difficult, multi-dimensional integrals without a closed-form solution. QMC uses quasi-random sequences to improve the convergence properties of these techniques.
- **Space-filling experimental designs.** In many experimental settings, taking measurements at every factor setting is expensive or infeasible. Quasi-random sequences provide efficient, uniform sampling of the design space.
- **Global optimization.** Optimization algorithms typically find a local optimum in the neighborhood of an initial value. By using a quasi-random sequence of initial values, searches for global optima uniformly sample the basins of attraction of all local minima.

#### Example: Using Scramble, Leap, and Skip

Imagine a simple 1-D sequence that produces the integers from 1 to 10. This is the basic sequence and the first three points are  $[1, 2, 3]$ :

1 2 3 4 5 6 7 8 9 10

Now look at how Scramble, Leap, and Skip work together:

- **Scramble** — Scrambling shuffles the points in one of several different ways. In this example, assume a scramble turns the sequence into 1, 3, 5, 7, 9, 2, 4, 6, 8, 10. The first three points are now [1, 3, 5]:

1 3 5 7 9 2 4 6 8 10

- **Skip** — A Skip value specifies the number of initial points to ignore. In this example, set the Skip value to 2. The sequence is now 5, 7, 9, 2, 4, 6, 8, 10 and the first three points are [5, 7, 9]:

1 3 5 7 9 2 4 6 8 10

- **Leap** — A Leap value specifies the number of points to ignore for each one you take. Continuing the example with the Skip set to 2, if you set the Leap to 1, the sequence uses every other point. In this example, the sequence is now 5, 9, 4, 8 and the first three points are [5, 9, 4]:

1 3 5 7 9 2 4 6 8 10

## Quasi-Random Point Sets

Statistics and Machine Learning Toolbox functions support these quasi-random sequences:

- **Halton sequences.** Produced by the `haltonset` function. These sequences use different prime bases to form successively finer uniform partitions of the unit interval in each dimension.
- **Sobol sequences.** Produced by the `sobolset` function. These sequences use a base of 2 to form successively finer uniform partitions of the unit interval, and then reorder the coordinates in each dimension.

- **Latin hypercube sequences.** Produced by the `lhsdesign` function. Though not quasi-random in the sense of minimizing discrepancy, these sequences nevertheless produce sparse uniform samples useful in experimental designs.

Quasi-random sequences are functions from the positive integers to the unit hypercube. To be useful in application, an initial point set of a sequence must be generated. Point sets are matrices of size  $n$ -by- $d$ , where  $n$  is the number of points and  $d$  is the dimension of the hypercube being sampled. The functions `haltonset` and `sobolset` construct point sets with properties of a specified quasi-random sequence. Initial segments of the point sets are generated by the `net` method of the `grandset` on page 32-5245 class (parent class of the `haltonset` on page 32-2987 class and `sobolset` on page 32-6138 class), but points can be generated and accessed more generally using parenthesis indexing.

Because of the way in which quasi-random sequences are generated, they may contain undesirable correlations, especially in their initial segments, and especially in higher dimensions. To address this issue, quasi-random point sets often skip, leap over, or scramble values in a sequence. The `haltonset` and `sobolset` functions allow you to specify both a `Skip` and a `Leap` property of a quasi-random sequence, and the `scramble` method of the `grandset` on page 32-5245 class allows you apply a variety of scrambling techniques. Scrambling reduces correlations while also improving uniformity.

### Generate a Quasi-Random Point Set

This example shows how to use `haltonset` to construct a 2-D Halton quasi-random point set.

Create a `haltonset` object `p`, that skips the first 1000 values of the sequence and then retains every 101st point.

```
rng default % For reproducibility
p = haltonset(2, 'Skip', 1e3, 'Leap', 1e2)

p =

Halton point set in 2 dimensions (89180190640991 points)

Properties:
    Skip : 1000
    Leap : 100
    ScrambleMethod : none
```



The object `p` encapsulates properties of the specified quasi-random sequence. The point set is finite, with a length determined by the `Skip` and `Leap` properties and by limits on the size of point set indices.

Use `scramble` to apply reverse-radix scrambling.

```
p = scramble(p, 'RR2')
```

```
p =
```

```
Halton point set in 2 dimensions (89180190640991 points)
```

```
Properties:
```

```
    Skip : 1000
    Leap : 100
    ScrambleMethod : RR2
```

Use `net` to generate the first 500 points.

```
X0 = net(p,500);
```

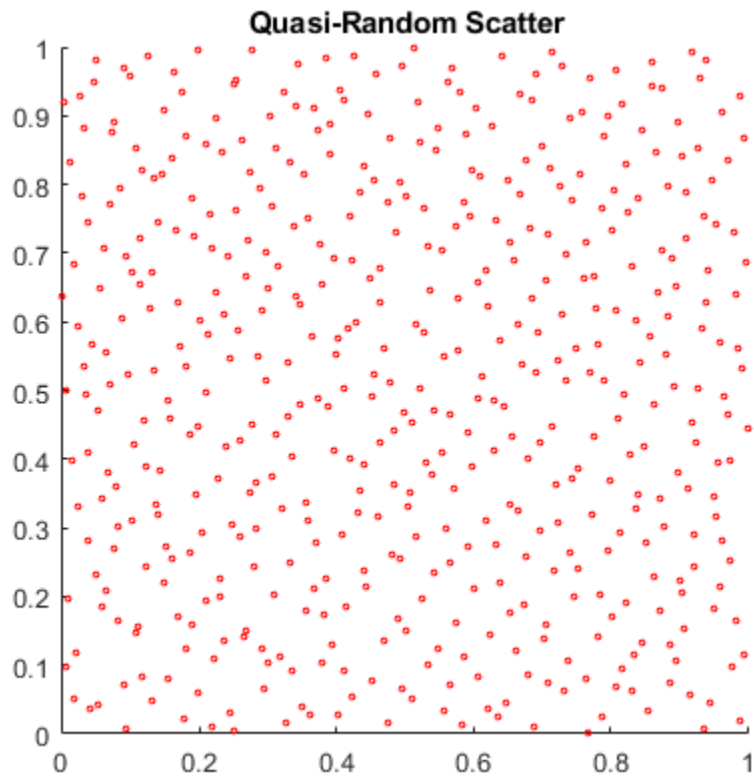
This is equivalent to

```
X0 = p(1:500,:);
```

Values of the point set `X0` are not generated and stored in memory until you access `p` using `net` or parenthesis indexing.

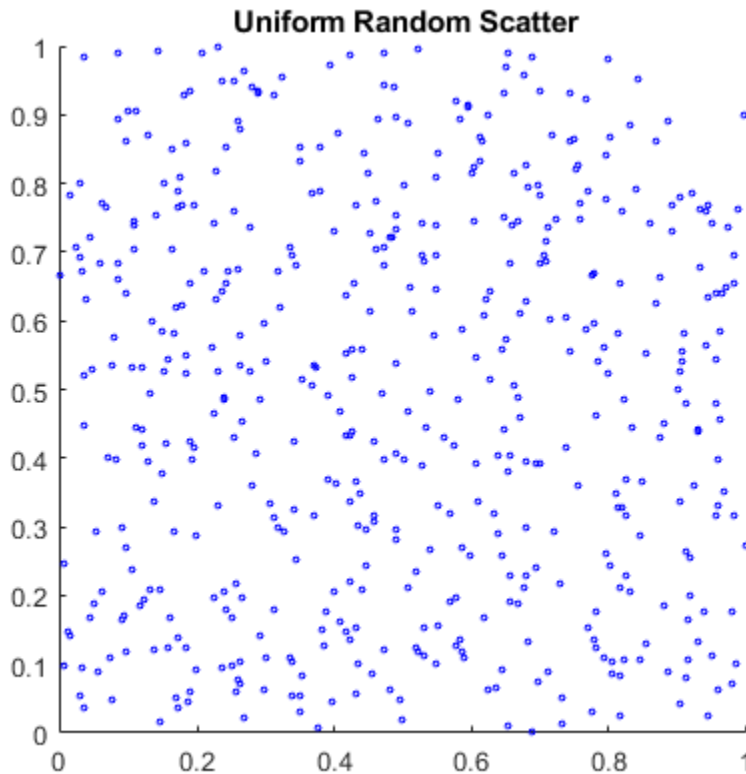
To appreciate the nature of quasi-random numbers, create a scatter plot of the two dimensions in `X0`.

```
scatter(X0(:,1),X0(:,2),5,'r')
axis square
title('\bf Quasi-Random Scatter')
```



Compare this to a scatter of uniform pseudorandom numbers generated by the `rand` function.

```
X = rand(500,2);  
scatter(X(:,1),X(:,2),5,'b')  
axis square  
title('\bf Uniform Random Scatter')
```



The quasi-random scatter appears more uniform, avoiding the clumping in the pseudorandom scatter.

In a statistical sense, quasi-random numbers are too uniform to pass traditional tests of randomness. For example, a Kolmogorov-Smirnov test, performed by `kstest`, is used to assess whether or not a point set has a uniform random distribution. When performed repeatedly on uniform pseudorandom samples, such as those generated by `rand`, the test produces a uniform distribution of  $p$ -values.

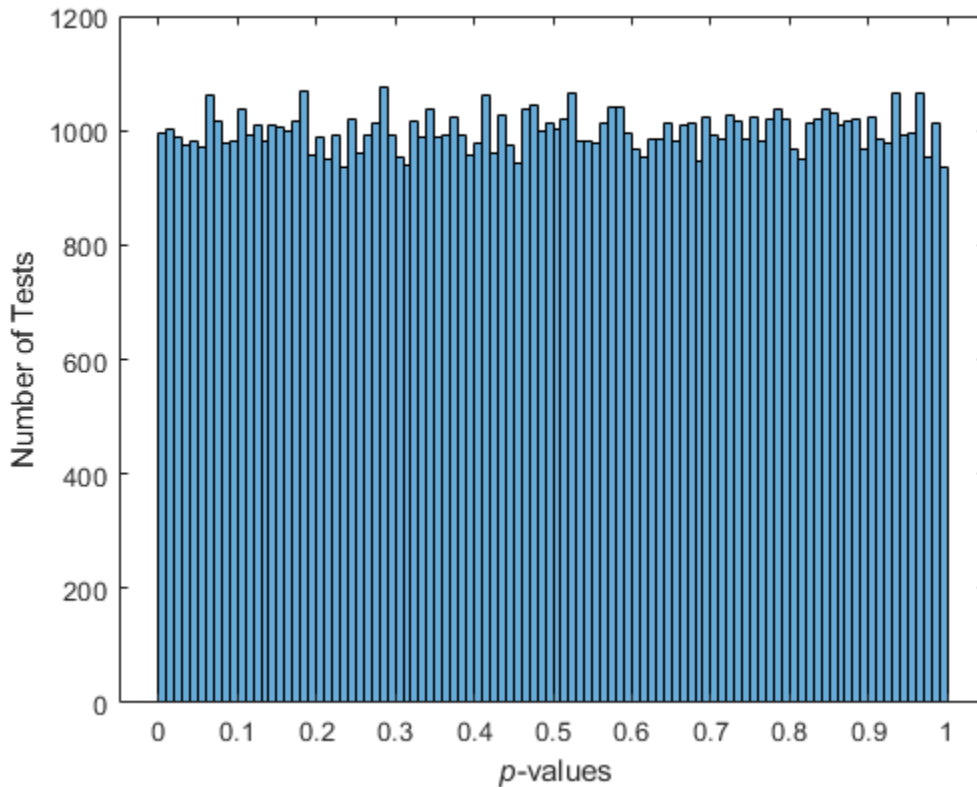
```
nTests = 1e5;
sampSize = 50;
PVALS = zeros(nTests,1);
for test = 1:nTests
    x = rand(sampSize,1);
```

```

[h,pval] = kstest(x,[x,x]);
PVALS(test) = pval;
end

histogram(PVALS,100)
h = findobj(gca,'Type','patch');
xlabel('\it p-values')
ylabel('Number of Tests')

```



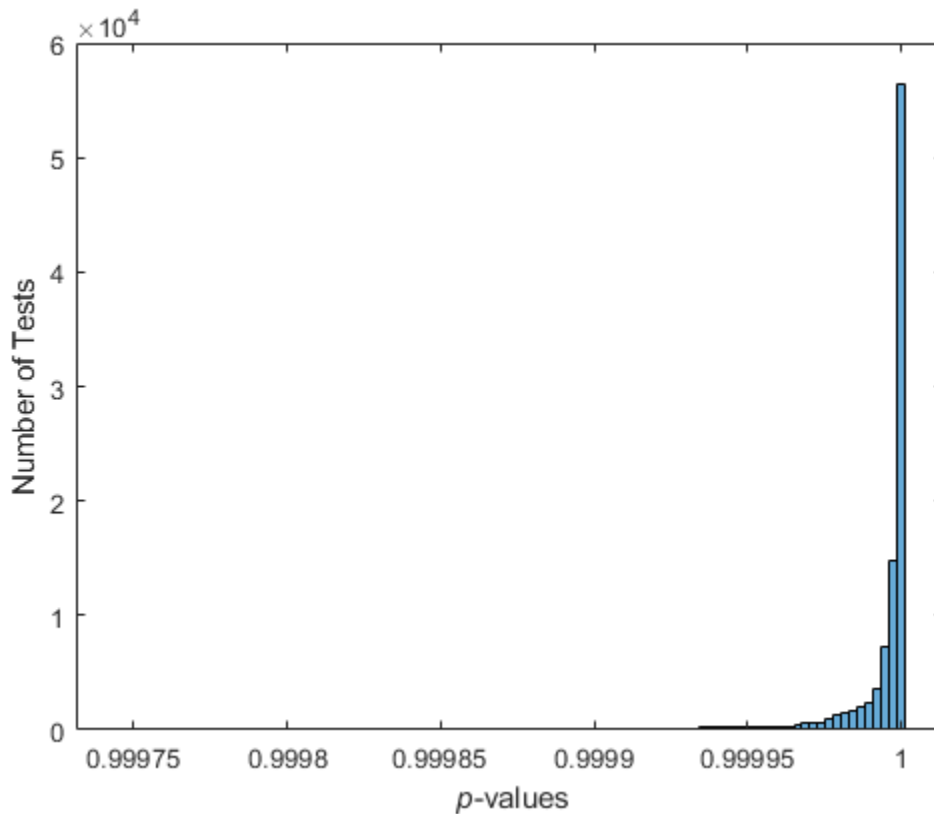
The results are quite different when the test is performed repeatedly on uniform quasi-random samples.

```

p = haltonset(1, 'Skip', 1e3, 'Leap', 1e2);
p = scramble(p, 'RR2');

```

```
nTests = 1e5;  
sampSize = 50;  
PVALS = zeros(nTests,1);  
for test = 1:nTests  
    x = p(test:test+(sampSize-1),:);  
    [h,pval] = kstest(x,[x,x]);  
    PVALS(test) = pval;  
end  
  
histogram(PVALS,100)  
xlabel('\it p-values')  
ylabel('Number of Tests')
```



Small  $p$ -values call into question the null hypothesis that the data are uniformly distributed. If the hypothesis is true, about 5% of the  $p$ -values are expected to fall below 0.05. The results are remarkably consistent in their failure to challenge the hypothesis.

## Quasi-Random Streams

Quasi-random streams, produced by the `grandstream` function, are used to generate sequential quasi-random outputs, rather than point sets of a specific size. Streams are used like pseudoRNGS, such as `rand`, when client applications require a source of quasi-random numbers of indefinite size that can be accessed intermittently. Properties of a quasi-random stream, such as its type (Halton or Sobol), dimension, skip, leap, and scramble, are set when the stream is constructed.

In implementation, quasi-random streams are essentially very large quasi-random point sets, though they are accessed differently. The state of a quasi-random stream is the scalar index of the next point to be taken from the stream. Use the `grand` method of the `grandstream` on page 32-5248 class to generate points from the stream, starting from the current state. Use the `reset` method to reset the state to 1. Unlike point sets, streams do not support parenthesis indexing.

### Generate a Quasi-Random Stream

This example shows how to generate samples from a quasi-random point set.

Use `haltonset` to create a quasi-random point set `p`, then repeatedly increment the index into the point set `test` to generate different samples.

```
p = haltonset(1, 'Skip', 1e3, 'Leap', 1e2);
p = scramble(p, 'RR2');

nTests = 1e5;
sampSize = 50;
PVALS = zeros(nTests, 1);
for test = 1:nTests
    x = p(test:test+(sampSize-1), :);
    [h, pval] = kstest(x, [x, x]);
    PVALS(test) = pval;
end
```

The same results are obtained by using `grandstream` to construct a quasi-random stream `q` based on the point set `p` and letting the stream take care of increments to the index.

```
p = haltonset(1, 'Skip', 1e3, 'Leap', 1e2);
p = scramble(p, 'RR2');
q = grandstream(p);

nTests = 1e5;
sampSize = 50;
PVALS = zeros(nTests,1);
for test = 1:nTests
    X = grand(q, sampSize);
    [h, pval] = kstest(X, [X, X]);
    PVALS(test) = pval;
end
```

## Generating Data Using Flexible Families of Distributions

<b>In this section...</b>
“Pearson and Johnson Systems” on page 7-28
“Generating Data Using the Pearson System” on page 7-29
“Generating Data Using the Johnson System” on page 7-31

### Pearson and Johnson Systems

As described in “Working with Probability Distributions” on page 5-3, choosing an appropriate parametric family of distributions to model your data can be based on *a priori* or *a posteriori* knowledge of the data-producing process, but the choice is often difficult. The Pearson and Johnson systems can make such a choice unnecessary. Each system is a flexible parametric family of distributions that includes a wide range of distribution shapes, and it is often possible to find a distribution within one of these two systems that provides a good match to your data.

#### Data Input

The following parameters define each member of the Pearson and Johnson systems

- Mean — Estimated by `mean`
- Standard deviation — Estimated by `std`
- Skewness — Estimated by `skewness`
- Kurtosis — Estimated by `kurtosis`

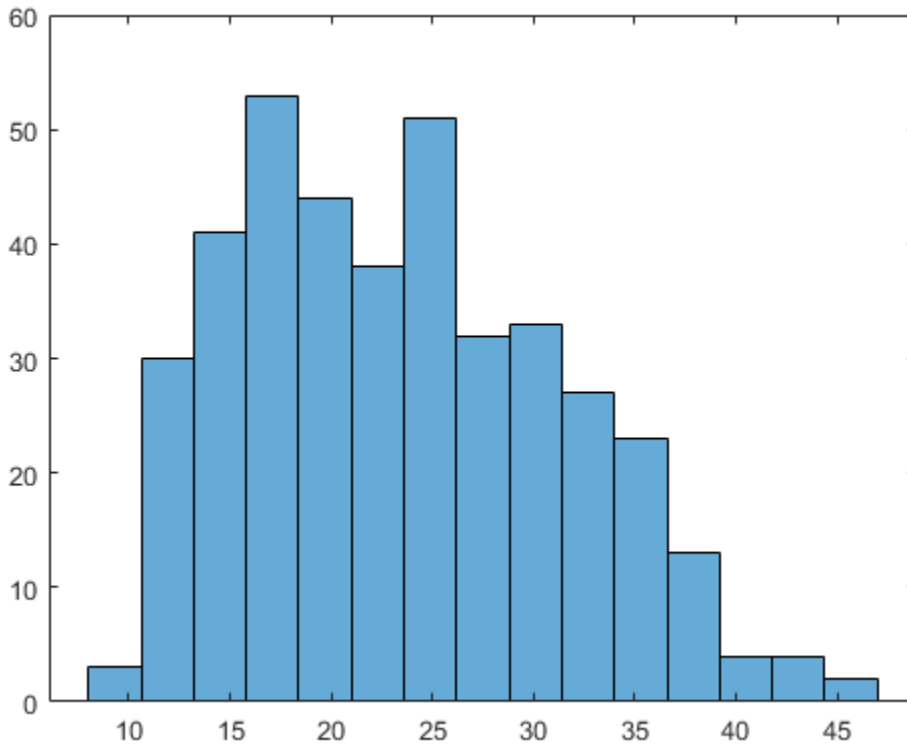
These statistics can also be computed with the `moment` function. The Johnson system, while based on these four parameters, is more naturally described using quantiles, estimated by the `quantile` function.

The Statistics and Machine Learning Toolbox functions `pearsrnd` and `johnsrnd` take input arguments defining a distribution (parameters or quantiles, respectively) and return the type and the coefficients of the distribution in the corresponding system. Both functions also generate random numbers from the specified distribution.

As an example, load the data in `carbig.mat`, which includes a variable `MPG` containing measurements of the gas mileage for each car.



```
load carbig
MPG = MPG(~isnan(MPG));
histogram(MPG,15)
```



The following two sections model the distribution with members of the Pearson and Johnson systems, respectively.

## Generating Data Using the Pearson System

The statistician Karl Pearson devised a system, or family, of distributions that includes a unique distribution corresponding to every valid combination of mean, standard deviation, skewness, and kurtosis. If you compute sample values for each of these

moments from data, it is easy to find the distribution in the Pearson system that matches these four moments and to generate a random sample.

The Pearson system embeds seven basic types of distribution together in a single parametric framework. It includes common distributions such as the normal and  $t$  distributions, simple transformations of standard distributions such as a shifted and scaled beta distribution and the inverse gamma distribution, and one distribution—the Type IV—that is not a simple transformation of any standard distribution.

For a given set of moments, there are distributions that are not in the system that also have those same first four moments, and the distribution in the Pearson system may not be a good match to your data, particularly if the data are multimodal. But the system does cover a wide range of distribution shapes, including both symmetric and skewed distributions.

To generate a sample from the Pearson distribution that closely matches the MPG data, simply compute the four sample moments and treat those as distribution parameters.

```
moments = {mean(MPG), std(MPG), skewness(MPG), kurtosis(MPG)};
rng default % For reproducibility
[r,type] = pearsrnd(moments{:}, 10000, 1);
```

The optional second output from `pearsrnd` indicates which type of distribution within the Pearson system matches the combination of moments.

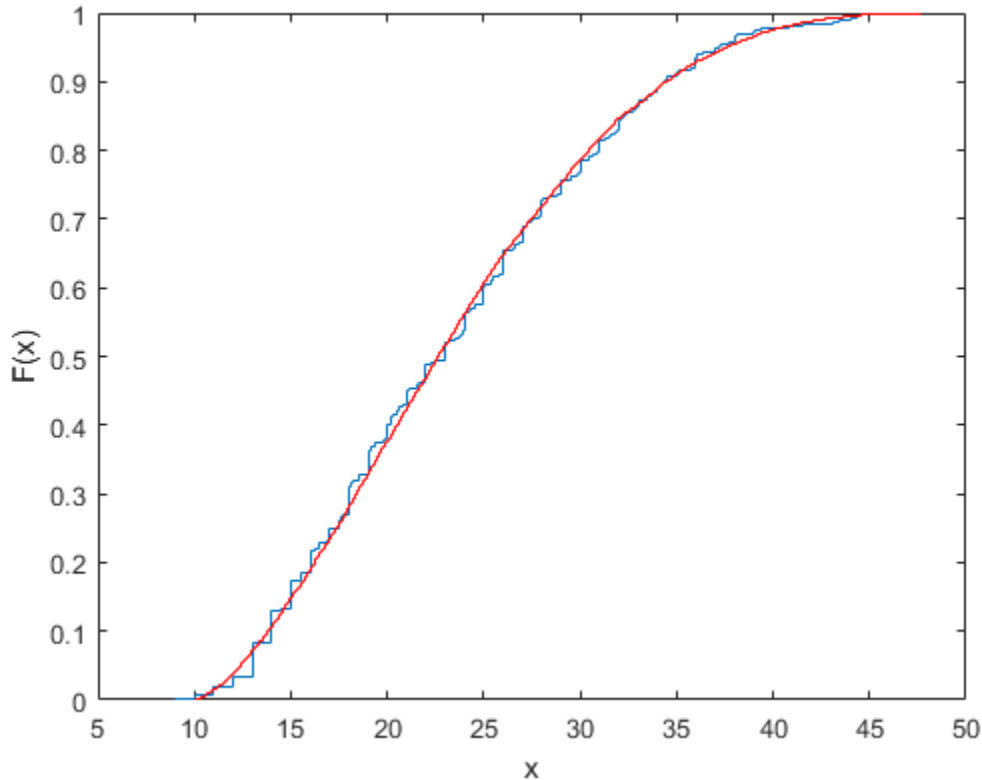
```
type
```

```
type =
     1
```

In this case, `pearsrnd` has determined that the data are best described with a Type I Pearson distribution, which is a shifted, scaled beta distribution.

Verify that the sample resembles the original data by overlaying the empirical cumulative distribution functions.

```
ecdf(MPG);
[Fi,xi] = ecdf(r);
hold on;
stairs(xi,Fi,'r');
hold off
```



## Generating Data Using the Johnson System

Statistician Norman Johnson devised a different system of distributions that also includes a unique distribution for every valid combination of mean, standard deviation, skewness, and kurtosis. However, since it is more natural to describe distributions in the Johnson system using quantiles, working with this system is different than working with the Pearson system.

The Johnson system is based on three possible transformations of a normal random variable, plus the identity transformation. The three nontrivial cases are known as SL, SU, and SB, corresponding to exponential, logistic, and hyperbolic sine transformations. All three can be written as

$$X = \gamma + \delta \cdot \Gamma\left(\frac{(Z-\xi)}{\lambda}\right)$$

where  $Z$  is a standard normal random variable,  $\Gamma$  is the transformation, and  $\gamma$ ,  $\delta$ ,  $\xi$ , and  $\lambda$  are scale and location parameters. The fourth case, SN, is the identity transformation.

To generate a sample from the Johnson distribution that matches the MPG data, first define the four quantiles to which the four evenly spaced standard normal quantiles of -1.5, -0.5, 0.5, and 1.5 should be transformed. That is, you compute the sample quantiles of the data for the cumulative probabilities of 0.067, 0.309, 0.691, and 0.933.

```
probs = normcdf([-1.5 -0.5 0.5 1.5])

probs =
    0.0668    0.3085    0.6915    0.9332

quantiles = quantile(MPG,probs)

quantiles =
    13.0000    18.0000    27.2000    36.0000
```

Then treat those quantiles as distribution parameters.

```
[r1,type] = johnsrnd(quantiles,10000,1);
```

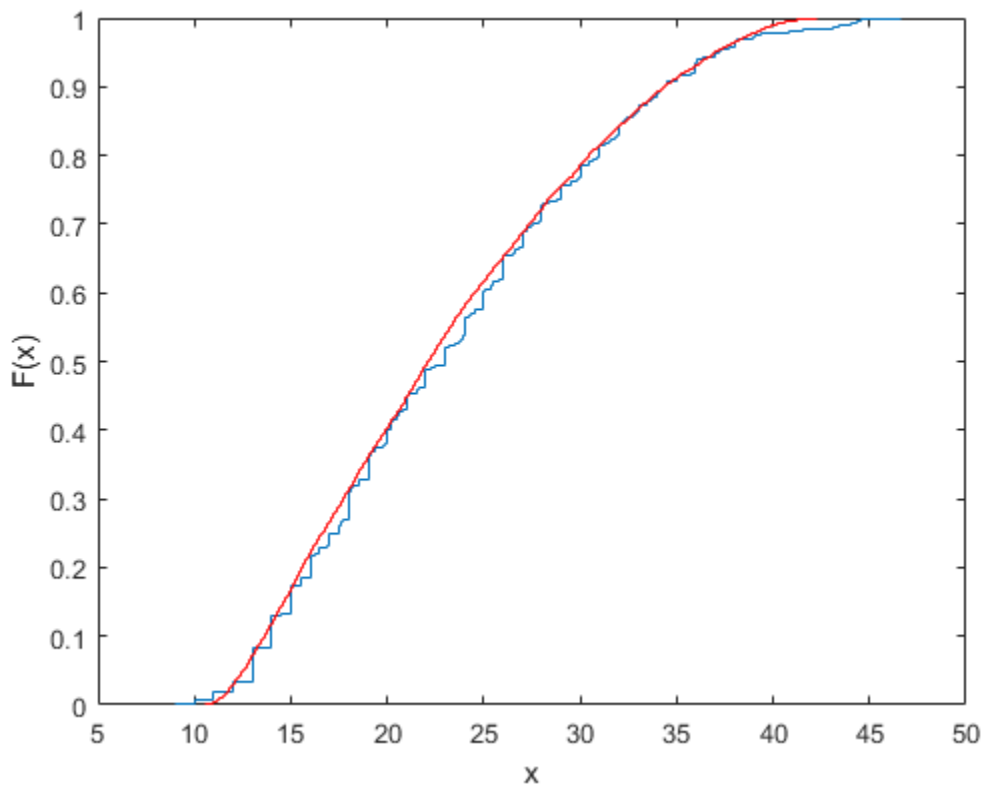
The optional second output from `johnsrnd` indicates which type of distribution within the Johnson system matches the quantiles.

```
type

type =
    'SB'
```

You can verify that the sample resembles the original data by overlaying the empirical cumulative distribution functions.

```
ecdf(MPG);  
[Fi,xi] = ecdf(r1);  
hold on;  
stairs(xi,Fi,'r');  
hold off
```



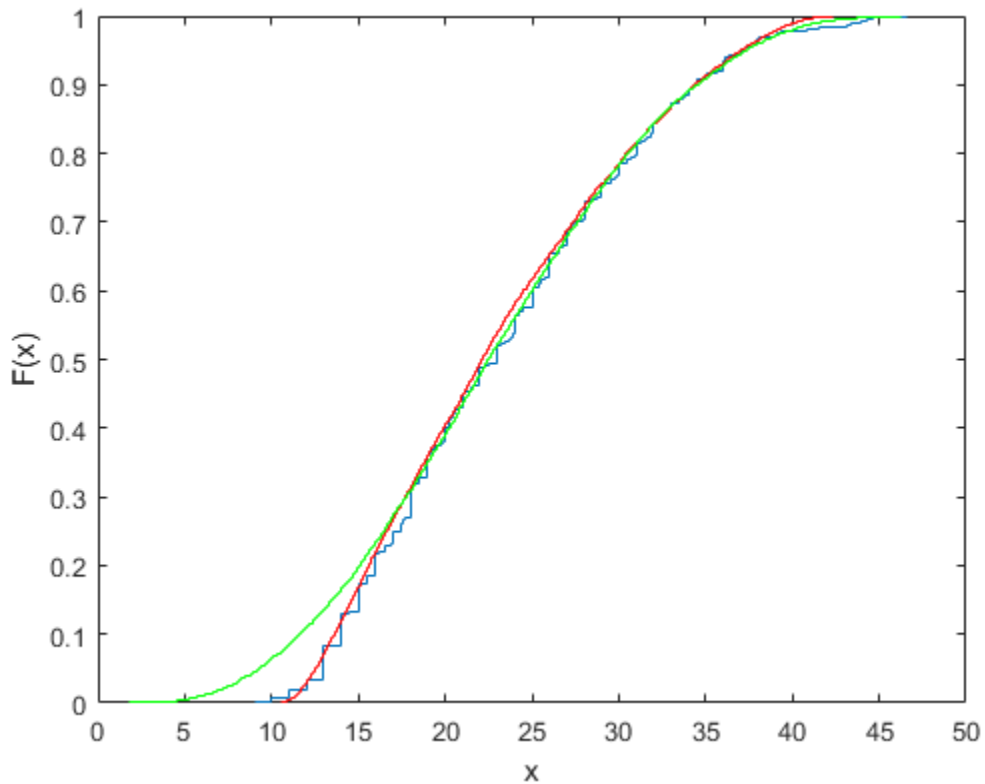
In some applications, it may be important to match the quantiles better in some regions of the data than in others. To do that, specify four evenly spaced standard normal quantiles at which you want to match the data, instead of the default -1.5, -0.5, 0.5, and 1.5. For example, you might care more about matching the data in the right tail than in the left, and so you specify standard normal quantiles that emphasizes the right tail.

```
qnorm = [-.5 .25 1 1.75];  
probs = normcdf(qnorm);
```

```
qemp = quantile(MPG,probs);  
r2 = johnsrnd([qnorm; qemp],10000,1);
```

However, while the new sample matches the original data better in the right tail, it matches much worse in the left tail.

```
[Fj,xj] = ecdf(r2);  
hold on;  
stairs(xj,Fj,'g');  
hold off
```



## Bayesian Linear Regression Using Hamiltonian Monte Carlo

This example shows how to perform Bayesian inference on a linear regression model using a Hamiltonian Monte Carlo (HMC) sampler.

In Bayesian parameter inference, the goal is to analyze statistical models with the incorporation of prior knowledge of model parameters. The posterior distribution of the free parameters  $\theta$  combines the likelihood function  $P(y|\theta)$  with the prior distribution  $P(\theta)$ , using Bayes' theorem:

$$P(\theta|y) = \frac{P(y|\theta)P(\theta)}{P(y)}.$$

Usually, the best way to summarize the posterior distribution is to obtain samples from that distribution using Monte Carlo methods. Using these samples, you can estimate marginal posterior distributions and derived statistics such as the posterior mean, median, and standard deviation. HMC is a gradient-based Markov Chain Monte Carlo sampler that can be more efficient than standard samplers, especially for medium-dimensional and high-dimensional problems.

### Linear Regression Model

Analyze a linear regression model with the intercept  $\alpha$ , the linear coefficients  $\beta$  (a column vector), and the noise variance  $\sigma^2$  of the data distribution as free parameters. Assume that each data point has an independent Gaussian distribution:

$$y_i|\theta \sim \mathcal{N}(\mu_i(\theta), \sigma^2).$$

Model the mean  $\mu_i$  of the Gaussian distribution as a function of the predictors  $x_i$  and model parameters as

$$\mu_i = \alpha + x_i^T \beta.$$

In a Bayesian analysis, you also must assign prior distributions to all free parameters. Assign independent Gaussian priors on the intercept and linear coefficients:

$$\alpha \sim \mathcal{N}(\alpha_0, \sigma_\alpha^2),$$

$$\beta_i \sim \mathcal{N}(\beta_0, \sigma_\beta^2).$$

To use HMC, all sampling variables must be unconstrained, meaning that the posterior density and its gradient must be well-defined for all real parameter values. If you have a parameter that is constrained to an interval, then you must transform this parameter into an unbounded one. To conserve probability, you must multiply the prior distribution by the corresponding Jacobian factor. Also, take this factor into account when calculating the gradient of the posterior.

The noise variance is a (squared) scale parameter that can only be positive. It then can be easier and more natural to consider its logarithm as the free parameter, which is unbounded. Assign a normal prior to the logarithm of the noise variance:

$$\log \sigma^2 \sim \mathcal{N}(\kappa, \omega^2).$$

Write the logarithm of the posterior density of the free parameters  $\theta = (\alpha; \beta; \log \sigma^2)$  as  $\log P(\theta|y) = \text{const.} + \log P(y|\theta) + \log P(\theta)$ .

Ignore the constant term and call the sum of the last two terms  $g(\theta)$ . To use HMC, create a function handle that evaluates  $g(\theta)$  and its gradient  $\partial g / \partial \theta$  for any value of  $\theta$ . The functions used to calculate  $g(\theta)$  are located at the end of the script.

### Create Data Set

Define true parameter values for the intercept, the linear coefficients Beta, and the noise standard deviation. Knowing the true parameter values makes it possible to compare with the output of the HMC sampler. Only the first predictor affects the response.

```
NumPredictors = 2;

trueIntercept = 2;
trueBeta = [3;0];
trueNoiseSigma = 1;
```

Use these parameter values to create a normally distributed sample data set at random values of the two predictors.

```
NumData = 100;
rng('default') %For reproducibility
X = rand(NumData,NumPredictors);
mu = X*trueBeta + trueIntercept;
y = normrnd(mu,trueNoiseSigma);
```



## Define Posterior Probability Density

Choose the means and standard deviations of the Gaussian priors.

```
InterceptPriorMean = 0;
InterceptPriorSigma = 10;
BetaPriorMean = 0;
BetaPriorSigma = 10;
LogNoiseVarianceMean = 0;
LogNoiseVarianceSigma = 2;
```

Save a function `logPosterior` on the MATLAB® path that returns the logarithm of the product of the prior and likelihood, and the gradient of this logarithm. The `logPosterior` function is defined at the end of this example. Then, call the function with arguments to define the `logpdf` input argument to the `hmcSampler` function.

```
logpdf = @(Parameters)logPosterior(Parameters,X,y, ...
    InterceptPriorMean,InterceptPriorSigma, ...
    BetaPriorMean,BetaPriorSigma, ...
    LogNoiseVarianceMean,LogNoiseVarianceSigma);
```

## Create HMC Sampler

Define the initial point to start sampling from, and then call the `hmcSampler` function to create the Hamiltonian sampler as a `HamiltonianSampler` object. Display the sampler properties.

```
Intercept = randn;
Beta = randn(NumPredictors,1);
LogNoiseVariance = randn;
startpoint = [Intercept;Beta;LogNoiseVariance];
smp = hmcSampler(logpdf,startpoint,'NumSteps',50);
```

```
smp
```

```
smp =
```

```
HamiltonianSampler with properties:

    StepSize: 0.1000
    NumSteps: 50
    MassVector: [4x1 double]
    JitterMethod: 'jitter-both'
```

```
StepSizeTuningMethod: 'dual-averaging'  
MassVectorTuningMethod: 'iterative-sampling'  
LogPDF: [function_handle]  
VariableNames: {4x1 cell}  
StartPoint: [4x1 double]
```

### Estimate MAP Point

Estimate the MAP (maximum-a-posteriori) point of the posterior density. You can start sampling from any point, but it is often more efficient to estimate the MAP point, and then use it as a starting point for tuning the sampler and drawing samples. Estimate and display the MAP point. You can show more information during optimization by setting the 'VerbosityLevel' value to 1.

```
[MAPpars,fitInfo] = estimateMAP(smp,'VerbosityLevel',0);  
MAPIntercept = MAPpars(1)  
MAPBeta = MAPpars(2:end-1)  
MAPLogNoiseVariance = MAPpars(end)
```

```
MAPIntercept =
```

```
2.3857
```

```
MAPBeta =
```

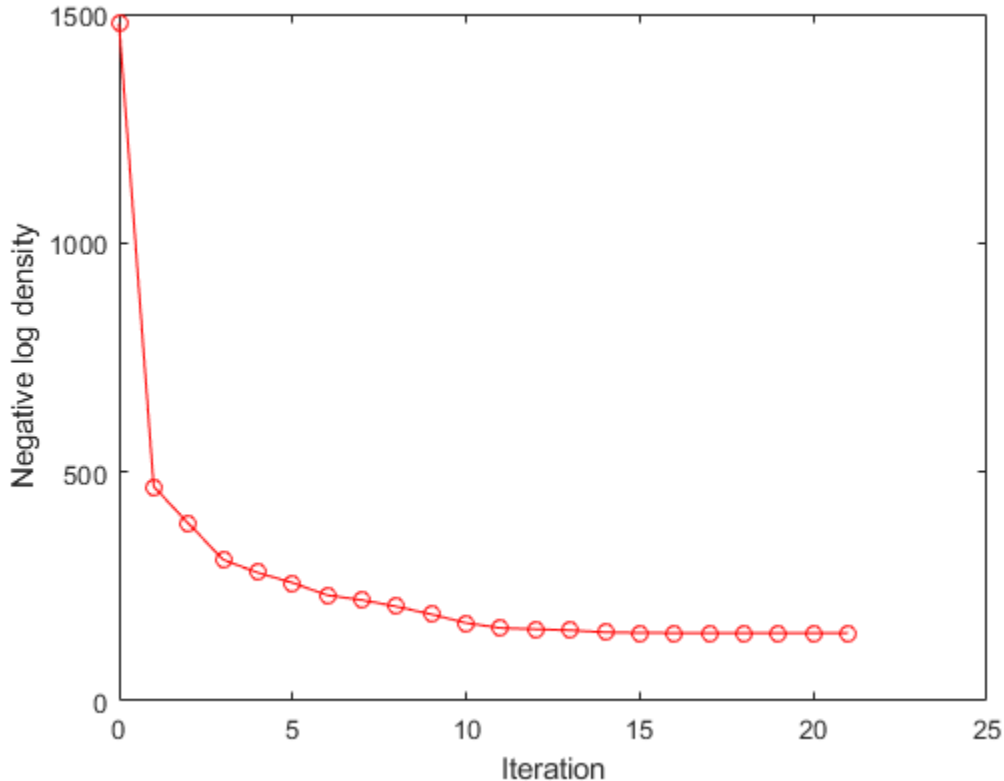
```
2.5495  
-0.4508
```

```
MAPLogNoiseVariance =
```

```
-0.1007
```

To check that the optimization has converged to a local optimum, plot the `fitInfo.Objective` field. This field contains the values of the negative log density at each iteration of the function optimization. The final values are all similar, so the optimization has converged.

```
plot(fitInfo.Iteration,fitInfo.Objective,'ro-');  
xlabel('Iteration');  
ylabel('Negative log density');
```



### Tune Sampler

It is important to select good values for the sampler parameters to get efficient sampling. The best way to find good values is to automatically tune the `MassVector`, `StepSize`, and `NumSteps` parameters using the `tuneSampler` method. Use the method to:

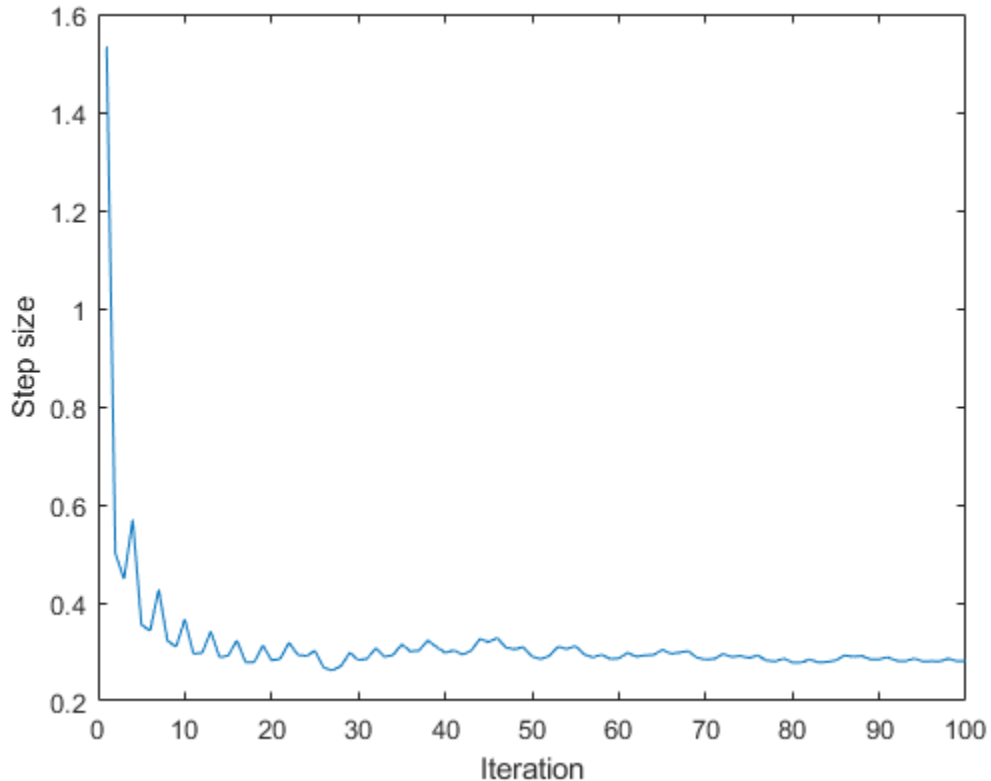
1. Tune the `MassVector` of the sampler.
2. Tune `StepSize` and `NumSteps` for a fixed simulation length to achieve a certain acceptance ratio. The default target acceptance ratio of 0.65 is good in most cases.

Start tuning at the estimated MAP point for more efficient tuning.

```
[smp,tuneinfo] = tuneSampler(smp, 'Start', MAPpars);
```

Plot the evolution of the step size during tuning to ensure that the step size tuning has converged. Display the achieved acceptance ratio.

```
figure;  
plot(tuneinfo.StepSizeTuningInfo.StepSizeProfile);  
xlabel('Iteration');  
ylabel('Step size');  
  
accratio = tuneinfo.StepSizeTuningInfo.AcceptanceRatio  
  
accratio =  
  
    0.6200
```



### Draw Samples

Draw samples from the posterior density, using a few independent chains. Choose different initial points for the chains, randomly distributed around the estimated MAP point. Specify the number of burn-in samples to discard from the beginning of the Markov chain and the number of samples to generate after the burn-in.

Set the 'VerbosityLevel' value to print details during sampling for the first chain.

```
NumChains = 4;  
chains = cell(NumChains,1);  
Burnin = 500;  
NumSamples = 1000;  
for c = 1:NumChains
```

```

if (c == 1)
  level = 1;
else
  level = 0;
end
chains{c} = drawSamples(smp, 'Start', MAPpars + randn(size(MAPpars)), ...
  'Burnin', Burnin, 'NumSamples', NumSamples, ...
  'VerbosityLevel', level, 'NumPrint', 300);
end

```

```

=====
| ITER | LOG PDF | STEP SIZE | NUM STEPS | ACC RATIO | DIVERGENT |
=====
| 300 | -1.486503e+02 | 2.534e-01 | 12 | 9.467e-01 | 0 |
| 600 | -1.492737e+02 | 2.130e-02 | 4 | 9.433e-01 | 0 |
| 900 | -1.508913e+02 | 2.173e-01 | 5 | 9.433e-01 | 0 |
| 1200 | -1.494464e+02 | 1.129e-01 | 16 | 9.350e-01 | 0 |
| 1500 | -1.487679e+02 | 2.534e-01 | 12 | 9.360e-01 | 0 |
=====

```

### Examine Convergence Diagnostics

Use the `diagnostics` method to compute standard MCMC diagnostics. For each sampling parameter, the method uses all the chains to compute these statistics:

- Posterior mean estimate (Mean)
- Estimate of the Monte Carlo standard error (MCSE), which is the standard deviation of the posterior mean estimate
- Estimate of the posterior standard deviation (SD)
- Estimates of the 5th and 95th quantiles of the marginal posterior distribution (Q5 and Q95)
- Effective sample size for the posterior mean estimate (ESS)
- Gelman-Rubin convergence statistic ( $R_{\text{Hat}}$ ). As a rule of thumb, values of  $R_{\text{Hat}}$  less than 1.1 are interpreted as a sign that the chain has converged to the desired distribution. If  $R_{\text{Hat}}$  for any variable is larger than 1.1, then try drawing more samples using the `drawSamples` method.

Display the diagnostics table and the true values of the sampling parameters defined in the beginning of the example. Since the prior distribution is noninformative for this data set, the true values are between (or near) the 5th and 95th quantiles.

```

diags = diagnostics(smp,chains)
truePars = [trueIntercept;trueBeta;log(trueNoiseSigma^2)]

diags =

4x8 table

  Name      Mean      MCSE      SD      Q5      Q95      ESS      RHat
-----
'x1'      2.3774    0.0052064  0.28192  1.9022  2.844    2932    1.000
'x2'      2.5557    0.0062552  0.33522  2.0043  3.1073  2871.9  1.000
'x3'     -0.44138  0.0063307  0.34375 -1.0273  0.10942  2948.3  1.000
'x4'     -0.063056 0.0028132  0.14143 -0.28367 0.17969  2527.7  1.000

truePars =

  2
  3
  0
  0

```

## Visualize Samples

Investigate issues such as convergence and mixing to determine whether the drawn samples represent a reasonable set of random realizations from the target distribution. To examine the output, plot the trace plots of the samples using the first chain.

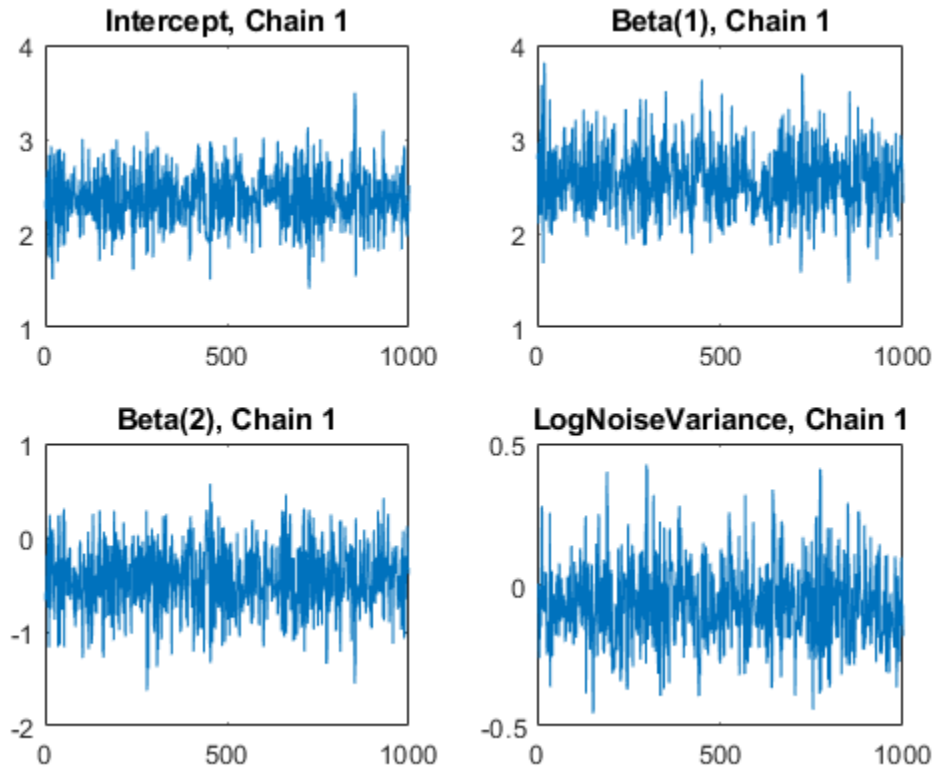
The `drawSamples` method discards burn-in samples from the beginning of the Markov chain to reduce the effect of the sampling starting point. Furthermore, the trace plots look like high-frequency noise, without any visible long-range correlation between the samples. This behavior indicates that the chain is mixed well.

```

figure;
subplot(2,2,1);
plot(chains{1}(:,1));
title(sprintf('Intercept, Chain 1'));
for p = 2:1+NumPredictors
    subplot(2,2,p);
    plot(chains{1}(:,p));
    title(sprintf('Beta(%d), Chain 1',p-1));
end

```

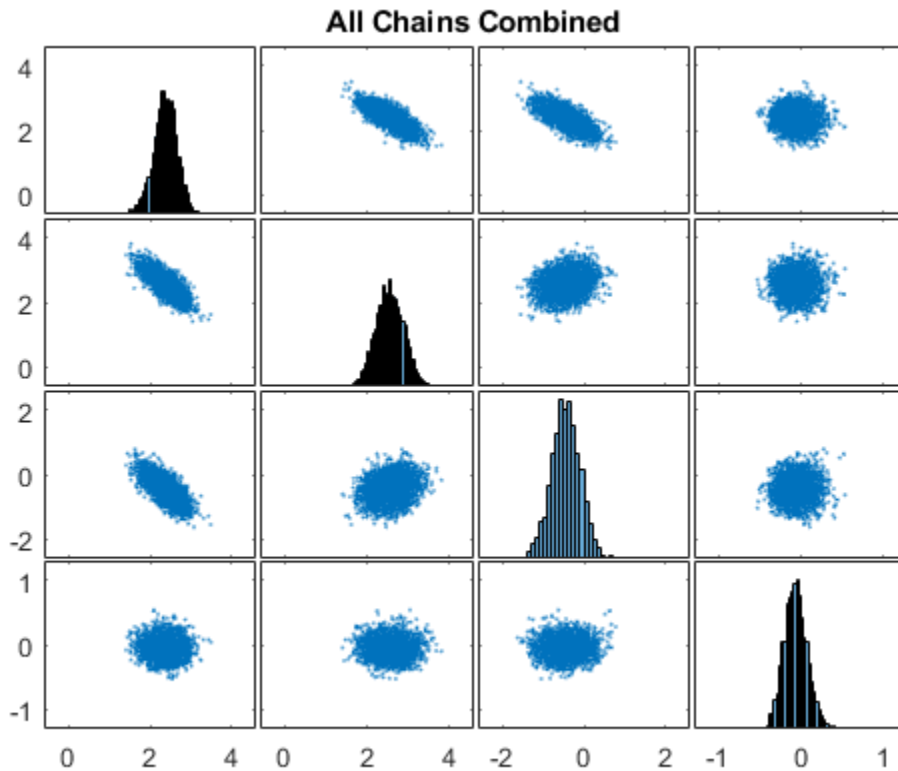
```
end
subplot(2,2,4);
plot(chains{1}(:,end));
title(sprintf('LogNoiseVariance, Chain 1'));
```



Combine the chains into one matrix and create scatter plots and histograms to visualize the 1-D and 2-D marginal posterior distributions.

```
concatenatedSamples = vertcat(chains{:});
figure;
plotmatrix(concatenatedSamples);
title('All Chains Combined');
```





### Functions for Computing Posterior Distribution

The `logPosterior` function returns the logarithm of the product of a normal likelihood and a normal prior for the linear model. The input argument `Parameter` has the format `[Intercept;Beta;LogNoiseVariance]`. `X` and `Y` contain the values of the predictors and response, respectively.

The `normalPrior` function returns the logarithm of the multivariate normal probability density with means `Mu` and standard deviations `Sigma`, specified as scalars or columns vectors the same length as `P`. The second output argument is the corresponding gradient.

```
function [logpdf, gradlogpdf] = logPosterior(Parameters,X,Y, ...
    InterceptPriorMean,InterceptPriorSigma, ...
```

```

BetaPriorMean, BetaPriorSigma, ...
LogNoiseVarianceMean, LogNoiseVarianceSigma)

% Unpack the parameter vector
Intercept      = Parameters(1);
Beta           = Parameters(2:end-1);
LogNoiseVariance = Parameters(end);
% Compute the log likelihood and its gradient
Sigma         = sqrt(exp(LogNoiseVariance));
Mu            = X*Beta + Intercept;
Z             = (Y - Mu)/Sigma;
loglik        = sum(-log(Sigma) - .5*log(2*pi) - .5*Z.^2);
gradIntercept1 = sum(Z/Sigma);
gradBeta1     = X'*Z/Sigma;
gradLogNoiseVariance1 = sum(-.5 + .5*(Z.^2));
% Compute log priors and gradients
[LPIntercept, gradIntercept2] = normalPrior(Intercept, InterceptPriorMean, InterceptPriorSigma);
[LPBeta, gradBeta2]           = normalPrior(Beta, BetaPriorMean, BetaPriorSigma);
[LPLogNoiseVar, gradLogNoiseVariance2] = normalPrior(LogNoiseVariance, LogNoiseVarianceMean, LogNoiseVarianceSigma);
logprior                      = LPIntercept + LPBeta + LPLogNoiseVar;
% Return the log posterior and its gradient
logpdf                        = loglik + logprior;
gradIntercept                 = gradIntercept1 + gradIntercept2;
gradBeta                      = gradBeta1 + gradBeta2;
gradLogNoiseVariance          = gradLogNoiseVariance1 + gradLogNoiseVariance2;
gradlogpdf                    = [gradIntercept; gradBeta; gradLogNoiseVariance];
end

function [logpdf, gradlogpdf] = normalPrior(P, Mu, Sigma)
Z = (P - Mu)./Sigma;
logpdf = sum(-log(Sigma) - .5*log(2*pi) - .5*(Z.^2));
gradlogpdf = -Z./Sigma;
end

```

## See Also

### Functions

hmcSampler

### Classes

HamiltonianSampler

# Hypothesis Tests

---

- “Introduction to Hypothesis Tests” on page 8-2
- “Hypothesis Test Terminology” on page 8-3
- “Hypothesis Test Assumptions” on page 8-5
- “Hypothesis Testing” on page 8-7
- “Available Hypothesis Tests” on page 8-14

## Introduction to Hypothesis Tests

Hypothesis testing is a common method of drawing inferences about a population based on statistical evidence from a sample.

As an example, suppose someone says that at a certain time in the state of Massachusetts the average price of a gallon of regular unleaded gas was \$1.15. How could you determine the truth of the statement? You could try to find prices at every gas station in the state at the time. That approach would be definitive, but it could be time-consuming, costly, or even impossible.

A simpler approach would be to find prices at a small number of randomly selected gas stations around the state, and then compute the sample average.

Sample averages differ from one another due to chance variability in the selection process. Suppose your sample average comes out to be \$1.18. Is the \$0.03 difference an artifact of random sampling or significant evidence that the average price of a gallon of gas was in fact greater than \$1.15? Hypothesis testing is a statistical method for making such decisions.

## Hypothesis Test Terminology

All hypothesis tests share the same basic terminology and structure.

- A *null hypothesis* is an assertion about a population that you would like to test. It is “null” in the sense that it often represents a status quo belief, such as the absence of a characteristic or the lack of an effect. It may be formalized by asserting that a population parameter, or a combination of population parameters, has a certain value. In the example given in the “Introduction to Hypothesis Tests” on page 8-2, the null hypothesis would be that the average price of gas across the state was \$1.15. This is written  $H_0: \mu = 1.15$ .
- An *alternative hypothesis* is a contrasting assertion about the population that can be tested against the null hypothesis. In the example given in the “Introduction to Hypothesis Tests” on page 8-2, possible alternative hypotheses are:

$H_1: \mu \neq 1.15$  — State average was different from \$1.15 (two-tailed test)

$H_1: \mu > 1.15$  — State average was greater than \$1.15 (right-tail test)

$H_1: \mu < 1.15$  — State average was less than \$1.15 (left-tail test)

- To conduct a hypothesis test, a random sample from the population is collected and a relevant *test statistic* is computed to summarize the sample. This statistic varies with the type of test, but its distribution under the null hypothesis must be known (or assumed).
- The *p value* of a test is the probability, under the null hypothesis, of obtaining a value of the test statistic as extreme or more extreme than the value computed from the sample.
- The *significance level* of a test is a threshold of probability  $\alpha$  agreed to before the test is conducted. A typical value of  $\alpha$  is 0.05. If the *p* value of a test is less than  $\alpha$ , the test rejects the null hypothesis. If the *p* value is greater than  $\alpha$ , there is insufficient evidence to reject the null hypothesis. Note that lack of evidence for rejecting the null hypothesis is not evidence for accepting the null hypothesis. Also note that substantive “significance” of an alternative cannot be inferred from the statistical significance of a test.
- The significance level  $\alpha$  can be interpreted as the probability of rejecting the null hypothesis when it is actually true—a *type I error*. The distribution of the test statistic under the null hypothesis determines the probability  $\alpha$  of a type I error. Even if the null hypothesis is not rejected, it may still be false—a *type II error*. The distribution of the test statistic under the alternative hypothesis determines the

probability  $\beta$  of a type II error. Type II errors are often due to small sample sizes. The *power* of a test,  $1 - \beta$ , is the probability of correctly rejecting a false null hypothesis.

- Results of hypothesis tests are often communicated with a *confidence interval*. A confidence interval is an estimated range of values with a specified probability of containing the true population value of a parameter. Upper and lower bounds for confidence intervals are computed from the sample estimate of the parameter and the known (or assumed) sampling distribution of the estimator. A typical assumption is that estimates will be normally distributed with repeated sampling (as dictated by the Central Limit Theorem). Wider confidence intervals correspond to poor estimates (smaller samples); narrow intervals correspond to better estimates (larger samples). If the null hypothesis asserts the value of a population parameter, the test rejects the null hypothesis when the hypothesized value lies outside the computed confidence interval for the parameter.

## Hypothesis Test Assumptions

Different hypothesis tests make different assumptions about the distribution of the random variable being sampled in the data. These assumptions must be considered when choosing a test and when interpreting the results.

For example, the  $z$ -test (`ztest`) and the  $t$ -test (`ttest`) both assume that the data are independently sampled from a normal distribution. Statistics and Machine Learning Toolbox functions are available for testing this assumption, such as `chi2gof`, `jbttest`, `lillietest`, and `normplot`.

Both the  $z$ -test and the  $t$ -test are relatively robust with respect to departures from this assumption, so long as the sample size  $n$  is large enough. Both tests compute a sample mean  $\bar{x}$ , which, by the Central Limit Theorem, has an approximately normal sampling distribution with mean equal to the population mean  $\mu$ , regardless of the population distribution being sampled.

The difference between the  $z$ -test and the  $t$ -test is in the assumption of the standard deviation  $\sigma$  of the underlying normal distribution. A  $z$ -test assumes that  $\sigma$  is known; a  $t$ -test does not. As a result, a  $t$ -test must compute an estimate  $s$  of the standard deviation from the sample.

Test statistics for the  $z$ -test and the  $t$ -test are, respectively,

$$z = \frac{\bar{x} - \mu}{\sigma / \sqrt{n}}$$

$$t = \frac{\bar{x} - \mu}{s / \sqrt{n}}$$

Under the null hypothesis that the population is distributed with mean  $\mu$ , the  $z$ -statistic has a standard normal distribution,  $N(0,1)$ . Under the same null hypothesis, the  $t$ -statistic has Student's  $t$  distribution with  $n - 1$  degrees of freedom. For small sample sizes, Student's  $t$  distribution is flatter and wider than  $N(0,1)$ , compensating for the decreased confidence in the estimate  $s$ . As sample size increases, however, Student's  $t$  distribution approaches the standard normal distribution, and the two tests become essentially equivalent.

Knowing the distribution of the test statistic under the null hypothesis allows for accurate calculation of  $p$ -values. Interpreting  $p$ -values in the context of the test assumptions allows for critical analysis of test results.

Assumptions underlying Statistics and Machine Learning Toolbox hypothesis tests are given in the reference pages for implementing functions.



## Hypothesis Testing

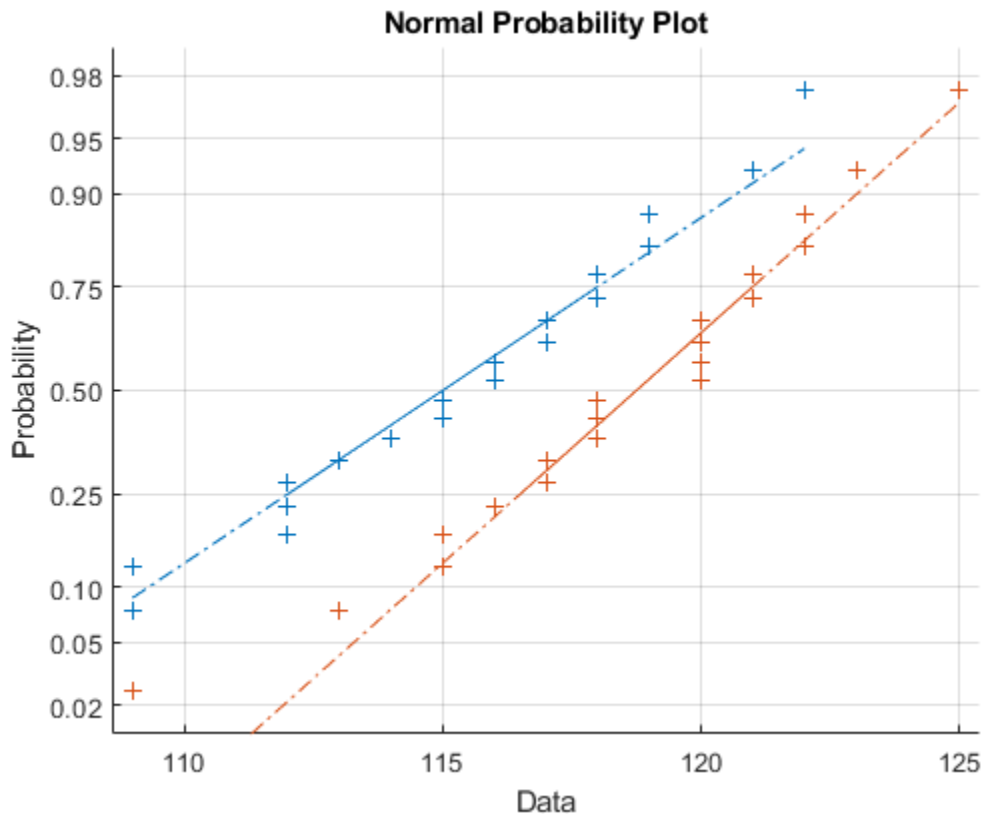
This example shows how to use hypothesis testing to analyze gas prices measured across the state of Massachusetts during two separate months.

This example uses the gas price data in the file `gas.mat`. The file contains two random samples of prices for a gallon of gas around the state of Massachusetts in 1993. The first sample, `price1`, contains 20 random observations around the state on a single day in January. The second sample, `price2`, contains 20 random observations around the state one month later.

```
load gas
prices = [price1 price2];
```

As a first step, you might want to test the assumption that the samples come from normal distributions. A normal probability plot gives a quick idea.

```
normplot(prices)
```



Both scatters approximately follow straight lines through the first and third quartiles of the samples, indicating approximate normal distributions. The February sample (the right-hand line) shows a slight departure from normality in the lower tail. A shift in the mean from January to February is evident. A hypothesis test is used to quantify the test of normality. Since each sample is relatively small, a Lilliefors test is recommended.

```
lillietest(price1)
lillietest(price2)
```

```
ans =
```

```
0
```

```
ans =
```

```
0
```

The default significance level of `lillietest` is 5%. The logical 0 returned by each test indicates a failure to reject the null hypothesis that the samples are normally distributed. This failure may reflect normality in the population or it may reflect a lack of strong evidence against the null hypothesis due to the small sample size.

Now compute the sample means.

```
sample_means = mean(prices)
```

```
sample_means =
```

```
115.1500 118.5000
```

You might want to test the null hypothesis that the mean price across the state on the day of the January sample was \$1.15. If you know that the standard deviation in prices across the state has historically, and consistently, been \$0.04, then a  $z$ -test is appropriate.

```
[h,pvalue,ci] = ztest(prices/100,1.15,0.04)
```

```
h =
```

```
0
```

```
pvalue =
```

```
0.8668
```

```
ci =
```

```
1.1340
```

```
1.1690
```

The logical output  $h = 0$  indicates a failure to reject the null hypothesis at the default significance level of 5%. This is a consequence of the high probability under the null hypothesis, indicated by the  $p$  value, of observing a value as extreme or more extreme of the  $z$ -statistic computed from the sample. The 95% confidence interval on the mean [1.1340 1.1690] includes the hypothesized population mean of \$1.15.

Does the later sample offer stronger evidence for rejecting a null hypothesis of a state-wide average price of \$1.15 in February? The shift shown in the probability plot and the difference in the computed sample means suggest this. The shift might indicate a significant fluctuation in the market, raising questions about the validity of using the historical standard deviation. If a known standard deviation cannot be assumed, a  $t$ -test is more appropriate.

```
[h,pvalue,ci] = ttest(price2/100,1.15)
```

```
h =
```

```
1
```

```
pvalue =
```

```
4.9517e-04
```

```
ci =
```

```
1.1675
```

```
1.2025
```

The logical output  $h = 1$  indicates a rejection of the null hypothesis at the default significance level of 5%. In this case, the 95% confidence interval on the mean does not include the hypothesized population mean of \$1.15.

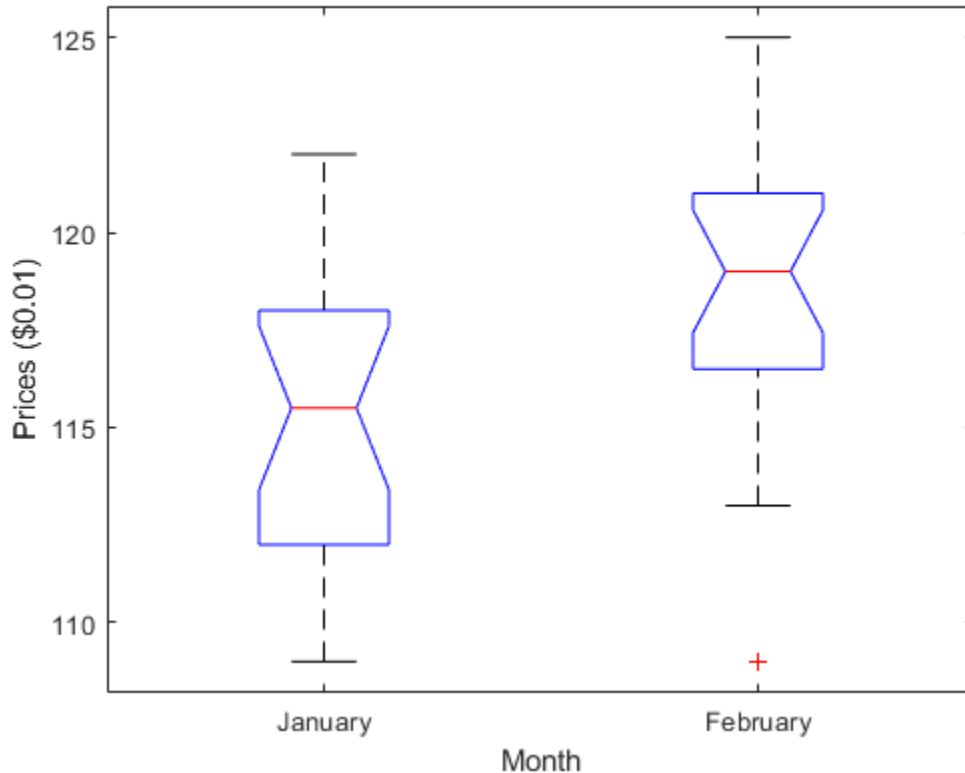
You might want to investigate the shift in prices a little more closely. The function `ttest2` tests if two independent samples come from normal distributions with equal but unknown standard deviations and the same mean, against the alternative that the means are unequal.

```
[h,sig,ci] = ttest2(price1,price2)
```

```
h =  
  
    1  
  
sig =  
  
    0.0083  
  
ci =  
  
    -5.7845  
    -0.9155
```

The null hypothesis is rejected at the default 5% significance level, and the confidence interval on the difference of means does not include the hypothesized value of 0. A notched box plot is another way to visualize the shift.

```
boxplot(prices,1)  
h = gca;  
h.XTick = [1 2];  
h.XTickLabel = {'January', 'February'};  
xlabel('Month')  
ylabel('Prices ($0.01)')
```



The plot displays the distribution of the samples around their medians. The heights of the notches in each box are computed so that the side-by-side boxes have nonoverlapping notches when their medians are different at a default 5% significance level. The computation is based on an assumption of normality in the data, but the comparison is reasonably robust for other distributions. The side-by-side plots provide a kind of visual hypothesis test, comparing medians rather than means. The plot above appears to barely reject the null hypothesis of equal medians.

The nonparametric Wilcoxon rank sum test, implemented by the function `ranksum`, can be used to quantify the test of equal medians. It tests if two independent samples come from identical continuous (not necessarily normal) distributions with equal medians, against the alternative that they do not have equal medians.

```
[p,h] = ranksum(price1,price2)
```

```
p =
```

```
0.0095
```

```
h =
```

```
logical
```

```
1
```

The test rejects the null hypothesis of equal medians at the default 5% significance level.

## Available Hypothesis Tests

Function	Description
<code>ansaribradley</code>	Ansari-Bradley test. Tests if two independent samples come from the same distribution, against the alternative that they come from distributions that have the same median and shape but different variances.
<code>barttest</code>	Bartlett's test. Tests if the variances of the data values along each principal component are equal, against the alternative that the variances are not all equal.
<code>chi2gof</code>	Chi-square goodness-of-fit test. Tests if a sample comes from a specified distribution, against the alternative that it does not come from that distribution.
<code>dwtest</code>	Durbin-Watson test. Tests if the residuals from a linear regression are uncorrelated, against the alternative that there is autocorrelation among them.
<code>friedman</code>	Friedman's test. Tests if the column effects in a two-way layout are all the same, against the alternative that the column effects are not all the same.
<code>jbtest</code>	Jarque-Bera test. Tests if a sample comes from a normal distribution with unknown mean and variance, against the alternative that it does not come from a normal distribution.
<code>kruskalwallis</code>	Kruskal-Wallis test. Tests if multiple samples are all drawn from the same populations (or equivalently, from different populations with the same distribution), against the alternative that they are not all drawn from the same population.
<code>kstest</code>	One-sample Kolmogorov-Smirnov test. Tests if a sample comes from a continuous distribution with specified parameters, against the alternative that it does not come from that distribution.
<code>kstest2</code>	Two-sample Kolmogorov-Smirnov test. Tests if two samples come from the same continuous distribution, against the alternative that they do not come from the same distribution.
<code>lillietest</code>	Lilliefors test. Tests if a sample comes from a distribution in the normal family, against the alternative that it does not come from a normal distribution.



Function	Description
linhypptest	Linear hypothesis test. Tests if $H^*b = c$ for parameter estimates $b$ with estimated covariance $H$ and specified $c$ , against the alternative that $H^*b \neq c$ .
ranksum	Wilcoxon rank sum test. Tests if two independent samples come from identical continuous distributions with equal medians, against the alternative that they do not have equal medians.
runstest	Runs test. Tests if a sequence of values comes in random order, against the alternative that the ordering is not random.
signrank	One-sample or paired-sample Wilcoxon signed rank test. Tests if a sample comes from a continuous distribution symmetric about a specified median, against the alternative that it does not have that median.
signtest	One-sample or paired-sample sign test. Tests if a sample comes from an arbitrary continuous distribution with a specified median, against the alternative that it does not have that median.
ttest	One-sample or paired-sample $t$ -test. Tests if a sample comes from a normal distribution with unknown variance and a specified mean, against the alternative that it does not have that mean.
ttest2	Two-sample $t$ -test. Tests if two independent samples come from normal distributions with unknown but equal (or, optionally, unequal) variances and the same mean, against the alternative that the means are unequal.
vartest	One-sample chi-square variance test. Tests if a sample comes from a normal distribution with specified variance, against the alternative that it comes from a normal distribution with a different variance.
vartest2	Two-sample $F$ -test for equal variances. Tests if two independent samples come from normal distributions with the same variance, against the alternative that they come from normal distributions with different variances.
vartestn	Bartlett multiple-sample test for equal variances. Tests if multiple samples come from normal distributions with the same variance, against the alternative that they come from normal distributions with different variances.

Function	Description
<code>ztest</code>	One-sample $z$ -test. Tests if a sample comes from a normal distribution with known variance and specified mean, against the alternative that it does not have that mean.

---

**Note** In addition to the previous functions, Statistics and Machine Learning Toolbox functions are available for analysis of variance (ANOVA), which perform hypothesis tests in the context of linear modeling.

---

# Analysis of Variance

---

- “Introduction to Analysis of Variance” on page 9-2
- “One-Way ANOVA” on page 9-3
- “Two-Way ANOVA” on page 9-15
- “Multiple Comparisons” on page 9-26
- “N-Way ANOVA” on page 9-36
- “ANOVA with Random Effects” on page 9-49
- “Other ANOVA Models” on page 9-58
- “Analysis of Covariance” on page 9-60
- “Nonparametric Methods” on page 9-69
- “MANOVA” on page 9-72
- “Model Specification for Repeated Measures Models” on page 9-79
- “Compound Symmetry Assumption and Epsilon Corrections” on page 9-81
- “Mauchly’s Test of Sphericity” on page 9-83
- “Multivariate Analysis of Variance for Repeated Measures” on page 9-85

## Introduction to Analysis of Variance

Analysis of variance (ANOVA) is a procedure for assigning sample variance to different sources and deciding whether the variation arises within or among different population groups. Samples are described in terms of variation around group means and variation of group means around an overall mean. If variations within groups are small relative to variations between groups, a difference in group means may be inferred. Hypothesis tests are used to quantify decisions.

## One-Way ANOVA

### In this section...

- “Introduction to One-Way ANOVA” on page 9-3
- “Prepare Data for One-Way ANOVA” on page 9-4
- “Perform One-Way ANOVA” on page 9-6
- “Mathematical Details” on page 9-12

### Introduction to One-Way ANOVA

You can use the Statistics and Machine Learning Toolbox function `anova1` to perform one-way analysis of variance (ANOVA). The purpose of one-way ANOVA is to determine whether data from several groups (levels) of a factor have a common mean. That is, one-way ANOVA enables you to find out whether different groups of an independent variable have different effects on the response variable  $y$ . Suppose, a hospital wants to determine if the two new proposed scheduling methods reduce patient wait times more than the old way of scheduling appointments. In this case, the independent variable is the scheduling method, and the response variable is the waiting time of the patients.

One-way ANOVA is a simple special case of the linear model on page 11-8. The one-way ANOVA form of the model is

$$y_{ij} = \alpha_j + \varepsilon_{ij}$$

with the following assumptions:

- $y_{ij}$  is an observation, in which  $i$  represents the observation number, and  $j$  represents a different group (level) of the predictor variable  $y$ . All  $y_{ij}$  are independent.
- $\alpha_j$  represents the population mean for the  $j$ th group (level or treatment).
- $\varepsilon_{ij}$  is the random error, independent and normally distributed, with zero mean and constant variance, i.e.,  $\varepsilon_{ij} \sim N(0, \sigma^2)$ .

This model is also called the *means model*. The model assumes that the columns of  $y$  are the constant  $\alpha_j$  plus the error component  $\varepsilon_{ij}$ . ANOVA helps determine if the constants are all the same.

ANOVA tests the hypothesis that all group means are equal versus the alternative hypothesis that at least one group is different from the others.

$$H_0 : \alpha_1 = \alpha_2 = \dots = \alpha_k$$

$H_1$  : not all group means are equal

`anova1(y)` tests the equality of column means for the data in matrix `y`, where each column is a different group and has the same number of observations (i.e., a balanced design). `anova1(y, group)` tests the equality of group means, specified in `group`, for the data in vector or matrix `y`. In this case, each group or column can have a different number of observations (i.e., an unbalanced design).

ANOVA is based on the assumption that all sample populations are normally distributed. It is known to be robust to modest violations of this assumption. You can check the normality assumption visually by using a normality plot (`normplot`). Alternatively, you can use one of the Statistics and Machine Learning Toolbox functions that checks for normality: the Anderson-Darling test (`adtest`), the chi-squared goodness of fit test (`chi2gof`), the Jarque-Bera test (`jbtest`), or the Lilliefors test (`lillietest`).

## Prepare Data for One-Way ANOVA

You can provide sample data as a vector or a matrix.

- If the sample data is in a vector, `y`, then you must provide grouping information using the `group` input variable: `anova1(y, group)`.

`group` must be a categorical variable, numeric vector, logical vector, character array, or cell array of character vectors, with one name for each element of `y`. The `anova1` function treats the `y` values corresponding to the same value of `group` as part of the same group. For example,

$$\begin{array}{ccccccccccc} y = [y_1 & y_2 & y_3 & y_4 & y_5 & \dots & \dots & y_N] \\ & \uparrow & \uparrow & \uparrow & \uparrow & & & & & \uparrow \\ g = \{ 'A', & 'A', & 'C', & 'B', & 'B', & \dots & \dots & , & 'D' \} \end{array}$$

Use this design when groups have different numbers of elements (unbalanced ANOVA).

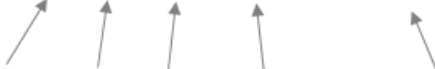
- If the sample data is in a matrix, `y`, providing the group information is optional.
  - If you do not specify the input variable `group`, then `anova1` treats each column of `y` as a separate group, and evaluates whether the population means of the columns are equal. For example,

$$Y = \begin{array}{c} \begin{array}{ccc} \text{group 1} & \text{group 2} & \\ \downarrow & \downarrow & \\ \text{group k} & & \end{array} \\ \begin{bmatrix} Y_{11} & Y_{12} & \cdots & Y_{1k} \\ Y_{21} & Y_{22} & \cdots & Y_{2k} \\ \vdots & \vdots & & \vdots \\ Y_{n1} & Y_{n2} & \cdots & Y_{nk} \end{bmatrix} \end{array}$$

Use this form of design when each group has the same number of elements (balanced ANOVA).

- If you specify the input variable `group`, then `group` must be a character array or cell array of character vectors, with one name for each column of `y`. The `anova1` function treats the columns with the same group name as part of the same group. For example,

$$Y = \begin{bmatrix} Y_{11} & Y_{12} & Y_{13} & Y_{14} & \cdots & Y_{1k} \\ Y_{21} & Y_{22} & Y_{23} & Y_{24} & \cdots & Y_{2k} \\ \vdots & \vdots & \vdots & \vdots & & \vdots \\ Y_{n1} & Y_{n2} & Y_{n3} & Y_{n4} & \cdots & Y_{nk} \end{bmatrix}$$


  
`group = ['Red', 'Black', 'Red', 'Yellow', . . . , 'Black']`

If `group` contains empty or NaN valued cells or character vectors, `anova1` disregards the corresponding observations in `y`.

## Perform One-Way ANOVA

This example shows how to perform one-way ANOVA to determine whether data from several groups have a common mean.

Load and display the sample data.

```
load hogg  
hogg
```

```
hogg =
```

```
    24    14    11     7    19  
    15     7     9     7    24  
    21    12     7     4    19  
    27    17    13     7    15  
    33    14    12    12    10  
    23    16    18    18    20
```

The data comes from a Hogg and Ledolter (1987) study on bacteria counts in shipments of milk. The columns of the matrix `hogg` represent different shipments. The rows are bacteria counts from cartons of milk chosen randomly from each shipment.

Test if some shipments have higher counts than others. By default, `anova1` returns two figures. One is the standard ANOVA table, and the other one is the box plots of data by group.

```
[p,tbl,stats] = anova1(hogg);  
p
```

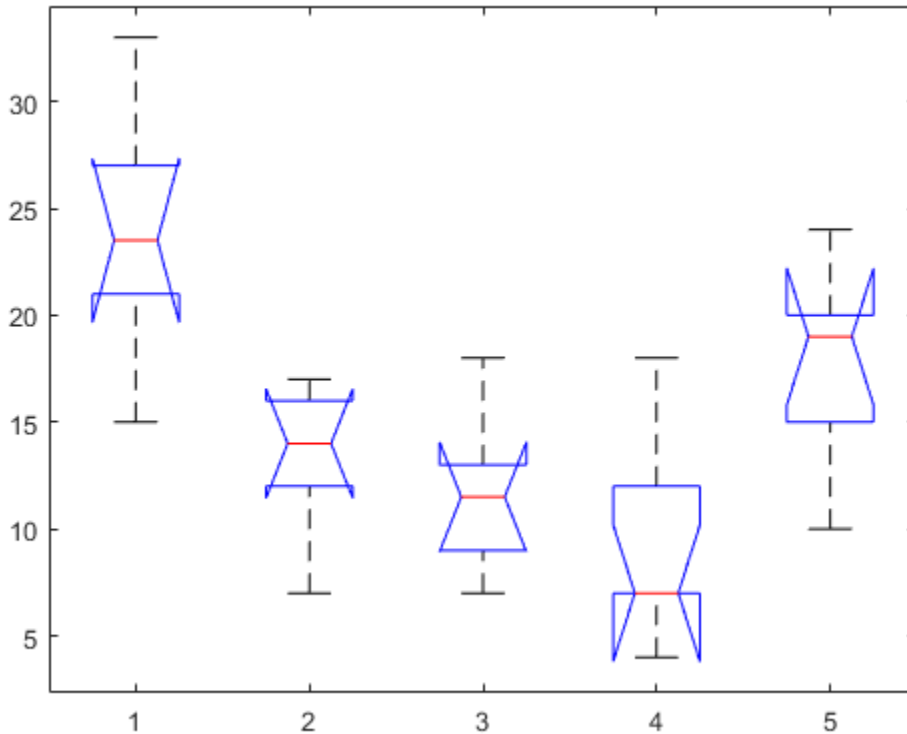
```
p =
```

```
1.1971e-04
```



## ANOVA Table

Source	SS	df	MS	F	Prob>F
Columns	803	4	200.75	9.01	0.0001
Error	557.17	25	22.287		
Total	1360.17	29			



The small  $p$ -value of about 0.0001 indicates that the bacteria counts from the different shipments are not the same.

You can get some graphical assurance that the means are different by looking at the box plots. The notches, however, compare the medians, not the means. For more information on this display, see `boxplot`.

View the standard ANOVA table. `anova1` saves the standard ANOVA table as a cell array in the output argument `tbl`.

```
tbl
```

```
tbl =
```

4x6 cell array

Columns 1 through 5

```

{'Source' }      {'SS'          }      {'df' }      {'MS'          }      {'F'          }
{'Columns' }     {[ 803.0000]}     {[ 4]}      {[200.7500]}     {[ 9.0076]}
{'Error'  }      {[ 557.1667]}     {[25]}      {[ 22.2867]}     {0x0 double}
{'Total'  }      {[1.3602e+03]}     {[29]}      {0x0 double}     {0x0 double}

```

Column 6

```

{'Prob>F'      }
{[1.1971e-04]}
{0x0 double   }
{0x0 double   }

```

Save the  $F$ -statistic value in the variable `Fstat`.

```
Fstat = tbl{2,5}
```

```
Fstat =
```

```
9.0076
```

View the statistics necessary to make a multiple pairwise comparison of group means. `anova1` saves these statistics in the structure `stats`.

```
stats
```

```
stats =
```

```
struct with fields:
```

```

gnames: [5x1 char]
n: [6 6 6 6 6]
source: 'anova1'
means: [23.8333 13.3333 11.6667 9.1667 17.8333]
df: 25
s: 4.7209

```

ANOVA rejects the null hypothesis that all group means are equal, so you can use the multiple comparisons to determine which group means are different from others. To conduct multiple comparison tests, use the function `multcompare`, which accepts `stats` as an input argument. In this example, `anova1` rejects the null hypothesis that the mean bacteria counts from all four shipments are equal to each other, i.e.,

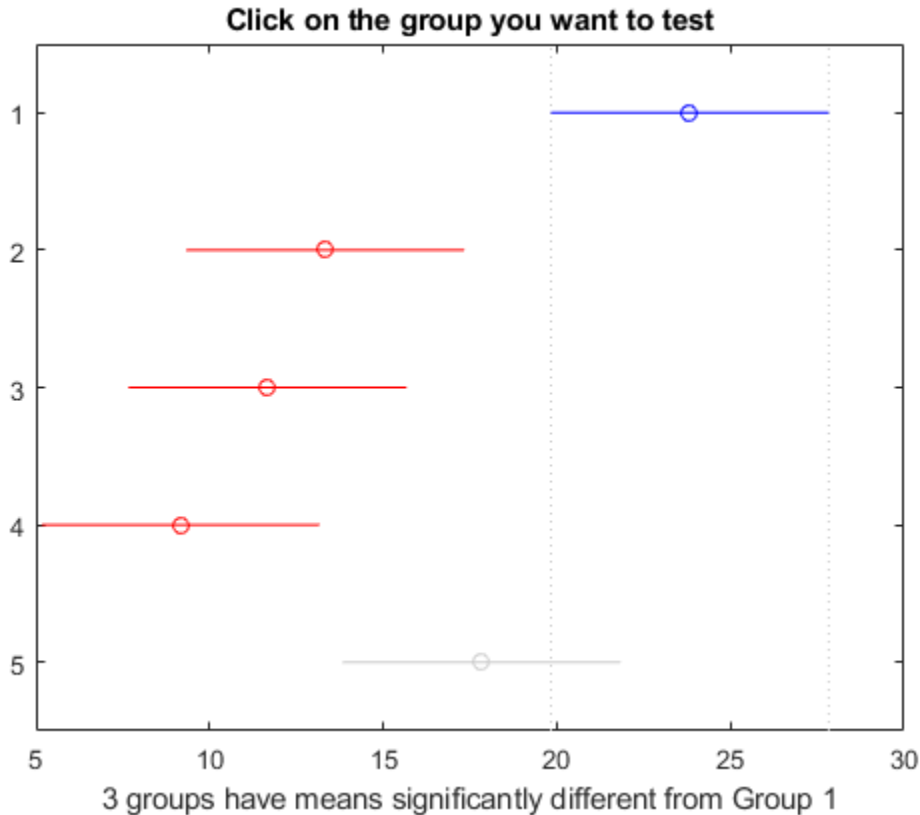
$$H_0 : \mu_1 = \mu_2 = \mu_3 = \mu_4.$$

Perform a multiple comparison test to determine which shipments are different than the others in terms of mean bacteria counts.

```
multcompare(stats)
```

```
ans =
```

1.0000	2.0000	2.4953	10.5000	18.5047	0.0059
1.0000	3.0000	4.1619	12.1667	20.1714	0.0013
1.0000	4.0000	6.6619	14.6667	22.6714	0.0001
1.0000	5.0000	-2.0047	6.0000	14.0047	0.2119
2.0000	3.0000	-6.3381	1.6667	9.6714	0.9719
2.0000	4.0000	-3.8381	4.1667	12.1714	0.5544
2.0000	5.0000	-12.5047	-4.5000	3.5047	0.4806
3.0000	4.0000	-5.5047	2.5000	10.5047	0.8876
3.0000	5.0000	-14.1714	-6.1667	1.8381	0.1905
4.0000	5.0000	-16.6714	-8.6667	-0.6619	0.0292



The first two columns show which group means are compared with each other. For example, the first row compares the means for groups 1 and 2. The last column shows the  $p$ -values for the tests. The  $p$ -values 0.0059, 0.0013, and 0.0001 indicate that the mean bacteria counts in the milk from the first shipment is different from the ones from the second, third, and fourth shipments. The  $p$ -value of 0.0292 indicates that the mean bacteria counts in the milk from the fourth shipment is different from the ones from the fifth. The procedure fails to reject the hypotheses that the other group means are different from each other.

The figure also illustrates the same result. The blue bar shows the comparison interval for the first group mean, which does not overlap with the comparison intervals for the second, third, and fourth group means, shown in red. The comparison interval for the mean of fifth group, shown in gray, overlaps with the comparison interval for the first

group mean. Hence, the group means for the first and fifth groups are not significantly different from each other.

## Mathematical Details

ANOVA tests for the difference in the group means by partitioning the total variation in the data into two components:

- Variation of group means from the overall mean, i.e.,  $\bar{y}_{.j} - \bar{y}_{..}$  (variation between groups), where  $\bar{y}_{.j}$  is the sample mean of group  $j$ , and  $\bar{y}_{..}$  is the overall sample mean.
- Variation of observations in each group from their group mean estimates,  $y_{ij} - \bar{y}_{.j}$  (variation within group).

In other words, ANOVA partitions the total sum of squares (SST) into sum of squares due to between-groups effect (SSR) and sum of squared errors (SSE).

$$\underbrace{\sum_i \sum_j (y_{ij} - \bar{y}_{..})^2}_{SST} = \underbrace{\sum_j n_j (\bar{y}_{.j} - \bar{y}_{..})^2}_{SSR} + \underbrace{\sum_i \sum_j (y_{ij} - \bar{y}_{.j})^2}_{SSE},$$

where  $n_j$  is the sample size for the  $j$ th group,  $j = 1, 2, \dots, k$ .

Then ANOVA compares the variation between groups to the variation within groups. If the ratio of within-group variation to between-group variation is significantly high, then you can conclude that the group means are significantly different from each other. You can measure this using a test statistic that has an  $F$ -distribution with  $(k - 1, N - k)$  degrees of freedom:

$$F = \frac{SSR / k - 1}{SSE / N - k} = \frac{MSR}{MSE} \sim F_{k-1, N-k},$$

where  $MSR$  is the mean squared treatment,  $MSE$  is the mean squared error,  $k$  is the number of groups, and  $N$  is the total number of observations. If the  $p$ -value for the  $F$ -statistic is smaller than the significance level, then the test rejects the null hypothesis that all group means are equal and concludes that at least one of the group means is different from the others. The most common significance levels are 0.05 and 0.01.

## ANOVA Table

The ANOVA table captures the variability in the model by source, the  $F$ -statistic for testing the significance of this variability, and the  $p$ -value for deciding on the significance of this variability. The  $p$ -value returned by `anova1` depends on assumptions about the random disturbances  $\varepsilon_{ij}$  in the model equation. For the  $p$ -value to be correct, these disturbances need to be independent, normally distributed, and have constant variance. The standard ANOVA table has this form:

Source	SS	df	MS	F	$p$ -value
Group (Between)	SSR	$k - 1$	$MSR = SSR / (k - 1)$	MSR/MSE	$P(F_{k-1, N-k}) > F$
Error (Within)	SSE	$N - k$	$MSE = SSE / (N - k)$		
Total	SST	$N - 1$			

`anova1` returns the standard ANOVA table as a cell array with six columns.

Column	Definition
Source	Source of the variability.
SS	Sum of squares due to each source.
df	Degrees of freedom associated with each source. Suppose $N$ is the total number of observations and $k$ is the number of groups. Then, $N - k$ is the within-groups degrees of freedom ( <code>ERROR</code> ), $k - 1$ is the between-groups degrees of freedom ( <code>COLUMNS</code> ), and $N - 1$ is the total degrees of freedom: $N - 1 = (N - k) + (k - 1)$ .
MS	Mean squares for each source, which is the ratio $SS/df$ .
F	$F$ -statistic, which is the ratio of the mean squares.
Prob>F	$p$ -value, which is the probability that the $F$ -statistic can take a value larger than the computed test-statistic value. <code>anova1</code> derives this probability from the cdf of the $F$ -distribution.

The rows of the ANOVA table show the variability in the data, divided by the source.

Row (Source)	Definition
Groups or Columns	Variability due to the differences among the group means (variability <i>between</i> groups)
Error	Variability due to the differences between the data in each group and the group mean (variability <i>within</i> groups)
Total	Total variability

## References

- [1] Wu, C. F. J., and M. Hamada. *Experiments: Planning, Analysis, and Parameter Design Optimization*, 2000.
- [2] Neter, J., M. H. Kutner, C. J. Nachtsheim, and W. Wasserman. 4th ed. *Applied Linear Statistical Models*. Irwin Press, 1996.

## See Also

`anova1` | `kruskalwallis` | `multcompare`

## More About

- “Two-Way ANOVA” on page 9-15
- “N-Way ANOVA” on page 9-36
- “Multiple Comparisons” on page 9-26
- “Nonparametric Methods” on page 9-69



## Two-Way ANOVA

### In this section...

“Introduction to Two-Way ANOVA” on page 9-15

“Prepare Data for Balanced Two-Way ANOVA” on page 9-17

“Perform Two-Way ANOVA” on page 9-18

“Mathematical Details” on page 9-22

### Introduction to Two-Way ANOVA

You can use the Statistics and Machine Learning Toolbox function `anova2` to perform a balanced two-way analysis of variance (ANOVA). To perform two-way ANOVA for an unbalanced design, use `anovan`. For an example, see “Two-Way ANOVA for Unbalanced Design” on page 32-92.

As in one-way ANOVA, the data for a two-way ANOVA study can be experimental or observational. The difference between one-way and two-way ANOVA is that in two-way ANOVA, the effects of two factors on a response variable are of interest. These two factors can be independent, and have no interaction effect, or the impact of one factor on the response variable can depend on the group (level) of the other factor. If the two factors have no interactions, the model is called an *additive* model.

Suppose an automobile company has two factories, and each factory makes the same three car models. The gas mileage in the cars can vary from factory to factory and from model to model. These two factors, factory and model, explain the differences in mileage, that is, the response. One measure of interest is the difference in mileage due to the production methods between factories. Another measure of interest is the difference in the mileage of the models (irrespective of the factory) due to different design specifications. The effects of these measures of interest are *additive*. In addition, suppose only one model has different gas mileage between factories, while the mileage of the other two models is the same between factories. This is called an *interaction* effect. To measure an interaction effect, there must be multiple observations for some combination of factory and car model. These multiple observations are called *replications*.

Two-way ANOVA is a special case of the linear model on page 11-8. The two-way ANOVA form of the model is

$$y_{ijr} = \mu + \alpha_i + \beta_j + (\alpha\beta)_{ij} + \varepsilon_{ijr}$$

where,

- $y_{ijr}$  is an observation of the response variable.
  - $i$  represents group  $i$  of row factor  $A$ ,  $i = 1, 2, \dots, I$
  - $j$  represents group  $j$  of column factor  $B$ ,  $j = 1, 2, \dots, J$
  - $r$  represents the replication number,  $r = 1, 2, \dots, R$

There are a total of  $N = I*J*R$  observations.

- $\mu$  is the overall mean.
- $\alpha_i$  are the deviations of groups of row factor  $A$  from the overall mean  $\mu$  due to row factor  $B$ . The values of  $\alpha_i$  sum to 0.

$$\sum_{i=1}^I \alpha_i = 0.$$

- $\beta_j$  are the deviations of groups in column factor  $B$  from the overall mean  $\mu$  due to row factor  $B$ . All values in a given column of  $\beta_j$  are identical, and the values of  $\beta_j$  sum to 0.

$$\sum_{j=1}^J \beta_j = 0.$$

- $\alpha\beta_{ij}$  are the interactions. The values in each row and in each column of  $\alpha\beta_{ij}$  sum to 0.

$$\sum_{i=1}^I (\alpha\beta)_{ij} = \sum_{j=1}^J (\alpha\beta)_{ij} = 0.$$

- $\varepsilon_{ijr}$  are the random disturbances. They are assumed to be independent, normally distributed, and have constant variance.

In the mileage example:

- $y_{ijr}$  are the gas mileage observations,  $\mu$  is the overall mean gas mileage.
- $\alpha_i$  are the deviations of each car's gas mileage from the mean gas mileage  $\mu$  due to the car's *model*.
- $\beta_j$  are the deviations of each car's gas mileage from the mean gas mileage  $\mu$  due to the car's *factory*.

anova2 requires that data be balanced, so each combination of model and factory must have the same number of cars.

Two-way ANOVA tests hypotheses about the effects of factors  $A$  and  $B$ , and their interaction on the response variable  $y$ . The hypotheses about the equality of the mean response for groups of row factor  $A$  are

$$H_0 : \alpha_1 = \alpha_2 \cdots = \alpha_I$$

$H_1$  : at least one  $\alpha_i$  is different,  $i = 1, 2, \dots, I$ .

The hypotheses about the equality of the mean response for groups of column factor  $B$  are

$$H_0 : \beta_1 = \beta_2 = \cdots = \beta_J$$

$H_1$  : at least one  $\beta_j$  is different,  $j = 1, 2, \dots, J$ .

The hypotheses about the interaction of the column and row factors are

$$H_0 : (\alpha\beta)_{ij} = 0$$

$H_1$  : at least one  $(\alpha\beta)_{ij} \neq 0$

## Prepare Data for Balanced Two-Way ANOVA

To perform balanced two-way ANOVA using `anova2`, you must arrange data in a specific matrix form. The columns of the matrix must correspond to groups of the column factor,  $B$ . The rows must correspond to the groups of the row factor,  $A$ , with the same number of replications for each combination of the groups of factors  $A$  and  $B$ .

Suppose that row factor  $A$  has three groups, and column factor  $B$  has two groups (levels). Also suppose that each combination of factors  $A$  and  $B$  has two measurements or observations (`reps = 2`). Then, each group of factor  $A$  has six observations and each group of factor  $B$  four observations.

$$\begin{array}{cc}
 B = 1 & B = 2 \\
 \left[ \begin{array}{cc}
 y_{111} & y_{121} \\
 y_{112} & y_{122} \\
 y_{211} & y_{221} \\
 y_{212} & y_{222} \\
 y_{311} & y_{321} \\
 y_{312} & y_{322}
 \end{array} \right] & \left. \begin{array}{l} \\ \\ \\ \\ \\ \\ \end{array} \right\} \begin{array}{l} A = 1 \\ \\ A = 2 \\ \\ A = 3 \end{array}
 \end{array}$$

The subscripts indicate row, column, and replication, respectively. For example,  $y_{221}$  corresponds to the measurement for the second group of factor  $A$ , the second group of factor  $B$ , and the first replication for this combination.

## Perform Two-Way ANOVA

This example shows how to perform two-way ANOVA to determine the effect of car model and factory on the mileage rating of cars.

Load and display the sample data.

```
load mileage
mileage

mileage =

    33.3000    34.5000    37.4000
    33.4000    34.8000    36.8000
    32.9000    33.8000    37.6000
    32.6000    33.4000    36.6000
    32.5000    33.7000    37.0000
    33.0000    33.9000    36.7000
```

There are three car models (columns) and two factories (rows). The data has six mileage rows because each factory provided three cars of each model for the study (i.e., the replication number is three). The data from the first factory is in the first three rows, and the data from the second factory is in the last three rows.

Perform two-way ANOVA. Return the structure of statistics, `stats`, to use in multiple comparisons.

```
nmbcars = 3; % Number of cars from each model, i.e., number of replications
[~,~,stats] = anova2(mileage,nmbcars);
```

ANOVA Table					
Source	SS	df	MS	F	Prob>F
Columns	53.3511	2	26.6756	234.22	0
Rows	1.445	1	1.445	12.69	0.0039
Interaction	0.04	2	0.02	0.18	0.8411
Error	1.3667	12	0.1139		
Total	56.2028	17			

You can use the  $F$ -statistics to do hypotheses tests to find out if the mileage is the same across models, factories, and model - factory pairs. Before performing these tests, you must adjust for the additive effects. `anova2` returns the  $p$ -value from these tests.

The  $p$ -value for the model effect (Columns) is zero to four decimal places. This result is a strong indication that the mileage varies from one model to another.

The  $p$ -value for the factory effect (Rows) is 0.0039, which is also highly significant. This value indicates that one factory is out-performing the other in the gas mileage of the cars it produces. The observed  $p$ -value indicates that an  $F$ -statistic as extreme as the observed  $F$  occurs by chance about four out of 1000 times, if the gas mileage were truly equal from factory to factory.

The factories and models appear to have no interaction. The  $p$ -value, 0.8411, means that the observed result is likely (84 out of 100 times), given that there is no interaction.

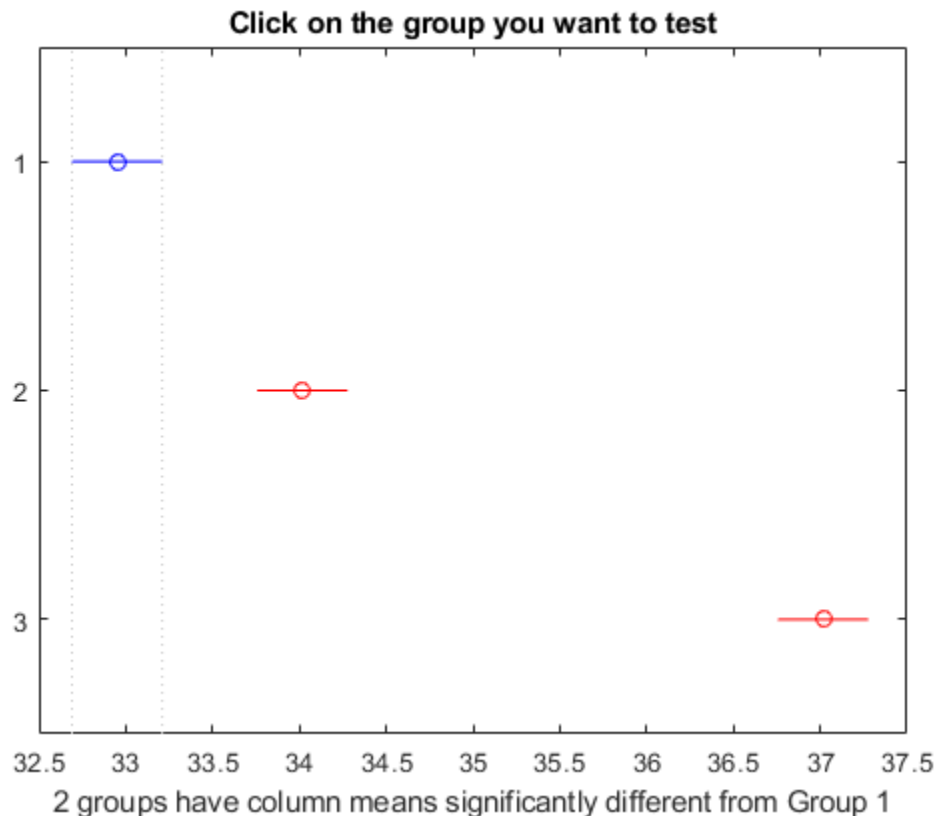
Perform “Multiple Comparisons” on page 9-26 to find out which pair of the three car models is significantly different.

```
c = multcompare(stats)
```

Note: Your model includes an interaction term. A test of main effects can be difficult to interpret when the model includes interactions.

```
c =
```

```
1.0000    2.0000   -1.5865   -1.0667   -0.5469    0.0004
1.0000    3.0000   -4.5865   -4.0667   -3.5469    0.0000
2.0000    3.0000   -3.5198   -3.0000   -2.4802    0.0000
```



In the matrix  $c$ , the first two columns show the pairs of car models that are compared. The last column shows the  $p$ -values for the test. All  $p$ -values are small (0.0004, 0, and 0), which indicates that the mean mileage of all car models are significantly different from each other.

In the figure, the blue bar is the comparison interval for the mean mileage of the first car model. The red bars are the comparison intervals for the mean mileage of the second and third car models. None of the second and third comparison intervals overlap with the first comparison interval, indicating that the mean mileage of the first car model is different from the mean mileage of the second and the third car models. If you click on one of the other bars, you can test for the other car models. None of the comparison

intervals overlap, indicating that the mean mileage of each car model is significantly different from the other two.

## Mathematical Details

The two-factor ANOVA partitions the total variation into the following components:

- Variation of row factor group means from the overall mean,  $\bar{y}_{i..} - \bar{y}_{...}$
- Variation of column factor group means from the overall mean,  $\bar{y}_{.j.} - \bar{y}_{...}$
- Variation of overall mean plus the replication mean from the column factor group mean plus row factor group mean,  $\bar{y}_{ij.} - \bar{y}_{i..} - \bar{y}_{.j.} + \bar{y}_{...}$
- Variation of observations from the replication means,  $y_{ijk} - \bar{y}_{ij.}$

ANOVA partitions the total sum of squares (SST) into the sum of squares due to row factor  $A$  ( $SS_A$ ), the sum of squares due to column factor  $B$  ( $SS_B$ ), the sum of squares due to interaction between  $A$  and  $B$  ( $SS_{AB}$ ), and the sum of squares error (SSE).

$$\underbrace{\sum_{i=1}^m \sum_{j=1}^k \sum_{r=1}^R (y_{ijk} - \bar{y}_{...})^2}_{SST} = \underbrace{kR \sum_{i=1}^m (\bar{y}_{i..} - \bar{y}_{...})^2}_{SS_B} + \underbrace{mR \sum_{j=1}^k (\bar{y}_{.j.} - \bar{y}_{...})^2}_{SS_A} + \underbrace{R \sum_{i=1}^m \sum_{j=1}^k (\bar{y}_{ij.} - \bar{y}_{i..} - \bar{y}_{.j.} + \bar{y}_{...})^2}_{SS_{AB}} + \underbrace{\sum_{i=1}^m \sum_{j=1}^k \sum_{r=1}^R (y_{ijk} - \bar{y}_{ij.})^2}_{SSE}$$

ANOVA takes the variation due to the factor or interaction and compares it to the variation due to error. If the ratio of the two variations is high, then the effect of the factor or the interaction effect is statistically significant. You can measure the statistical significance using a test statistic that has an  $F$ -distribution.

For the null hypothesis that the mean response for groups of the row factor  $A$  are equal, the test statistic is

$$F = \frac{SS_B / m - 1}{SSE / mk(R - 1)} \sim F_{m-1, mk(R-1)}.$$



For the null hypothesis that the mean response for groups of the column factor  $B$  are equal, the test statistic is

$$F = \frac{SS_A / (k - 1)}{SSE / (mk(R - 1))} \sim F_{k-1, mk(R-1)}.$$

For the null hypothesis that the interaction of the column and row factors are equal to zero, the test statistic is

$$F = \frac{SS_{AB} / ((m - 1)(k - 1))}{SSE / (mk(R - 1))} \sim F_{(m-1)(k-1), mk(R-1)}.$$

If the  $p$ -value for the  $F$ -statistic is smaller than the significance level, then ANOVA rejects the null hypothesis. The most common significance levels are 0.01 and 0.05.

### ANOVA Table

The ANOVA table captures the variability in the model by the source, the  $F$ -statistic for testing the significance of this variability, and the  $p$ -value for deciding on the significance of this variability. The  $p$ -value returned by `anova2` depends on assumptions about the random disturbances,  $\varepsilon_{ij}$ , in the model equation. For the  $p$ -value to be correct, these disturbances need to be independent, normally distributed, and have constant variance. The standard ANOVA table has this form:

Source	SS	df	MS	F	$p$ -value
Columns	$SS_A$	$k - 1$	$MS_A$	$MS_A / MSE$	$P(F_{k-1, mk(R-1)}) > F$
Rows	$SS_B$	$m - 1$	$MS_B$	$MS_B / MSE$	$P(F_{m-1, mk(R-1)}) > F$
Interaction	$SS_{AB}$	$(m - 1)(k - 1)$	$MS_{AB}$	$MS_{AB} / MSE$	$P(F_{(m-1)(k-1), mk(R-1)}) > F$
Error	$SSE$	$mk(R - 1)$	$MSE$		
Total	$SST$	$mkR - 1$			

`anova2` returns the standard ANOVA table as a cell array with six columns.

Column	Definition
Source	The source of the variability.

Column	Definition
SS	The sum of squares due to each source.
df	The degrees of freedom associated with each source. Suppose $J$ is the number of groups in the column factor, $I$ is the number of groups in the row factor, and $R$ is the number of replications. Then, the total number of observations is $IJR$ and the total degrees of freedom is $IJR - 1$ . $I - 1$ is the degrees of freedom for the row factor, $J - 1$ is the degrees of freedom for the column factor, $(I - 1)(J - 1)$ is the interaction degrees of freedom, and $IJ(R - 1)$ is the error degrees of freedom.
MS	The mean squares for each source, which is the ratio $SS/df$ .
F	$F$ -statistic, which is the ratio of the mean squares.
Prob>F	The $p$ -value, which is the probability that the $F$ -statistic can take a value larger than the computed test-statistic value. <code>anova2</code> derives this probability from the cdf of the $F$ -distribution.

The rows of the ANOVA table show the variability in the data that is divided by the source.

Row (Source)	Definition
Columns	Variability due to the column factor
Rows	Variability due to the row factor
Interaction	Variability due to the interaction of the row and column factors
Error	Variability due to the differences between the data in each group and the group mean (variability <i>within</i> groups)
Total	Total variability

## References

- [1] Wu, C. F. J., and M. Hamada. *Experiments: Planning, Analysis, and Parameter Design Optimization*, 2000.
- [2] Neter, J., M. H. Kutner, C. J. Nachtsheim, and W. Wasserman. 4th ed. *Applied Linear Statistical Models*. Irwin Press, 1996.

## See Also

[anova1](#) | [anova2](#) | [anovan](#) | [multcompare](#)

## Related Examples

- “Two-Way ANOVA for Unbalanced Design” on page 32-92

## More About

- “One-Way ANOVA” on page 9-3
- “N-Way ANOVA” on page 9-36
- “Multiple Comparisons” on page 9-26
- “Nonparametric Methods” on page 9-69

## Multiple Comparisons

In this section...
“Introduction” on page 9-26
“Multiple Comparisons Using One-Way ANOVA” on page 9-27
“Multiple Comparisons for Three-Way ANOVA” on page 9-29
“Multiple Comparison Procedures” on page 9-32

### Introduction

Analysis of variance (ANOVA) techniques test whether a set of group means (treatment effects) are equal or not. Rejection of the null hypothesis leads to the conclusion that not all group means are the same. This result, however, does not provide further information on which group means are different.

Performing a series of *t*-tests to determine which pairs of means are significantly different is not recommended. When you perform multiple *t*-tests, the probability that the means appear significant, and significant difference results might be due to large number of tests. These *t*-tests use the data from the same sample, hence they are not independent. This fact makes it more difficult to quantify the level of significance for multiple tests.

Suppose that in a single *t*-test, the probability that the null hypothesis ( $H_0$ ) is rejected when it is actually true is a small value, say 0.05. Suppose also that you conduct six independent *t*-tests. If the significance level for each test is 0.05, then the probability that the tests correctly fail to reject  $H_0$ , when  $H_0$  is true for each case, is  $(0.95)^6 = 0.735$ . And the probability that one of the tests incorrectly rejects the null hypothesis is  $1 - 0.735 = 0.265$ , which is much higher than 0.05.

To compensate for multiple tests, you can use multiple comparison procedures. The Statistics and Machine Learning Toolbox function `multcompare` performs multiple pairwise comparison of the group means, or treatment effects. The options are Tukey’s honestly significant difference criterion (default option), the Bonferroni method, Scheffe’s procedure, Fisher’s least significant differences (lsd) method, and Dunn & Sidak’s approach to *t*-test.

To perform multiple comparisons of group means, provide the structure `stats` as an input for `multcompare`. You can obtain `stats` from one of the following functions :

- `anova1` — One-way ANOVA on page 9-3
- `anova2` — Two-way ANOVA on page 9-15
- `anovan` —  $N$ -way ANOVA on page 9-36
- `aoctool` — Interactive ANCOVA on page 9-60
- `kruskalwallis` — Nonparametric method on page 9-69 for one-way layout
- `friedman` — Nonparametric method on page 9-69 for two-way layout

For multiple comparison procedure options for repeated measures, see `multcompare` (`RepeatedMeasuresModel`).

## Multiple Comparisons Using One-Way ANOVA

Load the sample data.

```
load carsmall
```

MPG represents the miles per gallon for each car, and `Cylinders` represents the number of cylinders in each car, either 4, 6, or 8 cylinders.

Test if the mean miles per gallon (mpg) is different across cars that have different numbers of cylinders. Also compute the statistics needed for multiple comparison tests.

```
[p,~,stats] = anova1(MPG,Cylinders,'off');
p
```

```
p =
```

```
4.4902e-24
```

The small  $p$ -value of about 0 is a strong indication that mean miles per gallon is significantly different across cars with different numbers of cylinders.

Perform a multiple comparison test, using the Bonferroni method, to determine which numbers of cylinders make a difference in the performance of the cars.

```
[results,means] = multcompare(stats,'CType','bonferroni')
```

```
results =
```

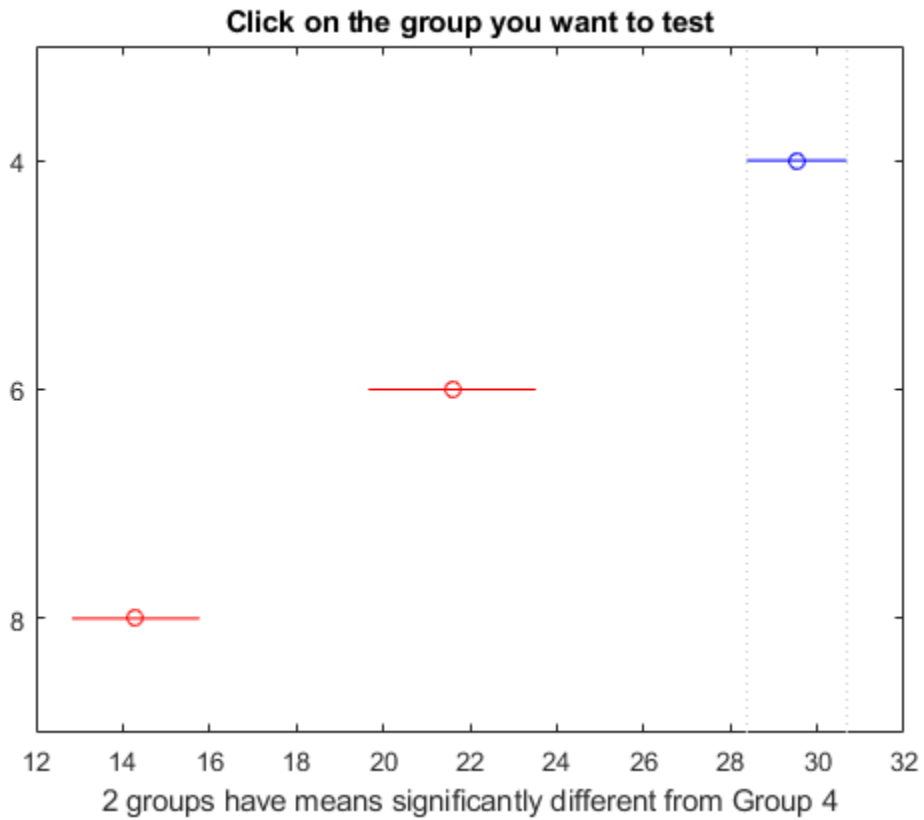
## 9 Analysis of Variance

---

1.0000	2.0000	4.8605	7.9418	11.0230	0.0000
1.0000	3.0000	12.6127	15.2337	17.8548	0.0000
2.0000	3.0000	3.8940	7.2919	10.6899	0.0000

means =

29.5300	0.6363
21.5882	1.0913
14.2963	0.8660



In the `results` matrix, 1, 2, and 3 correspond to cars with 4, 6, and 8 cylinders, respectively. The first two columns show which groups are compared. For example, the first row compares the cars with 4 and 6 cylinders. The fourth column shows the mean mpg difference for the compared groups. The third and fifth columns show the lower and upper limits for a 95% confidence interval for the difference in the group means. The last column shows the  $p$ -values for the tests. All  $p$ -values are zero, which indicates that the mean mpg for all groups differ across all groups.

In the figure the blue bar represents the group of cars with 4 cylinders. The red bars represent the other groups. None of the red comparison intervals for the mean mpg of cars overlap, which means that the mean mpg is significantly different for cars having 4, 6, or 8 cylinders.

The first column of the `means` matrix has the mean mpg estimates for each group of cars. The second column contains the standard errors of the estimates.

## Multiple Comparisons for Three-Way ANOVA

Load the sample data.

```
y = [52.7 57.5 45.9 44.5 53.0 57.0 45.9 44.0]';
g1 = [1 2 1 2 1 2 1 2];
g2 = {'hi'; 'hi'; 'lo'; 'lo'; 'hi'; 'hi'; 'lo'; 'lo'};
g3 = {'may'; 'may'; 'may'; 'may'; 'june'; 'june'; 'june'; 'june'};
```

`y` is the response vector and `g1`, `g2`, and `g3` are the grouping variables (factors). Each factor has two levels, and every observation in `y` is identified by a combination of factor levels. For example, observation `y(1)` is associated with level 1 of factor `g1`, level 'hi' of factor `g2`, and level 'may' of factor `g3`. Similarly, observation `y(6)` is associated with level 2 of factor `g1`, level 'hi' of factor `g2`, and level 'june' of factor `g3`.

Test if the response is the same for all factor levels. Also compute the statistics required for multiple comparison tests.

```
[~,~,stats] = anovan(y, {g1 g2 g3}, 'model', 'interaction', ...
    'varnames', {'g1', 'g2', 'g3'});
```

Figure 1: N-Way ANOVA

File Edit View Insert Tools Desktop Window Help

### Analysis of Variance

Source	Sum Sq.	d. f.	Mean Sq.	F	Prob>F
g1	3.781	1	3.781	336.11	0.0347
g2	199.001	1	199.001	17689	0.0048
g3	0.061	1	0.061	5.44	0.2578
g1*g2	18.301	1	18.301	1626.78	0.0158
g1*g3	0.211	1	0.211	18.78	0.1444
g2*g3	0.011	1	0.011	1	0.5
Error	0.011	1	0.011		
Total	221.379	7			

Constrained (Type III) sums of squares.

The  $p$ -value of 0.2578 indicates that the mean responses for levels 'may' and 'june' of factor  $g_3$  are not significantly different. The  $p$ -value of 0.0347 indicates that the mean responses for levels 1 and 2 of factor  $g_1$  are significantly different. Similarly, the  $p$ -value of 0.0048 indicates that the mean responses for levels 'hi' and 'lo' of factor  $g_2$  are significantly different.

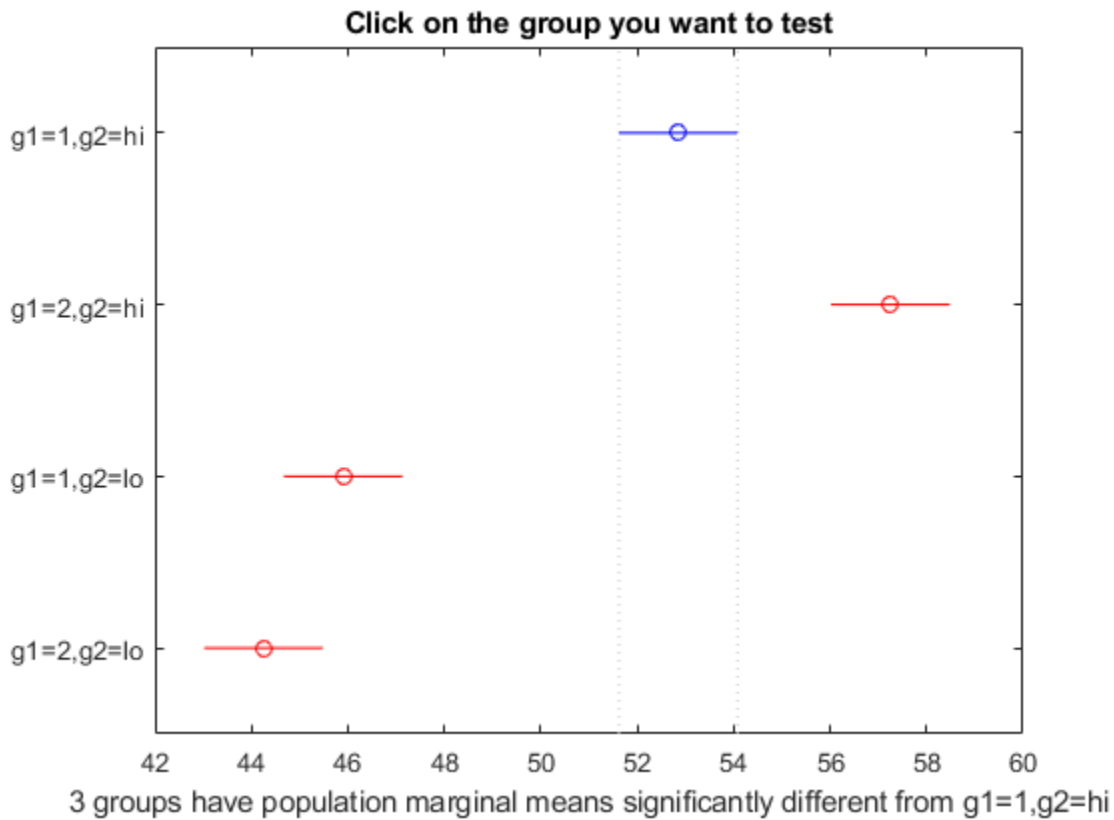
Perform multiple comparison tests to find out which groups of the factors  $g_1$  and  $g_2$  are significantly different.



```
results = multcompare(stats, 'Dimension', [1 2])
```

```
results =
```

1.0000	2.0000	-6.8604	-4.4000	-1.9396	0.0280
1.0000	3.0000	4.4896	6.9500	9.4104	0.0177
1.0000	4.0000	6.1396	8.6000	11.0604	0.0143
2.0000	3.0000	8.8896	11.3500	13.8104	0.0108
2.0000	4.0000	10.5396	13.0000	15.4604	0.0095
3.0000	4.0000	-0.8104	1.6500	4.1104	0.0745



`multcompare` compares the combinations of groups (levels) of the two grouping variables, `g1` and `g2`. In the `results` matrix, the number 1 corresponds to the combination of level 1 of `g1` and level `hi` of `g2`, the number 2 corresponds to the combination of level 2 of `g1` and level `hi` of `g2`. Similarly, the number 3 corresponds to the combination of level 1 of `g1` and level `lo` of `g2`, and the number 4 corresponds to the combination of level 2 of `g1` and level `lo` of `g2`. The last column of the matrix contains the  $p$ -values.

For example, the first row of the matrix shows that the combination of level 1 of `g1` and level `hi` of `g2` has the same mean response values as the combination of level 2 of `g1` and level `hi` of `g2`. The  $p$ -value corresponding to this test is 0.0280, which indicates that the mean responses are significantly different. You can also see this result in the figure. The blue bar shows the comparison interval for the mean response for the combination of level 1 of `g1` and level `hi` of `g2`. The red bars are the comparison intervals for the mean response for other group combinations. None of the red bars overlap with the blue bar, which means the mean response for the combination of level 1 of `g1` and level `hi` of `g2` is significantly different from the mean response for other group combinations.

You can test the other groups by clicking on the corresponding comparison interval for the group. The bar you click on turns to blue. The bars for the groups that are significantly different are red. The bars for the groups that are not significantly different are gray. For example, if you click on the comparison interval for the combination of level 1 of `g1` and level `lo` of `g2`, the comparison interval for the combination of level 2 of `g1` and level `lo` of `g2` overlaps, and is therefore gray. Conversely, the other comparison intervals are red, indicating significant difference.

## Multiple Comparison Procedures

To specify the multiple comparison procedure you want `multcompare` to conduct use the 'CType' name-value pair argument. `multcompare` provides the following procedures:

- “Tukey’s Honestly Significant Difference Procedure” on page 9-33
- “Bonferroni Method” on page 9-33
- “Dunn & Sidák’s Approach” on page 9-33
- “Least Significant Difference” on page 9-34
- “Scheffe’s Procedure” on page 9-35

### Tukey's Honestly Significant Difference Procedure

You can specify Tukey's honestly significant difference procedure using the 'CType', 'Tukey-Kramer' or 'CType', 'hsd' name-value pair argument. The test is based on studentized range distribution. Reject  $H_0: \alpha_i = \alpha_j$  if

$$|t| = \frac{|\bar{y}_i - \bar{y}_j|}{\sqrt{MSE \left( \frac{1}{n_i} + \frac{1}{n_j} \right)}} > \frac{1}{\sqrt{2}} q_{\alpha, k, N-k},$$

where  $q_{\alpha, k, N-k}$  is the upper  $100*(1 - \alpha)$ th percentile of the studentized range distribution with parameter  $k$  and  $N - k$  degrees of freedom.  $k$  is the number of groups (treatments or marginal means) and  $N$  is the total number of observations.

Tukey's honestly significant difference procedure is optimal for balanced one-way ANOVA and similar procedures with equal sample sizes. It has been proven to be conservative for one-way ANOVA with different sample sizes. According to the unproven Tukey-Kramer conjecture, it is also accurate for problems where the quantities being compared are correlated, as in analysis of covariance with unbalanced covariate values.

### Bonferroni Method

You can specify the Bonferroni method using the 'CType', 'bonferroni' name-value pair. This method uses critical values from Student's  $t$ -distribution after an adjustment

to compensate for multiple comparisons. The test rejects  $H_0: \alpha_i = \alpha_j$  at the  $\alpha / 2 \binom{k}{2}$  significance level, where  $k$  is the number of groups if

$$|t| = \frac{|\bar{y}_i - \bar{y}_j|}{\sqrt{MSE \left( \frac{1}{n_i} + \frac{1}{n_j} \right)}} > t_{\alpha / 2 \binom{k}{2}, N-k},$$

where  $N$  is the total number of observations and  $k$  is the number of groups (marginal means). This procedure is conservative, but usually less so than the Scheffé procedure.

### Dunn & Sidák's Approach

You can specify Dunn & Sidák's approach using the 'CType', 'dunn-sidak' name-value pair argument. It uses critical values from the  $t$ -distribution, after an adjustment

for multiple comparisons that was proposed by Dunn and proved accurate by Sidák. This test rejects  $H_0: a_i = a_j$  if

$$|t| = \frac{|\bar{y}_i - \bar{y}_j|}{\sqrt{MSE \left( \frac{1}{n_i} + \frac{1}{n_j} \right)}} > t_{1-\eta/2, v},$$

where

$$\eta = 1 - (1 - \alpha)^{\frac{1}{\binom{k}{2}}}$$

and  $k$  is the number of groups. This procedure is similar to, but less conservative than, the Bonferroni procedure.

### Least Significant Difference

You can specify the least significance difference procedure using the 'CType', 'lsd' name-value pair argument. This test uses the test statistic

$$t = \frac{\bar{y}_i - \bar{y}_j}{\sqrt{MSE \left( \frac{1}{n_i} + \frac{1}{n_j} \right)}}.$$

It rejects  $H_0: a_i = a_j$  if

$$|\bar{y}_i - \bar{y}_j| > \underbrace{t_{\alpha/2, N-k} \sqrt{MSE \left( \frac{1}{n_i} + \frac{1}{n_j} \right)}}_{LSD}.$$

Fisher suggests a protection against multiple comparisons by performing LSD only when the null hypothesis  $H_0: a_1 = a_2 = \dots = a_k$  is rejected by ANOVA  $F$ -test. Even in this case, LSD might not reject any of the individual hypotheses. It is also possible that ANOVA does not reject  $H_0$ , even when there are differences between some group means. This behavior occurs because the equality of the remaining group means can cause the  $F$ -test statistic to be nonsignificant. Without any condition, LSD does not provide any protection against the multiple comparison problem.

## Scheffe's Procedure

You can specify Scheffe's procedure using the 'CType', 'scheffe' name-value pair argument. The critical values are derived from the  $F$  distribution. The test rejects  $H_0: \alpha_i = \alpha_j$  if

$$\frac{|\bar{y}_i - \bar{y}_j|}{\sqrt{MSE \left( \frac{1}{n_i} + \frac{1}{n_j} \right)}} > \sqrt{(k-1) F_{k-1, N-k, \alpha}}$$

This procedure provides a simultaneous confidence level for comparisons of all linear combinations of the means. It is conservative for comparisons of simple differences of pairs.

## References

- [1] Milliken G. A. and D. E. Johnson. *Analysis of Messy Data. Volume I: Designed Experiments*. Boca Raton, FL: Chapman & Hall/CRC Press, 1992.
- [2] Neter J., M. H. Kutner, C. J. Nachtsheim, W. Wasserman. 4th ed. *Applied Linear Statistical Models*. Irwin Press, 1996.
- [3] Hochberg, Y., and A. C. Tamhane. *Multiple Comparison Procedures*. Hoboken, NJ: John Wiley & Sons, 1987.

## See Also

`anova1` | `anova2` | `anovan` | `aocool` | `friedman` | `kruskalwallis` | `multcompare`

## Related Examples

- “Perform One-Way ANOVA” on page 9-6
- “Perform Two-Way ANOVA” on page 9-18

## N-Way ANOVA

### In this section...

“Introduction to N-Way ANOVA” on page 9-36  
 “Prepare Data for N-Way ANOVA” on page 9-38  
 “Perform N-Way ANOVA” on page 9-39

### Introduction to N-Way ANOVA

You can use the Statistics and Machine Learning Toolbox function `anovan` to perform  $N$ -way ANOVA. Use  $N$ -way ANOVA to determine if the means in a set of data differ with respect to groups (levels) of multiple factors. By default, `anovan` treats all grouping variables as fixed effects. For an example of ANOVA with random effects, see “ANOVA with Random Effects” on page 9-49. For repeated measures, see `fitrm` and `ranova`.

$N$ -way ANOVA is a generalization of two-way ANOVA. For three factors, for example, the model can be written as

$$y_{ijk r} = \mu + \alpha_i + \beta_j + \gamma_k + (\alpha\beta)_{ij} + (\alpha\gamma)_{ik} + (\beta\gamma)_{jk} + (\alpha\beta\gamma)_{ijk} + \varepsilon_{ijk r},$$

where

- $y_{ijk r}$  is an observation of the response variable.  $i$  represents group  $i$  of factor  $A$ ,  $i = 1, 2, \dots, I$ ,  $j$  represents group  $j$  of factor  $B$ ,  $j = 1, 2, \dots, J$ ,  $k$  represents group  $k$  of factor  $C$ , and  $r$  represents the replication number,  $r = 1, 2, \dots, R$ . For constant  $R$ , there are a total of  $N = I*J*K*R$  observations, but the number of observations does not have to be the same for each combination of groups of factors.
- $\mu$  is the overall mean.
- $\alpha_i$  are the deviations of groups of factor  $A$  from the overall mean  $\mu$  due to factor  $A$ . The values of  $\alpha_i$  sum to 0.

$$\sum_{i=1}^I \alpha_i = 0.$$

- $\beta_j$  are the deviations of groups in factor  $B$  from the overall mean  $\mu$  due to factor  $B$ . The values of  $\beta_j$  sum to 0.

$$\sum_{j=1}^J \beta_j = 0.$$

- $\gamma_k$  are the deviations of groups in factor  $C$  from the overall mean  $\mu$  due to factor  $C$ . The values of  $\gamma_k$  sum to 0.

$$\sum_{k=1}^K \gamma_k = 0.$$

- $(\alpha\beta)_{ij}$  is the interaction term between factors  $A$  and  $B$ .  $(\alpha\beta)_{ij}$  sum to 0 over either index.

$$\sum_{i=1}^I (\alpha\beta)_{ij} = \sum_{j=1}^J (\alpha\beta)_{ij} = 0.$$

- $(\alpha\gamma)_{ik}$  is the interaction term between factors  $A$  and  $C$ . The values of  $(\alpha\gamma)_{ik}$  sum to 0 over either index.

$$\sum_{i=1}^I (\alpha\gamma)_{ik} = \sum_{k=1}^K (\alpha\gamma)_{ik} = 0.$$

- $(\beta\gamma)_{jk}$  is the interaction term between factors  $B$  and  $C$ . The values of  $(\beta\gamma)_{jk}$  sum to 0 over either index.

$$\sum_{j=1}^J (\beta\gamma)_{jk} = \sum_{k=1}^K (\beta\gamma)_{jk} = 0.$$

- $(\alpha\beta\gamma)_{ijk}$  is the three-way interaction term between factors  $A$ ,  $B$ , and  $C$ . The values of  $(\alpha\beta\gamma)_{ijk}$  sum to 0 over any index.

$$\sum_{i=1}^I (\alpha\beta\gamma)_{ijk} = \sum_{j=1}^J (\alpha\beta\gamma)_{ijk} = \sum_{k=1}^K (\alpha\beta\gamma)_{ijk} = 0.$$

- $\varepsilon_{ijk}$  are the random disturbances. They are assumed to be independent, normally distributed, and have constant variance.

Three-way ANOVA tests hypotheses about the effects of factors  $A$ ,  $B$ ,  $C$ , and their interactions on the response variable  $y$ . The hypotheses about the equality of the mean responses for groups of factor  $A$  are

$$H_0 : \alpha_1 = \alpha_2 \cdots = \alpha_I$$

$$H_1 : \text{at least one } \alpha_i \text{ is different, } i = 1, 2, \dots, I.$$

The hypotheses about the equality of the mean response for groups of factor  $B$  are

$$H_0 : \beta_1 = \beta_2 = \cdots = \beta_J$$

$$H_1 : \text{at least one } \beta_j \text{ is different, } j = 1, 2, \dots, J.$$

The hypotheses about the equality of the mean response for groups of factor  $C$  are

$$H_0 : \gamma_1 = \gamma_2 = \cdots = \gamma_K$$

$H_1$  : at least one  $\gamma_k$  is different,  $k = 1, 2, \dots, K$ .

The hypotheses about the interaction of the factors are

$$H_0 : (\alpha\beta)_{ij} = 0$$

$H_1$  : at least one  $(\alpha\beta)_{ij} \neq 0$

$$H_0 : (\alpha\gamma)_{ik} = 0$$

$H_1$  : at least one  $(\alpha\gamma)_{ik} \neq 0$

$$H_0 : (\beta\gamma)_{jk} = 0$$

$H_1$  : at least one  $(\beta\gamma)_{jk} \neq 0$

$$H_0 : (\alpha\beta\gamma)_{ijk} = 0$$

$H_1$  : at least one  $(\alpha\beta\gamma)_{ijk} \neq 0$

In this notation parameters with two subscripts, such as  $(\alpha\beta)_{ij}$ , represent the interaction effect of two factors. The parameter  $(\alpha\beta\gamma)_{ijk}$  represents the three-way interaction. An ANOVA model can have the full set of parameters or any subset, but conventionally it does not include complex interaction terms unless it also includes all simpler terms for those factors. For example, one would generally not include the three-way interaction without also including all two-way interactions.

## Prepare Data for N-Way ANOVA

Unlike `anova1` and `anova2`, `anova` does not expect data in a tabular form. Instead, it expects a vector of response measurements and a separate vector (or text array) containing the values corresponding to each factor. This input data format is more convenient than matrices when there are more than two factors or when the number of measurements per factor combination is not constant.



$$\begin{array}{cccccccc}
 y & = & [ & y_1, & y_2, & y_3, & y_4, & y_5, & \dots, & y_N & ] \\
 & & & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & & \uparrow & \\
 g1 & = & \{ & 'A', & 'A', & 'C', & 'B', & 'B', & \dots, & 'D' & \} \\
 g2 & = & [ & 1 & 2 & 1 & 3 & 1 & \dots, & 2 & ] \\
 g3 & = & \{ & 'hi', & 'mid', & 'low', & 'mid', & 'hi', & \dots, & 'low' & \}
 \end{array}$$

## Perform N-Way ANOVA

This example shows how to perform N-way ANOVA on car data with mileage and other information on 406 cars made between 1970 and 1982.

Load the sample data.

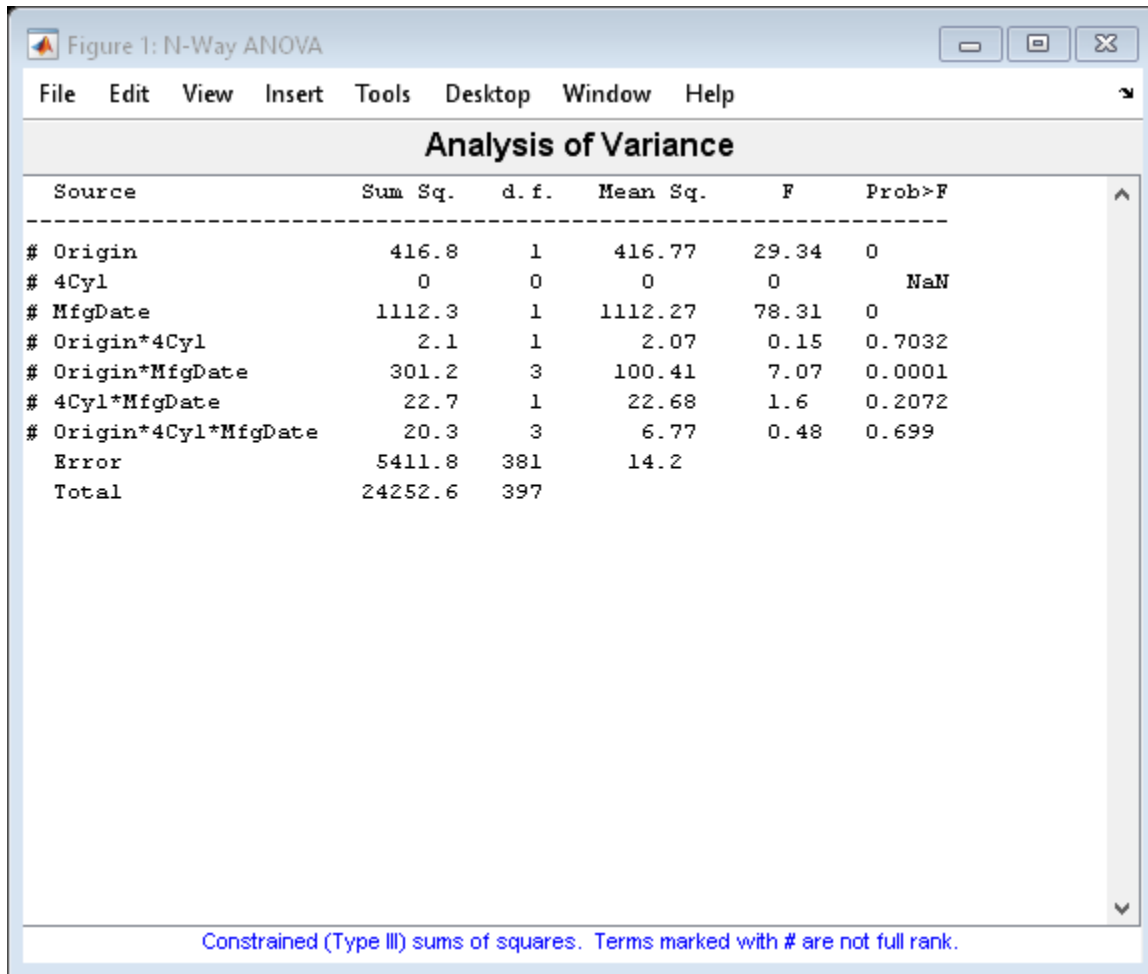
```
load carbig
```

The example focusses on four variables. MPG is the number of miles per gallon for each of 406 cars (though some have missing values coded as NaN). The other three variables are factors: `cyl4` (four-cylinder car or not), `org` (car originated in Europe, Japan, or the USA), and `when` (car was built early in the period, in the middle of the period, or late in the period).

Fit the full model, requesting up to three-way interactions and Type 3 sums-of-squares.

```
varnames = {'Origin'; '4Cyl'; 'MfgDate'};
anovan(MPG, {org cyl4 when}, 3, 3, varnames)
```

```
ans =
    0.0000
      NaN
    0.0000
    0.7032
    0.0001
    0.2072
    0.6990
```



Note that many terms are marked by a # symbol as not having full rank, and one of them has zero degrees of freedom and is missing a  $p$ -value. This can happen when there are missing factor combinations and the model has higher-order terms. In this case, the cross-tabulation below shows that there are no cars made in Europe during the early part of the period with other than four cylinders, as indicated by the 0 in `tbl(2, 1, 1)`.

```
[tbl,chi2,p,factorvals] = crosstab(org,when,cyl4)
```

```

tbl(:, :, 1) =
      82    75    25
       0     4     3
       3     3     4

tbl(:, :, 2) =
      12    22    38
      23    26    17
      12    25    32

chi2 =
      207.7689

p =
      8.0973e-38

factorvals =
      3x3 cell array

      {'USA'   }      {'Early'}      {'Other'   }
      {'Europe'}      {'Mid'   }      {'Four'    }
      {'Japan' }      {'Late' }      {0x0 double}

```

Consequently it is impossible to estimate the three-way interaction effects, and including the three-way interaction term in the model makes the fit singular.

Using even the limited information available in the ANOVA table, you can see that the three-way interaction has a  $p$ -value of 0.699, so it is not significant.

Examine only two-way interactions.

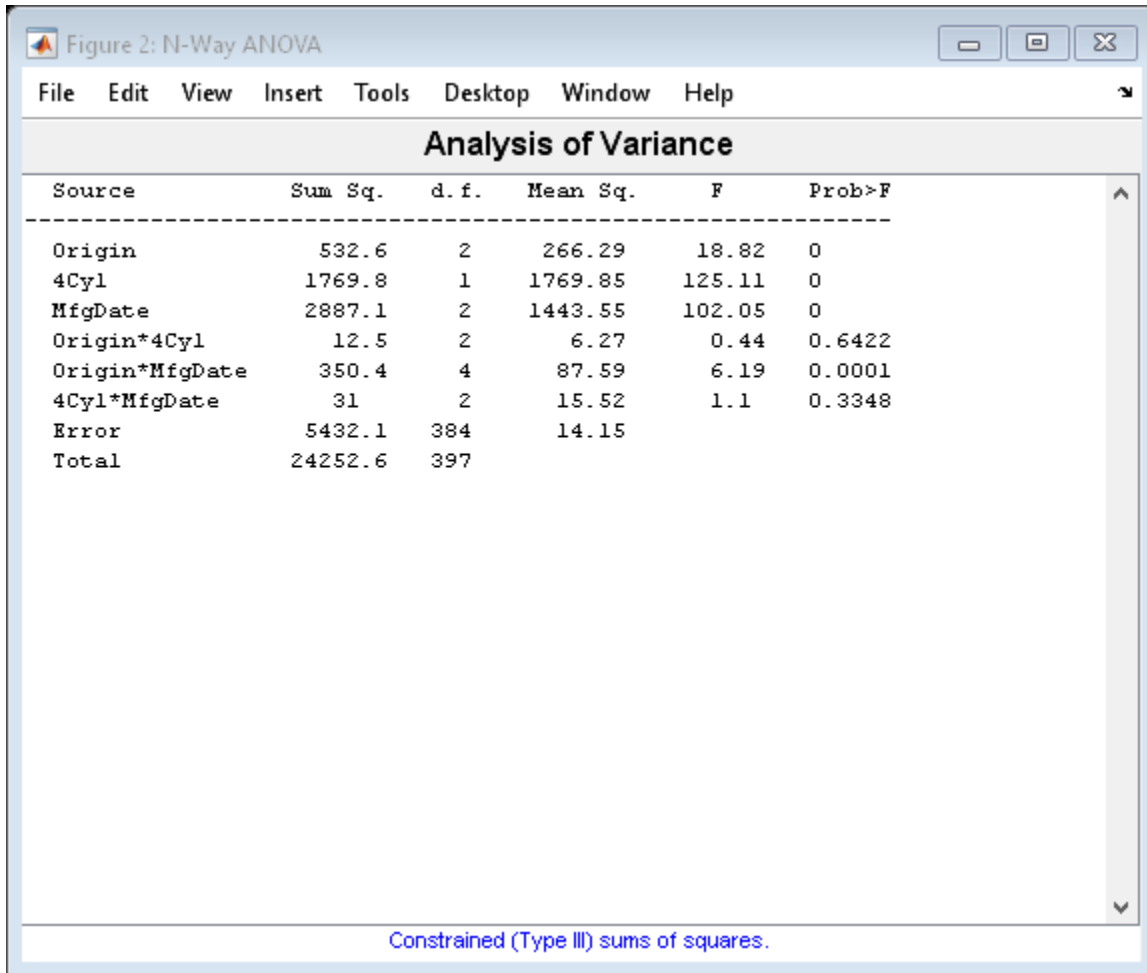
```

[p, tbl2, stats, terms] = anovan(MPG, {org cyl4 when}, 2, 3, varnames);
terms

```

terms =

1	0	0
0	1	0
0	0	1
1	1	0
1	0	1
0	1	1



Now all terms are estimable. The  $p$ -values for interaction term 4 (Origin\*4Cyl) and interaction term 6 (4Cyl\*MfgDate) are much larger than a typical cutoff value of 0.05, indicating these terms are not significant. You could choose to omit these terms and pool their effects into the error term. The output `terms` variable returns a matrix of codes, each of which is a bit pattern representing a term.

Omit terms from the model by deleting their entries from `terms`.

```
terms([4 6], :) = []
```

```
terms =
```

```
    1    0    0
    0    1    0
    0    0    1
    1    0    1
```

Run `anovan` again, this time supplying the resulting vector as the model argument. Also return the statistics required for multiple comparisons of factors.

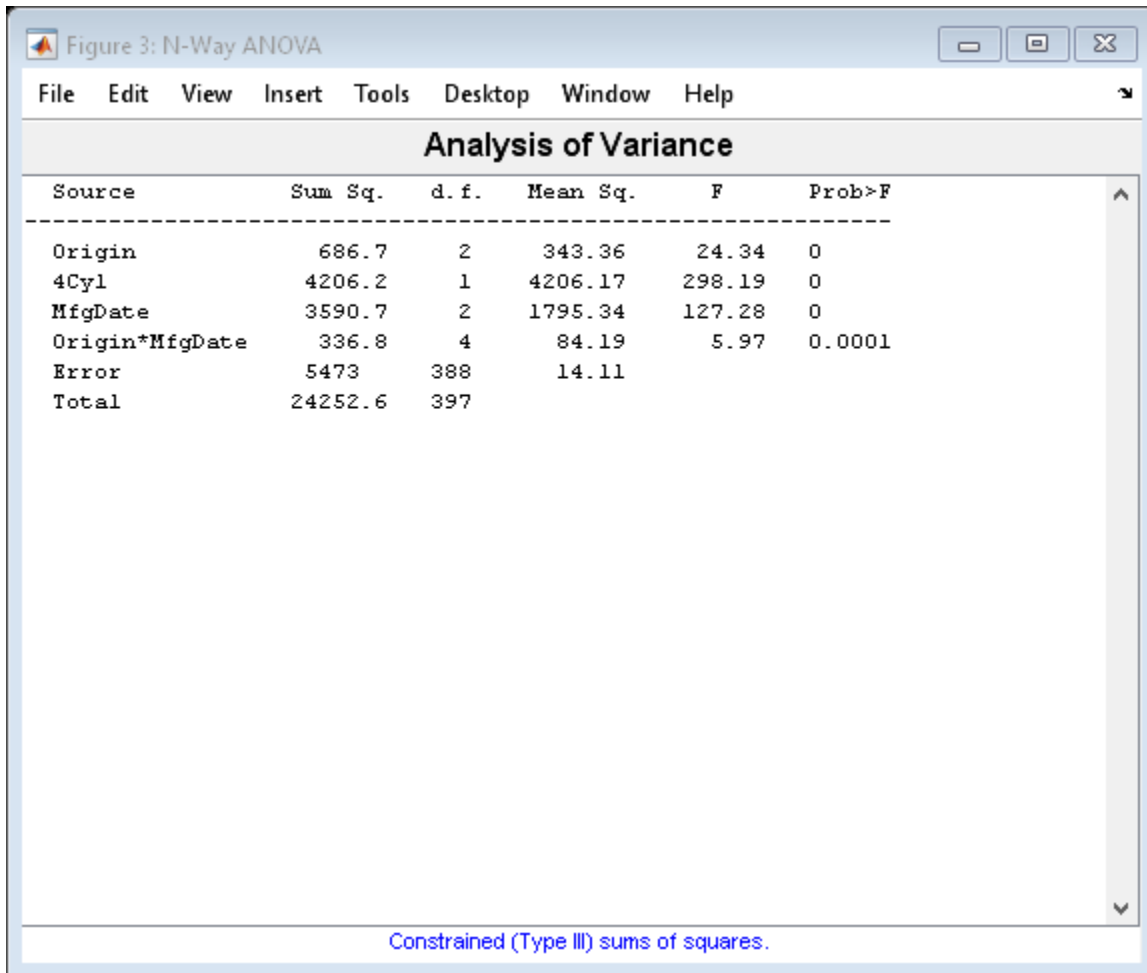
```
[~,~,stats] = anovan(MPG,{org cyl4 when},terms,3,varnames)
```

```
stats =
```

```
struct with fields:
```

```
    source: 'anovan'
    resid: [1x406 double]
    coeffs: [18x1 double]
    Rtr: [10x10 double]
    rowbasis: [10x18 double]
    dfe: 388
    mse: 14.1056
    nullproject: [18x10 double]
    terms: [4x3 double]
    nlevels: [3x1 double]
    continuous: [0 0 0]
    vmeans: [3x1 double]
    termcols: [5x1 double]
    coeffnames: {18x1 cell}
    vars: [18x3 double]
    varnames: {3x1 cell}
    grpnames: {3x1 cell}
    vnested: []
    ems: []
    denom: []
    dfdenom: []
    msdenom: []
    varest: []
    varci: []
    txtdenom: []
    txtems: []
```

```
rtnames: []
```



Now you have a more parsimonious model indicating that the mileage of these cars seems to be related to all three factors, and that the effect of the manufacturing date depends on where the car was made.

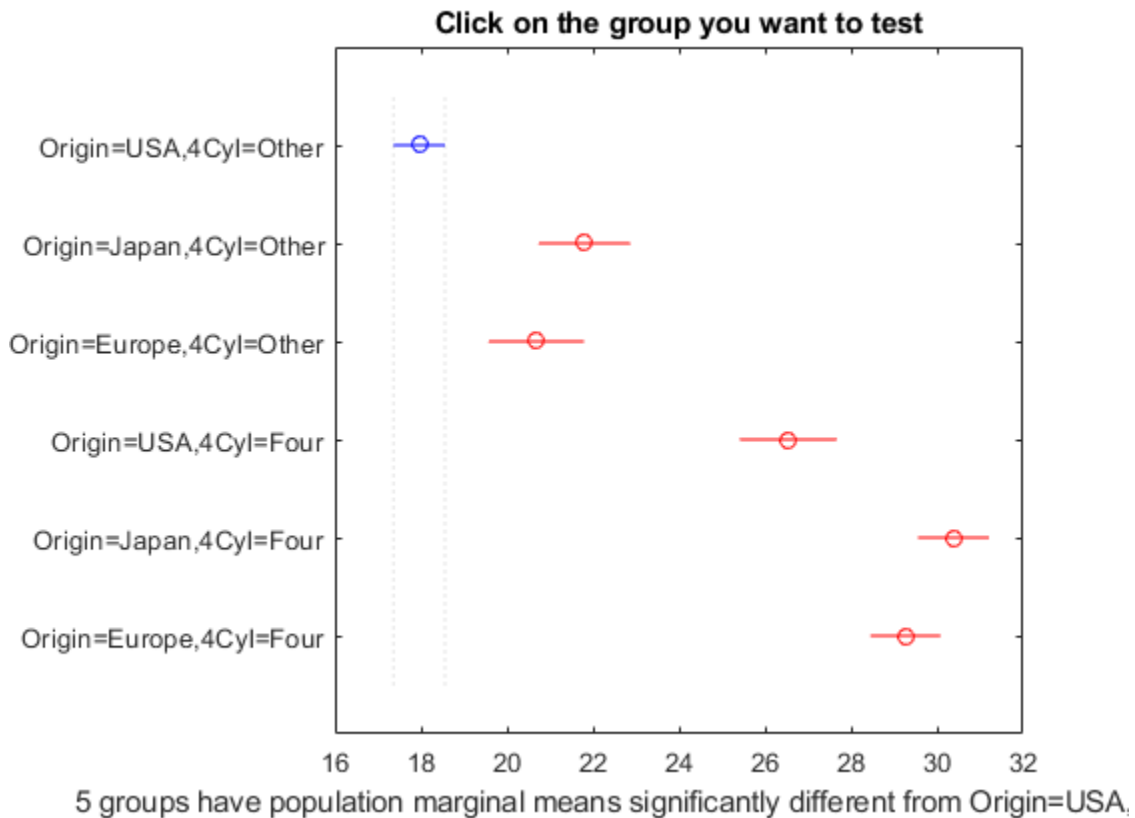
Perform multiple comparisons for Origin and Cylinder.

```
results = multcompare(stats, 'Dimension', [1,2])
```

results =

1.0000	2.0000	-5.4891	-3.8412	-2.1932	0.0000
1.0000	3.0000	-4.4146	-2.7251	-1.0356	0.0001
1.0000	4.0000	-9.9992	-8.5828	-7.1664	0.0000
1.0000	5.0000	-14.0237	-12.4240	-10.8242	0.0000
1.0000	6.0000	-12.8980	-11.3080	-9.7180	0.0000
2.0000	3.0000	-0.7171	1.1160	2.9492	0.5085
2.0000	4.0000	-7.3655	-4.7417	-2.1179	0.0000
2.0000	5.0000	-9.9992	-8.5828	-7.1664	0.0000
2.0000	6.0000	-9.7464	-7.4668	-5.1872	0.0000
3.0000	4.0000	-8.5396	-5.8577	-3.1757	0.0000
3.0000	5.0000	-12.0518	-9.6988	-7.3459	0.0000
3.0000	6.0000	-9.9992	-8.5828	-7.1664	0.0000
4.0000	5.0000	-5.4891	-3.8412	-2.1932	0.0000
4.0000	6.0000	-4.4146	-2.7251	-1.0356	0.0001
5.0000	6.0000	-0.7171	1.1160	2.9492	0.5085





## See Also

`anova1` | `anovan` | `kruskalwallis` | `multcompare`

## Related Examples

- “ANOVA with Random Effects” on page 9-49

## More About

- “One-Way ANOVA” on page 9-3

- “Two-Way ANOVA” on page 9-15
- “Multiple Comparisons” on page 9-26
- “Nonparametric Methods” on page 9-69

## ANOVA with Random Effects

This example shows how to use `anovan` to fit models where a factor's levels represent a random selection from a larger (infinite) set of possible levels.

In an ordinary ANOVA model, each grouping variable represents a fixed factor. The levels of that factor are a fixed set of values. The goal is to determine whether different factor levels lead to different response values.

### Set Up the Model

Load the sample data.

```
load mileage
```

The `anova2` function works only with balanced data, and it infers the values of the grouping variables from the row and column numbers of the input matrix. The `anovan` function, on the other hand, requires you to explicitly create vectors of grouping variable values. Create these vectors in the following way.

Create an array indicating the factory for each value in `mileage`. This array is 1 for the first column, 2 for the second, and 3 for the third.

```
factory = repmat(1:3,6,1);
```

Create an array indicating the car model for each `mileage` value. This array is 1 for the first three rows of `mileage`, and 2 for the remaining three rows.

```
carmod = [ones(3,3); 2*ones(3,3)];
```

Turn these matrices into vectors and display them.

```
mileage = mileage(:);
factory = factory(:);
carmod = carmod(:);
[mileage factory carmod]
```

```
ans =
```

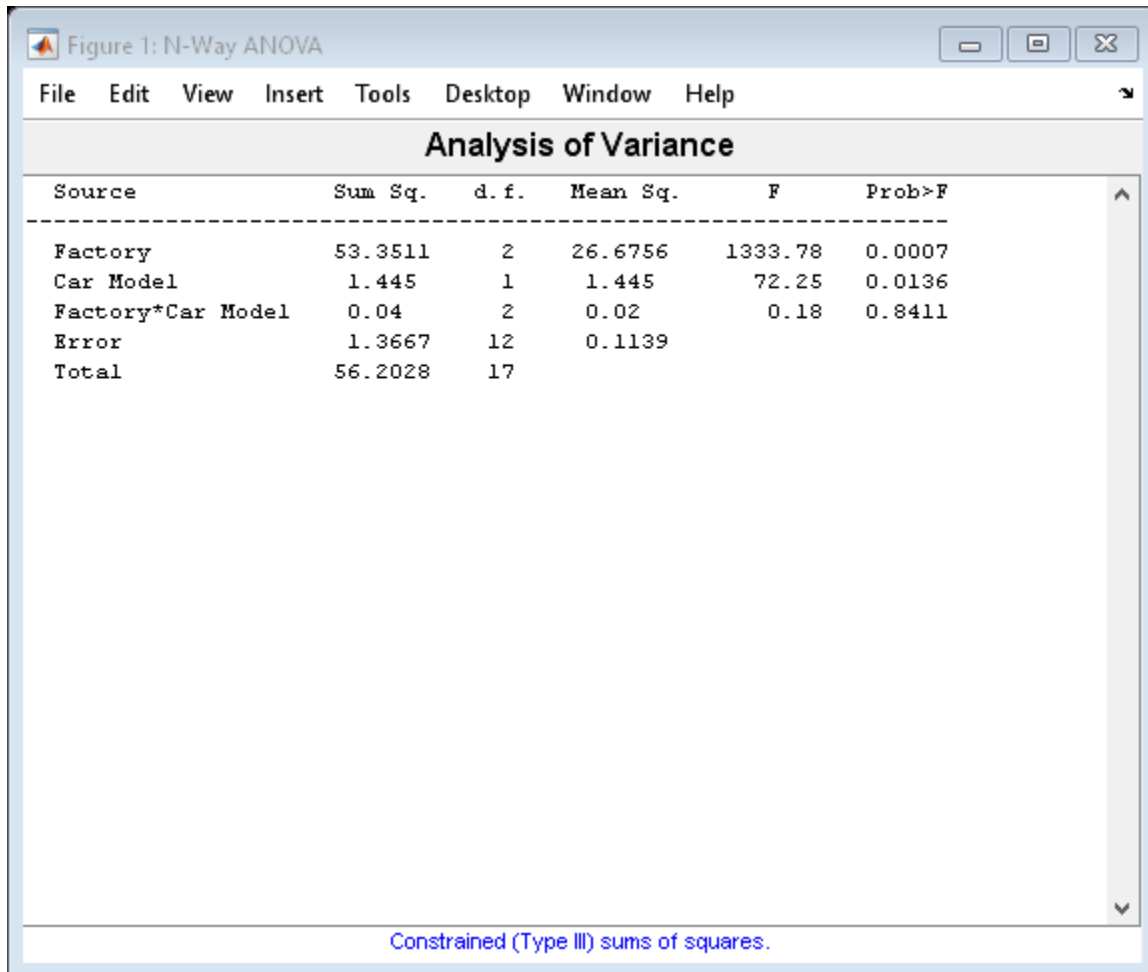
```
33.3000    1.0000    1.0000
33.4000    1.0000    1.0000
32.9000    1.0000    1.0000
```

32.6000	1.0000	2.0000
32.5000	1.0000	2.0000
33.0000	1.0000	2.0000
34.5000	2.0000	1.0000
34.8000	2.0000	1.0000
33.8000	2.0000	1.0000
33.4000	2.0000	2.0000
33.7000	2.0000	2.0000
33.9000	2.0000	2.0000
37.4000	3.0000	1.0000
36.8000	3.0000	1.0000
37.6000	3.0000	1.0000
36.6000	3.0000	2.0000
37.0000	3.0000	2.0000
36.7000	3.0000	2.0000

### Fit a Random Effects Model

Suppose you are studying a few factories but you want information about what would happen if you build these same car models in a different factory, either one that you already have or another that you might construct. To get this information, fit the analysis of variance model, specifying a model that includes an interaction term and that the factory factor is random.

```
[pvals,tbl,stats] = anovan(mileage, {factory carmod}, ...  
'model',2, 'random',1,'varnames',{'Factory' 'Car Model'});
```



In the fixed effects version of this fit, which you get by omitting the inputs 'random', 1 in the preceding code, the effect of car model is significant, with a  $p$ -value of 0.0039. But in this example, which takes into account the random variation of the effect of the variable 'Car Model' from one factory to another, the effect is still significant, but with a higher  $p$ -value of 0.0136.

### F-Statistics for Models with Random Effects

The  $F$ -statistic in a model having random effects is defined differently than in a model having all fixed effects. In the fixed effects model, you compute the  $F$ -statistic for any term by taking the ratio of the mean square for that term with the mean square for error. In a random effects model, however, some  $F$ -statistics use a different mean square in the denominator.

In the example described in **Set Up the Model**, the effect of the variable 'Factory' could vary across car models. In this case, the interaction mean square takes the place of the error mean square in the  $F$ -statistic.

Find the  $F$ -statistic.

$$F = 26.6756 / 0.02$$

F =

$$1.3338e+03$$

The degrees of freedom for the statistic are the degrees of freedom for the numerator (2) and denominator (2) mean squares.

Find the  $p$ -value.

$$pval = 1 - fcdf(F, 2, 2)$$

pval =

$$7.4919e-04$$

With random effects, the expected value of each mean square depends not only on the variance of the error term, but also on the variances contributed by the random effects. You can see these dependencies by writing the expected values as linear combinations of contributions from the various model terms.

Find the coefficients of these linear combinations.

stats.ems

```
ans =

    6.0000    0.0000    3.0000    1.0000
    0.0000    9.0000    3.0000    1.0000
    0.0000    0.0000    3.0000    1.0000
         0         0         0         1.0000
```

This returns the `ems` field of the `stats` structure.

Display text representations of the linear combinations.

```
stats.txtems
```

```
ans =

4x1 cell array

    {'6*V(Factory)+3*V(Factory*Car Model)+V(Error) ' }
    {'9*Q(Car Model)+3*V(Factory*Car Model)+V(Error) ' }
    {'3*V(Factory*Car Model)+V(Error) ' }
    {'V(Error) ' }
```

The expected value for the mean square due to car model (second term) includes contributions from a quadratic function of the car model effects, plus three times the variance of the interaction term's effect, plus the variance of the error term. Notice that if the car model effects were all zero, the expression would reduce to the expected mean square for the third term (the interaction term). That is why the  $F$ -statistic for the car model effect uses the interaction mean square in the denominator.

In some cases there is no single term whose expected value matches the one required for the denominator of the  $F$ -statistic. In that case, the denominator is a linear combination of mean squares. The `stats` structure contains fields giving the definitions of the denominators for each  $F$ -statistic. The `txtdenom` field, `stats.txtdenom`, contains a text representation, and the `denom` field contains a matrix that defines a linear combination of the variances of terms in the model. For balanced models like this one, the `denom` matrix, `stats.denom`, contains zeros and ones, because the denominator is just a single term's mean square.

Display the `txtdenom` field.

```
stats.txtdenom

ans =

3x1 cell array

    {'MS(Factory*Car Model)'}
    {'MS(Factory*Car Model)'}
    {'MS(Error)'}          }
```

Display the denom field.

```
stats.denom

ans =

    0.0000    1.0000         0
    0.0000    1.0000   -0.0000
   -0.0000    0.0000    1.0000
```

### Variance Components

For the model described in **Set Up the Model**, consider the mileage for a particular car of a particular model made at a random factory. The variance of that car is the sum of components, or contributions, one from each of the random terms.

Display the names of the random terms.

```
stats.rtnames

ans =

3x1 cell array

    {'Factory'}
    {'Factory*Car Model'}
    {'Error'}          }
```

You do not know the variances, but you can estimate them from the data. Recall that the `ems` field of the `stats` structure expresses the expected value of each term's mean square



as a linear combination of unknown variances for random terms, and unknown quadratic forms for fixed terms. If you take the expected mean square expressions for the random terms, and equate those expected values to the computed mean squares, you get a system of equations that you can solve for the unknown variances. These solutions are the variance component estimates.

Display the variance component estimate for each term.

```
stats.varest
```

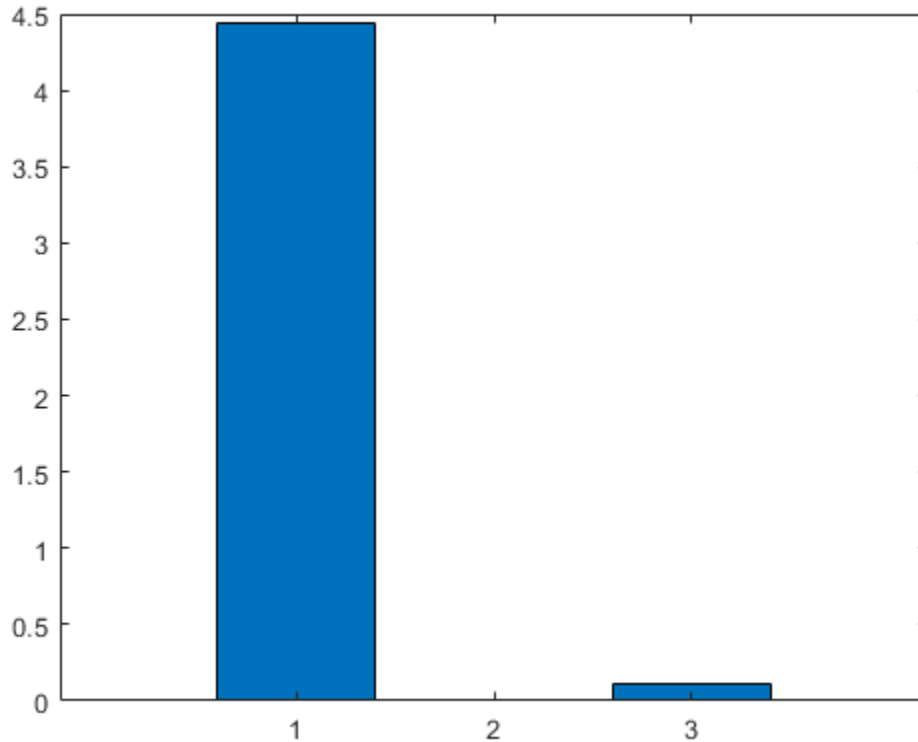
```
ans =
```

```
  4.4426  
 -0.0313  
  0.1139
```

Under some conditions, the variability attributed to a term is unusually low, and that term's variance component estimate is negative. In those cases it is common to set the estimate to zero, which you might do, for example, to create a bar graph of the components.

Create a bar graph of the components.

```
bar(max(0,stats.varest))  
gca.xtick = 1:3;  
gca.xticklabel = stats.rtnames;
```



You can also compute confidence bounds for the variance estimate. The `anovan` function does this by computing confidence bounds for the variance expected mean squares, and finding lower and upper limits on each variance component containing all of these bounds. This procedure leads to a set of bounds that is conservative for balanced data. (That is, 95% confidence bounds will have a probability of at least 95% of containing the true variances if the number of observations for each combination of grouping variables is the same.) For unbalanced data, these are approximations that are not guaranteed to be conservative.

Display the variance estimates and the confidence limits for the variance estimates of each component.

```
[{'Term' 'Estimate' 'Lower' 'Upper'};  
 stats.rtnames, num2cell([stats.varest stats.varci])]
```

```
ans =
```

```
4x4 cell array
```

{'Term' }	{'Estimate' }	{'Lower' }	{'Upper' }
{'Factory' }	{[ 4.4426]}	{[1.0736]}	{[175.6038]}
{'Factory*Car Model' }	{[ -0.0313]}	{[ NaN]}	{[ NaN]}
{'Error' }	{[ 0.1139]}	{[0.0586]}	{[ 0.3103]}

## Other ANOVA Models

The `anovan` function also has arguments that enable you to specify two other types of model terms:

- `'nested'` argument specifies a matrix that indicates which factors are nested within other factors. A nested factor is one that takes different values within each level its nested factor.

Suppose an automobile company has three factories, and each factory makes two car models. The gas mileage in the cars can vary from factory to factory and from model to model. These two factors, factory and model, explain the differences in mileage, that is, the response. One measure of interest is the difference in mileage due to the production methods between factories. Another measure of interest is the difference in the mileage of the models (irrespective of the factory) due to different design specifications. Suppose also that, each factory produces distinct car models for a total of six car models. Then, the car model is nested in factory.

Factory	Car Model
1	1
1	2
2	3
2	4
3	5
3	6

It is also common with nested models to number the nested factor the same way in each nested factor.

- `'continuous'` argument specifies that some factors are to be treated as continuous variables. The remaining factors are categorical variables. Although the `anovan` function can fit models with multiple continuous and categorical predictors, the simplest model that combines one predictor of each type is known as an *analysis of covariance* model. “Analysis of Covariance” on page 9-60 describes a specialized tool for fitting this model.

## See Also

`anova1` | `anova2` | `anovan` | `kruskalwallis` | `multcompare`

## **Related Examples**

- “ANOVA with Random Effects” on page 9-49

## **More About**

- “One-Way ANOVA” on page 9-3
- “Two-Way ANOVA” on page 9-15
- “Multiple Comparisons” on page 9-26
- “N-Way ANOVA” on page 9-36
- “Nonparametric Methods” on page 9-69

## Analysis of Covariance

### In this section...

“Introduction to Analysis of Covariance” on page 9-60

“Analysis of Covariance Tool” on page 9-60

“Confidence Bounds” on page 9-64

“Multiple Comparisons” on page 9-66

### Introduction to Analysis of Covariance

Analysis of covariance is a technique for analyzing grouped data having a response ( $y$ , the variable to be predicted) and a predictor ( $x$ , the variable used to do the prediction). Using analysis of covariance, you can model  $y$  as a linear function of  $x$ , with the coefficients of the line possibly varying from group to group.

### Analysis of Covariance Tool

The `aoctool` function opens an interactive graphical environment for fitting and prediction with analysis of covariance (ANOCOVA) models. It fits the following models for the  $i$ th group:

Same mean  $y = a + \varepsilon$

Separate means  $y = (a + a_i) + \varepsilon$

Same line  $y = a + \beta x + \varepsilon$

Parallel lines  $y = (a + a_i) + \beta x + \varepsilon$

Separate lines  $y = (a + a_i) + (\beta + \beta_i)x + \varepsilon$

For example, in the parallel lines model the intercept varies from one group to the next, but the slope is the same for each group. In the same mean model, there is a common intercept and no slope. In order to make the group coefficients well determined, the tool imposes the constraints

$$\sum \alpha_j = \sum \beta_j = 0$$

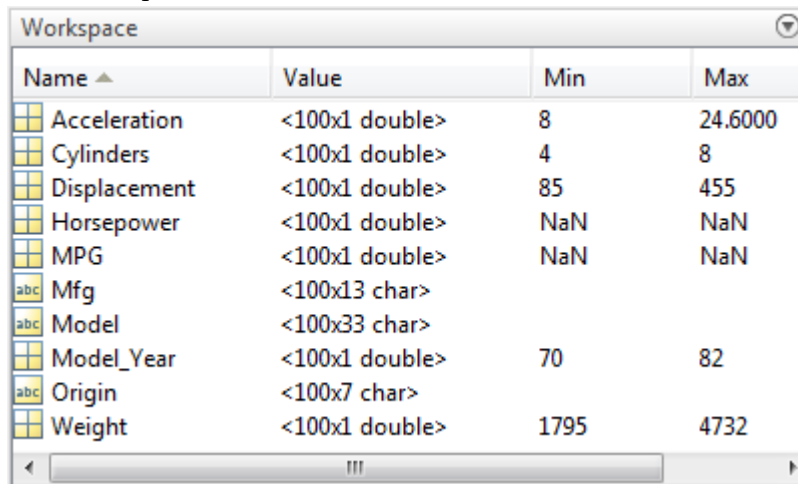
The following steps describe the use of `aoctool`.

- 1 Load the data.** The Statistics and Machine Learning Toolbox data set `carsmall.mat` contains information on cars from the years 1970, 1976, and 1982.

This example studies the relationship between the weight of a car and its mileage, and whether this relationship has changed over the years. To start the demonstration, load the data set.

```
load carsmall
```

The Workspace Browser shows the variables in the data set.



Name ▲	Value	Min	Max
Acceleration	<100x1 double>	8	24.6000
Cylinders	<100x1 double>	4	8
Displacement	<100x1 double>	85	455
Horsepower	<100x1 double>	NaN	NaN
MPG	<100x1 double>	NaN	NaN
Mfg	<100x13 char>		
Model	<100x33 char>		
Model_Year	<100x1 double>	70	82
Origin	<100x7 char>		
Weight	<100x1 double>	1795	4732

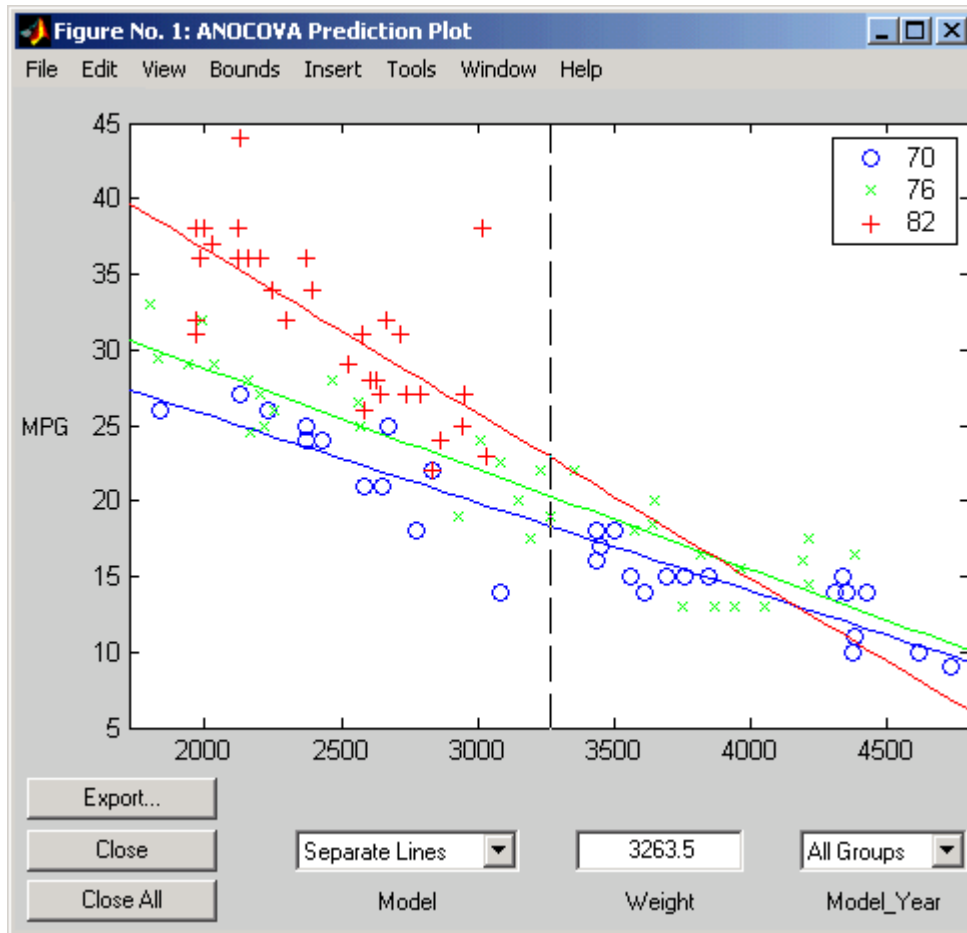
You can also use `aocool` with your own data.

- 2 Start the tool.** The following command calls `aocool` to fit a separate line to the column vectors `Weight` and `MPG` for each of the three model group defined in `Model_Year`. The initial fit models the  $y$  variable, `MPG`, as a linear function of the  $x$  variable, `Weight`.

```
[h,atab,ctab,stats] = aocool(Weight,MPG,Model_Year);
```

See the `aocool` function reference page for detailed information about calling `aocool`.

- 3 Examine the output.** The graphical output consists of a main window with a plot, a table of coefficient estimates, and an analysis of variance table. In the plot, each `Model_Year` group has a separate line. The data points for each group are coded with the same color and symbol, and the fit for each group has the same color as the data points.



The coefficients of the three lines appear in the figure titled ANCOVA Coefficients. You can see that the slopes are roughly  $-0.0078$ , with a small deviation for each group:

- Model year 1970:  $y = (45.9798 - 8.5805) + (-0.0078 + 0.002)x + \varepsilon$
- Model year 1976:  $y = (45.9798 - 3.8902) + (-0.0078 + 0.0011)x + \varepsilon$
- Model year 1982:  $y = (45.9798 + 12.4707) + (-0.0078 - 0.0031)x + \varepsilon$



**Figure No. 3: ANCOVA Coefficients**

File Edit View Insert Tools Window Help

**Coefficient Estimates**

Term	Estimate	Std. Err.	T	Prob> T
Intercept	45.9798	1.52085	30.23	0
70	-8.5805	1.96186	-4.37	0
76	-3.8902	1.86864	-2.08	0.0403
82	12.4707	2.5568	4.88	0
Slope	-0.0078	0.00056	-14	0
70	0.002	0.00066	2.96	0.0039
76	0.0011	0.00065	1.74	0.0849
82	-0.0031	0.001	-3.1	0.0026

Because the three fitted lines have slopes that are roughly similar, you may wonder if they really are the same. The `Model_Year*Weight` interaction expresses the difference in slopes, and the ANOVA table shows a test for the significance of this term. With an  $F$  statistic of 5.23 and a  $p$  value of 0.0072, the slopes are significantly different.

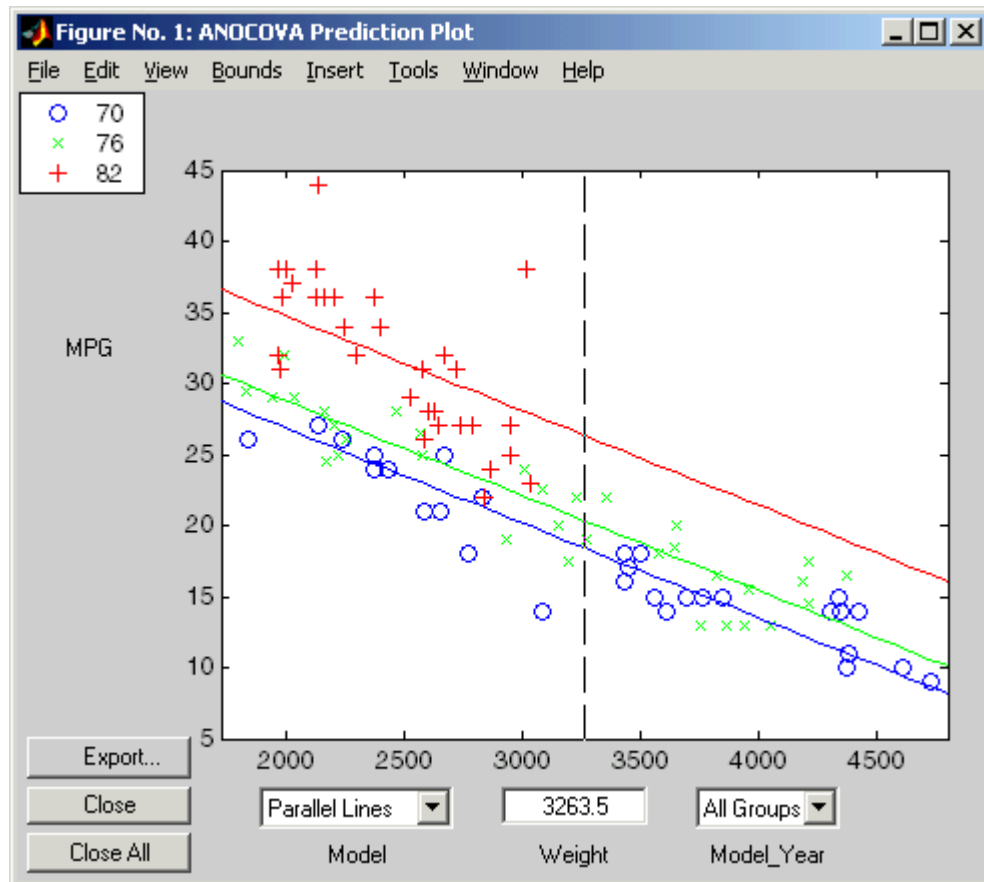
**Figure No. 2: ANCOVA Test Results**

File Edit View Insert Tools Window Help

**ANOVA Table**

Source	d.f	Sum Sq	Mean Sq	F	Prob>F
Model_Year	2	807.69	403.84	51.98	0
Weight	1	2050.2	2050.2	263.87	0
Model_Year*Weigh	2	81.22	40.61	5.23	0.0072
Error	88	683.74	7.77		

- 4 Constrain the slopes to be the same.** To examine the fits when the slopes are constrained to be the same, return to the ANCOVA Prediction Plot window and use the **Model** pop-up menu to select a **Parallel Lines** model. The window updates to show the following graph.

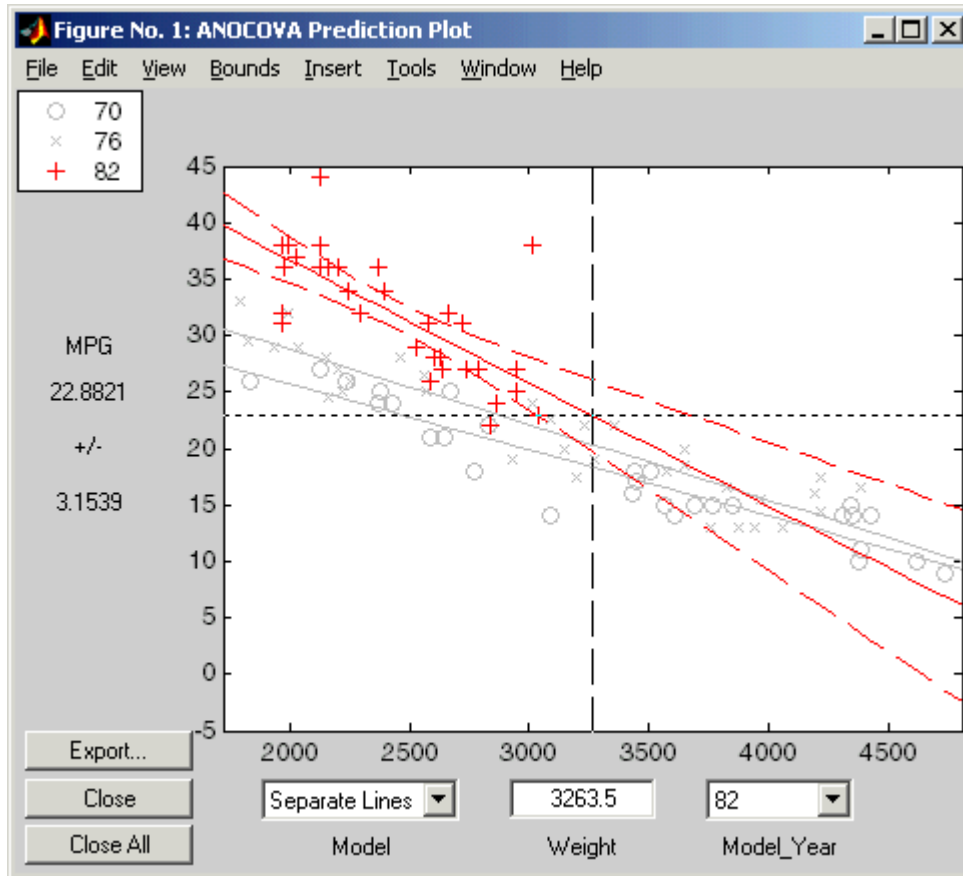


Though this fit looks reasonable, it is significantly worse than the `Separate Lines` model. Use the **Model** pop-up menu again to return to the original model.

## Confidence Bounds

The example in “Analysis of Covariance Tool” on page 9-60 provides estimates of the relationship between MPG and Weight for each Model\_Year, but how accurate are these estimates? To find out, you can superimpose confidence bounds on the fits by examining them one group at a time.

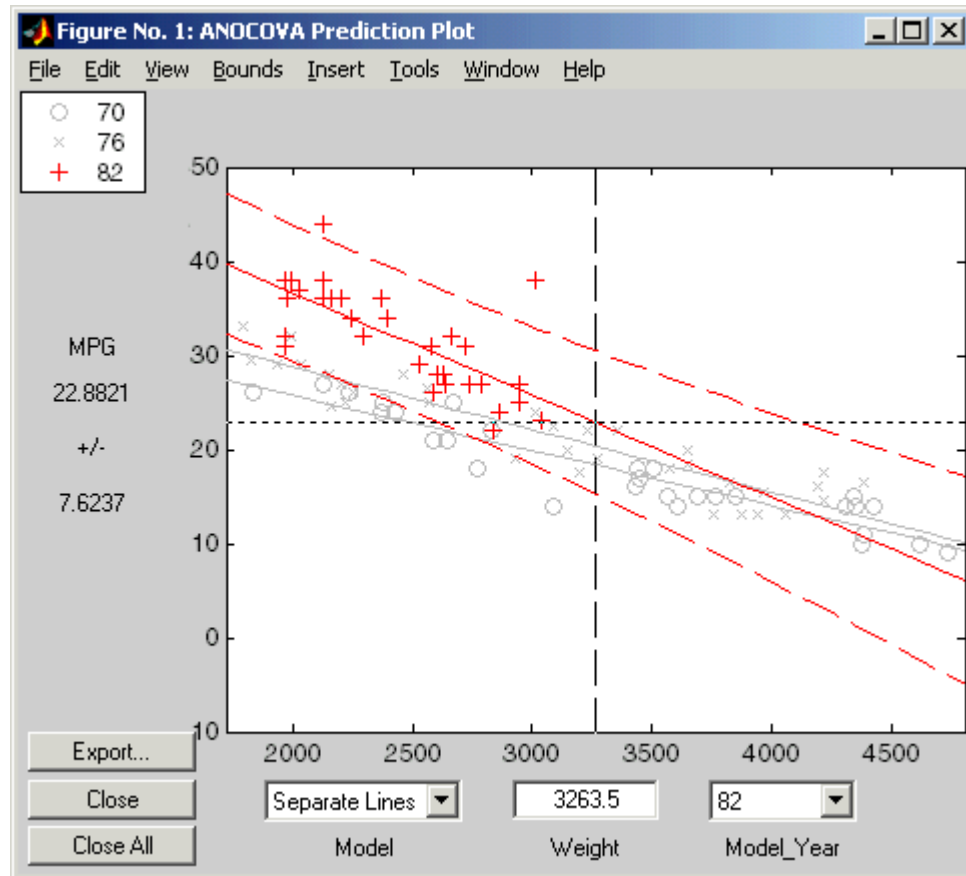
- 1 In the **Model\_Year** menu at the lower right of the figure, change the setting from All Groups to 82. The data and fits for the other groups are dimmed, and confidence bounds appear around the 82 fit.



The dashed lines form an envelope around the fitted line for model year 82. Under the assumption that the true relationship is linear, these bounds provide a 95% confidence region for the true line. Note that the fits for the other model years are well outside these confidence bounds for Weight values between 2000 and 3000.

- 2 Sometimes it is more valuable to be able to predict the response value for a new observation, not just estimate the average response value. Use the `aoctool` function **Bounds** menu to change the definition of the confidence bounds from Line to

Observation. The resulting wider intervals reflect the uncertainty in the parameter estimates as well as the randomness of a new observation.



Like the `polytool` function, the `aocool` function has cross hairs that you can use to manipulate the `Weight` and watch the estimate and confidence bounds along the `y`-axis update. These values appear only when a single group is selected, not when All Groups is selected.

## Multiple Comparisons

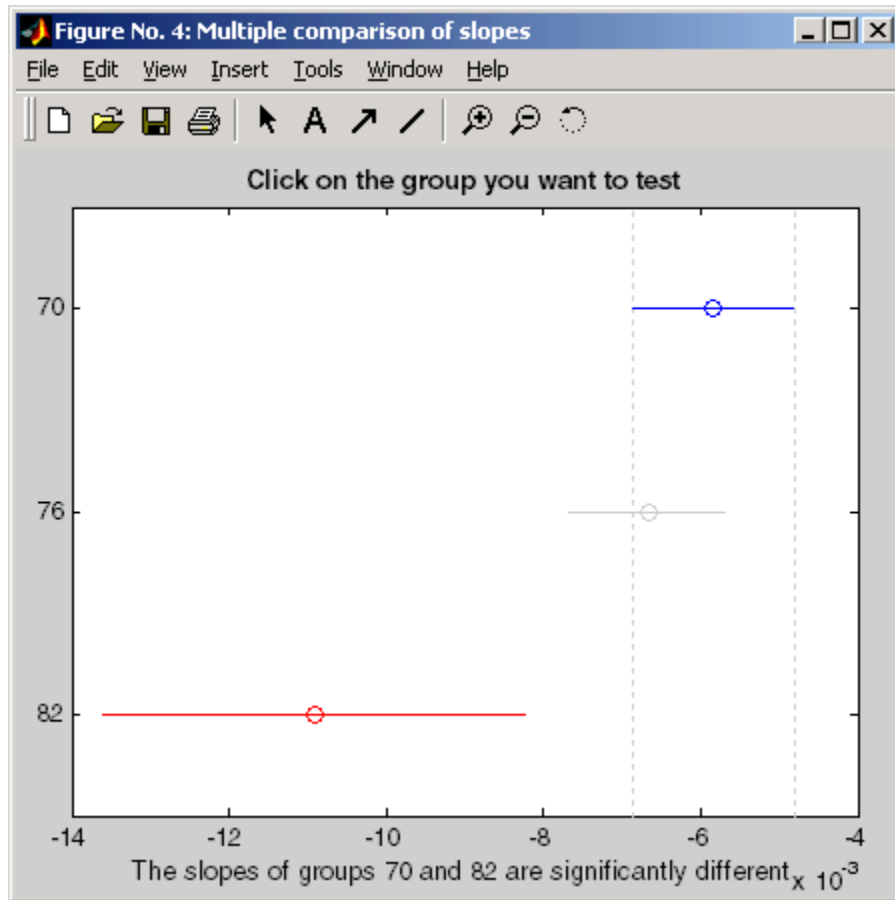
You can perform a multiple comparison test by using the `stats` output structure from `aocool` as input to the `multcompare` function. The `multcompare` function can test

either slopes, intercepts, or population marginal means (the predicted MPG of the mean weight for each group). The example in “Analysis of Covariance Tool” on page 9-60 shows that the slopes are not all the same, but could it be that two are the same and only the other one is different? You can test that hypothesis.

```
multcompare(stats,0.05,'on',' ','s')
```

```
ans =  
 1.0000    2.0000   -0.0012    0.0008    0.0029  
 1.0000    3.0000    0.0013    0.0051    0.0088  
 2.0000    3.0000    0.0005    0.0042    0.0079
```

This matrix shows that the estimated difference between the intercepts of groups 1 and 2 (1970 and 1976) is 0.0008, and a confidence interval for the difference is [-0.0012, 0.0029]. There is no significant difference between the two. There are significant differences, however, between the intercept for 1982 and each of the other two. The graph shows the same information.



Note that the `stats` structure was created in the initial call to the `aoctool` function, so it is based on the initial model fit (typically a separate-lines model). If you change the model interactively and want to base your multiple comparisons on the new model, you need to run `aoctool` again to get another `stats` structure, this time specifying your new model as the initial model.

## Nonparametric Methods

### In this section...

“Introduction to Nonparametric Methods” on page 9-69

“Kruskal-Wallis Test” on page 9-69

“Friedman's Test” on page 9-70

### Introduction to Nonparametric Methods

Statistics and Machine Learning Toolbox functions include nonparametric versions of one-way and two-way analysis of variance. Unlike classical tests, nonparametric tests make only mild assumptions about the data, and are appropriate when the distribution of the data is non-normal. On the other hand, they are less powerful than classical methods for normally distributed data.

Both of the nonparametric functions described here will return a `stats` structure that can be used as an input to the `multcompare` function for multiple comparisons.

### Kruskal-Wallis Test

The example “Perform One-Way ANOVA” on page 9-6 uses one-way analysis of variance to determine if the bacteria counts of milk varied from shipment to shipment. The one-way analysis rests on the assumption that the measurements are independent, and that each has a normal distribution with a common variance and with a mean that was constant in each column. You can conclude that the column means were not all the same. The following example repeats that analysis using a nonparametric procedure.

The Kruskal-Wallis test is a nonparametric version of one-way analysis of variance. The assumption behind this test is that the measurements come from a continuous distribution, but not necessarily a normal distribution. The test is based on an analysis of variance using the ranks of the data values, not the data values themselves. Output includes a table similar to an ANOVA table, and a box plot.

You can run this test as follows:

```
load hogg  
  
p = kruskalwallis(hogg)
```

```
p =  
  0.0020
```

The low  $p$  value means the Kruskal-Wallis test results agree with the one-way analysis of variance results.

## Friedman's Test

“Perform Two-Way ANOVA” on page 9-18 uses two-way analysis of variance to study the effect of car model and factory on car mileage. The example tests whether either of these factors has a significant effect on mileage, and whether there is an interaction between these factors. The conclusion of the example is there is no interaction, but that each individual factor has a significant effect. The next example examines whether a nonparametric analysis leads to the same conclusion.

Friedman's test is a nonparametric test for data having a two-way layout (data grouped by two categorical factors). Unlike two-way analysis of variance, Friedman's test does not treat the two factors symmetrically and it does not test for an interaction between them. Instead, it is a test for whether the columns are different after adjusting for possible row differences. The test is based on an analysis of variance using the ranks of the data across categories of the row factor. Output includes a table similar to an ANOVA table.

You can run Friedman's test as follows.

```
load mileage  
p = friedman(mileage,3)  
p =  
  7.4659e-004
```

Recall the classical analysis of variance gave a  $p$  value to test column effects, row effects, and interaction effects. This  $p$  value is for column effects. Using either this  $p$  value or the  $p$  value from ANOVA ( $p < 0.0001$ ), you conclude that there are significant column effects.

In order to test for row effects, you need to rearrange the data to swap the roles of the rows in columns. For a data matrix  $x$  with no replications, you could simply transpose the data and type

```
p = friedman(x')
```

With replicated data it is slightly more complicated. A simple way is to transform the matrix into a three-dimensional array with the first dimension representing the replicates, swapping the other two dimensions, and restoring the two-dimensional shape.



```
x = reshape(mileage, [3 2 3]);
x = permute(x, [1 3 2]);
x = reshape(x, [9 2])
x =
 33.3000    32.6000
 33.4000    32.5000
 32.9000    33.0000
 34.5000    33.4000
 34.8000    33.7000
 33.8000    33.9000
 37.4000    36.6000
 36.8000    37.0000
 37.6000    36.7000

friedman(x,3)
ans =
    0.0082
```

Again, the conclusion is similar to that of the classical analysis of variance. Both this  $p$  value and the one from ANOVA ( $p = 0.0039$ ) lead you to conclude that there are significant row effects.

You cannot use Friedman's test to test for interactions between the row and column factors.

## MANOVA

### In this section...

“Introduction to MANOVA” on page 9-72

“ANOVA with Multiple Responses” on page 9-72

### Introduction to MANOVA

The analysis of variance technique in “Perform One-Way ANOVA” on page 9-6 takes a set of grouped data and determine whether the mean of a variable differs significantly among groups. Often there are multiple response variables, and you are interested in determining whether the entire set of means is different from one group to the next. There is a multivariate version of analysis of variance that can address the problem.

### ANOVA with Multiple Responses

The `carsmall` data set has measurements on a variety of car models from the years 1970, 1976, and 1982. Suppose you are interested in whether the characteristics of the cars have changed over time.

Load the sample data.

```
load carsmall
whos
```

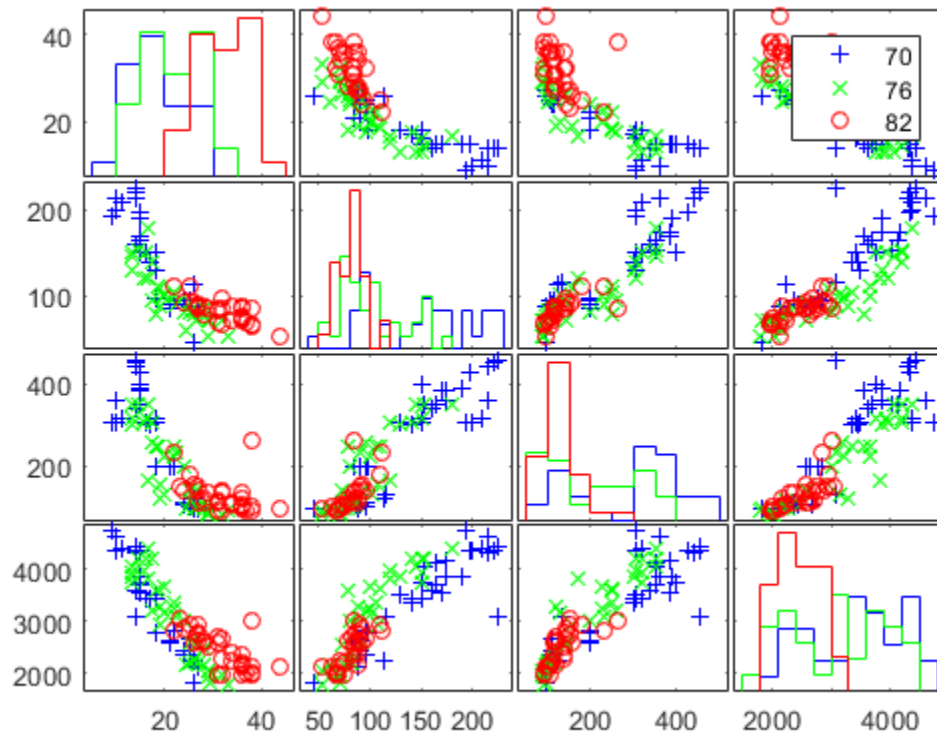
Name	Size	Bytes	Class	Attributes
Acceleration	100x1	800	double	
Cylinders	100x1	800	double	
Displacement	100x1	800	double	
Horsepower	100x1	800	double	
MPG	100x1	800	double	
Mfg	100x13	2600	char	
Model	100x33	6600	char	
Model_Year	100x1	800	double	
Origin	100x7	1400	char	
Weight	100x1	800	double	

Four of these variables (Acceleration, Displacement, Horsepower, and MPG) are continuous measurements on individual car models. The variable `Model_Year` indicates

the year in which the car was made. You can create a grouped plot matrix of these variables using the `gplotmatrix` function.

Create a grouped plot matrix of these variables using the `gplotmatrix` function.

```
x = [MPG Horsepower Displacement Weight];
gplotmatrix(x, [], Model_Year, [], '+xo')
```



(When the second argument of `gplotmatrix` is empty, the function graphs the columns of the `x` argument against each other, and places histograms along the diagonals. The empty fourth argument produces a graph with the default colors. The fifth argument controls the symbols used to distinguish between groups.)

It appears the cars do differ from year to year. The upper right plot, for example, is a graph of MPG versus Weight. The 1982 cars appear to have higher mileage than the older cars, and they appear to weigh less on average. But as a group, are the three years significantly different from one another? The `manova1` function can answer that question.

```
[d,p,stats] = manova1(x,Model_Year)
```

```
d =
```

```
2
```

```
p =
```

```
1.0e-06 *
```

```
0.0000
```

```
0.1141
```

```
stats =
```

```
struct with fields:
```

```
W: [4x4 double]
```

```
B: [4x4 double]
```

```
T: [4x4 double]
```

```
dfW: 90
```

```
dfB: 2
```

```
dfT: 92
```

```
lambda: [2x1 double]
```

```
chisq: [2x1 double]
```

```
chisqdf: [2x1 double]
```

```
eigenval: [4x1 double]
```

```
eigenvec: [4x4 double]
```

```
canon: [100x4 double]
```

```
mdist: [1x100 double]
```

```
gmdist: [3x3 double]
```

```
gnames: {3x1 cell}
```

The `manova1` function produces three outputs:

- The first output, `d`, is an estimate of the dimension of the group means. If the means were all the same, the dimension would be 0, indicating that the means are at the same point. If the means differed but fell along a line, the dimension would be 1. In the example the dimension is 2, indicating that the group means fall in a plane but not along a line. This is the largest possible dimension for the means of three groups.
- The second output, `p`, is a vector of  $p$ -values for a sequence of tests. The first  $p$  value tests whether the dimension is 0, the next whether the dimension is 1, and so on. In this case both  $p$ -values are small. That's why the estimated dimension is 2.
- The third output, `stats`, is a structure containing several fields, described in the following section.

### The Fields of the `stats` Structure

The `W`, `B`, and `T` fields are matrix analogs to the within, between, and total sums of squares in ordinary one-way analysis of variance. The next three fields are the degrees of freedom for these matrices. Fields `lambda`, `chisq`, and `chisqdf` are the ingredients of the test for the dimensionality of the group means. (The  $p$ -values for these tests are the first output argument of `manova1`.)

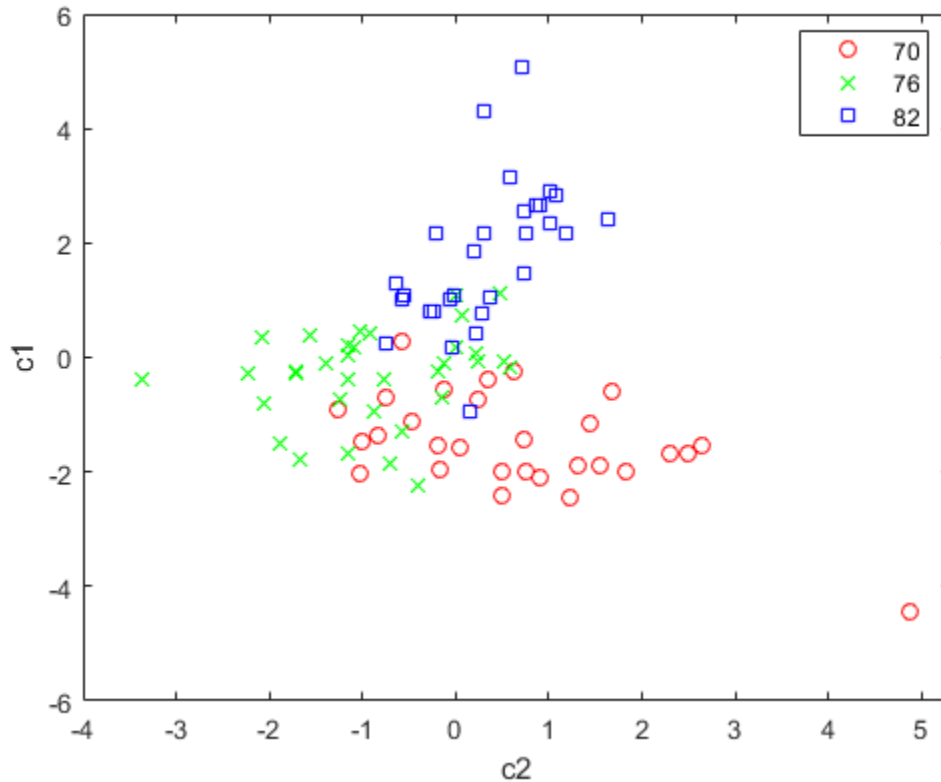
The next three fields are used to do a canonical analysis. Recall that in principal components analysis on page 15-83 (“Principal Component Analysis (PCA)” on page 15-83) you look for the combination of the original variables that has the largest possible variation. In multivariate analysis of variance, you instead look for the linear combination of the original variables that has the largest separation between groups. It is the single variable that would give the most significant result in a univariate one-way analysis of variance. Having found that combination, you next look for the combination with the second highest separation, and so on.

The `eigenvec` field is a matrix that defines the coefficients of the linear combinations of the original variables. The `eigenval` field is a vector measuring the ratio of the between-group variance to the within-group variance for the corresponding linear combination. The `canon` field is a matrix of the canonical variable values. Each column is a linear combination of the mean-centered original variables, using coefficients from the `eigenvec` matrix.

```
c1 = stats.canon(:,1);
c2 = stats.canon(:,2);
```

Plot the grouped scatter plot of the first two canonical variables.

```
figure()  
gscatter(c2,c1,Model_Year,[],'oxs')
```



A grouped scatter plot of the first two canonical variables shows more separation between groups than a grouped scatter plot of any pair of original variables. In this example, it shows three clouds of points, overlapping but with distinct centers. One point in the bottom right sits apart from the others. You can mark this point on the plot using the `gname` function.

Roughly speaking, the first canonical variable,  $c_1$ , separates the 1982 cars (which have high values of  $c_1$ ) from the older cars. The second canonical variable,  $c_2$ , reveals some separation between the 1970 and 1976 cars.

The final two fields of the `stats` structure are Mahalanobis distances. The `mdist` field measures the distance from each point to its group mean. Points with large values may be outliers. In this data set, the largest outlier is the one in the scatter plot, the Buick Estate station wagon. (Note that you could have supplied the model name to the `gname` function above if you wanted to label the point with its model name rather than its row number.)

Find the largest distance from the group mean.

```
max(stats.mdist)
```

```
ans =
```

```
31.5273
```

Find the point that has the largest distance from the group mean.

```
find(stats.mdist == ans)
```

```
ans =
```

```
20
```

Find the car model that corresponds to the largest distance from the group mean.

```
Model(20, :)
```

```
ans =
```

```
'buick estate wagon (sw)'
```

The `gmdist` field measures the distances between each pair of group means. Examine the group means using `grpstats`.

```
grpstats(x, Model_Year)
```

```
ans =
```

```

1.0e+03 *
0.0177  0.1489  0.2869  3.4413
0.0216  0.1011  0.1978  3.0787
0.0317  0.0815  0.1289  2.4535

```

Find the distances between the each pair of group means.

```
stats.gmdist
```

```
ans =
```

```

      0  3.8277  11.1106
3.8277  0  6.1374
11.1106  6.1374  0

```

As might be expected, the multivariate distance between the extreme years 1970 and 1982 (11.1) is larger than the difference between more closely spaced years (3.8 and 6.1). This is consistent with the scatter plots, where the points seem to follow a progression as the year changes from 1970 through 1976 to 1982. If you had more groups, you might find it instructive to use the `manovacluster` function to draw a diagram that presents clusters of the groups, formed using the distances between their means.



## Model Specification for Repeated Measures Models

Model specification for a repeated measures model is a character vector representing a formula in the form

'y1-yk ~ terms',

where the responses and terms are in Wilkinson notation.

For example, if you have five repeated measures  $y_1, y_2, y_3, y_4,$  and  $y_5$ , and you include the terms  $X_1, X_2, X_3, X_4,$  and  $X_3:X_4$  in your linear model, then you can specify `modelspec` as follows:

### Wilkinson Notation

Wilkinson notation describes the factors present in models. It does not describe the multipliers (coefficients) of those factors.

Use these rules to specify the responses in `modelspec`.

Wilkinson Notation	Description
Y1, Y2, Y3	Specific list of variables
Y1-Y5	All table variables from Y1 through Y5

The following rules are for specifying terms in `modelspec`.

Wilkinson Notation	Factors in Standard Notation
1	Constant (intercept) term
$X^k$ , where $k$ is a positive integer	$X, X^2, \dots, X^k$
$X_1 + X_2$	$X_1, X_2$
$X_1 * X_2$	$X_1, X_2, X_1 * X_2$
$X_1 : X_2$	$X_1 * X_2$ only
$-X_2$	Do not include $X_2$
$X_1 * X_2 + X_3$	$X_1, X_2, X_3, X_1 * X_2$
$X_1 + X_2 + X_3 + X_1 : X_2$	$X_1, X_2, X_3, X_1 * X_2$
$X_1 * X_2 * X_3 - X_1 : X_2 : X_3$	$X_1, X_2, X_3, X_1 * X_2, X_1 * X_3, X_2 * X_3$

Wilkinson Notation	Factors in Standard Notation
$X1 * (X2 + X3)$	X1, X2, X3, X1 * X2, X1 * X3

Statistics and Machine Learning Toolbox notation always includes a constant term unless you explicitly remove the term using `-1`.

### See Also

`fitrm`

## Compound Symmetry Assumption and Epsilon Corrections

The regular  $p$ -value calculations in the repeated measures anova (`ranova`) are accurate if the theoretical distribution of the response variables has compound symmetry. This means that all response variables have the same variance, and each pair of response variables share a common correlation. That is,

$$\Sigma = \sigma^2 \begin{pmatrix} 1 & \rho & \cdots & \rho \\ \rho & 1 & \cdots & \rho \\ \vdots & \vdots & \ddots & \vdots \\ \rho & \rho & \cdots & 1 \end{pmatrix}.$$

Under the compound symmetry assumption, the  $F$ -statistics in the repeated measures anova table have an  $F$ -distribution with degrees of freedom  $(v_1, v_2)$ . Here,  $v_1$  is the rank of the contrast being tested, and  $v_2$  is the degrees of freedom for error. If the compound symmetry assumption is not true, the  $F$ -statistic has an approximate  $F$ -distribution with degrees of freedom  $(\varepsilon v_1, \varepsilon v_2)$ , where  $\varepsilon$  is the correction factor. Then, the  $p$ -value must be computed using the adjusted values. The three different correction factor computations are as follows:

- **Greenhouse-Geisser approximation**

$$\varepsilon_{GG} = \frac{\left( \sum_{i=1}^p \lambda_i \right)^2}{d \sum_{i=1}^p \lambda_i^2},$$

where  $\lambda_i$   $i = 1, 2, \dots, p$  are the eigenvalues of the covariance matrix.  $p$  is the number of variables, and  $d$  is equal to  $p-1$ .

- **Huynh-Feldt approximation**

$$\varepsilon_{HF} = \min \left( 1, \frac{nd\varepsilon_{GG} - 2}{d(n - rx) - d^2\varepsilon_{GG}} \right),$$

where  $n$  is the number of rows in the design matrix and  $r$  is the rank of the design matrix.

- **Lower bound on the true  $p$ -value**

$$\varepsilon_{LB} = \frac{1}{d}.$$

## References

- [1] Huynh, H., and L. S. Feldt. “Estimation of the Box Correction for Degrees of Freedom from Sample Data in Randomized Block and Split-Plot Designs.” *Journal of Educational Statistics*. Vol. 1, 1976, pp. 69–82.
- [2] Greenhouse, S. W., and S. Geisser. “An Extension of Box’s Result on the Use of  $F$ -Distribution in Multivariate Analysis.” *Annals of Mathematical Statistics*. Vol. 29, 1958, pp. 885–891.

## See Also

`epsilon` | `mauchly` | `ranova`

## More About

- “Mauchly’s Test of Sphericity” on page 9-83

## Mauchly's Test of Sphericity

The regular  $p$ -value calculations in the repeated measures anova (ranova) are accurate if the theoretical distribution of the response variables have compound symmetry. This means that all response variables have the same variance, and each pair of response variables share a common correlation. That is,

$$\Sigma = \sigma^2 \begin{pmatrix} 1 & \rho & \cdots & \rho \\ \rho & 1 & \cdots & \rho \\ \vdots & \vdots & \ddots & \vdots \\ \rho & \rho & \cdots & 1 \end{pmatrix}.$$

If the compound symmetry assumption is false, then the degrees of freedom for the repeated measures anova test must be adjusted by a factor  $\varepsilon$ , and the  $p$ -value must be computed using the adjusted values.

Compound symmetry implies sphericity.

For a repeated measures model with responses  $y_1, y_2, \dots$ , sphericity means that all pairwise differences  $y_1 - y_2, y_1 - y_3, \dots$  have the same theoretical variance. Mauchly's test is the most accepted test for sphericity.

Mauchly's  $W$  statistic is

$$W = \frac{|T|}{(\text{trace}(T)/p)^d},$$

where

$$T = M' \Sigma M.$$

$M$  is a  $p$ -by- $d$  orthogonal contrast matrix,  $\Sigma$  is the covariance matrix,  $p$  is the number of variables, and  $d = p - 1$ .

A chi-square test statistic assesses the significance of  $W$ . If  $n$  is the number of rows in the design matrix, and  $r$  is the rank of the design matrix, then the chi-square statistic is

$$C = -(n - r) \log(W) D,$$

where

$$D = 1 - \frac{2d^2 + d + 2}{6d(n - r)}.$$

The  $C$  test statistic has a chi-square distribution with  $(p(p - 1)/2) - 1$  degrees of freedom. A small  $p$ -value for the Mauchly's test indicates that the sphericity assumption does not hold.

The `rmanova` method computes the  $p$ -values for the repeated measures anova based on the results of the Mauchly's test and each epsilon value.

## References

- [1] Mauchly, J. W. "Significance Test for Sphericity of a Normal  $n$ -Variate Distribution. *The Annals of Mathematical Statistics*. Vol. 11, 1940, pp. 204–209.

## See Also

`epsilon` | `mauchly` | `ranova`

## More About

- "Compound Symmetry Assumption and Epsilon Corrections" on page 9-81

## Multivariate Analysis of Variance for Repeated Measures

Multivariate analysis of variance analysis is a test of the form  $A^*B^*C = D$ , where  $B$  is the  $p$ -by- $r$  matrix of coefficients.  $p$  is the number of terms, such as the constant, linear predictors, dummy variables for categorical predictors, and products and powers,  $r$  is the number of repeated measures, and  $n$  is the number of subjects.  $A$  is an  $a$ -by- $p$  matrix, with rank  $a \leq p$ , defining hypotheses based on the between-subjects model.  $C$  is an  $r$ -by- $c$  matrix, with rank  $c \leq r \leq n - p$ , defining hypotheses based on the within-subjects model, and  $D$  is an  $a$ -by- $c$  matrix, containing the hypothesized value.

*manova* tests if the model terms are significant in their effect on the response by measuring how they contribute to the overall covariance. It includes all terms in the between-subjects model. *manova* always takes  $D$  as zero. The multivariate response for each observation (subject) is the vector of repeated measures.

*manova* uses four different methods to measure these contributions: Wilks' lambda, Pillai's trace, Hotelling-Lawley trace, Roy's maximum root statistic. Define

$$T = A\hat{B}C - D,$$

$$Z = A(XX)^{-1}A'.$$

Then, the hypotheses sum of squares and products matrix is

$$Q_h = TZ^{-1}T,$$

and the residuals sum of squares and products matrix is

$$Q_e = C'(R'R)C,$$

where

$$R = Y - X\hat{B}.$$

The matrix  $Q_h$  is analogous to the numerator of a univariate  $F$ -test, and  $Q_e$  is analogous to the error sum of squares. Hence, the four statistics *manova* uses are:

- **Wilks' lambda**

$$\Lambda = \frac{|Q_e|}{|Q_h + Q_e|} = \prod \frac{1}{1 + \lambda_i},$$

where  $\lambda_i$  are the solutions of the characteristic equation  $|Q_h - \lambda Q_e| = 0$ .

- **Pillai's trace**

$$V = \text{trace}\left(Q_h (Q_h + Q_e)^{-1}\right) = \sum \theta_i,$$

where  $\theta_i$  values are the solutions of the characteristic equation  $Q_h - \theta(Q_h + Q_e) = 0$ .

- **Hotelling-Lawley trace**

$$U = \text{trace}\left(Q_h Q_e^{-1}\right) = \sum \lambda_i.$$

- **Roy's maximum root statistic**

$$\Theta = \max\left(\text{eig}\left(Q_h Q_e^{-1}\right)\right).$$

## References

- [1] Charles, S. D. *Statistical Methods for the Analysis of Repeated Measurements*. Springer Texts in Statistics. Springer-Verlag, New York, Inc., 2002.

## See Also

`coefstest` | `manova`



# Bayesian Optimization

---

- “Bayesian Optimization Algorithm” on page 10-2
- “Parallel Bayesian Optimization” on page 10-9
- “Bayesian Optimization Plot Functions” on page 10-14
- “Bayesian Optimization Output Functions” on page 10-23
- “Bayesian Optimization Workflow” on page 10-30
- “Variables for a Bayesian Optimization” on page 10-39
- “Bayesian Optimization Objective Functions” on page 10-42
- “Constraints in Bayesian Optimization” on page 10-44
- “Optimize a Cross-Validated SVM Classifier Using Bayesian Optimization” on page 10-53
- “Optimize an SVM Classifier Fit Using Bayesian Optimization” on page 10-66
- “Optimize a Boosted Regression Ensemble” on page 10-75

## Bayesian Optimization Algorithm

<b>In this section...</b>
“Algorithm Outline” on page 10-2
“Gaussian Process Regression for Fitting the Model” on page 10-3
“Acquisition Function Types” on page 10-4
“Acquisition Function Maximization” on page 10-7

### Algorithm Outline

The Bayesian optimization algorithm attempts to minimize a scalar objective function  $f(x)$  for  $x$  in a bounded domain. The function can be deterministic or stochastic, meaning it can return different results when evaluated at the same point  $x$ . The components of  $x$  can be continuous reals, integers, or categorical, meaning a discrete set of names.

---

**Note** Throughout this discussion,  $D$  represents the number of components of  $x$ .

---

The key elements in the minimization are:

- A Gaussian process model of  $f(x)$ .
- A Bayesian update procedure for modifying the Gaussian process model at each new evaluation of  $f(x)$ .
- An acquisition function  $a(x)$  (based on the Gaussian process model of  $f$ ) that you maximize to determine the next point  $x$  for evaluation. For details, see “Acquisition Function Types” on page 10-4 and “Acquisition Function Maximization” on page 10-7.

Algorithm outline:

- Evaluate  $y_i = f(x_i)$  for `NumSeedPoints` points  $x_i$ , taken at random within the variable bounds. `NumSeedPoints` is a `bayesopt` setting. If there are evaluation errors, take more random points until there are `NumSeedPoints` successful evaluations. The probability distribution of each component is either uniform or log-scaled, depending on the `Transform` value in `optimizableVariable`.

Then repeat the following steps:

- 1 Update the Gaussian process model of  $f(x)$  to obtain a posterior distribution over functions  $Q(f|x_i, y_i \text{ for } i = 1, \dots, t)$ . (Internally, `bayesopt` uses `fitrgp` to fit a Gaussian process model to the data.)
- 2 Find the new point  $x$  that maximizes the acquisition function  $a(x)$ .

The algorithm stops after reaching any of the following:

- A fixed number of iterations (default 30).
- A fixed time (default is no time limit).
- A stopping criterion that you supply in “Bayesian Optimization Output Functions” on page 10-23 or “Bayesian Optimization Plot Functions” on page 10-14.

For the algorithmic differences in parallel, see “Parallel Bayesian Algorithm” on page 10-9.

## Gaussian Process Regression for Fitting the Model

The underlying probabilistic model for the objective function  $f$  is a Gaussian process prior with added Gaussian noise in the observations. So the prior distribution on  $f(x)$  is a Gaussian process with mean  $\mu(x; \theta)$  and covariance kernel function  $k(x, x'; \theta)$ . Here,  $\theta$  is a vector of kernel parameters. For the particular kernel function `bayesopt` uses, see “Kernel Function” on page 10-4.

In a bit more detail, denote a set of points  $X = x_i$  with associated objective function values  $F = f_i$ . The prior’s joint distribution of the function values  $F$  is multivariate normal, with mean  $\mu(X)$  and covariance matrix  $K(X, X)$ , where  $K_{ij} = k(x_i, x_j)$ .

Without loss of generality, the prior mean is given as 0.

Also, the observations are assumed to have added Gaussian noise with variance  $\sigma^2$ . So the prior distribution has covariance  $K(X, X; \theta) + \sigma^2 I$ .

Fitting a Gaussian process regression model to observations consists of finding values for the noise variance  $\sigma^2$  and kernel parameters  $\theta$ . This fitting is a computationally intensive process performed by `fitrgp`.

For details on fitting a Gaussian process to observations, see “Gaussian Process Regression”.

## Kernel Function

The kernel function  $k(x, x'; \theta)$  can significantly affect the quality of a Gaussian process regression. `bayesopt` uses the ARD Matérn 5/2 kernel defined in “Kernel (Covariance) Function Options” on page 6-6.

See Snoek, Larochelle, and Adams [3].

## Acquisition Function Types

Six choices of acquisition functions are available for `bayesopt`. There are three basic types, with `expected-improvement` also modified by `per-second` or `plus`:

- `'expected-improvement-per-second-plus'` (default)
- `'expected-improvement'`
- `'expected-improvement-plus'`
- `'expected-improvement-per-second'`
- `'lower-confidence-bound'`
- `'probability-of-improvement'`

The acquisition functions evaluate the “goodness” of a point  $x$  based on the posterior distribution function  $Q$ . When there are coupled constraints, including the Error constraint (see “Objective Function Errors” on page 10-43), all acquisition functions modify their estimate of “goodness” following a suggestion of Gelbart, Snoek, and Adams [2]. Multiply the “goodness” by an estimate of the probability that the constraints are satisfied, to arrive at the acquisition function.

- “Expected Improvement” on page 10-4
- “Probability of Improvement” on page 10-5
- “Lower Confidence Bound” on page 10-5
- “Per Second” on page 10-6
- “Plus” on page 10-6

## Expected Improvement

The `'expected-improvement'` family of acquisition functions evaluates the expected amount of improvement in the objective function, ignoring values that cause an increase in the objective. In other words, define

- $x_{\text{best}}$  as the location of the lowest posterior mean.
- $\mu_Q(x_{\text{best}})$  as the lowest value of the posterior mean.

Then the expected improvement

$$EI(x, Q) = E_Q \left[ \max(0, \mu_Q(x_{\text{best}}) - f(x)) \right].$$

### Probability of Improvement

The 'probability-of-improvement' acquisition function makes a similar, but simpler, calculation as 'expected-improvement'. In both cases, `bayesopt` first calculates  $x_{\text{best}}$  and  $\mu_Q(x_{\text{best}})$ . Then for 'probability-of-improvement', `bayesopt` calculates the probability  $PI$  that a new point  $x$  leads to a better objective function value, modified by a “margin” parameter  $m$ :

$$PI(x, Q) = P_Q(f(x) < \mu_Q(x_{\text{best}}) - m).$$

`bayesopt` takes  $m$  as the estimated noise standard deviation. `bayesopt` evaluates this probability as

$$PI = \Phi(v_Q(x)),$$

where

$$v_Q(x) = \frac{\mu_Q(x_{\text{best}}) - m - \mu_Q(x)}{\sigma_Q(x)}.$$

Here  $\Phi(\cdot)$  is the unit normal CDF, and  $\sigma_Q$  is the posterior standard deviation of the Gaussian process at  $x$ .

### Lower Confidence Bound

The 'lower-confidence-bound' acquisition function looks at the curve  $G$  two standard deviations below the posterior mean at each point:

$$G(x) = \mu_Q(x) - 2\sigma_Q(x).$$

$G(x)$  is the  $2\sigma_Q$  lower confidence envelope of the objective function model. `bayesopt` then maximizes the negative of  $G$ :

$$LCB = 2\sigma_Q(x) - \mu_Q(x).$$

### Per Second

Sometimes, the time to evaluate the objective function can depend on the region. For example, many Support Vector Machine calculations vary in timing a good deal over certain ranges of points. If so, `bayesopt` can obtain better improvement per second by using time-weighting in its acquisition function. The cost-weighted acquisition functions have the phrase `per-second` in their names.

These acquisition functions work as follows. During the objective function evaluations, `bayesopt` maintains another Bayesian model of objective function evaluation time as a function of position  $x$ . The expected improvement per second that the acquisition function uses is

$$EIP_S(x) = \frac{EI_Q(x)}{\mu_S(x)},$$

where  $\mu_S(x)$  is the posterior mean of the timing Gaussian process model.

### Plus

To escape a local objective function minimum, the acquisition functions with `plus` in their names modify their behavior when they estimate that they are overexploiting an area. To understand overexploiting, let  $\sigma_F(x)$  be the standard deviation of the posterior objective function at  $x$ . Let  $\sigma$  be the posterior standard deviation of the additive noise, so that

$$\sigma_Q^2(x) = \sigma_F^2(x) + \sigma^2.$$

Define  $t_\sigma$  to be the value of the `ExplorationRatio` option, a positive number. The `bayesopt plus` acquisition functions, after each iteration, evaluate whether the next point  $x$  satisfies

$$\sigma_F(x) < t_\sigma \sigma.$$

If so, the algorithm declares that  $x$  is overexploiting. Then the acquisition function modifies its “Kernel Function” on page 10-4 by multiplying  $\theta$  by the number of iterations, as suggested by Bull [1]. This modification raises the variance  $\sigma_Q$  for points in between observations. It then generates a new point based on the new fitted kernel function. If the new point  $x$  is again overexploiting, the acquisition function multiplies  $\theta$  by an additional factor of 10 and tries again. It continues in this way up to five times, trying to generate a point  $x$  that is not overexploiting. The algorithm accepts the new  $x$  as the next point.

ExplorationRatio therefore controls a tradeoff between exploring new points for a better global solution, versus concentrating near points that have already been examined.

## Acquisition Function Maximization

Internally, `bayesopt` maximizes an acquisition function using the following general steps:

- 1 For algorithms starting with 'expected-improvement' and for 'probability-of-improvement', `bayesopt` estimates the smallest feasible mean of the posterior distribution  $\mu_Q(x_{\text{best}})$  by sampling several thousand points within the variable bounds, taking several of the best (low mean value) feasible points, and improving them using local search, to find the ostensible best feasible point. Feasible means that the point satisfies constraints (see “Constraints in Bayesian Optimization” on page 10-44).
- 2 For all algorithms, `bayesopt` samples several thousand points within the variable bounds, takes several of the best (high acquisition function) feasible points, and improves them using local search, to find the ostensible best feasible point. The acquisition function value depends on the modeled posterior distribution, not a sample of the objective function, and so it can be calculated quickly.

## References

- [1] Bull, A. D. *Convergence rates of efficient global optimization algorithms*. <http://arxiv.org/abs/1101.3501v3>, 2011.
- [2] Gelbart, M., J. Snoek, R. P. Adams. *Bayesian Optimization with Unknown Constraints*. <http://arxiv.org/abs/1403.5607>, 2014.
- [3] Snoek, J., H. Larochelle, R. P. Adams. *Practical Bayesian Optimization of Machine Learning Algorithms*. <http://arxiv.org/abs/1206.2944>, 2012.

## See Also

### Related Examples

- “Bayesian Optimization Workflow” on page 10-30



## Parallel Bayesian Optimization

### In this section...

“Optimize in Parallel” on page 10-9

“Parallel Bayesian Algorithm” on page 10-9

“Settings for Best Parallel Performance” on page 10-10

“Differences in Parallel Bayesian Optimization Output” on page 10-12

### Optimize in Parallel

Running Bayesian optimization in parallel can save time. Running in parallel requires Parallel Computing Toolbox. `bayesopt` performs parallel objective function evaluations concurrently on parallel workers.

To optimize in parallel:

- `bayesopt` — Set the `UseParallel` name-value pair to `true`. For example,
 

```
results = bayesopt(fun,vars,'UseParallel',true);
```
- Fit functions — Set the `UseParallel` field of the `HyperparameterOptimizationOptions` structure to `true`. For example,
 

```
Mdl = fitcsvm(X,Y,'OptimizeHyperparameters','auto',...
              'HyperparameterOptimizationOptions',struct('UseParallel',true))
```

### Parallel Bayesian Algorithm

The parallel Bayesian optimization algorithm is similar to the serial algorithm, which is described in “Bayesian Optimization Algorithm” on page 10-2. The differences are:

- `bayesopt` assigns points to evaluate to the parallel workers, generally one point at a time. `bayesopt` calculates on the client to determine which point to assign.
- After `bayesopt` evaluates the initial random points, it chooses points to evaluate by fitting a Gaussian process (GP) model. To fit a GP model while some workers are still evaluating points, `bayesopt` imputes a value to each point that is still on a worker. The imputed value is the mean of the GP model value at the points it is evaluating, or some other value as specified by the `bayesopt` `'ParallelMethod'` name-value

pair. For parallel optimization of fit functions, `bayesopt` uses the default `ParallelMethod` imputed value.

- After `bayesopt` assigns a point to evaluate, and before it computes a new point to assign, it checks whether too many workers are idle. The threshold for active workers is determined by the `MinWorkerUtilization` name-value pair. If too many workers are idle, then `bayesopt` assigns random points, chosen uniformly within bounds, to all idle workers. This step causes the workers to be active more quickly, but the workers have random points rather than fitted points. If the number of idle workers does not exceed the threshold, then `bayesopt` chooses a point to evaluate as usual, by fitting a GP model and maximizing the acquisition function.

## Settings for Best Parallel Performance

Fit functions have no special settings for better parallel performance. In contrast, several `bayesopt` settings can help to speed an optimization.

### Solver Options

Setting the `GPActiveSetSize` option to a smaller value than the default (300) can speed the solution. The cost is potential inaccuracy in the points that `bayesopt` chooses to evaluate, because the GP model of the objective function can be less accurate than with a larger value. Setting the option to a larger value can result in a more accurate GP model, but requires more time to create the model.

Setting the `ParallelMethod` option to `'max-observed'` can lead `bayesopt` to search more widely for a global optimum. This choice can lead to a better solution in less time. However, the default value of `'clipped-model-prediction'` is often best.

Setting the `MinWorkerUtilization` option to a large value can result in higher parallel utilization. However, this setting causes more completely random points to be evaluated, which can lead to less accurate solutions. A large value, in this context, depends on how many workers you have. The default is `floor(0.8*N)`, where `N` is the number of parallel workers. Setting the option to a lower value can give lower parallel utilization, but with the benefit of higher quality points.

### Placing the Objective Function on Workers

You can place an objective function on the parallel workers in one of three ways. Some have better performance, but require a more complex setup.

**1. Automatic** If you give a function handle as the objective function, `bayesopt` sends the handle to all the parallel workers at the beginning of its run. For example,

```
load ionosphere
splits = optimizableVariable('splits',[1,100],'Type','integer');
minleaf = optimizableVariable('minleaf',[1,100],'Type','integer');
fun = @(params)kfoldLoss(fitctree(X,Y,'Kfold',5,...
    'MaxNumSplits',params.splits,'MinLeaf',params.minleaf));

results = bayesopt(fun,[splits,minleaf],'UseParallel',true);
```

This method is effective if the handle is small, or if you run the optimization only once. However, if you plan to run the optimization several times, you can save time by using one of the other two techniques.

**2. Parallel constant** If you plan to run an optimization several times, save time by transferring the objective function to the workers only once. This technique is especially effective when the function handle incorporates a large amount of data. Transfer the objective once by setting the function handle to a `parallel.pool.Constant` construct, as in this example.

```
load ionosphere
splits = optimizableVariable('splits',[1,100],'Type','integer');
minleaf = optimizableVariable('minleaf',[1,100],'Type','integer');
fun = @(params)kfoldLoss(fitctree(X,Y,'Kfold',5,...
    'MaxNumSplits',params.splits,'MinLeaf',params.minleaf));

C = copyFunctionHandleToWorkers(fun);

results1 = bayesopt(C,[splits,minleaf],'UseParallel',true);
results2 = bayesopt(C,[splits,minleaf],'UseParallel',true,...
    'MaxObjectiveEvaluations',50);
results3 = bayesopt(C,[splits,minleaf],'UseParallel',true,...
    'AcquisitionFunction','expected-improvement');
```

In this example, `copyFunctionHandleToWorkers` sends the function handle to the workers only once.

**3. Create objective function on workers** If you have a great deal of data to send to the workers, you can avoid loading the data in the client by using `spmd` to load the data on the workers. Use a Composite with `parallel.pool.Constant` to access the distributed objective functions.

```
% makeFun is at the end of this script
spmd
    fun = makeFun();
end

% ObjectiveFunction is now a Composite. Get a parallel.pool.Constant that
% refers to it, without copying it to the client:
C = parallel.pool.Constant(fun);

% You could also use the line
% C = parallel.pool.Constant(@MakeFun);
% In this case, you do not use spmd

% Call bayesopt, passing the Constant
splits = optimizableVariable('splits', [1 100]);
minleaf = optimizableVariable('minleaf', [1 100]);
bo = bayesopt(C,[splits minleaf], 'UseParallel', true);

function f = makeFun()
load('ionosphere', 'X', 'Y');
f = @fun;
    function L = fun(Params)
        L = kfoldLoss(fitctree(X,Y, ...
            'KFold', 5, ...
            'MaxNumSplits', Params.splits, ...
            'MinLeaf', Params.minleaf));
    end
end
```

In this example, the function handle exists only on the workers. The handle never appears on the client.

## Differences in Parallel Bayesian Optimization Output

When `bayesopt` runs in parallel, the Bayesian optimization output includes these differences.

- **Iterative Display** — Iterative display includes a column showing the number of active workers. This is the number after `bayesopt` assigns a job to the next worker.
- **Plot Functions**

- Objective Function Model plot (@plotObjectiveModel) shows the pending points (those points executing on parallel workers). The height of the points depends on the `ParallelMethod` name-value pair.
- Elapsed Time plot (@plotElapsedTime) shows the total elapsed time with the label **Real time** and the total objective function evaluation time, summed over all workers, with the label **Objective evaluation time (all workers)**. Objective evaluation time includes the time to start a worker on a job.

## See Also

`parallel.pool.Constant` | `spmd`

## More About

- “Bayesian Optimization Algorithm” on page 10-2
- “Feature Extraction Workflow” on page 15-169

## Bayesian Optimization Plot Functions

In this section...
“Built-In Plot Functions” on page 10-14
“Custom Plot Function Syntax” on page 10-15
“Create a Custom Plot Function” on page 10-16

### Built-In Plot Functions

There are two sets of built-in plot functions.

Model Plots — Apply When $D \leq 2$	Description
@plotAcquisitionFunction	Plot the acquisition function surface.
@plotConstraintModels	Plot each constraint model surface. Also plots a $P(\text{feasible})$ surface. Negative values indicate feasible points.  Also plot the error model, if it exists, which ranges from $-1$ to $1$ . Negative values mean that the model probably does not error, positive values mean that it probably does error. The model is:  Plotted error = $-2 * \text{Probability}(\text{error}) + 1$ .
@plotObjectiveEvaluationTimeModel	Plot the objective function evaluation time model surface.
@plotObjectiveModel	Plot the fun model surface, the estimated location of the minimum, and the location of the next proposed point to evaluate. For one-dimensional problems, plot envelopes one credible interval above and below the mean function, and envelopes one noise standard deviation above and below the mean.
Trace Plots — Apply to All D	Description
@plotObjective	Plot each observed function value versus the number of function evaluations.
@plotObjectiveEvaluationTime	Plot each observed function evaluation run time versus the number of function evaluations.

Trace Plots — Apply to All D	Description
@plotMinObjective	Plot the minimum observed and estimated function values versus the number of function evaluations.
@plotElapsedTime	Plot three curves: the total elapsed time of the optimization, the total function evaluation time, and the total modeling and point selection time, all versus the number of function evaluations.

---

**Note** When there are coupled constraints, iterative display and plot functions can give counterintuitive results such as:

- A *minimum objective* plot can increase.
- The optimization can declare a problem infeasible even when it showed an earlier feasible point.

The reason for this behavior is that the decision about whether a point is feasible can change as the optimization progresses. `bayesopt` determines feasibility with respect to its constraint model, and this model changes as `bayesopt` evaluates points. So a “minimum objective” plot can increase when the minimal point is later deemed infeasible, and the iterative display can show a feasible point that is later deemed infeasible.

---

## Custom Plot Function Syntax

A custom plot function has the same syntax as a custom output function (see “Bayesian Optimization Output Functions” on page 10-23):

```
stop = plotfun(results, state)
```

`bayesopt` passes the `results` and `state` variables to your function. Your function returns `stop`, which you set to `true` to halt the iterations, or to `false` to continue the iterations.

`results` is an object of class `BayesianOptimization` that contains the available information on the computations.

`state` has these possible values:

- 'initial' — bayesopt is about to start iterating. Use this state to set up a plot or to perform other initializations.
- 'iteration' — bayesopt just finished an iteration. Generally, you perform most of the plotting or other calculations in this state.
- 'done' — bayesopt just finished its final iteration. Clean up plots or otherwise prepare for the plot function to shut down.

## Create a Custom Plot Function

This example shows how to create a custom plot function for bayesopt. It further shows how to use information in the UserData property of a BayesianOptimization object.

### Problem Statement

The problem is to find parameters of a Support Vector Machine (SVM) classification to minimize the cross-validated loss. The specific model is the same as in “Optimize a Cross-Validated SVM Classifier Using Bayesian Optimization” on page 10-53. Therefore, the objective function is essentially the same, except it also computes UserData, in this case the number of support vectors in an SVM model fitted to the current parameters.

Create a custom plot function that plots the number of support vectors in the SVM model as the optimization progresses. To give the plot function access to the number of support vectors, create a third output, UserData, to return the number of support vectors.

### Objective Function

Create an objective function that computes the cross-validation loss for a fixed cross-validation partition, and that returns the number of support vectors in the resulting model.

```
function [f,viol,nsupp] = mysvminfn(x,cdata,grp,c)
SVMMModel = fitcsvm(cdata,grp,'KernelFunction','rbf',...
    'KernelScale',x.sigma,'BoxConstraint',x.box);
f = kfoldLoss(crossval(SVMMModel,'CVPartition',c));
viol = [];
nsupp = sum(SVMMModel.IsSupportVector);
end
```



## Custom Plot Function

Create a custom plot function that uses the information computed in `UserData`. Have the function plot both the current number of constraints and the number of constraints for the model with the best objective function found.

```
function stop = svmsuppvec(results,state)
persistent hs nbest besthist nsupptrace
stop = false;
switch state
    case 'initial'
        hs = figure;
        besthist = [];
        nbest = 0;
        nsupptrace = [];
    case 'iteration'
        figure(hs)
        nsupp = results.UserDataTrace{end}; % get nsupp from UserDataTrace property.
        nsupptrace(end+1) = nsupp; % accumulate nsupp values in a vector.
        if (results.ObjectiveTrace(end) == min(results.ObjectiveTrace)) || (length(resu
            nbest = nsupp;
        end
        besthist = [besthist,nbest];
        plot(1:length(nsupptrace),nsupptrace,'b',1:length(besthist),besthist,'r--')
        xlabel 'Iteration number'
        ylabel 'Number of support vectors'
        title 'Number of support vectors at each iteration'
        legend('Current iteration','Best objective','Location','best')
        drawnow
end
```

## Set Up the Model

Generate ten base points for each class.

```
rng default
grnpop = mvnrnd([1,0],eye(2),10);
redpop = mvnrnd([0,1],eye(2),10);
```

Generate 100 data points of each class.

```
redpts = zeros(100,2);grnpts = redpts;
for i = 1:100
```

```
    grnpts(i,:) = mvnrnd(grnpop(randi(10),:),eye(2)*0.02);
    redpts(i,:) = mvnrnd(redpop(randi(10),:),eye(2)*0.02);
end
```

Put the data into one matrix, and make a vector `grp` that labels the class of each point.

```
cdata = [grnpts;redpts];
grp = ones(200,1);
% Green label 1, red label -1
grp(101:200) = -1;
```

Check the basic classification of all the data using the default SVM parameters.

```
SVMModel = fitcsvm(cdata,grp,'KernelFunction','rbf','ClassNames',[-1 1]);
```

Set up a partition to fix the cross validation. Without this step, the cross validation is random, so the objective function is not deterministic.

```
c = cvpartition(200,'KFold',10);
```

Check the cross-validation accuracy of the original fitted model.

```
loss = kfoldLoss(fitcsvm(cdata,grp,'CVPartition',c,...
    'KernelFunction','rbf','BoxConstraint',SVMModel.BoxConstraints(1),...
    'KernelScale',SVMModel.KernelParameters.Scale))
```

```
loss =
    0.1350
```

### Prepare Variables for Optimization

The objective function takes an input `z = [rbf_sigma,boxconstraint]` and returns the cross-validation loss value of `z`. Take the components of `z` as positive, log-transformed variables between  $1e-5$  and  $1e5$ . Choose a wide range because you do not know which values are likely to be good.

```
sigma = optimizableVariable('sigma',[1e-5,1e5],'Transform','log');
box = optimizableVariable('box',[1e-5,1e5],'Transform','log');
```

## Set Plot Function and Call the Optimizer

Search for the best parameters `[sigma, box]` using `bayesopt`. For reproducibility, choose the 'expected-improvement-plus' acquisition function. The default acquisition function depends on run time, so it can give varying results.

Plot the number of support vectors as a function of the iteration number, and plot the number of support vectors for the best parameters found.

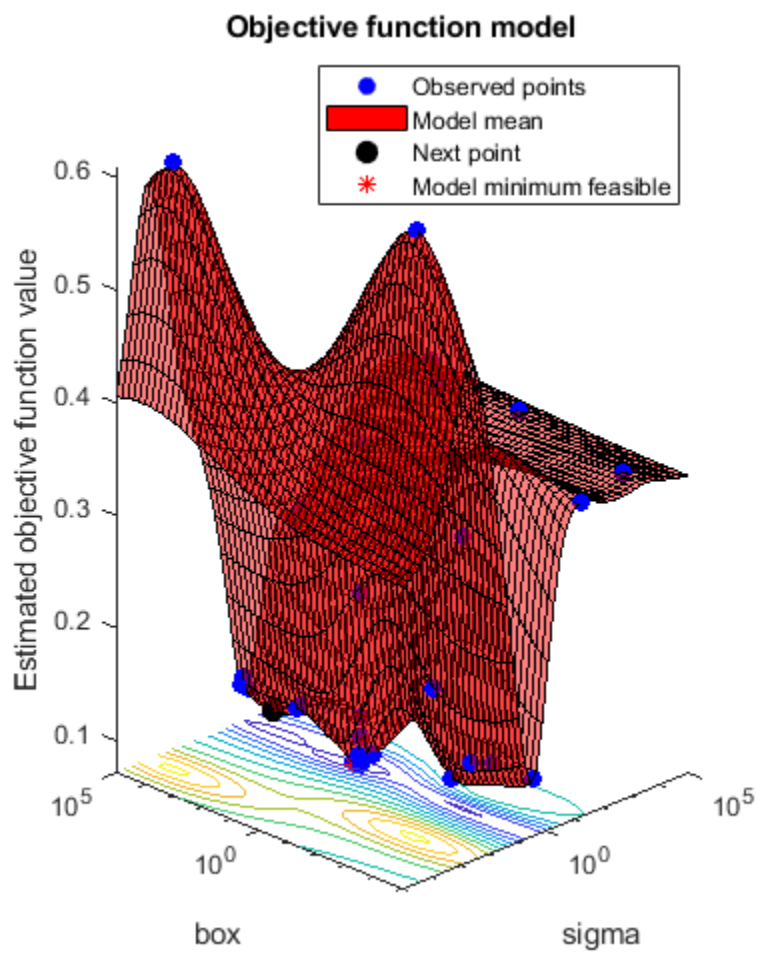
```
obj = @(x)mysvmmminfn(x,cdata,grp,c);
results = bayesopt(obj,[sigma,box],...
    'IsObjectiveDeterministic',true,'Verbose',0,...
    'AcquisitionFunctionName','expected-improvement-plus',...
    'PlotFcn',{@svmsuppvec,@plotObjectiveModel,@plotMinObjective})
```

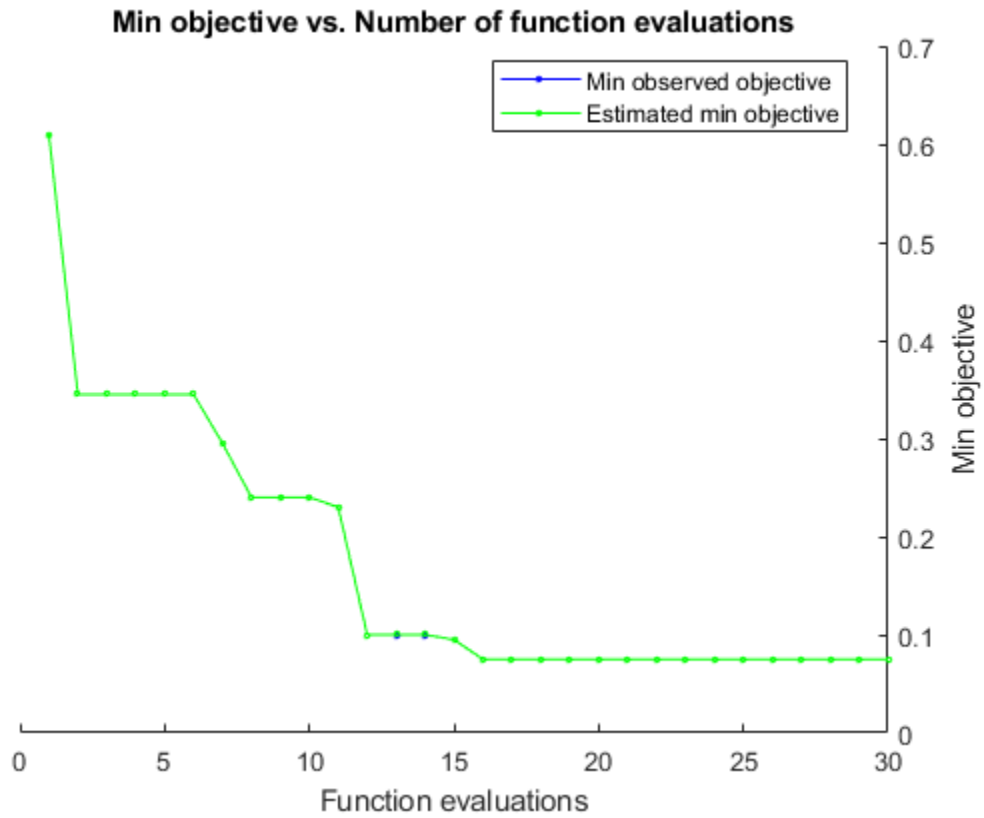
```
results =
```

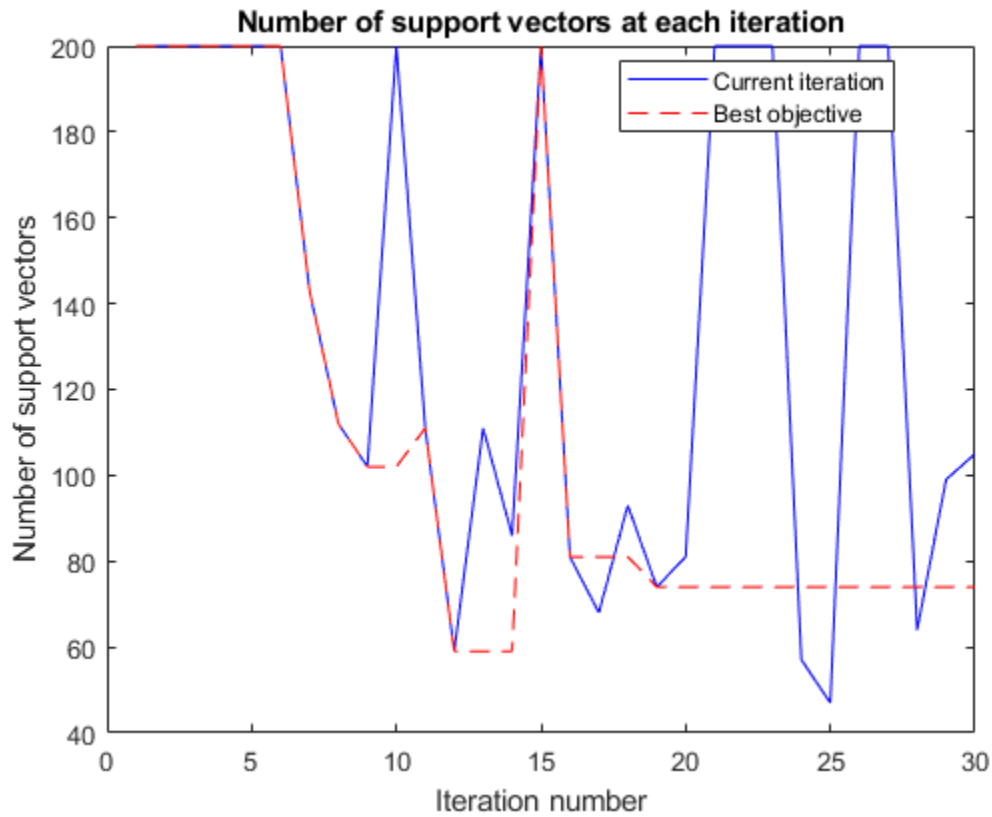
```
BayesianOptimization with properties:
```

```

    ObjectiveFcn: @(x)mysvmmminfn(x,cdata,grp,c)
  VariableDescriptions: [1x2 optimizableVariable]
        Options: [1x1 struct]
    MinObjective: 0.0750
    XAtMinObjective: [1x2 table]
MinEstimatedObjective: 0.0750
XAtMinEstimatedObjective: [1x2 table]
NumObjectiveEvaluations: 30
  TotalElapsedTime: 76.5847
        NextPoint: [1x2 table]
            XTrace: [30x2 table]
    ObjectiveTrace: [30x1 double]
  ConstraintsTrace: []
    UserDataTrace: {30x1 cell}
ObjectiveEvaluationTimeTrace: [30x1 double]
  IterationTimeTrace: [30x1 double]
        ErrorTrace: [30x1 double]
    FeasibilityTrace: [30x1 logical]
FeasibilityProbabilityTrace: [30x1 double]
  IndexOfMinimumTrace: [30x1 double]
    ObjectiveMinimumTrace: [30x1 double]
EstimatedObjectiveMinimumTrace: [30x1 double]
```







## See Also

### Related Examples

- “Bayesian Optimization Output Functions” on page 10-23
- “Constraints in Bayesian Optimization” on page 10-44

## Bayesian Optimization Output Functions

### In this section...

“What Is a Bayesian Optimization Output Function?” on page 10-23

“Built-In Output Functions” on page 10-23

“Custom Output Functions” on page 10-23

“Bayesian Optimization Output Function” on page 10-24

### What Is a Bayesian Optimization Output Function?

An output function is a function that is called at the end of every iteration of `bayesopt`. An output function can halt iterations. It can also create plots, save information to your workspace or to a file, or perform any other calculation you like.

Other than halting the iterations, output functions cannot change the course of a Bayesian optimization. They simply monitor the progress of the optimization.

### Built-In Output Functions

These built-in output functions save your optimization results to a file or to the workspace.

- `@assignInBase` — Saves your results after each iteration to a variable named `'BayesoptResults'` in your workspace. To choose a different name, pass the `SaveVariableName` name-value pair.
- `@saveToFile` — Saves your results after each iteration to a file named `'BayesoptResults.mat'` in your current folder. To choose a different name or folder, pass the `SaveFileName` name-value pair.

For example, to save the results after each iteration to a workspace variable named `'BayesIterations'`,

```
results = bayesopt(fun,vars,'OutputFcn',@assignInBase,'SaveVariableName','BayesIterations')
```

### Custom Output Functions

Write a custom output function with signature

```
stop = outputfun(results, state)
```

`bayesopt` passes the `results` and `state` variables to your function. Your function returns `stop`, which you set to `true` to halt the iterations, or to `false` to allow the iterations to continue.

`results` is an object of class `BayesianOptimization`. `results` contains the available information on the computations so far.

`state` has possible values:

- 'initial' — `bayesopt` is about to start iterating.
- 'iteration' — `bayesopt` just finished an iteration.
- 'done' — `bayesopt` just finished its final iteration.

For an example, see “Bayesian Optimization Output Function” on page 10-24.

## Bayesian Optimization Output Function

This example shows how to use a custom output function with Bayesian optimization. The output function halts the optimization when the objective function, which is the cross-validation error rate, drops below 13%. The output function also plots the time for each iteration.

```
function stop = outputfun(results, state)
persistent h
stop = false;
switch state
    case 'initial'
        h = figure;
    case 'iteration'
        if results.MinObjective < 0.13
            stop = true;
        end
        figure(h)
        tms = results.IterationTimeTrace;
        plot(1:numel(tms), tms')
        xlabel('Iteration Number')
        ylabel('Time for Iteration')
        title('Time for Each Iteration')
        drawnow
```



end

The objective function is the cross validation loss of the KNN classification of the ionosphere data. Load the data and, for reproducibility, set the default random stream.

```
load ionosphere
rng default
```

Optimize over neighborhood size from 1 through 30, and for three distance metrics.

```
num = optimizableVariable('n', [1, 30], 'Type', 'integer');
dst = optimizableVariable('dst', {'chebychev', 'euclidean', 'minkowski'}, 'Type', 'categorical');
vars = [num, dst];
```

Set the cross-validation partition and objective function. For reproducibility, set the AcquisitionFunctionName to 'expected-improvement-plus'. Run the optimization.

```
c = cvpartition(351, 'Kfold', 5);
fun = @(x) kfoldLoss(fitcknn(X, Y, 'CVPartition', c, 'NumNeighbors', x.n, ...
    'Distance', char(x.dst), 'NSMethod', 'exhaustive'));
results = bayesopt(fun, vars, 'OutputFcn', @outputfun, ...
    'AcquisitionFunctionName', 'expected-improvement-plus');
```

```
=====
| Iter | Eval  | Objective | Objective | BestSoFar | BestSoFar |          |
|      | result |           | runtime   | (observed) | (estim.)  |          |
|-----|-----|-----|-----|-----|-----|-----|
|  1  | Best  | 0.19943 | 0.60968 | 0.19943 | 0.19943 | 24      |
|  2  | Best  | 0.16809 | 0.29042 | 0.16809 | 0.1747  | 9       |
|  3  | Best  | 0.12536 | 0.26174 | 0.12536 | 0.12861 | 3       |
|-----|-----|-----|-----|-----|-----|
```

---

Optimization completed.

Total function evaluations: 3

Total elapsed time: 11.4868 seconds.

Total objective function evaluation time: 1.1618

Best observed feasible point:

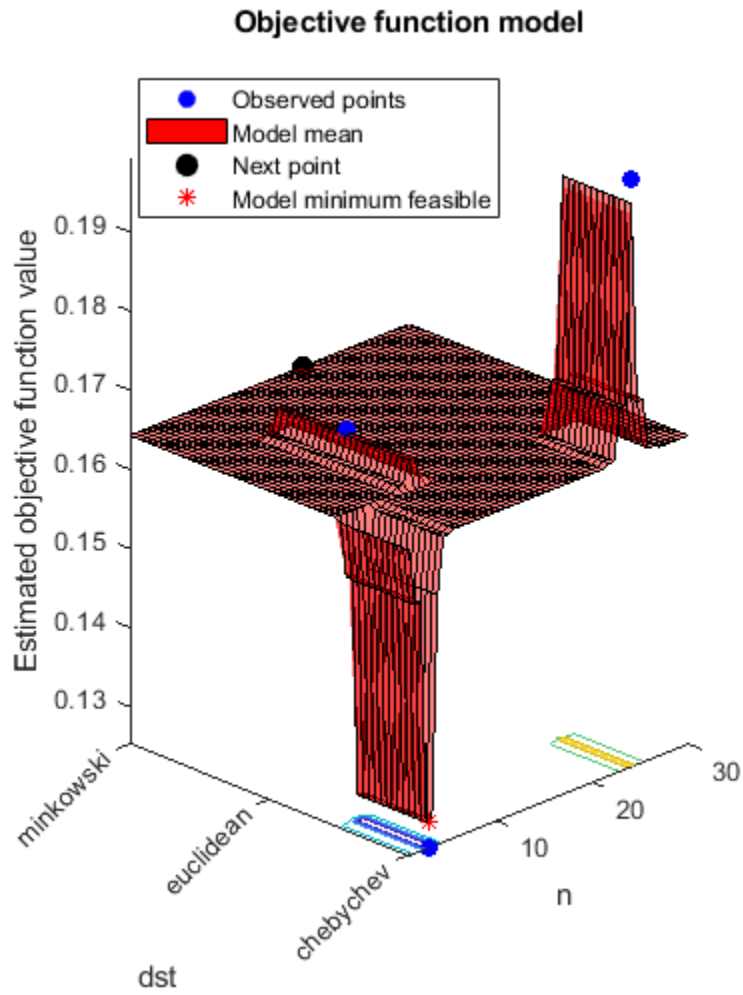
```
  n      dst
  -      -
  3      chebychev
```

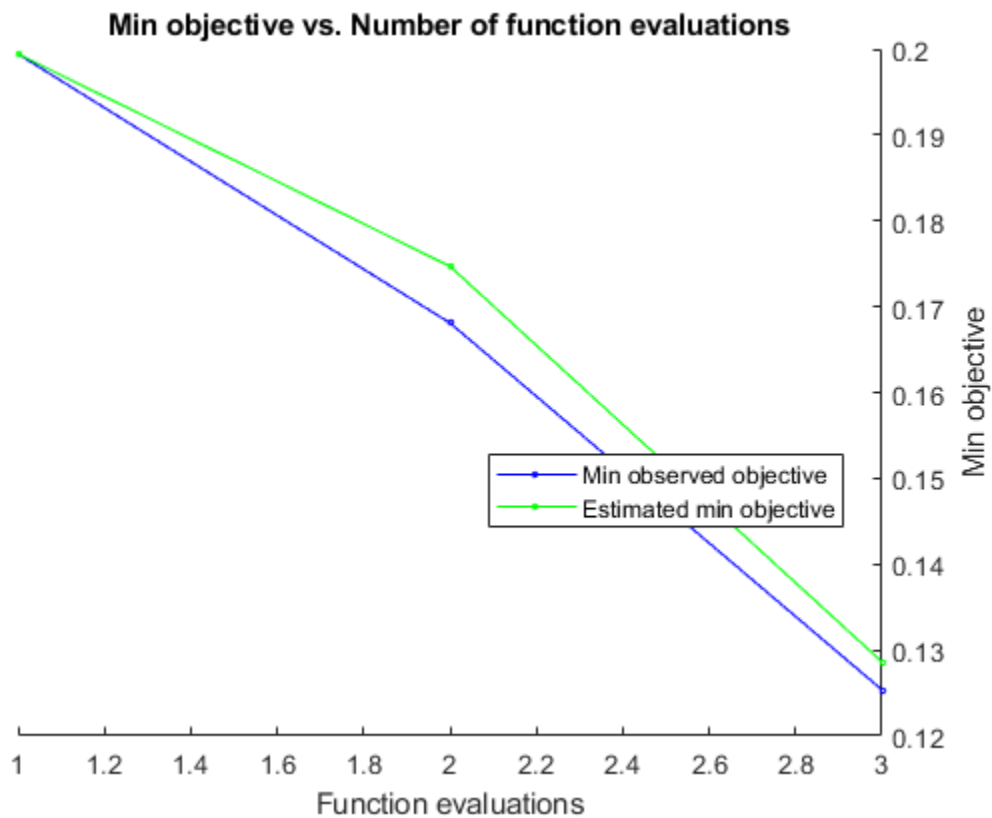
Observed objective function value = 0.12536  
Estimated objective function value = 0.12861  
Function evaluation time = 0.26174

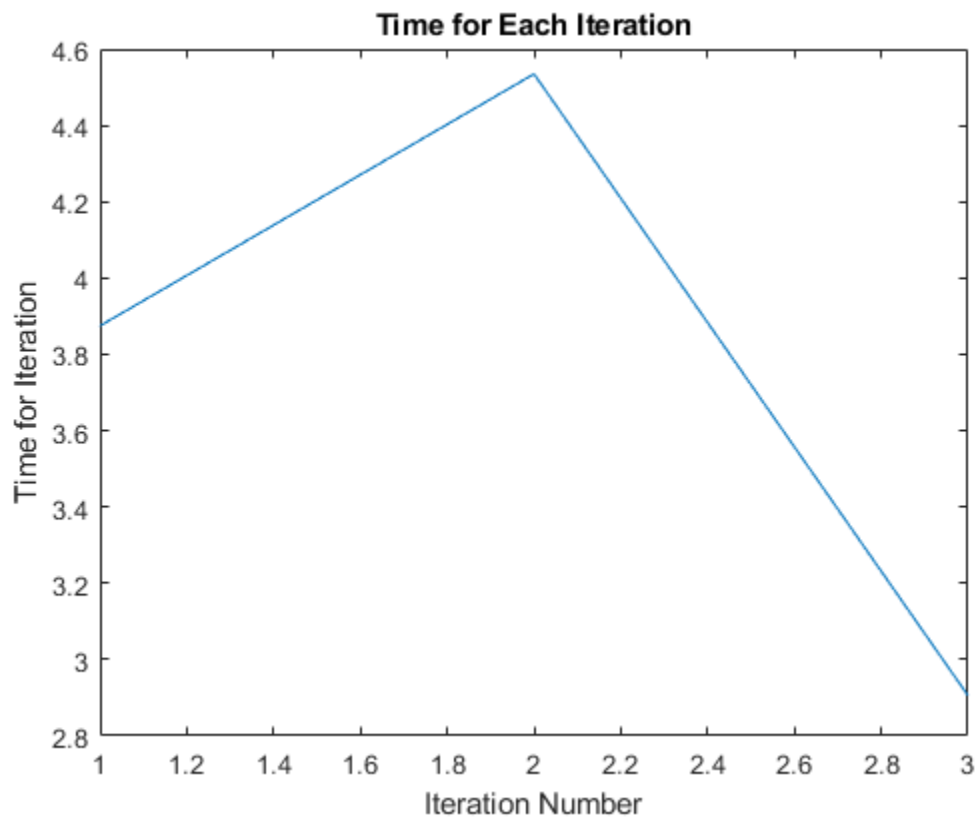
Best estimated feasible point (according to models):

n	dst
3	chebychev

Estimated objective function value = 0.12861  
Estimated function evaluation time = 0.262







## See Also

### Related Examples

- “Bayesian Optimization Plot Functions” on page 10-14

## Bayesian Optimization Workflow

In this section...
“What Is Bayesian Optimization?” on page 10-30
“Two Ways to Perform Bayesian Optimization” on page 10-30
“Bayesian Optimization Using a Fit Function” on page 10-31
“Bayesian Optimization Using bayesopt” on page 10-32
“Bayesian Optimization Characteristics” on page 10-32
“Parameters Available for Fit Functions” on page 10-34
“Hyperparameter Optimization Options for Fit Functions” on page 10-36

### What Is Bayesian Optimization?

Optimization, in its most general form, is the process of locating a point that minimizes a real-valued function called the objective function. Bayesian optimization is the name of one such process. Bayesian optimization internally maintains a Gaussian process model of the objective function, and uses objective function evaluations to train the model. One innovation in Bayesian optimization is the use of an acquisition function, which the algorithm uses to determine the next point to evaluate. The acquisition function can balance sampling at points that have low modeled objective functions, and exploring areas that have not yet been modeled well. For details, see “Bayesian Optimization Algorithm” on page 10-2.

Bayesian optimization is part of Statistics and Machine Learning Toolbox because it is well-suited to optimizing hyperparameters of classification and regression algorithms. A hyperparameter is an internal parameter of a classifier or regression function, such as the box constraint of a support vector machine, or the learning rate of a robust classification ensemble. These parameters can strongly affect the performance of a classifier or regressor, and yet it is typically difficult or time-consuming to optimize them. See “Bayesian Optimization Characteristics” on page 10-32.

Typically, optimizing the hyperparameters means that you try to minimize the cross-validation loss of a classifier or regression.

### Two Ways to Perform Bayesian Optimization

You can perform a Bayesian optimization in two distinct ways:

- **Fit function** — Include the `OptimizeHyperparameters` name-value pair in many fitting functions to have Bayesian optimization apply automatically. The optimization minimizes cross-validation loss. This way gives you fewer tuning options, but enables you to perform Bayesian optimization most easily. See “Bayesian Optimization Using a Fit Function” on page 10-31.
- `bayesopt` — Exert the most control over your optimization by calling `bayesopt` directly. This way requires you to write an objective function, which does not have to represent cross-validation loss. See “Bayesian Optimization Using `bayesopt`” on page 10-32.

## Bayesian Optimization Using a Fit Function

To minimize the error in a cross-validated response via Bayesian optimization, follow these steps.

- 1** Choose your classification or regression solver among `fitcdiscr`, `fitcecoc`, `fitcensemble`, `fitcknn`, `fitclinear`, `fitcnb`, `fitcsvm`, `fitctree`, `fitrensemble`, `fitrgp`, `fitrlinear`, `fitrsvm`, or `fitrtree`.
- 2** Decide on the hyperparameters to optimize, and pass them in the `OptimizeHyperparameters` name-value pair. For each fit function, you can choose from a set of hyperparameters. See [Eligible Hyperparameters for Fit Functions](#), or use the `hyperparameters` function, or consult the fit function reference page.

You can pass a cell array of parameter names. You can also set `'auto'` as the `OptimizeHyperparameters` value, which chooses a typical set of hyperparameters to optimize, or `'all'` to optimize all available parameters.

- 3** For ensemble fit functions `fitcecoc`, `fitcensemble`, and `fitrensemble`, also include parameters of the weak learners in the `OptimizeHyperparameters` cell array.
- 4** Optionally, create an options structure for the `HyperparameterOptimizationOptions` name-value pair. See “[Hyperparameter Optimization Options for Fit Functions](#)” on page 10-36.
- 5** Call the fit function with the appropriate name-value pairs.

For examples, see “[Optimize an SVM Classifier Fit Using Bayesian Optimization](#)” on page 10-66 and “[Optimize a Boosted Regression Ensemble](#)” on page 10-75. Also, every fit function reference page contains a Bayesian optimization example.

## Bayesian Optimization Using `bayesopt`

To perform a Bayesian optimization using `bayesopt`, follow these steps.

- 1 Prepare your variables. See “Variables for a Bayesian Optimization” on page 10-39.
- 2 Create your objective function. See “Bayesian Optimization Objective Functions” on page 10-42. If necessary, create constraints, too. See “Constraints in Bayesian Optimization” on page 10-44.
- 3 Decide on options, meaning the `bayesopt` Name, Value on page 32-160 pairs. You are not required to pass any options to `bayesopt` but you typically do, especially when trying to improve a solution.
- 4 Call `bayesopt`.
- 5 Examine the solution. You can decide to resume the optimization by using `resume`, or restart the optimization, usually with modified options.

For an example, see “Optimize a Cross-Validated SVM Classifier Using Bayesian Optimization” on page 10-53.

## Bayesian Optimization Characteristics

Bayesian optimization algorithms are best suited to these problem types.

Characteristic	Details
Low dimension	Bayesian optimization works best in a low number of dimensions, typically 10 or fewer. While Bayesian optimization can solve some problems with a few dozen variables, it is not recommended for dimensions higher than about 50.
Expensive objective	Bayesian optimization is designed for objective functions that are slow to evaluate. It has considerable overhead, typically several seconds for each iteration.
Low accuracy	Bayesian optimization does not necessarily give very accurate results. If you have a deterministic objective function, you can sometimes improve the accuracy by starting a standard optimization algorithm from the <code>bayesopt</code> solution.



Characteristic	Details
Global solution	Bayesian optimization is a global technique. Unlike many other algorithms, to search for a global solution you do not have to start the algorithm from various initial points.
Hyperparameters	Bayesian optimization is well-suited to optimizing hyperparameters of another function. A hyperparameter is a parameter that controls the behavior of a function. For example, the <code>fitcsvm</code> function fits an SVM model to data. It has hyperparameters <code>BoxConstraint</code> and <code>KernelScale</code> for its <code>'rbf'</code> <code>KernelFunction</code> . For an example of Bayesian optimization applied to hyperparameters, see “Optimize a Cross-Validated SVM Classifier Using Bayesian Optimization” on page 10-53.

## Parameters Available for Fit Functions

### Eligible Hyperparameters for Fit Functions

Function Name	Eligible Parameters
fitcdiscr	Delta Gamma DiscrimType
fitcecoc	Coding eligible fitcdiscr parameters for 'Learners', 'Discriminant' eligible fitcknn parameters for 'Learners', 'knn' eligible fitcllinear parameters for 'Learners', 'linear' eligible fitcsvm parameters for 'Learners', 'svm' eligible fitctree parameters for 'Learners', 'tree'
fitcensemble	Method NumLearningCycles LearnRate eligible fitcdiscr parameters for 'Learners', 'Discriminant' eligible fitcknn parameters for 'Learners', 'knn' eligible fitctree parameters for 'Learners', 'tree'
fitcknn	NumNeighbors Distance DistanceWeight Exponent Standardize
fitcllinear	Lambda Learner Regularization
fitcnb	DistributionNames Width Kernel

Function Name	Eligible Parameters
fitcsvm	BoxConstraint KernelScale KernelFunction PolynomialOrder Standardize
fitctree	MinLeafSize MaxNumSplits SplitCriterion NumVariablesToSample
fitrensemble	Method NumLearningCycles LearnRate <b>eligible fitrtree parameters for</b> 'Learners', 'tree': MinLeafSize MaxNumSplits NumVariablesToSample
fitrgp	Sigma BasisFunction KernelFunction KernelScale Standardize
fitrlinear	Lambda Learner Regularization
fitrsvm	BoxConstraint KernelScale Epsilon KernelFunction PolynomialOrder Standardize
fitrtree	MinLeafSize MaxNumSplits NumVariablesToSample

## Hyperparameter Optimization Options for Fit Functions

When optimizing using a fit function, you have these options available in the `HyperparameterOptimizationOptions` name-value pair. Give the value as a structure. All fields in the structure are optional.

Field Name	Values	Default
Optimizer	<ul style="list-style-type: none"> <li>'bayesopt' — Use Bayesian optimization. Internally, this setting calls <code>bayesopt</code>.</li> <li>'gridsearch' — Use grid search with <code>NumGridDivisions</code> values per dimension.</li> <li>'randomsearch' — Search at random among <code>MaxObjectiveEvaluations</code> points.</li> </ul> <p>'gridsearch' searches in a random order, using uniform sampling without replacement from the grid. After optimization, you can get a table in grid order by using the command <code>sortrows(Mdl.ParameterOptimizationResults)</code>.</p>	'bayesopt'
AcquisitionFunctionName	<ul style="list-style-type: none"> <li>'expected-improvement-per-second-plus'</li> <li>'expected-improvement'</li> <li>'expected-improvement-plus'</li> <li>'expected-improvement-per-second'</li> <li>'lower-confidence-bound'</li> <li>'probability-of-improvement'</li> </ul> <p>For details, see the <code>bayesopt</code> <code>AcquisitionFunctionName</code> name-value pair, or “Acquisition Function Types” on page 10-4.</p>	'expected-improvement-per-second-plus'

Field Name	Values	Default
MaxObjectiveEvaluations	Maximum number of objective function evaluations.	30 for 'bayesopt' or 'randomsearch', and the entire grid for 'gridsearch'
MaxTime	Time limit, specified as a positive real. The time limit is in seconds, as measured by <code>tic</code> and <code>toc</code> . Run time can exceed <code>MaxTime</code> because <code>MaxTime</code> does not interrupt function evaluations.	Inf
NumGridDivisions	For 'gridsearch', the number of values in each dimension. Can be a vector of positive integers giving the number of values for each dimension, or a scalar that applies to all dimensions. Ignored for categorical variables.	10
ShowPlots	Logical value indicating whether to show plots. If <code>true</code> , plots the best objective function value against iteration number. If there are one or two optimization parameters, and if <code>Optimizer</code> is 'bayesopt', then <code>ShowPlots</code> also plots a model of the objective function against the parameters.	<code>true</code>
SaveIntermediateResults	Logical value indicating whether to save results when <code>Optimizer</code> is 'bayesopt'. If <code>true</code> , overwrites a workspace variable named 'BayesoptResults' at each iteration. The variable is a <code>BayesianOptimization</code> object.	<code>false</code>

Field Name	Values	Default
Verbose	Display to the command line. <ul style="list-style-type: none"> <li>• 0 — No iterative display</li> <li>• 1 — Iterative display</li> <li>• 2 — Iterative display with extra information</li> </ul> For details, see the <code>bayesopt</code> Verbose name-value pair.	1
UseParallel	Logical value indicating whether to run Bayesian optimization in parallel, which requires Parallel Computing Toolbox. For details, see “Parallel Bayesian Optimization” on page 10-9.	false
Repartition	Logical value indicating whether to repartition the cross-validation at every iteration. If <code>false</code> , the optimizer uses a single partition for the optimization.  <code>true</code> usually gives the most robust results because this setting takes partitioning noise into account. However, for good results, <code>true</code> requires at least twice as many function evaluations.	false
Use no more than one of the following three field names.		
CVPartition	A <code>cvpartition</code> object, as created by <code>cvpartition</code>	Kfold = 5
Holdout	A scalar in the range $(0, 1)$ representing the holdout fraction.	
Kfold	An integer greater than 1.	

## Variables for a Bayesian Optimization

### In this section...

“Syntax for Creating Optimization Variables” on page 10-39

“Variables for Optimization Examples” on page 10-40

### Syntax for Creating Optimization Variables

For each variable in your objective function, create a variable description object using `optimizableVariable`. Each variable has a unique name and a range of values. The minimal syntax for variable creation is

```
variable = optimizableVariable(Name,Range)
```

This function creates a real variable that ranges from the lower bound `Range(1)` to the upper bound `Range(2)`.

You can specify three types of variables in the `Type` name-value pair:

- `'real'` — Continuous real values between finite bounds. Give `Range` as the two-element vector `[lower upper]`, which represent the lower and upper bounds.
- `'integer'` — Integer values between finite bounds, similar to `'real'`.
- `'categorical'` — Cell array of names of possible values, such as `{'red','green','blue'}`, that you specify in the `Range` argument.

For `'real'` or `'integer'` variables, you can specify that `bayesopt` searches in a log-scaled space by setting the `Transform` name-value pair to `'log'`. For this transformation, ensure that the lower bound in the `Range` is strictly positive.

Include variables for `bayesopt` as a vector in the second argument.

```
results = bayesopt(fun,[xvar,ivar,rvar])
```

To exclude a variable from an optimization, set `Optimize` to `false`, either in the name-value pair of `optimizableVariable`, or by dot notation:

```
xvar.Optimize = false;
```

---

### Tip

- There are two names associated with an `optimizableVariable`:
  - The MATLAB workspace variable name
  - The name of the variable in the optimization

For example,

```
xvar = optimizableVariable('spacevar',[1,100]);
```

`xvar` is the MATLAB workspace variable, and `'spacevar'` is the variable in the optimization.

Use these names as follows:

- Use `xvar` as an element in the vector of variables you pass to `bayesopt`. For example,

```
results = bayesopt(fun,[xvar,tvar])
```

- Use `'spacevar'` as the name of the variable in the optimization. For example, in an objective function,

```
function objective = mysvmfun(x,cdata,grp)
SVMModel = fitcsvm(cdata,grp,'KernelFunction','rbf',...
    'BoxConstraint',x.spacevar,...
    'KernelScale',x.tvar);
objective = kfoldLoss(crossval(SVMModel));
```

---

## Variables for Optimization Examples

Real variable from 0 to 1:

```
var1 = optimizableVariable('xvar',[0 1])
```

```
var1 =
```

```
optimizableVariable with properties:
```

```
    Name: 'xvar'
   Range: [0 1]
    Type: 'real'
 Transform: 'none'
```



```
Optimize: 1
```

### Integer variable from 1 to 1000 on a log scale:

```
var2 = optimizableVariable('ivar', [1 1000], 'Type', 'integer', 'Transform', 'log')
```

```
var2 =
```

```
optimizableVariable with properties:
```

```
    Name: 'ivar'  
    Range: [1 1000]  
    Type: 'integer'  
    Transform: 'log'  
    Optimize: 1
```

### Categorical variable of rainbow colors:

```
var3 = optimizableVariable('rvar', {'r' 'o' 'y' 'g' 'b' 'i' 'v'}, 'Type', 'categorical')
```

```
var3 =
```

```
optimizableVariable with properties:
```

```
    Name: 'rvar'  
    Range: {'r' 'o' 'y' 'g' 'b' 'i' 'v'}  
    Type: 'categorical'  
    Transform: 'none'  
    Optimize: 1
```

## See Also

### Related Examples

- “Bayesian Optimization Workflow” on page 10-30
- “Bayesian Optimization Objective Functions” on page 10-42

## Bayesian Optimization Objective Functions

### In this section...

“Objective Function Syntax” on page 10-42

“Objective Function Example” on page 10-42

“Objective Function Errors” on page 10-43

### Objective Function Syntax

`bayesopt` attempts to minimize an objective function. If, instead, you want to maximize a function, set the objective function to the negative of the function you want to maximize. See “Maximizing Functions” (MATLAB).

`bayesopt` passes a table of variables to the objective function. The variables have the names and types that you declare; see “Variables for a Bayesian Optimization” on page 10-39.

The objective function has the following signature:

```
[objective,coupledconstraints,userdata] = fun(x)
```

- 1 `objective` — The objective function value at `x`, a real scalar.
- 2 `coupledconstraints` — Value of coupled constraints, if any (optional output), a vector of real values. A negative value indicates that a constraint is satisfied, a positive value indicates that it is not satisfied. For details, see “Coupled Constraints” on page 10-47.
- 3 `userdata` — Optional data that your function can return for further uses, such as plotting or logging (optional output). For an example, see “Bayesian Optimization Plot Functions” on page 10-14.

### Objective Function Example

This objective function returns the loss in a cross-validated fit of an SVM model with parameters `box` and `sigma`. The objective also returns a coupled constraint function that is positive (infeasible) when the number of support vectors exceeds 100 (100 is feasible, 101 is not).

```
function [objective,constraint] = mysvmfun(x,cdata,grp)
SVMModel = fitcsvm(cdata,grp,'KernelFunction','rbf',...
```

```
'BoxConstraint', x.box, ...  
'KernelScale', x.sigma);  
objective = kfoldLoss(crossval(SVMModel));  
constraint = sum(SVMModel.SupportVectors) - 100.5;
```

## Objective Function Errors

`bayesopt` deems your objective function to return an error when the objective function returns anything other than a finite real scalar. For example, if your objective function returns a complex value, NaN, Inf, or matrix with more than one entry, then `bayesopt` deems that your objective function errors. If `bayesopt` encounters an error, it continues to optimize, and automatically updates a Bayesian model of points that lead to errors. This Bayesian model is the Error model. `bayesopt` incorporates the Error model as a coupled constraint. See “Coupled Constraints” on page 10-47.

When errors exist, you can plot the Error model by setting the `bayesopt` `PlotFcn` name-value pair `@plotConstraintModels`. Or you can retrospectively call `plot` on the results of a Bayesian optimization, and include `@plotConstraintModels`.

## See Also

### Related Examples

- “Bayesian Optimization Workflow” on page 10-30
- “Variables for a Bayesian Optimization” on page 10-39
- “Constraints in Bayesian Optimization” on page 10-44

## Constraints in Bayesian Optimization

In this section...
“Bounds” on page 10-44
“Deterministic Constraints — XConstraintFcn” on page 10-44
“Conditional Constraints — ConditionalVariableFcn” on page 10-45
“Coupled Constraints” on page 10-47
“Bayesian Optimization with Coupled Constraints” on page 10-48

### Bounds

`bayesopt` requires finite bounds on all variables. (categorical variables are, by nature, bounded in their possible values.) Pass the lower and upper bounds for real and integer-valued variables in `optimizableVariable`.

`bayesopt` uses these bounds to sample points, either uniformly or log-scaled. You set the scaling for sampling in `optimizableVariable`.

For example, to constrain a variable `X1` to values between `1e-6` and `1e3`, scaled logarithmically,

```
xvar = optimizableVariable('X1', [1e-6, 1e3], 'Transform', 'log')
```

`bayesopt` includes the endpoints in its range. Therefore, you cannot use 0 as a lower bound for a log-transformed variable.

---

**Tip** To use a zero lower bound in a log-transformed variable, set the lower bound to 1, then inside the objective function use  $x-1$ .

---

### Deterministic Constraints — XConstraintFcn

Sometimes your problem is valid or well-defined only for points in a certain region, called the feasible region. A deterministic constraint is a deterministic function that returns `true` when a point is feasible, and `false` when a point is infeasible. So deterministic constraints are not stochastic, and they are not functions of a group of points, but of individual points.

---

**Tip** It is more efficient to use `optimizableVariable` bounds, instead of deterministic constraints, to confine the optimization to a rectangular region.

---

Write a deterministic constraint function using the signature

```
tf = xconstraint(X)
```

- `X` is a width- $D$  table of arbitrary height.
- `tf` is a logical column vector, where `tf(i) = true` exactly when `X(i, :)` is feasible.

Pass the deterministic constraint function in the `bayesopt` `XConstraintFcn` name-value pair. For example,

```
results = bayesopt(fun, vars, 'XConstraintFcn', @xconstraint)
```

`bayesopt` evaluates deterministic constraints on thousands of points, and so runs faster when your constraint function is vectorized. See “Vectorization” (MATLAB).

For example, suppose that the variables named '`x1`' and '`x2`' are feasible when the norm of the vector `[x1 x2]` is less than 6, and when `x1 <= x2`. The following constraint function evaluates these constraints.

```
function tf = xconstraint(X)
tf1 = sqrt(X.x1.^2 + X.x2.^2) < 6;
tf2 = X.x1 <= X.x2;
tf = tf1 & tf2;
```

## Conditional Constraints — `ConditionalVariableFcn`

Conditional constraints are functions that enforce one of the following two conditions:

- When some variables have certain values, other variables are set to given values.
- When some variables have certain values, other variables have NaN or, for categorical variables, `<undefined>` values.

Specify a conditional constraint by setting the `bayesopt` `ConditionalVariableFcn` name-value pair to a function handle, say `@condvariablefcn`. The `@condvariablefcn` function must have the signature

```
Xnew = condvariablefcn(X)
```

- X is a width-D table of arbitrary height.
- Xnew is a table the same type and size as X.

condvariablefcn sets Xnew to be equal to X, except it also sets the relevant variables in each row of Xnew to the correct values for the constraint.

---

**Note** If you have both conditional constraints and deterministic constraints, `bayesopt` applies the conditional constraints first. Therefore, if your conditional constraint function can set variables to NaN or `<undefined>`, ensure that your deterministic constraint function can process these values correctly.

Conditional constraints ensure that variable values are sensible. Therefore, `bayesopt` applies conditional constraints first so that all passed values are sensible.

---

### Conditional Constraint That Sets a Variable Value

Suppose that you are optimizing a classification using `fitcdiscr`, and you optimize over both `DiscrimType` and `Gamma`. When `DiscrimType` is one of the quadratic types, `Gamma` must be 0 or the solver errors. In that case, use this conditional constraint function:

```
function XTable = fitcdiscrCVF(XTable)
% Gamma must be 0 if discrim type is a quadratic
XTable.Gamma(ismember(XTable.DiscrimType, {'quadratic',...
      'diagQuadratic', 'pseudoQuadratic'})) = 0;
end
```

### Conditional Constraint That Sets a Variable to NaN

Suppose that you are optimizing a classification using `fitcsvm`, and you optimize over both `KernelFunction` and `PolynomialOrder`. When `KernelFunction` is not 'polynomial', the `PolynomialOrder` setting does not apply. The following function enforces this conditional constraint.

```
function Xnew = condvariablefcn(X)
Xnew = X;
Xnew.PolynomialOrder(Xnew.KernelFunction ~= 'polynomial') = NaN;
```

You can save a line of code as follows:

```
function X = condvariablefcn(X)
X.PolynomialOrder(Xnew.KernelFunction ~= 'polynomial') = NaN;
```

## Coupled Constraints

Coupled constraints are constraints that you can evaluate only by calling the objective function. These constraints can be stochastic or deterministic. Return these constraint values from your objective function in the second argument. See “Bayesian Optimization Objective Functions” on page 10-42.

The objective function returns a numeric vector for the coupled constraints, one entry for each coupled constraint. For each entry, a negative value indicates that the constraint is satisfied (also called feasible). A positive value indicates that the constraint is not satisfied (infeasible).

`bayesopt` automatically creates a coupled constraint, called the Error constraint, for every run. This constraint enables `bayesopt` to model points that cause errors in objective function evaluation. For details, see “Objective Function Errors” on page 10-43 and `predictError`.

If you have coupled constraints in addition to the Error constraint:

- Include the `NumCoupledConstraints` name-value pair in your `bayesopt` call (required). Do not include the Error constraint in this number.
- If any of your coupled constraints are stochastic, include the `AreCoupledConstraintsDeterministic` name-value pair and pass `false` for any stochastic constraint.

Observe the coupled constraint values in each iteration by setting the `bayesopt` `Verbose` name-value pair to 1 or 2.

---

**Note** When there are coupled constraints, iterative display and plot functions can give counterintuitive results such as:

- A *minimum objective* plot can increase.
- The optimization can declare a problem infeasible even when it showed an earlier feasible point.

The reason for this behavior is that the decision about whether a point is feasible can change as the optimization progresses. `bayesopt` determines feasibility with respect to its constraint model, and this model changes as `bayesopt` evaluates points. So a “minimum objective” plot can increase when the minimal point is later deemed

infeasible, and the iterative display can show a feasible point that is later deemed infeasible.

---

For an example, see “Bayesian Optimization with Coupled Constraints” on page 10-48.

## Bayesian Optimization with Coupled Constraints

A coupled constraint is one that can be evaluated only by evaluating the objective function. In this case, the objective function is the cross-validated loss of an SVM model. The coupled constraint is that the number of support vectors is no more than 100. The model details are in “Optimize a Cross-Validated SVM Classifier Using Bayesian Optimization” on page 10-53.

Create the data for classification.

```
rng default
grnpop = mvnrnd([1,0],eye(2),10);
redpop = mvnrnd([0,1],eye(2),10);
redpts = zeros(100,2);
grnpts = redpts;
for i = 1:100
    grnpts(i,:) = mvnrnd(grnpop(randi(10),:),eye(2)*0.02);
    redpts(i,:) = mvnrnd(redpop(randi(10),:),eye(2)*0.02);
end
cdata = [grnpts;redpts];
grp = ones(200,1);
grp(101:200) = -1;
c = cvpartition(200,'KFold',10);
sigma = optimizableVariable('sigma',[1e-5,1e5],'Transform','log');
box = optimizableVariable('box',[1e-5,1e5],'Transform','log');
```

The objective function is the cross-validation loss of the SVM model for partition `c`. The coupled constraint is the number of support vectors minus 100.5. This ensures that 100 support vectors give a negative constraint value, but 101 support vectors give a positive value. The model has 200 data points, so the coupled constraint values range from -99.5 (there is always at least one support vector) to 99.5. Positive values mean the constraint is not satisfied.

```
function [objective,constraint] = mysvmfun(x,cdata,grp,c)
SVMModel = fitcsvm(cdata,grp,'KernelFunction','rbf',...
    'BoxConstraint',x.box,...
```



```

    'KernelScale', x.sigma);
cvModel = crossval(SVMModel, 'CVPartition', c);
objective = kfoldLoss(cvModel);
constraint = sum(SVMModel.IsSupportVector)-100.5;

fun = @(x)mysvmfun(x, cdata, grp, c);

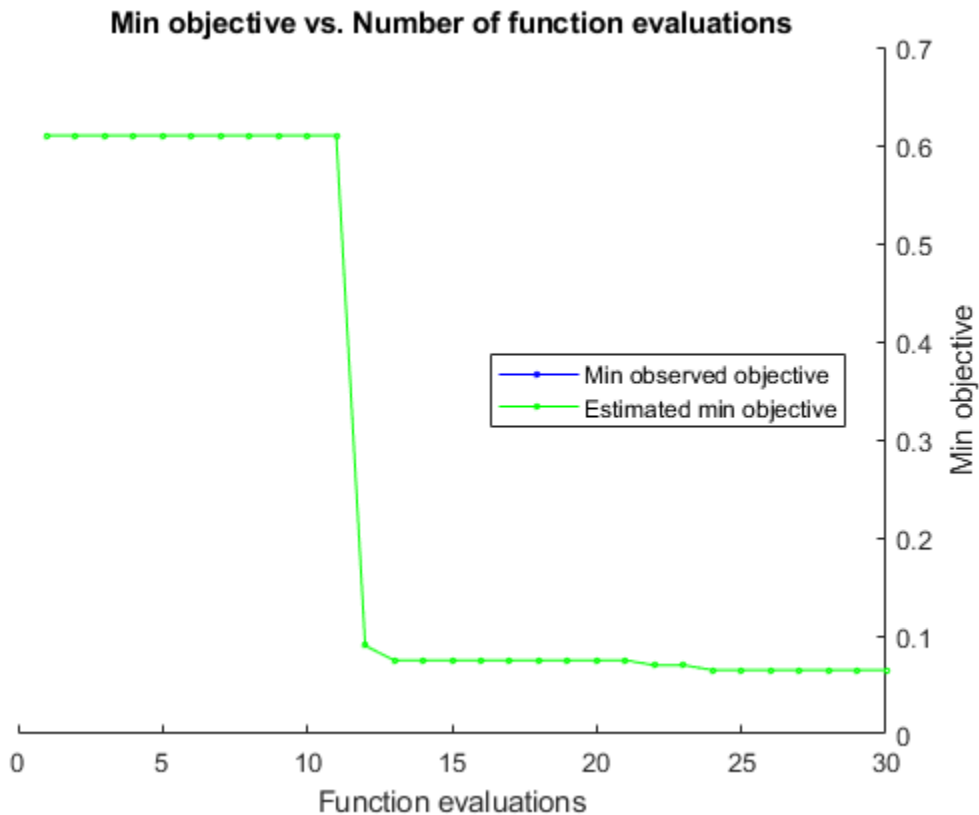
```

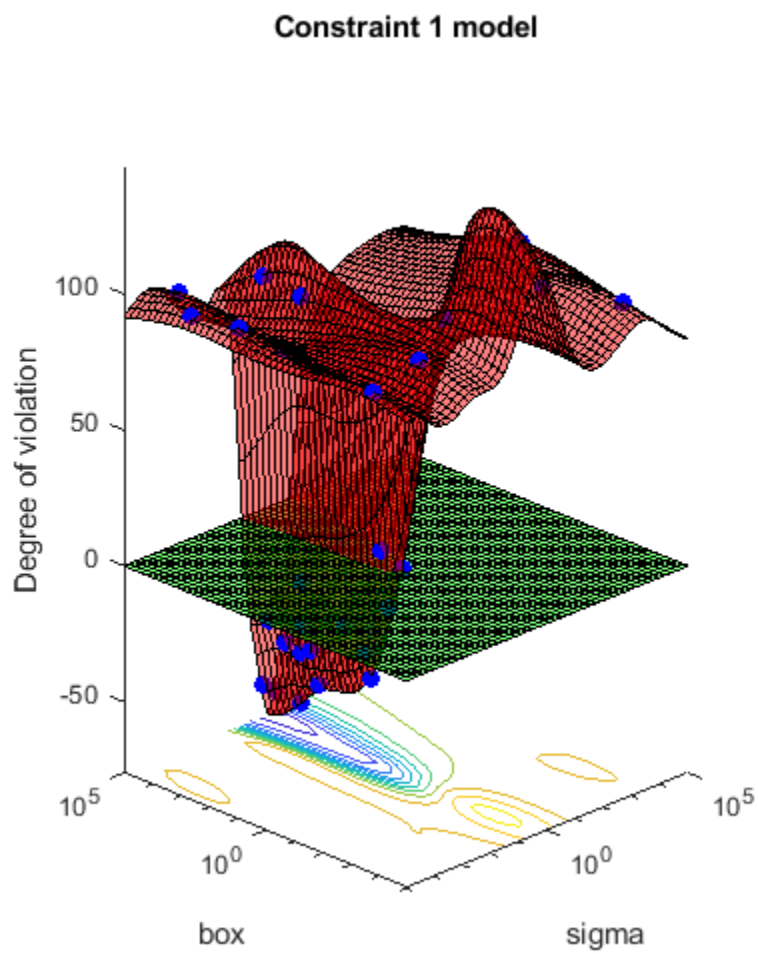
Set the `NumCoupledConstraints` to 1 so the optimizer knows that there is a coupled constraint. Set options to plot the constraint model.

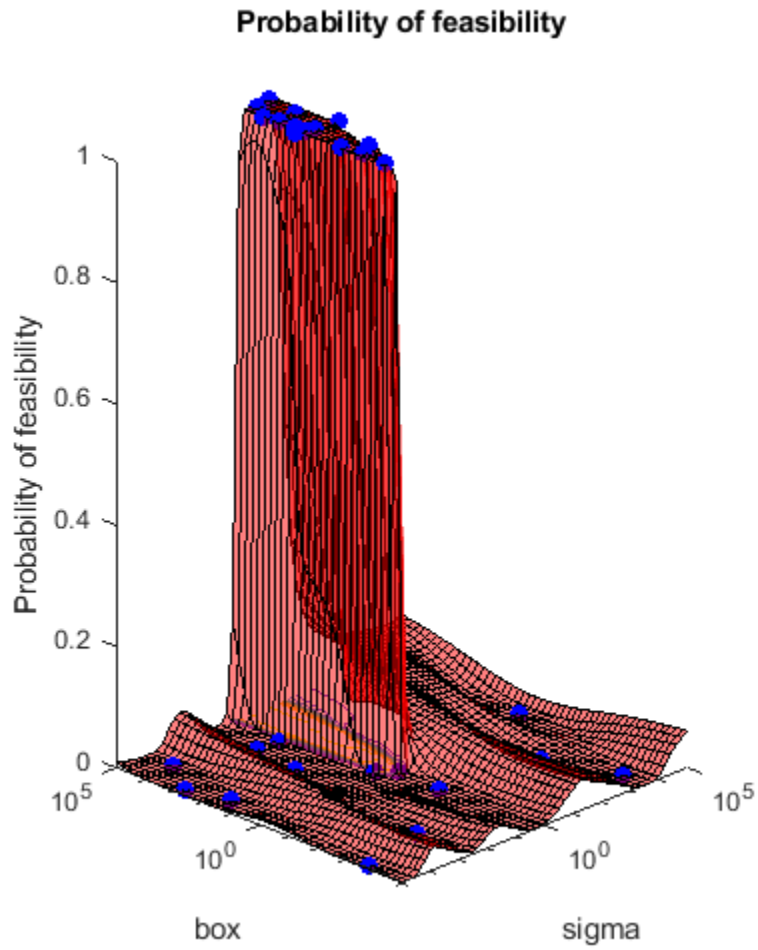
```

results = bayesopt(fun, [sigma, box], 'IsObjectiveDeterministic', true, ...
    'NumCoupledConstraints', 1, 'PlotFcn', ...
    {@plotMinObjective, @plotConstraintModels}, ...
    'AcquisitionFunctionName', 'expected-improvement-plus', 'Verbose', 0);

```







Most points lead to an infeasible number of support vectors.

## See Also

### Related Examples

- “Bayesian Optimization Workflow” on page 10-30
- “Variables for a Bayesian Optimization” on page 10-39
- “Bayesian Optimization Objective Functions” on page 10-42

## Optimize a Cross-Validated SVM Classifier Using Bayesian Optimization

This example shows how to optimize an SVM classification. The classification works on locations of points from a Gaussian mixture model. In *The Elements of Statistical Learning*, Hastie, Tibshirani, and Friedman (2009), page 17 describes the model. The model begins with generating 10 base points for a "green" class, distributed as 2-D independent normals with mean (1,0) and unit variance. It also generates 10 base points for a "red" class, distributed as 2-D independent normals with mean (0,1) and unit variance. For each class (green and red), generate 100 random points as follows:

- 1 Choose a base point  $m$  of the appropriate color uniformly at random.
- 2 Generate an independent random point with 2-D normal distribution with mean  $m$  and variance  $I/5$ , where  $I$  is the 2-by-2 identity matrix. In this example, use a variance  $I/50$  to show the advantage of optimization more clearly.

After generating 100 green and 100 red points, classify them using `fitcsvm`. Then use `bayesopt` to optimize the parameters of the resulting SVM model with respect to cross validation.

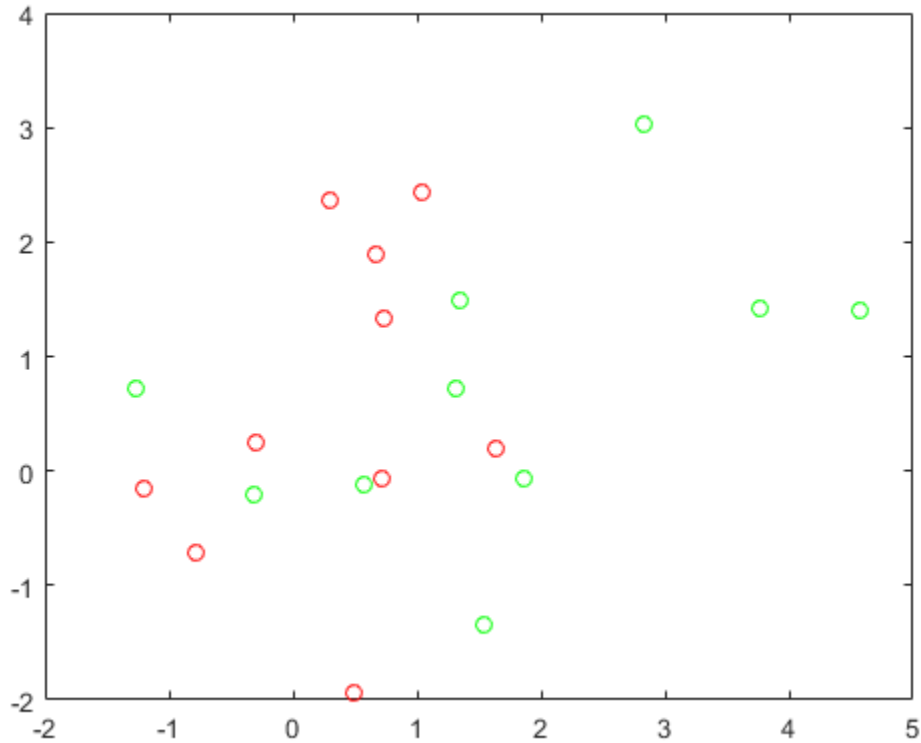
### Generate the Points and Classifier

Generate the 10 base points for each class.

```
rng default
grnpop = mvnrnd([1,0],eye(2),10);
redpop = mvnrnd([0,1],eye(2),10);
```

View the base points.

```
plot(grnpop(:,1),grnpop(:,2),'go')
hold on
plot(redpop(:,1),redpop(:,2),'ro')
hold off
```



Since some red base points are close to green base points, it can be difficult to classify the data points based on location alone.

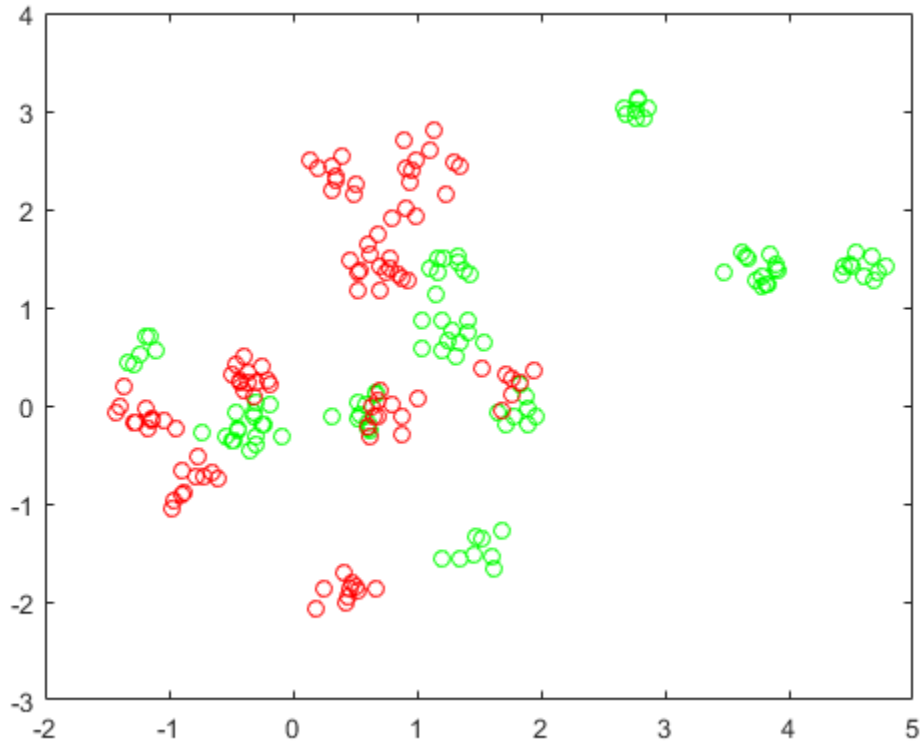
Generate the 100 data points of each class.

```
redpts = zeros(100,2);grnpts = redpts;
for i = 1:100
    grnpts(i,:) = mvnrnd(grnpop(randi(10),:),eye(2)*0.02);
    redpts(i,:) = mvnrnd(redpop(randi(10),:),eye(2)*0.02);
end
```

View the data points.

```
figure
plot(grnpts(:,1),grnpts(:,2),'go')
```

```
hold on
plot(redpts(:,1),redpts(:,2),'ro')
hold off
```



### Prepare Data For Classification

Put the data into one matrix, and make a vector `grp` that labels the class of each point.

```
cdata = [grnpts;redpts];
grp = ones(200,1);
% Green label 1, red label -1
grp(101:200) = -1;
```

### Prepare Cross-Validation

Set up a partition for cross-validation. This step fixes the train and test sets that the optimization uses at each step.

```
c = cvpartition(200, 'KFold', 10);
```

### Prepare Variables for Bayesian Optimization

Set up a function that takes an input  $z = [\text{rbf\_sigma}, \text{boxconstraint}]$  and returns the cross-validation loss value of  $z$ . Take the components of  $z$  as positive, log-transformed variables between  $1e-5$  and  $1e5$ . Choose a wide range, because you don't know which values are likely to be good.

```
sigma = optimizableVariable('sigma', [1e-5, 1e5], 'Transform', 'log');
box = optimizableVariable('box', [1e-5, 1e5], 'Transform', 'log');
```

### Objective Function

This function handle computes the cross-validation loss at parameters  $[\text{sigma}, \text{box}]$ . For details, see `ClassificationPartitionedModel.kfoldLoss`.

`bayesopt` passes the variable  $z$  to the objective function as a one-row table.

```
minfn = @(z) kfoldLoss(fitcsvm(cdata, grp, 'CVPartition', c, ...
    'KernelFunction', 'rbf', 'BoxConstraint', z.box, ...
    'KernelScale', z.sigma));
```

### Optimize Classifier

Search for the best parameters  $[\text{sigma}, \text{box}]$  using `bayesopt`. For reproducibility, choose the 'expected-improvement-plus' acquisition function. The default acquisition function depends on run time, and so can give varying results.

```
results = bayesopt(minfn, [sigma, box], 'IsObjectiveDeterministic', true, ...
    'AcquisitionFunctionName', 'expected-improvement-plus')
```

```
=====
| Iter | Eval  | Objective  | Objective  | BestSoFar  | BestSoFar  |          sigma
|      | result |            | runtime    | (observed) | (estim.)   |
|=====|=====|=====|=====|=====|=====|=====
|  1  | Best  | 0.61      | 0.2363     | 0.61      | 0.61      | 0.00013375
|  2  | Best  | 0.345     | 0.25656    | 0.345     | 0.345     | 24526
|  3  | Accept | 0.61     | 0.24767    | 0.345     | 0.345     | 0.0026459
```



4	Accept	0.345	0.30048	0.345	0.345	3506.3
5	Accept	0.345	0.25811	0.345	0.345	9135.2
6	Accept	0.345	0.25434	0.345	0.345	99701
7	Best	0.295	0.23028	0.295	0.295	455.88
8	Best	0.24	4.3282	0.24	0.24	31.56
9	Accept	0.24	5.0777	0.24	0.24	10.451
10	Accept	0.35	0.25787	0.24	0.24	17.331
11	Best	0.23	3.2936	0.23	0.23	16.005
12	Best	0.1	0.45354	0.1	0.1	0.36562
13	Accept	0.115	0.25502	0.1	0.1	0.1793
14	Accept	0.105	0.30202	0.1	0.1	0.2267
15	Best	0.095	0.26685	0.095	0.095	0.28999
16	Best	0.075	0.24041	0.075	0.075	0.30554
17	Accept	0.085	0.21978	0.075	0.075	0.41122
18	Accept	0.085	0.24691	0.075	0.075	0.25565
19	Accept	0.075	0.2465	0.075	0.075	0.32869
20	Accept	0.085	0.23484	0.075	0.075	0.32442
=====						
Iter	Eval	Objective	Objective	BestSoFar	BestSoFar	sigma
	result		runtime	(observed)	(estim.)	
=====						
21	Accept	0.3	0.2144	0.075	0.075	1.3592
22	Accept	0.12	0.23323	0.075	0.075	0.17515
23	Accept	0.175	0.24112	0.075	0.075	0.1252
24	Accept	0.105	0.25443	0.075	0.075	1.1664
25	Accept	0.1	0.28356	0.075	0.075	0.57465
26	Accept	0.12	0.24071	0.075	0.075	0.42922
27	Accept	0.12	0.23573	0.075	0.075	0.42956
28	Accept	0.095	0.23936	0.075	0.075	0.4806
29	Accept	0.105	0.26069	0.075	0.075	0.19755
30	Accept	0.205	0.30794	0.075	0.075	3.5051

Optimization completed.

MaxObjectiveEvaluations of 30 reached.

Total function evaluations: 30

Total elapsed time: 64.0366 seconds.

Total objective function evaluation time: 19.7181

Best observed feasible point:

sigma	box
-----	-----
0.30554	8.9017

```

Observed objective function value = 0.075
Estimated objective function value = 0.075
Function evaluation time = 0.24041

```

```

Best estimated feasible point (according to models):

```

sigma	box
0.32869	18.076

```

Estimated objective function value = 0.075
Estimated function evaluation time = 0.24164

```

```

results =

```

```

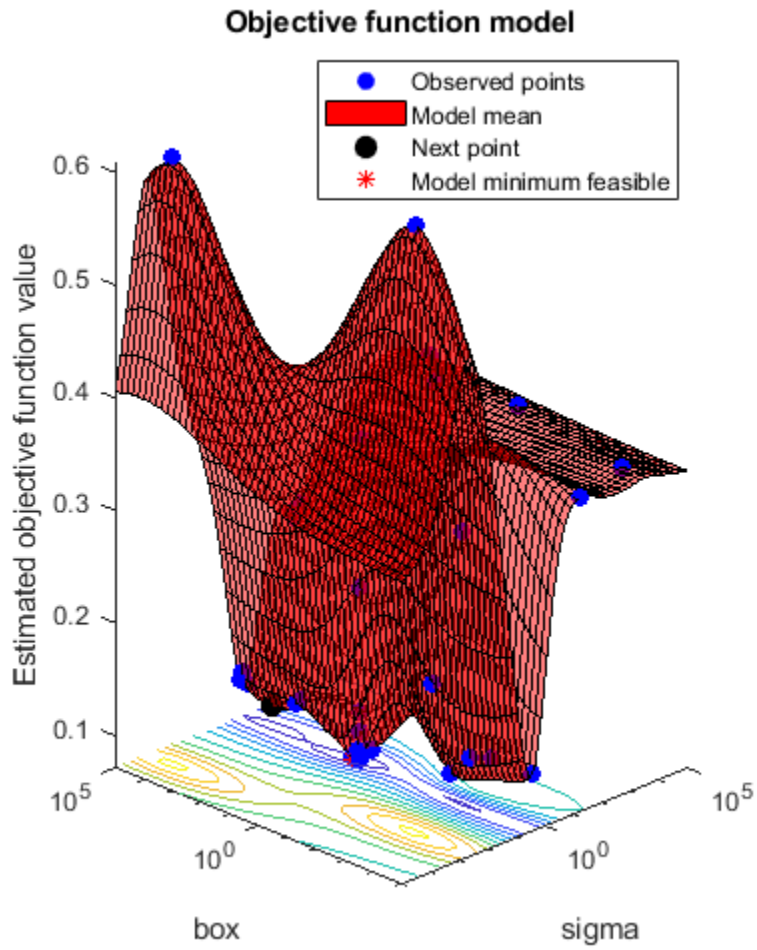
BayesianOptimization with properties:

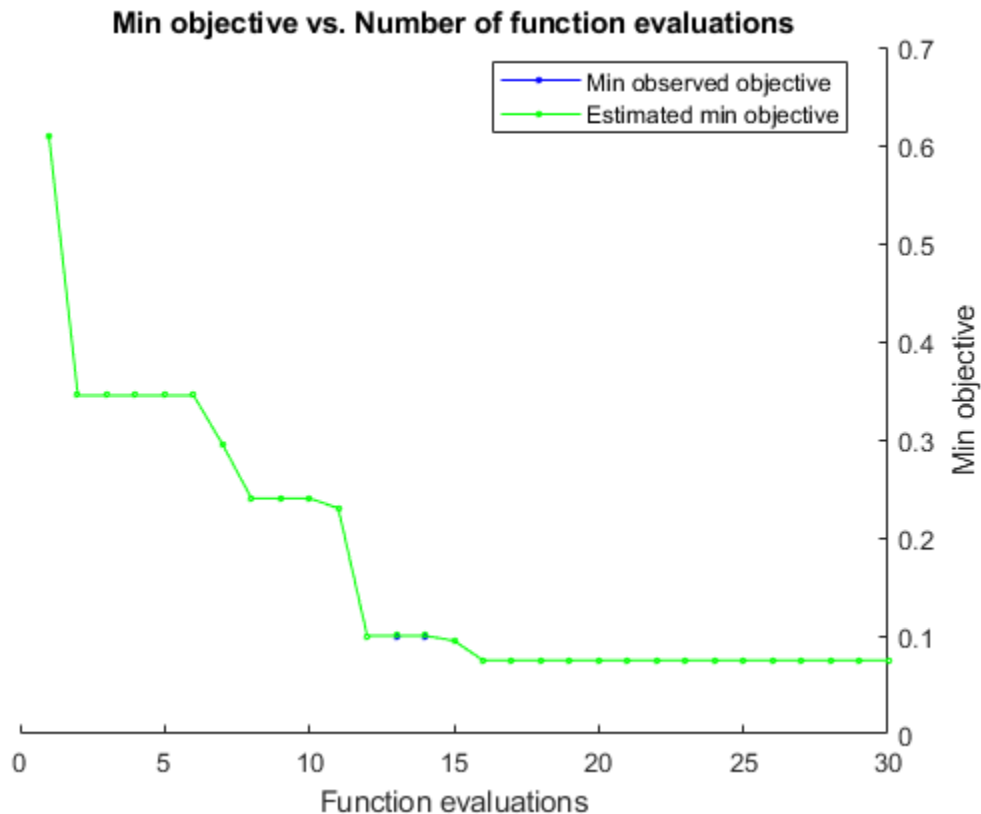
```

```

    ObjectiveFcn: [function_handle]
  VariableDescriptions: [1x2 optimizableVariable]
        Options: [1x1 struct]
    MinObjective: 0.0750
    XAtMinObjective: [1x2 table]
  MinEstimatedObjective: 0.0750
XAtMinEstimatedObjective: [1x2 table]
  NumObjectiveEvaluations: 30
    TotalElapsedTime: 64.0366
      NextPoint: [1x2 table]
        XTrace: [30x2 table]
    ObjectiveTrace: [30x1 double]
    ConstraintsTrace: []
      UserDataTrace: {30x1 cell}
  ObjectiveEvaluationTimeTrace: [30x1 double]
    IterationTimeTrace: [30x1 double]
      ErrorTrace: [30x1 double]
    FeasibilityTrace: [30x1 logical]
  FeasibilityProbabilityTrace: [30x1 double]
    IndexOfMinimumTrace: [30x1 double]
    ObjectiveMinimumTrace: [30x1 double]
  EstimatedObjectiveMinimumTrace: [30x1 double]

```





Use the results to train a new, optimized SVM classifier.

```
z(1) = results.XAtMinObjective.sigma;
z(2) = results.XAtMinObjective.box;
SVMModel = fitcsvm(cdata,grp,'KernelFunction','rbf',...
    'KernelScale',z(1),'BoxConstraint',z(2));
```

Plot the classification boundaries. To visualize the support vector classifier, predict scores over a grid.

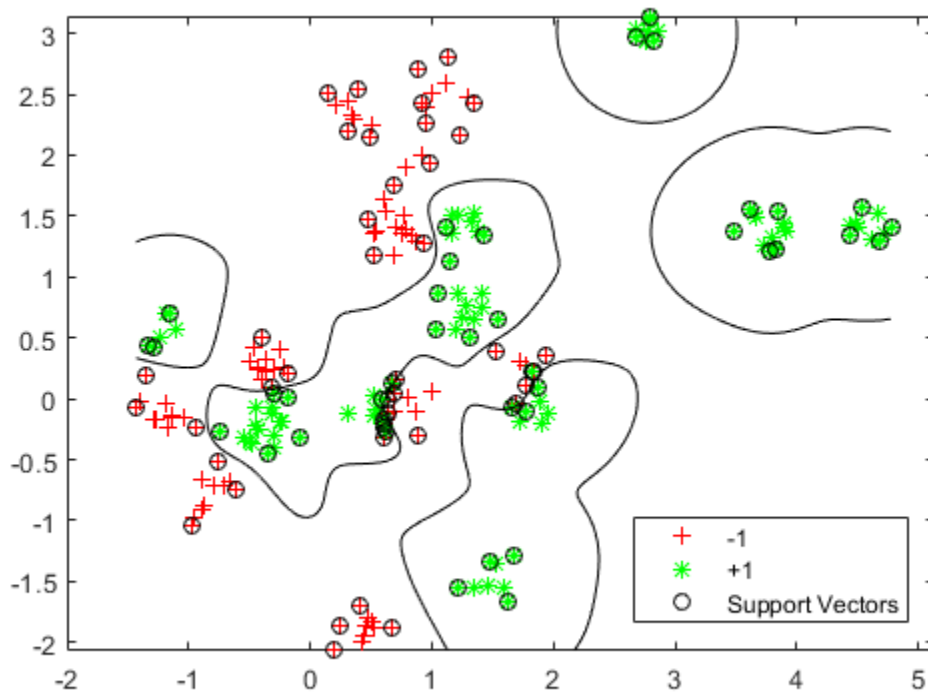
```
d = 0.02;
[x1Grid,x2Grid] = meshgrid(min(cdata(:,1)):d:max(cdata(:,1)),...
    min(cdata(:,2)):d:max(cdata(:,2)));
xGrid = [x1Grid(:),x2Grid(:)];
```

```

[~,scores] = predict(SVMModel,xGrid);

h = nan(3,1); % Preallocation
figure;
h(1:2) = gscatter(cdata(:,1),cdata(:,2),grp,'rg','+*');
hold on
h(3) = plot(cdata(SVMModel.IsSupportVector,1),...
            cdata(SVMModel.IsSupportVector,2),'ko');
contour(x1Grid,x2Grid,reshape(scores(:,2),size(x1Grid)),[0 0],'k');
legend(h,{'-1','+1','Support Vectors'},'Location','Southeast');
axis equal
hold off

```



## Evaluate Accuracy on New Data

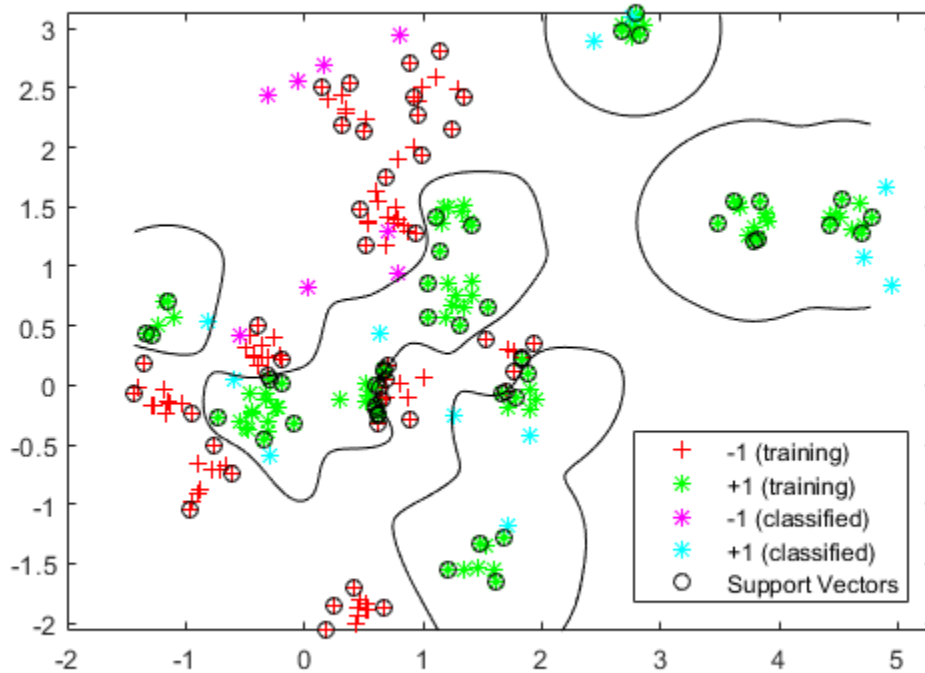
Generate and classify some new data points.

```
grnobj = gmdistribution(grnpop, .2*eye(2));
redobj = gmdistribution(redpop, .2*eye(2));

newData = random(grnobj,10);
newData = [newData;random(redobj,10)];
grpData = ones(20,1);
grpData(11:20) = -1; % red = -1

v = predict(SVMModel,newData);

g = nan(7,1);
figure;
h(1:2) = gscatter(cdata(:,1),cdata(:,2),grp,'rg','+*');
hold on
h(3:4) = gscatter(newData(:,1),newData(:,2),v,'mc','**');
h(5) = plot(cdata(SVMModel.IsSupportVector,1),...
           cdata(SVMModel.IsSupportVector,2),'ko');
contour(x1Grid,x2Grid,reshape(scores(:,2),size(x1Grid)),[0 0],'k');
legend(h(1:5),{'-1 (training)', '+1 (training)', '-1 (classified)', ...
              '+1 (classified)', 'Support Vectors'}, 'Location', 'Southeast');
axis equal
hold off
```



See which new data points are correctly classified. Circle the correctly classified points in red, and the incorrectly classified points in black.

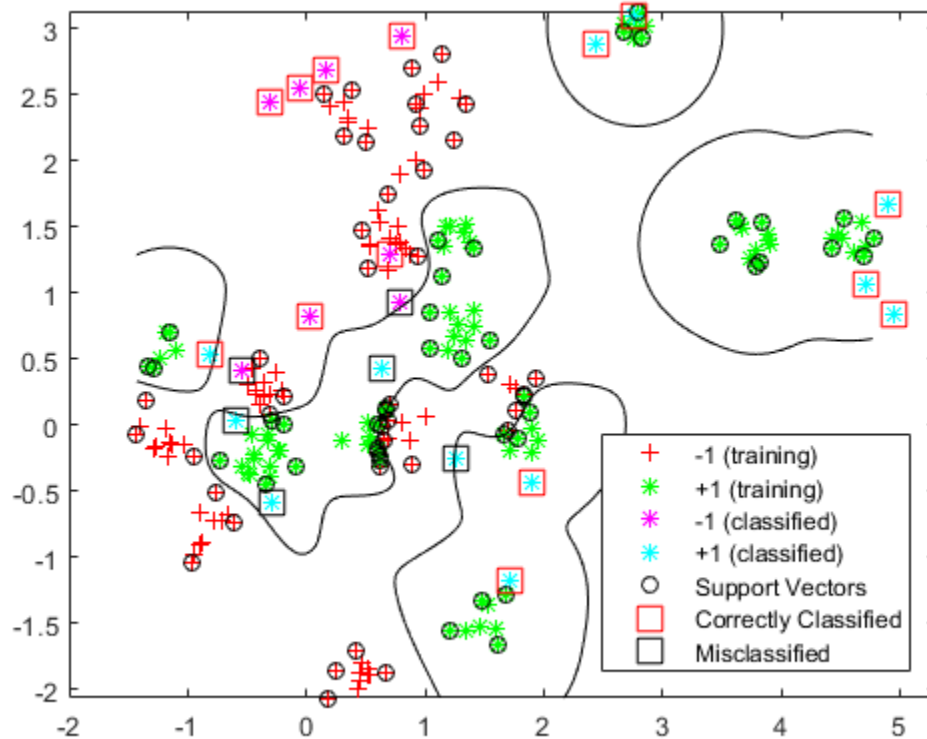
```
mydiff = (v == grpData); % Classified correctly
figure;
h(1:2) = gscatter(cdata(:,1),cdata(:,2),grp,'rg','+*');
hold on
h(3:4) = gscatter(newData(:,1),newData(:,2),v,'mc','**');
h(5) = plot(cdata(SVMModel.IsSupportVector,1),...
    cdata(SVMModel.IsSupportVector,2),'ko');
contour(x1Grid,x2Grid,reshape(scores(:,2),size(x1Grid)),[0 0],'k');

for ii = mydiff % Plot red squares around correct pts
    h(6) = plot(newData(ii,1),newData(ii,2),'rs','MarkerSize',12);
```

```
end

for ii = not(mydiff) % Plot black squares around incorrect pts
    h(7) = plot(newData(ii,1),newData(ii,2),'ks','MarkerSize',12);
end
legend(h,{'-1 (training)','+1 (training)','-1 (classified)',...
    '+1 (classified)','Support Vectors','Correctly Classified',...
    'Misclassified'},'Location','Southeast');
hold off
```





## See Also

### Related Examples

- “Optimize an SVM Classifier Fit Using Bayesian Optimization” on page 10-66

## Optimize an SVM Classifier Fit Using Bayesian Optimization

This example shows how to optimize an SVM classification using the `fitcsvm` function and `OptimizeHyperparameters` name-value pair. The classification works on locations of points from a Gaussian mixture model. In *The Elements of Statistical Learning*, Hastie, Tibshirani, and Friedman (2009), page 17 describes the model. The model begins with generating 10 base points for a "green" class, distributed as 2-D independent normals with mean (1,0) and unit variance. It also generates 10 base points for a "red" class, distributed as 2-D independent normals with mean (0,1) and unit variance. For each class (green and red), generate 100 random points as follows:

- 1 Choose a base point  $m$  of the appropriate color uniformly at random.
- 2 Generate an independent random point with 2-D normal distribution with mean  $m$  and variance  $I/5$ , where  $I$  is the 2-by-2 identity matrix. In this example, use a variance  $I/50$  to show the advantage of optimization more clearly.

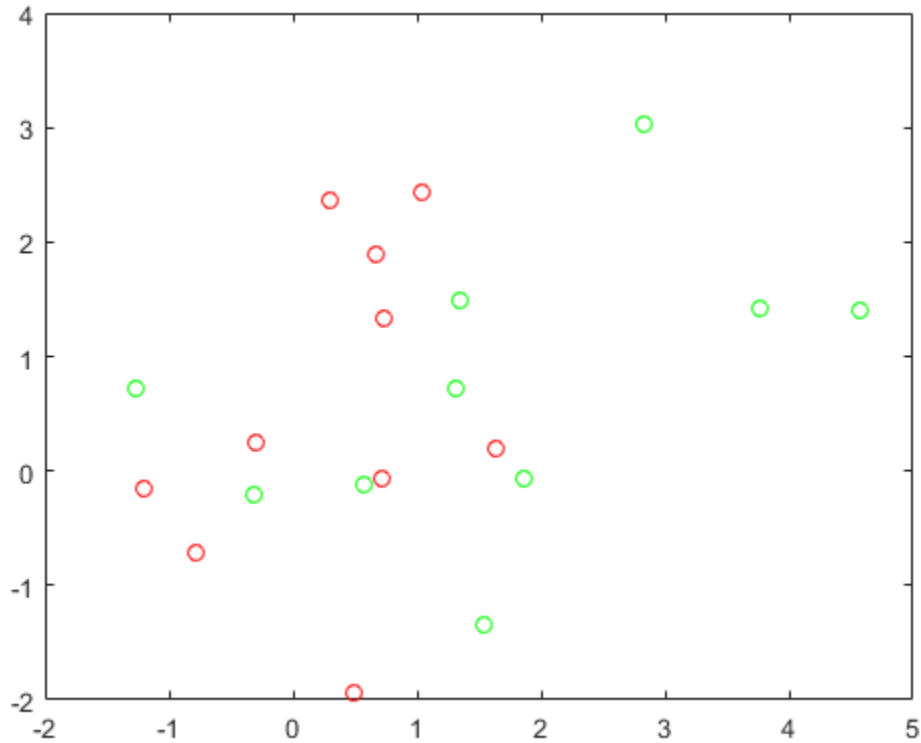
### Generate the Points and Classifier

Generate the 10 base points for each class.

```
rng default % For reproducibility
grnpop = mvnrnd([1,0],eye(2),10);
redpop = mvnrnd([0,1],eye(2),10);
```

View the base points.

```
plot(grnpop(:,1),grnpop(:,2),'go')
hold on
plot(redpop(:,1),redpop(:,2),'ro')
hold off
```



Since some red base points are close to green base points, it can be difficult to classify the data points based on location alone.

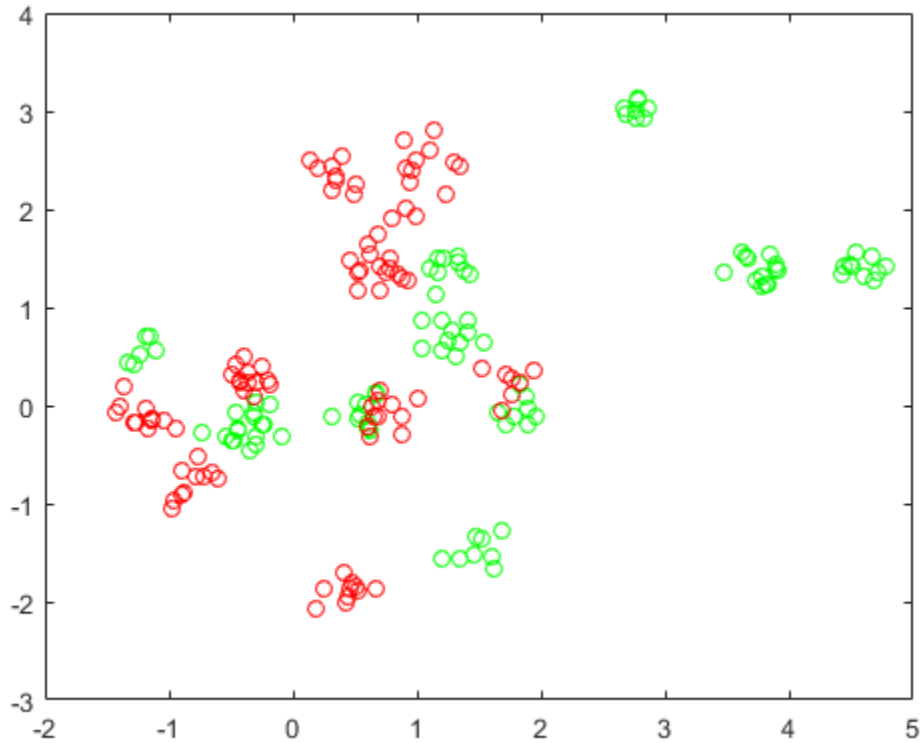
Generate the 100 data points of each class.

```
redpts = zeros(100,2);grnpts = redpts;  
for i = 1:100  
    grnpts(i,:) = mvnrnd(grnpop(randi(10),:),eye(2)*0.02);  
    redpts(i,:) = mvnrnd(redpop(randi(10),:),eye(2)*0.02);  
end
```

View the data points.

```
figure  
plot(grnpts(:,1),grnpts(:,2),'go')
```

```
hold on
plot(redpts(:,1),redpts(:,2),'ro')
hold off
```



### Prepare Data For Classification

Put the data into one matrix, and make a vector `grp` that labels the class of each point.

```
cdata = [grnpts;redpts];
grp = ones(200,1);
% Green label 1, red label -1
grp(101:200) = -1;
```

## Prepare Cross-Validation

Set up a partition for cross-validation. This step fixes the train and test sets that the optimization uses at each step.

```
c = cvpartition(200, 'Kfold', 10);
```

## Optimize the Fit

To find a good fit, meaning one with a low cross-validation loss, set options to use Bayesian optimization. Use the same cross-validation partition `c` in all optimizations.

For reproducibility, use the 'expected-improvement-plus' acquisition function.

```
opts = struct('Optimizer','bayesopt','ShowPlots',true,'CVPartition',c,...
    'AcquisitionFunctionName','expected-improvement-plus');
svmmod = fitcsvm(cdata,grp,'KernelFunction','rbf',...
    'OptimizeHyperparameters','auto','HyperparameterOptimizationOptions',opts)
```

```
=====
| Iter | Eval | Objective | Objective | BestSoFar | BestSoFar | BoxConstrain-
|      | result |          | runtime   | (observed) | (estim.)  | t
=====
|  1  | Best |    0.345 |  0.61163 |    0.345 |    0.345 |  0.00474
|  2  | Best |    0.115 |  0.23616 |    0.115 |    0.12678 |  430.31
|  3  | Accept |    0.52 |  0.21638 |    0.115 |    0.1152 |  0.028415
|  4  | Accept |    0.61 |  0.19302 |    0.115 |    0.11504 |  133.94
|  5  | Accept |    0.34 |  0.16922 |    0.115 |    0.11504 |  0.010993
|  6  | Best |    0.085 |   0.1686 |    0.085 |   0.085039 |  885.63
|  7  | Accept |    0.105 |  0.21779 |    0.085 |   0.085428 |  0.3057
|  8  | Accept |    0.21 |  0.16162 |    0.085 |   0.09566 |  0.16044
|  9  | Accept |    0.085 |  0.22575 |    0.085 |   0.08725 |  972.19
| 10  | Accept |    0.1 |  0.19483 |    0.085 |   0.090952 |  990.29
| 11  | Best |    0.08 |  0.17555 |    0.08 |   0.079362 |  2.5195
| 12  | Accept |    0.09 |  0.16849 |    0.08 |   0.08402 |  14.338
| 13  | Accept |    0.1 |  0.19318 |    0.08 |   0.08508 |  0.0022577
| 14  | Accept |    0.11 |   0.1788 |    0.08 |   0.087378 |  0.2115
| 15  | Best |    0.07 |  0.18913 |    0.07 |   0.081507 |  910.2
| 16  | Best |    0.065 |  0.20248 |    0.065 |   0.072457 |  953.22
| 17  | Accept |    0.075 |  0.18661 |    0.065 |   0.072554 |  998.74
| 18  | Accept |    0.295 |  0.22444 |    0.065 |   0.072647 |  996.18
| 19  | Accept |    0.07 |    0.201 |    0.065 |   0.06946 |  985.37
| 20  | Accept |    0.165 |  0.18236 |    0.065 |   0.071622 |  0.065103
=====
| Iter | Eval | Objective | Objective | BestSoFar | BestSoFar | BoxConstrain-
```

	result	runtime	(observed)	(estim.)	t	
21	Accept	0.345	0.25026	0.065	0.071764	971.7
22	Accept	0.61	0.30306	0.065	0.071967	0.0010168
23	Accept	0.345	0.26702	0.065	0.071959	0.0010674
24	Accept	0.35	0.27245	0.065	0.071863	0.0010003
25	Accept	0.24	0.46945	0.065	0.072124	996.55
26	Accept	0.61	0.30876	0.065	0.072068	958.64
27	Accept	0.47	0.25517	0.065	0.07218	993.69
28	Accept	0.3	0.23682	0.065	0.072291	993.15
29	Accept	0.16	0.54548	0.065	0.072104	992.81
30	Accept	0.365	0.25852	0.065	0.072112	0.0010017

---

Optimization completed.

MaxObjectiveEvaluations of 30 reached.

Total function evaluations: 30

Total elapsed time: 62.3692 seconds.

Total objective function evaluation time: 7.464

Best observed feasible point:

BoxConstraint	KernelScale
953.22	0.26253

Observed objective function value = 0.065

Estimated objective function value = 0.072112

Function evaluation time = 0.20248

Best estimated feasible point (according to models):

BoxConstraint	KernelScale
985.37	0.27389

Estimated objective function value = 0.072112

Estimated function evaluation time = 0.20119

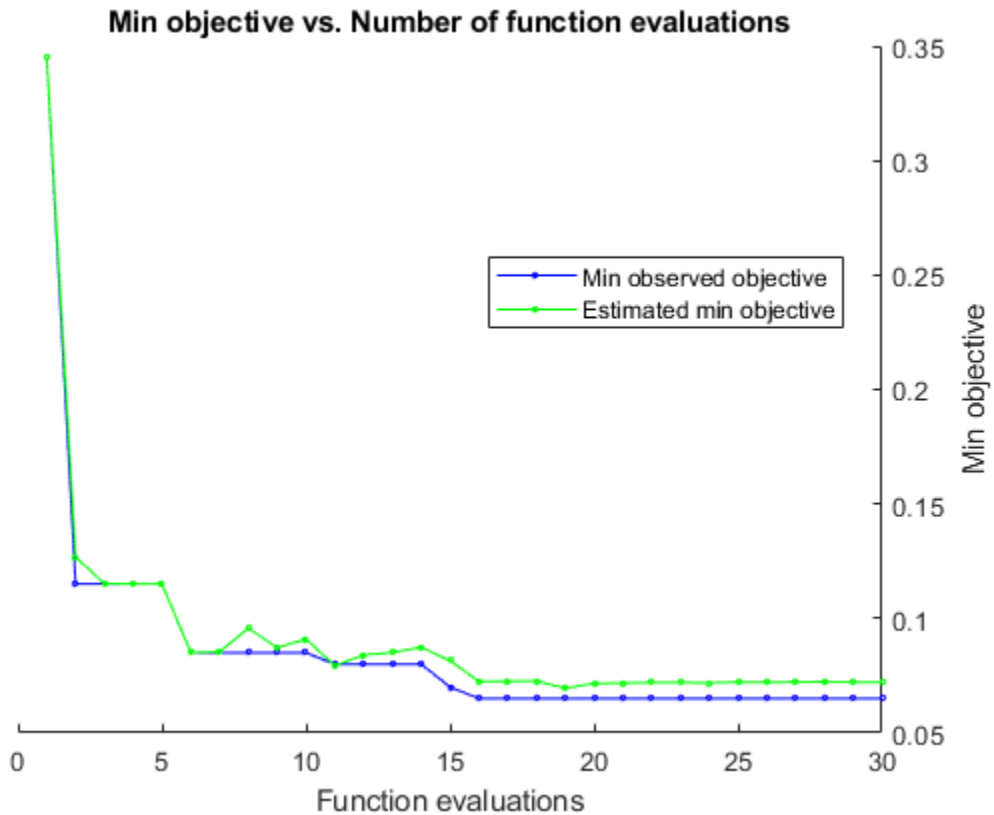
svmmmod =

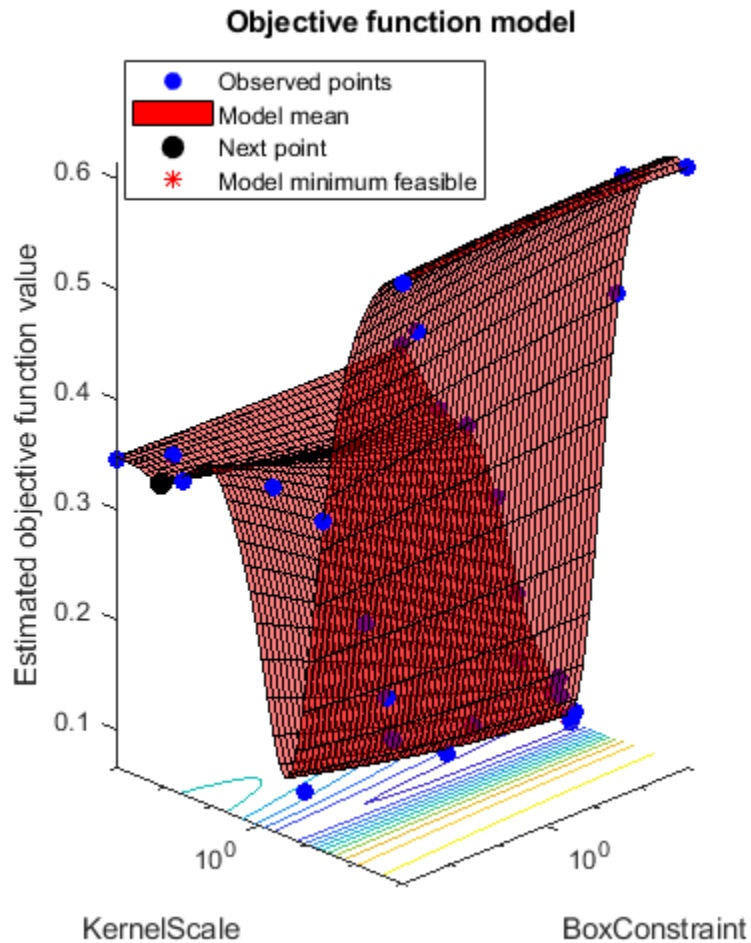
ClassificationSVM

ResponseName: 'Y'

```

CategoricalPredictors: []
ClassNames: [-1 1]
ScoreTransform: 'none'
NumObservations: 200
HyperparameterOptimizationResults: [1x1 BayesianOptimization]
    Alpha: [77x1 double]
    Bias: -0.2352
KernelParameters: [1x1 struct]
    BoxConstraints: [200x1 double]
    ConvergenceInfo: [1x1 struct]
    IsSupportVector: [200x1 logical]
    Solver: 'SMO'
    
```





Find the loss of the optimized model.

```
lossnew = kfoldLoss(fitcsvm(cdata,grp,'CVPartition',c,'KernelFunction','rbf',...
    'BoxConstraint',svmmod.HyperparameterOptimizationResults.XAtMinObjective.BoxConstra
    'KernelScale',svmmod.HyperparameterOptimizationResults.XAtMinObjective.KernelScale)

lossnew =
```

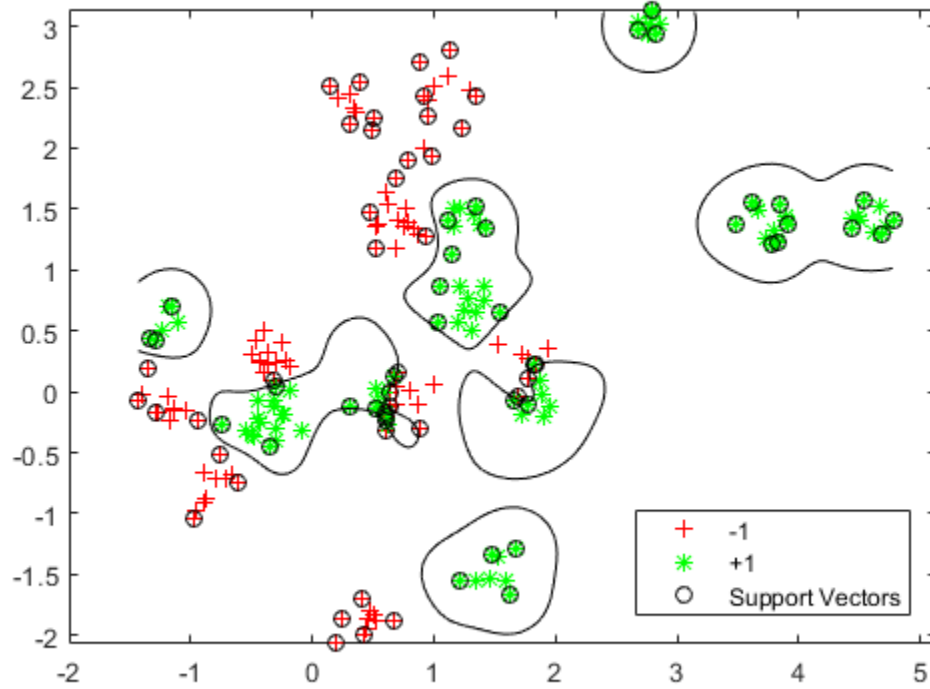


0.0650

This loss is the same as the loss reported in the optimization output under "Observed objective function value".

Visualize the optimized classifier.

```
d = 0.02;
[x1Grid,x2Grid] = meshgrid(min(cdata(:,1)):d:max(cdata(:,1)),...
    min(cdata(:,2)):d:max(cdata(:,2)));
xGrid = [x1Grid(:),x2Grid(:)];
[~,scores] = predict(svmmmod,xGrid);
figure;
h = nan(3,1); % Preallocation
h(1:2) = gscatter(cdata(:,1),cdata(:,2),grp,'rg','+*');
hold on
h(3) = plot(cdata(svmmmod.IsSupportVector,1),...
    cdata(svmmmod.IsSupportVector,2),'ko');
contour(x1Grid,x2Grid,reshape(scores(:,2),size(x1Grid)),[0 0],'k');
legend(h,{'-1','+1','Support Vectors'},'Location','Southeast');
axis equal
hold off
```



## See Also

### Related Examples

- “Optimize a Cross-Validated SVM Classifier Using Bayesian Optimization” on page 10-53

## Optimize a Boosted Regression Ensemble

This example shows how to optimize hyperparameters of a boosted regression ensemble. The optimization minimizes the cross-validation loss of the model.

The problem is to model the efficiency in miles per gallon of an automobile, based on its acceleration, engine displacement, horsepower, and weight. Load the `carsmall` data, which contains these and other predictors.

```
load carsmall
X = [Acceleration Displacement Horsepower Weight];
Y = MPG;
```

Fit a regression ensemble to the data using the `LSBoost` algorithm, and using surrogate splits. Optimize the resulting model by varying the number of learning cycles, the maximum number of surrogate splits, and the learn rate. Furthermore, allow the optimization to repartition the cross-validation between every iteration.

For reproducibility, set the random seed and use the `'expected-improvement-plus'` acquisition function.

```
rng default
Mdl = fitrensemble(X,Y,...
    'Method','LSBoost',...
    'Learner',templateTree('Surrogate','on'),...
    'OptimizeHyperparameters',{'NumLearningCycles','MaxNumSplits','LearnRate'},...
    'HyperparameterOptimizationOptions',struct('Repartition',true,...
    'AcquisitionFunctionName','expected-improvement-plus'))
```

```
=====
| Iter | Eval  | Objective | Objective | BestSoFar | BestSoFar | NumLearningC-
|      | result|           | runtime   | (observed) | (estim.)  | ycles
|=====|=====|=====|=====|=====|=====|=====|
|  1  | Best  | 3.5411 | 20.431 | 3.5411 | 3.5411 | 383
|  2  | Best  | 3.4755 | 1.1136 | 3.4755 | 3.479  | 16
|  3  | Best  | 3.1893 | 1.8908 | 3.1893 | 3.1893 | 33
|  4  | Accept| 6.3077 | 0.73858 | 3.1893 | 3.1898 | 13
|  5  | Accept| 3.4482 | 16.764 | 3.1893 | 3.1897 | 302
|  6  | Accept| 4.2638 | 0.62556 | 3.1893 | 3.1897 | 10
|  7  | Accept| 3.2449 | 0.54219 | 3.1893 | 3.1898 | 10
|  8  | Accept| 3.4495 | 0.62829 | 3.1893 | 3.19  | 14
|  9  | Accept| 5.8544 | 11.795 | 3.1893 | 3.1904 | 308
| 10  | Accept| 3.1985 | 0.46597 | 3.1893 | 3.1876 | 10
```

	11	Accept		3.3339		20.399		3.1893		3.1886		447
	12	Best		2.9764		0.68474		2.9764		3.1412		11
	13	Accept		3.1958		0.59478		2.9764		3.1537		10
	14	Accept		3.2951		19.856		2.9764		3.1458		487
	15	Accept		5.8041		0.57308		2.9764		3.1653		10
	16	Accept		3.4128		24.396		2.9764		3.1677		500
	17	Accept		3.2357		1.3228		2.9764		3.1653		24
	18	Accept		3.2848		6.275		2.9764		3.1605		129
	19	Accept		3.1073		1.0752		2.9764		3.1408		16
	20	Accept		6.422		0.83274		2.9764		3.1415		10
=====												
	Iter	Eval		Objective		Objective		BestSoFar		BestSoFar		NumLearningC-
		result				runtime		(observed)		(estim.)		ycles
=====												
	21	Accept		3.2146		1.214		2.9764		3.157		18
	22	Accept		3.0515		0.79558		2.9764		3.1365		10
	23	Accept		3.3721		35.206		2.9764		3.1357		500
	24	Accept		3.1053		36.448		2.9764		3.136		499
	25	Accept		3.1303		34.267		2.9764		3.1357		499
	26	Accept		3.1956		43.677		2.9764		3.1354		500
	27	Accept		3.2926		46.12		2.9764		3.1366		500
	28	Accept		4.4567		9.4518		2.9764		3.1372		74
	29	Accept		3.4466		18.549		2.9764		3.1383		186
	30	Accept		6.1348		7.8015		2.9764		3.137		68

---

Optimization completed.

MaxObjectiveEvaluations of 30 reached.

Total function evaluations: 30

Total elapsed time: 483.5987 seconds.

Total objective function evaluation time: 364.5336

Best observed feasible point:

NumLearningCycles	LearnRate	MaxNumSplits
11	0.26217	98

Observed objective function value = 2.9764

Estimated objective function value = 3.137

Function evaluation time = 0.68474

Best estimated feasible point (according to models):

NumLearningCycles	LearnRate	MaxNumSplits
-------------------	-----------	--------------

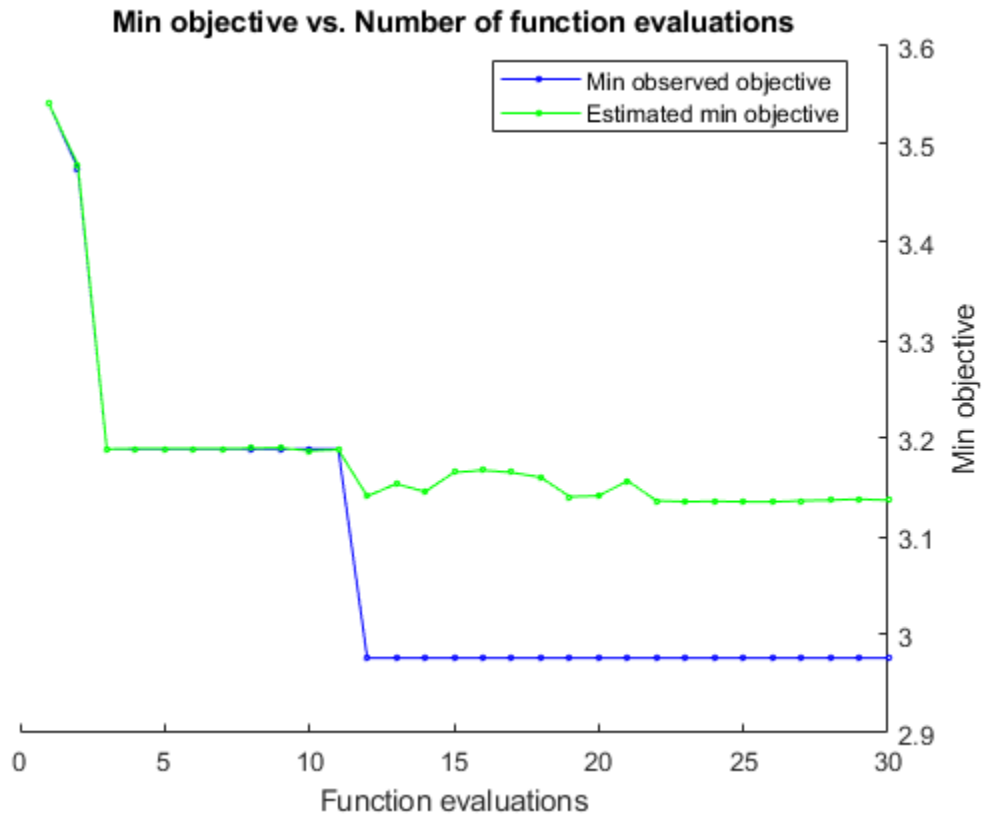
---

```
10                0.29884        66
```

```
Estimated objective function value = 3.137  
Estimated function evaluation time = 0.59873
```

```
Mdl =
```

```
classreg.learning.regr.RegressionEnsemble  
    ResponseName: 'Y'  
    CategoricalPredictors: []  
    ResponseTransform: 'none'  
    NumObservations: 94  
    HyperparameterOptimizationResults: [1x1 BayesianOptimization]  
        NumTrained: 10  
        Method: 'LSBoost'  
        LearnerNames: {'Tree'}  
    ReasonForTermination: 'Terminated normally after completing the request'  
        FitInfo: [10x1 double]  
    FitInfoDescription: {2x1 cell}  
    Regularization: []
```



Compare the loss to that of a boosted, unoptimized model, and to that of the default ensemble.

```
loss = kfoldLoss(crossval(Mdl, 'kfold', 10))
```

```
loss =
```

```
23.3445
```

```
Mdl2 = fitensemble(X, Y, ...
    'Method', 'LSBoost', ...
    'Learner', templateTree('Surrogate', 'on'));
loss2 = kfoldLoss(crossval(Mdl2, 'kfold', 10))
```

```
loss2 =  
    37.0534  
  
Mdl3 = fitrensemble(X,Y);  
loss3 = kfoldLoss(crossval(Mdl3,'kfold',10))  
  
loss3 =  
    38.4890
```

For a different way of optimizing this ensemble, see “Find the Optimal Number of Splits and Trees for an Ensemble” on page 32-2456.





# Parametric Regression Analysis

---

- “Parametric Regression Analysis” on page 11-3
- “What Are Linear Regression Models?” on page 11-8
- “Linear Regression” on page 11-11
- “Linear Regression Workflow” on page 11-43
- “Regression Using Dataset Arrays” on page 11-50
- “Regression Using Tables” on page 11-53
- “Linear Regression with Interaction Effects” on page 11-56
- “Interpret Linear Regression Results” on page 11-64
- “Cook’s Distance” on page 11-71
- “Coefficient Standard Errors and Confidence Intervals” on page 11-75
- “Coefficient of Determination (R-Squared)” on page 11-79
- “Delete-1 Statistics” on page 11-82
- “Durbin-Watson Test” on page 11-93
- “F-statistic and t-statistic” on page 11-95
- “Hat Matrix and Leverage” on page 11-102
- “Residuals” on page 11-106
- “Summary of Output and Diagnostic Statistics” on page 11-115
- “Wilkinson Notation” on page 11-117
- “Stepwise Regression” on page 11-127
- “Robust Regression — Reduce Outlier Effects” on page 11-133
- “Introduction to Ridge Regression” on page 11-136
- “Ridge Regression” on page 11-137
- “Lasso and Elastic Net” on page 11-141
- “Wide Data via Lasso and Parallel Computing” on page 11-144
- “Lasso Regularization” on page 11-149
- “Lasso and Elastic Net with Cross-Validation” on page 11-153

- “Partial Least Squares” on page 11-157
- “Linear Mixed-Effects Models” on page 11-162
- “Prepare Data for Linear Mixed-Effects Models” on page 11-166
- “Relationship Between Formula and Design Matrices” on page 11-172
- “Estimating Parameters in Linear Mixed-Effects Models” on page 11-179
- “Linear Mixed-Effects Model Workflow” on page 11-183
- “Fit Mixed-Effects Spline Regression” on page 11-197

## Parametric Regression Analysis

### In this section...

“What Is Parametric Regression?” on page 11-3

“Choose a Regression Function” on page 11-3

“Update Legacy Code with New Fitting Methods” on page 11-4

### What Is Parametric Regression?

Regression is the process of fitting models to data. The models must have numerical responses. For models with categorical responses, see “Parametric Classification” on page 17-2 or “Supervised Learning Workflow and Algorithms” on page 18-2. The regression process depends on the model. If a model is parametric, regression estimates the parameters from the data. If a model is linear in the parameters, estimation is based on methods from linear algebra that minimize the norm of a residual vector. If a model is nonlinear in the parameters, estimation is based on search methods from optimization that minimize the norm of a residual vector.

### Choose a Regression Function

You have:	You want:	Use this:
Continuous or categorical predictors, continuous response, linear model	Fitted model coefficients	<code>fitlm</code> . See “Linear Regression” on page 11-11.
Continuous or categorical predictors, continuous response, linear model of unknown complexity	Fitted model and fitted coefficients	<code>stepwiselm</code> . See “Stepwise Regression” on page 11-127.
Continuous or categorical predictors, response possibly with restrictions such as nonnegative or integer-valued, generalized linear model	Fitted generalized linear model coefficients	<code>fitglm</code> or <code>stepwiseglm</code> . See “Generalized Linear Models” on page 12-12.
Continuous predictors with a continuous nonlinear response, parametrized nonlinear model	Fitted nonlinear model coefficients	<code>fitnlm</code> . See “Nonlinear Regression” on page 13-2.

You have:	You want:	Use this:
Continuous predictors, continuous response, linear model	Set of models from ridge, lasso, or elastic net regression	lasso or ridge See “Lasso and Elastic Net” on page 11-141 or “Introduction to Ridge Regression” on page 11-136.
Correlated continuous predictors, continuous response, linear model	Fitted model and fitted coefficients	plsregress See “Partial Least Squares” on page 11-157.
Continuous or categorical predictors, continuous response, unknown model	Nonparametric model	fitrtree or fitrensemble See
Categorical predictors only	ANOVA	anova, anova1, anova2, anovan
Continuous predictors, multivariable response, linear model	Fitted multivariate regression model coefficients	mvregress
Continuous predictors, continuous response, mixed-effects model	Fitted mixed-effects model coefficients	nlmefit or nlmefitsa See “Mixed-Effects Models” on page 13-22.

## Update Legacy Code with New Fitting Methods

There are several Statistics and Machine Learning Toolbox functions for performing regression. The following sections describe how to replace calls to older functions to new versions:

- “regress into fitlm” on page 11-5
- “regstats into fitlm” on page 11-5
- “robustfit into fitlm” on page 11-5
- “stepwisefit into stepwiselm” on page 11-6
- “glmfit into fitglm” on page 11-6
- “nlinfit into fitnlm” on page 11-6

**regress into fitlm****Previous Syntax**

```
[b,bint,r,rint,stats] = regress(y,X)
```

where X contains a column of ones.

**Current Syntax**

```
mdl = fitlm(X,y)
```

where you do not add a column of ones to X.

Equivalent values of the previous outputs:

- `b` — `mdl.Coefficients.Estimate`
- `bint` — `coefCI(mdl)`
- `r` — `mdl.Residuals.Raw`
- `rint` — There is no exact equivalent. Try examining `mdl.Residuals.Studentized` to find outliers.
- `stats` — `mdl` contains various properties that replace components of `stats`.

**regstats into fitlm****Previous Syntax**

```
stats = regstats(y,X,model,whichstats)
```

**Current Syntax**

```
mdl = fitlm(X,y,model)
```

Obtain statistics from the properties and methods of `mdl`. For example, see the `mdl.Diagnostics` and `mdl.Residuals` properties.

**robustfit into fitlm****Previous Syntax**

```
[b,stats] = robustfit(X,y,wfun,tune,const)
```

**Current Syntax**

```
mdl = fitlm(X,y,'robust','on') % bisquare
```

Or to use the *wfun* weight and the *tune* tuning parameter:

```
opt.RobustWgtFun = 'wfun';  
opt.Tune = tune; % optional  
mdl = fitlm(X,y,'robust',opt)
```

Obtain statistics from the properties and methods of `mdl`. For example, see the `mdl.Diagnostics` and `mdl.Residuals` properties.

### **stepwisefit into stepwiselm**

#### **Previous Syntax**

```
[b,se,pval,inmodel,stats,nextstep,history] = stepwisefit(X,y,Name,Value)
```

#### **Current Syntax**

```
mdl = stepwiselm(ds,modelspec,Name,Value)
```

or

```
mdl = stepwiselm(X,y,modelspec,Name,Value)
```

Obtain statistics from the properties and methods of `mdl`. For example, see the `mdl.Diagnostics` and `mdl.Residuals` properties.

### **glmfit into fitglm**

#### **Previous Syntax**

```
[b,dev,stats] = glmfit(X,y,distr,param1,va11,...)
```

#### **Current Syntax**

```
mdl = fitglm(X,y,distr,...)
```

Obtain statistics from the properties and methods of `mdl`. For example, the deviance is `mdl.Deviance`, and to compare `mdl` against a constant model, use `devianceTest(mdl)`.

### **nlinfit into fitnlm**

#### **Previous Syntax**

```
[beta,r,J,COVB,mse] = nlinfit(X,y,fun,beta0,options)
```

**Current Syntax**

```
mdl = fitnlm(X,y,fun,beta0,'Options',options)
```

Equivalent values of the previous outputs:

- `beta` — `mdl.Coefficients.Estimate`
- `r` — `mdl.Residuals.Raw`
- `covb` — `mdl.CoefficientCovariance`
- `mse` — `mdl.mse`

`mdl` does not provide the Jacobian ( $J$ ) output. The primary purpose of  $J$  was to pass it into `nlparci` or `nlpredci` to obtain confidence intervals for the estimated coefficients (parameters) or predictions. Obtain those confidence intervals as:

```
parci = coefCI(mdl)  
[pred,predci] = predict(mdl)
```

## What Are Linear Regression Models?

Regression models describe the relationship between a *dependent variable*,  $y$ , and *independent variable* or variables,  $X$ . The dependent variable is also called the *response variable*. Independent variables are also called *explanatory* or *predictor variables*. Continuous predictor variables might be called *covariates*, whereas categorical predictor variables might be also referred to as *factors*. The matrix,  $X$ , of observations on predictor variables is usually called the *design matrix*.

A multiple linear regression model is

$$y_i = \beta_0 + \beta_1 X_{i1} + \beta_2 X_{i2} + \cdots + \beta_p X_{ip} + \varepsilon_i, \quad i = 1, \dots, n,$$

where

- $y_i$  is the  $i$ th response.
- $\beta_k$  is the  $k$ th coefficient, where  $\beta_0$  is the constant term in the model. Sometimes, design matrices might include information about the constant term. However, `fitlm` or `stepwiselm` by default includes a constant term in the model, so you must not enter a column of 1s into your design matrix  $X$ .
- $X_{ij}$  is the  $i$ th observation on the  $j$ th predictor variable,  $j = 1, \dots, p$ .
- $\varepsilon_i$  is the  $i$ th noise term, that is, random error.

In general, a linear regression model can be a model of the form

$$y_i = \beta_0 + \sum_{k=1}^K \beta_k f_k(X_{i1}, X_{i2}, \dots, X_{ip}) + \varepsilon_i, \quad i = 1, \dots, n,$$

where  $f(\cdot)$  is a scalar-valued function of the independent variables,  $X_{ij}$ s. The functions,  $f(X)$ , might be in any form including nonlinear functions or polynomials. The linearity, in the linear regression models, refers to the linearity of the coefficients  $\beta_k$ . That is, the response variable,  $y$ , is a linear function of the coefficients,  $\beta_k$ .

Some examples of linear models are:

$$y_i = \beta_0 + \beta_1 X_{1i} + \beta_2 X_{2i} + \beta_3 X_{3i} + \varepsilon_i$$

$$y_i = \beta_0 + \beta_1 X_{1i} + \beta_2 X_{2i} + \beta_3 X_{1i}^3 + \beta_4 X_{2i}^2 + \varepsilon_i$$

$$y_i = \beta_0 + \beta_1 X_{1i} + \beta_2 X_{2i} + \beta_3 X_{1i} X_{2i} + \beta_4 \log X_{3i} + \varepsilon_i$$

The following, however, are not linear models since they are not linear in the unknown coefficients,  $\beta_k$ .



$$\log y_i = \beta_0 + \beta_1 X_{1i} + \beta_2 X_{2i} + \varepsilon_i$$

$$y_i = \beta_0 + \beta_1 X_{1i} + \frac{1}{\beta_2 X_{2i}} + e^{\beta_3 X_{1i} X_{2i}} + \varepsilon_i$$

The usual assumptions for linear regression models are:

- The noise terms,  $\varepsilon_i$ , are uncorrelated.
- The noise terms,  $\varepsilon_i$ , have independent and identical normal distributions with mean zero and constant variance,  $\sigma^2$ . Thus

$$\begin{aligned} E(y_i) &= E\left(\sum_{k=0}^K \beta_k f_k(X_{i1}, X_{i2}, \dots, X_{ip}) + \varepsilon_i\right) \\ &= \sum_{k=0}^K \beta_k f_k(X_{i1}, X_{i2}, \dots, X_{ip}) + E(\varepsilon_i) \\ &= \sum_{k=0}^K \beta_k f_k(X_{i1}, X_{i2}, \dots, X_{ip}) \end{aligned}$$

and

$$V(y_i) = V\left(\sum_{k=0}^K \beta_k f_k(X_{i1}, X_{i2}, \dots, X_{ip}) + \varepsilon_i\right) = V(\varepsilon_i) = \sigma^2$$

So the variance of  $y_i$  is the same for all levels of  $X_{ij}$ .

- The responses  $y_i$  are uncorrelated.

The fitted linear function is

$$\hat{y}_i = \sum_{k=0}^K b_k f_k(X_{i1}, X_{i2}, \dots, X_{ip}), \quad i = 1, \dots, n,$$

where  $\hat{y}_i$  is the estimated response and  $b_k$ s are the fitted coefficients. The coefficients are estimated so as to minimize the mean squared difference between the prediction vector

$\hat{\mathbf{y}}$  and the true response vector  $\mathbf{y}$ , that is  $\hat{\mathbf{y}} - \mathbf{y}$ . This method is called the *method of least squares*. Under the assumptions on the noise terms, these coefficients also maximize the likelihood of the prediction vector.

In a linear regression model of the form  $y = \beta_1 X_1 + \beta_2 X_2 + \dots + \beta_p X_p$ , the coefficient  $\beta_k$  expresses the impact of a one-unit change in predictor variable,  $X_j$ , on the mean of the response,  $E(y)$  provided that all other variables are held constant. The sign of the coefficient gives the direction of the effect. For example, if the linear model is  $E(y) = 1.8 - 2.35X_1 + X_2$ , then  $-2.35$  indicates a 2.35 unit decrease in the mean response with a one-unit increase in  $X_1$ , given  $X_2$  is held constant. If the model is  $E(y) = 1.1 + 1.5X_1^2 + X_2$ , the coefficient of  $X_1^2$  indicates a 1.5 unit increase in the mean of  $Y$  with a one-unit increase in  $X_1^2$  given all else held constant. However, in the case of  $E(y) = 1.1 + 2.1X_1 + 1.5X_1^2$ , it is difficult to interpret the coefficients similarly, since it is not possible to hold  $X_1$  constant when  $X_1^2$  changes or vice versa.

## References

- [1] Neter, J., M. H. Kutner, C. J. Nachtsheim, and W. Wasserman. *Applied Linear Statistical Models*. IRWIN, The McGraw-Hill Companies, Inc., 1996.
- [2] Seber, G. A. F. *Linear Regression Analysis*. Wiley Series in Probability and Mathematical Statistics. John Wiley and Sons, Inc., 1977.

## See Also

`LinearModel` | `fitlm` | `stepwiselm`

## Related Examples

- “Interpret Linear Regression Results” on page 11-64
- “Regression Using Dataset Arrays” on page 11-50
- “Linear Regression with Interaction Effects” on page 11-56
- “Regression with Categorical Covariates” on page 2-65
- “Linear Regression Workflow” on page 11-43

# Linear Regression

## In this section...

“Prepare Data” on page 11-11  
“Choose a Fitting Method” on page 11-13  
“Choose a Model or Range of Models” on page 11-14  
“Fit Model to Data” on page 11-19  
“Examine Quality and Adjust the Fitted Model” on page 11-20  
“Predict or Simulate Responses to New Data” on page 11-38  
“Share Fitted Models” on page 11-41

## Prepare Data

To begin fitting a regression, put your data into a form that fitting functions expect. All regression techniques begin with input data in an array  $X$  and response data in a separate vector  $y$ , or input data in a table or dataset array `tbl` and response data as a column in `tbl`. Each row of the input data represents one observation. Each column represents one predictor (variable).

For a table or dataset array `tbl`, indicate the response variable with the 'ResponseVar' name-value pair:

```
mdl = fitlm(tbl, 'ResponseVar', 'BloodPressure');  
% or  
mdl = fitglm(tbl, 'ResponseVar', 'BloodPressure');
```

The response variable is the last column by default.

You can use numeric categorical predictors. A categorical predictor is one that takes values from a fixed set of possibilities.

- For a numeric array  $X$ , indicate the categorical predictors using the 'Categorical' name-value pair. For example, to indicate that predictors 2 and 3 out of six are categorical:

```
mdl = fitlm(X, y, 'Categorical', [2,3]);  
% or  
mdl = fitglm(X, y, 'Categorical', [2,3]);
```

```
% or equivalently  
mdl = fitlm(X,y,'Categorical',logical([0 1 1 0 0 0]));
```

- For a table or dataset array `tbl`, fitting functions assume that these data types are categorical:
  - Logical
  - Categorical (nominal or ordinal)
  - Character array

If you want to indicate that a numeric predictor is categorical, use the 'Categorical' name-value pair.

Represent missing numeric data as `NaN`. To represent missing data for other data types, see “Missing Group Values” on page 2-61.

### Dataset Array for Input and Response Data

To create a dataset array from an Excel spreadsheet:

```
ds = dataset('XLSFile','hospital.xls',...  
            'ReadObsNames',true);
```

To create a dataset array from workspace variables:

```
load carsmall  
ds = dataset(MPG,Weight);  
ds.Year = ordinal(Model_Year);
```

### Table for Input and Response Data

To create a table from an Excel spreadsheet:

```
tbl = readtable('hospital.xls',...  
               'ReadRowNames',true);
```

To create a table from workspace variables:

```
load carsmall  
tbl = table(MPG,Weight);  
tbl.Year = ordinal(Model_Year);
```

### Numeric Matrix for Input Data, Numeric Vector for Response

For example, to create numeric arrays from workspace variables:

```
load carsmall
X = [Weight Horsepower Cylinders Model_Year];
y = MPG;
```

To create numeric arrays from an Excel spreadsheet:

```
[X, Xnames] = xlsread('hospital.xls');
y = X(:,4); % response y is systolic pressure
X(:,4) = []; % remove y from the X matrix
```

Notice that the nonnumeric entries, such as `sex`, do not appear in `X`.

## Choose a Fitting Method

There are three ways to fit a model to data:

- “Least-Squares Fit” on page 11-13
- “Robust Fit” on page 11-13
- “Stepwise Fit” on page 11-13

### Least-Squares Fit

Use `fitlm` to construct a least-squares fit of a model to the data. This method is best when you are reasonably certain of the model’s form, and mainly need to find its parameters. This method is also useful when you want to explore a few models. The method requires you to examine the data manually to discard outliers, though there are techniques to help (see “Residuals — Model Quality for Training Data” on page 11-24).

### Robust Fit

Use `fitlm` with the `RobustOpts` name-value pair to create a model that is little affected by outliers. Robust fitting saves you the trouble of manually discarding outliers. However, `step` does not work with robust fitting. This means that when you use robust fitting, you cannot search stepwise for a good model.

### Stepwise Fit

Use `stepwiselm` to find a model, and fit parameters to the model. `stepwiselm` starts from one model, such as a constant, and adds or subtracts terms one at a time, choosing an optimal term each time in a greedy fashion, until it cannot improve further. Use stepwise fitting to find a good model, which is one that has only relevant terms.

The result depends on the starting model. Usually, starting with a constant model leads to a small model. Starting with more terms can lead to a more complex model, but one that has lower mean squared error. See “Compare large and small stepwise models” on page 11-127.

You cannot use robust options along with stepwise fitting. So after a stepwise fit, examine your model for outliers (see “Residuals — Model Quality for Training Data” on page 11-24).

## Choose a Model or Range of Models

There are several ways of specifying a model for linear regression. Use whichever you find most convenient.

- “Brief Name” on page 11-14
- “Terms Matrix” on page 11-15
- “Formula” on page 11-18

For `fitlm`, the model specification you give is the model that is fit. If you do not give a model specification, the default is `'linear'`.

For `stepwiselm`, the model specification you give is the starting model, which the stepwise procedure tries to improve. If you do not give a model specification, the default starting model is `'constant'`, and the default upper bounding model is `'interactions'`. Change the upper bounding model using the `Upper` name-value pair.

---

**Note** There are other ways of selecting models, such as using `lasso`, `lassoglm`, `sequentialfs`, or `plsregress`.

---

### Brief Name

Name	Model Type
<code>'constant'</code>	Model contains only a constant (intercept) term.
<code>'linear'</code>	Model contains an intercept and linear terms for each predictor.
<code>'interactions'</code>	Model contains an intercept, linear terms, and all products of pairs of distinct predictors (no squared terms).

Name	Model Type
'purequadratic'	Model contains an intercept, linear terms, and squared terms.
'quadratic'	Model contains an intercept, linear terms, interactions, and squared terms.
'polyijk'	Model is a polynomial with all terms up to degree $i$ in the first predictor, degree $j$ in the second predictor, etc. Use numerals 0 through 9. For example, 'poly2111' has a constant plus all linear and product terms, and also contains terms with predictor 1 squared.

For example, to specify an interaction model using `fitlm` with matrix predictors:

```
mdl = fitlm(X,y,'interactions');
```

To specify a model using `stepwiselm` and a table or dataset array `tbl` of predictors, suppose you want to start from a constant and have a linear model upper bound. Assume the response variable in `tbl` is in the third column.

```
mdl2 = stepwiselm(tbl,'constant',...
    'Upper','linear','ResponseVar',3);
```

### Terms Matrix

A terms matrix is a  $t$ -by- $(p + 1)$  matrix specifying terms in a model, where  $t$  is the number of terms,  $p$  is the number of predictor variables, and plus one is for the response variable.

The value of  $T(i, j)$  is the exponent of variable  $j$  in term  $i$ . Suppose there are three predictor variables A, B, and C:

```
[0 0 0 0] % Constant term or intercept
[0 1 0 0] % B; equivalently, A^0 * B^1 * C^0
[1 0 1 0] % A*C
[2 0 0 0] % A^2
[0 1 2 0] % B*(C^2)
```

The 0 at the end of each term represents the response variable. In general,

- If you have the variables in a table or dataset array, then 0 must represent the response variable depending on the position of the response variable. The following example illustrates this.

Load the sample data and define the dataset array.

```
load hospital
dsa = dataset(hospital.Sex,hospital.BloodPressure(:,1),hospital.Age,...
hospital.Smoker, 'VarNames',{'Sex','BloodPressure','Age','Smoker'});
```

Represent the linear model 'BloodPressure ~ 1 + Sex + Age + Smoker' in a terms matrix. The response variable is in the second column of the dataset array, so there must be a column of 0s for the response variable in the second column of the terms matrix.

```
T = [0 0 0 0;1 0 0 0;0 0 1 0;0 0 0 1]
```

```
T =
```

```
    0    0    0    0
    1    0    0    0
    0    0    1    0
    0    0    0    1
```

Redefine the dataset array.

```
dsa = dataset(hospital.BloodPressure(:,1),hospital.Sex,hospital.Age,...
hospital.Smoker, 'VarNames',{'BloodPressure','Sex','Age','Smoker'});
```

Now, the response variable is the first term in the dataset array. Specify the same linear model, 'BloodPressure ~ 1 + Sex + Age + Smoker', using a terms matrix.

```
T = [0 0 0 0;0 1 0 0;0 0 1 0;0 0 0 1]
```

```
T =
```

```
    0    0    0    0
    0    1    0    0
    0    0    1    0
    0    0    0    1
```

- If you have the predictor and response variables in a matrix and column vector, then you must include 0 for the response variable at the end of each term. The following example illustrates this.



Load the sample data and define the matrix of predictors.

```
load carsmall
X = [Acceleration,Weight];
```

Specify the model 'MPG ~ Acceleration + Weight + Acceleration:Weight + Weight^2' using a term matrix and fit the model to the data. This model includes the main effect and two-way interaction terms for the variables, Acceleration and Weight, and a second-order term for the variable, Weight.

```
T = [0 0 0;1 0 0;0 1 0;1 1 0;0 2 0]
```

```
T =
```

```

0     0     0
1     0     0
0     1     0
1     1     0
0     2     0
```

Fit a linear model.

```
mdl = fitlm(X,MPG,T)
```

```
mdl =
```

```
Linear regression model:
```

```
  y ~ 1 + x1*x2 + x2^2
```

```
Estimated Coefficients:
```

	Estimate	SE	tStat	pValue
(Intercept)	48.906	12.589	3.8847	0.00019665
x1	0.54418	0.57125	0.95261	0.34337
x2	-0.012781	0.0060312	-2.1192	0.036857
x1:x2	-0.00010892	0.00017925	-0.6076	0.545
x2^2	9.7518e-07	7.5389e-07	1.2935	0.19917

```
Number of observations: 94, Error degrees of freedom: 89
```

```
Root Mean Squared Error: 4.1
```

```
R-squared: 0.751, Adjusted R-Squared 0.739
```

```
F-statistic vs. constant model: 67, p-value = 4.99e-26
```

Only the intercept and x2 term, which correspond to the Weight variable, are significant at the 5% significance level.

Now, perform a stepwise regression with a constant model as the starting model and a linear model with interactions as the upper model.

```
T = [0 0 0;1 0 0;0 1 0;1 1 0];
mdl = stepwiselm(X,MPG,[0 0 0], 'upper', T)

1. Adding x2, FStat = 259.3087, pValue = 1.643351e-28

mdl =

Linear regression model:
  y ~ 1 + x2

Estimated Coefficients:
              Estimate      SE      tStat      pValue
(Intercept)    49.238      1.6411    30.002  2.7015e-49
x2             -0.0086119  0.0005348 -16.103  1.6434e-28

Number of observations: 94, Error degrees of freedom: 92
Root Mean Squared Error: 4.13
R-squared: 0.738, Adjusted R-Squared 0.735
F-statistic vs. constant model: 259, p-value = 1.64e-28
```

The results of the stepwise regression are consistent with the results of `fitlm` in the previous step.

## Formula

A formula for a model specification is a character vector of the form

```
'Y ~ terms',
```

- *Y* is the response name.
- *terms* contains
  - Variable names
  - + to include the next variable
  - - to exclude the next variable
  - : to define an interaction, a product of terms
  - \* to define an interaction and all lower-order terms
  - ^ to raise the predictor to a power, exactly as in \* repeated, so ^ includes lower order terms as well

- `()` to group terms

---

**Tip** Formulas include a constant (intercept) term by default. To exclude a constant term from the model, include `-1` in the formula.

---

Examples:

`'Y ~ A + B + C'` is a three-variable linear model with intercept.

`'Y ~ A + B + C - 1'` is a three-variable linear model without intercept.

`'Y ~ A + B + C + B^2'` is a three-variable model with intercept and a  $B^2$  term.

`'Y ~ A + B^2 + C'` is the same as the previous example, since  $B^2$  includes a  $B$  term.

`'Y ~ A + B + C + A:B'` includes an  $A*B$  term.

`'Y ~ A*B + C'` is the same as the previous example, since  $A*B = A + B + A:B$ .

`'Y ~ A*B*C - A:B:C'` has all interactions among  $A$ ,  $B$ , and  $C$ , except the three-way interaction.

`'Y ~ A*(B + C + D)'` has all linear terms, plus products of  $A$  with each of the other variables.

For example, to specify an interaction model using `fitlm` with matrix predictors:

```
mdl = fitlm(X,y,'y ~ x1*x2*x3 - x1:x2:x3');
```

To specify a model using `stepwiselm` and a table or dataset array `tbl` of predictors, suppose you want to start from a constant and have a linear model upper bound. Assume the response variable in `tbl` is named `'y'`, and the predictor variables are named `'x1'`, `'x2'`, and `'x3'`.

```
mdl2 = stepwiselm(tbl,'y ~ 1','Upper','y ~ x1 + x2 + x3');
```

## Fit Model to Data

The most common optional arguments for fitting:

- For robust regression in `fitlm`, set the `'RobustOpts'` name-value pair to `'on'`.
- Specify an appropriate upper bound model in `stepwiselm`, such as set `'Upper'` to `'linear'`.
- Indicate which variables are categorical using the `'CategoricalVars'` name-value pair. Provide a vector with column numbers, such as `[1 6]` to specify that predictors

1 and 6 are categorical. Alternatively, give a logical vector the same length as the data columns, with a 1 entry indicating that variable is categorical. If there are seven predictors, and predictors 1 and 6 are categorical, specify `logical([1,0,0,0,0,1,0])`.

- For a table or dataset array, specify the response variable using the 'ResponseVar' name-value pair. The default is the last column in the array.

For example,

```
mdl = fitlm(X,y,'linear',...  
    'RobustOpts','on','CategoricalVars',3);  
mdl2 = stepwiselm(tbl,'constant',...  
    'ResponseVar','MPG','Upper','quadratic');
```

## Examine Quality and Adjust the Fitted Model

After fitting a model, examine the result and make adjustments.

- “Model Display” on page 11-20
- “ANOVA” on page 11-21
- “Diagnostic Plots” on page 11-22
- “Residuals — Model Quality for Training Data” on page 11-24
- “Plots to Understand Predictor Effects” on page 11-29
- “Plots to Understand Terms Effects” on page 11-34
- “Change Models” on page 11-36

### Model Display

A linear regression model shows several diagnostics when you enter its name or enter `disp(mdl)`. This display gives some of the basic information to check whether the fitted model represents the data adequately.

For example, fit a linear model to data constructed with two out of five predictors not present and with no intercept term:

```
X = randn(100,5);  
y = X*[1;0;3;0;-1]+randn(100,1);  
mdl = fitlm(X,y)
```

```
mdl =
```

```
Linear regression model:
```

```
y ~ 1 + x1 + x2 + x3 + x4 + x5
```

```
Estimated Coefficients:
```

	Estimate	SE	tStat	pValue
(Intercept)	0.038164	0.099458	0.38372	0.70205
x1	0.92794	0.087307	10.628	8.5494e-18
x2	-0.075593	0.10044	-0.75264	0.45355
x3	2.8965	0.099879	29	1.1117e-48
x4	0.045311	0.10832	0.41831	0.67667
x5	-0.99708	0.11799	-8.4504	3.593e-13

```
Number of observations: 100, Error degrees of freedom: 94
```

```
Root Mean Squared Error: 0.972
```

```
R-squared: 0.93, Adjusted R-Squared 0.926
```

```
F-statistic vs. constant model: 248, p-value = 1.5e-52
```

Notice that:

- The display contains the estimated values of each coefficient in the Estimate column. These values are reasonably near the true values  $[0; 1; 0; 3; 0; -1]$ .
- There is a standard error column for the coefficient estimates.
- The reported pValue (which are derived from the  $t$  statistics under the assumption of normal errors) for predictors 1, 3, and 5 are extremely small. These are the three predictors that were used to create the response data  $y$ .
- The pValue for (Intercept),  $x_2$  and  $x_4$  are much larger than 0.01. These three predictors were not used to create the response data  $y$ .
- The display contains  $R^2$ , adjusted  $R^2$ , and  $F$  statistics.

## ANOVA

To examine the quality of the fitted model, consult an ANOVA table. For example, use `anova` on a linear model with five predictors:

```
X = randn(100,5);
y = X*[1;0;3;0;-1]+randn(100,1);
mdl = fitlm(X,y);
tbl = anova(mdl)
```

```
tbl =
```

	SumSq	DF	MeanSq	F	pValue
--	-------	----	--------	---	--------

x1	106.62	1	106.62	112.96	8.5494e-18
x2	0.53464	1	0.53464	0.56646	0.45355
x3	793.74	1	793.74	840.98	1.1117e-48
x4	0.16515	1	0.16515	0.17498	0.67667
x5	67.398	1	67.398	71.41	3.593e-13
Error	88.719	94	0.94382		

This table gives somewhat different results than the default display (see “Model Display” on page 11-20). The table clearly shows that the effects of `x2` and `x4` are not significant. Depending on your goals, consider removing `x2` and `x4` from the model.

### Diagnostic Plots

Diagnostic plots help you identify outliers, and see other problems in your model or fit. For example, load the `carsmall` data, and make a model of MPG as a function of Cylinders (nominal) and Weight:

```
load carsmall
tbl = table(Weight,MPG,Cylinders);
tbl.Cylinders = ordinal(tbl.Cylinders);
mdl = fitlm(tbl,'MPG ~ Cylinders*Weight + Weight^2');
```

Make a leverage plot of the data and model.

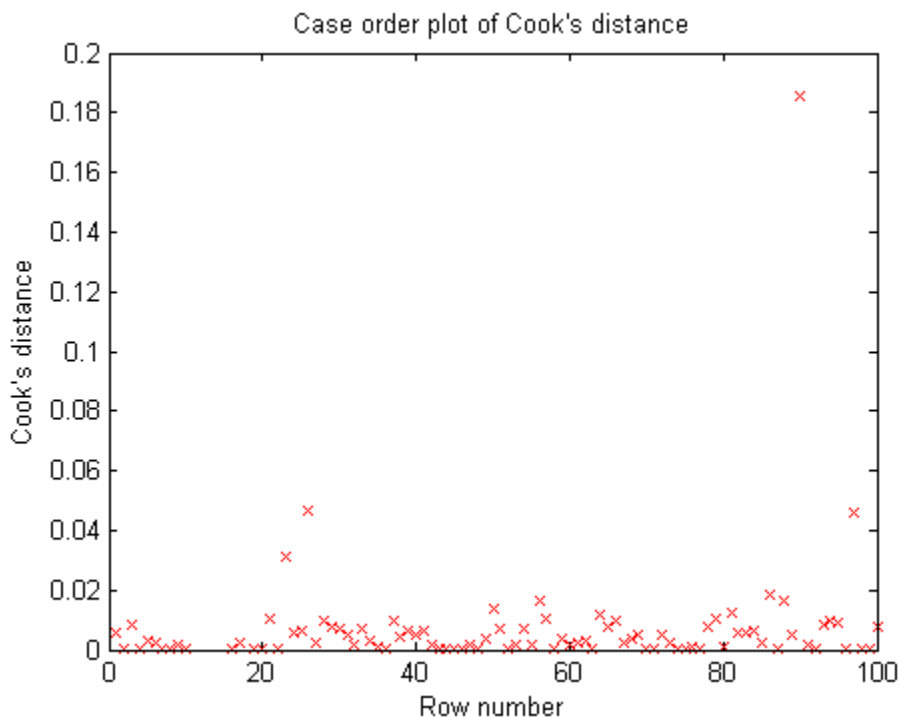
```
plotDiagnostics(mdl)
```



There are a few points with high leverage. But this plot does not reveal whether the high-leverage points are outliers.

Look for points with large Cook's distance.

```
plotDiagnostics mdl, 'cookd')
```



There is one point with large Cook's distance. Identify it and remove it from the model. You can use the Data Cursor to click the outlier and identify it, or identify it programmatically:

```
[~,larg] = max mdl.Diagnostics.CooksDistance);  
mdl2 = fitlm(tbl,'MPG ~ Cylinders*Weight + Weight^2',...  
            'Exclude',larg);
```

### Residuals — Model Quality for Training Data

There are several residual plots to help you discover errors, outliers, or correlations in the model or data. The simplest residual plots are the default histogram plot, which shows the range of the residuals and their frequencies, and the probability plot, which shows how the distribution of the residuals compares to a normal distribution with matched variance.

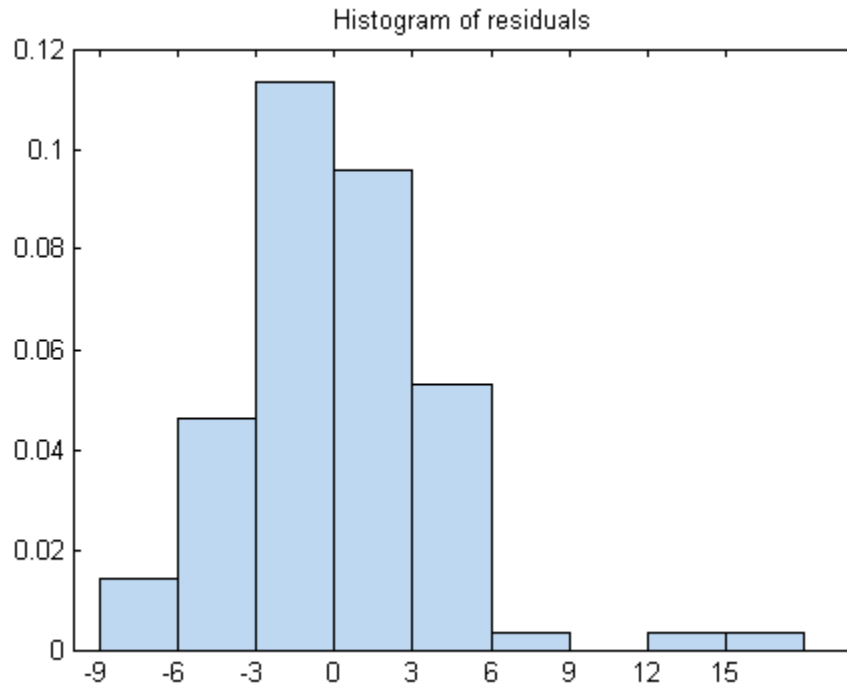


Load the `carsmall` data, and make a model of MPG as a function of Cylinders (nominal) and Weight:

```
load carsmall
tbl = table(Weight,MPG,Cylinders);
tbl.Cylinders = ordinal(tbl.Cylinders);
mdl = fitlm(tbl,'MPG ~ Cylinders*Weight + Weight^2');
```

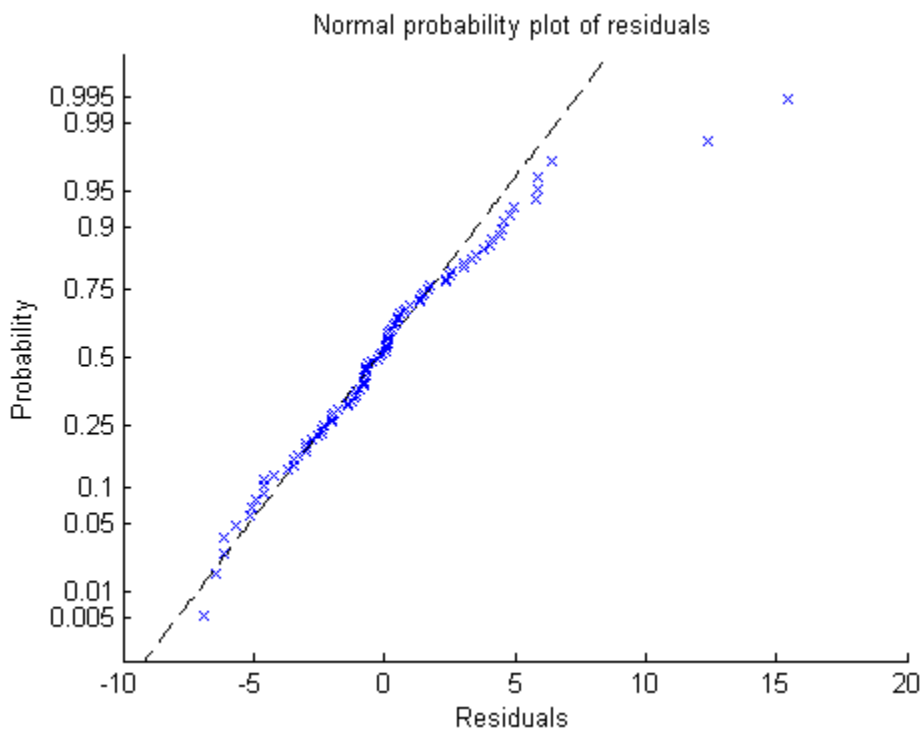
Examine the residuals:

```
plotResiduals(mdl)
```



The observations above 12 are potential outliers.

```
plotResiduals(mdl,'probability')
```



The two potential outliers appear on this plot as well. Otherwise, the probability plot seems reasonably straight, meaning a reasonable fit to normally distributed residuals.

You can identify the two outliers and remove them from the data:

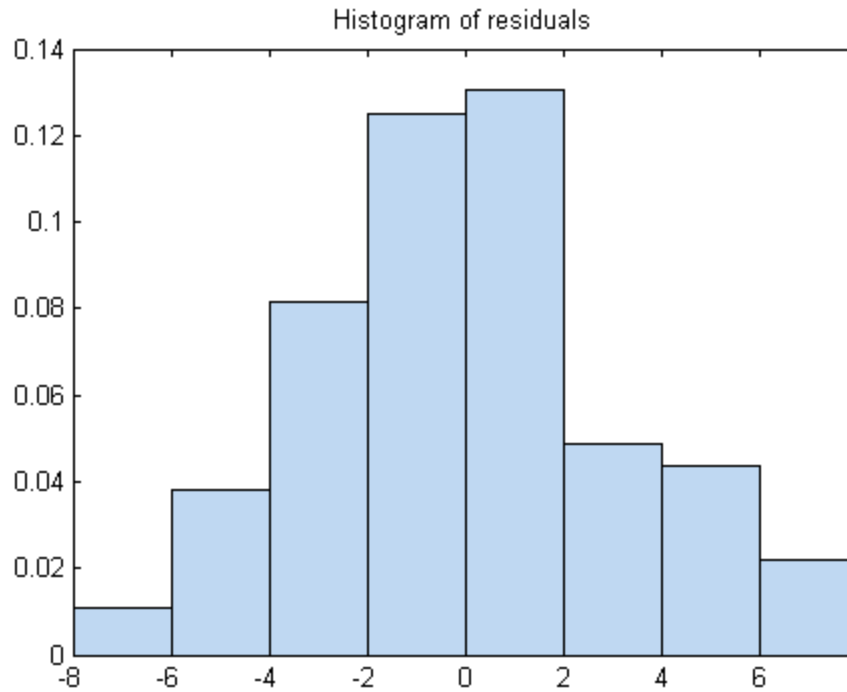
```
out1 = find mdl.Residuals.Raw > 12)
out1 =
    90
    97
```

To remove the outliers, use the `Exclude` name-value pair:

```
mdl2 = fitlm(tbl, 'MPG ~ Cylinders*Weight + Weight^2', ...
    'Exclude', out1);
```

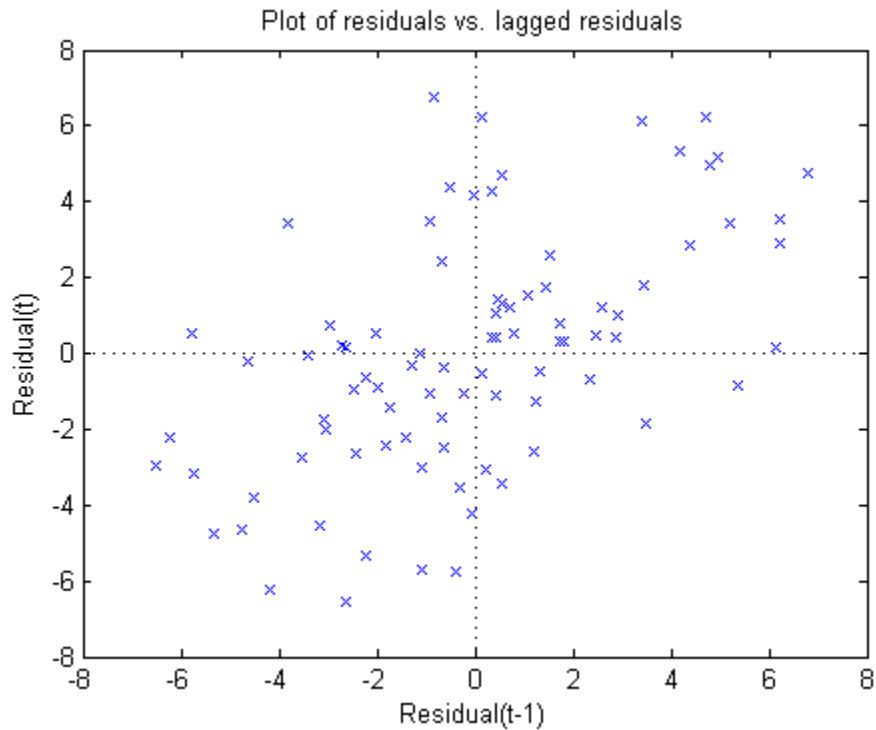
Examine a residuals plot of `mdl2`:

```
plotResiduals (mdl2)
```



The new residuals plot looks fairly symmetric, without obvious problems. However, there might be some serial correlation among the residuals. Create a new plot to see if such an effect exists.

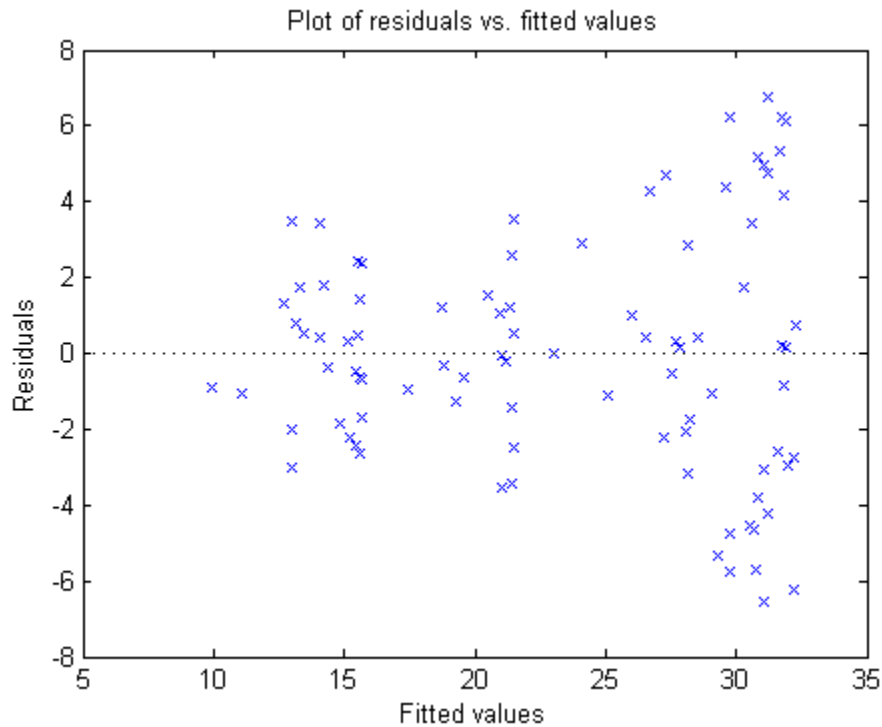
```
plotResiduals (mdl2, 'lagged')
```



The scatter plot shows many more crosses in the upper-right and lower-left quadrants than in the other two quadrants, indicating positive serial correlation among the residuals.

Another potential issue is when residuals are large for large observations. See if the current model has this issue.

```
plotResiduals(mdl2, 'fitted')
```



There is some tendency for larger fitted values to have larger residuals. Perhaps the model errors are proportional to the measured values.

### Plots to Understand Predictor Effects

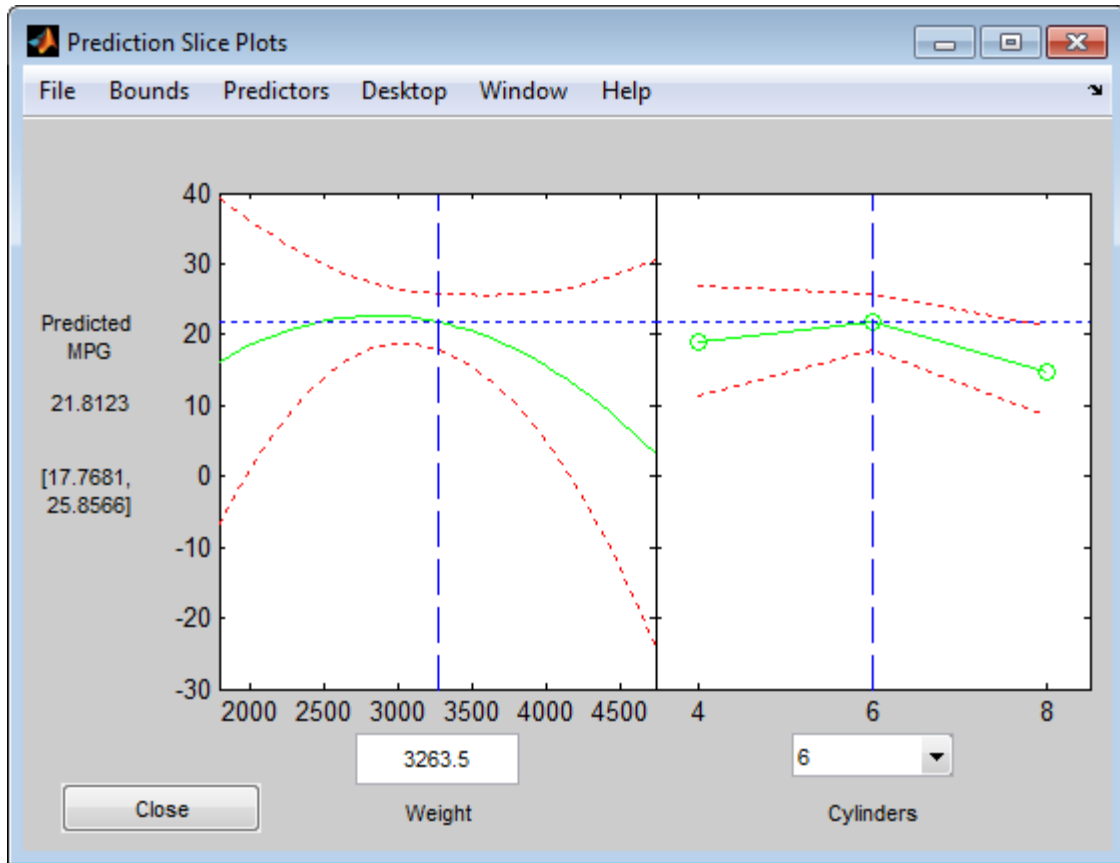
This example shows how to understand the effect each predictor has on a regression model using a variety of available plots.

- 1 Create a model of mileage from some predictors in the `carsmall` data.

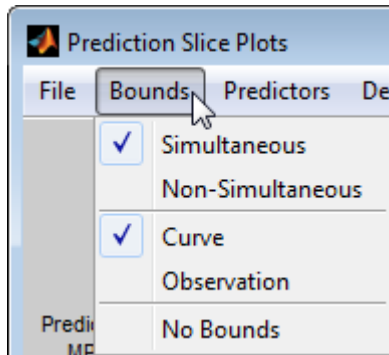
```
load carsmall
tbl = table(Weight,MPG,Cylinders);
tbl.Cylinders = ordinal(tbl.Cylinders);
mdl = fitlm(tbl,'MPG ~ Cylinders*Weight + Weight^2');
```

- 2 Examine a slice plot of the responses. This displays the effect of each predictor separately.

```
plotSlice mdl)
```

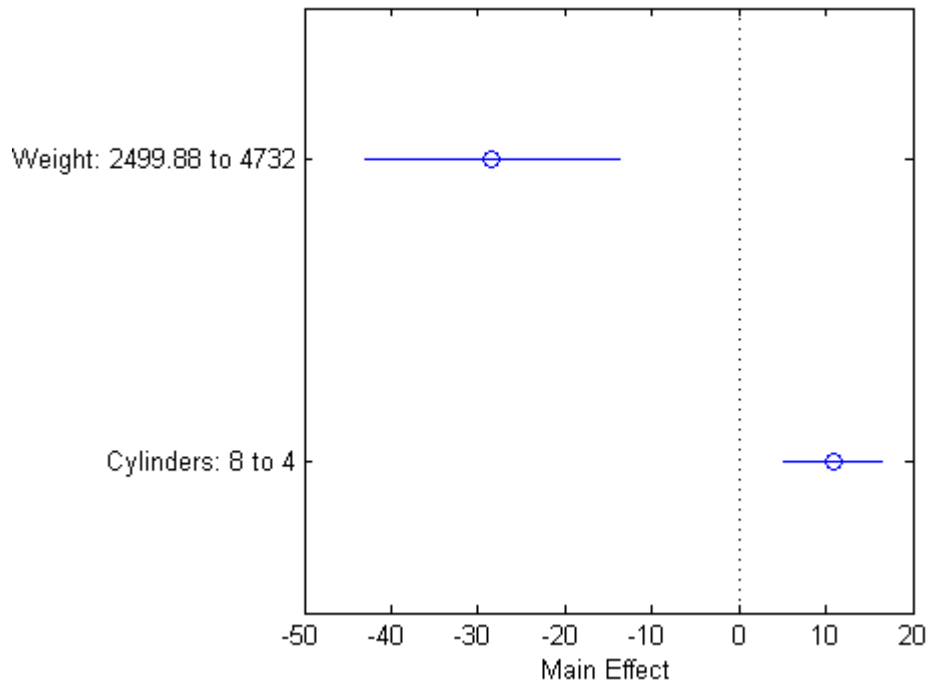


You can drag the individual predictor values, which are represented by dashed blue vertical lines. You can also choose between simultaneous and non-simultaneous confidence bounds, which are represented by dashed red curves.



- 3 Use an effects plot to show another view of the effect of predictors on the response.

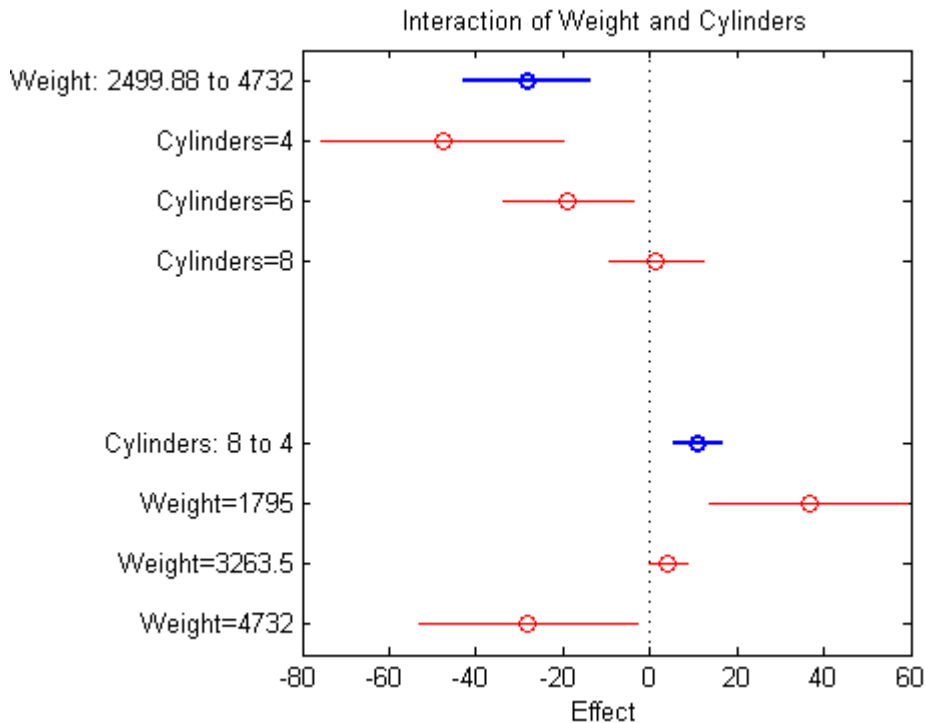
```
plotEffects(mdl)
```



This plot shows that changing `Weight` from about 2500 to 4732 lowers MPG by about 30 (the location of the upper blue circle). It also shows that changing the number of cylinders from 8 to 4 raises MPG by about 10 (the lower blue circle). The horizontal blue lines represent confidence intervals for these predictions. The predictions come from averaging over one predictor as the other is changed. In cases such as this, where the two predictors are correlated, be careful when interpreting the results.

- 4 Instead of viewing the effect of averaging over a predictor as the other is changed, examine the joint interaction in an interaction plot.

```
plotInteraction(mdl, 'Weight', 'Cylinders')
```

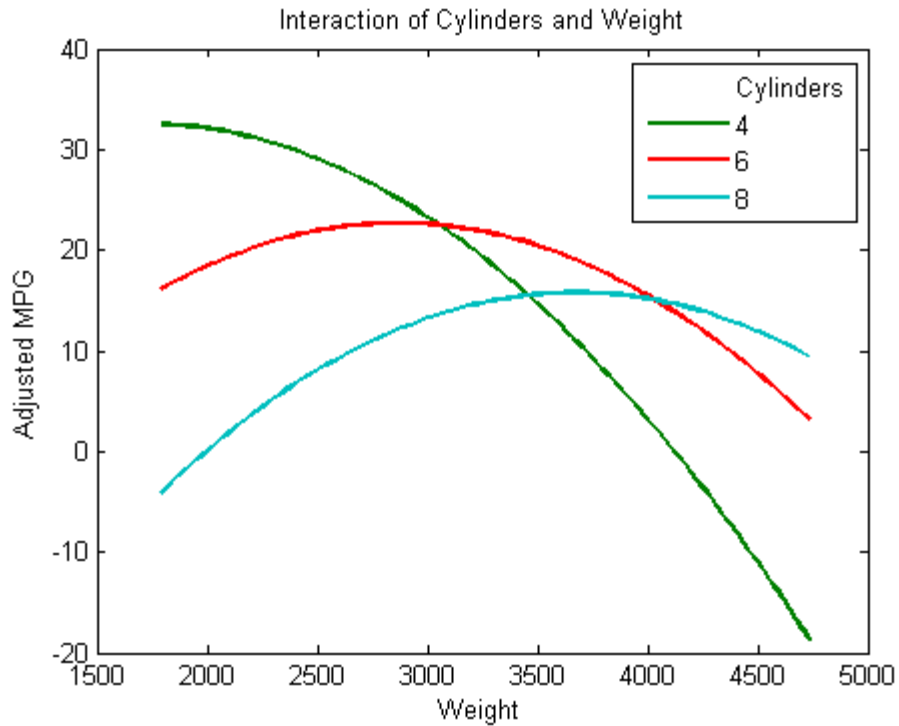


The interaction plot shows the effect of changing one predictor with the other held fixed. In this case, the plot is much more informative. It shows, for example, that lowering the number of cylinders in a relatively light car (`Weight = 1795`) leads to an increase in mileage, but lowering the number of cylinders in a relatively heavy car (`Weight = 4732`) leads to a decrease in mileage.



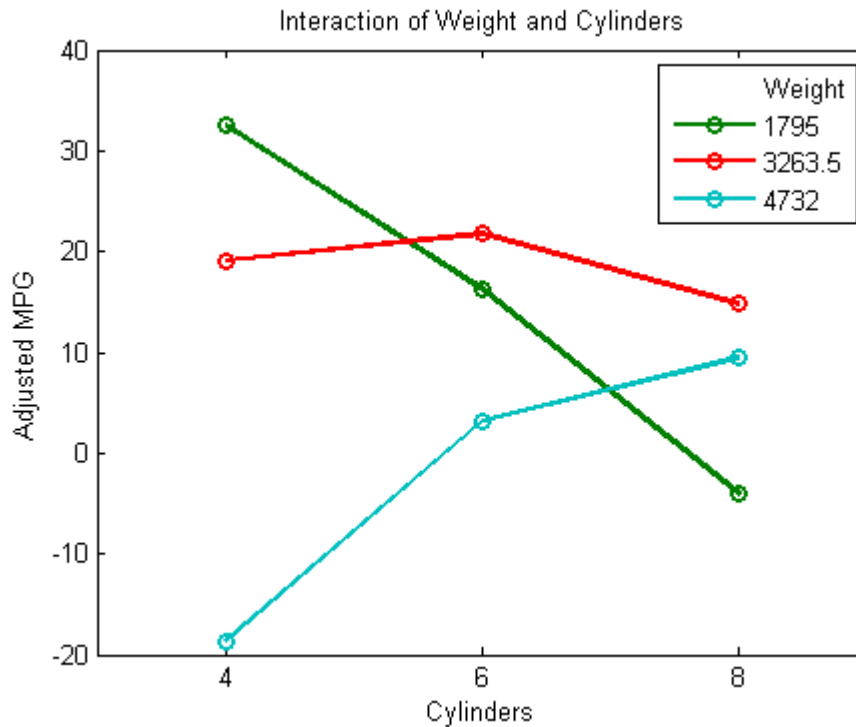
- 5 For an even more detailed look at the interactions, look at an interaction plot with predictions. This plot holds one predictor fixed while varying the other, and plots the effect as a curve. Look at the interactions for various fixed numbers of cylinders.

```
plotInteraction(mdl, 'Cylinders', 'Weight', 'predictions')
```



Now look at the interactions with various fixed levels of weight.

```
plotInteraction(mdl, 'Weight', 'Cylinders', 'predictions')
```



### Plots to Understand Terms Effects

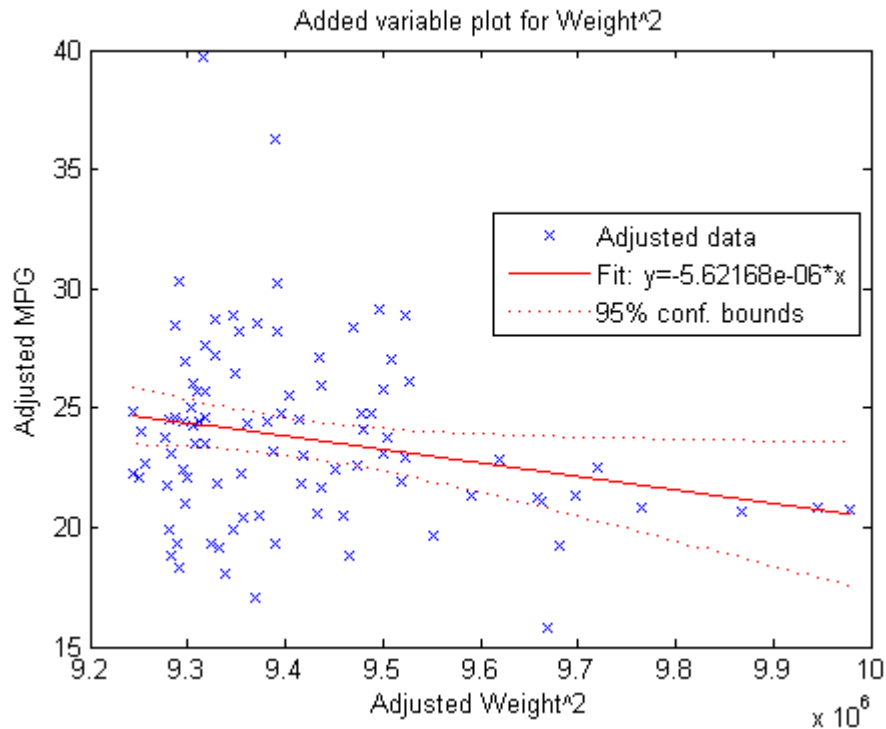
This example shows how to understand the effect of each term in a regression model using a variety of available plots.

- 1 Create a model of mileage from some predictors in the `carsmall` data.

```
load carsmall
tbl = table(Weight,MPG,Cylinders);
tbl.Cylinders = ordinal(tbl.Cylinders);
mdl = fitlm(tbl,'MPG ~ Cylinders*Weight + Weight^2');
```

- 2 Create an added variable plot with `Weight^2` as the added variable.

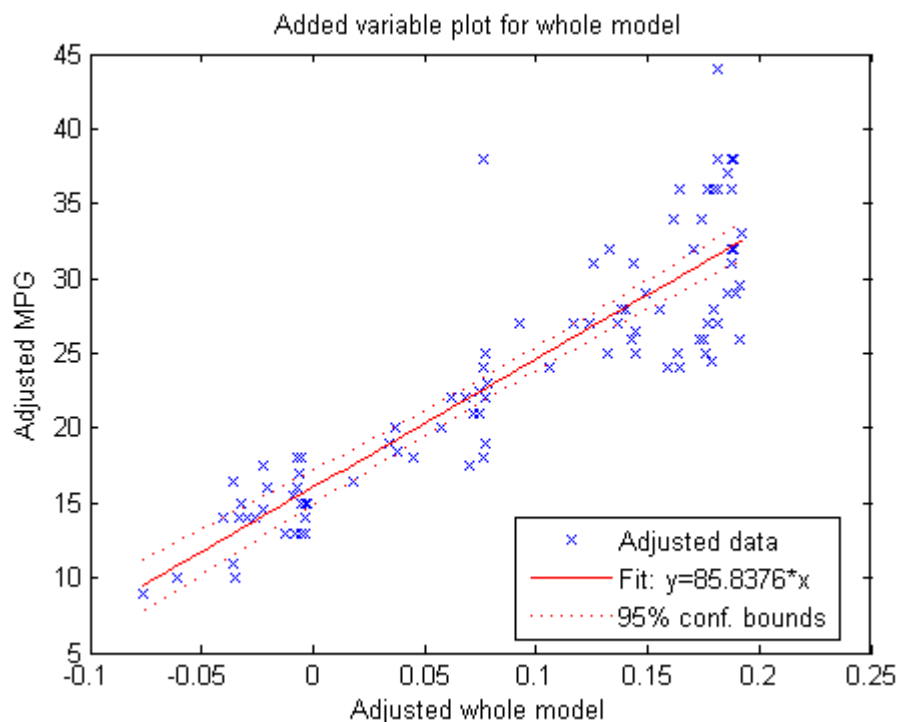
```
plotAdded(mdl,'Weight^2')
```



This plot shows the results of fitting both  $\text{Weight}^2$  and MPG to the terms other than  $\text{Weight}^2$ . The reason to use `plotAdded` is to understand what additional improvement in the model you get by adding  $\text{Weight}^2$ . The coefficient of a line fit to these points is the coefficient of  $\text{Weight}^2$  in the full model. The  $\text{Weight}^2$  predictor is just over the edge of significance ( $p\text{Value} < 0.05$ ) as you can see in the coefficients table display. You can see that in the plot as well. The confidence bounds look like they could not contain a horizontal line (constant  $y$ ), so a zero-slope model is not consistent with the data.

**3** Create an added variable plot for the model as a whole.

```
plotAdded(mdl)
```



The model as a whole is very significant, so the bounds don't come close to containing a horizontal line. The slope of the line is the slope of a fit to the predictors projected onto their best-fitting direction, or in other words, the norm of the coefficient vector.

### Change Models

There are two ways to change a model:

- `step` — Add or subtract terms one at a time, where `step` chooses the most important term to add or remove.
- `addTerms` and `removeTerms` — Add or remove specified terms. Give the terms in any of the forms described in “Choose a Model or Range of Models” on page 11-14.

If you created a model using `stepwiselm`, `step` can have an effect only if you give different upper or lower models. `step` does not work when you fit a model using `RobustOpts`.

For example, start with a linear model of mileage from the carbig data:

```
load carbig
tbl = table(Acceleration,Displacement,Horsepower,Weight,MPG);
mdl = fitlm(tbl, 'linear', 'ResponseVar', 'MPG')

mdl =
```

Linear regression model:

```
MPG ~ 1 + Acceleration + Displacement + Horsepower + Weight
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	45.251	2.456	18.424	7.0721e-55
Acceleration	-0.023148	0.1256	-0.1843	0.85388
Displacement	-0.0060009	0.0067093	-0.89441	0.37166
Horsepower	-0.043608	0.016573	-2.6312	0.008849
Weight	-0.0052805	0.00081085	-6.5123	2.3025e-10

Number of observations: 392, Error degrees of freedom: 387

Root Mean Squared Error: 4.25

R-squared: 0.707, Adjusted R-Squared 0.704

F-statistic vs. constant model: 233, p-value = 9.63e-102

Try to improve the model using step for up to 10 steps:

```
mdl1 = step(mdl, 'NSteps',10)
```

1. Adding Displacement:Horsepower, FStat = 87.4802, pValue = 7.05273e-19

```
mdl1 =
```

Linear regression model:

```
MPG ~ 1 + Acceleration + Weight + Displacement*Horsepower
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	61.285	2.8052	21.847	1.8593e-69
Acceleration	-0.34401	0.11862	-2.9	0.0039445
Displacement	-0.081198	0.010071	-8.0623	9.5014e-15
Horsepower	-0.24313	0.026068	-9.3265	8.6556e-19
Weight	-0.0014367	0.00084041	-1.7095	0.088166
Displacement:Horsepower	0.00054236	5.7987e-05	9.3531	7.0527e-19

Number of observations: 392, Error degrees of freedom: 386

Root Mean Squared Error: 3.84

R-squared: 0.761, Adjusted R-Squared 0.758

F-statistic vs. constant model: 246, p-value = 1.32e-117

step stopped after just one change.

To try to simplify the model, remove the Acceleration and Weight terms from mdl1:

```
mdl2 = removeTerms(mdl1, 'Acceleration + Weight')
```

```
mdl2 =
```

```
Linear regression model:  
MPG ~ 1 + Displacement*Horsepower
```

```
Estimated Coefficients:
```

	Estimate	SE	tStat	pValue
(Intercept)	53.051	1.526	34.765	3.0201e-121
Displacement	-0.098046	0.0066817	-14.674	4.3203e-39
Horsepower	-0.23434	0.019593	-11.96	2.8024e-28
Displacement:Horsepower	0.00058278	5.193e-05	11.222	1.6816e-25

```
Number of observations: 392, Error degrees of freedom: 388  
Root Mean Squared Error: 3.94  
R-squared: 0.747, Adjusted R-Squared 0.745  
F-statistic vs. constant model: 381, p-value = 3e-115
```

mdl2 uses just Displacement and Horsepower, and has nearly as good a fit to the data as mdl1 in the Adjusted R-Squared metric.

## Predict or Simulate Responses to New Data

There are three ways to use a linear model to predict or simulate the response to new data:

- “predict” on page 11-38
- “feval” on page 11-39
- “random” on page 11-40

### predict

This example shows how to predict and obtain confidence intervals on the predictions using the predict method.

- 1 Load the carbig data and make a default linear model of the response MPG to the Acceleration, Displacement, Horsepower, and Weight predictors.

```
load carbig  
X = [Acceleration, Displacement, Horsepower, Weight];  
mdl = fitlm(X, MPG);
```

- 2 Create a three-row array of predictors from the minimal, mean, and maximal values. There are some NaN values, so use functions that ignore NaN values.

```
Xnew = [nanmin(X);nanmean(X);nanmax(X)]; % new data
```

- 3 Find the predicted model responses and confidence intervals on the predictions.

```
[NewMPG, NewMPGCI] = predict mdl,Xnew
```

```
NewMPG =
```

```
34.1345
23.4078
4.7751
```

```
NewMPGCI =
```

```
31.6115 36.6575
22.9859 23.8298
0.6134 8.9367
```

The confidence bound on the mean response is narrower than those for the minimum or maximum responses, which is quite sensible.

## feval

When you construct a model from a table or dataset array, `feval` is often more convenient for predicting mean responses than `predict`. However, `feval` does not provide confidence bounds.

This example shows how to predict mean responses using the `feval` method.

- 1 Load the `carbig` data and make a default linear model of the response MPG to the Acceleration, Displacement, Horsepower, and Weight predictors.

```
load carbig
tbl = table(Acceleration,Displacement,Horsepower,Weight,MPG);
mdl = fitlm(tbl,'linear','ResponseVar','MPG');
```

- 2 Create a three-row array of predictors from the minimal, mean, and maximal values. There are some NaN values, so use functions that ignore NaN values.

```
X = [Acceleration,Displacement,Horsepower,Weight];
Xnew = [nanmin(X);nanmean(X);nanmax(X)]; % new data
```

The `Xnew` array has the correct number of columns for prediction, so `feval` can use it for predictions.

**3 Find the predicted model responses.**

```
NewMPG = feval mdl, Xnew

NewMPG =
    34.1345
    23.4078
     4.7751
```

**random**

The `random` method simulates new random response values, equal to the mean prediction plus a random disturbance with the same variance as the training data.

This example shows how to simulate responses using the `random` method.

**1 Load the `carbig` data and make a default linear model of the response MPG to the Acceleration, Displacement, Horsepower, and Weight predictors.**

```
load carbig
X = [Acceleration, Displacement, Horsepower, Weight];
mdl = fitlm(X, MPG);
```

**2 Create a three-row array of predictors from the minimal, mean, and maximal values. There are some NaN values, so use functions that ignore NaN values.**

```
Xnew = [nanmin(X); nanmean(X); nanmax(X)]; % new data
```

**3 Generate new predicted model responses including some randomness.**

```
rng('default') % for reproducibility
NewMPG = random(mdl, Xnew)
```

```
NewMPG =

    36.4178
    31.1958
    -4.8176
```

**4 Because a negative value of MPG does not seem sensible, try predicting two more times.**

```
NewMPG = random(mdl, Xnew)

NewMPG =

    37.7959
```



```

24.7615
-0.7783

NewMPG = random mdl, Xnew)

NewMPG =

32.2931
24.8628
19.9715

```

Clearly, the predictions for the third (maximal) row of `Xnew` are not reliable.

## Share Fitted Models

Suppose you have a linear regression model, such as `mdl` from the following commands:

```

load carbig
tbl = table(Acceleration, Displacement, Horsepower, Weight, MPG);
mdl = fitlm(tbl, 'linear', 'ResponseVar', 'MPG');

```

To share the model with other people, you can:

- Provide the model display.

```
mdl
```

```
mdl =
```

```
Linear regression model:
```

```
MPG ~ 1 + Acceleration + Displacement + Horsepower + Weight
```

```
Estimated Coefficients:
```

	Estimate	SE	tStat	pValue
(Intercept)	45.251	2.456	18.424	7.0721e-55
Acceleration	-0.023148	0.1256	-0.1843	0.85388
Displacement	-0.0060009	0.0067093	-0.89441	0.37166
Horsepower	-0.043608	0.016573	-2.6312	0.008849
Weight	-0.0052805	0.00081085	-6.5123	2.3025e-10

```
Number of observations: 392, Error degrees of freedom: 387
```

```
Root Mean Squared Error: 4.25
```

R-squared: 0.707, Adjusted R-Squared 0.704  
F-statistic vs. constant model: 233, p-value = 9.63e-102

- Provide just the model definition and coefficients.

```
mdl.CoefficientNames
```

```
ans =
```

```
    '(Intercept)'    'Acceleration'    'Displacement'    'Horsepower'    'Weight'
```

```
mdl.Coefficients.Estimate
```

```
ans =
```

```
    45.2511  
   -0.0231  
   -0.0060  
   -0.0436  
   -0.0053
```

```
mdl.Formula
```

```
ans =
```

```
MPG ~ 1 + Acceleration + Displacement + Horsepower + Weight
```

## Linear Regression Workflow

This example shows how to fit a linear regression model. A typical workflow involves the following: import data, fit a regression, test its quality, modify it to improve the quality, and share it.

### Step 1. Import the data into a table.

`hospital.xls` is an Excel® spreadsheet containing patient names, sex, age, weight, blood pressure, and dates of treatment in an experimental protocol. First read the data into a table.

```
patients = readtable('hospital.xls',...
    'ReadRowNames',true);
```

Examine the first row of data.

```
patients(1,:)
```

```
ans =
```

```
1x11 table
```

	name	sex	age	wgt	smoke	sys	dia	trial1	trial2
YPL-320	'SMITH'	'm'	38	176	1	124	93	18	-99

The sex and smoke fields seem to have two choices each. So change these fields to categorical.

```
patients.smoke = categorical(patients.smoke,0:1,{'No','Yes'});
patients.sex = categorical(patients.sex);
```

### Step 2. Create a fitted model.

Your goal is to model the systolic pressure as a function of a patient's age, weight, sex, and smoking status. Create a linear formula for 'sys' as a function of 'age', 'wgt', 'sex', and 'smoke'.

```
modelspec = 'sys ~ age + wgt + sex + smoke';
mdl = fitlm(patients,modelspec)
```

```
mdl =
```

```
Linear regression model:
```

```
sys ~ 1 + sex + age + wgt + smoke
```

```
Estimated Coefficients:
```

	Estimate	SE	tStat	pValue
(Intercept)	118.28	7.6291	15.504	9.1557e-28
sex_m	0.88162	2.9473	0.29913	0.76549
age	0.08602	0.06731	1.278	0.20438
wgt	-0.016685	0.055714	-0.29947	0.76524
smoke_Yes	9.884	1.0406	9.498	1.9546e-15

```
Number of observations: 100, Error degrees of freedom: 95
```

```
Root Mean Squared Error: 4.81
```

```
R-squared: 0.508, Adjusted R-Squared 0.487
```

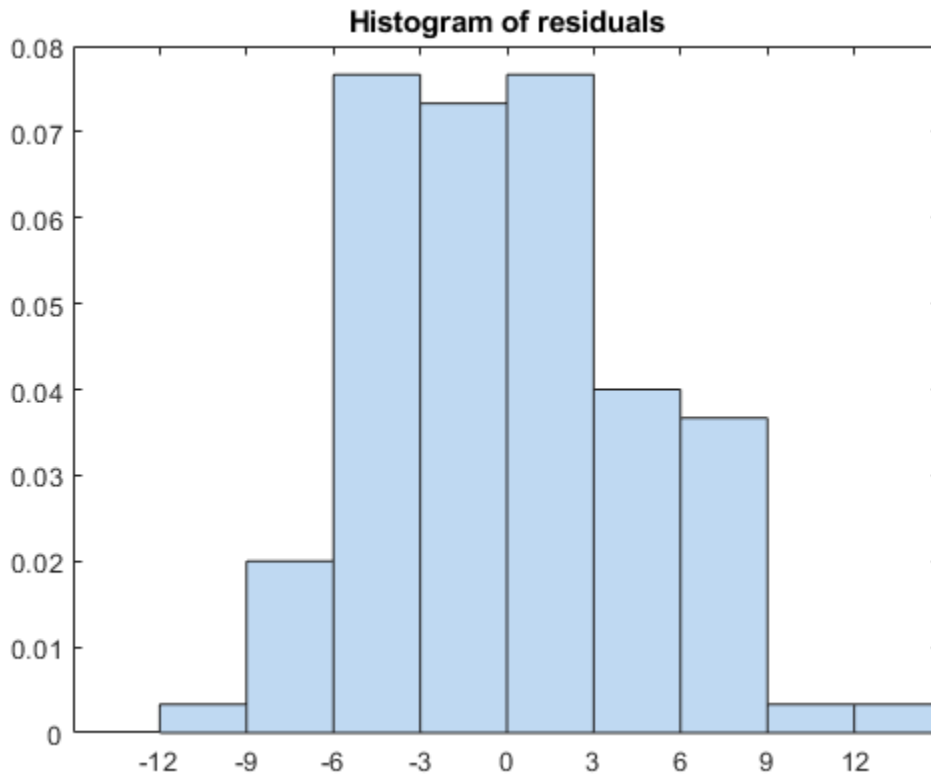
```
F-statistic vs. constant model: 24.5, p-value = 5.99e-14
```

The sex, age, and weight predictors have rather high  $P$ -values, indicating that some of these predictors might be unnecessary.

### Step 3. Locate and remove outliers.

See if there are outliers in the data that should be excluded from the fit. Plot the residuals.

```
plotResiduals(mdl)
```



There is one possible outlier, with a value greater than 12. This is probably not truly an outlier. For demonstration, here is how to find and remove it.

Find the outlier.

```
outlier = mdl.Residuals.Raw > 12;  
find(outlier)
```

```
ans =
```

```
84
```

Remove the outlier.

```
mdl = fitlm(patients,modelspec,...
            'Exclude',84);

mdl.ObservationInfo(84,:)

ans =

    1x4 table

           Weights   Excluded   Missing   Subset
           _____   _____   _____   _____
WXM-486     1         true       false     false
```

Observation 84 is no longer in the model.

#### Step 4. Simplify the model.

Try to obtain a simpler model, one with fewer predictors but the same predictive accuracy. `step` looks for a better model by adding or removing one term at a time. Allow `step` take up to 10 steps.

```
mdl1 = step(mdl, 'NSteps',10)

1. Removing wgt, FStat = 4.6001e-05, pValue = 0.9946
2. Removing sex, FStat = 0.063241, pValue = 0.80199

mdl1 =

Linear regression model:
  sys ~ 1 + age + smoke

Estimated Coefficients:
           Estimate           SE           tStat           pValue
           _____           _____           _____           _____
(Intercept)    115.11          2.5364          45.383          1.1407e-66
age             0.10782         0.064844         1.6628           0.09962
smoke_Yes      10.054          0.97696         10.291          3.5276e-17
```

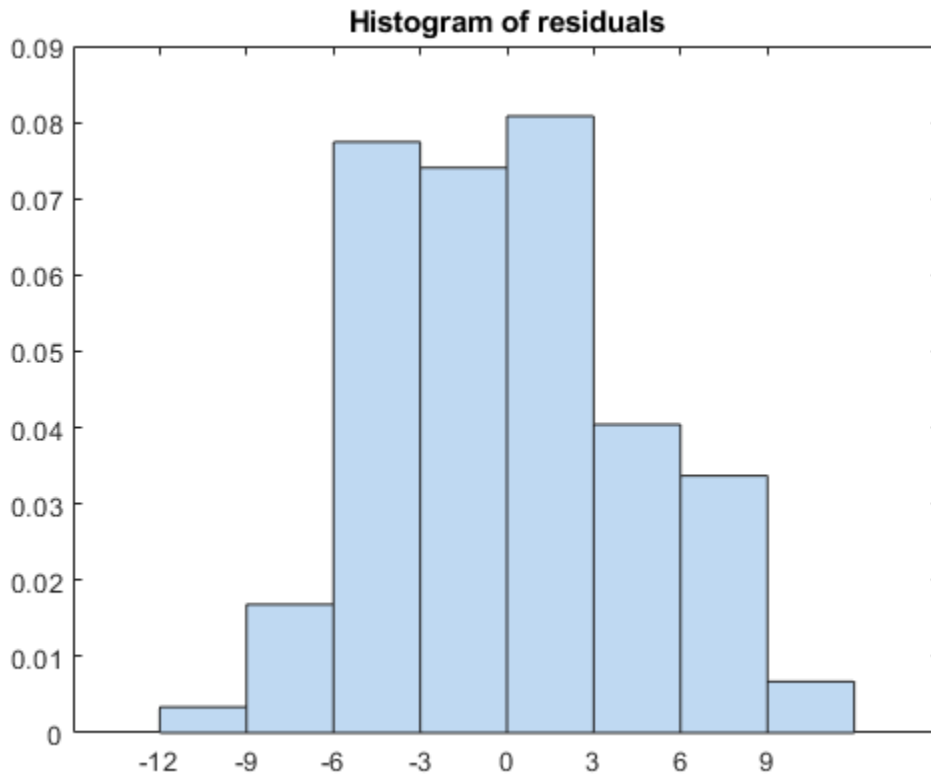
```
Number of observations: 99, Error degrees of freedom: 96
```

```
Root Mean Squared Error: 4.61  
R-squared: 0.536, Adjusted R-Squared 0.526  
F-statistic vs. constant model: 55.4, p-value = 1.02e-16
```

step took two steps. This means it could not improve the model further by adding or subtracting a single term.

Plot the effectiveness of the simpler model on the training data.

```
plotResiduals(md11)
```



The residuals look about as small as those of the original model.

**Step 5. Predict responses to new data.**

Suppose you have four new people, aged 25, 30, 40, and 65, and the first and third smoke. Predict their systolic pressure using `mdl1`.

```
ages = [25;30;40;65];
smoker = {'Yes';'No';'Yes';'No'};
systolicnew = feval(mdl1,ages,smoker)
```

```
systolicnew =

    127.8561
    118.3412
    129.4734
    122.1149
```

To make predictions, you need only the variables that `mdl1` uses.

**Step 6. Share the model.**

You might want others to be able to use your model for prediction. Access the terms in the linear model.

```
coefnames = mdl1.CoefficientNames

coefnames =

    1x3 cell array

    {'(Intercept)'}    {'age'}    {'smoke_Yes'}
```

View the model formula.

```
mdl1.Formula

ans =

sys ~ 1 + age + smoke
```

Access the coefficients of the terms.



```
coefvals = mdl1.Coefficients(:,1); % table  
coefvals = table2array(coefvals)
```

```
coefvals =
```

```
115.1066  
  0.1078  
 10.0540
```

**The model is  $\text{sys} = 115.1066 + 0.1078 \cdot \text{age} + 10.0540 \cdot \text{smoke}$ , where smoke is 1 for a smoker, and 0 otherwise.**

## Regression Using Dataset Arrays

This example shows how to perform linear and stepwise regression analyses using dataset arrays.

### Load sample data.

```
load imports-85
```

### Store predictor and response variables in dataset array.

```
ds = dataset(X(:,7),X(:,8),X(:,9),X(:,15), 'Varnames', ...
{'curb_weight', 'engine_size', 'bore', 'price'});
```

### Fit linear regression model.

Fit a linear regression model that explains the price of a car in terms of its curb weight, engine size, and bore.

```
fitlm(ds, 'price~curb_weight+engine_size+bore')
```

```
ans =
```

```
Linear regression model:
```

```
price ~ 1 + curb_weight + engine_size + bore
```

```
Estimated Coefficients:
```

	Estimate	SE	tStat	pValue
(Intercept)	64.095	3.703	17.309	2.0481e-41
curb_weight	-0.0086681	0.0011025	-7.8623	2.42e-13
engine_size	-0.015806	0.013255	-1.1925	0.23452
bore	-2.6998	1.3489	-2.0015	0.046711

```
Number of observations: 201, Error degrees of freedom: 197
```

```
Root Mean Squared Error: 3.95
```

```
R-squared: 0.674, Adjusted R-Squared 0.669
```

```
F-statistic vs. constant model: 136, p-value = 1.14e-47
```

The command `fitlm(ds)` also returns the same result because `fitlm`, by default, assumes the predictor variable is in the last column of the dataset array `ds`.

**Recreate dataset array and repeat analysis.**

This time, put the response variable in the first column of the dataset array.

```
ds = dataset(X(:,15),X(:,7),X(:,8),X(:,9), 'Varnames', ...
{'price', 'curb_weight', 'engine_size', 'bore'});
```

When the response variable is in the first column of `ds`, define its location. For example, `fitlm`, by default, assumes that `bore` is the response variable. You can define the response variable in the model using either:

```
fitlm(ds, 'ResponseVar', 'price');
```

or

```
fitlm(ds, 'ResponseVar', logical([1 0 0 0]));
```

**Perform stepwise regression.**

```
stepwiselm(ds, 'quadratic', 'lower', 'price~1', ...
'ResponseVar', 'price')
```

1. Removing `bore^2`, FStat = 0.01282, pValue = 0.90997
2. Removing `engine_size^2`, FStat = 0.078043, pValue = 0.78027
3. Removing `curb_weight:bore`, FStat = 0.70558, pValue = 0.40195

ans =

Linear regression model:

```
price ~ 1 + curb_weight*engine_size + engine_size*bore + curb_weight^2
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	131.13	14.273	9.1873	6.2319e-17
curb_weight	-0.043315	0.0085114	-5.0891	8.4682e-07
engine_size	-0.17102	0.13844	-1.2354	0.21819
bore	-12.244	4.999	-2.4493	0.015202
curb_weight:engine_size	-6.3411e-05	2.6577e-05	-2.386	0.017996
engine_size:bore	0.092554	0.037263	2.4838	0.013847
curb_weight^2	8.0836e-06	1.9983e-06	4.0451	7.5432e-05

```
Number of observations: 201, Error degrees of freedom: 194
Root Mean Squared Error: 3.59
R-squared: 0.735, Adjusted R-Squared 0.726
F-statistic vs. constant model: 89.5, p-value = 3.58e-53
```

The initial model is a quadratic formula, and the lowest model considered is the constant. Here, `stepwiselm` performs a backward elimination technique to determine the terms in the model. The final model is `price ~ 1 + curb_weight*engine_size + engine_size*bore + curb_weight^2`, which corresponds to

$$P = \beta_0 + \beta_C C + \beta_E E + \beta_B B + \beta_{CE} CE + \beta_{EB} EB + \beta_{C^2} C^2 + \epsilon$$

where  $P$  is price,  $C$  is curb weight,  $E$  is engine size,  $B$  is bore,  $\beta_i$  is the coefficient for the corresponding term in the model, and  $\epsilon$  is the error term. The final model includes all three main effects, the interaction effects for curb weight and engine size and engine size and bore, and the second-order term for curb weight.

## See Also

`LinearModel` | `fitlm` | `stepwiselm`

## Related Examples

- “Examine Quality and Adjust the Fitted Model” on page 11-20
- “Interpret Linear Regression Results” on page 11-64

## Regression Using Tables

This example shows how to perform linear and stepwise regression analyses using tables.

### Load sample data.

```
load imports-85
```

### Store predictor and response variables in a table.

```
tbl = table(X(:,7),X(:,8),X(:,9),X(:,15), 'VariableNames', ...
{'curb_weight', 'engine_size', 'bore', 'price'});
```

### Fit linear regression model.

Fit a linear regression model that explains the price of a car in terms of its curb weight, engine size, and bore.

```
fitlm(tbl, 'price~curb_weight+engine_size+bore')
```

```
ans =
```

```
Linear regression model:
```

```
price ~ 1 + curb_weight + engine_size + bore
```

```
Estimated Coefficients:
```

	Estimate	SE	tStat	pValue
(Intercept)	64.095	3.703	17.309	2.0481e-41
curb_weight	-0.0086681	0.0011025	-7.8623	2.42e-13
engine_size	-0.015806	0.013255	-1.1925	0.23452
bore	-2.6998	1.3489	-2.0015	0.046711

```
Number of observations: 201, Error degrees of freedom: 197
```

```
Root Mean Squared Error: 3.95
```

```
R-squared: 0.674, Adjusted R-Squared 0.669
```

```
F-statistic vs. constant model: 136, p-value = 1.14e-47
```

The command `fitlm(tbl)` also returns the same result because `fitlm`, by default, assumes the predictor variable is in the last column of the table `tbl`.

**Recreate table and repeat analysis.**

This time, put the response variable in the first column of the table.

```
tbl = table(X(:,15),X(:,7),X(:,8),X(:,9), 'VariableNames', ...
{'price', 'curb_weight', 'engine_size', 'bore'});
```

When the response variable is in the first column of `tbl`, define its location. For example, `fitlm`, by default, assumes that `bore` is the response variable. You can define the response variable in the model using either:

```
fitlm(tbl, 'ResponseVar', 'price');
```

or

```
fitlm(tbl, 'ResponseVar', logical([1 0 0 0]));
```

**Perform stepwise regression.**

```
stepwiselm(tbl, 'quadratic', 'lower', 'price~1', ...
'ResponseVar', 'price')
```

1. Removing `bore^2`, `FStat` = 0.01282, `pValue` = 0.90997
2. Removing `engine_size^2`, `FStat` = 0.078043, `pValue` = 0.78027
3. Removing `curb_weight:bore`, `FStat` = 0.70558, `pValue` = 0.40195

`ans =`

Linear regression model:

```
price ~ 1 + curb_weight*engine_size + engine_size*bore + curb_weight^2
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	131.13	14.273	9.1873	6.2319e-17
curb_weight	-0.043315	0.0085114	-5.0891	8.4682e-07
engine_size	-0.17102	0.13844	-1.2354	0.21819
bore	-12.244	4.999	-2.4493	0.015202
curb_weight:engine_size	-6.3411e-05	2.6577e-05	-2.386	0.017996
engine_size:bore	0.092554	0.037263	2.4838	0.013847
curb_weight^2	8.0836e-06	1.9983e-06	4.0451	7.5432e-05

```

Number of observations: 201, Error degrees of freedom: 194
Root Mean Squared Error: 3.59
R-squared: 0.735, Adjusted R-Squared 0.726
F-statistic vs. constant model: 89.5, p-value = 3.58e-53

```

The initial model is a quadratic formula, and the lowest model considered is the constant. Here, `stepwiselm` performs a backward elimination technique to determine the terms in the model. The final model is  $\text{price} \sim 1 + \text{curb\_weight} * \text{engine\_size} + \text{engine\_size} * \text{bore} + \text{curb\_weight}^2$ , which corresponds to

$$P = \beta_0 + \beta_C C + \beta_E E + \beta_B B + \beta_{CE} CE + \beta_{EB} EB + \beta_{C^2} C^2 + \epsilon$$

where  $P$  is price,  $C$  is curb weight,  $E$  is engine size,  $B$  is bore,  $\beta_i$  is the coefficient for the corresponding term in the model, and  $\epsilon$  is the error term. The final model includes all three main effects, the interaction effects for curb weight and engine size and engine size and bore, and the second-order term for curb weight.

## See Also

`LinearModel` | `fitlm` | `stepwiselm`

## Related Examples

- “Examine Quality and Adjust the Fitted Model” on page 11-20
- “Interpret Linear Regression Results” on page 11-64

## Linear Regression with Interaction Effects

Construct and analyze a linear regression model with interaction effects and interpret the results.

### Load sample data.

```
load hospital
```

To retain only the first column of blood pressure, store data in a new dataset array.

```
ds = dataset(hospital.Sex,hospital.Age,hospital.Weight,hospital.Smoker,...
hospital.BloodPressure(:,1),'Varnames',{'Sex','Age','Weight','Smoker',...
'BloodPressure'});
```

### Perform stepwise linear regression.

For the initial model, use the full model with all terms and their pairwise interactions.

```
mdl = stepwiselm(ds,'interactions')
```

```
1. Removing Sex:Smoker, FStat = 0.050738, pValue = 0.8223
2. Removing Weight:Smoker, FStat = 0.07758, pValue = 0.78124
3. Removing Age:Weight, FStat = 1.9717, pValue = 0.16367
4. Removing Sex:Age, FStat = 0.32389, pValue = 0.57067
5. Removing Age:Smoker, FStat = 2.4939, pValue = 0.11768
```

```
mdl =
```

Linear regression model:

```
BloodPressure ~ 1 + Age + Smoker + Sex*Weight
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	133.17	10.337	12.883	1.76e-22
Sex_Male	-35.269	17.524	-2.0126	0.047015
Age	0.11584	0.067664	1.712	0.090198
Weight	-0.1393	0.080211	-1.7367	0.085722
Smoker_1	9.8307	1.0229	9.6102	1.2391e-15
Sex_Male:Weight	0.2341	0.11192	2.0917	0.039162



Number of observations: 100, Error degrees of freedom: 94  
 Root Mean Squared Error: 4.72  
 R-squared: 0.53, Adjusted R-Squared 0.505  
 F-statistic vs. constant model: 21.2, p-value = 4e-14

The final model in formula form is  $\text{BloodPressure} \sim 1 + \text{Age} + \text{Smoker} + \text{Sex} * \text{Weight}$ . This model includes all four main effects (Age, Smoker, Sex, Weight) and the two-way interaction between Sex and Weight. This model corresponds to  $BP = \beta_0 + \beta_A X_A + \beta_{Sm} I_{Sm} + \beta_S I_S + \beta_W X_W + \beta_{SW} X_W I_S + \epsilon$ ,

where

- $BP$  is the blood pressure
- $\beta_i$  are the coefficients
- $I_{Sm}$  is the indicator variable for smoking;  $I_{Sm} = 1$  indicates a smoking patient whereas  $I_{Sm} = 0$  indicates a nonsmoking patient
- $I_S$  is the indicator variable for sex;  $I_S = 1$  indicates a male patient whereas  $I_S = 0$  indicates a female patient
- $X_A$  is the Age variable
- $X_W$  is the Weight variable
- $\epsilon$  is the error term

The following table shows the fitted linear model for each gender and smoking combination.

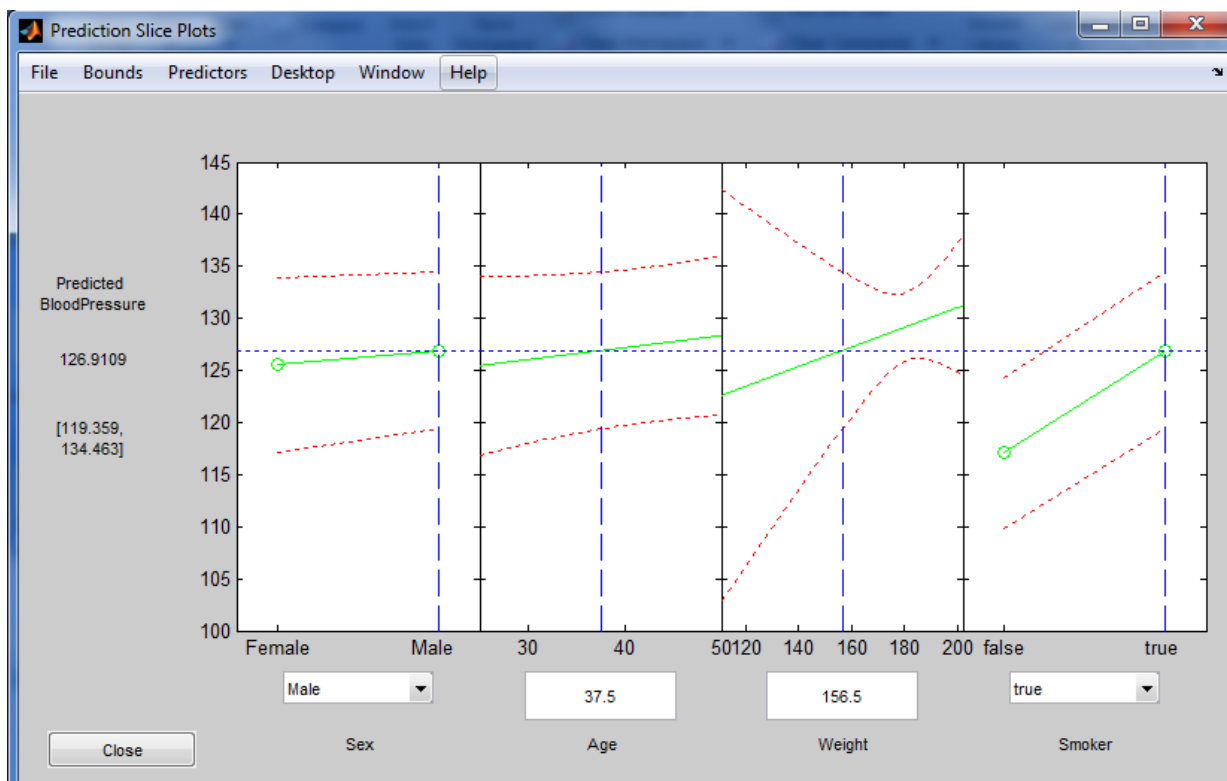
$I_{Sm}$	$I_S$	Linear Model
1(Smoker)	1(Male)	$BP = (\beta_0 + \beta_{Sm} + \beta_S) + \beta_A X_A + (\beta_W + \beta_{SW}) X_W$ $\hat{BP} = 107.5617 + 0.11584 X_A + 0.11826 X_W$
1(Smoker)	0(Female)	$BP = (\beta_0 + \beta_{Sm}) + \beta_A X_A + \beta_W X_W$ $\hat{BP} = 143.0007 + 0.11584 X_A - 0.1393 X_W$
0(Nonsmoker)	1(Male)	$BP = (\beta_0 + \beta_S) + \beta_A X_A + (\beta_W + \beta_{SW}) X_W$ $\hat{BP} = 97.901 + 0.11584 X_A + 0.11826 X_W$
0(Nonsmoker)	0(Female)	$BP = \beta_0 + \beta_A X_A + \beta_W X_W$ $\hat{BP} = 133.17 + 0.11584 X_A - 0.1393 X_W$

As seen from these models,  $\beta_{Sm}$  and  $\beta_S$  show how much the intercept of the response function changes when the indicator variable takes the value 1 compared to when it

takes the value 0.  $\beta_{sw}$ , however, shows the effect of the Weight variable on the response variable when the indicator variable for sex takes the value 1 compared to when it takes the value 0. You can explore the main and interaction effects in the final model using the methods of the `LinearModel` class as follows.

### Plot prediction slice plots.

```
figure
plotSlice mdl
```



This plot shows the main effects for all predictor variables. The green line in each panel shows the change in the response variable as a function of the predictor variable when all other predictor variables are held constant. For example, for a smoking male patient aged 37.5, the expected blood pressure increases as the weight of the patient increases, given all else the same.

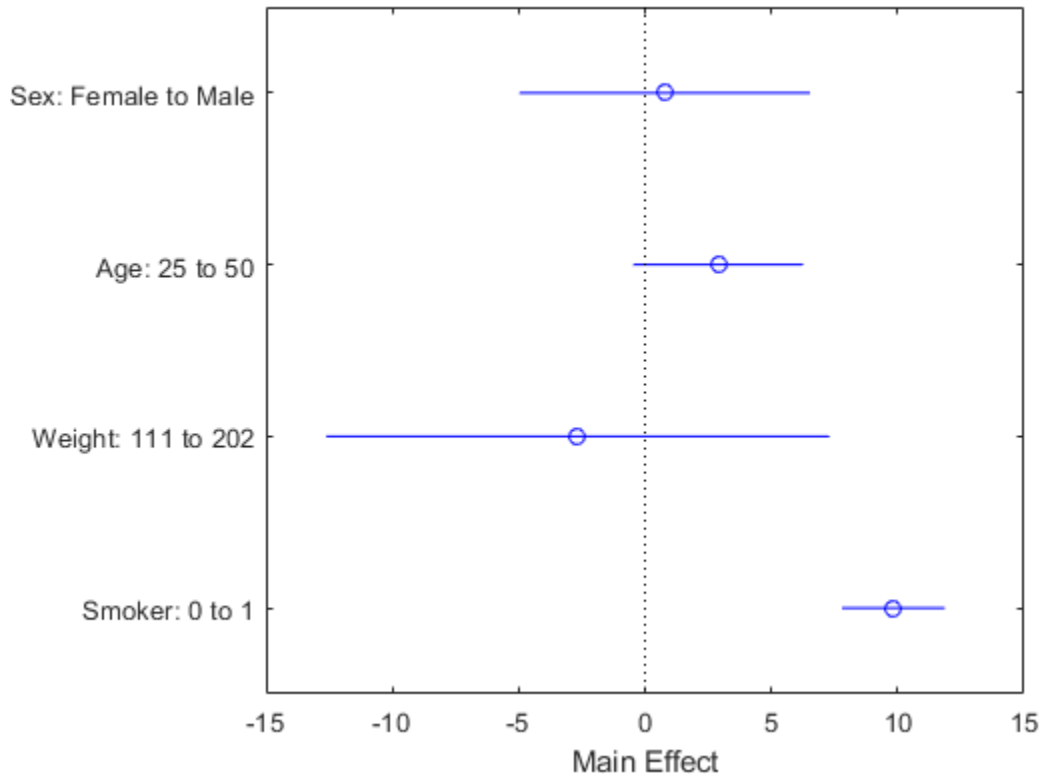
The dashed red curves in each panel show the 95% confidence bounds for the predicted response values.

The horizontal dashed blue line in each panel shows the predicted response for the specific value of the predictor variable corresponding to the vertical dashed blue line. You can drag these lines to get the predicted response values at other predictor values, as shown next.

For example, the predicted value of the response variable is 118.3497 when a patient is female, nonsmoking, age 40.3788, and weighs 139.9545 pounds. The values in the square brackets, [114.621, 122.079], show the lower and upper limits of a 95% confidence interval for the estimated response. Note that, for a nonsmoking female patient, the expected blood pressure decreases as the weight increases, given all else is held constant.

**Plot main effects.**

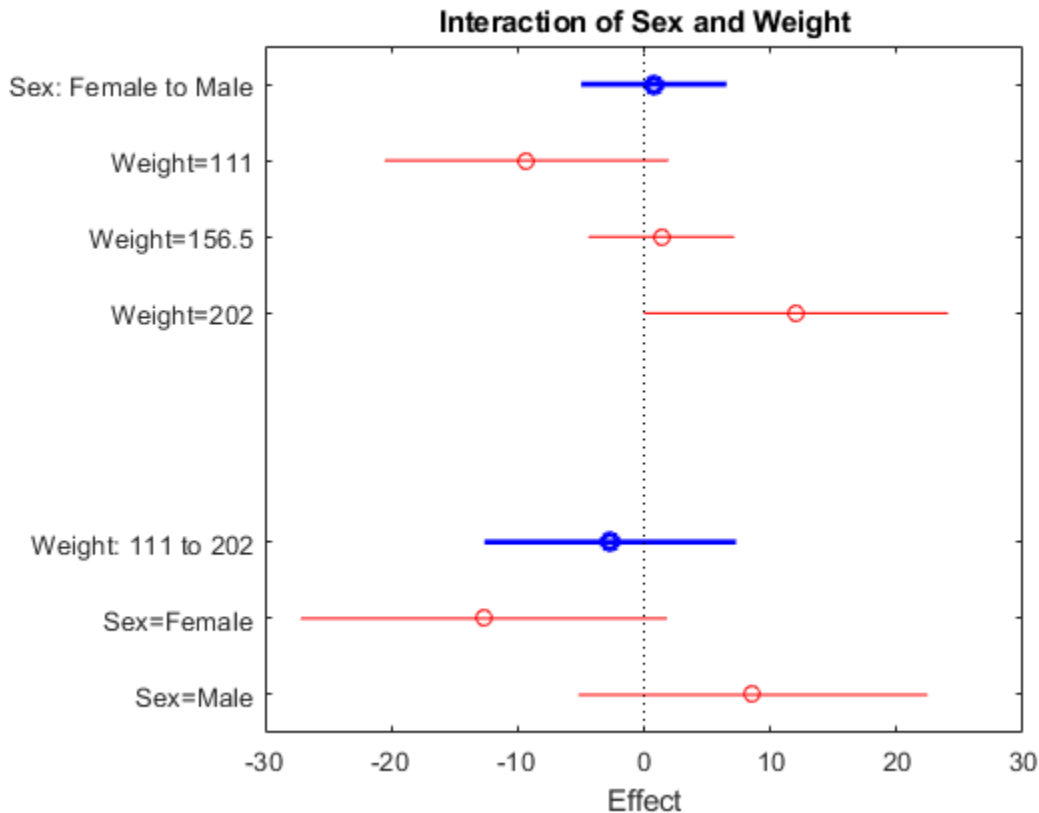
```
plotEffects mdl)
```



This plot displays the main effects. The circles show the magnitude of the effect and the blue lines show the upper and lower confidence limits for the main effect. For example, being a smoker increases the expected blood pressure by 10 units, compared to being a nonsmoker, given all else is held constant. Expected blood pressure increases about two units for males compared to females, again, given other predictors held constant. An increase in age from 25 to 50 causes an expected increase of 4 units, whereas a change in weight from 111 to 202 causes about a 4-unit decrease in the expected blood pressure, given all else held constant.

#### Plot interaction effects.

```
figure  
plotInteraction mdl, 'Sex', 'Weight')
```



This plot displays the impact of a change in one factor given the other factor is fixed at a value.

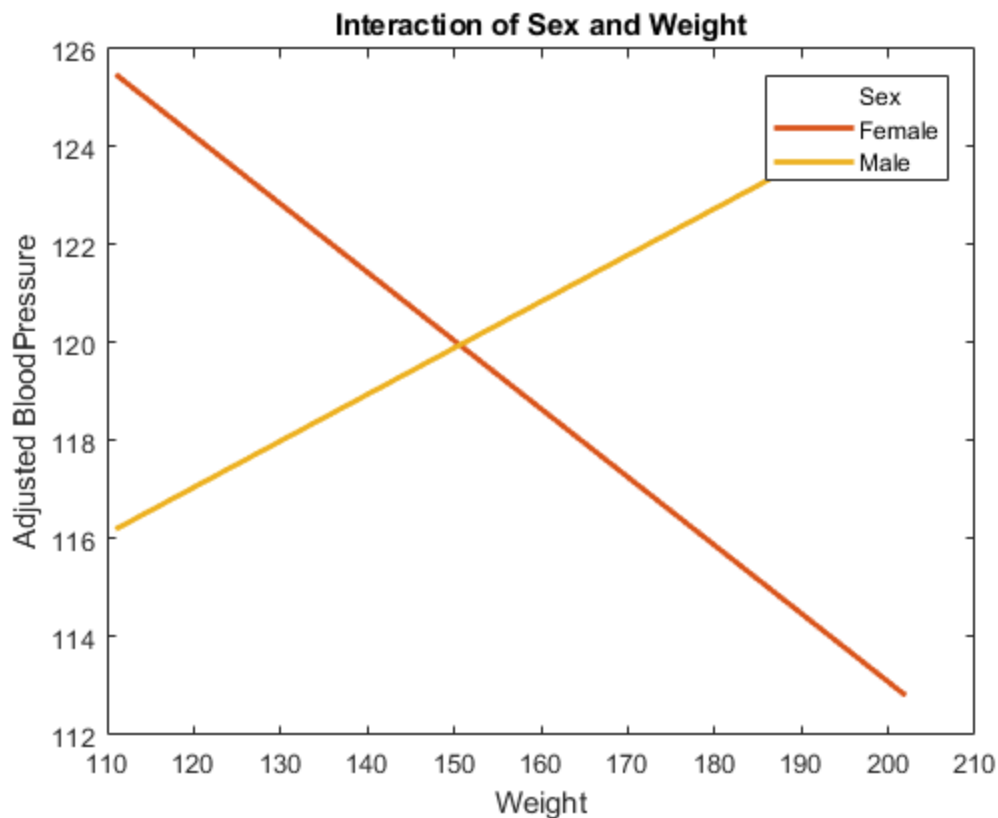
Be cautious while interpreting the interaction effects. When there is not enough data on all factor combinations or the data is highly correlated, it might be difficult to determine the interaction effect of changing one factor while keeping the other fixed. In such cases, the estimated interaction effect is an extrapolation from the data.

The blue circles show the main effect of a specific term, as in the main effects plot. The red circles show the impact of a change in one term for fixed values of the other term. For example, in the bottom half of this plot, the red circles show the impact of a weight change in female and male patients, separately. You can see that an increase in a female's weight from 111 to 202 pounds causes about a 14-unit decrease in the expected

blood pressure, while an increase of the same amount in the weight of a male patient causes about a 5-unit increase in the expected blood pressure, again given other predictors are held constant.

### Plot prediction effects.

```
figure  
plotInteraction(mdl, 'Sex', 'Weight', 'predictions')
```



This plot shows the effect of changing one variable as the other predictor variable is held constant. In this example, the last figure shows the response variable, blood pressure, as a function of weight, when the variable sex is fixed at males and females. The lines for males and females are crossing which indicates a strong interaction between weight and

sex. You can see that the expected blood pressure increases as the weight of a male patient increases, but decreases as the weight of a female patient increases.

## See Also

`LinearModel` | `fitlm` | `plotEffects` | `plotInteraction` | `plotSlice` | `stepwiselm`

## Related Examples

- “Plots to Understand Predictor Effects” on page 11-29

## Interpret Linear Regression Results

This example shows how to display and interpret linear regression output statistics.

### Load sample data and define predictor variables.

```
load carsmall
X = [Weight,Horsepower,Acceleration];
```

### Fit linear regression model.

```
lm = fitlm(X,MPG,'linear')
```

```
lm =
```

```
Linear regression model:
  y ~ 1 + x1 + x2 + x3
```

```
Estimated Coefficients:
```

	Estimate	SE	tStat	pValue
(Intercept)	47.977	3.8785	12.37	4.8957e-21
x1	-0.0065416	0.0011274	-5.8023	9.8742e-08
x2	-0.042943	0.024313	-1.7663	0.08078
x3	-0.011583	0.19333	-0.059913	0.95236

```
Number of observations: 93, Error degrees of freedom: 89
```

```
Root Mean Squared Error: 4.09
```

```
R-squared: 0.752, Adjusted R-Squared 0.744
```

```
F-statistic vs. constant model: 90, p-value = 7.38e-27
```

This linear regression output displays the following.

y ~ 1 + x1 + x2 + x3	Linear regression model in the formula form using Wilkinson notation. Here it corresponds to: $y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \beta_3 X_3 + \varepsilon$ .
First column (under Estimated Coefficients)	Terms included in the model.



Estimate	Coefficient estimates for each corresponding term in the model. For example, the estimate for the constant term (intercept) is 47.977.
SE	Standard error of the coefficients.
tStat	$t$ -statistic for each coefficient to test the null hypothesis that the corresponding coefficient is zero against the alternative that it is different from zero, given the other predictors in the model. Note that $tStat = Estimate/SE$ . For example, the $t$ -statistic for the intercept is $47.977/3.8785 = 12.37$ .
pValue	$p$ -value for the F statistic of the hypotheses test that the corresponding coefficient is equal to zero or not. For example, the $p$ -value of the F-statistic for <code>x2</code> is greater than 0.05, so this term is not significant at the 5% significance level given the other terms in the model.
Number of observations	Number of rows without any NaN values. For example, Number of observations is 93 because the MPG data vector has 6 NaN values and one of the data vectors, Horsepower, has one NaN value for a different observation.
Error degrees of freedom	$n - p$ , where $n$ is the number of observations, and $p$ is the number of coefficients in the model, including the intercept. For example, the model has four predictors, so the Error degrees of freedom is $93 - 4 = 89$ .
Root mean squared error	Square root of the mean squared error, which estimates the standard deviation of the error distribution.
R-squared and Adjusted R-squared	Coefficient of determination and adjusted coefficient of determination, respectively. For example, the R-squared value suggests that the model explains approximately 75% of the variability in the response variable MPG.
F-statistic vs. constant model	Test statistic for the F-test on the regression model. It tests for a significant linear regression relationship between the response variable and the predictor variables.
p-value	$p$ -value for the F-test on the model. For example, the model is significant with a $p$ -value of $7.3816e-27$ .

You can request this display by using `disp`. For example, if you name your model `lm`, then you can display the outputs using `disp(lm)`.

**Perform analysis of variance (ANOVA) for the model.**

```
anova(lm, 'summary')
```

```
ans =
```

	SumSq	DF	MeanSq	F	pValue
Total	6004.8	92	65.269		
Model	4516	3	1505.3	89.987	7.3816e-27
Residual	1488.8	89	16.728		

This ANOVA display shows the following.

SumSq	Sum of squares for the regression model, Model, the error term, Residual, and the total, Total.
DF	Degrees of freedom for each term. Degrees of freedom is $n - 1$ for the total, $p - 1$ for the model, and $n - p$ for the error term, where $n$ is the number of observations, and $p$ is the number of coefficients in the model, including the intercept. For example, MPG data vector has six NaN values and one of the data vectors, Horsepower, has one NaN value for a different observation, so the total degrees of freedom is $93 - 1 = 92$ . There are four coefficients in the model, so the model DF is $4 - 1 = 3$ , and the DF for error term is $93 - 4 = 89$ .
MeanSq	Mean squared error for each term. Note that $\text{MeanSq} = \text{SumSq}/\text{DF}$ . For example, the mean squared error for the error term is $1488.8/89 = 16.728$ . The square root of this value is the root mean squared error in the linear regression display, or 4.09.
F	F-statistic value, which is the same as F-statistic vs. constant model in the linear regression display. In this example, it is 89.987, and in the linear regression display this F-statistic value is rounded up to 90.
pValue	$p$ -value for the F-test on the model. In this example, it is 7.3816e-27.

If there are higher-order terms in the regression model, `anova` partitions the model SumSq into the part explained by the higher-order terms and the rest. The corresponding F-statistics are for testing the significance of the linear terms and higher-order terms as separate groups.

If the data includes replicates, or multiple measurements at the same predictor values, then the `anova` partitions the error SumSq into the part for the replicates and the rest. The corresponding F-statistic is for testing the lack-of-fit by comparing the model residuals with the model-free variance estimate computed on the replicates.

See the anova method for details.

### Decompose ANOVA table for model terms.

```
anova(lm)
```

```
ans =
```

	SumSq	DF	MeanSq	F	pValue
x1	563.18	1	563.18	33.667	9.8742e-08
x2	52.187	1	52.187	3.1197	0.08078
x3	0.060046	1	0.060046	0.0035895	0.95236
Error	1488.8	89	16.728		

This anova display shows the following:

First column	Terms included in the model.
SumSq	Sum of squared error for each term except for the constant.
DF	Degrees of freedom. In this example, DF is 1 for each term in the model and $n - p$ for the error term, where $n$ is the number of observations, and $p$ is the number of coefficients in the model, including the intercept. For example, the DF for the error term in this model is $93 - 4 = 89$ . If any of the variables in the model is a categorical variable, the DF for that variable is the number of indicator variables created for its categories (number of categories $- 1$ ).
MeanSq	Mean squared error for each term. Note that $\text{MeanSq} = \text{SumSq}/\text{DF}$ . For example, the mean squared error for the error term is $1488.8/89 = 16.728$ .
F	F-values for each coefficient. The F-value is the ratio of the mean squared of each term and mean squared error, that is, $F = \text{MeanSq}(x_i)/\text{MeanSq}(\text{Error})$ . Each F-statistic has an F distribution, with the numerator degrees of freedom, DF value for the corresponding term, and the denominator degrees of freedom, $n - p$ . $n$ is the number of observations, and $p$ is the number of coefficients in the model. In this example, each F-statistic has an $F_{(1, 89)}$ distribution.
pValue	$p$ -value for each hypothesis test on the coefficient of the corresponding term in the linear model. For example, the $p$ -value for the F-statistic coefficient of $x_2$ is 0.08078, and is not significant at the 5% significance level given the other terms in the model.

**Display coefficient confidence intervals.**

```
coefCI(lm)
ans =
    40.2702    55.6833
   -0.0088   -0.0043
   -0.0913    0.0054
   -0.3957    0.3726
```

The values in each row are the lower and upper confidence limits, respectively, for the default 95% confidence intervals for the coefficients. For example, the first row shows the lower and upper limits, 40.2702 and 55.6833, for the intercept,  $\beta_0$ . Likewise, the second row shows the limits for  $\beta_1$  and so on. Confidence intervals provide a measure of precision for linear regression coefficient estimates. A 100(1- $\alpha$ )% confidence interval gives the range the corresponding regression coefficient will be in with 100(1- $\alpha$ )% confidence.

You can also change the confidence level. Find the 99% confidence intervals for the coefficients.

```
coefCI(lm, 0.01)
ans =
    37.7677    58.1858
   -0.0095   -0.0036
   -0.1069    0.0211
   -0.5205    0.4973
```

**Perform hypothesis test on coefficients.**

Test the null hypothesis that all predictor variable coefficients are equal to zero versus the alternate hypothesis that at least one of them is different from zero.

```
[p,F,d] = coefTest(lm)
p =
    7.3816e-27

F =
    89.9874
```

```
d =  
  
    3
```

Here, `coefTest` performs an F-test for the hypothesis that all regression coefficients (except for the intercept) are zero versus at least one differs from zero, which essentially is the hypothesis on the model. It returns `p`, the  $p$ -value, `F`, the F-statistic, and `d`, the numerator degrees of freedom. The F-statistic and  $p$ -value are the same as the ones in the linear regression display and ANOVA for the model. The degrees of freedom is  $4 - 1 = 3$  because there are four predictors (including the intercept) in the model.

Now, perform a hypothesis test on the coefficients of the first and second predictor variables.

```
H = [0 1 0 0; 0 0 1 0];  
[p,F,d] = coefTest(lm,H)
```

```
p =  
  
    5.1702e-23
```

```
F =  
  
    96.4873
```

```
d =  
  
    2
```

The numerator degrees of freedom is the number of coefficients tested, which is 2 in this example. The results indicate that at least one of  $\beta_2$  and  $\beta_3$  differs from zero.

## See Also

`LinearModel` | `anova` | `fitlm` | `stepwiselm`

## **Related Examples**

- “Examine Quality and Adjust the Fitted Model” on page 11-20

## **More About**

- “Coefficient Standard Errors and Confidence Intervals” on page 11-75
- “Coefficient of Determination (R-Squared)” on page 11-79
- “F-statistic and t-statistic” on page 11-95
- “Summary of Output and Diagnostic Statistics” on page 11-115

## Cook's Distance

### Purpose

Cook's distance is useful for identifying outliers in the  $X$  values (observations for predictor variables). It also shows the influence of each observation on the fitted response values. An observation with Cook's distance larger than three times the mean Cook's distance might be an outlier.

### Definition

Cook's distance is the scaled change in fitted values. Each element in `CooksDistance` is the normalized change in the vector of coefficients due to the deletion of an observation. The Cook's distance,  $D_i$ , of observation  $i$  is

$$D_i = \frac{\sum_{j=1}^n (\hat{y}_j - \hat{y}_{j(i)})^2}{p \text{MSE}},$$

where

- $\hat{y}_j$  is the  $j$ th fitted response value.
- $\hat{y}_{j(i)}$  is the  $j$ th fitted response value, where the fit does not include observation  $i$ .
- $\text{MSE}$  is the mean squared error.
- $p$  is the number of coefficients in the regression model.

Cook's distance is algebraically equivalent to the following expression:

$$D_i = \frac{r_i^2}{p \text{MSE}} \left( \frac{h_{ii}}{(1 - h_{ii})^2} \right),$$

where  $r_i$  is the  $i$ th residual, and  $h_{ii}$  is the  $i$ th leverage value.

`CooksDistance` is an  $n$ -by-1 column vector in the `Diagnostics` table of the `LinearModel` object.

## How To

After obtaining a fitted model, say, `mdl`, using `fitlm` or `stepwiselm`, you can:

- Display the Cook's distance values by indexing into the property using dot notation,

```
mdl.Diagnostics.CooksDistance
```

- Plot the Cook's distance values using

```
plotDiagnostics(mdl, 'cookd')
```

For details, see the `plotDiagnostics` method of the `LinearModel` class.

## Determine Outliers Using Cook's Distance

This example shows how to use Cook's Distance to determine the outliers in the data.

Load the sample data and define the independent and response variables.

```
load hospital
X = double(hospital(:,2:5));
y = hospital.BloodPressure(:,1);
```

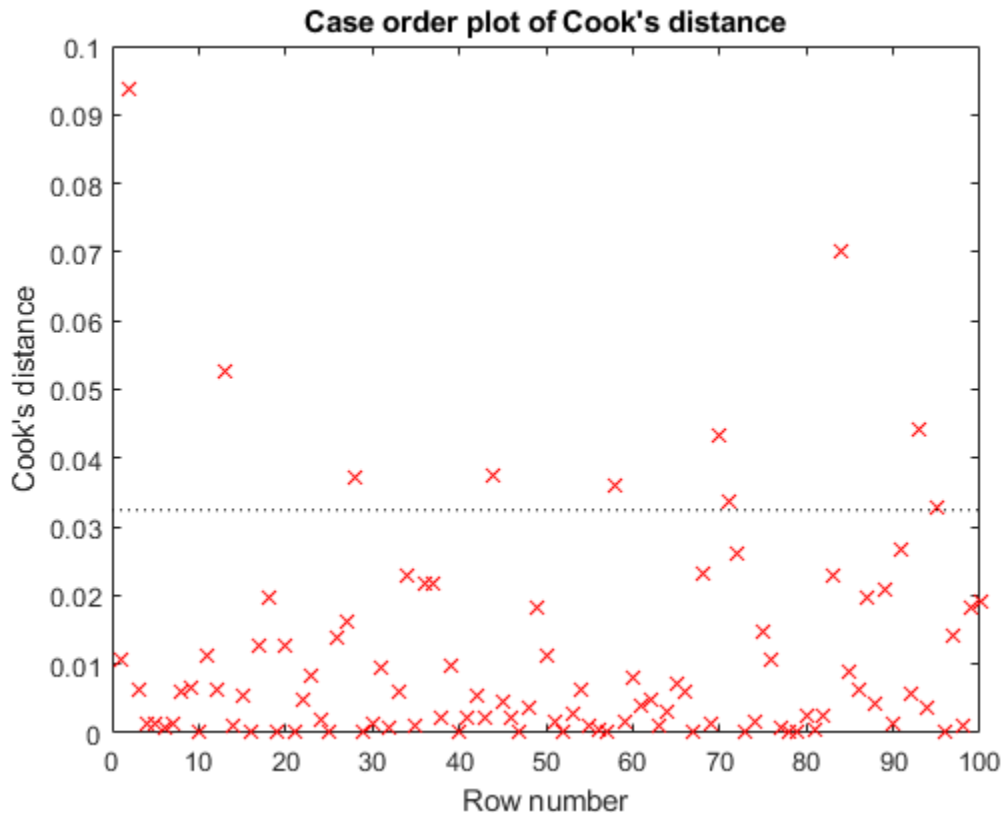
Fit the linear regression model.

```
mdl = fitlm(X,y);
```

Plot the Cook's distance values.

```
plotDiagnostics(mdl, 'cookd')
```





The dashed line in the figure corresponds to the recommended threshold value,  $3 \cdot \text{mean}(\text{mdl.Diagnostics.CooksDistance})$ . The plot has some observations with Cook's distance values greater than the threshold value, which for this example is  $3 \cdot (0.0108) = 0.0324$ . In particular, there are two Cook's distance values that are relatively higher than the others, which exceed the threshold value. You might want to find and omit these from your data and rebuild your model.

Find the observations with Cook's distance values that exceed the threshold value.

```
find((mdl.Diagnostics.CooksDistance) > 3 * mean(mdl.Diagnostics.CooksDistance))
```

```
ans =
```

```
2  
13  
28  
44  
58  
70  
71  
84  
93  
95
```

Find the observations with Cook's distance values that are relatively larger than the other observations with Cook's distances exceeding the threshold value.

```
find((mdl.Diagnostics.CooksDistance)>5*mean(mdl.Diagnostics.CooksDistance))
```

```
ans =
```

```
2  
84
```

## References

[1] Neter, J., M. H. Kutner, C. J. Nachtsheim, and W. Wasserman. *Applied Linear Statistical Models*. 4th ed. Chicago: Irwin, 1996.

## See Also

`LinearModel` | `fitlm` | `plotDiagnostics` | `stepwiselm`

## Related Examples

- “Examine Quality and Adjust the Fitted Model” on page 11-20
- “Interpret Linear Regression Results” on page 11-64

## Coefficient Standard Errors and Confidence Intervals

### In this section...

“Coefficient Covariance and Standard Errors” on page 11-75

“Compute Coefficient Covariance and Standard Errors” on page 11-75

“Coefficient Confidence Intervals” on page 11-76

“Compute Coefficient Confidence Intervals” on page 11-77

### Coefficient Covariance and Standard Errors

#### Purpose

Estimated coefficient variances and covariances capture the precision of regression coefficient estimates. The coefficient variances and their square root, the standard errors, are useful in testing hypotheses for coefficients.

#### Definition

The estimated covariance matrix is

$$\Sigma = MSE(XX)^{-1},$$

where  $MSE$  is the mean squared error, and  $X$  is the matrix of observations on the predictor variables. `CoefficientCovariance`, a property of the fitted model, is a  $p$ -by- $p$  covariance matrix of regression coefficient estimates.  $p$  is the number of coefficients in the regression model. The diagonal elements are the variances of the individual coefficients.

#### How To

After obtaining a fitted model, say, `mdl`, using `fitlm` or `stepwiselm`, you can display the coefficient covariances using

```
mdl.CoefficientCovariance
```

### Compute Coefficient Covariance and Standard Errors

This example shows how to compute the covariance matrix and standard errors of the coefficients.

Load the sample data and define the predictor and response variables.

```
load hospital
y = hospital.BloodPressure(:,1);
X = double(hospital(:,2:5));
```

Fit a linear regression model.

```
mdl = fitlm(X,y);
```

Display the coefficient covariance matrix.

```
CM = mdl.CoefficientCovariance
```

```
CM =
```

```
27.5113    11.0027   -0.1542   -0.2444    0.2702
11.0027     8.6864    0.0021   -0.1547   -0.0838
-0.1542     0.0021    0.0045   -0.0001   -0.0029
-0.2444   -0.1547   -0.0001    0.0031   -0.0026
 0.2702   -0.0838   -0.0029   -0.0026    1.0829
```

Compute the coefficient standard errors.

```
SE = diag(sqrt(CM))
```

```
SE =
```

```
5.2451
2.9473
0.0673
0.0557
1.0406
```

## Coefficient Confidence Intervals

### Purpose

The coefficient confidence intervals provide a measure of precision for linear regression coefficient estimates. A  $100(1-\alpha)\%$  confidence interval gives the range that the corresponding regression coefficient will be in with  $100(1-\alpha)\%$  confidence.

**Definition**

The  $100 \cdot (1 - \alpha)\%$  confidence intervals for linear regression coefficients are

$$b_i \pm t_{(1-\alpha/2, n-p)} SE(b_i),$$

where  $b_i$  is the coefficient estimate,  $SE(b_i)$  is the standard error of the coefficient estimate, and  $t_{(1-\alpha/2, n-p)}$  is the  $100(1-\alpha/2)$  percentile of  $t$ -distribution with  $n - p$  degrees of freedom.  $n$  is the number of observations and  $p$  is the number of regression coefficients.

**How To**

After obtaining a fitted model, say, `mdl`, using `fitlm` or `stepwiselm`, you can obtain the default 95% confidence intervals for coefficients using

```
coefCI (mdl)
```

You can also change the confidence level using

```
coefCI (mdl, alpha)
```

For details, see the `coefCI` and `coefTest` methods of `LinearModel` class.

**Compute Coefficient Confidence Intervals**

This example shows how to compute coefficient confidence intervals.

Load the sample data and fit a linear regression model.

```
load hald
mdl = fitlm(ingredients, heat);
```

Display the 95% coefficient confidence intervals.

```
coefCI (mdl)
```

```
ans =
```

```
-99.1786  223.9893
 -0.1663   3.2685
 -1.1589   2.1792
 -1.6385   1.8423
```

```
-1.7791    1.4910
```

The values in each row are the lower and upper confidence limits, respectively, for the default 95% confidence intervals for the coefficients. For example, the first row shows the lower and upper limits, -99.1786 and 223.9893, for the intercept,  $\beta_0$ . Likewise, the second row shows the limits for  $\beta_1$  and so on.

Display the 90% confidence intervals for the coefficients ( $\alpha = 0.1$ ).

```
coefCI mdl, 0.1)
```

```
ans =
```

```
-67.8949    192.7057  
  0.1662     2.9360  
-0.8358     1.8561  
-1.3015     1.5053  
-1.4626     1.1745
```

The confidence interval limits become narrower as the confidence level decreases.

## See Also

LinearModel | anova | coefCI | coefTest | fitlm | plotDiagnostics |  
stepwiselm

## Related Examples

- “Examine Quality and Adjust the Fitted Model” on page 11-20
- “Interpret Linear Regression Results” on page 11-64

## Coefficient of Determination (R-Squared)

### Purpose

Coefficient of determination (R-squared) indicates the proportionate amount of variation in the response variable  $y$  explained by the independent variables  $X$  in the linear regression model. The larger the R-squared is, the more variability is explained by the linear regression model.

### Definition

R-squared is the proportion of the total sum of squares explained by the model. Rsquared, a property of the fitted model, is a structure with two fields:

- Ordinary — Ordinary (unadjusted) R-squared

$$R^2 = \frac{SSR}{SST} = 1 - \frac{SSE}{SST}.$$

- Adjusted — R-squared adjusted for the number of coefficients

$$R_{adj}^2 = 1 - \left( \frac{n-1}{n-p} \right) \frac{SSE}{SST}.$$

$SSE$  is the sum of squared error,  $SSR$  is the sum of squared regression,  $SST$  is the sum of squared total,  $n$  is the number of observations, and  $p$  is the number of regression coefficients. Note that  $p$  includes the intercept, so for example,  $p$  is 2 for a linear fit. Because R-squared increases with added predictor variables in the regression model, the adjusted R-squared adjusts for the number of predictor variables in the model. This makes it more useful for comparing models with a different number of predictors.

### How To

After obtaining a fitted model, say, `mdl`, using `fitlm` or `stepwiselm`, you can obtain either R-squared value as a scalar by indexing into the property using dot notation, for example,

```
mdl.Rsquared.Ordinary
mdl.Rsquared.Adjusted
```

You can also obtain the SSE, SSR, and SST using the properties with the same name.

```
mdl.SSE
mdl.SSR
mdl.SST
```

## Display Coefficient of Determination

This example shows how to display R-squared (coefficient of determination) and adjusted R-squared. Load the sample data and define the response and independent variables.

```
load hospital
y = hospital.BloodPressure(:,1);
X = double(hospital(:,2:5));
```

Fit a linear regression model.

```
mdl = fitlm(X,y)
```

```
mdl =
```

```
Linear regression model:
  y ~ 1 + x1 + x2 + x3 + x4
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	117.4	5.2451	22.383	1.1667e-39
x1	0.88162	2.9473	0.29913	0.76549
x2	0.08602	0.06731	1.278	0.20438
x3	-0.016685	0.055714	-0.29947	0.76524
x4	9.884	1.0406	9.498	1.9546e-15

```
Number of observations: 100, Error degrees of freedom: 95
Root Mean Squared Error: 4.81
R-squared: 0.508, Adjusted R-Squared 0.487
F-statistic vs. constant model: 24.5, p-value = 5.99e-14
```

The R-squared and adjusted R-squared values are 0.508 and 0.487, respectively. Model explains about 50% of the variability in the response variable.



Access the R-squared and adjusted R-squared values using the property of the fitted `LinearModel` object.

```
mdl.Rsquared.Ordinary
```

```
ans =
```

```
    0.5078
```

```
mdl.Rsquared.Adjusted
```

```
ans =
```

```
    0.4871
```

The adjusted R-squared value is smaller than the ordinary R-squared value.

## See Also

`LinearModel` | `anova` | `fitlm` | `stepwiselm`

## Related Examples

- “Examine Quality and Adjust the Fitted Model” on page 11-20
- “Interpret Linear Regression Results” on page 11-64

## Delete-1 Statistics

### In this section...

“Delete-1 Change in Covariance (*covratio*)” on page 11-82

“Determine Influential Observations Using *CovRatio*” on page 11-83

“Delete-1 Scaled Difference in Coefficient Estimates (*Dfbetas*)” on page 11-85

“Determine Observations Influential on Coefficients Using *Dfbetas*” on page 11-86

“Delete-1 Scaled Change in Fitted Values (*Dffits*)” on page 11-86

“Determine Observations Influential on Fitted Response Using *Dffits*” on page 11-87

“Delete-1 Variance (*S2\_i*)” on page 11-89

“Compute and Examine Delete-1 Variance Values” on page 11-90

### Delete-1 Change in Covariance (*covratio*)

#### Purpose

Delete-1 change in covariance (*covratio*) identifies the observations that are influential in the regression fit. An influential observation is one where its exclusion from the model might significantly alter the regression function. Values of *covratio* larger than  $1 + 3p/n$  or smaller than  $1 - 3p/n$  indicate influential points, where  $p$  is the number of regression coefficients, and  $n$  is the number of observations.

#### Definition

The *covratio* statistic is the ratio of the determinant of the coefficient covariance matrix with observation  $i$  deleted to the determinant of the covariance matrix for the full model:

$$\text{covratio} = \frac{\det\{MSE(i)[X'(i)X(i)]^{-1}\}}{\det[MSE(X'X)^{-1]}}$$

*CovRatio* is an  $n$ -by-1 vector in the `Diagnostics` table of the fitted `LinearModel` object. Each element is the ratio of the generalized variance of the estimated coefficients when the corresponding element is deleted to the generalized variance of the coefficients using all the data.

## How To

After obtaining a fitted model, say, `mdl`, using `fitlm` or `stepwiselm`, you can:

- Display the `CovRatio` by indexing into the property using dot notation

```
mdl.Diagnostics.CovRatio
```

- Plot the delete-1 change in covariance using

```
plotDiagnostics(mdl, 'CovRatio')
```

For details, see the `plotDiagnostics` method of the `LinearModel` class.

## Determine Influential Observations Using CovRatio

This example shows how to use the `CovRatio` statistics to determine the influential points in data. Load the sample data and define the response and predictor variables.

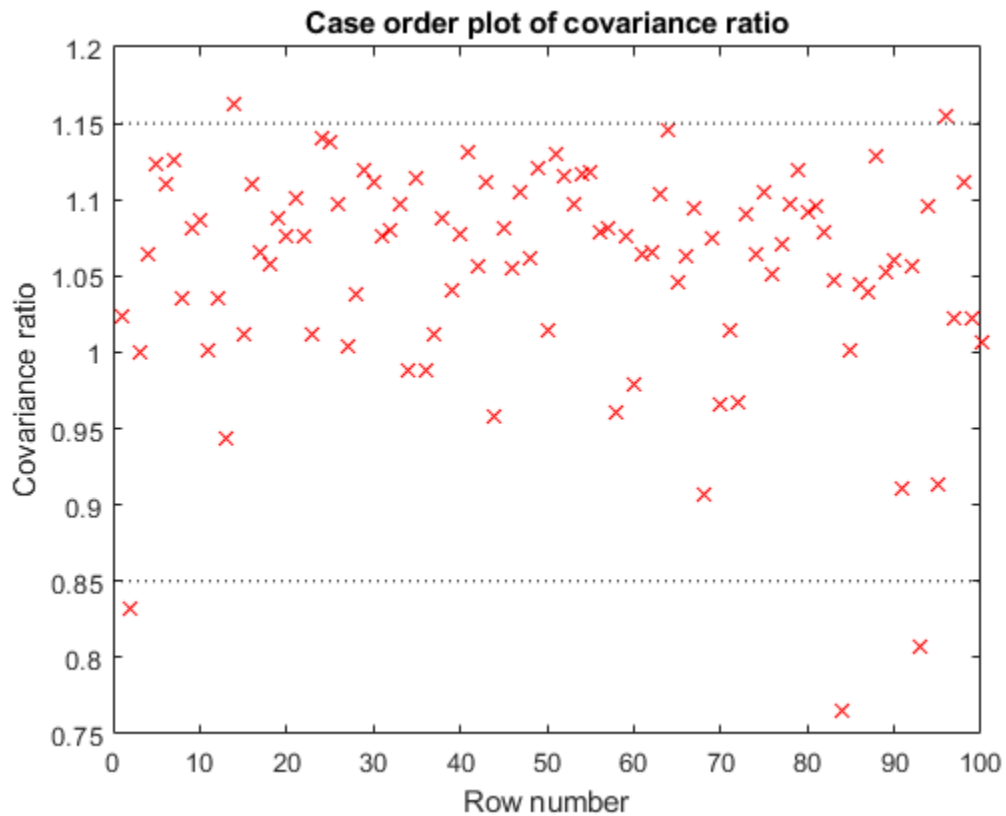
```
load hospital
y = hospital.BloodPressure(:,1);
X = double(hospital(:,2:5));
```

Fit a linear regression model.

```
mdl = fitlm(X,y);
```

Plot the `CovRatio` statistics.

```
plotDiagnostics(mdl, 'CovRatio')
```



For this example, the threshold limits are  $1 + 3*5/100 = 1.15$  and  $1 - 3*5/100 = 0.85$ . There are a few points beyond the limits, which might be influential points.

Find the observations that are beyond the limits.

```
find((mdl.Diagnostics.CovRatio)>1.15|(mdl.Diagnostics.CovRatio)<0.85)
```

```
ans =
```

```
2
14
84
93
```

## Delete-1 Scaled Difference in Coefficient Estimates (Dfbetas)

### Purpose

The sign of a delete-1 scaled difference in coefficient estimate (Dfbetas) for coefficient  $j$  and observation  $i$  indicates whether that observation causes an increase or decrease in the estimate of the regression coefficient. The absolute value of a Dfbetas indicates the magnitude of the difference relative to the estimated standard deviation of the regression coefficient. A Dfbetas value larger than  $3/\sqrt{n}$  in absolute value indicates that the observation has a large influence on the corresponding coefficient.

### Definition

Dfbetas for coefficient  $j$  and observation  $i$  is the ratio of the difference in the estimate of coefficient  $j$  using all observations and the one obtained by removing observation  $i$ , and the standard error of the coefficient estimate obtained by removing observation  $i$ . The Dfbetas for coefficient  $j$  and observation  $i$  is

$$Dfbetas_{ij} = \frac{b_j - b_{j(i)}}{\sqrt{MSE_{(i)}(1 - h_{ii})}},$$

where  $b_j$  is the estimate for coefficient  $j$ ,  $b_{j(i)}$  is the estimate for coefficient  $j$  by removing observation  $i$ ,  $MSE_{(i)}$  is the mean squared error of the regression fit by removing observation  $i$ , and  $h_{ii}$  is the leverage value for observation  $i$ . Dfbetas is an  $n$ -by- $p$  matrix in the `Diagnostics` table of the fitted `LinearModel` object. Each cell of Dfbetas corresponds to the Dfbetas value for the corresponding coefficient obtained by removing the corresponding observation.

### How To

After obtaining a fitted model, say, `mdl`, using `fitlm` or `stepwiselm`, you can obtain the Dfbetas values as an  $n$ -by- $p$  matrix by indexing into the property using dot notation,

```
mdl.Diagnostics.Dfbetas
```

## Determine Observations Influential on Coefficients Using Dfbetas

This example shows how to determine the observations that have large influence on coefficients using `Dfbetas`. Load the sample data and define the response and independent variables.

```
load hospital
y = hospital.BloodPressure(:,1);
X = double(hospital(:,2:5));
```

Fit a linear regression model.

```
mdl = fitlm(X,y);
```

Find the `Dfbetas` values that are high in absolute value.

```
[row,col] = find(abs(mdl.Diagnostics.Dfbetas)>3/sqrt(100));
disp([row col])
```

```
     2     1
    28     1
    84     1
    93     1
     2     2
    13     3
    84     3
     2     4
    84     4
```

## Delete-1 Scaled Change in Fitted Values (Dffits)

### Purpose

The delete-1 scaled change in fitted values (`Dffits`) show the influence of each observation on the fitted response values. `Dffits` values with an absolute value larger than  $2\sqrt{p/n}$  might be influential.

### Definition

`Dffits` for observation  $i$  is

$$Dffits_i = sr_i \sqrt{\frac{h_{ii}}{1-h_{ii}}},$$

where  $sr_i$  is the studentized residual, and  $h_{ii}$  is the leverage value of the fitted `LinearModel` object. `Dffits` is an  $n$ -by-1 column vector in the `Diagnostics` table of the fitted `LinearModel` object. Each element in `Dffits` is the change in the fitted value caused by deleting the corresponding observation and scaling by the standard error.

## How To

After obtaining a fitted model, say, `mdl`, using `fitlm` or `stepwiselm`, you can:

- Display the `Dffits` values by indexing into the property using dot notation

```
mdl.Diagnostics.Dffits
```

- Plot the delete-1 scaled change in fitted values using

```
plotDiagnostics(mdl, 'Dffits')
```

For details, see the `plotDiagnostics` method of the `LinearModel` class for details.

## Determine Observations Influential on Fitted Response Using Dffits

This example shows how to determine the observations that are influential on the fitted response values using `Dffits` values. Load the sample data and define the response and independent variables.

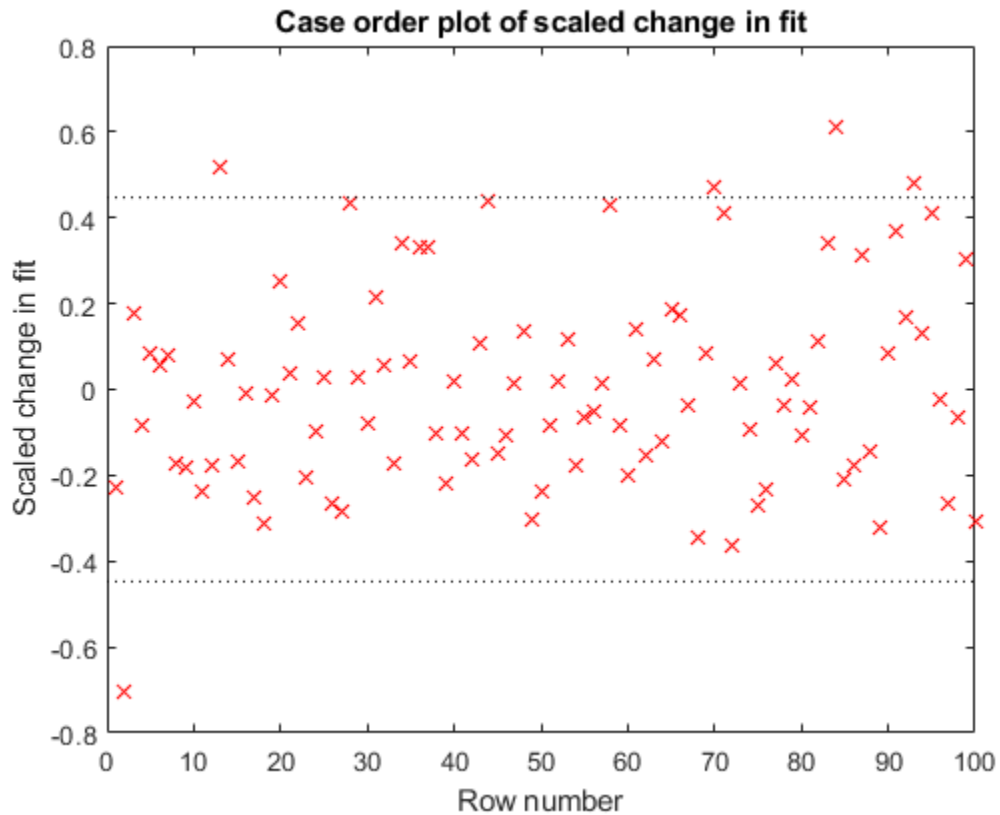
```
load hospital
y = hospital.BloodPressure(:,1);
X = double(hospital(:,2:5));
```

Fit a linear regression model.

```
mdl = fitlm(X,y);
```

Plot the `Dffits` values.

```
plotDiagnostics(mdl, 'Dffits')
```



The influential threshold limit for the absolute value of Dffits in this example is  $2*\sqrt{5/100} = 0.45$ . Again, there are some observations with Dffits values beyond the recommended limits.

Find the Dffits values that are large in absolute value.

```
find(abs mdl.Diagnostics.Dffits)>2*sqrt(4/100)
```

```
ans =
```

```
2
13
28
```



44  
58  
70  
71  
84  
93  
95

## Delete-1 Variance (S2\_i)

### Purpose

The delete-1 variance (S2\_i) shows how the mean squared error changes when an observation is removed from the data set. You can compare the S2\_i values with the value of the mean squared error.

### Definition

S2\_i is a set of residual variance estimates obtained by deleting each observation in turn. The S2\_i value for observation  $i$  is

$$S2\_i = MSE_{(i)} = \frac{\sum_{j \neq i}^n [y_j - y_{j(i)}]^2}{n - p - 1},$$

where  $y_j$  is the  $j$ th observed response value. S2\_i is an  $n$ -by-1 vector in the Diagnostics table of the fitted LinearModel object. Each element in S2\_i is the mean squared error of the regression obtained by deleting that observation.

### How To

After obtaining a fitted model, say, mdl, using fitlm or stepwiselm, you can:

- Display the S2\_i vector by indexing into the property using dot notation

```
mdl.Diagnostics.S2_i
```

- Plot the delete-1 variance values using

```
plotDiagnostics(mdl, 'S2_i')
```

For details, see the plotDiagnostics method of the LinearModel class.

## Compute and Examine Delete-1 Variance Values

This example shows how to compute and plot  $S^2_i$  values to examine the change in the mean squared error when an observation is removed from the data. Load the sample data and define the response and independent variables.

```
load hospital
y = hospital.BloodPressure(:,1);
X = double(hospital(:,2:5));
```

Fit a linear regression model.

```
mdl = fitlm(X,y);
```

Display the MSE value for the model.

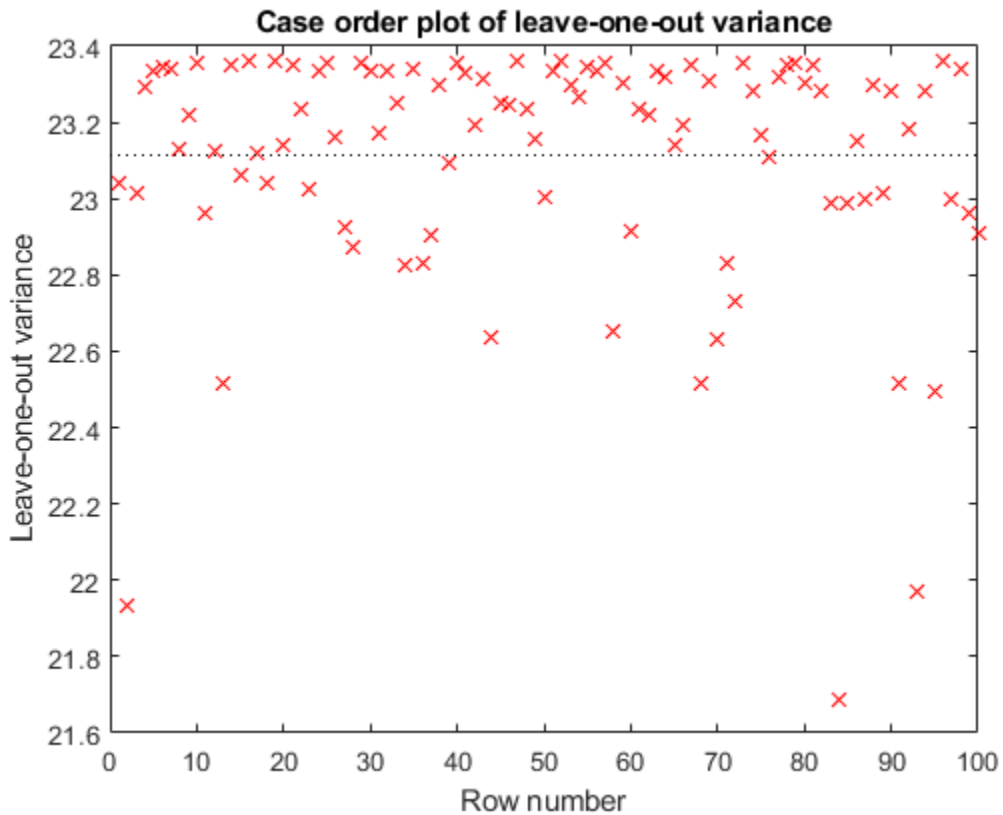
```
mdl.MSE
```

```
ans =
```

```
23.1140
```

Plot the  $S^2_i$  values.

```
plotDiagnostics(mdl, 'S2_i')
```



This plot makes it easy to compare the  $S^2_i$  values to the MSE value of 23.114, indicated by the horizontal dashed lines. You can see how deleting one observation changes the error variance.

## See Also

`LinearModel` | `fitlm` | `plotDiagnostics` | `plotResiduals` | `stepwiselm`

## Related Examples

- “Examine Quality and Adjust the Fitted Model” on page 11-20

- “Interpret Linear Regression Results” on page 11-64

## Durbin-Watson Test

### Purpose

The Durbin-Watson test assesses whether there is autocorrelation among the residuals or not.

### Definition

The Durbin-Watson test statistic,  $DW$ , is

$$DW = \frac{\sum_{i=1}^{n-1} (r_{i+1} - r_i)^2}{\sum_{i=1}^n r_i^2}.$$

Here,  $r_i$  is the  $i$ th raw residual, and  $n$  is the number of observations.

### How To

After obtaining a fitted model, say, `mdl`, using `fitlm` or `stepwiselm`, you can perform the Durbin-Watson test using

```
dwtest(mdl)
```

For details, see the `dwtest` method of the `LinearModel` class.

### Test for Autocorrelation Among Residuals

This example shows how to test for autocorrelation among the residuals of a linear regression model.

Load the sample data and fit a linear regression model.

```
load hald
mdl = fitlm(ingredients,heat);
```

Perform a two-sided Durbin-Watson test to determine if there is any autocorrelation among the residuals of the linear model, `mdl`.

```
[p,DW] = dwtest(mdl, 'exact', 'both')
```

```
p =
```

```
0.6285
```

```
DW =
```

```
2.0526
```

The value of the Durbin-Watson test statistic is 2.0526. The  $P$ -value of 0.6285 suggest that the residuals are not autocorrelated.

## See Also

[LinearModel](#) | [dwtest](#) | [fitlm](#) | [plotResiduals](#) | [stepwiselm](#)

## Related Examples

- “Examine Quality and Adjust the Fitted Model” on page 11-20
- “Interpret Linear Regression Results” on page 11-64

## F-statistic and t-statistic

In this section...
“F-statistic” on page 11-95
“Assess Fit of Model Using F-statistic” on page 11-96
“t-statistic” on page 11-98
“Assess Significance of Regression Coefficients Using t-statistic” on page 11-99

### F-statistic

#### Purpose

In linear regression, the F-statistic is the test statistic for the analysis of variance (ANOVA) approach to test the significance of the model or the components in the model.

#### Definition

The F-statistic in the linear model output display is the test statistic for testing the statistical significance of the model. The F-statistic values in the `anova` display are for assessing the significance of the terms or components in the model.

#### How To

After obtaining a fitted model, say, `mdl`, using `fitlm` or `stepwiselm`, you can:

- Find the F-statistic vs. constant model in the output display or by using

```
disp(mdl)
```

- Display the ANOVA for the model using

```
anova(mdl, 'summary')
```

- Obtain the F-statistic values for the components, except for the constant term using

```
anova(mdl)
```

For details, see the `anova` method of the `LinearModel` class.

## Assess Fit of Model Using F-statistic

This example shows how to use assess the fit of the model and the significance of the regression coefficients using F-statistic.

Load the sample data.

```
load hospital
tbl = table(hospital.Age, hospital.Weight, hospital.Smoker, hospital.BloodPressure(:,1), .
    'VariableNames', {'Age', 'Weight', 'Smoker', 'BloodPressure'});
tbl.Smoker = categorical(tbl.Smoker);
```

Fit a linear regression model.

```
mdl = fitlm(tbl, 'BloodPressure ~ Age*Weight + Smoker + Weight^2')
```

```
mdl =
```

```
Linear regression model:
```

```
BloodPressure ~ 1 + Smoker + Age*Weight + Weight^2
```

```
Estimated Coefficients:
```

	Estimate	SE	tStat	pValue
(Intercept)	168.02	27.694	6.067	2.7149e-08
Age	0.079569	0.39861	0.19962	0.84221
Weight	-0.69041	0.3435	-2.0099	0.047305
Smoker_true	9.8027	1.0256	9.5584	1.5969e-15
Age:Weight	0.00021796	0.0025258	0.086294	0.93142
Weight^2	0.0021877	0.0011037	1.9822	0.050375

```
Number of observations: 100, Error degrees of freedom: 94
```

```
Root Mean Squared Error: 4.73
```

```
R-squared: 0.528, Adjusted R-Squared 0.503
```

```
F-statistic vs. constant model: 21, p-value = 4.81e-14
```

The F-statistic of the linear fit versus the constant model is 168.02, with a  $p$ -value of 2.71e-08. The model is significant at the 5% significance level. The R-squared value of 0.528 means the model explains about 53% of the variability in the response. There might be other predictor (explanatory) variables that are not included in the current model.



Display the ANOVA table for the fitted model.

```
anova mdl, 'summary'
```

```
ans =
```

```
5x5 table
```

	SumSq	DF	MeanSq	F	pValue
	-----	---	-----	-----	-----
Total	4461.2	99	45.062		
Model	2354.5	5	470.9	21.012	4.8099e-14
. Linear	2263.3	3	754.42	33.663	7.2417e-15
. Nonlinear	91.248	2	45.624	2.0358	0.1363
Residual	2106.6	94	22.411		

This display separates the variability in the model into linear and nonlinear terms. Since there are two non-linear terms ( $\text{Weight}^2$  and the interaction between  $\text{Weight}$  and  $\text{Age}$ ), the nonlinear degrees of freedom in the  $\text{DF}$  column is 2. There are three linear terms in the model (one  $\text{Smoker}$  indicator variable,  $\text{Weight}$ , and  $\text{Age}$ ). The corresponding F-statistics in the  $F$  column are for testing the significance of the linear and nonlinear terms as separate groups.

When there are replicated observations, the residual term is also separated into two parts; first is the error due to the lack of fit, and second is the pure error independent from the model, obtained from the replicated observations. In that case, the F-statistic is for testing the lack of fit, that is, whether the fit is adequate or not. But, in this example, there are no replicated observations.

Display the ANOVA table for the model terms.

```
anova mdl
```

```
ans =
```

```
6x5 table
```

	SumSq	DF	MeanSq	F	pValue
	-----	---	-----	-----	-----

Age	62.991	1	62.991	2.8107	0.096959
Weight	0.064104	1	0.064104	0.0028604	0.95746
Smoker	2047.5	1	2047.5	91.363	1.5969e-15
Age:Weight	0.16689	1	0.16689	0.0074466	0.93142
Weight^2	88.057	1	88.057	3.9292	0.050375
Error	2106.6	94	22.411		

This display decomposes the ANOVA table into the model terms. The corresponding F-statistics in the F column are for assessing the statistical significance of each term. The F-test for `Cylinders` test whether the coefficient of the indicator variable for smoker is different from zero or not. That is, whether being a smoker has a significant effect on MPG or not. The degrees of freedom for each model term is the numerator degrees of freedom for the corresponding F-test. All of the terms have 1 degree of freedom. In case of a categorical variable, the degrees of freedom is the number of indicator variable. `Smoker` has only one indicator variable, so the degrees of freedom for that is also 1.

## t-statistic

### Purpose

In linear regression, the  $t$ -statistic is useful for making inferences about the regression coefficients. The hypothesis test on coefficient  $i$  tests the null hypothesis that it is equal to zero – meaning the corresponding term is not significant – versus the alternate hypothesis that the coefficient is different from zero.

### Definition

For a hypotheses test on coefficient  $i$ , with

$$H_0 : \beta_i = 0$$

$$H_1 : \beta_i \neq 0,$$

the  $t$ -statistic is:

$$t = \frac{b_i}{SE(b_i)},$$

where  $SE(b_i)$  is the standard error of the estimated coefficient  $b_i$ .

## How To

After obtaining a fitted model, say, `mdl`, using `fitlm` or `stepwiselm`, you can:

- Find the coefficient estimates, the standard errors of the estimates (SE), and the  $t$ -statistic values of hypothesis tests for the corresponding coefficients (`tStat`) in the output display.
- Call for the display using

```
display(mdl)
```

## Assess Significance of Regression Coefficients Using t-statistic

This example shows how to test for the significance of the regression coefficients using  $t$ -statistic.

Load the sample data and fit the linear regression model.

```
load hald
mdl = fitlm(ingredients,heat)
```

```
mdl =
```

```
Linear regression model:
y ~ 1 + x1 + x2 + x3 + x4
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	62.405	70.071	0.8906	0.39913
x1	1.5511	0.74477	2.0827	0.070822
x2	0.51017	0.72379	0.70486	0.5009
x3	0.10191	0.75471	0.13503	0.89592
x4	-0.14406	0.70905	-0.20317	0.84407

```
Number of observations: 13, Error degrees of freedom: 8
Root Mean Squared Error: 2.45
R-squared: 0.982, Adjusted R-Squared 0.974
F-statistic vs. constant model: 111, p-value = 4.76e-07
```

You can see that for each coefficient,  $tStat = Estimate/SE$ . The  $P$ -values for the hypotheses tests are in the `pValue` column. Each  $t$ -statistic tests for the significance of each term given other terms in the model. According to these results, none of the coefficients seem significant at the 5% significance level, although the R-squared value for the model is really high at 0.97. This often indicates possible multicollinearity among the predictor variables.

Use stepwise regression to decide which variables to include in the model.

```
load hald
mdl = stepwiselm(ingredients,heat)

1. Adding x4, FStat = 22.7985, pValue = 0.000576232
2. Adding x1, FStat = 108.2239, pValue = 1.105281e-06

mdl =

Linear regression model:
  y ~ 1 + x1 + x4

Estimated Coefficients:

```

	Estimate	SE	tStat	pValue
(Intercept)	103.1	2.124	48.54	3.3243e-13
x1	1.44	0.13842	10.403	1.1053e-06
x4	-0.61395	0.048645	-12.621	1.8149e-07

```

Number of observations: 13, Error degrees of freedom: 10
Root Mean Squared Error: 2.73
R-squared: 0.972, Adjusted R-Squared 0.967
F-statistic vs. constant model: 177, p-value = 1.58e-08

```

In this example, `stepwiselm` starts with the constant model (default) and uses forward selection to incrementally add `x4` and `x1`. Each predictor variable in the final model is significant given the other one is in the model. The algorithm stops when adding none of

the other predictor variables significantly improves in the model. For details on stepwise regression, see `stepwiselm`.

## See Also

`LinearModel` | `anova` | `coefCI` | `coefTest` | `fitlm` | `stepwiselm`

## Related Examples

- “Examine Quality and Adjust the Fitted Model” on page 11-20
- “Interpret Linear Regression Results” on page 11-64

## Hat Matrix and Leverage

In this section...
“Hat Matrix” on page 11-102
“Leverage” on page 11-103
“Determine High Leverage Observations” on page 11-104

### Hat Matrix

#### Purpose

The hat matrix provides a measure of leverage. It is useful for investigating whether one or more observations are outlying with regard to their  $X$  values, and therefore might be excessively influencing the regression results.

#### Definition

The hat matrix is also known as the *projection matrix* because it projects the vector of observations,  $y$ , onto the vector of predictions,  $\hat{y}$ , thus putting the "hat" on  $y$ . The hat matrix  $H$  is defined in terms of the data matrix  $X$ :

$$H = X(X^T X)^{-1} X^T$$

and determines the fitted or predicted values since

$$\hat{y} = Hy = Xb.$$

The diagonal elements of  $H$ ,  $h_{ii}$ , are called leverages and satisfy

$$0 \leq h_{ii} \leq 1$$

$$\sum_{i=1}^n h_{ii} = p,$$

where  $p$  is the number of coefficients, and  $n$  is the number of observations (rows of  $X$ ) in the regression model. `HatMatrix` is an  $n$ -by- $n$  matrix in the `Diagnostics` table.

#### How To

After obtaining a fitted model, say, `mdl`, using `fitlm` or `stepwiselm`, you can:

- Display the `HatMatrix` by indexing into the property using dot notation

```
mdl.Diagnostics.HatMatrix
```

When  $n$  is large, `HatMatrix` might be computationally expensive. In those cases, you can obtain the diagonal values directly, using

```
mdl.Diagnostics.Leverage
```

## Leverage

### Purpose

Leverage is a measure of the effect of a particular observation on the regression predictions due to the position of that observation in the space of the inputs. In general, the farther a point is from the center of the input space, the more leverage it has. Because the sum of the leverage values is  $p$ , an observation  $i$  can be considered as an outlier if its leverage substantially exceeds the mean leverage value,  $p/n$ , for example, a value larger than  $2^*p/n$ .

### Definition

The leverage of observation  $i$  is the value of the  $i$ th diagonal term,  $h_{ii}$ , of the hat matrix,  $H$ , where

$$H = X(X^T X)^{-1} X^T.$$

The diagonal terms satisfy

$$0 \leq h_{ii} \leq 1$$

$$\sum_{i=1}^n h_{ii} = p,$$

where  $p$  is the number of coefficients in the regression model, and  $n$  is the number of observations. The minimum value of  $h_{ii}$  is  $1/n$  for a model with a constant term. If the fitted model goes through the origin, then the minimum leverage value is 0 for an observation at  $x = 0$ .

It is possible to express the fitted values,  $\hat{y}$ , by the observed values,  $y$ , since  $\hat{y} = Hy = Xb$ .

Hence,  $h_{ii}$  expresses how much the observation  $y_i$  has impact on  $\hat{y}_i$ . A large value of  $h_{ii}$  indicates that the  $i$ th case is distant from the center of all X values for all  $n$  cases and has more leverage. Leverage is an  $n$ -by-1 column vector in the Diagnostics table.

### How To

After obtaining a fitted model, say, `mdl`, using `fitlm` or `stepwiselm`, you can:

- Display the Leverage vector by indexing into the property using dot notation

```
mdl.Diagnostics.Leverage
```

- Plot the leverage for the values fitted by your model using

```
plotDiagnostics(mdl)
```

See the `plotDiagnostics` method of the `LinearModel` class for details.

## Determine High Leverage Observations

This example shows how to compute Leverage values and assess high leverage observations. Load the sample data and define the response and independent variables.

```
load hospital
y = hospital.BloodPressure(:,1);
X = double(hospital(:,2:5));
```

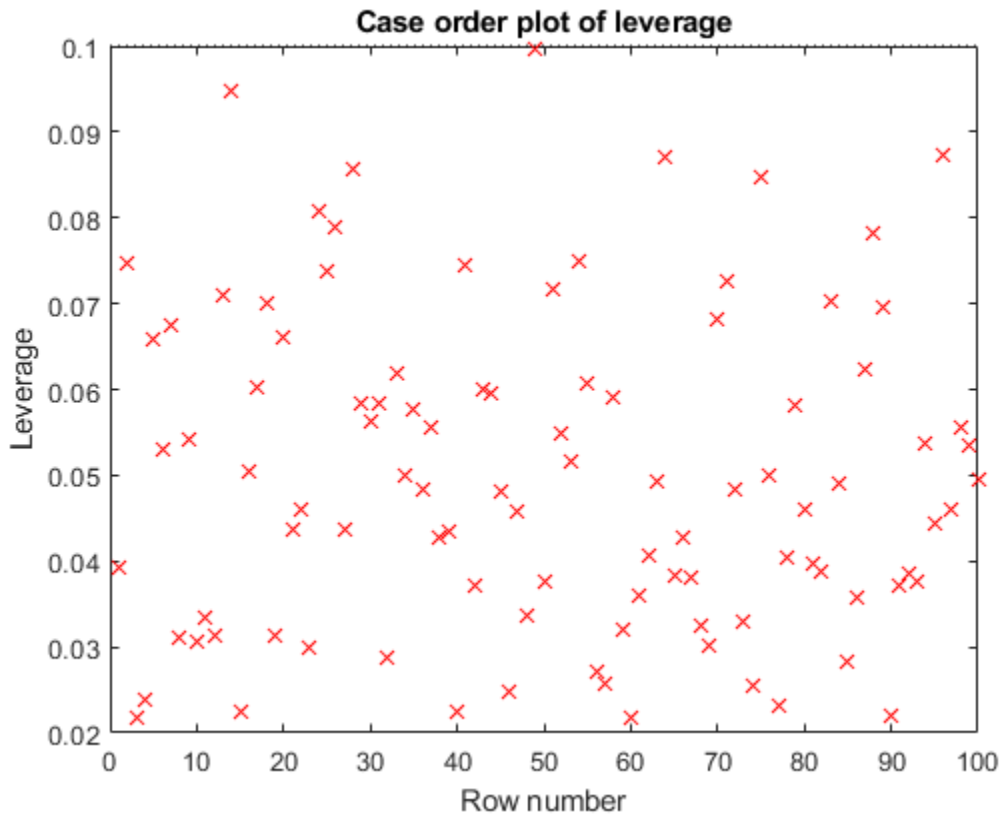
Fit a linear regression model.

```
mdl = fitlm(X,y);
```

Plot the leverage values.

```
plotDiagnostics(mdl)
```





For this example, the recommended threshold value is  $2*5/100 = 0.1$ . There is no indication of high leverage observations.

## See Also

`LinearModel` | `fitlm` | `plotDiagnostics` | `stepwiselm`

## Related Examples

- “Examine Quality and Adjust the Fitted Model” on page 11-20
- “Interpret Linear Regression Results” on page 11-64

## Residuals

### Purpose

Residuals are useful for detecting outlying  $y$  values and checking the linear regression assumptions with respect to the error term in the regression model. High-leverage observations have smaller residuals because they often shift the regression line or surface closer to them. You can also use residuals to detect some forms of heteroscedasticity and autocorrelation.

### Definition

The `Residuals` matrix is an  $n$ -by-4 table containing four types of residuals, with one row for each observation.

### Raw Residuals

Observed minus fitted values, that is,

$$r_i = y_i - \hat{y}_i.$$

### Pearson Residuals

Raw residuals divided by the root mean squared error, that is,

$$pr_i = \frac{r_i}{\sqrt{MSE}},$$

where  $r_i$  is the raw residual and  $MSE$  is the mean squared error.

### Standardized Residuals

Standardized residuals are raw residuals divided by their estimated standard deviation. The standardized residual for observation  $i$  is

$$st_i = \frac{r_i}{\sqrt{MSE(1 - h_{ii})}},$$

where  $MSE$  is the mean squared error and  $h_{ii}$  is the leverage value for observation  $i$ .

## Studentized Residuals

Studentized residuals are the raw residuals divided by an independent estimate of the residual standard deviation. The residual for observation  $i$  is divided by an estimate of the error standard deviation based on all observations except for observation  $i$ .

$$sr_i = \frac{r_i}{\sqrt{MSE_{(i)}(1-h_{ii})}},$$

where  $MSE_{(i)}$  is the mean squared error of the regression fit calculated by removing observation  $i$ , and  $h_{ii}$  is the leverage value for observation  $i$ . The studentized residual  $sr_i$  has a  $t$ -distribution with  $n - p - 1$  degrees of freedom.

## How To

After obtaining a fitted model, say, `mdl`, using `fitlm` or `stepwiselm`, you can:

- Find the `Residuals` table under `mdl` object.
- Obtain any of these columns as a vector by indexing into the property using dot notation, for example,

```
mdl.Residuals.Raw
```

- Plot any of the residuals for the values fitted by your model using

```
plotResiduals(mdl)
```

For details, see the `plotResiduals` method of the `LinearModel` class.

## Assess Model Assumptions Using Residuals

This example shows how to assess the model assumptions by examining the residuals of a fitted linear regression model.

Load the sample data and store the independent and response variables in a table.

```
load imports-85
tbl = table(X(:,7),X(:,8),X(:,9),X(:,15), 'VariableNames', ...
{'curb_weight', 'engine_size', 'bore', 'price'});
```

Fit a linear regression model.

```
mdl = fitlm(tbl)
```

```
mdl =
```

```
Linear regression model:
```

```
price ~ 1 + curb_weight + engine_size + bore
```

```
Estimated Coefficients:
```

	Estimate	SE	tStat	pValue
	-----	-----	-----	-----
(Intercept)	64.095	3.703	17.309	2.0481e-41
curb_weight	-0.0086681	0.0011025	-7.8623	2.42e-13
engine_size	-0.015806	0.013255	-1.1925	0.23452
bore	-2.6998	1.3489	-2.0015	0.046711

```
Number of observations: 201, Error degrees of freedom: 197
```

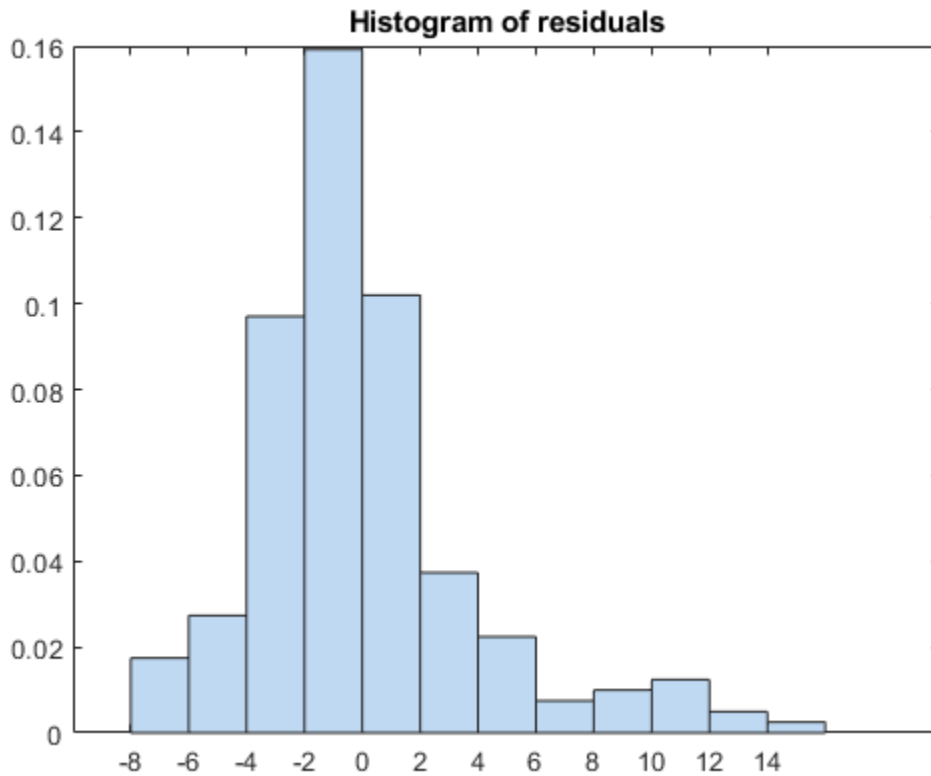
```
Root Mean Squared Error: 3.95
```

```
R-squared: 0.674, Adjusted R-Squared 0.669
```

```
F-statistic vs. constant model: 136, p-value = 1.14e-47
```

**Plot the histogram of raw residuals.**

```
plotResiduals(mdl)
```



The histogram shows that the residuals are slightly right skewed.

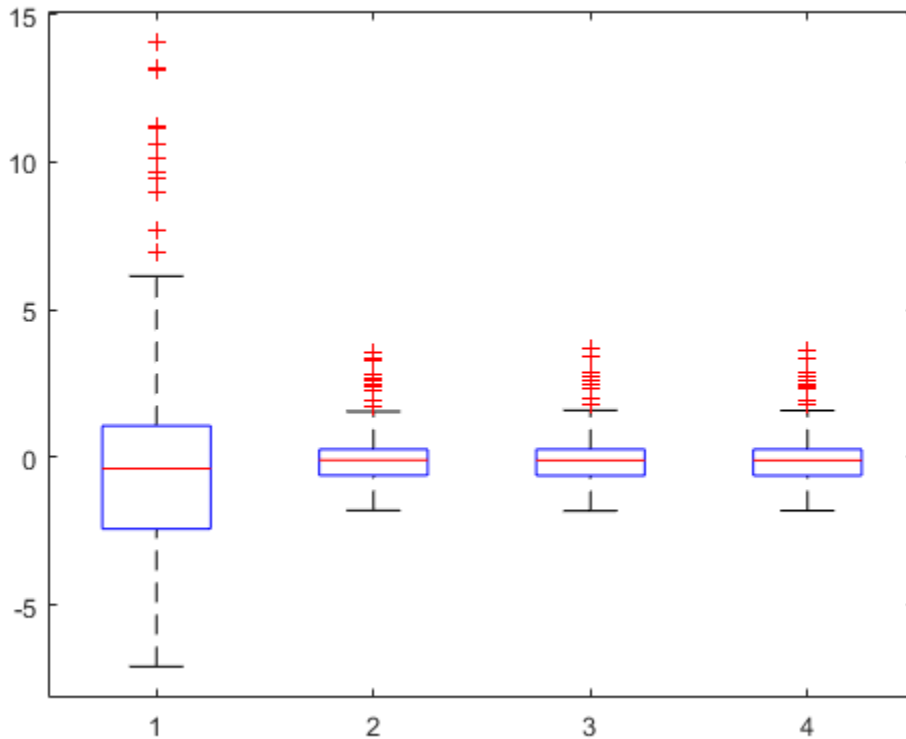
Plot the box plot of all four types of residuals.

```
Res = table2array mdl.Residuals;
```

You can see the right-skewed structure of the residuals in the box plot as well.

Plot the normal probability plot of the raw residuals.

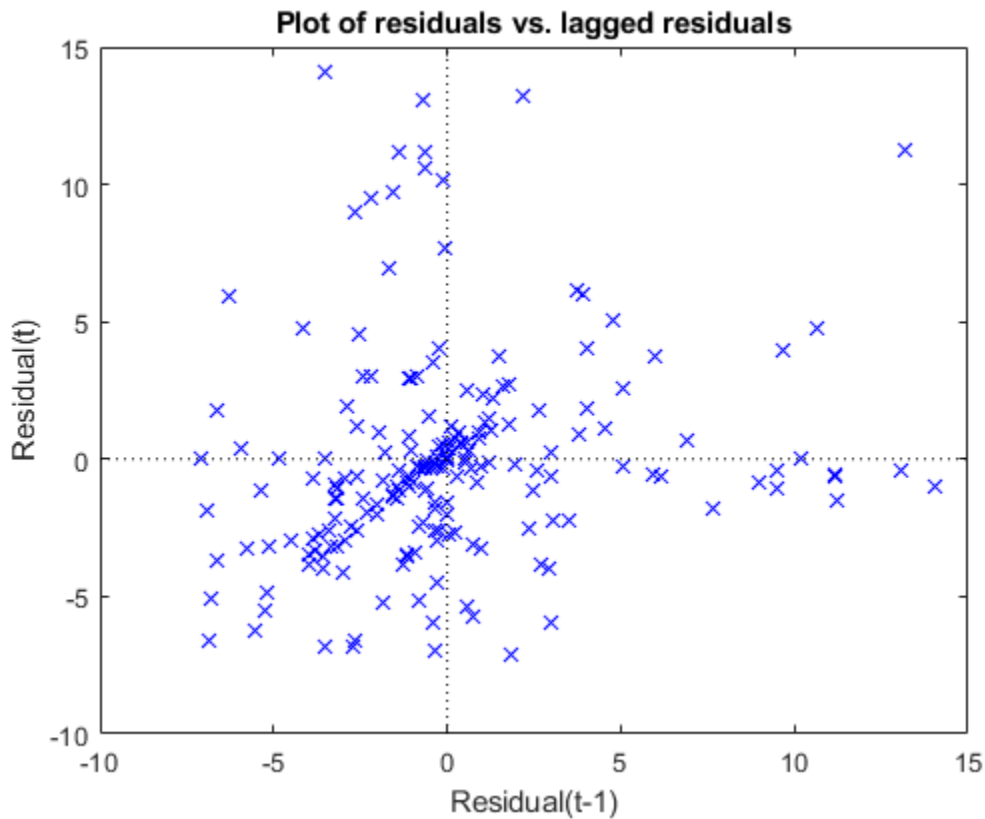
```
plotResiduals(mdl, 'probability')  
boxplot(Res)
```



This normal probability plot also shows the deviation from normality and the skewness on the right tail of the distribution of residuals.

Plot the residuals versus lagged residuals.

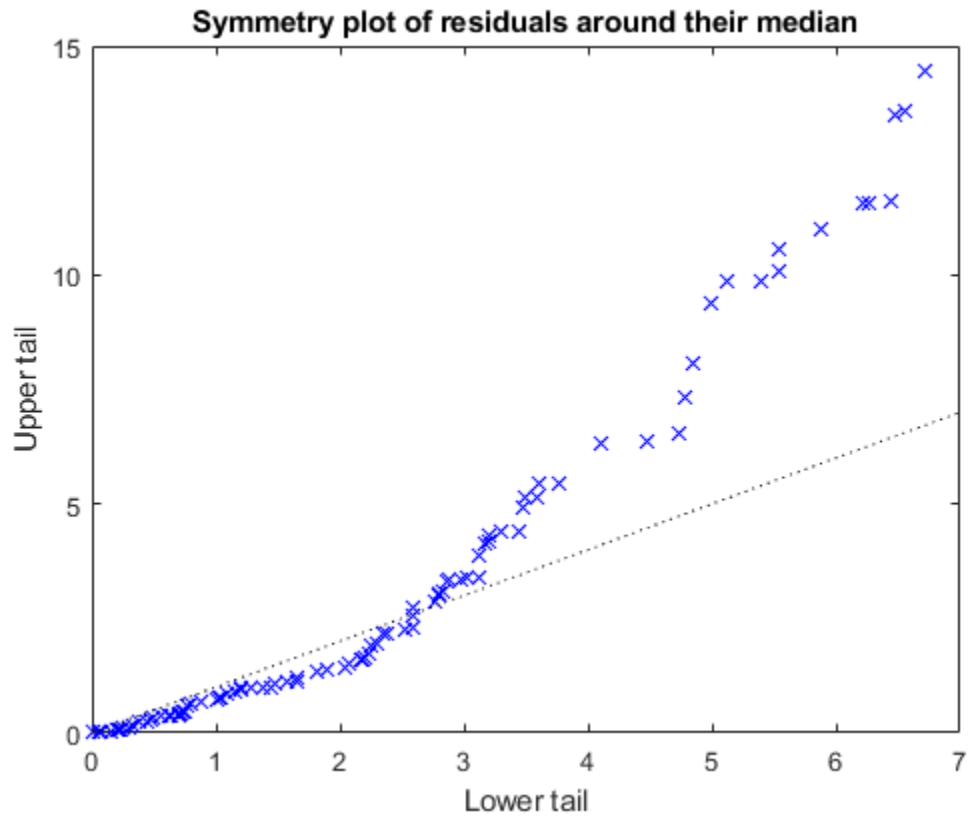
```
plotResiduals(mdl, 'lagged')
```



This graph shows a trend, which indicates a possible correlation among the residuals. You can further check this using `dwtest(mdl)`. Serial correlation among residuals usually means that the model can be improved.

Plot the symmetry plot of residuals.

```
plotResiduals(mdl, 'symmetry')
```

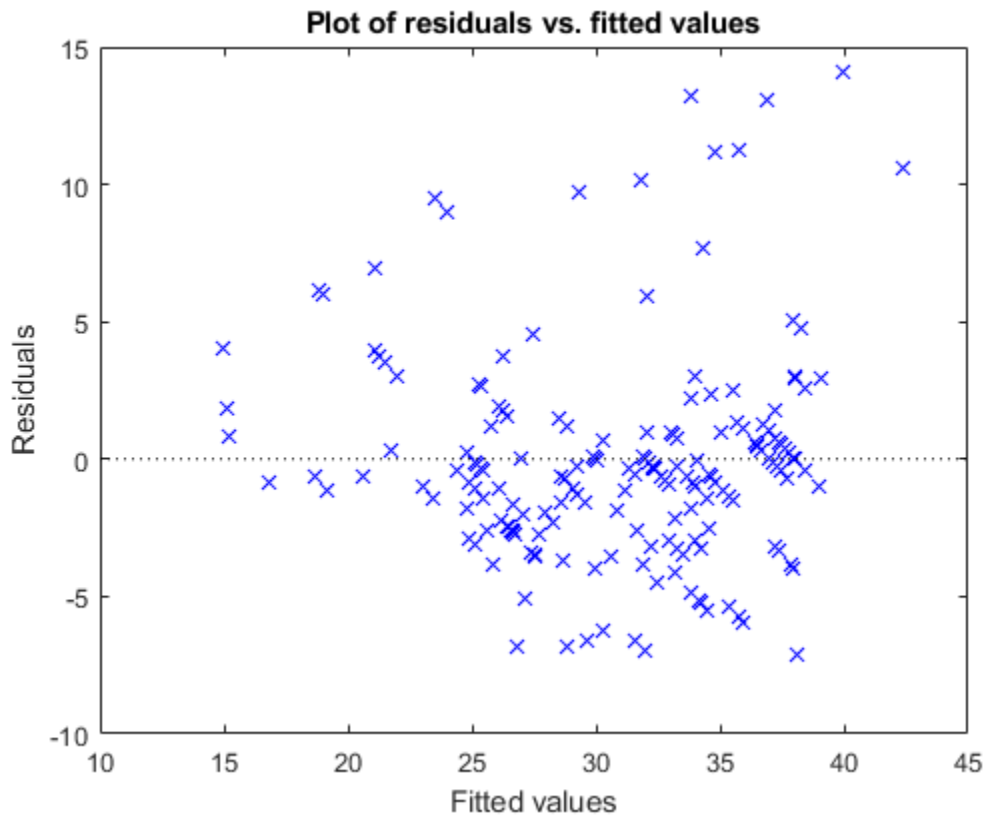


This plot also suggests that the residuals are not distributed equally around their median, as would be expected for normal distribution.

Plot the residuals versus the fitted values.

```
plotResiduals(mdl, 'fitted')
```





The increase in the variance as the fitted values increase suggests possible heteroscedasticity.

## References

- [1] Atkinson, A. T. *Plots, Transformations, and Regression. An Introduction to Graphical Methods of Diagnostic Regression Analysis*. New York: Oxford Statistical Science Series, Oxford University Press, 1987.
- [2] Neter, J., M. H. Kutner, C. J. Nachtsheim, and W. Wasserman. *Applied Linear Statistical Models*. IRWIN, The McGraw-Hill Companies, Inc., 1996.

[3] Belsley, D. A., E. Kuh, and R. E. Welsch. *Regression Diagnostics, Identifying Influential Data and Sources of Collinearity*. Wiley Series in Probability and Mathematical Statistics, John Wiley and Sons, Inc., 1980.

## See Also

`LinearModel` | `dwtest` | `fitlm` | `plotDiagnostics` | `plotResiduals` | `stepwiselm`

## Related Examples

- “Examine Quality and Adjust the Fitted Model” on page 11-20
- “Interpret Linear Regression Results” on page 11-64

## Summary of Output and Diagnostic Statistics

Name	LinearModel	regstats
“Cook’s Distance” on page 11-71	CooksDistance and cookd	cookd
“Coefficient Confidence Intervals” on page 11-76	coefCI	N/A
“Coefficient Covariance and Standard Errors” on page 11-75	CoefficientCovariance	covb
“Coefficient of Determination (R-Squared)” on page 11-79	Rsquared: Ordinary, Adjusted	rsquare, adjrsquare
“Delete-1 Change in Covariance (covratio)” on page 11-82	CovRatio	covratio
“Delete-1 Scaled Difference in Coefficient Estimates (Dfbetas)” on page 11-85	Dfbetas	dfbetas
“Delete-1 Scaled Change in Fitted Values (Dffits)” on page 11-86	Dffits	dffits
“Delete-1 Variance (S2_i)” on page 11-89	S2_i	s2_i
“Durbin-Watson Test” on page 11-93	dwtest	dwstat
“F-statistic” on page 11-95	Fstat	fstat
“Hat Matrix” on page 11-102	HatMatrix	hatmat
“Leverage” on page 11-103	Leverage	leverage
“Residuals” on page 11-106	Residuals: Raw, Pearson, Studentized, Standardized	r, studres, standres
“t-statistic” on page 11-98	tstats	tstat

## See Also

`LinearModel` | `dwtest` | `fitlm` | `plotDiagnostics` | `plotResiduals` | `stepwiselm`

## Related Examples

- “Examine Quality and Adjust the Fitted Model” on page 11-20
- “Interpret Linear Regression Results” on page 11-64

# Wilkinson Notation

## In this section...

“Overview” on page 11-117

“Formula Specification” on page 11-118

“Linear Model Examples” on page 11-121

“Linear Mixed-Effects Model Examples” on page 11-122

“Generalized Linear Model Examples” on page 11-124

“Generalized Linear Mixed-Effects Model Examples” on page 11-125

“Repeated Measures Model Examples” on page 11-126

## Overview

Wilkinson notation provides a way to describe regression and repeated measures models without specifying coefficient values. This specialized notation identifies the response variable and which predictor variables to include or exclude from the model. You can also include squared and higher-order terms, interaction terms, and grouping variables in the model formula.

Specifying a model using Wilkinson notation provides several advantages:

- You can include or exclude individual predictors and interaction terms from the model. For example, using the 'Interactions' name-value pair available in each model fitting functions includes interaction terms for all pairs of variables. Using Wilkinson notation instead allows you to include only the interaction terms of interest.
- You can change the model formula without changing the design matrix, if your input data uses the `table` data type. For example, if you fit an initial model using all the available predictor variables, but decide to remove a variable that is not statistically significant, then you can re-write the model formula to include only the variables of interest. You do not need to make any changes to the input data itself.

Statistics and Machine Learning Toolbox offers several model fitting functions that use Wilkinson notation, including:

- Linear models (using `fitlm` and `stepwiselm`)

- Generalized linear models (using `fitglm`)
- Linear mixed-effects models (using `fitlme` and `fitlmematrix`)
- Generalized linear mixed-effects models (using `fitglme`)
- Repeated measures models (using `fitrm`)

## Formula Specification

A formula for model specification is a character vector of the form `y ~ terms`, where `y` is the name of the response variable, and `terms` defines the model using the predictor variable names and the following operators.

### Predictor Variables

Predictor Terms in Model	Wilkinson Notation
intercept	1
no intercept	-1
$x_1$	x1
$x_1, x_2$	x1 + x2
$x_1, x_2, x_1x_2$	x1*x2 or x1 + x2 + x1:x2
$x_1x_2$	x1:x2
$x_1, x_1^2$	x1^2
$x_1^2$	x1^2 - x1

Wilkinson notation includes an intercept term in the model by default, even if you do not add 1 to the model formula. To exclude the intercept from the model, use -1 in the formula.

The \* operator (for interactions) and the ^ operator (for power and exponents) automatically include all lower-order terms. For example, if you specify  $x^3$ , the model will automatically include  $x^3$ ,  $x^2$ , and  $x$ . If you want to exclude certain variables from the model, use the - operator to remove the unwanted terms.

### Random-Effects and Mixed-Effects Models

For random-effects and mixed-effects models, the formula specification includes the names of the predictor variables and the grouping variables. For example, if the predictor

variable  $x_1$  is a random effect grouped by the variable  $g$ , then represent this in Wilkinson notation as follows:

```
(x1 | g)
```

### Repeated Measures Models

For repeated measures models, the formula specification includes all of the repeated measures as responses, and the factors as predictor variables. Specify the response variables for repeated measures models as described in the following table.

Response Terms in Model	Wilkinson Notation
$y_1$	y1
$y_1, y_2, y_3$	y1, y2, y3
$y_1, y_2, y_3, y_4, y_5,$	y1-y5

For example, if you have three repeated measures as responses and the factors  $x_1$ ,  $x_2$ , and  $x_3$  as the predictor variables, then you can define the repeated measures model using Wilkinson notation as follows:

```
y1, y2, y3 ~ x1 + x2 + x3
```

or

```
y1-y3 ~ x1 + x2 + x3
```

### Variable Names

If the input data (response and predictor variables) is stored in a table or dataset array, you can specify the formula using the variable names. For example, load the `carsmall` sample data. Create a table containing `Weight`, `Acceleration`, and `MPG`. Name each variable using the `'VariableNames'` name-value pair argument of the fitting function `fitlm`. Then fit the following model to the data:

$$MPG = \beta_0 + \beta_1 Weight + \beta_2 Acceleration$$

```
load carsmall
tbl = table(Weight, Acceleration, MPG, 'VariableNames', {'Weight', 'Acceleration', 'MPG'});
mdl = fitlm(tbl, 'MPG ~ Weight + Acceleration')

mdl =
```

Linear regression model:

MPG ~ 1 + Weight + Acceleration

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	45.155	3.4659	13.028	1.6266e-22
Weight	-0.0082475	0.00059836	-13.783	5.3165e-24
Acceleration	0.19694	0.14743	1.3359	0.18493

Number of observations: 94, Error degrees of freedom: 91

Root Mean Squared Error: 4.12

R-squared: 0.743, Adjusted R-Squared 0.738

F-statistic vs. constant model: 132, p-value = 1.38e-27

The model object display uses the variable names provided in the input table.

If the input data is stored as a matrix, you can specify the formula using default variable names such as `y`, `x1`, and `x2`. For example, load the `carsmall` sample data. Create a matrix containing the predictor variables `Weight` and `Acceleration`. Then fit the following model to the data:

$$MPG = \beta_0 + \beta_1 \text{Weight} + \beta_2 \text{Acceleration}$$

```
load carsmall
X = [Weight,Acceleration];
y = MPG;
mdl = fitlm(X,y,'y ~ x1 + x2')

mdl =
```

Linear regression model:

y ~ 1 + x1 + x2

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	45.155	3.4659	13.028	1.6266e-22
x1	-0.0082475	0.00059836	-13.783	5.3165e-24
x2	0.19694	0.14743	1.3359	0.18493



Number of observations: 94, Error degrees of freedom: 91  
 Root Mean Squared Error: 4.12  
 R-squared: 0.743, Adjusted R-Squared 0.738  
 F-statistic vs. constant model: 132, p-value = 1.38e-27

The term  $x_1$  in the model specification formula corresponds to the first column of the predictor variable matrix  $X$ . The term  $x_2$  corresponds to the second column of the input matrix. The term  $y$  corresponds to the response variable.

## Linear Model Examples

Use `fitlm` and `stepwiselm` to fit linear models.

### Intercept and Two Predictors

For a linear regression model with an intercept and two fixed-effects predictors, such as

$$y_i = \beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \varepsilon_i,$$

specify the model formula using Wilkinson notation as follows:

```
'y ~ x1 + x2'
```

### No Intercept and Two Predictors

For a linear regression model with no intercept and two fixed-effects predictors, such as

$$y_i = \beta_1 x_{i1} + \beta_2 x_{i2} + \varepsilon_i,$$

specify the model formula using Wilkinson notation as follows:

```
'y ~ -1 + x1 + x2'
```

### Intercept, Two Predictors, and an Interaction Term

For a linear regression model with an intercept, two fixed-effects predictors, and an interaction term, such as

$$y_i = \beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \beta_3 x_{i1} x_{i2} + \varepsilon_i,$$

specify the model formula using Wilkinson notation as follows:

```
'y ~ x1*x2'
```

or

```
'y ~ x1 + x2 + x1:x2'
```

### Intercept, Three Predictors, and All Interaction Effects

For a linear regression model with an intercept, three fixed-effects predictors, and interaction effects between all three predictors plus all lower-order terms, such as

$$y_i = \beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \beta_3 x_{i3} + \beta_4 x_1 x_{i2} + \beta_5 x_1 x_{i3} + \beta_6 x_2 x_{i3} + \beta_7 x_{i1} x_{i2} x_{i3} + \varepsilon_i,$$

specify the model formula using Wilkinson notation as follows:

```
'y ~ x1*x2*x3'
```

### Intercept, Three Predictors, and Selected Interaction Effects

For a linear regression model with an intercept, three fixed-effects predictors, and interaction effects between two of the predictors, such as

$$y_i = \beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \beta_3 x_{i3} + \beta_4 x_1 x_{i2} + \varepsilon_i,$$

specify the model formula using Wilkinson notation as follows:

```
'y ~ x1*x2 + x3'
```

or

```
'y ~ x1 + x2 + x3 + x1:x2'
```

### Intercept, Three Predictors, and Lower-Order Interaction Effects Only

For a linear regression model with an intercept, three fixed-effects predictors, and pairwise interaction effects between all three predictors, but excluding an interaction effect between all three predictors simultaneously, such as

$$y_i = \beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \beta_3 x_{i3} + \beta_4 x_1 x_{i2} + \beta_5 x_{i1} x_{i3} + \beta_6 x_{i2} x_{i3} + \varepsilon_i,$$

specify the model formula using Wilkinson notation as follows:

```
'y ~ x1*x2*x3 - x1:x2:x3'
```

## Linear Mixed-Effects Model Examples

Use `fitlme` and `fitlmematrix` to fit linear mixed-effects models.

### Random Effect Intercept, No Predictors

For a linear mixed-effects model that contains a random intercept but no predictor terms, such as

$$y_{im} = \beta_{0m},$$

where

$$\beta_{0m} = \beta_{00} + b_{0m}, b_{0m} \sim N(0, \sigma_0^2)$$

and  $g$  is the grouping variable with  $m$  levels, specify the model formula using Wilkinson notation as follows:

```
'y ~ (1 | g)'
```

### Random Intercept and Fixed Slope for One Predictor

For a linear mixed-effects model that contains a fixed intercept, random intercept, and fixed slope for the continuous predictor variable, such as

$$y_{im} = \beta_{0m} + \beta_1 x_{im},$$

where

$$\beta_{0m} = \beta_{00} + b_{0m}, b_{0m} \sim N(0, \sigma_0^2)$$

and  $g$  is the grouping variable with  $m$  levels, specify the model formula using Wilkinson notation as follows:

```
'y ~ x1 + (1 | g)'
```

### Random Intercept and Random Slope for One Predictor

For a linear mixed-effects model that contains a fixed intercept, plus a random intercept and a random slope that have a possible correlation between them, such as

$$y_{im} = \beta_{0m} + \beta_{1m} x_{im},$$

where

$$\beta_{0m} = \beta_{00} + b_{0m}$$

$$\beta_{1m} = \beta_{10} + b_{1m}$$

$$\begin{bmatrix} b_{0m} \\ b_{1m} \end{bmatrix} \sim N\{0, \sigma^2 D(\theta)\}$$

and  $D$  is a 2-by-2 symmetric and positive semidefinite covariance matrix, parameterized by a variance component vector  $\theta$ , specify the model formula using Wilkinson notation as follows:

```
'y ~ x1 + (x1 | g)'
```

The pattern of the random effects covariance matrix is determined by the model fitting function. To specify the covariance matrix pattern, use the name-value pairs available through `fitlme` when fitting the model. For example, you can specify the assumption that the random intercept and random slope are independent of one another using the 'CovariancePattern' name-value pair argument in `fitlme`.

## Generalized Linear Model Examples

Use `fitglm` and `stepwiseglm` to fit generalized linear models.

In a generalized linear model, the  $y$  response variable has a distribution other than normal, but you can represent the model as an equation that is linear in the regression coefficients. Specifying a generalized linear model requires three parts:

- Distribution of the response variable
- Link function
- Linear predictor

The distribution of the response variable and the link function are specified using name-value pair arguments in the fit function `fitglm` or `stepwiseglm`.

The linear predictor portion of the equation, which appears on the right side of the  $\sim$  symbol in the model specification formula, uses Wilkinson notation in the same way as for the linear model examples.

A generalized linear model models the link function, rather than the actual response, as  $y$ . This is reflected in the output display for the model object.

### Intercept and Two Predictors

For a generalized linear regression model with an intercept and two predictors, such as

$$\log(y_i) = \beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2},$$

specify the model formula using Wilkinson notation as follows:

```
'y ~ x1 + x2'
```

## Generalized Linear Mixed-Effects Model Examples

Use `fitglme` to fit generalized linear mixed-effects models.

In a generalized linear mixed-effects model, the  $y$  response variable has a distribution other than normal, but you can represent the model as an equation that is linear in the regression coefficients. Specifying a generalized linear model requires three parts:

- Distribution of the response variable
- Link function
- Linear predictor

The distribution of the response variable and the link function are specified using name-value pair arguments in the fit function `fitglme`.

The linear predictor portion of the equation, which appears on the right side of the `~` symbol in the model specification formula, uses Wilkinson notation in the same way as for the linear mixed-effects model examples.

A generalized linear model models the link function as  $y$ , not the response itself. This is reflected in the output display for the model object.

The pattern of the random effects covariance matrix is determined by the model fitting function. To specify the covariance matrix pattern, use the name-value pairs available through `fitglme` when fitting the model. For example, you can specify the assumption that the random intercept and random slope are independent of one another using the `'CovariancePattern'` name-value pair argument in `fitglme`.

### Random Intercept and Fixed Slope for One Predictor

For a generalized linear mixed-effects model that contains a fixed intercept, random intercept, and fixed slope for the continuous predictor variable, where the response can be modeled using a Poisson distribution, such as

$$\log(y_{im}) = \beta_0 + \beta_1 x_{im} + b_i,$$

where

$$b_i \sim N(0, \sigma_b^2)$$

and  $g$  is the grouping variable with  $m$  levels, specify the model formula using Wilkinson notation as follows:

```
'y ~ x1 + (1 | g)'
```

## Repeated Measures Model Examples

Use `fitrm` to fit repeated measures models.

### One Predictor

For a repeated measures model with five response measurements and one predictor variable, specify the model formula using Wilkinson notation as follows:

```
'y1-y5 ~ x1'
```

### Three Predictors and an Interaction Term

For a repeated measures model with five response measurements and three predictor variables, plus an interaction between two of the predictor variables, specify the model formula using Wilkinson notation as follows:

```
'y1-y5 ~ x1*x2 + x3'
```

## Stepwise Regression

### In this section...

“Stepwise Regression to Select Appropriate Models” on page 11-127

“Compare large and small stepwise models” on page 11-127

### Stepwise Regression to Select Appropriate Models

`stepwiselm` creates a linear model and automatically adds to or trims the model. To create a small model, start from a constant model. To create a large model, start with a model containing many terms. A large model usually has lower error as measured by the fit to the original data, but might not have any advantage in predicting new data.

`stepwiselm` can use all the name-value options from `fitlm`, with additional options relating to the starting and bounding models. In particular:

- For a small model, start with the default lower bounding model: 'constant' (a model that has no predictor terms).
- The default upper bounding model has linear terms and interaction terms (products of pairs of predictors). For an upper bounding model that also includes squared terms, set the Upper name-value pair to 'quadratic'.

### Compare large and small stepwise models

This example shows how to compare models that `stepwiselm` returns starting from a constant model and starting from a full interaction model.

Load the `carbig` data and create a table from some of the data.

```
load carbig
tbl = table(Acceleration, Displacement, Horsepower, Weight, MPG);
```

Create a mileage model stepwise starting from the constant model.

```
mdl1 = stepwiselm(tbl, 'constant', 'ResponseVar', 'MPG')
```

1. Adding Weight, FStat = 888.8507, pValue = 2.9728e-103
2. Adding Horsepower, FStat = 3.8217, pValue = 0.00049608
3. Adding Horsepower:Weight, FStat = 64.8709, pValue = 9.93362e-15

mdl1 =

Linear regression model:  
 MPG ~ 1 + Horsepower\*Weight

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	63.558	2.3429	27.127	1.2343e-91
Horsepower	-0.25084	0.027279	-9.1952	2.3226e-18
Weight	-0.010772	0.00077381	-13.921	5.1372e-36
Horsepower:Weight	5.3554e-05	6.6491e-06	8.0542	9.9336e-15

Number of observations: 392, Error degrees of freedom: 388  
 Root Mean Squared Error: 3.93  
 R-squared: 0.748, Adjusted R-Squared 0.746  
 F-statistic vs. constant model: 385, p-value = 7.26e-116

**Create a mileage model stepwise starting from the full interaction model.**

mdl2 = stepwiselm(tbl, 'interactions', 'ResponseVar', 'MPG')

1. Removing Acceleration:Displacement, FStat = 0.024186, pValue = 0.8765
2. Removing Displacement:Weight, FStat = 0.33103, pValue = 0.56539
3. Removing Acceleration:Horsepower, FStat = 1.7334, pValue = 0.18876
4. Removing Acceleration:Weight, FStat = 0.93269, pValue = 0.33477
5. Removing Horsepower:Weight, FStat = 0.64486, pValue = 0.42245

mdl2 =

Linear regression model:  
 MPG ~ 1 + Acceleration + Weight + Displacement\*Horsepower

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	61.285	2.8052	21.847	1.8593e-69
Acceleration	-0.34401	0.11862	-2.9	0.0039445
Displacement	-0.081198	0.010071	-8.0623	9.5014e-15



Horsepower	-0.24313	0.026068	-9.3265	8.6556e-19
Weight	-0.0014367	0.00084041	-1.7095	0.088166
Displacement:Horsepower	0.00054236	5.7987e-05	9.3531	7.0527e-19

Number of observations: 392, Error degrees of freedom: 386

Root Mean Squared Error: 3.84

R-squared: 0.761, Adjusted R-Squared 0.758

F-statistic vs. constant model: 246, p-value = 1.32e-117

Notice that:

- `mdl1` has four coefficients (the Estimate column), and `mdl2` has six coefficients.
- The adjusted R-squared of `mdl1` is 0.746, which is slightly less (worse) than that of `mdl2`, 0.758.

Create a mileage model stepwise with a full quadratic model as the upper bound, starting from the full quadratic model:

```
mdl3 = stepwiselm(tbl, 'quadratic', 'ResponseVar', 'MPG', 'Upper', 'quadratic');
```

1. Removing Acceleration:Horsepower, FStat = 0.075209, pValue = 0.78405
2. Removing Acceleration:Weight, FStat = 0.072756, pValue = 0.78751
3. Removing Horsepower:Weight, FStat = 0.12569, pValue = 0.72314
4. Removing Weight^2, FStat = 1.194, pValue = 0.27521
5. Removing Displacement:Weight, FStat = 1.2839, pValue = 0.25789
6. Removing Displacement^2, FStat = 2.069, pValue = 0.15114
7. Removing Horsepower^2, FStat = 0.74063, pValue = 0.39

Compare the three model complexities by examining their formulas.

```
mdl1.Formula
```

```
ans =
```

```
MPG ~ 1 + Horsepower*Weight
```

```
mdl2.Formula
```

```
ans =
```

```
MPG ~ 1 + Acceleration + Weight + Displacement*Horsepower
```

```
mdl3.Formula
```

```
ans =
```

```
MPG ~ 1 + Weight + Acceleration*Displacement + Displacement*Horsepower  
      + Acceleration^2
```

The adjusted  $R^2$  values improve slightly as the models become more complex:

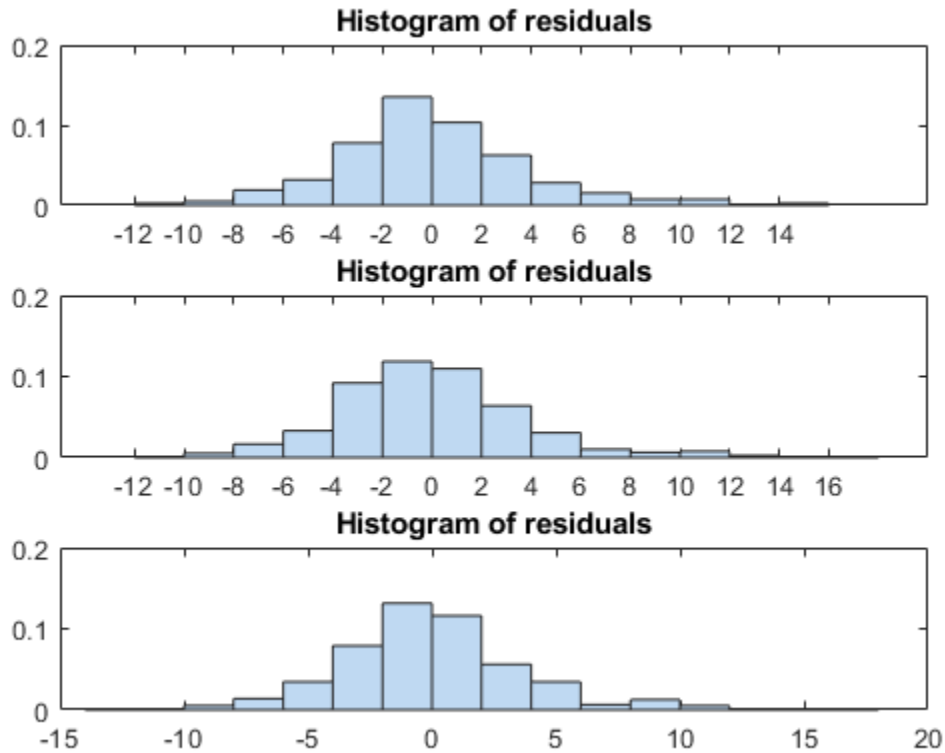
```
RSquared = [mdl1.Rsquared.Adjusted, ...  
            mdl2.Rsquared.Adjusted, mdl3.Rsquared.Adjusted]
```

```
RSquared =
```

```
0.7465    0.7580    0.7599
```

Compare residual plots of the three models.

```
subplot(3,1,1)  
plotResiduals(mdl1)  
subplot(3,1,2)  
plotResiduals(mdl2)  
subplot(3,1,3)  
plotResiduals(mdl3)
```



The models have similar residuals. It is not clear which fits the data better.

Interestingly, the more complex models have larger maximum deviations of the residuals:

```

Rrange1 = [min(md11.Residuals.Raw),max(md11.Residuals.Raw)];
Rrange2 = [min(md12.Residuals.Raw),max(md12.Residuals.Raw)];
Rrange3 = [min(md13.Residuals.Raw),max(md13.Residuals.Raw)];
Ranges = [Rrange1;Rrange2;Rrange3]

```

```
Ranges =
```

```
-10.7725    14.7314
```

-11.4407	16.7562
-12.2723	16.7927

## Robust Regression — Reduce Outlier Effects

### In this section...

“What Is Robust Regression?” on page 11-133

“Robust Regression versus Standard Least-Squares Fit” on page 11-133

### What Is Robust Regression?

The models described in “What Are Linear Regression Models?” on page 11-8 are based on certain assumptions, such as a normal distribution of errors in the observed responses. If the distribution of errors is asymmetric or prone to outliers, model assumptions are invalidated, and parameter estimates, confidence intervals, and other computed statistics become unreliable. Use `fitlm` with the `RobustOpts` name-value pair to create a model that is not much affected by outliers. The robust fitting method is less sensitive than ordinary least squares to large changes in small parts of the data.

Robust regression works by assigning a weight to each data point. Weighting is done automatically and iteratively using a process called *iteratively reweighted least squares*. In the first iteration, each point is assigned equal weight and model coefficients are estimated using ordinary least squares. At subsequent iterations, weights are recomputed so that points farther from model predictions in the previous iteration are given lower weight. Model coefficients are then recomputed using weighted least squares. The process continues until the values of the coefficient estimates converge within a specified tolerance.

### Robust Regression versus Standard Least-Squares Fit

This example shows how to use robust regression. It compares the results of a robust fit to a standard least-squares fit.

#### Step 1. Prepare data.

Load the `moore` data. The data is in the first five columns, and the response in the sixth.

```
load moore
X = [moore(:,1:5)];
y = moore(:,6);
```

**Step 2. Fit robust and nonrobust models.**

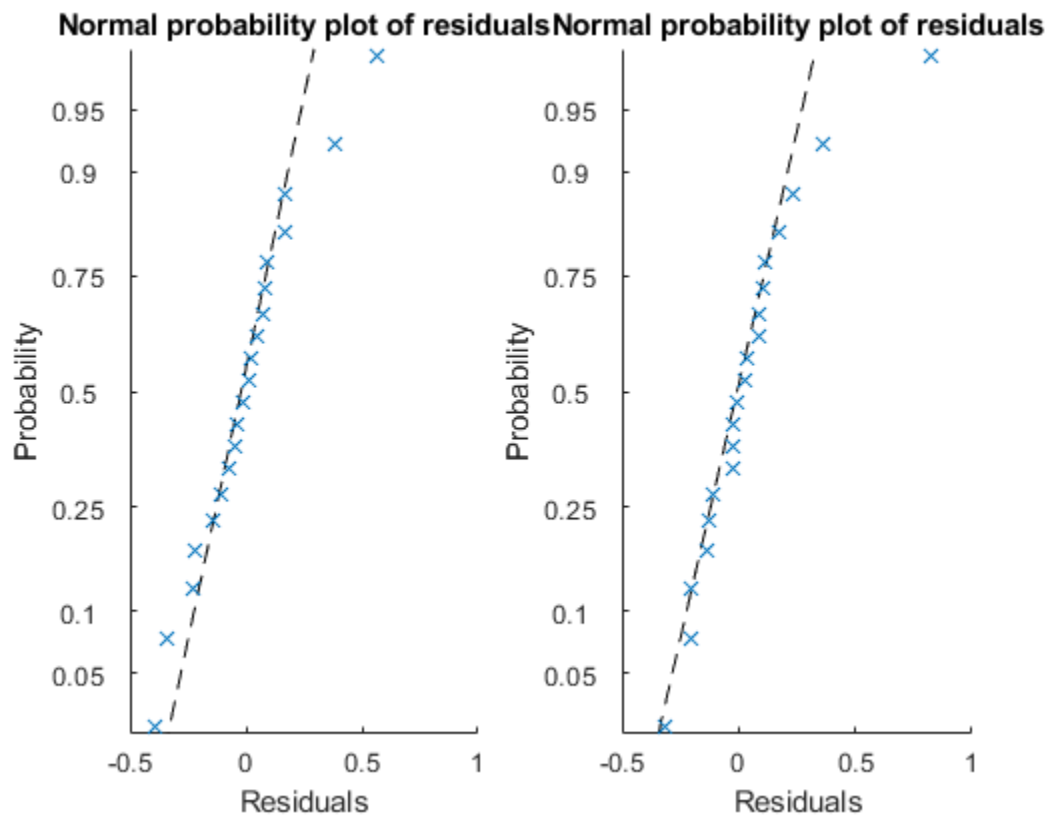
Fit two linear models to the data, one using robust fitting, one not.

```
mdl = fitlm(X,y); % not robust
mdlr = fitlm(X,y,'RobustOpts','on');
```

**Step 3. Examine model residuals.**

Examine the residuals of the two models.

```
subplot(1,2,1)
plotResiduals(mdl,'probability')
subplot(1,2,2)
plotResiduals(mdlr,'probability')
```



The residuals from the robust fit (right half of the plot) are nearly all closer to the straight line, except for the one obvious outlier.

**Step 4. Remove the outlier from the standard model.**

Find the index of the outlier. Examine the weight of the outlier in the robust fit.

```
[~,outlier] = max mdlr.Residuals.Raw);  
mdlr.Robust.Weights(outlier)
```

```
ans =  
  
    0.0246
```

Check the median weight.

```
median(mdlr.Robust.Weights)
```

```
ans =  
  
    0.9718
```

This weight of the outlier in the robust fit is much less than a typical weight of an observation.

## Introduction to Ridge Regression

Coefficient estimates for the models described in “Linear Regression” on page 11-11 rely on the independence of the model terms. When terms are correlated and the columns of the design matrix  $X$  have an approximate linear dependence, the matrix  $(X^T X)^{-1}$  becomes close to singular. As a result, the least-squares estimate

$$\hat{\beta} = (X^T X)^{-1} X^T y$$

becomes highly sensitive to random errors in the observed response  $y$ , producing a large variance. This situation of multicollinearity can arise, for example, when data are collected without an experimental design.

Ridge regression addresses the problem by estimating regression coefficients using

$$\hat{\beta} = (X^T X + kI)^{-1} X^T y$$

where  $k$  is the ridge parameter and  $I$  is the identity matrix. Small positive values of  $k$  improve the conditioning of the problem and reduce the variance of the estimates. While biased, the reduced variance of ridge estimates often result in a smaller mean square error when compared to least-squares estimates.

The Statistics and Machine Learning Toolbox function `ridge` carries out ridge regression.

### See Also

`ridge`

### Related Examples

- “Ridge Regression” on page 11-137



## Ridge Regression

This example shows how to perform ridge regression.

Load the data in `acetylene.mat`, with observations of the predictor variables `x1`, `x2`, `x3`, and the response variable `y`.

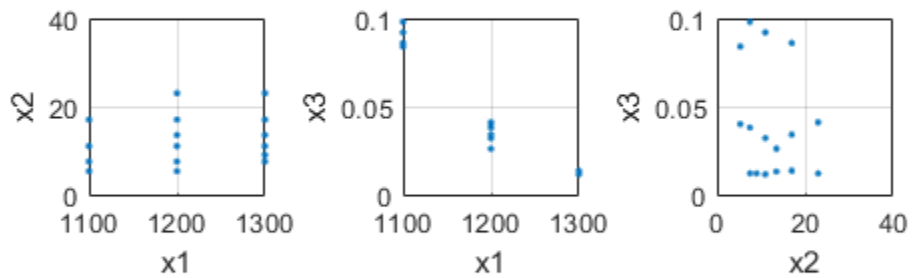
```
load acetylene
```

Plot the predictor variables against each other.

```
subplot(1,3,1)
plot(x1,x2, '.')
xlabel('x1')
ylabel('x2')
grid on
axis square
```

```
subplot(1,3,2)
plot(x1,x3, '.')
xlabel('x1')
ylabel('x3')
grid on
axis square
```

```
subplot(1,3,3)
plot(x2,x3, '.')
xlabel('x2')
ylabel('x3')
grid on
axis square
```



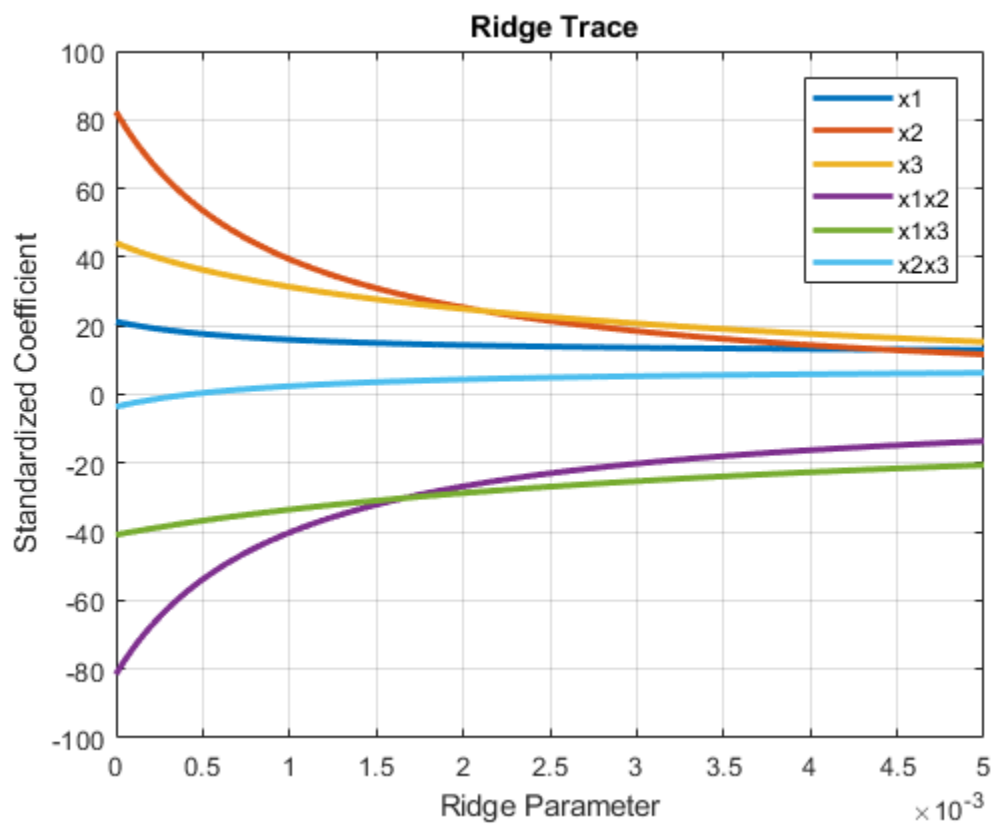
Note the correlation between  $x_1$  and the other two predictor variables.

Use `ridge` and `x2fx` to compute coefficient estimates for a multilinear model with interaction terms, for a range of ridge parameters.

```
X = [x1 x2 x3];
D = x2fx(X, 'interaction');
D(:,1) = []; % No constant term
k = 0:1e-5:5e-3;
betahat = ridge(y,D,k);
```

Plot the ridge trace.

```
figure
plot(k,betahat,'LineWidth',2)
ylim([-100 100])
grid on
xlabel('Ridge Parameter')
ylabel('Standardized Coefficient')
title('{\bf Ridge Trace}')
legend('x1','x2','x3','x1x2','x1x3','x2x3')
```



The estimates stabilize to the right of the plot. Note that the coefficient of the  $x_2 \times x_3$  interaction term changes sign at a value of the ridge parameter  $\approx 5 \times 10^{-4}$ .

### See Also

ridge

### Related Examples

- “Introduction to Ridge Regression” on page 11-136

## Lasso and Elastic Net

In this section...
“What Are Lasso and Elastic Net?” on page 11-141
“Lasso and Elastic Net Details” on page 11-141
“References” on page 11-143

### What Are Lasso and Elastic Net?

Lasso is a regularization technique. Use `lasso` to:

- Reduce the number of predictors in a regression model.
- Identify important predictors.
- Select among redundant predictors.
- Produce shrinkage estimates with potentially lower predictive errors than ordinary least squares.

Elastic net is a related technique. Use elastic net when you have several highly correlated variables. `lasso` provides elastic net regularization when you set the Alpha name-value pair to a number strictly between 0 and 1.

See “Lasso and Elastic Net Details” on page 11-141.

For lasso regularization of regression ensembles, see `regularize`.

### Lasso and Elastic Net Details

#### Overview of Lasso and Elastic Net

Lasso is a regularization technique for performing linear regression. Lasso includes a penalty term that constrains the size of the estimated coefficients. Therefore, it resembles ridge regression on page 11-136. Lasso is a shrinkage estimator: it generates coefficient estimates that are biased to be small. Nevertheless, a lasso estimator can have smaller mean squared error than an ordinary least-squares estimator when you apply it to new data.

Unlike ridge regression, as the penalty term increases, lasso sets more coefficients to zero. This means that the lasso estimator is a smaller model, with fewer predictors. As

such, lasso is an alternative to stepwise regression on page 11-127 and other model selection and dimensionality reduction techniques.

Elastic net is a related technique. Elastic net is a hybrid of ridge regression and lasso regularization. Like lasso, elastic net can generate reduced models by generating zero-valued coefficients. Empirical studies have suggested that the elastic net technique can outperform lasso on data with highly correlated predictors.

### Definition of Lasso

The lasso technique solves this regularization problem. For a given value of  $\lambda$ , a nonnegative parameter, `lasso` solves the problem

$$\min_{\beta_0, \beta} \left( \frac{1}{2N} \sum_{i=1}^N (y_i - \beta_0 - x_i^T \beta)^2 + \lambda \sum_{j=1}^p |\beta_j| \right).$$

- $N$  is the number of observations.
- $y_i$  is the response at observation  $i$ .
- $x_i$  is data, a vector of  $p$  values at observation  $i$ .
- $\lambda$  is a positive regularization parameter corresponding to one value of `Lambda`.
- The parameters  $\beta_0$  and  $\beta$  are scalar and  $p$ -vector respectively.

As  $\lambda$  increases, the number of nonzero components of  $\beta$  decreases.

The lasso problem involves the  $L^1$  norm of  $\beta$ , as contrasted with the elastic net algorithm.

### Definition of Elastic Net

The elastic net technique solves this regularization problem. For an  $\alpha$  strictly between 0 and 1, and a nonnegative  $\lambda$ , elastic net solves the problem

$$\min_{\beta_0, \beta} \left( \frac{1}{2N} \sum_{i=1}^N (y_i - \beta_0 - x_i^T \beta)^2 + \lambda P_\alpha(\beta) \right),$$

where

$$P_\alpha(\beta) = \frac{(1-\alpha)}{2} \|\beta\|_2^2 + \alpha \|\beta\|_1 = \sum_{j=1}^p \left( \frac{(1-\alpha)}{2} \beta_j^2 + \alpha |\beta_j| \right)$$

Elastic net is the same as lasso when  $\alpha = 1$ . As  $\alpha$  shrinks toward 0, elastic net approaches ridge regression. For other values of  $\alpha$ , the penalty term  $P_\alpha(\beta)$  interpolates between the  $L^1$  norm of  $\beta$  and the squared  $L^2$  norm of  $\beta$ .

## References

- [1] Tibshirani, R. *Regression shrinkage and selection via the lasso*. Journal of the Royal Statistical Society, Series B, Vol 58, No. 1, pp. 267–288, 1996.
- [2] Zou, H. and T. Hastie. *Regularization and variable selection via the elastic net*. Journal of the Royal Statistical Society, Series B, Vol. 67, No. 2, pp. 301–320, 2005.
- [3] Friedman, J., R. Tibshirani, and T. Hastie. *Regularization paths for generalized linear models via coordinate descent*. Journal of Statistical Software, Vol 33, No. 1, 2010. <http://www.jstatsoft.org/v33/i01>
- [4] Hastie, T., R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*, 2nd edition. Springer, New York, 2008.

## Wide Data via Lasso and Parallel Computing

This example shows how to identify important predictors using lasso and cross validation.

Lasso and elastic net are especially well suited to wide data, meaning data with more predictors than observations. Obviously, there are redundant predictors in this type of data. Use `lasso` along with cross validation to identify important predictors.

Cross validation can be slow. If you have a Parallel Computing Toolbox license, speed the computation using parallel computing.

Load the spectra data.

```
load spectra
Description

Description =

== Spectral and octane data of gasoline ==

NIR spectra and octane numbers of 60 gasoline samples

NIR:      NIR spectra, measured in 2 nm intervals from 900 nm to 1700 nm
octane:   octane numbers
spectra:  a dataset array containing variables for NIR and octane

Reference:
Kalivas, John H., "Two Data Sets of Near Infrared Spectra," Chemometrics
and Intelligent Laboratory Systems, v.37 (1997) pp.255-259
```

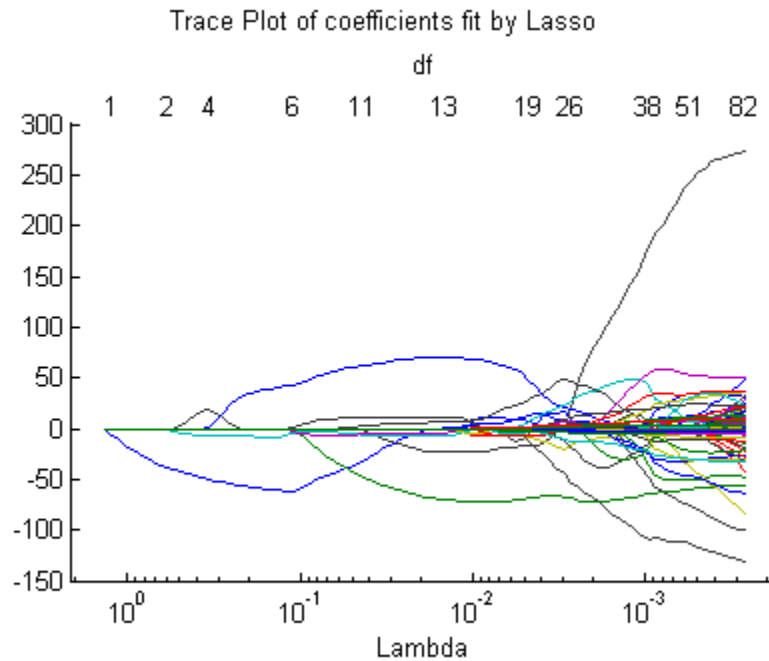
Compute the default lasso fit.

```
[b, fitinfo] = lasso(NIR,octane);
```

Plot the number of predictors in the fitted lasso regularization as a function of  $\lambda$ , using a logarithmic  $x$ -axis.

```
lassoPlot(b,fitinfo,'PlotType','Lambda','XScale','log');
```





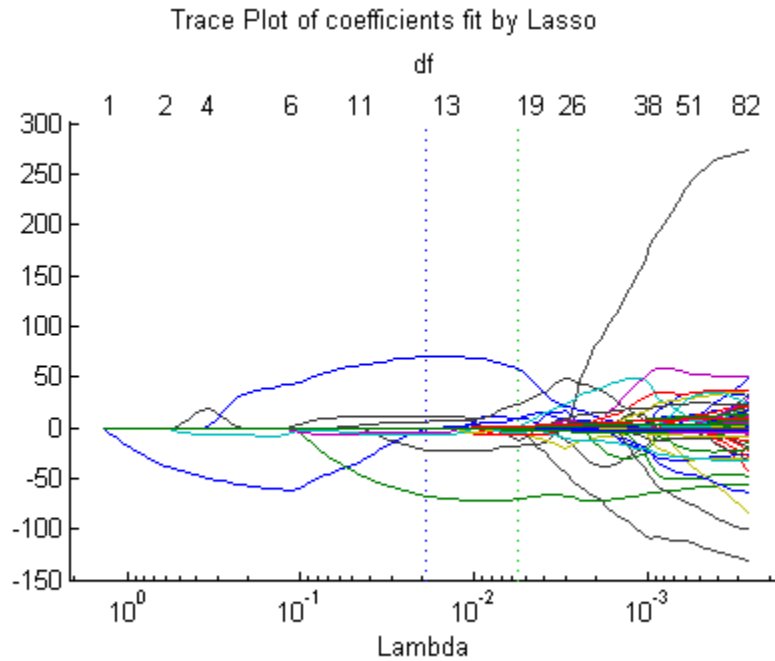
It is difficult to tell which value of Lambda is appropriate. To determine a good value, try fitting with cross validation.

```
tic
[b, fitinfo] = lasso(NIR,octane,'CV',10);
% A time-consuming operation
toc
```

Elapsed time is 226.876926 seconds.

Plot the result.

```
lassoPlot(b,fitinfo,'PlotType','Lambda','XScale','log');
```



You can see the suggested value of `Lambda` is over  $1e-2$ , and the `Lambda` with minimal MSE is under  $1e-2$ . These values are in the `fitinfo` structure.

```
fitinfo.LambdaMinMSE
fitinfo.Lambda1SE
```

```
ans =
    0.0057
```

```
ns =
    0.0190
```

Examine the quality of the fit for the suggested value of `Lambda`:

```
lambdaindex = fitinfo.Index1SE;
fitinfo.MSE(lambdaindex)
fitinfo.DF(lambdaindex)
```

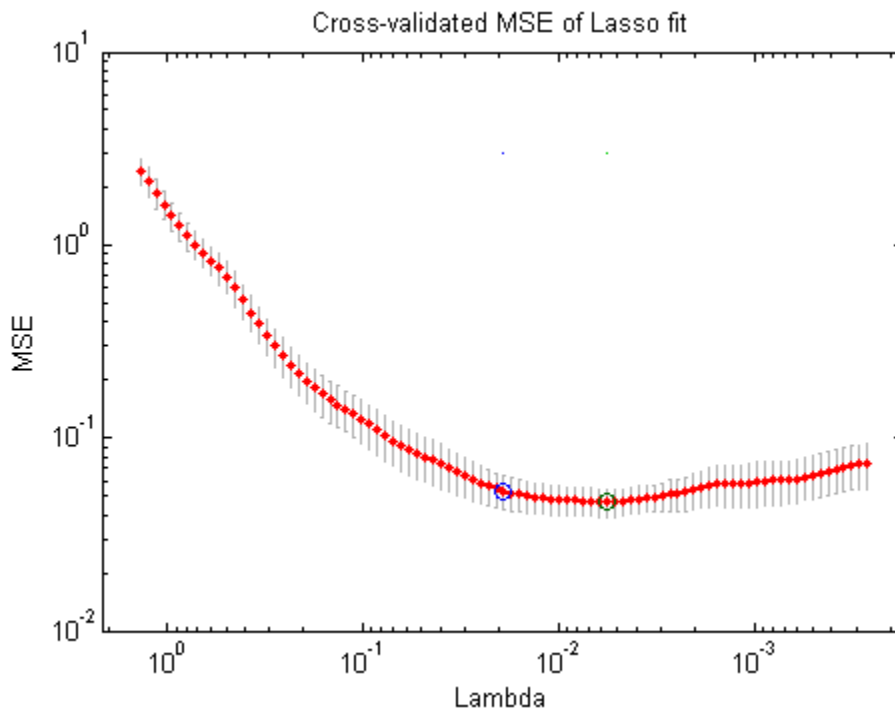
```
ans =
    0.0532
```

```
ans =
    11
```

The fit uses just 11 of the 401 predictors, and achieves a cross-validated MSE of 0.0532.

Examine the plot of cross-validated MSE.

```
lassoPlot(b, fitinfo, 'PlotType', 'CV');
% Use a log scale for MSE to see small MSE values better
set(gca, 'YScale', 'log');
```



As Lambda increases (toward the left), MSE increases rapidly. The coefficients are reduced too much and they do not adequately fit the responses.

As Lambda decreases, the models are larger (have more nonzero coefficients). The increasing MSE suggests that the models are overfitted.

The default set of `Lambda` values does not include values small enough to include all predictors. In this case, there does not appear to be a reason to look at smaller values. However, if you want smaller values than the default, use the `LambdaRatio` parameter, or supply a sequence of `Lambda` values using the `Lambda` parameter. For details, see the `lasso` reference page.

To compute the cross-validated lasso estimate faster, use parallel computing (available with a Parallel Computing Toolbox license). Open a parallel pool.

```
parpool()
```

```
Starting parpool using the 'local' profile ... connected to 2 workers.
```

```
ans =
```

```
Pool with properties:
```

```
AttachedFiles: {0x1 cell}
NumWorkers: 2
IdleTimeout: 30
Cluster: [1x1 parallel.cluster.Local]
RequestQueue: [1x1 parallel.RequestQueue]
SpmEnabled: 1
```

Fit the lasso regularization.

```
opts = statset('UseParallel',true);
tic;
[b, fitinfo] = lasso(NIR,octane,'CV',10,'Options',opts);
toc
```

```
Elapsed time is 114.712260 seconds.
```

Computing in parallel using two workers is faster on this problem.

## Lasso Regularization

To see how `lasso` identifies and discards unnecessary predictors:

- 1 Generate 200 samples of five-dimensional artificial data  $X$  from exponential distributions with various means:

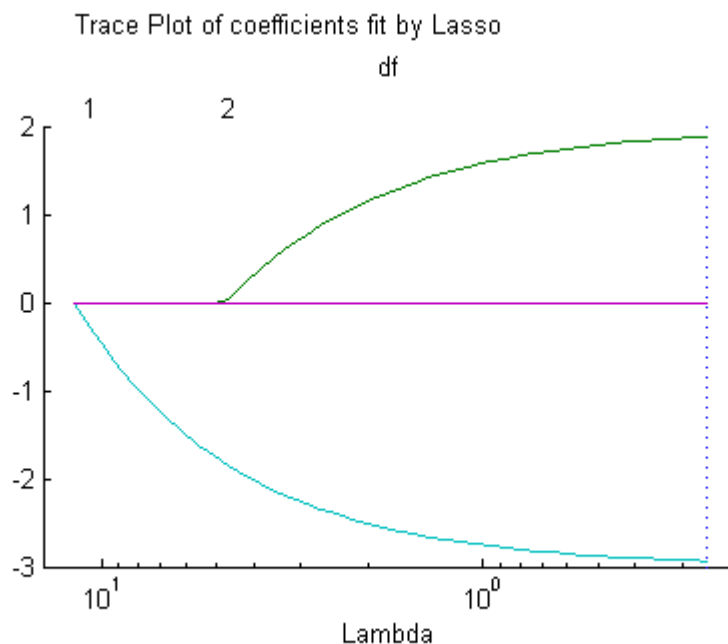
```
rng(3,'twister') % for reproducibility
X = zeros(200,5);
for ii = 1:5
    X(:,ii) = exprnd(ii,200,1);
end
```

- 2 Generate response data  $Y = X*r + \text{eps}$  where  $r$  has just two nonzero components, and the noise  $\text{eps}$  is normal with standard deviation 0.1:

```
r = [0;2;0;-3;0];
Y = X*r + randn(200,1)*.1;
```

- 3 Fit a cross-validated sequence of models with `lasso`, and plot the result:

```
[b, fitinfo] = lasso(X,Y,'CV',10);
lassoPlot(b,fitinfo,'PlotType','Lambda','XScale','log');
```



The plot shows the nonzero coefficients in the regression for various values of the Lambda regularization parameter. Larger values of Lambda appear on the left side of the graph, meaning more regularization, resulting in fewer nonzero regression coefficients.

The dashed vertical lines represent the Lambda value with minimal mean squared error (on the right), and the Lambda value with minimal mean squared error plus one standard deviation. This latter value is a recommended setting for Lambda. These lines appear only when you perform cross validation. Cross validate by setting the 'CV' name-value pair. This example uses 10-fold cross validation.

The upper part of the plot shows the degrees of freedom (df), meaning the number of nonzero coefficients in the regression, as a function of Lambda. On the left, the large value of Lambda causes all but one coefficient to be 0. On the right all five coefficients are nonzero, though the plot shows only two clearly. The other three coefficients are so small that you cannot visually distinguish them from 0.

For small values of `Lambda` (toward the right in the plot), the coefficient values are close to the least-squares estimate. See step “5” on page 11-151.

- 4** Find the `Lambda` value of the minimal cross-validated mean squared error plus one standard deviation. Examine the MSE and coefficients of the fit at that `Lambda`:

```
lam = fitinfo.Index1SE;
fitinfo.MSE(lam)
```

```
ans =
    0.1398
```

```
b(:,lam)
```

```
ans =
         0
    1.8855
         0
   -2.9367
         0
```

`lasso` did a good job finding the coefficient vector `r`.

- 5** For comparison, find the least-squares estimate of `r`:

```
rhat = X\Y
```

```
rhat =
   -0.0038
    1.9952
    0.0014
   -2.9993
    0.0031
```

The estimate `b(:,lam)` has slightly more mean squared error than the mean squared error of `rhat`:

```
res = X*rhat - Y; % calculate residuals
MSEmin = res'*res/200 % b(:,lam) value is 0.1398
```

```
MSEmin =
    0.0088
```

But  $b(:, \lambda_m)$  has only two nonzero components, and therefore can provide better predictive estimates on new data.



## Lasso and Elastic Net with Cross-Validation

Consider predicting the mileage (MPG) of a car based on its weight, displacement, horsepower, and acceleration. The `carbig` data contains these measurements. The data seem likely to be correlated, making elastic net an attractive choice.

- 1 Load the data:

```
load carbig
```

- 2 Extract the continuous (noncategorical) predictors (`lasso` does not handle categorical predictors):

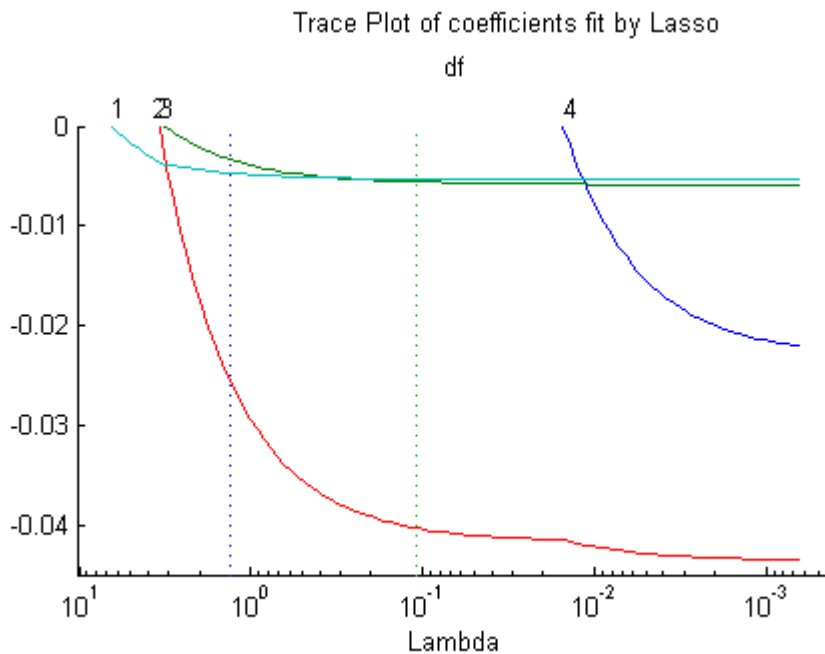
```
X = [Acceleration Displacement Horsepower Weight];
```

- 3 Perform a lasso fit with 10-fold cross validation:

```
[b, fitinfo] = lasso(X,MPG,'CV',10);
```

- 4 Plot the result:

```
lassoPlot(b,fitinfo,'PlotType','Lambda','XScale','log');
```



**5** Calculate the correlation of the predictors:

```
% Eliminate NaNs so corr runs
nonan = ~any(isnan([X MPG]),2);
Xnonan = X(nonan,:);
MPGnonan = MPG(nonan,:);
corr(Xnonan)
```

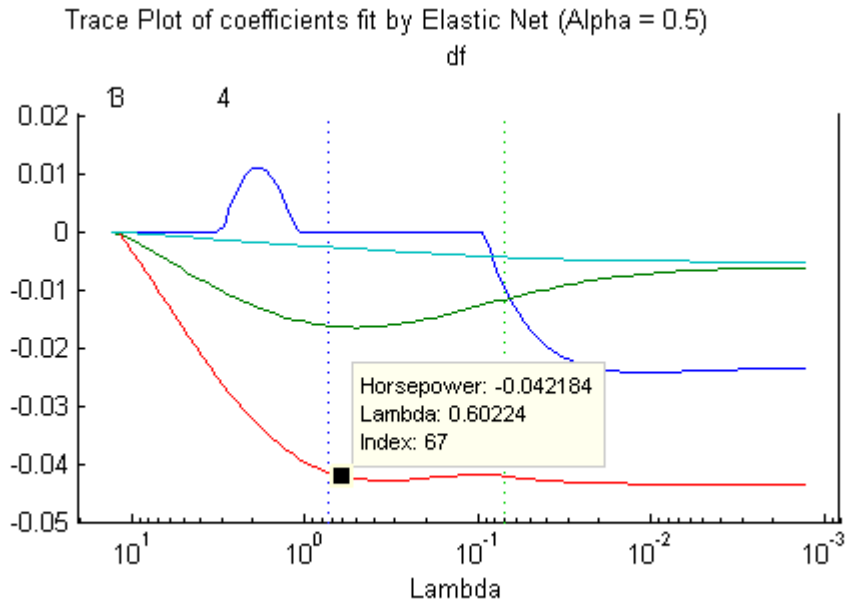
```
ans =
    1.0000    -0.5438    -0.6892    -0.4168
   -0.5438     1.0000     0.8973     0.9330
   -0.6892     0.8973     1.0000     0.8645
   -0.4168     0.9330     0.8645     1.0000
```

**6** Because some predictors are highly correlated, perform elastic net fitting. Use Alpha = 0.5:

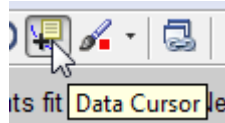
```
[ba fitinfoa] = lasso(X,MPG,'CV',10,'Alpha',.5);
```

7 Plot the result. Name each predictor so you can tell which curve is which:

```
pnames = {'Acceleration','Displacement',...
          'Horsepower','Weight'};
lassoPlot(ba,fitinfoa,'PlotType','Lambda',...
          'XScale','log','PredictorNames',pnames);
```



When you activate the data cursor



and click the plot, you see the name of the predictor, the coefficient, the value of Lambda, and the index of that point, meaning the column in  $b$  associated with that fit.

Here, the elastic net and lasso results are not very similar. Also, the elastic net plot reflects a notable qualitative property of the elastic net technique. The elastic net retains

three nonzero coefficients as  $\Lambda$  increases (toward the left of the plot), and these three coefficients reach 0 at about the same  $\Lambda$  value. In contrast, the lasso plot shows two of the three coefficients becoming 0 at the same value of  $\Lambda$ , while another coefficient remains nonzero for higher values of  $\Lambda$ .

This behavior exemplifies a general pattern. In general, elastic net tends to retain or drop groups of highly correlated predictors as  $\Lambda$  increases. In contrast, lasso tends to drop smaller groups, or even individual predictors.

# Partial Least Squares

<b>In this section...</b>
“Introduction to Partial Least Squares” on page 11-157
“Partial Least Squares” on page 11-158

## Introduction to Partial Least Squares

Partial least-squares (PLS) regression is a technique used with data that contain correlated predictor variables. This technique constructs new predictor variables, known as components, as linear combinations of the original predictor variables. PLS constructs these components while considering the observed response values, leading to a parsimonious model with reliable predictive power.

The technique is something of a cross between multiple linear regression on page 11-11 and principal component analysis on page 15-83:

- Multiple linear regression finds a combination of the predictors that best fit a response.
- Principal component analysis finds combinations of the predictors with large variance, reducing correlations. The technique makes no use of response values.
- PLS finds combinations of the predictors that have a large covariance with the response values.

PLS therefore combines information about the variances of both the predictors and the responses, while also considering the correlations among them.

PLS shares characteristics with other regression and feature transformation on page 15-78 techniques. It is similar to ridge regression on page 11-136 in that it is used in situations with correlated predictors. It is similar to stepwise regression on page 11-127 (or more general feature selection on page 15-73 techniques) in that it can be used to select a smaller set of model terms. PLS differs from these methods, however, by transforming the original predictor space into the new component space.

The Statistics and Machine Learning Toolbox function `plsregress` carries out PLS regression.

## Partial Least Squares

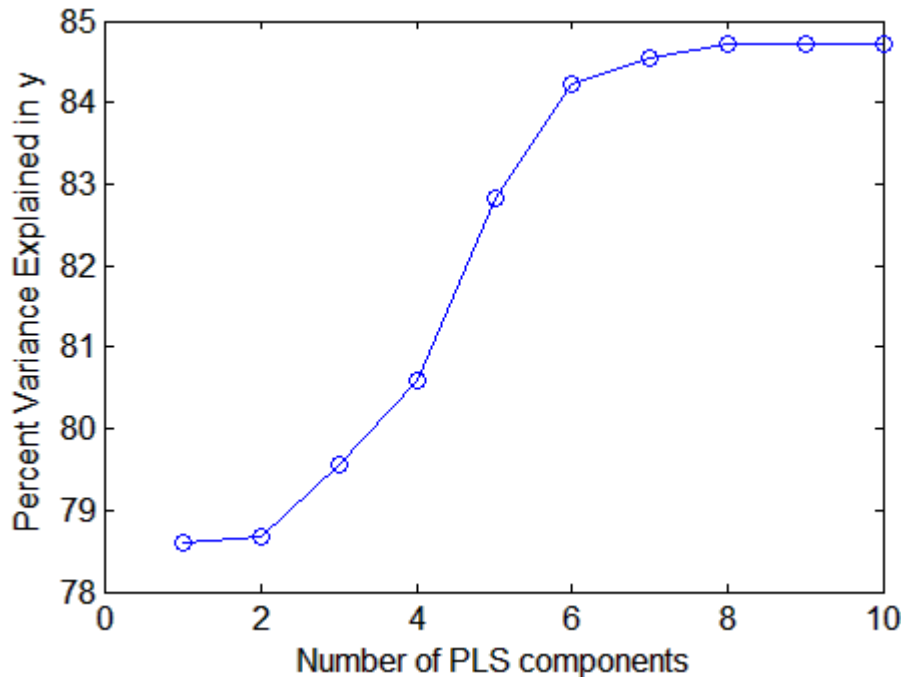
For example, consider the data on biochemical oxygen demand in `moore.mat`, padded with noisy versions of the predictors to introduce correlations:

```
load moore
y = moore(:,6);           % Response
X0 = moore(:,1:5);       % Original predictors
X1 = X0+10*randn(size(X0)); % Correlated predictors
X = [X0,X1];
```

Use `plsregress` to perform PLS regression with the same number of components as predictors, then plot the percentage variance explained in the response as a function of the number of components:

```
[XL,y1,XS,YS,beta,PCTVAR] = plsregress(X,y,10);
```

```
plot(1:10,cumsum(100*PCTVAR(2,:)),'-bo');
xlabel('Number of PLS components');
ylabel('Percent Variance Explained in y');
```

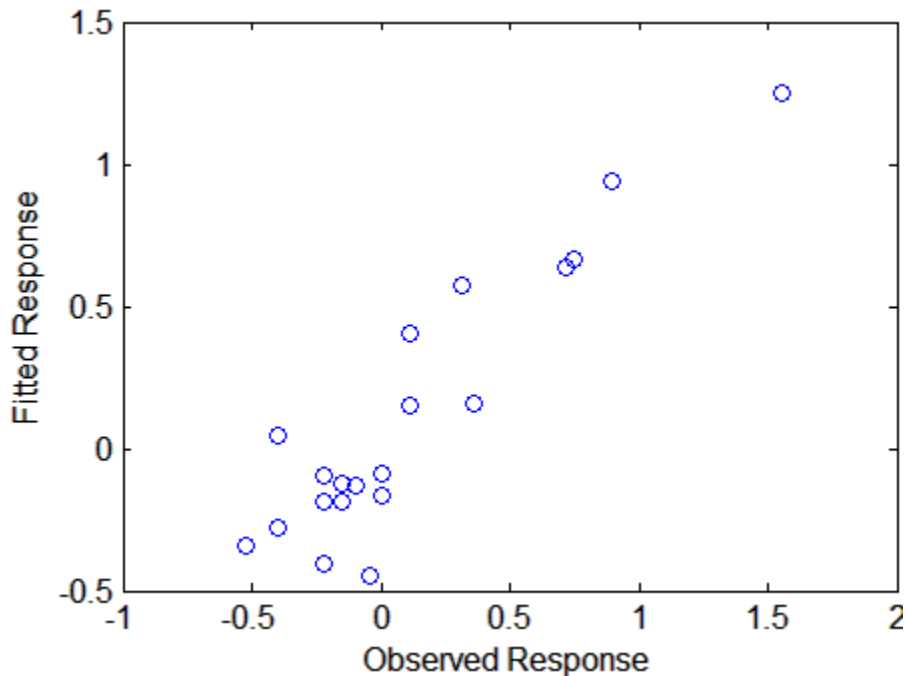


Choosing the number of components in a PLS model is a critical step. The plot gives a rough indication, showing nearly 80% of the variance in  $y$  explained by the first component, with as many as five additional components making significant contributions.

The following computes the six-component model:

```
[XL,y1,XS,YS,beta,PCTVAR,MSE,stats] = plsregress(X,y,6);
yfit = [ones(size(X,1),1) X]*beta;

plot(y,yfit,'o')
```

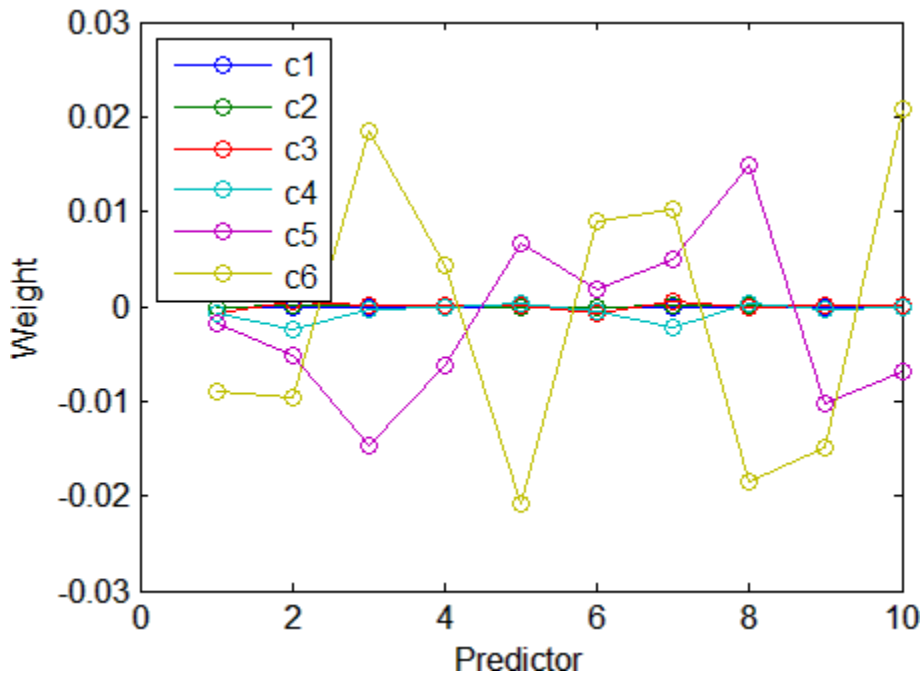


The scatter shows a reasonable correlation between fitted and observed responses, and this is confirmed by the  $R^2$  statistic:

```
TSS = sum((y-mean(y)).^2);
RSS = sum((y-yfit).^2);
Rsquared = 1 - RSS/TSS
Rsquared =
    0.8421
```

A plot of the weights of the ten predictors in each of the six components shows that two of the components (the last two computed) explain the majority of the variance in X:

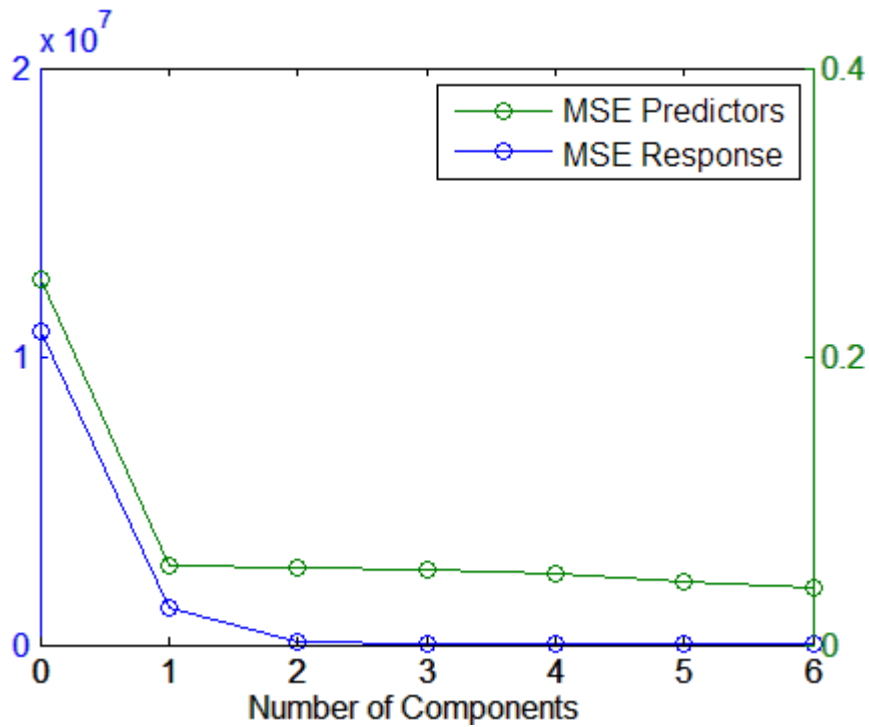
```
plot(1:10,stats.W,'o-');
legend({'c1','c2','c3','c4','c5','c6'},'Location','NW')
xlabel('Predictor');
ylabel('Weight');
```



A plot of the mean-squared errors suggests that as few as two components may provide an adequate model:

```
[axes,h1,h2] = plotyy(0:6,MSE(1,:),0:6,MSE(2,:));
set(h1,'Marker','o')
set(h2,'Marker','o')
legend('MSE Predictors','MSE Response')
xlabel('Number of Components')
```





The calculation of mean-squared errors by `plsregress` is controlled by optional parameter name/value pairs specifying cross-validation type and the number of Monte Carlo repetitions.

## Linear Mixed-Effects Models

Linear mixed-effects models are extensions of linear regression models for data that are collected and summarized in groups. These models describe the relationship between a response variable and independent variables, with coefficients that can vary with respect to one or more grouping variables. A mixed-effects model consists of two parts, fixed effects and random effects. Fixed-effects terms are usually the conventional linear regression part, and the random effects are associated with individual experimental units drawn at random from a population. The random effects have prior distributions whereas fixed effects do not. Mixed-effects models can represent the covariance structure related to the grouping of data by associating the common random effects to observations that have the same level of a grouping variable. The standard form of a linear mixed-effects model is

$$y = \underbrace{X\beta}_{\text{fixed}} + \underbrace{Zb}_{\text{random}} + \underbrace{\varepsilon}_{\text{error}},$$

where

- $y$  is the  $n$ -by-1 response vector, and  $n$  is the number of observations.
- $X$  is an  $n$ -by- $p$  fixed-effects design matrix.
- $\beta$  is a  $p$ -by-1 fixed-effects vector.
- $Z$  is an  $n$ -by- $q$  random-effects design matrix.
- $b$  is a  $q$ -by-1 random-effects vector.
- $\varepsilon$  is the  $n$ -by-1 observation error vector.

The assumptions for the linear mixed-effects model are:

- Random-effects vector,  $b$ , and the error vector,  $\varepsilon$ , have the following prior distributions:

$$b \sim N(0, \sigma^2 D(\theta)),$$

$$\varepsilon \sim N(0, \sigma^2 I),$$

where  $D$  is a symmetric and positive semidefinite matrix, parameterized by a variance component vector  $\theta$ ,  $I$  is an  $n$ -by- $n$  identity matrix, and  $\sigma^2$  is the error variance.

- Random-effects vector,  $b$ , and the error vector,  $\varepsilon$ , are independent from each other.

Mixed-effects models are also called *multilevel models* or *hierarchical models* depending on the context. Mixed-effects models is a more general term than the latter two. Mixed-

effects models might include factors that are not necessarily multilevel or hierarchical, for example crossed factors. That is why mixed-effects is the terminology preferred here. Sometimes mixed-effects models are expressed as multilevel regression models (first level and grouping level models) that are fit simultaneously. For example, a varying or random intercept model, with one continuous predictor variable  $x$  and one grouping variable with  $M$  levels, can be expressed as

$$y_{im} = \beta_{0m} + \beta_1 x_{im} + \varepsilon_{im}, \quad i = 1, 2, \dots, n, \quad m = 1, 2, \dots, M, \quad \varepsilon_{im} \sim N(0, \sigma^2),$$

$$\beta_{0m} = \beta_{00} + b_{0m}, \quad b_{0m} \sim N(0, \sigma_0^2),$$

where  $y_{im}$  corresponds to data for observation  $i$  and group  $m$ ,  $n$  is the total number of observations, and  $b_{0m}$  and  $\varepsilon_{im}$  are independent of each other. After substituting the group-level parameters in the first-level model, the model for the response vector becomes

$$y_{im} = \underbrace{\beta_{00} + \beta_1 x_{im}}_{\text{fixed effects}} + \underbrace{b_{0m}}_{\text{random effects}} + \varepsilon_{im}.$$

A random intercept and slope model with one continuous predictor variable  $x$ , where both the intercept and slope vary independently by a grouping variable with  $M$  levels is

$$y_{im} = \beta_{0m} + \beta_{1m} x_{im} + \varepsilon_{im}, \quad i = 1, 2, \dots, n, \quad m = 1, 2, \dots, M, \quad \varepsilon_{im} \sim N(0, \sigma^2),$$

$$\beta_{0m} = \beta_{00} + b_{0m}, \quad b_{0m} \sim N(0, \sigma_0^2),$$

$$\beta_{1m} = \beta_{10} + b_{1m}, \quad b_{1m} \sim N(0, \sigma_1^2),$$

or

$$b_m = \begin{pmatrix} b_{0m} \\ b_{1m} \end{pmatrix} \sim N\left(0, \begin{pmatrix} \sigma_0^2 & 0 \\ 0 & \sigma_1^2 \end{pmatrix}\right).$$

You might also have correlated random effects. In general, for a model with a random intercept and slope, the distribution of the random effects is

$$b_m = \begin{pmatrix} b_{0m} \\ b_{1m} \end{pmatrix} \sim N(0, \sigma^2 D(\theta)),$$

where  $D$  is a 2-by-2 symmetric and positive semidefinite matrix, parameterized by a variance component vector  $\theta$ .

After substituting the group-level parameters in the first-level model, the model for the response vector is

$$y_{im} = \underbrace{\beta_{00} + \beta_{10}x_{im}}_{\text{fixed effects}} + \underbrace{b_{0m} + b_{1m}x_{im}}_{\text{random effects}} + \varepsilon_{im}, \quad i = 1, 2, \dots, n, \quad m = 1, 2, \dots, M.$$

If you express the group-level variable,  $x_{im}$ , in the random-effects term by  $z_{im}$ , this model is

$$y_{im} = \underbrace{\beta_{00} + \beta_{10}x_{im}}_{\text{fixed effects}} + \underbrace{b_{0m} + b_{1m}z_{im}}_{\text{random effects}} + \varepsilon_{im}, \quad i = 1, 2, \dots, n, \quad m = 1, 2, \dots, M.$$

In this case, the same terms appear in both the fixed-effects design matrix and random-effects design matrix. Each  $z_{im}$  and  $x_{im}$  correspond to the level  $m$  of the grouping variable.

It is also possible to explain more of the group-level variations by adding more group-level predictor variables. A random-intercept and random-slope model with one continuous predictor variable  $x$ , where both the intercept and slope vary independently by a grouping variable with  $M$  levels, and one group-level predictor variable  $v_m$  is

$$y_{im} = \beta_{0im} + \beta_{1im}x_{im} + \varepsilon_{im}, \quad i = 1, 2, \dots, n, \quad m = 1, 2, \dots, M, \quad \varepsilon_{im} \sim N(0, \sigma^2),$$

$$\beta_{0im} = \beta_{00} + \beta_{01}v_{im} + b_{0m}, \quad b_{0m} \sim N(0, \sigma_0^2),$$

$$\beta_{1im} = \beta_{10} + \beta_{11}v_{im} + b_{1m}, \quad b_{1m} \sim N(0, \sigma_1^2).$$

This model results in main effects of the group-level predictor and an interaction term between the first-level and group-level predictor variables in the model for the response variable as

$$\begin{aligned} y_{im} &= \beta_{00} + \beta_{01}v_{im} + b_{0m} + (\beta_{10} + \beta_{11}v_{im} + b_{1m})x_{im} + \varepsilon_{im}, \quad i = 1, 2, \dots, n, \quad m = 1, 2, \dots, M, \\ &= \underbrace{\beta_{00} + \beta_{10}x_{im} + \beta_{01}v_{im} + \beta_{11}v_{im}x_{im}}_{\text{fixed effects}} + \underbrace{b_{0m} + b_{1m}x_{im}}_{\text{random effects}} + \varepsilon_{im}. \end{aligned}$$

The term  $\beta_{11}v_{im}x_{im}$  is often called a cross-level interaction in many textbooks on multilevel models. The model for the response variable  $y$  can be expressed as

$$y_{im} = \begin{bmatrix} 1 & x_{1im} & v_{im} & v_{im}x_{1im} \end{bmatrix} \begin{bmatrix} \beta_{00} \\ \beta_{10} \\ \beta_{01} \\ \beta_{11} \end{bmatrix} + \begin{bmatrix} 1 & x_{1im} \end{bmatrix} \begin{bmatrix} b_{0m} \\ b_{1m} \end{bmatrix} + \varepsilon_{im}, \quad i = 1, 2, \dots, n, \quad m = 1, 2, \dots, M,$$

which corresponds to the standard form given earlier,

$$y = X\beta + Zb + \varepsilon.$$

In general, if there are  $R$  grouping variables, and  $m(r,i)$  shows the level of grouping variable  $r$ , for observation  $i$ , then the model for the response variable for observation  $i$  is

$$y_i = x_i^T \beta + \sum_{r=1}^R z_{ir} b_{m(r,i)}^{(r)} + \varepsilon_i, \quad i = 1, 2, \dots, n,$$

where  $\beta$  is a  $p$ -by-1 fixed-effects vector,  $b_{m(r,i)}^{(r)}$  is a  $q(r)$ -by-1 random-effects vector for the  $r$ th grouping variable and level  $m(r,i)$ , and  $\varepsilon_i$  is a 1-by-1 error term for observation  $i$ .

## References

- [1] Pinheiro, J. C., and D. M. Bates. *Mixed-Effects Models in S and S-PLUS*. Statistics and Computing Series, Springer, 2004.
- [2] Hariharan, S. and J. H. Rogers. "Estimation Procedures for Hierarchical Linear Models." *Multilevel Modeling of Educational Data* (A. A. Connell and D. B. McCoach, eds.). Charlotte, NC: Information Age Publishing, Inc., 2008.
- [3] Hox, J. *Multilevel Analysis, Techniques and Applications*. Lawrence Erlbaum Associates, Inc., 2002.
- [4] Snijders, T. and R. Bosker. *Multilevel Analysis*. Thousand Oaks, CA: Sage Publications, 1999.
- [5] Gelman, A. and J. Hill. *Data Analysis Using Regression and Multilevel/Hierarchical Models*. New York, NY: Cambridge University Press, 2007.

## See Also

`LinearMixedModel` | `fitlme` | `fitlmematrix`

## More About

- "Prepare Data for Linear Mixed-Effects Models" on page 11-166

## Prepare Data for Linear Mixed-Effects Models

### In this section...

“Tables and Dataset Arrays” on page 11-166

“Design Matrices” on page 11-168

“Relation of Matrix Form to Tables and Dataset Arrays” on page 11-170

### Tables and Dataset Arrays

To fit a linear-mixed effects model, you must store your data in a table or dataset array. In your table or dataset array, you must have a column for each variable including the response variable. More specifically, the table or dataset array, say `tbl`, must contain the following:

- A response variable  $y$
- Predictive variables  $X_j$  which can be continuous or grouping variables
- Grouping variables  $g_1, g_2, \dots, g_R$ ,

where the grouping variables in  $X_j$  and  $g_r$  can be categorical, logical, a character array, or a cell array of character vectors,  $r = 1, 2, \dots, R$ .

You must organize your data so that each row represents an observation. And each row should contain the value of variables and the levels of grouping variables corresponding to that observation. For example, if you have data from an experiment with four treatment options, on five different types of individuals chosen randomly from a population of individuals (blocks), the table or dataset array must look like this.

Block	Treatment	Response
1	1	y11
1	2	y12
1	3	y13
1	4	y14
...	...	...
5	1	y51
5	2	y52

Block	Treatment	Response
5	3	y53
5	4	y54

Now, consider a split-plot experiment, where the effect of four different types of fertilizers on the yield of tomato plants is studied. The soil where the tomato plants are planted is divided into three blocks based on the soil type: sandy, silty, and loamy. Each block is divided into five plots, where five types of tomato plants, (cherry, heirloom, grape, vine, and plum) are randomly assigned to these plots. Then, the tomato plants in the plots are divided into subplots, where each subplot is treated by one of the four fertilizers. The data from this experiment looks like:

Soil	Tomato	Fertilizer	Yield
'Sandy'	'Plum'	1	104
'Sandy'	'Plum'	2	136
'Sandy'	'Plum'	3	158
'Sandy'	'Plum'	4	174
'Sandy'	'Cherry'	1	57
'Sandy'	'Cherry'	2	86
...	...	...	...
'Sandy'	'Vine'	3	99
'Sandy'	'Vine'	4	117
'Silty'	'Plum'	1	120
'Silty'	'Plum'	2	115
...	...	...	...
'Loamy'	'Vine'	3	111
'Loamy'	'Vine'	4	105

You must specify the model you want to fit using the `formula` input argument to `fitlme`.

In general, a formula for model specification is a character vector of the form `'y ~ terms'`. For linear mixed-effects models, this formula is in the form `'y ~ fixed + (random1|grouping1) + ... + (randomR|groupingR)'`, where `fixed` contains the fixed-effects terms and `random1`, ..., `randomR` contain the random-effects terms.

For example, for the previous fertilizer experiment, consider the following mixed-effects model

$$y_{im,jk} = \beta_0 + \sum_{m=2}^4 \beta_{1m} I[F]_{im} + \sum_{j=2}^5 \beta_{2j} I[T]_{ij} + b_{0k} S_k + b_{0jk} (S^* T)_{jk} + \varepsilon_{im,jk},$$

where  $i = 1, 2, \dots, 60$ , the index  $m$  corresponds to the fertilizer types,  $j$  corresponds to the tomato types, and  $k = 1, 2, 3$  corresponds to the blocks (soil).  $S_k$  represents the  $k$ th soil type, and  $I[F]_{im}$  is the dummy variable representing level  $m$  of the fertilizer. Similarly,  $I[T]_{ij}$  is the dummy variable representing the level  $j$  of the tomato type.

You can fit this model using the formula 'Yield ~ 1 + Fertilizer + Tomato + (1|Soil)+(1|Soil:Tomato) '.

For detailed information on how to specify your model using formula, see “Relationship Between Formula and Design Matrices” on page 11-172.

## Design Matrices

If you cannot easily describe your model using a formula, you can create design matrices to define the fixed and random effects, and fit the model using `fitlmematrix(X, y, Z, G)`. You must create your design matrices as follows.

Fixed-effects and random-effects design matrices  $X$  and  $Z$ :

- Enter a column of 1s for the intercept using `ones(n, 1)`, where  $n$  is the total number of observations.
- If  $X_1$  is a continuous variable, then enter  $X_1$  as it is in a separate column.
- If  $X_1$  is a categorical variable with  $m$  levels, then there must be  $m - 1$  dummy variables for  $m - 1$  levels of  $X_1$  in  $X$ .

For example, consider an experiment where you want to study the impact of quality of raw materials from four different providers on the productivity of a production line. If you fit a linear mixed-effects model with intercept and provider as the fixed-effects terms, intercept is the random-effects term, and you use reference contrasts coding, then you must construct your fixed- and random-effects design matrices as follows.

```
D = dummyvar(provider); % Create dummy variables
X = [ones(n,1) D(:,2) D(:,3) D(:,4)];
Z = [ones(n,1)];
```



Because reference contrast coding uses the first provider as the reference, and the model has an intercept, you must use the dummy variables for only the last three providers.

- If there is an interaction term of predictor variables `X1` and `X2`, then you must enter a column that you form by elementwise product of the vectors `X1` and `X2`.

For example, if you want to fit a model, where there is an intercept, a continuous treatment factor, a continuous time factor, and their interaction as the fixed-effects in a longitudinal study, and time is the random-effects term, then your fixed- and random-effects design matrices should look like

```
X = [ones(n,1), treatment, time, treatment.*time];
y = response;
Z = [time];
```

Grouping variables `G`:

There is one column for each grouping variable and a column of elementwise product of the grouping variables in case of a nesting.

For example, if you want to group plots (`plot`) within blocks (`block`), then you must add a column of elementwise product of `plot` by `block`. More specifically, if you want to fit a model where there is intercept and a continuous treatment factor as the fixed-effects in a split-block experiment, and the intercept and treatment are grouped by the plots nested within blocks, then the design matrices should look like this.

```
X = [ones(n,1), treatment];
y = response;
Z = [ones(n,1), treatment];
G = [block.*plot];
```

Suppose in the earlier quality of raw materials example, the raw materials arrive in bulks, and the bulks are nested within providers. If you want to fit a linear mixed-effects model, where intercept is grouped by the bulks within providers, then your design matrices should look like this.

```
D = dummyvar(provider);
X = [ones(n,1) D(:,2) D(:,3) D(:,4)];
y = response;
Z = ones(n,1);
G = [provider.*bulks];
```

In the earlier longitudinal study example, if you want to add random effects for intercept and time grouped by subjects that participated in the study, then your design matrices should look like

```
X = [ones(n,1), treatment, time, treatment.*time];
y = response;
Z = [ones(n,1), time];
G = subject;
```

## Relation of Matrix Form to Tables and Dataset Arrays

`fitlme(tbl, formula)` and `fitlmematrix(X, y, Z, G)` are equivalent in functionality, such that

- $y$  is the  $n$ -by-1 response vector.
- $X$  is an  $n$ -by- $p$  fixed-effects design matrix. `fitlme` constructs this from the expression `fixed` in `formula`.
- $Z$  is an  $R$ -by-1 cell array with  $Z\{r\}$  being an  $n$ -by- $q(r)$  random-effects design matrix constructed from the  $r$ th expression in `random` in `formula`,  $r = 1, 2, \dots, R$ .
- $G$  is an  $R$ -by-1 cell array with  $G\{r\}$  being an  $n$ -by-1 grouping variable,  $g_r$ , in `formula` with  $M(r)$  levels or groups.

For example, if `tbl` is a table or dataset array containing the response variable  $y$ , the continuous variables  $X1$  and  $X2$ , and the grouping variable  $g$ , then to fit a linear mixed-effects model that corresponds to the formula expression `'y ~ X1+ X2+ (X1*X2|g)'` using `fitlmematrix(X, y, Z, G)` the input arguments must correspond to the following:

```
y = tbl.y
X = [ones(n,1), tbl.X1, tbl.X2]
Z = [ones(n,1), tbl.X1, tbl.X2, tbl.X1.*tbl.X2]
G = tbl.g
```

## See Also

`LinearMixedModel` | `fitlme` | `fitlmematrix`

## More About

- “Linear Mixed-Effects Models” on page 11-162

- “Relationship Between Formula and Design Matrices” on page 11-172

## Relationship Between Formula and Design Matrices

### In this section...

“Formula” on page 11-172

“Design Matrices for Fixed and Random Effects” on page 11-174

“Grouping Variables” on page 11-176

### Formula

In general, a formula for model specification is a character vector of the form 'y ~ terms'. For the linear mixed-effects models, this formula is in the form 'y ~ fixed + (random1|grouping1) + ... + (randomR|groupingR)', where `fixed` and `random` contain the fixed-effects and the random-effects terms.

Suppose a table `tbl` contains the following:

- A response variable, `y`
- Predictor variables, `Xj`, which can be continuous or grouping variables
- Grouping variables, `g1`, `g2`, ..., `gR`,

where the grouping variables in `Xj` and `gr` can be categorical, logical, character arrays, or cell arrays of character vectors.

Then, in a formula of the form, 'y ~ fixed + (random<sub>1</sub>|g<sub>1</sub>) + ... + (random<sub>R</sub>|g<sub>R</sub>)', the term `fixed` corresponds to a specification of the fixed-effects design matrix `X`, `random1` is a specification of the random-effects design matrix `Z1` corresponding to grouping variable `g1`, and similarly `randomR` is a specification of the random-effects design matrix `ZR` corresponding to grouping variable `gR`. You can express the `fixed` and `random` terms using Wilkinson notation.

Wilkinson notation describes the factors present in models. The notation relates to factors present in models, not to the multipliers (coefficients) of those factors.

Wilkinson Notation	Factors in Standard Notation
1	Constant (intercept) term
X <sup>k</sup> , where k is a positive integer	X, X <sup>2</sup> , ..., X <sup>k</sup>

Wilkinson Notation	Factors in Standard Notation
X1 + X2	X1, X2
X1*X2	X1, X2, X1.*X2 (elementwise multiplication of X1 and X2)
X1:X2	X1.*X2 only
- X2	Do not include X2
X1*X2 + X3	X1, X2, X3, X1*X2
X1 + X2 + X3 + X1:X2	X1, X2, X3, X1*X2
X1*X2*X3 - X1:X2:X3	X1, X2, X3, X1*X2, X1*X3, X2*X3
X1*(X2 + X3)	X1, X2, X3, X1*X2, X1*X3

Statistics and Machine Learning Toolbox notation always includes a constant term unless you explicitly remove the term using `-1`. Here are some examples for linear mixed-effects model specification.

#### Examples:

Formula	Description
'y ~ X1 + X2'	Fixed effects for the intercept, X1 and X2. This is equivalent to 'y ~ 1 + X1 + X2'.
'y ~ -1 + X1 + X2'	No intercept and fixed effects for X1 and X2. The implicit intercept term is suppressed by including <code>-1</code> .
'y ~ 1 + (1   g1)'	Fixed effects for the intercept plus random effect for the intercept for each level of the grouping variable <code>g1</code> .
'y ~ X1 + (1   g1)'	Random intercept model with a fixed slope.
'y ~ X1 + (X1   g1)'	Random intercept and slope, with possible correlation between them. This is equivalent to 'y ~ 1 + X1 + (1 + X1   g1)'.
'y ~ X1 + (1   g1) + (-1 + X1   g1)'	Independent random effects terms for intercept and slope.

Formula	Description
'y ~ 1 + (1   g1) + (1   g2) + (1   g1:g2)'	Random intercept model with independent main effects for g1 and g2, plus an independent interaction effect.

## Design Matrices for Fixed and Random Effects

`fitlme` converts the expressions in the `fixed` and `random` parts (not grouping variables) of a formula into design matrices as follows:

- Each term in a formula adds one or more columns to the corresponding design matrix.
- A term containing a single continuous variable adds one column to the design matrix.
- A fixed term containing a categorical variable  $X$  with  $k$  levels adds  $(k - 1)$  dummy variables to the design matrix.

For example, if the variable `Supplier` represents three different suppliers a manufacturer receives parts from, i.e. a categorical variable with three levels, and out of six batches of parts, the first two batches come from supplier 1 (level 1), the second two batches come from supplier 2 (level 2), and the last two batches come from supplier 3 (level 3), such as

```
Supplier =
```

```
  1
  1
  2
  2
  3
  3
```

Then, adding `Supplier` to the formula as a fixed-effects or random-effects term adds the following two dummy variables to the corresponding design matrix, using the `'reference'` contrast:

```
  0  0
  0  0
  1  0
  1  0
  0  1
  0  1
```

For more details on dummy variables, see “Dummy Indicator Variables” on page 2-62. For other contrast options, see the 'DummyVarCoding' name-value pair argument of `fitlme`.

- If  $X_1$  and  $X_2$  are continuous variables, the product term  $X_1 : X_2$  adds one column obtained by elementwise multiplication of  $X_1$  and  $X_2$  to the design matrix.
- If  $X_1$  is continuous and  $X_2$  is categorical with  $k$  levels, the product term  $X_1 : X_2$  multiplies elementwise  $X_1$  with the  $(k - 1)$  dummy variables representing  $X_2$ , and adds these  $(k - 1)$  columns to the design matrix.

For example, if `Drug` is the amount of a drug given to patients, a continuous treatment, and `Time` is three distinct points in time when the health measures are taken, a categorical variable with three levels, and out of nine observations, the first three are observed at time point 1, the second three are observed at time point 2, and the last three are observed at time point 3 so that

```
[Drug Time] =
    0.1000    1.0000
    0.2000    1.0000
    0.5000    2.0000
    0.6000    2.0000
    0.3000    3.0000
    0.8000    3.0000
```

Then, the product term `Drug:Time` adds the following two variables to the design matrix:

```

    0          0
    0          0
  0.5000      0
  0.6000      0
    0      0.3000
    0      0.8000
```

- If  $X_1$  and  $X_2$  are categorical variables with  $k$  and  $m$  levels respectively, the product term  $X_1 : X_2$  adds  $(k - 1)(m - 1)$  dummy variables to the design matrix formed by taking the elementwise product of each dummy variable representing  $X_1$  with each dummy variable representing  $X_2$ .

For example, in an experiment to determine the impact of the type of corn and the popping method on the yield, suppose there are three types of `Corn` and two types of `Method` as follows:

```
1 oil
1 oil
1 air
1 air
2 oil
2 oil
2 air
2 air
3 oil
3 oil
3 air
3 air
```

Then, the interaction term `Corn:Method` adds the following to the design matrix:

```
0 0
0 0
0 0
0 0
1 0
1 0
0 0
0 0
0 0
0 1
0 1
0 0
0 0
```

- The term  $X1 * X2$  adds the necessary number of columns for  $X1$ ,  $X2$ , and  $X1 : X2$  to the design matrix.
- The term  $X1^2$  adds the necessary number of columns for  $X1$  and  $X1 : X1$  to the design matrix.
- The symbol 1 (one) in the formula stands for a column of all 1s. By default a column of 1s is included in the design matrix. To exclude a column of ones from the design matrix, you must explicitly specify `-1` as a term in the expression.

## Grouping Variables

`fitlme` handles the grouping variables in the `(. | group)` part of a formula as follows:

- If a grouping variable has  $k$  levels, then  $k$  dummy variables represent this grouping.



For example, suppose `District` is a categorical grouping variable with three levels, showing the three types of districts, and out of six schools, the first two are in district 1, the second two are in district 2, and the last two are in district 3, so that

`District =`

```
1
1
2
2
3
3
```

Then, the dummy variables that represent this grouping are:

```
1 0 0
1 0 0
0 1 0
0 1 0
0 0 1
0 0 1
```

- If  $X_1$  is a continuous random-effects variable and  $X_2$  is a grouping variable with  $k$  levels, then the random term  $(X_1 - 1|X_2)$  multiplies elementwise  $X_1$  with the  $k$  dummy variables representing  $X_2$  and adds these  $k$  columns to the random-effects design matrix.

For example, suppose `Score` is a continuous variable showing the scores of students from a math exam in a school, and `Class` is a categorical variable with three levels, showing the three different classes in a school. Also, suppose out of nine observations first three correspond to the scores of students in the first class, the second three correspond to scores of students in the second class, and the last three correspond to the scores of students in the third class, such as

`[Score Class] =`

```
78.0000 1.0000
68.0000 1.0000
81.0000 2.0000
53.0000 2.0000
85.0000 3.0000
72.0000 3.0000
```

Then, the random term  $(\text{Score} - 1|\text{Class})$  adds the following three columns to the random-effects design matrix:

```

78.0000      0      0
68.0000      0      0
      0     81.0000      0
      0     53.0000      0
      0      0     85.0000
      0      0     72.0000

```

- If  $X_1$  is a continuous predictor variable and  $X_2$  and  $X_3$  are grouping variables with  $k$  and  $m$  levels respectively, the term  $(X_1 | X_2 : X_3)$  represents this grouping of  $X_1$  with  $k*m$  dummy variables formed by taking the elementwise product of each dummy variable representing  $X_2$  with each dummy variable representing  $X_3$ .

For example, suppose `Treatment` is a continuous predictor variable, and there are three levels of `Block` and two levels of `Plot` nested within the block as follows:

```

0.1000      1      a
0.2000      1      b
0.5000      2      a
0.6000      2      b
0.3000      3      a
0.8000      3      b

```

Then, the random term  $(\text{Treatment} - 1 | \text{Block} : \text{Plot})$  adds the following to the random-effects design matrix:

```

0.1000      0      0      0      0      0
      0     0.2000      0      0      0      0
      0      0     0.5000      0      0      0
      0      0      0     0.6000      0      0
      0      0      0      0     0.3000      0
      0      0      0      0      0     0.8000

```

## See Also

`LinearMixedModel` | `fitlme` | `fitlmematrix`

## More About

- “Prepare Data for Linear Mixed-Effects Models” on page 11-166

## Estimating Parameters in Linear Mixed-Effects Models

### In this section...

“Maximum Likelihood (ML)” on page 11-179

“Restricted Maximum Likelihood (REML)” on page 11-181

A linear mixed-effects model is of the form

$$y = \underbrace{X\beta}_{\text{fixed}} + \underbrace{Zb}_{\text{random}} + \underbrace{\varepsilon}_{\text{error}},$$

where

- $y$  is the  $n$ -by-1 response vector, and  $n$  is the number of observations.
- $X$  is an  $n$ -by- $p$  fixed-effects design matrix.
- $\beta$  is a  $p$ -by-1 fixed-effects vector.
- $Z$  is an  $n$ -by- $q$  random-effects design matrix.
- $b$  is a  $q$ -by-1 random-effects vector.
- $\varepsilon$  is the  $n$ -by-1 observation error vector.

The random-effects vector,  $b$ , and the error vector,  $\varepsilon$ , are assumed to have the following independent prior distributions:

$$b \sim N(0, \sigma^2 D(\theta)),$$

$$\varepsilon \sim N(0, \sigma^2 I),$$

where  $D$  is a symmetric and positive semidefinite matrix, parameterized by a variance component vector  $\theta$ ,  $I$  is an  $n$ -by- $n$  identity matrix, and  $\sigma^2$  is the error variance.

In this model, the parameters to estimate are the fixed-effects coefficients  $\beta$ , and the variance components  $\theta$  and  $\sigma^2$ . The two most commonly used approaches to parameter estimation in linear mixed-effects models are maximum likelihood and restricted maximum likelihood methods.

### Maximum Likelihood (ML)

The maximum likelihood estimation includes both regression coefficients and the variance components, that is, both fixed-effects and random-effects terms in the likelihood function.

For a linear mixed-effects model defined above, the conditional response of the response variable  $y$  given  $\beta$ ,  $b$ ,  $\theta$ , and  $\sigma^2$  is

$$y | b, \beta, \theta, \sigma^2 \sim N(X\beta + Zb, \sigma^2 I_n).$$

The likelihood of  $y$  given  $\beta$ ,  $\theta$ , and  $\sigma^2$  is

$$P(y | \beta, \theta, \sigma^2) = \int P(y | b, \beta, \theta, \sigma^2) P(b | \theta, \sigma^2) db,$$

where

$$P(b | \theta, \sigma^2) = \frac{1}{(2\pi\sigma^2)^{q/2}} \frac{1}{|D(\theta)|^{1/2}} \exp\left\{-\frac{1}{2\sigma^2} b^T D^{-1} b\right\} \quad \text{and}$$

$$P(y | b, \beta, \theta, \sigma^2) = \frac{1}{(2\pi\sigma^2)^{n/2}} \exp\left\{-\frac{1}{2\sigma^2} (y - X\beta - Zb)^T (y - X\beta - Zb)\right\}.$$

Suppose  $\Delta(\theta)$  is the lower triangular Cholesky factor of  $D(\theta)$  and  $\Delta(\theta)$  is the inverse of  $\Delta(\theta)$ . Then,

$$D(\theta)^{-1} = \Delta(\theta)^T \Delta(\theta).$$

Define

$$r^2(\beta, b, \theta) = b^T \Delta(\theta)^T \Delta(\theta) b + (y - X\beta - Zb)^T (y - X\beta - Zb),$$

and suppose  $b^*$  is the value of  $b$  that satisfies

$$\left. \frac{\partial r^2(\beta, b, \theta)}{\partial b} \right|_{b^*} = 0$$

for given  $\beta$  and  $\theta$ . Then, the likelihood function is

$$P(y | \beta, \theta, \sigma^2) = (2\pi\sigma^2)^{-n/2} |D(\theta)|^{-1/2} \exp\left\{-\frac{1}{2\sigma^2} r^2(\beta, b^*(\beta), \theta)\right\} \frac{1}{|\Delta^T \Delta + Z^T Z|^{1/2}}.$$

$P(y | \beta, \theta, \sigma^2)$  is first maximized with respect to  $\beta$  and  $\sigma^2$  for a given  $\theta$ . Thus the optimized solutions  $\hat{\beta}(\theta)$  and  $\hat{\sigma}^2(\theta)$  are obtained as functions of  $\theta$ . Substituting these solutions

into the likelihood function produces  $P(y | \beta(\theta), \theta, \sigma^2(\theta))$ . This expression is called a profiled likelihood where  $\beta$  and  $\sigma^2$  have been profiled out.  $P(y | \beta(\theta), \theta, \sigma^2(\theta))$  is a function of  $\theta$ , and the algorithm then optimizes it with respect to  $\theta$ . Once it finds the optimal estimate of  $\theta$ , the estimates of  $\beta$  and  $\sigma^2$  are given by  $\hat{\beta}(\theta)$  and  $\hat{\sigma}^2(\theta)$ .

The ML method treats  $\beta$  as fixed but unknown quantities when the variance components are estimated, but does not take into account the degrees of freedom lost by estimating the fixed effects. This causes ML estimates to be biased with smaller variances. However, one advantage of ML over REML is that it is possible to compare two models in terms of their fixed- and random-effects terms. On the other hand, if you use REML to estimate the parameters, you can only compare two models, that are nested in their random-effects terms, with the same fixed-effects design.

## Restricted Maximum Likelihood (REML)

Restricted maximum likelihood estimation includes only the variance components, that is, the parameters that parameterize the random-effects terms in the linear mixed-effects model.  $\beta$  is estimated in a second step. Assuming a uniform improper prior distribution for  $\beta$  and integrating the likelihood  $P(y | \beta, \theta, \sigma^2)$  with respect to  $\beta$  results in the restricted likelihood  $P(y | \theta, \sigma^2)$ . That is,

$$P(y | \theta, \sigma^2) = \int P(y | \beta, \theta, \sigma^2) P(\beta) d\beta = \int P(y | \beta, \theta, \sigma^2) d\beta.$$

The algorithm first profiles out  $\hat{\sigma}_R^2$  and maximizes remaining objective function with respect to  $\theta$  to find  $\hat{\theta}_R$ . The restricted likelihood is then maximized with respect to  $\sigma^2$  to find  $\hat{\sigma}_R^2$ . Then, it estimates  $\beta$  by finding its expected value with respect to the posterior distribution

$$P(\beta | y, \theta_R, \sigma_R^2).$$

REML accounts for the degrees of freedom lost by estimating the fixed effects, and makes a less biased estimation of random effects variances. The estimates of  $\theta$  and  $\sigma^2$  are invariant to the value of  $\beta$  and less sensitive to outliers in the data compared to ML estimates. However, if you use REML to estimate the parameters, you can only compare

two models that have the identical fixed-effects design matrices and are nested in their random-effects terms.

## References

- [1] Pinheiro, J. C., and D. M. Bates. *Mixed-Effects Models in S and S-PLUS*. Statistics and Computing Series, Springer, 2004.
- [2] Hariharan, S. and J. H. Rogers. “Estimation Procedures for Hierarchical Linear Models.” *Multilevel Modeling of Educational Data* (A. A. Connell and D. B. McCoach, eds.). Charlotte, NC: Information Age Publishing, Inc., 2008.
- [3] Raudenbush, S. W. and A. S. Bryk. *Hierarchical Linear Models: Applications and Data Analysis Methods*, 2nd ed. Thousand Oaks, CA: Sage Publications, 2002.
- [4] Hox, J. *Multilevel Analysis, Techniques and Applications*. Lawrence Erlbaum Associates, Inc, 2002.
- [5] Snijders, T. and R. Bosker. *Multilevel Analysis*. Thousand Oaks, CA: Sage Publications, 1999.
- [6] McCulloch, C.E., R. S. Shayle, and J. M. Neuhaus. *Generalized, Linear, and Mixed Models*. Wiley, 2008.

## See Also

`LinearMixedModel` | `fitlme` | `fitlmematrix`

## More About

- “Linear Mixed-Effects Models” on page 11-162

## Linear Mixed-Effects Model Workflow

This example shows how to fit and analyze a linear mixed-effects model (LME).

### Load the sample data.

```
load flu
```

The `flu` dataset array has a `Date` variable, and 10 variables containing estimated influenza rates (in 9 different regions, estimated from Google® searches, plus a nationwide estimate from the CDC).

### Reorganize and plot the data.

To fit a linear-mixed effects model, your data must be in a properly formatted dataset array. To fit a linear mixed-effects model with the influenza rates as the responses, combine the nine columns corresponding to the regions into an array. The new dataset array, `flu2`, must have the response variable `FluRate`, the nominal variable `Region` that shows which region each estimate is from, the nationwide estimate `WtdILLI`, and the grouping variable `Date`.

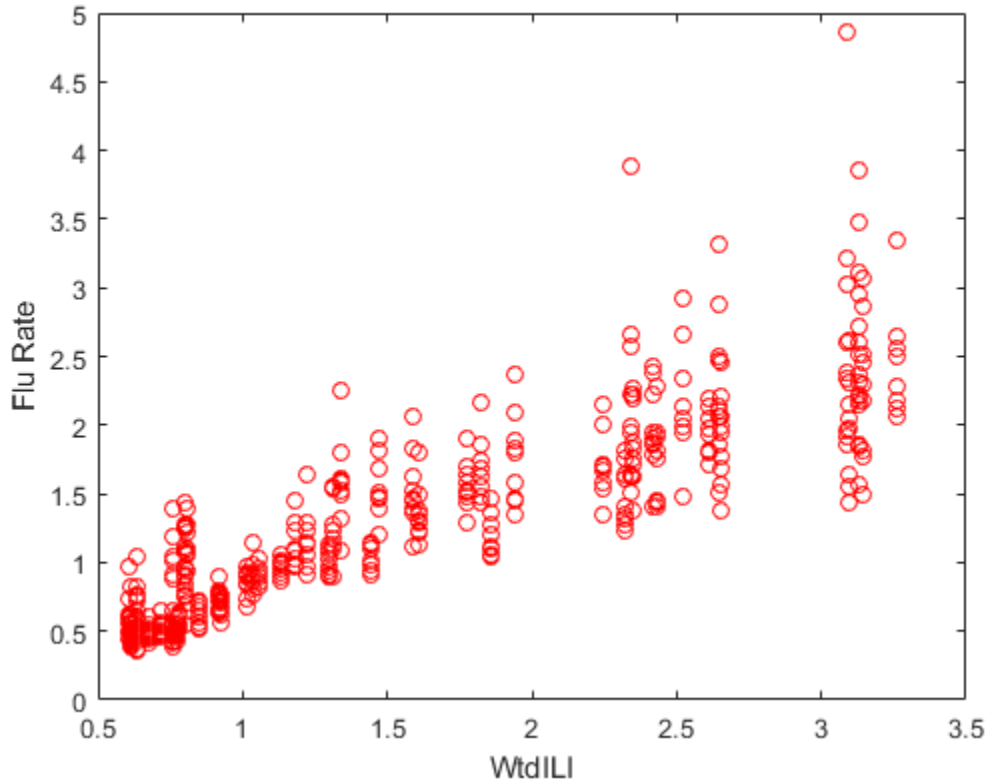
```
flu2 = stack(flu,2:10,'NewDataVarName','FluRate',...  
            'IndVarName','Region');  
flu2.Date = nominal(flu2.Date);
```

Define `flu2` as a table.

```
flu2 = dataset2table(flu2);
```

Plot flu rates versus the nationwide estimate.

```
plot(flu2.WtdILLI,flu2.FluRate,'ro')  
xlabel('WtdILLI')  
ylabel('Flu Rate')
```



You can see that the flu rates in regions have a direct relationship with the nationwide estimate.

**Fit an LME model and interpret the results.**

Fit a linear mixed-effects model with the nationwide estimate as the predictor variable and a random intercept that varies by Date.

```
lme = fitlme(flu2, 'FluRate ~ 1 + WtdILI + (1|Date)')
```

```
lme =
```



Linear mixed-effects model fit by ML

Model information:

Number of observations	468
Fixed effects coefficients	2
Random effects coefficients	52
Covariance parameters	2

Formula:

FluRate ~ 1 + WtdILI + (1 | Date)

Model fit statistics:

AIC	BIC	LogLikelihood	Deviance
286.24	302.83	-139.12	278.24

Fixed effects coefficients (95% CIs):

Name	Estimate	SE	tStat	DF	pValue
'(Intercept)'	0.16385	0.057525	2.8484	466	0.0045885
'WtdILI'	0.7236	0.032219	22.459	466	3.0502e-76

Lower	Upper
0.050813	0.27689
0.66028	0.78691

Random effects covariance parameters (95% CIs):

Group: Date (52 Levels)

Name1	Name2	Type	Estimate	Lower
'(Intercept)'	'(Intercept)'	'std'	0.17146	0.13227

Upper
0.22226

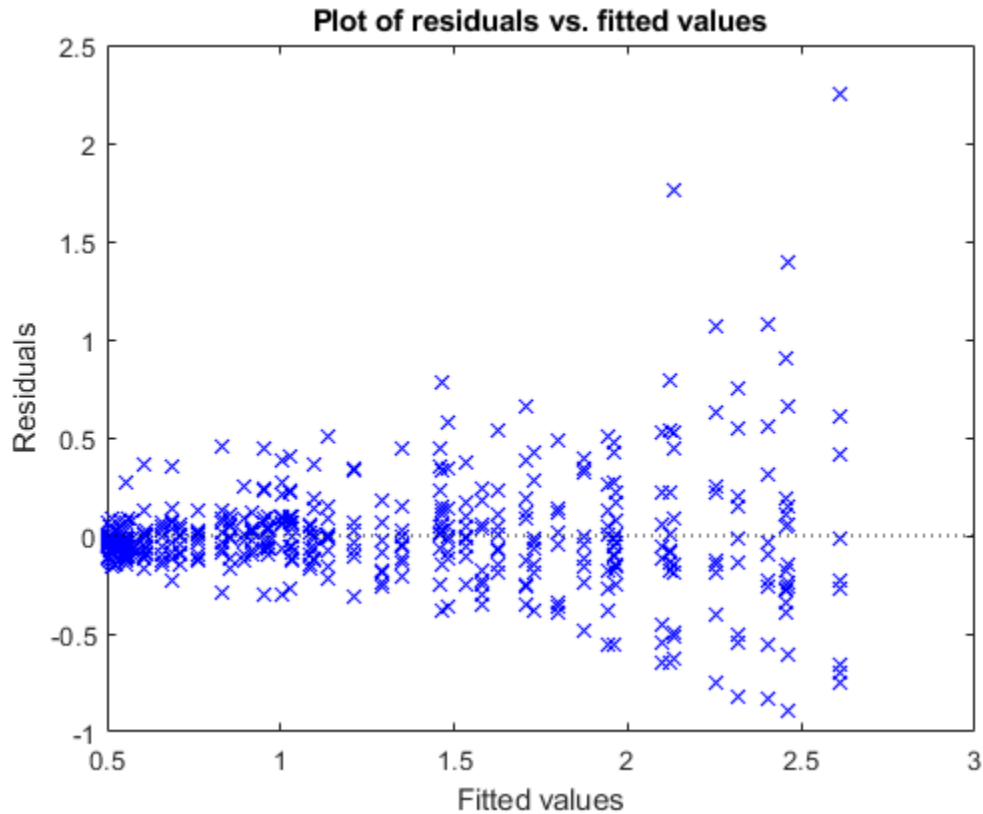
Group: Error

Name	Estimate	Lower	Upper
'Res Std'	0.30201	0.28217	0.32324

The small *P*-values of 0.0045885 and 3.0502e-76 indicate that both the intercept and nationwide estimate are significant. Also, the confidence limits for the standard deviation of the random-effects term,  $\sigma_b$ , do not include 0 (0.13227, 0.22226), which indicates that the random-effects term is significant.

Plot the raw residuals versus the fitted values.

```
figure();  
plotResiduals(lme, 'fitted')
```



The variance of residuals increases with increasing fitted response values, which is known as heteroscedasticity.

Find the two observations on the top right that appear like outliers.

```
find(residuals(lme) > 1.5)
```

```
ans =
```

98  
107

Refit the model by removing these observations.

```
lme = fitlme(flu2, 'FluRate ~ 1 + WtdILI + (1|Date)', 'Exclude', [98,107]);
```

### Improve the model.

Determine if including an independent random term for the nationwide estimate grouped by Date improves the model.

```
altlme = fitlme(flu2, 'FluRate ~ 1 + WtdILI + (1|Date) + (WtdILI-1|Date)', ...  
'Exclude', [98,107])
```

```
altlme =
```

Linear mixed-effects model fit by ML

Model information:

Number of observations	466
Fixed effects coefficients	2
Random effects coefficients	104
Covariance parameters	3

Formula:

```
FluRate ~ 1 + WtdILI + (1 | Date) + (WtdILI | Date)
```

Model fit statistics:

AIC	BIC	LogLikelihood	Deviance
179.39	200.11	-84.694	169.39

Fixed effects coefficients (95% CIs):

Name	Estimate	SE	tStat	DF	pValue
'(Intercept)'	0.17837	0.054585	3.2676	464	0.001165
'WtdILI'	0.70836	0.030594	23.153	464	2.123e-79

Lower	Upper
0.0711	0.28563
0.64824	0.76849

```

Random effects covariance parameters (95% CIs):
Group: Date (52 Levels)
  Name1          Name2          Type          Estimate    Lower
  '(Intercept)' '(Intercept)'    'std'        0.16631    0.12977

  Upper
  0.21313

Group: Date (52 Levels)
  Name1          Name2          Type          Estimate    Lower    Upper
  'WtdILI'       'WtdILI'       'std'        4.6147e-08    NaN     NaN

Group: Error
  Name          Estimate    Lower    Upper
  'Res Std'     0.26691    0.24934    0.28572

```

The estimated standard deviation of WtdILI term is nearly 0 and its confidence interval cannot be computed. This is an indication that the model is overparameterized and the (WtdILI-1|Date) term is not significant. You can formally test this using the compare method as follows: `compare(lme, altlme, 'CheckNesting', true)`.

Add a random effects-term for intercept grouped by Region to the initial model lme.

```

lme2 = fitlme(flu2, 'FluRate ~ 1 + WtdILI + (1|Date) + (1|Region)', ...
'Exclude', [98,107]);

```

Compare the models lme and lme2.

```

compare(lme, lme2, 'CheckNesting', true)

```

```

ans =

```

THEORETICAL LIKELIHOOD RATIO TEST

Model	DF	AIC	BIC	LogLik	LRStat	deltaDF	pValue
lme	4	177.39	193.97	-84.694			
lme2	5	62.265	82.986	-26.133	117.12	1	0

The *P*-value of 0 indicates that lme2 is a better fit than lme.

Now, check if adding a potentially correlated random-effects term for the intercept and national average improves the model `lme2`.

```
lme3 = fitlme(flu2, 'FluRate ~ 1 + WtdILI + (1|Date) + (1 + WtdILI|Region)', ...
'Exclude', [98,107])
```

```
lme3 =
```

```
Linear mixed-effects model fit by ML
```

```
Model information:
```

Number of observations	466
Fixed effects coefficients	2
Random effects coefficients	70
Covariance parameters	5

```
Formula:
```

```
FluRate ~ 1 + WtdILI + (1 | Date) + (1 + WtdILI | Region)
```

```
Model fit statistics:
```

AIC	BIC	LogLikelihood	Deviance
13.338	42.348	0.33076	-0.66153

```
Fixed effects coefficients (95% CIs):
```

Name	Estimate	SE	tStat	DF	pValue
'(Intercept)'	0.1795	0.054953	3.2665	464	0.0011697
'WtdILI'	0.70719	0.04252	16.632	464	4.6451e-49

Lower	Upper
0.071514	0.28749
0.62363	0.79074

```
Random effects covariance parameters (95% CIs):
```

```
Group: Date (52 Levels)
```

Name1	Name2	Type	Estimate	Lower
'(Intercept)'	'(Intercept)'	'std'	0.17634	0.14093

Upper
0.22064

```
Group: Region (9 Levels)
```

Name1	Name2	Type	Estimate
'(Intercept)'	'(Intercept)'	'std'	0.0077038
'WtdILI'	'(Intercept)'	'corr'	-0.059603
'WtdILI'	'WtdILI'	'std'	0.088069

Lower	Upper
3.1487e-16	1.8848e+11
-0.99996	0.99995
0.051694	0.15004

Group: Error

Name	Estimate	Lower	Upper
'Res Std'	0.20976	0.19568	0.22486

The estimate for the standard deviation of the random-effects term for intercept grouped by Region is 0.0077037, its confidence interval is very large and includes zero. This indicates that the random-effects for intercept grouped by Region is insignificant. The correlation between the random-effects for intercept and `WtdILI` is -0.059604. Its confidence interval is also very large and includes zero. This is an indication that the correlation is not significant.

Refit the model by eliminating the intercept from the `(1 + WtdILI | Region)` random-effects term.

```
lme3 = fitlme(flu2, 'FluRate ~ 1 + WtdILI + (1|Date) + (WtdILI - 1|Region)', ...
'Exclude', [98,107])
```

```
lme3 =
```

Linear mixed-effects model fit by ML

Model information:

Number of observations	466
Fixed effects coefficients	2
Random effects coefficients	61
Covariance parameters	3

Formula:

```
FluRate ~ 1 + WtdILI + (1 | Date) + (WtdILI | Region)
```

Model fit statistics:

AIC	BIC	LogLikelihood	Deviance
9.3395	30.06	0.33023	-0.66046

Fixed effects coefficients (95% CIs):

Name	Estimate	SE	tStat	DF	pValue
'(Intercept)'	0.1795	0.054892	3.2702	464	0.0011549
'WtdILI'	0.70718	0.042486	16.645	464	4.0496e-49

Lower	Upper
0.071637	0.28737
0.62369	0.79067

Random effects covariance parameters (95% CIs):

Group: Date (52 Levels)

Name1	Name2	Type	Estimate	Lower
'(Intercept)'	'(Intercept)'	'std'	0.17633	0.14092

Upper  
0.22062

Group: Region (9 Levels)

Name1	Name2	Type	Estimate	Lower	Upper
'WtdILI'	'WtdILI'	'std'	0.087925	0.054474	0.14192

Group: Error

Name	Estimate	Lower	Upper
'Res Std'	0.20979	0.19585	0.22473

**All terms in the new model lme3 are significant.**

**Compare lme2 and lme3.**

```
compare(lme2,lme3,'CheckNesting',true,'NSim',100)
```

ans =

SIMULATED LIKELIHOOD RATIO TEST: NSIM = 100, ALPHA = 0.05

Model	DF	AIC	BIC	LogLik	LRStat	pValue
-------	----	-----	-----	--------	--------	--------

```

lme2      5      62.265      82.986      -26.133
lme3      5      9.3395      30.06      0.33023      52.926      0.009901

Lower      Upper
0.00025064  0.053932

```

The *P*-value of 0.009901 indicates that lme3 is a better fit than lme2.

Add a quadratic fixed-effects term to the model lme3.

```

lme4 = fitlme(flu2, 'FluRate ~ 1 + WtdILI^2 + (1|Date) + (WtdILI - 1|Region)', ...
'Exclude', [98,107])

```

```

lme4 =

```

```

Linear mixed-effects model fit by ML

```

```

Model information:

```

```

Number of observations      466
Fixed effects coefficients      3
Random effects coefficients    61
Covariance parameters        3

```

```

Formula:

```

```

FluRate ~ 1 + WtdILI + WtdILI^2 + (1 | Date) + (WtdILI | Region)

```

```

Model fit statistics:

```

```

AIC      BIC      LogLikelihood      Deviance
6.7234   31.588   2.6383      -5.2766

```

```

Fixed effects coefficients (95% CIs):

```

```

Name      Estimate      SE      tStat      DF      pValue
'(Intercept)'      -0.063406      0.12236      -0.51821      463      0.60456
'WtdILI'      1.0594      0.16554      6.3996      463      3.8232e-10
'WtdILI^2'      -0.096919      0.0441      -2.1977      463      0.028463

```

```

Lower      Upper
-0.30385   0.17704
0.73406    1.3847

```



-0.18358      -0.010259

Random effects covariance parameters (95% CIs):

Group: Date (52 Levels)

Name1	Name2	Type	Estimate	Lower
'(Intercept)'	'(Intercept)'	'std'	0.16732	0.13326

Upper  
0.21009

Group: Region (9 Levels)

Name1	Name2	Type	Estimate	Lower	Upper
'WtdILI'	'WtdILI'	'std'	0.087865	0.054443	0.1418

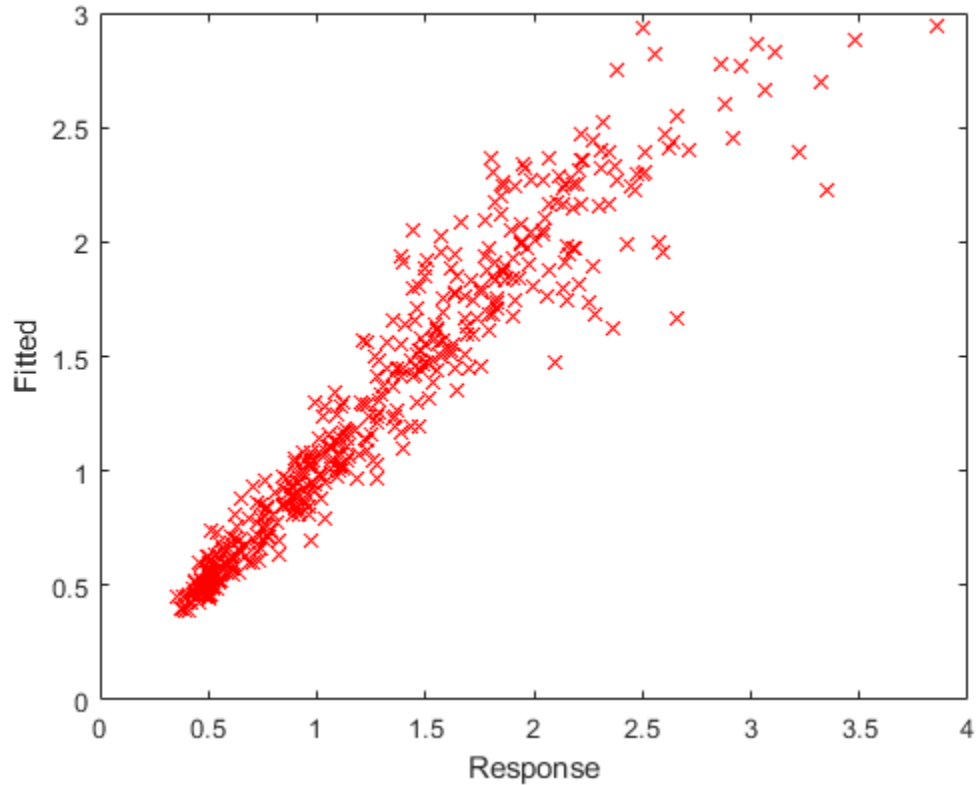
Group: Error

Name	Estimate	Lower	Upper
'Res Std'	0.20979	0.19585	0.22473

The *P*-value of 0.028463 indicates that the coefficient of the quadratic term  $WtdILI^2$  is significant.

**Plot the fitted response versus the observed response and residuals.**

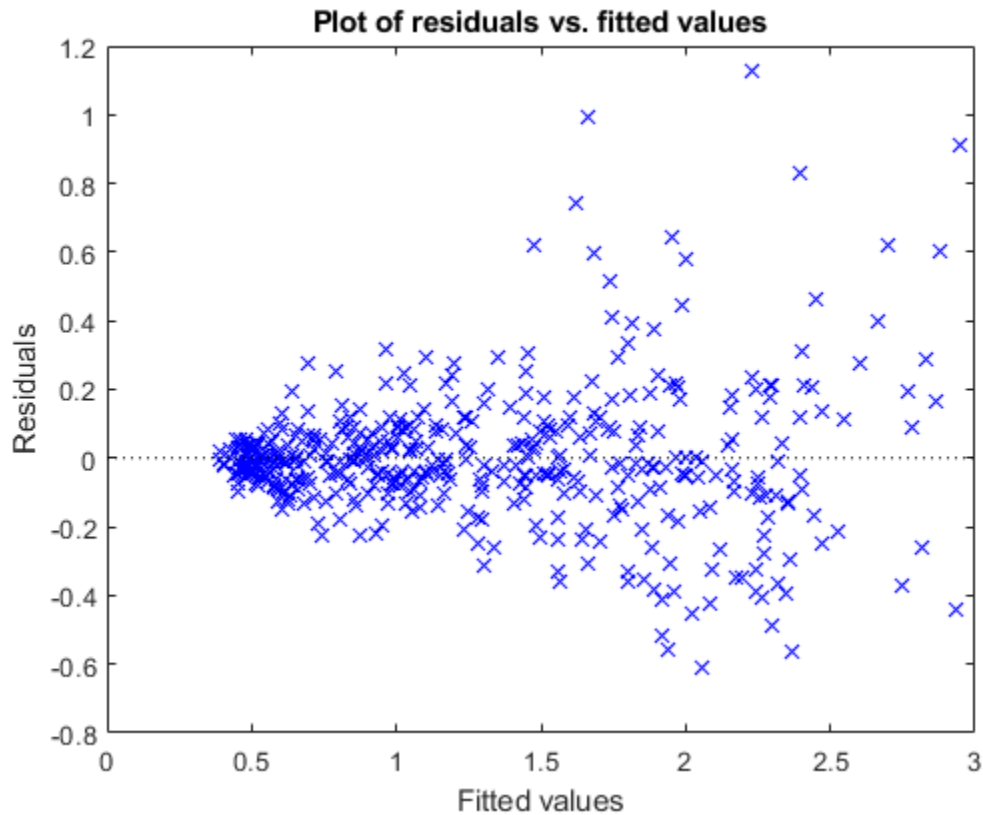
```
F = fitted(lme4);
R = response(lme4);
figure();
plot(R,F,'rx')
xlabel('Response')
ylabel('Fitted')
```



The fitted versus observed response values form almost 45-degree angle indicating a good fit.

Plot the residuals versus the fitted values.

```
figure();  
plotResiduals(lme4, 'fitted')
```



Although it has improved, you can still see some heteroscedasticity in the model. This might be due to another predictor that does not exist in the data set, hence not in the model.

**Find the fitted flu rate value for region ENCentral, date 11/6/2005.**

```
F(flu2.Region == 'ENCentral' & flu2.Date == '11/6/2005')
```

```
ans =
```

```
1.4860
```

**Randomly generate response values.**

Randomly generate response values for a national estimate of 1.625, region MidAtl, and date 4/23/2006. First, define the new table. Because Date and Region are nominal in the original table, you must define them similarly in the new table.

```
tblnew.Date = nominal('4/23/2006');  
tblnew.WtdILI = 1.625;  
tblnew.Region = nominal('MidAtl');  
tblnew = struct2table(tblnew);
```

Now, generate the response value.

```
random(lme4, tblnew)
```

```
ans =
```

```
1.2679
```

## Fit Mixed-Effects Spline Regression

This example shows how to fit a mixed-effects linear spline model.

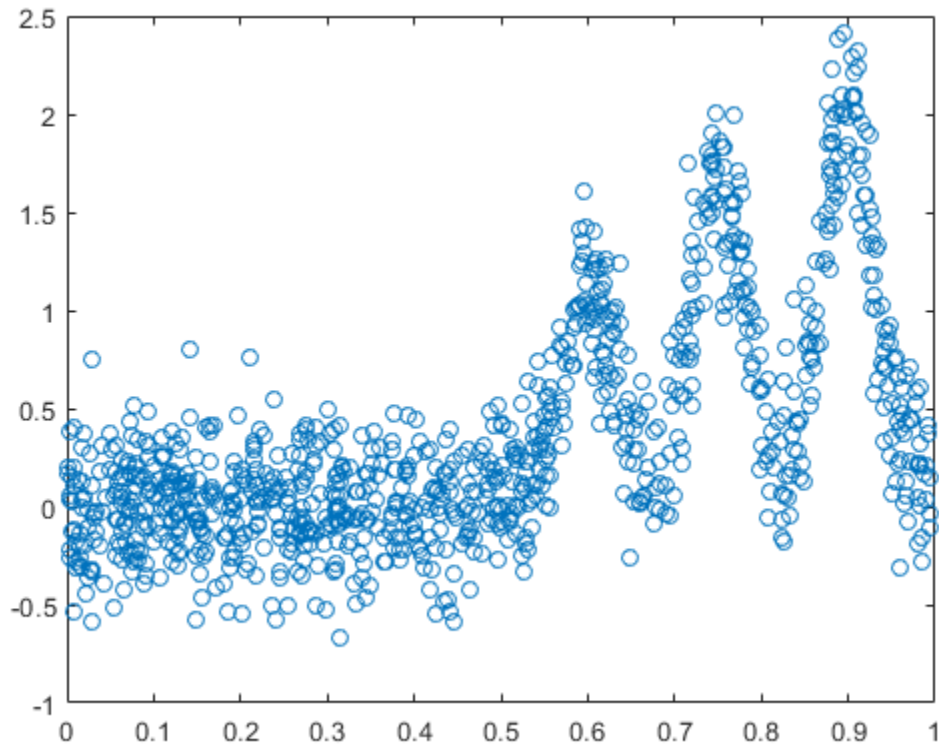
Load the sample data.

```
load(fullfile(matlabroot, 'examples', 'stats', 'mespline.mat'));
```

This is simulated data.

Plot  $y$  versus sorted  $x$ .

```
[x_sorted, I] = sort(x, 'ascend');  
plot(x_sorted, y(I), 'o')
```



Fit the following mixed-effects linear spline regression model

$$y_i = \beta_1 + \beta_2 x_i + \sum_{j=1}^K b_j (x_i - k_j)_+ + \epsilon_i$$

where  $k_j$  is the  $j$ th knot, and  $K$  is the total number of knots. Assume that  $b_j \sim N(0, \sigma_b^2)$  and  $\epsilon \sim N(0, \sigma^2)$ .

Define the knots.

```
k = linspace(0.05,0.95,100);
```

Define the design matrices.

```
X = [ones(1000,1), x];
Z = zeros(length(x), length(k));
for j = 1:length(k)
    Z(:,j) = max(X(:,2) - k(j), 0);
end
```

Fit the model with an isotropic covariance structure for the random effects.

```
lme = fitlmematrix(X,y,Z,[], 'CovariancePattern', 'Isotropic');
```

Fit a fixed-effects only model.

```
X = [X Z];
lme_fixed = fitlmematrix(X,y,[],[]);
```

Compare `lme_fixed` and `lme` via a simulated likelihood ratio test.

```
compare(lme,lme_fixed, 'NSim', 500, 'CheckNesting', true)
```

```
ans =
```

```
SIMULATED LIKELIHOOD RATIO TEST: NSIM = 500, ALPHA = 0.05
```

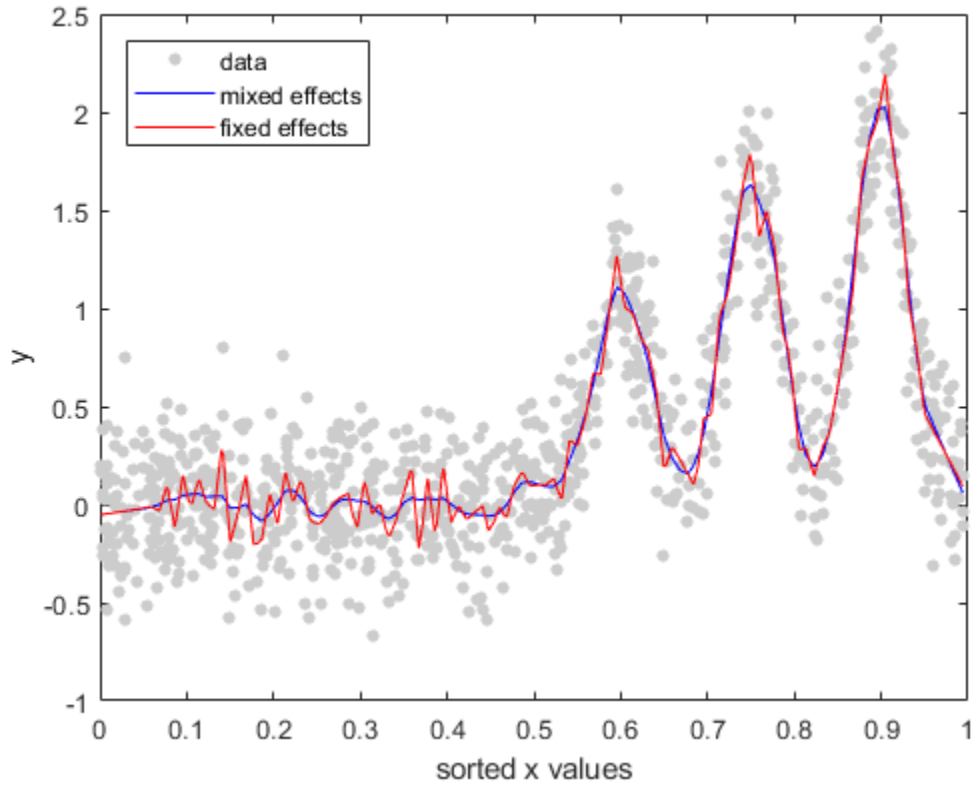
Model	DF	AIC	BIC	LogLik	LRStat	pValue
lme	4	170.62	190.25	-81.309		
lme_fixed	103	113.38	618.88	46.309	255.24	0.68064

Lower	Upper
0.63784	0.72129

The  $P$ -value indicates that the fixed-effects only model is not a better fit than the mixed-effects spline regression model.

Plot the fitted values from both models on top of the original response data.

```
R = response(lme);
figure();
plot(x_sorted,R(I),'o', 'MarkerFaceColor',[0.8,0.8,0.8],...
      'MarkerEdgeColor',[0.8,0.8,0.8],'MarkerSize',4);
hold on
F = fitted(lme);
F_fixed = fitted(lme_fixed);
plot(x_sorted,F(I),'b');
plot(x_sorted,F_fixed(I),'r');
legend('data','mixed effects','fixed effects','Location','NorthWest')
xlabel('sorted x values');
ylabel('y');
hold off
```



You can also see from the figure that the mixed-effects model provides a better fit to data than the fixed-effects only model.



# Generalized Linear Models

---

- “Multinomial Models for Nominal Responses” on page 12-2
- “Multinomial Models for Ordinal Responses” on page 12-5
- “Hierarchical Multinomial Models” on page 12-9
- “Generalized Linear Models” on page 12-12
- “Generalized Linear Model Workflow” on page 12-41
- “Lasso Regularization of Generalized Linear Models” on page 12-46
- “Regularize Poisson Regression” on page 12-49
- “Regularize Logistic Regression” on page 12-53
- “Regularize Wide Data in Parallel” on page 12-61
- “Generalized Linear Mixed-Effects Models” on page 12-67
- “Fit a Generalized Linear Mixed-Effects Model” on page 12-79

## Multinomial Models for Nominal Responses

The outcome of a response variable might be one of a restricted set of possible values. If there are only two possible outcomes, such as a yes or no answer to a question, these responses are called binary responses. If there are multiple outcomes, then they are called polytomous responses. Some examples include the degree of a disease (mild, medium, severe), preferred districts to live in a city, and so on. When the response variable is *nominal*, there is no natural order among the response variable categories. Nominal response models explain and predict the probability that an observation is in each category of a categorical response variable.

A nominal response model is one of several natural extensions of the binary logit model and is also called a *multinomial logit* model. The multinomial logit model explains the relative risk of being in one category versus being in the reference category,  $k$ , using a linear combination of predictor variables. Consequently, the probability of each outcome is expressed as a nonlinear function of  $p$  predictor variables. The 'interactions', 'on' name-value pair argument in `mnrfit` corresponds to this multinomial model with separate intercept and slopes among categories. `mnrfit` uses the default logit link function for multinomial models. You cannot specify a different link function for multinomial responses.

The multinomial logit model is

$$\begin{aligned} \ln\left(\frac{\pi_1}{\pi_k}\right) &= \alpha_1 + \beta_{11}X_1 + \beta_{12}X_2 + \cdots + \beta_{1p}X_p, \\ \ln\left(\frac{\pi_2}{\pi_k}\right) &= \alpha_2 + \beta_{21}X_1 + \beta_{22}X_2 + \cdots + \beta_{2p}X_p, \\ &\vdots \\ \ln\left(\frac{\pi_{k-1}}{\pi_k}\right) &= \alpha_{(k-1)} + \beta_{(k-1)1}X_1 + \beta_{(k-1)2}X_2 + \cdots + \beta_{(k-1)p}X_p, \end{aligned}$$

where  $\pi_j = P(y = j)$  is the probability of an outcome being in category  $j$ ,  $k$  is the number of response categories, and  $p$  is the number of predictor variables. Theoretically, any category can be the reference category, but `mnrfit` chooses the last one,  $k$ , as the reference category. Thus, `mnrfit` assumes the coefficients of the  $k$ th category are zero. The total of  $j - 1$  equations are solved simultaneously to estimate the coefficients. `mnrfit` uses the iteratively weighted least squares algorithm to find the maximum likelihood estimates.

The coefficients in the model express the effects of the predictor variables on the relative risk or the log odds of being in category  $j$  versus the reference category, here  $k$ . For example, the coefficient  $\beta_{23}$  indicates that the probability of the response variable being in category 2 compared to the probability of being in category  $k$  increases  $\exp(\beta_{23})$  times for each unit increase in  $X_3$ , given all else is held constant. Or it indicates that the relative log odds of the response variable being category 2 versus in category  $k$  increases  $\beta_{23}$  times with a one-unit increase in  $X_3$ , given all else equal.

Based on the nominal response model, and the assumption that the coefficients for the last category are zero, the probability of being in each category is

$$\pi_j = P(y = j) = \frac{e^{\alpha_j + \sum_{l=1}^p \beta_{jl} x_l}}{1 + \sum_{j=1}^{k-1} e^{\alpha_j + \sum_{l=1}^p \beta_{jl} x_l}}, \quad j = 1, \dots, k-1.$$

The probability of the  $k$ th category becomes

$$\pi_k = P(y = k) = \frac{1}{1 + \sum_{j=1}^{k-1} e^{\alpha_j + \sum_{l=1}^p \beta_{jl} x_l}},$$

which is simply equal to  $1 - \pi_1 - \pi_2 - \dots - \pi_{k-1}$ .

After estimating the model coefficients using `mnrfit`, you can estimate the category probabilities or the number in each category using `mnrval` (the default name-value pair is 'type', 'category'). This function accepts the coefficient estimates and the model statistics `mnrfit` returns and estimates the categorical probabilities or the number in each category and their confidence bounds. You can also specify the cumulative or conditional probabilities or numbers to estimate using the 'type' name-value pair argument in `mnrval`.

## References

- [1] McCullagh, P., and J. A. Nelder. *Generalized Linear Models*. New York: Chapman & Hall, 1990.
- [2] Long, J. S. *Regression Models for Categorical and Limited Dependent Variables*. Sage Publications, 1997.

[3] Dobson, A. J., and A. G. Barnett. *An Introduction to Generalized Linear Models*. Chapman and Hall/CRC. Taylor & Francis Group, 2008.

### See Also

`fitglm` | `glmfit` | `glmval` | `mnrfit` | `mnrval`

### More About

- “Multinomial Models for Ordinal Responses” on page 12-5
- “Hierarchical Multinomial Models” on page 12-9

## Multinomial Models for Ordinal Responses

The outcome of a response variable might be one of a restricted set of possible values. If there are only two possible outcomes, such as male and female for gender, these responses are called binary responses. If there are multiple outcomes, then they are called polytomous responses. Some examples of polytomous responses include levels of a disease (mild, medium, severe), preferred districts to live in a city, the species for a certain flower type, and so on. Sometimes there might be a natural order among the response categories. These responses are called *ordinal responses*.

The ordering might be inherent in the category choices, such as an individual being not satisfied, satisfied, or very satisfied with an online customer service. The ordering might also be introduced by categorization of a latent (continuous) variable, such as in the case of an individual being in the low risk, medium risk, or high risk group for developing a certain disease, based on a quantitative medical measure such as blood pressure.

You can specify a multinomial regression model that uses the natural ordering among the response categories. This ordinal model describes the relationship between the cumulative probabilities of the categories and predictor variables.

Different link functions can describe this relationship with logit and probit being the most used.

- **Logit:** The default link function `mnrfit` uses for ordinal categories is the *logit* link function. This models the *log cumulative odds*. The 'link', 'logit' name-value pair specifies this in `mnrfit`. Log cumulative odds is the logarithm of the ratio of the probability that a response belongs to a category with a value less than or equal to category  $j$ ,  $P(y \leq c_j)$ , and the probability that a response belongs to a category with a value greater than category  $j$ ,  $P(y > c_j)$ .

Ordinal models are usually based on the assumption that the effects of predictor variables are the same for all categories on the logarithmic scale. That is, the model has different intercepts but common slopes (coefficients) among categories. This model is called *parallel regression* or the *proportional odds* model. It is the default for ordinal responses, and the 'interactions', 'off' name-value pair specifies this model in `mnrfit`.

The proportional odds model is

$$\begin{aligned} \ln\left(\frac{P(y \leq c_1)}{P(y > c_1)}\right) &= \ln\left(\frac{\pi_1}{\pi_2 + \dots + \pi_k}\right) = \alpha_1 + \beta_1 X_1 + \beta_2 X_2 + \dots + \beta_p X_p, \\ \ln\left(\frac{P(y \leq c_2)}{P(y > c_2)}\right) &= \ln\left(\frac{\pi_1 + \pi_2}{\pi_3 + \dots + \pi_k}\right) = \alpha_2 + \beta_1 X_1 + \beta_2 X_2 + \dots + \beta_p X_p, \\ &\vdots \\ \ln\left(\frac{P(y \leq c_{k-1})}{P(y > c_{k-1})}\right) &= \ln\left(\frac{\pi_1 + \pi_2 + \dots + \pi_{k-1}}{\pi_k}\right) = \alpha_{k-1} + \beta_1 X_1 + \beta_2 X_2 + \dots + \beta_p X_p, \end{aligned}$$

where  $\pi_j$ ,  $j = 1, 2, \dots, k$ , are the category probabilities.

For example, for a response variable with three categories, there are  $3 - 1 = 2$  equations as follows:

$$\begin{aligned} \ln\left(\frac{\pi_1}{\pi_2 + \pi_3}\right) &= \alpha_1 + \beta_1 X_1 + \beta_2 X_2 + \dots + \beta_p X_p, \\ \ln\left(\frac{\pi_1 + \pi_2}{\pi_3}\right) &= \alpha_2 + \beta_1 X_1 + \beta_2 X_2 + \dots + \beta_p X_p. \end{aligned}$$

Under the proportional odds assumption, the partial effect of a predictor variable  $X$  is invariant to the choice of the response variable category,  $j$ . For example, if there are three categories, then the coefficients express the impact of a predictor variable on the relative risk or log odds of the response value being in category 1 versus categories 2 or 3, or in category 1 or 2 versus category 3.

Thus, a unit change in variable  $X_2$  would mean a change in the cumulative odds of the response value being in category 1 versus categories 2 or 3, or category 1 or 2 versus category 3 by a factor of  $\exp(\beta_2)$ , given all else equal.

You can alternatively fit a model with different intercept and slopes among the categories by using the 'interactions', 'on' name-value pair argument. However, using this option for ordinal models when the equal slopes model is true causes a loss of efficiency (you lose the advantage of estimating fewer parameters).

- **Probit:** The 'link', 'probit' name-value pair argument uses the *probit* link function which is based on a normally distributed latent variable assumption. For ordinal response variables this is also called an *ordered probit* model. Consider the regression model that describes the relationship of a latent variable  $y^*$  of an ordinal process and a vector of predictor variables,  $X$ ,

$$y^* = \beta X + \varepsilon,$$

where the error term  $\varepsilon$  has a standard normal distribution. Suppose there is the following relationship between the latent variable  $y^*$  and the observed variable  $y$ :

$$\begin{aligned} y = c_1 & \text{ if } \alpha_0 < y^* \leq \alpha_1, \\ y = c_2 & \text{ if } \alpha_1 < y^* \leq \alpha_2, \\ & \vdots \\ y = c_k & \text{ if } \alpha_{k-1} < y^* \leq \alpha_k, \end{aligned}$$

where  $\alpha_0 = -\infty$  and  $\alpha_k = \infty$ . Then, the cumulative probability of  $y$  being in category  $j$  or one of earlier categories,  $P(y \leq c_j)$ , is equal to

$$P(y \leq c_j) = P(y^* < \alpha_j) = P(\beta X + \varepsilon < \alpha_j) = P(\varepsilon < \alpha_j - \beta X) = \Phi(\alpha_j - \beta X),$$

where  $\Phi$  is standard normal cumulative distribution function. Thus,

$$\Phi^{-1}(P(y \leq c_j)) = \alpha_j - \beta X,$$

where  $\alpha_j$  corresponds to the cut points of the latent variable and the intercept in the regression model. This only holds under the assumptions of a normal latent variable and parallel regression. More generally, for a response variable with  $k$  categories and multiple predictors, the ordered probit model is

$$\begin{aligned} \Phi^{-1}(P(y \leq c_1)) &= \alpha_1 + \beta_1 X_1 + \cdots + \beta_p X_p, \\ \Phi^{-1}(P(y \leq c_2)) &= \alpha_2 + \beta_1 X_1 + \cdots + \beta_p X_p, \\ &\vdots \\ \Phi^{-1}(P(y \leq c_{k-1})) &= \alpha_{k-1} + \beta_1 X_1 + \cdots + \beta_p X_p, \end{aligned}$$

where  $P(y \leq c_j) = \pi_1 + \pi_2 + \dots + \pi_j$ .

The coefficients indicate the impact of a unit change in the predictor variable on the likelihood of a state. A positive coefficient,  $\beta_1$ , for example, indicates an increase in the underlying latent variable with an increase in the corresponding predictor variable,  $X_1$ . Hence, it causes a decrease in  $P(y \leq c_1)$  and an increase in  $P(y \leq c_k)$ .

After estimating the model coefficients using `mnrfit`, you can estimate the cumulative probabilities or the cumulative number in each category using `mnrval` with the

'type', 'cumulative' name-value pair option. `mnrval` accepts the coefficient estimates and the model statistics `mnrfit` returns, and estimates the categorical probabilities or the number in each category and their confidence intervals. You can specify which category or conditional probabilities or numbers to estimate by changing the value of the 'type' name-value pair argument.

## References

- [1] McCullagh, P., and J. A. Nelder. *Generalized Linear Models*. New York: Chapman & Hall, 1990.
- [2] Long, J. S. *Regression Models for Categorical and Limited Dependent Variables*. Sage Publications, 1997.
- [3] Dobson, A. J., and A. G. Barnett. *An Introduction to Generalized Linear Models*. Chapman and Hall/CRC. Taylor & Francis Group, 2008.

## See Also

`fitglm` | `glmfit` | `glmval` | `mnrfit` | `mnrval`

## More About

- “Multinomial Models for Nominal Responses” on page 12-2
- “Hierarchical Multinomial Models” on page 12-9



## Hierarchical Multinomial Models

The outcome of a response variable might sometimes be one of a restricted set of possible values. If there are only two possible outcomes, such as male and female for gender, these responses are called binary responses. If there are multiple outcomes, then they are called polytomous responses. These responses are usually qualitative rather than quantitative, such as preferred districts to live in a city, the severity level of a disease, the species for a certain flower type, and so on. Polytomous responses might also have categories which are not independent of each other. Instead the response happens in a sequential manner, or one category is nested in the previous one. These types of responses are called *hierarchical*, or *sequential*, or *nested multinomial responses*.

For example, if the response is the number of cigarettes a person smokes in a given day, the first level is whether the person is a smoker or not. Given that he or she is a smoker, the number of cigarettes he or she smokes can be from one to five or more than five a day. Given that it is more than 5, this person might be smoking from 6 to 10 or more than 10 cigarettes a day, and so on. The risk group at each level changes accordingly. At level one, the risk group is all of the individuals of interest (smoker or not), say  $m$ . If out of  $m$  individuals,  $y_1$  of them are not smokers, then at level two, the risk group is the number of all smoking individuals,  $m - y_1$ . If  $y_2$  of these  $m - y_1$  individuals smoke from one to five cigarettes a day, then at level three, the risk group is  $m - y_1 - y_2$ . So, at each level, the number of people in that category becomes a conditional binomial observation.

The hierarchical multinomial regression models are extensions of binary regression models based on conditional binary observations. The default is a model with different intercept and slopes (coefficients) among categories, in which case `mnrfit` fits a sequence of conditional binomial models. The `'interactions'`, `'on'` name-value pair specifies this in `mnrfit`. The default link function is logit and the `'link'`, `'logit'` name-value pair specifies this model in `mnrfit`.

Suppose the probability that an individual is in category  $j$  given that he or she is not in the previous categories is  $\pi_j$ , and the cumulative probability that a response belongs to a category  $j$  or a previous category is  $P(y \leq c_j)$ . Then the hierarchical model with a logit link function and different slopes assumption is

$$\begin{aligned} \ln\left(\frac{\pi_1}{1 - P(y \leq c_1)}\right) &= \ln\left(\frac{\pi_1}{1 - \pi_1}\right) = \alpha_1 + \beta_{11}X_1 + \beta_{12}X_2 + \cdots + \beta_{1p}X_p, \\ \ln\left(\frac{\pi_2}{1 - P(y \leq c_2)}\right) &= \ln\left(\frac{\pi_2}{1 - (\pi_1 + \pi_2)}\right) = \alpha_2 + \beta_{21}X_1 + \beta_{22}X_2 + \cdots + \beta_{2p}X_p, \\ &\vdots \\ \ln\left(\frac{\pi_{k-1}}{1 - P(y \leq c_{k-1})}\right) &= \ln\left(\frac{\pi_{k-1}}{1 - (\pi_1 + \cdots + \pi_{k-1})}\right) = \alpha_{k-1} + \beta_{(k-1)1}X_1 + \beta_{(k-1)2}X_2 + \cdots + \beta_{(k-1)p}X_p. \end{aligned}$$

For example, for a response variable with four sequential categories, there are  $4 - 1 = 3$  equations as follows:

$$\begin{aligned} \ln\left(\frac{\pi_1}{\pi_2 + \pi_3 + \pi_4}\right) &= \alpha_1 + \beta_{11}X_1 + \beta_{12}X_2 + \cdots + \beta_{1p}X_p, \\ \ln\left(\frac{\pi_2}{\pi_3 + \pi_4}\right) &= \alpha_2 + \beta_{21}X_1 + \beta_{22}X_2 + \cdots + \beta_{2p}X_p, \\ \ln\left(\frac{\pi_3}{\pi_4}\right) &= \alpha_3 + \beta_{31}X_1 + \beta_{32}X_2 + \cdots + \beta_{3p}X_p. \end{aligned}$$

The coefficients  $\beta_{ij}$  are interpreted within each level. For example, for the previous smoking example,  $\beta_{12}$  shows the impact of  $X_2$  on the log odds of a person being a smoker versus a nonsmoker, provided that everything else is held constant. Alternatively,  $\beta_{22}$  shows the impact of  $X_2$  on the log odds of a person smoking one to five cigarettes versus more than five cigarettes a day, given that he or she is a smoker, provided that everything else is held constant. Similarly,  $\beta_{23}$ , shows the effect of  $X_2$  on the log odds of a person smoking 6 to 10 cigarettes versus more than 10 cigarettes a day, given that he or she smokes more than 5 cigarettes a day, provided that everything else is held constant.

You can specify other link functions for hierarchical models. The 'link', 'probit' name-value pair argument uses the probit link function. With the separate slopes assumption, the model becomes

$$\begin{aligned}\Phi^{-1}(\pi_1) &= \alpha_1 + \beta_{11}X_1 + \cdots + \beta_{1p}X_p, \\ \Phi^{-1}(\pi_2) &= \alpha_2 + \beta_{21}X_1 + \cdots + \beta_{2p}X_p, \\ &\vdots \\ \Phi^{-1}(\pi_k) &= \alpha_k + \beta_{k1}X_1 + \cdots + \beta_{kp}X_p,\end{aligned}$$

where  $\pi_j$  is the conditional probability of being in category  $j$ , given that it is not in categories previous to category  $j$ . And  $\Phi^{-1}(\cdot)$  is the inverse of the standard normal cumulative distribution function.

After estimating the model coefficients using `mnrfit`, you can estimate the cumulative probabilities or the cumulative number in each category using `mnrval` with the `'type', 'conditional'` name-value pair argument. The function `mnrval` accepts the coefficient estimates and the model statistics `mnrfit` returns, and estimates the categorical probabilities or the number in each category and their confidence bounds. You can specify which category or cumulative probabilities or numbers to estimate by changing the value of the `'type'` name-value pair argument in `mnrval`.

## References

- [1] McCullagh, P., and J. A. Nelder. *Generalized Linear Models*. New York: Chapman & Hall, 1990.
- [2] Liao, T. F. *Interpreting Probability Models: Logit, Probit, and Other Generalized Linear Models* Series: Quantitative Applications in the Social Sciences. Sage Publications, 1994.

## See Also

`fitglm` | `glmfit` | `glmval` | `mnrfit` | `mnrval`

## More About

- “Multinomial Models for Nominal Responses” on page 12-2
- “Multinomial Models for Ordinal Responses” on page 12-5

## Generalized Linear Models

In this section...
“What Are Generalized Linear Models?” on page 12-12
“Prepare Data” on page 12-13
“Choose Generalized Linear Model and Link Function” on page 12-15
“Choose Fitting Method and Model” on page 12-18
“Fit Model to Data” on page 12-23
“Examine Quality and Adjust the Fitted Model” on page 12-24
“Predict or Simulate Responses to New Data” on page 12-35
“Share Fitted Models” on page 12-39

### What Are Generalized Linear Models?

Linear regression models describe a linear relationship between a response and one or more predictive terms. Many times, however, a nonlinear relationship exists. “Nonlinear Regression” on page 13-2 describes general nonlinear models. A special class of nonlinear models, called *generalized linear models*, uses linear methods.

Recall that linear models have these characteristics:

- At each set of values for the predictors, the response has a normal distribution with mean  $\mu$ .
- A coefficient vector  $b$  defines a linear combination  $Xb$  of the predictors  $X$ .
- The model is  $\mu = Xb$ .

In generalized linear models, these characteristics are generalized as follows:

- At each set of values for the predictors, the response has a distribution that can be normal on page B-134, binomial on page B-9, Poisson on page B-143, gamma on page B-47, or inverse Gaussian on page B-82, with parameters including a mean  $\mu$ .
- A coefficient vector  $b$  defines a linear combination  $Xb$  of the predictors  $X$ .
- A link function  $f$  defines the model as  $f(\mu) = Xb$ .

## Prepare Data

To begin fitting a regression, put your data into a form that fitting functions expect. All regression techniques begin with input data in an array  $X$  and response data in a separate vector  $y$ , or input data in a table or dataset array `tbl` and response data as a column in `tbl`. Each row of the input data represents one observation. Each column represents one predictor (variable).

For a table or dataset array `tbl`, indicate the response variable with the 'ResponseVar' name-value pair:

```
mdl = fitlm(tbl, 'ResponseVar', 'BloodPressure');
% or
mdl = fitglm(tbl, 'ResponseVar', 'BloodPressure');
```

The response variable is the last column by default.

You can use numeric categorical predictors. A categorical predictor is one that takes values from a fixed set of possibilities.

- For a numeric array  $X$ , indicate the categorical predictors using the 'Categorical' name-value pair. For example, to indicate that predictors 2 and 3 out of six are categorical:

```
mdl = fitlm(X, y, 'Categorical', [2,3]);
% or
mdl = fitglm(X, y, 'Categorical', [2,3]);
% or equivalently
mdl = fitlm(X, y, 'Categorical', logical([0 1 1 0 0 0]));
```

- For a table or dataset array `tbl`, fitting functions assume that these data types are categorical:
  - Logical
  - Categorical (nominal or ordinal)
  - Character array

If you want to indicate that a numeric predictor is categorical, use the 'Categorical' name-value pair.

Represent missing numeric data as `NaN`. To represent missing data for other data types, see “Missing Group Values” on page 2-61.

- For a 'binomial' model with data matrix  $X$ , the response  $y$  can be:
  - Binary column vector — Each entry represents success (1) or failure (0).
  - Two-column matrix of integers — The first column is the number of successes in each observation, the second column is the number of trials in that observation.
- For a 'binomial' model with table or dataset `tbl`:
  - Use the `ResponseVar` name-value pair to specify the column of `tbl` that gives the number of successes in each observation.
  - Use the `BinomialSize` name-value pair to specify the column of `tbl` that gives the number of trials in each observation.

### Dataset Array for Input and Response Data

For example, to create a dataset array from an Excel spreadsheet:

```
ds = dataset('XLSFile', 'hospital.xls', ...  
            'ReadObsNames', true);
```

To create a dataset array from workspace variables:

```
load carsmall  
ds = dataset(MPG, Weight);  
ds.Year = ordinal(Model_Year);
```

### Table for Input and Response Data

To create a table from workspace variables:

```
load carsmall  
tbl = table(MPG, Weight);  
tbl.Year = ordinal(Model_Year);
```

### Numeric Matrix for Input Data, Numeric Vector for Response

For example, to create numeric arrays from workspace variables:

```
load carsmall  
X = [Weight Horsepower Cylinders Model_Year];  
y = MPG;
```

To create numeric arrays from an Excel spreadsheet:

```
[X, Xnames] = xlsread('hospital.xls');
y = X(:,4); % response y is systolic pressure
X(:,4) = []; % remove y from the X matrix
```

Notice that the nonnumeric entries, such as `sex`, do not appear in `X`.

## Choose Generalized Linear Model and Link Function

Often, your data suggests the distribution type of the generalized linear model.

Response Data Type	Suggested Model Distribution Type
Any real number	'normal'
Any positive number	'gamma' or 'inverse gaussian'
Any nonnegative integer	'poisson'
Integer from 0 to $n$ , where $n$ is a fixed positive value	'binomial'

Set the model distribution type with the `Distribution` name-value pair. After selecting your model type, choose a link function to map between the mean  $\mu$  and the linear predictor  $Xb$ .

Value	Description
'compploglog'	$\log(-\log((1-\mu))) = Xb$
'identity', default for the distribution 'normal'	$\mu = Xb$
'log', default for the distribution 'poisson'	$\log(\mu) = Xb$
'logit', default for the distribution 'binomial'	$\log(\mu/(1-\mu)) = Xb$
'loglog'	$\log(-\log(\mu)) = Xb$
'probit'	$\Phi^{-1}(\mu) = Xb$ , where $\Phi$ is the normal (Gaussian) CDF function
'reciprocal', default for the distribution 'gamma'	$\mu^{-1} = Xb$

Value	Description
$p$ (a number), default for the distribution 'inverse gaussian' (with $p = -2$ )	$\mu^p = Xb$
A cell array of the form {FL FD FI}, containing three function handles, created using @, that define the link (FL), the derivative of the link (FD), and the inverse link (FI), or equivalently, a structure of function handles with field Link containing FL, field Derivative containing FD, and field Inverse containing FI	User-specified link function (see “Custom Link Function” on page 12-16)

The nondefault link functions are mainly useful for binomial models. These nondefault link functions are 'comploglog', 'loglog', and 'probit'.

### Custom Link Function

The link function defines the relationship  $f(\mu) = Xb$  between the mean response  $\mu$  and the linear combination  $Xb = X^*b$  of the predictors. You can choose one of the built-in link functions or define your own by specifying the link function FL, its derivative FD, and its inverse FI:

- The link function FL calculates  $f(\mu)$ .
- The derivative of the link function FD calculates  $df(\mu)/d\mu$ .
- The inverse function FI calculates  $g(Xb) = \mu$ .

You can specify a custom link function in either of two equivalent ways. Each way contains function handles that accept a single array of values representing  $\mu$  or  $Xb$ , and returns an array the same size. The function handles are either in a cell array or a structure:

- Cell array of the form {FL FD FI}, containing three function handles, created using @, that define the link (FL), the derivative of the link (FD), and the inverse link (FI).
- Structure  $s$  with three fields, each containing a function handle created using @:
  - $s.Link$  — Link function



- `s.Derivative` — Derivative of the link function
- `s.Inverse` — Inverse of the link function

For example, to fit a model using the 'probit' link function:

```
x = [2100 2300 2500 2700 2900 ...
     3100 3300 3500 3700 3900 4100 4300]';
n = [48 42 31 34 31 21 23 23 21 16 17 21]';
y = [1 2 0 3 8 8 14 17 19 15 17 21]';
g = fitglm(x, [y n], ...
           'linear', 'distr', 'binomial', 'link', 'probit')
```

g =

Generalized Linear regression model:

```
probit(y) ~ 1 + x1
Distribution = Binomial
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	-7.3628	0.66815	-11.02	3.0701e-28
x1	0.0023039	0.00021352	10.79	3.8274e-27

12 observations, 10 error degrees of freedom

Dispersion: 1

Chi^2-statistic vs. constant model: 241, p-value = 2.25e-54

You can perform the same fit using a custom link function that performs identically to the 'probit' link function:

```
s = {@norminv,@(x)1./normpdf(norminv(x)),@normcdf};
g = fitglm(x, [y n], ...
           'linear', 'distr', 'binomial', 'link', s)
```

g =

Generalized Linear regression model:

```
link(y) ~ 1 + x1
Distribution = Binomial
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	-7.3628	0.66815	-11.02	3.0701e-28
x1	0.0023039	0.00021352	10.79	3.8274e-27

```
12 observations, 10 error degrees of freedom
Dispersion: 1
Chi^2-statistic vs. constant model: 241, p-value = 2.25e-54
```

The two models are the same.

Equivalently, you can write `s` as a structure instead of a cell array of function handles:

```
s.Link = @norminv;
s.Derivative = @(x) 1./normpdf(norminv(x));
s.Inverse = @normcdf;
g = fitglm(x, [y n], ...
    'linear', 'distr', 'binomial', 'link', s)
```

```
g =
```

```
Generalized Linear regression model:
link(y) ~ 1 + x1
Distribution = Binomial
```

```
Estimated Coefficients:
```

	Estimate	SE	tStat	pValue
(Intercept)	-7.3628	0.66815	-11.02	3.0701e-28
x1	0.0023039	0.00021352	10.79	3.8274e-27

```
12 observations, 10 error degrees of freedom
Dispersion: 1
Chi^2-statistic vs. constant model: 241, p-value = 2.25e-54
```

## Choose Fitting Method and Model

There are two ways to create a fitted model.

- Use `fitglm` when you have a good idea of your generalized linear model, or when you want to adjust your model later to include or exclude certain terms.
- Use `stepwiseglm` when you want to fit your model using stepwise regression. `stepwiseglm` starts from one model, such as a constant, and adds or subtracts terms one at a time, choosing an optimal term each time in a greedy fashion, until it cannot improve further. Use stepwise fitting to find a good model, one that has only relevant terms.

The result depends on the starting model. Usually, starting with a constant model leads to a small model. Starting with more terms can lead to a more complex model, but one that has lower mean squared error.

In either case, provide a model to the fitting function (which is the starting model for `stepwiseglm`).

Specify a model using one of these methods.

- “Brief Model Name” on page 12-19
- “Terms Matrix” on page 12-19
- “Formula” on page 12-22

### Brief Model Name

Name	Model Type
'constant'	Model contains only a constant (intercept) term.
'linear'	Model contains an intercept and linear terms for each predictor.
'interactions'	Model contains an intercept, linear terms, and all products of pairs of distinct predictors (no squared terms).
'purequadratic'	Model contains an intercept, linear terms, and squared terms.
'quadratic'	Model contains an intercept, linear terms, interactions, and squared terms.
'polyijk'	Model is a polynomial with all terms up to degree $i$ in the first predictor, degree $j$ in the second predictor, etc. Use numerals 0 through 9. For example, 'poly2111' has a constant plus all linear and product terms, and also contains terms with predictor 1 squared.

### Terms Matrix

A terms matrix is a  $t$ -by- $(p + 1)$  matrix specifying terms in a model, where  $t$  is the number of terms,  $p$  is the number of predictor variables, and plus one is for the response variable.

The value of  $T(i, j)$  is the exponent of variable  $j$  in term  $i$ . Suppose there are three predictor variables A, B, and C:

```
[0 0 0 0] % Constant term or intercept
[0 1 0 0] % B; equivalently, A^0 * B^1 * C^0
[1 0 1 0] % A*C
[2 0 0 0] % A^2
[0 1 2 0] % B*(C^2)
```

The 0 at the end of each term represents the response variable. In general,

- If you have the variables in a table or dataset array, then 0 must represent the response variable depending on the position of the response variable. The following example illustrates this.

Load the sample data and define the dataset array.

```
load hospital
dsa = dataset(hospital.Sex,hospital.BloodPressure(:,1),hospital.Age,...
hospital.Smoker, 'VarNames', {'Sex', 'BloodPressure', 'Age', 'Smoker'});
```

Represent the linear model 'BloodPressure ~ 1 + Sex + Age + Smoker' in a terms matrix. The response variable is in the second column of the dataset array, so there must be a column of 0s for the response variable in the second column of the terms matrix.

```
T = [0 0 0 0;1 0 0 0;0 0 1 0;0 0 0 1]
```

```
T =
```

```
    0    0    0    0
    1    0    0    0
    0    0    1    0
    0    0    0    1
```

Redefine the dataset array.

```
dsa = dataset(hospital.BloodPressure(:,1),hospital.Sex,hospital.Age,...
hospital.Smoker, 'VarNames', {'BloodPressure', 'Sex', 'Age', 'Smoker'});
```

Now, the response variable is the first term in the dataset array. Specify the same linear model, 'BloodPressure ~ 1 + Sex + Age + Smoker', using a terms matrix.

```
T = [0 0 0 0;0 1 0 0;0 0 1 0;0 0 0 1]
```

```
T =
```

```

0      0      0      0
0      1      0      0
0      0      1      0
0      0      0      1

```

- If you have the predictor and response variables in a matrix and column vector, then you must include 0 for the response variable at the end of each term. The following example illustrates this.

Load the sample data and define the matrix of predictors.

```

load carsmall
X = [Acceleration,Weight];

```

Specify the model 'MPG ~ Acceleration + Weight + Acceleration:Weight + Weight^2' using a term matrix and fit the model to the data. This model includes the main effect and two-way interaction terms for the variables, Acceleration and Weight, and a second-order term for the variable, Weight.

```
T = [0 0 0;1 0 0;0 1 0;1 1 0;0 2 0]
```

```
T =
```

```

0      0      0
1      0      0
0      1      0
1      1      0
0      2      0

```

Fit a linear model.

```
mdl = fitlm(X,MPG,T)
```

```
mdl =
```

```
Linear regression model:
```

```
y ~ 1 + x1*x2 + x2^2
```

```
Estimated Coefficients:
```

	Estimate	SE	tStat	pValue
(Intercept)	48.906	12.589	3.8847	0.00019665
x1	0.54418	0.57125	0.95261	0.34337
x2	-0.012781	0.0060312	-2.1192	0.036857
x1:x2	-0.00010892	0.00017925	-0.6076	0.545
x2^2	9.7518e-07	7.5389e-07	1.2935	0.19917

```
Number of observations: 94, Error degrees of freedom: 89
Root Mean Squared Error: 4.1
R-squared: 0.751, Adjusted R-Squared 0.739
F-statistic vs. constant model: 67, p-value = 4.99e-26
```

Only the intercept and `x2` term, which correspond to the `Weight` variable, are significant at the 5% significance level.

Now, perform a stepwise regression with a constant model as the starting model and a linear model with interactions as the upper model.

```
T = [0 0 0;1 0 0;0 1 0;1 1 0];
mdl = stepwiselm(X,MPG,[0 0 0], 'upper', T)
```

```
1. Adding x2, FStat = 259.3087, pValue = 1.643351e-28
```

```
mdl =
```

```
Linear regression model:
```

```
y ~ 1 + x2
```

```
Estimated Coefficients:
```

	Estimate	SE	tStat	pValue
(Intercept)	49.238	1.6411	30.002	2.7015e-49
x2	-0.0086119	0.0005348	-16.103	1.6434e-28

```
Number of observations: 94, Error degrees of freedom: 92
Root Mean Squared Error: 4.13
R-squared: 0.738, Adjusted R-Squared 0.735
F-statistic vs. constant model: 259, p-value = 1.64e-28
```

The results of the stepwise regression are consistent with the results of `fitlm` in the previous step.

## Formula

A formula for a model specification is a character vector of the form

```
'Y ~ terms',
```

- `Y` is the response name.
- `terms` contains

- Variable names
- + to include the next variable
- - to exclude the next variable
- : to define an interaction, a product of terms
- \* to define an interaction and all lower-order terms
- ^ to raise the predictor to a power, exactly as in \* repeated, so ^ includes lower order terms as well
- () to group terms

---

**Tip** Formulas include a constant (intercept) term by default. To exclude a constant term from the model, include -1 in the formula.

---

### Examples:

'Y ~ A + B + C' is a three-variable linear model with intercept.

'Y ~ A + B + C - 1' is a three-variable linear model without intercept.

'Y ~ A + B + C + B^2' is a three-variable model with intercept and a B^2 term.

'Y ~ A + B^2 + C' is the same as the previous example, since B^2 includes a B term.

'Y ~ A + B + C + A:B' includes an A\*B term.

'Y ~ A\*B + C' is the same as the previous example, since  $A*B = A + B + A:B$ .

'Y ~ A\*B\*C - A:B:C' has all interactions among A, B, and C, except the three-way interaction.

'Y ~ A\*(B + C + D)' has all linear terms, plus products of A with each of the other variables.

## Fit Model to Data

Create a fitted model using `fitglm` or `stepwiseglm`. Choose between them as in “Choose Fitting Method and Model” on page 12-18. For generalized linear models other than those with a normal distribution, give a Distribution name-value pair as in “Choose Generalized Linear Model and Link Function” on page 12-15. For example,

```
mdl = fitglm(X,y,'linear','Distribution','poisson')
% or
mdl = fitglm(X,y,'quadratic',...
            'Distribution','binomial')
```

## Examine Quality and Adjust the Fitted Model

After fitting a model, examine the result.

- “Model Display” on page 12-24
- “Diagnostic Plots” on page 12-25
- “Residuals — Model Quality for Training Data” on page 12-28
- “Plots to Understand Predictor Effects and How to Modify a Model” on page 12-31

### Model Display

A linear regression model shows several diagnostics when you enter its name or enter `disp mdl`. This display gives some of the basic information to check whether the fitted model represents the data adequately.

For example, fit a Poisson model to data constructed with two out of five predictors not affecting the response, and with no intercept term:

```
rng('default') % for reproducibility
X = randn(100,5);
mu = exp(X(:, [1 4 5]) * [.4; .2; .3]);
y = poissrnd(mu);
mdl = fitglm(X, y, ...
    'linear', 'Distribution', 'poisson')
```

```
mdl =
```

```
Generalized Linear regression model:
log(y) ~ 1 + x1 + x2 + x3 + x4 + x5
Distribution = Poisson
```

```
Estimated Coefficients:
```

	Estimate	SE	tStat	pValue
(Intercept)	0.039829	0.10793	0.36901	0.71212
x1	0.38551	0.076116	5.0647	4.0895e-07
x2	-0.034905	0.086685	-0.40266	0.6872
x3	-0.17826	0.093552	-1.9054	0.056722
x4	0.21929	0.09357	2.3436	0.019097
x5	0.28918	0.1094	2.6432	0.0082126

```
100 observations, 94 error degrees of freedom
```



```
Dispersion: 1
Chi^2-statistic vs. constant model: 44.9, p-value = 1.55e-08
```

Notice that:

- The display contains the estimated values of each coefficient in the `Estimate` column. These values are reasonably near the true values `[0; .4; 0; 0; .2; .3]`, except possibly the coefficient of `x3` is not terribly near 0.
- There is a standard error column for the coefficient estimates.
- The reported `pValue` (which are derived from the  $t$  statistics under the assumption of normal errors) for predictors 1, 4, and 5 are small. These are the three predictors that were used to create the response data `y`.
- The `pValue` for `(Intercept)`, `x2` and `x3` are larger than 0.01. These three predictors were not used to create the response data `y`. The `pValue` for `x3` is just over .05, so might be regarded as possibly significant.
- The display contains the Chi-square statistic.

### Diagnostic Plots

Diagnostic plots help you identify outliers, and see other problems in your model or fit. To illustrate these plots, consider binomial regression with a logistic link function.

The logistic model is useful for proportion data. It defines the relationship between the proportion  $p$  and the weight  $w$  by:

$$\log[p/(1-p)] = b_1 + b_2w$$

This example fits a binomial model to data. The data are derived from `carbig.mat`, which contains measurements of large cars of various weights. Each weight in `w` has a corresponding number of cars in `total` and a corresponding number of poor-mileage cars in `poor`.

It is reasonable to assume that the values of `poor` follow binomial on page B-9 distributions, with the number of trials given by `total` and the percentage of successes depending on `w`. This distribution can be accounted for in the context of a logistic model by using a generalized linear model with link function  $\log(\mu/(1-\mu)) = Xb$ . This link function is called `'logit'`.

```
w = [2100 2300 2500 2700 2900 3100 ...
     3300 3500 3700 3900 4100 4300]';
```

```
total = [48 42 31 34 31 21 23 23 21 16 17 21]';  
poor = [1 2 0 3 8 8 14 17 19 15 17 21]';  
mdl = fitglm(w, [poor total], ...  
            'linear', 'Distribution', 'binomial', 'link', 'logit')
```

```
mdl =
```

```
Generalized Linear regression model:
```

```
logit(y) ~ 1 + x1  
Distribution = Binomial
```

```
Estimated Coefficients:
```

	Estimate	SE	tStat	pValue
(Intercept)	-13.38	1.394	-9.5986	8.1019e-22
x1	0.0041812	0.00044258	9.4474	3.4739e-21

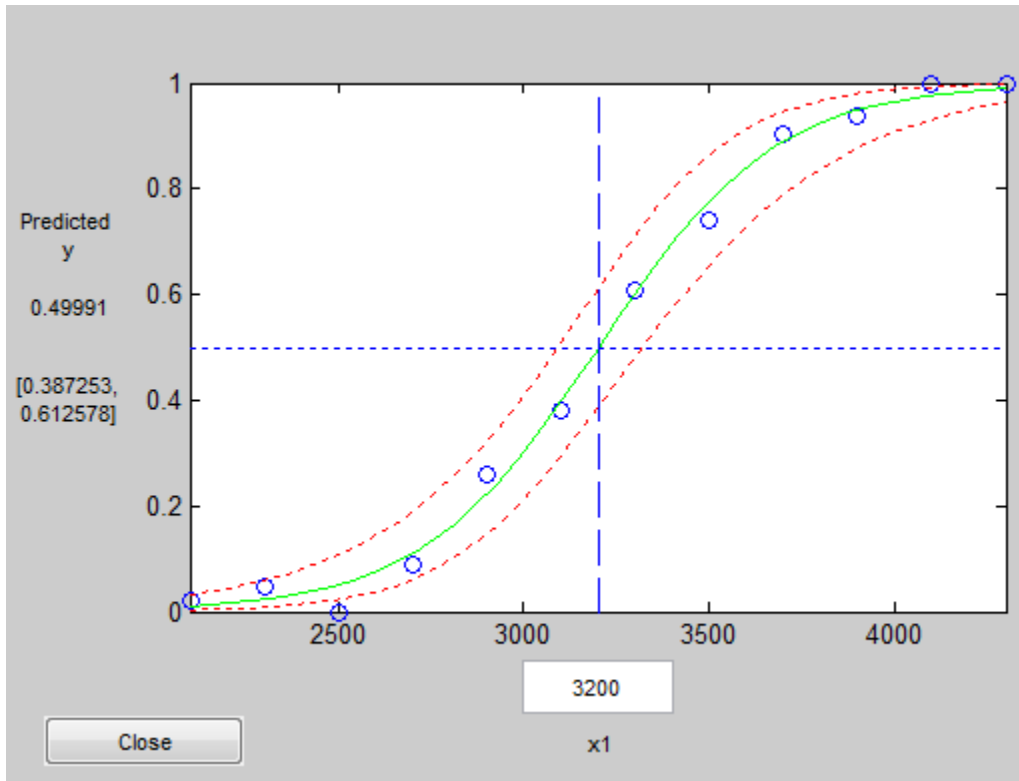
```
12 observations, 10 error degrees of freedom
```

```
Dispersion: 1
```

```
Chi^2-statistic vs. constant model: 242, p-value = 1.3e-54
```

**See how well the model fits the data.**

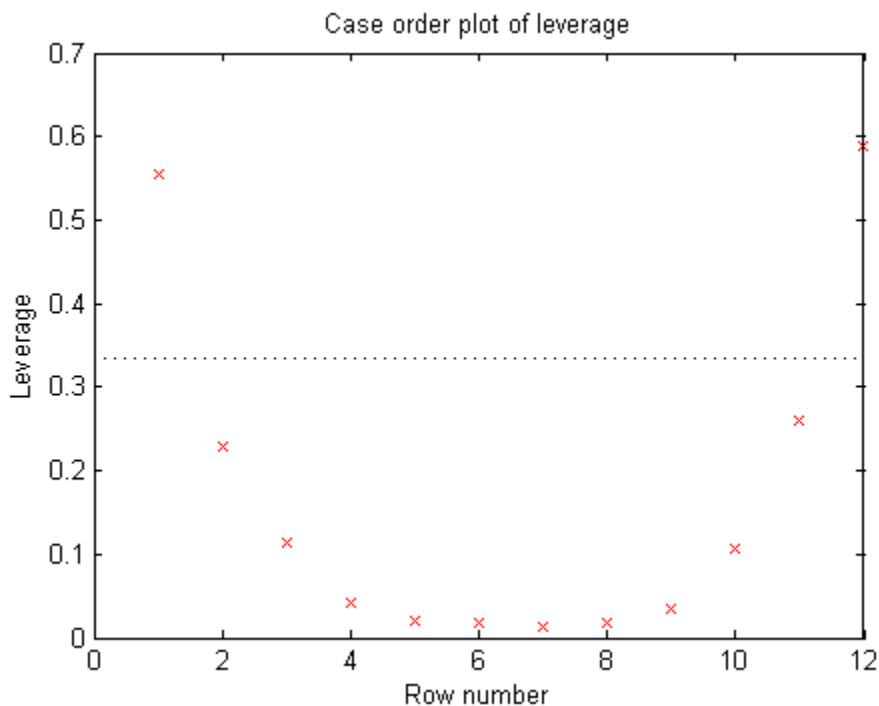
```
plotSlice(mdl)
```



The fit looks reasonably good, with fairly wide confidence bounds.

To examine further details, create a leverage plot.

```
plotDiagnostics (mdl)
```



This is typical of a regression with points ordered by the predictor variable. The leverage of each point on the fit is higher for points with relatively extreme predictor values (in either direction) and low for points with average predictor values. In examples with multiple predictors and with points not ordered by predictor value, this plot can help you identify which observations have high leverage because they are outliers as measured by their predictor values.

### Residuals — Model Quality for Training Data

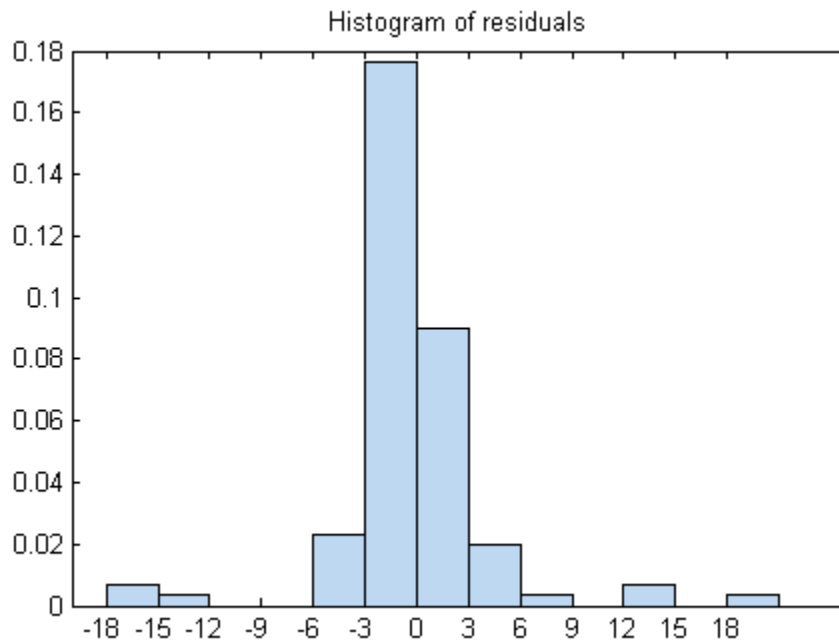
There are several residual plots to help you discover errors, outliers, or correlations in the model or data. The simplest residual plots are the default histogram plot, which shows the range of the residuals and their frequencies, and the probability plot, which shows how the distribution of the residuals compares to a normal distribution with matched variance.

This example shows residual plots for a fitted Poisson model. The data construction has two out of five predictors not affecting the response, and no intercept term:

```
rng('default') % for reproducibility
X = randn(100,5);
mu = exp(X(:, [1 4 5]) * [2; 1; .5]);
y = poissrnd(mu);
mdl = fitglm(X,y,...
    'linear', 'Distribution', 'poisson');
```

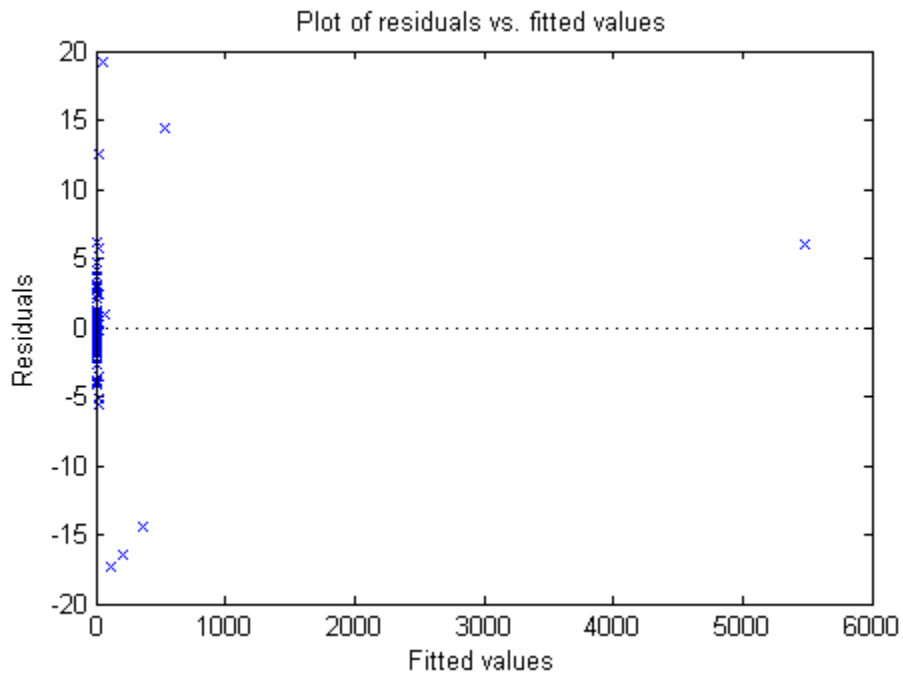
Examine the residuals:

```
plotResiduals(mdl)
```



While most residuals cluster near 0, there are several near  $\pm 18$ . So examine a different residuals plot.

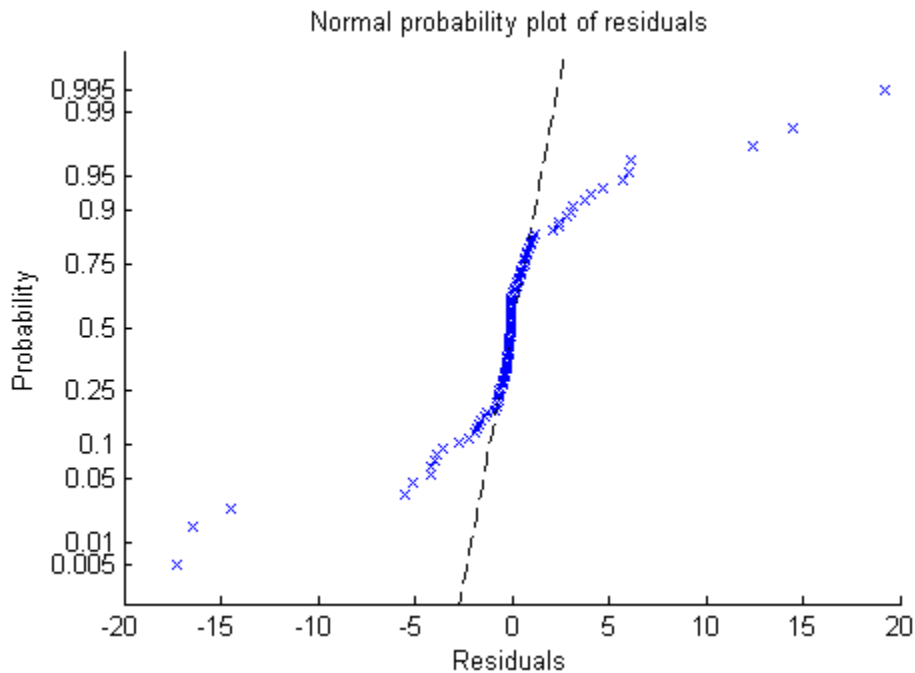
```
plotResiduals(mdl, 'fitted')
```



The large residuals don't seem to have much to do with the sizes of the fitted values.

Perhaps a probability plot is more informative.

```
plotResiduals(mdl, 'probability')
```



Now it is clear. The residuals do not follow a normal distribution. Instead, they have fatter tails, much as an underlying Poisson distribution.

### Plots to Understand Predictor Effects and How to Modify a Model

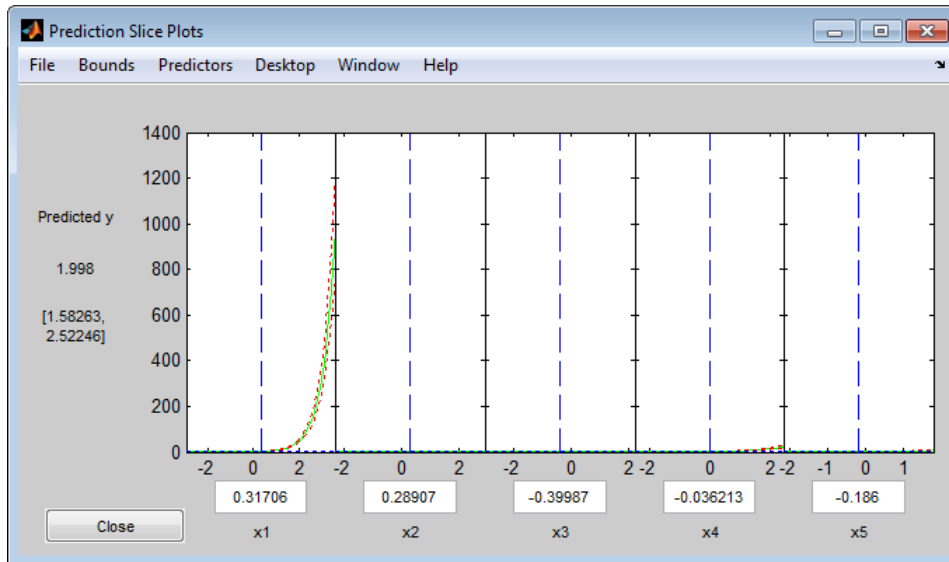
This example shows how to understand the effect each predictor has on a regression model, and how to modify the model to remove unnecessary terms.

- 1 Create a model from some predictors in artificial data. The data do not use the second and third columns in  $X$ . So you expect the model not to show much dependence on those predictors.

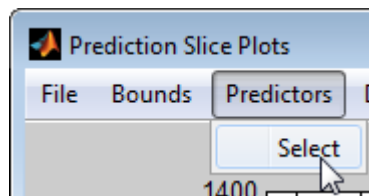
```
rng('default') % for reproducibility
X = randn(100,5);
mu = exp(X(:, [1 4 5]) * [2; 1; .5]);
y = poissrnd(mu);
mdl = fitglm(X,y,...
    'linear', 'Distribution', 'poisson');
```

- 2 Examine a slice plot of the responses. This displays the effect of each predictor separately.

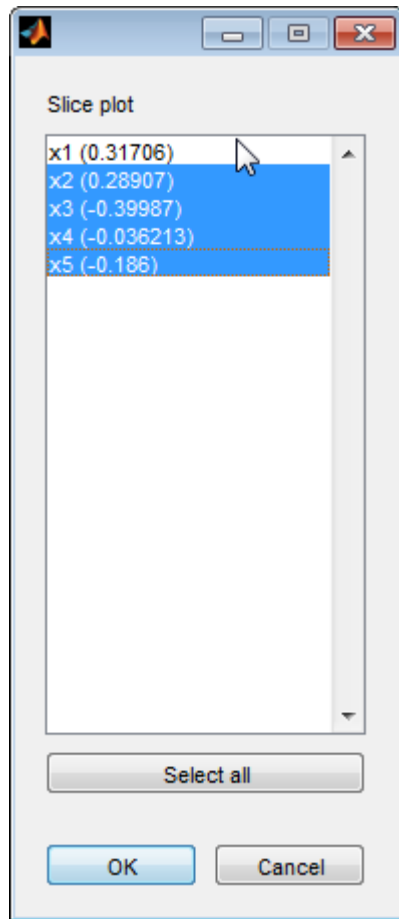
```
plotSlice(mdl)
```

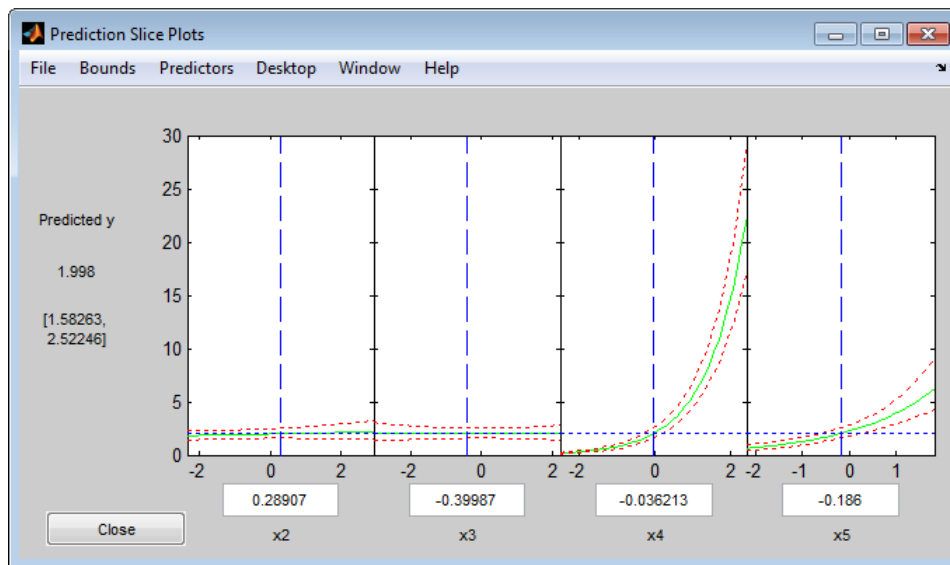


The scale of the first predictor is overwhelming the plot. Disable it using the **Predictors** menu.









Now it is clear that predictors 2 and 3 have little to no effect.

You can drag the individual predictor values, which are represented by dashed blue vertical lines. You can also choose between simultaneous and non-simultaneous confidence bounds, which are represented by dashed red curves. Dragging the predictor lines confirms that predictors 2 and 3 have little to no effect.

- 3 Remove the unnecessary predictors using either `removeTerms` or `step`. Using `step` can be safer, in case there is an unexpected importance to a term that becomes apparent after removing another term. However, sometimes `removeTerms` can be effective when `step` does not proceed. In this case, the two give identical results.

```
mdl1 = removeTerms(mdl, 'x2 + x3')
```

```
mdl1 =
```

```
Generalized Linear regression model:
```

```
log(y) ~ 1 + x1 + x4 + x5
Distribution = Poisson
```

```
Estimated Coefficients:
```

	Estimate	SE	tStat	pValue
(Intercept)	0.17604	0.062215	2.8295	0.004662

x1	1.9122	0.024638	77.614	0
x4	0.98521	0.026393	37.328	5.6696e-305
x5	0.61321	0.038435	15.955	2.6473e-57

100 observations, 96 error degrees of freedom

Dispersion: 1

Chi<sup>2</sup>-statistic vs. constant model: 4.97e+04, p-value = 0

```
mdl1 = step(mdl, 'NSteps', 5, 'Upper', 'linear')
```

1. Removing x3, Deviance = 93.856, Chi2Stat = 0.00075551, PValue = 0.97807

2. Removing x2, Deviance = 96.333, Chi2Stat = 2.4769, PValue = 0.11553

mdl1 =

Generalized Linear regression model:

log(y) ~ 1 + x1 + x4 + x5

Distribution = Poisson

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	0.17604	0.062215	2.8295	0.004662
x1	1.9122	0.024638	77.614	0
x4	0.98521	0.026393	37.328	5.6696e-305
x5	0.61321	0.038435	15.955	2.6473e-57

100 observations, 96 error degrees of freedom

Dispersion: 1

Chi<sup>2</sup>-statistic vs. constant model: 4.97e+04, p-value = 0

## Predict or Simulate Responses to New Data

There are three ways to use a linear model to predict the response to new data:

- “predict” on page 12-35
- “feval” on page 12-36
- “random” on page 12-38

### predict

The `predict` method gives a prediction of the mean responses and, if requested, confidence bounds.

This example shows how to predict and obtain confidence intervals on the predictions using the `predict` method.

- 1 Create a model from some predictors in artificial data. The data do not use the second and third columns in  $X$ . So you expect the model not to show much dependence on these predictors. Construct the model stepwise to include the relevant predictors automatically.

```
rng('default') % for reproducibility
X = randn(100,5);
mu = exp(X(:, [1 4 5])*[2;1;.5]);
y = poissrnd(mu);
mdl = stepwiseglm(X,y,...
    'constant','upper','linear','Distribution','poisson');
```

```
1. Adding x1, Deviance = 2515.02869, Chi2Stat = 47242.9622, PValue = 0
2. Adding x4, Deviance = 328.39679, Chi2Stat = 2186.6319, PValue = 0
3. Adding x5, Deviance = 96.3326, Chi2Stat = 232.0642, PValue = 2.114384e-52
```

- 2 Generate some new data, and evaluate the predictions from the data.

```
Xnew = randn(3,5) + repmat([1 2 3 4 5],[3,1]); % new data
[ynew,ynewci] = predict(mdl,Xnew)
```

```
ynew =
```

```
1.0e+04 *
    0.1130
    1.7375
    3.7471
```

```
ynewci =
```

```
1.0e+04 *
    0.0821    0.1555
    1.2167    2.4811
    2.8419    4.9407
```

### feval

When you construct a model from a table or dataset array, `feval` is often more convenient for predicting mean responses than `predict`. However, `feval` does not provide confidence bounds.

This example shows how to predict mean responses using the `feval` method.

- 1 Create a model from some predictors in artificial data. The data do not use the second and third columns in X. So you expect the model not to show much dependence on these predictors. Construct the model stepwise to include the relevant predictors automatically.

```
rng('default') % for reproducibility
X = randn(100,5);
mu = exp(X(:, [1 4 5])*[2;1;.5]);
y = poissrnd(mu);

X = array2table(X); % create data table
y = array2table(y);
tbl = [X y];

mdl = stepwiseglm(tbl,...
    'constant','upper','linear','Distribution','poisson');

1. Adding x1, Deviance = 2515.02869, Chi2Stat = 47242.9622, PValue = 0
2. Adding x4, Deviance = 328.39679, Chi2Stat = 2186.6319, PValue = 0
3. Adding x5, Deviance = 96.3326, Chi2Stat = 232.0642, PValue = 2.114384e-52
```

- 2 Generate some new data, and evaluate the predictions from the data.

```
Xnew = randn(3,5) + repmat([1 2 3 4 5],[3,1]); % new data
ynew = feval(mdl,Xnew(:,1),Xnew(:,4),Xnew(:,5)) % only need predictors 1,4,5

ynew =

    1.0e+04 *

    0.1130
    1.7375
    3.7471
```

Equivalently,

```
ynew = feval(mdl,Xnew(:, [1 4 5])) % only need predictors 1,4,5

ynew =

    1.0e+04 *

    0.1130
    1.7375
    3.7471
```

## random

The `random` method generates new random response values for specified predictor values. The distribution of the response values is the distribution used in the model. `random` calculates the mean of the distribution from the predictors, estimated coefficients, and link function. For distributions such as normal, the model also provides an estimate of the variance of the response. For the binomial and Poisson distributions, the variance of the response is determined by the mean; `random` does not use a separate “dispersion” estimate.

This example shows how to simulate responses using the `random` method.

- 1 Create a model from some predictors in artificial data. The data do not use the second and third columns in `X`. So you expect the model not to show much dependence on these predictors. Construct the model stepwise to include the relevant predictors automatically.

```
rng('default') % for reproducibility
X = randn(100,5);
mu = exp(X(:, [1 4 5])*[2;1;.5]);
y = poissrnd(mu);
mdl = stepwiseglm(X,y,...
    'constant','upper','linear','Distribution','poisson');
```

```
1. Adding x1, Deviance = 2515.02869, Chi2Stat = 47242.9622, PValue = 0
2. Adding x4, Deviance = 328.39679, Chi2Stat = 2186.6319, PValue = 0
3. Adding x5, Deviance = 96.3326, Chi2Stat = 232.0642, PValue = 2.114384e-52
```

- 2 Generate some new data, and evaluate the predictions from the data.

```
Xnew = randn(3,5) + repmat([1 2 3 4 5],[3,1]); % new data
ysim = random(mdl,Xnew)
```

```
ysim =
```

```
    1111
    17121
    37457
```

The predictions from `random` are Poisson samples, so are integers.

- 3 Evaluate the `random` method again, the result changes.

```
ysim = random(mdl,Xnew)
```

```
ysim =
```

```

1175
17320
37126

```

## Share Fitted Models

The model display contains enough information to enable someone else to recreate the model in a theoretical sense. For example,

```

rng('default') % for reproducibility
X = randn(100,5);
mu = exp(X(:, [1 4 5]) * [2;1;.5]);
y = poissrnd(mu);
mdl = stepwiseglm(X,y,...
    'constant','upper','linear','Distribution','poisson')

```

1. Adding x1, Deviance = 2515.02869, Chi2Stat = 47242.9622, PValue = 0
2. Adding x4, Deviance = 328.39679, Chi2Stat = 2186.6319, PValue = 0
3. Adding x5, Deviance = 96.3326, Chi2Stat = 232.0642, PValue = 2.114384e-52

```
mdl =
```

```

Generalized Linear regression model:
log(y) ~ 1 + x1 + x4 + x5
Distribution = Poisson

```

```
Estimated Coefficients:
```

	Estimate	SE	tStat	pValue
(Intercept)	0.17604	0.062215	2.8295	0.004662
x1	1.9122	0.024638	77.614	0
x4	0.98521	0.026393	37.328	5.6696e-305
x5	0.61321	0.038435	15.955	2.6473e-57

```

100 observations, 96 error degrees of freedom
Dispersion: 1
Chi^2-statistic vs. constant model: 4.97e+04, p-value = 0

```

You can access the model description programmatically, too. For example,

```
mdl.Coefficients.Estimate
```

```
ans =
```

```

0.1760
1.9122
0.9852
0.6132

```

```
mdl.Formula
```

ans =

$\log(\bar{y}) \sim 1 + x_1 + x_4 + x_5$

### References

- [1] Collett, D. *Modeling Binary Data*. New York: Chapman & Hall, 2002.
- [2] Dobson, A. J. *An Introduction to Generalized Linear Models*. New York: Chapman & Hall, 1990.
- [3] McCullagh, P., and J. A. Nelder. *Generalized Linear Models*. New York: Chapman & Hall, 1990.
- [4] Neter, J., M. H. Kutner, C. J. Nachtsheim, and W. Wasserman. *Applied Linear Statistical Models*, Fourth Edition. Irwin, Chicago, 1996.



## Generalized Linear Model Workflow

This example shows how to fit a generalized linear model and analyze the results. A typical workflow involves the following: import data, fit a generalized linear model, test its quality, modify it to improve the quality, and make predictions based on the model. It computes the probability that a flower is in one of two classes, based on the Fisher iris data.

### Step 1. Load the data.

Load the Fisher iris data. Extract the rows that have classification `versicolor` or `virginica`. These are rows 51 to 150. Create logical response variables that are `true` for `versicolor` flowers.

```
load fisheriris
X = meas(51:end,:); % versicolor and virginica
y = strcmp('versicolor',species(51:end));
```

### Step 2. Fit a generalized linear model.

Fit a binomial generalized linear model to the data.

```
mdl = fitglm(X,y,'linear',...
            'distr','binomial')
```

```
mdl =
```

```
Generalized linear regression model:
logit(y) ~ 1 + x1 + x2 + x3 + x4
Distribution = Binomial
```

```
Estimated Coefficients:
```

	Estimate	SE	tStat	pValue
(Intercept)	42.638	25.708	1.6586	0.097204
x1	2.4652	2.3943	1.0296	0.30319
x2	6.6809	4.4796	1.4914	0.13585
x3	-9.4294	4.7372	-1.9905	0.046537
x4	-18.286	9.7426	-1.8769	0.060529

```
100 observations, 95 error degrees of freedom
Dispersion: 1
Chi^2-statistic vs. constant model: 127, p-value = 1.95e-26
```

### Step 3. Examine the result, consider alternative models.

Some  $P$ -values in the `pValue` column are not very small. Perhaps the model can be simplified.

See if some 95% confidence intervals for the coefficients include 0. If so, perhaps these model terms could be removed.

```
confint = coefCI mdl)
```

```
confint =
    -8.3984    93.6740
    -2.2881     7.2185
    -2.2122    15.5739
   -18.8339    -0.0248
   -37.6277     1.0554
```

Only two of the predictors have coefficients whose confidence intervals do not include 0.

The coefficients of 'x1' and 'x2' have the largest  $P$ -values. Test whether both coefficients could be zero.

```
M = [0 1 0 0 0    % picks out coefficient for column 1
      0 0 1 0 0]; % picks out coefficient for column 2
p = coefTest(mdl,M)
```

```
p =
    0.1442
```

The  $P$ -value of about 0.14 is not very small. Drop those terms from the model.

```
mdl1 = removeTerms(mdl, 'x1 + x2')
```

```
mdl1 =
```

```
Generalized linear regression model:
  logit(y) ~ 1 + x3 + x4
  Distribution = Binomial
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	45.272	13.612	3.326	0.00088103
x3	-5.7545	2.3059	-2.4956	0.012576
x4	-10.447	3.7557	-2.7816	0.0054092

```
100 observations, 97 error degrees of freedom
Dispersion: 1
Chi^2-statistic vs. constant model: 118, p-value = 2.3e-26
```

Perhaps it would have been better to have `stepwiseglm` identify the model initially.

```
mdl2 = stepwiseglm(X, y, ...
  'constant', 'Distribution', 'binomial', 'upper', 'linear')

1. Adding x4, Deviance = 33.4208, Chi2Stat = 105.2086, PValue = 1.099298e-24
2. Adding x3, Deviance = 20.5635, Chi2Stat = 12.8573, PValue = 0.000336166
3. Adding x2, Deviance = 13.2658, Chi2Stat = 7.29767, PValue = 0.00690441
```

```
mdl2 =
```

```
Generalized linear regression model:
  logit(y) ~ 1 + x2 + x3 + x4
  Distribution = Binomial
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	50.527	23.995	2.1057	0.035227
x2	8.3761	4.7612	1.7592	0.078536
x3	-7.8745	3.8407	-2.0503	0.040334
x4	-21.43	10.707	-2.0014	0.04535

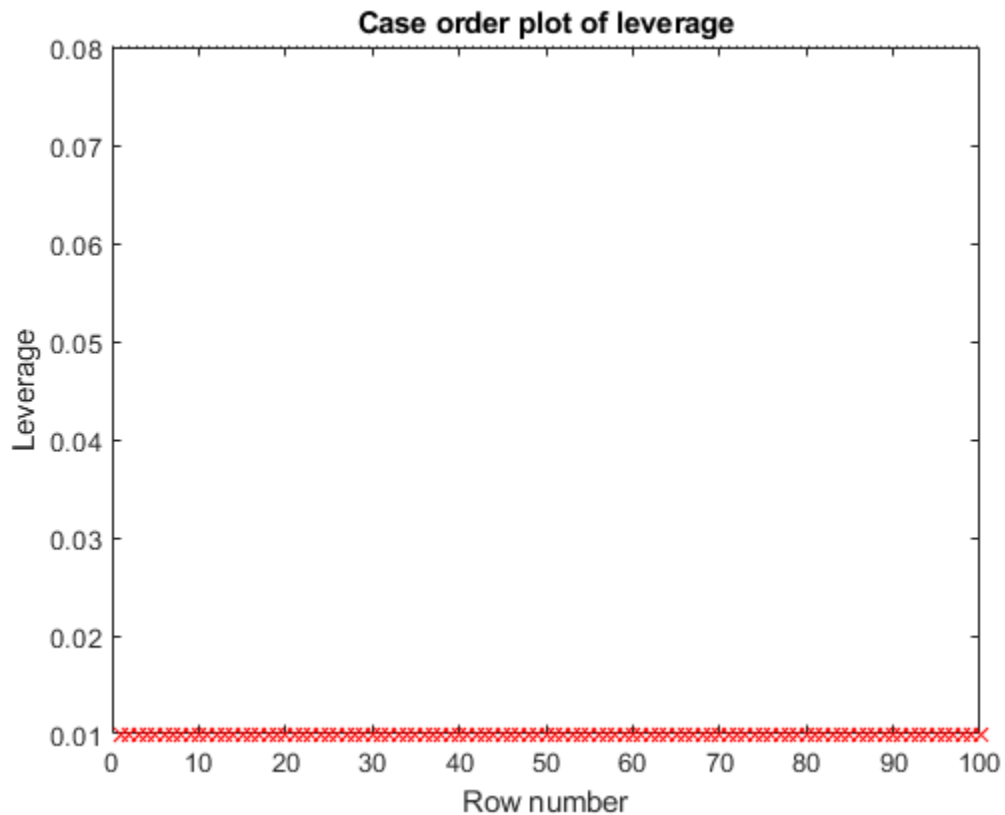
```
100 observations, 96 error degrees of freedom  
Dispersion: 1  
Chi^2-statistic vs. constant model: 125, p-value = 5.4e-27
```

stepwiseglm included 'x2' in the model, because it neither adds nor removes terms with  $p$ -values between 0.05 and 0.10.

#### Step 4. Look for outliers and exclude them.

Examine a leverage plot to look for influential outliers.

```
plotDiagnostics mdl2, 'leverage')
```



There is one observation with a leverage close to one. Using the Data Cursor, click the point, and find it has index 69.

See if the model coefficients change when you fit a model excluding this point.

```
oldCoeffs = mdl2.Coefficients.Estimate;
mdl3 = fitglm(X,y,'linear',...
    'distr','binomial','pred',2:4,'exclude',69);
newCoeffs = mdl3.Coefficients.Estimate;
disp([oldCoeffs newCoeffs])

    50.5268    50.5268
     8.3761     8.3761
    -7.8745    -7.8745
   -21.4296   -21.4296
```

The model coefficients do not change, suggesting that the response at the high-leverage point is consistent with the predicted value from the reduced model.

### Step 5. Predict the probability that a new flower is versicolor.

Use `mdl2` to predict the probability that a flower with average measurements is versicolor. Generate confidence intervals for your prediction.

```
[newf,newc] = predict(mdl2,mean(X))

newf =

    0.5086

newc =

    0.1863    0.8239
```

The model gives almost a 50% probability that the average flower is versicolor, with a wide confidence interval about this estimate.

## Lasso Regularization of Generalized Linear Models

<b>In this section...</b>
“What is Generalized Linear Model Lasso Regularization?” on page 12-46
“Generalized Linear Model Lasso and Elastic Net” on page 12-46
“References” on page 12-48

### What is Generalized Linear Model Lasso Regularization?

Lasso is a regularization technique. Use `lasso glm` to:

- Reduce the number of predictors in a generalized linear model.
- Identify important predictors.
- Select among redundant predictors.
- Produce shrinkage estimates with potentially lower predictive errors than ordinary least squares.

Elastic net is a related technique. Use it when you have several highly correlated variables. `lasso glm` provides elastic net regularization when you set the Alpha name-value pair to a number strictly between 0 and 1.

For details about lasso and elastic net computations and algorithms, see “Generalized Linear Model Lasso and Elastic Net” on page 12-46. For a discussion of generalized linear models, see “What Are Generalized Linear Models?” on page 12-12.

### Generalized Linear Model Lasso and Elastic Net

#### Overview of Lasso and Elastic Net

Lasso is a regularization technique for estimating generalized linear models. Lasso includes a penalty term that constrains the size of the estimated coefficients. Therefore, it resembles ridge regression on page 11-137. Lasso is a shrinkage estimator: it generates coefficient estimates that are biased to be small. Nevertheless, a lasso estimator can have smaller error than an ordinary maximum likelihood estimator when you apply it to new data.

Unlike ridge regression, as the penalty term increases, the lasso technique sets more coefficients to zero. This means that the lasso estimator is a smaller model, with fewer

predictors. As such, lasso is an alternative to stepwise regression on page 11-127 and other model selection and dimensionality reduction techniques.

Elastic net is a related technique. Elastic net is akin to a hybrid of ridge regression and lasso regularization. Like lasso, elastic net can generate reduced models by generating zero-valued coefficients. Empirical studies suggest that the elastic net technique can outperform lasso on data with highly correlated predictors.

### Definition of Lasso for Generalized Linear Models

For a nonnegative value of  $\lambda$ , `lasso` solves the problem

$$\min_{\beta_0, \beta} \left( \frac{1}{N} \text{Deviance}(\beta_0, \beta) + \lambda \sum_{j=1}^p |\beta_j| \right).$$

- The function `Deviance` in this equation is the deviance of the model fit to the responses using intercept  $\beta_0$  and predictor coefficients  $\beta$ . The formula for `Deviance` depends on the `dist` parameter you supply to `lasso`. Minimizing the  $\lambda$ -penalized deviance is equivalent to maximizing the  $\lambda$ -penalized log likelihood.
- $N$  is the number of observations.
- $\lambda$  is a nonnegative regularization parameter corresponding to one value of `Lambda`.
- Parameters  $\beta_0$  and  $\beta$  are a scalar and a vector of length  $p$ , respectively.

As  $\lambda$  increases, the number of nonzero components of  $\beta$  decreases.

The lasso problem involves the  $L^1$  norm of  $\beta$ , as contrasted with the elastic net algorithm.

### Definition of Elastic Net for Generalized Linear Models

For an  $\alpha$  strictly between 0 and 1, and a nonnegative  $\lambda$ , elastic net solves the problem

$$\min_{\beta_0, \beta} \left( \frac{1}{N} \text{Deviance}(\beta_0, \beta) + \lambda P_\alpha(\beta) \right),$$

where

$$P_\alpha(\beta) = \frac{(1-\alpha)}{2} \|\beta\|_2^2 + \alpha \|\beta\|_1 = \sum_{j=1}^p \left( \frac{(1-\alpha)}{2} \beta_j^2 + \alpha |\beta_j| \right)$$

Elastic net is the same as lasso when  $a = 1$ . For other values of  $a$ , the penalty term  $P_a(\beta)$  interpolates between the  $L^1$  norm of  $\beta$  and the squared  $L^2$  norm of  $\beta$ . As  $a$  shrinks toward 0, elastic net approaches ridge regression.

## References

- [1] Tibshirani, R. *Regression Shrinkage and Selection via the Lasso*. Journal of the Royal Statistical Society, Series B, Vol. 58, No. 1, pp. 267–288, 1996.
- [2] Zou, H. and T. Hastie. *Regularization and Variable Selection via the Elastic Net*. Journal of the Royal Statistical Society, Series B, Vol. 67, No. 2, pp. 301–320, 2005.
- [3] Friedman, J., R. Tibshirani, and T. Hastie. *Regularization Paths for Generalized Linear Models via Coordinate Descent*. Journal of Statistical Software, Vol. 33, No. 1, 2010. <http://www.jstatsoft.org/v33/i01>
- [4] Hastie, T., R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*, 2nd edition. Springer, New York, 2008.
- [5] McCullagh, P., and J. A. Nelder. *Generalized Linear Models*, 2nd edition. Chapman & Hall/CRC Press, 1989.



## Regularize Poisson Regression

This example shows how to identify and remove redundant predictors from a generalized linear model.

Create data with 20 predictors, and Poisson responses using just three of the predictors, plus a constant.

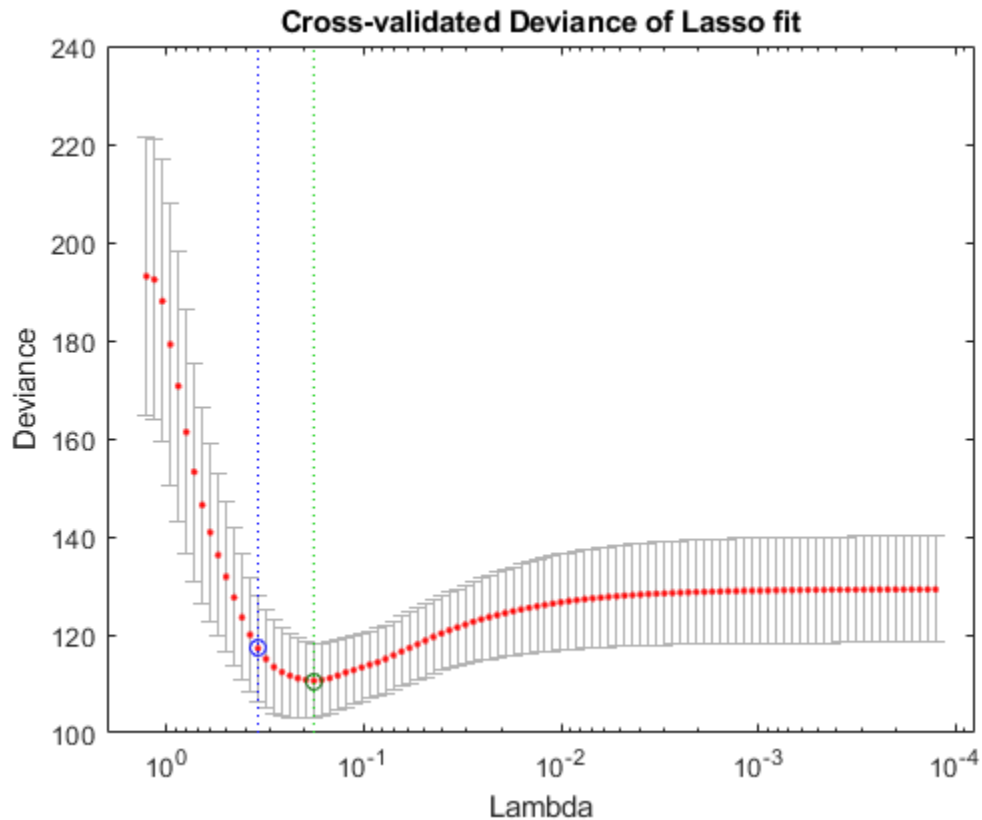
```
rng('default') % for reproducibility
X = randn(100,20);
mu = exp(X(:, [5 10 15]) * [.4; .2; .3] + 1);
y = poissrnd(mu);
```

Construct a cross-validated lasso regularization of a Poisson regression model of the data.

```
[B,FitInfo] = lassoglm(X,y,'poisson','CV',10);
```

Examine the cross-validation plot to see the effect of the Lambda regularization parameter.

```
lassoPlot(B,FitInfo,'plottype','CV');
```



The green circle and dashed line locate the  $\text{Lambda}$  with minimal cross-validation error. The blue circle and dashed line locate the point with minimal cross-validation error plus one standard deviation.

Find the nonzero model coefficients corresponding to the two identified points.

```
minpts = find(B(:,FitInfo.IndexMinDeviance))
```

```
minpts =
```

```
3
5
6
```

```
10
11
15
16
```

```
min1pts = find(B(:,FitInfo.Index1SE))
```

```
min1pts =
```

```
5
10
15
```

The coefficients from the minimal plus one standard error point are exactly those coefficients used to create the data.

Find the values of the model coefficients at the minimal plus one standard error point.

```
B(min1pts,FitInfo.Index1SE)
```

```
ans =
```

```
0.2903
0.0789
0.2081
```

The values of the coefficients are, as expected, smaller than the original  $[0.4, 0.2, 0.3]$ . Lasso works by "shrinkage," which biases predictor coefficients toward zero.

The constant term is in the `FitInfo.Intercept` vector.

```
FitInfo.Intercept(FitInfo.Index1SE)
```

```
ans =
```

```
1.0879
```

The constant term is near 1, which is the value used to generate the data.

## Regularize Logistic Regression

This example shows how to regularize binomial regression. The default (canonical) link function for binomial regression is the logistic function.

### Step 1. Prepare the data.

Load the `ionosphere` data. The response `Y` is a cell array of 'g' or 'b' characters. Convert the cells to logical values, with `true` representing 'g'. Remove the first two columns of `X` because they have some awkward statistical properties, which are beyond the scope of this discussion.

```
load ionosphere
Ybool = strcmp(Y, 'g');
X = X(:,3:end);
```

### Step 2. Create a cross-validated fit.

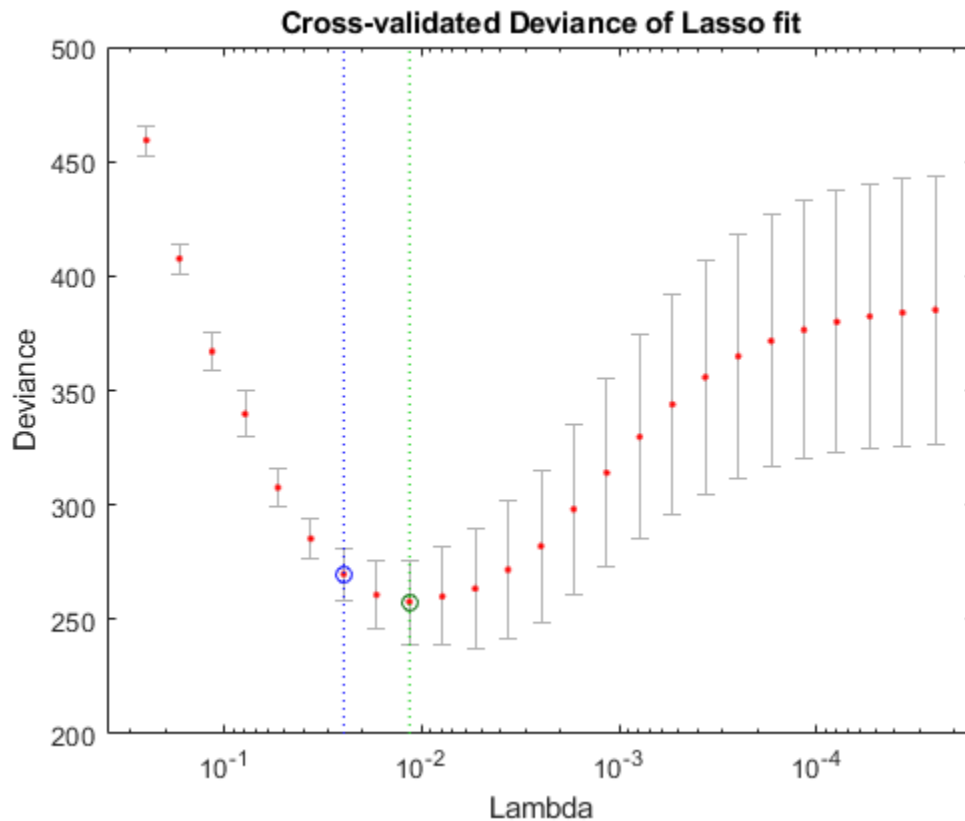
Construct a regularized binomial regression using 25 `Lambda` values and 10-fold cross validation. This process can take a few minutes.

```
rng('default') % for reproducibility
[B,FitInfo] = lassoglm(X,Ybool,'binomial',...
    'NumLambda',25,'CV',10);
```

### Step 3. Examine plots to find appropriate regularization.

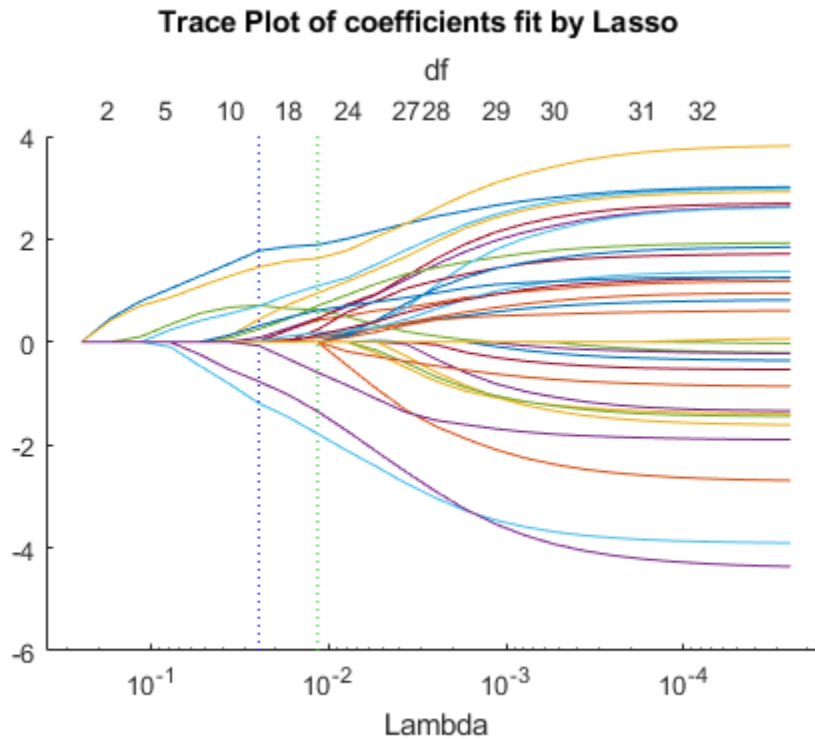
`lassoPlot` can give both a standard trace plot and a cross-validated deviance plot. Examine both plots.

```
lassoPlot(B,FitInfo,'PlotType','CV');
```



The plot identifies the minimum-deviance point with a green circle and dashed line as a function of the regularization parameter `Lambda`. The blue circled point has minimum deviance plus no more than one standard deviation.

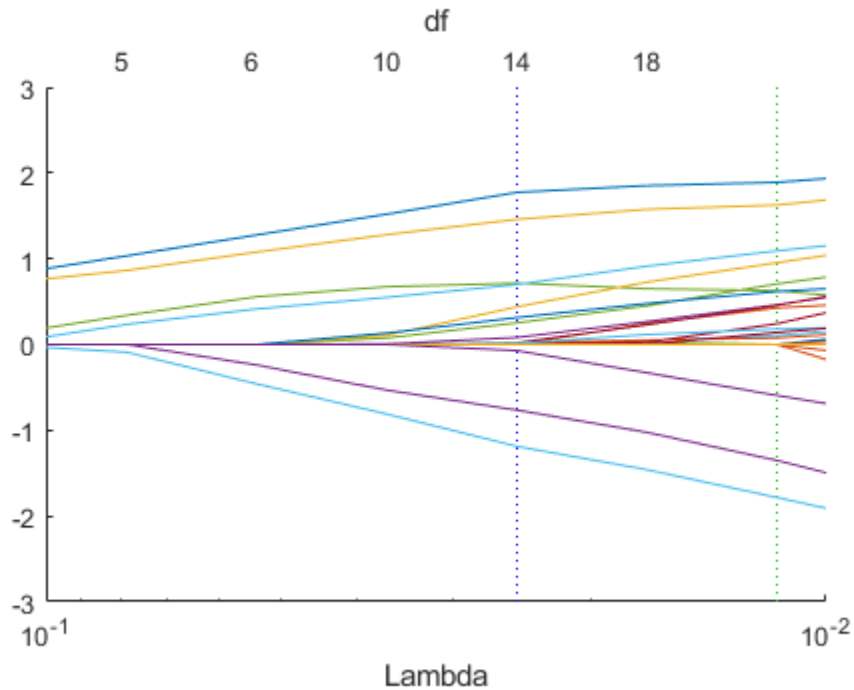
```
lassoPlot(B, FitInfo, 'PlotType', 'Lambda', 'XScale', 'log');
```



The trace plot shows nonzero model coefficients as a function of the regularization parameter `Lambda`. Because there are 32 predictors and a linear model, there are 32 curves. As `Lambda` increases to the left, `lassoglm` sets various coefficients to zero, removing them from the model.

The trace plot is somewhat compressed. Zoom in to see more detail.

```
xlim([.01 .1])
ylim([-3 3])
```



As Lambda increases toward the left side of the plot, fewer nonzero coefficients remain.

Find the number of nonzero model coefficients at the `Lambda` value with minimum deviance plus one standard deviation point. The regularized model coefficients are in column `FitInfo.Index1SE` of the B matrix.

```
indx = FitInfo.Index1SE;
B0 = B(:,indx);
nonzeros = sum(B0 ~= 0)
```

```
nonzeros =
```



When you set `Lambda` to `FitInfo.Index1SE`, `lassoglm` removes over half of the 32 original predictors.

#### Step 4. Create a regularized model.

The constant term is in the `FitInfo.Index1SE` entry of the `FitInfo.Intercept` vector. Call that value `cnst`.

The model is  $\text{logit}(\mu) = \log(\mu/(1 - \mu)) = X*B0 + \text{cnst}$ . Therefore, for predictions,  $\mu = \exp(X*B0 + \text{cnst}) / (1 + \exp(x*B0 + \text{cnst}))$ .

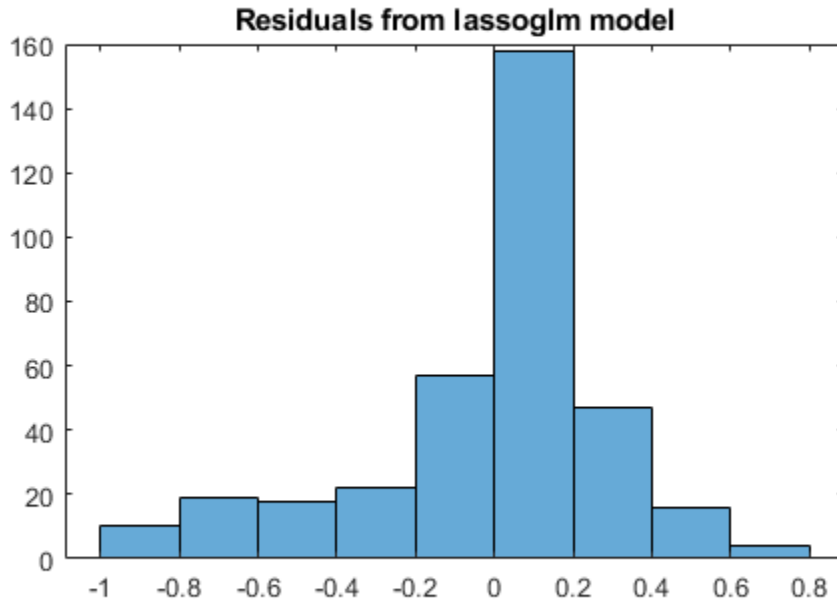
The `glmval` function evaluates model predictions. It assumes that the first model coefficient relates to the constant term. Therefore, create a coefficient vector with the constant term first.

```
cnst = FitInfo.Intercept(indx);  
B1 = [cnst;B0];
```

#### Step 5. Examine residuals.

Plot the training data against the model predictions for the regularized `lassoglm` model.

```
preds = glmval(B1,X,'logit');  
histogram(Ybool - preds) % plot residuals  
title('Residuals from lassoglm model')
```



**Step 6. Alternative: Use identified predictors in a least-squares generalized linear model.**

Instead of using the biased predictions from the model, you can make an unbiased model using just the identified predictors.

```
predictors = find(B0); % indices of nonzero predictors
mdl = fitglm(X,Ybool,'linear',...
    'Distribution','binomial','PredictorVars',predictors)
```

```
mdl =
```

```
Generalized linear regression model:
```

```
y ~ [Linear formula with 15 terms in 14 predictors]
Distribution = Binomial
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	-2.9367	0.50926	-5.7666	8.0893e-09
x1	2.492	0.60795	4.099	4.1502e-05
x3	2.5501	0.63304	4.0284	5.616e-05
x4	0.48816	0.50336	0.9698	0.33215
x5	0.6158	0.62192	0.99015	0.3221
x6	2.294	0.5421	4.2317	2.3198e-05
x7	0.77842	0.57765	1.3476	0.1778
x12	1.7808	0.54316	3.2786	0.0010432
x16	-0.070993	0.50515	-0.14054	0.88823
x20	-2.7767	0.55131	-5.0365	4.7402e-07
x24	2.0212	0.57639	3.5067	0.00045372
x25	-2.3796	0.58274	-4.0835	4.4363e-05
x27	0.79564	0.55904	1.4232	0.15467
x29	1.2689	0.55468	2.2876	0.022162
x32	-1.5681	0.54336	-2.8859	0.0039035

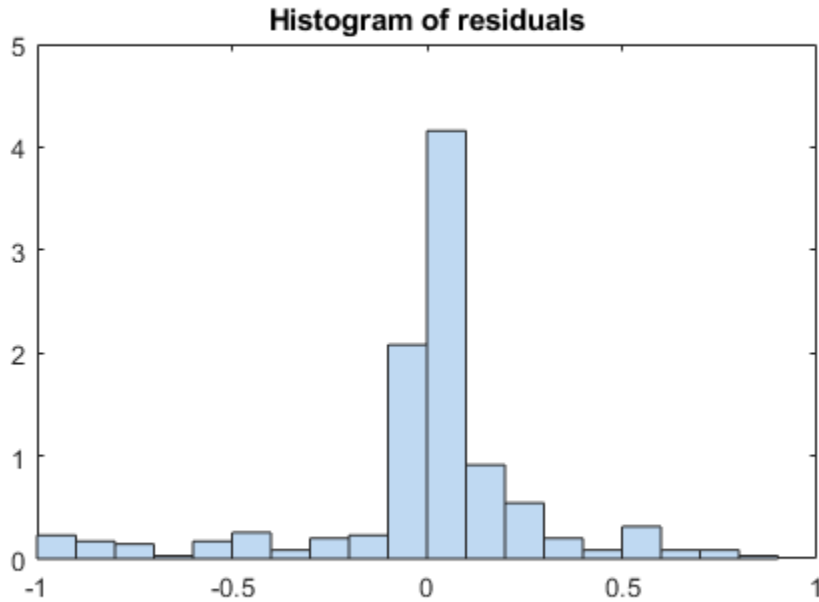
351 observations, 336 error degrees of freedom

Dispersion: 1

Chi<sup>2</sup>-statistic vs. constant model: 262, p-value = 1e-47

**Plot the residuals of the model.**

```
plotResiduals (mdl)
```



As expected, residuals from the least-squares model are slightly smaller than those of the regularized model. However, this does not mean that `mdl` is a better predictor for new data.

## Regularize Wide Data in Parallel

This example shows how to regularize a model with many more predictors than observations. *Wide data* is data with more predictors than observations. Typically, with wide data you want to identify important predictors. Use `lassoglm` as an exploratory or screening tool to select a smaller set of variables to prioritize your modeling and research. Use parallel computing to speed up cross validation.

Load the `ovariancancer` data. This data has 216 observations and 4000 predictors in the `obs` workspace variable. The responses are binary, either 'Cancer' or 'Normal', in the `grp` workspace variable. Convert the responses to binary for use in `lassoglm`.

```
load ovariancancer
y = strcmp(grp, 'Cancer');
```

Set options to use parallel computing. Prepare to compute in parallel using `parpool`.

```
opt = statset('UseParallel',true);
parpool()
```

```
Starting parallel pool (parpool) using the 'local' profile ...
connected to 12 workers.
```

```
ans =
```

```
Pool with properties:
```

```
    Connected: true
   NumWorkers: 12
      Cluster: local
 AttachedFiles: {}
AutoAddClientPath: true
   IdleTimeout: 30 minutes (30 minutes remaining)
   SpmdEnabled: true
```

Fit a cross-validated set of regularized models. Use the Alpha parameter to favor retaining groups of highly correlated predictors, as opposed to eliminating all but one member of the group. Commonly, you use a relatively large value of Alpha.

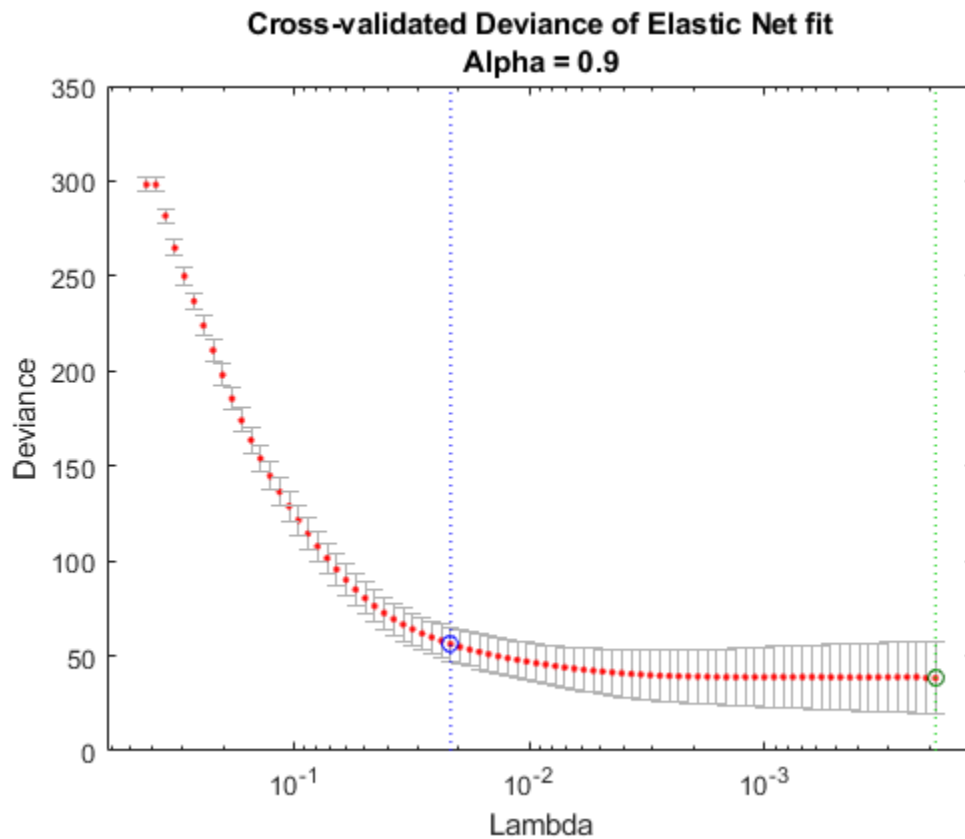
```
rng('default') % for reproducibility
tic
[B,S] = lassoglm(obs,y, 'binomial', 'NumLambda',100, ...
```

```
'Alpha',0.9,'LambdaRatio',1e-4,'CV',10,'Options',opt);
toc
```

Elapsed time is 149.196622 seconds.

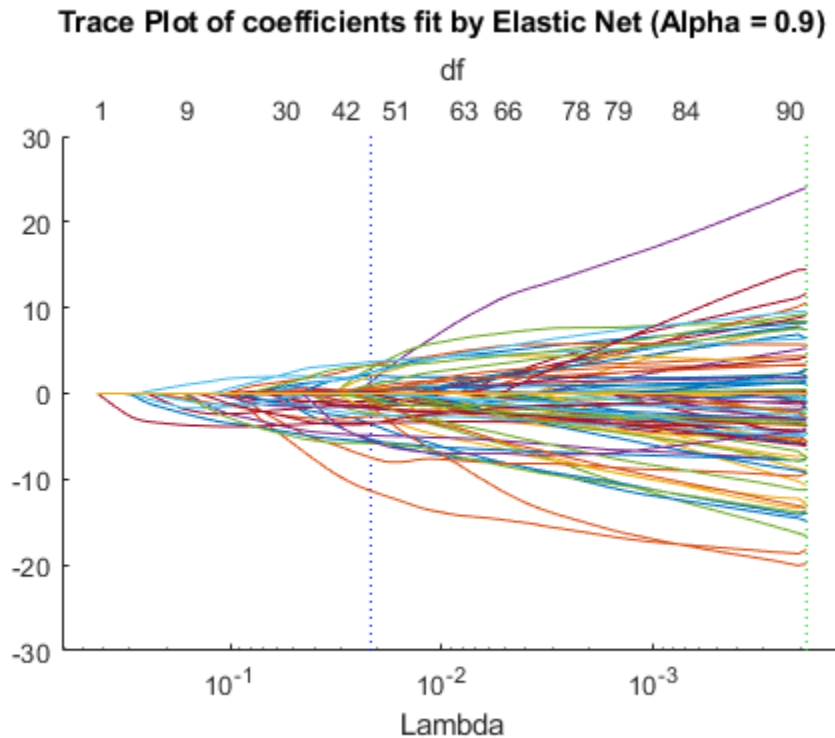
Examine cross-validation plot.

```
lassoPlot(B,S,'PlotType','CV');
```



Examine trace plot.

```
lassoPlot(B,S,'PlotType','Lambda','XScale','log')
```



The right (green) vertical dashed line represents the `Lambda` providing the smallest cross-validated deviance. The left (blue) dashed line has the minimal deviance plus no more than one standard deviation. This blue line has many fewer predictors:

```
[S.DF(S.Index1SE) S.DF(S.IndexMinDeviance)]
```

```
ans =
```

```
50    89
```

You asked `lassoglm` to fit using 100 different `Lambda` values. How many did it use?

```
size(B)
```

```
ans =  
  
      4000      84
```

`lassoglm` stopped after 84 values because the deviance was too small for small `Lambda` values. To avoid overfitting, `lassoglm` halts when the deviance of the fitted model is too small compared to the deviance in the binary responses, ignoring the predictor variables.

You can force `lassoglm` to include more terms by explicitly providing a set of `Lambda` values.

```
minLambda = min(S.Lambda);  
explicitLambda = [minLambda*[.1 .01 .001] S.Lambda];  
[B2,S2] = lassoglm(obs,y,'binomial','Lambda',explicitLambda,...  
    'LambdaRatio',1e-4, 'CV',10,'Options',opt);  
length(S2.Lambda)
```

```
ans =  
  
      87
```

`lassoglm` used the three smaller values in fitting.

To save time, you can use:

- Fewer `Lambda`, meaning fewer fits
- Fewer cross-validation folds
- A larger value for `LambdaRatio`

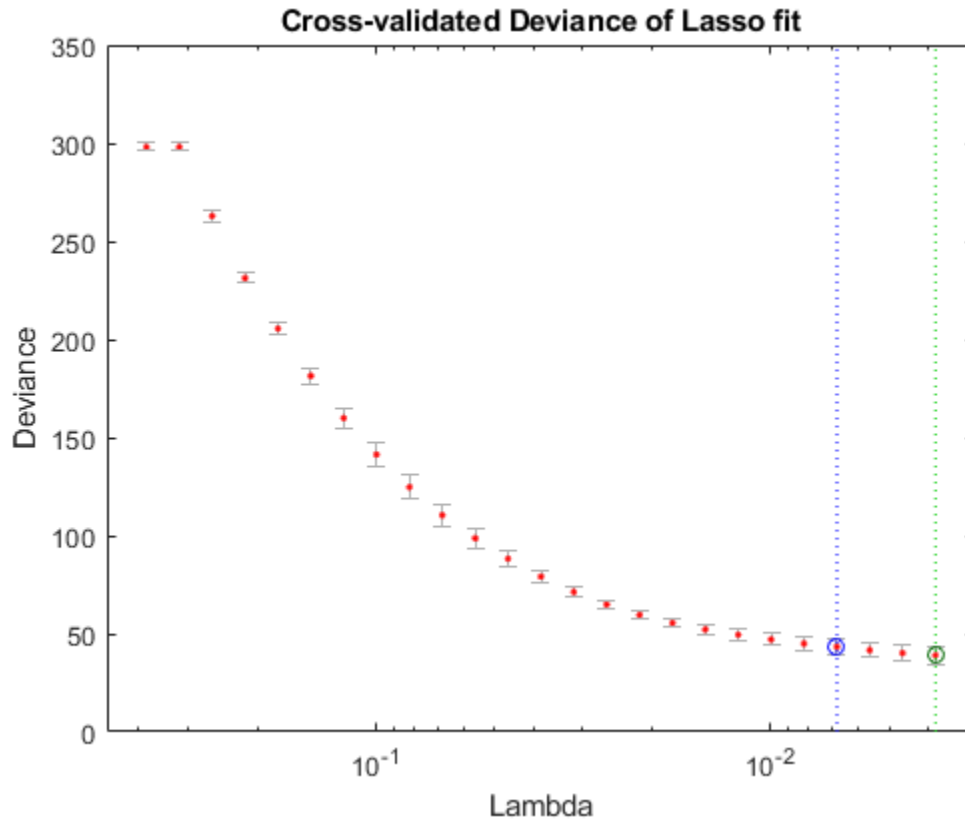
Use serial computation and all three of these time-saving methods:

```
tic  
[Bquick,Squick] = lassoglm(obs,y,'binomial','NumLambda',25,...  
    'LambdaRatio',1e-2,'CV',5);  
toc  
%  
% Elapsed time is 51.708074 seconds.  
  
Elapsed time is 47.856922 seconds.
```

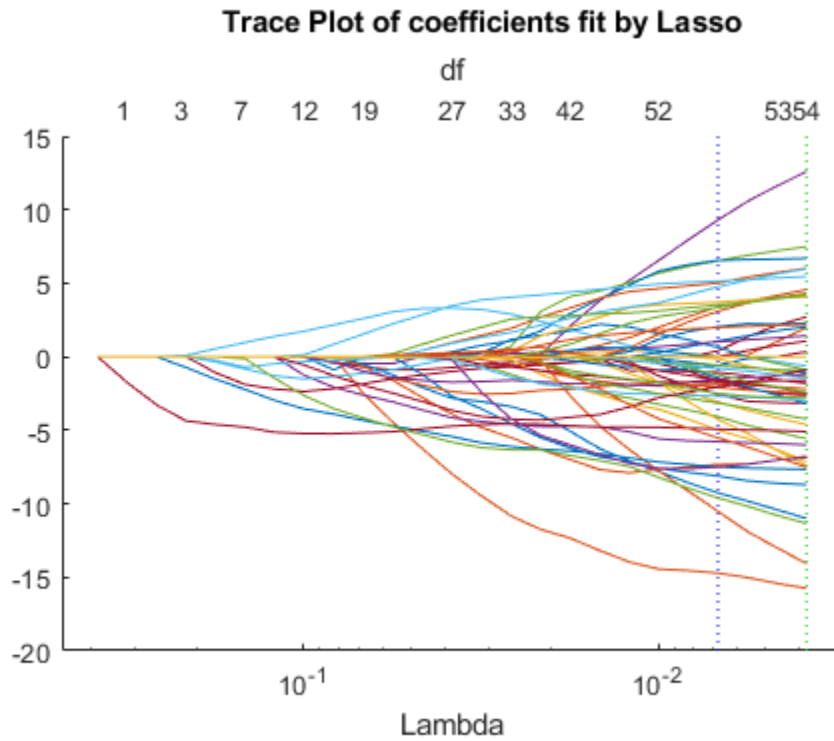


Graphically compare the new results to the first results.

```
lassoPlot(Bquick,Squick,'PlotType','CV');
```



```
lassoPlot(Bquick,Squick,'PlotType','Lambda','XScale','log')
```



The number of nonzero coefficients in the lowest plus one standard deviation model is around 50, similar to the first computation.

## Generalized Linear Mixed-Effects Models

### In this section...

“What Are Generalized Linear Mixed-Effects Models?” on page 12-67

“GLME Model Equations” on page 12-67

“Prepare Data for Model Fitting” on page 12-69

“Choose a Distribution Type for the Model” on page 12-69

“Choose a Link Function for the Model” on page 12-70

“Specify the Model Formula” on page 12-71

“Display the Model” on page 12-74

“Work with the Model” on page 12-76

### What Are Generalized Linear Mixed-Effects Models?

Generalized linear mixed-effects (GLME) models describe the relationship between a response variable and independent variables using coefficients that can vary with respect to one or more grouping variables, for data with a response variable distribution other than normal. You can think of GLME models as extensions of generalized linear models on page 12-12 (GLM) for data that are collected and summarized in groups. Alternatively, you can think of GLME models as a generalization of linear mixed-effects models on page 11-162 (LME) for data where the response variable is not normally distributed.

A mixed-effects model consists of fixed-effects and random-effects terms. Fixed-effects terms are usually the conventional linear regression part of the model. Random-effects terms are associated with individual experimental units drawn at random from a population, and account for variations between groups that might affect the response. The random effects have prior distributions, whereas the fixed effects do not.

### GLME Model Equations

The standard form of a generalized linear mixed-effects model is

$$y_i | b \sim \text{Distr} \left( \mu_i, \frac{\sigma^2}{w_i} \right)$$

$$g(\mu) = X\beta + Zb + \delta ,$$

where

- $y$  is an  $n$ -by-1 response vector, and  $y_i$  is its  $i$ th element.
- $b$  is the random-effects vector.
- $Distr$  is a specified conditional distribution of  $y$  given  $b$ .
- $\mu$  is the conditional mean of  $y$  given  $b$ , and  $\mu_i$  is its  $i$ th element.
- $\sigma^2$  is the dispersion parameter.
- $w$  is the effective observation weight vector, and  $w_i$  is the weight for observation  $i$ .
  - For a binomial distribution, the effective observation weight is equal to the prior weight specified using the 'Weights' name-value pair argument in `fitglm`, multiplied by the binomial size specified using the 'BinomialSize' name-value pair argument.
  - For all other distributions, the effective observation weight is equal to the prior weight specified using the 'Weights' name-value pair argument in `fitglm`.
- $g(\mu)$  is a link function that defines the relationship between the mean response  $\mu$  and the linear combination of the predictors.
- $X$  is an  $n$ -by- $p$  fixed-effects design matrix.
- $\beta$  is a  $p$ -by-1 fixed-effects vector.
- $Z$  is an  $n$ -by- $q$  random-effects design matrix.
- $b$  is a  $q$ -by-1 random-effects vector.
- $\delta$  is a model offset vector.

The model for the mean response  $\mu$  is

$$\mu = g^{-1}(\eta) ,$$

where  $g^{-1}$  is inverse of the link function  $g(\mu)$ , and  $\hat{\eta}_{ME}$  is the linear predictor of the fixed and random effects of the generalized linear mixed-effects model

$$\eta = X\beta + Zb + \delta .$$

A GLME model is parameterized by  $\beta$ ,  $\theta$ , and  $\sigma^2$ .

The assumptions for generalized linear mixed-effects models are:

- The random effects vector  $b$  has the prior distribution:

$$b | \sigma^2, \theta \sim N(0, \sigma^2 D(\theta)),$$

where  $\sigma^2$  is the dispersion parameter, and  $D$  is a symmetric and positive semidefinite matrix parameterized by an unconstrained parameter vector  $\theta$ .

- The observations  $y_i$  are conditionally independent given  $b$ .

## Prepare Data for Model Fitting

To fit a GLME model to your data, use `fitglme`. Format your input data using the `table` data type. Each row of the table represents one observation, and each column represents one predictor variable. For more information on creating and using `table`, see “Create and Work with Tables” (MATLAB).

Input data can include continuous and grouping variables. `fitglme` assumes that predictors using the following data types are categorical:

- Logical
- Categorical
- Character vector or character array

If the input data table contains any NaN values, then `fitglme` excludes that entire row of data from the fit. To exclude additional rows of data, you can use the 'Exclude' name-value pair argument of `fitglme` when fitting the model.

## Choose a Distribution Type for the Model

GLME models are used when the response data does not follow a normal distribution. Therefore, when fitting a model using `fitglme`, you must specify the response distribution type using the 'Distribution' name-value pair argument. Often, the type of response data suggests the appropriate distribution type for the model.

Type of Response Data	Suggested Response Distribution Type
Any real number	'Normal'
Any positive number	'Gamma' or 'InverseGaussian'
Any nonnegative integer	'Poisson'

Type of Response Data	Suggested Response Distribution Type
Integer from 0 to $n$ , where $n$ is a fixed positive value	'Binomial'

## Choose a Link Function for the Model

GLME models use a link function,  $g$ , to map the relationship between the mean response and the linear combination of the predictors. By default, `fitglme` uses a predefined, commonly accepted link function based on the specified distribution of the response data, as shown in the following table. However, you can specify a different link function from the list of predefined functions, or define your own, using the 'Link' name-value pair argument of `fitglme`.

Value	Description
'comploglog'	$g(\mu) = \log(-\log(1-\mu))$
'identity'	$g(\mu) = \mu$ Canonical link for the normal distribution.
'log'	$g(\mu) = \log(\mu)$ Canonical link for the Poisson distribution.
'logit'	$g(\mu) = \log(\mu/(1-\mu))$ Canonical link for the binomial distribution.
'loglog'	$g(\mu) = \log(-\log(\mu))$
'probit'	$g(\mu) = \text{norminv}(\mu)$
'reciprocal'	$g(\mu) = \mu.^{-1}$
Scalar value $P$	$g(\mu) = \mu.^P$

Value	Description
Structure S	<p>A structure containing four fields whose values are function handles:</p> <ul style="list-style-type: none"> <li>• S.Link — Link function</li> <li>• S.Derivative — Derivative</li> <li>• S.SecondDerivative — Second derivative</li> <li>• S.Inverse — Inverse of link</li> </ul> <p>If 'FitMethod' is 'MPL' or 'REML', or if S represents a canonical link for the specified distribution, you can omit the specification of S.SecondDerivative.</p>

When fitting a model to data, `fitglm` uses the canonical link function by default.

Distribution	Default Link Function
'Normal'	'identity'
'Binomial'	'logit'
'Poisson'	'log'
'Gamma'	-1
'InverseGaussian'	-2

The link functions 'comploglog', 'loglog', and 'probit' are mainly useful for binomial models.

## Specify the Model Formula

Model specification for `fitglm` uses Wilkinson notation, which is a character vector of the form 'y ~ terms', where y is the response variable name, and terms is written in the following notation.

Wilkinson Notation	Factors in Standard Notation
1	Constant (intercept) term
X^k, where k is a positive integer	X, X <sup>2</sup> , ..., X <sup>k</sup>

Wilkinson Notation	Factors in Standard Notation
$X1 + X2$	$X1, X2$
$X1 * X2$	$X1, X2, X1.*X2$ (element-wise multiplication of $X1$ and $X2$ )
$X1 : X2$	$X1.*X2$ only
$- X2$	Do not include $X2$
$X1 * X2 + X3$	$X1, X2, X3, X1 * X2$
$X1 + X2 + X3 + X1 : X2$	$X1, X2, X3, X1 * X2$
$X1 * X2 * X3 - X1 : X2 : X3$	$X1, X2, X3, X1 * X2, X1 * X3, X2 * X3$
$X1 * (X2 + X3)$	$X1, X2, X3, X1 * X2, X1 * X3$

Formulas include a constant (intercept) term by default. To exclude a constant term from the model, include  $-1$  in the formula.

For generalized linear mixed-effects models, the formula specification is of the form ' $y \sim \text{fixed} + (\text{random1}|\text{grouping1}) + \dots + (\text{randomR}|\text{groupingR})$ ', where *fixed* and *random* contain the fixed-effects and the random-effects terms, respectively.

Suppose the input data table contains the following:

- A response variable,  $y$
- Predictor variables,  $X1, X2, \dots, XJ$ , where  $J$  is the total number of predictor variables (including continuous and grouping variables).
- Grouping variables,  $g1, g2, \dots, gR$ , where  $R$  is the number of grouping variables.

The grouping variables in  $XJ$  and  $gR$  can be categorical, logical, character arrays, or cell arrays of character vectors.

Then, in a formula of the form ' $y \sim \text{fixed} + (\text{random1}|g1) + \dots + (\text{randomR}|gR)$ ', the term *fixed* corresponds to a specification of the fixed-effects design matrix  $X$ , *random1* is a specification of the random-effects design matrix  $Z1$  corresponding to grouping variable  $g1$ , and similarly *randomR* is a specification of the random-effects design matrix  $ZR$  corresponding to grouping variable  $gR$ . You can express the *fixed* and *random* terms using Wilkinson notation as follows.



Formula	Description
'y ~ X1 + X2'	Fixed effects for the intercept, X1, and X2. This formula is equivalent to 'y ~ 1 + X1 + X2'.
'y ~ -1 + X1 + X2'	No intercept, with fixed effects for X1 and X2. The implicit intercept term is suppressed by including -1.
'y ~ 1 + (1   g1)'	A fixed effect for the intercept, plus a random effect for the intercept for each level of the grouping variable g1.
'y ~ X1 + (1   g1)'	Random intercept model with a fixed slope.
'y ~ X1 + (X1   g1)'	Random intercept and slope, with possible correlation between them. This formula is equivalent to 'y ~ 1 + X1 + (1 + X1   g1)'.
'y ~ X1 + (1   g1) + (-1 + X1   g1)'	Independent random-effects terms for intercept and slope.
'y ~ 1 + (1   g1) + (1   g2) + (1   g1:g2)'	Random intercept model with independent main effects for g1 and g2, plus an independent interaction effect.

For example, the sample data `mfr` contains simulated data from a manufacturing company that operates 50 factories across the world. Each factory runs a batch process to create a finished product. The company wants to decrease the number of defects in each batch, so it developed a new manufacturing process. To test the effectiveness of the new process, the company selected 20 of its factories at random to participate in an experiment: Ten factories implemented the new process, while the other ten continued to run the old process. In each of the 20 factories, the company ran five batches (for a total of 100 batches), and recorded data on processing time (`time_dev`), temperature (`temp_dev`), number of defects (`defects`), and a categorical variable indicating the raw materials supplier (`supplier`) for each batch.

To determine whether the new process (represented by the predictor variable `newprocess`) significantly reduces the number of defects, fit a GLME model using `newprocess`, `time_dev`, `temp_dev`, and `supplier` as fixed-effects predictors. Include a random-effects intercept grouped by `factory`, to account for quality differences that

might exist due to factory-specific variations. The response variable `defects` has a Poisson distribution.

The number of defects can be modeled using a Poisson distribution

$$defects_{ij} \sim \text{Poisson}(\mu_{ij})$$

This corresponds to the generalized linear mixed-effects model

$$\log(\mu_{ij}) = \beta_0 + \beta_1 \text{newprocess}_{ij} + \beta_2 \text{time\_dev}_{ij} \\ + \beta_3 \text{temp\_dev}_{ij} + \beta_4 \text{supplier\_C}_{ij} + \beta_5 \text{supplier\_B}_{ij} + b_i,$$

where

- $defects_{ij}$  is the number of defects observed in the batch produced by factory  $i$  (where  $i = 1, 2, \dots, 20$ ) during batch  $j$  (where  $j = 1, 2, \dots, 5$ ).
- $\mu_{ij}$  is the mean number of defects corresponding to factory  $i$  during batch  $j$ .
- $supplier\_C_{ij}$  and  $supplier\_B_{ij}$  are dummy variables that indicate whether company C or B, respectively, supplied the process chemicals for the batch produced by factory  $i$  during batch  $j$ .
- $b_i \sim N(0, \sigma_b^2)$  is a random-effects intercept for each factory  $i$  that accounts for factory-specific variation in quality.

Using Wilkinson notation, specify this model as:

```
'defects ~ 1 + newprocess + time_dev + temp_dev + supplier + (1|
factory)'
```

To account for the Poisson distribution of the response variable, when fitting the model using `fitglm`, specify the `'Distribution'` name-value pair argument as `'Poisson'`. By default, `fitglm` uses a log link function for response variables with a Poisson distribution.

## Display the Model

The output of the fitting function `fitglm` provides information about generalized linear mixed-effects model.

Using the `mfr` manufacturing experiment data, fit a model using `newprocess`, `time_dev`, `temp_dev`, and `supplier` as fixed-effects predictors. Specify the response distribution as Poisson, the link function as log, and the fit method as Laplace.

```
load mfr

glme = fitglme(mfr, ...
              'defects ~ 1 + newprocess + time_dev + temp_dev + supplier + (1|factory)', ...
              'Distribution', 'Poisson', 'Link', 'log', 'FitMethod', 'Laplace', ...
              'DummyVarCoding', 'effects')

glme =

Generalized linear mixed-effects model fit by ML

Model information:
  Number of observations      100
  Fixed effects coefficients    6
  Random effects coefficients  20
  Covariance parameters       1
  Distribution                 Poisson
  Link                        Log
  FitMethod                   Laplace

Formula:
  defects ~ 1 + newprocess + time_dev + temp_dev + supplier + (1 | factory)

Model fit statistics:
  AIC      BIC      LogLikelihood      Deviance
  416.35   434.58   -201.17      402.35

Fixed effects coefficients (95% CIs):
  Name      Estimate      SE      tStat      DF      pValue
  '(Intercept)''      1.4689      0.15988      9.1875      94      9.8194e-15
  'newprocess''      -0.36766      0.17755      -2.0708      94      0.041122
  'time_dev''      -0.094521      0.82849      -0.11409      94      0.90941
  'temp_dev''      -0.28317      0.9617      -0.29444      94      0.76907
  'supplier_C''      -0.071868      0.078024      -0.9211      94      0.35936
  'supplier_B''      0.071072      0.07739      0.91836      94      0.36078

  Lower      Upper
  1.1515      1.7864
  -0.72019      -0.015134
  -1.7395      1.5505
  -2.1926      1.6263
  -0.22679      0.083051
  -0.082588      0.22473

Random effects covariance parameters:
Group: factory (20 Levels)
  Name1      Name2      Type      Estimate
  '(Intercept)''      '(Intercept)''      'std''      0.31381

Group: Error
  Name      Estimate
  'sqrt(Dispersion)''      1
```

The `Model` information table displays the total number of observations in the sample data (100), the number of fixed- and random-effects coefficients (6 and 20, respectively), and the number of covariance parameters (1). It also indicates that the response variable has a `Poisson` distribution, the link function is `Log`, and the fit method is `Laplace`.

`Formula` indicates the model specification using Wilkinson's notation.

The `Model fit statistics` table displays statistics used to assess the goodness of fit of the model. This includes the Akaike information criterion (AIC), Bayesian information criterion (BIC) values, log likelihood (`LogLikelihood`), and deviance (`Deviance`) values.

The `Fixed effects coefficients` table indicates that `fitglm` returned 95% confidence intervals. It contains one row for each fixed-effects predictor, and each column contains statistics corresponding to that predictor. Column 1 (`Name`) contains the name of each fixed-effects coefficient, column 2 (`Estimate`) contains its estimated value, and column 3 (`SE`) contains the standard error of the coefficient. Column 4 (`tStat`) contains the *t*-statistic for a hypothesis test that the coefficient is equal to 0. Column 5 (`DF`) and column 6 (`pValue`) contain the degrees of freedom and *p*-value that correspond to the *t*-statistic, respectively. The last two columns (`Lower` and `Upper`) display the lower and upper limits, respectively, of the 95% confidence interval for each fixed-effects coefficient.

`Random effects covariance parameters` displays a table for each grouping variable (here, only `factory`), including its total number of levels (20), and the type and estimate of the covariance parameter. Here, `std` indicates that `fitglm` returns the standard deviation of the random effect associated with the `factory` predictor, which has an estimated value of 0.31381. It also displays a table containing the error parameter type (here, the square root of the dispersion parameter), and its estimated value of 1.

The standard display generated by `fitglm` does not provide confidence intervals for the random-effects parameters. To compute and display these values, use `covarianceParameters`.

## Work with the Model

After you create a GLME model using `fitglm`, you can use additional functions to work with the model.

## Inspect and Test Coefficients and Confidence Intervals

To extract estimates of the fixed- and random-effects coefficients, covariance parameters, design matrices, and related statistics:

- `fixedEffects` extracts estimated fixed-effects coefficients and related statistics from a fitted model. Related statistics include the standard error; the  $t$ -statistic, degrees of freedom, and  $p$ -value for a hypothesis test of whether each parameter is equal to 0; and the confidence intervals.
- `randomEffects` extracts estimated random-effects coefficients and related statistics from a fitted GLME model. Related statistics include the estimated empirical Bayes predictor (EBP) of each random effect, the square root of the conditional mean squared error of prediction (CMSEP) given the covariance parameters and the response; the  $t$ -statistic, estimated degrees of freedom, and  $p$ -value for a hypothesis test of whether each random effect is equal to 0; and the confidence intervals.
- `covarianceParameters` extracts estimated covariance parameters and related statistics from a fitted GLME model. Related statistics include estimate of the covariance parameter, and the confidence intervals.
- `designMatrix` extracts the fixed- and random-effects design matrices, or a specified subset thereof, from the fitted GLME model.

To conduct customized hypothesis tests for the significance of fixed- and random-effects coefficients, and to compute custom confidence intervals:

- `anova` performs a marginal  $F$ -test (hypothesis test) on fixed-effects terms, to determine if all coefficients representing the fixed-effects terms are equal to 0. You can use `anova` to test the combined significance of the coefficients of categorical predictors.
- `coefCI` computes confidence intervals for fixed- and random-effects parameters from a fitted GLME model. By default, `fitglme` computes 95% confidence intervals. Use `coefCI` to compute the boundaries at a different confidence level.
- `coefTest` performs custom hypothesis tests on fixed-effects or random-effects vectors of a fitted generalized linear mixed-effects model. For example, you can specify contrast matrices.

## Generate New Response Values and Refit Model

To generate new response values, including fitted, predicted, and random responses, based on the fitted GLME model:

- `fitted` computes fitted response values using the original predictor values, and the estimated coefficient and parameter values from the fitted model.
- `predict` computes the predicted conditional or marginal mean of the response using either the original predictor values or new predictor values, and the estimated coefficient and parameter values from the fitted model.
- `random` generates random responses from a fitted model.
- `refit` creates a new fitted GLME model, based on the original model and a new response vector.

### Inspect and Visualize Residuals

To extract and visualize residuals from the fitted GLME model:

- `residuals` extracts the raw or Pearson residuals from the fitted model. You can also specify whether to compute the conditional or marginal residuals.
- `plotResiduals` creates plots using the raw or Pearson residuals from the fitted model, including:
  - A histogram of the residuals
  - A scatterplot of the residuals versus fitted values
  - A scatterplot of residuals versus lagged residuals

### See Also

`GeneralizedLinearMixedModel` | `fitglm`

### Related Examples

- “Fit a Generalized Linear Mixed-Effects Model” on page 12-79

## Fit a Generalized Linear Mixed-Effects Model

This example shows how to fit a generalized linear mixed-effects model (GLME) to sample data.

### Load the sample data.

Navigate to the folder containing the sample data. Load the sample data.

```
cd(matlabroot)
cd('help/toolbox/stats/examples')

load mfr
```

A manufacturing company operates 50 factories across the world, and each runs a batch process to create a finished product. The company wants to decrease the number of defects in each batch, so it developed a new manufacturing process. However, the company wants to test the new process in select factories to ensure that it is effective before rolling it out to all 50 locations.

To test whether the new process significantly reduces the number of defects in each batch, the company selected 20 of its factories at random to participate in an experiment. Ten factories implemented the new process, while the other ten used the old process.

In each of the 20 factories ( $i = 1, 2, \dots, 20$ ), the company ran five batches ( $j = 1, 2, \dots, 5$ ) and recorded the following data in the table `mfr`:

- Flag to indicate use of the new process:
  - If the batch used the new process, then `newprocess = 1`
  - If the batch used the old process, then `newprocess = 0`
- Processing time for the batch, in hours (`time`)
- Temperature of the batch, in degrees Celsius (`temp`)
- Supplier of the chemical used in the batch (`supplier`)
  - `supplier` is a categorical variable with levels A, B, and C, where each level represents one of the three suppliers
- Number of defects in the batch (`defects`)

The data also includes `time_dev` and `temp_dev`, which represent the absolute deviation of time and temperature, respectively, from the process standard of 3 hours and 20

degrees Celsius. The response variable `defects` has a Poisson distribution. This is simulated data.



The company wants to determine whether the new process significantly reduces the number of defects in each batch, while accounting for quality differences that might exist due to factory-specific variations in time, temperature, and supplier. The number of defects per batch can be modeled using a Poisson distribution:

$$defects_{ij} \sim \text{Poisson}(\mu_{ij})$$

Use a generalized linear mixed-effects model to model the number of defects per batch:

$$\log(\mu_{ij}) = \beta_0 + \beta_1 newprocess_{ij} + \beta_2 time\_dev_{ij} + \beta_3 temp\_dev_{ij} + \beta_4 supplier\_C_{ij} + \beta_5 supplier\_B_{ij} + b_i,$$

where

- $defects_{ij}$  is the number of defects observed in the batch produced by factory  $i$  during batch  $j$ .
- $\mu_{ij}$  is the mean number of defects corresponding to factory  $i$  (where  $i = 1, 2, \dots, 20$ ) during batch  $j$  (where  $j = 1, 2, \dots, 5$ ).
- $newprocess_{ij}$ ,  $time\_dev_{ij}$ , and  $temp\_dev_{ij}$  are the measurements for each variable that correspond to factory  $i$  during batch  $j$ . For example,  $newprocess_{ij}$  indicates whether the batch produced by factory  $i$  during batch  $j$  used the new process.
- $supplier\_C_{ij}$  and  $supplier\_B_{ij}$  are dummy variables that use effects (sum-to-zero) coding to indicate whether company C or B, respectively, supplied the process chemicals for the batch produced by factory  $i$  during batch  $j$ .
- $b_i \sim N(0, \sigma_b^2)$  is a random-effects intercept for each factory  $i$  that accounts for factory-specific variation in quality.

### Fit a GLME model and interpret the results.

Fit a generalized linear mixed-effects model using `newprocess`, `time_dev`, `temp_dev`, and `supplier` as fixed-effects predictors. Include a random-effects term for intercept grouped by `factory`, to account for quality differences that might exist due to factory-



specific variations. The response variable `defects` has a Poisson distribution, and the appropriate link function for this model is `log`. Use the Laplace fit method to estimate the coefficients. Specify the dummy variable encoding as `'effects'`, so the dummy variable coefficients sum to 0.

```
glme = fitglme(mfr,...
'defects ~ 1 + newprocess + time_dev + temp_dev + supplier + (1|factory)',...
'Distribution','Poisson','Link','log','FitMethod','Laplace',...
'DummyVarCoding','effects')
```

```
glme =
```

```
Generalized linear mixed-effects model fit by ML
```

```
Model information:
```

Number of observations	100
Fixed effects coefficients	6
Random effects coefficients	20
Covariance parameters	1
Distribution	Poisson
Link	Log
FitMethod	Laplace

```
Formula:
```

```
defects ~ 1 + newprocess + time_dev + temp_dev + supplier + (1 | factory)
```

```
Model fit statistics:
```

AIC	BIC	LogLikelihood	Deviance
416.35	434.58	-201.17	402.35

```
Fixed effects coefficients (95% CIs):
```

Name	Estimate	SE	tStat	DF	pValue
'(Intercept)'	1.4689	0.15988	9.1875	94	9.8194e-15
'newprocess'	-0.36766	0.17755	-2.0708	94	0.041122
'time_dev'	-0.094521	0.82849	-0.11409	94	0.90941
'temp_dev'	-0.28317	0.9617	-0.29444	94	0.76907
'supplier_C'	-0.071868	0.078024	-0.9211	94	0.35936
'supplier_B'	0.071072	0.07739	0.91836	94	0.36078

Lower	Upper
1.1515	1.7864
-0.72019	-0.015134
-1.7395	1.5505
-2.1926	1.6263
-0.22679	0.083051
-0.082588	0.22473

```
Random effects covariance parameters:
```

```
Group: factory (20 Levels)
```

Name1	Name2	Type	Estimate
'(Intercept)'	'(Intercept)'	'std'	0.31381

```
Group: Error
```

Name	Estimate
'sqrt(Dispersion)'	1

The `Model` information table displays the total number of observations in the sample data (100), the number of fixed- and random-effects coefficients (6 and 20, respectively), and the number of covariance parameters (1). It also indicates that the response variable has a Poisson distribution, the link function is Log, and the fit method is Laplace.

`Formula` indicates the model specification using Wilkinson's notation.

The `Model fit statistics` table displays statistics used to assess the goodness of fit of the model. This includes the Akaike information criterion (AIC), Bayesian information criterion (BIC) values, log likelihood (`LogLikelihood`), and deviance (`Deviance`) values.

The `Fixed effects coefficients` table indicates that `fitglm` returned 95% confidence intervals. It contains one row for each fixed-effects predictor, and each column contains statistics corresponding to that predictor. Column 1 (`Name`) contains the name of each fixed-effects coefficient, column 2 (`Estimate`) contains its estimated value, and column 3 (`SE`) contains the standard error of the coefficient. Column 4 (`tStat`) contains the *t*-statistic for a hypothesis test that the coefficient is equal to 0. Column 5 (`DF`) and column 6 (`pValue`) contain the degrees of freedom and *p*-value that correspond to the *t*-statistic, respectively. The last two columns (`Lower` and `Upper`) display the lower and upper limits, respectively, of the 95% confidence interval for each fixed-effects coefficient.

`Random effects covariance parameters` displays a table for each grouping variable (here, only `factory`), including its total number of levels (20), and the type and estimate of the covariance parameter. Here, `std` indicates that `fitglm` returns the standard deviation of the random effect associated with the `factory` predictor, which has an estimated value of 0.31381. It also displays a table containing the error parameter type (here, the square root of the dispersion parameter), and its estimated value of 1.

The standard display generated by `fitglm` does not provide confidence intervals for the random-effects parameters. To compute and display these values, use `covarianceParameters`.

### Check significance of random effect.

To determine whether the random-effects intercept grouped by `factory` is statistically significant, compute the confidence intervals for the estimated covariance parameter.

```
[psi,dispersion,stats] = covarianceParameters(glm);
```

`covarianceParameters` returns the estimated covariance parameter in `psi`, the estimated dispersion parameter `dispersion`, and a cell array of related statistics

stats. The first cell of stats contains statistics for factory, while the second cell contains statistics for the dispersion parameter.

Display the first cell of stats to see the confidence intervals for the estimated covariance parameter for factory.

```
stats{1}

ans =

    Covariance Type: Isotropic

    Group      Name1                Name2                Type
    factory    '(Intercept)'          '(Intercept)'        'std'

    Estimate   Lower      Upper
    0.31381    0.19253    0.51148
```

The columns Lower and Upper display the default 95% confidence interval for the estimated covariance parameter for factory. Because the interval [0.19253,0.51148] does not contain 0, the random-effects intercept is significant at the 5% significance level. Therefore, the random effect due to factory-specific variation must be considered before drawing any conclusions about the effectiveness of the new manufacturing process.

### Compare two models.

Compare the mixed-effects model that includes a random-effects intercept grouped by factory with a model that does not include the random effect, to determine which model is a better fit for the data. Fit the first model, FEglme, using only the fixed-effects predictors newprocess, time\_dev, temp\_dev, and supplier. Fit the second model, glme, using these same fixed-effects predictors, but also including a random-effects intercept grouped by factory.

```
FEglme = fitglme(mfr,...
    'defects ~ 1 + newprocess + time_dev + temp_dev + supplier',...
    'Distribution','Poisson','Link','log','FitMethod','Laplace');

glme = fitglme(mfr,...
    'defects ~ 1 + newprocess + time_dev + temp_dev + supplier + (1|factory)',...
    'Distribution','Poisson','Link','log','FitMethod','Laplace');
```

Compare the two models using a likelihood ratio test. Specify 'CheckNesting' as true, so compare returns a warning if the nesting requirements are not satisfied.

```
results = compare(FEglme, glme, 'CheckNesting', true)
```

```
results =
```

```
Theoretical Likelihood Ratio Test
```

Model	DF	AIC	BIC	LogLik	LRStat	deltaDF
FEglme	6	431.02	446.65	-209.51		
glme	7	416.35	434.58	-201.17	16.672	1

```
pValue
```

```
4.4435e-05
```

compare returns the degrees of freedom (DF), the Akaike information criterion (AIC), Bayesian information criterion (BIC), and log likelihood values for each model. glme has smaller AIC, BIC, and log likelihood values than FEglme, which indicates that glme (the model containing the random-effects term for intercept grouped by factory) is the better-fitting model for this data. Additionally, the small  $p$ -value indicates that compare rejects the null hypothesis that the response vector was generated by the fixed-effects-only model FEglme, in favor of the alternative that the response vector was generated by the mixed-effects model glme.

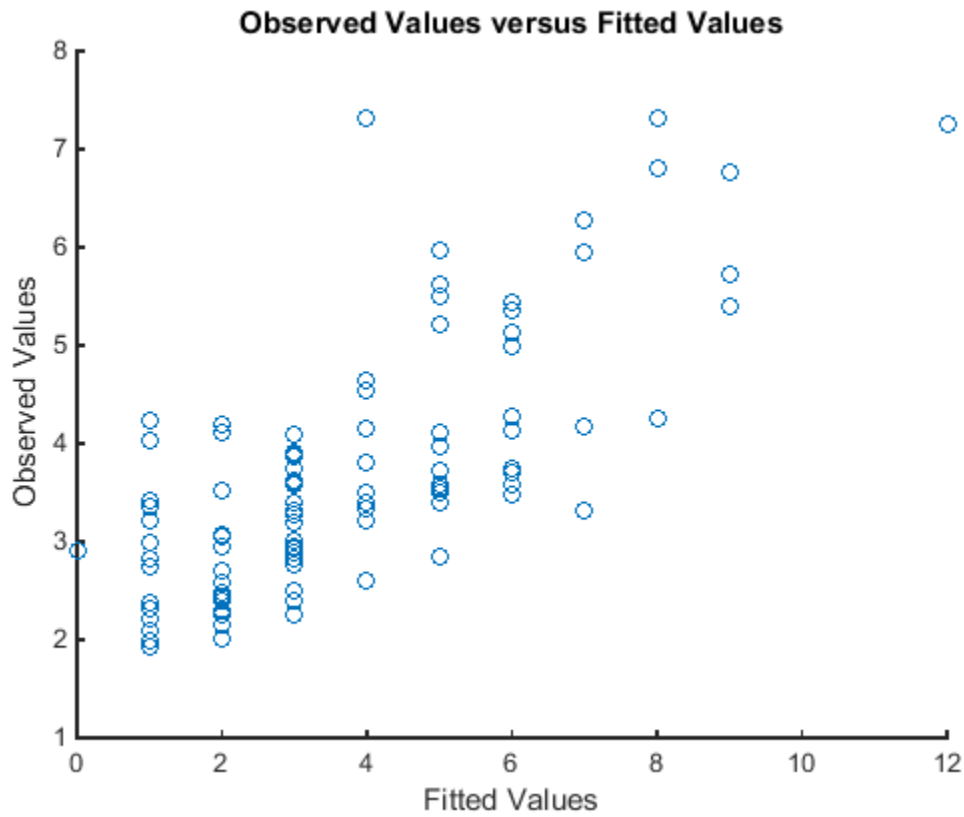
### Plot the results.

Generate the fitted conditional mean values for the model.

```
mufit = fitted(glme);
```

Plot the observed response values versus the fitted response values.

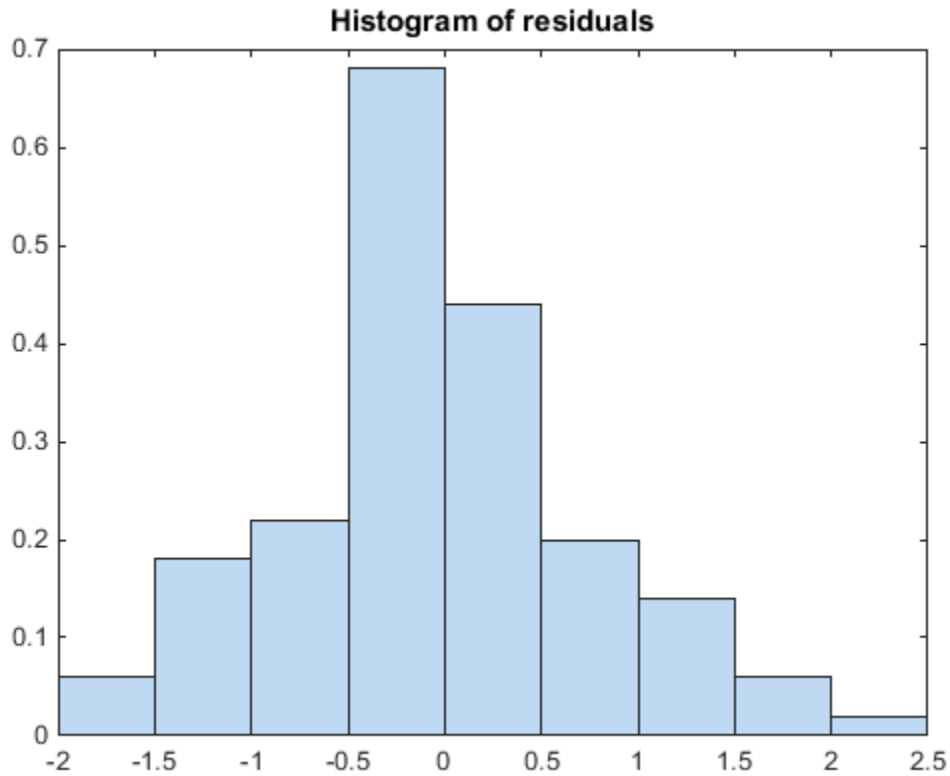
```
figure
scatter(mfr.defects, mufit)
title('Observed Values versus Fitted Values')
xlabel('Fitted Values')
ylabel('Observed Values')
```



Create diagnostic plots using conditional Pearson residuals to test model assumptions. Since raw residuals for generalized linear mixed-effects models do not have a constant variance across observations, use the conditional Pearson residuals instead.

Plot a histogram to visually confirm that the mean of the Pearson residuals is equal to 0. If the model is correct, we expect the Pearson residuals to be centered at 0.

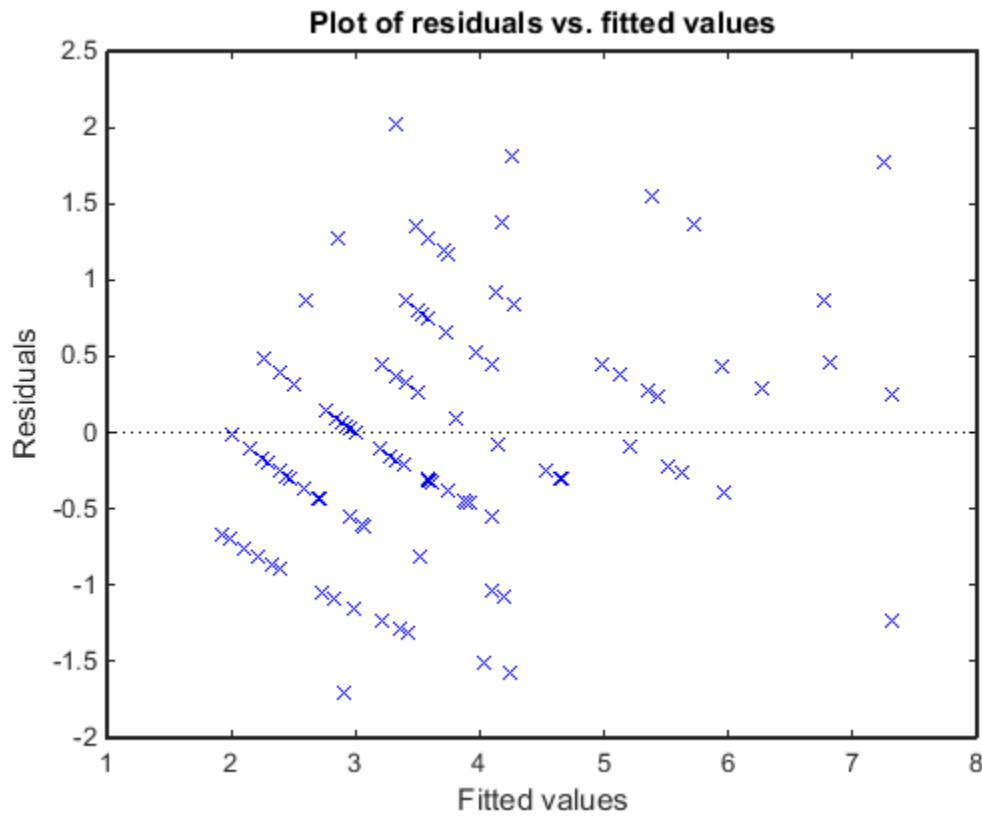
```
plotResiduals(glme, 'histogram', 'ResidualType', 'Pearson')
```



The histogram shows that the Pearson residuals are centered at 0.

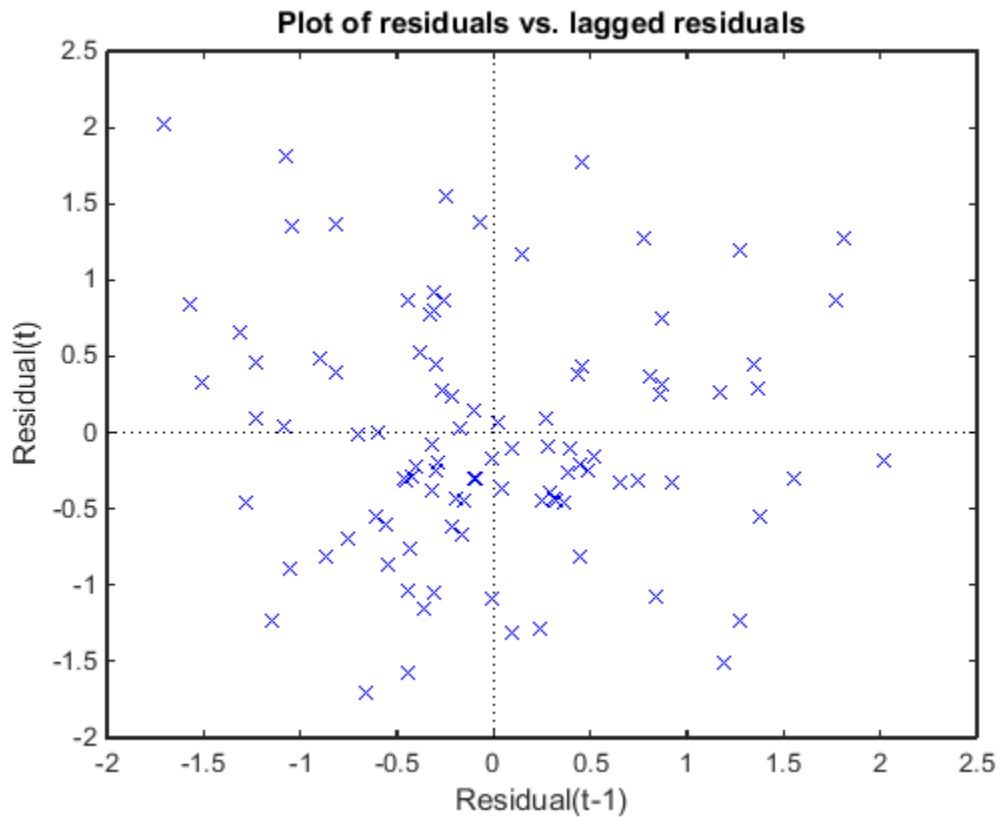
Plot the Pearson residuals versus the fitted values, to check for signs of nonconstant variance among the residuals (heteroscedasticity). We expect the conditional Pearson residuals to have a constant variance. Therefore, a plot of conditional Pearson residuals versus conditional fitted values should not reveal any systematic dependence on the conditional fitted values.

```
plotResiduals(glme, 'fitted', 'ResidualType', 'Pearson')
```



The plot does not show a systematic dependence on the fitted values, so there are no signs of nonconstant variance among the residuals.

Plot the Pearson residuals versus lagged residuals, to check for correlation among the residuals. The conditional independence assumption in GLME implies that the conditional Pearson residuals are approximately uncorrelated.



There is no pattern to the plot, so there are no signs of correlation among the residuals.

## See Also

`GeneralizedLinearMixedModel` | `fitglm`

## More About

- “Generalized Linear Models” on page 12-12



# Nonlinear Regression

---

- “Nonlinear Regression” on page 13-2
- “Nonlinear Regression Workflow” on page 13-15
- “Mixed-Effects Models” on page 13-22
- “Examining Residuals for Model Verification” on page 13-37
- “Mixed-Effects Models Using nlmeFit and nlmeFitsa” on page 13-43

## Nonlinear Regression

### In this section...

“What Are Parametric Nonlinear Regression Models?” on page 13-2

“Prepare Data” on page 13-3

“Represent the Nonlinear Model” on page 13-4

“Choose Initial Vector  $\beta_0$ ” on page 13-6

“Fit Nonlinear Model to Data” on page 13-7

“Examine Quality and Adjust the Fitted Nonlinear Model” on page 13-7

“Predict or Simulate Responses Using a Nonlinear Model” on page 13-10

### What Are Parametric Nonlinear Regression Models?

Parametric nonlinear models represent the relationship between a continuous response variable and one or more continuous predictor variables in the form

$$y = f(X, \beta) + \varepsilon,$$

where

- $y$  is an  $n$ -by-1 vector of observations of the response variable.
- $f$  is any function of  $X$  and  $\beta$  that evaluates each row of  $X$  along with the vector  $\beta$  to compute the prediction for the corresponding row of  $y$ .
- $X$  is an  $n$ -by- $p$  matrix of predictors, with one row for each observation, and one column for each predictor.
- $\beta$  is a  $p$ -by-1 vector of unknown parameters to be estimated.
- $\varepsilon$  is an  $n$ -by-1 vector of independent, identically distributed random disturbances.

In contrast, nonparametric models do not attempt to characterize the relationship between predictors and response with model parameters. Descriptions are often graphical, as in the case of “Decision Trees” on page 19-2.

`fitnlm` attempts to find values of the parameters  $\beta$  that minimize the mean squared differences between the observed responses  $y$  and the predictions of the model  $f(X, \beta)$ . To do so, it needs a starting value `beta0` before iteratively modifying the vector  $\beta$  to a vector with minimal mean squared error.

## Prepare Data

To begin fitting a regression, put your data into a form that fitting functions expect. All regression techniques begin with input data in an array  $X$  and response data in a separate vector  $y$ , or input data in a table or dataset array `tbl` and response data as a column in `tbl`. Each row of the input data represents one observation. Each column represents one predictor (variable).

For a table or dataset array `tbl`, indicate the response variable with the 'ResponseVar' name-value pair:

```
mdl = fitlm(tbl, 'ResponseVar', 'BloodPressure');
```

The response variable is the last column by default.

You cannot use numeric categorical predictors for nonlinear regression. A categorical predictor is one that takes values from a fixed set of possibilities.

Represent missing data as NaN for both input data and response data.

### Dataset Array for Input and Response Data

For example, to create a dataset array from an Excel spreadsheet:

```
ds = dataset('XLSFile', 'hospital.xls', ...
    'ReadObsNames', true);
```

To create a dataset array from workspace variables:

```
load carsmall
ds = dataset(Weight, Model_Year, MPG);
```

### Table for Input and Response Data

To create a table from an Excel spreadsheet:

```
tbl = readtable('hospital.xls', ...
    'ReadRowNames', true);
```

To create a table from workspace variables:

```
load carsmall
tbl = table(Weight, Model_Year, MPG);
```

### Numeric Matrix for Input Data and Numeric Vector for Response

For example, to create numeric arrays from workspace variables:

```
load carsmall
X = [Weight Horsepower Cylinders Model_Year];
y = MPG;
```

To create numeric arrays from an Excel spreadsheet:

```
[X, Xnames] = xlsread('hospital.xls');
y = X(:,4); % response y is systolic pressure
X(:,4) = []; % remove y from the X matrix
```

Notice that the nonnumeric entries, such as `sex`, do not appear in `X`.

### Represent the Nonlinear Model

There are several ways to represent a nonlinear model. Use whichever is most convenient.

The nonlinear model is a required input to `fitnlm`, in the `modelfun` input.

`fitnlm` assumes that the response function  $f(X,\beta)$  is smooth in the parameters  $\beta$ . If your function is not smooth, `fitnlm` can fail to provide optimal parameter estimates.

- “Function Handle to Anonymous Function or Function File” on page 13-4
- “Text Representation of Formula” on page 13-5

#### Function Handle to Anonymous Function or Function File

The function handle `@modelfun(b,x)` accepts a vector `b` and matrix, table, or dataset array `x`. The function handle should return a vector `f` with the same number of rows as `x`. For example, the function file `hougen.m` computes

$$\text{hougen}(b,x) = \frac{b(1)x(2) - x(3) / b(5)}{1 + b(2)x(1) + b(3)x(2) + b(4)x(3)}.$$

Examine the function by entering `type hougen` at the MATLAB command line.

```
function yhat = hougen(beta,x)
%HOUGEN Hougen-Watson model for reaction kinetics.
```

```

%   YHAT = HOUGEN(BETA,X) gives the predicted values of the
%   reaction rate, YHAT, as a function of the vector of
%   parameters, BETA, and the matrix of data, X.
%   BETA must have 5 elements and X must have three
%   columns.
%
%   The model form is:
%    $y = (b1*x2 - x3/b5) ./ (1+b2*x1+b3*x2+b4*x3)$ 
%
%   Reference:
%   [1] Bates, Douglas, and Watts, Donald, "Nonlinear
%   Regression Analysis and Its Applications", Wiley
%   1988 p. 271-272.

%   Copyright 1993-2004 The MathWorks, Inc.
%   B.A. Jones 1-06-95.

b1 = beta(1);
b2 = beta(2);
b3 = beta(3);
b4 = beta(4);
b5 = beta(5);

x1 = x(:,1);
x2 = x(:,2);
x3 = x(:,3);

yhat = (b1*x2 - x3/b5) ./ (1+b2*x1+b3*x2+b4*x3);

```

You can write an anonymous function that performs the same calculation as `hougen.m`.

```

modelfun = @(b,x) (b(1)*x(:,2) - x(:,3)/b(5)) ./ ...
(1 + b(2)*x(:,1) + b(3)*x(:,2) + b(4)*x(:,3));

```

### Text Representation of Formula

For data in a matrix  $X$  and response in a vector  $y$ :

- Represent the formula using 'x1' as the first predictor (column) in  $X$ , 'x2' as the second predictor, etc.
- Represent the vector of parameters to optimize as 'b1', 'b2', etc.
- Write the formula as ' $y \sim$  (mathematical expressions)'.

For example, to represent the response to the reaction data:

```
modelfun = 'y ~ (b1*x2 - x3/b5)/(1 + b2*x1 + b3*x2 + b4*x3)';
```

For data in a table or dataset array, you can use formulas represented as the variable names from the table or dataset array. Put the response variable name at the left of the formula, followed by a ~, followed by a character vector representing the response formula.

This example shows how to create a character vector to represent the response to the reaction data that is in a dataset array.

- 1 Load the reaction data.

```
load reaction
```

- 2 Put the data into a dataset array, where each variable has a name given in `xn` or `yn`.

```
ds = dataset({reactants,xn(1,:),xn(2,:),xn(3,:)},...
            {rate,yn});
```

- 3 Examine the first row of the dataset array.

```
ds(1,:)
```

```
ans =
```

Hydrogen	n_Pentane	Isopentane	ReactionRate
470	300	10	8.55

- 4 Write the hougen formula using names in the dataset array.

```
modelfun = ['ReactionRate ~ (b1*n_Pentane - Isopentane/b5) /'...
            '(1 + Hydrogen*b2 + n_Pentane*b3 + Isopentane*b4)']
```

```
modelfun =
```

```
ReactionRate ~ (b1*n_Pentane - Isopentane/b5) / ...
            (1 + Hydrogen*b2 + n_Pentane*b3 + Isopentane*b4)
```

## Choose Initial Vector beta0

The initial vector for the fitting iterations, `beta0`, can greatly influence the quality of the resulting fitted model. `beta0` gives the dimensionality of the problem, meaning it needs the correct length. A good choice of `beta0` leads to a quick, reliable model, while a poor choice can lead to a long computation, or to an inadequate model.

It is difficult to give advice on choosing a good `beta0`. If you believe certain components of the vector should be positive or negative, set your `beta0` to have those characteristics.

If you know the approximate value of other components, include them in `beta0`. However, if you don't know good values, try a random vector, such as

```
beta0 = randn(nVars,1);  
% or  
beta0 = 10*rand(nVars,1);
```

## Fit Nonlinear Model to Data

The syntax for fitting a nonlinear regression model using a table or dataset array `tbl` is

```
mdl = fitnlm(tbl,modelfun,beta0)
```

The syntax for fitting a nonlinear regression model using a numeric array `X` and numeric response vector `y` is

```
mdl = fitnlm(X,y,modelfun,beta0)
```

For information on representing the input parameters, see “Prepare Data” on page 13-3, “Represent the Nonlinear Model” on page 13-4, and “Choose Initial Vector `beta0`” on page 13-6.

`fitnlm` assumes that the response variable in a table or dataset array `tbl` is the last column. To change this, use the `ResponseVar` name-value pair to name the response column.

## Examine Quality and Adjust the Fitted Nonlinear Model

There are diagnostic plots to help you examine the quality of a model.

`plotDiagnostics(mdl)` gives a variety of plots, including leverage and Cook's distance plots. `plotResiduals(mdl)` gives the difference between the fitted model and the data.

There are also properties of `mdl` that relate to the model quality. `mdl.RMSE` gives the root mean square error between the data and the fitted model. `mdl.Residuals.Raw` gives the raw residuals. `mdl.Diagnostics` contains several fields, such as `Leverage` and `CooksDistance`, that can help you identify particularly interesting observations.

This example shows how to examine a fitted nonlinear model using diagnostic, residual, and slice plots.

Load the sample data.

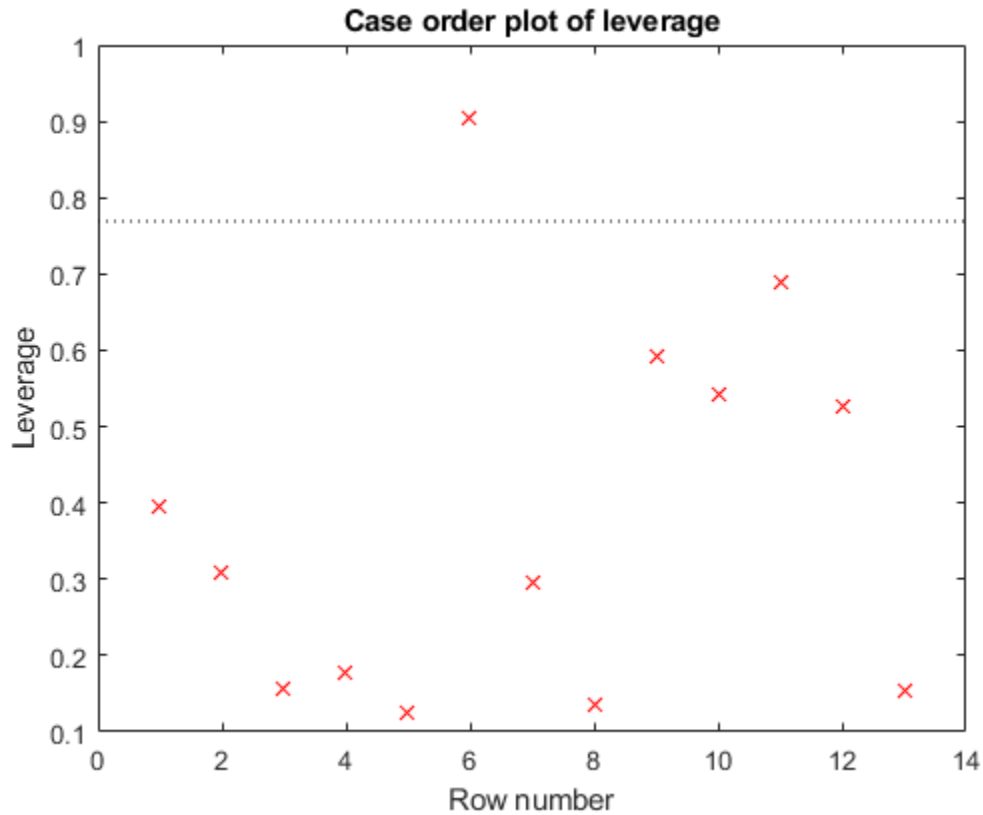
```
load reaction
```

Create a nonlinear model of rate as a function of reactants using the `hougen.m` function.

```
beta0 = ones(5,1);  
mdl = fitnlm(reactants,...  
    rate,@hougen,beta0);
```

Make a leverage plot of the data and model.

```
plotDiagnostics(mdl)
```



There is one point that has high leverage. Locate the point.



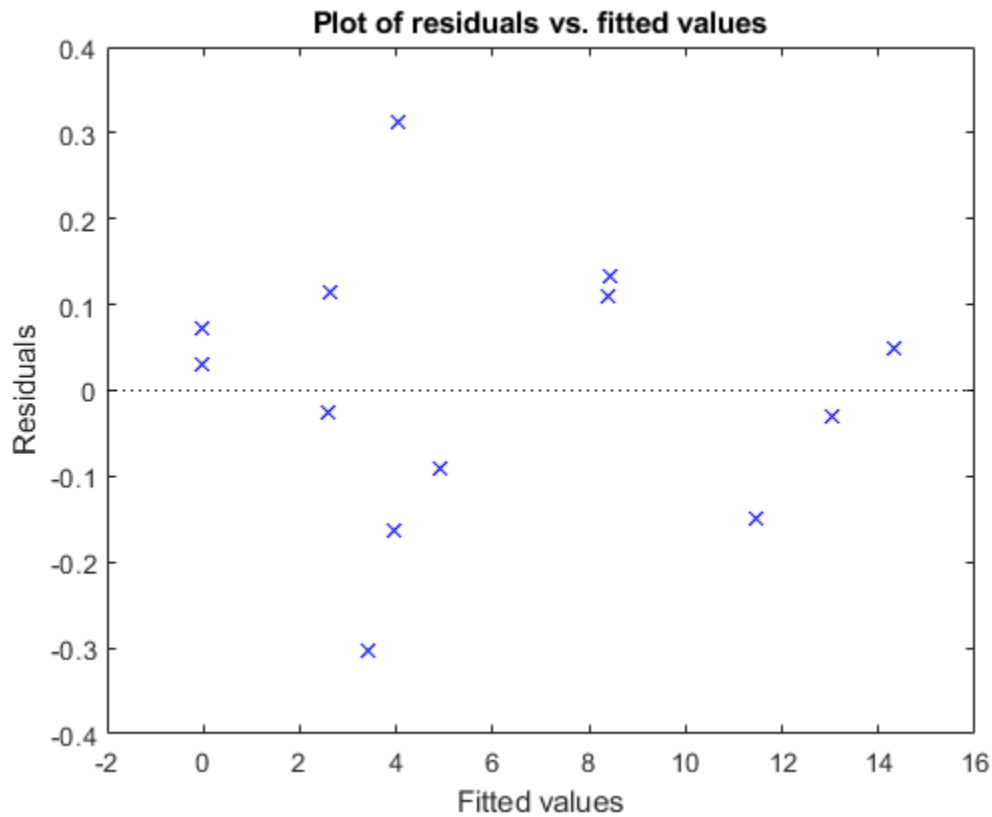
```
[~,max1] = max mdl.Diagnostics.Leverage)
```

```
max1 =
```

```
6
```

Examine a residuals plot.

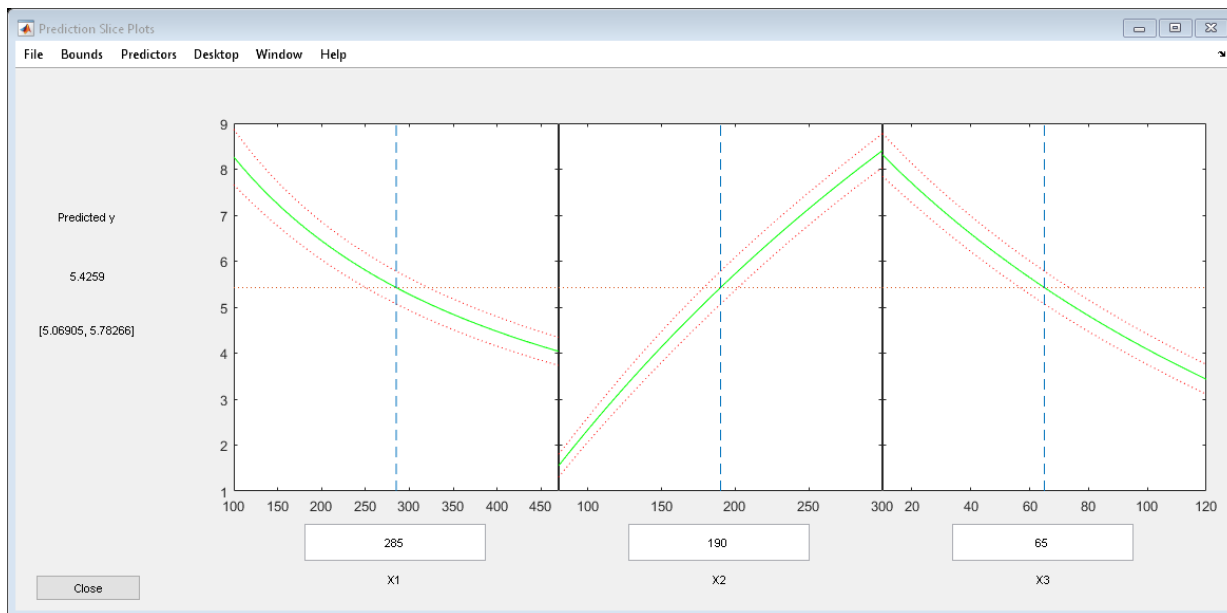
```
plotResiduals(mdl, 'fitted')
```



Nothing stands out as an outlier.

Use a slice plot to show the effect of each predictor on the model.

```
plotSlice mdl
```



You can drag the vertical dashed blue lines to see the effect of a change in one predictor on the response. For example, drag the X2 line to the right, and notice that the slope of the X3 line changes.

## Predict or Simulate Responses Using a Nonlinear Model

This example shows how to use the methods `predict`, `feval`, and `random` to predict and simulate responses to new data.

Randomly generate a sample from a Cauchy distribution.

```
rng('default')
X = rand(100,1);
X = tan(pi*X - pi/2);
```

Generate the response according to the model  $y = b1 * (\pi / 2 + \text{atan}((x - b2) / b3))$  and add noise to the response.

```

modelfun = @(b,x) b(1) * ...
    (pi/2 + atan((x - b(2))/b(3)));
y = modelfun([12 5 10],X) + randn(100,1);

```

Fit a model starting from the arbitrary parameters  $b = [1,1,1]$ .

```

beta0 = [1 1 1]; % An arbitrary guess
mdl = fitnlm(X,y,modelfun,beta0)

```

```
mdl =
```

```

Nonlinear regression model:
y ~ b1*(pi/2 + atan((x - b2)/b3))

```

```
Estimated Coefficients:
```

	Estimate	SE	tStat	pValue
b1	12.082	0.80028	15.097	3.3151e-27
b2	5.0603	1.0825	4.6747	9.5063e-06
b3	9.64	0.46499	20.732	2.0382e-37

```

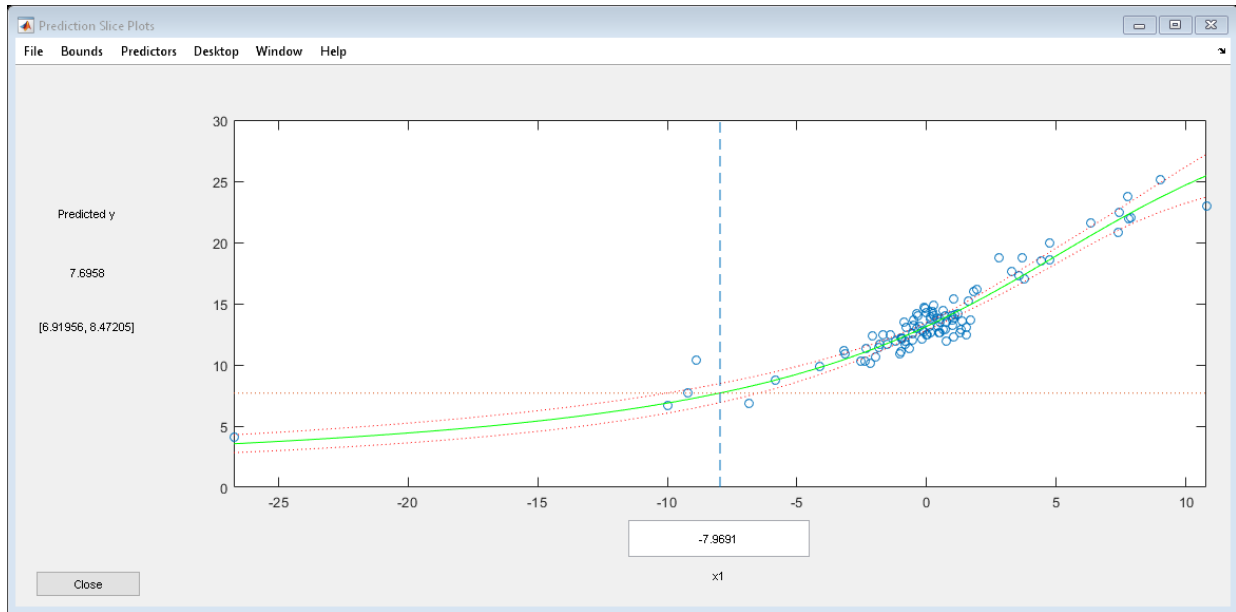
Number of observations: 100, Error degrees of freedom: 97
Root Mean Squared Error: 1.02
R-Squared: 0.92, Adjusted R-Squared 0.918
F-statistic vs. zero model: 6.45e+03, p-value = 1.72e-111

```

The fitted values are within a few percent of the parameters [12,5,10].

Examine the fit.

```
plotSlice(mdl)
```



### predict

The `predict` method predicts the mean responses and, if requested, gives confidence bounds. Find the predicted response values and predicted confidence intervals about the response at  $X$  values  $[-15;5;12]$ .

```
Xnew = [-15;5;12];
[ynew,ynewci] = predict mdl,Xnew
```

```
ynew =
```

```
5.4122
18.9022
26.5161
```

```
ynewci =
```

```
4.8233    6.0010
18.4555   19.3490
```

```
25.0170 28.0151
```

The confidence intervals are reflected in the slice plot.

### **feval**

The `feval` method predicts the mean responses. `feval` is often more convenient to use than `predict` when you construct a model from a dataset array.

Create the nonlinear model from a dataset array.

```
ds = dataset({X, 'X'}, {y, 'y'});
mdl2 = fitnlm(ds,modelfun,beta0);
```

Find the predicted model responses (CDF) at  $X$  values [-15;5;12].

```
Xnew = [-15;5;12];
ynew = feval(mdl2,Xnew)
```

```
ynew =
    5.4122
   18.9022
   26.5161
```

### **random**

The `random` method simulates new random response values, equal to the mean prediction plus a random disturbance with the same variance as the training data.

```
Xnew = [-15;5;12];
ysim = random(mdl,Xnew)
```

```
ysim =
    6.0505
   19.0893
   25.4647
```

Rerun the random method. The results change.

```
ysim = random mdl, Xnew)
```

```
ysim =
```

```
    6.3813  
   19.2157  
   26.6541
```

## Nonlinear Regression Workflow

This example shows how to do a typical nonlinear regression workflow: import data, fit a nonlinear regression, test its quality, modify it to improve the quality, and make predictions based on the model.

### Step 1. Prepare the data.

Load the reaction data.

```
load reaction
```

Examine the data in the workspace. `reactants` is a matrix with 13 rows and 3 columns. Each row corresponds to one observation, and each column corresponds to one variable. The variable names are in `xn`:

```
xn
```

```
xn =
```

```
3x10 char array
```

```
 'Hydrogen '
 'n-Pentane '
 'Isopentane'
```

Similarly, `rate` is a vector of 13 responses, with the variable name in `yn`:

```
yn
```

```
yn =
```

```
 'Reaction Rate'
```

The `hougen.m` file contains a nonlinear model of reaction rate as a function of the three predictor variables. For a 5-D vector  $b$  and 3-D vector  $x$ ,

$$hougen(b, x) = \frac{b(1)x(2) - x(3)/b(5)}{1 + b(2)x(1) + b(3)x(2) + b(4)x(3)}$$

As a start point for the solution, take `b` as a vector of ones.

```
beta0 = ones(5,1);
```

### Step 2. Fit a nonlinear model to the data.

```
mdl = fitnlm(reactants,...  
            rate,@hougen,beta0)
```

```
mdl =
```

```
Nonlinear regression model:  
y ~ hougen(b,X)
```

```
Estimated Coefficients:
```

	Estimate	SE	tStat	pValue
b1	1.2526	0.86702	1.4447	0.18654
b2	0.062776	0.043562	1.4411	0.18753
b3	0.040048	0.030885	1.2967	0.23089
b4	0.11242	0.075158	1.4957	0.17309
b5	1.1914	0.83671	1.4239	0.1923

```
Number of observations: 13, Error degrees of freedom: 8  
Root Mean Squared Error: 0.193  
R-Squared: 0.999, Adjusted R-Squared 0.998  
F-statistic vs. zero model: 3.91e+03, p-value = 2.54e-13
```

### Step 3. Examine the quality of the model.

The root mean squared error is fairly low compared to the range of observed values.

```
[mdl.RMSE min(rate) max(rate)]
```

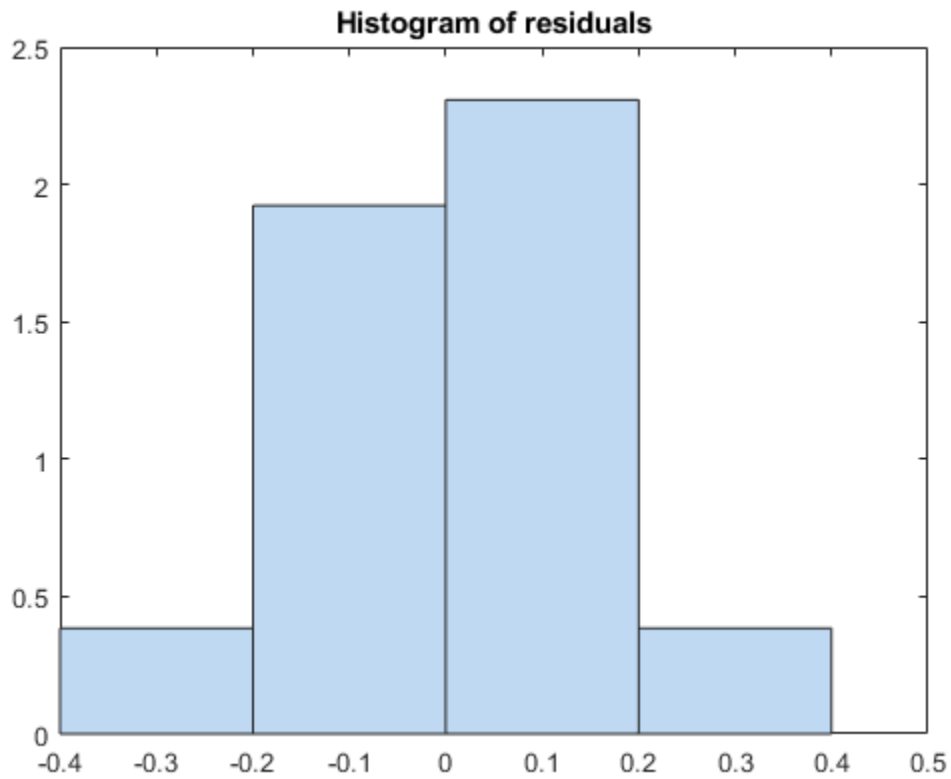
```
ans =
```

```
0.1933    0.0200    14.3900
```

Examine a residuals plot.



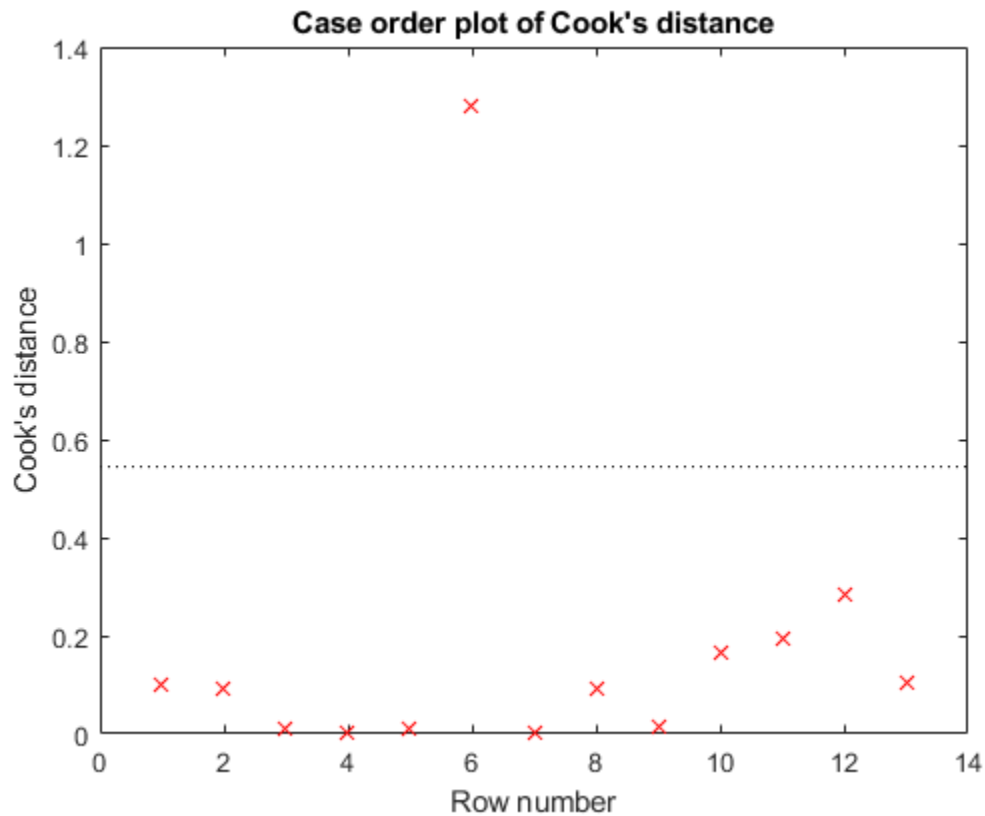
```
plotResiduals (mdl)
```



The model seems adequate for the data.

Examine a diagnostic plot to look for outliers.

```
plotDiagnostics (mdl, 'cookd')
```



Observation 6 seems out of line.

**Step 4. Remove the outlier.**

Remove the outlier from the fit using the `Exclude` name-value pair.

```
mdl1 = fitnlm(reactants,...  
    rate,@hougen,ones(5,1),'Exclude',6)
```

```
mdl1 =
```

```
Nonlinear regression model:
```

```
y ~ hougen(b,X)
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
b1	0.619	0.4552	1.3598	0.21605
b2	0.030377	0.023061	1.3172	0.22924
b3	0.018927	0.01574	1.2024	0.26828
b4	0.053411	0.041084	1.3	0.23476
b5	2.4125	1.7903	1.3475	0.2198

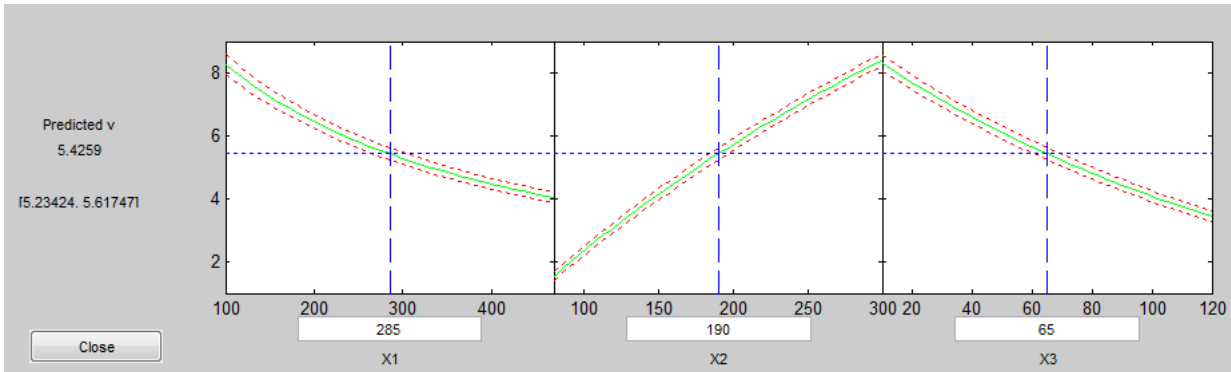
Number of observations: 12, Error degrees of freedom: 7  
 Root Mean Squared Error: 0.198  
 R-Squared: 0.999, Adjusted R-Squared 0.998  
 F-statistic vs. zero model: 2.67e+03, p-value = 2.54e-11

The model coefficients changed quite a bit from those in mdl.

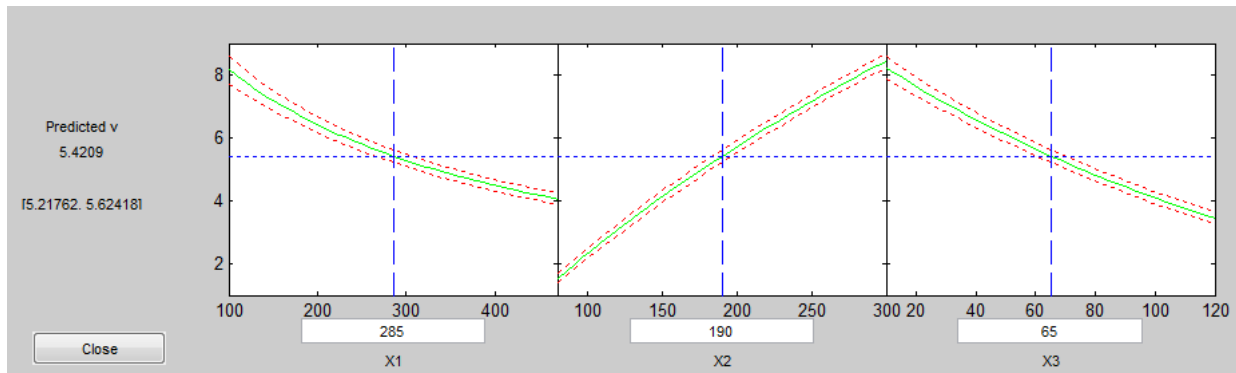
**Step 5. Examine slice plots of both models.**

To see the effect of each predictor on the response, make a slice plot using `plotSlice(mdl)`.

```
plotSlice(mdl)
```



```
plotSlice(mdl1)
```



The plots look very similar, with slightly wider confidence bounds for `mdl1`. This difference is understandable, since there is one less data point in the fit, representing over 7% fewer observations.

#### Step 6. Predict for new data.

Create some new data and predict the response from both models.

```
Xnew = [200,200,200;100,200,100;500,50,5];
[ypred yci] = predict(mdl,Xnew)
```

```
ypred =
```

```
1.8762
6.2793
1.6718
```

```
y ci =
```

```
1.6283 2.1242
5.9789 6.5797
1.5589 1.7846
```

```
[ypred1 yci1] = predict(mdl1,Xnew)
```

```
ypred1 =
```

```
1.8984  
6.2555  
1.6594
```

```
yci1 =
```

```
1.6260  2.1708  
5.9323  6.5787  
1.5345  1.7843
```

Even though the model coefficients are dissimilar, the predictions are nearly identical.

## Mixed-Effects Models

In this section...
“Introduction to Mixed-Effects Models” on page 13-22
“Mixed-Effects Model Hierarchy” on page 13-23
“Specifying Mixed-Effects Models” on page 13-24
“Specifying Covariate Models” on page 13-26
“Choosing nlmeFit or nlmeFitsa” on page 13-28
“Using Output Functions with Mixed-Effects Models” on page 13-31

### Introduction to Mixed-Effects Models

In statistics, an effect is anything that influences the value of a response variable at a particular setting of the predictor variables. Effects are translated into model parameters. In linear models, effects become coefficients, representing the proportional contributions of model terms. In nonlinear models, effects often have specific physical interpretations, and appear in more general nonlinear combinations.

Fixed effects represent population parameters, assumed to be the same each time data is collected. Estimating fixed effects is the traditional domain of regression modeling. Random effects, by comparison, are sample-dependent random variables. In modeling, random effects act like additional error terms, and their distributions and covariances must be specified.

For example, consider a model of the elimination of a drug from the bloodstream. The model uses time  $t$  as a predictor and the concentration of the drug  $C$  as the response. The nonlinear model term  $C_0 e^{-rt}$  combines parameters  $C_0$  and  $r$ , representing, respectively, an initial concentration and an elimination rate. If data is collected across multiple individuals, it is reasonable to assume that the elimination rate is a random variable  $r_i$  depending on individual  $i$ , varying around a population mean  $\bar{r}$ . The term  $C_0 e^{-rt}$  becomes  $C_0 e^{-(\bar{r} + (r_i - \bar{r}))t} = C_0 e^{-(\beta + b_i)t}$ ,

where  $\beta = \bar{r}$  is a fixed effect and  $b_i = r_i - \bar{r}$  is a random effect.

Random effects are useful when data falls into natural groups. In the drug elimination model, the groups are simply the individuals under study. More sophisticated models

might group data by an individual's age, weight, diet, etc. Although the groups are not the focus of the study, adding random effects to a model extends the reliability of inferences beyond the specific sample of individuals.

Mixed-effects models account for both fixed and random effects. As with all regression models, their purpose is to describe a response variable as a function of the predictor variables. Mixed-effects models, however, recognize correlations within sample subgroups. In this way, they provide a compromise between ignoring data groups entirely and fitting each group with a separate model.

## Mixed-Effects Model Hierarchy

Suppose data for a nonlinear regression model falls into one of  $m$  distinct groups  $i = 1, \dots, m$ . To account for the groups in a model, write response  $j$  in group  $i$  as:

$$y_{ij} = f(\varphi, x_{ij}) + \varepsilon_{ij}$$

$y_{ij}$  is the response,  $x_{ij}$  is a vector of predictors,  $\varphi$  is a vector of model parameters, and  $\varepsilon_{ij}$  is the measurement or process error. The index  $j$  ranges from 1 to  $n_i$ , where  $n_i$  is the number of observations in group  $i$ . The function  $f$  specifies the form of the model. Often,  $x_{ij}$  is simply an observation time  $t_{ij}$ . The errors are usually assumed to be independent and identically, normally distributed, with constant variance.

Estimates of the parameters in  $\varphi$  describe the population, assuming those estimates are the same for all groups. If, however, the estimates vary by group, the model becomes

$$y_{ij} = f(\varphi_i, x_{ij}) + \varepsilon_{ij}$$

In a mixed-effects model,  $\varphi_i$  may be a combination of a fixed and a random effect:

$$\varphi_i = \beta + b_i$$

The random effects  $b_i$  are usually described as multivariate normally distributed, with mean zero and covariance  $\Psi$ . Estimating the fixed effects  $\beta$  and the covariance of the random effects  $\Psi$  provides a description of the population that does not assume the parameters  $\varphi_i$  are the same across groups. Estimating the random effects  $b_i$  also gives a description of specific groups within the data.

Model parameters do not have to be identified with individual effects. In general, design matrices  $A$  and  $B$  are used to identify parameters with linear combinations of fixed and random effects:

$$\varphi_i = A\beta + Bb_i$$

If the design matrices differ among groups, the model becomes

$$\varphi_i = A_i\beta + B_ib_i$$

If the design matrices also differ among observations, the model becomes

$$\varphi_{ij} = A_{ij}\beta + B_{ij}b_i$$

$$y_{ij} = f(\varphi_{ij}, x_{ij}) + \varepsilon_{ij}$$

Some of the group-specific predictors in  $x_{ij}$  may not change with observation  $j$ . Calling those  $v_i$ , the model becomes

$$y_{ij} = f(\varphi_{ij}, x_{ij}, v_i) + \varepsilon_{ij}$$

## Specifying Mixed-Effects Models

Suppose data for a nonlinear regression model falls into one of  $m$  distinct groups  $i = 1, \dots, m$ . (Specifically, suppose that the groups are not nested.) To specify a general nonlinear mixed-effects model for this data:

- 1 Define group-specific model parameters  $\varphi_i$  as linear combinations of fixed effects  $\beta$  and random effects  $b_i$ .
- 2 Define response values  $y_i$  as a nonlinear function  $f$  of the parameters and group-specific predictor variables  $X_i$ .

The model is:

$$\varphi_i = A_i\beta + B_ib_i$$

$$y_i = f(\varphi_i, X_i) + \varepsilon_i$$

$$b_i \sim N(0, \Psi)$$

$$\varepsilon_i \sim N(0, \sigma^2)$$

This formulation of the nonlinear mixed-effects model uses the following notation:

$\varphi_i$             A vector of group-specific model parameters

$\beta$              A vector of fixed effects, modeling population parameters

$b_i$              A vector of multivariate normally distributed group-specific random effects



$A_i$	A group-specific design matrix for combining fixed effects
$B_i$	A group-specific design matrix for combining random effects
$X_i$	A data matrix of group-specific predictor values
$y_i$	A data vector of group-specific response values
$f$	A general, real-valued function of $\varphi_i$ and $X_i$
$\varepsilon_i$	A vector of group-specific errors, assumed to be independent, identically, normally distributed, and independent of $b_i$
$\Psi$	A covariance matrix for the random effects
$\sigma^2$	The error variance, assumed to be constant across observations

For example, consider a model of the elimination of a drug from the bloodstream. The model incorporates two overlapping phases:

- An initial phase  $p$  during which drug concentrations reach equilibrium with surrounding tissues
- A second phase  $q$  during which the drug is eliminated from the bloodstream

For data on multiple individuals  $i$ , the model is

$$y_{ij} = C_{pi}e^{-r_{pi}t_{ij}} + C_{qi}e^{-r_{qi}t_{ij}} + \varepsilon_{ij},$$

where  $y_{ij}$  is the observed concentration in individual  $i$  at time  $t_{ij}$ . The model allows for different sampling times and different numbers of observations for different individuals.

The elimination rates  $r_{pi}$  and  $r_{qi}$  must be positive to be physically meaningful. Enforce this by introducing the log rates  $R_{pi} = \log(r_{pi})$  and  $R_{qi} = \log(r_{qi})$  and reparametrizing the model:

$$y_{ij} = C_{pi}e^{-\exp(R_{pi})t_{ij}} + C_{qi}e^{-\exp(R_{qi})t_{ij}} + \varepsilon_{ij}$$

Choosing which parameters to model with random effects is an important consideration when building a mixed-effects model. One technique is to add random effects to all parameters, and use estimates of their variances to determine their significance in the model. An alternative is to fit the model separately to each group, without random effects, and look at the variation of the parameter estimates. If an estimate varies widely across groups, or if confidence intervals for each group have minimal overlap, the parameter is a good candidate for a random effect.

To introduce fixed effects  $\beta$  and random effects  $b_i$  for all model parameters, reexpress the model as follows:

$$\begin{aligned} y_{ij} &= [\bar{C}_p + (C_{pi} - \bar{C}_p)]e^{-\exp[\bar{R}_p + (R_{pi} - \bar{R}_p)]t_{ij}} + \\ &\quad [\bar{C}_q + (C_{qi} - \bar{C}_q)]e^{-\exp[\bar{R}_q + (R_{qi} - \bar{R}_q)]t_{ij}} + \varepsilon_{ij} \\ &= (\beta_1 + b_{1i})e^{-\exp(\beta_2 + b_{2i})t_{ij}} + \\ &\quad (\beta_3 + b_{3i})e^{-\exp(\beta_4 + b_{4i})t_{ij}} + \varepsilon_{ij} \end{aligned}$$

In the notation of the general model:

$$\beta = \begin{pmatrix} \beta_1 \\ \vdots \\ \beta_4 \end{pmatrix}, b_i = \begin{pmatrix} b_{i1} \\ \vdots \\ b_{i4} \end{pmatrix}, y_i = \begin{pmatrix} y_{i1} \\ \vdots \\ y_{in_i} \end{pmatrix}, X_i = \begin{pmatrix} t_{i1} \\ \vdots \\ t_{in_i} \end{pmatrix},$$

where  $n_i$  is the number of observations of individual  $i$ . In this case, the design matrices  $A_i$  and  $B_i$  are, at least initially, 4-by-4 identity matrices. Design matrices may be altered, as necessary, to introduce weighting of individual effects, or time dependency.

Fitting the model and estimating the covariance matrix  $\Psi$  often leads to further refinements. A relatively small estimate for the variance of a random effect suggests that it can be removed from the model. Likewise, relatively small estimates for covariances among certain random effects suggests that a full covariance matrix is unnecessary. Since random effects are unobserved,  $\Psi$  must be estimated indirectly. Specifying a diagonal or block-diagonal covariance pattern for  $\Psi$  can improve convergence and efficiency of the fitting algorithm.

Statistics and Machine Learning Toolbox functions `nlmefit` and `nlmefitsa` fit the general nonlinear mixed-effects model to data, estimating the fixed and random effects. The functions also estimate the covariance matrix  $\Psi$  for the random effects. Additional diagnostic outputs allow you to assess tradeoffs between the number of model parameters and the goodness of fit.

## Specifying Covariate Models

If the model in “Specifying Mixed-Effects Models” on page 13-24 assumes a group-dependent covariate such as weight ( $w$ ) the model becomes:

$$\begin{pmatrix} \varphi_1 \\ \varphi_2 \\ \varphi_3 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & w_i \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} \beta_1 \\ \beta_2 \\ \beta_3 \\ \beta_4 \end{pmatrix} + \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix}$$

Thus, the parameter  $\varphi_i$  for any individual in the  $i$ th group is:

$$\begin{pmatrix} \varphi_{1_i} \\ \varphi_{2_i} \\ \varphi_{3_i} \end{pmatrix} = \begin{pmatrix} \beta_1 + \beta_4 * w_i \\ \beta_2 \\ \beta_3 \end{pmatrix} + \begin{pmatrix} b_{1_i} \\ b_{2_i} \\ b_{3_i} \end{pmatrix}$$

To specify a covariate model, use the 'FEGroupDesign' option.

'FEGroupDesign' is a  $p$ -by- $q$ -by- $m$  array specifying a different  $p$ -by- $q$  fixed-effects design matrix for each of the  $m$  groups. Using the previous example, the array resembles the following:

$$\begin{bmatrix} \begin{bmatrix} 1 & 0 & 0 & W_1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} & \begin{bmatrix} 1 & 0 & 0 & W_2 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} & \begin{bmatrix} 1 & 0 & 0 & W_3 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} & \dots & \begin{bmatrix} 1 & 0 & 0 & W_m \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \end{bmatrix}$$

## 1 Create the array.

```
% Number of parameters in the model (Phi)
num_params = 3;
% Number of covariates
num_cov = 1;
% Assuming number of groups in the data set is 7
num_groups = 7;
% Array of covariate values
covariates = [75; 52; 66; 55; 70; 58; 62 ];
A = repmat(eye(num_params, num_params+num_cov), ...
```

```
[1,1,num_groups]);  
A(1,num_params+1,1:num_groups) = covariates(:,1)
```

- 2 Create a struct with the specified design matrix.

```
options.FEGroupDesign = A;
```

- 3 Specify the arguments for `nlmefit` (or `nlmefitsa`) as shown in “Mixed-Effects Models Using `nlmefit` and `nlmefitsa`” on page 13-43.

## Choosing `nlmefit` or `nlmefitsa`

Statistics and Machine Learning Toolbox provides two functions, `nlmefit` and `nlmefitsa` for fitting nonlinear mixed-effects models. Each function provides different capabilities, which may help you decide which to use.

- “Approximation Methods” on page 13-28
- “Parameters Specific to `nlmefitsa`” on page 13-29
- “Model and Data Requirements” on page 13-29

### Approximation Methods

`nlmefit` provides the following four approximation methods for fitting nonlinear mixed-effects models:

- 'LME' — Use the likelihood for the linear mixed-effects model at the current conditional estimates of  $\beta$  and  $B$ . This is the default.
- 'RELME' — Use the restricted likelihood for the linear mixed-effects model at the current conditional estimates of  $\beta$  and  $B$ .
- 'FO' — First-order Laplacian approximation without random effects.
- 'FOCE' — First-order Laplacian approximation at the conditional estimates of  $B$ .

`nlmefitsa` provides an additional approximation method, Stochastic Approximation Expectation-Maximization (SAEM) [24] with three steps :

- 1 Simulation: Generate simulated values of the random effects  $b$  from the posterior density  $p(b|\Sigma)$  given the current parameter estimates.
- 2 Stochastic approximation: Update the expected value of the log likelihood function by taking its value from the previous step, and moving part way toward the average value of the log likelihood calculated from the simulated random effects.

- 3 Maximization step: Choose new parameter estimates to maximize the log likelihood function given the simulated values of the random effects.

Both `nlmefit` and `nlmefitsa` attempt to find parameter estimates to maximize a likelihood function, which is difficult to compute. `nlmefit` deals with the problem by approximating the likelihood function in various ways, and maximizing the approximate function. It uses traditional optimization techniques that depend on things like convergence criteria and iteration limits.

`nlmefitsa`, on the other hand, simulates random values of the parameters in such a way that in the long run they converge to the values that maximize the exact likelihood function. The results are random, and traditional convergence tests don't apply. Therefore `nlmefitsa` provides options to plot the results as the simulation progresses, and to restart the simulation multiple times. You can use these features to judge whether the results have converged to the accuracy you desire.

### Parameters Specific to `nlmefitsa`

The following parameters are specific to `nlmefitsa`. Most control the stochastic algorithm.

- `Cov0` — Initial value for the covariance matrix `PSI`. Must be an  $r$ -by- $r$  positive definite matrix. If empty, the default value depends on the values of `BETA0`.
- `ComputeStdErrors` — `true` to compute standard errors for the coefficient estimates and store them in the output `STATS` structure, or `false` (default) to omit this computation.
- `LogLikMethod` — Specifies the method for approximating the log likelihood.
- `NBurnIn` — Number of initial burn-in iterations during which the parameter estimates are not recomputed. Default is 5.
- `NIterations` — Controls how many iterations are performed for each of three phases of the algorithm.
- `NMCMCIterations` — Number of Markov Chain Monte Carlo (MCMC) iterations.

### Model and Data Requirements

There are some differences in the capabilities of `nlmefit` and `nlmefitsa`. Therefore some data and models are usable with either function, but some may require you to choose just one of them.

- **Error models** — `nlmefitsa` supports a variety of error models. For example, the standard deviation of the response can be constant, proportional to the function value, or a combination of the two. `nlmefit` fits models under the assumption that the standard deviation of the response is constant. One of the error models, `'exponential'`, specifies that the log of the response has a constant standard deviation. You can fit such models using `nlmefit` by providing the log response as input, and by rewriting the model function to produce the log of the nonlinear function value.
- **Random effects** — Both functions fit data to a nonlinear function with parameters, and the parameters may be simple scalar values or linear functions of covariates. `nlmefit` allows any coefficients of the linear functions to have both fixed and random effects. `nlmefitsa` supports random effects only for the constant (intercept) coefficient of the linear functions, but not for slope coefficients. So in the example in “Specifying Covariate Models” on page 13-26, `nlmefitsa` can treat only the first three beta values as random effects.
- **Model form** — `nlmefit` supports a very general model specification, with few restrictions on the design matrices that relate the fixed coefficients and the random effects to the model parameters. `nlmefitsa` is more restrictive:
  - The fixed effect design must be constant in every group (for every individual), so an observation-dependent design is not supported.
  - The random effect design must be constant for the entire data set, so neither an observation-dependent design nor a group-dependent design is supported.
  - As mentioned under **Random Effects**, the random effect design must not specify random effects for slope coefficients. This implies that the design must consist of zeros and ones.
  - The random effect design must not use the same random effect for multiple coefficients, and cannot use more than one random effect for any single coefficient.
  - The fixed effect design must not use the same coefficient for multiple parameters. This implies that it can have at most one nonzero value in each column.

If you want to use `nlmefitsa` for data in which the covariate effects are random, include the covariates directly in the nonlinear model expression. Don't include the covariates in the fixed or random effect design matrices.

- **Convergence** — As described in the **Model form**, `nlmefit` and `nlmefitsa` have different approaches to measuring convergence. `nlmefit` uses traditional optimization measures, and `nlmefitsa` provides diagnostics to help you judge the convergence of a random simulation.

In practice, `nlmefitsa` tends to be more robust, and less likely to fail on difficult problems. However, `nlmefit` may converge faster on problems where it converges at all. Some problems may benefit from a combined strategy, for example by running `nlmefitsa` for a while to get reasonable parameter estimates, and using those as a starting point for additional iterations using `nlmefit`.

## Using Output Functions with Mixed-Effects Models

The `OutputFcn` field of the `options` structure specifies one or more functions that the solver calls after each iteration. Typically, you might use an output function to plot points at each iteration or to display optimization quantities from the algorithm. To set up an output function:

- 1 Write the output function as a MATLAB file function or local function.
- 2 Use `statset` to set the value of `OutputFcn` to be a function handle, that is, the name of the function preceded by the `@` sign. For example, if the output function is `outfun.m`, the command

```
options = statset('OutputFcn', @outfun);
```

specifies `OutputFcn` to be the handle to `outfun`. To specify multiple output functions, use the syntax:

```
options = statset('OutputFcn',{@outfun, @outfun2});
```

- 3 Call the optimization function with `options` as an input argument.

For an example of an output function, see “Sample Output Function” on page 13-35.

### Structure of the Output Function

The function definition line of the output function has the following form:

```
stop = outfun(beta, status, state)
```

where

- *beta* is the current fixed effects.
- *status* is a structure containing data from the current iteration. “Fields in *status*” on page 13-32 describes the structure in detail.
- *state* is the current state of the algorithm. “States of the Algorithm” on page 13-33 lists the possible values.

- *stop* is a flag that is `true` or `false` depending on whether the optimization routine should quit or continue. See “Stop Flag” on page 13-34 for more information.

The solver passes the values of the input arguments to `outfun` at each iteration.

### Fields in `status`

The following table lists the fields of the `status` structure:

Field	Description
<code>procedure</code>	<ul style="list-style-type: none"><li>• 'ALT' — alternating algorithm for the optimization of the linear mixed effects or restricted linear mixed effects approximations</li><li>• 'LAP' — optimization of the Laplacian approximation for first order or first order conditional estimation</li></ul>
<code>iteration</code>	An integer starting from 0.



Field	Description
inner	<p>A structure describing the status of the inner iterations within the ALT and LAP procedures, with the fields:</p> <ul style="list-style-type: none"> <li>• procedure — When procedure is 'ALT': <ul style="list-style-type: none"> <li>• 'PNLS' (penalized nonlinear least squares)</li> <li>• 'LME' (linear mixed-effects estimation)</li> <li>• 'none'</li> </ul> </li> </ul> <p>When procedure is 'LAP',</p> <ul style="list-style-type: none"> <li>• 'PNLS' (penalized nonlinear least squares)</li> <li>• 'PLM' (profiled likelihood maximization)</li> <li>• 'none'</li> </ul> <ul style="list-style-type: none"> <li>• state — one of the following: <ul style="list-style-type: none"> <li>• 'init'</li> <li>• 'iter'</li> <li>• 'done'</li> <li>• 'none'</li> </ul> </li> <li>• iteration — an integer starting from 0, or NaN. For nlme fitsa with burn-in iterations, the output function is called after each of those iterations with a negative value for STATUS.iteration.</li> </ul>
fval	The current log likelihood
Psi	The current random-effects covariance matrix
theta	The current parameterization of Psi
mse	The current error variance

### States of the Algorithm

The following table lists the possible values for state:

state	Description
'init'	The algorithm is in the initial state before the first iteration.

state	Description
'iter'	The algorithm is at the end of an iteration.
'done'	The algorithm is in the final state after the last iteration.

The following code illustrates how the output function might use the value of `state` to decide which tasks to perform at the current iteration:

```
switch state
    case 'iter'
        % Make updates to plot or guis as needed
    case 'init'
        % Setup for plots or guis
    case 'done'
        % Cleanup of plots, guis, or final plot
otherwise
end
```

### Stop Flag

The output argument `stop` is a flag that is `true` or `false`. The flag tells the solver whether it should quit or continue. The following examples show typical ways to use the `stop` flag.

#### Stopping an Optimization Based on Intermediate Results

The output function can stop the estimation at any iteration based on the values of arguments passed into it. For example, the following code sets `stop` to `true` based on the value of the log likelihood stored in the `'fval'` field of the status structure:

```
stop = outfun(beta,status,state)
stop = false;
% Check if loglikelihood is more than 132.
if status.fval > -132
    stop = true;
end
```

#### Stopping an Iteration Based on GUI Input

If you design a GUI to perform `nlmefit` iterations, you can make the output function stop when a user clicks a **Stop** button on the GUI. For example, the following code implements a dialog to cancel calculations:

```
function retval = stop_outfcn(beta,str,status)
persistent h stop;
```

```

if isequal(str.inner.state,'none')
    switch(status)
        case 'init'
            % Initialize dialog
            stop = false;
            h = msgbox('Press STOP to cancel calculations.',...
                'NLMEFIT: Iteration 0 ');
            button = findobj(h,'type','uicontrol');
            set(button,'String','STOP','Callback',@stopper)
            pos = get(h,'Position');
            pos(3) = 1.1 * pos(3);
            set(h,'Position',pos)
            drawnow
        case 'iter'
            % Display iteration number in the dialog title
            set(h,'Name',sprintf('NLMEFIT: Iteration %d',...
                str.iteration))
            drawnow;
        case 'done'
            % Delete dialog
            delete(h);
    end
end
if stop
    % Stop if the dialog button has been pressed
    delete(h)
end
retval = stop;

function stopper(varargin)
    % Set flag to stop when button is pressed
    stop = true;
    disp('Calculation stopped.')
end
end

```

### Sample Output Function

`nlmefitoutputfcn` is the sample Statistics and Machine Learning Toolbox output function for `nlmefit` and `nlmefitsa`. It initializes or updates a plot with the fixed-effects (BETA) and variance of the random effects (`diag(STATUS.Psi)`). For `nlmefit`, the plot also includes the log-likelihood (`STATUS.fval`).

`nlmefitoutputfcn` is the default output function for `nlmefitsa`. To use it with `nlmefit`, specify a function handle for it in the options structure:

```
opt = statset('OutputFcn', @nlmefitoutputfcn, ...)  
beta = nlmefit(..., 'Options', opt, ...)
```

To prevent `nlmefitsa` from using of this function, specify an empty value for the output function:

```
opt = statset('OutputFcn', [], ...)  
beta = nlmefitsa(..., 'Options', opt, ...)
```

`nlmefitoutputfcn` stops `nlmefit` or `nlmefitsa` if you close the figure that it produces.

## Examining Residuals for Model Verification

You can examine the `stats` structure, which is returned by both `nlmefit` and `nlmefitsa`, to determine the quality of your model. The `stats` structure contains fields with conditional weighted residuals (`cwres` field) and individual weighted residuals (`iwres` field). Since the model assumes that residuals are normally distributed, you can examine the residuals to see how well this assumption holds.

This example generates synthetic data using normal distributions. It shows how the fit statistics look:

- Good when testing against the same type of model as generates the data
- Poor when tested against incorrect data models

### 1 Initialize a 2-D model with 100 individuals:

```
nGroups = 100; % 100 Individuals
nlmefun = @(PHI,t) (PHI(:,1)*5 + PHI(:,2)^2.*t); % Regression fcn
REParamSelect = [1 2]; % Both Parameters have random effect
errorParam = .03;
beta0 = [ 1.5 5]; % Parameter means
psi = [ 0.35 0; ... % Covariance Matrix
       0 0.51 ];
time = [0.25;0.5;0.75;1;1.25;2;3;4;5;6];
nParameters = 2;
rng(0,'twister') % for reproducibility
```

### 2 Generate the data for fitting with a proportional error model:

```
b_i = mvnrnd(zeros(1, numel(REParamSelect)), psi, nGroups);
individualParameters = zeros(nGroups,nParameters);
individualParameters(:, REParamSelect) = ...
    bsxfun(@plus,beta0(REParamSelect), b_i);

groups = repmat(1:nGroups,numel(time),1);
groups = vertcat(groups(:));

y = zeros(numel(time)*nGroups,1);
x = zeros(numel(time)*nGroups,1);
for i = 1:nGroups
    idx = groups == i;
    f = nlmefun(individualParameters(i,:), time);
    % Make a proportional error model for y:
    y(idx) = f + errorParam*f.*randn(numel(f),1);
```

```
        x(idx) = time;
    end
```

```
P = [ 1 0 ; 0 1 ];
```

- 3 Fit the data using the same regression function and error model as the model generator:

```
[~,~,stats] = nlmefit(x,y,groups, ...
    [],nlmefun,[1 1], 'REParamsSelect',REParamSelect,...
    'ErrorModel', 'Proportional', 'CovPattern',P);
```

- 4 Create a plotting routine by copying the following function definition, and creating a file `plotResiduals.m` on your MATLAB path:

```
function plotResiduals(stats)
pwres = stats.pwres;
iwres = stats.iwres;
cwres = stats.cwres;
figure
subplot(2,3,1);
normplot(pwres); title('PWRES')
subplot(2,3,4);
createhistplot(pwres);

subplot(2,3,2);
normplot(cwres); title('CWRES')
subplot(2,3,5);
createhistplot(cwres);

subplot(2,3,3);
normplot(iwres); title('IWRES')
subplot(2,3,6);
createhistplot(iwres); title('IWRES')

function createhistplot(pwres)
h = histogram(pwres);

% x is the probability/height for each bin
x = h.Values/sum(h.Values*h.BinWidth)

% n is the center of each bin
n = h.BinEdges + (0.5*h.BinWidth)
n(end) = [];

bar(n,x);
```

```

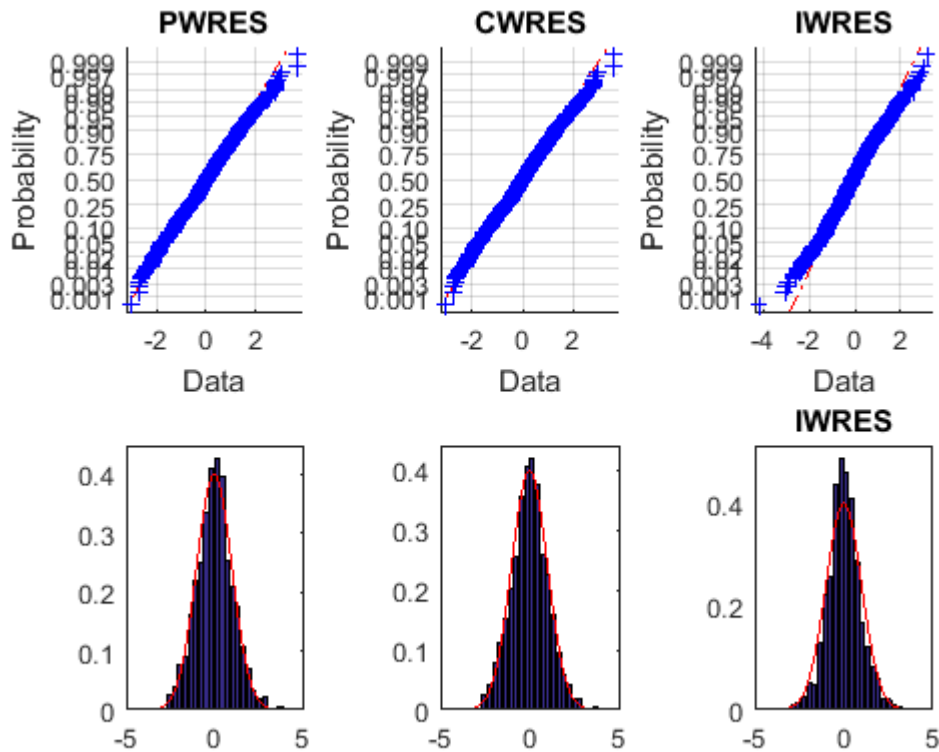
ylim([0 max(x)*1.05]);
hold on;
x2 = -4:0.1:4;
f2 = normpdf(x2,0,1);
plot(x2,f2,'r');
end

```

```
end
```

- 5 Plot the residuals using the `plotResiduals` function:

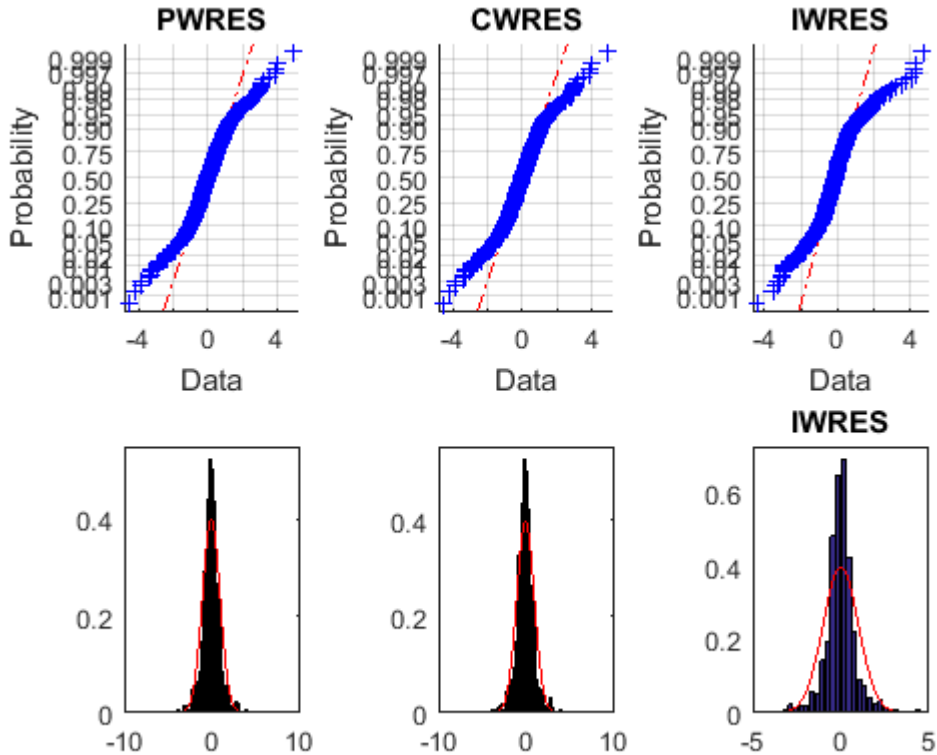
```
plotResiduals(stats);
```



The upper probability plots look straight, meaning the residuals are normally distributed. The bottom histogram plots match the superimposed normal density plot. So you can conclude that the error model matches the data.

- 6 For comparison, fit the data using a constant error model, instead of the proportional model that created the data:

```
[~,~,stats] = nlmefit(x,y,groups, ...
    [],nlmefun,[0 0], 'REParamsSelect',REParamSelect,...
    'ErrorModel','Constant','CovPattern',P);
plotResiduals(stats);
```

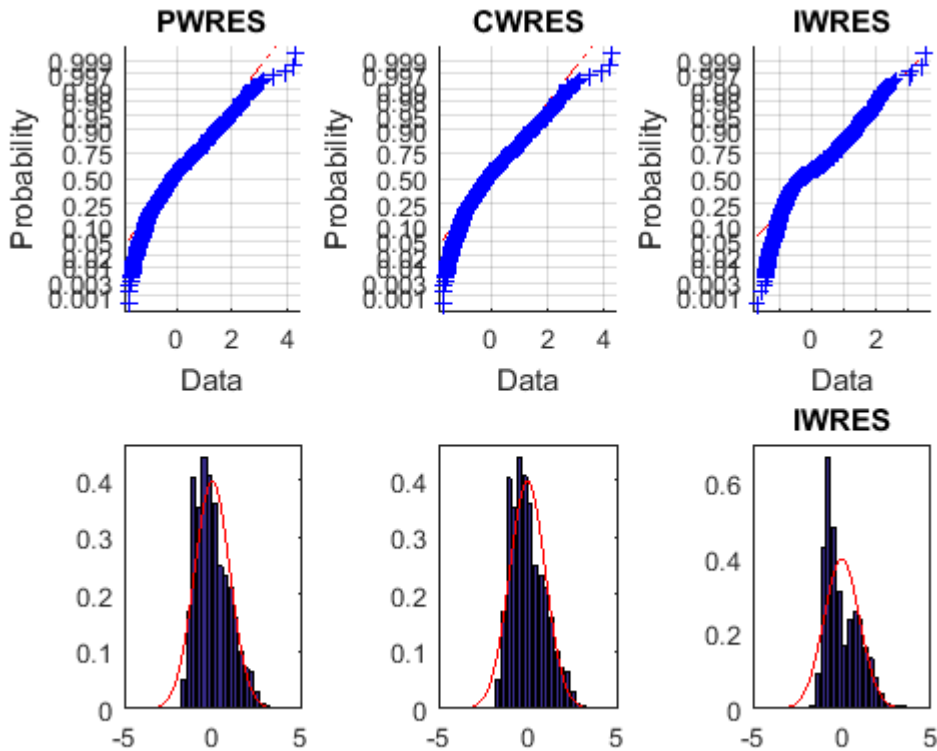




The upper probability plots are not straight, indicating the residuals are not normally distributed. The bottom histogram plots are fairly close to the superimposed normal density plots.

- For another comparison, fit the data to a different structural model than created the data:

```
nlmefun2 = @(PHI,t)(PHI(:,1)*5 + PHI(:,2).*t.^4);
[~,~,stats] = nlmeft(x,y,groups, ...
    [],nlmefun2,[0 0], 'REParamsSelect',REParamSelect,...
    'ErrorModel','constant', 'CovPattern',P);
plotResiduals(stats);
```



Not only are the upper probability plots not straight, but the histogram plot is quite skewed compared to the superimposed normal density. These residuals are not normally distributed, and do not match the model.

## Mixed-Effects Models Using nlmeFit and nlmeFitsa

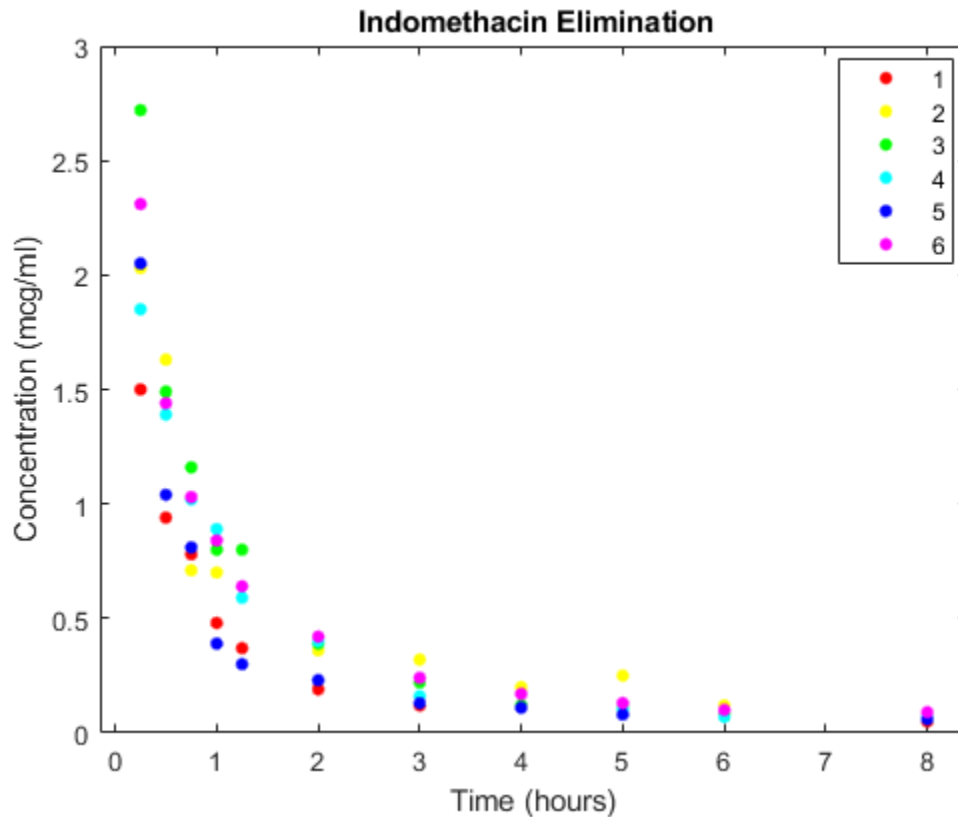
Load the sample data.

```
load indomethacin
```

The data in `indomethacin.mat` records concentrations of the drug indomethacin in the bloodstream of six subjects over eight hours.

Plot the scatter plot of indomethacin in the bloodstream grouped by subject.

```
gscatter(time, concentration, subject)
xlabel('Time (hours)')
ylabel('Concentration (mcg/ml)')
title('\bf Indomethacin Elimination')
hold on
```



Specifying Mixed-Effects Models page discusses a useful model for this type of data.

Construct the model via an anonymous function.

```
model = @(phi,t)(phi(1)*exp(-exp(phi(2))*t) + ...
                phi(3)*exp(-exp(phi(4))*t));
```

Use the `nlinfit` function to fit the model to all of the data, ignoring subject-specific effects.

```
phi0 = [1 2 1 1];
[phi,res] = nlinfit(time,concentration,model,phi0);
```

Compute the mean squared error.

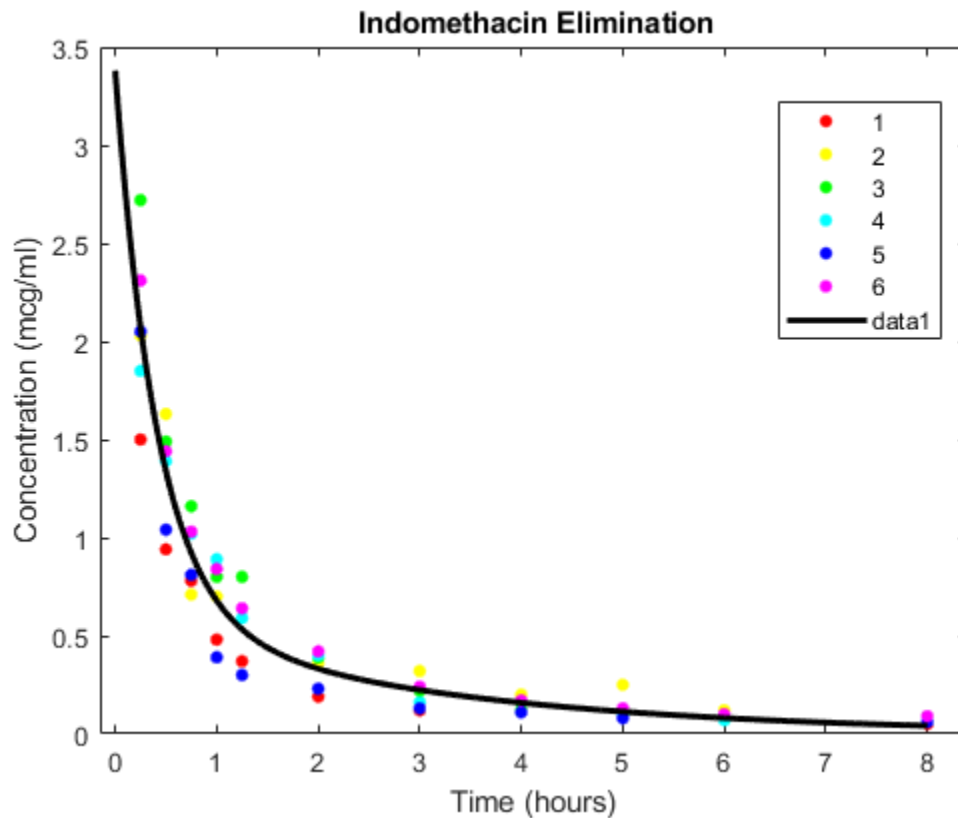
```
numObs = length(time);  
numParams = 4;  
df = numObs-numParams;  
mse = (res'*res)/df
```

```
mse =
```

```
    0.0304
```

Super impose the model on the scatter plot of data.

```
tplot = 0:0.01:8;  
plot(tplot,model(phi,tplot),'k','LineWidth',2)  
hold off
```

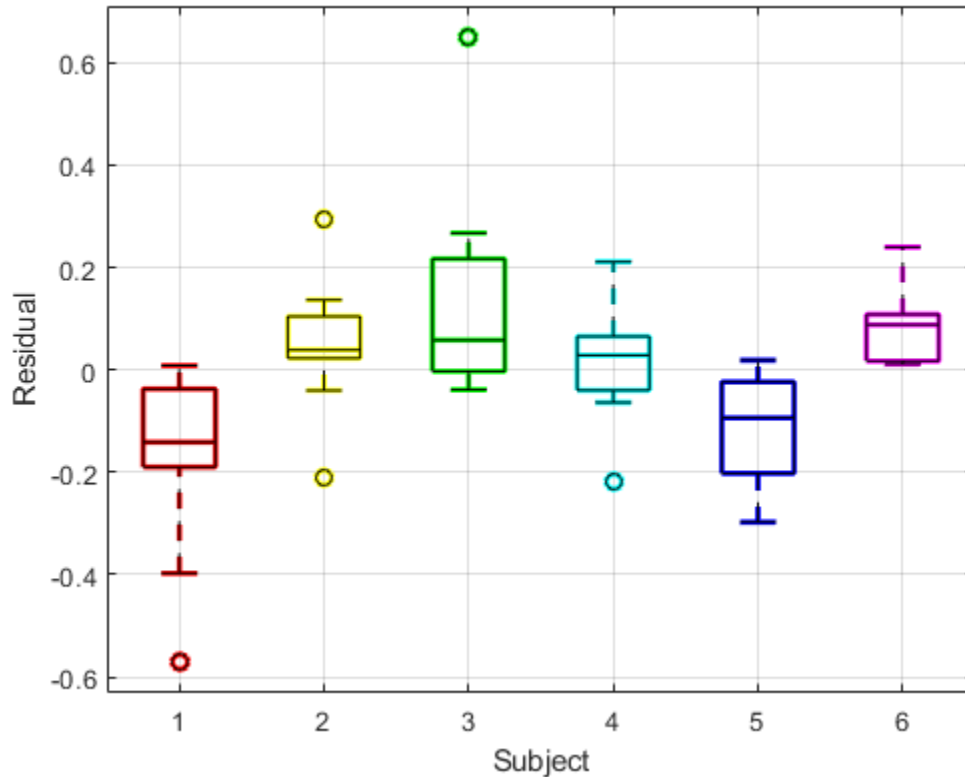


Draw the box-plot of residuals by subject.

```

colors = 'rygcbm';
h = boxplot(res,subject,'colors',colors,'symbol','o');
set(h(~isnan(h)), 'LineWidth',2)
hold on
boxplot(res,subject,'colors','k','symbol','ko')
grid on
xlabel('Subject')
ylabel('Residual')
hold off

```



The box plot of residuals by subject shows that the boxes are mostly above or below zero, indicating that the model has failed to account for subject-specific effects.

To account for subject-specific effects, fit the model separately to the data for each subject.

```
phi0 = [1 2 1 1];
PHI = zeros(4,6);
RES = zeros(11,6);
for I = 1:6
    tI = time(subject == I);
    cI = concentration(subject == I);
    [PHI(:,I),RES(:,I)] = nlinfit(tI,cI,model,phi0);
```

```
end
PHI

PHI =

    2.0293    2.8277    5.4683    2.1981    3.5661    3.0023
    0.5794    0.8013    1.7498    0.2423    1.0408    1.0882
    0.1915    0.4989    1.6757    0.2545    0.2915    0.9685
   -1.7878   -1.6354   -0.4122   -1.6026   -1.5069   -0.8731
```

Compute the mean squared error.

```
numParams = 24;
df = numObs-numParams;
mse = (RES(:)'*RES(:))/df
```

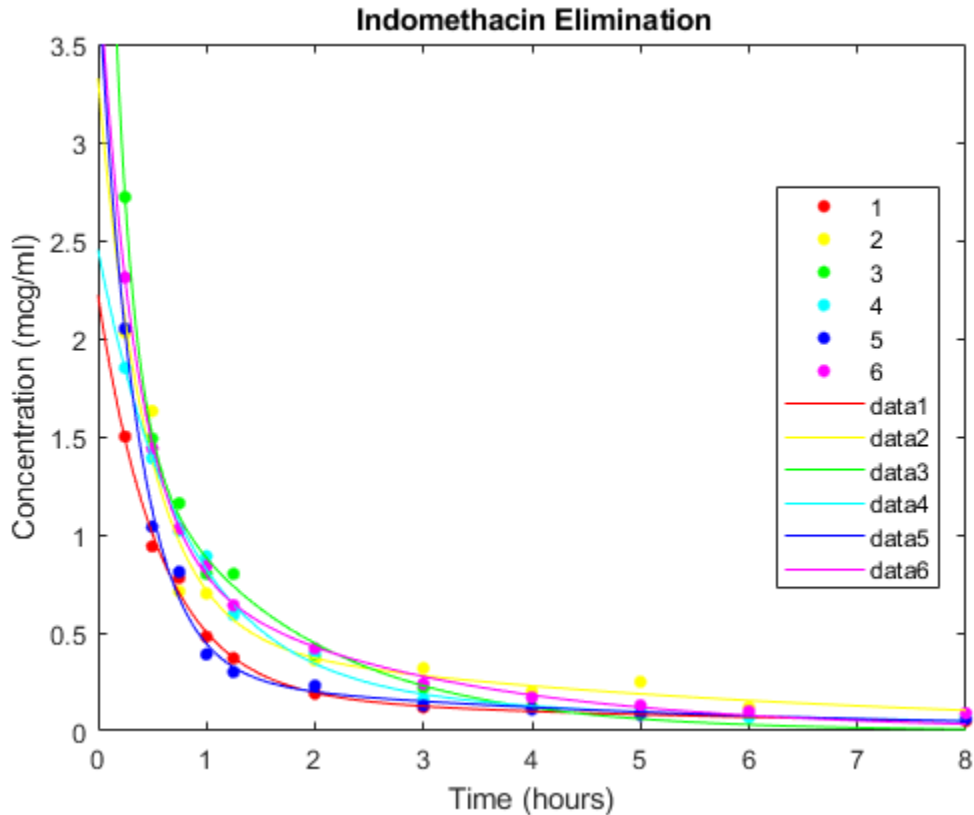
```
mse =

    0.0057
```

Plot the scatter plot of the data and superimpose the model for each subject.

```
gscatter(time, concentration, subject)
xlabel('Time (hours)')
ylabel('Concentration (mcg/ml)')
title('\bf Indomethacin Elimination')
hold on
for I = 1:6
    plot(tplot, model(PHI(:, I), tplot), 'Color', colors(I))
end
axis([0 8 0 3.5])
hold off
```



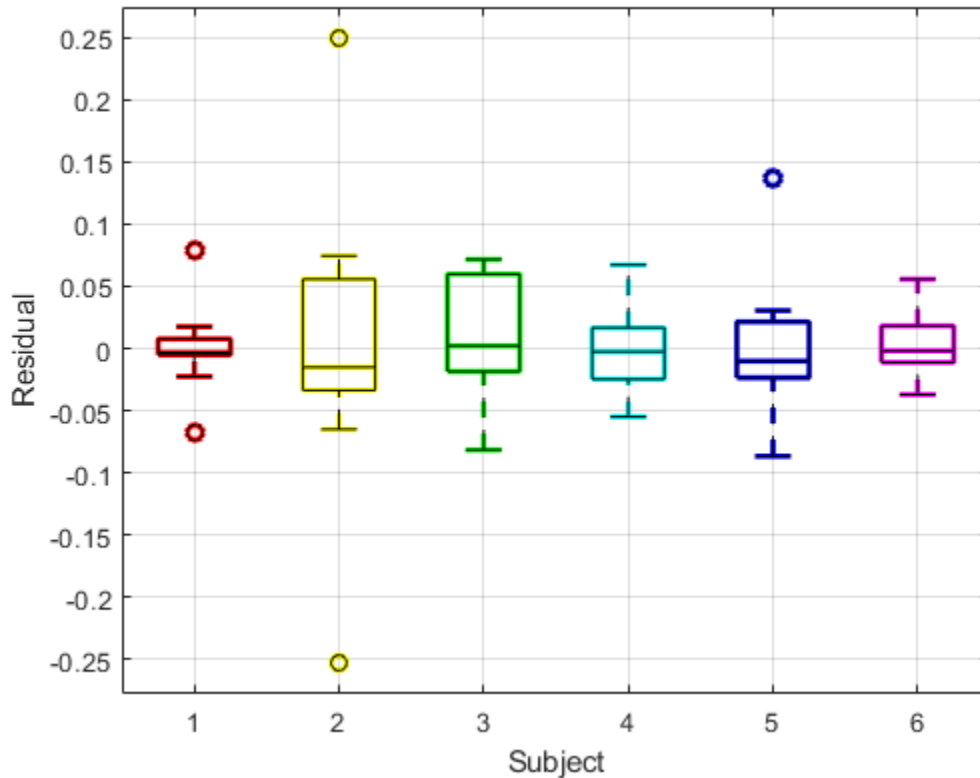


PHI gives estimates of the four model parameters for each of the six subjects. The estimates vary considerably, but taken as a 24-parameter model of the data, the mean-squared error of 0.0057 is a significant reduction from 0.0304 in the original four-parameter model.

Draw the box plot of residuals by subject.

```
h = boxplot(RES, 'colors', colors, 'symbol', 'o');
set(h(~isnan(h)), 'LineWidth', 2)
hold on
boxplot(RES, 'colors', 'k', 'symbol', 'ko')
grid on
xlabel('Subject')
```

```
ylabel('Residual')  
hold off
```



Now the box plot shows that the larger model accounts for most of the subject-specific effects. The spread of the residuals (the vertical scale of the box plot) is much smaller than in the previous box plot, and the boxes are now mostly centered on zero.

While the 24-parameter model successfully accounts for variations due to the specific subjects in the study, it does not consider the subjects as representatives of a larger population. The sampling distribution from which the subjects are drawn is likely more interesting than the sample itself. The purpose of mixed-effects models is to account for subject-specific variations more broadly, as random effects varying around population means.

Use the `nlmeFit` function to fit a mixed-effects model to the data. You can also use `nlmeFitsa` in place of `nlmeFit`.

The following anonymous function, `nlme_model`, adapts the four-parameter model used by `nlmFit` to the calling syntax of `nlmeFit` by allowing separate parameters for each individual. By default, `nlmeFit` assigns random effects to all the model parameters. Also by default, `nlmeFit` assumes a diagonal covariance matrix (no covariance among the random effects) to avoid overparametrization and related convergence issues.

```
nlme_model = @(PHI,t) (PHI(:,1).*exp(-exp(PHI(:,2)).*t) + ...
                    PHI(:,3).*exp(-exp(PHI(:,4)).*t));
phi0 = [1 2 1 1];
[phi,PSI,stats] = nlmeFit(time,concentration,subject, ...
                        [],nlme_model,phi0)
```

```
phi =
```

```
    2.8277
    0.7729
    0.4606
   -1.3459
```

```
PSI =
```

```
    0.3264         0         0         0
         0    0.0250         0         0
         0         0    0.0124         0
         0         0         0    0.0000
```

```
stats =
```

```
struct with fields:
```

```
    dfe: 57
   logl: 54.5882
    mse: 0.0066
   rmse: 0.0787
errorparam: 0.0815
    aic: -91.1765
    bic: -93.0506
   covb: [4x4 double]
```

```
sebeta: [0.2558 0.1066 0.1092 0.2244]
ires: [66x1 double]
pres: [66x1 double]
iwres: [66x1 double]
pwres: [66x1 double]
cwres: [66x1 double]
```

The mean-squared error of 0.0066 is comparable to the 0.0057 of the 24-parameter model without random effects, and significantly better than the 0.0304 of the four-parameter model without random effects.

The estimated covariance matrix  $\Psi$  shows that the variance of the fourth random effect is essentially zero, suggesting that you can remove it to simplify the model. To do this, use the 'REParamsSelect' name-value pair to specify the indices of the parameters to be modeled with random effects in `nlmefit`.

```
[phi,PSI,stats] = nlmefit(time,concentration,subject, ...
                        [],nlme_model,phi0, ...
                        'REParamsSelect',[1 2 3])
```

```
phi =
```

```
2.8277
0.7728
0.4605
-1.3460
```

```
PSI =
```

```
0.3270    0    0
    0    0.0250    0
    0    0    0.0124
```

```
stats =
```

```
struct with fields:
```

```
    dfe: 58
   logl: 54.5875
    mse: 0.0066
```

```

    rmse: 0.0780
  errorparam: 0.0815
    aic: -93.1750
    bic: -94.8410
   covb: [4x4 double]
 sebeta: [0.2560 0.1066 0.1092 0.2244]
   ires: [66x1 double]
   pres: [66x1 double]
  iwres: [66x1 double]
  pwres: [66x1 double]
  cwres: [66x1 double]

```

The log-likelihood `logl` is almost identical to what it was with random effects for all of the parameters, the Akaike information criterion `aic` is reduced from -91.1765 to -93.1750, and the Bayesian information criterion `bic` is reduced from -93.0506 to -94.8410. These measures support the decision to drop the fourth random effect.

Refitting the simplified model with a full covariance matrix allows for identification of correlations among the random effects. To do this, use the `CovPattern` parameter to specify the pattern of nonzero elements in the covariance matrix.

```

[phi,PSI,stats] = nlmeFit(time,concentration,subject, ...
                        [],nlme_model,phi0, ...
                        'REParamsSelect',[1 2 3], ...
                        'CovPattern',ones(3))

```

`phi =`

```

    2.8149
    0.8293
    0.5613
   -1.1407

```

`PSI =`

```

    0.4767    0.1151    0.0499
    0.1151    0.0320    0.0032
    0.0499    0.0032    0.0236

```

`stats =`

```
struct with fields:

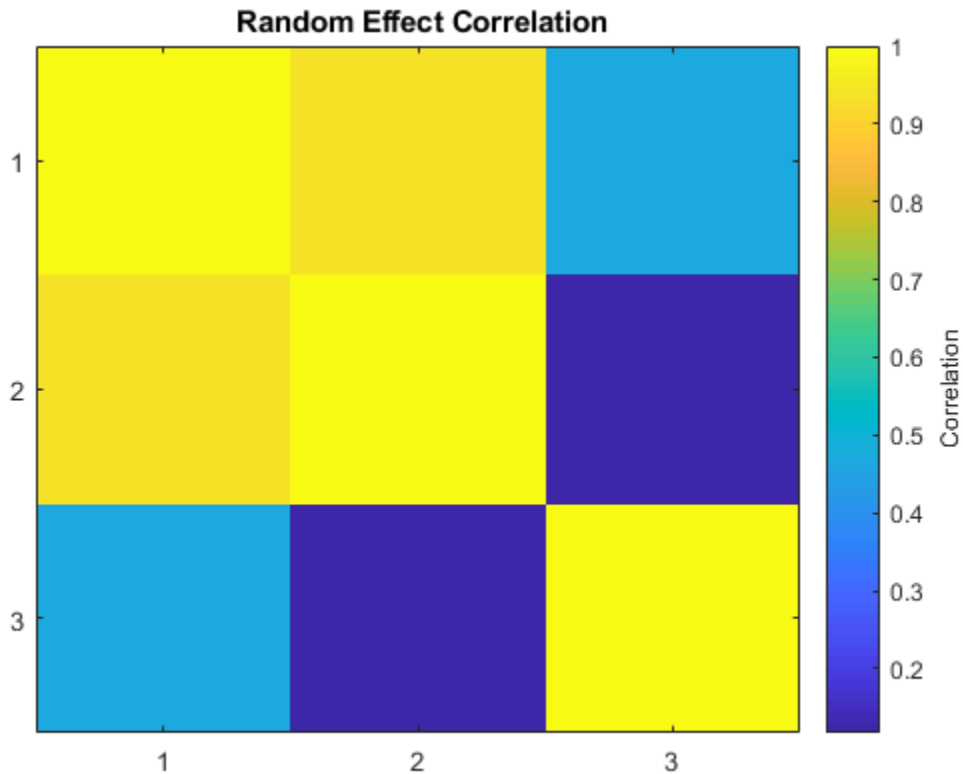
    dfe: 55
    logl: 58.4730
    mse: 0.0061
    rmse: 0.0782
    errorparam: 0.0781
    aic: -94.9461
    bic: -97.2367
    covb: [4x4 double]
    sebeta: [0.3028 0.1103 0.1179 0.1662]
    ires: [66x1 double]
    pres: [66x1 double]
    iwres: [66x1 double]
    pwres: [66x1 double]
    cwres: [66x1 double]
```

The estimated covariance matrix  $\Psi$  shows that the random effects on the first two parameters have a relatively strong correlation, and both have a relatively weak correlation with the last random effect. This structure in the covariance matrix is more apparent if you convert  $\Psi$  to a correlation matrix using `corr cov`.

```
RHO = corrcov(PHI)
clf;
imagesc(RHO)
set(gca, 'XTick', [1 2 3], 'YTick', [1 2 3])
title('\bf Random Effect Correlation')
h = colorbar;
set(get(h, 'YLabel'), 'String', 'Correlation');
```

```
RHO =

    1.0000    0.9316    0.4705
    0.9316    1.0000    0.1175
    0.4705    0.1175    1.0000
```



Incorporate this structure into the model by changing the specification of the covariance pattern to block-diagonal.

```
P = [1 1 0;1 1 0;0 0 1] % Covariance pattern
[phi,PSI,stats,b] = nlmeFit(time,concentration,subject, ...
    [],nlme_model,phi0, ...
    'REParamsSelect',[1 2 3], ...
    'CovPattern',P)
```

P =

```
1    1    0
1    1    0
```

```
      0      0      1

phi =

    2.7830
    0.8981
    0.6581
   -1.0000

PSI =

    0.5180    0.1069         0
    0.1069    0.0221         0
         0         0    0.0454

stats =

struct with fields:

    dfe: 57
    logl: 58.0804
    mse: 0.0061
    rmse: 0.0768
    errorparam: 0.0782
    aic: -98.1608
    bic: -100.0350
    covb: [4x4 double]
    sebeta: [0.3171 0.1073 0.1384 0.1453]
    ires: [66x1 double]
    pres: [66x1 double]
    iwres: [66x1 double]
    pwres: [66x1 double]
    cwres: [66x1 double]

b =

   -0.8507   -0.1563    1.0427   -0.7559    0.5652    0.1550
   -0.1756   -0.0323    0.2152   -0.1560    0.1167    0.0320
   -0.2756    0.0519    0.2620    0.1064   -0.2835    0.1389
```



The block-diagonal covariance structure reduces `aic` from -94.9462 to -98.1608 and `bic` from -97.2368 to -100.0350 without significantly affecting the log-likelihood. These measures support the covariance structure used in the final model. The output `b` gives predictions of the three random effects for each of the six subjects. These are combined with the estimates of the fixed effects in `phi` to produce the mixed-effects model.

Plot the mixed-effects model for each of the six subjects. For comparison, the model without random effects is also shown.

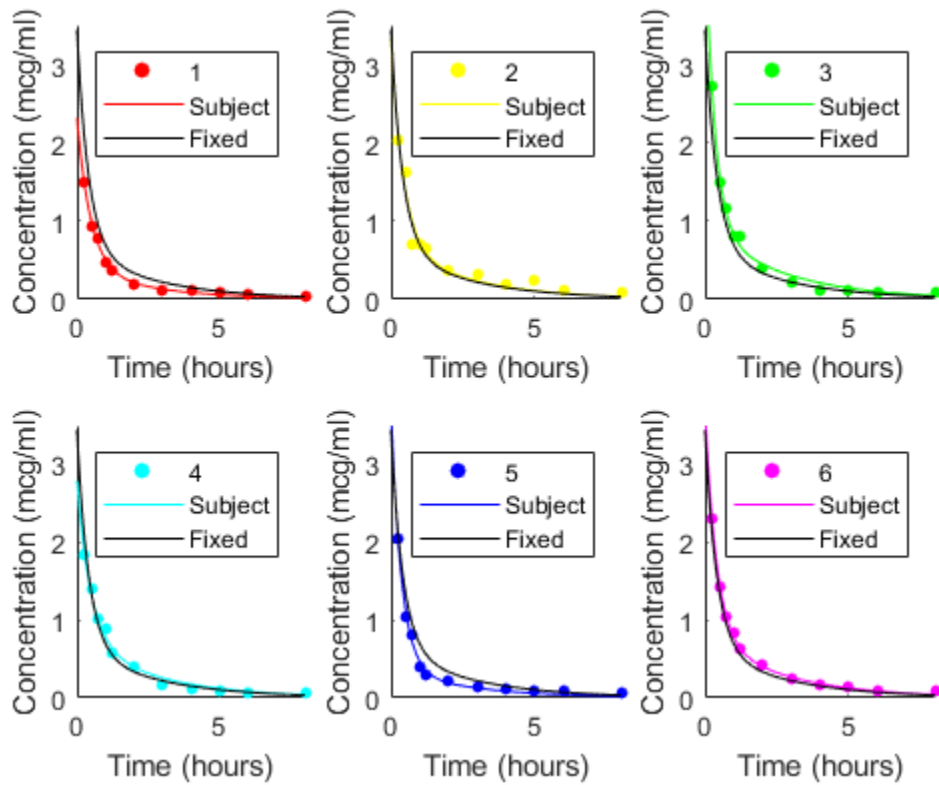
```

PHI = repmat(phi,1,6) + ...           % Fixed effects
      [b(1,:);b(2,:);b(3,:);zeros(1,6)]; % Random effects
RES = zeros(11,6); % Residuals
colors = 'rygcbm';
for I = 1:6
    fitted_model = @(t) (PHI(1,I)*exp(-exp(PHI(2,I))*t) + ...
                        PHI(3,I)*exp(-exp(PHI(4,I))*t));

    tI = time(subject == I);
    cI = concentration(subject == I);
    RES(:,I) = cI - fitted_model(tI);

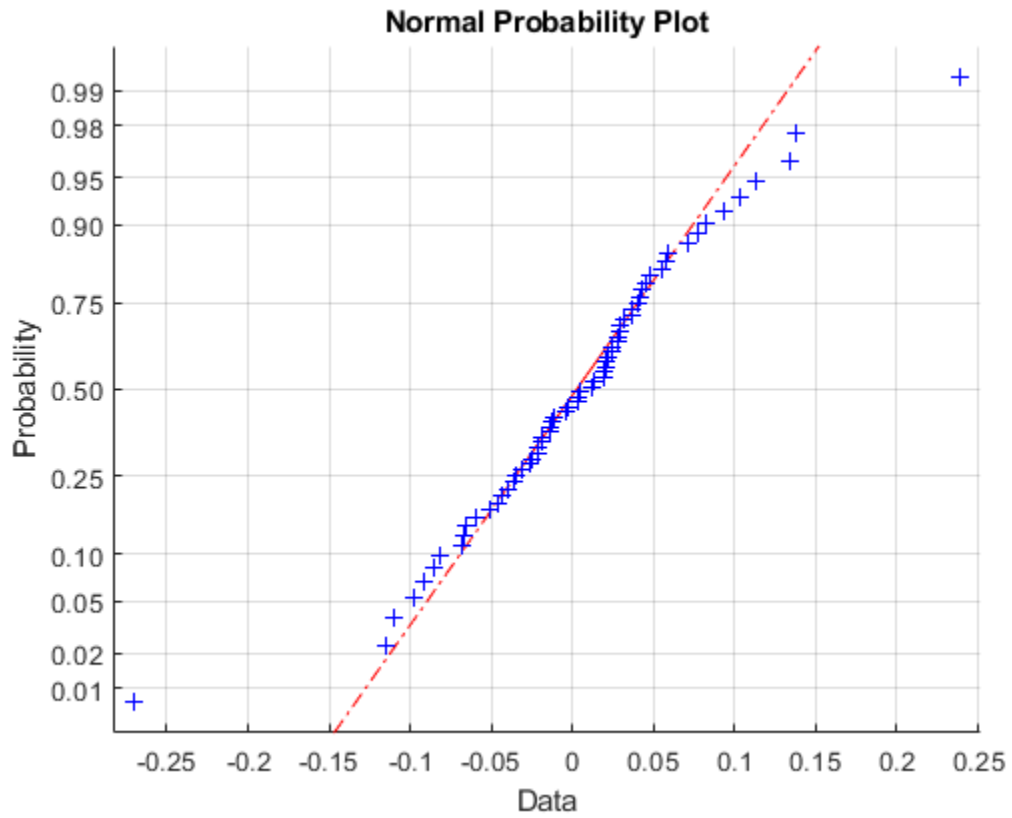
    subplot(2,3,I)
    scatter(tI,cI,20,colors(I),'filled')
    hold on
    plot(tplot,fitted_model(tplot),'Color',colors(I))
    plot(tplot,model(phi,tplot),'k')
    axis([0 8 0 3.5])
    xlabel('Time (hours)')
    ylabel('Concentration (mcg/ml)')
    legend(num2str(I),'Subject','Fixed')
end

```



If obvious outliers in the data (visible in previous box plots) are ignored, a normal probability plot of the residuals shows reasonable agreement with model assumptions on the errors.

```
clf; normplot(RES(:))
```





# Survival Analysis

---

- “What Is Survival Analysis?” on page 14-2
- “Kaplan-Meier Method” on page 14-10
- “Hazard and Survivor Functions for Different Groups” on page 14-17
- “Survivor Functions for Two Groups” on page 14-24
- “Cox Proportional Hazards Model” on page 14-29
- “Cox Proportional Hazards Model for Censored Data” on page 14-35
- “Cox Proportional Hazards Model with Time-Dependent Covariates” on page 14-41

## What Is Survival Analysis?

In this section...
“Introduction” on page 14-2
“Censoring” on page 14-2
“Data” on page 14-3
“Survivor Function” on page 14-4
“Hazard Function” on page 14-6

### Introduction

Survival analysis is time-to-event analysis, that is, when the outcome of interest is the time until an event occurs. Examples of time-to-events are the time until infection, reoccurrence of a disease, or recovery in health sciences, duration of unemployment in economics, time until the failure of a machine part or lifetime of light bulbs in engineering, and so on. Survival analysis is a part of reliability studies in engineering. In this case, it is usually used to study the lifetime of industrial components. In reliability analyses, survival times are usually called failure times as the variable of interest is how much time a component functions properly before it fails.

Survival analysis consists of parametric, semiparametric, and nonparametric methods. You can use these to estimate the most commonly used measures in survival studies, survivor and hazard functions, compare them for different groups, and assess the relationship of predictor variables to survival time. Some statistical probability distributions describe survival times well. Commonly used distributions are exponential, Weibull, lognormal, Burr, and Birnbaum-Saunders distributions. Statistics and Machine Learning Toolbox functions `ecdf` and `ksdensity` compute the empirical and kernel density estimates of the cdf, cumulative hazard, and survivor functions. `coxphfit` fits the Cox proportional hazards model to the data.

### Censoring

One important concept in survival analysis is censoring. The survival times of some individuals might not be fully observed due to different reasons. In life sciences, this might happen when the survival study (e.g., the clinical trial) stops before the full survival times of all individuals can be observed, or a person drops out of a study, or for long-term studies, when the patient is lost to follow up. In the industrial context, not all

components might have failed before the end of the reliability study. In such cases, the individual survives beyond the time of the study, and the exact survival time is unknown. This is called right censoring.

During a survival study either the individual is observed to fail at time  $T$ , or the observation on that individual ceases at time  $c$ . Then the observation is  $\min(T,c)$  and an indicator variable  $I_c$  shows if the individual is censored or not. The calculations for hazard and survivor functions must be adjusted to account for censoring. Statistics and Machine Learning Toolbox functions such as `ecdf`, `ksdensity`, `coxphfit`, `mle` account for censoring.

## Data

Survival data usually consists of the time until an event of interest occurs and the censoring information for each individual or component. The following table shows the fictitious unemployment time of individuals in a 6-month study. Two individuals are right censored (indicated by a censoring value of 1). One individual was still unemployed after the 24th week, when the study ended. Contact with the other censored individual was lost at the end of the 21st week.

Unemployment Time (Weeks)	Censoring
14	0
23	0
7	0
21	1
19	0
16	0
24	1
8	0

Survival data might also include the number of failures at a certain time (the number of times a particular survival or failure time was observed). The following table shows the simulated time until a light-emitting diodes drops to 70% of its full light output level, in hours, in an accelerated life test.

Failure Time (hrs)	Frequency
8600	6

Failure Time (hrs)	Frequency
15300	19
22000	11
28600	20
35300	17
42000	14
48700	8
55400	2
62100	0
68800	2

Data might also have information on the predictor variables, to use in semi-parametric regression-like methods such as Cox proportional hazards regression.

Time Until Recovery (weeks)	Censoring	Gender	Systolic Blood Pressure	Diastolic Blood Pressure
12	1	Male	124	93
20	0	Female	109	77
7	0	Female	125	83
13	0	Male	117	75
9	1	Male	122	80
15	0	Female	121	70
17	1	Male	130	88
8	0	Female	115	82
14	0	Male	118	86

## Survivor Function

The survivor function is the probability of survival as a function of time. It is also called the survival function. It gives the probability that the survival time of an individual exceeds a certain value. Since the cumulative distribution function,  $F(t)$ , is the probability that the survival time is less than or equal to a given point in time, the

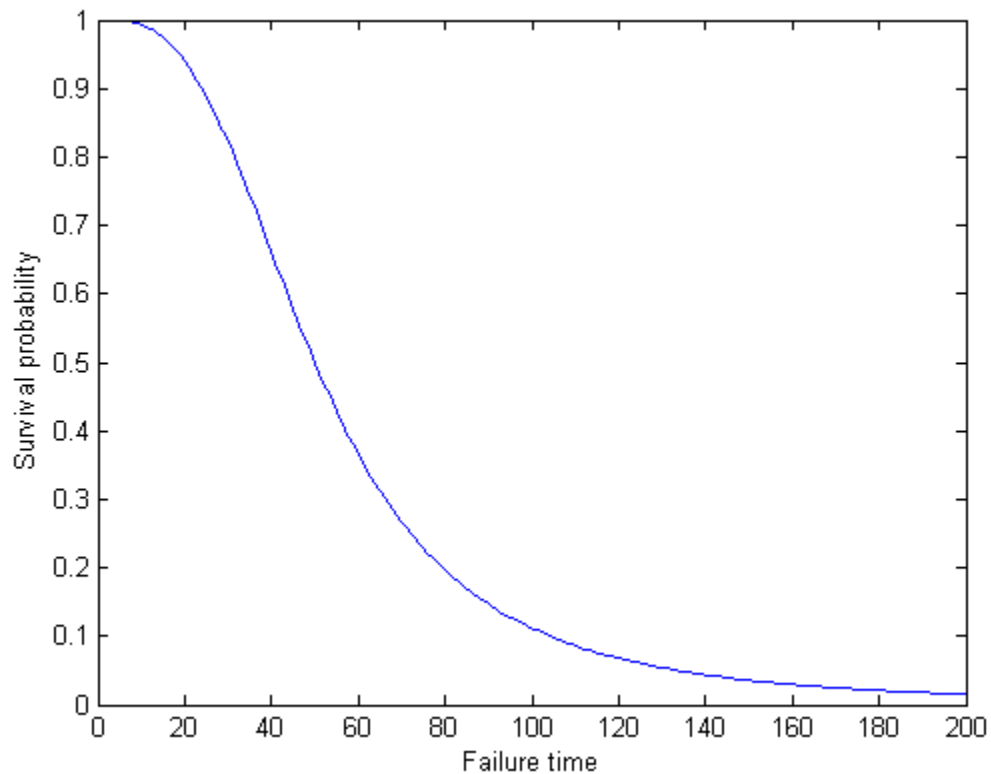


survival function for a continuous distribution,  $S(t)$ , is the complement of the cumulative distribution function:

$$S(t) = 1 - F(t).$$

For example, for data coming from a Burr distribution with parameters 50, 3, and 1, you can calculate and plot the survivor function.

```
x = 0:0.1:200;  
figure()  
plot(x,1-cdf('Burr',x,50,3,1))  
xlabel('Failure time');  
ylabel('Survival probability');
```



The survivor function is also related to the hazard function on page 14-6. If the data has the hazard function,  $h(t)$ , then the survivor function is

$$S(t) = \exp\left(-\int_0^t h(u) du\right),$$

which corresponds to

$$S(t) = \exp(-H(t)),$$

where  $H(t)$  is the cumulative hazard function.

## Hazard Function

The hazard function gives the instantaneous failure rate of an individual conditioned on the fact that the individual survived until a given time. That is,

$$h(t) = \lim_{\Delta t \rightarrow 0} \frac{P(t \leq T < t + \Delta t \mid T \geq t)}{\Delta t},$$

where  $\Delta t$  is a very small time interval. The hazard rate, therefore, is sometimes called the conditional failure rate. The hazard function always takes a positive value. However, these values do not correspond to probabilities and might be greater than 1.

The hazard function is related to the probability density function,  $f(t)$ , cumulative distribution function,  $F(t)$ , and survivor function,  $S(t)$ , as follows:

$$h(t) = \frac{f(t)}{S(t)} = \frac{f(t)}{1 - F(t)},$$

which is also equivalent to

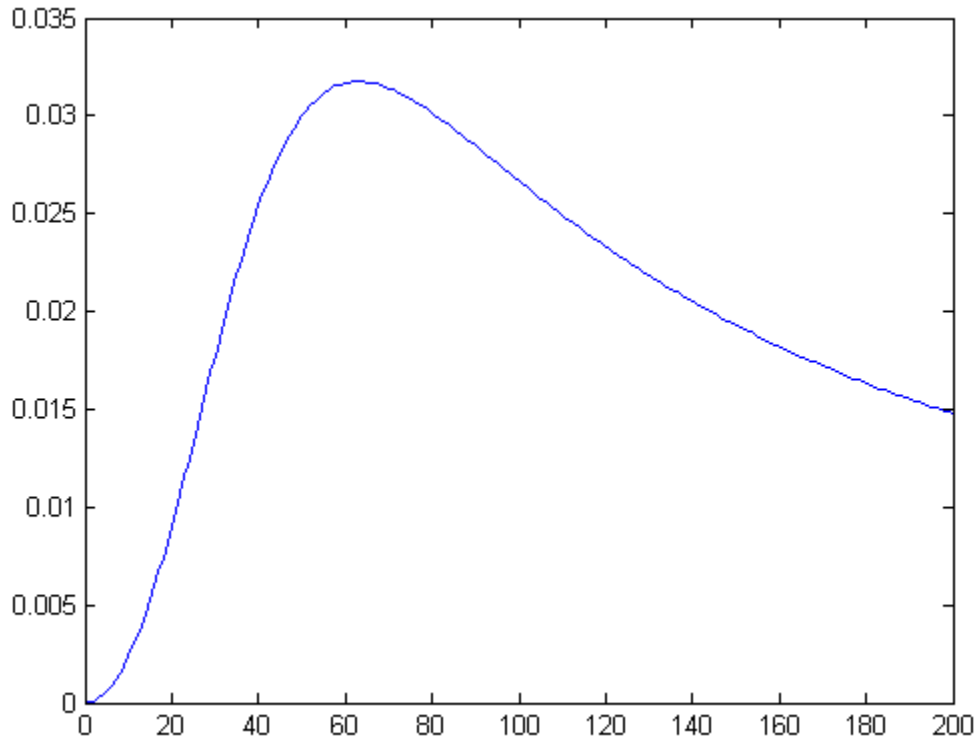
$$h(t) = -\frac{d}{dt} \ln S(t).$$

So, if you know the shape of the survival function, you can also derive the corresponding hazard function.

For example, for data coming from a Burr distribution with parameters 50, 3, and 1, you can calculate and plot the hazard function.

```
x = 0:1:200;
Burrhazard = pdf('Burr', x, 50, 3, 1) ./ (1 - cdf('Burr', x, 50, 3, 1));
figure()
plot(x, Burrhazard)
```

```
xlabel('Failure time');
ylabel('Hazard rate');
```



There are different types of hazard functions. The previous figure shows a situation when the hazard rate increases for the early time periods and then gradually decreases. The hazard rate might also be monotonically decreasing, increasing, or constant over time. The following figure shows examples of different types of hazard functions for data coming from different Weibull distributions.

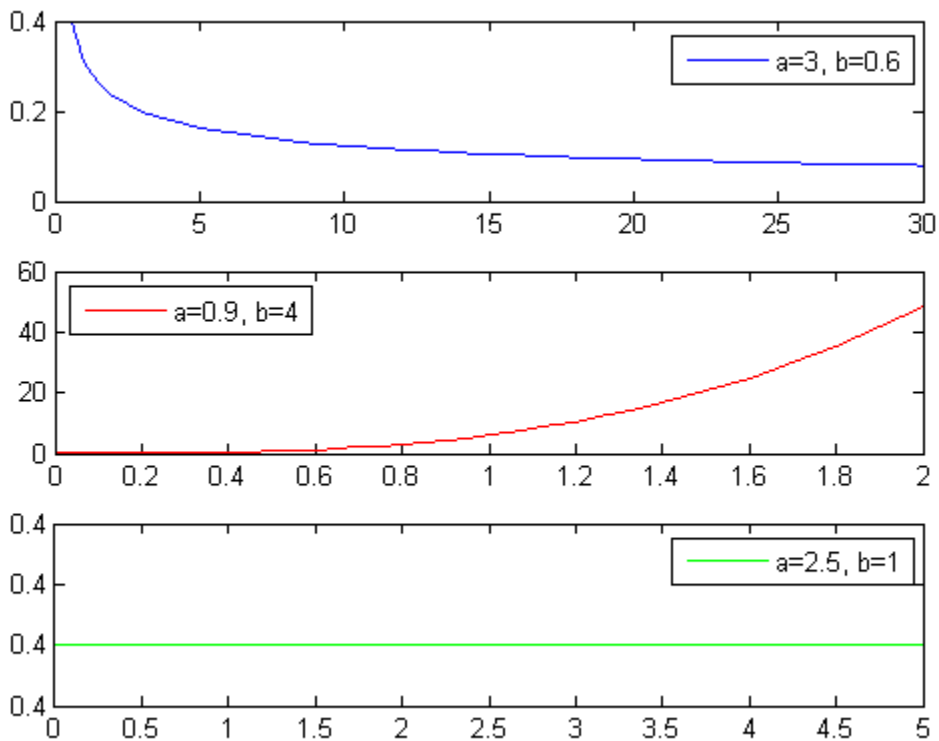
```
ax1 = subplot(3,1,1);
x1 = 0:0.5:30;
hazard1 = pdf('wbl',x1,3,0.6)./(1-cdf('wbl',x1,3,0.6));
plot(x1,hazard1)

ax2 = subplot(3,1,2);
```

```
x2 = 0:0.05:2;
hazard2 = pdf('wbl',x2,0.9,4) ./ (1-cdf('wbl',x2,0.9,4));
plot(x2,hazard2,'color','r')

ax3 = subplot(3,1,3);
x3 = 0:0.05:5;
hazard3 = pdf('wbl',x3,2.5,1) ./ (1-cdf('wbl',x3,2.5,1));
plot(x3,hazard3)

set(ax1,'Ylim',[0 0.4]);
legend(ax1,'a=3, b=0.6');
legend(ax2,'a=0.9, b=4','location','northwest');
legend(ax3,'a=2.5, b=1');
```



In the third case, the Weibull distribution has a shape parameter value of 1, which corresponds to the exponential distribution. The exponential distribution always has a constant hazard rate over time.

## References

- [1] Cox, D. R., and D. Oakes. *Analysis of Survival Data*. London: Chapman & Hall, 1984.
- [2] Lawless, J. F. *Statistical Models and Methods for Lifetime Data*. Hoboken, NJ: Wiley-Interscience, 2002.
- [3] Kleinbaum, D. G., and M. Klein. *Survival Analysis*. Statistics for Biology and Health. 2nd edition. Springer, 2005.

## See Also

`coxphfit` | `ecdf` | `ksdensity`

## Related Examples

- “Hazard and Survivor Functions for Different Groups” on page 14-17
- “Survivor Functions for Two Groups” on page 14-24
- “Cox Proportional Hazards Model for Censored Data” on page 14-35
- “Cox Proportional Hazards Model with Time-Dependent Covariates” on page 14-41

## More About

- “Kaplan-Meier Method” on page 14-10
- “Cox Proportional Hazards Model” on page 14-29

## Kaplan-Meier Method

The Statistics and Machine Learning Toolbox function `ecdf` produces the empirical cumulative hazard, survivor, and cumulative distribution functions by using the Kaplan-Meier nonparametric method. The Kaplan-Meier estimator for the survivor function is also called the *product-limit estimator*.

The Kaplan-Meier method uses survival data summarized in life tables. Life tables order data according to ascending failure times, but you don't have to enter the failure/survival times in an ordered manner to use `ecdf`.

A life table usually consists of:

- Failure times
- Number of items failed at a time/time period
- Number of items censored at a time/time period
- Number of items at risk at the beginning of a time/time period

The number at risk is the total number of survivors at the beginning of each period. The number at risk at the beginning of the first period is all individuals in the lifetime study. At the beginning of each remaining period, the number at risk is reduced by the number of failures plus individuals censored at the end of the previous period.

This life table shows fictitious survival data. At the beginning of the first failure time, there are seven items at risk. At time 4, three fail. So at the beginning of time 7, there are four items at risk. Only one fails at time 7, so the number at risk at the beginning of time 11 is three. Two fail at time 11, so at the beginning of time 12, the number at risk is one. The remaining item fails at time 12.

Failure Time (t)	Number Failed	Number at Risk
4	3	7
7	1	4
11	2	3
12	1	1

You can estimate the hazard, cumulative hazard, survival, and cumulative distribution functions using the life tables as described next.

### Cumulative Hazard Rate (Failure Rate)

The hazard rate at each period is the number of failures in the given period divided by the number of surviving individuals at the beginning of the period (number at risk).

Failure Time ( $t$ )	Hazard Rate ( $h(t)$ )	Cumulative Hazard Rate
0	0	0
$t_1$	$d_1/r_1$	$d_1/r_1$
$t_2$	$d_2/r_2$	$h(t_1) + d_2/r_2$
...	...	...
$t_n$	$d_n/r_n$	$h(t_{n-1}) + d_n/r_n$

### Survival Probability

For each period, the survival probability is the product of the complement of hazard rates. The initial survival probability at the beginning of the first time period is 1. If the hazard rate for the each period is  $h(t_i)$ , then the survivor probability is as shown.

Time ( $t$ )	Survival Probability ( $S(t)$ )
0	1
$t_1$	$1*(1 - h(t_1))$
$t_2$	$S(t_1)*(1 - h(t_2))$
...	...
$t_n$	$S(t_{n-1})*(1 - h(t_n))$

### Cumulative Distribution Function

Because the cumulative distribution function (cdf) and the survivor function are complements of each other, you can find the cdf from the life tables using  $F(t) = 1 - S(t)$ .

You can compute the cumulative hazard rate, survival rate, and cumulative distribution function for the simulated data in the first table on this page as follows.

$t$	Number Failed ( $d$ )	Number at Risk ( $r$ )	Hazard Rate	Survival Probability	Cumulative Distribution Function
4	3	7	$3/7$	$1 - 3/7 = 4/7 = 0.5714$	0.4286
7	1	4	$1/4$	$4/7*(1 - 1/4) = 3/7 = .4286$	0.5714

$t$	Number Failed ( $d$ )	Number at Risk ( $r$ )	Hazard Rate	Survival Probability	Cumulative Distribution Function
11	2	3	2/3	$3/7 * (1 - 2/3) = 1/7 = 0.1429$	0.8571
12	1	1	1/1	$1/7 * (1 - 1) = 0$	1

This rates in this example are based on the discrete failure times, and hence the calculations do not necessarily follow the derivative-based definition in “What Is Survival Analysis?” on page 14-2

Here is how you can enter the data and calculate these measures using `ecdf`. The data does not necessarily have to be in ascending order. Suppose the failure times are stored in an array `y`.

```
y = [4 7 11 12];
freq = [3 1 2 1];
[f,x] = ecdf(y, 'frequency', freq)
```

```
f =
```

```

0
0.4286
0.5714
0.8571
1.0000
```

```
x =
```

```

4
4
7
11
12
```

When you have censored data, the life table might look like the following:



Time ( $t$ )	Number failed ( $d$ )	Censoring	Number at Risk ( $r$ )	Hazard Rate	Survival Probability	Cumulative Distribution Function
4	2	1	7	$2/7$	$1 - 2/7 = 0.7143$	0.2857
7	1	0	4	$1/4$	$0.7143 * (1 - 1/4) = 0.5357$	0.4643
11	1	1	3	$2/3$	$0.5357 * (1 - 1/3) = 0.3571$	0.6429
12	1	0	1	$1/1$	$0.3571 * (1 - 1) = 0$	1.0000

At any given time, the censored items are also considered in the total of number at risk, and the hazard rate formula is based on the number failed and the total number at risk. While updating the number at risk at the beginning of each period, the total number failed and censored in the previous period is reduced from the number at risk at the beginning of that period.

While using `ecdf`, you must also enter the censoring information using an array of binary variables. Enter 1 for censored data, and enter 0 for exact failure time.

```
y = [4 4 4 7 11 11 12];
cens = [0 1 0 0 1 0 0];
[f,x] = ecdf(y, 'censoring', cens)
```

```
f =
```

```

0
0.2857
0.4643
0.6429
1.0000
```

```
x =
```

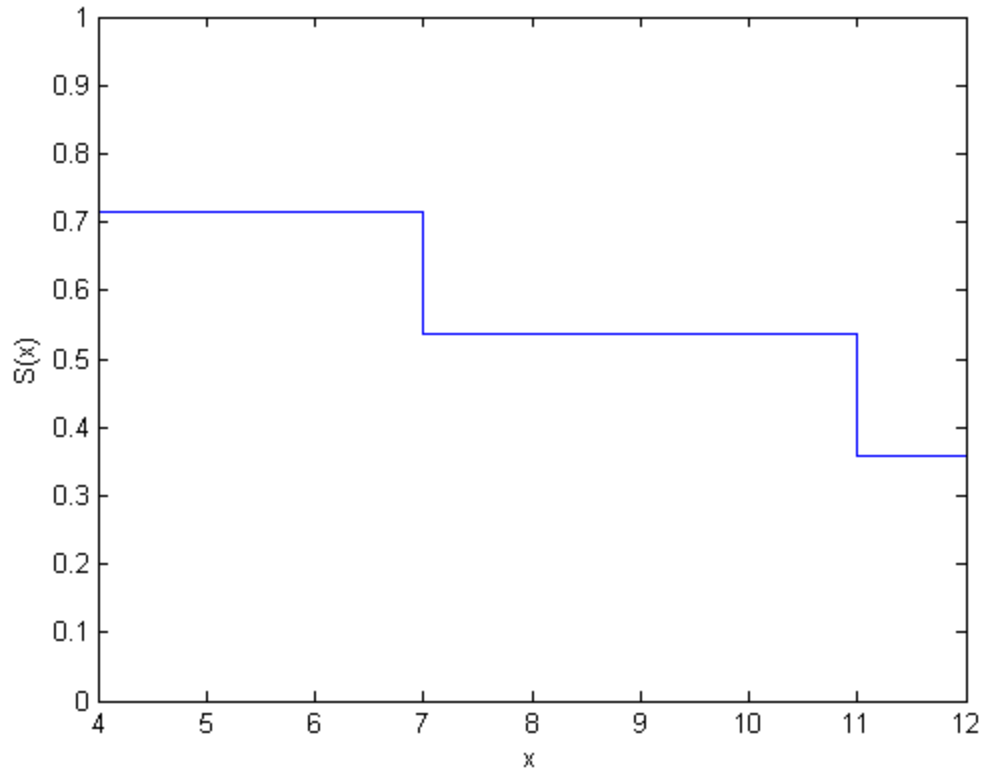
```

4
4
7
```

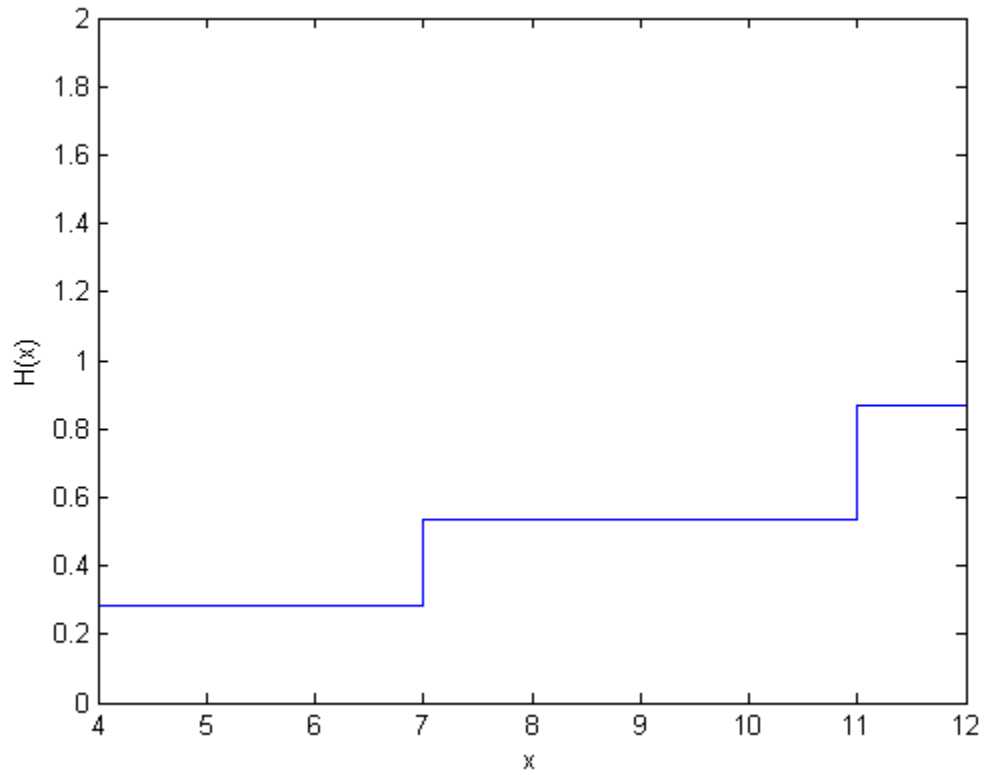
```
11  
12
```

`ecdf`, by default, produces the cumulative distribution function values. You have to specify the survivor function or the hazard function using optional name-value pair arguments. You can also plot the results as follows.

```
figure()  
ecdf(y, 'censoring', cens, 'function', 'survivor');
```



```
figure()  
ecdf(y, 'censoring', cens, 'function', 'cumulative hazard');
```



## References

- [1] Cox, D. R., and D. Oakes. *Analysis of Survival Data*. London: Chapman & Hall, 1984.
- [2] Lawless, J. F. *Statistical Models and Methods for Lifetime Data*. Hoboken, NJ: Wiley-Interscience, 2002.
- [3] Kleinbaum, D. G., and M. Klein. *Survival Analysis*. Statistics for Biology and Health. 2nd edition. Springer, 2005.

### See Also

`coxphfit` | `ecdf` | `ksdensity`

### Related Examples

- “Hazard and Survivor Functions for Different Groups” on page 14-17
- “Survivor Functions for Two Groups” on page 14-24
- “Cox Proportional Hazards Model for Censored Data” on page 14-35
- “Cox Proportional Hazards Model with Time-Dependent Covariates” on page 14-41

### More About

- “What Is Survival Analysis?” on page 14-2
- “Cox Proportional Hazards Model” on page 14-29

## Hazard and Survivor Functions for Different Groups

This example shows how to estimate and plot the cumulative hazard and survivor functions for different groups.

### Step 1. Load and organize sample data.

Load the sample data.

```
load(fullfile(matlabroot, 'examples', 'stats', 'readmissiontimes.mat'))
```

The data has readmission times of patients with information on their gender, age, weight, smoking status, and censorship. This is simulated data.

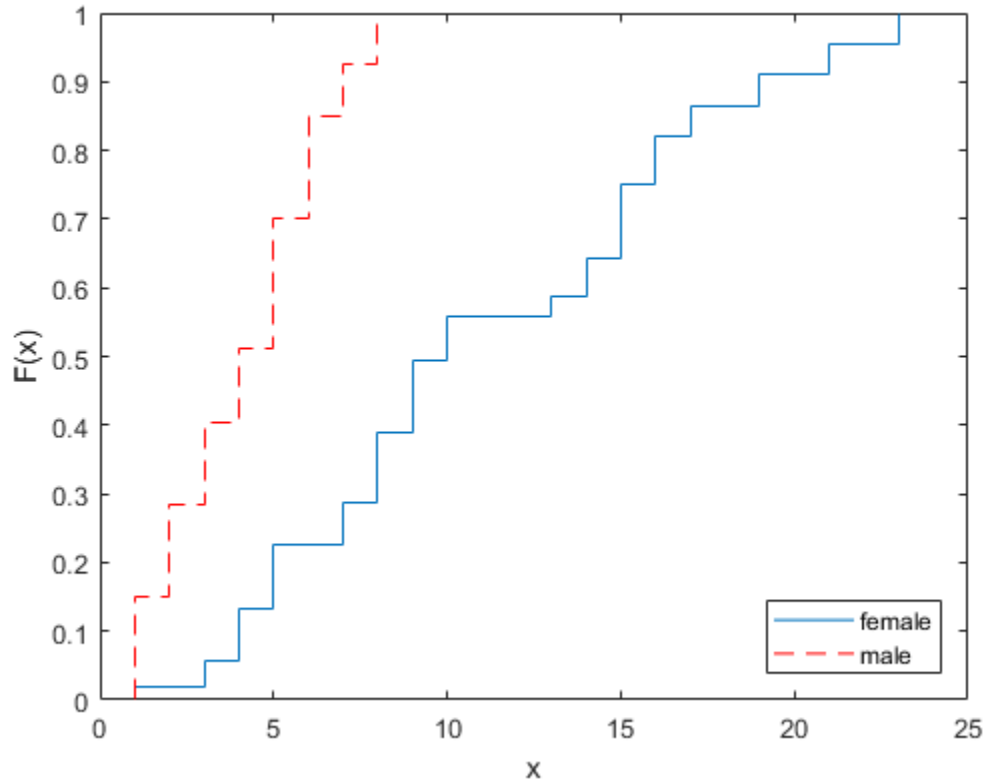
Create a matrix of readmission times and censoring for each gender.

```
female = [ReadmissionTime(Sex==1), Censored(Sex==1)];  
male = [ReadmissionTime(Sex==0), Censored(Sex==0)];
```

### Step 2. Estimate and plot cumulative distribution function for each gender.

Plot the Kaplan-Meier estimate of the cumulative distribution function for female and male patients.

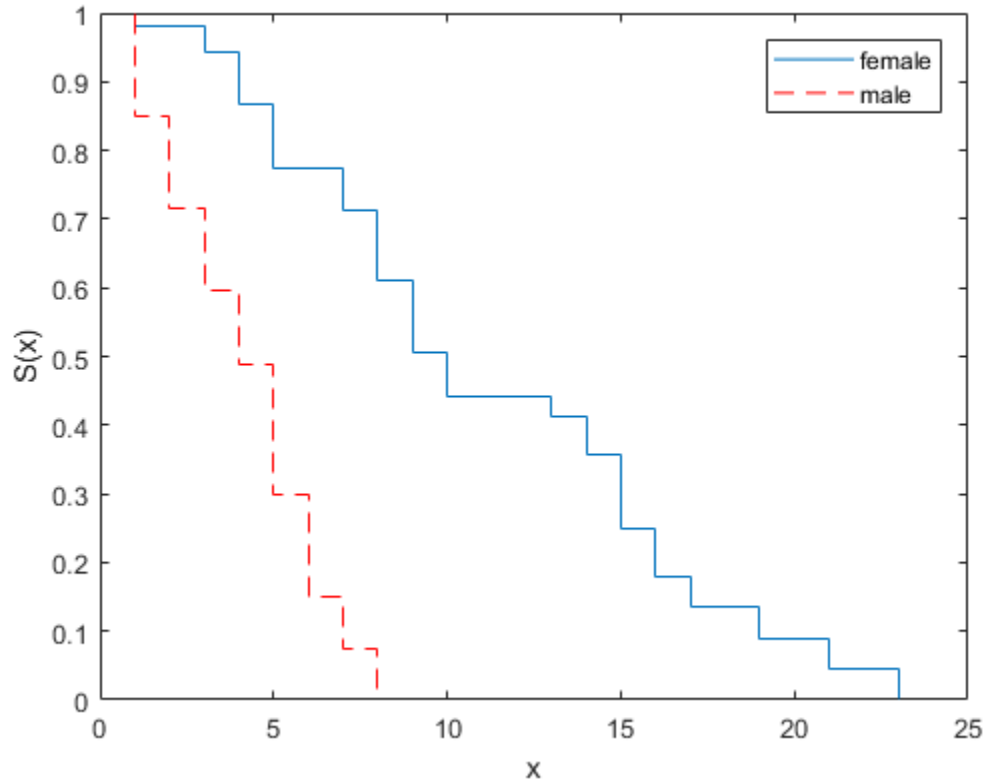
```
figure()  
ecdf(gca, female(:,1), 'Censoring', female(:,2));  
hold on  
[f,x] = ecdf(male(:,1), 'Censoring', male(:,2));  
stairs(x,f, '--r')  
hold off  
legend('female', 'male', 'Location', 'SouthEast')
```



### Step 3. Plot survivor functions.

Compare the survivor functions for female and male patients.

```
figure()
ax1 = gca;
ecdf(ax1, female(:,1), 'Censoring', female(:,2), 'function', 'survivor');
hold on
[f,x] = ecdf(male(:,1), 'Censoring', male(:,2), 'function', 'survivor');
stairs(x,f, '--r')
legend('female', 'male')
```



This figure shows that readmission times are shorter for male patients than female patients.

#### Step 4. Fit Weibull survivor functions.

Fit Weibull distributions to readmission times of female and male patients.

```
pd = fitdist(female(:,1), 'wbl', 'Censoring', female(:,2))
```

```
pd =
```

```
WeibullDistribution
```

```
Weibull distribution
  A = 12.5593    [10.749, 14.6745]
  B = 1.99834   [1.56489, 2.55185]

pd2 = fitdist(male(:,1), 'wbl', 'Censoring', male(:,2))

pd2 =

WeibullDistribution

Weibull distribution
  A = 4.63991    [3.91039, 5.50551]
  B = 1.94422   [1.48496, 2.54552]

pd2 = fitdist(male(:,1), 'wbl', 'Censoring', male(:,2))

pd2 =

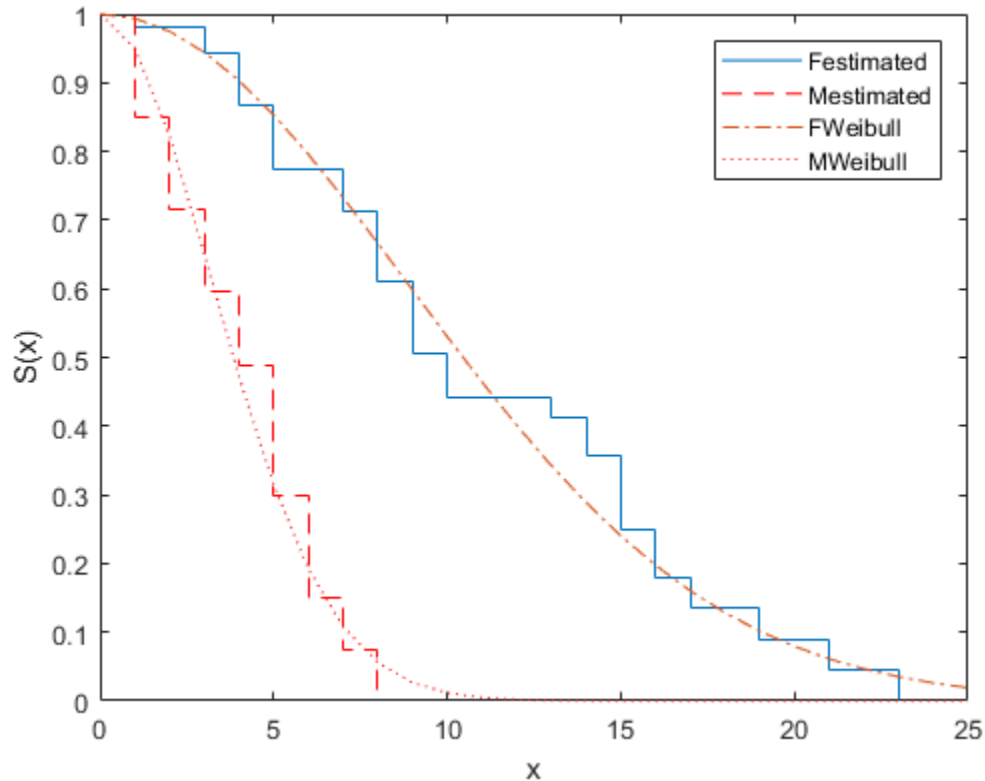
WeibullDistribution

Weibull distribution
  A = 4.63991    [3.91039, 5.50551]
  B = 1.94422   [1.48496, 2.54552]
```

Plot the Weibull survivor functions for female and male patients on estimated survivor functions.

```
plot(0:1:25, 1-cdf('wbl', 0:1:25, 12.5593, 1.99834), '-.')
plot(0:1:25, 1-cdf('wbl', 0:1:25, 4.63991, 1.94422), ':r')
hold off
legend('Festimated', 'Mestimated', 'FWeibull', 'MWeibull')
```





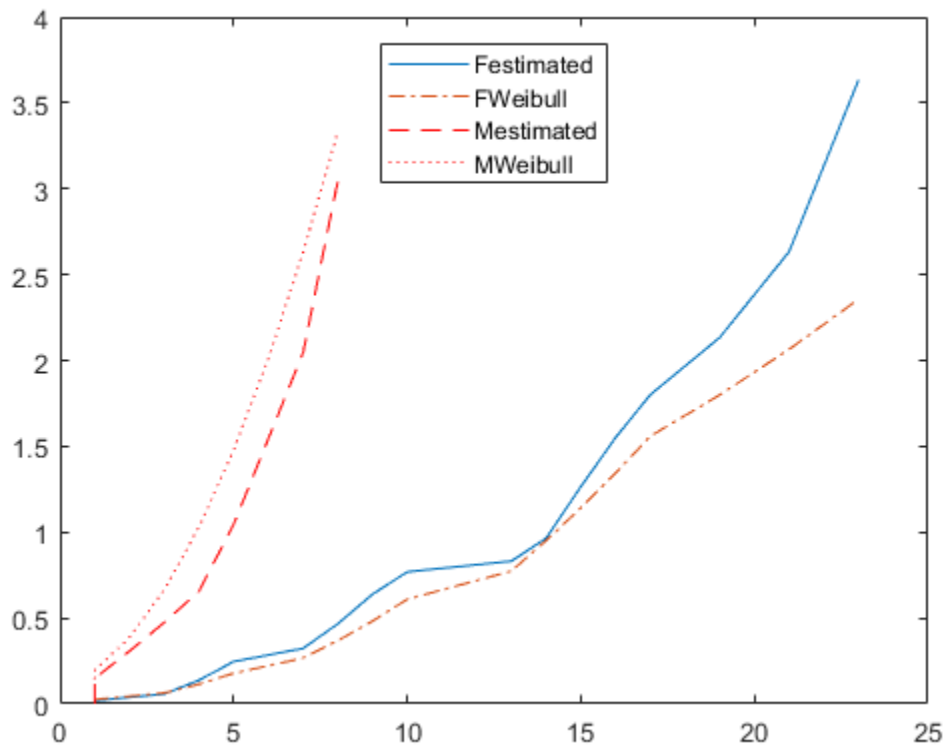
Weibull distribution provides a good fit for the data.

### Step 5. Estimate cumulative hazard and fit Weibull cumulative hazard functions.

Estimate the cumulative hazard function for the genders and fit Weibull cumulative hazard functions.

```
figure()
[f,x] = ecdf(female(:,1), 'Censoring', female(:,2), ...
'function', 'cumhazard');
plot(x,f)
hold on
plot(x,cumsum(pdf(pd,x) ./ (1-cdf(pd,x))), '-.')
[f,x] = ecdf(male(:,1), 'Censoring', male(:,2), ...
```

```
'function','cumhazard');  
plot(x,f,'--r')  
plot(x,cumsum(pdf(pd2,x)/(1-cdf(pd2,x))),':r')  
legend('Festimated','FWeibull','Mestimated','MWeibull',...  
'Location','North')
```



## See Also

coxphfit | ecdf | ksdensity

## **Related Examples**

- “Survivor Functions for Two Groups” on page 14-24
- “Cox Proportional Hazards Model for Censored Data” on page 14-35
- “Cox Proportional Hazards Model with Time-Dependent Covariates” on page 14-41

## **More About**

- “What Is Survival Analysis?” on page 14-2
- “Kaplan-Meier Method” on page 14-10
- “Cox Proportional Hazards Model” on page 14-29

## Survivor Functions for Two Groups

This example shows how to find the empirical survivor functions and the parametric survivor functions using the Burr type XII distribution fit to data for two groups.

### Step 1. Load and prepare sample data.

Load the sample data.

```
load(fullfile(matlabroot, 'examples', 'stats', 'lightbulb.mat'))
```

The first column of the data has the lifetime (in hours) of two types of light bulbs. The second column has information about the type of light bulb. 0 indicates fluorescent bulbs whereas 1 indicates the incandescent bulb. The third column has censoring information. 1 indicates censored data, and 0 indicates the exact failure time. This is simulated data.

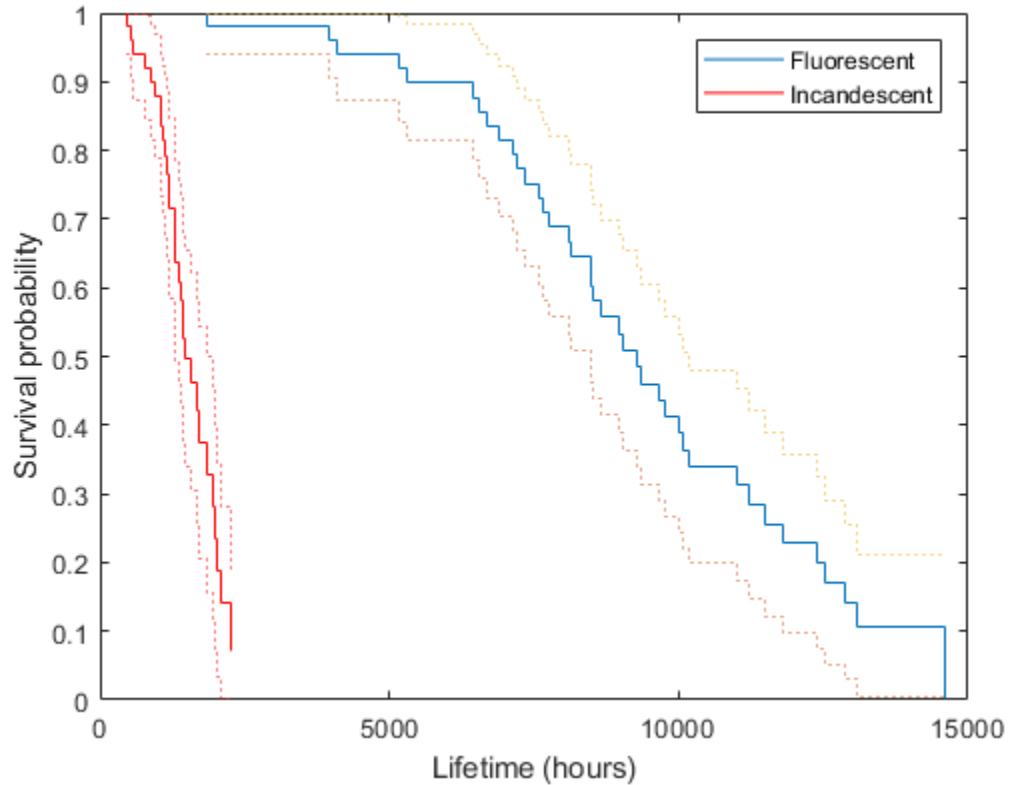
Create a variable for each light bulb type and also include the censorship information.

```
fluo = [lightbulb(lightbulb(:,2)==0,1), ...  
        lightbulb(lightbulb(:,2)==0,3)];  
insc = [lightbulb(lightbulb(:,2)==1,1), ...  
        lightbulb(lightbulb(:,2)==1,3)];
```

### Step 2. Plot estimated survivor functions.

Plot the estimated survivor functions for the two different types of light bulbs.

```
figure()  
[f,x,flow,fup] = ecdf(fluo(:,1), 'censoring', fluo(:,2), ...  
                     'function', 'survivor');  
ax1 = stairs(x,f);  
hold on  
stairs(x,flow, ':')  
stairs(x,fup, ':')  
[f,x,flow,fup] = ecdf(insc(:,1), 'censoring', insc(:,2), ...  
                     'function', 'survivor');  
ax2 = stairs(x,f, 'color', 'r');  
stairs(x,flow, ':r')  
stairs(x,fup, ':r')  
legend([ax1,ax2], {'Fluorescent', 'Incandescent'})  
xlabel('Lifetime (hours)')  
ylabel('Survival probability')
```



You can see that the survival probability of incandescent light bulbs is much smaller than that of fluorescent light bulbs.

### Step 3. Fit Burr Type XII distribution.

Fit Burr distribution to the lifetime data of fluorescent and incandescent type bulbs.

```
pd = fitdist(fluc(:,1), 'burr', 'Censoring', fluc(:,2))
```

```
pd =
```

```
BurrDistribution
```

```
Burr distribution
  alpha = 29143.5      [0.903899, 9.39642e+08]
    c = 3.44582      [2.13013, 5.57417]
    k = 33.704      [8.10669e-14, 1.40126e+16]

pd2 = fitdist(insc(:,1), 'burr', 'Censoring', insc(:,2))

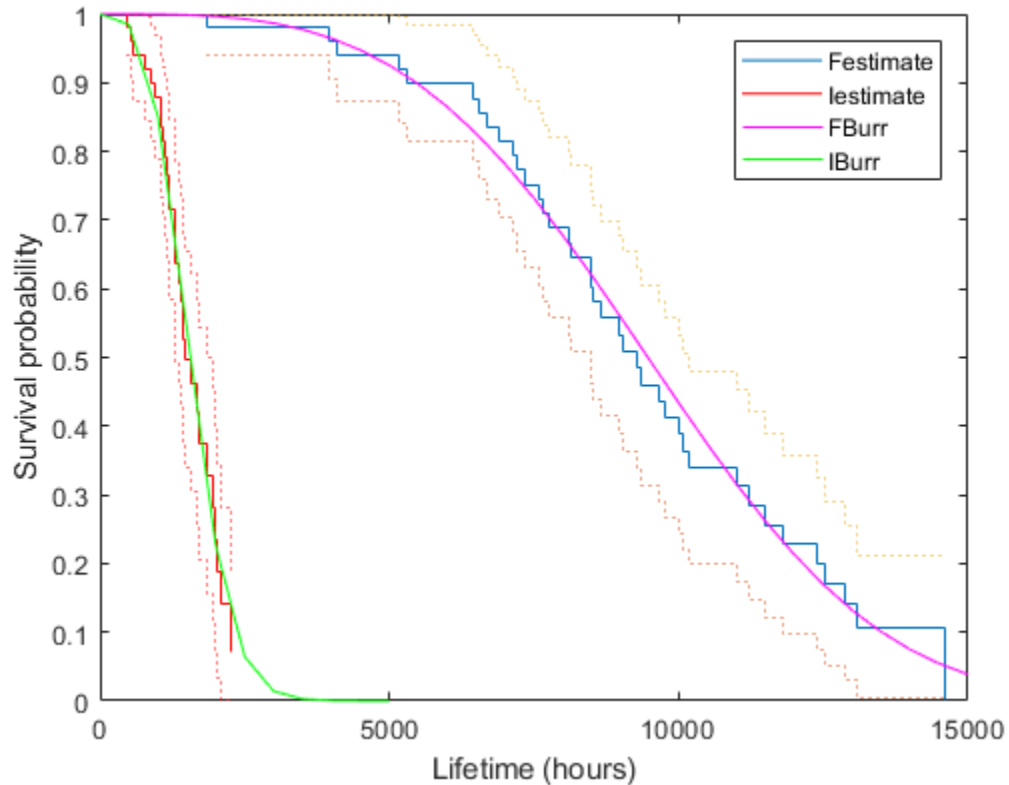
pd2 =

BurrDistribution

Burr distribution
  alpha = 2650.76      [430.773, 16311.4]
    c = 3.41898      [2.16794, 5.39197]
    k = 4.5891      [0.0307809, 684.185]

Superimpose Burr type XII survivor functions.

ax3 = plot(0:500:15000, 1-cdf('burr', 0:500:15000, 29143.5, ...
    3.44582, 33.704), 'm');
ax4 = plot(0:500:5000, 1-cdf('burr', 0:500:5000, 2650.76, ...
    3.41898, 4.5891), 'g');
legend([ax1; ax2; ax3; ax4], 'Festimate', 'Iestimate', 'FBurr', 'IBurr')
```



Burr distribution provides a good fit for the lifetime of light bulbs in this example.

#### Step 4. Fit a Cox proportional hazards model.

Fit a Cox proportional hazards regression where the type of the bulb is the explanatory variable.

```
[b,logl,H,stats] = coxphfit(lightbulb(:,2),lightbulb(:,1),...
'Censoring',lightbulb(:,3));
stats
```

```
stats =
```

```
struct with fields:

  covb: 1.0757
  beta: 4.7262
  se: 1.0372
  z: 4.5568
  p: 5.1936e-06
  csres: [100x1 double]
  devres: [100x1 double]
  martres: [100x1 double]
  schres: [100x1 double]
  sschres: [100x1 double]
  scores: [100x1 double]
  sscores: [100x1 double]
```

The  $P$ -value,  $p$ , indicates that the type of light bulb is statistically significant. The estimate of the hazard ratio is  $\exp(b) = 112.8646$ . This means that the hazard for the incandescent bulbs is 112.86 times the hazard for the fluorescent bulbs.

## See Also

`coxphfit` | `ecdf` | `ksdensity`

## Related Examples

- “Hazard and Survivor Functions for Different Groups” on page 14-17
- “Cox Proportional Hazards Model for Censored Data” on page 14-35
- “Cox Proportional Hazards Model with Time-Dependent Covariates” on page 14-41

## More About

- “What Is Survival Analysis?” on page 14-2
- “Kaplan-Meier Method” on page 14-10
- “Cox Proportional Hazards Model” on page 14-29



## Cox Proportional Hazards Model

### In this section...

“Introduction” on page 14-29

“Hazard Ratio” on page 14-29

“Extension of Cox Proportional Hazards Model” on page 14-30

“Partial Likelihood Function” on page 14-31

“Partial Likelihood Function for Tied Events” on page 14-32

“Frequency or Weights of Observations” on page 14-33

### Introduction

Cox proportional hazards regression is a semiparametric method for adjusting survival rate estimates to quantify the effect of predictor variables. The method represents the effects of explanatory variables as a multiplier of a common baseline hazard function,  $h_0(t)$ . The hazard function is the nonparametric part of the Cox proportional hazards regression function, whereas the impact of the predictor variables is a loglinear regression. For a baseline relative to 0, this model corresponds to

$$h(X_i, t) = h_0(t) \exp \left[ \sum_{j=1}^p x_{ij} b_j \right],$$

where  $X_i = (x_{i1}, x_{i2}, \dots, x_{ip})$  is the predictor variable for the  $i$ th subject,  $h(X_i, t)$  is the hazard rate at time  $t$  for  $X_i$ , and  $h_0(t)$  is the baseline hazard rate function.

### Hazard Ratio

The Cox proportional hazards model relates the hazard rate for individuals or items at the value  $X_i$ , to the hazard rate for individuals or items at the baseline value. It produces an estimate for the hazard ratio:

$$HR(X_i) = \frac{h(X_i, t)}{h_0(t)} = \exp \left[ \sum_{j=1}^p x_{ij} b_j \right].$$

The model is based on the assumption that the baseline hazard function depends on time,  $t$ , but the predictor variables do not. This assumption is also called the proportional

hazards assumption, which states that the hazard ratio does not change over time for any individual.

The hazard ratio represents the relative risk of instant failure for individuals or items having the predictive variable value  $X_i$  compared to the ones having the baseline values. For example, if the predictive variable is smoking status, where nonsmoking is the baseline category, the hazard ratio shows the relative instant failure rate of smokers compared to the baseline category, that is, nonsmokers. For a baseline relative to  $X^*$  and the predictor variable value  $X_i$ , the hazard ratio is

$$HR(X_i) = \frac{h(X_i, t)}{h(X^*, t)} = \exp \left[ \sum_{j=1}^p (x_{ij} - x_j^*) b_j \right].$$

For example, if the baseline is the mean values of the predictor variables ( $\text{mean}(X)$ ), then the hazard ratio becomes

$$HR(X_i) = \frac{h(X_i, t)}{h(\bar{X}, t)} = \exp \left[ \sum_{j=1}^p (x_{ij} - \bar{x}_j) b_j \right].$$

Hazard rates are related to survival rates, such that the survival rate at time  $t$  for an individual with the explanatory variable value  $X_i$  is

$$S_{X_i}(t) = S_0(t)^{HR(X_i)},$$

where  $S_0(t)$  is the survivor function with the baseline hazard rate function  $h_0(t)$ , and  $HR(X_i)$  is the hazard ratio of the predictor variable value  $X_i$  relative to the baseline value.

## Extension of Cox Proportional Hazards Model

When you have variables that do not satisfy the proportional hazards (PH) assumption, you can consider using two extensions of Cox proportional hazards model: the stratified Cox model and the Cox model with time-dependent variables.

If the variables that do not satisfy the PH assumption are categorizable, use the stratified Cox model:

$$h_s(X_i, t) = h_{0s}(t) \exp \left[ \sum_{j=1}^p x_{ij} b_j \right],$$

where the subscript  $s$  indicates the  $s$ th stratum. The stratified Cox model has a different baseline hazard rate function for each stratum but shares coefficients. Therefore, it has

the same hazard ratio across all strata if the predictor variable values are the same. You can include stratification variables in `coxphfit` by using the name-value pair `'Strata'`.

If the variables that do not satisfy the PH assumption are time-dependent variables, use the Cox model with time-dependent variables:

$$h(X_i, t) = h_0(t) \exp \left[ \sum_{j=1}^{p_1} x_{ij} b_j + \sum_{k=1}^{p_2} x_{ik}(t) c_k \right],$$

where  $x_{ij}$  is an element of a time-independent predictor and  $x_{ik}(t)$  is an element of a time-dependent predictor. For an example of how to include time-dependent variables in `coxphfit`, see “Cox Proportional Hazards Model with Time-Dependent Covariates” on page 14-41.

## Partial Likelihood Function

A point estimate of the effect of each explanatory variable, that is, the estimated hazard ratio for the effect of each explanatory variable is  $\exp(b)$ , given all other variables are held constant, where  $b$  is the coefficient estimate for that variable. The coefficient estimates are found by maximizing the partial likelihood function of the model. The partial likelihood function for the proportional hazards regression model is based on the observed order of events. It is the product of partial likelihoods of failures estimated for

each failure time. If there are  $n$  failures at  $n$  distinct failure times,  $t_1 < t_2 < \dots < t_n$ , then the partial likelihood is

$$L = \left[ \frac{h(X_1, t_1)}{\sum_{j=1}^n h(X_j, t_j)} \right] \times \left[ \frac{h(X_2, t_2)}{\sum_{j=2}^n h(X_j, t_j)} \right] \times \dots \times \left[ \frac{h(X_n, t_n)}{h(X_n, t_n)} \right] = \prod_{i=1}^n \frac{h(X_i, t_i)}{\sum_{j=i}^n h(X_j, t_j)}$$

You can rewrite the partial likelihood by using a risk set  $R_i$ :

$$L = \prod_{i=1}^n \frac{h(X_i, t_i)}{\sum_{j \in R_i} h(X_j, t_j)},$$

where  $R_i$  represents the index set of subjects who are under study but do not experience the event until the  $i$ th failure time.

You can use a likelihood ratio test to assess the significance of adding a term or terms in a model. Consider the two models where the first model has  $p$  predictive variables and

the second model has  $p + r$  predictive variables. Then, comparing the two models,  $-2 \ln(L_1/L_2)$  has a chi-square distribution with  $r$  degrees of freedom (the number of terms being tested).

## Partial Likelihood Function for Tied Events

When you have tied events, `coxphfit` approximates the partial likelihood of the model by either Breslow's method (default) or Efron's method, instead of computing the exact partial likelihood. Computing the exact partial likelihood requires a large amount of computation, which involves an entire permutation of the risk sets for the tied event times.

The simplest approximation method is Breslow's method. This method uses the same denominator for each tied set.

$$L = \prod_{i=1}^d \prod_{j \in D_i} \frac{h(X_j, t_j)}{\sum_{k \in R_i} h(X_k, t_k)},$$

where  $d$  is the number of distinct event times, and  $D_i$  is the index set of all subjects whose event time is equal to the  $i$ th event time.

Efron's method is more accurate than Breslow's method, yet simple. This method adjusts the denominator of the tied events as follows:

$$L = \prod_{i=1}^d \prod_{j \in D_i} \frac{h(X_j, t_j)}{\sum_{k \in R_i} h(X_k, t_k) - \frac{j-1}{d_i} \sum_{k \in D_i} h(X_k, t_k)},$$

where  $d_i$  is the number of indexes in  $D_i$ .

For an example, assume that the first two events are tied, that is,  $t_1 = t_2$  and

$t_2 < t_3 < \dots < t_n$ . In Breslow's method, the denominators of the first two terms are the same:

$$L = \left[ \frac{h(X_1, t_1)}{\sum_{j=1}^n h(X_j, t_j)} \right] \times \left[ \frac{h(X_2, t_2)}{\sum_{j=1}^n h(X_j, t_j)} \right] \times \left[ \frac{h(X_3, t_3)}{\sum_{j=3}^n h(X_j, t_j)} \right] \times \left[ \frac{h(X_4, t_4)}{\sum_{j=4}^n h(X_j, t_j)} \right] \times \dots \times \left[ \frac{h(X_n, t_n)}{h(X_n, t_n)} \right].$$

Efron's method adjusts the denominator of the second term:

$$L = \left[ \frac{h(X_1, t_1)}{\sum_{j=1}^n h(X_j, t_j)} \right] \times \left[ \frac{h(X_2, t_2)}{0.5h(X_1, t_1) + 0.5h(X_2, t_2) + \sum_{j=3}^n h(X_j, t_j)} \right] \times \left[ \frac{h(X_3, t_3)}{\sum_{j=3}^n h(X_j, t_j)} \right] \times \left[ \frac{h(X_4, t_4)}{\sum_{j=4}^n h(X_j, t_j)} \right]$$

You can specify an approximation method by using the name-value pair 'Ties' in `coxphfit`.

## Frequency or Weights of Observations

The Cox proportional hazards model can incorporate with the frequency or weights of observations. Let  $w_i$  be the weight of the  $i$ th observation. Then, the partial likelihoods of the Cox model with weights become as follows:

- Partial likelihood with weights

$$L = \prod_{i=1}^n \frac{w_i h(X_i, t_i)}{\sum_{j \in R_i} w_j h(X_j, t_j)}$$

- Partial likelihood with weights and Breslow's method

$$L = \prod_{i=1}^d \prod_{j \in D_i} \frac{w_j h(X_j, t_j)}{\left[ \sum_{k \in R_i} w_k h(X_k, t_k) \right]^{\frac{1}{d_i} \sum_{j \in D_i} w_j}}$$

- Partial likelihood with weights and Efron's method

$$L = \prod_{i=1}^d \prod_{j \in D_i} \frac{w_j h(X_j, t_j)}{\left[ \sum_{k \in R_i} w_k h(X_k, t_k) - \frac{j-1}{d_i} \sum_{k \in D_i} w_k h(X_k, t_k) \right]^{\frac{1}{d_i} \sum_{j \in D_i} w_j}}$$

You can specify the frequency or weights of observations by using the name-value pair 'Frequency' in `coxphfit`.

## References

- [1] Cox, D. R., and D. Oakes. *Analysis of Survival Data*. London: Chapman & Hall, 1984.
- [2] Lawless, J. F. *Statistical Models and Methods for Lifetime Data*. Hoboken, NJ: Wiley-Interscience, 2002.
- [3] Kleinbaum, D. G., and M. Klein. *Survival Analysis*. Statistics for Biology and Health. 2nd edition. Springer, 2005.
- [4] Klein, J. P., and M. L. Moeschberger. *Survival Analysis*. Statistics for Biology and Health. 2nd edition. Springer, 2003.

## See Also

`coxphfit` | `ecdf` | `ksdensity`

## Related Examples

- “Hazard and Survivor Functions for Different Groups” on page 14-17
- “Survivor Functions for Two Groups” on page 14-24
- “Cox Proportional Hazards Model for Censored Data” on page 14-35
- “Cox Proportional Hazards Model with Time-Dependent Covariates” on page 14-41

## More About

- “What Is Survival Analysis?” on page 14-2
- “Kaplan-Meier Method” on page 14-10

## Cox Proportional Hazards Model for Censored Data

This example shows how to construct a Cox proportional hazards model, and assess the significance of the predictor variables.

### Step 1. Load sample data.

Load the sample data.

```
load(fullfile(matlabroot, 'examples', 'stats', 'readmissiontimes.mat'));
```

The response variable is `Readmission Time`, which shows the readmission times for 100 patients. The predictor variables are `Age`, `Sex`, `Weight`, and the smoking status of each patient, `Smoker`. 1 indicates the patient is a smoker, and 0 indicates that the patient does not smoke. The column vector `Censored` has the censorship information for each patient, where 1 indicates censored data, and 0 indicates the exact readmission times are observed. This is simulated data.

### Step 2. Fit Cox proportional hazards function.

Fit a Cox proportional hazard function with the variable `Sex` as the predictor variable, taking the censoring into account.

```
X = Sex;
[b, logl, H, stats] = coxphfit(X, ReadmissionTime, 'censoring', Censored);
```

Assess the statistical significance of the term `Sex`.

```
stats

stats =

    struct with fields:

        covb: 0.1016
        beta: -1.7642
         se: 0.3188
         z: -5.5335
         p: 3.1392e-08
        csres: [100x1 double]
        devres: [100x1 double]
        martres: [100x1 double]
```

```
schres: [100x1 double]
sschres: [100x1 double]
scores: [100x1 double]
sscores: [100x1 double]
```

The *P*-value, *p*, indicates that the term Sex is statistically significant.

Save the loglikelihood value with a different name. You will use this to assess the significance of the extended models.

```
loglSex = logl
```

```
loglSex =
-262.1365
```

### Step 3. Add Age and Weight to the model.

Fit a Cox proportional hazards model with the variables Sex, Age, and Weight.

```
X = [Sex Age Weight];
[b,logl,H,stats] = coxphfit(X,ReadmissionTime,'censoring',Censored);
```

Assess the significance of the terms.

```
stats.beta
```

```
ans =
-0.5441
0.0143
0.0250
```

```
stats.p
```

```
ans =
0.4953
0.3842
```



```
0.0960
```

None of the terms, adjusted for others, is statistically significant.

Assess the significance of the terms using the log likelihood ratio. You can assess the significance of the new model using the likelihood ratio statistic. First find the difference between the log-likelihood statistic of the model without the terms Age and Weight and the log-likelihood of the model with Sex, Age, and Weight.

```
-2*[loglSex - logl]
```

```
ans =
```

```
3.6705
```

Now, compute the *P*-value for the likelihood ratio statistic. The likelihood ratio statistic has a Chi-square distribution with a degrees of freedom equal to the number of predictor variables being assessed. In this case, the degrees of freedom is 2.

```
p = 1 - cdf('chi2',3.6705,2)
```

```
p =
```

```
0.1596
```

The *P*-value of 0.1596 indicates that the terms Age and Weight are not statistically significant, given the term Sex in the model.

#### Step 4. Add Smoker to the model.

Fit a Cox proportional hazards model with the variables Sex and Smoker.

```
X = [Sex Smoker];
[b,logl,H,stats] = coxphfit(X,ReadmissionTime,...
'censoring',Censored);
```

Assess the significance of the terms in the model.

```
stats.p
```

```
ans =  
  
0.0000  
0.0148
```

Compare this model to the first model where Sex is the only term.

```
-2*[loglSex - logl]
```

```
ans =  
  
5.5789
```

Compute the  $P$ -value for the likelihood ratio statistic. The likelihood ratio statistic has a Chi-square distribution with a degree of freedom of 1.

```
p = 1 - cdf('chi2',5.5789,1)
```

```
p =  
  
0.0182
```

The  $P$ -value of 0.0182 indicates that Sex and Smoker are statistically significant given the other is in the model. The model with Sex and Smoker is a better fit compared to the model with only Sex.

Request the coefficient estimates.

```
stats.beta  
  
ans =  
  
-1.7165  
0.6338
```

The default baseline is the mean of  $X$ , so the final model for the hazard ratio is

$$HR = \frac{h_X(t)}{h_{\bar{X}}(t)} = \exp [\beta_s (X_s - \bar{X}_s) + \beta_\alpha (X_\alpha - \bar{X}_\alpha)].$$

Fit a Cox ph model with a baseline of 0.

```
X = [Sex Smoker];
[b,logl,H,stats] = coxphfit(X,ReadmissionTime,...
'censoring',Censored,'baseline',0);
```

The model for the hazard ratio is

$$HR = \frac{h_X(t)}{h_0(t)} = \exp [\beta_s X_s + \beta_\alpha X_\alpha].$$

Request the coefficient estimates.

```
stats.beta

ans =

-1.7165
0.6338
```

The coefficients are not affected, but the hazard rate differs from when the baseline is the mean of X.

## See Also

`coxphfit` | `ecdf` | `ksdensity`

## Related Examples

- “Hazard and Survivor Functions for Different Groups” on page 14-17
- “Survivor Functions for Two Groups” on page 14-24
- “Cox Proportional Hazards Model with Time-Dependent Covariates” on page 14-41

## More About

- “What Is Survival Analysis?” on page 14-2

- “Kaplan-Meier Method” on page 14-10
- “Cox Proportional Hazards Model” on page 14-29

## Cox Proportional Hazards Model with Time-Dependent Covariates

This example shows how to convert survival data to counting process form and then construct a Cox proportional hazards model with time-dependent covariates.

### Step 1. Compare standard layout and counting process form.

A Cox model with time-dependent covariates requires survival data to be in counting process form and not in standard layout. To see the difference between survival data in standard layout and in counting process form, load the following sample data.

```
load(fullfile(matlabroot, 'examples', 'stats', 'simpleSurvivalData.mat'));
```

This sample data contains two tables: `relapseS` and `relapseCP`. These two tables represent the same simple survival data in standard layout and in counting process form, respectively.

Display the data in standard layout.

```
relapseS
```

```
relapseS =
```

```
2x5 table
```

ID	Time	Censoring	Age	StopTreatment
1	5	0	20	NaN
2	20	1	30	12

This data represents two patients whose treatment status changes over time. Patient 1 was not taking treatment for the interval from week 0 to 5 and relapsed at the end of the interval. Patient 2 was taking treatment for the interval from week 0 to 12, but not for the interval from week 12 to 20. Patient 2 did not relapse and left the study after week 20.

Now display the same data in counting process form.

```
relapseCP
```

```
relapseCP =
```

```
3x6 table
```

ID	tStart	tStop	Censoring	Age	TreatmentStatus
1	0	5	0	20	0
2	0	12	1	30	1
2	12	20	1	30	0

In counting process form, each row represents the risk interval  $(tStart, tStop]$  instead of a single value of an event time. `Censoring` is 0 if the event is observed at the end of the risk interval, and 1 if it is not. `TreatmentStatus` corresponds to a time-dependent covariate, which represents the same information with `StopTreatment` in standard layout. Note that a Cox model assumes time-dependent covariates to be constant in each risk interval.

### Step 2. Load sample data.

Next, load sample data to convert.

```
load(fullfile(matlabroot, 'examples', 'stats', 'survivaldatacp.mat'));
```

This sample data contains a table `labS`, which is simulated survival data including repeated measurement for each patient in standard layout.

Display the simulated survival data in standard layout.

```
labS
```

```
labS =
```

```
6x7 table
```

ID	Time	Censoring	Sex	Lab_0	Lab_50	Lab_100
1	46	0	1	0.3	NaN	NaN
2	138	1	0	0.2	0.23	0.39
3	94	0	1	0.18	0.22	NaN

4	50	0	0	0.21	0.2	NaN
5	106	0	0	0.25	0.21	0.42
6	98	0	0	0.21	0.22	NaN

In standard layout, each row of the table shows information for one patient.

- ID indicates the ID of a patient. You do not include ID as an input of a Cox model. Include ID in a data set to confirm that the data set is correctly converted to counting process form.
- Time represents time to event in days, which corresponds to a response variable.
- Censoring has the censorship information for each patient, where 1 indicates censored data and 0 indicates that the exact time to event is observed at the end of the observation period.
- Sex is a time-independent predictor where 1 indicates female, and 0 indicates male.
- Lab\_0, Lab\_50, and Lab\_100 represent three consecutive laboratory results measured at day 0, 50, and 100, which correspond to a time-dependent predictor.

### Step 3. Convert survival data to counting process form.

To convert the survival data `labS` to counting process form, execute the code below. This code converts `Time` to a risk interval (`tStart`, `tStop`] and combines three vectors of the time-dependent predictor, `Lab_0`, `Lab_50`, and `Lab_100`, into one vector, `Lab`.

```
mTime = [0 50 100]; % Measurement time
threeLabs = [labS.Lab_0 labS.Lab_50 labS.Lab_100];
nLabMeasure = sum(sum(~isnan(threeLabs))); % Number of lab measurements
data = zeros(nLabMeasure,6); % One row for each observation
oID = 0; % Observation ID
for i = 1 : size(labS,1)
    idx = find(mTime <= labS.Time(i));
    for j = 1 : length(idx)-1
        oID = oID + 1;
        data(oID,:) = [labS.ID(i) mTime(j:j+1) 1 labS.Sex(i) threeLabs(i,j)];
    end
    oID = oID + 1;
    data(oID,:) = [labS.ID(i) mTime(length(idx)) labS.Time(i) ...
        labS.Censoring(i) labS.Sex(i) threeLabs(i,length(idx))];
end
labCP = table(data(:,1),data(:,2),data(:,3),data(:,4),data(:,5),data(:,6), ...
    'VariableNames', {'ID','tStart','tStop','Censoring','Sex','Lab'});
```

Display the survival data in counting process form.

```
labCP
```

```
labCP =
```

```
13x6 table
```

ID	tStart	tStop	Censoring	Sex	Lab
1	0	46	0	1	0.3
2	0	50	1	0	0.2
2	50	100	1	0	0.23
2	100	138	1	0	0.39
3	0	50	1	1	0.18
3	50	94	0	1	0.22
4	0	50	1	0	0.21
4	50	50	0	0	0.2
5	0	50	1	0	0.25
5	50	100	1	0	0.21
5	100	106	0	0	0.42
6	0	50	1	0	0.21
6	50	98	0	0	0.22

In counting process form, each row of table `labCP` shows information of one observation corresponding to one risk interval. Note that a Cox model assumes `Lab` to be constant in the risk interval  $(tStart, tStop]$ . The value in `Censoring` is 0 if an event is observed at the end of the risk interval, and 1 if an event is not observed.

For example, patient 3 has two laboratory measurements at day 0 and 50, so there are two rows of data for patient 3 in counting process form. A Cox model assumes the lab results 0.18 and 0.22 to be constant in the interval  $(0,50]$  and  $(50,94]$ , respectively. `Censoring` is 1 in  $(0,50]$  and 0 in  $(50,94]$  because the exact event time of patient 3 is observed at day 94.

#### Step 4. Adjust zero-length risk interval.

Find a patient who has a zero-length risk interval.

```
idxInvalid = labCP.ID(find(labCP.tStart == labCP.tStop))
```



```
idxInvalid =
```

```
4
```

Review the data for patient 4.

```
labCP(find(labCP.ID==idxInvalid),:)
```

```
ans =
```

```
2x6 table
```

ID	tStart	tStop	Censoring	Sex	Lab
4	0	50	1	0	0.21
4	50	50	0	0	0.2

The time to event of patient 4 coincides with the measurement day 50. However, (50,50] is an invalid risk interval for a Cox model because the model does not accept a zero length interval. Adjust the risk interval to be valid. You can choose any value less than the time unit as an adjustment amount. The choice of an adjustment amount is arbitrary, and it does not change the result.

```
idxAdjust = find(labCP.ID==idxInvalid);
labCP.tStop(idxAdjust(1)) = labCP.tStop(idxAdjust(1))-0.5;
labCP.tStart(idxAdjust(2)) = labCP.tStart(idxAdjust(2))-0.5;
labCP(idxAdjust,:)
```

```
ans =
```

```
2x6 table
```

ID	tStart	tStop	Censoring	Sex	Lab
4	0	49.5	1	0	0.21
4	49.5	50	0	0	0.2

**Step 5. Construct a Cox proportional hazards model.**

Fit a Cox proportional hazards model with the time-independent variable `Sex` and time-dependent variable `Lab`.

```
X = [labCP.Sex labCP.Lab];  
T = [labCP.tStart labCP.tStop];  
b = coxphfit(X,T,'Censoring',labCP.Censoring,'Baseline',0)
```

```
b =  
  
    2.0054  
   29.7530
```

For details on how to assess a Cox proportional hazards model, see “Cox Proportional Hazards Model for Censored Data” on page 14-35.

**See Also**

`coxphfit`

**Related Examples**

- “Hazard and Survivor Functions for Different Groups” on page 14-17
- “Survivor Functions for Two Groups” on page 14-24
- “Cox Proportional Hazards Model for Censored Data” on page 14-35

**More About**

- “What Is Survival Analysis?” on page 14-2
- “Kaplan-Meier Method” on page 14-10
- “Cox Proportional Hazards Model” on page 14-29

# Multivariate Methods

---

- “Introduction to Multivariate Methods” on page 15-3
- “Multivariate Linear Regression” on page 15-4
- “Estimation of Multivariate Regression Models” on page 15-7
- “Set Up Multivariate Regression Problems” on page 15-16
- “Multivariate General Linear Model” on page 15-29
- “Fixed Effects Panel Model with Concurrent Correlation” on page 15-35
- “Longitudinal Analysis” on page 15-43
- “Multidimensional Scaling” on page 15-50
- “Nonclassical and Nonmetric Multidimensional Scaling” on page 15-51
- “Classical Multidimensional Scaling” on page 15-57
- “Example: Multidimensional Scaling” on page 15-60
- “Procrustes Analysis” on page 15-63
- “Compare Handwritten Shapes Using Procrustes Analysis” on page 15-65
- “Feature Selection” on page 15-73
- “Select Subset of Features with Comparative Predictive Power” on page 15-75
- “Feature Transformation” on page 15-78
- “Nonnegative Matrix Factorization” on page 15-79
- “Perform Nonnegative Matrix Factorization” on page 15-80
- “Principal Component Analysis (PCA)” on page 15-83
- “Analyze Quality of Life in U.S. Cities Using PCA” on page 15-85
- “Factor Analysis” on page 15-97
- “Analyze Stock Prices Using Factor Analysis” on page 15-98
- “Robust Feature Selection Using NCA for Regression” on page 15-107
- “Neighborhood Component Analysis (NCA) Feature Selection” on page 15-126
- “t-SNE” on page 15-132
- “t-SNE Output Function” on page 15-140

- “Visualize High-Dimensional Data Using t-SNE” on page 15-144
- “tsne Settings” on page 15-149
- “Feature Extraction” on page 15-163
- “Feature Extraction Workflow” on page 15-169
- “Extract Mixed Signals” on page 15-205

## Introduction to Multivariate Methods

Large, high-dimensional data sets are common in the modern era of computer-based instrumentation and electronic data storage. High-dimensional data present many challenges for statistical visualization, analysis, and modeling.

Data visualization, of course, is impossible beyond a few dimensions. As a result, pattern recognition, data preprocessing, and model selection must rely heavily on numerical methods.

A fundamental challenge in high-dimensional data analysis is the so-called curse of dimensionality. Observations in a high-dimensional space are necessarily sparser and less representative than those in a low-dimensional space. In higher dimensions, data over-represent the edges of a sampling distribution, because regions of higher-dimensional space contain the majority of their volume near the surface. (A  $d$ -dimensional spherical shell has a volume, relative to the total volume of the sphere, that approaches 1 as  $d$  approaches infinity.) In high dimensions, typical data points at the interior of a distribution are sampled less frequently.

Often, many of the dimensions in a data set—the measured features—are not useful in producing a model. Features may be irrelevant or redundant. Regression and classification algorithms may require large amounts of storage and computation time to process raw data, and even if the algorithms are successful the resulting models may contain an incomprehensible number of terms.

Because of these challenges, multivariate statistical methods often begin with some type of dimension reduction, in which data are approximated by points in a lower-dimensional space. Dimension reduction is the goal of the methods presented in this chapter. Dimension reduction often leads to simpler models and fewer measured variables, with consequent benefits when measurements are expensive and visualization is important.

## Multivariate Linear Regression

### In this section...

“Multivariate Linear Regression Model” on page 15-4

“Solving Multivariate Regression Problems” on page 15-4

### Multivariate Linear Regression Model

The multivariate linear regression model expresses a  $d$ -dimensional continuous response vector as a linear combination of predictor terms plus a vector of error terms with a

multivariate normal distribution. Let  $\mathbf{y}_i = (y_{i1}, \dots, y_{id})'$  denote the response vector for observation  $i$ ,  $i = 1, \dots, n$ . In the most general case, given the  $d$ -by- $K$  design matrix  $\mathbf{X}_i$  and the  $K$ -by-1 vector of coefficients  $\boldsymbol{\beta}$ , the multivariate linear regression model is

$$\mathbf{y}_i = \mathbf{X}_i \boldsymbol{\beta} + \boldsymbol{\varepsilon}_i,$$

where the  $d$ -dimensional vector of error terms follows a multivariate normal distribution,  $\boldsymbol{\varepsilon}_i \sim MVN_d(\mathbf{0}, \boldsymbol{\Sigma})$ .

The model assumes independence between observations, meaning the error variance-covariance matrix for the  $n$  stacked  $d$ -dimensional response vectors is

$$\mathbf{I}_n \otimes \boldsymbol{\Sigma} = \begin{pmatrix} \boldsymbol{\Sigma} & & \mathbf{0} \\ & \ddots & \\ \mathbf{0} & & \boldsymbol{\Sigma} \end{pmatrix}.$$

If  $\mathbf{Y}$  denotes the  $nd$ -by-1 vector of stacked  $d$ -dimensional responses, and  $\mathbf{X}$  denotes the  $nd$ -by- $K$  matrix of stacked design matrices, then the distribution of the response vector is  $\mathbf{y} \sim MVN_{nd}(\mathbf{X}\boldsymbol{\beta}, \mathbf{I}_n \otimes \boldsymbol{\Sigma})$ .

### Solving Multivariate Regression Problems

To fit multivariate linear regression models of the form

$$\mathbf{y}_i = \mathbf{X}_i \boldsymbol{\beta} + \boldsymbol{\varepsilon}_i, \quad \boldsymbol{\varepsilon}_i \sim MVN_d(\mathbf{0}, \boldsymbol{\Sigma})$$

in Statistics and Machine Learning Toolbox, use `mvregress`. This function fits multivariate regression models with a diagonal (heteroscedastic) or unstructured

(heteroscedastic and correlated) error variance-covariance matrix,  $\Sigma$ , using least squares or maximum likelihood estimation.

Many variations of multivariate regression might not initially appear to be of the form supported by `mvregress`, such as:

- Multivariate general linear model
- Multivariate analysis of variance (MANOVA)
- Longitudinal analysis
- Panel data analysis
- Seemingly unrelated regression (SUR)
- Vector autoregressive (VAR) model

In many cases, you can frame these problems in the form used by `mvregress` (but `mvregress` does not support parameterized error variance-covariance matrices). For the special case of one-way MANOVA, you can alternatively use `manova1`. Econometrics Toolbox™ has functions for VAR estimation.

---

**Note** The multivariate linear regression model is distinct from the multiple linear regression model, which models a *univariate* continuous response as a linear combination of exogenous terms plus an independent and identically distributed error term. To fit a multiple linear regression model, use `LinearModel.fit`.

---

## See Also

`LinearModel.fit` | `manova1` | `mvregress` | `mvregresslike`

## Related Examples

- “Set Up Multivariate Regression Problems” on page 15-16
- “Multivariate General Linear Model” on page 15-29
- “Fixed Effects Panel Model with Concurrent Correlation” on page 15-35
- “Longitudinal Analysis” on page 15-43

## More About

- “Estimation of Multivariate Regression Models” on page 15-7



## Estimation of Multivariate Regression Models

### In this section...

- “Least Squares Estimation” on page 15-7
- “Maximum Likelihood Estimation” on page 15-10
- “Missing Response Data” on page 15-13

### Least Squares Estimation

- “Ordinary Least Squares” on page 15-7
- “Covariance-Weighted Least Squares” on page 15-8
- “Error Covariance Estimation” on page 15-9
- “Feasible Generalized Least Squares” on page 15-9
- “Panel Corrected Standard Errors” on page 15-10

#### Ordinary Least Squares

When you fit multivariate linear regression models using `mvregress`, you can use the optional name-value pair `'algorithm'`, `'cwlsl'` to choose least squares estimation. In this case, by default, `mvregress` returns ordinary least squares (OLS) estimates using  $\Sigma = \mathbf{I}_d$ . Alternatively, if you specify a covariance matrix for weighting, you can return covariance-weighted least squares (CWLS) estimates. If you combine OLS and CWLS, you can get feasible generalized least squares (FGLS) estimates.

The OLS estimate for the coefficient vector is the vector  $\mathbf{b}$  that minimizes

$$\sum_{i=1}^n (\mathbf{y}_i - \mathbf{X}_i \mathbf{b})' (\mathbf{y}_i - \mathbf{X}_i \mathbf{b}).$$

Let  $\mathbf{y}$  denote the  $nd$ -by-1 vector of stacked  $d$ -dimensional responses, and  $\mathbf{X}$  denote the  $nd$ -by- $K$  matrix of stacked design matrices. The  $K$ -by-1 vector of OLS regression coefficient estimates is

$$\mathbf{b}_{OLS} = (\mathbf{X}'\mathbf{X})^{-1} \mathbf{X}'\mathbf{y}.$$

This is the first `mvregress` output.

Given  $\Sigma = \mathbf{I}_d$  (the `mvregress` OLS default), the variance-covariance matrix of the OLS estimates is

$$V(\mathbf{b}_{OLS}) = (\mathbf{X}'\mathbf{X})^{-1}.$$

This is the fourth `mvregress` output. The standard errors of the OLS regression coefficients are the square root of the diagonal of this variance-covariance matrix.

If your data is not scaled such that  $\Sigma = \sigma^2 \mathbf{I}_d$ , then you can multiply the `mvregress` variance-covariance matrix by the mean squared error (MSE), an unbiased estimate of  $\sigma^2$ . To compute the MSE, return the  $n$ -by- $d$  matrix of residuals,  $\mathbf{E}$  (the third `mvregress` output). Then,

$$\text{MSE} = \frac{\sum_{i=1}^n \mathbf{e}_i \mathbf{e}_i'}{n - K},$$

where  $\mathbf{e}_i = (\mathbf{y}_i - \mathbf{X}_i \boldsymbol{\beta})'$  is the  $i$ th row of  $\mathbf{E}$ .

### Covariance-Weighted Least Squares

For most multivariate problems, an identity error covariance matrix is insufficient, and leads to inefficient or biased standard error estimates. You can specify a matrix for CWLS estimation using the optional name-value pair argument `covar0`, for example, an invertible  $d$ -by- $d$  matrix named  $\mathbf{C}_0$ . Usually,  $\mathbf{C}_0$  is a diagonal matrix such that the inverse matrix  $\mathbf{C}_0^{-1}$  contains weights for each dimension to model heteroscedasticity. However,  $\mathbf{C}_0$  can also be a nondiagonal matrix that models correlation.

Given  $\mathbf{C}_0$ , the CWLS solution is the vector  $\mathbf{b}$  that minimizes

$$\sum_{i=1}^n (\mathbf{y}_i - \mathbf{X}_i \mathbf{b})' \mathbf{C}_0 (\mathbf{y}_i - \mathbf{X}_i \mathbf{b}).$$

In this case, the  $K$ -by-1 vector of CWLS regression coefficient estimates is

$$\mathbf{b}_{CWLS} = \left( \mathbf{X}' (\mathbf{I}_n \otimes \mathbf{C}_0)^{-1} \mathbf{X} \right)^{-1} \mathbf{X}' (\mathbf{I}_n \otimes \mathbf{C}_0)^{-1} \mathbf{y}.$$

This is the first `mvregress` output.

If  $\Sigma = \mathbf{C}_0$ , this is the generalized least squares (GLS) solution. The corresponding variance-covariance matrix of the CWLS estimates is

$$V(\mathbf{b}_{CWLS}) = \left( \mathbf{X}'(\mathbf{I}_n \otimes \mathbf{C}_0)^{-1} \mathbf{X} \right)^{-1}.$$

This is the fourth `mvregress` output. The standard errors of the CWLS regression coefficients are the square root of the diagonal of this variance-covariance matrix.

If you only know the error covariance matrix up to a proportion, that is,  $\Sigma = \sigma^2 \mathbf{C}_0$ , you can multiply the `mvregress` variance-covariance matrix by the MSE, as described in “Ordinary Least Squares” on page 15-7.

### Error Covariance Estimation

Regardless of which least squares method you use, the estimate for the error variance-covariance matrix is

$$\hat{\Sigma} = \begin{pmatrix} \hat{\sigma}_1^2 & \hat{\sigma}_{12} & \cdots & \hat{\sigma}_{1d} \\ \hat{\sigma}_{12} & \hat{\sigma}_2^2 & \cdots & \hat{\sigma}_{2d} \\ \vdots & \vdots & \ddots & \vdots \\ \hat{\sigma}_{1d} & \hat{\sigma}_{2d} & \cdots & \hat{\sigma}_d^2 \end{pmatrix} = \frac{\mathbf{E}'\mathbf{E}}{n},$$

where  $\mathbf{E}$  is the  $n$ -by- $d$  matrix of residuals. The  $i$ th row of  $\mathbf{E}$  is  $\mathbf{e}_i = (\mathbf{y}_i - \mathbf{X}_i \mathbf{b})'$ .

The error covariance estimate,  $\hat{\Sigma}$ , is the second `mvregress` output, and the matrix of residuals,  $\mathbf{E}$ , is the third output. If you specify the optional name-value pair 'covtype', 'diagonal', then `mvregress` returns  $\hat{\Sigma}$  with zeros in the off-diagonal entries,

$$\hat{\Sigma} = \begin{pmatrix} \hat{\sigma}_1^2 & & \mathbf{0} \\ & \ddots & \\ \mathbf{0} & & \hat{\sigma}_d^2 \end{pmatrix}.$$

### Feasible Generalized Least Squares

The generalized least squares estimate is the CWLS estimate with a known covariance matrix. That is, given  $\Sigma$  is known, the GLS solution is

$$\mathbf{b}_{GLS} = \left( \mathbf{X}'(\mathbf{I}_n \otimes \Sigma)^{-1} \mathbf{X} \right)^{-1} \mathbf{X}'(\mathbf{I}_n \otimes \Sigma)^{-1} \mathbf{y},$$

with variance-covariance matrix

$$V(\mathbf{b}_{GLS}) = \left( \mathbf{X}'(\mathbf{I}_n \otimes \Sigma)^{-1} \mathbf{X} \right)^{-1}.$$

In most cases, the error covariance is unknown. The feasible generalized least squares (FGLS) estimate uses  $\hat{\Sigma}$  in place of  $\Sigma$ . You can obtain two-step FGLS estimates as follows:

- 1 Perform OLS regression, and return an estimate  $\hat{\Sigma}$ .
- 2 Perform CWLS regression, using  $\mathbf{C}_0 = \hat{\Sigma}$ .

You can also iterate between these two steps until convergence is reached.

For some data, the OLS estimate  $\hat{\Sigma}$  is positive semidefinite, and has no unique inverse. In this case, you cannot get the FGLS estimate using `mvregress`. As an alternative, you can use `lscov`, which uses a generalized inverse to return weighted least squares solutions for positive semidefinite covariance matrices.

### Panel Corrected Standard Errors

An alternative to FGLS is to use OLS coefficient estimates (which are consistent) and make a standard error correction to improve efficiency. One such standard error adjustment—which does not require inversion of the covariance matrix—is panel corrected standard errors (PCSE) [1]. The panel corrected variance-covariance matrix for OLS estimates is

$$V_{pcse}(\mathbf{b}_{OLS}) = (\mathbf{X}'\mathbf{X})^{-1} \mathbf{X}'(\mathbf{I}_n \otimes \Sigma) \mathbf{X}(\mathbf{X}'\mathbf{X}).$$

The PCSE are the square root of the diagonal of this variance-covariance matrix. “Fixed Effects Panel Model with Concurrent Correlation” on page 15-35 illustrates PCSE computation.

### Maximum Likelihood Estimation

- “Maximum Likelihood Estimates” on page 15-11

- “Standard Errors” on page 15-11

### Maximum Likelihood Estimates

The default estimation algorithm used by `mvregress` is maximum likelihood estimation (MLE). The loglikelihood function for the multivariate linear regression model is

$$\log L(\beta, \Sigma | \mathbf{y}, \mathbf{X}) = \frac{1}{2} n d \log(2\pi) + \frac{1}{2} n \log(\det(\Sigma)) \\ + \frac{1}{2} \sum_{i=1}^n (\mathbf{y}_i - \mathbf{X}_i \beta)' \Sigma^{-1} (\mathbf{y}_i - \mathbf{X}_i \beta).$$

The MLEs for  $\beta$  and  $\Sigma$  are the values that maximize the loglikelihood objective function.

`mvregress` finds the MLEs using an iterative two-stage algorithm. At iteration  $m + 1$ , the estimates are

$$\mathbf{b}_{MLE}^{(m+1)} = \left( \mathbf{X}' (\mathbf{I}_n \otimes \Sigma^{(m)})^{-1} \mathbf{X} \right)^{-1} \mathbf{X}' (\mathbf{I}_n \otimes \Sigma^{(m)})^{-1} \mathbf{y}$$

and

$$\hat{\Sigma}^{(m+1)} = \frac{1}{n} \sum_{i=1}^n (\mathbf{y}_i - \mathbf{X}_i \mathbf{b}_{MLE}^{(m+1)}) (\mathbf{y}_i - \mathbf{X}_i \mathbf{b}_{MLE}^{(m+1)})'.$$

The algorithm terminates when the changes in the coefficient estimates and loglikelihood objective function are less than a specified tolerance, or when the specified maximum number of iterations is reached. The optional name-value pair arguments for changing these convergence criteria are `tolbeta`, `tolobj`, and `maxiter`, respectively.

### Standard Errors

The variance-covariance matrix of the MLEs is an optional `mvregress` output. By default, `mvregress` returns the variance-covariance matrix for only the regression

coefficients, but you can also get the variance-covariance matrix of  $\hat{\Sigma}$  using the optional name-value pair `'vartype', 'full'`. In this case, `mvregress` returns the variance-covariance matrix for all  $K$  regression coefficients, and  $d$  or  $d(d + 1)/2$  covariance terms (depending on whether the error covariance is diagonal or full).

By default, the variance-covariance matrix is the inverse of the observed Fisher information matrix (the 'hessian' option). You can request the expected Fisher information matrix using the optional name-value pair 'vartype', 'fisher'. Provided there is no missing response data, the observed and expected Fisher information matrices are the same. If response data is missing, the observed Fisher information accounts for the added uncertainty due to the missing values, whereas the expected Fisher information matrix does not.

The variance-covariance matrix for the regression coefficient MLEs is

$$V(\mathbf{b}_{MLE}) = \left( \mathbf{X}'(\mathbf{I}_n \otimes \Sigma)^{-1} \mathbf{X} \right)^{-1},$$

evaluated at the MLE of the error covariance matrix. This is the fourth `mvregress` output. The standard errors of the MLEs are the square root of the diagonal of this variance-covariance matrix.

For  $\hat{\Sigma}$ , let  $\theta$  denote the vector of parameters in the estimated error variance-covariance matrix. For example, if  $d = 2$ , then:

- If the estimated covariance matrix is diagonal, then  $\theta = (\sigma_1^2, \sigma_2^2)$ .
- If the estimated covariance matrix is full, then  $\theta = (\sigma_1^2, \sigma_{12}, \sigma_2^2)$ .

The Fisher information matrix for  $\theta$ ,  $I(\theta)$ , has elements

$$I(\theta)_{u,v} = \frac{1}{2} \text{tr} \left( \Sigma^{-1} \frac{\partial \Sigma}{\partial \theta_u} \Sigma^{-1} \frac{\partial \Sigma}{\partial \theta_v} \right), \quad u, v = 1, \dots, n_\theta,$$

where  $n_\theta$  is the length of  $\theta$  (either  $d$  or  $d(d+1)/2$ ). The resulting variance-covariance matrix is

$$V(\theta) = I(\theta)^{-1}.$$

When you request the full variance-covariance matrix, `mvregress` returns (as the fourth output) the block diagonal matrix

$$\begin{pmatrix} V(\mathbf{b}_{MLE}) & \mathbf{0} \\ \mathbf{0} & V(\theta) \end{pmatrix}.$$

## Missing Response Data

- “Expectation/Conditional Maximization” on page 15-13
- “Observed Information Matrix” on page 15-14

### Expectation/Conditional Maximization

If any response values are missing, indicated by `NaN`, `mvregress` uses an expectation/conditional maximization (ECM) algorithm for estimation (if enough data is available). In this case, the algorithm is iterative for both least squares and maximum likelihood estimation. During each iteration, `mvregress` imputes missing response values using their conditional expectation.

Consider organizing the data so that the joint distribution of the missing and observed responses, denoted  $\tilde{\mathbf{y}}$  and  $\mathbf{y}$  respectively, can be written as

$$\begin{pmatrix} \tilde{\mathbf{y}} \\ \mathbf{y} \end{pmatrix} \sim MVN \left\{ \begin{pmatrix} \tilde{\mathbf{X}}\boldsymbol{\beta} \\ \mathbf{X}\boldsymbol{\beta} \end{pmatrix}, \begin{pmatrix} \Sigma_{\tilde{y}} & \Sigma_{\tilde{y}y} \\ \Sigma_{y\tilde{y}} & \Sigma_y \end{pmatrix} \right\}.$$

Using properties of the multivariate normal distribution, the conditional expectation of the missing responses given the observed responses is

$$\mathbf{E}(\tilde{\mathbf{y}}|\mathbf{y}) = \tilde{\mathbf{X}}\boldsymbol{\beta} + \Sigma_{\tilde{y}y}\Sigma_y^{-1}(\mathbf{y} - \mathbf{X}\boldsymbol{\beta}).$$

Also, the variance-covariance matrix of the conditional distribution is

$$\text{COV}(\tilde{\mathbf{y}}|\mathbf{y}) = \Sigma_{\tilde{y}} - \Sigma_{\tilde{y}y}\Sigma_y^{-1}\Sigma_{y\tilde{y}}.$$

At each iteration of the ECM algorithm, `mvregress` uses the parameter values from the previous iteration to:

- Update the regression coefficients using the combined vector of observed responses and conditional expectations of missing responses.
- Update the variance-covariance matrix, adjusting for missing responses using the variance-covariance matrix of the conditional distribution.

Finally, the residuals that `mvregress` returns for missing responses are the difference between the conditional expectation and the fitted value, both evaluated at the final parameter estimates.

If you prefer to ignore any observations that have missing response values, use the name-value pair `'algorithm', 'mvm'`. Note that `mvregress` always ignores observations that have missing predictor values.

## Observed Information Matrix

By default, `mvregress` uses the observed Fisher information matrix (the 'hessian' option) to compute the variance-covariance matrix of the regression parameters. This accounts for the additional uncertainty due to missing response values.

The observed information matrix includes contributions from only the observed responses. That is, the observed Fisher information matrix for the parameters in the error variance-covariance matrix has elements

$$I(\theta)_{u,v} = \frac{1}{2} \sum_{i=1}^n \text{tr} \left( \Sigma_i^{-1} \frac{\partial \Sigma_i}{\partial \theta_u} \Sigma_i^{-1} \frac{\partial \Sigma_i}{\partial \theta_v} \right), \quad u, v = 1, \dots, n_\theta,$$

where  $\hat{\Sigma}_i$  is the subset of  $\hat{\Sigma}$  corresponding to the observed responses in  $\mathbf{y}_i$ .

For example, if  $d = 3$ , but  $y_{i2}$  is missing, then

$$\hat{\Sigma}_i = \begin{pmatrix} \hat{\sigma}_1^2 & \hat{\sigma}_{13} \\ \hat{\sigma}_{13} & \hat{\sigma}_3^2 \end{pmatrix}.$$

The observed Fisher information for the regression coefficients has similar contributions from the design and covariance matrices.

## References

- [1] Beck, N. and J. N. Katz. *What to Do (and Not to Do) with Time-Series-Cross-Section Data in Comparative Politics*. *American Political Science Review*, Vol. 89, No. 3, pp. 634–647, 1995.

## See Also

`mvregress` | `mvregresslike`

## Related Examples

- “Set Up Multivariate Regression Problems” on page 15-16
- “Multivariate General Linear Model” on page 15-29
- “Fixed Effects Panel Model with Concurrent Correlation” on page 15-35



- “Longitudinal Analysis” on page 15-43

## **More About**

- “Multivariate Linear Regression” on page 15-4

## Set Up Multivariate Regression Problems

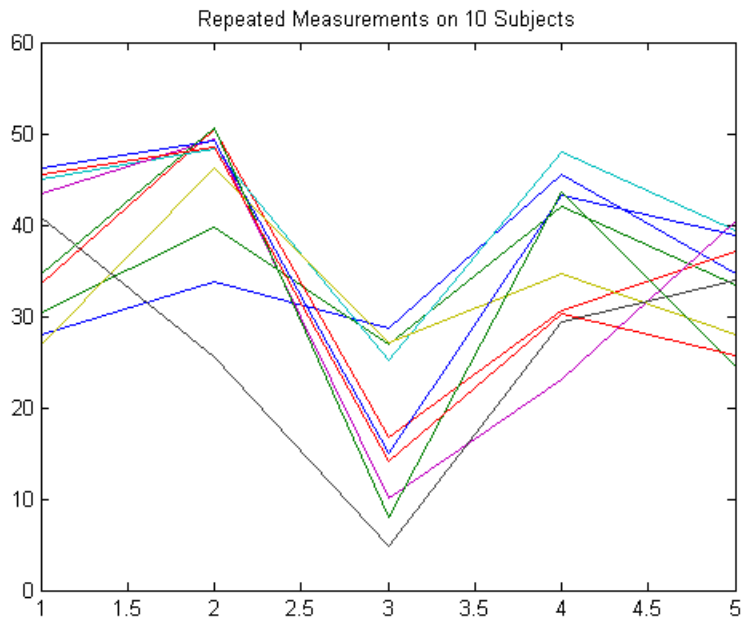
In this section...
“Response Matrix” on page 15-16
“Design Matrices” on page 15-21
“Common Multivariate Regression Problems” on page 15-22

### Response Matrix

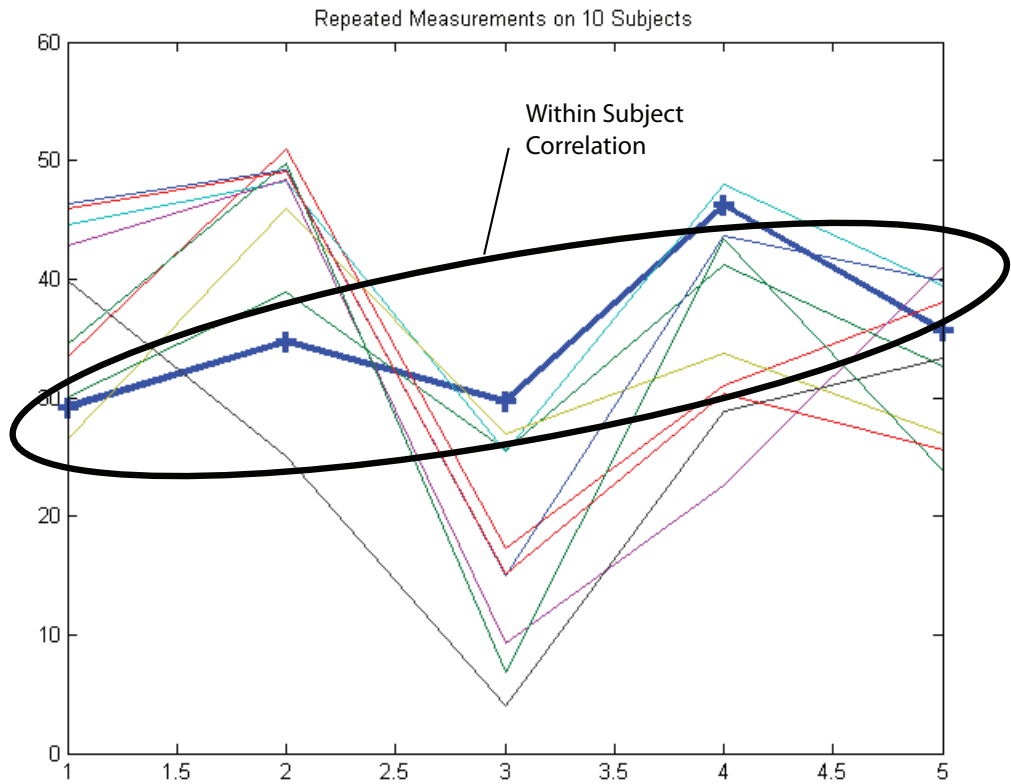
To fit a multivariate linear regression model using `mvregress`, you must set up your response matrix and design matrices in a particular way. Given properly formatted inputs, `mvregress` can handle a variety of multivariate regression problems.

`mvregress` expects the  $n$  observations of potentially correlated  $d$ -dimensional responses to be in an  $n$ -by- $d$  matrix, named  $Y$ , for example. That is, set up your responses so that the dependency structure is between observations in the same *row*. If you specify  $Y$  as a vector of length  $n$  (either a row or column vector), then `mvregress` assumes that  $d = 1$ , and treats the elements as  $n$  independent observations. It does *not* model the vector as one realization of a correlated series (such as a time series).

To illustrate how to set up a response matrix, suppose that your multivariate responses are repeated measurements made on subjects at multiple time points, as in the following figure.



Suppose that observations within a subject are correlated.

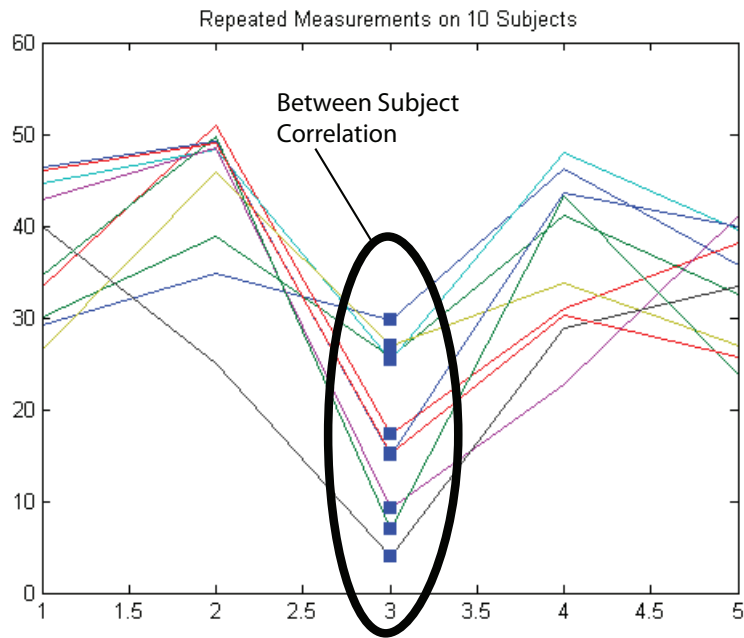


In this case, set up the response matrix  $Y$  such that each row corresponds to a subject, and each column corresponds to a time point.

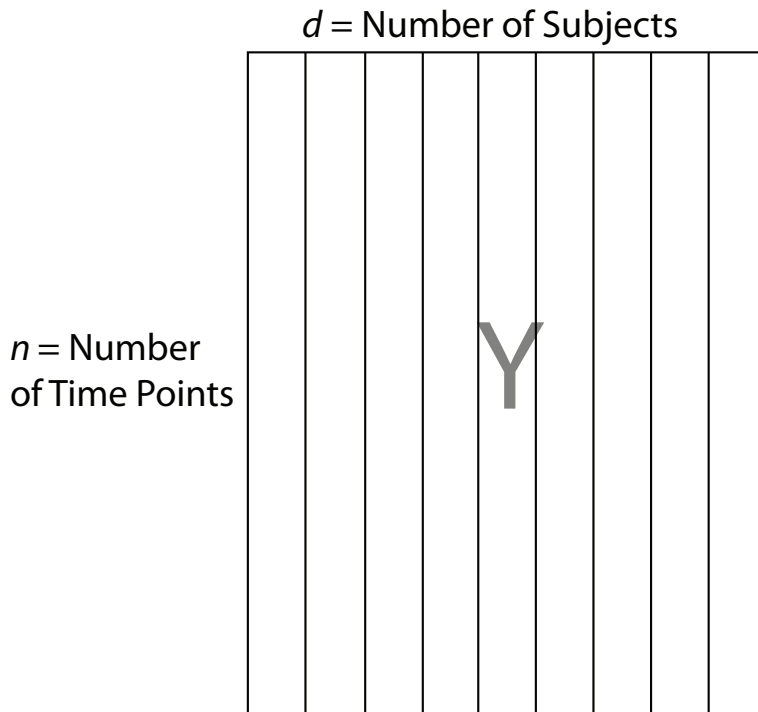
$d = \text{Number of Time Points}$ 

$n = \text{Number of Subjects}$	Y

Then again, suppose that observations made on subjects at the same time are correlated (concurrent correlation).



In this case, set up the response matrix  $Y$  such that each row corresponds to a time point, and each column corresponds to a subject.



## Design Matrices

In the multivariate linear regression model, each  $d$ -dimensional response has a corresponding design matrix. Depending on the model, the design matrix might be comprised of exogenous predictor variables, dummy variables, lagged responses, or a combination of these and other covariate terms.

- If  $d > 1$  and all  $d$  dimensions have the same design matrix, then specify one  $n$ -by- $p$  design matrix, where  $p$  is the number of predictor variables. To determine an intercept for each dimension, add a column of ones to the design matrix. In this case, `mvregress` applies the design matrix to all  $d$  dimensions.
- If  $d > 1$  and all  $d$  dimensions do not have the same design matrix, then specify the design matrices using a length- $n$  cell array of  $d$ -by- $K$  arrays, named `X`, for example.  $K$  is the total number of regression coefficients in the model. Note that the rows of the arrays in `X` correspond to the columns of the response matrix, `Y`.

$$\mathbf{X} = \left\{ \begin{array}{c} \xrightarrow{K} \\ \boxed{\mathbf{X}_1} \\ \uparrow d \end{array} , \begin{array}{c} \xrightarrow{K} \\ \boxed{\mathbf{X}_2} \\ \uparrow d \end{array} , \dots , \begin{array}{c} \xrightarrow{K} \\ \boxed{\mathbf{X}_n} \\ \uparrow d \end{array} \right\}$$

If all  $n$  observations have the same design matrix, you can specify a cell array containing one  $d$ -by- $K$  design matrix. In this case, `mvregress` applies the design matrix to all  $n$  observations. For example, this situation might arise if the predictors are functions of time, and all observations were measured at the same time points.

- In the special case that  $d = 1$ , you can specify one  $n$ -by- $K$  design matrix (not in a cell array). However, you should consider using `fitlm` to fit regression models to univariate, continuous responses.

The following sections illustrate how to set up the some common multivariate regression problems for estimation using `mvregress`.

## Common Multivariate Regression Problems

- “Multivariate General Linear Model” on page 15-22
- “Longitudinal Analysis” on page 15-24
- “Panel Analysis” on page 15-25
- “Seemingly Unrelated Regression” on page 15-26
- “Vector Autoregressive Model” on page 15-26

### Multivariate General Linear Model

The multivariate general linear model is of the form

$$\mathbf{Y}_{n \times d} = \mathbf{X}_{n \times (p+1)} \mathbf{B}_{(p+1) \times d} + \mathbf{E}_{n \times d}.$$

In expanded form,

$$\begin{bmatrix} y_{11} & y_{12} & \cdots & y_{1d} \\ y_{21} & y_{22} & \cdots & y_{2d} \\ \vdots & \vdots & \ddots & \vdots \\ y_{n1} & y_{n2} & \cdots & y_{nd} \end{bmatrix} = \begin{bmatrix} 1 & x_{11} & x_{12} & \cdots & x_{1p} \\ 1 & x_{21} & x_{22} & \cdots & x_{2p} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n1} & x_{n2} & \cdots & x_{np} \end{bmatrix} \begin{bmatrix} \beta_{01} & \beta_{02} & \cdots & \beta_{0d} \\ \beta_{11} & \beta_{12} & \cdots & \beta_{1d} \\ \vdots & \vdots & \ddots & \vdots \\ \beta_{p1} & \beta_{p2} & \cdots & \beta_{pd} \end{bmatrix} + \begin{bmatrix} \varepsilon_{11} & \varepsilon_{12} & \cdots & \varepsilon_{1d} \\ \varepsilon_{21} & \varepsilon_{22} & \cdots & \varepsilon_{2d} \\ \vdots & \vdots & \ddots & \vdots \\ \varepsilon_{n1} & \varepsilon_{n2} & \cdots & \varepsilon_{nd} \end{bmatrix}.$$




That is, each  $d$ -dimensional response has an intercept and  $p$  predictor variables, and each dimension has its own set of regression coefficients. In this form, the least squares solution is  $B = X \backslash Y$ . To estimate this model using `mvregress`, use the  $n$ -by- $d$  matrix of responses, as above.

If all  $d$  dimensions have the same design matrix, use the  $n$ -by- $(p+1)$  design matrix, as above. Adding a column of ones to the  $p$  predictor variables computes the intercept for each dimension.


If all  $d$  dimensions do not have the same design matrix, reformat the  $n$ -by- $(p+1)$  design matrix into a length- $n$  cell array of  $d$ -by- $K$  matrices. Here,  $K = (p+1)d$  for an intercept and slopes for each dimension.

For example, suppose  $n = 4$ ,  $d = 3$ , and  $p = 2$  (two predictor terms in addition to an intercept). This figure shows how to format the  $i$ th element in the cell array.

$$\begin{bmatrix} y_{11} & y_{12} & y_{13} \\ y_{21} & y_{22} & y_{23} \\ y_{31} & y_{32} & y_{33} \\ y_{41} & y_{42} & y_{43} \end{bmatrix} = \begin{bmatrix} 1 & x_{11} & x_{12} \\ 1 & x_{21} & x_{22} \\ 1 & x_{31} & x_{32} \\ 1 & x_{41} & x_{42} \end{bmatrix} \begin{bmatrix} \beta_{01} & \beta_{02} & \beta_{03} \\ \beta_{11} & \beta_{12} & \beta_{13} \\ \beta_{21} & \beta_{22} & \beta_{23} \end{bmatrix} + \begin{bmatrix} \varepsilon_{11} & \varepsilon_{12} & \varepsilon_{13} \\ \varepsilon_{21} & \varepsilon_{22} & \varepsilon_{23} \\ \varepsilon_{31} & \varepsilon_{32} & \varepsilon_{33} \\ \varepsilon_{41} & \varepsilon_{42} & \varepsilon_{43} \end{bmatrix}$$



$$\begin{bmatrix} 1 & 0 & 0 & x_{i1} & 0 & 0 & x_{i2} & 0 & 0 \\ 0 & 1 & 0 & 0 & x_{i1} & 0 & 0 & x_{i2} & 0 \\ 0 & 0 & 1 & 0 & 0 & x_{i1} & 0 & 0 & x_{i2} \end{bmatrix} \begin{bmatrix} \beta_{01} \\ \beta_{02} \\ \beta_{03} \\ \beta_{11} \\ \beta_{12} \\ \beta_{13} \\ \beta_{21} \\ \beta_{22} \\ \beta_{23} \end{bmatrix}$$


  
 $X\{i\}$

If you prefer, you can reshape the  $K$ -by-1 vector of coefficients back into a  $(p + 1)$ -by- $d$  matrix after estimation.

To put constraints on the model parameters, adjust the design matrix accordingly. For example, suppose that the three dimensions in the previous example have a common slope. That is,  $\beta_{11} = \beta_{12} = \beta_{13} = \beta_1$  and  $\beta_{21} = \beta_{22} = \beta_{23} = \beta_2$ . In this case, each design matrix is 3-by-5, as shown in the following figure.

$$\underbrace{\begin{bmatrix} 1 & 0 & 0 & x_{i1} & x_{i2} \\ 0 & 1 & 0 & x_{i1} & x_{i2} \\ 0 & 0 & 1 & x_{i1} & x_{i2} \end{bmatrix}}_{X\{i\}} \begin{bmatrix} \beta_{01} \\ \beta_{02} \\ \beta_{03} \\ \beta_1 \\ \beta_2 \end{bmatrix}$$

### Longitudinal Analysis

In a longitudinal analysis, you might measure responses on  $n$  subjects at  $d$  time points, with correlation between observations made on the same subject. For example, suppose that you measure responses  $y_{ij}$  at times  $t_{ij}$ ,  $i = 1, \dots, n$  and  $j = 1, \dots, d$ . In addition, suppose that each subject is in one of two groups (such as male or female), specified by the indicator variable  $G_i$ . You could model  $y_{ij}$  as a function of  $G_i$  and  $t_{ij}$ , with group-specific intercepts and slopes, as follows:

$$y_{ij} = \beta_0 + \beta_1 G_i + \beta_2 t_{ij} + \beta_3 G_i \times t_{ij} + \varepsilon_{ij}, \quad i = 1, \dots, n; j = 1, \dots, d,$$

where

$$\varepsilon_i = (\varepsilon_{i1}, \dots, \varepsilon_{id})' \sim MVN(\mathbf{0}, \Sigma).$$

Most longitudinal models include time as an explicit predictor.

To fit this model using `mvregress`, arrange the responses in an  $n$ -by- $d$  matrix, where  $n$  is the number of subjects and  $d$  is the number of time points. Specify the design matrices in an  $n$ -length cell array of  $d$ -by- $K$  matrices, where here  $K = 4$  for the four regression coefficients.

For example, suppose  $d = 5$  (five observations per subject). The  $i$ th design matrix and corresponding parameter vector for the specified model are shown in the following figure.

$$\underbrace{\begin{bmatrix} 1 & G_i & t_{i1} & G_i * t_{i1} \\ 1 & G_i & t_{i2} & G_i * t_{i2} \\ 1 & G_i & t_{i3} & G_i * t_{i3} \\ 1 & G_i & t_{i4} & G_i * t_{i4} \\ 1 & G_i & t_{i5} & G_i * t_{i5} \end{bmatrix}}_{X\{i\}} \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \beta_3 \end{bmatrix}$$

### Panel Analysis

In a panel analysis, you might measure responses and covariates on  $d$  subjects (such as individuals or countries) at  $n$  time points. For example, suppose you measure responses  $y_{ij}$  and covariates  $x_{ij}$  on subjects  $j = 1, \dots, d$  at times  $t = 1, \dots, n$ . A fixed effects panel model, with subject-specific fixed effects, and concurrent correlation might look like:

$$y_{tj} = \alpha_j + \beta x_{tj} + \varepsilon_{tj},$$

where

$$\varepsilon_t = (\varepsilon_{t1}, \dots, \varepsilon_{td})' \sim MVN(\mathbf{0}, \Sigma).$$

In contrast to longitudinal models, the panel analysis model typically includes covariates measured at each time point, instead of using time as an explicit predictor.

To fit this model using `mvregress`, arrange the responses in an  $n$ -by- $d$  matrix, such that each column corresponds to a subject. Specify the design matrices in an  $n$ -length cell array of  $d$ -by- $K$  matrices, where here  $K = d + 1$  for the  $d$  intercepts and a slope term.

For example, suppose  $d = 4$  (four subjects). The  $t$ th design matrix and corresponding parameter vector are shown in the following figure.

$$\underbrace{\begin{bmatrix} 1 & 0 & 0 & 0 & x_{t1} \\ 0 & 1 & 0 & 0 & x_{t2} \\ 0 & 0 & 1 & 0 & x_{t3} \\ 0 & 0 & 0 & 1 & x_{t4} \end{bmatrix}}_{X\{t\}} \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \\ \alpha_4 \\ \beta \end{bmatrix}$$

### Seemingly Unrelated Regression

In a seemingly unrelated regression (SUR), you model  $d$  separate regressions, each with its own intercept and slope, but a common error variance-covariance matrix. For example, suppose you measure responses  $y_{ij}$  and covariates  $x_{ij}$  for regression models  $j = 1, \dots, d$ , with  $i = 1, \dots, n$  observations to fit each regression. The SUR model might look like:

$$y_{ij} = \beta_{0j} + \beta_j x_{ij} + \varepsilon_{ij},$$

where

$$\varepsilon_i = (\varepsilon_{i1}, \dots, \varepsilon_{id})' \sim MVN(\mathbf{0}, \Sigma).$$

This model is very similar to the multivariate general linear model, except that it has different covariates for each dimension.

To fit this model using `mvregress`, arrange the responses in an  $n$ -by- $d$  matrix, such that each column has the data for the  $j$ th regression model. Specify the design matrices in an  $n$ -length cell array of  $d$ -by- $K$  matrices, where here  $K = 2d$  for  $d$  intercepts and  $d$  slopes.

For example, suppose  $d = 3$  (three regressions). The  $i$ th design matrix and corresponding parameter vector are shown in the following figure.

$$\underbrace{\begin{bmatrix} 1 & 0 & 0 & x_{i1} & 0 & 0 \\ 0 & 1 & 0 & 0 & x_{i2} & 0 \\ 0 & 0 & 1 & 0 & 0 & x_{i3} \end{bmatrix}}_{X\{i\}} \begin{bmatrix} \beta_{01} \\ \beta_{02} \\ \beta_{03} \\ \beta_1 \\ \beta_2 \\ \beta_3 \end{bmatrix}$$

### Vector Autoregressive Model

The VAR( $p$ ) vector autoregressive model expresses  $d$ -dimensional time series responses as a linear function of  $p$  lagged  $d$ -dimensional responses from previous times. For example, suppose you measure responses  $y_{jt}$  for time series  $j = 1, \dots, d$  at times  $t = 1, \dots, n$ . The VAR( $p$ ) model might look like:

$$\begin{bmatrix} y_{t1} \\ y_{t2} \\ \vdots \\ y_{td} \end{bmatrix} = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_d \end{bmatrix} + \begin{bmatrix} \varphi_{11}^{(1)} & \varphi_{12}^{(1)} & \cdots & \varphi_{1d}^{(1)} \\ \vdots & \vdots & \ddots & \vdots \\ \varphi_{d1}^{(1)} & \varphi_{d2}^{(1)} & \cdots & \varphi_{dd}^{(1)} \end{bmatrix} \begin{bmatrix} y_{t-1,1} \\ y_{t-1,2} \\ \vdots \\ y_{t-1,d} \end{bmatrix} + \cdots + \begin{bmatrix} \varphi_{11}^{(p)} & \varphi_{12}^{(p)} & \cdots & \varphi_{1d}^{(p)} \\ \vdots & \vdots & \ddots & \vdots \\ \varphi_{d1}^{(p)} & \varphi_{d2}^{(p)} & \cdots & \varphi_{dd}^{(p)} \end{bmatrix} \begin{bmatrix} y_{t-p,1} \\ y_{t-p,2} \\ \vdots \\ y_{t-p,d} \end{bmatrix} + \begin{bmatrix} \varepsilon_{t1} \\ \varepsilon_{t2} \\ \vdots \\ \varepsilon_{td} \end{bmatrix},$$

where

$$\varepsilon_t = (\varepsilon_{t1}, \dots, \varepsilon_{td})' \sim MVN(\mathbf{0}, \Sigma).$$

When estimating vector autoregressive models, you typically need to use the first  $p$  observations to initiate the model, or provide some other presample response values.

To fit this model using `mvregress`, arrange the responses in an  $n$ -by- $d$  matrix, such that each column corresponds to a time series. Specify the design matrices in an  $n$ -length cell array of  $d$ -by- $K$  matrices, where here  $K = d + pd^2$ .

For example, suppose  $d = 2$  (two time series) and  $p = 1$  (one lag). The  $t$ th design matrix and corresponding parameter vector are shown in the following figure.

$$\underbrace{\begin{bmatrix} 1 & 0 & y_{t-1,1} & 0 & y_{t-1,2} & 0 \\ 0 & 1 & 0 & y_{t-1,1} & 0 & y_{t-1,2} \end{bmatrix}}_{X\{t\}} \begin{bmatrix} c_1 \\ c_2 \\ \varphi_{11}^{(1)} \\ \varphi_{21}^{(1)} \\ \varphi_{12}^{(1)} \\ \varphi_{22}^{(1)} \end{bmatrix}$$

Alternatively, Econometrics Toolbox has functions for fitting and forecasting VAR( $p$ ) models, including the option to specify exogenous predictor variables.

## See Also

`mvregress` | `mvregresslike`

## Related Examples

- “Multivariate General Linear Model” on page 15-29
- “Fixed Effects Panel Model with Concurrent Correlation” on page 15-35

- “Longitudinal Analysis” on page 15-43

### **More About**

- “Multivariate Linear Regression” on page 15-4
- “Estimation of Multivariate Regression Models” on page 15-7

## Multivariate General Linear Model

This example shows how to set up a multivariate general linear model for estimation using `mvregress`.

### Load sample data.

This data contains measurements on a sample of 205 auto imports from 1985.

Here, model the bivariate response of city and highway MPG (columns 14 and 15).

For predictors, use wheel base (column 3), curb weight (column 7), and fuel type (column 18). The first two predictors are continuous, and for this example are centered and scaled. Fuel type is a categorical variable with two categories (11 and 20), so a dummy indicator variable is needed for the regression.

```
load('imports-85')
Y = X(:,14:15);
[n,d] = size(Y);

X1 = zscore(X(:,3));
X2 = zscore(X(:,7));
X3 = X(:,18)==20;

Xmat = [ones(n,1) X1 X2 X3];
```

The variable `X3` is coded to have value 1 for the fuel type 20, and value 0 otherwise.

For convenience, the three predictors (wheel base, curb weight, and fuel type indicator) are combined into one design matrix, with an added intercept term.

### Set up design matrices.

Given these predictors, the multivariate general linear model for the bivariate MPG response is

$$\begin{bmatrix} y_{11} & y_{12} \\ y_{21} & y_{22} \\ \vdots & \vdots \\ y_{n1} & y_{n2} \end{bmatrix} = \begin{bmatrix} 1 & x_{11} & x_{12} & x_{13} \\ 1 & x_{21} & x_{22} & x_{23} \\ \vdots & \vdots & \vdots & \vdots \\ 1 & x_{n1} & x_{n2} & x_{n3} \end{bmatrix} \begin{bmatrix} \beta_{01} & \beta_{02} \\ \beta_{11} & \beta_{12} \\ \beta_{21} & \beta_{22} \\ \beta_{31} & \beta_{32} \end{bmatrix} + \begin{bmatrix} \epsilon_{11} & \epsilon_{12} \\ \epsilon_{21} & \epsilon_{22} \\ \vdots & \vdots \\ \epsilon_{n1} & \epsilon_{n2} \end{bmatrix},$$

where  $\epsilon_i = (\epsilon_{i1}, \epsilon_{i2})' \sim MVN(0, \Sigma)$ . There are  $K = 8$  regression coefficients in total.

Create a length  $n = 205$  cell array of 2-by-8 (d-by-K) matrices for use with `mvregress`. The  $i$ th matrix in the cell array is

$$X(i) = \begin{bmatrix} 1 & 0 & x_{i1} & 0 & x_{i2} & 0 & x_{i3} & 0 \\ 0 & 1 & 0 & x_{i1} & 0 & x_{i2} & 0 & x_{i3} \end{bmatrix}.$$

```
Xcell = cell(1,n);  
for i = 1:n  
    Xcell{i} = [kron([Xmat(i,:)],eye(d))];  
end
```

Given this specification of the design matrices, the corresponding parameter vector is

$$\beta = \begin{bmatrix} \beta_{01} \\ \beta_{02} \\ \beta_{11} \\ \beta_{12} \\ \beta_{21} \\ \beta_{22} \\ \beta_{31} \\ \beta_{32} \end{bmatrix}$$

**Estimate regression coefficients.**

Fit the model using maximum likelihood estimation.

```
[beta,sigma,E,V] = mvregress(Xcell,Y);  
beta
```

```
beta =  
  
33.5476  
38.5720  
0.9723  
0.3950  
-6.3064  
-6.3584  
-9.2284  
-8.6663
```

These coefficient estimates show:



- The expected city and highway MPG for cars of average wheel base, curb weight, and fuel type 11 are 33.5 and 38.6, respectively. For fuel type 20, the expected city and highway MPG are  $33.5476 - 9.2284 = 24.3192$  and  $38.5720 - 8.6663 = 29.9057$ .
- An increase of one standard deviation in curb weight has almost the same effect on expected city and highway MPG. Given all else is equal, the expected MPG decreases by about 6.3 with each one standard deviation increase in curb weight, for both city and highway MPG.
- For each one standard deviation increase in wheel base, the expected city MPG increases 0.972, while the expected highway MPG increases by only 0.395, given all else is equal.

### Compute standard errors.

The standard errors for the regression coefficients are the square root of the diagonal of the variance-covariance matrix,  $V$ .

```
se = sqrt(diag(V))
```

```
se =
```

```
0.7365
0.7599
0.3589
0.3702
0.3497
0.3608
0.7790
0.8037
```

### Reshape coefficient matrix.

You can easily reshape the regression coefficients into the original 4-by-2 matrix.

```
B = reshape(beta, 2, 4)'
```

```
B =
```

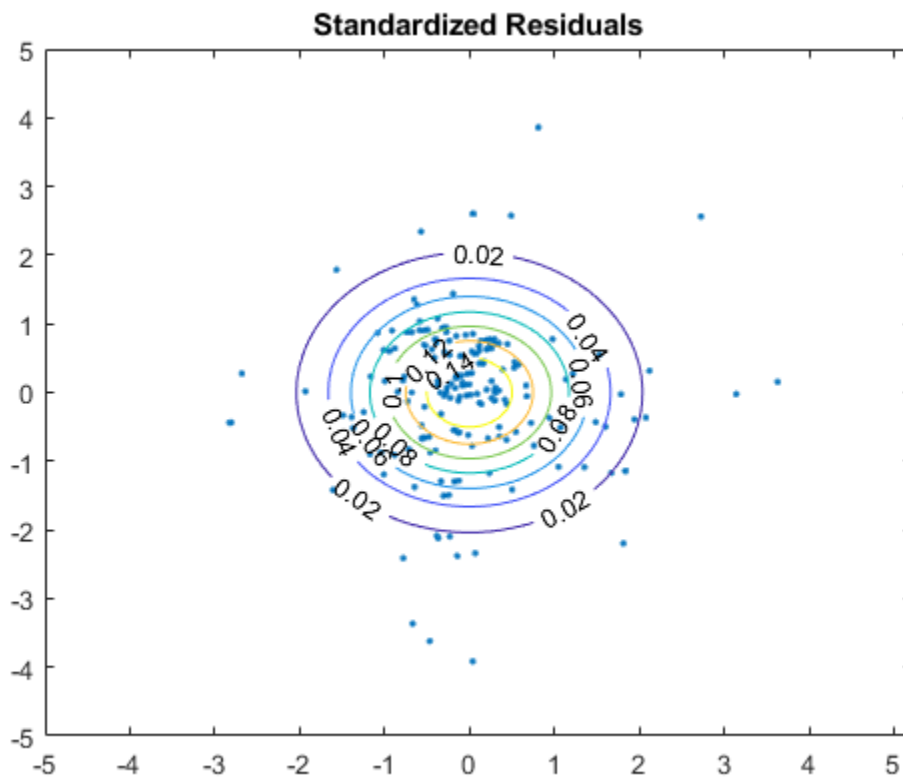
```
33.5476    38.5720
 0.9723     0.3950
```

```
-6.3064    -6.3584  
-9.2284    -8.6663
```

**Check model assumptions.**

Under the model assumptions,  $z = \mathbf{E}\Sigma^{-1/2}$  should be independent, with a bivariate standard normal distribution. In this 2-D case, you can assess the validity of this assumption using a scatter plot.

```
z = E/chol(sigma);  
figure()  
plot(z(:,1),z(:,2),'.')  
title('Standardized Residuals')  
hold on  
  
% Overlay standard normal contours  
z1 = linspace(-5,5);  
z2 = linspace(-5,5);  
[zx,zy] = meshgrid(z1,z2);  
zgrid = [reshape(zx,100^2,1),reshape(zy,100^2,1)];  
zn = reshape(mvnpdf(zgrid),100,100);  
[c,h] = contour(zx,zy,zn);  
clabel(c,h)
```



Several residuals are larger than expected, but overall, there is little evidence against the multivariate normality assumption.

## See Also

`mvregress` | `mvregresslike`

## Related Examples

- “Set Up Multivariate Regression Problems” on page 15-16
- “Fixed Effects Panel Model with Concurrent Correlation” on page 15-35

- “Longitudinal Analysis” on page 15-43

### **More About**

- “Multivariate Linear Regression” on page 15-4
- “Estimation of Multivariate Regression Models” on page 15-7

## Fixed Effects Panel Model with Concurrent Correlation

This example shows how to perform panel data analysis using `mvregress`. First, a fixed effects model with concurrent correlation is fit by ordinary least squares (OLS) to some panel data. Then, the estimated error covariance matrix is used to get panel corrected standard errors for the regression coefficients.

### Load sample data.

Navigate to the folder containing sample data. Load the sample panel data.

```
cd(matlabroot)
cd('help/toolbox/stats/examples')
load('panelData')
```

The dataset array, `panelData`, contains yearly observations on eight cities for 6 years. This is simulated data.

### Define variables.

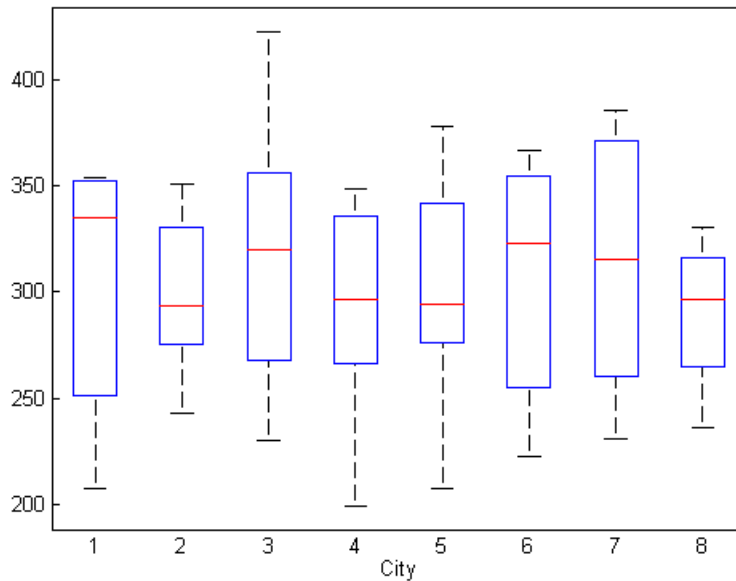
The first variable, `Growth`, measures economic growth (the response variable). The second and third variables are `city` and `year` indicators, respectively. The last variable, `Employ`, measures employment (the predictor variable).

```
y = panelData.Growth;
city = panelData.City;
year = panelData.Year;
x = panelData.Employ;
```

### Plot data grouped by category.

To look for potential city-specific fixed effects, create a box plot of the response grouped by city.

```
figure()
boxplot(y,city)
xlabel('City')
```

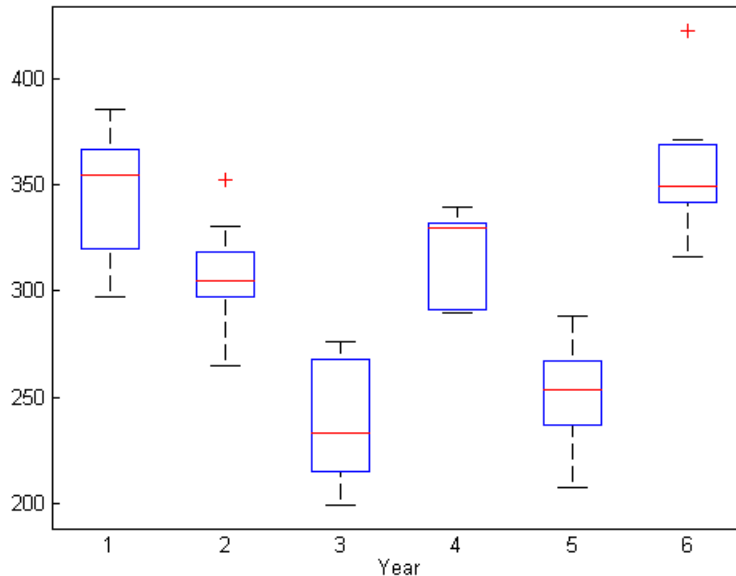


There does not appear to be any systematic differences in the mean response among cities.

**Plot data grouped by a different category.**

To look for potential year-specific fixed effects, create a box plot of the response grouped by year.

```
figure()  
boxplot(y, year)  
xlabel('Year')
```



Some evidence of systematic differences in the mean response between years seems to exist.

### Format response data.

Let  $y_{ij}$  denote the response for city  $j = 1, \dots, d$ , in year  $i = 1, \dots, n$ . Similarly,  $x_{ij}$  is the corresponding value of the predictor variable. In this example,  $n = 6$  and  $d = 8$ .

Consider fitting a year-specific fixed effects model with a constant slope and concurrent correlation among cities in the same year,

$$y_{ij} = \alpha_i + \beta_1 x_{ij} + \varepsilon_{ij}, \quad i = 1, \dots, n, j = 1, \dots, d,$$

where  $\varepsilon_i = (\varepsilon_{i1}, \dots, \varepsilon_{id})' \sim MVN(\mathbf{0}, \Sigma)$ . The concurrent correlation accounts for any unmeasured, time-static factors that might impact growth similarly for some cities. For example, cities with close spatial proximity might be more likely to have similar economic growth.

To fit this model using `mvregress`, reshape the response data into an  $n$ -by- $d$  matrix.

```
n = 6; d = 8;  
Y = reshape(y,n,d);
```

**Format design matrices.**

Create a length- $n$  cell array of  $d$ -by- $K$  design matrices. For this model, there are  $K = 7$  parameters ( $d = 6$  intercept terms and a slope).

Suppose the vector of parameters is arranged as

$$\beta = \begin{pmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_6 \\ \beta_1 \end{pmatrix}.$$

In this case, the first design matrix for year 1 looks like

$$X\{1\} = \begin{pmatrix} 1 & 0 & \cdots & 0 & x_{11} \\ 1 & 0 & \cdots & 0 & x_{12} \\ \vdots & \vdots & \cdots & 0 & \vdots \\ 1 & 0 & \cdots & 0 & x_{18} \end{pmatrix},$$

and the second design matrix for year 2 looks like

$$X\{2\} = \begin{pmatrix} 0 & 1 & 0 & \cdots & 0 & x_{21} \\ 0 & 1 & 0 & \cdots & 0 & x_{22} \\ \vdots & \vdots & 0 & \cdots & 0 & \vdots \\ 0 & 1 & 0 & \cdots & 0 & x_{28} \end{pmatrix}.$$

The design matrices for the remaining 4 years are similar.

```
K = 7; N = n*d;  
X = cell(n,1);  
for i = 1:n  
    x0 = zeros(d,K-1);  
    x0(:,i) = 1;  
    X{i} = [x0,x(i:n:N)];  
end
```

**Fit the model.**

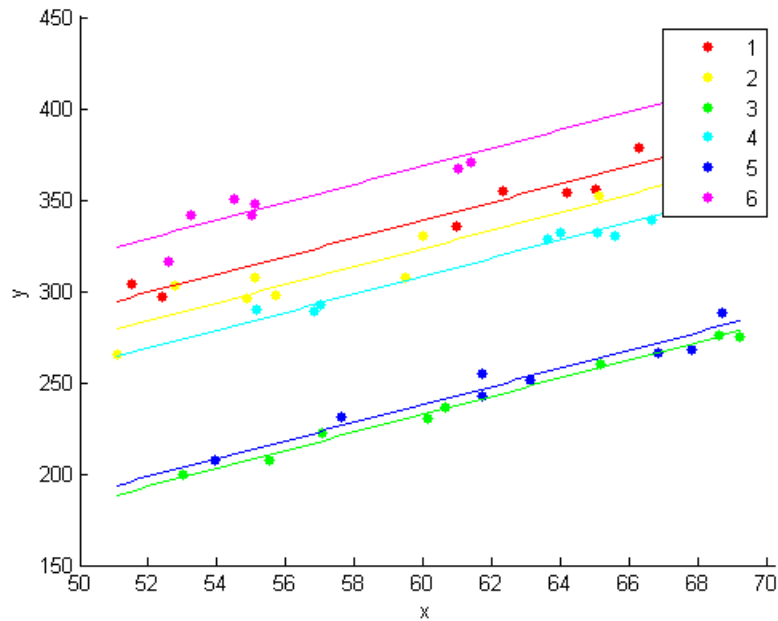
Fit the model using ordinary least squares (OLS).



```
[b,sig,E,V] = mvregress(X,Y,'algorithm','cqls');  
b  
  
b =  
  
    41.6878  
    26.1864  
   -64.5107  
    11.0924  
   -59.1872  
    71.3313  
     4.9525
```

### Plot fitted model.

```
xx = linspace(min(x),max(x));  
axx = repmat(b(1:K-1),1,length(xx));  
bxx = repmat(b(K)*xx,n,1);  
yhat = axx + bxx;  
  
figure()  
hPoints = gscatter(x,y,year);  
hold on  
hLines = plot(xx,yhat);  
for i=1:n  
    set(hLines(i),'color',get(hPoints(i),'color'));  
end  
hold off
```

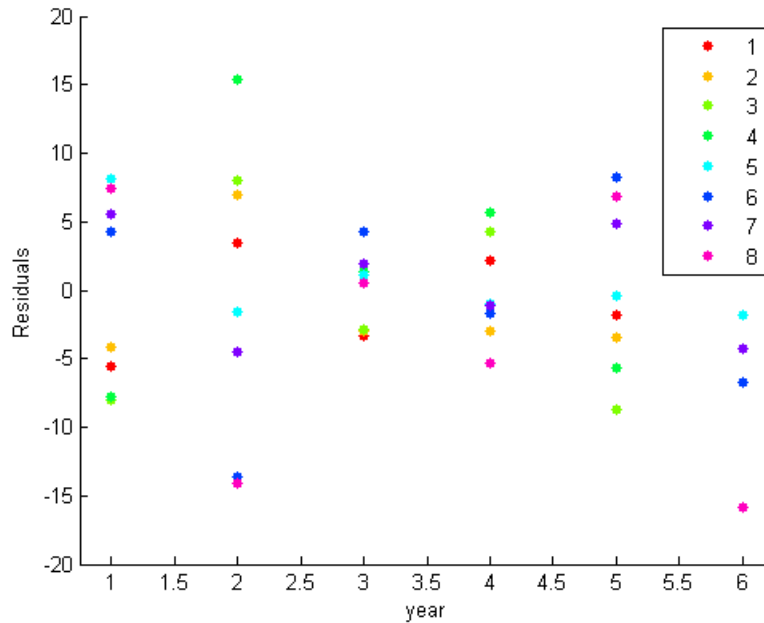


The model with year-specific intercepts and common slope appears to fit the data quite well.

### Residual correlation.

Plot the residuals, grouped by year.

```
figure()  
gscatter(year,E(:),city)  
ylabel('Residuals')
```



The residual plot suggests concurrent correlation is present. For examples, cities 1, 2, 3, and 4 are consistently above or below average as a group in any given year. The same is true for the collection of cities 5, 6, 7, and 8. As seen in the exploratory plots, there are no systematic city-specific effects.

### Panel corrected standard errors.

Use the estimated error variance-covariance matrix to compute panel corrected standard errors for the regression coefficients.

```
XX = cell2mat(X);
S = kron(eye(n), sig);
Vpcse = inv(XX'*XX)*XX'*S*XX*inv(XX'*XX);
se = sqrt(diag(Vpcse))
```

se =

```
9.3750
8.6698
9.3406
```

9.4286  
9.5729  
8.8207  
0.1527

## See Also

`mvregress` | `mvregresslike`

## Related Examples

- “Set Up Multivariate Regression Problems” on page 15-16
- “Multivariate General Linear Model” on page 15-29
- “Longitudinal Analysis” on page 15-43

## More About

- “Multivariate Linear Regression” on page 15-4
- “Estimation of Multivariate Regression Models” on page 15-7

## Longitudinal Analysis

This example shows how to perform longitudinal analysis using `mvregress`.

### Load sample data.

Navigate to the folder containing sample data. Load the sample longitudinal data.

```
cd(matlabroot)
cd('help/toolbox/stats/examples')
load('longitudinalData')
```

The matrix `Y` contains response data for 16 individuals. The response is the blood level of a drug measured at five time points ( $t = 0, 2, 4, 6,$  and  $8$ ). Each row of `Y` corresponds to an individual, and each column corresponds to a time point. The first eight subjects are female, and the second eight subjects are male. This is simulated data.

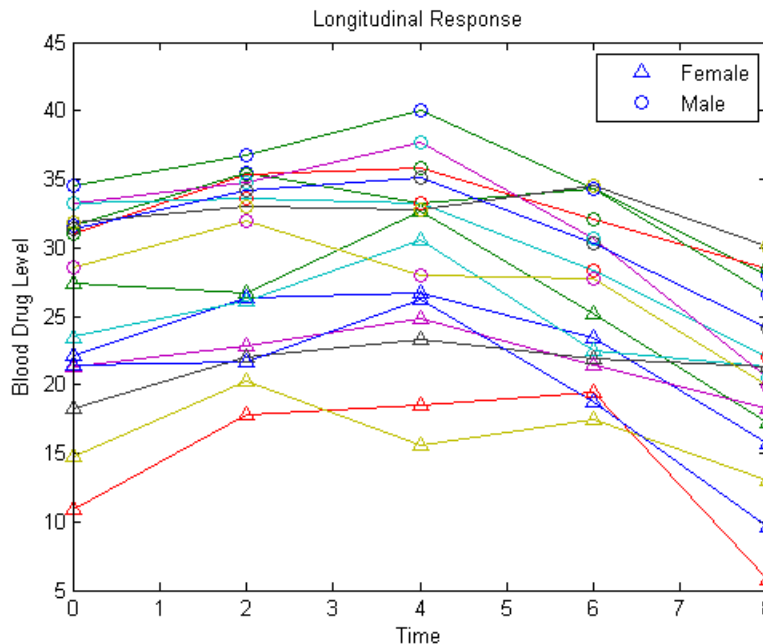
### Plot data.

Plot the data for all 16 subjects.

```
figure()
t = [0,2,4,6,8];
plot(t,Y)
hold on

hf = plot(t,Y(1:8,:), '^');
hm = plot(t,Y(9:16,:), 'o');
legend([hf(1),hm(1)], 'Female', 'Male', 'Location', 'NorthEast')

title('Longitudinal Response')
ylabel('Blood Drug Level')
xlabel('Time')
hold off
```



### Define design matrices.

Let  $y_{ij}$  denote the response for individual  $i = 1, \dots, n$  measured at times  $t_{ij}, j = 1, \dots, d$ . In this example,  $n = 16$  and  $d = 5$ . Let  $G_i$  denote the gender of individual  $i$ , where  $G_i = 1$  for males and 0 for females.

Consider fitting a quadratic longitudinal model, with a separate slope and intercept for each gender,

$$y_{ij} = \beta_0 + \beta_1 G_i + \beta_2 t_{ij} + \beta_3 t_{ij}^2 + \beta_4 G_i \times t_{ij} + \beta_5 G_i \times t_{ij}^2 + \varepsilon_{ij},$$

where  $\varepsilon_i = (\varepsilon_{i1}, \dots, \varepsilon_{id})' \sim MVN(\mathbf{0}, \Sigma)$ . The error correlation accounts for clustering within an individual.

To fit this model using `mvregress`, the response data should be in an  $n$ -by- $d$  matrix.  $\mathbf{Y}$  is already in the proper format.

Next, create a length- $n$  cell array of  $d$ -by- $K$  design matrices. For this model, there are  $K = 6$  parameters.

For individual  $i$ , the 5-by-6 design matrix is

$$X(i) = \begin{pmatrix} 1 & G_i & t_{i1} & t_{i1}^2 & G_i \times t_{i1} & G_i \times t_{i1}^2 \\ 1 & G_i & t_{i2} & t_{i2}^2 & G_i \times t_{i2} & G_i \times t_{i2}^2 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & G_i & t_{i5} & t_{i5}^2 & G_i \times t_{i5} & G_i \times t_{i5}^2 \end{pmatrix},$$

corresponding to the parameter vector

$$\beta = \begin{pmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_5 \end{pmatrix}.$$

The matrix X1 has the design matrix for a female, and X2 has the design matrix for a male.

Create a cell array of design matrices. The first eight individuals are females, and the second eight are males.

```
X = cell(8,1);
X(1:8) = {X1};
X(9:16) = {X2};
```

### Fit the model.

Fit the model using maximum likelihood estimation. Display the estimated coefficients and standard errors.

```
[b,sig,E,V,loglikF] = mvregress(X,Y);
[b sqrt(diag(V))]
```

```
ans =
```

```
18.8619    0.7432
13.0942    1.0511
 2.5968    0.2845
-0.3771    0.0398
-0.5929    0.4023
 0.0290    0.0563
```

The coefficients on the interaction terms (in the last two rows of b) do not appear significant. You can use the value of the loglikelihood objective function for this fit,

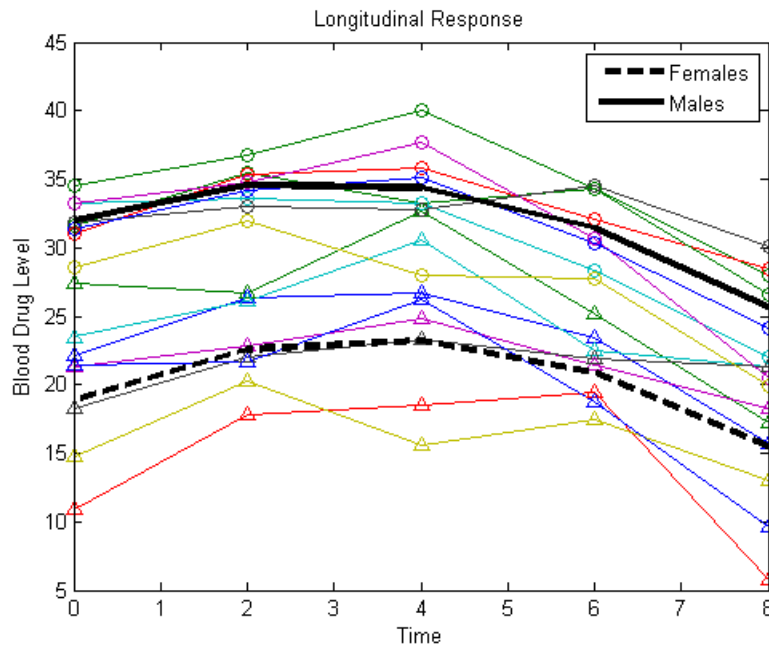
loglikF, to compare this model to one without the interaction terms using a likelihood ratio test.

### Plot fitted model.

Plot the fitted lines for females and males.

```
Yhatf = X1*b;  
Yhatm = X2*b;  
  
figure()  
plot(t,Y)  
hold on  
  
plot(t,Y(1:8,:), '^', t,Y(9:16,:), 'o')  
  
hf = plot(t,Yhatf, 'k--', 'LineWidth', 3);  
hm = plot(t,Yhatm, 'k', 'LineWidth', 3);  
legend([hf,hm], 'Females', 'Males', 'Location', 'NorthEast')  
  
title('Longitudinal Response')  
ylabel('Blood Drug Level')  
xlabel('Time')  
hold off
```





### Define a reduced model.

Fit the model without interaction terms,

$$y_{ij} = \beta_0 + \beta_1 G_i + \beta_2 t_{ij} + \beta_3 t_{ij}^2 + \varepsilon_{ij},$$

where  $\varepsilon_i = (\varepsilon_{i1}, \dots, \varepsilon_{id})' \sim MVN(\mathbf{0}, \Sigma)$ .

This model has four coefficients, which correspond to the first four columns of the design matrices X1 and X2 (for females and males, respectively).

```
X1R = X1(:, 1:4);
X2R = X2(:, 1:4);

XR = cell(8, 1);
XR(1:8) = {X1R};
XR(9:16) = {X2R};
```

**Fit the reduced model.**

Fit this model using maximum likelihood estimation. Display the estimated coefficients and their standard errors.

```
[bR, sigR, ER, VR, loglikR] = mvregress(XR, Y);  
[bR, sqrt(diag(VR))] ]
```

```
ans =
```

```
    19.3765    0.6898  
    12.0936    0.8591  
     2.2919    0.2139  
    -0.3623    0.0283
```

**Conduct a likelihood ratio test.**

Compare the two models using a likelihood ratio test. The null hypothesis is that the reduced model is sufficient. The alternative is that the reduced model is inadequate (compared to the full model with the interaction terms).

The likelihood ratio test statistic is compared to a chi-squared distribution with two degrees of freedom (for the two coefficients being dropped).

```
LR = 2*(loglikF-loglikR);  
pval = 1 - chi2cdf(LR,2)
```

```
pval =
```

```
    0.0803
```

The  $p$ -value 0.0803 indicates that the null hypothesis is not rejected at the 5% significance level. Therefore, there is insufficient evidence that the extra terms improve the fit.

**See Also**

`mvregress` | `mvregresslike`

**Related Examples**

- “Set Up Multivariate Regression Problems” on page 15-16

- “Multivariate General Linear Model” on page 15-29
- “Fixed Effects Panel Model with Concurrent Correlation” on page 15-35

### **More About**

- “Multivariate Linear Regression” on page 15-4
- “Estimation of Multivariate Regression Models” on page 15-7

## Multidimensional Scaling

One of the most important goals in visualizing data is to get a sense of how near or far points are from each other. Often, you can do this with a scatter plot. However, for some analyses, the data that you have might not be in the form of points at all, but rather in the form of pairwise similarities or dissimilarities between cases, observations, or subjects. There are no points to plot.

Even if your data are in the form of points rather than pairwise distances, a scatter plot of those data might not be useful. For some kinds of data, the relevant way to measure how near two points are might not be their Euclidean distance. While scatter plots of the raw data make it easy to compare Euclidean distances, they are not always useful when comparing other kinds of inter-point distances, city block distance for example, or even more general dissimilarities. Also, with a large number of variables, it is very difficult to visualize distances unless the data can be represented in a small number of dimensions. Some sort of dimension reduction is usually necessary.

Multidimensional scaling (MDS) is a set of methods that address all these problems. MDS allows you to visualize how near points are to each other for many kinds of distance or dissimilarity metrics and can produce a representation of your data in a small number of dimensions. MDS does not require raw data, but only a matrix of pairwise distances or dissimilarities.

### See Also

`cmdscales` | `mdscale`

### Related Examples

- “Nonclassical and Nonmetric Multidimensional Scaling” on page 15-51
- “Classical Multidimensional Scaling” on page 15-57

## Nonclassical and Nonmetric Multidimensional Scaling

Perform nonclassical multidimensional scaling using `mdscale`.

### In this section...

“Nonclassical Multidimensional Scaling” on page 15-51

“Nonmetric Multidimensional Scaling” on page 15-53

### Nonclassical Multidimensional Scaling

The function `mdscale` performs nonclassical multidimensional scaling. As with `cmdscale`, you use `mdscale` either to visualize dissimilarity data for which no “locations” exist, or to visualize high-dimensional data by reducing its dimensionality. Both functions take a matrix of dissimilarities as an input and produce a configuration of points. However, `mdscale` offers a choice of different criteria to construct the configuration, and allows missing data and weights.

For example, the cereal data include measurements on 10 variables describing breakfast cereals. You can use `mdscale` to visualize these data in two dimensions. First, load the data. For clarity, this example code selects a subset of 22 of the observations.

```
load cereal.mat
X = [Calories Protein Fat Sodium Fiber ...
     Carbo Sugars Shelf Potass Vitamins];
% Take a subset from a single manufacturer
mfg1 = strcmp('G',cellstr(Mfg));
X = X(mfg1,:);
size(X)
ans =
    22 10
```

Then use `pdist` to transform the 10-dimensional data into dissimilarities. The output from `pdist` is a symmetric dissimilarity matrix, stored as a vector containing only the  $(23 \times 22/2)$  elements in its upper triangle.

```
dissimilarities = pdist(zscore(X),'cityblock');
size(dissimilarities)
ans =
    1    231
```

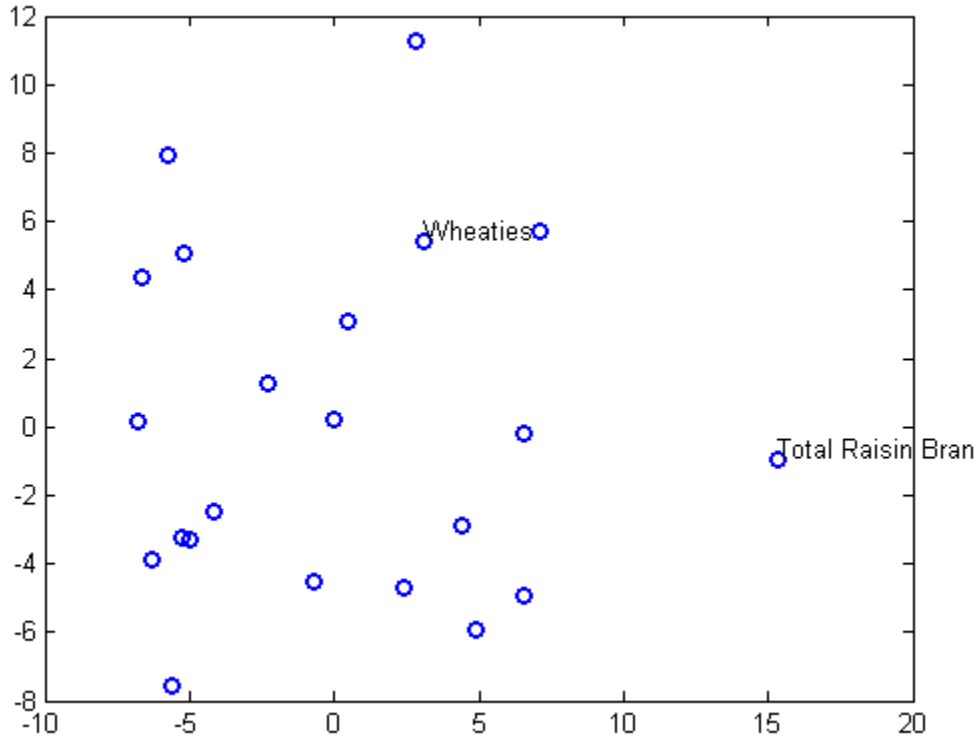
This example code first standardizes the cereal data, and then uses city block distance as a dissimilarity. The choice of transformation to dissimilarities is application-dependent, and the choice here is only for simplicity. In some applications, the original data are already in the form of dissimilarities.

Next, use `mdscale` to perform metric MDS. Unlike `cmdscale`, you must specify the desired number of dimensions, and the method to use to construct the output configuration. For this example, use two dimensions. The metric STRESS criterion is a common method for computing the output; for other choices, see the `mdscale` reference page in the online documentation. The second output from `mdscale` is the value of that criterion evaluated for the output configuration. It measures the how well the inter-point distances of the output configuration approximate the original input dissimilarities:

```
[Y, stress] =...
mdscale(dissimilarities, 2, 'criterion', 'metricstress');
stress
stress =
    0.1856
```

A scatterplot of the output from `mdscale` represents the original 10-dimensional data in two dimensions, and you can use the `gname` function to label selected points:

```
plot(Y(:,1), Y(:,2), 'o', 'LineWidth', 2);
gname(Name(mfg1))
```



## Nonmetric Multidimensional Scaling

Metric multidimensional scaling creates a configuration of points whose inter-point distances approximate the given dissimilarities. This is sometimes too strict a requirement, and non-metric scaling is designed to relax it a bit. Instead of trying to approximate the dissimilarities themselves, non-metric scaling approximates a nonlinear, but monotonic, transformation of them. Because of the monotonicity, larger or smaller distances on a plot of the output will correspond to larger or smaller dissimilarities, respectively. However, the nonlinearity implies that `mdscale` only attempts to preserve the ordering of dissimilarities. Thus, there may be contractions or expansions of distances at different scales.

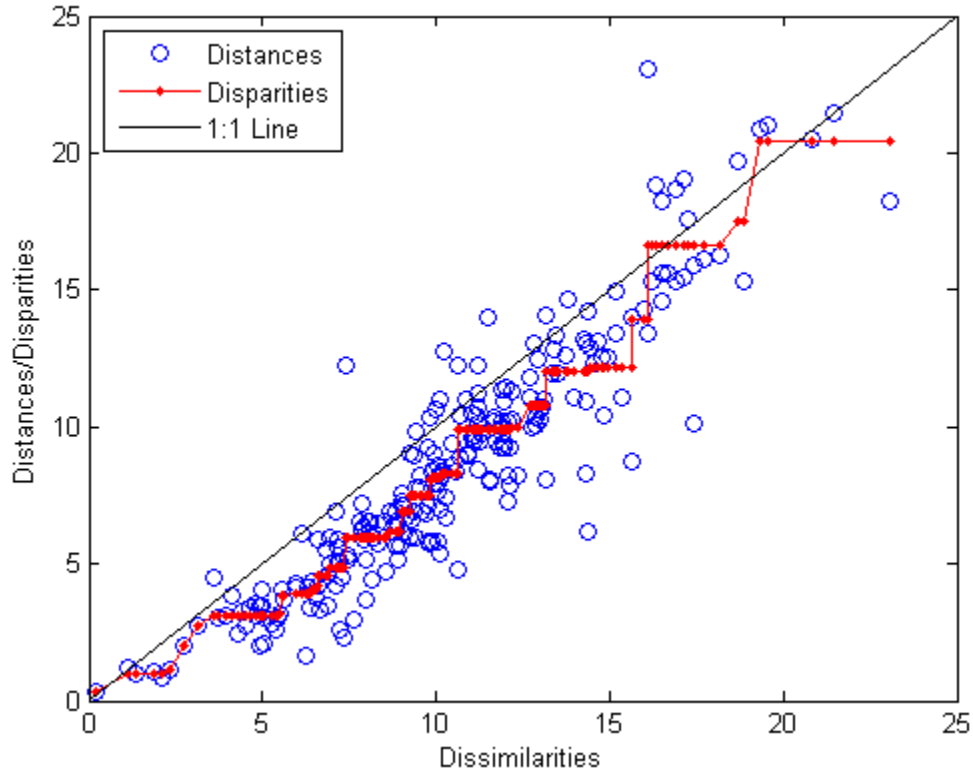
You use `mdscale` to perform nonmetric MDS in much the same way as for metric scaling. The nonmetric STRESS criterion is a common method for computing the output; for more choices, see the `mdscale` reference page in the online documentation. As with metric scaling, the second output from `mdscale` is the value of that criterion evaluated for the output configuration. For nonmetric scaling, however, it measures the how well the inter-point distances of the output configuration approximate the disparities. The disparities are returned in the third output. They are the transformed values of the original dissimilarities:

```
[Y, stress, disparities] = ...
mdscale(dissimilarities, 2, 'criterion', 'stress');
stress
stress =
    0.1562
```

To check the fit of the output configuration to the dissimilarities, and to understand the disparities, it helps to make a Shepard plot:

```
distances = pdist(Y);
[dum, ord] = sortrows([disparities(:) dissimilarities(:)]);
plot(dissimilarities, distances, 'bo', ...
     dissimilarities(ord), disparities(ord), 'r.-', ...
     [0 25], [0 25], 'k-')
xlabel('Dissimilarities')
ylabel('Distances/Disparities')
legend({'Distances' 'Disparities' '1:1 Line'}, ...
     'Location', 'NorthWest');
```





This plot shows that `mdscale` has found a configuration of points in two dimensions whose inter-point distances approximates the disparities, which in turn are a nonlinear transformation of the original dissimilarities. The concave shape of the disparities as a function of the dissimilarities indicates that fit tends to contract small distances relative to the corresponding dissimilarities. This may be perfectly acceptable in practice.

`mdscale` uses an iterative algorithm to find the output configuration, and the results can often depend on the starting point. By default, `mdscale` uses `cmdscale` to construct an initial configuration, and this choice often leads to a globally best solution. However, it is possible for `mdscale` to stop at a configuration that is a local minimum of the criterion. Such cases can be diagnosed and often overcome by running `mdscale` multiple times with different starting points. You can do this using the `'start'` and `'replicates'`

parameters. The following code runs five replicates of MDS, each starting at a different randomly-chosen initial configuration. The criterion value is printed out for each replication; `mdscale` returns the configuration with the best fit.

```
opts = statset('Display','final');  
[Y, stress] =...  
mdscale(dissimilarities,2,'criterion','stress',...  
'start','random','replicates',5,'Options',opts);
```

```
35 iterations, Final stress criterion = 0.156209  
31 iterations, Final stress criterion = 0.156209  
48 iterations, Final stress criterion = 0.171209  
33 iterations, Final stress criterion = 0.175341  
32 iterations, Final stress criterion = 0.185881
```

Notice that `mdscale` finds several different local solutions, some of which do not have as low a stress value as the solution found with the `cmdscale` starting point.

## Classical Multidimensional Scaling

This example shows how to use `cmdscale` to perform classical (metric) multidimensional scaling, also known as principal coordinates analysis.

`cmdscale` takes as an input a matrix of inter-point distances and creates a configuration of points. Ideally, those points are in two or three dimensions, and the Euclidean distances between them reproduce the original distance matrix. Thus, a scatter plot of the points created by `cmdscale` provides a visual representation of the original distances.

As a very simple example, you can reconstruct a set of points from only their inter-point distances. First, create some four dimensional points with a small component in their fourth coordinate, and reduce them to distances.

```
rng default; % For reproducibility
X = [normrnd(0,1,10,3), normrnd(0,.1,10,1)];
D = pdist(X, 'euclidean');
```

Next, use `cmdscale` to find a configuration with those inter-point distances. `cmdscale` accepts distances as either a square matrix, or, as in this example, in the vector upper-triangular form produced by `pdist`.

```
[Y, eigvals] = cmdscale(D);
```

`cmdscale` produces two outputs. The first output, `Y`, is a matrix containing the reconstructed points. The second output, `eigvals`, is a vector containing the sorted eigenvalues of what is often referred to as the "scalar product matrix," which, in the simplest case, is equal to  $Y^*Y'$ . The relative magnitudes of those eigenvalues indicate the relative contribution of the corresponding columns of `Y` in reproducing the original distance matrix `D` with the reconstructed points.

```
format short g
[eigvals eigvals/max(abs(eigvals))]
```

```
ans =
```

```
    35.41         1
    11.158        0.31511
     1.6894       0.04771
     0.1436       0.0040553
```

```
7.3463e-15  2.0746e-16
5.8409e-15  1.6495e-16
2.3307e-15  6.582e-17
8.5177e-16  2.4054e-17
-4.1421e-16 -1.1698e-17
-2.9669e-15 -8.3787e-17
```

If `eigvals` contains only positive and zero (within round-off error) eigenvalues, the columns of `Y` corresponding to the positive eigenvalues provide an exact reconstruction of `D`, in the sense that their inter-point Euclidean distances, computed using `pdist`, for example, are identical (within round-off) to the values in `D`.

```
maxerr4 = max(abs(D - pdist(Y))) % Exact reconstruction

maxerr4 =

    7.1054e-15
```

If two or three of the eigenvalues in `eigvals` are much larger than the rest, then the distance matrix based on the corresponding columns of `Y` nearly reproduces the original distance matrix `D`. In this sense, those columns form a lower-dimensional representation that adequately describes the data. However it is not always possible to find a good low-dimensional reconstruction.

```
maxerr3 = max(abs(D - pdist(Y(:,1:3)))) % Good reconstruction in 3D
maxerr2 = max(abs(D - pdist(Y(:,1:2)))) % Poor reconstruction in 2D

maxerr3 =

    0.043142

maxerr2 =

    0.98315
```

The reconstruction in three dimensions reproduces `D` very well, but the reconstruction in two dimensions has errors that are of the same order of magnitude as the largest values in `D`.

```
max (max (D) )
```

```
ans =
```

```
5.8974
```

Often, `eigvals` contains some negative eigenvalues, indicating that the distances in  $D$  cannot be reproduced exactly. That is, there might not be any configuration of points whose inter-point Euclidean distances are given by  $D$ . If the largest negative eigenvalue is small in magnitude relative to the largest positive eigenvalues, then the configuration returned by `cmdscale` might still reproduce  $D$  well.

## Example: Multidimensional Scaling

This example shows how to construct a map of 10 US cities based on the distances between those cities, using `cmdscale`.

First, create the distance matrix and pass it to `cmdscale`. In this example, `D` is a full distance matrix: it is square and symmetric, has positive entries off the diagonal, and has zeros on the diagonal.

```
cities = ...
{'Atl', 'Chi', 'Den', 'Hou', 'LA', 'Mia', 'NYC', 'SF', 'Sea', 'WDC'};
D = [
    0 587 1212 701 1936 604 748 2139 2182 543;
    587 0 920 940 1745 1188 713 1858 1737 597;
    1212 920 0 879 831 1726 1631 949 1021 1494;
    701 940 879 0 1374 968 1420 1645 1891 1220;
    1936 1745 831 1374 0 2339 2451 347 959 2300;
    604 1188 1726 968 2339 0 1092 2594 2734 923;
    748 713 1631 1420 2451 1092 0 2571 2408 205;
    2139 1858 949 1645 347 2594 2571 0 678 2442;
    2182 1737 1021 1891 959 2734 2408 678 0 2329;
    543 597 1494 1220 2300 923 205 2442 2329 0];
[Y,eigvals] = cmdscale(D);
```

Next, look at the eigenvalues returned by `cmdscale`. Some of these are negative, indicating that the original distances are not Euclidean. This is because of the curvature of the earth.

```
format short g
[eigvals eigvals/max(abs(eigvals))]
```

```
ans =
    9.5821e+06         1
    1.6868e+06     0.17604
    8157.3     0.0008513
    1432.9     0.00014954
    508.67     5.3085e-05
    25.143     2.624e-06
    8.9331e-10     9.3227e-17
    -897.7     -9.3685e-05
    -5467.6     -0.0005706
    -35479     -0.0037026
```

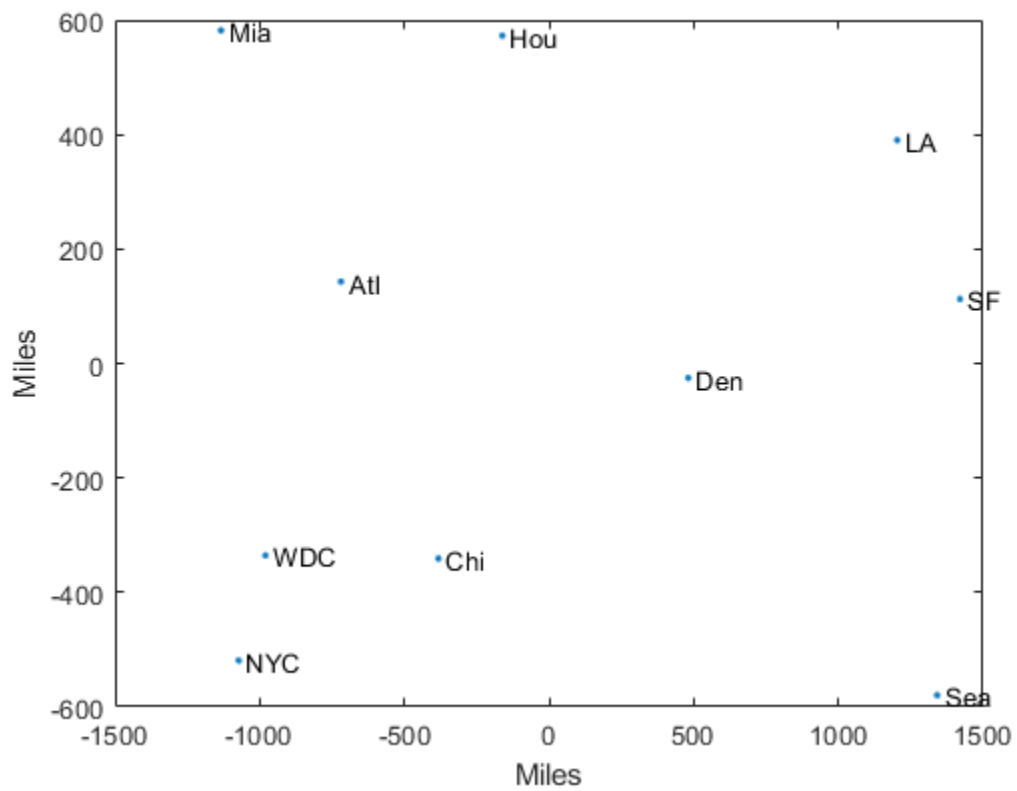
However, in this case, the two largest positive eigenvalues are much larger in magnitude than the remaining eigenvalues. So, despite the negative eigenvalues, the first two coordinates of  $Y$  are sufficient for a reasonable reproduction of  $D$ .

```
Dtriu = D(find(tril(ones(10),-1)))';  
maxrelerr = max(abs(Dtriu-pdist(Y(:,1:2))))./max(Dtriu)
```

```
maxrelerr =  
  
    0.0075371
```

Here is a plot of the reconstructed city locations as a map. The orientation of the reconstruction is arbitrary.

```
plot(Y(:,1),Y(:,2),'.')  
text(Y(:,1)+25,Y(:,2),cities)  
xlabel('Miles')  
ylabel('Miles')
```





## Procrustes Analysis

### In this section...

“Compare Landmark Data” on page 15-63

“Data Input” on page 15-63

“Preprocess Data for Accurate Results” on page 15-64

### Compare Landmark Data

The `procrustes` function analyzes the distribution of a set of shapes using Procrustes analysis. This analysis method matches landmark data (geometric locations representing significant features in a given shape) to calculate the best shape-preserving Euclidian transformations. These transformations minimize the differences in location between compared landmark data.

Procrustes analysis is also useful in conjunction with multidimensional scaling. In “Example: Multidimensional Scaling” on page 15-60 there is an observation that the orientation of the reconstructed points is arbitrary. Two different applications of multidimensional scaling could produce reconstructed points that are very similar in principle, but that look different because they have different orientations. The `procrustes` function transforms one set of points to make them more comparable to the other.

### Data Input

The `procrustes` function takes two matrices as input:

- The target shape matrix  $X$  has dimension  $n \times p$ , where  $n$  is the number of landmarks in the shape and  $p$  is the number of measurements per landmark.
- The comparison shape matrix  $Y$  has dimension  $n \times q$  with  $q \leq p$ . If there are fewer measurements per landmark for the comparison shape than the target shape ( $q < p$ ), the function adds columns of zeros to  $Y$ , yielding an  $n \times p$  matrix.

The equation to obtain the transformed shape,  $Z$ , is

$$Z = bYT + c$$

where:

- $b$  is a scaling factor that stretches ( $b > 1$ ) or shrinks ( $b < 1$ ) the points.
- $T$  is the orthogonal rotation and reflection matrix.
- $c$  is a matrix with constant values in each column, used to shift the points.

The `procrustes` function chooses  $b$ ,  $T$ , and  $c$  to minimize the distance between the target shape  $X$  and the transformed shape  $Z$  as measured by the least squares criterion:

$$\sum_{i=1}^n \sum_{j=1}^p (X_{ij} - Z_{ij})^2$$

## Preprocess Data for Accurate Results

Procrustes analysis is appropriate when all  $p$  measurement dimensions have similar scales. The analysis would be inaccurate, for example, if the columns of  $Z$  had different scales:

- The first column is measured in milliliters ranging from 2,000 to 6,000.
- The second column is measured in degrees Celsius ranging from 10 to 25.
- The third column is measured in kilograms ranging from 50 to 230.

In such cases, standardize your variables by:

- 1 Subtracting the sample mean from each variable.
- 2 Dividing each resultant variable by its sample standard deviation.

Use the `zscore` function to perform this standardization.

## See Also

`procrustes`

## Related Examples

- “Compare Handwritten Shapes Using Procrustes Analysis” on page 15-65

## Compare Handwritten Shapes Using Procrustes Analysis

This example shows how to use Procrustes analysis to compare two handwritten number threes. Visually and analytically explore the effects of forcing size and reflection changes.

### Load and Display the Original Data

Input landmark data for two handwritten number threes.

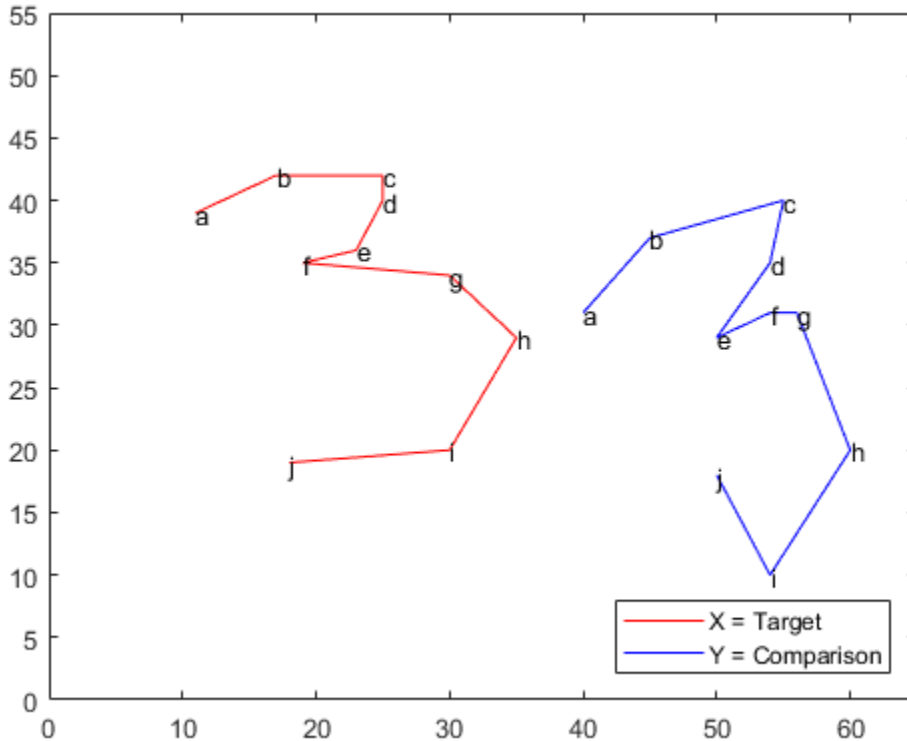
```
A = [11 39;17 42;25 42;25 40;23 36;19 35;30 34;35 29;...
30 20;18 19];
B = [15 31;20 37;30 40;29 35;25 29;29 31;31 31;35 20;...
29 10;25 18];
```

Create X and Y from A and B , moving B to the side to make each shape more visible.

```
X = A;
Y = B + repmat([25 0], 10,1);
```

Plot the shapes, using letters to designate the landmark points. Lines in the figure join the points to indicate the drawing path of each shape.

```
plot(X(:,1), X(:,2), 'r-', Y(:,1), Y(:,2), 'b-');
text(X(:,1), X(:,2), ('abcdefghij'))
text(Y(:,1), Y(:,2), ('abcdefghij'))
legend('X = Target', 'Y = Comparison', 'location', 'SE')
xlim([0 65]);
ylim([0 55]);
```



### Calculate the Best Transformation

Use Procrustes analysis to find the transformation that minimizes distances between landmark data points.

```
[d,Z,tr] = procrustes(X,Y);
```

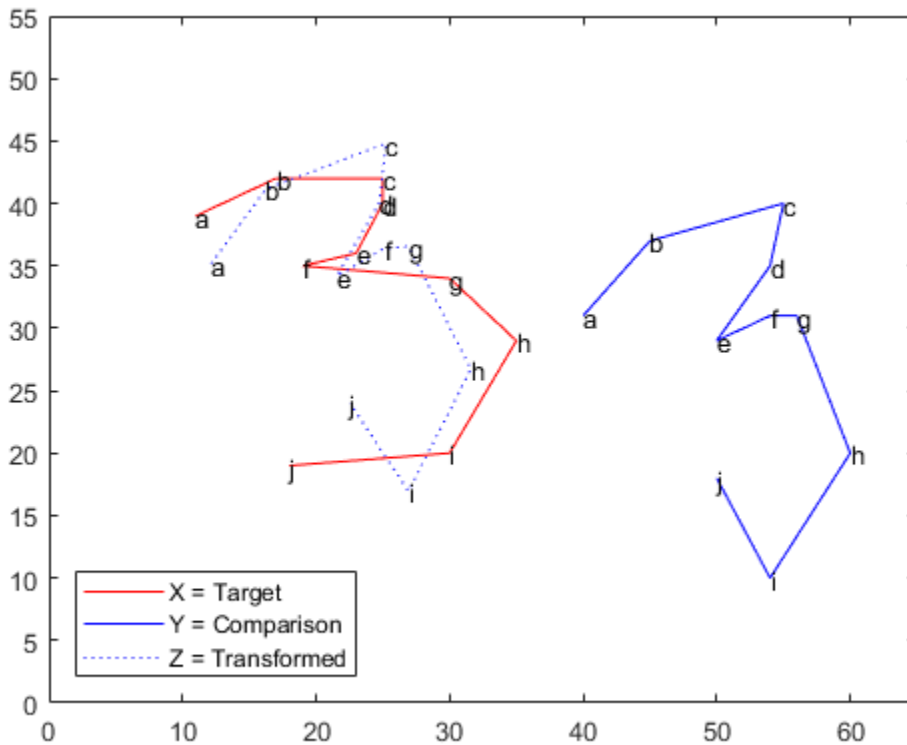
The outputs of the function are  $d$  (a standardized dissimilarity measure),  $Z$  (a matrix of the transformed landmarks), and  $tr$  (a structure array of the computed transformation with fields  $T$ ,  $b$ , and  $c$  which correspond to the transformation equation).

Visualize the transformed shape,  $Z$ , using a dashed blue line.

```

plot(X(:,1), X(:,2), 'r-', Y(:,1), Y(:,2), 'b-', ...
     Z(:,1), Z(:,2), 'b:');
text(X(:,1), X(:,2), ('abcdefghij'))
text(Y(:,1), Y(:,2), ('abcdefghij'))
text(Z(:,1), Z(:,2), ('abcdefghij'))
legend('X = Target', 'Y = Comparison', ...
       'Z = Transformed', 'location', 'SW')
xlim([0 65]);
ylim([0 55]);

```



### Examine the Similarity of the Two Shapes

Use two different numerical values, the dissimilarity measure  $d$  and the scaling measure  $b$ , to assess the similarity of the target shape and the transformed shape.

The dissimilarity measure  $d$  gives a number between 0 and 1 describing the difference between the target shape and the transformed shape. Values near 0 imply more similar shapes, while values near 1 imply dissimilarity.

```
d
```

```
d =
```

```
0.1502
```

The small value of  $d$  in this case shows that the two shapes are similar. `procrustes` calculates  $d$  by comparing the sum of squared deviations between the set of points with the sum of squared deviations of the original points from their column means.

```
numerator = sum(sum((X-Z).^2))
denominator = sum(sum(bsxfun(@minus,X,mean(X)).^2))
ratio = numerator/denominator
```

```
numerator =
```

```
166.5321
```

```
denominator =
```

```
1.1085e+03
```

```
ratio =
```

```
0.1502
```

The resulting measure  $d$  is independent of the scale of the size of the shapes and takes into account only the similarity of landmark data.

Examine the size similarity of the shapes.

```
tr.b
```

```
ans =
```

```
0.9291
```

The sizes of the target and comparison shapes in the previous figure appear similar. This visual impression is reinforced by the value of  $b = \% 0.93$ , which implies that the best transformation results in shrinking the comparison shape by a factor .93 (only 7%).

### Restrict the Form of the Transformations

Explore the effects of manually adjusting the scaling and reflection coefficients.

Force  $b$  to equal 1 (set 'Scaling' to false) to examine the amount of dissimilarity in size of the target and transformed figures.

```
ds = procrustes(X,Y, 'Scaling', false)
```

```
ds =
```

```
0.1552
```

In this case, setting 'Scaling' to false increases the calculated value of  $d$  only 0.0049, which further supports the similarity in the size of the two number threes. A larger increase in  $d$  would have indicated a greater size discrepancy.

This example requires only a rotation, not a reflection, to align the shapes. You can show this by observing that the determinant of the matrix  $T$  is 1 in this analysis.

```
det(tr.T)
```

```
ans =
```

```
1.0000
```

If you need a reflection in the transformation, the determinant of  $T$  is -1. You can force a reflection into the transformation as follows.

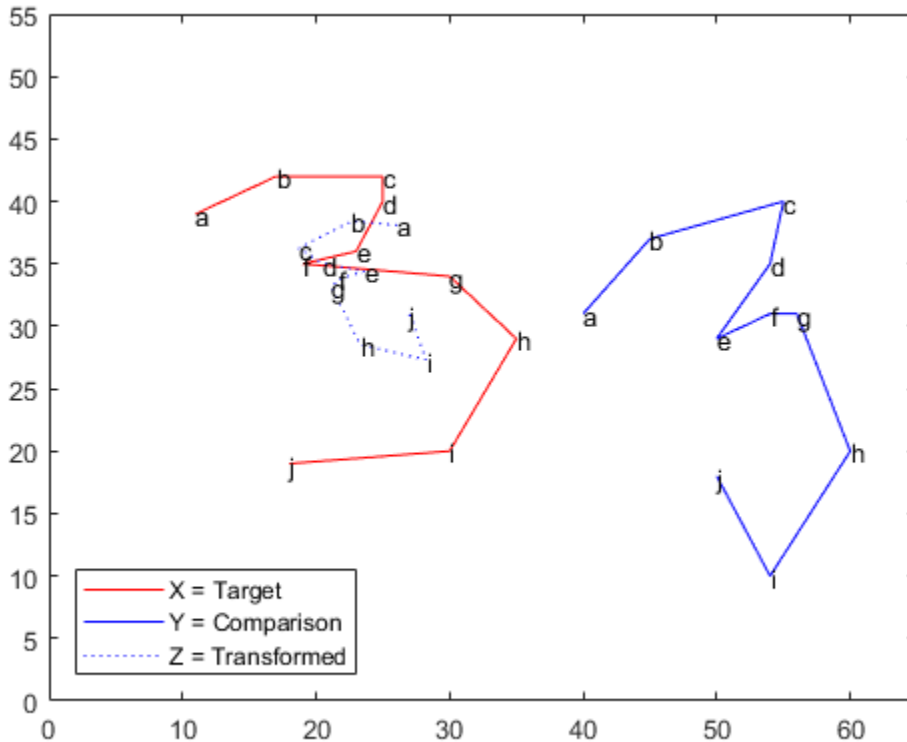
```
[dr,Zr,trr] = procrustes(X,Y, 'Reflection', true);  
dr
```

```
dr =  
  
0.8130
```

The *d* value increases dramatically, indicating that a forced reflection leads to a poor transformation of the landmark points. A plot of the transformed shape shows a similar result.

```
plot(X(:,1), X(:,2), 'r-', Y(:,1), Y(:,2), 'b-', ...  
Zr(:,1), Zr(:,2), 'b:');  
text(X(:,1), X(:,2), ('abcdefghij'))  
text(Y(:,1), Y(:,2), ('abcdefghij'))  
text(Zr(:,1), Zr(:,2), ('abcdefghij'))  
legend('X = Target', 'Y = Comparison', ...  
'Z = Transformed', 'Location', 'SW')  
xlim([0 65]);  
ylim([0 55]);
```





The landmark data points are now further away from their target counterparts. The transformed three is now an undesirable mirror image of the target three.

It appears that the shapes might be better matched if you flipped the transformed shape upside down. Flipping the shapes would make the transformation even worse, however, because the landmark data points would be further away from their target counterparts. From this example, it is clear that manually adjusting the scaling and reflection parameters is generally not optimal.

## See Also

`procrustes`

## **More About**

- “Procrustes Analysis” on page 15-63

# Feature Selection

<b>In this section...</b>
“Introduction to Feature Selection” on page 15-73
“Sequential Feature Selection” on page 15-73

## Introduction to Feature Selection

Feature selection reduces the dimensionality of data by selecting only a subset of measured features (predictor variables) to create a model. Selection criteria usually involve the minimization of a specific measure of predictive error for models fit to different subsets. Algorithms search for a subset of predictors that optimally model measured responses, subject to constraints such as required or excluded features and the size of the subset.

Feature selection is preferable to feature transformation on page 15-78 when the original units and meaning of features are important and the modeling goal is to identify an influential subset. When categorical features are present, and numerical transformations are inappropriate, feature selection becomes the primary means of dimension reduction.

## Sequential Feature Selection

A common method of feature selection is sequential feature selection. This method has two components:

- An objective function, called the criterion, which the method seeks to minimize over all feasible feature subsets. Common criteria are mean squared error (for regression models) and misclassification rate (for classification models).
- A sequential search algorithm, which adds or removes features from a candidate subset while evaluating the criterion. Since an exhaustive comparison of the criterion value at all  $2^n$  subsets of an  $n$ -feature data set is typically infeasible (depending on the size of  $n$  and the cost of objective calls), sequential searches move in only one direction, always growing or always shrinking the candidate set.

The method has two variants:

- Sequential forward selection (SFS), in which features are sequentially added to an empty candidate set until the addition of further features does not decrease the criterion.
- Sequential backward selection (SBS), in which features are sequentially removed from a full candidate set until the removal of further features increase the criterion.

Stepwise regression is a sequential feature selection technique designed specifically for least-squares fitting. The functions `stepwise` and `stepwisefit` make use of optimizations that are only possible with least-squares criteria. Unlike generalized sequential feature selection, stepwise regression may remove features that have been added or add features that have been removed.

The Statistics and Machine Learning Toolbox function `sequentialfs` carries out sequential feature selection. Input arguments include predictor and response data and a function handle to a file implementing the criterion function. Optional inputs allow you to specify SFS or SBS, required or excluded features, and the size of the feature subset. The function calls `cvpartition` and `crossval` to evaluate the criterion at different candidate sets.

## See Also

`sequentialfs`

## Related Examples

- “Select Subset of Features with Comparative Predictive Power” on page 15-75

## Select Subset of Features with Comparative Predictive Power

For example, consider a data set with 100 observations of 10 predictors. The following generates random data from a logistic model, with a binomial distribution of responses at each set of values for the predictors. Some coefficients are set to zero so that not all of the predictors affect the response:

```
n = 100;
m = 10;
X = rand(n,m);
b = [1 0 0 2 .5 0 0 0.1 0 1];
Xb = X*b';
p = 1./(1+exp(-Xb));
N = 50;
y = binornd(N,p);
```

The `glmfit` function fits a logistic model to the data:

```
Y = [y N*ones(size(y))];
[b0,dev0,stats0] = glmfit(X,Y,'binomial');

% Display coefficient estimates and their standard errors:
model0 = [b0 stats0.se]
model0 =
    0.3115    0.2596
    0.9614    0.1656
   -0.1100    0.1651
   -0.2165    0.1683
    1.9519    0.1809
    0.5683    0.2018
   -0.0062    0.1740
    0.0651    0.1641
   -0.1034    0.1685
    0.0017    0.1815
    0.7979    0.1806

% Display the deviance of the fit:
dev0
dev0 =
    101.2594
```

This is the full model, using all of the features (and an initial constant term). Sequential feature selection searches for a subset of the features in the full model with comparative predictive power.

First, you must specify a criterion for selecting the features. The following function, which calls `glmfit` and returns the deviance of the fit (a generalization of the residual sum of squares) is a useful criterion in this case:

```
function dev = critfun(X,Y)

[b,dev] = glmfit(X,Y,'binomial');
```

You should create this function as a file on the MATLAB path.

The function `sequentialfs` performs feature selection, calling the criterion function via a function handle:

```
maxdev = chi2inv(.95,1);
opt = statset('display','iter',...
             'TolFun',maxdev,...
             'TolTypeFun','abs');

inmodel = sequentialfs(@critfun,X,Y,...
                      'cv','none',...
                      'nullmodel',true,...
                      'options',opt,...
                      'direction','forward');
```

```
Start forward sequential feature selection:
Initial columns included: none
Columns that cannot be included: none
Step 1, used initial columns, criterion value 309.118
Step 2, added column 4, criterion value 180.732
Step 3, added column 1, criterion value 138.862
Step 4, added column 10, criterion value 114.238
Step 5, added column 5, criterion value 103.503
Final columns included: 1 4 5 10
```

The iterative display shows a decrease in the criterion value as each new feature is added to the model. The final result is a reduced model with only four of the original ten features: columns 1, 4, 5, and 10 of `X`. These features are indicated in the logical vector `inmodel` returned by `sequentialfs`.

The deviance of the reduced model is higher than for the full model, but the addition of any other single feature would not decrease the criterion by more than the absolute tolerance, `maxdev`, set in the options structure. Adding a feature with no effect reduces the deviance by an amount that has a chi-square distribution with one degree of freedom.

Adding a significant feature results in a larger change. By setting `maxdev` to `chi2inv(.95,1)`, you instruct `sequentialfs` to continue adding features so long as the change in deviance is more than would be expected by random chance.

The reduced model (also with an initial constant term) is:

```
[b,dev,stats] = glmfit(X(:,inmodel),Y,'binomial');

% Display coefficient estimates and their standard errors:
model = [b stats.se]
model =
    0.0784    0.1642
    1.0040    0.1592
    1.9459    0.1789
    0.6134    0.1872
    0.8245    0.1730
```

## See Also

`sequentialfs`

## More About

- “Feature Selection” on page 15-73

## Feature Transformation

Feature transformation is a group of methods that create new features (predictor variables). The methods are useful for dimension reduction when the transformed features have a descriptive power that is more easily ordered than the original features. In this case, less descriptive features can be dropped from consideration when building models.

Feature transformation methods are contrasted with the methods presented in “Feature Selection” on page 15-73, where dimension reduction is achieved by computing an optimal subset of predictive features measured in the original data.

The methods presented in this section share some common methodology. Their goals, however, are essentially different:

- Nonnegative matrix factorization on page 15-79 is used when model terms must represent nonnegative quantities, such as physical quantities.
- Principal component analysis on page 15-83 is used to summarize data in fewer dimensions, for example, to visualize it.
- Factor analysis on page 15-97 is used to build explanatory models of data correlations.



## Nonnegative Matrix Factorization

Nonnegative matrix factorization (NMF) is a dimension-reduction technique based on a low-rank approximation of the feature space. Besides providing a reduction in the number of features, NMF guarantees that the features are nonnegative, producing additive models that respect, for example, the nonnegativity of physical quantities.

Given a nonnegative  $m$ -by- $n$  matrix  $X$  and a positive integer  $k < \min(m, n)$ , NMF finds nonnegative  $m$ -by- $k$  and  $k$ -by- $n$  matrices  $W$  and  $H$ , respectively, that minimize the norm of the difference  $X - WH$ .  $W$  and  $H$  are thus approximate nonnegative factors of  $X$ .

The  $k$  columns of  $W$  represent transformations of the variables in  $X$ ; the  $k$  rows of  $H$  represent the coefficients of the linear combinations of the original  $n$  variables in  $X$  that produce the transformed variables in  $W$ . Since  $k$  is generally smaller than the rank of  $X$ , the product  $WH$  provides a compressed approximation of the data in  $X$ . A range of possible values for  $k$  is often suggested by the modeling context.

The Statistics and Machine Learning Toolbox function `nnmf` carries out nonnegative matrix factorization. `nnmf` uses one of two iterative algorithms that begin with random initial values for  $W$  and  $H$ . Because the norm of the residual  $X - WH$  may have local minima, repeated calls to `nnmf` may yield different factorizations. Sometimes the algorithm converges to a solution of lower rank than  $k$ , which may indicate that the result is not optimal.

### See Also

`nnmf`

### Related Examples

- “Perform Nonnegative Matrix Factorization” on page 15-80

## Perform Nonnegative Matrix Factorization

This example shows how to perform nonnegative matrix factorization.

Load the sample data.

```
load moore
X = moore(:,1:5);
rng('default'); % For reproducibility
```

Compute a rank-two approximation of X using a multiplicative update algorithm that begins from five random initial values for W and H.

```
opt = statset('MaxIter',10,'Display','final');
[W0,H0] = nnmf(X,2,'replicates',5,'options',opt,'algorithm','mult');
```

rep	iteration	rms resid	delta x
1	10	358.296	0.00190554
2	10	78.3556	0.000351747
3	10	230.962	0.0172839
4	10	326.347	0.00739552
5	10	361.547	0.00705539

Final root mean square residual = 78.3556

The 'mult' algorithm is sensitive to initial values, which makes it a good choice when using 'replicates' to find W and H from multiple random starting values.

Now perform the factorization using alternating least-squares algorithm, which converges faster and more consistently. Run 100 times more iterations, beginning from the initial W0 and H0 identified above.

```
opt = statset('Maxiter',1000,'Display','final');
[W,H] = nnmf(X,2,'w0',W0,'h0',H0,'options',opt,'algorithm','als');
```

rep	iteration	rms resid	delta x
1	2	77.5315	0.000830334

Final root mean square residual = 77.5315

The two columns of W are the transformed predictors. The two rows of H give the relative contributions of each of the five predictors in X to the predictors in W. Display H.

H

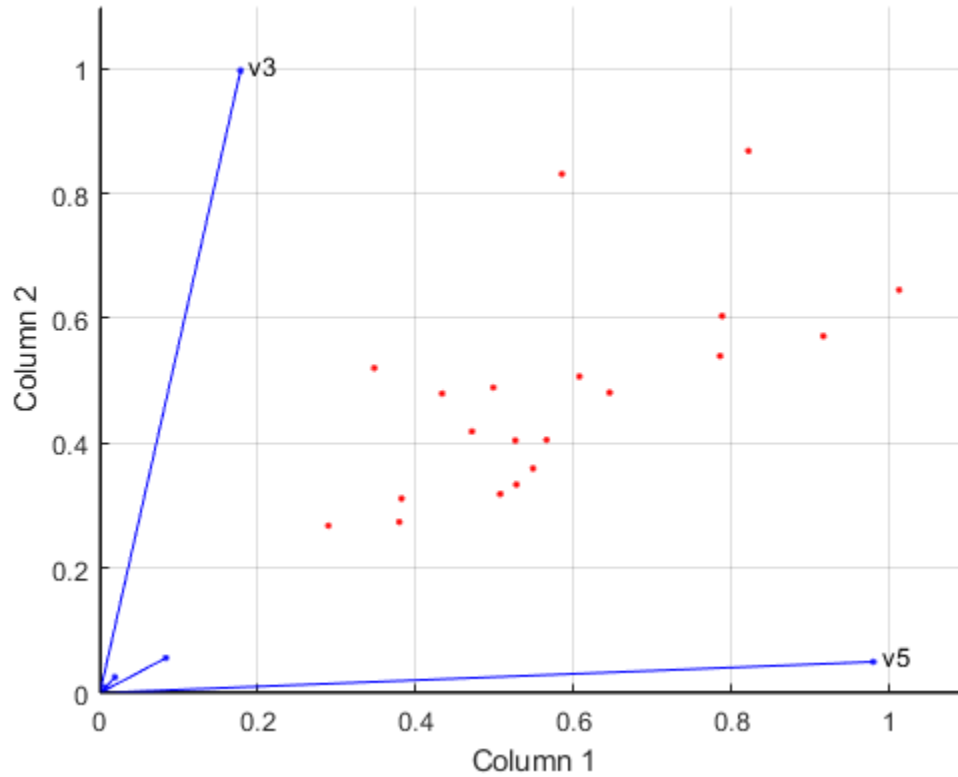
H =

```
0.0835    0.0190    0.1782    0.0072    0.9802
0.0559    0.0250    0.9969    0.0085    0.0497
```

The fifth predictor in  $X$  (weight 0.9802) strongly influences the first predictor in  $W$ . The third predictor in  $X$  (weight 0.9969) strongly influences the second predictor in  $W$ .

Visualize the relative contributions of the predictors in  $X$  with `biplot`, showing the data and original variables in the column space of  $W$ .

```
biplot(H, 'scores', W, 'varlabels', {'', '', 'v3', '', 'v5'});
axis([0 1.1 0 1.1])
xlabel('Column 1')
ylabel('Column 2')
```



## See Also

nmmf

## More About

- “Nonnegative Matrix Factorization” on page 15-79

## Principal Component Analysis (PCA)

One of the difficulties inherent in multivariate statistics is the problem of visualizing data that has many variables. The MATLAB function `plot` displays a graph of the relationship between two variables. The `plot3` and `surf` commands display different three-dimensional views. But when there are more than three variables, it is more difficult to visualize their relationships.

Fortunately, in data sets with many variables, groups of variables often move together. One reason for this is that more than one variable might be measuring the same driving principle governing the behavior of the system. In many systems there are only a few such driving forces. But an abundance of instrumentation enables you to measure dozens of system variables. When this happens, you can take advantage of this redundancy of information. You can simplify the problem by replacing a group of variables with a single new variable.

Principal component analysis is a quantitatively rigorous method for achieving this simplification. The method generates a new set of variables, called *principal components*. Each principal component is a linear combination of the original variables. All the principal components are orthogonal to each other, so there is no redundant information. The principal components as a whole form an orthogonal basis for the space of the data.

There are an infinite number of ways to construct an orthogonal basis for several columns of data. What is so special about the principal component basis?

The first principal component is a single axis in space. When you project each observation on that axis, the resulting values form a new variable. And the variance of this variable is the maximum among all possible choices of the first axis.

The second principal component is another axis in space, perpendicular to the first. Projecting the observations on this axis generates another new variable. The variance of this variable is the maximum among all possible choices of this second axis.

The full set of principal components is as large as the original set of variables. But it is commonplace for the sum of the variances of the first few principal components to exceed 80% of the total variance of the original data. By examining plots of these few new variables, researchers often develop a deeper understanding of the driving forces that generated the original data.

You can use the function `pca` to find the principal components. To use `pca`, you need to have the actual measured data you want to analyze. However, if you lack the actual data,

but have the sample covariance or correlation matrix for the data, you can still use the function `pcacov` to perform a principal components analysis. See the reference page for `pcacov` for a description of its inputs and outputs.

### See Also

`pca` | `pcacov` | `pcares` | `ppca`

### Related Examples

- “Analyze Quality of Life in U.S. Cities Using PCA” on page 15-85

## Analyze Quality of Life in U.S. Cities Using PCA

This example shows how to perform a weighted principal components analysis and interpret the results.

### Load sample data.

Load the sample data. The data includes ratings for 9 different indicators of the quality of life in 329 U.S. cities. These are climate, housing, health, crime, transportation, education, arts, recreation, and economics. For each category, a higher rating is better. For example, a higher rating for crime means a lower crime rate.

Display the `categories` variable.

```
load cities
categories

categories =
  climate
  housing
  health
  crime
  transportation
  education
  arts
  recreation
  economics
```

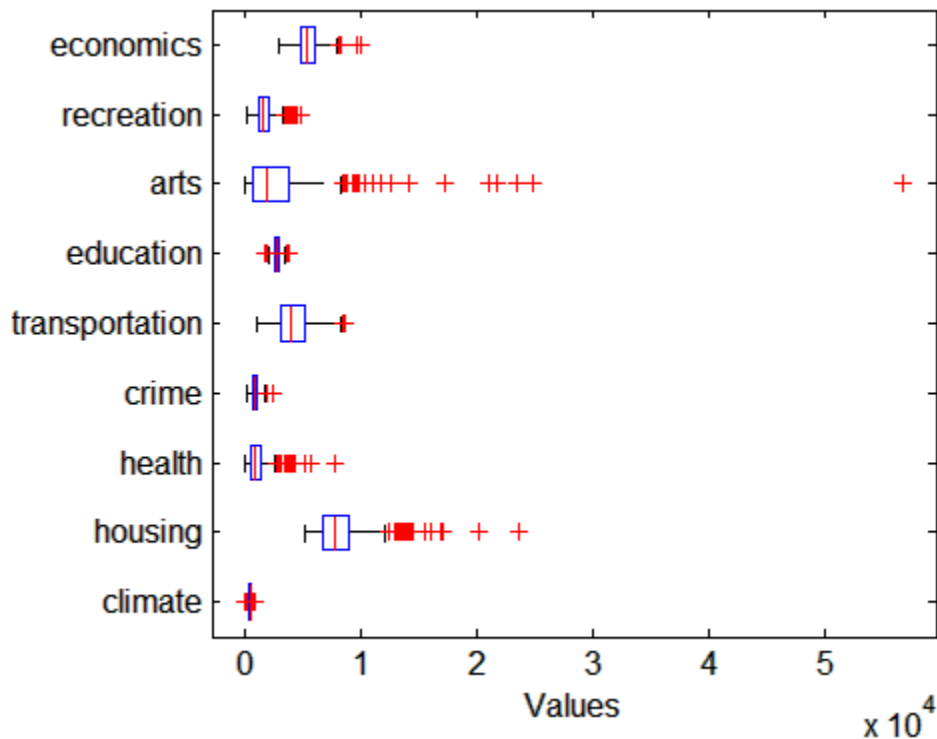
In total, the `cities` data set contains three variables:

- `categories`, a character matrix containing the names of the indices
- `names`, a character matrix containing the 329 city names
- `ratings`, the data matrix with 329 rows and 9 columns

### Plot data.

Make a boxplot to look at the distribution of the `ratings` data.

```
figure()
boxplot(ratings, 'orientation', 'horizontal', 'labels', categories)
```



There is more variability in the ratings of the arts and housing than in the ratings of crime and climate.

#### Check pairwise correlation.

Check the pairwise correlation between the variables.

```
C = corr(ratings, ratings);
```

The correlation among some variables is as high as 0.85. Principal components analysis constructs independent new variables which are linear combinations of the original variables.

#### Compute principal components.

When all variables are in the same unit, it is appropriate to compute principal components for raw data. When the variables are in different units or the difference in



the variance of different columns is substantial (as in this case), scaling of the data or use of weights is often preferable.

Perform the principal component analysis by using the inverse variances of the ratings as weights.

```
w = 1./var(ratings);
[wcoeff,score,latent,tsquared,explained] = pca(ratings,...
'VariableWeights',w);
```

Or equivalently:

```
[wcoeff,score,latent,tsquared,explained] = pca(ratings,...
'VariableWeights','variance');
```

The following sections explain the five outputs of `pca`.

### Component coefficients.

The first output, `wcoeff`, contains the coefficients of the principal components.

The first three principal component coefficient vectors are:

```
c3 = wcoeff(:,1:3)
c3 = wcoeff(:,1:3)
c3 =
1.0e+03 *
0.0249    -0.0263    -0.0834
0.8504    -0.5978    -0.4965
0.4616     0.3004    -0.0073
0.1005    -0.1269     0.0661
0.5096     0.2606     0.2124
0.0883     0.1551     0.0737
2.1496     0.9043    -0.1229
0.2649    -0.3106    -0.0411
0.1469    -0.5111     0.6586
```

These coefficients are weighted, hence the coefficient matrix is not orthonormal.

### Transform coefficients.

Transform the coefficients so that they are orthonormal.

```
coefforth = inv(diag(std(ratings))) * wcoeff;
```

Note that if you use a weights vector,  $w$ , while conducting the `pca`, then

```
coefforth = diag(sqrt(w)) * wcoeff;
```

### Check coefficients are orthonormal.

The transformed coefficients are now orthonormal.

```
I = coefforth' * coefforth;
I(1:3,1:3)

ans =
    1.0000    -0.0000    -0.0000
   -0.0000     1.0000    -0.0000
   -0.0000    -0.0000     1.0000
```

### Component scores.

The second output, `score`, contains the coordinates of the original data in the new coordinate system defined by the principal components. The `score` matrix is the same size as the input data matrix. You can also obtain the component scores using the orthonormal coefficients and the standardized ratings as follows.

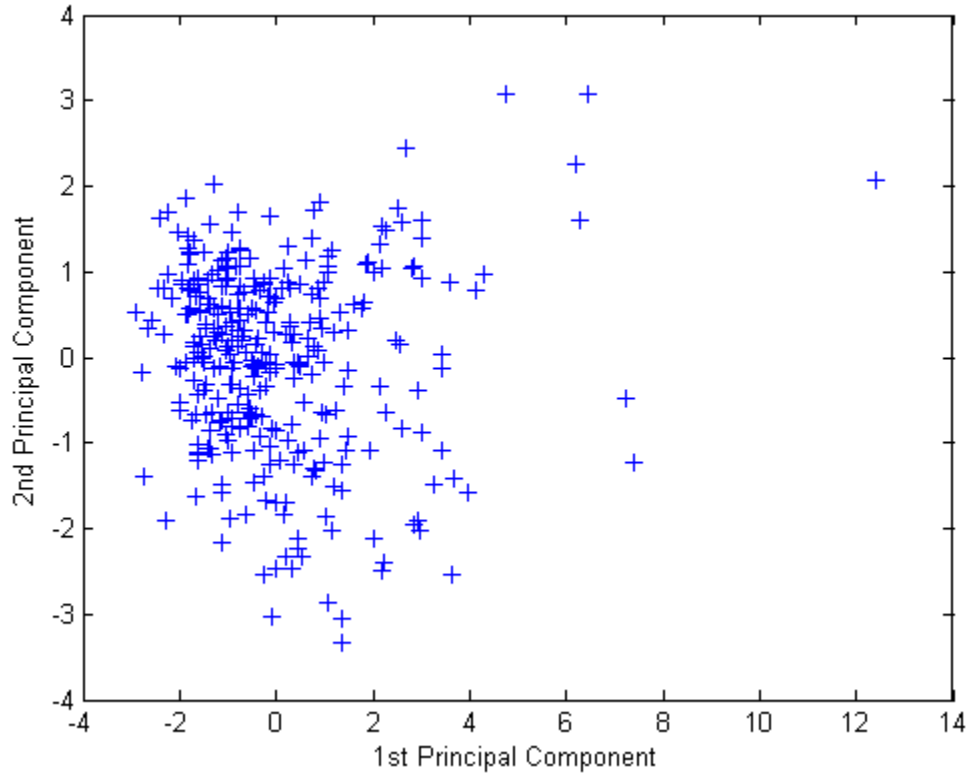
```
cscores = zscore(ratings) * coefforth;
```

`cscores` and `score` are identical matrices.

### Plot component scores.

Create a plot of the first two columns of `score`.

```
figure()
plot(score(:,1), score(:,2), '+')
xlabel('1st Principal Component')
ylabel('2nd Principal Component')
```



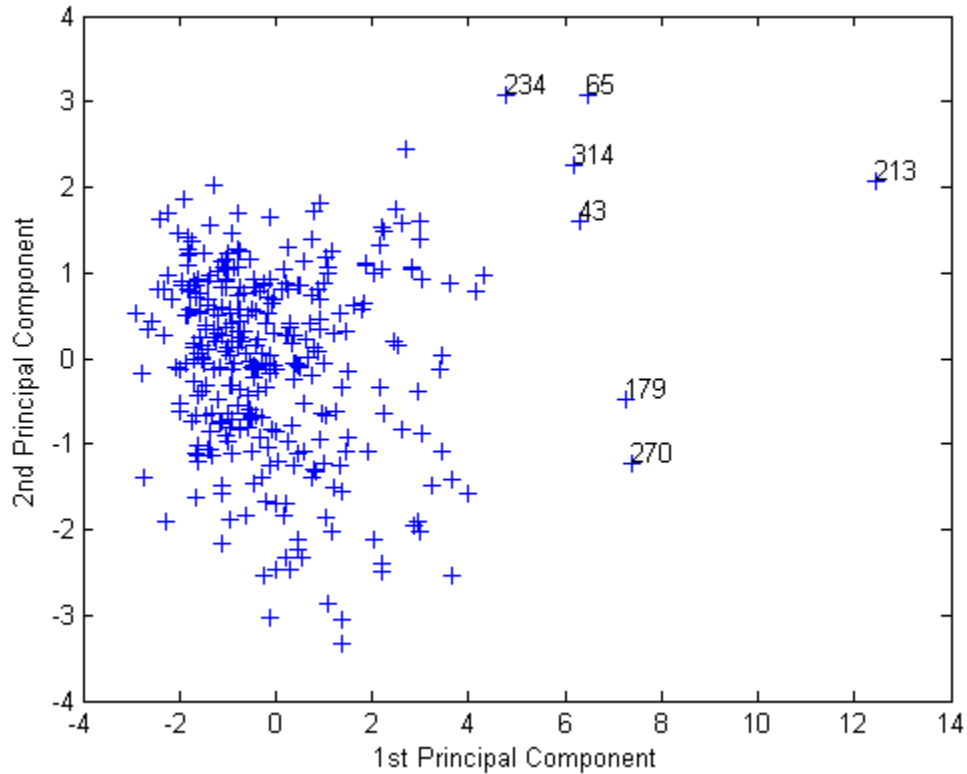
This plot shows the centered and scaled ratings data projected onto the first two principal components. `pca` computes the scores to have mean zero.

**Explore plot interactively.**

Note the outlying points in the right half of the plot. You can graphically identify these points as follows.

```
gname
```

Move your cursor over the plot and click once near the rightmost seven points. This labels the points by their row numbers as in the following figure.



After labeling points, press **Return**.

**Extract observation names.**

Create an index variable containing the row numbers of all the cities you chose and get the names of the cities.

```
metro = [43 65 179 213 234 270 314];  
names(metro,:)  
  
ans =  
Boston, MA  
Chicago, IL  
Los Angeles, Long Beach, CA
```

```
New York, NY
Philadelphia, PA-NJ
San Francisco, CA
Washington, DC-MD-VA
```

These labeled cities are some of the biggest population centers in the United States and they appear more extreme than the remainder of the data.

### **Component variances.**

The third output, `latent`, is a vector containing the variance explained by the corresponding principal component. Each column of `score` has a sample variance equal to the corresponding row of `latent`.

```
latent
latent =
    3.4083
    1.2140
    1.1415
    0.9209
    0.7533
    0.6306
    0.4930
    0.3180
    0.1204
```

### **Percent variance explained.**

The fifth output, `explained`, is a vector containing the percent variance explained by the corresponding principal component.

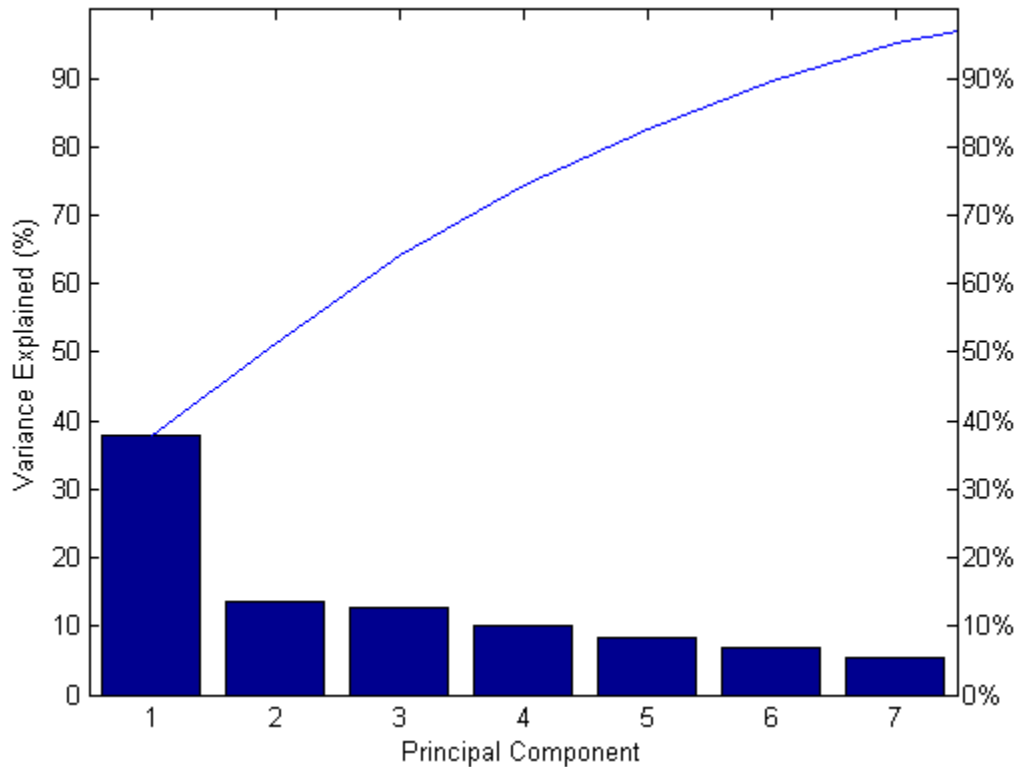
```
explained
explained =
    37.8699
    13.4886
    12.6831
    10.2324
     8.3698
     7.0062
     5.4783
```

```
3.5338  
1.3378
```

### Create scree plot.

Make a scree plot of the percent variability explained by each principal component.

```
figure()  
pareto(explained)  
xlabel('Principal Component')  
ylabel('Variance Explained (%)')
```



This scree plot only shows the first seven (instead of the total nine) components that explain 95% of the total variance. The only clear break in the amount of variance

accounted for by each component is between the first and second components. However, the first component by itself explains less than 40% of the variance, so more components might be needed. You can see that the first three principal components explain roughly two-thirds of the total variability in the standardized ratings, so that might be a reasonable way to reduce the dimensions.

### Hotelling's T-squared statistic.

The last output from `pca` is `tsquared`, which is Hotelling's  $T^2$ , a statistical measure of the multivariate distance of each observation from the center of the data set. This is an analytical way to find the most extreme points in the data.

```
[st2,index] = sort(tsquared,'descend'); % sort in descending order
extreme = index(1);
names(extreme,:)
```

```
ans =
```

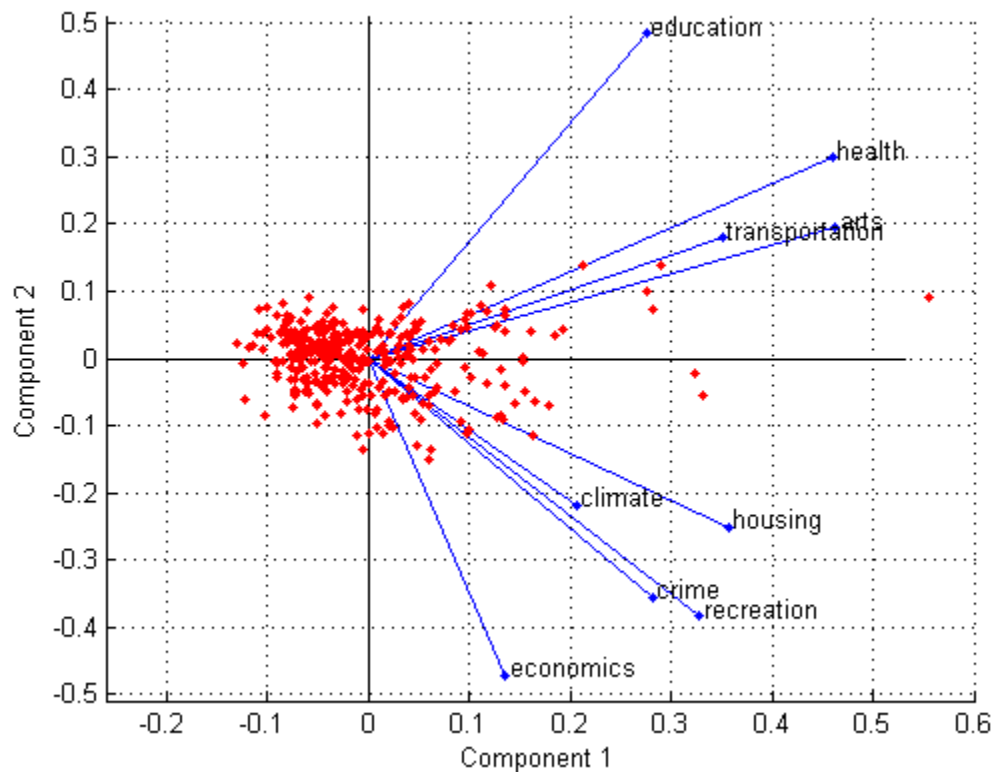
```
New York, NY
```

The ratings for New York are the furthest from the average U.S. city.

### Visualize the results.

Visualize both the orthonormal principal component coefficients for each variable and the principal component scores for each observation in a single plot.

```
biplot(coefforth(:,1:2),'scores',score(:,1:2),'varlabels',categories);
axis([-0.26 0.6 -0.51 0.51]);
```



All nine variables are represented in this bi-plot by a vector, and the direction and length of the vector indicate how each variable contributes to the two principal components in the plot. For example, the first principal component, on the horizontal axis, has positive coefficients for all nine variables. That is why the nine vectors are directed into the right half of the plot. The largest coefficients in the first principal component are the third and seventh elements, corresponding to the variables *health* and *arts*.

The second principal component, on the vertical axis, has positive coefficients for the variables *education*, *health*, *arts*, and *transportation*, and negative coefficients for the remaining five variables. This indicates that the second component distinguishes among cities that have high values for the first set of variables and low for the second, and cities that have the opposite.



The variable labels in this figure are somewhat crowded. You can either exclude the `VarLabels` parameter when making the plot, or select and drag some of the labels to better positions using the Edit Plot tool from the figure window toolbar.

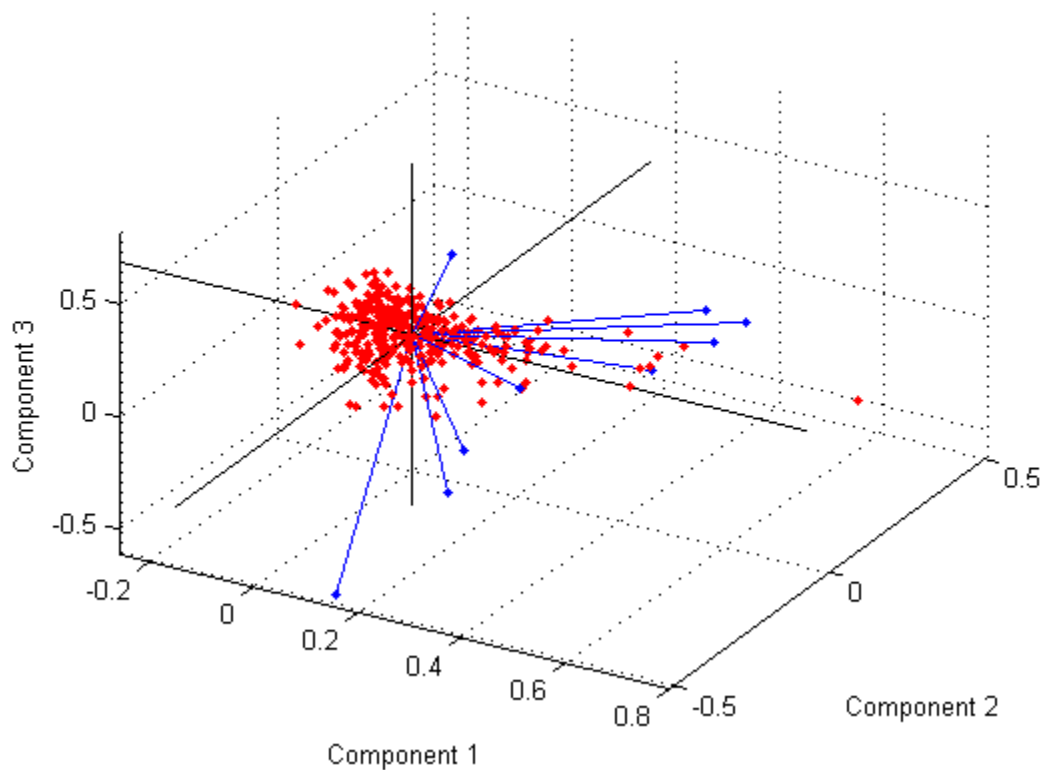
This 2-D bi-plot also includes a point for each of the 329 observations, with coordinates indicating the score of each observation for the two principal components in the plot. For example, points near the left edge of this plot have the lowest scores for the first principal component. The points are scaled with respect to the maximum score value and maximum coefficient length, so only their relative locations can be determined from the plot.

You can identify items in the plot by selecting **Tools>Data Cursor** from the figure window. By clicking a variable (vector), you can read that variable's coefficients for each principal component. By clicking an observation (point), you can read that observation's scores for each principal component.

### Create a three-dimensional bi-plot.

You can also make a bi-plot in three dimensions.

```
figure()
biplot(coefforth(:,1:3), 'scores', score(:,1:3), 'obslabels', names);
axis([-0.26 0.8 -0.51 0.51 -0.61 0.81]);
view([30 40]);
```



This graph is useful if the first two principal coordinates do not explain enough of the variance in your data. You can also rotate the figure to see it from different angles by selecting the **Tools** > Rotate 3D.

## See Also

`pca` | `pcacov` | `pcarets` | `ppca`

## More About

- “Principal Component Analysis (PCA)” on page 15-83

## Factor Analysis

Multivariate data often includes a large number of measured variables, and sometimes those variables overlap, in the sense that groups of them might be dependent. For example, in a decathlon, each athlete competes in 10 events, but several of them can be thought of as speed events, while others can be thought of as strength events, etc. Thus, you can think of a competitor's 10 event scores as largely dependent on a smaller set of three or four types of athletic ability.

Factor analysis is a way to fit a model to multivariate data to estimate just this sort of interdependence. In a factor analysis model, the measured variables depend on a smaller number of unobserved (latent) factors. Because each factor might affect several variables in common, they are known as common factors. Each variable is assumed to be dependent on a linear combination of the common factors, and the coefficients are known as loadings. Each measured variable also includes a component due to independent random variability, known as specific variance because it is specific to one variable.

Specifically, factor analysis assumes that the covariance matrix of your data is of the form

$$\Sigma_x = \Lambda\Lambda^T + \Psi$$

where  $\Lambda$  is the matrix of loadings, and the elements of the diagonal matrix  $\Psi$  are the specific variances. The function `factoran` fits the Factor Analysis model using maximum likelihood.

### See Also

`factoran`

### Related Examples

- “Analyze Stock Prices Using Factor Analysis” on page 15-98

## Analyze Stock Prices Using Factor Analysis

This example shows how to analyze if companies within the same sector experience similar week-to-week changes in stock price.

### Factor Loadings

Load the sample data.

```
load stockreturns
```

Suppose that over the course of 100 weeks, the percent change in stock prices for ten companies has been recorded. Of the ten companies, the first four can be classified as primarily technology, the next three as financial, and the last three as retail. It seems reasonable that the stock prices for companies that are in the same sector might vary together as economic conditions change. Factor analysis can provide quantitative evidence.

First specify a model fit with three common factors. By default, `factoran` computes rotated estimates of the loadings to try and make their interpretation simpler. But in this example, specify an unrotated solution.

```
[Loadings,specificVar,T,stats] = factoran(stocks,3,'rotate','none');
```

The first two `factoran` output arguments are the estimated loadings and the estimated specific variances. Each row of the loadings matrix represents one of the ten stocks, and each column corresponds to a common factor. With unrotated estimates, interpretation of the factors in this fit is difficult because most of the stocks contain fairly large coefficients for two or more factors.

Loadings

```
Loadings =
```

```
0.8885    0.2367   -0.2354
0.7126    0.3862    0.0034
0.3351    0.2784   -0.0211
0.3088    0.1113   -0.1905
0.6277   -0.6643    0.1478
0.4726   -0.6383    0.0133
0.1133   -0.5416    0.0322
0.6403    0.1669    0.4960
```

```
0.2363    0.5293    0.5770
0.1105    0.1680    0.5524
```

Factor rotation helps to simplify the structure in the `Loadings` matrix, to make it easier to assign meaningful interpretations to the factors.

From the estimated specific variances, you can see that the model indicates that a particular stock price varies quite a lot beyond the variation due to the common factors. Display estimated specific variances.

```
specificVar

specificVar =

    0.0991
    0.3431
    0.8097
    0.8559
    0.1429
    0.3691
    0.6928
    0.3162
    0.3311
    0.6544
```

A specific variance of 1 would indicate that there is no common factor component in that variable, while a specific variance of 0 would indicate that the variable is entirely determined by common factors. These data seem to fall somewhere in between.

Display the  $p$ -value.

```
stats.p

ans =

    0.8144
```

The  $p$ -value returned in the `stats` structure fails to reject the null hypothesis of three common factors, suggesting that this model provides a satisfactory explanation of the covariation in these data.

Fit a model with two common factors to determine whether fewer than three factors can provide an acceptable fit.

```
[Loadings2,specificVar2,T2,stats2] = factoran(stocks, 2, 'rotate', 'none');
```

Display the  $p$ -value.

```
stats2.p  
  
ans =  
  
    3.5610e-06
```

The  $p$ -value for this second fit is highly significant, and rejects the hypothesis of two factors, indicating that the simpler model is not sufficient to explain the pattern in these data.

### Factor Rotation

As the results illustrate, the estimated loadings from an unrotated factor analysis fit can have a complicated structure. The goal of factor rotation is to find a parameterization in which each variable has only a small number of large loadings. That is, each variable is affected by a small number of factors, preferably only one. This can often make it easier to interpret what the factors represent.

If you think of each row of the loadings matrix as coordinates of a point in  $M$ -dimensional space, then each factor corresponds to a coordinate axis. Factor rotation is equivalent to rotating those axes and computing new loadings in the rotated coordinate system. There are various ways to do this. Some methods leave the axes orthogonal, while others are oblique methods that change the angles between them. For this example, you can rotate the estimated loadings by using the promax criterion, a common oblique method.

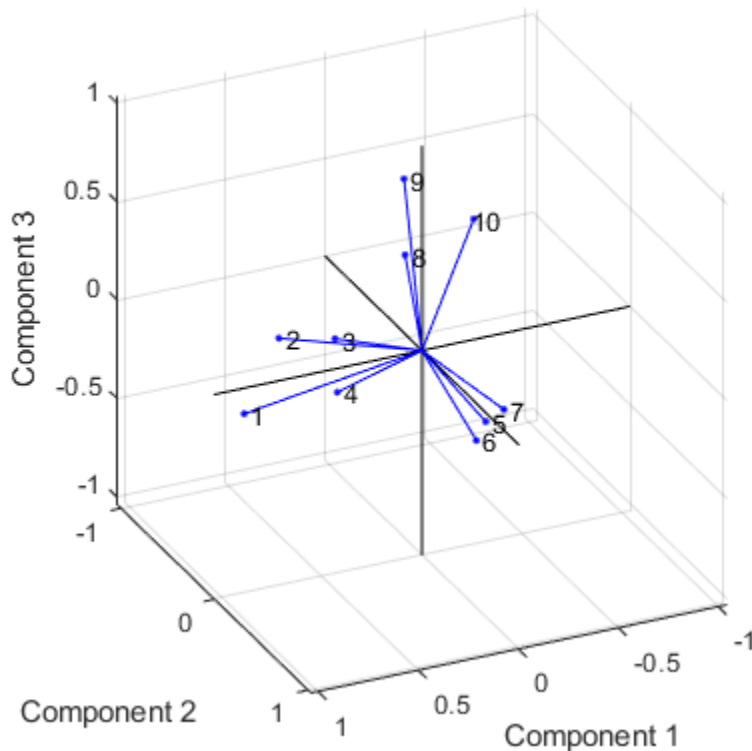
```
[LoadingsPM,specVarPM] = factoran(stocks,3, 'rotate', 'promax');  
LoadingsPM
```

```
LoadingsPM =  
  
    0.9452    0.1214   -0.0617  
    0.7064   -0.0178    0.2058  
    0.3885   -0.0994    0.0975  
    0.4162   -0.0148   -0.1298
```

0.1021	0.9019	0.0768
0.0873	0.7709	-0.0821
-0.1616	0.5320	-0.0888
0.2169	0.2844	0.6635
0.0016	-0.1881	0.7849
-0.2289	0.0636	0.6475

Promax rotation creates a simpler structure in the loadings, one in which most of the stocks have a large loading on only one factor. To see this structure more clearly, you can use the biplot function to plot each stock using its factor loadings as coordinates.

```
biplot(LoadingsPM, 'varlabels', num2str((1:10)'));  
axis square  
view(155,27);
```



This plot shows that promax has rotated the factor loadings to a simpler structure. Each stock depends primarily on only one factor, and it is possible to describe each factor in terms of the stocks that it affects. Based on which companies are near which axes, you could reasonably conclude that the first factor axis represents the financial sector, the second retail, and the third technology. The original conjecture, that stocks vary primarily within sector, is apparently supported by the data.

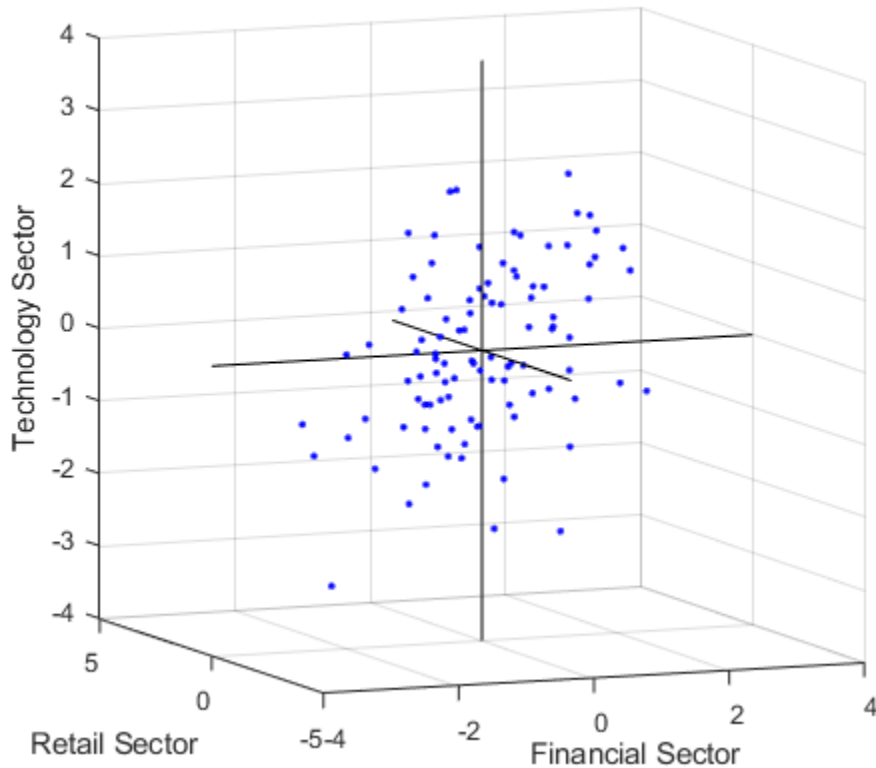
### Factor Scores

Sometimes, it is useful to be able to classify an observation based on its factor scores. For example, if you accepted the three-factor model and the interpretation of the rotated factors, you might want to categorize each week in terms of how favorable it was for each of the three stock sectors, based on the data from the 10 observed stocks. Because the



data in this example are the raw stock price changes, and not just their correlation matrix, you can have factoran return estimates of the value of each of the three rotated common factors for each week. You can then plot the estimated scores to see how the different stock sectors were affected during each week.

```
[LoadingsPM,specVarPM,TPM,stats,F] = factoran(stocks, 3, 'rotate', 'promax');  
  
plot3(F(:,1),F(:,2),F(:,3), 'b.')  
line([-4 4 NaN 0 0 NaN 0 0], [0 0 NaN -4 4 NaN 0 0],[0 0 NaN 0 0 NaN -4 4], 'Color','bl')  
xlabel('Financial Sector')  
ylabel('Retail Sector')  
zlabel('Technology Sector')  
grid on  
axis square  
view(-22.5, 8)
```



Oblique rotation often creates factors that are correlated. This plot shows some evidence of correlation between the first and third factors, and you can investigate further by computing the estimated factor correlation matrix.

```
inv(TPM'*TPM);
```

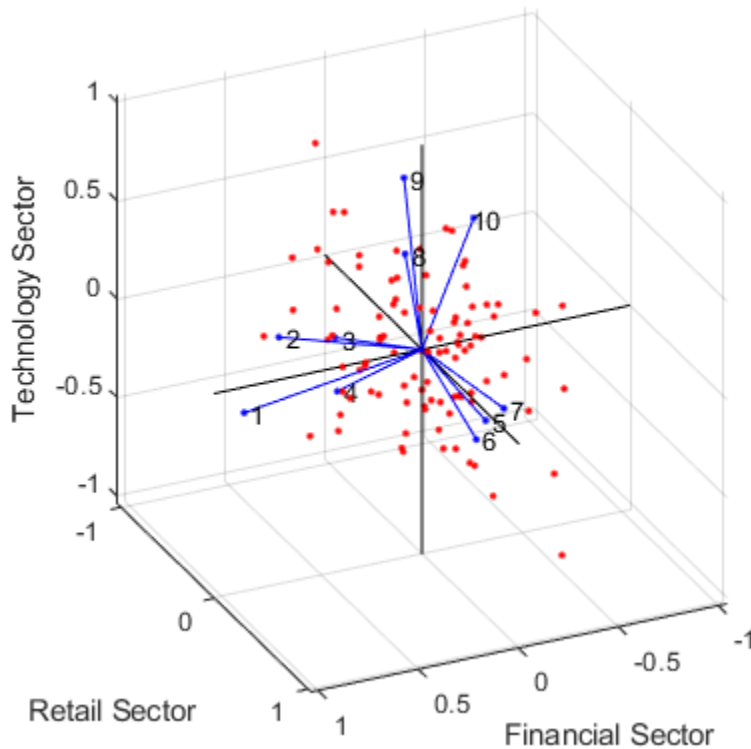
### Visualize the Results

You can use the biplot function to help visualize both the factor loadings for each variable and the factor scores for each observation in a single plot. For example, the following command plots the results from the factor analysis on the stock data and labels each of the 10 stocks.

```

biplot(LoadingsPM, 'scores', F, 'varlabels', num2str((1:10)'))
xlabel('Financial Sector')
ylabel('Retail Sector')
zlabel('Technology Sector')
axis square
view(155,27)

```



In this case, the factor analysis includes three factors, and so the biplot is three-dimensional. Each of the 10 stocks is represented in this plot by a vector, and the direction and length of the vector indicates how each stock depends on the underlying factors. For example, you have seen that after promax rotation, the first four stocks have positive loadings on the first factor, and unimportant loadings on the other two factors. That first factor, interpreted as a financial sector effect, is represented in this biplot as

one of the horizontal axes. The dependence of those four stocks on that factor corresponds to the four vectors directed approximately along that axis. Similarly, the dependence of stocks 5, 6, and 7 primarily on the second factor, interpreted as a retail sector effect, is represented by vectors directed approximately along that axis.

Each of the 100 observations is represented in this plot by a point, and their locations indicate the score of each observation for the three factors. For example, points near the top of this plot have the highest scores for the technology sector factor. The points are scaled to fit within the unit square, so only their relative locations can be determined from the plot.

You can use the **Data Cursor** tool from the **Tools** menu in the figure window to identify the items in this plot. By clicking a stock (vector), you can read off that stock's loadings for each factor. By clicking an observation (point), you can read off that observation's scores for each factor.

## Robust Feature Selection Using NCA for Regression

Perform feature selection that is robust to outliers using a custom robust loss function in NCA.

### Generate data with outliers

Generate sample data for regression where the response depends on three of the predictors, namely predictors 4, 7, and 13.

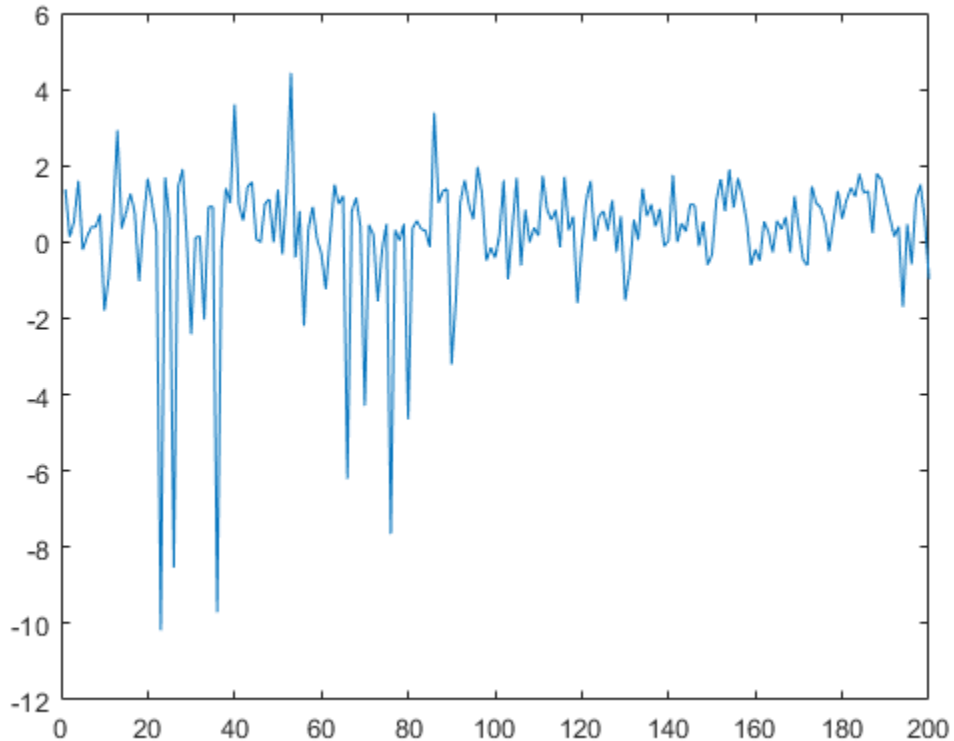
```
rng(123, 'twister'); % For reproducibility
n = 200;
X = randn(n, 20);
y = cos(X(:, 7)) + sin(X(:, 4) .* X(:, 13)) + 0.1 * randn(n, 1);
```

Add outliers to data.

```
numoutliers = 25;
outlieridx = floor(linspace(10, 90, numoutliers));
y(outlieridx) = 5 * randn(numoutliers, 1);
```

Plot the data.

```
figure;
plot(y);
```



### Use non-robust loss function

The performance of the feature selection algorithm highly depends on the value of the regularization parameter. It is a good practice to tune the regularization parameter for the best value to use in feature selection. Tune the regularization parameter using five-fold cross validation. Use the mean squared error (mse):

$$mse = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_j)^2$$

First, partition the data into five folds. In each fold, the software uses 4/5th of the data for training and 1/5th of the data for validation (testing).

```

cvp          = cvpartition(length(y), 'kfold', 5);
numtestsets = cvp.NumTestSets;

```

Compute the lambda values to test for and create an array to store the loss values.

```

lambdavalss = linspace(0, 3, 50) * std(y) / length(y);
lossvals    = zeros(length(lambdavalss), numtestsets);

```

Perform NCA and compute the loss for each  $\lambda$  value and each fold.

```

for i = 1:length(lambdavalss)
    for k = 1:numtestsets

        Xtrain = X(cvp.training(k), :);
        ytrain = y(cvp.training(k), :);
        Xtest  = X(cvp.test(k), :);
        ytest  = y(cvp.test(k), :);

        nca = fsrnca(Xtrain, ytrain, 'FitMethod', 'exact', ...
                    'Solver', 'lbfgs', 'Verbose', 0, 'Lambda', lambdavalss(i), ...
                    'LossFunction', 'mse');

        lossvals(i, k) = loss(nca, Xtest, ytest, 'LossFunction', 'mse');
    end
end

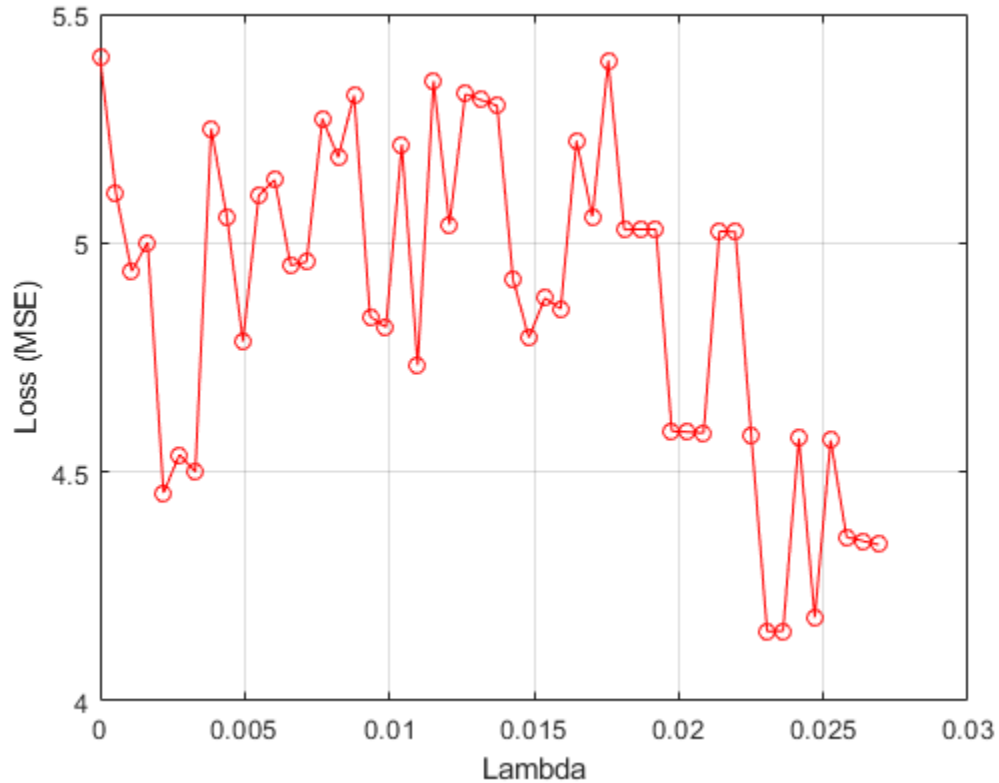
```

Plot the mean loss corresponding to each lambda value.

```

meanloss = mean(lossvals, 2);
figure;
plot(lambdavalss, meanloss, 'ro-');
xlabel('Lambda');
ylabel('Loss (MSE)');
grid on;

```



Find the  $\lambda$  value that produces the minimum average loss.

```
[~,idx] = min(mean(lossvals,2));
bestlambda = lambdaval(idx)
```

```
bestlambda =
```

```
0.0231
```

Perform feature selection using the best  $\lambda$  value and mse.

```
nca = fsrnca(X,y,'FitMethod','exact','Solver','lbfgs',...
    'Verbose',1,'Lambda',bestlambda,'LossFunction','mse');
```



o Solver = LBFGS, HessianHistorySize = 15, LineSearchMethod = weakwolfe

ITER	FUN VALUE	NORM GRAD	NORM STEP	CURV	GAMMA	ALPHA
0	6.414642e+00	8.430e-01	0.000e+00		7.117e-01	0.000
1	6.066100e+00	9.952e-01	1.264e+00	OK	3.741e-01	1.000
2	5.498221e+00	4.267e-01	4.250e-01	OK	4.016e-01	1.000
3	5.108548e+00	3.933e-01	8.564e-01	OK	3.599e-01	1.000
4	4.808456e+00	2.505e-01	9.352e-01	OK	8.798e-01	1.000
5	4.677382e+00	2.085e-01	6.014e-01	OK	1.052e+00	1.000
6	4.487789e+00	4.726e-01	7.374e-01	OK	5.593e-01	1.000
7	4.310099e+00	2.484e-01	4.253e-01	OK	3.367e-01	1.000
8	4.258539e+00	3.629e-01	4.521e-01	OK	4.705e-01	5.000
9	4.175345e+00	1.972e-01	2.608e-01	OK	4.018e-01	1.000
10	4.122340e+00	9.169e-02	2.947e-01	OK	3.487e-01	1.000
11	4.095525e+00	9.798e-02	2.529e-01	OK	1.188e+00	1.000
12	4.059690e+00	1.584e-01	5.213e-01	OK	9.930e-01	1.000
13	4.029208e+00	7.411e-02	2.076e-01	OK	4.886e-01	1.000
14	4.016358e+00	1.068e-01	2.696e-01	OK	6.919e-01	1.000
15	4.004521e+00	5.434e-02	1.136e-01	OK	5.647e-01	1.000
16	3.986929e+00	6.158e-02	2.993e-01	OK	1.353e+00	1.000
17	3.976342e+00	4.966e-02	2.213e-01	OK	7.668e-01	1.000
18	3.966646e+00	5.458e-02	2.529e-01	OK	1.988e+00	1.000
19	3.959586e+00	1.046e-01	4.169e-01	OK	1.858e+00	1.000

ITER	FUN VALUE	NORM GRAD	NORM STEP	CURV	GAMMA	ALPHA
20	3.953759e+00	8.248e-02	2.892e-01	OK	1.040e+00	1.000
21	3.945475e+00	3.119e-02	1.698e-01	OK	1.095e+00	1.000
22	3.941567e+00	2.350e-02	1.293e-01	OK	1.117e+00	1.000
23	3.939468e+00	1.296e-02	1.805e-01	OK	2.287e+00	1.000
24	3.938662e+00	8.591e-03	5.955e-02	OK	1.553e+00	1.000
25	3.938239e+00	6.421e-03	5.334e-02	OK	1.102e+00	1.000
26	3.938013e+00	5.449e-03	6.773e-02	OK	2.085e+00	1.000
27	3.937896e+00	6.226e-03	3.368e-02	OK	7.541e-01	1.000
28	3.937820e+00	2.497e-03	2.397e-02	OK	7.940e-01	1.000
29	3.937791e+00	2.004e-03	1.339e-02	OK	1.863e+00	1.000
30	3.937784e+00	2.448e-03	1.265e-02	OK	9.667e-01	1.000
31	3.937778e+00	6.973e-04	2.906e-03	OK	4.672e-01	1.000
32	3.937778e+00	3.038e-04	9.502e-04	OK	1.060e+00	1.000
33	3.937777e+00	2.327e-04	1.069e-03	OK	1.597e+00	1.000

	34		3.937777e+00		1.959e-04		1.537e-03		OK		4.026e+00		1.000
	35		3.937777e+00		1.162e-04		1.464e-03		OK		3.418e+00		1.000
	36		3.937777e+00		8.353e-05		3.660e-04		OK		7.304e-01		5.000
	37		3.937777e+00		1.412e-05		1.412e-04		OK		7.842e-01		1.000
	38		3.937777e+00		1.277e-05		3.808e-05		OK		1.021e+00		1.000
	39		3.937777e+00		8.614e-06		3.698e-05		OK		2.561e+00		1.000

=====													
ITER	FUN VALUE	NORM GRAD	NORM STEP	CURV	GAMMA	ALPHA							
=====													
	40		3.937777e+00		3.159e-06		5.299e-05		OK		4.331e+00		1.000
	41		3.937777e+00		2.657e-06		1.080e-05		OK		7.038e-01		5.000
	42		3.937777e+00		7.054e-07		7.036e-06		OK		9.519e-01		1.000

Infinity norm of the final gradient = 7.054e-07

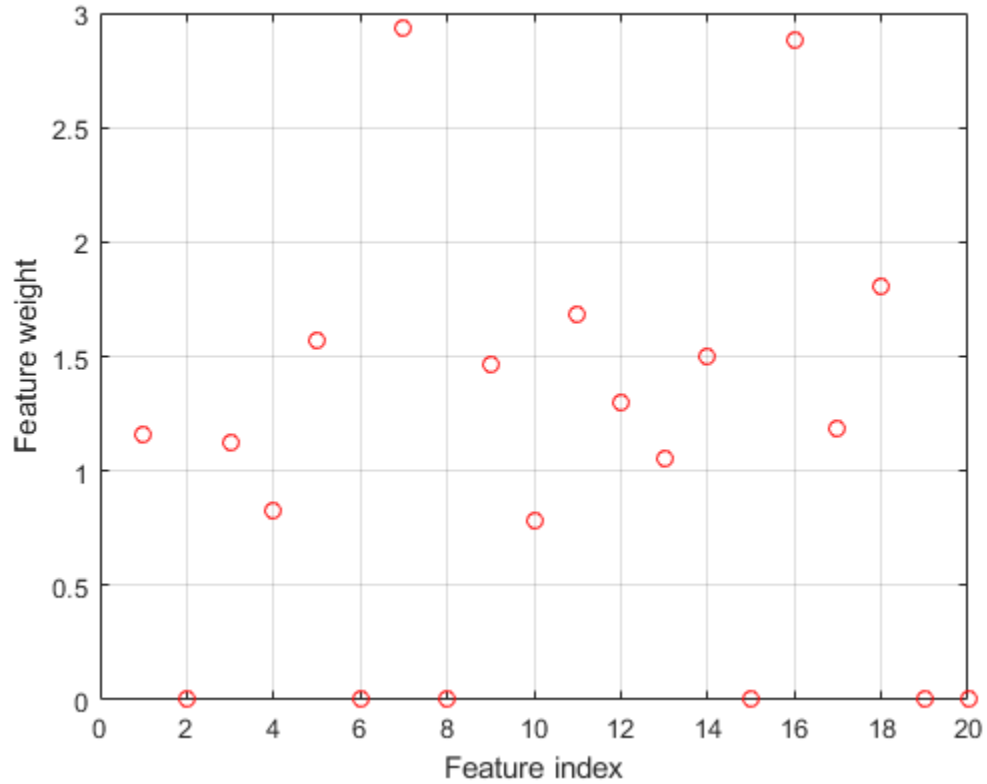
Two norm of the final step = 7.036e-06, TolX = 1.000e-06

Relative infinity norm of the final gradient = 7.054e-07, TolFun = 1.000e-06

EXIT: Local minimum found.

Plot selected features.

```
figure;
plot(nca.FeatureWeights,'ro');
grid on;
xlabel('Feature index');
ylabel('Feature weight');
```



Predict the response values using the nca model and plot the fitted (predicted) response values and the actual response values.

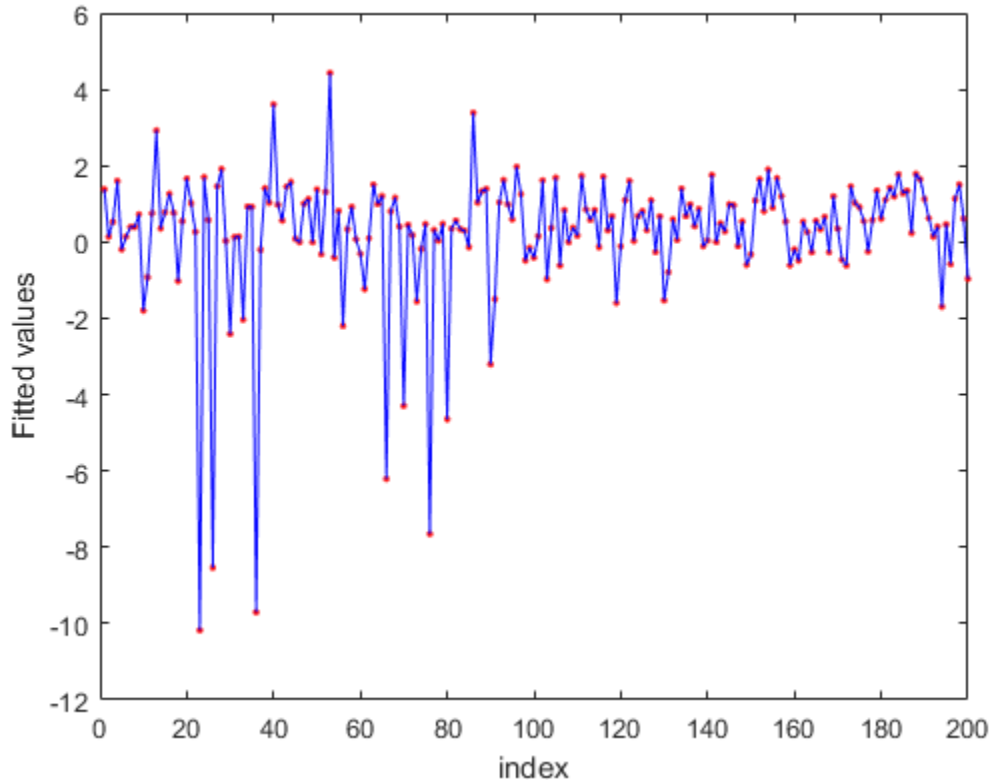
```
fitted = predict(nca,X);  
figure;  
plot(y,'r.');
```

hold on;

```
plot(fitted,'b-');
```

xlabel('index');

```
ylabel('Fitted values');
```



`fsrcna` tries to fit every point in data including the outliers. As a result it assigns nonzero weights to many features besides predictors 4, 7, and 13.

### Use built-in robust loss function

Repeat the same process of tuning the regularization parameter, this time using the built-in  $\epsilon$ -insensitive loss function:

$$l(y_i, y_j) = \max(0, |y_i - y_j| - \epsilon)$$

$\epsilon$ -insensitive loss function is more robust to outliers than mean squared error.

```
lambdavals = linspace(0, 3, 50) * std(y) / length(y);
cvp        = cvpartition(length(y), 'kfold', 5);
```

```

numtestsets = cvp.NumTestSets;
lossvals    = zeros(length(lambdaval), numtestsets);

    for i = 1:length(lambdaval)
        for k = 1:numtestsets

            Xtrain = X(cvp.training(k),:);
            ytrain = y(cvp.training(k),:);
            Xtest  = X(cvp.test(k),:);
            ytest  = y(cvp.test(k),:);

            nca = fsrnca(Xtrain,ytrain,'FitMethod','exact',...
                'Solver','sgd','Verbose',0,'Lambda',lambdaval(i),...
                'LossFunction','epsiloninsensitive','Epsilon',0.8);

            lossvals(i,k) = loss(nca,Xtest,ytest,'LossFunction','mse');
        end
    end
end

```

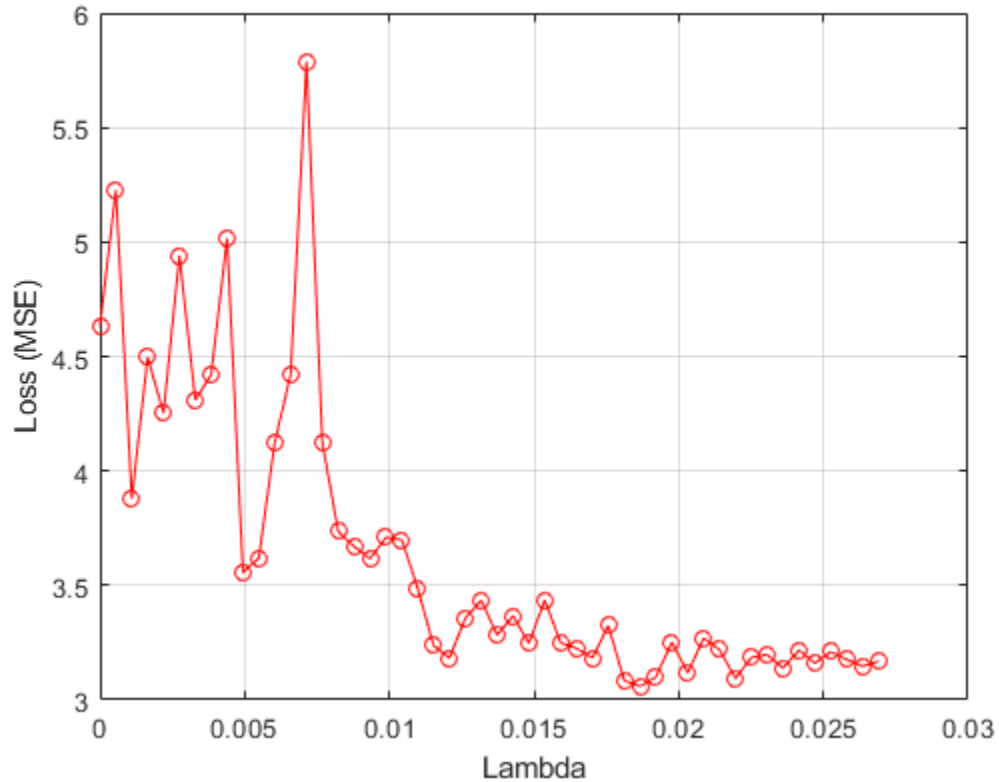
The  $\epsilon$  value to use depends on the data and the best value can be determined using cross-validation as well. But choosing the  $\epsilon$  value is out of scope of this example. The choice of  $\epsilon$  in this example is mainly for illustrating the robustness of the method.

Plot the mean loss corresponding to each lambda value.

```

meanloss = mean(lossvals,2);
figure;
plot(lambdaval,meanloss,'ro-');
xlabel('Lambda');
ylabel('Loss (MSE)');
grid on;

```



Find the lambda value that produces the minimum average loss.

```
[~,idx] = min(mean(lossvals,2));
bestlambda = lambdaval(idx)
```

```
bestlambda =
```

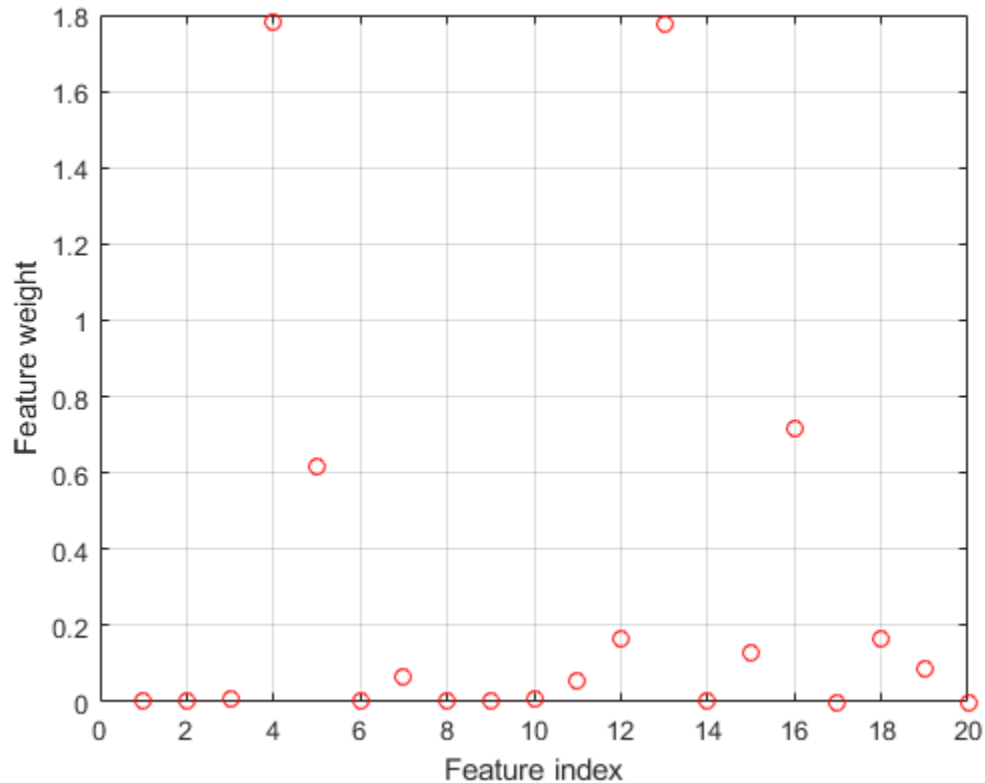
```
0.0187
```

Fit neighborhood component analysis model using  $\epsilon$ -insensitive loss function and best lambda value.

```
nca = fsrnca(X,y,'FitMethod','exact','Solver','sgd',...
    'Lambda',bestlambda,'LossFunction','epsiloninsensitive','Epsilon',0.8);
```

Plot selected features.

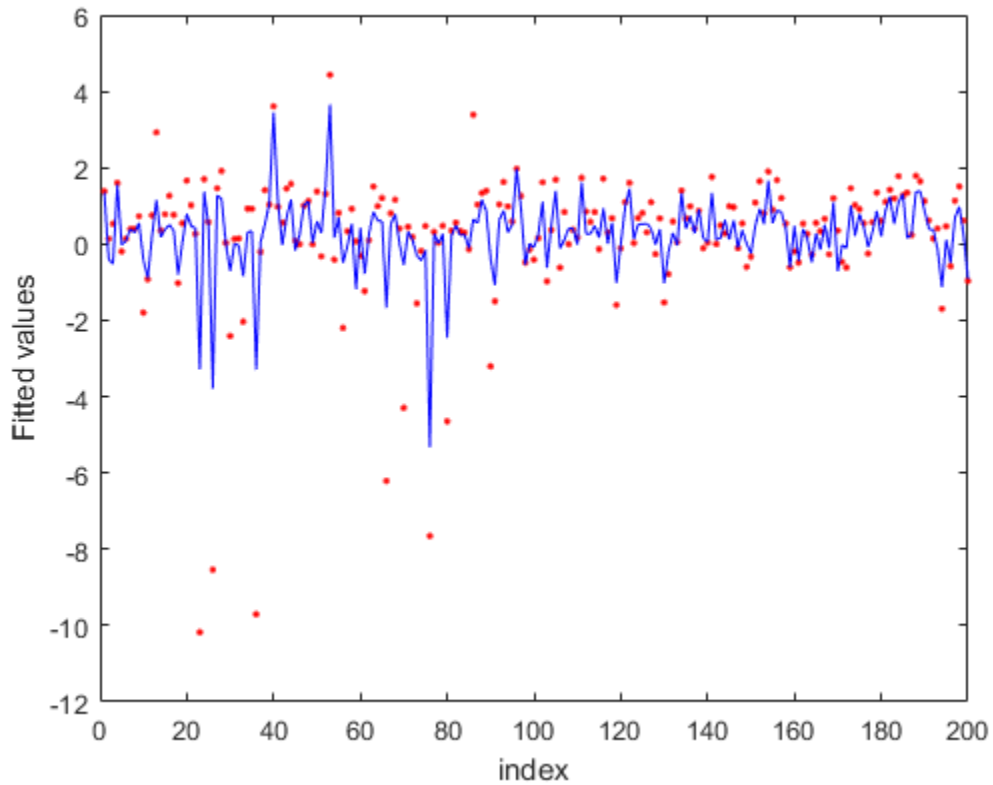
```
figure;
plot(nca.FeatureWeights,'ro');
grid on;
xlabel('Feature index');
ylabel('Feature weight');
```



Plot fitted values.

```
fitted = predict(nca,X);
figure;
```

```
plot(y, 'r. ');  
hold on;  
plot(fitted, 'b-');  
xlabel('index');  
ylabel('Fitted values');
```



$\epsilon$ -insensitive loss seems more robust to outliers. It identified fewer features than mse as relevant. The fit shows that it is still impacted by some of the outliers.

### Use custom robust loss function

Define a custom robust loss function that is robust to outliers to use in feature selection for regression:



$$f(y_i, y_j) = 1 - \exp(-|y_i, y_j|).$$

```
customlossFcn = @(yi,yj) 1 - exp(-abs(bsxfun(@minus,yi,yj')));
```

Tune the regularization parameter using the custom-defined robust loss function.

```
lambdavalss = linspace(0,3,50)*std(y)/length(y);
cvp         = cvpartition(length(y),'kfold',5);
numtestsets = cvp.NumTestSets;
lossvals    = zeros(length(lambdavalss),numtestsets);

    for i = 1:length(lambdavalss)
        for k = 1:numtestsets

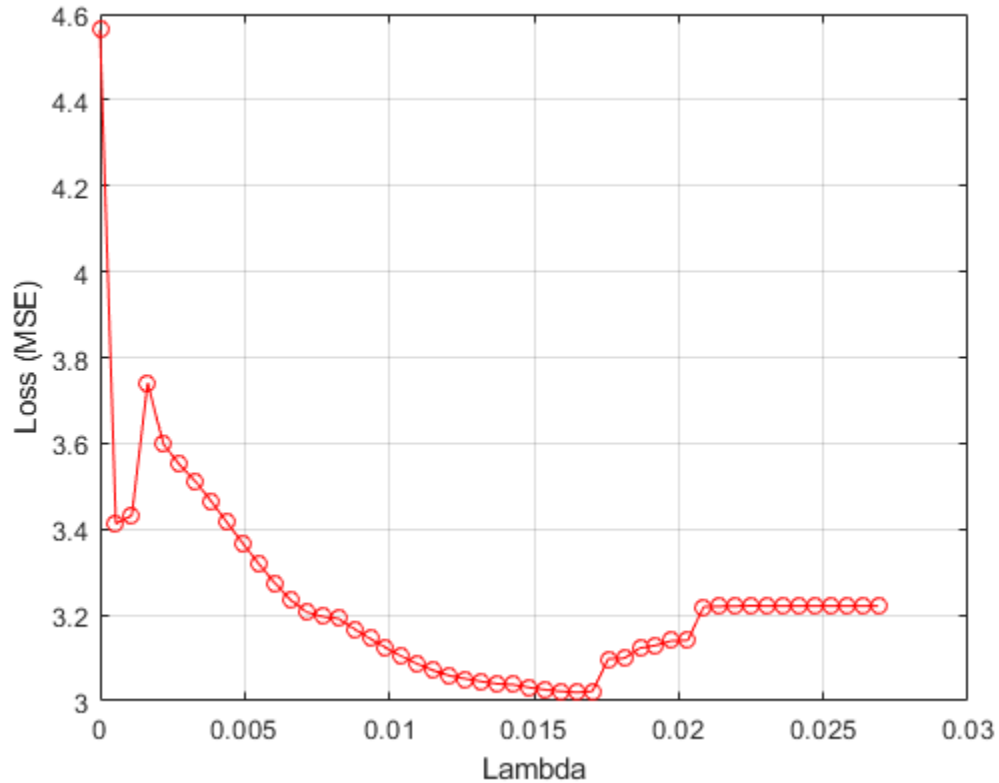
            Xtrain = X(cvp.training(k),:);
            ytrain = y(cvp.training(k),:);
            Xtest  = X(cvp.test(k),:);
            ytest  = y(cvp.test(k),:);

            nca = fsrnca(Xtrain,ytrain,'FitMethod','exact',...
                'Solver','lbfgs','Verbose',0,'Lambda',lambdavalss(i),...
                'LossFunction',customlossFcn);

            lossvals(i,k) = loss(nca,Xtest,ytest,'LossFunction','mse');
        end
    end
```

Plot the mean loss corresponding to each lambda value.

```
meanloss = mean(lossvals,2);
figure;
plot(lambdavalss,meanloss,'ro-');
xlabel('Lambda');
ylabel('Loss (MSE)');
grid on;
```



Find the  $\lambda$  value that produces the minimum average loss.

```
[~,idx] = min(mean(lossvals,2));
bestlambda = lambdaval(idx)
```

```
bestlambda =
```

```
0.0165
```

Perform feature selection using the custom robust loss function and best  $\lambda$  value.

```
nca = fsrnca(X,y,'FitMethod','exact','Solver','lbfgs',...
    'Verbose',1,'Lambda',bestlambda,'LossFunction',customlossFcn);
```

```
o Solver = LBFGS, HessianHistorySize = 15, LineSearchMethod = weakwolfe
```

```
=====
| ITER | FUN VALUE | NORM GRAD | NORM STEP | CURV | GAMMA | ALPH
=====
| 0 | 8.610073e-01 | 4.921e-02 | 0.000e+00 | | 1.219e+01 | 0.000
| 1 | 6.582278e-01 | 2.328e-02 | 1.820e+00 | OK | 2.177e+01 | 1.000
| 2 | 5.706490e-01 | 2.241e-02 | 2.360e+00 | OK | 2.541e+01 | 1.000
| 3 | 5.677090e-01 | 2.666e-02 | 7.583e-01 | OK | 1.092e+01 | 1.000
| 4 | 5.620806e-01 | 5.524e-03 | 3.335e-01 | OK | 9.973e+00 | 1.000
| 5 | 5.616054e-01 | 1.428e-03 | 1.025e-01 | OK | 1.736e+01 | 1.000
| 6 | 5.614779e-01 | 4.446e-04 | 8.350e-02 | OK | 2.507e+01 | 1.000
| 7 | 5.614653e-01 | 4.118e-04 | 2.466e-02 | OK | 2.105e+01 | 1.000
| 8 | 5.614620e-01 | 1.307e-04 | 1.373e-02 | OK | 2.002e+01 | 1.000
| 9 | 5.614615e-01 | 9.318e-05 | 4.128e-03 | OK | 3.683e+01 | 1.000
| 10 | 5.614611e-01 | 4.579e-05 | 8.785e-03 | OK | 6.170e+01 | 1.000
| 11 | 5.614610e-01 | 1.232e-05 | 1.582e-03 | OK | 2.000e+01 | 5.000
| 12 | 5.614610e-01 | 3.174e-06 | 4.742e-04 | OK | 2.510e+01 | 1.000
| 13 | 5.614610e-01 | 7.896e-07 | 1.683e-04 | OK | 2.959e+01 | 1.000
```

```
Infinity norm of the final gradient = 7.896e-07
```

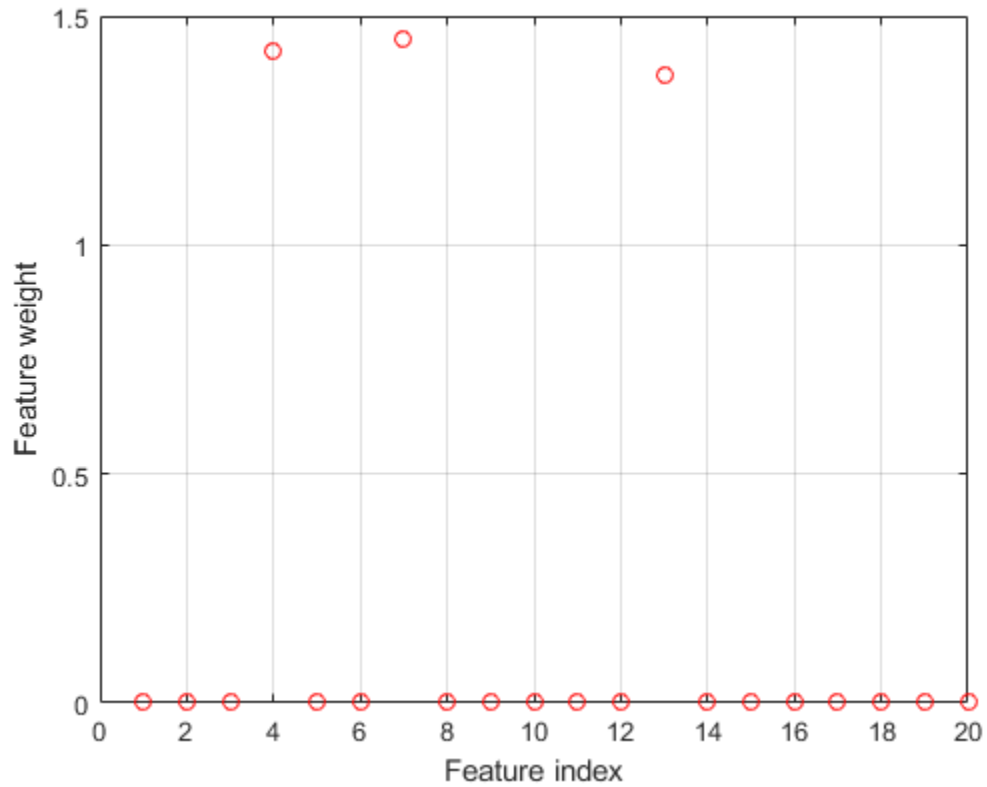
```
Two norm of the final step = 1.683e-04, TolX = 1.000e-06
```

```
Relative infinity norm of the final gradient = 7.896e-07, TolFun = 1.000e-06
```

```
EXIT: Local minimum found.
```

Plot selected features.

```
figure;
plot(nca.FeatureWeights, 'ro');
grid on;
xlabel('Feature index');
ylabel('Feature weight');
```

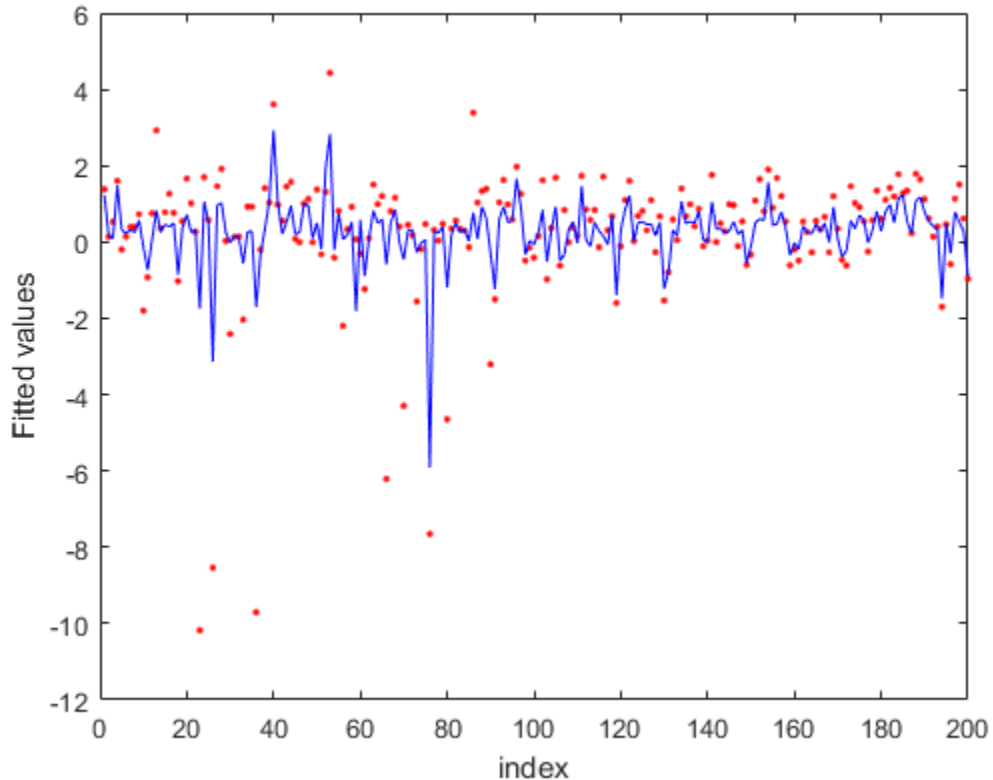


Plot fitted values.

```
fitted = predict(nca,X);  
figure;  
plot(y, 'r.');
```

hold on;

```
plot(fitted, 'b-');  
xlabel('index');  
ylabel('Fitted values');
```



In this case, the loss is not affected by the outliers and results are based on most of the observation values. `fsrnca` detects the predictors 4, 7, and 13 as relevant features and does not select any other features.

### Why does the loss function choice affect the results?

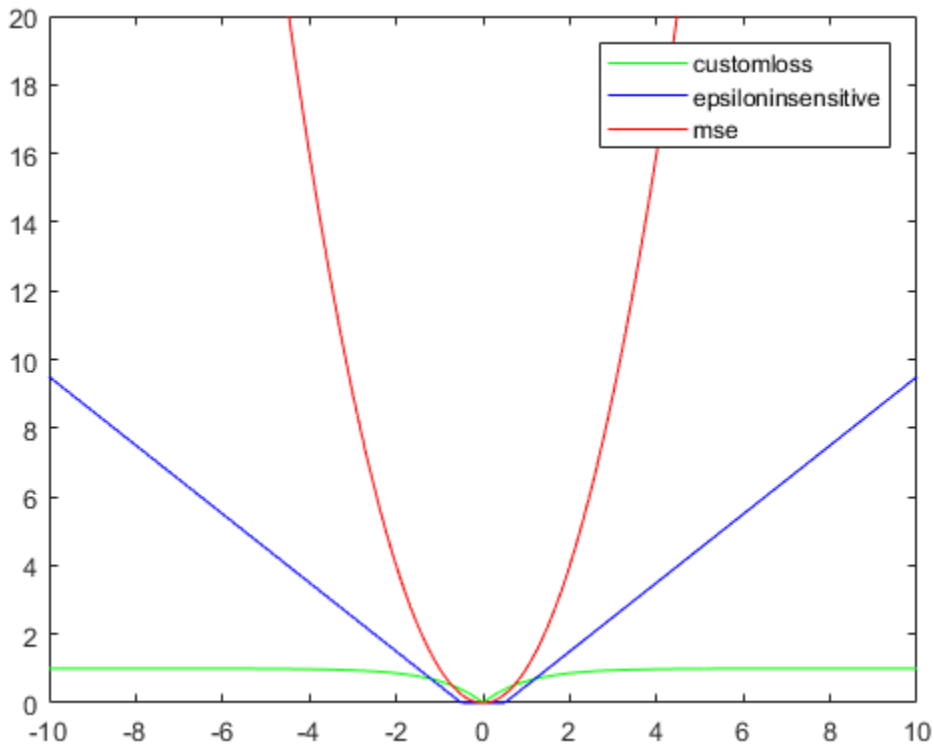
First, compute the loss functions for a series of values for the difference between two observations.

```
deltay = linspace(-10,10,1000)';
% Compute custom loss function values
customlossvals = customlossFcn(deltay,0);
% Compute epsilon insensitive loss function and values
epsinsensitive = @(yi,yj,E) max(0,abs(bsxfun(@minus,yi,yj'))-E);
```

```
epsinsenvals = epsinsensitive(deltay,0,0.5);  
% Compute mse loss function and values  
mse = @(yi,yj) (yi-yj).^2;  
msevals = mse(deltay,0);
```

Now, plot the loss functions to see their difference and why they affect the results in the way they do.

```
figure;  
xlabel('(yi - yj)')  
ylabel('loss(yi,yj)')  
plot(deltay,customlossvals,'g-',deltay,epsinsenvals,'b-',deltay,msevals,'r-')  
legend('customloss','epsiloninsensitive','mse')  
ylim([0 20])
```



As the difference between two response values increases, mse increases quadratically, which makes it very sensitive to outliers. As fsrnca tries to minimize this loss, it ends up identifying more features as relevant. The epsilon insensitive loss is more resistant to outliers than mse, but eventually it does start to increase linearly as the difference between two observations increase. As the difference between two observations increase, the robust loss function does approach 1 and stays at that value even though the difference between the observations keeps increasing. Out of three, it is the most robust to outliers.

## Neighborhood Component Analysis (NCA) Feature Selection

### In this section...

“NCA Feature Selection for Classification” on page 15-126

“NCA Feature Selection for Regression” on page 15-129

“Impact of Standardization” on page 15-130

“Choosing the Regularization Parameter Value” on page 15-131

Neighborhood component analysis (NCA) is a non-parametric and embedded method for selecting features with the goal of maximizing prediction accuracy of regression and classification algorithms. The Statistics and Machine Learning Toolbox functions `fscnca` and `fsrcnca` perform NCA feature selection with regularization to learn feature weights for minimization of an objective function that measures the average leave-one-out classification or regression loss over the training data.

### NCA Feature Selection for Classification

Consider a multi-class classification problem with a training set containing  $n$  observations:

$$S = \{(\mathbf{x}_i, y_i), i = 1, 2, \dots, n\},$$

where  $\mathbf{x}_i \in \mathbb{R}^p$  are the feature vectors,  $y_i \in \{1, 2, \dots, c\}$  are the class labels, and  $c$  is the number of classes. The aim is to learn a classifier  $f: \mathbb{R}^p \rightarrow \{1, 2, \dots, c\}$  that accepts a feature vector and makes a prediction  $f(\mathbf{x})$  for the true label  $y$  of  $\mathbf{x}$ .

Consider a randomized classifier that:

- Randomly picks a point,  $\text{Ref}(\mathbf{x})$ , from  $S$  as the ‘reference point’ for  $\mathbf{x}$
- Labels  $\mathbf{x}$  using the label of the reference point  $\text{Ref}(\mathbf{x})$ .

This scheme is similar to that of a 1-NN classifier where the reference point is chosen to be the nearest neighbor of the new point  $\mathbf{x}$ . In NCA, the reference point is chosen randomly and all points in  $S$  have some probability of being selected as the reference



point. The probability  $P(\text{Ref}(\mathbf{x}) = \mathbf{x}_j | S)$  that point  $\mathbf{x}_j$  is picked from  $S$  as the reference point for  $\mathbf{x}$  is higher if  $\mathbf{x}_j$  is closer to  $\mathbf{x}$  as measured by the distance function  $d_w$ , where

$$d_w(\mathbf{x}_i, \mathbf{x}_j) = \sum_{r=1}^p w_r^2 |x_{ir} - x_{jr}|,$$

and  $w_r$  are the feature weights. Assume that

$$P(\text{Ref}(\mathbf{x}) = \mathbf{x}_j | S) \propto k(d_w(\mathbf{x}, \mathbf{x}_j)),$$

where  $k$  is some kernel or a similarity function that assumes large values when

$d_w(\mathbf{x}, \mathbf{x}_j)$  is small. Suppose it is

$$k(z) = \exp\left(-\frac{z}{\sigma}\right),$$

as suggested in [1]. The reference point for  $\mathbf{x}$  is chosen from  $S$ , so sum of

$P(\text{Ref}(\mathbf{x}) = \mathbf{x}_j | S)$  for all  $j$  must be equal to 1. Therefore, it is possible to write

$$P(\text{Ref}(\mathbf{x}) = \mathbf{x}_j | S) = \frac{k(d_w(\mathbf{x}, \mathbf{x}_j))}{\sum_{j=1}^n k(d_w(\mathbf{x}, \mathbf{x}_j))}.$$

Now consider the leave-one-out application of this randomized classifier, that is,

predicting the label of  $\mathbf{x}_i$  using the data in  $S^{-i}$ , the training set  $S$  excluding the point

$(\mathbf{x}_i, y_i)$ . The probability that point  $\mathbf{x}_j$  is picked as the reference point for  $\mathbf{x}_i$  is

$$p_{ij} = P(\text{Ref}(\mathbf{x}_i) = \mathbf{x}_j | S^{-i}) = \frac{k(d_w(\mathbf{x}_i, \mathbf{x}_j))}{\sum_{j=1, j \neq i}^N k(d_w(\mathbf{x}_i, \mathbf{x}_j))}.$$

The average leave-one-out probability of correct classification is the probability  $p_i$  that the randomized classifier correctly classifies observation  $i$  using  $S^{-i}$ .

$$p_i = \sum_{j=1, j \neq i}^N P(\text{Ref}(\mathbf{x}_i) = \mathbf{x}_j | S^{-i}) \times \mathbb{I}(y_i = y_j)$$

where  $\mathbb{I}(\text{true}) = 1$ ,  $\mathbb{I}(\text{false}) = 0$ . Then,

$$p_i = \sum_{j=1, j \neq i}^N p_{ij} y_{ij},$$

where

$$y_{ij} = \begin{cases} 1, & \text{if } y_i = y_j \\ 0, & \text{otherwise} \end{cases}.$$

The average leave-one-out probability of correct classification using the randomized classifier can be written as

$$F(\mathbf{w}) = \sum_{i=1}^N p_i.$$

The right hand side of  $F(\mathbf{w})$  depends on the weight vector  $\mathbf{w}$ . The goal of neighborhood component analysis is to maximize  $F(\mathbf{w})$  with respect to  $\mathbf{w}$ . `fscnca` uses the regularized objective function as introduced in [1].

$$\begin{aligned} F(\mathbf{w}) &= \sum_{i=1}^N p_i - \lambda \sum_{r=1}^p w_r^2 \\ &= \sum_{i=1}^N \underbrace{\sum_{j=1, j \neq i}^N p_{ij} y_{ij}}_{F_i(\mathbf{w})} - \lambda \sum_{r=1}^p w_r^2, \\ &= \sum_{i=1}^N F_i(\mathbf{w}) \end{aligned}$$

where  $\lambda$  is the regularization parameter.

The regularization term drives many of the weights in  $\mathbf{w}$  to 0. After choosing the kernel parameter  $\sigma$  in  $p_{ij}$  as 1, finding the weight vector  $\mathbf{w}$  can be expressed as the following minimization problem for given  $\lambda$ .

$$\begin{aligned}\hat{\mathbf{w}} &= \underbrace{\operatorname{argmin}}_{\mathbf{w}} f(\mathbf{w}) = \sum_{i=1}^N a_i f_i(\mathbf{w}), \\ f(\mathbf{w}) &= -F(\mathbf{w}), \\ f_i(\mathbf{w}) &= -F_i(\mathbf{w}).\end{aligned}$$

So, it finds the weights that minimize the classification error. You can specify a custom loss function using the `LossFunction` name-value pair argument in the call to `fscnca`.

## NCA Feature Selection for Regression

The `fsrcnca` function performs NCA feature selection modified for regression. Given  $n$  observations

$$S = \{(\mathbf{x}_i, y_i), i = 1, 2, \dots, n\},$$

the only difference from the classification problem is that the response values  $y_i \in \mathbb{R}$  are continuous. In this case, the aim is to predict the response  $y$  given the training set  $S$ .

Consider a randomized regression model that:

- Randomly picks a point (`Ref(x)`) from  $S$  as the ‘reference point’ for  $\mathbf{x}$
- Sets the response value at  $\mathbf{x}$  equal to the response value of the reference point `Ref(x)`.

Again, the probability  $P(\operatorname{Ref}(\mathbf{x}) = \mathbf{x}_j | S)$  that point  $\mathbf{x}_j$  is picked from  $S$  as the reference point for  $\mathbf{x}$  is

$$P(\operatorname{Ref}(\mathbf{x}) = \mathbf{x}_j | S) = \frac{k(d_w(\mathbf{x}, \mathbf{x}_j))}{\sum_{j=1}^n k(d_w(\mathbf{x}, \mathbf{x}_j))}.$$

Now consider the leave-one-out application of this randomized regression model, that is, predicting the response for  $\mathbf{x}_i$  using the data in  $S^{-i}$ , the training set  $S$  excluding the point  $(\mathbf{x}_i, y_i)$ . The probability that point  $\mathbf{x}_j$  is picked as the reference point for  $\mathbf{x}_i$  is

$$p_{ij} = P(\text{Ref}(\mathbf{x}_i) = \mathbf{x}_j | S^{-i}) = \frac{k(d_w(\mathbf{x}_i, \mathbf{x}_j))}{\sum_{j=1, j \neq i}^N k(d_w(\mathbf{x}_i, \mathbf{x}_j))}.$$

Let  $\hat{y}_i$  be the response value the randomized regression model predicts and  $y_i$  be the actual response for  $\mathbf{x}_i$ . And let  $l: \mathbb{R}^2 \rightarrow \mathbb{R}$  be a loss function that measures the disagreement between  $\hat{y}_i$  and  $y_i$ . Then, the average value of  $l(y_i, y_j)$  is

$$\begin{aligned} l_i = E(l(y_i, y_j) | S^{-i}) &= \sum_{j=1, j \neq i}^N \text{probability that } \mathbf{x}_j \text{ is the reference point for } \mathbf{x}_i \times l(y_i, y_j) \\ &= \sum_{j=1, j \neq i}^N p_{ij} l(y_i, y_j) \end{aligned}$$

After adding the regularization term, the objective function for minimization is:

$$f(\mathbf{w}) = \frac{1}{n} \sum_{i=1}^n l_i + \lambda \sum_{r=1}^p w_r^2.$$

The default loss function  $l(y_i, y_j)$  for nca for regression is mean absolute deviation, but you can specify other loss functions, including a custom one, using the `LossFunction` name-value pair argument in the call to `fsrnca`.

## Impact of Standardization

The regularization term derives the weights of irrelevant predictors to zero. In the objective functions for NCA for classification or regression, there is only one regularization parameter  $\lambda$  for all weights. This fact requires the magnitudes of the weights to be comparable to each other. When the feature vectors  $\mathbf{x}_i$  in  $S$  are in

different scales, this might result in weights that are in different scales and not meaningful. To avoid this situation, standardize the predictors to have zero mean and unit standard deviation before applying NCA. You can standardize the predictors using the `'Standardize'`, `true` name-value pair argument in the call to `fscnca` or `fsrcnca`.

## Choosing the Regularization Parameter Value

It is usually necessary to select a value of the regularization parameter by calculating the accuracy of the randomized NCA classifier or regression model on an independent test set. If you use cross-validation instead of a single test set, select the  $\lambda$  value that minimizes the average loss across the cross-validation folds. For examples, see “Tune Regularization Parameter to Detect Features Using NCA for Classification” and “Tune Regularization Parameter in NCA for Regression” on page 32-2757.

## References

- [1] Yang, W., K. Wang, W. Zuo. "Neighborhood Component Feature Selection for High-Dimensional Data." *Journal of Computers*. Vol. 7, Number 1, January, 2012.

## See Also

`fscnca` | `fsrcnca`

## t-SNE

In this section...
“What Is t-SNE?” on page 15-132
“t-SNE Algorithm” on page 15-133
“Barnes-Hut Variation of t-SNE” on page 15-136
“Characteristics of t-SNE” on page 15-136

### What Is t-SNE?

t-SNE ( $t_{sne}$ ) is an algorithm for dimensionality reduction that is well-suited to visualizing high-dimensional data. The name stands for *t*-distributed Stochastic Neighbor Embedding. The idea is to embed high-dimensional points in low dimensions in a way that respects similarities between points. Nearby points in the high-dimensional space correspond to nearby embedded low-dimensional points, and distant points in high-dimensional space correspond to distant embedded low-dimensional points. (Generally, it is impossible to match distances exactly between high-dimensional and low-dimensional spaces.)

The `tsne` function creates a set of low-dimensional points from high-dimensional data. Typically, you visualize the low-dimensional points to see natural clusters in the original high-dimensional data.

The algorithm takes the following general steps to embed the data in low dimensions.

- 1 Calculate the pairwise distances between the high-dimensional points.
- 2 Create a standard deviation  $\sigma_i$  for each high-dimensional point  $i$  so that the perplexity of each point is at a predetermined level. For the definition of perplexity, see “Compute Distances, Gaussian Variances, and Similarities” on page 15-133.
- 3 Calculate the similarity matrix. This is the joint probability distribution of  $X$ , defined by “Equation 15-2” on page 15-134.
- 4 Create an initial set of low-dimensional points.
- 5 Iteratively update the low-dimensional points to minimize the Kullback-Leibler divergence between a Gaussian distribution in the high-dimensional space and a  $t$  distribution in the low-dimensional space. This optimization procedure is the most time-consuming part of the algorithm.

See van der Maaten and Hinton [1].

## t-SNE Algorithm

The basic t-SNE algorithm performs the following steps.

- “Prepare Data” on page 15-133
- “Compute Distances, Gaussian Variances, and Similarities” on page 15-133
- “Initialize the Embedding and Divergence” on page 15-134
- “Gradient Descent of Kullback-Leibler Divergence” on page 15-135

### Prepare Data

`tsne` first removes each row of the input data  $X$  that contains any `NaN` values. Then, if the `Standardize` name-value pair is `true`, `tsne` centers  $X$  by subtracting the mean of each column, and scales  $X$  by dividing its columns by their standard deviations.

The original authors van der Maaten and Hinton [1] recommend reducing the original data  $X$  to a lower-dimensional version using “Principal Component Analysis (PCA)” on page 15-83. You can set the `tsne NumPCAComponents` name-value pair to the number of dimensions you like, perhaps 50. To exercise more control over this step, preprocess the data using the `pca` function.

### Compute Distances, Gaussian Variances, and Similarities

After the preprocessing, `tsne` calculates the distance  $d(x_i, x_j)$  between each pair of points  $x_i$  and  $x_j$  in  $X$ . You can choose various distance metrics using the `Distance` name-value pair. By default, `tsne` uses the standard Euclidean metric. `tsne` uses the square of the distance metric in its subsequent calculations.

Then for each row  $i$  of  $X$ , `tsne` calculates a standard deviation  $\sigma_i$  so that the perplexity of row  $i$  is equal to the `Perplexity` name-value pair. The perplexity is defined in terms of a model Gaussian distribution as follows. As van der Maaten and Hinton [1] describe,

“The similarity of datapoint  $x_j$  to datapoint  $x_i$  is the conditional probability,  $p_{j|i}$ , that  $x_i$  would pick  $x_j$  as its neighbor if neighbors were picked in proportion to their probability

density under a Gaussian centered at  $x_i$ . For nearby datapoints,  $p_{j|i}$  is relatively high,

whereas for widely separated datapoints,  $p_{j|i}$  will be almost infinitesimal (for reasonable values of the variance of the Gaussian,  $\sigma_i$ .”

Define the conditional probability of  $j$  given  $i$  as

$$p_{j|i} = \frac{\exp\left(-d(x_i, x_j)^2 / (2\sigma_i^2)\right)}{\sum_{k \neq i} \exp\left(-d(x_i, x_k)^2 / (2\sigma_i^2)\right)}$$
$$p_{i|i} = 0.$$

Then define the joint probability  $p_{ij}$  by symmetrizing the conditional probabilities:

$$p_{ij} = \frac{p_{j|i} + p_{i|j}}{2N},$$

where  $N$  is the number of rows of  $X$ .

The distributions still do not have their standard deviations  $\sigma_i$  defined in terms of the Perplexity name-value pair. Let  $P_i$  represents the conditional probability distribution over all other datapoints given datapoint  $x_i$ . The perplexity of the distribution is

$$\text{perplexity}(P_i) = 2^{H(P_i)},$$

where  $H(P_i)$  is the Shannon entropy of  $P_i$ :

$$H(P_i) = -\sum_j p_{j|i} \log_2(p_{j|i}).$$

The perplexity measures the effective number of neighbors of point  $i$ . `tsne` performs a binary search over the  $\sigma_i$  to achieve a fixed perplexity for each point  $i$ .

### Initialize the Embedding and Divergence

To embed the points in  $X$  into a low-dimensional space, `tsne` performs an optimization. `tsne` attempts to minimize the Kullback-Leibler divergence between the model Gaussian distribution of the points in  $X$  and a Student  $t$  distribution of points  $Y$  in the low-dimensional space.

The minimization procedure begins with an initial set of points  $Y$ . `tsne` create the points by default as random Gaussian-distributed points. You can also create these points yourself and include them in the 'InitialY' name-value pair for `tsne`. `tsne` then calculates the similarities between each pair of points in  $Y$ .



The probability model  $q_{ij}$  of the distribution of the distances between points  $y_i$  and  $y_j$  is

$$q_{ij} = \frac{\left(1 + \|y_i - y_j\|^2\right)^{-1}}{\sum_k \sum_{l \neq k} \left(1 + \|y_k - y_l\|^2\right)^{-1}}$$

$$q_{ii} = 0.$$

Using this definition and the model of distances in  $X$  given by “Equation 15-2” on page 15-134, the Kullback-Liebler divergence between the joint distribution  $P$  and  $Q$  is

$$KL(P \parallel Q) = \sum_j \sum_{i \neq j} p_{ij} \log \frac{p_{ij}}{q_{ij}}.$$

For consequences of this definition, see “Helpful Nonlinear Distortion” on page 15-137.

### Gradient Descent of Kullback-Leibler Divergence

To minimize the Kullback-Leibler divergence, the 'exact' algorithm uses a modified gradient descent procedure. The gradient with respect to the points in  $Y$  of the divergence is

$$\frac{\partial KL(P \parallel Q)}{\partial y_i} = 4 \sum_{j \neq i} Z (p_{ij} - q_{ij}) q_{ij} (y_i - y_j),$$

where the normalization term

$$Z = \sum_k \sum_{l \neq k} \left(1 + \|y_k - y_l\|^2\right)^{-1}.$$

The modified gradient descent algorithm uses a few tuning parameters to attempt to reach a good local minimum.

- 'Exaggeration' — During the first 99 gradient descent steps, `tsne` multiplies the probabilities  $p_{ij}$  from “Equation 15-2” on page 15-134 by the exaggeration value. This step tends to create more space between clusters in the output  $Y$ .
- 'LearnRate' — `tsne` uses adaptive learning to improve the convergence of the gradient descent iterations. The descent algorithm has iterative steps that are a linear combination of the previous step in the descent and the current gradient.

'LearnRate' is a multiplier of the current gradient for the linear combination. For details, see Jacobs [3].

## Barnes-Hut Variation of t-SNE

To speed the t-SNE algorithm and to cut down on its memory usage, `tsne` offers an approximate optimization scheme. The Barnes-Hut algorithm groups nearby points together to lower the complexity and memory usage of the t-SNE optimization step. The Barnes-Hut algorithm is an approximate optimizer, not an exact optimizer. There is a nonnegative tuning parameter `Theta` that effects a tradeoff between speed and accuracy. Larger values of 'Theta' give faster but less accurate optimization results. The algorithm is relatively insensitive to 'Theta' values in the range (0.2,0.8).

The Barnes-Hut algorithm groups nearby points in the low-dimensional space, and performs an approximate gradient descent based on these groups. The idea, originally used in astrophysics, is that the gradient is similar for nearby points, so the computations can be simplified.

See van der Maaten [2].

## Characteristics of t-SNE

- “Cannot Use Embedding to Classify New Data” on page 15-136
- “Performance Depends on Data Sizes and Algorithm” on page 15-136
- “Helpful Nonlinear Distortion” on page 15-137

### Cannot Use Embedding to Classify New Data

Because t-SNE often separates data clusters well, it can seem that t-SNE can classify new data points. However, t-SNE cannot classify new points. The t-SNE embedding is a nonlinear map that is data-dependent. To embed a new point in the low-dimensional space, you cannot use the previous embedding as a map. Instead, run the entire algorithm again.

### Performance Depends on Data Sizes and Algorithm

t-SNE can take a good deal of time to process data. If you have  $N$  data points in  $D$  dimensions that you want to map to  $Y$  dimensions, then

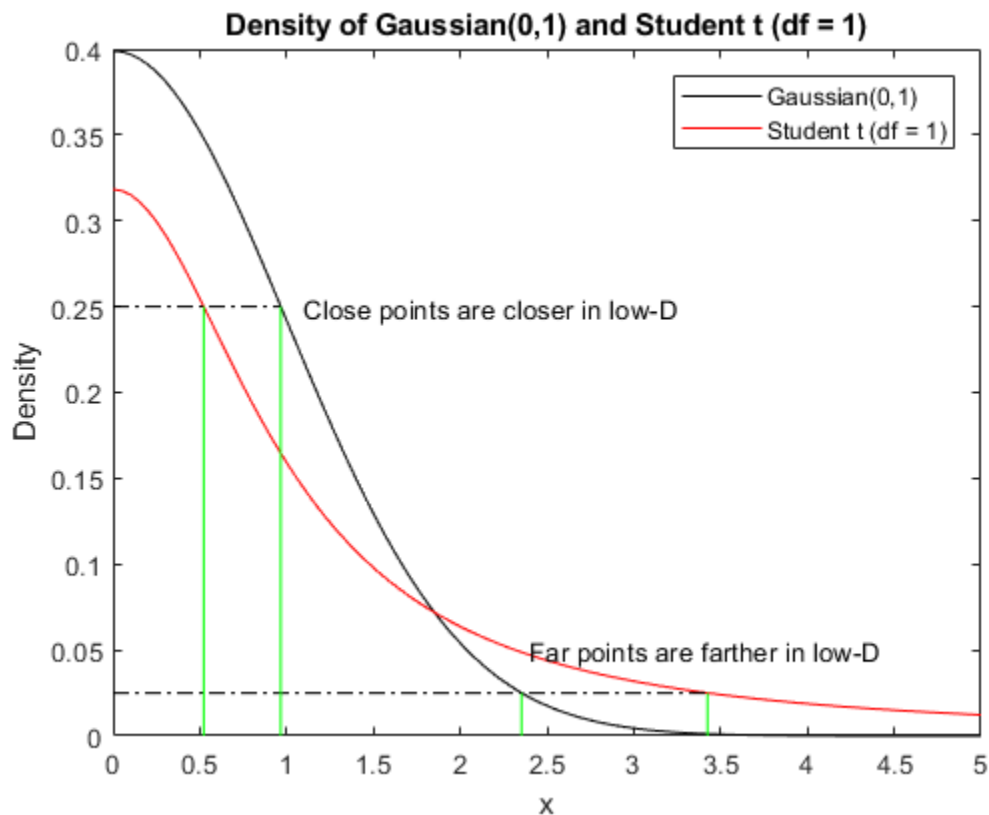
- Exact t-SNE takes of order  $D*N^2$  operations.
- Barnes-Hut t-SNE takes of order  $D*N\log(N)*\exp(\text{dimension}(Y))$  operations.

So for large data sets, where  $N$  is greater than 1000 or so, and where the embedding dimension  $Y$  is 2 or 3, the Barnes-Hut algorithm can be faster than the exact algorithm.

### Helpful Nonlinear Distortion

T-SNE maps high-dimensional distances to distorted low-dimensional analogues. Because of the fatter tail of the Student  $t$  distribution in the low-dimensional space, `tsne` often moves close points closer together, and moves far points farther apart than in the high-dimensional space, as illustrated in the following figure. The figure shows both Gaussian and Student  $t$  distributions at the points where the densities are at 0.25 and 0.025. The Gaussian density relates to high-dimensional distances, and the  $t$  density relates to low-dimensional distances. The  $t$  density corresponds to close points being closer, and far points being farther, compared to the Gaussian density.

```
t = linspace(0,5);
y1 = normpdf(t,0,1);
y2 = tpdf(t,1);
plot(t,y1,'k',t,y2,'r')
hold on
x1 = fzero(@(x)normpdf(x,0,1)-0.25,[0,2]);
x2 = fzero(@(x)tpdf(x,1)-0.25,[0,2]);
z1 = fzero(@(x)normpdf(x,0,1)-0.025,[0,5]);
z2 = fzero(@(x)tpdf(x,1)-0.025,[0,5]);
plot([0,x1],[0.25,0.25],'k-.')
plot([0,z2],[0.025,0.025],'k-.')
plot([x1,x1],[0,0.25],'g-',[x2,x2],[0,0.25],'g-')
plot([z1,z1],[0,0.025],'g-',[z2,z2],[0,0.025],'g-')
text(1.1,.25,'Close points are closer in low-D')
text(2.4,.05,'Far points are farther in low-D')
legend('Gaussian(0,1)','Student t (df = 1)')
xlabel('x')
ylabel('Density')
title('Density of Gaussian(0,1) and Student t (df = 1)')
hold off
```



This distortion is helpful when it applies. It does not apply in cases such as when the Gaussian variance is high, which lowers the Gaussian peak and flattens the distribution. In such a case, `tsne` can move close points farther apart than in the original space. To achieve a helpful distortion,

- Set the 'Verbose' name-value pair to 2.
- Adjust the 'Perplexity' name-value pair so the reported range of variances is not too far from 1, and the mean variance is near 1.

If you can achieve this range of variances, then the diagram applies, and the `tsne` distortion is helpful.

For effective ways to tune `tsne`, see Wattenberg, Viégas and Johnson [4].

## References

- [1] van der Maaten, Laurens, and Geoffrey Hinton. *Visualizing Data using t-SNE*. J. Machine Learning Research 9, 2008, pp. 2579–2605.
- [2] van der Maaten, Laurens. *Barnes-Hut-SNE*. arXiv:1301.3342 [cs.LG], 2013.
- [3] Jacobs, Robert A. *Increased rates of convergence through learning rate adaptation*. Neural Networks 1.4, 1988, pp. 295–307.
- [4] Wattenberg, Martin, Fernanda Viégas, and Ian Johnson. *How to Use t-SNE Effectively*. Distill, 2016. Available at [How to Use t-SNE Effectively](#).

## See Also

### Related Examples

- “Visualize High-Dimensional Data Using t-SNE” on page 15-144
- “t-SNE Output Function” on page 15-140
- “tsne Settings” on page 15-149

### More About

- “Principal Component Analysis (PCA)” on page 15-83
- “Classical Multidimensional Scaling” on page 15-57
- “Factor Analysis” on page 15-97

## t-SNE Output Function

### In this section...

“t-SNE Output Function Description” on page 15-140

“tsne optimValues Structure” on page 15-141

“t-SNE Custom Output Function” on page 15-141

### t-SNE Output Function Description

A `tsne` output function is a function that runs after every `NumPrint` optimization iterations of the t-SNE algorithm. An output function can create plots, or log data to a file or to a workspace variable. The function cannot change the progress of the algorithm, but can halt the iterations.

Set output functions using the `Options` name-value pair argument to the `tsne` function. Set `Options` to a structure created using `statset` or `struct`. Set the `'OutputFcn'` field of the `Options` structure to a function handle or cell array of function handles.

For example, to set an output function named `outfun.m`, use the following commands.

```
opts = statset('OutputFcn',@outfun);
Y = tsne(X,'Options',opts);
```

Write an output function using the following syntax.

```
function stop = outfun(optimValues,state)

stop = false; % do not stop by default
switch state
    case 'init'
        % Set up plots or open files
    case 'iter'
        % Draw plots or update variables
    case 'done'
        % Clean up plots or files
end
```

`tsne` passes the `state` and `optimValues` variables to your function. `state` takes on the values `'init'`, `'iter'`, or `'done'` as shown in the code snippet.

## tsne optimValues Structure

optimValues Field	Description
'iteration'	Iteration number
'fval'	Kullback-Leibler divergence, modified by exaggeration during the first 99 iterations
'grad'	Gradient of the Kullback-Leibler divergence, modified by exaggeration during the first 99 iterations
'Exaggeration'	Value of the exaggeration parameter in use in the current iteration
'Y'	Current embedding

## t-SNE Custom Output Function

This example shows how to use an output function in `tsne`.

### Custom Output Function

The following code is an output function that performs these tasks:

- Keep a history of the Kullback-Leibler divergence and the norm of its gradient in a workspace variable.
- Plot the solution and the history as the iterations proceed.
- Display a `Stop` button on the plot to stop the iterations early without losing any information.

The output function has an extra input variable, `species`, that enables its plots to show the correct classification of the data. For information on including extra parameters such as `species` in a function, see “Parameterizing Functions” (MATLAB).

```
function stop = KLLogging(optimValues,state,species)
persistent h kllog iters stopnow
switch state
    case 'init'
        stopnow = false;
        kllog = [];
        iters = [];
        h = figure;
```

```
c = uicontrol('Style','pushbutton','String','Stop','Position', ...
    [10 10 50 20],'Callback',@stopme);
case 'iter'
    kllog = [kllog; optimValues.fval,log(norm(optimValues.grad))];
    assignin('base','history',kllog)
    iters = [iters; optimValues.iteration];
    if length(iters) > 1
        figure(h)
        subplot(2,1,2)
        plot(iters,kllog);
        xlabel('Iterations')
        ylabel('Loss and Gradient')
        legend('Divergence','log(norm(gradient))')
        title('Divergence and log(norm(gradient))')
        subplot(2,1,1)
        gscatter(optimValues.Y(:,1),optimValues.Y(:,2),species)
        title('Embedding')
        drawnow
    end
case 'done'
    % Nothing here
end
stop = stopnow;

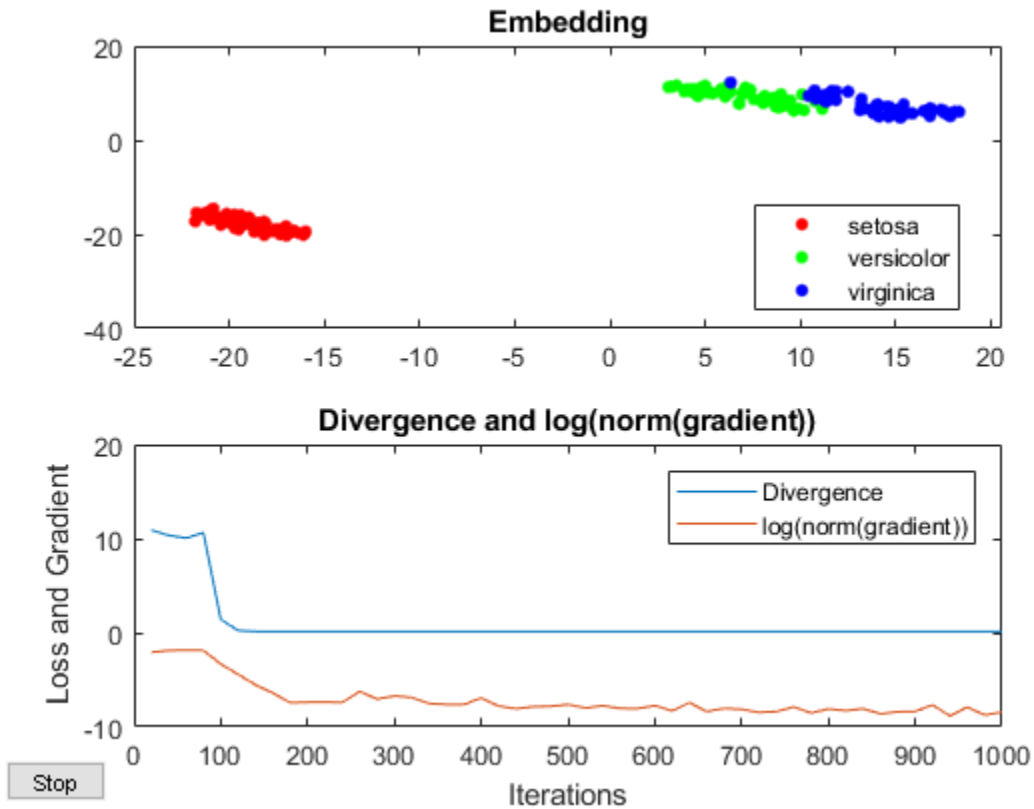
function stopme(~,~)
    stopnow = true;
end
end
```

### Use the Custom Output Function

Plot the Fisher iris data, a 4-D data set, in two dimensions using `tsne`. There is a drop in the Divergence value at iteration 100 because the divergence is scaled by the exaggeration value for earlier iterations. The embedding remains largely unchanged for the last several hundred iterations, so you can save time by clicking the Stop button during the iterations.

```
load fisheriris
rng default % for reproducibility
opts = statset('OutputFcn',@(optimValues,state) KLLogging(optimValues,state,species));
Y = tsne(meas,'Options',opts,'Algorithm','exact');
```





## See Also

### Related Examples

- “Visualize High-Dimensional Data Using t-SNE” on page 15-144
- “tsne Settings” on page 15-149

## Visualize High-Dimensional Data Using t-SNE

This example shows how to visualize the MNIST data, which consists of images of handwritten digits, using the `tsne` function. The images are 28-by-28 pixels in grayscale. Each image has an associated label from 0 through 9, which is the digit that the image represents. `tsne` reduces the dimension of the data from 784 original dimensions to 50 using PCA, and then to two or three using the t-SNE Barnes-Hut algorithm.

### Obtain Data

Begin by obtaining image and label data from

<http://yann.lecun.com/exdb/mnist/>

Unzip the files. For this example, use the `t10k-images` data.

```
imageFileName = 't10k-images.idx3-ubyte';  
labelFileName = 't10k-labels.idx1-ubyte';
```

Process the files to load them in the workspace. The code for this processing function appears at the end of this example. To execute the code, change to the directory of the function.

```
cd(fullfile(matlabroot, 'examples', 'stats', 'main'));  
[X,L] = processMNISTdata(imageFileName,labelFileName);
```

```
Read MNIST image data...  
Number of images in the dataset: 10000 ...  
Each image is of 28 by 28 pixels...  
The image data is read to a matrix of dimensions: 10000 by 784...  
End of reading image data.
```

```
Read MNIST label data...  
Number of labels in the dataset: 10000 ...  
The label data is read to a matrix of dimensions: 10000 by 1...  
End of reading label data.
```

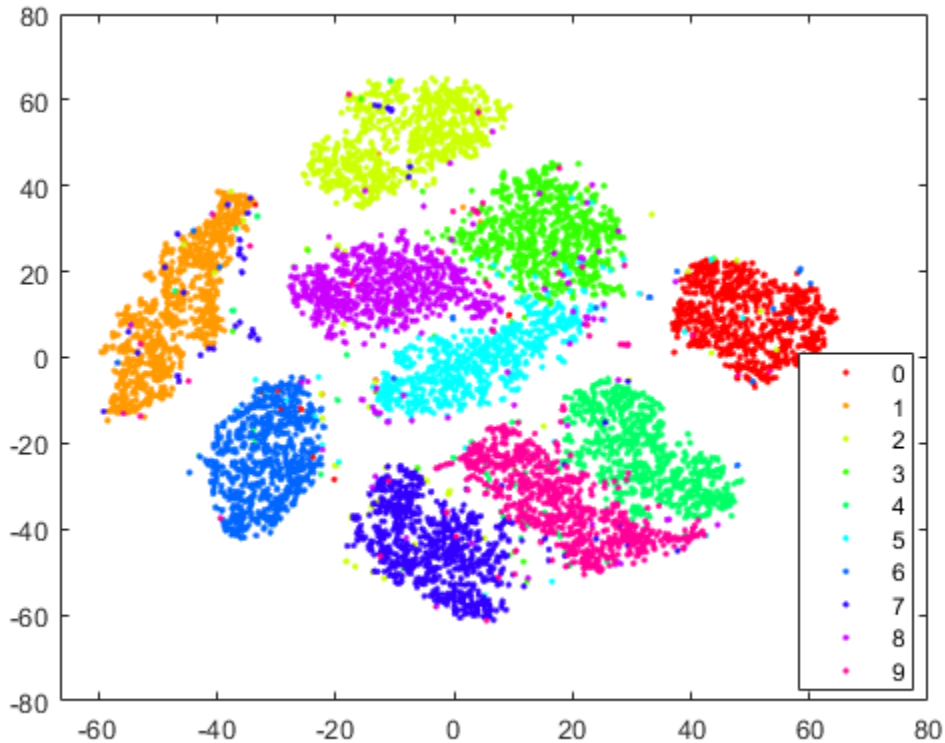
### Reduce Dimension of Data to Two

Obtain two-dimensional analogues of the data clusters using t-SNE. Use PCA to reduce the initial dimensionality to 50. Use the Barnes-Hut variant of the t-SNE algorithm to save time on this relatively large data set.

```
rng default % for reproducibility
Y = tsne(X, 'Algorithm', 'barneshut', 'NumPCAComponents', 50);
```

Display the result, colored with the correct labels.

```
figure
gscatter(Y(:,1), Y(:,2), L)
```



t-SNE creates clusters of points based solely on their relative similarities that correspond closely to the true labels.

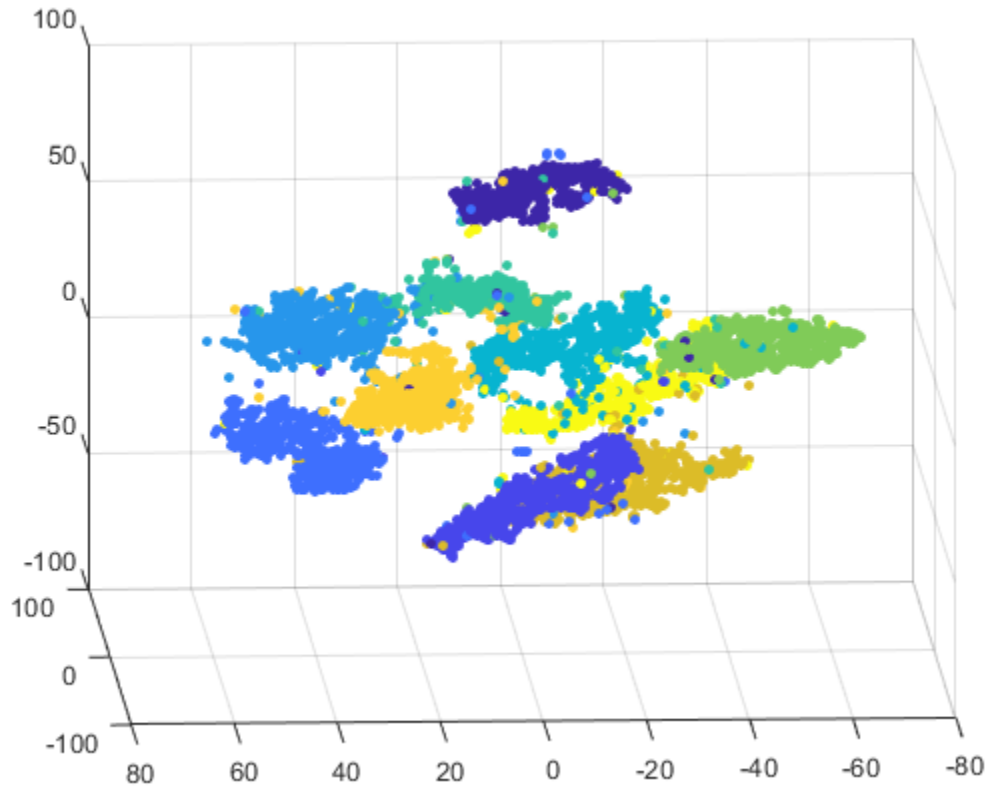
### Reduce Dimension of Data to Three

t-SNE can also reduce the data to three dimensions. Set the `tsne` `'NumDimensions'` name-value pair to 3.

```

rng default % for fair comparison
Y3 = tsne(X,'Algorithm','barneshut','NumPCAComponents',50,'NumDimensions',3);
figure
scatter3(Y3(:,1),Y3(:,2),Y3(:,3),15,L,'filled');
view(-93,14)

```



Here is the code of the function that reads the data into the workspace.

```

function [X,L] = processMNISTdata(imageFileName,labelFileName)

[fileID,errmsg] = fopen(imageFileName,'r','b');
if fileID < 0
    error(errmsg);

```

```

end
%%
% First read the magic number. This number is 2051 for image data, and
% 2049 for label data
magicNum = fread(fileID,1,'int32',0,'b');
if magicNum == 2051
    fprintf('\nRead MNIST image data...\n')
end
%%
% Then read the number of images, number of rows, and number of columns
numImages = fread(fileID,1,'int32',0,'b');
fprintf('Number of images in the dataset: %6d ...\n',numImages);
numRows = fread(fileID,1,'int32',0,'b');
numCols = fread(fileID,1,'int32',0,'b');
fprintf('Each image is of %2d by %2d pixels...\n',numRows,numCols);
%%
% Read the image data
X = fread(fileID,inf,'unsigned char');
%%
% Reshape the data to array X
X = reshape(X,numCols,numRows,numImages);
X = permute(X,[2 1 3]);
%%
% Then flatten each image data into a 1 by (numRows*numCols) vector, and
% store all the image data into a numImages by (numRows*numCols) array.
X = reshape(X,numRows*numCols,numImages)';
fprintf(['The image data is read to a matrix of dimensions: %6d by %4d...\n',...
        'End of reading image data.\n'],size(X,1),size(X,2));
%%
% Close the file
fclose(fileID);
%%
% Similarly, read the label data.
[fileID,errmsg] = fopen(labelFileName,'r','b');
if fileID < 0
    error(errmsg);
end
magicNum = fread(fileID,1,'int32',0,'b');
if magicNum == 2049
    fprintf('\nRead MNIST label data...\n')
end
numItems = fread(fileID,1,'int32',0,'b');
fprintf('Number of labels in the dataset: %6d ...\n',numItems);

```

```
L = fread(fileID,inf,'unsigned char');
fprintf(['The label data is read to a matrix of dimensions: %6d by %2d...\n',...
        'End of reading label data.\n'],size(L,1),size(L,2));
fclose(fileID);
```

## See Also

### Related Examples

- “tsne Settings” on page 15-149

### More About

- “t-SNE” on page 15-132

## tsne Settings

This example shows the effects of various tsne settings.

### Obtain Data

Begin by obtaining image and label data from

<http://yann.lecun.com/exdb/mnist/>

Unzip the files. For this example, use the t10k-images data.

```
imageFileName = 't10k-images.idx3-ubyte';
labelFileName = 't10k-labels.idx1-ubyte';
```

Process the files to load them in the workspace. The code for this processing function appears at the end of this example. To execute the code, add the directory of the function to the search path.

```
addpath(fullfile(matlabroot, 'examples', 'stats')); % add to the path
[X,L] = processMNISTdata(imageFileName, labelFileName);
rmpath(fullfile(matlabroot, 'examples', 'stats')); % remove from the path
```

```
Read MNIST image data...
Number of images in the dataset: 10000 ...
Each image is of 28 by 28 pixels...
The image data is read to a matrix of dimensions: 10000 by 784...
End of reading image data.
```

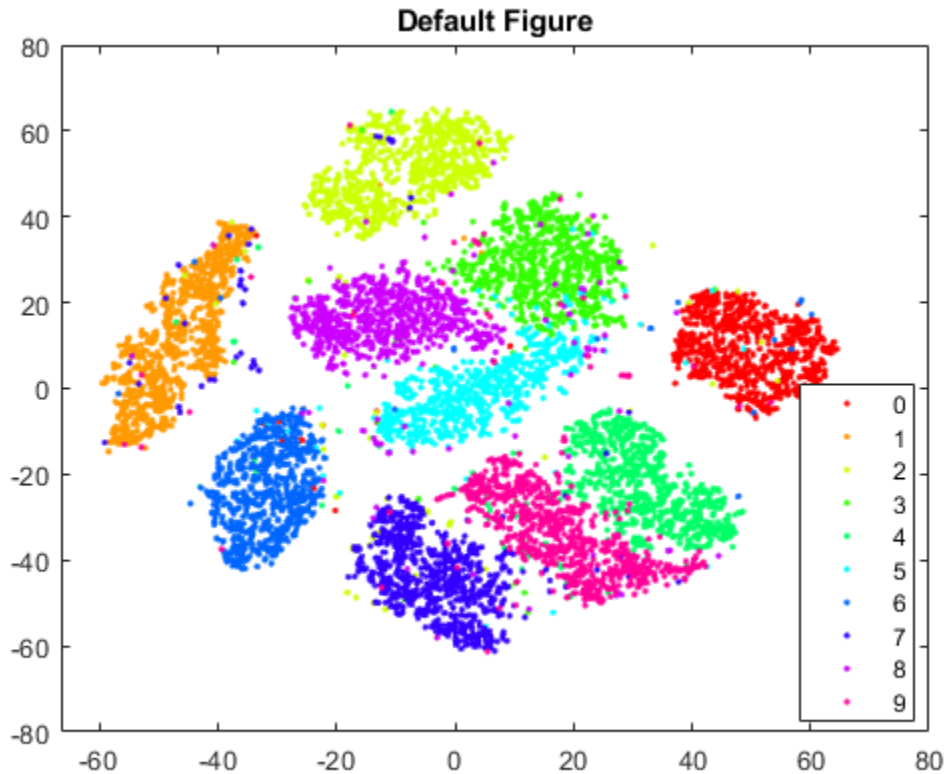
```
Read MNIST label data...
Number of labels in the dataset: 10000 ...
The label data is read to a matrix of dimensions: 10000 by 1...
End of reading label data.
```

### Process Data Using t-SNE

Obtain two-dimensional analogs of the data clusters using t-SNE. Use the Barnes-Hut algorithm for better performance on this large data set. Use PCA to reduce the initial dimensions from 784 to 50.

```
rng default % for reproducibility
Y = tsne(X, 'Algorithm', 'barneshut', 'NumPCAComponents', 50);
figure
```

```
gscatter(Y(:,1),Y(:,2),L)
title('Default Figure')
```



t-SNE creates a figure with well separated clusters and relatively few data points that seem misplaced.

### Perplexity

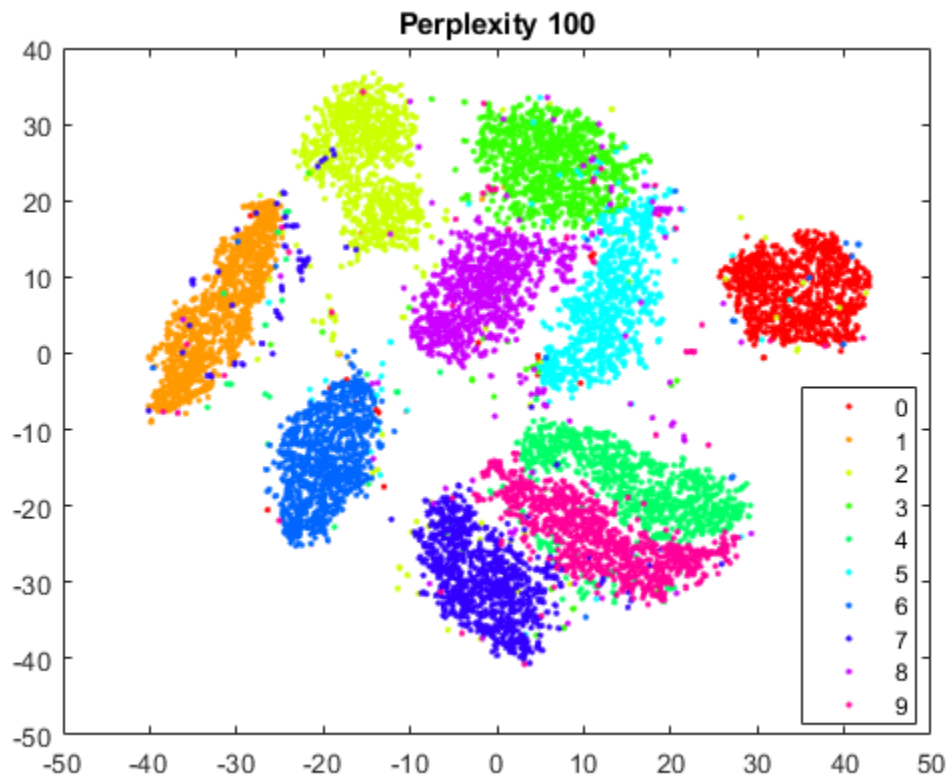
Try altering the perplexity setting to see the effect on the figure.

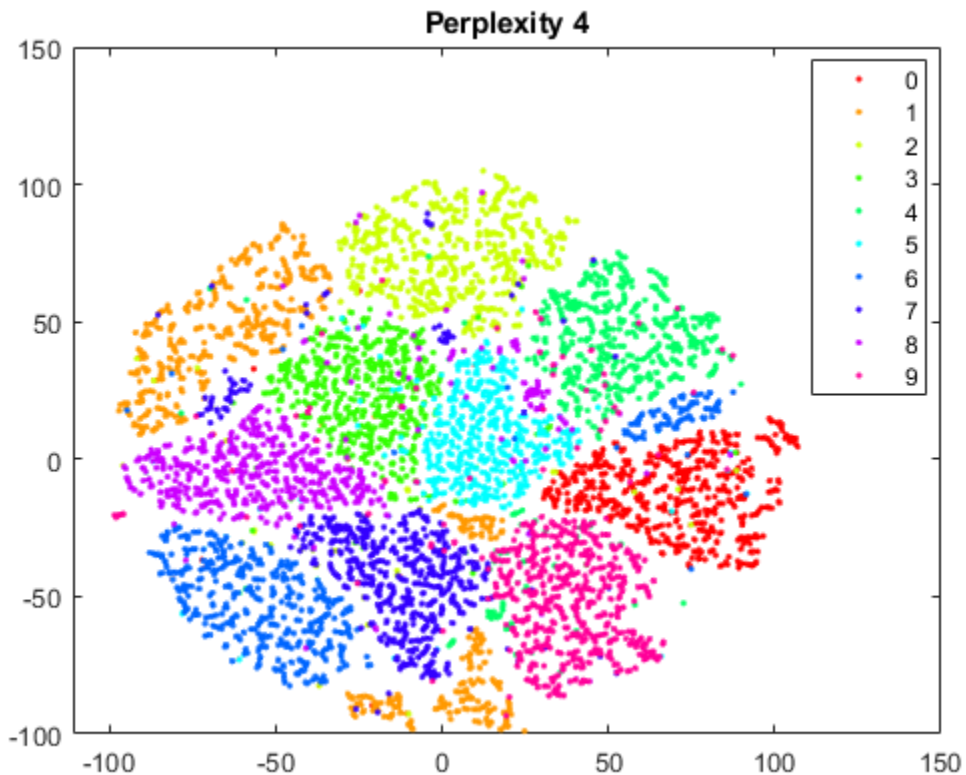
```
rng default % for fair comparison
Y100 = tsne(X,'Algorithm','barneshut','NumPCAComponents',50,'Perplexity',100);
figure
gscatter(Y100(:,1),Y100(:,2),L)
```



```
title('Perplexity 100')

rng default % for fair comparison
Y4 = tsne(X,'Algorithm','barneshut','NumPCAComponents',50,'Perplexity',4);
figure
gscatter(Y4(:,1),Y4(:,2),L)
title('Perplexity 4')
```





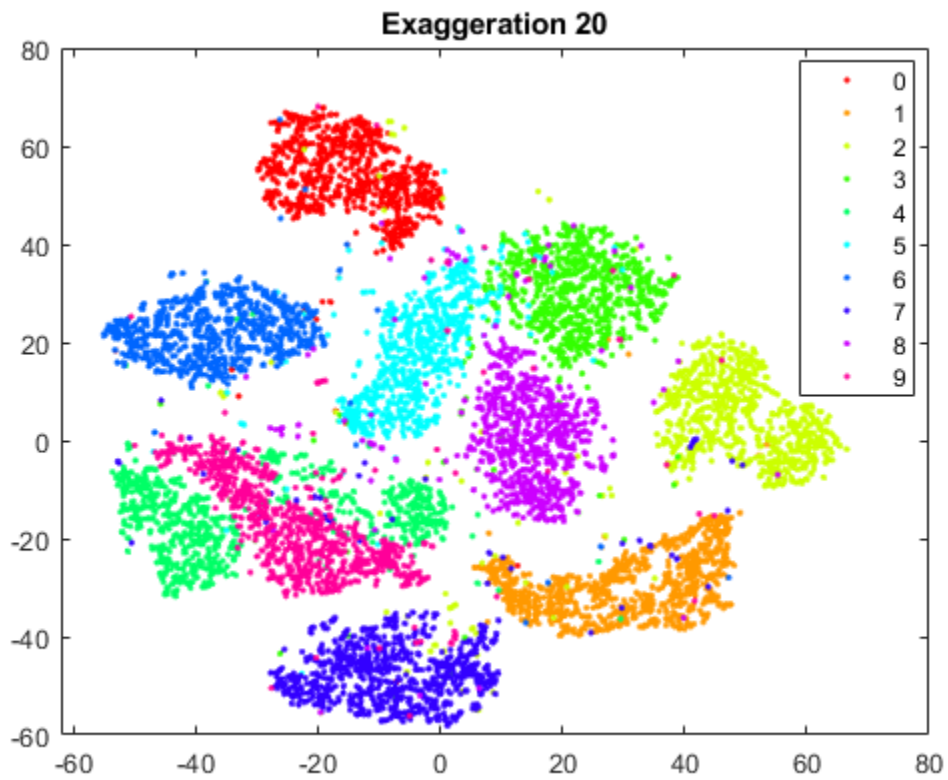
Setting the perplexity to 100 yields a figure that is largely similar to the default figure. The clusters are tighter than with the default setting. However, setting the perplexity to 4 gives a figure without well separated clusters. The clusters are looser than with the default setting.

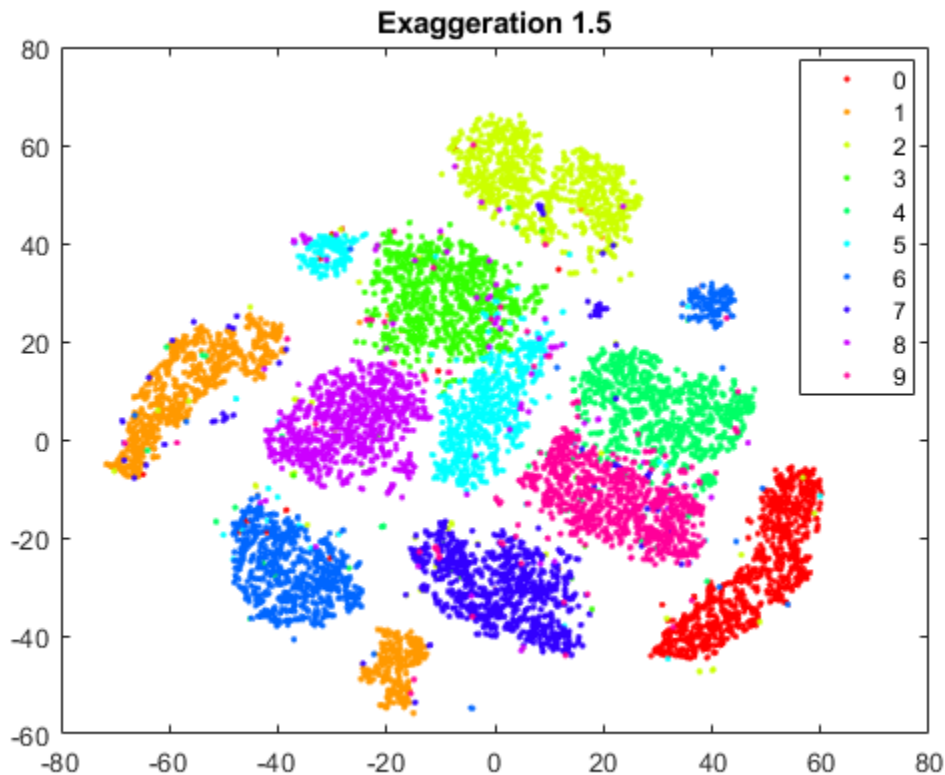
### Exaggeration

Try altering the exaggeration setting to see the effect on the figure.

```
rng default % for fair comparison
YEX0 = tsne(X, 'Algorithm', 'barneshut', 'NumPCAComponents', 50, 'Exaggeration', 20);
figure
gscatter(YEX0(:,1), YEX0(:,2), L)
title('Exaggeration 20')
```

```
rng default % for fair comparison
YEx15 = tsne(X, 'Algorithm', 'barnesht', 'NumPCAComponents', 50, 'Exaggeration', 1.5);
figure
gscatter(YEx15(:,1), YEx15(:,2), L)
title('Exaggeration 1.5')
```





While the exaggeration setting has an effect on the figure, it is not clear whether any nondefault setting gives a better picture than the default setting. The figure with an exaggeration of 20 is similar to the default figure. In general, a larger exaggeration creates more empty space between embedded clusters. An exaggeration of 1.5 causes the groups labeled 1 and 6 to split into two groups each, an undesirable outcome.

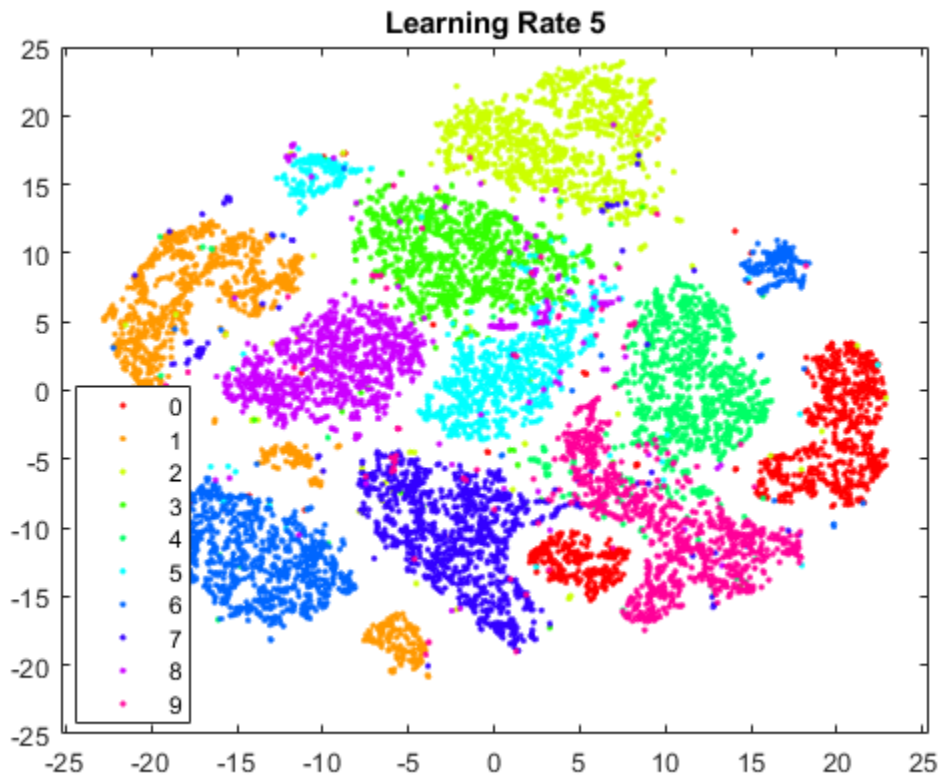
Exaggerating the values in the joint distribution of  $X$  makes the values in the joint distribution of  $Y$  smaller. This makes it much easier for the embedded points to move relative to one another. The splitting of clusters 1 and 6 reflects this effect.

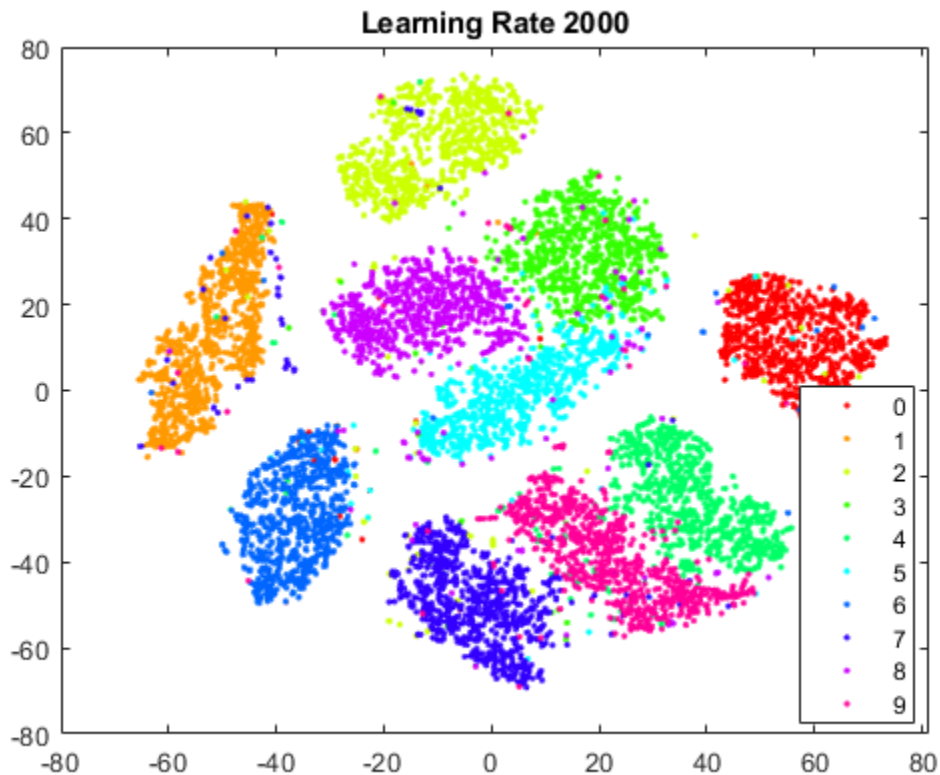
### Learning Rate

Try altering the learning rate setting to see the effect on the figure.

```
rng default % for fair comparison
YL5 = tsne(X,'Algorithm','barneshut','NumPCAComponents',50,'LearnRate',5);
figure
gscatter(YL5(:,1),YL5(:,2),L)
title('Learning Rate 5')

rng default % for fair comparison
YL2000 = tsne(X,'Algorithm','barneshut','NumPCAComponents',50,'LearnRate',2000);
figure
gscatter(YL2000(:,1),YL2000(:,2),L)
title('Learning Rate 2000')
```





The figure with a learning rate of 5 has several clusters that split into two or more pieces. This shows that if the learning rate is too small, the minimization process can get stuck in a bad local minimum. A learning rate of 2000 gives a figure similar to the default figure.

#### Initial Behavior with Various Settings

Large learning rates or large exaggeration values can lead to undesirable initial behavior. To see this, set large values of these parameters and set `NumPrint` and `Verbose` to 1 to show all the iterations. Stop the iterations after 10, as the goal of this experiment is simply to look at the initial behavior.

Begin by setting the exaggeration to 200.

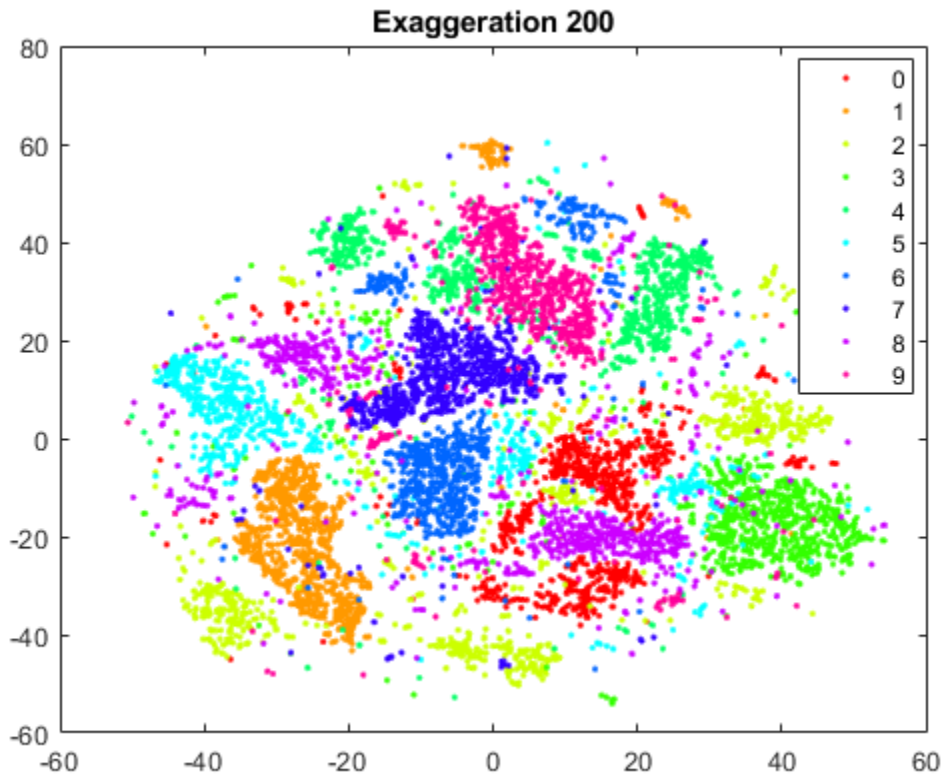
```
rng default % for fair comparison
opts = statset('MaxIter',10);
YEX200 = tsne(X, 'Algorithm', 'barneshut', 'NumPCAComponents', 50, 'Exaggeration', 200, ...
    'NumPrint', 1, 'Verbose', 1, 'Options', opts);
```

```
=====|
| ITER | KL DIVERGENCE | NORM GRAD USING |
|      | FUN VALUE USING | EXAGGERATED DIST |
|      | EXAGGERATED DIST | OF X |
|      | OF X |
|=====|
| 1 | 2.190347e+03 | 6.078667e-05 |
| 2 | 2.190352e+03 | 4.769050e-03 |
| 3 | 2.204061e+03 | 9.423678e-02 |
| 4 | 2.464585e+03 | 2.113271e-02 |
| 5 | 2.501222e+03 | 2.616407e-02 |
| 6 | 2.529362e+03 | 3.022570e-02 |
| 7 | 2.553233e+03 | 3.108418e-02 |
| 8 | 2.562822e+03 | 3.278873e-02 |
| 9 | 2.538056e+03 | 3.222265e-02 |
| 10 | 2.504932e+03 | 3.671708e-02 |
```

The Kullback-Leibler divergence increases during the first few iterations, and the norm of the gradient increases as well.

To see the final result of the embedding, allow the algorithm to run to completion using the default stopping criteria.

```
rng default % for fair comparison
YEX200 = tsne(X, 'Algorithm', 'barneshut', 'NumPCAComponents', 50, 'Exaggeration', 200);
figure
gscatter(YEX200(:,1), YEX200(:,2), L)
title('Exaggeration 200')
```



This exaggeration value does not give a clean separation into clusters.

Show the initial behavior when the learning rate is 100,000.

```
rng default % for fair comparison
YL100k = tsne(X,'Algorithm','barneshut','NumPCAComponents',50,'LearnRate',1e5,...
    'NumPrint',1,'Verbose',1,'Options',opts);
```

```
=====|
| ITER   | KL DIVERGENCE   | NORM GRAD USING |
|        | FUN VALUE USING | EXAGGERATED DIST|
|        | EXAGGERATED DIST| OF X             |
|        | OF X           |                   |
```

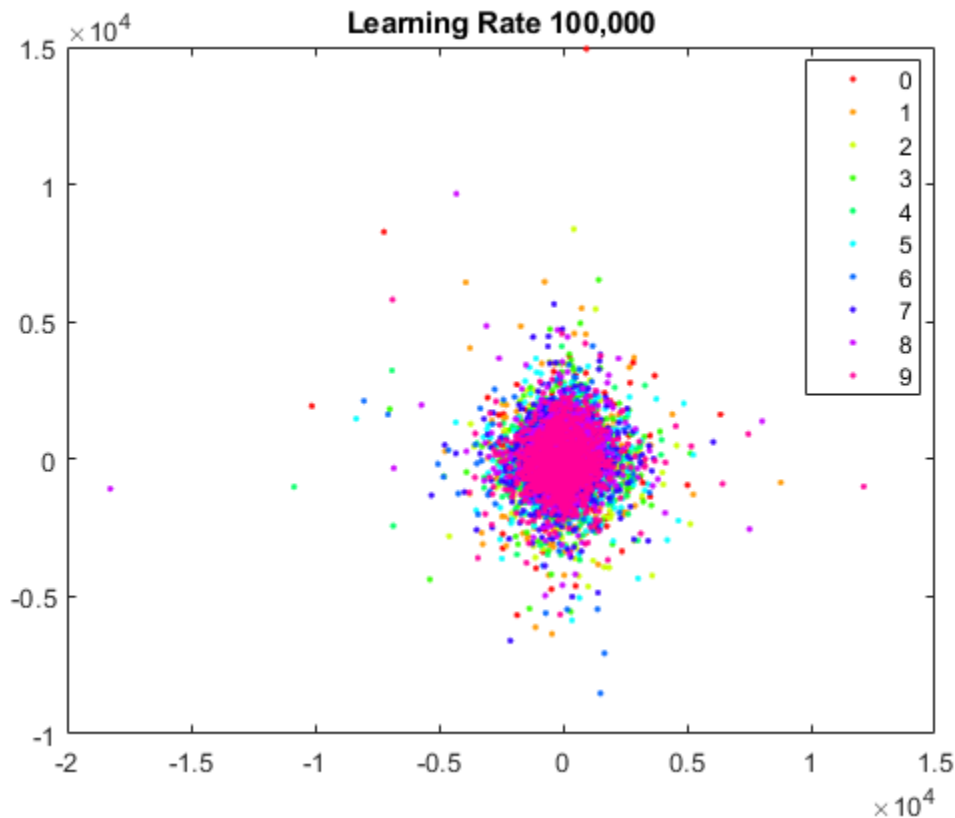


```
|=====|
|      1 |      2.815885e+01 |      1.024049e-06 |
|      2 |      2.816002e+01 |      2.902059e-04 |
|      3 |      3.195873e+01 |      7.355889e-04 |
|      4 |      3.348151e+01 |      3.958901e-04 |
|      5 |      3.365935e+01 |      2.876905e-04 |
|      6 |      3.342462e+01 |      3.906245e-04 |
|      7 |      3.303205e+01 |      4.037983e-04 |
|      8 |      3.263320e+01 |      5.665630e-04 |
|      9 |      3.235384e+01 |      4.319099e-04 |
|     10 |      3.211238e+01 |      4.803526e-04 |
```

Again, the Kullback-Leibler divergence increases during the first few iterations, and the norm of the gradient increases as well.

To see the final result of the embedding, allow the algorithm to run to completion using the default stopping criteria.

```
rng default % for fair comparison
YL100k = tsne(X, 'Algorithm', 'barneshut', 'NumPCAComponents', 50, 'LearnRate', 1e5);
figure
gscatter(YL100k(:,1), YL100k(:,2), L)
title('Learning Rate 100,000')
```



The learning rate is far too large, and gives no useful embedding.

### Conclusion

`tsne` with default settings does a good job of embedding the high-dimensional initial data into two-dimensional points that have well defined clusters. The effects of algorithm settings are difficult to predict. Sometimes they can improve the clustering, but for the most part the default settings seem good. While speed is not part of this investigation, settings can affect the speed of the algorithm. In particular, the Barnes-Hut algorithm is notably faster on this data.

### Code to Process MNIST Data

Here is the code of the function that reads the data into the workspace.

```

function [X,L] = processMNISTdata(imageFileName,labelFileName)

[fileID,errmsg] = fopen(imageFileName,'r','b');
if fileID < 0
    error(errmsg);
end
%%
% First read the magic number. This number is 2051 for image data, and
% 2049 for label data
magicNum = fread(fileID,1,'int32',0,'b');
if magicNum == 2051
    fprintf('\nRead MNIST image data...\n')
end
%%
% Then read the number of images, number of rows, and number of columns
numImages = fread(fileID,1,'int32',0,'b');
fprintf('Number of images in the dataset: %6d ...\n',numImages);
numRows = fread(fileID,1,'int32',0,'b');
numCols = fread(fileID,1,'int32',0,'b');
fprintf('Each image is of %2d by %2d pixels...\n',numRows,numCols);
%%
% Read the image data
X = fread(fileID,inf,'unsigned char');
%%
% Reshape the data to array X
X = reshape(X,numCols,numRows,numImages);
X = permute(X,[2 1 3]);
%%
% Then flatten each image data into a 1 by (numRows*numCols) vector, and
% store all the image data into a numImages by (numRows*numCols) array.
X = reshape(X,numRows*numCols,numImages)';
fprintf(['The image data is read to a matrix of dimensions: %6d by %4d...\n',...
        'End of reading image data.\n'],size(X,1),size(X,2));
%%
% Close the file
fclose(fileID);
%%
% Similarly, read the label data.
[fileID,errmsg] = fopen(labelFileName,'r','b');
if fileID < 0
    error(errmsg);
end
magicNum = fread(fileID,1,'int32',0,'b');

```

```
if magicNum == 2049
    fprintf('\nRead MNIST label data...\n')
end
numItems = fread(fileID,1,'int32',0,'b');
fprintf('Number of labels in the dataset: %6d ...\n',numItems);

L = fread(fileID,inf,'unsigned char');
fprintf(['The label data is read to a matrix of dimensions: %6d by %2d...\n',...
        'End of reading label data.\n'],size(L,1),size(L,2));
fclose(fileID);
```

## See Also

### Related Examples

- “Visualize High-Dimensional Data Using t-SNE” on page 15-144
- “t-SNE Output Function” on page 15-140

### More About

- “t-SNE” on page 15-132

### External Websites

- [How to Use t-SNE Effectively](#)

## Feature Extraction

In this section...
“What Is Feature Extraction?” on page 15-163
“Sparse Filtering Algorithm” on page 15-163
“Reconstruction ICA Algorithm” on page 15-166

### What Is Feature Extraction?

Feature extraction is a set of methods that map input features to new output features. Many feature extraction methods use unsupervised learning to extract features. Unlike some feature extraction methods such as PCA and NNMF, the methods described in this section can increase dimensionality (and decrease dimensionality). Internally, the methods involve optimizing nonlinear objective functions. For details, see “Sparse Filtering Algorithm” on page 15-163 or “Reconstruction ICA Algorithm” on page 15-166.

One typical use of feature extraction is finding features in images. Using these features can lead to improved classification accuracy. For an example, see “Feature Extraction Workflow” on page 15-169. Another typical use is extracting individual signals from superpositions, which is often termed blind source separation. For an example, see “Extract Mixed Signals” on page 15-205.

There are two feature extraction functions: `rica` and `sparsefilt`. Associated with these functions are the objects that they create: `ReconstructionICA` and `SparseFiltering`.

### Sparse Filtering Algorithm

The sparse filtering algorithm begins with a data matrix  $X$  that has  $n$  rows and  $p$  columns. Each row represents one observation and each column represents one measurement. The columns are also called the features or predictors. The algorithm then takes either an initial random  $p$ -by- $q$  weight matrix  $W$  or uses the weight matrix passed in the `InitialTransformWeights` name-value pair.  $q$  is the requested number of features that `sparsefilt` computes.

The algorithm attempts to minimize the “Sparse Filtering Objective Function” on page 15-164 by using a standard limited memory Broyden-Fletcher-Goldfarb-Shanno (LBFGS) quasi-Newton optimizer. See Nocedal and Wright [2]. This optimizer takes up to

`IterationLimit` iterations. It stops iterating earlier when it takes a step whose norm is less than `StepTolerance`, or when it computes that the norm of the gradient at the current point is less than `GradientTolerance` times a scalar  $\tau$ , where

$$\tau = \max\left(1, \min\left(|f|, \|g_0\|_\infty\right)\right).$$

$|f|$  is the norm of the objective function, and  $\|g_0\|_\infty$  is the infinity norm of the initial gradient.

The objective function attempts to simultaneously obtain few nonzero features for each data point, and for each resulting feature to have nearly equal weight. To understand how the objective function attempts to achieve these goals, see Ngiam, Koh, Chen, Bhaksar, and Ng [1].

Frequently, you obtain good features by setting a relatively small value of `IterationLimit`, from as low as 5 to a few hundred. Allowing the optimizer to continue can result in overtraining, where the extracted features do not generalize well to new data.

After constructing a `SparseFiltering` object, use the `transform` method to map input data to the new output features.

### Sparse Filtering Objective Function

To compute an objective function, the sparse filtering algorithm uses the following steps. The objective function depends on the  $n$ -by- $p$  data matrix  $X$  and a weight matrix  $W$  that the optimizer varies. The weight matrix  $W$  has dimensions  $p$ -by- $q$ , where  $p$  is the number of original features and  $q$  is the number of requested features.

- 1 Compute the  $n$ -by- $q$  matrix  $X*W$ . Apply the approximate absolute value function

$\phi(u) = \sqrt{u^2 + 10^{-8}}$  to each element of  $X*W$  to obtain the matrix  $F$ .  $\phi$  is a smooth nonnegative symmetric function that closely approximates the absolute value function.

- 2 Normalize the columns of  $F$  by the approximate  $L^2$  norm. In other words, define the normalized matrix  $\tilde{F}(i, j)$  by

$$\|F(j)\| = \sqrt{\sum_{i=1}^n (F(i, j))^2 + 10^{-8}}$$

$$\tilde{F}(i, j) = F(i, j) / \|F(j)\|.$$

- 3 Normalize the rows of  $\tilde{F}(i, j)$  by the approximate  $L^2$  norm. In other words, define the normalized matrix  $\hat{F}(i, j)$  by

$$\|\tilde{F}(i)\| = \sqrt{\sum_{j=1}^q (\tilde{F}(i, j))^2 + 10^{-8}}$$

$$F(i, j) = \tilde{F}(i, j) / \|\tilde{F}(i)\|.$$

The matrix  $\hat{F}$  is the matrix of converted features in  $X$ . Once `sparsefilt` finds the weights  $W$  that minimize the objective function  $h$  (see below), which the function stores in the output object `Mdl` in the `Mdl.TransformWeights` property, the `transform` function can follow the same transformation steps to convert new data to output features.

- 4 Compute the objective function  $h(W)$  as the 1-norm of the matrix  $\hat{F}(i, j)$ , meaning the sum of all the elements in the matrix (which are nonnegative by construction):

$$h(W) = \sum_{j=1}^q \sum_{i=1}^n \hat{F}(i, j).$$

- 5 If you set the `Lambda` name-value pair to a strictly positive value, `sparsefilt` uses the following modified objective function:

$$h(W) = \sum_{j=1}^q \sum_{i=1}^n \hat{F}(i, j) + \lambda \sum_{j=1}^q w_j^T w_j.$$

Here,  $w_j$  is the  $j$ th column of the matrix  $W$  and  $\lambda$  is the value of `Lambda`. The effect of this term is to shrink the weights  $W$ . If you plot the columns of  $W$  as images, with positive `Lambda` these images appear smooth compared to the same images with zero `Lambda`.

## Reconstruction ICA Algorithm

The Reconstruction Independent Component Analysis (RICA) algorithm is based on minimizing an objective function. The algorithm maps input data to output features.

The ICA source model is the following. Each observation  $x$  is generated by a random vector  $s$  according to

$$x = \mu + As.$$

- $x$  is a column vector of length  $p$ .
- $\mu$  is a column vector of length  $p$  representing a constant term.
- $s$  is a column vector of length  $q$  whose elements are zero mean, unit variance random variables that are statistically independent of each other.
- $A$  is a mixing matrix of size  $p$ -by- $q$ .

You can use this model in `rica` to estimate  $A$  from observations of  $x$ . See “Extract Mixed Signals” on page 15-205.

The RICA algorithm begins with a data matrix  $X$  that has  $n$  rows and  $p$  columns consisting of the observations  $x_i$ :

$$X = \begin{bmatrix} x_1^T \\ x_2^T \\ \vdots \\ x_n^T \end{bmatrix}.$$

Each row represents one observation and each column represents one measurement. The columns are also called the features or predictors. The algorithm then takes either an initial random  $p$ -by- $q$  weight matrix  $W$  or uses the weight matrix passed in the `InitialTransformWeights` name-value pair.  $q$  is the requested number of features that `rica` computes. The weight matrix  $W$  is composed of columns  $w_i$  of size  $p$ -by-1:

$$W = [w_1 \quad w_2 \quad \dots \quad w_q].$$

The algorithm attempts to minimize the “Reconstruction ICA Objective Function” on page 15-167 by using a standard limited memory Broyden-Fletcher-Goldfarb-Shanno (LBFGS) quasi-Newton optimizer. See Nocedal and Wright [2]. This optimizer takes up to `IterationLimit` iterations. It stops iterating when it takes a step whose norm is less



than `StepTolerance`, or when it computes that the norm of the gradient at the current point is less than `GradientTolerance` times a scalar  $\tau$ , where

$$\tau = \max\left(1, \min\left(|f|, \|g_0\|_\infty\right)\right).$$

$|f|$  is the norm of the objective function, and  $\|g_0\|_\infty$  is the infinity norm of the initial gradient.

The objective function attempts to obtain a nearly orthonormal weight matrix that minimizes the sum of elements of  $g(XW)$ , where  $g$  is a function (described below) that is applied elementwise to  $XW$ . To understand how the objective function attempts to achieve these goals, see Le, Karpenko, Ngiam, and Ng [3].

After constructing a `ReconstructionICA` object, use the `transform` method to map input data to the new output features.

### Reconstruction ICA Objective Function

The objective function uses a contrast function, which you specify by using the `ContrastFcn` name-value pair. The contrast function is a smooth convex function that is

similar to an absolute value. By default, the contrast function is  $g = \frac{1}{2} \log(\cosh(2x))$ . For other available contrast functions, see `ContrastFcn`.

For an  $n$ -by- $p$  data matrix  $X$  and  $q$  output features, with a regularization parameter  $\lambda$  as the value of the `Lambda` name-value pair, the objective function in terms of the  $p$ -by- $q$  matrix  $W$  is

$$h = \frac{\lambda}{n} \sum_{i=1}^n \|WW^T x_i - x_i\|_2^2 + \frac{1}{n} \sum_{i=1}^n \sum_{j=1}^q \sigma_j g(w_j^T x_i)$$

The  $\sigma_j$  are known constants that are  $\pm 1$ . When  $\sigma_j = +1$ , minimizing the objective function

$h$  encourages the histogram of  $w_j^T x_i$  to be sharply peaked at 0 (super Gaussian). When

$\sigma_j = -1$ , minimizing the objective function  $h$  encourages the histogram of  $w_j^T x_i$  to be flatter near 0 (sub Gaussian). Specify the  $\sigma_j$  values using the `ricaNonGaussianityIndicator` name-value pair.

The objective function  $h$  can have a spurious minimum of zero when  $\lambda$  is zero. Therefore, `rica` minimizes  $h$  over  $W$  that are normalized to 1. In other words, each column  $w_j$  of  $W$  is defined in terms of a column vector  $v_j$  by

$$w_j = \frac{v_j}{\sqrt{v_j^T v_j + 10^{-8}}}.$$

`rica` minimizes over the  $v_j$ . The resulting minimal matrix  $W$  provides the transformation from input data  $X$  to output features  $XW$ .

## References

- [1] Ngiam, Jiquan, Zhenghao Chen, Sonia A. Bhaskar, Pang W. Koh, and Andrew Y. Ng. “Sparse Filtering.” *Advances in Neural Information Processing Systems*. Vol. 24, 2011, pp. 1125–1133. <https://papers.nips.cc/paper/4334-sparse-filtering.pdf>.
- [2] Nocedal, J. and S. J. Wright. *Numerical Optimization*, Second Edition. Springer Series in Operations Research, Springer Verlag, 2006.
- [3] Le, Quoc V., Alexandre Karpenko, Jiquan Ngiam, and Andrew Y. Ng. “ICA with Reconstruction Cost for Efficient Overcomplete Feature Learning.” *Advances in Neural Information Processing Systems*. Vol. 24, 2011, pp. 1017–1025. <https://papers.nips.cc/paper/4467-ica-with-reconstruction-cost-for-efficient-overcomplete-feature-learning.pdf>.

## See Also

`ReconstructionICA` | `SparseFiltering` | `rica` | `sparsefilt`

## Related Examples

- “Feature Extraction Workflow” on page 15-169
- “Extract Mixed Signals” on page 15-205

## Feature Extraction Workflow

This example shows a complete workflow for feature extraction from image data.

### Obtain Data

This example uses the MNIST image data, which consists of images of handwritten digits. The images are 28-by-28 pixels in grayscale. Each image has an associated label from 0 through 9, which is the digit that the image represents.

Begin by obtaining image and label data from

<http://yann.lecun.com/exdb/mnist/>

Unzip the files. For better performance on this long example, use the test data as training data and the training data as test data.

```
imageFileName = 't10k-images.idx3-ubyte';  
labelFileName = 't10k-labels.idx1-ubyte';
```

Process the files to load them in the workspace. The code for this processing function appears at the end of this example.

```
[Xtrain,LabelTrain] = processMNISTdata(imageFileName,labelFileName);  
numrows = size(Xtrain,1);
```

```
Read MNIST image data...  
Number of images in the dataset: 10000 ...  
Each image is of 28 by 28 pixels...  
The image data is read to a matrix of dimensions: 10000 by 784...  
End of reading image data.
```

```
Read MNIST label data...  
Number of labels in the dataset: 10000 ...  
The label data is read to a matrix of dimensions: 10000 by 1...  
End of reading label data.
```

View a few of the images.

```
rng default % For reproducibility  
ims = randi(numrows,4,1);  
imgs = Xtrain(ims,:);  
for i = 1:4
```

```
pp{i} = reshape(imgs(i,:),28,28);  
end  
ppf = [pp{1},pp{2};pp{3},pp{4}];  
imshow(ppf);
```



### Choose New Feature Dimensions

There are several considerations in choosing the number of features to extract:

- More features use more memory and computational time.
- Fewer features can produce a poor classifier.

For this example, choose 100 features.

```
q = 100;
```

### Extract Features

There are two feature extraction functions, `sparsefilt` and `rica`. Begin with the `sparsefilt` function. Set the number of iterations to 10 so that the extraction does not take too long.

Typically, you get good results by running the `sparsefilt` algorithm for a few iterations to a few hundred iterations. Running the algorithm for too many iterations can lead to decreased classification accuracy, a type of overfitting problem.

Use `sparsefilt` to obtain the sparse filtering model while using 10 iterations.

```
Mdl = sparsefilt(Xtrain,q,'IterationLimit',10);
```

```
Warning: Solver LBFGS was not able to converge to a solution.
```

`sparsefilt` warns that the internal LBFGS optimizer did not converge. The optimizer did not converge because you set the iteration limit to 10. Nevertheless, you can use the result to train a classifier.

## Create Classifier

Transform the original data into the new feature representation.

```
NewX = transform(Mdl,Xtrain);
```

Train a linear classifier based on the transformed data and the correct classification labels in `LabelTrain`. The accuracy of the learned model is sensitive to the `fitcecoc` regularization parameter `Lambda`. Try to find the best value for `Lambda` by using the `OptimizeHyperparameters` name-value pair. Be aware that this optimization takes time. If you have a Parallel Computing Toolbox™ license, use parallel computing for faster execution. If you don't have a parallel license, remove the `UseParallel` calls before running this script.

```
t = templateLinear('Solver','lbfgs');
options = struct('UseParallel',true);
Cmdl = fitcecoc(NewX,LabelTrain,'Learners',t,...
    'OptimizeHyperparameters',{'Lambda'},...
    'HyperparameterOptimizationOptions',options);
```

Copying objective function to workers...

Done copying objective function to workers.

Iter	Parallel jobs	Eval result	Objective	Objective runtime	BestSoFar (observed)	BestSoFar (estim.)
1	4	Best	0.8865	6.1325	0.8865	0.8865
2	4	Best	0.2041	6.8636	0.2041	0.33802
3	4	Best	0.1091	12.571	0.1091	0.18238
4	4	Accept	0.2042	6.5827	0.1091	0.10913
5	4	Accept	0.2311	6.0866	0.1091	0.10912
6	4	Best	0.0973	20.342	0.0973	0.097317
7	4	Best	0.0946	14.394	0.0946	0.094625
8	4	Accept	0.1456	7.7484	0.0946	0.09457
9	4	Accept	0.1071	13.472	0.0946	0.094599
10	4	Accept	0.1057	11.223	0.0946	0.094853
11	4	Accept	0.1027	16.709	0.0946	0.095012
12	4	Best	0.0929	17.644	0.0929	0.093437
13	4	Best	0.0911	15.599	0.0911	0.092669
14	4	Accept	0.0933	16.775	0.0911	0.092679

15	4	Accept	0.1082	12.626	0.0911	0.092683	1.00	
16	4	Accept	0.093	15.941	0.0911	0.092573	6.52	
17	4	Accept	0.7999	5.6052	0.0911	0.092397	0	
18	4	Accept	0.105	15.386	0.0911	0.09239	1.23	
19	4	Accept	0.1075	12.329	0.0911	0.092385	7.50	
20	4	Accept	0.117	9.2747	0.0911	0.092402	0.0	
=====								
Iter	Parallel	Eval	Objective	Objective	BestSoFar	BestSoFar		
	jobs	result		runtime	(observed)	(estim.)		
=====								
21	4	Accept	0.101	17.878	0.0911	0.092399	2.07	
22	4	Accept	0.0915	15.524	0.0911	0.091934	8.72	
23	4	Accept	0.0972	12.625	0.0911	0.092009	0.00	
24	4	Accept	0.097	19.339	0.0911	0.091991	1.89	
25	4	Accept	0.0917	15.805	0.0911	0.091911	8.26	
26	4	Accept	0.5107	5.4732	0.0911	0.091839	0	
27	4	Accept	0.8865	5.7215	0.0911	0.091885		
28	4	Accept	0.0915	15.617	0.0911	0.0918	8.49	
29	4	Accept	0.1075	12.487	0.0911	0.091804	1.49	
30	4	Accept	0.1292	8.5929	0.0911	0.091807	0.0	

---

Optimization completed.

MaxObjectiveEvaluations of 30 reached.

Total function evaluations: 30

Total elapsed time: 105.1449 seconds.

Total objective function evaluation time: 372.368

Best observed feasible point:

Lambda

-----  
8.337e-05

Observed objective function value = 0.0911

Estimated objective function value = 0.091807

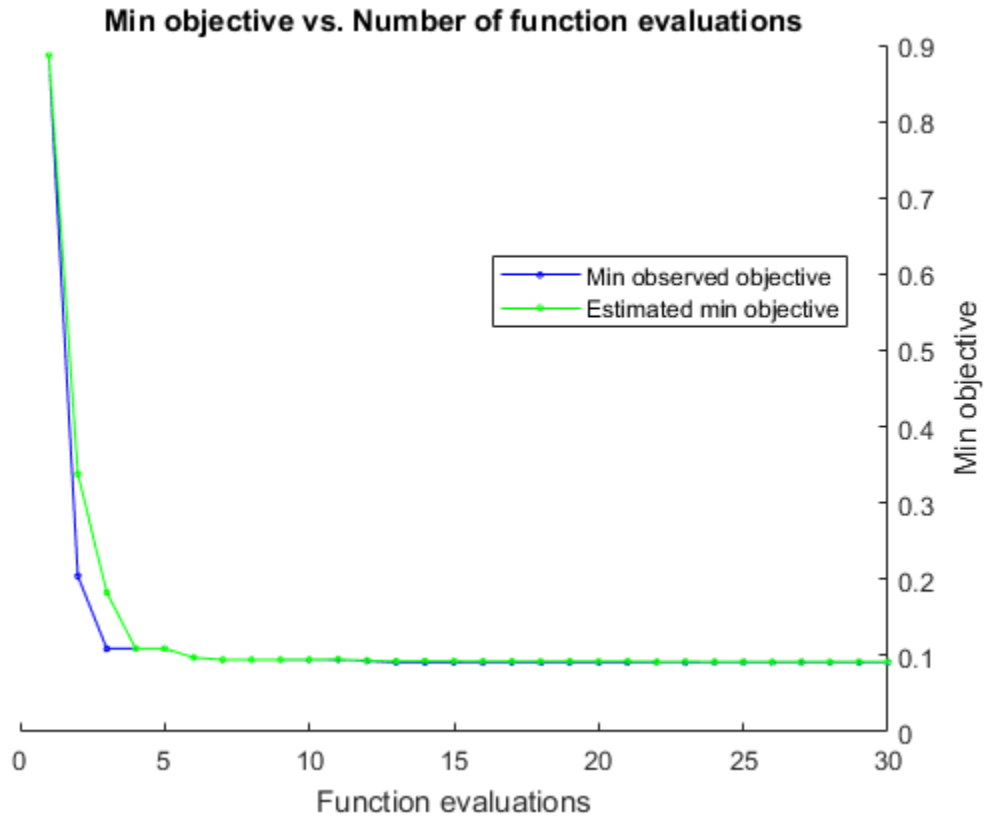
Function evaluation time = 15.5993

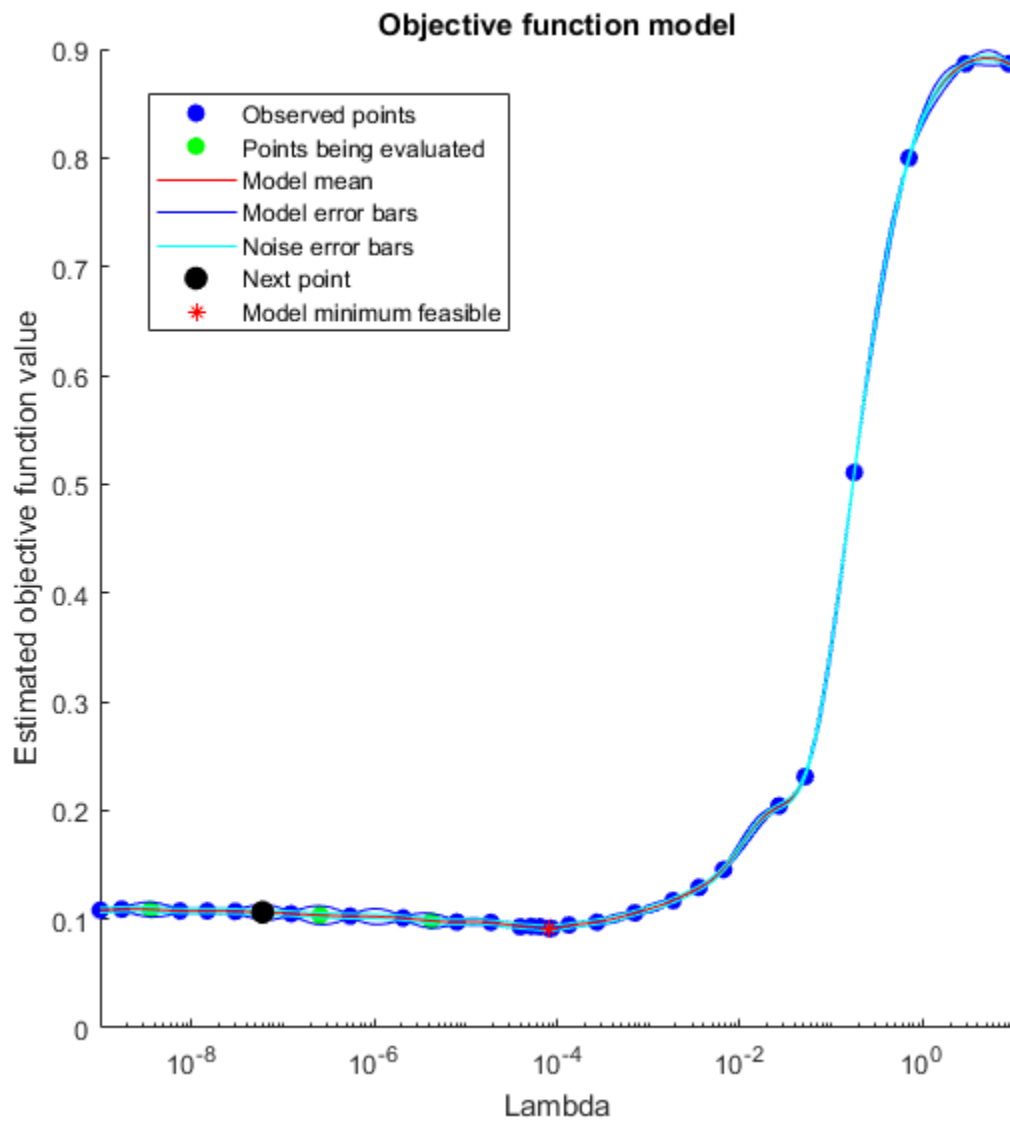
Best estimated feasible point (according to models):

Lambda

-----  
8.2616e-05

Estimated objective function value = 0.091807  
Estimated function evaluation time = 15.6212







## Evaluate Classifier

Check the error of the classifier when applied to test data. First, load the test data.

```
imageFileName = 'train-images.idx3-ubyte';
labelFileName = 'train-labels.idx1-ubyte';
[Xtest,LabelTest] = processMNISTdata(imageFileName,labelFileName);
```

```
Read MNIST image data...
Number of images in the dataset: 60000 ...
Each image is of 28 by 28 pixels...
The image data is read to a matrix of dimensions: 60000 by 784...
End of reading image data.
```

```
Read MNIST label data...
Number of labels in the dataset: 60000 ...
The label data is read to a matrix of dimensions: 60000 by 1...
End of reading label data.
```

Calculate the classification loss when applying the classifier to the test data.

```
TestX = transform(Mdl,Xtest);
Loss = loss(Cmdl,TestX,LabelTest)
```

```
Loss =
    0.1007
```

Did this transformation result in a better classifier than one trained on the original data?  
Create a classifier based on the original training data and evaluate its loss.

```
Omdl = fitcecoc(Xtrain,LabelTrain,'Learners',t,...
    'OptimizeHyperparameters',{'Lambda'},...
    'HyperparameterOptimizationOptions',options);
Losso = loss(Omdl,Xtest,LabelTest)
```

```
Copying objective function to workers...
Done copying objective function to workers.
```

```
|=====|
| Iter | Parallel | Eval   | Objective | Objective | BestSoFar | BestSoFar |
|      | jobs     | result |           | runtime   | (observed) | (estim.)  |
|=====|
```

1	4	Best	0.0779	57.11	0.0779	0.0779	1.32
2	4	Accept	0.0779	57.208	0.0779	0.0779	3.86
3	4	Accept	0.0779	57.159	0.0779	0.0779	1.00
4	4	Accept	0.0787	173.74	0.0779	0.0779	0.
5	4	Accept	0.0779	57.848	0.0779	0.0779	1.35
6	4	Best	0.0776	207.36	0.0776	0.077983	0.0
7	4	Accept	0.0779	57.409	0.0776	0.077971	4.87
8	4	Accept	0.0783	187.99	0.0776	0.078012	0.
9	4	Best	0.0744	312.93	0.0744	0.0744	
10	4	Accept	0.0783	91.396	0.0744	0.0744	4.21
11	4	Accept	0.0769	169.28	0.0744	0.074808	0.
12	4	Accept	0.0767	247.13	0.0744	0.075749	
13	4	Best	0.0716	334.13	0.0716	0.074214	
14	4	Accept	0.0728	332.66	0.0716	0.073604	
15	4	Accept	0.0779	56.869	0.0716	0.073559	2.14
16	4	Accept	0.0719	329.36	0.0716	0.07307	
17	4	Accept	0.078	58.842	0.0716	0.073048	7.92
18	4	Accept	0.073	319.72	0.0716	0.072646	
19	4	Accept	0.0779	58.209	0.0716	0.072618	5.26
20	4	Accept	0.0733	338.53	0.0716	0.07294	
=====							
Iter	Parallel	Eval	Objective	Objective	BestSoFar	BestSoFar	
	jobs	result		runtime	(observed)	(estim.)	
=====							
21	4	Accept	0.0784	166.29	0.0716	0.072774	0.
22	4	Accept	0.0779	57.901	0.0716	0.072757	3.35
23	4	Accept	0.0779	175.52	0.0716	0.072748	0.00
24	4	Accept	0.0779	57.903	0.0716	0.072734	8.64
25	4	Accept	0.0767	192.41	0.0716	0.072785	0.
26	4	Accept	0.0781	67.523	0.0716	0.072774	1.80
27	4	Accept	0.0779	56.024	0.0716	0.072762	1.75
28	4	Accept	0.0782	158.12	0.0716	0.072752	0.
29	4	Accept	0.0779	58.37	0.0716	0.072742	2.57
30	4	Accept	0.0727	327.58	0.0716	0.072726	

Optimization completed.  
 MaxObjectiveEvaluations of 30 reached.  
 Total function evaluations: 30  
 Total elapsed time: 1272.7602 seconds.  
 Total objective function evaluation time: 4822.525

Best observed feasible point:  
 Lambda

—————  
9.1102

Observed objective function value = 0.0716  
Estimated objective function value = 0.072726  
Function evaluation time = 334.1341

Best estimated feasible point (according to models):

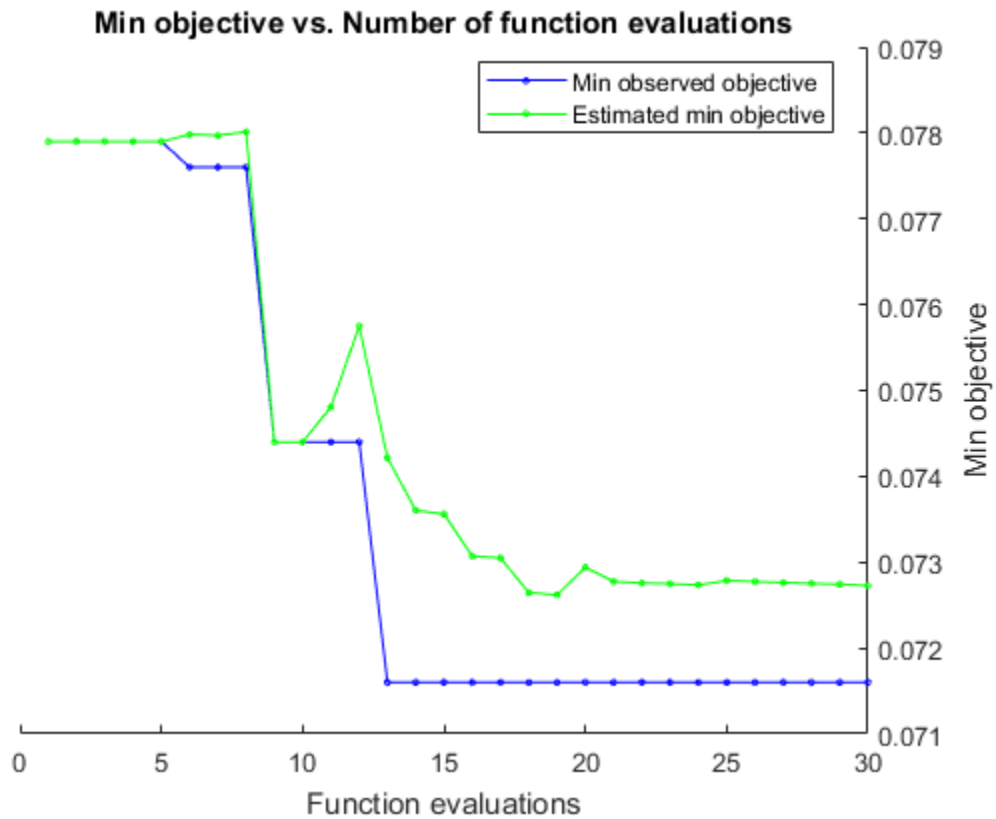
Lambda  
—————

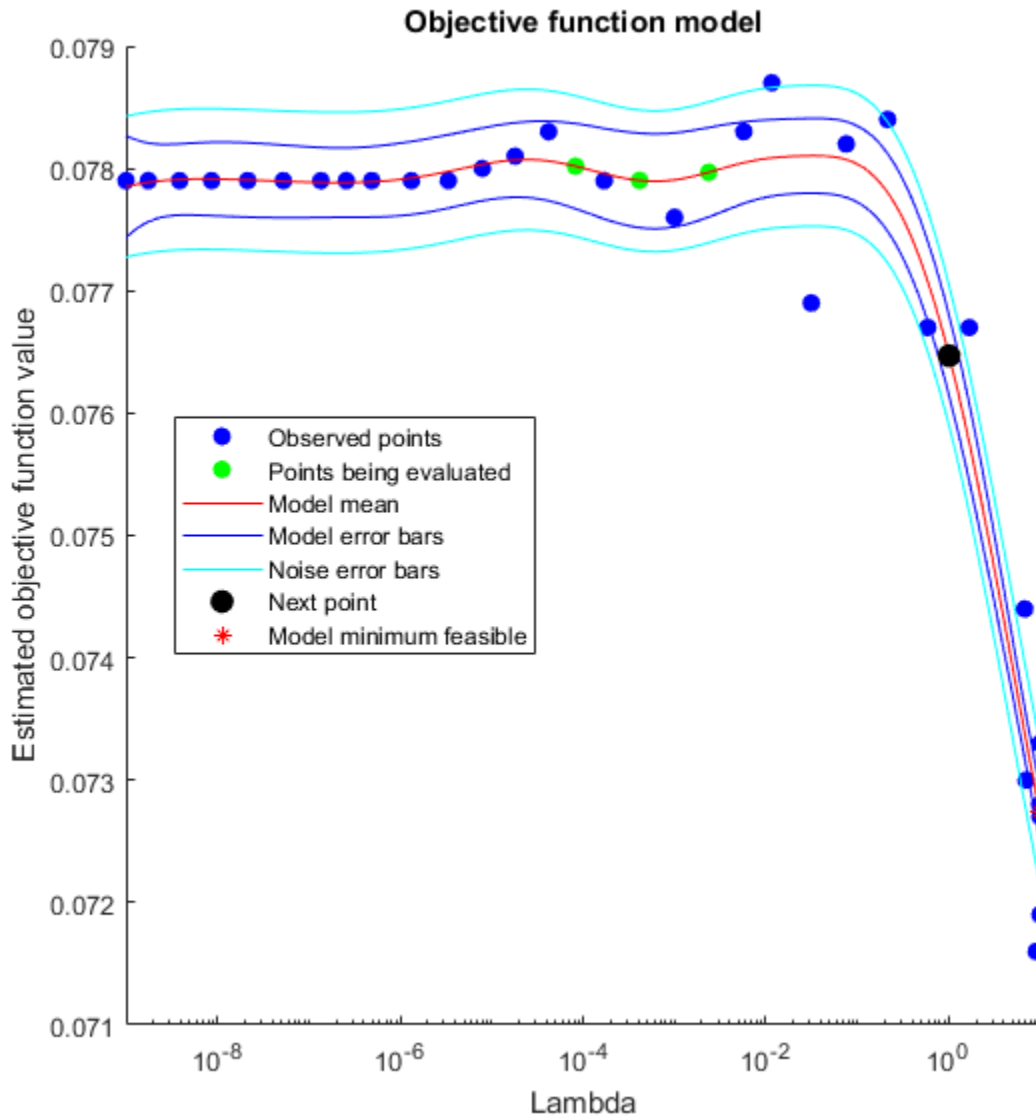
9.9965

Estimated objective function value = 0.072726  
Estimated function evaluation time = 332.7126

Lasso =

0.0859





The classifier based on sparse filtering has a somewhat higher loss than the classifier based on the original data. However, the classifier uses only 100 features rather than the 784 features in the original data, and is much faster to create. Try to make a better sparse filtering classifier by increasing  $q$  from 100 to 200, which is still far less than 784.

```
q = 200;
Mdl2 = sparsefilt(Xtrain,q,'IterationLimit',10);
NewX = transform(Mdl2,Xtrain);
TestX = transform(Mdl2,Xtest);
Cmd1 = fitcecoc(NewX,LabelTrain,'Learners',t,...
    'OptimizeHyperparameters',{'Lambda'},...
    'HyperparameterOptimizationOptions',options);
Loss2 = loss(Cmd1,TestX,LabelTest)
```

```
Warning: Solver LBFGS was not able to converge to a solution.
Copying objective function to workers...
Done copying objective function to workers.
```

Iter	Parallel jobs	Eval result	Objective	Objective runtime	BestSoFar (observed)	BestSoFar (estim.)
1	4	Best	0.2146	10.042	0.2146	0.2146
2	4	Best	0.0615	11.875	0.0615	0.069571
3	4	Best	0.0582	19.778	0.0582	0.058213
4	4	Accept	0.067	21.605	0.0582	0.060443
5	4	Accept	0.0613	11.27	0.0582	0.058198
6	4	Accept	0.0582	20.023	0.0582	0.058055
7	4	Accept	0.0596	12.579	0.0582	0.058056
8	4	Accept	0.0621	11.577	0.0582	0.05805
9	4	Accept	0.8865	9.6478	0.0582	0.058195
10	4	Best	0.0565	28.664	0.0565	0.058005
11	4	Accept	0.0909	16.912	0.0565	0.056423
12	4	Best	0.0553	37.868	0.0553	0.055225
13	4	Accept	0.0564	30.593	0.0553	0.054954
14	4	Accept	0.0581	27.268	0.0553	0.054913
15	4	Accept	0.0556	39.009	0.0553	0.05494
16	4	Accept	0.0556	39.835	0.0553	0.055186
17	4	Best	0.0551	37.67	0.0551	0.055142
18	4	Accept	0.0601	10.875	0.0551	0.055143
19	4	Accept	0.1338	11.441	0.0551	0.055159
20	4	Accept	0.0588	16.072	0.0551	0.05516

```

=====
| 21 | 4 | Accept | 0.8001 | 8.6042 | 0.0551 | 0.055413 | 0.0000 |
| 22 | 4 | Accept | 0.0554 | 35.233 | 0.0551 | 0.055267 | 3.4000 |
| 23 | 4 | Accept | 0.0575 | 22.67 | 0.0551 | 0.055265 | 5.1200 |
| 24 | 4 | Accept | 0.0576 | 34.061 | 0.0551 | 0.055293 | 4.9700 |
| 25 | 4 | Accept | 0.0606 | 27.611 | 0.0551 | 0.055293 | 0.0000 |
| 26 | 4 | Accept | 0.0556 | 34.85 | 0.0551 | 0.055331 | 6.0200 |
| 27 | 4 | Accept | 0.0565 | 38.832 | 0.0551 | 0.055283 | 1.3900 |
| 28 | 4 | Accept | 0.0557 | 35.797 | 0.0551 | 0.05531 | 5.7000 |
| 29 | 4 | Best | 0.0548 | 35.756 | 0.0548 | 0.055255 | 5.1600 |
| 30 | 4 | Accept | 0.5449 | 8.4334 | 0.0548 | 0.055266 | 0.0000 |
=====

```

Optimization completed.

MaxObjectiveEvaluations of 30 reached.

Total function evaluations: 30

Total elapsed time: 199.1591 seconds.

Total objective function evaluation time: 706.4522

Best observed feasible point:

Lambda

-----  
5.1687e-05

Observed objective function value = 0.0548

Estimated objective function value = 0.055266

Function evaluation time = 35.756

Best estimated feasible point (according to models):

Lambda

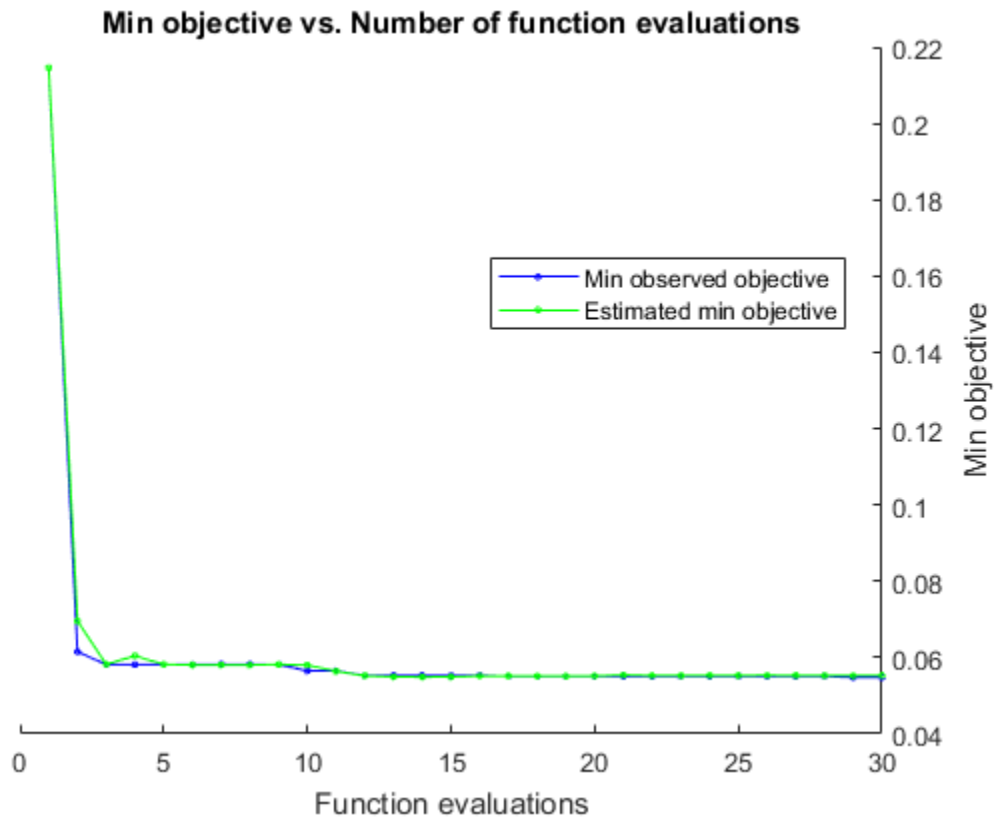
-----  
5.1687e-05

Estimated objective function value = 0.055266

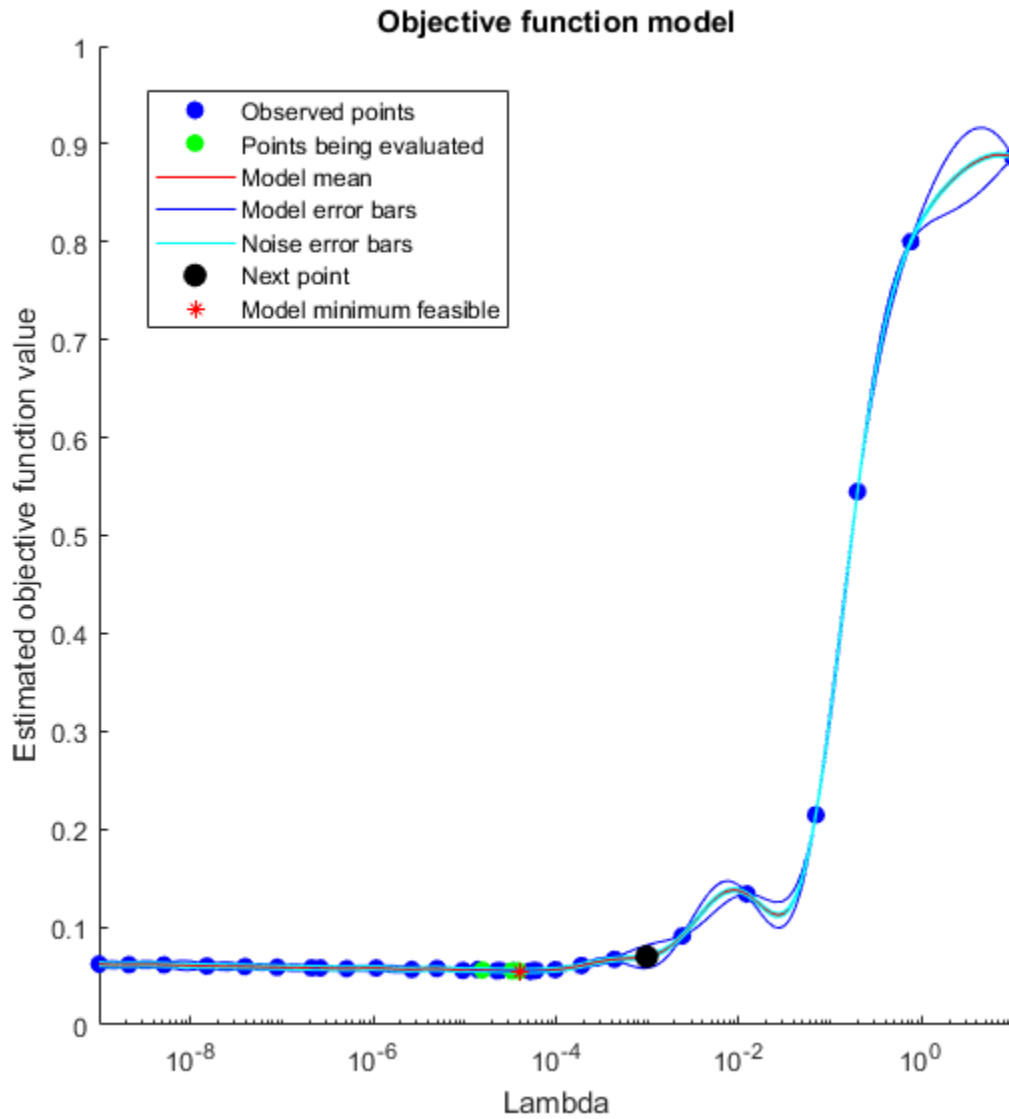
Estimated function evaluation time = 35.4954

Loss2 =

0.0653







This time the classification loss is lower than that of the original data classifier.

## Try RICA

Try the other feature extraction function, `rica`. Extract 200 features, create a classifier, and examine its loss on the test data. Use more iterations for the `rica` function, because `rica` can perform better with more iterations than `sparsefilt` uses.

Often, you "prewhiten" the input data prior to applying `rica`, which makes the preprocessed predictors have zero mean and identity covariance. Preprocess the predictors to make them zero mean and unit variance using the `Standardize` name-value pair. Alternatively, you can transform images for contrast normalization individually by applying the `zscore` transformation before calling `sparsefilt` or `rica`.

```
Mdl3 = rica(Xtrain,q,'IterationLimit',400,'Standardize',true);
NewX = transform(Mdl3,Xtrain);
TestX = transform(Mdl3,Xtest);
Cmdl = fitcecoc(NewX,LabelTrain,'Learners',t,...
    'OptimizeHyperparameters',{'Lambda'},...
    'HyperparameterOptimizationOptions',options);
Loss3 = loss(Cmdl,TestX,LabelTest)
```

```
Warning: Solver LBFGS was not able to converge to a solution.
Copying objective function to workers...
Done copying objective function to workers.
```

Iter	Parallel jobs	Eval result	Objective	Objective runtime	BestSoFar (observed)	BestSoFar (estim.)
1	4	Best	0.11	15.069	0.11	0.11
2	4	Best	0.0828	16.65	0.0828	0.084192
3	4	Best	0.0825	23.347	0.0825	0.082502
4	4	Best	0.0671	31.728	0.0671	0.070823
5	4	Accept	0.0833	15.697	0.0671	0.067103
6	4	Accept	0.0831	15.868	0.0671	0.067103
7	4	Accept	0.0831	23.287	0.0671	0.067103
8	4	Best	0.0625	29.925	0.0625	0.062506
9	4	Accept	0.068	32.448	0.0625	0.062512
10	4	Accept	0.0781	36.021	0.0625	0.062571
11	4	Accept	0.0709	32.144	0.0625	0.062566
12	4	Best	0.062	28.614	0.062	0.06222
13	4	Best	0.0614	29.659	0.0614	0.061643
14	4	Accept	0.0628	30.446	0.0614	0.061584
15	4	Accept	0.0631	28.154	0.0614	0.061513
16	4	Accept	0.082	15.822	0.0614	0.061545
17	4	Accept	0.0615	29.805	0.0614	0.061591

18	4	Best	0.0613	29.221	0.0613	0.061374	0.
19	4	Accept	0.0626	29.106	0.0613	0.061782	0.
20	4	Accept	0.0625	29.502	0.0613	0.061875	0.
=====							
Iter	Parallel	Eval	Objective	Objective	BestSoFar	BestSoFar	
	jobs	result		runtime	(observed)	(estim.)	
=====							
21	4	Accept	0.0776	19.345	0.0613	0.061943	0.
22	4	Accept	0.0807	24.343	0.0613	0.061944	7.12
23	4	Accept	0.0825	20.78	0.0613	0.061943	3.42
24	4	Accept	0.0658	24.693	0.0613	0.06195	0.
25	4	Accept	0.0617	29.622	0.0613	0.061921	0.
26	4	Accept	0.1246	13.382	0.0613	0.061933	
27	4	Accept	0.0613	27.734	0.0613	0.061868	0.
28	4	Accept	0.0613	28.522	0.0613	0.061814	0.
29	4	Accept	0.0813	16.097	0.0613	0.061815	2.22
30	4	Accept	0.0806	23.225	0.0613	0.061817	4.65

Optimization completed.

MaxObjectiveEvaluations of 30 reached.

Total function evaluations: 30

Total elapsed time: 210.9819 seconds.

Total objective function evaluation time: 750.2569

Best observed feasible point:

Lambda

0.064793

Observed objective function value = 0.0613

Estimated objective function value = 0.061817

Function evaluation time = 29.2211

Best estimated feasible point (according to models):

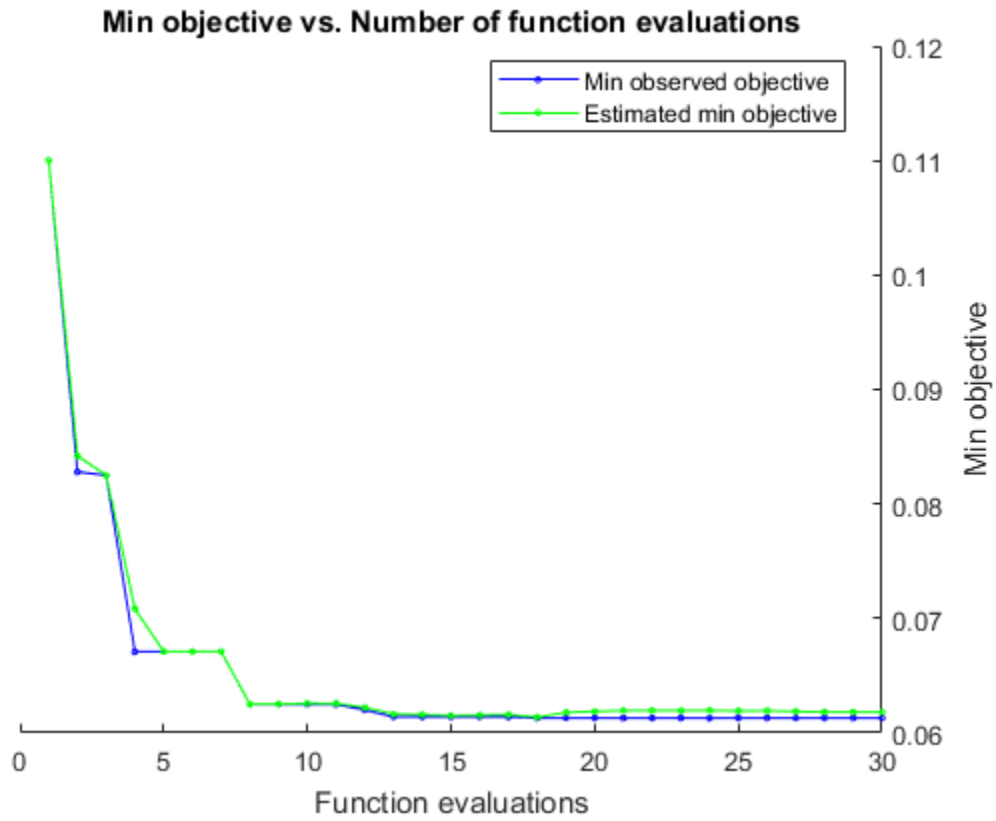
Lambda

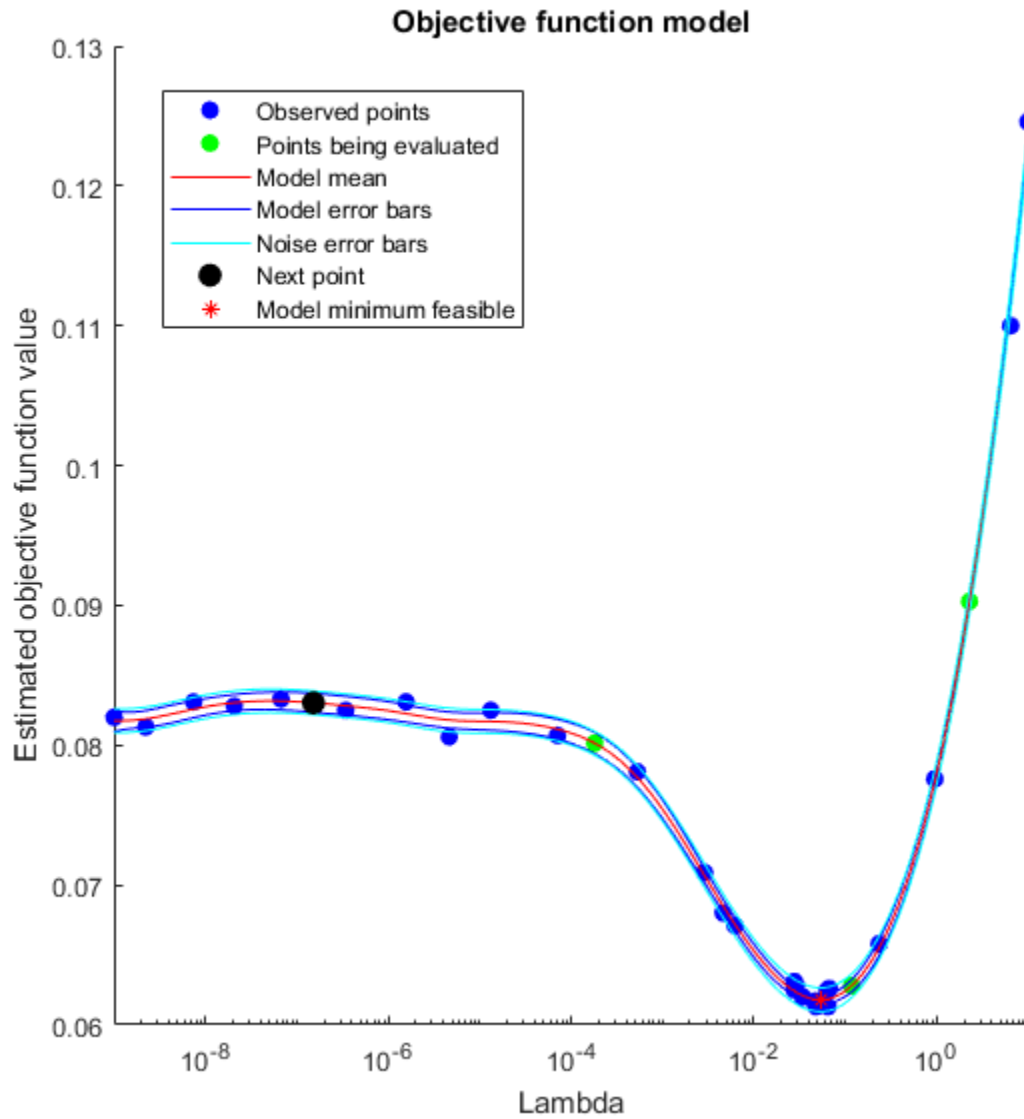
0.048215

Estimated objective function value = 0.061817

Estimated function evaluation time = 29.1709

```
Loss3 =  
0.0745
```





The `rica`-based classifier has somewhat higher test loss compared to the sparse filtering classifier.

### Try More Features

The feature extraction functions have few tuning parameters. One parameter that can affect results is the number of requested features. See how well classifiers work when based on 1000 features, rather than the 200 features previously tried, or the 784 features in the original data. Using more features than appear in the original data is called "overcomplete" learning. Conversely, using fewer features is called "undercomplete" learning. Overcomplete learning can lead to increased classification accuracy, while undercomplete learning can save memory and time.

```
q = 1000;
Mdl4 = sparsefilt(Xtrain,q,'IterationLimit',10);
NewX = transform(Mdl4,Xtrain);
TestX = transform(Mdl4,Xtest);
Cmd1 = fitcecoc(NewX,LabelTrain,'Learners',t,...
    'OptimizeHyperparameters',{'Lambda'},...
    'HyperparameterOptimizationOptions',options);
Loss4 = loss(Cmd1,TestX,LabelTest)
```

Warning: Solver LBFGS was not able to converge to a solution.

Copying objective function to workers...

Done copying objective function to workers.

Iter	Parallel jobs	Eval result	Objective	Objective runtime	BestSoFar (observed)	BestSoFar (estim.)
1	4	Best	0.8865	64.379	0.8865	0.8865
2	4	Best	0.0417	65.812	0.0417	0.08212
3	4	Accept	0.1195	103.19	0.0417	0.10094
4	4	Accept	0.0419	65.801	0.0417	0.041722
5	4	Best	0.0409	65.937	0.0409	0.040862
6	4	Accept	0.1474	88.664	0.0409	0.040952
7	4	Best	0.0403	72.769	0.0403	0.04034
8	4	Best	0.0368	313.66	0.0368	0.036834
9	4	Accept	0.0388	182.27	0.0368	0.036847
10	4	Accept	0.039	83.211	0.0368	0.036869
11	4	Accept	0.0461	232.6	0.0368	0.036823
12	4	Accept	0.0377	249.8	0.0368	0.036841
13	4	Accept	0.0368	350.11	0.0368	0.036879
14	4	Best	0.0364	398	0.0364	0.036693
15	4	Best	0.0363	398.64	0.0363	0.036343

16	4	Accept	0.0406	67.555	0.0363	0.036342	5.39
17	4	Accept	0.7228	52.853	0.0363	0.036179	0
18	4	Accept	0.037	379.28	0.0363	0.03651	1.19
19	4	Accept	0.0366	362.87	0.0363	0.036754	9.47
20	4	Accept	0.0396	124.02	0.0363	0.036755	1.57
=====							
Iter	Parallel	Eval	Objective	Objective	BestSoFar	BestSoFar	
	jobs	result		runtime	(observed)	(estim.)	
=====							
21	4	Accept	0.0363	381.82	0.0363	0.036297	3.04
22	4	Accept	0.0388	310.98	0.0363	0.036355	0.00
23	4	Accept	0.0408	68.59	0.0363	0.036355	2.82
24	4	Accept	0.0382	213.56	0.0363	0.036356	7.27
25	4	Accept	0.0369	282.89	0.0363	0.036355	2.79
26	4	Accept	0.0371	360.5	0.0363	0.036341	6.23
27	4	Accept	0.0365	381.1	0.0363	0.036337	4.14
28	4	Best	0.0362	387.77	0.0362	0.036293	1.80
29	4	Accept	0.0365	303.78	0.0362	0.036291	3.31
30	4	Best	0.0361	388.37	0.0361	0.036245	3.79

---

Optimization completed.  
 MaxObjectiveEvaluations of 30 reached.  
 Total function evaluations: 30  
 Total elapsed time: 1828.8744 seconds.  
 Total objective function evaluation time: 6800.7948

Best observed feasible point:  
 Lambda

---

3.7922e-05

Observed objective function value = 0.0361  
 Estimated objective function value = 0.036245  
 Function evaluation time = 388.3677

Best estimated feasible point (according to models):  
 Lambda

---

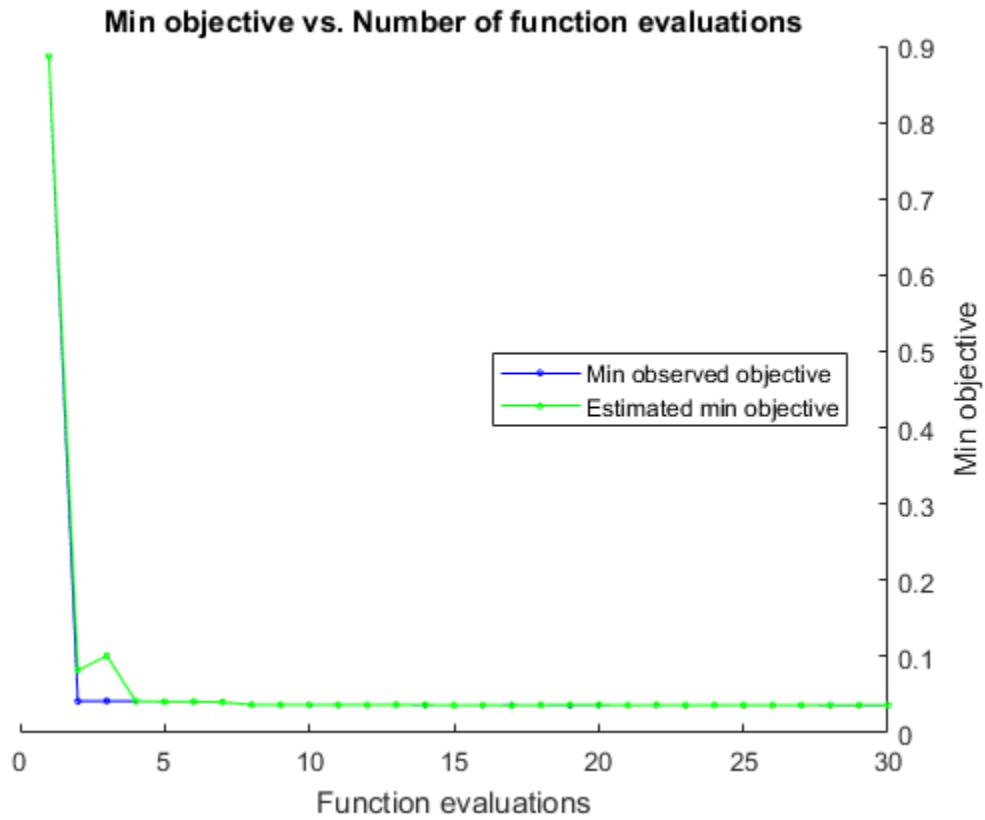
3.0466e-05

Estimated objective function value = 0.036245

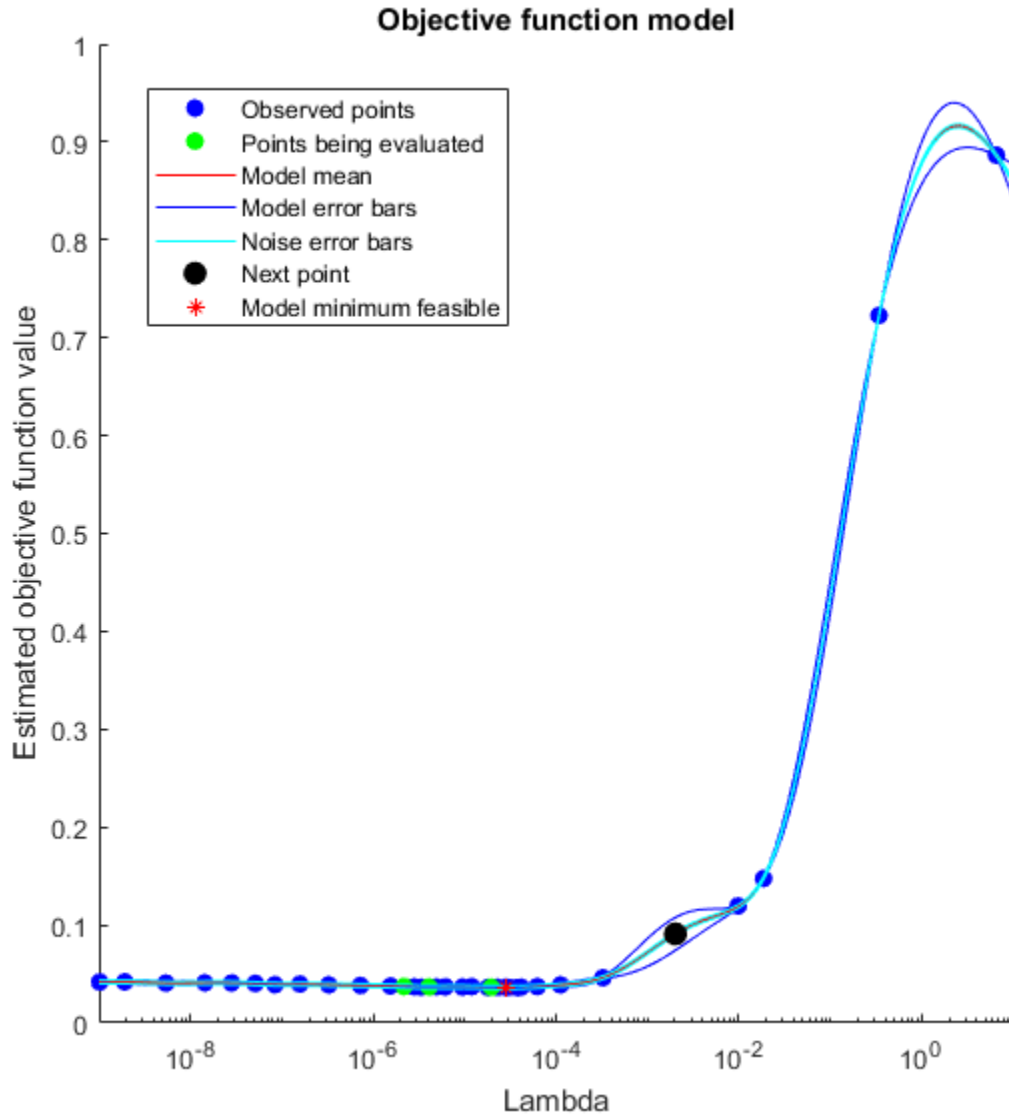
Estimated function evaluation time = 391.4562

Loss4 =

0.0438







The classifier based on overcomplete sparse filtering with 1000 extracted features has the lowest test loss of any classifier yet tested.

```
Mdl5 = rica(Xtrain,q, 'IterationLimit',400, 'Standardize',true);
NewX = transform(Mdl5,Xtrain);
TestX = transform(Mdl5,Xtest);
Cmdl = fitcecoc(NewX,LabelTrain, 'Learners',t,...
    'OptimizeHyperparameters',{'Lambda'},...
    'HyperparameterOptimizationOptions',options);
Loss5 = loss(Cmdl,TestX,LabelTest)
```

Warning: Solver LBFGS was not able to converge to a solution.

Copying objective function to workers...

Done copying objective function to workers.

Iter	Parallel jobs	Eval result	Objective	Objective runtime	BestSoFar (observed)	BestSoFar (estim.)	
1	4	Best	0.0802	55.922	0.0802	0.0802	1.44
2	4	Accept	0.1005	139.27	0.0802	0.081235	
3	4	Best	0.0801	56.1	0.0801	0.080105	4.50
4	4	Best	0.0788	199.43	0.0788	0.078803	3.1
5	4	Accept	0.0799	55.68	0.0788	0.078804	1.33
6	4	Best	0.0664	299.06	0.0664	0.066402	0.
7	4	Accept	0.0682	253.72	0.0664	0.066401	0.
8	4	Accept	0.0793	201.26	0.0664	0.066401	6.06
9	4	Accept	0.0788	215.51	0.0664	0.0664	1.37
10	4	Accept	0.0795	223.56	0.0639	0.063896	8.59
11	4	Best	0.0639	316.29	0.0639	0.063896	0.
12	4	Accept	0.07	396.93	0.0639	0.063898	0.0
13	4	Accept	0.0682	345.56	0.0639	0.063895	0.0
14	4	Accept	0.0641	317.25	0.0639	0.063926	0.
15	4	Accept	0.0649	326.07	0.0639	0.064254	0.
16	4	Best	0.0637	312.77	0.0637	0.064088	0.
17	4	Accept	0.0795	56.141	0.0637	0.064094	6.55
18	4	Accept	0.0649	322.81	0.0637	0.064237	0.
19	4	Accept	0.1258	117	0.0637	0.064304	
20	4	Accept	0.0644	309.13	0.0637	0.064302	0.
Iter	Parallel jobs	Eval result	Objective	Objective runtime	BestSoFar (observed)	BestSoFar (estim.)	
21	4	Accept	0.0646	317.53	0.0637	0.064343	0.
22	4	Accept	0.0803	55.455	0.0637	0.064343	1.00
23	4	Accept	0.0799	187.16	0.0637	0.064343	1.8

24	4	Accept	0.077	208.49	0.0637	0.064328	0
25	4	Accept	0.08	57.795	0.0637	0.064328	1.76
26	4	Accept	0.0779	359.13	0.0637	0.064324	0.0
27	4	Accept	0.0652	286.35	0.0637	0.064317	0
28	4	Accept	0.0642	312.19	0.0637	0.064295	0
29	4	Accept	0.08	56.205	0.0637	0.064295	2.96
30	4	Accept	0.0796	187.78	0.0637	0.064294	8.06

---

Optimization completed.

MaxObjectiveEvaluations of 30 reached.

Total function evaluations: 30

Total elapsed time: 1820.3114 seconds.

Total objective function evaluation time: 6547.5583

Best observed feasible point:

Lambda

\_\_\_\_\_

0.10092

Observed objective function value = 0.0637

Estimated objective function value = 0.064294

Function evaluation time = 312.7696

Best estimated feasible point (according to models):

Lambda

\_\_\_\_\_

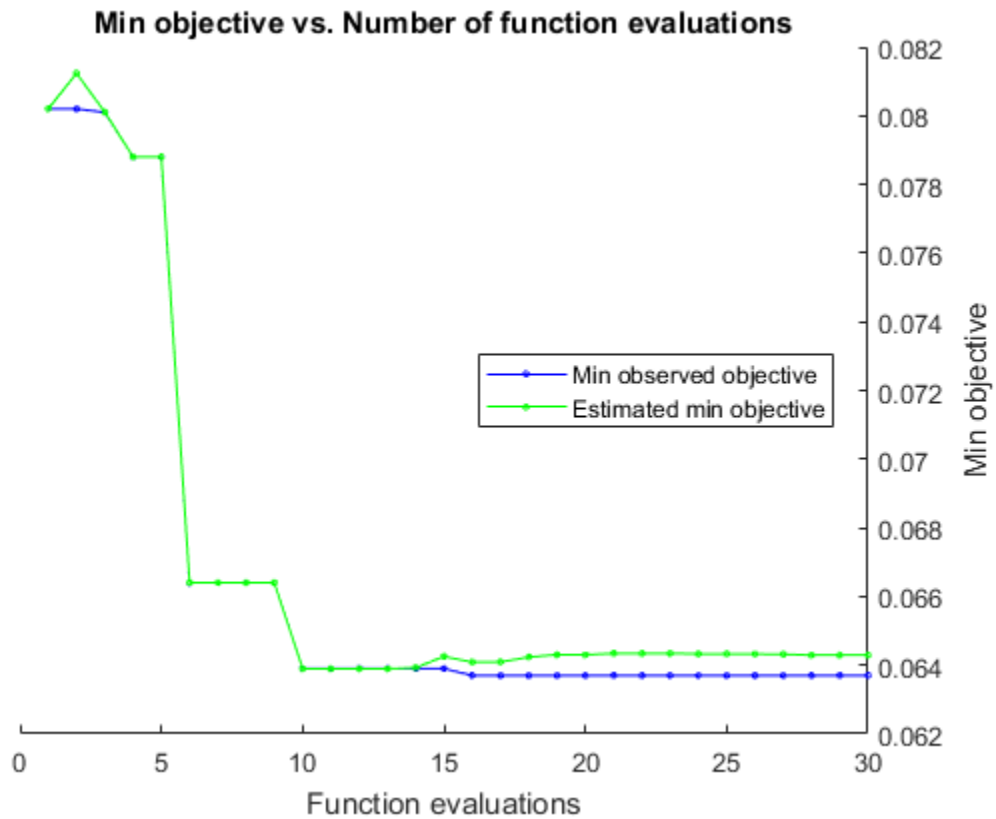
0.10092

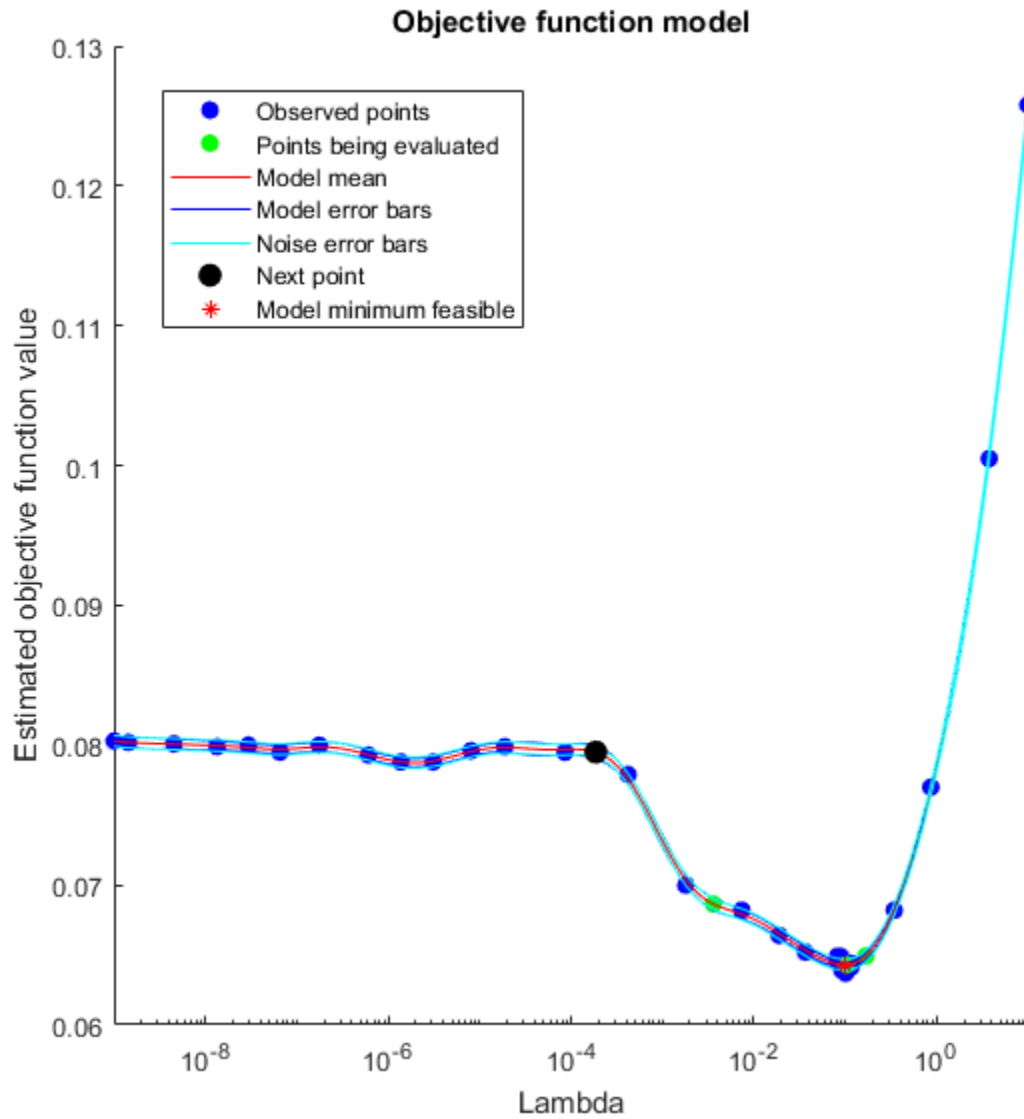
Estimated objective function value = 0.064294

Estimated function evaluation time = 316.708

Loss5 =

0.0746





The classifier based on RICA with 1000 extracted features has a similar test loss to the RICA classifier based on 200 extracted features.

### Optimize Hyperparameters by Using `bayesopt`

Feature extraction functions have these tuning parameters:

- Iteration limit
- Function, either `rica` or `sparsefilt`
- Parameter `Lambda`
- Number of learned features `q`

The `fitcecoc` regularization parameter also affects the accuracy of the learned classifier. Include that parameter in the list of hyperparameters as well.

To search among the available parameters effectively, try `bayesopt`. Use the following objective function, which includes parameters passed from the workspace.

```
function objective = filterica(x,Xtrain,Xtest,LabelTrain,LabelTest,winit)

initW = winit(1:size(Xtrain,2),1:x.q);
if char(x.solver) == 'r'
    Mdl = rica(Xtrain,x.q,'Lambda',x.lambda,'IterationLimit',x.iterlim, ...
              'InitialTransformWeights',initW,'Standardize',true);
else
    Mdl = sparsefilt(Xtrain,x.q,'Lambda',x.lambda,'IterationLimit',x.iterlim, ...
                    'InitialTransformWeights',initW);
end

NewX = transform(Mdl,Xtrain);
TestX = transform(Mdl,Xtest);
t = templateLinear('Lambda',x.lambdareg,'Solver','lbfgs');
Cmdl = fitcecoc(NewX,LabelTrain,'Learners',t);
objective = loss(Cmdl,TestX,LabelTest);
```

To remove sources of variation, fix an initial transform weight matrix.

```
W = randn(1e4,1e3);
```

Create hyperparameters for the objective function.

```

iterlim = optimizableVariable('iterlim',[5,500],'Type','integer');
lambda = optimizableVariable('lambda',[0,10]);
solver = optimizableVariable('solver',{'r','s'},'Type','categorical');
qvar = optimizableVariable('q',[10,1000],'Type','integer');
lambdareg = optimizableVariable('lambdareg',[1e-6,1],'Transform','log');
vars = [iterlim,lambda,solver,qvar,lambdareg];

```

Run the optimization without the warnings that occur when the internal optimizations do not run to completion. Run for 60 iterations instead of the default 30 to give the optimization a better chance of locating a good value.

```

warning('off','stats:classreg:learning:fsutils:Solver:LBFSGUnableToConverge');
results = bayesopt(@(x) filterica(x,Xtrain,Xtest,LabelTrain,LabelTest,W),vars, ...
    'UseParallel',true,'MaxObjectiveEvaluations',60);
warning('on','stats:classreg:learning:fsutils:Solver:LBFSGUnableToConverge');

```

Copying objective function to workers...

Done copying objective function to workers.

```

=====
| Iter | Parallel | Eval | Objective | Objective | BestSoFar | BestSoFar |
|      | jobs    | result |           | runtime   | (observed) | (estim.) |
=====
|  1  |    4    | Best  | 0.1166    | 25.711   | 0.1166    | 0.1166    |
|  2  |    4    | Accept| 0.71605   | 76.241   | 0.1166    | 0.168     |
|  3  |    4    | Best  | 0.087949  | 91.947   | 0.087949  | 0.089846  |
|  4  |    4    | Accept| 0.089246  | 181.63   | 0.087949  | 0.08788   |
|  5  |    4    | Best  | 0.075686  | 186.94   | 0.075686  | 0.088336  |
|  6  |    4    | Accept| 0.083408  | 87.642   | 0.075686  | 0.087038  |
|  7  |    4    | Accept| 0.085505  | 121.9    | 0.075686  | 0.083637  |
|  8  |    4    | Accept| 0.089916  | 435.74   | 0.075686  | 0.08727   |
|  9  |    4    | Accept| 0.081326  | 737.08   | 0.075686  | 0.083747  |
| 10  |    4    | Accept| 0.092619  | 411.11   | 0.075686  | 0.083186  |
| 11  |    4    | Accept| 0.12474   | 224.74   | 0.075686  | 0.081713  |
| 12  |    4    | Accept| 0.086312  | 104.82   | 0.075686  | 0.081801  |
| 13  |    4    | Accept| 0.085033  | 119.54   | 0.075686  | 0.082712  |
| 14  |    4    | Accept| 0.077026  | 415.92   | 0.075686  | 0.081297  |
| 15  |    4    | Accept| 0.20179   | 138.29   | 0.075686  | 0.078728  |
| 16  |    4    | Accept| 0.089593  | 1687.4   | 0.075686  | 0.079257  |
| 17  |    4    | Accept| 0.086069  | 1355.9   | 0.075686  | 0.082026  |
| 18  |    4    | Accept| 0.094693  | 13.743   | 0.075686  | 0.079135  |
| 19  |    4    | Accept| 0.15955   | 755.64   | 0.075686  | 0.082258  |
| 20  |    4    | Accept| 0.45329   | 5.8813   | 0.075686  | 0.076457  |
=====
| Iter | Parallel | Eval | Objective | Objective | BestSoFar | BestSoFar |
|      | jobs    | result |           | runtime   | (observed) | (estim.) |
=====

```

21	4	Accept	0.1278	6.8923	0.075686	0.074967
22	4	Accept	0.29678	19.171	0.075686	0.075986
23	4	Accept	0.096232	17.294	0.075686	0.072978
24	4	Accept	0.11569	8.09	0.075686	0.071506
25	4	Accept	0.17124	29.686	0.075686	0.076131
26	4	Accept	0.11884	8.0113	0.075686	0.075665
27	4	Accept	0.092672	59.528	0.075686	0.075077
28	4	Accept	0.15897	11.83	0.075686	0.070814
29	4	Accept	0.14063	6.5558	0.075686	0.069568
30	4	Accept	0.095535	72.362	0.075686	0.068216
31	4	Accept	0.082814	1406	0.075686	0.069761
32	4	Accept	0.078977	12.791	0.075686	0.069219
33	4	Accept	0.14244	6.2437	0.075686	0.069104
34	4	Accept	0.25919	5.1841	0.075686	0.068823
35	4	Accept	0.082781	115.91	0.075686	0.071099
36	4	Accept	0.080993	138.05	0.075686	0.077505
37	4	Accept	0.18643	7.7357	0.075686	0.07372
38	4	Accept	0.096094	142.51	0.075686	0.073644
39	4	Accept	0.091199	10.004	0.075686	0.07333
40	4	Best	0.074537	403.37	0.074537	0.068331
Iter	Parallel jobs	Eval result	Objective	Objective runtime	BestSoFar (observed)	BestSoFar (estim.)
41	4	Accept	0.087459	314.12	0.074537	0.068088
42	4	Accept	0.089712	113.91	0.074537	0.068136
43	4	Accept	0.1144	25.419	0.074537	0.073893
44	4	Accept	0.083762	99.778	0.074537	0.079615
45	4	Accept	0.088283	25.546	0.074537	0.079767
46	4	Accept	0.13867	25.672	0.074537	0.079374
47	4	Accept	0.10812	1196.2	0.074537	0.076877
48	4	Accept	0.090668	64.519	0.074537	0.076929
49	4	Accept	0.080217	151.22	0.074537	0.076138
50	4	Best	0.066878	118.07	0.066878	0.075555
51	4	Best	0.050424	110.41	0.050424	0.055395
52	4	Accept	0.078742	103.03	0.050424	0.056657
53	4	Accept	0.054667	129.35	0.050424	0.051813
54	4	Accept	0.069942	253.63	0.050424	0.057901
55	4	Accept	0.087118	235.77	0.050424	0.057902
56	4	Best	0.046525	81.362	0.046525	0.05457
57	4	Accept	0.067507	303.24	0.046525	0.056027
58	4	Accept	0.089009	1696	0.046525	0.057513
59	4	Accept	0.12258	1218.6	0.046525	0.052949



---

| 60 | 4 | Accept | 0.053558 | 87.08 | 0.046525 | 0.052416 |

---

Optimization completed.

MaxObjectiveEvaluations of 60 reached.

Total function evaluations: 60

Total elapsed time: 4526.2828 seconds.

Total objective function evaluation time: 16017.8491

Best observed feasible point:

iterlim	lambda	solver	q	lambdareg
7	0.15531	s	913	5.0013e-05

Observed objective function value = 0.046525

Estimated objective function value = 0.052416

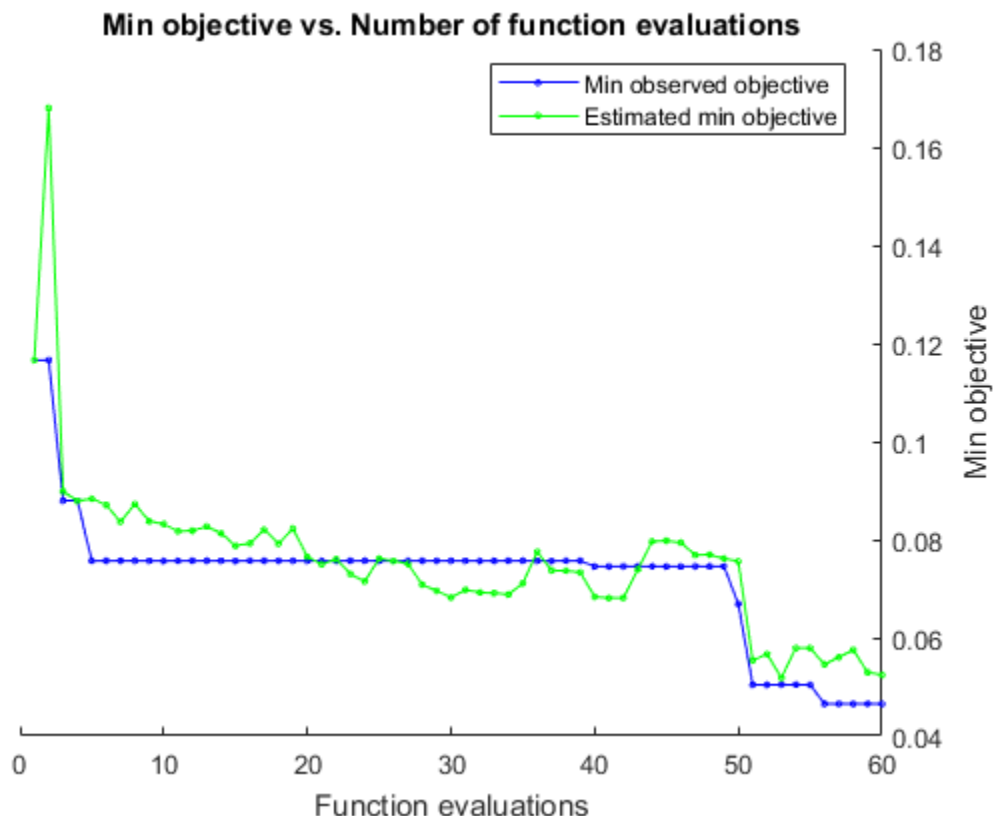
Function evaluation time = 81.3616

Best estimated feasible point (according to models):

iterlim	lambda	solver	q	lambdareg
7	0.15531	s	913	5.0013e-05

Estimated objective function value = 0.052416

Estimated function evaluation time = 91.2652



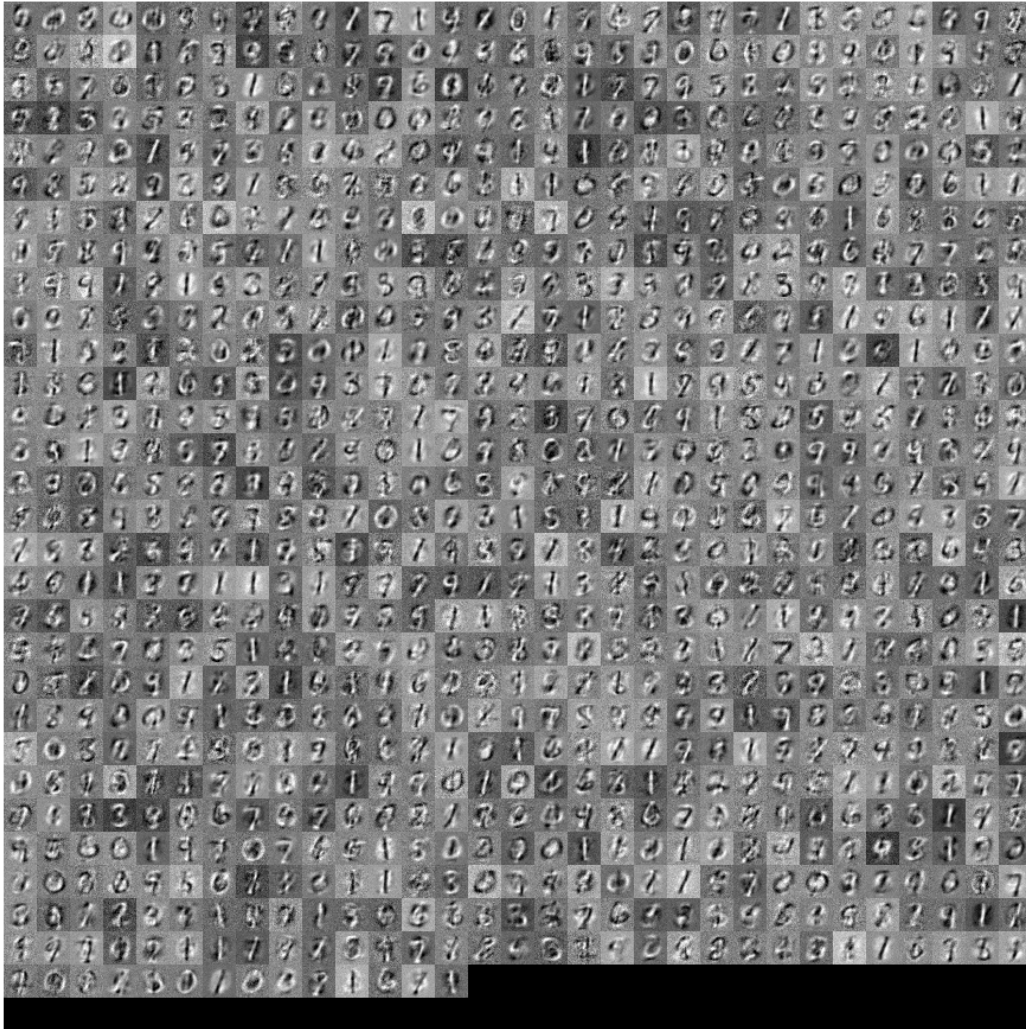
The resulting classifier does not have better (lower) loss than the classifier using `sparsefilt` for 1000 features, trained for 10 iterations.

View the filter coefficients for the best hyperparameters that `bayesopt` found. The resulting images show the shapes of the extracted features. These shapes are recognizable as portions of handwritten digits.

```
Xtbl = results.XAtMinObjective;
Q = Xtbl.q;
initW = W(1:size(Xtrain,2),1:Q);
if char(Xtbl.solver) == 'r'
    mdl = rica(Xtrain,Q,'Lambda',Xtbl.lambda,'IterationLimit',Xtbl.iterlim, ...
              'InitialTransformWeights',initW,'Standardize',true);
else
```

```
Mdl = sparsefilt(Xtrain,Q,'Lambda',Xtbl.lambda,'IterationLimit',Xtbl.iterlim, ...
    'InitialTransformWeights',initW);
end
Wts = Mdl.TransformWeights;
Wts = reshape(Wts,[28,28,Q]);
[dx,dy,~,~] = size(Wts);
for f = 1:Q
    Wvec = Wts(:, :, f);
    Wvec = Wvec(:);
    Wvec = (Wvec - min(Wvec))/(max(Wvec) - min(Wvec));
    Wts(:, :, f) = reshape(Wvec,dx,dy);
end
m = ceil(sqrt(Q));
n = m;
img = zeros(m*dx,n*dy);
f = 1;
for i = 1:m
    for j = 1:n
        if (f <= Q)
            img((i-1)*dx+1:i*dx,(j-1)*dy+1:j*dy,:) = Wts(:, :, f);
            f = f+1;
        end
    end
end
end
imshow(img);
```

Warning: Solver LBFGS was not able to converge to a solution.



### Code for Reading MNIST Data

The code of the function that reads the data into the workspace is:

```
function [X,L] = processMNISTdata(imageFileName,labelFileName)

[fileID,errmsg] = fopen(imageFileName,'r','b');
if fileID < 0
    error(errmsg);
end
%%
% First read the magic number. This number is 2051 for image data, and
% 2049 for label data
magicNum = fread(fileID,1,'int32',0,'b');
if magicNum == 2051
    fprintf('\nRead MNIST image data...\n')
end
%%
% Then read the number of images, number of rows, and number of columns
numImages = fread(fileID,1,'int32',0,'b');
fprintf('Number of images in the dataset: %6d ...\n',numImages);
numRows = fread(fileID,1,'int32',0,'b');
numCols = fread(fileID,1,'int32',0,'b');
fprintf('Each image is of %2d by %2d pixels...\n',numRows,numCols);
%%
% Read the image data
X = fread(fileID,inf,'unsigned char');
%%
% Reshape the data to array X
X = reshape(X,numCols,numRows,numImages);
X = permute(X,[2 1 3]);
%%
% Then flatten each image data into a 1 by (numRows*numCols) vector, and
% store all the image data into a numImages by (numRows*numCols) array.
X = reshape(X,numRows*numCols,numImages)';
fprintf(['The image data is read to a matrix of dimensions: %6d by %4d...\n',...
        'End of reading image data.\n'],size(X,1),size(X,2));
%%
% Close the file
fclose(fileID);
%%
% Similarly, read the label data.
[fileID,errmsg] = fopen(labelFileName,'r','b');
if fileID < 0
    error(errmsg);
end
magicNum = fread(fileID,1,'int32',0,'b');
```

```
if magicNum == 2049
    fprintf('\nRead MNIST label data...\n')
end
numItems = fread(fileID,1,'int32',0,'b');
fprintf('Number of labels in the dataset: %6d ...\n',numItems);

L = fread(fileID,inf,'unsigned char');
fprintf(['The label data is read to a matrix of dimensions: %6d by %2d...\n',...
        'End of reading label data.\n'],size(L,1),size(L,2));
fclose(fileID);
```

### See Also

ReconstructionICA | SparseFiltering | rica | sparsefilt

### Related Examples

- “Extract Mixed Signals” on page 15-205

### More About

- “Feature Extraction” on page 15-163

## Extract Mixed Signals

This example shows how to use `rica` to disentangle mixed audio signals. You can use `rica` to perform independent component analysis (ICA) when prewhitening is included as a preprocessing step. The ICA model is

$$x = \mu + As.$$

Here,  $x$  is a  $P$ -by-1 vector of mixed signals,  $\mu$  is a  $P$ -by-1 vector of offset values,  $A$  is a  $P$ -by- $Q$  mixing matrix, and  $s$  is a  $Q$ -by-1 vector of original signals. Suppose first that  $A$  is a square matrix. If you know  $\mu$  and  $A$ , you can recover an original signal  $s$  from the data  $x$ :

$$s = A^{-1}(x - \mu).$$

Using the `rica` function, you can perform this recovery even without knowing the mixing matrix  $A$  or the mean  $\mu$ . Given a set of several observations  $x(1)$ ,  $x(2)$ , ..., `rica` extracts the original signals  $s(1)$ ,  $s(2)$ , ....

### Load Data

Load a set of six audio files, which ship with MATLAB®. Trim each file to 10,000 samples.

```
files = {'chirp.mat'
        'gong.mat'
        'handel.mat'
        'laughter.mat'
        'splat.mat'
        'train.mat'};

S = zeros(10000,6);
for i = 1:6
    test    = load(files{i});
    y       = test.y(1:10000,1);
    S(:,i)  = y;
end
```

### Mix Signals

Mix the signals together by using a random mixing matrix and add a random offset.

```
rng default % For reproducibility
mixdata = S*randn(6) + randn(1,6);
```

To listen to the original sounds, execute this code:

```
for i = 1:6
    disp(i);
    sound(S(:,i));
    pause;
end
```

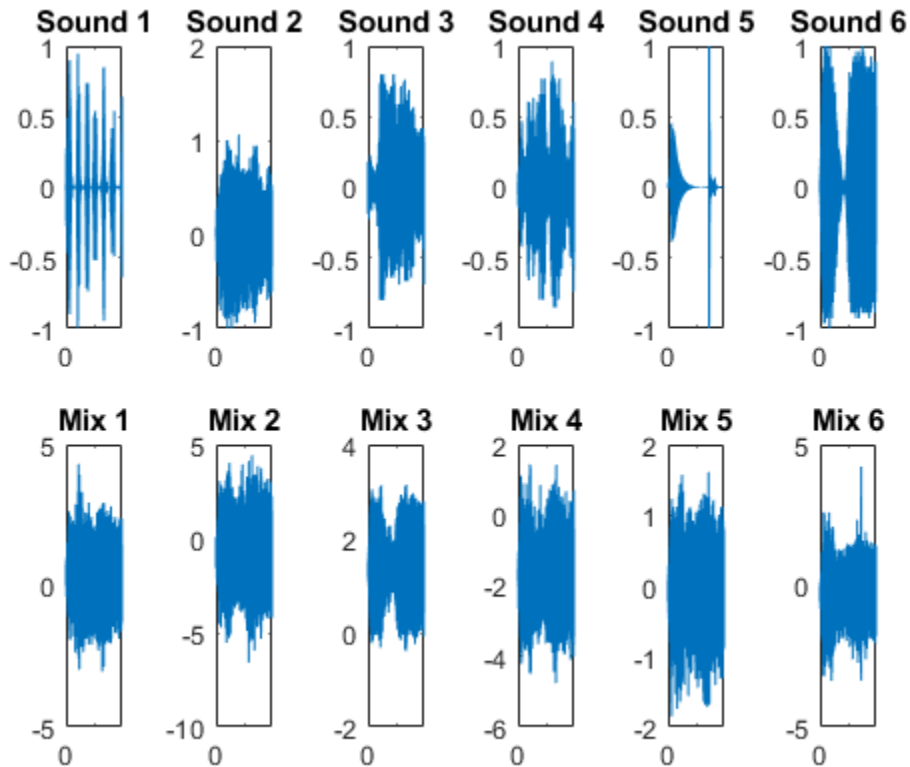
To listen to the mixed sounds, execute this code:

```
for i = 1:6
    disp(i);
    sound(mixdata(:,i));
    pause;
end
```

Plot the signals.

```
figure
for i = 1:6
    subplot(2,6,i)
    plot(S(:,i))
    title(['Sound ', num2str(i)])
    subplot(2,6,i+6)
    plot(mixdata(:,i))
    title(['Mix ', num2str(i)])
end
```





The original signals have clear structure. The mixed signals have much less structure.

### Prewhiten Mixed Signals

To separate the signals effectively, "prewhiten" the signals by using the `prewhiten` function that appears at the end of this example. This function transforms `mixdata` so that it has zero mean and identity covariance.

The idea is the following. If  $s$  is a zero-mean source with statistically independent components, then

$$E(s) = 0$$

$$E(ss^T) = I.$$

Then the mean and covariance of  $x$  are

$$E(x) = \mu$$

$$\text{Cov}(x) = AA^T = C.$$

Suppose that you know  $\mu$  and  $C$ . In practice, you would estimate these quantities from the sample mean and covariance of the columns of  $x$ . You can solve for  $s$  in terms of  $x$  by

$$s = A^{-1}(x - \mu) = (A^T A)^{-1} A^T (x - \mu).$$

The latter equation holds even when  $A$  is not a square invertible matrix.

Suppose that  $U$  is a  $p$ -by- $q$  matrix of left eigenvectors of the positive semidefinite matrix  $C$ , and  $\Sigma$  is the  $q$ -by- $q$  matrix of eigenvalues. Then

$$C = U\Sigma U^T$$

$$U^T U = I.$$

Then

$$AA^T = U\Sigma U^T.$$

There are many mixing matrices  $A$  that satisfy this last equation. If  $W$  is a  $q$ -by- $q$  orthonormal matrix, then

$$W^T W = W W^T = I$$

$$A = U\Sigma^{1/2}W.$$

Substituting into the equation for  $s$ ,

$$s = W^T \tilde{x}, \text{ where}$$

$$\tilde{x} = \Sigma^{-1/2} U^T (x - \mu).$$

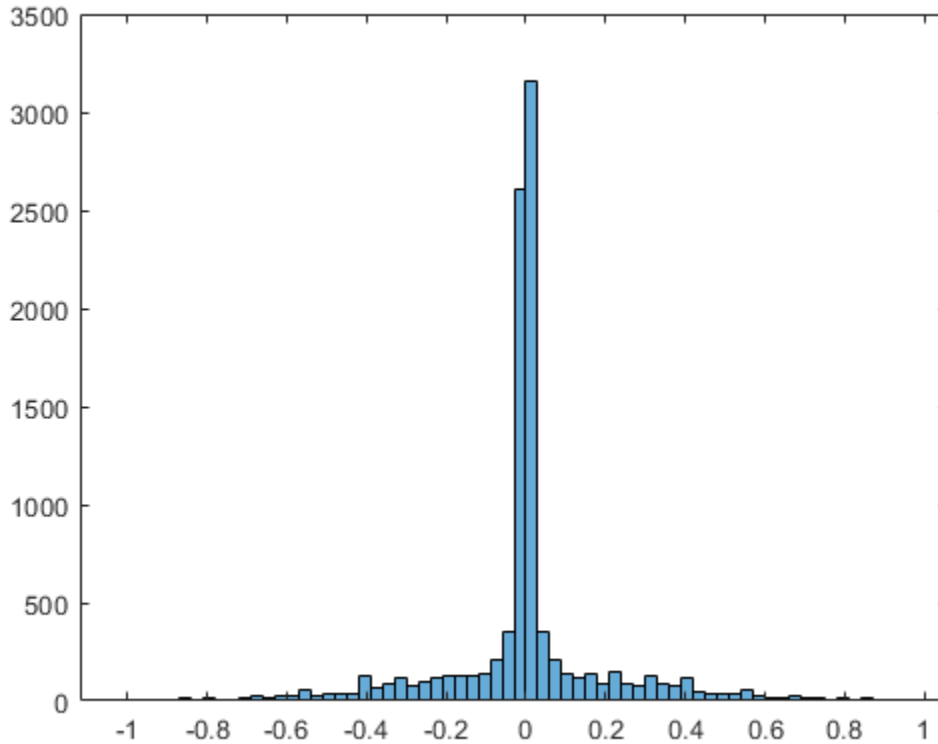
$\tilde{x}$  is the prewhitened data. `rica` computes the unknown matrix  $W$  under the assumption that the components of  $s$  are as independent as possible.

```
mixdata = prewhiten(mixdata);
```

### Separate All Signals

A super-Gaussian source has a sharp peak near zero, such as a histogram of sound 1 shows.

```
figure  
histogram(S(:,1))
```



Perform Reconstruction ICA while asking for six features. Indicate that each source is super-Gaussian.

```
q = 6;  
Mdl = rica(mixdata,q, 'NonGaussianityIndicator',ones(6,1));
```

Extract the features. If the unmixing procedure is successful, the features are proportional to the original signals.

```
unmixed = transform(Mdl,mixdata);
```

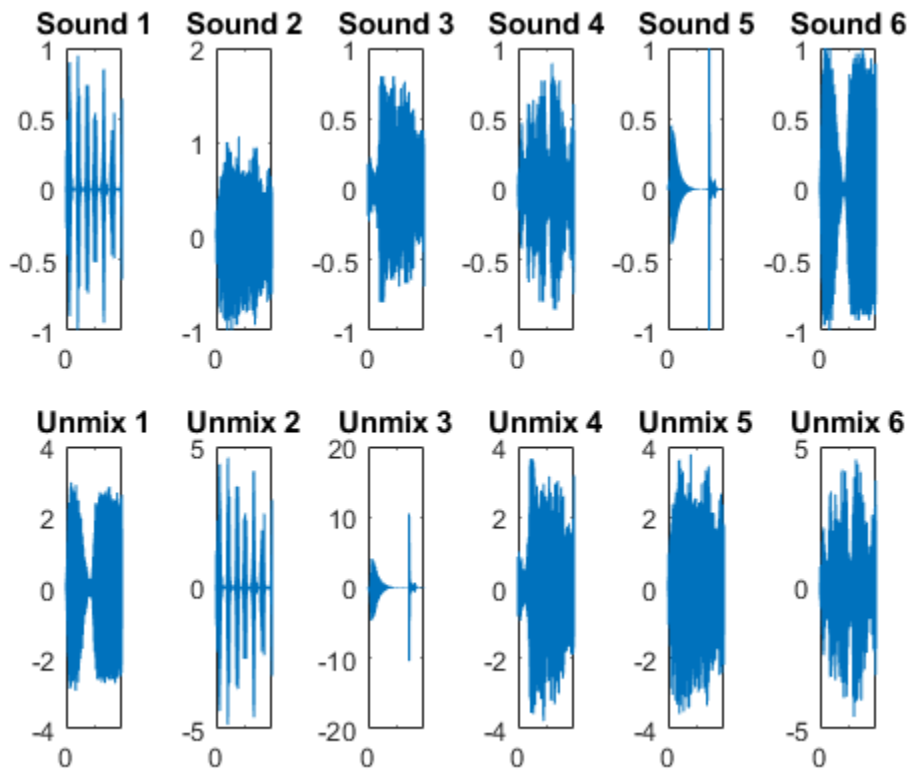
### Compare Unmixed Signals To Original Signals

Plot the original and unmixed signals.

```

figure
for i = 1:6
    subplot(2,6,i)
    plot(S(:,i))
    title(['Sound ',num2str(i)])
    subplot(2,6,i+6)
    plot(unmixed(:,i))
    title(['Unmix ',num2str(i)])
end

```



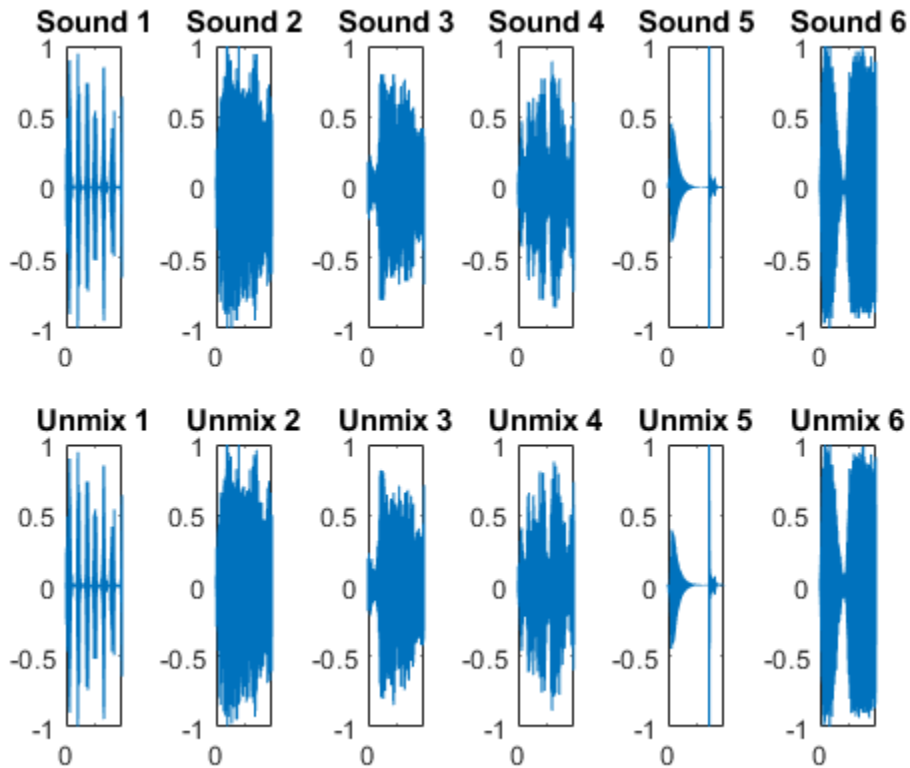
The order of the unmixed signals is different than the original order. Reorder the columns so that the unmixed signals match the corresponding original signals. Scale the unmixed signals to have the same norms as the corresponding original signals. (rica

cannot identify the scale of the original signals because any scale can lead to the same signal mixture.)

```
unmixed = unmixed(:, [2, 5, 4, 6, 3, 1]);  
for i = 1:6  
    unmixed(:, i) = unmixed(:, i) / norm(unmixed(:, i)) * norm(S(:, i));  
end
```

Plot the original and unmixed signals.

```
figure  
for i = 1:6  
    subplot(2, 6, i)  
    plot(S(:, i))  
    ylim([-1, 1])  
    title(['Sound ', num2str(i)])  
    subplot(2, 6, i+6)  
    plot(unmixed(:, i))  
    ylim([-1, 1])  
    title(['Unmix ', num2str(i)])  
end
```



The unmixed signals look similar to the original signals. To listen to the unmixed sounds, execute this code.

```
for i = 1:6
    disp(i);
    sound(unmixed(:,i));
    pause;
end
```

Here is the code for the prewhiten function.

```
function Z = prewhiten(X)
% X = N-by-P matrix for N observations and P predictors
```

```
% Z = N-by-P prewhitened matrix

% 1. Size of X.
[N,P] = size(X);
assert(N >= P);

% 2. SVD of covariance of X. We could also use svd(X) to proceed but N
% can be large and so we sacrifice some accuracy for speed.
[U,Sig] = svd(cov(X));
Sig      = diag(Sig);
Sig      = Sig(:)';

% 3. Figure out which values of Sig are non-zero.
tol = eps(class(X));
idx = (Sig > max(Sig)*tol);
assert(~all(idx == 0));

% 4. Get the non-zero elements of Sig and corresponding columns of U.
Sig = Sig(idx);
U   = U(:,idx);

% 5. Compute prewhitened data.
mu = mean(X,1);
Z = bsxfun(@minus,X,mu);
Z = bsxfun(@times,Z*U,1./sqrt(Sig));
end
```

## See Also

ReconstructionICA | SparseFiltering | rica | sparsefilt

## Related Examples

- “Feature Extraction Workflow” on page 15-169

## More About

- “Feature Extraction” on page 15-163





# Cluster Analysis

---

- “Introduction to Cluster Analysis” on page 16-2
- “Hierarchical Clustering” on page 16-3
- “k-Means Clustering” on page 16-21
- “Clustering Using Gaussian Mixture Models” on page 16-28
- “Cluster Data from Mixture of Gaussian Distributions” on page 16-37
- “Cluster Gaussian Mixture Data Using Soft Clustering” on page 16-47
- “Tune Gaussian Mixture Models” on page 16-55

## Introduction to Cluster Analysis

Cluster analysis, also called segmentation analysis or taxonomy analysis, creates groups, or *clusters*, of data. Clusters are formed in such a way that objects in the same cluster are very similar and objects in different clusters are very distinct. Measures of similarity depend on the application.

“Hierarchical Clustering” on page 16-3 groups data over a variety of scales by creating a cluster tree or dendrogram. The tree is not a single set of clusters, but rather a multilevel hierarchy, where clusters at one level are joined as clusters at the next level. This allows you to decide the level or scale of clustering that is most appropriate for your application. The Statistics and Machine Learning Toolbox function `clusterdata` performs all of the necessary steps for you. It incorporates the `pdist`, `linkage`, and `cluster` functions, which may be used separately for more detailed analysis. The `dendrogram` function plots the cluster tree.

“k-Means Clustering” on page 16-21 is a partitioning method. The function `kmeans` partitions data into  $k$  mutually exclusive clusters, and returns the index of the cluster to which it has assigned each observation. Unlike hierarchical clustering,  $k$ -means clustering operates on actual observations (rather than the larger set of dissimilarity measures), and creates a single level of clusters. The distinctions mean that  $k$ -means clustering is often more suitable than hierarchical clustering for large amounts of data.

“Clustering Using Gaussian Mixture Models” on page 16-28 form clusters by representing the probability density function of observed variables as a mixture of multivariate normal densities. Mixture models of the `gmdistribution` class use an expectation maximization (EM) algorithm to fit data, which assigns posterior probabilities to each component density with respect to each observation. Clusters are assigned by selecting the component that maximizes the posterior probability. Clustering using Gaussian mixture models is sometimes considered a soft clustering method. The posterior probabilities for each point indicate that each data point has some probability of belonging to each cluster. Like  $k$ -means clustering, Gaussian mixture modeling uses an iterative algorithm that converges to a local optimum. Gaussian mixture modeling may be more appropriate than  $k$ -means clustering when clusters have different sizes and correlation within them.

# Hierarchical Clustering

## In this section...

“Introduction to Hierarchical Clustering” on page 16-3

“Algorithm Description” on page 16-3

“Similarity Measures” on page 16-4

“Linkages” on page 16-6

“Dendrograms” on page 16-8

“Verify the Cluster Tree” on page 16-9

“Create Clusters” on page 16-16

## Introduction to Hierarchical Clustering

Hierarchical clustering groups data over a variety of scales by creating a cluster tree or dendrogram. The tree is not a single set of clusters, but rather a multilevel hierarchy, where clusters at one level are joined as clusters at the next level. This allows you to decide the level or scale of clustering that is most appropriate for your application. The Statistics and Machine Learning Toolbox function `clusterdata` supports agglomerative clustering and performs all of the necessary steps for you. It incorporates the `pdist`, `linkage`, and `cluster` functions, which you can use separately for more detailed analysis. The `dendrogram` function plots the cluster tree.

## Algorithm Description

To perform agglomerative hierarchical cluster analysis on a data set using Statistics and Machine Learning Toolbox functions, follow this procedure:

- 1 Find the similarity or dissimilarity between every pair of objects in the data set.** In this step, you calculate the *distance* between objects using the `pdist` function. The `pdist` function supports many different ways to compute this measurement. See “Similarity Measures” on page 16-4 for more information.
- 2 Group the objects into a binary, hierarchical cluster tree.** In this step, you link pairs of objects that are in close proximity using the `linkage` function. The `linkage` function uses the distance information generated in step 1 to determine the proximity of objects to each other. As objects are paired into binary clusters, the

newly formed clusters are grouped into larger clusters until a hierarchical tree is formed. See “Linkages” on page 16-6 for more information.

- 3 Determine where to cut the hierarchical tree into clusters.** In this step, you use the `cluster` function to prune branches off the bottom of the hierarchical tree, and assign all the objects below each cut to a single cluster. This creates a partition of the data. The `cluster` function can create these clusters by detecting natural groupings in the hierarchical tree or by cutting off the hierarchical tree at an arbitrary point.

The following sections provide more information about each of these steps.

---

**Note** The Statistics and Machine Learning Toolbox function `clusterdata` performs all of the necessary steps for you. You do not need to execute the `pdist`, `linkage`, or `cluster` functions separately.

---

## Similarity Measures

You use the `pdist` function to calculate the distance between every pair of objects in a data set. For a data set made up of  $m$  objects, there are  $m*(m - 1)/2$  pairs in the data set. The result of this computation is commonly known as a distance or dissimilarity matrix.

There are many ways to calculate this distance information. By default, the `pdist` function calculates the Euclidean distance between objects; however, you can specify one of several other options. See `pdist` for more information.

---

**Note** You can optionally normalize the values in the data set before calculating the distance information. In a real world data set, variables can be measured against different scales. For example, one variable can measure Intelligence Quotient (IQ) test scores and another variable can measure head circumference. These discrepancies can distort the proximity calculations. Using the `zscore` function, you can convert all the values in the data set to use the same proportional scale. See `zscore` for more information.

---

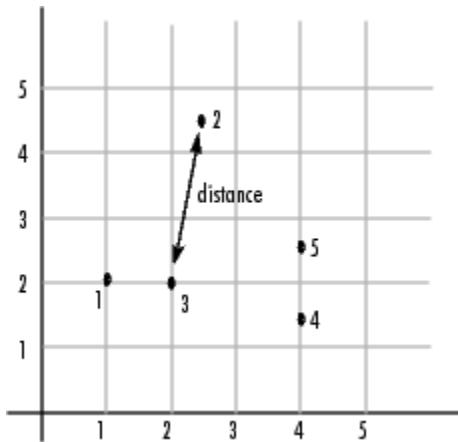
For example, consider a data set,  $X$ , made up of five objects where each object is a set of  $x,y$  coordinates.

- **Object 1:** 1, 2
- **Object 2:** 2.5, 4.5
- **Object 3:** 2, 2
- **Object 4:** 4, 1.5
- **Object 5:** 4, 2.5

You can define this data set as a matrix

```
rng default; % For reproducibility
X = [1 2; 2.5 4.5; 2 2; 4 1.5; ...
     4 2.5];
```

and pass it to `pdist`. The `pdist` function calculates the distance between object 1 and object 2, object 1 and object 3, and so on until the distances between all the pairs have been calculated. The following figure plots these objects in a graph. The Euclidean distance between object 2 and object 3 is shown to illustrate one interpretation of distance.



### Distance Information

The `pdist` function returns this distance information in a vector, `Y`, where each element contains the distance between a pair of objects.

```
Y = pdist(X)
```

```
Y =
```

```
Columns 1 through 7
  2.9155    1.0000    3.0414    3.0414    2.5495    3.3541    2.5000

Columns 8 through 10
  2.0616    2.0616    1.0000
```

To make it easier to see the relationship between the distance information generated by `pdist` and the objects in the original data set, you can reformat the distance vector into a matrix using the `squareform` function. In this matrix, element  $i,j$  corresponds to the distance between object  $i$  and object  $j$  in the original data set. In the following example, element 1,1 represents the distance between object 1 and itself (which is zero). Element 1,2 represents the distance between object 1 and object 2, and so on.

```
squareform(Y)

ans =

     0    2.9155    1.0000    3.0414    3.0414
  2.9155         0    2.5495    3.3541    2.5000
  1.0000    2.5495         0    2.0616    2.0616
  3.0414    3.3541    2.0616         0    1.0000
  3.0414    2.5000    2.0616    1.0000         0
```

## Linkages

Once the proximity between objects in the data set has been computed, you can determine how objects in the data set should be grouped into clusters, using the `linkage` function. The `linkage` function takes the distance information generated by `pdist` and links pairs of objects that are close together into binary clusters (clusters made up of two objects). The `linkage` function then links these newly formed clusters to each other and to other objects to create bigger clusters until all the objects in the original data set are linked together in a hierarchical tree.

For example, given the distance vector  $Y$  generated by `pdist` from the sample data set of  $x$ - and  $y$ -coordinates, the `linkage` function generates a hierarchical cluster tree, returning the linkage information in a matrix,  $Z$ .

```
Z = linkage(Y)
```

```
Z =
```

```

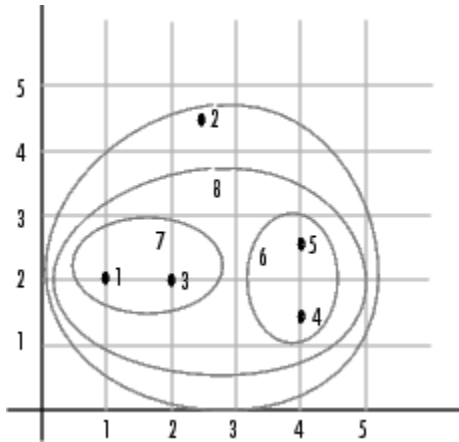
4.0000    5.0000    1.0000
1.0000    3.0000    1.0000
6.0000    7.0000    2.0616
2.0000    8.0000    2.5000
```

In this output, each row identifies a link between objects or clusters. The first two columns identify the objects that have been linked. The third column contains the distance between these objects. For the sample data set of  $x$ - and  $y$ -coordinates, the `linkage` function begins by grouping objects 4 and 5, which have the closest proximity (distance value = 1.0000). The `linkage` function continues by grouping objects 1 and 3, which also have a distance value of 1.0000.

The third row indicates that the `linkage` function grouped objects 6 and 7. If the original sample data set contained only five objects, what are objects 6 and 7? Object 6 is the newly formed binary cluster created by the grouping of objects 4 and 5. When the `linkage` function groups two objects into a new cluster, it must assign the cluster a unique index value, starting with the value  $m + 1$ , where  $m$  is the number of objects in the original data set. (Values 1 through  $m$  are already used by the original data set.) Similarly, object 7 is the cluster formed by grouping objects 1 and 3.

`linkage` uses distances to determine the order in which it clusters objects. The distance vector `Y` contains the distances between the original objects 1 through 5. But `linkage` must also be able to determine distances involving clusters that it creates, such as objects 6 and 7. By default, `linkage` uses a method known as single linkage. However, there are a number of different methods available. See the `linkage` reference page for more information.

As the final cluster, the `linkage` function grouped object 8, the newly formed cluster made up of objects 6 and 7, with object 2 from the original data set. The following figure graphically illustrates the way `linkage` groups the objects into a hierarchy of clusters.

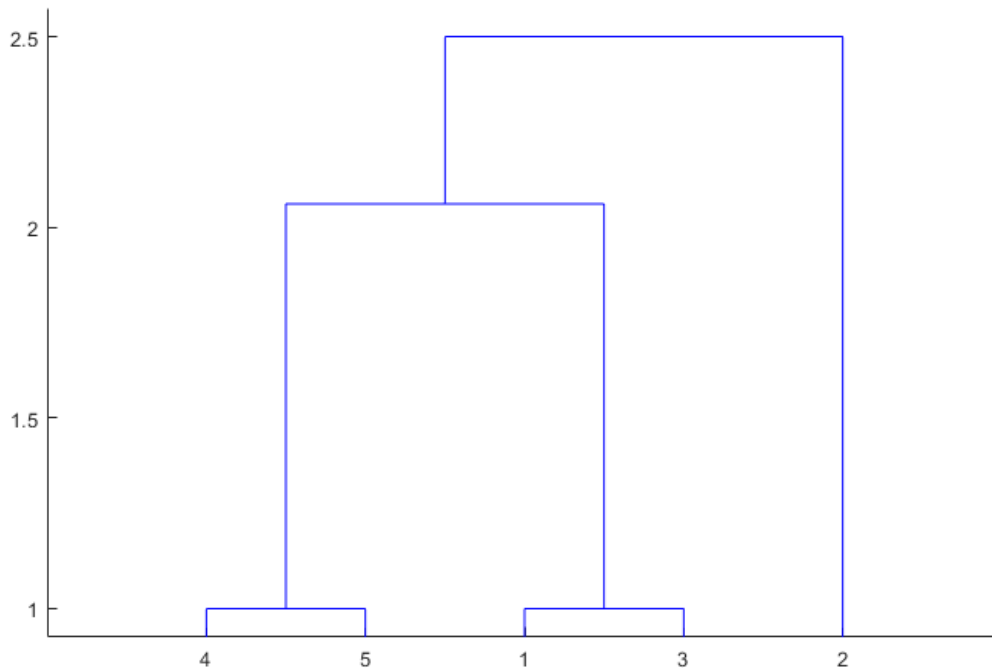


## Dendrograms

The hierarchical, binary cluster tree created by the `linkage` function is most easily understood when viewed graphically. The Statistics and Machine Learning Toolbox function `dendrogram` plots the tree as follows.

```
dendrogram(Z)
```





In the figure, the numbers along the horizontal axis represent the indices of the objects in the original data set. The links between objects are represented as upside-down U-shaped lines. The height of the U indicates the distance between the objects. For example, the link representing the cluster containing objects 1 and 3 has a height of 1. The link representing the cluster that groups object 2 together with objects 1, 3, 4, and 5, (which are already clustered as object 8) has a height of 2.5. The height represents the distance linkage computes between objects 2 and 8. For more information about creating a dendrogram diagram, see the [dendrogram reference page](#).

## Verify the Cluster Tree

After linking the objects in a data set into a hierarchical cluster tree, you might want to verify that the distances (that is, heights) in the tree reflect the original distances accurately. In addition, you might want to investigate natural divisions that exist among

links between objects. Statistics and Machine Learning Toolbox functions are available for both of these tasks, as described in the following sections.

- “Verify Dissimilarity” on page 16-10
- “Verify Consistency” on page 16-11

### Verify Dissimilarity

In a hierarchical cluster tree, any two objects in the original data set are eventually linked together at some level. The height of the link represents the distance between the two clusters that contain those two objects. This height is known as the *cophenetic distance* between the two objects. One way to measure how well the cluster tree generated by the `linkage` function reflects your data is to compare the cophenetic distances with the original distance data generated by the `pdist` function. If the clustering is valid, the linking of objects in the cluster tree should have a strong correlation with the distances between objects in the distance vector. The `cophenet` function compares these two sets of values and computes their correlation, returning a value called the *cophenetic correlation coefficient*. The closer the value of the cophenetic correlation coefficient is to 1, the more accurately the clustering solution reflects your data.

You can use the cophenetic correlation coefficient to compare the results of clustering the same data set using different distance calculation methods or clustering algorithms. For example, you can use the `cophenet` function to evaluate the clusters created for the sample data set.

```
c = cophenet(Z,Y)
```

```
c =
```

```
0.8615
```

Z is the matrix output by the `linkage` function and Y is the distance vector output by the `pdist` function.

Execute `pdist` again on the same data set, this time specifying the city block metric. After running the `linkage` function on this new `pdist` output using the average linkage method, call `cophenet` to evaluate the clustering solution.

```
Y = pdist(X, 'cityblock');  
Z = linkage(Y, 'average');  
c = cophenet(Z, Y)
```

```
c =  
  
    0.9047
```

The cophenetic correlation coefficient shows that using a different distance and linkage method creates a tree that represents the original distances slightly better.

### Verify Consistency

One way to determine the natural cluster divisions in a data set is to compare the height of each link in a cluster tree with the heights of neighboring links below it in the tree.

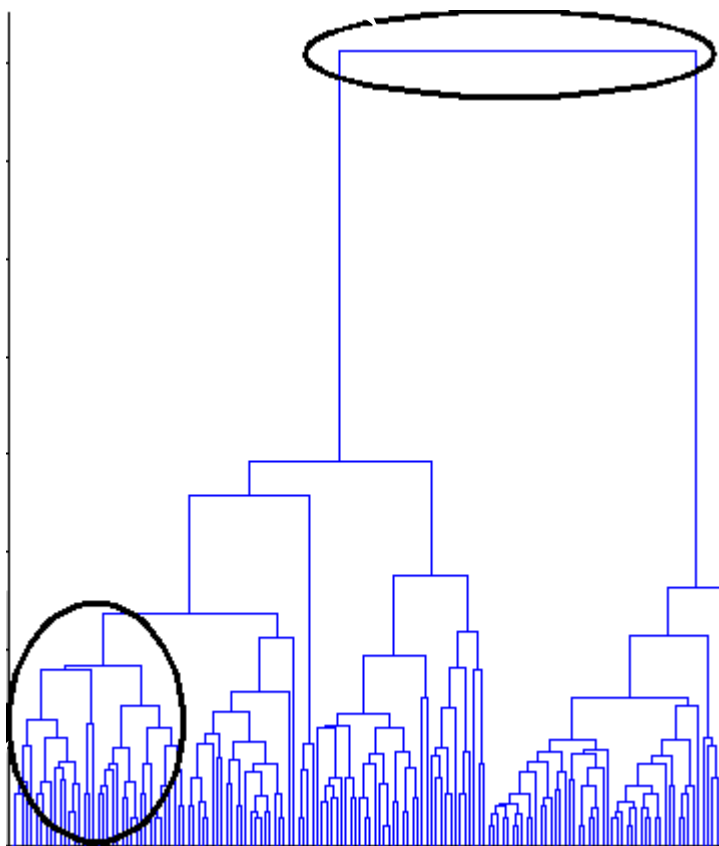
A link that is approximately the same height as the links below it indicates that there are no distinct divisions between the objects joined at this level of the hierarchy. These links are said to exhibit a high level of consistency, because the distance between the objects being joined is approximately the same as the distances between the objects they contain.

On the other hand, a link whose height differs noticeably from the height of the links below it indicates that the objects joined at this level in the cluster tree are much farther apart from each other than their components were when they were joined. This link is said to be inconsistent with the links below it.

In cluster analysis, inconsistent links can indicate the border of a natural division in a data set. The `cluster` function uses a quantitative measure of inconsistency to determine where to partition your data set into clusters.

The following dendrogram illustrates inconsistent links. Note how the objects in the dendrogram fall into two groups that are connected by links at a much higher level in the tree. These links are inconsistent when compared with the links below them in the hierarchy.

These links show inconsistency when compared to the links below them.



These links show consistency.

The relative consistency of each link in a hierarchical cluster tree can be quantified and expressed as the *inconsistency coefficient*. This value compares the height of a link in a cluster hierarchy with the average height of links below it. Links that join distinct clusters have a high inconsistency coefficient; links that join indistinct clusters have a low inconsistency coefficient.

To generate a listing of the inconsistency coefficient for each link in the cluster tree, use the `inconsistent` function. By default, the `inconsistent` function compares each link

in the cluster hierarchy with adjacent links that are less than two levels below it in the cluster hierarchy. This is called the *depth* of the comparison. You can also specify other depths. The objects at the bottom of the cluster tree, called leaf nodes, that have no further objects below them, have an inconsistency coefficient of zero. Clusters that join two leaves also have a zero inconsistency coefficient.

For example, you can use the `inconsistent` function to calculate the inconsistency values for the links created by the `linkage` function in “Linkages” on page 16-6.

First, recompute the distance and linkage values using the default settings.

```
Y = pdist(X);
Z = linkage(Y);
```

Next, use `inconsistent` to calculate the inconsistency values.

```
I = inconsistent(Z)
```

```
I =
```

```

1.0000         0    1.0000         0
1.0000         0    1.0000         0
1.3539    0.6129    3.0000    1.1547
2.2808    0.3100    2.0000    0.7071
```

The `inconsistent` function returns data about the links in an  $(m-1)$ -by-4 matrix, whose columns are described in the following table.

Column	Description
1	Mean of the heights of all the links included in the calculation
2	Standard deviation of all the links included in the calculation
3	Number of links included in the calculation
4	Inconsistency coefficient

In the sample output, the first row represents the link between objects 4 and 5. This cluster is assigned the index 6 by the `linkage` function. Because both 4 and 5 are leaf nodes, the inconsistency coefficient for the cluster is zero. The second row represents the link between objects 1 and 3, both of which are also leaf nodes. This cluster is assigned the index 7 by the `linkage` function.

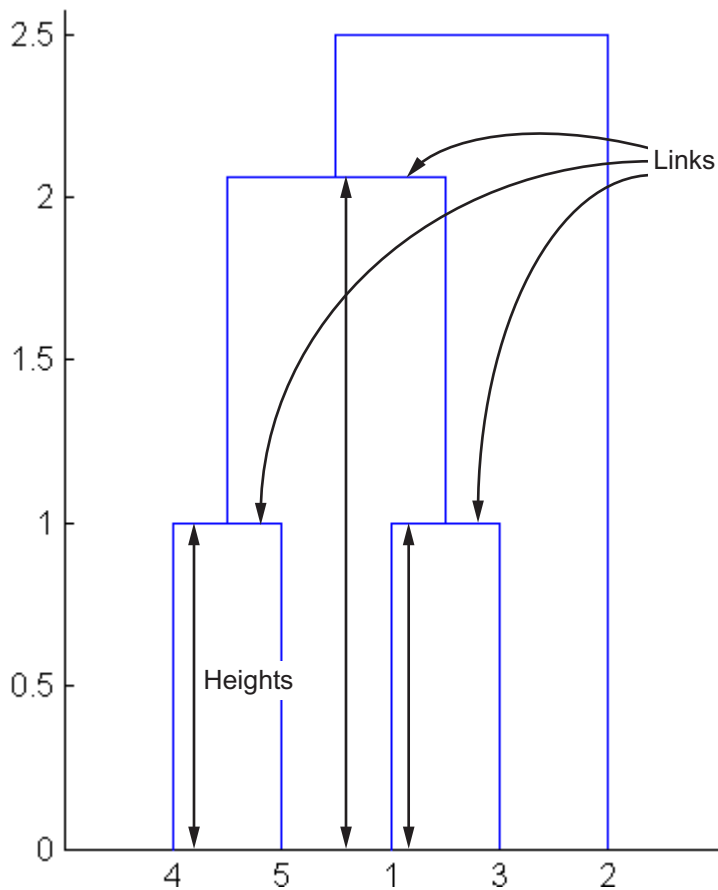
The third row evaluates the link that connects these two clusters, objects 6 and 7. (This new cluster is assigned index 8 in the `linkage` output). Column 3 indicates that three links are considered in the calculation: the link itself and the two links directly below it in the hierarchy. Column 1 represents the mean of the heights of these links. The `inconsistent` function uses the height information output by the `linkage` function to calculate the mean. Column 2 represents the standard deviation between the links. The last column contains the inconsistency value for these links, 1.1547. It is the difference between the current link height and the mean, normalized by the standard deviation.

```
(2.0616 - 1.3539) / .6129
```

```
ans =
```

```
1.1547
```

The following figure illustrates the links and heights included in this calculation.



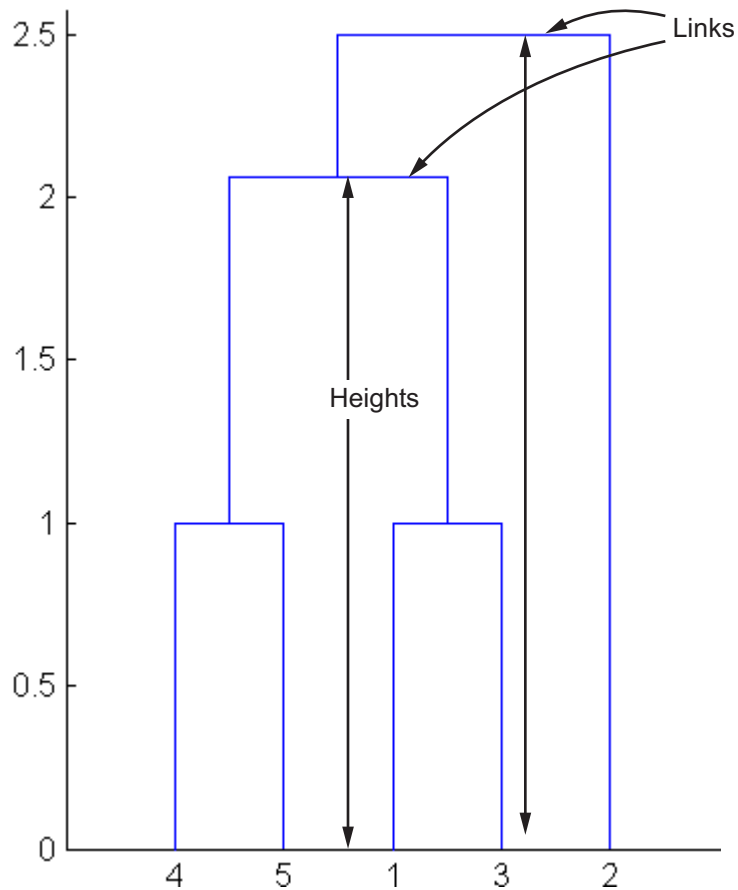

---

**Note** In the preceding figure, the lower limit on the y-axis is set to 0 to show the heights of the links. To set the lower limit to 0, select **Axes Properties** from the **Edit** menu, click the **Y Axis** tab, and enter 0 in the field immediately to the right of **Y Limits**.

---

Row 4 in the output matrix describes the link between object 8 and object 2. Column 3 indicates that two links are included in this calculation: the link itself and the link directly below it in the hierarchy. The inconsistency coefficient for this link is 0.7071.

The following figure illustrates the links and heights included in this calculation.



## Create Clusters

After you create the hierarchical tree of binary clusters, you can prune the tree to partition your data into clusters using the `cluster` function. The `cluster` function lets you create clusters in two ways, as discussed in the following sections:

- “Find Natural Divisions in Data” on page 16-17
- “Specify Arbitrary Clusters” on page 16-18



## Find Natural Divisions in Data

The hierarchical cluster tree may naturally divide the data into distinct, well-separated clusters. This can be particularly evident in a dendrogram diagram created from data where groups of objects are densely packed in certain areas and not in others. The inconsistency coefficient of the links in the cluster tree can identify these divisions where the similarities between objects change abruptly. (See “Verify the Cluster Tree” on page 16-9 for more information about the inconsistency coefficient.) You can use this value to determine where the `cluster` function creates cluster boundaries.

For example, if you use the `cluster` function to group the sample data set into clusters, specifying an inconsistency coefficient threshold of 1.2 as the value of the `cutoff` argument, the `cluster` function groups all the objects in the sample data set into one cluster. In this case, none of the links in the cluster hierarchy had an inconsistency coefficient greater than 1.2.

```
T = cluster(Z, 'cutoff', 1.2)
```

```
T =  
  
    1  
    1  
    1  
    1  
    1
```

The `cluster` function outputs a vector, `T`, that is the same size as the original data set. Each element in this vector contains the number of the cluster into which the corresponding object from the original data set was placed.

If you lower the inconsistency coefficient threshold to 0.8, the `cluster` function divides the sample data set into three separate clusters.

```
T = cluster(Z, 'cutoff', 0.8)
```

```
T =  
  
    3  
    2  
    3
```

```
1  
1
```

This output indicates that objects 1 and 3 are in one cluster, objects 4 and 5 are in another cluster, and object 2 is in its own cluster.

When clusters are formed in this way, the cutoff value is applied to the inconsistency coefficient. These clusters may, but do not necessarily, correspond to a horizontal slice across the dendrogram at a certain height. If you want clusters corresponding to a horizontal slice of the dendrogram, you can either use the `criterion` option to specify that the cutoff should be based on distance rather than inconsistency, or you can specify the number of clusters directly as described in the following section.

### Specify Arbitrary Clusters

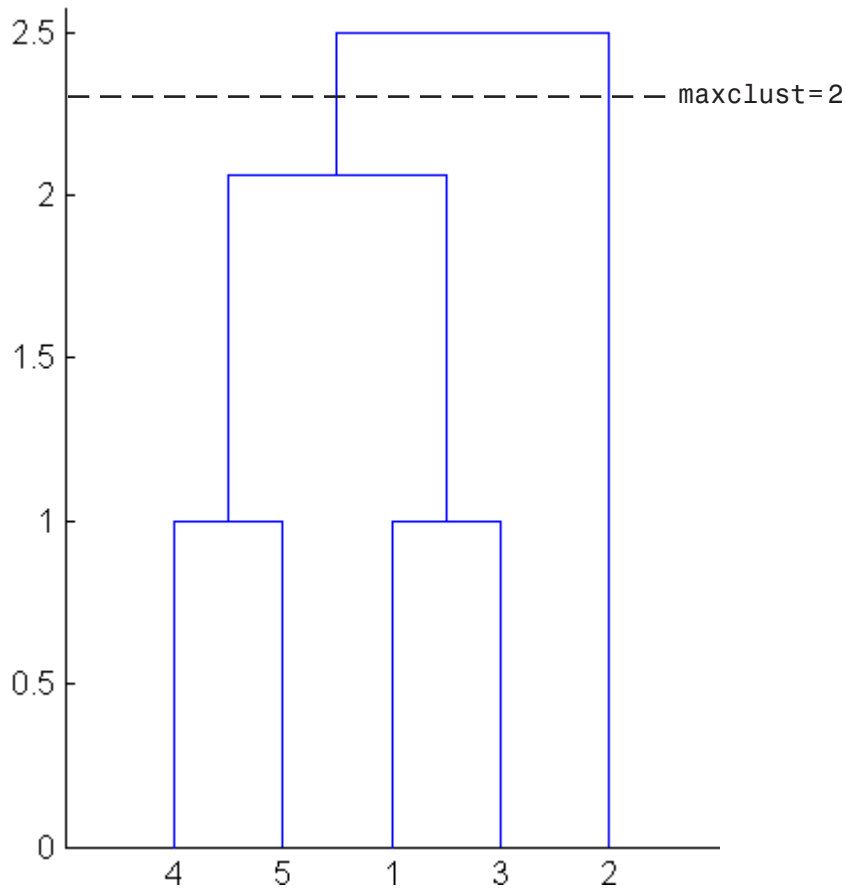
Instead of letting the `cluster` function create clusters determined by the natural divisions in the data set, you can specify the number of clusters you want created.

For example, you can specify that you want the `cluster` function to partition the sample data set into two clusters. In this case, the `cluster` function creates one cluster containing objects 1, 3, 4, and 5 and another cluster containing object 2.

```
T = cluster(Z, 'maxclust', 2)
```

```
T =  
  
2  
1  
2  
2  
2
```

To help you visualize how the `cluster` function determines these clusters, the following figure shows the dendrogram of the hierarchical cluster tree. The horizontal dashed line intersects two lines of the dendrogram, corresponding to setting `'maxclust'` to 2. These two lines partition the objects into two clusters: the objects below the left-hand line, namely 1, 3, 4, and 5, belong to one cluster, while the object below the right-hand line, namely 2, belongs to the other cluster.



On the other hand, if you set 'maxclust' to 3, the cluster function groups objects 4 and 5 in one cluster, objects 1 and 3 in a second cluster, and object 2 in a third cluster. The following command illustrates this.

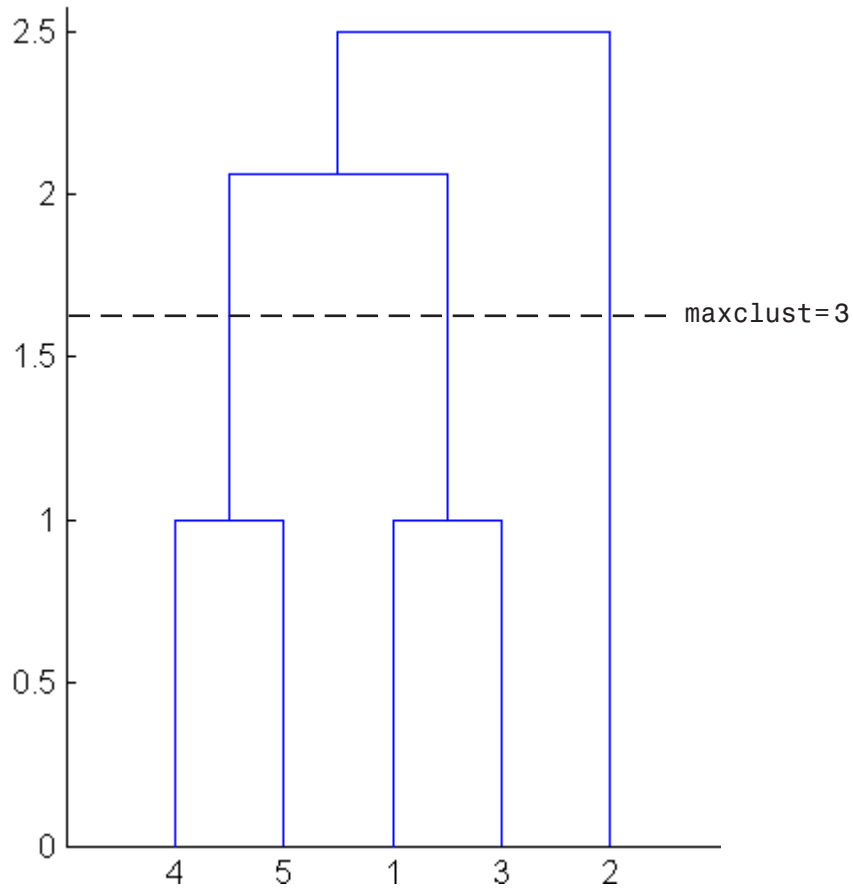
```
T = cluster(Z, 'maxclust', 3)
```

```
T =
```

```
 2
 3
 2
 1
```

1

This time, the `cluster` function cuts off the hierarchy at a lower point, corresponding to the horizontal line that intersects three lines of the dendrogram in the following figure.



## k-Means Clustering

### In this section...

“Introduction to k-Means Clustering” on page 16-21

“Create Clusters and Determine Separation” on page 16-21

“Determine the Correct Number of Clusters” on page 16-23

“Avoid Local Minima” on page 16-26

### Introduction to k-Means Clustering

*k*-means clustering is a partitioning method. The function `kmeans` partitions data into *k* mutually exclusive clusters, and returns the index of the cluster to which it has assigned each observation. Unlike hierarchical clustering, *k*-means clustering operates on actual observations (rather than the larger set of dissimilarity measures), and creates a single level of clusters. The distinctions mean that *k*-means clustering is often more suitable than hierarchical clustering for large amounts of data.

`kmeans` treats each observation in your data as an object having a location in space. It finds a partition in which objects within each cluster are as close to each other as possible, and as far from objects in other clusters as possible. You can choose from five different distance measures, depending on the kind of data you are clustering.

Each cluster in the partition is defined by its member objects and by its centroid, or center. The centroid for each cluster is the point to which the sum of distances from all objects in that cluster is minimized. `kmeans` computes cluster centroids differently for each distance measure, to minimize the sum with respect to the measure that you specify.

You can control the details of the minimization using several optional input parameters to `kmeans`, including ones for the initial values of the cluster centroids, and for the maximum number of iterations. By default, `kmeans` uses the *k*-means++ algorithm for cluster center initialization and the squared Euclidean metric to determine distances.

### Create Clusters and Determine Separation

The following example explores possible clustering in four-dimensional data by analyzing the results of partitioning the points into three, four, and five clusters.

**Note** Because each part of this example generates random numbers sequentially, i.e., without setting a new state, you must perform all steps in sequence to duplicate the results shown. If you perform the steps out of sequence, the answers will be essentially the same, but the intermediate results, number of iterations, or ordering of the silhouette plots may differ.

---

First, load some data.

```
rng default % For reproducibility
load kmeansdata
size(X)
```

```
ans =

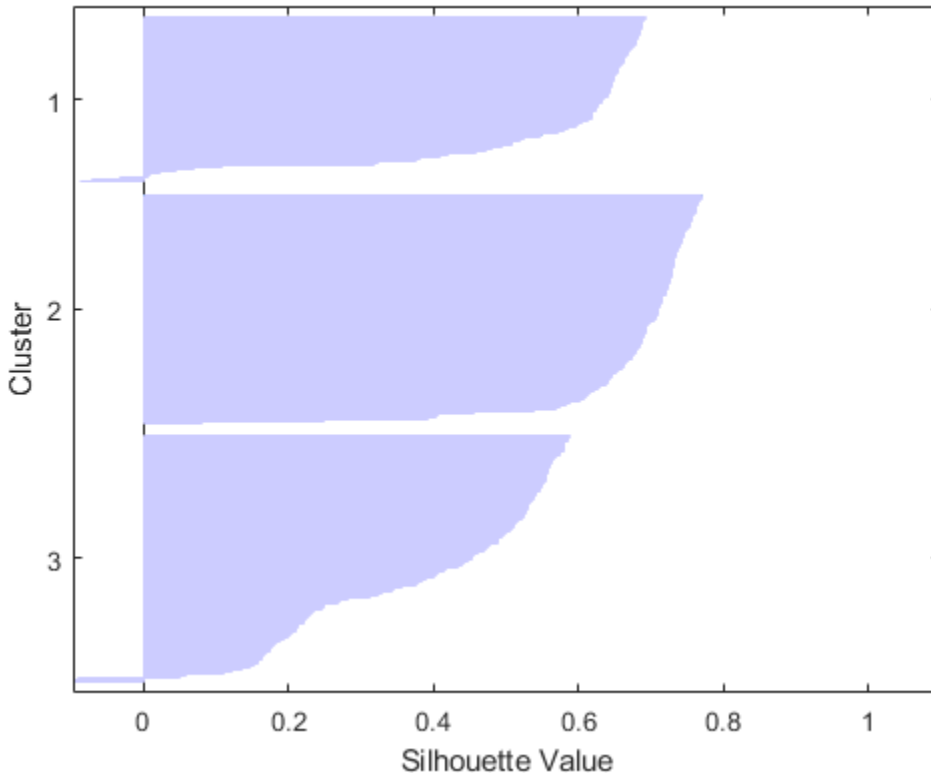
    560     4
```

Even though these data are four-dimensional, and cannot be easily visualized, `kmeans` enables you to investigate whether a group structure exists in them. Call `kmeans` with `k`, the desired number of clusters, equal to 3. For this example, specify the city block distance measure, and use the default *k*-means++ algorithm for cluster center initialization.

```
idx3 = kmeans(X,3,'Distance','cityblock');
```

To get an idea of how well-separated the resulting clusters are, you can make a silhouette plot using the cluster indices output from `kmeans`. The silhouette plot displays a measure of how close each point in one cluster is to points in the neighboring clusters. This measure ranges from +1, indicating points that are very distant from neighboring clusters, through 0, indicating points that are not distinctly in one cluster or another, to -1, indicating points that are probably assigned to the wrong cluster. `silhouette` returns these values in its first output.

```
figure
[silh3,h] = silhouette(X,idx3,'cityblock');
h = gca;
h.Children.EdgeColor = [.8 .8 1];
xlabel 'Silhouette Value'
ylabel 'Cluster'
```



From the silhouette plot, you can see that most points in the second cluster have a large silhouette value, greater than 0.6, indicating that the cluster is somewhat separated from neighboring clusters. However, the third cluster contains many points with low silhouette values, and the first and third contain a few points with negative values, indicating that those two clusters are not well separated.

## Determine the Correct Number of Clusters

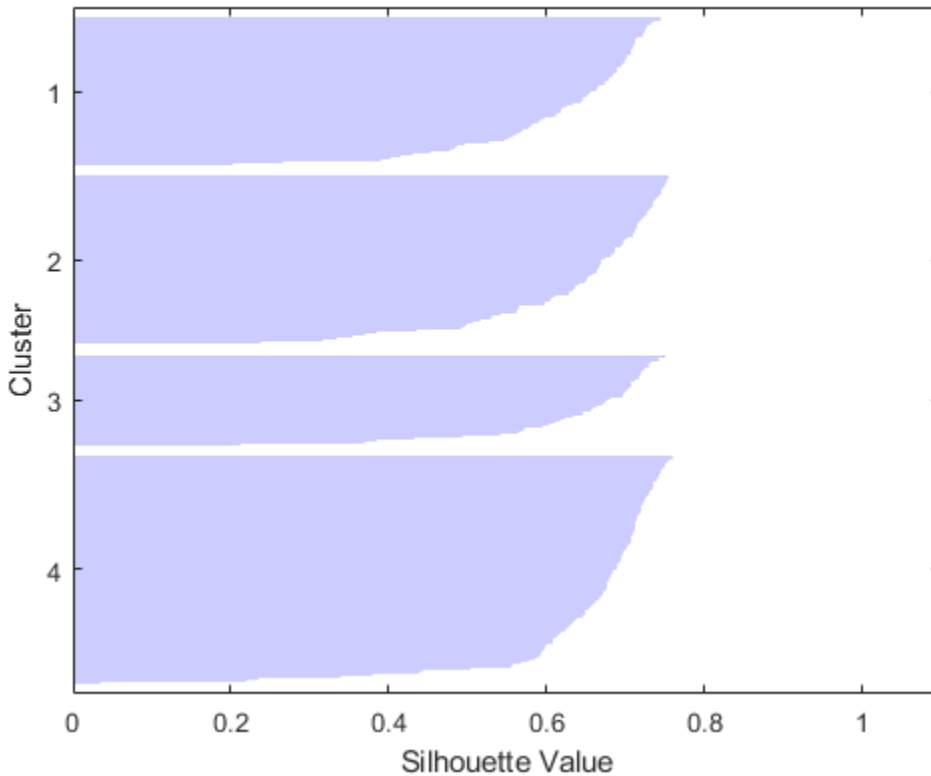
Increase the number of clusters to see if `kmeans` can find a better grouping of the data. This time, use the `'Display'` name-value pair argument to print information about each iteration.

```
idx4 = kmeans(X,4, 'Distance','cityblock','Display','iter');
```

```
iter    phase    num    sum
  1      1      560  1792.72
  2      1       6   1771.1
Best total sum of distances = 1771.1
```

A silhouette plot for this solution indicates that these four clusters are better separated than the three in the previous solution.

```
figure
[silh4,h] = silhouette(X,idx4,'cityblock');
h = gca;
h.Children.EdgeColor = [.8 .8 1];
xlabel 'Silhouette Value'
ylabel 'Cluster'
```





A more quantitative way to compare the two solutions is to look at the average silhouette values for the two cases.

```
cluster3 = mean(silh3)
cluster4 = mean(silh4)
```

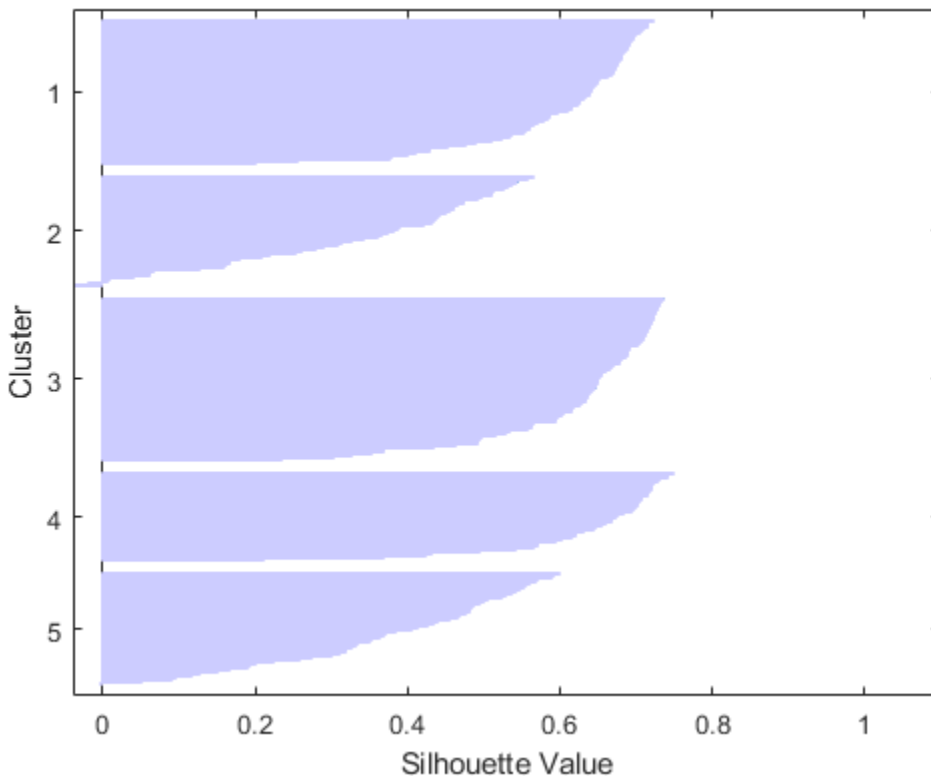
```
cluster3 =
    0.5352
```

```
cluster4 =
    0.6400
```

Finally, try clustering the data using five clusters.

```
idx5 = kmeans(X,5,'Distance','cityblock','Replicates',5);
figure
[silh5,h] = silhouette(X,idx5,'city');
h = gca;
h.Children.EdgeColor = [.8 .8 1];
xlabel 'Silhouette Value'
ylabel 'Cluster'
mean(silh5)
```

```
ans =
    0.5266
```



This silhouette plot indicates that this is probably not the right number of clusters, since two of the clusters contain points with mostly low silhouette values. Without some knowledge of how many clusters are really in the data, it is a good idea to experiment with a range of values for  $k$ .

## Avoid Local Minima

Like many other types of numerical minimizations, the solution that `kmeans` reaches often depends on the starting points. It is possible for `kmeans` to reach a local minimum, where reassigning any one point to a new cluster would increase the total sum of point-to-centroid distances, but where a better solution does exist. However, you can use the `'Replicates'` name-value pair argument to overcome that problem.

For four clusters, specify five replicates, and use the 'Display' name-value pair argument to print out the final sum of distances for each of the solutions.

```
[idx4,cent4,sumdist] = kmeans(X,4,'Distance','cityblock',...  
                             'Display','final','Replicates',5);
```

```
Replicate 1, 4 iterations, total sum of distances = 1771.1.  
Replicate 2, 2 iterations, total sum of distances = 1771.1.  
Replicate 3, 5 iterations, total sum of distances = 2303.45.  
Replicate 4, 4 iterations, total sum of distances = 2303.45.  
Replicate 5, 6 iterations, total sum of distances = 1771.1.  
Best total sum of distances = 1771.1
```

In two of the five replications, `kmeans` found a local (nonglobal) minimum. Since each of these five replicates begin from a different randomly selected set of initial centroids, sometimes `kmeans` finds more than one local minimum. However, the final solution that `kmeans` returns is the one with the lowest total sum of distances, over all replicates.

```
sum(sumdist)
```

```
ans =
```

```
1.7711e+03
```

## Clustering Using Gaussian Mixture Models

### How Gaussian Mixture Models Cluster Data

Gaussian mixture models (GMM) are often used for data clustering. Usually, fitted GMMs cluster by assigning query data points to the multivariate normal components that maximize the component posterior probability given the data. That is, given a fitted GMM, `gmdistribution.cluster` assigns query data to the component yielding the highest posterior probability. This method of assigning a data point to exactly one cluster is called *hard* clustering. For an example showing how to fit a GMM to data, cluster using the fitted model, and estimate component posterior probabilities, see “Cluster Data from Mixture of Gaussian Distributions” on page 16-37.

However, GMM clustering is more flexible because you can view it as a *fuzzy* or *soft clustering* method. Soft clustering methods assign a score to a data point for each cluster. The value of the score indicates the association strength of the data point to the cluster. As opposed to hard clustering methods, soft clustering methods are flexible in that they can assign a data point to more than one cluster. When clustering with GMMs, the score is the posterior probability. For an example of soft clustering using GMM, see “Cluster Gaussian Mixture Data Using Soft Clustering” on page 16-47.

Moreover, GMM clustering can accommodate clusters that have different sizes and correlation structures within them. Because of this, GMM clustering can be more appropriate to use than, e.g, *k*-means clustering.

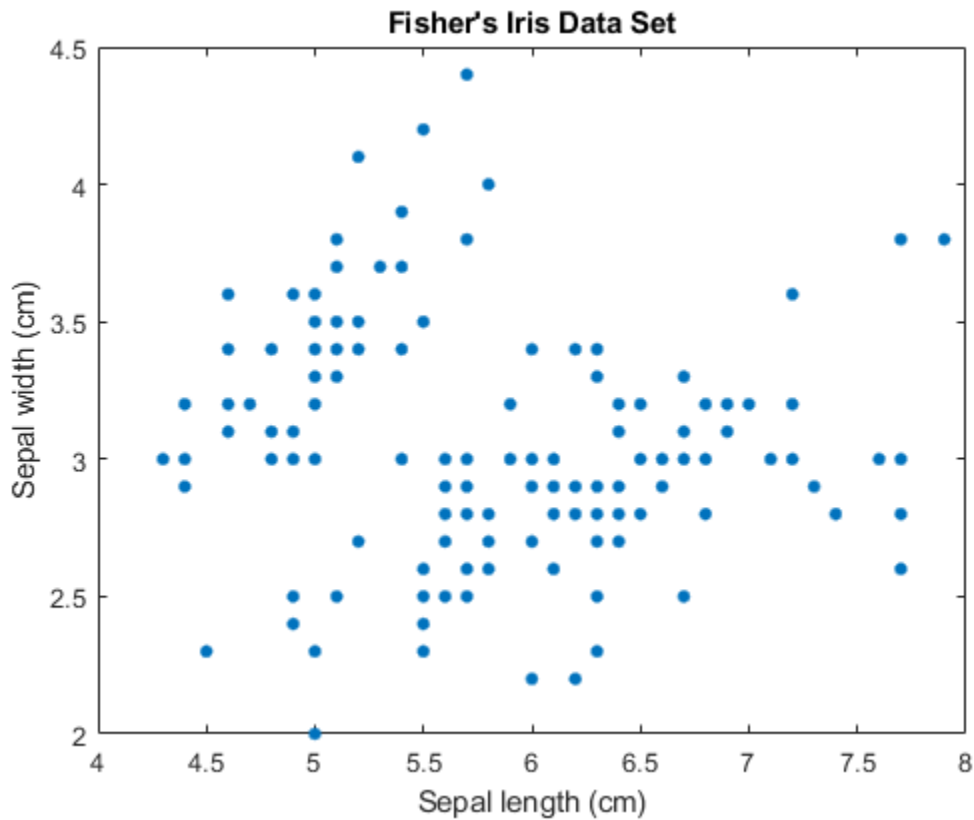
Like most clustering methods, you must specify the number of desired clusters before fitting the model. The number of clusters specifies the number of components in the GMM. For GMMs, it is best practice to also consider the:

- Component covariance structure. You can specify diagonal or full covariance matrices, or whether all components have the same covariance matrix.
- Initial conditions. The Expectation-Maximization (EM) algorithm fits the GMM. Like the *k*-means clustering algorithm, EM is sensitive to initial conditions and might converge to a local optimum. You can specify your own starting values for the parameters, specify initial cluster assignments for data points or let them be randomly chosen, or specify to use the *k*-means ++ algorithm.
- Regularization parameter. If, for example, you have more predictors than data points, then you can regularize for estimation stability.

## Covariance Structure Options

Load Fisher's iris data set. Consider clustering the sepal measurements.

```
load fisheriris;  
X = meas(:,1:2);  
[n,p] = size(X);  
rng(3); % For reproducibility  
  
figure;  
plot(X(:,1),X(:,2),'.','MarkerSize',15);  
title('Fisher's Iris Data Set');  
xlabel('Sepal length (cm)');  
ylabel('Sepal width (cm)');
```



The number of components,  $k$ , in a GMM determines number of subpopulations or clusters. In this figure, it is difficult to determine whether two, three, or perhaps more components are appropriate. A GMM increases in complexity as  $k$  increases.

Each component has a covariance matrix. Geometrically, the covariance structure determines the shape of a confidence ellipsoid drawn over a subpopulation or cluster. You can specify whether the covariance matrices for all components are diagonal or full, or whether all components have the same covariance matrix. Each combination of specifications determines the shape and orientation of the ellipsoids.

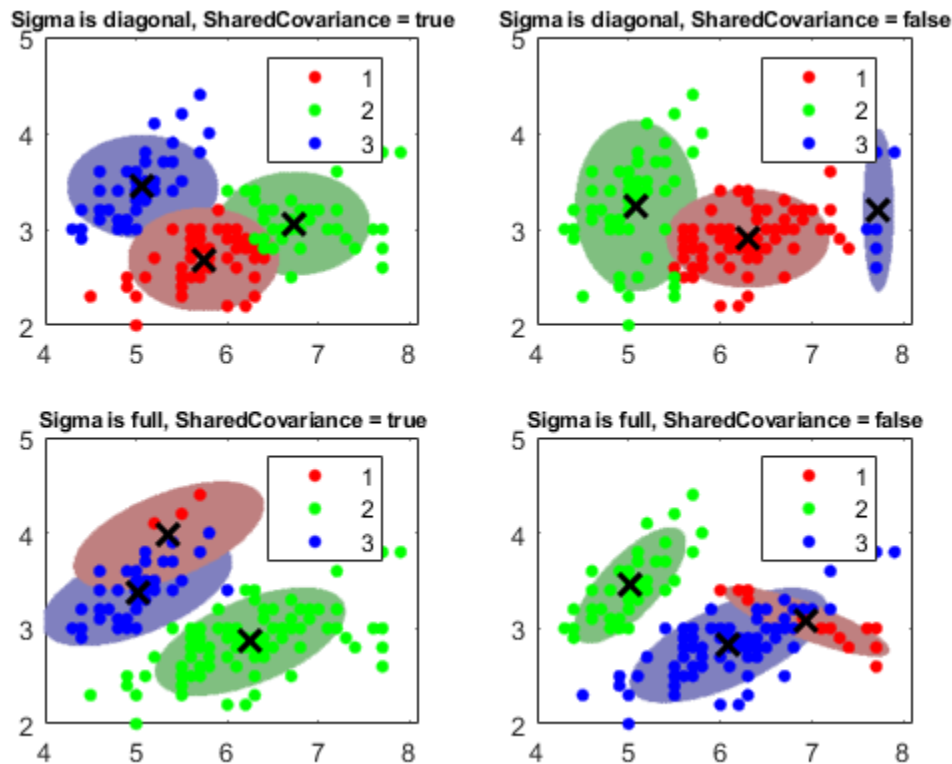
Fit GMMs to the data and examine the effects of specifying all combinations of covariance structure options on the shape of the ellipsoids. That is, specify all combinations of the name-value pair arguments 'CovarianceType' and 'SharedCovariance'. Covariance structure specifications apply to all components. For illustration, specify that there are three components. To draw the ellipsoids:

- 1 Use the fitted GMM to cluster a grid covering the plane composed of the extremes of the measurements.
- 2 Obtain the score that specifies a 99% probability threshold for each confidence region. This specification determines the length of the major and minor axes of the ellipsoids.
- 3 Color the ellipse using a similar color to its cluster.

```
k = 3;
Sigma = {'diagonal','full'};
nSigma = numel(Sigma);
SharedCovariance = {true,false};
SCtext = {'true','false'};
nSC = numel(SharedCovariance);
d = 500;
x1 = linspace(min(X(:,1)) - 2,max(X(:,1)) + 2,d);
x2 = linspace(min(X(:,2)) - 2,max(X(:,2)) + 2,d);
[x1grid,x2grid] = meshgrid(x1,x2);
X0 = [x1grid(:) x2grid(:)];
threshold = sqrt(chi2inv(0.99,2));
options = statset('MaxIter',1000); % Increase number of EM iterations

figure;
c = 1;
for i = 1:nSigma;
    for j = 1:nSC;
        gmfit = fitgmdist(X,k,'CovarianceType',Sigma{i},...
```

```
        'SharedCovariance', SharedCovariance{j}, 'Options', options);
clusterX = cluster(gmfit, X);
mahalDist = mahal(gmfit, X0);
subplot(2, 2, c);
h1 = gscatter(X(:, 1), X(:, 2), clusterX);
hold on;
    for m = 1:k;
        idx = mahalDist(:, m) <= threshold;
        Color = h1(m).Color*0.75 + -0.5*(h1(m).Color - 1);
        h2 = plot(X0(idx, 1), X0(idx, 2), '.', 'Color', Color, 'MarkerSize', 1);
        uistack(h2, 'bottom');
    end
plot(gmfit.mu(:, 1), gmfit.mu(:, 2), 'kx', 'LineWidth', 2, 'MarkerSize', 10)
title(sprintf('Sigma is %s, SharedCovariance = %s', ...
    Sigma{i}, SCText{j}), 'FontSize', 8)
legend(h1, {'1', '2', '3'});
hold off
c = c + 1;
end
end
```



While the probability threshold for the confidence region determines the length of the major and minor axes, the covariance type determines the orientation of the axes.

- Diagonal covariance matrices indicate that the predictors are uncorrelated. The major and minor axes of the ellipses are parallel or perpendicular to the  $x$  and  $y$  axes. This specification increases the total number of parameters by  $p$ , the number of predictors, for each component, but is more parsimonious than the full covariance specification.
- Full covariance matrices allow for correlated predictors. There is no restriction to the orientation of the ellipses relative to the  $x$  and  $y$  axes. Each component increases the total number of parameters by  $p(p+1)/2$ , but captures correlation structure among the predictors. This specification can cause overfitting.



- Shared covariance matrices indicate that all components have the same covariance matrix. All ellipses are the same size and have the same orientation. This specification is more parsimonious than the unshared specification because the total number of parameters only increases by the number of covariance parameters for one component.
- Unshared covariance matrices indicate that all components have their own covariance matrix. The size and orientation of all ellipses might differ. This specification increases the number of parameters by  $k$  times the number of covariance parameters for a component, but can capture covariance differences among components.

The figure also shows that `cluster` does not always preserve cluster order. That is, if you cluster several fitted `gmdistribution` models, `cluster` might assign different cluster labels for similar components.

In most applications, the number of components,  $k$ , and appropriate covariance structure,  $\Sigma$ , are unknown. One way you can tune a GMM is by comparing information criteria. Two popular information criteria are Akaike's Information Criterion (AIC) and the Bayesian Information Criterion (BIC). Both take the optimized, negative log likelihood, and then penalize it with the number of parameters in the model (i.e., the model complexity). However, the BIC penalizes for complexity more severely than AIC. Therefore, the AIC tends to choose more complex models that might overfit, and BIC tends to choose simpler models that might underfit. A good practice is to look at both criteria when deciding on a model. Lower AIC or BIC values indicate better fitting models. You should also ensure that your choices for  $k$  and the covariance matrix structure is appropriate for your application. `fitgmdist` stores the AIC and BIC of fitted `gmdistribution` model objects in the properties `AIC` and `BIC`. You can access them using dot notation. For an example on choosing the appropriate parameters, see “Tune Gaussian Mixture Models” on page 16-55.

### Effects of Initial Conditions

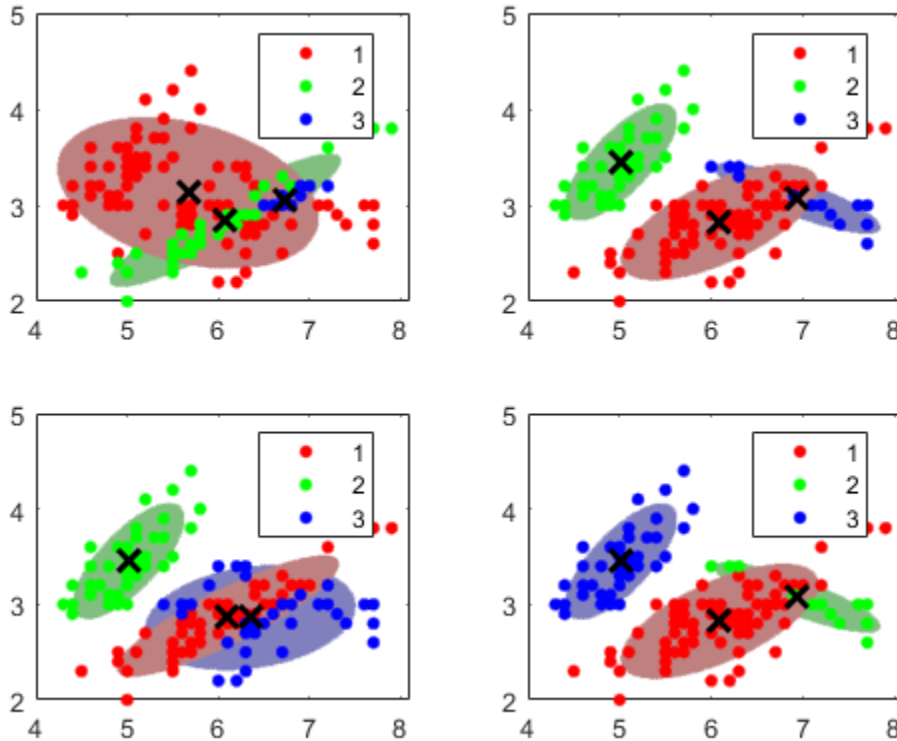
The algorithm that fits a GMM to the data can be sensitive to initial conditions. To illustrate this, consider fitting several GMMs and specify different initial conditions. Specifically, specify that, initially, most data points belong to the first cluster, two sets of random, initial assignments, and use `k-means ++` to obtain initial cluster centers. For all instances, specify three components, unshared and full covariance matrices, the same initial mixture proportions, and the same initial covariance matrices. For stability when you try different sets of initial values, increase the number of EM algorithm iterations.

```
cluster0 = {[ones(n-8,1); [2; 2; 2; 2]; [3; 3; 3; 3]]; ...
            randsample(1:k,n,true); randsample(1:k,n,true); 'plus'};
```

```
converged = nan(4,1);
figure;
for j = 1:4;
    gmfit = fitgmdist(X,k,'CovarianceType','full',...
        'SharedCovariance',false,'Start',cluster0{j},...
        'Options',options);
    clusterX = cluster(gmfit,X);
    mahalDist = mahal(gmfit,X0);
    subplot(2,2,j);
    h1 = gscatter(X(:,1),X(:,2),clusterX);
    hold on;
    nK = numel(unique(clusterX));
    for m = 1:nK;
        idx = mahalDist(:,m)<=threshold;
        Color = h1(m).Color*0.75 + -0.5*(h1(m).Color - 1);
        h2 = plot(X0(idx,1),X0(idx,2),'.','Color',Color,'MarkerSize',1);
        uistack(h2,'bottom');
    end
    plot(gmfit.mu(:,1),gmfit.mu(:,2),'kx','LineWidth',2,'MarkerSize',10)
    legend(h1,{'1','2','3'});
    hold off
    converged(j) = gmfit.Converged;
end
sum(converged)
```

```
ans =
```

```
4
```



All algorithms converged. Each of the different sets of starting cluster assignments for the data points leads to a different, fitted cluster assignment. You can specify a positive integer for the name-value pair argument 'Replicates', which runs the algorithm the specified number of times. Subsequently, `fitgmdist` chooses the fit that yields the largest likelihood.

### When to Regularize

Sometimes, during an EM iteration, a fitted covariance matrix can become ill conditioned, that is, the likelihood is escaping to infinity. This can happen if:

- There are more predictors than data points.

- You specify to fit with too many components.
- Variables are highly correlated.

To overcome this problem, you can specify a small, positive number using the 'Regularize' name-value pair argument. `fitgmdist` adds the specified small, positive number to the diagonal elements of all covariance matrices, which ensures all matrices are positive definite. Regularizing can reduce the maximal likelihood value. For more details, see `fitgmdist`.

## See Also

`cluster` | `fitgmdist` | `gmdistribution`

## Related Examples

- “Cluster Data from Mixture of Gaussian Distributions” on page 16-37
- “Cluster Gaussian Mixture Data Using Soft Clustering” on page 16-47
- “Tune Gaussian Mixture Models” on page 16-55

## More About

- “Gaussian Mixture Models” on page 5-148

## Cluster Data from Mixture of Gaussian Distributions

This example shows how to cluster simulated data from a mixture of Gaussian distributions, and how to work with `gmdistribution` objects.

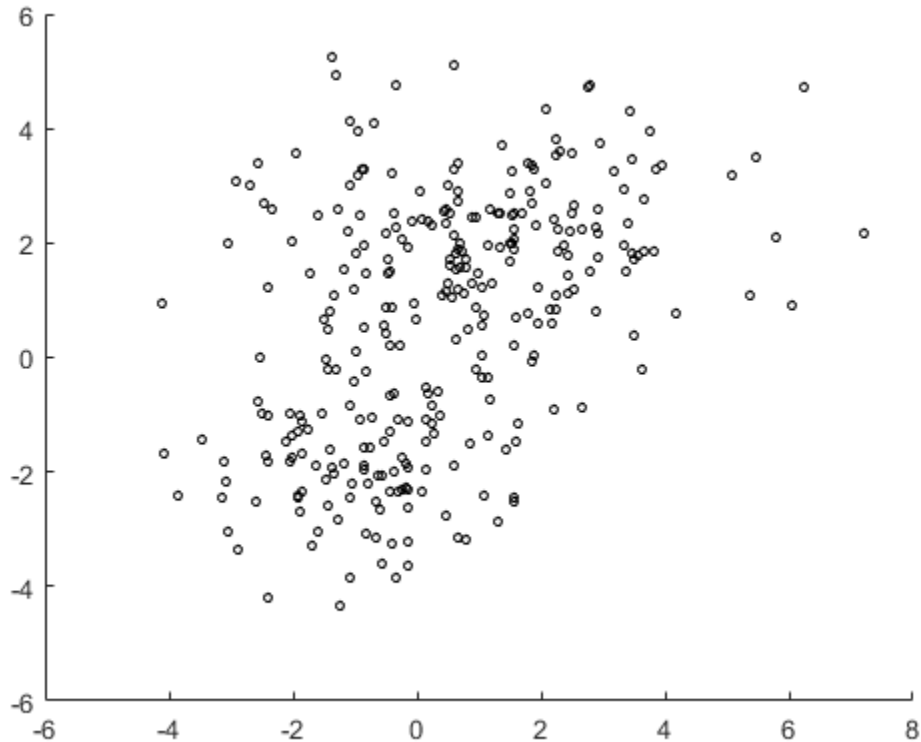
Gaussian mixture models can be used for clustering data, by realizing that the multivariate normal components of the fitted model can represent clusters.

### Simulate Data from a Mixture of Gaussian Distributions

Simulate data from a mixture of two bivariate Gaussian distributions using `mvnrnd`.

```
rng default; % For reproducibility
mu1 = [1 2];
sigma1 = [3 .2; .2 2];
mu2 = [-1 -2];
sigma2 = [2 0; 0 1];
X = [mvnrnd(mu1, sigma1, 200); mvnrnd(mu2, sigma2, 100)];
n = size(X, 1);

figure;
scatter(X(:, 1), X(:, 2), 10, 'ko')
```



### Fit the Simulated Data to a Gaussian Mixture Model

Fit a two-component Gaussian mixture model (GMM). Here, you know the correct number of components to use. In practice, with real data, this decision would require comparing models with different numbers of components. Also, request to display the final iteration of the expectation-maximization fitting routine.

```
options = statset('Display','final');  
gm = fitgmdist(X,2,'Options',options)
```

```
26 iterations, log-likelihood = -1210.59
```

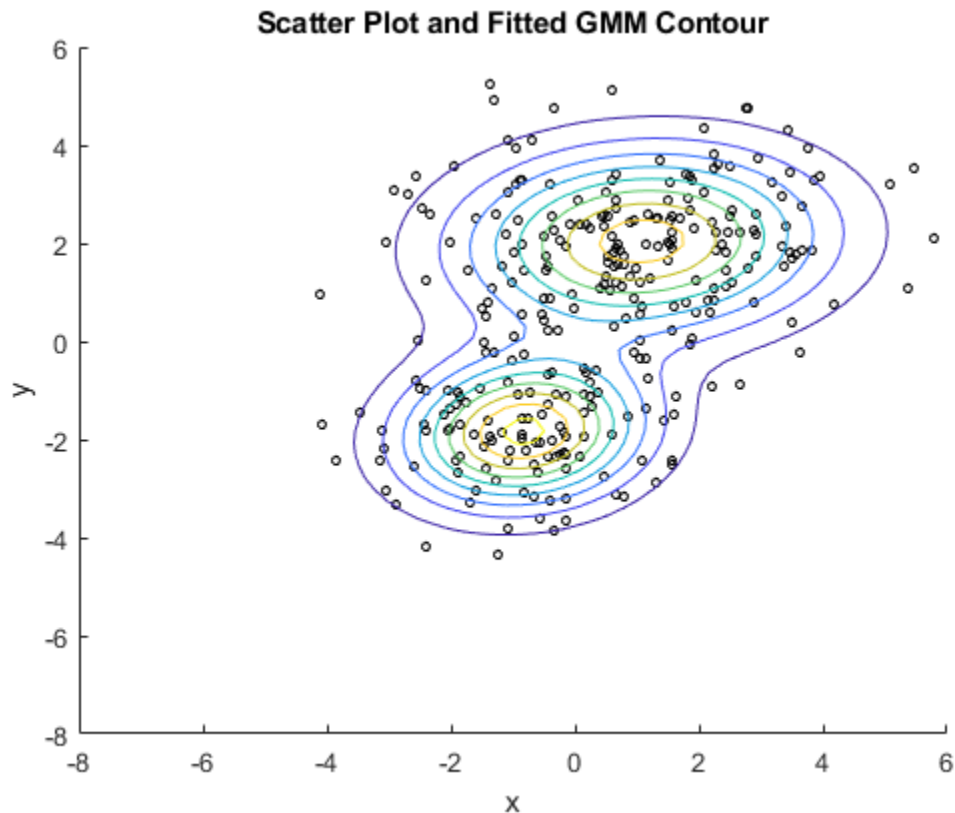
```
gm =
```

```
Gaussian mixture distribution with 2 components in 2 dimensions
Component 1:
Mixing proportion: 0.629514
Mean:    1.0756    2.0421

Component 2:
Mixing proportion: 0.370486
Mean:    -0.8296   -1.8488
```

Plot the estimated probability density contours for the two-component mixture distribution. The two bivariate normal components overlap, but their peaks are distinct. This suggests that the data could reasonably be divided into two clusters.

```
hold on
ezcontour(@(x,y)pdf(gm,[x y]),[-8 6],[-8 6]);
title('Scatter Plot and Fitted GMM Contour')
hold off
```



### Cluster the Data Using the Fitted GMM

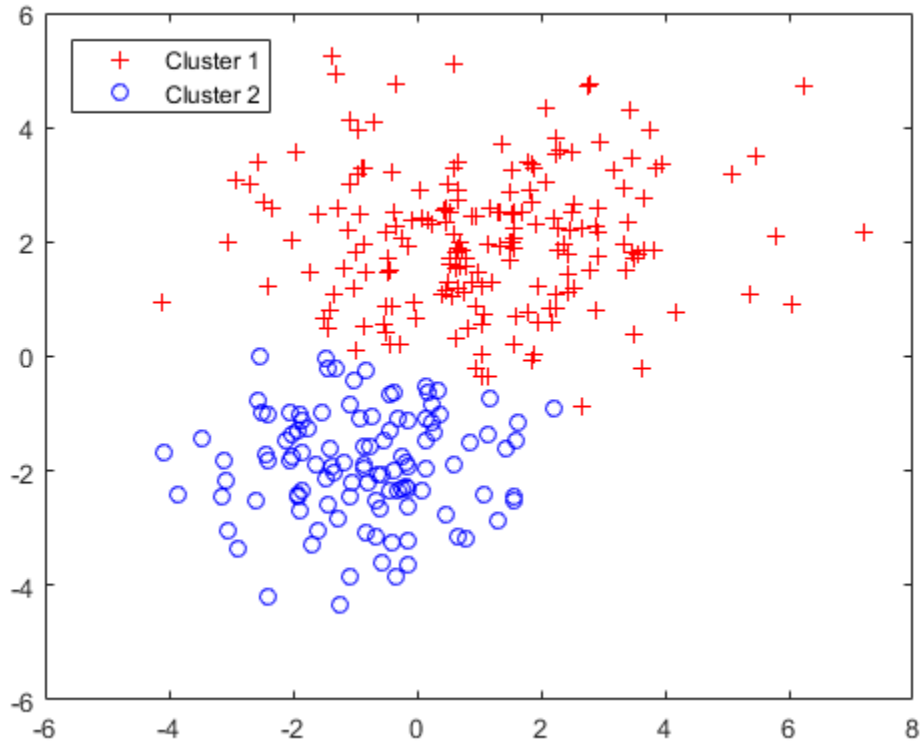
`gmdistribution.cluster` implements "hard clustering", a method that assigns each data point to exactly one cluster. For GMM, `cluster` assigns each point to one of the two mixture components in the GMM. The center of each cluster is the corresponding mixture component mean. For details on "soft clustering," see "Cluster Gaussian Mixture Data Using Soft Clustering" on page 16-47.

Partition the data into clusters by passing the fitted GMM and the data to `cluster`.

```
idx = cluster(gm,X);  
cluster1 = (idx == 1); % |1| for cluster 1 membership  
cluster2 = (idx == 2); % |2| for cluster 2 membership
```



```
figure;  
gscatter(X(:,1),X(:,2),idx,'rb','+o');  
legend('Cluster 1','Cluster 2','Location','NorthWest');
```



Each cluster corresponds to one of the bivariate normal components in the mixture distribution. `cluster` assigns data to clusters based on a cluster membership score. Each cluster membership score is the estimated posterior probability that the data point came from the corresponding component. `cluster` assigns each point to the mixture component corresponding to the highest posterior probability.

You can estimate cluster membership posterior probabilities by passing the fitted GMM and data to either:

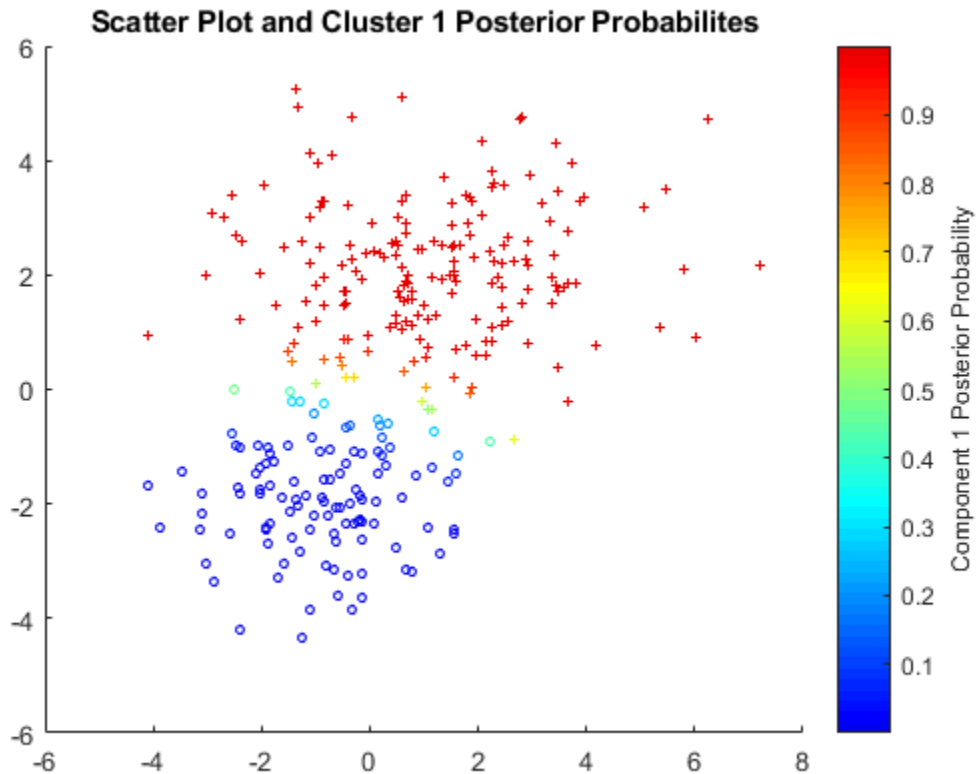
- `gmdistribution.posterior`
- `cluster`, and request to return the third output argument

### Estimate Cluster Membership Posterior Probabilities

Estimate and plot the posterior probability of the first component for each point.

```
P = posterior(gm,X);

figure;
scatter(X(cluster1,1),X(cluster1,2),10,P(cluster1,1),'+')
hold on
scatter(X(cluster2,1),X(cluster2,2),10,P(cluster2,1),'o')
hold off
clrmap = jet(80);
colormap(clrmap(9:72,:))
ylabel(colorbar,'Component 1 Posterior Probability')
title('Scatter Plot and Cluster 1 Posterior Probabilites')
```



$P$  is an  $n$ -by-2 matrix of cluster membership posterior probabilities. The first column contains the probabilities for cluster 1 and the second column corresponds to cluster 2.

### Assign New Data to Clusters

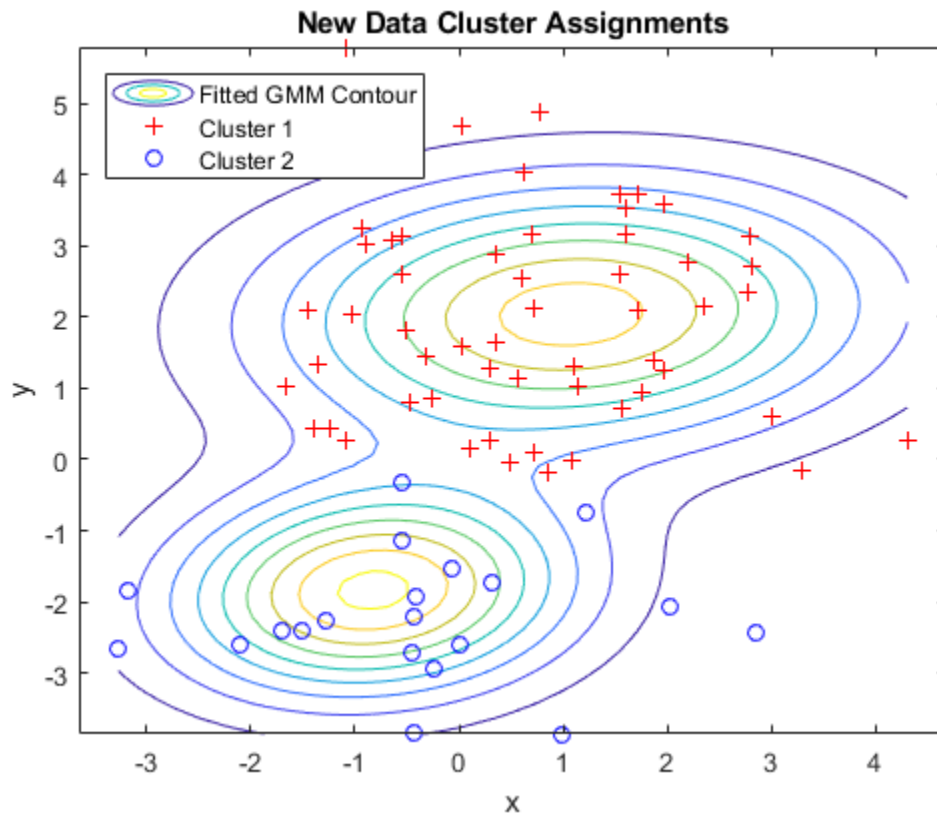
You can also use the `cluster` method to assign new data points to the mixture components found in the original data.

Simulate new data from a mixture of Gaussian distributions. Rather than using `mvnrnd`, you can create a GMM with the true mixture component means and standard deviations using `gmdistribution`, and then pass the GMM to `gmdistribution.random` to simulate data.

```
Mu = [mu1; mu2];  
Sigma = cat(3, sigma1, sigma2);  
p = [0.75 0.25]; % Mixing proportions  
  
gmTrue = gmdistribution(Mu, Sigma, p);  
X0 = random(gmTrue, 75);
```

Assign clusters to the new data by pass the fitted GMM (gm) and the new data to cluster. Request cluster membership posterior probabilities.

```
[idx0,~,P0] = cluster(gm,X0);  
  
figure;  
ezcontour(@(x,y)pdf(gm,[x y]),[min(X0(:,1)) max(X0(:,1))],...  
    [min(X0(:,2)) max(X0(:,2))]);  
hold on;  
gscatter(X0(:,1),X0(:,2),idx0,'rb','+o');  
legend('Fitted GMM Contour','Cluster 1','Cluster 2','Location','NorthWest');  
title('New Data Cluster Assignments')  
hold off;
```



For `cluster` to provide meaningful results when clustering new data, `X0` should come from the same population as `X`, the original data used to create the mixture distribution. In particular, when computing the posterior probabilities for `X0`, `cluster` and `posterior` use the estimated mixing probabilities.

## See Also

`cluster` | `fitgmdist` | `gmdistribution` | `posterior` | `random`

## Related Examples

- “Cluster Gaussian Mixture Data Using Soft Clustering” on page 16-47

- “Tune Gaussian Mixture Models” on page 16-55

### **More About**

- “Gaussian Mixture Models” on page 5-148
- “Clustering Using Gaussian Mixture Models” on page 16-28

## Cluster Gaussian Mixture Data Using Soft Clustering

This example shows how to implement soft clustering on simulated data from a mixture of Gaussian distributions.

`gmdistribution.cluster` estimates cluster membership posterior probabilities, and then assigns each point to the cluster corresponding to the maximum posterior probability. Soft clustering is an alternative clustering method that allows some data points to belong to multiple clusters. To implement soft clustering:

- 1 Assign a cluster membership score to each data point that describes how similar each point is to each cluster's archetype. For a mixture of Gaussian distributions, the cluster archetype is corresponding component mean, and the component can be the estimated cluster membership posterior probability.
- 2 Rank the points by their cluster membership score.
- 3 Inspect the scores and determine cluster memberships.

For algorithms that use posterior probabilities as scores, a data point is a member of the cluster corresponding to the maximum posterior probability. However, if there are other clusters with corresponding posterior probabilities that are close to the maximum, then the data point can also be a member of those clusters. It is good practice to determine the threshold on scores that yield multiple cluster memberships before clustering.

This example follows from “Cluster Data from Mixture of Gaussian Distributions” on page 16-37.

Simulate data from a mixture of two bivariate Gaussian distributions.

```
rng(0, 'twister') % For reproducibility
mu1 = [1 2];
sigma1 = [3 .2; .2 2];
mu2 = [-1 -2];
sigma2 = [2 0; 0 1];
X = [mvnrnd(mu1, sigma1, 200); mvnrnd(mu2, sigma2, 100)];
```

Fit a two-component Gaussian mixture model (GMM). Because there are two components, suppose that any data point with cluster membership posterior probabilities in the interval  $[0.4, 0.6]$  can be a member of both clusters.

```
gm = fitgmdist(X, 2);
threshold = [0.4 0.6];
```

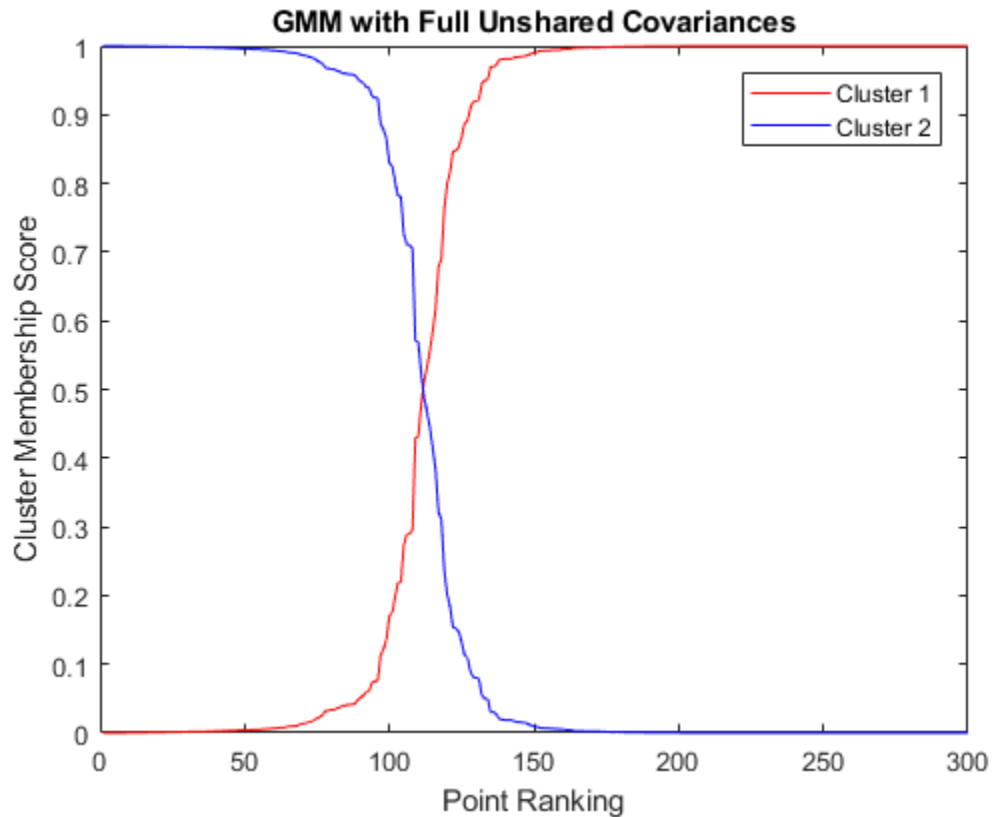
Estimate component-member posterior probabilities for all data points using the fitted GMM `gm`. These represent cluster membership scores.

```
P = posterior(gm,X);
```

For each cluster, rank the membership scores for all data points. For each cluster, plot each data points membership score with respect to its ranking relative to all other data points.

```
n = size(X,1);  
[~,order] = sort(P(:,1));  
  
figure  
plot(1:n,P(order,1), 'r-', 1:n,P(order,2), 'b-')  
legend({'Cluster 1', 'Cluster 2'})  
ylabel('Cluster Membership Score')  
xlabel('Point Ranking')  
title('GMM with Full Unshared Covariances')
```





Although a clear separation is hard to see in a scatter plot of the data, plotting the membership scores indicates that the fitted distribution does a good job of separating the data into groups.

Plot the data and assign clusters by maximum posterior probability. Identify points that could be in either cluster.

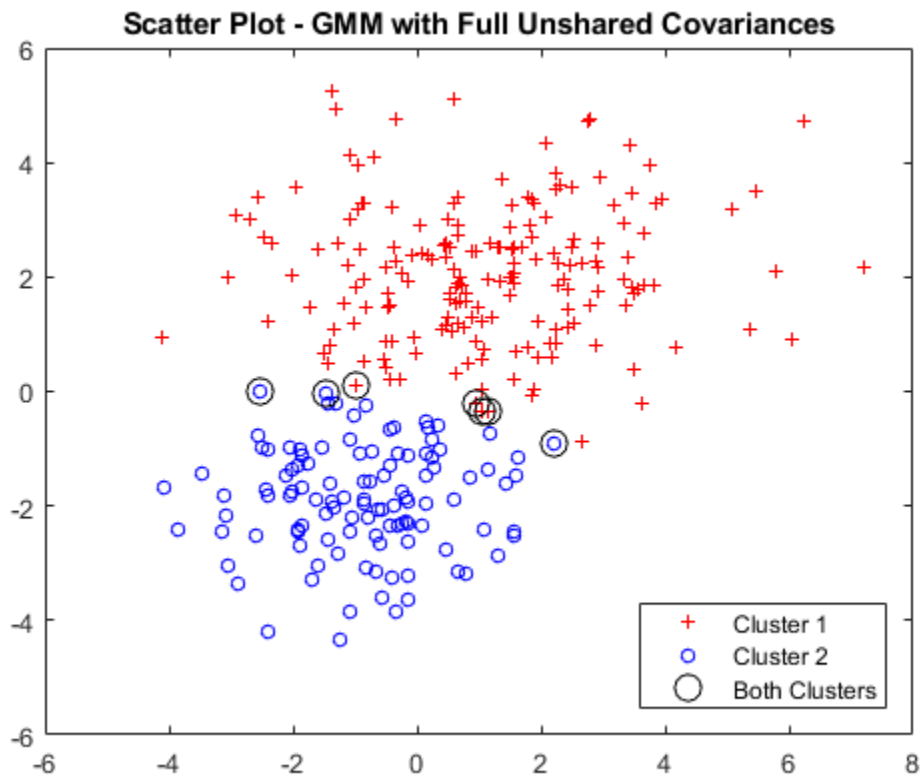
```
idx = cluster(gm,X);
idxBoth = find(P(:,1)>=threshold(1) & P(:,1)<=threshold(2));
numInBoth = numel(idxBoth)
```

```
figure
gscatter(X(:,1),X(:,2),idx,'rb','+o',5)
hold on
```

```
plot(X(idxBoth,1),X(idxBoth,2),'ko','MarkerSize',10)
legend({'Cluster 1','Cluster 2','Both Clusters'},'Location','SouthEast')
title('Scatter Plot - GMM with Full Unshared Covariances')
hold off
```

```
numInBoth =
```

```
7
```



Using the score threshold interval, seven data points can be in either cluster.

Soft clustering using a GMM is similar to fuzzy  $k$ -means clustering, which also assigns each point to each cluster with a membership score. The fuzzy  $k$ -means algorithm assumes that clusters are roughly spherical in shape, and all of roughly equal size. This is comparable to a Gaussian mixture distribution with a single covariance matrix that is shared across all components, and is a multiple of the identity matrix. In contrast, `gmdistribution` allows you to specify different covariance structures. The default is to estimate a separate, unconstrained covariance matrix for each component. A more restricted option, closer to  $k$ -means, is to estimate a shared, diagonal covariance matrix.

Fit a GMM to the data, but specify that the components share the same, diagonal covariance matrix. This specification is similar to implementing fuzzy  $k$ -means clustering, but provides more flexibility by allowing unequal variances for different variables.

```
gmSharedDiag = fitgmdist(X,2,'CovType','Diagonal','SharedCovariance',true');
```

Estimate component-member posterior probabilities for all data points using the fitted GMM `gmSharedDiag`. Estimate soft cluster assignments.

```
[idxSharedDiag,~,PSharedDiag] = cluster(gmSharedDiag,X);
idxBothSharedDiag = find(PSharedDiag(:,1)>=threshold(1) & ...
    PSharedDiag(:,1)<=threshold(2));
numInBoth = numel(idxBothSharedDiag)
```

```
numInBoth =
```

```
5
```

Assuming shared, diagonal covariances among components, five data points could be in either cluster.

For each cluster:

- 1 Rank the membership scores for all data points.
- 2 Plot each data points membership score with respect to its ranking relative to all other data points.

```
[~,orderSharedDiag] = sort(PSharedDiag(:,1));
```

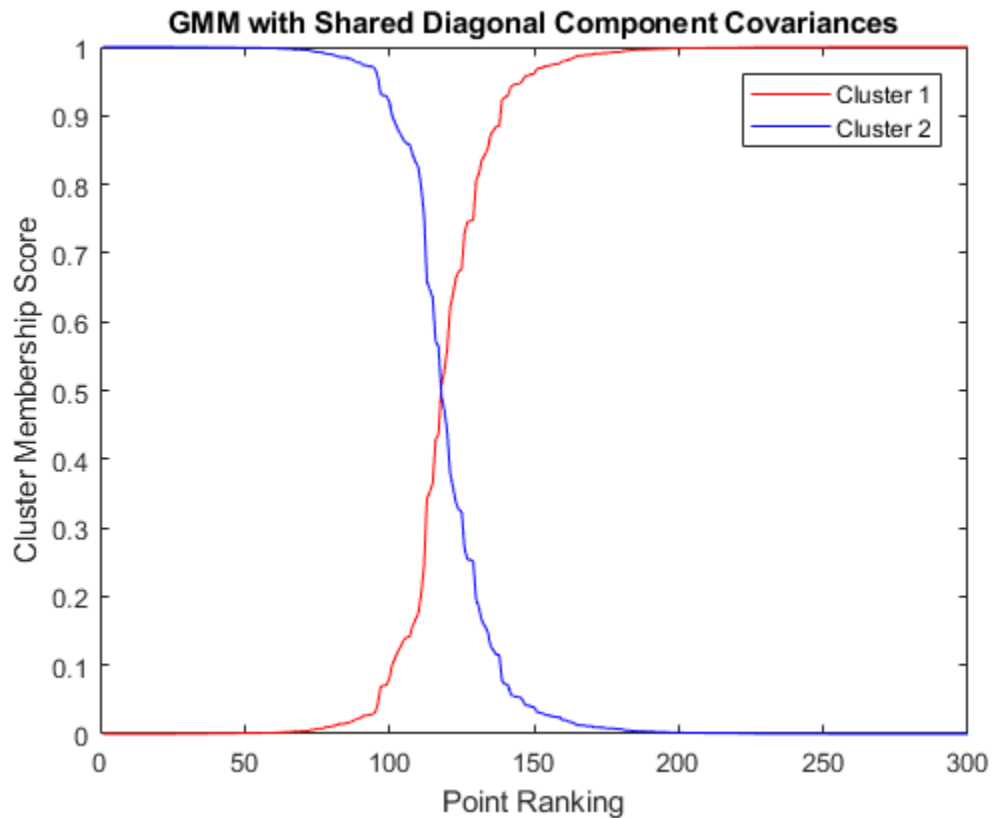
```
figure
```

```
plot(1:n,PSharedDiag(orderSharedDiag,1),'r',...)
```

```

1:n,PSharedDiag(orderSharedDiag,2),'b-')
legend({'Cluster 1' 'Cluster 2'},'Location','NorthEast')
ylabel('Cluster Membership Score')
xlabel('Point Ranking')
title('GMM with Shared Diagonal Component Covariances')

```



Plot the data and identify the hard, clustering assignments from the GMM analysis assuming the shared, diagonal covariances among components. Also, identify those data points that could be in either cluster.

```

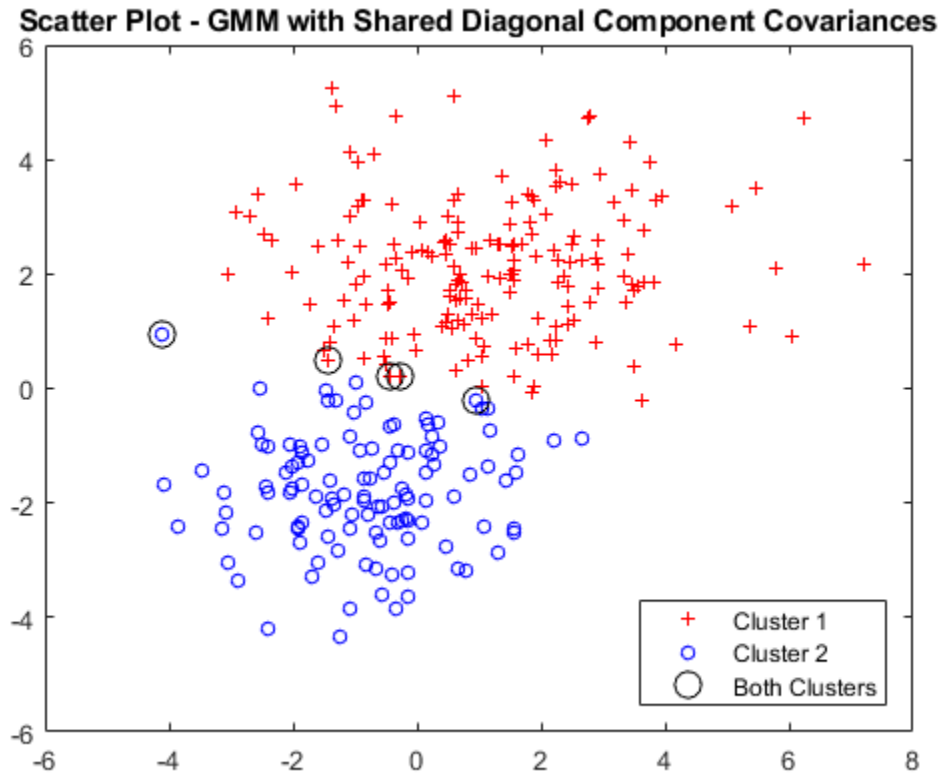
figure
gscatter(X(:,1),X(:,2),idxSharedDiag,'rb','+o',5)
hold on
plot(X(idxBothSharedDiag,1),X(idxBothSharedDiag,2),'ko','MarkerSize',10)

```

```

legend({'Cluster 1', 'Cluster 2', 'Both Clusters'}, 'Location', 'SouthEast')
title('Scatter Plot - GMM with Shared Diagonal Component Covariances')
hold off

```



## See Also

`cluster` | `fitgmdist` | `gmdistribution`

## Related Examples

- “Cluster Data from Mixture of Gaussian Distributions” on page 16-37
- “Tune Gaussian Mixture Models” on page 16-55

## **More About**

- “Gaussian Mixture Models” on page 5-148
- “Clustering Using Gaussian Mixture Models” on page 16-28

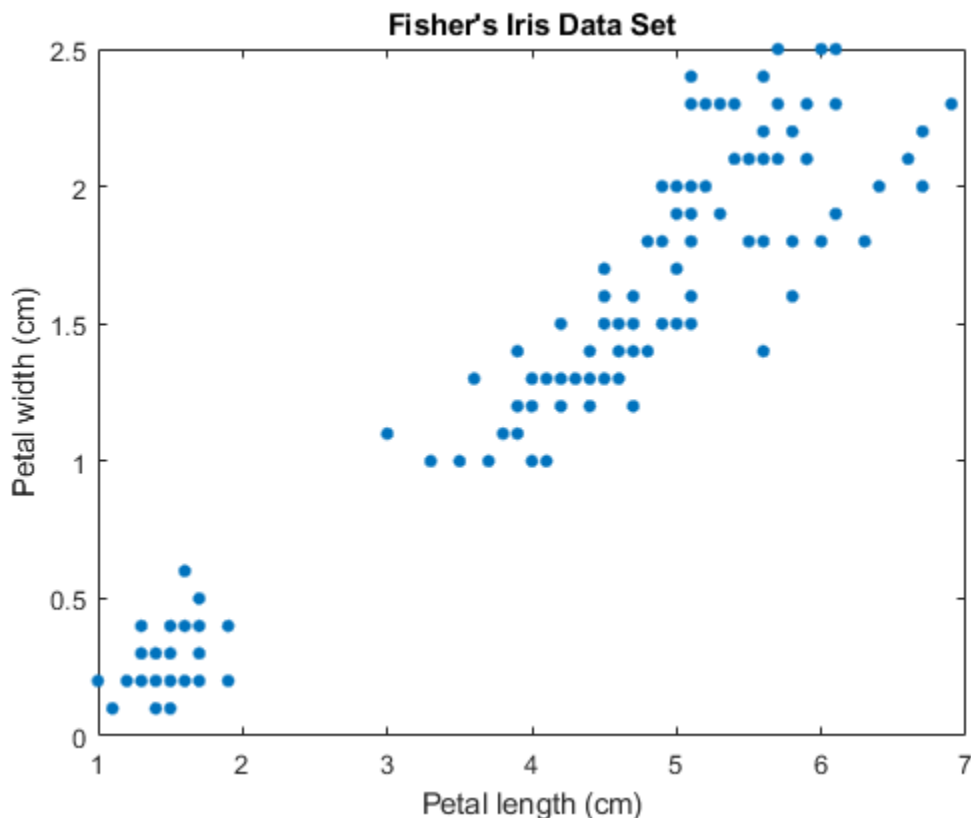
## Tune Gaussian Mixture Models

This example shows how to determine the best Gaussian mixture model (GMM) fit by adjusting the number of components and the component covariance matrix structure.

Load Fisher's iris data set. Consider the petal measurements as predictors.

```
load fisheriris;
X = meas(:,3:4);
[n,p] = size(X);
rng(1); % For reproducibility

figure;
plot(X(:,1),X(:,2),'.','MarkerSize',15);
title('Fisher's Iris Data Set');
xlabel('Petal length (cm)');
ylabel('Petal width (cm)');
```



Suppose  $k$  is the number of desired components or clusters, and  $\Sigma$  is the covariance structure for all components. Follow these steps to tune a GMM.

- 1 Choose a  $(k, \Sigma)$  pair, and then fit a GMM using the chosen parameter specification and the entire data set.
- 2 Estimate the AIC and BIC.
- 3 Repeat steps 1 and 2 until you exhaust all  $(k, \Sigma)$  pairs of interest.
- 4 Choose the fitted GMM that balances low AIC with simplicity.

For this example, choose a grid of values for  $k$  that include 2 and 3, and some surrounding numbers. Specify all available choices for covariance structure. If  $k$  is too high for the data set, then the estimated component covariances can be badly



conditioned. Specify to use regularization to avoid badly conditioned covariance matrices. Increase the number of EM algorithm iterations to 10000.

```
k = 1:5;
nK = numel(k);
Sigma = {'diagonal','full'};
nSigma = numel(Sigma);
SharedCovariance = {true,false};
SCText = {'true','false'};
nSC = numel(SharedCovariance);
RegularizationValue = 0.01;
options = statset('MaxIter',10000);
```

Fit the GMMs using all parameter combination. Compute the AIC and BIC for each fit. Track the terminal convergence status of each fit.

```
% Preallocation
gm = cell(nK,nSigma,nSC);
aic = zeros(nK,nSigma,nSC);
bic = zeros(nK,nSigma,nSC);
converged = false(nK,nSigma,nSC);

% Fit all models
for m = 1:nSC;
    for j = 1:nSigma;
        for i = 1:nK;
            gm{i,j,m} = fitgmdist(X,k(i),...
                'CovarianceType',Sigma{j},...
                'SharedCovariance',SharedCovariance{m},...
                'RegularizationValue',RegularizationValue,...
                'Options',options);
            aic(i,j,m) = gm{i,j,m}.AIC;
            bic(i,j,m) = gm{i,j,m}.BIC;
            converged(i,j,m) = gm{i,j,m}.Converged;
        end
    end
end

allConverge = (sum(converged(:)) == nK*nSigma*nSC)

allConverge =

    logical
```

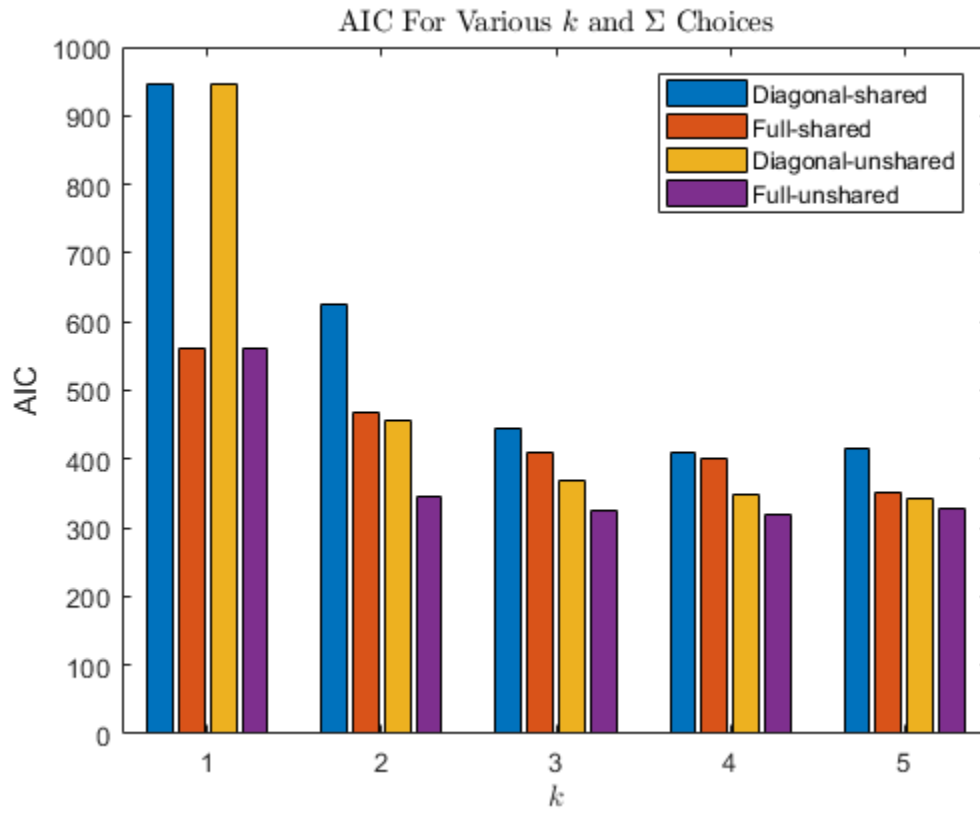
1

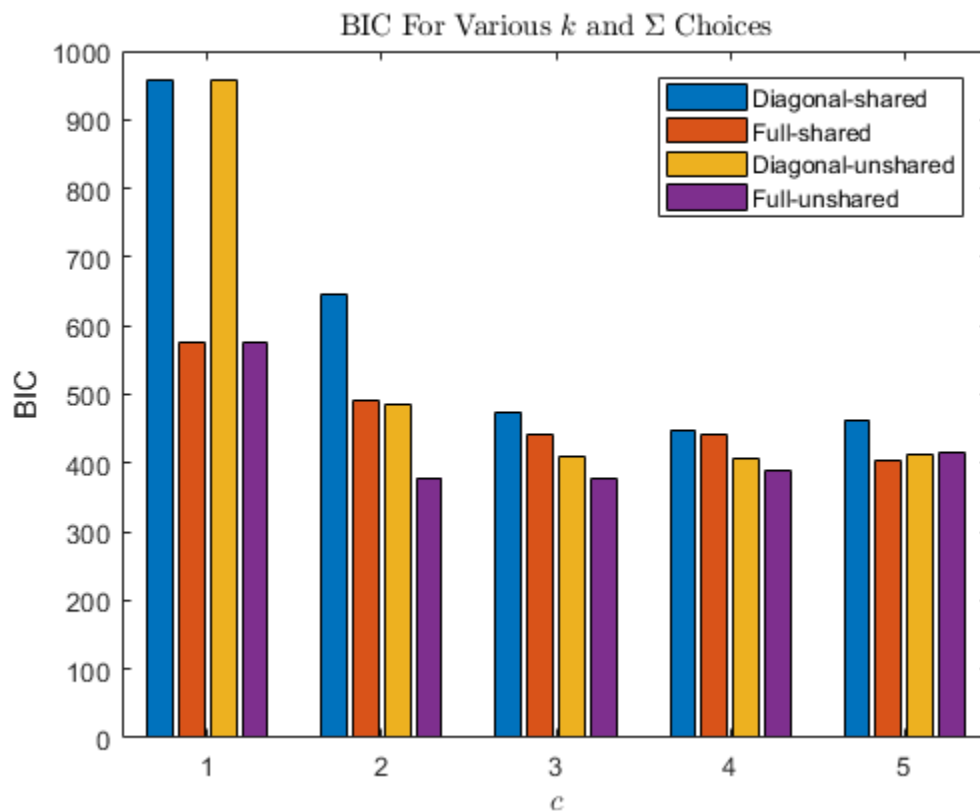
`gm` is a cell array containing all of the fitted `gmdistribution` model objects. All of the fitting instances converged.

Plot separate bar charts to compare the AIC and BIC among all fits. Group the bars by  $k$ .

```
figure;
bar(reshape(aic,nK,nSigma*nSC));
title('AIC For Various  $k$  and  $\Sigma$  Choices','Interpreter','latex');
xlabel(' $k$ ','Interpreter','Latex');
ylabel('AIC');
legend({'Diagonal-shared','Full-shared','Diagonal-unshared',...
       'Full-unshared'});

figure;
bar(reshape(bic,nK,nSigma*nSC));
title('BIC For Various  $k$  and  $\Sigma$  Choices','Interpreter','latex');
xlabel(' $k$ ','Interpreter','Latex');
ylabel('BIC');
legend({'Diagonal-shared','Full-shared','Diagonal-unshared',...
       'Full-unshared'});
```





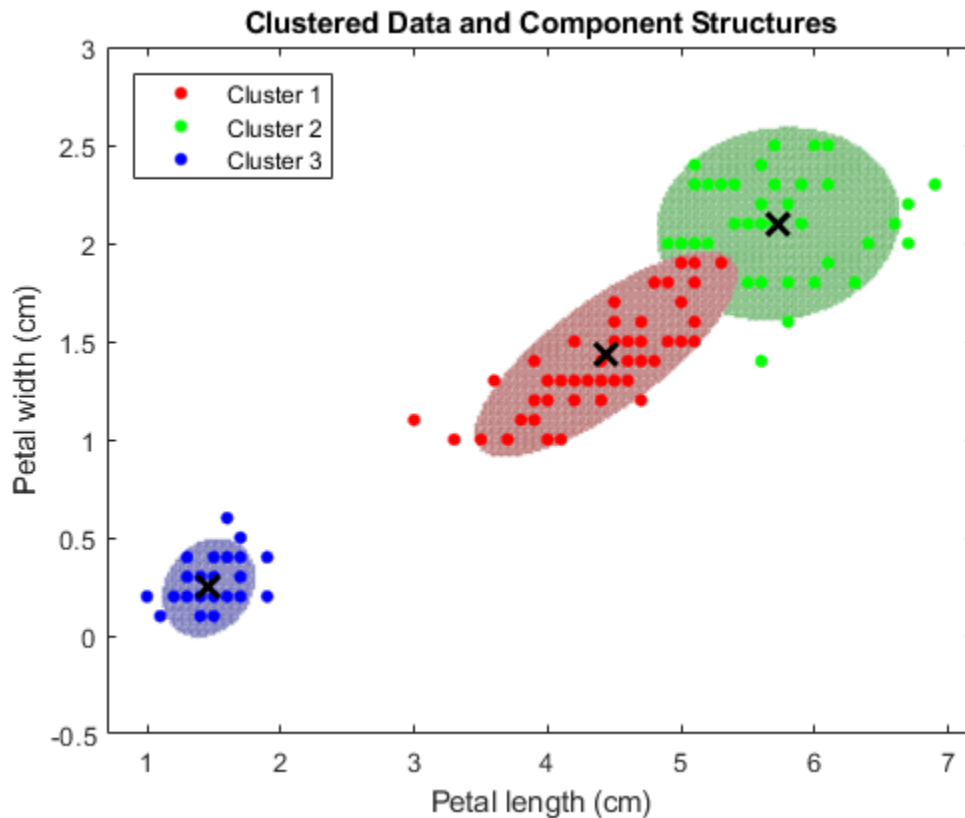
According to the AIC and BIC values, the best model has 3 components and a full, unshared covariance matrix structure.

Cluster the training data using the best fitting model. Plot the clustered data and the component ellipses.

```
gmBest = gm(3,2,2);
clusterX = cluster(gmBest,X);
kGMM = gmBest.NumComponents;
d = 500;
x1 = linspace(min(X(:,1)) - 2,max(X(:,1)) + 2,d);
x2 = linspace(min(X(:,2)) - 2,max(X(:,2)) + 2,d);
[x1grid,x2grid] = meshgrid(x1,x2);
X0 = [x1grid(:) x2grid(:)];
```

```
mahalDist = mahal(gmBest,X0);
threshold = sqrt(chi2inv(0.99,2));

figure;
h1 = gscatter(X(:,1),X(:,2),clusterX);
hold on;
for j = 1:kGMM;
    idx = mahalDist(:,j)<=threshold;
    Color = h1(j).Color*0.75 + -0.5*(h1(j).Color - 1);
    h2 = plot(X0(idx,1),X0(idx,2),'.','Color',Color,'MarkerSize',1);
    uistack(h2,'bottom');
end
h3 = plot(gmBest.mu(:,1),gmBest.mu(:,2),'kx','LineWidth',2,'MarkerSize',10);
title('Clustered Data and Component Structures');
xlabel('Petal length (cm)');
ylabel('Petal width (cm)');
legend(h1,'Cluster 1','Cluster 2','Cluster 3','Location','NorthWest');
hold off
```



This data set includes labels. Determine how well `gmBest` clusters the data by comparing each prediction to the true labels.

```
species = categorical(species);
Y = zeros(n,1);
Y(species == 'versicolor') = 1;
Y(species == 'virginica') = 2;
Y(species == 'setosa') = 3;

miscluster = Y ~= clusterX;
clusterError = sum(miscluster)/n

clusterError =
```

0.0800

The best fitting GMM groups 8% of the observations into the wrong cluster.

`cluster` does not always preserve cluster order. That is, if you cluster several fitted `gmdistribution` models, `cluster` might assign different cluster labels for similar components.

## See Also

`cluster` | `fitgmdist` | `gmdistribution`

## Related Examples

- “Cluster Data from Mixture of Gaussian Distributions” on page 16-37
- “Cluster Gaussian Mixture Data Using Soft Clustering” on page 16-47

## More About

- “Gaussian Mixture Models” on page 5-148
- “Clustering Using Gaussian Mixture Models” on page 16-28





# Parametric Classification

---

- “Parametric Classification” on page 17-2
- “Performance Curves” on page 17-3

## Parametric Classification

Models of data with a categorical response are called classifiers. A classifier is built from training data, for which classifications are known. The classifier assigns new test data to one of the categorical levels of the response.

Parametric methods, like “Discriminant Analysis Classification” on page 20-2, fit a parametric model to the training data and interpolate to classify test data.

Nonparametric methods, like classification and regression trees, use other means to determine classifications.

### See Also

`fitcdiscr` | `fitcnb`

### Related Examples

- “Discriminant Analysis Classification” on page 20-2
- “Naive Bayes Classification” on page 21-2

# Performance Curves

**In this section...**

“Introduction to Performance Curves” on page 17-3

“What are ROC Curves?” on page 17-3

“Evaluate Classifier Performance Using `perfcurve`” on page 17-3

## Introduction to Performance Curves

After a classification algorithm such as `NaiveBayes` or `TreeBagger` has trained on data, you may want to examine the performance of the algorithm on a specific test dataset. One common way of doing this would be to compute a gross measure of performance such as quadratic loss or accuracy, averaged over the entire test dataset.

## What are ROC Curves?

You may want to inspect the classifier performance more closely, for example, by plotting a Receiver Operating Characteristic (ROC) curve. By definition, a ROC curve [1,2] shows true positive rate versus false positive rate (equivalently, sensitivity versus  $1 - \text{specificity}$ ) for different thresholds of the classifier output. You can use it, for example, to find the threshold that maximizes the classification accuracy or to assess, in more broad terms, how the classifier performs in the regions of high sensitivity and high specificity.

## Evaluate Classifier Performance Using `perfcurve`

`perfcurve` computes measures for a plot of classifier performance. You can use this utility to evaluate classifier performance on test data after you train the classifier. Various measures such as mean squared error, classification error, or exponential loss can summarize the predictive power of a classifier in a single number. However, a performance curve offers more information as it lets you explore the classifier performance across a range of thresholds on its output.

You can use `perfcurve` with any classifier or, more broadly, with any method that returns a numeric score for an instance of input data. By convention adopted here,

- A high score returned by a classifier for any given instance signifies that the instance is likely from the positive class.

- A low score signifies that the instance is likely from the negative classes.

For some classifiers, you can interpret the score as the posterior probability of observing an instance of the positive class at point  $x$ . An example of such a score is the fraction of positive observations in a leaf of a decision tree. In this case, scores fall into the range from 0 to 1 and scores from positive and negative classes add up to unity. Other methods can return scores ranging between minus and plus infinity, without any obvious mapping from the score to the posterior class probability.

`perfcurve` does not impose any requirements on the input score range. Because of this lack of normalization, you can use `perfcurve` to process scores returned by any classification, regression, or fit method. `perfcurve` does not make any assumptions about the nature of input scores or relationships between the scores for different classes. As an example, consider a problem with three classes, A, B, and C, and assume that the scores returned by some classifier for two instances are as follows:

	A	B	C
instance 1	0.4	0.5	0.1
instance 2	0.4	0.1	0.5

If you want to compute a performance curve for separation of classes A and B, with C ignored, you need to address the ambiguity in selecting A over B. You could opt to use the score ratio,  $s(A) / s(B)$ , or score difference,  $s(A) - s(B)$ ; this choice could depend on the nature of these scores and their normalization. `perfcurve` always takes one score per instance. If you only supply scores for class A, `perfcurve` does not distinguish between observations 1 and 2. The performance curve in this case may not be optimal.

`perfcurve` is intended for use with classifiers that return scores, not those that return only predicted classes. As a counter-example, consider a decision tree that returns only hard classification labels, 0 or 1, for data with two classes. In this case, the performance curve reduces to a single point because classified instances can be split into positive and negative categories in one way only.

For input, `perfcurve` takes true class labels for some data and scores assigned by a classifier to these data. By default, this utility computes a Receiver Operating Characteristic (ROC) curve and returns values of 1-specificity, or false positive rate, for  $X$  and sensitivity, or true positive rate, for  $Y$ . You can choose other criteria for  $X$  and  $Y$  by selecting one out of several provided criteria or specifying an arbitrary criterion through an anonymous function. You can display the computed performance curve using `plot(X, Y)`.

`perfcurve` can compute values for various criteria to plot either on the  $x$ - or the  $y$ -axis. All such criteria are described by a 2-by-2 confusion matrix, a 2-by-2 cost matrix, and a 2-by-1 vector of scales applied to class counts.

The confusion matrix,  $C$ , is defined as

$$\begin{pmatrix} TP & FN \\ FP & TN \end{pmatrix}$$

where

- $P$  stands for "positive".
- $N$  stands for "negative".
- $T$  stands for "true".
- $F$  stands for "false".

For example, the first row of the confusion matrix defines how the classifier identifies instances of the positive class:  $C(1, 1)$  is the count of correctly identified positive instances and  $C(1, 2)$  is the count of positive instances misidentified as negative.

The cost matrix defines the cost of misclassification for each category:

$$\begin{pmatrix} Cost(P|P) & Cost(N|P) \\ Cost(P|N) & Cost(N|N) \end{pmatrix}$$

where  $Cost(I|J)$  is the cost of assigning an instance of class  $J$  to class  $I$ . Usually  $Cost(I|J)=0$  for  $I=J$ . For flexibility, `perfcurve` allows you to specify nonzero costs for correct classification as well.

The two scales include prior information about class probabilities. `perfcurve` computes these scales by taking  $scale(P)=prior(P)*N$  and  $scale(N)=prior(N)*P$  and normalizing the sum  $scale(P)+scale(N)$  to 1.  $P=TP+FN$  and  $N=TN+FP$  are the total instance counts in the positive and negative class, respectively. The function then applies the scales as multiplicative factors to the counts from the corresponding class: `perfcurve` multiplies counts from the positive class by  $scale(P)$  and counts from the negative class by  $scale(N)$ . Consider, for example, computation of positive predictive value,  $PPV = TP/(TP+FP)$ .  $TP$  counts come from the positive class and  $FP$  counts come from the negative class. Therefore, you need to scale  $TP$  by  $scale(P)$  and  $FP$  by  $scale(N)$ , and the modified formula for  $PPV$  with prior probabilities taken into account is now:

$$PPV = \frac{scale(P) * TP}{scale(P) * TP + scale(N) * FP}$$

If all scores in the data are above a certain threshold, `perfcurve` classifies all instances as 'positive'. This means that `TP` is the total number of instances in the positive class and `FP` is the total number of instances in the negative class. In this case, `PPV` is simply given by the prior:

$$PPV = \frac{prior(P)}{prior(P) + prior(N)}$$

The `perfcurve` function returns two vectors, `X` and `Y`, of performance measures. Each measure is some function of `confusion`, `cost`, and `scale` values. You can request specific measures by name or provide a function handle to compute a custom measure. The function you provide should take `confusion`, `cost`, and `scale` as its three inputs and return a vector of output values.

The criterion for `X` must be a monotone function of the positive classification count, or equivalently, threshold for the supplied scores. If `perfcurve` cannot perform a one-to-one mapping between values of the `X` criterion and score thresholds, it exits with an error message.

By default, `perfcurve` computes values of the `X` and `Y` criteria for all possible score thresholds. Alternatively, it can compute a reduced number of specific `X` values supplied as an input argument. In either case, for `M` requested values, `perfcurve` computes `M+1` values for `X` and `Y`. The first value out of these `M+1` values is special. `perfcurve` computes it by setting the `TP` instance count to zero and setting `TN` to the total count in the negative class. This value corresponds to the 'reject all' threshold. On a standard ROC curve, this translates into an extra point placed at  $(0, 0)$ .

If there are `NaN` values among input scores, `perfcurve` can process them in either of two ways:

- It can discard rows with `NaN` scores.
- It can add them to false classification counts in the respective class.

That is, for any threshold, instances with `NaN` scores from the positive class are counted as false negative (`FN`), and instances with `NaN` scores from the negative class are counted as false positive (`FP`). In this case, the first value of `X` or `Y` is computed by setting `TP` to zero and setting `TN` to the total count minus the `NaN` count in the negative class. For

illustration, consider an example with two rows in the positive and two rows in the negative class, each pair having a NaN score:

Class	Score
Negative	0.2
Negative	NaN
Positive	0.7
Positive	NaN

If you discard rows with NaN scores, then as the score cutoff varies, `perfcurve` computes performance measures as in the following table. For example, a cutoff of 0.5 corresponds to the middle row where rows 1 and 3 are classified correctly, and rows 2 and 4 are omitted.

TP	FN	FP	TN
0	1	0	1
1	0	0	1
1	0	1	0

If you add rows with NaN scores to the false category in their respective classes, `perfcurve` computes performance measures as in the following table. For example, a cutoff of 0.5 corresponds to the middle row where now rows 2 and 4 are counted as incorrectly classified. Notice that only the FN and FP columns differ between these two tables.

TP	FN	FP	TN
0	2	1	1
1	1	1	1
1	1	2	0

For data with three or more classes, `perfcurve` takes one positive class and a list of negative classes for input. The function computes the X and Y values using counts in the positive class to estimate TP and FN, and using counts in all negative classes to estimate TN and FP. `perfcurve` can optionally compute Y values for each negative class separately and, in addition to Y, return a matrix of size M-by-C, where M is the number of elements in X or Y and C is the number of negative classes. You can use this functionality to monitor components of the negative class contribution. For example, you can plot TP counts on the X-axis and FP counts on the Y-axis. In this case, the returned matrix shows how the FP component is split across negative classes.

You can also use `perfcurve` to estimate confidence intervals. `perfcurve` computes confidence bounds using either cross-validation or bootstrap. If you supply cell arrays for `labels` and `scores`, `perfcurve` uses cross-validation and treats elements in the cell arrays as cross-validation folds. If you set input parameter `NBoot` to a positive integer, `perfcurve` generates `nboot` bootstrap replicas to compute pointwise confidence bounds.

`perfcurve` estimates the confidence bounds using one of two methods:

- Vertical averaging (VA) — estimate confidence bounds on  $Y$  and  $T$  at fixed values of  $X$ . Use the `XVals` input parameter to use this method for computing confidence bounds.
- Threshold averaging (TA) — estimate confidence bounds for  $X$  and  $Y$  at fixed thresholds for the positive class score. Use the `TVals` input parameter to use this method for computing confidence bounds.

To use observation weights instead of observation counts, you can use the `'Weights'` parameter in your call to `perfcurve`. When you use this parameter, to compute  $X$ ,  $Y$  and  $T$  or to compute confidence bounds by cross-validation, `perfcurve` uses your supplied observation weights instead of observation counts. To compute confidence bounds by bootstrap, `perfcurve` samples  $N$  out of  $N$  with replacement using your weights as multinomial sampling probabilities.



# Nonparametric Supervised Learning

---

- “Supervised Learning Workflow and Algorithms” on page 18-2
- “Visualize Decision Surfaces of Different Classifiers” on page 18-11
- “Classification Using Nearest Neighbors” on page 18-16
- “Framework for Ensemble Learning” on page 18-41
- “Ensemble Algorithms” on page 18-51
- “Train Classification Ensemble” on page 18-67
- “Train Regression Ensemble” on page 18-70
- “Select Predictors for Random Forests” on page 18-73
- “Test Ensemble Quality” on page 18-81
- “Ensemble Regularization” on page 18-88
- “Classification with Imbalanced Data” on page 18-101
- “Handle Imbalanced Data or Unequal Misclassification Costs in Classification Ensembles” on page 18-107
- “Classification with Many Categorical Levels” on page 18-115
- “Surrogate Splits” on page 18-121
- “LPBoost and TotalBoost for Small Ensembles” on page 18-124
- “Tune RobustBoost” on page 18-131
- “Random Subspace Classification” on page 18-135
- “Bootstrap Aggregation (Bagging) of Regression Trees” on page 18-141
- “Bootstrap Aggregation (Bagging) of Classification Trees” on page 18-154
- “Detect Outliers Using Quantile Regression” on page 18-170
- “Conditional Quantile Estimation Using Kernel Smoothing” on page 18-177
- “Tune Random Forest Using Quantile Error and Bayesian Optimization” on page 18-181
- “Support Vector Machines for Binary Classification” on page 18-186
- “Bibliography” on page 18-228

## Supervised Learning Workflow and Algorithms

### In this section...

“What is Supervised Learning?” on page 18-2

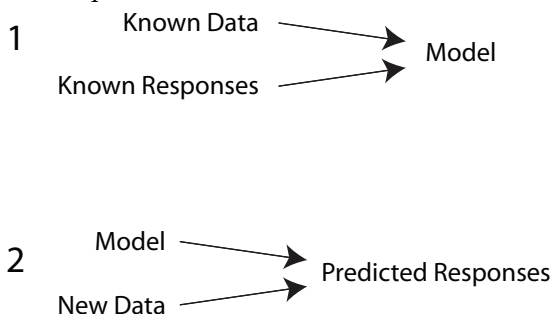
“Steps in Supervised Learning” on page 18-3

“Characteristics of Classification Algorithms” on page 18-7

### What is Supervised Learning?

The aim of supervised, machine learning is to build a model that makes predictions based on evidence in the presence of uncertainty. As adaptive algorithms identify patterns in data, a computer "learns" from the observations. When exposed to more observations, the computer improves its predictive performance.

Specifically, a supervised learning algorithm takes a known set of input data and known responses to the data (output), and trains a model to generate reasonable predictions for the response to new data.



For example, suppose you want to predict whether someone will have a heart attack within a year. You have a set of data on previous patients, including age, weight, height, blood pressure, etc. You know whether the previous patients had heart attacks within a year of their measurements. So, the problem is combining all the existing data into a model that can predict whether a new person will have a heart attack within a year.

You can think of the entire set of input data as a heterogeneous matrix. Rows of the matrix are called observations, examples, or instances, and each contain a set of measurements for a subject (patients in the example). Columns of the matrix are called predictors, attributes, or features, and each are variables representing a measurement

taken on every subject (age, weight, height, etc. in the example). You can think of the response data as a column vector where each row contains the output of the corresponding observation in the input data (whether the patient had a heart attack). To fit or train a supervised learning model, choose an appropriate algorithm, and then pass the input and response data to it.

Supervised learning splits into two broad categories: classification and regression.

- In classification, the goal is to assign a class (or label) from a finite set of classes to an observation. That is, responses are categorical variables. Applications include spam filters, advertisement recommendation systems, and image and speech recognition. Predicting whether a patient will have a heart attack within a year is a classification problem, and the possible classes are `true` and `false`. Classification algorithms usually apply to nominal response values. However, some algorithms can accommodate ordinal classes (see `fitcecoc`).
- In regression, the goal is to predict a continuous measurement for an observation. That is, the responses variables are real numbers. Applications include forecasting stock prices, energy consumption, or disease incidence.

Statistics and Machine Learning Toolbox supervised learning functionalities comprise a stream-lined, object framework. You can efficiently train a variety of algorithms, combine models into an ensemble, assess model performances, cross-validate, and predict responses for new data.

## Steps in Supervised Learning

While there are many Statistics and Machine Learning Toolbox algorithms for supervised learning, most use the same basic workflow for obtaining a predictor model. (Detailed instruction on the steps for ensemble learning is in “Framework for Ensemble Learning” on page 18-41.) The steps for supervised learning are:

1. “Prepare Data” on page 18-4
2. “Choose an Algorithm” on page 18-4
3. “Fit a Model” on page 18-5
4. “Choose a Validation Method” on page 18-5
5. “Examine Fit and Update Until Satisfied” on page 18-6
6. “Use Fitted Model for Predictions” on page 18-7

## Prepare Data

All supervised learning methods start with an input data matrix, usually called  $X$  here. Each row of  $X$  represents one observation. Each column of  $X$  represents one variable, or predictor. Represent missing entries with NaN values in  $X$ . Statistics and Machine Learning Toolbox supervised learning algorithms can handle NaN values, either by ignoring them or by ignoring any row with a NaN value.

You can use various data types for response data  $Y$ . Each element in  $Y$  represents the response to the corresponding row of  $X$ . Observations with missing  $Y$  data are ignored.

- For regression,  $Y$  must be a numeric vector with the same number of elements as the number of rows of  $X$ .
- For classification,  $Y$  can be any of these data types. This table also contains the method of including missing entries.

Data Type	Missing Entry
Numeric vector	NaN
Categorical vector	<undefined>
Character array	Row of spaces
Cell array of character vectors	' '
Logical vector	(Cannot represent)

## Choose an Algorithm

There are tradeoffs between several characteristics of algorithms, such as:

- Speed of training
- Memory usage
- Predictive accuracy on new data
- Transparency or interpretability, meaning how easily you can understand the reasons an algorithm makes its predictions

Details of the algorithms appear in “Characteristics of Classification Algorithms” on page 18-7. More detail about ensemble algorithms is in “Choose an Applicable Ensemble Aggregation Method” on page 18-43.

## Fit a Model

The fitting function you use depends on the algorithm you choose.

Algorithm	Fitting Function
Classification Trees	<code>fitctree</code>
Regression Trees	<code>fitrtree</code>
Discriminant Analysis (classification)	<code>fitcdiscr</code>
$k$ -Nearest Neighbors (classification)	<code>fitcknn</code>
Naive Bayes (classification)	<code>fitcnb</code>
Support Vector Machines (SVM) for classification	<code>fitcsvm</code>
SVM for regression	<code>fitrsvm</code>
Multiclass models for SVM or other classifiers	<code>fitcecoc</code>
Classification Ensembles	<code>fitcensemble</code>
Regression Ensembles	<code>fitrensemble</code>
Classification or Regression Tree Ensembles (e.g., Random Forests [1]) in Parallel	<code>TreeBagger</code>

For a comparison of these algorithms, see “Characteristics of Classification Algorithms” on page 18-7.

## Choose a Validation Method

The three main methods to examine the accuracy of the resulting fitted model are:

- Examine the resubstitution error. For examples, see:
  - “Classification Tree Resubstitution Error” on page 19-17
  - “Cross Validate a Regression Tree” on page 19-18
  - “Test Ensemble Quality” on page 18-81
  - “Example: Resubstitution Error of a Discriminant Analysis Classifier” on page 20-22
- Examine the cross-validation error. For examples, see:

- “Cross Validate a Regression Tree” on page 19-18
- “Test Ensemble Quality” on page 18-81
- “Classification with Many Categorical Levels” on page 18-115
- “Cross Validating a Discriminant Analysis Classifier” on page 20-23
- Examine the out-of-bag error for bagged decision trees. For examples, see:
  - “Test Ensemble Quality” on page 18-81
  - “Regression of Insurance Risk Rating for Car Imports Using TreeBagger” on page 18-141
  - “Classifying Radar Returns for Ionosphere Data Using TreeBagger” on page 18-154

### **Examine Fit and Update Until Satisfied**

After validating the model, you might want to change it for better accuracy, better speed, or to use less memory.

- Change fitting parameters to try to get a more accurate model. For examples, see:
  - “Tune RobustBoost” on page 18-131
  - “Train Ensemble With Unequal Classification Costs” on page 18-108
  - “Improving Discriminant Analysis Models” on page 20-20
- Change fitting parameters to try to get a smaller model. This sometimes gives a model with more accuracy. For examples, see:
  - “Select Appropriate Tree Depth” on page 19-22
  - “Prune a Classification Tree” on page 19-26
  - “Surrogate Splits” on page 18-121
  - “Regularize a Regression Ensemble” on page 18-89
  - “Regression of Insurance Risk Rating for Car Imports Using TreeBagger” on page 18-141
  - “Classifying Radar Returns for Ionosphere Data Using TreeBagger” on page 18-154
- Try a different algorithm. For applicable choices, see:
  - “Characteristics of Classification Algorithms” on page 18-7

- “Choose an Applicable Ensemble Aggregation Method” on page 18-43

When satisfied with a model of some types, you can trim it using the appropriate compact function (`compact` for classification trees, `compact` for regression trees, `compact` for discriminant analysis, `compact` for naive Bayes, `compact` for SVM, `compact` for ECOC models, `compact` for classification ensembles, and `compact` for regression ensembles). `compact` removes training data and other properties not required for prediction, e.g., pruning information for decision trees, from the model to reduce memory consumption. Because  $k$ NN classification models require all of the training data to predict labels, you cannot reduce the size of a `ClassificationKNN` model.

### Use Fitted Model for Predictions

To predict classification or regression response for most fitted models, use the `predict` method:

```
Ypredicted = predict(obj, Xnew)
```

- `obj` is the fitted model or fitted compact model.
- `Xnew` is the new input data.
- `Ypredicted` is the predicted response, either classification or regression.

## Characteristics of Classification Algorithms

This table shows typical characteristics of the various supervised learning algorithms. The characteristics in any particular case can vary from the listed ones. Use the table as a guide for your initial choice of algorithms. Decide on the tradeoff you want in speed, memory usage, flexibility, and interpretability.

---

**Tip** Try a decision tree or discriminant first, because these classifiers are fast and easy to interpret. If the models are not accurate enough predicting the response, try other classifiers with higher flexibility.

To control flexibility, see the details for each classifier type. To avoid overfitting, look for a model of lower flexibility that provides sufficient accuracy.

---

Classifier	Multiclass Support	Categorical Predictor Support on page 18-9	Prediction Speed	Memory Usage	Interpretability
“Decision Trees” on page 19-2 — <code>fitctree</code>	Yes	Yes	Fast	Small	Easy
Discriminant analysis on page 20-2 — <code>fitcdiscr</code>	Yes	No	Fast	Small for linear, large for quadratic	Easy
SVM on page 18-186 — <code>fitcsvm</code>	No. Combine multiple binary SVM classifiers using <code>fitcecoc</code> .	Yes	Medium for linear. Slow for others.	Medium for linear. All others: medium for multiclass, large for binary.	Easy for linear SVM. Hard for all other kernel types.
Naive Bayes on page 21-2 — <code>fitcnb</code>	Yes	Yes	Medium for simple distributions. Slow for kernel distributions or high-dimensional data	Small for simple distributions. Medium for kernel distributions or high-dimensional data	Easy
Nearest neighbor on page 18-16 — <code>fitcknn</code>	Yes	Yes	Slow for cubic. Medium for others.	Medium	Hard
Ensembles on page 18-41 — <code>fitensemble</code>	Yes	Yes	Fast to medium depending on choice of algorithm	Low to high depending on choice of algorithm.	Hard



The results in this table are based on an analysis of many data sets. The data sets in the study have up to 7000 observations, 80 predictors, and 50 classes. This list defines the terms in the table.

Speed:

- Fast — 0.01 second
- Medium — 1 second
- Slow — 100 seconds

Memory

- Small — 1MB
- Medium — 4MB
- Large — 100MB

---

**Note** The table provides a general guide. Your results depend on your data and the speed of your machine.

---

### Categorical Predictor Support

This table describes the data-type support of predictors for each classifier.

Classifier	All predictors numeric	All predictors categorical	Some categorical, some numeric
Decision Trees	Yes	Yes	Yes
Discriminant Analysis	Yes	No	No
SVM	Yes	Yes	Yes
Naive Bayes	Yes	Yes	Yes
Nearest Neighbor	Euclidean distance only	Hamming distance only	No
Ensembles	Yes	Yes, except subspace ensembles of discriminant analysis classifiers	Yes, except subspace ensembles

## References

- [1] Breiman, L. *Random Forests*. *Machine Learning* 45, 2001, pp. 5–32.

## Visualize Decision Surfaces of Different Classifiers

This example shows how to plot the decision surface of different classification algorithms.

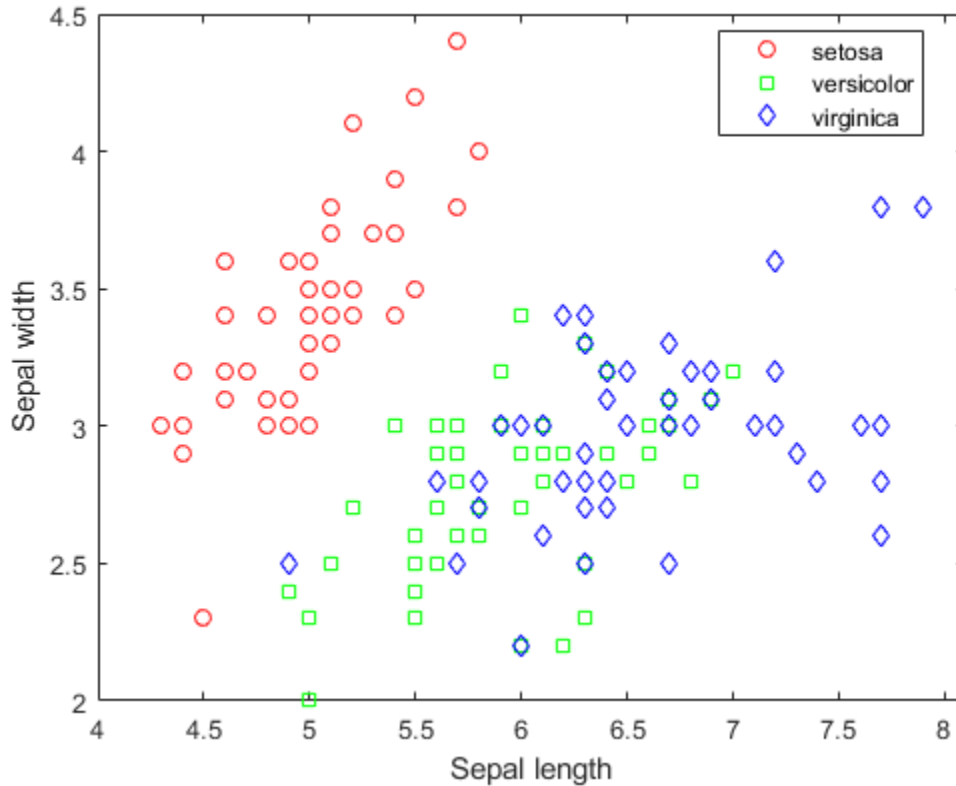
Load Fisher's iris data set.

```
load fisheriris
X = meas(:,1:2);
y = categorical(species);
labels = categories(y);
```

X is a numeric matrix that contains two petal measurements for 150 irises. Y is a cell array of character vectors that contains the corresponding iris species.

Visualize the data using a scatter plot. Group the variables by iris species.

```
gscatter(X(:,1),X(:,2),species,'rgb','osd');
xlabel('Sepal length');
ylabel('Sepal width');
```



Train four different classifiers and store the models in a cell array.

```
classifier_name = {'Naive Bayes', 'Discriminant Analysis', 'Classification Tree', 'Nearest
```

Train a naive Bayes model.

```
classifier{1} = fitcnb(X,y);
```

Train a discriminant analysis classifier.

```
classifier{2} = fitcdiscr(X,y);
```

Train a classification decision tree.

```
classifier{3} = fitctree(X,y);
```

Train a  $k$ -nearest neighbor classifier.

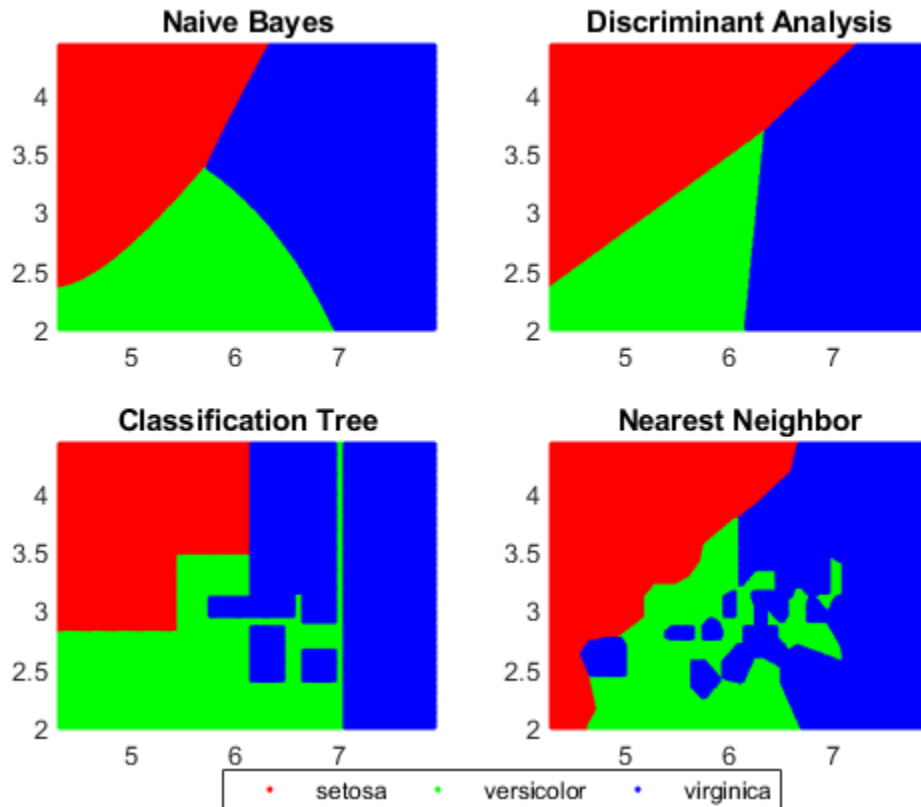
```
classifier{4} = fitcknn(X,y);
```

Create a grid of points spanning the entire space within some bounds of the actual data values.

```
x1range = min(X(:,1)):.01:max(X(:,1));  
x2range = min(X(:,2)):.01:max(X(:,2));  
[xx1, xx2] = meshgrid(x1range,x2range);  
XGrid = [xx1(:) xx2(:)];
```

Predict the iris species of each observation in XGrid using all classifiers. Plot the a scatter plot of the results.

```
for i = 1:numel(classifier)  
    predictedspecies = predict(classifier{i},XGrid);  
  
    subplot(2,2,i);  
    gscatter(xx1(:), xx2(:), predictedspecies, 'rgb');  
  
    title(classifier_name{i})  
    legend off, axis tight  
end  
  
legend(labels, 'Location', [0.35,0.01,0.35,0.05], 'Orientation', 'Horizontal')
```



Each classification algorithm generates different decision making rules. A decision surface can help you visualize these rules.

## See Also

### Functions

`fitcdiscr` | `fitcknn` | `fitcnb` | `fitctree`

## Related Examples

- “Plot Posterior Classification Probabilities” on page 21-7

## Classification Using Nearest Neighbors

### In this section...

“Pairwise Distance Metrics” on page 18-16

“k-Nearest Neighbor Search and Radius Search” on page 18-18

“Classify Query Data” on page 18-24

“Find Nearest Neighbors Using a Custom Distance Metric” on page 18-32

“K-Nearest Neighbor Classification for Supervised Learning” on page 18-36

“Construct KNN Classifier” on page 18-36

“Examine Quality of KNN Classifier” on page 18-37

“Predict Classification Using KNN Classifier” on page 18-38

“Modify KNN Classifier” on page 18-39

### Pairwise Distance Metrics

Categorizing query points based on their distance to points in a training data set can be a simple yet effective way of classifying new points. You can use various metrics to determine the distance, described next. Use `pdist2` to find the distance between a set of data and query points.

#### Distance Metrics

Given an  $m \times n$  data matrix  $X$ , which is treated as  $m$  (1-by- $n$ ) row vectors  $x_1, x_2, \dots, x_m$ , and an  $m' \times n$  data matrix  $Y$ , which is treated as  $m'$  (1-by- $n$ ) row vectors  $y_1, y_2, \dots, y_{m'}$ , the various distances between the vector  $x_s$  and  $y_t$  are defined as follows:

- Euclidean distance

$$d_{st}^2 = (x_s - y_t)(x_s - y_t)'$$

The Euclidean distance is a special case of the Minkowski distance, where  $p = 2$ .

- Standardized Euclidean distance

$$d_{st}^2 = (x_s - y_t)V^{-1}(x_s - y_t)'$$

where  $V$  is the  $n$ -by- $n$  diagonal matrix whose  $j$ th diagonal element is  $(S(j))^2$ , where  $S$  is a vector of scaling factors for each dimension.



- Mahalanobis distance

$$d_{st}^2 = (x_s - y_t)C^{-1}(x_s - y_t)'$$

where  $C$  is the covariance matrix.

- City block distance

$$d_{st} = \sum_{j=1}^n |x_{sj} - y_{tj}|.$$

The city block distance is a special case of the Minkowski distance, where  $p = 1$ .

- Minkowski distance

$$d_{st} = \sqrt[p]{\sum_{j=1}^n |x_{sj} - y_{tj}|^p}.$$

For the special case of  $p = 1$ , the Minkowski distance gives the city block distance. For the special case of  $p = 2$ , the Minkowski distance gives the Euclidean distance. For the special case of  $p = \infty$ , the Minkowski distance gives the Chebychev distance.

- Chebychev distance

$$d_{st} = \max_j \{|x_{sj} - y_{tj}|\}.$$

The Chebychev distance is a special case of the Minkowski distance, where  $p = \infty$ .

- Cosine distance

$$d_{st} = \left( 1 - \frac{x_s y_t'}{\sqrt{(x_s x_s')(y_t y_t')}} \right).$$

- Correlation distance

$$d_{st} = 1 - \frac{(x_s - \bar{x}_s)(y_t - \bar{y}_t)'}{\sqrt{(x_s - \bar{x}_s)(x_s - \bar{x}_s)'}\sqrt{(y_t - \bar{y}_t)(y_t - \bar{y}_t)'}}$$

where

$$\bar{x}_s = \frac{1}{n} \sum_j x_{sj}$$

and

$$\bar{y}_t = \frac{1}{n} \sum_j y_{tj}.$$

- Hamming distance

$$d_{st} = (\#(x_{sj} \neq y_{tj}) / n).$$

- Jaccard distance

$$d_{st} = \frac{\#[(x_{sj} \neq y_{tj}) \cap ((x_{sj} \neq 0) \cup (y_{tj} \neq 0))]}{\#[(x_{sj} \neq 0) \cup (y_{tj} \neq 0)]}.$$

- Spearman distance

$$d_{st} = 1 - \frac{(r_s - \bar{r}_s)(r_t - \bar{r}_t)'}{\sqrt{(r_s - \bar{r}_s)(r_s - \bar{r}_s)' \sqrt{(r_t - \bar{r}_t)(r_t - \bar{r}_t)'}}$$

where

- $r_{sj}$  is the rank of  $x_{sj}$  taken over  $x_{1j}, x_{2j}, \dots, x_{mj}$ , as computed by `tiedrank`.
- $r_{tj}$  is the rank of  $y_{tj}$  taken over  $y_{1j}, y_{2j}, \dots, y_{mj}$ , as computed by `tiedrank`.
- $r_s$  and  $r_t$  are the coordinate-wise rank vectors of  $x_s$  and  $y_t$ , i.e.,  $r_s = (r_{s1}, r_{s2}, \dots, r_{sn})$  and  $r_t = (r_{t1}, r_{t2}, \dots, r_{tn})$ .

•

$$\bar{r}_s = \frac{1}{n} \sum_j r_{sj} = \frac{(n+1)}{2}.$$

•

$$\bar{r}_t = \frac{1}{n} \sum_j r_{tj} = \frac{(n+1)}{2}.$$

## k-Nearest Neighbor Search and Radius Search

Given a set  $X$  of  $n$  points and a distance function,  $k$ -nearest neighbor ( $k$ NN) search lets you find the  $k$  closest points in  $X$  to a query point or set of points  $\mathcal{Y}$ . The  $k$ NN search technique and  $k$ NN-based algorithms are widely used as benchmark learning rules. The relative simplicity of the  $k$ NN search technique makes it easy to compare the results

from other classification techniques to  $k$ NN results. The technique has been used in various areas such as:

- bioinformatics
- image processing and data compression
- document retrieval
- computer vision
- multimedia database
- marketing data analysis

You can use  $k$ NN search for other machine learning algorithms, such as:

- $k$ NN classification
- local weighted regression
- missing data imputation and interpolation
- density estimation

You can also use  $k$ NN search with many distance-based learning functions, such as K-means clustering.

In contrast, for a positive real value  $r$ , `rangesearch` finds all points in  $X$  that are within a distance  $r$  of each point in  $Y$ . This fixed-radius search is closely related to  $k$ NN search, as it supports the same distance metrics and search classes, and uses the same search algorithms.

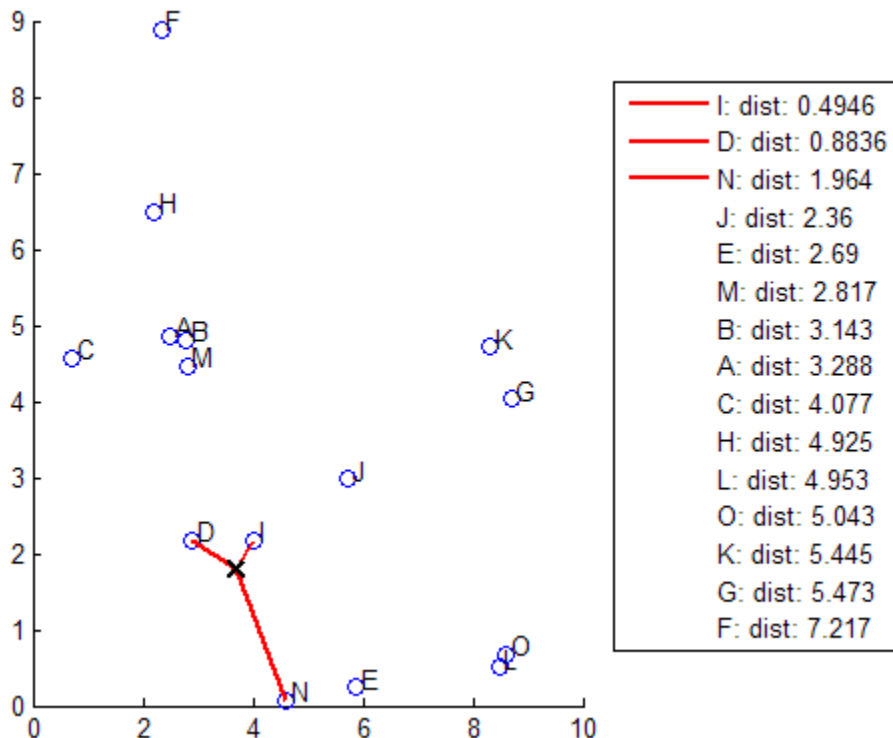
### **k-Nearest Neighbor Search Using Exhaustive Search**

When your input data meets any of the following criteria, `knnsearch` uses the exhaustive search method by default to find the  $k$ -nearest neighbors:

- The number of columns of  $X$  is more than 10.
- $X$  is sparse.
- The distance measure is either:
  - 'seuclidean'
  - 'mahalanobis'
  - 'cosine'

- 'correlation'
- 'spearman'
- 'hamming'
- 'jaccard'
- A custom distance function

`knnsearch` also uses the exhaustive search method if your search object is an `ExhaustiveSearcher` model object. The exhaustive search method finds the distance from each query point to every point in  $X$ , ranks them in ascending order, and returns the  $k$  points with the smallest distances. For example, this diagram shows the  $k = 3$  nearest neighbors.



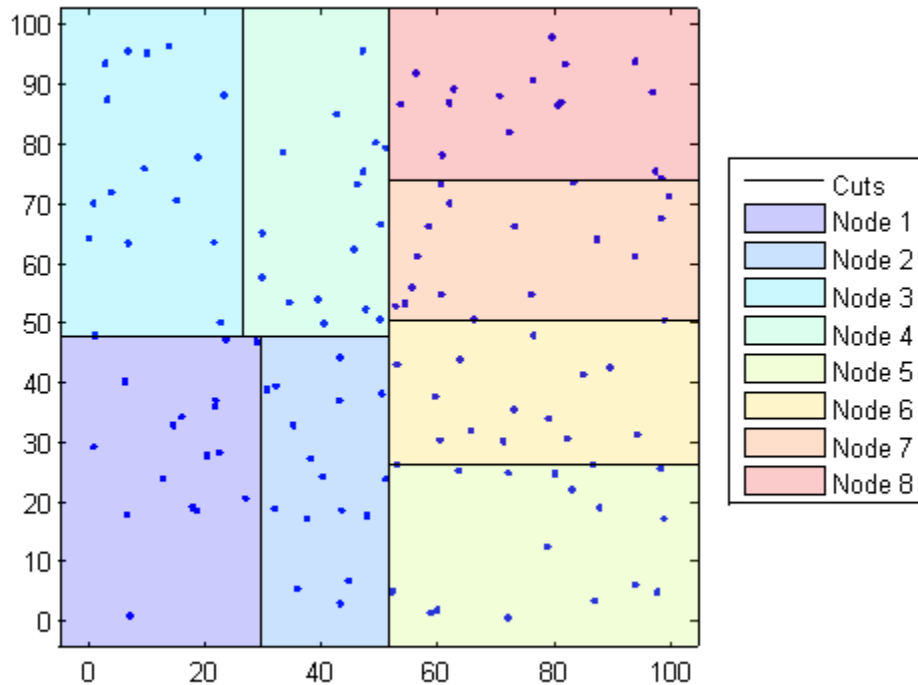
### k-Nearest Neighbor Search Using a Kd-Tree

When your input data meets all of the following criteria, `knnsearch` creates a *Kd*-tree by default to find the  $k$ -nearest neighbors:

- The number of columns of  $X$  is less than 10.
- $X$  is not sparse.
- The distance measure is either:
  - 'euclidean' (default)
  - 'cityblock'
  - 'minkowski'
  - 'chebychev'

`knnsearch` also uses a *Kd*-tree if your search object is a `KDTreeSearcher` model object.

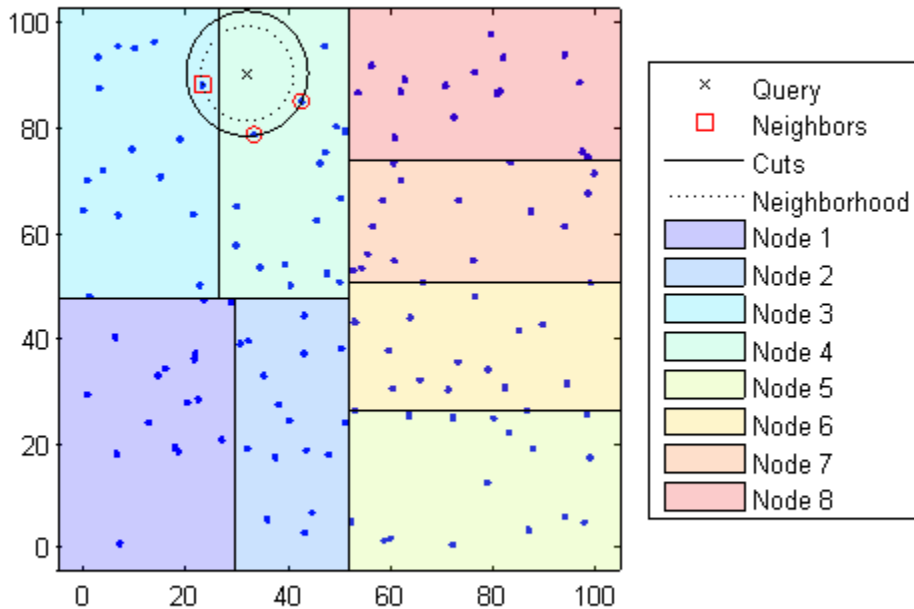
*Kd*-trees divide your data into nodes with at most `BucketSize` (default is 50) points per node, based on coordinates (as opposed to categories). The following diagrams illustrate this concept using `patch` objects to color code the different “buckets.”



When you want to find the  $k$ -nearest neighbors to a given query point, `knnsearch` does the following:

- 1 Determines the node to which the query point belongs. In the following example, the query point (32,90) belongs to Node 4.
- 2 Finds the closest  $k$  points within that node and its distance to the query point. In the following example, the points in red circles are equidistant from the query point, and are the closest points to the query point within Node 4.
- 3 Chooses all other nodes having any area that is within the same distance, in any direction, from the query point to the  $k$ th closest point. In this example, only Node 3 overlaps the solid black circle centered at the query point with radius equal to the distance to the closest points within Node 4.

- 4 Searches nodes within that range for any points closer to the query point. In the following example, the point in a red square is slightly closer to the query point than those within Node 4.



Using a *Kd-tree* for large data sets with fewer than 10 dimensions (columns) can be much more efficient than using the exhaustive search method, as `knnsearch` needs to calculate only a subset of the distances. To maximize the efficiency of *Kd-trees*, use a `KDTreeSearcher` model.

### What Are Search Model Objects?

Basically, model objects are a convenient way of storing information. Related models have the same properties with values and types relevant to a specified search method. In

addition to storing information within models, you can perform certain actions on models.

You can efficiently perform a  $k$ -nearest neighbors search on your search model using `knnsearch`. Or, you can search for all neighbors within a specified radius using your search model and `rangesearch`. In addition, there are a generic `knnsearch` and `rangesearch` functions that search without creating or using a model.

To determine which type of model and search method is best for your data, consider the following:

- Does your data have many columns, say more than 10? The `ExhaustiveSearcher` model may perform better.
- Is your data sparse? Use the `ExhaustiveSearcher` model.
- Do you want to use one of these distance measures to find the nearest neighbors? Use the `ExhaustiveSearcher` model.
  - 'seuclidean'
  - 'mahalanobis'
  - 'cosine'
  - 'correlation'
  - 'spearman'
  - 'hamming'
  - 'jaccard'
  - A custom distance function
- Is your data set huge (but with fewer than 10 columns)? Use the `KDTreeSearcher` model.
- Are you searching for the nearest neighbors for a large number of query points? Use the `KDTreeSearcher` model.

## Classify Query Data

This example shows how to classify query data by:

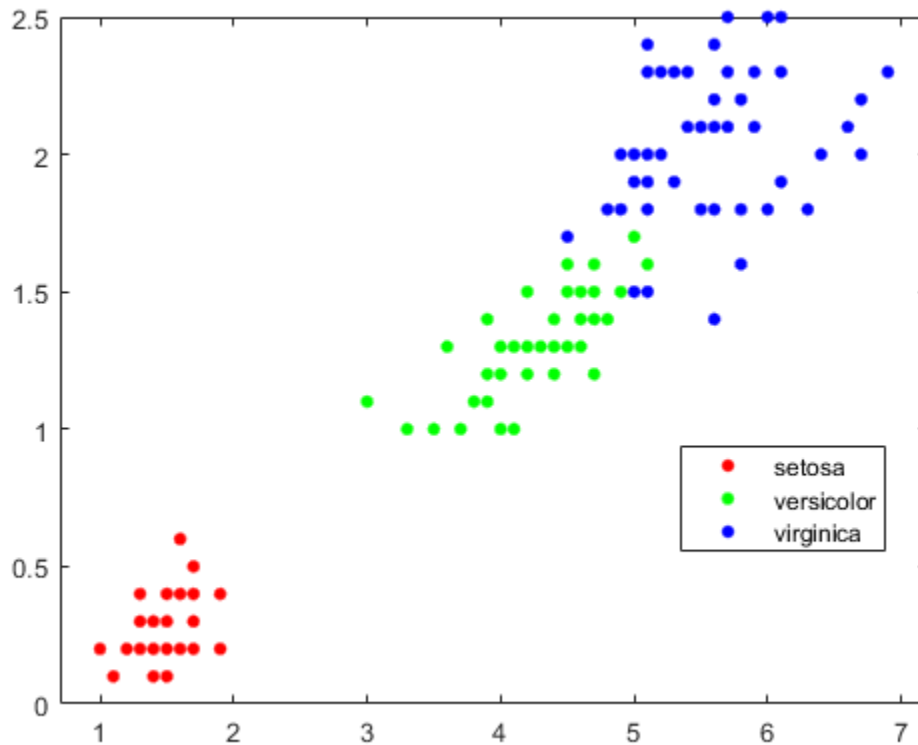
- 1 Growing a  $K$  d-tree
- 2 Conducting a  $k$  nearest neighbors search using the grown tree.



- Assigning each query point the class with the highest representation among their respective nearest neighbors.

Classify a new point based on the last two columns of the Fisher iris data. Using only the last two columns makes it easier to plot.

```
load fisheriris
x = meas(:,3:4);
gscatter(x(:,1),x(:,2),species)
legend('Location','best')
```

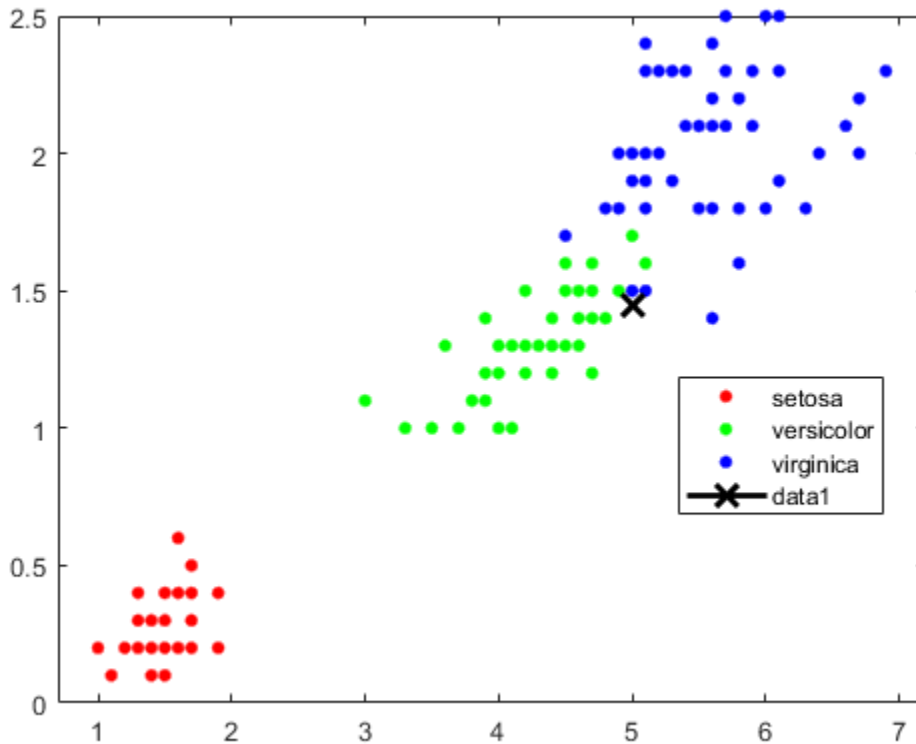


Plot the new point.

```

newpoint = [5 1.45];
line(newpoint(1),newpoint(2), 'marker', 'x', 'color', 'k', ...
     'markersize', 10, 'linewidth', 2)

```



Prepare a  $K$  d-tree neighbor searcher model.

```
Mdl = KDTreeSearcher(x)
```

```
Mdl =
```

```
KDTreeSearcher with properties:
```

```

    BucketSize: 50
    Distance: 'euclidean'

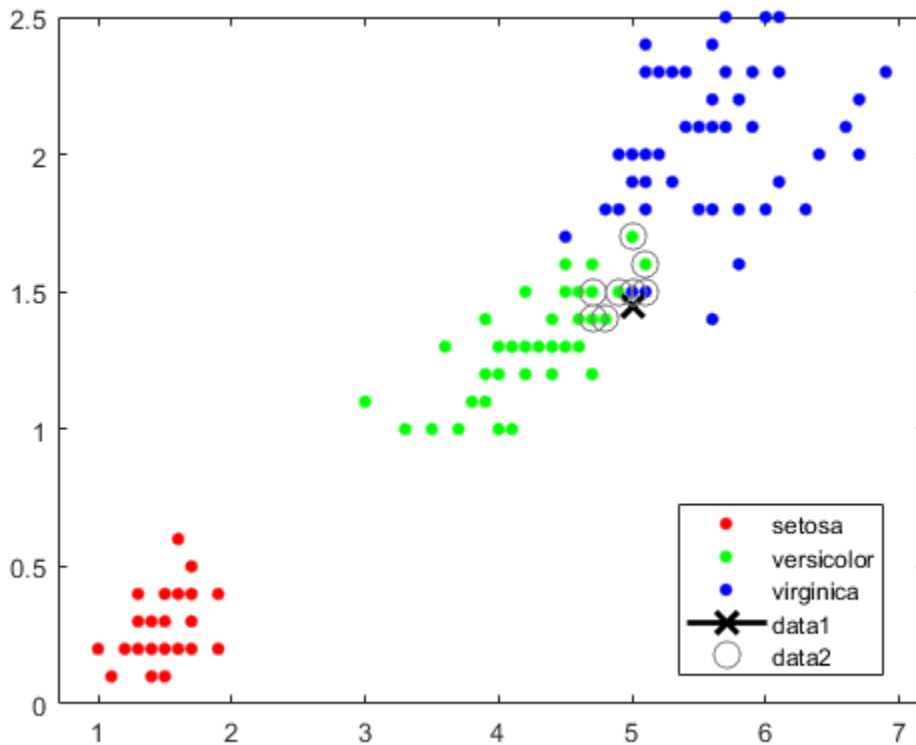
```

```
DistParameter: []  
X: [150x2 double]
```

Mdl is a `KDTreeSearcher` model. By default, the distance metric it uses to search for neighbors is Euclidean distance.

Find the 10 sample points closest to the new point.

```
[n,d] = knnsearch(Mdl,newpoint,'k',10);  
line(x(n,1),x(n,2),'color',[.5 .5 .5],'marker','o',...  
      'linestyle','none','markersize',10)
```



It appears that `knnsearch` has found only the nearest eight neighbors. In fact, this particular dataset contains duplicate values.

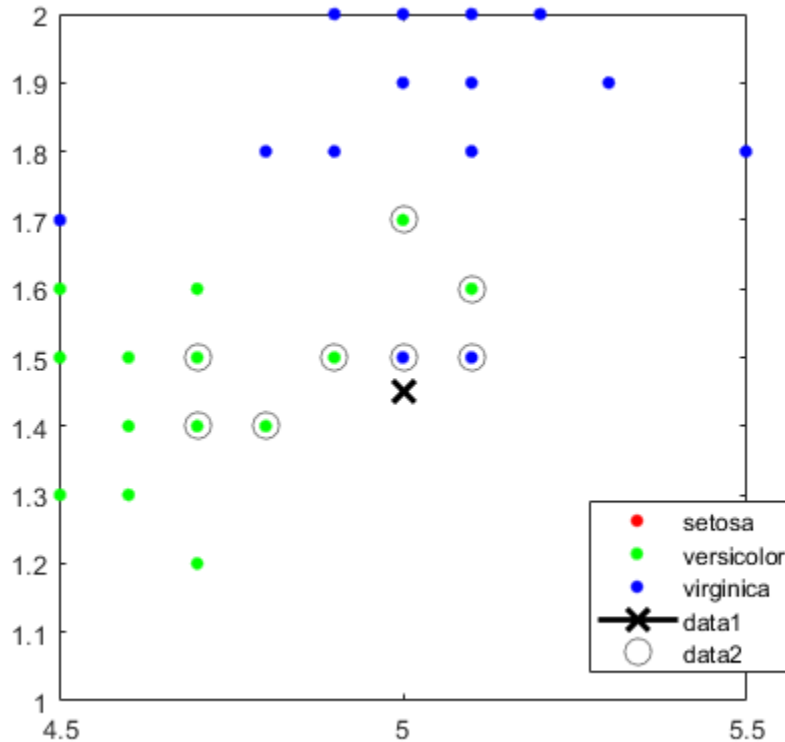
```
x(n, :)
```

```
ans =
```

```
5.0000    1.5000
4.9000    1.5000
4.9000    1.5000
5.1000    1.5000
5.1000    1.6000
4.8000    1.4000
5.0000    1.7000
4.7000    1.4000
4.7000    1.4000
4.7000    1.5000
```

Make the axes equal so the calculated distances correspond to the apparent distances on the plot axis equal and zoom in to see the neighbors better.

```
xlim([4.5 5.5]);
ylim([1 2]);
axis square
```



Find the species of the 10 neighbors.

```
tabulate(species(n))
```

Value	Count	Percent
virginica	2	20.00%
versicolor	8	80.00%

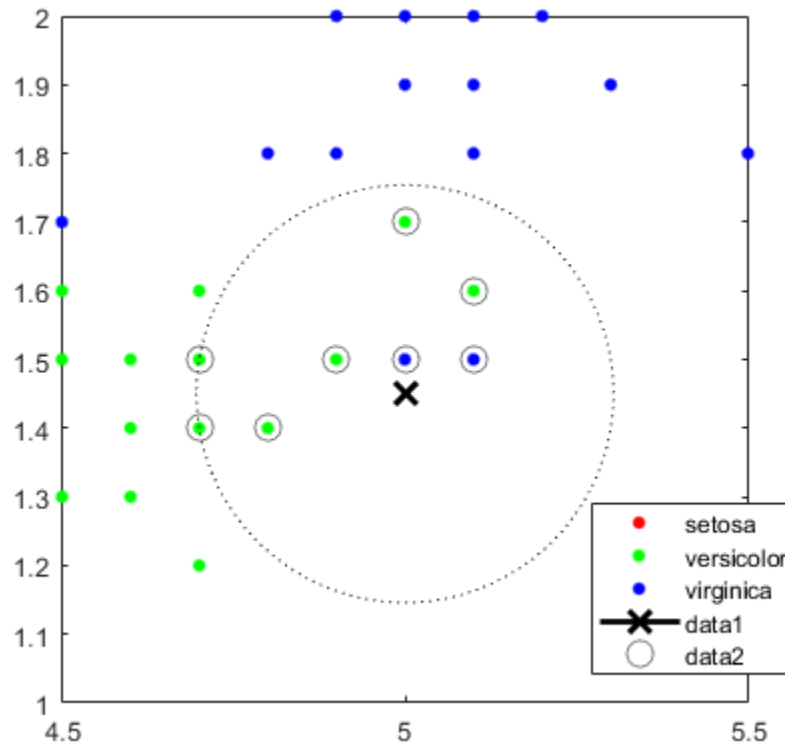
Using a rule based on the majority vote of the 10 nearest neighbors, you can classify this new point as a versicolor.

Visually identify the neighbors by drawing a circle around the group of them. Define the center and diameter of a circle, based on the location of the new point.

```

ctr = newpoint - d(end);
diameter = 2*d(end);
% Draw a circle around the 10 nearest neighbors.
h = rectangle('position',[ctr,diameter,diameter],...
    'curvature',[1 1]);
h.LineStyle = ':';

```



Using the same dataset, find the 10 nearest neighbors to three new points.

```

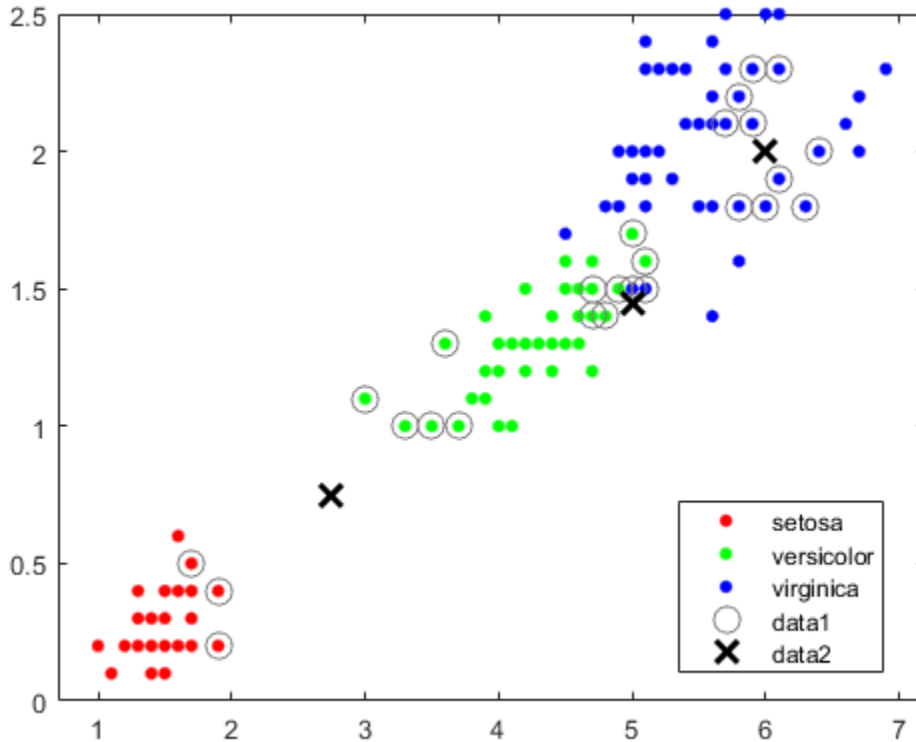
figure
newpoint2 = [5 1.45;6 2;2.75 .75];
gscatter(x(:,1),x(:,2),species)
legend('location','best')
[n2,d2] = knnsearch(Mdl,newpoint2,'k',10);

```

```

line(x(n2,1),x(n2,2),'color',[.5 .5 .5],'marker','o',...
     'linestyle','none','markersize',10)
line(newpoint2(:,1),newpoint2(:,2),'marker','x','color','k',...
     'markersize',10,'linewidth',2,'linestyle','none')

```



Find the species of the 10 nearest neighbors for each new point.

```
tabulate(species(n2(1,:)))
```

Value	Count	Percent
virginica	2	20.00%
versicolor	8	80.00%

```
tabulate(species(n2(2,:)))
```

```
      Value      Count      Percent
virginica         10      100.00%

tabulate(species(n2(3,:)))

      Value      Count      Percent
versicolor         7       70.00%
setosa              3       30.00%
```

For more examples using `knnsearch` methods and function, see the individual reference pages.

## Find Nearest Neighbors Using a Custom Distance Metric

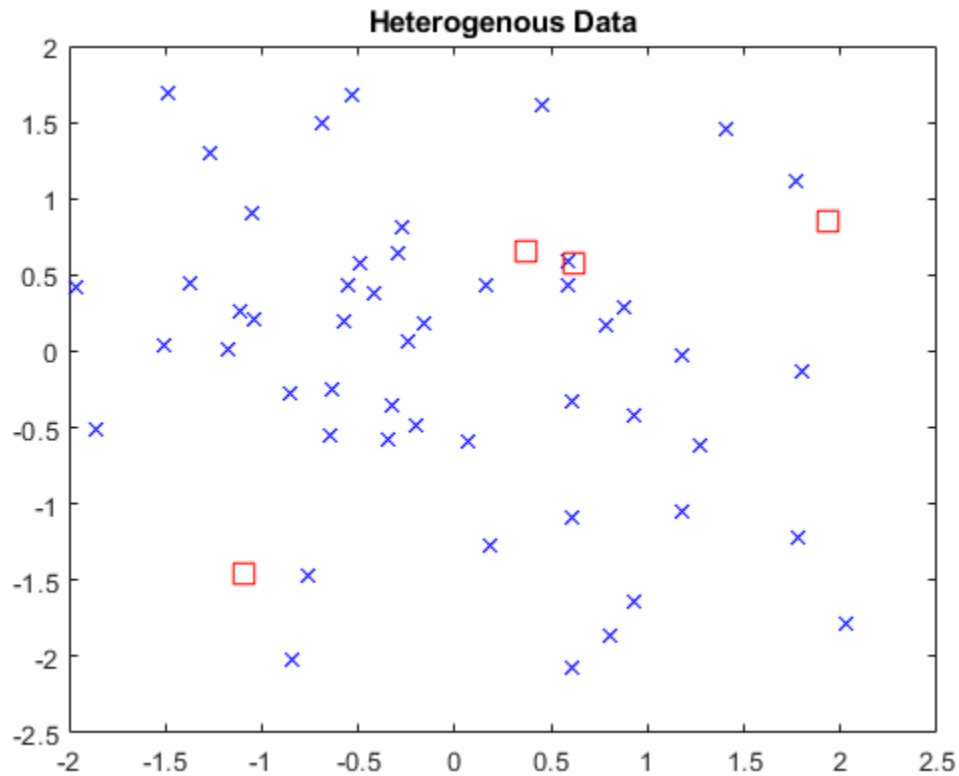
This example shows how to find the indices of the three nearest observations in  $X$  to each observation in  $Y$  with respect to the chi-square distance. This distance metric is used in correspondence analysis, particularly in ecological applications.

Randomly generate normally distributed data into two matrices. The number of rows can vary, but the number of columns must be equal. This example uses 2-D data for plotting.

```
rng(1); % For reproducibility
X = randn(50,2);
Y = randn(4,2);

h = zeros(3,1);
figure;
h(1) = plot(X(:,1),X(:,2),'bx');
hold on;
h(2) = plot(Y(:,1),Y(:,2),'rs','MarkerSize',10);
title('Heterogenous Data')
```





The rows of  $X$  and  $Y$  correspond to observations, and the columns are, in general, dimensions (for example, predictors).

The chi-square distance between  $j$ -dimensional points  $x$  and  $z$  is

$$\chi(x, z) = \sqrt{\sum_{j=1}^J w_j (x_j - z_j)^2},$$

where  $w_j$  is the weight associated with dimension  $j$ .

Choose weights for each dimension, and specify the chi-square distance function. The distance function must:

- Take as input arguments one row of  $X$ , e.g.,  $x$ , and the matrix  $Z$ .
- Compare  $x$  to each row of  $Z$ .
- Return a vector  $D$  of length  $n_z$ , where  $n_z$  is the number of rows of  $Z$ . Each element of  $D$  is the distance between the observation corresponding to  $x$  and the observations corresponding to each row of  $Z$ .

```
w = [0.4; 0.6];  
chiSqrDist = @(x,Z) sqrt((bsxfun(@minus,x,Z).^2)*w);
```

This example uses arbitrary weights for illustration.

Find the indices of the three nearest observations in  $X$  to each observation in  $Y$ .

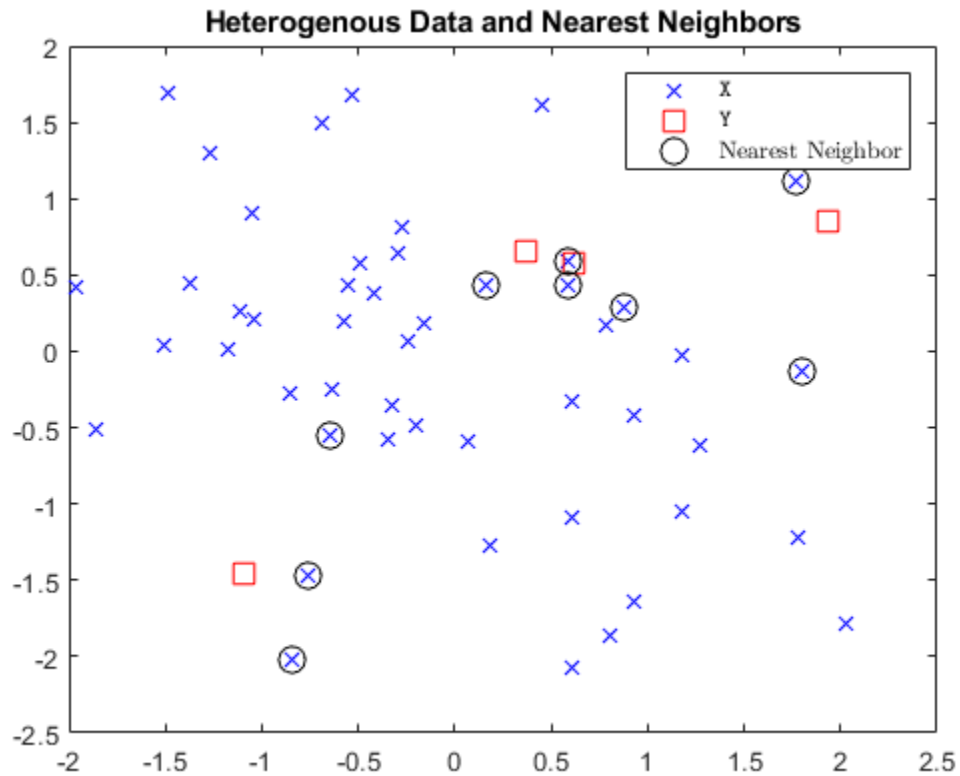
```
k = 3;  
[Idx,D] = knnsearch(X,Y,'Distance',chiSqrDist,'k',k);
```

`idx` and `D` are 4-by-3 matrices.

- `idx(j,1)` is the row index of the closest observation in  $X$  to observation  $j$  of  $Y$ , and `D(j,1)` is their distance.
- `idx(j,2)` is the row index of the next closest observation in  $X$  to observation  $j$  of  $Y$ , and `D(j,2)` is their distance.
- And so on.

Identify the nearest observations in the plot.

```
for j = 1:k;  
    h(3) = plot(X(Idx(:,j),1),X(Idx(:,j),2),'ko','MarkerSize',10);  
end  
legend(h,{'\texttt{X}','\texttt{Y}','Nearest Neighbor'},'Interpreter','latex');  
title('Heterogenous Data and Nearest Neighbors')  
hold off;
```



Several observations of Y share nearest neighbors.

Verify that the chi-square distance metric is equivalent to the Euclidean distance metric, but with an optional scaling parameter.

```
[IdxE,DE] = knnsearch(X,Y,'Distance','seuclidean','k',k,...
    'Scale',1./sqrt(w));
AreDiffIdx = sum(sum(Idx ~= IdxE))
AreDiffDist = sum(sum(abs(D - DE) > eps))
```

```
AreDiffIdx =
```

```
0
```

```
AreDiffDist =  
  
    0
```

The indices and distances between the two implementations of three nearest neighbors are practically equivalent.

## K-Nearest Neighbor Classification for Supervised Learning

The `ClassificationKNN` classification model lets you:

- “Construct KNN Classifier” on page 18-36
- “Examine Quality of KNN Classifier” on page 18-37
- “Predict Classification Using KNN Classifier” on page 18-38
- “Modify KNN Classifier” on page 18-39

Prepare your data for classification according to the procedure in “Steps in Supervised Learning” on page 18-3. Then, construct the classifier using `fitcknn`.

### Construct KNN Classifier

This example shows how to construct a  $k$ -nearest neighbor classifier for the Fisher iris data.

Load the Fisher iris data.

```
load fisheriris  
X = meas;      % Use all data for fitting  
Y = species;  % Response data
```

Construct the classifier using `fitcknn`.

```
Mdl = fitcknn(X,Y)
```

```
Mdl =
```

```
ClassificationKNN
```

```

        ResponseName: 'Y'
    CategoricalPredictors: []
        ClassNames: {'setosa' 'versicolor' 'virginica'}
        ScoreTransform: 'none'
    NumObservations: 150
        Distance: 'euclidean'
    NumNeighbors: 1

```

A default  $k$ -nearest neighbor classifier uses a single nearest neighbor only. Often, a classifier is more robust with more neighbors than that.

Change the neighborhood size of `Mdl` to 4, meaning that `Mdl` classifies using the four nearest neighbors.

```
Mdl.NumNeighbors = 4;
```

## Examine Quality of KNN Classifier

This example shows how to examine the quality of a  $k$ -nearest neighbor classifier using resubstitution and cross validation.

Construct a KNN classifier for the Fisher iris data as in “Construct KNN Classifier” on page 18-36.

```

load fisheriris
X = meas;
Y = species;
rng(10); % For reproducibility
Mdl = fitcknn(X,Y,'NumNeighbors',4);

```

Examine the resubstitution loss, which, by default, is the fraction of misclassifications from the predictions of `Mdl`. (For nondefault cost, weights, or priors, see `ClassificationKNN.loss`).

```
rloss = resubLoss(Mdl)
```

```

rloss =
    0.0400

```

The classifier predicts incorrectly for 4% of the training data.

Construct a cross-validated classifier from the model.

```
CVMdl = crossval(Mdl);
```

Examine the cross-validation loss, which is the average loss of each cross-validation model when predicting on data that is not used for training.

```
kloss = kfoldLoss(CVMdl)
```

```
kloss =
```

```
    0.0333
```

The cross-validated classification accuracy resembles the resubstitution accuracy. Therefore, you can expect `Mdl` to misclassify approximately 4% of new data, assuming that the new data has about the same distribution as the training data.

## Predict Classification Using KNN Classifier

This example shows how to predict classification for a  $k$ -nearest neighbor classifier.

Construct a KNN classifier for the Fisher iris data as in “Construct KNN Classifier” on page 18-36.

```
load fisheriris
X = meas;
Y = species;
Mdl = fitcknn(X,Y,'NumNeighbors',4);
```

Predict the classification of an average flower.

```
flwr = mean(X); % an average flower
flwrClass = predict(Mdl,flwr)
```

```
flwrClass =
```

```
    1x1 cell array
```

```
{'versicolor'}
```

## Modify KNN Classifier

This example shows how to modify a  $k$ -nearest neighbor classifier.

Construct a KNN classifier for the Fisher iris data as in “Construct KNN Classifier” on page 18-36.

```
load fisheriris
X = meas;
Y = species;
Mdl = fitcknn(X,Y,'NumNeighbors',4);
```

Modify the model to use the three nearest neighbors, rather than the default one nearest neighbor.

```
Mdl.NumNeighbors = 3;
```

Compare the resubstitution predictions and cross-validation loss with the new number of neighbors.

```
loss = resubLoss(Mdl)

rng(10); % For reproducibility
CVMdl = crossval(Mdl,'KFold',5);
kloss = kfoldLoss(CVMdl)
```

```
loss =

    0.0400

kloss =

    0.0333
```

In this case, the model with three neighbors has the same cross-validated loss as the model with four neighbors (see “Examine Quality of KNN Classifier” on page 18-37).

Modify the model to use cosine distance instead of the default, and examine the loss. To use cosine distance, you must recreate the model using the exhaustive search method.

```
CMdl = fitcknn(X,Y,'NSMethod','exhaustive','Distance','cosine');  
CMdl.NumNeighbors = 3;  
closs = resubLoss(CMdl)
```

```
closs =  
  
    0.0200
```

The classifier now has lower resubstitution error than before.

Check the quality of a cross-validated version of the new model.

```
CVCMdl = crossval(CMdl);  
kcloss = kfoldLoss(CVCMdl)
```

```
kcloss =  
  
    0.0200
```

CVCMdl has a better cross-validated loss than CVMdl. However, in general, improving the resubstitution error does not necessarily produce a model with better test-sample predictions.

## See Also

ClassificationKNN | ExhaustiveSearcher | KDTreeSearcher | fitcknn



## Framework for Ensemble Learning

Using various methods, you can meld results from many weak learners into one high-quality ensemble predictor. These methods closely follow the same syntax, so you can try different methods with minor changes in your commands.

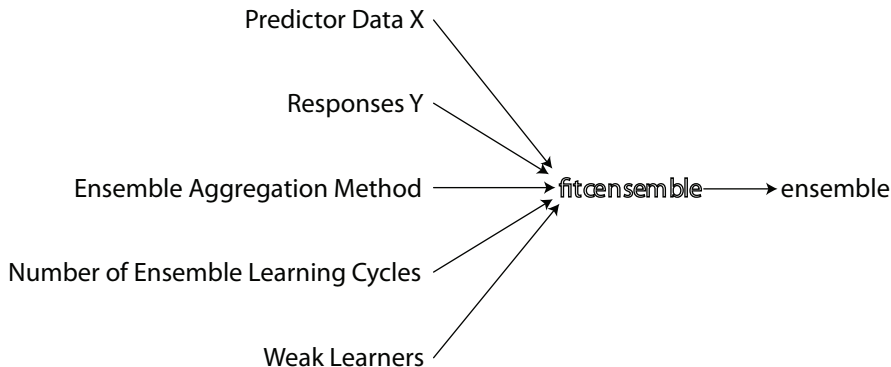
You can create an ensemble for classification by using `fitcensemble` or for regression by using `fitrensemble`.

To train an ensemble for classification using `fitcensemble`, use this syntax.

```
ens = fitcensemble(X,Y,Name,Value)
```

- `X` is the matrix of data. Each row contains one observation, and each column contains one predictor variable.
- `Y` is the vector of responses, with the same number of observations as the rows in `X`.
- `Name, Value` specify additional options using one or more name-value pair arguments. For example, you can specify the ensemble aggregation method with the `'Method'` argument, the number of ensemble learning cycles with the `'NumLearningCycles'` argument, and the type of weak learners with the `'Learners'` argument. For a complete list of name-value pair arguments, see the `fitcensemble` function page.

This figure shows the information you need to create a classification ensemble.



Similarly, you can train an ensemble for regression by using `fitrensemble`, which follows the same syntax as `fitcensemble`. For details on the input arguments and name-value pair arguments, see the `fitrensemble` function page.

For all classification or nonlinear regression problems, follow these steps to create an ensemble:

1. “Prepare the Predictor Data” on page 18-42
2. “Prepare the Response Data” on page 18-42
3. “Choose an Applicable Ensemble Aggregation Method” on page 18-43
4. “Set the Number of Ensemble Members” on page 18-46
5. “Prepare the Weak Learners” on page 18-47
6. “Call `fitcensemble` or `fitrensemble`” on page 18-49

## Prepare the Predictor Data

All supervised learning methods start with predictor data, usually called  $X$  in this documentation.  $X$  can be stored in a matrix or a table. Each row of  $X$  represents one observation, and each column of  $X$  represents one variable or predictor.

## Prepare the Response Data

You can use a wide variety of data types for the response data.

- For regression ensembles,  $Y$  must be a numeric vector with the same number of elements as the number of rows of  $X$ .
- For classification ensembles,  $Y$  can be any of the following data types. This table also contains the method of including missing entries.

Data Type	Missing Entry
Numeric vector	NaN
Categorical vector	<undefined>
Character array	Row of spaces
Cell array of character vectors	' '
Logical vector	(not possible to represent)

`fitcensemble` and `fitrensemble` ignore missing values in  $Y$  when creating an ensemble.

For example, suppose your response data consists of three observations in the following order: `true`, `false`, `true`. You could express  $Y$  as:

- `[1;0;1]` (numeric vector)
- `nominal({'true','false','true'})` (categorical vector)
- `[true;false;true]` (logical vector)
- `['true ','false ','true ']` (character array, padded with spaces so each row has the same length)
- `{'true','false','true'}` (cell array of character vectors)

Use whichever data type is most convenient. Because you cannot represent missing values with logical entries, do not use logical entries when you have missing values in  $Y$ .

## Choose an Applicable Ensemble Aggregation Method

To create classification and regression ensembles with `fitcensemble` and `fitrensemble`, respectively, choose appropriate algorithms from this list.

- For classification with two classes:
  - `'AdaBoostM1'`
  - `'LogitBoost'`
  - `'GentleBoost'`
  - `'RobustBoost'` (requires Optimization Toolbox)
  - `'LPBoost'` (requires Optimization Toolbox)
  - `'TotalBoost'` (requires Optimization Toolbox)
  - `'RUSBoost'`
  - `'Subspace'`
  - `'Bag'`
- For classification with three or more classes:
  - `'AdaBoostM2'`

- 'LPBoost' (requires Optimization Toolbox)
- 'TotalBoost' (requires Optimization Toolbox)
- 'RUSBoost'
- 'Subspace'
- 'Bag'
- For regression:
  - 'LSBoost'
  - 'Bag'

For descriptions of the various algorithms, see “Ensemble Algorithms” on page 18-51.

See “Suggestions for Choosing an Appropriate Ensemble Algorithm” on page 18-45.

This table lists characteristics of the various algorithms. In the table titles:

- **Regress.** — Regression
- **Classif.** — Classification
- **Preds.** — Predictors
- **Imbalance** — Good for imbalanced data (one class has many more observations than the other)
- **Stop** — Algorithm self-terminates
- **Sparse** — Requires fewer weak learners than other ensemble algorithms

Algorithm	Regress.	Binary Classif.	Binary Classif. Multi-Level Preds.	Classif. 3+ Classes	Class Imbalance	Stop	Sparse
Bag	×	×		×			
AdaBoostM1		×					
AdaBoostM2				×			
LogitBoost		×	×				
GentleBoost		×	×				
RobustBoost		×					

Algorithm	Regress.	Binary Classif.	Binary Classif. Multi-Level Preds.	Classif. 3+ Classes	Class Imbalance	Stop	Sparse
LPBoost		×		×		×	×
TotalBoost		×		×		×	×
RUSBoost		×		×	×		
LSBoost	×						
Subspace		×		×			

RobustBoost, LPBoost, and TotalBoost require an Optimization Toolbox license. Try TotalBoost before LPBoost, as TotalBoost can be more robust.

### Suggestions for Choosing an Appropriate Ensemble Algorithm

- **Regression** — Your choices are LSBoost or Bag. See “General Characteristics of Ensemble Algorithms” on page 18-46 for the main differences between boosting and bagging.
- **Binary Classification** — Try AdaBoostM1 first, with these modifications:

Data Characteristic	Recommended Algorithm
Many predictors	Subspace
Skewed data (many more observations of one class)	RUSBoost
Categorical predictors with over 31 levels	LogitBoost or GentleBoost
Label noise (some training data has the wrong class)	RobustBoost
Many observations	Avoid LPBoost, TotalBoost, and Bag

- **Multiclass Classification** — Try AdaBoostM2 first, with these modifications:

Data Characteristic	Recommended Algorithm
Many predictors	Subspace
Skewed data (many more observations of one class)	RUSBoost

Data Characteristic	Recommended Algorithm
Many observations	Avoid <code>LPBoost</code> , <code>TotalBoost</code> , and <code>Bag</code>

For details of the algorithms, see “Ensemble Algorithms” on page 18-51.

### General Characteristics of Ensemble Algorithms

- `Bag` generally constructs deep trees. This construction is both time consuming and memory-intensive. This also leads to relatively slow predictions.
- `Boost` algorithms generally use very shallow trees. This construction uses relatively little time or memory. However, for effective predictions, boosted trees might need more ensemble members than bagged trees. Therefore it is not always clear which class of algorithms is superior.
- `Bag` can estimate the generalization error without additional cross validation. See `oobLoss`.
- Except for `Subspace`, all boosting and bagging algorithms are based on decision tree on page 19-2 learners. `Subspace` can use either discriminant analysis on page 20-2 or  $k$ -nearest neighbor on page 18-16 learners.

For details of the characteristics of individual ensemble members, see “Characteristics of Classification Algorithms” on page 18-7.

### Set the Number of Ensemble Members

Choosing the size of an ensemble involves balancing speed and accuracy.

- Larger ensembles take longer to train and to generate predictions.
- Some ensemble algorithms can become overtrained (inaccurate) when too large.

To set an appropriate size, consider starting with several dozen to several hundred members in an ensemble, training the ensemble, and then checking the ensemble quality, as in “Test Ensemble Quality” on page 18-81. If it appears that you need more members, add them using the `resume` method (classification) or the `resume` method (regression). Repeat until adding more members does not improve ensemble quality.

---

**Tip** For classification, the `LPBoost` and `TotalBoost` algorithms are self-terminating, meaning you do not have to investigate the appropriate ensemble size. Try setting `NumLearningCycles` to 500. The algorithms usually terminate with fewer members.

---

## Prepare the Weak Learners

Currently the weak learner types are:

- 'Discriminant' (recommended for Subspace ensemble)
- 'KNN' (only for Subspace ensemble)
- 'Tree' (for any ensemble except Subspace)

There are two ways to set the weak learner type in an ensemble.

- To create an ensemble with default weak learner options, specify the value of the 'Learners' name-value pair argument as the character vector of the weak learner name. For example:

```
ens = fitcensemble(X,Y,'Method','Subspace','NumLearningCycles',50,'Learners','KNN');
% or
ens = fitensemble(X,Y,'Method','Bag','NumLearningCycles',50,'Learners','Tree');
```

- To create an ensemble with nondefault weak learner options, create a nondefault weak learner using the appropriate template method.

For example, if you have missing data, and want to use classification trees with surrogate splits for better accuracy:

```
templ = templateTree('Surrogate','all');
ens = fitcensemble(X,Y,'Method','AdaBoostM2','NumLearningCycles',50,'Learners',templ);
```

To grow trees with leaves containing a number of observations that is at least 10% of the sample size:

```
templ = templateTree('MinLeafSize',size(X,1)/10);
ens = fitcensemble(X,Y,'Method','AdaBoostM2','NumLearningCycles',50,'Learners',templ);
```

Alternatively, choose the maximal number of splits per tree:

```
templ = templateTree('MaxNumSplits',4);
ens = fitcensemble(X,Y,'Method','AdaBoostM2','NumLearningCycles',50,'Learners',templ);
```

You can also use nondefault weak learners in `fitensemble`.

While you can give `fitcensemble` and `fitensemble` a cell array of learner templates, the most common usage is to give just one weak learner template.

For examples using a template, see “Train Ensemble With Unequal Classification Costs” on page 18-108 and “Surrogate Splits” on page 18-121.

Decision trees can handle NaN values in  $X$ . Such values are called “missing”. If you have some missing values in a row of  $X$ , a decision tree finds optimal splits using nonmissing values only. If an entire row consists of NaN, `fitcensemble` and `fitrensemble` ignore that row. If you have data with a large fraction of missing values in  $X$ , use surrogate decision splits. For examples of surrogate splits, see “Train Ensemble With Unequal Classification Costs” on page 18-108 and “Surrogate Splits” on page 18-121.

### Common Settings for Tree Weak Learners

- The depth of a weak learner tree makes a difference for training time, memory usage, and predictive accuracy. You control the depth these parameters:
  - `MaxNumSplits` — The maximal number of branch node splits is `MaxNumSplits` per tree. Set large values of `MaxNumSplits` to get deep trees. The default for bagging is  $\text{size}(X, 1) - 1$ . The default for boosting is 1.
  - `MinLeafSize` — Each leaf has at least `MinLeafSize` observations. Set small values of `MinLeafSize` to get deep trees. The default for classification is 1 and 5 for regression.
  - `MinParentSize` — Each branch node in the tree has at least `MinParentSize` observations. Set small values of `MinParentSize` to get deep trees. The default for classification is 2 and 10 for regression.

If you supply both `MinParentSize` and `MinLeafSize`, the learner uses the setting that gives larger leaves (shallower trees):

```
MinParent = max(MinParent, 2*MinLeaf)
```

If you additionally supply `MaxNumSplits`, then the software splits a tree until one of the three splitting criteria is satisfied.

- `Surrogate` — Grow decision trees with surrogate splits when `Surrogate` is 'on'. Use surrogate splits when your data has missing values.

---

**Note** Surrogate splits cause slower training and use more memory.

---

- `PredictorSelection` — `fitcensemble`, `fitrensemble`, and `TreeBagger` grow trees using the standard CART algorithm [11] by default. If the predictor variables



are heterogeneous or there are predictors having many levels and other having few levels, then standard CART tends to select predictors having many levels as split predictors. For split-predictor selection that is robust to the number of levels that the predictors have, consider specifying 'curvature' or 'interaction-curvature'. These specifications conduct chi-square tests of association between each predictor and the response or each pair of predictors and the response, respectively. The predictor that yields the minimal  $p$ -value is the split predictor for a particular node. For more details, see “Choose Split Predictor Selection Technique” on page 19-19.

---

**Note** When boosting decision trees, selecting split predictors using the curvature or interaction tests is not recommended.

---

## Call `fitcensemble` or `fitrensemble`

The syntaxes for `fitcensemble` and `fitrensemble` are identical. For `fitrensemble`, the syntax is:

```
ens = fitrensemble(X,Y,Name,Value)
```

- `X` is the matrix of data. Each row contains one observation, and each column contains one predictor variable.
- `Y` is the responses, with the same number of observations as rows in `X`.
- `Name, Value` specify additional options using one or more name-value pair arguments. For example, you can specify the ensemble aggregation method with the 'Method' argument, the number of ensemble learning cycles with the 'NumLearningCycles' argument, and the type of weak learners with the 'Learners' argument. For a complete list of name-value pair arguments, see the `fitrensemble` function page.

The result of `fitrensemble` and `fitcensemble` is an ensemble object, suitable for making predictions on new data. For a basic example of creating a regression ensemble, see “Train Regression Ensemble” on page 18-70. For a basic example of creating a classification ensemble, see “Train Classification Ensemble” on page 18-67.

### Where to Set Name-Value Pairs

There are several name-value pairs you can pass to `fitcensemble` or `fitrensemble`, and several that apply to the weak learners (`templateDiscriminant`, `templateKNN`,

and `templateTree`). To determine which name-value pair argument is appropriate, the ensemble or the weak learner:

- Use template name-value pairs to control the characteristics of the weak learners.
- Use `fitcensemble` or `fitrensemble` name-value pair arguments to control the ensemble as a whole, either for algorithms or for structure.

For example, for an ensemble of boosted classification trees with each tree deeper than the default, set the `templateTree` name-value pair arguments `MinLeafSize` and `MinParentSize` to smaller values than the defaults. Or, `MaxNumSplits` to a larger value than the defaults. The trees are then leafier (deeper).

To name the predictors in a classification ensemble (part of the structure of the ensemble), use the `PredictorNames` name-value pair in `fitcensemble`.

## See Also

`fitcensemble` | `fitrensemble` | `oobLoss` | `resume` | `resume` | `templateDiscriminant` | `templateKNN` | `templateTree`

## Related Examples

- “Train Classification Ensemble” on page 18-67
- “Train Regression Ensemble” on page 18-70
- “Ensemble Algorithms” on page 18-51
- “Decision Trees” on page 19-2
- “Choose Split Predictor Selection Technique” on page 19-19

## Ensemble Algorithms

You can use various algorithms for ensemble learning. Some of the algorithms apply only to classification ensembles, and others apply only to regression ensembles. You can use the algorithms to create ensembles by specifying a value for the 'Method' name-value pair argument of `fitcensemble` or `fitrensemble`. This topic provides descriptions of the various algorithms. Note that usage of some algorithms, such as `LPBoost`, `TotalBoost`, and `RobustBoost`, requires an Optimization Toolbox license.

### In this section...

“AdaBoostM1” on page 18-51

“AdaBoostM2” on page 18-53

“Bag” on page 18-54

“GentleBoost” on page 18-58

“LogitBoost” on page 18-58

“LPBoost” on page 18-60

“LSBoost” on page 18-61

“RobustBoost” on page 18-61

“RUSBoost” on page 18-62

“Subspace” on page 18-63

“TotalBoost” on page 18-64

### AdaBoostM1

`AdaBoostM1` is a very popular boosting algorithm for binary classification. The algorithm trains learners sequentially. For every learner with index  $t$ , `AdaBoostM1` computes the weighted classification error

$$\varepsilon_t = \sum_{n=1}^N d_n^{(t)} \mathbb{I}(y_n \neq h_t(x_n)),$$

where

- $x_n$  is a vector of predictor values for observation  $n$ .

- $y_n$  is the true class label.
- $h_t$  is the prediction of learner (hypothesis) with index  $t$ .
- $\mathbb{I}$  is the indicator function.
- $d_n^{(t)}$  is the weight of observation  $n$  at step  $t$ .

AdaBoostM1 then increases weights for observations misclassified by learner  $t$  and reduces weights for observations correctly classified by learner  $t$ . The next learner  $t + 1$  is then trained on the data with updated weights  $d_n^{(t+1)}$ .

After training finishes, AdaBoostM1 computes prediction for new data using

$$f(x) = \sum_{t=1}^T \alpha_t h_t(x),$$

where

$$\alpha_t = \frac{1}{2} \log \frac{1 - \varepsilon_t}{\varepsilon_t}$$

are weights of the weak hypotheses in the ensemble.

Training by AdaBoostM1 can be viewed as stagewise minimization of the exponential loss

$$\sum_{n=1}^N w_n \exp(-y_n f(x_n)),$$

where

- $y_n \in \{-1, +1\}$  is the true class label.
- $w_n$  are observation weights normalized to add up to 1.
- $f(x_n) \in (-\infty, +\infty)$  is the predicted classification score.

The observation weights  $w_n$  are the original observation weights you passed to `fitcensemble`.

The second output from the `predict` method of an AdaBoostM1 classification ensemble is an  $N$ -by-2 matrix of classification scores for the two classes and  $N$  observations. The

second column in this matrix is always equal to minus the first column. The `predict` method returns two scores to be consistent with multiclass models, though this is redundant because the second column is always the negative of the first.

Most often `AdaBoostM1` is used with decision stumps (default) or shallow trees. If boosted stumps give poor performance, try setting the minimal parent node size to one quarter of the training data.

By default, the learning rate for boosting algorithms is 1. If you set the learning rate to a lower number, the ensemble learns at a slower rate, but can converge to a better solution. 0.1 is a popular choice for the learning rate. Learning at a rate less than 1 is often called “shrinkage”.

For examples using `AdaBoostM1`, see “Tune `RobustBoost`” on page 18-131.

For references related to `AdaBoostM1`, see Freund and Schapire [20], Schapire et al. [40], Friedman, Hastie, and Tibshirani [22], and Friedman [21].

## AdaBoostM2

`AdaBoostM2` is an extension of `AdaBoostM1` for multiple classes. Instead of weighted classification error, `AdaBoostM2` uses weighted pseudo-loss for  $N$  observations and  $K$  classes

$$\varepsilon_t = \frac{1}{2} \sum_{n=1}^N \sum_{k \neq y_n} d_{n,k}^{(t)} (1 - h_t(x_n, y_n) + h_t(x_n, k)),$$

where

- $h_t(x_n, k)$  is the confidence of prediction by learner at step  $t$  into class  $k$  ranging from 0 (not at all confident) to 1 (highly confident).
- $d_{n,k}^{(t)}$  are observation weights at step  $t$  for class  $k$ .
- $y_n$  is the true class label taking one of the  $K$  values.
- The second sum is over all classes other than the true class  $y_n$ .

Interpreting the pseudo-loss is harder than classification error, but the idea is the same. Pseudo-loss can be used as a measure of the classification accuracy from any learner in an ensemble. Pseudo-loss typically exhibits the same behavior as a weighted

classification error for `AdaBoostM1`: the first few learners in a boosted ensemble give low pseudo-loss values. After the first few training steps, the ensemble begins to learn at a slower pace, and the pseudo-loss value approaches 0.5 from below.

For examples using `AdaBoostM2`, see “Train Classification Ensemble” on page 18-67.

For references related to `AdaBoostM2`, see Freund and Schapire [20].

## Bag

Bagging, which stands for “bootstrap aggregation,” is a type of ensemble learning. To bag a weak learner such as a decision tree on a dataset, generate many bootstrap replicas of this dataset and grow decision trees on these replicas. Obtain each bootstrap replica by randomly selecting  $N$  observations out of  $N$  with replacement, where  $N$  is the dataset size. To find the predicted response of a trained ensemble, take an average over predictions from individual trees.

Bagged decision trees were introduced in MATLAB R2009a as `TreeBagger`. The `fitcensemble` and `fitrensemble` functions let you bag in a manner consistent with boosting. An ensemble of bagged trees, either `ClassificationBaggedEnsemble` or `RegressionBaggedEnsemble`, returned by `fitcensemble` or `fitrensemble`, respectively, offers almost the same functionality as `TreeBagger`. Discrepancies between `TreeBagger` and the new framework are described in detail in `TreeBagger Features Not in fitcensemble or fitrensemble`.

Bagging works by training learners on resampled versions of the data. This resampling is usually done by bootstrapping observations, that is, selecting  $N$  out of  $N$  observations with replacement for every new learner. In addition, every tree in the ensemble can randomly select predictors for decision splits—a technique known to improve the accuracy of bagged trees.

By default, the minimal leaf sizes for bagged trees are set to 1 for classification and 5 for regression. Trees grown with the default leaf size are usually very deep. These settings are close to optimal for the predictive power of an ensemble. Often you can grow trees with larger leaves without losing predictive power. Doing so reduces training and prediction time, as well as memory usage for the trained ensemble.

Another important parameter is the number of predictors selected at random for every decision split. This random selection is made for every split, and every deep tree involves many splits. By default, this parameter is set to a square root of the number of predictors for classification, and one third of predictors for regression.

Several features of bagged decision trees make them a unique algorithm. Drawing  $N$  out of  $N$  observations with replacement omits on average 37% of observations for each decision tree. These are “out-of-bag” observations. You can use them to estimate the predictive power and feature importance. For each observation, you can estimate the out-of-bag prediction by averaging over predictions from all trees in the ensemble for which this observation is out of bag. You can then compare the computed prediction against the observed response for this observation. By comparing the out-of-bag predicted responses against the observed responses for all observations used for training, you can estimate the average out-of-bag error. This out-of-bag average is an unbiased estimator of the true ensemble error. You can also obtain out-of-bag estimates of feature importance by randomly permuting out-of-bag data across one variable or column at a time and estimating the increase in the out-of-bag error due to this permutation. The larger the increase, the more important the feature. Thus, you need not supply test data for bagged ensembles because you obtain reliable estimates of the predictive power and feature importance in the process of training, which is an attractive feature of bagging.

Another attractive feature of bagged decision trees is the proximity matrix. Every time two observations land on the same leaf of a tree, their proximity increases by 1. For normalization, sum these proximities over all trees in the ensemble and divide by the number of trees. The resulting matrix is symmetric with diagonal elements equal to 1 and off-diagonal elements ranging from 0 to 1. You can use this matrix for finding outlier observations and discovering clusters in the data through multidimensional scaling.

For examples using bagging, see:

- “Test Ensemble Quality” on page 18-81
- “Surrogate Splits” on page 18-121
- “Regression of Insurance Risk Rating for Car Imports Using TreeBagger” on page 18-141
- “Classifying Radar Returns for Ionosphere Data Using TreeBagger” on page 18-154

For references related to bagging, see Breiman [8], [9], and [10].

### **Comparison of TreeBagger and Bagged Ensembles**

`fitcensemble` and `fitrensemble` produce bagged ensembles that have most, but not all, of the functionality of `TreeBagger` objects. Additionally, some functionalities have different names in the new bagged ensembles.

**TreeBagger Features Not in fitensemble or fitrensemble**

Feature	TreeBagger Property	TreeBagger Method
Computation of proximity matrix	Proximity	fillprox, mdsprox
Computation of outliers	OutlierMeasure	N/A
Out-of-bag estimates of predictor importance	OOBPermutedPredictorDeltaMean Margin and OOBPermutedPredictorCountRaiseMargin	N/A
Merging two ensembles trained separately	N/A	append
Parallel computation for creating ensemble	Set the UseParallel name-value pair to true	N/A

---

**Note** When you estimate the proximity matrix and outliers of a `TreeBagger` model using `fillprox`, MATLAB must fit an  $n$ -by- $n$  matrix in memory, where  $n$  is the number of observations. Therefore, if  $n$  is moderate to large, then you should avoid estimating the proximity matrix and outliers.

---



### Differing Names Between TreeBagger and Bagged Ensembles

Feature	TreeBagger	Bagged Ensembles
Split criterion contributions for each predictor	DeltaCriterionDecisionSplit property	First output of predictorImportance (classification) or predictorImportance (regression)
Predictor associations	SurrogateAssociation property	Second output of predictorImportance (classification) or predictorImportance (regression)
Out-of-bag estimates of predictor importance	OOBPermutedPredictorDeltaError property	Output of oobPermutedPredictorImportance (classification) or oobPermutedPredictorImportance (regression)
Error (misclassification probability or mean-squared error)	error and oobError methods	loss and oobLoss methods (classification); loss and oobLoss methods (regression)
Train additional trees and add to ensemble	growTrees method	resume method (classification); resume method (regression)
Mean classification margin per tree	meanMargin and oobMeanMargin methods	edge and oobEdge methods (classification)

In addition, two important changes were made to training and prediction for bagged classification ensembles:

- If you pass a misclassification cost matrix to `TreeBagger`, it passes this matrix along to the trees. If you pass a misclassification cost matrix to `fitcensemble`, it uses this matrix to adjust the class prior probabilities. `fitcensemble` then passes the adjusted prior probabilities and the default cost matrix to the trees. The default cost matrix is  $\text{ones}(K) - \text{eye}(K)$  for  $K$  classes.
- Unlike the `loss` and `edge` methods in the new framework, the `TreeBagger` `error` and `meanMargin` methods do not normalize input observation weights of the prior probabilities in the respective class.

## GentleBoost

GentleBoost (also known as Gentle AdaBoost) combines features of AdaBoostM1 and LogitBoost. Like AdaBoostM1, GentleBoost minimizes the exponential loss. But its numeric optimization is set up differently. Like LogitBoost, every weak learner fits a regression model to response values  $y_n \in \{-1, +1\}$ . This makes GentleBoost another good candidate for binary classification of data with multilevel categorical predictors.

`fitcensemble` computes and stores the mean-squared error in the `FitInfo` property of the ensemble object. The mean-squared error is

$$\sum_{n=1}^N d_n^{(t)} (\tilde{y}_n - h_t(x_n))^2,$$

where

- $d_n^{(t)}$  are observation weights at step  $t$  (the weights add up to 1).
- $h_t(x_n)$  are predictions of the regression model  $h_t$  fitted to response values  $y_n$ .

As the strength of individual learners weakens, the weighted mean-squared error approaches 1.

For examples using GentleBoost, see “Train Ensemble With Unequal Classification Costs” on page 18-108 and “Classification with Many Categorical Levels” on page 18-115.

For references related to GentleBoost, see Friedman, Hastie, and Tibshirani [22].

## LogitBoost

LogitBoost is another popular algorithm for binary classification. LogitBoost works similarly to AdaBoostM1, except it minimizes binomial deviance

$$\sum_{n=1}^N w_n \log(1 + \exp(-2y_n f(x_n))),$$

where

- $y_n \in \{-1, +1\}$  is the true class label.

- $w_n$  are observation weights normalized to add up to 1.
- $f(x_n) \in (-\infty, +\infty)$  is the predicted classification score.

Binomial deviance assigns less weight to badly misclassified observations (observations with large negative values of  $y_n f(x_n)$ ). `LogitBoost` can give better average accuracy than `AdaBoostM1` for data with poorly separable classes.

Learner  $t$  in a `LogitBoost` ensemble fits a regression model to response values

$$\tilde{y}_n = \frac{y_n^* - p_t(x_n)}{p_t(x_n)(1 - p_t(x_n))},$$

where

- $y_n^* \in \{0, +1\}$  are relabeled classes (0 instead of  $-1$ ).
- $p_t(x_n)$  is the current ensemble estimate of the probability for observation  $x_n$  to be of class 1.

Fitting a regression model at each boosting step turns into a great computational advantage for data with multilevel categorical predictors. Take a categorical predictor with  $L$  levels. To find the optimal decision split on such a predictor, a classification tree needs to consider  $2^{L-1} - 1$  splits. A regression tree needs to consider only  $L - 1$  splits, so the processing time can be much shorter. `LogitBoost` is recommended for categorical predictors with many levels.

`fitensemble` computes and stores the mean-squared error in the `FitInfo` property of the ensemble object. The mean-squared error is

$$\sum_{n=1}^N d_n^{(t)} (\tilde{y}_n - h_t(x_n))^2,$$

where

- $d_n^{(t)}$  are observation weights at step  $t$  (the weights add up to 1).
- $h_t(x_n)$  are predictions of the regression model  $h_t$  fitted to response values  $\tilde{y}_n$ .

Values  $y_n$  can range from  $-\infty$  to  $+\infty$ , so the mean-squared error does not have well-defined bounds.

For examples using `LogitBoost`, see “Classification with Many Categorical Levels” on page 18-115.

For references related to `LogitBoost`, see Friedman, Hastie, and Tibshirani [22].

## LPBoost

`LPBoost` (linear programming boost), like `TotalBoost`, performs multiclass classification by attempting to maximize the minimal margin in the training set. This attempt uses optimization algorithms, namely linear programming for `LPBoost`. So you need an Optimization Toolbox license to use `LPBoost` or `TotalBoost`.

The margin of a classification is the difference between the predicted soft classification score for the true class, and the largest score for the false classes. For trees, the score of a classification of a leaf node is the posterior probability of the classification at that node. The posterior probability of the classification at a node is the number of training sequences that lead to that node with the classification, divided by the number of training sequences that lead to that node. For more information, see “Definitions” on page 32-3937 in `margin`.

Why maximize the minimal margin? For one thing, the generalization error (the error on new data) is the probability of obtaining a negative margin. Schapire and Singer [41] establish this inequality on the probability of obtaining a negative margin:

$$P_{\text{test}}(m \leq 0) \leq P_{\text{train}}(m \leq \theta) + O\left(\frac{1}{\sqrt{N}} \sqrt{\frac{V \log^2(N/V)}{\theta^2} + \log(1/\delta)}\right).$$

Here  $m$  is the margin,  $\theta$  is any positive number,  $V$  is the Vapnik-Chervonenkis dimension of the classifier space,  $N$  is the size of the training set, and  $\delta$  is a small positive number. The inequality holds with probability  $1-\delta$  over many i.i.d. training and test sets. This inequality says: To obtain a low generalization error, minimize the number of observations below margin  $\theta$  in the training set.

`LPBoost` iteratively maximizes the minimal margin through a sequence of linear programming problems. Equivalently, by duality, `LPBoost` minimizes the maximal edge, where edge is the weighted mean margin (see “Definitions” on page 32-1551). At each iteration, there are more constraints in the problem. So, for large problems, the optimization problem becomes increasingly constrained, and slow to solve.

`LPBoost` typically creates ensembles with many learners having weights that are orders of magnitude smaller than those of other learners. Therefore, to better enable you to

remove the unimportant ensemble members, the `compact` method reorders the members of an `LPBoost` ensemble from largest weight to smallest. Therefore, you can easily remove the least important members of the ensemble using the `removeLearners` method.

For examples using `LPBoost`, see “`LPBoost` and `TotalBoost` for Small Ensembles” on page 18-124.

For references related to `LPBoost`, see Warmuth, Liao, and Ratsch [44].

## LSBoost

`LSBoost` (least squares boosting) fits regression ensembles. At every step, the ensemble fits a new learner to the difference between the observed response and the aggregated prediction of all learners grown previously. The ensemble fits to minimize mean-squared error.

You can use `LSBoost` with shrinkage by passing in the `LearnRate` parameter. By default this parameter is set to 1, and the ensemble learns at the maximal speed. If you set `LearnRate` to a value from 0 to 1, the ensemble fits every new learner to  $y_n - \eta f(x_n)$ , where

- $y_n$  is the observed response.
- $f(x_n)$  is the aggregated prediction from all weak learners grown so far for observation  $x_n$ .
- $\eta$  is the learning rate.

For examples using `LSBoost`, see “Train Regression Ensemble” on page 18-70 and “Regularize a Regression Ensemble” on page 18-89.

For references related to `LSBoost`, see Hastie, Tibshirani, and Friedman [24], Chapters 7 (Model Assessment and Selection) and 15 (Random Forests, see also [9]).

## RobustBoost

Boosting algorithms such as `AdaBoostM1` and `LogitBoost` increase weights for misclassified observations at every boosting step. These weights can become very large. If this happens, the boosting algorithm sometimes concentrates on a few misclassified observations and neglects the majority of training data. Consequently the average

classification accuracy suffers. You need an Optimization Toolbox license to use `RobustBoost`.

In this situation, you can try using `RobustBoost`. This algorithm does not assign almost the entire data weight to badly misclassified observations. It can produce better average classification accuracy.

Unlike `AdaBoostM1` and `LogitBoost`, `RobustBoost` does not minimize a specific loss function. Instead, it maximizes the number of observations with the classification margin above a certain threshold.

`RobustBoost` trains based on time evolution. The algorithm starts at  $t = 0$ . At every step, `RobustBoost` solves an optimization problem to find a positive step in time  $\Delta t$  and a corresponding positive change in the average margin for training data  $\Delta m$ . `RobustBoost` stops training and exits if at least one of these three conditions is true:

- Time  $t$  reaches 1.
- `RobustBoost` cannot find a solution to the optimization problem with positive updates  $\Delta t$  and  $\Delta m$ .
- `RobustBoost` grows as many learners as you requested.

Results from `RobustBoost` can be usable for any termination condition. Estimate the classification accuracy by cross validation or by using an independent test set.

To get better classification accuracy from `RobustBoost`, you can adjust three parameters in `fitcensemble`: `RobustErrorGoal`, `RobustMaxMargin`, and `RobustMarginSigma`. Start by varying values for `RobustErrorGoal` from 0 to 1. The maximal allowed value for `RobustErrorGoal` depends on the two other parameters. If you pass a value that is too high, `fitcensemble` produces an error message showing the allowed range for `RobustErrorGoal`.

For examples using `RobustBoost`, see “Tune `RobustBoost`” on page 18-131.

For references related to `RobustBoost`, see Freund [19].

## RUSBoost

`RUSBoost` is especially effective at classifying imbalanced data, meaning some class in the training data has many fewer members than another. RUS stands for Random Under

Sampling. The algorithm takes  $N$ , the number of members in the class with the fewest members in the training data, as the basic unit for sampling. Classes with more members are under sampled by taking only  $N$  observations of every class. In other words, if there are  $K$  classes, then, for each weak learner in the ensemble, `RUSBoost` takes a subset of the data with  $N$  observations from each of the  $K$  classes. The boosting procedure follows the procedure in “AdaBoostM2” on page 18-53 for reweighting and constructing the ensemble.

When you construct a `RUSBoost` ensemble, there is an optional name-value pair called `RatioToSmallest`. Give a vector of  $K$  values, each value representing the multiple of  $N$  to sample for the associated class. For example, if the smallest class has  $N = 100$  members, then `RatioToSmallest = [2, 3, 4]` means each weak learner has 200 members in class 1, 300 in class 2, and 400 in class 3. If `RatioToSmallest` leads to a value that is larger than the number of members in a particular class, then `RUSBoost` samples the members with replacement. Otherwise, `RUSBoost` samples the members without replacement.

For examples using `RUSBoost`, see “Classification with Imbalanced Data” on page 18-101.

For references related to `RUSBoost`, see Seiffert et al. [43].

## Subspace

Use random subspace ensembles (`Subspace`) to improve the accuracy of discriminant analysis (`ClassificationDiscriminant`) or  $k$ -nearest neighbor (`ClassificationKNN`) classifiers. `Subspace` ensembles also have the advantage of using less memory than ensembles with all predictors, and can handle missing values (`NaNs`).

The basic random subspace algorithm uses these parameters.

- $m$  is the number of dimensions (variables) to sample in each learner. Set  $m$  using the `NPredToSample` name-value pair.
- $d$  is the number of dimensions in the data, which is the number of columns (predictors) in the data matrix  $X$ .
- $n$  is the number of learners in the ensemble. Set  $n$  using the `NLearn` input.

The basic random subspace algorithm performs the following steps:

- 1 Choose without replacement a random set of  $m$  predictors from the  $d$  possible values.
- 2 Train a weak learner using just the  $m$  chosen predictors.
- 3 Repeat steps 1 and 2 until there are  $n$  weak learners.
- 4 Predict by taking an average of the `score` prediction of the weak learners, and classify the category with the highest average `score`.

You can choose to create a weak learner for every possible set of  $m$  predictors from the  $d$  dimensions. To do so, set  $n$ , the number of learners, to `'AllPredictorCombinations'`. In this case, there are `nchoosek(size(X,2), NPredToSample)` weak learners in the ensemble.

`fitcensemble` downweights predictors after choosing them for a learner, so subsequent learners have a lower chance of using a predictor that was previously used. This weighting tends to make predictors more evenly distributed among learners than in uniform weighting.

For examples using `Subspace`, see “Random Subspace Classification” on page 18-135.

For references related to random subspace ensembles, see Ho [26].

## TotalBoost

`TotalBoost`, like linear programming boost (`LPBoost`), performs multiclass classification by attempting to maximize the minimal margin in the training set. This attempt uses optimization algorithms, namely quadratic programming for `TotalBoost`. So you need an Optimization Toolbox license to use `LPBoost` or `TotalBoost`.

The margin of a classification is the difference between the predicted soft classification score for the true class, and the largest score for the false classes. For trees, the score of a classification of a leaf node is the posterior probability of the classification at that node. The posterior probability of the classification at a node is the number of training sequences that lead to that node with the classification, divided by the number of training sequences that lead to that node. For more information, see “Definitions” on page 32-3937 in `margin`.

Why maximize the minimal margin? For one thing, the generalization error (the error on new data) is the probability of obtaining a negative margin. Schapire and Singer [41] establish this inequality on the probability of obtaining a negative margin:



$$P_{\text{test}}(m \leq 0) \leq P_{\text{train}}(m \leq \theta) + O\left(\frac{1}{\sqrt{N}} \sqrt{\frac{V \log^2(N/V)}{\theta^2} + \log(1/\delta)}\right).$$

Here  $m$  is the margin,  $\theta$  is any positive number,  $V$  is the Vapnik-Chervonenkis dimension of the classifier space,  $N$  is the size of the training set, and  $\delta$  is a small positive number. The inequality holds with probability  $1-\delta$  over many i.i.d. training and test sets. This inequality says: To obtain a low generalization error, minimize the number of observations below margin  $\theta$  in the training set.

`TotalBoost` minimizes a proxy of the Kullback-Leibler divergence between the current weight distribution and the initial weight distribution, subject to the constraint that the edge (the weighted margin) is below a certain value. The proxy is a quadratic expansion of the divergence:

$$D(W, W_0) = \sum_{n=1}^N \log \frac{W(n)}{W_0(n)} \approx \sum_{n=1}^N \left(1 + \frac{W(n)}{W_0(n)}\right) \Delta + \frac{1}{2W(n)} \Delta^2,$$

where  $\Delta$  is the difference between  $W(n)$ , the weights at the current and next iteration, and  $W_0$ , the initial weight distribution, which is uniform. This optimization formulation keeps weights from becoming zero. At each iteration, there are more constraints in the problem. So, for large problems, the optimization problem becomes increasingly constrained, and slow to solve.

`TotalBoost` typically creates ensembles with many learners having weights that are orders of magnitude smaller than those of other learners. Therefore, to better enable you to remove the unimportant ensemble members, the `compact` method reorders the members of a `TotalBoost` ensemble from largest weight to smallest. Therefore you can easily remove the least important members of the ensemble using the `removeLearners` method.

For examples using `TotalBoost`, see “LPBoost and TotalBoost for Small Ensembles” on page 18-124.

For references related to `TotalBoost`, see Warmuth, Liao, and Ratsch [44].

## See Also

`ClassificationBaggedEnsemble` | `ClassificationDiscriminant` |  
`ClassificationEnsemble` | `ClassificationKNN` |

ClassificationPartitionedEnsemble | CompactClassificationEnsemble |  
CompactRegressionEnsemble | RegressionBaggedEnsemble |  
RegressionEnsemble | RegressionPartitionedEnsemble | TreeBagger |  
fitcensemble | fitrensemble

## Related Examples

- “Framework for Ensemble Learning” on page 18-41
- “Tune RobustBoost” on page 18-131
- “Surrogate Splits” on page 18-121
- “Train Ensemble With Unequal Classification Costs” on page 18-108
- “LPBoost and TotalBoost for Small Ensembles” on page 18-124
- “Random Subspace Classification” on page 18-135

## Train Classification Ensemble

This example shows how to create a classification tree ensemble for the `ionosphere` data set, and use it to predict the classification of a radar return with average measurements.

Load the `ionosphere` data set.

```
load ionosphere
```

Train a classification ensemble. For binary classification problems, `fitcensemble` aggregates 100 classification trees using LogitBoost.

```
Mdl = fitcensemble(X,Y)
```

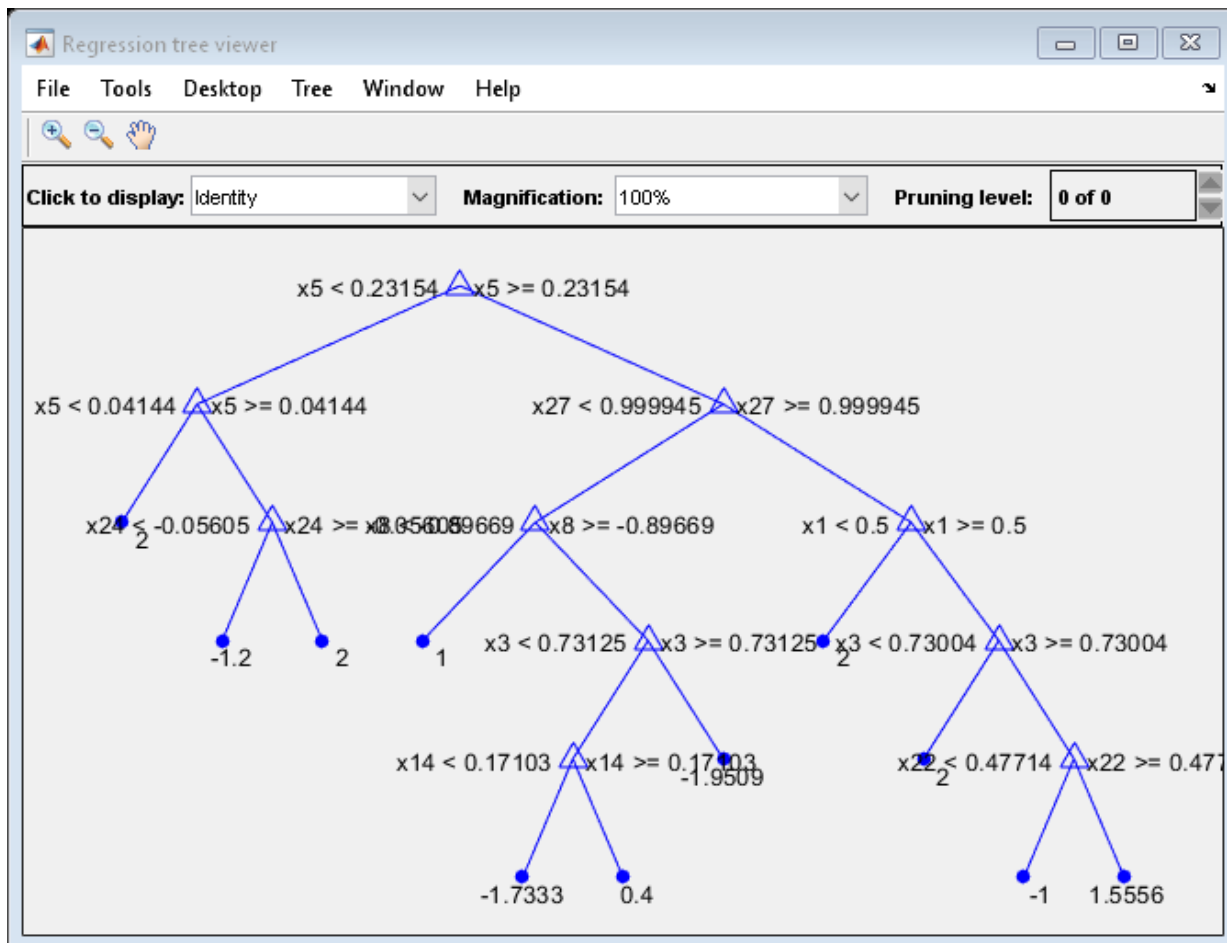
```
Mdl =
```

```
classreg.learning.classif.ClassificationEnsemble
    ResponseName: 'Y'
    CategoricalPredictors: []
    ClassNames: {'b' 'g'}
    ScoreTransform: 'none'
    NumObservations: 351
    NumTrained: 100
    Method: 'LogitBoost'
    LearnerNames: {'Tree'}
    ReasonForTermination: 'Terminated normally after completing the requested number of
    FitInfo: [100x1 double]
    FitInfoDescription: {2x1 cell}
```

`Mdl` is a `ClassificationEnsemble` model.

Plot a graph of the first trained classification tree in the ensemble.

```
view(Mdl.Trained{1}.CompactRegressionLearner, 'Mode', 'graph');
```



By default, `fitcensemble` grows shallow trees for boosting algorithms. You can alter the tree depth by passing a tree template object to `fitcensemble`. For more details, see `templateTree`.

Predict the quality of a radar return with average predictor measurements.

```
label = predict(Mdl,mean(X))
```

```
label =
```

```
1x1 cell array
```

```
{'g'}
```

## See Also

`fitcensemble` | `predict`

## Related Examples

- “Train Regression Ensemble” on page 18-70
- “Select Predictors for Random Forests” on page 18-73
- “Decision Trees” on page 19-2
- “Ensemble Algorithms” on page 18-51
- “Framework for Ensemble Learning” on page 18-41

## Train Regression Ensemble

This example shows how to create a regression ensemble to predict mileage of cars based on their horsepower and weight, trained on the `carsmall` data.

Load the `carsmall` data set.

```
load carsmall
```

Prepare the predictor data.

```
X = [Horsepower Weight];
```

The response data is `MPG`. The only available boosted regression ensemble type is `LSBoost`. For this example, arbitrarily choose an ensemble of 100 trees, and use the default tree options.

Train an ensemble of regression trees.

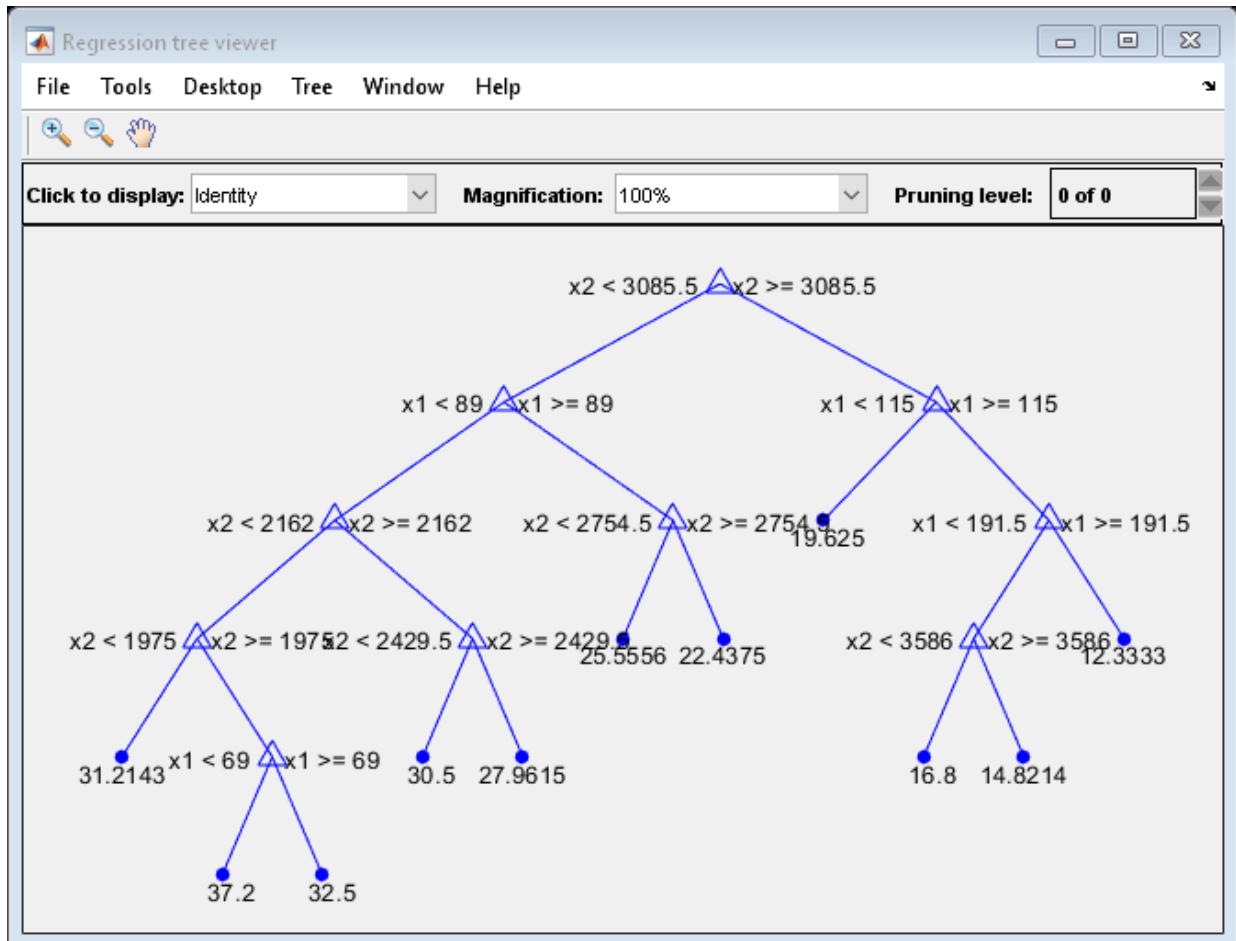
```
Mdl = fitensemble(X,MPG,'Method','LSBoost','NumLearningCycles',100)
```

```
Mdl =
```

```
classreg.learning.regr.RegistrationEnsemble
    ResponseName: 'Y'
    CategoricalPredictors: []
    ResponseTransform: 'none'
    NumObservations: 94
    NumTrained: 100
    Method: 'LSBoost'
    LearnerNames: {'Tree'}
    ReasonForTermination: 'Terminated normally after completing the requested number of
    FitInfo: [100x1 double]
    FitInfoDescription: {2x1 cell}
    Regularization: []
```

Plot a graph of the first trained regression tree in the ensemble.

```
view(Mdl.Trained{1},'Mode','graph');
```



By default, `fitrensemble` grows shallow trees for LSBoost.

Predict the mileage of a car with 150 horsepower weighing 2750 lbs.

```
mileage = predict(Mdl, [150 2750])
```

```
mileage =
```

23.6713

## See Also

`fitrensemble` | `predict`

## Related Examples

- “Train Classification Ensemble” on page 18-67
- “Select Predictors for Random Forests” on page 18-73
- “Decision Trees” on page 19-2
- “Ensemble Algorithms” on page 18-51
- “Framework for Ensemble Learning” on page 18-41



## Select Predictors for Random Forests

This example shows how to choose the appropriate split predictor selection technique for your data set when growing a random forest of regression trees. This example also shows how to decide which predictors are most important to include in the training data.

### Load and Preprocess Data

Load the `carbig` data set. Consider a model that predicts the fuel economy of a car given its number of cylinders, engine displacement, horsepower, weight, acceleration, model year, and country of origin. Consider `Cylinders`, `Model_Year`, and `Origin` as categorical variables.

```
load carbig
Cylinders = categorical(Cylinders);
Model_Year = categorical(Model_Year);
Origin = categorical(cellstr(Origin));
X = table(Cylinders, Displacement, Horsepower, Weight, Acceleration, Model_Year, ...
          Origin, MPG);
```

### Determine Levels in Predictors

The standard CART algorithm tends to split predictors with many unique values (levels), e.g., continuous variables, over those with fewer levels, e.g., categorical variables. If your data is heterogeneous, or your predictor variables vary greatly in their number of levels, then consider using the curvature or interaction tests for split-predictor selection instead of standard CART.

For each predictor, determine the number of levels in the data. One way to do this is define an anonymous function that:

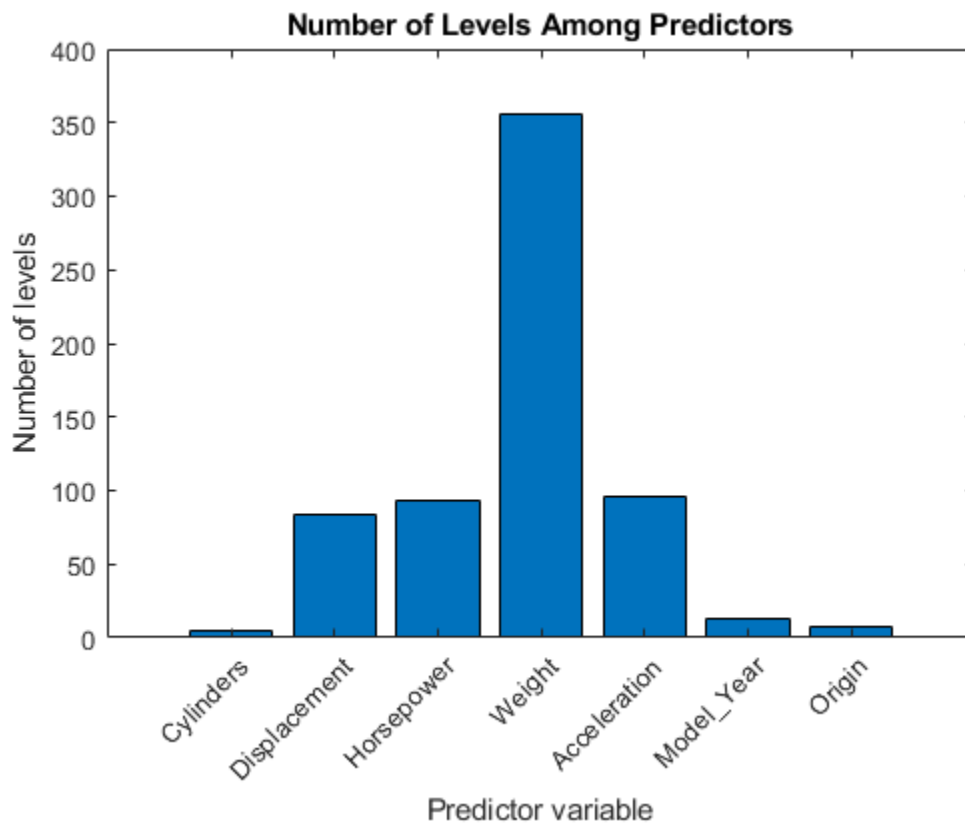
- 1 Converts all variables to the categorical data type using `categorical`
- 2 Determines all unique categories while ignoring missing values using `categories`
- 3 Counts the categories using `numel`

Then, apply the function to each variable using `varfun`.

```
countLevels = @(x) numel(categories(categorical(x)));
numLevels = varfun(countLevels, X(:, 1:end-1), 'OutputFormat', 'uniform');
```

Compare the number of levels among the predictor variables.

```
figure;  
bar(numLevels);  
title('Number of Levels Among Predictors');  
xlabel('Predictor variable');  
ylabel('Number of levels');  
h = gca;  
h.XTickLabel = X.Properties.VariableNames(1:end-1);  
h.XTickLabelRotation = 45;  
h.TickLabelInterpreter = 'none';
```



The continuous variables have many more levels than the categorical variables. Because the number of levels among the predictors vary so much, using standard CART to select split predictors at each node of the trees in a random forest can yield inaccurate predictor importance estimates.

## Grow Robust Random Forest

Grow a random forest of 200 regression trees. Specify sampling all variables at each node. Specify usage of the interaction test to select split predictors. Because there are missing values in the data, specify usage of surrogate splits to increase accuracy.

```
t = templateTree('NumPredictorsToSample','all',...
    'PredictorSelection','interaction-curve','Surrogate','on');
rng(1); % For reproducibility
Mdl = fitrensemble(X, 'MPG', 'Method', 'bag', 'NumLearningCycles', 200, ...
    'Learners', t);
```

Mdl is a RegressionBaggedEnsemble model.

Estimate the model  $R^2$  using out-of-bag predictions.

```
yHat = oobPredict(Mdl);
R2 = corr(Mdl.Y, yHat)^2
```

```
R2 =
```

```
    0.8739
```

Mdl explains 87.39% of the variability around the mean.

## Predictor Importance Estimation

Estimate predictor importance values by permuting out-of-bag observations among the trees.

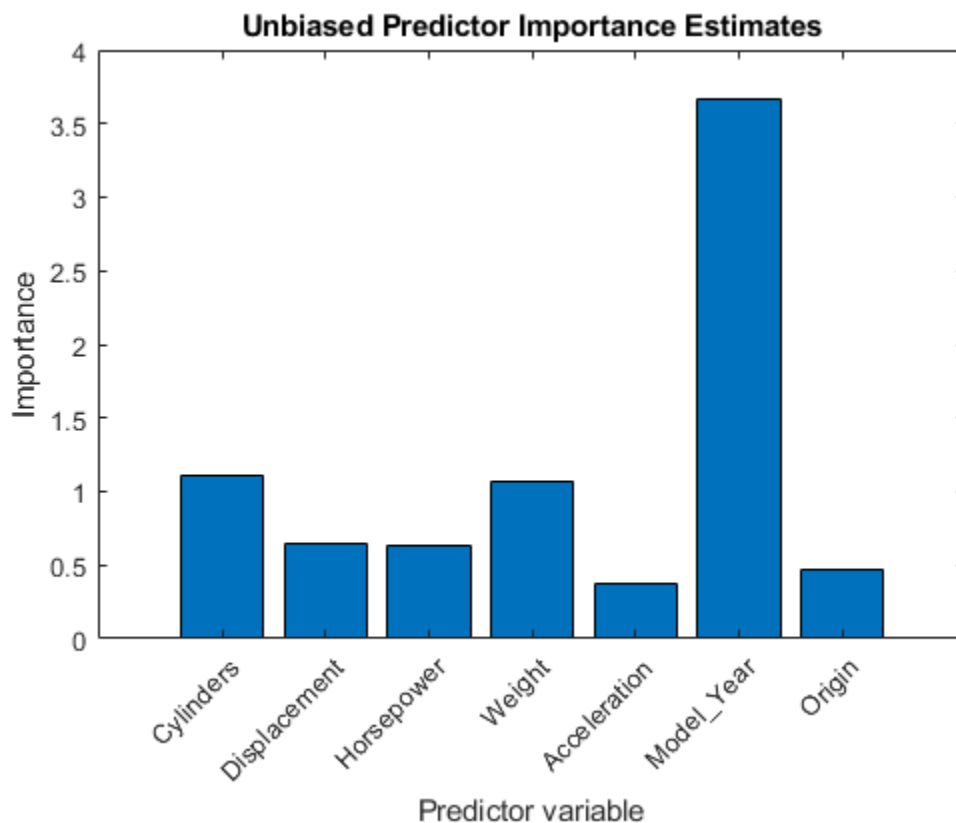
```
impOOB = oobPermutedPredictorImportance(Mdl);
```

impOOB is a 1-by-7 vector of predictor importance estimates corresponding to the predictors in Mdl.PredictorNames. The estimates are not biased toward predictors containing many levels.

Compare the predictor importance estimates.

```
figure;
bar(impOOB);
title('Unbiased Predictor Importance Estimates');
xlabel('Predictor variable');
```

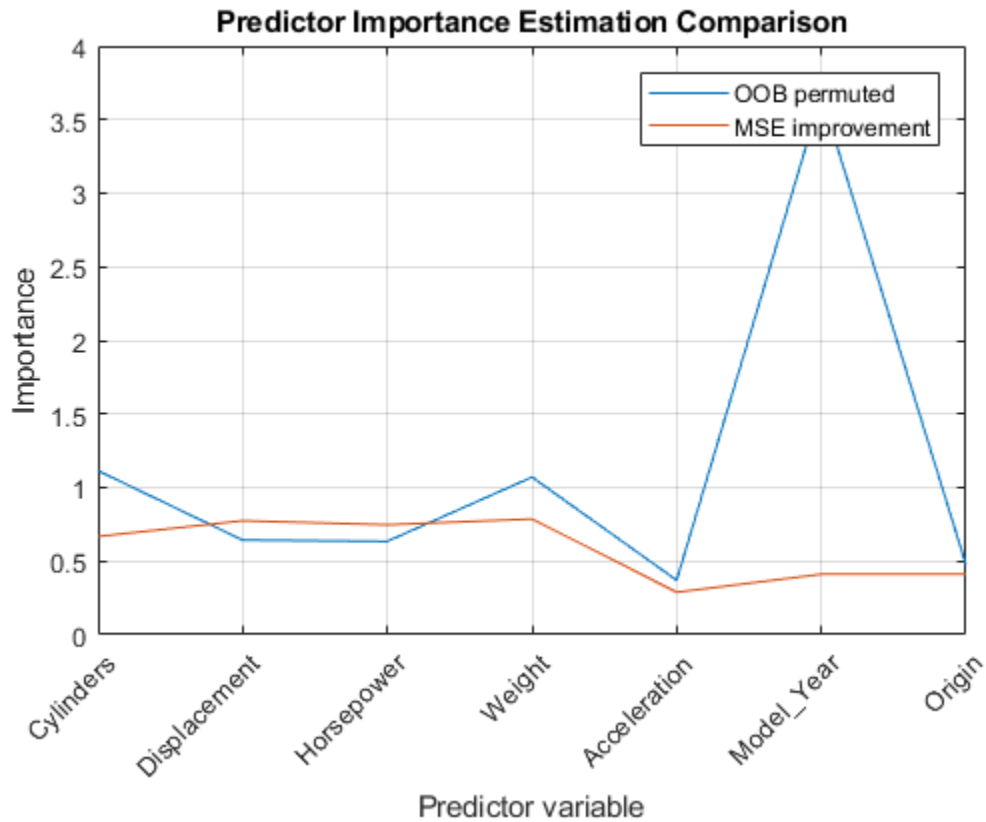
```
ylabel('Importance');  
h = gca;  
h.XTickLabel = Mdl.PredictorNames;  
h.XTickLabelRotation = 45;  
h.TickLabelInterpreter = 'none';
```



Greater importance estimates indicate more important predictors. The bar graph suggests that `Model_Year` is the most important predictor, followed by `Weight`. `Model_Year` has 13 distinct levels only, whereas `Weight` has over 300.

Compare predictor importance estimates by permuting out-of-bag observations and those estimates obtained by summing gains in the mean squared error due to splits on each predictor. Also, obtain predictor association measures estimated by surrogate splits.

```
[impGain,predAssociation] = predictorImportance(Mdl);  
  
figure;  
plot(1:numel(Mdl.PredictorNames),[impOOB' impGain]);  
title('Predictor Importance Estimation Comparison')  
xlabel('Predictor variable');  
ylabel('Importance');  
h = gca;  
h.XTickLabel = Mdl.PredictorNames;  
h.XTickLabelRotation = 45;  
h.TickLabelInterpreter = 'none';  
legend('OOB permuted','MSE improvement')  
grid on
```



`impGain` is commensurate with `impOOB`. According to the values of `impGain`, `Model_Year` and `Weight` do not appear to be the most important predictors.

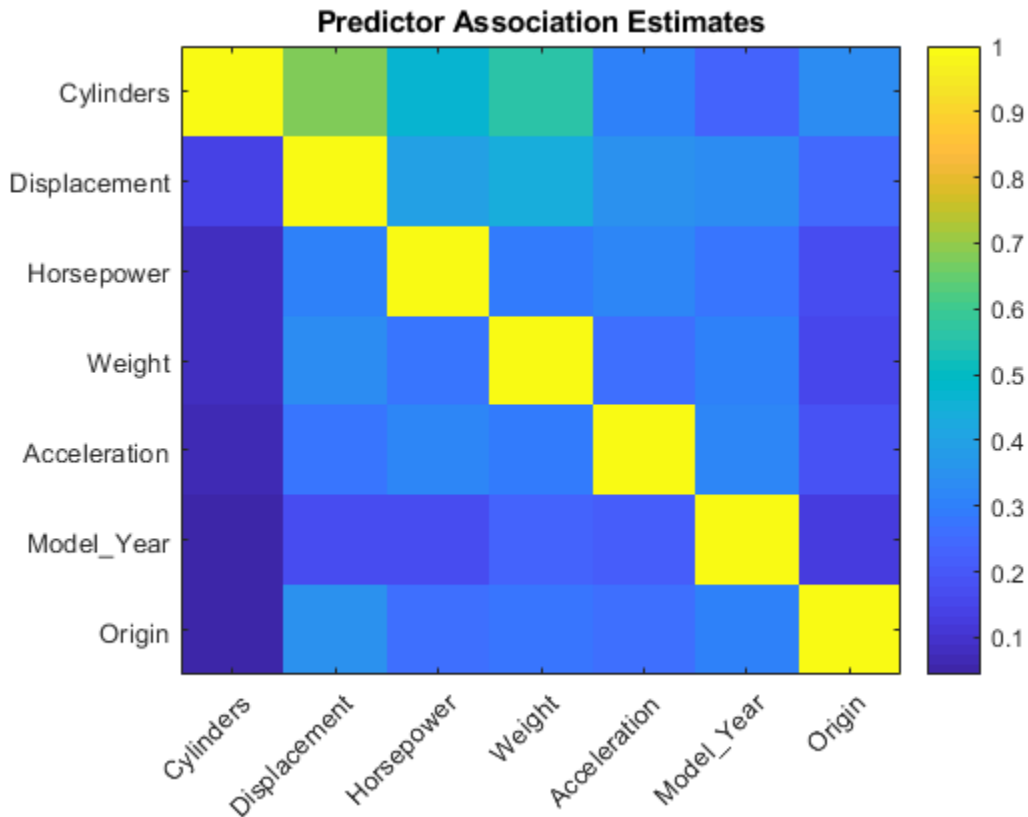
`predAssociation` is a 7-by-7 matrix of predictor association measures. Rows and columns correspond to the predictors in `Mdl.PredictorNames`. You can infer the strength of the relationship between pairs of predictors using the elements of `predAssociation`. Larger values indicate more highly correlated pairs of predictors.

```
figure;  
imagesc(predAssociation);  
title('Predictor Association Estimates');  
colorbar;  
h = gca;  
h.XTickLabel = Mdl.PredictorNames;  
h.XTickLabelRotation = 45;  
h.TickLabelInterpreter = 'none';  
h.YTickLabel = Mdl.PredictorNames;
```

```
predAssociation(1,2)
```

```
ans =
```

```
0.6830
```



The largest association is between `Cylinders` and `Displacement`, but the value is not high enough to indicate a strong relationship between the two predictors.

### Grow Random Forest Using Reduced Predictor Set

Because prediction time increases with the number of predictors in random forests, it is good practice to create a model using as few predictors as possible.

Grow a random forest of 200 regression trees using the best two predictors only.

```
MdlReduced = fitrensemble(X(:, {'Model_Year' 'Weight' 'MPG'}), 'MPG', 'Method', 'bag', ...
    'NumLearningCycles', 200, 'Learners', t);
```

Compute the  $R^2$  of the reduced model.

```
yHatReduced = oobPredict (MdlReduced) ;  
r2Reduced = corr (Mdl.Y, yHatReduced) ^2
```

```
r2Reduced =  
  
    0.8524
```

The  $R^2$  for the reduced model is close to the  $R^2$  of the full model. This result suggests that the reduced model is sufficient for prediction.

## See Also

```
corr | fitensemble | oobPermutedPredictorImportance | oobPredict |  
predictorImportance | templateTree
```

## Related Examples

- “Improving Classification Trees and Regression Trees” on page 19-17
- “Bootstrap Aggregation (Bagging) of Regression Trees” on page 18-141
- “Surrogate Splits” on page 18-121



## Test Ensemble Quality

You cannot evaluate the predictive quality of an ensemble based on its performance on training data. Ensembles tend to “overtrain,” meaning they produce overly optimistic estimates of their predictive power. This means the result of `resubLoss` for classification (`resubLoss` for regression) usually indicates lower error than you get on new data.

To obtain a better idea of the quality of an ensemble, use one of these methods:

- Evaluate the ensemble on an independent test set (useful when you have a lot of training data).
- Evaluate the ensemble by cross validation (useful when you don't have a lot of training data).
- Evaluate the ensemble on out-of-bag data (useful when you create a bagged ensemble with `fitcensemble` or `fitrensemble`).

### Test Ensemble Quality

This example uses a bagged ensemble so it can use all three methods of evaluating ensemble quality.

Generate an artificial dataset with 20 predictors. Each entry is a random number from 0 to 1. The initial classification is  $Y = 1$  if  $X_1 + X_2 + X_3 + X_4 + X_5 > 2.5$  and  $Y = 0$  otherwise.

```
rng(1, 'twister') % for reproducibility
X = rand(2000,20);
Y = sum(X(:,1:5),2) > 2.5;
```

In addition, to add noise to the results, randomly switch 10% of the classifications:

```
idx = randsample(2000,200);
Y(idx) = ~Y(idx);
```

### Independent Test Set

Create independent training and test sets of data. Use 70% of the data for a training set by calling `cvpartition` using the `holdout` option:

```
cvpart = cvpartition(Y, 'holdout', 0.3);
Xtrain = X(training(cvpart), :);
```

```
Ytrain = Y(training(cvpart), :);
Xtest = X(test(cvpart), :);
Ytest = Y(test(cvpart), :);
```

Create a bagged classification ensemble of 200 trees from the training data:

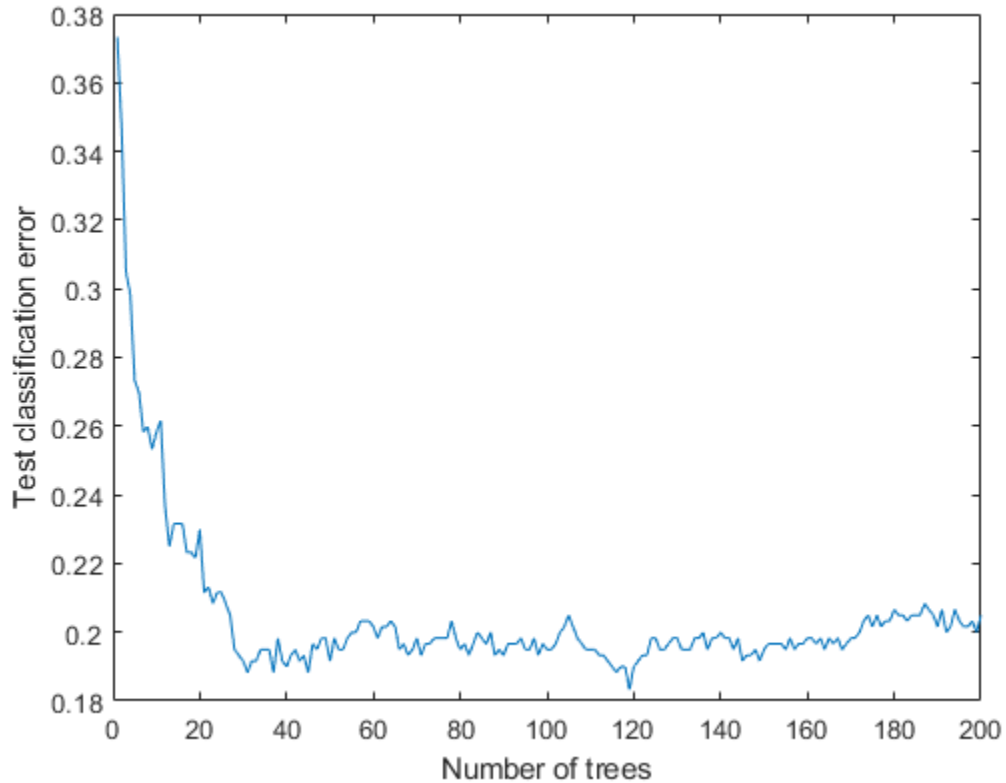
```
bag = fitcensemble(Xtrain, Ytrain, 'Method', 'Bag', 'NumLearningCycles', 200)
```

```
bag =
```

```
classreg.learning.classif.ClassificationBaggedEnsemble
    ResponseName: 'Y'
    CategoricalPredictors: []
        ClassNames: [0 1]
        ScoreTransform: 'none'
    NumObservations: 1400
    NumTrained: 200
        Method: 'Bag'
        LearnerNames: {'Tree'}
    ReasonForTermination: 'Terminated normally after completing the requested number o
        FitInfo: []
    FitInfoDescription: 'None'
        FResample: 1
        Replace: 1
    UseObsForLearner: [1400x200 logical]
```

Plot the loss (misclassification) of the test data as a function of the number of trained trees in the ensemble:

```
figure;
plot(loss(bag, Xtest, Ytest, 'mode', 'cumulative'));
xlabel('Number of trees');
ylabel('Test classification error');
```



### Cross Validation

Generate a five-fold cross-validated bagged ensemble:

```
cv = fitcensemble(X,Y,'Method','Bag','NumLearningCycles',200,'Kfold',5)
```

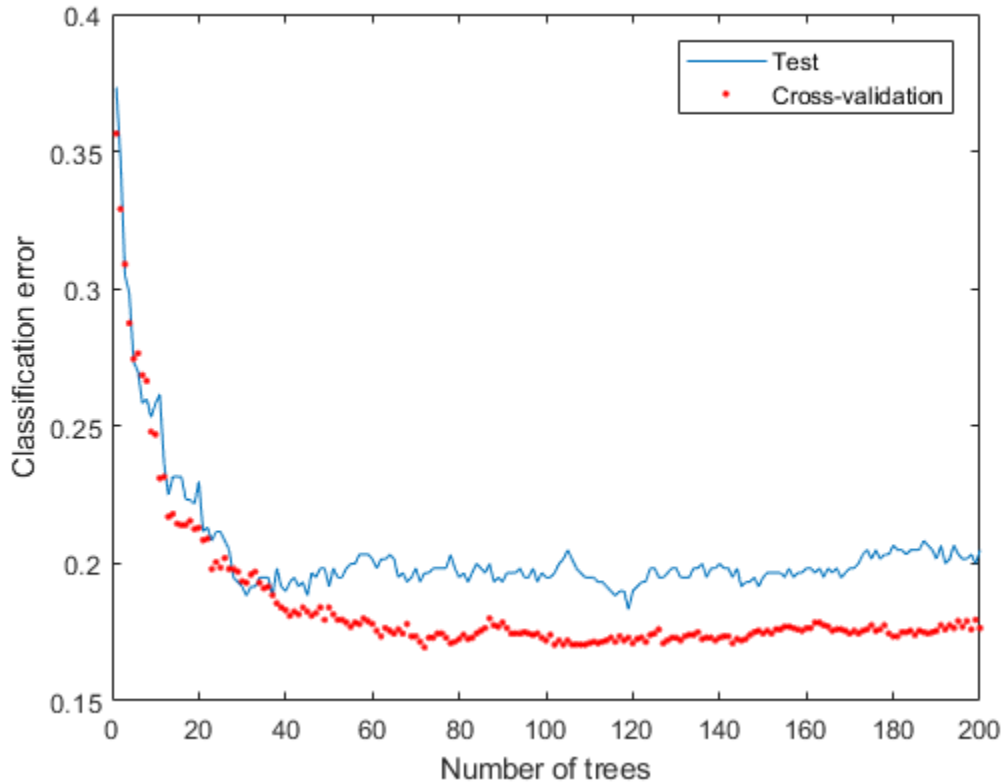
```
cv =
```

```
classreg.learning.partition.ClassificationPartitionedEnsemble  
  CrossValidatedModel: 'Bag'  
    PredictorNames: {1x20 cell}  
      ResponseName: 'Y'  
    NumObservations: 2000
```

```
      KFold: 5
      Partition: [1x1 cvpartition]
      NumTrainedPerFold: [200 200 200 200 200]
      ClassNames: [0 1]
      ScoreTransform: 'none'
```

Examine the cross-validation loss as a function of the number of trees in the ensemble:

```
figure;
plot(loss(bag,Xtest,Ytest,'mode','cumulative'));
hold on;
plot(kfoldLoss(cv,'mode','cumulative'),'r. ');
hold off;
xlabel('Number of trees');
ylabel('Classification error');
legend('Test','Cross-validation','Location','NE');
```



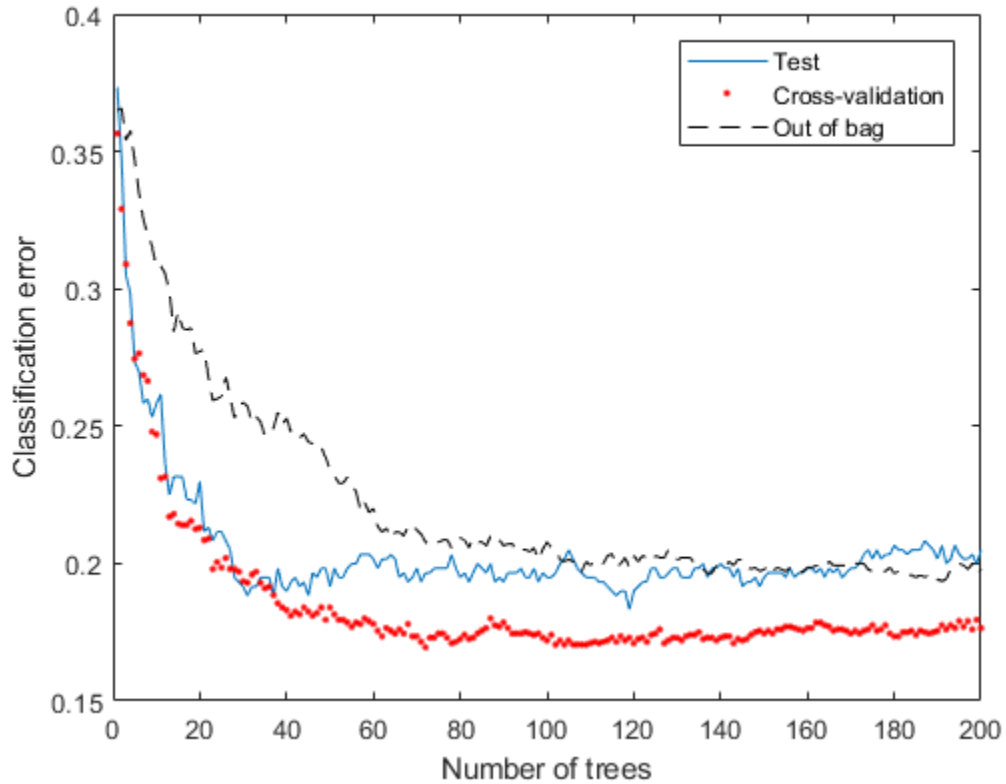
Cross validating gives comparable estimates to those of the independent set.

### Out-of-Bag Estimates

Generate the loss curve for out-of-bag estimates, and plot it along with the other curves:

```
figure;
plot(loss(bag,Xtest,Ytest,'mode','cumulative'));
hold on;
plot(kfoldLoss(cv,'mode','cumulative'),'r.');
plot(oobLoss(bag,'mode','cumulative'),'k--');
hold off;
xlabel('Number of trees');
```

```
ylabel('Classification error');  
legend('Test', 'Cross-validation', 'Out of bag', 'Location', 'NE');
```



The out-of-bag estimates are again comparable to those of the other methods.

## See Also

`cvpartition` | `cvpartition` | `fitcensemble` | `fitrensemble` | `kfoldLoss` | `loss`  
| `oobLoss` | `resubLoss` | `resubLoss`

## Related Examples

- “Framework for Ensemble Learning” on page 18-41
- “Ensemble Algorithms” on page 18-51
- “Classifying Radar Returns for Ionosphere Data Using TreeBagger” on page 18-154

## Ensemble Regularization

Regularization is a process of choosing fewer weak learners for an ensemble in a way that does not diminish predictive performance. Currently you can regularize regression ensembles. (You can also regularize a discriminant analysis classifier in a non-ensemble context; see “Regularize a Discriminant Analysis Classifier” on page 20-28.)

The `regularize` method finds an optimal set of learner weights  $\alpha_t$  that minimize

$$\sum_{n=1}^N w_n g \left( \left( \sum_{t=1}^T \alpha_t h_t(x_n) \right), y_n \right) + \lambda \sum_{t=1}^T |\alpha_t|.$$

Here

- $\lambda \geq 0$  is a parameter you provide, called the lasso parameter.
- $h_t$  is a weak learner in the ensemble trained on  $N$  observations with predictors  $x_n$ , responses  $y_n$ , and weights  $w_n$ .
- $g(f,y) = (f - y)^2$  is the squared error.

The ensemble is regularized on the same  $(x_n, y_n, w_n)$  data used for training, so

$$\sum_{n=1}^N w_n g \left( \left( \sum_{t=1}^T \alpha_t h_t(x_n) \right), y_n \right)$$

is the ensemble resubstitution error. The error is measured by mean squared error (MSE).

If you use  $\lambda = 0$ , `regularize` finds the weak learner weights by minimizing the resubstitution MSE. Ensembles tend to overtrain. In other words, the resubstitution error is typically smaller than the true generalization error. By making the resubstitution error even smaller, you are likely to make the ensemble accuracy worse instead of improving it. On the other hand, positive values of  $\lambda$  push the magnitude of the  $\alpha_t$  coefficients to 0. This often improves the generalization error. Of course, if you choose  $\lambda$  too large, all the optimal coefficients are 0, and the ensemble does not have any accuracy. Usually you can find an optimal range for  $\lambda$  in which the accuracy of the regularized ensemble is better or comparable to that of the full ensemble without regularization.

A nice feature of lasso regularization is its ability to drive the optimized coefficients precisely to 0. If a learner's weight  $\alpha_t$  is 0, this learner can be excluded from the



regularized ensemble. In the end, you get an ensemble with improved accuracy and fewer learners.

## Regularize a Regression Ensemble

This example uses data for predicting the insurance risk of a car based on its many attributes.

Load the `imports-85` data into the MATLAB workspace.

```
load imports-85;
```

Look at a description of the data to find the categorical variables and predictor names.

Description

```
Description =
```

```
9x79 char array
```

```
'1985 Auto Imports Database from the UCI repository'
'http://archive.ics.uci.edu/ml/machine-learning-databases/autos/imports-85.names'
'Variables have been reordered to place variables with numeric values (referred'
'to as "continuous" on the UCI site) to the left and categorical values to the'
'right. Specifically, variables 1:16 are: symboling, normalized-losses,'
'wheel-base, length, width, height, curb-weight, engine-size, bore, stroke,'
'compression-ratio, horsepower, peak-rpm, city-mpg, highway-mpg, and price.'
'Variables 17:26 are: make, fuel-type, aspiration, num-of-doors, body-style,'
'drive-wheels, engine-location, engine-type, num-of-cylinders, and fuel-system.'
```

The objective of this process is to predict the "symboling," the first variable in the data, from the other predictors. "symboling" is an integer from  $-3$  (good insurance risk) to  $3$  (poor insurance risk). You could use a classification ensemble to predict this risk instead of a regression ensemble. When you have a choice between regression and classification, you should try regression first.

Prepare the data for ensemble fitting.

```
Y = X(:,1);
X(:,1) = [];
VarNames = {'normalized-losses' 'wheel-base' 'length' 'width' 'height' ...
```

```
'curb-weight' 'engine-size' 'bore' 'stroke' 'compression-ratio' ...  
'horsepower' 'peak-rpm' 'city-mpg' 'highway-mpg' 'price' 'make' ...  
'fuel-type' 'aspiration' 'num-of-doors' 'body-style' 'drive-wheels' ...  
'engine-location' 'engine-type' 'num-of-cylinders' 'fuel-system'};  
catidx = 16:25; % indices of categorical predictors
```

Create a regression ensemble from the data using 300 trees.

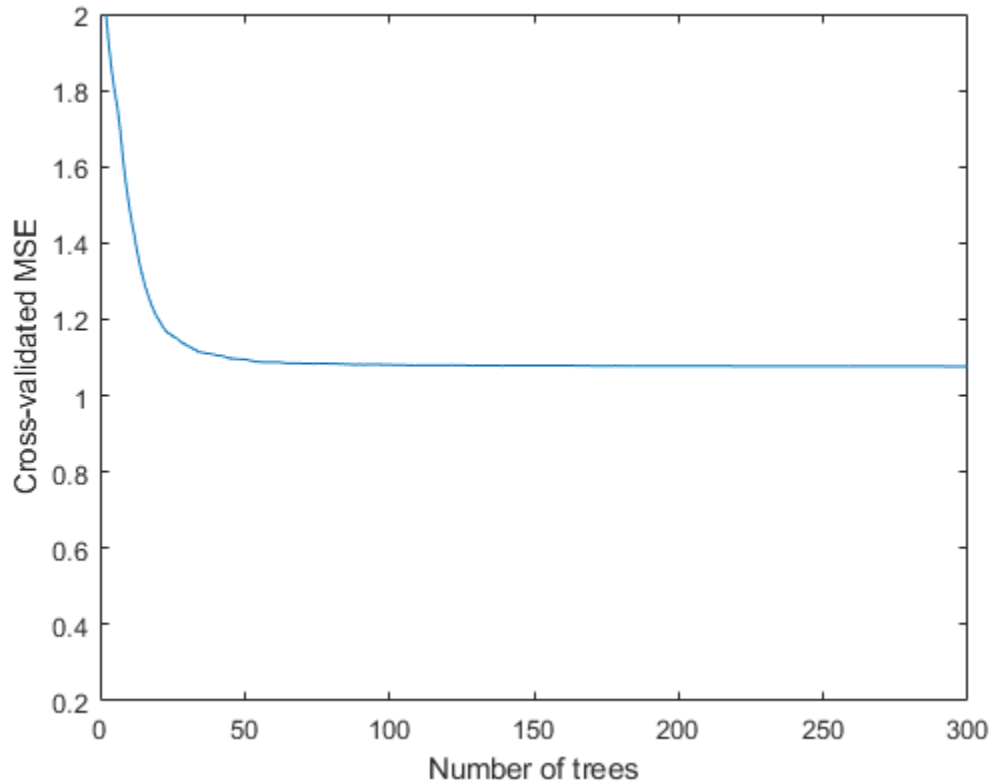
```
ls = fitensemble(X,Y,'Method','LSBoost','NumLearningCycles',300, ...  
    'LearnRate',0.1,'PredictorNames',VarNames, ...  
    'ResponseName','Symboling','CategoricalPredictors',catidx)
```

```
ls =
```

```
classreg.learning.regr.RegressionEnsemble  
    PredictorNames: {1x25 cell}  
    ResponseName: 'Symboling'  
    CategoricalPredictors: [16 17 18 19 20 21 22 23 24 25]  
    ResponseTransform: 'none'  
    NumObservations: 205  
    NumTrained: 300  
    Method: 'LSBoost'  
    LearnerNames: {'Tree'}  
    ReasonForTermination: 'Terminated normally after completing the requested number o  
    FitInfo: [300x1 double]  
    FitInfoDescription: {2x1 cell}  
    Regularization: []
```

The final line, Regularization, is empty ([]). To regularize the ensemble, you have to use the `regularize` method.

```
cv = crossval(ls,'KFold',5);  
figure;  
plot(kfoldLoss(cv,'Mode','Cumulative'));  
xlabel('Number of trees');  
ylabel('Cross-validated MSE');  
ylim([0.2,2])
```



It appears you might obtain satisfactory performance from a smaller ensemble, perhaps one containing from 50 to 100 trees.

Call the `regularize` method to try to find trees that you can remove from the ensemble. By default, `regularize` examines 10 values of the lasso (`Lambda`) parameter spaced exponentially.

```
ls = regularize(ls)
```

```
ls =
```

```
classreg.learning.regr.RegressionEnsemble  
    PredictorNames: {1x25 cell}
```

```

        ResponseName: 'Symboling'
CategoricalPredictors: [16 17 18 19 20 21 22 23 24 25]
        ResponseTransform: 'none'
        NumObservations: 205
            NumTrained: 300
                Method: 'LSBoost'
            LearnerNames: {'Tree'}
ReasonForTermination: 'Terminated normally after completing the requested number of
        FitInfo: [300x1 double]
FitInfoDescription: {2x1 cell}
        Regularization: [1x1 struct]

```

The Regularization property is no longer empty.

Plot the resubstitution mean-squared error (MSE) and number of learners with nonzero weights against the lasso parameter. Separately plot the value at  $\text{Lambda} = 0$ . Use a logarithmic scale because the values of  $\text{Lambda}$  are exponentially spaced.

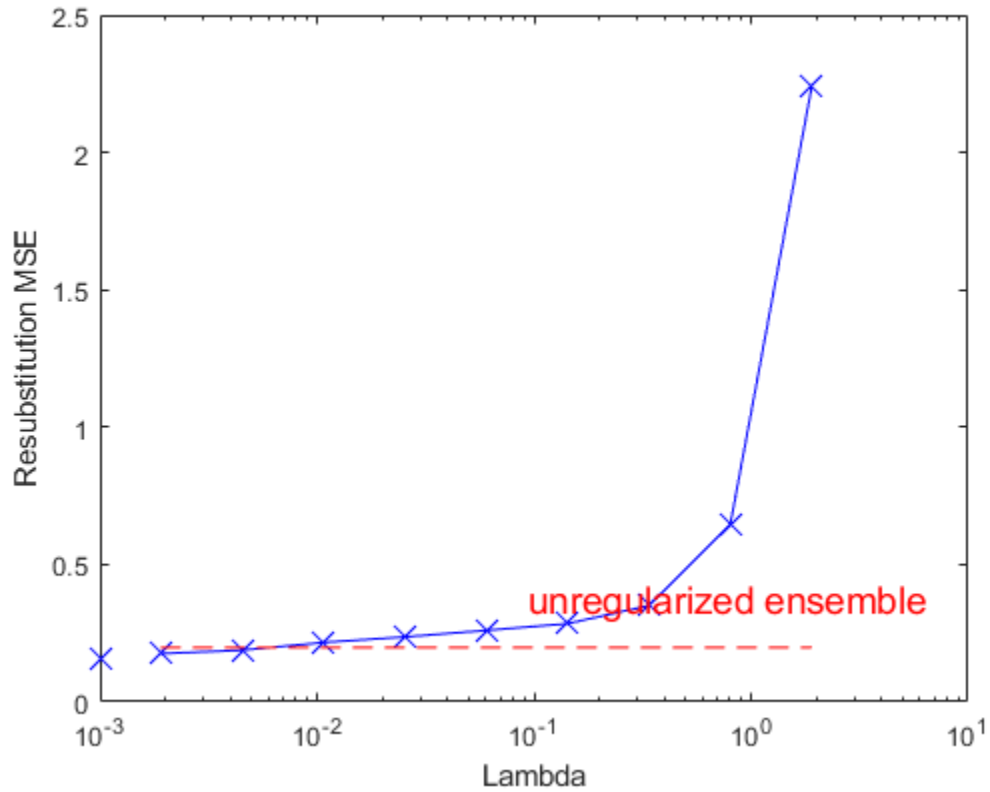
```

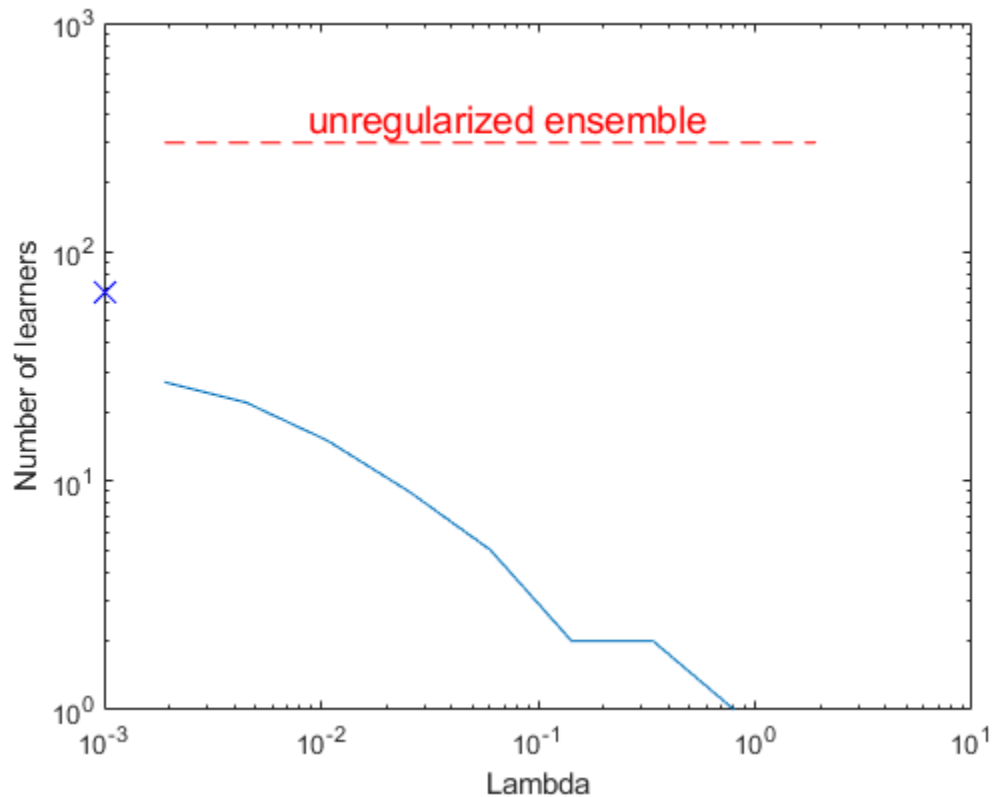
figure;
semilogx(ls.Regularization.Lambda,ls.Regularization.ResubstitutionMSE, ...
    'bx-', 'Markersize',10);
line([1e-3 1e-3],[ls.Regularization.ResubstitutionMSE(1) ...
    ls.Regularization.ResubstitutionMSE(1)],...
    'Marker','x','Markersize',10,'Color','b');
r0 = resubLoss(ls);
line([ls.Regularization.Lambda(2) ls.Regularization.Lambda(end)],...
    [r0 r0],'Color','r','LineStyle','--');
xlabel('Lambda');
ylabel('Resubstitution MSE');
annotation('textbox',[0.5 0.22 0.5 0.05],'String','unregularized ensemble', ...
    'Color','r','FontSize',14,'LineStyle','none');

figure;
loglog(ls.Regularization.Lambda,sum(ls.Regularization.TrainedWeights>0,1));
line([1e-3 1e-3],...
    [sum(ls.Regularization.TrainedWeights(:,1)>0) ...
    sum(ls.Regularization.TrainedWeights(:,1)>0)],...
    'marker','x','markersize',10,'color','b');
line([ls.Regularization.Lambda(2) ls.Regularization.Lambda(end)],...
    [ls.NTrained ls.NTrained],...
    'color','r','LineStyle','--');
xlabel('Lambda');

```

```
ylabel('Number of learners');  
annotation('textbox',[0.3 0.8 0.5 0.05],'String','unregularized ensemble',...  
          'color','r','FontSize',14,'LineStyle','none');
```





The resubstitution MSE values are likely to be overly optimistic. To obtain more reliable estimates of the error associated with various values of  $\Lambda$ , cross validate the ensemble using `cvshrink`. Plot the resulting cross-validation loss (MSE) and number of learners against  $\Lambda$ .

```
rng(0, 'Twister') % for reproducibility
[mse, nlearn] = cvshrink(ls, 'Lambda', ls.Regularization.Lambda, 'Kfold', 5);

figure;
semilogx(ls.Regularization.Lambda, ls.Regularization.ResubstitutionMSE, ...
         'bx-', 'Markersize', 10);
hold on;
semilogx(ls.Regularization.Lambda, mse, 'ro-', 'Markersize', 10);
hold off;
```

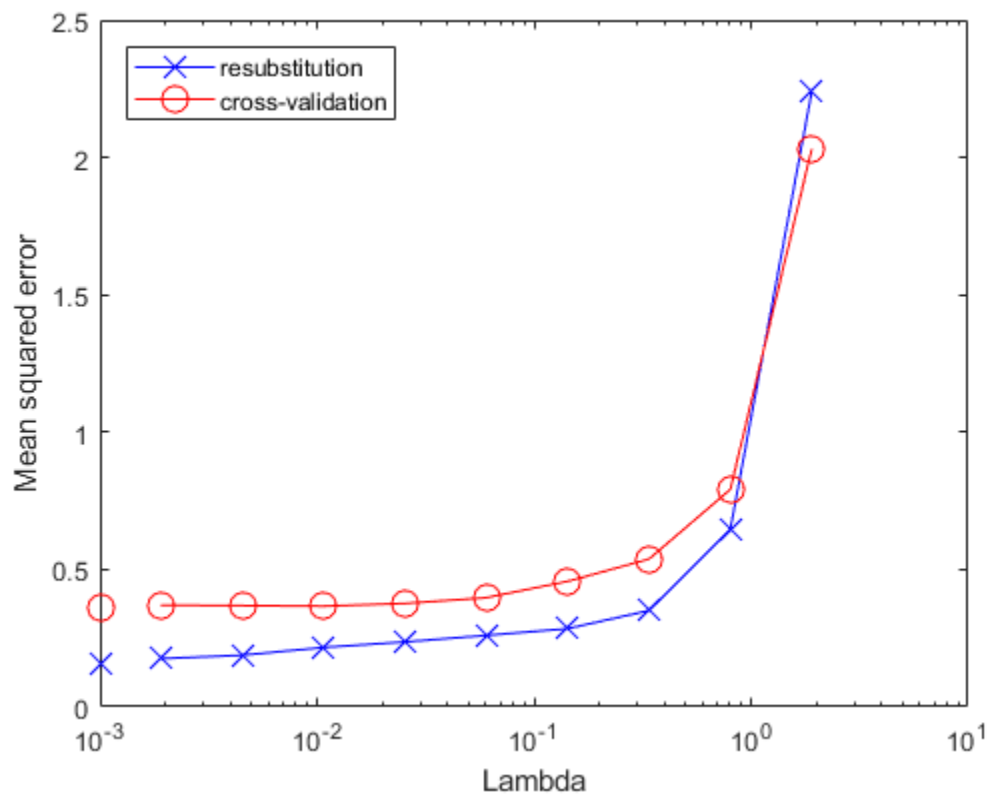
```

xlabel('Lambda');
ylabel('Mean squared error');
legend('resubstitution','cross-validation','Location','NW');
line([1e-3 1e-3],[ls.Regularization.ResubstitutionMSE(1) ...
    ls.Regularization.ResubstitutionMSE(1)],...
    'Marker','x','Markersize',10,'Color','b','HandleVisibility','off');
line([1e-3 1e-3],[mse(1) mse(1)],'Marker','o',...
    'Markersize',10,'Color','r','LineStyle','--','HandleVisibility','off');

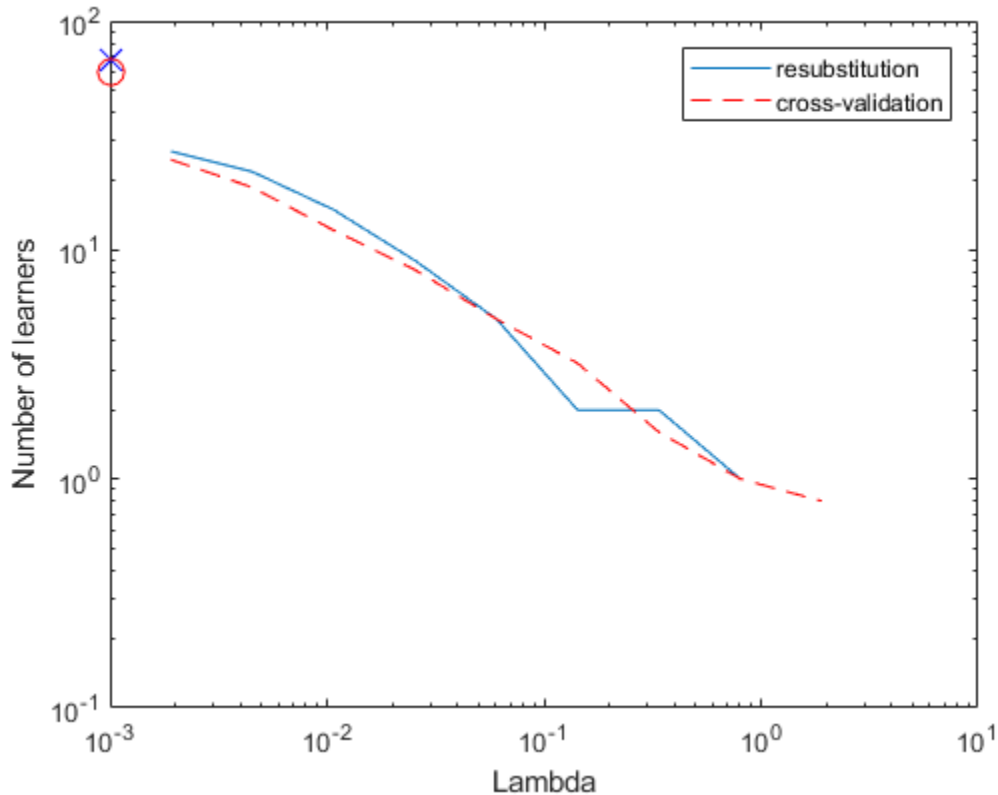
figure;
loglog(ls.Regularization.Lambda,sum(ls.Regularization.TrainedWeights>0,1));
hold;
loglog(ls.Regularization.Lambda,nlearn,'r--');
hold off;
xlabel('Lambda');
ylabel('Number of learners');
legend('resubstitution','cross-validation','Location','NE');
line([1e-3 1e-3],...
    [sum(ls.Regularization.TrainedWeights(:,1)>0) ...
    sum(ls.Regularization.TrainedWeights(:,1)>0)],...
    'Marker','x','Markersize',10,'Color','b','HandleVisibility','off');
line([1e-3 1e-3],[nlearn(1) nlearn(1)],'marker','o',...
    'Markersize',10,'Color','r','LineStyle','--','HandleVisibility','off');

Warning: Some folds do not have any trained weak learners.
Current plot held

```







Examining the cross-validated error shows that the cross-validation MSE is almost flat for Lambda up to a bit over  $1e-2$ .

Examine `ls.Regularization.Lambda` to find the highest value that gives MSE in the flat region (up to a bit over  $1e-2$ ).

```
jj = 1:length(ls.Regularization.Lambda);
[jj;ls.Regularization.Lambda]
```

```
ans =
```

```
Columns 1 through 7
```

```
1.0000    2.0000    3.0000    4.0000    5.0000    6.0000    7.0000
          0    0.0019    0.0045    0.0107    0.0254    0.0602    0.1428

Columns 8 through 10

8.0000    9.0000   10.0000
0.3387    0.8033    1.9048
```

Element 5 of `ls.Regularization.Lambda` has value 0.0254, the largest in the flat range.

Reduce the ensemble size using the `shrink` method. `shrink` returns a compact ensemble with no training data. The generalization error for the new compact ensemble was already estimated by cross validation in `mse(5)`.

```
cmp = shrink(ls, 'weightcolumn', 5)

cmp =

classreg.learning.regr.CompactRegressionEnsemble
    PredictorNames: {1x25 cell}
    ResponseName: 'Symboling'
    CategoricalPredictors: [16 17 18 19 20 21 22 23 24 25]
    ResponseTransform: 'none'
    NumTrained: 9
```

The number of trees in the new ensemble has notably reduced from the 300 in `ls`.

Compare the sizes of the ensembles.

```
sz(1) = whos('cmp'); sz(2) = whos('ls');
[sz(1).bytes sz(2).bytes]

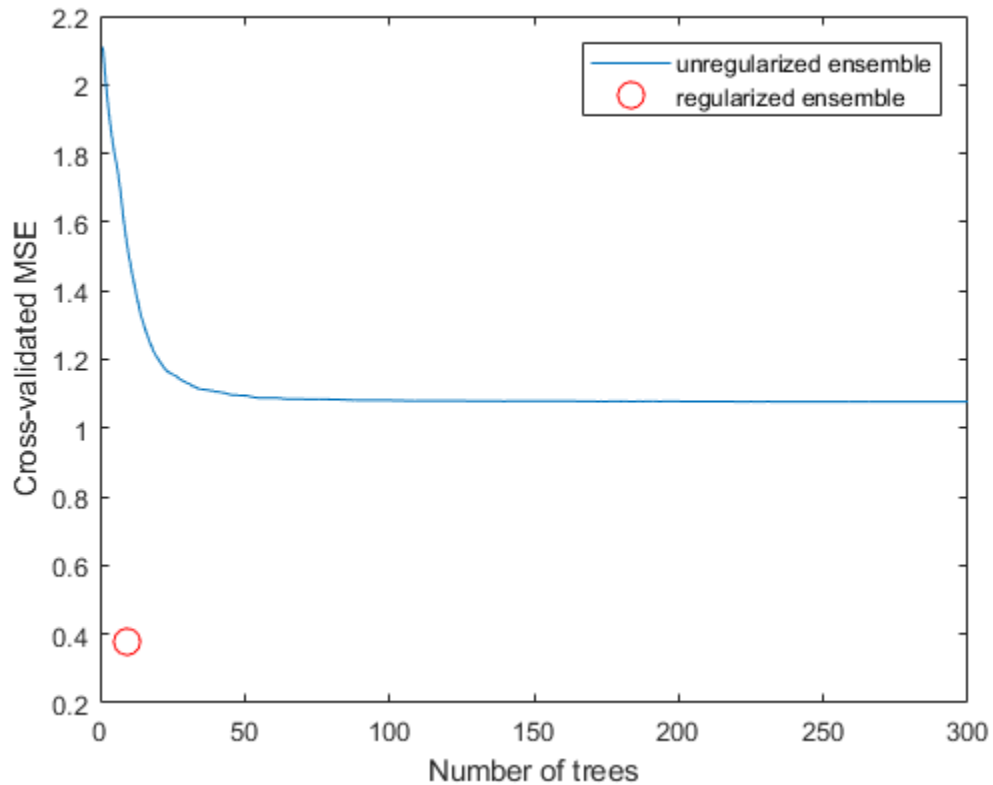
ans =

    95692    3014096
```

The size of the reduced ensemble is a fraction of the size of the original. Note that your ensemble sizes can vary depending on your operating system.

Compare the MSE of the reduced ensemble to that of the original ensemble.

```
figure;  
plot(kfoldLoss(cv, 'mode', 'cumulative'));  
hold on  
plot(cmp.NTrained, mse(5), 'ro', 'MarkerSize', 10);  
xlabel('Number of trees');  
ylabel('Cross-validated MSE');  
legend('unregularized ensemble', 'regularized ensemble', ...  
      'Location', 'NE');  
hold off
```



The reduced ensemble gives low loss while using many fewer trees.

### See Also

`crossval` | `cvshrink` | `fitrensemble` | `kfoldLoss` | `regularize` | `resubLoss` | `shrink`

### Related Examples

- “Regularize a Discriminant Analysis Classifier” on page 20-28
- “Framework for Ensemble Learning” on page 18-41
- “Ensemble Algorithms” on page 18-51

## Classification with Imbalanced Data

This example shows how to classify when one class has many more observations than another. Try the RUSBoost algorithm first, because it is designed to handle this case.

This example uses the "Cover type" data from the UCI machine learning archive, described in <http://archive.ics.uci.edu/ml/datasets/Covertype>. The data classifies types of forest (ground cover), based on predictors such as elevation, soil type, and distance to water. The data has over 500,000 observations and over 50 predictors, so training and using a classifier is time consuming.

Blackard and Dean [4] describe a neural net classification of this data. They quote a 70.6% classification accuracy. RUSBoost obtains over 81% classification accuracy.

### Obtain the data

Import the data into your workspace. Extract the last data column into a variable named `Y`.

```
gunzip('http://archive.ics.uci.edu/ml/machine-learning-databases/covtype/covtype.data.gz')
load covtype.data
Y = covtype(:,end);
covtype(:,end) = [];
```

### Examine the response data

```
tabulate(Y)
```

Value	Count	Percent
1	211840	36.46%
2	283301	48.76%
3	35754	6.15%
4	2747	0.47%
5	9493	1.63%
6	17367	2.99%
7	20510	3.53%

There are hundreds of thousands of data points. Those of class 4 are less than 0.5% of the total. This imbalance indicates that RUSBoost is an appropriate algorithm.

### Partition the data for quality assessment

Use half the data to fit a classifier, and half to examine the quality of the resulting classifier.

```
rng(10, 'twister')           % For reproducibility
part = cvpartition(Y, 'Holdout', 0.5);
istrain = training(part); % Data for fitting
istest = test(part);      % Data for quality assessment
tabulate(Y(istrain))
```

Value	Count	Percent
1	105919	36.46%
2	141651	48.76%
3	17877	6.15%
4	1374	0.47%
5	4747	1.63%
6	8684	2.99%
7	10254	3.53%

### Create the ensemble

Use deep trees for higher ensemble accuracy. To do so, set the trees to have maximal number of decision splits of  $N$ , where  $N$  is the number of observations in the training sample. Set `LearnRate` to 0.1 in order to achieve higher accuracy as well. The data is large, and, with deep trees, creating the ensemble is time consuming.

```
N = sum(istrain);           % Number of observations in the training sample
t = templateTree('MaxNumSplits', N);
tic
rusTree = fitcensemble(covtype(istrain,:), Y(istrain), 'Method', 'RUSBoost', ...
    'NumLearningCycles', 1000, 'Learners', t, 'LearnRate', 0.1, 'nprint', 100);
toc
```

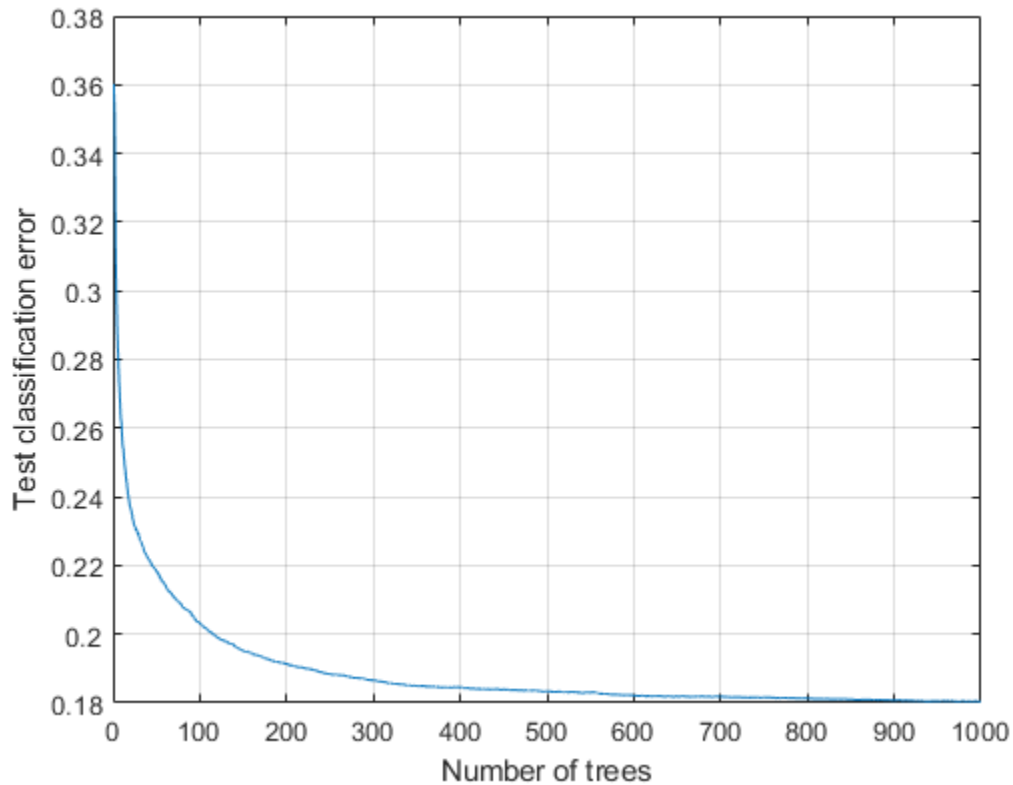
```
Training RUSBoost...
Grown weak learners: 100
Grown weak learners: 200
Grown weak learners: 300
Grown weak learners: 400
Grown weak learners: 500
Grown weak learners: 600
Grown weak learners: 700
Grown weak learners: 800
Grown weak learners: 900
Grown weak learners: 1000
Elapsed time is 426.143168 seconds.
```

### Inspect the classification error

Plot the classification error against the number of members in the ensemble.

```
figure;  
tic  
plot(loss(rusTree,covtype(istest,:),Y(istest),'mode','cumulative'));  
toc  
grid on;  
xlabel('Number of trees');  
ylabel('Test classification error');
```

Elapsed time is 267.560092 seconds.



The ensemble achieves a classification error of under 20% using 116 or more trees. For 500 or more trees, the classification error decreases at a slower rate.

Examine the confusion matrix for each class as a percentage of the true class.

```
tic
Yfit = predict(rusTree,covtype(istest,:));
toc
tab = tabulate(Y(istest));
bsxfun(@rdivide,confusionmat(Y(istest),Yfit),tab(:,2))*100

Elapsed time is 245.604008 seconds.

ans =

    90.5354    4.1040    0.0434         0    1.0480    0.1511    4.1182
   17.5171   71.2467    1.8292    0.0162    6.4335    2.2803    0.6770
         0    0.0671   93.6678    1.6558    0.5594    4.0499         0
         0         0    3.7145   94.6832         0    1.6023         0
    0.1054    0.1896    0.5057         0   98.8622    0.3371         0
         0    0.1037    2.7064    1.1056    0.3340   95.7503         0
    0.2340    0.0098         0         0    0.0098         0   99.7465
```

All classes except class 2 have over 90% classification accuracy. But class 2 makes up close to half the data, so the overall accuracy is not that high.

### Compact the ensemble

The ensemble is large. Remove the data using the `compact` method.

```
cmpctRus = compact(rusTree);

sz(1) = whos('rusTree');
sz(2) = whos('cmpctRus');
[sz(1).bytes sz(2).bytes]
```

```
ans =

    1.0e+09 *

    1.6575    0.9418
```

The compacted ensemble is about half the size of the original.

Remove half the trees from `cmpctRus`. This action is likely to have minimal effect on the predictive performance, based on the observation that 500 out of 1000 trees give nearly optimal accuracy.



```
cmpctRus = removeLearners(cmpctRus, [500:1000]);  
  
sz(3) = whos('cmpctRus');  
sz(3).bytes  
  
ans =  
  
    452637153
```

The reduced compact ensemble takes about a quarter of the memory of the full ensemble. Its overall loss rate is under 19%:

```
L = loss(cmpctRus, covtype(istest,:), Y(istest))  
  
L =  
  
    0.1833
```

The predictive accuracy on new data might differ, because the ensemble accuracy might be biased. The bias arises because the same data used for assessing the ensemble was used for reducing the ensemble size. To obtain an unbiased estimate of requisite ensemble size, you should use cross validation. However, that procedure is time consuming.

## See Also

[compact](#) | [confusionmat](#) | [cvpartition](#) | [cvpartition](#) | [fitcensemble](#) | [loss](#) | [predict](#) | [removeLearners](#) | [tabulate](#) | [templateTree](#) | [test](#) | [training](#)

## Related Examples

- “Surrogate Splits” on page 18-121
- “Ensemble Algorithms” on page 18-51
- “Test Ensemble Quality” on page 18-81
- “Handle Imbalanced Data or Unequal Misclassification Costs in Classification Ensembles” on page 18-107

- “Classification with Many Categorical Levels” on page 18-115
- “LPBoost and TotalBoost for Small Ensembles” on page 18-124
- “Tune RobustBoost” on page 18-131

## Handle Imbalanced Data or Unequal Misclassification Costs in Classification Ensembles

In many real-world applications, you might prefer to treat classes in your data asymmetrically. For example, you might have data with many more observations of one class than of any other. Or you might work on a problem in which misclassifying observations of one class has more severe consequences than misclassifying observations of another class. In such situations, you can use two optional parameters for `fitcensemble`: `prior` and `cost`.

By using `prior`, you set prior class probabilities (that is, class probabilities used for training). Use this option if some classes are under- or overrepresented in your training set. For example, you might obtain your training data by simulation. Because simulating class A is more expensive than class B, you opt to generate fewer observations of class A and more observations of class B. You expect, however, that class A and class B are mixed in a different proportion in the real world. In this case, set prior probabilities for class A and B approximately to the values you expect to observe in the real world. `fitcensemble` normalizes prior probabilities to make them add up to 1; multiplying all prior probabilities by the same positive factor does not affect the result of classification.

If classes are adequately represented in the training data but you want to treat them asymmetrically, use the `cost` parameter. Suppose you want to classify benign and malignant tumors in cancer patients. Failure to identify a malignant tumor (false negative) has far more severe consequences than misidentifying benign as malignant (false positive). You should assign high cost to misidentifying malignant as benign and low cost to misidentifying benign as malignant.

You must pass misclassification costs as a square matrix with nonnegative elements. Element  $C(i, j)$  of this matrix is the cost of classifying an observation into class  $j$  if the true class is  $i$ . The diagonal elements  $C(i, i)$  of the cost matrix must be 0. For the previous example, you can choose malignant tumor to be class 1 and benign tumor to be class 2. Then you can set the cost matrix to

$$\begin{bmatrix} 0 & c \\ 1 & 0 \end{bmatrix}$$

where  $c > 1$  is the cost of misidentifying a malignant tumor as benign. Costs are relative—multiplying all costs by the same positive factor does not affect the result of classification.

If you have only two classes, `fitcensemble` adjusts their prior probabilities using

$\tilde{P}_i = C_{ij}P_i$  for class  $i = 1, 2$  and  $j \neq i$ .  $P_i$  are prior probabilities either passed into `fitcensemble` or computed from class frequencies in the training data, and  $\tilde{P}_i$  are adjusted prior probabilities. Then `fitcensemble` uses the default cost matrix

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

and these adjusted probabilities for training its weak learners. Manipulating the cost matrix is thus equivalent to manipulating the prior probabilities.

If you have three or more classes, `fitcensemble` also converts input costs into adjusted prior probabilities. This conversion is more complex. First, `fitcensemble` attempts to solve a matrix equation described in Zhou and Liu [51]. If it fails to find a solution, `fitcensemble` applies the “average cost” adjustment described in Breiman et al. [11]. For more information, see Zadrozny, Langford, and Abe [50].

## Train Ensemble With Unequal Classification Costs

This example shows how to train an ensemble of classification trees with unequal classification costs. This example uses data on patients with hepatitis to see if they live or die as a result of the disease. The data set is described at UCI Machine Learning Data Repository.

Read the hepatitis data set from the UCI repository as a character array. Then convert the result to a cell array of character vectors using `textscan`. Specify a cell array of character vectors containing the variable names.

```
hepatitis = textscan(urlread(['http://archive.ics.uci.edu/ml/' ...
    'machine-learning-databases/hepatitis/hepatitis.data']), ...
    '%f%f%f%f%f%f%f%f%f%f%f%f%f%f%f%f', 'TreatAsEmpty', '?', ...
    'Delimiter', ',');
size(hepatitis)
VarNames = {'dieOrLive' 'age' 'sex' 'steroid' 'antivirals' 'fatigue' ...
    'malaise' 'anorexia' 'liverBig' 'liverFirm' 'spleen' ...
    'spiders' 'ascites' 'varices' 'bilirubin' 'alkPhosphate' 'sgot' ...
    'albumin' 'protime' 'histology'};

ans =
```

1 20

hepatitis is a 1-by-20 cell array of character vectors. The cells correspond to the response (`liveOrDie`) and 19 heterogeneous predictors.

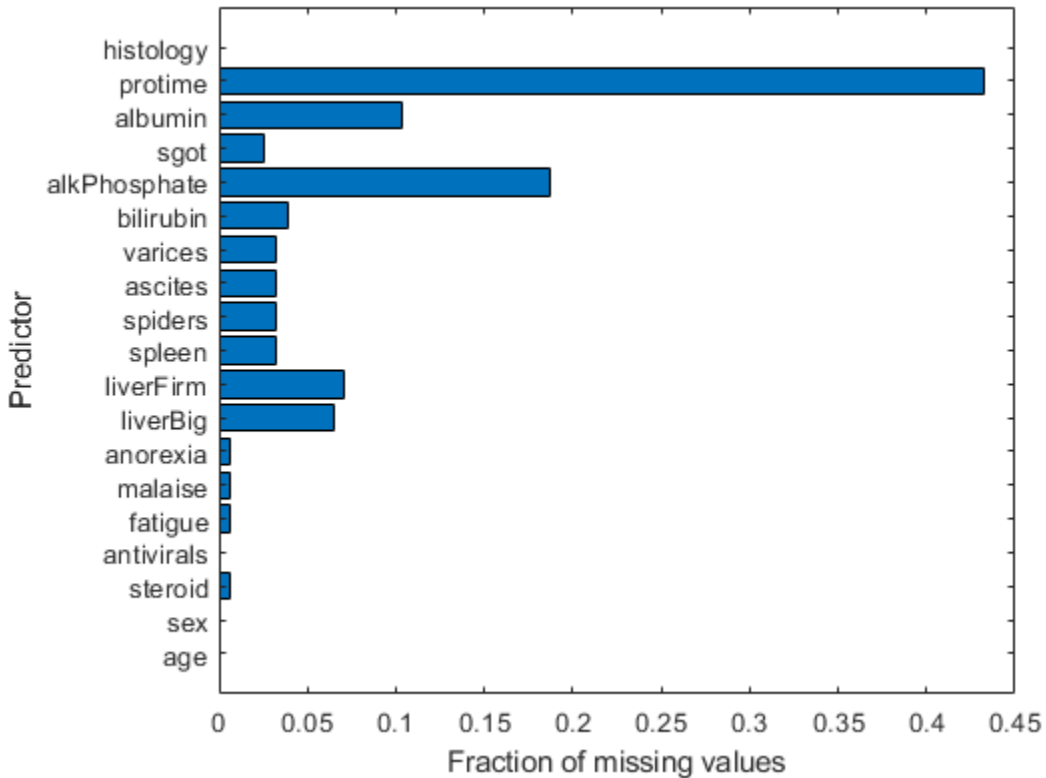
Specify a numeric matrix containing the predictors and a cell vector containing 'Die' and 'Live', which are response categories. The response contains two values: 1 indicates that a patient died, and 2 indicates that a patient lived. Specify a cell array of character vectors for the response using the response categories. The first variable in `hepatitis` contains the response.

```
X = cell2mat(hepatitis(2:end));  
ClassNames = {'Die' 'Live'};  
Y = ClassNames(hepatitis{:},1);
```

`X` is a numeric matrix containing the 19 predictors. `Y` is a cell array of character vectors containing the response.

Inspect the data for missing values.

```
figure;  
barh(sum(isnan(X),1)/size(X,1));  
h = gca;  
h.YTick = 1:numel(VarNames) - 1;  
h.YTickLabel = VarNames(2:end);  
ylabel 'Predictor';  
xlabel 'Fraction of missing values';
```



Most predictors have missing values, and one has nearly 45% of the missing values. Therefore, use decision trees with surrogate splits for better accuracy. Because the data set is small, training time with surrogate splits should be tolerable.

Create a classification tree template that uses surrogate splits.

```
rng(0, 'twister') % for reproducibility
t = templateTree('surrogate', 'all');
```

Examine the data or the description of the data to see which predictors are categorical.

```
X(1:5, :)
```

```
ans =
```

Columns 1 through 7

30.0000	2.0000	1.0000	2.0000	2.0000	2.0000	2.0000
50.0000	1.0000	1.0000	2.0000	1.0000	2.0000	2.0000
78.0000	1.0000	2.0000	2.0000	1.0000	2.0000	2.0000
31.0000	1.0000	NaN	1.0000	2.0000	2.0000	2.0000
34.0000	1.0000	2.0000	2.0000	2.0000	2.0000	2.0000

Columns 8 through 14

1.0000	2.0000	2.0000	2.0000	2.0000	2.0000	1.0000
1.0000	2.0000	2.0000	2.0000	2.0000	2.0000	0.9000
2.0000	2.0000	2.0000	2.0000	2.0000	2.0000	0.7000
2.0000	2.0000	2.0000	2.0000	2.0000	2.0000	0.7000
2.0000	2.0000	2.0000	2.0000	2.0000	2.0000	1.0000

Columns 15 through 19

85.0000	18.0000	4.0000	NaN	1.0000
135.0000	42.0000	3.5000	NaN	1.0000
96.0000	32.0000	4.0000	NaN	1.0000
46.0000	52.0000	4.0000	80.0000	1.0000
NaN	200.0000	4.0000	NaN	1.0000

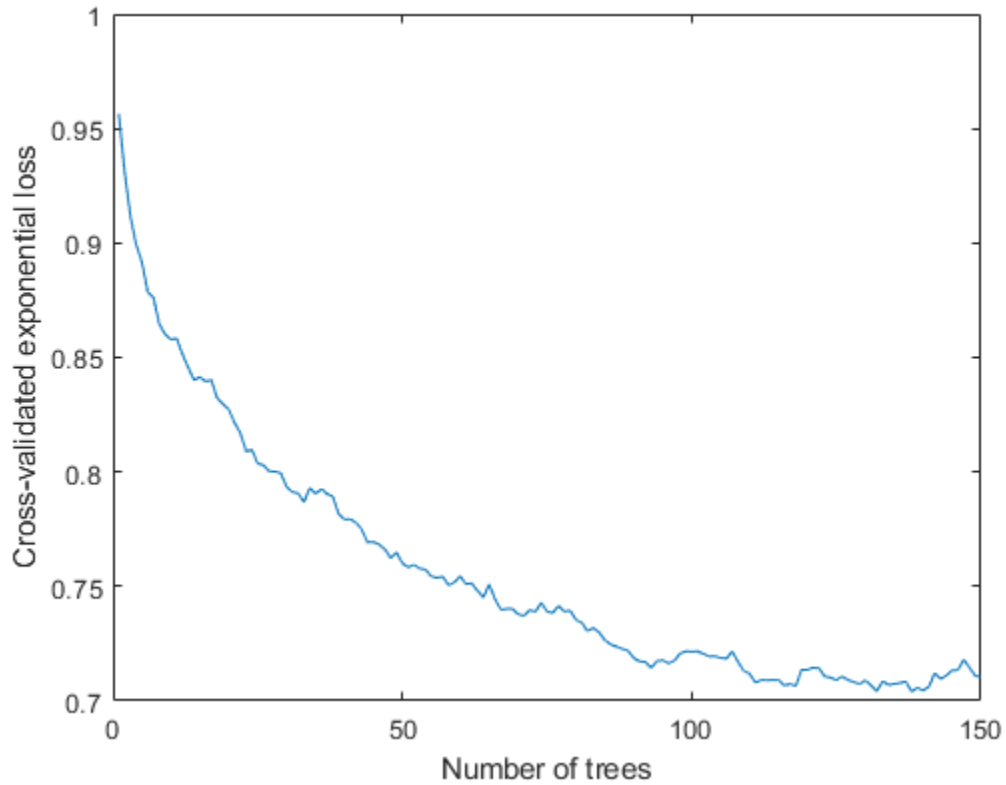
It appears that predictors 2 through 13 are categorical, as well as predictor 19. You can confirm this inference using the data set description at [UCI Machine Learning Data Repository](#).

List the categorical variables.

```
catIdx = [2:13,19];
```

Create a cross-validated ensemble using 150 learners and the GentleBoost algorithm.

```
Ensemble = fitcensemble(X,Y,'Method','GentleBoost', ...
    'NumLearningCycles',150,'Learners',t,'PredictorNames',VarNames(2:end), ...
    'LearnRate',0.1,'CategoricalPredictors',catIdx,'KFold',5);
figure;
plot(kfoldLoss(Ensemble,'Mode','cumulative','LossFun','exponential'));
xlabel('Number of trees');
ylabel('Cross-validated exponential loss');
```



Inspect the confusion matrix to see which patients the ensemble predicts correctly.

```
[yFit,sFit] = kfoldPredict(Ensemble);  
confusionmat(Y,yFit,'Order',ClassNames)
```

```
ans =
```

```
19    13  
11   112
```



Of the 123 patients who live, the ensemble predicts correctly that 112 will live. But for the 32 patients who die of hepatitis, the ensemble only predicts correctly that about half will die of hepatitis.

There are two types of error in the predictions of the ensemble:

- Predicting that the patient lives, but the patient dies
- Predicting that the patient dies, but the patient lives

Suppose you believe that the first error is five times worse than the second. Create a new classification cost matrix that reflects this belief.

```
cost.ClassNames = ClassNames;
cost.ClassificationCosts = [0 5; 1 0];
```

Create a new cross-validated ensemble using `cost` as the misclassification cost, and inspect the resulting confusion matrix.

```
EnsembleCost = fitcensemble(X,Y,'Method','GentleBoost', ...
    'NumLearningCycles',150,'Learners',t,'PredictorNames',VarNames(2:end), ...
    'LearnRate',0.1,'CategoricalPredictors',catIdx,'KFold',5,'Cost',cost);
[yFitCost,sFitCost] = kfoldPredict(EnsembleCost);
confusionmat(Y,yFitCost,'Order',ClassNames)
```

```
ans =
    20    12
     8   115
```

As expected, the new ensemble does a better job classifying the patients who die. Somewhat surprisingly, the new ensemble also does a better job classifying the patients who live, though the result is not statistically significantly better. The results of the cross validation are random, so this result is simply a statistical fluctuation. The result seems to indicate that the classification of patients who live is not very sensitive to the cost.

## See Also

`confusionmat` | `fitcensemble` | `kfoldLoss` | `kfoldPredict` | `templateTree`

## **Related Examples**

- “Surrogate Splits” on page 18-121
- “Ensemble Algorithms” on page 18-51
- “Test Ensemble Quality” on page 18-81
- “Classification with Imbalanced Data” on page 18-101
- “Classification with Many Categorical Levels” on page 18-115
- “LPBoost and TotalBoost for Small Ensembles” on page 18-124
- “Tune RobustBoost” on page 18-131

## Classification with Many Categorical Levels

This example shows how to train an ensemble of classification trees using data containing predictors with many categorical levels.

Generally, you cannot use classification with more than 31 levels in any categorical predictor. However, two boosting algorithms can classify data with many categorical predictor levels and binary responses: `LogitBoost` and `GentleBoost`. For details, see “`LogitBoost`” on page 18-58 and “`GentleBoost`” on page 18-58.

The example uses the `census1994` data set, which consists of demographic data from the U.S. Census Bureau to predict whether an individual makes over \$50,000 a year.

Load the `census1994` file. Specify a cell array of character vectors containing the variable names.

```
load census1994
VarNames = adultdata.Properties.VariableNames;
```

Some variable names in the `adultdata` table contain the `_` character. Replace instances of `_` with a space.

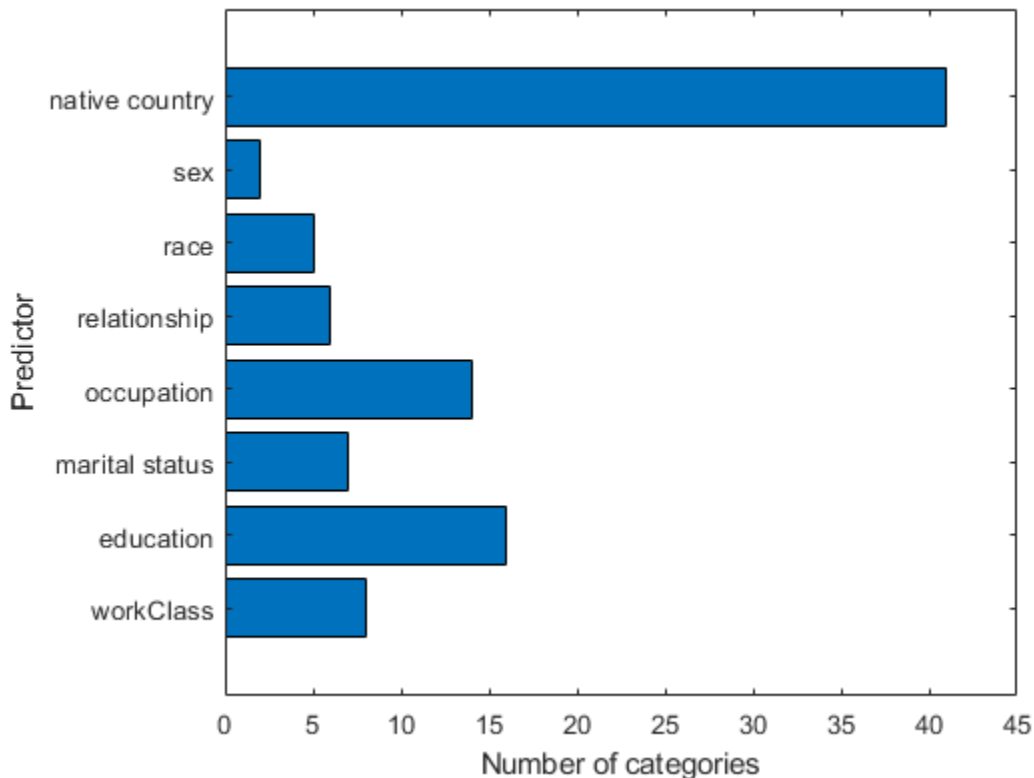
```
VarNames = strrep(VarNames, '_', ' ');
```

Some categorical variables have many levels. Plot the number of levels of each categorical predictor.

```
cat = ~varfun(@isnumeric, adultdata(:, 1:end - 1), ...
    'OutputFormat', 'uniform'); % Logical flag for categorical variables
catVars = find(cat);           % Indices of categorical variables

countCats = @(var) numel(categories(nominal(var)));
numCat = varfun(@(var) countCats(var), adultdata(:, catVars), ...
    'OutputFormat', 'uniform');

figure
barh(numCat);
h = gca;
h.YTickLabel = VarNames(catVars);
ylabel 'Predictor'
xlabel 'Number of categories'
```



The anonymous function `countCats` converts a predictor to a nominal array, then counts the unique, nonempty categories of the predictor. Predictor 14 ('native country') has more than 40 categorical levels. For binary classification, `fitctree` uses a computational shortcut to find an optimal split for categorical predictors with many categories. For classification with more than two classes, you can choose a heuristic algorithm to find a good split. For details, see “Splitting Categorical Predictors” on page 19-42.

Specify the predictor matrix using `classreg.regr.modelutils.predictormatrix` and the response vector.

```
X = classreg.regr.modelutils.predictormatrix(adultdata, 'ResponseVar', ...
    size(adultdata,2));
Y = nominal(adultdata.salary);
```

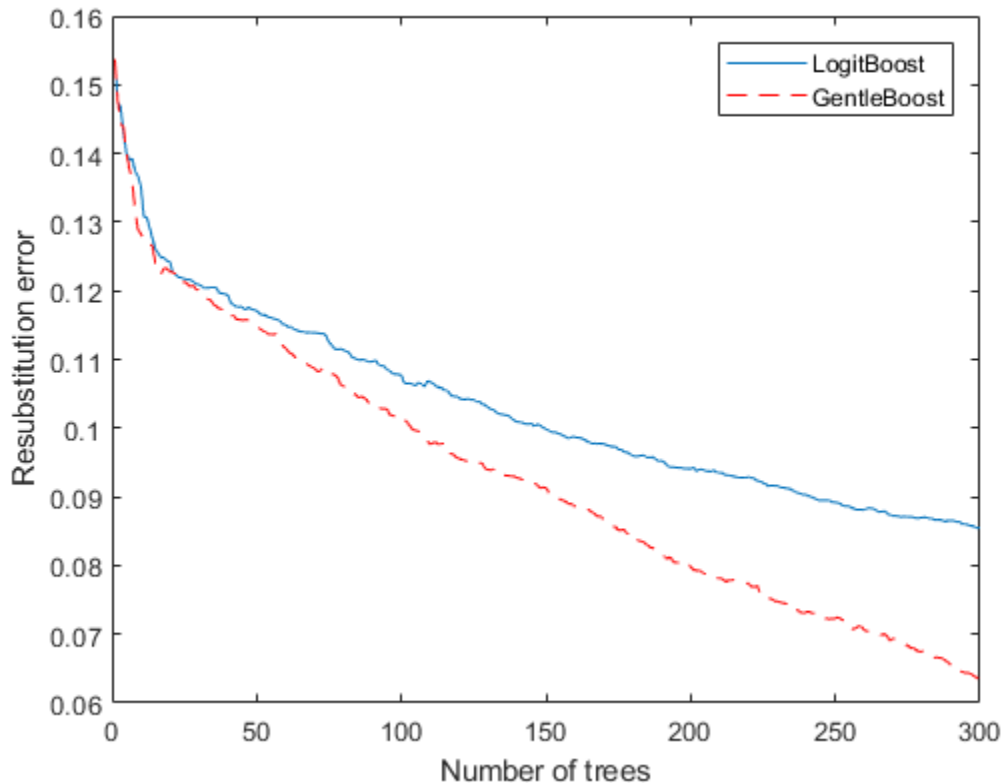
X is a numeric matrix; `predictormatrix` converts all categorical variables into group indices. The name-value pair argument `ResponseVar` indicates that the last column is the response variable, and excludes it from the predictor matrix. Y is a nominal, categorical array.

Train classification ensembles using both `LogitBoost` and `GentleBoost`.

```
rng(1); % For reproducibility
LBEnsemble = fitcensemble(X,Y, 'Method', 'LogitBoost', ...
    'NumLearningCycles', 300, 'Learners', 'Tree', ...
    'CategoricalPredictors', cat, 'PredictorNames', VarNames(1:end-1), ...
    'ResponseName', 'income');
GBEnsemble = fitcensemble(X,Y, 'Method', 'GentleBoost', ...
    'NumLearningCycles', 300, 'Learners', 'Tree', ...
    'CategoricalPredictors', cat, 'PredictorNames', VarNames(1:end-1), ...
    'ResponseName', 'income');
```

Examine the resubstitution error for both ensembles.

```
figure
plot(resubLoss(LBEnsemble, 'Mode', 'cumulative'))
hold on
plot(resubLoss(GBEnsemble, 'Mode', 'cumulative'), 'r--')
hold off
xlabel('Number of trees')
ylabel('Resubstitution error')
legend('LogitBoost', 'GentleBoost', 'Location', 'NE')
```

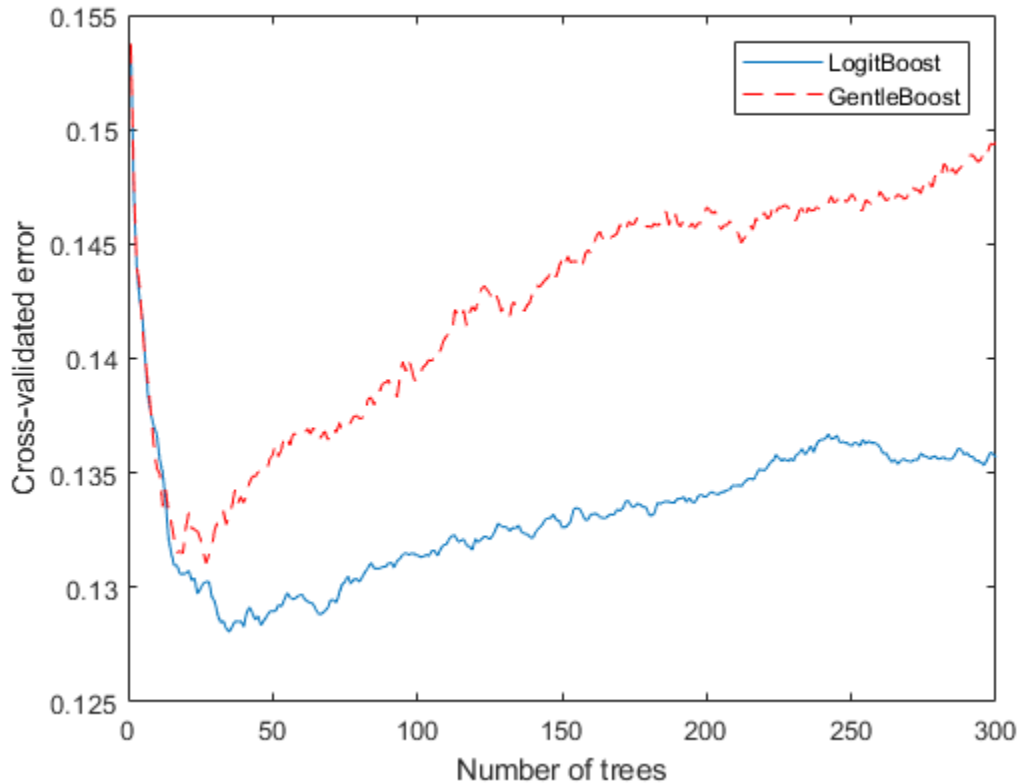


For both ensembles, the resubstitution error continues to decrease as the number of trees increases.

Estimate the generalization error for both algorithms by cross validation.

```
CVLBEnsemble = crossval(LBEnsemble, 'KFold', 5);
CVGBEnsemble = crossval(GBEnsemble, 'KFold', 5);
figure
plot(kfoldLoss(CVLBEnsemble, 'Mode', 'cumulative'))
hold on
plot(kfoldLoss(CVGBEnsemble, 'Mode', 'cumulative'), 'r--')
hold off
xlabel('Number of trees')
```

```
ylabel('Cross-validated error')  
legend('LogitBoost', 'GentleBoost', 'Location', 'NE')
```



As the number of trees increases to 300, the cross-validated loss decreases up to a point, and then starts to increase. A guideline for determining the number of trees to use in the ensemble is to stop adding more trees when the validation error stops improving and starts increasing. This happens after about 50 trees for `LogitBoost` and 25 trees for `GentleBoost`.

## See Also

`crossval` | `fitcensemble` | `resubLoss`

## Related Examples

- “Surrogate Splits” on page 18-121
- “Ensemble Algorithms” on page 18-51
- “Test Ensemble Quality” on page 18-81
- “Handle Imbalanced Data or Unequal Misclassification Costs in Classification Ensembles” on page 18-107
- “Classification with Imbalanced Data” on page 18-101
- “LPBoost and TotalBoost for Small Ensembles” on page 18-124
- “Tune RobustBoost” on page 18-131



## Surrogate Splits

When you have missing data, trees and ensembles of trees give better predictions when they include surrogate splits. Furthermore, estimates of predictor importance are often different with surrogate splits. Eliminating unimportant predictors can save time and memory for predictions, and can make predictions easier to understand.

This example shows the effects of surrogate splits for predictions for data containing missing entries in the test set.

Load sample data. Partition it into a training and test set.

```
load ionosphere;

rng(10) % for reproducibility
cv = cvpartition(Y, 'Holdout', 0.3);
Xtrain = X(training(cv), :);
Ytrain = Y(training(cv));
Xtest = X(test(cv), :);
Ytest = Y(test(cv));
```

Bag decision trees with and without surrogate splits.

```
b = fitcensemble(Xtrain, Ytrain, 'Method', 'Bag', 'NumLearningCycles', 50);

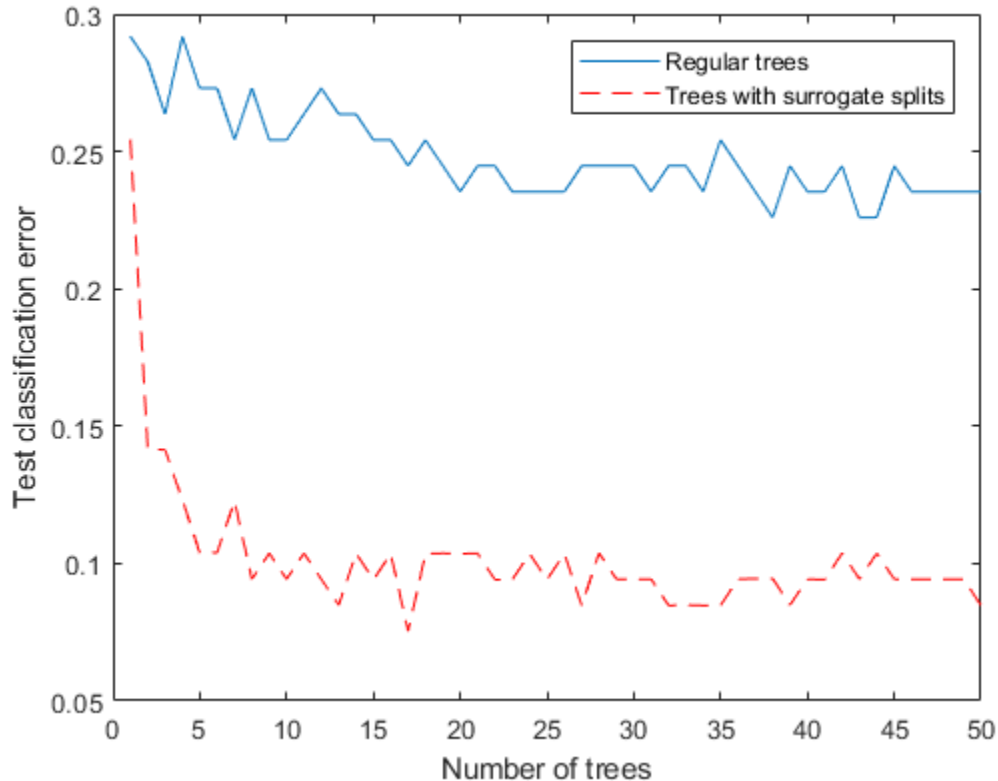
templS = templateTree('Surrogate', 'On');
bs = fitcensemble(Xtrain, Ytrain, 'Method', 'Bag', 'NumLearningCycles', 50, ...
    'Learners', templS);
```

Suppose half of the values in the test set are missing.

```
Xtest(rand(size(Xtest))>0.5) = NaN;
```

Test accuracy with and without surrogate splits.

```
figure;
plot(loss(b, Xtest, Ytest, 'Mode', 'Cumulative'));
hold on;
plot(loss(bs, Xtest, Ytest, 'Mode', 'Cumulative'), 'r--');
legend('Regular trees', 'Trees with surrogate splits');
xlabel('Number of trees');
ylabel('Test classification error');
```



Check the statistical significance of the difference in results with the McNemar test. Convert the labels to a nominal data type to make it easier to check for equality.

```
Yfit = nominal(predict(b,Xtest));
YfitS = nominal(predict(bs,Xtest));
N10 = sum(Yfit==nominal(Ytest) & YfitS~=nominal(Ytest));
N01 = sum(Yfit~=nominal(Ytest) & YfitS==nominal(Ytest));
mcnemar = (abs(N10-N01) - 1)^2/(N10+N01);
pval = 1 - chi2cdf(mcnemar,1)
```

```
pval =
```

1.7683e-04

The extremely low  $p$ -value indicates that the ensemble with surrogate splits is better in a statistically significant manner.

## See Also

`fitcensemble` | `fitrensemble`

## Related Examples

- “Ensemble Algorithms” on page 18-51
- “Test Ensemble Quality” on page 18-81
- “Handle Imbalanced Data or Unequal Misclassification Costs in Classification Ensembles” on page 18-107
- “Classification with Imbalanced Data” on page 18-101
- “Classification with Many Categorical Levels” on page 18-115
- “LPBoost and TotalBoost for Small Ensembles” on page 18-124
- “Tune RobustBoost” on page 18-131

## LPBoost and TotalBoost for Small Ensembles

This example shows how to obtain the benefits of the LPBoost and TotalBoost algorithms. These algorithms share two beneficial characteristics:

- They are self-terminating, which means you do not have to figure out how many members to include.
- They produce ensembles with some very small weights, enabling you to safely remove ensemble members.

### Load the data

Load the ionosphere data set.

```
load ionosphere
```

### Create the classification ensembles

Create ensembles for classifying the ionosphere data using the LPBoost, TotalBoost, and, for comparison, AdaBoostM1 algorithms. It is hard to know how many members to include in an ensemble. For LPBoost and TotalBoost, try using 500. For comparison, also use 500 for AdaBoostM1.

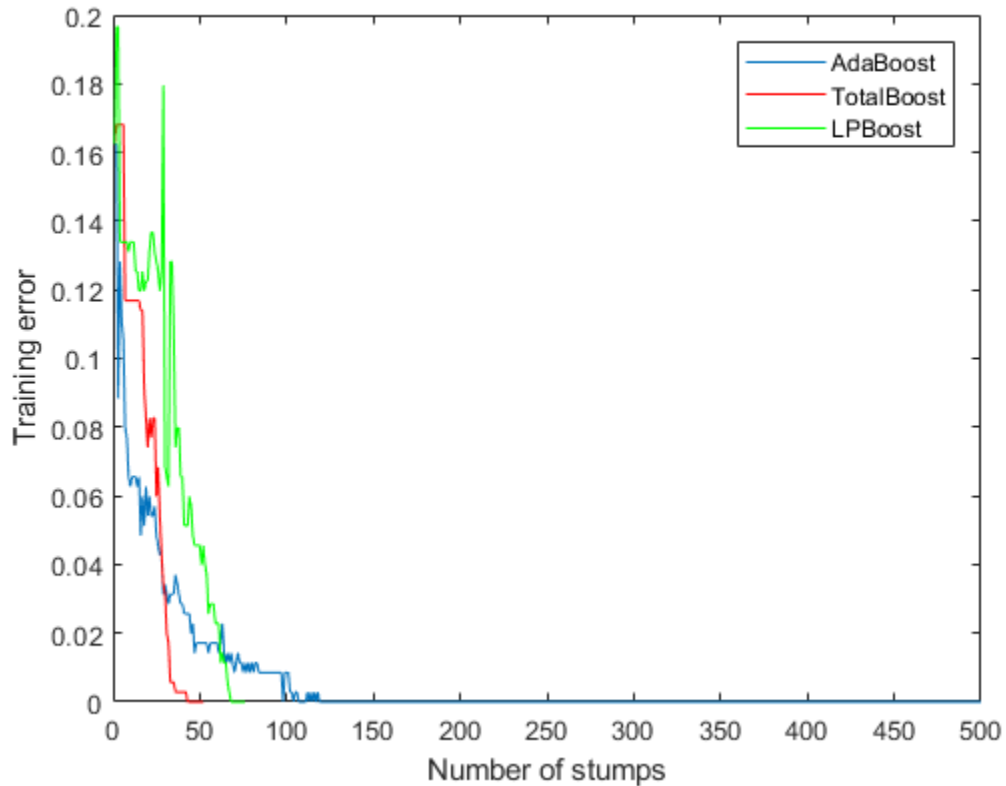
The default weak learners for boosting methods are decision trees with the `MaxNumSplits` property set to 10. These trees tend to fit better than tree stumps (with 1 maximum split) and may overfit more. Therefore, to prevent overfitting, use tree stumps as weak learners for the ensembles.

```
rng default % For reproducibility
T = 500;
treeStump = templateTree('MaxNumSplits',1);
adaStump = fitcensemble(X,Y,'Method','AdaBoostM1','NumLearningCycles',T, ...
    'Learners',treeStump);
totalStump = fitcensemble(X,Y,'Method','TotalBoost','NumLearningCycles',T, ...
    'Learners',treeStump);
lpStump = fitcensemble(X,Y,'Method','LPBoost','NumLearningCycles',T, ...
    'Learners',treeStump);
figure;
plot(resubLoss(adaStump,'Mode','Cumulative'));
hold on
plot(resubLoss(totalStump,'Mode','Cumulative'),'r');
plot(resubLoss(lpStump,'Mode','Cumulative'),'g');
```

```

hold off
xlabel('Number of stumps');
ylabel('Training error');
legend('AdaBoost','TotalBoost','LPBoost','Location','NE');

```



All three algorithms achieve perfect prediction on the training data after a while.

Examine the number of members in all three ensembles.

```
[adaStump.NTrained totalStump.NTrained lpStump.NTrained]
```

```
ans =
```

500    52    76

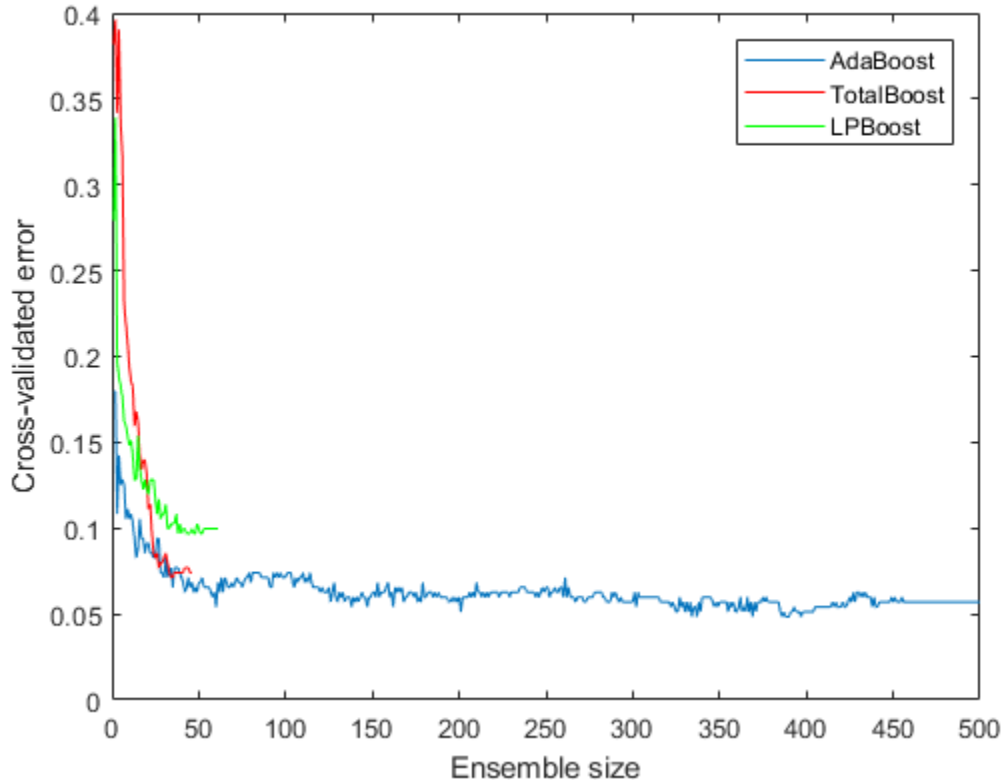
AdaBoostM1 trained all 500 members. The other two algorithms stopped training early.

### Cross validate the ensembles

Cross validate the ensembles to better determine ensemble accuracy.

```
cvlp = crossval(lpStump, 'Kfold', 5);
cvtotal = crossval(totalStump, 'Kfold', 5);
cvada = crossval(adaStump, 'Kfold', 5);

figure;
plot(kfoldLoss(cvada, 'Mode', 'Cumulative'));
hold on
plot(kfoldLoss(cvtotal, 'Mode', 'Cumulative'), 'r');
plot(kfoldLoss(cvlp, 'Mode', 'Cumulative'), 'g');
hold off
xlabel('Ensemble size');
ylabel('Cross-validated error');
legend('AdaBoost', 'TotalBoost', 'LPBoost', 'Location', 'NE');
```



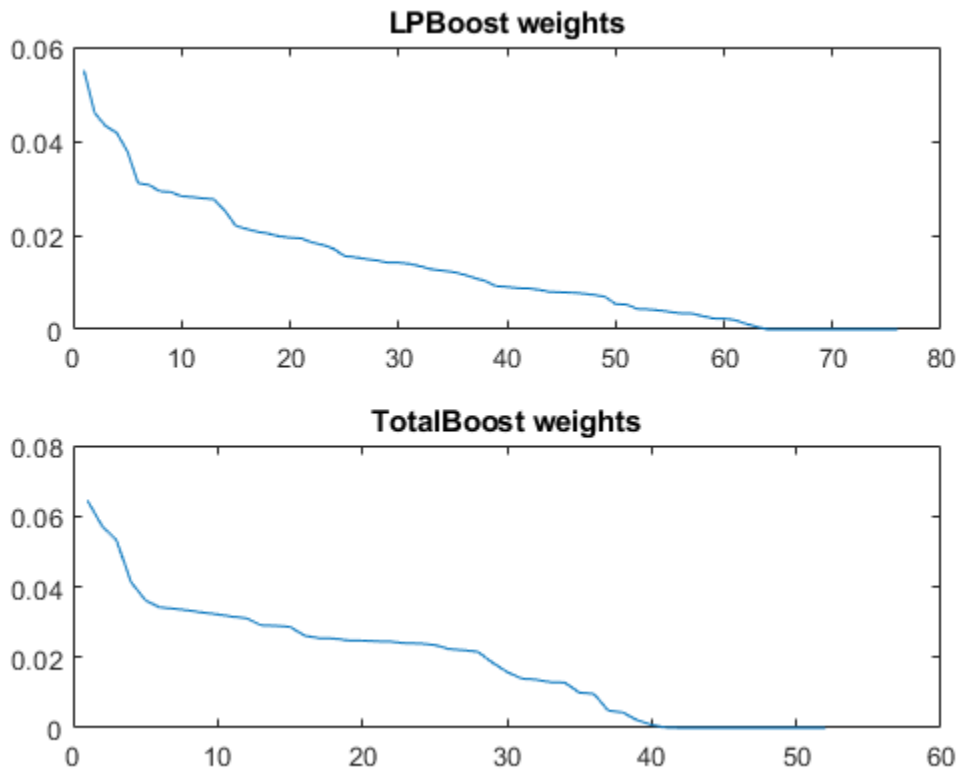
The results show that each boosting algorithm achieves a loss of 10% or lower with 50 ensemble members.

### Compact and remove ensemble members

To reduce the ensemble sizes, compact them, and then use `removeLearners`. The question is, how many learners should you remove? The cross-validated loss curves give you one measure. For another, examine the learner weights for LPBoost and TotalBoost after compacting.

```
cada = compact(adaStump);
clp = compact(lpStump);
ctotal = compact(totalStump);
```

```
figure
subplot(2,1,1)
plot(clp.TrainedWeights)
title('LPBoost weights')
subplot(2,1,2)
plot(ctotal.TrainedWeights)
title('TotalBoost weights')
```



Both LPBoost and TotalBoost show clear points where the ensemble member weights become negligible.

Remove the unimportant ensemble members.



```
cada = removeLearners(cada,150:cada.NTrained);  
clp = removeLearners(clp,60:clp.NTrained);  
ctotal = removeLearners(ctotal,40:ctotal.NTrained);
```

Check that removing these learners does not affect ensemble accuracy on the training data.

```
[loss(cada,X,Y) loss(clp,X,Y) loss(ctotal,X,Y)]
```

```
ans =  
  
    0    0    0
```

Check the resulting compact ensemble sizes.

```
s(1) = whos('cada');  
s(2) = whos('clp');  
s(3) = whos('ctotal');  
s.bytes
```

```
ans =  
  
    503987
```

```
ans =  
  
    201283
```

```
ans =  
  
    134023
```

The sizes of the compact ensembles are approximately proportional to the number of members in each.

## See Also

`compact` | `crossval` | `fitcensemble` | `kfoldLoss` | `loss` | `removeLearners` | `resubLoss`

## Related Examples

- “Surrogate Splits” on page 18-121
- “Ensemble Algorithms” on page 18-51
- “Test Ensemble Quality” on page 18-81
- “Handle Imbalanced Data or Unequal Misclassification Costs in Classification Ensembles” on page 18-107
- “Classification with Imbalanced Data” on page 18-101
- “Classification with Many Categorical Levels” on page 18-115
- “Tune RobustBoost” on page 18-131

## Tune RobustBoost

The `RobustBoost` algorithm can make good classification predictions even when the training data has noise. However, the default `RobustBoost` parameters can produce an ensemble that does not predict well. This example shows one way of tuning the parameters for better predictive accuracy.

Generate data with label noise. This example has twenty uniform random numbers per observation, and classifies the observation as 1 if the sum of the first five numbers exceeds 2.5 (so is larger than average), and 0 otherwise:

```
rng(0, 'twister') % for reproducibility
Xtrain = rand(2000,20);
Ytrain = sum(Xtrain(:,1:5),2) > 2.5;
```

To add noise, randomly switch 10% of the classifications:

```
idx = randsample(2000,200);
Ytrain(idx) = ~Ytrain(idx);
```

Create an ensemble with `AdaBoostM1` for comparison purposes:

```
ada = fitcensemble(Xtrain,Ytrain,'Method','AdaBoostM1', ...
    'NumLearningCycles',300,'Learners','Tree','LearnRate',0.1);
```

Create an ensemble with `RobustBoost`. Because the data has 10% incorrect classification, perhaps an error goal of 15% is reasonable.

```
rb1 = fitcensemble(Xtrain,Ytrain,'Method','RobustBoost', ...
    'NumLearningCycles',300,'Learners','Tree','RobustErrorGoal',0.15, ...
    'RobustMaxMargin',1);
```

Note that if you set the error goal to a high enough value, then the software returns an error.

Create an ensemble with very optimistic error goal, 0.01:

```
rb2 = fitcensemble(Xtrain,Ytrain,'Method','RobustBoost', ...
    'NumLearningCycles',300,'Learners','Tree','RobustErrorGoal',0.01);
```

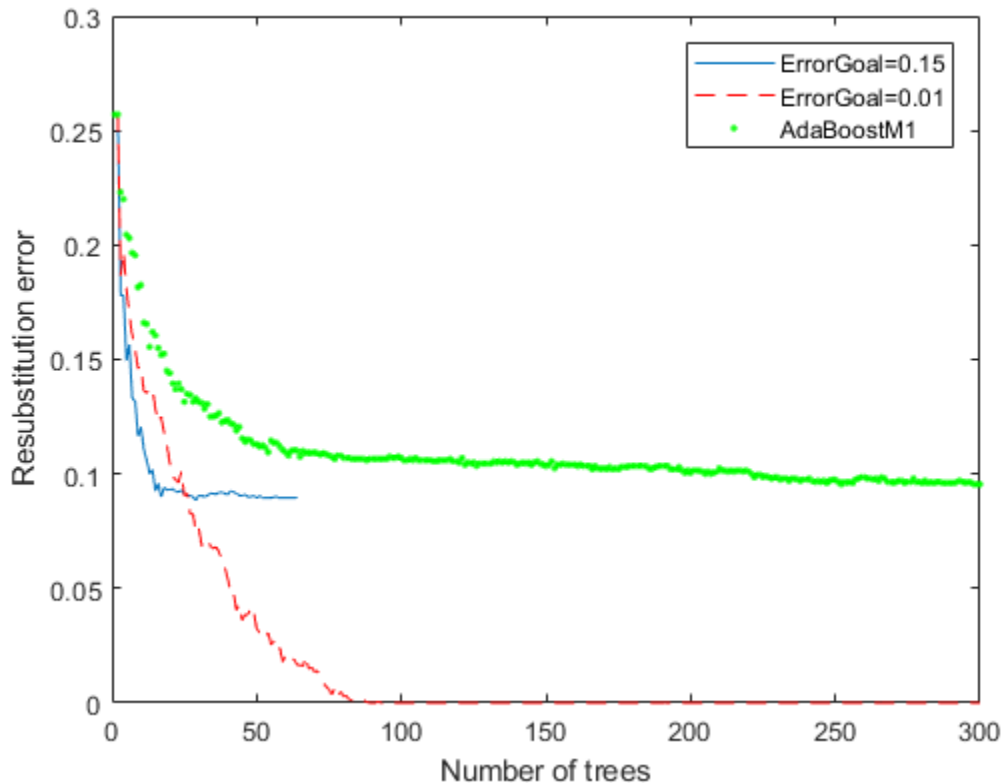
Compare the resubstitution error of the three ensembles:

```
figure
plot(resubLoss(rb1,'Mode','Cumulative'));
```

```

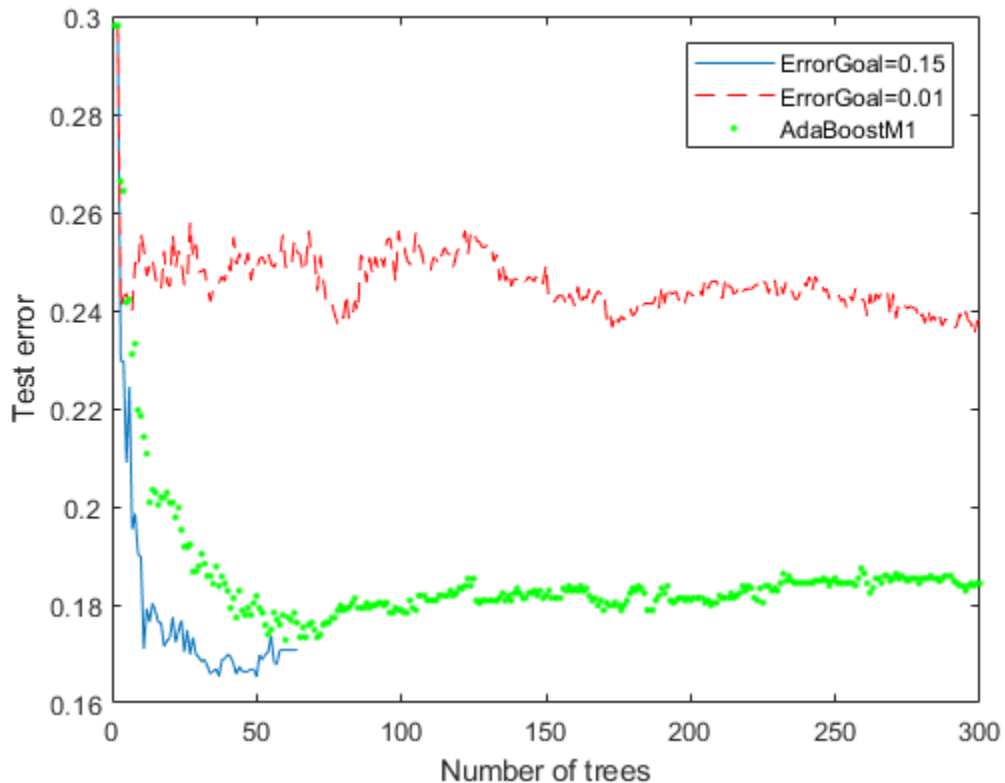
hold on
plot(resubLoss(rb2, 'Mode', 'Cumulative'), 'r--');
plot(resubLoss(ada, 'Mode', 'Cumulative'), 'g.');
hold off;
xlabel('Number of trees');
ylabel('Resubstitution error');
legend('ErrorGoal=0.15', 'ErrorGoal=0.01', ...
      'AdaBoostM1', 'Location', 'NE');

```



All the RobustBoost curves show lower resubstitution error than the AdaBoostM1 curve. The error goal of 0.01 curve shows the lowest resubstitution error over most of the range.

```
Xtest = rand(2000,20);
Ytest = sum(Xtest(:,1:5),2) > 2.5;
idx = randsample(2000,200);
Ytest(idx) = ~Ytest(idx);
figure;
plot(loss(rb1,Xtest,Ytest,'Mode','Cumulative'));
hold on
plot(loss(rb2,Xtest,Ytest,'Mode','Cumulative'),'r--');
plot(loss(ada,Xtest,Ytest,'Mode','Cumulative'),'g.');
hold off;
xlabel('Number of trees');
ylabel('Test error');
legend('ErrorGoal=0.15','ErrorGoal=0.01',...
      'AdaBoostM1','Location','NE');
```



The error curve for error goal 0.15 is lowest (best) in the plotted range. AdaBoostM1 has higher error than the curve for error goal 0.15. The curve for the too-optimistic error goal 0.01 remains substantially higher (worse) than the other algorithms for most of the plotted range.

## See Also

`fitcensemble` | `loss` | `resubLoss`

## Related Examples

- “Surrogate Splits” on page 18-121
- “Ensemble Algorithms” on page 18-51
- “Test Ensemble Quality” on page 18-81
- “Handle Imbalanced Data or Unequal Misclassification Costs in Classification Ensembles” on page 18-107
- “Classification with Imbalanced Data” on page 18-101
- “Classification with Many Categorical Levels” on page 18-115
- “LPBoost and TotalBoost for Small Ensembles” on page 18-124

## Random Subspace Classification

This example shows how to use a random subspace ensemble to increase the accuracy of classification. It also shows how to use cross validation to determine good parameters for both the weak learner template and the ensemble.

### Load the data

Load the ionosphere data. This data has 351 binary responses to 34 predictors.

```
load ionosphere;
[N,D] = size(X)
resp = unique(Y)
```

```
N =

    351

D =

    34

resp =

    2x1 cell array

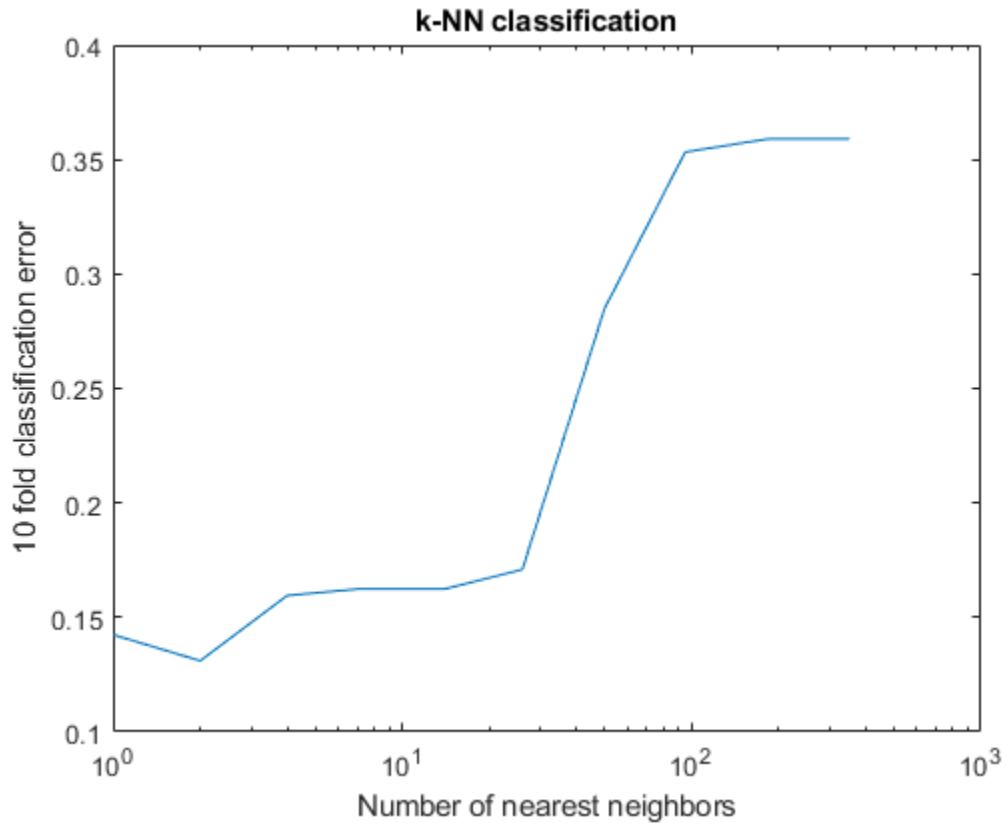
    {'b'}
    {'g'}
```

### Choose the number of nearest neighbors

Find a good choice for  $k$ , the number of nearest neighbors in the classifier, by cross validation. Choose the number of neighbors approximately evenly spaced on a logarithmic scale.

```
rng(8000,'twister') % for reproducibility
K = round(logspace(0,log10(N),10)); % number of neighbors
cvloss = zeros(numel(K),1);
for k=1:numel(K)
    knn = fitcknn(X,Y,...
```

```
    'NumNeighbors',K(k), 'CrossVal', 'On');  
    cvloss(k) = kfoldLoss(knn);  
end  
figure; % Plot the accuracy versus k  
semilogx(K,cvloss);  
xlabel('Number of nearest neighbors');  
ylabel('10 fold classification error');  
title('k-NN classification');
```



The lowest cross-validation error occurs for  $k = 2$ .



## Create the ensembles

Create ensembles for 2-nearest neighbor classification with various numbers of dimensions, and examine the cross-validated loss of the resulting ensembles.

This step takes a long time. To keep track of the progress, print a message as each dimension finishes.

```

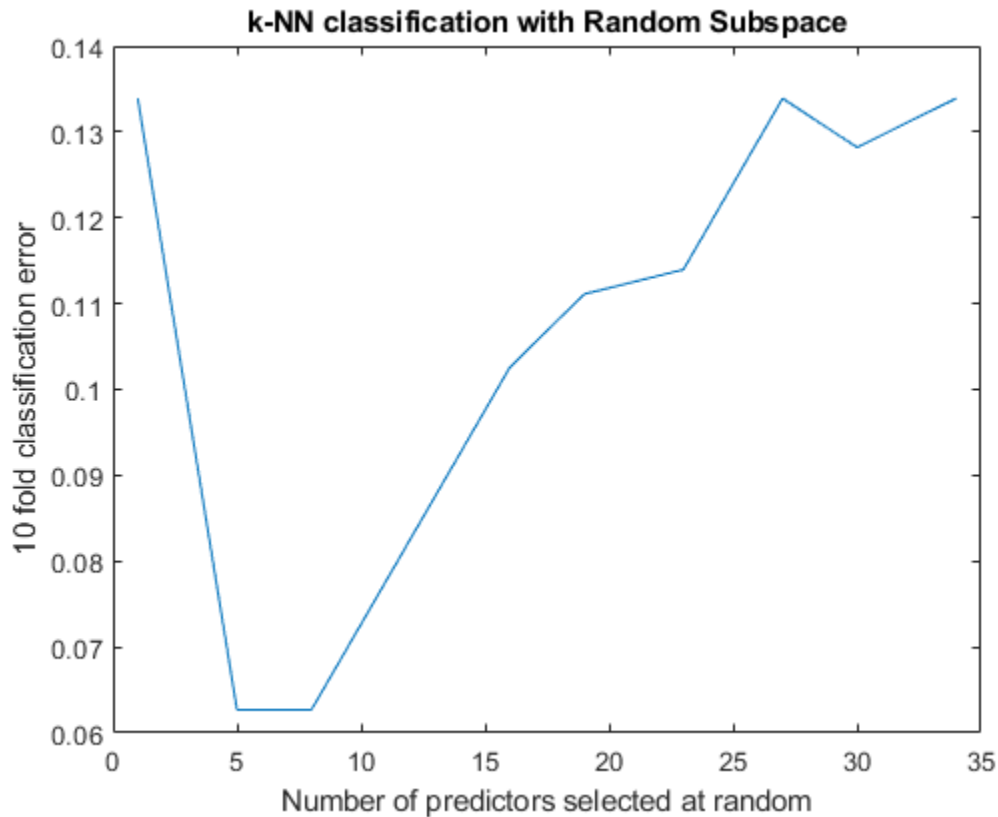
NPredToSample = round(linspace(1,D,10)); % linear spacing of dimensions
cvloss = zeros(numel(NPredToSample),1);
learner = templateKNN('NumNeighbors',2);
for npred=1:numel(NPredToSample)
    subspace = fitcensemble(X,Y,'Method','Subspace','Learners',learner, ...
        'NPredToSample',NPredToSample(npred),'CrossVal','On');
    cvloss(npred) = kfoldLoss(subspace);
    fprintf('Random Subspace %i done.\n',npred);
end
figure; % plot the accuracy versus dimension
plot(NPredToSample,cvloss);
xlabel('Number of predictors selected at random');
ylabel('10 fold classification error');
title('k-NN classification with Random Subspace');

```

```

Random Subspace 1 done.
Random Subspace 2 done.
Random Subspace 3 done.
Random Subspace 4 done.
Random Subspace 5 done.
Random Subspace 6 done.
Random Subspace 7 done.
Random Subspace 8 done.
Random Subspace 9 done.
Random Subspace 10 done.

```



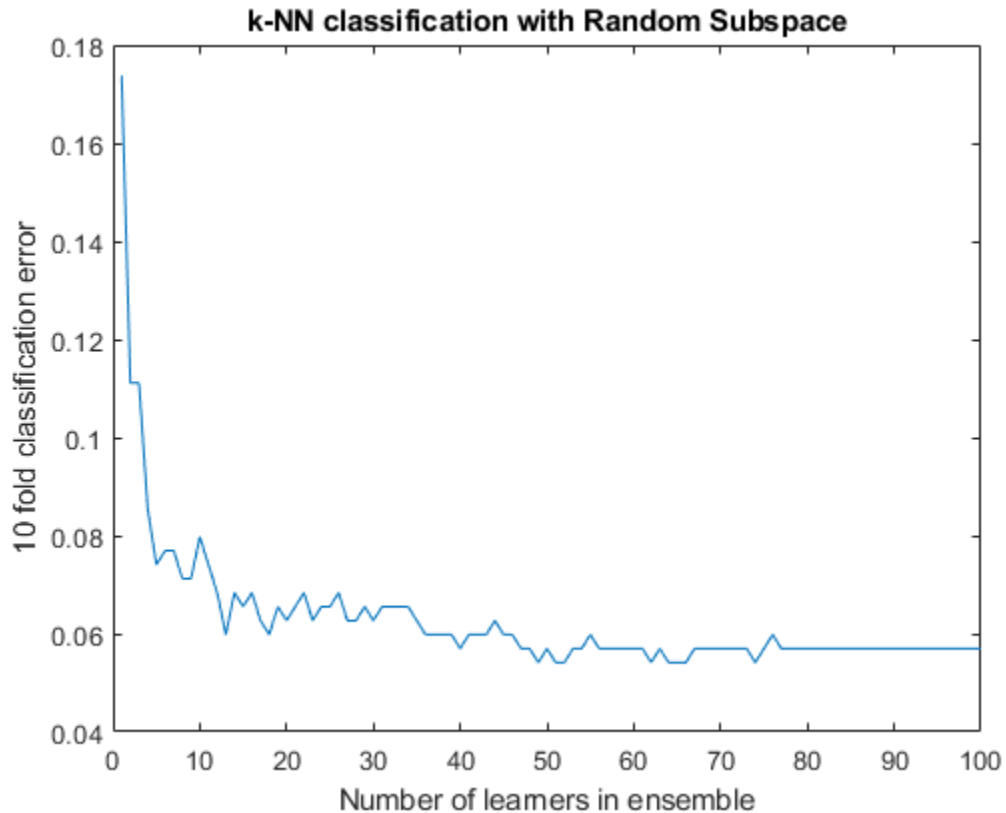
The ensembles that use five and eight predictors per learner have the lowest cross-validated error. The error rate for these ensembles is about 0.06, while the other ensembles have cross-validated error rates that are approximately 0.1 or more.

### Find a good ensemble size

Find the smallest number of learners in the ensemble that still give good classification.

```
ens = fitensemble(X,Y,'Method','Subspace','Learners',learner, ...
    'NPredToSample',5,'CrossVal','on');
figure; % Plot the accuracy versus number in ensemble
plot(kfoldLoss(ens,'Mode','Cumulative'))
xlabel('Number of learners in ensemble');
```

```
ylabel('10 fold classification error');
title('k-NN classification with Random Subspace');
```



There seems to be no advantage in an ensemble with more than 50 or so learners. It is possible that 25 learners gives good predictions.

### Create a final ensemble

Construct a final ensemble with 50 learners. Compact the ensemble and see if the compacted version saves an appreciable amount of memory.

```
ens = fitcensemble(X,Y,'Method','Subspace','NumLearningCycles',50,...
    'Learners',learner,'NPredToSample',5);
cens = compact(ens);
```

```
s1 = whos('ens');  
s2 = whos('cens');  
[s1.bytes s2.bytes] % si.bytes = size in bytes  
  
ans =  
  
    1728705    1499584
```

The compact ensemble is about 10% smaller than the full ensemble. Both give the same predictions.

### See Also

`compact` | `fitcensemble` | `fitcknn` | `kfoldLoss` | `templateKNN`

### Related Examples

- “Framework for Ensemble Learning” on page 18-41
- “Ensemble Algorithms” on page 18-51
- “Train Classification Ensemble” on page 18-67
- “Test Ensemble Quality” on page 18-81

## Bootstrap Aggregation (Bagging) of Regression Trees

`TreeBagger` ensembles have more functionality than those constructed with `fitensemble`; see `TreeBagger` Features Not in `fitensemble` or `fitensemble`. Also, some property and method names differ from their `fitensemble` counterparts. This section contains an example workflow for regression that uses this extra `TreeBagger` functionality.

### Regression of Insurance Risk Rating for Car Imports Using `TreeBagger`

In this example, use a database of 1985 car imports with 205 observations, 25 predictors, and 1 response, which is insurance risk rating, or "symboling." The first 15 variables are numeric and the last 10 are categorical. The symboling index takes integer values from -3 to 3.

Load the data set and split it into predictor and response arrays.

```
load imports-85
Y = X(:,1);
X = X(:,2:end);
isCategorical = [zeros(15,1);ones(size(X,2)-15,1)]; % Categorical variable flag
```

Because bagging uses randomized data drawings, its exact outcome depends on the initial random seed. To reproduce the results in this example, use the random stream settings.

```
rng(1945,'twister')
```

#### Finding the Optimal Leaf Size

For regression, the general rule is to set leaf size to 5 and select one third of the input features for decision splits at random. In the following step, verify the optimal leaf size by comparing mean squared errors obtained by regression for various leaf sizes.

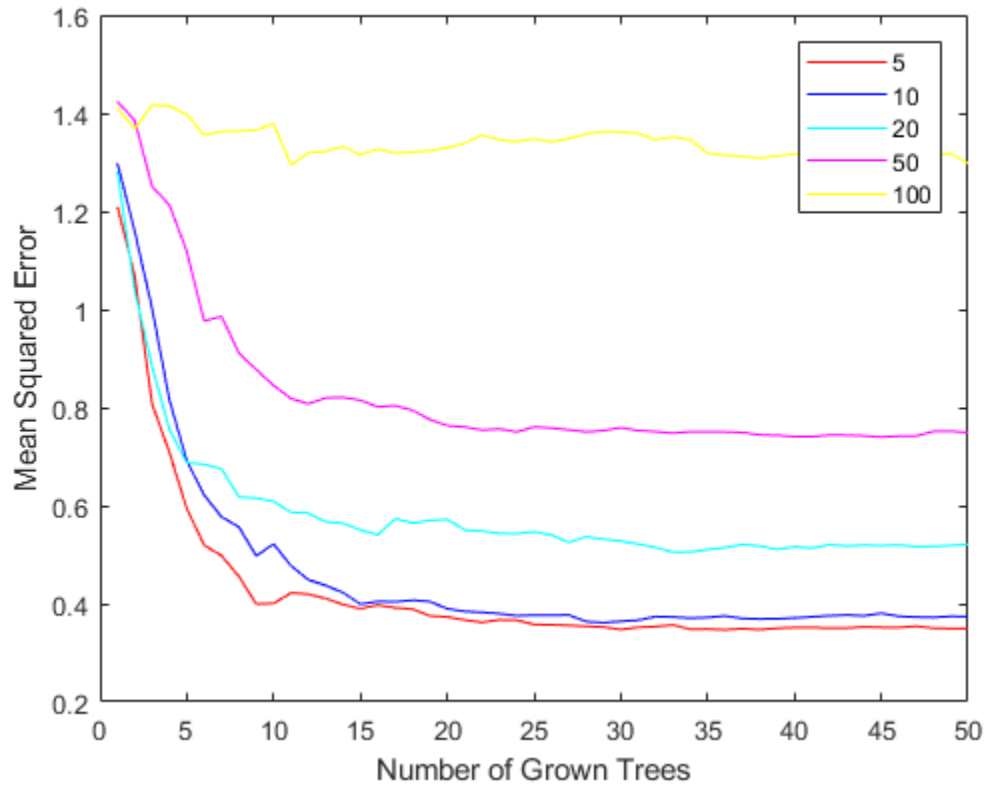
`oobError` computes MSE versus the number of grown trees. You must set `OOBPred` to 'On' to obtain out-of-bag predictions later.

```
leaf = [5 10 20 50 100];
col = 'rbcmY';
figure
for i=1:length(leaf)
    b = TreeBagger(50,X,Y,'Method','R','OOBPred','On',...
```

```

        'CategoricalPredictors', find(isCategorical == 1), ...
        'MinLeafSize', leaf(i));
    plot(oobError(b), col(i))
    hold on
end
xlabel 'Number of Grown Trees'
ylabel 'Mean Squared Error'
legend({'5' '10' '20' '50' '100'}, 'Location', 'NorthEast')
hold off

```



The red curve (leaf size 5) yields the lowest MSE values.

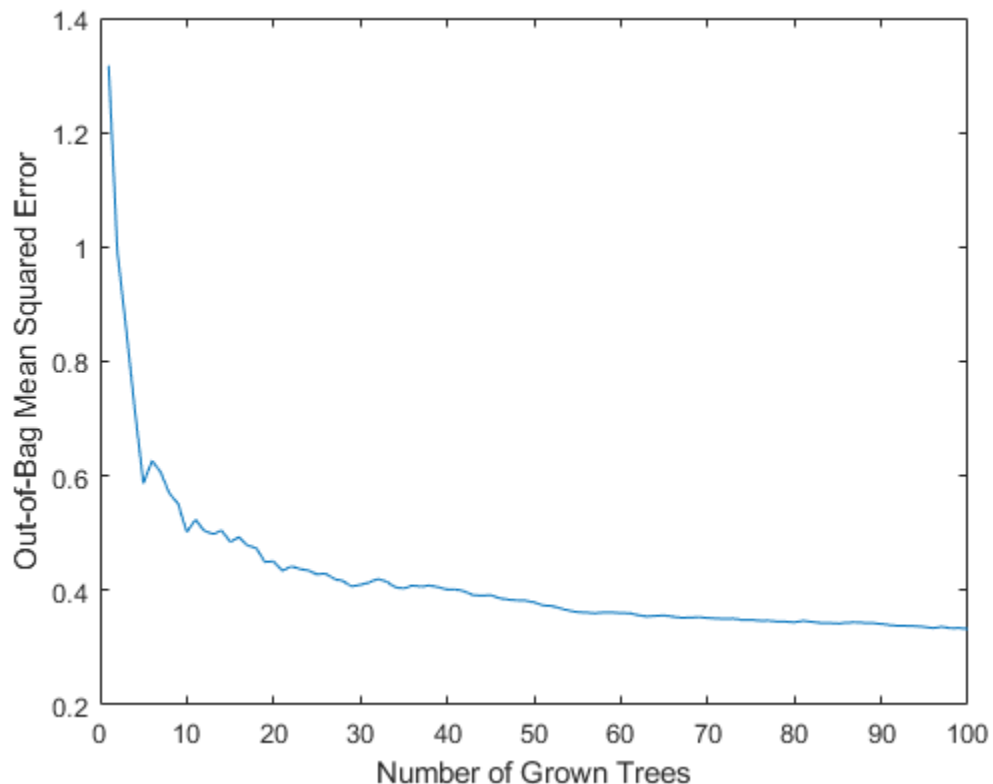
## Estimating Feature Importance

In practical applications, you typically grow ensembles with hundreds of trees. For example, the previous code block uses 50 trees for faster processing. Now that you have estimated the optimal leaf size, grow a larger ensemble with 100 trees and use it to estimate feature importance.

```
b = TreeBagger(100,X,Y,'Method','R','OOBVarImp','On',...  
              'CategoricalPredictors',find(isCategorical == 1),...  
              'MinLeafSize',5);
```

Inspect the error curve again to make sure nothing went wrong during training.

```
figure  
plot(oobError(b))  
xlabel 'Number of Grown Trees'  
ylabel 'Out-of-Bag Mean Squared Error'
```



Prediction ability should depend more on important features than unimportant features. You can use this idea to measure feature importance.

For each feature, permute the values of this feature across every observation in the data set and measure how much worse the MSE becomes after the permutation. You can repeat this for each feature.

Plot the increase in MSE due to permuting out-of-bag observations across each input variable. The `OOBPermutedVarDeltaError` array stores the increase in MSE averaged over all trees in the ensemble and divided by the standard deviation taken over the trees, for each variable. The larger this value, the more important the variable. Imposing an arbitrary cutoff at 0.7, you can select the four most important features.



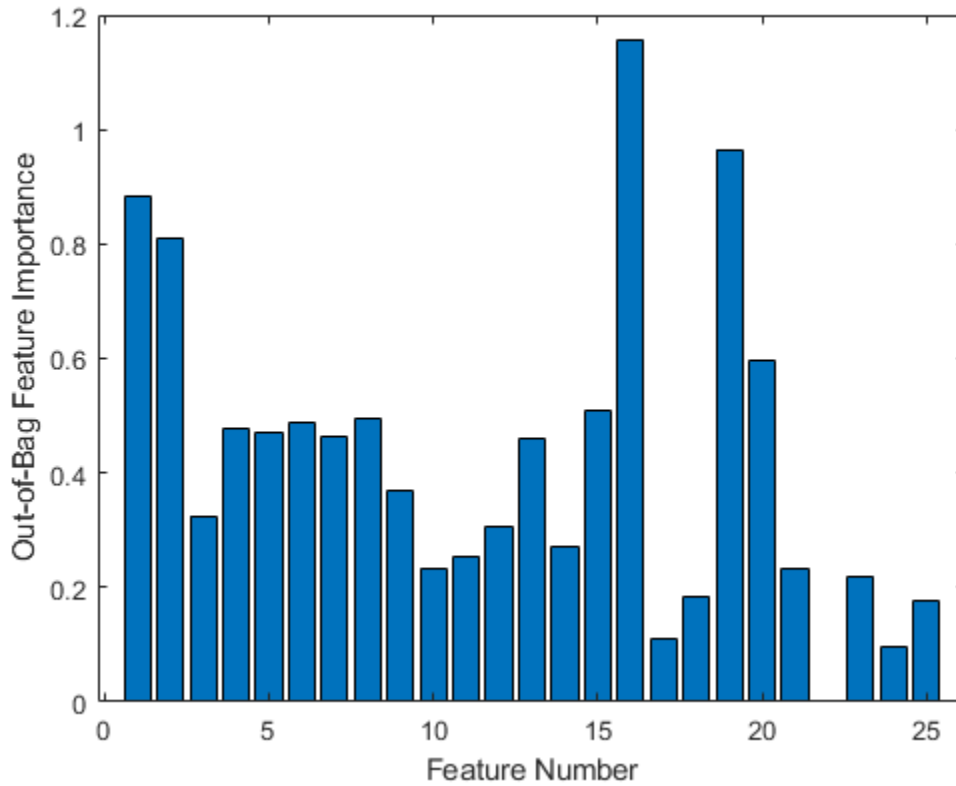
```

figure
bar(b.OOBPermutedVarDeltaError)
xlabel 'Feature Number'
ylabel 'Out-of-Bag Feature Importance'
idxvar = find(b.OOBPermutedVarDeltaError>0.7)
idxCategorical = find(isCategorical(idxvar)==1);

```

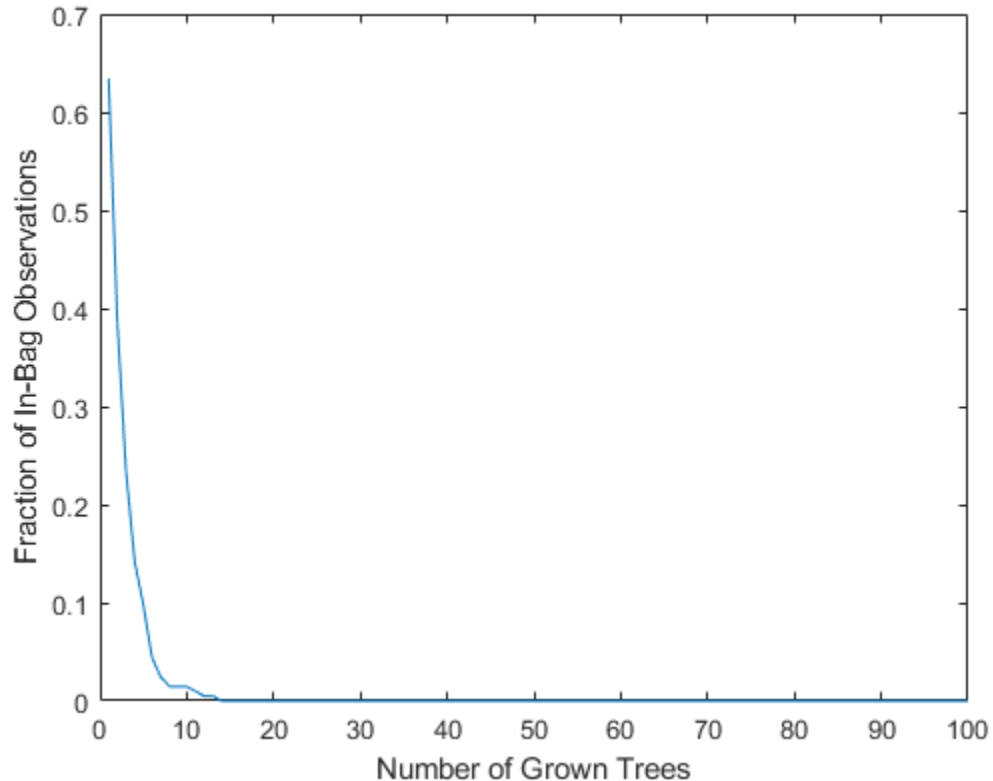
idxvar =

1 2 16 19



The `OOBIndices` property of `TreeBagger` tracks which observations are out of bag for what trees. Using this property, you can monitor the fraction of observations in the training data that are in bag for all trees. The curve starts at approximately  $2/3$ , which is the fraction of unique observations selected by one bootstrap replica, and goes down to 0 at approximately 10 trees.

```
finbag = zeros(1,b.NTrees);  
for t=1:b.NTrees  
    finbag(t) = sum(all(~b.OOBIndices(:,1:t),2));  
end  
finbag = finbag / size(X,1);  
figure  
plot(finbag)  
xlabel 'Number of Grown Trees'  
ylabel 'Fraction of In-Bag Observations'
```

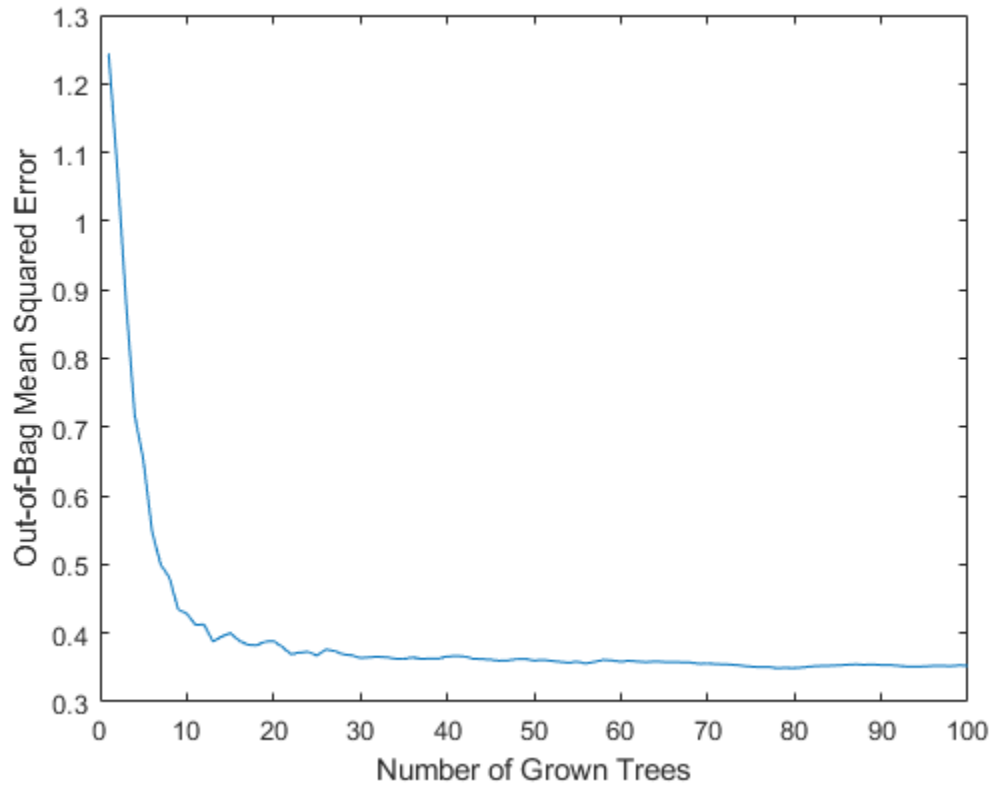


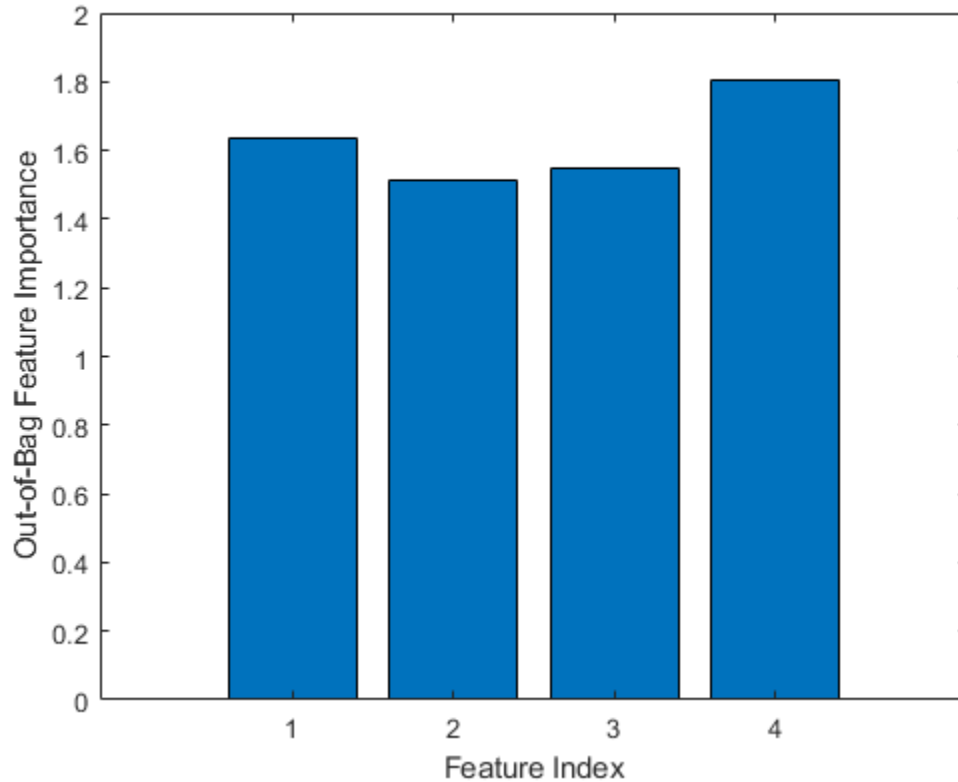
### Growing Trees on a Reduced Set of Features

Using just the four most powerful features, determine if it is possible to obtain a similar predictive power. To begin, grow 100 trees on these features only. The first two of the four selected features are numeric and the last two are categorical.

```
b5v = TreeBagger(100,X(:,idxvar),Y,'Method','R',...
    'OOBVarImp','On','CategoricalPredictors',idxCategorical,...
    'MinLeafSize',5);
figure
plot(oobError(b5v))
xlabel 'Number of Grown Trees'
ylabel 'Out-of-Bag Mean Squared Error'
figure
```

```
bar(b5v.OOBPermutedVarDeltaError)
xlabel 'Feature Index'
ylabel 'Out-of-Bag Feature Importance'
```





These four most powerful features give the same MSE as the full set, and the ensemble trained on the reduced set ranks these features similarly to each other. If you remove features 1 and 2 from the reduced set, then the predictive power of the algorithm might not decrease significantly.

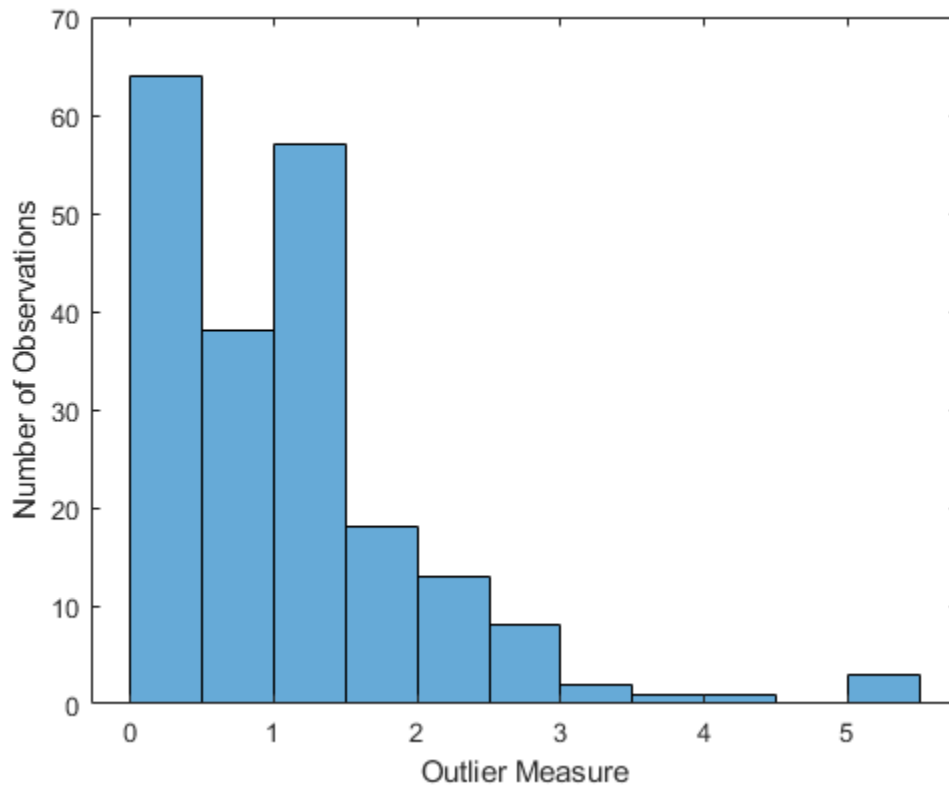
### Finding Outliers

To find outliers in the training data, compute the proximity matrix using `fillProximities`.

```
b5v = fillProximities(b5v);
```

The method normalizes this measure by subtracting the mean outlier measure for the entire sample. Then it takes the magnitude of this difference and divides the result by the median absolute deviation for the entire sample.

```
figure
histogram(b5v.OutlierMeasure)
xlabel 'Outlier Measure'
ylabel 'Number of Observations'
```

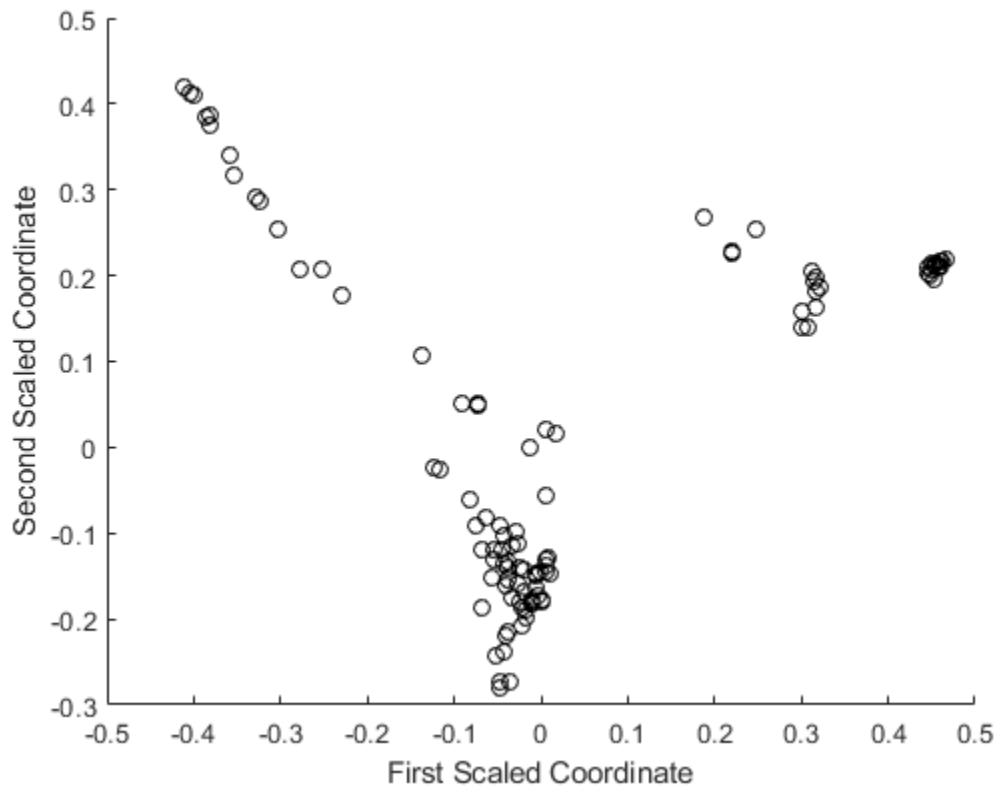


### Discovering Clusters in the Data

By applying multidimensional scaling to the computed matrix of proximities, you can inspect the structure of the input data and look for possible clusters of observations. The `mdsProx` method returns scaled coordinates and eigenvalues for the computed proximity

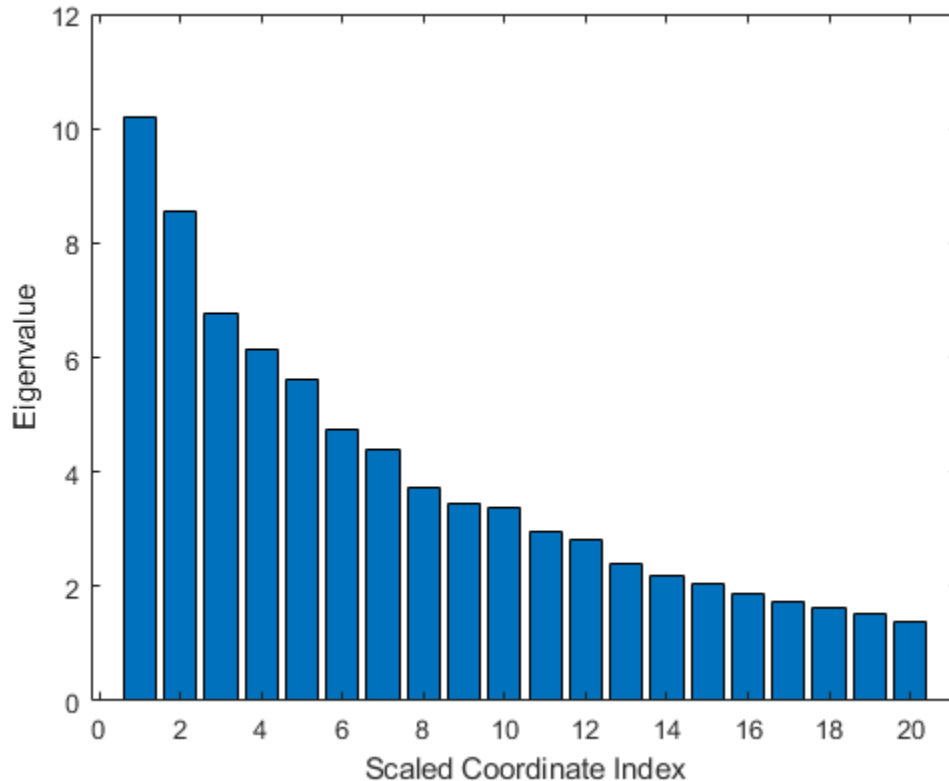
matrix. If you run it with the `Colors` name-value-pair argument, then this method creates a scatter plot of two scaled coordinates.

```
figure(8)
[~,e] = mdsProx(b5v, 'Colors', 'K');
xlabel 'First Scaled Coordinate'
ylabel 'Second Scaled Coordinate'
```



Assess the relative importance of the scaled axes by plotting the first 20 eigenvalues.

```
figure
bar(e(1:20))
xlabel 'Scaled Coordinate Index'
ylabel 'Eigenvalue'
```



### Saving the Ensemble Configuration for Future Use

To use the trained ensemble for predicting the response on unseen data, store the ensemble to disk and retrieve it later. If you do not want to compute predictions for out-of-bag data or reuse training data in any other way, there is no need to store the ensemble object itself. Saving the compact version of the ensemble is enough in this case. Extract the compact object from the ensemble.

```
c = compact(b5v)
```

```
c =
```

```
CompactTreeBagger
```



```
Ensemble with 100 bagged decision trees:  
      Method:          regression  
  NumPredictors:      4
```

You can save the resulting `CompactTreeBagger` model in a `*.mat` file.

## See Also

`TreeBagger` | `compact` | `fillprox` | `fitensemble` | `mdsprox` | `oobError`

## Related Examples

- “Bootstrap Aggregation (Bagging) of Classification Trees” on page 18-154
- `TreeBagger` Features Not in `fitensemble` or `fitrensemble`
- “Use Parallel Processing for Regression `TreeBagger` Workflow” on page 30-6

## Bootstrap Aggregation (Bagging) of Classification Trees

`TreeBagger` ensembles have more functionality than those constructed with `fitcensemble`; see `TreeBagger` Features Not in `fitcensemble` or `fitrensemble`. Also, some property and method names differ from their `fitcensemble` counterparts. This section contains an example workflow for classification that uses this extra `TreeBagger` functionality.

### Classifying Radar Returns for Ionosphere Data Using `TreeBagger`

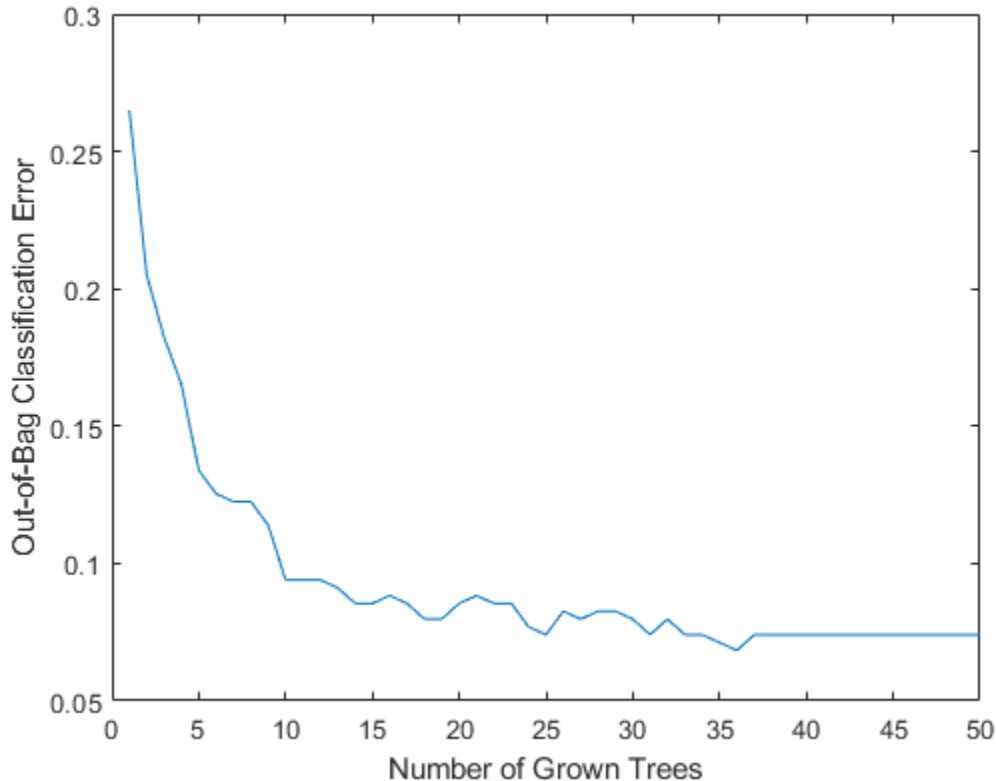
You can also use ensembles of decision trees for classification. For this example, use ionosphere data with 351 observations and 34 real-valued predictors. The response variable is categorical with two levels:

- 'g' represents good radar returns.
- 'b' represents bad radar returns.

The goal is to predict good or bad returns using a set of 34 measurements.

Fix the initial random seed, grow 50 trees, inspect how the ensemble error changes with accumulation of trees, and estimate feature importance. For classification, it is best to set the minimal leaf size to 1 and select the square root of the total number of features for each decision split at random. These settings are defaults for `TreeBagger` used for classification.

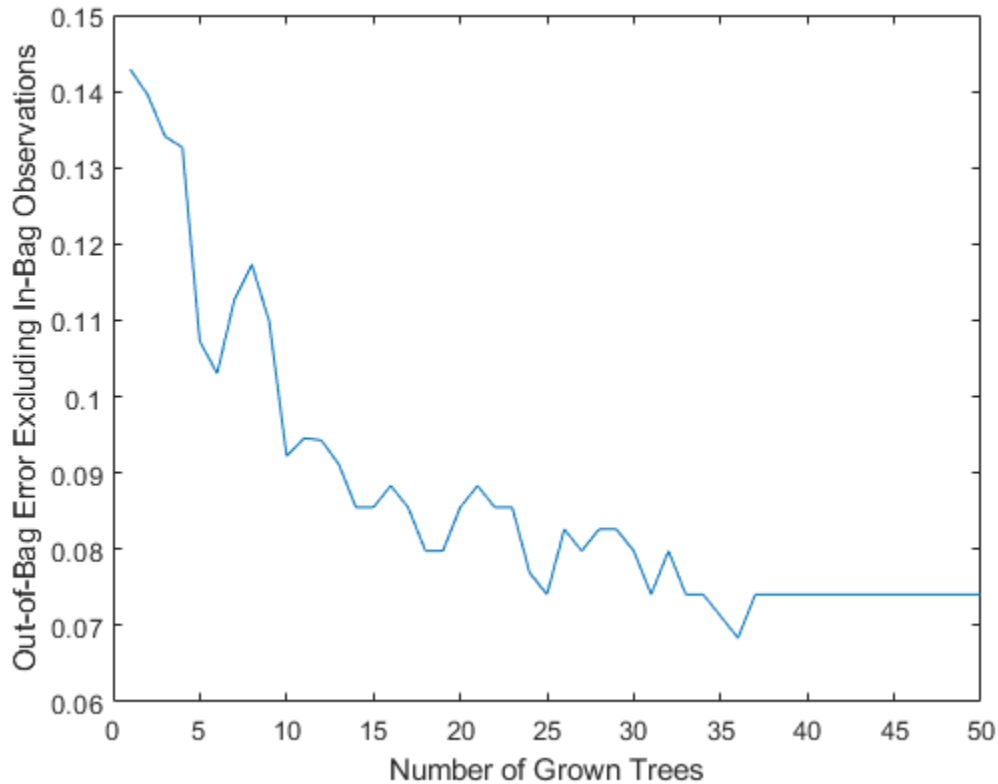
```
load ionosphere
rng(1945, 'twister')
b = TreeBagger(50, X, Y, 'OOBVarImp', 'On');
figure
plot(oobError(b))
xlabel('Number of Grown Trees')
ylabel('Out-of-Bag Classification Error')
```



The method trains ensembles with few trees on observations that are in bag for all trees. For such observations, it is impossible to compute the true out-of-bag prediction, and `TreeBagger` returns the most probable class for classification and the sample mean for regression. You can change the default value returned for in-bag observations using the `DefaultYfit` property. If you set the default value to an empty character vector for classification, the method excludes in-bag observations from computation of the out-of-bag error. In this case, the curve is more variable when the number of trees is small, either because some observations are never out of bag (and are therefore excluded) or because their predictions are based on few trees.

```
b.DefaultYfit = '';  
figure  
plot(oobError(b))
```

```
xlabel('Number of Grown Trees')
ylabel('Out-of-Bag Error Excluding In-Bag Observations')
```

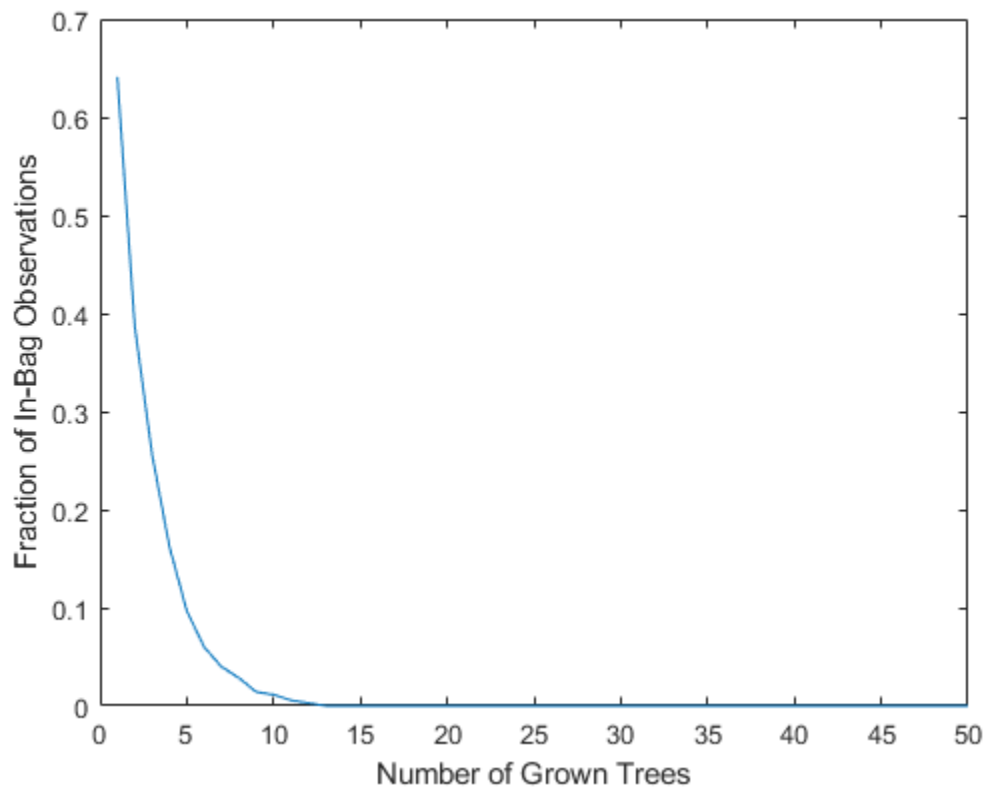


The `OOBIndices` property of `TreeBagger` tracks which observations are out of bag for what trees. Using this property, you can monitor the fraction of observations in the training data that are in bag for all trees. The curve starts at approximately  $2/3$ , which is the fraction of unique observations selected by one bootstrap replica, and goes down to 0 at approximately 10 trees.

```
finbag = zeros(1,b.NTrees);
for t=1:b.NTrees
    finbag(t) = sum(all(~b.OOBIndices(:,1:t),2));
end
finbag = finbag / size(X,1);
```

```

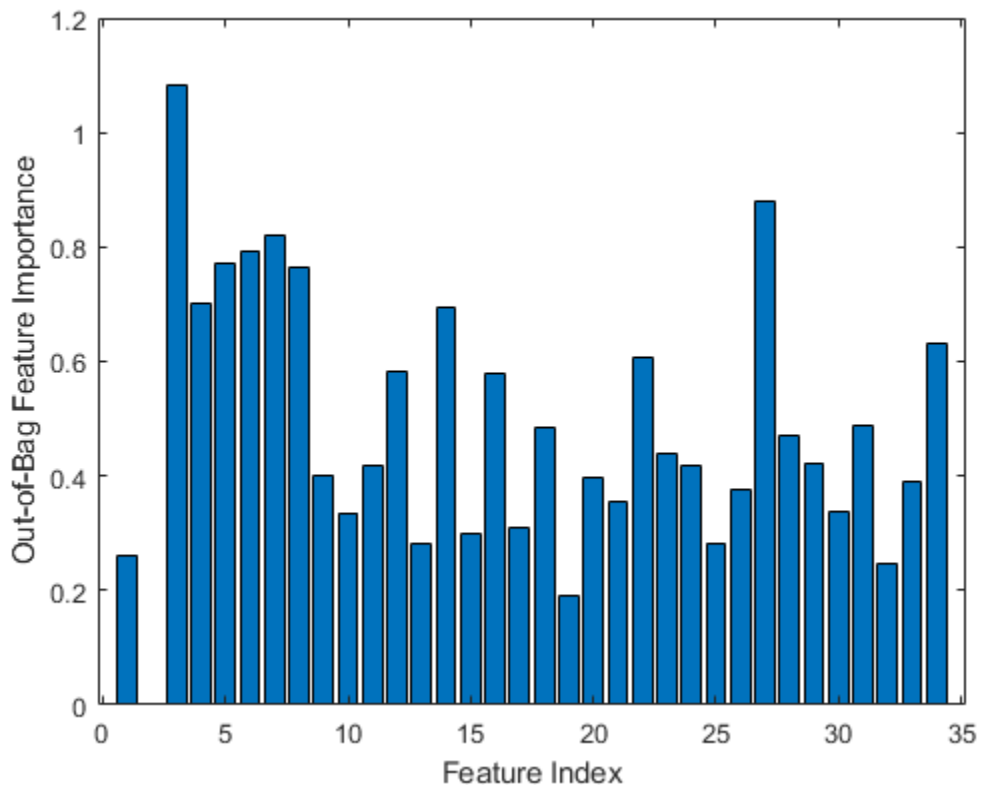
figure
plot(finbag)
xlabel('Number of Grown Trees')
ylabel('Fraction of In-Bag Observations')
    
```



Estimate feature importance.

```

figure
bar(b.OOBPermutedVarDeltaError)
xlabel('Feature Index')
ylabel('Out-of-Bag Feature Importance')
    
```



Select the features yielding an importance measure greater than 0.75. This threshold is chosen arbitrarily.

```
idxvar = find(b.OOBPermutedVarDeltaError>0.75)
```

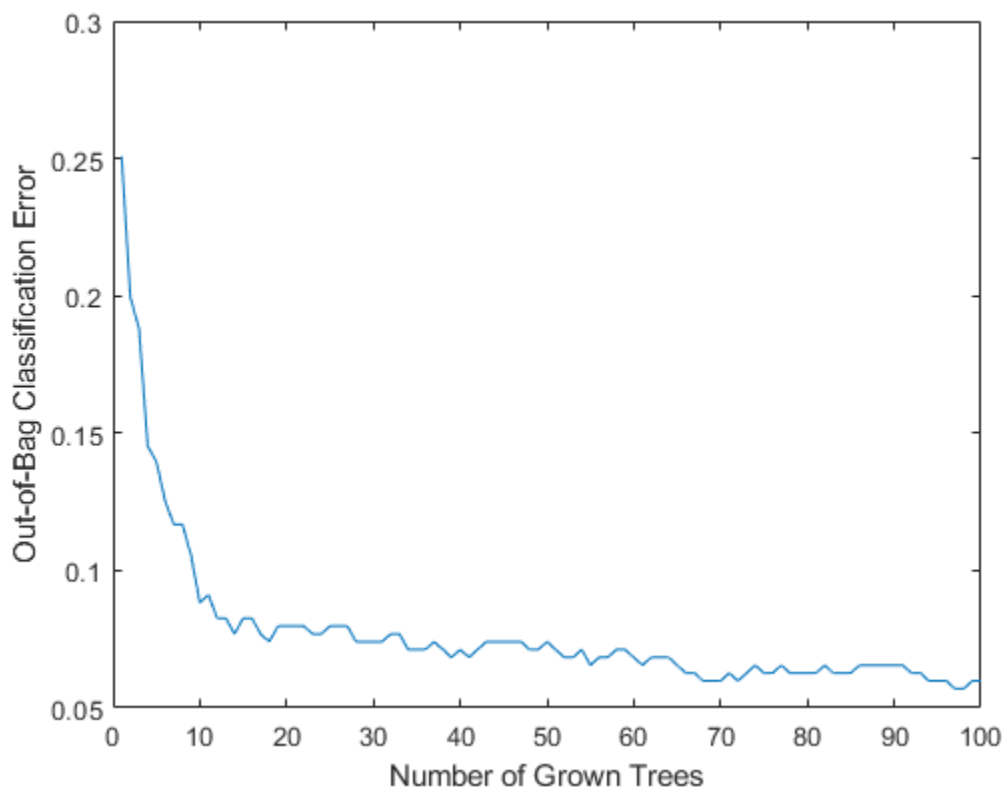
```
idxvar =
```

```
     3     5     6     7     8    27
```

Having selected the most important features, grow a larger ensemble on the reduced feature set. Save time by not permuting out-of-bag observations to obtain new estimates of feature importance for the reduced feature set (set `OOBVarImp` to `'off'`). You would

still be interested in obtaining out-of-bag estimates of classification error (set `OOBPred` to `'on'`).

```
b5v = TreeBagger(100,X(:,idxvar),Y,'OOBVarImp','off','OOBPred','on');
figure
plot(oobError(b5v))
xlabel('Number of Grown Trees')
ylabel('Out-of-Bag Classification Error')
```



For classification ensembles, in addition to classification error (fraction of misclassified observations), you can also monitor the average classification margin. For each observation, the *margin* is defined as the difference between the score for the true class and the maximal score for other classes predicted by this tree. The cumulative classification margin uses the scores averaged over all trees and the mean cumulative

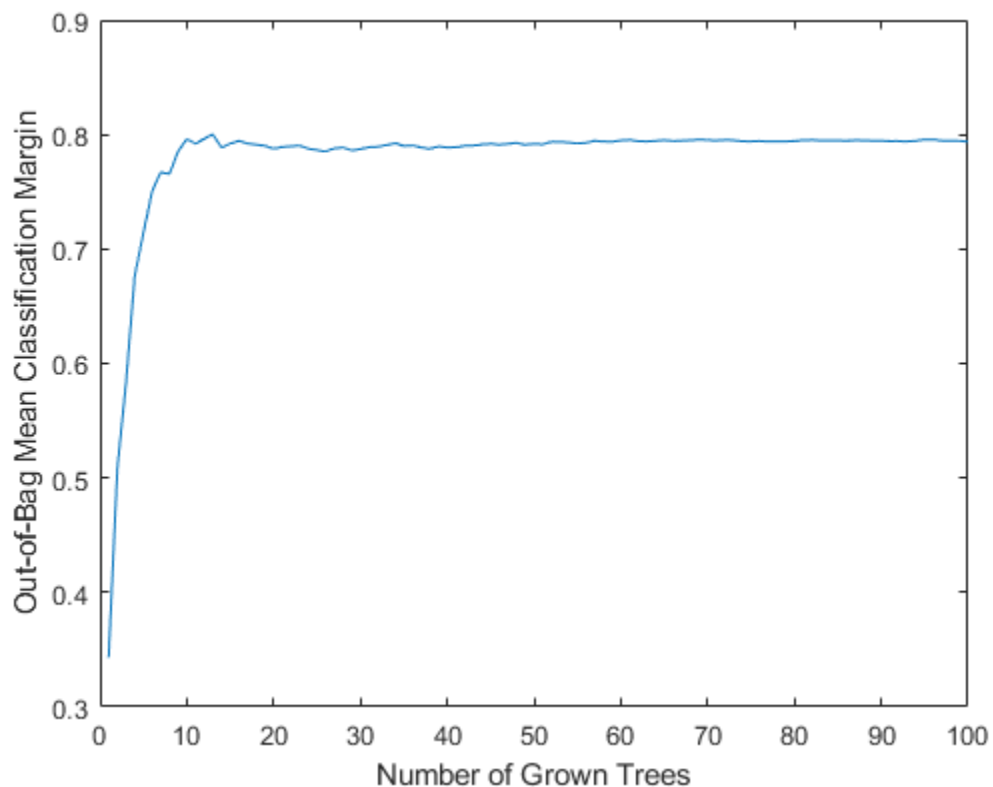
classification margin is the cumulative margin averaged over all observations. The `oobMeanMargin` method with the `'mode'` argument set to `'cumulative'` (default) shows how the mean cumulative margin changes as the ensemble grows: every new element in the returned array represents the cumulative margin obtained by including a new tree in the ensemble. If training is successful, you would expect to see a gradual increase in the mean classification margin.

The method trains ensembles with few trees on observations that are in bag for all trees. For such observations, it is impossible to compute the true out-of-bag prediction, and `TreeBagger` returns the most probable class for classification and the sample mean for regression.

For decision trees, a classification score is the probability of observing an instance of this class in this tree leaf. For example, if the leaf of a grown decision tree has five `'good'` and three `'bad'` training observations in it, the scores returned by this decision tree for any observation fallen on this leaf are  $5/8$  for the `'good'` class and  $3/8$  for the `'bad'` class. These probabilities are called `'scores'` for consistency with other classifiers that might not have an obvious interpretation for numeric values of returned predictions.

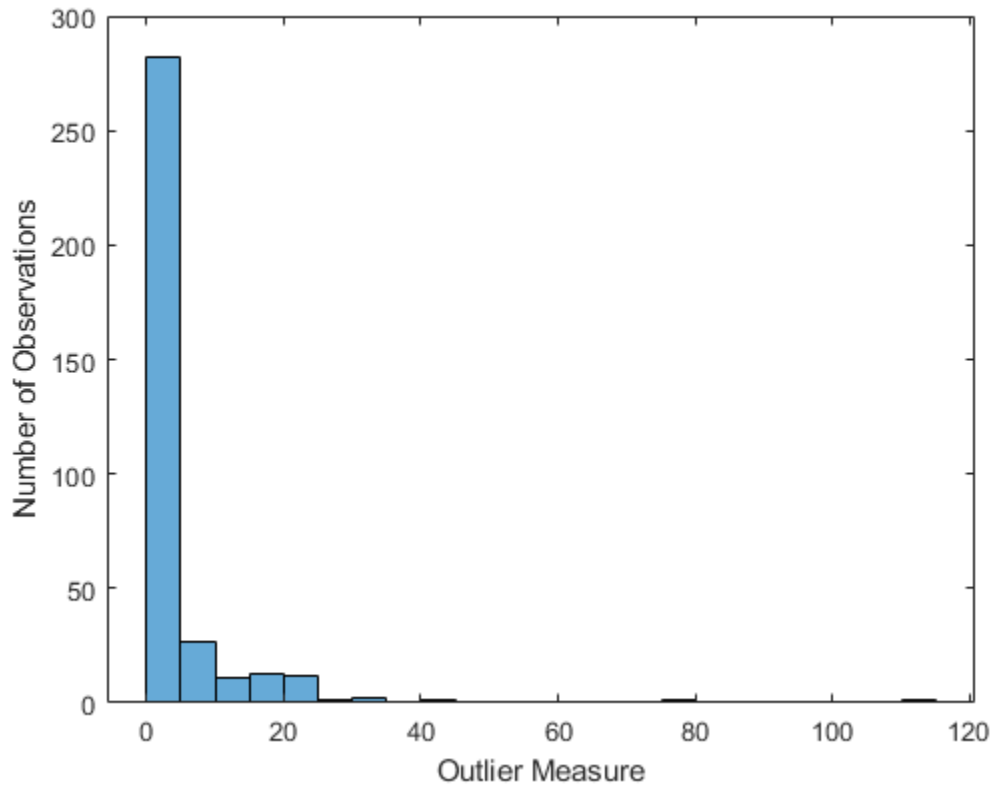
```
figure
plot(oobMeanMargin(b5v));
xlabel('Number of Grown Trees')
ylabel('Out-of-Bag Mean Classification Margin')
```





Compute the matrix of proximities and examine the distribution of outlier measures. Unlike regression, outlier measures for classification ensembles are computed within each class separately.

```
b5v = fillProximities(b5v);  
figure  
histogram(b5v.OutlierMeasure)  
xlabel('Outlier Measure')  
ylabel('Number of Observations')
```



Find the class of the extreme outliers.

```
extremeOutliers = b5v.Y(b5v.OutlierMeasure>40)
percentGood = 100*sum(strcmp(extremeOutliers,'g'))/numel(extremeOutliers)
```

```
extremeOutliers =
```

```
3x1 cell array
```

```
 {'g'}
```

```
 {'g'}
```

```
 {'g'}
```

```
percentGood =  
    100
```

All of the extreme outliers are labeled 'good'.

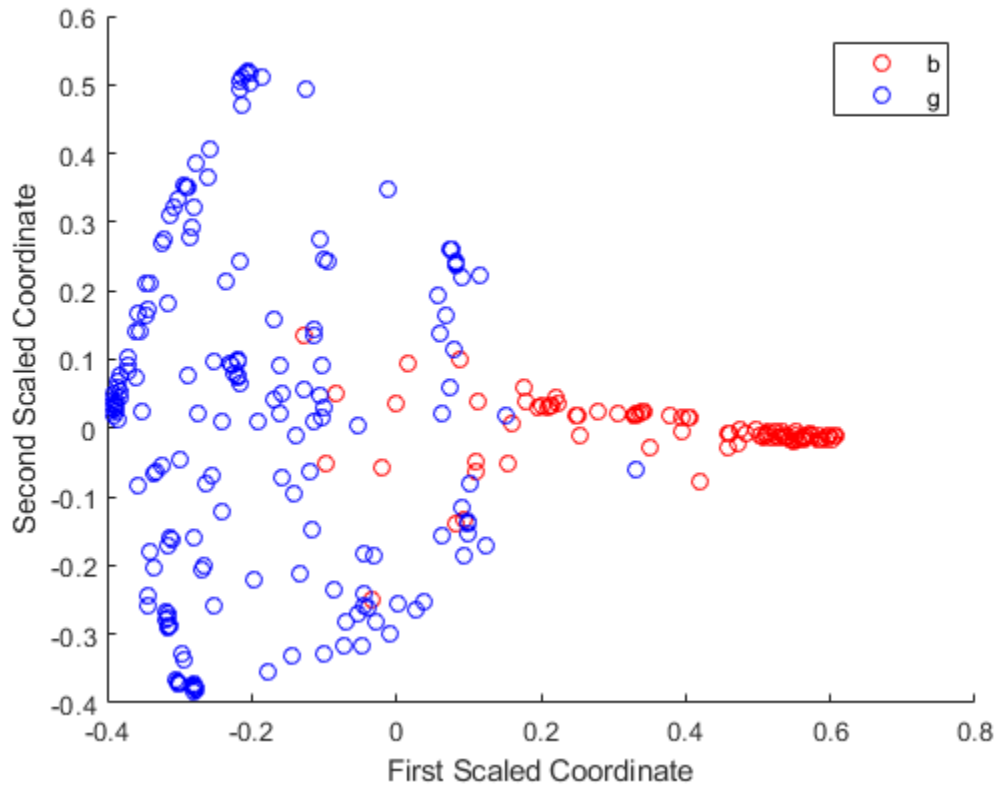
As for regression, you can plot scaled coordinates, displaying the two classes in different colors using the 'Colors' name-value pair argument of `mdsProx`. This argument takes a character vector in which every character represents a color. The software does not rank class names. Therefore, it is best practice to determine the position of the classes in the `ClassNames` property of the ensemble.

```
gPosition = find(strcmp('g',b5v.ClassNames))
```

```
gPosition =  
    2
```

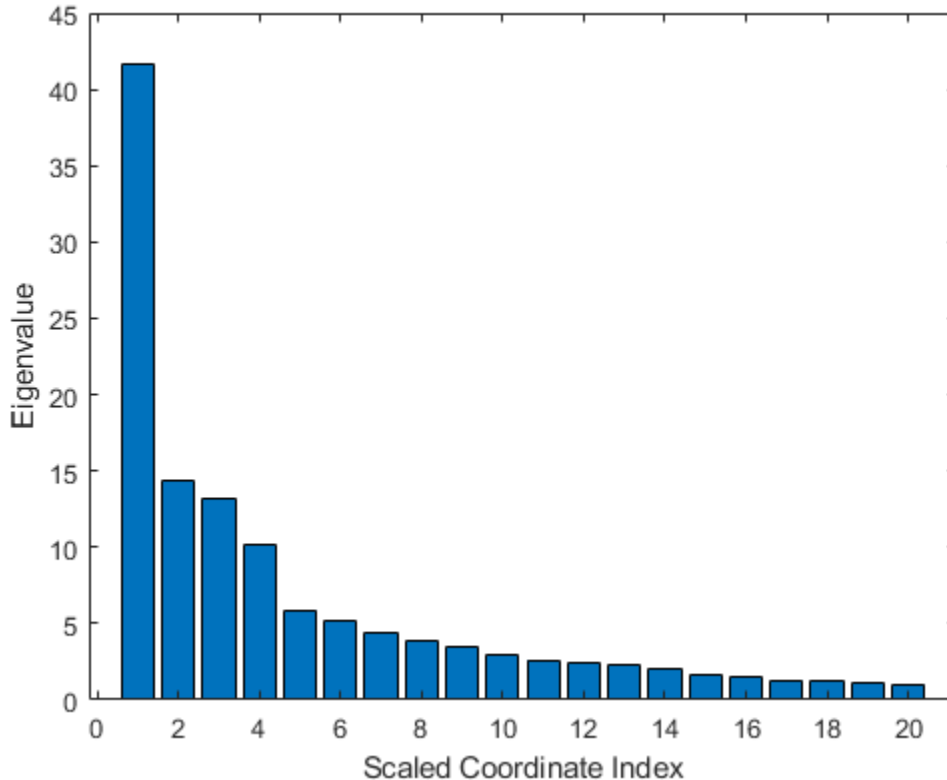
The 'bad' class is first and the 'good' class is second. Display scaled coordinates using red for the 'bad' class and blue for the 'good' class observations.

```
figure  
[s,e] = mdsProx(b5v,'Colors','rb');  
xlabel('First Scaled Coordinate')  
ylabel('Second Scaled Coordinate')
```



Plot the first 20 eigenvalues obtained by scaling. The first eigenvalue clearly dominates and the first scaled coordinate is most important.

```
figure
bar(e(1:20))
xlabel('Scaled Coordinate Index')
ylabel('Eigenvalue')
```

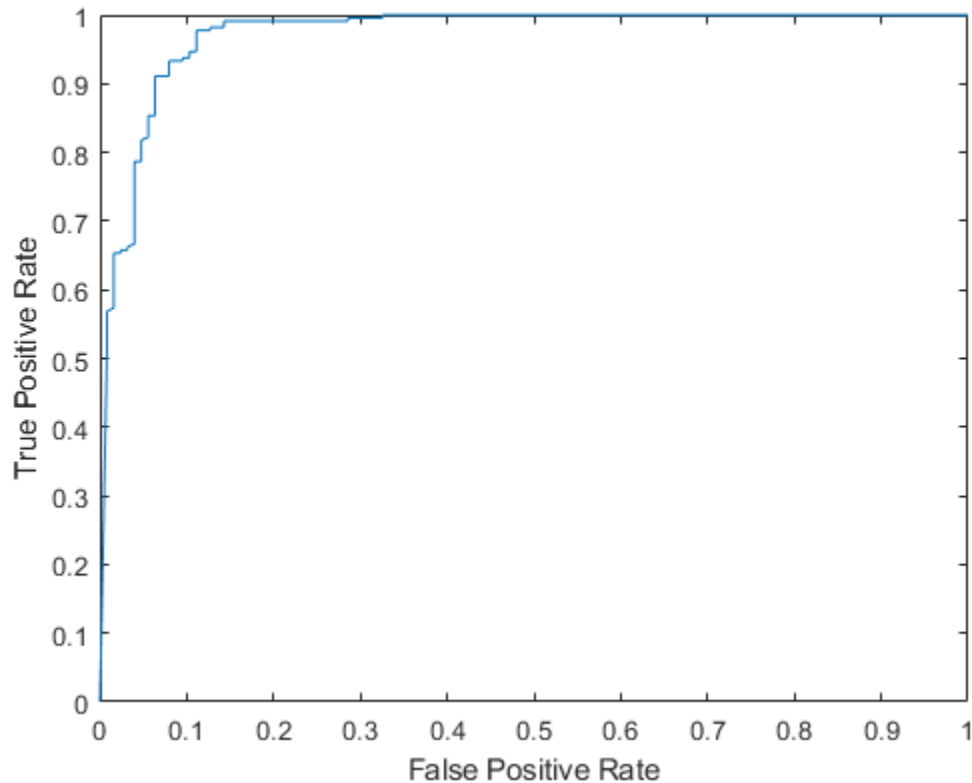


Another way of exploring the performance of a classification ensemble is to plot its Receiver Operating Characteristic (ROC) curve or another performance curve suitable for the current problem. Obtain predictions for out-of-bag observations. For a classification ensemble, the `oobPredict` method returns a cell array of classification labels as the first output argument and a numeric array of scores as the second output argument. The returned array of scores has two columns, one for each class. In this case, the first column is for the 'bad' class and the second column is for the 'good' class. One column in the score matrix is redundant because the scores represent class probabilities in tree leaves and by definition add up to 1.

```
[Yfit,Sfit] = oobPredict(b5v);
```

Use `perfcurve` to compute a performance curve. By default, `perfcurve` returns the standard ROC curve, which is the true positive rate versus the false positive rate. `perfcurve` requires true class labels, scores, and the positive class label for input. In this case, choose the 'good' class as positive.

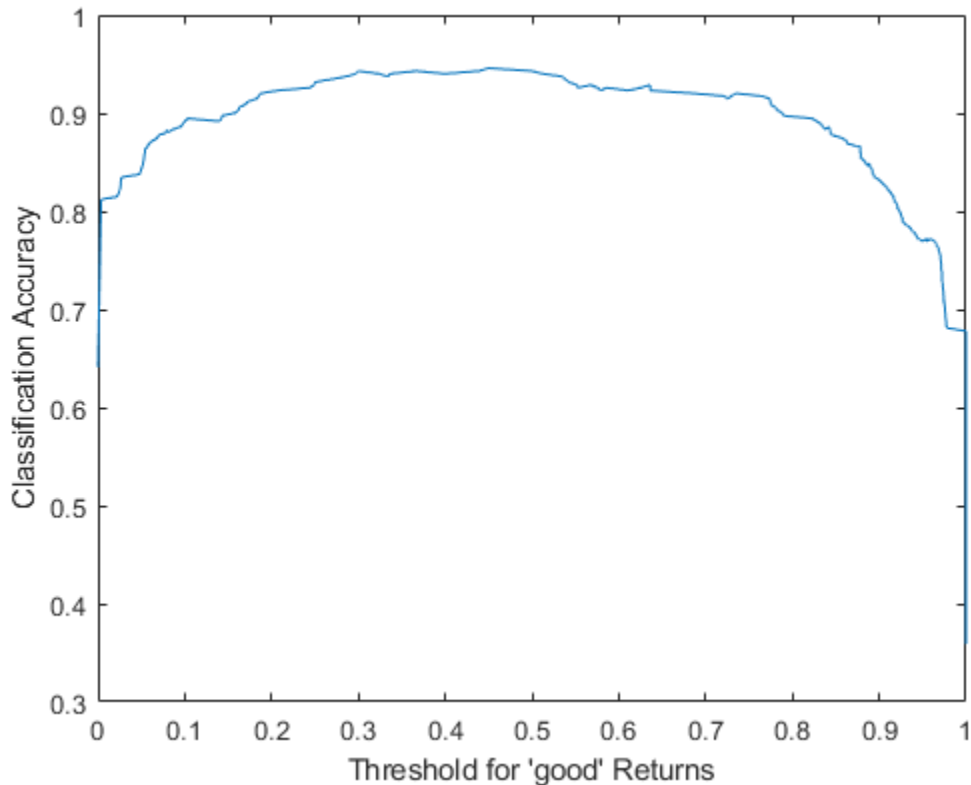
```
[fpr,tpr] = perfcurve(b5v.Y,Sfit(:,gPosition),'g');  
figure  
plot(fpr,tpr)  
xlabel('False Positive Rate')  
ylabel('True Positive Rate')
```



Instead of the standard ROC curve, you might want to plot, for example, ensemble accuracy versus threshold on the score for the 'good' class. The `ycrit` input argument

of `perfcurve` lets you specify the criterion for the y-axis, and the third output argument of `perfcurve` returns an array of thresholds for the positive class score. Accuracy is the fraction of correctly classified observations, or equivalently, 1 minus the classification error.

```
[fpr,accu,thre] = perfcurve(b5v.Y,Sfit(:,gPosition),'g','YCrit','Accu');  
figure(20)  
plot(thre,accu)  
xlabel('Threshold for 'good' Returns')  
ylabel('Classification Accuracy')
```



The curve shows a flat region indicating that any threshold from 0.2 to 0.6 is a reasonable choice. By default, the `perfcurve` assigns classification labels using 0.5 as

the boundary between the two classes. You can find exactly what accuracy this corresponds to.

```
accu(abs(thre-0.5)<eps)
```

```
ans =
```

```
0x1 empty double column vector
```

The maximal accuracy is a little higher than the default one.

```
[maxaccu,iaccu] = max(accu)
```

```
maxaccu =
```

```
0.9459
```

```
iaccu =
```

```
105
```

The optimal threshold is therefore.

```
thre(iaccu)
```

```
ans =
```

```
0.4503
```

## See Also

[TreeBagger](#) | [compact](#) | [fitcensemble](#) | [mdsprox](#) | [oobError](#) | [oobMeanMargin](#) | [oobPredict](#) | [perfcurve](#)



## Related Examples

- “Bootstrap Aggregation (Bagging) of Regression Trees” on page 18-141
- TreeBagger Features Not in fitcensemble or fitrensemble
- “Use Parallel Processing for Regression TreeBagger Workflow” on page 30-6

## Detect Outliers Using Quantile Regression

This example shows how to detect outliers using quantile random forest. Quantile random forest can detect outliers with respect to the conditional distribution of  $Y$  given  $X$ . However, this method cannot detect outliers in the predictor data. For outlier detection in the predictor data using a bag of decision trees, see the `OutlierMeasure` property of a `TreeBagger` model.

An *outlier* is an observation that is located far enough from most of the other observations in a data set and can be considered anomalous. Causes of outlying observations include inherent variability or measurement error. Outliers significant affect estimates and inference, so it is important to detect them and decide whether to remove them or consider a robust analysis.

The Statistics and Machine Learning Toolbox™ software provides several functionalities to detect or remove outliers including:

- `zscore` — Compute  $z$  scores of observations.
- `trimmean` — Estimate mean of data, excluding outliers.
- `boxplot` — Draw box plot of data.
- `probplot` — Draw probability plot.
- `robustcov` — Estimate robust covariance of multivariate data.
- `fitsvm` — Fit a one-class support vector machine (SVM) to determine which observations are located far from the decision boundary.

To demonstrate outlier detection, this example:

- 1 Generates data from a nonlinear model with heteroscedasticity and simulates a few outliers.
- 2 Grows a quantile random forest of regression trees.
- 3 Estimates conditional quartiles ( $Q_1$ ,  $Q_2$ , and  $Q_3$ ) and the interquartile range ( $IQR$ ) within the ranges of the predictor variables.
- 4 Compares the observations to the *fences*, which are the quantities  $F_1 = Q_1 - 1.5IQR$  and  $F_2 = Q_3 + 1.5IQR$ . Any observation that is less than  $F_1$  or greater than  $F_2$  is an outlier.

**Generate Data**

Generate 500 observations from the model

$$y_t = 10 + 3t + t \sin(2t) + \varepsilon_t.$$

$t$  is uniformly distributed between 0 and  $4\pi$ , and  $\varepsilon_t \sim N(0, t + 0.01)$ . Store the data in a table.

```
n = 500;
rng('default'); % For reproducibility
t = randsample(linspace(0, 4*pi, 1e6), n, true)';
epsilon = randn(n, 1) .* sqrt((t+0.01));
y = 10 + 3*t + t.*sin(2*t) + epsilon;
```

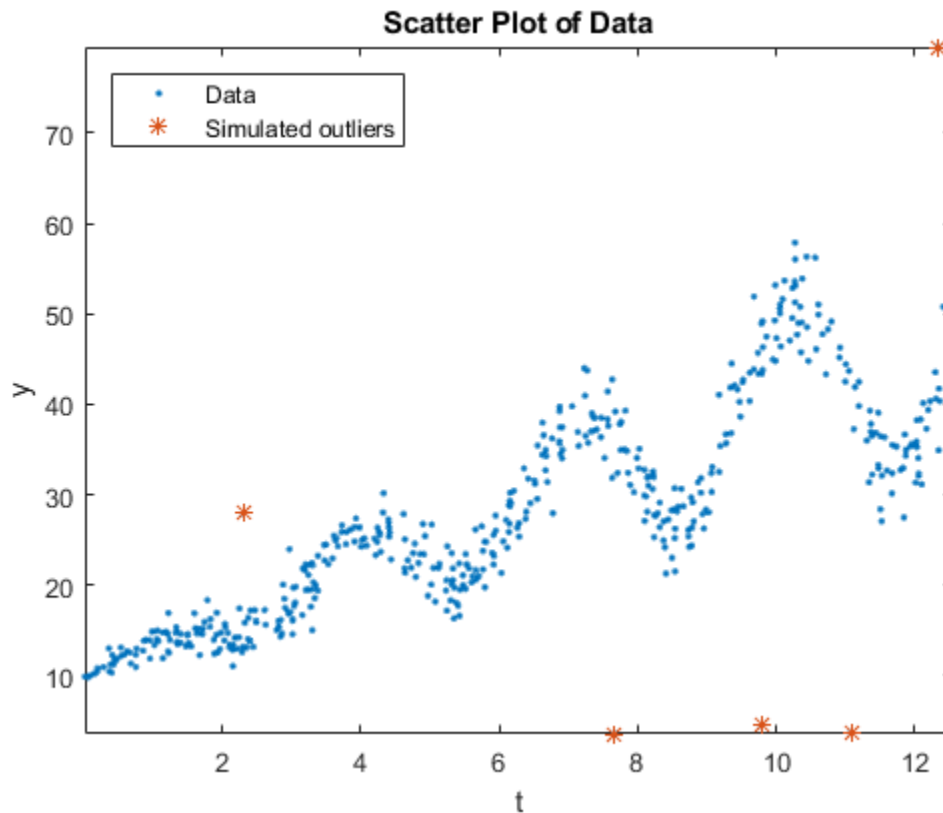
```
Tbl = table(t, y);
```

Move five observations in a random vertical direction by 90% of the value of the response.

```
numOut = 5;
[~, idx] = datasample(Tbl, numOut);
Tbl.y(idx) = Tbl.y(idx) + randsample([-1 1], numOut, true)' .* (0.9*Tbl.y(idx));
```

Draw a scatter plot of the data and identify the outliers.

```
figure;
plot(Tbl.t, Tbl.y, '.');
hold on
plot(Tbl.t(idx), Tbl.y(idx), '*');
axis tight;
ylabel('y');
xlabel('t');
title('Scatter Plot of Data');
legend('Data', 'Simulated outliers', 'Location', 'NorthWest');
```



### Grow Quantile Random Forest

Grow a bag of 200 regression trees using `TreeBagger`.

```
Mdl = TreeBagger(200, Tbl, 'y', 'Method', 'regression');
```

`Mdl` is a `TreeBagger` ensemble.

### Predict Conditional Quartiles and Interquartile Ranges

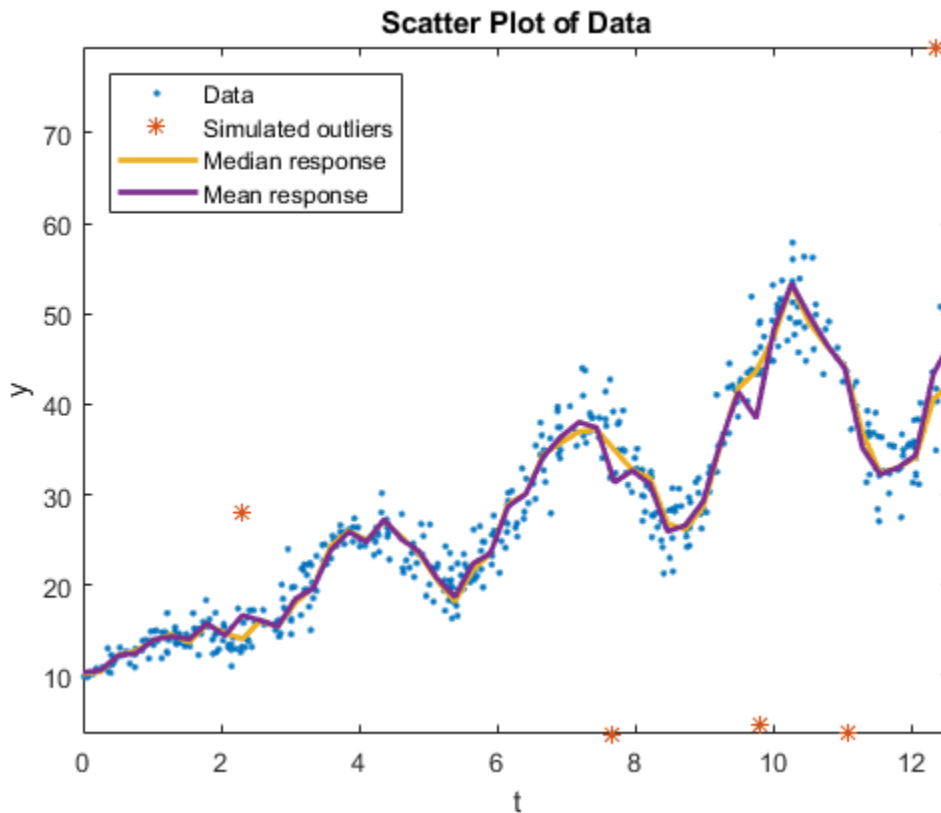
Using quantile regression, estimate the conditional quartiles of 50 equally spaced values within the range of  $t$ .

```
tau = [0.25 0.5 0.75];  
predT = linspace(0,4*pi,50)';  
quartiles = quantilePredict(Mdl,predT,'Quantile',tau);
```

quartiles is a 500-by-3 matrix of conditional quartiles. Rows correspond to the observations in `t`, and columns correspond to the probabilities in `tau`.

On the scatter plot of the data, plot the conditional mean and median responses.

```
meanY = predict(Mdl,predT);  
  
plot(predT,[quartiles(:,2) meanY],'LineWidth',2);  
legend('Data','Simulated outliers','Median response','Mean response',...  
       'Location','NorthWest');  
hold off;
```



Although the conditional mean and median curves are close, the simulated outliers can affect the mean curve.

Compute the conditional  $IQR$ ,  $F_1$ , and  $F_2$ .

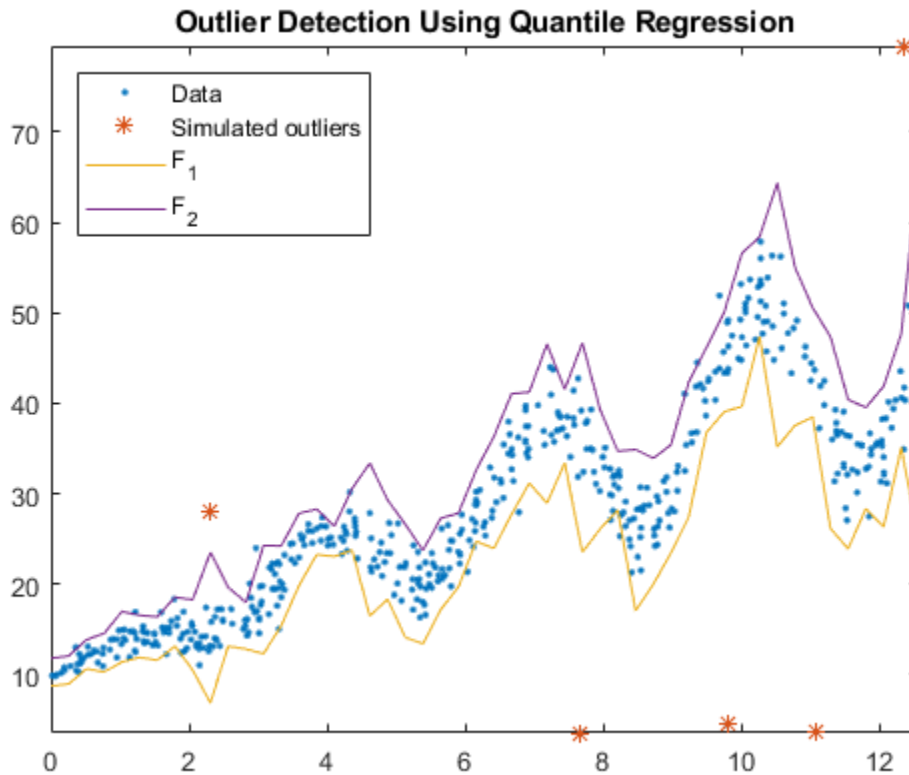
```
iqr = quartiles(:,3) - quartiles(:,1);
k = 1.5;
f1 = quartiles(:,1) - k*iqr;
f2 = quartiles(:,3) + k*iqr;
```

$k = 1.5$  means that all observations less than  $f_1$  or greater than  $f_2$  are considered outliers, but this threshold does not disambiguate from extreme outliers. A  $k$  of 3 identifies extreme outliers.

## Compare Observations to Fences

Plot the observations and the fences.

```
figure;  
plot(Tbl.t,Tbl.y, '.');  
hold on  
plot(Tbl.t(idx),Tbl.y(idx), '*');  
plot(predT,[f1 f2]);  
legend('Data','Simulated outliers','F_1','F_2','Location','NorthWest');  
axis tight  
title('Outlier Detection Using Quantile Regression')  
hold off
```



All simulated outliers fall outside  $[F_1, F_2]$ , and some observations are outside this interval as well.

## See Also

### Classes

`TreeBagger`

### Functions

`TreeBagger` | `predict` | `quantilePredict`

## Related Examples

- “Conditional Quantile Estimation Using Kernel Smoothing” on page 18-177
- “Tune Random Forest Using Quantile Error and Bayesian Optimization” on page 18-181



## Conditional Quantile Estimation Using Kernel Smoothing

This example shows how to estimate conditional quantiles of a response given predictor data using quantile random forest and by estimating the conditional distribution function of the response using kernel smoothing.

For quantile-estimation speed, `TreeBagger.quantilePredict`, `TreeBagger.oobQuantilePredict`, `TreeBagger.quantileError`, and `TreeBagger.oobQuantileError` use linear interpolation to predict quantiles in the conditional distribution of the response. However, you can obtain response weights, which comprise the distribution function, and then pass them to `ksdensity` to possibly gain accuracy at the cost of computation speed.

Generate 2000 observations from the model

$$y_t = 0.5 + t + \varepsilon_t.$$

$t$  is uniformly distributed between 0 and 1, and  $\varepsilon_t \sim N(0, t^2/2 + 0.01)$ . Store the data in a table.

```
n = 2000;
rng('default'); % For reproducibility
t = randsample(linspace(0,1,1e2),n,true)';
epsilon = randn(n,1).*sqrt(t.^2/2 + 0.01);
y = 0.5 + t + epsilon;
```

```
Tbl = table(t,y);
```

Train an ensemble of bagged regression trees using the entire data set. Specify 200 weak learners and save the out-of-bag indices.

```
rng('default'); % For reproducibility
Mdl = TreeBagger(200,Tbl,'y','Method','regression',...
    'OOBPrediction','on');
```

`Mdl` is a `TreeBagger` ensemble.

Predict out-of-bag, conditional 0.05 and 0.95 quantiles (90% confidence intervals) for all training-sample observations using `oobQuantilePredict`, that is, by interpolation. Request response weights. Record the execution time.

```
tau = [0.05 0.95];
tic
```

```
[quantInterp,yw] = oobQuantilePredict(Mdl,'Quantile',tau);  
timeInterp = toc;
```

quantInterp is a 94-by-2 matrix of predicted quantiles; rows correspond to the observations in Mdl.X and columns correspond to the quantile probabilities in tau. yw is a 94-by-94 sparse matrix of response weights; rows correspond to training-sample observations and columns correspond to the observations in Mdl.X. Response weights are independent of tau.

Predict out-of-bag, conditional 0.05 and 0.95 quantiles using kernel smoothing and record the execution time.

```
n = numel(Tbl.y);  
quantKS = zeros(n,numel(tau)); % Preallocation  
  
tic  
for j = 1:n  
    quantKS(j,:) = ksdensity(Tbl.y,tau,'Function','icdf','Weights',yw(:,j));  
end  
timeKS = toc;
```

quantKS is commensurate with quantInterp.

Evaluate the ratio of execution times between kernel smoothing estimation and interpolation.

```
timeKS/timeInterp
```

```
ans =  
  
    6.0287
```

It takes much more time to execute kernel smoothing than interpolation. This ratio is dependent on the memory of your machine, so your results will vary.

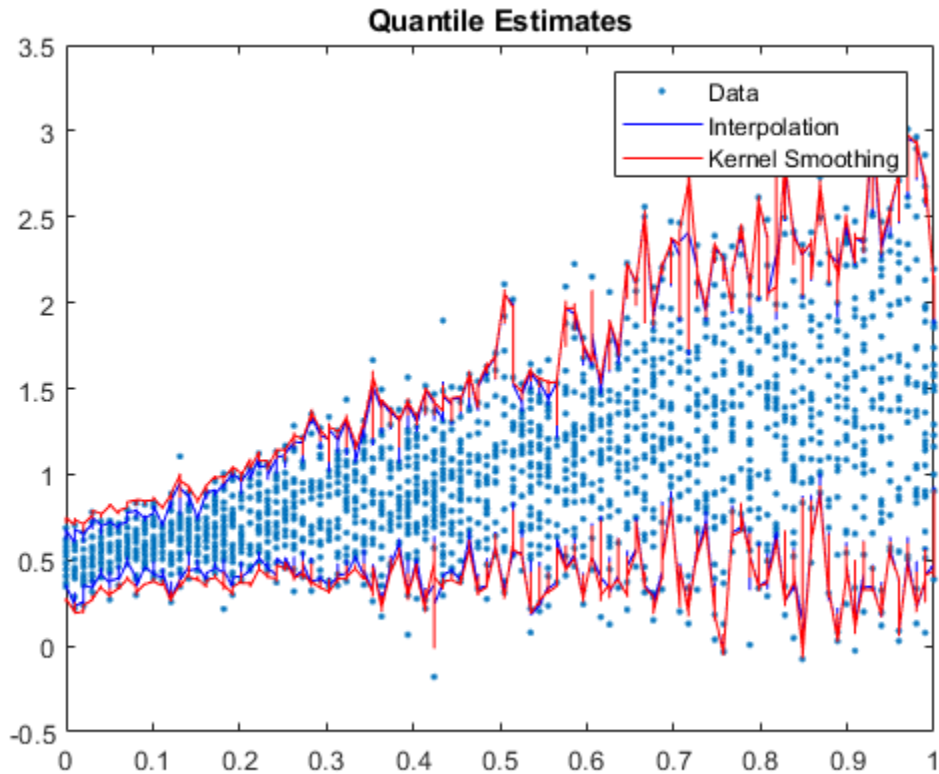
Plot the data with both sets of predicted quantiles.

```
[sT,idx] = sort(t);  
  
figure;  
h1 = plot(t,y,'.');  
hold on
```

```

h2 = plot(sT,quantInterp(idx,:), 'b');
h3 = plot(sT,quantKS(idx,:), 'r');
legend([h1 h2(1) h3(1)], 'Data', 'Interpolation', 'Kernel Smoothing');
title('Quantile Estimates')
hold off

```



Both sets of estimated quantiles agree fairly well. However, the quantile intervals from interpolation appear slightly tighter for smaller values of  $\tau$  than the ones from kernel smoothing.

## See Also

TreeBagger | TreeBagger | ksdensity | oobQuantilePredict

## **Related Examples**

- “Detect Outliers Using Quantile Regression” on page 18-170
- “Tune Random Forest Using Quantile Error and Bayesian Optimization” on page 18-181

## Tune Random Forest Using Quantile Error and Bayesian Optimization

This example shows how to implement Bayesian optimization to tune the hyperparameters of a random forest of regression trees using quantile error. Tuning a model using quantile error, rather than mean squared error, is appropriate if you plan to use the model to predict conditional quantiles rather than conditional means.

### Load and Preprocess Data

Load the `carsmall` data set. Consider a model that predicts the median fuel economy of a car given its acceleration, number of cylinders, engine displacement, horsepower, manufacturer, model year, and weight. Consider `Cylinders`, `Mfg`, and `Model_Year` as categorical variables.

```
load carsmall
Cylinders = categorical(Cylinders);
Mfg = categorical(cellstr(Mfg));
Model_Year = categorical(Model_Year);
X = table(Acceleration,Cylinders,Displacement,Horsepower,Mfg,...
    Model_Year,Weight,MPG);
rng('default'); % For reproducibility
```

### Specify Tuning Parameters

Consider tuning:

- The complexity (depth) of the trees in the forest. Deep trees tend to over-fit, but shallow trees tend to underfit. Therefore, specify that the minimum number of observations per leaf be at most 20.
- When growing the trees, the number of predictors to sample at each node. Specify sampling from 1 through all of the predictors.

`bayesopt`, the function that implements Bayesian optimization, requires you to pass these specifications as `optimizableVariable` objects.

```
maxMinLS = 20;
minLS = optimizableVariable('minLS',[1,maxMinLS],'Type','integer');
numPTS = optimizableVariable('numPTS',[1,size(X,2)-1],'Type','integer');
hyperparametersRF = [minLS; numPTS];
```

`hyperparametersRF` is a 2-by-1 array of `OptimizableVariable` objects.

You should also consider tuning the number of trees in the ensemble. `bayesopt` tends to choose random forests containing many trees because ensembles with more learners are more accurate. If available computation resources is a consideration, and you prefer ensembles with as few trees, then consider tuning the number of trees separately from the other parameters or penalizing models containing many learners.

### Declare Objective Function

Declare an objective function for the Bayesian optimization algorithm to optimize. The function should:

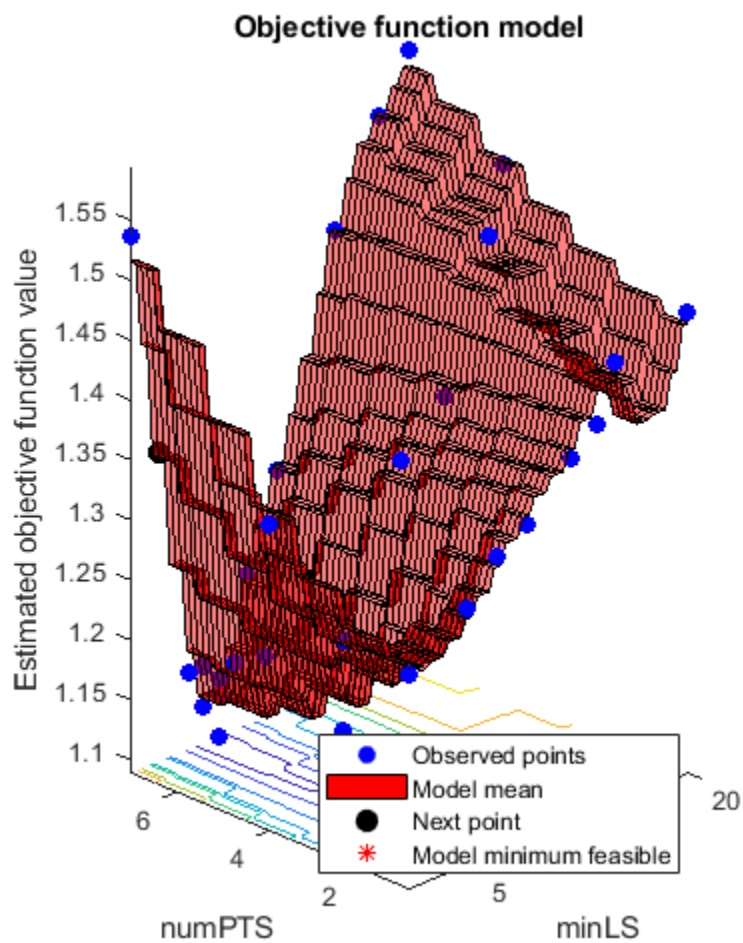
- Accept the parameters to tune as an input.
- Train a random forest using `TreeBagger`. In the `TreeBagger` call, specify the parameters to tune and specify returning the out-of-bag indices.
- Estimate the out-of-bag quantile error based on the median.
- Return the out-of-bag quantile error.

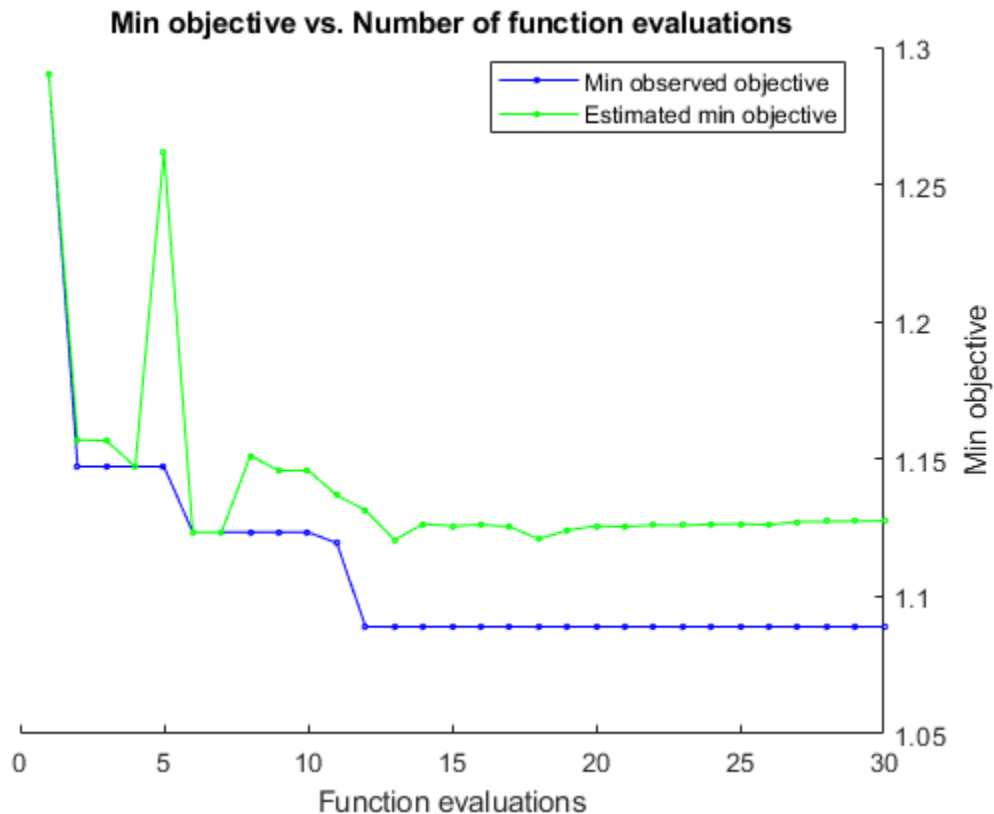
```
function oobErr = oobErrRF(params,X)
%oobErrRF Trains random forest and estimates out-of-bag quantile error
% oobErr trains a random forest of 300 regression trees using the
% predictor data in X and the parameter specification in params, and then
% returns the out-of-bag quantile error based on the median. X is a table
% and params is an array of OptimizableVariable objects corresponding to
% the minimum leaf size and number of predictors to sample at each node.
randomForest = TreeBagger(300,X,'MPG','Method','regression',...
    'OOBPrediction','on','MinLeafSize',params.minLS,...
    'NumPredictorstoSample',params.numPTS);
oobErr = oobQuantileError(randomForest);
end
```

### Minimize Objective Using Bayesian Optimization

Find the model achieving the minimal, penalized, out-of-bag quantile error with respect to tree complexity and number of predictors to sample at each node using Bayesian optimization. Specify the expected improvement plus function as the acquisition function and suppress printing the optimization information.

```
results = bayesopt(@(params)oobErrRF(params,X),hyperparametersRF,...
    'AcquisitionFunctionName','expected-improvement-plus','Verbose',0);
```





results is a BayesianOptimization object containing, among other things, the minimum of the objective function and the optimized hyperparameter values.

Display the observed minimum of the objective function and the optimized hyperparameter values.

```
bestOOBErr = results.MinObjective
bestHyperparameters = results.XAtMinObjective
```

```
bestOOBErr =
```

```
1.0890
```



```
bestHyperparameters =  
  
1x2 table  
  
   minLS   numPTS  
-----  
7         7
```

### Train Model Using Optimized Hyperparameters

Train a random forest using the entire data set and the optimized hyperparameter values.

```
Mdl = TreeBagger(300,X,'MPG','Method','regression',...  
    'MinLeafSize',bestHyperparameters.minLS,...  
    'NumPredictorstoSample',bestHyperparameters.numPTS);
```

Mdl is TreeBagger object optimized for median prediction. You can predict the median fuel economy given predictor data by passing Mdl and the new data to `quantilePredict`.

## See Also

`TreeBagger` | `TreeBagger` | `bayesopt` | `oobQuantileError` | `optimizableVariable`

### Related Examples

- “Detect Outliers Using Quantile Regression” on page 18-170
- “Conditional Quantile Estimation Using Kernel Smoothing” on page 18-177

## Support Vector Machines for Binary Classification

### In this section...

- “Understanding Support Vector Machines” on page 18-186
- “Using Support Vector Machines” on page 18-192
- “Train SVM Classifiers Using a Gaussian Kernel” on page 18-195
- “Train SVM Classifier Using Custom Kernel” on page 18-199
- “Optimize a Cross-Validated SVM Classifier Using Bayesian Optimization” on page 18-205
- “Plot Posterior Probability Regions for SVM Classification Models” on page 18-218
- “Analyze Images Using Linear Support Vector Machines” on page 18-221

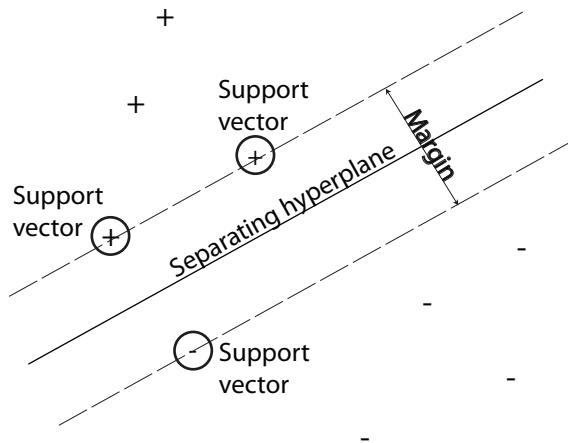
### Understanding Support Vector Machines

- “Separable Data” on page 18-186
- “Nonseparable Data” on page 18-189
- “Nonlinear Transformation with Kernels” on page 18-191

#### Separable Data

You can use a support vector machine (SVM) when your data has exactly two classes. An SVM classifies data by finding the best hyperplane that separates all data points of one class from those of the other class. The best hyperplane for an SVM means the one with the largest margin between the two classes. Margin means the maximal width of the slab parallel to the hyperplane that has no interior data points.

The support vectors are the data points that are closest to the separating hyperplane; these points are on the boundary of the slab. The following figure illustrates these definitions, with + indicating data points of type 1, and – indicating data points of type – 1.



### Mathematical Formulation: Primal

This discussion follows Hastie, Tibshirani, and Friedman [1] and Christianini and Shawe-Taylor [2].

The data for training is a set of points (vectors)  $x_j$  along with their categories  $y_j$ . For some dimension  $d$ , the  $x_j \in R^d$ , and the  $y_j = \pm 1$ . The equation of a hyperplane is

$$f(x) = x'\beta + b = 0$$

where  $\beta \in R^d$  and  $b$  is a real number.

The following problem defines the best separating hyperplane (i.e., the decision boundary). Find  $\beta$  and  $b$  that minimize  $\|\beta\|$  such that for all data points  $(x_j, y_j)$ ,

$$y_j f(x_j) \geq 1.$$

The support vectors are the  $x_j$  on the boundary, those for which  $y_j f(x_j) = 1$ .

For mathematical convenience, the problem is usually given as the equivalent problem of

minimizing  $\|\beta\|$ . This is a quadratic programming problem. The optimal solution  $(\hat{\beta}, \hat{b})$  enables classification of a vector  $z$  as follows:

$$\text{class}(z) = \text{sign}(z'\hat{\beta} + \hat{b}) = \text{sign}(f(z)).$$

$\hat{f}(z)$  is the classification score and represents the distance  $z$  is from the decision boundary.

**Mathematical Formulation: Dual**

It is computationally simpler to solve the dual quadratic programming problem. To obtain the dual, take positive Lagrange multipliers  $\alpha_j$  multiplied by each constraint, and subtract from the objective function:

$$L_P = \frac{1}{2} \beta' \beta - \sum_j \alpha_j (y_j (x_j' \beta + b) - 1),$$

where you look for a stationary point of  $L_P$  over  $\beta$  and  $b$ . Setting the gradient of  $L_P$  to 0, you get

$$\begin{aligned} \beta &= \sum_j \alpha_j y_j x_j \\ 0 &= \sum_j \alpha_j y_j. \end{aligned}$$

Substituting into  $L_P$ , you get the dual  $L_D$ :

$$L_D = \sum_j \alpha_j - \frac{1}{2} \sum_j \sum_k \alpha_j \alpha_k y_j y_k x_j' x_k,$$

which you maximize over  $\alpha_j \geq 0$ . In general, many  $\alpha_j$  are 0 at the maximum. The nonzero  $\alpha_j$  in the solution to the dual problem define the hyperplane, as seen in “Equation 18-1” on page 18-188, which gives  $\beta$  as the sum of  $\alpha_j y_j x_j$ . The data points  $x_j$  corresponding to nonzero  $\alpha_j$  are the support vectors.

The derivative of  $L_D$  with respect to a nonzero  $\alpha_j$  is 0 at an optimum. This gives

$$y_j f(x_j) - 1 = 0.$$

In particular, this gives the value of  $b$  at the solution, by taking any  $j$  with nonzero  $\alpha_j$ .

The dual is a standard quadratic programming problem. For example, the Optimization Toolbox `quadprog` solver solves this type of problem.

## Nonseparable Data

Your data might not allow for a separating hyperplane. In that case, SVM can use a soft margin, meaning a hyperplane that separates many, but not all data points.

There are two standard formulations of soft margins. Both involve adding slack variables  $\xi_j$  and a penalty parameter  $C$ .

- The  $L^1$ -norm problem is:

$$\min_{\beta, b, \xi} \left( \frac{1}{2} \beta' \beta + C \sum_j \xi_j \right)$$

such that

$$\begin{aligned} y_j f(x_j) &\geq 1 - \xi_j \\ \xi_j &\geq 0. \end{aligned}$$

The  $L^1$ -norm refers to using  $\xi_j$  as slack variables instead of their squares. The three solver options SMO, ISDA, and L1QP of `fitcsvm` minimize the  $L^1$ -norm problem.

- The  $L^2$ -norm problem is:

$$\min_{\beta, b, \xi} \left( \frac{1}{2} \beta' \beta + C \sum_j \xi_j^2 \right)$$

subject to the same constraints.

In these formulations, you can see that increasing  $C$  places more weight on the slack variables  $\xi_j$ , meaning the optimization attempts to make a stricter separation between classes. Equivalently, reducing  $C$  towards 0 makes misclassification less important.

### Mathematical Formulation: Dual

For easier calculations, consider the  $L^1$  dual problem to this soft-margin formulation. Using Lagrange multipliers  $\mu_j$ , the function to minimize for the  $L^1$ -norm problem is:

$$L_P = \frac{1}{2} \beta' \beta + C \sum_j \xi_j - \sum_j \alpha_j (y_j f(x_j) - (1 - \xi_j)) - \sum_j \mu_j \xi_j,$$

where you look for a stationary point of  $L_P$  over  $\beta$ ,  $b$ , and positive  $\xi_j$ . Setting the gradient of  $L_P$  to 0, you get

$$\begin{aligned}\beta &= \sum_j \alpha_j y_j x_j \\ \sum_j \alpha_j y_j &= 0 \\ \alpha_j &= C - \mu_j \\ \alpha_j, \mu_j, \xi_j &\geq 0.\end{aligned}$$

These equations lead directly to the dual formulation:

$$\max_{\alpha} \sum_j \alpha_j - \frac{1}{2} \sum_j \sum_k \alpha_j \alpha_k y_j y_k x_j' x_k$$

subject to the constraints

$$\begin{aligned}\sum_j y_j \alpha_j &= 0 \\ 0 &\leq \alpha_j \leq C.\end{aligned}$$

The final set of inequalities,  $0 \leq \alpha_j \leq C$ , shows why  $C$  is sometimes called a box constraint.  $C$  keeps the allowable values of the Lagrange multipliers  $\alpha_j$  in a “box”, a bounded region.

The gradient equation for  $b$  gives the solution  $b$  in terms of the set of nonzero  $\alpha_j$ , which correspond to the support vectors.

You can write and solve the dual of the  $L^2$ -norm problem in an analogous manner. For details, see Christianini and Shawe-Taylor [2], Chapter 6.

### **fitsvm Implementation**

Both dual soft-margin problems are quadratic programming problems. Internally, `fitsvm` has several different algorithms for solving the problems.

- For one-class or binary classification, if you do not set a fraction of expected outliers in the data (see `OutlierFraction`), then the default solver is Sequential Minimal Optimization (SMO). SMO minimizes the one-norm problem by a series of two-point

minimizations. During optimization, SMO respects the linear constraint  $\sum \alpha_i y_i = 0$ , and explicitly includes the bias term in the model. SMO is relatively fast. For more details on SMO, see [3].

- For binary classification, if you set a fraction of expected outliers in the data, then the default solver is the Iterative Single Data Algorithm. Like SMO, ISDA solves the one-norm problem. Unlike SMO, ISDA minimizes by a series on one-point minimizations, does not respect the linear constraint, and does not explicitly include the bias term in the model. For more details on ISDA, see [4].
- For one-class or binary classification, and if you have an Optimization Toolbox license, you can choose to use `quadprog` to solve the one-norm problem. `quadprog` uses a good deal of memory, but solves quadratic programs to a high degree of precision. For more details, see “Quadratic Programming Definition” (Optimization Toolbox).

### Nonlinear Transformation with Kernels

Some binary classification problems do not have a simple hyperplane as a useful separating criterion. For those problems, there is a variant of the mathematical approach that retains nearly all the simplicity of an SVM separating hyperplane.

This approach uses these results from the theory of reproducing kernels:

- There is a class of functions  $G(x_1, x_2)$  with the following property. There is a linear space  $S$  and a function  $\varphi$  mapping  $x$  to  $S$  such that

$$G(x_1, x_2) = \langle \varphi(x_1), \varphi(x_2) \rangle.$$

The dot product takes place in the space  $S$ .

- This class of functions includes:

- Polynomials: For some positive integer  $p$ ,

$$G(x_1, x_2) = (1 + x_1'x_2)^p.$$

- Radial basis function (Gaussian):

$$G(x_1, x_2) = \exp(-\|x_1 - x_2\|^2).$$

- Multilayer perceptron or sigmoid (neural network): For a positive number  $p_1$  and a negative number  $p_2$ ,

$$G(x_1, x_2) = \tanh(p_1 x_1'x_2 + p_2).$$

---

#### Note

- Not every set of  $p_1$  and  $p_2$  yields a valid reproducing kernel.
  - `fitcsvm` does not support the sigmoid kernel.
- 

The mathematical approach using kernels relies on the computational method of hyperplanes. All the calculations for hyperplane classification use nothing more than dot products. Therefore, nonlinear kernels can use identical calculations and solution algorithms, and obtain classifiers that are nonlinear. The resulting classifiers are hypersurfaces in some space  $S$ , but the space  $S$  does not have to be identified or examined.

## Using Support Vector Machines

As with any supervised learning model, you first train a support vector machine, and then cross validate the classifier. Use the trained machine to classify (predict) new data. In addition, to obtain satisfactory predictive accuracy, you can use various SVM kernel functions, and you must tune the parameters of the kernel functions.

- “Training an SVM Classifier” on page 18-192
- “Classifying New Data with an SVM Classifier” on page 18-193
- “Tuning an SVM Classifier” on page 18-194

### Training an SVM Classifier

Train, and optionally cross validate, an SVM classifier using `fitcsvm`. The most common syntax is:

```
SVMModel = fitcsvm(X,Y,'KernelFunction','rbf',...  
    'Standardize',true,'ClassNames',{'negClass','posClass'});
```

The inputs are:

- $X$  — Matrix of predictor data, where each row is one observation, and each column is one predictor.
- $Y$  — Array of class labels with each row corresponding to the value of the corresponding row in  $X$ .  $Y$  can be a character array, categorical, logical or numeric vector, or cell vector of character vectors. Column vector with each row corresponding to the value of the corresponding row in  $X$ .  $Y$  can be a categorical or character array, logical or numeric vector, or cell array of character vectors.



- `KernelFunction` — The default value is `'linear'` for two-class learning, which separates the data by a hyperplane. The value `'gaussian'` (or `'rbf'`) is the default for one-class learning, and specifies to use the Gaussian (or radial basis function) kernel. An important step to successfully train an SVM classifier is to choose an appropriate kernel function.
- `Standardize` — Flag indicating whether the software should standardize the predictors before training the classifier.
- `ClassNames` — Distinguishes between the negative and positive classes, or specifies which classes to include in the data. The negative class is the first element (or row of a character array), e.g., `'negClass'`, and the positive class is the second element (or row of a character array), e.g., `'posClass'`. `ClassNames` must be the same data type as `Y`. It is good practice to specify the class names, especially if you are comparing the performance of different classifiers.

The resulting, trained model (`SVMMModel`) contains the optimized parameters from the SVM algorithm, enabling you to classify new data.

For more name-value pairs you can use to control the training, see the `fitcsvm` reference page.

### Classifying New Data with an SVM Classifier

Classify new data using `predict`. The syntax for classifying new data using a trained SVM classifier (`SVMMModel`) is:

```
[label,score] = predict(SVMMModel,newX);
```

The resulting vector, `label`, represents the classification of each row in `X`. `score` is an  $n$ -by-2 matrix of soft scores. Each row corresponds to a row in `X`, which is a new observation. The first column contains the scores for the observations being classified in the negative class, and the second column contains the scores observations being classified in the positive class.

To estimate posterior probabilities rather than scores, first pass the trained SVM classifier (`SVMMModel`) to `fitPosterior`, which fits a score-to-posterior-probability transformation function to the scores. The syntax is:

```
ScoreSVMMModel = fitPosterior(SVMMModel,X,Y);
```

The property `ScoreTransform` of the classifier `ScoreSVMMModel` contains the optimal transformation function. Pass `ScoreSVMMModel` to `predict`. Rather than returning the

scores, the output argument `score` contains the posterior probabilities of an observation being classified in the negative (column 1 of `score`) or positive (column 2 of `score`) class.

### Tuning an SVM Classifier

Try tuning parameters of your classifier according to this scheme:

- 1 Pass the data to `fitcsvm`, and set the name-value pair arguments `'KernelScale', 'auto'`. Suppose that the trained SVM model is called `SVModel`. The software uses a heuristic procedure to select the kernel scale. The heuristic procedure uses subsampling. Therefore, to reproduce results, set a random number seed using `rng` before training the classifier.
- 2 Cross validate the classifier by passing it to `crossval`. By default, the software conducts 10-fold cross validation.
- 3 Pass the cross-validated SVM model to `kFoldLoss` to estimate and retain the classification error.
- 4 Retrain the SVM classifier, but adjust the `'KernelScale'` and `'BoxConstraint'` name-value pair arguments.
  - `BoxConstraint` — One strategy is to try a geometric sequence of the box constraint parameter. For example, take 11 values, from  $1e-5$  to  $1e5$  by a factor of 10. Increasing `BoxConstraint` might decrease the number of support vectors, but also might increase training time.
  - `KernelScale` — One strategy is to try a geometric sequence of the RBF sigma parameter scaled at the original kernel scale. Do this by:
    - a Retrieving the original kernel scale, e.g., `ks`, using dot notation: `ks = SVModel.KernelParameters.Scale`.
    - b Use as new kernel scales factors of the original. For example, multiply `ks` by the 11 values  $1e-5$  to  $1e5$ , increasing by a factor of 10.

Choose the model that yields the lowest classification error.

You might want to further refine your parameters to obtain better accuracy. Start with your initial parameters and perform another cross-validation step, this time using a factor of 1.2. Alternatively, optimize your parameters with `fminsearch`, as shown in “Optimize a Cross-Validated SVM Classifier Using Bayesian Optimization” on page 18-205.

## Train SVM Classifiers Using a Gaussian Kernel

This example shows how to generate a nonlinear classifier with Gaussian kernel function. First, generate one class of points inside the unit disk in two dimensions, and another class of points in the annulus from radius 1 to radius 2. Then, generates a classifier based on the data with the Gaussian radial basis function kernel. The default linear classifier is obviously unsuitable for this problem, since the model is circularly symmetric. Set the box constraint parameter to `Inf` to make a strict classification, meaning no misclassified training points. Other kernel functions might not work with this strict box constraint, since they might be unable to provide a strict classification. Even though the `rbf` classifier can separate the classes, the result can be overtrained.

Generate 100 points uniformly distributed in the unit disk. To do so, generate a radius  $r$  as the square root of a uniform random variable, generate an angle  $t$  uniformly in  $(0, 2\pi)$ , and put the point at  $(r \cos(t), r \sin(t))$ .

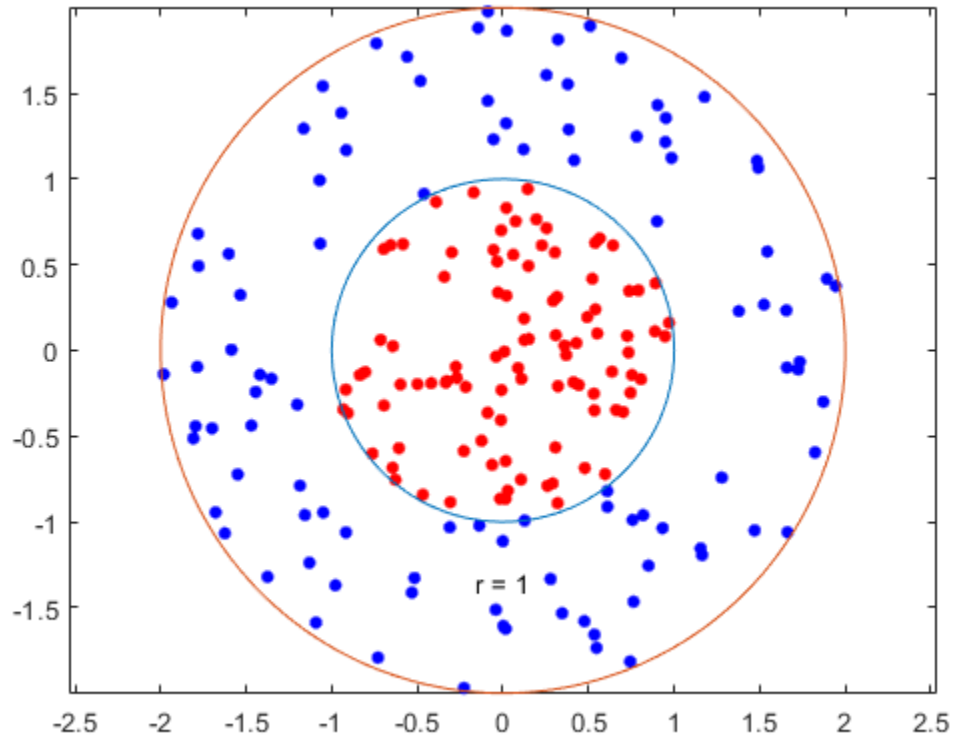
```
rng(1); % For reproducibility
r = sqrt(rand(100,1)); % Radius
t = 2*pi*rand(100,1); % Angle
data1 = [r.*cos(t), r.*sin(t)]; % Points
```

Generate 100 points uniformly distributed in the annulus. The radius is again proportional to a square root, this time a square root of the uniform distribution from 1 through 4.

```
r2 = sqrt(3*rand(100,1)+1); % Radius
t2 = 2*pi*rand(100,1); % Angle
data2 = [r2.*cos(t2), r2.*sin(t2)]; % points
```

Plot the points, and plot circles of radii 1 and 2 for comparison.

```
figure;
plot(data1(:,1),data1(:,2),'r.','MarkerSize',15)
hold on
plot(data2(:,1),data2(:,2),'b.','MarkerSize',15)
ezpolar(@(x)1);ezpolar(@(x)2);
axis equal
hold off
```



Put the data in one matrix, and make a vector of classifications.

```
data3 = [data1;data2];
theclass = ones(200,1);
theclass(1:100) = -1;
```

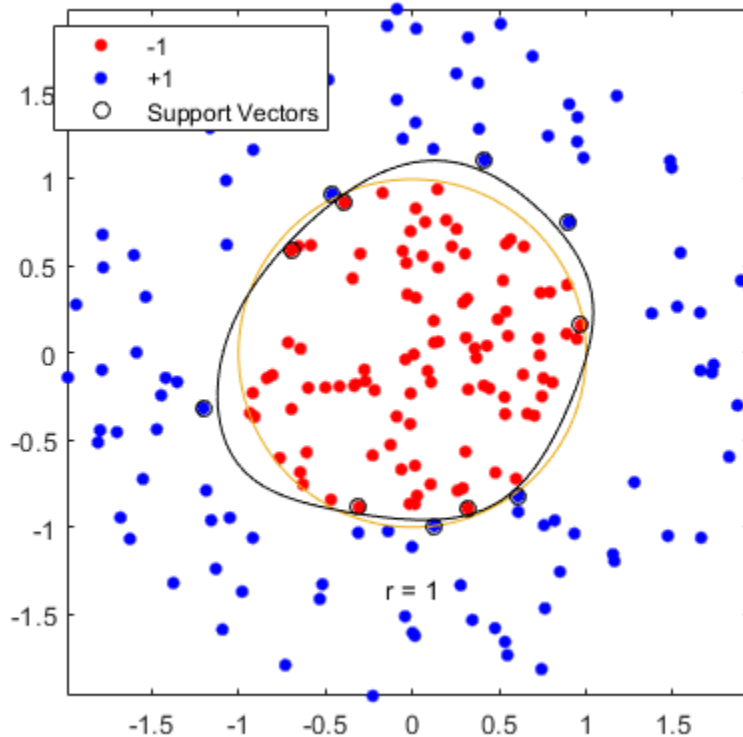
Train an SVM classifier with `KernelFunction` set to `'rbf'` and `BoxConstraint` set to `Inf`. Plot the decision boundary and flag the support vectors.

```
%Train the SVM Classifier
cl = fitcsvm(data3,theclass,'KernelFunction','rbf',...
    'BoxConstraint',Inf,'ClassNames',[-1,1]);

% Predict scores over the grid
```

```
d = 0.02;
[x1Grid,x2Grid] = meshgrid(min(data3(:,1)):d:max(data3(:,1)),...
    min(data3(:,2)):d:max(data3(:,2)));
xGrid = [x1Grid(:),x2Grid(:)];
[~,scores] = predict(c1,xGrid);

% Plot the data and the decision boundary
figure;
h(1:2) = gscatter(data3(:,1),data3(:,2),theclass,'rb','.');
hold on
ezpolar(@(x)1);
h(3) = plot(data3(c1.IsSupportVector,1),data3(c1.IsSupportVector,2),'ko');
contour(x1Grid,x2Grid,reshape(scores(:,2),size(x1Grid)),[0 0],'k');
legend(h,{'-1','+1','Support Vectors'});
axis equal
hold off
```



`fitcsvm` generates a classifier that is close to a circle of radius 1. The difference is due to the random training data.

Training with the default parameters makes a more nearly circular classification boundary, but one that misclassifies some training data. Also, the default value of `BoxConstraint` is 1, and, therefore, there are more support vectors.

```

c12 = fitcsvm(data3,theclass,'KernelFunction','rbf');
[~,scores2] = predict(c12,xGrid);

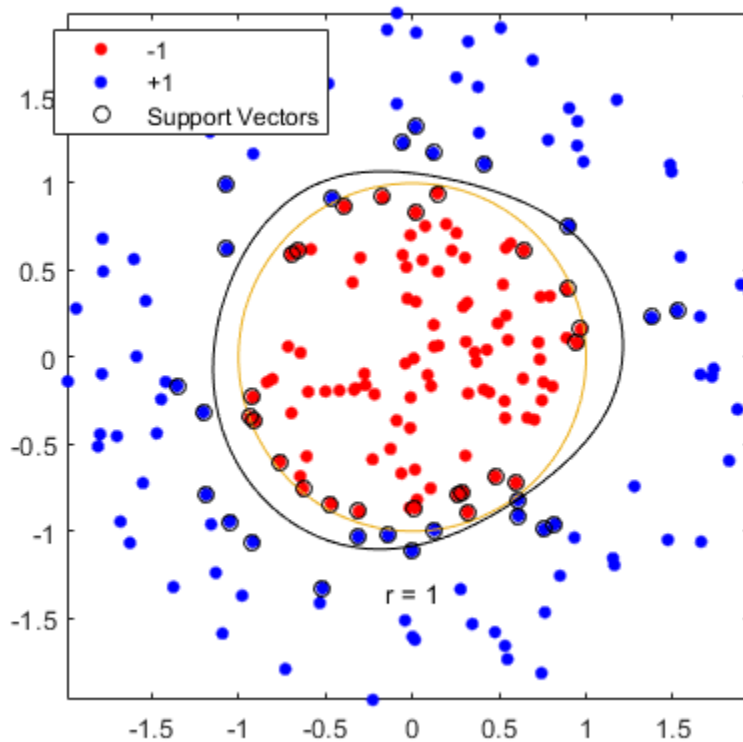
figure;
h(1:2) = gscatter(data3(:,1),data3(:,2),theclass,'rb','.');
hold on
ezpolar(@(x)1);

```

```

h(3) = plot(data3(c12.IsSupportVector,1),data3(c12.IsSupportVector,2), 'ko');
contour(x1Grid,x2Grid,reshape(scores2(:,2),size(x1Grid)),[0 0], 'k');
legend(h,{'-1','+1','Support Vectors'});
axis equal
hold off

```



## Train SVM Classifier Using Custom Kernel

This example shows how to use a custom kernel function, such as the sigmoid kernel, to train SVM classifiers, and adjust custom kernel function parameters.

Generate a random set of points within the unit circle. Label points in the first and third quadrants as belonging to the positive class, and those in the second and fourth quadrants in the negative class.

```
rng(1); % For reproducibility
n = 100; % Number of points per quadrant

r1 = sqrt(rand(2*n,1)); % Random radii
t1 = [pi/2*rand(n,1); (pi/2*rand(n,1)+pi)]; % Random angles for Q1 and Q3
X1 = [r1.*cos(t1) r1.*sin(t1)]; % Polar-to-Cartesian conversion

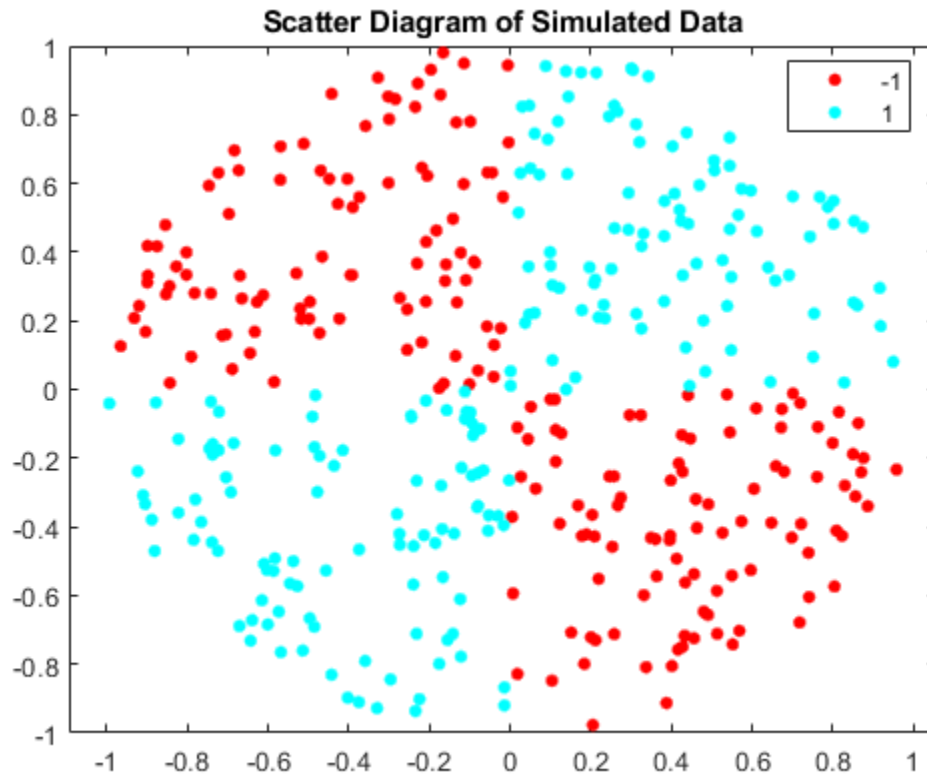
r2 = sqrt(rand(2*n,1));
t2 = [pi/2*rand(n,1)+pi/2; (pi/2*rand(n,1)-pi/2)]; % Random angles for Q2 and Q4
X2 = [r2.*cos(t2) r2.*sin(t2)];

X = [X1; X2]; % Predictors
Y = ones(4*n,1);
Y(2*n + 1:end) = -1; % Labels
```

Plot the data.

```
figure;
gscatter(X(:,1),X(:,2),Y);
title('Scatter Diagram of Simulated Data')
```





Write a function that accepts two matrices in the feature space as inputs, and transforms them into a Gram matrix using the sigmoid kernel.

```
function G = mysigmoid(U,V)
% Sigmoid kernel function with slope gamma and intercept c
gamma = 1;
c = -1;
G = tanh(gamma*U*V' + c);
end
```

Save this code as a file named `mysigmoid` on your MATLAB® path.

Train an SVM classifier using the sigmoid kernel function. It is good practice to standardize the data.

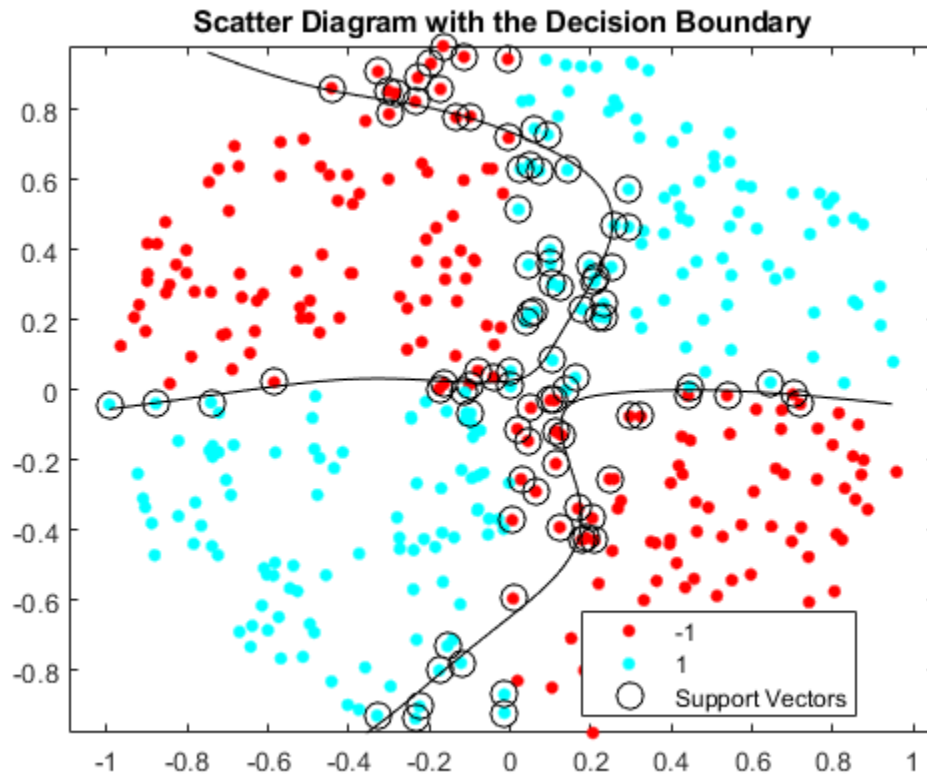
```
Mdl1 = fitcsvm(X,Y,'KernelFunction','mysigmoid','Standardize',true);
```

Mdl1 is a ClassificationSVM classifier containing the estimated parameters.

Plot the data, and identify the support vectors and the decision boundary.

```
% Compute the scores over a grid
d = 0.02; % Step size of the grid
[x1Grid,x2Grid] = meshgrid(min(X(:,1)):d:max(X(:,1)),...
    min(X(:,2)):d:max(X(:,2)));
xGrid = [x1Grid(:),x2Grid(:)]; % The grid
[~,scores1] = predict(Mdl1,xGrid); % The scores

figure;
h(1:2) = gscatter(X(:,1),X(:,2),Y);
hold on
h(3) = plot(X(Mdl1.IsSupportVector,1),...
    X(Mdl1.IsSupportVector,2),'ko','MarkerSize',10);
% Support vectors
contour(x1Grid,x2Grid,reshape(scores1(:,2),size(x1Grid)),[0 0],'k');
% Decision boundary
title('Scatter Diagram with the Decision Boundary')
legend({'-1','1','Support Vectors'},'Location','Best');
hold off
```



You can adjust the kernel parameters in an attempt to improve the shape of the decision boundary. This might also decrease the within-sample misclassification rate, but, you should first determine the out-of-sample misclassification rate.

Determine the out-of-sample misclassification rate by using 10-fold cross validation.

```
CVmdl1 = crossval(Mdl1);
misclass1 = kfoldLoss(CVmdl1);
misclass1
```

```
misclass1 =
```

```
0.1350
```

The out-of-sample misclassification rate is 13.5%.

Write another sigmoid function, but Set `gamma = 0.5;`.

```
function G = mysigmoid2(U,V)
% Sigmoid kernel function with slope gamma and intercept c
gamma = 0.5;
c = -1;
G = tanh(gamma*U*V' + c);
end
```

Save this code as a file named `mysigmoid2` on your MATLAB® path.

Train another SVM classifier using the adjusted sigmoid kernel. Plot the data and the decision region, and determine the out-of-sample misclassification rate.

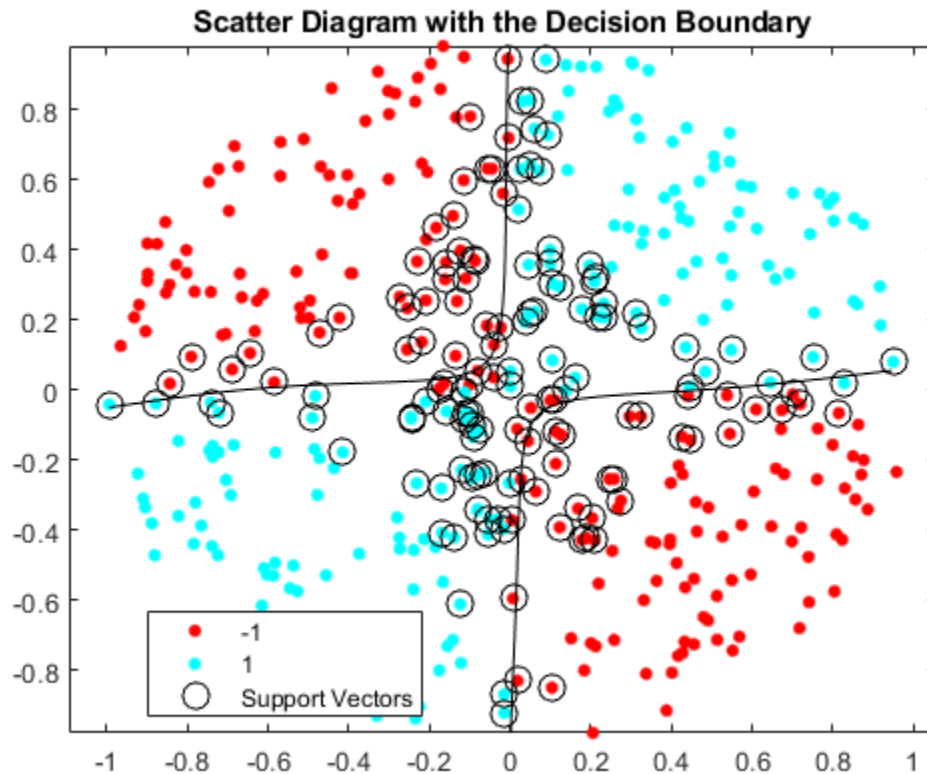
```
Mdl2 = fitcsvm(X,Y,'KernelFunction','mysigmoid2','Standardize',true);
[~,scores2] = predict(Mdl2,xGrid);

figure;
h(1:2) = gscatter(X(:,1),X(:,2),Y);
hold on
h(3) = plot(X(Mdl2.IsSupportVector,1),...
           X(Mdl2.IsSupportVector,2),'ko','MarkerSize',10);
title('Scatter Diagram with the Decision Boundary')
contour(x1Grid,x2Grid,reshape(scores2(:,2),size(x1Grid)),[0 0],'k');
legend({'-1','1','Support Vectors'},'Location','Best');
hold off

CVMdl2 = crossval(Mdl2);
misclass2 = kfoldLoss(CVMdl2);
misclass2

misclass2 =

0.0450
```



After the sigmoid slope adjustment, the new decision boundary seems to provide a better within-sample fit, and the cross-validation rate contracts by more than 66%.

## Optimize a Cross-Validated SVM Classifier Using Bayesian Optimization

This example shows how to optimize an SVM classification. The classification works on locations of points from a Gaussian mixture model. In *The Elements of Statistical Learning*, Hastie, Tibshirani, and Friedman (2009), page 17 describes the model. The model begins with generating 10 base points for a "green" class, distributed as 2-D independent normals with mean (1,0) and unit variance. It also generates 10 base points

for a "red" class, distributed as 2-D independent normals with mean (0,1) and unit variance. For each class (green and red), generate 100 random points as follows:

- 1 Choose a base point  $m$  of the appropriate color uniformly at random.
- 2 Generate an independent random point with 2-D normal distribution with mean  $m$  and variance  $I/5$ , where  $I$  is the 2-by-2 identity matrix. In this example, use a variance  $I/50$  to show the advantage of optimization more clearly.

After generating 100 green and 100 red points, classify them using `fitcsvm`. Then use `bayesopt` to optimize the parameters of the resulting SVM model with respect to cross validation.

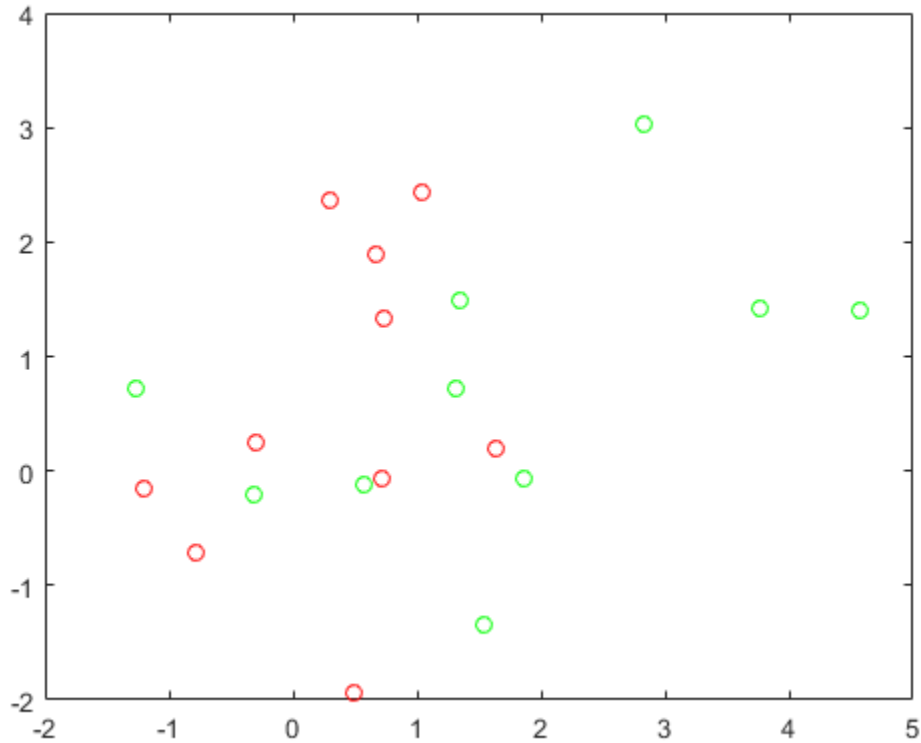
### Generate the Points and Classifier

Generate the 10 base points for each class.

```
rng default
grnpop = mvnrnd([1,0],eye(2),10);
redpop = mvnrnd([0,1],eye(2),10);
```

View the base points.

```
plot(grnpop(:,1),grnpop(:,2),'go')
hold on
plot(redpop(:,1),redpop(:,2),'ro')
hold off
```



Since some red base points are close to green base points, it can be difficult to classify the data points based on location alone.

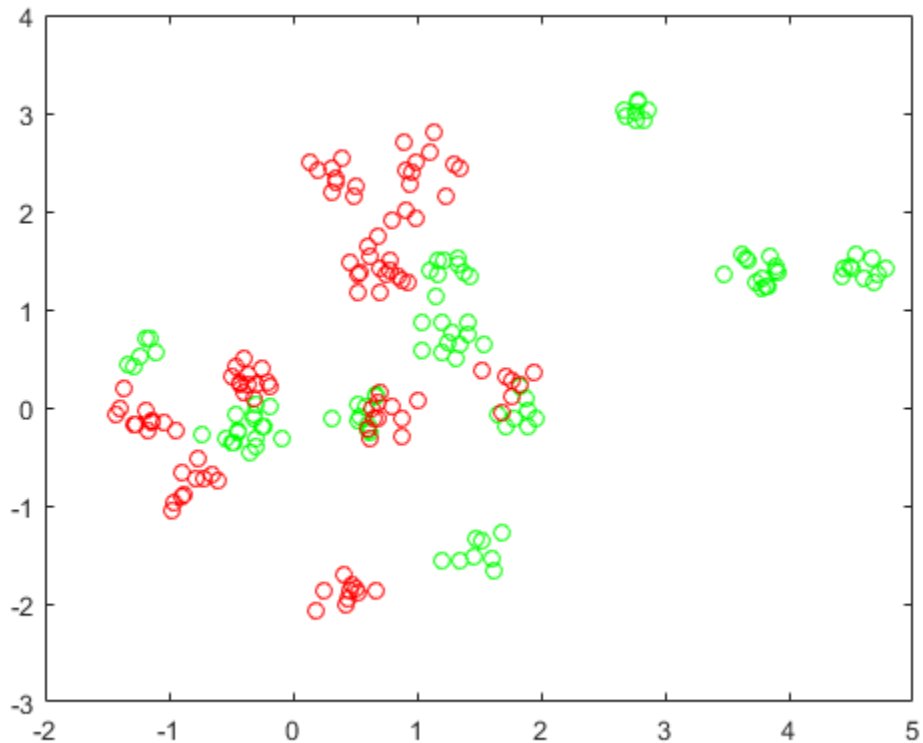
Generate the 100 data points of each class.

```
redpts = zeros(100,2);grnpts = redpts;  
for i = 1:100  
    grnpts(i,:) = mvnrnd(grnpop(randi(10),:),eye(2)*0.02);  
    redpts(i,:) = mvnrnd(redpop(randi(10),:),eye(2)*0.02);  
end
```

View the data points.

```
figure  
plot(grnpts(:,1),grnpts(:,2),'go')
```

```
hold on
plot(redpts(:,1),redpts(:,2),'ro')
hold off
```



### Prepare Data For Classification

Put the data into one matrix, and make a vector `grp` that labels the class of each point.

```
cdata = [grnpts;redpts];
grp = ones(200,1);
% Green label 1, red label -1
grp(101:200) = -1;
```



## Prepare Cross-Validation

Set up a partition for cross-validation. This step fixes the train and test sets that the optimization uses at each step.

```
c = cvpartition(200, 'KFold', 10);
```

## Prepare Variables for Bayesian Optimization

Set up a function that takes an input  $z = [\text{rbf\_sigma}, \text{boxconstraint}]$  and returns the cross-validation loss value of  $z$ . Take the components of  $z$  as positive, log-transformed variables between  $1e-5$  and  $1e5$ . Choose a wide range, because you don't know which values are likely to be good.

```
sigma = optimizableVariable('sigma', [1e-5, 1e5], 'Transform', 'log');
box = optimizableVariable('box', [1e-5, 1e5], 'Transform', 'log');
```

## Objective Function

This function handle computes the cross-validation loss at parameters  $[\text{sigma}, \text{box}]$ . For details, see `ClassificationPartitionedModel.kfoldLoss`.

`bayesopt` passes the variable  $z$  to the objective function as a one-row table.

```
minfn = @(z) kfoldLoss(fitcsvm(cdata, grp, 'CVPartition', c, ...
    'KernelFunction', 'rbf', 'BoxConstraint', z.box, ...
    'KernelScale', z.sigma));
```

## Optimize Classifier

Search for the best parameters  $[\text{sigma}, \text{box}]$  using `bayesopt`. For reproducibility, choose the 'expected-improvement-plus' acquisition function. The default acquisition function depends on run time, and so can give varying results.

```
results = bayesopt(minfn, [sigma, box], 'IsObjectiveDeterministic', true, ...
    'AcquisitionFunctionName', 'expected-improvement-plus')
```

```
=====
| Iter | Eval  | Objective  | Objective  | BestSoFar  | BestSoFar  |          sigma
|      | result |            | runtime    | (observed) | (estim.)   |
|-----|-----|-----|-----|-----|-----|-----|
|  1  | Best  |      0.61 |    0.2363 |      0.61 |      0.61 |    0.00013375
|  2  | Best  |    0.345 |    0.25656 |    0.345 |    0.345 |          24526
|  3  | Accept |    0.61 |    0.24767 |    0.345 |    0.345 |    0.0026459
```

	4	Accept		0.345		0.30048		0.345		0.345		3506.3
	5	Accept		0.345		0.25811		0.345		0.345		9135.2
	6	Accept		0.345		0.25434		0.345		0.345		99701
	7	Best		0.295		0.23028		0.295		0.295		455.88
	8	Best		0.24		4.3282		0.24		0.24		31.56
	9	Accept		0.24		5.0777		0.24		0.24		10.451
	10	Accept		0.35		0.25787		0.24		0.24		17.331
	11	Best		0.23		3.2936		0.23		0.23		16.005
	12	Best		0.1		0.45354		0.1		0.1		0.36562
	13	Accept		0.115		0.25502		0.1		0.1		0.1793
	14	Accept		0.105		0.30202		0.1		0.1		0.2267
	15	Best		0.095		0.26685		0.095		0.095		0.28999
	16	Best		0.075		0.24041		0.075		0.075		0.30554
	17	Accept		0.085		0.21978		0.075		0.075		0.41122
	18	Accept		0.085		0.24691		0.075		0.075		0.25565
	19	Accept		0.075		0.2465		0.075		0.075		0.32869
	20	Accept		0.085		0.23484		0.075		0.075		0.32442
=====												
	Iter	Eval		Objective		Objective		BestSoFar		BestSoFar		sigma
		result				runtime		(observed)		(estim.)		
=====												
	21	Accept		0.3		0.2144		0.075		0.075		1.3592
	22	Accept		0.12		0.23323		0.075		0.075		0.17515
	23	Accept		0.175		0.24112		0.075		0.075		0.1252
	24	Accept		0.105		0.25443		0.075		0.075		1.1664
	25	Accept		0.1		0.28356		0.075		0.075		0.57465
	26	Accept		0.12		0.24071		0.075		0.075		0.42922
	27	Accept		0.12		0.23573		0.075		0.075		0.42956
	28	Accept		0.095		0.23936		0.075		0.075		0.4806
	29	Accept		0.105		0.26069		0.075		0.075		0.19755
	30	Accept		0.205		0.30794		0.075		0.075		3.5051

---

Optimization completed.

MaxObjectiveEvaluations of 30 reached.

Total function evaluations: 30

Total elapsed time: 64.0366 seconds.

Total objective function evaluation time: 19.7181

Best observed feasible point:

sigma	box
_____	_____
0.30554	8.9017

Observed objective function value = 0.075  
 Estimated objective function value = 0.075  
 Function evaluation time = 0.24041

Best estimated feasible point (according to models):

sigma	box
0.32869	18.076

Estimated objective function value = 0.075  
 Estimated function evaluation time = 0.24164

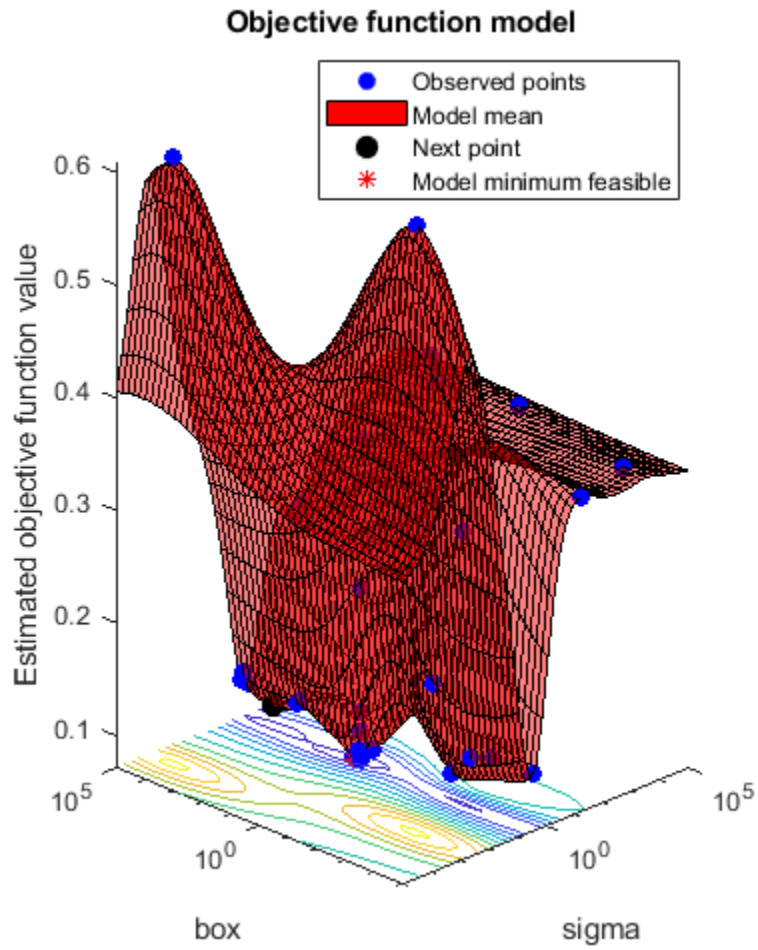
results =

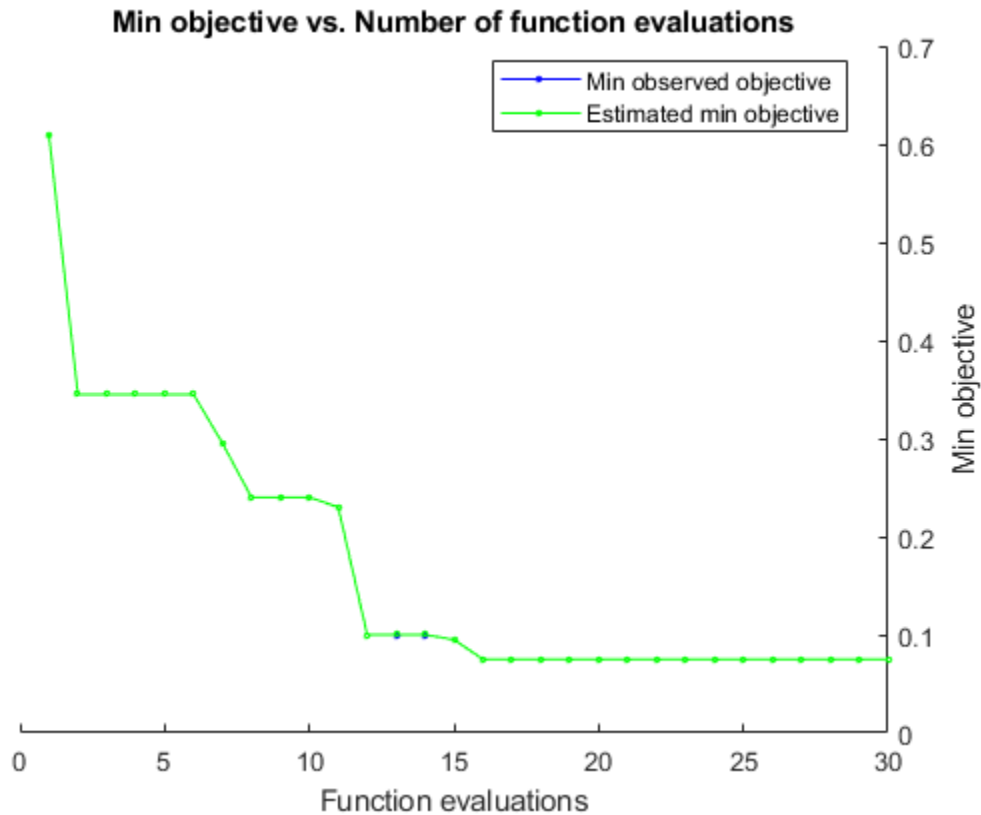
BayesianOptimization with properties:

```

    ObjectiveFcn: [function_handle]
  VariableDescriptions: [1x2 optimizableVariable]
        Options: [1x1 struct]
    MinObjective: 0.0750
    XAtMinObjective: [1x2 table]
  MinEstimatedObjective: 0.0750
  XAtMinEstimatedObjective: [1x2 table]
  NumObjectiveEvaluations: 30
    TotalElapsedTime: 64.0366
        NextPoint: [1x2 table]
          XTrace: [30x2 table]
    ObjectiveTrace: [30x1 double]
    ConstraintsTrace: []
        UserDataTrace: {30x1 cell}
  ObjectiveEvaluationTimeTrace: [30x1 double]
    IterationTimeTrace: [30x1 double]
        ErrorTrace: [30x1 double]
    FeasibilityTrace: [30x1 logical]
  FeasibilityProbabilityTrace: [30x1 double]
    IndexOfMinimumTrace: [30x1 double]
    ObjectiveMinimumTrace: [30x1 double]
  EstimatedObjectiveMinimumTrace: [30x1 double]

```





Use the results to train a new, optimized SVM classifier.

```
z(1) = results.XAtMinObjective.sigma;
z(2) = results.XAtMinObjective.box;
SVMModel = fitcsvm(cdata,grp,'KernelFunction','rbf',...
    'KernelScale',z(1),'BoxConstraint',z(2));
```

Plot the classification boundaries. To visualize the support vector classifier, predict scores over a grid.

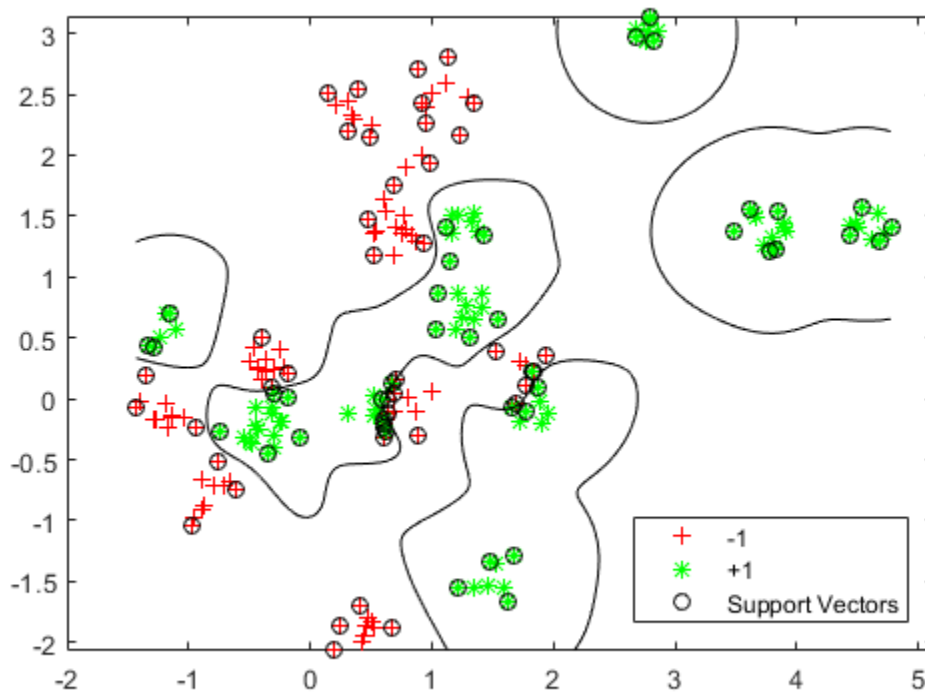
```
d = 0.02;
[x1Grid,x2Grid] = meshgrid(min(cdata(:,1)):d:max(cdata(:,1)),...
    min(cdata(:,2)):d:max(cdata(:,2)));
xGrid = [x1Grid(:),x2Grid(:)];
```

```

[~,scores] = predict(SVMModel,xGrid);

h = nan(3,1); % Preallocation
figure;
h(1:2) = gscatter(cdata(:,1),cdata(:,2),grp,'rg','+*');
hold on
h(3) = plot(cdata(SVMModel.IsSupportVector,1),...
            cdata(SVMModel.IsSupportVector,2),'ko');
contour(x1Grid,x2Grid,reshape(scores(:,2),size(x1Grid)),[0 0],'k');
legend(h,{'-1','+1','Support Vectors'},'Location','Southeast');
axis equal
hold off

```



## Evaluate Accuracy on New Data

Generate and classify some new data points.

```

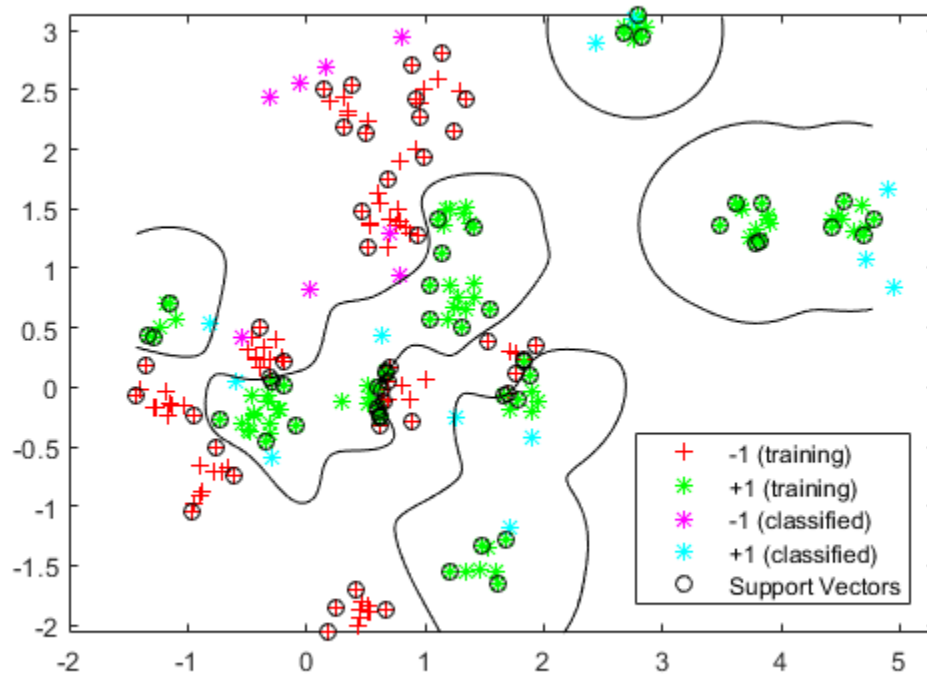
grnobj = gmdistribution(grnpop, .2*eye(2));
redobj = gmdistribution(redpop, .2*eye(2));

newData = random(grnobj,10);
newData = [newData;random(redobj,10)];
grpData = ones(20,1);
grpData(11:20) = -1; % red = -1

v = predict(SVMModel,newData);

g = nan(7,1);
figure;
h(1:2) = gscatter(cdata(:,1),cdata(:,2),grp,'rg','+*');
hold on
h(3:4) = gscatter(newData(:,1),newData(:,2),v,'mc','**');
h(5) = plot(cdata(SVMModel.IsSupportVector,1),...
           cdata(SVMModel.IsSupportVector,2),'ko');
contour(x1Grid,x2Grid,reshape(scores(:,2),size(x1Grid)),[0 0],'k');
legend(h(1:5),{'-1 (training)', '+1 (training)', '-1 (classified)', ...
              '+1 (classified)', 'Support Vectors'}, 'Location', 'Southeast');
axis equal
hold off

```



See which new data points are correctly classified. Circle the correctly classified points in red, and the incorrectly classified points in black.

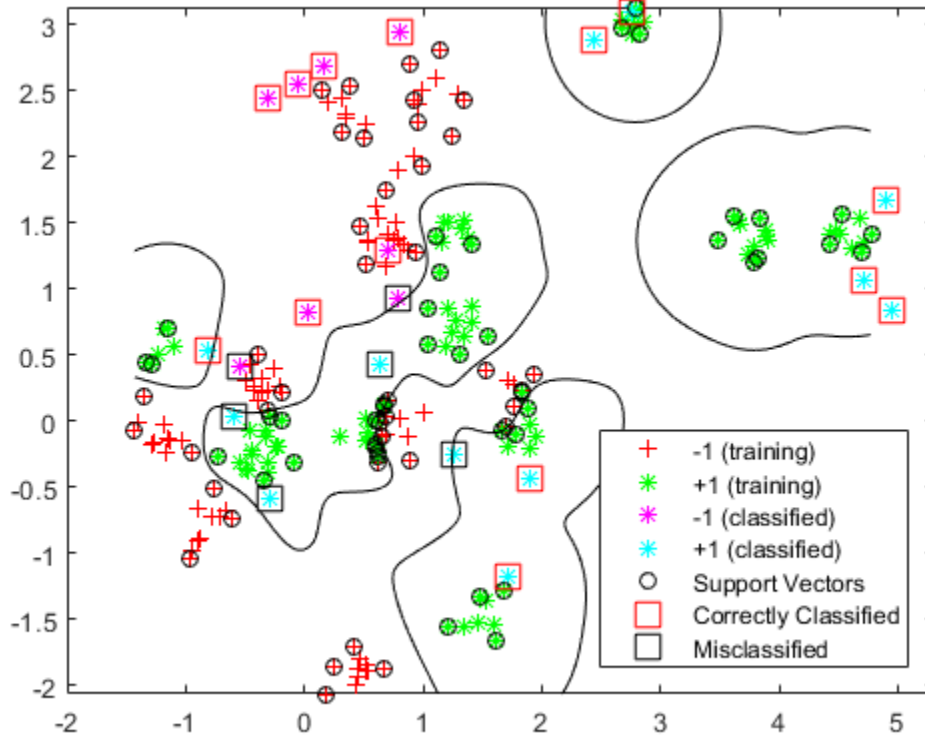
```
mydiff = (v == grpData); % Classified correctly
figure;
h(1:2) = gscatter(cdata(:,1),cdata(:,2),grp,'rg','+*');
hold on
h(3:4) = gscatter(newData(:,1),newData(:,2),v,'mc','**');
h(5) = plot(cdata(SVMModel.IsSupportVector,1),...
    cdata(SVMModel.IsSupportVector,2),'ko');
contour(x1Grid,x2Grid,reshape(scores(:,2),size(x1Grid)),[0 0],'k');

for ii = mydiff % Plot red squares around correct pts
    h(6) = plot(newData(ii,1),newData(ii,2),'rs','MarkerSize',12);
```



```
end

for ii = not(mydiff) % Plot black squares around incorrect pts
    h(7) = plot(newData(ii,1),newData(ii,2),'ks','MarkerSize',12);
end
legend(h,{'-1 (training)','+1 (training)','-1 (classified)',...
    '+1 (classified)','Support Vectors','Correctly Classified',...
    'Misclassified'},'Location','Southeast');
hold off
```



## Plot Posterior Probability Regions for SVM Classification Models

This example shows how to predict posterior probabilities of SVM models over a grid of observations, and then plot the posterior probabilities over the grid. Plotting posterior probabilities exposes decision boundaries.

Load Fisher's iris data set. Train the classifier using the petal lengths and widths, and remove the virginica species from the data.

```
load fisheriris
classKeep = ~strcmp(species, 'virginica');
X = meas(classKeep, 3:4);
y = species(classKeep);
```

Train an SVM classifier using the data. It is good practice to specify the order of the classes.

```
SVMModel = fitcsvm(X, y, 'ClassNames', {'setosa', 'versicolor'});
```

Estimate the optimal score transformation function.

```
rng(1); % For reproducibility
[SVMModel, ScoreParameters] = fitPosterior(SVMModel);
ScoreParameters
```

Warning: Classes are perfectly separated. The optimal score-to-posterior transformation is a step function.

```
ScoreParameters =
    struct with fields:
        Type: 'step'
        LowerBound: -0.8431
        UpperBound: 0.6897
        PositiveClassProbability: 0.5000
```

The optimal score transformation function is the step function because the classes are separable. The fields `LowerBound` and `UpperBound` of `ScoreParameters` indicate the lower and upper end points of the interval of scores corresponding to observations within the class-separating hyperplanes (the margin). No training observation falls within the margin. If a new score is in the interval, then the software assigns the corresponding observation a positive class posterior probability, i.e., the value in the `PositiveClassProbability` field of `ScoreParameters`.

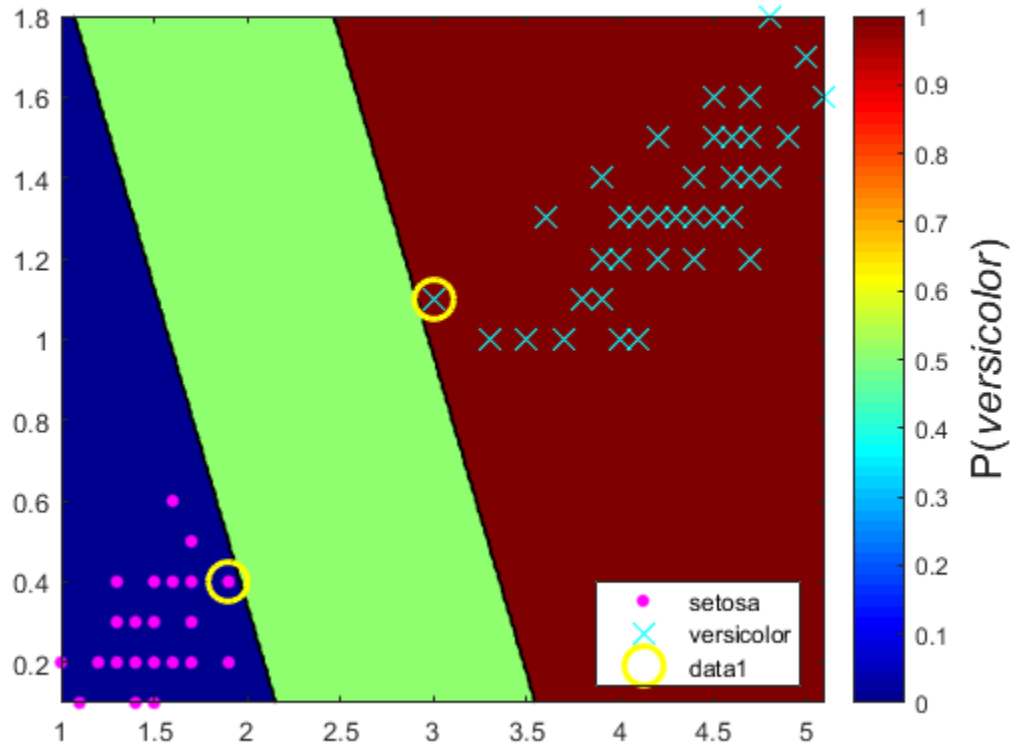
Define a grid of values in the observed predictor space. Predict the posterior probabilities for each instance in the grid.

```
xMax = max(X);
xMin = min(X);
d = 0.01;
[x1Grid, x2Grid] = meshgrid(xMin(1):d:xMax(1), xMin(2):d:xMax(2));
[~, PosteriorRegion] = predict(SVMModel, [x1Grid(:), x2Grid(:)]);
```

Plot the positive class posterior probability region and the training data.

```
figure;
contourf(x1Grid,x2Grid,...
         reshape(PosteriorRegion(:,2),size(x1Grid,1),size(x1Grid,2)));
h = colorbar;
h.Label.String = 'P({\it{versicolor}})';
h.YLabel.FontSize = 16;
caxis([0 1]);
colormap jet;

hold on
gscatter(X(:,1),X(:,2),y,'mc','.x',[15,10]);
sv = X(SVMModel.IsSupportVector,:);
plot(sv(:,1),sv(:,2),'yo','MarkerSize',15,'LineWidth',2);
axis tight
hold off
```



In two-class learning, if the classes are separable, then there are three regions: one where observations have positive class posterior probability 0, one where it is 1, and the other where it is the positive class prior probability.

## Analyze Images Using Linear Support Vector Machines

This example shows how to determine which quadrant of an image a shape occupies by training an error-correcting output codes (ECOC) model comprised of linear SVM binary learners. This example also illustrates the disk-space consumption of ECOC models that store support vectors, their labels, and the estimated  $\alpha$  coefficients.

### Create the Data Set

Randomly place a circle with radius five in a 50-by-50 image. Make 5000 images. Create a label for each image indicating the quadrant that the circle occupies. Quadrant 1 is in the upper right, quadrant 2 is in the upper left, quadrant 3 is in the lower left, and quadrant 4 is in the lower right. The predictors are the intensities of each pixel.

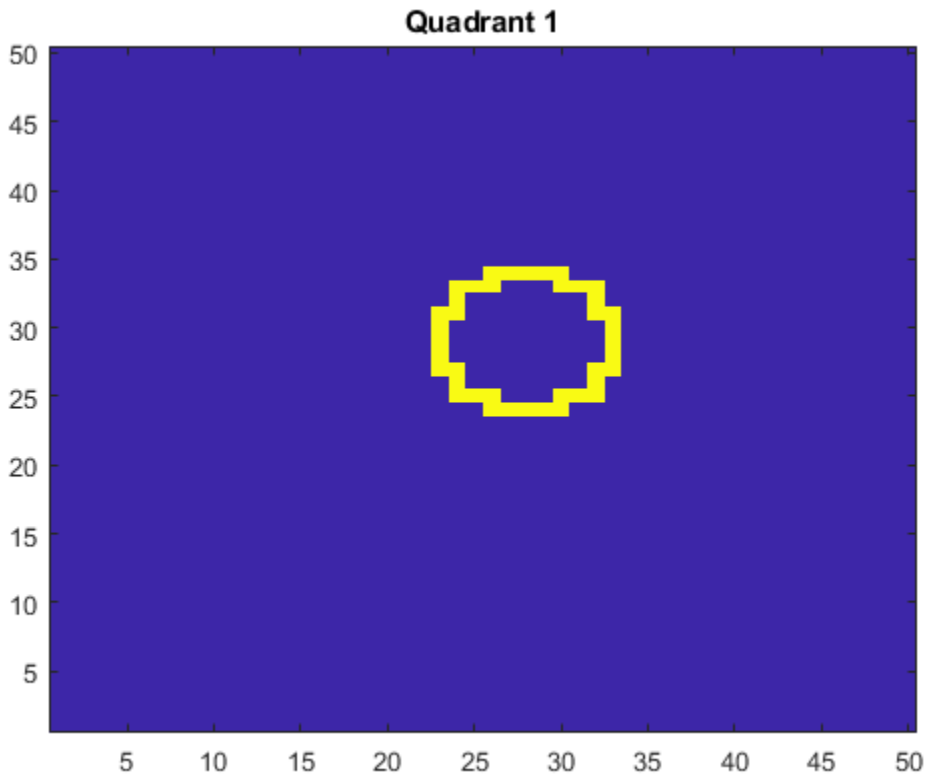
```
d = 50; % Height and width of the images in pixels
n = 5e4; % Sample size

X = zeros(n,d^2); % Predictor matrix preallocation
Y = zeros(n,1); % Label preallocation
theta = 0:(1/d):(2*pi);
r = 5; % Circle radius
rng(1); % For reproducibility

for j = 1:n;
    figmat = zeros(d); % Empty image
    c = datasample((r + 1):(d - r - 1),2); % Random circle center
    x = r*cos(theta) + c(1); % Make the circle
    y = r*sin(theta) + c(2);
    idx = sub2ind([d d],round(y),round(x)); % Convert to linear indexing
    figmat(idx) = 1; % Draw the circle
    X(j,:) = figmat(:); % Store the data
    Y(j) = (c(2) >= floor(d/2)) + 2*(c(2) < floor(d/2)) + ...
           (c(1) < floor(d/2)) + ...
           2*((c(1) >= floor(d/2)) & (c(2) < floor(d/2))); % Determine the quadrant
end
```

Plot an observation.

```
figure;
imagesc(figmat);
h = gca;
h.YDir = 'normal';
title(sprintf('Quadrant %d',Y(end)));
```



### Train the ECOC Model

Use a 25% holdout sample and specify the training and holdout sample indices.

```
p = 0.25;
CVP = cvpartition(Y, 'Holdout', p); % Cross-validation data partition
isIdx = training(CVP); % Training sample indices
oosIdx = test(CVP); % Test sample indices
```

Create an SVM template that specifies storing the support vectors of the binary learners. Pass it and the training data to `fitcecoc` to train the model. Determine the training sample classification error.

```
t = templateSVM('SaveSupportVectors',true);
MdlSV = fitcecoc(X(isIdx,:),Y(isIdx),'Learners',t);
isLoss = resubLoss(MdlSV)
```

```
isLoss =
    0
```

MdlSV is a trained `ClassificationECOC` multiclass model. It stores the training data and the support vectors of each binary learner. For large data sets, such as those in image analysis, the model can consume a lot of memory.

Determine the amount of disk space that the ECOC model consumes.

```
infoMdlSV = whos('MdlSV');
mbMdlSV = infoMdlSV.bytes/1.049e6
```

```
mbMdlSV =
    763.5919
```

The model consumes 1477.5 MB.

### Improve Model Efficiency

You can assess out-of-sample performance. You can also assess whether the model has been overfit with a compacted model that does not contain the support vectors, their related parameters, and the training data.

Discard the support vectors and related parameters from the trained ECOC model. Then, discard the training data from the resulting model by using `compact`.

```
Mdl = discardSupportVectors(MdlSV);
CMdl = compact(Mdl);
info = whos('Mdl','CMdl');
[bytesCMdl,bytesMdl] = info.bytes;
memReduction = 1 - [bytesMdl bytesCMdl]/infoMdlSV.bytes
```

```
memReduction =
```



```
0.0626    0.9996
```

In this case, discarding the support vectors reduces the memory consumption by about 3%. Compacting and discarding support vectors reduces the size by about 99.99%.

An alternative way to manage support vectors is to reduce their numbers during training by specifying a larger box constraint, such as 100. Though SVM models that use fewer support vectors are more desirable and consume less memory, increasing the value of the box constraint tends to increase the training time.

Remove `MdlSV` and `Mdl` from the workspace.

```
clear Mdl MdlSV;
```

### Assess Holdout Sample Performance

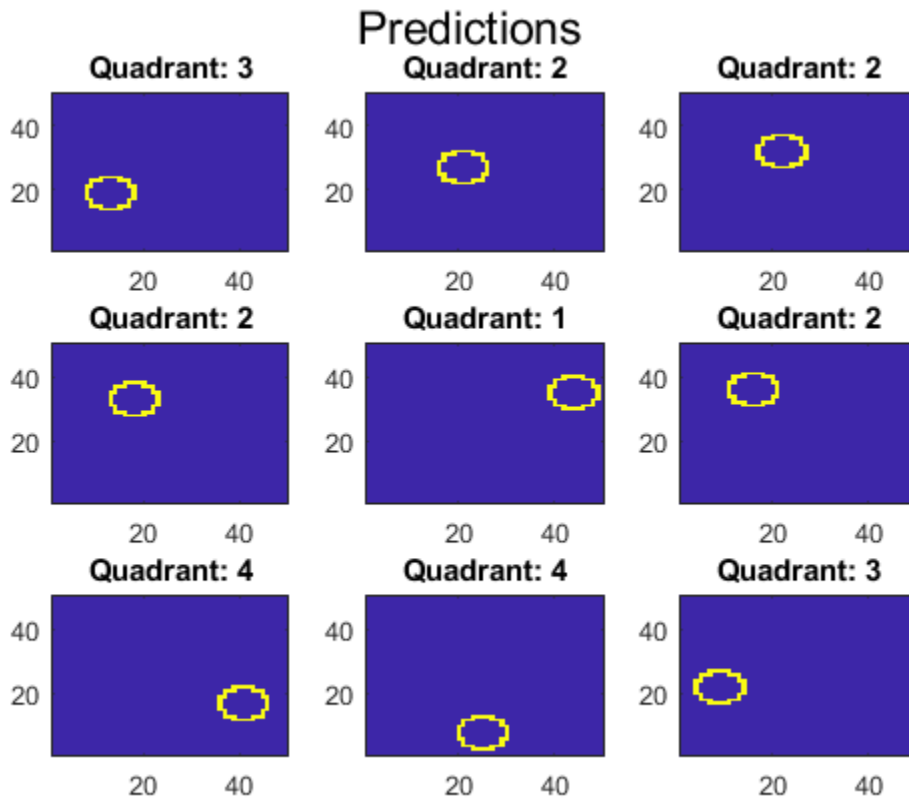
Calculate the classification error of the holdout sample. Plot a sample of the holdout sample predictions.

```
oosLoss = loss(CMdl,X(oosIdx,:),Y(oosIdx))
yHat = predict(CMdl,X(oosIdx,:));
nVec = 1:size(X,1);
oosIdx = nVec(oosIdx);

figure;
for j = 1:9;
    subplot(3,3,j)
        imagesc(reshape(X(oosIdx(j),:),[d d]));
        h = gca;
        h.YDir = 'normal';
        title(sprintf('Quadrant: %d',yHat(j)))
end
text(-1.33*d,4.5*d + 1,'Predictions','FontSize',17)

oosLoss =

    0
```



The model does not misclassify any holdout sample observations.

## References

- [1] Hastie, T., R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*, second edition. New York: Springer, 2008.
- [2] Christianini, N., and J. Shawe-Taylor. *An Introduction to Support Vector Machines and Other Kernel-Based Learning Methods*. Cambridge, UK: Cambridge University Press, 2000.

- [3] Fan, R.-E., P.-H. Chen, and C.-J. Lin. “Working set selection using second order information for training support vector machines.” *Journal of Machine Learning Research*, Vol 6, 2005, pp. 1889–1918.
- [4] Kecman V., T. -M. Huang, and M. Vogt. “Iterative Single Data Algorithm for Training Kernel Machines from Huge Data Sets: Theory and Performance.” In *Support Vector Machines: Theory and Applications*. Edited by Lipo Wang, 255–274. Berlin: Springer-Verlag, 2005.

## Bibliography

- [1] Agresti, A. *Categorical Data Analysis*, 2nd Ed. Hoboken, NJ: John Wiley & Sons, Inc., 2002.
- [2] Allwein, E., R. Schapire, and Y. Singer. "Reducing multiclass to binary: A unifying approach for margin classifiers." *Journal of Machine Learning Research*. Vol. 1, 2000, pp. 113–141.
- [3] Alpaydin, E. "Combined 5 x 2 CV F Test for Comparing Supervised Classification Learning Algorithms." *Neural Computation*, Vol. 11, No. 8, 1999, pp. 1885–1992.
- [4] Blackard, J. A. and D. J. Dean. "Comparative accuracies of artificial neural networks and discriminant analysis in predicting forest cover types from cartographic variables". *Computers and Electronics in Agriculture* Vol. 24, Issue 3, 1999, pp. 131–151.
- [5] Bottou, L., and Chih-Jen Lin. "Support Vector Machine Solvers." *Large Scale Kernel Machines* (L. Bottou, O. Chapelle, D. DeCoste, and J. Weston, eds.). Cambridge, MA: MIT Press, 2007.
- [6] Bouckaert, R. "Choosing Between Two Learning Algorithms Based on Calibrated Tests." *International Conference on Machine Learning*, pp. 51–58, 2003.
- [7] Bouckaert, R. and E. Frank. "Evaluating the Replicability of Significance Tests for Comparing Learning Algorithms." *In Advances in Knowledge Discovery and Data Mining, 8th Pacific-Asia Conference*, 2004, pp. 3–12.
- [8] Breiman, L. *Bagging Predictors*. *Machine Learning* 26, 1996, pp. 123–140.
- [9] Breiman, L. *Random Forests*. *Machine Learning* 45, 2001, pp. 5–32.
- [10] Breiman, L. <http://www.stat.berkeley.edu/~breiman/RandomForests/>
- [11] Breiman, L., J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*. Boca Raton, FL: Chapman & Hall, 1984.
- [12] Christianini, N., and J. Shawe-Taylor. *An Introduction to Support Vector Machines and Other Kernel-Based Learning Methods*. Cambridge, UK: Cambridge University Press, 2000.

- [13] Dietterich, T. “Approximate statistical tests for comparing supervised classification learning algorithms.” *Neural Computation*, Vol. 10, No. 7, 1998, pp. 1895–1923.
- [14] Dietterich, T., and G. Bakiri. “Solving Multiclass Learning Problems Via Error-Correcting Output Codes.” *Journal of Artificial Intelligence Research*. Vol. 2, 1995, pp. 263–286.
- [15] Escalera, S., O. Pujol, and P. Radeva. “On the decoding process in ternary error-correcting output codes.” *IEEE Transactions on Pattern Analysis and Machine Intelligence*. Vol. 32, Issue 7, 2010, pp. 120–134.
- [16] Escalera, S., O. Pujol, and P. Radeva. “Separability of ternary codes for sparse designs of error-correcting output codes.” *Pattern Recogn.* Vol. 30, Issue 3, 2009, pp. 285–297.
- [17] Fan, R.-E., P.-H. Chen, and C.-J. Lin. “Working set selection using second order information for training support vector machines.” *Journal of Machine Learning Research*, Vol 6, 2005, pp. 1889–1918.
- [18] Fagerlan, M.W., S Lydersen, P. Laake. “The McNemar Test for Binary Matched-Pairs Data: Mid-p and Asymptotic Are Better Than Exact Conditional.” *BMC Medical Research Methodology*. Vol. 13, 2013, pp. 1–8.
- [19] Freund, Y. *A more robust boosting algorithm*. arXiv:0905.2138v1, 2009.
- [20] Freund, Y. and R. E. Schapire. *A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting*. J. of Computer and System Sciences, Vol. 55, 1997, pp. 119–139.
- [21] Friedman, J. *Greedy function approximation: A gradient boosting machine*. Annals of Statistics, Vol. 29, No. 5, 2001, pp. 1189–1232.
- [22] Friedman, J., T. Hastie, and R. Tibshirani. *Additive logistic regression: A statistical view of boosting*. Annals of Statistics, Vol. 28, No. 2, 2000, pp. 337–407.
- [23] Hastie, T., and R. Tibshirani. “Classification by Pairwise Coupling.” *Annals of Statistics*. Vol. 26, Issue 2, 1998, pp. 451–471.
- [24] Hastie, T., R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*, second edition. New York: Springer, 2008.
- [25] Ho, C. H. and C. J. Lin. “Large-Scale Linear Support Vector Regression.” *Journal of Machine Learning Research*, Vol. 13, 2012, pp. 3323–3348.

- [26] Ho, T. K. *The random subspace method for constructing decision forests*. IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 20, No. 8, 1998, pp. 832–844.
- [27] Hsieh, C. J., K. W. Chang, C. J. Lin, S. S. Keerthi, and S. Sundararajan. “A Dual Coordinate Descent Method for Large-Scale Linear SVM.” *Proceedings of the 25th International Conference on Machine Learning, ICML '08*, 2001, pp. 408–415.
- [28] Hsu, Chih-Wei, Chih-Chung Chang, and Chih-Jen Lin. *A Practical Guide to Support Vector Classification*. Available at <http://www.csie.ntu.edu.tw/~cjlin/papers/guide/guide.pdf>.
- [29] Hu, Q., X. Che, L. Zhang, and D. Yu. “Feature Evaluation and Selection Based on Neighborhood Soft Margin.” *Neurocomputing*. Vol. 73, 2010, pp. 2114–2124.
- [30] Kecman V., T. -M. Huang, and M. Vogt. “Iterative Single Data Algorithm for Training Kernel Machines from Huge Data Sets: Theory and Performance.” In *Support Vector Machines: Theory and Applications*. Edited by Lipo Wang, 255–274. Berlin: Springer-Verlag, 2005.
- [31] Kohavi, R. “Scaling Up the Accuracy of Naive-Bayes Classifiers: a Decision-Tree Hybrid.” *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, 1996.
- [32] Lancaster, H.O. “Significance Tests in Discrete Distributions.” *JASA*, Vol. 56, Number 294, 1961, pp. 223–234.
- [33] Langford, J., L. Li, and T. Zhang. “Sparse Online Learning Via Truncated Gradient.” *J. Mach. Learn. Res.*, Vol. 10, 2009, pp. 777–801.
- [34] Loh, W.Y. “Regression Trees with Unbiased Variable Selection and Interaction Detection.” *Statistica Sinica*, Vol. 12, 2002, pp. 361–386.
- [35] Loh, W.Y. and Y.S. Shih. “Split Selection Methods for Classification Trees.” *Statistica Sinica*, Vol. 7, 1997, pp. 815–840.
- [36] McNemar, Q. “Note on the Sampling Error of the Difference Between Correlated Proportions or Percentages.” *Psychometrika*, Vol. 12, Number 2, 1947, pp. 153–157.
- [37] Meinshausen, N. “Quantile Regression Forests.” *Journal of Machine Learning Research*, Vol. 7, 2006, pp. 983–999.

- [38] Mosteller, F. “Some Statistical Problems in Measuring the Subjective Response to Drugs.” *Biometrics*, Vol. 8, Number 3, 1952, pp. 220–226.
- [39] Nocedal, J. and S. J. Wright. *Numerical Optimization*, 2nd ed., New York: Springer, 2006.
- [40] Schapire, R. E. et al. *Boosting the margin: A new explanation for the effectiveness of voting methods*. *Annals of Statistics*, Vol. 26, No. 5, 1998, pp. 1651–1686.
- [41] Schapire, R., and Y. Singer. *Improved boosting algorithms using confidence-rated predictions*. *Machine Learning*, Vol. 37, No. 3, 1999, pp. 297–336.
- [42] Shalev-Shwartz, S., Y. Singer, and N. Srebro. “Pegasos: Primal Estimated Sub-Gradient Solver for SVM.” *Proceedings of the 24th International Conference on Machine Learning, ICML '07*, 2007, pp. 807–814.
- [43] Seiffert, C., T. Khoshgoftaar, J. Hulse, and A. Napolitano. *RUSBoost: Improving clasification performance when training data is skewed*. 19th International Conference on Pattern Recognition, 2008, pp. 1–4.
- [44] Warmuth, M., J. Liao, and G. Ratsch. *Totally corrective boosting algorithms that maximize the margin*. Proc. 23rd Int'l. Conf. on Machine Learning, ACM, New York, 2006, pp. 1001–1008.
- [45] Wu, T. F., C. J. Lin, and R. Weng. “Probability Estimates for Multi-Class Classification by Pairwise Coupling.” *Journal of Machine Learning Research*. Vol. 5, 2004, pp. 975–1005.
- [46] Wright, S. J., R. D. Nowak, and M. A. T. Figueiredo. “Sparse Reconstruction by Separable Approximation.” *Trans. Sig. Proc.*, Vol. 57, No 7, 2009, pp. 2479–2493.
- [47] Xiao, Lin. “Dual Averaging Methods for Regularized Stochastic Learning and Online Optimization.” *J. Mach. Learn. Res.*, Vol. 11, 2010, pp. 2543–2596.
- [48] Xu, Wei. “Towards Optimal One Pass Large Scale Learning with Averaged Stochastic Gradient Descent.” *CoRR*, abs/1107.2490, 2011.
- [49] Zadrozny, B. “Reducing Multiclass to Binary by Coupling Probability Estimates.” *NIPS 2001: Proceedings of Advances in Neural Information Processing Systems 14*, 2001, pp. 1041–1048.

- [50] Zadrozny, B., J. Langford, and N. Abe. “Cost-Sensitive Learning by Cost-Proportionate Example Weighting.” *Third IEEE International Conference on Data Mining*, 435–442. 2003.
- [51] Zhou, Z.-H. and X.-Y. Liu. “On Multi-Class Cost-Sensitive Learning.” *Computational Intelligence*. Vol. 26, Issue 3, 2010, pp. 232–257 CiteSeerX.



# Decision Trees

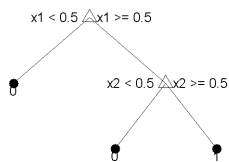
---

- “Decision Trees” on page 19-2
- “View Decision Tree” on page 19-5
- “Growing Decision Trees” on page 19-9
- “Prediction Using Classification and Regression Trees” on page 19-11
- “Predict Out-of-Sample Responses of Subtrees” on page 19-13
- “Improving Classification Trees and Regression Trees” on page 19-17
- “Alternative: classregtree” on page 19-32
- “Splitting Categorical Predictors” on page 19-42

## Decision Trees

Decision trees, or classification trees and regression trees, predict responses to data. To predict a response, follow the decisions in the tree from the root (beginning) node down to a leaf node. The leaf node contains the response. Classification trees give responses that are nominal, such as 'true' or 'false'. Regression trees give numeric responses.

Statistics and Machine Learning Toolbox trees are binary. Each step in a prediction involves checking the value of one predictor (variable). For example, here is a simple classification tree:



This tree predicts classifications based on two predictors,  $x_1$  and  $x_2$ . To predict, start at the top node, represented by a triangle ( $\Delta$ ). The first decision is whether  $x_1$  is smaller than 0.5. If so, follow the left branch, and see that the tree classifies the data as type 0.

If, however,  $x_1$  exceeds 0.5, then follow the right branch to the lower-right triangle node. Here the tree asks if  $x_2$  is smaller than 0.5. If so, then follow the left branch to see that the tree classifies the data as type 0. If not, then follow the right branch to see that the tree classifies the data as type 1.

To learn how to prepare your data for classification or regression using decision trees, see “Steps in Supervised Learning” on page 18-3.

### Train Classification Tree

This example shows how to train a classification tree.

Create a classification tree using the entire `ionosphere` data set.

```
load ionosphere % Contains X and Y variables
Mdl = fitctree(X,Y)
```

```
Mdl =
```

```
ClassificationTree
```

```
        ResponseName: 'Y'  
CategoricalPredictors: []  
        ClassNames: {'b' 'g'}  
        ScoreTransform: 'none'  
        NumObservations: 351
```

## Train Regression Tree

This example shows how to train a regression tree.

Create a regression tree using all observation in the `carsmall` data set. Consider the `Horsepower` and `Weight` vectors as predictor variables, and the `MPG` vector as the response.

```
load carsmall % Contains Horsepower, Weight, MPG  
X = [Horsepower Weight];  
  
Mdl = fitrtree(X,MPG)
```

```
Mdl =  
  
    RegressionTree  
        ResponseName: 'Y'  
    CategoricalPredictors: []  
        ResponseTransform: 'none'  
        NumObservations: 94
```

## References

- [1] Breiman, L., J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*. Boca Raton, FL: Chapman & Hall, 1984.

## See Also

`ClassificationTree` | `RegressionTree` | `fitctree` | `fitrtree`

## **Related Examples**

- “View Decision Tree” on page 19-5
- “Growing Decision Trees” on page 19-9
- “Prediction Using Classification and Regression Trees” on page 19-11
- “Improving Classification Trees and Regression Trees” on page 19-17

## View Decision Tree

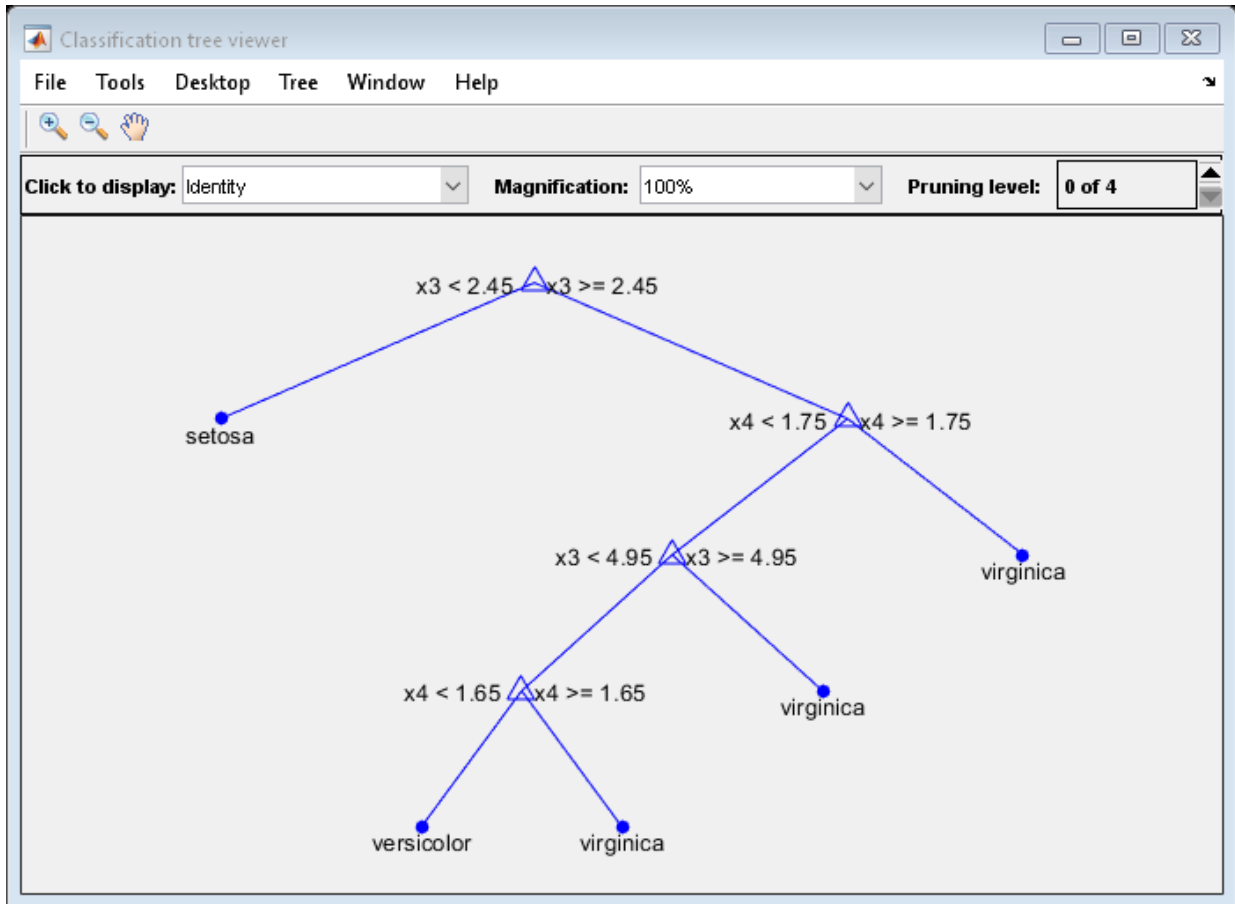
This example shows how to view a classification or regression tree. There are two ways to view a tree: `view(tree)` returns a text description and `view(tree, 'mode', 'graph')` returns a graphic description of the tree.

Create and view a classification tree.

```
load fisheriris % load the sample data
ctree = fitctree(meas,species); % create classification tree
view(ctree) % text description
```

```
Decision tree for classification
1  if x3<2.45 then node 2 elseif x3>=2.45 then node 3 else setosa
2  class = setosa
3  if x4<1.75 then node 4 elseif x4>=1.75 then node 5 else versicolor
4  if x3<4.95 then node 6 elseif x3>=4.95 then node 7 else versicolor
5  class = virginica
6  if x4<1.65 then node 8 elseif x4>=1.65 then node 9 else versicolor
7  class = virginica
8  class = versicolor
9  class = virginica
```

```
view(ctree, 'mode', 'graph') % graphic description
```



Now, create and view a regression tree.

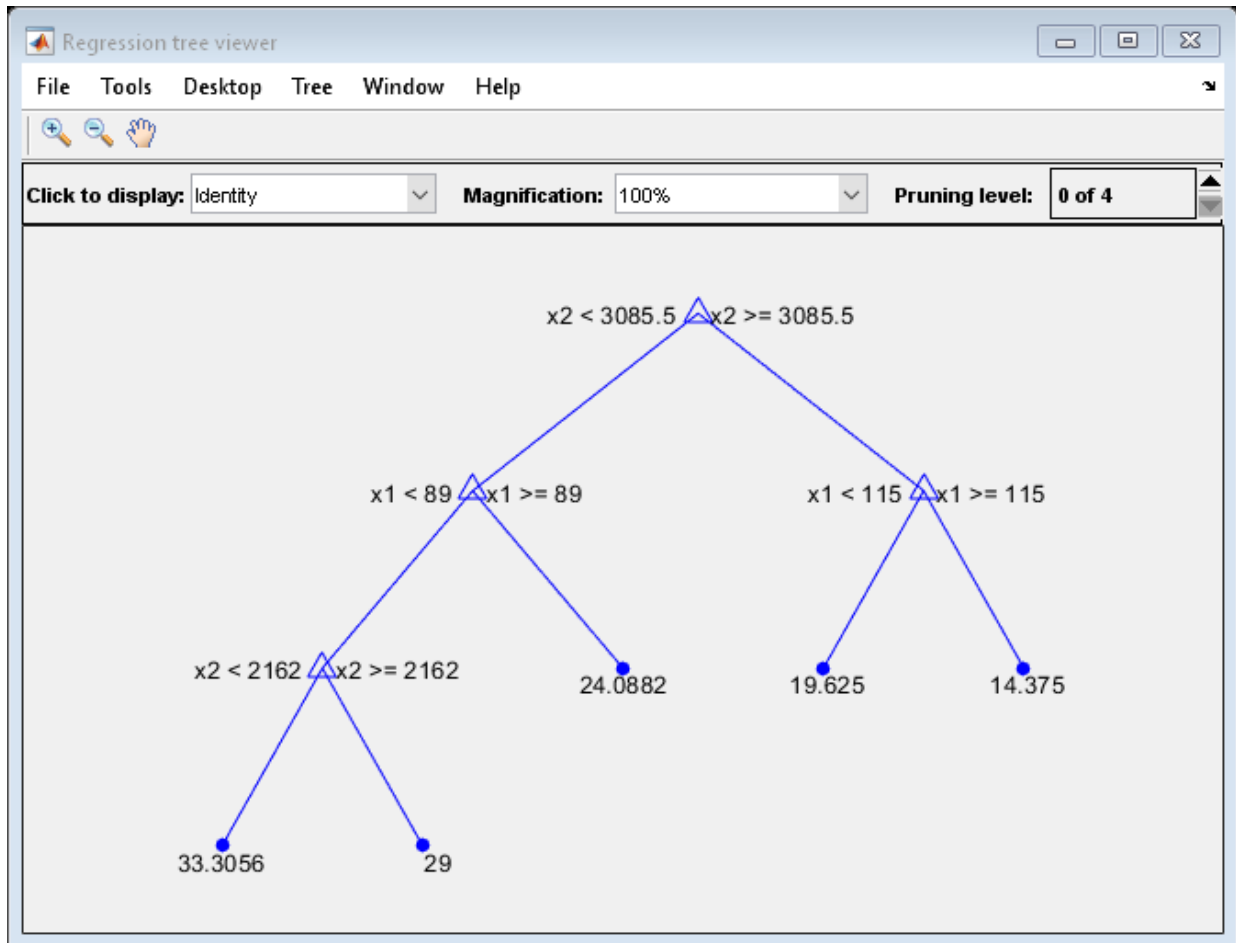
```
load carsmall % load the sample data, contains Horsepower, Weight, MPG
X = [Horsepower Weight];
rtree = fitrtree(X,MPG,'MinParent',30); % create classification tree
view(rtree) % text description
```

Decision tree for regression

```
1 if x2<3085.5 then node 2 elseif x2>=3085.5 then node 3 else 23.7181
2 if x1<89 then node 4 elseif x1>=89 then node 5 else 28.7931
3 if x1<115 then node 6 elseif x1>=115 then node 7 else 15.5417
```

```
4 if x2<2162 then node 8 elseif x2>=2162 then node 9 else 30.9375
5 fit = 24.0882
6 fit = 19.625
7 fit = 14.375
8 fit = 33.3056
9 fit = 29
```

```
view(rtree,'mode','graph') % graphic description
```



## See Also

`CompactClassificationTree.view` | `CompactRegressionTree.view` | `fitctree`  
| `fitrtree`

## Related Examples

- “Decision Trees” on page 19-2



## Growing Decision Trees

By default, `fitctree` and `fitrtree` use the standard CART algorithm [1] to create decision trees. That is, they perform the following steps:

- 1 Start with all input data, and examine all possible binary splits on every predictor.
- 2 Select a split with best optimization criterion.
  - A split might lead to a child node having too few observations (less than the `MinLeafSize` parameter). To avoid this, the software chooses a split that yields the best optimization criterion subject to the `MinLeafSize` constraint.
- 3 Impose the split.
- 4 Repeat recursively for the two child nodes.

The explanation requires two more items: description of the optimization criterion and stopping rule.

**Stopping rule:** Stop splitting when any of the following hold:

- The node is pure.
  - For classification, a node is pure if it contains only observations of one class.
  - For regression, a node is pure if the mean squared error (MSE) for the observed response in this node drops below the MSE for the observed response in the entire data multiplied by the tolerance on quadratic error per node (`QuadraticErrorTolerance` parameter).
- There are fewer than `MinParentSize` observations in this node.
- Any split imposed on this node produces children with fewer than `MinLeafSize` observations.
- The algorithm splits `MaxNumSplits` nodes.

**Optimization criterion:**

- Regression: mean-squared error (MSE). Choose a split to minimize the MSE of predictions compared to the training data.
- Classification: One of three measures, depending on the setting of the `SplitCriterion` name-value pair:
  - `'gdi'` (Gini's diversity index, the default)

- 'twoing'
- 'deviance'

For details, see `ClassificationTree` “Definitions” on page 32-578.

For alternative split predictor selection techniques, see “Choose Split Predictor Selection Technique” on page 19-19.

For a continuous predictor, a tree can split halfway between any two adjacent unique values found for this predictor. For a categorical predictor with  $L$  levels, a classification tree needs to consider  $2^{L-1}-1$  splits to find the optimal split. Alternatively, you can choose a heuristic algorithm to find a good split, as described in “Splitting Categorical Predictors” on page 19-42.

For dual-core systems and above, `fitctree` and `fitrtree` parallelize training decision trees using Intel® Threading Building Blocks (TBB). For details on Intel TBB, see <https://software.intel.com/en-us/intel-tbb>.

## References

- [1] Breiman, L., J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*. Boca Raton, FL: Chapman & Hall, 1984.

## See Also

`ClassificationTree` | `RegressionTree` | `fitctree` | `fitrtree`

## Related Examples

- “Decision Trees” on page 19-2
- “Splitting Categorical Predictors” on page 19-42

## Prediction Using Classification and Regression Trees

This example shows how to predict class labels or responses using trained classification and regression trees.

After creating a tree, you can easily predict responses for new data. Suppose  $X_{new}$  is new data that has the same number of columns as the original data  $X$ . To predict the classification or regression based on the tree ( $Mdl$ ) and the new data, enter

```
Ynew = predict(Mdl,Xnew)
```

For each row of data in  $X_{new}$ , `predict` runs through the decisions in  $Mdl$  and gives the resulting prediction in the corresponding element of  $Y_{new}$ . For more information on classification tree prediction, see the `CompactClassificationTree.predict`. For regression, see `CompactRegressionTree.predict`.

For example, find the predicted classification of a point at the mean of the `ionosphere` data.

```
load ionosphere
CMdl = fitctree(X,Y);
Ynew = predict(CMdl,mean(X))
```

```
Ynew =
    1x1 cell array
    {'g'}
```

Find the predicted MPG of a point at the mean of the `carsmall` data.

```
load carsmall
X = [Horsepower Weight];
RMdl = fitrtree(X,MPG);
Ynew = predict(RMdl,mean(X))
```

```
Ynew =
```

28.7931

### See Also

`ClassificationTree` | `CompactClassificationTree.predict` |  
`CompactRegressionTree.predict` | `RegressionTree` | `fitctree` | `fitrtree`

### Related Examples

- “Decision Trees” on page 19-2
- “Predict Out-of-Sample Responses of Subtrees” on page 19-13
- “Improving Classification Trees and Regression Trees” on page 19-17

## Predict Out-of-Sample Responses of Subtrees

This example shows how to predict out-of-sample responses of regression trees, and then plot the results.

Load the `carsmall` data set. Consider `Weight` as a predictor of the response `MPG`.

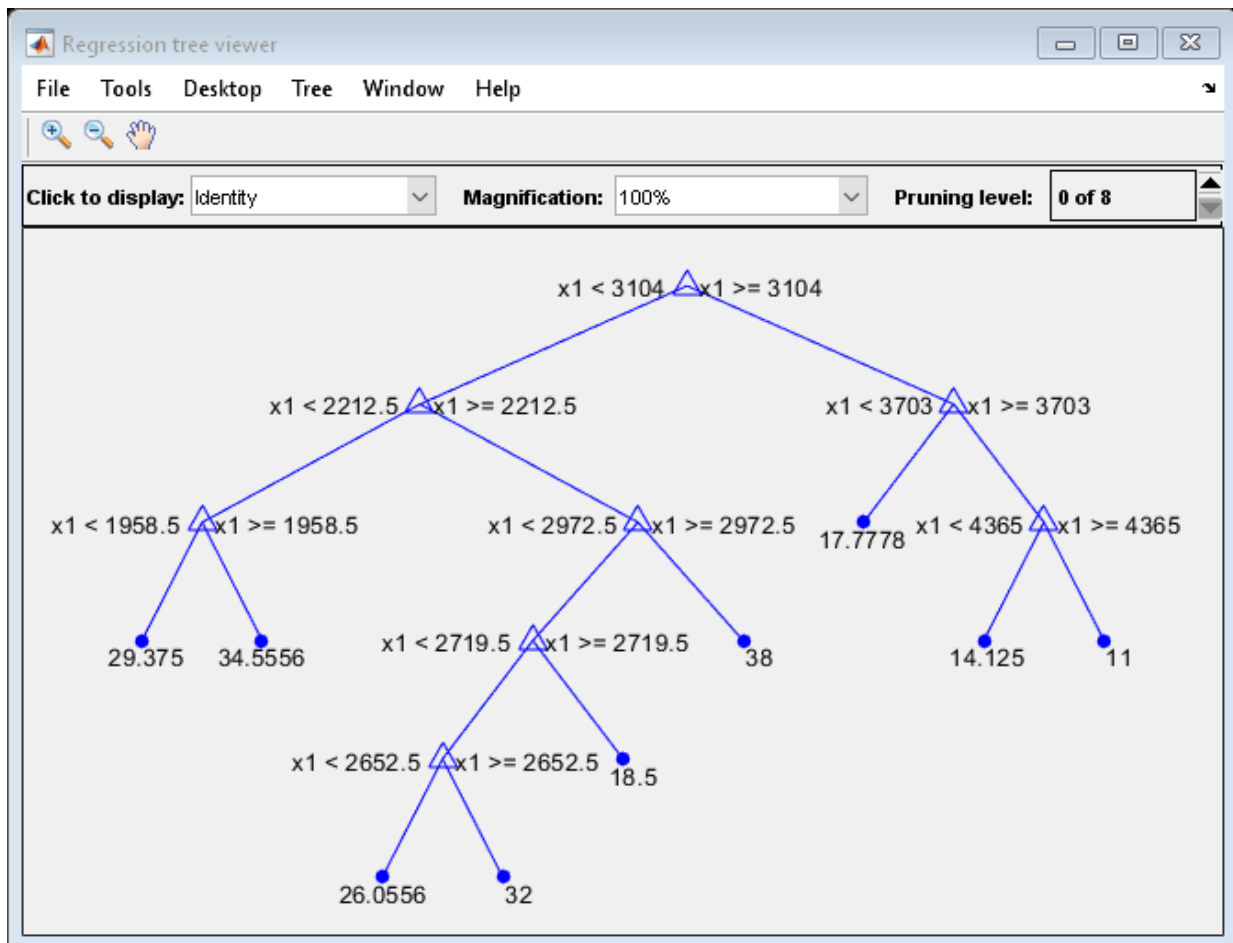
```
load carsmall
idxNaN = isnan(MPG + Weight);
X = Weight(~idxNaN);
Y = MPG(~idxNaN);
n = numel(X);
```

Partition the data into training (50%) and validation (50%) sets.

```
rng(1) % For reproducibility
idxTrn = false(n,1);
idxTrn(randsample(n,round(0.5*n))) = true; % Training set logical indices
idxVal = idxTrn == false; % Validation set logical indices
```

Grow a regression tree using the training observations.

```
Mdl = fitrtree(X(idxTrn),Y(idxTrn));
view(Mdl,'Mode','graph')
```



Compute fitted values of the validation observations for each of several subtrees.

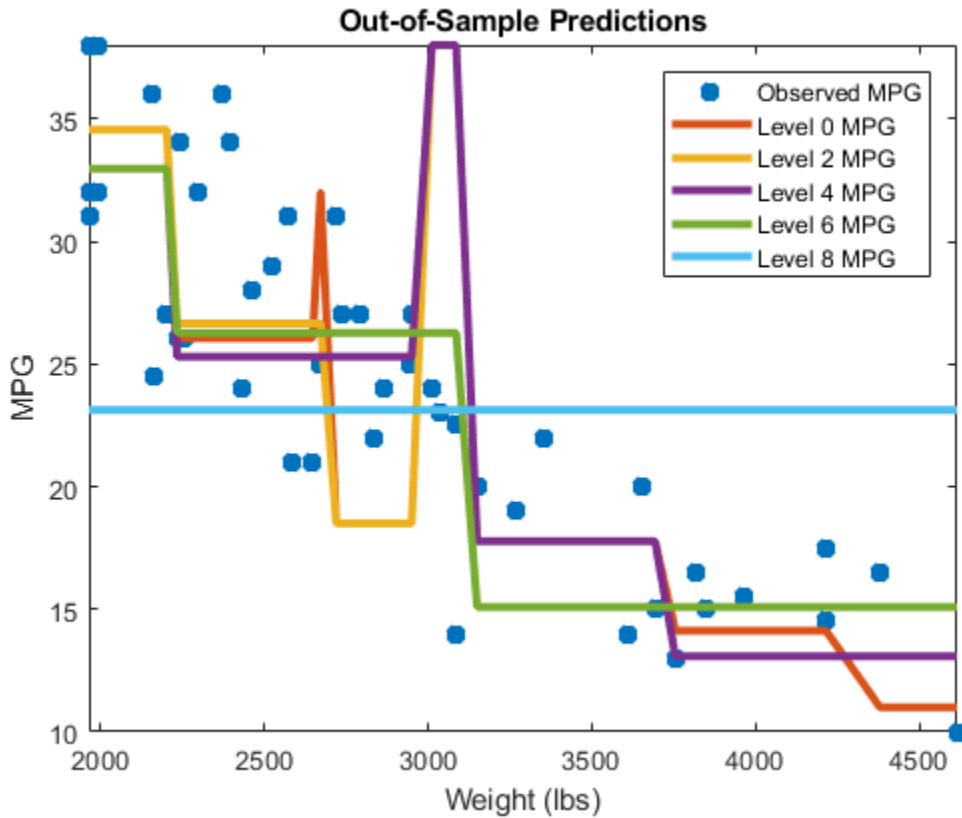
```
m = max(Mdl.PruneList);
pruneLevels = 0:2:m; % Pruning levels to consider
z = numel(pruneLevels);
Yfit = predict(Mdl,X(idxVal),'SubTrees',pruneLevels);
```

$Y_{fit}$  is an  $n$ -by-  $z$  matrix of fitted values in which the rows correspond to observations and the columns correspond to a subtree.

Plot  $Y_{fit}$  and  $Y$  against  $X$ .

```

figure;
sortDat = sortrows([X(idxVal) Y(idxVal) Yfit],1); % Sort all data with respect to X
plot(sortDat(:,1),sortDat(:,2),'*');
hold on;
plot(repmat(sortDat(:,1),1,size(Yfit,2)),sortDat(:,3:end));
lev = cellstr(num2str((pruneLevels)', 'Level %d MPG'));
legend(['Observed MPG'; lev])
title 'Out-of-Sample Predictions'
xlabel 'Weight (lbs)';
ylabel 'MPG';
h = findobj(gcf);
axis tight;
set(h(4:end), 'LineWidth', 3) % Widen all lines
    
```



The values of `yfit` for lower pruning levels tend to follow the data more closely than higher levels. Higher pruning levels tend to be flat for large  $x$  intervals.

### See Also

`CompactRegressionTree.predict` | `RegressionTree`

### Related Examples

- “Decision Trees” on page 19-2
- “Improving Classification Trees and Regression Trees” on page 19-17
- “Prediction Using Classification and Regression Trees” on page 19-11



## Improving Classification Trees and Regression Trees

You can tune trees by setting name-value pairs in `fitctree` and `fitrtree`. The remainder of this section describes how to determine the quality of a tree, how to decide which name-value pairs to set, and how to control the size of a tree:

### In this section...

“Examining Resubstitution Error” on page 19-17

“Cross Validation” on page 19-18

“Choose Split Predictor Selection Technique” on page 19-19

“Control Depth or “Leafiness”” on page 19-21

“Pruning” on page 19-26

### Examining Resubstitution Error

Resubstitution error is the difference between the response training data and the predictions the tree makes of the response based on the input training data. If the resubstitution error is high, you cannot expect the predictions of the tree to be good. However, having low resubstitution error does not guarantee good predictions for new data. Resubstitution error is often an overly optimistic estimate of the predictive error on new data.

#### Classification Tree Resubstitution Error

This example shows how to examine the resubstitution error of a classification tree.

Load Fisher's iris data.

```
load fisheriris
```

Train a default classification tree using the entire data set.

```
Mdl = fitctree(meas,species);
```

Examine the resubstitution error.

```
resuberror = resubLoss(Mdl)
```

```
resuberror =
```

```
0.0200
```

The tree classifies nearly all the Fisher iris data correctly.

## Cross Validation

To get a better sense of the predictive accuracy of your tree for new data, cross validate the tree. By default, cross validation splits the training data into 10 parts at random. It trains 10 new trees, each one on nine parts of the data. It then examines the predictive accuracy of each new tree on the data not included in training that tree. This method gives a good estimate of the predictive accuracy of the resulting tree, since it tests the new trees on new data.

### Cross Validate a Regression Tree

This example shows how to examine the resubstitution and cross-validation accuracy of a regression tree for predicting mileage based on the `carsmall` data.

Load the `carsmall` data set. Consider acceleration, displacement, horsepower, and weight as predictors of MPG.

```
load carsmall
X = [Acceleration Displacement Horsepower Weight];
```

Grow a regression tree using all of the observations.

```
rtree = fitrtree(X,MPG);
```

Compute the in-sample error.

```
resuberror = resubLoss(rtree)
```

```
resuberror =
```

```
4.7188
```

The resubstitution loss for a regression tree is the mean-squared error. The resulting value indicates that a typical predictive error for the tree is about the square root of 4.7, or a bit over 2.

Estimate the cross-validation MSE.

```
rng 'default';  
cvrtree = crossval(rtree);  
cvloss = kfoldLoss(cvrtree)
```

```
cvloss =  
  
    23.8065
```

The cross-validated loss is almost 25, meaning a typical predictive error for the tree on new data is about 5. This demonstrates that cross-validated loss is usually higher than simple resubstitution loss.

## Choose Split Predictor Selection Technique

The standard CART algorithm tends to select continuous predictors that have many levels. Sometimes, such a selection can be spurious and can also mask more important predictors that have fewer levels, such as categorical predictors. That is, the predictor-selection process at each node is biased. Also, standard CART tends to miss the important interactions between pairs of predictors and the response.

To mitigate selection bias and increase detection of important interactions, you can specify usage of the curvature or interaction tests using the 'PredictorSelection' name-value pair argument. Using the curvature or interaction test has the added advantage of producing better predictor importance estimates than standard CART.

This table summarizes the supported predictor-selection techniques.

Technique	'PredictorSelection' Value	Description	Training speed	When to specify
Standard CART [1]	Default	Selects the split predictor that maximizes the split-criterion gain over all possible splits of all predictors.	Baseline for comparison	Specify if any of these conditions are true: <ul style="list-style-type: none"> <li>• All predictors are continuous</li> <li>• Predictor importance is not the analysis goal</li> <li>• For boosting decision trees</li> </ul>
Curvature test [2][3]	'curvature'	Selects the split predictor that minimizes the $p$ -value of chi-square tests of independence between each predictor and the response.	Comparable to standard CART	Specify if any of these conditions are true: <ul style="list-style-type: none"> <li>• The predictor variables are heterogeneous</li> <li>• Predictor importance is an analysis goal</li> <li>• Enhance tree interpretation</li> </ul>

Technique	'PredictorSelection' Value	Description	Training speed	When to specify
Interaction test [3]	'interaction - curvature'	Chooses the split predictor that minimizes the $p$ -value of chi-square tests of independence between each predictor and the response (that is, conducts curvature tests), and that minimizes the $p$ -value of a chi-square test of independence between each pair of predictors and response.	Slower than standard CART, particularly when data set contains many predictor variables.	Specify if any of these conditions are true: <ul style="list-style-type: none"> <li>• The predictor variables are heterogeneous</li> <li>• You suspect associations between pairs of predictors and the response</li> <li>• Predictor importance is an analysis goal</li> <li>• Enhance tree interpretation</li> </ul>

For more details on predictor selection techniques:

- For classification trees, see `PredictorSelection` and “Node Splitting Rules” on page 32-2174.
- For regression trees, see `PredictorSelection` and “Node Splitting Rules” on page 32-2174.

## Control Depth or “Leafiness”

When you grow a decision tree, consider its simplicity and predictive power. A deep tree with many leaves is usually highly accurate on the training data. However, the tree is not guaranteed to show a comparable accuracy on an independent test set. A leafy tree tends to overtrain (or overfit), and its test accuracy is often far less than its training (resubstitution) accuracy. In contrast, a shallow tree does not attain high training accuracy. But a shallow tree can be more robust — its training accuracy could be close to that of a representative test set. Also, a shallow tree is easy to interpret. If you do not have enough data for training and test, estimate tree accuracy by cross validation.

`fitctree` and `fitrtree` have three name-value pair arguments that control the depth of resulting decision trees:

- `MaxNumSplits` — The maximal number of branch node splits is `MaxNumSplits` per tree. Set a large value for `MaxNumSplits` to get a deep tree. The default is  $\text{size}(X, 1) - 1$ .
- `MinLeafSize` — Each leaf has at least `MinLeafSize` observations. Set small values of `MinLeafSize` to get deep trees. The default is 1.
- `MinParentSize` — Each branch node in the tree has at least `MinParentSize` observations. Set small values of `MinParentSize` to get deep trees. The default is 10.

If you specify `MinParentSize` and `MinLeafSize`, the learner uses the setting that yields trees with larger leaves (i.e., shallower trees):

```
MinParent = max(MinParentSize, 2*MinLeafSize)
```

If you supply `MaxNumSplits`, the software splits a tree until one of the three splitting criteria is satisfied.

For an alternative method of controlling the tree depth, see “Pruning” on page 19-26.

### Select Appropriate Tree Depth

This example shows how to control the depth of a decision tree, and how to choose an appropriate depth.

Load the `ionosphere` data.

```
load ionosphere
```

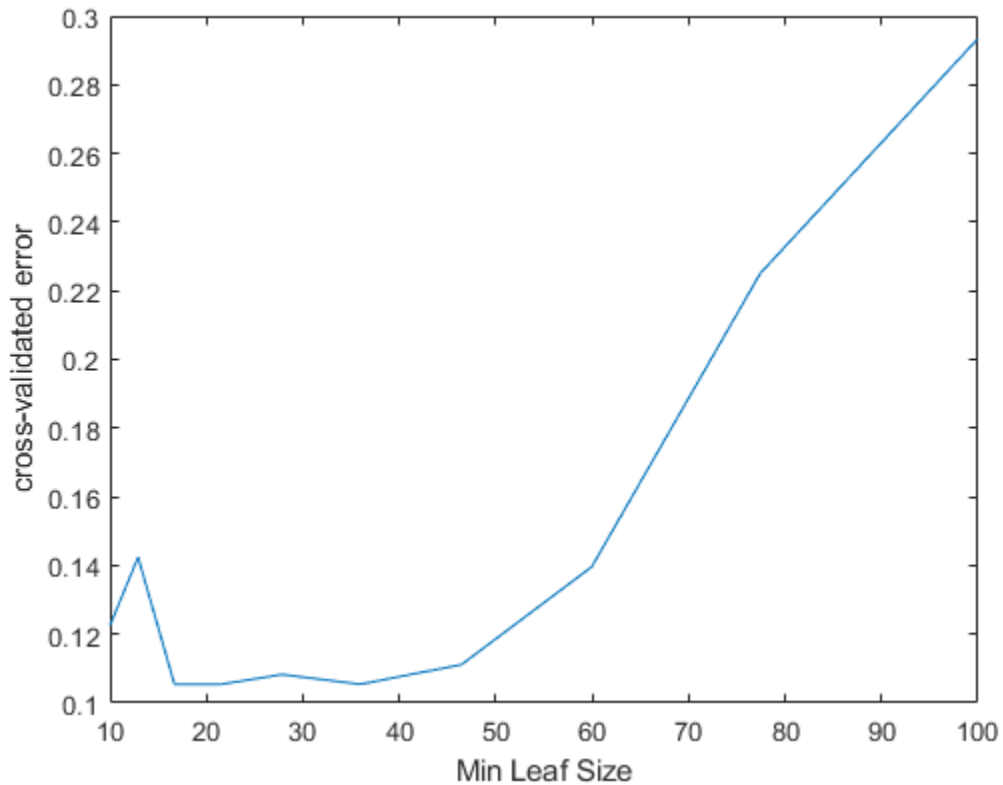
Generate an exponentially spaced set of values from 10 through 100 that represent the minimum number of observations per leaf node.

```
leafs = logspace(1,2,10);
```

Create cross-validated classification trees for the `ionosphere` data. Specify to grow each tree using a minimum leaf size in `leafs`.

```
rng('default')  
N = numel(leafs);  
err = zeros(N,1);  
for n=1:N
```

```
t = fitctree(X,Y,'CrossVal','On',...  
            'MinLeafSize',leafs(n));  
err(n) = kfoldLoss(t);  
end  
plot(leafs,err);  
xlabel('Min Leaf Size');  
ylabel('cross-validated error');
```

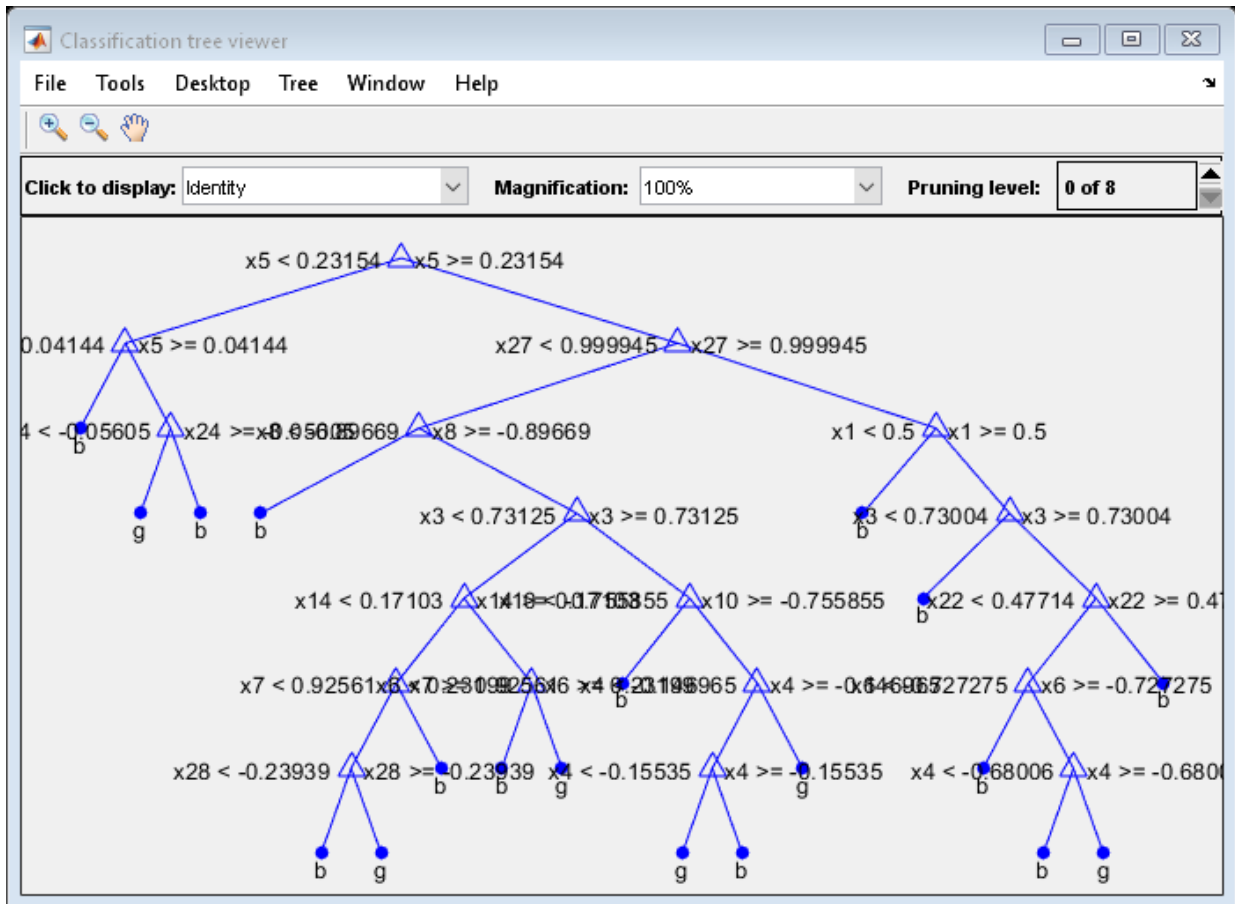


The best leaf size is between about 20 and 50 observations per leaf.

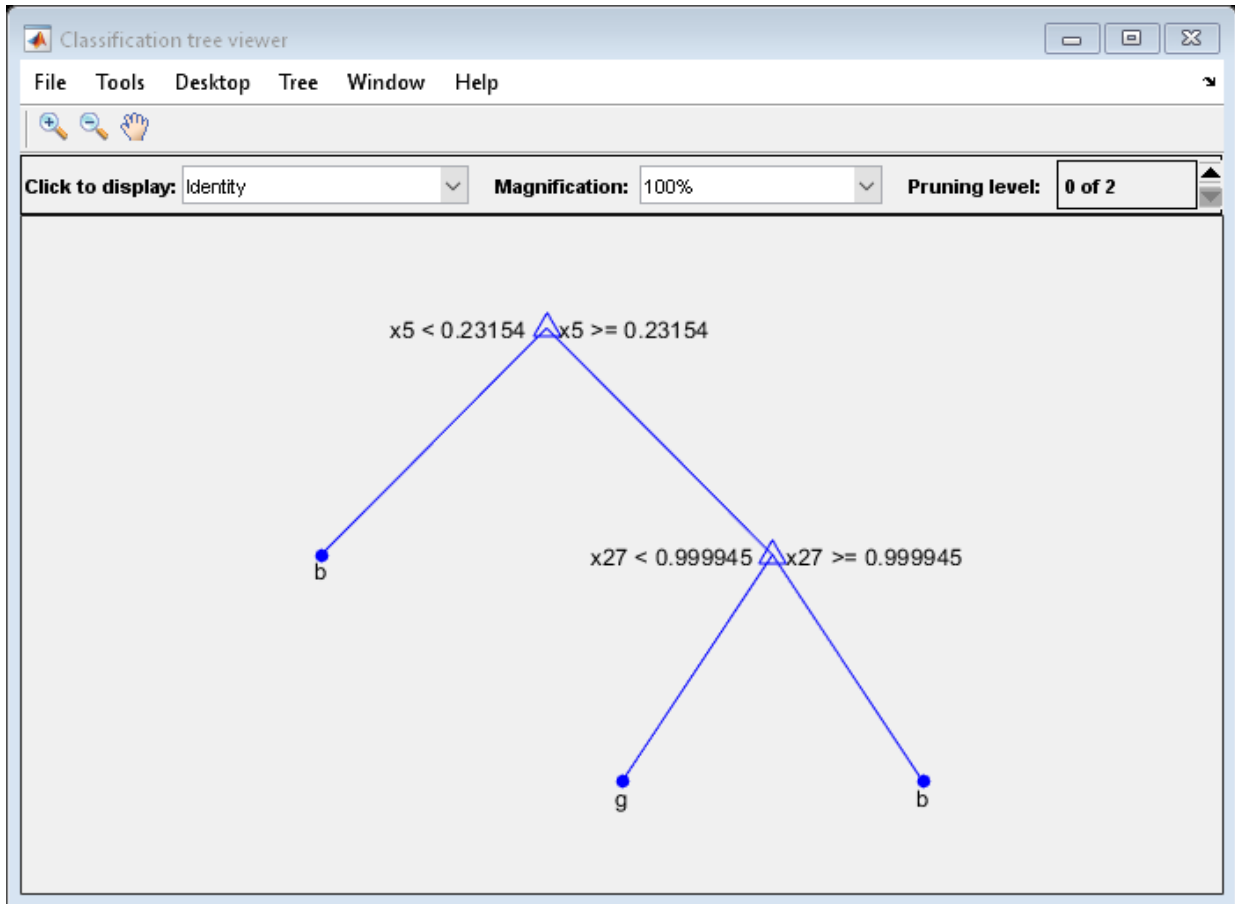
Compare the near-optimal tree with at least 40 observations per leaf with the default tree, which uses 10 observations per parent node and 1 observation per leaf.

```
DefaultTree = fitctree(X,Y);
view(DefaultTree,'Mode','Graph')
```

```
OptimalTree = fitctree(X,Y,'MinLeafSize',40);
view(OptimalTree,'mode','graph')
```







```

resubOpt = resubLoss(OptimalTree);
lossOpt = kfoldLoss(crossval(OptimalTree));
resubDefault = resubLoss(DefaultTree);
lossDefault = kfoldLoss(crossval(DefaultTree));
resubOpt, resubDefault, lossOpt, lossDefault

```

```

resubOpt =
    0.0883

```

```
resubDefault =  
    0.0114  
  
lossOpt =  
    0.1054  
  
lossDefault =  
    0.1111
```

The near-optimal tree is much smaller and gives a much higher resubstitution error. Yet, it gives similar accuracy for cross-validated data.

## Pruning

Pruning optimizes tree depth (leafiness) by merging leaves on the same tree branch. “Control Depth or “Leafiness”” on page 19-21 describes one method for selecting the optimal depth for a tree. Unlike in that section, you do not need to grow a new tree for every node size. Instead, grow a deep tree, and prune it to the level you choose.

Prune a tree at the command line using the `prune` method (classification) or `prune` method (regression). Alternatively, prune a tree interactively with the tree viewer:

```
view(tree, 'mode', 'graph')
```

To prune a tree, the tree must contain a pruning sequence. By default, both `fitctree` and `fitrtree` calculate a pruning sequence for a tree during construction. If you construct a tree with the 'Prune' name-value pair set to 'off', or if you prune a tree to a smaller level, the tree does not contain the full pruning sequence. Generate the full pruning sequence with the `prune` method (classification) or `prune` method (regression).

### Prune a Classification Tree

This example creates a classification tree for the `ionosphere` data, and prunes it to a good level.

Load the `ionosphere` data:

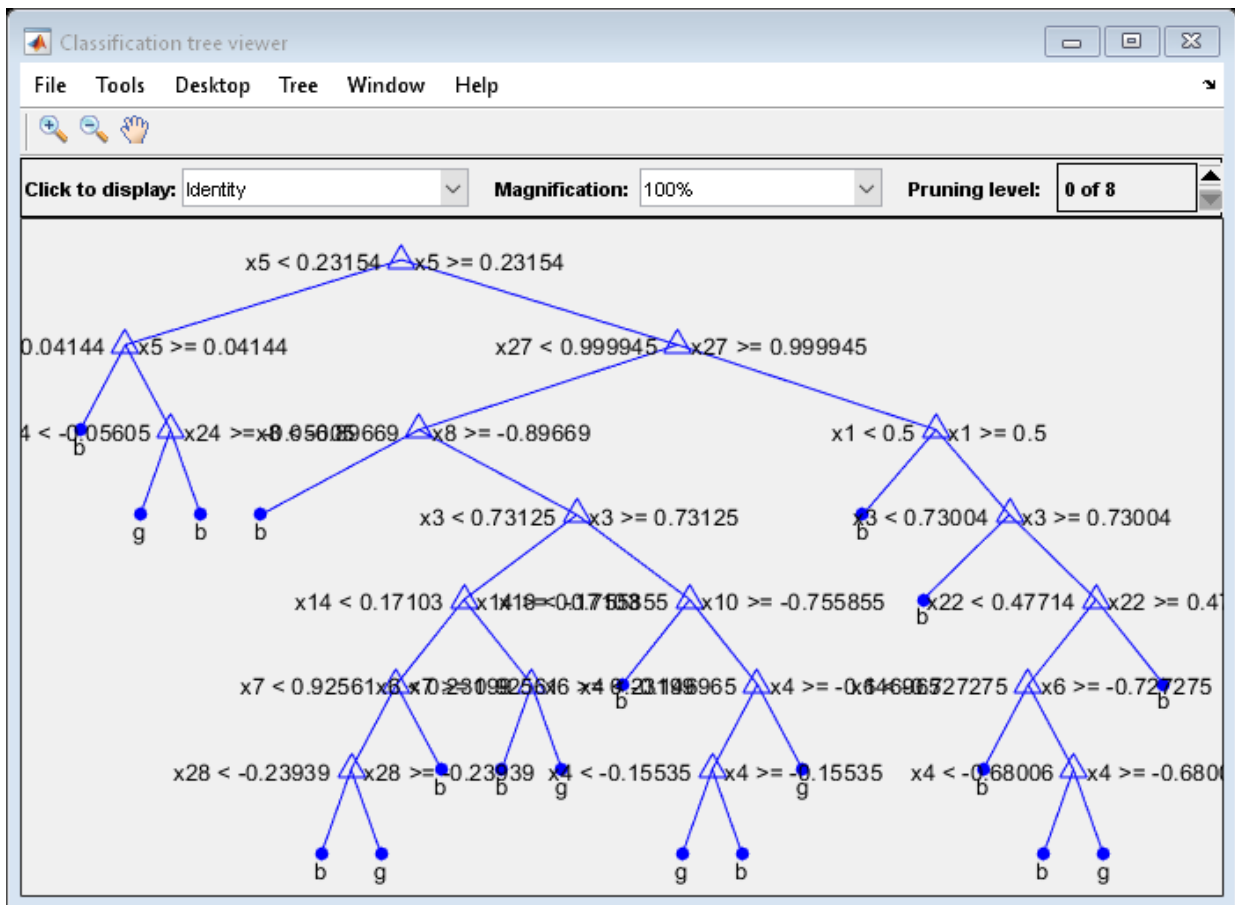
```
load ionosphere
```

Construct a default classification tree for the data:

```
tree = fitctree(X,Y);
```

View the tree in the interactive viewer:

```
view(tree, 'Mode', 'Graph')
```



Find the optimal pruning level by minimizing cross-validated loss:

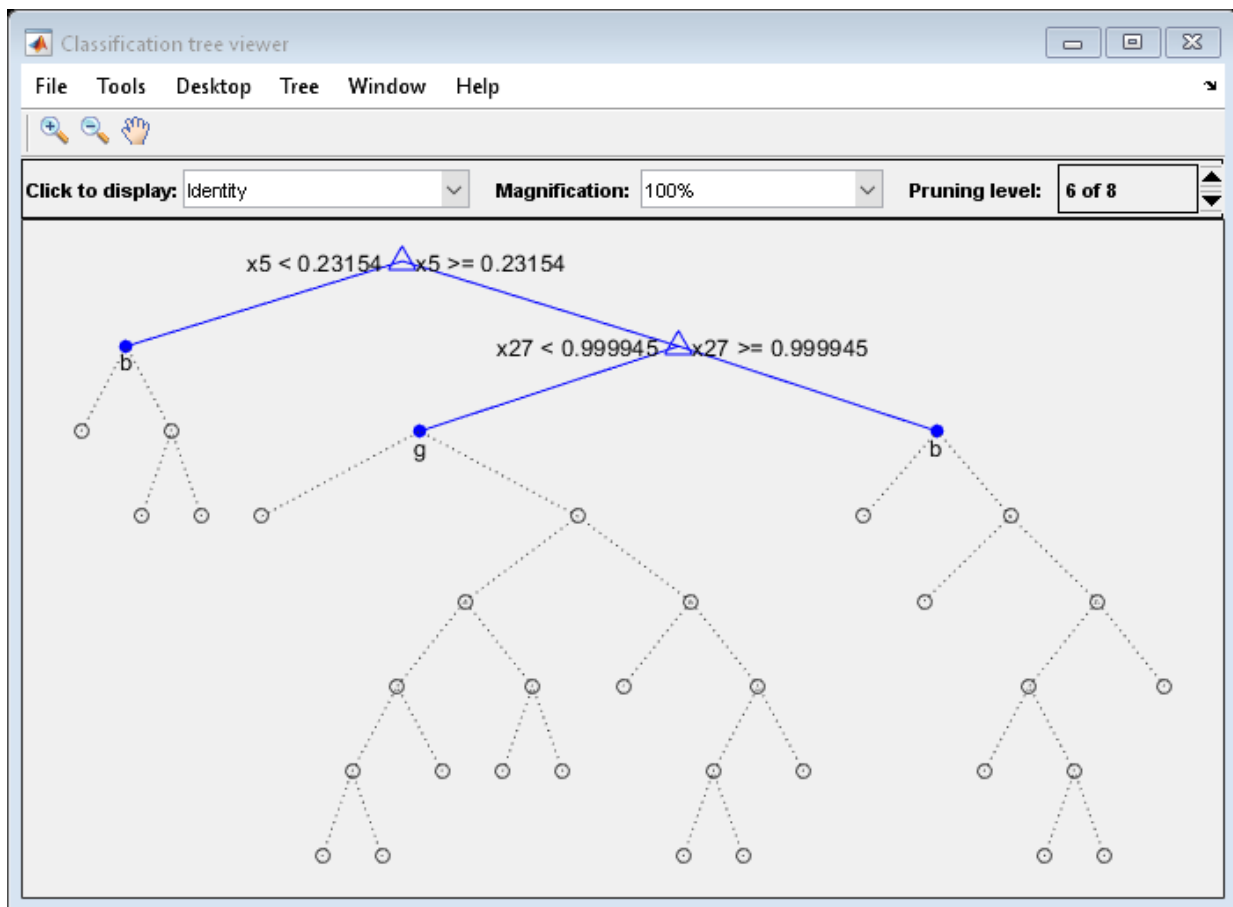
```
[~,~,~,bestlevel] = cvLoss(tree,...
    'SubTrees','All','TreeSize','min')
```

```
bestlevel =
```

```
6
```

Prune the tree to level 6:

```
view(tree,'Mode','Graph','Prune',6)
```



Alternatively, use the interactive window to prune the tree.

The pruned tree is the same as the near-optimal tree in the "Select Appropriate Tree Depth" example.

Set 'TreeSize' to 'SE' (default) to find the maximal pruning level for which the tree error does not exceed the error from the best level plus one standard deviation:

```
[~,~,~,bestlevel] = cvLoss(tree, 'SubTrees', 'All')
```

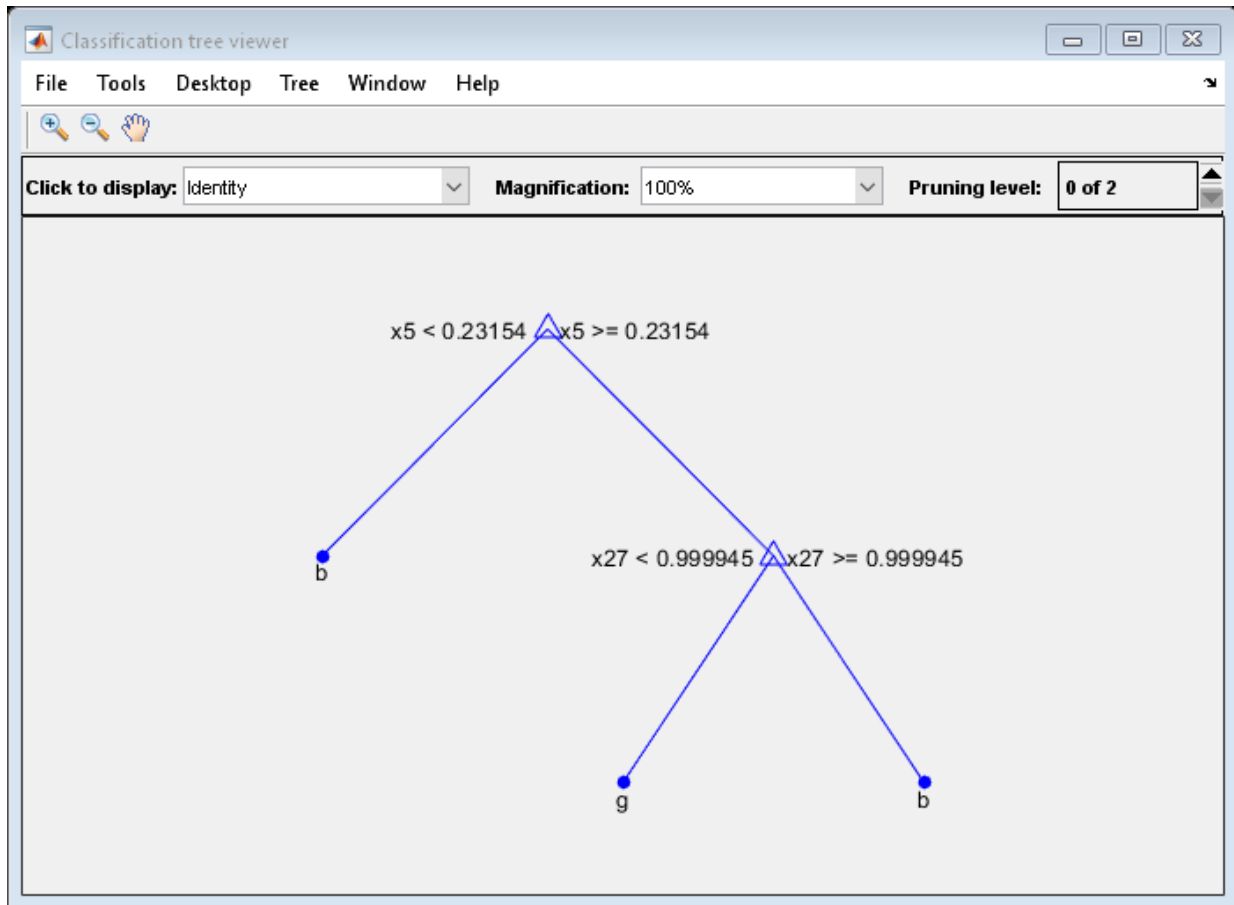
```
bestlevel =
```

```
6
```

In this case the level is the same for either setting of 'TreeSize'.

Prune the tree to use it for other purposes:

```
tree = prune(tree, 'Level', 6);  
view(tree, 'Mode', 'Graph')
```



## References

- [1] Breiman, L., J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*. Boca Raton, FL: Chapman & Hall, 1984.
- [2] Loh, W.Y. and Y.S. Shih. "Split Selection Methods for Classification Trees." *Statistica Sinica*, Vol. 7, 1997, pp. 815–840.
- [3] Loh, W.Y. "Regression Trees with Unbiased Variable Selection and Interaction Detection." *Statistica Sinica*, Vol. 12, 2002, pp. 361–386.

## See Also

`ClassificationTree` | `ClassificationTree.prune` |  
`CompactClassificationTree.predict` | `CompactRegressionTree.predict` |  
`RegressionTree` | `RegressionTree.prune` | `fitctree` | `fitrtree`

## Related Examples

- “Decision Trees” on page 19-2
- “Prediction Using Classification and Regression Trees” on page 19-11
- “Predict Out-of-Sample Responses of Subtrees” on page 19-13

## Alternative: `classregtree`

The `ClassificationTree` and `RegressionTree` classes were released in MATLAB R2011a. Previously, you represented both classification trees and regression trees with a `classregtree` object. The new classes provide all the functionality of the `classregtree` class, and are more convenient when used with “Framework for Ensemble Learning” on page 18-41.

Statistics and Machine Learning Toolbox software maintains `classregtree` for backward compatibility. This function will be removed in a future release.

### Train Classification Trees Using `classregtree`

This example uses Fisher's iris data in `fisheriris.mat` to create a classification tree for predicting species using measurements of sepal length, sepal width, petal length, and petal width as predictors. Here, the predictors are continuous and the response is categorical.

Load the data and use the `classregtree` constructor of the `classregtree` class to create the classification tree.

```
load fisheriris

t = classregtree(meas, species, ...
                'Names', {'SL' 'SW' 'PL' 'PW'})
```

Warning: `classregtree` will be removed in a future release. Use `fitctree` or `fitrtree` instead.

```
t =

Decision tree for classification
1  if PL<2.45 then node 2 elseif PL>=2.45 then node 3 else setosa
2  class = setosa
3  if PW<1.75 then node 4 elseif PW>=1.75 then node 5 else versicolor
4  if PL<4.95 then node 6 elseif PL>=4.95 then node 7 else versicolor
5  class = virginica
6  if PW<1.65 then node 8 elseif PW>=1.65 then node 9 else versicolor
7  class = virginica
8  class = versicolor
9  class = virginica
```



`t` is a `classregtree` object and can be operated on with any class method.

Use the `type` method of the `classregtree` class to show the type of the tree.

```
treetype = type(t)
```

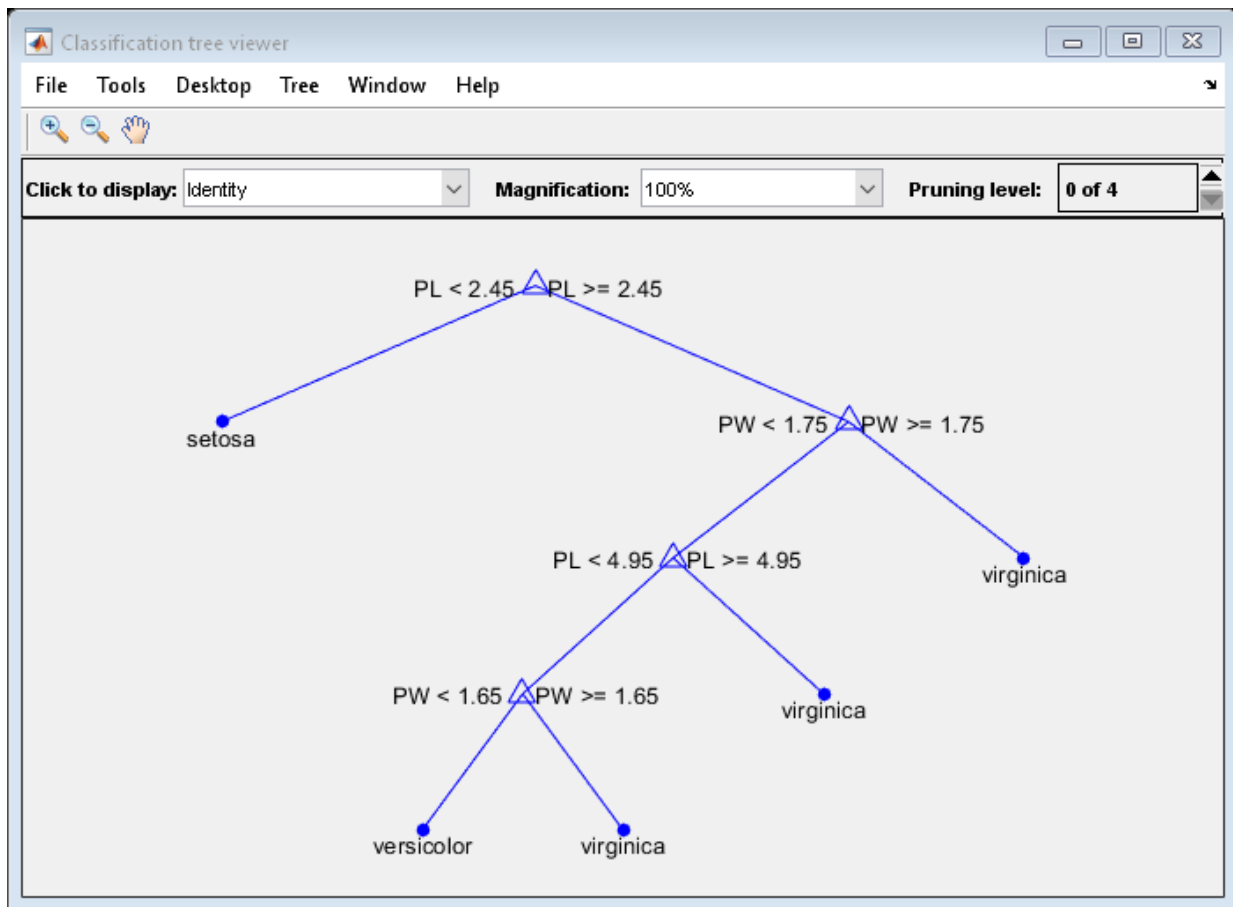
```
treetype =
```

```
    'classification'
```

`classregtree` creates a classification tree because `species` is a cell array of character vectors, and the response is assumed to be categorical.

To view the tree, use the `view` method of the `classregtree` class.

```
view(t)
```



The tree predicts the response values at the circular leaf nodes based on a series of questions about the iris at the triangular branching nodes. A `true` answer to any question follows the branch to the left. A `false` follows the branch to the right.

The tree does not use sepal measurements for predicting species. These can go unmeasured in new data, and you can enter them as `NaN` values for predictions. For example, use the tree to predict the species of an iris with petal length 4.8 and petal width 1.6.

```
predicted = t([NaN NaN 4.8 1.6])
```

Warning: classregtree will be removed in a future release. Use the predict method of an object returned by fitctree or fitrtree instead.

```
predicted =
    1x1 cell array
    {'versicolor'}
```

The object allows for functional evaluation, of the form  $t(X)$ . This is a shorthand way of calling the eval method of the classregtree class. The predicted species is the left leaf node at the bottom of the tree in the previous view.

You can use a variety of methods of the classregtree class, such as cutvar and cuttype to get more information about the split at node 6 that makes the final distinction between versicolor and virginica.

```
var6 = cutvar(t,6) % What variable determines the split?
type6 = cuttype(t,6) % What type of split is it?
```

```
var6 =
    1x1 cell array
    {'PW'}
```

```
type6 =
    1x1 cell array
    {'continuous'}
```

Classification trees fit the original (training) data well, but can do a poor job of classifying new values. Lower branches, especially, can be strongly affected by outliers. A simpler tree often avoids overfitting. You can use the prune method of the classregtree class to find the next largest tree from an optimal pruning sequence.

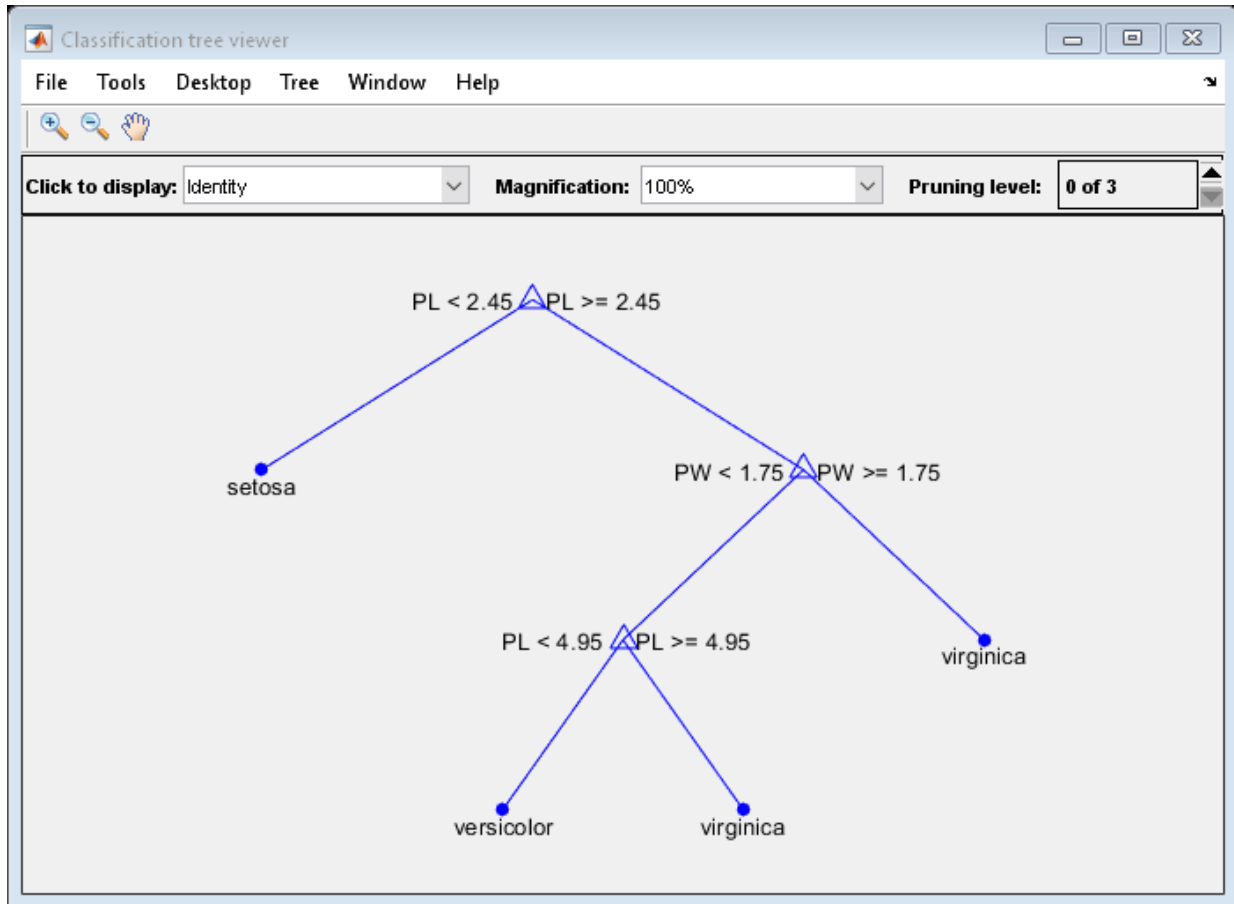
```
pruned = prune(t, 'Level', 1)
view(pruned)
```

Warning: `classregtree` will be removed in a future release. Use the `prune` method of an object returned by `fitctree` or `fitrtree` instead.

```
pruned =
```

```
Decision tree for classification
```

```
1  if PL<2.45 then node 2 elseif PL>=2.45 then node 3 else setosa
2  class = setosa
3  if PW<1.75 then node 4 elseif PW>=1.75 then node 5 else versicolor
4  if PL<4.95 then node 6 elseif PL>=4.95 then node 7 else versicolor
5  class = virginica
6  class = versicolor
7  class = virginica
```



To find the best classification tree, employing the techniques of resubstitution and cross validation, use the `test` method of the `classregtree` class.

## Train Regression Trees Using `classregtree`

This example uses the data on cars in `carsmall.mat` to create a regression tree for predicting mileage using measurements of weight and the number of cylinders as predictors. Here, one predictor (weight) is continuous and the other (cylinders) is categorical. The response (mileage) is continuous.

Load the data and use the `classregtree` constructor of the `classregtree` class to create the regression tree:

```
load carsmall
```

```
t = classregtree([Weight, Cylinders],MPG,...  
                'Categorical',2,'MinParent',20,...  
                'Names',{'W','C'})
```

Warning: `classregtree` will be removed in a future release. Use `fitctree` or `fitrtree` instead.

```
t =
```

```
Decision tree for regression
```

```
1  if W<3085.5 then node 2 elseif W>=3085.5 then node 3 else 23.7181  
2  if W<2371 then node 4 elseif W>=2371 then node 5 else 28.7931  
3  if C=8 then node 6 elseif C in {4 6} then node 7 else 15.5417  
4  if W<2162 then node 8 elseif W>=2162 then node 9 else 32.0741  
5  if C=6 then node 10 elseif C=4 then node 11 else 25.9355  
6  if W<4381 then node 12 elseif W>=4381 then node 13 else 14.2963  
7  fit = 19.2778  
8  fit = 33.3056  
9  fit = 29.6111  
10 fit = 23.25  
11 if W<2827.5 then node 14 elseif W>=2827.5 then node 15 else 27.2143  
12 if W<3533.5 then node 16 elseif W>=3533.5 then node 17 else 14.8696  
13 fit = 11  
14 fit = 27.6389  
15 fit = 24.6667  
16 fit = 16.6  
17 fit = 14.3889
```

`t` is a `classregtree` object and can be operated on with any of the methods of the class.

Use the `type` method of the `classregtree` class to show the type of the tree:

```
treetype = type(t)
```

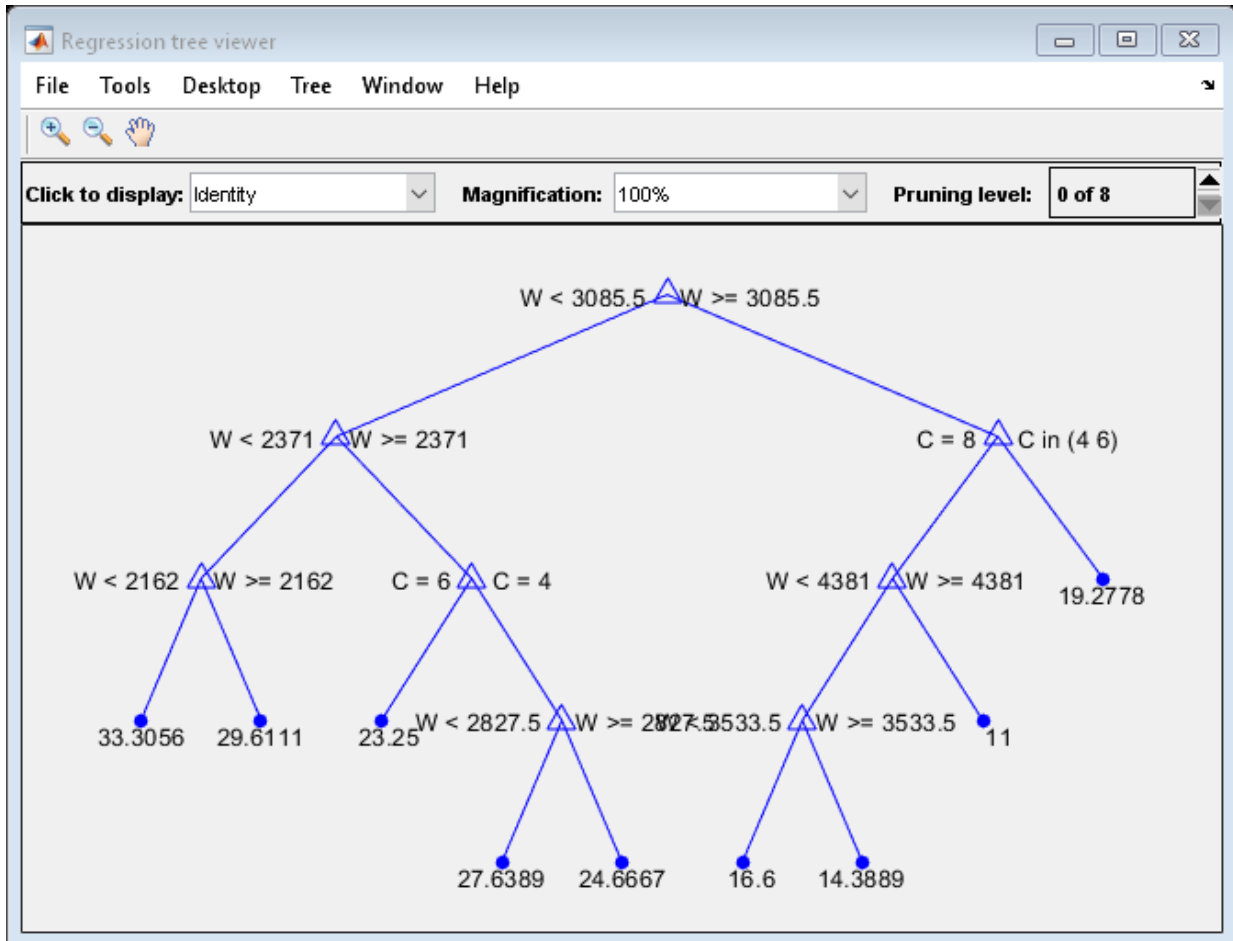
```
treetype =
```

```
    'regression'
```

`classregtree` creates a regression tree because MPG is a numerical vector, and the response is assumed to be continuous.

To view the tree, use the `view` method of the `classregtree` class:

```
view(t)
```



The tree predicts the response values at the circular leaf nodes based on a series of questions about the car at the triangular branching nodes. A `true` answer to any question follows the branch to the left; a `false` follows the branch to the right.

Use the tree to predict the mileage for a 2000-pound car with either 4, 6, or 8 cylinders:

```
mileage2K = t([2000 4; 2000 6; 2000 8])
```

```
Warning: classregtree will be removed in a future release. Use the predict
method of an object returned by fitctree or fitrtree instead.
```

```
mileage2K =
    33.3056
    33.3056
    33.3056
```

The object allows for functional evaluation, of the form `t(X)`. This is a shorthand way of calling the `eval` method of the `classregtree` class.

The predicted responses computed above are all the same. This is because they follow a series of splits in the tree that depend only on weight, terminating at the leftmost leaf node in the view above. A 4000-pound car, following the right branch from the top of the tree, leads to different predicted responses:

```
mileage4K = t([4000 4; 4000 6; 4000 8])
```

```
Warning: classregtree will be removed in a future release. Use the predict
method of an object returned by fitctree or fitrtree instead.
```

```
mileage4K =
    19.2778
    19.2778
    14.3889
```

You can use a variety of other methods of the `classregtree` class, such as `cutvar`, `cuttype`, and `cutcategories`, to get more information about the split at node 3 that distinguishes the 8-cylinder car:

```
var3 = cutvar(t,3)           % What variable determines the split?
type3 = cuttype(t,3)        % What type of split is it?
c = cutcategories(t,3);     % Which classes are sent to the left
                             % child node, and which to the right?

leftChildNode = c{1}
rightChildNode = c{2}
```



```
var3 =  
    1x1 cell array  
    {'C'}  
  
type3 =  
    1x1 cell array  
    {'categorical'}  
  
leftChildNode =  
    8  
  
rightChildNode =  
    4    6
```

Regression trees fit the original (training) data well, but may do a poor job of predicting new values. Lower branches, especially, may be strongly affected by outliers. A simpler tree often avoids overfitting. To find the best regression tree, employing the techniques of resubstitution and cross validation, use the `test` method of the `classregtree` class.

## See Also

`ClassificationTree` | `CompactClassificationTree.predict` |  
`CompactRegressionTree.predict` | `RegressionTree` | `fitctree` | `fitrtree`

## Related Examples

- “Decision Trees” on page 19-2

## Splitting Categorical Predictors

### In this section...

“Challenges in Splitting Multilevel Predictors” on page 19-42

“Pull Left By Purity” on page 19-43

“Principal Component-Based Partitioning” on page 19-43

“One Versus All By Class” on page 19-43

### Challenges in Splitting Multilevel Predictors

When growing a classification tree, finding an optimal binary split for a categorical predictor with many levels is significantly more computationally challenging than finding a split for a continuous predictor. For a continuous predictor, a tree can split halfway between any two adjacent unique values of this predictor.

In contrast, to find an exact optimal binary split for a categorical predictor with  $L$  levels, a classification tree needs to consider  $2^{L-1}-1$  splits. To obtain this formula, observe that you can assign  $L$  distinct values to the left and right nodes in  $2^L$  ways. Two out of these  $2^L$  configurations leave either the left or right node empty, and therefore should be discarded. Now, divide by 2 because left and right can be swapped.

For regression and binary classification problems, with  $K = 2$  response classes, there is a computational shortcut [1]. The tree can order the categories by mean response (for regression) or class probability for one of the classes (for classification). Then, the optimal split is one of the  $L - 1$  splits for the ordered list. When  $K = 2$ , `fitctree` always uses an exact search.

Therefore, computational challenges really only arise when growing classification trees for data with  $K \geq 3$  classes. To reduce computation, there are several heuristic algorithms for finding a good split. When using `fitctree` to grow a classification tree, you can choose an algorithm for splitting categorical predictors using the `AlgorithmForCategorical` name-value pair argument. You can also set this algorithm when creating a classification `template`.

If you do not specify an algorithm, `fitctree` splits categorical predictors using the exact search algorithm, provided the predictor has at most `MaxNumCategories` levels (the default is 10 levels, and, depending on your platform, you cannot perform an exact search

on categorical predictors with more than 32 or 64 levels). Otherwise, `fitctree` chooses a good inexact search algorithm based on the number of classes and levels.

The available heuristic algorithms are: pull left by purity, a principal component-based partitioning, and one versus all by class.

## Pull Left By Purity

This algorithm starts with all  $L$  categorical levels on the right branch. Inspect the  $K$  categories that have the largest class probabilities for each class. Move the category with the maximum value of the split criterion to the left branch. Continue moving categories from right to left, recording the split criterion at each move, until the right child has only one category remaining. Out of this sequence, the chosen split is the one that maximizes the split criterion.

Select this pull left by purity algorithm by using the `'AlgorithmForCategorical', 'PullLeft'` name-value pair in `fitctree`.

## Principal Component-Based Partitioning

This algorithm was developed by Coppersmith, Hong, and Hosking [2]. It finds a close-to-optimal binary partition of the  $L$  predictor levels by searching for a separating hyperplane that is perpendicular to the first principal component of the weighted covariance matrix of the centered class probability matrix.

The algorithm assigns a score to each of the  $L$  categories, computed as the inner product between the found principal component and the vector of class probabilities for that category. Then, the chosen split is the one of the  $L - 1$  splits of the scores that maximizes the split criterion.

Select this principal component-based partitioning by using the `'AlgorithmForCategorical', 'PCA'` name-value pair in `fitctree`.

## One Versus All By Class

This algorithm starts with all  $L$  categorical levels on the right branch. For each of the  $K$  classes, order the categories based on their probability for that class.

For the first class, move each category to the left branch in order, recording the split criterion at each move. Repeat for the remaining classes. Out of this sequence, the chosen split is the one that maximizes the split criterion.

Select this one versus all by class algorithm by using the 'AlgorithmForCategorical', 'OVAbyClass' name-value pair in `fitctree`.

## References

- [1] Breiman, L., J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*. Chapman & Hall, Boca Raton, 1984.
- [2] Coppersmith, D., S. J. Hong, and J. R. M. Hosking. “Partitioning Nominal Attributes in Decision Trees.” *Data Mining and Knowledge Discovery*, Vol. 3, 1999, pp. 197–217.

## See Also

`fitctree` | `fitrtree` | `template`

## Related Examples

- “Decision Trees” on page 19-2
- “Growing Decision Trees” on page 19-9

# Discriminant Analysis

---

- “Discriminant Analysis Classification” on page 20-2
- “Creating Discriminant Analysis Model” on page 20-5
- “Prediction Using Discriminant Analysis Models” on page 20-8
- “Create and Visualize Discriminant Analysis Classifier” on page 20-11
- “Improving Discriminant Analysis Models” on page 20-20
- “Regularize a Discriminant Analysis Classifier” on page 20-28
- “Examine the Gaussian Mixture Assumption” on page 20-36

## Discriminant Analysis Classification

Discriminant analysis is a classification method. It assumes that different classes generate data based on different Gaussian distributions.

- To train (create) a classifier, the fitting function estimates the parameters of a Gaussian distribution for each class (see “Creating Discriminant Analysis Model” on page 20-5).
- To predict the classes of new data, the trained classifier finds the class with the smallest misclassification cost (see “Prediction Using Discriminant Analysis Models” on page 20-8).

Linear discriminant analysis is also known as the Fisher discriminant, named for its inventor, Sir R. A. Fisher [1].

### Create Discriminant Analysis Classifiers

This example shows how to train a basic discriminant analysis classifier to classify irises in Fisher's iris data.

Load the data.

```
load fisheriris
```

Create a default (linear) discriminant analysis classifier.

```
MdlLinear = fitcdiscr(meas,species);
```

To visualize the classification boundaries of a 2-D linear classification of the data, see “Create and Visualize Discriminant Analysis Classifier” on page 20-11.

Classify an iris with average measurements.

```
meanmeas = mean(meas);  
meanclass = predict(MdlLinear,meanmeas)
```

```
meanclass =
```

```
    1x1 cell array  
  
    {'versicolor'}
```

Create a quadratic classifier.

```
MdlQuadratic = fitcdiscr(meas,species,'DiscrimType','quadratic');
```

To visualize the classification boundaries of a 2-D quadratic classification of the data, see “Create and Visualize Discriminant Analysis Classifier” on page 20-11.

Classify an iris with average measurements using the quadratic classifier.

```
meanclass2 = predict(MdlQuadratic,meanmeas)
```

```
meanclass2 =  
  
    1x1 cell array  
  
    {'versicolor'}
```

## References

- [1] Fisher, R. A. *The Use of Multiple Measurements in Taxonomic Problems*. Annals of Eugenics, Vol. 7, pp. 179–188, 1936. Available at <http://digital.library.adelaide.edu.au/dspace/handle/2440/15227>.

## See Also

### Functions

`fitcdiscr`

### Using Objects

`ClassificationDiscriminant`

## Related Examples

- “Creating Discriminant Analysis Model” on page 20-5
- “Create and Visualize Discriminant Analysis Classifier” on page 20-11
- “Improving Discriminant Analysis Models” on page 20-20
- “Regularize a Discriminant Analysis Classifier” on page 20-28

- “Examine the Gaussian Mixture Assumption” on page 20-36
- “Prediction Using Discriminant Analysis Models” on page 20-8



## Creating Discriminant Analysis Model

The model for discriminant analysis is:

- Each class ( $Y$ ) generates data ( $X$ ) using a multivariate normal distribution. In other words, the model assumes  $X$  has a Gaussian mixture distribution (`gmdistribution`).
- For linear discriminant analysis, the model has the same covariance matrix for each class; only the means vary.
- For quadratic discriminant analysis, both means and covariances of each class vary.

Under this modeling assumption, `fitcdiscr` infers the mean and covariance parameters of each class.

- For linear discriminant analysis, it computes the sample mean of each class. Then it computes the sample covariance by first subtracting the sample mean of each class from the observations of that class, and taking the empirical covariance matrix of the result.
- For quadratic discriminant analysis, it computes the sample mean of each class. Then it computes the sample covariances by first subtracting the sample mean of each class from the observations of that class, and taking the empirical covariance matrix of each class.

The `fit` method does not use prior probabilities or costs for fitting.

### Weighted Observations

`fitcdiscr` constructs weighted classifiers using the following scheme. Suppose  $M$  is an  $N$ -by- $K$  class membership matrix:

$M_{nk} = 1$  if observation  $n$  is from class  $k$   
 $M_{nk} = 0$  otherwise.

The estimate of the class mean for unweighted data is

$$\hat{\mu}_k = \frac{\sum_{n=1}^N M_{nk} x_n}{\sum_{n=1}^N M_{nk}}.$$

For weighted data with positive weights  $w_n$ , the natural generalization is

$$\hat{\mu}_k = \frac{\sum_{n=1}^N M_{nk} w_n x_n}{\sum_{n=1}^N M_{nk} w_n}.$$

The unbiased estimate of the pooled-in covariance matrix for unweighted data is

$$\hat{\Sigma} = \frac{\sum_{n=1}^N \sum_{k=1}^K M_{nk} (x_n - \hat{\mu}_k)(x_n - \hat{\mu}_k)^T}{N - K}.$$

For quadratic discriminant analysis, `fitcdiscr` uses  $K = 1$ .

For weighted data, assuming the weights sum to 1, the unbiased estimate of the pooled-in covariance matrix is

$$\hat{\Sigma} = \frac{\sum_{n=1}^N \sum_{k=1}^K M_{nk} w_n (x_n - \hat{\mu}_k)(x_n - \hat{\mu}_k)^T}{1 - \sum_{k=1}^K \frac{W_k^{(2)}}{W_k}},$$

where

•

$W_k = \sum_{n=1}^N M_{nk} w_n$  is the sum of the weights for class  $k$ .

•

$W_k^{(2)} = \sum_{n=1}^N M_{nk} w_n^2$  is the sum of squared weights per class.

## See Also

### Functions

`fitcdiscr`

### Using Objects

`ClassificationDiscriminant` | `gmdistribution`

## Related Examples

- “Discriminant Analysis Classification” on page 20-2
- “Examine the Gaussian Mixture Assumption” on page 20-36

## Prediction Using Discriminant Analysis Models

`predict` uses three quantities to classify observations: posterior probability on page 20-8, prior probability on page 20-9, and cost on page 20-9.

`predict` classifies so as to minimize the expected classification cost:

$$\hat{y} = \arg \min_{y=1, \dots, K} \sum_{k=1}^K \hat{P}(k | x) C(y | k),$$

where

- $\hat{y}$  is the predicted classification.
- $K$  is the number of classes.
- $\hat{P}(k | x)$  is the posterior probability of class  $k$  for observation  $x$ .
- $C(y | k)$  is the cost of classifying an observation as  $y$  when its true class is  $k$ .

The space of  $x$  values divides into regions where a classification  $Y$  is a particular value. The regions are separated by straight lines for linear discriminant analysis, and by conic sections (ellipses, hyperbolas, or parabolas) for quadratic discriminant analysis. For a visualization of these regions, see “Create and Visualize Discriminant Analysis Classifier” on page 20-11.

### Posterior Probability

The posterior probability that a point  $x$  belongs to class  $k$  is the product of the prior probability on page 20-9 and the multivariate normal density. The density function of the multivariate normal with mean  $\mu_k$  and covariance  $\Sigma_k$  at a point  $x$  is

$$P(x | k) = \frac{1}{(2\pi|\Sigma_k|)^{1/2}} \exp\left(-\frac{1}{2}(x - \mu_k)^T \Sigma_k^{-1} (x - \mu_k)\right),$$

where  $|\Sigma_k|$  is the determinant of  $\Sigma_k$ , and  $\Sigma_k^{-1}$  is the inverse matrix.

Let  $P(k)$  represent the prior probability of class  $k$ . Then the posterior probability that an observation  $x$  is of class  $k$  is

$$\hat{P}(k|x) = \frac{P(x|k)P(k)}{P(x)},$$

where  $P(x)$  is a normalization constant, namely, the sum over  $k$  of  $P(x|k)P(k)$ .

## Prior Probability

The prior probability is one of three choices:

- 'uniform' — The prior probability of class  $k$  is 1 over the total number of classes.
- 'empirical' — The prior probability of class  $k$  is the number of training samples of class  $k$  divided by the total number of training samples.
- A numeric vector — The prior probability of class  $k$  is the  $j$ th element of the `Prior` vector. See `fitcdiscr`.

After creating a classifier `obj`, you can set the prior using dot notation:

```
obj.Prior = v;
```

where `v` is a vector of positive elements representing the frequency with which each element occurs. You do not need to retrain the classifier when you set a new prior.

## Cost

There are two costs associated with discriminant analysis classification: the true misclassification cost per class, and the expected misclassification cost per observation.

### True Misclassification Cost per Class

$\text{Cost}(i, j)$  is the cost of classifying an observation into class  $j$  if its true class is  $i$ . By default,  $\text{Cost}(i, j) = 1$  if  $i \neq j$ , and  $\text{Cost}(i, j) = 0$  if  $i = j$ . In other words, the cost is 0 for correct classification, and 1 for incorrect classification.

You can set any cost matrix you like when creating a classifier. Pass the cost matrix in the `Cost` name-value pair in `fitcdiscr`.

After you create a classifier `obj`, you can set a custom cost using dot notation:

```
obj.Cost = B;
```

$B$  is a square matrix of size  $K$ -by- $K$  when there are  $K$  classes. You do not need to retrain the classifier when you set a new cost.

### Expected Misclassification Cost per Observation

Suppose you have  $N_{\text{obs}}$  observations that you want to classify with a trained discriminant analysis classifier `obj`. Suppose you have  $K$  classes. You place the observations into a matrix  $X_{\text{new}}$  with one observation per row. The command

```
[label, score, cost] = predict(obj, Xnew)
```

returns, among other outputs, a cost matrix of size  $N_{\text{obs}}$ -by- $K$ . Each row of the cost matrix contains the expected (average) cost of classifying the observation into each of the  $K$  classes. `cost(n, k)` is

$$\sum_{i=1}^K \hat{P}(i | X_{\text{new}(n)}) C(k | i),$$

where

- $K$  is the number of classes.
- $\hat{P}(i | X_{\text{new}(n)})$  is the posterior probability on page 20-8 of class  $i$  for observation  $X_{\text{new}(n)}$ .
- $C(k | i)$  is the cost on page 20-9 of classifying an observation as  $k$  when its true class is  $i$ .

## See Also

### Functions

`fitcdiscr` | `predict`

### Using Objects

`ClassificationDiscriminant` | `CompactClassificationDiscriminant`

## Related Examples

- “Discriminant Analysis Classification” on page 20-2

## Create and Visualize Discriminant Analysis Classifier

This example shows how to perform linear and quadratic classification of Fisher iris data.

Load the sample data.

```
load fisheriris
```

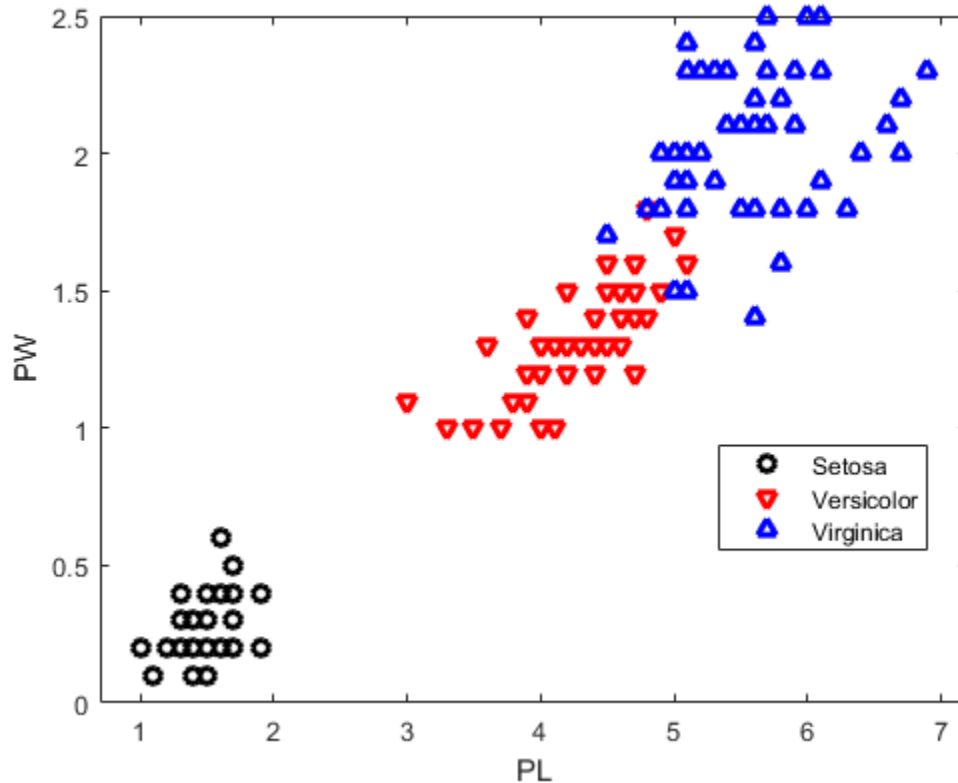
The column vector, `species`, consists of iris flowers of three different species, `setosa`, `versicolor`, `virginica`. The double matrix `meas` consists of four types of measurements on the flowers, the length and width of sepals and petals in centimeters, respectively.

Use petal length (third column in `meas`) and petal width (fourth column in `meas`) measurements. Save these as variables `PL` and `PW`, respectively.

```
PL = meas(:,3);  
PW = meas(:,4);
```

Plot the data, showing the classification, that is, create a scatter plot of the measurements, grouped by species.

```
h1 = gscatter(PL,PW,species,'krb','ov^',[],'off');  
h1(1).LineWidth = 2;  
h1(2).LineWidth = 2;  
h1(3).LineWidth = 2;  
legend('Setosa','Versicolor','Virginica','Location','best')  
hold on
```



Create a linear classifier.

```
X = [PL, PW];
MdlLinear = fitcdiscr(X, species);
```

Retrieve the coefficients for the linear boundary between the second and third classes.

```
MdlLinear.ClassNames([2 3])
K = MdlLinear.Coeffs(2,3).Const;
L = MdlLinear.Coeffs(2,3).Linear;
```

```
ans =
```

```
2x1 cell array
```

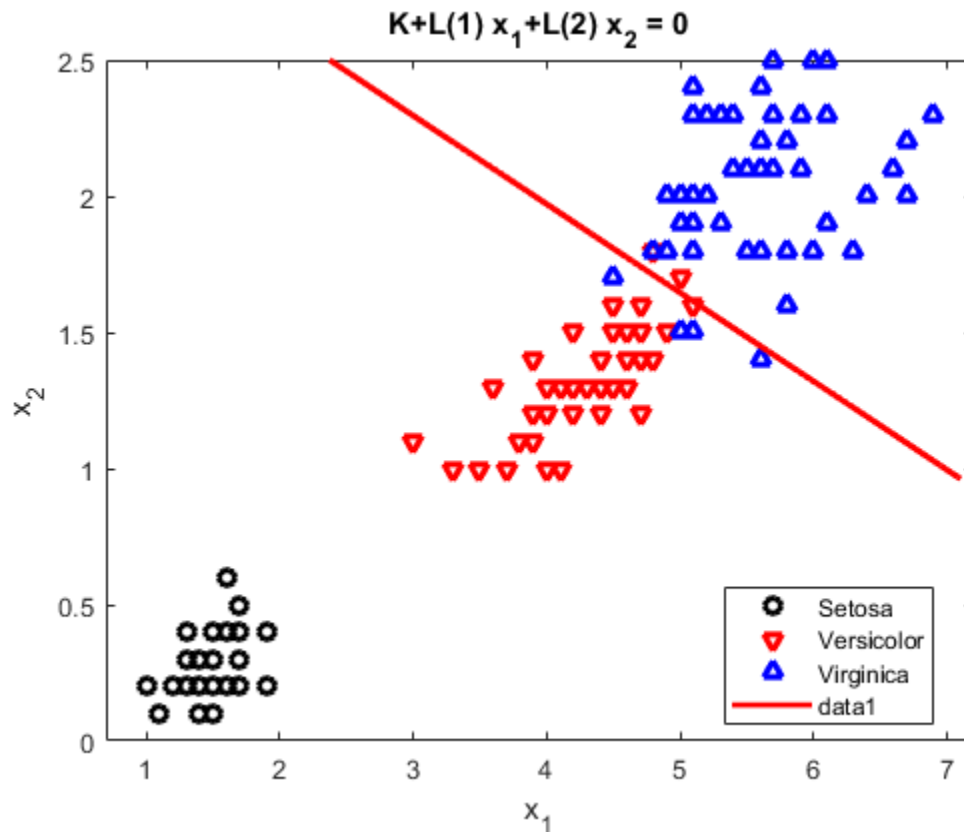


```
{'versicolor'}
{'virginica' }
```

Plot the curve that separates the second and third classes

$K + [x_1 \ x_2] L = 0.$

```
f = @(x1,x2) K + L(1)*x1 + L(2)*x2;
h2 = ezplot(f,[.9 7.1 0 2.5]);
h2.Color = 'r';
h2.LineWidth = 2;
```



Retrieve the coefficients for the linear boundary between the first and second classes.

```
MdlLinear.ClassNames([1 2])
K = MdlLinear.Coeffs(1,2).Const;
L = MdlLinear.Coeffs(1,2).Linear;
```

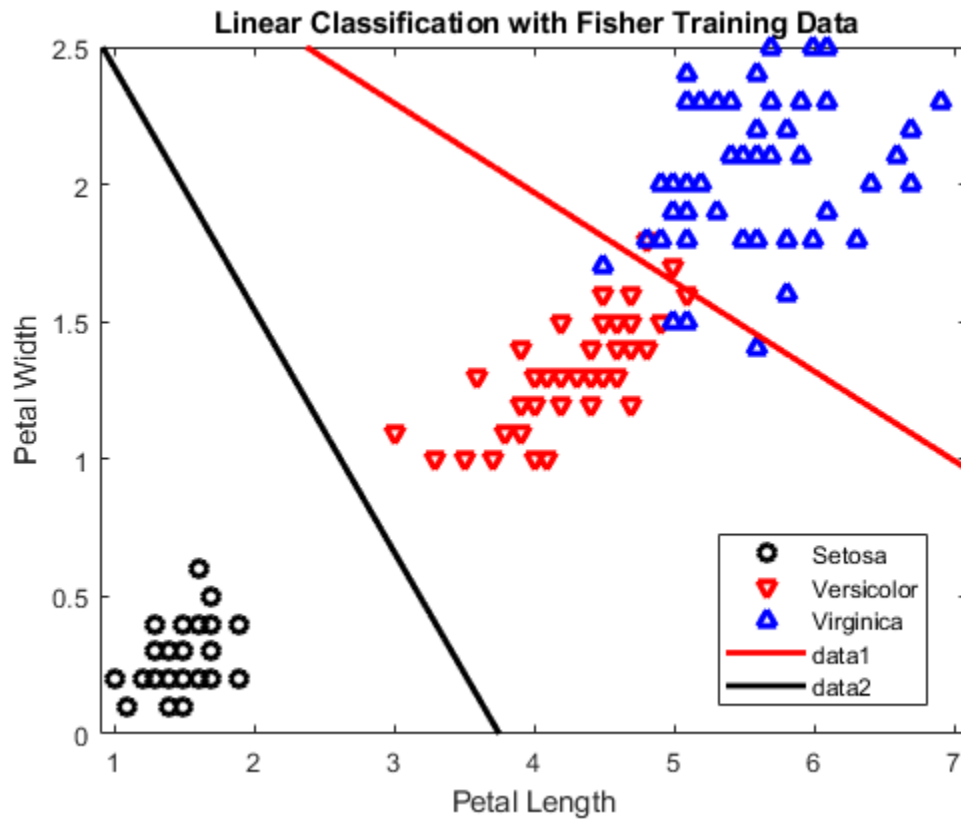
```
ans =
```

```
2x1 cell array

    {'setosa'    }
    {'versicolor'}
```

Plot the curve that separates the first and second classes.

```
f = @(x1,x2) K + L(1)*x1 + L(2)*x2;
h3 = ezplot(f,[.9 7.1 0 2.5]);
h3.Color = 'k';
h3.LineWidth = 2;
axis([.9 7.1 0 2.5])
xlabel('Petal Length')
ylabel('Petal Width')
title('{\bf Linear Classification with Fisher Training Data}')
```

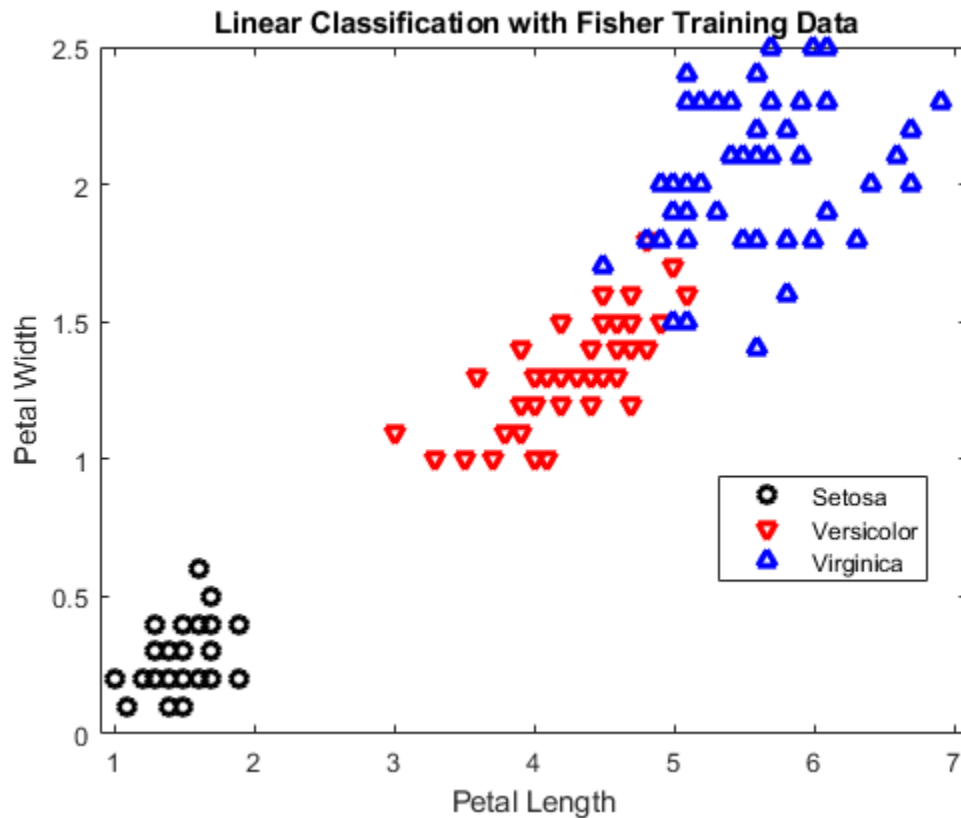


Create a quadratic discriminant classifier.

```
MdlQuadratic = fitcdiscr(X,species,'DiscrimType','quadratic');
```

Remove the linear boundaries from the plot.

```
delete(h2);
delete(h3);
```



Retrieve the coefficients for the quadratic boundary between the second and third classes.

```
MdlQuadratic.ClassNames([2 3])
K = MdlQuadratic.Coeffs(2,3).Const;
L = MdlQuadratic.Coeffs(2,3).Linear;
Q = MdlQuadratic.Coeffs(2,3).Quadratic;
```

ans =

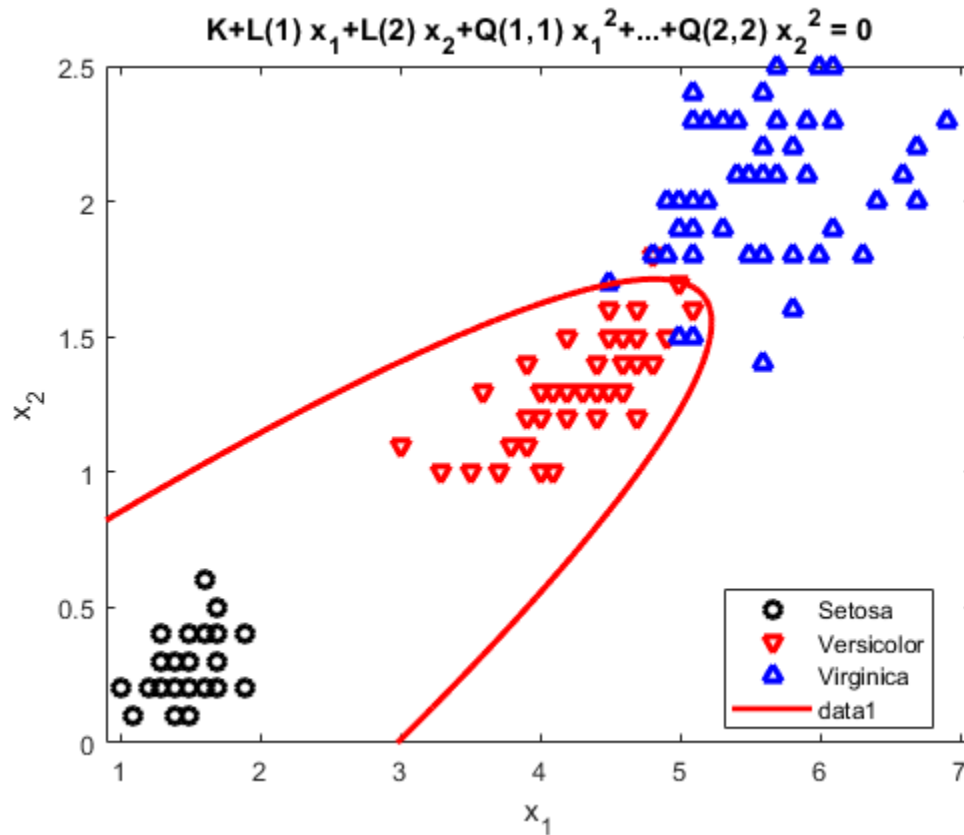
```
2x1 cell array
    {'versicolor'}
```

```
{'virginica' }
```

Plot the curve that separates the second and third classes

$$K + [x_1 \ x_2] L + [x_1 \ x_2] Q \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = 0.$$

```
f = @(x1,x2) K + L(1)*x1 + L(2)*x2 + Q(1,1)*x1.^2 + ...
    (Q(1,2)+Q(2,1))*x1.*x2 + Q(2,2)*x2.^2;
h2 = ezplot(f,[.9 7.1 0 2.5]);
h2.Color = 'r';
h2.LineWidth = 2;
```



Retrieve the coefficients for the quadratic boundary between the first and second classes.

```
MdlQuadratic.ClassNames([1 2])
K = MdlQuadratic.Coeffs(1,2).Const;
L = MdlQuadratic.Coeffs(1,2).Linear;
Q = MdlQuadratic.Coeffs(1,2).Quadratic;
```

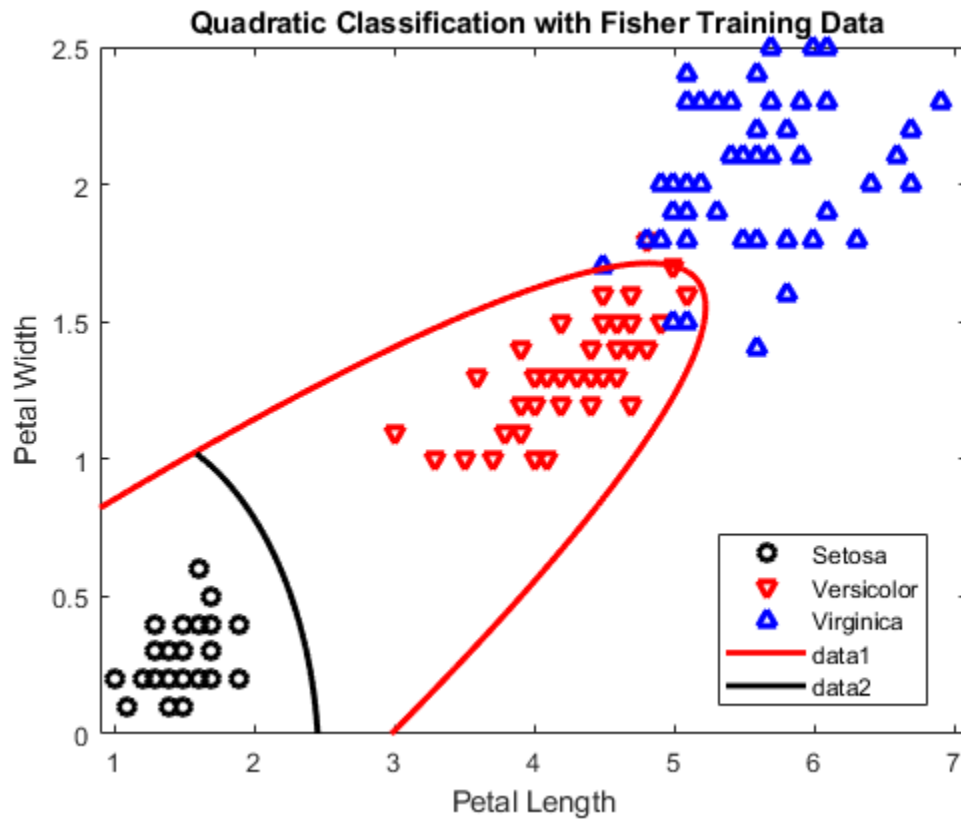
```
ans =
```

```
2x1 cell array

    {'setosa'      }
    {'versicolor'}
```

Plot the curve that separates the first and second and classes.

```
f = @(x1,x2) K + L(1)*x1 + L(2)*x2 + Q(1,1)*x1.^2 + ...
    (Q(1,2)+Q(2,1))*x1.*x2 + Q(2,2)*x2.^2;
h3 = ezplot(f,[.9 7.1 0 1.02]); % Plot the relevant portion of the curve.
h3.Color = 'k';
h3.LineWidth = 2;
axis([.9 7.1 0 2.5])
xlabel('Petal Length')
ylabel('Petal Width')
title('{\bf Quadratic Classification with Fisher Training Data}')
hold off
```



## See Also

### Functions

`fitcdiscr`

### Using Objects

`ClassificationDiscriminant`

## Related Examples

- “Discriminant Analysis Classification” on page 20-2

## Improving Discriminant Analysis Models

### In this section...

“Deal with Singular Data” on page 20-20

“Choose a Discriminant Type” on page 20-21

“Examine the Resubstitution Error and Confusion Matrix” on page 20-22

“Cross Validation” on page 20-23

“Change Costs and Priors” on page 20-24

### Deal with Singular Data

Discriminant analysis needs data sufficient to fit Gaussian models with invertible covariance matrices. If your data is not sufficient to fit such a model uniquely, `fitcdiscr` fails. This section shows methods for handling failures.

---

**Tip** To obtain a discriminant analysis classifier without failure, set the `DiscrimType` name-value pair to 'pseudoLinear' or 'pseudoQuadratic' in `fitcdiscr`.

“Pseudo” discriminants never fail, because they use the pseudoinverse of the covariance matrix  $\Sigma_k$  (see `pinv`).

---

### Example: Singular Covariance Matrix

When the covariance matrix of the fitted classifier is singular, `fitcdiscr` can fail:

```
load popcorn
X = popcorn(:, [1 2]);
X(:,3) = 0; % a zero-variance column
Y = popcorn(:,3);
ppcrn = fitcdiscr(X,Y);

Error using ClassificationDiscriminant (line 635)
Predictor x3 has zero variance. Either exclude this predictor or set 'discrimType' to
'pseudoLinear' or 'diagLinear'.

Error in classreg.learning.FitTemplate/fit (line 243)
    obj = this.MakeFitObject(X,Y,W,this.ModelParameters,fitArgs{});

Error in fitcdiscr (line 296)
    this = fit(temp,X,Y);
```



To proceed with linear discriminant analysis, use a `pseudoLinear` or `diagLinear` discriminant type:

```
ppcrn = fitcdiscr(X,Y,...
    'discrimType','pseudoLinear');
meanpredict = predict(ppcrn,mean(X))

meanpredict =
    3.5000
```

## Choose a Discriminant Type

There are six types of discriminant analysis classifiers: linear and quadratic, with diagonal and pseudo variants of each type.

---

**Tip** To see if your covariance matrix is singular, set `discrimType` to `'linear'` or `'quadratic'`. If the matrix is singular, the `fitcdiscr` method fails for `'quadratic'`, and the `Gamma` property is nonzero for `'linear'`.

To obtain a quadratic classifier even when your covariance matrix is singular, set `DiscrimType` to `'pseudoQuadratic'` or `'diagQuadratic'`.

```
obj = fitcdiscr(X,Y,'DiscrimType','pseudoQuadratic') % or 'diagQuadratic'
```

---

Choose a classifier type by setting the `discrimType` name-value pair to one of:

- `'linear'` (default) — Estimate one covariance matrix for all classes.
- `'quadratic'` — Estimate one covariance matrix for each class.
- `'diagLinear'` — Use the diagonal of the `'linear'` covariance matrix, and use its pseudoinverse if necessary.
- `'diagQuadratic'` — Use the diagonals of the `'quadratic'` covariance matrices, and use their pseudoinverses if necessary.
- `'pseudoLinear'` — Use the pseudoinverse of the `'linear'` covariance matrix if necessary.
- `'pseudoQuadratic'` — Use the pseudoinverses of the `'quadratic'` covariance matrices if necessary.

`fitcdiscr` can fail for the 'linear' and 'quadratic' classifiers. When it fails, it returns an explanation, as shown in “Deal with Singular Data” on page 20-20.

`fitcdiscr` always succeeds with the diagonal and pseudo variants. For information about pseudoinverses, see `pinv`.

You can set the discriminant type using dot notation after constructing a classifier:

```
obj.DiscrimType = 'discrimType'
```

You can change between linear types or between quadratic types, but cannot change between a linear and a quadratic type.

## Examine the Resubstitution Error and Confusion Matrix

The resubstitution error is the difference between the response training data and the predictions the classifier makes of the response based on the input training data. If the resubstitution error is high, you cannot expect the predictions of the classifier to be good. However, having low resubstitution error does not guarantee good predictions for new data. Resubstitution error is often an overly optimistic estimate of the predictive error on new data.

The confusion matrix shows how many errors, and which types, arise in resubstitution. When there are  $K$  classes, the confusion matrix  $R$  is a  $K$ -by- $K$  matrix with

$R(i, j)$  = the number of observations of class  $i$  that the classifier predicts to be of class  $j$ .

### Example: Resubstitution Error of a Discriminant Analysis Classifier

Examine the resubstitution error of the default discriminant analysis classifier for the Fisher iris data:

```
load fisheriris
obj = fitcdiscr(meas, species);
resuberror = resubLoss(obj)

resuberror =
    0.0200
```

The resubstitution error is very low, meaning `obj` classifies nearly all the Fisher iris data correctly. The total number of misclassifications is:

```
resuberror * obj.NumObservations
```

```
ans =
    3.0000
```

To see the details of the three misclassifications, examine the confusion matrix:

```
R = confusionmat(obj.Y, resubPredict(obj))
```

```
R =
    50     0     0
     0    48     2
     0     1    49
```

```
obj.ClassNames
```

```
ans =
    'setosa'
    'versicolor'
    'virginica'
```

- $R(1, :) = [50 \ 0 \ 0]$  means `obj` classifies all 50 setosa irises correctly.
- $R(2, :) = [0 \ 48 \ 2]$  means `obj` classifies 48 versicolor irises correctly, and misclassifies two versicolor irises as virginica.
- $R(3, :) = [0 \ 1 \ 49]$  means `obj` classifies 49 virginica irises correctly, and misclassifies one virginica iris as versicolor.

## Cross Validation

Typically, discriminant analysis classifiers are robust and do not exhibit overtraining when the number of predictors is much less than the number of observations. Nevertheless, it is good practice to cross validate your classifier to ensure its stability.

### Cross Validating a Discriminant Analysis Classifier

This example shows how to perform five-fold cross validation of a quadratic discriminant analysis classifier.

Load the sample data.

```
load fisheriris
```

Create a quadratic discriminant analysis classifier for the data.

```
quadisc = fitcdiscr(meas, species, 'DiscrimType', 'quadratic');
```

Find the resubstitution error of the classifier.

```
qerror = resubLoss(quadisc)
```

```
qerror =  
    0.0200
```

The classifier does an excellent job. Nevertheless, resubstitution error can be an optimistic estimate of the error when classifying new data. So proceed to cross validation.

Create a cross-validation model.

```
cvmodel = crossval(quadisc, 'kfold', 5);
```

Find the cross-validation loss for the model, meaning the error of the out-of-fold observations.

```
cverror = kfoldLoss(cvmodel)
```

```
cverror =  
    0.0200
```

The cross-validated loss is as low as the original resubstitution loss. Therefore, you can have confidence that the classifier is reasonably accurate.

## Change Costs and Priors

Sometimes you want to avoid certain misclassification errors more than others. For example, it might be better to have oversensitive cancer detection instead of undersensitive cancer detection. Oversensitive detection gives more false positives (unnecessary testing or treatment). Undersensitive detection gives more false negatives (preventable illnesses or deaths). The consequences of underdetection can be high. Therefore, you might want to set costs to reflect the consequences.

Similarly, the training data  $Y$  can have a distribution of classes that does not represent their true frequency. If you have a better estimate of the true frequency, you can include this knowledge in the classification `Prior` property.

### Example: Setting Custom Misclassification Costs

Consider the Fisher iris data. Suppose that the cost of classifying a versicolor iris as virginica is 10 times as large as making any other classification error. Create a classifier from the data, then incorporate this cost and then view the resulting classifier.

- 1 Load the Fisher iris data and create a default (linear) classifier as in “Example: Resubstitution Error of a Discriminant Analysis Classifier” on page 20-22:

```
load fisheriris
obj = fitcdiscr(meas,species);
resuberror = resubLoss(obj)

resuberror =
    0.0200

R = confusionmat(obj.Y, resubPredict(obj))

R =
    50     0     0
     0    48     2
     0     1    49

obj.ClassNames

ans =
    'setosa'
    'versicolor'
    'virginica'
```

$R(2, :) = [0 \ 48 \ 2]$  means obj classifies 48 versicolor irises correctly, and misclassifies two versicolor irises as virginica.

- 2 Change the cost matrix to make fewer mistakes in classifying versicolor irises as virginica:

```
obj.Cost(2,3) = 10;
R2 = confusionmat(obj.Y, resubPredict(obj))

R2 =
    50     0     0
```

```
0    50    0
0     7   43
```

obj now classifies all versicolor irises correctly, at the expense of increasing the number of misclassifications of virginica irises from 1 to 7.

### Example: Setting Alternative Priors

Consider the Fisher iris data. There are 50 irises of each kind in the data. Suppose that, in a particular region, you have historical data that shows virginica are five times as prevalent as the other kinds. Create a classifier that incorporates this information.

- 1 Load the Fisher iris data and make a default (linear) classifier as in “Example: Resubstitution Error of a Discriminant Analysis Classifier” on page 20-22:

```
load fisheriris
obj = fitcdiscr(meas,species);
resuberror = resubLoss(obj)

resuberror =
    0.0200

R = confusionmat(obj.Y, resubPredict(obj))

R =
    50     0     0
     0    48     2
     0     1    49

obj.ClassNames

ans =
    'setosa'
    'versicolor'
    'virginica'
```

R(3,:) = [0 1 49] means obj classifies 49 virginica irises correctly, and misclassifies one virginica iris as versicolor.

- 2 Change the prior to match your historical data, and examine the confusion matrix of the new classifier:

```
obj.Prior = [1 1 5];
R2 = confusionmat(obj.Y, resubPredict(obj))
```

```
R2 =  
  50    0    0  
   0   46    4  
   0    0   50
```

The new classifier classifies all virginica irises correctly, at the expense of increasing the number of misclassifications of versicolor irises from 2 to 4.

## See Also

### Functions

`crossval` | `cvshrink` | `fitcdiscr` | `logP` | `loss` | `predict` | `resubLoss`

### Using Objects

`ClassificationDiscriminant`

## Related Examples

- “Discriminant Analysis Classification” on page 20-2
- “Regularize a Discriminant Analysis Classifier” on page 20-28

## Regularize a Discriminant Analysis Classifier

This example shows how to make a more robust and simpler model by trying to remove predictors without hurting the predictive power of the model. This is especially important when you have many predictors in your data. Linear discriminant analysis uses the two regularization parameters, Gamma and Delta (see “Definitions” on page 32-1303), to identify and remove redundant predictors. The `ClassificationDiscriminant.cvshrink` method helps identify appropriate settings for these parameters.

### Load data and create a classifier.

Create a linear discriminant analysis classifier for the `ovariancancer` data. Set the `SaveMemory` and `FillCoeffs` name-value pair arguments to keep the resulting model reasonably small. For computational ease, this example uses a random subset of about one third of the predictors to train the classifier.

```
load ovariancancer
rng(1); % For reproducibility
numPred = size(obs,2);
obs = obs(:,randsample(numPred,ceil(numPred/3)));
Mdl = fitcdiscr(obs,grp,'SaveMemory','on','FillCoeffs','off');
```

### Cross validate the classifier.

Use 25 levels of Gamma and 25 levels of Delta to search for good parameters. This search is time consuming. Set `Verbose` to 1 to view the progress.

```
[err,gamma,delta,numpred] = cvshrink(Mdl,...
    'NumGamma',24,'NumDelta',24,'Verbose',1);
```

```
Done building cross-validated model.
Processing Gamma step 1 out of 25.
Processing Gamma step 2 out of 25.
Processing Gamma step 3 out of 25.
Processing Gamma step 4 out of 25.
Processing Gamma step 5 out of 25.
Processing Gamma step 6 out of 25.
Processing Gamma step 7 out of 25.
Processing Gamma step 8 out of 25.
Processing Gamma step 9 out of 25.
Processing Gamma step 10 out of 25.
Processing Gamma step 11 out of 25.
```

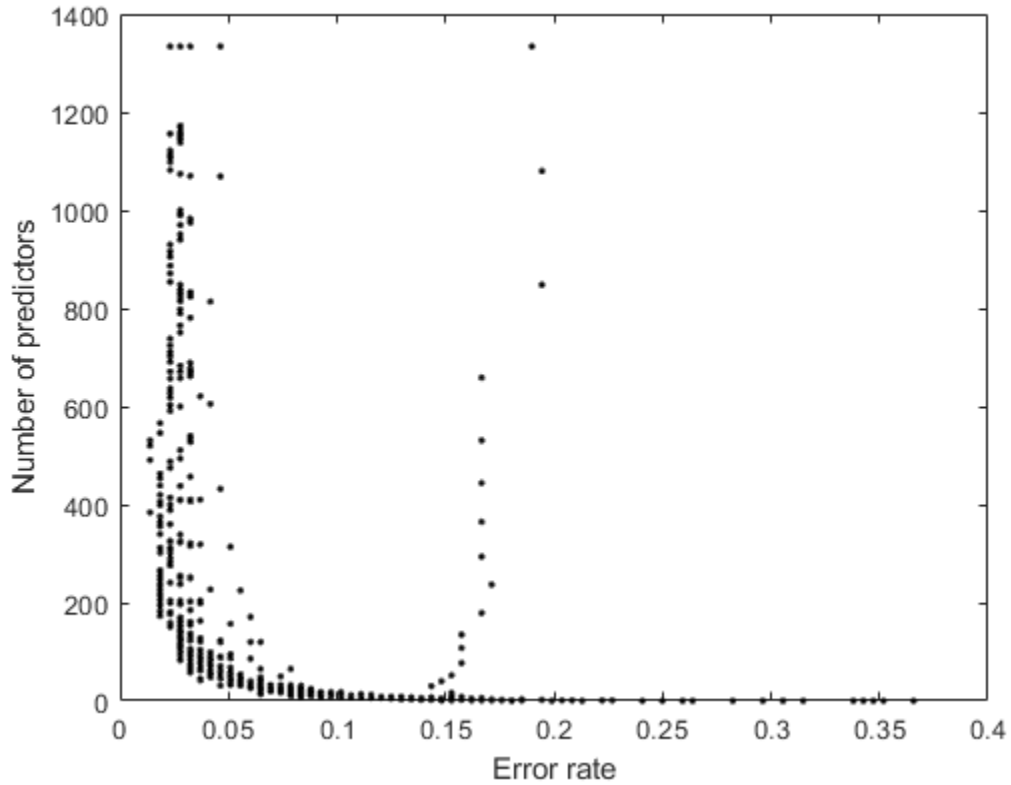


```
Processing Gamma step 12 out of 25.  
Processing Gamma step 13 out of 25.  
Processing Gamma step 14 out of 25.  
Processing Gamma step 15 out of 25.  
Processing Gamma step 16 out of 25.  
Processing Gamma step 17 out of 25.  
Processing Gamma step 18 out of 25.  
Processing Gamma step 19 out of 25.  
Processing Gamma step 20 out of 25.  
Processing Gamma step 21 out of 25.  
Processing Gamma step 22 out of 25.  
Processing Gamma step 23 out of 25.  
Processing Gamma step 24 out of 25.  
Processing Gamma step 25 out of 25.
```

**Examine the quality of the regularized classifiers.**

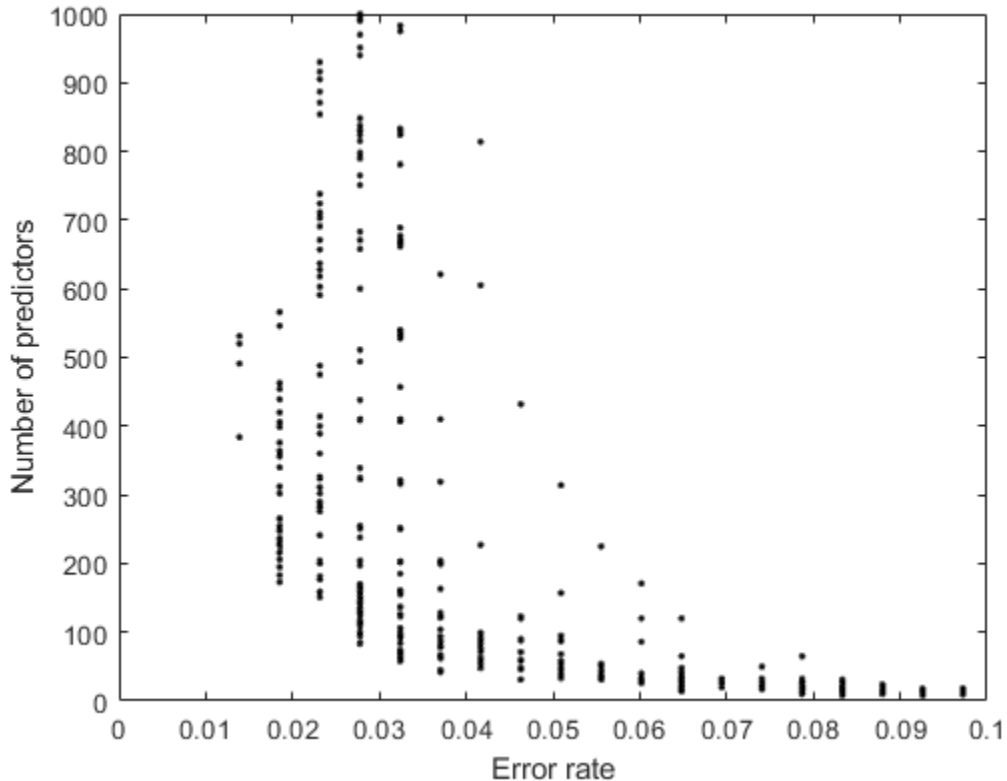
Plot the number of predictors against the error.

```
figure;  
plot(err,numpred,'k.')  
xlabel('Error rate');  
ylabel('Number of predictors');
```



Examine the lower-left part of the plot more closely.

```
axis([0 .1 0 1000])
```



There is a clear tradeoff between lower number of predictors and lower error.

**Choose an optimal tradeoff between model size and accuracy.**

Multiple pairs of Gamma and Delta values produce about the same minimal error.

Display the indices of these pairs and their values.

```
minerr = min(min(err))
[p,q] = find(err < minerr + 1e-4); % Subscripts of err producing minimal error
numel(p)
idx = sub2ind(size(delta),p,q); % Convert from subscripts to linear indices
[gamma(p) delta(idx)]
```

```
minerr =
```

```
0.0139

ans =

4

ans =

0.7202    0.1145
0.7602    0.1131
0.8001    0.1128
0.8001    0.1410
```

These points have as few as 20% of the total predictors that have nonzero coefficients in the model.

```
numpred(idx)/ceil(numPred/3)*100

ans =

39.8051
38.9805
36.8066
28.7856
```

To further lower the number of predictors, you must accept larger error rates. For example, to choose the Gamma and Delta that give the lowest error rate with 200 or fewer predictors.

```
low200 = min(min(err(numpred <= 200)));
lownum = min(min(numpred(err == low200)));
[low200 lownum]

ans =

0.0185    173.0000
```

You need 195 predictors to achieve an error rate of 0.0185, and this is the lowest error rate among those that have 200 predictors or fewer.

Display the Gamma and Delta that achieve this error/number of predictors.

```
[r,s] = find((err == low200) & (numpred == lownum));
[gamma(r); delta(r,s)]
```

```
ans =
```

```
    0.6403
    0.2399
```

### Set the regularization parameters.

To set the classifier with these values of Gamma and Delta, use dot notation.

```
Mdl.Gamma = gamma(r);
Mdl.Delta = delta(r,s);
```

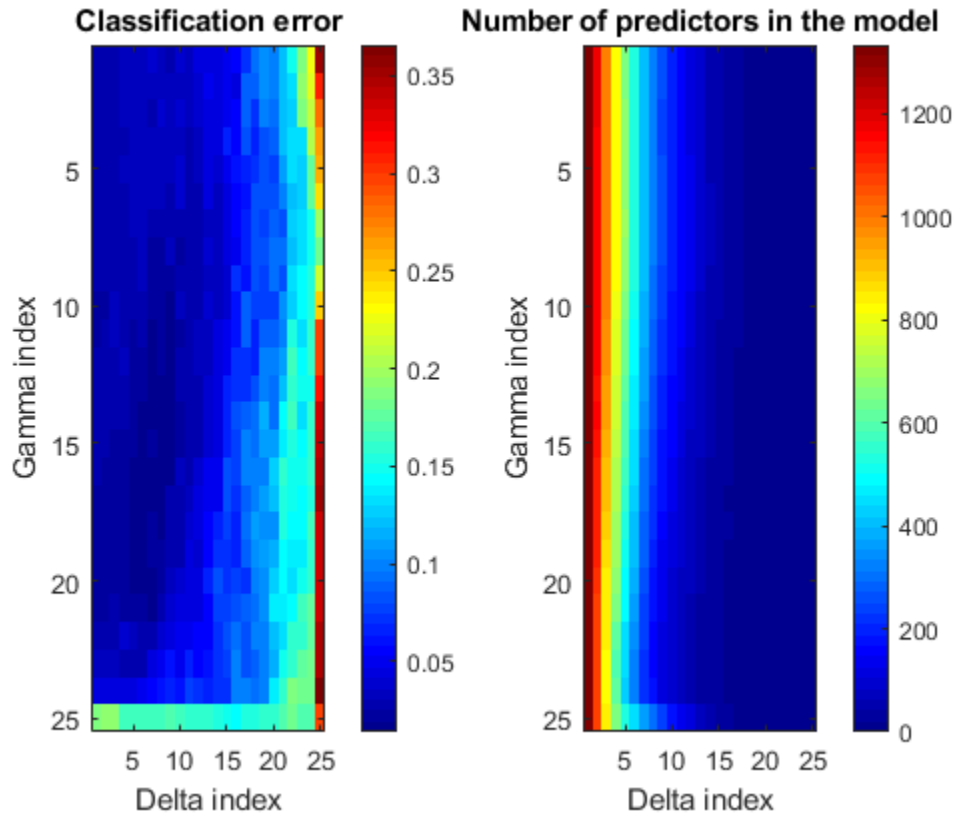
### Heat map plot

To compare the `cvshrink` calculation to that in Guo, Hastie, and Tibshirani [1], plot heat maps of error and number of predictors against Gamma and the index of the Delta parameter. (The Delta parameter range depends on the value of the Gamma parameter. So to get a rectangular plot, use the Delta index, not the parameter itself.)

```
% Create the Delta index matrix
indx = repmat(1:size(delta,2),size(delta,1),1);
figure
subplot(1,2,1)
imagesc(err);
colorbar;
colormap('jet')
title 'Classification error';
xlabel 'Delta index';
ylabel 'Gamma index';

subplot(1,2,2)
imagesc(numpred);
colorbar;
title 'Number of predictors in the model';
```

```
xlabel 'Delta index' ;  
ylabel 'Gamma index' ;
```



You see the best classification error when Delta is small, but fewest predictors when Delta is large.

## References

- [1] Guo, Y., T. Hastie, and R. Tibshirani. *Regularized Discriminant Analysis and Its Application in Microarray*. *Biostatistics*, Vol. 8, No. 1, pp. 86–100, 2007.

## See Also

### Functions

`cvshrink` | `fitcdiscr`

### Using Objects

`ClassificationDiscriminant`

### Related Examples

- “Discriminant Analysis Classification” on page 20-2
- “Improving Discriminant Analysis Models” on page 20-20

## Examine the Gaussian Mixture Assumption

Discriminant analysis assumes that the data comes from a Gaussian mixture model (see “Creating Discriminant Analysis Model” on page 20-5). If the data appears to come from a Gaussian mixture model, you can expect discriminant analysis to be a good classifier. Furthermore, the default linear discriminant analysis assumes that all class covariance matrices are equal. This section shows methods to check these assumptions:

### In this section...

“Bartlett Test of Equal Covariance Matrices for Linear Discriminant Analysis” on page 20-36

“Q-Q Plot” on page 20-37

“Mardia Kurtosis Test of Multivariate Normality” on page 20-41

## Bartlett Test of Equal Covariance Matrices for Linear Discriminant Analysis

The Bartlett test (see Box [1]) checks equality of the covariance matrices of the various classes. If the covariance matrices are equal, the test indicates that linear discriminant analysis is appropriate. If not, consider using quadratic discriminant analysis, setting the `DiscrimType` name-value pair to 'quadratic' in `fitcdiscr`.

The Bartlett test assumes normal (Gaussian) samples, where neither the means nor covariance matrices are known. To determine whether the covariances are equal, compute the following quantities:

- Sample covariance matrices per class  $\sigma_i$ ,  $1 \leq i \leq k$ , where  $k$  is the number of classes.
- Pooled-in covariance matrix  $\sigma$ .
- Test statistic  $V$ :

$$V = (n - k) \log(|\Sigma|) - \sum_{i=1}^k (n_i - 1) \log(|\Sigma_i|)$$

where  $n$  is the total number of observations, and  $n_i$  is the number of observations in class  $i$ , and  $|\Sigma|$  means the determinant of the matrix  $\Sigma$ .

- Asymptotically, as the number of observations in each class  $n_i$  become large,  $V$  is distributed approximately  $\chi^2$  with  $kd(d + 1)/2$  degrees of freedom, where  $d$  is the number of predictors (number of dimensions in the data).



The Bartlett test is to check whether  $V$  exceeds a given percentile of the  $\chi^2$  distribution with  $kd(d+1)/2$  degrees of freedom. If it does, then reject the hypothesis that the covariances are equal.

### Example: Bartlett Test for Equal Covariance Matrices

Check whether the Fisher iris data is well modeled by a single Gaussian covariance, or whether it would be better to model it as a Gaussian mixture.

```
load fisheriris;
prednames = {'SepalLength','SepalWidth','PetalLength','PetalWidth'};
L = fitcdiscr(meas,species,'PredictorNames',prednames);
Q = fitcdiscr(meas,species,'PredictorNames',prednames,'DiscrimType','quadratic');
D = 4; % Number of dimensions of X
Nclass = [50 50 50];
N = L.NumObservations;
K = numel(L.ClassNames);
SigmaQ = Q.Sigma;
SigmaL = L.Sigma;
logV = (N-K)*log(det(SigmaL));
for k=1:K
    logV = logV - (Nclass(k)-1)*log(det(SigmaQ(:,:,k)));
end
nu = (K-1)*D*(D+1)/2;
pval = 1 - chi2cdf(logV,nu)
```

```
pval =
    0
```

The Bartlett test emphatically rejects the hypothesis of equal covariance matrices. If `pval` had been greater than 0.05, the test would not have rejected the hypothesis. The result indicates to use quadratic discriminant analysis, as opposed to linear discriminant analysis.

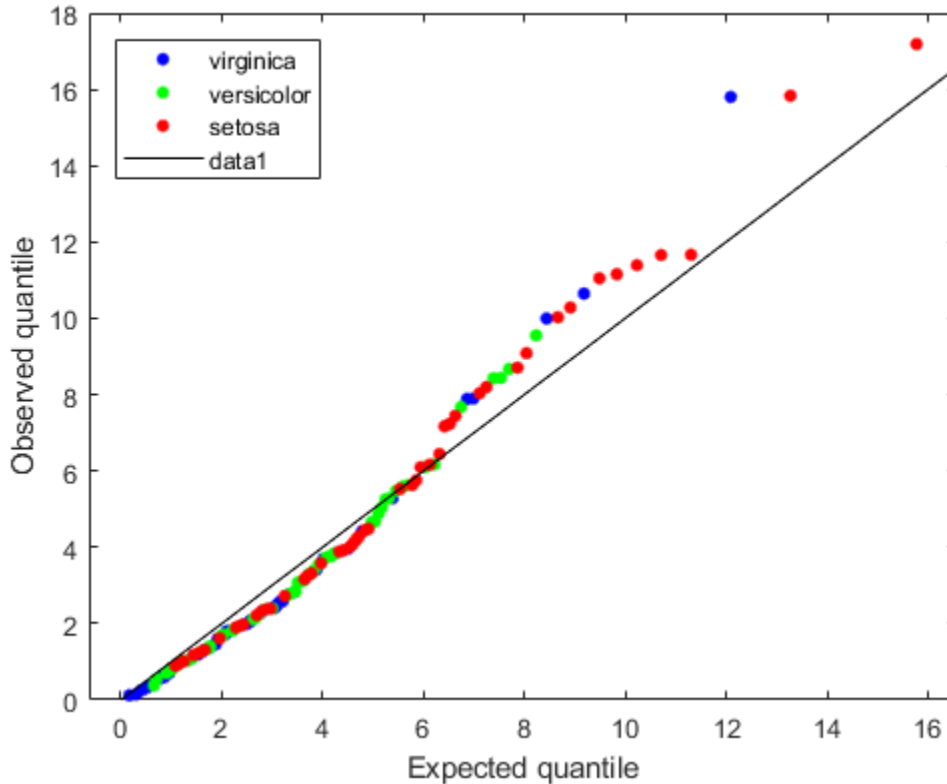
### Q-Q Plot

A Q-Q plot graphically shows whether an empirical distribution is close to a theoretical distribution. If the two are equal, the Q-Q plot lies on a 45° line. If not, the Q-Q plot strays from the 45° line.

### Check Q-Q Plots for Linear and Quadratic Discriminants

For linear discriminant analysis, use a single covariance matrix for all classes.

```
load fisheriris;
prednames = {'SepalLength', 'SepalWidth', 'PetalLength', 'PetalWidth'};
L = fitcdiscr(meas, species, 'PredictorNames', prednames);
N = L.NumObservations;
K = numel(L.ClassNames);
mahL = mahal(L, L.X, 'ClassLabels', L.Y);
D = 4;
expQ = chi2inv((1:N)-0.5)/N, D); % expected quantiles
[mahL, sorted] = sort(mahL); % sorted observed quantiles
figure;
gscatter(expQ, mahL, L.Y(sorted), 'bgr', [], [], 'off');
legend('virginica', 'versicolor', 'setosa', 'Location', 'NW');
xlabel('Expected quantile');
ylabel('Observed quantile');
line([0 20], [0 20], 'color', 'k');
```



Overall, the agreement between the expected and observed quantiles is good. Look at the right half of the plot. The deviation of the plot from the 45° line upward indicates that the data has tails heavier than a normal distribution. There are three possible outliers on the right: two observations from class 'setosa' and one observation from class 'virginica'.

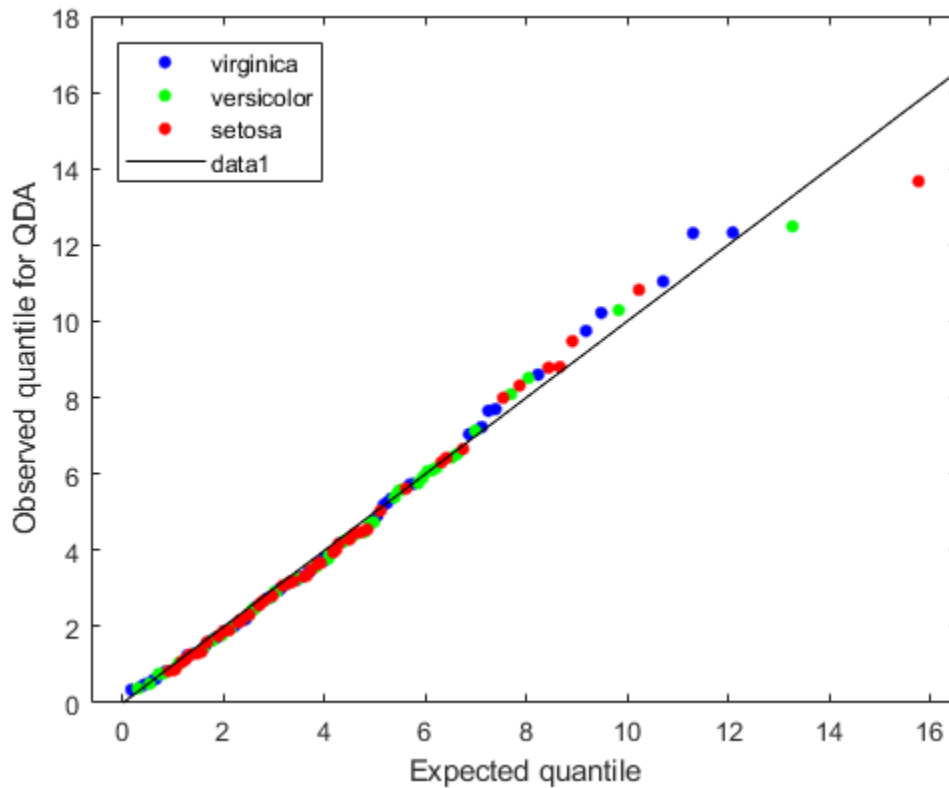
As shown in “Bartlett Test of Equal Covariance Matrices for Linear Discriminant Analysis” on page 20-36, the data does not match a single covariance matrix. Redo the calculations for a quadratic discriminant.

```
load fisheriris;
prednames = {'SepalLength', 'SepalWidth', 'PetalLength', 'PetalWidth'};
Q = fitcdiscr(meas, species, 'PredictorNames', prednames, 'DiscrimType', 'quadratic');
```

```

Nclass = [50 50 50];
N = L.NumObservations;
K = numel(L.ClassNames);
mahQ = mahal(Q,Q.X,'ClassLabels',Q.Y);
expQ = chi2inv(((1:N)-0.5)/N,D);
[mahQ,sorted] = sort(mahQ);
figure;
gscatter(expQ,mahQ,Q.Y(sorted),'bgr',[[],[],'off']);
legend('virginica','versicolor','setosa','Location','NW');
xlabel('Expected quantile');
ylabel('Observed quantile for QDA');
line([0 20],[0 20],'color','k');

```



The Q-Q plot shows a better agreement between the observed and expected quantiles. There is only one outlier candidate, from class 'setosa'.

## Mardia Kurtosis Test of Multivariate Normality

The Mardia kurtosis test (see Mardia [2]) is an alternative to examining a Q-Q plot. It gives a numeric approach to deciding if data matches a Gaussian mixture model.

In the Mardia kurtosis test you compute  $M$ , the mean of the fourth power of the Mahalanobis distance of the data from the class means. If the data is normally distributed with constant covariance matrix (and is thus suitable for linear discriminant analysis),  $M$  is asymptotically distributed as normal with mean  $d(d + 2)$  and variance  $8d(d + 2)/n$ , where

- $d$  is the number of predictors (number of dimensions in the data).
- $n$  is the total number of observations.

The Mardia test is two sided: check whether  $M$  is close enough to  $d(d + 2)$  with respect to a normal distribution of variance  $8d(d + 2)/n$ .

### Example: Mardia Kurtosis Test for Linear and Quadratic Discriminants

Check whether the Fisher iris data is approximately normally distributed for both linear and quadratic discriminant analysis. According to “Bartlett Test of Equal Covariance Matrices for Linear Discriminant Analysis” on page 20-36, the data is not normal for linear discriminant analysis (the covariance matrices are different). “Check Q-Q Plots for Linear and Quadratic Discriminants” on page 20-38 indicates that the data is well modeled by a Gaussian mixture model with different covariances per class. Check these conclusions with the Mardia kurtosis test:

```
load fisheriris;
prednames = {'SepalLength', 'SepalWidth', 'PetalLength', 'PetalWidth'};
L = fitcdiscr(meas, species, 'PredictorNames', prednames);
mahL = mahal(L, L.X, 'ClassLabels', L.Y);
D = 4;
N = L.NumObservations;
obsKurt = mean(mahL.^2);
expKurt = D*(D+2);
varKurt = 8*D*(D+2)/N;
[~, pval] = ztest(obsKurt, expKurt, sqrt(varKurt))

pval =
```

```
0.0208
```

The Mardia test indicates to reject the hypothesis that the data is normally distributed.

Continuing the example with quadratic discriminant analysis:

```
Q = fitcdiscr(meas, species, 'PredictorNames', prednames, 'DiscrimType', 'quadratic');  
mahQ = mahal(Q, Q.X, 'ClassLabels', Q.Y);  
obsKurt = mean(mahQ.^2);  
[~, pval] = ztest(obsKurt, expKurt, sqrt(varKurt))
```

```
pval =
```

```
0.7230
```

Because `pval` is high, you conclude the data are consistent with the multivariate normal distribution.

## References

- [1] Box, G. E. P. *A General Distribution Theory for a Class of Likelihood Criteria*. *Biometrika* 36(3), pp. 317–346, 1949.
- [2] Mardia, K. V. *Measures of multivariate skewness and kurtosis with applications*. *Biometrika* 57 (3), pp. 519–530, 1970.

## See Also

### Functions

```
cvshrink | fitcdiscr
```

### Using Objects

```
ClassificationDiscriminant | gmdistribution
```

## Related Examples

- “Discriminant Analysis Classification” on page 20-2
- “Bartlett Test of Equal Covariance Matrices for Linear Discriminant Analysis” on page 20-36
- “Check Q-Q Plots for Linear and Quadratic Discriminants” on page 20-38





# Naive Bayes

---

- “Naive Bayes Classification” on page 21-2
- “Plot Posterior Classification Probabilities” on page 21-7

## Naive Bayes Classification

The naive Bayes classifier is designed for use when predictors are independent of one another within each class, but it appears to work well in practice even when that independence assumption is not valid. It classifies data in two steps:

- 1 Training step: Using the training data, the method estimates the parameters of a probability distribution, assuming predictors are conditionally independent given the class.
- 2 Prediction step: For any unseen test data, the method computes the posterior probability of that sample belonging to each class. The method then classifies the test data according the largest posterior probability.

The class-conditional independence assumption greatly simplifies the training step since you can estimate the one-dimensional class-conditional density for each predictor individually. While the class-conditional independence between predictors is not true in general, research shows that this optimistic assumption works well in practice. This assumption of class-conditional independence of the predictors allows the naive Bayes classifier to estimate the parameters required for accurate classification while using less training data than many other classifiers. This makes it particularly effective for data sets containing many predictors.

### Supported Distributions

The training step in naive Bayes classification is based on estimating  $P(X|Y)$ , the probability or probability density of predictors  $X$  given class  $Y$ . The naive Bayes classification model `ClassificationNaiveBayes` and training function `fitcnb` provide support for normal (Gaussian), kernel, multinomial, and multivariate, multinomial predictor conditional distributions. To specify distributions for the predictors, use the `DistributionNames` name-value pair argument of `fitcnb`. You can specify one type of distribution for all predictors by supplying the character vector corresponding to the distribution name, or specify different distributions for the predictors by supplying a length  $D$  cell array of character vectors, where  $D$  is the number of predictors (that is, the number of columns of  $X$ ).

#### Normal (Gaussian) Distribution

The 'normal' distribution (specify using 'normal' ) is appropriate for predictors that have normal distributions in each class. For each predictor you model with a normal

distribution, the naive Bayes classifier estimates a separate normal distribution for each class by computing the mean and standard deviation of the training data in that class.

### Kernel Distribution

The 'kernel' distribution (specify using 'kernel') is appropriate for predictors that have a continuous distribution. It does not require a strong assumption such as a normal distribution and you can use it in cases where the distribution of a predictor may be skewed or have multiple peaks or modes. It requires more computing time and more memory than the normal distribution. For each predictor you model with a kernel distribution, the naive Bayes classifier computes a separate kernel density estimate for each class based on the training data for that class. By default the kernel is the normal kernel, and the classifier selects a width automatically for each class and predictor. The software supports specifying different kernels for each predictor, and different widths for each predictor or class.

### Multivariate Multinomial Distribution

The multivariate, multinomial distribution (specify using 'mvnm') is appropriate for a predictor whose observations are categorical. Naive Bayes classifier construction using a multivariate multinomial predictor is described below. To illustrate the steps, consider an example where observations are labeled 0, 1, or 2, and a predictor the weather when the sample was conducted.

- 1 Record the distinct categories represented in the observations of the entire predictor. For example, the distinct categories (or predictor levels) might include sunny, rain, snow, and cloudy.
- 2 Separate the observations by response class. For example, segregate observations labeled 0 from observations labeled 1 and 2, and observations labeled 1 from observations labeled 2.
- 3 For each response class, fit a multinomial model using the category relative frequencies and total number of observations. For example, for observations labeled 0, the estimated probability it was sunny is  $p_{\text{sunny}|0} = (\text{number of sunny observations with label 0}) / (\text{number of observations with label 0})$ , and similar for the other categories and response labels.

The class-conditional, multinomial random variables comprise a multivariate multinomial random variable.

Here are some other properties of naive Bayes classifiers that use multivariate multinomial.

- For each predictor you model with a multivariate multinomial distribution, the naive Bayes classifier:
  - Records a separate set of distinct predictor levels for each predictor
  - Computes a separate set of probabilities for the set of predictor levels for each class.
- The software supports modeling continuous predictors as multivariate multinomial. In this case, the predictor levels are the distinct occurrences of a measurement. This can lead a predictor having many predictor levels. It is good practice to discretize such predictors.

If an *observation* is a set of successes for various categories (represented by all of the predictors) out of a fixed number of independent trials, then specify that the predictors comprise a multinomial distribution. For details, see “Multinomial Distribution” on page 21-4.

### Multinomial Distribution

The multinomial distribution (specify using 'DistributionNames', 'mn') is appropriate when, given the class, each *observation* is a multinomial random variable. That is, observation, or row,  $j$  of the predictor data  $X$  represents  $D$  categories, where  $x_{jd}$  is

the number of successes for category (i.e., predictor)  $d$  in  $n_j = \sum_{d=1}^D x_{jd}$  independent trials. The steps to train a naive Bayes classifier are outlined next.

- 1 For each class, fit a multinomial distribution for the predictors given the class by:
  - a Aggregating the weighted, category counts over all observations. Additionally, the software implements additive smoothing [1].
  - b Estimating the  $D$  category probabilities within each class using the aggregated category counts. These category probabilities compose the probability parameters of the multinomial distribution.
- 2 Let a new observation have a total count of  $m$ . Then, the naive Bayes classifier:
  - a Sets the total count parameter of each multinomial distribution to  $m$

- b** For each class, estimates the class posterior probability using the estimated multinomial distributions
- c** Predicts the observation into the class corresponding to the highest posterior probability

Consider the so-called the bag-of-tokens model, where there is a bag containing a number of tokens of various types and proportions. Each predictor represents a distinct type of token in the bag, an observation is  $n$  independent draws (i.e., with replacement) of tokens from the bag, and the data is a vector of counts, where element  $d$  is the number of times token  $d$  appears.

A machine-learning application is the construction of an email spam classifier, where each predictor represents a word, character, or phrase (i.e., token), an observation is an email, and the data are counts of the tokens in the email. One predictor might count the number of exclamation points, another might count the number of times the word "money" appears, and another might count the number of times the recipient's name appears. This is a naive Bayes model under the further assumption that the total number of tokens (or the total document length) is independent of response class.

Other properties of naive Bayes classifiers that use multinomial observations include:

- Classification is based on the relative frequencies of the categories. If  $n_j = 0$  for observation  $j$ , then classification is not possible for that observation.
- The predictors are not conditionally independent since they must sum to  $n_j$ .
- Naive Bayes is not appropriate when  $n_j$  provides information about the class. That is, this classifier requires that  $n_j$  is independent of the class.
- If you specify that the predictors are conditionally multinomial, then the software applies this specification to all predictors. In other words, you cannot include 'mn' in a cell array when specifying 'DistributionNames'.

If a *predictor* is categorical, i.e., is multinomial within a response class, then specify that it is multivariate multinomial. For details, see “Multivariate Multinomial Distribution” on page 21-3.

## References

- [1] Manning, C. D., P. Raghavan, and M. Schütze. *Introduction to Information Retrieval*, NY: Cambridge University Press, 2008.

## See Also

### Functions

`fitcnb` | `predict`

### Using Objects

`ClassificationNaiveBayes`

## Related Examples

- “Plot Posterior Classification Probabilities” on page 21-7
- “Visualize Decision Surfaces of Different Classifiers” on page 18-11
- “Classification”

## Plot Posterior Classification Probabilities

This example shows how to visualize posterior classification probabilities predicted by a naive Bayes classification model.

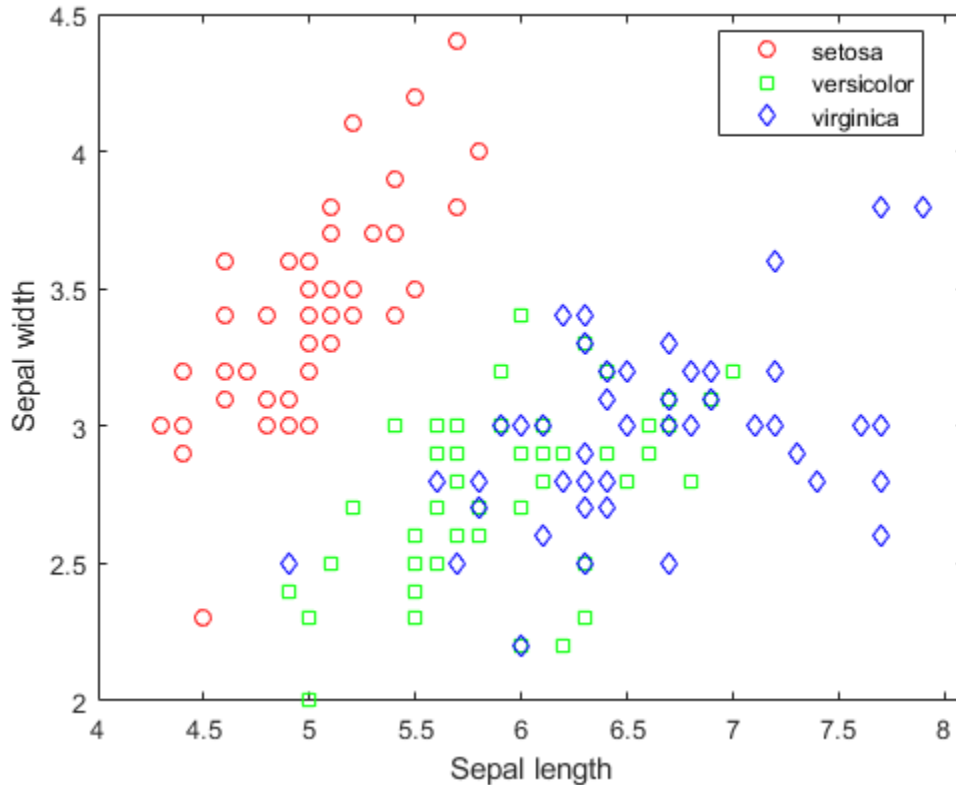
Load Fisher's iris data set.

```
load fisheriris
X = meas(:,1:2);
Y = species;
labels = unique(Y);
```

X is a numeric matrix that contains two petal measurements for 150 irises. Y is a cell array of character vectors that contains the corresponding iris species.

Visualize the data using a scatter plot. Group the variables by iris species.

```
figure;
gscatter(X(:,1), X(:,2), species, 'rgb', 'osd');
xlabel('Sepal length');
ylabel('Sepal width');
```



Train a naive Bayes classifier.

```
mdl = fitcnb(X,Y);
```

mdl is a trained `ClassificationNaiveBayes` classifier.

Create a grid of points spanning the entire space within some bounds of the data. The data in `X(:,1)` ranges between 4.3 and 7.9. The data in `X(:,2)` ranges between 2 and 4.4.

```
[xx1, xx2] = meshgrid(4:.01:8,2:.01:4.5);  
XGrid = [xx1(:) xx2(:)];
```

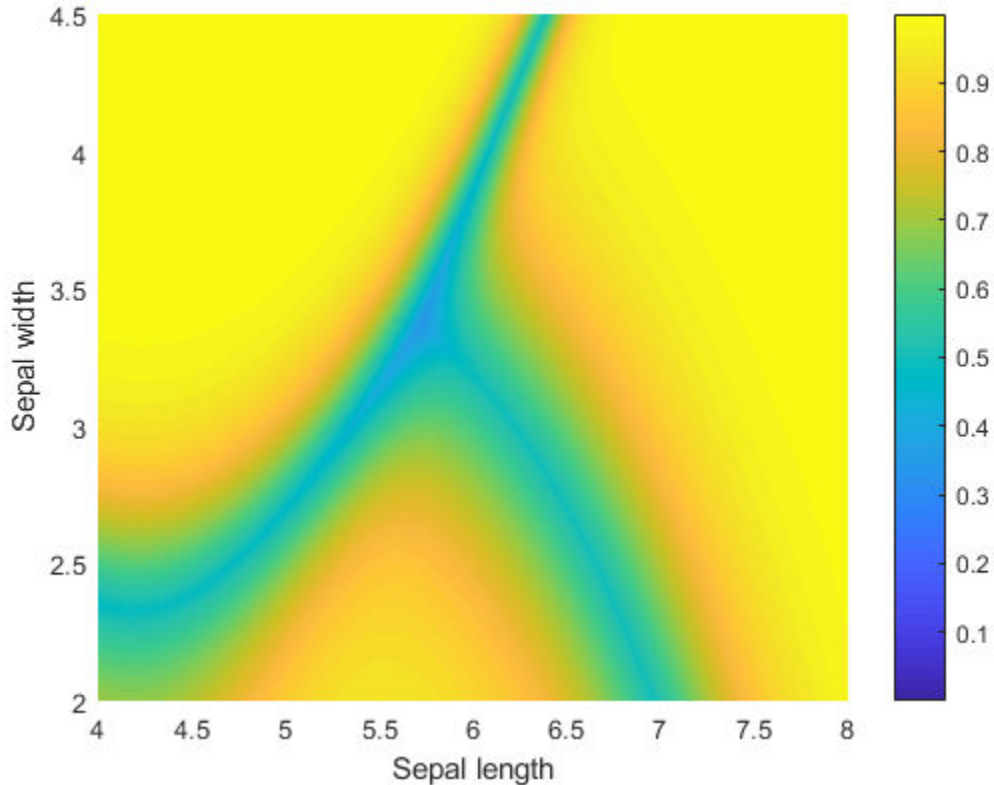


Predict the iris species and posterior class probabilities of each observation in XGrid using mdl.

```
[predictedspecies,Posterior,~] = predict(mdl,XGrid);
```

Plot the posterior probability distribution for each species.

```
sz = size(xx1);  
s = max(Posterior,[],2);  
  
figure  
hold on  
surf(xx1,xx2,reshape(Posterior(:,1),sz),'EdgeColor','none')  
surf(xx1,xx2,reshape(Posterior(:,2),sz),'EdgeColor','none')  
surf(xx1,xx2,reshape(Posterior(:,3),sz),'EdgeColor','none')  
xlabel('Sepal length');  
ylabel('Sepal width');  
colorbar  
view(2)  
hold off
```

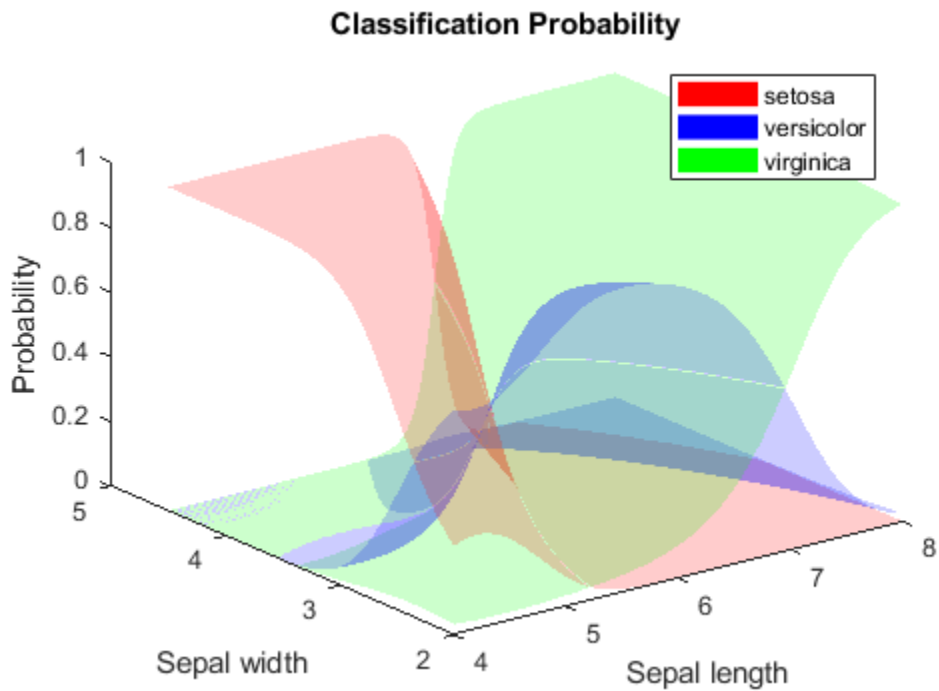


The closer an observation gets to the decision surface, the less probable it is that the data belongs to a certain species.

Plot the classification probability distributions individually.

```
figure('Units','Normalized','Position',[0.25,0.55,0.4,0.35]);
hold on
surf(xx1,xx2,reshape(Posterior(:,1),sz),'FaceColor','red','EdgeColor','none')
surf(xx1,xx2,reshape(Posterior(:,2),sz),'FaceColor','blue','EdgeColor','none')
surf(xx1,xx2,reshape(Posterior(:,3),sz),'FaceColor','green','EdgeColor','none')
xlabel('Sepal length');
ylabel('Sepal width');
zlabel('Probability');
legend(labels)
```

```
title('Classification Probability')
alpha(0.2)
view(3)
hold off
```



## See Also

### Functions

`fitcnb` | `predict`

### Using Objects

`ClassificationNaiveBayes`

## Related Examples

- “Naive Bayes Classification” on page 21-2

# Classification and Regression for High-Dimensional Data

---

## Alternatives for Lower-Dimensional Data

High-dimensional linear classification and regression models minimize objective functions relatively quickly, but at the cost of some accuracy, the numeric-only predictor variables restriction, and the model must be linear with respect to the parameters. If your predictor data set is low- through medium-dimensional, or contains heterogeneous variables, then you should use the appropriate classification or regression fitting function. To help you decide which fitting function is appropriate for your low-dimensional data set, use this table.

Model to Fit	Function	Notable Algorithmic Differences
SVM	<ul style="list-style-type: none"> <li>• Binary classification: <code>fitcsvm</code></li> <li>• Multiclass classification: <code>fitcecoc</code></li> <li>• Regression: <code>fitrsvm</code></li> </ul>	<ul style="list-style-type: none"> <li>• Computes the Gram matrix of the predictor variables, which is convenient for nonlinear kernel transformations.</li> <li>• Solves dual problem using SMO, ISDA, or <math>L1</math> minimization via quadratic programming using <code>quadprog</code>.</li> </ul>
Linear regression	<ul style="list-style-type: none"> <li>• Least-squares without regularization: <code>fitlm</code></li> <li>• Regularized least-squares using a lasso penalty: <code>lasso</code></li> <li>• Ridge regression: <code>ridge</code> or <code>lasso</code></li> </ul>	<ul style="list-style-type: none"> <li>• <code>lasso</code> implements cyclic coordinate descent.</li> </ul>
Logistic regression	<ul style="list-style-type: none"> <li>• Logistic regression without regularization: <code>fitglm</code>.</li> <li>• Regularized logistic regression using a lasso penalty: <code>lassoglm</code></li> </ul>	<ul style="list-style-type: none"> <li>• <code>fitglm</code> implements iteratively reweighted least squares.</li> <li>• <code>lassoglm</code> implements cyclic coordinate descent.</li> </ul>

# Classification Learner

---

- “Machine Learning in MATLAB” on page 23-2
- “Train Classification Models in Classification Learner App” on page 23-11
- “Select Data and Validation for Classification Problem” on page 23-20
- “Choose Classifier Options” on page 23-26
- “Feature Selection and Feature Transformation Using Classification Learner App” on page 23-47
- “Assess Classifier Performance in Classification Learner” on page 23-54
- “Export Classification Model to Predict New Data” on page 23-61
- “Train Decision Trees Using Classification Learner App” on page 23-67
- “Train Discriminant Analysis Classifiers Using Classification Learner App” on page 23-80
- “Train Logistic Regression Classifiers Using Classification Learner App” on page 23-84
- “Train Support Vector Machines Using Classification Learner App” on page 23-88
- “Train Nearest Neighbor Classifiers Using Classification Learner App” on page 23-92
- “Train Ensemble Classifiers Using Classification Learner App” on page 23-96

## Machine Learning in MATLAB

### In this section...

“What Is Machine Learning?” on page 23-2

“Selecting the Right Algorithm” on page 23-3

“Train Classification Models in Classification Learner App” on page 23-7

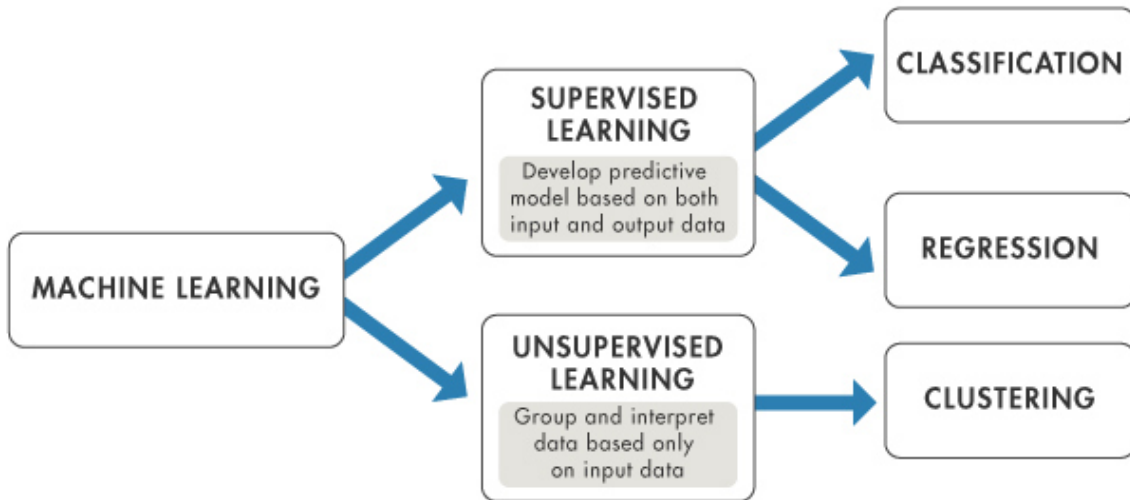
“Train Regression Models in Regression Learner App” on page 23-8

“Train Neural Networks for Deep Learning” on page 23-9

### What Is Machine Learning?

Machine learning teaches computers to do what comes naturally to humans: learn from experience. Machine learning algorithms use computational methods to “learn” information directly from data without relying on a predetermined equation as a model. The algorithms adaptively improve their performance as the number of samples available for learning increases.

Machine learning uses two types of techniques: supervised learning, which trains a model on known input and output data so that it can predict future outputs, and unsupervised learning, which finds hidden patterns or intrinsic structures in input data.





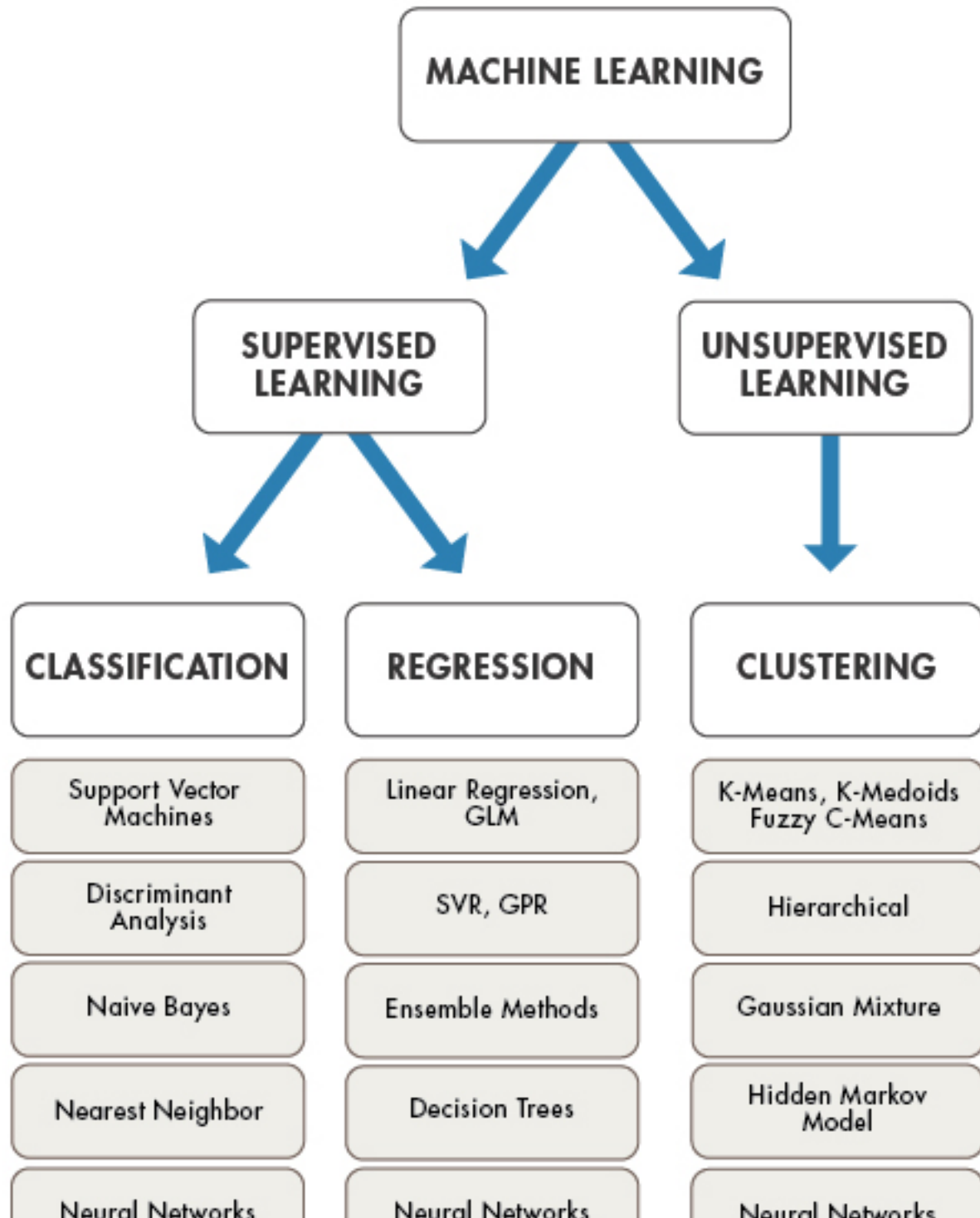
The aim of supervised machine learning is to build a model that makes predictions based on evidence in the presence of uncertainty. A supervised learning algorithm takes a known set of input data and known responses to the data (output) and trains a model to generate reasonable predictions for the response to new data. Supervised learning uses classification and regression techniques to develop predictive models.

- Classification techniques predict categorical responses, for example, whether an email is genuine or spam, or whether a tumor is cancerous or benign. Classification models classify input data into categories. Typical applications include medical imaging, image and speech recognition, and credit scoring.
- Regression techniques predict continuous responses, for example, changes in temperature or fluctuations in power demand. Typical applications include electricity load forecasting and algorithmic trading.

Unsupervised learning finds hidden patterns or intrinsic structures in data. It is used to draw inferences from datasets consisting of input data without labeled responses. Clustering is the most common unsupervised learning technique. It is used for exploratory data analysis to find hidden patterns or groupings in data. Applications for clustering include gene sequence analysis, market research, and object recognition.

## Selecting the Right Algorithm

Choosing the right algorithm can seem overwhelming—there are dozens of supervised and unsupervised machine learning algorithms, and each takes a different approach to learning. There is no best method or one size fits all. Finding the right algorithm is partly based on trial and error—even highly experienced data scientists cannot tell whether an algorithm will work without trying it out. Highly flexible models tend to overfit data by modeling minor variations that could be noise. Simple models are easier to interpret but might have lower accuracy. Therefore, choosing the right algorithm requires trading off one benefit against another, including model speed, accuracy, and complexity. Trial and error is at the core of machine learning—if one approach or algorithm does not work, you try another. MATLAB provides tools to help you try out a variety of machine learning models and choose the best.

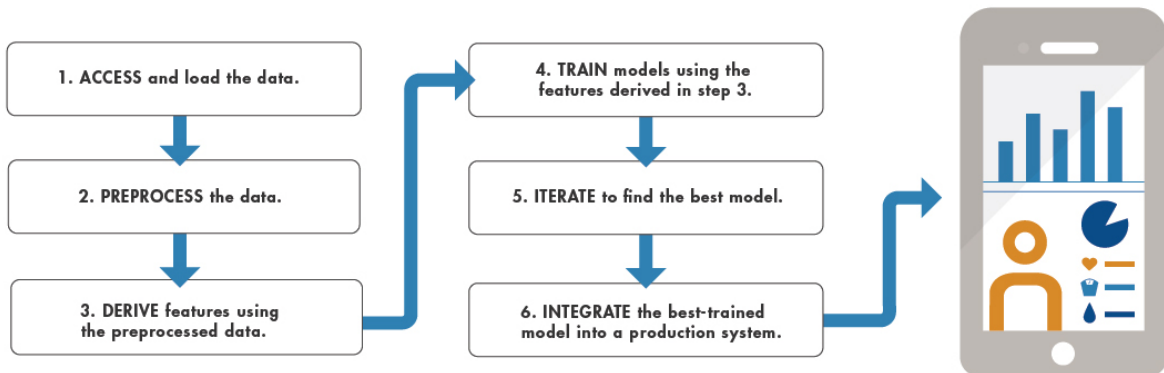


To find MATLAB apps and functions to help you solve machine learning tasks, consult the following table. Some machine learning tasks are made easier by using apps, and others use command-line features.

Task	MATLAB Apps and Functions	Product	Learn More
Classification to predict categorical responses	Use the Classification Learner app to automatically train a selection of models and help you choose the best. You can generate MATLAB code to work with scripts.  For more options, you can use the command-line interface.	Statistics and Machine Learning Toolbox	“Train Classification Models in Classification Learner App” on page 23-7  Classification Functions
Regression to predict continuous responses	Use the Regression Learner app to automatically train a selection of models and help you choose the best. You can generate MATLAB code to work with scripts and other function options.  For more options, you can use the command-line interface.	Statistics and Machine Learning Toolbox	“Train Regression Models in Regression Learner App” on page 23-8  Regression Functions
Clustering	Use cluster analysis functions.	Statistics and Machine Learning Toolbox	“Cluster Analysis”

Task	MATLAB Apps and Functions	Product	Learn More
Computational finance tasks such as credit scoring	Use tools for modeling credit risk analysis.	Financial Toolbox™ Risk Management Toolbox™	“Credit Risk” (Financial Toolbox)
Deep learning with neural networks for classification and regression	Use pretrained networks and functions to train convolutional neural networks.	Neural Network Toolbox™	“Deep Learning in MATLAB” (Neural Network Toolbox)
Facial recognition, motion detection, and object detection	Use deep learning tools for image processing and computer vision.	Neural Network Toolbox Computer Vision System Toolbox™	“Deep Learning, Object Detection and Recognition” (Computer Vision System Toolbox)

The following systematic machine learning workflow can help you tackle machine learning challenges. You can complete the entire workflow in MATLAB.



To integrate the best trained model into a production system, you can deploy Statistics and Machine Learning Toolbox machine learning models using MATLAB Compiler™. For many models, you can generate C-code for prediction using MATLAB Coder™.

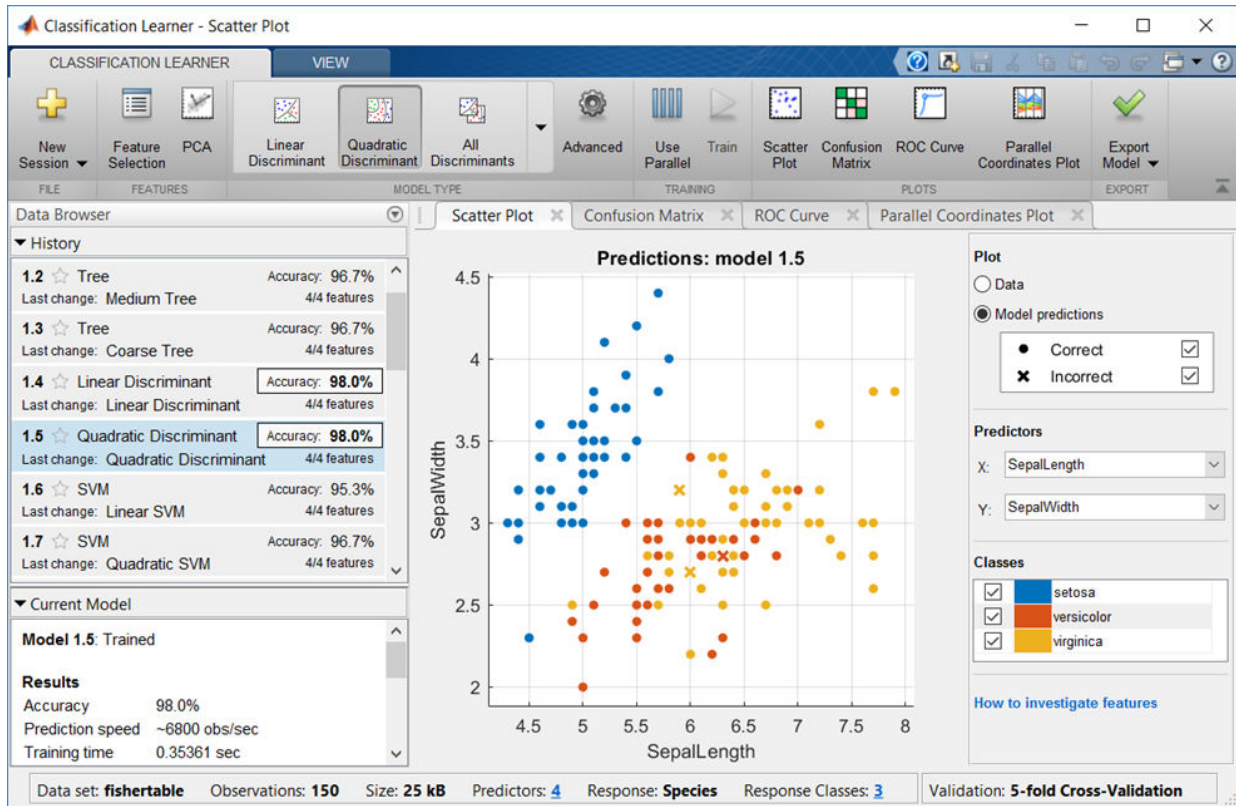
## Train Classification Models in Classification Learner App

Use the Classification Learner app to train models to classify data using supervised machine learning. The app lets you explore supervised machine learning interactively using various classifiers.

- Automatically train a selection of models and help you choose the best model. Model types include decision trees, discriminant analysis, support vector machines, logistic regression, nearest neighbors, and ensemble classification.
- Explore your data, select features, and visualize results.
- Export models to the workspace to make predictions with new data.
- Generate MATLAB code from the app to create scripts, train with new data, work with huge data sets, or modify the code for further analysis.

By default, the app protects against overfitting by applying cross-validation. Alternatively, you can choose holdout validation.

To learn more, see “Train Classification Models in Classification Learner App” on page 23-11.



For more options, you can use the command-line interface. See “Classification”.

## Train Regression Models in Regression Learner App

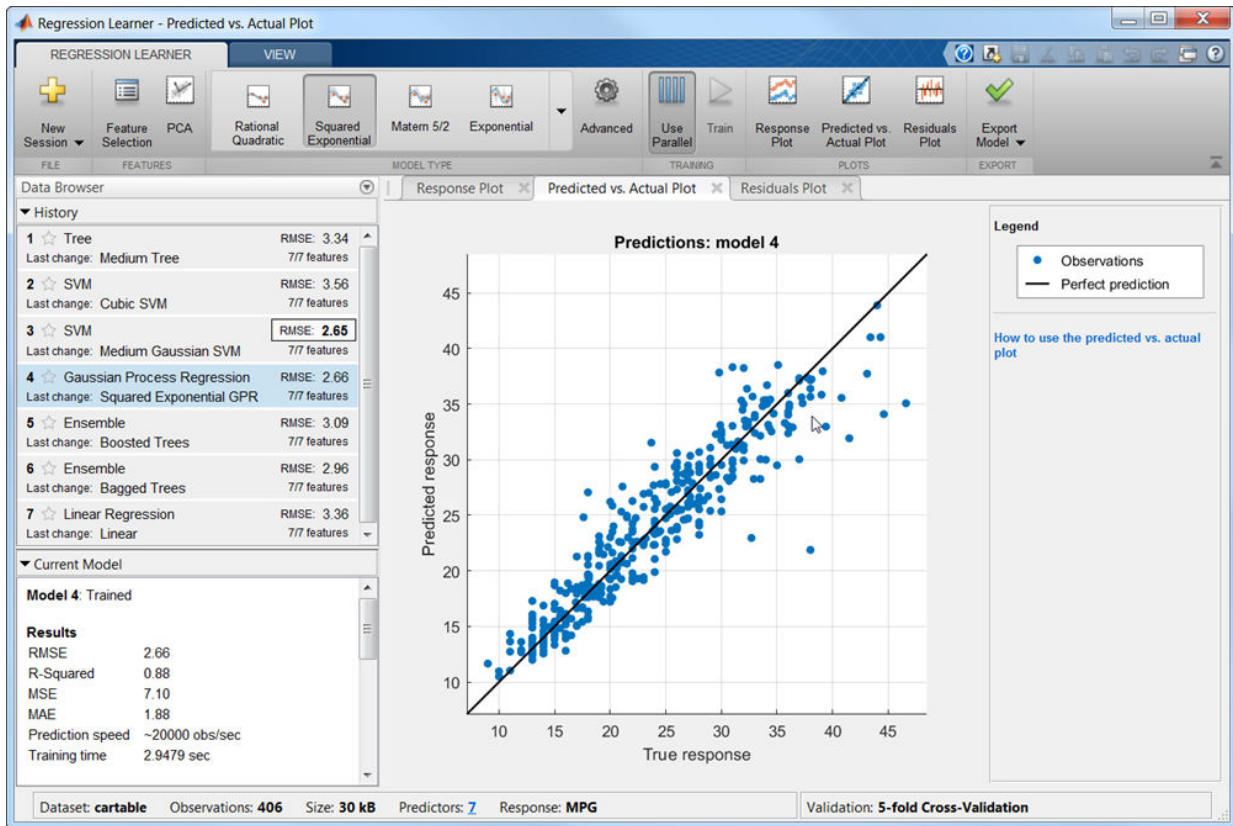
Use the Regression Learner app to train models to predict continuous data using supervised machine learning. The app lets you explore supervised machine learning interactively using various regression models.

- Automatically train a selection of models and help you choose the best model. Model types include linear regression models, regression trees, Gaussian process regression models, support vector machines, and ensembles of regression trees.
- Explore your data, select features, and visualize results.
- Export models to the workspace to make predictions with new data.

- Generate MATLAB code from the app to create scripts, train with new data, work with huge data sets, or modify the code for further analysis.

By default, the app protects against overfitting by applying cross-validation. Alternatively, you can choose holdout validation.

To learn more, see “Train Regression Models in Regression Learner App” on page 24-2.



For more options, you can use the command-line interface. See “Regression”.

## Train Neural Networks for Deep Learning

Neural Network Toolbox enables you to perform deep learning with convolutional neural networks for classification, regression, feature extraction, and transfer learning. The

toolbox provides simple MATLAB commands for creating and interconnecting the layers of a deep neural network. Examples and pretrained networks make it easy to use MATLAB for deep learning, even without extensive knowledge of advanced computer vision algorithms or neural networks.

To learn more, see “Deep Learning in MATLAB” (Neural Network Toolbox).

## See Also

### Related Examples

- “Train Regression Models in Regression Learner App” on page 24-2
- “Train Classification Models in Classification Learner App” on page 23-11
- “Cluster Analysis”
- “Credit Risk” (Financial Toolbox)
- “Deep Learning, Object Detection and Recognition” (Computer Vision System Toolbox)
- “Deep Learning in MATLAB” (Neural Network Toolbox)
- Machine Learning Made Easy (34 min 34 sec)



## Train Classification Models in Classification Learner App

### In this section...

“What Is Supervised Machine Learning?” on page 23-11

“Automated Classifier Training” on page 23-12

“Manual Classifier Training” on page 23-14

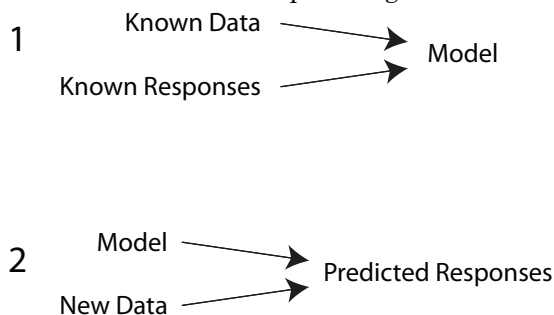
“Parallel Classifier Training” on page 23-15

“Compare and Improve Classification Models” on page 23-16

### What Is Supervised Machine Learning?

You can use Classification Learner to train models to classify data. Using this app, you can explore supervised machine learning using various classifiers. You can explore your data, select features, specify validation schemes, train models, and assess results. You can perform automated training to search for the best classification model type, including decision trees, discriminant analysis, support vector machines, logistic regression, nearest neighbors, and ensemble classification.

Perform supervised machine learning by supplying a known set of input data (observations or examples) and known responses to the data (i.e., labels or classes). Use the data to train a model that generates predictions for the response to new data. To use the model with new data, or to learn about programmatic classification, you can export the model to the workspace or generate MATLAB code to recreate the trained model.

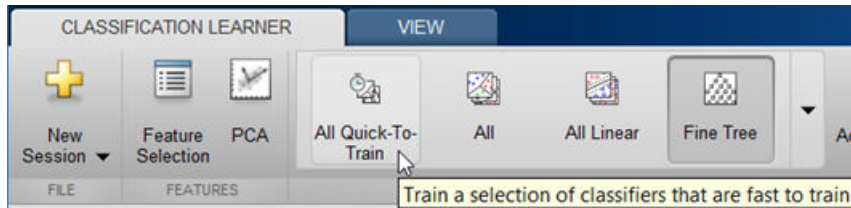


Get started by training a selection of model types. See “Automated Classifier Training” on page 23-12.

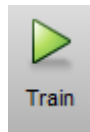
## Automated Classifier Training

You can use Classification Learner to automatically train a selection of different classification models on your data.

- Get started by automatically training multiple models at once. You can quickly try a selection of models, then explore promising models interactively.
  - If you already know what classifier type you want, train individual classifiers instead. See “Manual Classifier Training” on page 23-14.
- 1 On the **Apps** tab, in the **Math, Statistics and Optimization** group, click **Classification Learner**.
  - 2 Click **New Session** and select data from the workspace or from file. Specify a response variable and variables to use as predictors. See “Select Data and Validation for Classification Problem” on page 23-20.
  - 3 On the **Classification Learner** tab, in the **Model Type** section, click **All Quick-To-Train**. This option will train all the model presets available for your data set that are fast to fit.



4



Click **Train**.

---

**Note** The app trains models in parallel if you have Parallel Computing Toolbox. See “Parallel Classifier Training” on page 23-15.

---

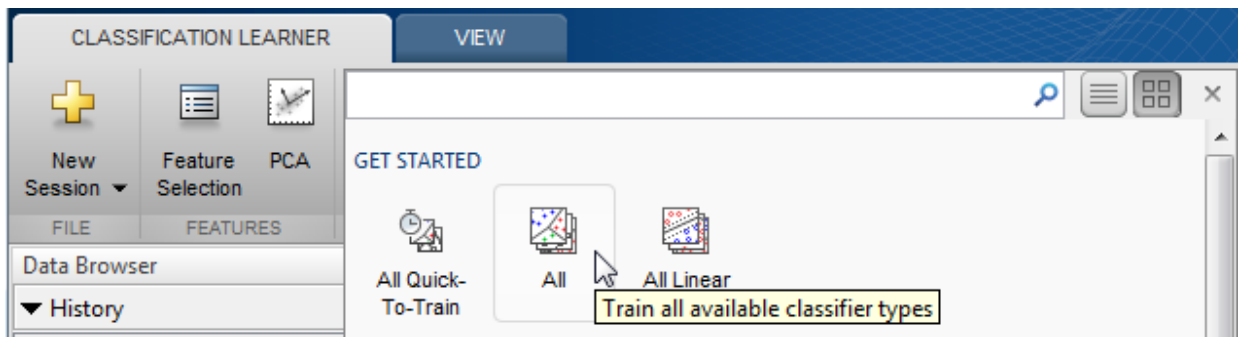
A selection of model types appears in the History list. When they finish training, the best percentage **Accuracy** score is highlighted in a box.

▼ History		
1.1	☆ Tree Last change: Fine Tree	Accuracy: 94.0% 3/3 features
1.2	☆ Tree Last change: Medium Tree	Accuracy: 94.0% 3/3 features
1.3	☆ Tree Last change: Coarse Tree	Accuracy: 94.0% 3/3 features
1.4	☆ KNN Last change: Fine KNN	Accuracy: 90.0% 3/3 features
1.5	☆ KNN Last change: Medium KNN	Accuracy: <b>94.7%</b> 3/3 features
1.6	☆ KNN Last change: Coarse KNN	Accuracy: 68.7% 3/3 features
1.7	☆ KNN Last change: Cosine KNN	Accuracy: 84.0% 3/3 features
1.8	☆ KNN Last change: Cubic KNN	Accuracy: <b>94.7%</b> 3/3 features
1.9	☆ KNN Last change: Weighted KNN	Accuracy: 93.3% 3/3 features

- Click models in the history list to explore results in the plots.

For next steps, see “Manual Classifier Training” on page 23-14 or “Compare and Improve Classification Models” on page 23-16.

- To try all the classifier model presets available for your data set, click **All**, then click **Train**.

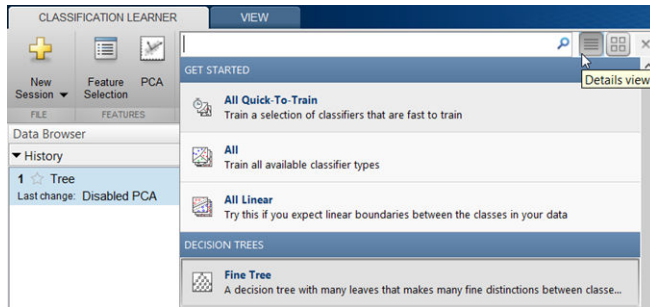


## Manual Classifier Training

If you want to explore individual model types, or if you already know what classifier type you want, you can train classifiers one at a time, or a train a group of the same type.

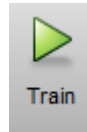
- 1 Choose a classifier. On the **Classification Learner** tab, in the **Model Type** section, click a classifier type. To see all available classifier options, click the arrow on the far right of the **Model Type** section to expand the list of classifiers. The options in the **Model Type** gallery are preset starting points with different settings, suitable for a range of different classification problems.

To read a description of each classifier, switch to the details view.



For more information on each option, see “Choose Classifier Options” on page 23-26.

2



After selecting a classifier, click **Train**.

Repeat to try different classifiers.

**Tip** Try decision trees and discriminants first. If the models are not accurate enough predicting the response, try other classifiers with higher flexibility. To avoid overfitting, look for a model of lower flexibility that provides sufficient accuracy.

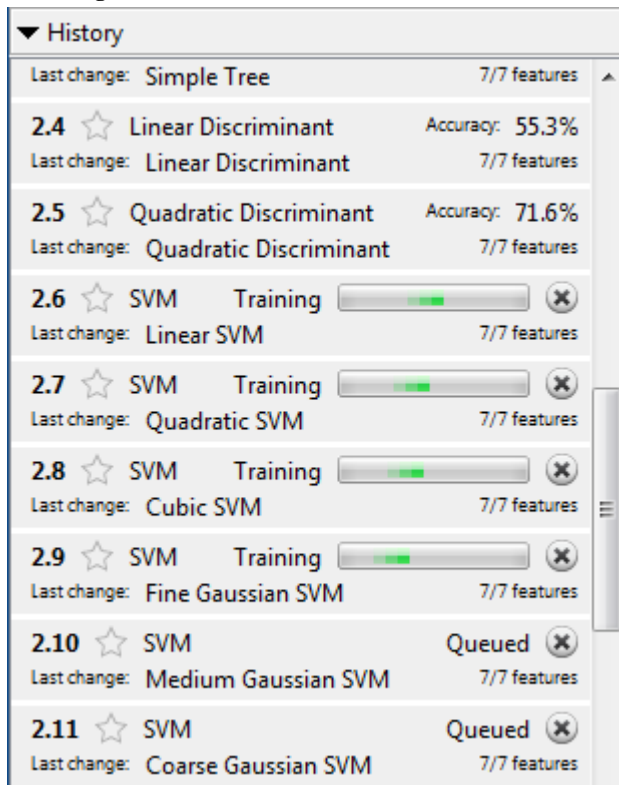
- 3 If you decide you want to try all model types or train a group of the same type, try one of the **All** options in the **Model Type** gallery.

For next steps, see “Compare and Improve Classification Models” on page 23-16

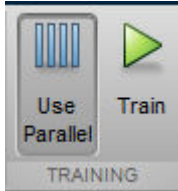
## Parallel Classifier Training

You can train models in parallel using Classification Learner if you have Parallel Computing Toolbox. When you train classifiers, the app automatically starts a parallel pool of workers, unless you turn off the default parallel preference **Automatically create a parallel pool**. If a pool is already open, the app uses it for training. Parallel training allows you to train multiple classifiers at once and continue working.

- 1 The first time you click **Train**, you see a dialog while the app opens a parallel pool of workers. After the pool opens, you can train multiple classifiers at once.
- 2 When classifiers are training in parallel, you see progress indicators on each training and queued model in the history list, and you can cancel individual models if you want. During training, you can examine results and plots from models, and initiate training of more classifiers.



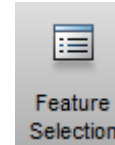
To control parallel training, toggle the **Use Parallel** button in the app toolstrip. The **Use Parallel** button is only available if you have Parallel Computing Toolbox.



If you have Parallel Computing Toolbox, then parallel training is available in Classification Learner, and you do not need to set the `UseParallel` option of the `statset` function. If you turn off the parallel preference to **Automatically create a parallel pool**, then the app will not start a pool for you without asking first.

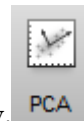
## Compare and Improve Classification Models

- 1 Click models in the history list to explore the results in the plots. Compare model performance by inspecting results in the scatter plot and confusion matrix. Examine the percentage accuracy reported in the history list for each model. See “Assess Classifier Performance in Classification Learner” on page 23-54.
- 2 Select the best model in the history list and then try including and excluding



different features in the model. Click Feature Selection.

Try the parallel coordinates plot to help you identify features to remove. See if you can improve the model by removing features with low predictive power. Specify predictors to include in the model, and train new models using the new options. Compare results among the models in the history list.



You can also try transforming features with PCA to reduce dimensionality.

See “Feature Selection and Feature Transformation Using Classification Learner App” on page 23-47.

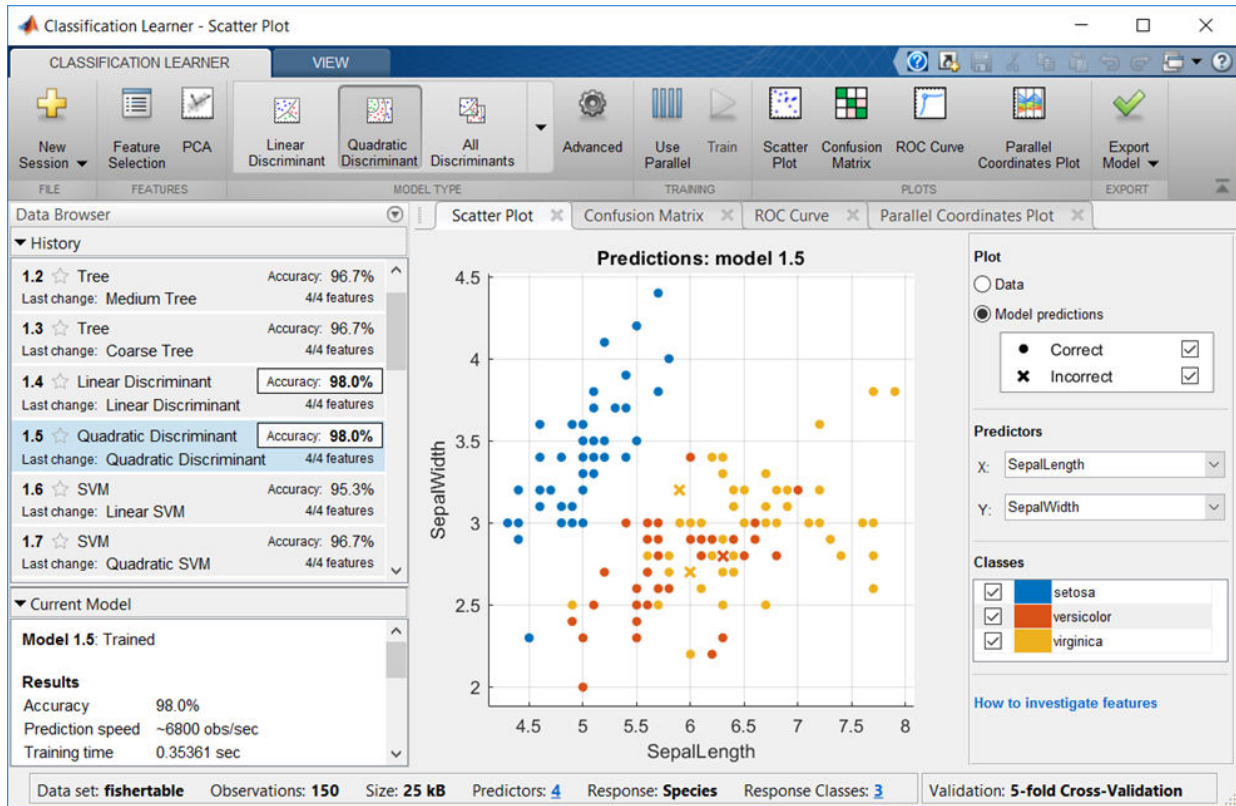
- 3 To improve the model further, you can try changing classifier parameter settings in the Advanced dialog box, and then train using the new options. To learn how to control model flexibility, see “Choose Classifier Options” on page 23-26.
- 4 If feature selection, PCA, or new parameter settings improve your model, try training **All** model types with the new settings. See if another model type does better with the new settings.

---

**Tip** To avoid overfitting, look for a model of lower flexibility that provides sufficient accuracy. For example, look for simple models such as decision trees and discriminants that are fast and easy to interpret. If the models are not accurate enough predicting the response, choose other classifiers with higher flexibility, such as ensembles. To learn about the model flexibility, see “Choose Classifier Options” on page 23-26.

---

The figure shows the app with a history list containing various classifier types.



**Tip** For a step-by-step example comparing different classifiers, see “Train Decision Trees Using Classification Learner App” on page 23-67.

For next steps, generate code to train the model with different data, or export trained models to the workspace to make predictions using new data. See “Export Classification Model to Predict New Data” on page 23-61.



## See Also

### Related Examples

- “Select Data and Validation for Classification Problem” on page 23-20
- “Choose Classifier Options” on page 23-26
- “Feature Selection and Feature Transformation Using Classification Learner App” on page 23-47
- “Assess Classifier Performance in Classification Learner” on page 23-54
- “Export Classification Model to Predict New Data” on page 23-61
- “Train Decision Trees Using Classification Learner App” on page 23-67
- “Machine Learning in MATLAB” on page 23-2

## Select Data and Validation for Classification Problem

In this section...
“Select Data from the Workspace” on page 23-20
“Import Data from File” on page 23-21
“Example Data for Classification” on page 23-21
“Choose Validation Scheme” on page 23-23

### Select Data from the Workspace

---

**Tip** In Classification Learner, tables are the easiest way to use your data, because they can contain numeric and label data. Use the Import Tool to bring your data into the MATLAB workspace as a table, or use the table functions to create a `table` from workspace variables. See “Tables” (MATLAB).

If your predictors are a matrix and the response is a vector, combine them into a table using the `table` function.

---

- 1 Load your data into the MATLAB workspace.

Predictor and response variables can be a categorical array, cell array of character vectors, character array, logical vector, or a numeric vector.

For example data sets, see “Example Data for Classification” on page 23-21.

- 2 On the **Apps** tab, click **Classification Learner**.
- 3 In Classification Learner, on the **Classification Learner** tab, in the **File** section, click **New Session**.
- 4 In the New Session dialog box, select a table or matrix from the workspace variables.

If you select a matrix, choose whether to use rows or columns for observations by clicking the option buttons.

- 5 Observe the roles the app selects for the variables based on their data type. The app tries to select a suitable response variable, and all other variables are predictors. Change the selections if needed. Add or remove predictors using the check boxes. Add or remove all predictors by clicking **Add All** or **Remove All**. You can also add

or remove multiple predictors by selecting them in the table, and then clicking **Add N** or **Remove N**, where **N** is the number of selected predictors. The **Add All** and **Remove All** buttons change to **Add N** and **Remove N** when you select multiple predictors.

- To accept the default validation scheme and continue, click **Start Session**. The default validation option is 5-fold cross-validation, which protects against overfitting.

---

**Tip** If you have a large data set you might want to switch to holdout validation. To learn more, see “Choose Validation Scheme” on page 23-23.

---

For next steps, see “Train Classification Models in Classification Learner App” on page 23-11.

## Import Data from File

- On the **Classification Learner** tab, in the **File** section, select **New Session > From File**.
- Select a file type in the list, such as spreadsheets, text files, or comma separated values (.csv) files, or select **All Files** to browse for other file types such as .dat.

## Example Data for Classification

To get started using Classification Learner, try the following example data sets.

Name	Size	Description
Fisher Iris	Number of predictors: 4 Number of observations: 150 Number of classes: 3 Response: species	Measurements from three species of iris. Try to classify the species.  For a step-by-step example, see “Train Decision Trees Using Classification Learner App” on page 23-67.
	Create a table from the .csv file:  <code>fishertable = readtable('fisheriris.csv');</code>	

Name	Size	Description
Credit Rating	Number of predictors: 6 Number of observations: 3932 Number of classes: 7 Response: Rating	Financial ratios and industry sectors information for a list of corporate customers. The response variable consists of credit ratings (AAA, AA, A, BBB, BB, B, CCC) assigned by a rating agency.
	Create a table from the CreditRating_Historical.dat file:  <code>creditrating = readtable('CreditRating_Historical.dat');</code>	
Cars	Number of predictors: 7 Number of observations: 100 Number of classes: 7 Response: Origin	Measurements of cars, in 1970, 1976, and 1982. Try to classify the country of origin.
	Create a table from variables in the carsmall.mat file:  <code>load carsmall cartable = table(Acceleration, Cylinders, Displacement, ... Horsepower, Model_Year, MPG, Weight, Origin);</code>	
Arrhythmia	Number of predictors: 279 Number of observations: 150 Number of classes: 16 Response: Class (Y)	Patient information and response variables that indicate the presence and absence of cardiac arrhythmia. Misclassifying a patient as "normal" has more severe consequences than false positives classified as "has arrhythmia".
	Create a table from the .mat file:  <code>load arrhythmia Arrhythmia = array2table(X); Arrhythmia.Class = categorical(Y);</code>	
Ovarian Cancer	Number of predictors: 4000 Number of observations: 216 Number of classes: 2 Response: Group	Ovarian cancer data generated using the WCX2 protein array. Includes 95 controls and 121 ovarian cancers.

Name	Size	Description
	Create a table from the .mat file:  <pre>load ovariancancer ovariancancer = array2table(obs); ovariancancer.Group = categorical(grp);</pre>	
Ionosphere	Number of predictors: 34 Number of observations: 351 Number of classes: 2 Response: Group (Y)	Signals from a phased array of 16 high-frequency antennas. Good (“g”) returned radar signals are those showing evidence of some type of structure in the ionosphere. Bad (“b”) signals are those that pass through the ionosphere.
	Create a table from the .mat file:  <pre>load ionosphere ionosphere = array2table(X); ionosphere.Group = Y;</pre>	

## Choose Validation Scheme

Choose a validation method to examine the predictive accuracy of the fitted models. Validation estimates model performance on new data compared to the training data, and helps you choose the best model. Validation protects against overfitting. Choose a validation scheme before training any models, so that you can compare all the models in your session using the same validation scheme.

---

**Tip** Try the default validation scheme and click **Start Session** to continue. The default option is 5-fold cross-validation, which protects against overfitting.

If you have a large data set and training models takes too long using cross-validation, reimport your data and try the faster holdout validation instead.

---

- **Cross-Validation:** Select a number of folds (or divisions) to partition the data set using the slider control.

If you choose  $k$  folds, then the app:

- 1 Partitions the data into  $k$  disjoint sets or folds
- 2 For each fold:
  - a Trains a model using the out-of-fold observations
  - b Assesses model performance using in-fold data
- 3 Calculates the average test error over all folds

This method gives a good estimate of the predictive accuracy of the final model trained with all the data. It requires multiple fits but makes efficient use of all the data, so it is recommended for small data sets.

- **Holdout Validation:** Select a percentage of the data to use as a test set using the slider control. The app trains a model on the training set and assesses its performance with the test set. The model used for validation is based on only a portion of the data, so **Holdout Validation** is recommended only for large data sets. The final model is trained with the full data set.
- **No Validation:** No protection against overfitting. The app uses all of the data for training and computes the error rate on the same data. Without any test data, you get an unrealistic estimate of the model's performance on new data. That is, the training sample accuracy is likely to be unrealistically high, and the predictive accuracy is likely to be lower.

To help you avoid overfitting to the training data, choose a validation scheme instead.

---

**Note** The validation scheme only affects the way that Classification Learner computes validation metrics. The final model is always trained using the full data set.

---

All the classification models you train after selecting data use the same validation scheme that you select in this dialog box. You can compare all the models in your session using the same validation scheme.

To change the validation selection and train new models, you can select data again, but you lose any trained models. The app warns you that importing data starts a new session. Save any trained models you want to keep to the workspace, and then import the data.

For next steps training models, see “Train Classification Models in Classification Learner App” on page 23-11.

## See Also

### Related Examples

- “Train Classification Models in Classification Learner App” on page 23-11
- “Choose Classifier Options” on page 23-26
- “Feature Selection and Feature Transformation Using Classification Learner App” on page 23-47
- “Assess Classifier Performance in Classification Learner” on page 23-54
- “Export Classification Model to Predict New Data” on page 23-61
- “Train Decision Trees Using Classification Learner App” on page 23-67

## Choose Classifier Options

### In this section...

“Choose a Classifier Type” on page 23-26

“Decision Trees” on page 23-30

“Discriminant Analysis” on page 23-34

“Logistic Regression” on page 23-35

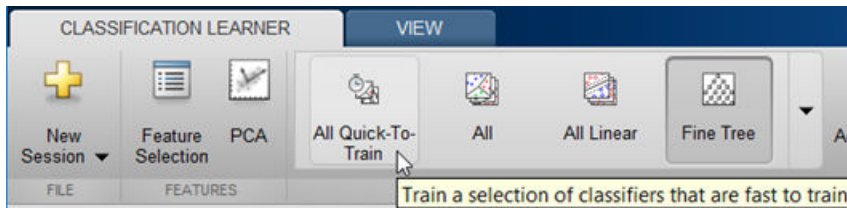
“Support Vector Machines” on page 23-35

“Nearest Neighbor Classifiers” on page 23-39

“Ensemble Classifiers” on page 23-42

## Choose a Classifier Type

You can use Classification Learner to automatically train a selection of different classification models on your data. Use automated training to quickly try a selection of model types, then explore promising models interactively. To get started, try these options first:



Get Started Classifier Buttons	Description
<b>All Quick-To-Train</b>	Try this first. The app will train all the model types available for your data set that are typically fast to fit.
<b>All Linear</b>	Try this if you expect linear boundaries between the classes in your data. This option fits only Linear SVM and Linear Discriminant.
<b>All</b>	Use this to train all available model types. Trains every type regardless of any prior trained models. Can be time-consuming.



See “Automated Classifier Training” on page 23-12.

If you want to explore classifiers one at a time, or you already know what classifier type you want, you can select individual models or train a group of the same type. To see all available classifier options, on the **Classification Learner** tab, click the arrow on the far right of the **Model Type** section to expand the list of classifiers. The options in the **Model Type** gallery are preset starting points with different settings, suitable for a range of different classification problems.







For help choosing the best classifier type for your problem, see the table showing typical characteristics of different supervised learning algorithms. Use the table as a guide for your final choice of algorithms. Decide on the tradeoff you want in speed, memory usage, flexibility, and interpretability. The best classifier type depends on your data.

---

**Tip** To avoid overfitting, look for a model of lower flexibility that provides sufficient accuracy. For example, look for simple models such as decision trees and discriminants that are fast and easy to interpret. If the models are not accurate enough predicting the response, choose other classifiers with higher flexibility, such as ensembles. To control flexibility, see the details for each classifier type.

---

### Characteristics of Classifier Types

Classifier	Prediction Speed	Memory Usage	Interpretability
“Decision Trees” on page 23-30 	Fast	Small	Easy
“Discriminant Analysis” on page 23-34 	Fast	Small for linear, large for quadratic	Easy
“Logistic Regression” on page 23-35 	Fast	Medium	Easy
“Support Vector Machines” on page 23-35 	Medium for linear. Slow for others.	Medium for linear. All others: medium for multiclass, large for binary.	Easy for Linear SVM. Hard for all other kernel types.
<b>Note</b> Supports C code generation for prediction.			
“Nearest Neighbor Classifiers” on page 23-39 	Slow for cubic. Medium for others.	Medium	Hard
“Ensemble Classifiers” on page 23-42 	Fast to medium depending on choice of algorithm	Low to high depending on choice of algorithm	Hard

The tables on this page describe general characteristics of speed and memory usage for all the preset classifiers. The classifiers were tested with various data sets (up to 7000 observations, 80 predictors, and 50 classes), and the results define the following groups:

#### Speed

- Fast 0.01 second

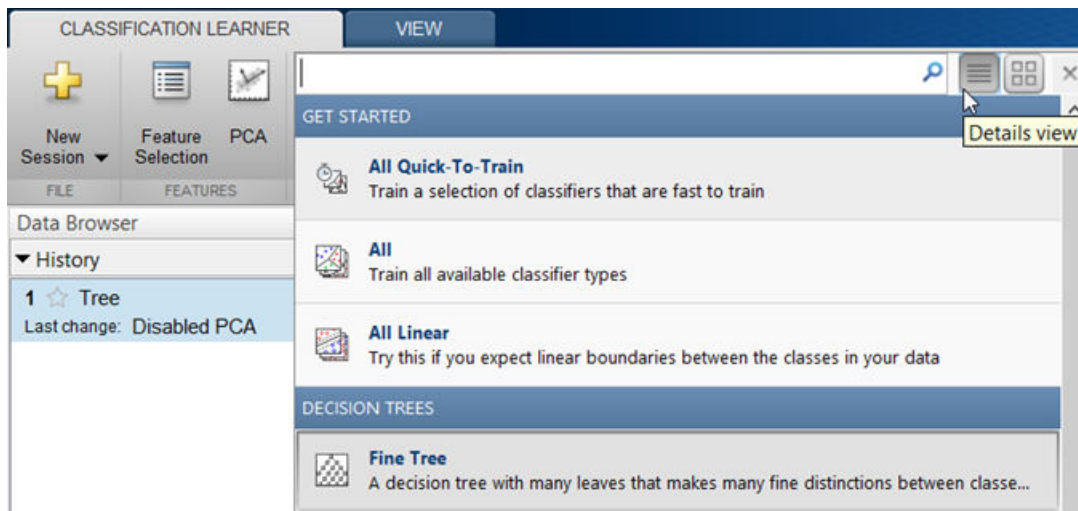
- Medium 1 second
- Slow 100 seconds

#### Memory

- Small 1MB
- Medium 4MB
- Large 100MB

These tables provide a general guide. Your results depend on your data and the speed of your machine.

To read a description of each classifier in Classification Learner, switch to the details view.




---

**Tip** After you choose a classifier type (e.g., decision trees), try training using each of the classifiers. The options in the **Model Type** gallery are starting points with different settings. Try them all to see which option produces the best model with your data.

---

For workflow instructions, see “Train Classification Models in Classification Learner App” on page 23-11.

### Categorical Predictor Support


In Classification Learner, the **Model Type** gallery only shows classifier types that support your selected data.



Classifier	All predictors numeric	All predictors categorical	Some categorical, some numeric
Decision Trees	Yes	Yes	Yes
Discriminant Analysis	Yes	No	No
Logistic Regression	Yes	Yes	Yes
SVM	Yes	Yes	Yes
Nearest Neighbor	Euclidean distance only	Hamming distance only	No
Ensembles	Yes	Yes, except Subspace Discriminant	Yes, except any Subspace

### Decision Trees

Decision trees are easy to interpret, fast for fitting and prediction, and low on memory usage, but they can have low predictive accuracy. Try to grow simpler trees to prevent overfitting. Control the depth with the **Maximum number of splits** setting.

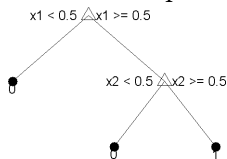
**Tip** Model flexibility increases with the **Maximum number of splits** setting.

Classifier Type	Prediction Speed	Memory Usage	Interpretability	Model Flexibility
Coarse Tree 	Fast	Small	Easy	Low.  Few leaves to make coarse distinctions between classes (maximum number of splits is 4).

Classifier Type	Prediction Speed	Memory Usage	Interpretability	Model Flexibility
Medium Tree 	Fast	Small	Easy	Medium  Medium number of leaves for finer distinctions between classes (maximum number of splits is 20).
Fine Tree 	Fast	Small	Easy	High  Many leaves to make many fine distinctions between classes (maximum number of splits is 100).

**Tip** In the **Model Type** gallery click All Trees to try each of the decision tree options . Train them all to see which settings produce the best model with your data. Select the best model in the History list. To try to improve your model, try feature selection, and then try changing some advanced options.

You train classification trees to predict responses to data. To predict a response, follow the decisions in the tree from the root (beginning) node down to a leaf node. The leaf node contains the response. Statistics and Machine Learning Toolbox trees are binary. Each step in a prediction involves checking the value of one predictor (variable). For example, here is a simple classification tree:

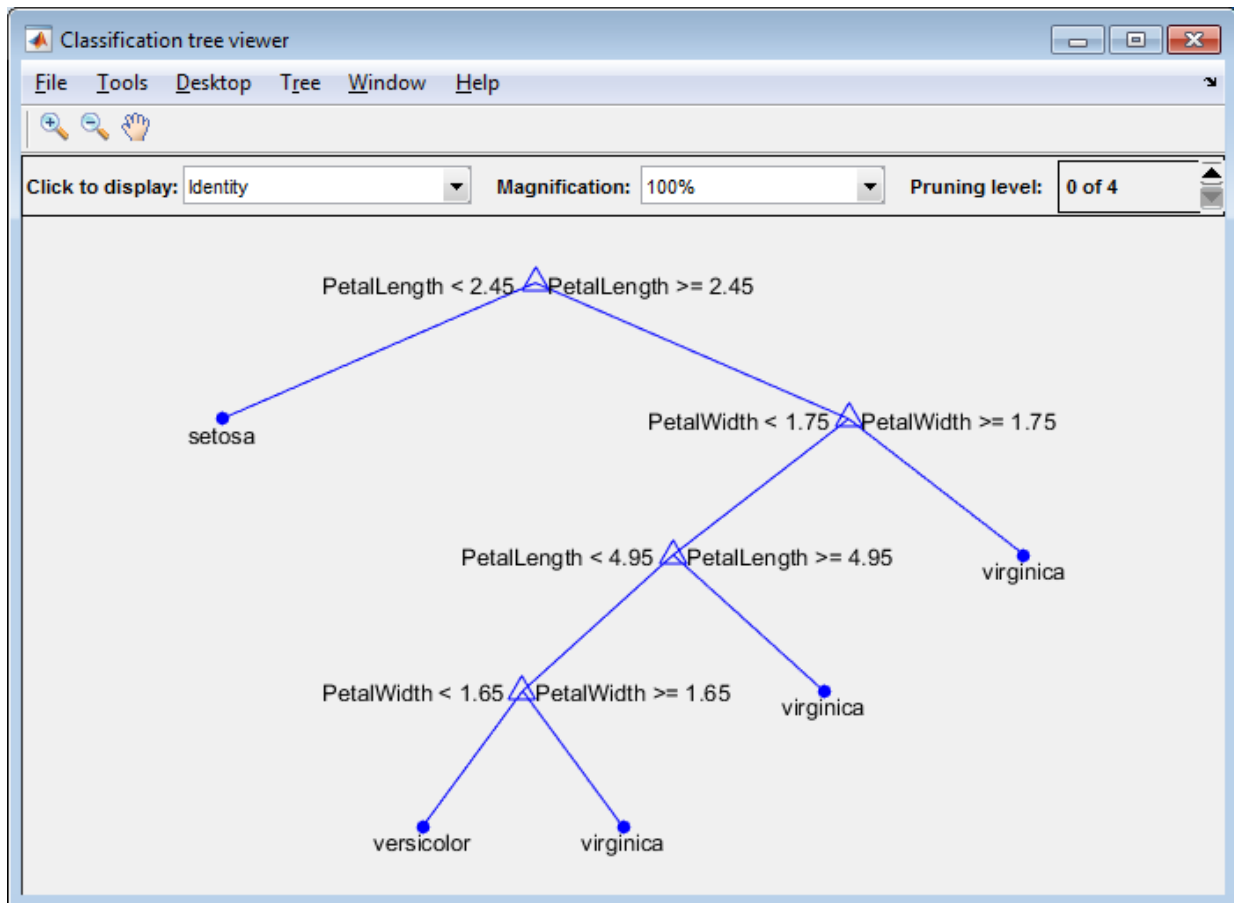


This tree predicts classifications based on two predictors,  $x_1$  and  $x_2$ . To predict, start at the top node. At each decision, check the values of the predictors to decide which branch to follow. When the branches reach a leaf node, the data is classified either as type 0 or 1.

You can visualize your decision tree model by exporting the model from the app, and then entering:

```
view(trainedModel.ClassificationTree, 'Mode', 'graph')
```

The figure shows an example fine tree trained with the `fisheriris` data.



---

**Tip** For an example, see “Train Decision Trees Using Classification Learner App” on page 23-67.

---

## Advanced Tree Options

Classification trees in Classification Learner use the `fitctree` function. You can set these options:

- **Maximum number of splits**

Specify the maximum number of splits or branch points to control the depth of your tree. When you grow a decision tree, consider its simplicity and predictive power. To change the number of splits, click the buttons or enter a positive integer value in the **Maximum number of splits** box.

- A fine tree with many leaves is usually highly accurate on the training data. However, the tree might not show comparable accuracy on an independent test set. A leafy tree tends to overtrain, and its validation accuracy is often far lower than its training (or resubstitution) accuracy.
- In contrast, a coarse tree does not attain high training accuracy. But a coarse tree can be more robust in that its training accuracy can approach that of a representative test set. Also, a coarse tree is easy to interpret.

- **Split criterion**

Specify the split criterion measure for deciding when to split nodes. Try each of the three settings to see if they improve the model with your data.

Split criterion options are Gini's diversity index, Twoing rule, or Maximum deviance reduction (also known as cross entropy).

The classification tree tries to optimize to pure nodes containing only one class. Gini's diversity index (the default) and the deviance criterion measure node impurity. The twoing rule is a different measure for deciding how to split a node, where maximizing the twoing rule expression increases node purity.

For details of these split criteria, see `ClassificationTree` “Definitions” on page 32-578.

- **Surrogate decision splits** — Only for missing data.

Specify surrogate use for decision splits. If you have data with missing values, use surrogate splits to improve the accuracy of predictions.



When you set **Surrogate decision splits** to `On`, the classification tree finds at most 10 surrogate splits at each branch node. To change the number, click the buttons or enter a positive integer value in the **Maximum surrogates per node** box.

When you set **Surrogate decision splits** to `Find All`, the classification tree finds all surrogate splits at each branch node. The `Find All` setting can use considerable time and memory.

## Discriminant Analysis

Discriminant analysis is a popular first classification algorithm to try because it is fast, accurate and easy to interpret. Discriminant analysis is good for wide datasets.

Discriminant analysis assumes that different classes generate data based on different Gaussian distributions. To train a classifier, the fitting function estimates the parameters of a Gaussian distribution for each class.

Classifier Type	Prediction Speed	Memory Usage	Interpretability	Model Flexibility
Linear Discriminant 	Fast	Small	Easy	Low.  Creates linear boundaries between classes.
Quadratic Discriminant 	Fast	Large	Easy	Low.  Creates nonlinear boundaries between classes (ellipse, parabola or hyperbola).

### Advanced Discriminant Options


Discriminant analysis in Classification Learner uses the `fitcdiscr` function. For both linear and quadratic discriminants, you can change the **Covariance structure** option. If you have predictors with zero variance or if any of the covariance matrices of your



predictors are singular, training can fail using the default, `Full` covariance structure. If training fails, select the `Diagonal` covariance structure instead.

## Logistic Regression

If you have 2 classes, logistic regression is a popular simple classification algorithm to try because it is easy to interpret. The classifier models the class probabilities as a function of the linear combination of predictors.

Classifier Type	Prediction Speed	Memory Usage	Interpretability	Model Flexibility
Logistic Regression 	Fast	Medium	Easy	Low.  You cannot change any parameters to control model flexibility.

Logistic regression in Classification Learner uses the `fitglm` function. You cannot set any options for this classifier in the app.


## Support Vector Machines






In Classification Learner, you can train SVMs when your data has two or more classes.

---

**Note** After you train an SVM model, you can generate C code for prediction. Requires MATLAB Coder. See “Generate C Code for Prediction” on page 23-64.

---

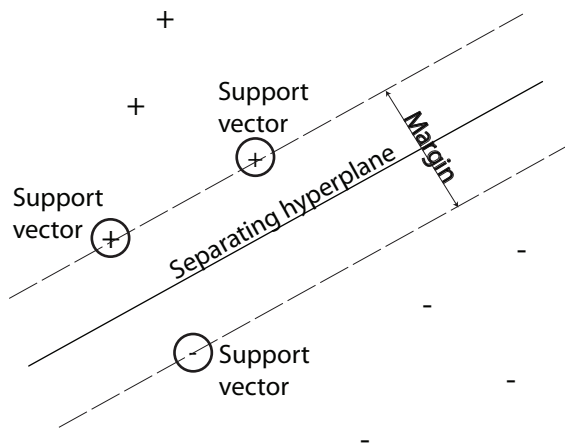
Classifier Type	Prediction Speed	Memory Usage	Interpretability	Model Flexibility
Linear SVM 	Binary: Fast  Multiclass: Medium	Medium	Easy	Low Makes a simple linear separation between classes.

Classifier Type	Prediction Speed	Memory Usage	Interpretability	Model Flexibility
Quadratic SVM 	Binary: Fast  Multiclass: Slow	Binary: Medium  Multiclass: Large	Hard	Medium
Cubic SVM 	Binary: Fast  Multiclass: Slow	Binary: Medium  Multiclass: Large	Hard	Medium
Fine Gaussian SVM 	Binary: Fast  Multiclass: Slow	Binary: Medium  Multiclass: Large	Hard	High — decreases with kernel scale setting.  Makes finely detailed distinctions between classes, with kernel scale set to $\sqrt{P}/4$ .
Medium Gaussian SVM 	Binary: Fast  Multiclass: Slow	Binary: Medium  Multiclass: Large	Hard	Medium  Medium distinctions, with kernel scale set to $\sqrt{P}$ .
Coarse Gaussian SVM 	Binary: Fast  Multiclass: Slow	Binary: Medium  Multiclass: Large	Hard	Low  Makes coarse distinctions between classes, with kernel scale set to $\sqrt{P} * 4$ , where P is the number of predictors.

**Tip** Try training each of the support vector machine options in the **Model Type** gallery. Train them all to see which settings produce the best model with your data. Select the best model in the History list. To try to improve your model, try feature selection, and then try changing some advanced options.

An SVM classifies data by finding the best hyperplane that separates data points of one class from those of the other class. The best hyperplane for an SVM means the one with the largest margin between the two classes. Margin means the maximal width of the slab parallel to the hyperplane that has no interior data points.

The support vectors are the data points that are closest to the separating hyperplane; these points are on the boundary of the slab. The following figure illustrates these definitions, with + indicating data points of type 1, and - indicating data points of type -1.



SVMs can also use a soft margin, meaning a hyperplane that separates many, but not all data points.

For an example, see “Train Support Vector Machines Using Classification Learner App” on page 23-88.

### Advanced SVM Options

If you have exactly two classes, Classification Learner uses the `fitcsvm` function to train the classifier. If you have more than two classes, the app uses the `fitcecoc` function to reduce the multiclass classification problem to a set of binary classification subproblems, with one SVM learner for each subproblem. To examine the code for the binary and multiclass classifier types, you can generate code from your trained classifiers in the app.

You can set these options in the app:

- **Kernel function**

Specify the Kernel function to compute the classifier.

- Linear kernel, easiest to interpret
- Gaussian or Radial Basis Function (RBF) kernel
- Quadratic
- Cubic

- **Box constraint level**

Specify the box constraint to keep the allowable values of the Lagrange multipliers in a box, a bounded region.

To tune your SVM classifier, try increasing the box constraint level. Click the buttons or enter a positive scalar value in the **Box constraint level** box. Increasing the box constraint level can decrease the number of support vectors, but also can increase training time.

The Box Constraint parameter is the soft-margin penalty known as  $C$  in the primal equations, and is a hard “box” constraint in the dual equations.

- **Kernel scale mode**

Specify manual kernel scaling if desired.

When you set **Kernel scale mode** to `Auto`, then the software uses a heuristic procedure to select the scale value. The heuristic procedure uses subsampling. Therefore, to reproduce results, set a random number seed using `rng` before training the classifier.

When you set **Kernel scale mode** to `Manual`, you can specify a value. Click the buttons or enter a positive scalar value in the **Manual kernel scale** box. The software divides all elements of the predictor matrix by the value of the kernel scale. Then, the software applies the appropriate kernel norm to compute the Gram matrix.

---

**Tip** Model flexibility decreases with the kernel scale setting.

---

- **Multiclass method**

Only for data with 3 or more classes. This method reduces the multiclass classification problem to a set of binary classification subproblems, with one SVM

learner for each subproblem. *One-vs-One* trains one learner for each pair of classes. It learns to distinguish one class from the other. *One-vs-All* trains one learner for each class. It learns to distinguish one class from all others.

- **Standardize data**

Specify whether to scale each coordinate distance. If predictors have widely different scales, standardizing can improve the fit.




## Nearest Neighbor Classifiers




Nearest neighbor classifiers typically have good predictive accuracy in low dimensions, but might not in high dimensions. They have high memory usage, and are not easy to interpret.

---

**Tip** Model flexibility decreases with the **Number of neighbors** setting.

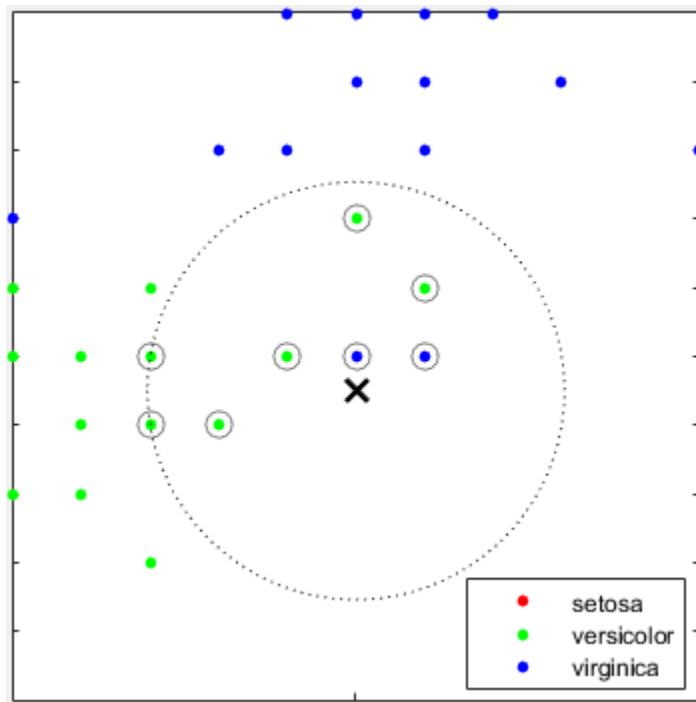
---

Classifier Type	Prediction Speed	Memory Usage	Interpretability	Model Flexibility
Fine KNN 	Medium	Medium	Hard	Finely detailed distinctions between classes. The number of neighbors is set to 1.
Medium KNN 	Medium	Medium	Hard	Medium distinctions between classes. The number of neighbors is set to 10.
Coarse KNN 	Medium	Medium	Hard	Coarse distinctions between classes. The number of neighbors is set to 100.

Classifier Type	Prediction Speed	Memory Usage	Interpretability	Model Flexibility
Cosine KNN 	Medium	Medium	Hard	Medium distinctions between classes, using a Cosine distance metric. The number of neighbors is set to 10.
Cubic KNN 	Slow	Medium	Hard	Medium distinctions between classes, using a cubic distance metric. The number of neighbors is set to 10.
Weighted KNN 	Medium	Medium	Hard	Medium distinctions between classes, using a distance weight. The number of neighbors is set to 10.

**Tip** Try training each of the nearest neighbor options in the **Model Type** gallery. Train them all to see which settings produce the best model with your data. Select the best model in the History list. To try to improve your model, try feature selection, and then (optionally) try changing some advanced options.

What is  $k$ -Nearest Neighbor classification? Categorizing query points based on their distance to points (or neighbors) in a training dataset can be a simple yet effective way of classifying new points. You can use various metrics to determine the distance. Given a set  $X$  of  $n$  points and a distance function,  $k$ -nearest neighbor ( $k$ NN) search lets you find the  $k$  closest points in  $X$  to a query point or set of points.  $k$ NN-based algorithms are widely used as benchmark machine learning rules.



For an example, see “Train Nearest Neighbor Classifiers Using Classification Learner App” on page 23-92.

### Advanced KNN Options

Nearest Neighbor classifiers in Classification Learner use the `fitcknn` function. You can set these options:

- **Number of neighbors**

Specify the number of nearest neighbors to find for classifying each point when predicting. Specify a fine (low number) or coarse classifier (high number) by changing the number of neighbors. For example, a fine KNN uses one neighbor, and a coarse KNN uses 100. Many neighbors can be time consuming to fit.

- **Distance metric**

You can use various metrics to determine the distance to points. For definitions, see the class `ClassificationKNN`.

- **Distance weight**

Specify the distance weighting function. You can choose `Equal` (no weights), `Inverse` (weight is  $1/\text{distance}$ ), or `Squared Inverse` (weight is  $1/\text{distance}^2$ ).

- **Standardize data**

Specify whether to scale each coordinate distance. If predictors have widely different scales, standardizing can improve the fit.

## Ensemble Classifiers


Ensemble classifiers meld results from many weak learners into one high-quality ensemble model. Qualities depend on the choice of algorithm.

---





**Tip** Model flexibility increases with the **Number of learners** setting.

All ensemble classifiers tend to be slow to fit because they often need many learners.

---

Classifier Type	Prediction Speed	Memory Usage	Interpretability	Ensemble Method	Model Flexibility
Boosted Trees 	Fast	Low	Hard	AdaBoost, with Decision Tree learners	Medium to high — increases with <b>Number of learners</b> or <b>Maximum number of splits</b> setting.  <hr/> <b>Tip</b> Boosted trees can usually do better than bagged, but might require parameter tuning and more learners



Classifier Type	Prediction Speed	Memory Usage	Interpretability	Ensemble Method	Model Flexibility
Bagged Trees 	Medium	High	Hard	Random forest Bag, with Decision Tree learners	High — increases with <b>Number of learners</b> setting. <hr/> <b>Tip</b> Try this classifier first.
Subspace Discriminant 	Medium	Low	Hard	Subspace, with Discriminant learners	Medium — increases with <b>Number of learners</b> setting.  Good for many predictors
Subspace KNN 	Medium	Medium	Hard	Subspace, with Nearest Neighbor learners	Medium — increases with <b>Number of learners</b> setting.  Good for many predictors
RUSBoost Trees 	Fast	Low	Hard	RUSBoost, with Decision Tree learners	Medium — increases with <b>Number of learners</b> or <b>Maximum number of splits</b> setting.  Good for skewed data (with many more observations of 1 class)

Classifier Type	Prediction Speed	Memory Usage	Interpretability	Ensemble Method	Model Flexibility
GentleBoost or LogitBoost — not available in the <b>Model Type</b> gallery. If you have 2 class data, select manually.	Fast	Low	Hard	GentleBoost or LogitBoost, with Decision Tree learners Choose Boosted Trees and change to GentleBoost method.	Medium — increases with <b>Number of learners</b> or <b>Maximum number of splits</b> setting.  For binary classification only

Bagged trees use Breiman's 'random forest' algorithm. For reference, see Breiman, L. Random Forests. *Machine Learning* 45, pp. 5–32, 2001.

### Tips

- Try bagged trees first. Boosted trees can usually do better but might require searching many parameter values, which is time-consuming.
- Try training each of the ensemble classifier options in the **Model Type** gallery. Train them all to see which settings produce the best model with your data. Select the best model in the History list. To try to improve your model, try feature selection, PCA, and then (optionally) try changing some advanced options.
- For boosting ensemble methods, you can get fine detail with either deeper trees or larger numbers of shallow trees. As with single tree classifiers, deep trees can cause overfitting. You need to experiment to choose the best tree depth for the trees in the ensemble, in order to trade-off data fit with tree complexity. Use the **Number of learners** and **Maximum number of splits** settings.

For an example, see “Train Ensemble Classifiers Using Classification Learner App” on page 23-96.

### Advanced Ensemble Options

Ensemble classifiers in Classification Learner use the `fitcensemble` function. You can set these options:

- For help choosing **Ensemble method** and **Learner type**, see the Ensemble table. Try the presets first.

- **Maximum number of splits**

For boosting ensemble methods, specify the maximum number of splits or branch points to control the depth of your tree learners. Many branches tend to overfit, and simpler trees can be more robust and easy to interpret. Experiment to choose the best tree depth for the trees in the ensemble.

- **Number of learners**

Try changing the number of learners to see if you can improve the model. Many learners can produce high accuracy, but can be time consuming to fit. Start with a few dozen learners, and then inspect the performance. An ensemble with good predictive power can need a few hundred learners.

- **Learning rate**

Specify the learning rate for shrinkage. If you set the learning rate to less than 1, the ensemble requires more learning iterations but often achieves better accuracy. 0.1 is a popular choice.

- **Subspace dimension**

For subspace ensembles, specify the number of predictors to sample in each learner. The app chooses a random subset of the predictors for each learner. The subsets chosen by different learners are independent.

For next steps training models, see “Train Classification Models in Classification Learner App” on page 23-11.

## See Also

### Related Examples

- “Train Classification Models in Classification Learner App” on page 23-11
- “Select Data and Validation for Classification Problem” on page 23-20
- “Feature Selection and Feature Transformation Using Classification Learner App” on page 23-47
- “Assess Classifier Performance in Classification Learner” on page 23-54

- “Export Classification Model to Predict New Data” on page 23-61
- “Train Decision Trees Using Classification Learner App” on page 23-67

# Feature Selection and Feature Transformation Using Classification Learner App

## In this section...

“Investigate Features in the Scatter Plot” on page 23-47

“Select Features to Include” on page 23-49

“Transform Features with PCA in Classification Learner” on page 23-50

“Investigate Features in the Parallel Coordinates Plot” on page 23-50

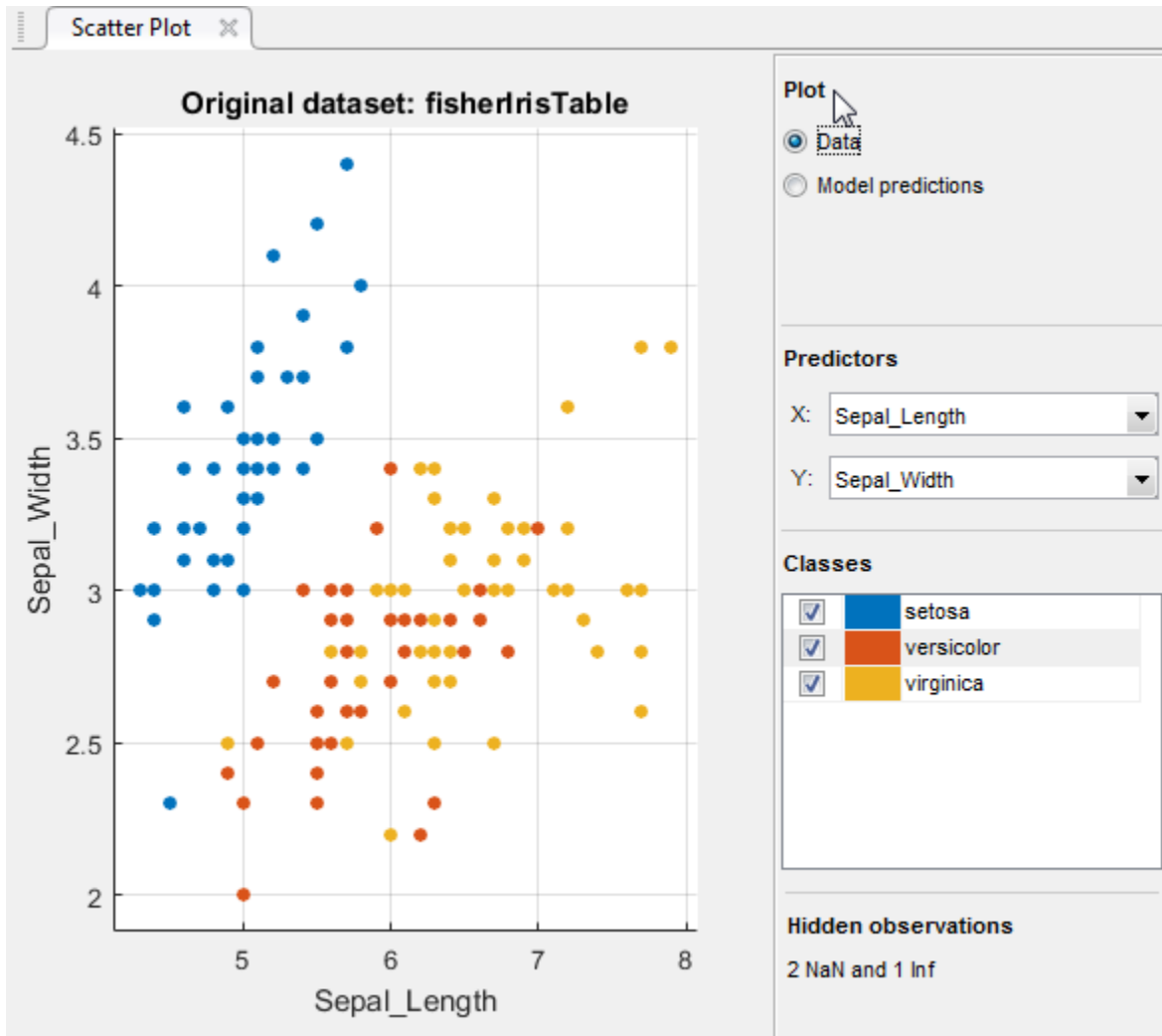
## Investigate Features in the Scatter Plot

In Classification Learner, try to identify predictors that separate classes well by plotting different pairs of predictors on the scatter plot. The plot can help you investigate features to include or exclude. You can visualize training data and misclassified points on the scatter plot.

Before you train a classifier, the scatter plot shows the data. If you have trained a classifier, the scatter plot shows model prediction results. Switch to plotting only the data by selecting **Data** in the **Plot** controls.

- Choose features to plot using the **X** and **Y** lists under **Predictors**
- Look for predictors that separate classes well.

Plotting the `fisheriris` data, you can see that sepal length and sepal width separate one of the classes well (`setosa`). You need to plot other predictors to see if you can separate the other two classes.



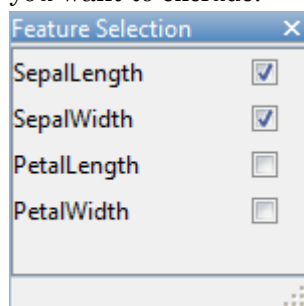
- Show or hide specific classes using the check boxes under **Classes**.
- If you identify predictors that are not useful for separating out classes, use **Feature Selection** to remove them and train classifiers including only the most useful predictors.

After you train a classifier, the scatter plot shows model prediction results. You can show or hide correct or incorrect results and visualize the results by class. See “Plot Classifier Results” on page 23-54.

## Select Features to Include

In Classification Learner, you can specify different features (or predictors) to include in the model. See if you can improve models by removing features with low predictive power. If data collection is expensive or difficult, you might prefer a model that performs satisfactorily without some predictors.

- 1 On the **Classification Learner** tab, in the **Features** section, click **Feature Selection**.
- 2 In the Feature Selection tearaway window, clear the check boxes for the predictors you want to exclude.



---

**Tip** You can close the Feature Selection tearaway window, or move it. Your choices in the tearaway remain.

---

- 3 Click **Train** to train a new model using the new predictor options.
- 4 Observe the new model in the History list. The Current model pane displays how many predictors are excluded.
- 5 To check which predictors are included in a trained model, click the model in the History list and observe the check boxes in the Feature Selection dialog box.
- 6 You can try to improve the model by including different features in the model.

For an example using feature selection, see “Train Decision Trees Using Classification Learner App” on page 23-67.

## Transform Features with PCA in Classification Learner

Use principal component analysis (PCA) to reduce the dimensionality of the predictor space. Reducing the dimensionality can create classification models in Classification Learner that help prevent overfitting. PCA linearly transforms predictors in order to remove redundant dimensions, and generates a new set of variables called principal components.

- 1 On the **Classification Learner** tab, in the **Features** section, select **PCA**.
- 2 In the Advanced PCA Options tearaway window, select the **Enable PCA** check box.

You can close the PCA tearaway window, or move it. Your choices in the tearaway remain.

- 3 When you next click **Train**, the `pca` function transforms your selected features before training the classifier.
- 4 By default, PCA keeps only the components that explain 95% of the variance. In the PCA tearaway window, you can change the percentage of variance to explain in the **Explained variance** box. A higher value risks overfitting, while a lower value risks removing useful dimensions.
- 5 If you want to manually limit the number of PCA components, in the **Component reduction criterion** list, select `Specify number of components`. Edit the number in the **Number of numeric components** box. The number of components cannot be larger than the number of numeric predictors. PCA is not applied to categorical predictors.

Check PCA options for trained models in the Current model pane information. Check the explained variance percentages to decide whether to change the number of components. For example:

```
PCA is keeping enough components to explain 95% variance.  
After training, 2 components were kept.  
Explained variance per component (in order): 92.5%, 5.3%, 1.7%, 0.5%
```

To learn more about how Classification Learner applies PCA to your data, generate code for your trained classifier. For more information on PCA, see the `pca` function.

## Investigate Features in the Parallel Coordinates Plot

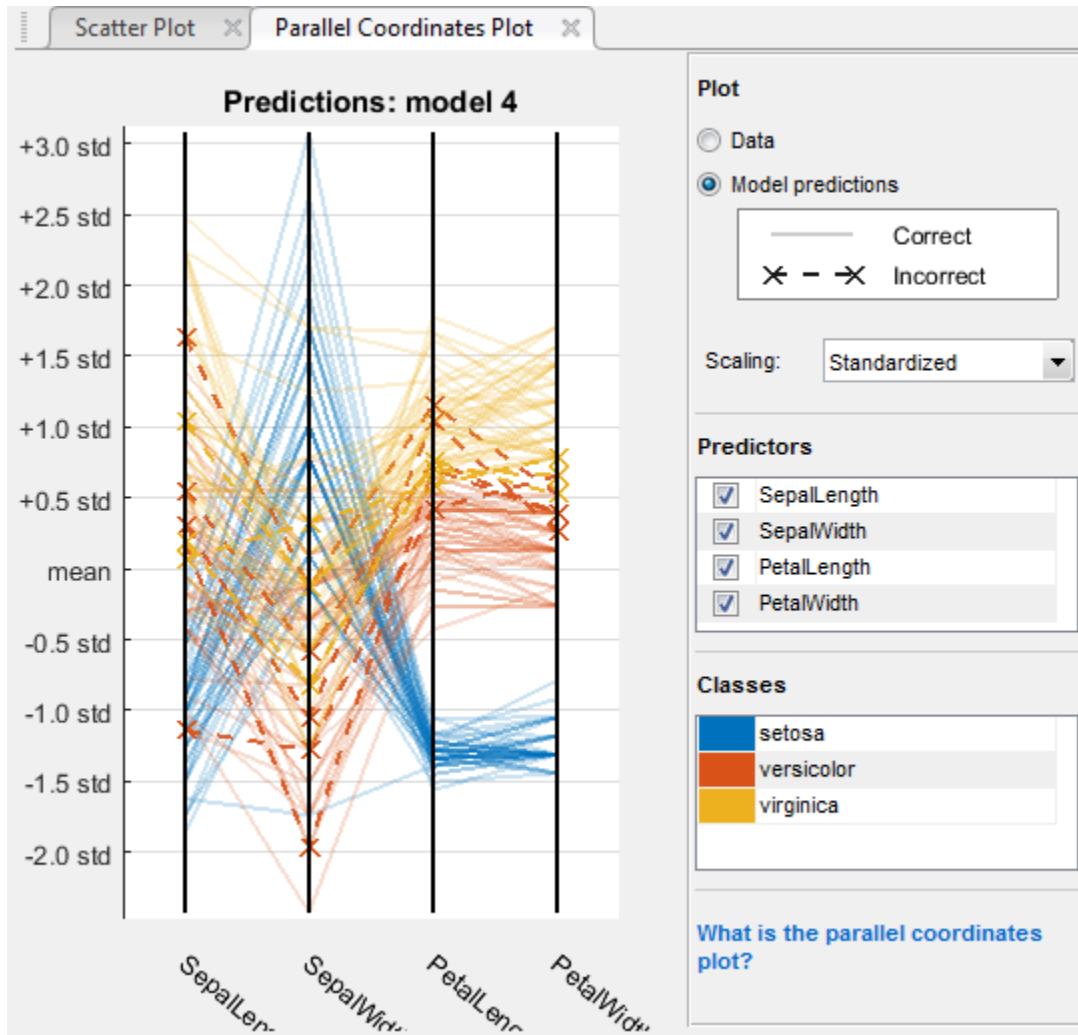
To investigate features to include or exclude, use the parallel coordinates plot. You can visualize high dimensional data on a single plot to see 2D patterns. This plot can help



you understand relationships between features and identify useful predictors for separating classes. You can visualize training data and misclassified points on the parallel coordinates plot. When you plot classifier results, misclassified points show dashed lines.

- 1 On the **Classification Learner** tab, in the **Plots** section, select **Parallel Coordinates Plot**.
- 2 On the plot, you can drag the bars to reorder the predictors. Changing the order can help you identify predictors that separate classes well.
- 3 To specify which predictors to plot, use the **Predictors** check boxes. It can be helpful to plot a few predictors at a time. If your data has many predictors, the plot shows the first 10 by default.
- 4 If the predictors have very different scales, scale the data to make it easier to visualize. Try different options in the **Scaling** list:
  - **Normalization** plots all predictors on the same range from 0 to 1.
  - **Standardization** plots the mean of each predictor at zero and scales the predictors by their standard deviations.
- 5 If you identify predictors that are not useful for separating out classes, use **Feature Selection** to remove them and train classifiers including only the most useful predictors.

Plotting the `fisheriris` data, you can see that petal length and petal width are the features that separate the classes best.



## See Also

### Related Examples

- “Train Classification Models in Classification Learner App” on page 23-11

- “Select Data and Validation for Classification Problem” on page 23-20
- “Choose Classifier Options” on page 23-26
- “Assess Classifier Performance in Classification Learner” on page 23-54
- “Generate MATLAB Code to Train the Model with New Data” on page 23-63

## Assess Classifier Performance in Classification Learner

### In this section...

“Check Performance in the History List” on page 23-54

“Plot Classifier Results” on page 23-54

“Check Performance Per Class in the Confusion Matrix” on page 23-56

“Check the ROC Curve” on page 23-59

### Check Performance in the History List

After training a model in Classification Learner, check the History list to see which model has the best overall accuracy in percent. The best **Accuracy** score is highlighted in a box. This score is the validation accuracy (unless you opted for no validation scheme). The validation accuracy score estimates a model's performance on new data compared to the training data. Use the score to help you choose the best model.

- For cross-validation, the score is the accuracy on all observations, counting each observation when it was in a held-out fold.

---

**Note** When you imported data into the app, if you accepted the defaults, you are using cross-validation. To learn more, see “Choose Validation Scheme” on page 23-23.

---

- For holdout validation, the score is the accuracy on the held-out observations.
- For no validation, the score is the resubstitution accuracy against all the training data observations.

The best overall score might not be the best model for your goal. A model with a slightly lower overall accuracy might be the best classifier for your goal. For example, false positives in a particular class might be important to you. You might want to exclude some predictors where data collection is expensive or difficult.

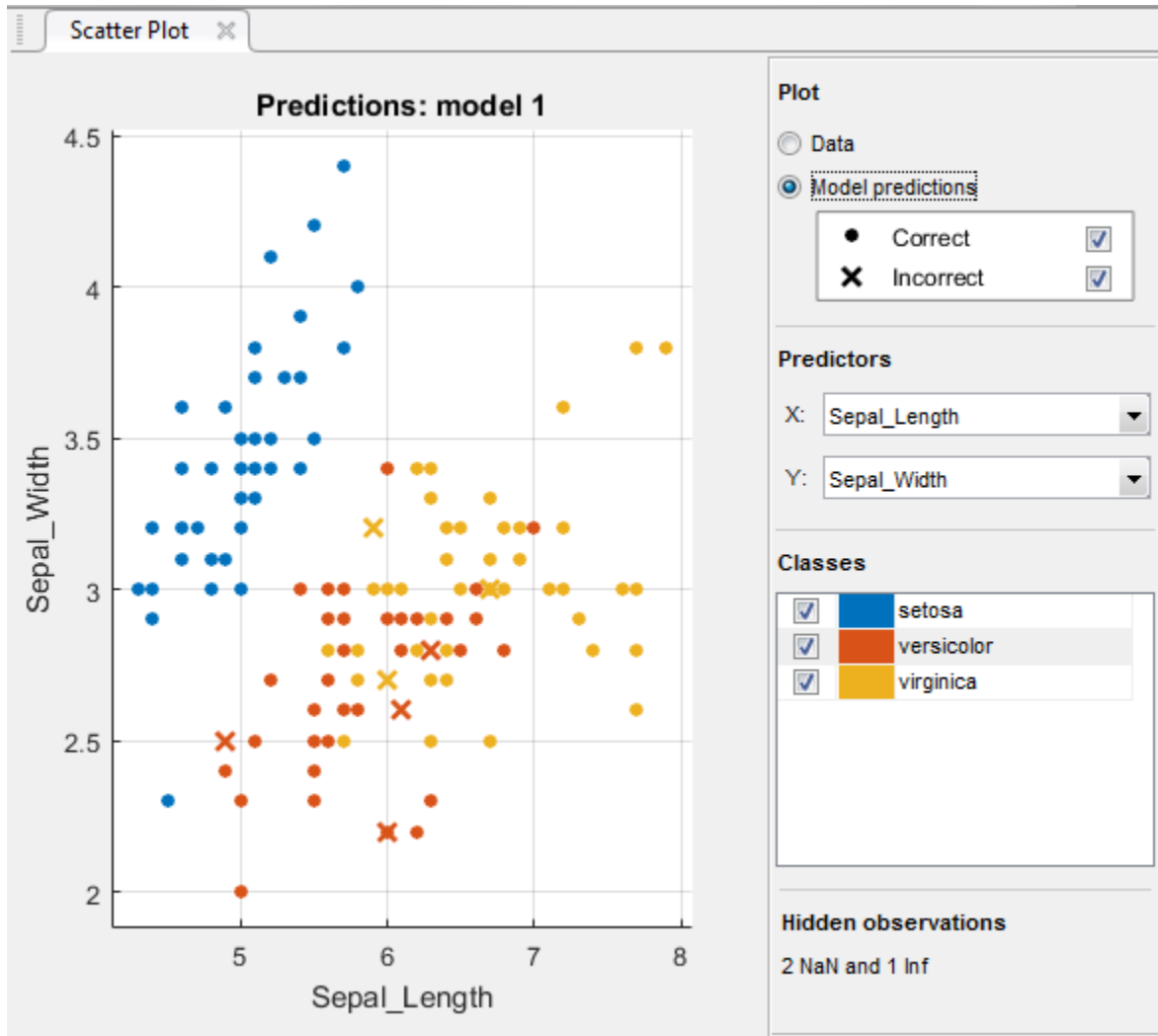
To find out how the classifier performed in each class, examine the confusion matrix.

### Plot Classifier Results

In the scatter plot, view the classifier results. After you train a classifier, the scatter plot switches from displaying the data to showing model predictions. If you are using holdout

or cross-validation, then these predictions are the predictions on the held-out observations. In other words, each prediction is obtained using a model that was trained without using the corresponding observation. To investigate your results, use the controls on the right. You can:

- Choose whether to plot model predictions or the data alone
- Show or hide correct or incorrect results using the check boxes under **Model predictions**
- Choose features to plot using the **X** and **Y** lists under **Predictors**
- Visualize results by class by showing or hiding specific classes using the check boxes under **Classes**



See also “Investigate Features in the Scatter Plot” on page 23-47.

## Check Performance Per Class in the Confusion Matrix

Use the confusion matrix plot to understand how the currently selected classifier performed in each class. To view the confusion matrix after training a model, on the

**Classification Learner** tab, in the **Plots** section, click **Confusion Matrix**. The confusion matrix helps you identify the areas where the classifier has performed poorly.

When you open the plot, the rows show the true class, and the columns show the predicted class. If you are using holdout or cross-validation, then the confusion matrix is calculated using the predictions on the held-out observations. The diagonal cells show where the true class and predicted class match. If these cells are green, the classifier has performed well and classified observations of this true class correctly.

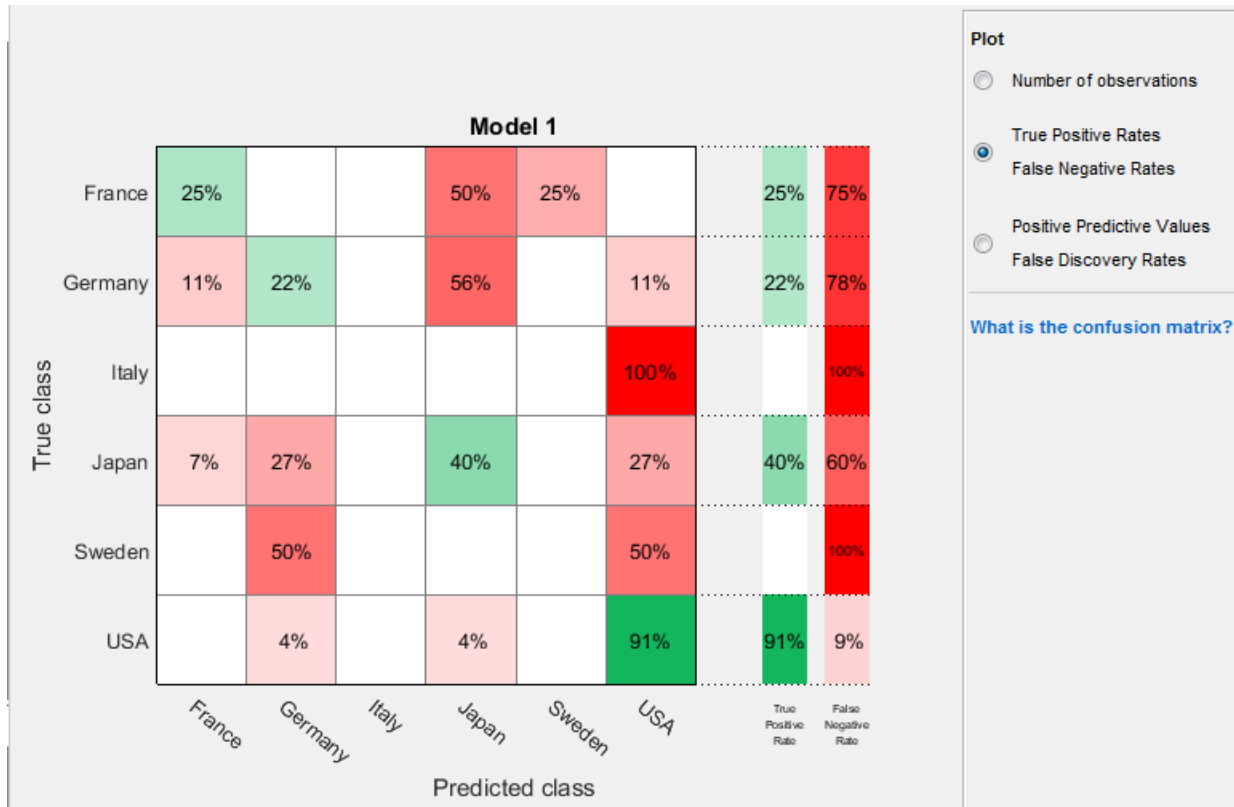
The default view shows number of observations in each cell.

To see how the classifier performed per class, under **Plot**, select the **True Positive Rates, False Negative Rates** option. The plot shows summaries per true class in the last two columns on the right.

---

**Tip** Look for areas where the classifier performed poorly by examining cells off the diagonal that display high percentages and are red. The higher the percentage, the brighter the hue of the cell color. In these red cells, the true class and the predicted class do not match. The data points are misclassified.

---



In this example, using the `carsmall` data set, the top row shows all cars with true class France. The columns show the predicted classes. In the top row, 25% of the cars from France are correctly classified, so **25%** is the true positive rate for correctly classified points in this class, shown in the green cell in the **True Positive Rate** column.

The other cars in the France row are misclassified: 50% of the cars are incorrectly classified as from Japan, and 25% are classified as from Sweden. **75%** is the false negative rate for incorrectly classified points in this class, shown in the red cell in the **False Negative Rate** column.

If you want to see numbers of observations (cars, in this example) instead of percentages, under **Plot**, select **Number of observations**.



If false positives are important in your classification problem, plot results per predicted class (instead of true class) to investigate false discovery rates. To see results per predicted class, under **Plot**, select the **Positive Predictive Values False Discovery Rates** option. The confusion matrix now shows summary rows underneath the table. Positive predictive values are shown in green for the correctly predicted points in each class, and false discovery rates are shown below it in red for the incorrectly predicted points in each class.

If you decide there are too many misclassified points in the classes of interest, try changing classifier settings or feature selection to search for a better model.

## Check the ROC Curve

To view the ROC curve after training a model, on the **Classification Learner** tab, in the **Plots** section, click **ROC Curve**. View the receiver operating characteristic (ROC) curve showing true and false positive rates. The ROC curve shows true positive rate versus false positive rate for the currently selected trained classifier. You can select different classes to plot.

The marker on the plot shows the performance of the currently selected classifier. The marker shows the values of the false positive rate (FPR) and the true positive rate (TPR) for the currently selected classifier. For example, a false positive rate (FPR) of 0.2 indicates that the current classifier assigns 20% of the observations incorrectly to the positive class. A true positive rate of 0.9 indicates that the current classifier assigns 90% of the observations correctly to the positive class.

A perfect result with no misclassified points is a right angle to the top left of the plot. A poor result that is no better than random is a line at 45 degrees. The **Area Under Curve** number is a measure of the overall quality of the classifier. Larger **Area Under Curve** values indicate better classifier performance. Compare classes and trained models to see if they perform differently in the ROC curve.

For more information, see [perfcurve](#).

## See Also

### Related Examples

- “Train Classification Models in Classification Learner App” on page 23-11
- “Select Data and Validation for Classification Problem” on page 23-20
- “Choose Classifier Options” on page 23-26
- “Feature Selection and Feature Transformation Using Classification Learner App” on page 23-47
- “Export Classification Model to Predict New Data” on page 23-61
- “Train Decision Trees Using Classification Learner App” on page 23-67

## Export Classification Model to Predict New Data

### In this section...

“Export the Model to the Workspace to Make Predictions for New Data” on page 23-61

“Make Predictions for New Data” on page 23-62

“Generate MATLAB Code to Train the Model with New Data” on page 23-63

“Generate C Code for Prediction” on page 23-64

### Export the Model to the Workspace to Make Predictions for New Data

After you create classification models interactively in Classification Learner, you can export your best model to the workspace. You can then use the trained model to make predictions using new data.

- 1 In Classification Learner, select the model you want to export in the History list.
- 2 On the **Classification Learner** tab, in the **Export** section, click one of the export options:
  - If you want to include the data used for training the model, then select **Export Model**.  
  
You export the trained model to the workspace as a structure containing a classification object, such as a `ClassificationTree`, `ClassificationDiscriminant`, `ClassificationSVM`, `ClassificationKNN`, `ClassificationEnsemble`, etc.
  - If you do not want to include the training data, select **Export Compact Model**. This option exports the model with unnecessary data removed where possible. For some classifiers this is a compact classification object that does not include the training data (e.g., `CompactClassificationTree`). You can use a compact classification object for making predictions of new data, but you can use fewer other methods with it.
- 3 In the Export Model dialog box, edit the name for your exported variable if you want, and then click **OK**. The default name for your exported model, `trainedModel`, increments every time you export to avoid overwriting your classifiers, e.g., `trainedModel1`.

The new variable, e.g., `trainedModel`, appears in your workspace.

The app displays information about the exported model in the command window. Read the message to learn how to make predictions with new data.

---

**Note** The final exported model is always trained using the full data set. The validation scheme that you use only affects the way that Classification Learner computes validation metrics.

---

## Make Predictions for New Data

After you export a model to the workspace from Classification Learner, or run the code generated from the app, you get a `trainedModel` structure that you can use to make predictions using new data. The structure contains a classification object and a function for prediction. The structure allows you to make predictions for models that include principal component analysis (PCA).

- 1 To use the exported classifier to make predictions for new data, `T`, use the form:

```
yfit = C.predictFcn(T)
```

where `C` is the name of your variable, e.g., `trainedModel`.

Supply the data `T` in same data type as your training data used in the app (table or matrix).

- If you supply a table, ensure it contains the same predictor names as your training data. The `predictFcn` ignores additional variables in tables. Variable formats (e.g. matrix or vector, data type) must match the original training data.
- If you supply a matrix, it must contain the same predictor columns or rows as your training data, in the same order and format. Do not include a response variable, any variables that you did not import in the app, or other unused variables.

The output `yfit` contains a class prediction for each data point.

- 2 Examine the fields of the exported structure. For help making predictions, enter:

```
C.HowToPredict
```

You can also extract the classification object from the exported struct for further analysis (e.g., `trainedModel.ClassificationSVM`, `trainedModel.ClassificationTree`,

etc., depending on your model type). Be aware that if you used feature transformation such as PCA in the app, you will need to take account of this transformation by using the information in the PCA fields of the struct.

## Generate MATLAB Code to Train the Model with New Data

After you create classification models interactively in Classification Learner, you can generate MATLAB code for your best model. You can then use the code to train the model with new data.

Generate MATLAB code to:

- Train on huge data sets. Explore models in the app trained on a subset of your data, then generate code to train a selected model on a larger data set
- Create scripts for training models without needing to learn syntax of the different functions
- Examine the code to learn how to train classifiers programmatically
- Modify the code for further analysis, for example to set options that you cannot change in the app
- Repeat your analysis on different data and automate training

- 1 In Classification Learner, in the History list, select the model you want to generate code for.
- 2 On the **Classification Learner** tab, in the **Export** section, click **Export Model > Generate Code**.

The app generates code from your session and displays the file in the MATLAB Editor. The file includes the predictors and response, the classifier training methods, and validation methods. Save the file.

- 3 To retrain your classifier model, call the function from the command line with your original data or new data as the input argument. New data must be the same shape.

Copy the first line of the generated code excluding the word `function`, and edit the `trainingData` input argument to the variable name of your training data or new data. For example, to retrain a classifier trained with the `fisheriris` data set, enter:

```
[trainedModel, validationAccuracy] = trainClassifier(fisheriris)
```

The generated code returns a `trainedModel` structure that contains the same fields as the struct you create when you export a classifier from Classification Learner to the workspace.

- 4 If you want to automate training the same classifier with new data, or learn how to programmatically train classifiers, examine the generated code. The code shows you how to:
  - Process the data into the right shape
  - Train a classifier and specify all the classifier options
  - Perform cross-validation
  - Compute validation accuracy
  - Compute validation predictions and scores

## Generate C Code for Prediction

If you train an SVM model using Classification Learner, you can generate C code for prediction.

C code generation requires:

- MATLAB Coder license
- SVM model (binary or multiclass)
- No categorical predictors or response in your data set

- 1 After you train an SVM model in Classification Learner, export the model to the workspace.

Find the name of the classification model object in the exported struct. Examine the fields of the struct to find the model name, for example, `C.ClassificationSVM`, where `C` is the name of your struct, e.g., `trainedModel`.

Model name depends on what type of SVM you trained (binary or multiclass) and whether you exported a compact model or not. Models can be `ClassificationSVM`, `CompactClassificationSVM`, `ClassificationECOC`, or `CompactClassificationECOC`.

- 2 Use the function `saveCompactModel` to prepare the model for code generation:  
`saveCompactModel(Mdl, filename)`. For example:

```
saveCompactModel(C.ClassificationSVM, 'mySVM')
```

- 3 Create a function that loads the saved model and makes predictions on new data. For example:

```
function label = classifyX (X) %#codegen
%CLASSIFYX Classify using SVM Model
% CLASSIFYX classifies the measurements in X
% using the SVM model in the file mySVM.mat, and then
% returns class labels in label.

CompactMdl = loadCompactModel('mySVM');
label = predict(CompactMdl,X);
end
```

- 4 Generate a MEX function from your function. For example:

```
codegen classifyX.m -args {data}
```

The `%codegen` compilation directive indicates that the MATLAB code is intended for code generation. To ensure that the MEX function can use the same input, specify the data in the workspace as arguments to the function using the `-args` option. `data` must be a matrix containing only the predictor columns used to train the model.

- 5 Use the MEX function to make predictions. For example:

```
labels = classifyX_mex(data);
```

If you used feature selection or PCA feature transformation in the app, then you need additional steps. If you used manual feature selection, supply the same columns in `X`. `X` is the input to your function.

If you used PCA in the app, use the information in the PCA fields of the exported struct to take account of this transformation. It does not matter whether you imported a table or a matrix into the app, as long as `X` contains the matrix columns in the same order. Before generating code, follow these steps:

- 1 Save the `PCACenters` and `PCACoefficients` fields of the trained classifier struct, `C`, to file using the following command:

```
save('pcaInfo.mat', '-struct', 'C', 'PCACenters', 'PCACoefficients');
```

- 2 In your function file, include additional lines to perform the PCA transformation. Create a function that loads the saved model, performs PCA, and makes predictions on new data. For example:

```
function label = classifyX (X) %#codegen
%CLASSIFYX Classify using SVM Model
% CLASSIFYX classifies the measurements in X
% using the SVM model in the file mySVM.mat, and then
% returns class labels in label.
% If you used manual feature selection in the app, ensure that X contains only the

CompactMdl = loadCompactModel('mySVM');
pcaInfo = load('pcaInfo.mat', 'PCACenters', 'PCACoefficients');
% performs pca transformation
pcaTransformedX = bsxfun(@minus, X, pcaInfo.PCACenters) * pcaInfo.PCACoefficients;
[label, scores] = predict(CompactMdl, pcaTransformedX);
end
```

For more information on C code generation workflow and limitations, see “Code Generation”. For examples, see `saveCompactModel` and `loadCompactModel`.

## See Also

### Functions

`fitcdiscr` | `fitcecoc` | `fitcensemble` | `fitcknn` | `fitcsvm` | `fitctree` | `fitglm`

### Classes

`ClassificationBaggedEnsemble` | `ClassificationDiscriminant` |  
`ClassificationECOC` | `ClassificationEnsemble` | `ClassificationKNN` |  
`ClassificationSVM` | `ClassificationTree` |  
`CompactClassificationDiscriminant` | `CompactClassificationECOC` |  
`CompactClassificationEnsemble` | `CompactClassificationSVM` |  
`CompactClassificationTree` | `GeneralizedLinearModel`

## Related Examples

- “Train Classification Models in Classification Learner App” on page 23-11

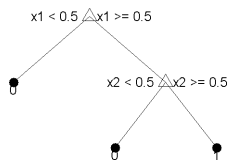


## Train Decision Trees Using Classification Learner App

This example shows how to create and compare various classification trees using Classification Learner, and export trained models to the workspace to make predictions for new data.

You can train classification trees to predict responses to data. To predict a response, follow the decisions in the tree from the root (beginning) node down to a leaf node. The leaf node contains the response.

Statistics and Machine Learning Toolbox trees are binary. Each step in a prediction involves checking the value of one predictor (variable). For example, here is a simple classification tree:



This tree predicts classifications based on two predictors,  $x_1$  and  $x_2$ . To predict, start at the top node. At each decision, check the values of the predictors to decide which branch to follow. When the branches reach a leaf node, the data is classified either as type 0 or 1.

This example use Fisher's 1936 iris data. The iris data contains measurements of flowers: the petal length, petal width, sepal length, and sepal width for specimens from three species. Train a classifier to predict the species based on the predictor measurements.

- 1 In MATLAB, load the `fisheriris` data set and create a table of measurement predictors (or features) using variables from the data set to use for a classification.

```
fishertable = readtable('fisheriris.csv');
```

- 2 On the **Apps** tab, in the **Math, Statistics and Optimization** group, click **Classification Learner**.
- 3 On the **Classification Learner** tab, in the **File** section, click **New Session**.
- 4 In the New Session dialog box, select the table `fishertable` from the workspace list.

Observe that the app has selected response and predictor variables based on their data type. Petal and sepal length and width are predictors, and species is the response that you want to classify. For this example, do not change the selections.

**Data set**

Workspace Variable: fishertable (150x5 table)

Response: Species (cell, 3 unique)

	Name	Type	Range
<input checked="" type="checkbox"/>	SepalLength	double	4.3 .. 7.9
<input checked="" type="checkbox"/>	SepalWidth	double	2 .. 4.4
<input checked="" type="checkbox"/>	PetalLength	double	1 .. 6.9
<input checked="" type="checkbox"/>	PetalWidth	double	0.1 .. 2.5
<input type="checkbox"/>	Species	cell	3 unique

Buttons: Add All, Remove All

[How to prepare data](#)

**Validation**

**Cross-Validation**  
Protects against overfitting by partitioning the data set into folds and estimating accuracy on each fold.  
Cross-validation folds: 5 folds

**Holdout Validation**  
Recommended for large data sets.  
Percent held out: 25%

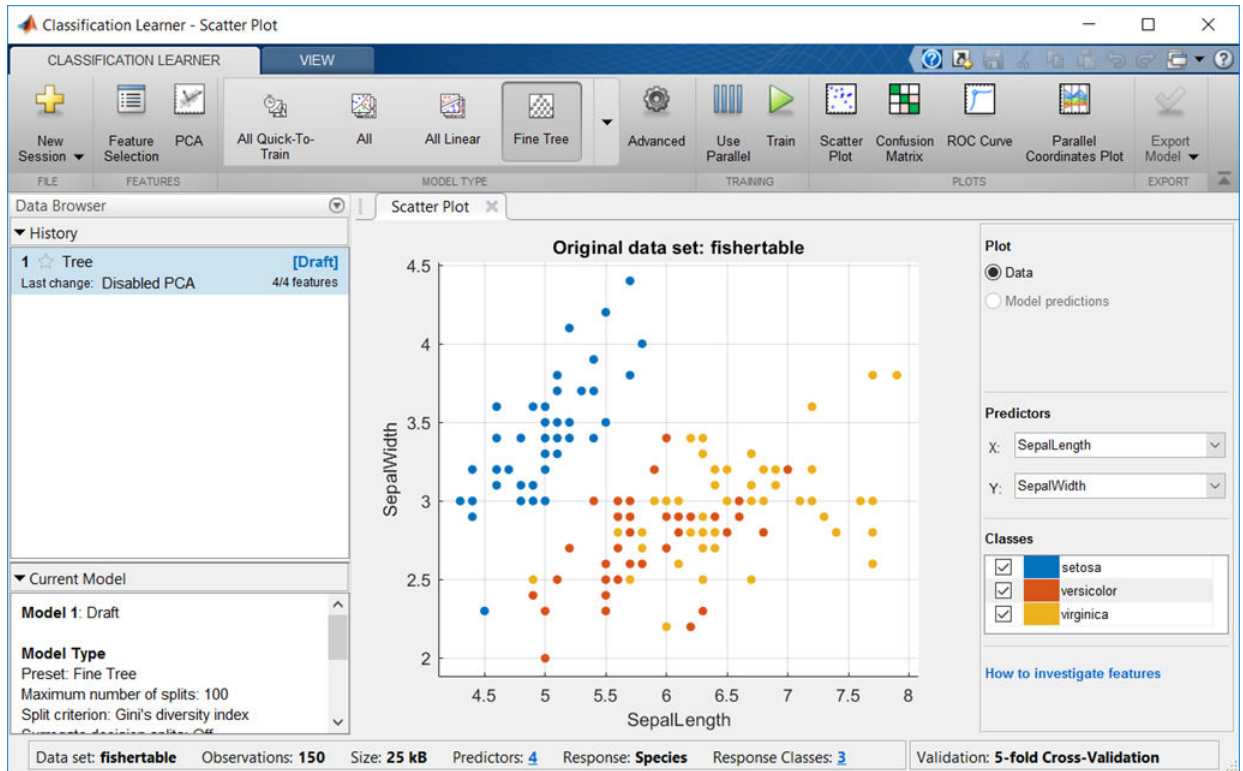
**No Validation**  
No protection against overfitting.

[Read about validation](#)

Buttons: Start Session, Cancel

- To accept the default validation scheme and continue, click **Start Session**. The default validation option is cross-validation, to protect against overfitting.

Classification Learner creates a scatter plot of the data.



- Use the scatter plot to investigate which variables are useful for predicting the response. Select different options on the **X** and **Y** lists under **Predictors** to visualize the distribution of species and measurements. Observe which variables separate the species colors most clearly.

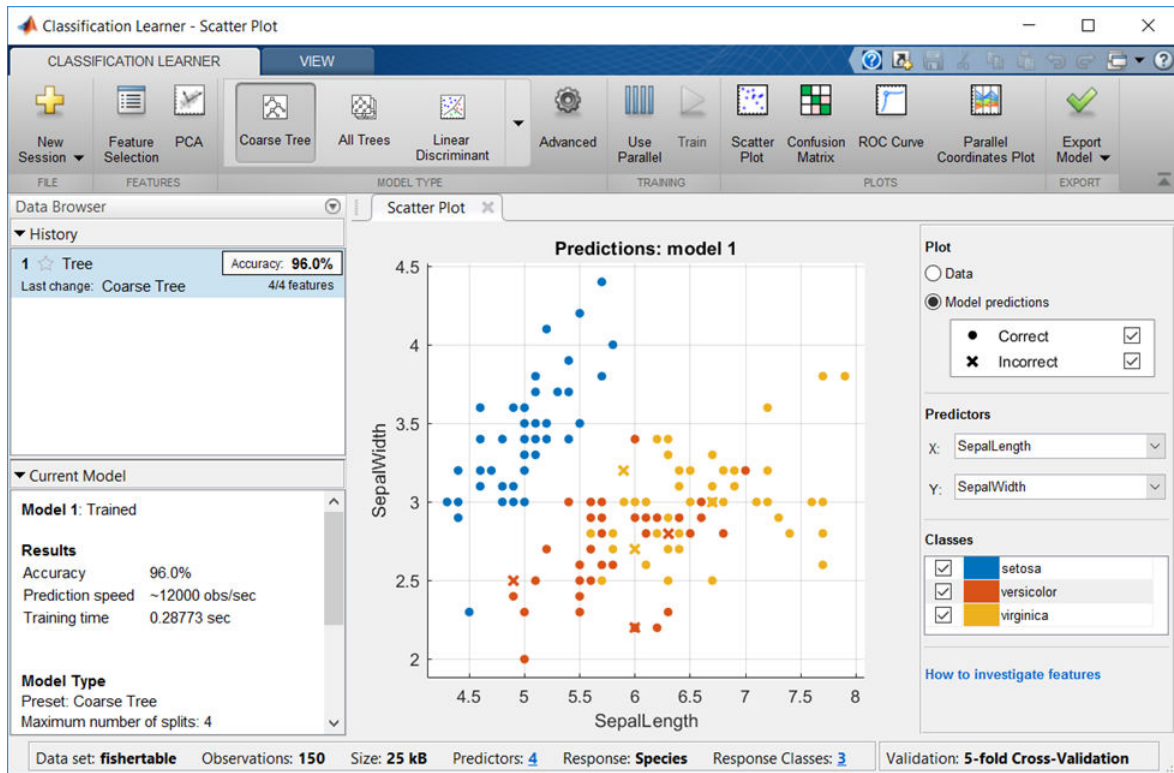
Observe that the *setosa* species (blue points) is easy to separate from the other two species with all four predictors. The *versicolor* and *virginica* species are much closer together in all predictor measurements, and overlap especially when you plot sepal length and width. *setosa* is easier to predict than the other two species.

- To create a classification tree model, on the **Classification Learner** tab, in the **Model Type** section, click the down arrow to expand the gallery and click **Coarse Tree**. Then click **Train**.

**Tip** If you have Parallel Computing Toolbox then the first time you click **Train** you see a dialog while the app opens a parallel pool of workers. After the pool opens, you can train multiple classifiers at once and continue working.

The app creates a simple classification tree, and plots the results.

Observe the **Coarse Tree** model in the History list. Check the model validation score in the **Accuracy** box. The model has performed well.



**Note** With validation, there is some randomness in the results, so your model validation score results can vary from those shown.

- 8 Examine the scatter plot. An X indicates misclassified points. The blue points (setosa species) are all correctly classified, but some of the other two species are misclassified. Under **Plot**, switch between the **Data** and **Model Predictions**

options. Observe the color of the incorrect (X) points. Alternatively, while plotting model predictions, to view only the incorrect points, clear the **Correct** check box.

- 9 Train a different model to compare. Click **Medium Tree**, and then click **Train**.

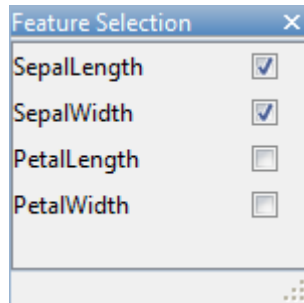
When you click **Train**, the app displays a new model in the History list.

- 10 Observe the **Medium Tree** model in the History list. The model validation score is no better than the coarse tree score. The app outlines in a box the `Accuracy` score of the best model. Click each model in the History list to view and compare the results.
- 11 Examine the scatter plot for the **Medium Tree** model. The medium tree classifies as many points correctly as the previous coarse tree. You want to avoid overfitting, and the coarse tree performs well, so base all further models on the coarse tree.
- 12 Select **Coarse Tree** in the History list. To try to improve the model, try including different features in the model. See if you can improve the model by removing features with low predictive power.

On the **Classification Learner** tab, in the **Features** section, click **Feature Selection**.

In the Feature Selection dialog box, clear the check boxes for **PetalLength** and **PetalWidth** to exclude them from the predictors. A new draft model appears in the model history list with your new settings 2/4 features, based on the coarse tree.

Click **Train** to train a new tree model using the new predictor options.

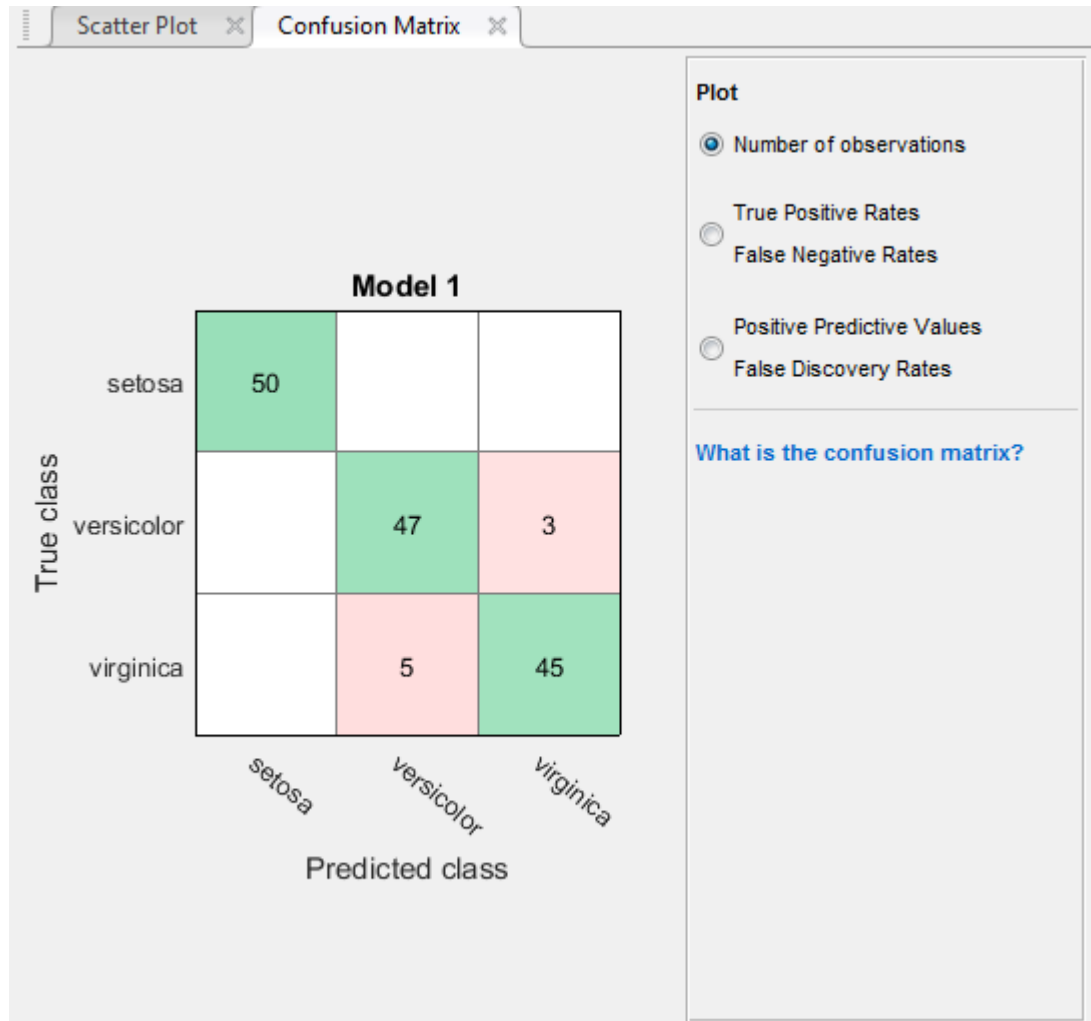


- 13 Observe the third model in the History list. It is also a **Coarse Tree** model, trained using only 2 of 4 predictors. The History list displays how many predictors are excluded. To check which predictors are included, click a model in the History list and observe the check boxes in the Feature Selection dialog box. The model with only sepal measurements has a much lower accuracy score than the petals-only model.
- 14 Train another model including only the petal measurements. Change the selections in the Feature Selection dialog box and click **Train**.

The model trained using only petal measurements performs comparably to the models containing all predictors. The models predict no better using all the measurements compared to only the petal measurements. If data collection is expensive or difficult, you might prefer a model that performs satisfactorily without some predictors.

- 15 Repeat to train two more models including only the width measurements and then the length measurements. There is not much difference in score between several of the models.
- 16 Choose a best model among those of similar scores by examining the performance in each class. Select the coarse tree that includes all the predictors. To inspect the accuracy of the predictions in each class, on the **Classification Learner** tab, in the **Plots** section, click **Confusion Matrix**. Use this plot to understand how the currently selected classifier performed in each class. View the matrix of true class and predicted class results.

Look for areas where the classifier performed poorly by examining cells off the diagonal that display high numbers and are red. In these red cells, the true class and the predicted class do not match. The data points are misclassified.



**Note** With validation, there is some randomness in the results, so your confusion matrix results can vary from those shown.

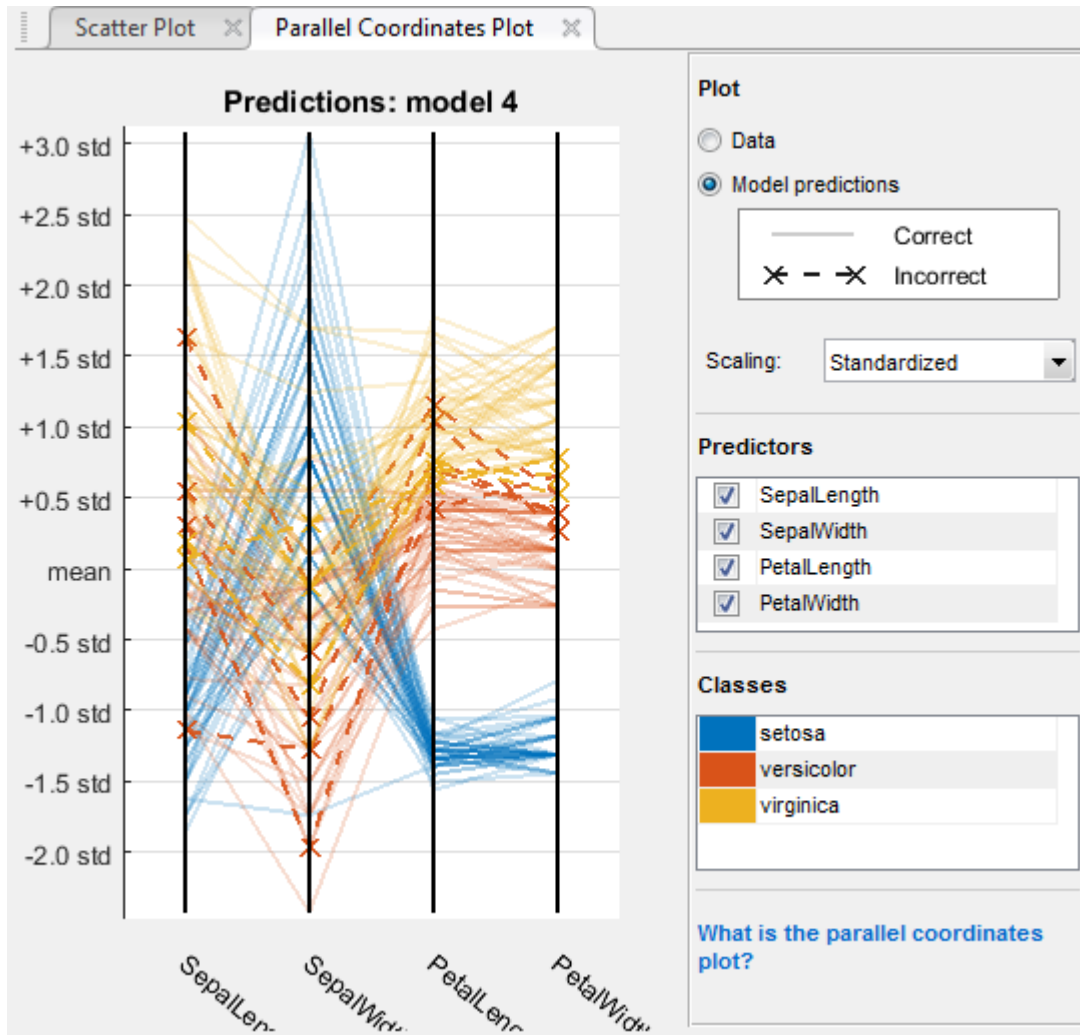
In this figure, examine the third cell in the middle row. In this cell, true class is *versicolor*, but the model misclassified the points as *virginica*. For this model,

the cell shows 3 misclassified (your results can vary). To view percentages instead of numbers of observations, select the **True Positive Rates** option under **Plot** controls..

You can use this information to help you choose the best model for your goal. If false positives in this class are very important to your classification problem, then choose the best model at predicting this class. If false positives in this class are not very important, and models with fewer predictors do better in other classes, then choose a model to tradeoff some overall accuracy to exclude some predictors and make future data collection easier.

- 17 Compare the confusion matrix for each model in the History list. Check the Feature Selection dialog box to see which predictors are included in each model.
- 18 To investigate features to include or exclude, use the scatter plot and the parallel coordinates plot. On the **Classification Learner** tab, in the **Plots** section, click **Parallel Coordinates Plot**. You can see that petal length and petal width are the features that separate the classes best.





- 19 To learn about model settings, choose a model in the History list and view the advanced settings. The options in the **Model Type** gallery are preset starting points, and you can change further settings. On the **Classification Learner** tab, in the **Model Type** section, click **Advanced**. Compare the simple and medium tree models in the history, and observe the differences in the Advanced Tree Options dialog box. The **Maximum Number of Splits** setting controls tree depth.

To try to improve the coarse tree model further, try changing the **Maximum Number of Splits** setting, then train a new model by clicking **Train**.

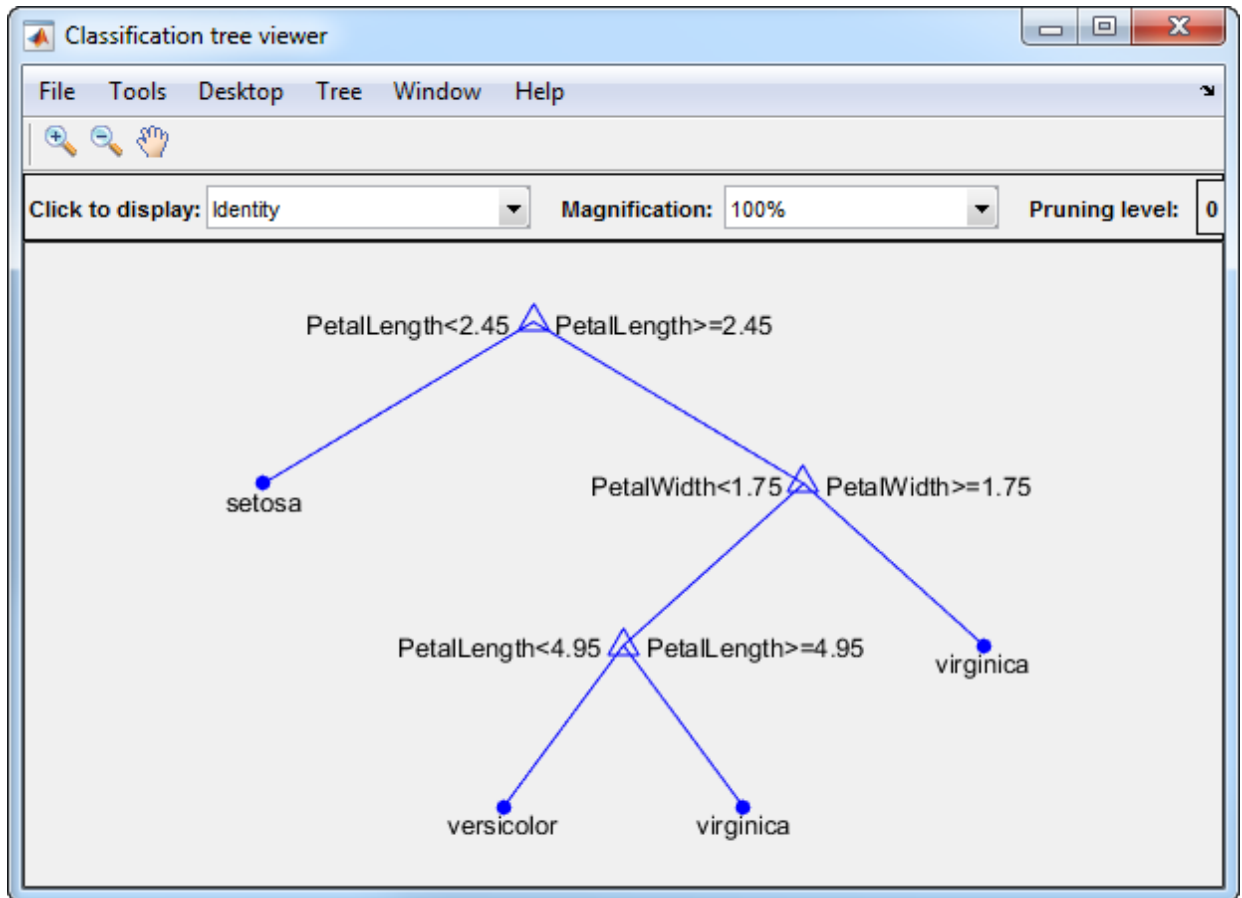
View the settings for the selected trained model in the Current model pane, or in the Advanced dialog box.

- 20** To export the best trained model to the workspace, on the **Classification Learner** tab, in the **Export** section, click **Export Model**. In the Export Model dialog box, click **OK** to accept the default variable name `trainedModel`.

Look in the command window to see information about the results.

- 21** To visualize your decision tree model, enter:

```
view(trainedModel.ClassificationTree, 'Mode', 'graph')
```



- 22 You can use the exported classifier to make predictions on new data. For example, to make predictions for the `fishertable` data in your workspace, enter:

```
yfit = trainedModel.predictFcn(fishertable)
```

The output `yfit` contains a class prediction for each data point.

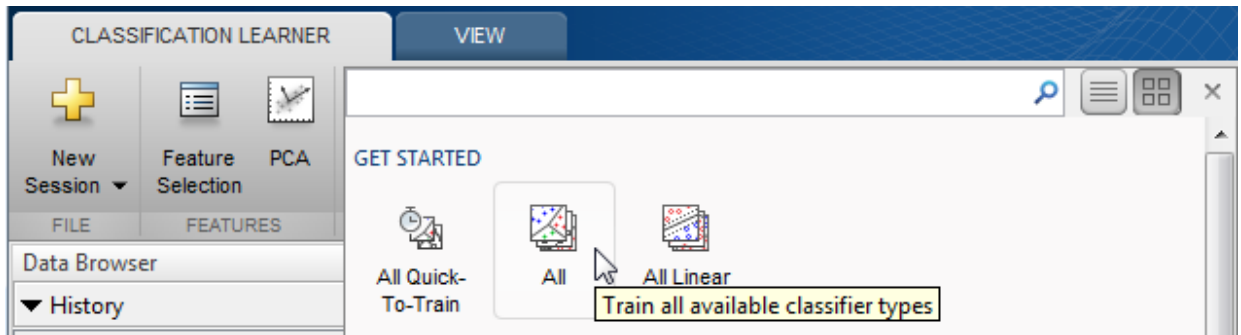
- 23 If you want to automate training the same classifier with new data, or learn how to programmatically train classifiers, you can generate code from the app. To generate code for the best trained model, on the **Classification Learner** tab, in the **Export** section, click **Export Model > Generate Code**.

The app generates code from your model and displays the file in the MATLAB Editor. To learn more, see “Generate MATLAB Code to Train the Model with New Data” on page 23-63.

Use the same workflow to evaluate and compare the other classifier types you can train in Classification Learner.

To try all the classifier model presets available for your data set:

- 1 Click the arrow on the far right of the **Model Type** section to expand the list of classifiers.
- 2 Click **All**, then click **Train**.



To learn about other classifier types, see “Train Classification Models in Classification Learner App” on page 23-11.

## See Also

### Related Examples

- “Train Classification Models in Classification Learner App” on page 23-11
- “Select Data and Validation for Classification Problem” on page 23-20
- “Choose Classifier Options” on page 23-26
- “Feature Selection and Feature Transformation Using Classification Learner App” on page 23-47
- “Assess Classifier Performance in Classification Learner” on page 23-54

- “Export Classification Model to Predict New Data” on page 23-61

## Train Discriminant Analysis Classifiers Using Classification Learner App

This example shows how to construct discriminant analysis classifiers in the Classification Learner app, using the `fisheriris` data set. You can use discriminant analysis with two or more classes in Classification Learner.

- 1 In MATLAB, load the `fisheriris` data set.

```
fishertable = readtable('fisheriris.csv');
```

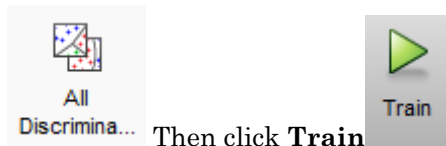
- 2 On the **Apps** tab, in the **Math, Statistics and Optimization** group, click **Classification Learner**.
- 3 On the **Classification Learner** tab, in the **File** section, click **New Session**.

In the New Session dialog box, observe that the app has selected response and predictor variables based on their data type. Petal and sepal length and width are predictors, and species is the response that you want to classify. For this example, do not change the selections.

- 4 Click **Start Session**.

Classification Learner creates a scatter plot of the data.

- 5 Use the scatter plot to visualize which variables are useful for predicting the response. Select different variables in the X- and Y-axis controls. Observe which variables separate the classes most clearly.
- 6 To train both discriminant analysis classifiers, on the **Classification Learner** tab, in the **Model Type** section, click the down arrow to expand the list of classifiers, and under **Discriminant Analysis**, click **All Discriminants**.

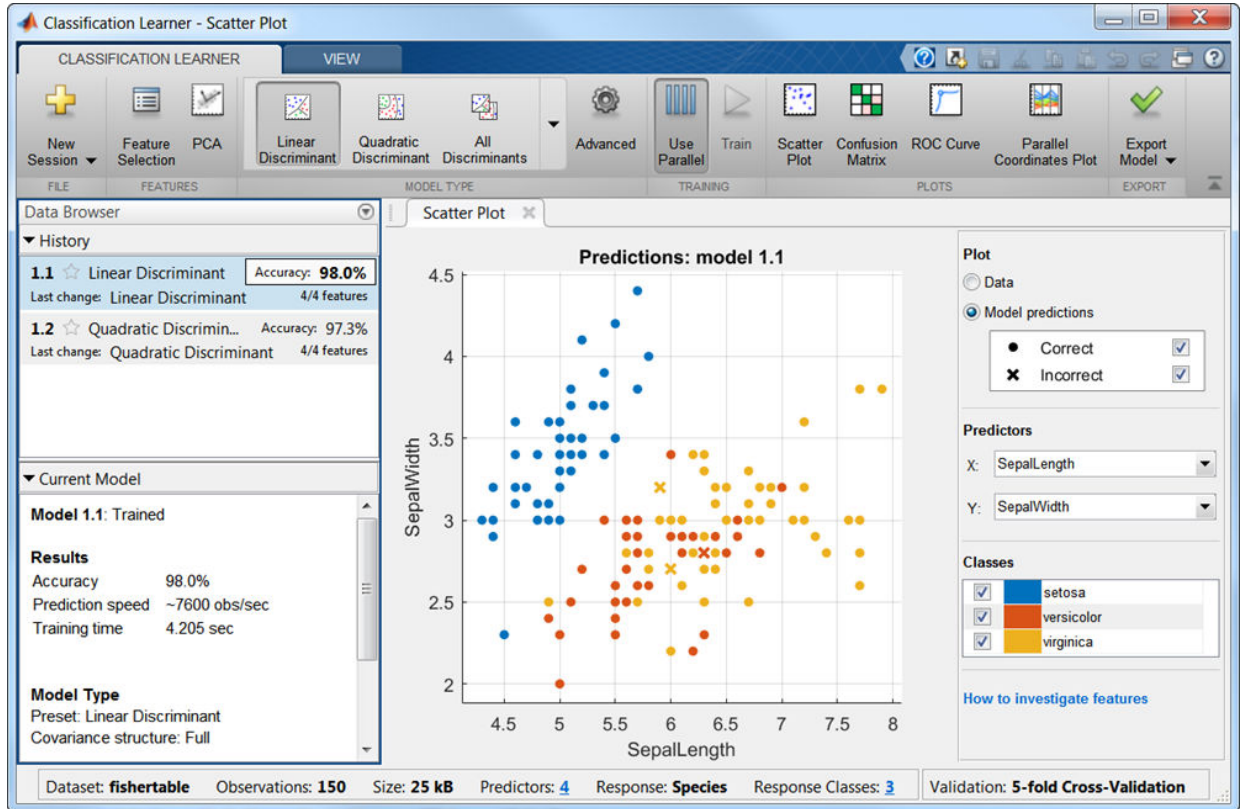


---

**Tip** If you have Parallel Computing Toolbox then the first time you click **Train** you see a dialog while the app opens a parallel pool of workers. After the pool opens, you can train multiple classifiers at once and continue working.

---

Classification Learner trains one of each classification option in the gallery, linear and quadratic discriminants, and highlights the best score. The app outlines in a box the **Accuracy** score of the best model.



- 7 Select a model in the History list to view the results. Examine the scatter plot for the trained model and try plotting different predictors. Misclassified points are shown as an X.
- 8 To inspect the accuracy of the predictions in each class, on the **Classification Learner** tab, in the **Plots** section, click **Confusion Matrix**. View the matrix of true class and predicted class results.
- 9 Select the other model in the list to compare.

For information on the strengths of different model types, see “Discriminant Analysis” on page 23-34.

- 10 Choose the best model in the History list (the best score is highlighted in a box). To improve the model, try including different features in the model. See if you can improve the model by removing features with low predictive power.

On the **Classification Learner** tab, in the **Features** section, click **Feature Selection**. In the Feature Selection dialog box, specify predictors to remove from the model, and click **Train** to train a new model using the new options. Compare results among the classifiers in the History list.

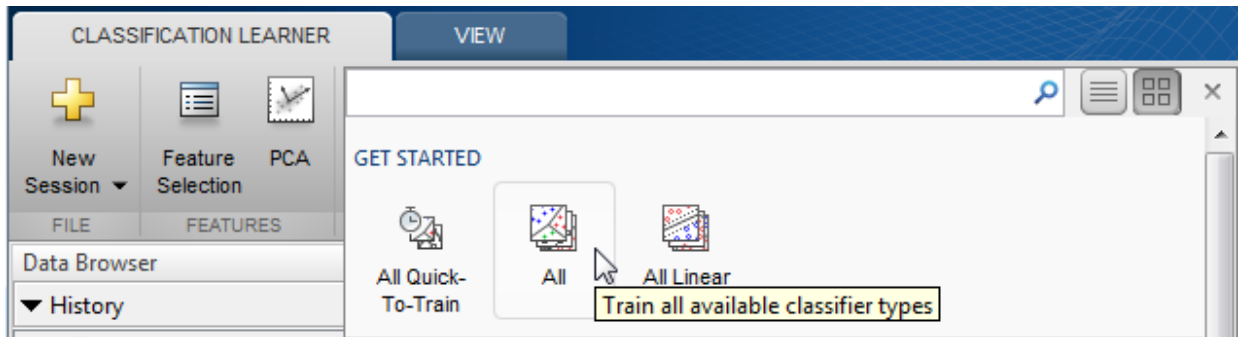
- 11 To investigate features to include or exclude, use the parallel coordinates plot. On the **Classification Learner** tab, in the **Plots** section, select **Parallel Coordinates Plot**.
- 12 Choose the best model in the History list. To try to improve the model further, try changing classifier settings. On the **Classification Learner** tab, in the **Model Type** section, click **Advanced**. Try changing a setting, then train the new model by clicking **Train**. For information on settings, see “Discriminant Analysis” on page 23-34.
- 13 To export the trained model to the workspace, select the Classification Learner tab and click **Export model**. See “Export Classification Model to Predict New Data” on page 23-61.
- 14 To examine the code for training this classifier, select **Export Model > Generate Code**.

Use the same workflow to evaluate and compare the other classifier types you can train in Classification Learner.

To try all the classifier model presets available for your data set:

- 1 Click the arrow on the far right of the **Model Type** section to expand the list of classifiers.
- 2 Click **All**, then click **Train**.





To learn about other classifier types, see “Train Classification Models in Classification Learner App” on page 23-11.

## See Also

### Related Examples

- “Train Classification Models in Classification Learner App” on page 23-11
- “Select Data and Validation for Classification Problem” on page 23-20
- “Choose Classifier Options” on page 23-26
- “Feature Selection and Feature Transformation Using Classification Learner App” on page 23-47
- “Assess Classifier Performance in Classification Learner” on page 23-54
- “Export Classification Model to Predict New Data” on page 23-61
- “Train Decision Trees Using Classification Learner App” on page 23-67

## Train Logistic Regression Classifiers Using Classification Learner App

This example shows how to construct logistic regression classifiers in the Classification Learner app, using the `ionosphere` data set that contains two classes. You can use logistic regression with two classes in Classification Learner. In the `ionosphere` data, the response variable is categorical with two levels: `g` represents good radar returns, and `b` represents bad radar returns.

- 1 In MATLAB, load the `ionosphere` data set and define some variables from the data set to use for a classification.

```
load ionosphere
ionosphere = array2table(X);
ionosphere.Group = Y;
```

- 2 On the **Apps** tab, in the **Math, Statistics and Optimization** group, click **Classification Learner**.
- 3 On the **Classification Learner** tab, in the **File** section, click **New Session**.

In the New Session dialog box, observe that the app has selected `Group` for the response variable, and the rest as predictors. `Group` has two levels.

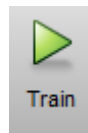
- 4 Click **Start Session**.

Classification Learner creates a scatter plot of the data.

- 5 Use the scatter plot to visualize which variables are useful for predicting the response. Select different variables in the X- and Y-axis controls. Observe which variables separate the class colors most clearly.
- 6 To train the logistic regression classifier, on the **Classification Learner** tab, in the **Model Type** section, click the down arrow to expand the list of classifiers, and under **Logistic Regression Classifiers**, click **Logistic Regression**.

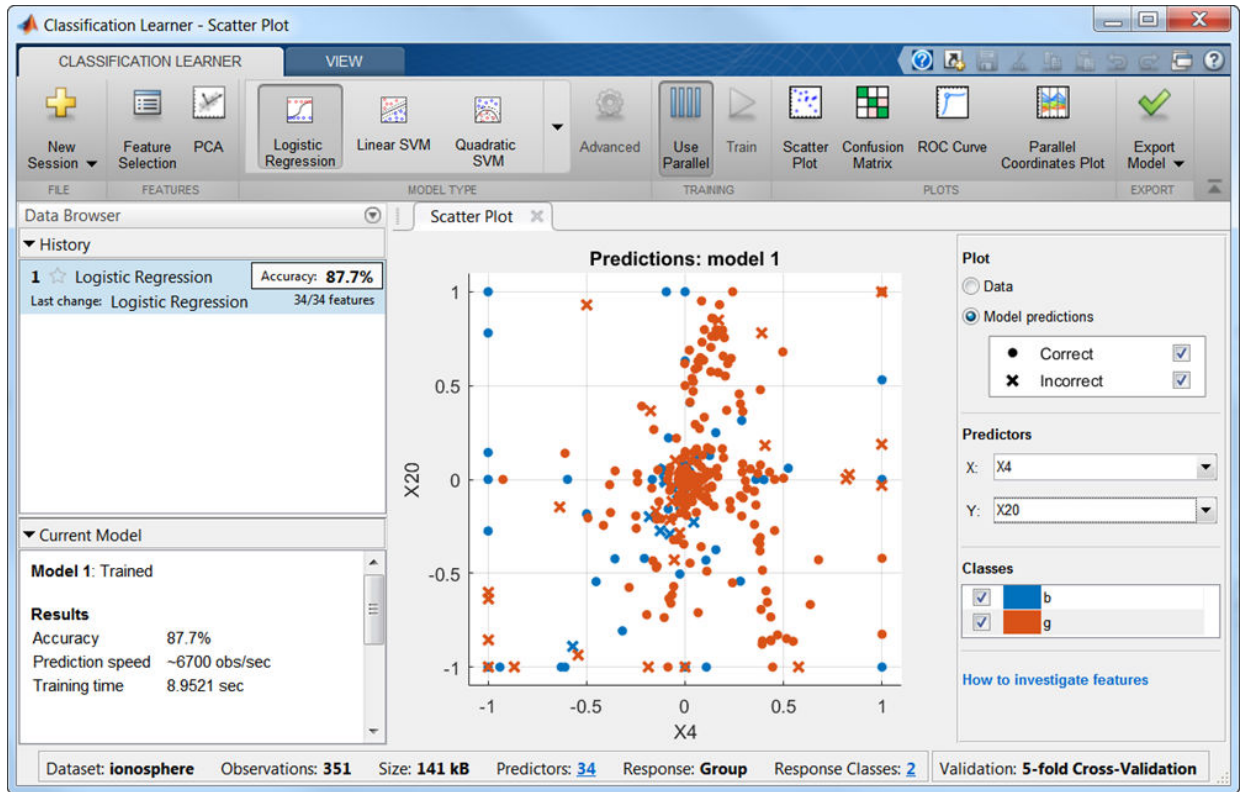


Then click **Train**.



**Tip** If you have Parallel Computing Toolbox then the first time you click **Train** you see a dialog while the app opens a parallel pool of workers. After the pool opens, you can train multiple classifiers at once and continue working.

Classification Learner trains the model. The app outlines in a box the **Accuracy** score of the best model (in this case, there is only one model).



- 7 Select the model in the History list to view the results. Examine the scatter plot for the trained model and try plotting different predictors. Misclassified points are shown as an X.
- 8 To inspect the accuracy of the predictions in each class, on the **Classification Learner** tab, in the **Plots** section, click **Confusion Matrix**. View the matrix of true class and predicted class results.

- 9 Choose the best model in the History list (the best score is highlighted in a box). To improve the model, try including different features in the model. See if you can improve the model by removing features with low predictive power.

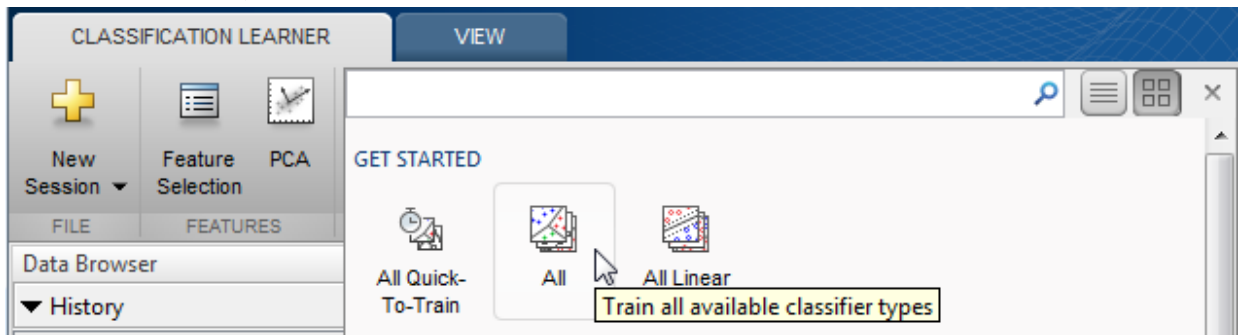
On the **Classification Learner** tab, in the **Features** section, click **Feature Selection**. In the Feature Selection dialog box, specify predictors to remove from the model, and click **Train** to train a new model using the new options. Compare results among the classifiers in the History list.

- 10 To investigate features to include or exclude, use the parallel coordinates plot. On the **Classification Learner** tab, in the **Plots** section, select **Parallel Coordinates Plot**.
- 11 To export the trained model to the workspace, select the Classification Learner tab and click **Export model**. See “Export Classification Model to Predict New Data” on page 23-61.
- 12 To examine the code for training this classifier, select **Export Model > Generate Code**.

Use the same workflow to evaluate and compare the other classifier types you can train in Classification Learner.

To try all the classifier model presets available for your data set:

- 1 Click the arrow on the far right of the **Model Type** section to expand the list of classifiers.
- 2 Click **All**, then click **Train**.



To learn about other classifier types, see “Train Classification Models in Classification Learner App” on page 23-11.

## See Also

### Related Examples

- “Train Classification Models in Classification Learner App” on page 23-11
- “Select Data and Validation for Classification Problem” on page 23-20
- “Choose Classifier Options” on page 23-26
- “Logistic Regression” on page 23-35
- “Feature Selection and Feature Transformation Using Classification Learner App” on page 23-47
- “Assess Classifier Performance in Classification Learner” on page 23-54
- “Export Classification Model to Predict New Data” on page 23-61
- “Train Decision Trees Using Classification Learner App” on page 23-67

## Train Support Vector Machines Using Classification Learner App

This example shows how to construct support vector machine (SVM) classifiers in the Classification Learner app, using the `ionosphere` data set that contains two classes. You can use a support vector machine (SVM) with two or more classes in Classification Learner. An SVM classifies data by finding the best hyperplane that separates all data points of one class from those of another class. In the `ionosphere` data, the response variable is categorical with two levels: `g` represents good radar returns, and `b` represents bad radar returns.

- 1 In MATLAB, load the `ionosphere` data set and define some variables from the data set to use for a classification.

```
load ionosphere
ionosphere = array2table(X);
ionosphere.Group = Y;
```

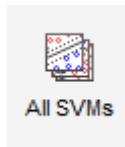
- 2 On the **Apps** tab, in the **Math, Statistics and Optimization** group, click **Classification Learner**.
- 3 On the **Classification Learner** tab, in the **File** section, click **New Session**.

In the New Session dialog box, observe that the app has selected response and predictor variables based on their data type. The response variable `Group` has two levels. All the other variables are predictors.

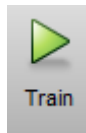
- 4 Click **Start Session**.

Classification Learner creates a scatter plot of the data.

- 5 Use the scatter plot to visualize which variables are useful for predicting the response. Select different variables in the X- and Y-axis controls. Observe which variables separate the class colors most clearly.
- 6 To create a selection of SVM models, on the **Classification Learner** tab, in the **Model Type** section, click the down arrow to expand the list of classifiers, and under **Support Vector Machines**, click **All SVMs**.

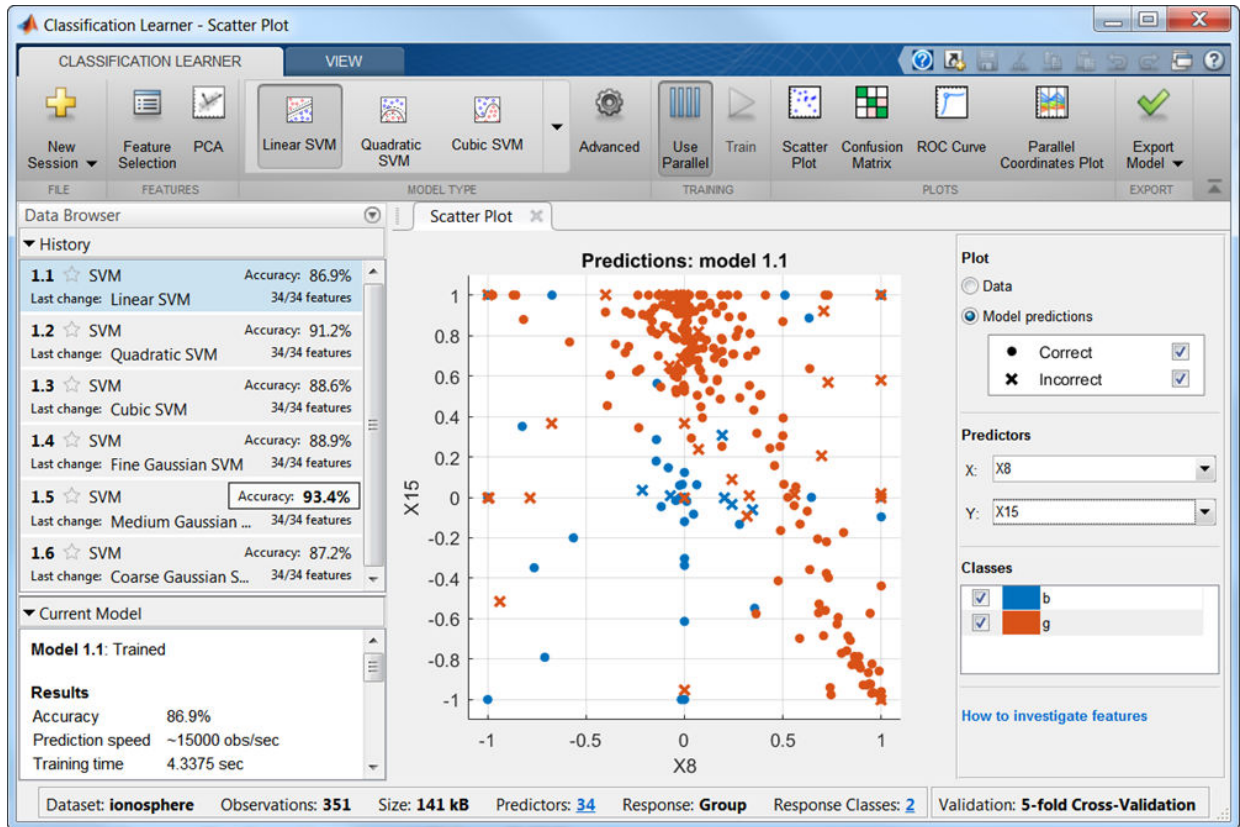


Then click **Train**.



**Tip** If you have Parallel Computing Toolbox then the first time you click **Train** you see a dialog while the app opens a parallel pool of workers. After the pool opens, you can train multiple classifiers at once and continue working.

Classification Learner trains one of each SVM classification option in the gallery, and highlights the best score. The app outlines in a box the **Accuracy** score of the best model.



- 7 Select a model in the History list to view the results. Examine the scatter plot for the trained model and try plotting different predictors. Misclassified points are shown as an X.

- 8 To inspect the accuracy of the predictions in each class, on the **Classification Learner** tab, in the **Plots** section, click **Confusion Matrix**. View the matrix of true class and predicted class results.
- 9 Select the other models in the list to compare.
- 10 Choose the best model in the History list (the best score is highlighted in a box). To improve the model, try including different features in the model. See if you can improve the model by removing features with low predictive power.

On the **Classification Learner** tab, in the **Data Transformation** section, click **Features**. In the Feature Selection dialog box, specify predictors to remove from the model, and click **Train** to train a new model using the new options. Compare results among the classifiers in the History list.

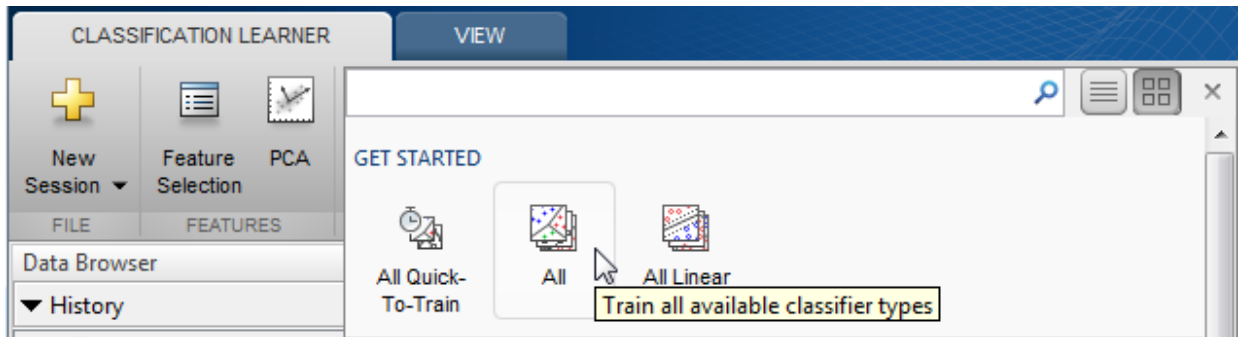
- 11 To investigate features to include or exclude, use the parallel coordinates plot. On the **Classification Learner** tab, in the **Plots** section, select **Parallel Coordinates Plot**.
- 12 Choose the best model in the History list. To try to improve the model further, try changing SVM settings. On the **Classification Learner** tab, in the **Model Type** section, click **Advanced**. Try changing a setting, then train the new model by clicking **Train**. For information on settings, see “Support Vector Machines” on page 23-35.
- 13 To export the trained model to the workspace, select the Classification Learner tab and click **Export model**. See “Export Classification Model to Predict New Data” on page 23-61.
- 14 To examine the code for training this classifier, select **Export Model > Generate Code**. For SVM models, see also “Generate C Code for Prediction” on page 23-64.

Use the same workflow to evaluate and compare the other classifier types you can train in Classification Learner.

To try all the classifier model presets available for your data set:

- 1 Click the arrow on the far right of the **Model Type** section to expand the list of classifiers.
- 2 Click **All**, then click **Train**.





To learn about other classifier types, see “Train Classification Models in Classification Learner App” on page 23-11.

## See Also

### Related Examples

- “Train Classification Models in Classification Learner App” on page 23-11
- “Select Data and Validation for Classification Problem” on page 23-20
- “Choose Classifier Options” on page 23-26
- “Feature Selection and Feature Transformation Using Classification Learner App” on page 23-47
- “Assess Classifier Performance in Classification Learner” on page 23-54
- “Export Classification Model to Predict New Data” on page 23-61
- “Generate C Code for Prediction” on page 23-64
- “Train Decision Trees Using Classification Learner App” on page 23-67

## Train Nearest Neighbor Classifiers Using Classification Learner App

This example shows how to construct nearest neighbors classifiers in the Classification Learner app.

- 1 In MATLAB, load the `fisheriris` data set and define some variables from the data set to use for a classification.

```
fishertable = readtable('fisheriris.csv');
```

- 2 On the **Apps** tab, in the **Math, Statistics and Optimization** group, click **Classification Learner**.
- 3 On the **Classification Learner** tab, in the **File** section, click **New Session**.

In the New Session dialog box, observe that the app has selected response and predictor variables based on their data type. Petal and sepal length and width are predictors, and species is the response that you want to classify. For this example, do not change the selections.

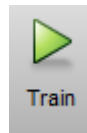
- 4 Click **Start Session**.

The app creates a scatter plot of the data.

- 5 Use the scatter plot to investigate which variables are useful for predicting the response. To visualize the distribution of species and measurements, select different options on the **Variable on X axis** and **Variable on Y axis** menus. Observe which variables separate the species colors most clearly.
- 6 To create a selection of nearest neighbors models, on the **Classification Learner** tab, on the far right of the **Model Type** section, click the arrow to expand the list of classifiers, and under **Nearest Neighbor Classifiers**, click **All KNNs**.



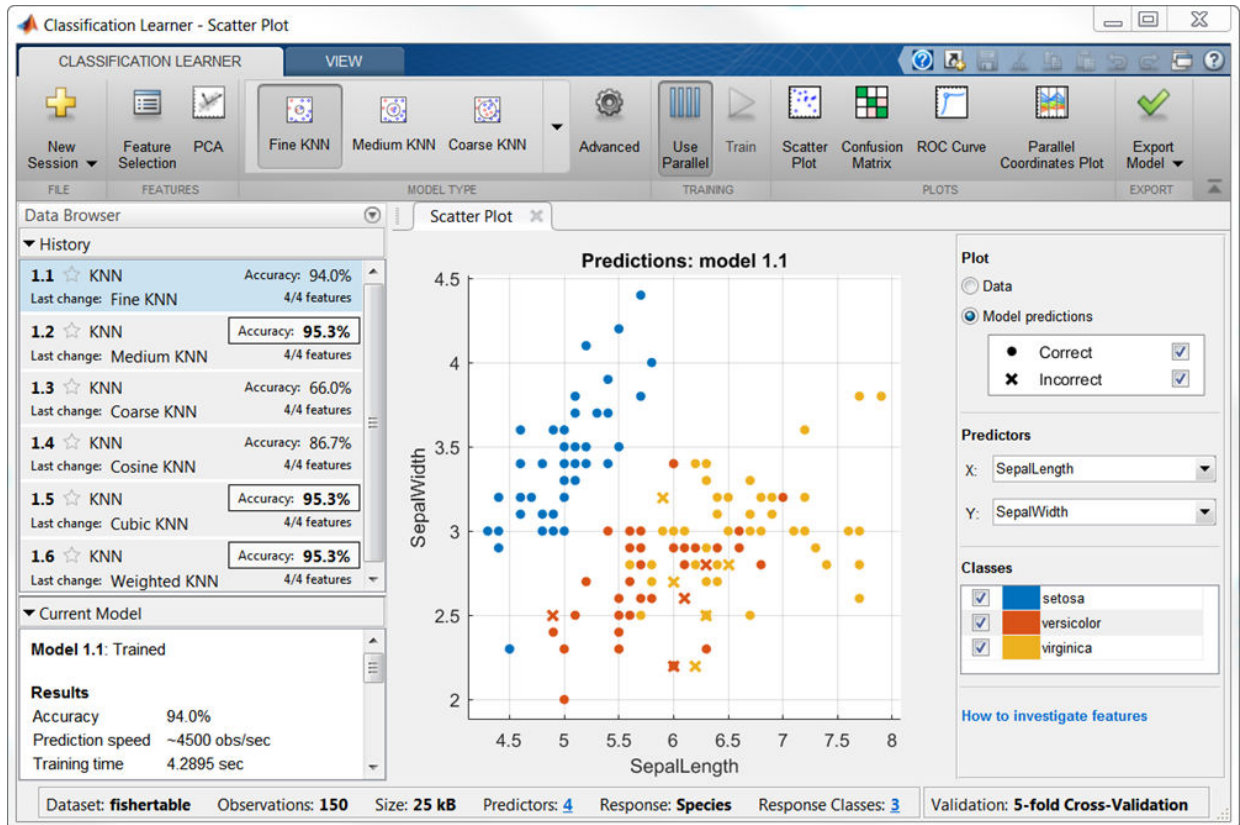
- 7



In the **Training** section, click **Train**.

**Tip** If you have Parallel Computing Toolbox then the first time you click **Train** you see a dialog while the app opens a parallel pool of workers. After the pool opens, you can train multiple classifiers at once and continue working.

Classification Learner trains one of each nearest neighbor classification option in the gallery, and highlights the best score. The app outlines in a box the **Accuracy** score of the best model.



- 8 Select a model in the History list to view the results. Examine the scatter plot for the trained model. An X indicates a misclassified point.
- 9 To inspect the accuracy of the predictions in each class, on the **Classification Learner** tab, in the **Plots** section, click **Confusion Matrix**. View the matrix of true class and predicted class results.

- 10 Select the other models in the list to compare.
- 11 Choose the best model in the History list (the best score is highlighted in a box). To improve the model, try including different features in the model. See if you can improve the model by removing features with low predictive power.

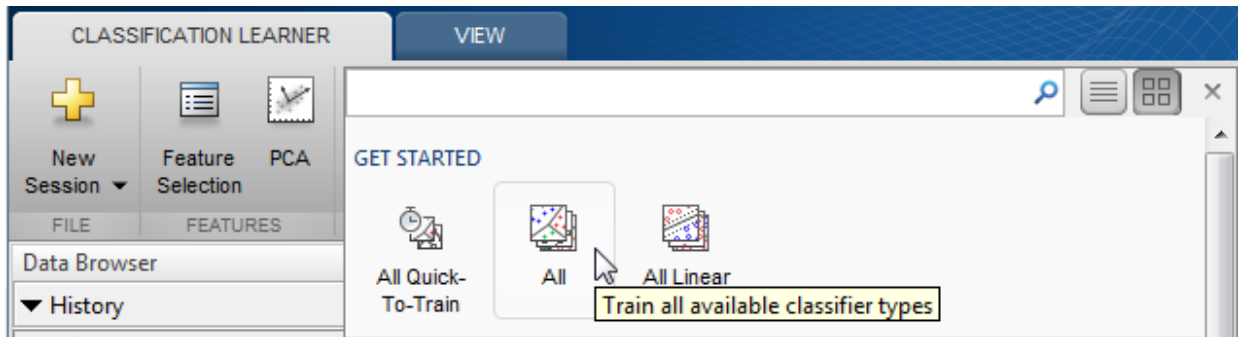
On the **Classification Learner** tab, in the **Data Transformation** section, click **Features**. In the Feature Selection dialog box, select predictors to remove from the model, and click **Train** to train a new model using the new options. Compare results among the classifiers in the History list.

- 12 To investigate features to include or exclude, use the parallel coordinates plot. On the **Classification Learner** tab, in the **Plots** section, select **Parallel Coordinates Plot**.
- 13 Choose the best model in the History list. To try to improve the model further, try changing settings. On the **Classification Learner** tab, in the **Model Type** section, click **Advanced**. Try changing a setting, and then train the new model by clicking **Train**. For information on settings and the strengths of different nearest neighbor model types, see “Nearest Neighbor Classifiers” on page 23-39.
- 14 To export the trained model to the workspace, in the **Export** section of the toolbar, click **Export model**. See “Export Classification Model to Predict New Data” on page 23-61.
- 15 To examine the code for training this classifier, select **Export Model > Generate Code**.

Use the same workflow to evaluate and compare the other classifier types you can train in Classification Learner.

To try all the classifier model presets available for your data set:

- 1 Click the arrow on the far right of the **Model Type** section to expand the list of classifiers.
- 2 Click **All**, then click **Train**.



To learn about other classifier types, see “Train Classification Models in Classification Learner App” on page 23-11.

## See Also

### Related Examples

- “Train Classification Models in Classification Learner App” on page 23-11
- “Select Data and Validation for Classification Problem” on page 23-20
- “Choose Classifier Options” on page 23-26
- “Feature Selection and Feature Transformation Using Classification Learner App” on page 23-47
- “Assess Classifier Performance in Classification Learner” on page 23-54
- “Export Classification Model to Predict New Data” on page 23-61
- “Train Decision Trees Using Classification Learner App” on page 23-67

## Train Ensemble Classifiers Using Classification Learner App

This example shows how to construct ensembles of classifiers in the Classification Learner app. Ensemble classifiers meld results from many weak learners into one high-quality ensemble predictor. Qualities depend on the choice of algorithm, but ensemble classifiers tend to be slow to fit because they often need many learners.

- 1 In MATLAB, load the `fisheriris` data set and define some variables from the data set to use for a classification.

```
fishertable = readtable('fisheriris.csv');
```

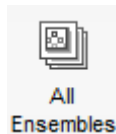
- 2 On the **Apps** tab, in the **Math, Statistics and Optimization** group, click **Classification Learner**.
- 3 On the **Classification Learner** tab, in the **File** section, click **New Session**.

In the New Session dialog box, observe that the app has selected response and predictor variables based on their data type. Petal and sepal length and width are predictors. Species is the response that you want to classify. For this example, do not change the selections.

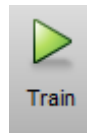
- 4 Click **Start Session**.

Classification Learner creates a scatter plot of the data.

- 5 Use the scatter plot to investigate which variables are useful for predicting the response. Select different variables in the X- and Y-axis controls to visualize the distribution of species and measurements. Observe which variables separate the species colors most clearly.
- 6 To create a selection of ensemble models, on the **Classification Learner** tab, in the **Model Type** section, click the down arrow to expand the list of classifiers, then under **Ensemble Classifiers**, click **All Ensembles**.



- 7

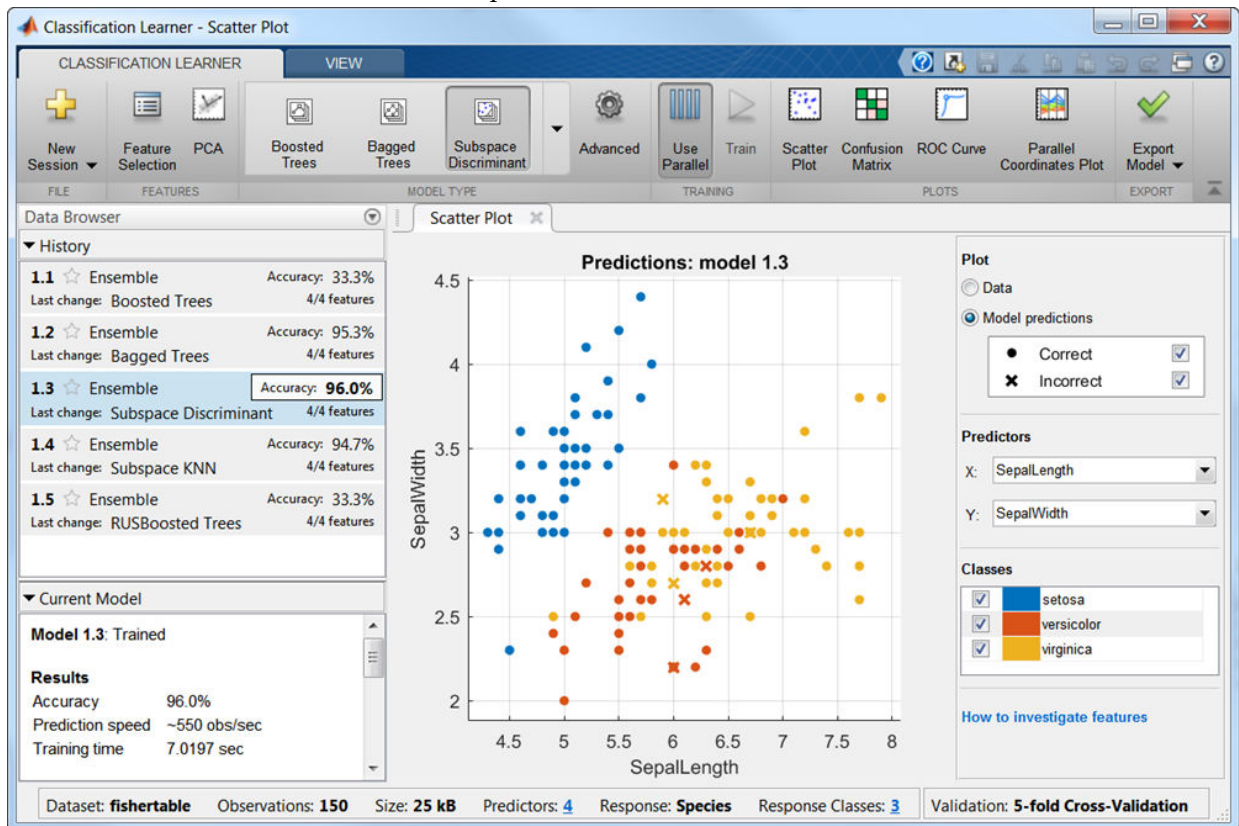


In the **Training** section, click **Train**.

**Tip** If you have Parallel Computing Toolbox then the first time you click **Train** you see a dialog while the app opens a parallel pool of workers. After the pool opens, you can train multiple classifiers at once and continue working.

Classification Learner trains one of each ensemble classification option in the gallery, and highlights the best score. The app outlines in a box the **Accuracy** score of the best model.

- 8 Select a model in the History list to view the results. Examine the scatter plot for the trained model. Misclassified points are shown as an X.



- 9 To inspect the accuracy of the predictions in each class, on the **Classification Learner** tab, in the **Plots** section, click **Confusion Matrix**. View the matrix of true class and predicted class results.

- 10 Select the other models in the list to compare.
- 11 Choose the best model in the History list (the best score is highlighted in the **Accuracy** box). To improve the model, try including different features in the model. See if you can improve the model by removing features with low predictive power.

On the **Classification Learner** tab, in the **Data Transformation** section, click **Features**. In the Feature Selection dialog box, specify predictors to remove from the model, and click **Train** to train a new model using the new options. Compare results among the classifiers in the History list.

- 12 To investigate features to include or exclude, use the scatter and parallel coordinates plots. On the **Classification Learner** tab, in the **Plots** section, select **Parallel Coordinates Plot**.
- 13 Choose the best model in the History list. To try to improve the model further, try changing settings. On the **Classification Learner** tab, in the **Model Type** section, click **Advanced**. Try changing a setting, then train the new model by clicking **Train**.

For information on the settings to try and the strengths of different ensemble model types, see “Ensemble Classifiers” on page 23-42.

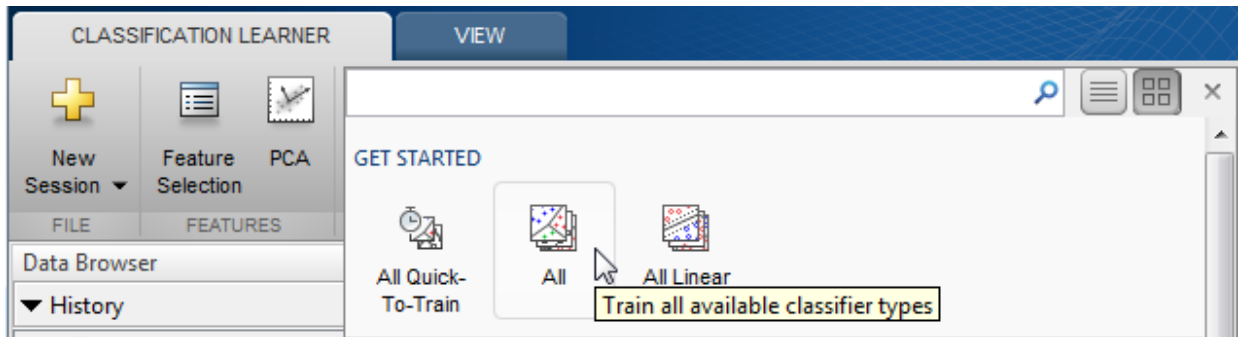
- 14 To export the trained model to the workspace, select the Classification Learner tab and click **Export model**. See “Export Classification Model to Predict New Data” on page 23-61.
- 15 To examine the code for training this classifier, select **Export Model > Generate Code**.

Use the same workflow to evaluate and compare the other classifier types you can train in Classification Learner.

To try all the classifier model presets available for your data set:

- 1 Click the arrow on the far right of the **Model Type** section to expand the list of classifiers.
- 2 Click **All**, then click **Train**.





To learn about other classifier types, see “Train Classification Models in Classification Learner App” on page 23-11.

## See Also

### Related Examples

- “Train Classification Models in Classification Learner App” on page 23-11
- “Select Data and Validation for Classification Problem” on page 23-20
- “Choose Classifier Options” on page 23-26
- “Feature Selection and Feature Transformation Using Classification Learner App” on page 23-47
- “Assess Classifier Performance in Classification Learner” on page 23-54
- “Export Classification Model to Predict New Data” on page 23-61
- “Train Decision Trees Using Classification Learner App” on page 23-67



# Regression Learner

---

- “Train Regression Models in Regression Learner App” on page 24-2
- “Select Data and Validation for Regression Problem” on page 24-10
- “Choose Regression Model Options” on page 24-15
- “Feature Selection and Feature Transformation Using Regression Learner App” on page 24-32
- “Assess Model Performance in Regression Learner App” on page 24-36
- “Export Regression Model to Predict New Data” on page 24-44
- “Train Regression Trees Using Regression Learner App” on page 24-48

## Train Regression Models in Regression Learner App

### In this section...

“What Is Supervised Machine Learning?” on page 24-2

“Automated Regression Model Training” on page 24-3

“Manual Regression Model Training” on page 24-4

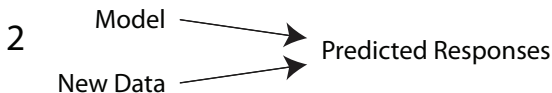
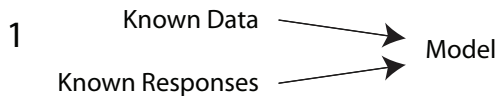
“Parallel Regression Model Training” on page 24-6

“Compare and Improve Regression Models” on page 24-6

### What Is Supervised Machine Learning?

You can use Regression Learner to train regression models to predict data. Using this app, you can explore your data, select features, specify validation schemes, train models, and assess results. You can perform automated training to search for the best regression model type, including linear regression models, regression trees, Gaussian process regression models, support vector machines, and ensembles of regression trees.


Perform supervised machine learning by supplying a known set of observations of input data (predictors) and known responses. Use the observations to train a model that generates predicted responses for new input data. To use the model with new data, or to learn about programmatic regression, you can export the model to the workspace or generate MATLAB code to recreate the trained model.

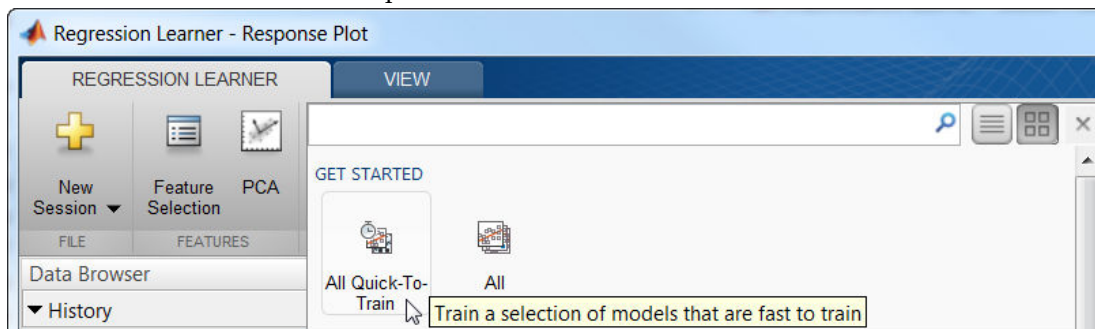


Get started by training a selection of model types. See “Automated Regression Model Training” on page 24-3.

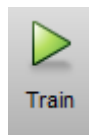
## Automated Regression Model Training

You can use Regression Learner to automatically train a selection of different regression models on your data.

- Get started by automatically training multiple models simultaneously. You can quickly try a selection of models, and then explore promising models interactively.
  - If you already know what model type you want, then you can train individual models instead. See “Manual Regression Model Training” on page 24-4.
- 1 On the **Apps** tab, in the **Math, Statistics and Optimization** group, click **Regression Learner**.
  - 2 Click **New Session** and select data from the workspace or from file. Specify a response variable and variables to use as predictors. See “Select Data and Validation for Regression Problem” on page 24-10.
  - 3 On the **Regression Learner** tab, in the **Model Type** section, click the arrow to expand the list of regression models. Select **All Quick-To-Train** . This option trains all the model presets that are fast to fit.



4



Click **Train**.

---

**Note** If you have Parallel Computing Toolbox, the app trains models in parallel. See “Parallel Regression Model Training” on page 24-6.

---

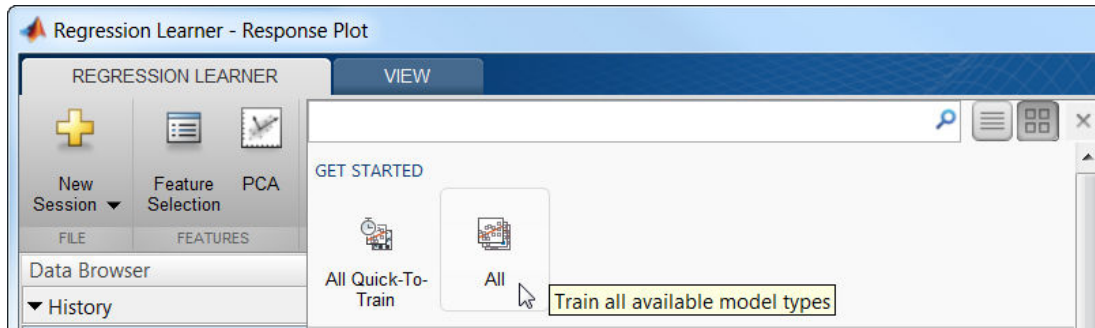
A selection of model types appears in the History list. When the models finish training, the best **RMSE** score is highlighted in a box.

▼ History		
1.1	☆ Linear Regression Last change: Linear	RMSE: 3.3815 77 features
1.2	☆ Tree Last change: Fine Tree	RMSE: 3.3849 77 features
1.3	☆ Tree Last change: Medium Tree	RMSE: <b>3.3102</b> 77 features
1.4	☆ Tree Last change: Coarse Tree	RMSE: 3.8819 77 features

- 5 Click models in the History list to explore results in the plots.

For the next steps, see “Manual Regression Model Training” on page 24-4 or “Compare and Improve Regression Models” on page 24-6.

- 6 To try all the model presets available, click **All** , and then click **Train**.

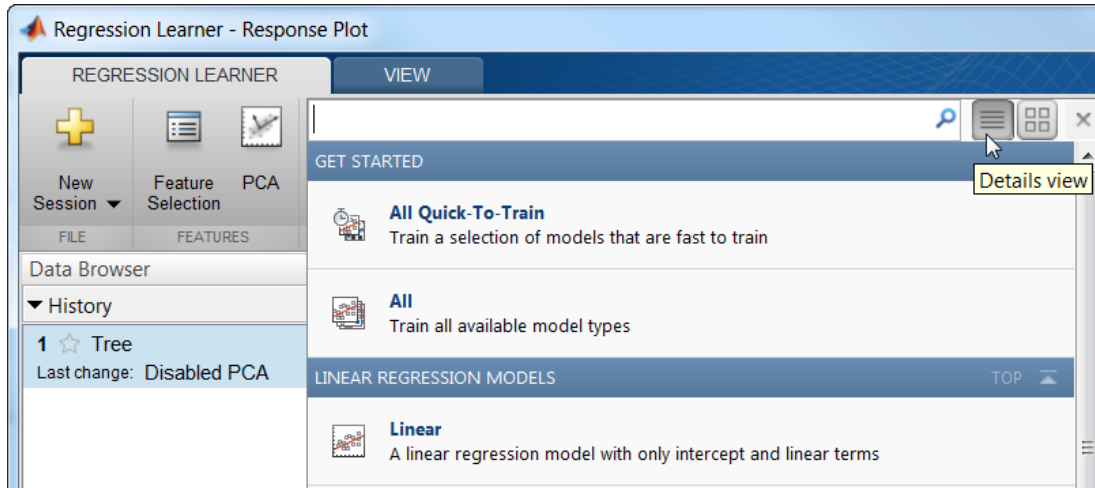


## Manual Regression Model Training

To explore individual model types, you can train models one at a time or train a group of models of the same type.

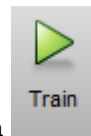
- 1 Choose a model type. On the **Regression Learner** tab, in the **Model Type** section, click a model type. To see all available model options, click the arrow in the **Model Type** section to expand the list of regression models. The options in the gallery are preset starting points with different settings, suitable for a range of different regression problems.

To read descriptions of the models, switch to the details view or hover the mouse over a button to display its tooltip.



For more information on each option, see “Choose Regression Model Options” on page 24-15.

2



After selecting a model, click **Train**.

Repeat to explore different models.

---

**Tip** Select regression trees first. If your trained models do not predict the response accurately enough, then try other models with higher flexibility. To avoid overfitting, look for a less flexible model that provides sufficient accuracy.

---

3 If you want to try all model types or train a group of the same type, then select one of the **All** options in the gallery.

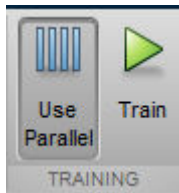
For next steps, see “Compare and Improve Regression Models” on page 24-6.

## Parallel Regression Model Training

You can train models in parallel using Regression Learner if you have Parallel Computing Toolbox. When you train models, the app automatically starts a parallel pool of workers, unless you turn off the default parallel preference **Automatically create a parallel pool**. If a pool is already open, the app uses it for training. Parallel training allows you to train multiple models simultaneously and continue working.

- 1 The first time you click **Train**, you see a dialog box while the app opens a parallel pool of workers. After the pool opens, you can train multiple models at once.
- 2 When models are training in parallel, you see progress indicators on each training and queued model in the History list. If you want, you can cancel individual models. During training, you can examine results and plots from models, and initiate training of more models.

To control parallel training, toggle the **Use Parallel** button on the app toolstrip. (The **Use Parallel** button is only available if you have Parallel Computing Toolbox.)



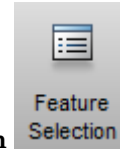
If you have Parallel Computing Toolbox, then parallel training is available in Regression Learner, and you do not need to set the `UseParallel` option of the `statset` function. If you turn off the parallel preference to **Automatically create a parallel pool**, then the app does not start a pool for you without asking first.

## Compare and Improve Regression Models

- 1 Click models in the History list to explore the results in the plots. Compare model performance by inspecting results in the plots. Examine the **RMSE** score reported in the History list for each model. See “Assess Model Performance in Regression Learner App” on page 24-36.

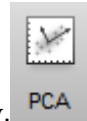


- 2 Select the best model in the History list and then try including and excluding



different features in the model. Click **Feature Selection**

Try the response plot to help you identify features to remove. See if you can improve the model by removing features with low predictive power. Specify predictors to include in the model, and train new models using the new options. Compare results among the models in the History list.




You also can try transforming features with PCA to reduce dimensionality.

See “Feature Selection and Feature Transformation Using Regression Learner App” on page 24-32.

- 3 Improve the model further by changing model parameter settings in the Advanced dialog box. Then, train using the new options. To learn how to control model flexibility, see “Choose Regression Model Options” on page 24-15.

If feature selection, PCA, or new parameter settings improve your model, try

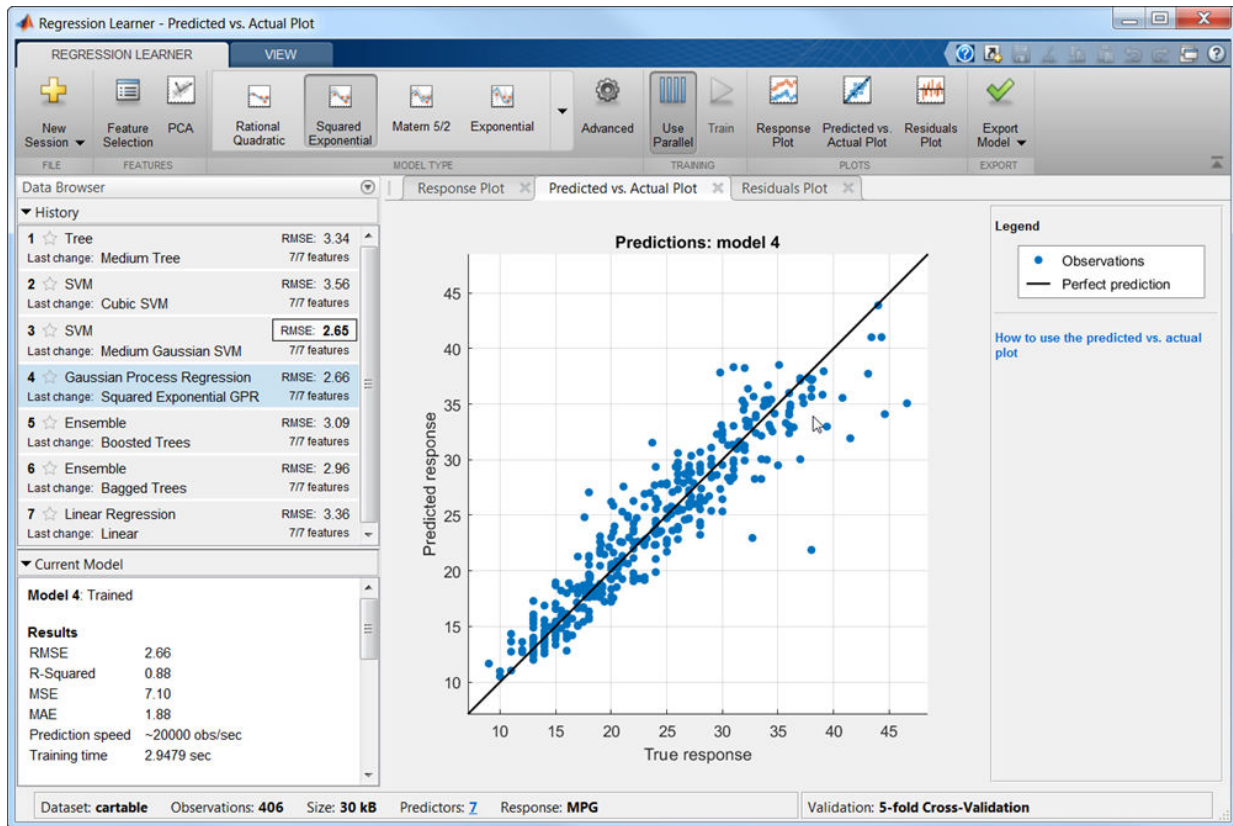
training **All**  model types with the new settings. See if another model type does better with the new settings.

---

**Tip** To avoid overfitting, look for a less flexible model that provides sufficient accuracy. For example, look for simple models, such as regression trees that are fast and easy to interpret. If your models are not accurate enough, then try other models with higher flexibility, such as ensembles. To learn about the model flexibility, see “Choose Regression Model Options” on page 24-15.

---

This figure shows the app with a History list containing various regression model types.



**Tip** For a step-by-step example comparing different regression models, see “Train Regression Trees Using Regression Learner App” on page 24-48.

Next, you can generate code to train the model with different data or export trained models to the workspace to make predictions using new data. See “Export Regression Model to Predict New Data” on page 24-44.

## See Also

### Related Examples

- “Select Data and Validation for Regression Problem” on page 24-10
- “Choose Regression Model Options” on page 24-15
- “Feature Selection and Feature Transformation Using Regression Learner App” on page 24-32
- “Assess Model Performance in Regression Learner App” on page 24-36
- “Export Regression Model to Predict New Data” on page 24-44
- “Train Regression Trees Using Regression Learner App” on page 24-48

## Select Data and Validation for Regression Problem

In this section...
“Select Data from Workspace” on page 24-10
“Import Data from File” on page 24-11
“Example Data for Regression” on page 24-11
“Choose Validation Scheme” on page 24-12

### Select Data from Workspace

---

**Tip** In Regression Learner, tables are the easiest way to work with your data, because they can contain numeric and label data. Use the Import Tool to bring your data into the MATLAB workspace as a table, or use the table functions to create a `table` from workspace variables. See “Tables” (MATLAB).

If your predictors are a matrix and the response is a vector, combine them into a table using the `table` function.

---

- 1 Load your data into the MATLAB workspace.

Predictor variables can be categorical arrays, cell arrays of character vectors, character arrays, logical vectors, or numeric vectors. The response variable must be a floating-point vector (single or double precision).

For example data sets, see “Example Data for Regression” on page 24-11.

- 2 On the **Apps** tab, click **Regression Learner** to open the app.
- 3 On the **Regression Learner** tab, in the **File** section, click **New Session**.
- 4 In the New Session dialog box, select a table or matrix from the workspace variables.

If you select a matrix, choose whether to use rows or columns for observations by clicking the option buttons.

- 5 Observe the roles the app selects for the variables based on their data type. The app tries to select a suitable response variable, and all other variables are predictors. Change the selections if needed. Add or remove predictors using the check boxes. Add or remove all predictors by clicking **Add All** or **Remove All**. You can also add

or remove multiple predictors by selecting them in the table, and then clicking **Add N** or **Remove N**, where **N** is the number of selected predictors. The **Add All** and **Remove All** buttons change to **Add N** and **Remove N** when you select multiple predictors.

- 6 Click **Start Session** to accept the default validation scheme and continue. The default validation option is 5-fold cross-validation, which protects against overfitting.

---

**Tip** If you have a large data set, you might want to switch to holdout validation. To learn more, see “Choose Validation Scheme” on page 24-12.

---

For next steps, see “Train Regression Models in Regression Learner App” on page 24-2.

## Import Data from File

- 1 On the **Regression Learner** tab, in the **File** section, select **New Session > From File**.
- 2 Select a file type in the list, such as spreadsheets, text files, or comma-separated values (.csv) files, or select **All Files** to browse for other file types such as .dat.

## Example Data for Regression

To get started using Regression Learner, try these example data sets.

Name	Size	Description
Cars	Number of predictors: 7 Number of observations: 406 Response: MPG (miles per gallon)	Data on different car models, 1970–1982. Predict the fuel economy (in miles per gallon), or one of the other characteristics.  For a step-by-step example, see “Train Regression Trees Using Regression Learner App” on page 24-48.
	Create a table from variables in the <code>carbig.mat</code> file:  <pre>load carbig cartable = table(Acceleration, Cylinders, Displacement, ... Horsepower, Model_Year, Weight, Origin, MPG);</pre>	

Name	Size	Description
Abalone	Number of predictors: 8 Number of observations: 4177 Response: Rings	Measurements of abalone (a group of sea snails). Predict the age of abalones, which is closely related to the number of rings in their shells.
	Download the data from the UCI Machine Learning Repository and save it in your current folder. Read the data into a table and specify the variable names.  <pre>url = 'https://archive.ics.uci.edu/ml/machine-learning-databases/abalone/'; websave('abalone.csv',url); varnames = {'Sex'; 'Length'; 'Diameter'; 'Height'; 'Whole_weight'; ... 'Shucked_weight'; 'Viscera_weight'; 'Shell_weight'; 'Rings'}; abalonetable = readtable('abalone.csv'); abalonetable.Properties.VariableNames = varnames;</pre>	
Hospital	Number of predictors: 5 Number of observations: 100 Response: BloodPressure_2	Simulated hospital data. Predict the blood pressure of patients.
	Create a table from the hospital variable in the hospital.mat file:  <pre>load hospital.mat hospitaltable = dataset2table(hospital(:,2:end-1));</pre>	

## Choose Validation Scheme

Choose a validation method to examine the predictive accuracy of the fitted models. Validation estimates model performance on new data, and helps you choose the best model. Validation protects against overfitting. A model that is too flexible and suffers from overfitting has a worse validation accuracy. Choose a validation scheme before training any models so that you can compare all the models in your session using the same validation scheme.

---

**Tip** Try the default validation scheme and click **Start Session** to continue. The default option is 5-fold cross-validation, which protects against overfitting.

If you have a large data set and training the models takes too long using cross-validation, reimport your data and try the faster holdout validation instead.

---

- **Cross-Validation:** Select the number of folds (or divisions) to partition the data set using the slider control.

If you choose  $k$  folds, then the app:

- 1 Partitions the data into  $k$  disjoint sets or folds
- 2 For each fold:
  - a Trains a model using the out-of-fold observations
  - b Assesses model performance using in-fold data
- 3 Calculates the average test error over all folds

This method gives a good estimate of the predictive accuracy of the final model trained using the full data set. The method requires multiple fits, but makes efficient use of all the data, so it works well for small data sets.

- **Holdout Validation:** Select a percentage of the data to use as a validation set using the slider control. The app trains a model on the training set and assesses its performance with the validation set. The model used for validation is based on only a portion of the data, so holdout validation is appropriate only for large data sets. The final model is trained using the full data set.
- **No Validation:** No protection against overfitting. The app uses all the data for training and computes the error rate on the same data. Without any test data, you get an unrealistic estimate of the model's performance on new data. That is, the training sample accuracy is likely to be unrealistically high, and the predictive accuracy is likely to be lower.

To help you avoid overfitting to the training data, choose a validation scheme instead.

---

**Note** The validation scheme only affects the way that Regression Learner computes validation metrics. The final model is always trained using the full data set.

---

All the models you train after selecting data use the same validation scheme that you select in this dialog box. You can compare all the models in your session using the same validation scheme.

To change the validation selection and train new models, you can select data again, but you lose any trained models. The app warns you that importing data starts a new session. Save any trained models you want to keep to the workspace, and then import the data.

For next steps training models, see “Train Regression Models in Regression Learner App” on page 24-2.

## See Also

### Related Examples

- “Train Regression Models in Regression Learner App” on page 24-2
- “Choose Regression Model Options” on page 24-15
- “Feature Selection and Feature Transformation Using Regression Learner App” on page 24-32
- “Assess Model Performance in Regression Learner App” on page 24-36
- “Export Regression Model to Predict New Data” on page 24-44
- “Train Regression Trees Using Regression Learner App” on page 24-48





## Choose Regression Model Options

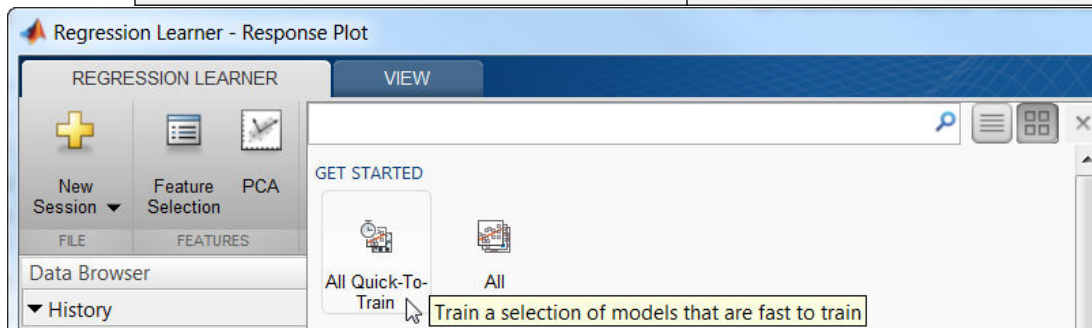
### In this section...

- “Choose Regression Model Type” on page 24-15
- “Linear Regression Models” on page 24-18
- “Regression Trees” on page 24-21
- “Support Vector Machines” on page 24-23
- “Gaussian Process Regression Models” on page 24-26
- “Ensembles of Trees” on page 24-29

## Choose Regression Model Type

You can use the Regression Learner app to automatically train a selection of different models on your data. Use automated training to quickly try a selection of model types, and then explore promising models interactively. To get started, try these options first

Get Started Regression Model Buttons	Description
<b>All Quick-To-Train</b> 	Try the <b>All Quick-To-Train</b> button first. The app trains all model types that are typically quick to train.
<b>All</b> 	Use the <b>All</b> button to train all available model types. Trains every type regardless of any prior trained models. Can be time-consuming.



To learn more about automated model training, see “Automated Regression Model Training” on page 24-3.

If you want to explore models one at a time, or if you already know what model type you want, you can select individual models or train a group of the same type. To see all available regression model options, on the **Regression Learner** tab, click the arrow in the **Model Type** section to expand the list of regression models. The options in the gallery are preset starting points with different settings, suitable for a range of different regression problems.






For help choosing the best model type for your problem, see the tables showing typical characteristics of different regression model types. Decide on the tradeoff you want in speed, flexibility, and interpretability. The best model type depends on your data.

---

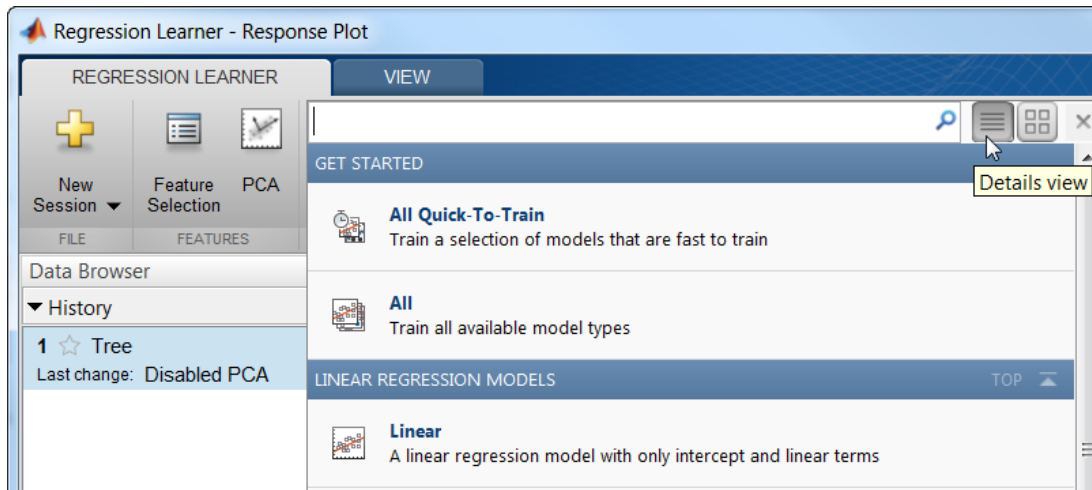
**Tip** To avoid overfitting, look for a less flexible model that provides sufficient accuracy. For example, look for simple models such as regression trees that are fast and easy to interpret. If the models are not accurate enough predicting the response, choose other models with higher flexibility, such as ensembles. To control flexibility, see the details for each model type.

---

### Characteristics of Regression Model Types

Regression Model Type	Interpretability
“Linear Regression Models” on page 24-18 	Easy
“Regression Trees” on page 24-21 	Easy
“Support Vector Machines” on page 24-23 	Easy for linear SVMs. Hard for other kernels.
“Gaussian Process Regression Models” on page 24-26 	Hard
“Ensembles of Trees” on page 24-29 	Hard

To read a description of each model in Regression Learner, switch to the details view in the list of all model presets.




---

**Tip** The options in the **Model Type** gallery are preset starting points with different settings. After you choose a model type, such as regression trees, try training all the presets to see which one produces the best model with your data.

---

For workflow instructions, see “Train Regression Models in Regression Learner App” on page 24-2.

### Categorical Predictor Support

In Regression Learner, all model types support categorical predictors.

---

**Tip** If you have categorical predictors with many unique values, training linear models with interaction or quadratic terms and stepwise linear models can use a lot of memory. If the model fails to train, try removing these categorical predictors.


---

### Linear Regression Models





Linear regression models have predictions that are linear in the model parameters, are easy to interpret, and are fast for making predictions. These characteristics make linear regression models popular models to try first. However, the highly constrained form of these models means that they often have low predictive accuracy. After fitting a linear

regression model, try creating more flexible models, such as regression trees, and compare the results.

---

**Tip** In the **Model Type** gallery, click **All Linear**  to try each of the linear regression options and see which settings produce the best model with your data. Select the best model in the History list and try to improve that model by using feature selection and changing some advanced options.

---

Regression Model Type	Interpretability	Model Flexibility
<b>Linear</b> 	Easy	Very low
<b>Interactions Linear</b> 	Easy	Medium
<b>Robust Linear</b> 	Easy	Very low. Less sensitive to outliers, but can be slow to train.
<b>Stepwise Linear</b> 	Easy	Medium

---

**Tip** For a workflow example, see “Train Regression Trees Using Regression Learner App” on page 24-48.

---

### Advanced Linear Regression Options

Regression Learner uses the `fitlm` function to train Linear, Interactions Linear, and Robust Linear models. The app uses the `stepwiselm` function to train Stepwise Linear models.

For Linear, Interactions Linear, and Robust Linear models you can set these options:

- **Terms**

Specify which terms to use in the linear model. You can choose from:

- **Linear.** A constant term and linear terms in the predictors
- **Interactions.** A constant term, linear terms, and interaction terms between the predictors
- **Pure Quadratic.** A constant term, linear terms, and terms that are purely quadratic in each of the predictors
- **Quadratic.** A constant term, linear terms, and quadratic terms (including interactions)
- **Robust option**

Specify whether to use a robust objective function and make your model less sensitive to outliers. With this option, the fitting method automatically assigns lower weights to data points that are more likely to be outliers.

Stepwise linear regression starts with an initial model and systematically adds and removes terms to the model based on the explanatory power of these incrementally larger and smaller models. For Stepwise Linear models, you can set these options:

- **Initial terms**

Specify the terms that are included in the initial model of the stepwise procedure. You can choose from `Constant`, `Linear`, `Interactions`, `Pure Quadratic`, and `Quadratic`.

- **Upper bound on terms**

Specify the highest order of the terms that the stepwise procedure can add to the model. You can choose from `Linear`, `Interactions`, `Pure Quadratic`, and `Quadratic`.

- **Maximum number of steps**

Specify the maximum number of different linear models that can be tried in the stepwise procedure. To speed up training, try reducing the maximum number of steps. Selecting a small maximum number of steps decreases your chances of finding a good model.


---




**Tip** If you have categorical predictors with many unique values, training linear models with interaction or quadratic terms and stepwise linear models can use a lot of memory. If the model fails to train, try removing these categorical predictors.

---

## Regression Trees

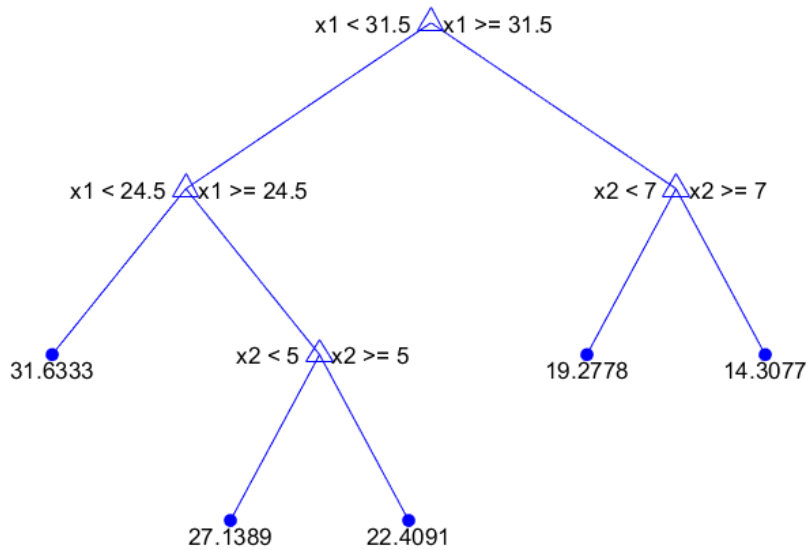
Regression trees are easy to interpret, fast for fitting and prediction, and low on memory usage. Try to grow smaller trees with fewer larger leaves to prevent overfitting. Control the leaf size with the **Minimum leaf size** setting.

**Tip** In the **Model Type** gallery, click **All Trees**  to try each of the regression tree options and see which settings produce the best model with your data. Select the best model in the History list, and try to improve that model by using feature selection and changing some advanced options.

Regression Model Type	Interpretability	Model Flexibility
<b>Fine Tree</b> 	Easy	High  Many small leaves for a highly flexible response function (Minimum leaf size is 4.)
<b>Medium Tree</b> 	Easy	Medium  Medium-sized leaves for a less flexible response function (Minimum leaf size is 12.)
<b>Coarse Tree</b> 	Easy	Low  Few large leaves for a coarse response function (Minimum leaf size is 36.)

To predict a response of a regression tree, follow the tree from the root (beginning) node down to a leaf node. The leaf node contains the value of the response.

Statistics and Machine Learning Toolbox trees are binary. Each step in a prediction involves checking the value of one predictor variable. For example, here is a simple regression tree



This tree predicts the response based on two predictors,  $x_1$  and  $x_2$ . To make a prediction, start at the top node. At each node, check the values of the predictors to decide which branch to follow. When the branches reach a leaf node, the response is set to the value corresponding to that node.

You can visualize your regression tree model by exporting the model from the app, and then entering:

```
view(trainedModel.RegressionTree, 'Mode', 'graph')
```

---

**Tip** For a workflow example, see “Train Regression Trees Using Regression Learner App” on page 24-48.

---

### Advanced Regression Tree Options

The Regression Learner app uses the `fitrtree` function to train regression trees. You can set these options:



- **Minimum leaf size**

Specify the minimum number of training samples used to calculate the response of each leaf node. When you grow a regression tree, consider its simplicity and predictive power. To change the minimum leaf size, click the buttons or enter a positive integer value in the **Minimum leaf size** box.

- A fine tree with many small leaves is usually highly accurate on the training data. However, the tree might not show comparable accuracy on an independent test set. A very leafy tree tends to overfit, and its validation accuracy is often far lower than its training (or resubstitution) accuracy.
- In contrast, a coarse tree with fewer large leaves does not attain high training accuracy. But a coarse tree can be more robust in that its training accuracy can be near that of a representative test set.

---

**Tip** Decrease the **Minimum leaf size** to create a more flexible model.

---

- **Surrogate decision splits** — For missing data only.

Specify surrogate use for decision splits. If you have data with missing values, use surrogate splits to improve the accuracy of predictions.


When you set **Surrogate decision splits** to On, the regression tree finds at most 10 surrogate splits at each branch node. To change the number of surrogate splits, click the buttons or enter a positive integer value in the **Maximum surrogates per node** box.

When you set **Surrogate decision splits** to Find All, the regression tree finds all surrogate splits at each branch node. The Find All setting can use considerable time and memory.







## Support Vector Machines

You can train regression support vector machines (SVMs) in Regression Learner. Linear SVMs are easy to interpret, but can have low predictive accuracy. Nonlinear SVMs are more difficult to interpret, but can be more accurate.

---

**Tip** In the **Model Type** gallery, click **All SVMs**  to try each of the SVM options and see which settings produce the best model with your data. Select the best model in the

History list, and try to improve that model by using feature selection and changing some advanced options.

Regression Model Type	Interpretability	Model Flexibility
<b>Linear SVM</b> 	Easy	Low
<b>Quadratic SVM</b> 	Hard	Medium
<b>Cubic SVM</b> 	Hard	Medium
<b>Fine Gaussian SVM</b> 	Hard	High  Allows rapid variations in the response function. Kernel scale is set to $\sqrt{P} / 4$ , where $P$ is the number of predictors.
<b>Medium Gaussian SVM</b> 	Hard	Medium  Gives a less flexible response function. Kernel scale is set to $\sqrt{P}$ .
<b>Coarse Gaussian SVM</b> 	Hard	Low  Gives a rigid response function. Kernel scale is set to $\sqrt{P} * 4$ .

Statistics and Machine Learning Toolbox implements linear epsilon-insensitive SVM regression. This SVM ignores prediction errors that are less than some fixed number  $\epsilon$ . The support vectors are the data points that have errors larger than  $\epsilon$ . The function the SVM uses to predict new values depends only on the support vectors. To learn more about SVM regression, see Understanding Support Vector Machine Regression on page 25-2.

**Tip** For a workflow example, see “Train Regression Trees Using Regression Learner App” on page 24-48.

## Advanced SVM Options

Regression Learner uses the `fitrsvm` function to train SVM regression models.

You can set these options in the app:

- **Kernel function**

The kernel function determines the nonlinear transformation applied to the data before the SVM is trained. You can choose from:

- Gaussian or Radial Basis Function (RBF) kernel
- Linear kernel, easiest to interpret
- Quadratic kernel
- Cubic kernel

- **Box constraint mode**

The box constraint controls the penalty imposed on observations with large residuals. A larger box constraint gives a more flexible model. A smaller value gives a more rigid model, less sensitive to overfitting.

When **Box constraint mode** is set to `Auto`, the app uses a heuristic procedure to select the box constraint.

Try to fine-tune your model by specifying the box constraint manually. Set **Box constraint mode** to `Manual` and specify a value. Change the value by clicking the buttons or entering a positive scalar value in the **Manual box constraint** box. The app automatically preselects a reasonable value for you. Try to increase or decrease this value slightly and see if this improves your model.

---

**Tip** Increase the box constraint value to create a more flexible model.

---

- **Epsilon mode**

Prediction errors that are smaller than the epsilon ( $\epsilon$ ) value are ignored and treated as equal to zero. A smaller epsilon value gives a more flexible model.

When **Epsilon mode** is set to `Auto`, the app uses a heuristic procedure to select the kernel scale.

Try to fine-tune your model by specifying the epsilon value manually. Set **Epsilon mode** to `Manual` and specify a value. Change the value by clicking the buttons or entering a positive scalar value in the **Manual epsilon** box. The app automatically preselects a reasonable value for you. Try to increase or decrease this value slightly and see if this improves your model.

---

**Tip** Decrease the epsilon value to create a more flexible model.

---

- **Kernel scale mode**

The kernel scale controls the scale of the predictors on which the kernel varies significantly. A smaller kernel scale gives a more flexible model.

When **Kernel scale mode** is set to `Auto`, the app uses a heuristic procedure to select the kernel scale.

Try to fine-tune your model by specifying the kernel scale manually. Set **Kernel scale mode** to `Manual` and specify a value. Change the value by clicking the buttons or entering a positive scalar value in the **Manual kernel scale** box. The app automatically preselects a reasonable value for you. Try to increase or decrease this value slightly and see if this improves your model.

---

**Tip** Decrease the kernel scale value to create a more flexible model.

---


- **Standardize data**

Standardizing the predictors transforms them so that they have mean 0 and standard deviation 1. Standardizing removes the dependence on arbitrary scales in the predictors and generally improves performance.





## Gaussian Process Regression Models

You can train Gaussian process regression (GPR) models in Regression Learner. GPR models are often highly accurate, but can be difficult to interpret.

---

**Tip** In the **Model Type** gallery, click **All GPR Models**  to try each of the GPR model options and see which settings produce the best model with your data. Select the best model in the History list, and try to improve that model by using feature selection and changing some advanced options.

---

Regression Model Type	Interpretability	Model Flexibility
<b>Rational Quadratic</b> 	Hard	Automatic
<b>Squared Exponential</b> 	Hard	Automatic
<b>Matern 5/2</b> 	Hard	Automatic
<b>Exponential</b> 	Hard	Automatic

In Gaussian process regression, the response is modeled using a probability distribution over a space of functions. The flexibility of the presets in the **Model Type** gallery is automatically chosen to give a small training error and, simultaneously, protection against overfitting. To learn more about Gaussian process regression, see Gaussian Process Regression Models on page 6-2.

---

**Tip** For a workflow example, see “Train Regression Trees Using Regression Learner App” on page 24-48.

---

### Advanced Gaussian Process Regression Options

Regression Learner uses the `fitrgp` function to train GPR models.

You can set these options in the app:

- **Basis function**

The basis function specifies the form of the prior mean function of the Gaussian process regression model. You can choose from `Zero`, `Constant`, and `Linear`. Try to choose a different basis function and see if this improves your model.

- **Kernel function**

The kernel function determines the correlation in the response as a function of the distance between the predictor values. You can choose from Rational Quadratic, Squared Exponential, Matern 5/2, Matern 3/2, and Exponential.

To learn more about kernel functions, see Kernel (Covariance) Function Options on page 6-6.

- **Use isotropic kernel**

If you use an isotropic kernel, the correlation length scales are the same for all the predictors. With a nonisotropic kernel, each predictor variable has its own separate correlation length scale.

Using a nonisotropic kernel can improve the accuracy of your model, but can make the model slow to fit.

To learn more about nonisotropic kernels, see Kernel (Covariance) Function Options. on page 6-6

- **Kernel mode**

You can manually specify *initial* values of the kernel parameters **Kernel scale** and **Signal standard deviation**. The signal standard deviation is the prior standard deviation of the response values. By default the app locally optimizes the kernel parameters starting from the initial values. To use fixed kernel parameters, clear the **Optimize numeric parameters** check box in the advanced options.

When **Kernel scale mode** is set to Auto, the app uses a heuristic procedure to select the initial kernel parameters.

If you set **Kernel scale mode** to Manual, you can specify the initial values. Click the buttons or enter a positive scalar value in the **Kernel scale** box and the **Signal standard deviation** box.

If you clear the **Use isotropic kernel** check box, you cannot set initial kernel parameters manually.

- **Sigma mode**

You can specify manually the *initial* value of the observation noise standard deviation **Sigma**. By default the app optimizes the observation noise standard deviation,

starting from the initial value. To use fixed kernel parameters, clear the **Optimize numeric parameters** check box in the advanced options.

When **Sigma mode** is set to `Auto`, the app uses a heuristic procedure to select the initial observation noise standard deviation.

If you set **Sigma mode** to `Manual`, you can specify the initial values. Click the buttons or enter a positive scalar value in the **Sigma** box.

- **Standardize data**

Standardizing the predictors transforms them so that they have mean 0 and standard deviation 1. Standardizing removes the dependence on arbitrary scales in the predictors and generally improves performance.


- **Optimize numeric parameters**

With this option, the app automatically optimizes numeric parameters of the GPR model. The optimized parameters are the coefficients of the **Basis function**, the kernel parameters **Kernel scale** and **Signal standard deviation**, and the observation noise standard deviation **Sigma**.



## Ensembles of Trees

You can train ensembles of regression trees in Regression Learner. Ensemble models combine results from many weak learners into one high-quality ensemble model.

---

**Tip** In the **Model Type** gallery, click **All Ensembles**  to try each of the ensemble options and see which settings produce the best model with your data. Select the best model in the History list, and try to improve that model by using feature selection and changing some advanced options.

---

Regression Model Type	Interpretability	Ensemble Method	Model Flexibility
<b>Boosted Trees</b> 	Hard	Least-squares boosting (LSBoost) with regression tree learners.	Medium to high
<b>Bagged Trees</b> 	Hard	Bootstrap aggregating or bagging, with regression tree learners.	High

---

**Tip** For a workflow example, see “Train Regression Trees Using Regression Learner App” on page 24-48.

---

### Advanced Ensemble Options

Regression Learner uses the `fitensemble` function to train ensemble models. You can set these options:

- **Minimum leaf size**

Specify the minimum number of training samples used to calculate the response of each leaf node. When you grow a regression tree, consider its simplicity and predictive power. To change the minimum leaf size, click the buttons or enter a positive integer value in the **Minimum leaf size** box.

- A fine tree with many small leaves is usually highly accurate on the training data. However, the tree might not show comparable accuracy on an independent test set. A very leafy tree tends to overfit, and its validation accuracy is often far lower than its training (or resubstitution) accuracy.
- In contrast, a coarse tree with fewer large leaves does not attain high training accuracy. But a coarse tree can be more robust in that its training accuracy can be near that of a representative test set.

---

**Tip** Decrease the **Minimum leaf size** to create a more flexible model.

---

- **Number of learners**



Try changing the number of learners to see if you can improve the model. Many learners can produce high accuracy, but can be time consuming to fit.

---

**Tip** Increase the **Number of learners** to create a more flexible model.

---

- **Learning rate**

For boosted trees, specify the learning rate for shrinkage. If you set the learning rate to less than 1, the ensemble requires more learning iterations but often achieves better accuracy. 0.1 is a popular initial choice.

## See Also

### Related Examples

- “Train Regression Models in Regression Learner App” on page 24-2
- “Select Data and Validation for Regression Problem” on page 24-10
- “Feature Selection and Feature Transformation Using Regression Learner App” on page 24-32
- “Assess Model Performance in Regression Learner App” on page 24-36
- “Export Regression Model to Predict New Data” on page 24-44
- “Train Regression Trees Using Regression Learner App” on page 24-48

## Feature Selection and Feature Transformation Using Regression Learner App

### In this section...

“Investigate Features in the Response Plot” on page 24-32

“Select Features to Include” on page 24-33

“Transform Features with PCA in Regression Learner” on page 24-34

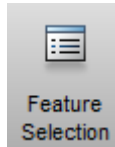
### Investigate Features in the Response Plot

In Regression Learner, use the response plot to try to identify predictors that are useful for predicting the response. To visualize the relation between different predictors and the response, under **X-axis**, select different variables in the **X** list.

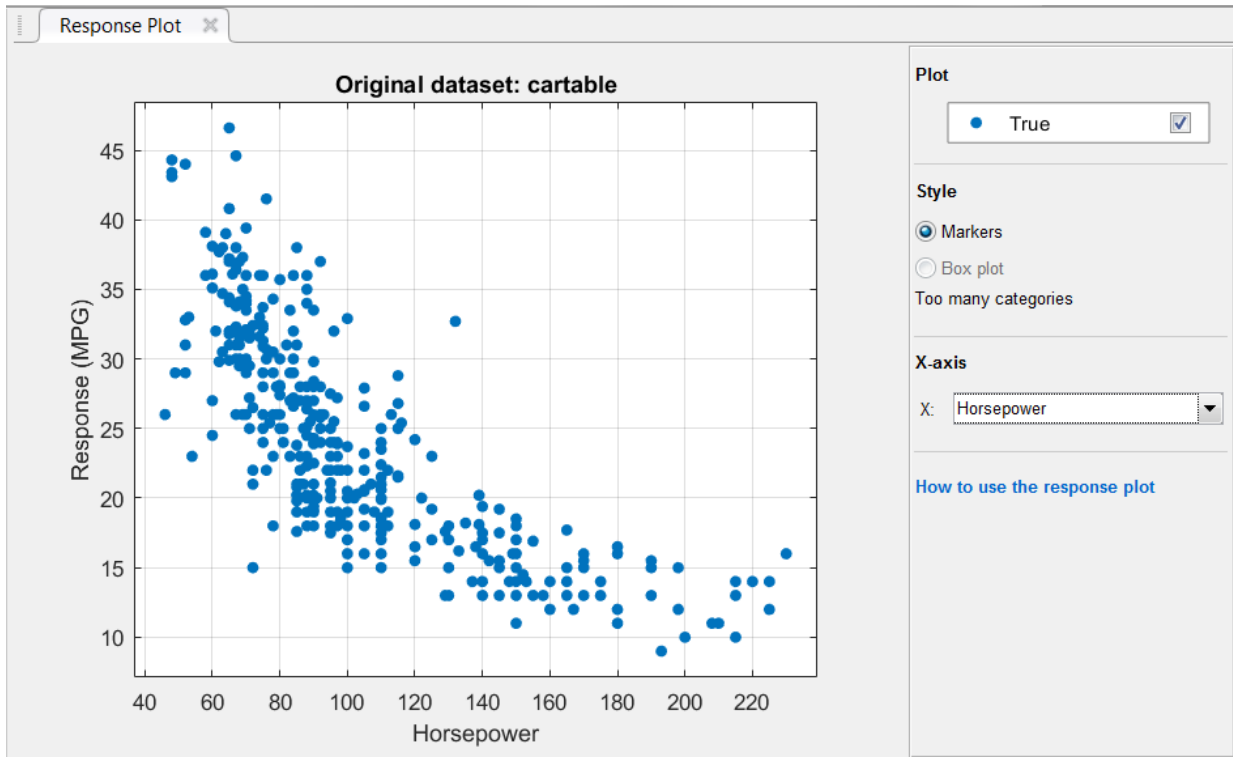
Before you train a regression model, the response plot shows the training data. If you have trained a regression model, then the response plot also shows the model predictions.

Observe which variables are associated most clearly with the response. When you plot the `carbig` data set, the predictor `Horsepower` shows a clear negative association with the response.

Look for features that do not seem to have any association with the response and use



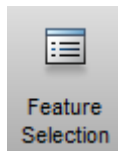
**Feature Selection** to remove those features from the set of used predictors.



## Select Features to Include

In Regression Learner, you can specify different features (or predictors) to include in the model. See if you can improve models by removing features with low predictive power. If data collection is expensive or difficult, you might prefer a model that performs satisfactorily with fewer predictors.

- 1 On the **Regression Learner** tab, in the **Features** section, click **Feature Selection**



- 2 In the Feature Selection window, clear the check boxes for the predictors you want to exclude.

---

**Tip** You can close the Feature Selection window, or move it. The choices you make in the window remain.

---

- 3 Click **Train** to train a new model using the new predictor options.
- 4 Observe the new model in the History list. The Current Model window displays how many predictors are excluded.
- 5 Check which predictors are included in a trained model. Click the model in the History list and look at the check boxes in the Feature Selection window.
- 6 Try to improve the model by including different features.

For an example using feature selection, see “Train Regression Trees Using Regression Learner App” on page 24-48.

## Transform Features with PCA in Regression Learner

Use principal component analysis (PCA) to reduce the dimensionality of the predictor space. Reducing the dimensionality can create regression models in Regression Learner that help prevent overfitting. PCA linearly transforms predictors to remove redundant dimensions, and generates a new set of variables called principal components.

- 1 On the **Regression Learner** tab, in the **Features** section, select **PCA**.
- 2 In the Advanced PCA Options window, select the **Enable PCA** check box.

You can close the PCA window, or move it. The choices you make in the window remain.

- 3 Click **Train** again. The `pca` function transforms your selected features before training the model.

By default, PCA keeps only the components that explain 95% of the variance. In the PCA window, you can change the percentage of variance to explain in the **Explained variance** box. A higher value risks overfitting, while a lower value risks removing useful dimensions.

- 4 Manually limit the number of PCA components. In the **Component reduction criterion** list, select `Specify number of components`. Edit the number in the **Number of numeric components** box. The number of components cannot be larger than the number of numeric predictors. PCA is not applied to categorical predictors.

You can check PCA Options for trained models in the Current Model window. For example:

PCA is keeping enough components to explain 95% variance.  
After training, 2 components were kept.  
Explained variance per component (in order): 92.5%, 5.3%, 1.7%, 0.5%

Check the explained variance percentages to decide whether to change the number of components.

To learn more about how Regression Learner applies PCA to your data, generate code for your trained regression model. For more information on PCA, see the `pca` function.

## See Also

### Related Examples

- “Train Regression Models in Regression Learner App” on page 24-2
- “Select Data and Validation for Regression Problem” on page 24-10
- “Choose Regression Model Options” on page 24-15
- “Assess Model Performance in Regression Learner App” on page 24-36
- “Export Regression Model to Predict New Data” on page 24-44
- “Train Regression Trees Using Regression Learner App” on page 24-48

## Assess Model Performance in Regression Learner App

### In this section...

“Check Performance in History List” on page 24-36

“View Model Statistics in Current Model Window” on page 24-37

“Explore Data and Results in Response Plot” on page 24-38

“Plot Predicted vs. Actual Response” on page 24-40

“Evaluate Model Using Residuals Plot” on page 24-41

### Check Performance in History List

After training a model in Regression Learner, check the History list to see which model has the best overall score. The best score is highlighted in a box. This score is the root mean square error (RMSE) on the validation set. The score estimates the performance of the trained model on new data. Use the score to help you choose the best model.

▼ History	
1.1 ☆ Linear Regression Last change: Linear	RMSE: 3.3815 7/7 features
1.2 ☆ Tree Last change: Fine Tree	RMSE: 3.3849 7/7 features
1.3 ☆ Tree Last change: Medium Tree	RMSE: <b>3.3102</b> 7/7 features
1.4 ☆ Tree Last change: Coarse Tree	RMSE: 3.8819 7/7 features

- For cross-validation, the score is the RMSE on all observations, counting each observation when it was in a held-out fold.

**Note** When you imported data into the app, if you accepted the defaults, you are using cross-validation. To learn more, see “Choose Validation Scheme” on page 24-12.

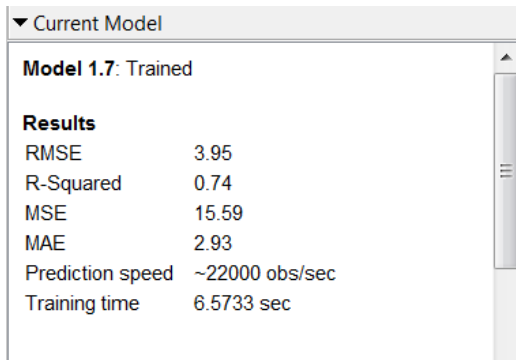
- For holdout validation, the score is the RMSE on the held-out observations.
- For no validation, the score is the resubstitution RMSE on all the training data.

The best overall score might not be the best model for your goal. Sometimes a model with slightly lower overall score is the better model for your goal. You want to avoid

overfitting, and you might want to exclude some predictors where data collection is expensive or difficult.

## View Model Statistics in Current Model Window

You can view model statistics in the Current Model window and use these statistics to assess and compare models. The statistics are calculated on the validation set.



### Model Statistics

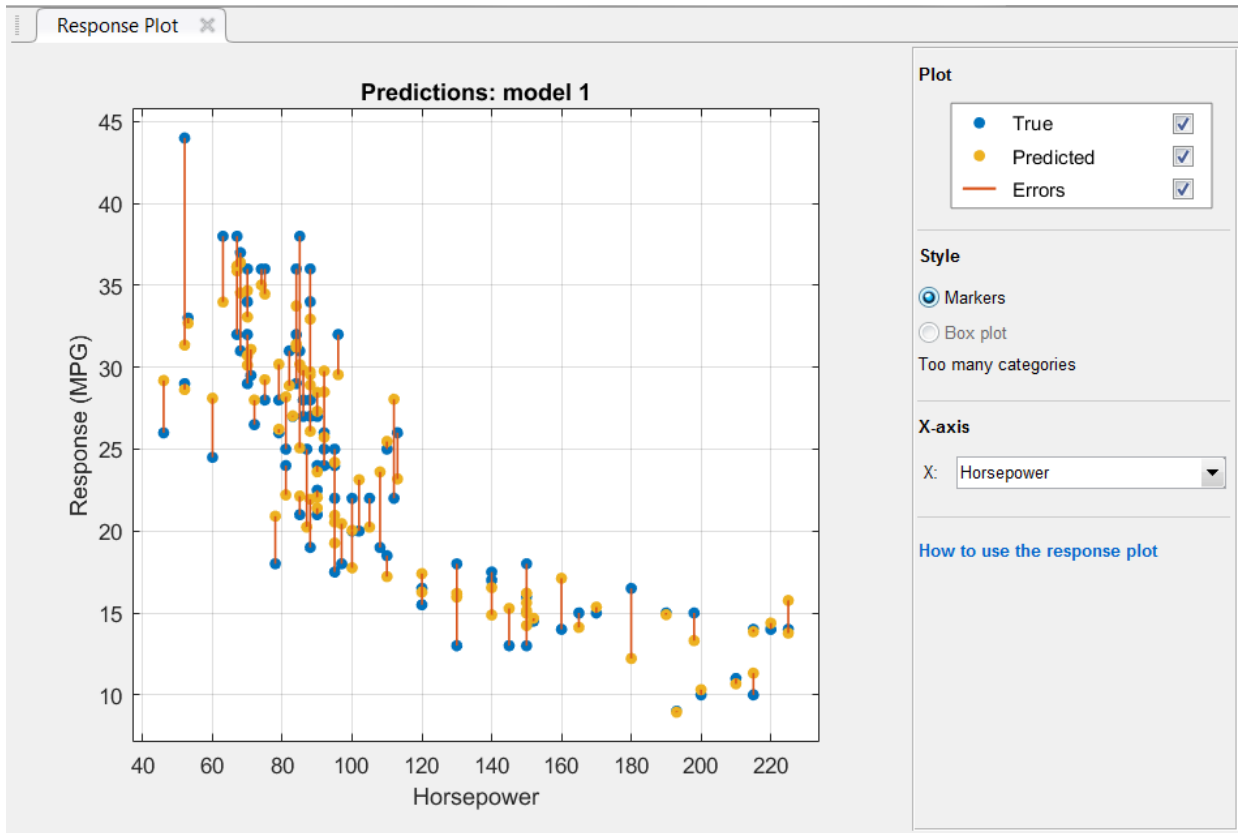
Statistic	Description	Tip
RMSE	Root mean square error. The RMSE is always positive and its units match the units of your response.	Look for smaller values of the RMSE.
R-Squared	Coefficient of determination. R-squared is always smaller than 1 and usually larger than 0. It compares the trained model with the model where the response is constant and equals the mean of the training response. If your model is worse than this constant model, then R-Squared is negative.	Look for an R-Squared close to 1.
MSE	Mean squared error. The MSE is the square of the RMSE.	Look for smaller values of the MSE.
MAE	Mean absolute error. The MAE is always positive and similar to the RMSE, but less sensitive to outliers.	Look for smaller values of the MAE.

## Explore Data and Results in Response Plot

In the response plot, view the regression model results. After you train a regression model, the response plot displays the predicted response versus record number. If you are using holdout or cross-validation, then these predictions are the predictions on the held-out observations. In other words, each prediction is obtained using a model that was trained without using the corresponding observation. To investigate your results, use the controls on the right. You can:

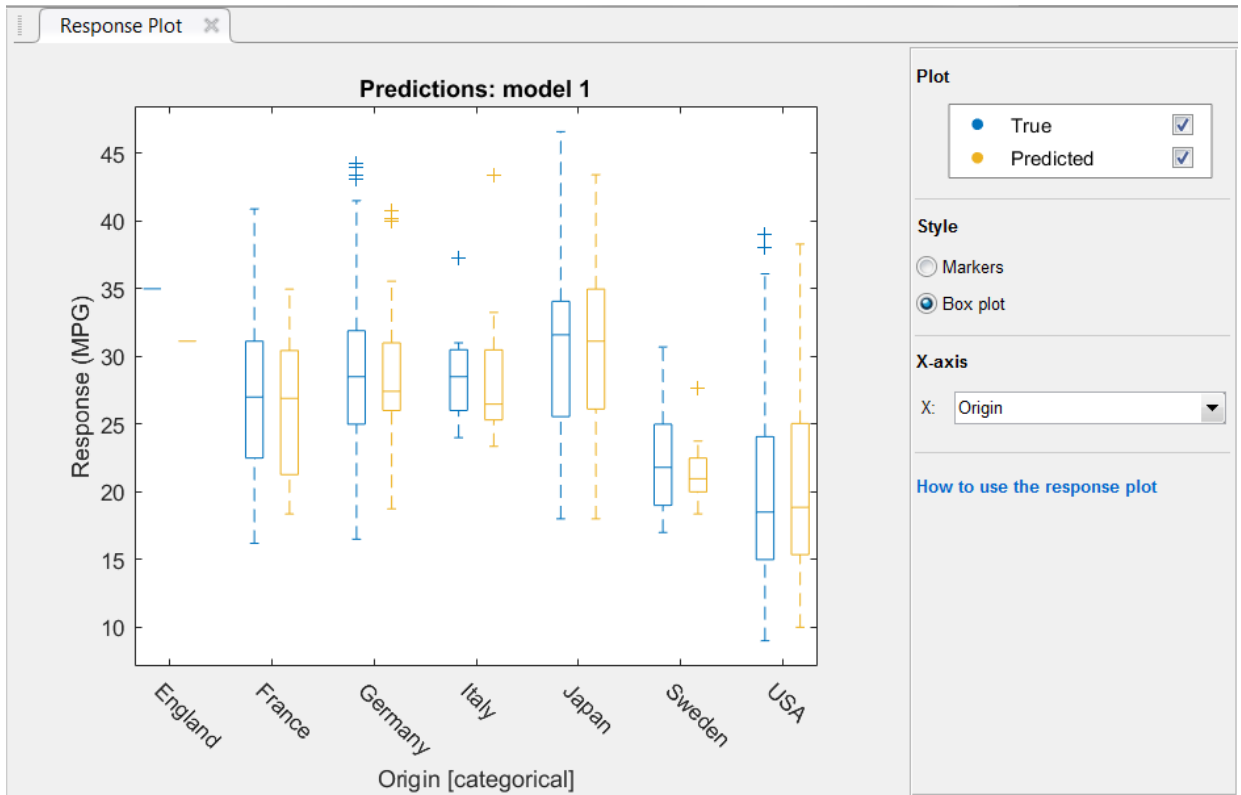
- Plot predicted and/or true responses. Use the check boxes under **Plot** to make your selection.
- Show prediction errors, drawn as vertical lines between the predicted and true responses, by selecting the **Errors** check box.
- Choose the variable to plot on the  $x$ -axis under **X-axis**. You can choose either the record number or one of your predictor variables.






- Plot the response as markers, or as a box plot under **Style**. You can only select **Box plot** when the variable on the x-axis has few unique values.

A box plot displays the typical values of the response and any possible outliers. The central mark indicates the median, and the bottom and top edges of the box are the 25th and 75th percentiles, respectively. Vertical lines, called whiskers, extend from the boxes to the most extreme data points that are not considered outliers. The outliers are plotted individually using the '+' symbol. For more information about box plots, see `boxplot`.

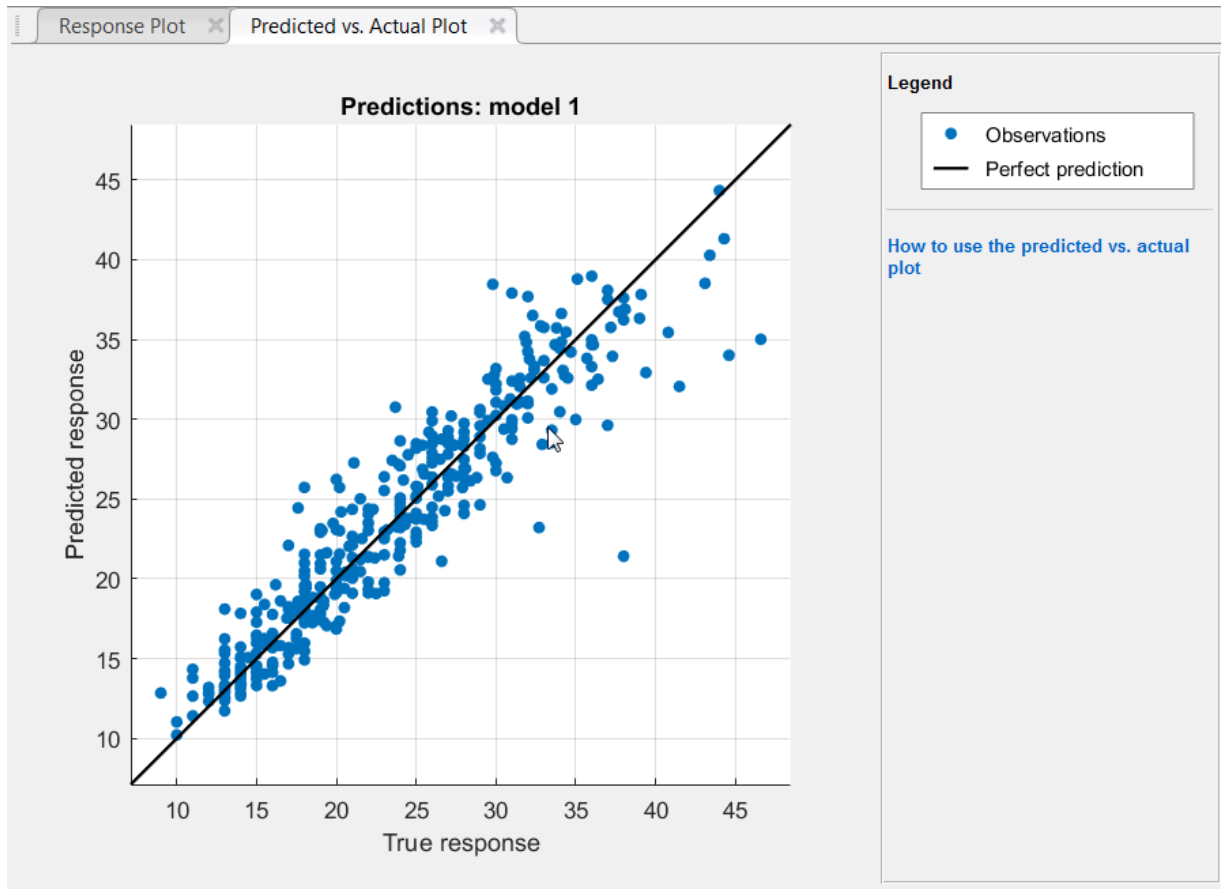


## Plot Predicted vs. Actual Response

Use the Predicted vs. Actual plot to check model performance. Use this plot to understand how well the regression model makes predictions for different response values. To view the Predicted vs. Actual plot after training a model, on the **Regression**

**Learner** tab, in the **Plots** section, click **Predicted vs. Actual Plot** .


When you open the plot, the predicted response of your model is plotted against the actual, true response. A perfect regression model has a predicted response equal to the true response, so all the points lie on a diagonal line. The vertical distance from the line to any point is the error of the prediction for that point. A good model has small errors, and so the predictions are scattered near the line.

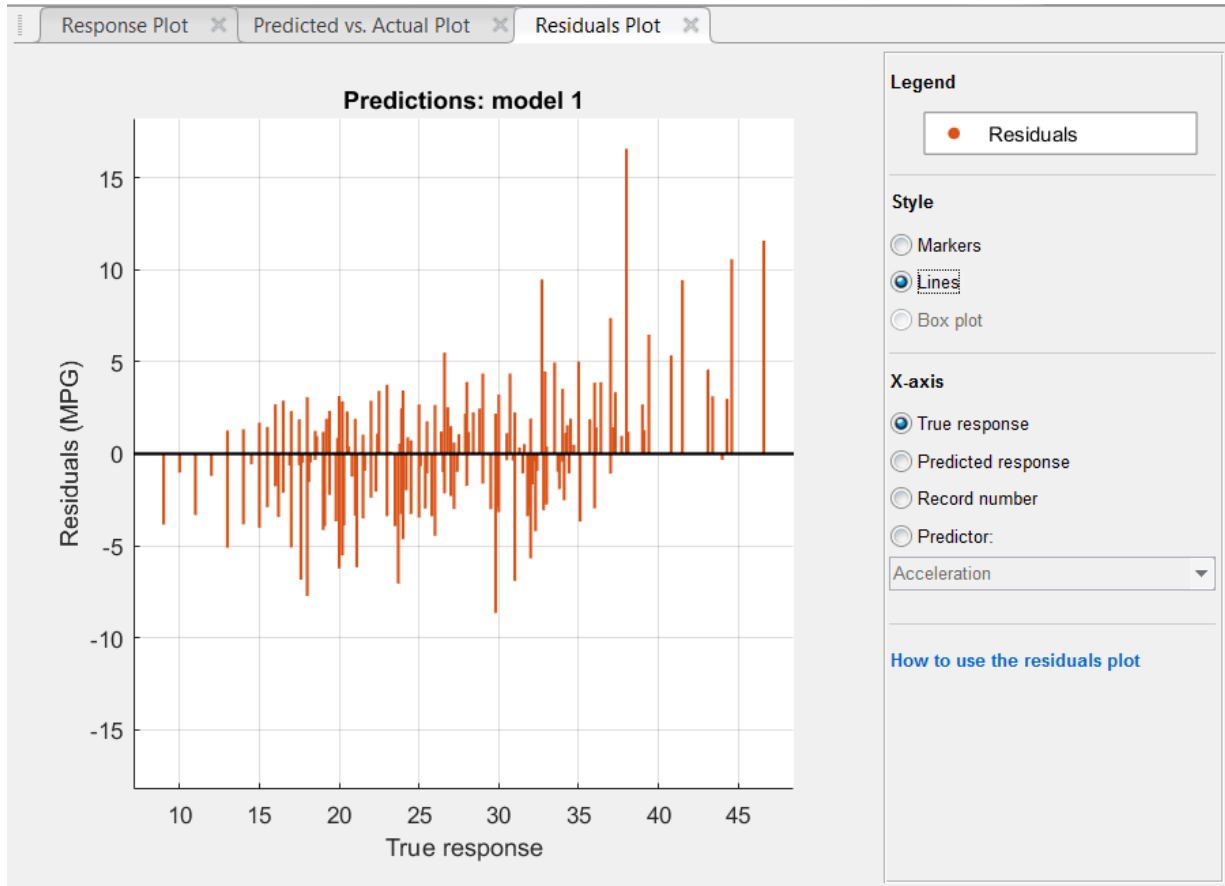


Usually a good model has points scattered roughly symmetrically around the diagonal line. If you can see any clear patterns in the plot, it is likely that you can improve your model. Try training a different model type or making your current model type more flexible using the **Advanced** options in the **Model Type** section. If you are unable to improve your model, it is possible that you need more data, or that you are missing an important predictor.

## Evaluate Model Using Residuals Plot

Use the residuals plot to check model performance. To view the residuals plot after training a model, on the **Regression Learner** tab, in the **Plots** section, click **Residuals**

**Plot** . The residuals plot displays the difference between the predicted and true responses. Choose the variable to plot on the  $x$ -axis under **X-axis**. Choose either the true response, predicted response, record number, or one of your predictors.



Usually a good model has residuals scattered roughly symmetrically around 0. If you can see any clear patterns in the residuals, it is likely that you can improve your model. Look for these patterns:

- Residuals are not symmetrically distributed around 0.
- Residuals change significantly in size from left to right in the plot.
- Outliers occur, that is, residuals that are much larger than the rest of the residuals.

- Clear, nonlinear pattern appears in the residuals.

Try training a different model type, or making your current model type more flexible using the **Advanced** options in the **Model Type** section. If you are unable to improve your model, it is possible that you need more data, or that you are missing an important predictor.

## See Also

### Related Examples

- “Train Regression Models in Regression Learner App” on page 24-2
- “Select Data and Validation for Regression Problem” on page 24-10
- “Choose Regression Model Options” on page 24-15
- “Feature Selection and Feature Transformation Using Regression Learner App” on page 24-32
- “Export Regression Model to Predict New Data” on page 24-44
- “Train Regression Trees Using Regression Learner App” on page 24-48

## Export Regression Model to Predict New Data

In this section...
“Export Model to Workspace” on page 24-44
“Make Predictions for New Data” on page 24-45
“Generate MATLAB Code to Train Model with New Data” on page 24-45

### Export Model to Workspace

After you create regression models interactively in the Regression Learner app, you can export your best model to the workspace. Then you can use that trained model to make predictions using new data.

- 1 In the app, select the model you want to export in the History list.
- 2 On the **Regression Learner** tab, in the **Export** section, click one of the export options:
  - To include the data used for training the model, select **Export Model**.  
You export the trained model to the workspace as a structure containing a regression model object.
  - To exclude the training data, select **Export Compact Model**. This option exports the model with unnecessary data removed where possible. For some models this is a compact object that does not include the training data, but you can still use it for making predictions on new data.
- 3 In the Export Model dialog box, check the name of your exported variable, and edit it if you want. Then, click **OK**. The default name for your exported model, `trainedModel`, increments every time you export to avoid overwriting your models; for example, `trainedModel1`.

The new variable (for example, `trainedModel`) appears in your workspace.

The app displays information about the exported model in the command window. Read the message to learn how to make predictions with new data.

---

**Note** The final exported model is always trained using the full data set. The validation scheme that you use only affects the way that Regression Learner computes validation metrics.

---

## Make Predictions for New Data

After you export a model to the workspace from Regression Learner, or run the code generated from the app, you get a `trainedModel` structure that you can use to make predictions using new data. The structure contains a model object and a function for prediction. The structure enables you to make predictions for models that include principal component analysis (PCA).

- 1 Use the exported model to make predictions for new data, `T`:

```
yfit = trainedModel.predictFcn(T)
```

where `trainedModel` is the name of your exported variable.

Supply the data `T` in same data type as your training data used in the app (table or matrix).

- If you supply a table, then ensure that it contains the same predictor names as your training data. The `predictFcn` ignores additional variables in tables. Variable formats and types must match the original training data.
- If you supply a matrix, it must contain the same predictor columns or rows as your training data, in the same order and format. Do not include a response variable, any variables that you did not import in the app, or other unused variables.

The output `yfit` contains a prediction for each data point.

- 2 Examine the fields of the exported structure. For help making predictions, enter:

```
trainedModel.HowToPredict
```

You also can extract the model object from the exported structure for further analysis. If you use feature transformation such as PCA in the app, you must take into account this transformation by using the information in the PCA fields of the structure.

## Generate MATLAB Code to Train Model with New Data

After you create regression models interactively in the Regression Learner app, you can generate MATLAB code for your best model. Then you can use the code to train the model with new data.

Generate MATLAB code to:

- Train on huge data sets. Explore models in the app trained on a subset of your data, and then generate code to train a selected model on a larger data set.
- Create scripts for training models without needing to learn syntax of the different functions.
- Examine the code to learn how to train models programmatically.
- Modify the code for further analysis, for example to set options that you cannot change in the app.
- Repeat your analysis on different data and automate training.

To generate code and use it to train a model with new data:

- 1 In the app, from the History list, select the model you want to generate code for.
- 2 On the **Regression Learner** tab, in the **Export** section, click **Export Model > Generate Code**.

The app generates code from your session and displays the file in the MATLAB Editor. The file includes the predictors and response, the model training methods, and the validation methods. Save the file.

- 3 To retrain your model, call the function from the command line with your original data or new data as the input argument. New data must have the same shape.

Copy the first line of the generated code, excluding the word `function`, and edit the `trainingData` input argument to reflect the variable name of your training data or new data.

The generated code returns a `trainedModel` structure that contains the same fields as the structure you create when you export a model from Regression Learner to the workspace.

If you want to automate training the same model with new data, or learn how to programmatically train models, examine the generated code. The code shows you how to:

- Process the data into the right shape.
- Train a model and specify all the model options.
- Perform cross-validation.
- Compute statistics.
- Compute validation predictions and scores.



## See Also

### Functions

`fitlm` | `fitrensemble` | `fitrgp` | `fitrsvm` | `fitrtree` | `stepwiselm`

### Classes

`CompactLinearModel` | `CompactRegressionEnsemble` | `CompactRegressionGP` | `CompactRegressionSVM` | `CompactRegressionTree` | `LinearModel` | `RegressionEnsemble` | `RegressionGP` | `RegressionSVM` | `RegressionTree`

### Related Examples

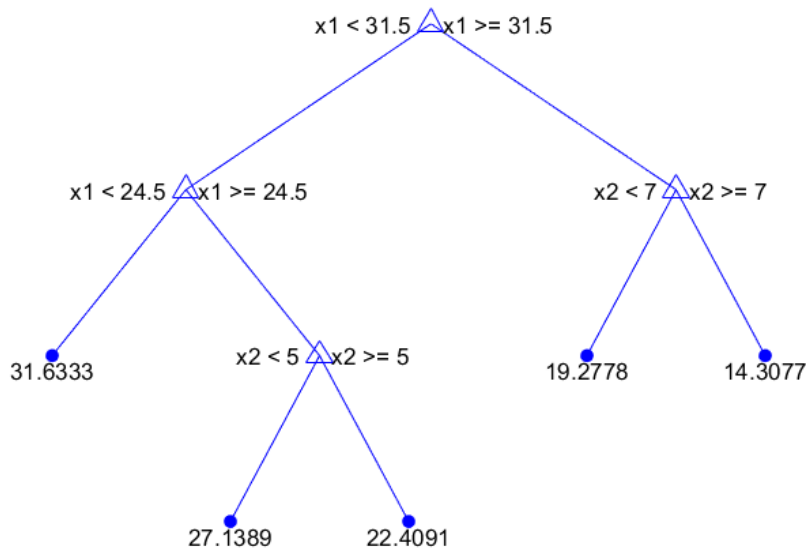
- “Train Regression Models in Regression Learner App” on page 24-2
- “Select Data and Validation for Regression Problem” on page 24-10
- “Choose Regression Model Options” on page 24-15
- “Feature Selection and Feature Transformation Using Regression Learner App” on page 24-32
- “Assess Model Performance in Regression Learner App” on page 24-36
- “Train Regression Trees Using Regression Learner App” on page 24-48

## Train Regression Trees Using Regression Learner App

This example shows how to create and compare various regression trees using the Regression Learner app, and export trained models to the workspace to make predictions for new data.

You can train regression trees to predict responses to given input data. To predict the response of a regression tree, follow the tree from the root (beginning) node down to a leaf node. At each node, decide which branch to follow using the rule associated to that node. Continue until you arrive at a leaf node. The predicted response is the value associated to that leaf node.

Statistics and Machine Learning Toolbox trees are binary. Each step in a prediction involves checking the value of one predictor variable. For example, here is a simple regression tree:



This tree predicts the response based on two predictors,  $x_1$  and  $x_2$ . To predict, start at the top node. At each node, check the values of the predictors to decide which branch to follow. When the branches reach a leaf node, the response is set to the value corresponding to that node.

This example uses the `carbig` data set. This data set contains characteristics of different car models produced from 1970 through 1982, including:

- Acceleration
- Number of cylinders
- Engine displacement
- Engine power (Horsepower)
- Model year
- Country of origin
- Miles per gallon (MPG)

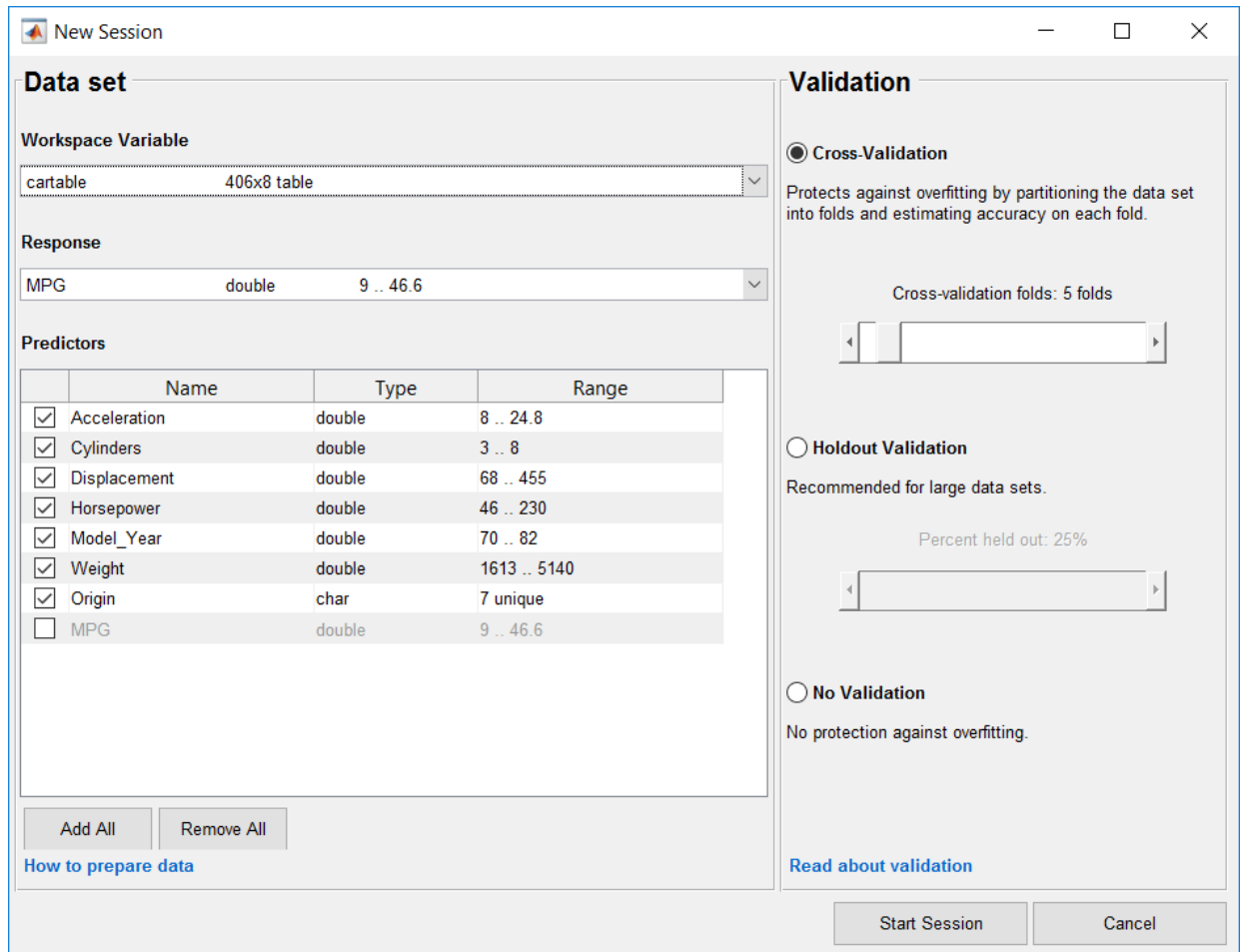
Train regression trees to predict the fuel economy in miles per gallon of a car model, given the other variables as inputs.

- 1 In MATLAB, load the `carbig` data set and create a table containing the different variables:

```
load carbig
cartable = table(Acceleration, Cylinders, Displacement,...
Horsepower, Model_Year, Weight, Origin, MPG);
```

- 2 On the **Apps** tab, in the **Math, Statistics and Optimization** group, click **Regression Learner**.
- 3 On the **Regression Learner** tab, click **New Session** and select **From Workspace**.
- 4 Under **Workspace Variable** in the New Session dialog box, select `cartable` from the list of tables and matrices in your workspace.

Observe that the app has preselected response and predictor variables. `MPG` is chosen as the response, and all the other variables as predictors. For this example, do not change the selections.

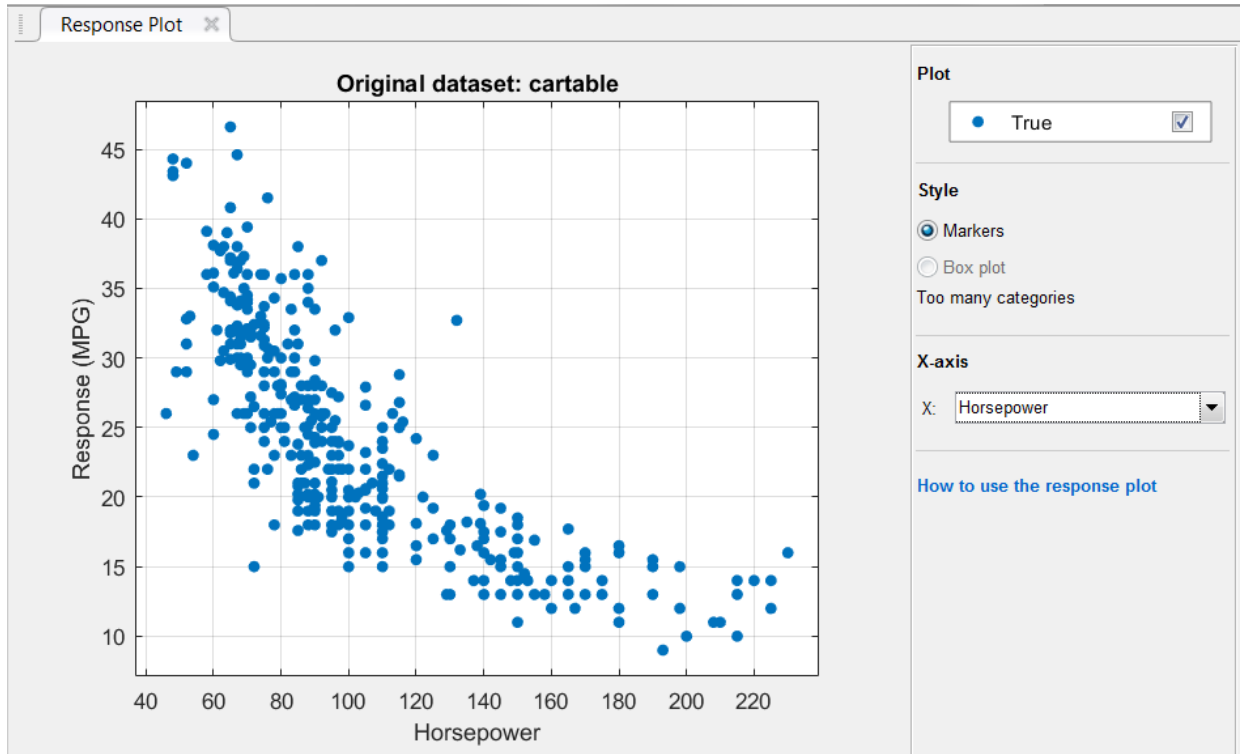


- To accept the default validation scheme and continue, click **Start Session**. The default validation option is cross-validation, to protect against overfitting.

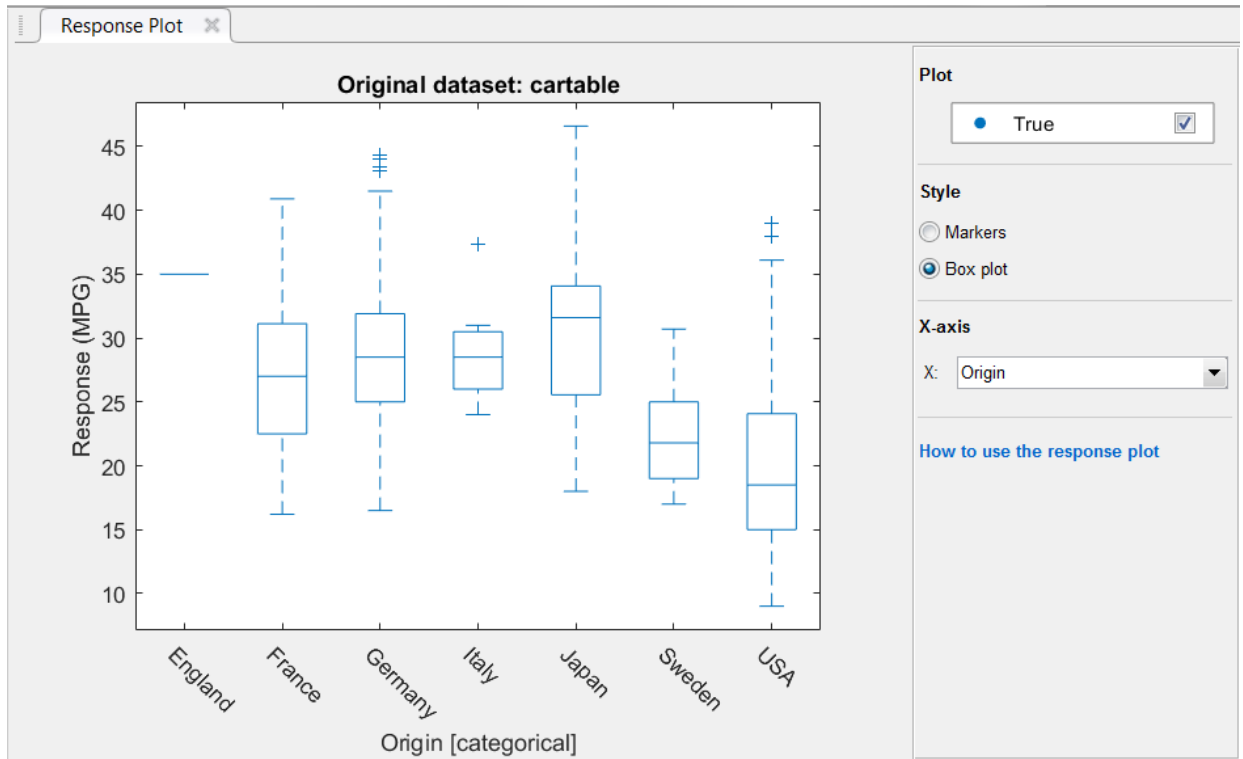
Regression Learner creates a plot of the response with the record number on the  $x$ -axis.


- Use the response plot to investigate which variables are useful for predicting the response. To visualize the relation between different predictors and the response, select different variables in the **X** list under **X-axis**.

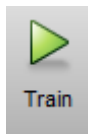
Observe which variables are correlated most clearly with the response. Displacement, Horsepower, and Weight all have a clearly visible impact on the response and all show a negative association with the response.



- 7 Select the variable `Origin` under **X-axis**. A box plot is automatically displayed. A box plot shows the typical values of the response and any possible outliers. The box plot is useful when plotting markers results in many points overlapping. To show a box plot when the variable on the *x*-axis has few unique values, under **Style**, select **Box plot**.



- 8 Create a selection of regression trees. On the **Regression Learner** tab, in the **Model Type** section, click **All Trees** .



Then click **Train**.

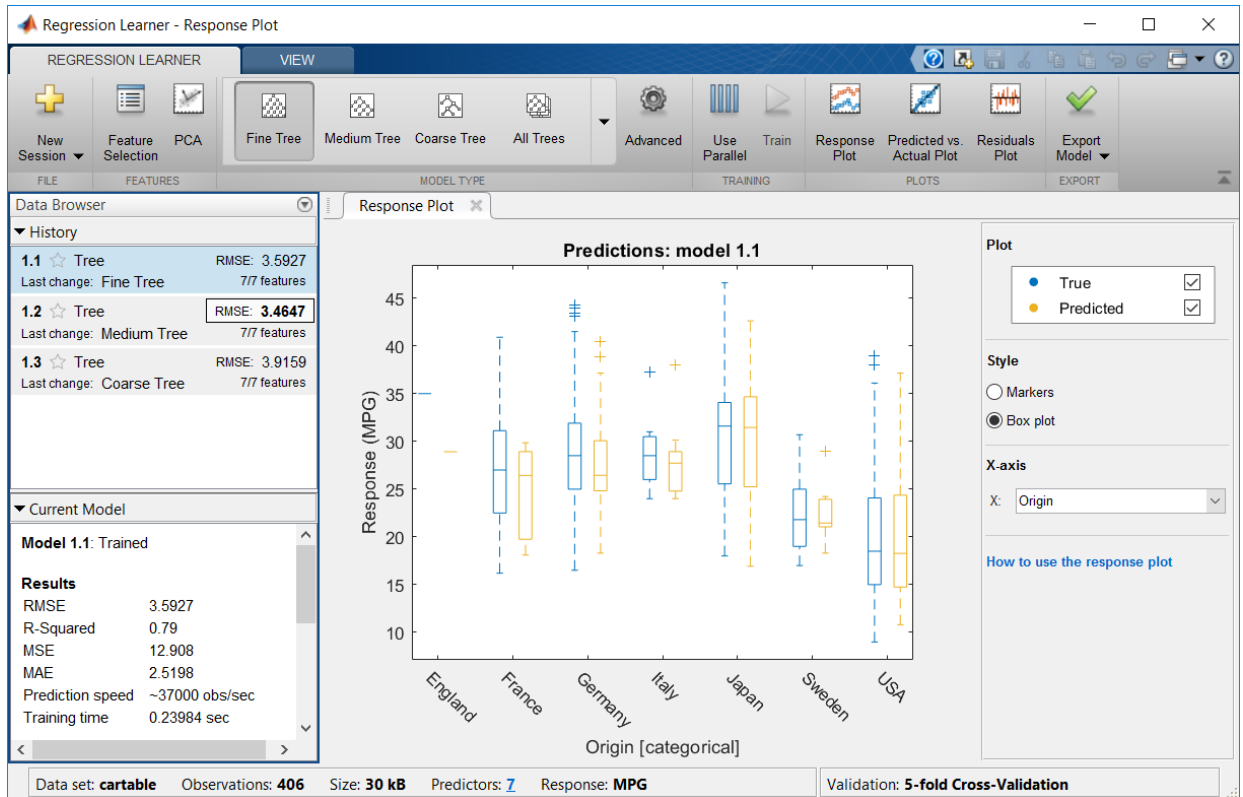
**Tip** If you have Parallel Computing Toolbox, then the first time you click **Train** you see a dialog box while the app opens a parallel pool of workers. After the pool opens, you can train multiple regression models simultaneously and continue working.

Regression Learner creates and trains three regression trees: a **Fine Tree**, a **Medium Tree**, and a **Coarse Tree**.


The three models appear in the History list. Check the validation **RMSE** (root mean square error) of the models. The best score is highlighted in a box.

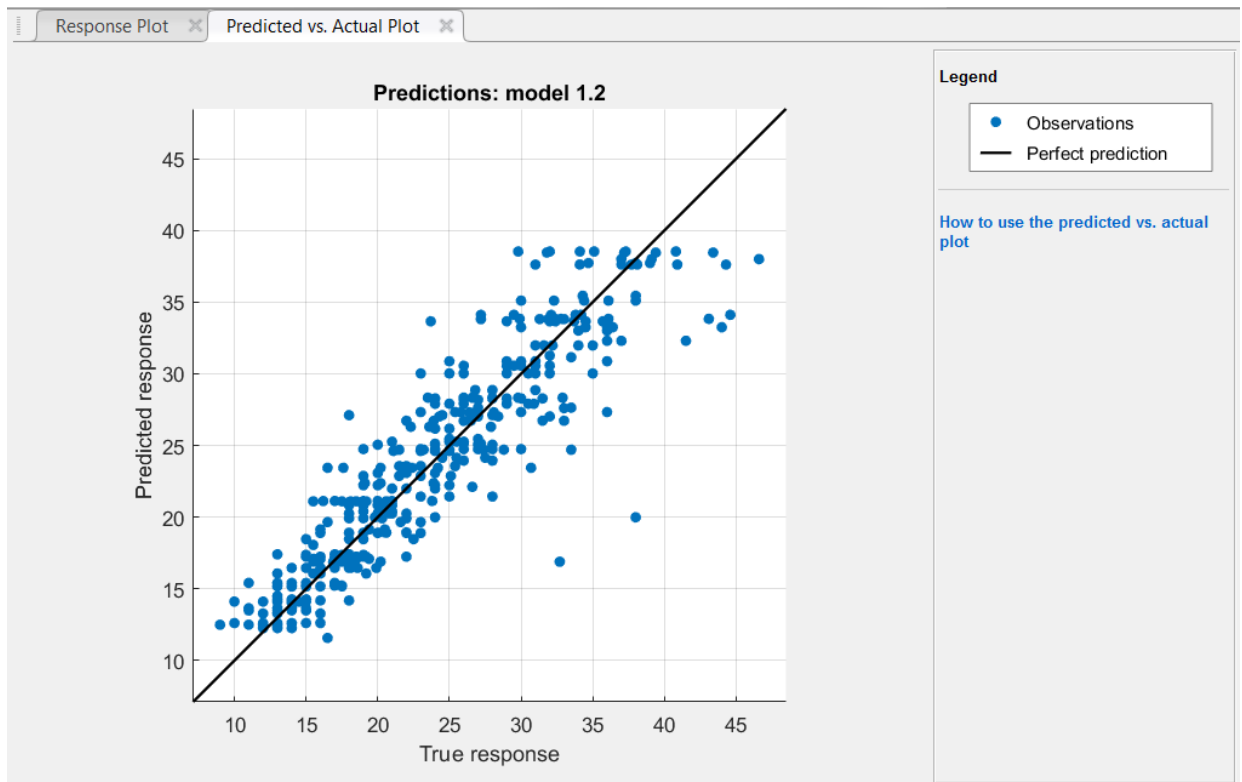
The **Fine Tree** and the **Medium Tree** have similar RMSEs, while the **Coarse Tree** is less accurate.

Regression Learner plots both the true training response and the predicted response of the currently selected model.



**Note** If you are using validation, there is some randomness in the results and so your model validation score can differ from the results shown.

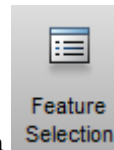
- 9 Choose a model in the History list to view the results of that model. Under **X-axis**, select `Horsepower` and examine the response plot. Both the true and predicted responses are now plotted. Show the prediction errors, drawn as vertical lines between the predicted and true responses, by selecting the **Errors** check box.
- 10 See more details on the currently selected model in the Current Model window. Check and compare additional model characteristics, such as R-squared (coefficient of determination), MAE (mean absolute error), and prediction speed. To learn more, see “View Model Statistics in Current Model Window” on page 24-37. In the Current Model window you also can find details on the currently selected model type, such as options used for training the model.
- 11 Plot the predicted response versus true response. On the **Regression Learner** tab, in the **Plots** section, click **Predicted vs. Actual Plot** . Use this plot to understand how well the regression model makes predictions for different response values.





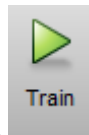
A perfect regression model has predicted response equal to true response, so all the points lie on a diagonal line. The vertical distance from the line to any point is the error of the prediction for that point. A good model has small errors, so the predictions are scattered near the line. Usually a good model has points scattered roughly symmetrically around the diagonal line. If you can see any clear patterns in the plot, it is likely that you can improve your model.

- 12 Select the other models in the History list and compare the predicted versus actual plots.
- 13 In the **Model Type** gallery, select **All Trees** again. To try to improve the model, try including different features in the model. See if you can improve the model by removing features with low predictive power. On the **Regression Learner** tab, in



the **Features** section, click **Feature Selection**.

In the Feature Selection window, clear the check boxes for **Acceleration** and **Cylinders** to exclude them from the predictors.



Click **Train** to train new regression trees using the new predictor settings.

- 14 Observe the new models in the History list. These models are the same regression trees as before, but trained using only five of seven predictors. The History list displays how many predictors are used. To check which predictors are used, click a model in the History list and observe the check boxes in the Feature Selection window.


The models with the two features removed perform comparably to the models using all predictors. The models predict no better using all the predictors compared to using only a subset of the predictors. If data collection is expensive or difficult, you might prefer a model that performs satisfactorily without some predictors.

- 15 Train the three regression tree presets using only `Horsepower` as predictor. Change the selections in the Feature Selection window and click **Train**.

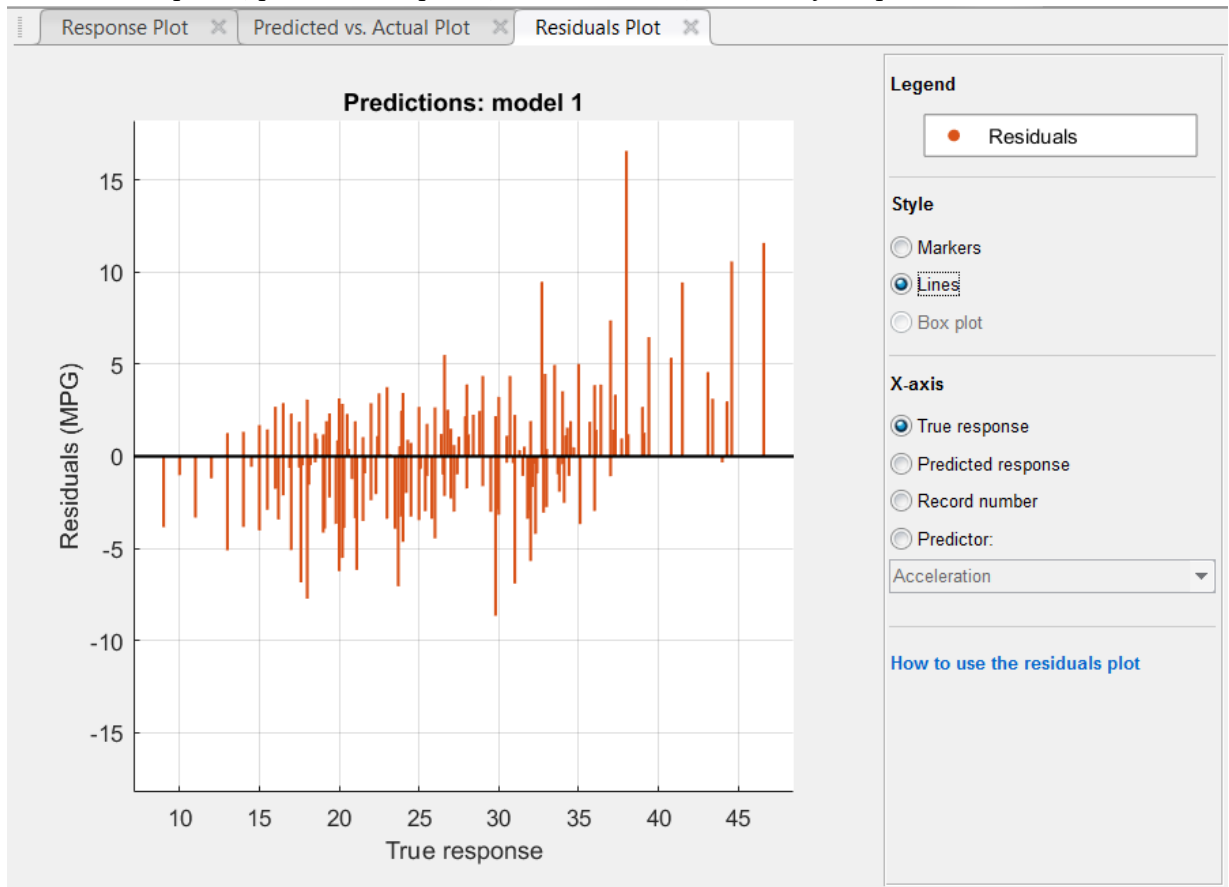
Using only the engine power as predictor results in models with lower accuracy. However, the models perform well given that they are using only a single predictor.

With this simple one-dimensional predictor space, the coarse tree now performs as well as the medium and fine trees.

- 16 Select the best model in the History list and view the residuals plot. On the

**Regression Learner** tab, in the **Plots** section, click **Residuals Plot** . The residuals plot displays the difference between the predicted and true responses. To display the residuals as a line graph, in the **Style** section, choose **Lines**.

Under **X-axis**, select the variable to plot on the  $x$ -axis. Choose either the true response, predicted response, record number, or one of your predictors.



Usually a good model has residuals scattered roughly symmetrically around 0. If you can see any clear patterns in the residuals, it is likely that you can improve your model.

- 17 To learn about model settings, choose the best model in the History list and view the advanced settings. The options in the **Model Type** gallery are preset starting points, and you can change additional settings. On the **Regression Learner** tab, in the **Model Type** section, click **Advanced**. Compare the different regression tree models in the History list, and observe the differences in the Advanced Regression Tree Options dialog box. The **Minimum leaf size** setting controls the size of the tree leaves, and through that the size and depth of the regression tree.

To try to improve the model further, change the **Minimum leaf size** setting to 8, and then train a new model by clicking **Train**.

View the settings for the selected trained model in the Current Model window or in the Advanced Regression Tree Options dialog box.

To learn more about regression tree settings, see “Regression Trees” on page 24-21.

- 18 Export the selected model to the workspace. On the **Regression Learner** tab, in the **Export** section, click **Export Model**. In the Export Model dialog box, click **OK** to accept the default variable name `trainedModel`.

To see information about the results, look in the command window.

- 19 Use the exported model to make predictions on new data. For example, to make predictions for the `cartable` data in your workspace, enter:

```
yfit = trainedModel.predictFcn(cartable)
```

The output `yfit` contains the predicted response for each data point.

- 20 If you want to automate training the same model with new data or learn how to programmatically train regression models, you can generate code from the app. To generate code for the best trained model, on the **Regression Learner** tab, in the **Export** section, click **Export Model > Generate Code**.


The app generates code from your model and displays the file in the MATLAB Editor. To learn more, see “Generate MATLAB Code to Train Model with New Data” on page 24-45.

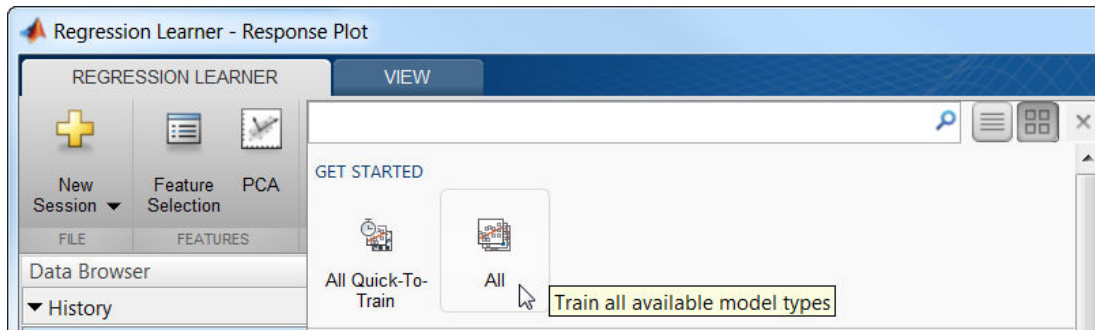
---

**Tip** Use the same workflow as in this example to evaluate and compare the other regression model types you can train in Regression Learner.

---

Train all the regression model presets available:

- 1 On the far right of the **Model Type** section, click the arrow to expand the list of regression models.
- 2 Click **All** , and then click **Train**.



To learn about other regression model types, see “Train Regression Models in Regression Learner App” on page 24-2.

## See Also

### Related Examples

- “Train Regression Models in Regression Learner App” on page 24-2
- “Select Data and Validation for Regression Problem” on page 24-10
- “Choose Regression Model Options” on page 24-15
- “Feature Selection and Feature Transformation Using Regression Learner App” on page 24-32
- “Assess Model Performance in Regression Learner App” on page 24-36
- “Export Regression Model to Predict New Data” on page 24-44

# Support Vector Machines

---

## Understanding Support Vector Machine Regression

### In this section...

“Mathematical Formulation of SVM Regression” on page 25-2

“Solving the SVM Regression Optimization Problem” on page 25-6

### Mathematical Formulation of SVM Regression

#### Overview

Support vector machine (SVM) analysis is a popular machine learning tool for classification and regression, first identified by Vladimir Vapnik and his colleagues in 1992[5]. SVM regression is considered a nonparametric technique because it relies on kernel functions.

Statistics and Machine Learning Toolbox implements linear epsilon-insensitive SVM ( $\varepsilon$ -SVM) regression, which is also known as  $L1$  loss. In  $\varepsilon$ -SVM regression, the set of training data includes predictor variables and observed response values. The goal is to find a function  $f(x)$  that deviates from  $y_n$  by a value no greater than  $\varepsilon$  for each training point  $x$ , and at the same time is as flat as possible.

#### Linear SVM Regression: Primal Formula

Suppose we have a set of training data where  $x_n$  is a multivariate set of  $N$  observations with observed response values  $y_n$ .

To find the linear function

$$f(x) = x'\beta + b,$$

and ensure that it is as flat as possible, find  $f(x)$  with the minimal norm value ( $\beta'\beta$ ). This is formulated as a convex optimization problem to minimize

$$J(\beta) = \frac{1}{2} \beta' \beta$$

subject to all residuals having a value less than  $\varepsilon$ ; or, in equation form:

$$\forall n : |y_n - (x_n' \beta + b)| \leq \varepsilon.$$

It is possible that no such function  $f(x)$  exists to satisfy these constraints for all points. To deal with otherwise infeasible constraints, introduce slack variables  $\xi_n$  and  $\xi_n^*$  for each point. This approach is similar to the “soft margin” concept in SVM classification, because the slack variables allow regression errors to exist up to the value of  $\xi_n$  and  $\xi_n^*$ , yet still satisfy the required conditions.

Including slack variables leads to the objective function, also known as the primal formula[5]:

$$J(\beta) = \frac{1}{2} \beta' \beta + C \sum_{n=1}^N (\xi_n + \xi_n^*),$$

subject to:

$$\forall n : y_n - (x_n' \beta + b) \leq \varepsilon + \xi_n$$

$$\forall n : (x_n' \beta + b) - y_n \leq \varepsilon + \xi_n^*$$

$$\forall n : \xi_n^* \geq 0$$

$$\forall n : \xi_n \geq 0.$$

The constant  $C$  is the box constraint, a positive numeric value that controls the penalty imposed on observations that lie outside the epsilon margin ( $\varepsilon$ ) and helps to prevent overfitting (regularization). This value determines the trade-off between the flatness of  $f(x)$  and the amount up to which deviations larger than  $\varepsilon$  are tolerated.

The linear  $\varepsilon$ -insensitive loss function ignores errors that are within  $\varepsilon$  distance of the observed value by treating them as equal to zero. The loss is measured based on the distance between observed value  $y$  and the  $\varepsilon$  boundary. This is formally described by

$$L_\varepsilon = \begin{cases} 0 & \text{if } |y - f(x)| \leq \varepsilon \\ |y - f(x)| - \varepsilon & \text{otherwise} \end{cases}$$

### Linear SVM Regression: Dual Formula

The optimization problem previously described is computationally simpler to solve in its Lagrange dual formulation. The solution to the dual problem provides a lower bound to the solution of the primal (minimization) problem. The optimal values of the primal and dual problems need not be equal, and the difference is called the “duality gap.” But when the problem is convex and satisfies a constraint qualification condition, the value of the optimal solution to the primal problem is given by the solution of the dual problem.

To obtain the dual formula, construct a Lagrangian function from the primal function by introducing nonnegative multipliers  $\alpha_n$  and  $\alpha_n^*$  for each observation  $x_n$ . This leads to the dual formula, where we minimize

$$L(\alpha) = \frac{1}{2} \sum_{i=1}^N \sum_{j=1}^N (\alpha_i - \alpha_i^*) (\alpha_j - \alpha_j^*) x_i' x_j + \varepsilon \sum_{i=1}^N (\alpha_i + \alpha_i^*) + \sum_{i=1}^N y_i (\alpha_i^* - \alpha_i)$$

subject to the constraints

$$\sum_{n=1}^N (\alpha_n - \alpha_n^*) = 0$$

$$\forall n : 0 \leq \alpha_n \leq C$$

$$\forall n : 0 \leq \alpha_n^* \leq C.$$

The  $\beta$  parameter can be completely described as a linear combination of the training observations using the equation

$$\beta = \sum_{n=1}^N (\alpha_n - \alpha_n^*) x_n.$$

The function used to predict new values depends only on the support vectors:

$$f(x) = \sum_{n=1}^N (\alpha_n - \alpha_n^*) (x_n' x) + b.$$

The Karush-Kuhn-Tucker (KKT) complementarity conditions are optimization constraints required to obtain optimal solutions. For linear SVM regression, these conditions are

$$\forall n : \alpha_n (\varepsilon + \xi_n - y_n + x_n' \beta + b) = 0$$

$$\forall n : \alpha_n^* (\varepsilon + \xi_n^* + y_n - x_n' \beta - b) = 0$$

$$\forall n : \xi_n (C - \alpha_n) = 0$$

$$\forall n : \xi_n^* (C - \alpha_n^*) = 0.$$

These conditions indicate that all observations strictly inside the epsilon tube have Lagrange multipliers  $\alpha_n = 0$  and  $\alpha_n^* = 0$ . If either  $\alpha_n$  or  $\alpha_n^*$  is not zero, then the corresponding observation is called a *support vector*.



The property Alpha of a trained SVM model stores the difference between two Lagrange multipliers of support vectors,  $a_n - a_n^*$ . The properties SupportVectors and Bias store  $x_n$  and  $b$ , respectively.

### Nonlinear SVM Regression: Primal Formula

Some regression problems cannot adequately be described using a linear model. In such a case, the Lagrange dual formulation allows the previously-described technique to be extended to nonlinear functions.

Obtain a nonlinear SVM regression model by replacing the dot product  $x_1'x_2$  with a nonlinear kernel function  $G(x_1, x_2) = \langle \varphi(x_1), \varphi(x_2) \rangle$ , where  $\varphi(x)$  is a transformation that maps  $x$  to a high-dimensional space. Statistics and Machine Learning Toolbox provides the following built-in semidefinite kernel functions.

Kernel Name	Kernel Function
Linear (dot product)	$G(x_j, x_k) = x_j'x_k$
Gaussian	$G(x_j, x_k) = \exp\left(-\ x_j - x_k\ ^2\right)$
Polynomial	$G(x_j, x_k) = (1 + x_j'x_k)^p$ , where $p$ is in the set $\{2, 3, \dots\}$ .

The *Gram matrix* is an  $n$ -by- $n$  matrix that contains elements  $g_{ij} = G(x_i, x_j)$ . Each element  $g_{ij}$  is equal to the inner product of the predictors as transformed by  $\varphi$ . However, we do not need to know  $\varphi$ , because we can use the kernel function to generate Gram matrix directly. Using this method, nonlinear SVM finds the optimal function  $f(x)$  in the transformed predictor space.

### Nonlinear SVM Regression: Dual Formula

The dual formula for nonlinear SVM regression replaces the inner product of the predictors ( $x_i'x_j$ ) with the corresponding element of the Gram matrix ( $g_{ij}$ ).

Nonlinear SVM regression finds the coefficients that minimize

$$L(\alpha) = \frac{1}{2} \sum_{i=1}^N \sum_{j=1}^N (\alpha_i - \alpha_i^*) (\alpha_j - \alpha_j^*) G(x_i, x_j) + \varepsilon \sum_{i=1}^N (\alpha_i + \alpha_i^*) - \sum_{i=1}^N y_i (\alpha_i - \alpha_i^*)$$

subject to

$$\begin{aligned}\sum_{n=1}^N (\alpha_n - \alpha_n^*) &= 0 \\ \forall n : 0 &\leq \alpha_n \leq C \\ \forall n : 0 &\leq \alpha_n^* \leq C.\end{aligned}$$

The function used to predict new values is equal to

$$f(x) = \sum_{n=1}^N (\alpha_n - \alpha_n^*) G(x_n, x) + b.$$

The KKT complementarity conditions are

$$\begin{aligned}\forall n : \alpha_n (\varepsilon + \xi_n - y_n + f(x_n)) &= 0 \\ \forall n : \alpha_n^* (\varepsilon + \xi_n^* + y_n - f(x_n)) &= 0 \\ \forall n : \xi_n (C - \alpha_n) &= 0 \\ \forall n : \xi_n^* (C - \alpha_n^*) &= 0.\end{aligned}$$

## Solving the SVM Regression Optimization Problem

### Solver Algorithms

The minimization problem can be expressed in standard quadratic programming form and solved using common quadratic programming techniques. However, it can be computationally expensive to use quadratic programming algorithms, especially since the Gram matrix may be too large to be stored in memory. Using a decomposition method instead can speed up the computation and avoid running out of memory.

*Decomposition methods* (also called *chunking and working set methods*) separate all observations into two disjoint sets: the working set and the remaining set. A decomposition method modifies only the elements in the working set in each iteration. Therefore, only some columns of the Gram matrix are needed in each iteration, which reduces the amount of storage needed for each iteration.

*Sequential minimal optimization* (SMO) is the most popular approach for solving SVM problems[4]. SMO performs a series of two-point optimizations. In each iteration, a

working set of two points are chosen based on a selection rule that uses second-order information. Then the Lagrange multipliers for this working set are solved analytically using the approach described in [2] and [1].

In SVM regression, the gradient vector  $\nabla L$  for the active set is updated after each iteration. The decomposed equation for the gradient vector is

$$(\nabla L)_n = \begin{cases} \sum_{i=1}^N (\alpha_i - \alpha_i^*) G(x_i, x_n) + \varepsilon - y_n, n \leq N \\ -\sum_{i=1}^N (\alpha_i - \alpha_i^*) G(x_i, x_n) + \varepsilon + y_n, n > N \end{cases}.$$

*Iterative single data algorithm* (ISDA) updates one Lagrange multiplier with each iteration[3]. ISDA is often conducted without the bias term  $b$  by adding a small positive constant  $a$  to the kernel function. Dropping  $b$  drops the sum constraint

$$\sum_{n=1}^N (\alpha_n - \alpha_n^*) = 0$$

in the dual equation. This allows us to update one Lagrange multiplier in each iteration, which makes it easier than SMO to remove outliers. ISDA selects the worst KKT violator among all the  $\alpha_n$  and  $\alpha_n^*$  values as the working set to be updated.

### Convergence Criteria

Each of these solver algorithms iteratively compute until the specified convergence criterion is met. There several options for convergence criteria:

- *Feasibility gap* — The feasibility gap is expressed as

$$\Delta = \frac{J(\beta) + L(\alpha)}{J(\beta) + 1},$$

where  $J(\beta)$  is the primal objective and  $L(\alpha)$  is the dual objective. After each iteration, the software evaluates the feasibility gap. If the feasibility gap is less than the value specified by `GapTolerance`, then the algorithm met the convergence criterion and the software returns a solution.

- *Gradient difference* — After each iteration, the software evaluates the gradient vector,  $\nabla L$ . If the difference in gradient vector values for the current iteration and the

previous iteration is less than the value specified by `DeltaGradientTolerance`, then the algorithm met the convergence criterion and the software returns a solution.

- *Largest KKT violation* — After each iteration, the software evaluates the KKT violation for all the  $a_n$  and  $a_n^*$  values. If the largest violation is less than the value specified by `KKTTolerance`, then the algorithm met the convergence criterion and the software returns a solution.

## References

- [1] Fan, R.E. , P.H. Chen, and C.J. Lin. *A Study on SMO-Type Decomposition Methods for Support Vector Machines*. IEEE Transactions on Neural Networks, 17:893–908, 2006.
- [2] Fan, R.E. , P.H. Chen, and C.J. Lin. *Working Set Selection Using Second Order Information for Training Support Vector Machines*. The Journal of machine Learning Research, 6:1871–1918, 2005.
- [3] Huang, T.M., V. Kecman, and I. Kopriva. *Kernel Based Algorithms for Mining Huge Data Sets: Supervised, Semi-Supervised, and Unsupervised Learning*. Springer, New York, 2006.
- [4] Platt, J. *Sequential Minimal Optimization: A Fast Algorithm for Training Support Vector Machines*. Technical Report MSR-TR-98–14, 1999.
- [5] Vapnik, V. *The Nature of Statistical Learning Theory*. Springer, New York, 1995.

## See Also

`RegressionSVM` | `fitrsvm` | `predict` | `resubPredict`

## Related Examples

- “Train Linear Support Vector Machine Regression Model” on page 32-2599
- “Train Support Vector Machine Regression Model” on page 32-2601
- “Cross-Validate SVM Regression Model” on page 32-2603
- “Optimize SVM Regression” on page 32-2605

# Markov Models

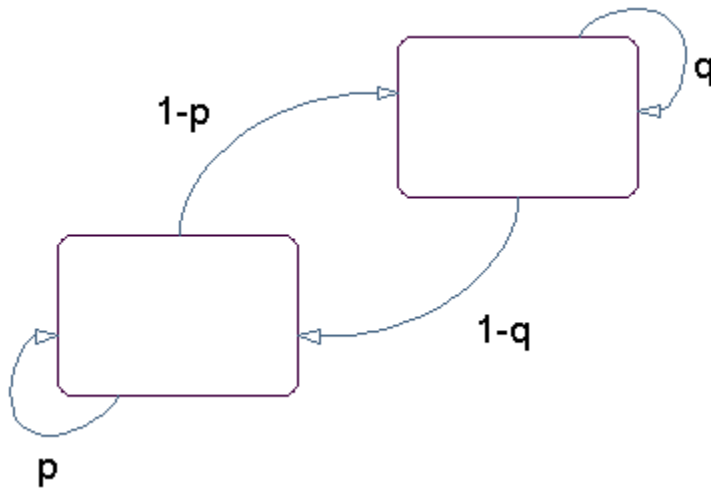
---

- “Markov Chains” on page 26-2
- “Hidden Markov Models (HMM)” on page 26-4

## Markov Chains

Markov processes are examples of stochastic processes—processes that generate random sequences of outcomes or states according to certain probabilities. Markov processes are distinguished by being memoryless—their next state depends only on their current state, not on the history that led them there. Models of Markov processes are used in a wide variety of applications, from daily stock prices to the positions of genes in a chromosome.

A Markov model is given visual representation with a *state diagram*, such as the one below.



### State Diagram for a Markov Model

The rectangles in the diagram represent the possible states of the process you are trying to model, and the arrows represent transitions between states. The label on each arrow represents the probability of that transition. At each step of the process, the model may generate an output, or emission, depending on which state it is in, and then make a transition to another state. An important characteristic of Markov models is that the next state depends only on the current state, and not on the history of transitions that lead to the current state.

For example, for a sequence of coin tosses the two states are heads and tails. The most recent coin toss determines the current state of the model and each subsequent toss determines the transition to the next state. If the coin is fair, the transition probabilities

are all  $1/2$ . The emission might simply be the current state. In more complicated models, random processes at each state will generate emissions. You could, for example, roll a die to determine the emission at any step.

Markov chains are mathematical descriptions of Markov models with a discrete set of states. Markov chains are characterized by:

- A set of states  $\{1, 2, \dots, M\}$
- An  $M$ -by- $M$  transition matrix  $T$  whose  $i,j$  entry is the probability of a transition from state  $i$  to state  $j$ . The sum of the entries in each row of  $T$  must be 1, because this is the sum of the probabilities of making a transition from a given state to each of the other states.
- A set of possible outputs, or *emissions*,  $\{s_1, s_2, \dots, s_N\}$ . By default, the set of emissions is  $\{1, 2, \dots, N\}$ , where  $N$  is the number of possible emissions, but you can choose a different set of numbers or symbols.
- An  $M$ -by- $N$  emission matrix  $E$  whose  $i,k$  entry gives the probability of emitting symbol  $s_k$  given that the model is in state  $i$ .

Markov chains begin in an initial state  $i_0$  at step 0. The chain then transitions to state  $i_1$  with probability  $T_{1i_1}$ , and emits an output  $s_{k_1}$  with probability  $E_{i_1k_1}$ . Consequently, the probability of observing the sequence of states  $i_1i_2\dots i_r$  and the sequence of emissions

$s_{k_1}s_{k_2}\dots s_{k_r}$  in the first  $r$  steps, is

$$T_{1i_1} E_{i_1k_1} T_{i_1i_2} E_{i_2k_2} \dots T_{i_{r-1}i_r} E_{i_rk_r}$$

## See Also

### Related Examples

- “Hidden Markov Models (HMM)” on page 26-4

## Hidden Markov Models (HMM)

<b>In this section...</b>
“Introduction to Hidden Markov Models (HMM)” on page 26-4
“Analyzing Hidden Markov Models” on page 26-6

### Introduction to Hidden Markov Models (HMM)

A hidden Markov model (HMM) is one in which you observe a sequence of emissions, but do not know the sequence of states the model went through to generate the emissions. Analyses of hidden Markov models seek to recover the sequence of states from the observed data.

As an example, consider a Markov model with two states and six possible emissions. The model uses:

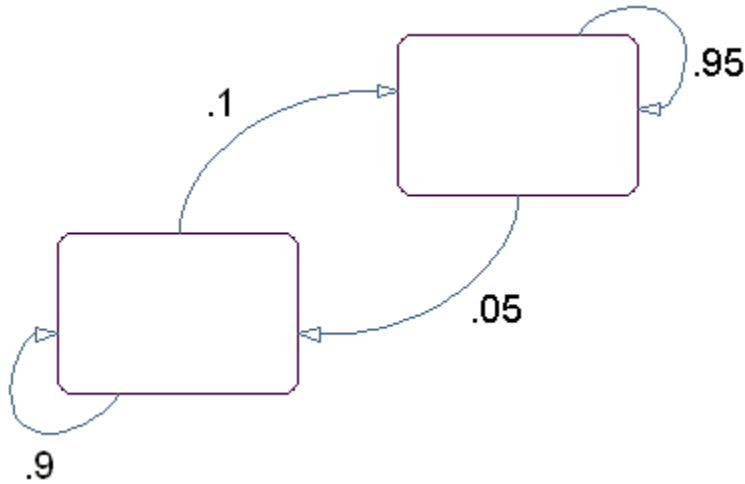
- A red die, having six sides, labeled 1 through 6.
- A green die, having twelve sides, five of which are labeled 2 through 6, while the remaining seven sides are labeled 1.
- A weighted red coin, for which the probability of heads is .9 and the probability of tails is .1.
- A weighted green coin, for which the probability of heads is .95 and the probability of tails is .05.

The model creates a sequence of numbers from the set {1, 2, 3, 4, 5, 6} with the following rules:

- Begin by rolling the red die and writing down the number that comes up, which is the emission.
- Toss the red coin and do one of the following:
  - If the result is heads, roll the red die and write down the result.
  - If the result is tails, roll the green die and write down the result.
- At each subsequent step, you flip the coin that has the same color as the die you rolled in the previous step. If the coin comes up heads, roll the same die as in the previous step. If the coin comes up tails, switch to the other die.



The state diagram for this model has two states, red and green, as shown in the following figure.



You determine the emission from a state by rolling the die with the same color as the state. You determine the transition to the next state by flipping the coin with the same color as the state.

The transition matrix is:

$$T = \begin{bmatrix} 0.9 & 0.1 \\ 0.05 & 0.95 \end{bmatrix}$$

The emissions matrix is:

$$E = \begin{bmatrix} \frac{1}{6} & \frac{1}{6} & \frac{1}{6} & \frac{1}{6} & \frac{1}{6} & \frac{1}{6} \\ \frac{7}{12} & \frac{1}{12} & \frac{1}{12} & \frac{1}{12} & \frac{1}{12} & \frac{1}{12} \end{bmatrix}$$

The model is not hidden because you know the sequence of states from the colors of the coins and dice. Suppose, however, that someone else is generating the emissions without showing you the dice or the coins. All you see is the sequence of emissions. If you start seeing more 1s than other numbers, you might suspect that the model is in the green state, but you cannot be sure because you cannot see the color of the die being rolled.

Hidden Markov models raise the following questions:

- Given a sequence of emissions, what is the most likely state path?
- Given a sequence of emissions, how can you estimate transition and emission probabilities of the model?
- What is the *forward probability* that the model generates a given sequence?
- What is the *posterior probability* that the model is in a particular state at any point in the sequence?

## Analyzing Hidden Markov Models

- “Generating a Test Sequence” on page 26-6
- “Estimating the State Sequence” on page 26-7
- “Estimating Transition and Emission Matrices” on page 26-7
- “Estimating Posterior State Probabilities” on page 26-9
- “Changing the Initial State Distribution” on page 26-10

Statistics and Machine Learning Toolbox functions related to hidden Markov models are:

- `hmmgenerate` — Generates a sequence of states and emissions from a Markov model
- `hmmestimate` — Calculates maximum likelihood estimates of transition and emission probabilities from a sequence of emissions and a known sequence of states
- `hmmtrain` — Calculates maximum likelihood estimates of transition and emission probabilities from a sequence of emissions
- `hmmviterbi` — Calculates the most probable state path for a hidden Markov model
- `hmmdecode` — Calculates the posterior state probabilities of a sequence of emissions

This section shows how to use these functions to analyze hidden Markov models.

### Generating a Test Sequence

The following commands create the transition and emission matrices for the model described in the “Introduction to Hidden Markov Models (HMM)” on page 26-4:

```
TRANS = [.9 .1; .05 .95];  
  
EMIS = [1/6, 1/6, 1/6, 1/6, 1/6, 1/6; ...  
7/12, 1/12, 1/12, 1/12, 1/12, 1/12];
```

To generate a random sequence of states and emissions from the model, use `hmmgenerate`:

```
[seq, states] = hmmgenerate(1000, TRANS, EMIS);
```

The output `seq` is the sequence of emissions and the output `states` is the sequence of states.

`hmmgenerate` begins in state 1 at step 0, makes the transition to state  $i_1$  at step 1, and returns  $i_1$  as the first entry in `states`. To change the initial state, see “Changing the Initial State Distribution” on page 26-10.

### Estimating the State Sequence

Given the transition and emission matrices `TRANS` and `EMIS`, the function `hmmviterbi` uses the Viterbi algorithm to compute the most likely sequence of states the model would go through to generate a given sequence `seq` of emissions:

```
likelystates = hmmviterbi(seq, TRANS, EMIS);
```

`likelystates` is a sequence the same length as `seq`.

To test the accuracy of `hmmviterbi`, compute the percentage of the actual sequence states that agrees with the sequence `likelystates`.

```
sum(states==likelystates)/1000
ans =
    0.8200
```

In this case, the most likely sequence of states agrees with the random sequence 82% of the time.

### Estimating Transition and Emission Matrices

- “Using `hmmestimate`” on page 26-8
- “Using `hmmtrain`” on page 26-8

The functions `hmmestimate` and `hmmtrain` estimate the transition and emission matrices `TRANS` and `EMIS` given a sequence `seq` of emissions.

### Using `hmmestimate`

The function `hmmestimate` requires that you know the sequence of states `states` that the model went through to generate `seq`.

The following takes the emission and state sequences and returns estimates of the transition and emission matrices:

```
[TRANS_EST, EMIS_EST] = hmmestimate(seq, states)
```

```
TRANS_EST =  
0.8989    0.1011  
0.0585    0.9415
```

```
EMIS_EST =  
0.1721    0.1721    0.1749    0.1612    0.1803    0.1393  
0.5836    0.0741    0.0804    0.0789    0.0726    0.1104
```

You can compare the outputs with the original transition and emission matrices, `TRANS` and `EMIS`:

```
TRANS  
TRANS =  
0.9000    0.1000  
0.0500    0.9500
```

```
EMIS  
EMIS =  
0.1667    0.1667    0.1667    0.1667    0.1667    0.1667  
0.5833    0.0833    0.0833    0.0833    0.0833    0.0833
```

### Using `hmmtrain`

If you do not know the sequence of states `states`, but you have initial guesses for `TRANS` and `EMIS`, you can still estimate `TRANS` and `EMIS` using `hmmtrain`.

Suppose you have the following initial guesses for `TRANS` and `EMIS`.

```
TRANS_GUESS = [.85 .15; .1 .9];  
EMIS_GUESS = [.17 .16 .17 .16 .17 .17;.6 .08 .08 .08 .08 .08];
```

You estimate `TRANS` and `EMIS` as follows:

```
[TRANS_EST2, EMIS_EST2] = hmmtrain(seq, TRANS_GUESS, EMIS_GUESS)
```

```
TRANS_EST2 =
0.2286    0.7714
0.0032    0.9968
```

```
EMIS_EST2 =
0.1436    0.2348    0.1837    0.1963    0.2350    0.0066
0.4355    0.1089    0.1144    0.1082    0.1109    0.1220
```

`hmmtrain` uses an iterative algorithm that alters the matrices `TRANS_GUESS` and `EMIS_GUESS` so that at each step the adjusted matrices are more likely to generate the observed sequence, `seq`. The algorithm halts when the matrices in two successive iterations are within a small tolerance of each other.

If the algorithm fails to reach this tolerance within a maximum number of iterations, whose default value is 100, the algorithm halts. In this case, `hmmtrain` returns the last values of `TRANS_EST` and `EMIS_EST` and issues a warning that the tolerance was not reached.

If the algorithm fails to reach the desired tolerance, increase the default value of the maximum number of iterations with the command:

```
hmmtrain(seq, TRANS_GUESS, EMIS_GUESS, 'maxiterations', maxiter)
```

where `maxiter` is the maximum number of steps the algorithm executes.

Change the default value of the tolerance with the command:

```
hmmtrain(seq, TRANS_GUESS, EMIS_GUESS, 'tolerance', tol)
```

where `tol` is the desired value of the tolerance. Increasing the value of `tol` makes the algorithm halt sooner, but the results are less accurate.

Two factors reduce the reliability of the output matrices of `hmmtrain`:

- The algorithm converges to a local maximum that does not represent the true transition and emission matrices. If you suspect this, use different initial guesses for the matrices `TRANS_EST` and `EMIS_EST`.
- The sequence `seq` may be too short to properly train the matrices. If you suspect this, use a longer sequence for `seq`.

### Estimating Posterior State Probabilities

The posterior state probabilities of an emission sequence `seq` are the conditional probabilities that the model is in a particular state when it generates a symbol in `seq`,

given that `seq` is emitted. You compute the posterior state probabilities with `hmmdecode`:

```
PSTATES = hmmdecode(seq, TRANS, EMIS)
```

The output `PSTATES` is an  $M$ -by- $L$  matrix, where  $M$  is the number of states and  $L$  is the length of `seq`. `PSTATES(i, j)` is the conditional probability that the model is in state  $i$  when it generates the  $j$ th symbol of `seq`, given that `seq` is emitted.

`hmmdecode` begins with the model in state 1 at step 0, prior to the first emission. `PSTATES(i, 1)` is the probability that the model is in state  $i$  at the following step 1. To change the initial state, see “Changing the Initial State Distribution” on page 26-10.

To return the logarithm of the probability of the sequence `seq`, use the second output argument of `hmmdecode`:

```
[PSTATES, logpseq] = hmmdecode(seq, TRANS, EMIS)
```

The probability of a sequence tends to 0 as the length of the sequence increases, and the probability of a sufficiently long sequence becomes less than the smallest positive number your computer can represent. `hmmdecode` returns the logarithm of the probability to avoid this problem.

### Changing the Initial State Distribution

By default, Statistics and Machine Learning Toolbox hidden Markov model functions begin in state 1. In other words, the distribution of initial states has all of its probability mass concentrated at state 1. To assign a different distribution of probabilities,  $p = [p_1, p_2, \dots, p_M]$ , to the  $M$  initial states, do the following:

- 1 Create an  $M+1$ -by- $M+1$  augmented transition matrix,  $\hat{T}$  of the following form:

$$\hat{T} = \begin{bmatrix} 0 & p \\ 0 & T \end{bmatrix}$$

where  $T$  is the true transition matrix. The first column of  $\hat{T}$  contains  $M+1$  zeros.  $p$  must sum to 1.

- 2 Create an  $M+1$ -by- $N$  augmented emission matrix,  $\hat{E}$ , that has the following form:

$$\hat{E} = \begin{bmatrix} 0 \\ E \end{bmatrix}$$

If the transition and emission matrices are `TRANS` and `EMIS`, respectively, you create the augmented matrices with the following commands:

```
TRANS_HAT = [0 p; zeros(size(TRANS,1),1) TRANS];
```

```
EMIS_HAT = [zeros(1,size(EMIS,2)); EMIS];
```

## See Also

`hmmdecode` | `hmmestimate` | `hmmgenerate` | `hmmtrain` | `hmmviterbi`

## More About

- “Markov Chains” on page 26-2





# Design of Experiments

---

- “Design of Experiments” on page 27-2
- “Full Factorial Designs” on page 27-3
- “Fractional Factorial Designs” on page 27-5
- “Response Surface Designs” on page 27-9
- “D-Optimal Designs” on page 27-15
- “Improve an Engine Cooling Fan Using Design for Six Sigma Techniques” on page 27-24

## Design of Experiments

Passive data collection leads to a number of problems in statistical modeling. Observed changes in a response variable may be correlated with, but not caused by, observed changes in individual factors (process variables). Simultaneous changes in multiple factors may produce interactions that are difficult to separate into individual effects. Observations may be dependent, while a model of the data considers them to be independent.

Designed experiments address these problems. In a designed experiment, the data-producing process is actively manipulated to improve the quality of information and to eliminate redundant data. A common goal of all experimental designs is to collect data as parsimoniously as possible while providing sufficient information to accurately estimate model parameters.

For example, a simple model of a response  $y$  in an experiment with two controlled factors  $x_1$  and  $x_2$  might look like this:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_1 x_2 + \varepsilon$$

Here  $\varepsilon$  includes both experimental error and the effects of any uncontrolled factors in the experiment. The terms  $\beta_1 x_1$  and  $\beta_2 x_2$  are main effects and the term  $\beta_3 x_1 x_2$  is a two-way interaction effect. A designed experiment would systematically manipulate  $x_1$  and  $x_2$  while measuring  $y$ , with the objective of accurately estimating  $\beta_0$ ,  $\beta_1$ ,  $\beta_2$ , and  $\beta_3$ .

## Full Factorial Designs

### In this section...

“Multilevel Designs” on page 27-3

“Two-Level Designs” on page 27-3

### Multilevel Designs

To systematically vary experimental factors, assign each factor a discrete set of levels. Full factorial designs measure response variables using every treatment (combination of the factor levels). A full factorial design for  $n$  factors with  $N_1, \dots, N_n$  levels requires  $N_1 \times \dots \times N_n$  experimental runs—one for each treatment. While advantageous for separating individual effects, full factorial designs can make large demands on data collection.

As an example, suppose a machine shop has three machines and four operators. If the same operator always uses the same machine, it is impossible to determine if a machine or an operator is the cause of variation in production. By allowing every operator to use every machine, effects are separated. A full factorial list of treatments is generated by the Statistics and Machine Learning Toolbox function `fullfact`:

```
dFF = fullfact([3,4])
dFF =
     1     1
     2     1
     3     1
     1     2
     2     2
     3     2
     1     3
     2     3
     3     3
     1     4
     2     4
     3     4
```

Each of the  $3 \times 4 = 12$  rows of `dFF` represent one machine/operator combination.

### Two-Level Designs

Many experiments can be conducted with two-level factors, using two-level designs. For example, suppose the machine shop in the previous example always keeps the same

operator on the same machine, but wants to measure production effects that depend on the composition of the day and night shifts. The Statistics and Machine Learning Toolbox function `ff2n` generates a full factorial list of treatments:

```
dFF2 = ff2n(4)
dFF2 =
     0     0     0     0
     0     0     0     1
     0     0     1     0
     0     0     1     1
     0     1     0     0
     0     1     0     1
     0     1     1     0
     0     1     1     1
     1     0     0     0
     1     0     0     1
     1     0     1     0
     1     0     1     1
     1     1     0     0
     1     1     0     1
     1     1     1     0
     1     1     1     1
```

Each of the  $2^4 = 16$  rows of `dFF2` represent one schedule of operators for the day (0) and night (1) shifts.

## Fractional Factorial Designs

### In this section...

“Introduction to Fractional Factorial Designs” on page 27-5

“Plackett-Burman Designs” on page 27-5

“General Fractional Designs” on page 27-6

### Introduction to Fractional Factorial Designs

Two-level designs are sufficient for evaluating many production processes. Factor levels of  $\pm 1$  can indicate categorical factors, normalized factor extremes, or simply “up” and “down” from current factor settings. Experimenters evaluating process *changes* are interested primarily in the factor directions that lead to process improvement.

For experiments with many factors, two-level full factorial designs can lead to large amounts of data. For example, a two-level full factorial design with 10 factors requires  $2^{10} = 1024$  runs. Often, however, individual factors or their interactions have no distinguishable effects on a response. This is especially true of higher order interactions. As a result, a well-designed experiment can use fewer runs for estimating model parameters.

Fractional factorial designs use a fraction of the runs required by full factorial designs. A subset of experimental treatments is selected based on an evaluation (or assumption) of which factors and interactions have the most significant effects. Once this selection is made, the experimental design must separate these effects. In particular, significant effects should not be confounded, that is, the measurement of one should not depend on the measurement of another.

### Plackett-Burman Designs

Plackett-Burman designs are used when only main effects are considered significant. Two-level Plackett-Burman designs require a number of experimental runs that are a multiple of 4 rather than a power of 2. The MATLAB function `hadamard` generates these designs:

```
dPB = hadamard(8)
dPB =
    1    1    1    1    1    1    1    1
```

1	-1	1	-1	1	-1	1	-1
1	1	-1	-1	1	1	-1	-1
1	-1	-1	1	1	-1	-1	1
1	1	1	1	-1	-1	-1	-1
1	-1	1	-1	-1	1	-1	1
1	1	-1	-1	-1	-1	1	1
1	-1	-1	1	-1	1	1	-1

Binary factor levels are indicated by  $\pm 1$ . The design is for eight runs (the rows of  $d_{PB}$ ) manipulating seven two-level factors (the last seven columns of  $d_{PB}$ ). The number of runs is a fraction  $8/2^7 = 0.0625$  of the runs required by a full factorial design. Economy is achieved at the expense of confounding main effects with any two-way interactions.

## General Fractional Designs

At the cost of a larger fractional design, you can specify which interactions you wish to consider significant. A design of resolution  $R$  is one in which no  $n$ -factor interaction is confounded with any other effect containing less than  $R - n$  factors. Thus, a resolution III design does not confound main effects with one another but may confound them with two-way interactions (as in “Plackett-Burman Designs” on page 27-5), while a resolution IV design does not confound either main effects or two-way interactions but may confound two-way interactions with each other.

Specify general fractional factorial designs using a full factorial design for a selected subset of basic factors and generators for the remaining factors. Generators are products of the basic factors, giving the levels for the remaining factors. Use the Statistics and Machine Learning Toolbox function `fracfact` to generate these designs:

```
dfF = fracfact('a b c d bcd acd')
dfF =
-1    -1    -1    -1    -1    -1
-1    -1    -1     1     1     1
-1    -1     1    -1     1     1
-1    -1     1     1    -1    -1
-1     1    -1    -1     1    -1
-1     1    -1     1    -1     1
-1     1     1    -1    -1     1
-1     1     1     1     1    -1
 1    -1    -1    -1    -1     1
 1    -1    -1     1     1    -1
 1    -1     1    -1     1    -1
 1    -1     1     1    -1     1
```

1	1	-1	-1	1	1
1	1	-1	1	-1	-1
1	1	1	-1	-1	-1
1	1	1	1	1	1

This is a six-factor design in which four two-level basic factors (a, b, c, and d in the first four columns of `dfF`) are measured in every combination of levels, while the two remaining factors (in the last three columns of `dfF`) are measured only at levels defined by the generators `bcd` and `acd`, respectively. Levels in the generated columns are products of corresponding levels in the columns that make up the generator.

The challenge of creating a fractional factorial design is to choose basic factors and generators so that the design achieves a specified resolution in a specified number of runs. Use the Statistics and Machine Learning Toolbox function `fracfactgen` to find appropriate generators:

```
generators = fracfactgen('a b c d e f',4,4)
generators =
    'a'
    'b'
    'c'
    'd'
    'bcd'
    'acd'
```

These are generators for a six-factor design with factors a through f, using  $2^4 = 16$  runs to achieve resolution IV. The `fracfactgen` function uses an efficient search algorithm to find generators that meet the requirements.

An optional output from `fracfact` displays the confounding pattern of the design:

```
[dfF,confounding] = fracfact(generators);
confounding
confounding =
    'Term'      'Generator'      'Confounding'
    'X1'        'a'              'X1'
    'X2'        'b'              'X2'
    'X3'        'c'              'X3'
    'X4'        'd'              'X4'
    'X5'        'bcd'            'X5'
    'X6'        'acd'            'X6'
    'X1*X2'     'ab'             'X1*X2 + X5*X6'
    'X1*X3'     'ac'             'X1*X3 + X4*X6'
```

'X1*X4'	'ad'	'X1*X4 + X3*X6'
'X1*X5'	'abcd'	'X1*X5 + X2*X6'
'X1*X6'	'cd'	'X1*X6 + X2*X5 + X3*X4'
'X2*X3'	'bc'	'X2*X3 + X4*X5'
'X2*X4'	'bd'	'X2*X4 + X3*X5'
'X2*X5'	'cd'	'X1*X6 + X2*X5 + X3*X4'
'X2*X6'	'abcd'	'X1*X5 + X2*X6'
'X3*X4'	'cd'	'X1*X6 + X2*X5 + X3*X4'
'X3*X5'	'bd'	'X2*X4 + X3*X5'
'X3*X6'	'ad'	'X1*X4 + X3*X6'
'X4*X5'	'bc'	'X2*X3 + X4*X5'
'X4*X6'	'ac'	'X1*X3 + X4*X6'
'X5*X6'	'ab'	'X1*X2 + X5*X6'

The confounding pattern shows that main effects are effectively separated by the design, but two-way interactions are confounded with various other two-way interactions.



## Response Surface Designs

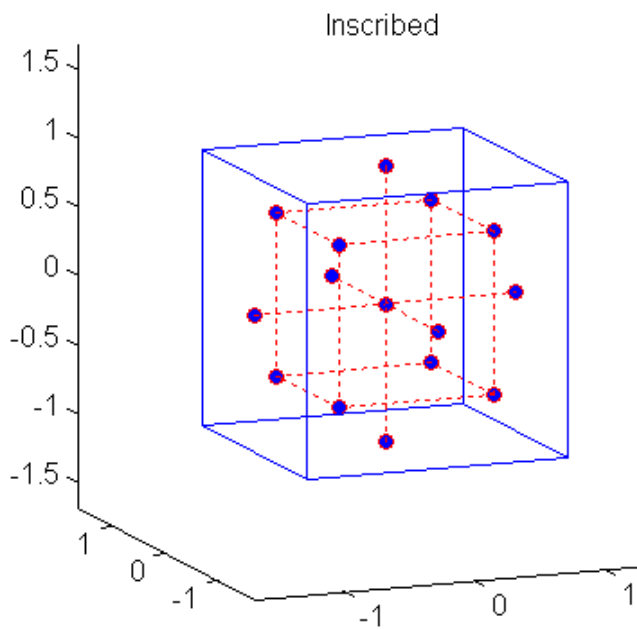
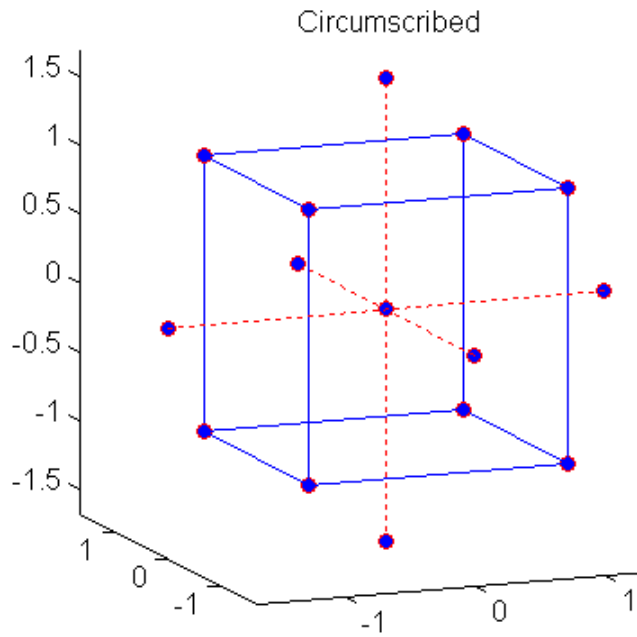
In this section...
“Introduction to Response Surface Designs” on page 27-9
“Central Composite Designs” on page 27-9
“Box-Behnken Designs” on page 27-12

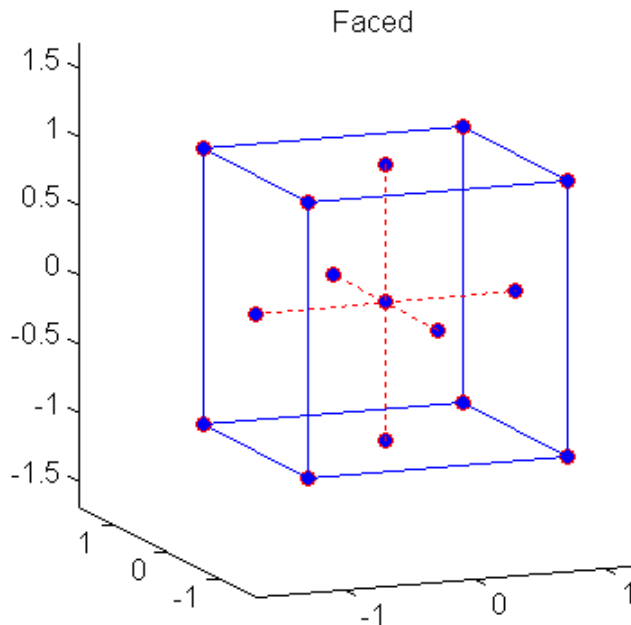
### Introduction to Response Surface Designs

Quadratic response surfaces are simple models that provide a maximum or minimum without making additional assumptions about the form of the response. Quadratic models can be calibrated using full factorial designs with three or more levels for each factor, but these designs generally require more runs than necessary to accurately estimate model parameters. This section discusses designs for calibrating quadratic models that are much more efficient, using three or five levels for each factor, but not using all combinations of levels.

### Central Composite Designs

Central composite designs (CCDs), also known as Box-Wilson designs, are appropriate for calibrating full quadratic models. There are three types of CCDs—circumscribed, inscribed, and faced—pictured below:





Each design consists of a factorial design (the corners of a cube) together with center and star points that allow for estimation of second-order effects. For a full quadratic model with  $n$  factors, CCDs have enough design points to estimate the  $(n+2)(n+1)/2$  coefficients in a full quadratic model with  $n$  factors.

The type of CCD used (the position of the factorial and star points) is determined by the number of factors and by the desired properties of the design. The following table summarizes some important properties. A design is rotatable if the prediction variance depends only on the distance of the design point from the center of the design.

Design	Rotatable	Factor Levels	Uses Points Outside $\pm 1$	Accuracy of Estimates
Circumscribed (CCC)	Yes	5	Yes	Good over entire design space
Inscribed (CCI)	Yes	5	No	Good over central subset of design space
Faced (CCF)	No	3	No	Fair over entire design space; poor for pure quadratic coefficients

Generate CCDs with the Statistics and Machine Learning Toolbox function `ccdesign`:

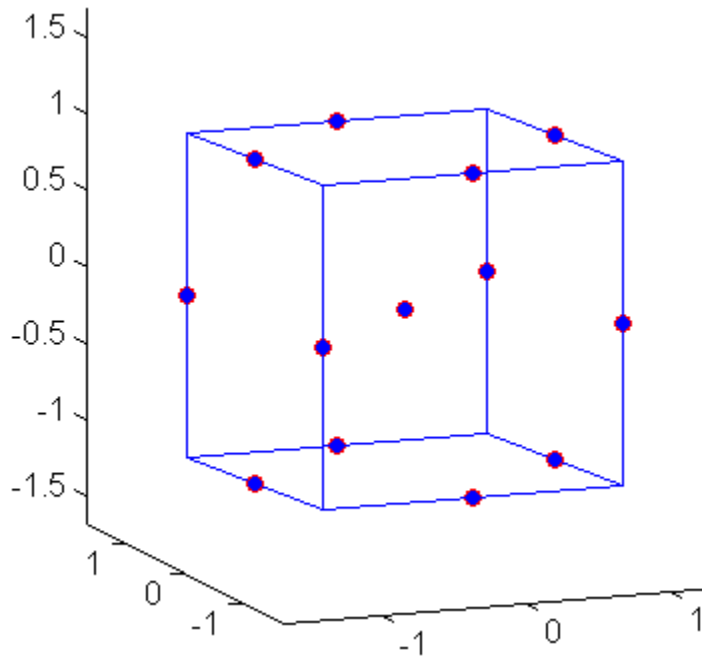
```
dCC = ccdesign(3,'type','circumscribed')
dCC =
-1.0000    -1.0000    -1.0000
-1.0000    -1.0000     1.0000
-1.0000     1.0000    -1.0000
-1.0000     1.0000     1.0000
 1.0000    -1.0000    -1.0000
 1.0000    -1.0000     1.0000
 1.0000     1.0000    -1.0000
 1.0000     1.0000     1.0000
-1.6818         0         0
 1.6818         0         0
     0    -1.6818         0
     0     1.6818         0
     0         0    -1.6818
     0         0     1.6818
     0         0         0
     0         0         0
     0         0         0
     0         0         0
     0         0         0
     0         0         0
     0         0         0
     0         0         0
     0         0         0
     0         0         0
     0         0         0
```

The repeated center point runs allow for a more uniform estimate of the prediction variance over the entire design space.

## Box-Behnken Designs

Like the designs described in “Central Composite Designs” on page 27-9, Box-Behnken designs are used to calibrate full quadratic models. Box-Behnken designs are rotatable and, for a small number of factors (four or less), require fewer runs than CCDs. By avoiding the corners of the design space, they allow experimenters to work around extreme factor combinations. Like an inscribed CCD, however, extremes are then poorly estimated.

The geometry of a Box-Behnken design is pictured in the following figure.



Design points are at the midpoints of edges of the design space and at the center, and do not contain an embedded factorial design.

Generate Box-Behnken designs with the Statistics and Machine Learning Toolbox function `bbdesign`:

```
dBB = bbdesign(3)
dBB =
-1    -1     0
-1     1     0
 1    -1     0
 1     1     0
-1     0    -1
-1     0     1
 1     0    -1
 1     0     1
 0    -1    -1
 0    -1     1
 0     1    -1
 0     1     1
```

0	0	0
0	0	0
0	0	0

Again, the repeated center point runs allow for a more uniform estimate of the prediction variance over the entire design space.

## D-Optimal Designs

### In this section...

“Introduction to D-Optimal Designs” on page 27-15

“Generate D-Optimal Designs” on page 27-16

“Augment D-Optimal Designs” on page 27-18

“Specify Fixed Covariate Factors” on page 27-19

“Specify Categorical Factors” on page 27-20

“Specify Candidate Sets” on page 27-21

### Introduction to D-Optimal Designs

Traditional experimental designs (“Full Factorial Designs” on page 27-3, “Fractional Factorial Designs” on page 27-5, and “Response Surface Designs” on page 27-9) are appropriate for calibrating linear models in experimental settings where factors are relatively unconstrained in the region of interest. In some cases, however, models are necessarily nonlinear. In other cases, certain treatments (combinations of factor levels) may be expensive or infeasible to measure. D-optimal designs are model-specific designs that address these limitations of traditional designs.

A D-optimal design is generated by an iterative search algorithm and seeks to minimize the covariance of the parameter estimates for a specified model. This is equivalent to maximizing the determinant  $D = |X^T X|$ , where  $X$  is the design matrix of model terms (the columns) evaluated at specific treatments in the design space (the rows). Unlike traditional designs, D-optimal designs do not require orthogonal design matrices, and as a result, parameter estimates may be correlated. Parameter estimates may also be locally, but not globally, D-optimal.

There are several Statistics and Machine Learning Toolbox functions for generating D-optimal designs:

Function	Description
candexch	Uses a row-exchange algorithm to generate a D-optimal design with a specified number of runs for a specified model and a specified candidate set. This is the second component of the algorithm used by rowexch.
candgen	Generates a candidate set for a specified model. This is the first component of the algorithm used by rowexch.

Function	Description
<code>cordexch</code>	Uses a coordinate-exchange algorithm to generate a D-optimal design with a specified number of runs for a specified model.
<code>daugment</code>	Uses a coordinate-exchange algorithm to augment an existing D-optimal design with additional runs to estimate additional model terms.
<code>dcovary</code>	Uses a coordinate-exchange algorithm to generate a D-optimal design with fixed covariate factors.
<code>rowexch</code>	Uses a row-exchange algorithm to generate a D-optimal design with a specified number of runs for a specified model. The algorithm calls <code>candgen</code> and then <code>candexch</code> . (Call <code>candexch</code> separately to specify a candidate set.)

The following sections explain how to use these functions to generate D-optimal designs.

---

**Note** The Statistics and Machine Learning Toolbox function `rsmdemo` generates simulated data for experimental settings specified by either the user or by a D-optimal design generated by `cordexch`. It uses the `rstool` interface to visualize response surface models fit to the data, and it uses the `nlintool` interface to visualize a nonlinear model fit to the data.

---

## Generate D-Optimal Designs

Two Statistics and Machine Learning Toolbox algorithms generate D-optimal designs:

- The `cordexch` function uses a coordinate-exchange algorithm
- The `rowexch` function uses a row-exchange algorithm

Both `cordexch` and `rowexch` use iterative search algorithms. They operate by incrementally changing an initial design matrix  $X$  to increase  $D = |X^T X|$  at each step. In both algorithms, there is randomness built into the selection of the initial design and into the choice of the incremental changes. As a result, both algorithms may return locally, but not globally, D-optimal designs. Run each algorithm multiple times and select the best result for your final design. Both functions have a 'tries' parameter that automates this repetition and comparison.

At each step, the row-exchange algorithm exchanges an entire row of  $X$  with a row from a design matrix  $C$  evaluated at a candidate set of feasible treatments. The `rowexch`



function automatically generates a  $C$  appropriate for a specified model, operating in two steps by calling the `candgen` and `candexch` functions in sequence. Provide your own  $C$  by calling `candexch` directly. In either case, if  $C$  is large, its static presence in memory can affect computation.

The coordinate-exchange algorithm, by contrast, does not use a candidate set. (Or rather, the candidate set is the entire design space.) At each step, the coordinate-exchange algorithm exchanges a single element of  $X$  with a new element evaluated at a neighboring point in design space. The absence of a candidate set reduces demands on memory, but the smaller scale of the search means that the coordinate-exchange algorithm is more likely to become trapped in a local minimum than the row-exchange algorithm.

For example, suppose you want a design to estimate the parameters in the following three-factor, seven-term interaction model:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + \beta_{12} x_1 x_2 + \beta_{13} x_1 x_3 + \beta_{23} x_2 x_3 + \varepsilon$$

Use `cordexch` to generate a D-optimal design with seven runs:

```
nfactors = 3;
nruns = 7;
[dCE,X] = cordexch(nfactors,nruns,'interaction','tries',10)
dCE =
    -1     1     1
    -1    -1    -1
     1     1     1
    -1     1    -1
     1    -1     1
     1    -1    -1
    -1    -1     1
X =
     1    -1     1     1    -1    -1     1
     1    -1    -1    -1     1     1     1
     1     1     1     1     1     1     1
     1    -1     1    -1    -1     1    -1
     1     1    -1     1    -1     1    -1
     1     1    -1    -1    -1    -1     1
     1    -1    -1     1     1    -1    -1
```

Columns of the design matrix  $X$  are the model terms evaluated at each row of the design `dCE`. The terms appear in order from left to right:

- 1 Constant term
- 2 Linear terms (1, 2, 3)
- 3 Interaction terms (12, 13, 23)

Use `X` in a linear regression model fit to response data measured at the design points in `dCE`.

Use `rowexch` in a similar fashion to generate an equivalent design:

```
[dRE,X] = rowexch(nfactors,nruns,'interaction','tries',10)
```

```
dRE =
```

```
-1    -1     1
  1    -1     1
  1    -1    -1
  1     1     1
 -1    -1    -1
 -1     1    -1
 -1     1     1
```

```
X =
```

```
  1    -1    -1     1     1    -1    -1
  1     1    -1     1    -1     1    -1
  1     1    -1    -1    -1    -1     1
  1     1     1     1     1     1     1
  1    -1    -1    -1     1     1     1
  1    -1     1    -1    -1     1    -1
  1    -1     1     1    -1    -1     1
```

## Augment D-Optimal Designs

In practice, you may want to add runs to a completed experiment to learn more about a process and estimate additional model coefficients. The `daugment` function uses a coordinate-exchange algorithm to augment an existing D-optimal design.

For example, the following eight-run design is adequate for estimating main effects in a four-factor model:

```
dCEmain = cordexch(4,8)
```

```
dCEmain =
```

```
  1    -1    -1     1
 -1    -1     1     1
 -1     1    -1     1
  1     1     1    -1
```

```

1      1      1      1
-1     1     -1    -1
1     -1     -1    -1
-1    -1      1    -1

```

To estimate the six interaction terms in the model, augment the design with eight additional runs:

```

dCEinteraction = daugment(dCEmain,8,'interaction')
dCEinteraction =
  1     -1     -1      1
 -1    -1      1      1
 -1     1     -1      1
  1      1      1     -1
  1      1      1      1
 -1     1     -1     -1
  1     -1     -1     -1
 -1    -1      1     -1
 -1     1      1      1
 -1    -1     -1     -1
  1     -1      1     -1
  1      1     -1      1
 -1     1      1     -1
  1      1     -1     -1
  1     -1      1      1
  1      1      1     -1

```

The augmented design is full factorial, with the original eight runs in the first eight rows.

The 'start' parameter of the `candexch` function provides the same functionality as `daugment`, but uses a row exchange algorithm rather than a coordinate-exchange algorithm.

## Specify Fixed Covariate Factors

In many experimental settings, certain factors and their covariates are constrained to a fixed set of levels or combinations of levels. These cannot be varied when searching for an optimal design. The `dcovary` function allows you to specify fixed covariate factors in the coordinate exchange algorithm.

For example, suppose you want a design to estimate the parameters in a three-factor linear additive model, with eight runs that necessarily occur at different times. If the

process experiences temporal linear drift, you may want to include the run time as a variable in the model. Produce the design as follows:

```
time = linspace(-1,1,8)';
[dCV,X] = dcovary(3,time,'linear')
dCV =
-1.0000    1.0000    1.0000   -1.0000
 1.0000   -1.0000   -1.0000   -0.7143
-1.0000   -1.0000   -1.0000   -0.4286
 1.0000   -1.0000    1.0000   -0.1429
 1.0000    1.0000   -1.0000    0.1429
-1.0000    1.0000   -1.0000    0.4286
 1.0000    1.0000    1.0000    0.7143
-1.0000   -1.0000    1.0000    1.0000
X =
 1.0000   -1.0000    1.0000    1.0000   -1.0000
 1.0000    1.0000   -1.0000   -1.0000   -0.7143
 1.0000   -1.0000   -1.0000   -1.0000   -0.4286
 1.0000    1.0000   -1.0000    1.0000   -0.1429
 1.0000    1.0000    1.0000   -1.0000    0.1429
 1.0000   -1.0000    1.0000   -1.0000    0.4286
 1.0000    1.0000    1.0000    1.0000    0.7143
 1.0000   -1.0000   -1.0000    1.0000    1.0000
```

The column vector `time` is a fixed factor, normalized to values between  $\pm 1$ . The number of rows in the fixed factor specifies the number of runs in the design. The resulting design `dCV` gives factor settings for the three controlled model factors at each time.

## Specify Categorical Factors

Categorical factors take values in a discrete set of levels. Both `cordexch` and `rowexch` have a `'categorical'` parameter that allows you to specify the indices of categorical factors and a `'levels'` parameter that allows you to specify a number of levels for each factor.

For example, the following eight-run design is for a linear additive model with five factors in which the final factor is categorical with three levels:

```
dCEcat = cordexch(5,8,'linear','categorical',5,'levels',3)
dCEcat =
-1    -1     1     1     2
-1    -1    -1    -1     3
 1     1     1     1     3
```

```

1      1      -1     -1     2
1     -1     -1      1     3
-1     1     -1      1     1
-1     1      1     -1     3
1     -1      1     -1     1

```

## Specify Candidate Sets

The row-exchange algorithm exchanges rows of an initial design matrix  $X$  with rows from a design matrix  $C$  evaluated at a candidate set of feasible treatments. The `rowexch` function automatically generates a  $C$  appropriate for a specified model, operating in two steps by calling the `candgen` and `candexch` functions in sequence. Provide your own  $C$  by calling `candexch` directly.

For example, the following uses `rowexch` to generate a five-run design for a two-factor pure quadratic model using a candidate set that is produced internally:

```

dRE1 = rowexch(2,5,'purequadratic','tries',10)
dRE1 =
-1      1
 0      0
 1     -1
 1      0
 1      1

```

The same thing can be done using `candgen` and `candexch` in sequence:

```

[dC,C] = candgen(2,'purequadratic') % Candidate set, C
dC =
-1     -1
 0     -1
 1     -1
-1      0
 0      0
 1      0
-1      1
 0      1
 1      1
C =
 1     -1     -1      1      1
 1      0     -1      0      1
 1      1     -1      1      1
 1     -1      0      1      0

```

```
1 0 0 0 0
1 1 0 1 0
1 -1 1 1 1
1 0 1 0 1
1 1 1 1 1
treatments = candexch(C,5,'tries',10) % D-opt subset
treatments =
2
1
7
3
4
dRE2 = dC(treatments,:) % Display design
dRE2 =
0 -1
-1 -1
-1 1
1 -1
-1 0
```

You can replace `C` in this example with a design matrix evaluated at your own candidate set. For example, suppose your experiment is constrained so that the two factors cannot have extreme settings simultaneously. The following produces a restricted candidate set:

```
constraint = sum(abs(dC),2) < 2; % Feasible treatments
my_dC = dC(constraint,:)
my_dC =
0 -1
-1 0
0 0
1 0
0 1
```

Use the `x2fx` function to convert the candidate set to a design matrix:

```
my_C = x2fx(my_dC,'purequadratic')
my_C =
1 0 -1 0 1
1 -1 0 1 0
1 0 0 0 0
1 1 0 1 0
1 0 1 0 1
```

Find the required design in the same manner:

```
my_treatments = candexch(my_C,5,'tries',10) % D-opt subset
my_treatments =
    2
    4
    5
    1
    3
my_dRE = my_dC(my_treatments,:) % Display design
my_dRE =
    -1     0
     1     0
     0     1
     0    -1
     0     0
```

## Improve an Engine Cooling Fan Using Design for Six Sigma Techniques

This example shows how to improve the performance of an engine cooling fan through a Design for Six Sigma approach using Define, Measure, Analyze, Improve, and Control (DMAIC). The initial fan does not circulate enough air through the radiator to keep the engine cool during difficult conditions. First the example shows how to design an experiment to investigate the effect of three performance factors: fan distance from the radiator, blade-tip clearance, and blade pitch angle. It then shows how to estimate optimum values for each factor, resulting in a design that produces airflows beyond the goal of 875 ft<sup>3</sup> per minute using test data. Finally it shows how to use simulations to verify that the new design produces airflow according to the specifications in more than 99.999% of the fans manufactured. This example uses MATLAB, Statistics and Machine Learning Toolbox, and Optimization Toolbox.

### Define the Problem

This example addresses an engine cooling fan design that is unable to pull enough air through the radiator to keep the engine cool during difficult conditions, such as stop-and-go traffic or hot weather). Suppose you estimate that you need airflow of at least 875 ft<sup>3</sup>/min to keep the engine cool during difficult conditions. You need to evaluate the current design and develop an alternative design that can achieve the target airflow.

### Assess Cooling Fan Performance

Navigate to a folder containing sample data.

```
cd(matlabroot)
cd('help/toolbox/stats/examples')
```

Load the sample data.

```
load OriginalFan
```

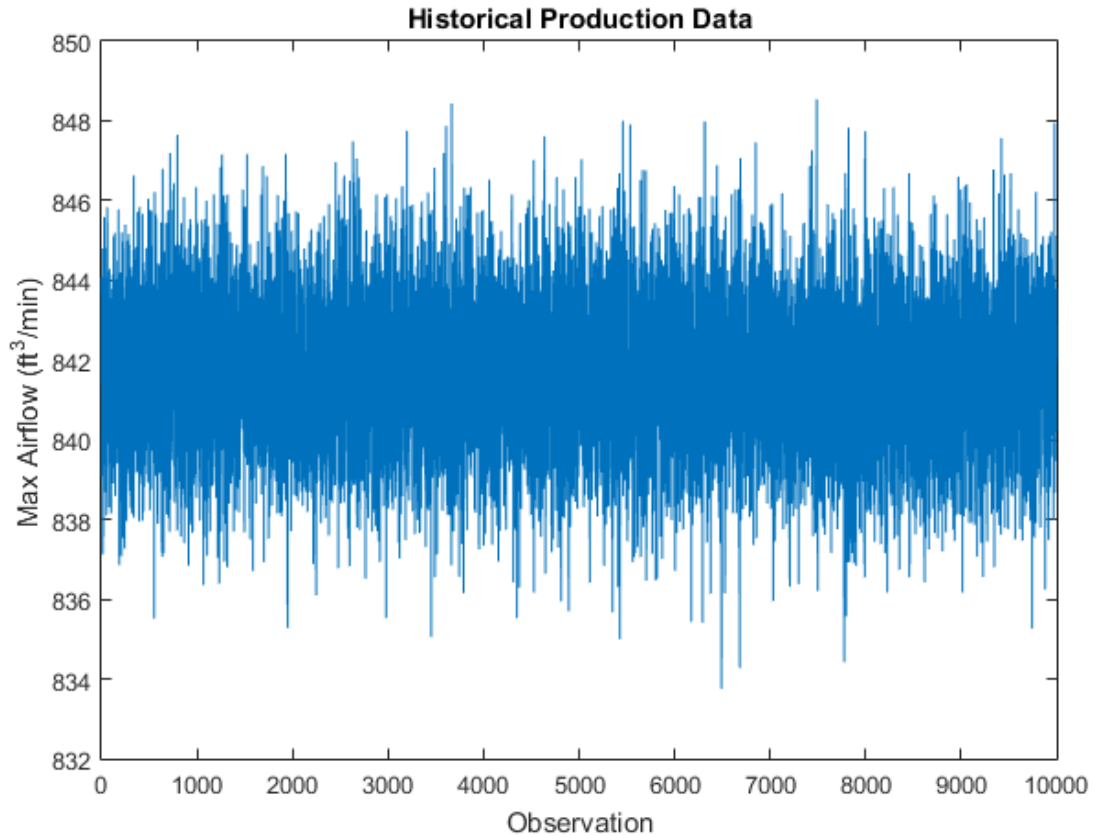
The data consists of 10,000 measurements (historical production data) of the existing cooling fan performance.

Plot the data to analyze the current fan's performance.

```
plot(originalfan)
xlabel('Observation')
```

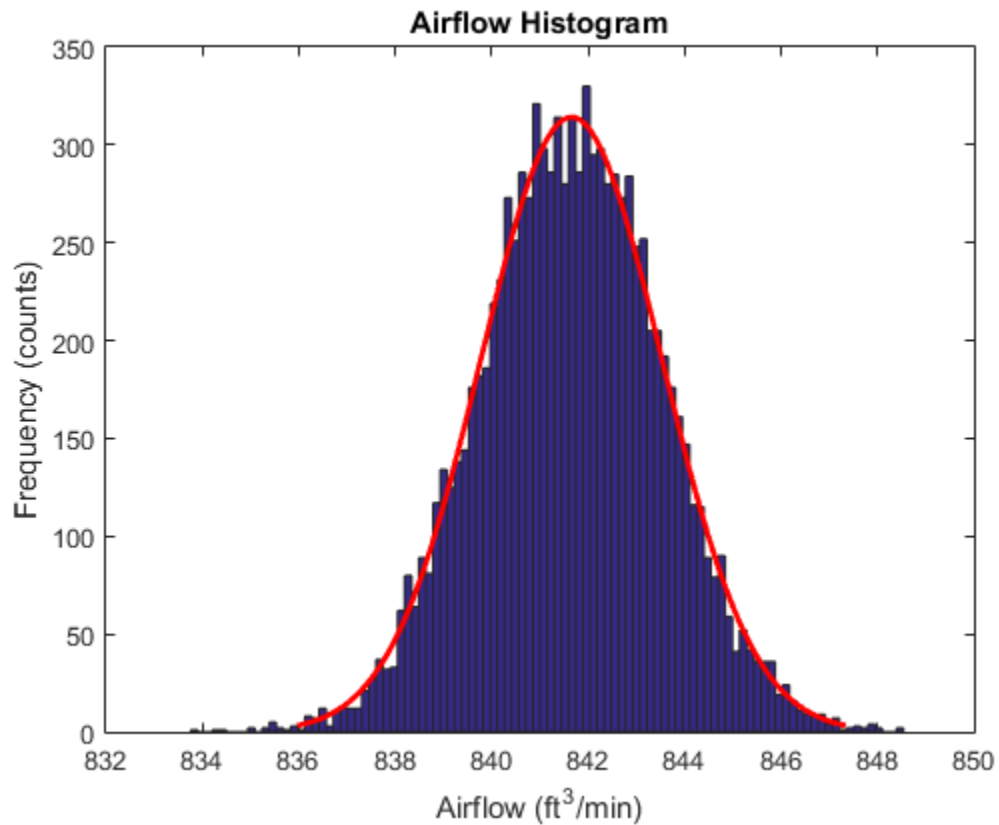


```
ylabel('Max Airflow (ft3/min)')  
title('Historical Production Data')
```



The data is centered around 842 ft<sup>3</sup>/min and most values fall within the range of about 8 ft<sup>3</sup>/min. The plot does not tell much about the underlying distribution of data, however. Plot the histogram and fit a normal distribution to the data.

```
figure()  
histfit(originalfan) % Plot histogram with normal distribution fit  
format shortg  
xlabel('Airflow (ft3/min)')  
ylabel('Frequency (counts)')  
title('Airflow Histogram')
```



```
pd = fitdist(originalfan,'normal') % Fit normal distribution to data
```

```
pd =
```

```
NormalDistribution
```

```
Normal distribution
```

```
mu = 841.652 [841.616, 841.689]
```

```
sigma = 1.8768 [1.85114, 1.90318]
```

`fitdist` fits a normal distribution to data and estimates the parameters from data. The estimate for the mean airflow speed is 841.652 ft<sup>3</sup>/min, and the 95% confidence interval for the mean airflow speed is (841.616, 841.689). This estimate makes it clear that the

current fan is not close to the required 875 ft<sup>3</sup>/min. There is need to improve the fan design to achieve the target airflow.

### Determine Factors That Affect Fan Performance

Evaluate the factors that affect cooling fan performance using design of experiments (DOE). The response is the cooling fan airflow rate (ft<sup>3</sup>/min). Suppose that the factors that you can modify and control are:

- Distance from radiator
- Pitch angle
- Blade tip clearance

In general, fluid systems have nonlinear behavior. Therefore, use a response surface design to estimate any nonlinear interactions among the factors. Generate the experimental runs for a Box-Behnken design on page 27-12 in coded (normalized) variables [-1, 0, +1].

```
CodedValue = bbdesign(3)
```

```
CodedValue =
```

```

-1    -1    0
-1     1    0
 1    -1    0
 1     1    0
-1     0   -1
-1     0    1
 1     0   -1
 1     0    1
 0    -1   -1
 0    -1    1
 0     1   -1
 0     1    1
 0     0    0
 0     0    0
 0     0    0

```

The first column is for the distance from radiator, the second column is for the pitch angle, and the third column is for the blade tip clearance. Suppose you want to test the effects of the variables at the following minimum and maximum values.

Distance from radiator: 1 to 1.5 inches

Pitch angle: 15 to 35 degrees  
Blade tip clearance: 1 to 2 inches

Randomize the order of the runs, convert the coded design values to real-world units, and perform the experiment in the order specified.

```
runorder = randperm(15);      % Random permutation of the runs
bounds = [1 1.5;15 35;1 2]; % Min and max values for each factor

RealValue = zeros(size(CodedValue));
for i = 1:size(CodedValue,2) % Convert coded values to real-world units
    zmax = max(CodedValue(:,i));
    zmin = min(CodedValue(:,i));
    RealValue(:,i) = interp1([zmin zmax],bounds(i,:),CodedValue(:,i));
end
```

Suppose that at the end of the experiments, you collect the following response values in the variable `TestResult`.

```
TestResult = [837 864 829 856 880 879 872 874 834 833 860 859 874 876 875]';
```

Display the design values and the response.

```
disp({'Run Number', 'Distance', 'Pitch', 'Clearance', 'Airflow'})
disp(sortrows([runorder' RealValue TestResult]))
```

'Run Number'	'Distance'	'Pitch'	'Clearance'	'Airflow'
1	1.5	35	1.5	856
2	1.25	25	1.5	876
3	1.5	25	1	872
4	1.25	25	1.5	875
5	1	35	1.5	864
6	1.25	25	1.5	874
7	1.25	15	2	833
8	1.5	15	1.5	829
9	1.25	15	1	834
10	1	15	1.5	837
11	1.5	25	2	874
12	1	25	1	880
13	1.25	35	1	860
14	1	25	2	879
15	1.25	35	2	859

Save the design values and the response in a table.

```
Expmt = table(runorder', CodedValue(:,1), CodedValue(:,2), CodedValue(:,3), ...
    TestResult, 'VariableNames', {'RunNumber', 'D', 'P', 'C', 'Airflow'});
```

D stands for Distance, P stands for Pitch, and C stands for Clearance. Based on the experimental test results, the airflow rate is sensitive to the changing factors values. Also, four experimental runs meet or exceed the target airflow rate of 875 ft<sup>3</sup>/min (runs 2, 4, 12, and 14). However, it is not clear which, if any, of these runs is the optimal one. In addition, it is not obvious how robust the design is to variation in the factors. Create a model based on the current experimental data and use the model to estimate the optimal factor settings.

### Improve the Cooling Fan Performance

The Box-Behnken design enables you to test for nonlinear (quadratic) effects. The form of the quadratic model is:

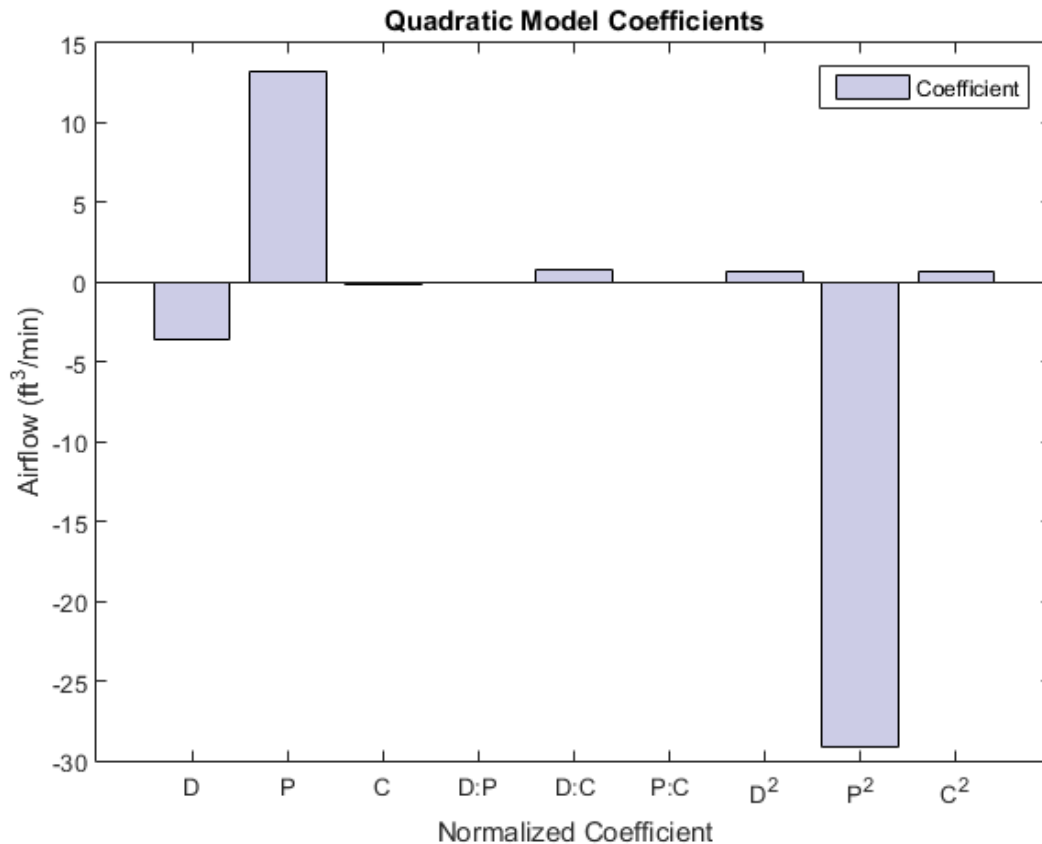
$$AF = \beta_0 + \beta_1 * Distance + \beta_2 * Pitch + \beta_3 * Clearance + \beta_4 * Distance * Pitch + \beta_5 * Distance * Clearance + \beta_6 * Pitch * Clearance + \beta_7 * Distance^2 + \beta_8 * Pitch^2 + \beta_9 * Clearance^2,$$

where  $AF$  is the airflow rate and  $B_i$  is the coefficient for the term  $i$ . Estimate the coefficients of this model using the `fitlm` function from Statistics and Machine Learning Toolbox.

```
mdl = fitlm(Expmt, 'Airflow~D*P*C-D:P:C+D^2+P^2+C^2');
```

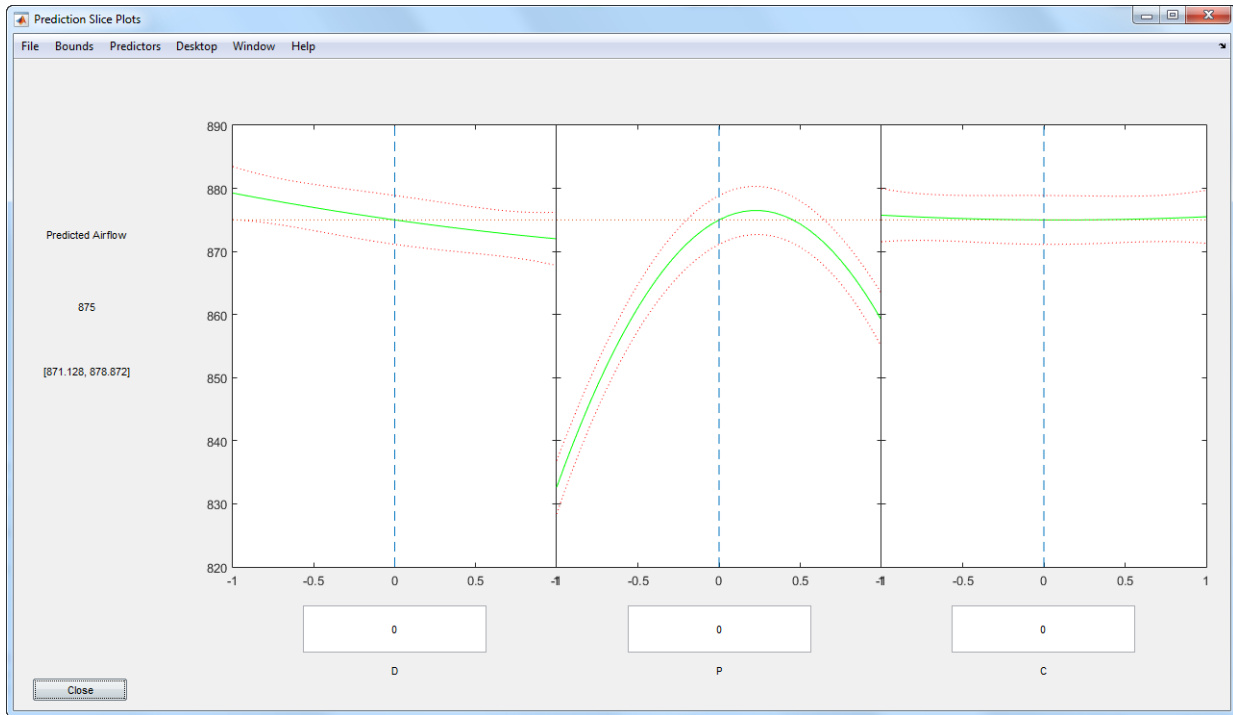
Display the magnitudes of the coefficients (for normalized values) in a bar chart.

```
figure()
h = bar(mdl.Coefficients.Estimate(2:10));
set(h, 'facecolor', [0.8 0.8 0.9])
legend('Coefficient')
set(gcf, 'units', 'normalized', 'position', [0.05 0.4 0.35 0.4])
set(gca, 'xticklabel', mdl.CoefficientNames(2:10))
ylabel('Airflow (ft^3/min)')
xlabel('Normalized Coefficient')
title('Quadratic Model Coefficients')
```



The bar chart shows that *Pitch* and *Pitch*<sup>2</sup> are dominant factors. You can look at the relationship between multiple input variables and one output variable by generating a response surface plot. Use `plotSlice` to generate response surface plots for the model `mdl` interactively.

```
plotSlice(mdl)
```



The plot shows the nonlinear relationship of airflow with pitch. Move the blue dashed lines around and see the effect the different factors have on airflow. Although you can use `plotSlice` to determine the optimum factor settings, you can also use Optimization Toolbox to automate the task.

Find the optimal factor settings using the constrained optimization function `fmincon`.

Write the objective function.

```
f = @(x) -x2fx(x, 'quadratic')*mdl.Coefficients.Estimate;
```

The objective function is a quadratic response surface fit to the data. Minimizing the negative airflow using `fmincon` is the same as maximizing the original objective function. The constraints are the upper and lower limits tested (in coded values). Set the initial starting point to be the center of the design of the experimental test matrix.

```
lb = [-1 -1 -1]; % Lower bound
ub = [1 1 1];   % Upper bound
```

```
x0 = [0 0 0]; % Starting point
[optfactors,fval] = fmincon(f,x0,[],[],[],[],lb,ub,[]); % Invoke the solver
```

Local minimum found that satisfies the constraints.

Optimization completed because the objective function is non-decreasing in feasible directions, to within the default value of the function tolerance, and constraints are satisfied to within the default value of the constraint tolerance.

Convert the results to a maximization problem and real-world units.

```
maxval = -fval;
maxloc = (optfactors + 1)';
bounds = [1 1.5;15 35;1 2];
maxloc=bounds(:,1)+maxloc .* ((bounds(:,2) - bounds(:,1))/2);
disp('Optimal Values:')
disp({'Distance', 'Pitch', 'Clearance', 'Airflow'})
disp([maxloc' maxval])
```

```
Optimal Values:
    'Distance'    'Pitch'    'Clearance'    'Airflow'
         1         27.275         1         882.26
```

The optimization result suggests placing the new fan one inch from the radiator, with a one-inch clearance between the tips of the fan blades and the shroud.

Because pitch angle has such a significant effect on airflow, perform additional analysis to verify that a 27.3 degree pitch angle is optimal.

```
load AirflowData
tbl = table(pitch,airflow);
mdl2 = fitlm(tbl,'airflow~pitch^2');
mdl2.Rsquared.Ordinary
```

```
ans =
    0.99632
```

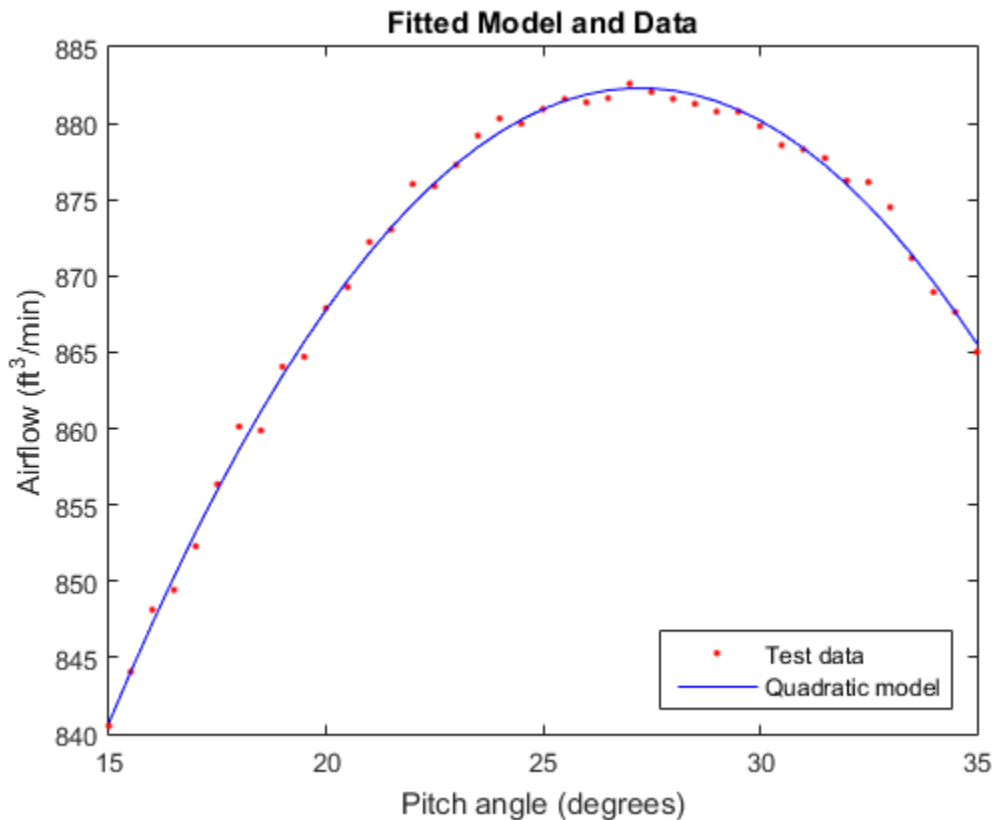
The results show that a quadratic model explains the effect of pitch on the airflow well.

Plot the pitch angle against airflow and impose the fitted model.

```
figure()
plot(pitch,airflow,'.r')
```



```
hold on
ylim([840 885])
line(pitch,mdl2.Fitted,'color','b')
title('Fitted Model and Data')
xlabel('Pitch angle (degrees)')
ylabel('Airflow (ft^3/min)')
legend('Test data','Quadratic model','Location','se')
hold off
```



Find the pitch value that corresponds to the maximum airflow.

```
pitch(find(airflow==max(airflow)))
```

```
ans =
```

```
27
```

The additional analysis confirms that a 27.3 degree pitch angle is optimal.

The improved cooling fan design meets the airflow requirements. You also have a model that approximates the fan performance well based on the factors you can modify in the design. Ensure that the fan performance is robust to variability in manufacturing and installation by performing a sensitivity analysis.

### Sensitivity Analysis

Suppose that, based on historical experience, the manufacturing uncertainty is as follows.

Factor	Real Values	Coded Values
Distance from radiator	1.00 +/- 0.05 inch	1.00 +/- 0.20 inch
Blade pitch angle	27.3 +/- 0.25 degrees	0.227 +/- 0.028 degrees
Blade tip clearance	1.00 +/- 0.125 inch	-1.00 +/- 0.25 inch

Verify that these variations in factors will enable to maintain a robust design around the target airflow. The philosophy of Six Sigma targets a defect rate of no more than 3.4 per 1,000,000 fans. That is, the fans must hit the 875 ft<sup>3</sup>/min target 99.999% of the time.

You can verify the design using Monte Carlo simulation. Generate 10,000 random numbers for three factors with the specified tolerance. First, set the state of the random number generators so results are consistent across different runs.

```
rng('default')
```

Perform the Monte Carlo simulation. Include a noise variable that is proportional to the noise in the fitted model, `mdl` (that is, the RMS error of the model). Because the model coefficients are in coded variables, you must generate `dist`, `pitch`, and `clearance` using the coded definition.

```
dist = random('normal',optfactors(1),0.20,[10000 1]);
pitch = random('normal',optfactors(2),0.028,[10000 1]);
clearance = random('normal',optfactors(3),0.25,[10000 1]);
noise = random('normal',0,mdl2.RMSE,[10000 1]);
```

Calculate airflow for 10,000 random factor combinations using the model.

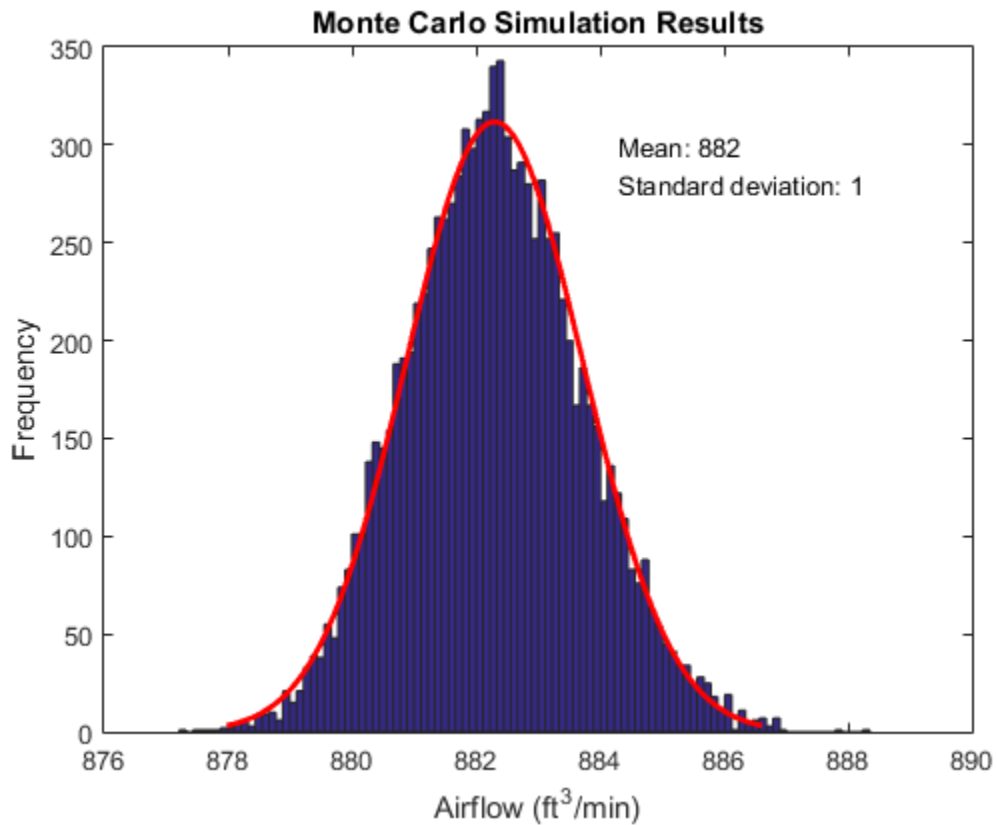
```
simfactor = [dist pitch clearance];
X = x2fx(simfactor, 'quadratic');
```

Add noise to the model (the variation in the data that the model did not account for).

```
simflow = X*mdl.Coefficients.Estimate+noise;
```

Evaluate the variation in the model's predicted airflow using a histogram. To estimate the mean and standard deviation, fit a normal distribution to data.

```
pd = fitdist(simflow, 'normal');
histfit(simflow)
hold on
text(pd.mu+2, 300, ['Mean: ' num2str(round(pd.mu))])
text(pd.mu+2, 280, ['Standard deviation: ' num2str(round(pd.sigma))])
hold off
xlabel('Airflow (ft^3/min)')
ylabel('Frequency')
title('Monte Carlo Simulation Results')
```



The results look promising. The average airflow is 882 ft<sup>3</sup>/min and appears to be better than 875 ft<sup>3</sup>/min for most of the data.

Determine the probability that the airflow is at 875 ft<sup>3</sup>/min or below.

```
format long
pfail = cdf(pd,875)
pass = (1-pfail)*100

pfail =

1.509289008603141e-07
```

```
pass =
    99.999984907109919
```

The design appears to achieve at least 875 ft<sup>3</sup>/min of airflow 99.999% of the time.

Use the simulation results to estimate the process capability.

```
S = capability(simflow, [875.0 890])
pass = (1-S.Pl)*100
```

```
S =
    mu: 8.822982645666709e+02
    sigma: 1.424806876923940
    P: 0.999999816749816
    Pl: 1.509289008603141e-07
    Pu: 3.232128339675335e-08
    Cp: 1.754623760237126
    Cpl: 1.707427788957002
    Cpu: 1.801819731517250
    Cpk: 1.707427788957002
```

```
pass =
    99.9999849071099
```

The Cp value is 1.75. A process is considered high quality when Cp is greater than or equal to 1.6. The Cpk is similar to the Cp value, which indicates that the process is centered. Now implement this design. Monitor it to verify the design process and to ensure that the cooling fan delivers high-quality performance.

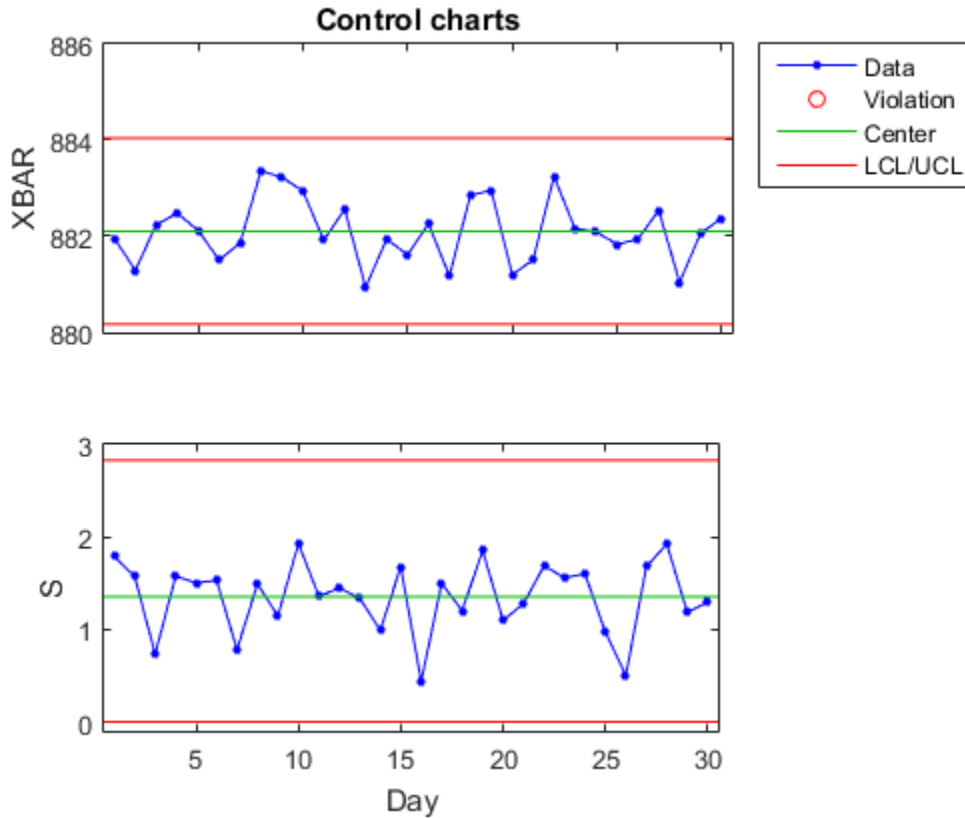
### Control Manufacturing of the Improved Cooling Fan

You can monitor and evaluate the manufacturing and installation process of the new fan using control charts. Evaluate the first 30 days of production of the new cooling fan. Initially, five cooling fans per day were produced. First, load the sample data from the new process.

```
load spcdata
```

Plot the X-bar and S charts.

```
figure()
controlchart(spcflow, 'chart', {'xbar', 's'}) % Reshape the data into daily sets
xlabel('Day')
```



According to the results, the manufacturing process is in statistical control, as indicated by the absence of violations of control limits or nonrandom patterns in the data over time. You can also run a capability analysis on the data to evaluate the process.

```
[row,col] = size(spcflow);
S2 = capability(reshape(spcflow,row*col,1), [875.0 890])
pass = (1-S.P1)*100
```

```
S2 =
```

```
mu: 8.821061141685465e+02
sigma: 1.423887508874697
P: 0.999999684316149
Pl: 3.008932155898586e-07
Pu: 1.479063578225176e-08
Cp: 1.755756676295137
Cpl: 1.663547652525458
Cpu: 1.847965700064817
Cpk: 1.663547652525458
```

pass =

99.9999699106784

The  $C_p$  value of 1.755 is very similar to the estimated value of 1.73. The  $C_{pk}$  value of 1.66 is smaller than the  $C_p$  value. However, only a  $C_{pk}$  value less than 1.33, which indicates that the process shifted significantly toward one of the process limits, is a concern. The process is well within the limits and it achieves the target airflow (875 ft<sup>3</sup>/min) more than 99.999% of the time.





# Statistical Process Control

---

- “Introduction to Statistical Process Control” on page 28-2
- “Control Charts” on page 28-3
- “Capability Studies” on page 28-7

## Introduction to Statistical Process Control

Statistical process control (SPC) refers to a number of different methods for monitoring and assessing the quality of manufactured goods. Combined with methods from the design of experiments, SPC is used in programs that define, measure, analyze, improve, and control development and production processes. These programs are often implemented using “Design for Six Sigma” methodologies.

## Control Charts

A control chart displays measurements of process samples over time. The measurements are plotted together with user-defined *specification limits* and process-defined *control limits*. The process can then be compared with its specifications—to see if it is *in control* or *out of control*.

The chart is just a monitoring tool. Control activity might occur if the chart indicates an undesirable, systematic change in the process. The control chart is used to discover the variation, so that the process can be adjusted to reduce it.

Control charts are created with the `controlchart` function. Any of the following chart types may be specified:

- Xbar or mean
- Standard deviation
- Range
- Exponentially weighted moving average
- Individual observation
- Moving range of individual observations
- Moving average of individual observations
- Proportion defective
- Number of defectives
- Defects per unit
- Count of defects

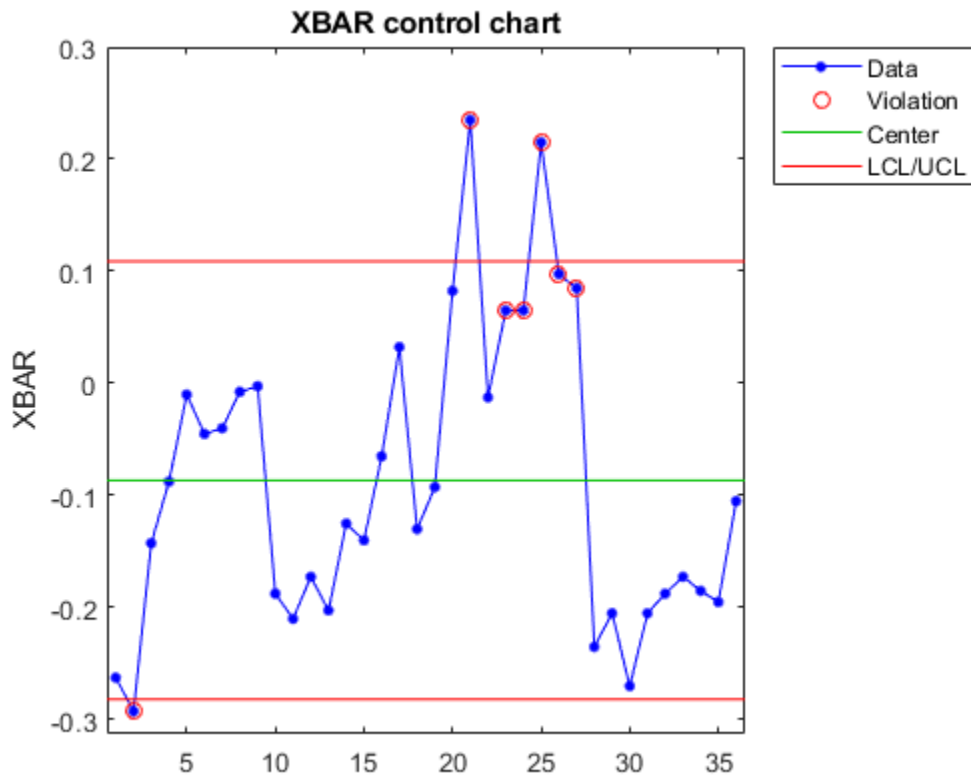
Control rules are specified with the `controlrules` function. The following example illustrates how to use Western Electric rules to mark out of control measurements on an Xbar chart.

First load the sample data.

```
load parts;
```

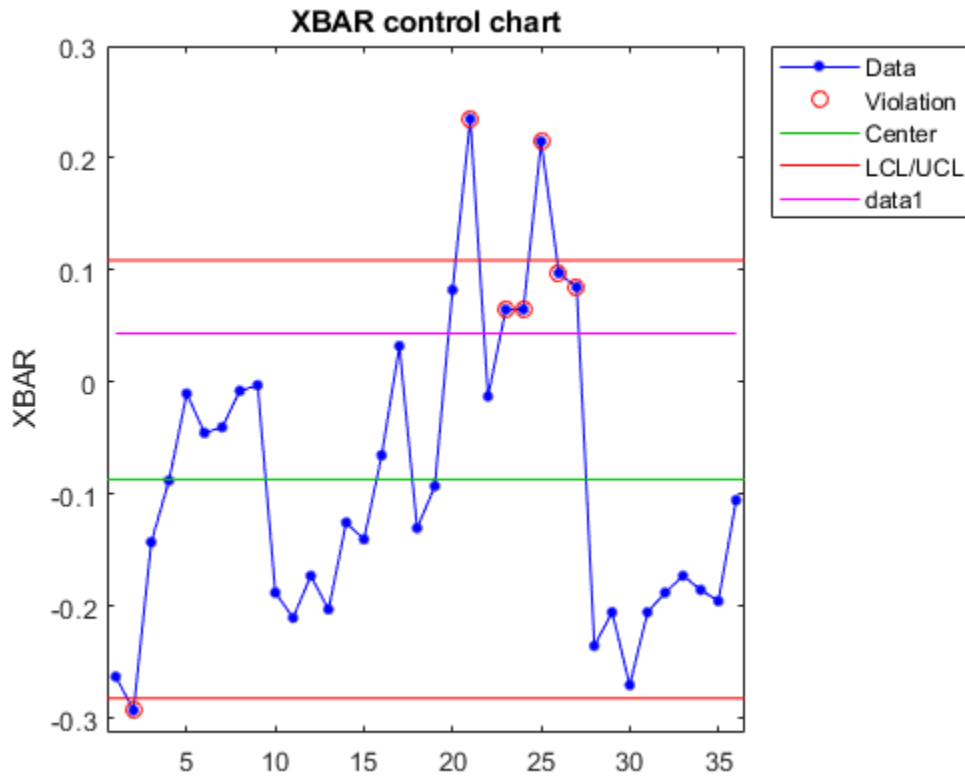
Construct the Xbar control chart using the Western Electric 2 rule (2 of 3 points at least 2 standard errors above the center line) to mark the out of control measurements.

```
st = controlchart(runout, 'rules', 'we2');
```



For a better understanding of the Western Electric 2 rule, calculate and plot the 2 standard errors line on the chart.

```
x = st.mean;  
cl = st.mu;  
se = st.sigma./sqrt(st.n);  
hold on  
plot(cl+2*se, 'm')
```



Identify the measurements that violate the control rule.

```
R = controlrules('we2',x,cl,se);
I = find(R)
```

I =

```
21
23
24
25
26
```



## Capability Studies

Before going into production, many manufacturers run a *capability study* to determine if their process will run within specifications enough of the time. *Capability indices* produced by such a study are used to estimate expected percentages of defective parts.

Capability studies are conducted with the `capability` function. The following capability indices are produced:

- `mu` — Sample mean
- `sigma` — Sample standard deviation
- `P` — Estimated probability of being within the lower (L) and upper (U) specification limits
- `PL` — Estimated probability of being below L
- `PU` — Estimated probability of being above U
- `Cp` —  $(U-L) / (6 * \text{sigma})$
- `CpL` —  $(\text{mu}-L) ./ (3 * \text{sigma})$
- `CpU` —  $(U-\text{mu}) ./ (3 * \text{sigma})$
- `Cpk` —  $\min(C_{pL}, C_{pU})$

As an example, simulate a sample from a process with a mean of 3 and a standard deviation of 0.005:

```
rng default; % For reproducibility
data = normrnd(3,0.005,100,1);
```

Compute capability indices if the process has an upper specification limit of 3.01 and a lower specification limit of 2.99:

```
S = capability(data,[2.99 3.01])
```

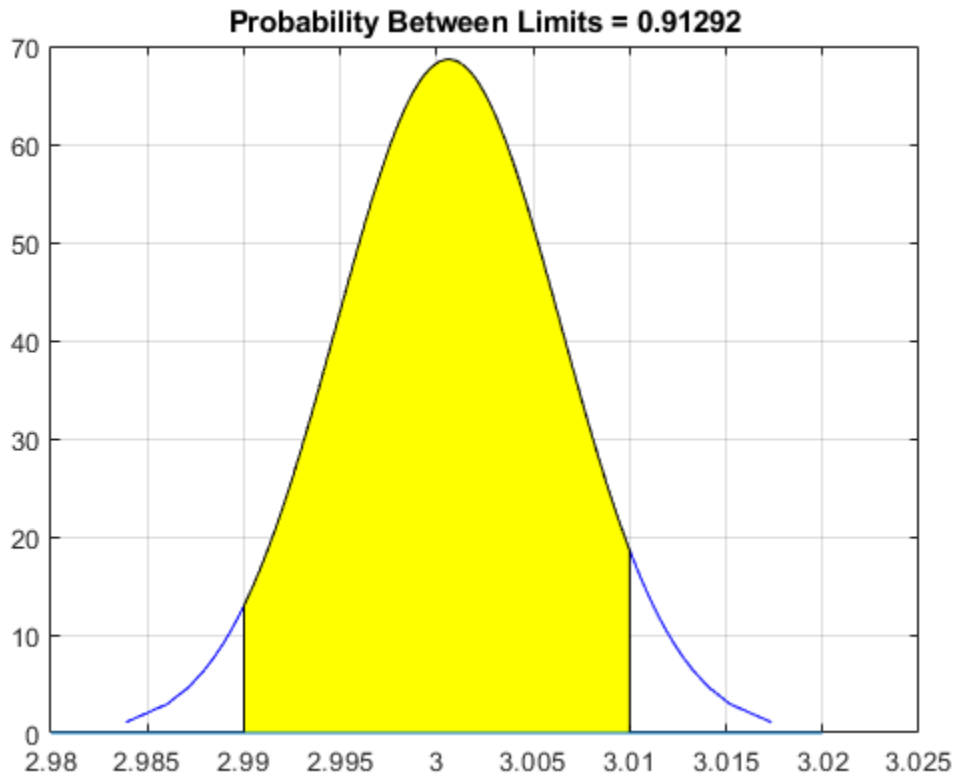
```
S =
```

```
struct with fields:
    mu: 3.0006
    sigma: 0.0058
    P: 0.9129
```

```
Pl: 0.0339
Pu: 0.0532
Cp: 0.5735
Cpl: 0.6088
Cpu: 0.5382
Cpk: 0.5382
```

Visualize the specification and process widths:

```
capaplot(data, [2.99 3.01]);
grid on
```





# Tall Arrays

---

- “Tall Array Support, Usage Notes, and Limitations” on page 29-2
- “Logistic Regression with Tall Arrays” on page 29-30

## Tall Array Support, Usage Notes, and Limitations

### Descriptive Statistics and Visualization

Function	Notes or Limitations
geomean	
harmmean	
kurtosis	
range	
skewness	
zscore	
corr	Only 'Pearson' type is supported.
tabulate	
crosstab	The fourth output, labels, is returned as a cell array containing M unevaluated tall cell arrays, where M is the number of input grouping variables. Each unevaluated tall cell array, labels{j}, contains the labels for one grouping variable.

Function	Notes or Limitations
grpstats	<ul style="list-style-type: none"> <li>• If the input data is a tall array, then all grouping variables must also be tall and have the same number of rows as the data.</li> <li>• The <code>whichstats</code> option cannot be specified as a function handle. In addition to the current built-in options, <code>whichstats</code> can also be: <ul style="list-style-type: none"> <li>• 'Count' — Number of non-NaNs.</li> <li>• 'NNZ' — Number of nonzeros and non-NaNs.</li> <li>• 'Kurtosis' — Compute kurtosis.</li> <li>• 'Skewness' — Compute skewness.</li> <li>• 'all-stats' — Compute all summary statistics.</li> </ul> </li> <li>• Group order is not guaranteed to be the same as the in-memory <code>grpstats</code> computation.</li> <li>• Summary statistics for <i>nonnumeric</i> variables return NaNs.</li> <li>• <code>grpstats</code> always operates on the first dimension.</li> <li>• If the input is a tall table, then the output is also a tall table. However, rather than including row names, the output tall table contains an extra variable <code>GroupLabel</code> that contains the same information.</li> </ul>

Function	Notes or Limitations
binScatterPlot	This function is specifically designed for visualizing large data sets. Instead of plotting millions of data points, which is not very feasible, binScatterPlot summarizes the data points into bins. This “scatter plot of bins” reveals high-level trends in the data.
ksdensity	<ul style="list-style-type: none"><li>• Some options that require extra passes or sorting of the input data are not supported:<ul style="list-style-type: none"><li>• 'BoundaryCorrection'</li><li>• 'Censoring'</li><li>• 'Support' (support is always unbounded).</li></ul></li><li>• Uses standard deviation (instead of median absolute deviation) to compute the bandwidth.</li></ul>

## Probability Distributions

Function	Notes or Limitations
<p><code>datasample</code></p>	<ul style="list-style-type: none"> <li>• <code>datasample</code> is useful as a precursor to plotting a random subset of a very large data set. Sampling a large data set preserves trends in the data without requiring that you plot all the data points.</li> <li>• <code>datasample</code> supports sampling only along the first dimension of the data.</li> <li>• For tall arrays, <code>datasample</code> does not support sampling with replacement. You must specify <code>'Replace', false</code>, for example, <code>datasample(data, k, 'Replace', false)</code>.</li> <li>• The value of <code>'Weights'</code> must be a numeric tall array of the same height as <code>data</code>.</li> <li>• For the syntax <code>[Y, idx] = datasample(___)</code>, the output <code>idx</code> is a tall logical vector of the same height as <code>data</code>. The vector indicates whether each data point is included in the sample.</li> <li>• If you specify a random number stream, then the underlying generator must support multiple streams and substreams. If you do not specify a random number stream, then <code>datasample</code> uses the stream controlled by <code>tallrng</code>.</li> </ul>

## Cluster Analysis

Function	Notes or Limitations
kmeans	<p>Only random sample initialization is supported. Supported syntaxes:</p> <ul style="list-style-type: none"> <li>• <code>idx = kmeans(X,k)</code> performs classic k-means clustering.</li> <li>• <code>[idx,C] = kmeans(X,k)</code> also returns the k cluster centroid locations.</li> <li>• <code>[idx,C,sumd] = kmeans(X,k)</code> additionally returns the k within-cluster sums of point-to-centroid distances.</li> <li>• <code>[___] = kmeans(___ ,Name,Value)</code> specifies additional name-value pair options using any of the other syntaxes. Valid options are: <ul style="list-style-type: none"> <li>• <code>'Start'</code> — Method used to choose the initial cluster centroid positions. Value can be: <ul style="list-style-type: none"> <li>• <code>'plus'</code> (default) — Select k observations from X using a variant of the <code>kmeans++</code> algorithm adapted for tall data.</li> <li>• <code>'sample'</code> — Select k observations from X at random.</li> <li>• Numeric matrix — A k-by-p matrix to explicitly specify starting locations.</li> </ul> </li> <li>• <code>'Options'</code> — An options structure created using the <code>statset</code> function. For tall arrays, <code>kmeans</code> uses the fields listed here and ignores all other fields in the options structure:</li> </ul> </li> </ul>

Function	Notes or Limitations
	<ul style="list-style-type: none"> <li>• 'Display' — Level of display. Choices are 'iter' (default), 'off', and 'final'.</li> <li>• 'MaxIter' — Maximum number of iterations. Default is 100.</li> <li>• 'TolFun' — Convergency tolerance for the within-cluster sums of point-to-centroid distances. Default is 1e-4. This option field only works with tall arrays.</li> </ul>

## Regression

Function	Notes or Limitations
<p>The <code>loss</code> and <code>predict</code> methods of these regression classes support tall arrays:</p> <ul style="list-style-type: none"> <li>• <code>CompactRegressionSVM</code></li> <li>• <code>CompactRegressionGP</code></li> <li>• <code>CompactRegressionTree</code></li> <li>• <code>CompactRegressionEnsemble</code></li> <li>• <code>RegressionLinear</code></li> </ul>	<ul style="list-style-type: none"> <li>• You can use models trained on either in-memory or tall data with these methods.</li> <li>• The <code>loss</code> method of <code>CompactRegressionTree</code> only supports one output argument.</li> </ul>

Function	Notes or Limitations
cvpartition	<ul style="list-style-type: none"><li>• For tall arrays only stratified-HoldOut partitions are supported.</li><li>• <code>C = cvpartition(group, 'HoldOut', P)</code> randomly partitions observations into a training set and a test set with stratification, using the class information in <code>group</code>. <code>P</code> is a scalar such that <math>0 &lt; P &lt; 1</math>.</li><li>• To obtain nonstratified partitions, set a uniform grouping variable from the data samples. For example, assuming <code>X</code> is a tall numeric array, you can use<pre>groups = X(:,1).*0; C = cvpartition(groups, 'HoldOut', P)</pre></li></ul>



Function	Notes or Limitations
fitlm	<ul style="list-style-type: none"> <li>• If any input argument to fitlm is a tall array, then all of the other inputs must be tall arrays as well. This includes nonempty variables supplied with the 'Weights' and 'Exclude' name-value pairs.</li> <li>• The 'RobustOpts' name-value pair is not supported with tall arrays.</li> <li>• For tall data, fitlm returns a CompactLinearModel object that contains most of the same properties as a LinearModel object. The main difference is that the compact object is sensitive to memory requirements. The compact object does not include properties that include the data, or that include an array of the same size as the data. The compact object does not contain these LinearModel properties: <ul style="list-style-type: none"> <li>• Diagnostics</li> <li>• Fitted</li> <li>• ObservationInfo</li> <li>• ObservationNames</li> <li>• Residuals</li> <li>• Steps</li> <li>• Variables</li> </ul> </li> </ul> <p>You can compute the residuals directly from the compact object returned by LM = fitlm(X,Y) using</p> <pre>RES = Y - predict(LM,X); S = LM.RMSE; histogram(RES, linspace(-3*S, 3*S, 51))</pre>

Function	Notes or Limitations
	<ul style="list-style-type: none"><li data-bbox="798 302 1328 395">• If the <code>CompactLinearModel</code> object is missing lower order terms that include categorical factors:<ul style="list-style-type: none"><li data-bbox="836 421 1328 522">• The <code>plotEffects</code> and <code>plotInteraction</code> methods are not supported.</li><li data-bbox="836 534 1328 633">• The <code>anova</code> method with the <code>'components'</code> option is not supported.</li></ul></li></ul>

Function	Notes or Limitations
fitglm	<ul style="list-style-type: none"> <li>• If any input argument to fitglm is a tall array, then all of the other inputs must be tall arrays as well. This includes nonempty variables supplied with the 'Weights', 'Exclude', 'Offset', and 'BinomialSize' name-value pairs.</li> <li>• The default number of iterations is 5. You can change the number of iterations using the 'Options' name-value pair to pass in an options structure. Create an options structure using statset to specify a different value for MaxIter.</li> <li>• For tall data, fitglm returns a CompactGeneralizedLinearModel object that contains most of the same properties as a GeneralizedLinearModel object. The main difference is that the compact object is sensitive to memory requirements. The compact object does not include properties that include the data, or that include an array of the same size as the data. The compact object does not contain these GeneralizedLinearModel properties: <ul style="list-style-type: none"> <li>• Diagnostics</li> <li>• Fitted</li> <li>• Offset</li> <li>• ObservationInfo</li> <li>• ObservationNames</li> <li>• Residuals</li> <li>• Steps</li> </ul> </li> </ul>

Function	Notes or Limitations
	<ul style="list-style-type: none"><li data-bbox="832 302 1018 326">• Variables</li></ul> <p data-bbox="832 354 1314 447">You can compute the residuals directly from the compact object returned by <code>GLM = fitglm(X, Y)</code> using</p> <pre data-bbox="832 475 1348 557">RES = Y - predict(GLM, X); S = sqrt(GLM.SSE/GLM.DFE); histogram(RES, linspace(-3*S, 3*S, 51))</pre>

Function	Notes or Limitations
lasso	<ul style="list-style-type: none"> <li>• With tall arrays, <code>lasso</code> uses an algorithm based on ADMM (alternating direction method of multipliers).</li> <li>• No elastic-net support. The 'Alpha' parameter is always 1.</li> <li>• No cross-validation ('CV' parameter) support, which includes the related parameter 'MCReps'.</li> <li>• The second output <code>FitInfo</code> does not contain the additional fields: 'SE', 'LambdaMinMSE', 'Lambda1SE', 'IndexMinMSE', and 'Index1SE'.</li> <li>• The 'Options' parameter is not supported, since it does not contain options that apply to the ADMM algorithm. You can tune the ADMM algorithm using name-value pairs.</li> <li>• Supported name-value pairs are: <ul style="list-style-type: none"> <li>• 'Lambda'</li> <li>• 'LambdaRatio'</li> <li>• 'NumLambda'</li> <li>• 'Standardize'</li> <li>• 'PredictorNames'</li> <li>• 'RelTol'</li> <li>• 'Weights'</li> </ul> </li> <li>• Additional name-value pairs to control the ADMM algorithm are: <ul style="list-style-type: none"> <li>• 'Rho' — Augmented Lagrangian parameter, <math>\rho</math>. Default value is automatic selection.</li> </ul> </li> </ul>

Function	Notes or Limitations
	<ul style="list-style-type: none"><li data-bbox="838 300 1273 392">• 'AbsTol' — Absolute tolerance used to determine convergence. Default value is <math>1e-4</math>.</li><li data-bbox="838 409 1322 466">• 'MaxIter' — Maximum number of iterations. Default value is <math>1e4</math>.</li><li data-bbox="838 487 1262 579">• 'E0' — Initial values for the coefficients <math>x</math>. Default value is a vector of zeros.</li><li data-bbox="838 597 1322 689">• 'U0' — Initial values of scaled dual variable <math>u</math>. Default value is a vector of zeros.</li></ul>

Function	Notes or Limitations
fitrlinear	<ul style="list-style-type: none"> <li>• Supported syntaxes for tall arrays X and Y are: <ul style="list-style-type: none"> <li>• <code>obj = fitrlinear(X,Y)</code></li> <li>• <code>obj = fitrlinear(X,Y,Name,Value)</code></li> </ul> </li> <li>• Some name-value pairs have different defaults and values compared to the in-memory <code>fitrlinear</code> function. Supported name-value pairs, and any differences, are: <ul style="list-style-type: none"> <li>• 'Epsilon'</li> <li>• 'ObservationsIn' — Supports only 'rows'.</li> <li>• 'Lambda' — Can be 'auto' (default) or a scalar.</li> <li>• 'Learner'</li> <li>• 'Regularization' — Supports only 'ridge'.</li> <li>• 'Solver' — Supports only 'lbfgs'.</li> <li>• 'Verbose' — Default value is 1</li> <li>• 'Beta'</li> <li>• 'Bias'</li> <li>• 'FitBias' — Supports only true.</li> <li>• 'Weights' — Value must be a tall array.</li> <li>• 'HessianHistorySize'</li> <li>• 'BetaTolerance' — Default value is relaxed to 1e-3.</li> <li>• 'GradientTolerance' — Default value is relaxed to 1e-3.</li> </ul> </li> </ul>

Function	Notes or Limitations
	<ul style="list-style-type: none"> <li>• 'IterationLimit' — Default value is relaxed to 20.</li> <li>• For tall arrays <code>fitrlinear</code> implements LBFGS by distributing the calculation of the loss and the gradient among different parts of the tall array at each iteration. Other solvers are not available for tall arrays.</li> </ul> <p>When initial values for <code>Beta</code> and <code>Bias</code> are not given, <code>fitrlinear</code> first refines the initial estimates of the parameters by fitting the model locally to parts of the data and combining the coefficients by averaging.</p>

## Classification

Function	Notes or Limitations
<p>The <code>predict</code>, <code>loss</code>, <code>margin</code>, and <code>edge</code> methods of these classification classes support tall arrays:</p> <ul style="list-style-type: none"> <li>• <code>CompactClassificationEnsemble</code></li> <li>• <code>CompactClassificationTree</code></li> <li>• <code>CompactClassificationDiscriminant</code></li> <li>• <code>CompactClassificationNaiveBayes</code></li> <li>• <code>CompactClassificationSVM</code></li> <li>• <code>CompactClassificationECOC</code></li> <li>• <code>ClassificationKernel</code></li> <li>• <code>ClassificationKNN</code></li> <li>• <code>ClassificationLinear</code></li> </ul>	<ul style="list-style-type: none"> <li>• You can use models trained on either in-memory or tall data with these methods.</li> <li>• The <code>loss</code> method of <code>CompactClassificationTree</code> only supports one output argument.</li> </ul>



Function	Notes or Limitations
The <code>resume</code> method of <code>ClassificationKernel</code> supports tall arrays.	<ul style="list-style-type: none"><li>• You can use models trained on either in-memory or tall data.</li><li>• The <code>'IterationLimit'</code> name-value pair argument has a different default compared to the in-memory <code>resume</code> function. The default value is relaxed to 20.</li><li>• <code>resume</code> uses a block-wise strategy. For details, see “Algorithms” on page 32-7094 of <code>fitckernel</code>.</li></ul>

Function	Notes or Limitations
fitcdiscr	<ul style="list-style-type: none"> <li>• Supported name-value pairs are: <ul style="list-style-type: none"> <li>• 'ClassNames'</li> <li>• 'Cost'</li> <li>• 'DiscrimType'</li> <li>• 'PredictorNames'</li> <li>• 'Prior'</li> <li>• 'ResponseName'</li> <li>• 'ScoreTransform'</li> <li>• 'Weights'</li> </ul> </li> <li>• For tall arrays and tall tables, fitcdiscr returns a CompactClassificationDiscriminant object, which contains most of the same properties as a ClassificationDiscriminant object. The main difference is that the compact object is sensitive to memory requirements. The compact object does not include properties that include the data, or that include an array of the same size as the data. The compact object does not contain these ClassificationDiscriminant properties: <ul style="list-style-type: none"> <li>• ModelParameters</li> <li>• NumObservations</li> <li>• ParameterOptimizationResults</li> <li>• RowsUsed</li> <li>• XCentered</li> <li>• W</li> <li>• X</li> <li>• Y</li> </ul> </li> </ul>

Function	Notes or Limitations
	<p>Additionally, the compact object does not support these ClassificationDiscriminant methods:</p> <ul style="list-style-type: none"><li>• compact</li><li>• crossval</li><li>• cvshrink</li><li>• resubEdge</li><li>• resubLoss</li><li>• resubMargin</li><li>• resubPredict</li></ul>

Function	Notes or Limitations
fitckernel	<ul style="list-style-type: none"><li>• Some name-value pairs have different defaults compared to the in-memory fitckernel function.<ul style="list-style-type: none"><li>• 'Verbose' — Default value is 1.</li><li>• 'BetaTolerance' — Default value is relaxed to 1e-3.</li><li>• 'GradientTolerance' — Default value is relaxed to 1e-5.</li><li>• 'IterationLimit' — Default value is relaxed to 20.</li></ul></li><li>• If 'KernelScale' is 'auto', then fitckernel uses the random stream controlled by tallrng for subsampling. For reproducibility, you must set a random number seed for both the global stream and the random stream controlled by tallrng.</li><li>• If 'Lambda' is 'auto', then fitckernel might take an extra pass through the data to calculate the number of observations in X.</li><li>• fitckernel uses a block-wise strategy. For details, see “Algorithms” on page 32-7094.</li></ul>

Function	Notes or Limitations
fitclinear	<ul style="list-style-type: none"> <li>• Supported syntaxes for tall arrays X and Y are: <ul style="list-style-type: none"> <li>• <code>obj = fitclinear(X,Y)</code></li> <li>• <code>obj = fitclinear(___,Name,Value)</code></li> </ul> </li> <li>• Some name-value pairs have different defaults compared to the in-memory fitclinear function. Supported name-value pairs, and any differences, are: <ul style="list-style-type: none"> <li>• 'ObservationsIn' — Supports only 'rows'.</li> <li>• 'Lambda' — Can be 'auto' (default) or a scalar.</li> <li>• 'Learner'</li> <li>• 'Regularization' — Supports only 'ridge'.</li> <li>• 'Solver' — Supports only 'lbfgs'.</li> <li>• 'FitBias' — Supports only true.</li> <li>• 'Verbose' — Default value is 1.</li> <li>• 'Beta'</li> <li>• 'Bias'</li> <li>• 'ClassNames'</li> <li>• 'Cost'</li> <li>• 'Prior'</li> <li>• 'Weights' — Value must be a tall array.</li> <li>• 'HessianHistorySize'</li> <li>• 'BetaTolerance' — Default value is relaxed to 1e-3.</li> </ul> </li> </ul>

Function	Notes or Limitations
	<ul style="list-style-type: none"><li>• 'GradientTolerance' — Default value is relaxed to 1e-3.</li><li>• 'IterationLimit' — Default value is relaxed to 20.</li><li>• For tall arrays, <code>fitclinear</code> implements LBFGS by distributing the calculation of the loss and gradient among different parts of the tall array at each iteration. Other solvers are not available for tall arrays.</li></ul> <p>When initial values for <code>Beta</code> and <code>Bias</code> are not given, <code>fitclinear</code> refines the initial estimates of the parameters by fitting the model locally to parts of the data and combining the coefficients by averaging.</p>

Function	Notes or Limitations
fitcnb	<ul style="list-style-type: none"><li>• Supported syntaxes are:<ul style="list-style-type: none"><li>• <code>discr = fitcnb(Tbl,Y)</code></li><li>• <code>discr = fitcnb(X,Y)</code></li><li>• <code>discr = fitcnb(___,Name,Value)</code></li></ul></li><li>• Options related to kernel densities, cross-validation, and hyperparameter optimization are not supported. The supported name-value pairs are:<ul style="list-style-type: none"><li>• <code>'DistributionNames'</code> — <code>'kernel'</code> value is not supported.</li><li>• <code>'CategoricalPredictors'</code></li><li>• <code>'PredictorNames'</code></li><li>• <code>'ResponseName'</code></li><li>• <code>'ScoreTransform'</code></li><li>• <code>'Weights'</code> — Value must be a tall array.</li></ul></li></ul>

Function	Notes or Limitations
fitctree	<ul style="list-style-type: none"> <li>• Supported syntaxes for tall arrays are: <ul style="list-style-type: none"> <li>• <code>tree = fitctree(Tbl,Y)</code></li> <li>• <code>tree = fitctree(X,Y)</code></li> <li>• <code>tree = fitctree(___,Name,Value)</code></li> </ul> </li> <li>• Supported name-value pairs are: <ul style="list-style-type: none"> <li>• 'AlgorithmForCategorical'</li> <li>• 'CategoricalPredictors'</li> <li>• 'ClassNames'</li> <li>• 'MaxNumCategories'</li> <li>• 'MaxNumSplits'</li> <li>• 'MergeLeaves'</li> <li>• 'MinLeafSize'</li> <li>• 'MinParentSize'</li> <li>• 'NumVariablesToSample'</li> <li>• 'PredictorNames'</li> <li>• 'ResponseName'</li> <li>• 'ScoreTransform'</li> <li>• 'SplitCriterion'</li> <li>• 'Weights'</li> </ul> </li> <li>• There is an additional name-value pair specific to tall arrays: <ul style="list-style-type: none"> <li>• 'MaxDepth' — A positive integer specifying the maximum depth of the output tree. Specify a value for this parameter to return a tree with fewer levels that requires fewer passes through the tall array to compute. Generally the algorithm of <code>fitctree</code> takes one pass through the data and an additional pass for</li> </ul> </li> </ul>



Function	Notes or Limitations
	each tree level. There is no maximum tree depth by default.

Function	Notes or Limitations
TreeBagger	<ul style="list-style-type: none"> <li>• Supported syntaxes for tall X, Y, Tbl are: <ul style="list-style-type: none"> <li>• B = TreeBagger (NumTrees, Tbl, Y)</li> <li>• B = TreeBagger (NumTrees, X, Y)</li> <li>• B = TreeBagger (___, Name, Value)</li> </ul> </li> <li>• For tall arrays, TreeBagger supports classification. Regression is not supported.</li> <li>• Supported name-value pairs are: <ul style="list-style-type: none"> <li>• 'NumPredictorsToSample' — Default value is the square root of the number of variables for classification.</li> <li>• 'MinLeafSize' — Default value is 1 if the number of observations is less than 50,000. If the number of observations is larger than 50,000, then the default value is <math>\max(1, \min(5, \text{floor}(0.01 * \text{NobsChunk})))</math>.</li> <li>• 'ChunkSize' (only for tall arrays) — Default value is 50000.</li> </ul> </li> </ul> <p>In addition, TreeBagger supports these optional arguments of fitctree:</p> <ul style="list-style-type: none"> <li>• 'AlgorithmForCategorical'</li> <li>• 'CategoricalPredictors'</li> <li>• 'MaxNumCategories'</li> <li>• 'MergeLeaves'</li> <li>• 'PredictorNames'</li> </ul>

Function	Notes or Limitations
	<ul style="list-style-type: none"> <li>• 'PredictorSelection'</li> <li>• 'Prune'</li> <li>• 'PruneCriterion'</li> <li>• 'Surrogate'</li> <li>• 'MaxNumSplits'</li> <li>• 'SplitCriterion'</li> </ul> <ul style="list-style-type: none"> <li>• For tall data, <code>TreeBagger</code> returns a <code>CompactTreeBagger</code> object that contains most of the same properties as a full <code>TreeBagger</code> object. The main difference is that the compact object is more memory efficient. The compact object does not include properties that include the data, or that include an array of the same size as the data.</li> <li>• Supported <code>CompactTreeBagger</code> methods are: <ul style="list-style-type: none"> <li>• <code>combine</code></li> <li>• <code>error</code></li> <li>• <code>margin</code></li> <li>• <code>meanMargin</code></li> <li>• <code>predict</code></li> <li>• <code>setDefaultYfit</code></li> </ul> <p>The <code>error</code>, <code>margin</code>, <code>meanMargin</code>, and <code>predict</code> methods do not support the name-value pairs <code>'Trees'</code>, <code>'TreeWeights'</code>, or <code>'UseInstanceForTree'</code>. The <code>error</code> and <code>meanMargin</code> methods additionally do not support <code>'Weights'</code>.</p> </li> <li>• <code>TreeBagger</code> creates a random forest by generating trees on disjoint chunks of</li> </ul>

Function	Notes or Limitations
	<p>the data. When more data is available than is required to create the random forest, the data is subsampled. For a similar example, see Random Forests for Big Data (Genuer, Poggi, Tuleau-Malot, Villa-Vialaneix 2015).</p> <p>Depending on how the data is stored, it is possible that some chunks of data contain observations from only a few classes out of all the classes. In this case, <code>TreeBagger</code> might produce inferior results compared to the case where each chunk of data contains observations from most of the classes.</p>

## Dimensionality Reduction

Function	Notes or Limitations
<code>pcacov</code> , <code>factoran</code>	<p><code>pcacov</code> and <code>factoran</code> do <i>not</i> work directly on tall arrays. Instead, use <code>C = gather(cov(X))</code> to compute the covariance matrix of a tall array. Then, you can use <code>pcacov</code> or <code>factoran</code> on the in-memory covariance matrix. Alternatively, you can use <code>pca</code> directly on a tall array.</p>

Function	Notes or Limitations
pca	<ul style="list-style-type: none"> <li>• <code>pca</code> works directly with tall arrays by computing the covariance matrix and using the in-memory <code>pcacov</code> function to compute the principle components.</li> <li>• Supported syntaxes are: <ul style="list-style-type: none"> <li>• <code>coeff = pca(X)</code></li> <li>• <code>[coeff, score, latent] = pca(X)</code></li> <li>• <code>[coeff, score, latent, explained] = pca(X)</code></li> <li>• <code>[coeff, score, latent, tsquared] = pca(X)</code></li> <li>• <code>[coeff, score, latent, tsquared, explained] = pca(X)</code></li> </ul> </li> <li>• Name-value pair arguments are not supported.</li> </ul>

## See Also

`datastore` | `tall`

## More About

- “Tall Arrays” (MATLAB)
- “Functions That Support Tall Arrays (A–Z)” (MATLAB)

## Logistic Regression with Tall Arrays

This example shows how to use logistic regression and other techniques to perform data analysis on tall arrays. Tall arrays represent data that is too large to fit into computer memory.

### Get Data into MATLAB®

Create a datastore that references the folder location with the data. The data can be contained in a single file, a collection of files, or an entire folder. Treat 'NA' values as missing data so that `datastore` replaces them with NaN values. Select a subset of the variables to work with, and include the name of the airline (`UniqueCarrier`) as a categorical variable. Create a tall table on top of the datastore.

```
ds = datastore('airlinesmall.csv');
ds.TreatAsMissing = 'NA';
ds.SelectedVariableNames = {'DayOfWeek', 'UniqueCarrier', ...
    'ArrDelay', 'DepDelay', 'Distance'};
ds.SelectedFormats{2} = '%C';
tt = tall(ds);
tt.DayOfWeek = categorical(tt.DayOfWeek, 1:7, ...
    {'Sun', 'Mon', 'Tues', 'Wed', 'Thu', 'Fri', 'Sat'}, 'Ordinal', true)
```

Starting parallel pool (parpool) using the 'local' profile ...  
connected to 12 workers.

```
tt =
```

Mx5 tall table

DayOfWeek	UniqueCarrier	ArrDelay	DepDelay	Distance
Tues	PS	8	12	308
Sun	PS	8	1	296
Thu	PS	21	20	480
Thu	PS	13	12	296
Wed	PS	4	-1	373
Tues	PS	59	63	308
Wed	PS	3	-2	447
Fri	PS	11	-1	954
:	:	:	:	:
:	:	:	:	:

## Late Flights

Determine the flights that are late by 20 minutes or more by defining a logical variable that is true for a late flight. Add this variable to the tall table of data, noting that it is not yet evaluated. A preview of this variable includes the first few rows.

```
tt.LateFlight = tt.ArrDelay>=20
```

```
tt =
```

```
Mx6 tall table
```

DayOfWeek	UniqueCarrier	ArrDelay	DepDelay	Distance	LateFlight
Tues	PS	8	12	308	false
Sun	PS	8	1	296	false
Thu	PS	21	20	480	true
Thu	PS	13	12	296	false
Wed	PS	4	-1	373	false
Tues	PS	59	63	308	true
Wed	PS	3	-2	447	false
Fri	PS	11	-1	954	false
:	:	:	:	:	:
:	:	:	:	:	:

Calculate the mean of `LateFlight` to determine the overall proportion of late flights. Use `gather` to trigger evaluation of the tall array and bring the result into memory.

```
m = mean(tt.LateFlight)
```

```
m =
```

```
tall double
```

```
?
```

```
m = gather(m)
```

```
Evaluating tall expression using the Parallel Pool 'local':
- Pass 1 of 1: Completed in 10 sec
```

```
Evaluation completed in 14 sec
```

```
m =
```

```
0.1580
```

### Late Flights by Carrier

Examine whether certain types of flights tend to be late. First, check to see if certain carriers are more likely to have late flights.

```
tt.LateFlight = double(tt.LateFlight);
late_by_carrier = gather(grpstats(tt, 'UniqueCarrier', 'mean', 'DataVar', 'LateFlight'))
```

```
Evaluating tall expression using the Parallel Pool 'local':
```

```
- Pass 1 of 1: Completed in 5 sec
```

```
Evaluation completed in 21 sec
```

```
late_by_carrier =
```

```
29x4 table
```

GroupLabel	UniqueCarrier	GroupCount	mean_LateFlight
'B6'	B6	806	0.23821
'DL'	DL	16578	0.15261
'US'	US	13997	0.15296
'F9'	F9	335	0.18209
'OH'	OH	1457	0.18874
'PA (1)'	PA (1)	318	0.16981
'PS'	PS	83	0.13253
'YV'	YV	849	0.19081
'AQ'	AQ	154	0.051948
'CO'	CO	8138	0.16319
'EA'	EA	920	0.15217
'FL'	FL	1263	0.19952
'MQ'	MQ	3962	0.18778
'OO'	OO	3090	0.13916
'NW'	NW	10349	0.14542
'PI'	PI	871	0.1814
'WN'	WN	15931	0.13722
'AS'	AS	2910	0.16014
'EV'	EV	1699	0.21248



'9E'	9E	521	0.13436
'TZ'	TZ	216	0.125
'UA'	UA	13286	0.17447
'XE'	XE	2357	0.17947
'DH'	DH	696	0.17672
'HP'	HP	3660	0.13907
'ML (1)'	ML (1)	69	0.043478
'TW'	TW	3805	0.159
'AA'	AA	14930	0.16236
'HA'	HA	273	0.047619

Carriers B6 and EV have higher proportions of late flights. Carriers AQ, ML (1), and HA have relatively few flights, but lower proportions of them are late.

### Late Flights by Day of Week

Next, check to see if different days of the week tend to have later flights.

```
late_by_day = gather(grpstats(tt, 'DayOfWeek', 'mean', 'DataVar', 'LateFlight'))
```

```
Evaluating tall expression using the Parallel Pool 'local':
- Pass 1 of 1: Completed in 3 sec
Evaluation completed in 8 sec
```

```
late_by_day =
```

7x4 table

GroupLabel	DayOfWeek	GroupCount	mean_LateFlight
'Wed'	Wed	18240	0.18399
'Tues'	Tues	18163	0.15526
'Fri'	Fri	15839	0.12899
'Sat'	Sat	16958	0.15603
'Mon'	Mon	18077	0.14234
'Sun'	Sun	18019	0.15117
'Thu'	Thu	18227	0.18418

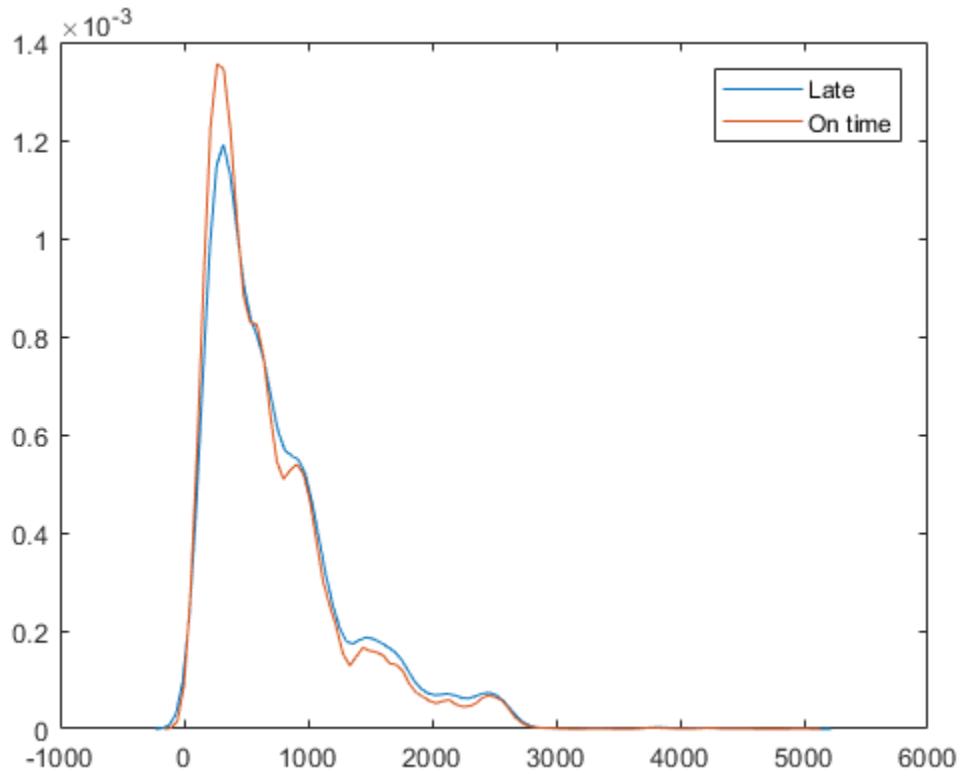
Wednesdays and Thursdays have the highest proportion of late flights, while Fridays have the lowest proportion.

### Late Flights by Distance

Check to see if longer or shorter flights tend to be late. First, look at the density of the flight distance for flights that are late, and compare that with flights that are on time.

```
ksdensity(tt.Distance(tt.LateFlight==1))
hold on
ksdensity(tt.Distance(tt.LateFlight==0))
hold off
legend('Late','On time')
```

```
Evaluating tall expression using the Parallel Pool 'local':
- Pass 1 of 2: Completed in 3 sec
- Pass 2 of 2: Completed in 3 sec
Evaluation completed in 8 sec
Evaluating tall expression using the Parallel Pool 'local':
- Pass 1 of 2: Completed in 3 sec
- Pass 2 of 2: Completed in 2 sec
Evaluation completed in 7 sec
```



Flight distance does not make a dramatic difference in whether a flight is early or late. However, the density appears to be slightly higher for on-time flights at distances of about 400 miles. The density is also higher for late flights at distances of about 2000 miles. Calculate some simple descriptive statistics for the late and on-time flights.

```
late_by_distance = gather(grpstats(tt, 'LateFlight', {'mean' 'std'}, 'DataVar', 'Distance'))
```

```
Evaluating tall expression using the Parallel Pool 'local':
```

```
- Pass 1 of 1: Completed in 3 sec
```

```
Evaluation completed in 9 sec
```

```
late_by_distance =
```

```
2x5 table
```

GroupLabel	LateFlight	GroupCount	mean_Distance	std_Distance
'0'	0	1.04e+05	693.14	544.75
'1'	1	19519	750.24	574.12

Late flights are about 60 miles longer on average, although this value makes up only a small portion of the standard deviation of the distance values.

### Logistic Regression Model

Build a model for the probability of a late flight, using both continuous variables (such as Distance) and categorical variables (such as DayOfWeek) to predict the probabilities. This model can help to determine if the previous results observed for each predictor individually also hold true when you consider them together.

```
glm = fitglm(tt, 'LateFlight~Distance+DayOfWeek', 'Distribution', 'binomial')
```

```
Iteration [1]:      0% completed
Iteration [1]:     100% completed
Iteration [2]:      0% completed
Iteration [2]:     100% completed
Iteration [3]:      0% completed
Iteration [3]:     100% completed
Iteration [4]:      0% completed
Iteration [4]:     100% completed
Iteration [5]:      0% completed
Iteration [5]:     100% completed
```

```
glm =
```

```
Compact generalized linear regression model:
logit(LateFlight) ~ 1 + DayOfWeek + Distance
Distribution = Binomial
```

```
Estimated Coefficients:
```

	Estimate	SE	tStat	pValue
(Intercept)	-1.855	0.023052	-80.469	0
DayOfWeek_Mon	-0.072603	0.029798	-2.4365	0.01483

DayOfWeek_Tues	0.026909	0.029239	0.92029	0.35742
DayOfWeek_Wed	0.2359	0.028276	8.343	7.2452e-17
DayOfWeek_Thu	0.23569	0.028282	8.3338	7.8286e-17
DayOfWeek_Fri	-0.19285	0.031583	-6.106	1.0213e-09
DayOfWeek_Sat	0.033542	0.029702	1.1293	0.25879
Distance	0.00018373	1.3507e-05	13.602	3.8741e-42

123319 observations, 123311 error degrees of freedom

Dispersion: 1

Chi<sup>2</sup>-statistic vs. constant model: 504, p-value = 8.74e-105

The model confirms that the previously observed conclusions hold true here as well:

- The Wednesday and Thursday coefficients are positive, indicating a higher probability of a late flight on those days. The Friday coefficient is negative, indicating a lower probability.
- The Distance coefficient is positive, indicating that longer flights have a higher probability of being late.

All of these coefficients have very small p-values. This is common with data sets that have many observations, since one can reliably estimate small effects with large amounts of data. In fact, the uncertainty in the model is larger than the uncertainty in the estimates for the parameters in the model.

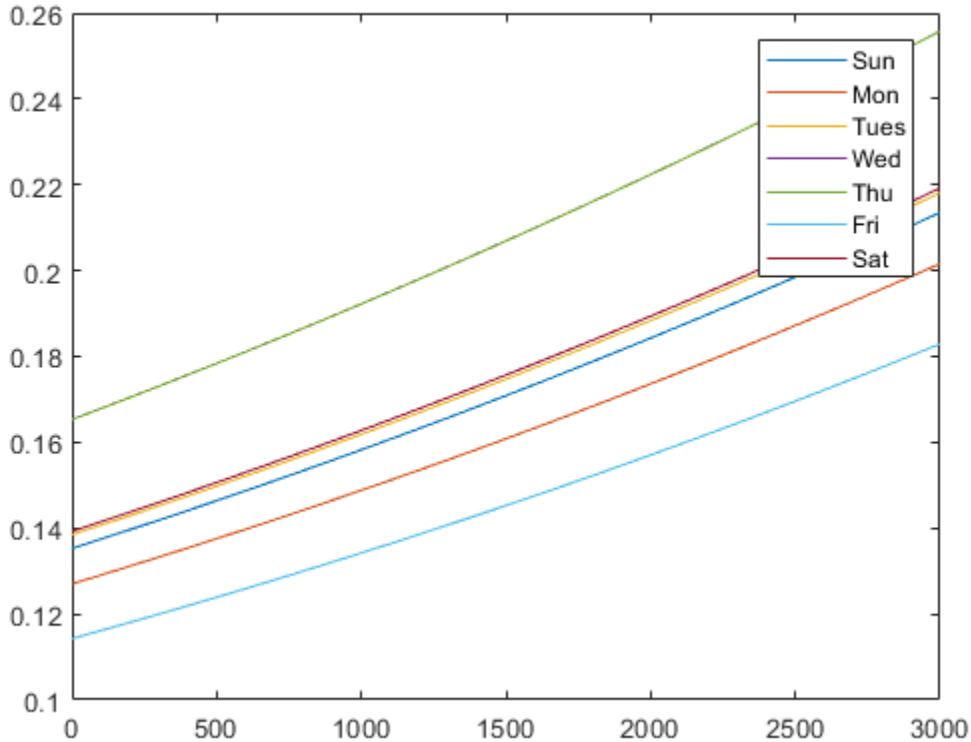
### Prediction with Model

Predict the probability of a late flight for each day of the week, and for distances ranging from 0 to 3000 miles. Create a table to hold the predictor values by indexing the first 100 rows in the original table `tt`.

```
x = gather(tt(1:100, {'Distance' 'DayOfWeek'}));
x.Distance = linspace(0,3000)';
x.DayOfWeek(:) = 'Sun';
plot(x.Distance,predict(glm,x));

days = {'Sun' 'Mon' 'Tues' 'Wed' 'Thu' 'Fri' 'Sat'};
hold on
for j=2:length(days)
    x.DayOfWeek(:) = days{j};
    plot(x.Distance,predict(glm,x));
end
legend(days)
```

```
Evaluating tall expression using the Parallel Pool 'local':
- Pass 1 of 1: Completed in 1 sec
Evaluation completed in 1 sec
```



According to this model, a Wednesday or Thursday flight of 500 miles has the same probability of being late, about 18%, as a Friday flight of about 3000 miles.

Since these probabilities are all much less than 50%, the model is unlikely to predict that any given flight will be late using this information. Investigate the model more by focusing on the flights for which the model predicts a probability of 20% or more of being late, and compare that to the actual results.

```
C = gather(crosstab(tt.LateFlight, predict(glm, tt) > .20))
```

```
Evaluating tall expression using the Parallel Pool 'local':  
- Pass 1 of 1: Completed in 4 sec  
Evaluation completed in 5 sec
```

C =

99613	4391
18394	1125

Among the flights predicted to have a 20% or higher probability of being late, about 20% were late  $1125 / (1125 + 4391)$ . Among the remainder, less than 16% were late  $18394 / (18394 + 99613)$ .





# Parallel Statistics

---

- “Quick Start Parallel Computing for Statistics and Machine Learning Toolbox” on page 30-2
- “Use Parallel Processing for Regression TreeBagger Workflow” on page 30-6
- “Concepts of Parallel Computing in Statistics and Machine Learning Toolbox” on page 30-8
- “When to Run Statistical Functions in Parallel” on page 30-9
- “Working with parfor” on page 30-11
- “Reproducibility in Parallel Statistical Computations” on page 30-14
- “Implement Jackknife Using Parallel Computing” on page 30-19
- “Implement Cross-Validation Using Parallel Computing” on page 30-21
- “Implement Bootstrap Using Parallel Computing” on page 30-24

## Quick Start Parallel Computing for Statistics and Machine Learning Toolbox

**Note** To use parallel computing as described in this chapter, you must have a Parallel Computing Toolbox license.

---

<b>In this section...</b>
---------------------------

“What Is Parallel Statistics Functionality?” on page 30-2
---

“How To Compute in Parallel” on page 30-4
---

### What Is Parallel Statistics Functionality?

You can use any of the Statistics and Machine Learning Toolbox functions with Parallel Computing Toolbox constructs such as `parfor` and `spmd`. However, some functions, such as those with interactive displays, can lose functionality in parallel. In particular, displays and interactive usage are not effective on workers (see “Vocabulary for Parallel Computation” on page 30-8).

Additionally, the following functions are enhanced to use parallel computing internally. These functions use `parfor` internally to parallelize calculations.

- `bootci`
- `bootstrp`
- `candexch`
- `cordexch`
- `crossval`
- `daugment`
- `dcovary`
- `jackknife`
- `kmeans`
- `lasso`
- `lassoglm`

- `nnmf`
- `plsregress`
- `rowexch`
- `sequentialfs`
- `TreeBagger`
- `TreeBagger.growTrees`

The following functions for fitting multiclass models for support vector machines and other classifiers are also enhanced to use parallel computing internally.

- `fitcecoc`
- **Methods of the class `ClassificationECOC`:**
  - `resubEdge`
  - `resubLoss`
  - `resubMargin`
  - `resubPredict`
  - `crossval`
- **Methods of the class `CompactClassificationECOC`**
  - `edge`
  - `loss`
  - `margin`
  - `predict`
- **Methods of the class `ClassificationPartitionedECOC`**
  - `kfoldEdge`
  - `kfoldLoss`
  - `kfoldMargin`
  - `kfoldPredict`

This chapter gives the simplest way to use these enhanced functions in parallel. For more advanced topics, including the issues of reproducibility and nested `parfor` loops, see the other sections in this chapter.

For information on parallel statistical computing at the command line, enter

```
help parallelstats
```

## How To Compute in Parallel

To have a function compute in parallel:

1. “Set Up a Parallel Environment” on page 30-4
2. “Set the UseParallel Option to true” on page 30-4
3. “Call the Function Using the Options Structure” on page 30-4

### Set Up a Parallel Environment

To run a statistical computation in parallel, first set up a parallel environment.

---

**Note** Setting up a parallel environment can take several seconds.

---

For a multicore machine, enter the following at the MATLAB command line:

```
parpool(n)
```

$n$  is the number of workers you want to use.

### Set the UseParallel Option to true

Create an options structure with the `statset` function. To run in parallel, set the `UseParallel` option to true:

```
paroptions = statset('UseParallel',true);
```

### Call the Function Using the Options Structure

Call your function with syntax that uses the options structure. For example:

```
% Run crossval in parallel
cvMse = crossval('mse',x,y,'predfun',regf,'Options',paroptions);
```

```
% Run bootstrp in parallel
sts = bootstrp(100,@(x)[mean(x) std(x)],y,'Options',paroptions);
```

```
% Run TreeBagger in parallel
b = TreeBagger(50,meas,spec,'OOBPred','on','Options',paroptions);
```

For more complete examples of parallel statistical functions, see “Use Parallel Processing for Regression TreeBagger Workflow” on page 30-6, “Implement Jackknife Using Parallel Computing” on page 30-19, “Implement Cross-Validation Using Parallel Computing” on page 30-21, and “Implement Bootstrap Using Parallel Computing” on page 30-24.

After you have finished computing in parallel, close the parallel environment:

```
delete mypool
```

---

**Tip** To save time, keep the pool open if you expect to compute in parallel again soon.

---

## Use Parallel Processing for Regression TreeBagger Workflow

This example shows you how to:

- Use an ensemble of bagged regression trees to estimate feature importance.
- Improve computation speed by using parallel computing.

The sample data is a database of 1985 car imports with 205 observations, 25 predictors, and 1 response, which is insurance risk rating, or "symboling." The first 15 variables are numeric and the last 10 are categorical. The symboling index takes integer values from -3 to 3.

Load the sample data and separate it into predictor and response arrays.

```
load imports-85;  
Y = X(:,1);  
X = X(:,2:end);
```

Set up the parallel environment to use two cores.

```
mypool = parpool(2)
```

```
Starting parallel pool (parpool) using the 'local' profile ...  
connected to 2 workers.
```

```
mypool =
```

```
Pool with properties:
```

```
    Connected: true  
    NumWorkers: 2  
    Cluster: local  
    AttachedFiles: {}  
    AutoAddClientPath: true  
    IdleTimeout: 30 minutes (30 minutes remaining)  
    SpmdEnabled: true
```

Set the options to use parallel processing.

```
paroptions = statset('UseParallel',true);
```

Estimate feature importance using leaf size 1 and 5000 trees in parallel. Time the function for comparison purposes.

```
tic
b = TreeBagger(5000,X,Y,'Method','r','OOBVarImp','on', ...
'cat',16:25,'MinLeafSize',1,'Options',paroptions);
toc
```

Elapsed time is 54.082494 seconds.

Perform the same computation in serial for timing comparison.

```
tic
b = TreeBagger(5000,X,Y,'Method','r','OOBVarImp','on', ...
'cat',16:25,'MinLeafSize',1);
toc
```

Elapsed time is 124.101529 seconds.

The results show that computing in parallel takes a fraction of the time it takes to compute serially. Note that the elapsed time can vary depending on your operating system.

## See Also

TreeBagger | parpool | statset

## Related Examples

- “Regression of Insurance Risk Rating for Car Imports Using TreeBagger” on page 18-141
- “Classifying Radar Returns for Ionosphere Data Using TreeBagger” on page 18-154
- “Comparison of TreeBagger and Bagged Ensembles” on page 18-55

## Concepts of Parallel Computing in Statistics and Machine Learning Toolbox

In this section...
“Subtleties in Parallel Computing” on page 30-8
“Vocabulary for Parallel Computation” on page 30-8

### Subtleties in Parallel Computing

There are two main subtleties in parallel computations:

- Nested parallel evaluations (see “No Nested `parfor` Loops” on page 30-12). Only the outermost `parfor` loop runs in parallel, the others run serially.
- Reproducible results when using random numbers (see “Reproducibility in Parallel Statistical Computations” on page 30-14). How can you get exactly the same results when repeatedly running a parallel computation that uses random numbers?

### Vocabulary for Parallel Computation

- `worker` — An independent MATLAB session that runs code distributed by the client.
- `client` — The MATLAB session with which you interact, and that distributes jobs to workers.
- `parfor` — A Parallel Computing Toolbox function that distributes independent code segments to workers (see “Working with `parfor`” on page 30-11).
- `random stream` — A pseudorandom number generator, and the sequence of values it generates. MATLAB implements random streams with the `RandStream` class.
- `reproducible computation` — A computation that can be exactly replicated, even in the presence of random numbers (see “Reproducibility in Parallel Statistical Computations” on page 30-14).



## When to Run Statistical Functions in Parallel

In this section...
“Why Run in Parallel?” on page 30-9
“Factors Affecting Speed” on page 30-9
“Factors Affecting Results” on page 30-10

### Why Run in Parallel?

The main reason to run statistical computations in parallel is to gain speed, meaning to reduce the execution time of your program or functions. “Factors Affecting Speed” on page 30-9 discusses the main items affecting the speed of programs or functions.

“Factors Affecting Results” on page 30-10 discusses details that can cause a parallel run to give different results than a serial run.

---

**Note** Some Statistics and Machine Learning Toolbox functions have built-in parallel computing capabilities. See Quick Start Parallel Computing for Statistics and Machine Learning Toolbox on page 30-2. You can also use any Statistics and Machine Learning Toolbox functions with Parallel Computing Toolbox functions such as `parfor` loops. To decide when to call functions in parallel, consider the factors affecting speed and results.

---

### Factors Affecting Speed

Some factors that can affect the speed of execution of parallel processing are:

- Parallel environment setup. It takes time to run `parpool` to begin computing in parallel. If your computation is fast, the setup time can exceed any time saved by computing in parallel.
- Parallel overhead. There is overhead in communication and coordination when running in parallel. If function evaluations are fast, this overhead could be an appreciable part of the total computation time. Thus, solving a problem in parallel can be slower than solving the problem serially. For an example, see Improving Optimization Performance with Parallel Computing in MATLAB Digest, March 2009.
- No nested `parfor` loops. This is described in “Working with `parfor`” on page 30-11. `parfor` does not work in parallel when called from within another `parfor` loop. If you have programmed your custom functions to take advantage of parallel processing,

the limitation of no nested `parfor` loops can cause a parallel function to run slower than expected.

- When executing serially, `parfor` loops run slightly slower than `for` loops.
- Passing parameters. Parameters are automatically passed to worker sessions during the execution of parallel computations. If there are many parameters, or they take a large amount of memory, passing parameters can slow the execution of your computation.
- Contention for resources: network and computing. If the pool of workers has low bandwidth or high latency, parallel computation can be slow.

## Factors Affecting Results

Some factors can affect results when using parallel processing. You might need to adjust your code to run in parallel, for example, you need independent loops and the workers must be able to access the variables. Some important factors are:

- Persistent or global variables. If any functions use persistent or global variables, these variables can take different values on different worker processors. The body of a `parfor` loop cannot contain global or persistent variable declarations.
- Accessing external files. The order of computations is not guaranteed during parallel processing, so external files can be accessed in unpredictable order, leading to unpredictable results. Furthermore, if multiple processors try to read an external file simultaneously, the file can become locked, leading to a read error, and halting function execution.
- Noncomputational functions, such as `input`, `plot`, and `keyboard`, can behave badly when used in your custom functions. Do not use these functions in a `parfor` loop, because they can cause a worker to become nonresponsive, since it is waiting for input.
- `parfor` does not allow `break` or `return` statements.
- The random numbers you use can affect the results of your computations. See “Reproducibility in Parallel Statistical Computations” on page 30-14.

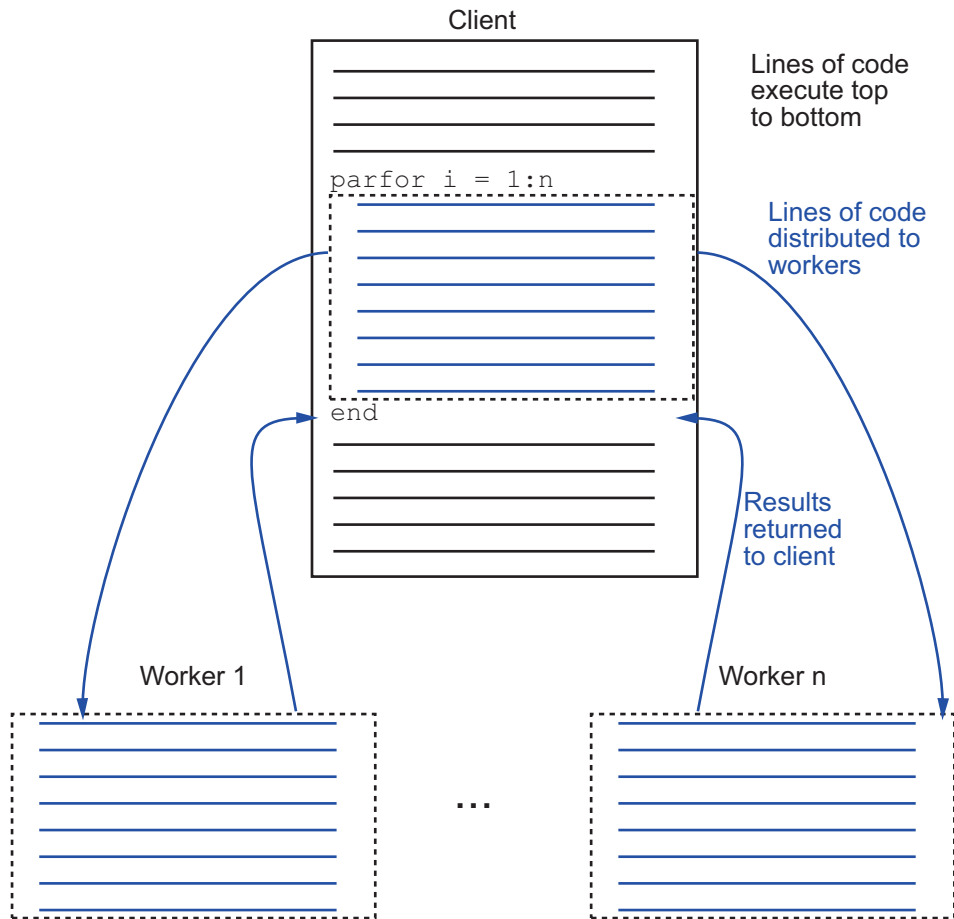
For advice on converting `for` loops to use `parfor`, see “Parallel `for`-Loops (`parfor`)” (Parallel Computing Toolbox).

## Working with parfor

In this section...
“How Statistical Functions Use parfor” on page 30-11
“Characteristics of parfor” on page 30-12

### How Statistical Functions Use parfor

`parfor` is a Parallel Computing Toolbox function similar to a `for` loop. Parallel statistical functions call `parfor` internally. `parfor` distributes computations to worker processors.



## Characteristics of parfor

You might need to adjust your code to run in parallel, for example, you need independent loops and the workers must be able to access the variables. For advice on using `parfor`, see “Parallel for-Loops (`parfor`)” (Parallel Computing Toolbox).

### No Nested `parfor` Loops

`parfor` does not work in parallel when called from within another `parfor` loop, or from an `spmd` block. Parallelization occurs only at the outermost level.

Suppose, for example, you want to apply `jackknife` to your function `userfcn`, which calls `parfor`, and you want to call `jackknife` in a loop. The following figure shows three cases:

- 1 The outermost loop is `parfor`. Only that loop runs in parallel.
- 2 The outermost `parfor` loop is in `jackknife`. Only `jackknife` runs in parallel.
- 3 The outermost `parfor` loop is in `userfcn`. `userfcn` uses `parfor` in parallel.

**Bold** indicates the function that runs in parallel

① `... parfor i=1:10`  
`x(i)=jackknife(@userfcn,...)`  
`... end`  
 Only the outermost `parfor` loop runs in parallel

② `... for i=1:10`  
`x(i)=jackknife(@userfcn,...)`  
`... end`  
 If `UseParallel = 'always'` `jackknife` runs in parallel

③ `... for i=1:10`  
`x(i)=jackknife(@userfcn,...)`  
`... end`  
 If `UseParallel = 'never'` `userfcn` can use `parfor` in parallel

### When parfor Runs in Parallel

For help converting nested loops to use `parfor`, see “Convert for-Loops Into parfor-Loops” (Parallel Computing Toolbox).

See also Quick Start Parallel Computing for Statistics and Machine Learning Toolbox on page 30-2.

## Reproducibility in Parallel Statistical Computations

<b>In this section...</b>
“Issues and Considerations in Reproducing Parallel Computations” on page 30-14
“Running Reproducible Parallel Computations” on page 30-14
“Parallel Statistical Computation Using Random Numbers” on page 30-15

### Issues and Considerations in Reproducing Parallel Computations

A reproducible computation is one that gives the same results every time it runs. Reproducibility is important for:

- Debugging — To correct an anomalous result, you need to reproduce the result.
- Confidence — When you can reproduce results, you can investigate and understand them.
- Modifying existing code — When you change existing code, you want to ensure that you do not break anything.

Generally, you do not need to ensure reproducibility for your computation. Often, when you want reproducibility, the simplest technique is to run in serial instead of in parallel. In serial computation you can simply call the `rng` function as follows:

```
s = rng % Obtain the current state of the random stream
% run the statistical function
rng(s) % Reset the stream to the previous state
% run the statistical function again, obtain identical results
```

This section addresses the case when your function uses random numbers, and you want reproducible results in parallel. This section also addresses the case when you want the same results in parallel as in serial.

### Running Reproducible Parallel Computations

To run a Statistics and Machine Learning Toolbox function reproducibly:

- 1 Set the `UseSubstreams` option to `true`.

- 2 Set the `Streams` option to a type that supports substreams: `'mlfg6331_64'` or `'mrg32k3a'`. For information on these streams, see “Choosing a Random Number Generator” (MATLAB).
- 3 To compute in parallel, set the `UseParallel` option to `true`.
- 4 Call the function with the options structure.
- 5 To reproduce the computation, reset the stream, then call the function again.

To understand why this technique gives reproducibility, see “How Substreams Enable Reproducible Parallel Computations” on page 30-16.

For example, to use the `'mlfg6331_64'` stream for reproducible computation:

- 1 Create an appropriate options structure:

```
s = RandStream('mlfg6331_64');
options = statset('UseParallel',true, ...
    'Streams',s,'UseSubstreams',true);
```

- 2 Run your parallel computation. For instructions, see Quick Start Parallel Computing for Statistics and Machine Learning Toolbox on page 30-2.
- 3 Reset the random stream:

```
reset(s);
```

- 4 Rerun your parallel computation. You obtain identical results.

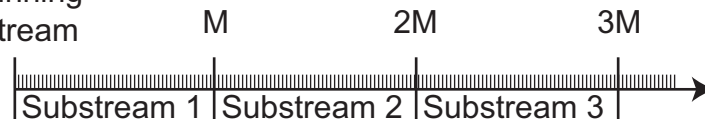
For an example of a parallel computation run this reproducible way, see “Reproducible Parallel Bootstrap” on page 30-26.

## Parallel Statistical Computation Using Random Numbers

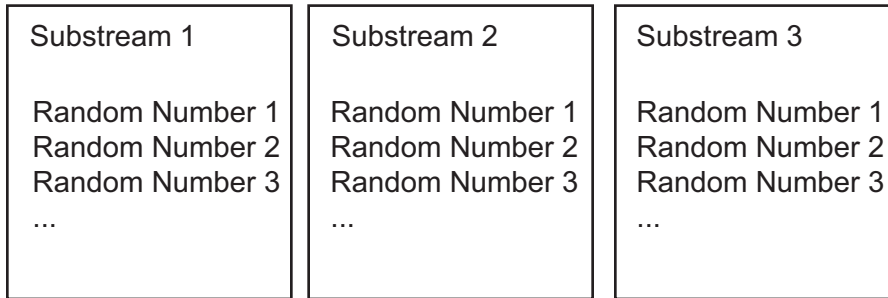
### What Are Substreams?

A substream is a portion of a random stream that `RandStream` can access quickly. There is a number  $M$  such that for any positive integer  $k$ , `RandStream` can go to the  $kM$ th pseudorandom number in the stream. From that point, `RandStream` can generate the subsequent entries in the stream. Currently, `RandStream` has  $M = 2^{72}$ , about  $5e21$ , or more.

Beginning  
of stream



The entries in different substreams have good statistical properties, similar to the properties of entries in a single stream: independence, and lack of  $k$ -way correlation at various lags. The substreams are so long that you can view the substreams as being independent streams, as in the following picture.



Two `RandStream` stream types support substreams: `'mlfg6331_64'` and `'mrg32k3a'`.

### How Substreams Enable Reproducible Parallel Computations

When MATLAB performs computations in parallel with `parfor`, each worker receives loop iterations in an unpredictable order. Therefore, you cannot predict which worker gets which iteration, so cannot determine the random numbers associated with each iteration.

Substreams allow MATLAB to tie each iteration to a particular sequence of random numbers. `parfor` gives each iteration an index. The iteration uses the index as the substream number. Since the random numbers are associated with the iterations, not with the workers, the entire computation is reproducible.

To obtain reproducible results, simply reset the stream, and all the substreams generate identical random numbers when called again. This method succeeds when all the workers use the same stream, and the stream supports substreams. This concludes the discussion of how the procedure in “Running Reproducible Parallel Computations” on page 30-14 gives reproducible parallel results.

### Random Numbers on the Client or Workers

A few functions generate random numbers on the client before distributing them to parallel workers. The workers do not use random numbers, so operate purely deterministically. For these functions, you can run a parallel computation reproducibly using any random stream type.



The functions that operate this way include:

- `crossval`
- `plsregress`
- `sequentialfs`

To obtain identical results, reset the random stream on the client, or the random stream you pass to the client. For example:

```
s = rng % Obtain the current state of the random stream
% run the statistical function
rng(s) % Reset the stream to the previous state
% run the statistical function again, obtain identical results
```

While this method enables you to run reproducibly in parallel, the results can differ from a serial computation. The reason for the difference is `parfor` loops run in reverse order from `for` loops. Therefore, a serial computation can generate random numbers in a different order than a parallel computation. For unequivocal reproducibility, use the technique in “Running Reproducible Parallel Computations” on page 30-14.

### Distributing Streams Explicitly

For testing or comparison using particular random number algorithms, you must set the random number generators. How do you set these generators in parallel, or initialize streams on each worker in a particular way? Or you might want to run a computation using a different sequence of random numbers than any other you have run. How can you ensure the sequence you use is statistically independent?

Parallel Statistics and Machine Learning Toolbox functions allow you to set random streams on each worker explicitly. For information on *creating* multiple streams, enter `help RandStream/create` at the command line. To create four independent streams using the 'mrg32k3a' generator:

```
s = RandStream.create('mrg32k3a', 'NumStreams', 4, ...
    'CellOutput', true);
```

Pass these streams to a statistical function using the `Streams` option. For example:

```
parpool(4) % if you have at least 4 cores
s = RandStream.create('mrg32k3a', 'NumStreams', 4, ...
    'CellOutput', true); % create 4 independent streams
paroptions = statset('UseParallel', true, ...
```

```
'Streams',s); % set the 4 different streams
x = [randn(700,1); 4 + 2*randn(300,1)];
latt = -4:0.01:12;
myfun = @(X) ksdensity(X,latt);
pdfestimate = myfun(x);
B = bootstrp(200,myfun,x,'Options',paroptions);
```

This method of distributing streams gives each worker a different stream for the computation. However, it does not allow for a reproducible computation, because the workers perform the 200 bootstraps in an unpredictable order. If you want to perform a reproducible computation, use substreams as described in “Running Reproducible Parallel Computations” on page 30-14.

If you set the `UseSubstreams` option to `true`, then set the `Streams` option to a single random stream of the type that supports substreams (`'mlfg6331_64'` or `'mrg32k3a'`). This setting gives reproducible computations.

## Implement Jackknife Using Parallel Computing

This example is from the `jackknife` function reference page, but runs in parallel.

```

mypool=parpool()
Starting parpool using the 'local' profile ... connected to 2 workers.

mypool =

    Pool with properties:

        AttachedFiles: {0x1 cell}
        NumWorkers: 2
        IdleTimeout: 30
        Cluster: [1x1 parallel.cluster.Local]
        RequestQueue: [1x1 parallel.RequestQueue]
        SpmdEnabled: 1

opts = statset('UseParallel',true);
sigma = 5;
rng('default')
y = normrnd(0,sigma,100,1);
m = jackknife(@var, y,1,'Options',opts);
n = length(y);
bias = -sigma^2 / n % known bias formula
jbias = (n - 1)*(mean(m)-var(y,1)) % jackknife bias estimate

bias =

    -0.2500

jbias =

    -0.3378

```

This simple example is not a good candidate for parallel computation:

```

% How long to compute in serial?
tic;m = jackknife(@var,y,1);toc
Elapsed time is 0.022771 seconds.

% How long to compute in parallel?
tic;m = jackknife(@var,y,1,'Options',opts);toc
Elapsed time is 0.299066 seconds.

```

`jackknife` does not use random numbers, so gives the same results every time, whether run in parallel or serial.

## Implement Cross-Validation Using Parallel Computing

### In this section...

“Simple Parallel Cross Validation” on page 30-21

“Reproducible Parallel Cross Validation” on page 30-22

### Simple Parallel Cross Validation

This example is the same as the first in the `crossval` function reference page, but runs in parallel.

```

mypool = parpool()
Starting parpool using the 'local' profile ... connected to 2 workers.

mypool =

    Pool with properties:

        AttachedFiles: {0x1 cell}
        NumWorkers: 2
        IdleTimeout: 30
        Cluster: [1x1 parallel.cluster.Local]
        RequestQueue: [1x1 parallel.RequestQueue]
        SpmdEnabled: lopts = statset('UseParallel',true);

load('fisheriris');
y = meas(:,1);
X = [ones(size(y,1),1),meas(:,2:4)];
regf=@(XTRAIN,ytrain,XTEST)(XTEST*regress(ytrain,XTRAIN));

cvMse = crossval('mse',X,y,'Predfun',regf,'Options',opts)

cvMse =

    0.1028

```

This simple example is not a good candidate for parallel computation:

```

% How long to compute in serial?
tic;cvMse = crossval('mse',X,y,'Predfun',regf);toc
Elapsed time is 0.073438 seconds.

```

```
% How long to compute in parallel?
tic;cvMse = crossval('mse',X,y,'Predfun',regf,...
    'Options',opts);toc
Elapsed time is 0.289585 seconds.
```

## Reproducible Parallel Cross Validation

To run `crossval` in parallel in a reproducible fashion, set the options and reset the random stream appropriately (see “Running Reproducible Parallel Computations” on page 30-14).

```
mypool = parpool()
```

```
Starting parpool using the 'local' profile ... connected to 2 workers.
```

```
mypool =
```

```
Pool with properties:
```

```
AttachedFiles: {0x1 cell}
NumWorkers: 2
IdleTimeout: 30
Cluster: [1x1 parallel.cluster.Local]
RequestQueue: [1x1 parallel.RequestQueue]
SpmEnabled: 1
```

```
s = RandStream('mlfg6331_64');
opts = statset('UseParallel',true,...
    'Streams',s,'UseSubstreams',true);

load('fisheriris');
y = meas(:,1);
X = [ones(size(y,1),1),meas(:,2:4)];
regf=@(XTRAIN,ytrain,XTEST)(XTEST*regress(ytrain,XTRAIN));

cvMse = crossval('mse',X,y,'Predfun',regf,'Options',opts)

cvMse =

    0.1020
```

Reset the stream:

```
reset(s)
cvMse = crossval('mse',X,y,'Predfun',regf,'Options',opts)

cvMse =

    0.1020
```

## Implement Bootstrap Using Parallel Computing

<b>In this section...</b>
“Bootstrap in Serial and Parallel” on page 30-24
“Reproducible Parallel Bootstrap” on page 30-26

### Bootstrap in Serial and Parallel

Here is an example timing a bootstrap in parallel versus in serial. The example generates data from a mixture of two Gaussians, constructs a nonparametric estimate of the resulting data, and uses a bootstrap to get a sense of the sampling variability.

- 1 Generate the data:

```
% Generate a random sample of size 1000,  
% from a mixture of two Gaussian distributions  
x = [randn(700,1); 4 + 2*randn(300,1)];
```

- 2 Construct a nonparametric estimate of the density from the data:

```
latt = -4:0.01:12;  
myfun = @(X) ksdensity(X,latt);  
pdfestimate = myfun(x);
```

- 3 Bootstrap the estimate to get a sense of its sampling variability. Run the bootstrap in serial for timing comparison.

```
tic;B = bootstrp(200,myfun,x);toc
```

```
Elapsed time is 10.878654 seconds.
```

- 4 Run the bootstrap in parallel for timing comparison:

```
mypool = parpool()  
Starting parpool using the 'local' profile ... connected to 2 workers.
```

```
mypool =
```

```
Pool with properties:
```

```
AttachedFiles: {0x1 cell}  
NumWorkers: 2  
IdleTimeout: 30  
Cluster: [1x1 parallel.cluster.Local]
```



```
RequestQueue: [1x1 parallel.RequestQueue]
SpmEnabled: 1

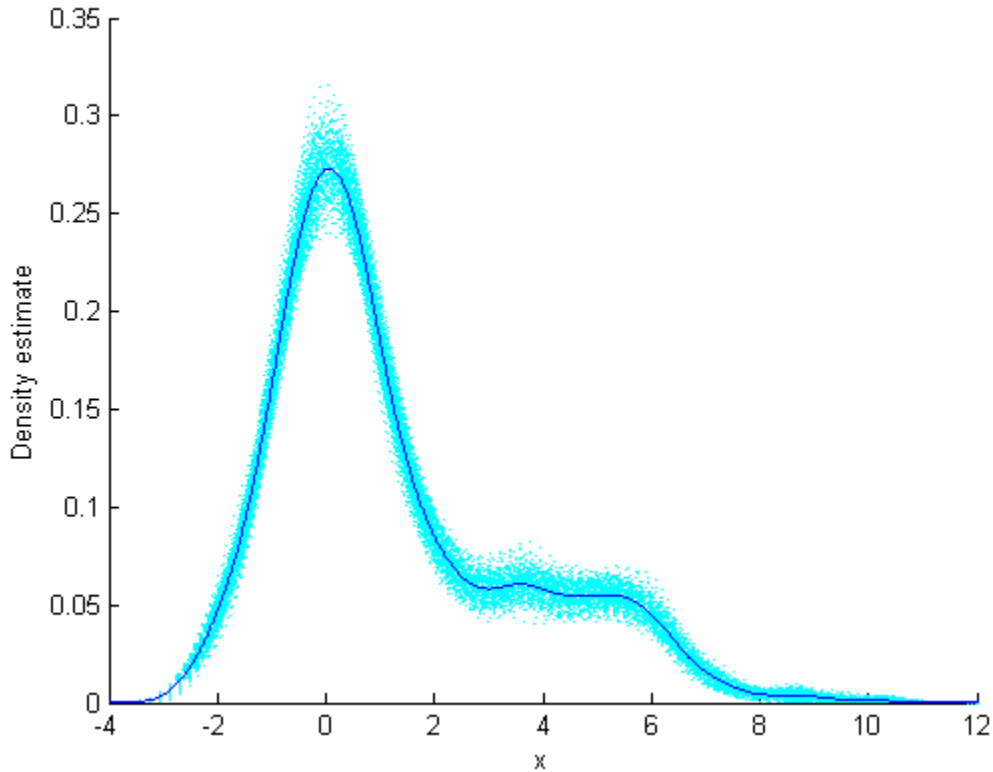
opt = statset('UseParallel',true);
tic;B = bootstrp(200,myfun,x,'Options',opt);toc

Elapsed time is 6.304077 seconds.
```

Computing in parallel is nearly twice as fast as computing in serial for this example.

Overlay the `ksdensity` density estimate with the 200 bootstrapped estimates obtained in the parallel bootstrap. You can get a sense of how to assess the accuracy of the density estimate from this plot.

```
hold on
for i=1:size(B,1),
    plot(latt,B(i,:), 'c:')
end
plot(latt,pdfestimate);
xlabel('x');ylabel('Density estimate')
```



## Reproducible Parallel Bootstrap

To run the example in parallel in a reproducible fashion, set the options appropriately (see “Running Reproducible Parallel Computations” on page 30-14). First set up the problem and parallel environment as in “Bootstrap in Serial and Parallel” on page 30-24. Then set the options to use substreams along with a stream that supports substreams.

```
s = RandStream('mlfg6331_64'); % has substreams
opts = statset('UseParallel',true,...
    'Streams',s,'UseSubstreams',true);
B2 = bootstrap(200,myfun,x,'Options',opts);
```

To rerun the bootstrap and get the same result:

```
reset(s) % set the stream to initial state
B3 = bootstrp(200,myfun,x,'Options',opts);
isequal(B2,B3) % check if same results

ans =
     1
```



# Code Generation

---

- “Code Generation Workflows” on page 31-2
- “C Code Generation and Classification Learner” on page 31-8
- “Predict Class Labels Using MATLAB Function Block” on page 31-18
- “Code Generation Support, Usage Notes, and Limitations” on page 31-24
- “Specify Variable-Sized Arguments for Code Generation” on page 31-32
- “System Objects for Classification and Code Generation” on page 31-36
- “Code Generation Workflow for Exhaustive Searcher” on page 31-47
- “Predict Class Labels Using Stateflow” on page 31-52

## Code Generation Workflows

Certain Statistics and Machine Learning Toolbox functions have been enabled to generate C code using MATLAB Coder. However, such functions have limitations that can require workflow adjustments. To determine which Statistics and Machine Learning Toolbox functions support code generation and whether there are limitations, see “Code Generation Support, Usage Notes, and Limitations” on page 31-24.

---

**Note** To generate code that uses Statistics and Machine Learning Toolbox functions, you must install MATLAB Coder software.

---

In this section...
“General Code Generation Support and Limitations” on page 31-2
“Code Generation Workflow” on page 31-2
“Code Generation Workflow for Prediction” on page 31-5

### General Code Generation Support and Limitations

The `codegen` function, which is included in MATLAB Coder, generates C code for a function given these conditions:

- Call the function within a MATLAB function that you declare, for example, `foo.m`, and then generate code from `foo.m`. That is, MATLAB Coder cannot generate code for functions at the top level. All functions within `foo.m` must support code-generation. For a list of all functions that support code generation, see “Functions and Objects Supported for C/C++ Code Generation — Category List” (MATLAB Coder).
- The input arguments of `foo.m` are not objects, for example, classification or regression model objects. That is, the `-args` option of `codegen` does not accept objects. For supported classification and regression models, work around this limitation by saving the model at the command line using `saveCompactModel`. Then load the model within `foo.m` using `loadCompactModel`.

### Code Generation Workflow

To generate code for Statistics and Machine Learning Toolbox functions at the command line or using the MATLAB Coder app, follow these steps:

- 1 Declare a MATLAB function, as you would normally, using functions from Statistics and Machine Learning Toolbox that support code generation. Ensure that you type the `%#codegen` compiler directive somewhere in the function. For example, to generate code that estimates the interquartile range of a data set, declare this function.

```
function r = iqrCodeGen(x) %#codegen
%IQRCODEGEN Estimate interquartile range
%   iqrCodeGen returns the interquartile range of the data x,
%   a single- or double-precision vector.
r = iqr(x);
end
```

You can allow for optional input arguments by specifying `varargin` as an input argument. For an example, see “Specify Variable-Sized Arguments for Code Generation” on page 31-32.

- 2 To generate C code, you must have access to a C compiler, and the compiler must be configured properly. For more details, see “Setting Up Your C Compiler” (MATLAB Coder). For example, to set up the compiler for MEX-code generation, enter

```
mex -setup
```

- 3 Determine the environment in which you plan to deploy the generated code. Generated code can run:

- Inside the MATLAB environment as a C-MEX file
- Outside the MATLAB environment as a standalone executable
- Outside the MATLAB environment as a shared utility linked to another standalone executable

Generating a MEX file allows you to analyze its input and output arguments using MATLAB tools before deploying the function outside the MATLAB environment. In a MEX function, you can include code for verification, but not for code generation by declaring those commands as extrinsic using `coder.extrinsic`. Extrinsic commands can include functions that do not have code generation support. All extrinsic commands in the MEX function run in MATLAB, but `codegen` does not generate code for them.

If you plan to deploy the code outside the MATLAB environment, then you must generate a standalone executable. One way to specify your compiler choice is by using the `-config` option of `codegen`. For example, to generate a static C

executable, specify `-config:exe` when you call `codegen`. For more details on setting code generation options, see the `-config` option of `codegen`.

- 4 Generate code for the function. When you generate code, you must define the size, class, and complexity of all inputs to the function. Create a commensurate variable in the Workspace.

- To generate code at the command line, use `codegen`. Declare all input argument dimensions and data types when you call `codegen` using the `-args` option.
- To specify the data type and exact dimensions, pass such a variable with the desired dimensions. For example, to specify that the MEX file generated from `iqrCodeGen.m` must accept a double-precision numeric column vector with 100 elements, enter:

```
testX = randn(100,1);  
codegen iqrCodeGen.m -args {testX} -report
```

The `-report` flag generates a code generation report.

- To specify that at least one of the dimensions can have any length, then create a variable by using `coder.typeof`. Specify the data type, maximal dimensions, and which dimension can vary, respectively. Such a specification is convenient for an input data set that, at compile time, has an unknown number of observations. For example, to specify that the MEX file generated from `iqrCodeGen.m` can accept a double-precision numeric column vector of any length, enter:

```
testX = coder.typeof(0,[Inf,1],[1,0]);  
codegen iqrCodeGen.m -args {testX} -report
```

For an example, see “Specify Variable-Sized Arguments for Code Generation” on page 31-32.

- To specify character array, such as supported name-value pair arguments, declare the character array as a constant using `coder.Constant`. For example, suppose that `'Name'` is a valid name-value pair argument for `iqrCodeGen.m`, and the corresponding value, `value`, is numeric. Enter:

```
codegen iqrCodeGen.m -args {testX,coder.Constant('Name'),value} -report
```

- To generate code using the MATLAB Coder app, see “Generate MEX Functions by Using the MATLAB Coder App” (MATLAB Coder).



Irrespective of how you generate code, MATLAB creates a folder in your current working folder containing, among other things, the generated code. In the example, suppose that you generate the code on a 64-bit windows machine and *pwd* is your current working folder. Then, the generated MEX-file is `iqrCodeGen_mex.mexw64` in `pwd/codegen/mex/iqrCodeGen`.

- 5 If code generation failed, then review the report. A report link populates at the command line, but you can also find the report in the `html` folder at the same level as the MEX file. In the report, MATLAB Coder identifies:
  - Readiness issues, that is, MATLAB Coder flags unsupported features or functions.
  - Build issues, that is, coding patterns that are not supported for code generation.

After viewing the report, modify your MATLAB code until you get a successful build.

## Code Generation Workflow for Prediction

Certain classification and regression model classes have a `predict` function that supports code generation. Because prediction requires a trained classification or regression model object, you must adjust your workflow slightly to accommodate code generation. The following steps describe a brief workflow for label prediction using a classification model. For more detailed example code, see “Specify Variable-Sized Arguments for Code Generation” on page 31-32.

To generate code for label prediction using a classification model at the command line, follow these steps (for regression model prediction, the workflow is similar):

- 1 Train a classification model object equipped with a code-generation enabled `predict` function. Ensure that you specify only those name-value pairs arguments that are enabled for code-generation. For example, to train a support vector machine (SVM) using the predictor data in `X` and class labels in `Y`, enter:

```
Mdl = fitcsvm(X,Y);
```

This step includes data preprocessing, feature selection, and optimizing the model using, for example, cross-validation.

- 2 Save the classification model to disk using `saveCompactModel`. For example, enter

```
saveCompactModel(Mdl, 'SVMModel');
```

MATLAB makes the full classification model `Mdl` compact, and then saves it to the MATLAB binary file `SVModel.mat` as a structure array in the current working folder.

- 3 Declare a MATLAB function that returns predicted labels for input predictor data. Within the function, load the trained classification model, and then pass the loaded model to `predict`. For example, declare this function, which predicts labels using the SVM model `Mdl`.

```
function label = predictLabelsSVM(x) %#codegen
%PREDICTLABELSSVM Label new observations using trained SVM model Mdl
% predictLabelsSVM predicts the vector of labels label using
% the saved SVM model Mdl and the predictor data x.
Mdl = loadCompactModel('SVModel');
label = predict(Mdl,x);
end
```

Ensure that you type the `%codegen` compiler directive somewhere in the function.

- 4 Follow steps 2–5 of “Code Generation Workflow” on page 31-2.
- 5 Verify whether the MEX-file returns the expected labels by passing some predictor data to `predict` and to the MEX-file. Then, compare the predicted labels. For example,

```
predictLabels = resubPredict(Mdl);
mexLabels = predictLabelsSVM_mex(X);
verifyMEX = sum(mexLabels == predictLabels) == numel(predictLabels)
```

If `verifyMEX` is 1, then the MEX-file returns the expected labels.

## See Also

`codegen`

## Related Examples

- “Code Generation Workflow for Exhaustive Searcher” on page 31-47
- “Code Generation Support, Usage Notes, and Limitations” on page 31-24
- “Specify Variable-Sized Arguments for Code Generation” on page 31-32
- “C Code Generation and Classification Learner” on page 31-8

- “C Code Generation for Image Classifier”
- “Getting Started with MATLAB Coder” (MATLAB Coder)

## C Code Generation and Classification Learner

Classification Learner is well suited for choosing and training classification models interactively, but it does not automatically generate C code from a function that labels data based on a trained model. This example shows how to generate C code from a function that predicts labels using an exported classification model. The example builds a model that predicts the credit rating of a business given various financial ratios, according to these steps:

- 1 Use the credit rating data set in the file `CreditRating_Historical.dat`, which is included with Statistics and Machine Learning Toolbox.
- 2 Reduce the data dimensionality using PCA.
- 3 Train a set of models that support C code generation for label prediction.
- 4 Export the model with the minimum 5-fold, cross-validated classification accuracy.
- 5 Generate code from a function that transforms the new predictor data and then predicts corresponding labels using the exported model.

Load the historical credit rating data set in the file `CreditRating_Historical.dat`, which is on the MATLAB path, into a table.

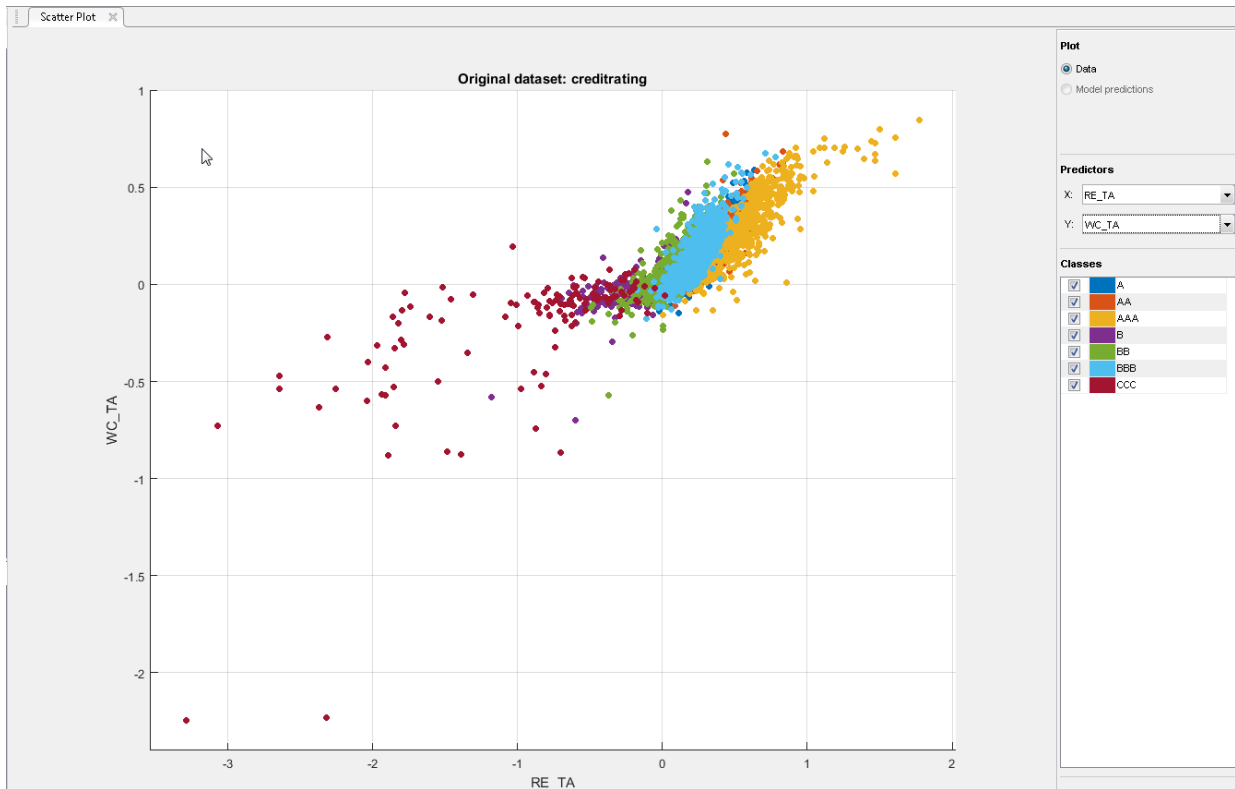
```
creditrating = readtable('CreditRating_Historical.dat');
```

On the **Apps** tab, click **Classification Learner**.

In Classification Learner, on the **Classification Learner** tab, in the **File** section, click **New Session** and select **From Workspace**.

In the New Session dialog box, select the table `creditrating`. All variables, except the one identified as the response, are double-precision numeric vectors. Click **Start Session** to compare classification models based on the 5-fold, cross-validated classification accuracy.

Classification Learner loads the data and plots a scatter plot of the variables `WC_TA` versus `ID`. Because identification numbers are not helpful to display in a plot, choose `RE_TA` for **X** under **Predictors**.



The scatter plot suggests that the two variables can separate the classes AAA, BBB, BB, and CCC fairly well. However, the observations corresponding to the remaining classes are mixed into these classes.

Identification numbers are not helpful for prediction. Therefore, in the **Features** section, click **Feature Selection** and then clear the **ID** check box. You can also remove unnecessary predictors from the beginning by using the check boxes in the New Session dialog box. This example shows how to remove unused predictors for code generation when you have included all predictors.

In the **Features** section, click **PCA**, and then select **Enable PCA**. This action applies PCA to the predictor data, and then transforms the data before training the models. Classification Learner uses only components that collectively explain 95% of the variability.

Select the following classification models and options, which support code generation for label prediction, and then perform cross-validation (for more details, see “Code Generation Support, Usage Notes, and Limitations” on page 31-24). To select each model, in the **Model Type** section, click the **Show more** arrow, and then click the model. After selecting a model and specifying any options, close any open menus, and then click **Train** in the **Training** section.

Models and Options to Select	Description
Under <b>Decision Trees</b> , select <b>All Trees</b>	Classification trees of various complexities
Under <b>Support Vector Machines</b> , select <b>All SVMs</b>	SVMs of various complexities and using various kernels. Complex SVMs require time to fit.
Under <b>Ensemble Classifiers</b> , select <b>Boosted Trees</b> . In the <b>Model Type</b> section, click <b>Advanced</b> . Reduce <b>Maximum number of splits</b> to <b>5</b> and increase <b>Number of learners</b> to <b>100</b> .	Boosted ensemble of classification trees
Under <b>Ensemble Classifiers</b> , select <b>Bagged Trees</b> . In the <b>Model Type</b> section, click <b>Advanced</b> . Increase <b>Maximum number of splits</b> to <b>50</b> and increase <b>Number of learners</b> to <b>100</b> .	Random forest of classification trees

After cross-validating each model type, the Data Browser displays each model and its 5-fold, cross-validated classification accuracy, and highlights the model with the best accuracy.

Data Browser

▼ History

1.1 ☆ Tree	Accuracy: 69.6%
Last change: Complex Tree	2/6 features (PCA on)
1.2 ☆ Tree	Accuracy: 67.0%
Last change: Medium Tree	2/6 features (PCA on)
1.3 ☆ Tree	Accuracy: 57.8%
Last change: Simple Tree	2/6 features (PCA on)
2.1 ☆ SVM	Accuracy: 70.5%
Last change: Linear SVM	2/6 features (PCA on)
2.2 ☆ SVM	Accuracy: 64.4%
Last change: Quadratic SVM	2/6 features (PCA on)
2.3 ☆ SVM	Accuracy: 46.5%
Last change: Cubic SVM	2/6 features (PCA on)
2.4 ☆ SVM	Accuracy: <b>70.7%</b>
Last change: Fine Gaussian SVM	2/6 features (PCA on)
2.5 ☆ SVM	Accuracy: 69.6%
Last change: Medium Gaussian SVM	2/6 features (PCA on)
2.6 ☆ SVM	Accuracy: 61.7%
Last change: Coarse Gaussian SVM	2/6 features (PCA on)
3 ☆ Ensemble	Accuracy: 66.3%
Last change: 'Number of learners' = '100'	2/6 features (PCA on)
4 ☆ Ensemble	Accuracy: 70.6%
Last change: 'Number of learners' = '100'	2/6 features (PCA on)

▼ Current Model

**Model 4: Trained**

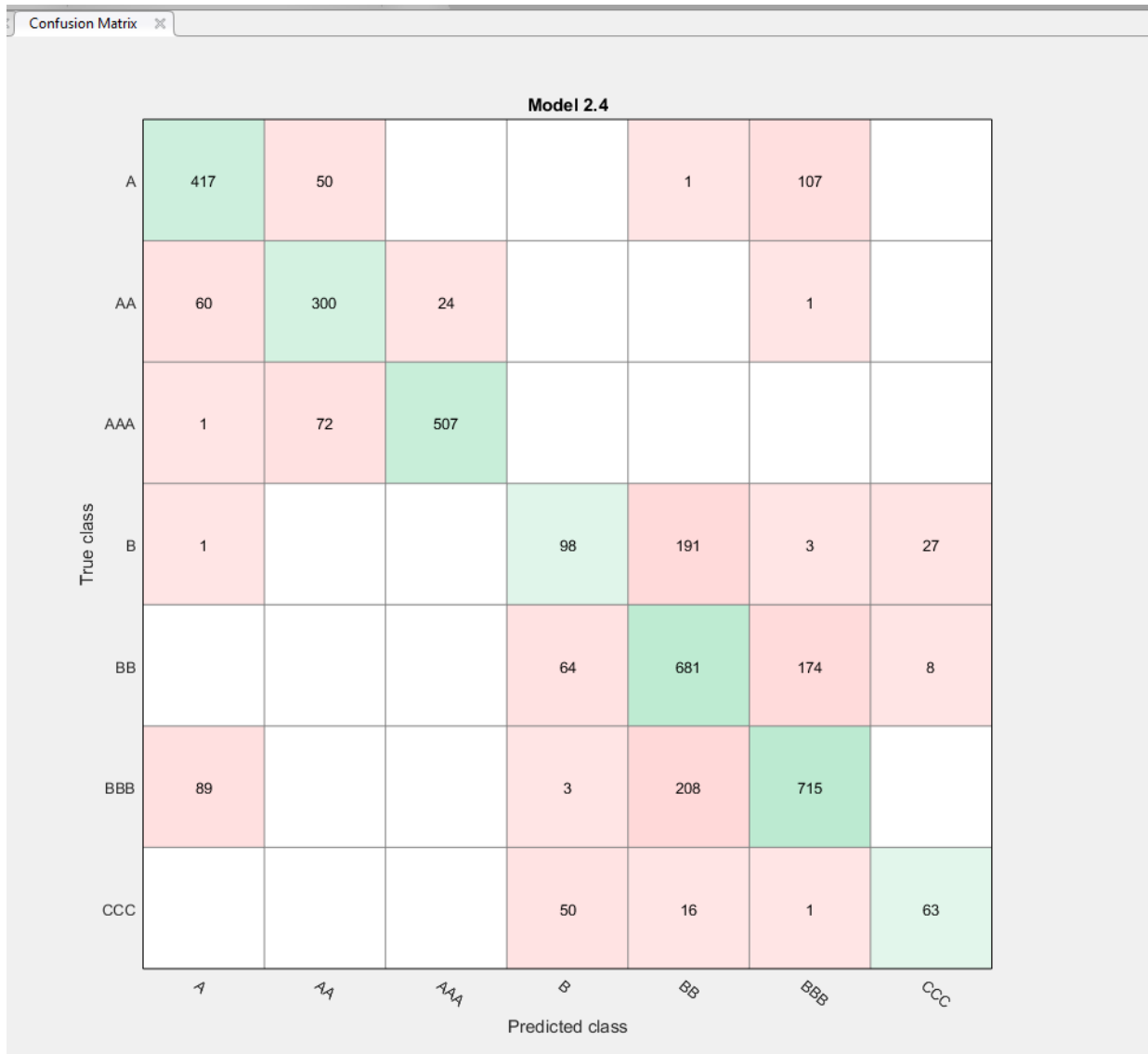
**Results**

Accuracy	70.6%
Prediction speed	~4900 obs/sec
Training time	15.212 sec

Select the model that yields the maximum 5-fold, cross-validated classification accuracy, which is the error-correcting output codes (ECOC) model of Fine Gaussian SVM learners. With PCA enabled, Classification Learner uses two predictors out of six.

In the **Plots** section, click **Confusion Matrix**.





The model does well distinguishing between A, B, and C classes. However, the model does not do as well distinguishing between particular levels within those groups, the lower B levels in particular.

In the **Export** section, click **Export Model**, and then select **Export Compact Model**. Click **OK** in the dialog box.

The structure `trainedModel` appears in the MATLAB Workspace. The field `ClassificationSVM` of `trainedModel` contains the compact model.

At the command line, save the compact model to a file called `ClassificationLearnerModel.mat` in your current folder.

```
saveCompactModel(trainedModel.ClassificationSVM, 'ClassificationLearnerModel');
```

To preprocess new data in the same way you preprocessed the training data, you need the following three model parameters:

- `removeVars` — Column vector of at most  $p$  elements identifying indices of variables to remove from the data, where  $p$  is the number of predictor variables in the raw data
- `pcaCenters` — Row vector of exactly  $q$  PCA centers
- `pcaCoefficients` —  $q$ -by- $r$  matrix of PCA coefficients, where  $r$  is at most  $q$

Specify the indices of predictor variables that you removed while selecting data using **Feature Selection** in Classification Learner. Extract the PCA statistics from `trainedModel`.

```
removeVars = 1;  
pcaCenters = trainedModel.PCACenters;  
pcaCoefficients = trainedModel.PCACoefficients;
```

Save the model parameters to a file named `ModelParameters.mat` in your current folder.

```
save('ModelParameters.mat', 'removeVars', 'pcaCenters', 'pcaCoefficients');
```

In your current folder, declare a function named `mypredictCL.m` that:

- Accepts a numeric matrix ( $X$ ) of raw observations containing the same predictor variables as the ones passed into Classification Learner
- Loads the classification model in `ClassificationLearnerModel.mat` and the model parameters in `ModelParameters.mat`
- Removes the predictor variables corresponding to the indices in `removeVars`
- Transforms the remaining predictor data using the PCA centers (`pcaCenters`) and coefficients (`pcaCoefficients`) estimated by Classification Learner

- Returns predicted labels using the model

```
function label = mypredictCL(X) %#codegen
%MPREDICTCL Classify credit rating using model exported from
%Classification Learner
% MYPREDICTCL loads trained classification model (SVM) and model
% parameters (removeVars, pcaCenters, and pcaCoefficients), removes the
% columns of the raw matrix of predictor data in X corresponding to the
% indices in removeVars, transforms the resulting matrix using the PCA
% centers in pcaCenters and PCA coefficients in pcaCoefficients, and then
% uses the transformed data to classify credit ratings. X is a numeric
% matrix with n rows and 7 columns. label is an n-by-1 cell array of
% predicted labels.

% Load trained classification model and model parameters
SVM = loadCompactModel('ClassificationLearnerModel');
data = coder.load('ModelParameters');
removeVars = data.removeVars;
pcaCenters = data.pcaCenters;
pcaCoefficients = data.pcaCoefficients;
SVM = loadCompactModel('ClassificationLearnerModel');

% Remove unused predictor variables
keepvars = 1:size(X,2);
idx = ~ismember(keepvars,removeVars);
keepvars = keepvars(idx);
XwoID = X(:,keepvars);

% Transform predictors via PCA
Xpca = bsxfun(@minus,XwoID,pcaCenters)*pcaCoefficients;

% Generate label from SVM
label = predict(SVM,Xpca);
end
```

Create a double-precision matrix called `x` for code generation using `coder.typeof`. Specify that the number of rows of `x` is arbitrary, but that `x` must have `p` columns.

```
p = size(creditrating,2) - 1;
x = coder.typeof(0, [Inf,p], [1 0]);
```

Generate a MEX function from `mypredictCL.m`. Use the `-args` option to specify `x` as an argument.

```
codegen -config:mex mypredictCL.m -args x
```

codegen generates the MEX file `mypredictCL_mex.mexw64` in your current folder. The file extension depends on your platform.

Verify that the MEX function returns the expected labels.

Remove the response variable from the original data set, and then randomly draw 15 observations.

```
rng('default'); % For reproducibility
m = 15;
testsampleT = datasample(creditrating(:,1:(end - 1)),m);
```

Predict corresponding labels by using `predictFcn` in the classification model trained by Classification Learner.

```
testLabels = trainedModel.predictFcn(testsampleT);
```

Convert the resulting table to a matrix.

```
testsample = table2array(testsampleT);
```

The columns of `testsample` correspond to the columns of the predictor data loaded by Classification Learner.

Pass the test data to `mypredictCL`. The function `mypredictCL` predicts corresponding labels by using `predict` and the classification model trained by Classification Learner.

```
testLabelsPredict = mypredictCL(testsample);
```

Predict corresponding labels by using the generated MEX function `mypredictCL_mex`.

```
testLabelsMEX = mypredictCL_mex(testsample);
```

Compare the sets of predictions.

```
sum(strcmp(testLabels,testLabelsPredict)) == m
sum(strcmp(testLabels,testLabelsMEX)) == m
```

```
ans =
```

```
logical
```

```
1
```

```
ans =  
  
    logical  
  
     1
```

`predictFcn`, `mypredictCL`, and the `MEX` function return the same values.

## See Also

`codegen` | `coder.typeof` | `loadCompactModel` | `saveCompactModel`

## Related Examples

- “Code Generation Workflows” on page 31-2
- “Classification Learner App”
- “C Code Generation for Image Classifier”

## Predict Class Labels Using MATLAB Function Block

This example shows how to add a MATLAB Function block to a Simulink® for label prediction. The MATLAB Function block accepts streaming data, and predicts the label and classification score using a trained, support vector machine (SVM) classification model. For details on using the MATLAB Function block, see “Create Model That Uses MATLAB Function Block” (Simulink).

The `ionosphere` data set, which is included in the Statistics and Machine Learning Toolbox™, contains radar-return qualities (`Y`) and predictor data (`X`). Radar returns are either of good quality ('g') or of bad quality ('b').

Load the `ionosphere` data set. Determine the sample size.

```
load ionosphere
n = numel(Y)
```

```
n =
    351
```

The MATLAB Function block cannot return cell arrays. So, convert the response variable to a logical vector whose elements are 1 if the radar returns are good, and 0 otherwise.

```
Y = strcmp(Y, 'g');
```

Suppose that the radar returns are detected in sequence, and you have the first 300 observations, but you have not received the last 51 yet. Partition the data into present and future samples.

```
prsntX = X(1:300, :);
prsntY = Y(1:300);
ftrX = X(301:end, :);
ftrY = Y(301:end);
```

Train an SVM model using all, presently available data. Specify predictor data standardization.

```
Mdl = fitcsvm(prsntX, prsntY, 'Standardize', true);
```

Mdl is a `ClassificationSVM` model. At the command line, you can use Mdl to make predictions for new observations. However, you cannot use Mdl as an input argument in a function meant for code generation.

Prepare Mdl to be loaded within the function using `saveCompactModel`.

```
saveCompactModel(Mdl, 'SVMIonosphere');
```

`saveCompactModel` compacts Mdl, and then saves it in the MAT-file `SVMIonosphere.mat`.

Declare a function named `svmIonospherePredict.m` that predicts whether a radar return is of good quality. The function should:

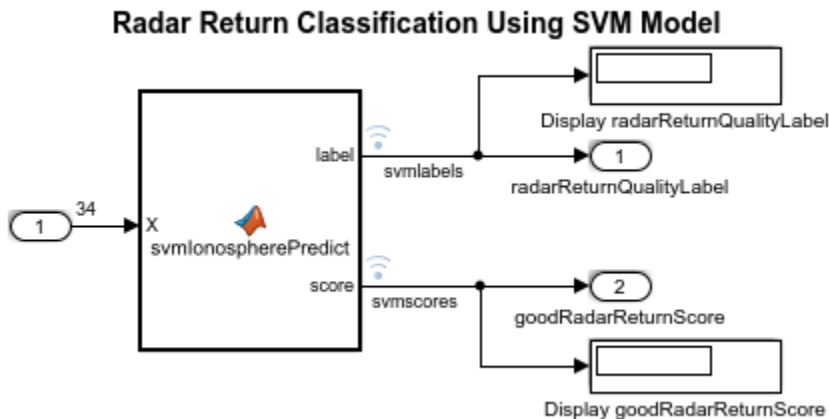
- Include the code generation directive `codegen` somewhere in the function.
- Accept radar-return predictor data. The data must be commensurate with X except for the number of rows.
- Load `SVMIonosphere.mat` using `loadCompactModel`.
- Return predicted labels and classification scores for predicting the quality of the radar return as good (that is, the positive-class score).

```
function [label,score] = svmIonospherePredict(X) %codegen
%svmIonospherePredict Predict radar-return quality using SVM model
% svmIonospherePredict predicts labels and estimates classification
% scores of the radar returns in the numeric matrix of predictor data X
% using the compact SVM model in the file SVMIonosphere.mat. Rows of X
% correspond to observations and columns to predictor variables. label
% is the predicted label and score is the confidence measure for
% classifying the radar-return quality as good.
%
% Copyright 2016 The MathWorks Inc.
Mdl = loadCompactModel('SVMIonosphere');
[label,bothscores] = predict(Mdl,X);
score = bothscores(:,2);
end
```

`svmIonospherePredict.m` is located in your `mlr/help/toolbox/stats/examples` folder, where `mlr` is the value of `matlabroot`.

Load the Simulink® model `slexSVMIonospherePredictExample.slx` located in `mlr/examples/stats`.

```
mlr = matlabroot;
SimMdlName = 'slexSVMIonospherePredictExample';
pathToModel = fullfile(mlr, 'examples', 'stats', SimMdlName);
open_system(pathToModel);
```



The figure displays the Simulink® model. When the input node detects a radar return, it directs that observation into the MATLAB Function block that dispatches to `svmIonospherePredict.m`. After predicting the label and score, the model returns these values to the workspace and displays the values within the model one at a time. When you load `slexSVMIonospherePredictExample.slx`, MATLAB® also loads the data set that it requires called `radarReturnInput`. However, this example shows how to construct the required data set.

The model expects to receive input data as a structure array called `radarReturnInput` containing these fields:

- `time` - The points in time at which the observations enter the model. In the example, the duration includes the integers from 0 through 50. The orientation must correspond to the observations in the predictor data. So, for this example, `time` must be a column vector.



- `signals` - A 1-by-1 structure array describing the input data, and containing the fields `values` and `dimensions`. `values` is a matrix of predictor data. `dimensions` is the number of predictor variables.

Create an appropriate structure array for future radar returns.

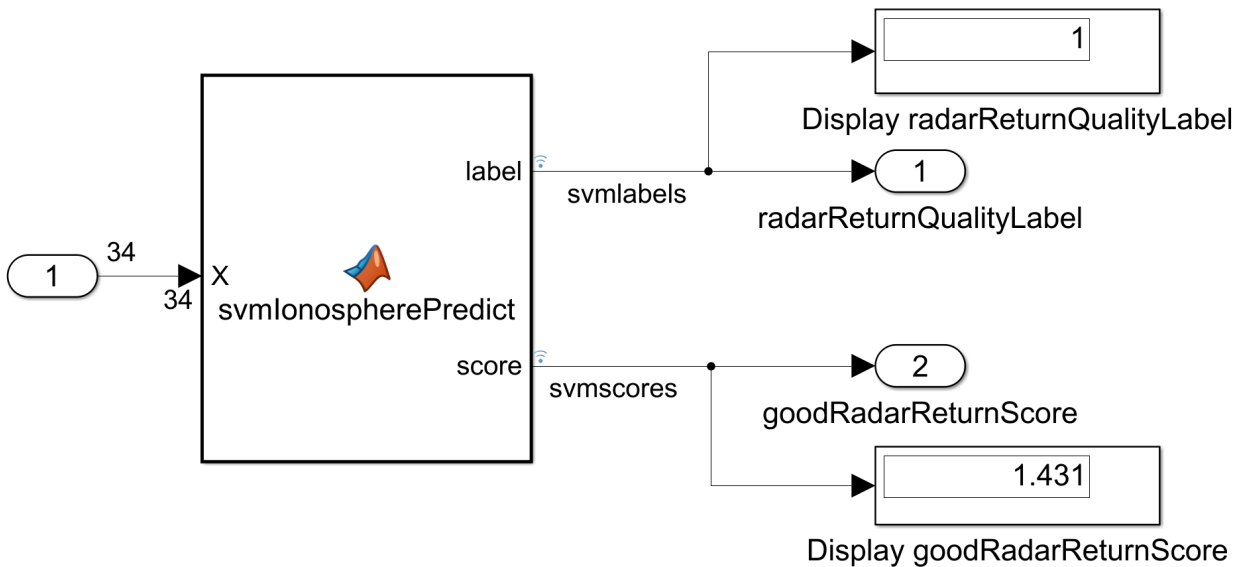
```
radarReturnInput.time = (0:50)';
radarReturnInput.signals(1).values = ftrX;
radarReturnInput.signals(1).dimensions = size(ftrX,2);
```

You can change the name from `radarReturnInput`, and then specify the new name in the model. However, Simulink® expects the structure array to contain the described field names.

Simulate the model using the data held out of training, that is, the data in `radarReturnInput`.

```
sim(SimMdlName);
```

## Radar Return Classification Using SVM Model



The figure shows the model after it processes all observations in `radarReturnInput` one at a time. The predicted label of `X(351, :)` is 1 and its positive-class score is 1.431. The

variables `tout`, `yout`, and `svmlogout` appear in the workspace. `yout` and `svmlogout` are `SimulinkData.Dataset` objects containing the predicted labels and scores. For more details, see “Data Format for Logged Simulation Data” (Simulink).

Extract the simulation data from the simulation log.

```
labelsSL = svmlogout.getElement(1).Values.Data;  
scoresSL = svmlogout.getElement(2).Values.Data;
```

`labelsSL` is a 51-by-1 numeric vector of predicted labels. `labelsSL(j) = 1` means that the SVM model predicts that radar return `j` in the future sample is of good quality, and 0 means otherwise. `scoresSL` is a 51-by-1 numeric vector of positive-class scores, that is, signed distances from the decision boundary. Positive scores correspond to predicted labels of 1, and negative scores correspond to predicted labels of 0.

Predict labels and positive-class scores at the command line using `CompactClassificationSVM.predict`.

```
[labelCMD, scoresCMD] = predict(Mdl, ftrX);  
scoresCMD = scoresCMD(:,2);
```

`labelCMD` and `scoresCMD` are commensurate with `labelsSL` and `scoresSL`.

Compare the future-sample, positive-class scores returned by `slexSVMIonospherePredictExample` to those returned by calling `predict` at the command line.

```
err = sum((scoresCMD - scoresSL).^2);  
err < eps
```

```
ans =
```

```
logical
```

```
1
```

The sum of squared deviations between the sets of scores is negligible.

If you also have a Simulink® Coder™ license, then you can generate C code from `slexSVMIonospherePredictExample.slx` in Simulink® or from the command line using `rtwbuild`. For more details, see “Generate C Code for a Model” (Simulink Coder).

## See Also

`loadCompactModel` | `predict` | `rtwbuild` | `saveCompactModel`

## Related Examples

- “Create Model That Uses MATLAB Function Block” (Simulink)
- “Data Format for Logged Simulation Data” (Simulink)
- “Code Generation Support, Usage Notes, and Limitations” on page 31-24
- “Generate C Code for a Model” (Simulink Coder)
- “C Code Generation and Classification Learner” on page 31-8
- “C Code Generation for Image Classifier”
- “System Objects for Classification and Code Generation” on page 31-36
- “Predict Class Labels Using Stateflow” on page 31-52

## Code Generation Support, Usage Notes, and Limitations

To generate C code from built-in Statistics and Machine Learning Toolbox functions, you must install MATLAB Coder software. This table lists the built-in Statistics and Machine Learning Toolbox functions that support code generation, and describes their limitations and caveats.

- All functions in this table generate reference C code.
- You cannot call any function at the top level. Instead, call the function within a function that you declare, and then generate code from the declared function.

An asterisk (\*) indicates that the reference page has usage notes and limitations for C/C++ code generation.

<b>Descriptive Statistics and Visualization</b>
geomean
harmmean
iqr
kurtosis
mad*
moment*
nancov*
nanmax
nanmean
nanmedian
nanmin
nanstd
nansum
nanvar
prctile*
quantile
skewness
zscore

<b>Probability Distributions</b>
betacdf
betafit
betainv
betalike
betapdf
betarnd*
betastat
binocdf
binoinv
binopdf
binornd*
binostat
cdf
chi2cdf
chi2inv
chi2pdf
chi2rnd*
chi2stat
evcdf
evinv
evpdf
evrnd*
evstat
expcdf
expinv
exppdf
exprnd*
expstat

fcdf
finv
fpdf
frnd*
fstat
gamcdf
gaminv
gampdf
gamrnd*
gamstat
geocdf
geoinv
geopdf
geornrd*
geostat
gevcdf
gevinv
gevpdf
gevrnd*
gevstat
gpcdf
gpinv
gppdf
gprnd*
gpstat
hygecdf
hygeinv
hygepdf

hygernd*
hygestat
icdf
logncdf
logninv
lognpdf
lognrnd*
lognstat
mnpdf
nbincdf
nbininv
nbinpdf
nbinrnd*
nbinstat
ncfcdf
ncfinv
ncfpdf
ncfrnd*
ncfstat
nctcdf
nctinv
nctpdf
nctrnd*
nctstat
ncx2cdf
ncx2rnd*
ncx2stat
normcdf

norminv
normpdf
normrnd*
normstat
pdf
pearsrnd*
poisscdf
poissinv
poisspdf
poissrnd*
poisstat
randg
random
randsample*
raylcdf
raylinv
raylpdf
raylrnd*
raylstat
tcdf
tinvs
tpdf
trnd*
tstat
unidcdf
unidinv
unidpdf
unidrnd*



unidstat
unifcdf
unifinv
unifpdf
unifrnd*
unifstat
wblcdf
wblinv
wblpdf
wblrnd*
wblstat
<b>Cluster Analysis</b>
kmeans*
knnsearch* and knnsearch* of ExhaustiveSearcher
pdist2*
rangearch* and rangearch* of ExhaustiveSearcher
ExhaustiveSearcher*
<b>Regression</b>
glmval*
loadCompactModel
predict* of GeneralizedLinearModel and CompactGeneralizedLinearModel
predict* of LinearModel and CompactLinearModel
predict* of RegressionEnsemble, RegressionBaggedEnsemble, and CompactRegressionEnsemble
predict* of RegressionGP and CompactRegressionGP
predict* of RegressionLinear
predict* of RegressionSVM and CompactRegressionSVM
predict* of RegressionTree and CompactRegressionTree

random* of GeneralizedLinearModel and CompactGeneralizedLinearModel
random* of LinearModel and CompactLinearModel
GeneralizedLinearModel* and CompactGeneralizedLinearModel*
LinearModel* and CompactLinearModel*
RegressionEnsemble*, RegressionBaggedEnsemble*, and CompactRegressionEnsemble*
RegressionGP* and CompactRegressionGP*
RegressionLinear*
RegressionSVM* and CompactRegressionSVM*
RegressionTree* and CompactRegressionTree*
<b>Classification</b>
loadCompactModel
predict* of ClassificationECOC and CompactClassificationECOC
predict* of ClassificationEnsemble, ClassificationBaggedEnsemble, and CompactClassificationEnsemble
predict* of ClassificationDiscriminant and CompactClassificationDiscriminant
predict* of ClassificationKNN
predict* of ClassificationLinear
predict* of ClassificationSVM and CompactClassificationSVM
predict* of ClassificationTree and CompactClassificationTree
ClassificationECOC* and CompactClassificationECOC*
ClassificationEnsemble*, ClassificationBaggedEnsemble*, and CompactClassificationEnsemble*
ClassificationDiscriminant* and CompactClassificationDiscriminant*
ClassificationKNN*
ClassificationLinear*
ClassificationSVM* and CompactClassificationSVM*
ClassificationTree* and CompactClassificationTree*
<b>Dimensionality Reduction</b>

pca\*

## See Also

### Related Examples

- “Code Generation Workflows” on page 31-2
- “C Code Generation and Classification Learner” on page 31-8
- “Specify Variable-Sized Arguments for Code Generation” on page 31-32

## Specify Variable-Sized Arguments for Code Generation

This example shows how to generate code from a prediction function that has variable-sized input arguments. Specifically, the example creates a function that predicts labels based on a trained classification tree. At compile time, the number of observations to label and the number of subtrees are unknown.

Load Fisher's iris data set. Convert the labels to a character matrix.

```
load fisheriris
species = char(species);
```

Train a classification tree using the entire data set.

```
Mdl = fitctree(meas,species);
```

Mdl is a `ClassificationTree` model.

Save the trained classification tree to a file called `ClassTreeIris.mat` in your current folder.

```
MdlName = 'ClassTreeIris';
saveCompactModel(Mdl,MdlName);
```

In your current folder, declare a function called `mypredictTree.m` that:

- Accepts measurements with columns corresponding to `meas` and accepts valid name-value pair arguments
- Loads a trained classification tree
- Predicts labels and corresponding scores, node numbers, and class numbers from the loaded classification tree

```
function [label,score,node,cnum] = mypredictTree(x,savedmdl,varargin) %#codegen
%MPREDICTTREE Predict iris species using classification tree
% MPREDICTTREE predicts iris species for the n observations in the
% n-by-4 matrix x using the classification tree stored in the MAT-file
% whose name is in savedmdl, and then returns the predictions in the
% array label. Each row of x contains the lengths and widths of the petal
% and sepal of an iris (see the fisheriris data set). For other output
% argument descriptions, see the predict reference page.
CompactMdl = loadCompactModel(savedmdl);
```

```
[label,score,node,cnum] = predict(CompactMdl,x,varargin{:});
end
```

Declare code generation variables for the predictor data and subtrees from which to predict labels using `coder.typeof`. Specify that the predictor data variable is double-precision with the same number of columns as the predictor data used in training the model, but that the number of observations (rows) is arbitrary. Specify that the number of subtrees from which to predict labels is a double-precision row vector with an arbitrary number of columns.

```
p = numel(Mdl.PredictorNames);
x = coder.typeof(0,[Inf,p],[1,0]);
subtrees = coder.typeof(0,[1,Inf],[0,1]);
```

Generate a MEX function from `mypredictTree.m`. Because C uses static typing, `codegen` must determine the properties of all variables in MATLAB® files at compile time. To designate arguments as compile-time constants, use `coder.Constant`.

```
codegen mypredictTree.m -config:mex -args {x,coder.Constant(MdlName),coder.Constant('Subtrees')}
```

`codegen` generates the MEX file `mypredictTree_mex.mexw64` in your current folder. The file extension depends on your platform. Although `predict` accepts single-precision values, double-precision values, and 'all' for 'SubTrees', you can specify only double-precision values when you use the MEX function for prediction.

Predict labels for a random selection of 15 values from the training data using the generated MEX function and the subtree at pruning level 1. Compare the labels from the MEX function with those predicted by `predict`.

```
rng(1); % For reproducibility
Xnew = datasample(meas,15);
labelsMEX = mypredictTree_mex(Xnew,MdlName,'Subtrees',1)
labelsPREDICT = predict(Mdl,Xnew,'Subtrees',1)
```

```
labelsMEX =
```

```
15x1 cell array

    {'versicolor'}
    {'virginica' }
```

```
    {'setosa'   }
    {'setosa'   }
    {'setosa'   }
    {'setosa'   }
    {'setosa'   }
    {'versicolor'}
    {'versicolor'}
    {'versicolor'}
    {'versicolor'}
    {'virginica' }
    {'setosa'   }
    {'virginica' }
    {'setosa'   }

labelsPREDICT =

    15x10 char array

    'versicolor'
    'virginica '
    'setosa    '
    'setosa    '
    'setosa    '
    'setosa    '
    'setosa    '
    'setosa    '
    'versicolor'
    'versicolor'
    'versicolor'
    'versicolor'
    'virginica '
    'setosa    '
    'virginica '
    'setosa    '
```

The predicted labels are the same as the MEX function labels except for the data type. When the 'Subtrees' value is an arbitrary double at compile time and the response is a

character array, `codegen` must cast the predicted labels as a cell array of character vectors.

## See Also

`codegen` | `coder.typeof`

## Related Examples

- “Code Generation Workflows” on page 31-2
- “Code Generation Support, Usage Notes, and Limitations” on page 31-24
- “C Code Generation for Image Classifier”

## System Objects for Classification and Code Generation

Products used: Statistics and Machine Learning Toolbox™, MATLAB® Coder™, Simulink®, and Computer Vision Toolbox™.

This example shows how to generate C code from a MATLAB® System object™ that classifies images of digits using a trained classification model. This example also shows how to use the System object for classification in Simulink. A benefit of using System objects over a MATLAB functions is System objects are more appropriate for processing large amounts of streaming data. For more details, see “What Are System Objects?” (MATLAB).

This example is based on “C Code Generation for Image Classifier”, which is an alternative workflow to “Digit Classification Using HOG Features” (Computer Vision System Toolbox).

### Load Data

Load the `digitimages` data set from the `matlabroot/examples/stats` directory.

```
load(fullfile(matlabroot, 'examples', 'stats', 'digitimages.mat'))
```

`images` is a 28-by-28-by-3000 array of `uint16` integers. Each page is a raster image of a digit. Each element is a pixel intensity. Corresponding labels are in the 3000-by-1 numeric vector `Y`. For more details, enter `Description` at the command line.

Store the number of observations and number of predictor variables. Create a data partition that specifies to hold out 20% of the data. Extract training and test set indices from the data partition.

```
rng(1); % For reproducibility
n = size(images, 3);
p = numel(images(:, :, 1));
cvp = cvpartition(n, 'Holdout', 0.20);
idxTrn = training(cvp);
idxTest = test(cvp);
```

### Rescale Data

Rescale the pixel intensities so that they range in the interval  $[0,1]$  within each image. Specifically, suppose  $P_{ij}$  is pixel intensity  $j$  within image  $i$ . For image  $i$ , rescale all of its pixel intensities using this formula:



$$\hat{p}_{ij} = \frac{p_{ij} - \min_j(p_{ij})}{\max_j(p_{ij}) - \min_j(p_{ij})}.$$

```
X = double(images);

for i = 1:n
    minX = min(min(X(:, :, i)));
    maxX = max(max(X(:, :, i)));
    X(:, :, i) = (X(:, :, i) - minX) / (maxX - minX);
end
```

### Reshape Data

For code generation, the predictor data for training must be in a table of numeric variables or a numeric matrix.

Reshape the data to a matrix such that predictor variables correspond to columns and images to rows. Because `reshape` takes elements columnwise, transpose its result.

```
X = reshape(X, [p, n])';
```

### Train and Optimize Classification Models

Cross-validate an ECOC model of SVM binary learners and a random forest based on the training observations;. Use 5-fold cross-validation.

For the ECOC model, specify predictor standardization and optimize classification error over the ECOC coding design and the SVM box constraint. Explore all combinations of these values:

- For the ECOC coding design, use one-versus-one and one-versus-all.
- For the SVM box constraint, use three logarithmically-spaced values from 0.1 to 100 each. For all models, store the 5-fold cross-validated misclassification rates.

```
coding = {'onevsone' 'onevsall'};
boxconstraint = logspace(-1, 2, 3);
cvLossECOC = nan(numel(coding), numel(boxconstraint)); % For preallocation

for i = 1:numel(coding)
    for j = 1:numel(boxconstraint)
        t = templateSVM('BoxConstraint', boxconstraint(j), 'Standardize', true);
        CVMdl = fitcecoc(X(idxTrn, :), Y(idxTrn), 'Learners', t, 'KFold', 5, ...
            'Coding', coding{i});
```

```

cvLossECOC(i,j) = kfoldLoss(CVMdl);
fprintf('cvLossECOC = %f for model using %s coding and box constraint=%f\n',...
        cvLossECOC(i,j),coding{i},boxconstraint(j))
end
end

cvLossECOC = 0.058333 for model using onevsone coding and box constraint=0.100000
cvLossECOC = 0.057083 for model using onevsone coding and box constraint=3.162278
cvLossECOC = 0.050000 for model using onevsone coding and box constraint=100.000000
cvLossECOC = 0.120417 for model using onevsall coding and box constraint=0.100000
cvLossECOC = 0.121667 for model using onevsall coding and box constraint=3.162278
cvLossECOC = 0.127917 for model using onevsall coding and box constraint=100.000000

```

For the random forest, vary the maximum number of splits using the values in the sequence  $\{3^2, 3^3, \dots, 3^m\}$ .  $m$  is such that  $3^m$  is no greater than  $n - 1$ .

```

n = size(X,1);
m = floor(log(n - 1)/log(3));
maxNumSplits = 3.^(2:m);
cvLossRF = nan(numel(maxNumSplits));
for i = 1:numel(maxNumSplits)
    t = templateTree('MaxNumSplits',maxNumSplits(i));
    CVMdl = fitcensemble(X(idxTrn,:),Y(idxTrn),'Method','bag','Learners',t,...
        'KFold',5);
    cvLossRF(i) = kfoldLoss(CVMdl);
    fprintf('cvLossRF = %f for model using %d as the maximum number of splits\n',...
            cvLossRF(i),maxNumSplits(i))
end

cvLossRF = 0.323750 for model using 9 as the maximum number of splits
cvLossRF = 0.198333 for model using 27 as the maximum number of splits
cvLossRF = 0.075417 for model using 81 as the maximum number of splits
cvLossRF = 0.017083 for model using 243 as the maximum number of splits
cvLossRF = 0.012083 for model using 729 as the maximum number of splits
cvLossRF = 0.012083 for model using 2187 as the maximum number of splits

```

For each algorithm, determine the hyperparameter indices that yield the minimal misclassification rates.

```

minCVLossECOC = min(cvLossECOC(:))
linIdx = find(cvLossECOC == minCVLossECOC,1);
[bestI,bestJ] = ind2sub(size(cvLossECOC),linIdx);
bestCoding = coding{bestI}
bestBoxConstraint = boxconstraint(bestJ)

```

```

minCVLossRF = min(cvLossRF(:))
linIdx = find(cvLossRF == minCVLossRF,1);
[bestI,bestJ] = ind2sub(size(cvLossRF),linIdx);
bestMNS = maxNumSplits(bestI)

```

```

minCVLossECOC =

    0.0500

```

```

bestCoding =

    'onevsone'

```

```

bestBoxConstraint =

    100

```

```

minCVLossRF =

    0.0121

```

```

bestMNS =

    729

```

The random forest achieves a smaller cross-validated misclassification rate.

Train an ECOC model and a random forest using the training data. Supply the optimal hyperparameter combinations.

```

t = templateSVM('BoxConstraint',bestBoxConstraint,'Standardize',true);
MdlECOC = fitcecoc(X(idxTrn,:),Y(idxTrn),'Learners',t,'Coding',bestCoding);
t = templateTree('MaxNumSplits',bestMNS);
MdlRF = fitcensemble(X(idxTrn,:),Y(idxTrn),'Method','bag','Learners',t);

```

Create a variable for the test sample images, and use the trained models to predict test sample labels.

```
testImages = X(idxTest,:);  
testLabelsECOC = predict(MdlECOC,testImages);  
testLabelsRF = predict(MdlRF,testImages);
```

### Save Classification Model to Disk

MdlECOC and MdlRF are predictive classification models, but you must prepare them for code generation. Save MdlECOC and MdlRF to your present working directory using `saveCompactModel`.

```
saveCompactModel(MdlECOC,'DigitImagesECOC');  
saveCompactModel(MdlRF,'DigitImagesRF');
```

### Create System Object for Prediction

Create two System objects that, one for the ECOC model and the other for the random forest, that:

- Load the previously saved trained model using `loadCompactModel`
- Makes sequential predictions by the `step` method
- Enforces no size changes to the input data
- Enforces double-precision, scalar output

```
classdef ECOCClassifier < matlab.System  
    % ECOCCLASSIFIER Predict image labels from trained ECOC model  
    %  
    % ECOCCLASSIFIER loads the trained ECOC model from  
    % '|DigitImagesECOC.mat|', and predicts labels for new observations  
    % based on the trained model. The ECOC model in  
    % '|DigitImagesECOC.mat|' was cross-validated using the training data  
    % in the sample data |digitimages.mat|.   
  
    properties(Access = private)  
        CompactMdl % The compacted, trained ECOC model  
    end  
  
    methods(Access = protected)  
  
        function setupImpl(obj)  
            % Load ECOC model from file  
            obj.CompactMdl = loadCompactModel('DigitImagesECOC');  
        end
```

```

function y = stepImpl(obj,u)
    y = predict(obj.CompactMdl,u);
end

function flag = isInputSizeLockedImpl(obj,index)
    % Return true if input size is not allowed to change while
    % system is running
    flag = true;
end

function dataout = getOutputDataTypeImpl(~)
    dataout = 'double';
end

function sizeout = getOutputSizeImpl(~)
    sizeout = [1 1];
end
end
end

classdef RFClassifier < matlab.System
    % RFCLASSIFIER Predict image labels from trained random forest
    %
    % RFCLASSIFIER loads the trained random forest from
    % |'DigitImagesRF.mat'|, and predicts labels for new observations based
    % on the trained model. The random forest in |'DigitImagesRF.mat'|
    % was cross-validated using the training data in the sample data
    % |digitimages.mat|.

    properties(Access = private)
        CompactMdl % The compacted, trained random forest
    end

    methods(Access = protected)

        function setupImpl(obj)
            % Load random forest from file
            obj.CompactMdl = loadCompactModel('DigitImagesRF');
        end

        function y = stepImpl(obj,u)
            y = predict(obj.CompactMdl,u);
        end
    end
end

```

```
end

function flag = isInputSizeLockedImpl(obj,index)
    % Return true if input size is not allowed to change while
    % system is running
    flag = true;
end

function dataout = getOutputDataTypeImpl(~)
    dataout = 'double';
end

function sizeout = getOutputSizeImpl(~)
    sizeout = [1 1];
end
end
end
```

For System object basic requirements, see “Define Basic System Objects” (MATLAB).

### Declare Prediction Functions for Code Generation

Declare two MATLAB functions called `predictDigitECOC`.m and `predictDigitRFS`.m. The functions should:

- Include the code generation directive  `%#codegen`.
- Accept image data commensurate with `X`.
- Predict labels using the `ECOCClassifier` and `RFCClassifier` System objects, respectively.
- Return predicted labels.

```
function label = predictDigitECOC(X) %#codegen
%PREDICTDIGITECOC Classify digit in image using ECOC Model System object
%   PREDICTDIGITECOC classifies the 28-by-28 images in the rows of X
%   using the compact ECOC model in the System object ECOCClassifier, and
%   then returns class labels in label.
classifier = ECOCClassifier;
label = step(classifier,X);
end
```

```

function label = predictDigitRFSO(X) %#codegen
%PREDICTDIGITRFSO Classify digit in image using RF Model System object
%   PREDICTDIGITRFSO classifies the 28-by-28 images in the rows of X
%   using the compact random forest in the System object RFClassifier, and
%   then returns class labels in label.
classifier = RFClassifier;
label = step(classifier,X);
end

```

### Compile MATLAB Function to MEX File

Compile the prediction function that achieves better test-sample accuracy to a MEX file using `codegen`. Specify the test set images using the `-args` argument.

```

if(minCVLossECOC <= minCVLossRF)
    codegen predictDigitECOCSO -args testImages
else
    codegen predictDigitRFSO -args testImages
end

```

Verify that the generated MEX file produces the same predictions as the MATLAB function.

```

if(minCVLossECOC <= minCVLossRF)
    mexLabels = predictDigitECOCSO_mex(testImages);
    verifyMEX = sum(mexLabels == testLabelsECOC) == numel(testLabelsECOC)
else
    mexLabels = predictDigitRFSO_mex(testImages);
    verifyMEX = sum(mexLabels == testLabelsRF) == numel(testLabelsRF)
end

```

```

verifyMEX =

    logical

     1

```

`verifyMEX` is 1, which indicates that the predictions made by the generated MEX file and the corresponding MATLAB function are the same.

## Predict Labels Using System Objects in Simulink

Create a video file that displays the test-set images frame by frame.

```
v = VideoWriter('testImages.avi','Uncompressed AVI');
v.FrameRate = 1;
open(v);
dim = sqrt(p)*[1 1];
for j = 1:size(testImages,1)
    writeVideo(v,reshape(testImages(j,:),dim));
end
close(v);
```

Declare a function called `scalePixelIntensities.m` that converts RGB images to grayscale, and then scales the resulting pixel intensities so that their values are in the interval `[0,1]`.

```
function x = scalePixelIntensities(imdat)
%SCALEPIXELINTENSITIES Scales image pixel intensities
% SCALEPIXELINTENSITIES scales the pixel intensities of the image such
% that the result x is a row vector of values in the interval [0,1].

imdat = rgb2gray(imdat);

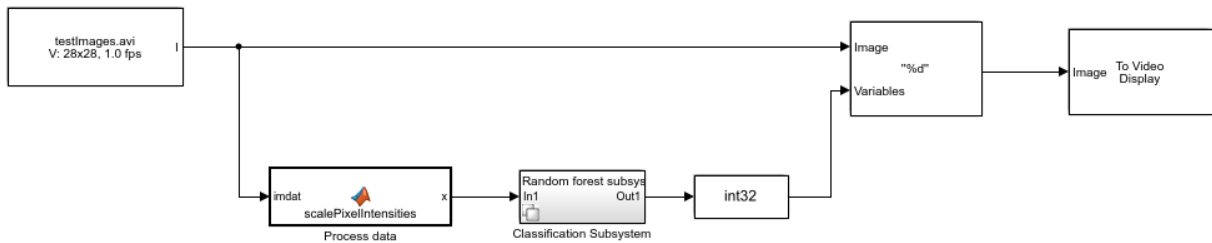
minimdat = min(min(imdat));
maximdat = max(max(imdat));
x = (imdat - minimdat)/(maximdat - minimdat);

end
```

Load the Simulink® model `slexClassifyAndDisplayDigitImages.slx` located in `mlr/examples/stats`, where `mlr` is the MATLAB root folder.

```
mlr = matlabroot;
SimMdlName = 'slexClassifyAndDisplayDigitImages';
pathToModel = fullfile(mlr,'examples','stats',SimMdlName);
open_system(pathToModel);
```





The figure displays the Simulink® model. At the beginning of simulation, the From Multimedia File block loads the video file of the test-set images. For each image in the video:

- 1 The From Multimedia File block converts and outputs the image to a 28-by-28 matrix of pixel intensities.
- 2 The Process Data block scales the pixel intensities using `scalePixelIntensities.m`, and outputs a 1-by-784 vector of scaled intensities.
- 3 The Classification Subsystem block predicts labels given the processed image data. The block chooses the System object that minimizes classification error. In this case, the block chooses the random forest. The block outputs a double-precision scalar label.
- 4 The Data Type Conversion block converts the label to an `int32` scalar.
- 5 The Insert Text block embeds the predicted label on the current frame.
- 6 The To Video Display block displays the annotated frame.

Simulate the model.

```
sim(SimMdlName);
```



The model displays all 600 test-set images and its prediction quickly; the last image remains in the video display. You can generate predictions and display them with corresponding images one by one by pressing the **Step Forward** button instead.

## See Also

### Related Examples

- “Code Generation Workflows” on page 31-2
- “C Code Generation for Image Classifier”
- “Predict Class Labels Using MATLAB Function Block” on page 31-18
- “Predict Class Labels Using Stateflow” on page 31-52

## Code Generation Workflow for Exhaustive Searcher

This example shows how to generate code for finding the nearest neighbor using an exhaustive searcher object at the command line. This example shows two different methods depending on the way you use the object: load the object by using `loadCompactModel` in a function you declare, or pass a compile-time constant object to the generated code.

### Train Exhaustive Nearest Neighbor Searcher

Load Fisher's iris data set.

```
load fisheriris
```

Remove five irises randomly from the predictor data to use as a query set.

```
rng('default');           % For reproducibility
n = size(meas,1);         % Sample size
qIdx = randsample(n,5);   % Indices of query data
X = meas(~ismember(1:n,qIdx),:);
Y = meas(qIdx,:);
```

Prepare an exhaustive nearest neighbor searcher using the training data. Specify the 'Distance' and 'P' name-value pair arguments to use the Minkowski distance with an exponent of 1 for finding the nearest neighbor later.

```
Mdl = ExhaustiveSearcher(X,'Distance','minkowski','P',1);
```

Find the index of the training data (X) that is the nearest neighbor of each point in the query data (Y).

```
Idx = knnsearch(Mdl,Y);
```

### Set Up C Compiler

To generate C code, you must have access to a C compiler, and the compiler must be configured properly. For more details, see “Setting Up Your C Compiler” (MATLAB Coder). Set up the compiler for MEX-code generation.

```
mex -setup
```

```
MEX configured to use 'Microsoft Visual C++ 2015 (C)' for C language compilation.
Warning: The MATLAB C and Fortran API has changed to support MATLAB
variables with more than 2^32-1 elements. You will be required
```

to update your code to utilize the new API.  
You can find more information about this at:  
[http://www.mathworks.com/help/matlab/matlab\\_external/upgrading-mex-files-to-use-64](http://www.mathworks.com/help/matlab/matlab_external/upgrading-mex-files-to-use-64)

To choose a different C compiler, execute one from the following:  
For MinGW64 Compiler (C): `mex -setup:B:\matlab\bin\win64\mexopts\mingw64.xml C`  
For Microsoft Visual C++ 2013 (C): `mex -setup:B:\matlab\bin\win64\mexopts\msvc2013.xml C`  
For Intel C++ Composer XE 2013 with Microsoft Visual Studio 2013 (C): `mex -setup:B:\matlab\bin\win64\mexopts\intel_cxx.xml C`  
For Microsoft Visual C++ 2012 (C): `mex -setup:B:\matlab\bin\win64\mexopts\msvc2012.xml C`  
For Microsoft Visual C++ 2015 (C): `mex -setup:B:\matlab\bin\win64\mexopts\msvc2015.xml C`

To choose a different language, execute one from the following:  
`mex -setup C++`  
`mex -setup FORTRAN`

### Generate Code Using `saveCompactModel` and `loadCompactModel`

Generate code that loads an exhaustive searcher, takes query data as an input argument, and then finds the nearest neighbor.

Save the exhaustive searcher to a file using `saveCompactModel`.

```
saveCompactModel(Mdl, 'searcherModel')
```

`saveCompactModel` saves the model to the MATLAB binary file `searcherModel.mat` as a structure array in the current folder.

Declare a function `myknnsearch1` that takes query data as an input argument. Within the function, load the searcher object by using `loadCompactModel`, and then pass the loaded model to `knnsearch`.

```
function idx = myknnsearch1(x) %#codegen
Mdl = loadCompactModel('searcherModel');
idx = knnsearch(Mdl,x);
end
```

Be sure to type the `%#codegen` compiler directive somewhere in the function.

Generate code for `myknnsearch1` by using `codegen`. Specify the data type and dimension of the input argument by using `coder.typeof`.

```
codegen myknnsearch1.m -args {coder.typeof(Y,[Inf,4],[1,0])}
```

Warning: C Compiler produced warnings. See the build log for further details.

Code generation successful (with warnings): To view the report, open('codegen\mex\myknn

Verify that `myknnsearch1` and the MEX file return the expected indices by passing the query data (`Y`).

```
myIdx1 = myknnsearch1(Y);
myIdx1_mex = myknnsearch1_mex(Y);
verifyMEX1 = isequal(Idx,myIdx1,myIdx1_mex)
```

```
verifyMEX1 =
```

```
    logical
```

```
    1
```

`isequal` returns logical 1 (true) if all the inputs are equal. This comparison confirms that the MEX file returns the expected results.

### Generate Code with Constant Folded Model Object

Declare a function `myknnsearch2` that takes both an exhaustive searcher model and query data as input arguments instead of loading the model in the function.

```
function idx = myknnsearch2(Mdl,x) %#codegen
idx = knnsearch(Mdl,x);
end
```

To generate code that takes the model object as well as the query data, designate the model object as a compile-time constant by using `coder.Constant` and include the constant folded model object in the `-args` value of `codegen`.

```
codegen myknnsearch2.m -args {coder.Constant(Mdl),coder.typeof(Y,[Inf,4],[1,0])}
```

Warning: C Compiler produced warnings. See the build log for further details.

Code generation successful (with warnings): To view the report, open('codegen\mex\myknn

Verify that `myknnsearch2` and the MEX file return the expected results.

```
myIdx2 = myknnsearch2(Mdl,Y);  
myIdx2_mex = myknnsearch2_mex(Mdl,Y);  
verifyMEX2 = isequal(Idx,myIdx2,myIdx2_mex)
```

```
verifyMEX2 =  
  
    logical  
  
    1
```

### Generate Code with Name-Value Pair Arguments

Declare a function that takes a model object, query data, and name-value pair arguments.

```
function idx = myknnsearch3(Mdl,x,varargin) %#codegen  
idx = knnsearch(Mdl,x,varargin{:});  
end
```

To generate code that allows a user-defined exponent for the Minkowski distance, include `{coder.Constant('P'),0}` in the `-args` value of `codegen`.

```
codegen myknnsearch3.m -args {coder.Constant(Mdl),coder.typeof(Y,[Inf,4],[1,0]),coder.C
```

```
Warning: C Compiler produced warnings. See the build log for further details.
```

```
Code generation successful (with warnings): To view the report, open('codegen\mex\myknn
```

Verify that `myknnsearch3` and the MEX file return the expected results.

```
newIdx = knnsearch(Mdl,Y,'P',2);  
myIdx3 = myknnsearch3(Mdl,Y,'P',2);  
myIdx3_mex = myknnsearch3_mex(Mdl,Y,'P',2);  
verifyMEX3 = isequal(newIdx,myIdx3,myIdx3_mex)
```

```
verifyMEX3 =  
  
    logical
```

1

## See Also

`ExhaustiveSearcher` | `codegen` | `coder.Constant` | `coder.typeof` | `knnsearch`  
| `rangesearch`

## More About

- “Code Generation Workflows” on page 31-2
- “Code Generation Support, Usage Notes, and Limitations” on page 31-24
- “Specify Variable-Sized Arguments for Code Generation” on page 31-32
- “Getting Started with MATLAB Coder” (MATLAB Coder)

## Predict Class Labels Using Stateflow

This example shows how to use a Stateflow® chart for label prediction. The example trains a discriminant analysis model for the Fisher iris data set by using `fitcdiscr`, and declares a function for code generation that loads the trained model and predicts labels for new data. The Stateflow chart in this example accepts streaming data and predicts labels using the function you declare.

Fisher's iris data set, which is included in Statistics and Machine Learning Toolbox™, contains species (`species`) and measurements (`meas`) on sepal length, sepal width, petal length, and petal width for 150 iris specimens. The data set contains 50 specimens from each of three species: *setosa*, *versicolor*, and *virginica*.

Load the Fisher iris data set.

```
load fisheriris
```

Convert `species` to an index vector where 1, 2, and 3 correspond to *setosa*, *versicolor*, and *virginica*, respectively.

```
species = grp2idx(species);
```

Partition the data into a training set and a test set.

```
rng('default') % For reproducibility
idx1 = randperm(150,75)';
idx2 = setdiff((1:150)',idx1);
X = meas(idx1,:);
Y = species(idx1,:);
trainX = meas(idx2,:);
trainY = species(idx2,:);
```

Use `trainX` and `trainY` to train a model, and use `X` and `Y` to test the trained model.

Train a quadratic discriminant analysis model.

```
Mdl = fitcdiscr(trainX,trainY,'DiscrimType','quadratic');
```

`Mdl` is a `ClassificationDiscriminant` model. At the command line, you can use `Mdl` to make predictions for new observations. However, you cannot use `Mdl` as an input argument in a function for code generation. Prepare `Mdl` to be loaded within the function by using `saveCompactModel`.



```
saveCompactModel(Mdl, 'DiscrIris');
```

`saveCompactModel` compacts `Mdl` and saves it in the MAT-file `DiscrIris.mat`.

To display the predicted species in the display box of the Stateflow® model, define an enumeration class by using a `classdef` block in the MATLAB file `IrisSpecies.m`.

```
classdef IrisSpecies < Simulink.IntEnumType
    enumeration
        Setosa(1)
        Versicolor(2)
        Virginica(3)
    end
end
```

For details about enumerated data, see “Define Enumerated Data in a Chart” (Stateflow).

Declare a function named `mypredict.m` that predicts the iris species from new measurement data by using the trained model. The function should:

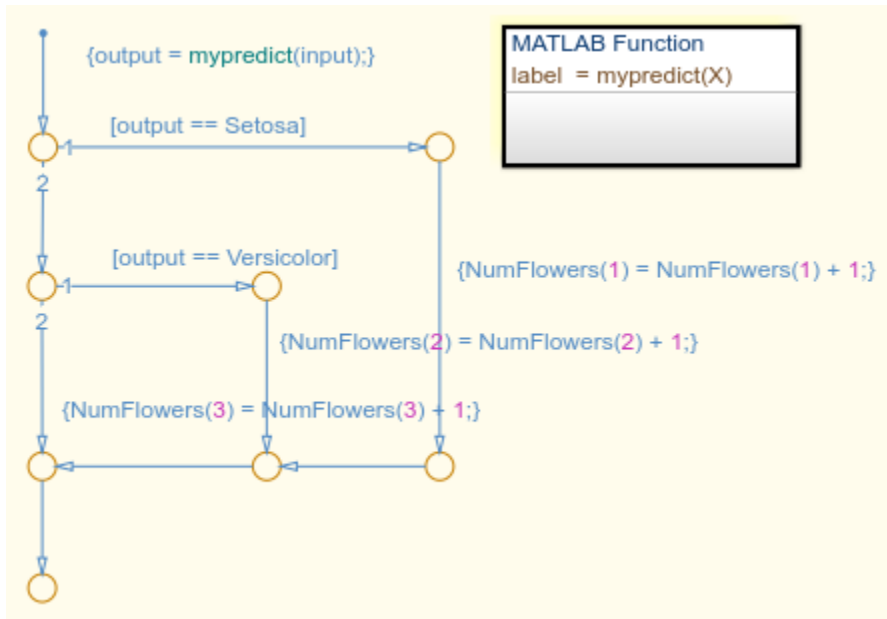
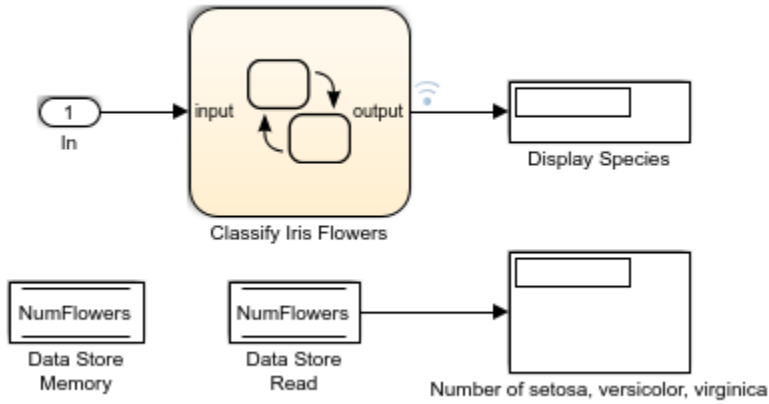
- Include the code generation directive `codegen` somewhere in the function.
- Accept iris measurement data. The data must be consistent with `X` except for the number of rows.
- Load `DiscrIris.mat` using `loadCompactModel`.
- Return predicted iris species.

```
function label = mypredict(X) %codegen
%MYPREDICT Predict species of iris flowers using discriminant model
% mypredict predicts species of iris flowers using the compact
% discriminant model in the file DiscrIris.mat. Rows of X correspond to
% observations and columns correspond to predictor variables. label is
% the predicted species.
mdl = loadCompactModel('DiscrIris');
labelTemp = predict(mdl,X);
label = IrisSpecies(labelTemp);
end
```

Open the Simulink® model `sf_countflowers.slx`.

```
sfName = 'sf_countflowers';
open_system(sfName);
```

Iris Flower Classification Using Discriminant Model



The figures display the Simulink® model and the flow graph contained in the Stateflow® chart. When the input node detects measurement data, it directs the data into the chart. The chart then predicts a species of iris flower and counts the number of flowers for each species. The chart returns the predicted species to the workspace and displays the species within the model, one at a time. The data store memory block NumFlowers stores the number of flowers for each species.

The chart expects to receive input data as a structure array called `fisheririsInput` containing these fields:

- `time` - The points in time at which the observations enter the model. In the example, the duration includes the integers from 0 through 74. The orientation of `time` must correspond to the observations in the predictor data. So, for this example, `time` must be a column vector.
- `signals` - A 1-by-1 structure array describing the input data and containing the fields `values` and `dimensions`. The `values` field is a matrix of predictor data. The `dimensions` field is the number of predictor variables.

Create an appropriate structure array for iris flower measurements.

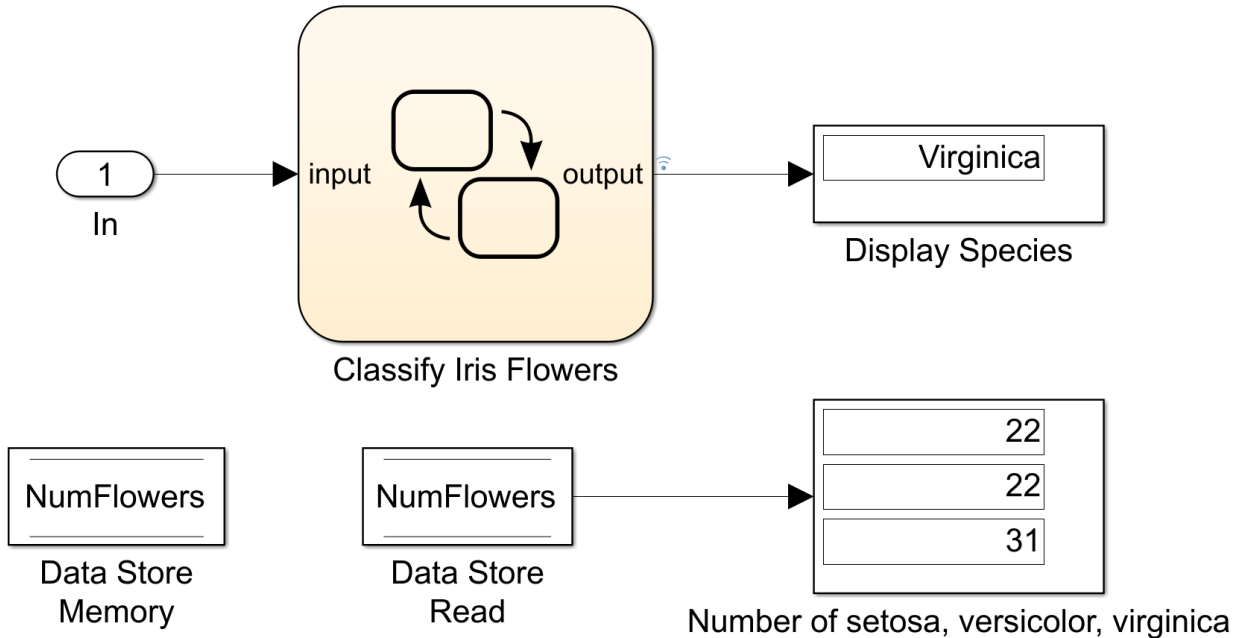
```
fisheririsInput.time = [0:74]';  
fisheririsInput.signals.dimensions = 4;  
fisheririsInput.signals.values = X;
```

You can change the name from `fisheririsInput`, and then specify the new name in the model. However, Stateflow® expects the structure array to contain the described field names. For more details, see “Create Data Structures for Root-Level Inputs” (Simulink).

Simulate the model.

```
sim(sfName)
```

## Iris Flower Classification Using Discriminant Model



The figure shows the model after it processes all observations in `fisheririsInput`, one at a time. The predicted species of `X(75, :)` is `virginica`. The number of `setosa`, `versicolor`, and `virginica` in `X` is 22, 22, and 31, respectively.

The variable `logout` appears in the workspace. `logout` is a `SimulinkData.Dataset` object containing the predicted species. Extract the predicted species data from the simulation log.

```
labelSF = logout.getElement(1).Values.Data;
```

Predict species at the command line using `CompactClassificationDiscriminant.predict`.

```
labelCMD = predict(Mdl,X);
```

Compare the predicted species returned by `sf_countflowers` to those returned by calling `predict` at the command line.

```
isequal(labelCMD,labelSF)
```

```
ans =  
  
    logical  
  
     1
```

`isequal` returns logical 1 (true) if all the inputs are equal. This comparison confirms that `sf_countflowers` returns the expected results.

If you also have a Simulink® Coder™ license, then you can generate C code from `sf_countflowers.slx` in Simulink® or from the command line using `rtwbuild`. For more details, see “Generate C Code for a Model” (Simulink Coder).

## See Also

`ClassificationDiscriminant` | `fitcdiscr` | `predict` | `rtwbuild`

## More About

- “Chart Programming” (Stateflow)
- “Code Generation Support, Usage Notes, and Limitations” on page 31-24
- “Code Generation Workflows” on page 31-2
- “Predict Class Labels Using MATLAB Function Block” on page 31-18
- “System Objects for Classification and Code Generation” on page 31-36



## Functions — Alphabetical List

---

## addedvarplot

Added variable plot

### Syntax

```
addedvarplot(X, y, num, inmodel)
addedvarplot(X, y, num, inmodel, stats)
```

### Description

`addedvarplot(X, y, num, inmodel)` displays an added variable plot using the predictive terms in `X`, the response values in `y`, the added term in column `num` of `X`, and the model with current terms specified by `inmodel`. `X` is an  $n$ -by- $p$  matrix of  $n$  observations of  $p$  predictive terms. `y` is vector of  $n$  response values. `num` is a scalar index specifying the column of `X` with the term to be added. `inmodel` is a logical vector of  $p$  elements specifying the columns of `X` in the current model. By default, all elements of `inmodel` are `false`.

---

**Note** `addedvarplot` automatically includes a constant term in all models. Do not enter a column of 1s directly into `X`.

---

`addedvarplot(X, y, num, inmodel, stats)` uses the `stats` output from the `stepwisefit` function to improve the efficiency of repeated calls to `addedvarplot`. Otherwise, this syntax is equivalent to the previous syntax.

Added variable plots are used to determine the unique effect of adding a new term to a multilinear model. The plot shows the relationship between the part of the response unexplained by terms already in the model and the part of the new term unexplained by terms already in the model. The “unexplained” parts are measured by the residuals of the respective regressions. A scatter of the residuals from the two regressions forms the added variable plot.

In addition to the scatter of residuals, the plot produced by `addedvarplot` shows 95% confidence intervals on predictions from the fitted line. The fitted line has intercept zero



because, under typical linear model assumptions, both of the plotted variables have mean zero. The slope of the fitted line is the coefficient that the new term would have if it were added to the model with terms `inmodel`.

Added variable plots are sometimes known as partial regression leverage plots.

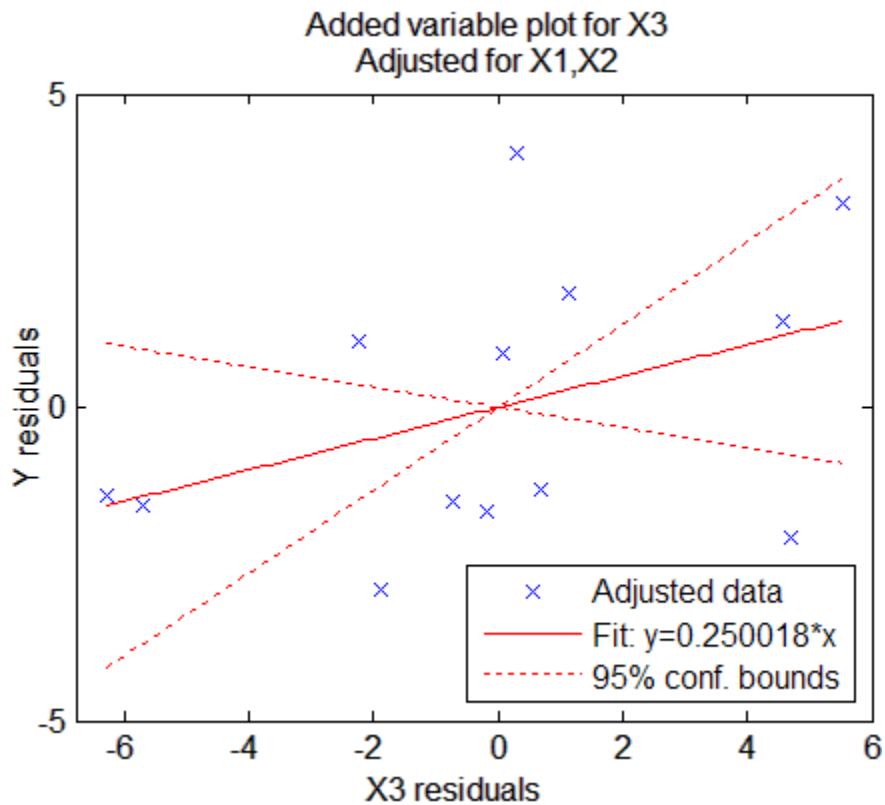
## Examples

Load the data in `hald.mat`, which contains observations of the heat of reaction of various cement mixtures:

```
load hald
whos
  Name          Size      Bytes   Class   Attributes
  Description   22x58    2552    char
  hald          13x5      520     double
  heat         13x1      104     double
  ingredients   13x4      416     double
```

Create an added variable plot to investigate the addition of the third column of `ingredients` to a model consisting of the first two columns:

```
inmodel = [true true false false];
addedvarplot(ingredients,heat,3,inmodel)
```



The wide scatter and the low slope of the fitted line are evidence against the statistical significance of adding the third column to the model.

## See Also

`stepwise` | `stepwisefit`

Introduced before R2006a

# addK

**Class:** clustering.evaluation.ClusterCriterion

**Package:** clustering.evaluation

Evaluate additional numbers of clusters

## Syntax

```
eva_out = addK(eva, klist)
```

## Description

`eva_out = addK(eva, klist)` returns a clustering evaluation object `eva_out` that contains the evaluation data stored in the input object `eva`, plus additional evaluation data for the proposed number of clusters specified in `klist`.

## Input Arguments

**eva** — Clustering evaluation data

clustering evaluation object

Clustering evaluation data, specified as a clustering evaluation object. Create a clustering evaluation object using `evalclusters`.

**klist** — Additional numbers of clusters to evaluate

vector of positive integer values

Additional numbers of clusters to evaluate, specified as a vector of positive integer values. If any values in `klist` overlap with clustering solutions already evaluated in the input object `eva`, then `addK` ignores the overlapping values.

## Output Arguments

### **eva\_out** — Updated clustering evaluation data

clustering evaluation object

Updated clustering evaluation data, returned as a clustering evaluation object. `eva_out` contains data on the proposed clustering solutions included in the input clustering evaluation object `eva`, plus data on the additional proposed numbers of clusters specified in `klist`.

For all clustering evaluation object classes, `addK` updates the `InspectedK` and `CriterionValues` properties to include the proposed clustering solutions specified in `klist` and their corresponding criterion values. `addK` might also update the `OptimalK` and `OptimalY` properties to reflect the new optimal number of clusters and optimal clustering solution.

For certain cluster evaluation objects classes, `addK` might also update the following additional property values:

- For gap evaluation objects — `LogW`, `ExpectedLogW`, `StdLogW`, and `SE`
- For silhouette evaluation objects — `ClusterSilhouettes`

## Examples

### **Evaluate Additional Numbers of Clusters**

Create a clustering evaluation object using `evalclusters`, then use `addK` to evaluate additional numbers of clusters.

Load the sample data.

```
load fisheriris
```

The data contains length and width measurements from the sepals and petals of three species of iris flowers.

Cluster the flower measurement data using `kmeans`, and use the Calinski-Harabasz criterion to evaluate proposed solutions of one through five clusters.

```
eva = evalclusters(meas, 'kmeans', 'calinski', 'klist', 1:5)

eva =

    CalinskiHarabaszEvaluation with properties:

        NumObservations: 150
           InspectedK: [1 2 3 4 5]
      CriterionValues: [NaN 513.9245 561.6278 530.4871 456.1279]
         OptimalK: 3
```

The clustering evaluation object `eva` contains data on each proposed clustering solution. The returned value of `OptimalK` indicates that the optimal solution is three clusters.

Evaluate proposed solutions of 6 through 10 clusters using the same criteria. Add these evaluations to the original clustering evaluation object `eva`.

```
eva = addK(eva, 6:10)

eva =

    CalinskiHarabaszEvaluation with properties:

        NumObservations: 150
           InspectedK: [1 2 3 4 5 6 7 8 9 10]
      CriterionValues: [1x10 double]
         OptimalK: 3
```

The updated values for `InspectedK` and `CriterionValues` show that `eva` now evaluates proposed solutions of 1 through 10 clusters. The `OptimalK` value still equals 3, indicating that three clusters remain the optimal solution.

## See Also

`clustering.evaluation.CalinskiHarabaszEvaluation` |  
`clustering.evaluation.DaviesBouldinEvaluation` |  
`clustering.evaluation.GapEvaluation` |  
`clustering.evaluation.SilhouetteEvaluation` | `evalclusters`

## addlevels

Add levels to nominal or ordinal arrays

---

**Note** The `nominal` and `ordinal` array data types might be removed in a future release. To represent ordered and unordered discrete, nonnumeric data, use the MATLAB® `categorical` data type instead.

---

### Syntax

```
B = addlevels(A,newlevels)
```

### Description

`B = addlevels(A,newlevels)` adds new levels specified by `newlevels` to the nominal or ordinal array `A`. `addlevels` adds the new levels at the end of the list of possible levels in `A`, but does not modify the value of any element. `B` does not contain elements at the new levels.

### Examples

#### Add Levels To A Nominal Array

Add levels for additional species to Fisher's iris data.

Create a nominal array of the existing species in Fisher's iris data.

```
load fisheriris
species = nominal(species);
getlevels(species)
```

```
ans =
```

```
1x3 nominal array
    setosa    versicolor    virginica
```

Add two additional species.

```
species = addlevels(species, {'spuria', 'ruthenica'});
getlevels(species)
```

```
ans =
```

```
1x5 nominal array
    setosa    versicolor    virginica    spuria    ruthenica
```

Even though there are new levels, there are no elements in `species` that are in these new levels.

```
sum(species=='spuria')
sum(species=='ruthenica')
```

```
ans =
```

```
0
```

```
ans =
```

```
0
```

- “Add and Drop Category Levels” on page 2-25

## Input Arguments

**A** — Nominal or ordinal array

nominal array | ordinal array

Nominal or ordinal array, specified as a `nominal` or `ordinal` array object created using `nominal` or `ordinal`.

**newlevels** — Levels to add

cell array of character vectors | 2-D character matrix

Levels to add to the input `nominal` or `ordinal` array, specified as a cell array of character vectors or 2-D character matrix.

Data Types: `char` | `cell`

## Output Arguments

**B** — Nominal or ordinal array

`nominal` array | `ordinal` array

Nominal or ordinal array, returned as a `nominal` or `ordinal` array object.

## See Also

`droplevels` | `mergelevels` | `nominal` | `ordinal` | `reorderlevels`

## Topics

“Add and Drop Category Levels” on page 2-25

`nominal`

`ordinal`

Introduced in R2007a



# addlistener

**Class:** grandstream

Add listener for event

## Syntax

```
e1 = addlistener(hsource, 'eventname', callback)
e1 = addlistener(hsource, property, 'eventname', callback)
```

## Description

`e1 = addlistener(hsource, 'eventname', callback)` creates a listener for the event named `eventname`, the source of which is handle object `hsource`. If `hsource` is an array of source handles, the listener responds to the named event on any handle in the array. `callback` is a function handle that is invoked when the event is triggered.

`e1 = addlistener(hsource, property, 'eventname', callback)` adds a listener for a property event. `eventname` must be 'PreGet', 'PostGet', 'PreSet', or 'PostSet'. `property` must be either a property name or cell array of property names, or a `meta.property` or array of `meta.property`. The properties must belong to the class of `hsource`. If `hsource` is scalar, `property` can include dynamic properties.

For all forms, `addlistener` returns an `event.listener`. To remove a listener, delete the object returned by `addlistener`. For example, `delete(e1)` calls the handle class `delete` method to remove the listener and delete it from the workspace.

## See Also

`delete` | `dynamicprops` | `event.listener` | `events` | `meta.property` | `notify` | `grandstream` | `reset`

## anova

**Class:** GeneralizedLinearMixedModel

Analysis of variance for generalized linear mixed-effects model

## Syntax

```
stats = anova(glme)
stats = anova(glme, Name, Value)
```

## Description

`stats = anova(glme)` returns a table, `stats`, that contains the results of  $F$ -tests to determine if all coefficients representing each fixed-effects term in the generalized linear mixed-effects model `glme` are equal to 0.

`stats = anova(glme, Name, Value)` returns a table, `stats`, using additional options specified by one or more `Name, Value` pair arguments. For example, you can specify the method used to compute the approximate denominator degrees of freedom for the  $F$ -tests.

## Input Arguments

**glme** — Generalized linear mixed-effects model

GeneralizedLinearMixedModel object

Generalized linear mixed-effects model, specified as a GeneralizedLinearMixedModel object. For properties and methods of this object, see GeneralizedLinearMixedModel.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

**DFMethod** — Method for computing approximate denominator degrees of freedom`'residual'` (default) | `'none'`

Method for computing approximate denominator degrees of freedom to use in the  $F$ -test, specified as the comma-separated pair consisting of `'DFMethod'` and one of the following.

Value	Description
<code>'residual'</code>	The degrees of freedom are assumed to be constant and equal to $n - p$ , where $n$ is the number of observations and $p$ is the number of fixed effects.
<code>'none'</code>	All degrees of freedom are set to infinity.

The denominator degrees of freedom for the  $F$ -statistic correspond to the column `DF2` in the output structure `stats`.

Example: `'DFMethod', 'none'`

## Output Arguments

**stats** — Results of  $F$ -tests for fixed-effects terms

table

Results of  $F$ -tests for fixed-effects terms, returned as a table with one row for each fixed-effects term in `glme` and the following columns.

Column Name	Description
Term	Name of the fixed-effects term
FStat	$F$ -statistic for the term
DF1	Numerator degrees of freedom for the $F$ -statistic
DF2	Denominator degrees of freedom for the $F$ -statistic
pValue	$p$ -value for the term

Each fixed-effects term is a continuous variable, a grouping variable, or an interaction between two or more continuous or grouping variables. For each fixed-effects term,

`anova` performs an  $F$ -test (marginal test) to determine if all coefficients representing the fixed-effects term are equal to 0.

To perform tests for the type III hypothesis, when fitting the generalized linear mixed-effects model `fitglme`, you must use the `'effects'` contrasts for the `'DummyVarCoding'` name-value pair argument.

## Examples

### F-Tests for Fixed Effects

Load the sample data.

```
load mfr
```

This simulated data is from a manufacturing company that operates 50 factories across the world, with each factory running a batch process to create a finished product. The company wants to decrease the number of defects in each batch, so it developed a new manufacturing process. To test the effectiveness of the new process, the company selected 20 of its factories at random to participate in an experiment: Ten factories implemented the new process, while the other ten continued to run the old process. In each of the 20 factories, the company ran five batches (for a total of 100 batches) and recorded the following data:

- Flag to indicate whether the batch used the new process (`newprocess`)
- Processing time for each batch, in hours (`time`)
- Temperature of the batch, in degrees Celsius (`temp`)
- Categorical variable indicating the supplier (A, B, or C) of the chemical used in the batch (`supplier`)
- Number of defects in the batch (`defects`)

The data also includes `time_dev` and `temp_dev`, which represent the absolute deviation of time and temperature, respectively, from the process standard of 3 hours at 20 degrees Celsius.

Fit a generalized linear mixed-effects model using `newprocess`, `time_dev`, `temp_dev`, and `supplier` as fixed-effects predictors. Include a random-effects term for `intercept`

grouped by `factory`, to account for quality differences that might exist due to factory-specific variations. The response variable `defects` has a Poisson distribution, and the appropriate link function for this model is `log`. Use the Laplace fit method to estimate the coefficients. Specify the dummy variable encoding as `'effects'`, so the dummy variable coefficients sum to 0.

The number of defects can be modeled using a Poisson distribution

$$\text{defect}_{ij} \sim \text{Poisson}(\mu_{ij})$$

This corresponds to the generalized linear mixed-effects model

$$\log \mu_{ij} = \beta_0 + \beta_1 \text{newprocess}_{ij} + \beta_2 \text{time.dev}_{ij} + \beta_3 \text{temp.dev}_{ij} + \beta_4 \text{supplier.C}_{ij} + \beta_5 \text{supplier.B}_{ij} + b_i,$$

where

- `defectsij` is the number of defects observed in the batch produced by factory *i* during batch *j*.
- $\mu_{ij}$  is the mean number of defects corresponding to factory *i* (where  $i = 1, 2, \dots, 20$ ) during batch *j* (where  $j = 1, 2, \dots, 5$ ).
- `newprocessij`, `time.devij`, and `temp.devij` are the measurements for each variable that correspond to factory *i* during batch *j*. For example, `newprocessij` indicates whether the batch produced by factory *i* during batch *j* used the new process.
- `supplier.Cij` and `supplier.Bij` are dummy variables that use effects (sum-to-zero) coding to indicate whether company C or B, respectively, supplied the process chemicals for the batch produced by factory *i* during batch *j*.
- $b_i \sim N(0, \sigma_b^2)$  is a random-effects intercept for each factory *i* that accounts for factory-specific variation in quality.

```
glme = fitglme(mfr, 'defects ~ 1 + newprocess + time_dev + temp_dev + supplier + (1|factory)
Distribution', 'Poisson', 'Link', 'log', 'FitMethod', 'Laplace', 'DummyVarCoding', 'effects')
```

```
glme =
```

```
Generalized linear mixed-effects model fit by ML
```

```
Model information:
```

```
Number of observations          100
```

```

Fixed effects coefficients      6
Random effects coefficients    20
Covariance parameters         1
Distribution                   Poisson
Link                           Log
FitMethod                      Laplace

```

Formula:

```
defects ~ 1 + newprocess + time_dev + temp_dev + supplier + (1 | factory)
```

Model fit statistics:

```

AIC      BIC      LogLikelihood  Deviance
416.35   434.58   -201.17      402.35

```

Fixed effects coefficients (95% CIs):

Name	Estimate	SE	tStat	DF	pValue
'(Intercept)'	1.4689	0.15988	9.1875	94	9.8194e-15
'newprocess'	-0.36766	0.17755	-2.0708	94	0.041122
'time_dev'	-0.094521	0.82849	-0.11409	94	0.90941
'temp_dev'	-0.28317	0.9617	-0.29444	94	0.76907
'supplier_C'	-0.071868	0.078024	-0.9211	94	0.35936
'supplier_B'	0.071072	0.07739	0.91836	94	0.36078

Lower	Upper
1.1515	1.7864
-0.72019	-0.015134
-1.7395	1.5505
-2.1926	1.6263
-0.22679	0.083051
-0.082588	0.22473

Random effects covariance parameters:

Group: factory (20 Levels)

Name1	Name2	Type	Estimate
'(Intercept)'	'(Intercept)'	'std'	0.31381

Group: Error

Name	Estimate
'sqrt(Dispersion)'	1

Perform an  $F$ -test to determine if all fixed-effects coefficients are equal to 0.

```
stats = anova(glme)
```

```
stats =
```

```
ANOVA MARGINAL TESTS: DFMETHOD = 'RESIDUAL'
```

Term	FStat	DF1	DF2	pValue
'(Intercept)'	84.41	1	94	9.8194e-15
'newprocess'	4.2881	1	94	0.041122
'time_dev'	0.013016	1	94	0.90941
'temp_dev'	0.086696	1	94	0.76907
'supplier'	0.59212	2	94	0.5552

The  $P$ -values for the intercept, newprocess, time\_dev, and temp\_dev are the same as in the coefficient table of the `glme` display. The small  $P$ -values for the intercept and newprocess indicate that these are significant predictors at the 5% significance level. The large  $P$ -values for time\_dev and temp\_dev indicate that these are not significant predictors at this level.

The  $P$ -value of 0.5552 for supplier measures the combined significance for both coefficients representing the categorical variable supplier. This includes the dummy variables supplier\_C and supplier\_B as shown in the coefficient table of the `glme` display. The large  $P$ -value indicates that supplier is not a significant predictor at the 5% significance level.

## Tips

- For each fixed-effects term, `anova` performs an  $F$ -test (marginal test) to determine if all coefficients representing the fixed-effects term are equal to 0.

When fitting a generalized linear mixed-effects (GLME) model using `fitglme` and one of the maximum likelihood fit methods ('Laplace' or 'ApproximateLaplace'):

- If you specify the 'CovarianceMethod' name-value pair argument as 'conditional', then the  $F$ -tests are conditional on the estimated covariance parameters.

- If you specify the 'CovarianceMethod' name-value pair as 'JointHessian', then the  $F$ -tests account for the uncertainty in estimation of covariance parameters.

When fitting a GLME model using `fitglme` and one of the pseudo likelihood fit methods ('MPL' or 'REML'), `anova` uses the fitted linear mixed effects model from the final pseudo likelihood iteration for inference on fixed effects.

## See Also

`GeneralizedLinearMixedModel` | `coefCI` | `coefTest` | `disp` | `fitglme` | `fixedEffects`



# addTerms

**Class:** GeneralizedLinearModel

Add terms to generalized linear model

## Syntax

```
mdl1 = addTerms(mdl, terms)
```

## Description

`mdl1 = addTerms(mdl, terms)` returns a generalized linear model the same as `mdl` but with additional terms.

## Input Arguments

**mdl**

Generalized linear model, as constructed by `fitglm` or `stepwiseglm`.

**terms**

Terms to add to the `mdl` regression model. Specify as either a:

- Text representing one or more terms to add. For details, see “Wilkinson Notation” on page 32-21.
- Row or rows in the terms matrix (see `modelspec` in `fitglm`). For example, if there are three variables A, B, and C:

```
[0 0 0] represents a constant term or intercept  
[0 1 0] represents B; equivalently, A^0 * B^1 * C^0  
[1 0 1] represents A*C  
[2 0 0] represents A^2  
[0 1 2] represents B*(C^2)
```

## Output Arguments

**mdl1**

Generalized linear model, the same as `mdl` but with additional terms given in `terms`. You can set `mdl1` equal to `mdl` to overwrite `mdl`.

## Examples

### Add a term to a generalized linear regression model

Create a model using just one predictor, then add a second.

Generate artificial data for the model, Poisson random numbers with two underlying predictors `X(1)` and `X(2)`.

```
rng default % for reproducibility
rndvars = randn(100,2);
X = [2+rndvars(:,1), rndvars(:,2)];
mu = exp(1 + X*[1;2]);
y = poissrnd(mu);
```

Create a generalized linear regression model of Poisson data. Use just the first predictor in the model.

```
mdl = fitglm(X,y,...
    'y ~ x1', 'distr', 'poisson')
```

```
mdl =
```

Generalized linear regression model:

```
log(y) ~ 1 + x1
Distribution = Poisson
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	2.7784	0.014043	197.85	0

```
x1          1.1732      0.0033653      348.6      0
```

```
100 observations, 98 error degrees of freedom
Dispersion: 1
Chi^2-statistic vs. constant model: 1.25e+05, p-value = 0
```

Add the second predictor to the model.

```
mdl1 = addTerms(mdl, 'x2')
```

```
mdl1 =
```

```
Generalized linear regression model:
log(y) ~ 1 + x1 + x2
Distribution = Poisson
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	1.0405	0.022122	47.034	0
x1	0.9968	0.003362	296.49	0
x2	1.987	0.0063433	313.24	0

```
100 observations, 97 error degrees of freedom
Dispersion: 1
Chi^2-statistic vs. constant model: 2.95e+05, p-value = 0
```

## Definitions

### Wilkinson Notation

Wilkinson notation describes the factors present in models. The notation relates to factors present in models, not to the multipliers (coefficients) of those factors.

Wilkinson Notation	Factors in Standard Notation
1	Constant (intercept) term

Wilkinson Notation	Factors in Standard Notation
$A^k$ , where $k$ is a positive integer	$A, A^2, \dots, A^k$
$A + B$	$A, B$
$A*B$	$A, B, A*B$
$A:B$	$A*B$ only
$-B$	Do not include $B$
$A*B + C$	$A, B, C, A*B$
$A + B + C + A:B$	$A, B, C, A*B$
$A*B*C - A:B:C$	$A, B, C, A*B, A*C, B*C$
$A*(B + C)$	$A, B, C, A*B, A*C$

Statistics and Machine Learning Toolbox notation always includes a constant term unless you explicitly remove the term using  $-1$ .

For details, see Wilkinson and Rogers [1].

## Alternatives

`step` adds or removes terms from a model using a greedy one-step algorithm.

## References

[1] Wilkinson, G. N., and C. E. Rogers. *Symbolic description of factorial models for analysis of variance*. *J. Royal Statistics Society* 22, pp. 392–399, 1973.

## See Also

`GeneralizedLinearModel` | `removeTerms` | `step` | `stepwiseglm`

## Topics

“Generalized Linear Models” on page 12-12

# addTerms

**Class:** LinearModel

Add terms to linear regression model

## Syntax

```
mdl1 = addTerms(mdl, terms)
```

## Description

`mdl1 = addTerms(mdl, terms)` returns a linear model `mdl1` that is the same as the input model `mdl`, but with additional terms as specified by `terms`.

## Input Arguments

**mdl** — Full, fitted linear regression model

LinearModel object

Full, fitted linear regression model, specified as a LinearModel object constructed using `fitlm` or `stepwiselm`.

**terms** — Terms to add to regression model

formula | matrix

Terms to add to the regression model `mdl`, specified as one of the following:

- Formula representing one or more terms to add. For details, see [Wilkinson Notation](#) on page 32-25.
- Row or rows in the terms matrix (see the `modelspec` argument description in the fitting function `fitlm`). For example, if there are three variables A, B, and C:

```
[0 0 0] represents a constant term or intercept  
[0 1 0] represents B; equivalently, A^0 * B^1 * C^0
```

```
[1 0 1] represents A*C
[2 0 0] represents A^2
[0 1 2] represents B*(C^2)
```

## Output Arguments

### **mdl1** — Regression model with additional terms

`LinearModel` object

Regression model with additional terms, returned as a `LinearModel` object. `mdl1` is the same as `mdl` but includes the additional terms specified in `terms`. To overwrite `mdl`, set `mdl1` equal to `mdl`.

## Examples

### Add Terms to Linear Regression Model

Create a model of the `carsmall` data without any interactions, then add an interaction term.

Load the `carsmall` data and make a model of the MPG as a function of weight and model year.

```
load carsmall
tbl = table(MPG,Weight);
tbl.Year = categorical(Model_Year);
mdl = fitlm(tbl,'MPG ~ Year + Weight^2');
```

Add an interaction term to `mdl`.

```
terms = 'Year*Weight';
mdl1 = addTerms(mdl,terms)
```

```
mdl1 =
```

```
Linear regression model:
MPG ~ 1 + Weight*Year + Weight^2
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	48.045	6.779	7.0874	3.3967e-10
Weight	-0.012624	0.0041455	-3.0454	0.0030751
Year_76	2.7768	3.0538	0.90931	0.3657
Year_82	16.416	4.9802	3.2962	0.0014196
Weight:Year_76	-0.00020693	0.00092403	-0.22394	0.82333
Weight:Year_82	-0.0032574	0.0018919	-1.7217	0.088673
Weight^2	1.0121e-06	6.12e-07	1.6538	0.10177

Number of observations: 94, Error degrees of freedom: 87

Root Mean Squared Error: 2.76

R-squared: 0.89, Adjusted R-Squared 0.882

F-statistic vs. constant model: 117, p-value = 1.88e-39

## Definitions

### Wilkinson Notation

Wilkinson notation describes the factors present in models. The notation relates to factors present in models, not to the multipliers (coefficients) of those factors.

Wilkinson Notation	Factors in Standard Notation
1	Constant (intercept) term
$A^k$ , where $k$ is a positive integer	$A, A^2, \dots, A^k$
$A + B$	$A, B$
$A*B$	$A, B, A*B$
$A:B$	$A*B$ only
$-B$	Do not include $B$
$A*B + C$	$A, B, C, A*B$
$A + B + C + A:B$	$A, B, C, A*B$
$A*B*C - A:B:C$	$A, B, C, A*B, A*C, B*C$

Wilkinson Notation	Factors in Standard Notation
$A^*(B + C)$	A, B, C, A*B, A*C

Statistics and Machine Learning Toolbox notation always includes a constant term unless you explicitly remove the term using `-1`.

For details, see Wilkinson and Rogers [1].

## Alternatives

Use `stepwiselm` to select a model from a starting model, continuing until no single step is beneficial.

Use `removeTerms` to remove particular terms.

Use `step` to optimally improve the model by adding or removing terms.

## References

[1] Wilkinson, G. N., and C. E. Rogers. *Symbolic description of factorial models for analysis of variance*. J. Royal Statistics Society 22, pp. 392–399, 1973.

## See Also

`LinearModel` | `removeTerms` | `step` | `stepwiselm`

## Topics

“Linear Regression” on page 11-11



# adtest

Anderson-Darling test

## Syntax

```
h = adtest(x)
h = adtest(x,Name,Value)
[h,p] = adtest(____)
[h,p,adstat,cv] = adtest(____)
```

## Description

`h = adtest(x)` returns a test decision for the null hypothesis that the data in vector `x` is from a population with a normal distribution, using the Anderson-Darling test on page 32-34. The alternative hypothesis is that `x` is not from a population with a normal distribution. The result `h` is 1 if the test rejects the null hypothesis at the 5% significance level, or 0 otherwise.

`h = adtest(x,Name,Value)` returns a test decision for the Anderson-Darling test with additional options specified by one or more name-value pair arguments. For example, you can specify a null distribution other than normal, or select an alternative method for calculating the  $p$ -value.

`[h,p] = adtest(____)` also returns the  $p$ -value, `p`, of the Anderson-Darling test, using any of the input arguments from the previous syntaxes.

`[h,p,adstat,cv] = adtest(____)` also returns the test statistic, `adstat`, and the critical value, `cv`, for the Anderson-Darling test.

## Examples

### Anderson-Darling Test for a Normal Distribution

Load the sample data. Create a vector containing the first column of the students' exam grades data.

```
load examgrades
x = grades(:,1);
```

Test the null hypothesis that the exam grades come from a normal distribution. You do not need to specify values for the population parameters.

```
[h,p,adstat,cv] = adtest(x)
```

```
h =
    logical
     0

p =
    0.1854

adstat =
    0.5194

cv =
    0.7470
```

The returned value of  $h = 0$  indicates that `adtest` fails to reject the null hypothesis at the default 5% significance level.

### Anderson-Darling Test for Extreme Value Distribution

Load the sample data. Create a vector containing the first column of the students' exam grades data.

```
load examgrades
x = grades(:,1);
```

Test the null hypothesis that the exam grades come from an extreme value distribution. You do not need to specify values for the population parameters.

```
[h,p] = adtest(x, 'Distribution', 'ev')
```

```
h =
    logical
     0
```

```
p =
    0.0714
```

The returned value of  $h = 0$  indicates that `adtest` fails to reject the null hypothesis at the default 5% significance level.

### Anderson-Darling Test Using Specified Probability Distribution

Load the sample data. Create a vector containing the first column of the students' exam grades data.

```
load examgrades
x = grades(:,1);
```

Create a normal probability distribution object with mean  $\mu = 75$  and standard deviation  $\sigma = 10$ .

```
dist = makedist('normal', 'mu', 75, 'sigma', 10)
```

```
dist =
    NormalDistribution
    Normal distribution
```

```
mu = 75
sigma = 10
```

Test the null hypothesis that `x` comes from the hypothesized normal distribution.

```
[h,p] = adtest(x, 'Distribution', dist)
```

```
h =
```

```
logical
```

```
0
```

```
p =
```

```
0.4687
```

The returned value of `h = 0` indicates that `adtest` fails to reject the null hypothesis at the default 5% significance level.

## Input Arguments

### **x** — Sample data

vector

Sample data, specified as a vector. Missing observations in `x`, indicated by `NaN`, are ignored.

Data Types: `single` | `double`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'Alpha', 0.01, 'MCTol', 0.01` conducts the hypothesis test at the 1% significance level, and determines the p-value,  $p$ , using a Monte Carlo simulation with a maximum Monte Carlo standard error for  $p$  of 0.01.

### **Distribution — Hypothesized distribution**

`'norm'` (default) | `'exp'` | `'ev'` | `'logn'` | `'weibull'` | probability distribution object

Hypothesized distribution of data vector  $x$ , specified as the comma-separated pair consisting of `'Distribution'` and one of the following.

<code>'norm'</code>	Normal distribution
<code>'exp'</code>	Exponential distribution
<code>'ev'</code>	Extreme value distribution
<code>'logn'</code>	Lognormal distribution
<code>'weibull'</code>	Weibull distribution

In this case, you do not need to specify population parameters. Instead, `adtest` estimates the distribution parameters from the sample data and tests  $x$  against a composite hypothesis that it comes from the selected distribution family with parameters unspecified.

Alternatively, you can specify any continuous probability distribution object for the null distribution. In this case, you must specify all the distribution parameters, and `adtest` tests  $x$  against a simple hypothesis that it comes from the given distribution with its specified parameters.

Example: `'Distribution', 'exp'`

### **Alpha — Significance level**

0.05 (default) | scalar value in the range (0,1)

Significance level of the hypothesis test, specified as the comma-separated pair consisting of `'Alpha'` and a scalar value in the range (0,1).

Example: `'Alpha', 0.01`

Data Types: `single` | `double`

### **MCTol — Maximum Monte Carlo standard error**

positive scalar value

Maximum Monte Carlo standard error on page 32-35 for the  $p$ -value,  $p$ , specified as the comma-separated pair consisting of 'MCTol' and a positive scalar value. If you use MCTol, adtest determines  $p$  using a Monte Carlo simulation, and the name-value pair argument Asymptotic must have the value false.

Example: 'MCTol', 0.01

Data Types: single | double

### **Asymptotic** — Method for calculating $p$ -value

false (default) | true

Method for calculating the  $p$ -value of the Anderson-Darling test, specified as the comma-separated pair consisting of 'Asymptotic' and either true or false. If you specify 'true', adtest estimates the  $p$ -value using the limiting distribution of the Anderson-Darling test statistic. If you specify false, adtest calculates the  $p$ -value based on an analytical formula. For sample sizes greater than 120, the limiting distribution estimate is likely to be more accurate than the small sample size approximation method.

- If you specify a distribution family with unknown parameters for the Distribution name-value pair, Asymptotic must be false.
- If you use MCTol to calculate the  $p$ -value using a Monte Carlo simulation, Asymptotic must be false.

Example: 'Asymptotic', true

Data Types: logical

## Output Arguments

### **h** — Hypothesis test result

1 | 0

Hypothesis test result, returned as a logical value.

- If  $h = 1$ , this indicates the rejection of the null hypothesis at the Alpha significance level.
- If  $h = 0$ , this indicates a failure to reject the null hypothesis at the Alpha significance level.

**p —  $p$ -value**

scalar value in the range [0,1]

$p$ -value of the Anderson-Darling test, returned as a scalar value in the range [0,1].  $p$  is the probability of observing a test statistic as extreme as, or more extreme than, the observed value under the null hypothesis.  $p$  is calculated using one of these methods:

- If the hypothesized distribution is a fully specified probability distribution object, `adtest` calculates  $p$  analytically. If 'Asymptotic' is true, `adtest` uses the asymptotic distribution of the test statistic. If you specify a value for 'MCTol', `adtest` uses a Monte Carlo simulation.
- If the hypothesized distribution is specified as a distribution family with unknown parameters, `adtest` retrieves the critical value from a table and uses inverse interpolation to determine the  $p$ -value. If you specify a value for 'MCTol', `adtest` uses a Monte Carlo simulation.

**adstat — Test statistic**

scalar value

Test statistic for the Anderson-Darling test, returned as a scalar value.

- If the hypothesized distribution is a fully specified probability distribution object, `adtest` computes `adstat` using specified parameters.
- If the hypothesized distribution is specified as a distribution family with unknown parameters, `adtest` computes `adstat` using parameters estimated from the sample data.

**cv — Critical value**

scalar value

Critical value for the Anderson-Darling test at the significance level `Alpha`, returned as a scalar value. `adtest` determines `cv` by interpolating into a table based on the specified `Alpha` significance level.

## Definitions

### Anderson-Darling Test

The Anderson-Darling test is commonly used to test whether a data sample comes from a normal distribution. However, it can be used to test for another hypothesized distribution, even if you do not fully specify the distribution parameters. Instead, the test estimates any unknown parameters from the data sample.

The test statistic belongs to the family of quadratic empirical distribution function statistics, which measure the distance between the hypothesized distribution,  $F(x)$  and the empirical cdf,  $F_n(x)$  as

$$n \int_{-\infty}^{\infty} (F_n(x) - F(x))^2 w(x) dF(x),$$

over the ordered sample values  $x_1 < x_2 < \dots < x_n$ , where  $w(x)$  is a weight function and  $n$  is the number of data points in the sample.

The weight function for the Anderson-Darling test is

$$w(x) = [F(x)(1 - F(x))]^{-1},$$

which places greater weight on the observations in the tails of the distribution, thus making the test more sensitive to outliers and better at detecting departure from normality in the tails of the distribution.

The Anderson-Darling test statistic is

$$A_n^2 = -n - \sum_{i=1}^n \frac{2i-1}{n} [\ln(F(X_i)) + \ln(1 - F(X_{n+1-i}))],$$

where  $\{X_1 < \dots < X_n\}$  are the ordered sample data points and  $n$  is the number of data points in the sample.

In a test, the decision to reject or not reject the null hypothesis is based on comparing the  $p$ -value for the hypothesis test with the specified significance level, not on comparing the test statistic with the critical value.



## Monte Carlo Standard Error

The Monte Carlo standard error is the error due to simulating the  $p$ -value.

The Monte Carlo standard error is calculated as

$$SE = \sqrt{\frac{(\hat{p})(1 - \hat{p})}{\text{mcreps}}},$$

where  $\hat{p}$  is the estimated  $p$ -value of the hypothesis test, and `mcreps` is the number of Monte Carlo replications performed.

`adtest` chooses the number of Monte Carlo replications, `mcreps`, large enough to make the Monte Carlo standard error for  $\hat{p}$  less than the value specified for `MCTol`.

## See Also

`jbtest` | `kstest`

**Introduced in R2013a**

## AIC property

**Class:** `gmdistribution`

Akaike Information Criterion

## Description

The Akaike Information Criterion:  $2 \cdot N \log L + 2 \cdot m$ , where  $N \log L$  is the negative loglikelihood and  $m$  is the number of estimated parameters.

---

**Note** This property applies only to `gmdistribution` objects constructed with `fitgmdist`.

---

# andrewsplot

Andrews plot

## Syntax

```
andrewsplot(X)
andrewsplot(X, ..., 'Standardize', standopt)
andrewsplot(X, ..., 'Quantile', alpha)
andrewsplot(X, ..., 'Group', group)
andrewsplot(X, ..., 'PropName', PropVal, ...)
h = andrewsplot(X, ...)
```

## Description

`andrewsplot(X)` creates an Andrews plot of the multivariate data in the matrix `X`. The rows of `X` correspond to observations, the columns to variables. Andrews plots represent each observation by a function  $f(t)$  of a continuous dummy variable  $t$  over the interval  $[0,1]$ .  $f(t)$  is defined for the  $i$ th observation in `X` as

$$f(t) = X(i,1) / \sqrt{2} + X(i,2)\sin(2\pi t) + X(i,3)\cos(2\pi t) + \dots$$

`andrewsplot` treats NaN values in `X` as missing values and ignores the corresponding rows.

`andrewsplot(X, ..., 'Standardize', standopt)` creates an Andrews plot where `standopt` is one of the following:

- `'on'` — scales each column of `X` to have mean 0 and standard deviation 1 before making the plot.
- `'PCA'` — creates an Andrews plot from the principal component scores of `X`, in order of decreasing eigenvalue. (See `pca`.)
- `'PCAStd'` — creates an Andrews plot using the standardized principal component scores. (See `pca`.)

`andrewsplot(X, ..., 'Quantile', alpha)` plots only the median and the  $\alpha$  and  $(1 - \alpha)$  quantiles of  $f(t)$  at each value of  $t$ . This is useful if  $X$  contains many observations.

`andrewsplot(X, ..., 'Group', group)` plots the data in different groups with different colors. Groups are defined by `group`, a numeric array containing a group index for each observation. `group` can also be a categorical array, character matrix, or cell array of character vectors containing a group name for each observation.

`andrewsplot(X, ..., 'PropName', PropVal, ...)` sets optional lineseries object properties to the specified values for all lineseries objects created by `andrewsplot`. (See Chart Line.)

`h = andrewsplot(X, ...)` returns a column vector of handles to the lineseries objects created by `andrewsplot`, one handle per row of  $X$ . If you use the 'Quantile' input parameter, `h` contains one handle for each of the three lineseries objects created. If you use both the 'Quantile' and the 'Group' input parameters, `h` contains three handles for each group.

## Examples

### Create Andrews Plot to Visualize Grouped Data

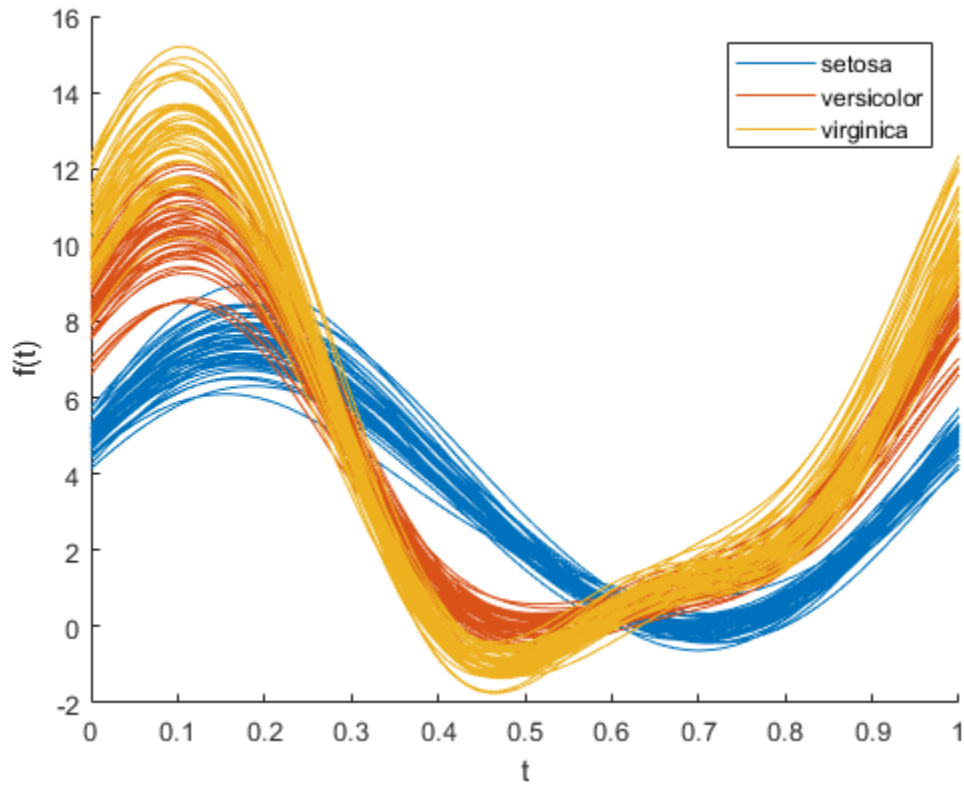
This example shows how to create an Andrews plot to visualize grouped sample data.

Load the sample data.

```
load fisheriris
```

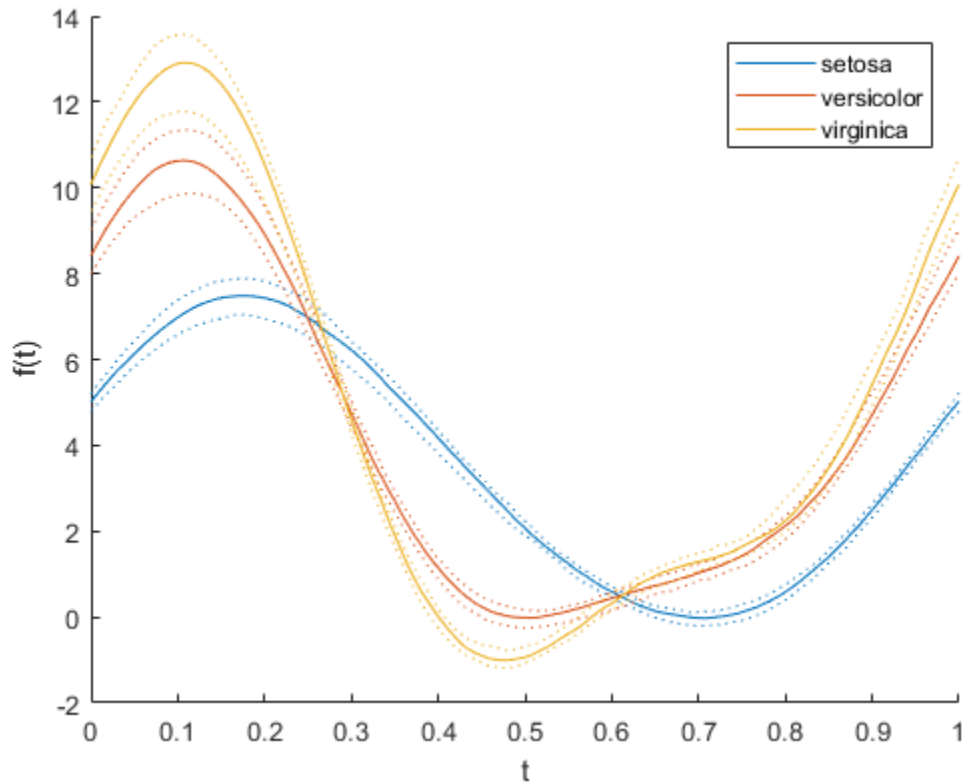
Create an Andrews plot, grouping the sample data by species.

```
andrewsplot(meas, 'group', species)
```



Create a second, simplified Andrews plot that only displays the median and quartiles of each group.

```
andrewsplot(meas, 'group', species, 'quantile', .25)
```



## See Also

`glyphplot` | `parallelcoords`

## Topics

“Grouping Variables” on page 2-59

Introduced before R2006a

# anova

**Class:** CompactLinearModel

Analysis of variance for linear model

## Syntax

```
tbl = anova mdl
tbl = anova mdl, anovatype
tbl = anova mdl, anovatype, sstype
```

## Description

`tbl = anova mdl` returns a table with summary ANOVA statistics.

`tbl = anova mdl, anovatype` returns ANOVA statistics of the chosen type.

`tbl = anova mdl, anovatype, sstype` computes ANOVA statistics using the chosen type of sum of squares.

## Input Arguments

**mdl** — Linear model object

LinearModel object | CompactLinearModel object

Linear model object, specified as a full LinearModel object constructed using `fitlm` or `stepwiselm`, or a compacted CompactLinearModel object constructed using `compact`.

**anovatype** — ANOVA type

'component' (default) | 'summary'

ANOVA type, specified as one of the following:

- 'component' — `tbl` displays a 'components' ANOVA table, with sums of squares and  $F$  tests attributable to each term in the model except the constant term.

- 'summary' — `tbl` displays a summary ANOVA table with an  $F$  test for the model as a whole.
  - If there are both linear and higher-order terms, there is also an  $F$  test for the higher-order terms as a group.
  - If `mdl` is a full `LinearModel` object and the sample data contains replications (multiple observations sharing the same predictor values), there is also an  $F$  test for lack-of-fit computed by decomposing the residual sum of squares into a sum of squares for the replicated observations and the remaining sum of squares. If `mdl` is a compacted `CompactLinearModel` object, then the ANOVA table presents, but does not decompose, the residual sum of squares.

### **sstype** — Sum of squares type

'h' (default) | 1 | 2 | 3

Sum of squares type, specified as one of the following:

Value	Description
1	Type I sum of squares. The reduction in residual sum of squares obtained by adding that term to a fit that already includes the terms listed before it.
2	Type II sum of squares. The reduction in residual sum of squares obtained by adding that term to a model consisting of all other terms that do not contain the term in question
3	Type III sum of squares. The reduction in residual sum of squares obtained by adding that term to a model containing all other terms, but with their effects constrained to obey the usual “sigma restrictions” that make models estimable.
'h'	Hierarchical model. Similar to Type II, but uses both continuous and categorical factors to determine the hierarchy of terms.

For Type III sum of squares:



- If `mdl` is a `CompactLinearModel` object and the model is non-hierarchical, `anova` errors.
- If `mdl` is a full `LinearModel` object and the model is non-hierarchical, then `anova` refits the model using effects coding whenever it needs to compute Type III sums of squares.
- If `mdl` is either a `CompactLinearModel` or a `LinearModel` object and the model is hierarchical, `anova` computes the results without refitting the model.

`sstype` only applies if `anovatype` is `'component'`.

For more information about sum of squares types, see the `sstype` argument on the `anova` function reference page.

## Output Arguments

### `tbl` — ANOVA summary statistics table

table

ANOVA summary statistics table, returned as table

The contents of `tbl` depends on the ANOVA type specified in `anovatype`.

- If `anovatype` is `'component'`, then `tbl` contains:
  - Sum of squares
  - Degrees of freedom
  - Mean squares
  - $F$  statistic
  - $p$ -value
  - Formula used for model
- If `anovatype` is `'summary'`, then `tbl` contains:
  - Total Sum of Squares
  - Model Sum of Squares
    - Linear Sum of Squares (present if model has powers or interactions)

- Nonlinear Sum of Squares (present if model has powers or interactions)
- Residual Sum of Squares
  - Lack-of-fit Sum of Squares (present if model has replicates)
  - Pure error Sum of Squares (present if model has replicates)

## Examples

### Component ANOVA Table for Linear Model

Create a component ANOVA table from a model of the `carsmall` data.

Load the `hospital` data and make a model of the blood pressure as a function of age and gender.

```
load hospital
tbl = table(hospital.Age, hospital.Sex, hospital.BloodPressure(:,2), ...
           'VariableNames', {'Age', 'Sex', 'BloodPressure'});
tbl.Sex = categorical(tbl.Sex);
mdl = fitlm(tbl, 'BloodPressure ~ Sex + Age^2');
```

Create an ANOVA table.

```
tbl = anova(mdl)
```

```
tbl =
```

```
4x5 table
```

	SumSq	DF	MeanSq	F	pValue
	-----	---	-----	-----	-----
Age	18.705	1	18.705	0.40055	0.52831
Sex	222.09	1	222.09	4.7558	0.031643
Age^2	30.934	1	30.934	0.66242	0.41772
Error	4483.1	96	46.699		

## Summary ANOVA Table for Linear Model

Create a summary ANOVA table from a model of the `carsmall` data.

Load the `carsmall` data and make a model of the MPG as a function of weight and model year.

```
load carsmall
cars = table(MPG,Weight);
cars.Year = ordinal(Model_Year);
mdl = fitlm(cars, 'MPG ~ Year + Weight^2');
```

Create a summary ANOVA table.

```
tbl = anova(mdl, 'summary')
```

```
tbl =
```

```
7x5 table
```

	SumSq	DF	MeanSq	F	pValue
Total	6005.3	93	64.573		
Model	5317	4	1329.3	171.88	5.5208e-41
. Linear	5240.3	3	1746.8	225.87	1.7302e-41
. Nonlinear	76.688	1	76.688	9.9164	0.0022303
Residual	688.27	89	7.7334		
. Lack of fit	663.77	86	7.7183	0.9451	0.62874
. Pure error	24.5	3	8.1667		

The summary ANOVA table shows tests for groups of terms. The nonlinear group consists of just the `Weight^2` term, so it has the same *P*-value as that term in the component ANOVA table. The *F* statistic comparing the residual sum of squares to a "pure error" estimate from replicated observations shows no evidence of lack of fit.

- “ANOVA” on page 11-21

## Alternatives

More complete ANOVA statistics are available in the `anova1`, `anova2`, and `anovan` functions.

## See Also

`CompactLinearModel` | `LinearModel` | `table`

## Topics

“ANOVA” on page 11-21

“Linear Regression” on page 11-11

## anova

**Class:** LinearMixedModel

Analysis of variance for linear mixed-effects model

## Syntax

```
stats = anova(lme)
stats = anova(lme, Name, Value)
```

## Description

`stats = anova(lme)` returns the dataset array `stats` that includes the results of the *F*-tests for each fixed-effects term in the linear mixed-effects model `lme`.

`stats = anova(lme, Name, Value)` also returns the dataset array `stats` with additional options specified by one or more `Name, Value` pair arguments.

## Input Arguments

**lme** — Linear mixed-effects model

LinearMixedModel object

Linear mixed-effects model, specified as a LinearMixedModel object constructed using `fitlme` or `fitlmematrix`.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

**DFMethod** — Method for computing approximate degrees of freedom`'residual' (default) | 'satterthwaite' | 'none'`

Method for computing approximate degrees of freedom to use in the  $F$ -test, specified as the comma-separated pair consisting of 'DFMethod' and one of the following.

<code>'residual'</code>	Default. The degrees of freedom are assumed to be constant and equal to $n - p$ , where $n$ is the number of observations and $p$ is the number of fixed effects.
<code>'satterthwaite'</code>	Satterthwaite approximation.
<code>'none'</code>	All degrees of freedom are set to infinity.

For example, you can specify the Satterthwaite approximation as follows.

Example: `'DFMethod', 'satterthwaite'`

## Output Arguments

**stats** — Results of  $F$ -tests for fixed-effects terms`dataset array`

Results of  $F$ -tests for fixed-effects terms, returned as a dataset array with the following columns.

Term	Name of the fixed effects term
Fstat	$F$ -statistic for the term
DF1	Numerator degrees of freedom for the $F$ -statistic
DF2	Denominator degrees of freedom for the $F$ -statistic
pValue	$p$ -value of the test for the term

There is one row for each fixed-effects term. Each term is a continuous variable, a grouping variable, or an interaction between two or more continuous or grouping variables. For each fixed-effects term, `anova` performs an  $F$ -test (marginal test) to determine if all coefficients representing the fixed-effects term are 0. To perform tests for the type III hypothesis, you must use the `'effects'` contrasts while fitting the linear mixed-effects model.

## Examples

### F-Tests for Fixed Effects

Load the sample data.

```
load(fullfile(matlabroot, 'examples', 'stats', 'shift.mat'))
```

The data shows the deviations from the target quality characteristic measured from the products that five operators manufacture during three shifts: morning, evening, and night. This is a randomized block design, where the operators are the blocks. The experiment is designed to study the impact of the time of shift on the performance. The performance measure is the deviation of the quality characteristics from the target value. This is simulated data.

Shift and Operator are nominal variables.

```
shift.Shift = nominal(shift.Shift);
shift.Operator = nominal(shift.Operator);
```

Fit a linear mixed-effects model with a random intercept grouped by operator to assess if performance significantly differs according to the time of the shift. Use the restricted maximum likelihood method and 'effects' contrasts.

'effects' contrasts indicate that the coefficients sum to 0, and fitlme creates two contrast-coded variables in the fixed-effects design matrix,  $X_1$  and  $X_2$ , where

$$Shift\_Evening = \begin{cases} 0, & \text{if Morning} \\ 1, & \text{if Evening} \\ -1, & \text{if Night} \end{cases}$$

and

$$Shift\_Morning = \begin{cases} 1, & \text{if Morning} \\ 0, & \text{if Evening} \\ -1, & \text{if Night} \end{cases}.$$

The model corresponds to

$$\text{MorningShift : } QCDev_{im} = \beta_0 + \beta_2 Shift\_Morning_i + b_{0m} + \varepsilon_{im}, \quad m = 1, 2, \dots, 5,$$

$$\text{EveningShift : } QCDev_{im} = \beta_0 + \beta_1 Shift\_Evening_i + b_{0m} + \varepsilon_{im},$$

$$\text{NightShift : } QCDev_{im} = \beta_0 - \beta_1 Shift\_Evening_i - \beta_2 Shift\_Morning_i + b_{0m} + \varepsilon_{im},$$

where  $b \sim N(0, \sigma_b^2)$  and  $\epsilon \sim N(0, \sigma^2)$ .

```
lme = fitlme(shift, 'QCDev ~ Shift + (1|Operator)', ...
'FitMethod', 'REML', 'DummyVarCoding', 'effects')
```

```
lme =
```

```
Linear mixed-effects model fit by REML
```

```
Model information:
```

Number of observations	15
Fixed effects coefficients	3
Random effects coefficients	5
Covariance parameters	2

```
Formula:
```

```
QCDev ~ 1 + Shift + (1 | Operator)
```

```
Model fit statistics:
```

AIC	BIC	LogLikelihood	Deviance
58.913	61.337	-24.456	48.913

```
Fixed effects coefficients (95% CIs):
```

Name	Estimate	SE	tStat	DF	pValue
'(Intercept)'	3.6525	0.94109	3.8812	12	0.0021832
'Shift_Evening'	-0.53293	0.31206	-1.7078	12	0.11339
'Shift_Morning'	-0.91973	0.31206	-2.9473	12	0.012206

Lower	Upper
1.6021	5.703
-1.2129	0.14699
-1.5997	-0.23981

```
Random effects covariance parameters (95% CIs):
```

```
Group: Operator (5 Levels)
```

Name1	Name2	Type	Estimate	Lower
'(Intercept)'	'(Intercept)'	'std'	2.0457	0.98207

Upper
4.2612



```

Group: Error
      Name      Estimate      Lower      Upper
      'Res Std'      0.85462      0.52357      1.395

```

Perform an  $F$ -test to determine if all fixed-effects coefficients are 0.

```
anova(lme)
```

```
ans =
```

```
ANOVA MARGINAL TESTS: DFMETHOD = 'RESIDUAL'
```

Term	FStat	DF1	DF2	pValue
'(Intercept)'	15.063	1	12	0.0021832
'Shift'	11.091	2	12	0.0018721

The  $P$ -value for the constant term, 0.0021832, is the same as in the coefficient table in the `lme` display. The  $P$ -value of 0.0018721 for `Shift` measures the combined significance for both coefficients representing `Shift`.

## ANOVA for Fixed-Effects in LME Model

Load the sample data.

```
load(fullfile(matlabroot, 'examples', 'stats', 'fertilizer.mat'))
```

The dataset array includes data from a split-plot experiment, where soil is divided into three blocks based on the soil type: sandy, silty, and loamy. Each block is divided into five plots, where five types of tomato plants (cherry, heirloom, grape, vine, and plum) are randomly assigned to these plots. The tomato plants in the plots are then divided into subplots, where each subplot is treated by one of four fertilizers. This is simulated data.

Store the data in a dataset array called `ds`, for practical purposes, and define `Tomato`, `Soil`, and `Fertilizer` as categorical variables.

```
ds = fertilizer;
ds.Tomato = nominal(ds.Tomato);
```

```
ds.Soil = nominal(ds.Soil);
ds.Fertilizer = nominal(ds.Fertilizer);
```

Fit a linear mixed-effects model, where `Fertilizer` and `Tomato` are the fixed-effects variables, and the mean yield varies by the block (soil type) and the plots within blocks (tomato types within soil types) independently. Use the `'effects'` contrasts when fitting the data for the type III sum of squares.

```
lme = fitlme(ds, 'Yield ~ Fertilizer * Tomato + (1|Soil) + (1|Soil:Tomato)', ...
'DummyVarCoding', 'effects')
```

```
lme =
```

```
Linear mixed-effects model fit by ML
```

```
Model information:
```

Number of observations	60
Fixed effects coefficients	20
Random effects coefficients	18
Covariance parameters	3

```
Formula:
```

```
Yield ~ 1 + Tomato*Fertilizer + (1 | Soil) + (1 | Soil:Tomato)
```

```
Model fit statistics:
```

AIC	BIC	LogLikelihood	Deviance
522.57	570.74	-238.29	476.57

```
Fixed effects coefficients (95% CIs):
```

Name	Estimate	SE	tStat	DF
'(Intercept)'	104.6	3.3008	31.69	40
'Tomato_Cherry'	1.4	5.9353	0.23588	40
'Tomato_Grape'	-7.7667	5.9353	-1.3085	40
'Tomato_Heirloom'	-11.183	5.9353	-1.8842	40
'Tomato_Plum'	30.233	5.9353	5.0938	40
'Fertilizer_1'	-28.267	2.3475	-12.041	40
'Fertilizer_2'	-1.9333	2.3475	-0.82356	40
'Fertilizer_3'	10.733	2.3475	4.5722	40
'Tomato_Cherry:Fertilizer_1'	-0.73333	4.6951	-0.15619	40
'Tomato_Grape:Fertilizer_1'	-7.5667	4.6951	-1.6116	40
'Tomato_Heirloom:Fertilizer_1'	5.1833	4.6951	1.104	40
'Tomato_Plum:Fertilizer_1'	2.7667	4.6951	0.58927	40

'Tomato_Cherry:Fertilizer_2'	7.6	4.6951	1.6187	40
'Tomato_Grape:Fertilizer_2'	-1.9	4.6951	-0.40468	40
'Tomato_Heirloom:Fertilizer_2'	5.5167	4.6951	1.175	40
'Tomato_Plum:Fertilizer_2'	-3.9	4.6951	-0.83066	40
'Tomato_Cherry:Fertilizer_3'	-6.0667	4.6951	-1.2921	40
'Tomato_Grape:Fertilizer_3'	3.7667	4.6951	0.80226	40
'Tomato_Heirloom:Fertilizer_3'	3.1833	4.6951	0.67802	40
'Tomato_Plum:Fertilizer_3'	1.1	4.6951	0.23429	40

pValue	Lower	Upper
5.9086e-30	97.929	111.27
0.81473	-10.596	13.396
0.19816	-19.762	4.2291
0.066821	-23.179	0.81242
8.777e-06	18.238	42.229
7.0265e-15	-33.011	-23.522
0.41507	-6.6779	2.8112
4.577e-05	5.9888	15.478
0.87667	-10.222	8.7558
0.11491	-17.056	1.9224
0.27619	-4.3058	14.672
0.55899	-6.7224	12.256
0.11337	-1.8891	17.089
0.68787	-11.389	7.5891
0.24695	-3.9724	15.006
0.4111	-13.389	5.5891
0.20373	-15.556	3.4224
0.42714	-5.7224	13.256
0.50167	-6.3058	12.672
0.81596	-8.3891	10.589

Random effects covariance parameters (95% CIs):

Group: Soil (3 Levels)

Name1	Name2	Type	Estimate	Lower
'(Intercept)'	'(Intercept)'	'std'	2.5028	0.027711

Upper  
226.05

Group: Soil:Tomato (15 Levels)

Name1	Name2	Type	Estimate	Lower
'(Intercept)'	'(Intercept)'	'std'	10.225	6.1497

```
Upper
17.001
```

```
Group: Error
```

Name	Estimate	Lower	Upper
'Res Std'	10.499	8.5389	12.908

Perform an analysis of variance to test for the fixed-effects.

```
anova(lme)
```

```
ans =
```

```
ANOVA MARGINAL TESTS: DFMETHOD = 'RESIDUAL'
```

Term	FStat	DF1	DF2	pValue
'(Intercept)'	1004.2	1	40	5.9086e-30
'Tomato'	7.1663	4	40	0.00018935
'Fertilizer'	58.833	3	40	1.0024e-14
'Tomato:Fertilizer'	1.4182	12	40	0.19804

The  $P$ -value for the constant term, 5.9086e-30, is the same as in the coefficient table in the `lme` display. The  $P$ -values of 0.00018935, 1.0024e-14, and 0.19804 for `Tomato`, `Fertilizer`, and `Tomato:Fertilizer` represent the combined significance for all tomato coefficients, fertilizer coefficients, and coefficients representing the interaction between the tomato and fertilizer, respectively. The  $P$ -value of 0.19804 indicates that the interaction between tomato and fertilizer is not significant.

### Satterthwaite Approximation for Degrees of Freedom

Load the sample data.

```
load(fullfile(matlabroot, 'examples', 'stats', 'weight.mat'))
```

`weight` contains data from a longitudinal study, where 20 subjects are randomly assigned 4 exercise programs, and their weight loss is recorded over six 2-week time periods. This is simulated data.

Store the data in a table. Define Subject and Program as categorical variables.

```
tbl = table(InitialWeight, Program, Subject, Week, y);
tbl.Subject = nominal(tbl.Subject);
tbl.Program = nominal(tbl.Program);
```

Fit the model using the 'effects' contrasts.

```
lme = fitlme(tbl, 'y ~ InitialWeight + Program*Week + (Week|Subject)', ...
             'DummyVarCoding', 'effects')
```

```
lme =
```

Linear mixed-effects model fit by ML

Model information:

Number of observations	120
Fixed effects coefficients	9
Random effects coefficients	40
Covariance parameters	4

Formula:

```
y ~ 1 + InitialWeight + Program*Week + (1 + Week | Subject)
```

Model fit statistics:

AIC	BIC	LogLikelihood	Deviance
-22.981	13.257	24.49	-48.981

Fixed effects coefficients (95% CIs):

Name	Estimate	SE	tStat	DF
'(Intercept)'	0.77122	0.24309	3.1725	111
'InitialWeight'	0.0031879	0.0013814	2.3078	111
'Program_A'	-0.11017	0.080377	-1.3707	111
'Program_B'	0.25061	0.08045	3.1151	111
'Program_C'	-0.14344	0.080475	-1.7824	111
'Week'	0.19881	0.033727	5.8946	111
'Program_A:Week'	-0.025607	0.058417	-0.43835	111
'Program_B:Week'	0.013164	0.058417	0.22535	111
'Program_C:Week'	0.0049357	0.058417	0.084492	111

pValue	Lower	Upper
0.0019549	0.28951	1.2529

```

0.022863    0.00045067    0.0059252
0.17323     -0.26945             0.0491
0.0023402   0.091195             0.41003
0.077424    -0.3029              0.016031
4.1099e-08   0.13198              0.26564
0.66198     -0.14136             0.090149
0.82212     -0.10259             0.12892
0.93282     -0.11082             0.12069

```

Random effects covariance parameters (95% CIs):  
Group: Subject (20 Levels)

Name1	Name2	Type	Estimate	Lower
'(Intercept)'	'(Intercept)'	'std'	0.18407	0.12281
'Week'	'(Intercept)'	'corr'	0.66841	0.21076
'Week'	'Week'	'std'	0.15033	0.11004

```

Upper
0.27587
0.88573
0.20537

```

Group: Error

Name	Estimate	Lower	Upper
'Res Std'	0.10261	0.087882	0.11981

The *P*-values 0.022863 and 4.1099e-08 indicate significant effects of the initial weights of the subjects and the time factor in the amount of weight lost. The weight loss of subjects who are in program B is significantly different relative to the weight loss of subjects that are in program A. The lower and upper limits of the covariance parameters for the random effects do not include zero, thus they are significant.

Perform an F-test that all fixed-effects coefficients are zero.

```
anova(lme)
```

```
ans =
```

```
ANOVA MARGINAL TESTS: DFMETHOD = 'RESIDUAL'
```

Term	FStat	DF1	DF2	pValue
------	-------	-----	-----	--------

'(Intercept)'	10.065	1	111	0.0019549
'InitialWeight'	5.326	1	111	0.022863
'Program'	3.6798	3	111	0.014286
'Week'	34.747	1	111	4.1099e-08
'Program:Week'	0.066648	3	111	0.97748

The  $P$ -values for the constant term, initial weight, and week are the same as in the coefficient table in the previous `lme` output display. The  $P$ -value of 0.014286 for `Program` represents the combined significance for all program coefficients. Similarly, the  $P$ -value for the interaction between program and week (`Program:Week`) measures the combined significance for all coefficients representing this interaction.

Now, use the Satterthwaite method to compute the degrees of freedom.

```
anova(lme, 'DFMethod', 'satterthwaite')
```

```
ans =
```

```
ANOVA MARGINAL TESTS: DFMETHOD = 'SATERTHWAITE'
```

Term	FStat	DF1	DF2	pValue
'(Intercept)'	10.065	1	20.445	0.004695
'InitialWeight'	5.326	1	20	0.031827
'Program'	3.6798	3	19.14	0.030233
'Week'	34.747	1	20	9.1346e-06
'Program:Week'	0.066648	3	20	0.97697

The Satterthwaite method produces smaller denominator degrees of freedom and slightly larger  $P$ -values.

## Tips

- For each fixed-effects term, `anova` performs an  $F$ -test (marginal test), that all coefficients representing the fixed-effects term are 0. To perform tests for type III hypotheses, you must set the `'DummyVarCoding'` name-value pair argument to `'effects'` contrasts while fitting your linear mixed-effects model.

## See Also

`LinearMixedModel` | `fitlme` | `fitlmematrix`



# anova1

One-way analysis of variance

## Syntax

```
p = anova1(y)
p = anova1(y,group)
p = anova1(y,group,displayopt)
[p,tbl] = anova1(____)
[p,tbl,stats] = anova1(____)
```

## Description

`p = anova1(y)` returns the  $p$ -value for a balanced one-way ANOVA on page 9-3. It also displays the standard ANOVA table (`tbl`) and a box plot of the columns of `y`. `anova1` tests the hypothesis that the samples in `y` are drawn from populations with the same mean against the alternative hypothesis that the population means are not all the same.

`p = anova1(y,group)` returns the  $p$ -value for a balanced one-way ANOVA by group. It also displays the standard ANOVA table and a box-plot on page 32-74 of the observations of `y` by group.

`p = anova1(y,group,displayopt)` enables the ANOVA table and box plot displays when `displayopt` is 'on' (default) and suppresses the displays when `displayopt` is 'off'.

`[p,tbl] = anova1(____)` returns the ANOVA table (including column and row labels) in the cell array `tbl`. To copy a text version of the ANOVA table to the clipboard, select **Edit > Copy Text**.

`[p,tbl,stats] = anova1(____)` returns a structure, `stats`, which you can use to perform a multiple comparison test on page 9-26. A multiple comparison test enables you to determine which pairs of group means are significantly different. To perform this test, use `multcompare`, providing the `stats` structure as an input argument.

## Examples

### One-Way ANOVA

Create sample data matrix `y` with columns that are constants, plus random normal disturbances with mean 0 and standard deviation 1.

```
y = meshgrid(1:5);  
rng default; % For reproducibility  
y = y + normrnd(0,1,5,5)
```

`y =`

```
    1.5377    0.6923    1.6501    3.7950    5.6715  
    2.8339    1.5664    6.0349    3.8759    3.7925  
   -1.2588    2.3426    3.7254    5.4897    5.7172  
    1.8622    5.5784    2.9369    5.4090    6.6302  
    1.3188    4.7694    3.7147    5.4172    5.4889
```

Perform one-way ANOVA.

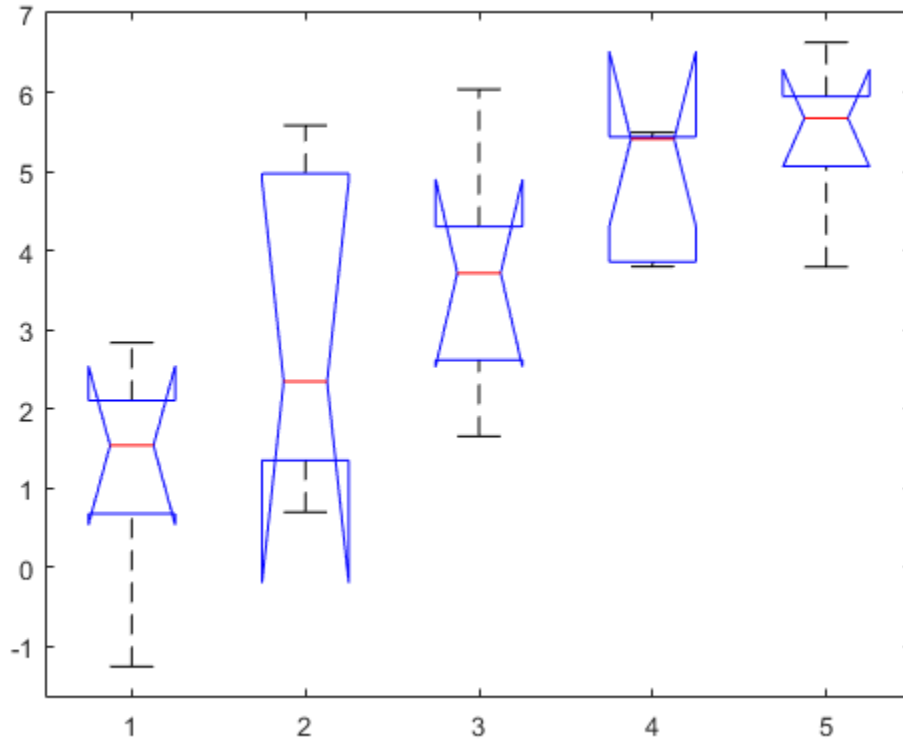
```
p = anova1(y)
```

`p =`

```
    0.0023
```

**ANOVA Table**

Source	SS	df	MS	F	Prob>F
Columns	53.7238	4	13.4309	6.05	0.0023
Error	44.408	20	2.2204		
Total	98.1318	24			



The ANOVA table shows the between-groups variation (Columns) and within-groups variation (Error). *SS* is the sum of squares, and *df* is the degrees of freedom. The total degrees of freedom is total number of observations minus one, which is  $25 - 1 = 24$ . The between-groups degrees of freedom is number of groups minus one, which is  $5 - 1 = 4$ . The within-groups degrees of freedom is total degrees of freedom minus the between groups degrees of freedom, which is  $24 - 4 = 20$ .

*MS* is the mean squared error, which is  $SS/df$  for each source of variation. The *F*-statistic is the ratio of the mean squared errors (13.4309/2.2204). The *p*-value is the probability that the test statistic can take a value greater than the value of the computed test

statistic, i.e.,  $P(F > 6.05)$ . The small  $p$ -value of 0.0023 indicates that differences between column means are significant.

## Compare Beam Strength Using One-Way ANOVA

Input the sample data.

```
strength = [82 86 79 83 84 85 86 87 74 82 ...
            78 75 76 77 79 79 77 78 82 79];
alloy = {'st', 'st', 'st', 'st', 'st', 'st', 'st', 'st', ...
        'al1', 'al1', 'al1', 'al1', 'al1', 'al1', ...
        'al2', 'al2', 'al2', 'al2', 'al2', 'al2'};
```

The data are from a study of the strength of structural beams in Hogg (1987). The vector `strength` measures deflections of beams in thousandths of an inch under 3000 pounds of force. The vector `alloy` identifies each beam as steel ('st'), alloy 1 ('al1'), or alloy 2 ('al2'). Although `alloy` is sorted in this example, grouping variables do not need to be sorted.

Test the null hypothesis that the steel beams are equal in strength to the beams made of the two more expensive alloys. Turn the figure display off and return the ANOVA results in a cell array.

```
[p,tbl] = anova1(strength,alloy,'off')
```

```
p =
```

```
1.5264e-04
```

```
tbl =
```

```
4x6 cell array
```

```
Columns 1 through 5
```

{'Source'}	{'SS'}	{'df'}	{'MS'}	{'F'}
{'Groups'}	{[184.8000]}	{[ 2]}	{[ 92.4000]}	{[ 15.4000]}
{'Error' }	{[102.0000]}	{[17]}	{[ 6.0000]}	{0x0 double}
{'Total' }	{[286.8000]}	{[19]}	{0x0 double}	{0x0 double}

```
Column 6
```

```
{'Prob>F'      }  
{[1.5264e-04]}  
{0x0 double  }  
{0x0 double  }
```

The total degrees of freedom is total number of observations minus one, which is  $20 - 1 = 19$ . The between-groups degrees of freedom is number of groups minus one, which is  $3 - 1 = 2$ . The within-groups degrees of freedom is total degrees of freedom minus the between groups degrees of freedom, which is  $19 - 2 = 17$ .

MS is the mean squared error, which is  $SS/df$  for each source of variation. The  $F$ -statistic is the ratio of the mean squared errors. The  $p$ -value is the probability that the test statistic can take a value greater than or equal to the value of the test statistic. The  $p$ -value of  $1.5264e-04$  suggests rejection of the null hypothesis.

You can retrieve the values in the ANOVA table by indexing into the cell array. Save the  $F$ -statistic value and the  $p$ -value in the new variables `Fstat` and `pvalue`.

```
Fstat = tbl{2,5}  
pvalue = tbl{2,6}
```

```
Fstat =  
  
    15.4000  
  
pvalue =  
  
    1.5264e-04
```

## Multiple Comparisons for One-Way ANOVA

Input the sample data.

```
strength = [82 86 79 83 84 85 86 87 74 82 ...  
           78 75 76 77 79 79 77 78 82 79];  
alloy = {'st', 'st', 'st', 'st', 'st', 'st', 'st', 'st', ...
```

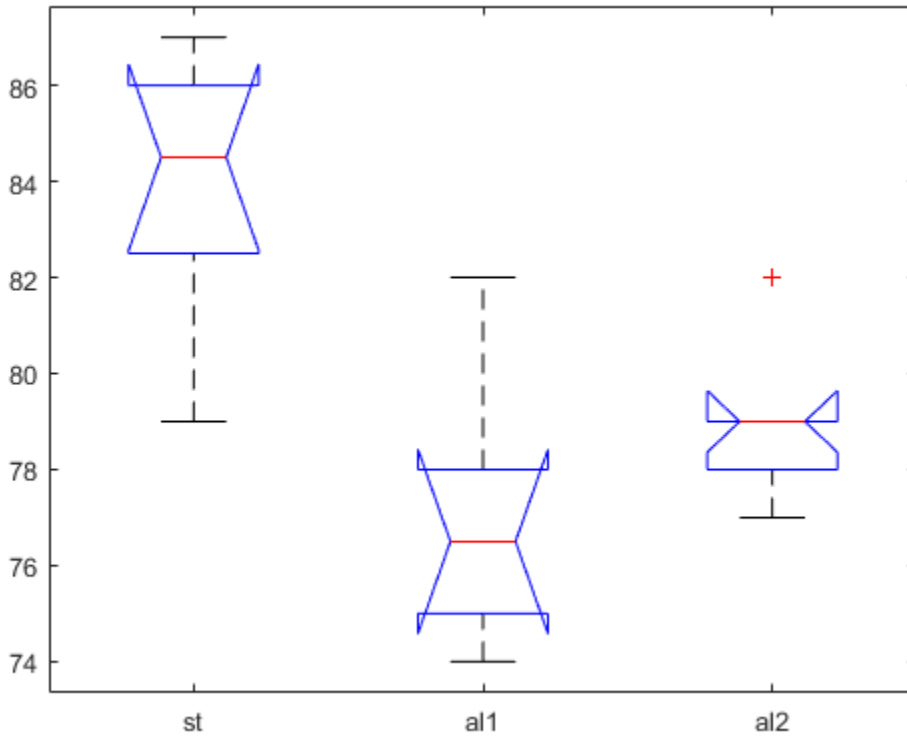
```
'a11','a11','a11','a11','a11','a11',...
'a12','a12','a12','a12','a12','a12'};
```

The data are from a study of the strength of structural beams in Hogg (1987). The vector `strength` measures deflections of beams in thousandths of an inch under 3000 pounds of force. The vector `alloy` identifies each beam as steel (`st`), alloy 1 (`a11`), or alloy 2 (`a12`). Although alloy is sorted in this example, grouping variables do not need to be sorted.

Perform one-way ANOVA using `anova1`. Return the structure `stats`, which contains the statistics `multcompare` needs for performing “Multiple Comparisons” on page 9-26.

```
[~,~,stats] = anova1(strength,alloy);
```

ANOVA Table					
Source	SS	df	MS	F	Prob>F
Groups	184.8	2	92.4	15.4	0.0002
Error	102	17	6		
Total	286.8	19			

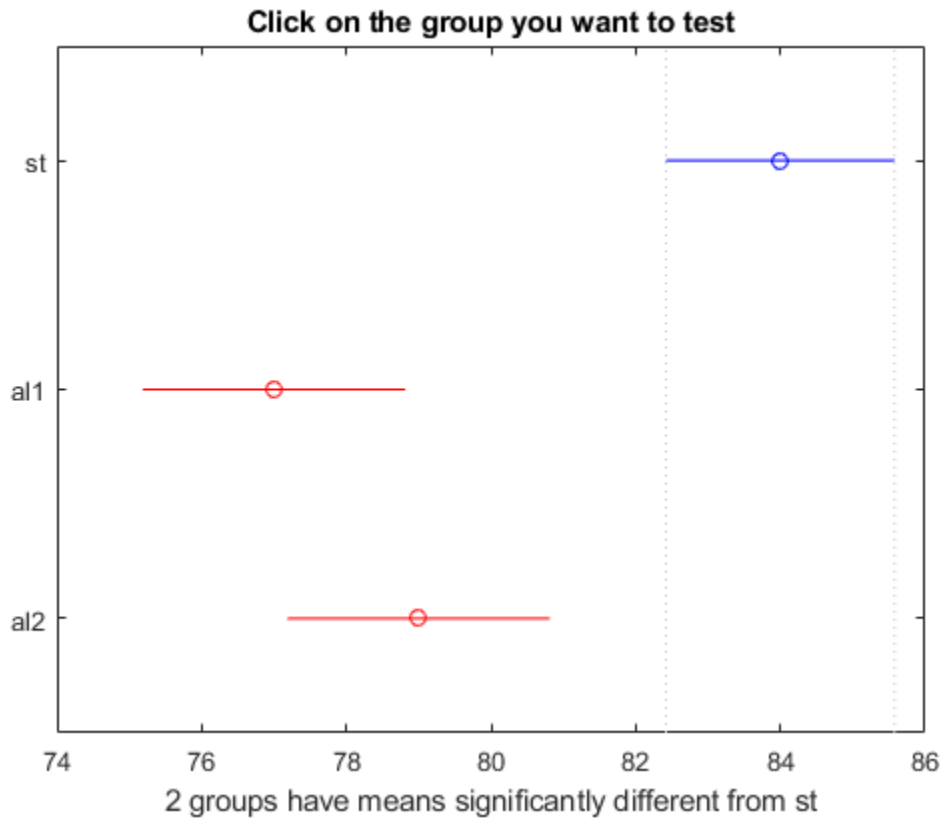


The small  $p$ -value of 0.0002 suggests that the strength of the beams is not the same.

Perform a multiple comparison of the mean strength of the beams.

```
[c,~,~,gnames] = multcompare(stats);
```





Display the comparison results with the corresponding group names.

```
[gnames(c(:,1)), gnames(c(:,2)), num2cell(c(:,3:6))]
```

```
ans =
```

```
3x6 cell array
```

```
Columns 1 through 5
```

```

{'st' }      {'al1'}      {[ 3.6064]}      {[ 7]}      {[10.3936]}
{'st' }      {'al2'}      {[ 1.6064]}      {[ 5]}      {[ 8.3936]}
{'al1'}      {'al2'}      {[ -5.6280]}     {[ -2]}     {[ 1.6280]}
```

```
Column 6
```

```
{ [1.6831e-04] }  
{ [ 0.0040] }  
{ [ 0.3560] }
```

The first two columns show the pair of groups that are compared. The fourth column shows the difference between the estimated group means. The third and fifth columns show the lower and upper limits for the 95% confidence intervals of the true difference of means. The sixth column shows the  $p$ -value for a hypothesis that the true difference of means for the corresponding groups is equal to zero.

The first two rows show that both comparisons involving the first group (steel) have confidence intervals that do not include zero. Because the corresponding  $p$ -values (1.6831e-04 and 0.0040, respectively) are small, those differences are significant.

The third row shows that the differences in strength between the two alloys is not significant. A 95% confidence interval for the difference is [-5.6,1.6], so you cannot reject the hypothesis that the true difference is zero. The corresponding  $p$ -value of 0.3560 in the sixth column confirms this result.

In the figure, the blue bar represents the comparison interval for mean material strength for steel. The red bars represent the comparison intervals for the mean material strength for alloy 1 and alloy 2. Neither of the red bars overlap with the blue bar, which indicates that the mean material strength for steel is significantly different from that of alloy 1 and alloy 2. To confirm the significant difference by clicking the bars that represent alloy 1 and 2.

- “Perform One-Way ANOVA” on page 9-6

## Input Arguments


### **y** — sample data

vector | matrix

Sample data, specified as a vector or a matrix.

- If  $y$  is a vector, you must specify the `group` input argument. `group` must be a categorical variable, numeric vector, logical vector, character vector, or cell array of

character vectors, with one name for each element of  $y$ . The `anova1` function treats the  $y$  values corresponding to the same value of `group` as part of the same group. Use this design when groups have different numbers of elements (unbalanced ANOVA).

$$Y = [y_1 \ y_2 \ y_3 \ y_4 \ y_5 \ \dots \ y_N]$$



$$g = \{ 'A', 'A', 'C', 'B', 'B', \dots, 'D' \}$$

- If  $y$  is a matrix and you do not specify `group`, `anova1` treats each column of  $y$  as a separate group. In this design, the function evaluates whether the population means of the columns are equal. Use this design when each group has the same number of elements (balanced ANOVA).

$$Y = \begin{matrix} & \begin{matrix} \text{group 1} \\ \downarrow \\ Y_{11} & Y_{12} & \dots & Y_{1k} \\ Y_{21} & Y_{22} & \dots & Y_{2k} \\ \vdots & \vdots & & \\ \vdots & \vdots & & \\ Y_{n1} & Y_{n2} & \dots & Y_{nk} \end{matrix} & \begin{matrix} \text{group 2} \\ \downarrow \\ \\ \\ \\ \\ \end{matrix} & \begin{matrix} \text{group k} \\ \downarrow \\ \\ \\ \\ \\ \end{matrix} \end{matrix}$$

- If  $y$  is a matrix and you specify `group`, then `group` must be a character array or cell array of character vectors, with one name for each column of  $y$ . The `anova1` function treats the columns that have the same group name as part of the same group.

$$Y = \begin{bmatrix} Y_{11} & Y_{12} & Y_{13} & Y_{14} & \cdots & Y_{1k} \\ Y_{21} & Y_{22} & Y_{23} & Y_{24} & \cdots & Y_{2k} \\ \vdots & \vdots & \vdots & \vdots & & \\ \vdots & \vdots & \vdots & \vdots & & \\ Y_{n1} & Y_{n2} & Y_{n3} & Y_{n4} & \cdots & Y_{nk} \end{bmatrix}$$


  
`group = ['Red', 'Black', 'Red', 'Yellow', . . . , 'Black']`

If `group` contains empty or NaN valued cells or character vectors, `anova1` disregards the corresponding observations in `y`.

Data Types: `single` | `double`


### **group** — Grouping variable

numeric vector | logical vector | character array | cell array of character vectors

Grouping variable, specified as a numeric or logical vector, character array, or a cell array of character vectors, containing group names.

- If `y` is a vector, `group` must be a categorical variable, numeric vector, logical vector, character array, or cell array of character vectors, with one name for each element of `y`. The `anova1` function treats the `y` values corresponding to the same value of `group` as part of the same group.


$$Y = [y_1 \quad y_2 \quad y_3 \quad y_4 \quad y_5 \quad \cdots \quad y_N]$$


  
`g = {'A', 'A', 'C', 'B', 'B', . . . , 'D'}`

$N$  is the total number of observations.

- If `y` is a matrix, then `group` must be a character array or cell array of character vectors, with one group name for each column of `y`. The `anova1` function treats the columns of `y` that have the same group name as part of the same group.

$$Y = \begin{bmatrix} Y_{11} & Y_{12} & Y_{13} & Y_{14} & \cdots & Y_{1k} \\ Y_{21} & Y_{22} & Y_{23} & Y_{24} & \cdots & Y_{2k} \\ \vdots & \vdots & \vdots & \vdots & & \\ \vdots & \vdots & \vdots & \vdots & & \\ Y_{n1} & Y_{n2} & Y_{n3} & Y_{n4} & \cdots & Y_{nk} \end{bmatrix}$$


  
`group = ['Red', 'Black', 'Red', 'Yellow', . . . , 'Black']`

If you do not want to specify group names, enter an empty array (`[]`) or omit this argument.

If `group` contains empty or NaN valued cells or character vectors, the corresponding observations in `y` are disregarded.

For more information on grouping variables, see “Grouping Variables” on page 2-59.

For example, if `y` is a vector, with observations categorized into groups 1, 2, and 3, then you can specify the grouping variables as follows.

Example: `'group', [1,2,1,3,1,...,3,1]`

For example, if `y` is a matrix, with six columns categorized into groups red, white, and black, then you can specify the grouping variables as follows.

Example: `'group', {'white', 'red', 'white', 'black', 'red'}`

Data Types: `single` | `double` | `logical` | `char` | `cell`

**displayopt** — Indicator to display ANOVA table and box plot

`'on'` (default) | `'off'`

Indicator to display ANOVA table and box plot, specified as `'on'` or `'off'`. When `displayopt` is `'off'`, `anova1` returns the output arguments, only. It does not display the standard ANOVA table and box-plot of the columns of `y`.

Example: `p = anova(x,group,'off')`

## Output Arguments

### **p** — *p*-value for the *F*-test

scalar value

*p*-value for the *F*-test, returned as a scalar value. *p*-value is the probability that the *F*-statistic can take a value larger than the computed test-statistic value. `anova1` tests the null hypothesis that all group means are equal to each other against the alternative hypothesis that at least one group mean is different from the others. The function derives the *p*-value from the cdf of the *F*-distribution.

A *p*-value that is smaller than the significance level indicates that at least one of the sample means is significantly different from the others. Common significance levels are 0.05 or 0.01.

### **tbl** — ANOVA table

cell array

ANOVA table, returned as a cell array. `tbl` has six columns.

Column	Definition
source	The source of the variability.
SS	The sum of squares due to each source.
df	The degrees of freedom associated with each source. Suppose $N$ is the total number of observations and $k$ is the number of groups. Then, $N - k$ is the within-groups degrees of freedom ( <code>ERROR</code> ), $k - 1$ is the between-groups degrees of freedom ( <code>COLUMNS</code> ), and $N - 1$ is the total degrees of freedom. $N - 1 = (N - k) + (k - 1)$
MS	The mean squares for each source, which is the ratio $SS/df$ .
F	<i>F</i> -statistic, which is the ratio of the mean squares.

Column	Definition
Prob>F	The $p$ -value, which is the probability that the $F$ -statistic can take a value larger than the computed test-statistic value. <code>anova1</code> derives this probability from the cdf of $F$ -distribution.

The rows of the ANOVA table show the variability in the data that is divided by the source.

Row	Definition
Groups	Variability due to the differences among the group means (variability <i>between</i> groups)
Error	Variability due to the differences between the data in each group and the group mean (variability <i>within</i> groups)
Total	Total variability

### **stats** — Statistics for multiple comparison tests

structure

Statistics for multiple comparison tests on page 9-26, returned as a structure. `stats` has six fields.

Field name	Definition
<code>gnames</code>	Names of the groups
<code>n</code>	Number of observations in each group
<code>source</code>	Source of the <code>stats</code> output
<code>means</code>	Estimated values of the means
<code>df</code>	Error (within-groups) degrees of freedom ( $N - k$ , where $N$ is the total number of observations and $k$ is the number of groups)
<code>s</code>	Square root of the mean squared error

## Definitions

### Box-Plot

`anova1` returns box plots of the observations in `y`, by group. Box plots provide a visual comparison of the group location parameters.

If `y` is a vector, then the plot shows one box for each value of `group`. If `y` is a matrix and you do not specify `group`, then the plot shows one box for each column of `y`. On each box, the central mark is the median and the edges of the box are the 25th and 75th percentiles (1st and 3rd quantiles). The whiskers extend to the most extreme data points that are not considered outliers. The outliers are plotted individually. The interval endpoints are the extremes of the notches. The extremes correspond to  $q2 - 1.57(q3 - q1)/\sqrt{n}$  and  $q2 + 1.57(q3 - q1)/\sqrt{n}$ , where  $q2$  is the median (50th percentile),  $q1$  and  $q3$  are the 25th and 75th percentiles, respectively, and  $n$  is the number of observations without any NaN values.

Two medians are significantly different at the 5% significance level if their intervals do not overlap. This test is different from the  $F$ -test that ANOVA performs, but large differences in the center lines of the boxes correspond to large  $F$ -statistic values and correspondingly small  $p$ -values. For more information about box plots, see `boxplot`.

### References

[1] Hogg, R. V., and J. Ledolter. *Engineering Statistics*. New York: MacMillan, 1987.

### See Also

`anova2` | `anovan` | `boxplot` | `multcompare`

### Topics

“Perform One-Way ANOVA” on page 9-6

“One-Way ANOVA” on page 9-3

“Multiple Comparisons” on page 9-26

Introduced before R2006a



## anova2

Two-way analysis of variance

anova2 performs two-way analysis of variance (ANOVA) with balanced designs. To perform two-way ANOVA with unbalanced designs, see `anovan`.

### Syntax

```
p = anova2(y, reps)
p = anova2(y, reps, displayopt)
[p, tbl] = anova2(____)
[p, tbl, stats] = anova2(____)
```

### Description

`p = anova2(y, reps)` returns the  $p$ -values for a balanced two-way ANOVA for comparing the means of two or more columns and two or more rows of the observations in `y`.

`reps` is the number of replicates for each combination of factor groups, which must be constant, indicating a balanced design. For unbalanced designs, use `anovan`. The `anova2` function tests the main effects for column and row factors and their interaction effect. To test the interaction effect, `reps` must be greater than 1.

`anova2` also displays the standard ANOVA table.

`p = anova2(y, reps, displayopt)` enables the ANOVA table display when `displayopt` is 'on' (default) and suppresses the display when `displayopt` is 'off'.

`[p, tbl] = anova2(____)` returns the ANOVA table (including column and row labels) in cell array `tbl`. To copy a text version of the ANOVA table to the clipboard, select **Edit > Copy Text** menu.

`[p, tbl, stats] = anova2(____)` returns a `stats` structure, which you can use to perform a multiple comparison test on page 9-26. A multiple comparison test enables you

to determine which pairs of group means are significantly different. To perform this test, use `multcompare`, providing the `stats` structure as input.

## Examples

### Two-Way ANOVA

Load the sample data.

```
load popcorn
popcorn
```

```
popcorn =
    5.5000    4.5000    3.5000
    5.5000    4.5000    4.0000
    6.0000    4.0000    3.0000
    6.5000    5.0000    4.0000
    7.0000    5.5000    5.0000
    7.0000    5.0000    4.5000
```

The data is from a study of popcorn brands and popper types (Hogg 1987). The columns of the matrix `popcorn` are brands, Gourmet, National, and Generic, respectively. The rows are popper types, oil and air. In the study, researchers popped a batch of each brand three times with each popper, that is, the number of replications is 3. The first three rows correspond to the oil popper, and the last three rows correspond to the air popper. The response values are the yield in cups of popped popcorn.

Perform a two-way ANOVA. Save the ANOVA table in the cell array `tbl` for easy access to results.

```
[p,tbl] = anova2(popcorn,3);
```

ANOVA Table					
Source	SS	df	MS	F	Prob>F
Columns	15.75	2	7.875	56.7	0
Rows	4.5	1	4.5	32.4	0.0001
Interaction	0.0833	2	0.04167	0.3	0.7462
Error	1.6667	12	0.13889		
Total	22	17			

The column `Prob>F` shows the  $p$ -values for the three brands of popcorn (0.0000), the two popper types (0.0001), and the interaction between brand and popper type (0.7462). These values indicate that popcorn brand and popper type affect the yield of popcorn, but there is no evidence of an interaction effect of the two.

Display the cell array containing the ANOVA table.

```
tbl
```

```
tbl =
```

```
6x6 cell array
```

Columns 1 through 5

```
{'Source'      } {'SS'      } {'df' } {'MS'      } {'F'      }
{'Columns'    } {[15.7500]} {[ 2]} {[ 7.8750]} {[ 56.7000]}
{'Rows'       } {[ 4.5000]} {[ 1]} {[ 4.5000]} {[ 32.4000]}
{'Interaction' } {[ 0.0833]} {[ 2]} {[ 0.0417]} {[ 0.3000]}
{'Error'      } {[ 1.6667]} {[12]} {[ 0.1389]} {0x0 double}
{'Total'      } {[      22]} {[17]} {0x0 double} {0x0 double}
```

Column 6

```
{'Prob>F'     }
{[7.6790e-07]}
{[1.0037e-04]}
{[ 0.7462]}
{0x0 double }
{0x0 double }
```

Store the  $F$ -statistic for the factors and factor interaction in separate variables.

```
Fbrands = tbl{2,5}
Fpoppertype = tbl{3,5}
Finteraction = tbl{4,5}
```

```
Fbrands =
```

```
56.7000
```

```
Fpoppertype =
```

```
32.4000
```

```
Finteraction =
```

```
0.3000
```

## Multiple Comparisons for Two-Way ANOVA

Load the sample data.

```
load popcorn
popcorn
```

```
popcorn =
    5.5000    4.5000    3.5000
    5.5000    4.5000    4.0000
    6.0000    4.0000    3.0000
    6.5000    5.0000    4.0000
    7.0000    5.5000    5.0000
    7.0000    5.0000    4.5000
```

The data is from a study of popcorn brands and popper types (Hogg 1987). The columns of the matrix `popcorn` are brands (Gourmet, National, and Generic). The rows are popper types oil and air. In the study, researchers popped a batch of each brand three times with each popper. The values are the yield in cups of popped popcorn.

Perform a two-way ANOVA. Also compute the statistics that you need to perform a multiple comparison test on the main effects.

```
[~,~,stats] = anova2(popcorn,3,'off')
```

```
stats =
    struct with fields:
        source: 'anova2'
        sigmasq: 0.1389
        colmeans: [6.2500 4.7500 4]
        coln: 6
        rowmeans: [4.5000 5.5000]
        rown: 9
        inter: 1
        pval: 0.7462
        df: 12
```

The `stats` structure includes

- The mean squared error (`sigmasq`)
- The estimates of the mean yield for each popcorn brand (`colmeans`)
- The number of observations for each popcorn brand (`coln`)
- The estimate of the mean yield for each popper type (`rowmeans`)
- The number of observations for each popper type (`rown`)
- The number of interactions (`inter`)
- The  $p$ -value that shows the significance level of the interaction term (`pval`)
- The error degrees of freedom (`df`).

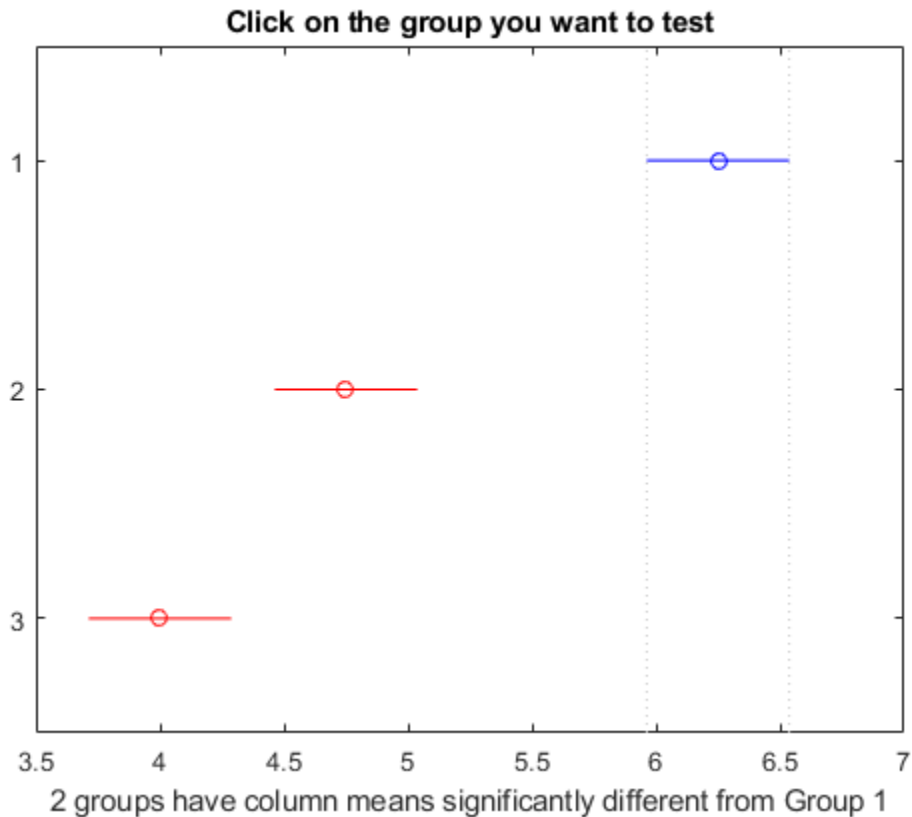
Perform a multiple comparison test to see if the popcorn yield differs between pairs of popcorn brands (columns).

```
c = multcompare(stats)
```

Note: Your model includes an interaction term. A test of main effects can be difficult to interpret when the model includes interactions.

```
c =
```

1.0000	2.0000	0.9260	1.5000	2.0740	0.0000
1.0000	3.0000	1.6760	2.2500	2.8240	0.0000
2.0000	3.0000	0.1760	0.7500	1.3240	0.0116



The first two columns of  $c$  show the groups that are compared. The fourth column shows the difference between the estimated group means. The third and fifth columns show the lower and upper limits for 95% confidence intervals for the true mean difference. The sixth column contains the  $p$ -value for a hypothesis test that the corresponding mean difference is equal to zero. All  $p$ -values (0, 0, and 0.0116) are very small, which indicates that the popcorn yield differs across all three brands.

The figure shows the multiple comparison of the means. By default, the group 1 mean is highlighted and the comparison interval is in blue. Because the comparison intervals for the other two groups do not intersect with the intervals for the group 1 mean, they are highlighted in red. This lack of intersection indicates that both means are different than group 1 mean. Select other group means to confirm that all group means are significantly different from each other.

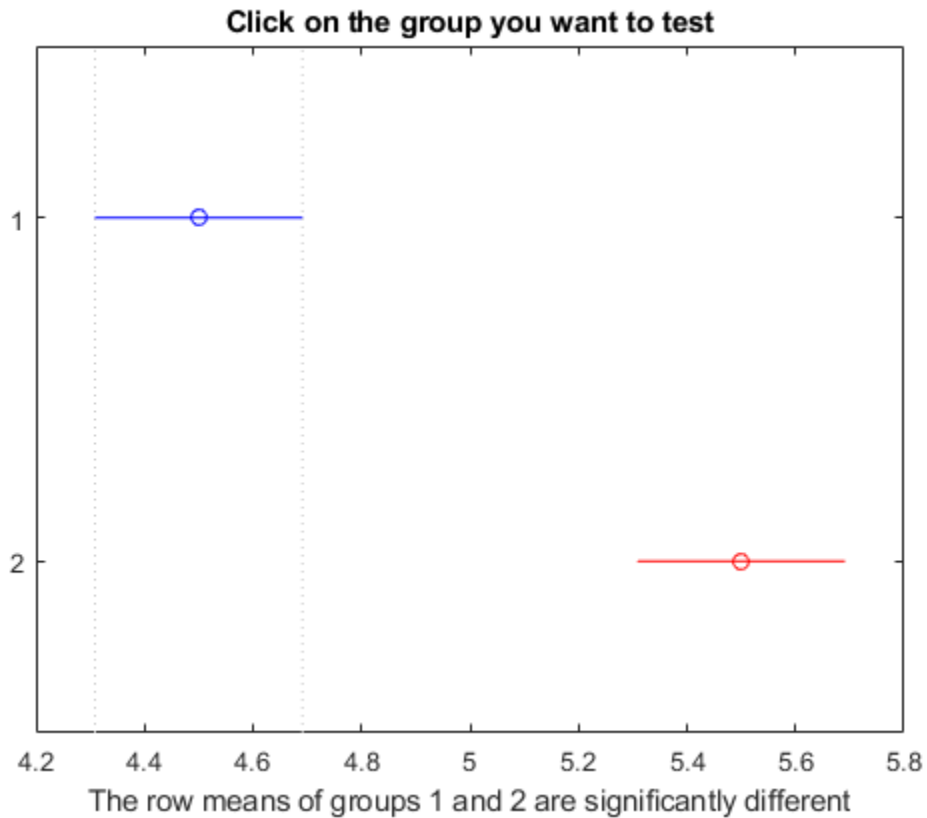
Perform a multiple comparison test to see the popcorn yield differs between the two popper types (rows).

```
c = multcompare(stats, 'Estimate', 'row')
```

Note: Your model includes an interaction term. A test of main effects can be difficult to interpret when the model includes interactions.

```
c =
```

```
1.0000    2.0000   -1.3828   -1.0000   -0.6172    0.0001
```





The small  $p$ -value of 0.0001 indicates that the popcorn yield differs between the two popper types (air and oil). The figure shows the same results. The disjoint comparison intervals indicate that the group means are significantly different from each other.

- “Perform Two-Way ANOVA” on page 9-18

## Input Arguments

### **y** — Sample data

matrix

Sample data, specified as a matrix. The columns correspond to groups of one factor, and the rows correspond to the groups of the other factor and the replications. Replications are the measurements or observations for each combination of groups (levels) of the row and column factor. For example, in the following data the row factor  $A$  has three levels, column factor  $B$  has two levels, and there are two replications ( $\text{reps} = 2$ ). The subscripts indicate row, column, and replication, respectively.

$$\begin{array}{cc}
 B = 1 & B = 2 \\
 \left[ \begin{array}{cc}
 y_{111} & y_{121} \\
 y_{112} & y_{122} \\
 y_{211} & y_{221} \\
 y_{212} & y_{222} \\
 y_{311} & y_{321} \\
 y_{312} & y_{322}
 \end{array} \right] \left. \begin{array}{l} \\ \\ \\ \\ \\ \\ \end{array} \right\} \begin{array}{l} A = 1 \\ \\ A = 2 \\ \\ A = 3 \end{array}
 \end{array}$$

Data Types: `single` | `double`

### **reps** — Number of replications

1 (default) | an integer number

Number of replications for each combination of groups, specified as an integer number. For example, the following data has two replications ( $\text{reps} = 2$ ) for each group combination of row factor  $A$  and column factor  $B$ .

$$\begin{array}{cc} B = 1 & B = 2 \\ \left. \begin{array}{cc} y_{111} & y_{121} \\ y_{112} & y_{122} \end{array} \right\} A = 1 \\ \left. \begin{array}{cc} y_{211} & y_{221} \\ y_{212} & y_{222} \end{array} \right\} A = 2 \\ \left. \begin{array}{cc} y_{311} & y_{321} \\ y_{312} & y_{322} \end{array} \right\} A = 3 \end{array}$$

- When `reps` is 1 (default), `anova2` returns two  $p$ -values in vector `p`:
  - The  $p$ -value for the null hypothesis that all samples from factor  $B$  (i.e., all column samples in  $y$ ) are drawn from the same population.
  - The  $p$ -value for the null hypothesis, that all samples from factor  $A$  (i.e., all row samples in  $y$ ) are drawn from the same population.
- When `reps` is greater than 1, `anova2` also returns the  $p$ -value for the null hypothesis that factors  $A$  and  $B$  have no interaction (i.e., the effects due to factors  $A$  and  $B$  are *additive*).

Example: `p = anova(y, 3)` specifies that each combination of groups (levels) has three replications.

Data Types: `single` | `double`

**displayopt** — Indicator to display the ANOVA table

`'on'` (default) | `'off'`

Indicator to display the ANOVA table as a figure, specified as `'on'` or `'off'`.

Data Types: `logical`

## Output Arguments

**p** —  $p$ -value

scalar value

$p$ -value for the  $F$ -test, returned as a scalar value. A small  $p$ -value indicates that the results are statistically significant. Common significance levels are 0.05 or 0.01. For example:

- A sufficiently small  $p$ -value for the null hypothesis for group means of row factor  $A$  suggests that at least one row-sample mean is significantly different from the other row-sample means; i.e., there is a main effect due to factor  $A$
- A sufficiently small  $p$ -value for the null hypothesis for group (level) means of column factor  $B$  suggests that at least one column-sample mean is significantly different from the other column-sample means; i.e., there is a main effect due to factor  $B$ .
- A sufficiently small  $p$ -value for combinations of groups (levels) of factors  $A$  and  $B$  suggests that there is an interaction between factors  $A$  and  $B$ .

### tbl — ANOVA table

cell array

ANOVA table, returned as a cell array. `tbl` has six columns.

Column name	Definition
source	Source of the variability.
SS	Sum of squares due to each source.
df	Degrees of freedom associated with each source.
MS	Mean squares for each source, which is the ratio $SS/df$ .
F	$F$ -statistic, which is the ratio of the mean squares.
Prob>F	$p$ -value, which is the probability that the $F$ -statistic can take a value larger than the computed test-statistic value. <code>anova2</code> derives this probability from the cdf of the $F$ -distribution.

The rows of the ANOVA table show the variability in the data, divided by the source into three or four parts, depending on the value of `reps`.

Row	Definition
Columns	Variability due to the differences among the column means
Rows	Variability due to the differences among the row means

Row	Definition
Interaction	Variability due to the interaction between rows and columns (if <code>reps</code> is greater than its default value of 1)
Error	Remaining variability not explained by any systematic source

Data Types: `cell`

### **stats** — Statistics for multiple comparison test structure

Statistics for multiple comparisons tests on page 9-26, returned as a structure. Use `multcompare` to perform multiple comparison tests, supplying `stats` as an input argument. `stats` has nine fields.

Field	Definition
<code>source</code>	Source of the <code>stats</code> output
<code>sigmasq</code>	Mean squared error
<code>colmeans</code>	Estimated values of the column means
<code>coln</code>	Number of observations for each group in columns
<code>rowmeans</code>	Estimated values of the row means
<code>rown</code>	Number of observations for each group in rows
<code>inter</code>	Number of interactions
<code>pval</code>	$p$ -value for the interaction term
<code>df</code>	Error degrees of freedom $(reps - 1) * r * c$ where <code>reps</code> is the number of replications and <code>c</code> and <code>r</code> are the number of groups in factors, respectively.

Data Types: `struct`

## References

[1] Hogg, R. V., and J. Ledolter. *Engineering Statistics*. New York: MacMillan, 1987.

## See Also

`anova1` | `anovan` | `multcompare`

## Topics

“Perform Two-Way ANOVA” on page 9-18

“Two-Way ANOVA” on page 9-15

“Multiple Comparisons” on page 9-26

**Introduced before R2006a**

## anovan

N-way analysis of variance

### Syntax

```
p = anovan(y,group)
p = anovan(y,group,Name,Value)
[p,tbl] = anovan(____)
[p,tbl,stats] = anovan(____)
[p,tbl,stats,terms] = anovan(____)
```

### Description

`p = anovan(y,group)` returns a vector of  $p$ -values, one per term, for multiway ( $n$ -way) analysis of variance (ANOVA) for testing the effects of multiple factors on the mean of the vector  $y$ .

`anovan` also displays a figure showing the standard ANOVA table.

`p = anovan(y,group,Name,Value)` returns a vector of  $p$ -values for multiway ( $n$ -way) ANOVA using additional options specified by one or more `Name, Value` pair arguments.

For example, you can specify which predictor variable is continuous, if any, or the type of sum of squares to use.

`[p,tbl] = anovan(____)` returns the ANOVA table (including factor labels) in cell array `tbl` for any of the input arguments specified in the previous syntaxes. Copy a text version of the ANOVA table to the clipboard by using the `Copy Text` item on the **Edit** menu.

`[p,tbl,stats] = anovan(____)` returns a `stats` structure that you can use to perform a multiple comparison test on page 9-26, which enables you to determine which pairs of group means are significantly different. You can perform such a test using the `multcompare` function by providing the `stats` structure as input.

`[p,tbl,stats,terms] = anovan(____)` returns the main and interaction terms used in the ANOVA computations in terms.

## Examples

### Three-Way ANOVA

Load the sample data.

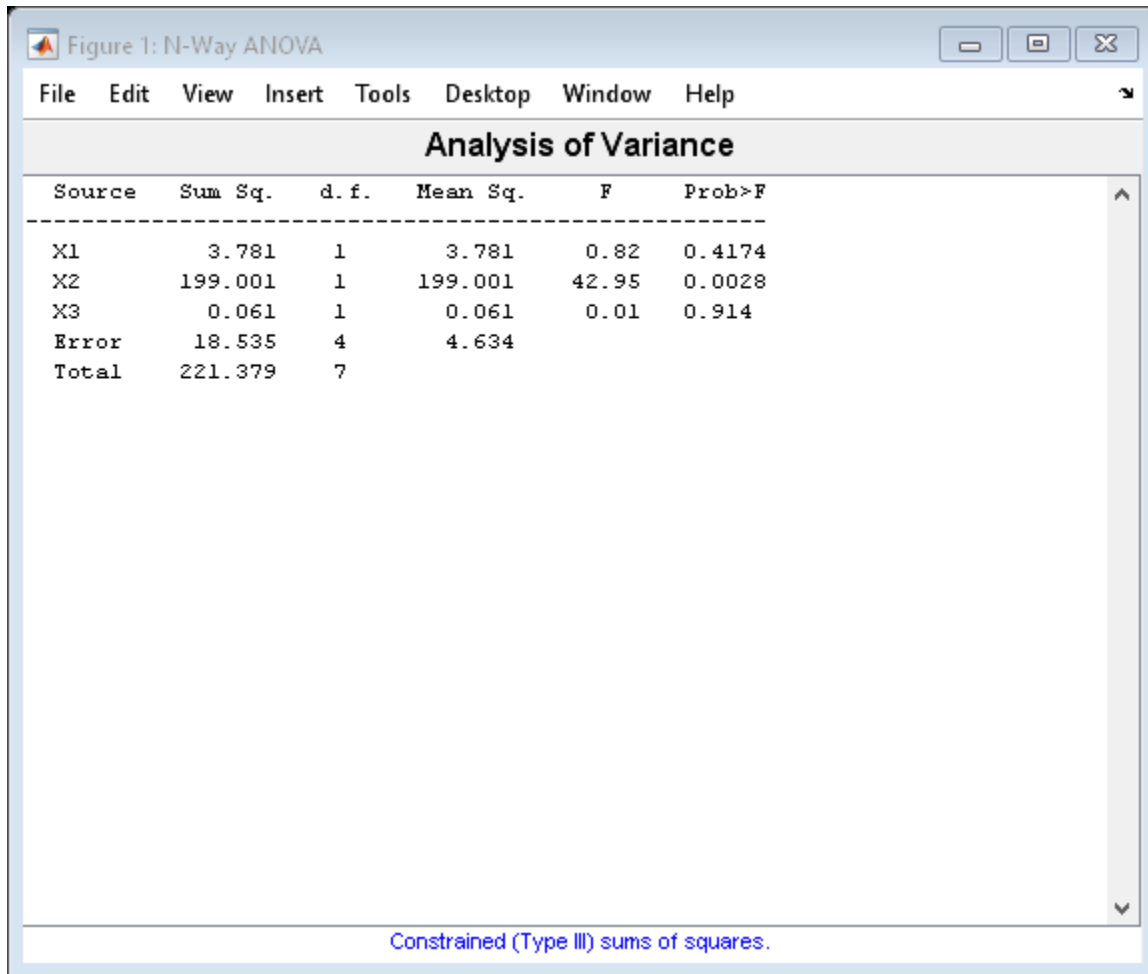
```
y = [52.7 57.5 45.9 44.5 53.0 57.0 45.9 44.0]';
g1 = [1 2 1 2 1 2 1 2];
g2 = {'hi'; 'hi'; 'lo'; 'lo'; 'hi'; 'hi'; 'lo'; 'lo'};
g3 = {'may'; 'may'; 'may'; 'may'; 'june'; 'june'; 'june'; 'june'};
```

`y` is the response vector and `g1`, `g2`, and `g3` are the grouping variables (factors). Each factor has two levels, and every observation in `y` is identified by a combination of factor levels. For example, observation `y(1)` is associated with level 1 of factor `g1`, level 'hi' of factor `g2`, and level 'may' of factor `g3`. Similarly, observation `y(6)` is associated with level 2 of factor `g1`, level 'hi' of factor `g2`, and level 'june' of factor `g3`.

Test if the response is the same for all factor levels.

```
p = anovan(y, {g1, g2, g3})
```

```
p =
    0.4174
    0.0028
    0.9140
```



In the ANOVA table,  $x_1$ ,  $x_2$ , and  $x_3$  correspond to the factors  $g_1$ ,  $g_2$ , and  $g_3$ , respectively. The  $p$ -value 0.4174 indicates that the mean responses for levels 1 and 2 of the factor  $g_1$  are not significantly different. Similarly, the  $p$ -value 0.914 indicates that the mean responses for levels 'may' and 'june', of the factor  $g_3$  are not significantly different. However, the  $p$ -value 0.0028 is small enough to conclude that the mean responses are significantly different for the two levels, 'hi' and 'lo' of the factor  $g_2$ . By default, `anovan` computes  $p$ -values just for the three main effects.



Test the two-factor interactions. This time specify the variable names.

```
p = anovan(y, {g1 g2 g3}, 'model', 'interaction', 'varnames', {'g1', 'g2', 'g3'})
```

```
p =
```

```
0.0347  
0.0048  
0.2578  
0.0158  
0.1444  
0.5000
```

Figure 2: N-Way ANOVA

File Edit View Insert Tools Desktop Window Help

### Analysis of Variance

Source	Sum Sq.	d. f.	Mean Sq.	F	Prob>F
g1	3.781	1	3.781	336.11	0.0347
g2	199.001	1	199.001	17689	0.0048
g3	0.061	1	0.061	5.44	0.2578
g1*g2	18.301	1	18.301	1626.78	0.0158
g1*g3	0.211	1	0.211	18.78	0.1444
g2*g3	0.011	1	0.011	1	0.5
Error	0.011	1	0.011		
Total	221.379	7			

Constrained (Type III) sums of squares.

The interaction terms are represented by  $g_1 * g_2$ ,  $g_1 * g_3$ , and  $g_2 * g_3$  in the ANOVA table. The first three entries of  $p$  are the  $p$ -values for the main effects. The last three entries are the  $p$ -values for the two-way interactions. The  $p$ -value of 0.0158 indicates that the interaction between  $g_1$  and  $g_2$  is significant. The  $p$ -values of 0.1444 and 0.5 indicate that the corresponding interactions are not significant.

## Two-Way ANOVA for Unbalanced Design

Load the sample data.

```
load carbig
```

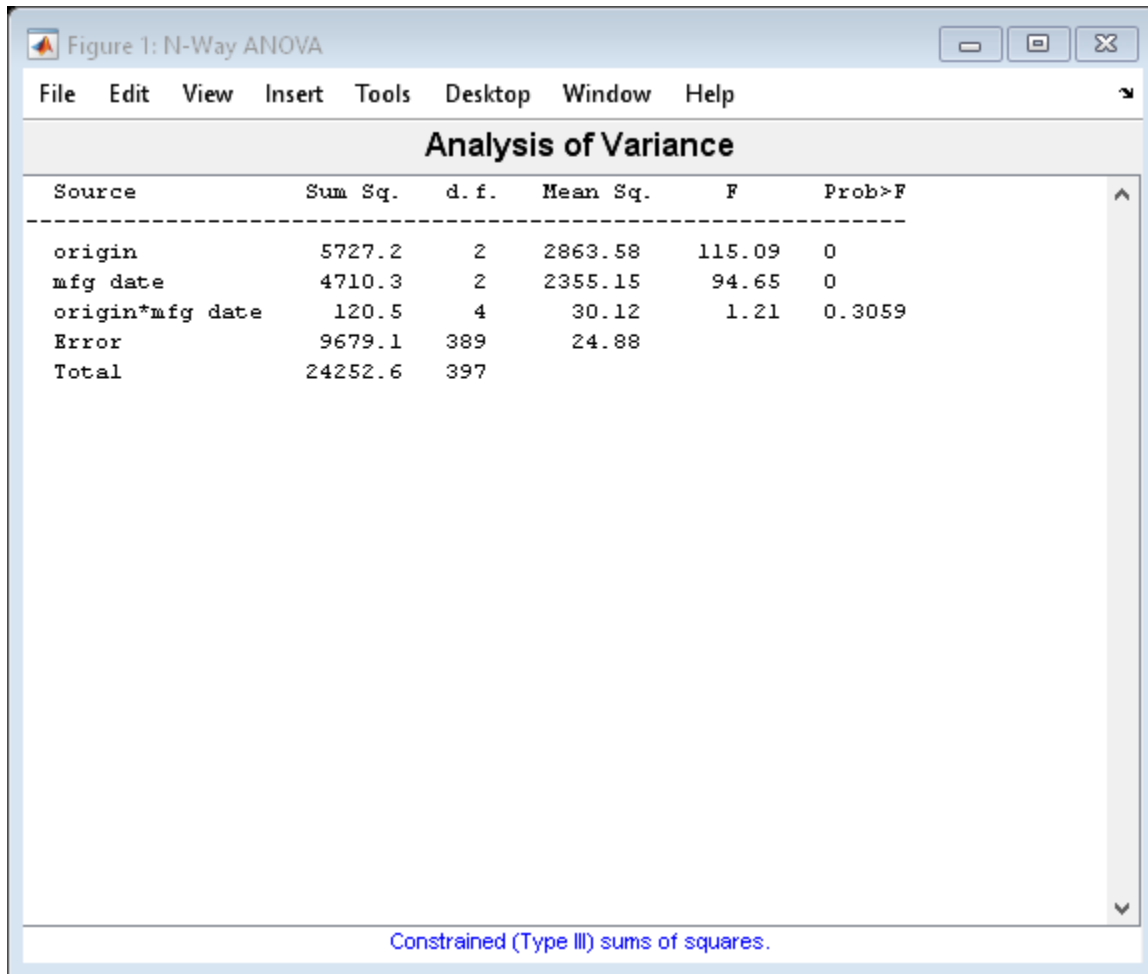
The data has measurements on 406 cars. The variable `org` shows where the cars were made and `when` shows when in the year the cars were manufactured.

Study how the mileage depends on when and where the cars were made. Also include the two-way interactions in the model.

```
p = anovan(MPG, {org when}, 'model', 2, 'varnames', {'origin', 'mfg date'})
```

```
p =
```

```
0.0000  
0.0000  
0.3059
```



The 'model', 2 name-value pair argument represents the two-way interactions. The  $p$ -value for the interaction term, 0.3059, is not small, indicating little evidence that the effect of the time of manufacture (mfg date) depends on where the car was made (origin). The main effects of origin and manufacturing date, however, are significant, both  $p$ -values are 0.

## Multiple Comparisons for Three-Way ANOVA

Load the sample data.

```
y = [52.7 57.5 45.9 44.5 53.0 57.0 45.9 44.0]';  
g1 = [1 2 1 2 1 2 1 2];  
g2 = {'hi'; 'hi'; 'lo'; 'lo'; 'hi'; 'hi'; 'lo'; 'lo'};  
g3 = {'may'; 'may'; 'may'; 'may'; 'june'; 'june'; 'june'; 'june'};
```

$y$  is the response vector and  $g_1$ ,  $g_2$ , and  $g_3$  are the grouping variables (factors). Each factor has two levels, and every observation in  $y$  is identified by a combination of factor levels. For example, observation  $y(1)$  is associated with level 1 of factor  $g_1$ , level 'hi' of factor  $g_2$ , and level 'may' of factor  $g_3$ . Similarly, observation  $y(6)$  is associated with level 2 of factor  $g_1$ , level 'hi' of factor  $g_2$ , and level 'june' of factor  $g_3$ .

Test if the response is the same for all factor levels. Also compute the statistics required for multiple comparison tests.

```
[~,~,stats] = anovan(y,{g1 g2 g3},'model','interaction',...  
    'varnames',{'g1','g2','g3'});
```

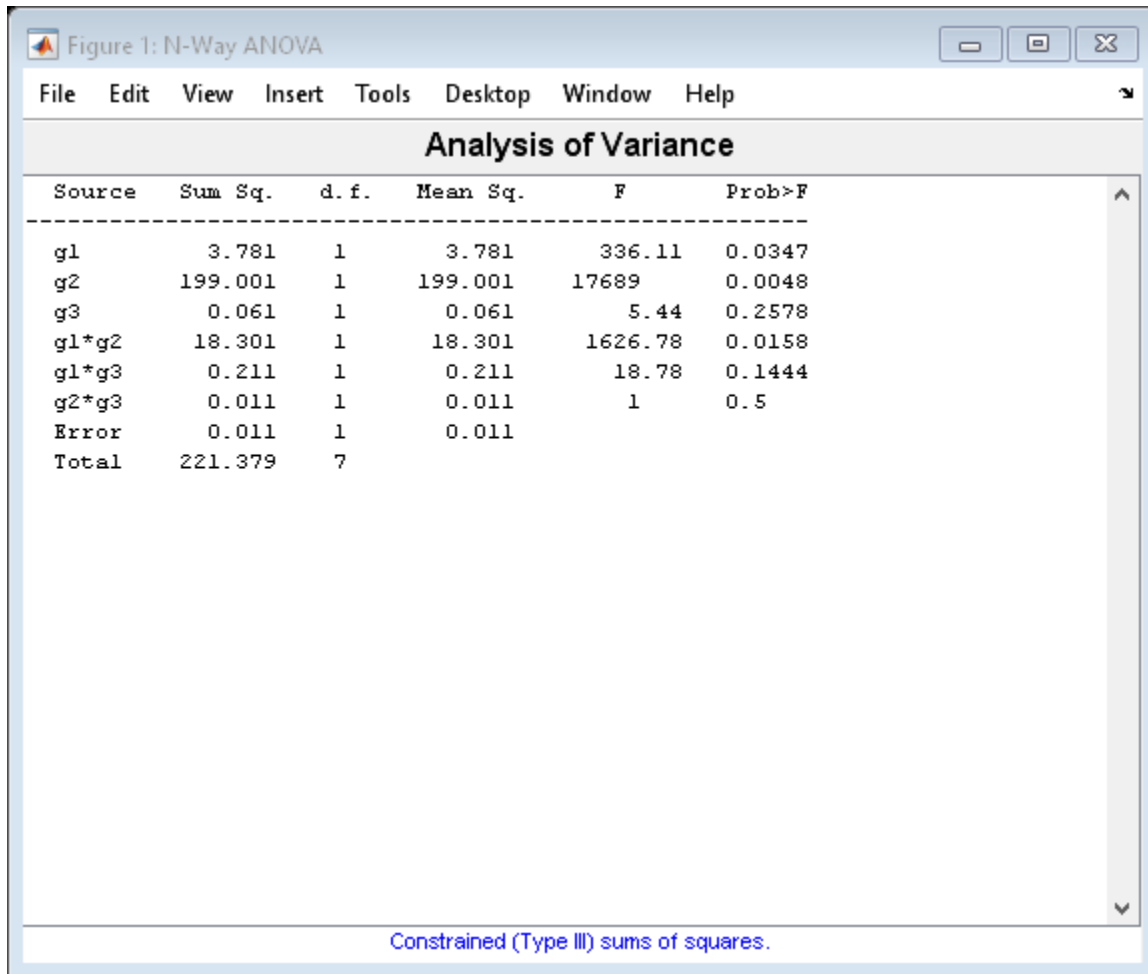


Figure 1: N-Way ANOVA

File Edit View Insert Tools Desktop Window Help

### Analysis of Variance

Source	Sum Sq.	d. f.	Mean Sq.	F	Prob>F
g1	3.781	1	3.781	336.11	0.0347
g2	199.001	1	199.001	17689	0.0048
g3	0.061	1	0.061	5.44	0.2578
g1*g2	18.301	1	18.301	1626.78	0.0158
g1*g3	0.211	1	0.211	18.78	0.1444
g2*g3	0.011	1	0.011	1	0.5
Error	0.011	1	0.011		
Total	221.379	7			

Constrained (Type III) sums of squares.

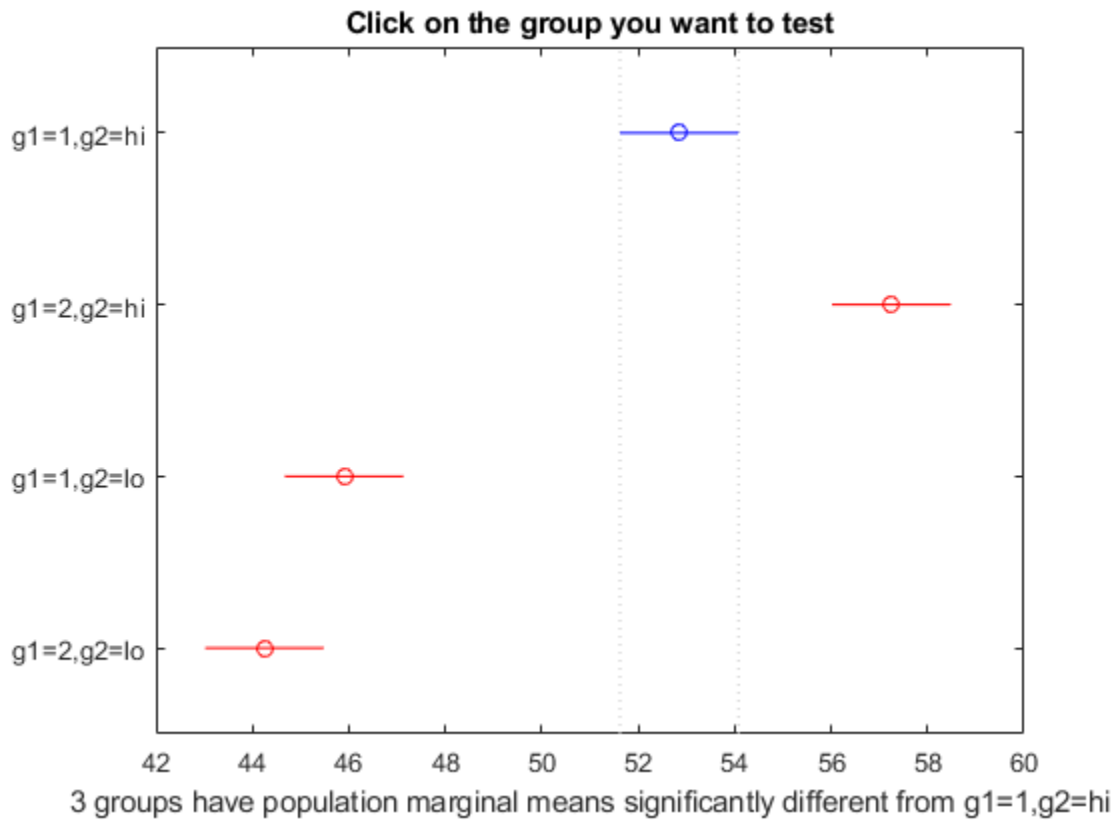
The  $p$ -value of 0.2578 indicates that the mean responses for levels 'may' and 'june' of factor  $g_3$  are not significantly different. The  $p$ -value of 0.0347 indicates that the mean responses for levels 1 and 2 of factor  $g_1$  are significantly different. Similarly, the  $p$ -value of 0.0048 indicates that the mean responses for levels 'hi' and 'lo' of factor  $g_2$  are significantly different.

Perform multiple comparison tests to find out which groups of the factors  $g_1$  and  $g_2$  are significantly different.

```
results = multcompare(stats, 'Dimension', [1 2])
```

```
results =
```

1.0000	2.0000	-6.8604	-4.4000	-1.9396	0.0280
1.0000	3.0000	4.4896	6.9500	9.4104	0.0177
1.0000	4.0000	6.1396	8.6000	11.0604	0.0143
2.0000	3.0000	8.8896	11.3500	13.8104	0.0108
2.0000	4.0000	10.5396	13.0000	15.4604	0.0095
3.0000	4.0000	-0.8104	1.6500	4.1104	0.0745



`multcompare` compares the combinations of groups (levels) of the two grouping variables, `g1` and `g2`. In the `results` matrix, the number 1 corresponds to the combination of level 1 of `g1` and level `hi` of `g2`, the number 2 corresponds to the combination of level 2 of `g1` and level `hi` of `g2`. Similarly, the number 3 corresponds to the combination of level 1 of `g1` and level `lo` of `g2`, and the number 4 corresponds to the combination of level 2 of `g1` and level `lo` of `g2`. The last column of the matrix contains the  $p$ -values.

For example, the first row of the matrix shows that the combination of level 1 of `g1` and level `hi` of `g2` has the same mean response values as the combination of level 2 of `g1` and level `hi` of `g2`. The  $p$ -value corresponding to this test is 0.0280, which indicates that the mean responses are significantly different. You can also see this result in the figure. The blue bar shows the comparison interval for the mean response for the combination of level 1 of `g1` and level `hi` of `g2`. The red bars are the comparison intervals for the mean response for other group combinations. None of the red bars overlap with the blue bar, which means the mean response for the combination of level 1 of `g1` and level `hi` of `g2` is significantly different from the mean response for other group combinations.

You can test the other groups by clicking on the corresponding comparison interval for the group. The bar you click on turns to blue. The bars for the groups that are significantly different are red. The bars for the groups that are not significantly different are gray. For example, if you click on the comparison interval for the combination of level 1 of `g1` and level `lo` of `g2`, the comparison interval for the combination of level 2 of `g1` and level `lo` of `g2` overlaps, and is therefore gray. Conversely, the other comparison intervals are red, indicating significant difference.

- “Perform N-Way ANOVA” on page 9-39
- “ANOVA with Random Effects” on page 9-49
- “Multiple Comparisons” on page 9-26

## Input Arguments

### **y** — Sample data

numeric vector

Sample data, specified as a numeric vector.

Data Types: `single` | `double`



**group — Grouping variables**

cell array

Grouping variables, i.e. the factors and factor levels of the observations in  $y$ , specified as a cell array. Each of the cells in `group` contains a list of factor levels identifying the observations in  $y$  with respect to one of the factors. The list within each cell can be a categorical array, numeric vector, character matrix, or single-column cell array of character vectors, and must have the same number of elements as  $y$ .

$$y = [ \begin{matrix} y_1 & y_2 & y_3 & y_4 & y_5 & \dots & y_N \end{matrix} ]$$

$$\begin{matrix} \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & & \uparrow \end{matrix}$$

$$g1 = \{ 'A', 'A', 'C', 'B', 'B', \dots, 'D' \}$$

$$g2 = [ 1 \quad 2 \quad 1 \quad 3 \quad 1 \quad \dots, \quad 2 ]$$

$$g3 = \{ 'hi', 'mid', 'low', 'mid', 'hi', \dots, 'low' \}$$

By default, `anovan` treats all grouping variables as fixed effects.

For example, in a study you want to investigate the effects of gender, school, and the education method on the academic success of elementary school students, then you can specify the grouping variables as follows.

Example: `{ 'Gender', 'School', 'Method' }`

Data Types: `cell`

**Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( `' '` ). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'alpha', 0.01, 'model', 'interaction', 'sstype', 2` specifies `anovan` to compute the 99% confidence bounds and p-values for the main effects and two-way interactions using type II sum of squares.

**alpha — Confidence level**

0.05 (default) | scalar value in the range 0 to 1

Confidence level for confidence bounds, specified as the comma-separated pair consisting of 'alpha' and a scalar value in the range 0 to 1. For a value  $\alpha$ , the confidence level is  $100*(1-\alpha)\%$ .

Example: 'alpha', 0.01 corresponds to 99% confidence intervals

Data Types: single | double

### **continuous** — Indicator for continuous predictors

vector of indices

Indicator for continuous predictors, representing which grouping variables should be treated as continuous predictors rather than as categorical predictors, specified as the comma-separated pair consisting of 'continuous' and a vector of indices.

For example, if there are three grouping variables and second one is continuous, then you can specify as follows.

Example: continuous', [2]

Data Types: single | double

### **display** — Indicator to display ANOVA table

'on' (default) | 'off'

Indicator to display ANOVA table, specified as the comma-separated pair consisting of 'display' and 'on' or 'off'. When displayopt is 'off', anova1 only returns the output arguments, and does not display the standard ANOVA table as a figure.

Example: 'display', 'off'

### **model** — Type of the model

'linear' (default) | 'interaction' | 'full' | integer value | terms matrix

Type of the model, specified as the comma-separated pair consisting of 'model' and one of the following:

- 'linear' — The default 'linear' model computes only the  $p$ -values for the null hypotheses on the  $N$  main effects.
- 'interaction' — The 'interaction' model computes the  $p$ -values for null

hypotheses on the  $N$  main effects and the  $\binom{N}{2}$  two-factor interactions.

- 'full' — The 'full' model computes the  $p$ -values for null hypotheses on the  $N$  main effects and interactions at all levels.
- An integer — For an integer value of  $k$ , ( $k \leq N$ ) for model type, `anovan` computes all interaction levels through the  $k$ th level. For example, the value 3 means main effects plus two- and three-factor interactions. The values  $k = 1$  and  $k = 2$  are equivalent to the 'linear' and 'interaction' specifications, respectively. The value  $k = N$  is equivalent to the 'full' specification.
- Terms matrix — A matrix of term definitions having the same form as the input to the `x2fx` function. All entries must be 0 or 1 (no higher powers).

For more precise control over the main and interaction terms that `anovan` computes, you can specify a matrix containing one row for each main or interaction term to include in the ANOVA model. Each row defines one term using a vector of  $N$  zeros and ones. The table below illustrates the coding for a 3-factor ANOVA for factors  $A$ ,  $B$ , and  $C$ .

Matrix Row	ANOVA Term
[1 0 0]	Main term $A$
[0 1 0]	Main term $B$
[0 0 1]	Main term $C$
[1 1 0]	Interaction term $AB$
[1 0 1]	Interaction term $AC$
[0 1 1]	Interaction term $BC$
[1 1 1]	Interaction term $ABC$

For example, if there are three factors  $A$ ,  $B$ , and  $C$ , and 'model', [0 1 0;0 0 1;0 1 1], then `anovan` tests for the main effects  $B$  and  $C$ , and the interaction effect  $BC$ , respectively.

A simple way to generate the terms matrix is to modify the `terms` output, which codes the terms in the current model using the format described above. If `anovan` returns [0 1 0;0 0 1;0 1 1] for `terms`, for example, and there is no significant interaction  $BC$ , then you can recompute ANOVA on just the main effects  $B$  and  $C$  by specifying [0 1 0;0 0 1] for `model`.

Example: 'model', [0 1 0;0 0 1;0 1 1]

Example: 'model', 'interaction'

**nested** — Nesting relationships

matrix of 0's and 1's

Nesting relationships among the grouping variables, specified as the comma-separated pair consisting of 'nested' and a matrix  $M$  of 0's and 1's, i.e.  $M(i,j) = 1$  if variable  $i$  is nested in variable  $j$ .

For example, if there are two grouping variables District and School, where School is nested in District, then you can express this relationship as follows.

Example: 'nested', [0,0;1 0]

Data Types: single | double

**random** — Indicator for random variables

vector of indices

Indicator for random variables, representing which grouping variables are random, specified as the comma-separated pair consisting of 'random' and a vector of indices. By default, anovan treats all grouping variables as fixed.

anovan treats an interaction term as random if any of the variables in the interaction term is random.

Example: 'random', [3]

Data Types: single | double

**sstype** — Type of sum of squares

3 (default) | 1 | 2 | h

Type of sum squares, specified as the comma-separated pair consisting of 'sstype' and the following:

- 1 — Type I sum of squares. The reduction in residual sum of squares obtained by adding that term to a fit that already includes the terms listed before it.
- 2 — Type II sum of squares. The reduction in residual sum of squares obtained by adding that term to a model consisting of all other terms that do not contain the term in question.
- 3 — Type III sum of squares. The reduction in residual sum of squares obtained by adding that term to a model containing all other terms, but with their effects constrained to obey the usual “sigma restrictions” that make models estimable.

- **h** — Hierarchical model. Similar to type 2, but with continuous as well as categorical factors used to determine the hierarchy of terms.

The sum of squares for any term is determined by comparing two models. For a model containing main effects but no interactions, the value of `sstype` only influences computations on unbalanced data.

Suppose you are fitting a model with two factors and their interaction, and that the terms appear in the order  $A, B, AB$ . Let  $R(\cdot)$  represent the residual sum of squares for a model, so for example  $R(A, B, AB)$  is the residual sum of squares fitting the whole model,  $R(A)$  is the residual sum of squares fitting just the main effect of  $A$ , and  $R(1)$  is the residual sum of squares fitting just the mean. The three types of sums of squares are as follows:

Term	Type 1 Sum of Squares	Type 2 Sum of Squares	Type 3 Sum of Squares
$A$	$R(1) - R(A)$	$R(B) - R(A, B)$	$R(B, AB) - R(A, B, AB)$
$B$	$R(A) - R(A, B)$	$R(A) - R(A, B)$	$R(A, AB) - R(A, B, AB)$
$AB$	$R(A, B) - R(A, B, AB)$	$R(A, B) - R(A, B, AB)$	$R(A, B) - R(A, B, AB)$

The models for Type 3 sum of squares have sigma restrictions imposed. This means, for example, that in fitting  $R(B, AB)$ , the array of  $AB$  effects is constrained to sum to 0 over  $A$  for each value of  $B$ , and over  $B$  for each value of  $A$ .

Example: `'sstype', 'h'`

Data Types: `single | double`

#### **varnames** — Names of grouping variables

$x_1, x_2, \dots, x_N$  (default) | character matrix | cell array of character vectors

Names of grouping variables, specified as the comma-separating pair consisting of `'varnames'` and a character matrix or a cell array of character vectors.

Example: `'varnames', {'Gender', 'City'}`

Data Types: `char | cell`

## Output Arguments

### **p** — $p$ -values

vector

$p$ -values, returned as a vector.

Output vector `p` contains  $p$ -values for the null hypotheses on the  $N$  main effects and any interaction terms specified. Element `p(1)` contains the  $p$ -value for the null hypotheses that samples at all levels of factor  $A$  are drawn from the same population; element `p(2)` contains the  $p$ -value for the null hypotheses that samples at all levels of factor  $B$  are drawn from the same population; and so on.

For example, if there are three factors  $A$ ,  $B$ , and  $C$ , and `'model'`, `[0 1 0;0 0 1;0 1 1]`, then the output vector `p` contains the  $p$ -values for the null hypotheses on the main effects  $B$  and  $C$  and the interaction effect  $BC$ , respectively.

A sufficiently small  $p$ -value corresponding to a factor suggests that at least one group mean is significantly different from the other group means; that is, there is a main effect due to that factor. It is common to declare a result significant if the  $p$ -value is less than 0.05 or 0.01.

### **tb1 — ANOVA table**

cell array

ANOVA table, returned as a cell array. The ANOVA table has seven columns:

Column name	Definition
source	Source of the variability.
SS	Sum of squares due to each source.
df	Degrees of freedom associated with each source.
MS	Mean squares for each source, which is the ratio $SS/df$ .
Singular?	Indication of whether the term is singular.
F	$F$ -statistic, which is the ratio of the mean squares.
Prob>F	The $p$ -values, which is the probability that the $F$ -statistic can take a value larger than a computed test-statistic value. <code>anovan</code> derives these probabilities from the cdf of $F$ -distribution.

The ANOVA table also contains the following columns if at least one of the grouping variables is specified as random using the name-value pair argument 'random':

Column name	Definition
Type	Type of each source; 'fixed' for a fixed effect or 'random' for a random effect.
Expected MS	Text representation of the expected value for the mean square. $Q(\text{source})$ represents a quadratic function of source and $V(\text{source})$ represents the variance of source.
MS denom	Denominator of the $F$ -statistic.
d.f. denom	Degrees of freedom for the denominator of the $F$ -statistic.
Denom. defn.	Text representation of the denominator of the $F$ -statistic. $MS(\text{source})$ represents the mean square of source.
Var. est.	Variance component estimate.
Var. lower bnd	Lower bound of the 95% confidence interval for the variance component estimate.
Var. upper bnd	Upper bound of the 95% confidence interval for the variance component estimate.

### stats — Statistics

structure

Statistics to use in a multiple comparison test on page 9-26 using the `multcompare` function, returned as a structure.

`anovan` evaluates the hypothesis that the different groups (levels) of a factor (or more generally, a term) have the same effect, against the alternative that they do not all have the same effect. Sometimes it is preferable to perform a test to determine which pairs of levels are significantly different, and which are not. Use the `multcompare` function to perform such tests by supplying the `stats` structure as input.

The `stats` structure contains the fields listed below, in addition to a number of other fields required for doing multiple comparisons using the `multcompare` function:

Field	Description
<code>coeffs</code>	Estimated coefficients
<code>coeffnames</code>	Name of term for each coefficient
<code>vars</code>	Matrix of grouping variable values for each term
<code>resid</code>	Residuals from the fitted model

The `stats` structure also contains the following fields if at least one of the grouping variables is specified as random using the name-value pair argument `'random'`:

Field	Description
<code>ems</code>	Expected mean squares
<code>denom</code>	Denominator definition
<code>rtnames</code>	Names of random terms
<code>varest</code>	Variance component estimates (one per random term)
<code>varci</code>	Confidence intervals for variance components

### **terms** — Main and interaction terms

matrix

Main and interaction terms, returned as a matrix. The terms are encoded in the output matrix `terms` using the same format described above for input `model`. When you specify `model` itself in this format, the matrix returned in `terms` is identical.

## References

- [1] Dunn, O.J., and V.A. Clark. *Applied Statistics: Analysis of Variance and Regression*. New York: Wiley, 1974.
- [2] Goodnight, J.H., and F.M. Speed. *Computing Expected Mean Squares*. Cary, NC: SAS Institute, 1978.
- [3] Seber, G. A. F. and A. J. Lee. *Linear Regression Analysis*. 2nd ed. Hoboken, NJ: Wiley-Interscience, 2003.

## See Also

`anova1` | `anova2` | `fitrm` | `multcompare` | `ranova`



## Topics

“Perform N-Way ANOVA” on page 9-39

“ANOVA with Random Effects” on page 9-49

“Multiple Comparisons” on page 9-26

“N-Way ANOVA” on page 9-36

**Introduced before R2006a**

## anova

**Class:** RepeatedMeasuresModel

Analysis of variance for between-subject effects

## Syntax

```
anovatbl = anova(rm)
anovatbl = anova(rm, 'WithinModel', WM)
```

## Description

`anovatbl = anova(rm)` returns the analysis of variance results for the repeated measures model `rm`.

`anovatbl = anova(rm, 'WithinModel', WM)` returns the analysis of variance results it performs using the response or responses specified by the within-subject model `WM`.

## Input Arguments

**rm** — Repeated measures model

RepeatedMeasuresModel object

Repeated measures model, returned as a RepeatedMeasuresModel object.

For properties and methods of this object, see RepeatedMeasuresModel.

**WM** — Within-subject model

'separatemeans' (default) | 'orthogonalcontrasts' | character vector defining a model specification | *r*-by-*nc* matrix specifying *nc* contrasts

Within-subject model, specified as one of the following:

- 'separatemeans' — The response is the average of the repeated measures (average across the within-subject model).

- `'orthogonalcontrasts'` — This is valid when the within-subject model has a single numeric factor  $T$ . Responses are the average, the slope of centered  $T$ , and, in general, all orthogonal contrasts for a polynomial up to  $T^{(p-1)}$ , where  $p$  is the number of rows in the within-subject model. `anova` multiplies  $Y$ , the response you use in the repeated measures model `rm` by the orthogonal contrasts, and uses the columns of the resulting product matrix as the responses.

`anova` computes the orthogonal contrasts for  $T$  using the  $Q$  factor of a QR factorization on page 32-114 of the Vandermonde matrix on page 32-114.

- A character vector that defines a model specification in the within-subject factors. Responses are defined by the terms in that model. `anova` multiplies  $Y$ , the response matrix you use in the repeated measures model `rm` by the terms of the model, and uses the columns of the result as the responses.

For example, if there is a Time factor and `'Time'` is the model specification, then `anova` uses two terms, the constant and the uncentered Time term. The default is `'1'` to perform on the average response.

- An  $r$ -by- $nc$  matrix,  $C$ , specifying  $nc$  contrasts among the  $r$  repeated measures. If  $Y$  represents the matrix of repeated measures you use in the repeated measures model `rm`, then the output `tbl` contains a separate analysis of variance for each column of  $Y * C$ .

The `anova` table contains a separate univariate analysis of variance results for each response.

Example: `'WithinModel', 'Time'`

Example: `'WithinModel', 'orthogonalcontrasts'`

## Output Arguments

### `anovatbl` — Results of analysis of variance

table

Results of analysis of variance for between-subject effects, returned as a table. This includes all terms on the between-subjects model and the following columns.

Column Name	Definition
Within	Within-subject factors

Column Name	Definition
Between	Between-subject factors
SumSq	Sum of squares
DF	Degrees of freedom
MeanSq	Mean squared error
F	$F$ -statistic
pValue	$p$ -value corresponding to the $F$ -statistic

## Examples

### Analysis of Variance for Average Response

Load the sample data.

```
load fisheriris
```

The column vector `species` consists of iris flowers of three different species: `setosa`, `versicolor`, and `virginica`. The double matrix `meas` consists of four types of measurements on the flowers: the length and width of sepals and petals in centimeters, respectively.

Store the data in a table array.

```
t = table(species,meas(:,1),meas(:,2),meas(:,3),meas(:,4),...
'VariableNames',{'species','meas1','meas2','meas3','meas4'});
Meas = dataset([1 2 3 4]','VarNames',{'Measurements'});
```

Fit a repeated measures model where the measurements are the responses and the species is the predictor variable.

```
rm = fitrm(t,'meas1-meas4~species','WithinDesign',Meas);
```

Perform analysis of variance.

```
anova(rm)
```

```
ans =
```

3x7 table

Within	Between	SumSq	DF	MeanSq	F	pValue
Constant	constant	7201.7	1	7201.7	19650	2.0735e-158
Constant	species	309.61	2	154.8	422.39	1.1517e-61
Constant	Error	53.875	147	0.36649		

There are 150 observations and 3 species. The degrees of freedom for species is  $3 - 1 = 2$ , and for error it is  $150 - 3 = 147$ . The small *P*-value of 1.1517e-61 indicates that the measurements differ significantly according to species.

## Panel Data

Load the sample panel data.

```
load(fullfile(matlabroot, 'examples', 'stats', 'panelData.mat'));
```

The dataset array, `panelData`, contains yearly observations on eight cities for 6 years. The first variable, `Growth`, measures economic growth (the response variable). The second and third variables are city and year indicators, respectively. The last variable, `Employ`, measures employment (the predictor variable). This is simulated data.

Store the data in a table array and define city as a nominal variable.

```
t = table(panelData.Growth, panelData.City, panelData.Year, ...
    'VariableNames', {'Growth', 'City', 'Year'});
```

Convert the data in a proper format to do repeated measures analysis.

```
t = unstack(t, 'Growth', 'Year', 'NewDataVariableNames', ...
    {'year1', 'year2', 'year3', 'year4', 'year5', 'year6'});
```

Add the mean employment level over the years as a predictor variable to the table `t`.

```
t(:,8) = table(grpstats(panelData.Employ, panelData.City));
t.Properties.VariableNames{'Var8'} = 'meanEmploy';
```

Define the within-subjects variable.

```
Year = [1 2 3 4 5 6]';
```

Fit a repeated measures model, where the growth figures over the 6 years are the responses and the mean employment is the predictor variable.

```
rm = fitrm(t, 'year1-year6 ~ meanEmploy', 'WithinDesign', Year);
```

Perform analysis of variance.

```
anovatbl = anova(rm, 'WithinModel', Year)
```

```
anovatbl =
```

```
3x7 table
```

Within	Between	SumSq	DF	MeanSq	F	pValue
Contrast1	constant	588.17	1	588.17	0.038495	0.85093
Contrast1	meanEmploy	3.7064e+05	1	3.7064e+05	24.258	0.0026428
Contrast1	Error	91675	6	15279		

## Longitudinal Data

Load the sample data.

```
load(fullfile(matlabroot, 'examples', 'stats', 'longitudinalData.mat'));
```

The matrix  $Y$  contains response data for 16 individuals. The response is the blood level of a drug measured at five time points (time = 0, 2, 4, 6, and 8). Each row of  $Y$  corresponds to an individual, and each column corresponds to a time point. The first eight subjects are female, and the second eight subjects are male. This is simulated data.

Define a variable that stores gender information.

```
Gender = ['F' 'F' 'F' 'F' 'F' 'F' 'F' 'F' 'M' 'M' 'M' 'M' 'M' 'M' 'M' 'M'];
```

Store the data in a proper table array format to do repeated measures analysis.

```
t = table(Gender, Y(:,1), Y(:,2), Y(:,3), Y(:,4), Y(:,5), ...  
'VariableNames', {'Gender', 't0', 't2', 't4', 't6', 't8'});
```

Define the within-subjects variable.

```
Time = [0 2 4 6 8]';
```

Fit a repeated measures model, where blood levels are the responses and gender is the predictor variable.

```
rm = fitrm(t, 't0-t8 ~ Gender', 'WithinDesign', Time);
```

Perform analysis of variance.

```
anovatbl = anova(rm)
```

```
anovatbl =
```

```
3x7 table
```

Within	Between	SumSq	DF	MeanSq	F	pValue
Constant	constant	54702	1	54702	1079.2	1.1897e-14
Constant	Gender	2251.7	1	2251.7	44.425	1.0693e-05
Constant	Error	709.6	14	50.685		

There are 2 genders and 16 observations, so the degrees of freedom for gender is  $(2 - 1) = 1$  and for error it is  $(16 - 2) * (2 - 1) = 14$ . The small  $P$ -value of  $1.0693e-05$  indicates that there is a significant effect of gender on blood pressure.

Repeat analysis of variance using orthogonal contrasts.

```
anovatbl = anova(rm, 'WithinModel', 'orthogonalcontrasts')
```

```
anovatbl =
```

```
15x7 table
```

Within	Between	SumSq	DF	MeanSq	F	pValue
Constant	constant	54702	1	54702	1079.2	1.1897e-14
Constant	Gender	2251.7	1	2251.7	44.425	1.0693e-05
Constant	Error	709.6	14	50.685		
Time	constant	310.83	1	310.83	31.023	6.9065e-05
Time	Gender	13.341	1	13.341	1.3315	0.26785

Time	Error	140.27	14	10.019		
Time^2	constant	565.42	1	565.42	98.901	1.0003e-07
Time^2	Gender	1.4076	1	1.4076	0.24621	0.62746
Time^2	Error	80.039	14	5.7171		
Time^3	constant	2.6127	1	2.6127	1.4318	0.25134
Time^3	Gender	7.8853e-06	1	7.8853e-06	4.3214e-06	0.99837
Time^3	Error	25.546	14	1.8247		
Time^4	constant	2.8404	1	2.8404	0.47924	0.50009
Time^4	Gender	2.9016	1	2.9016	0.48956	0.49559
Time^4	Error	82.977	14	5.9269		

## Definitions

### Vandermonde Matrix

Vandermonde matrix is the matrix where columns are the powers of the vector  $a$ , that is,  $V(i,j) = a(i)^{(n-j)}$ , where  $n$  is the length of  $a$ .

### QR Factorization

QR factorization of an  $m$ -by- $n$  matrix  $A$  is the factorization that matrix into the product  $A = Q^*R$ , where  $R$  is an  $m$ -by- $n$  upper triangular matrix and  $Q$  is an  $m$ -by- $m$  unitary matrix.

## See Also

`fitrm` | `manova` | `qr` | `ranova` | `vander`

## Topics

“Model Specification for Repeated Measures Models” on page 9-79



# ansaribradley

Ansari-Bradley test

## Syntax

```
h = ansaribradley(x, y)
h = ansaribradley(x, y, Name, Value)
[h, p] = ansaribradley(____)
[h, p, stats] = ansaribradley(____)
```

## Description

`h = ansaribradley(x, y)` returns a test decision for the null hypothesis that the data in vectors `x` and `y` comes from the same distribution, using the Ansari-Bradley test on page 32-121. The alternative hypothesis is that the data in `x` and `y` comes from distributions with the same median and shape but different dispersions (e.g., variances). The result `h` is 1 if the test rejects the null hypothesis at the 5% significance level, or 0 otherwise.

`h = ansaribradley(x, y, Name, Value)` returns a test decision for the Ansari-Bradley test with additional options specified by one or more name-value pair arguments. For example, you can change the significance level, conduct a one-sided test, or use a normal approximation to calculate the value of the test statistic.

`[h, p] = ansaribradley(____)` also returns the  $p$ -value, `p`, of the test, using any of the input arguments in the previous syntaxes.

`[h, p, stats] = ansaribradley(____)` also returns the structure `stats` containing information about the test statistic.

## Examples

### Ansari-Bradley Test for Equal Variances

Load the sample data. Create data vectors of miles per gallon (MPG) measurements for the model years 1982 and 1976.

```
load carsmall
x = MPG(Model_Year==82);
y = MPG(Model_Year==76);
```

Test the null hypothesis that the miles per gallon measured in cars from 1982 and 1976 have equal variances.

```
[h,p,stats] = ansaribradley(x,y)
```

```
h =
```

```
    0
```

```
p =
```

```
    0.8426
```

```
stats =
```

```
    struct with fields:
```

```
        W: 526.9000
```

```
        Wstar: 0.1986
```

The returned value of `h = 0` indicates that `ansaribradley` does not reject the null hypothesis at the default 5% significance level.

### Ansari-Bradley One-Sided Hypothesis Test

Load the sample data. Create data vectors of miles per gallon (MPG) measurements for the model years 1982 and 1976.

```
load carsmall
x = MPG(Model_Year==82);
y = MPG(Model_Year==76);
```

Test the null hypothesis that the miles per gallon measured in cars from 1982 and 1976 have equal variances, against the alternative hypothesis that the variance of cars from 1982 is greater than that of cars from 1976.

```
[h,p,stats] = ansaribradley(x,y,'Tail','right')
```

```
h =
```

```
    0
```

```
p =
```

```
    0.5787
```

```
stats =
```

```
    struct with fields:
```

```
        W: 526.9000
```

```
       Wstar: 0.1986
```

The returned value of  $h = 0$  indicates that `ansaribradley` does not reject the null hypothesis that the variance in miles per gallon is the same for the two model years, when the alternative is that the variance of cars from 1982 is greater than that of cars from 1976.

## Input Arguments

### **x** — Sample data

vector | matrix | multidimensional array

Sample data, specified as a vector, matrix, or multidimensional array.

- If  $x$  and  $y$  are specified as vectors, they do not need to be the same length.

- If `x` and `y` are specified as matrices, they must have the same number of columns. `ansaribradley` performs separate tests along each column and returns a vector of results.
- If `x` and `y` are specified as multidimensional arrays on page 32-121, `ansaribradley` works along the first nonsingleton dimension on page 32-121. `x` and `y` must have the same size along all remaining dimensions.

Data Types: `single` | `double`

### **y** — Sample data

vector | matrix | multidimensional array

Sample data, specified as a vector, matrix, or multidimensional array.

- If `x` and `y` are specified as vectors, they do not need to be the same length.
- If `x` and `y` are specified as matrices, they must have the same number of columns. `ansaribradley` performs separate tests along each column and returns a vector of results.
- If `x` and `y` are specified as multidimensional arrays on page 32-121, `ansaribradley` works along the first nonsingleton dimension on page 32-121. `x` and `y` must have the same size along all remaining dimensions.

Data Types: `single` | `double`

## **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

Example: `'Tail', 'right', 'Alpha', 0.01` specifies a right-tailed hypothesis test at the 1% significance level.

### **Alpha** — Significance level

0.05 (default) | scalar value in the range (0,1)

Significance level of the hypothesis test, specified as the comma-separated pair consisting of `'Alpha'` and a scalar value in the range (0,1).

Example: 'Alpha',0.01

Data Types: single | double

### **Dim — Dimension**

first nonsingleton dimension (default) | positive integer value

Dimension of the input matrix along which to test the means, specified as the comma-separated pair consisting of 'Dim' and a positive integer value. For example, specifying 'Dim',1 tests the column means, while 'Dim',2 tests the row means.

Example: 'Dim',2

Data Types: single | double

### **Tail — Type of alternative hypothesis**

'both' (default) | 'left' | 'right'

Type of alternative hypothesis to evaluate, specified as the comma-separated pair consisting of 'Tail' and one of the following.

'both'	Test the alternative hypothesis that the dispersion parameters of $x$ and $y$ are not equal.
'right'	Test the alternative hypothesis that the dispersion parameter of $x$ is greater than that of $y$ .
'left'	Test the alternative hypothesis that the dispersion parameter of $x$ is less than that of $y$ .

Example: 'Tail','right'

### **Method — Computation method**

'exact' | 'approximate'

Computation method for the test statistic, specified as the comma-separated pair consisting of 'Method' and one of the following.

'exact'	Compute $p$ using an exact calculation of the distribution of the test statistic $w$ . This is the default if $n$ , the total number of rows in $x$ and $y$ , is 25 or less. Note that $n$ is computed before any NaN values (representing missing data) are removed.
---------	---

'approximate'	Compute $p$ using a normal approximation for the statistic $w^*$ . This is the default if $n$ , the total number of rows in $x$ and $y$ , is greater than 25.
---------------	---

Example: 'Method', 'exact'

## Output Arguments

### **h** — Hypothesis test result

1 | 0

Hypothesis test result, returned as a logical value.

- If  $h = 1$ , this indicates the rejection of the null hypothesis at the Alpha significance level.
- If  $h = 0$ , this indicates a failure to reject the null hypothesis at the Alpha significance level.

### **p** — *p*-value

scalar value in the range [0,1]

*p*-value of the test, returned as a scalar value in the range [0,1].  $p$  is the probability of observing a test statistic as extreme as, or more extreme than, the observed value under the null hypothesis. Small values of  $p$  cast doubt on the validity of the null hypothesis.

### **stats** — Test statistics

structure

Test statistics for the Ansari-Bradley test, returned as a structure containing:

- $w$  — Value of the test statistic, which is the sum of the Ansari-Bradley ranks for the  $x$  sample.
- $w_{star}$  — Approximate normal statistic  $w^*$ .

## Definitions

### Ansari-Bradley Test

The Ansari-Bradley test is a nonparametric alternative to the two-sample  $F$ -test of equal variances. It does not require the assumption that  $x$  and  $y$  come from normal distributions. The dispersion of a distribution is generally measured by its variance or standard deviation, but the Ansari-Bradley test can be used with samples from distributions that do not have finite variances.

This test requires that the samples have equal medians. Under that assumption, and if the distributions of the samples are continuous and identical, the test is independent of the distributions. If the samples do not have the same medians, the results can be misleading. In that case, Ansari and Bradley recommend subtracting the median, but then the distribution of the resulting test under the null hypothesis is no longer independent of the common distribution of  $x$  and  $y$ . If you want to perform the tests with medians subtracted, you should subtract the medians from  $x$  and  $y$  before calling `ansaribradley`.

### Multidimensional Array

A multidimensional array has more than two dimensions. For example, if  $x$  is a 1-by-3-by-4 array, then  $x$  is a three-dimensional array.

### First Nonsingleton Dimension

The first nonsingleton dimension is the first dimension of an array whose size is not equal to 1. For example, if  $x$  is a 1-by-2-by-3-by-4 array, then the second dimension is the first nonsingleton dimension of  $x$ .

### See Also

`ttest2` | `vartest2` | `vartestn`

Introduced before R2006a

## aoctool

Interactive analysis of covariance

### Syntax

```
aoctool(x,y,group)
aoctool(x,y,group,alpha)
aoctool(x,y,group,alpha,xname,yname,gname)
aoctool(x,y,group,alpha,xname,yname,gname,displayopt)
aoctool(x,y,group,alpha,xname,yname,gname,displayopt,model)
h = aoctool(...)
[h,atab,ctab] = aoctool(...)
[h,atab,ctab,stats] = aoctool(...)
```

### Description

`aoctool(x,y,group)` fits a separate line to the column vectors, `x` and `y`, for each group defined by the values in the array `group`. `group` may be a categorical variable, vector, character array, or cell array of character vectors. These types of models are known as one-way analysis of covariance (ANOCOVA) models. The output consists of three figures:

- An interactive graph of the data and prediction curves
- An ANOVA table
- A table of parameter estimates

You can use the figures to change models and to test different parts of the model. More information about interactive use of the `aoctool` function appears in “Analysis of Covariance Tool” on page 9-60.

`aoctool(x,y,group,alpha)` determines the confidence levels of the prediction intervals. The confidence level is  $100(1-\alpha)\%$ . The default value of `alpha` is 0.05.

`aoctool(x,y,group,alpha,xname,yname,gname)` specifies the name to use for the `x`, `y`, and `g` variables in the graph and tables. If you enter simple variable names for the `x`, `y`, and `g` arguments, the `aoctool` function uses those names. If you enter an expression



for one of these arguments, you can specify a name to use in place of that expression by supplying these arguments. For example, if you enter `m(:,2)` as the `x` argument, you might choose to enter `'Col 2'` as the `xname` argument.

`aoctool(x,y,group,alpha,xname,yname,gname,displayopt)` enables the graph and table displays when `displayopt` is `'on'` (default) and suppresses those displays when `displayopt` is `'off'`.

`aoctool(x,y,group,alpha,xname,yname,gname,displayopt,model)` specifies the initial model to fit. The value of `model` can be any of the following:

- `'same mean'` — Fit a single mean, ignoring grouping
- `'separate means'` — Fit a separate mean to each group
- `'same line'` — Fit a single line, ignoring grouping
- `'parallel lines'` — Fit a separate line to each group, but constrain the lines to be parallel
- `'separate lines'` — Fit a separate line to each group, with no constraints

`h = aoctool(...)` returns a vector of handles to the line objects in the plot.

`[h,atab,ctab] = aoctool(...)` returns cell arrays containing the entries in ANOVA table (`atab`) and the table of coefficient estimates (`ctab`). (You can copy a text version of either table to the clipboard by using the Copy Text item on the **Edit** menu.)

`[h,atab,ctab,stats] = aoctool(...)` returns a `stats` structure that you can use to perform a follow-up multiple comparison test. The ANOVA table output includes tests of the hypotheses that the slopes or intercepts are all the same, against a general alternative that they are not all the same. Sometimes it is preferable to perform a test to determine which pairs of values are significantly different, and which are not. You can use the `multcompare` function to perform such tests by supplying the `stats` structure as input. You can test either the slopes, the intercepts, or population marginal means (the heights of the curves at the mean `x` value).

## Examples

This example illustrates how to fit different models non-interactively. After loading the smaller car data set and fitting a separate-slopes model, you can examine the coefficient estimates.

```
load carsmall
[h,a,c,s] = aoctool(Weight,MPG,Model_Year,0.05,...
                  ' ',' ',' ','off','separate lines');
c(:,1:2)
ans =
  'Term'      'Estimate'
  'Intercept' [45.97983716833132]
  ' 70'       [-8.58050531454973]
  ' 76'       [-3.89017396094922]
  ' 82'       [12.47067927549897]
  'Slope'     [-0.00780212907455]
  ' 70'       [ 0.00195840368824]
  ' 76'       [ 0.00113831038418]
  ' 82'       [-0.00309671407243]
```

Roughly speaking, the lines relating MPG to Weight have an intercept close to 45.98 and a slope close to -0.0078. Each group's coefficients are offset from these values somewhat. For instance, the intercept for the cars made in 1970 is  $45.98 - 8.58 = 37.40$ .

Next, try a fit using parallel lines. (The ANOVA table shows that the parallel-lines fit is significantly worse than the separate-lines fit.)

```
[h,a,c,s] = aoctool(Weight,MPG,Model_Year,0.05,...
                  ' ',' ',' ','off','parallel lines');
c(:,1:2)
ans =
  'Term'      'Estimate'
  'Intercept' [43.38984085130596]
  ' 70'       [-3.27948192983761]
  ' 76'       [-1.35036234809006]
  ' 82'       [ 4.62984427792768]
  'Slope'     [-0.00664751826198]
```

Again, there are different intercepts for each group, but this time the slopes are constrained to be the same.

## See Also

`anova1` | `multcompare` | `polytool`

**Introduced before R2006a**

## append

**Class:** `TreeBagger`

Append new trees to ensemble

## Syntax

```
B = append(B, other)
```

## Description

`B = append(B, other)` appends the trees from the `other` ensemble to those in `B`. This method checks for consistency of the `X` and `Y` properties of the two ensembles, as well as consistency of their compact objects and out-of-bag indices, before appending the trees. The output ensemble `B` takes training parameters such as `FBoot`, `Prior`, `Cost`, and `other` from the `B` input. There is no attempt to check if these training parameters are consistent between the two objects.

## See Also

`CompactTreeBagger.combine`

## BandWidth property

**Class:** ProbDistKernel

Read-only value specifying bandwidth of kernel smoothing function for ProbDistKernel object

### Description

BandWidth is a read-only property of the ProbDistKernel class. BandWidth is a value specifying the width of the kernel smoothing function used to compute a nonparametric estimate of the probability distribution when creating a ProbDistKernel object.

### Values

For a distribution specified to cover only the positive numbers or only a finite interval, the data are transformed before the kernel density is applied, and the bandwidth is on the scale of the transformed data.

Use this information to view and compare the width of the kernel smoothing function used to create distributions.

### See Also

`ksdensity`

## barttest

Bartlett's test

### Syntax

```
ndim = barttest(x,alpha)
[ndim,prob,chisquare] = barttest(x,alpha)
```

### Description

`ndim = barttest(x,alpha)` returns the number of dimensions necessary to explain the nonrandom variation in the data matrix `x` at the `alpha` significance level.

`[ndim,prob,chisquare] = barttest(x,alpha)` also returns the significance values for the hypothesis tests `prob`, and the  $\chi^2$  values associated with the tests `chisquare`.

### Examples

#### Determine Dimensions Needed to Explain Nonrandom Data Variation

Generate a 20-by-6 matrix of random numbers from a multivariate normal distribution with mean `mu = [0 0]` and covariance `sigma = [1 0.99; 0.99 1]`.

```
rng default % for reproducibility
mu = [0 0];
sigma = [1 0.99; 0.99 1];
X = mvnrnd(mu,sigma,20); % columns 1 and 2
X(:,3:4) = mvnrnd(mu,sigma,20); % columns 3 and 4
X(:,5:6) = mvnrnd(mu,sigma,20); % columns 5 and 6
```

Determine the number of dimensions necessary to explain the nonrandom variation in data matrix `X`. Report the significance values for the hypothesis tests.

```
[ndim, prob] = barttest(X,0.05)
```

```
ndim =  
    3  
  
prob =  
    0.0000  
    0.0000  
    0.0000  
    0.5148  
    0.3370
```

The returned value of `ndim` indicates that three dimensions are necessary to explain the nonrandom variation in `X`.

## Input Arguments

### **x** — Input data

matrix of scalar values

Input data, specified as a matrix of scalar values.

Data Types: `single` | `double`

### **alpha** — Significance level

0.05 (default) | scalar value in the range (0,1)

Significance level of the hypothesis test, specified as a scalar value in the range (0,1).

Example: 0.1

Data Types: `single` | `double`

## Output Arguments

### **ndim** — Number of dimensions

positive integer value

Number of dimensions, returned as a positive integer value. The dimension is determined by a series of hypothesis tests. The test for `ndim = 1` tests the hypothesis that the variances of the data values along each principal component are equal, the test for `ndim = 2` tests the hypothesis that the variances along the second through last components are equal, and so on. The null hypothesis is that the number of dimensions is equal to the number of the largest unequal eigenvalues of the covariance matrix of `x`.

**prob — Significance value**

vector of scalar values in the range  $(0, 1)$

Significance value for the hypothesis tests, returned as a vector of scalar values in the range  $(0, 1)$ . Each element in `prob` corresponds to an element of `chisquare`.

**chisquare — Test statistics**

vector of scalar values

Test statistics for each dimension's hypothesis test, returned as a vector of scalar values.

## See Also

Introduced before R2006a



# BayesianOptimization

Bayesian optimization results

## Description

A `BayesianOptimization` object contains the results of a Bayesian optimization. It is the output of `bayesopt` or a fit function that accepts the `OptimizeHyperparameters` name-value pair such as `fitcdiscr`. In addition, a `BayesianOptimization` object contains data for each iteration of `bayesopt` that can be accessed by a plot function or an output function.

## Creation

Create a `BayesianOptimization` object using the `bayesopt` function or a fit function with the `OptimizeHyperparameters` name-value pair.

## Properties

### Problem Definition Properties

**ObjectiveFcn** — **ObjectiveFcn** argument that `bayesopt` used  
function handle

ObjectiveFcn argument that `bayesopt` used, returned as a function handle.

- If you called `bayesopt` directly, `ObjectiveFcn` is the `bayesopt` objective function argument.
- If you called a fit function with the `OptimizeHyperparameters` name-value pair, `ObjectiveFcn` is the logarithm of one plus the cross-validation loss.

Data Types: `function_handle`

**VariableDescriptions** — **VariableDescriptions** argument that `bayesopt` used  
vector of `optimizableVariable` objects

`VariableDescriptions` argument that `bayesopt` used, returned as a vector of `optimizableVariable` objects.

- If you called `bayesopt` directly, `VariableDescriptions` is the `bayesopt` variable description argument.
- If you called a fit function with the `OptimizeHyperparameters` name-value pair, `VariableDescriptions` is the vector of hyperparameters.

### Options — Options that `bayesopt` used

structure

Options that `bayesopt` used, returned as a structure.

- If you called `bayesopt` directly, `Options` is the options used in `bayesopt`, which are the name-value pairs See `bayesopt` “Input Arguments” on page 32-159.
- If you called a fit function with the `OptimizeHyperparameters` name-value pair, `Options` are the default `bayesopt` options, modified by the `HyperparameterOptimizationOptions` name-value pair.

`Options` is a read-only structure containing the following fields.

Option Name	Meaning
<code>AcquisitionFunctionName</code>	Acquisition function name. See “Acquisition Function Types” on page 10-4.
<code>IsObjectiveDeterministic</code>	true means the objective function is deterministic, false otherwise.
<code>ExplorationRatio</code>	Used only when <code>AcquisitionFunctionName</code> is 'expected-improvement-plus' or 'expected-improvement-per-second-plus'. See “Plus” on page 10-6.
<code>MaxObjectiveEvaluations</code>	Objective function evaluation limit.
<code>MaxTime</code>	Time limit.
<code>XConstraintFcn</code>	Deterministic constraints on variables. See “Deterministic Constraints — <code>XConstraintFcn</code> ” on page 10-44.

Option Name	Meaning
ConditionalVariableFcn	Conditional variable constraints. See “Conditional Constraints — ConditionalVariableFcn” on page 10-45.
NumCoupledConstraints	Number of coupled constraints. See “Coupled Constraints” on page 10-47.
CoupledConstraintTolerances	Coupled constraint tolerances. See “Coupled Constraints” on page 10-47.
AreCoupledConstraintsDeterministic	Logical vector specifying whether each coupled constraint is deterministic.
Verbose	Command-line display level.
OutputFcn	Function called after each iteration. See “Bayesian Optimization Output Functions” on page 10-23.
SaveVariableName	Variable name for the @assignInBase output function.
SaveFileName	File name for the @saveToFile output function.
PlotFcn	Plot function called after each iteration. See “Bayesian Optimization Plot Functions” on page 10-14
InitialX	Points where bayesopt evaluated the objective function.
InitialObjective	Objective function values at InitialX.
InitialConstraintViolations	Coupled constraint function values at InitialX.
InitialErrorValues	Error values at InitialX.
InitialObjectiveEvaluationTimes	Objective function evaluation times at InitialX.
InitialIterationTimes	Time for each iteration, including objective function evaluation and other computations.

Data Types: struct

**Solution Properties****MinObjective** — Minimum observed value of objective function

real scalar

Minimum observed value of objective function, returned as a real scalar. When there are coupled constraints or evaluation errors, this value is the minimum over all observed points that are feasible according to the final constraint and Error models.

Data Types: double

**XAtMinObjective** — Observed point with minimum objective function value

1-by-D table

Observed point with minimum objective function value, returned as a 1-by-D table, where D is the number of variables.

Data Types: table

**MinEstimatedObjective** — Minimum estimated value of objective function

real scalar

Minimum estimated value of objective function, returned as a real scalar. `MinEstimatedObjective` uses the final objective model.

`MinEstimatedObjective` is the same as the `CriterionValue` result of `bestPoint` with default criterion.

Data Types: double

**XAtMinEstimatedObjective** — Point with minimum estimated objective function value

1-by-D table

Point with minimum estimated objective function value, returned as a 1-by-D table, where D is the number of variables. `XAtMinEstimatedObjective` uses the final objective model.

Data Types: table

**NumObjectiveEvaluations** — Number of objective function evaluations

positive integer

Number of objective function evaluations, returned as a positive integer. This includes the initial evaluations to form a posterior model as well as evaluation during the optimization iterations.

Data Types: `double`

**TotalElapsedTime** — Total elapsed time of optimization in seconds

positive scalar

Total elapsed time of optimization in seconds, returned as a positive scalar.

Data Types: `double`

**NextPoint** — Next point to evaluate if optimization continues

1-by-D table

Next point to evaluate if optimization continues, returned as a 1-by-D table, where D is the number of variables.

Data Types: `table`

**Trace Properties**

**xTrace** — Points where the objective function was evaluated

T-by-D table

Points where the objective function was evaluated, returned as a T-by-D table, where T is the number of evaluation points and D is the number of variables.

Data Types: `table`

**ObjectiveTrace** — Objective function values

column vector of length T

Objective function values, returned as a column vector of length T, where T is the number of evaluation points. `ObjectiveTrace` contains the history of objective function evaluations.

Data Types: `double`

**ObjectiveEvaluationTimeTrace** — Objective function evaluation times

column vector of length T

Objective function evaluation times, returned as a column vector of length T, where T is the number of evaluation points. `ObjectiveEvaluationTimeTrace` includes the time

in evaluating coupled constraints, because the objective function computes these constraints.

Data Types: `double`

**IterationTimeTrace** — Iteration times

column vector of length `T`

Iteration times, returned as a column vector of length `T`, where `T` is the number of evaluation points. `IterationTimeTrace` includes both objective function evaluation time and other overhead.

Data Types: `double`

**ConstraintsTrace** — Coupled constraint values

`T`-by-`K` array

Coupled constraint values, returned as a `T`-by-`K` array, where `T` is the number of evaluation points and `K` is the number of coupled constraints.

Data Types: `double`

**ErrorTrace** — Error indications

column vector of length `T` of `-1` or `1` entries

Error indications, returned as a column vector of length `T` of `-1` or `1` entries, where `T` is the number of evaluation points. Each `1` entry indicates that the objective function errored or returned `NaN` on the corresponding point in `XTrace`. Each `-1` entry indicates that the objective function value was computed.

Data Types: `double`

**FeasibilityTrace** — Feasibility indications

logical column vector of length `T`

Feasibility indications, returned as a logical column vector of length `T`, where `T` is the number of evaluation points. Each `1` entry indicates that the final constraint model predicts feasibility at the corresponding point in `XTrace`.

Data Types: `logical`

**FeasibilityProbabilityTrace** — Probability that evaluation point is feasible

column vector of length `T`

Probability that evaluation point is feasible, returned as a column vector of length  $T$ , where  $T$  is the number of evaluation points. The probabilities come from the final constraint model, including the error constraint model, on the corresponding points in `XTrace`.

Data Types: `double`

**IndexOfMinimumTrace** — Which evaluation gave minimum feasible objective

column vector of integer indices of length  $T$

Which evaluation gave minimum feasible objective, returned as a column vector of integer indices of length  $T$ , where  $T$  is the number of evaluation points. Feasibility is determined with respect to the constraint models that existed at each iteration, including the error constraint model.

Data Types: `double`

**ObjectiveMinimumTrace** — Minimum observed objective

column vector of length  $T$

Minimum observed objective, returned as a column vector of integer indices of length  $T$ , where  $T$  is the number of evaluation points.

Data Types: `double`

**EstimatedObjectiveMinimumTrace** — Minimum estimated objective

column vector of length  $T$

Minimum estimated objective, returned as a column vector of integer indices of length  $T$ , where  $T$  is the number of evaluation points. The estimated objective at each iteration is determined with respect to the objective model that existed at that iteration.

Data Types: `double`

**UserDataTrace** — Auxiliary data from the objective function

cell array of length  $T$

Auxiliary data from the objective function, returned as a cell array of length  $T$ , where  $T$  is the number of evaluation points. Each entry in the cell array is the `UserData` returned in the third output of the objective function.

Data Types: `cell`

## Object Functions

<code>bestPoint</code>	Best point in a Bayesian optimization according to a criterion
<code>plot</code>	Plot Bayesian optimization results
<code>predictConstraints</code>	Predict coupled constraint violations at a set of points
<code>predictError</code>	Predict error value at a set of points
<code>predictObjective</code>	Predict objective function at a set of points
<code>predictObjectiveEvaluationTime</code>	Predict objective function run times at a set of points
<code>resume</code>	Resume a Bayesian optimization

## Examples

### Create a `BayesianOptimization` Object Using `bayesopt`

This example shows how to create a `BayesianOptimization` object by using `bayesopt` to minimize cross-validation loss.

Optimize hyperparameters of a KNN classifier for the `ionosphere` data, that is, find KNN hyperparameters that minimize the cross-validation loss. Have `bayesopt` minimize over the following hyperparameters:

- Nearest-neighborhood sizes from 1 to 30
- Distance functions `'chebychev'`, `'euclidean'`, and `'minkowski'`.

For reproducibility, set the random seed, set the partition, and set the `AcquisitionFunctionName` option to `'expected-improvement-plus'`. Set options give no iterative display.

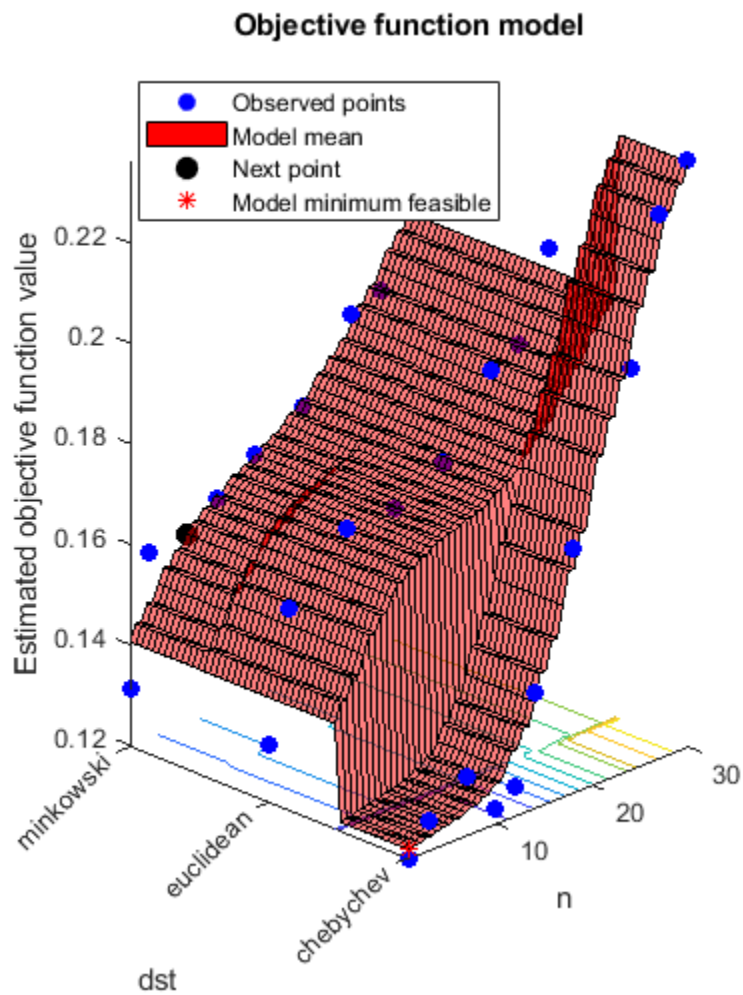
```
load ionosphere
rng default
num = optimizableVariable('n', [1, 30], 'Type', 'integer');
dst = optimizableVariable('dst', {'chebychev', 'euclidean', 'minkowski'}, 'Type', 'categorical');
c = cvpartition(351, 'Kfold', 5);
fun = @(x) kfoldLoss(fitcknn(X, Y, 'CVPartition', c, 'NumNeighbors', x.n, ...
    'Distance', char(x.dst), 'NSMethod', 'exhaustive'));
results = bayesopt(fun, [num, dst], 'Verbose', 0, ...
    'AcquisitionFunctionName', 'expected-improvement-plus')

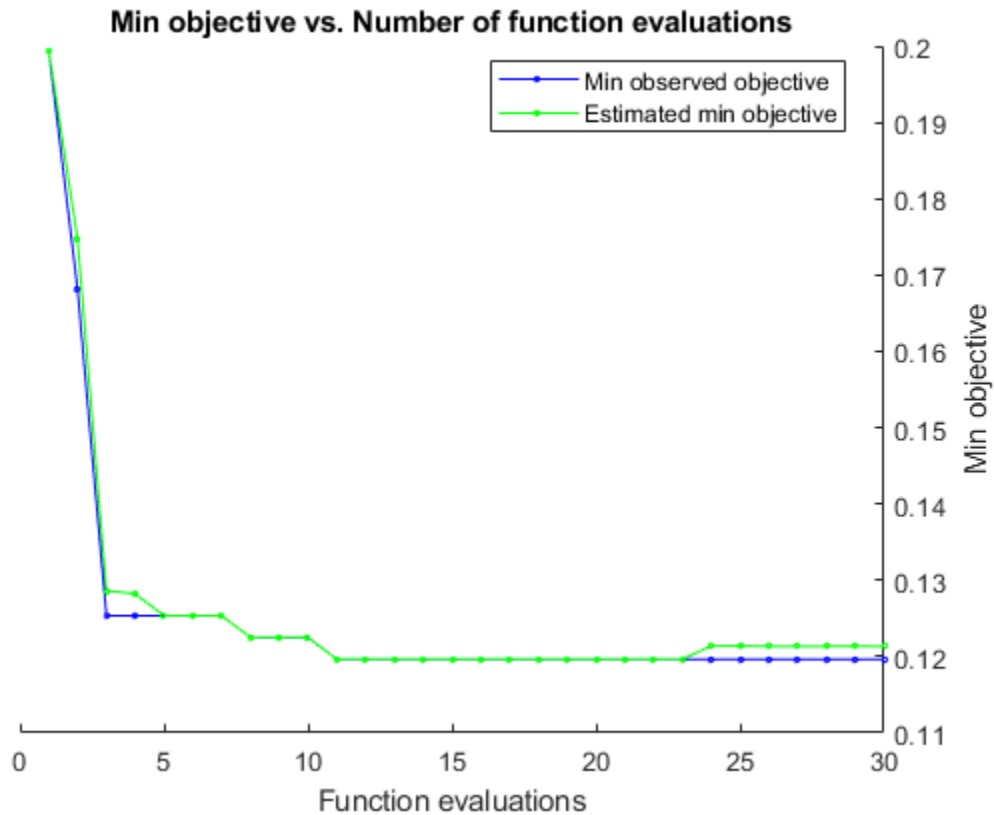
results =
```



BayesianOptimization with properties:

```
ObjectiveFcn: [function_handle]
VariableDescriptions: [1x2 optimizableVariable]
Options: [1x1 struct]
MinObjective: 0.1197
XAtMinObjective: [1x2 table]
MinEstimatedObjective: 0.1213
XAtMinEstimatedObjective: [1x2 table]
NumObjectiveEvaluations: 30
TotalElapsedTime: 60.8615
NextPoint: [1x2 table]
XTrace: [30x2 table]
ObjectiveTrace: [30x1 double]
ConstraintsTrace: []
UserDataTrace: {30x1 cell}
ObjectiveEvaluationTimeTrace: [30x1 double]
IterationTimeTrace: [30x1 double]
ErrorTrace: [30x1 double]
FeasibilityTrace: [30x1 logical]
FeasibilityProbabilityTrace: [30x1 double]
IndexOfMinimumTrace: [30x1 double]
ObjectiveMinimumTrace: [30x1 double]
EstimatedObjectiveMinimumTrace: [30x1 double]
```





### Create a BayesianOptimization Object Using a Fit Function

This example shows how to minimize the cross-validation loss in the `ionosphere` data using Bayesian optimization of an SVM classifier.

Load the data.

```
load ionosphere
```

Optimize the classification using the 'auto' parameters.

```
rng default % For reproducibility
Mdl = fitcsvm(X,Y,'OptimizeHyperparameters','auto')
```

Iter	Eval result	Objective	Objective runtime	BestSoFar (observed)	BestSoFar (estim.)	BoxConstrain- t
1	Best	0.21937	48.952	0.21937	0.21937	64.836
2	Accept	0.35897	0.42569	0.21937	0.22807	0.036335
3	Best	0.13105	20.133	0.13105	0.14149	0.0022147
4	Accept	0.35897	0.29583	0.13105	0.13108	5.1259
5	Best	0.12821	0.45085	0.12821	0.12823	0.0010294
6	Accept	0.12821	0.50179	0.12821	0.12673	0.0010607
7	Accept	0.1339	4.0301	0.12821	0.12601	0.025984
8	Best	0.12536	0.44954	0.12536	0.12651	0.0010012
9	Best	0.12536	0.40723	0.12536	0.12576	0.0010456
10	Best	0.12251	0.3836	0.12251	0.12478	0.0010611
11	Accept	0.12536	0.3982	0.12251	0.12481	0.0010061
12	Best	0.12251	0.40891	0.12251	0.12426	0.0011285
13	Accept	0.35897	0.33095	0.12251	0.12436	2.2698
14	Accept	0.12821	0.42859	0.12251	0.12486	0.0016594
15	Accept	0.1339	15.877	0.12251	0.12473	961.97
16	Accept	0.21652	48.484	0.12251	0.12444	988.71
17	Accept	0.35897	0.2432	0.12251	0.1244	0.09133
18	Accept	0.35897	0.22666	0.12251	0.1245	0.0011299
19	Accept	0.35897	0.24528	0.12251	0.12452	0.0010007
20	Accept	0.35897	0.26499	0.12251	0.12458	387.65
Iter	Eval result	Objective	Objective runtime	BestSoFar (observed)	BestSoFar (estim.)	BoxConstrain- t
21	Accept	0.35897	0.27067	0.12251	0.12455	0.0010001
22	Accept	0.35897	0.25497	0.12251	0.12459	0.0010662
23	Accept	0.1339	0.20336	0.12251	0.12388	994.18
24	Accept	0.12821	0.23057	0.12251	0.12382	991.83
25	Best	0.12251	0.24136	0.12251	0.12384	998.07
26	Accept	0.1339	0.68605	0.12251	0.12385	991.65
27	Accept	0.12536	0.41208	0.12251	0.12383	156.92
28	Accept	0.12821	0.37692	0.12251	0.12384	83.623
29	Accept	0.13105	0.53405	0.12251	0.12383	227.76
30	Accept	0.12536	0.46054	0.12251	0.12383	17.867

Optimization completed.

MaxObjectiveEvaluations of 30 reached.

Total function evaluations: 30

Total elapsed time: 226.0289 seconds.

Total objective function evaluation time: 146.6083

Best observed feasible point:

BoxConstraint	KernelScale
998.07	50.892

Observed objective function value = 0.12251  
 Estimated objective function value = 0.12383  
 Function evaluation time = 0.24136

Best estimated feasible point (according to models):

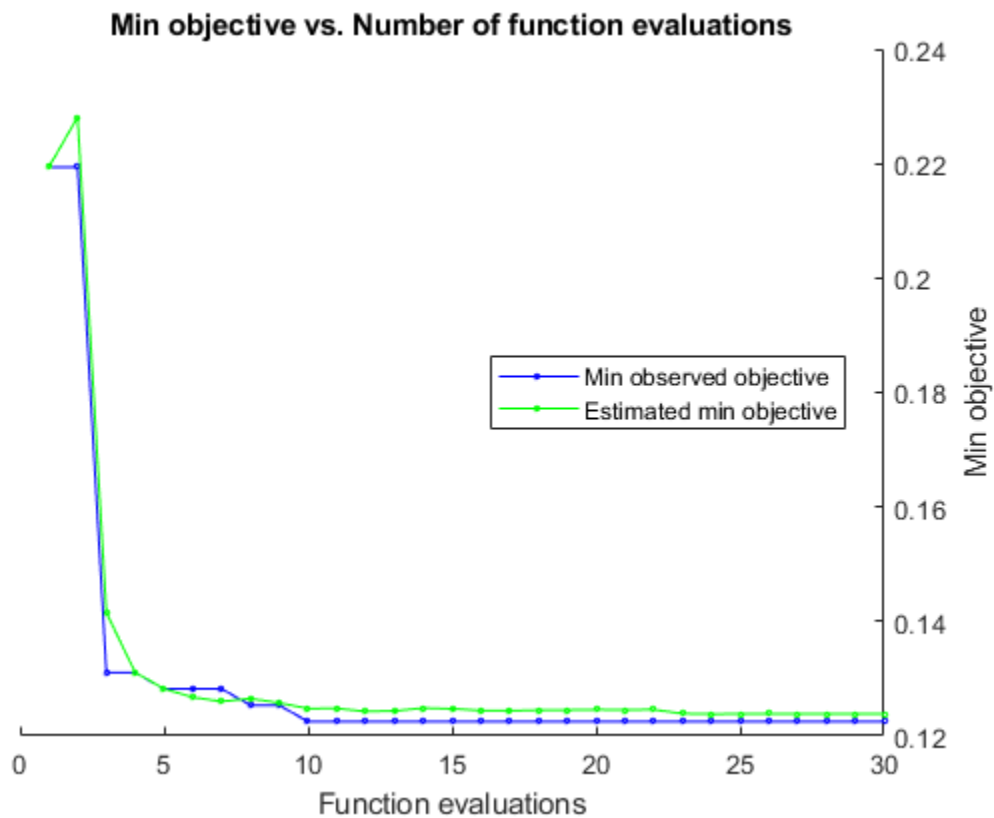
BoxConstraint	KernelScale
0.0010061	0.019341

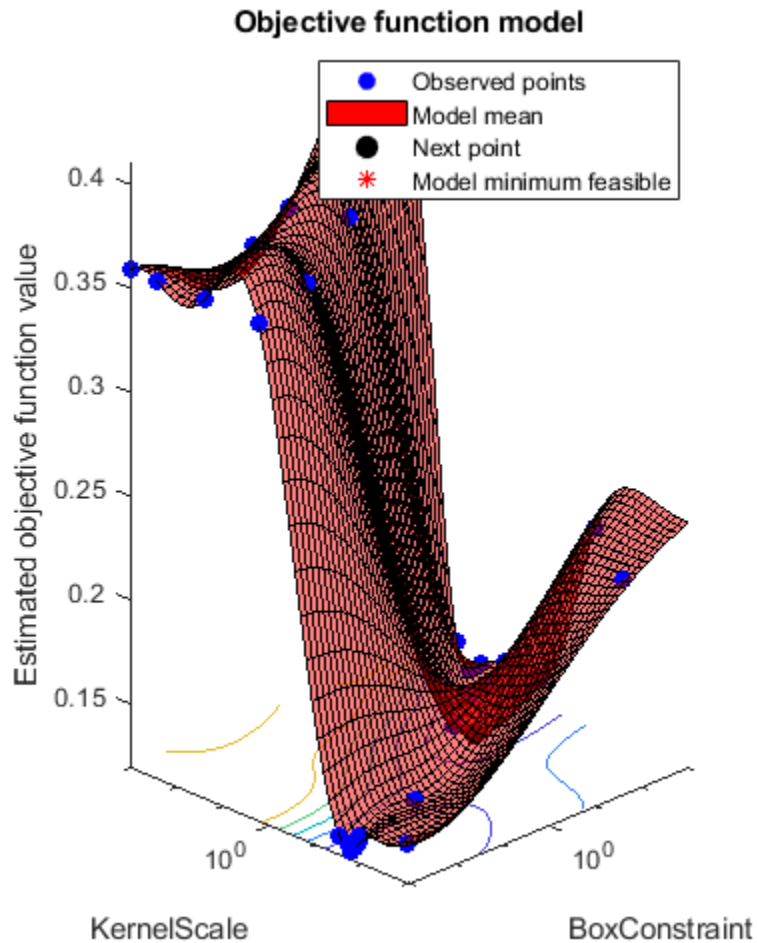
Estimated objective function value = 0.12383  
 Estimated function evaluation time = 0.39463

Mdl =

```

ClassificationSVM
    ResponseName: 'Y'
    CategoricalPredictors: []
    ClassNames: {'b' 'g'}
    ScoreTransform: 'none'
    NumObservations: 351
    HyperparameterOptimizationResults: [1x1 BayesianOptimization]
        Alpha: [92x1 double]
        Bias: -5.4961
    KernelParameters: [1x1 struct]
        BoxConstraints: [351x1 double]
        ConvergenceInfo: [1x1 struct]
        IsSupportVector: [351x1 logical]
        Solver: 'SMO'
  
```





The fit achieved about 12% loss for the default 5-fold cross validation.

Examine the `BayesianOptimization` object that is returned in the `HyperparameterOptimizationResults` property of the returned model.

```
disp(Mdl.HyperparameterOptimizationResults)
```

BayesianOptimization with properties:

```
ObjectiveFcn: @createObjFcn/theObjFcn
VariableDescriptions: [5x1 optimizableVariable]
Options: [1x1 struct]
MinObjective: 0.1225
XAtMinObjective: [1x2 table]
MinEstimatedObjective: 0.1238
XAtMinEstimatedObjective: [1x2 table]
NumObjectiveEvaluations: 30
TotalElapsedTime: 226.0289
NextPoint: [1x2 table]
XTrace: [30x2 table]
ObjectiveTrace: [30x1 double]
ConstraintsTrace: []
UserDataTrace: {30x1 cell}
ObjectiveEvaluationTimeTrace: [30x1 double]
IterationTimeTrace: [30x1 double]
ErrorTrace: [30x1 double]
FeasibilityTrace: [30x1 logical]
FeasibilityProbabilityTrace: [30x1 double]
IndexOfMinimumTrace: [30x1 double]
ObjectiveMinimumTrace: [30x1 double]
EstimatedObjectiveMinimumTrace: [30x1 double]
```

## See Also

bayesopt | fitcdiscr | fitcecoc | fitcensemble | fitcknn | fitclinear |  
fitcnb | fitcsvm | fitctree | fitrensemble | fitrgp | fitrlinear | fitrsvm  
| fitrtree

## Topics

“Bayesian Optimization Workflow” on page 10-30

Introduced in R2016b



# bayesopt

Select optimal machine learning hyperparameters using Bayesian optimization

## Syntax

```
results = bayesopt(fun, vars)
results = bayesopt(fun, vars, Name, Value)
```

## Description

`results = bayesopt(fun, vars)` attempts to find values of `vars` that minimize `fun(vars)`.

`results = bayesopt(fun, vars, Name, Value)` modifies the optimization process according to the `Name, Value` arguments.

## Examples

### Create a `BayesianOptimization` Object Using `bayesopt`

This example shows how to create a `BayesianOptimization` object by using `bayesopt` to minimize cross-validation loss.

Optimize hyperparameters of a KNN classifier for the `ionosphere` data, that is, find KNN hyperparameters that minimize the cross-validation loss. Have `bayesopt` minimize over the following hyperparameters:

- Nearest-neighborhood sizes from 1 to 30
- Distance functions 'chebychev', 'euclidean', and 'minkowski'.

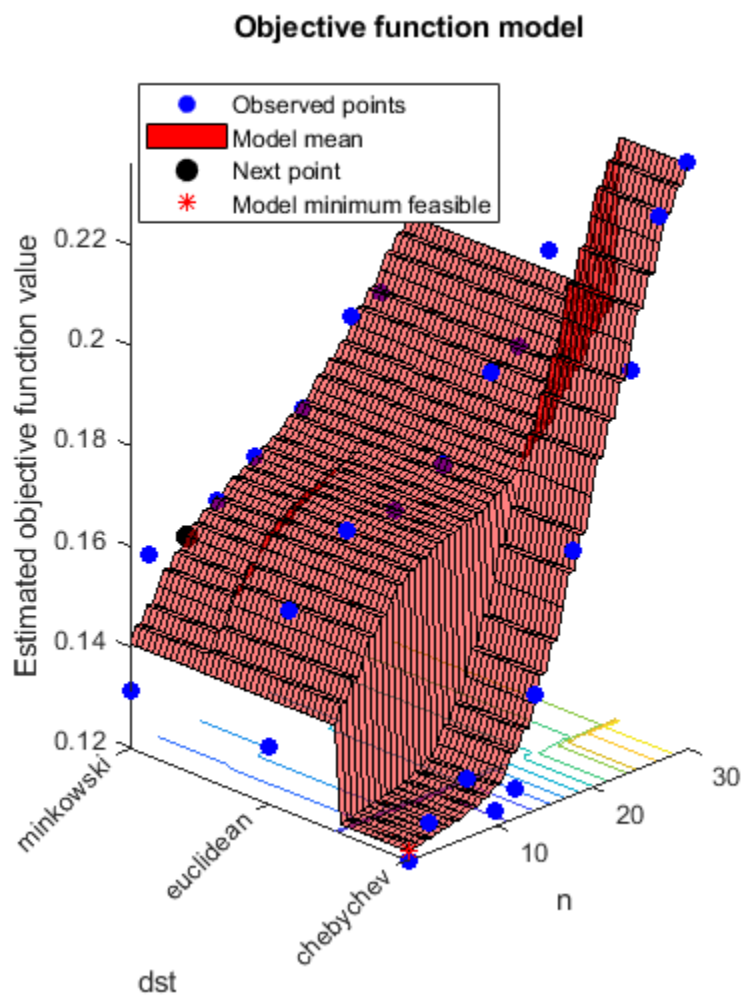
For reproducibility, set the random seed, set the partition, and set the `AcquisitionFunctionName` option to 'expected-improvement-plus'. Set options give no iterative display.

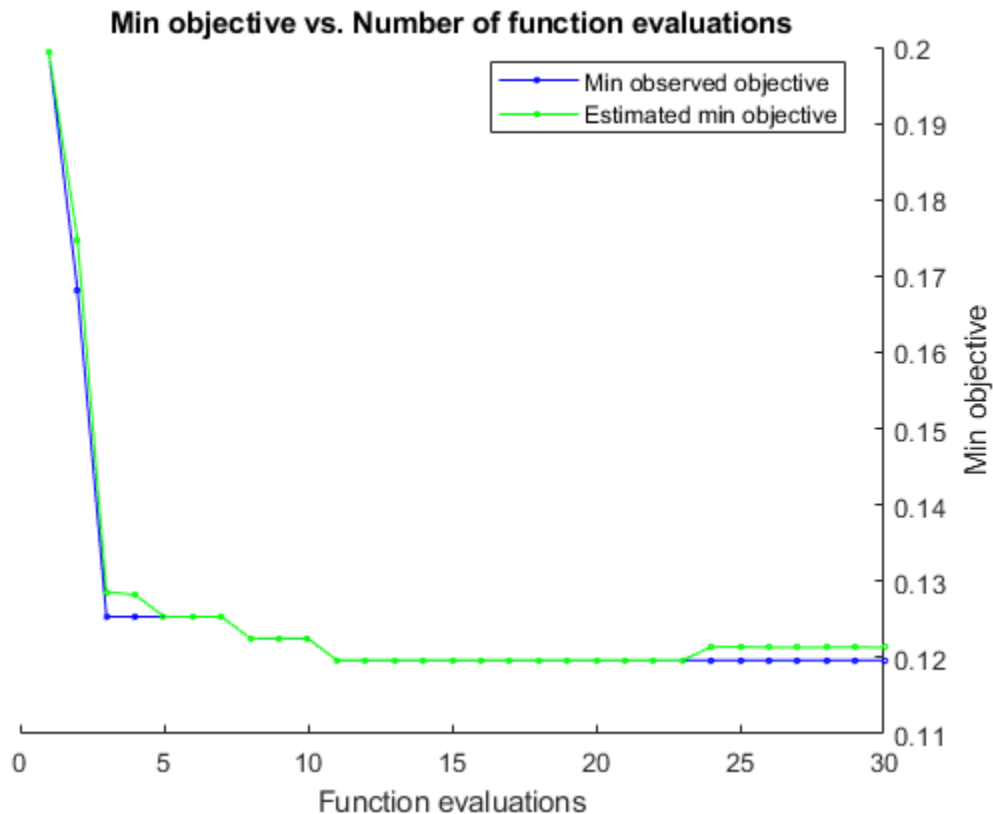
```
load ionosphere
rng default
num = optimizableVariable('n', [1, 30], 'Type', 'integer');
dst = optimizableVariable('dst', {'chebychev', 'euclidean', 'minkowski'}, 'Type', 'categorical');
c = cvpartition(351, 'Kfold', 5);
fun = @(x) kfoldLoss(fitcknn(X, Y, 'CVPARTITION', c, 'NumNeighbors', x.n, ...
    'Distance', char(x.dst), 'NSMethod', 'exhaustive'));
results = bayesopt(fun, [num, dst], 'Verbose', 0, ...
    'AcquisitionFunctionName', 'expected-improvement-plus')
```

```
results =
```

```
BayesianOptimization with properties:
```

```
ObjectiveFcn: [function_handle]
VariableDescriptions: [1x2 optimizableVariable]
Options: [1x1 struct]
MinObjective: 0.1197
XAtMinObjective: [1x2 table]
MinEstimatedObjective: 0.1213
XAtMinEstimatedObjective: [1x2 table]
NumObjectiveEvaluations: 30
TotalElapsedTime: 60.8615
NextPoint: [1x2 table]
XTrace: [30x2 table]
ObjectiveTrace: [30x1 double]
ConstraintsTrace: []
UserDataTrace: {30x1 cell}
ObjectiveEvaluationTimeTrace: [30x1 double]
IterationTimeTrace: [30x1 double]
ErrorTrace: [30x1 double]
FeasibilityTrace: [30x1 logical]
FeasibilityProbabilityTrace: [30x1 double]
IndexOfMinimumTrace: [30x1 double]
ObjectiveMinimumTrace: [30x1 double]
EstimatedObjectiveMinimumTrace: [30x1 double]
```





### Bayesian Optimization with Coupled Constraints

A coupled constraint is one that can be evaluated only by evaluating the objective function. In this case, the objective function is the cross-validated loss of an SVM model. The coupled constraint is that the number of support vectors is no more than 100. The model details are in “Optimize a Cross-Validated SVM Classifier Using Bayesian Optimization” on page 10-53.

Create the data for classification.

```
rng default
grpnpop = mvnrnd([1,0],eye(2),10);
```

```

redpop = mvnrnd([0,1],eye(2),10);
redpts = zeros(100,2);
grnpts = redpts;
for i = 1:100
    grnpts(i,:) = mvnrnd(grnpop(randi(10),:),eye(2)*0.02);
    redpts(i,:) = mvnrnd(redpop(randi(10),:),eye(2)*0.02);
end
cdata = [grnpts;redpts];
grp = ones(200,1);
grp(101:200) = -1;
c = cvpartition(200,'KFold',10);
sigma = optimizableVariable('sigma',[1e-5,1e5],'Transform','log');
box = optimizableVariable('box',[1e-5,1e5],'Transform','log');

```

The objective function is the cross-validation loss of the SVM model for partition *c*. The coupled constraint is the number of support vectors minus 100.5. This ensures that 100 support vectors give a negative constraint value, but 101 support vectors give a positive value. The model has 200 data points, so the coupled constraint values range from -99.5 (there is always at least one support vector) to 99.5. Positive values mean the constraint is not satisfied.

```

function [objective,constraint] = mysvmfun(x,cdata,grp,c)
SVMModel = fitcsvm(cdata,grp,'KernelFunction','rbf',...
    'BoxConstraint',x.box,...
    'KernelScale',x.sigma);
cvModel = crossval(SVMModel,'CVPartition',c);
objective = kfoldLoss(cvModel);
constraint = sum(SVMModel.IsSupportVector)-100.5;

fun = @(x)mysvmfun(x,cdata,grp,c);

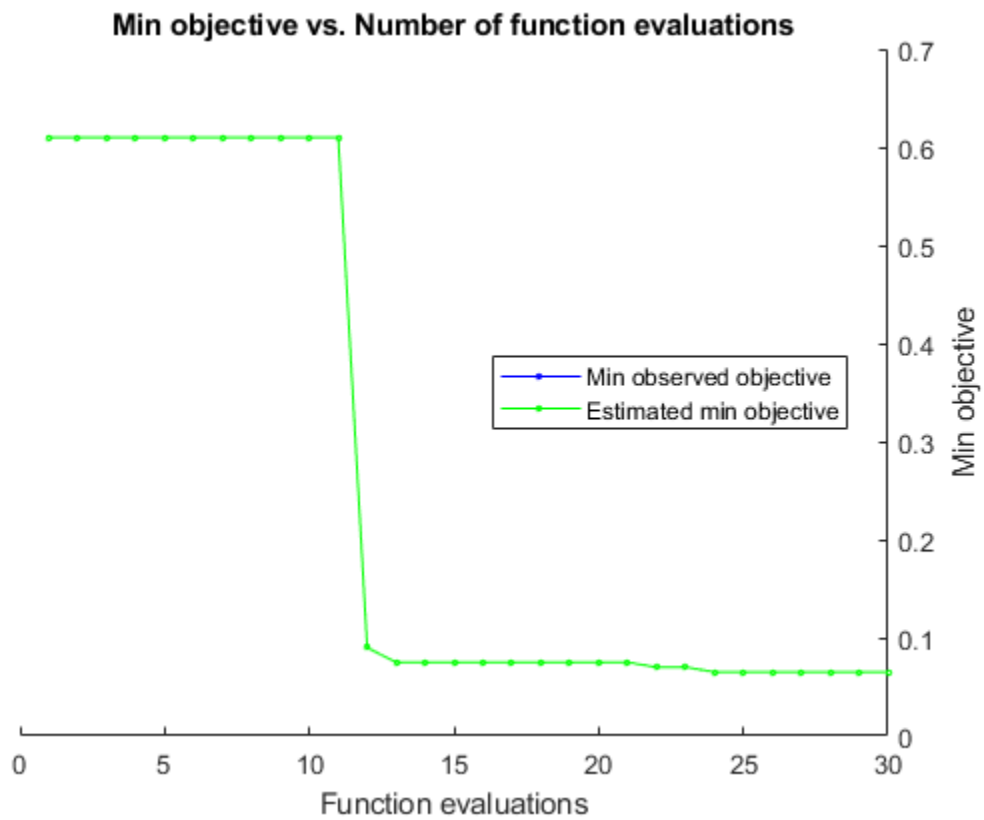
```

Set the `NumCoupledConstraints` to 1 so the optimizer knows that there is a coupled constraint. Set options to plot the constraint model.

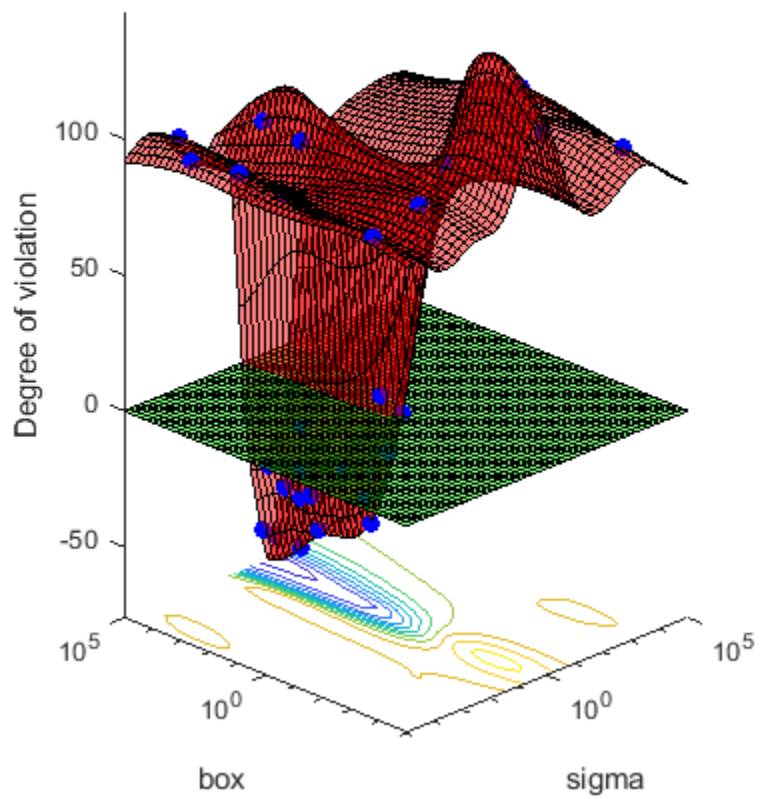
```

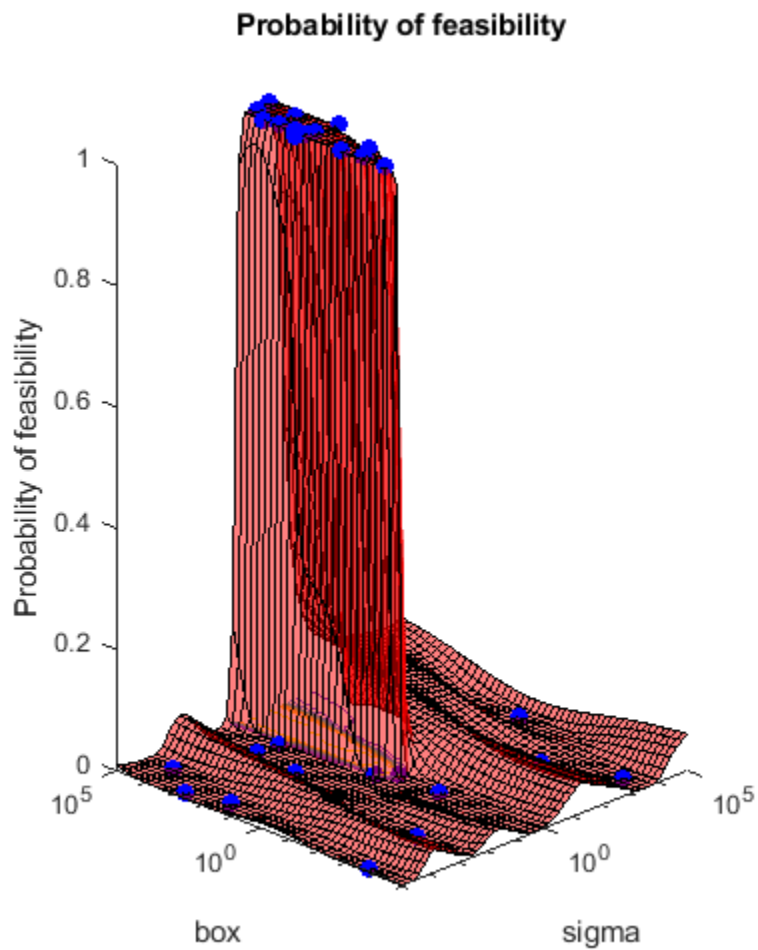
results = bayesopt(fun,[sigma,box],'IsObjectiveDeterministic',true,...
    'NumCoupledConstraints',1,'PlotFcn',...
    {@plotMinObjective,@plotConstraintModels},...
    'AcquisitionFunctionName','expected-improvement-plus','Verbose',0);

```



## Constraint 1 model





Most points lead to an infeasible number of support vectors.

### Parallel Bayesian Optimization

Speed a Bayesian optimization by using parallel objective function evaluation.



## Objective Function and Variables

The objective function is the cross-validation error rate for the ionosphere data, a binary classification problem. Use `fitcsvm` as the classifier, with `BoxConstraint` and `KernelScale` as the parameters to optimize.

```
load ionosphere
box = optimizableVariable('box', [1e-4, 1e3], 'Transform', 'log');
kern = optimizableVariable('kern', [1e-4, 1e3], 'Transform', 'log');
vars = [box, kern];
fun = @(vars) kfoldLoss(fitcsvm(X, Y, 'BoxConstraint', vars.box, 'KernelScale', vars.kern, ...
    'Kfold', 5));
```

## Optimize in Parallel

Search for the parameters that give the lowest cross-validation error by using parallel Bayesian optimization.

```
results = bayesopt(fun, vars, 'UseParallel', true);
```

Copying objective function to workers...

Done copying objective function to workers.

Iter	Active workers	Eval result	Objective	Objective runtime	BestSoFar (observed)	BestSoFar (estim.)
1	3	Best	0.26496	0.10234	0.26496	0.26497
2	3	Accept	0.35897	0.11516	0.26496	0.26497
3	3	Accept	0.35897	0.12859	0.26496	0.26497
4	3	Accept	0.35897	0.10744	0.26496	0.26497
5	3	Best	0.1453	0.090465	0.1453	0.14531
6	3	Accept	0.35897	0.086777	0.1453	0.14531
7	3	Accept	0.35897	0.087497	0.1453	0.14531
8	3	Accept	0.35328	0.086763	0.1453	0.14531
9	6	Accept	0.35897	0.083032	0.1453	0.18843
10	5	Best	0.12536	0.13637	0.12536	0.13682
11	5	Accept	0.12536	0.26935	0.12536	0.13682
12	6	Accept	0.14815	0.089356	0.12536	0.13386
13	6	Best	0.11396	17.748	0.11396	0.11367
14	6	Accept	0.1453	0.15733	0.11396	0.12396
15	6	Accept	0.35897	0.084854	0.11396	0.12348
16	6	Accept	0.22792	19.372	0.11396	0.12448
17	6	Accept	0.23647	20.253	0.11396	0.12352
18	6	Accept	0.12821	0.090442	0.11396	0.12264
19	5	Accept	0.18234	20.225	0.11396	0.12192

Iter	Active workers	Eval result	Objective	Objective runtime	BestSoFar (observed)	BestSoFar (estim.)
20	5	Accept	0.13105	0.11041	0.11396	0.12192
21	5	Accept	0.35897	0.083642	0.11396	0.12358
22	5	Accept	0.12251	0.090143	0.11396	0.11451
23	5	Accept	0.35897	0.091465	0.11396	0.11455
24	6	Accept	0.35897	0.09739	0.11396	0.11455
25	6	Accept	0.12821	0.18462	0.11396	0.11456
26	6	Accept	0.14815	0.1144	0.11396	0.11456
27	6	Best	0.11111	0.14838	0.11111	0.11117
28	5	Accept	0.2906	16.959	0.11111	0.11111
29	5	Accept	0.11966	0.090904	0.11111	0.11111
30	6	Accept	0.1339	0.26711	0.11111	0.11178

---

Optimization completed.  
MaxObjectiveEvaluations of 30 reached.  
Total function evaluations: 30  
Total elapsed time: 37.6057 seconds.  
Total objective function evaluation time: 97.5515

Best observed feasible point:

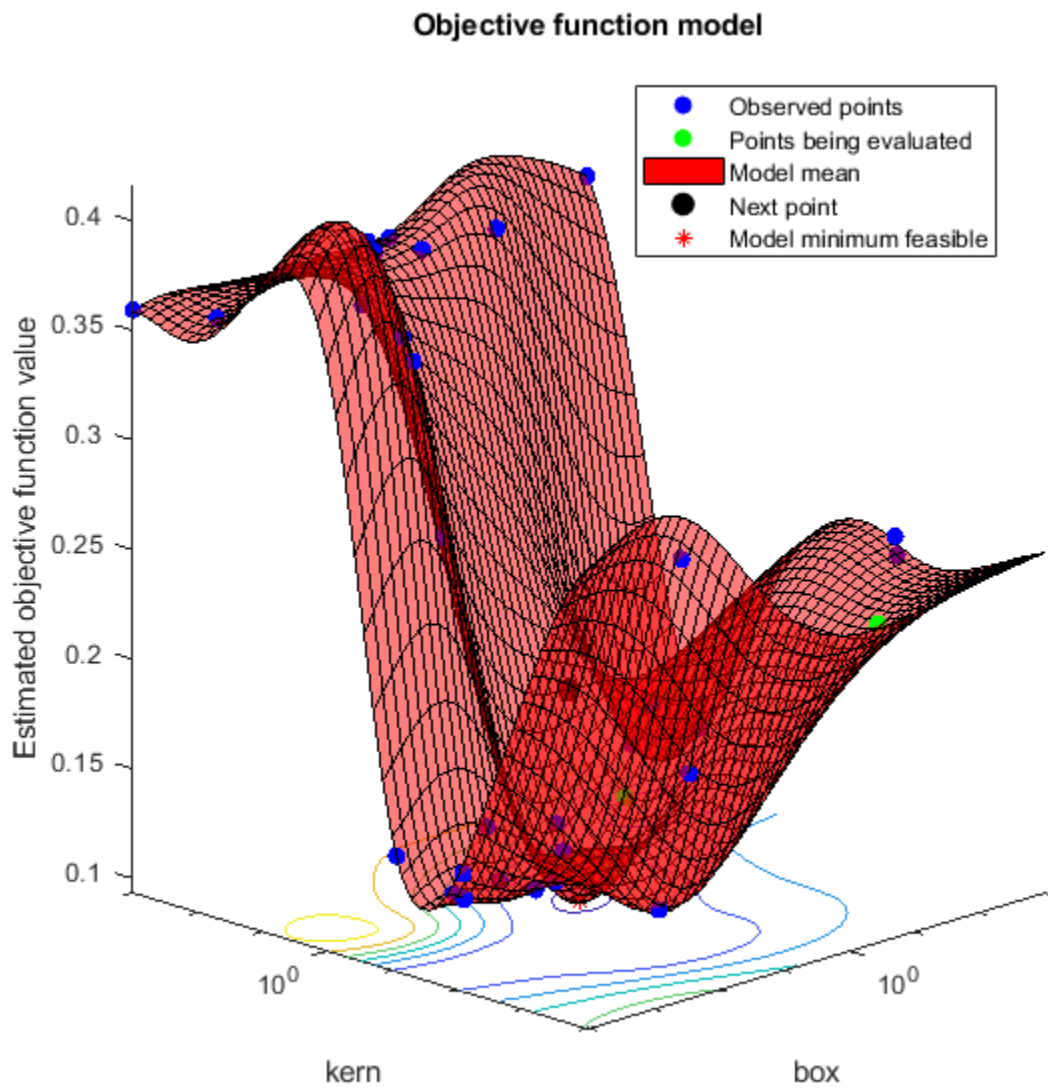
box	kern
0.034506	0.10621

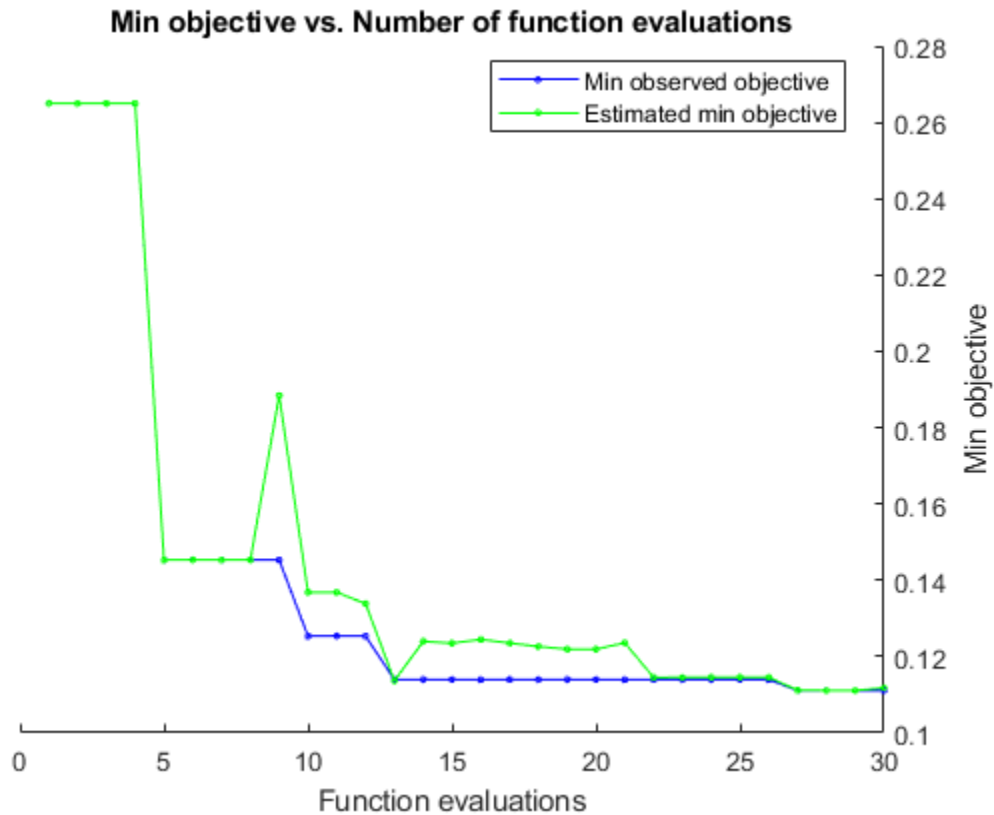
Observed objective function value = 0.11111  
Estimated objective function value = 0.11178  
Function evaluation time = 0.14838

Best estimated feasible point (according to models):

box	kern
0.034506	0.10621

Estimated objective function value = 0.11178  
Estimated function evaluation time = 0.14887





Train a classifier using the best point.

```
Mdl = fitcsvm(X,Y, 'BoxConstraint', results.XAtMinObjective.box, ...
              'KernelScale', results.XAtMinObjective.kern);
```

Observe that the optimal parameters are in Mdl.

```
Mdl.BoxConstraints(1)
Mdl.KernelParameters.Scale
```

```
ans =
```

```
0.0345
```

```
ans =
    0.1062
```

- “Optimize a Cross-Validated SVM Classifier Using Bayesian Optimization” on page 10-53

## Input Arguments

### **fun** — Objective function

function handle | `parallel.pool.Constant` whose Value is a function handle

Objective function, specified as a function handle or, when the `UseParallel` name-value pair is `true`, a `parallel.pool.Constant` whose Value is a function handle. Typically, `fun` returns a measure of loss (such as a misclassification error) for a machine learning model that has tunable hyperparameters to control its training. `fun` has these signatures:

```
objective = fun(x)
% or
[objective,constraints] = fun(x)
% or
[objective,constraints,UserData] = fun(x)
```

`fun` accepts `x`, a 1-by-D table of variable values, and returns `objective`, a real scalar representing the objective function value `fun(x)`.

Optionally, `fun` also returns:

- `constraints`, a real vector of coupled constraint violations. For a definition, see “Coupled Constraints” on page 32-170. `constraint(j) > 0` means constraint `j` is violated. `constraint(j) < 0` means constraint `j` is satisfied.
- `UserData`, an entity of any type (such as a scalar, matrix, structure, or object). For an example of a custom plot function that uses `UserData`, see “Create a Custom Plot Function” on page 10-16.

For details about using `parallel.pool.Constant` with `bayesopt`, see “Placing the Objective Function on Workers” on page 10-10.

Example: @objfun

Data Types: `function_handle`

### **vars** — Variable descriptions

vector of `optimizableVariable` objects defining the hyperparameters to be tuned

Variable descriptions, specified as a vector of `optimizableVariable` objects defining the hyperparameters to be tuned.

Example: `[X1,X2]`, where `X1` and `X2` are `optimizableVariable` objects

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `results = bayesopt(fun,vars,'AcquisitionFunctionName','expected-improvement-plus')`

### Algorithm Control

#### **AcquisitionFunctionName** — Function to choose next evaluation point

'expected-improvement-per-second-plus' (default) | 'expected-improvement' | 'expected-improvement-plus' | 'expected-improvement-per-second' | 'lower-confidence-bound' | 'probability-of-improvement'

Function to choose next evaluation point, specified as one of the listed choices. For algorithm descriptions, see “Acquisition Function Types” on page 10-4.

Example: 'expected-improvement-per-second'

Data Types: `char`

#### **IsObjectiveDeterministic** — Specify deterministic objective function

false (default) | true

Specify deterministic objective function, specified as `false` or `true`. If `fun` is stochastic (that is, `fun(x)` can return different values for the same `x`), then set

IsObjectiveDeterministic to false. In this case, bayesopt estimates a noise level during optimization.

Example: true

Data Types: logical

### **ExplorationRatio — Propensity to explore**

0.5 (default) | positive real

Propensity to explore, specified as a positive real. Applies to the 'expected-improvement-plus' and 'expected-improvement-per-second-plus' acquisition functions. See “Plus” on page 10-6.

Example: 0.2

Data Types: double

### **GPAciveSetSize — Fit Gaussian Process model to GPAciveSetSize or fewer points**

300 (default) | positive integer

Fit Gaussian Process model to GPAciveSetSize or fewer points, specified as a positive integer. When bayesopt has visited more than GPAciveSetSize points, subsequent iterations that use a GP model fit the model to GPAciveSetSize points. bayesopt chooses points uniformly at random without replacement among visited points. Using fewer points leads to faster GP model fitting, at the expense of possibly less accurate fitting.

Example: 80

Data Types: double

### **UseParallel — Compute in parallel**

false (default) | true

Compute in parallel, specified as false (do not compute in parallel) or true (compute in parallel). bayesopt performs parallel objective function evaluations concurrently on parallel workers. For algorithmic details, see “Parallel Bayesian Optimization” on page 10-9.

Example: true

Data Types: logical

**ParallelMethod** — Imputation method for parallel worker objective function values

'clipped-model-prediction' (default) | 'model-prediction' | 'max-observed'  
| 'min-observed'

Imputation method for parallel worker objective function values, specified as 'clipped-model-prediction', 'model-prediction', 'max-observed', or 'min-observed'. To generate a new point to evaluate, `bayesopt` fits a Gaussian process to all points, including the points being evaluated on workers. To fit the process, `bayesopt` imputes objective function values for the points that are currently on workers. `ParallelMethod` specifies the method used for imputation.

- 'clipped-model-prediction' — Impute the maximum of these quantities:
  - Mean Gaussian process prediction at the point  $x$
  - Minimum observed objective function among feasible points visited
  - Minimum model prediction among all feasible points
- 'model-prediction' — Impute the mean Gaussian process prediction at the point  $x$ .
- 'max-observed' — Impute the maximum observed objective function value among feasible points.
- 'min-observed' — Impute the minimum observed objective function value among feasible points.

Example: 'max-observed'

Data Types: char

**MinWorkerUtilization** — Tolerance on number of active parallel workers

`floor(0.8*Nworkers)` (default) | positive integer

Tolerance on the number of active parallel workers, specified as a positive integer. After `bayesopt` assigns a point to evaluate, and before it computes a new point to assign, it checks whether fewer than `MinWorkerUtilization` workers are active. If so, `bayesopt` assigns random points within bounds to all available workers. Otherwise, `bayesopt` calculates the best point for one worker. `bayesopt` creates random points much faster than fitted points, so this behavior leads to higher utilization of workers, at the cost of possibly poorer points. For details, see “Parallel Bayesian Optimization” on page 10-9.

Example: 3



Data Types: double

### Starting and Stopping

#### **MaxObjectiveEvaluations** — Objective function evaluation limit

30 (default) | positive integer

Objective function evaluation limit, specified as a positive integer.

Example: 60

Data Types: double

#### **MaxTime** — Time limit

Inf (default) | positive real

Time limit, specified as a positive real. The time limit is in seconds, as measured by `tic` and `toc`.

Run time can exceed `MaxTime` because `bayesopt` does not interrupt function evaluations.

Example: 3600

Data Types: double

#### **NumSeedPoints** — Number of initial evaluation points

4 (default) | positive integer

Number of initial evaluation points, specified as a positive integer. `bayesopt` chooses these points randomly within the variable bounds, according to the setting of the `Transform` setting for each variable (uniform for 'none', logarithmically spaced for 'log').

Example: 10

Data Types: double

### Constraints

#### **XConstraintFcn** — Deterministic constraints on variables

[] (default) | function handle

Deterministic constraints on variables, specified as a function handle.

For details, see “Deterministic Constraints — XConstraintFcn” on page 10-44.

Example: `@xconstraint`

Data Types: `function_handle`

### **ConditionalVariableFcn** — Conditional variable constraints

`[]` (default) | `function handle`

Conditional variable constraints, specified as a function handle.

For details, see “Conditional Constraints — ConditionalVariableFcn” on page 10-45.

Example: `@condfun`

Data Types: `function_handle`

### **NumCoupledConstraints** — Number of coupled constraints

`0` (default) | `positive integer`

Number of coupled constraints, specified as a positive integer. For details, see “Coupled Constraints” on page 10-47.

---

**Note** `NumCoupledConstraints` is required when you have coupled constraints.

---

Example: `3`

Data Types: `double`

### **AreCoupledConstraintsDeterministic** — Indication of whether coupled constraints are deterministic

`true` for all coupled constraints (default) | `logical vector`

Indication of whether coupled constraints are deterministic, specified as a logical vector of length `NumCoupledConstraints`. For details, see “Coupled Constraints” on page 10-47.

Example: `[true, false, true]`

Data Types: `logical`

## Reports, Plots, and Halting

### Verbose — Command-line display level

1 (default) | 0 | 2

Command-line display level, specified as 0, 1, or 2.

- 0 — No command-line display.
- 1 — At each iteration, display the iteration number, result report (see the next paragraph), objective function model, objective function evaluation time, best (lowest) observed objective function value, best (lowest) estimated objective function value, and the observed constraint values (if any). When optimizing in parallel, the display also includes a column showing the number of active workers, counted after assigning a job to the next worker.

The result report for each iteration is one of the following:

- `Accept` — The objective function returns a finite value, and all constraints are satisfied.
- `Best` — Constraints are satisfied, and the objective function returns the lowest value among feasible points.
- `Error` — The objective function returns a value that is not a finite real scalar.
- `Infeas` — At least one constraint is violated.
- 2 — Same as 1, adding diagnostic information such as time to select the next point, model fitting time, indication that "plus" acquisition functions declare overexploiting, and parallel workers are being assigned to random points due to low parallel utilization.

Example: 2

Data Types: `double`

### OutputFcn — Function called after each iteration

{ } (default) | function handle | cell array of function handles

Function called after each iteration, specified as a function handle or cell array of function handles. An output function can halt the solver, and can perform arbitrary calculations, including creating variables or plotting. Specify several output functions using a cell array of function handles.

There are two built-in output functions:

- `@assignInBase` — Constructs a `BayesianOptimization` instance at each iteration and assigns it to a variable in the base workspace. Choose a variable name using the `SaveVariableName` name-value pair.
- `@saveToFile` — Constructs a `BayesianOptimization` instance at each iteration and saves it to a file in the current folder. Choose a file name using the `SaveFileName` name-value pair.

You can write your own output functions. For details, see “Bayesian Optimization Output Functions” on page 10-23.

Example: `{@saveToFile @myOutputFunction}`

Data Types: `cell` | `function_handle`

#### **SaveFileName** — File name for the `@saveToFile` output function

'BayesoptResults.mat' (default) | character vector

File name for the `@saveToFile` output function, specified as a character vector. The file name can include a path, such as `'../optimizations/September2.mat'`.

Example: `'September2.mat'`

Data Types: `char`

#### **SaveVariableName** — Variable name for the `@assignInBase` output function

'BayesoptResults' (default) | character vector

Variable name for the `@assignInBase` output function, specified as a character vector.

Example: `'September2Results'`

Data Types: `char`

#### **PlotFcn** — Plot function called after each iteration

`{@plotObjectiveModel,@plotMinObjective}` (default) | 'all' | function handle | cell array of function handles

Plot function called after each iteration, specified as 'all', a function handle, or a cell array of function handles. A plot function can halt the solver, and can perform arbitrary calculations, including creating variables, in addition to plotting.

Specify no plot function as `[]`.

'all' calls all built-in plot functions. Specify several plot functions using a cell array of function handles.

The built-in plot functions appear in the following tables.

<b>Model Plots — Apply When <math>D \leq 2</math></b>	<b>Description</b>
@plotAcquisitionFunction	Plot the acquisition function surface.
@plotConstraintModels	Plot each constraint model surface. Also plots a $P(\text{feasible})$ surface. Negative values indicate feasible points.  Also plot the error model, if it exists, which ranges from $-1$ to $1$ . Negative values mean that the model probably does not error, positive values mean that it probably does error. The model is:  Plotted error = $-2 * \text{Probability}(\text{error}) + 1$ .
@plotObjectiveEvaluationTimeModel	Plot the objective function evaluation time model surface.
@plotObjectiveModel	Plot the fun model surface, the estimated location of the minimum, and the location of the next proposed point to evaluate. For one-dimensional problems, plot envelopes one credible interval above and below the mean function, and envelopes one noise standard deviation above and below the mean.
<b>Trace Plots — Apply to All <math>D</math></b>	<b>Description</b>
@plotObjective	Plot each observed function value versus the number of function evaluations.
@plotObjectiveEvaluationTime	Plot each observed function evaluation run time versus the number of function evaluations.
@plotMinObjective	Plot the minimum observed and estimated function values versus the number of function evaluations.
@plotElapsedTime	Plot three curves: the total elapsed time of the optimization, the total function evaluation time, and the total modeling and point selection time, all versus the number of function evaluations.

You can write your own plot functions. For details, see “Bayesian Optimization Plot Functions” on page 10-14.

---

**Note** When there are coupled constraints, iterative display and plot functions can give counterintuitive results such as:

- A *minimum objective* plot can increase.
- The optimization can declare a problem infeasible even when it showed an earlier feasible point.

The reason for this behavior is that the decision about whether a point is feasible can change as the optimization progresses. `bayesopt` determines feasibility with respect to its constraint model, and this model changes as `bayesopt` evaluates points. So a “minimum objective” plot can increase when the minimal point is later deemed infeasible, and the iterative display can show a feasible point that is later deemed infeasible.

---

Example: `'all'`

Data Types: `char` | `cell` | `function_handle`

### Initialization

#### **InitialX** — Initial evaluation points

`NumSeedPoints`-by-`D` random initial points within bounds (default) | `N`-by-`D` table

Initial evaluation points, specified as an `N`-by-`D` table, where `N` is the number of evaluation points, and `D` is the number of variables.

---

**Note** If only `InitialX` is provided, it is interpreted as initial points to evaluate. The objective function is evaluated at `InitialX`.

If any other initialization parameters are also provided, `InitialX` is interpreted as prior function evaluation data. The objective function is not evaluated. Any missing values are set to `NaN`.

---

Data Types: `table`

**InitialObjective** — Objective values corresponding to `InitialX`

[] (default) | length-N vector

Objective values corresponding to `InitialX`, specified as a length-N vector, where N is the number of evaluation points.

Example: [17;-3;-12.5]

Data Types: double

**InitialConstraintViolations** — Constraint violations of coupled constraints

[] (default) | N-by-K matrix

Constraint violations of coupled constraints, specified as an N-by-K matrix, where N is the number of evaluation points and K is the number of coupled constraints. For details, see “Coupled Constraints” on page 10-47.

Data Types: double

**InitialErrorValues** — Errors for `InitialX`

[] (default) | length-N vector with entries -1 or 1

Errors for `InitialX`, specified as a length-N vector with entries -1 or 1, where N is the number of evaluation points. Specify -1 for no error, and 1 for an error.

Example: [-1,-1,-1,-1,1]

Data Types: double

**InitialUserData** — Initial data corresponding to `InitialX`

[] (default) | length-N cell vector

Initial data corresponding to `InitialX`, specified as a length-N cell vector, where N is the number of evaluation points.

Example: {2,3,-1}

Data Types: cell

**InitialObjectiveEvaluationTimes** — Evaluation times of objective function at `InitialX`

[] (default) | length-N vector

Evaluation times of objective function at `InitialX`, specified as a length-`N` vector, where `N` is the number of evaluation points. Time is measured in seconds.

Data Types: `double`

**InitialIterationTimes** — Times for the first `N` iterations

`{}` (default) | length-`N` vector

Times for the first `N` iterations, specified as a length-`N` vector, where `N` is the number of evaluation points. Time is measured in seconds.

Data Types: `double`

## Output Arguments

**results** — Bayesian optimization results

`BayesianOptimization` object

Bayesian optimization results, returned as a `BayesianOptimization` object.

## Definitions

### Coupled Constraints

Coupled constraints are those constraints whose value comes from the objective function calculation. See “Coupled Constraints” on page 10-47.

## See Also

`BayesianOptimization` | `optimizableVariable`

## Topics

“Optimize a Cross-Validated SVM Classifier Using Bayesian Optimization” on page 10-53

“Bayesian Optimization Algorithm” on page 10-2



**Introduced in R2016b**

## bbdesign

Box-Behnken design

### Syntax

```
dBB = bbdesign(n)
[dBB,blocks] = bbdesign(n)
[...] = bbdesign(n,param,val)
```

### Description

`dBB = bbdesign(n)` generates a Box-Behnken design for  $n$  factors.  $n$  must be an integer 3 or larger. The output matrix `dBB` is  $m$ -by- $n$ , where  $m$  is the number of runs in the design. Each row represents one run, with settings for all factors represented in the columns. Factor values are normalized so that the cube points take values between  $-1$  and  $1$ .

`[dBB,blocks] = bbdesign(n)` requests a blocked design. The output `blocks` is an  $m$ -by-1 vector of block numbers for each run. Blocks indicate runs that are to be measured under similar conditions to minimize the effect of inter-block differences on the parameter estimates.

`[...] = bbdesign(n,param,val)` specifies one or more optional parameter/value pairs for the design. The following table lists valid parameter/value pairs.

Parameter	Description	Values
'center'	Number of center points.	Integer. The default depends on $n$ .
'blocksize'	Maximum number of points per block.	Integer. The default is <code>Inf</code> .

### Examples

The following creates a 3-factor Box-Behnken design:

```

dBB = bbdesign(3)
dBB =
    -1    -1     0
    -1     1     0
     1    -1     0
     1     1     0
    -1     0    -1
    -1     0     1
     1     0    -1
     1     0     1
     0    -1    -1
     0    -1     1
     0     1    -1
     0     1     1
     0     0     0
     0     0     0
     0     0     0

```

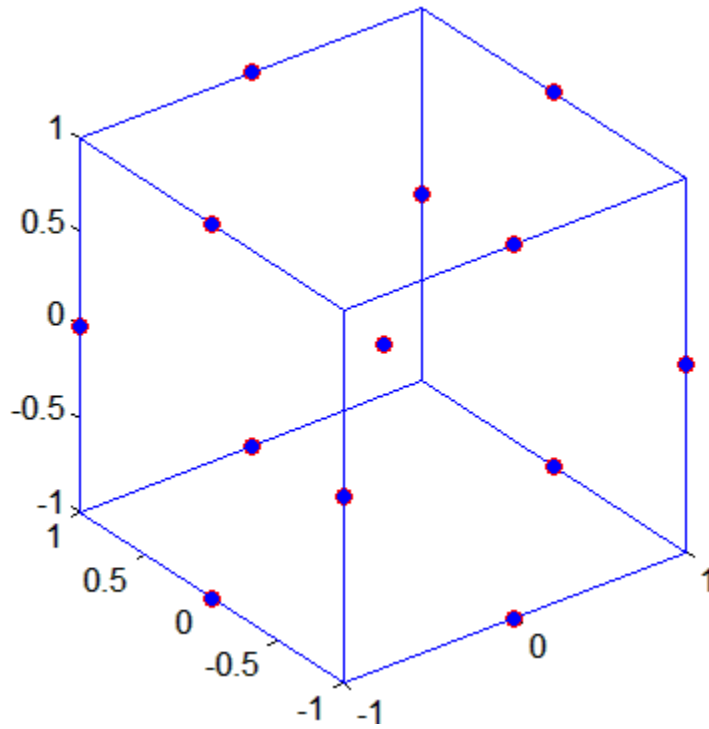
The center point is run 3 times to allow for a more uniform estimate of the prediction variance over the entire design space.

Visualize the design as follows:

```

plot3(dBB(:,1),dBB(:,2),dBB(:,3),'ro',...
      'MarkerFaceColor','b')
X = [1 -1 -1 -1 1 -1 -1 -1 1 1 -1 -1; ...
     1 1 1 -1 1 1 1 -1 1 1 -1 -1];
Y = [-1 -1 1 -1 -1 -1 1 -1 1 -1 1 -1; ...
     1 -1 1 1 1 -1 1 1 1 -1 1 -1];
Z = [1 1 1 1 -1 -1 -1 -1 -1 -1 -1 -1; ...
     1 1 1 1 -1 -1 -1 -1 1 1 1 1];
line(X,Y,Z,'Color','b')
axis square equal

```



## See Also

`ccdesign`

Introduced before R2006a

# bestPoint

Best point in a Bayesian optimization according to a criterion

## Syntax

```
x = bestPoint(results)
x = bestPoint(results, Name, Value)
[x, CriterionValue] = bestPoint( ___ )
[x, CriterionValue, iteration] = bestPoint( ___ )
```

## Description

`x = bestPoint(results)` returns the best feasible point in the Bayesian model results according to the default criterion 'min-visited-upper-confidence-interval'.

`x = bestPoint(results, Name, Value)` modifies the best point using name-value pairs.

`[x, CriterionValue] = bestPoint( ___ )`, for any previous syntax, also returns the value of the criterion at `x`.

`[x, CriterionValue, iteration] = bestPoint( ___ )` also returns the iteration number at which the best point was returned. Applies when the Criterion name-value pair is 'min-observed', 'min-visited-mean', or the default 'min-visited-upper-confidence-interval'.

## Examples

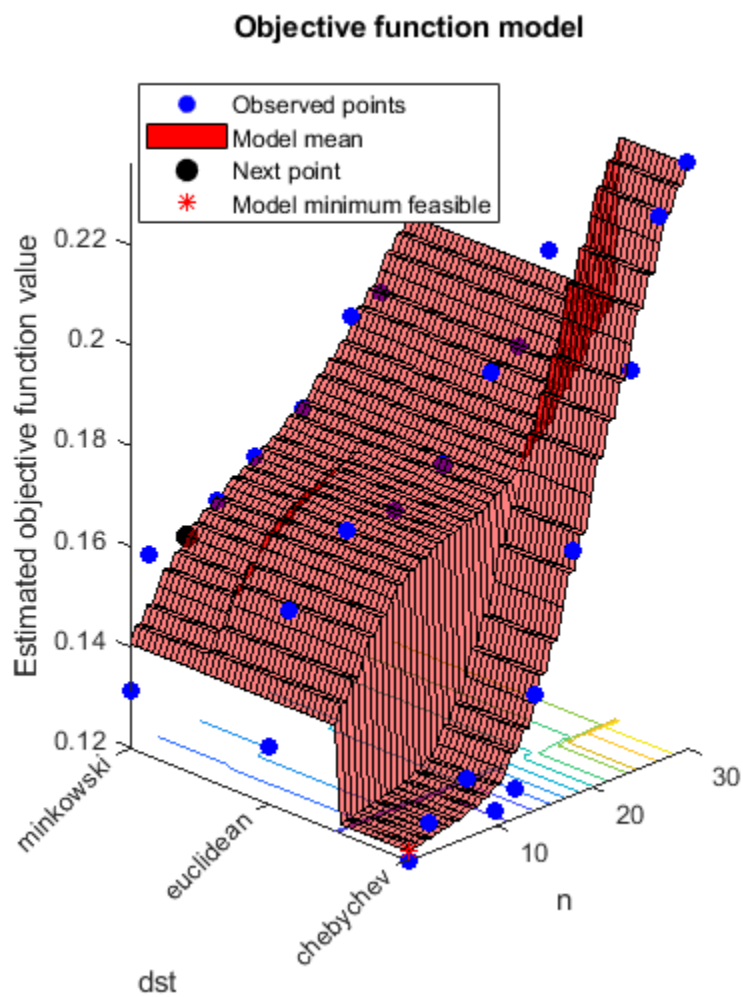
### Best Point of an Optimized KNN Classifier

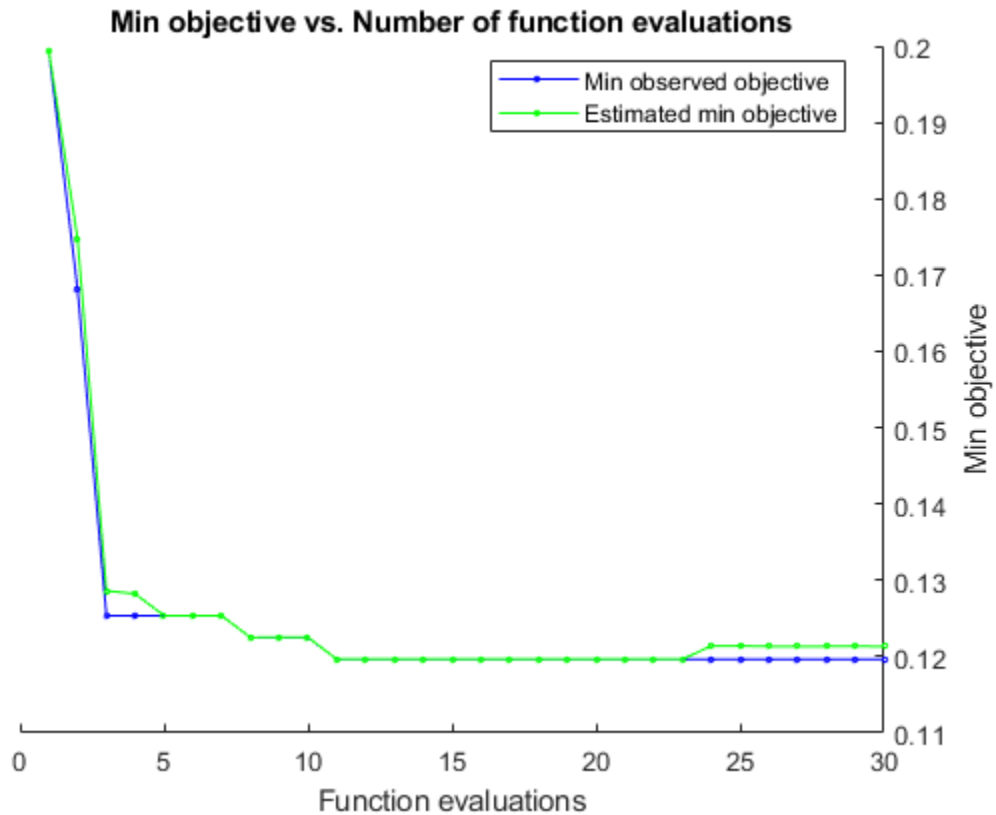
This example shows how to obtain the best point of an optimized classifier.

Optimize a KNN classifier for the `ionosphere` data, meaning find parameters that minimize the cross-validation loss. Minimize over nearest-neighborhood sizes from 1 to 30, and over the distance functions 'chebychev', 'euclidean', and 'minkowski'.

For reproducibility, set the random seed, and set the `AcquisitionFunctionName` option to 'expected-improvement-plus'.

```
load ionosphere
rng default
num = optimizableVariable('n', [1, 30], 'Type', 'integer');
dst = optimizableVariable('dst', {'chebychev', 'euclidean', 'minkowski'}, 'Type', 'categorical');
c = cvpartition(351, 'Kfold', 5);
fun = @(x) kfoldLoss(fitcknn(X, Y, 'CVPartition', c, 'NumNeighbors', x.n, ...
    'Distance', char(x.dst), 'NSMethod', 'exhaustive'));
results = bayesopt(fun, [num, dst], 'Verbose', 0, ...
    'AcquisitionFunctionName', 'expected-improvement-plus');
```





Obtain the best point according to the default 'min-visited-upper-confidence-interval' criterion.

```
x = bestPoint(results)
```

```
x =
```

```
1x2 table
```

```

n      dst
--  -----

```



```
1 chebychev
```

The lowest estimated cross-validation loss occurs for ten nearest neighbors and 'chebychev' distance.

Careful examination of the objective function model plot shows a point with one nearest neighbor and 'chebychev' distance that has a lower objective function value. Find this point using a different criterion.

```
x = bestPoint(results, 'Criterion', 'min-observed')
```

```
x =
```

```
1x2 table
```

n	dst
1	chebychev

Also find the minimum observed objective function value, and the iteration number at which it was observed.

```
[x,CriterionValue,iteration] = bestPoint(results, 'Criterion', 'min-observed')
```

```
x =
```

```
1x2 table
```

n	dst
1	chebychev

```
CriterionValue =
```

```
0.1197
```

```
iteration =
```

## Input Arguments

### **results** — Bayesian optimization results

BayesianOptimization object

Bayesian optimization results, specified as a BayesianOptimization object.

### Name-Value Pair Arguments

Specify optional comma-separated pairs of Name, Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as Name1, Value1, ..., NameN, ValueN.

Example: `x = bestPoint(results, 'Criterion', 'min-observed')`

### **Criterion** — Best point criterion

'min-visited-upper-confidence-interval' (default) | criterion name

Best point criterion, specified as a criterion name. The names are case-insensitive, do not require - characters, and require only enough characters to make the name uniquely distinguishable.

Criterion Name	Meaning
'min-observed'	x is the feasible point with minimum observed objective.
'min-mean'	x is the feasible point where the objective model mean is minimized.
'min-upper-confidence-interval'	x is the feasible point minimizing an upper confidence interval of the objective model. See alpha.
'min-visited-mean'	x is the feasible point where the objective model mean is minimized among the visited points.

Criterion Name	Meaning
'min-visited-upper-confidence-interval'	x is the feasible point minimizing an upper confidence interval of the objective model among the visited points. See alpha.

Example: 'min-visited-mean'

Data Types: char

### **alpha** — Probability that modeled objective mean exceeds CriterionValue

0.01 (default) | scalar between 0 and 1

Probability that the modeled objective mean exceeds CriterionValue, specified as a scalar between 0 and 1. alpha relates to the 'min-upper-confidence-interval' and 'min-visited-upper-confidence-interval' Criterion values. The definition for the upper confidence interval is the value  $\bar{Y}$  where

$$P(\text{mean}_Q(\text{fun}(x)) > \bar{Y}) = \text{alpha},$$

where fun is the objective function, and the mean is calculated with respect to the posterior distribution  $Q$ .

Example: 0.05

Data Types: double

## Output Arguments

### **x** — Best point

1-by-D table

Best point, returned as a 1-by-D table, where D is the number of variables. The meaning of “best” is with respect to Criterion.

### **CriterionValue** — Value of criterion

real scalar

Value of criterion, returned as a real scalar. The value depends on the setting of the Criterion name-value pair, which has a default value of 'min-visited-upper-confidence-interval'.

Criterion Name	Meaning
'min-observed'	Minimum observed objective.
'min-mean'	Minimum of model mean.
'min-upper-confidence-interval'	Value $Y$ satisfying the equation $P(\text{mean}_Q(\text{fun}(x)) > Y) = \text{alpha}$ .
'min-visited-mean'	Minimum of observed model mean.
'min-visited-upper-confidence-interval'	Value $Y$ satisfying the equation $P(\text{mean}_Q(\text{fun}(x)) > Y) = \text{alpha}$ among observed points.

**iteration** — Iteration number at which best point was observed

positive integer

Iteration number at which best point was observed, returned as a positive integer. The best point is defined by `CriterionValue`.

## See Also

`BayesianOptimization`**Introduced in R2016b**

# betacdf

Beta cumulative distribution function

## Syntax

```
p = betacdf(x, a, b)
p = betacdf(x, a, b, 'upper')
```

## Description

`p = betacdf(x, a, b)` returns the beta cdf at each of the values in `x` using the corresponding parameters in `a` and `b`. `x`, `a`, and `b` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array with the same dimensions as the other inputs. The parameters in `a` and `b` must all be positive, and the values in `x` must lie on the interval  $[0,1]$ .

`p = betacdf(x, a, b, 'upper')` returns the complement of the beta cdf at each of the values in `x`, using an algorithm that more accurately computes the extreme upper tail probabilities.

The beta cdf for a given value `x` and given pair of parameters `a` and `b` is

$$p = F(x | a, b) = \frac{1}{B(a, b)} \int_0^x t^{a-1} (1-t)^{b-1} dt$$

where  $B(\cdot)$  is the Beta function.

## Examples

### Compute Beta Distribution CDF

Compute the cdf for a beta distribution with parameters `a = 2` and `b = 2`.

```
x = 0.1:0.2:0.9;  
a = 2;  
b = 2;  
p = betacdf(x,a,b)
```

```
p =  
  
    0.0280    0.2160    0.5000    0.7840    0.9720
```

```
a = [1 2 3];  
p = betacdf(0.5,a,a)
```

```
p =  
  
    0.5000    0.5000    0.5000
```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

[betafit](#) | [betainv](#) | [betalike](#) | [betapdf](#) | [betarnd](#) | [betastat](#) | [cdf](#)

### Topics

“Beta Distribution” on page B-4

Introduced before R2006a

# betafit

Beta parameter estimates

## Syntax

```
phat = betafit(data)
[phat,pci] = betafit(data,alpha)
```

## Description

`phat = betafit(data)` computes the maximum likelihood estimates of the beta distribution parameters  $a$  and  $b$  from the data in the vector `data` and returns a column vector containing the  $a$  and  $b$  estimates, where the beta cdf is given by

$$F(x | a, b) = \frac{1}{B(a, b)} \int_0^x t^{a-1} (1-t)^{b-1} dt$$

and  $B(\cdot)$  is the Beta function. The elements of `data` must lie in the open interval  $(0, 1)$ , where the beta distribution is defined. However, it is sometimes also necessary to fit a beta distribution to data that include exact zeros or ones. For such data, the beta likelihood function is unbounded, and standard maximum likelihood estimation is not possible. In that case, `betafit` maximizes a modified likelihood that incorporates the zeros or ones by treating them as if they were values that have been left-censored at `sqrt(realmin)` or right-censored at `1-eps/2`, respectively.

`[phat,pci] = betafit(data,alpha)` returns confidence intervals on the  $a$  and  $b$  parameters in the 2-by-2 matrix `pci`. The first column of the matrix contains the lower and upper confidence bounds for parameter  $a$ , and the second column contains the confidence bounds for parameter  $b$ . The optional input argument `alpha` is a value in the range  $[0, 1]$  specifying the width of the confidence intervals. By default, `alpha` is `0.05`, which corresponds to 95% confidence intervals. The confidence intervals are based on a normal approximation for the distribution of the logs of the parameter estimates.

## Examples

This example generates 100 beta distributed observations. The true  $a$  and  $b$  parameters are 4 and 3, respectively. Compare these to the values returned in `p` by the beta fit. Note that the columns of `ci` both bracket the true parameters.

```
data = betarnd(4,3,100,1);
[p,ci] = betafit(data,0.01)
p =
    5.5328    3.8097
ci =
    3.6538    2.6197
    8.3781    5.5402
```

## References

- [1] Hahn, Gerald J., and S. S. Shapiro. *Statistical Models in Engineering*. Hoboken, NJ: John Wiley & Sons, Inc., 1994, p. 95.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`betacdf` | `betainv` | `betalike` | `betapdf` | `betarnd` | `betastat` | `mle`

### Topics

“Beta Distribution” on page B-4

Introduced before R2006a



## betainv

Beta inverse cumulative distribution function

### Syntax

```
X = betainv(P,A,B)
```

### Description

`X = betainv(P,A,B)` computes the inverse of the beta cdf with parameters specified by `A` and `B` for the corresponding probabilities in `P`. `P`, `A`, and `B` can be vectors, matrices, or multidimensional arrays that are all the same size. A scalar input is expanded to a constant array with the same dimensions as the other inputs. The parameters in `A` and `B` must all be positive, and the values in `P` must lie on the interval  $[0, 1]$ .

The inverse beta cdf for a given probability  $p$  and a given pair of parameters  $a$  and  $b$  is

$$x = F^{-1}(p | a, b) = \{x : F(x | a, b) = p\}$$

where

$$p = F(x | a, b) = \frac{1}{B(a, b)} \int_0^x t^{a-1} (1-t)^{b-1} dt$$

and  $B(\cdot)$  is the Beta function. Each element of output `X` is the value whose cumulative probability under the beta cdf defined by the corresponding parameters in `A` and `B` is specified by the corresponding value in `P`.

### Examples

```
p = [0.01 0.5 0.99];
x = betainv(p,10,5)
x =
    0.3726    0.6742    0.8981
```

According to this result, for a beta cdf with  $a = 10$  and  $b = 5$ , a value less than or equal to 0.3726 occurs with probability 0.01. Similarly, values less than or equal to 0.6742 and 0.8981 occur with respective probabilities 0.5 and 0.99.

## Algorithms

The `betainv` function uses Newton's method with modifications to constrain steps to the allowable range for  $x$ , i.e.,  $[0 \ 1]$ .

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`betacdf` | `betafit` | `betainv` | `betalike` | `betapdf` | `betarnd` | `betastat` | `icdf`

### Topics

“Beta Distribution” on page B-4

Introduced before R2006a

# betalike

Beta negative log-likelihood

## Syntax

```
nlogL = betalike(params, data)
[nlogL, AVAR] = betalike(params, data)
```

## Description

`nlogL = betalike(params, data)` returns the negative of the beta log-likelihood function for the beta parameters  $a$  and  $b$  specified in vector `params` and the observations specified in the column vector `data`.

The elements of `data` must lie in the open interval  $(0, 1)$ , where the beta distribution is defined. However, it is sometimes also necessary to fit a beta distribution to data that include exact zeros or ones. For such data, the beta likelihood function is unbounded, and standard maximum likelihood estimation is not possible. In that case, `betalike` computes a modified likelihood that incorporates the zeros or ones by treating them as if they were values that have been left-censored at `sqrt(realmin)` or right-censored at  $1 - \text{eps}/2$ , respectively.

`[nlogL, AVAR] = betalike(params, data)` also returns `AVAR`, which is the asymptotic variance-covariance matrix of the parameter estimates if the values in `params` are the maximum likelihood estimates. `AVAR` is the inverse of Fisher's information matrix. The diagonal elements of `AVAR` are the asymptotic variances of their respective parameters.

`betalike` is a utility function for maximum likelihood estimation of the beta distribution. The likelihood assumes that all the elements in the data sample are mutually independent. Since `betalike` returns the negative beta log-likelihood function, minimizing `betalike` using `fminsearch` is the same as maximizing the likelihood.

## Examples

This example continues the `betafit` example, which calculates estimates of the beta parameters for some randomly generated beta distributed data.

```
r = betarnd(4,3,100,1);
[nlogl,AVAR] = betalike(betafit(r),r)
nlogl =

    -27.5996

AVAR =

    0.2783    0.1316
    0.1316    0.0867
```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

[betacdf](#) | [betafit](#) | [betainv](#) | [betapdf](#) | [betarnd](#) | [betastat](#)

### Topics

“Beta Distribution” on page B-4

**Introduced before R2006a**

# betapdf

Beta probability density function

## Syntax

`Y = betapdf(X,A,B)`

## Description

`Y = betapdf(X,A,B)` computes the beta pdf at each of the values in `X` using the corresponding parameters in `A` and `B`. `X`, `A`, and `B` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array with the same dimensions of the other inputs. The parameters in `A` and `B` must all be positive, and the values in `X` must lie on the interval  $[0, 1]$ .

The beta probability density function for a given value  $x$  and given pair of parameters  $a$  and  $b$  is

$$y = f(x | a, b) = \frac{1}{B(a, b)} x^{a-1} (1-x)^{b-1} I_{[0,1]}(x)$$

where  $B(\cdot)$  is the Beta function. The uniform distribution on  $(0, 1)$  is a degenerate case of the beta pdf where  $a = 1$  and  $b = 1$ .

A *likelihood function* is the pdf viewed as a function of the parameters. Maximum likelihood estimators (MLEs) are the values of the parameters that maximize the likelihood function for a fixed value of  $x$ .

## Examples

```
a = [0.5 1; 2 4]
a =
    0.5000    1.0000
    2.0000    4.0000
y = betapdf(0.5, a, a)
```

```
y =  
    0.6366    1.0000  
    1.5000    2.1875
```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

[betacdf](#) | [betafit](#) | [betainv](#) | [betalike](#) | [betarnd](#) | [betastat](#) | [pdf](#)

### Topics

“Beta Distribution” on page B-4

**Introduced before R2006a**

# betarnd

Beta random numbers

## Syntax

```
R = betarnd(A,B)
R = betarnd(A,B,m,n,...)
R = betarnd(A,B,[m,n,...])
```

## Description

`R = betarnd(A,B)` generates random numbers from the beta distribution with parameters specified by `A` and `B`. `A` and `B` can be vectors, matrices, or multidimensional arrays that have the same size, which is also the size of `R`. A scalar input for `A` or `B` is expanded to a constant array with the same dimensions as the other input.

`R = betarnd(A,B,m,n,...)` or `R = betarnd(A,B,[m,n,...])` generates an `m`-by-`n`-by-... array containing random numbers from the beta distribution with parameters `A` and `B`. `A` and `B` can each be scalars or arrays of the same size as `R`.

## Examples

```
a = [1 1;2 2];
b = [1 2;1 2];

r = betarnd(a,b)
r =
    0.6987    0.6139
    0.9102    0.8067

r = betarnd(10,10,[1 5])
r =
    0.5974    0.4777    0.5538    0.5465    0.6327

r = betarnd(4,2,2,3)
```

```
r =  
    0.3943    0.6101    0.5768  
    0.5990    0.2760    0.5474
```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

The generated code can return a different sequence of numbers than MATLAB if either of the following is true:

- The output is nonscalar.
- An input parameter is invalid for the distribution.

### See Also

`betacdf` | `betafit` | `betainv` | `betalike` | `betapdf` | `betastat` | `random`

### Topics

“Beta Distribution” on page B-4

**Introduced before R2006a**



# betastat

Beta mean and variance

## Syntax

```
[M,V] = betastat(A,B)
```

## Description

`[M,V] = betastat(A,B)`, with  $A > 0$  and  $B > 0$ , returns the mean of and variance for the beta distribution with parameters specified by  $A$  and  $B$ .  $A$  and  $B$  can be vectors, matrices, or multidimensional arrays that have the same size, which is also the size of  $M$  and  $V$ . A scalar input for  $A$  or  $B$  is expanded to a constant array with the same dimensions as the other input.

The mean of the beta distribution with parameters  $a$  and  $b$  is  $a / (a + b)$  and the variance is

$$\frac{ab}{(a+b+1)(a+b)^2}$$

## Examples

If parameters  $a$  and  $b$  are equal, the mean is 1/2.

```
a = 1:6;
[m,v] = betastat(a,a)
m =
    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000
v =
    0.0833    0.0500    0.0357    0.0278    0.0227    0.0192
```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`betacdf` | `betafit` | `betainv` | `betalike` | `betapdf` | `betarnd`

### Topics

“Beta Distribution” on page B-4

Introduced before R2006a

## BIC property

**Class:** gmdistribution

Bayes Information Criterion

## Description

The Bayes Information Criterion:  $2 \cdot N \log L + m \cdot \log(n)$ , where  $N \log L$  is the negative loglikelihood,  $n$  is the number of observations, and  $m$  is the number of estimated parameters.

## binocdf

Binomial cumulative distribution function

### Syntax

```
y = binocdf(x,N,p)
y = binocdf(x,N,p,'upper')
```

### Description

`y = binocdf(x,N,p)` computes a binomial cdf at each of the values in `x` using the corresponding number of trials in `N` and probability of success for each trial in `p`. `x`, `N`, and `p` can be vectors, matrices, or multidimensional arrays that are all the same size. A scalar input is expanded to a constant array with the same dimensions of the other inputs. The values in `N` must all be positive integers, the values in `x` must lie on the interval  $[0, N]$ , and the values in `p` must lie on the interval  $[0, 1]$ .

`y = binocdf(x,N,p,'upper')` returns the complement of the binomial cdf at each value in `x`, using an algorithm that more accurately computes the extreme upper tail probabilities.

The binomial cdf for a given value  $x$  and a given pair of parameters  $n$  and  $p$  is

$$y = F(x | n, p) = \sum_{i=0}^x \binom{n}{i} p^i (1-p)^{(n-i)} I_{(0,1,\dots,n)}(i).$$

The result,  $y$ , is the probability of observing up to  $x$  successes in  $n$  independent trials,

where the probability of success in any given trial is  $p$ . The indicator function  $I_{(0,1,\dots,n)}(i)$  ensures that  $x$  only adopts values of  $0, 1, \dots, n$ .

### Examples

## Compute Binomial CDF

If a baseball team plays 162 games in a season and has a 50-50 chance of winning any game, then the probability of that team winning more than 100 games in a season is:

```
1 - binocdf(100,162,0.5)
```

```
ans =
```

```
0.0010
```

The result is 0.001 (i.e., 1-0.999). If a team wins 100 or more games in a season, this result suggests that it is likely that the team's true probability of winning any game is greater than 0.5.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

[binofit](#) | [binoinv](#) | [binopdf](#) | [binornd](#) | [binostat](#) | [cdf](#)

### Topics

“Binomial Distribution” on page B-9

Introduced before R2006a

## binofit

Binomial parameter estimates

### Syntax

```
phat = binofit(x,n)
[phat,pci] = binofit(x,n)
[phat,pci] = binofit(x,n,alpha)
```

### Description

`phat = binofit(x,n)` returns a maximum likelihood estimate of the probability of success in a given binomial trial based on the number of successes, `x`, observed in `n` independent trials. If `x = (x(1), x(2), ... x(k))` is a vector, `binofit` returns a vector of the same size as `x` whose `i`th entry is the parameter estimate for `x(i)`. All `k` estimates are independent of each other. If `n = (n(1), n(2), ..., n(k))` is a vector of the same size as `x`, the binomial fit, `binofit`, returns a vector whose `i`th entry is the parameter estimate based on the number of successes `x(i)` in `n(i)` independent trials. A scalar value for `x` or `n` is expanded to the same size as the other input.

`[phat,pci] = binofit(x,n)` returns the probability estimate, `phat`, and the 95% confidence intervals, `pci`. `binofit` uses the Clopper-Pearson method to calculate confidence intervals.

`[phat,pci] = binofit(x,n,alpha)` returns the  $100(1 - \text{alpha})\%$  confidence intervals. For example, `alpha = 0.01` yields 99% confidence intervals.

---

**Note** `binofit` behaves differently than other Statistics and Machine Learning Toolbox functions that compute parameter estimates, in that it returns independent estimates for each entry of `x`. By comparison, `expfit` returns a single parameter estimate based on all the entries of `x`.

---

Unlike most other distribution fitting functions, the `binofit` function treats its input `x` vector as a collection of measurements from separate samples. If you want to treat `x` as a

single sample and compute a single parameter estimate for it, you can use `binofit(sum(x),sum(n))` when `n` is a vector, and `binofit(sum(X),N*length(X))` when `n` is a scalar.

## Examples

This example generates a binomial sample of 100 elements, where the probability of success in a given trial is 0.6, and then estimates this probability from the outcomes in the sample.

```
r = binornd(100,0.6);
[phat,pci] = binofit(r,100)
phat =
    0.5800
pci =
    0.4771    0.6780
```

The 95% confidence interval, `pci`, contains the true value, 0.6.

## References

- [1] Johnson, N. L., S. Kotz, and A. W. Kemp. *Univariate Discrete Distributions*. Hoboken, NJ: Wiley-Interscience, 1993.

## See Also

`binocdf` | `binoinv` | `binopdf` | `binornd` | `binostat` | `mle`

## Topics

“Binomial Distribution” on page B-9

Introduced before R2006a

## binoinv

Binomial inverse cumulative distribution function

### Syntax

```
X = binoinv(Y,N,P)
```

### Description

`X = binoinv(Y,N,P)` returns the smallest integer `X` such that the binomial cdf evaluated at `X` is equal to or exceeds `Y`. You can think of `Y` as the probability of observing `X` successes in `N` independent trials where `P` is the probability of success in each trial. Each `X` is a positive integer less than or equal to `N`.

`Y`, `N`, and `P` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array with the same dimensions as the other inputs. The parameters in `N` must be positive integers, and the values in both `P` and `Y` must lie on the interval `[0 1]`.

### Examples

If a baseball team has a 50-50 chance of winning any game, what is a reasonable range of games this team might win over a season of 162 games?

```
binoinv([0.05 0.95],162,0.5)
ans =
    71    91
```

This result means that in 90% of baseball seasons, a .500 team should win between 71 and 91 games.



## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`binocdf` | `binofit` | `binopdf` | `binornd` | `binostat` | `icdf`

### Topics

“Binomial Distribution” on page B-9

Introduced before R2006a

## binopdf

Binomial probability density function

### Syntax

```
Y = binopdf(X, N, P)
```

### Description

`Y = binopdf(X, N, P)` computes the binomial pdf at each of the values in `X` using the corresponding number of trials in `N` and probability of success for each trial in `P`. `Y`, `N`, and `P` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array with the same dimensions of the other inputs.

The parameters in `N` must be positive integers, and the values in `P` must lie on the interval  $[0, 1]$ .

The binomial probability density function for a given value  $x$  and given pair of parameters  $n$  and  $p$  is

$$y = f(x | n, p) = \binom{n}{x} p^x q^{(n-x)} I_{(0,1,\dots,n)}(x)$$

where  $q = 1 - p$ . The result,  $y$ , is the probability of observing  $x$  successes in  $n$  independent trials, where the probability of success in any *given* trial is  $p$ . The indicator function  $I_{(0,1,\dots,n)}(x)$  ensures that  $x$  only adopts values of 0, 1, ...,  $n$ .

### Examples

A Quality Assurance inspector tests 200 circuit boards a day. If 2% of the boards have defects, what is the probability that the inspector will find no defective boards on any given day?

```
binopdf(0,200,0.02)
ans =
    0.0176
```

What is the most likely number of defective boards the inspector will find?

```
defects=0:200;
y = binopdf(defects,200,.02);
[x,i]=max(y);
defects(i)
ans =
    4
```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

[binocdf](#) | [binofit](#) | [binoinv](#) | [binornd](#) | [binostat](#) | [pdf](#)

### Topics

“Binomial Distribution” on page B-9

**Introduced before R2006a**

## binornd

Binomial random numbers

### Syntax

```
R = binornd(N,P)
R = binornd(N,P,m,n,...)
R = binornd(N,P,[m,n,...])
```

### Description

`R = binornd(N,P)` generates random numbers from the binomial distribution with parameters specified by the number of trials, `N`, and probability of success for each trial, `P`. `N` and `P` can be vectors, matrices, or multidimensional arrays that have the same size, which is also the size of `R`. A scalar input for `N` or `P` is expanded to a constant array with the same dimensions as the other input.

`R = binornd(N,P,m,n,...)` or `R = binornd(N,P,[m,n,...])` generates an `m`-by-`n`-by-... array containing random numbers from the binomial distribution with parameters `N` and `P`. `N` and `P` can each be scalars or arrays of the same size as `R`.

### Examples

```
n = 10:10:60;

r1 = binornd(n,1./n)
r1 =
     2     1     0     1     1     2

r2 = binornd(n,1./n,[1 6])
r2 =
     0     1     2     1     3     1

r3 = binornd(n,1./n,1,6)
```

```
r3 =  
    0    1    1    1    0    3
```

## Algorithms

The `binornd` function uses the direct method using the definition of the binomial distribution as a sum of Bernoulli random variables.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

The generated code can return a different sequence of numbers than MATLAB if either of the following is true:

- The output is nonscalar.
- An input parameter is invalid for the distribution.

### See Also

`binocdf` | `binofit` | `binoinv` | `binopdf` | `binostat` | `random`

### Topics

“Binomial Distribution” on page B-9

**Introduced before R2006a**

## binostat

Binomial mean and variance

### Syntax

```
[M,V] = binostat(N,P)
```

### Description

`[M,V] = binostat(N,P)` returns the mean of and variance for the binomial distribution with parameters specified by the number of trials,  $N$ , and probability of success for each trial,  $P$ .  $N$  and  $P$  can be vectors, matrices, or multidimensional arrays that have the same size, which is also the size of  $M$  and  $V$ . A scalar input for  $N$  or  $P$  is expanded to a constant array with the same dimensions as the other input.

The mean of the binomial distribution with parameters  $n$  and  $p$  is  $np$ . The variance is  $npq$ , where  $q = 1 - p$ .

### Examples

```
n = logspace(1,5,5)
n =
    10    100   1000  10000 100000
```

```
[m,v] = binostat(n,1./n)
m =
     1     1     1     1     1
v =
    0.9000    0.9900    0.9990    0.9999    1.0000
```

```
[m,v] = binostat(n,1/2)
m =
     5     50     500    5000   50000
v =
    1.0e+04 *
    0.0003    0.0025    0.0250    0.2500    2.5000
```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`binocdf` | `binofit` | `binoinv` | `binopdf` | `binornd`

### Topics

“Binomial Distribution” on page B-9

Introduced before R2006a

## binScatterPlot

Scatter plot of bins for tall arrays

### Syntax

```
binScatterPlot(X, Y)
binScatterPlot(X, Y, nbins)
binScatterPlot(X, Y, Xedges, Yedges)
binScatterPlot(X, Y, Name, Value)
h = binScatterPlot( ___ )
```

### Description

`binScatterPlot(X, Y)` creates a binned scatter plot of the data in `X` and `Y`. The `binScatterPlot` function uses an automatic binning algorithm that returns bins with a uniform area, chosen to cover the range of elements in `X` and `Y` and reveal the underlying shape of the distribution.

`binScatterPlot(X, Y, nbins)` specifies the number of bins to use in each dimension.

`binScatterPlot(X, Y, Xedges, Yedges)` specifies the edges of the bins in each dimension using the vectors `Xedges` and `Yedges`.

`binScatterPlot(X, Y, Name, Value)` specifies additional options with one or more name-value pair arguments using any of the previous syntaxes. For example, you can specify `'Color'` and a valid color option to change the color theme of the plot, or `'Gamma'` with a positive scalar to adjust the level of detail.

`h = binScatterPlot( ___ )` returns a `Histogram2` object. Use this object to inspect properties of the plot.

### Examples

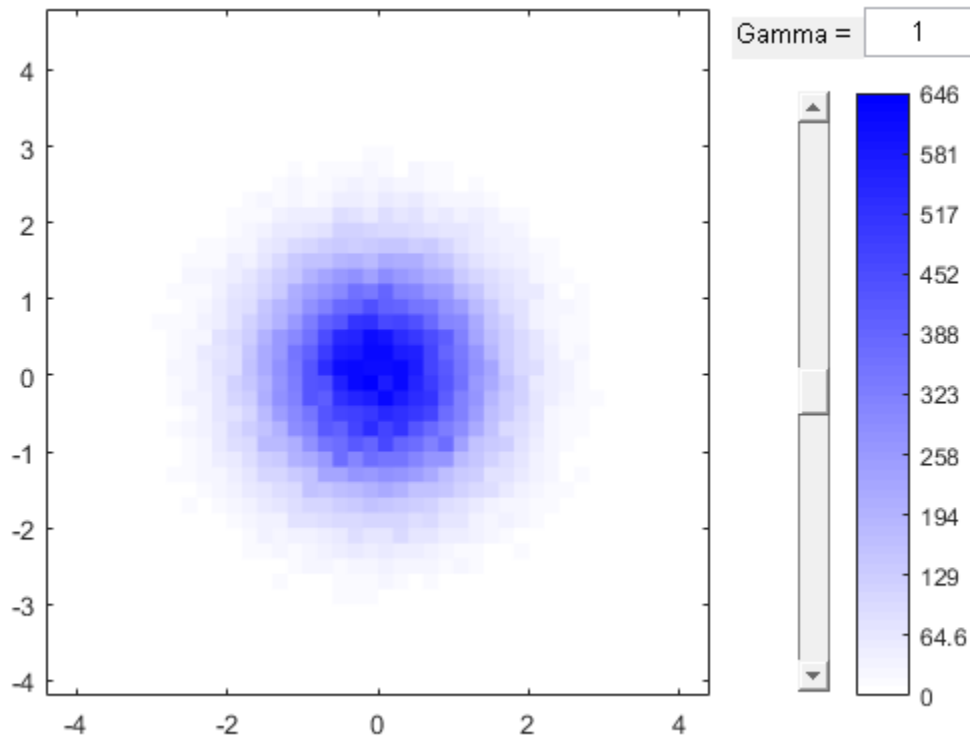


## Binned Scatter Plot of Normally-Distributed Random Data

Create two tall vectors of random data. Create a binned scatter plot for the data.

```
X = tall(randn(1e5,1));  
Y = tall(randn(1e5,1));  
binScatterPlot(X,Y)
```

```
Starting parallel pool (parpool) using the 'local' profile ...  
connected to 12 workers.  
Evaluating tall expression using the Parallel Pool 'local':  
Evaluation completed in 3 sec  
Evaluating tall expression using the Parallel Pool 'local':  
Evaluation completed in 0 sec
```



The resulting figure contains a slider to adjust the level of detail in the image.

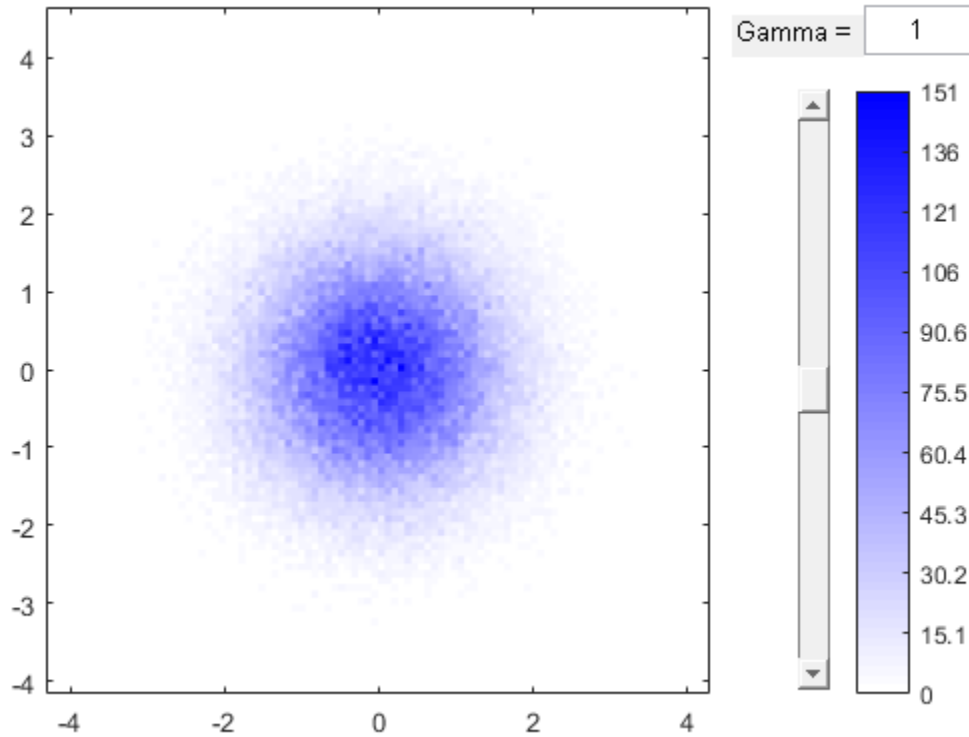
### Specify Number of Scatter Plot Bins

Specify a scalar value as the third input argument to use the same number of bins in each dimension, or a two-element vector to use a different number of bins in each dimension.

Plot a binned scatter plot of random data sorted into 100 bins in each dimension.

```
X = tall(randn(1e5,1));  
Y = tall(randn(1e5,1));  
binScatterPlot(X,Y,100)
```

```
Starting parallel pool (parpool) using the 'local' profile ...  
connected to 12 workers.  
Evaluating tall expression using the Parallel Pool 'local':  
Evaluation completed in 1 sec  
Evaluating tall expression using the Parallel Pool 'local':  
Evaluation completed in 0 sec
```



Use 20 bins in the  $x$ -dimension and continue to use 100 bins in the  $y$ -dimension.

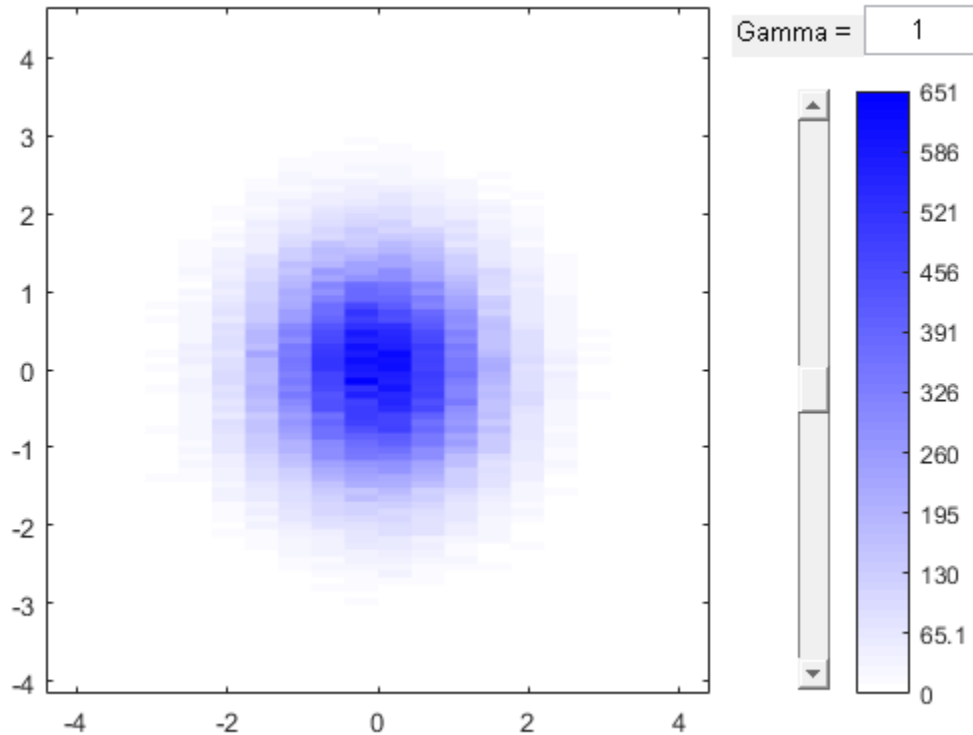
```
binScatterPlot(X,Y,[20 100])
```

```
Evaluating tall expression using the Parallel Pool 'local':
```

```
Evaluation completed in 0 sec
```

```
Evaluating tall expression using the Parallel Pool 'local':
```

```
Evaluation completed in 0 sec
```



### Specify Bin Edges for Scatter Plot

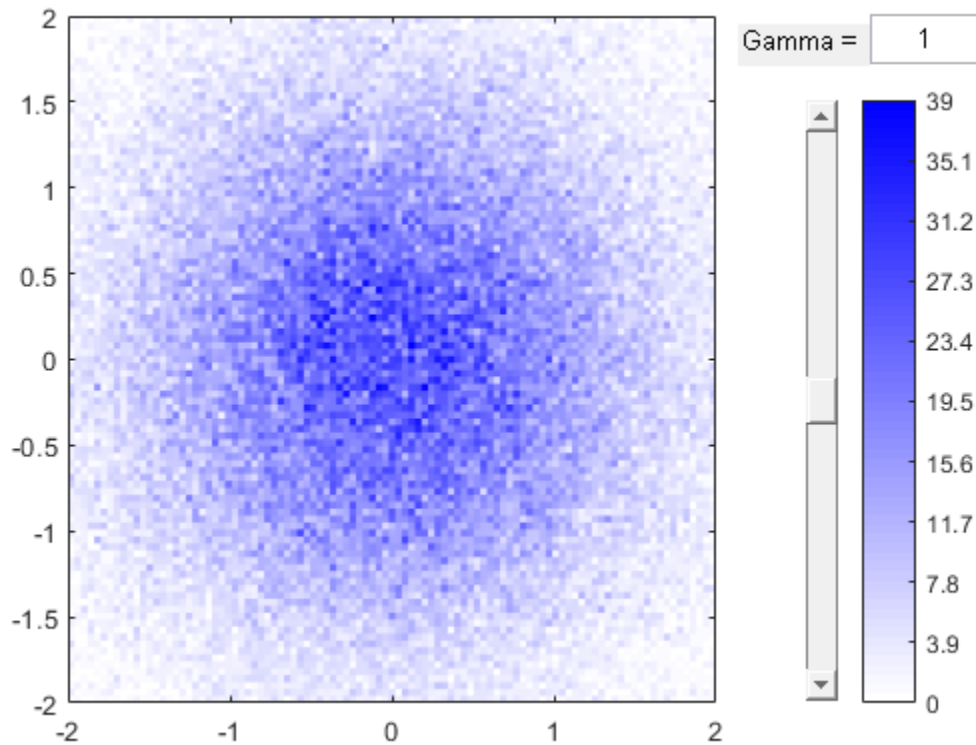
Plot a binned scatter plot of random data with specific bin edges. Use bin edges of  $\text{Inf}$  and  $-\text{Inf}$  to capture outliers.

Create a binned scatter plot with 100 bin edges between  $[-2 \ 2]$  in each dimension. The data outside the specified bin edges is not included in the plot.

```
X = tall(randn(1e5,1));  
Y = tall(randn(1e5,1));  
Xedges = linspace(-2,2);
```

```
Yedges = linspace(-2,2);
binScatterPlot(X,Y,Xedges,Yedges)
```

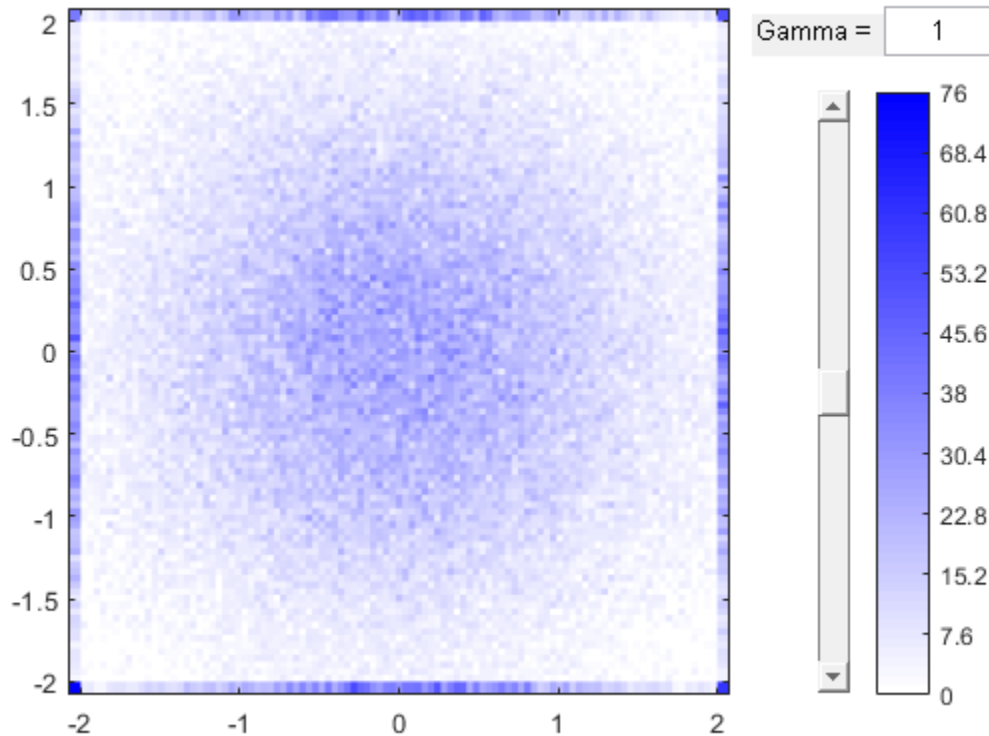
```
Starting parallel pool (parpool) using the 'local' profile ...
connected to 12 workers.
Evaluating tall expression using the Parallel Pool 'local':
Evaluation completed in 1 sec
```



Use coarse bins extending to infinity on the edges of the plot to capture outliers.

```
Xedges = [-Inf linspace(-2,2) Inf];
Yedges = [-Inf linspace(-2,2) Inf];
binScatterPlot(X,Y,Xedges,Yedges)
```

```
Evaluating tall expression using the Parallel Pool 'local':  
Evaluation completed in 0 sec
```



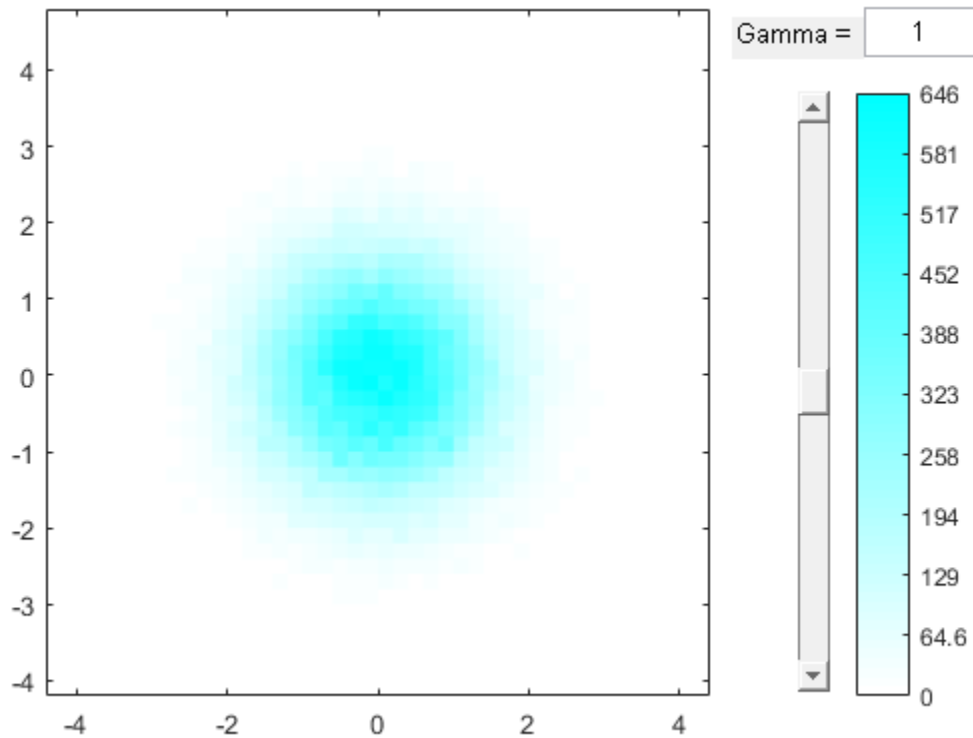
### Adjust Plot Color Theme

Plot a binned scatter plot of random data, specifying 'Color' as 'c'.

```
X = tall(randn(1e5,1));  
Y = tall(randn(1e5,1));  
binScatterPlot(X,Y,'Color','c')
```

```
Starting parallel pool (parpool) using the 'local' profile ...  
connected to 12 workers.
```

```
Evaluating tall expression using the Parallel Pool 'local':  
Evaluation completed in 2 sec  
Evaluating tall expression using the Parallel Pool 'local':  
Evaluation completed in 0 sec
```



## Input Arguments

**x, y** — Data to distribute among bins (as separate arguments)

tall vectors | tall matrices | tall multidimensional arrays

Data to distribute among bins, specified as separate arguments of tall vectors, matrices, or multidimensional arrays.  $X$  and  $Y$  must be the same size. If  $X$  and  $Y$  are not vectors, then `binScatterPlot` treats them as single column vectors,  $X(:)$  and  $Y(:)$ .

Corresponding elements in  $X$  and  $Y$  specify the  $x$  and  $y$  coordinates of 2-D data points,  $[X(k), Y(k)]$ . The underlying data types of  $X$  and  $Y$  can be different, but `binScatterPlot` concatenates these inputs into a single  $N$ -by-2 tall matrix of the dominant underlying data type.

`binScatterPlot` ignores all NaN values. Similarly, `binScatterPlot` ignores Inf and -Inf values, unless the bin edges explicitly specify Inf or -Inf as a bin edge.

---

**Note** If  $X$  or  $Y$  contain integers of type `int64` or `uint64` that are larger than `flintmax`, then it is recommended that you explicitly specify the bin edges. `binScatterPlot` automatically bins the input data using double precision, which lacks integer precision for numbers greater than `flintmax`.

---

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

### **nbins** — Number of bins in each dimension

scalar | vector

Number of bins in each dimension, specified as a positive scalar integer or two-element vector of positive integers. If you do not specify `nbins`, then `binScatterPlot` automatically calculates how many bins to use based on the values in  $X$  and  $Y$ .

- If `nbins` is a scalar, then `binScatterPlot` uses that many bins in each dimension.
- If `nbins` is a vector, then `nbins(1)` specifies the number of bins in the  $x$ -dimension and `nbins(2)` specifies the number of bins in the  $y$ -dimension.

Example: `binScatterPlot(X, Y, 20)` uses 20 bins in each dimension.

Example: `binScatterPlot(X, Y, [10 20])` uses 10 bins in the  $x$ -dimension and 20 bins in the  $y$ -dimension.

### **xedges** — Bin edges in $x$ -dimension

vector



Bin edges in  $x$ -dimension, specified as a vector.  $Xedges(1)$  is the first edge of the first bin in the  $x$ -dimension, and  $Xedges(end)$  is the outer edge of the last bin.

The value  $[X(k), Y(k)]$  is in the  $(i, j)$ th bin if  $Xedges(i) \leq X(k) < Xedges(i+1)$  **and**  $Yedges(j) \leq Y(k) < Yedges(j+1)$ . The last bins in each dimension also include the last (outer) edge. For example,  $[X(k), Y(k)]$  falls into the  $i$ th bin in the last row if  $Xedges(end-1) \leq X(k) \leq Xedges(end)$  **and**  $Yedges(i) \leq Y(k) < Yedges(i+1)$ .

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

### **Yedges** — Bin edges in $y$ -dimension

vector

Bin edges in  $y$ -dimension, specified as a vector.  $Yedges(1)$  is the first edge of the first bin in the  $y$ -dimension, and  $Yedges(end)$  is the outer edge of the last bin.

The value  $[X(k), Y(k)]$  is in the  $(i, j)$ th bin if  $Xedges(i) \leq X(k) < Xedges(i+1)$  **and**  $Yedges(j) \leq Y(k) < Yedges(j+1)$ . The last bins in each dimension also include the last (outer) edge. For example,  $[X(k), Y(k)]$  falls into the  $i$ th bin in the last row if  $Xedges(end-1) \leq X(k) \leq Xedges(end)$  **and**  $Yedges(i) \leq Y(k) < Yedges(i+1)$ .

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

## **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `binScatterPlot(X, Y, 'BinWidth', [5 10])`

### **BinMethod** — Binning algorithm

'auto' (default) | 'scott' | 'integers'

Binning algorithm, specified as the comma-separated pair consisting of 'BinMethod' and one of these values.

Value	Description
'auto'	The default 'auto' algorithm uses a maximum of 100 bins and chooses a bin width to cover the data range and reveal the shape of the underlying distribution.
'scott'	Scott's rule is optimal if the data is close to being jointly normally distributed. This rule is appropriate for most other distributions, as well. It uses a bin size of $[3.5 * \text{std}(X) * \text{numel}(X)^{-1/4}, 3.5 * \text{std}(Y) * \text{numel}(Y)^{-1/4}]$ .
'integers'	The integer rule is useful with integer data, as it creates a bin for each integer. It uses a bin width of 1 and places bin edges halfway between integers. To avoid accidentally creating too many bins, you can use this rule to create a limit of 65536 bins ( $2^{16}$ ). If the data range is greater than 65536, then the integer rule uses wider bins instead.

---

**Note** The `BinMethod` property of the resulting `Histogram2` object always has a value of 'manual'.

---

### **BinWidth** — Width of bins in each dimension

scalar | vector

Width of bins in each dimension, specified as the comma-separated pair consisting of 'BinWidth' and a scalar or two-element vector of positive integers, `[xWidth yWidth]`. A scalar value indicates the same bin width for each dimension.

If you specify `BinWidth`, then `binScatterPlot` can use a maximum of 1024 bins ( $2^{10}$ ) along each dimension. If instead the specified bin width requires more bins, then `binScatterPlot` uses a larger bin width corresponding to the maximum number of bins.

Example: `binScatterPlot(X,Y,'BinWidth',[5 10])` uses bins with size 5 in the x-dimension and size 10 in the y-dimension.

**Color — Plot color theme**

'b' (default) | 'y' | 'm' | 'c' | 'r' | 'g' | 'k'

Plot color theme, specified as the comma-separated pair consisting of 'Color' and one of these options.

Option	Description
'b'	Blue
'm'	Magenta
'c'	Cyan
'r'	Red
'g'	Green
'y'	Yellow
'k'	Black

**Gamma — Gamma correction**

1 (default) | positive scalar

Gamma correction, specified as the comma-separated pair consisting of 'Gamma' and a positive scalar. Use this option to adjust the brightness and color intensity to affect the amount of detail in the image.

- $\text{gamma} < 1$  — As gamma decreases, the shading of bins with smaller bin counts becomes progressively darker, including more detail in the image.
- $\text{gamma} > 1$  — As gamma increases, the shading of bins with smaller bin counts becomes progressively lighter, removing detail from the image.
- The default value of 1 does not apply any correction to the display.

**XBinLimits — Bin limits in x-dimension**

vector

Bin limits in  $x$ -dimension, specified as the comma-separated pair consisting of 'XBinLimits' and a two-element vector, [xbmin, xbmax]. The vector indicates the first and last bin edges in the  $x$ -dimension.

binScatterPlot only plots data that falls within the bin limits inclusively,  
`Data(Data(:,1)>=xbmin & Data(:,1)<=xbmax).`

**YBinLimits** — Bin limits in *y*-dimension

vector

Bin limits in *y*-dimension, specified as the comma-separated pair consisting of 'YBinLimits' and a two-element vector, [ybmin, ybmax]. The vector indicates the first and last bin edges in the *y*-dimension.

binScatterPlot only plots data that falls within the bin limits inclusively, `Data(Data(:,2)>=ybmin & Data(:,2)<=ybmax)`.

## Output Arguments

**h** — Binned scatter plot

histogram2 object

Binned scatter plot, returned as a histogram2 object. For more information, see Histogram2.

## See Also

Introduced in R2016b

# biplot

Biplot

## Syntax

```
biplot(coefs)
h = biplot(coefs, 'Name', Value)
```

## Description

`biplot(coefs)` creates a biplot of the coefficients in the matrix `coefs`. The biplot is 2-D if `coefs` has two columns or 3-D if it has three columns. `coefs` usually contains principal component coefficients created with `pca`, `pcacov`, or factor loadings estimated with `factoran`. The axes in the biplot represent the principal components or latent factors (columns of `coefs`), and the observed variables (rows of `coefs`) are represented as vectors.

A biplot allows you to visualize the magnitude and sign of each variable's contribution to the first two or three principal components, and how each observation is represented in terms of those components.

`biplot` imposes a sign convention, forcing the element with largest magnitude in each column of `coefs` to be positive. This flips some of the vectors in `coefs` to the opposite direction, but often makes the plot easier to read. Interpretation of the plot is unaffected, because changing the sign of a coefficient vector does not change its meaning.

`biplot` scales the scores so that they fit on the plot: It divides each score by the maximum absolute value of all scores, and multiplies by the maximum coefficient length of `coefs`. Then `biplot` changes the sign of score coordinates according to the sign convention for the `coefs`.

`h = biplot(coefs, 'Name', Value)` specifies one or more name/value input pairs and returns a column vector of handles to the graphics objects created by `biplot`. The `h` contains, in order, handles corresponding to variables (line handles, followed by marker

handles, followed by text handles), to observations (if present, marker handles followed by text handles), and to the axis lines.

## Input Arguments

### Name-Value Pair Arguments

#### **Scores**

Plots both `coefs` and the scores in the matrix `scores` in the biplot. `scores` usually contains principal component scores created with `pca` or factor scores estimated with `factoran`. Each observation (row of scores) is represented as a point in the biplot.

#### **VarLabels**

Labels each vector (variable) with the text in the character array or cell array `varlabels`.

#### **ObsLabels**

Uses the text in the character array or cell array `obslabels` as observation names when displaying data cursors.

#### **Positive**

- `'true'` — restricts the biplot to the positive quadrant (in 2-D) or octant (in 3-D).
- `'false'` — makes the biplot over the range  $\pm \max(\text{coefs}(:))$  for all coordinates.

**Default:** `false`

#### **PropertyName**

Specifies optional property name/value pairs for all Primitive Line graphics objects created by `biplot`.

## Examples

## Biplot of Coefficients and Scores

Load the sample data.

```
load carsmall
```

Define the variable matrix and delete the rows with missing values.

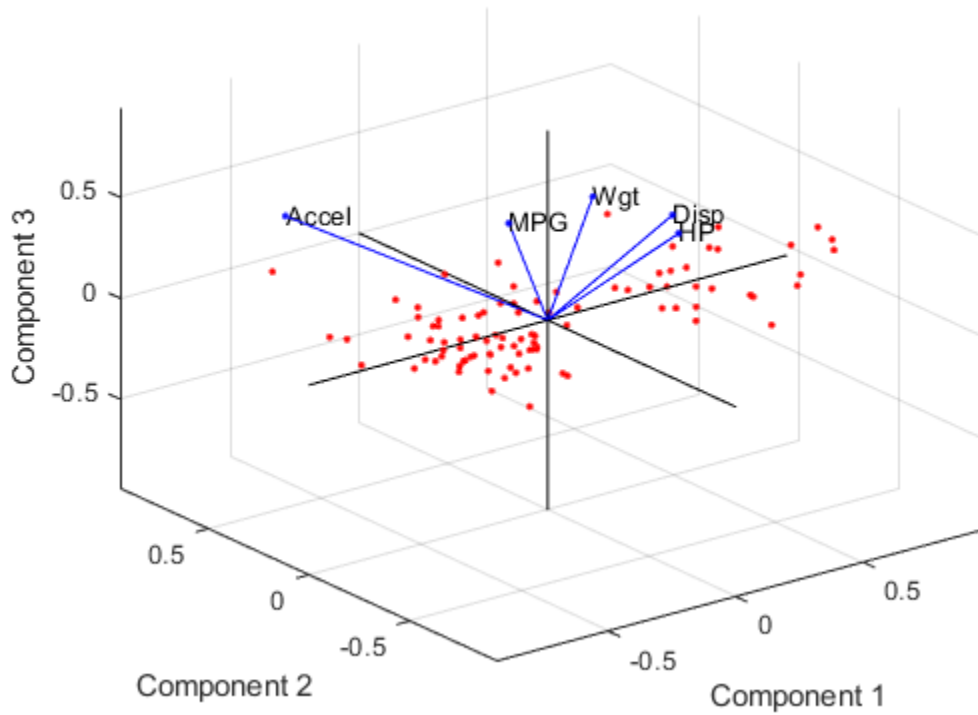
```
x = [Acceleration Displacement Horsepower MPG Weight];  
x = x(all(~isnan(x),2),:);
```

Perform a principal component analysis of the data.

```
[coefs,score] = pca(zscore(x));
```

View the data and the original variables in the space of the first three principal components.

```
vbls = {'Accel','Disp','HP','MPG','Wgt'};  
biplot(coefs(:,1:3),'scores',score(:,1:3),'varlabels',vbls);
```



## See Also

`factoran` | `nmmf` | `pca` | `pcacov` | `rotatefactors`

Introduced before R2006a



# bootci

Bootstrap confidence interval

## Syntax

```
ci = bootci(nboot,bootfun,...)
ci = bootci(nboot,{bootfun,...},'alpha',alpha)
ci = bootci(nboot,{bootfun,...},...,'type',type)
ci = bootci(nboot,
{bootfun,...},...,'type','student','nbootstd',nbootstd)
ci = bootci(nboot,
{bootfun,...},...,'type','student','stderr',stderr)
ci = bootci(nboot,{bootfun,...},...,'Weights',weights)
ci = bootci(nboot,{bootfun,...},...,'Options',options)
[ci,bootstat] = bootci(...)
```

## Description

`ci = bootci(nboot,bootfun,...)` computes the 95% bootstrap confidence interval of the statistic computed by the function `bootfun`. `nboot` is a positive integer indicating the number of bootstrap samples used in the computation. `bootfun` is a function handle specified with `@`. The third and later input arguments to `bootci` are data (scalars, column vectors, or matrices) that are used to create inputs to `bootfun`. `bootci` creates each bootstrap sample by sampling with replacement from the rows of the non-scalar data arguments (these must have the same number of rows). Scalar data are passed to `bootfun` unchanged.

If `bootfun` returns a scalar, `ci` is a vector containing the lower and upper bounds of the confidence interval. If `bootfun` returns a vector of length  $m$ , `ci` is an array of size 2-by- $m$ , where `ci(1,:)` are lower bounds and `ci(2,:)` are upper bounds. If `bootfun` returns an array of size  $m$ -by- $n$ -by- $p$ -by-..., `ci` is an array of size 2-by- $m$ -by- $n$ -by- $p$ -by-..., where `ci(1,:,:,...)` is an array of lower bounds and `ci(2,:,:,...)` is an array of upper bounds.

`ci = bootci(nboot, {bootfun, ...}, 'alpha', alpha)` computes the  $100 \cdot (1 - \alpha)$  bootstrap confidence interval of the statistic defined by the function `bootfun`. `bootfun` and the data that `bootci` passes to it are contained in a single cell array. `alpha` is a scalar between 0 and 1. The default value of `alpha` is 0.05.

`ci = bootci(nboot, {bootfun, ...}, ..., 'type', type)` computes the bootstrap confidence interval of the statistic defined by the function `bootfun`. `type` is the confidence interval type, chosen from among the following:

- 'norm' or 'normal' — Normal approximated interval with bootstrapped bias and standard error.
- 'per' or 'percentile' — Basic percentile method.
- 'cper' or 'corrected percentile' — Bias corrected percentile method.
- 'bca' — Bias corrected and accelerated percentile method. This is the default.
- 'stud' or 'student' — Studentized confidence interval.

`ci = bootci(nboot, {bootfun, ...}, ..., 'type', 'student', 'nbootstd', nbootstd)` computes the studentized bootstrap confidence interval of the statistic defined by the function `bootfun`. The standard error of the bootstrap statistics is estimated using bootstrap, with `nbootstd` bootstrap data samples. `nbootstd` is a positive integer value. The default value of `nbootstd` is 100.

`ci = bootci(nboot, {bootfun, ...}, ..., 'type', 'student', 'stderr', stderr)` computes the studentized bootstrap confidence interval of statistics defined by the function `bootfun`. The standard error of the bootstrap statistics is evaluated by the function `stderr`. `stderr` is a function handle. `stderr` takes the same arguments as `bootfun` and returns the standard error of the statistic computed by `bootfun`.

`ci = bootci(nboot, {bootfun, ...}, ..., 'Weights', weights)` specifies observation weights. `weights` must be a vector of non-negative numbers with at least one positive element. The number of elements in `weights` must be equal to the number of rows in non-scalar input arguments to `bootfun`. To obtain one bootstrap replicate, `bootstrp` samples  $N$  out of  $N$  with replacement using these weights as multinomial sampling probabilities.

`ci = bootci(nboot, {bootfun, ...}, ..., 'Options', options)` specifies options that govern the computation of bootstrap iterations. One option requests that `bootci`

perform bootstrap iterations using multiple processors, if the Parallel Computing Toolbox is available. Two options specify the random number streams to be used in bootstrap resampling. This argument is a struct that you can create with a call to `statset`. You can retrieve values of the individual fields with a call to `statget`. Applicable `statset` parameters are:

- 'UseParallel' — If `true` and if a `parpool` of the Parallel Computing Toolbox is open, compute bootstrap iterations in parallel. If the Parallel Computing Toolbox is not installed, or a `parpool` is not open, computation occurs in serial mode. Default is `false`, or serial computation.
- `UseSubstreams` — Set to `true` to compute in parallel in a reproducible fashion. Default is `false`. To compute reproducibly, set `Streams` to a type allowing substreams: 'mlfg6331\_64' or 'mrg32k3a'.
- `Streams` — A `RandStream` object or cell array of such objects. If you do not specify `Streams`, `bootci` uses the default stream or streams. If you choose to specify `Streams`, use a single object except in the case
  - You have an open Parallel pool
  - `UseParallel` is `true`
  - `UseSubstreams` is `false`

In that case, use a cell array the same size as the Parallel pool.

`[ci,bootstat] = bootci(...)` also returns the bootstrapped statistic computed for each of the `nboot` bootstrap replicate samples. Each row of `bootstat` contains the results of applying `bootfun` to one bootstrap sample. If `bootfun` returns a matrix or array, then this output is converted to a row vector for storage in `bootstat`.

## Examples

Compute the confidence interval for the capability index in statistical process control:

```
y = normrnd(1,1,30,1);           % Simulated process data
LSL = -3; USL = 3;             % Process specifications
capable = @(x)(USL-LSL)./(6* std(x)); % Process capability
ci = bootci(2000, capable, y)   % BCa confidence interval
ci =
    0.8122
    1.2657
```

```
sci = bootci(2000, {capable, y}, 'type', 'student') % Studentized ci
sci =
  0.7739
  1.2707
```

### See Also

`bootstrp` | `jackknife` | `parfor` | `randsample` | `statget` | `statset`

**Introduced in R2006a**

# bootstrap

Bootstrap sampling

## Syntax

```
bootstat = bootstrap(nboot,bootfun,d1,...)
[bootstat,bootsam] = bootstrap(...)
bootstat = bootstrap(...,'Name',Value)
```

## Description

`bootstat = bootstrap(nboot,bootfun,d1,...)` draws `nboot` bootstrap data samples, computes statistics on each sample using `bootfun`, and returns the results in the matrix `bootstat`. `nboot` must be a positive integer. `bootfun` is a function handle specified with `@`. Each row of `bootstat` contains the results of applying `bootfun` to one bootstrap sample. If `bootfun` returns a matrix or array, then this output is converted to a row vector for storage in `bootstat`.

The third and later input arguments (`d1, . . .`) are data (scalars, column vectors, or matrices) used to create inputs to `bootfun`. `bootstrap` creates each bootstrap sample by sampling with replacement from the rows of the non-scalar data arguments (these must have the same number of rows). `bootfun` accepts scalar data unchanged.

`[bootstat,bootsam] = bootstrap(...)` returns an `n`-by-`nboot` matrix of bootstrap indices, `bootsam`. Each column in `bootsam` contains indices of the values that were drawn from the original data sets to constitute the corresponding bootstrap sample. For example, if `d1, . . .` each contain 16 values, and `nboot = 4`, then `bootsam` is a 16-by-4 matrix. The first column contains the indices of the 16 values drawn from `d1, . . .`, for the first of the four bootstrap samples, the second column contains the indices for the second of the four bootstrap samples, and so on. (The bootstrap indices are the same for all input data sets.) To get the output samples `bootsam` without applying a function, set `bootfun` to empty (`[]`).

`bootstat = bootstrap(..., 'Name', Value)` uses additional arguments specified by one or more `Name, Value` pair arguments. The name-value pairs must appear after the data arguments. The available name-value pairs:

- `'Weights'` — Observation weights. The `weights` value must be a vector of nonnegative numbers with at least one positive element. The number of elements in `weights` must be equal to the number of rows in non-scalar input arguments to `bootstrap`. To obtain one bootstrap replicate, `bootstrap` samples  $N$  out of  $N$  with replacement using these weights as multinomial sampling probabilities.
- `'Options'` — The value is a structure that contains options specifying whether to compute bootstrap iterations in parallel, and specifying how to use random numbers during the bootstrap sampling. Create the options structure with `statset`.

Applicable `statset` parameters are:

- `'UseParallel'` — If `true` and if a `parpool` of the Parallel Computing Toolbox is open, compute bootstrap iterations in parallel. If the Parallel Computing Toolbox is not installed, or a `parpool` is not open, computation occurs in serial mode. Default is `false`, meaning serial computation.
- `UseSubstreams` — Set to `true` to compute in parallel in a reproducible fashion. Default is `false`. To compute reproducibly, set `Streams` to a type allowing substreams: `'mlfg6331_64'` or `'mrg32k3a'`.
- `Streams` — A `RandStream` object or cell array of such objects. If you do not specify `Streams`, `bootstrap` uses the default stream or streams. If you choose to specify `Streams`, use a single object except in the case
  - You have an open Parallel pool
  - `UseParallel` is `true`
  - `UseSubstreams` is `false`

In that case, use a cell array the same size as the Parallel pool.

## Examples

### Bootstrapping a Correlation Coefficient Standard Error

This example shows how to compute a correlation coefficient standard error using bootstrap resampling of the sample data.

Load a data set containing the LSAT scores and law-school GPA for 15 students. These 15 data points are resampled to create 1000 different data sets, and the correlation between the two variables is computed for each data set.

```
load lawdata
rng default % For reproducibility
[bootstat,bootsam] = bootstrap(1000,@corr,lsat,gpa);
```

Display the first 5 bootstrapped correlation coefficients.

```
bootstat(1:5,:)
```

```
ans =
```

```
    0.9874
    0.4918
    0.5459
    0.8458
    0.8959
```

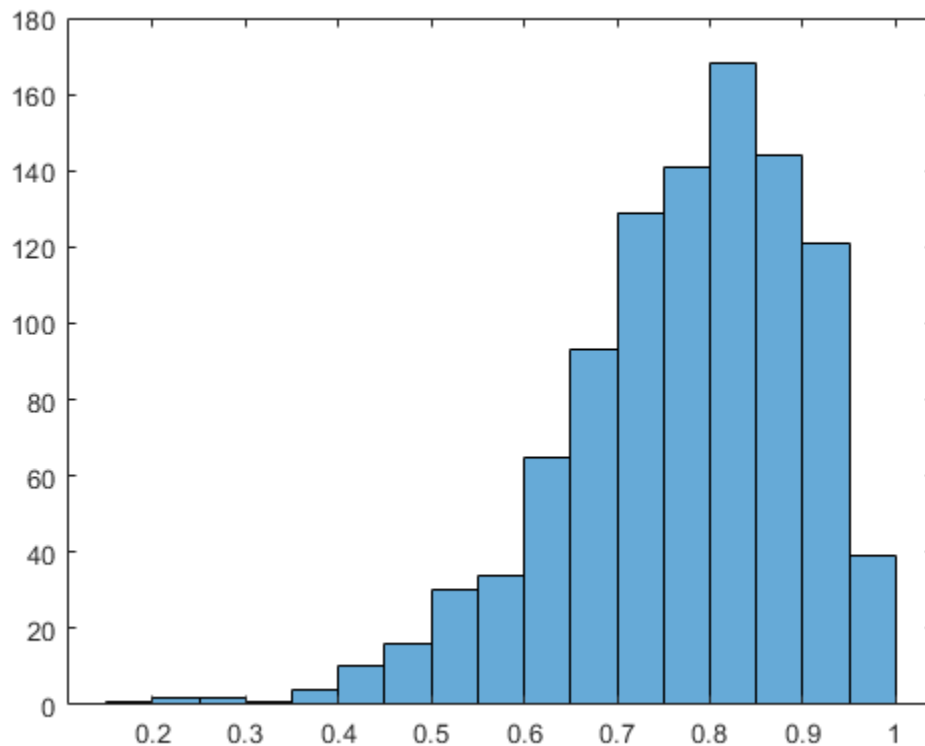
Display the indices of the data selected for the first 5 bootstrap samples.

```
bootsam(:,1:5)
figure
histogram(bootstat)
```

```
ans =
```

```
    13     3    11     8    12
    14     7     1     7     4
     2    14     5    10     8
    14    12     1    11    11
    10    15     2    12    14
     2    10    13     5    15
     5     1    11    11     9
     9    13     5    10     3
    15    15    15     3     3
    15    11     1     2     4
     3    12     7     8    13
    15    12     6    15     4
    15     6    12     6    13
     8    10    12     9     4
```

13 3 3 4 14



The histogram shows the variation of the correlation coefficient across all the bootstrap samples. The sample minimum is positive, indicating that the relationship between LSAT score and GPA is not accidental.

Finally, compute a bootstrap standard of error for the estimated correlation coefficient.

```
se = std(bootstat)
```

```
se =
```



0.1285

### Estimate the Density of Bootstrapped Statistic

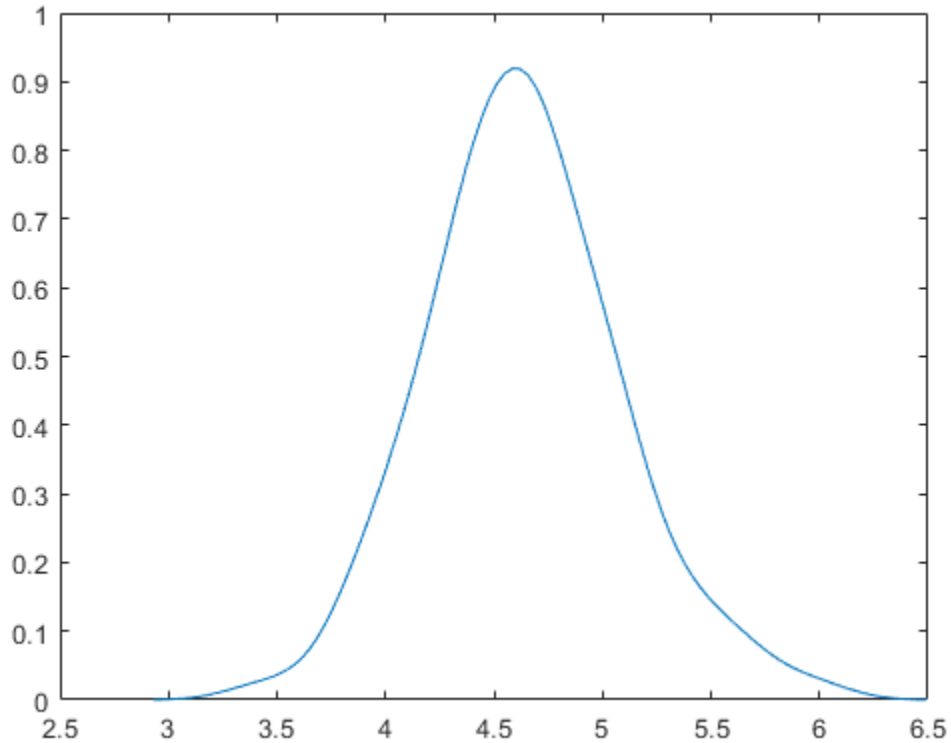
This example shows how to estimate the kernel density of bootstrapped means.

Compute a sample of 100 bootstrapped means of random samples taken from the vector  $Y$ .

```
rng default; % For reproducibility
y = exprnd(5,100,1);
m = bootstrp(100,@mean,y);
```

Plot an estimate of the density of these bootstrapped means.

```
figure;
[fi,xi] = ksdensity(m);
plot(xi,fi);
```



### Bootstrapping Multiple Statistics

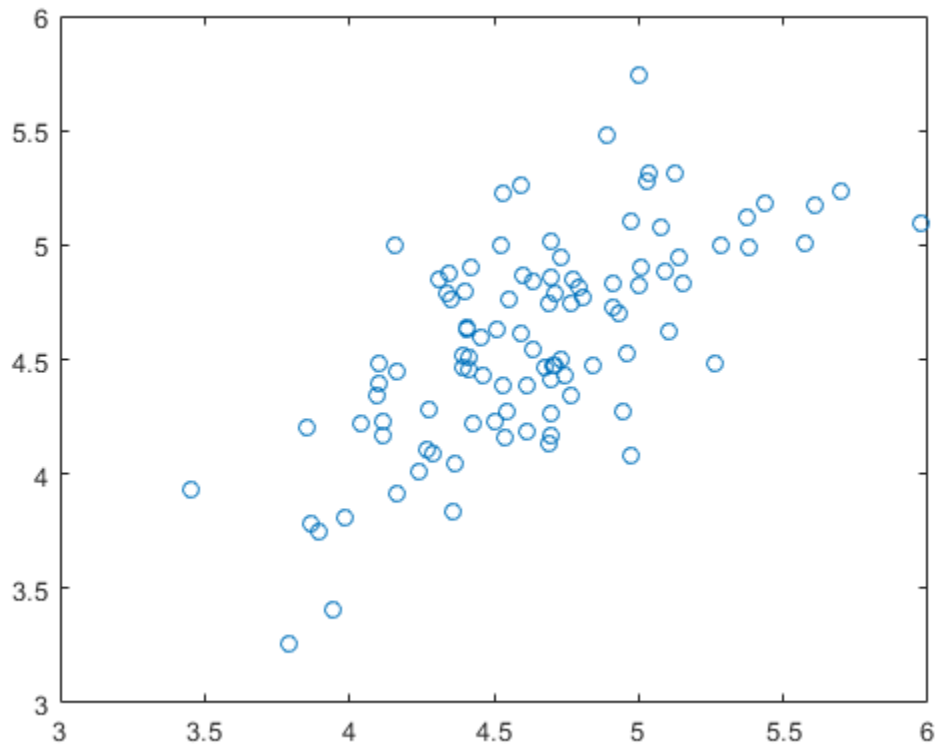
This example shows how to compute and plot the means and standard deviations of bootstrapped 100 samples from a data vector.

Compute a sample of 100 bootstrapped means and standard deviations of random samples taken from the vector  $y$ .

```
rng default % For reproducibility
y = exprnd(5,100,1);
stats = bootstrp(100,@(x) [mean(x) std(x)],y);
```

Plot the bootstrap estimate pairs.

```
plot(stats(:,1),stats(:,2),'o')
```



### Bootstrapping a Regression Model

This example shows how to estimate the standard errors for a coefficient vector in a linear regression by bootstrapping the residuals.

Load the sample data.

```
load hald
```

Perform a linear regression and compute the residuals.

```
x = [ones(size(heat)), ingredients];
y = heat;
b = regress(y,x);
yfit = x*b;
resid = y - yfit;
```

Estimate the standard errors by bootstrapping the residuals.

```
se = std(bootstrp(...
        1000,@(bootr) regress(yfit+bootr,x), resid))
```

```
se =
```

```
    56.1752    0.5940    0.5815    0.5989    0.5691
```

## See Also

[RandStream](#) | [bootci](#) | [histogram](#) | [ksdensity](#) | [parfor](#) | [random](#) | [randsample](#)  
| [statget](#) | [statset](#)

**Introduced before R2006a**

# boxplot

Box plot

## Syntax

```
boxplot(x)
```

```
boxplot(x,g)
```

```
boxplot(ax, ___)
```

```
boxplot( ___, Name, Value)
```

## Description

`boxplot(x)` creates a box plot of the data in `x`. If `x` is a vector, `boxplot` plots one box. If `x` is a matrix, `boxplot` plots one box for each column of `x`.

On each box, the central mark indicates the median, and the bottom and top edges of the box indicate the 25th and 75th percentiles, respectively. The whiskers extend to the most extreme data points not considered outliers, and the outliers are plotted individually using the '+' symbol.

`boxplot(x,g)` creates a box plot using one or more grouping variables contained in `g`. `boxplot` produces a separate box for each set of `x` values that share the same `g` value or values.

`boxplot(ax, ___)` creates a box plot using the axes specified by the axes graphic object `ax`, using any of the previous syntaxes.

`boxplot( ___, Name, Value)` creates a box plot with additional options specified by one or more `Name, Value` pair arguments. For example, you can specify the box style or order.

## Examples

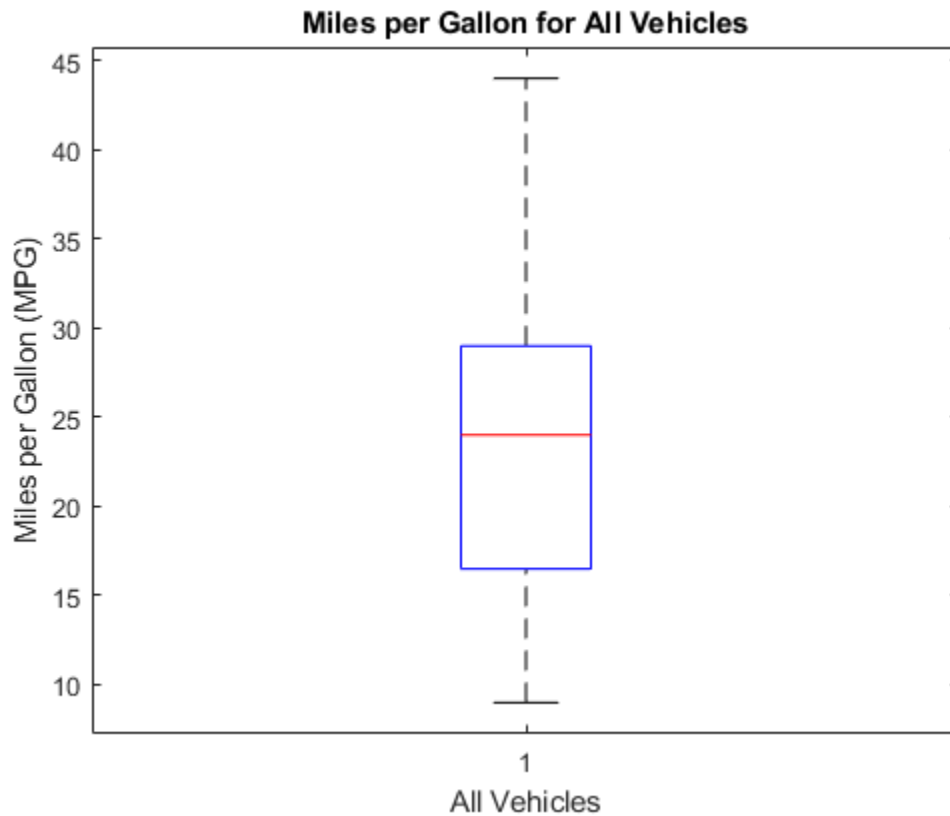
### Create a Box Plot

Load the sample data.

```
load carsmall
```

Create a box plot of the miles per gallon (MPG) measurements. Add a title and label the axes.

```
boxplot(MPG)  
xlabel('All Vehicles')  
ylabel('Miles per Gallon (MPG)')  
title('Miles per Gallon for All Vehicles')
```



The boxplot shows that the median miles per gallon for all vehicles in the sample data is approximately 24. The minimum value is about 9, and the maximum value is about 44.

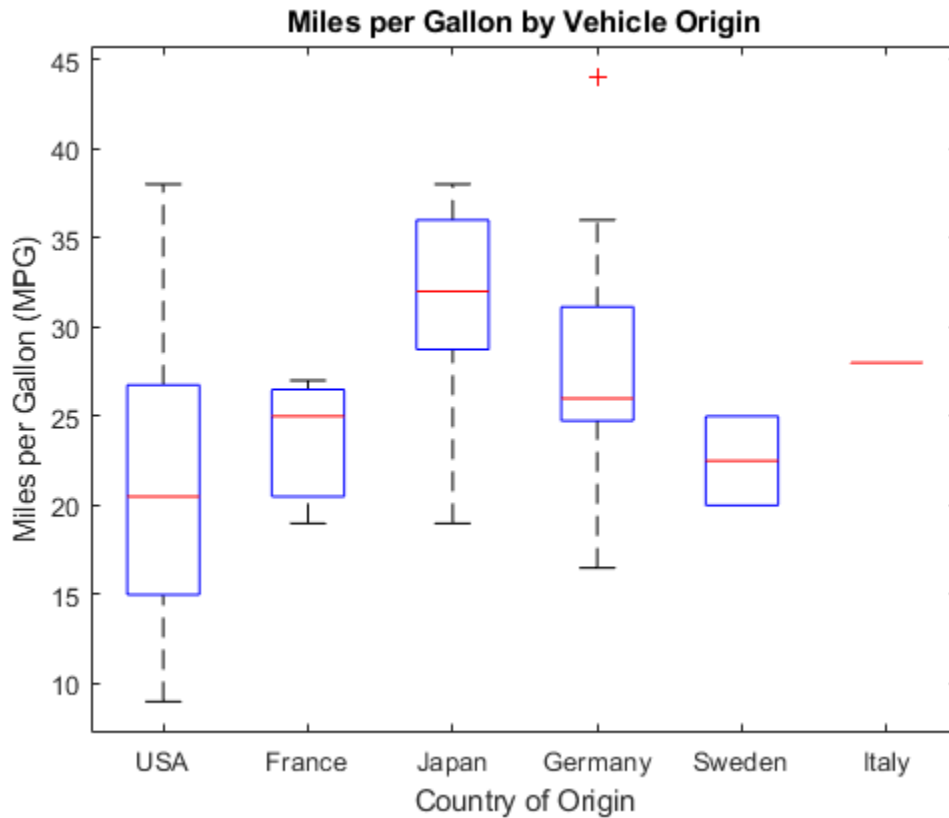
### Create Box Plots for Grouped Data

Load the sample data.

```
load carsmall
```

Create a box plot of the miles per gallon (MPG) measurements from the sample data, grouped by the vehicles' country of origin (`Origin`). Add a title and label the axes.

```
boxplot(MPG,Origin)
title('Miles per Gallon by Vehicle Origin')
xlabel('Country of Origin')
ylabel('Miles per Gallon (MPG)')
```



Each box visually represents the MPG data for cars from the specified country. Italy's "box" appears as a single line because the sample data contains only one observation for this group.



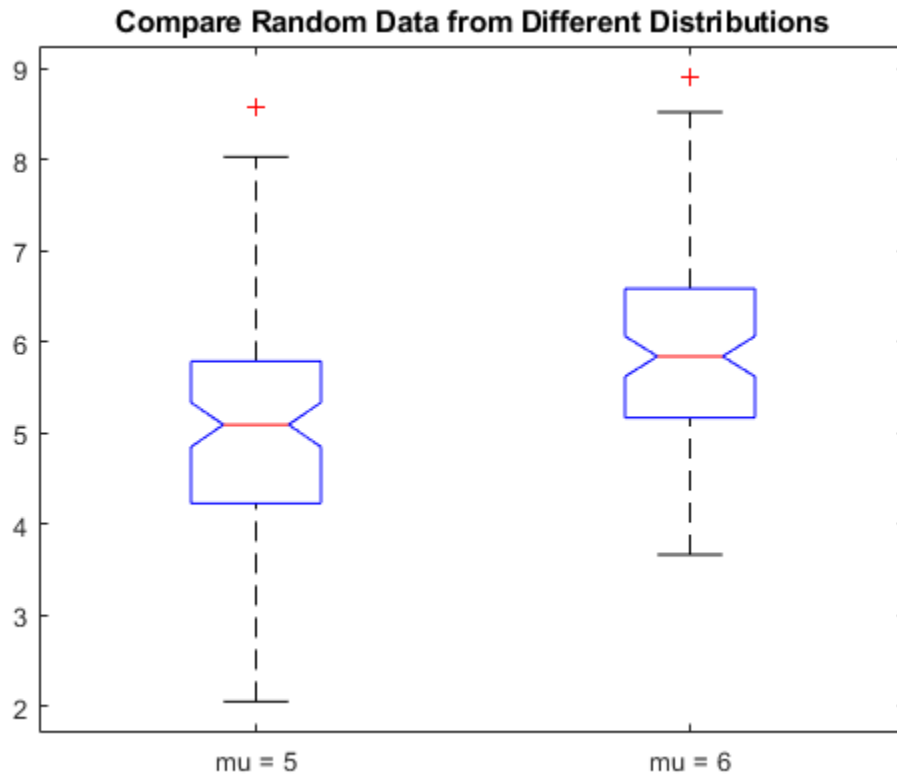
## Create Notched Box Plots

Generate two sets of sample data. The first sample, `x1`, contains random numbers generated from a normal distribution with  $\mu = 5$  and  $\sigma = 1$ . The second sample, `x2`, contains random numbers generated from a normal distribution with  $\mu = 6$  and  $\sigma = 1$ .

```
rng default % For reproducibility
x1 = normrnd(5,1,100,1);
x2 = normrnd(6,1,100,1);
```

Create notched box plots of `x1` and `x2`. Label each box with its corresponding  $\mu$  value.

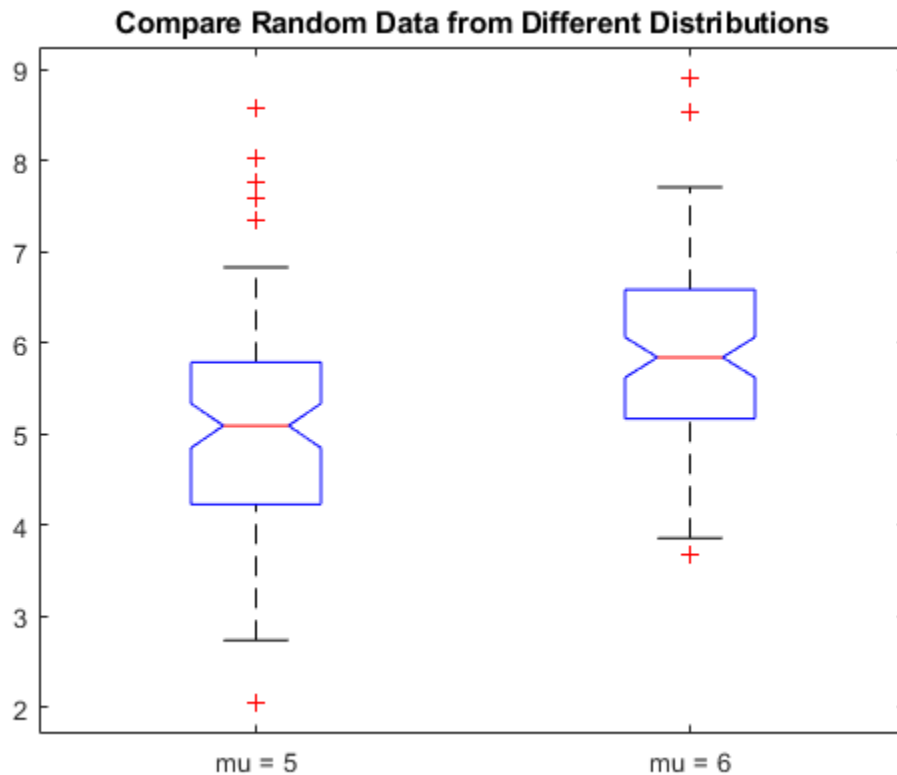
```
figure
boxplot([x1,x2], 'Notch', 'on', 'Labels', {'mu = 5', 'mu = 6'})
title('Compare Random Data from Different Distributions')
```



The boxplot shows that the difference between the medians of the two groups is approximately 1. Since the notches in the box plot do not overlap, you can conclude, with 95% confidence, that the true medians do differ.

The following figure shows the box plot for the same data with the maximum whisker length specified as 1.0 times the interquartile range. Data points beyond the whiskers are displayed using +.

```
figure
boxplot([x1,x2], 'Notch', 'on', 'Labels', {'mu = 5', 'mu = 6'}, 'Whisker', 1)
title('Compare Random Data from Different Distributions')
```



With the smaller whiskers, `boxplot` displays more data points as outliers.

### Create Compact Box Plots

Create a 100-by-25 matrix of random numbers generated from a standard normal distribution to use as sample data.

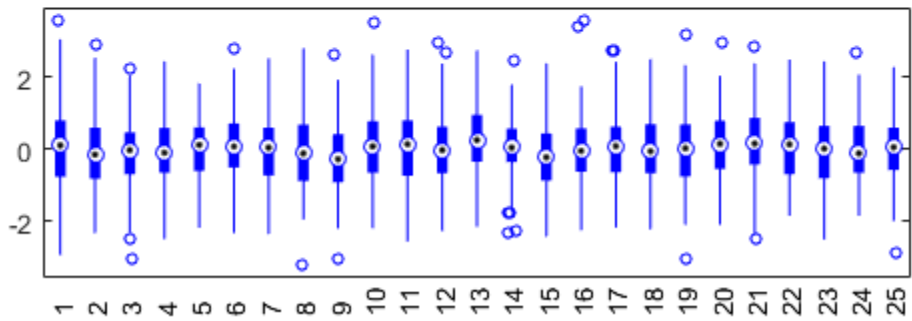
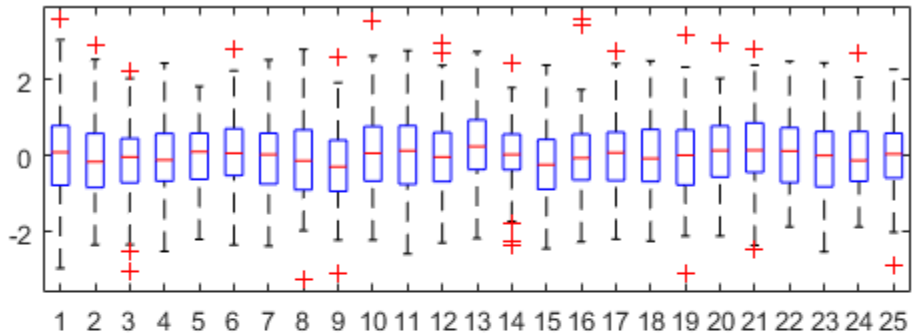
```
rng default % For reproducibility
x = randn(100,25);
```

Create two box plots for the data in `x` on the same figure. Use the default formatting for the top plot, and compact formatting for the bottom plot.

```
figure
```

```
subplot(2,1,1)  
boxplot(x)
```

```
subplot(2,1,2)  
boxplot(x, 'PlotStyle', 'compact')
```



Each plot presents the same data, but the compact formatting may improve readability for plots with many boxes.

## Input Arguments

### **x** — Input data

numeric vector | numeric matrix

Input data, specified as a numeric vector or numeric matrix. If  $x$  is a vector, `boxplot` plots one box. If  $x$  is a matrix, `boxplot` plots one box for each column of  $x$ .

On each box, the central mark indicates the median, and the bottom and top edges of the box indicate the 25th and 75th percentiles, respectively. The whiskers extend to the most extreme data points not considered outliers, and the outliers are plotted individually using the '+' symbol.

Data Types: `single` | `double`

### **g** — Grouping variables

vector | character array | cell array | categorical array

Grouping variables, specified as a vector, character array, cell array, or categorical array. You can specify multiple grouping variables in  $g$  by using a cell array of these variable types or a matrix. If you specify multiple grouping variables, they must all be the same length.

If  $x$  is a vector, then the grouping variables must contain one row for each element of  $x$ . If  $x$  is a matrix, then the grouping variables must contain one row for each column of  $x$ . Groups that contain a missing value (NaN) or an empty character vector in a grouping variable are omitted, and are not counted in the number of groups considered by other parameters.

By default, `boxplot` sorts character grouping variables in the order they initially appear in the data, categorical grouping variables by the order of their levels, and numeric grouping variables in numeric order. To control the order of groups, do one of the following:

- Use categorical variables in  $g$  and specify the order of their levels.
- Use the 'GroupOrder' name-value pair argument.

- Pre-sort your data.

Data Types: `single` | `double` | `char` | `cell` | `categorical`

### **ax** — Axes on which to plot

axes graphic object

Axes on which to plot, specified as an axes graphic object. If you do not specify `ax`, then `boxplot` creates the plot using the current axis. For more information on creating an axes graphic object, see `axes` and `Axes`.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'Notch', 'on', 'Labels', {'mu = 5', 'mu = 6'}` creates a notched box plot and labels the two boxes `mu = 5` and `mu = 6`, from left to right

### Box Appearance

#### **BoxStyle** — Box style

'outline' | 'filled'

Box style, specified as the comma-separated pair consisting of 'BoxStyle' and one of the following.

Name	Value
'outline'	Plot boxes using an unfilled box with dashed whiskers. This is the default if 'PlotStyle' is 'traditional'.
'filled'	Plot boxes using a narrow filled box with lines for whiskers. This is the default if 'PlotStyle' is 'compact'.

Example: `'BoxStyle', 'filled'`

#### **Colors** — Box colors

RGB triplet | character vector of color names

Box colors, specified as the comma-separated pair consisting of `'Colors'` and an RGB triplet or character vector. An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color, respectively. Each intensity must be in the range  $[0,1]$ .

The following table lists the available color characters and their equivalent RGB triplet values.

Long Name	Short Name	RGB Triplet
Yellow	'y'	[1 1 0]
Magenta	'm'	[1 0 1]
Cyan	'c'	[0 1 1]
Red	'r'	[1 0 0]
Green	'g'	[0 1 0]
Blue	'b'	[0 0 1]
White	'w'	[1 1 1]
Black	'k'	[0 0 0]

You can specify multiple colors either as a character vector of color names (for example, `'rgbm'`) or a three-column matrix of RGB values. The sequence is replicated or truncated as required, so for example, `'rb'` gives boxes that alternate red and blue.

If you do not specify the name-value pair `'ColorGroup'`, then `boxplot` uses the same color scheme for all boxes. If you do specify `'ColorGroup'`, then the default is a modified `hsv` colormap.

Example: `'Colors','rgbm'`

### MedianStyle — Median style

`'line' | 'target'`

Median style, specified as the comma-separated pair consisting of `'MedianStyle'` and one of the following.

Name	Value
<code>'line'</code>	Draw a line to represent the median in each box. This is the default when <code>'PlotStyle'</code> is <code>'traditional'</code> .

Name	Value
'target'	Draw a black dot inside a white circle to represent the median in each box. This is the default when 'PlotStyle' is 'compact'.

Example: 'MedianStyle', 'target'

### Notch — Marker for comparison intervals

'off' (default) | 'on' | 'marker'

Marker for comparison intervals, specified as the comma-separated pair consisting of 'Notch' and one of the following.

Name	Value
'off'	Omit comparison intervals from box display.
'on'	If 'PlotStyle' is 'traditional', draw comparison intervals using notches. If 'PlotStyle' is 'compact', draw comparison intervals using triangular markers.
'marker'	Draw comparison intervals using triangular markers.

Two medians are significantly different at the 5% significance level if their intervals do not overlap. `boxplot` represents interval endpoints using the extremes of the notches or the centers of the triangular markers. The notch extremes correspond to  $q_2 - 1.57(q_3 - q_1)/\sqrt{n}$  and  $q_2 + 1.57(q_3 - q_1)/\sqrt{n}$ , where  $q_2$  is the median (50th percentile),  $q_1$  and  $q_3$  are the 25th and 75th percentiles, respectively, and  $n$  is the number of observations without any NaN values. If the sample size is small, the notches might extend beyond the end of the box.

Example: 'Notch', 'on'

### OutlierSize — Marker size for outliers

positive numeric value



Marker size for outliers, specified as the comma-separated pair consisting of 'OutlierSize' and a positive numeric value. The specified value represents the marker size in points.

If 'PlotStyle' is 'traditional', then the default value for OutlierSize is 6. If 'PlotStyle' is 'compact', then the default value for OutlierSize is 4.

Example: 'OutlierSize',8

Data Types: single | double

### **PlotStyle** — Plot style

'traditional' (default) | 'compact'

Plot style, specified as the comma-separated pair consisting of 'PlotStyle' and one of the following.

Name	Value
'traditional'	Plot boxes using a traditional box style.
'compact'	Plot boxes using a smaller box style designed for plots with many groups. This style changes the defaults for some other parameters.

Example: 'PlotStyle', 'compact'

### **Symbol** — Symbol and color for outliers

line specification

Symbol and color for outliers, specified as the comma-separated pair consisting of 'Symbol' and a line specification. See the LineSpec parameter in plot for available line specifications.

If 'PlotStyle' is 'traditional', then the default value is 'r+', which plots each outlier using a red '+' symbol.

If 'PlotStyle' is 'compact', then the default value is 'o', which plots each outlier using an 'o' symbol in the same color as the corresponding box.

If you omit the symbol, then the outliers appear invisible. If you omit the color, then the outliers appear in the same color as the box.

Example: 'kx'

**widths — Box width**

numeric scalar | numeric vector

Box width, specified as the comma-separated pair consisting of 'widths' and a numeric scalar or numeric vector. If the number of boxes is not equal to the number of width values specified, then the list of values is replicated or truncated as necessary.

This name-value pair argument does not alter the spacing between boxes. Therefore, if you specify a large value for 'widths', the boxes might overlap.

The default box width is equal to half of the minimum separation between boxes, which is 0.5 when the 'positions' name-value pair argument takes its default value.

Example: 'widths',0.3

Data Types: single | double

**Group Appearance****ColorGroup — Grouping variable for box color change**

[] (default) | vector | character array | cell array | categorical array

Grouping variable for box color change, specified as the comma-separated pair consisting of 'ColorGroup' and a grouping variable. The grouping variable is a vector, character array, cell array, or categorical array. The box color changes when the specified grouping variable changes. The default value [] indicates that the box color does not change based on the group.

Data Types: single | double | char | cell | categorical

**FactorDirection — Order of factors on plot**

'data' (default) | 'list' | 'auto'

Order of factors on plot, specified as the comma-separated pair consisting of 'FactorDirection' and one of the following.

Name	Value
'data'	Factors appear with the first value next to the plot origin.
'list'	Factors appear left-to-right if on the x-axis, or top-to-bottom if on the y-axis.

Name	Value
'auto'	If the grouping variables are numeric, then <code>boxplot</code> uses 'data'. If the grouping variables are character arrays, cell arrays, or categorical arrays, then <code>boxplot</code> uses 'list'.

### **FullFactors** — Plot all group factors

'off' (default) | 'on'

Plot all group factors, specified as the comma-separated pair consisting of 'FullFactors' and either 'off' or 'on'. If 'off', then `boxplot` plots one box for each unique row of grouping variables. If 'on', then `boxplot` plots one box for each possible combination of grouping variable values, including combinations that do not appear in the data.

Example: 'FullFactors', 'on'

### **FactorGap** — Distance between different grouping factors

[] | positive numeric value | vector of positive numeric values | 'auto'

Distance between different grouping factors, specified as the comma-separated pair consisting of 'FactorGap' and a positive numeric value, a vector of positive numeric values, or 'auto'. If you specify a vector, then the vector length must be less than or equal to the number of grouping variables.

'FactorGap' represents the distance of the gap between different factors of a grouping variable, expressed as a percentage of the width of the plot. For example, if you specify [3, 1], then the gap is three percent of the width of the plot between groups with different values of the first grouping variable, and one percent between groups with the same value of the first grouping variable but different values for the second.

If you specify 'auto', then `boxplot` selects a gap distance automatically. The value [] indicates no change in gap size between different factors.

If 'PlotStyle' is 'traditional', then the default value for `FactorGap` is []. If 'PlotStyle' is 'compact', then the default value is 'auto'.

Example: 'FactorGap', [3,1]

Data Types: single | double

**FactorSeparator — Separation between grouping factors**

[] | positive integer | vector of positive integers | 'auto'

Separation between grouping factors, specified as the comma-separated pair consisting of 'FactorSeparator' and a positive integer or a vector of positive integers, or 'auto'. If you specify a vector, then the length of the vector should be less than or equal to the number of grouping variables. The integer values must be in the range  $[1,G]$ , where  $G$  is the number of grouping variables.

'FactorSeparator' specifies which factors should have their values separated by a grid line. For example, `[1,2]` adds a separator line when the first or second grouping variable changes value.

If 'PlotStyle' is 'traditional', then the default value for FactorSeparator is []. If 'PlotStyle' is 'compact', then the default value is 'auto'.

Example: 'FactorSeparator', [1,2]

Data Types: single | double

**GroupOrder — Plotting order of groups**

[] (default) | cell array

Plotting order of groups, specified as the comma-separated pair consisting of 'GroupOrder' and a cell array containing the names of the grouping variables. If you have multiple grouping variables, separate values with a comma. You can also use categorical arrays as grouping variables to control the order of the boxes. The default value [] does not reorder the boxes.

Data Types: single | double

**Data Limits and Maximum Distances****DataLim — Extreme data limits**

[-Inf, Inf] (default) | two-element numeric vector

Extreme data limits, specified as the comma-separated pair consisting of 'DataLim' and a two-element numeric vector containing the lower and upper limits, respectively. The values specified for 'DataLim' are used by 'ExtremeMode' to determine which data points are extreme.

Data Types: single | double

**ExtremeMode** — Handling method for extreme data`'clip' (default) | 'compress'`

Handling method for extreme data, specified as the comma-separated pair consisting of 'ExtremeMode' and one of the following.

Name	Value
'clip'	If any data values fall outside the limits specified by 'DataLim', then boxplot displays these values at DataLim on the plot.
'compress'	If any data values fall outside the limits specified by 'DataLim', then boxplot displays these values evenly distributed in a region just outside DataLim, retaining the relative order of the points.

If any data points lie outside the limit specified by 'DataLim', then the limit is marked with a dotted line. If any data points are compressed, then two gray lines mark the compression region. Values at  $-\text{Inf}$  or  $\text{Inf}$  can be clipped or compressed, but NaN values do not appear on the plot. Box notches are drawn to scale and may extend beyond the bounds if the median is inside the limit. Box notches are not drawn if the median is outside the limits.

Example: 'ExtremeMode', 'compress'

**Jitter** — Maximum outlier displacement distance

numeric value

Maximum outlier displacement distance, specified as the comma-separated pair consisting of 'Jitter' and a numeric value. Jitter is the maximum distance to displace outliers along the factor axis by a uniform random amount, in order to make duplicate points visible. If you specify 'Jitter' equal to 1, then the jitter regions just touch between the closest adjacent groups.

If 'PlotStyle' is 'traditional', then the default value for Jitter is 0. If 'PlotStyle' is 'compact', then the default value is 0.5.

Example: 'Jitter', 1

Data Types: single | double

**Whisker — Maximum whisker length**

1.5 (default) | positive numeric value

Maximum whisker length, specified as the comma-separated pair consisting of 'Whisker' and a positive numeric value.

boxplot draws points as outliers if they are greater than  $q_3 + w \times (q_3 - q_1)$  or less than  $q_1 - w \times (q_3 - q_1)$ , where  $w$  is the maximum whisker length, and  $q_1$  and  $q_3$  are the 25th and 75th percentiles of the sample data, respectively.

The default value for 'Whisker' corresponds to approximately  $\pm 2.7\sigma$  and 99.3 percent coverage if the data are normally distributed. The plotted whisker extends to the *adjacent value*, which is the most extreme data value that is not an outlier.

Specify 'Whisker' as 0 to give no whiskers and to make every point outside of  $q_1$  and  $q_3$  an outlier.

Example: 'Whisker', 0

Data Types: single | double

**Plot Appearance****Labels — Box labels**

character array | cell array | numeric vector | numeric matrix

Box labels, specified as the comma-separated pair consisting of 'Labels' and a character array, cell array, or numeric vector containing the box label names. Specify one label per  $x$  value or one label per group. To specify multiple label variables, use a numeric matrix or a cell array containing any of the accepted data types.

To remove labels from a plot, use the following command: `set(gca, 'XTickLabel', {''})`.

Data Types: char | cell | single | double

**LabelOrientation — Label orientation**

'inline' | 'horizontal'

Label orientation, specified as the comma-separated pair consisting of 'LabelOrientation' and one of the following.

Name	Value
'inline'	Rotate box labels to be vertical. This is the default when 'PlotStyle' is 'compact'.
'horizontal'	Leave box labels horizontal. This is the default when 'PlotStyle' is 'traditional'.

If the labels are on the  $y$  axis, then both settings leave the labels horizontal.

Example: `'LabelOrientation', 'inline'`

### **LabelVerbosity** — Labels to display on plot

`'all' | 'minor' | 'majorminor'`

Labels to display on plot, specified as the comma-separated pair consisting of LabelVerbosity and one of the following.

Name	Value
'all'	Display a label for every factor. This is the default when 'PlotStyle' is 'traditional'.
'minor'	Display a label for a factor only when that factor has a different value from the previous group.
'majorminor'	Display a label for a factor when that factor or any factor major to it has a different value from the previous group. This is the default when 'PlotStyle' is 'compact'.

Example: `'LabelVerbosity', 'minor'`

### **Orientation** — Plot orientation

`'vertical' (default) | 'horizontal'`

Plot orientation, specified as the comma-separated pair consisting of Orientation and one of the following.

Name	Value
'vertical'	Plot $x$ on the $y$ -axis.

Name	Value
'horizontal'	Plot x on the x-axis.

Example: 'horizontal'

### Positions — Box positions

numeric vector

Box positions, specified as the comma-separated pair consisting of 'Positions' and a numeric vector containing one entry for each group or  $x$  value. The default is  $1:NumGroups$ , where *NumGroups* is the number of groups.

Data Types: single | double

## Tips

- `boxplot` creates a visual representation of the data, but does not return numeric values. To calculate the relevant summary statistics for the sample data, use the following functions:
  - `min` — Find the minimum value in the sample data.
  - `max` — Find the maximum value in the sample data.
  - `median` — Find the median value in the sample data.
  - `quantile` — Find the quantile values in the sample data.
  - `grpstats` — Calculate summary statistics for the sample data, organized by group.
- You can see data values and group names using the data cursor (MATLAB) in the figure window. The cursor shows the original values of any points affected by the `datalim` parameter. You can label the group to which an outlier belongs using the `gname` function.
- To modify graphics properties of a box plot component, use `findobj` with the `Tag` property to find the component's handle. `Tag` values for box plot components depend on parameter settings, and are listed in the following table.



Parameter Settings	Tag Values
All settings	<ul style="list-style-type: none"> <li>'Box'</li> <li>'Outliers'</li> </ul>
When 'PlotStyle' is 'traditional'	<ul style="list-style-type: none"> <li>'Median'</li> <li>'Upper Whisker'</li> <li>'Lower Whisker'</li> <li>'Upper Adjacent Value'</li> <li>'Lower Adjacent Value'</li> </ul>
When 'PlotStyle' is 'compact'	<ul style="list-style-type: none"> <li>'Whisker'</li> <li>'MedianOuter'</li> <li>'MedianInner'</li> </ul>
When 'Notch' is 'marker'	<ul style="list-style-type: none"> <li>'NotchLo'</li> <li>'NotchHi'</li> </ul>

## References

- [1] McGill, R., J. W. Tukey, and W. A. Larsen. “Variations of Boxplots.” *The American Statistician*. Vol. 32, No. 1, 1978, pp. 12–16.
- [2] Velleman, P.F., and D.C. Hoaglin. *Applications, Basics, and Computing of Exploratory Data Analysis*. Pacific Grove, CA: Duxbury Press, 1981.
- [3] Nelson, L. S. “Evaluating Overlapping Confidence Intervals.” *Journal of Quality Technology*. Vol. 21, 1989, pp. 140–141.
- [4] Langford, E. “Quartiles in Elementary Statistics”, *Journal of Statistics Education*. Vol. 14, No. 3, 2006.

## See Also

anova1 | grpstats | kruskalwallis | max | median | min | multcompare | quantile

## Topics

“Grouping Variables” on page 2-59

**Introduced before R2006a**

# boundary

**Class:** `piecewisedistribution`

Piecewise distribution boundaries

## Syntax

```
[p,q] = boundary(obj)
[p,q] = boundary(obj,i)
```

## Description

`[p,q] = boundary(obj)` returns the boundary points between segments of the piecewise distribution object, `obj`. `p` is a vector of cumulative probabilities at each boundary. `q` is a vector of quantiles at each boundary.

`[p,q] = boundary(obj,i)` returns `p` and `q` for the *i*th boundary.

## Examples

Fit Pareto tails to a *t* distribution at cumulative probabilities 0.1 and 0.9:

```
t = trnd(3,100,1);
obj = paretotails(t,0.1,0.9);
[p,q] = boundary(obj)
p =
    0.1000
    0.9000
q =
   -1.7766
    1.8432
```

## See Also

`cdf` | `icdf` | `nsegments` | `paretotails`

# clustering.evaluation.CalinskiHarabaszEvaluation class

**Package:** clustering.evaluation

**Superclasses:**

Calinski-Harabasz criterion clustering evaluation object

## Description

`clustering.evaluation.CalinskiHarabaszEvaluation` is an object consisting of sample data, clustering data, and Calinski-Harabasz criterion values used to evaluate the optimal number of clusters. Create a Calinski-Harabasz criterion clustering evaluation object using `evalclusters`.

## Construction

`eva = evalclusters(x, clust, 'CalinskiHarabasz')` creates a Calinski-Harabasz criterion clustering evaluation object.

`eva = evalclusters(x, clust, 'CalinskiHarabasz', Name, Value)` creates a Calinski-Harabasz criterion clustering evaluation object using additional options specified by one or more name-value pair arguments.

## Input Arguments

**x** — Input data

matrix

Input data, specified as an  $N$ -by- $P$  matrix.  $N$  is the number of observations, and  $P$  is the number of variables.

Data Types: `single` | `double`

**clust** — Clustering algorithm

'kmeans' | 'linkage' | 'gmdistribution' | matrix of clustering solutions | function handle

Clustering algorithm, specified as one of the following.

'kmeans'	Cluster the data in <i>x</i> using the <code>kmeans</code> clustering algorithm, with 'EmptyAction' set to 'singleton' and 'Replicates' set to 5.
'linkage'	Cluster the data in <i>x</i> using the <code>clusterdata</code> agglomerative clustering algorithm, with 'Linkage' set to 'ward'.
'gmdistribution'	Cluster the data in <i>x</i> using the <code>gmdistribution</code> Gaussian mixture distribution algorithm, with 'SharedCov' set to true and 'Replicates' set to 5.

If `Criterion` is 'CalinskiHarabasz', 'DaviesBouldin', or 'silhouette', you can specify a clustering algorithm using a function handle (MATLAB). The function must be of the form `C = clustfun(DATA,K)`, where `DATA` is the data to be clustered, and `K` is the number of clusters. The output of `clustfun` must be one of the following:

- A vector of integers representing the cluster index for each observation in `DATA`. There must be `K` unique values in this vector.
- A numeric  $n$ -by- $K$  matrix of score for  $n$  observations and  $K$  classes. In this case, the cluster index for each observation is determined by taking the largest score value in each row.

If `Criterion` is 'CalinskiHarabasz', 'DaviesBouldin', or 'silhouette', you can also specify `clust` as a  $n$ -by- $K$  matrix containing the proposed clustering solutions.  $n$  is the number of observations in the sample data, and  $K$  is the number of proposed clustering solutions. Column  $j$  contains the cluster indices for each of the  $N$  points in the  $j$ th clustering solution.

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: 'KList', [1:6] specifies to test 1, 2, 3, 4, 5, and 6 clusters to find the optimal number.

**KList** — List of number of clusters to evaluate

vector

List of number of clusters to evaluate, specified as the comma-separated pair consisting of 'KList' and a vector of positive integer values. You must specify KList when clust is a clustering algorithm name or a function handle. When criterion is 'gap', clust must be a character vector or a function handle, and you must specify KList.

Example: 'KList', [1:6]

## Properties

**ClusteringFunction**

Clustering algorithm used to cluster the input data, stored as a valid clustering algorithm name or function handle. If the clustering solutions are provided in the input, ClusteringFunction is empty.

**CriterionName**

Name of the criterion used for clustering evaluation, stored as a valid criterion name.

**CriterionValues**

Criterion values corresponding to each proposed number of clusters in InspectedK, stored as a vector of numerical values.

**Distance**

Distance measure used for clustering data, stored as a valid distance measure name.

**InspectedK**

List of the number of proposed clusters for which to compute criterion values, stored as a vector of positive integer values.

**Missing**

Logical flag for excluded data, stored as a column vector of logical values. If Missing equals true, then the corresponding value in the data matrix  $x$  is not used in the clustering solution.

**NumObservations**

Number of observations in the data matrix  $X$ , minus the number of missing (NaN) values in  $X$ , stored as a positive integer value.

**OptimalK**

Optimal number of clusters, stored as a positive integer value.

**OptimalY**

Optimal clustering solution corresponding to `OptimalK`, stored as a column vector of positive integer values. If the clustering solutions are provided in the input, `OptimalY` is empty.

**x**

Data used for clustering, stored as a matrix of numerical values.

## Methods

### Inherited Methods

<code>addK</code>	Evaluate additional numbers of clusters
<code>plot</code>	Plot clustering evaluation object criterion values
<code>compact</code>	Compact clustering evaluation object

## Examples

### Evaluate the Clustering Solution Using Calinski-Harabasz Criterion

Evaluate the optimal number of clusters using the Calinski-Harabasz clustering evaluation criterion.

Load the sample data.

```
load fisheriris;
```



The data contains length and width measurements from the sepals and petals of three species of iris flowers.

Evaluate the optimal number of clusters using the Calinski-Harabasz criterion. Cluster the data using `kmeans`.

```
rng('default'); % For reproducibility
eva = evalclusters(meas, 'kmeans', 'CalinskiHarabasz', 'KList', [1:6])

eva =

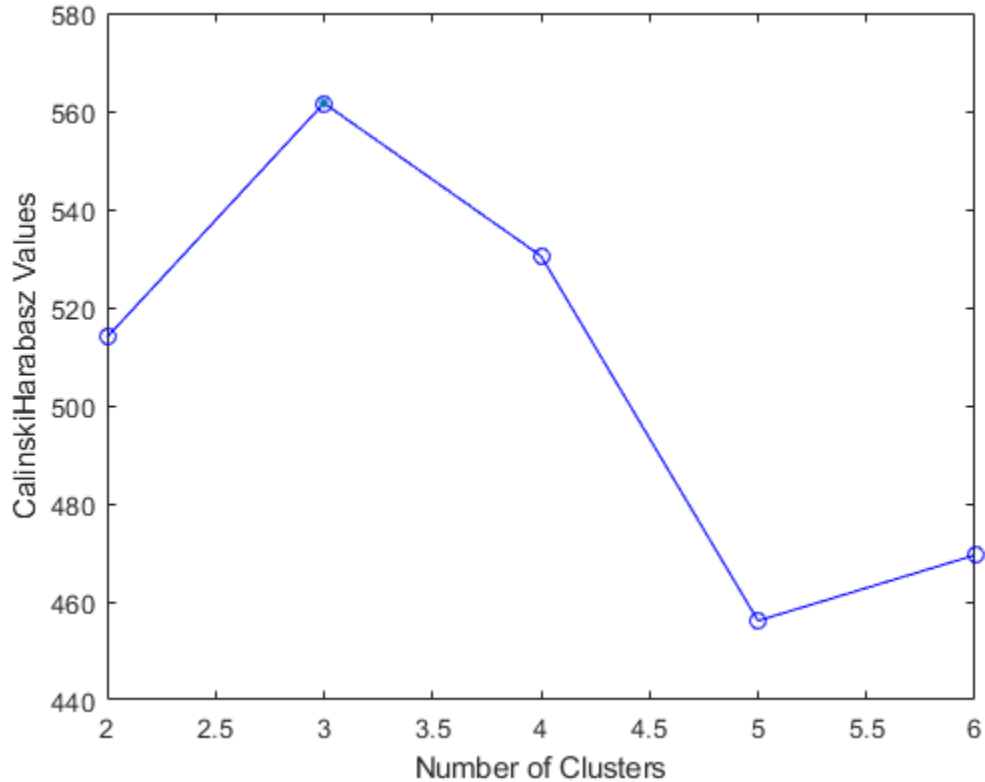
CalinskiHarabaszEvaluation with properties:

    NumObservations: 150
      InspectedK: [1 2 3 4 5 6]
  CriterionValues: [NaN 513.9245 561.6278 530.4871 456.1279 469.5068]
      OptimalK: 3
```

The `OptimalK` value indicates that, based on the Calinski-Harabasz criterion, the optimal number of clusters is three.

Plot the Calinski-Harabasz criterion values for each number of clusters tested.

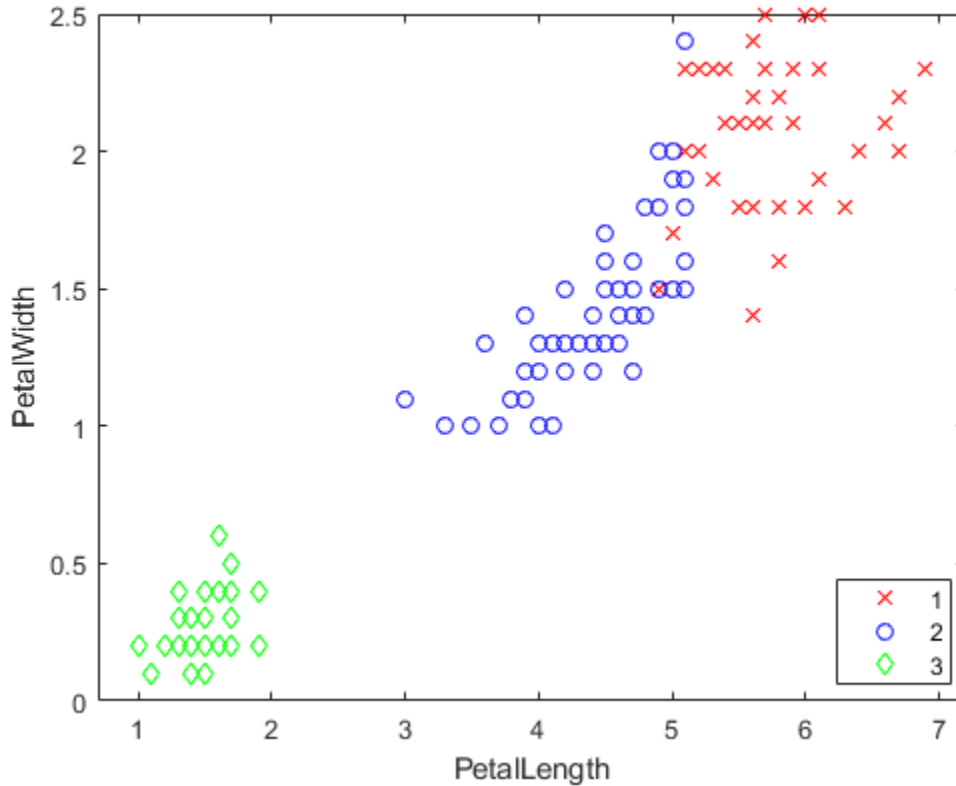
```
figure;
plot(eva);
```



The plot shows that the highest Calinski-Harabasz value occurs at three clusters, suggesting that the optimal number of clusters is three.

Create a grouped scatter plot to examine the relationship between petal length and width. Group the data by suggested clusters.

```
PetalLength = meas(:,3);  
PetalWidth = meas(:,4);  
ClusterGroup = eva.OptimalY;  
figure;  
gscatter(PetalLength,PetalWidth,ClusterGroup,'rbg','xod');
```



The plot shows cluster 1 in the lower-left corner, completely separated from the other two clusters. Cluster 1 contains flowers with the smallest petal widths and lengths. Cluster 3 is in the upper-right corner, and contains flowers with the largest petal widths and

lengths. Cluster 2 is near the center of the plot, and contains flowers with measurements between these two extremes.

## Definitions

### Calinski-Harabasz Criterion

The Calinski-Harabasz criterion is sometimes called the variance ratio criterion (VRC). The Calinski-Harabasz index is defined as

$$VRC_k = \frac{SS_B}{SS_W} \times \frac{(N - k)}{(k - 1)},$$

, where  $SS_B$  is the overall between-cluster variance,  $SS_W$  is the overall within-cluster variance,  $k$  is the number of clusters, and  $N$  is the number of observations.

The overall between-cluster variance  $SS_B$  is defined as

$$SS_B = \sum_{i=1}^k n_i \|m_i - m\|^2,$$

where  $k$  is the number of clusters,  $m_i$  is the centroid of cluster  $i$ ,  $m$  is the overall mean of the sample data, and  $\|m_i - m\|$  is the  $L^2$  norm (Euclidean distance) between the two vectors.

The overall within-cluster variance  $SS_W$  is defined as

$$SS_W = \sum_{i=1}^k \sum_{x \in c_i} \|x - m_i\|^2,$$

where  $k$  is the number of clusters,  $x$  is a data point,  $c_i$  is the  $i$ th cluster,  $m_i$  is the centroid of cluster  $i$ , and  $\|x - m_i\|$  is the  $L^2$  norm (Euclidean distance) between the two vectors.

Well-defined clusters have a large between-cluster variance ( $SS_B$ ) and a small within-cluster variance ( $SS_W$ ). The larger the  $VRC_k$  ratio, the better the data partition. To determine the optimal number of clusters, maximize  $VRC_k$  with respect to  $k$ . The optimal number of clusters is the solution with the highest Calinski-Harabasz index value.

The Calinski-Harabasz criterion is best suited for  $k$ -means clustering solutions with squared Euclidean distances.

## References

- [1] Calinski, T., and J. Harabasz. "A dendrite method for cluster analysis." *Communications in Statistics*. Vol. 3, No. 1, 1974, pp. 1–27.

## See Also

`clustering.evaluation.DaviesBouldinEvaluation` |  
`clustering.evaluation.GapEvaluation` |  
`clustering.evaluation.SilhouetteEvaluation` | `evalclusters`

## Topics

Class Attributes (MATLAB)

Property Attributes (MATLAB)

## candexch

*D*-optimal design from candidate set using row exchanges

### Syntax

```
rlist = candexch(C,nrows)
rlist = candexch(C,nrows,Name,Value)
```

### Description

`rlist = candexch(C,nrows)` uses a row-exchange algorithm to select a *D*-optimal design from the candidate set *C*.

`rlist = candexch(C,nrows,Name,Value)` generates a *D*-optimal design with additional options specified by one or more *Name, Value* pair arguments.

### Input Arguments

**C**

*N*-by-*P* matrix containing the values of *P* model terms at each of *N* points.

**Default:**

**nrows**

The desired number of rows in the design.

### Name-Value Pair Arguments

Specify optional comma-separated pairs of *Name, Value* arguments. *Name* is the argument name and *Value* is the corresponding value. *Name* must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as *Name1, Value1, . . . , NameN, ValueN*.

**display**

When 'on', displays iteration number. Disable the display by setting to 'off'.

**Default:** 'on', except when the `UseParallel` option is true

**init**

nrows-by-P matrix giving an initial design.

**Default:** A random subset of the rows of C

**maxiter**

Maximum number of iterations, a positive integer.

**Default:** 10

**options**

A structure that specifies whether to run in parallel, and specifies the random stream or streams. Create the `options` structure with `statset`. Option fields:

- `UseParallel` — Set to true to compute in parallel. Default is false.
- `UseSubstreams` — Set to true to compute in parallel in a reproducible fashion. Default is false. To compute reproducibly, set `Streams` to a type allowing substreams: 'mlfg6331\_64' or 'mrg32k3a'.
- `Streams` — A `RandStream` object or cell array of such objects. If you do not specify `Streams`, `candexch` uses the default stream or streams. If you choose to specify `Streams`, use a single object except in the case
  - You have an open Parallel pool
  - `UseParallel` is true
  - `UseSubstreams` is false

In that case, use a cell array the same size as the Parallel pool.

**Default:** []

**start**

An  $n$ obs-by- $p$  matrix of factor settings, specifying a set of `nobs` fixed design points to include in the design. `candexch` finds `nrows` additional rows to add to the `start` design. The parameter provides the same functionality as the `daugment` function, using a row-exchange algorithm rather than a coordinate-exchange algorithm.

**Default:** []

**tries**

Number of times to try to generate a design from a new starting point. The algorithm uses random points for each try, except possibly the first.

**Default:** 1

## Output Arguments

**rlist**

Vector of length `nrows` listing the selected rows.

## Examples

This example shows how to generate a  $D$ -optimal design when there is a restriction on the candidate set, so the `rowexch` function isn't appropriate.

```
F = (fullfact([5 5 5])-1)/4; % factor settings in unit cube
T = sum(F,2)<=1.51;        % find rows matching a restriction
F = F(T,:);               % take only those rows
C = [ones(size(F,1),1) F F.^2];
                           % compute model terms including
                           % a constant and all squared terms
R = candexch(C,12);       % find a D-optimal 12-point subset
X = F(R,:);               % get factor settings
```



## Algorithms

`candexch` selects a starting design  $X$  at random, and uses a row-exchange algorithm to iteratively replace rows of  $X$  by rows of  $C$  in an attempt to improve the determinant of  $X' * X$ .

## Alternatives

The `rowexch` function also generates  $D$ -optimal designs using a row-exchange algorithm, but it automatically generates a candidate set that is appropriate for a specified model. The `daugment` function augments a set of fixed design points using a coordinate-exchange algorithm; the `'start'` parameter provides the same functionality using the row exchange algorithm.

## See Also

`candgen` | `cordexch` | `daugment` | `rowexch` | `x2fx`

## Topics

“Specify Candidate Sets” on page 27-21

“D-Optimal Designs” on page 27-15

Introduced before R2006a

## candgen

Candidate set generation

### Syntax

```
dC = candgen(nfactors, 'model')
[dC,C] = candgen(nfactors, 'model')
[...] = candgen(nfactors, 'model', 'Name', value)
```

### Description

`dC = candgen(nfactors, 'model')` generates a candidate set `dC` of treatments appropriate for estimating the parameters in the `model` with `nfactors` factors. `dC` has `nfactors` columns and one row for each candidate treatment. `model` is one of the following:

- 'linear' — Constant and linear terms. This is the default.
- 'interaction' — Constant, linear, and interaction terms
- 'quadratic' — Constant, linear, interaction, and squared terms
- 'purequadratic' — Constant, linear, and squared terms

Alternatively, `model` can be a matrix specifying polynomial terms of arbitrary order. In this case, `model` should have one column for each factor and one row for each term in the model. The entries in any row of `model` are powers for the factors in the columns. For example, if a model has factors `X1`, `X2`, and `X3`, then a row `[0 1 2]` in `model` specifies the term  $(X1.^0) .* (X2.^1) .* (X3.^2)$ . A row of all zeros in `model` specifies a constant term, which can be omitted.

`[dC,C] = candgen(nfactors, 'model')` also returns the design matrix `C` evaluated at the treatments in `dC`. The order of the columns of `C` for a full quadratic model with  $n$  terms is:

- 1 The constant term

- 2 The linear terms in order 1, 2, ...,  $n$
- 3 The interaction terms in order (1, 2), (1, 3), ..., (1,  $n$ ), (2, 3), ..., ( $n - 1$ ,  $n$ )
- 4 The squared terms in order 1, 2, ...,  $n$

Other models use a subset of these terms, in the same order.

Pass `C` to `candexch` to generate a  $D$ -optimal design using a coordinate-exchange algorithm.

`[...] = candgen(nfactors, 'model', 'Name', value)` specifies one or more optional name/value pairs for the design. Valid parameters and their values are listed in the following table. Specify `Name` inside single quotes.

Name	Value
bounds	Lower and upper bounds for each factor, specified as a 2-by- <code>nfactors</code> matrix. Alternatively, this value can be a cell array containing <code>nfactors</code> elements, each element specifying the vector of allowable values for the corresponding factor.
categorical	Indices of categorical predictors.
levels	Vector of number of levels for each factor.

**Note** The `rowexch` function automatically generates a candidate set using `candgen`, and then creates a  $D$ -optimal design from that candidate set using `candexch`. Call `candexch` separately to specify your own candidate set to the row-exchange algorithm.

## Examples

The following example uses `rowexch` to generate a five-run design for a two-factor pure quadratic model using a candidate set that is produced internally:

```
dRE1 = rowexch(2,5,'purequadratic','tries',10)
dRE1 =
    -1     1
     0     0
     1    -1
     1     0
     1     1
```

The same thing can be done using `candgen` and `candexch` in sequence:

```
[dC,C] = candgen(2,'purequadratic') % Candidate set, C
dC =
    -1    -1
     0    -1
     1    -1
    -1     0
     0     0
     1     0
    -1     1
     0     1
     1     1
C =
     1    -1    -1     1     1
     1     0    -1     0     1
     1     1    -1     1     1
     1    -1     0     1     0
     1     0     0     0     0
     1     1     0     1     0
     1    -1     1     1     1
     1     0     1     0     1
     1     1     1     1     1
treatments = candexch(C,5,'tries',10) % Find D-opt subset
treatments =
     2
     1
     7
     3
     4
dRE2 = dC(treatments,:) % Display design
dRE2 =
     0    -1
    -1    -1
    -1     1
     1    -1
    -1     0
```

## See Also

`candexch` | `rowexch`

Introduced before R2006a

## canoncorr

Canonical correlation

### Syntax

```
[A,B] = canoncorr(X,Y)
[A,B,r] = canoncorr(X,Y)
[A,B,r,U,V] = canoncorr(X,Y)
[A,B,r,U,V,stats] = canoncorr(X,Y)
```

### Description

`[A,B] = canoncorr(X,Y)` computes the sample canonical coefficients for the  $n$ -by- $d_1$  and  $n$ -by- $d_2$  data matrices  $X$  and  $Y$ .  $X$  and  $Y$  must have the same number of observations (rows) but can have different numbers of variables (columns).  $A$  and  $B$  are  $d_1$ -by- $d$  and  $d_2$ -by- $d$  matrices, where  $d = \min(\text{rank}(X), \text{rank}(Y))$ . The  $j$ th columns of  $A$  and  $B$  contain the canonical coefficients, i.e., the linear combination of variables making up the  $j$ th canonical variable for  $X$  and  $Y$ , respectively. Columns of  $A$  and  $B$  are scaled to make the covariance matrices of the canonical variables the identity matrix (see  $U$  and  $V$  below). If  $X$  or  $Y$  is less than full rank, `canoncorr` gives a warning and returns zeros in the rows of  $A$  or  $B$  corresponding to dependent columns of  $X$  or  $Y$ .

`[A,B,r] = canoncorr(X,Y)` also returns a 1-by- $d$  vector containing the sample canonical correlations. The  $j$ th element of  $r$  is the correlation between the  $j$ th columns of  $U$  and  $V$  (see below).

`[A,B,r,U,V] = canoncorr(X,Y)` also returns the canonical variables, scores.  $U$  and  $V$  are  $n$ -by- $d$  matrices computed as

```
U = (X-repmat(mean(X),N,1))*A
V = (Y-repmat(mean(Y),N,1))*B
```

`[A,B,r,U,V,stats] = canoncorr(X,Y)` also returns a structure `stats` containing information relating to the sequence of  $d$  null hypotheses  $H_0^{(k)}$ , that the  $(k+1)$ st through

$d$ th correlations are all zero, for  $k = 0 : (d-1)$ . `stats` contains seven fields, each a 1-by- $d$  vector with elements corresponding to the values of  $k$ , as described in the following table:

Field	Description
<code>Wilks</code>	Wilks' lambda (likelihood ratio) statistic
<code>df1</code>	Degrees of freedom for the chi-squared statistic, and the numerator degrees of freedom for the $F$ statistic
<code>df2</code>	Denominator degrees of freedom for the $F$ statistic
<code>F</code>	Rao's approximate $F$ statistic for $H_0^{(k)}$
<code>pF</code>	Right-tail significance level for <code>F</code>
<code>chisq</code>	Bartlett's approximate chi-squared statistic for $H_0^{(k)}$ with Lawley's modification
<code>pChisq</code>	Right-tail significance level for <code>chisq</code>

`stats` has two other fields (`dfe` and `p`) which are equal to `df1` and `pChisq`, respectively, and exist for historical reasons.

## Examples

### Compute Sample Canonical Correlation

Load the sample data.

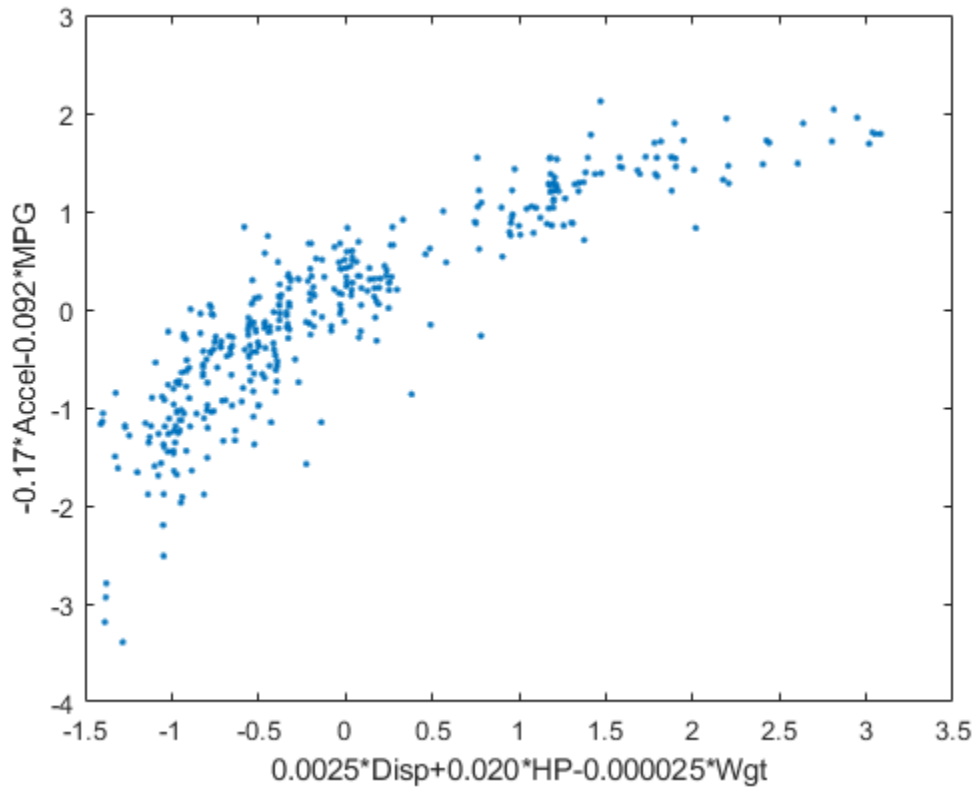
```
load carbig;
X = [Displacement Horsepower Weight Acceleration MPG];
nans = sum(isnan(X),2) > 0;
```

Compute the sample canonical correlation.

```
[A,B,r,U,V] = canoncorr(X(~nans,1:3),X(~nans,4:5));
```

Plot the canonical variables scores.

```
plot(U(:,1),V(:,1),'.')
xlabel('0.0025*Disp+0.020*HP-0.000025*Wgt')
ylabel('-0.17*Accel-0.092*MPG')
```



## References

- [1] Krzanowski, W. J. *Principles of Multivariate Analysis: A User's Perspective*. New York: Oxford University Press, 1988.
- [2] Seber, G. A. F. *Multivariate Observations*. Hoboken, NJ: John Wiley & Sons, Inc., 1984.

## See Also

`manova1` | `pca`

**Introduced before R2006a**



---

# capability

Process capability indices

## Syntax

```
S = capability(data, specs)
```

## Description

`S = capability(data, specs)` estimates capability indices for measurements in `data` given the specifications in `specs`. `data` can be either a vector or a matrix of measurements. If `data` is a matrix, indices are computed for the columns. `specs` can be either a two-element vector of the form `[L, U]` containing lower and upper specification limits, or (if `data` is a matrix) a two-row matrix with the same number of columns as `data`. If there is no lower bound, use `-Inf` as the first element of `specs`. If there is no upper bound, use `Inf` as the second element of `specs`.

The output `S` is a structure with the following fields:

- `mu` — Sample mean
- `sigma` — Sample standard deviation
- `P` — Estimated probability of being within limits
- `P1` — Estimated probability of being below `L`
- `Pu` — Estimated probability of being above `U`
- `Cp` —  $(U-L) / (6 * \text{sigma})$
- `Cpl` —  $(\text{mu}-L) ./ (3 * \text{sigma})$
- `Cpu` —  $(U-\text{mu}) ./ (3 * \text{sigma})$
- `Cpk` —  $\min(Cpl, Cpu)$

Indices are computed under the assumption that data values are independent samples from a normal population with constant mean and variance.

Indices divide a “specification width” (between specification limits) by a “process width” (between control limits). Higher ratios indicate a process with fewer measurements outside of specification.

## Examples

### Compute Capability Indices

Simulate a sample from a process with a mean of 3 and a standard deviation of 0.005.

```
rng default; % for reproducibility
data = normrnd(3,0.005,100,1);
```

Compute capability indices if the process has an upper specification limit of 3.01 and a lower specification limit of 2.99.

```
S = capability(data,[2.99 3.01])
```

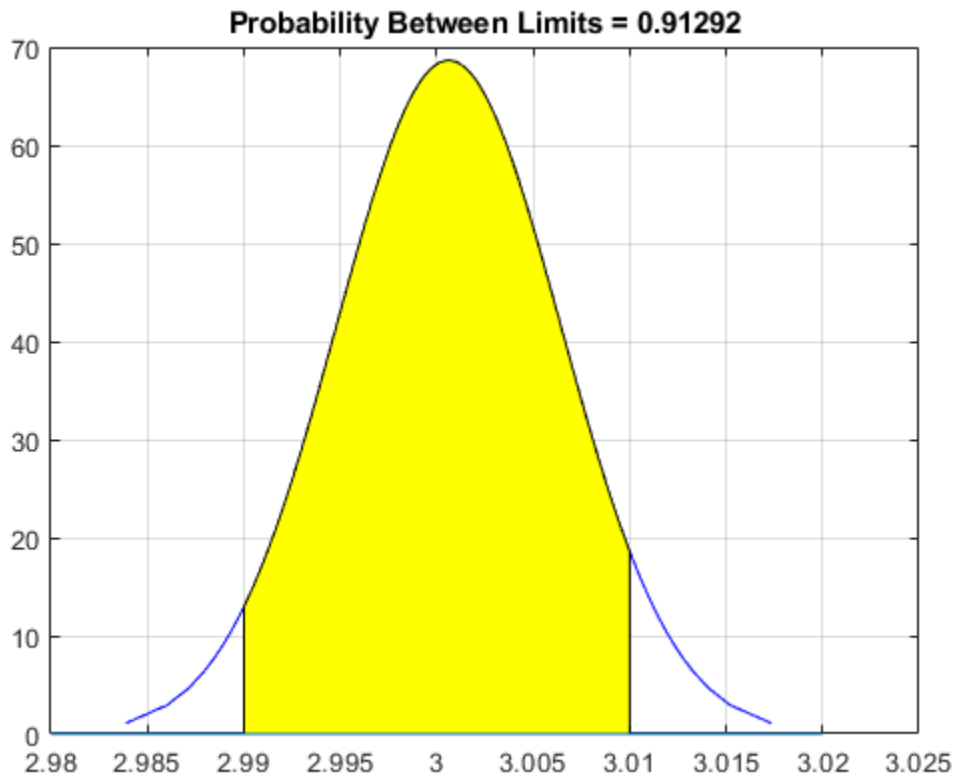
```
S =
```

```
struct with fields:
```

```
mu: 3.0006
sigma: 0.0058
P: 0.9129
Pl: 0.0339
Pu: 0.0532
Cp: 0.5735
Cpl: 0.6088
Cpu: 0.5382
Cpk: 0.5382
```

Visualize the specification and process widths.

```
capaplot(data,[2.99 3.01]);
grid on
```



## References

- [1] Montgomery, D. *Introduction to Statistical Quality Control*. Hoboken, NJ: John Wiley & Sons, 1991, pp. 369–374.

## See Also

`capaplot` | `histfit`

Introduced in R2006b

## capaplot

Process capability plot

### Syntax

```
p = capaplot(data,specs)
[p,h] = capaplot(data,specs)
```

### Description

`p = capaplot(data,specs)` estimates the mean of and variance for the observations in input vector `data`, and plots the pdf of the resulting T distribution. The observations in `data` are assumed to be normally distributed. The output, `p`, is the probability that a new observation from the estimated distribution will fall within the range specified by the two-element vector `specs`. The portion of the distribution between the lower and upper bounds specified in `specs` is shaded in the plot.

`[p,h] = capaplot(data,specs)` additionally returns handles to the plot elements in `h`.

`capaplot` treats NaN values in `data` as missing, and ignores them.

### Examples

#### Create a Process Capability Plot

Randomly generate sample data from a normal process with a mean of 3 and a standard deviation of 0.005.

```
rng default; % For reproducibility
data = normrnd(3,0.005,100,1);
```

Compute capability indices if the process has an upper specification limit of 3.01 and a lower specification limit of 2.99.

```
S = capability(data, [2.99 3.01])
```

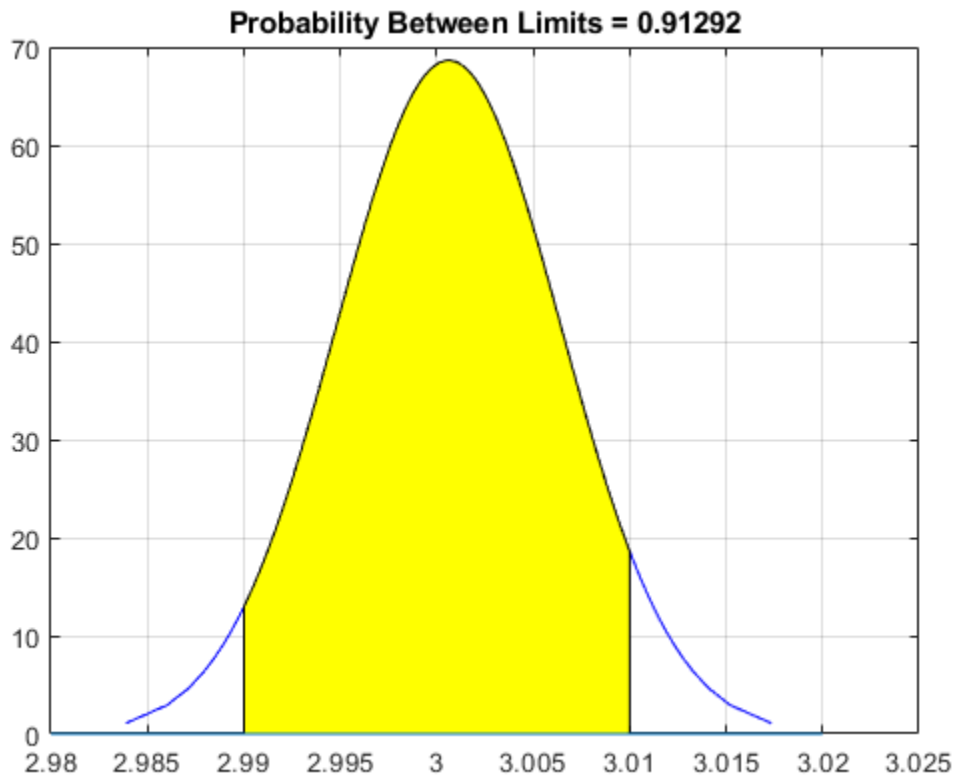
```
S =
```

```
struct with fields:
```

```
    mu: 3.0006  
  sigma: 0.0058  
      P: 0.9129  
    Pl: 0.0339  
    Pu: 0.0532  
    Cp: 0.5735  
    Cpl: 0.6088  
    Cpu: 0.5382  
    Cpk: 0.5382
```

**Visualize the specification and process widths.**

```
capaplot(data, [2.99 3.01]);  
grid on
```



## See Also

`capability` | `histfit`

Introduced before R2006a

## caseread

Read case names from file

### Syntax

```
names = caseread('filename')
names = caseread
```

### Description

`names = caseread('filename')` reads the contents of *filename* and returns a matrix of names. *filename* is the name of a file in the current folder, or the complete path name of any file elsewhere. `caseread` treats each line as a separate case.

`names = caseread` displays the **Select File to Open** dialog box for interactive selection of the input file.

### Examples

Read the file `months.dat` created using the `casewrite` function.

```
type months.dat
```

```
January
February
March
April
May
```

```
names = caseread('months.dat')
names =
January
February
March
April
May
```

## See Also

`casewrite` | `gname` | `tblread` | `tdfread`

**Introduced before R2006a**



## casewrite

Write case names to file

### Syntax

```
casewrite(strmat, 'filename')  
casewrite(strmat)
```

### Description

`casewrite(strmat, 'filename')` writes the contents of matrix `strmat` to `filename`. Each row of `strmat` represents one case name. `filename` is the name of a file in the current folder, or the complete path name of any file elsewhere. `casewrite` writes each name to a separate line in `filename`.

`casewrite(strmat)` displays the **Select File to Write** dialog box for interactive specification of the output file.

### Examples

```
strmat = char('January', 'February', ...  
             'March', 'April', 'May')
```

```
strmat =  
January  
February  
March  
April  
May
```

```
casewrite(strmat, 'months.dat')  
type months.dat
```

```
January  
February  
March
```

April

May

## See Also

`caseread` | `gname` | `tblwrite` | `tdfread`

**Introduced before R2006a**

# clustering.evaluation.DaviesBouldinEvaluation class

**Package:** clustering.evaluation

**Superclasses:**

Davies-Bouldin criterion clustering evaluation object

## Description

`clustering.evaluation.DaviesBouldinEvaluation` is an object consisting of sample data, clustering data, and Davies-Bouldin criterion values used to evaluate the optimal number of clusters. Create a Davies-Bouldin criterion clustering evaluation object using `evalclusters`.

## Construction

`eva = evalclusters(x, clust, 'DaviesBouldin')` creates a Davies-Bouldin criterion clustering evaluation object.

`eva = evalclusters(x, clust, 'DaviesBouldin', Name, Value)` creates a Davies-Bouldin criterion clustering evaluation object using additional options specified by one or more name-value pair arguments.

## Input Arguments

**x** — Input data

matrix

Input data, specified as an  $N$ -by- $P$  matrix.  $N$  is the number of observations, and  $P$  is the number of variables.

Data Types: `single` | `double`

**clust** — Clustering algorithm

'kmeans' | 'linkage' | 'gmdistribution' | matrix of clustering solutions | function handle

Clustering algorithm, specified as one of the following.

'kmeans'	Cluster the data in <i>x</i> using the <code>kmeans</code> clustering algorithm, with 'EmptyAction' set to 'singleton' and 'Replicates' set to 5.
'linkage'	Cluster the data in <i>x</i> using the <code>clusterdata</code> agglomerative clustering algorithm, with 'Linkage' set to 'ward'.
'gmdistribution'	Cluster the data in <i>x</i> using the <code>gmdistribution</code> Gaussian mixture distribution algorithm, with 'SharedCov' set to true and 'Replicates' set to 5.

If `Criterion` is 'CalinskiHarabasz', 'DaviesBouldin', or 'silhouette', you can specify a clustering algorithm using a function handle (MATLAB). The function must be of the form `C = clustfun(DATA,K)`, where `DATA` is the data to be clustered, and `K` is the number of clusters. The output of `clustfun` must be one of the following:

- A vector of integers representing the cluster index for each observation in `DATA`. There must be `K` unique values in this vector.
- A numeric  $n$ -by- $K$  matrix of score for  $n$  observations and  $K$  classes. In this case, the cluster index for each observation is determined by taking the largest score value in each row.

If `Criterion` is 'CalinskiHarabasz', 'DaviesBouldin', or 'silhouette', you can also specify `clust` as a  $n$ -by- $K$  matrix containing the proposed clustering solutions.  $n$  is the number of observations in the sample data, and  $K$  is the number of proposed clustering solutions. Column  $j$  contains the cluster indices for each of the  $N$  points in the  $j$ th clustering solution.

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'KList', [1:5]` specifies to test 1, 2, 3, 4, and 5 clusters to find the optimal number.

**KList** — List of number of clusters to evaluate

vector

List of number of clusters to evaluate, specified as the comma-separated pair consisting of 'KList' and a vector of positive integer values. You must specify KList when clust is a clustering algorithm name or a function handle. When criterion is 'gap', clust must be a character vector or a function handle, and you must specify KList.

Example: 'KList', [1:6]

## Properties

**ClusteringFunction**

Clustering algorithm used to cluster the input data, stored as a valid clustering algorithm name or function handle. If the clustering solutions are provided in the input, ClusteringFunction is empty.

**CriterionName**

Name of the criterion used for clustering evaluation, stored as a valid criterion name.

**CriterionValues**

Criterion values corresponding to each proposed number of clusters in InspectedK, stored as a vector of numerical values.

**InspectedK**

List of the number of proposed clusters for which to compute criterion values, stored as a vector of positive integer values.

**Missing**

Logical flag for excluded data, stored as a column vector of logical values. If Missing equals true, then the corresponding value in the data matrix x is not used in the clustering solution.

**NumObservations**

Number of observations in the data matrix  $X$ , minus the number of missing (NaN) values in  $X$ , stored as a positive integer value.

**OptimalK**

Optimal number of clusters, stored as a positive integer value.

**OptimalY**

Optimal clustering solution corresponding to `OptimalK`, stored as a column vector of positive integer values. If the clustering solutions are provided in the input, `OptimalY` is empty.

**x**

Data used for clustering, stored as a matrix of numerical values.

## Methods

### Inherited Methods

<code>addK</code>	Evaluate additional numbers of clusters
<code>plot</code>	Plot clustering evaluation object criterion values
<code>compact</code>	Compact clustering evaluation object

## Examples

### Evaluate the Clustering Solution Using Davies-Bouldin Criterion

Evaluate the optimal number of clusters using the Davies-Bouldin clustering evaluation criterion.

Generate sample data containing random numbers from three multivariate distributions with different parameter values.

```
rng('default'); % For reproducibility
mu1 = [2 2];
sigma1 = [0.9 -0.0255; -0.0255 0.9];

mu2 = [5 5];
sigma2 = [0.5 0 ; 0 0.3];

mu3 = [-2, -2];
sigma3 = [1 0 ; 0 0.9];

N = 200;

X = [mvnrnd(mu1, sigma1, N); ...
     mvnrnd(mu2, sigma2, N); ...
     mvnrnd(mu3, sigma3, N)];
```

Evaluate the optimal number of clusters using the Davies-Bouldin criterion. Cluster the data using `kmeans`.

```
E = evalclusters(X, 'kmeans', 'DaviesBouldin', 'klist', [1:6])
```

```
E =
```

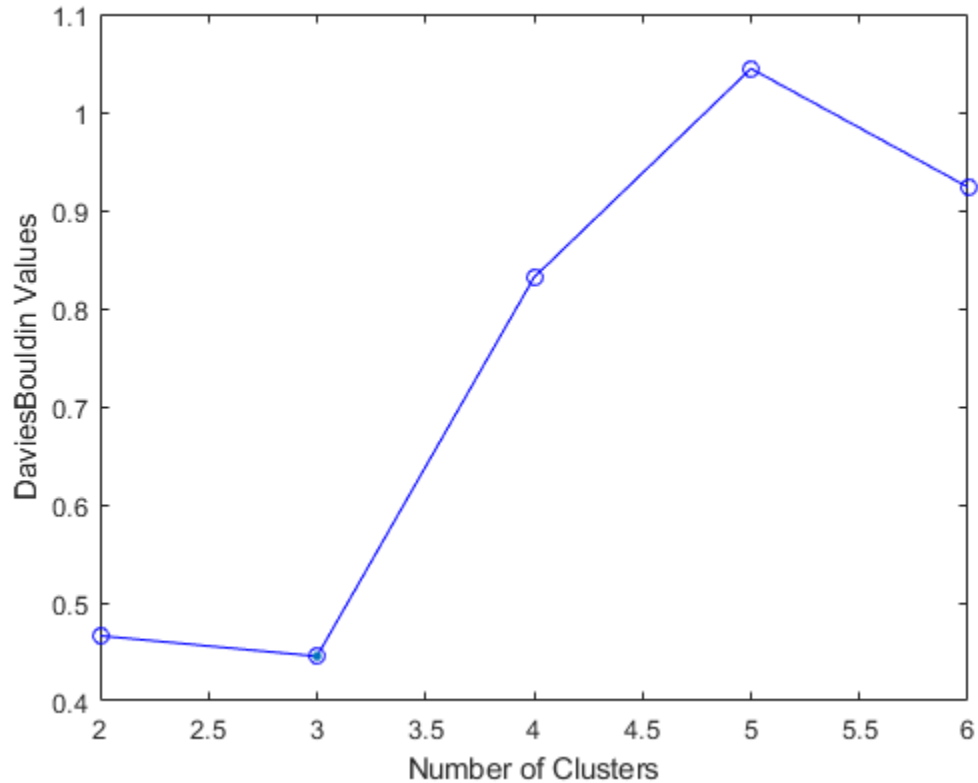
```
DaviesBouldinEvaluation with properties:
```

```
NumObservations: 600
  InspectedK: [1 2 3 4 5 6]
 CriterionValues: [NaN 0.4663 0.4454 0.8316 1.0444 0.9236]
   OptimalK: 3
```

The `OptimalK` value indicates that, based on the Davies-Bouldin criterion, the optimal number of clusters is three.

Plot the Davies-Bouldin criterion values for each number of clusters tested.

```
figure;
plot(E)
```

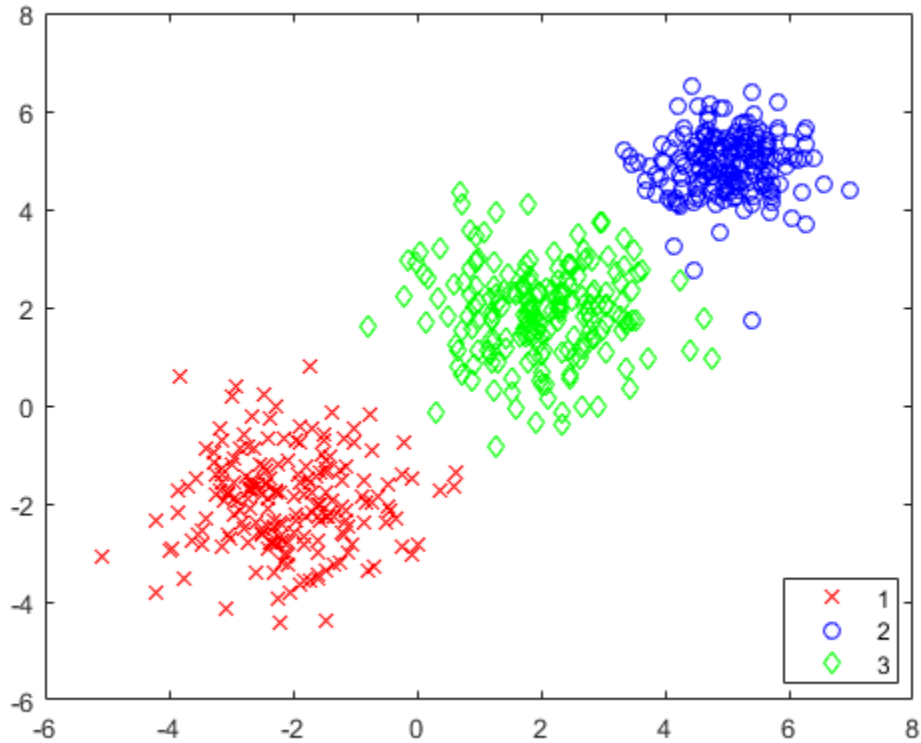


The plot shows that the lowest Davies-Bouldin value occurs at three clusters, suggesting that the optimal number of clusters is three.

Create a grouped scatter plot to visually examine the suggested clusters.

```
figure;  
gscatter(X(:,1),X(:,2),E.OptimalY,'rbg','xod')
```





The plot shows three distinct clusters within the data: Cluster 1 is in the lower-left corner, cluster 2 is near the center of the plot, and cluster 3 is in the upper-right corner.

## Definitions

### Davies-Bouldin Criterion

The Davies-Bouldin criterion is based on a ratio of within-cluster and between-cluster distances. The Davies-Bouldin index is defined as

$$DB = \frac{1}{k} \sum_{i=1}^k \max_{j \neq i} \{D_{i,j}\},$$

where  $D_{i,j}$  is the within-to-between cluster distance ratio for the  $i$ th and  $j$ th clusters. In mathematical terms,

$$D_{i,j} = \frac{(\bar{d}_i + \bar{d}_j)}{d_{i,j}}.$$

$\bar{d}_i$  is the average distance between each point in the  $i$ th cluster and the centroid of the  $i$ th cluster.  $\bar{d}_j$  is the average distance between each point in the  $j$ th cluster and the centroid of the  $j$ th cluster.  $d_{i,j}$  is the Euclidean distance between the centroids of the  $i$ th and  $j$ th clusters.

The maximum value of  $D_{i,j}$  represents the worst-case within-to-between cluster ratio for cluster  $i$ . The optimal clustering solution has the smallest Davies-Bouldin index value.

## References

- [1] Davies, D. L., and D. W. Bouldin. "A Cluster Separation Measure." *IEEE Transactions on Pattern Analysis and Machine Intelligence*. Vol. PAMI-1, No. 2, 1979, pp. 224–227.

## See Also

`clustering.evaluation.CalinskiHarabaszEvaluation` |  
`clustering.evaluation.GapEvaluation` |  
`clustering.evaluation.SilhouetteEvaluation` | `evalclusters`

## Topics

Class Attributes (MATLAB)  
Property Attributes (MATLAB)

# cat

**Class:** dataset

Concatenate dataset arrays

---

**Note** The `dataset` data type might be removed in a future release. To work with heterogeneous data, use the MATLAB® `table` data type instead. See MATLAB `table` documentation for more information.

---

## Syntax

```
ds = cat(dim, ds1, ds2, ...)
```

## Description

`ds = cat(dim, ds1, ds2, ...)` concatenates the dataset arrays `ds1`, `ds2`, ... along dimension `dim` by calling the `dataset/horzcat` or `dataset/vertcat` method. `dim` must be 1 or 2.

## See Also

`horzcat` | `vertcat`

## catsplit

**Class:** classregtree

Categorical splits used for branches in decision tree

---

**Note** `classregtree` will be removed in a future release. See `fitctree`, `fitrtree`, `ClassificationTree`, or `RegressionTree` instead.

---

## Syntax

```
v=catsplit(t)
v=catsplit(t,j)
```

## Description

`v=catsplit(t)` returns an  $n$ -by-2 cell array `v`. Each row in `v` gives left and right values for a categorical split. For each branch node  $j$  based on a categorical predictor variable  $z$ , the left child is chosen if  $z$  is in  $v(j,1)$  and the right child is chosen if  $z$  is in  $v(j,2)$ . The splits are in the same order as nodes of the tree. Nodes for these splits can be found by running `cuttype` and selecting 'categorical' cuts from top to bottom.

`v=catsplit(t,j)` takes an array `j` of rows and returns the splits for the specified rows.

## See Also

`classregtree`

## cdf

**Class:** `gmdistribution`

Cumulative distribution function for Gaussian mixture distribution

## Syntax

```
y = cdf(obj,X)
```

## Description

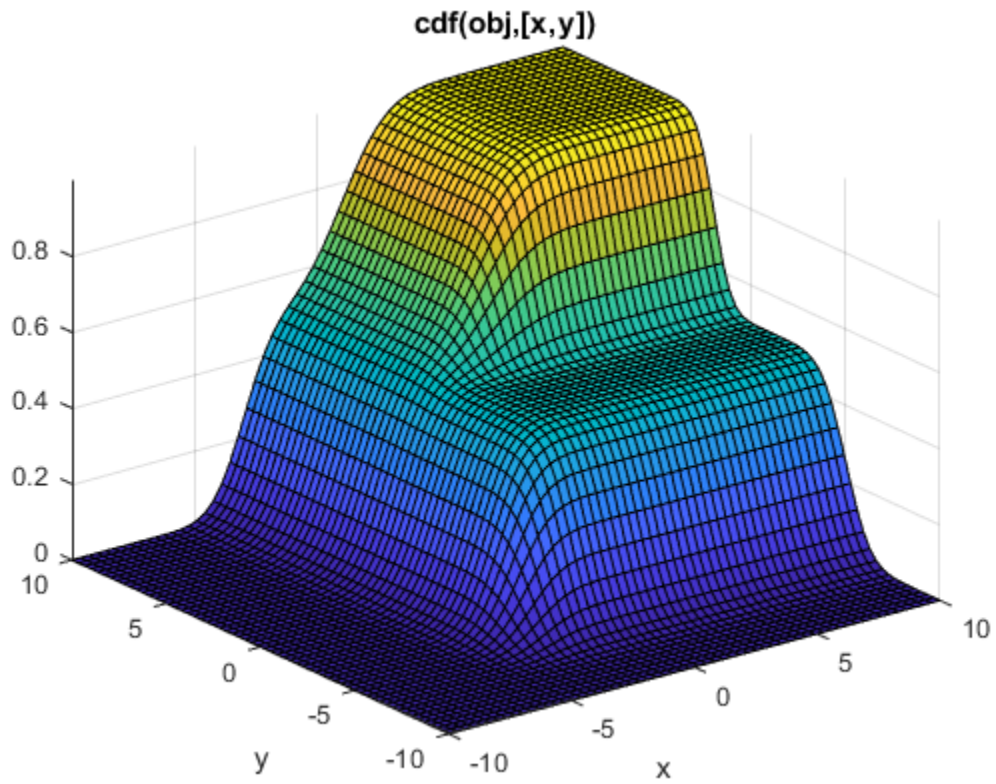
`y = cdf(obj,X)` returns a vector `y` of length  $n$  containing the values of the cumulative distribution function (cdf) for the `gmdistribution` object `obj`, evaluated at the  $n$ -by- $d$  data matrix `X`, where  $n$  is the number of observations and  $d$  is the dimension of the data. `obj` is an object created by `gmdistribution` or `fitgmdist`. `y(I)` is the cdf of observation `I`.

## Examples

### Plot a Gaussian Mixture CDF

Create a `gmdistribution` object defining a two-component mixture of bivariate Gaussian distributions.

```
MU = [1 2;-3 -5];  
SIGMA = cat(3,[2 0;0 .5],[1 0;0 1]);  
p = ones(1,2)/2;  
obj = gmdistribution(MU,SIGMA,p);  
  
ezsurf(@(x,y)cdf(obj,[x y]),[-10 10],[-10 10])
```



## See Also

`fitgmdist` | `gmdistribution` | `mvncdf` | `pdf`

# ccdesign

Central composite design

## Syntax

```
dCC = ccdesign(n)
[dCC,blocks] = ccdesign(n)
[...] = ccdesign(n, 'Name', value)
```

## Description

`dCC = ccdesign(n)` generates a central composite design for  $n$  factors.  $n$  must be an integer 2 or larger. The output matrix `dCC` is  $m$ -by- $n$ , where  $m$  is the number of runs in the design. Each row represents one run, with settings for all factors represented in the columns. Factor values are normalized so that the cube points take values between  $-1$  and  $1$ .

`[dCC,blocks] = ccdesign(n)` requests a blocked design. The output `blocks` is an  $m$ -by-1 vector of block numbers for each run. Blocks indicate runs that are to be measured under similar conditions to minimize the effect of inter-block differences on the parameter estimates.

`[...] = ccdesign(n, 'Name', value)` specifies one or more optional name/value pairs for the design. Valid parameters and their values are listed in the following table. Specify *Name* in single quotes.

Parameter	Description	Values	Value Description
center	Number of center points.	Integer	Number of center points to include.
		'uniform'	Select number of center points to give uniform precision.

Parameter	Description	Values	Value Description
		'orthogonal'	Select number of center points to give an orthogonal design. This is the default.
fraction	Fraction of full-factorial cube, expressed as an exponent of 1/2.	0	Whole design. Default when $n \leq 4$ .
		1	1/2 fraction. Default when $4 < n \leq 7$ or $n > 11$ .
		2	1/4 fraction. Default when $7 < n \leq 9$
		3	1/8 fraction. Default when $n = 10$ .
		4	1/16 fraction. Default when $n = 11$ .
type	Type of CCD.	'circumscribed'	Circumscribed (CCC). This is the default.
		'inscribed'	Inscribed (CCI).
		'faced'	Faced (CCF).
blocksize	Maximum number of points per block.	Integer	The default is Inf.

## Examples

### Two-Factor Central Composite Design

Create a 2-factor central composite design.

```
dCC = ccdesign(2, 'type', 'circumscribed')
```

```
dCC =
```

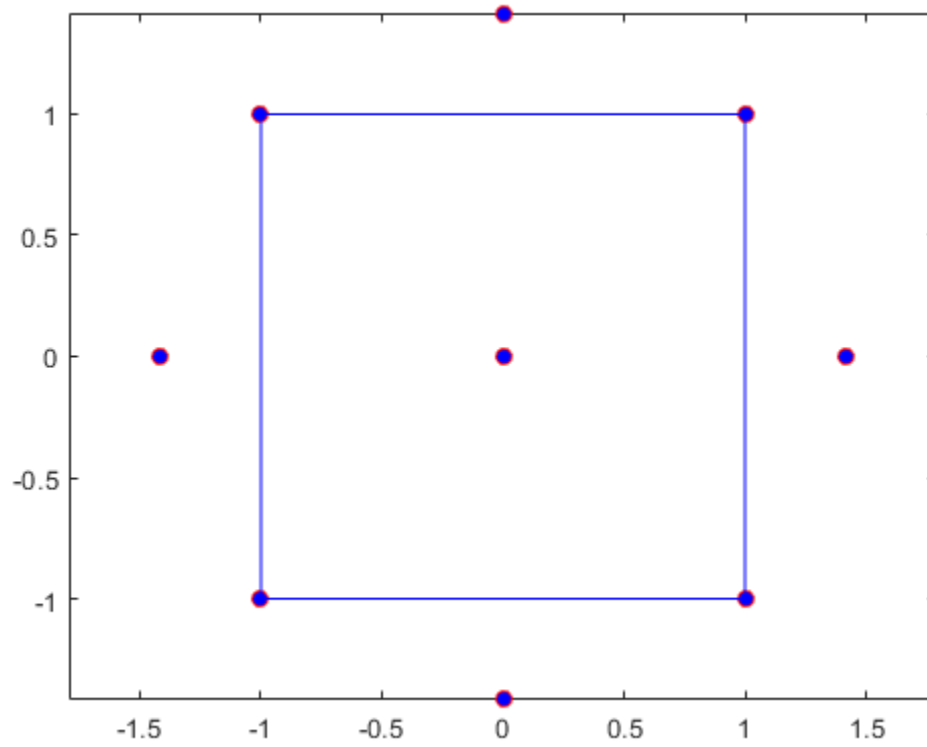


```
-1.0000 -1.0000
-1.0000  1.0000
 1.0000 -1.0000
 1.0000  1.0000
-1.4142      0
 1.4142      0
      0 -1.4142
      0  1.4142
      0      0
      0      0
      0      0
      0      0
      0      0
      0      0
      0      0
      0      0
      0      0
```

The center point is run 8 times to reduce the correlations among the coefficient estimates.

Visualize the design.

```
plot(dCC(:,1),dCC(:,2),'ro','MarkerFaceColor','b')
X = [1 -1 -1 -1; 1 1 1 -1];
Y = [-1 -1 1 -1; 1 -1 1 1];
line(X,Y,'Color','b')
axis square equal
```



## See Also

`bbdesign`

Introduced before R2006a

---

# cdf

Cumulative distribution functions

## Syntax

```
y = cdf('name', x, A)
y = cdf('name', x, A, B)
y = cdf('name', x, A, B, C)
y = cdf('name', x, A, B, C, D)

y = cdf(pd, x)

y = cdf( ____, 'upper')
```

## Description

`y = cdf('name', x, A)` returns the cumulative distribution function (cdf) for the one-parameter distribution family specified by 'name', evaluated at the values in `x`. `A` contains the parameter value for the distribution.

`y = cdf('name', x, A, B)` returns the cdf for the two-parameter distribution family specified by 'name', evaluated at the values in `x`. `A` and `B` contain the parameter values for the distribution.

`y = cdf('name', x, A, B, C)` returns the cdf for the three-parameter distribution family specified by 'name', evaluated at the values in `x`. `A`, `B`, and `C` contain the parameter values for the distribution.

`y = cdf('name', x, A, B, C, D)` returns the cdf for the four-parameter distribution family specified by 'name', evaluated at the values in `x`. `A`, `B`, `C`, and `D` contain the parameter values for the distribution.

`y = cdf(pd, x)` returns the cumulative distribution function of the probability distribution object, `pd`, evaluated at the values in `x`.

`y = cdf( ____, 'upper' )` returns the complement of the cumulative distribution function using an algorithm that more accurately computes the extreme upper tail probabilities. You can use the 'upper' argument with any of the previous syntaxes.

## Examples

### Compute the Normal Distribution cdf

Create a standard normal distribution object with the mean,  $\mu$ , equal to 0 and the standard deviation,  $\sigma$ , equal to 1.

```
mu = 0;  
sigma = 1;  
pd = makedist('Normal',mu,sigma);
```

Define the input vector  $x$  to contain the values at which to calculate the cdf.

```
x = [-2,-1,0,1,2];
```

Compute the cdf values for the standard normal distribution at the values in  $x$ .

```
y = cdf(pd,x)
```

```
y =
```

```
    0.0228    0.1587    0.5000    0.8413    0.9772
```

Each value in  $y$  corresponds to a value in the input vector  $x$ . For example, at the value  $x$  equal to 1, the corresponding cdf value  $y$  is equal to 0.8413.

Alternatively, you can compute the same cdf values without creating a probability distribution object. Use the `cdf` function, and specify a standard normal distribution using the same parameter values for  $\mu$  and  $\sigma$ .

```
y2 = cdf('Normal',x,mu,sigma)
```

```
y2 =
```

```
0.0228    0.1587    0.5000    0.8413    0.9772
```

The cdf values are the same as those computed using the probability distribution object.

### Compute the Poisson Distribution cdf

Create a Poisson distribution object with the rate parameter,  $\lambda$ , equal to 2.

```
lambda = 2;  
pd = makedist('Poisson', lambda);
```

Define the input vector  $x$  to contain the values at which to calculate the cdf.

```
x = [0, 1, 2, 3, 4];
```

Compute the cdf values for the Poisson distribution at the values in  $x$ .

```
y = cdf(pd, x)
```

```
y =
```

```
0.1353    0.4060    0.6767    0.8571    0.9473
```

Each value in  $y$  corresponds to a value in the input vector  $x$ . For example, at the value  $x$  equal to 3, the corresponding cdf value  $y$  is equal to 0.8571.

Alternatively, you can compute the same cdf values without creating a probability distribution object. Use the `cdf` function, and specify a Poisson distribution using the same value for the rate parameter,  $\lambda$ .

```
y2 = cdf('Poisson', x, lambda)
```

```
y2 =
```

```
0.1353    0.4060    0.6767    0.8571    0.9473
```

The cdf values are the same as those computed using the probability distribution object.

## Input Arguments

**'name'** — Probability distribution name

valid probability distribution name

Probability distribution name, specified as one of the following.

name	Distribution	Input Parameter A	Input Parameter B	Input Parameter C	Input Parameter D
'Beta '	“Beta Distribution” on page B-4	$\alpha$ : first shape parameter	$b$ : second shape parameter	—	—
'Binomial '	“Binomial Distribution” on page B-9	$n$ : number of trials	$p$ : probability of success for each trial	—	—
'BirnbauSaunders '	“Birnbau-Saunders Distribution” on page B-13	$\beta$ : scale parameter	$\gamma$ : shape parameter	—	—
'Burr '	“Burr Type XII Distribution” on page B-15	$\alpha$ : scale parameter	$c$ : first shape parameter	$k$ : second shape parameter	—
'Chisquare '	“Chi-Square Distribution” on page B-28	$\nu$ : degrees of freedom	—	—	—
'Exponential '	“Exponential Distribution” on page B-34	$\mu$ : mean	—	—	—
'Extreme Value '	“Extreme Value Distribution” on page B-38	$\mu$ : location parameter	$\sigma$ : scale parameter	—	—
'F '	“F Distribution” on page B-44	$\nu_1$ : numerator degrees of freedom	$\nu_2$ : denominator degrees of freedom	—	—

name	Distribution	Input Parameter A	Input Parameter B	Input Parameter C	Input Parameter D
'Gamma '	“Gamma Distribution” on page B-47	$\alpha$ : shape parameter	$b$ : scale parameter	—	—
'Generalized Extreme Value '	“Generalized Extreme Value Distribution” on page B-53	$k$ : shape parameter	$\sigma$ : scale parameter	$\mu$ : location parameter	—
'Generalized Pareto '	“Generalized Pareto Distribution” on page B-58	$k$ : tail index (shape) parameter	$\sigma$ : scale parameter	$\mu$ : threshold (location) parameter	—
'Geometric '	“Geometric Distribution” on page B-64	$p$ : probability parameter	—	—	—
'HalfNormal '	“Half-Normal Distribution” on page B-73	$\mu$ : location parameter	$\sigma$ : scale parameter	—	—
'Hypergeometric '	“Hypergeometric Distribution” on page B-79	$m$ : size of the population	$k$ : number of items with the desired characteristic in the population	$n$ : number of samples drawn	—
'InverseGaussian '	“Inverse Gaussian Distribution” on page B-82	$\mu$ : scale parameter	$\lambda$ : shape parameter	—	—
'Logistic '	“Logistic Distribution” on page B-96	$\mu$ : mean	$\sigma$ : scale parameter	—	—
'LogLogistic '	“Loglogistic Distribution” on page B-98	$\mu$ : log mean	$\sigma$ : log scale parameter	—	—
'Lognormal '	“Lognormal Distribution” on page B-100	$\mu$ : log mean	$\sigma$ : log standard deviation	—	—
'Nakagami '	“Nakagami Distribution” on page B-118	$\mu$ : shape parameter	$\omega$ : scale parameter	—	—

name	Distribution	Input Parameter A	Input Parameter B	Input Parameter C	Input Parameter D
'Negative Binomial'	“Negative Binomial Distribution” on page B-120	$r$ : number of successes	$p$ : probability of success in a single trial	—	—
'Noncentral F'	“Noncentral F Distribution” on page B-128	$\nu_1$ : numerator degrees of freedom	$\nu_2$ : denominator degrees of freedom	$\delta$ : noncentrality parameter	—
'Noncentral t'	“Noncentral t Distribution” on page B-131	$\nu$ : degrees of freedom	$\delta$ : noncentrality parameter	—	—
'Noncentral Chi-square'	“Noncentral Chi-Square Distribution” on page B-125	$\nu$ : degrees of freedom	$\delta$ : noncentrality parameter	—	—
'Normal'	“Normal Distribution” on page B-134	$\mu$ : mean	$\sigma$ : standard deviation	—	—
'Poisson'	“Poisson Distribution” on page B-143	$\lambda$ : mean	—	—	—
'Rayleigh'	“Rayleigh Distribution” on page B-146	$b$ : scale parameter	—	—	—
'Rician'	“Rician Distribution” on page B-149	$s$ : noncentrality parameter	$\sigma$ : scale parameter	—	—
'Stable'	“Stable Distribution” on page B-151	$\alpha$ : first shape parameter	$\beta$ : second shape parameter	$\gamma$ : scale parameter	$\delta$ : location parameter
'T'	“Student's t Distribution” on page B-166	$\nu$ : degrees of freedom	—	—	—



name	Distribution	Input Parameter A	Input Parameter B	Input Parameter C	Input Parameter D
'tLocationScale'	"t Location-Scale Distribution" on page B-175	$\mu$ : location parameter	$\sigma$ : scale parameter	$\nu$ : shape parameter	—
'Uniform'	"Uniform Distribution (Continuous)" on page B-184	$a$ : lower endpoint (minimum)	$b$ : upper endpoint (maximum)	—	—
'Discrete Uniform'	"Uniform Distribution (Discrete)" on page B-190	$n$ : maximum observable value	—	—	—
'Weibull'	"Weibull Distribution" on page B-193	$a$ : scale parameter	$b$ : shape parameter	—	—

### **x** — Values at which to evaluate cdf

scalar value | array of scalar values

Values at which to evaluate the cdf, specified as a scalar value, or an array of scalar values.

- If  $x$  is a scalar value, and if you specify distribution parameters A, B, C, or D as arrays, then `cdf` expands  $x$  into a constant array of the same size as the parameters.
- If  $x$  is an array, and if you specify distribution parameters A, B, C, or D as arrays, then  $x$ , A, B, C, and D must all be the same size.

Example: `[0.1, 0.25, 0.5, 0.75, 0.9]`

Data Types: `single` | `double`

### **A** — First probability distribution parameter

scalar value | array of scalar values

First probability distribution parameter, specified as a scalar value, or an array of scalar values.

If  $x$  and A are arrays, they must be the same size. If  $x$  is a scalar, then `cdf` expands it into a constant matrix the same size as A. If A is a scalar, then `cdf` expands it into a constant matrix the same size as  $x$ .

Data Types: `single` | `double`

**B — Second probability distribution parameter**

scalar value | array of scalar values

Second probability distribution parameter, specified as a scalar value, or an array of scalar values.

If `x`, `A`, and `B` are arrays, they must be the same size. If `x` is a scalar, then `cdf` expands it into a constant matrix the same size as `A` and `B`. If `A` or `B` are scalars, then `cdf` expands them into constant matrices the same size as `x`.

Data Types: `single` | `double`

**C — Third probability distribution parameter**

scalar value | array of scalar values

Third probability distribution parameter, specified as a scalar value, or an array of scalar values.

If `x`, `A`, `B`, and `C` are arrays, they must be the same size. If `x` is a scalar, then `cdf` expands it into a constant matrix the same size as `A`, `B`, and `C`. If any of `A`, `B` or `C` are scalars, then `cdf` expands them into constant matrices the same size as `x`.

Data Types: `single` | `double`

**D — Fourth probability distribution parameter**

scalar value | array of scalar values

Fourth probability distribution parameter, specified as a scalar value, or an array of scalar values.

If `x`, `A`, `B`, `C`, and `D` are arrays, they must be the same size. If `x` is a scalar, then `cdf` expands it into a constant array the same size as `A`, `B`, `C`, and `D`. If any of `A`, `B`, `C`, or `D` are scalars, then `cdf` expands them into constant matrices the same size as `x`.

Data Types: `single` | `double`

**pd — Probability distribution**

probability distribution object

Probability distribution, specified as a probability distribution object created using one of the following.

<code>makedist</code>	Create a probability distribution object using specified parameter values.
<code>fitdist</code>	Fit a probability distribution object to sample data.
<code>dfittool</code>	Fit a probability distribution object to sample data using the interactive Distribution Fitting app.
<code>paretotails</code>	Create a Pareto tails object.
<code>gmdistribution</code>	Create a Gaussian mixture distribution object.

## Output Arguments

### **y** — Cumulative distribution function

array

Cumulative distribution function of the specified probability distribution, returned as an array.

- If you specify distribution parameters A, B, C, or D, then **y** is the same size as **x**, A, B, C, and D after any necessary scalar expansion.
- If you specify a probability distribution object, **pd**, then **y** has the same dimensions as **x**.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`ecdf` | `icdf` | `pdf`

## **Topics**

“Working with Probability Distributions” on page 5-3

**Introduced before R2006a**

## cdf

**Class:** `piecwisedistribution`

Cumulative distribution function for piecewise distribution

## Syntax

```
p= cdf(obj,x)
p= cdf(obj,x,'upper')
```

## Description

`p= cdf(obj,x)` returns an array `P` of values of the cumulative distribution function for the piecewise distribution object `obj`, evaluated at the values in the array `X`.

`p= cdf(obj,x,'upper')` returns the complement of the piecewise distribution cdf evaluated at the values in `x`, using an algorithm that more accurately computes the extreme upper tail probabilities.

## Examples

### Fit Pareto Tails to *t* Distribution

Fit Pareto tails to a *t* distribution at cumulative probabilities 0.1 and 0.9.

```
t = trnd(3,100,1);
obj = paretotails(t,0.1,0.9);
[p,q] = boundary(obj)
```

```
p =
    0.1000
    0.9000
```

```
q =  
-1.8487  
2.0766
```

Compute the cdf at the values in `q`.

```
cdf(obj, q)
```

```
ans =  
0.1000  
0.9000
```

## See Also

[icdf](#) | [paretotails](#) | [pdf](#)

# cdf

**Class:** ProbDist

Return cumulative distribution function (CDF) for ProbDist object

## Syntax

$Y = \text{cdf}(PD, X)$

## Description

$Y = \text{cdf}(PD, X)$  returns  $Y$ , an array containing the cumulative distribution function (CDF) for the ProbDist object  $PD$ , evaluated at values in  $X$ .

## Input Arguments

$PD$	An object of the class ProbDistUnivParam or ProbDistUnivKernel.
$X$	A numeric array of values where you want to evaluate the CDF.

## Output Arguments

$Y$	An array containing the cumulative distribution function (CDF) for the ProbDist object $PD$ .
-----	---

## See Also

[cdf](#)

## cdf

**Class:** prob.TruncatableDistribution

**Package:** prob

Cumulative distribution function of probability distribution object

## Syntax

```
y = cdf(pd, x)
y = cdf(pd, x, 'upper')
```

## Description

`y = cdf(pd, x)` returns the cumulative distribution function (cdf) of the probability distribution `pd` at the values in `x`.

`y = cdf(pd, x, 'upper')` returns the complement of the cumulative distribution function (cdf) of the probability distribution `pd` at the values in `x`, using an algorithm that more accurately computes the extreme upper tail probabilities.

## Input Arguments

**pd** — Probability distribution

probability distribution object

Probability distribution, specified as a probability distribution object. Create a probability distribution object with specified parameter values using `makedist`. Alternatively, for fittable distributions, create a probability distribution object by fitting it to data using `fitdist` or the Distribution Fitting app.

**x** — Values at which to calculate cdf

array

Values at which to calculate the cdf, specified as an array.



Data Types: `single` | `double`

**upper** — Upper tail probability flag

`'upper'`

Upper tail probability flag, specified as `'upper'`. If you specify `'upper'`, then `cdf` returns the complement of the cdf of `pd`, using an algorithm that more accurately computes the extreme upper tail probabilities.

## Output Arguments

**y** — Cumulative distribution function

array

Cumulative distribution function of the specified probability distribution, evaluated at the values in `x`, returned as a array. `y` has the same dimensions as `x`.

## Examples

### Plot Standard Normal Distribution cdf

Create a standard normal distribution object.

```
pd = makedist('Normal')
```

```
pd =
```

```
NormalDistribution
```

```
Normal distribution
```

```
mu = 0
```

```
sigma = 1
```

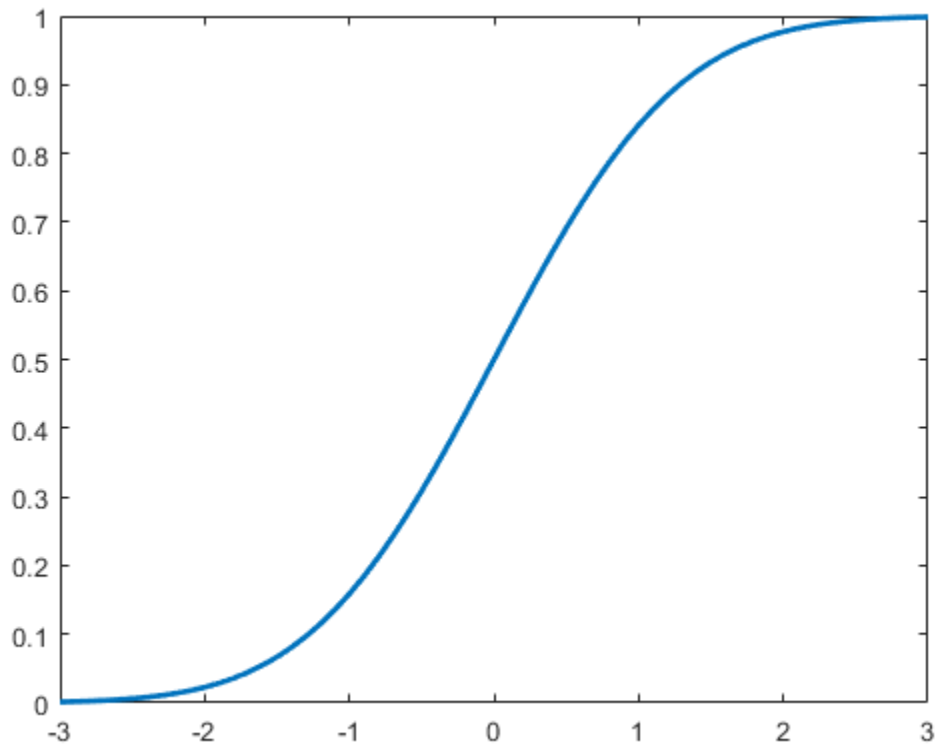
Specify the `x` values and compute the cdf.

```
x = -3:.1:3;
```

```
cdf_normal = cdf(pd,x);
```

Plot the cdf of the standard normal distribution.

```
plot(x,cdf_normal,'LineWidth',2)
```



### Plot Gamma Distribution cdf

Create three gamma distribution objects. The first uses the default parameter values. The second specifies  $a = 1$  and  $b = 2$ . The third specifies  $a = 2$  and  $b = 1$ .

```
pd_gamma = makedist('Gamma')
```

```
pd_gamma =
```

```
GammaDistribution

Gamma distribution
a = 1
b = 1

pd_12 = makedist('Gamma','a',1,'b',2)

pd_12 =

GammaDistribution

Gamma distribution
a = 1
b = 2

pd_21 = makedist('Gamma','a',2,'b',1)

pd_21 =

GammaDistribution

Gamma distribution
a = 2
b = 1
```

Specify the  $x$  values and compute the cdf for each distribution.

```
x = 0:.1:5;
cdf_gamma = cdf(pd_gamma,x);
cdf_12 = cdf(pd_12,x);
cdf_21 = cdf(pd_21,x);
```

Create a plot to visualize how the cdf of the gamma distribution changes when you specify different values for the shape parameters  $a$  and  $b$ .

```
figure;
J = plot(x,cdf_gamma)
hold on;
```

```
K = plot(x,cdf_12,'r--')
L = plot(x,cdf_21,'k-.')
set(J,'LineWidth',2);
set(K,'LineWidth',2);
legend([J K L], 'a = 1, b = 1', 'a = 1, b = 2', 'a = 2, b = 1', 'Location', 'southeast');
hold off;
```

J =

Line with properties:

```
Color: [0 0.4470 0.7410]
LineStyle: '-'
LineWidth: 0.5000
Marker: 'none'
MarkerSize: 6
MarkerFaceColor: 'none'
XData: [1x51 double]
YData: [1x51 double]
ZData: [1x0 double]
```

Use GET to show all properties

K =

Line with properties:

```
Color: [1 0 0]
LineStyle: '--'
LineWidth: 0.5000
Marker: 'none'
MarkerSize: 6
MarkerFaceColor: 'none'
XData: [1x51 double]
YData: [1x51 double]
ZData: [1x0 double]
```

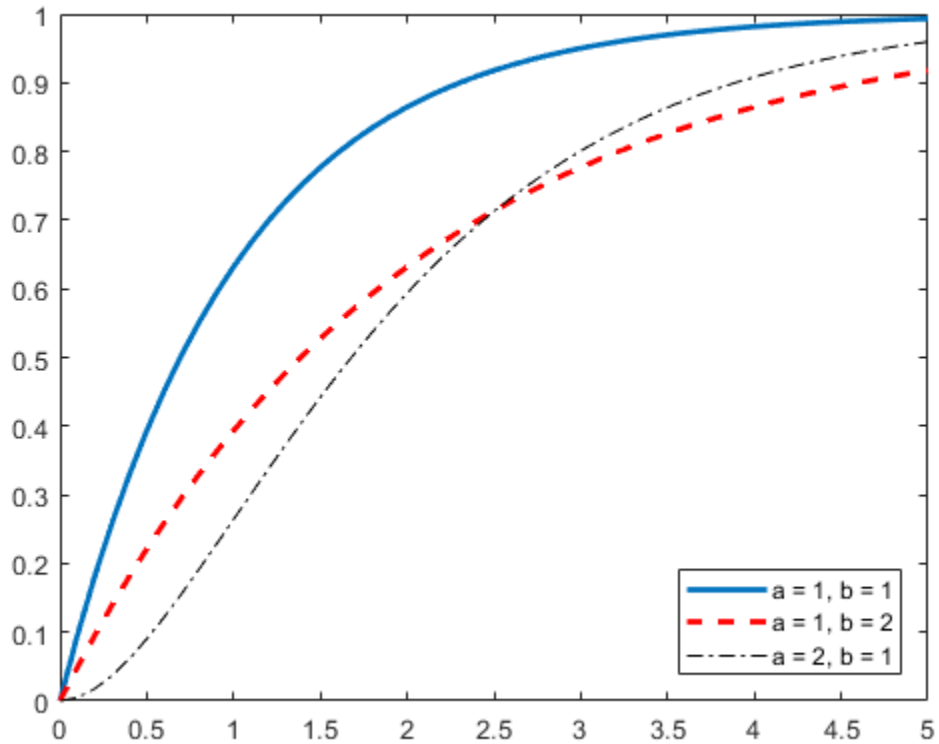
Use GET to show all properties

L =

Line with properties:

```
        Color: [0 0 0]
        LineStyle: '-.'
        LineWidth: 0.5000
        Marker: 'none'
    MarkerSize: 6
    MarkerFaceColor: 'none'
        XData: [1x51 double]
        YData: [1x51 double]
        ZData: [1x0 double]
```

Use GET to show all properties



## See Also

[cdf](#) | [distributionFitter](#) | [ecdf](#) | [fitdist](#) | [icdf](#) | [makedist](#) | [pdf](#)

# cdfplot

Empirical cumulative distribution function plot

## Syntax

```
cdfplot(X)
h = cdfplot(X)
[h,stats] = cdfplot(X)
```

## Description

`cdfplot(X)` displays a plot of the empirical cumulative distribution function (cdf) for the data in the vector `X`. The empirical cdf  $F(x)$  is defined as the proportion of `X` values less than or equal to  $x$ .

This plot is useful for examining the distribution of a sample of data. You can overlay a theoretical cdf on the same plot to compare the empirical distribution of the sample to the theoretical distribution.

The `kstest`, `kstest2`, and `lillietest` functions compute test statistics that are derived from the empirical cdf. You may find the empirical cdf plot produced by `cdfplot` useful in helping you to understand the output from those functions.

`h = cdfplot(X)` returns a handle to the cdf curve.

`[h,stats] = cdfplot(X)` also returns a `stats` structure with the following fields.

Field	Description
<code>stats.min</code>	Minimum value
<code>stats.max</code>	Maximum value
<code>stats.mean</code>	Sample mean
<code>stats.median</code>	Sample median (50th percentile)
<code>stats.std</code>	Sample standard deviation

## Examples

### Compare Empirical cdf with Sampling Distribution

This example shows how to plot the empirical cdf of sample data and compare it with a plot of the cdf for the sampling distribution. In practice, the sampling distribution would be unknown, and would be chosen to match the empirical cdf.

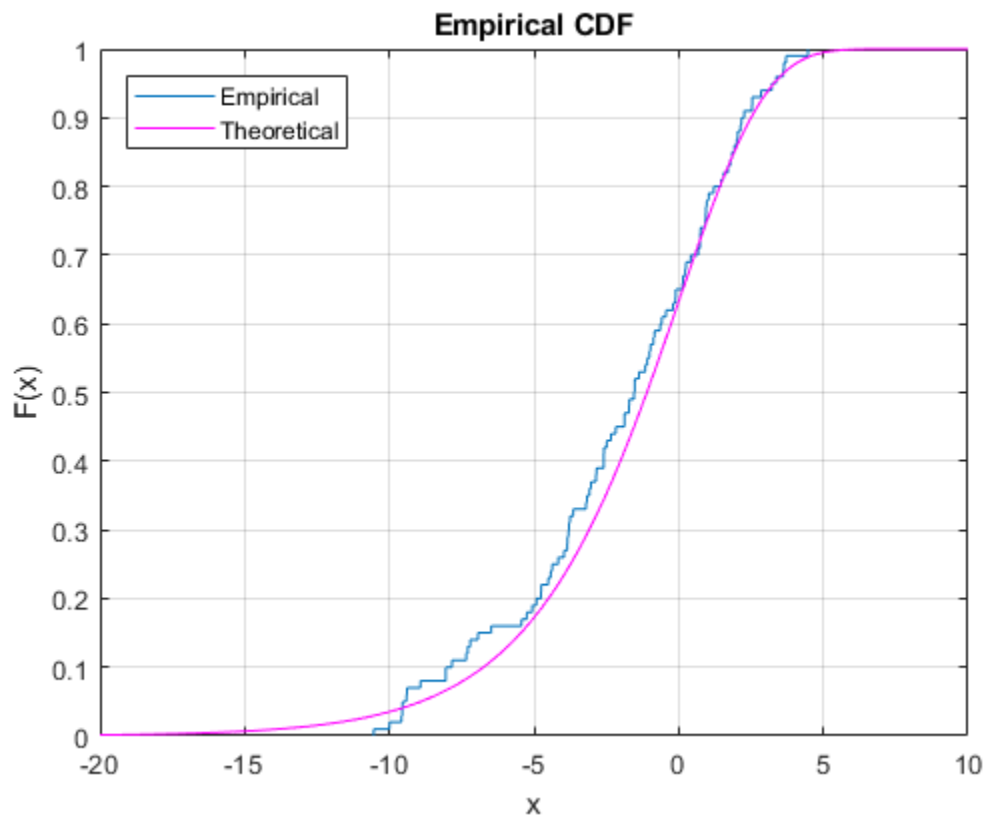
Generate random sample data from an extreme value distribution with a location parameter  $\mu = 0$  and scale parameter  $\sigma = 3$ .

```
rng default; % For reproducibility
y = evrnd(0,3,100,1);
```

Plot the empirical cdf of the sample data on the same figure as the cdf of the sampling distribution.

```
cdfplot(y)
hold on
x = -20:0.1:10;
f = evcdf(x,0,3);
plot(x,f,'m')
legend('Empirical','Theoretical','Location','NW')
```





## See Also

`ecdf`

Introduced before R2006a

## cell2dataset

Convert cell array to dataset array

---

**Note** The `dataset` data type might be removed in a future release. To work with heterogeneous data, use the MATLAB® `table` data type instead. See MATLAB `table` documentation for more information.

---

## Syntax

```
ds = cell2dataset(C)
ds = cell2dataset(C,Name,Value)
```

## Description

`ds = cell2dataset(C)` converts a cell array to a dataset array.

`ds = cell2dataset(C,Name,Value)` performs the conversion using additional options specified by one or more `Name,Value` pair arguments.

## Examples

### Convert Cell Array to Dataset Array

Convert a cell array to a dataset array using the default options.

Create a cell array to convert.

```
C = {'Name','Gender','SystolicBP','DiastolicBP';
     'CLARK','M',124,93;
     'BROWN','F',122,80;
     'MARTIN','M',130,92}
```

```
C =
```

```
4x4 cell array
```

```
{'Name'   } {'Gender'} {'SystolicBP'} {'DiastolicBP'}
{'CLARK'  } {'M'      } {[      124]} {[      93]}
{'BROWN'  } {'F'      } {[      122]} {[      80]}
{'MARTIN' } {'M'      } {[      130]} {[      92]}
```

Convert the cell array to a dataset array.

```
ds = cell2dataset(C)
```

```
ds =
```

Name	Gender	SystolicBP	DiastolicBP
'CLARK'	'M'	124	93
'BROWN'	'F'	122	80
'MARTIN'	'M'	130	92

The first row of C become the variable names in the output dataset array, ds.

### Create a Dataset Array with Multicolumn Variables

Convert a cell array to a dataset array containing multicolumn variables.

Create a cell array to convert.

```
C = {'Name','Gender','SystolicBP','DiastolicBP';
     'CLARK','M',124,93;
     'BROWN','F',122,80;
     'MARTIN','M',130,92}
```

```
C =
```

```
4x4 cell array
```

```
{'Name'   } {'Gender'} {'SystolicBP'} {'DiastolicBP'}
{'CLARK'  } {'M'      } {[      124]} {[      93]}
{'BROWN'  } {'F'      } {[      122]} {[      80]}
```

```
{'MARTIN'} {'M'} {[ 130]} {[ 92]}
```

Convert the cell array to a dataset array, combining the systolic and diastolic blood pressure measurements into one variable named `BloodPressure`.

```
ds = cell2dataset(C, 'NumCols', [1,1,2]);  
ds.Properties.VarNames{3} = 'BloodPressure';  
ds
```

```
ds =
```

Name	Gender	BloodPressure
'CLARK'	'M'	124 93
'BROWN'	'F'	122 80
'MARTIN'	'M'	130 92

The output dataset array has three observations and three variables.

- “Create a Dataset Array from Workspace Variables” on page 2-70
- “Create a Dataset Array from a File” on page 2-77

## Input Arguments

### **c** — Input cell array

cell array

Input cell array to convert to a dataset array, specified as an  $M$ -by- $N$  cell array. Each column of `C` becomes a variable in the output dataset array, `ds`. By default, `cell2dataset` assumes that the first row of `C` contains variable names.

Data Types: `cell`

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'ReadVarNames', false, 'ReadObsNames', true` specifies that the first row of the cell array does not contain variable names, but the first column contains observation names.

#### **ReadVarNames** — Indicator for whether or not to read variable names

`true` (default) | `false`

Indicator for whether or not to read variable names from the first row of the input cell array, specified as the comma-separated pair consisting of `'ReadVarNames'` and either `true` or `false`. The default value is `true`, unless variable names are specified using the name-value pair argument `VarNames`. When `ReadVarNames` is `false`, `cell2dataset` creates default variable names if you do not provide any.

Example: `'ReadVarNames', false`

#### **VarNames** — Variable names for output dataset array

cell array of character vectors

Variable names for the output dataset array, specified as the comma-separated pair consisting of `'VarNames'` and a cell array of character vectors. You must provide a variable name for each variable in `ds`. The names must be valid MATLAB identifiers, and must be unique.

Example: `'VarNames', {'myVar1', 'myVar2', 'myVar3'}`

#### **ReadObsNames** — Indicator for whether or not to read observation names

`false` (default) | `true`

Indicator for whether or not to read observation names from the input cell array, specified as the comma-separated pair consisting of `'ReadObsNames'` and either `true` or `false`. When `ReadObsNames` has the value `true`, `cell2dataset` creates observation names in `ds` using the first column of `C`, and sets `ds.Properties.DimNames` equal to `{C{1,1}, 'Variables'}`.

Example: `'ReadObsNames', true`

#### **ObsNames** — Observation names for output dataset array

cell array of character vectors

Observation names for the output dataset array, specified as the comma-separated pair consisting of `'ObsNames'` and a cell array of character vectors. The names do not need to be valid MATLAB identifiers, but they must be unique.

**NumCols** — Number of columns for each variable

vector of nonnegative integers

Number of columns for each variable in `ds`, specified as the comma-separated pair consisting of `'NumCols'` and a vector of nonnegative integers. When the number of columns for a variable is greater than one, `cell2dataset` combines multiple columns in `C` into a single variable in `ds`. The vector you assign to `NumCols` must sum to `size(C, 2)`, or `size(C, 1)` of `ReadObsNames` is equal to `true`.

For example, to convert a cell array with eight columns into a dataset array with five variables, specify a vector with five elements that sum to eight, such as `'NumCols', [1, 1, 3, 1, 2]`.

## Output Arguments

**ds** — Output dataset array

dataset array

Output dataset array, returned by default with a variable for each column of `C`, an observation for each row of `C` (except for the first row), and variable names corresponding to the first row of `C`.

- If you set `ReadVarNames` equal to `false` (or specify `VarNames`), then there is an observation in `ds` for each row of `C`, and `cell2dataset` creates default variable names (or uses the names in `VarNames`).
- If you set `ReadObsNames` equal to `true`, then `cell2dataset` uses the first column of `C` as observation names.
- If you specify `NumCols`, then the number of variables in `ds` is equal to the length of the specified vector of column numbers.

## See Also

`dataset` | `dataset2cell` | `struct2dataset`

## Topics

“Create a Dataset Array from Workspace Variables” on page 2-70

“Create a Dataset Array from a File” on page 2-77

“Dataset Arrays” on page 2-142

**Introduced in R2012b**

## cellstr

**Class:** dataset

Create cell array of character vectors from dataset array

---

**Note** The `dataset` data type might be removed in a future release. To work with heterogeneous data, use the MATLAB® `table` data type instead. See MATLAB `table` documentation for more information.

---

## Syntax

```
B = cellstr(A)
B = cellstr(A, VARS)
```

## Description

`B = cellstr(A)` returns the contents of the dataset `A`, converted to a cell array of character vectors. The variables in the dataset must support the conversion and must have compatible sizes.

`B = cellstr(A, VARS)` returns the contents of the dataset variables specified by `VARS`. `VARS` is a positive integer, a vector of positive integers, a variable name, a cell array containing one or more variable names, or a logical vector.

## See Also

`dataset.double` | `dataset.replacedata`



## chi2cdf

Chi-square cumulative distribution function

### Syntax

```
p = chi2cdf(x,v)
p = chi2cdf(x,v,'upper')
```

### Description

`p = chi2cdf(x,v)` computes the chi-square cdf at each of the values in `x` using the corresponding degrees of freedom in `v`. `x` and `v` can be vectors, matrices, or multidimensional arrays that have the same size. A scalar input is expanded to a constant array with the same dimensions as the other input. The degrees of freedom parameters in `v` must be positive, and the values in `x` must lie on the interval `[0 Inf]`.

`p = chi2cdf(x,v,'upper')` returns the complement of the chi-square cdf at each value in `x`, using an algorithm that more accurately computes the extreme upper tail probabilities.

The  $\chi^2$  cdf for a given value  $x$  and degrees-of-freedom  $\nu$  is

$$p = F(x|\nu) = \int_0^x \frac{t^{(\nu-2)/2} e^{-t/2}}{2^{\nu/2} \Gamma(\nu/2)} dt$$

where  $\Gamma(\cdot)$  is the Gamma function.

The chi-square density function with  $\nu$  degrees-of-freedom is the same as the gamma density function with parameters  $\nu/2$  and 2.

### Examples

**Compute Chi-Square CDF**

```
probability = chi2cdf(5,1:5)
```

```
probability =
```

```
    0.9747    0.9179    0.8282    0.7127    0.5841
```

```
probability = chi2cdf(1:5,1:5)
```

```
probability =
```

```
    0.6827    0.6321    0.6084    0.5940    0.5841
```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

[cdf](#) | [chi2inv](#) | [chi2pdf](#) | [chi2rnd](#) | [chi2stat](#)

### Topics

“Chi-Square Distribution” on page B-28

**Introduced before R2006a**

## chi2gof

Chi-square goodness-of-fit test

### Syntax

```
h = chi2gof(x)
h = chi2gof(x,Name,Value)
[h,p] = chi2gof(____)
[h,p,stats] = chi2gof(____)
```

### Description

`h = chi2gof(x)` returns a test decision for the null hypothesis that the data in vector `x` comes from a normal distribution with a mean and variance estimated from `x`, using the chi-square goodness-of-fit test on page 32-349. The alternative hypothesis is that the data does not come from such a distribution. The result `h` is 1 if the test rejects the null hypothesis at the 5% significance level, and 0 otherwise.

`h = chi2gof(x,Name,Value)` returns a test decision for the chi-square goodness-of-fit test with additional options specified by one or more name-value pair arguments. For example, you can test for a distribution other than normal, or change the significance level of the test.

`[h,p] = chi2gof(____)` also returns the  $p$ -value `p` of the hypothesis test, using any of the input arguments from the previous syntaxes.

`[h,p,stats] = chi2gof(____)` also returns the structure `stats`, containing information about the test statistic.

### Examples

### Test for a Normal Distribution

Create a standard normal probability distribution object. Generate a data vector  $x$  using random numbers from the distribution.

```
pd = makedist('Normal');  
rng default; % for reproducibility  
x = random(pd,100,1);
```

Test the null hypothesis that the data in  $x$  comes from a population with a normal distribution.

```
h = chi2gof(x)
```

```
h =  
  
    0
```

The returned value  $h = 0$  indicates that `chi2gof` does not reject the null hypothesis at the default 5% significance level.

### Test the Hypothesis at a Different Significance Level

Create a standard normal probability distribution object. Generate a data vector  $x$  using random numbers from the distribution.

```
pd = makedist('Normal');  
rng default; % for reproducibility  
x = random(pd,100,1);
```

Test the null hypothesis that the data in  $x$  comes from a population with a normal distribution at the 1% significance level.

```
[h,p] = chi2gof(x,'Alpha',0.01)
```

```
h =  
  
    0
```

```
p =  
    0.3775
```

The returned value  $h = 0$  indicates that `chi2gof` does not reject the null hypothesis at the 1% significance level.

### Test for a Weibull Distribution Using a Probability Distribution Object

Load the lightbulb lifetime sample data.

```
load(fullfile(matlabroot, 'examples', 'stats', 'lightbulb.mat'));
```

Create a vector from the first column of the data matrix, which contains the lifetime in hours of the lightbulbs.

```
x = lightbulb(:,1);
```

Test the null hypothesis that the data in `x` comes from a population with a Weibull distribution. Use `fitdist` to create a probability distribution object with `A` and `B` parameters estimated from the data.

```
pd = fitdist(x, 'Weibull');  
h = chi2gof(x, 'CDF', pd)
```

```
h =  
    1
```

The returned value  $h = 1$  indicates that `chi2gof` rejects the null hypothesis at the default 5% significance level.

### Test for a Poisson Distribution

Create six bins, numbered 0 through 5, to use for data pooling.

```
bins = 0:5;
```

Create a vector containing the observed counts for each bin and compute the total number of observations.

```
obsCounts = [6 16 10 12 4 2];  
n = sum(obsCounts);
```

Fit a Poisson probability distribution object to the data and compute the expected count for each bin. Use the transpose operator `.'` to transform `bins` and `obsCounts` from row vectors to column vectors.

```
pd = fitdist(bins', 'Poisson', 'Frequency', obsCounts');  
expCounts = n * pdf(pd, bins);
```

Test the null hypothesis that the data in `obsCounts` comes from a Poisson distribution with a lambda parameter equal to `lambdaHat`.

```
[h,p,st] = chi2gof(bins, 'Ctrs', bins, ...  
                  'Frequency', obsCounts, ...  
                  'Expected', expCounts, ...  
                  'NParams', 1)  
  
h =  
  
    0  
  
p =  
  
    0.4654  
  
st =  
  
struct with fields:  
  
    chi2stat: 2.5550  
         df: 3  
    edges: [-0.5000 0.5000 1.5000 2.5000 3.5000 5.5000]  
         O: [6 16 10 12 6]  
         E: [7.0429 13.8041 13.5280 8.8383 6.0284]
```

The returned value  $h = 0$  indicates that `chi2gof` does not reject the null hypothesis at the default 5% significance level. The vector `E` contains the expected counts for each bin under the null hypothesis, and `O` contains the observed counts for each bin.

## Input Arguments

### **x** — Sample data

vector

Sample data for the hypothesis test, specified as a vector.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'NBins', 8, 'Alpha', 0.01` pools the data into eight bins and conducts the hypothesis test at the 1% significance level.

### **NBins** — Number of bins

10 (default) | positive integer value

Number of bins to use for the data pooling, specified as the comma-separated pair consisting of `'NBins'` and a positive integer value. If you specify a value for `NBins`, do not specify a value for `Ctrs` or `Edges`.

Example: `'NBins', 8`

Data Types: `single` | `double`

### **Ctrs** — Bin centers

vector

Bin centers, specified as the comma-separated pair consisting of `'Ctrs'` and a vector of center values for each bin. If you specify a value for `Ctrs`, do not specify a value for `NBins` or `Edges`.

Example: `'Ctrs', [1 2 3 4 5]`

Data Types: `single` | `double`

### **Edges — Bin edges**

vector

Bin edges, specified as the comma-separated pair consisting of 'Edges' and a vector of edge values for each bin. If you specify a value for Edges, do not specify a value for NBins or Ctrs.

Example: 'Edges', [-2.5 -1.5 -0.5 0.5 1.5 2.5]

Data Types: `single` | `double`

### **CDF — cdf of hypothesized distribution**

probability distribution object | function handle | cell array

The cdf of the hypothesized distribution, specified as the comma-separated pair consisting of 'CDF' and a probability distribution object, function handle, or cell array.

- If CDF is a probability distribution object, the degrees of freedom account for whether you estimate the parameters using `fitdist` or specify them using `makedist`.
- If CDF is a function handle, the distribution function must take `x` as its only argument.
- If CDF is a cell array, the first element must be a function handle, and the remaining elements must be parameter values, one per cell. The function must take `x` as its first argument, and the other parameters in the array as later arguments.

If you specify a value for CDF, do not specify a value for Expected.

Example: 'CDF', `pd_object`

Data Types: `single` | `double`

### **Expected — Expected counts**

vector of nonnegative values

Expected counts for each bin, specified as the comma-separated pair of 'Expected' and a vector of nonnegative values. If Expected depends on estimated parameters, use NParams to ensure that `chi2gof` correctly calculates the degrees of freedom. If you specify a value for Expected, do not specify a value for CDF.

Example: 'Expected', [19.1446 18.3789 12.3224 8.2432 4.1378]



Data Types: `single` | `double`

### **NParams** — Number of estimated parameters

positive integer value

Number of estimated parameters used to describe the null distribution, specified as the comma-separated pair consisting of 'NParams' and a positive integer value. This value adjusts the degrees of freedom of the test based on the number of estimated parameters used to compute the cdf or expected counts.

The default value for NParams depends on how you specify the null distribution:

- If you specify CDF as a probability distribution object, NParams is equal to the number of estimated parameters used to create the object.
- If you specify CDF as a function name or handle, the default value of NParams is 0.
- If you specify CDF as a cell array, the default value of NParams is the number of parameters in the array.
- If you specify Expected, the default value of NParams is 0.

Example: 'NParams',1

Data Types: `single` | `double`

### **EMin** — Minimum expected count per bin

5 (default) | nonnegative integer value

Minimum expected count per bin, specified as the comma-separated pair consisting of 'EMin' and a nonnegative integer value. If the bin at the extreme end of either tail has an expected value less than EMin, it is combined with a neighboring bin until the count in each extreme bin is at least 5. If any interior bins have a count less than 5, chi2gof displays a warning, but does not combine the interior bins. In that case, you should use fewer bins, or provide bin centers or edges, to increase the expected counts in all bins. Specify EMin as 0 to prevent the combining of bins.

Example: 'EMin',0

Data Types: `single` | `double`

### **Frequency** — Frequency

vector of nonnegative integer values

Frequency of data values, specified as the comma-separated pair consisting of 'Frequency' and a vector of nonnegative integer values that is the same length as the vector  $x$ .

Example: 'Frequency', [20 16 13 10 8]

Data Types: single | double

### **Alpha — Significance level**

0.05 (default) | scalar value in the range (0,1)

Significance level of the hypothesis test, specified as the comma-separated pair consisting of 'Alpha' and a scalar value in the range (0,1).

Example: 'Alpha', 0.01

Data Types: single | double

## Output Arguments

### **h — Hypothesis test result**

1 | 0

Hypothesis test result, returned as a logical value.

- If  $h = 1$ , this indicates the rejection of the null hypothesis at the Alpha significance level.
- If  $h = 0$ , this indicates a failure to reject the null hypothesis at the Alpha significance level.

### **p — $p$ -value**

scalar value in the range [0,1]

$p$ -value of the test, returned as a scalar value in the range [0,1].  $p$  is the probability of observing a test statistic as extreme as, or more extreme than, the observed value under the null hypothesis. Small values of  $p$  cast doubt on the validity of the null hypothesis.

### **stats — Test statistics**

structure

Test statistics, returned as a structure containing the following:

- `chi2stat` — Value of the test statistic.
- `df` — Degrees of freedom of the test.
- `edges` — Vector of bin edges after pooling.
- `O` — Vector of observed counts for each bin.
- `E` — Vector of expected counts for each bin.

## Definitions

### Chi-Square Goodness-of-Fit Test

The chi-square goodness-of-fit test determines if a data sample comes from a specified probability distribution, with parameters estimated from the data.

The test groups the data into bins, calculating the observed and expected counts for those bins, and computing the chi-square test statistic

$$\chi^2 = \sum_{i=1}^N (O_i - E_i)^2 / E_i,$$

where  $O_i$  are the observed counts and  $E_i$  are the expected counts based on the hypothesized distribution. The test statistic has an approximate chi-square distribution when the counts are sufficiently large.

## Algorithms

`chi2gof` compares the value of the test statistic to a chi-square distribution with degrees of freedom equal to  $nbins - 1 - nparams$ , where  $nbins$  is the number of bins used for the data pooling and  $nparams$  is the number of estimated parameters used to determine the expected counts. If there are not enough degrees of freedom to conduct the test, `chi2gof` returns the  $p$ -value as NaN.

## See Also

`kstest` | `lillietest`

## **Topics**

“Chi-Square Distribution” on page B-28

**Introduced before R2006a**

# chi2inv

Chi-square inverse cumulative distribution function

## Syntax

`X = chi2inv(P,V)`

## Description

`X = chi2inv(P,V)` computes the inverse of the chi-square cdf with degrees of freedom specified by `V` for the corresponding probabilities in `P`. `P` and `V` can be vectors, matrices, or multidimensional arrays that have the same size. A scalar input is expanded to a constant array with the same dimensions as the other inputs. The degrees of freedom parameters in `V` must be positive, and the values in `P` must lie in the interval `[0 1]`.

The inverse chi-square cdf for a given probability  $p$  and  $v$  degrees of freedom is

$$x = F^{-1}(p | v) = \{x : F(x | v) = p\}$$

where

$$p = F(x | v) = \int_0^x \frac{t^{(v-2)/2} e^{-t/2}}{2^{v/2} \Gamma(v/2)} dt$$

and  $\Gamma(\cdot)$  is the Gamma function. Each element of output `X` is the value whose cumulative probability under the chi-square cdf defined by the corresponding degrees of freedom parameter in `V` is specified by the corresponding value in `P`.

## Examples

Find a value that exceeds 95% of the samples from a chi-square distribution with 10 degrees of freedom.

```
x = chi2inv(0.95,10)
x =
    18.3070
```

You would observe values greater than 18.3 only 5% of the time by chance.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

[chi2cdf](#) | [chi2pdf](#) | [chi2rnd](#) | [chi2stat](#) | [icdf](#)

### Topics

“Chi-Square Distribution” on page B-28

Introduced before R2006a

## chi2pdf

Chi-square probability density function

### Syntax

`Y = chi2pdf(X,V)`

### Description

`Y = chi2pdf(X,V)` computes the chi-square pdf at each of the values in `X` using the corresponding degrees of freedom in `V`. `X` and `V` can be vectors, matrices, or multidimensional arrays that have the same size, which is also the size of the output `Y`. A scalar input is expanded to a constant array with the same dimensions as the other input. The degrees of freedom parameters in `V` must be positive, and the values in `X` must lie on the interval `[0 Inf]`.

The chi-square pdf for a given value  $x$  and  $\nu$  degrees of freedom is

$$y = f(x | \nu) = \frac{x^{(\nu-2)/2} e^{-x/2}}{2^{\nu/2} \Gamma(\nu/2)}$$

where  $\Gamma(\cdot)$  is the Gamma function.

If  $x$  is standard normal, then  $x^2$  is distributed chi-square with one degree of freedom. If  $x_1, x_2, \dots, x_n$  are  $n$  independent standard normal observations, then the sum of the squares of the  $x$ 's is distributed chi-square with  $n$  degrees of freedom (and is equivalent to the gamma density function with parameters  $\nu/2$  and 2).

### Examples

```
nu = 1:6;
x = nu;
y = chi2pdf(x,nu)
y =
    0.2420    0.1839    0.1542    0.1353    0.1220    0.1120
```

The mean of the chi-square distribution is the value of the degrees of freedom parameter, `nu`. The above example shows that the probability density of the mean falls as `nu` increases.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`chi2cdf` | `chi2inv` | `chi2rnd` | `chi2stat` | `pdf`

### Topics

“Chi-Square Distribution” on page B-28

**Introduced before R2006a**



# chi2rnd

Chi-square random numbers

## Syntax

```
R = chi2rnd(V)
R = chi2rnd(V,m,n,...)
R = chi2rnd(V,[m,n,...])
```

## Description

`R = chi2rnd(V)` generates random numbers from the chi-square distribution with degrees of freedom parameters specified by `V`. `V` can be a vector, a matrix, or a multidimensional array. `R` is the same size as `V`. The degrees of freedom parameters in `V` must be positive.

`R = chi2rnd(V,m,n,...)` or `R = chi2rnd(V,[m,n,...])` generates an `m`-by-`n`-by-... array containing random numbers from the chi-square distribution with degrees of freedom parameter `V`. `V` can be a scalar or an array of the same size as `R`.

## Examples

Note that the first and third commands are the same, but are different from the second command.

```
r = chi2rnd(1:6)
r =
    0.0037    3.0377    7.8142    0.9021    3.2019    9.0729

r = chi2rnd(6,[1 6])
r =
    6.5249    2.6226   12.2497    3.0388    6.3133    5.0388

r = chi2rnd(1:6,1,6)
```

```
r =  
    0.7638    6.0955    0.8273    3.2506    1.5469    10.9197
```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

The generated code can return a different sequence of numbers than MATLAB if either of the following is true:

- The output is nonscalar.
- An input parameter is invalid for the distribution.

### See Also

`chi2cdf` | `chi2inv` | `chi2pdf` | `chi2stat` | `random`

### Topics

“Chi-Square Distribution” on page B-28

Introduced before R2006a

# chi2stat

Chi-square mean and variance

## Syntax

```
[M,V] = chi2stat(NU)
```

## Description

`[M,V] = chi2stat(NU)` returns the mean of and variance for the chi-square distribution with degrees of freedom parameters specified by `NU`. `NU` can be a vector, a matrix, or a multidimensional array. The degrees of freedom parameters in `NU` must be positive.

The mean of the chi-square distribution is `v`, the degrees of freedom parameter, and the variance is `2v`.

## Examples

```
nu = 1:10;
nu = nu'*nu;
[m,v] = chi2stat(nu)
m =
    1     2     3     4     5     6     7     8     9    10
    2     4     6     8    10    12    14    16    18    20
    3     6     9    12    15    18    21    24    27    30
    4     8    12    16    20    24    28    32    36    40
    5    10    15    20    25    30    35    40    45    50
    6    12    18    24    30    36    42    48    54    60
    7    14    21    28    35    42    49    56    63    70
    8    16    24    32    40    48    56    64    72    80
    9    18    27    36    45    54    63    72    81    90
   10    20    30    40    50    60    70    80    90   100

v =
```

2	4	6	8	10	12	14	16	18	20
4	8	12	16	20	24	28	32	36	40
6	12	18	24	30	36	42	48	54	60
8	16	24	32	40	48	56	64	72	80
10	20	30	40	50	60	70	80	90	100
12	24	36	48	60	72	84	96	108	120
14	28	42	56	70	84	98	112	126	140
16	32	48	64	80	96	112	128	144	160
18	36	54	72	90	108	126	144	162	180
20	40	60	80	100	120	140	160	180	200

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

[chi2cdf](#) | [chi2inv](#) | [chi2pdf](#) | [chi2rnd](#)

### Topics

“Chi-Square Distribution” on page B-28

Introduced before R2006a

# children

**Class:** classregtree

Child nodes

---

**Note** classregtree will be removed in a future release. See fitctree, fitrtree, ClassificationTree, or RegressionTree instead.

---

## Syntax

```
C = children(t)
C = children(t,nodes)
```

## Description

`C = children(t)` returns an  $n$ -by-2 array `C` containing the numbers of the child nodes for each node in the tree `t`, where  $n$  is the number of nodes. Leaf nodes have child node 0.

`C = children(t,nodes)` takes a vector `nodes` of node numbers and returns the children for the specified nodes.

## Examples

Create a classification tree for Fisher's iris data:

```
load fisheriris;

t = classregtree(meas,species,...
                'names',{'SL' 'SW' 'PL' 'PW'})
t =
Decision tree for classification
1 if PL<2.45 then node 2 elseif PL>=2.45 then node 3 else setosa
2 class = setosa
3 if PW<1.75 then node 4 elseif PW>=1.75 then node 5 else versicolor
4 if PL<4.95 then node 6 elseif PL>=4.95 then node 7 else versicolor
5 class = virginica
6 if PW<1.65 then node 8 elseif PW>=1.65 then node 9 else versicolor
```

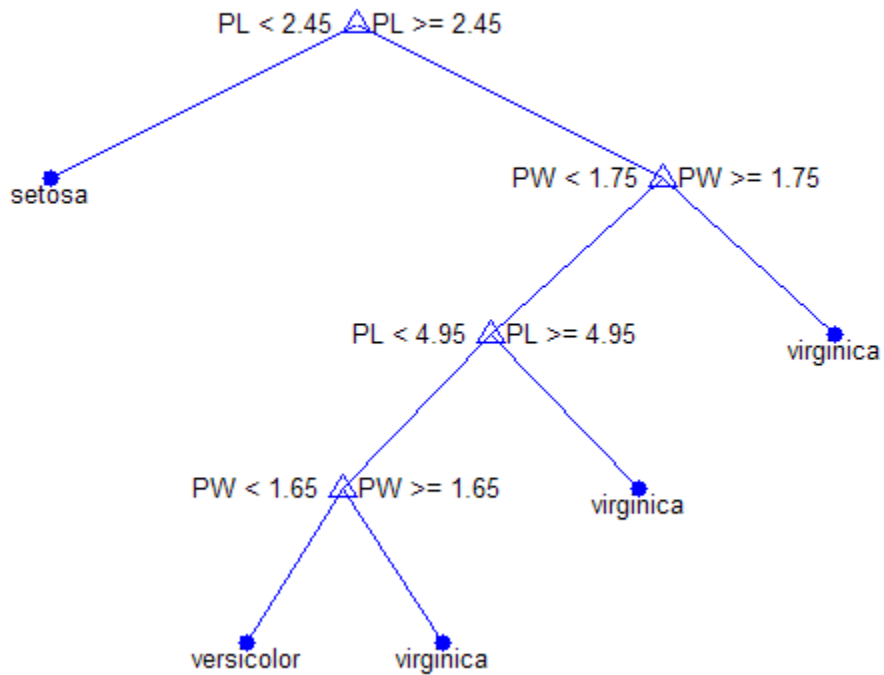
```

7 class = virginica
8 class = versicolor
9 class = virginica

```

```
view(t)
```

Click to display:  Magnification:  Pruning level:



```
C = children(t)
```

```

C =
  2  3
  0  0
  4  5
  6  7
  0  0
  8  9
  0  0

```

0 0  
0 0

## References

- [1] Breiman, L., J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Boca Raton, FL: CRC Press, 1984.

## See Also

`classregtree` | `numnodes` | `parent`

## cholcov

Cholesky-like covariance decomposition

### Syntax

```
T = cholcov(SIGMA)
[T, num] = cholcov(SIGMA)
[T, num] = cholcov(SIGMA, 0)
```

### Description

`T = cholcov(SIGMA)` computes `T` such that  $SIGMA = T' * T$ . `SIGMA` must be square, symmetric, and positive semi-definite. If `SIGMA` is positive definite, then `T` is the square, upper triangular Cholesky factor. If `SIGMA` is not positive definite, `T` is computed from an eigenvalue decomposition of `SIGMA`. `T` is not necessarily triangular or square in this case. Any eigenvectors whose corresponding eigenvalue is close to zero (within a small tolerance) are omitted. If any remaining eigenvalues are negative, `T` is empty.

`[T, num] = cholcov(SIGMA)` returns the number `num` of negative eigenvalues of `SIGMA`, and `T` is empty if `num` is positive. If `num` is zero, `SIGMA` is positive semi-definite. If `SIGMA` is not square and symmetric, `num` is NaN and `T` is empty.

`[T, num] = cholcov(SIGMA, 0)` returns `num` equal to zero if `SIGMA` is positive definite, and `T` is the Cholesky factor. If `SIGMA` is not positive definite, `num` is a positive integer and `T` is empty. `[...] = cholcov(SIGMA, 1)` is equivalent to `[...] = cholcov(SIGMA)`.

### Examples

The following 4-by-4 covariance matrix is rank-deficient:

```
C1 = [2 1 1 2; 1 2 1 2; 1 1 2 2; 2 2 2 3]
C1 =
     2     1     1     2
```



```

      1      2      1      2
      1      1      2      2
      2      2      2      3
rank(C1)
ans =
      3

```

Use cholcov to factor C1:

```

T = cholcov(C1)
T =
    -0.2113    0.7887   -0.5774         0
     0.7887   -0.2113   -0.5774         0
     1.1547    1.1547    1.1547    1.7321

C2 = T'*T
C2 =
     2.0000     1.0000     1.0000     2.0000
     1.0000     2.0000     1.0000     2.0000
     1.0000     1.0000     2.0000     2.0000
     2.0000     2.0000     2.0000     3.0000

```

Use T to generate random data with the specified covariance:

```

C3 = cov(randn(1e6,3)*T)
C3 =
     1.9973     0.9982     0.9995     1.9975
     0.9982     1.9962     0.9969     1.9956
     0.9995     0.9969     1.9980     1.9972
     1.9975     1.9956     1.9972     2.9951

```

## See Also

chol | cov

Introduced in R2007a

## CIIsNonEmpty property

**Class:** NaiveBayes

Flag for non-empty classes

### Description

The `CIIsNonEmpty` property is a logical vector of length `NClasses` specifying which classes are not empty. When the grouping variable is categorical, it may contain categorical levels that don't appear in the elements of the grouping variable. Those levels are empty and `NaiveBayes` ignores them for the purposes of training the classifier.

# classcount

**Class:** classregtree

Class counts

---

**Note** classregtree will be removed in a future release. See fitctree, fitrtree, ClassificationTree, or RegressionTree instead.

---

## Syntax

```
P = classcount(t)
P = classcount(t,nodes)
```

## Description

`P = classcount(t)` returns an  $n$ -by- $m$  array `P` of class counts for the nodes in the classification tree `t`, where  $n$  is the number of nodes and  $m$  is the number of classes. For any node number  $i$ , the class counts `P(i, :)` are counts of observations (from the data used in fitting the tree) from each class satisfying the conditions for node  $i$ .

`P = classcount(t,nodes)` takes a vector `nodes` of node numbers and returns the class counts for the specified nodes.

## Examples

Create a classification tree for Fisher's iris data:

```
load fisheriris;

t = classregtree(meas,species,...
                'names',{'SL' 'SW' 'PL' 'PW'})
t =
Decision tree for classification
1  if PL<2.45 then node 2 elseif PL>=2.45 then node 3 else setosa
2  class = setosa
3  if PW<1.75 then node 4 elseif PW>=1.75 then node 5 else versicolor
```

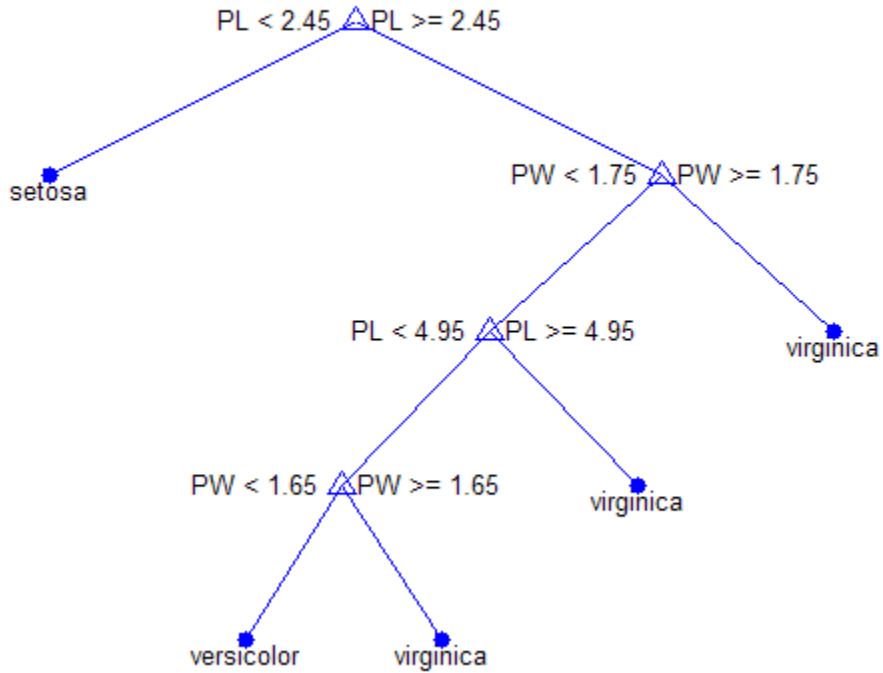
```

4 if PL<4.95 then node 6 elseif PL>=4.95 then node 7 else versicolor
5 class = virginica
6 if PW<1.65 then node 8 elseif PW>=1.65 then node 9 else versicolor
7 class = virginica
8 class = versicolor
9 class = virginica

```

view(t)

Click to display:  Magnification:  Pruning level:



```
P = classcount(t)
```

```

P =
  50   50   50
  50    0    0
   0   50   50
   0   49    5
   0    1   45

```

0	47	1
0	2	4
0	47	0
0	0	1

## References

- [1] Breiman, L., J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Boca Raton, FL: CRC Press, 1984.

## See Also

`classregtree` | `numnodes`

## ClassificationBaggedEnsemble class

Classification ensemble grown by resampling

### Description

`ClassificationBaggedEnsemble` combines a set of trained weak learner models and data on which these learners were trained. It can predict ensemble response for new data by aggregating predictions from its weak learners.

### Construction

Create a bagged classification ensemble object using `fitcensemble`.

### Properties

#### **CategoricalPredictors**

Indices of categorical predictors, stored as a vector of positive integers. `CategoricalPredictors` contains index values corresponding to the columns of the predictor data that contain categorical predictors. If none of the predictors are categorical, then this property is empty (`[]`).

#### **ClassNames**

List of the elements in `Y` with duplicates removed. `ClassNames` can be a numeric vector, categorical vector, logical vector, character array, or cell array of character vectors. `ClassNames` has the same data type as the data in the argument `Y`.

#### **CombineWeights**

Character vector describing how `ens` combines weak learner weights, either `'WeightedSum'` or `'WeightedAverage'`.

**ExpandedPredictorNames**

Expanded predictor names, stored as a cell array of character vectors.

If the model uses encoding for categorical variables, then `ExpandedPredictorNames` includes the names that describe the expanded variables. Otherwise, `ExpandedPredictorNames` is the same as `PredictorNames`.

**FitInfo**

Numeric array of fit information. The `FitInfoDescription` property describes the content of this array.

**FitInfoDescription**

Character vector describing the meaning of the `FitInfo` array.

**FResample**

Numeric scalar between 0 and 1. `FResample` is the fraction of training data `fitcensemble` resampled at random for every weak learner when constructing the ensemble.

**HyperparameterOptimizationResults**

Description of the cross-validation optimization of hyperparameters, stored as a `BayesianOptimization` object or a table of hyperparameters and associated values. Nonempty when the `OptimizeHyperparameters` name-value pair is nonempty at creation. Value depends on the setting of the `HyperparameterOptimizationOptions` name-value pair at creation:

- 'bayesopt' (default) — Object of class `BayesianOptimization`
- 'gridsearch' or 'randomsearch' — Table of hyperparameters used, observed objective function values (cross-validation loss), and rank of observations from lowest (best) to highest (worst)

**Method**

Character vector describing the method that creates `ens`.

**ModelParameters**

Parameters used in training `ens`.

**NumTrained**

Number of trained weak learners in `ens`, a scalar.

**PredictorNames**

Cell array of names for the predictor variables, in the order in which they appear in `X`.

**ReasonForTermination**

Character vector describing the reason `fitcensemble` stopped adding weak learners to the ensemble.

**Replace**

Logical value indicating if the ensemble was trained with replacement (`true`) or without replacement (`false`).

**ResponseName**

Character vector with the name of the response variable `Y`.

**ScoreTransform**

Function handle for transforming scores, or character vector representing a built-in transformation function. `'none'` means no transformation; equivalently, `'none'` means `@(x)x`. For a list of built-in transformation functions and the syntax of custom transformation functions, see `fitctree`.

Add or change a `ScoreTransform` function using dot notation:

```
ens.ScoreTransform = 'function'
```

or

```
ens.ScoreTransform = @function
```

**Trained**

Trained learners, a cell array of compact classification models.



**TrainedWeights**

Numeric vector of trained weights for the weak learners in `ens`. `TrainedWeights` has `T` elements, where `T` is the number of weak learners in `learners`.

**UseObsForLearner**

Logical matrix of size `N`-by-`NumTrained`, where `N` is the number of observations in the training data and `NumTrained` is the number of trained weak learners.

`UseObsForLearner(I, J)` is true if observation `I` was used for training learner `J`, and is false otherwise.

**W**

Scaled `weights`, a vector with length `n`, the number of rows in `X`. The sum of the elements of `W` is 1.

**X**

Matrix of predictor values that trained the ensemble. Each column of `X` represents one variable, and each row represents one observation.

**Y**

A categorical array, cell array of character vectors, character array, logical vector, or a numeric vector with the same number of rows as `X`. Each row of `Y` represents the classification of the corresponding row of `X`.

## Methods

<code>oobEdge</code>	Out-of-bag classification edge
<code>oobLoss</code>	Out-of-bag classification error
<code>oobMargin</code>	Out-of-bag classification margins
<code>oobPermutedPredictorImportance</code>	Predictor importance estimates by permutation of out-of-bag predictor observations for random forest of classification trees
<code>oobPredict</code>	Predict out-of-bag response of ensemble

## Inherited Methods

<code>compact</code>	Compact classification ensemble
<code>crossval</code>	Cross validate ensemble
<code>resubEdge</code>	Classification edge by resubstitution
<code>resubLoss</code>	Classification error by resubstitution
<code>resubMargin</code>	Classification margins by resubstitution
<code>resubPredict</code>	Predict ensemble response by resubstitution
<code>resume</code>	Resume training ensemble
<code>compareHoldout</code>	Compare accuracies of two classification models using new data
<code>edge</code>	Classification edge
<code>loss</code>	Classification error
<code>margin</code>	Classification margins
<code>predict</code>	Predict labels using ensemble of classification models
<code>predictorImportance</code>	Estimates of predictor importance
<code>removeLearners</code>	Remove members of compact classification ensemble

## Copy Semantics

Value. To learn how value classes affect copy operations, see [Copying Objects \(MATLAB\)](#).

## Examples

### Train Bagged Ensemble of Classification Trees

Load the `ionosphere` data set.

```
load ionosphere
```

Train a bagged ensemble of 100 classification trees using all measurements.

```
rng(1) % For reproducibility
Mdl = fitensemble(X,Y,'bag',100,'Tree','Type','classification')

Mdl =

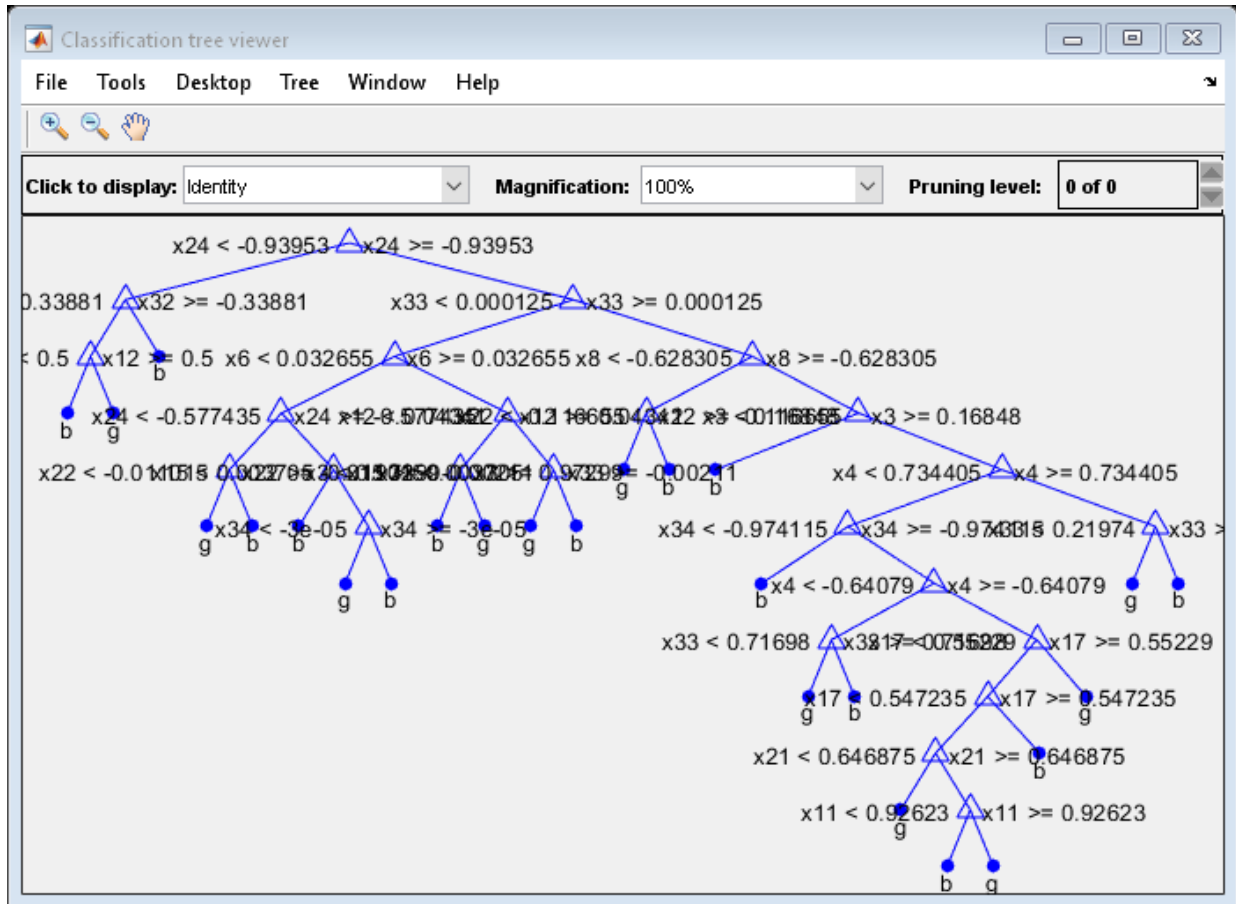
classreg.learning.classif.ClassificationBaggedEnsemble
    ResponseName: 'Y'
    CategoricalPredictors: []
        ClassNames: {'b' 'g'}
        ScoreTransform: 'none'
    NumObservations: 351
    NumTrained: 100
        Method: 'Bag'
    LearnerNames: {'Tree'}
    ReasonForTermination: 'Terminated normally after completing the requested number o
        FitInfo: []
    FitInfoDescription: 'None'
        FResample: 1
        Replace: 1
    UseObsForLearner: [351x100 logical]
```

Mdl is a ClassificationBaggedEnsemble model object.

Mdl.Trained is the property that stores a 100-by-1 cell vector of the trained classification trees (CompactClassificationTree model objects) that compose the ensemble.

Plot a graph of the first trained classification tree.

```
view(Mdl.Trained{1}, 'Mode', 'graph')
```



By default, `fitensemble` grows deep decision trees for bagged ensembles.

Estimate the in-sample misclassification rate.

```
L = resubLoss(Mdl)
```

```
L =
```

```
0
```

`L` is 0, which indicates that `Mdl` is perfect at classifying the training data.

## Tip

For a bagged ensemble of classification trees, the `Trained` property of `ens` stores a cell vector of `ens.NumTrained CompactClassificationTree` model objects. For a textual or graphical display of tree `t` in the cell vector, enter

```
view(ens.Trained{t})
```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Only the `predict` function supports code generation.
- When you train an ensemble by using `fitcensemble`, the following restrictions apply.
  - You can specify only 'Tree' or tree templates (see `templateTree`) as weak learners using the 'Learners' name-value pair argument.
  - Code generation limitations for classification trees also apply to ensembles of classification trees. For more details, see “Code Generation” on page 32-0 of the `CompactClassificationTree` class.

### See Also

`ClassificationEnsemble` | `fitcensemble` | `fitctree` | `view`

### Topics

“Framework for Ensemble Learning” on page 18-41

**Introduced in R2011a**

# ClassificationECOC class

## Superclasses:

Multiclass model for support vector machines or other classifiers

## Description

`ClassificationECOC` is an error-correcting output codes on page 32-394 (ECOC) classifier for multiclass learning by reduction to multiple binary classifiers such as support vector machines (SVMs). Train a `ClassificationECOC` classifier using `fitcecoc` and the training data.

Trained `ClassificationECOC` classifiers store the training data, parameter values, prior probabilities, and coding matrices. You can use these classifiers to:

- Estimate resubstitution predictions. For details, see `resubPredict`.
- Predict labels or posterior probabilities for new data. For details, see `predict`.

## Construction

`Mdl = fitcecoc(Tbl, ResponseVarName)` returns a full, trained, multiclass error-correcting output codes (ECOC) model on page 32-1919 using the predictors in table `Tbl` and the class labels in `Tbl.ResponseVarName`. By default, `fitcecoc` uses  $K(K-1)/2$  binary support vector machine (SVM) models using the one-versus-one coding design on page 32-1915, where  $K$  is the number of unique class labels (levels). `Mdl` is a `ClassificationECOC` model.

`Mdl = fitcecoc(Tbl, formula)` returns an ECOC model using the predictor data and class labels in table `Tbl`. `formula` is an explanatory model of the response and a subset of predictor variables in `Tbl` used for training.

`Mdl = fitcecoc(Tbl, Y)` returns an ECOC model using the predictors in table `Tbl` and the class labels in vector `Y`.

`Mdl = fitcecoc(X, Y)` returns a full trained ECOC model using the predictors `X` and the class labels `Y`.

`Mdl = fitcecoc( ____, Name, Value)` returns an ECOC model with additional options specified by one or more `Name, Value` pair arguments, using any of the previous syntaxes. For example, specify different binary learners, a different coding design, or cross-validation. It is good practice to cross-validate using the `Kfold Name, Value` pair argument. The cross-validation results determine how well the ECOC classifier generalizes.

If you specify one of the following five options, then `Mdl` is a `ClassificationPartitionedECOC` model: `'CrossVal'`, `'CVPartition'`, `'Holdout'`, `'Kfold'`, or `'Leaveout'`. Otherwise, `Mdl` is a `ClassificationECOC` classifier.

## Input Arguments

### **Tbl** — Sample data

table

Sample data, specified as a table. Each row of `Tbl` corresponds to one observation, and each column corresponds to one predictor. Optionally, `Tbl` can contain one additional column for the response variable. Multicolumn variables and cell arrays other than cell arrays of character vectors are not accepted.

If `Tbl` contains the response variable, and you want to use all remaining variables in `Tbl` as predictors, then specify the response variable using `ResponseVarName`.

If `Tbl` contains the response variable, and you want to use only a subset of the remaining variables in `Tbl` as predictors, specify a formula using `formula`.

If `Tbl` does not contain the response variable, specify a response variable using `Y`. The length of response variable and the number of `Tbl` rows must be equal.

---

**Note** For training linear classification models, `fitcecoc` does not support tables. That is, if `Learners` is `'linear'` or contains a linear classification model learner template (see `templateLinear`), you cannot supply `Tbl`, `ResponseVarName`, or `formula`. Supply a matrix of predictor data (`X`) and an array of responses (`Y`) instead.

---



Data Types: `table`

**ResponseVarName** — Response variable name

name of variable in `Tbl`

Response variable name, specified as the name of a variable in `Tbl`.

You must specify `ResponseVarName` as a character vector. For example, if the response variable `Y` is stored as `Tbl.Y`, then specify it as `'Y'`. Otherwise, the software treats all columns of `Tbl`, including `Y`, as predictors when training the model.

The response variable must be a categorical or character array, logical or numeric vector, or cell array of character vectors. If `Y` is a character array, then each element must correspond to one row of the array.

It is good practice to specify the order of the classes using the `ClassNames` name-value pair argument.

Data Types: `char`

**formula** — Explanatory model of response and subset of predictor variables

character vector

Explanatory model of the response and a subset of the predictor variables, specified as a character vector in the form of `'Y~X1+X2+X3'`. In this form, `Y` represents the response variable, and `X1`, `X2`, and `X3` represent the predictor variables. The variables must be variable names in `Tbl` (`Tbl.Properties.VariableNames`).

To specify a subset of variables in `Tbl` as predictors for training the model, use a formula. If you specify a formula, then the software does not use any variables in `Tbl` that do not appear in `formula`.

Data Types: `char`

**y** — Class labels

categorical array | character array | logical vector | vector of numeric values | cell array of character vectors

Class labels to which the ECOC model is trained, specified as a categorical or character array, logical or numeric vector, or cell array of character vectors.

If `Y` is a character array, then each element must correspond to one row of the array.

The length of `Y` and the number of rows of `Tbl` or `X` must be equal.

It is good practice to specify the class order using the `ClassNames` name-value pair argument.

Data Types: `char` | `cell` | `categorical` | `logical` | `single` | `double`

### **x** — Predictor data

matrix of numeric values

Predictor data to which the ECOC classifier is trained, specified as a matrix of numeric values.

- If `Learners` is `'linear'` or contains a linear classification model learner template (see `templateLinear`), the rows of `X` must correspond to predictor (feature) variables, and columns must correspond to observations (instances or examples). This orientation enhances memory-allocation management of high-dimensional data.
- Otherwise, each row of `X` corresponds to one observation (also known as an instance or example), and each column corresponds to one variable (also known as a feature).

Each row of `X` corresponds to one observation (also known as an instance or example), and each column corresponds to one predictor.

The length of `Y` and the number of rows of `X` must be equal.

Data Types: `double` | `single`

---

**Note** The software treats NaN, empty character vector (''), and `<undefined>` elements as missing data. The software removes rows of `X` corresponding to missing values in `Y`. However, the treatment of missing values in `X` varies among binary learners. For details, see the training functions for your binary learners: `fitcdiscr`, `fitcknn`, `fitclinear`, `fitcnb`, `fitcsvm`, `fitctree`, or `fitcensemble`. Removing observations decreases the effective training or cross-validation sample size.

---

## Properties

### **BinaryLearners** — Trained binary learners

cell vector of model objects

Trained binary learners, specified as a cell vector of model objects. `BinaryLearners` has as many elements as classes in `Y`.

The software trained `BinaryLearner{j}` according to the binary problem specified by `CodingMatrix(:,j)`. For example, for multiclass learning using SVM learners, each element of `BinaryLearners` is a `CompactClassificationSVM` classifier.

Data Types: `cell`

### **BinaryLoss** — Binary learner loss function

character vector representing the loss function name

Binary learner loss function, specified as a character vector representing the loss function name.

If you train using binary learners that use different loss functions, then the software sets `BinaryLoss` to `'hamming'`. To potentially increase accuracy, set a different binary loss function than this default during prediction or loss computation using the `BinaryLoss` name-value pair argument of `predict` or `loss`.

Data Types: `char`

### **BinaryY** — Binary learner class labels

numeric matrix

Binary learner class labels, specified as a numeric matrix. `BinaryY` is a `NumObservations-by-L` matrix, where `L` is the number of binary learners (`size(CodingMatrix,2)`).

Elements of `BinaryY` are `-1`, `0`, or `1`, and the value corresponds to a dichotomous class assignment. This table describes how learner `j` assigns observation `k` to a dichotomous class corresponding to the value of `BinaryY(k,j)`.

Value	Dichotomous Class Assignment
-1	Negative class
0	Before training, learner <code>j</code> removes observations in class <code>i</code> from the data set.
1	Positive class

Data Types: `double`

**CategoricalPredictors — Indices of categorical predictors**

vector of positive integers

Indices of categorical predictors, stored as a vector of positive integers. `CategoricalPredictors` contains index values corresponding to the columns of the predictor data that contain categorical predictors. If none of the predictors are categorical, then this property is empty (`[]`).

Data Types: `single` | `double`**ClassNames — Unique class labels**

categorical array | character array | logical vector | vector of numeric values | cell array of character vectors

Unique class labels used in training, specified as a categorical or character array, logical or numeric vector, or cell array of character vectors. `ClassNames` has the same data type as class labels (`Y`). `ClassNames` also determines class order.

Data Types: `categorical` | `char` | `logical` | `single` | `double` | `cell`**CodingMatrix — Class-assignment codes**

numeric matrix

Class-assignment codes for the binary learners, specified as a numeric matrix. `CodingMatrix` is a  $K$ -by- $L$  matrix, where  $K$  is the number of classes and  $L$  is the number of binary learners.

Elements of `CodingMatrix` are -1, 0, or 1, and the value corresponds to a dichotomous class assignment. This table describes the meaning of `CodingMatrix(i, j)`, that is, the class that learner  $j$  assigns to observations in class  $i$ .

Value	Dichotomous Class Assignment
-1	Negative class
0	Before training, learner $j$ removes observations in class $i$ from the data set.
1	Positive class

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64`**CodingName — Coding design name**

character vector

Coding design name, specified as a character vector. For more details, see “Coding Design” on page 32-395.

Data Types: `char`

### **Cost — Misclassification costs**

square numeric matrix

This property is read-only.

Misclassification costs, specified as a square numeric matrix. `Cost` has  $K$  rows and columns, where  $K$  is the number of classes.

`Cost(i, j)` is the cost of classifying a point into class  $j$  if its true class is  $i$ . The order of the rows and columns of `Cost` corresponds to the order of the classes in `ClassNames`.

`fitcecoc` incorporates misclassification costs differently among different types of binary learners.

Data Types: `double`

### **ExpandedPredictorNames — Expanded predictor names**

cell array of character vectors

Expanded predictor names, stored as a cell array of character vectors.

If the model uses encoding for categorical variables, then `ExpandedPredictorNames` includes the names that describe the expanded variables. Otherwise, `ExpandedPredictorNames` is the same as `PredictorNames`.

### **HyperparameterOptimizationResults — Description of the cross-validation optimization of hyperparameters**

`BayesianOptimization` object | table of hyperparameters and associated values

Description of the cross-validation optimization of hyperparameters, stored as a `BayesianOptimization` object or a table of hyperparameters and associated values. Nonempty when the `OptimizeHyperparameters` name-value pair is nonempty at creation. Value depends on the setting of the `HyperparameterOptimizationOptions` name-value pair at creation:

- `'bayesopt'` (default) — Object of class `BayesianOptimization`

- 'gridsearch' or 'randomsearch' — Table of hyperparameters used, observed objective function values (cross-validation loss), and rank of observations from lowest (best) to highest (worst)

Data Types: `table`

### **LearnerWeights** — Binary learner weights

numeric row vector

Binary learner weights, specified as a numeric row vector. `LearnerWeights` has length equal to the number of binary learners (`size(CodingMatrix,2)`).

`LearnerWeights(j)` is the sum of the observation weights that binary learner  $j$  used to train its classifier.

The software uses `LearnerWeights` to fit posterior probabilities by minimizing the Kullback-Leibler divergence.

Data Types: `double` | `single`

### **ModelParameters** — Parameter values

object

Parameter values, e.g., the name-value pair argument values, used to train the ECOC classifier, specified as an object. `ModelParameters` does not contain estimated parameters.

Access properties of `ModelParameters` using dot notation. For example, list the templates containing parameters the binary learners using `Mdl.ModelParameters.BinaryLearner`.

### **NumObservations** — Number of observations

positive numeric scalar

Number of observations in the training data, specified as a positive numeric scalar.

Data Types: `double`

### **PredictorNames** — Predictor names

cell array of character vectors

Predictor names in their order of appearance in predictor data ( $X$ ), specified as a cell array of character vectors. `PredictorNames` has length equal to the number of columns in  $X$ .

Data Types: `cell`

**Prior** — Prior class probabilities

numeric vector

This property is read-only.

Prior class probabilities, specified as a numeric vector. `Prior` has as many elements as classes in `ClassNames`, and the order of the elements corresponds to the elements of `ClassNames`.

`fitcecoc` incorporates misclassification costs differently among different types of binary learners.

Data Types: `double`

**ResponseName** — Response variable name

character array

Response variable name, specified as a character array.

Data Types: `char`

**ScoreTransform** — Score transformation function to apply to predicted scores

character vector | function handle

Score transformation function to apply to predicted scores, specified as a character vector or function handle.

To change the score transformation function to, for example, *function*, use dot notation.

- For a built-in function, enter this code and replace *function* with a character vector in the table.

```
Mdl.ScoreTransform = 'function';
```

Value	Description
'doublelogit'	$1/(1 + e^{-2x})$

Value	Description
'invlogit'	$\log(x / (1-x))$
'ismax'	Set the score for the class with the largest score to 1, and set the scores for all other classes to 0.
'logit'	$1/(1 + e^{-x})$
'none' or 'identity'	$x$ (no transformation)
'sign'	-1 for $x < 0$ 0 for $x = 0$ 1 for $x > 0$
'symmetric'	$2x - 1$
'symmetricismax'	Set the score for the class with the largest score to 1, and set the scores for all other classes to -1.
'symmetriclogit'	$2/(1 + e^{-x}) - 1$

- For a MATLAB function, or a function that you define, enter its function handle.

```
Mdl.ScoreTransform = @function;
```

*function* must accept a matrix (the original scores) and return a matrix of the same size (the transformed scores).

Data Types: `char` | `function_handle`

### **w** — Observation weights

numeric vector

Observation weights used to train the classifier, specified as a numeric vector. *w* has `NumObservations` elements.

The software normalizes the weights used for training so that `nansum(W)` is 1.

Data Types: `single` | `double`

### **x** — Unstandardized predictor data used to train the classifier

numeric matrix



Unstandardized predictor data used to train the classifier, specified as a numeric matrix.  $X$  is a NumObservations-by- $p$  matrix, where  $p$  is the number of predictors.

Each row of  $X$  corresponds to one observation, and each column corresponds to one variable.

Data Types: `single` | `double`

### **$Y$ — Observed class labels**

categorical array | character array | logical vector | numeric vector | cell array of character vectors

Observed class labels used to train the classifier, specified as a categorical or character array, logical or numeric vector, or cell array of character vectors.  $Y$  has NumObservations elements, and is the same data type as the input argument  $Y$  of `fitcecoc`.

Each row of  $Y$  represents the observed classification of the corresponding row of `Tbl` or  $X$ .

## Methods

<code>compact</code>	Compact multiclass, error-correcting output codes model
<code>crossval</code>	Cross-validate multiclass, error-correcting output codes model
<code>resubEdge</code>	Classification edge by resubstitution for multiclass, error-correcting output codes model
<code>resubLoss</code>	Classification loss by resubstitution for multiclass, error-correcting output codes model
<code>resubMargin</code>	Classification margins by resubstitution for multiclass, error-correcting output codes model
<code>resubPredict</code>	Predict resubstitution responses for multiclass, error-correcting output codes model

## Inherited Methods

<code>compareHoldout</code>	Compare accuracies of two classification models using new data
<code>discardSupportVectors</code>	Discard support vectors of linear support vector machine binary learners
<code>edge</code>	Classification edge for multiclass, error-correcting output codes model
<code>loss</code>	Classification loss for multiclass, error-correcting output codes model
<code>margin</code>	Classification margins for multiclass, error-correcting output codes model
<code>predict</code>	Predict labels using multiclass, error-correcting output codes model
<code>selectModels</code>	Choose subset of regularized linear classification models

## Copy Semantics

Value. To learn how value classes affect copy operations, see [Copying Objects \(MATLAB\)](#).

## Examples

### Train Multiclass Model Using SVM Learners

Train an error-correcting output codes (ECOC) multiclass model using support vector machine (SVM) binary learners.

Load Fisher's iris data set.

```
load fisheriris
X = meas;
Y = species;
```

Train an ECOC multiclass model using the default options.

```
Mdl = fitcecoc(X,Y)
```

```
Mdl =

ClassificationECOC
    ResponseName: 'Y'
    CategoricalPredictors: []
        ClassNames: {'setosa' 'versicolor' 'virginica'}
        ScoreTransform: 'none'
        BinaryLearners: {3x1 cell}
        CodingName: 'onevsone'
```

Mdl is a ClassificationECOC model. By default, `fitcecoc` uses SVM binary learners, and uses a one-versus-one coding design. You can access Mdl properties using dot notation.

Display the coding design matrix.

```
Mdl.ClassNames
CodingMat = Mdl.CodingMatrix
```

```
ans =

3x1 cell array

    {'setosa'    }
    {'versicolor'}
    {'virginica' }
```

```
CodingMat =

     1     1     0
    -1     0     1
     0    -1    -1
```

A one-versus-one coding design on three classes yields three binary learners. Columns of CodingMat correspond to learners, and rows correspond to classes. The class order corresponds to the order in Mdl.ClassNames. For example, CodingMat(:,1) is [1; -1; 0] and indicates that the software trains the first SVM binary learner using all observations classified as 'setosa' and 'versicolor'. Since 'setosa' corresponds to

1, it is the positive class, and since 'versicolor' corresponds to -1, it is the negative class.

You can access each binary learner using cell indexing and dot notation.

```
Mdl.BinaryLearners{1}           % The first binary learner

ans =

classreg.learning.classif.CompactClassificationSVM
    ResponseName: 'Y'
    CategoricalPredictors: []
        ClassNames: [-1 1]
        ScoreTransform: 'none'
            Beta: [4x1 double]
            Bias: 1.4505
        KernelParameters: [1x1 struct]
```

Compute the in-sample classification error.

```
isLoss = resubLoss(Mdl)

isLoss =

    0.0067
```

The classification error is small, but the classifier might have been overfit. You can cross-validate the classifier using `crossval`.

### Inspect Binary Learner Properties of ECOC Classifiers

You can access properties of binary learners, such as estimated parameters, using dot notation.

Load Fisher's iris data set. Use the petal dimensions as predictors.

```
load fisheriris
X = meas(:,3:4);
Y = species;
```

Train an ECOC classifier using the SVM binary learners and the default coding design (one-versus-one). Standardize the predictors and save the support vectors.

```
t = templateSVM('Standardize',1,'SaveSupportVectors',true);
predictorNames = {'petalLength','petalWidth'};
responseName = 'irisSpecies';
classNames = {'setosa','versicolor','virginica'}; % Specify class order
Mdl = fitcecoc(X,Y,'Learners',t,'ResponseName',responseName,...
    'PredictorNames',predictorNames,'ClassNames',classNames)
```

`t` is a template object that contains options for SVM classification. All properties are empty (`[]`), except for `Method`, `SaveSupportVectors`, `StandardizeData`, and `Type`. `fitcecoc` uses the default values of the empty properties. `Mdl` is a `ClassificationECOC` classifier. You can access its properties using dot notation.

Display the class names and coding design matrix.

```
Mdl.ClassNames
Mdl.CodingMatrix
L = size(Mdl.CodingMatrix,2); % Number of SVMs
```

The columns correspond to SVM binary learners, and the rows to the distinct classes. The row order corresponds to the order of the `ClassNames` property of `Mdl`. For each column:

- A 1 indicates that `fitcecoc` trained the SVM using observations in the corresponding class as members of the positive group.
- A -1 indicates that `fitcecoc` trained the SVM using observations in the corresponding class as members of the negative group.
- A 0 indicates that the SVM did not use observations in the corresponding class.

For example, `fitcecoc` assigns all observations to 'setosa' or 'versicolor', but not 'virginica' in the first SVM.

You can access properties of the SVMs using cell subscripting and dot notation. Store the standardized support vectors of each SVM. Unstandardize the support vectors.

```
sv = cell(L,1); % Preallocate for support vector indices
for j = 1:L;
```

```
SVM = Mdl.BinaryLearners{j};
sv{j} = SVM.SupportVectors;
sv{j} = bsxfun(@plus,bsxfun(@times,sv{j},SVM.Sigma),SVM.Mu);
end
```

`sv` is a cell array of matrices containing the unstandardized support vectors for the SVMs.

Plot the data, and identify the support vectors.

```
figure;
h = zeros(3 + L,1); % Preallocate for handles
h(1:3) = gscatter(X(:,1),X(:,2),Y);
hold on;
markers = {'ko','ro','bo'};
for j = 1:L;
    svj = sv{j};
    h(j + 3) = plot(svj(:,1),svj(:,2),markers{j},...
        'MarkerSize',10 + (j - 1)*3);
end
title 'Fisher''s Iris -- ECOC Support Vectors';
xlabel(predictorNames{1});
ylabel(predictorNames{2});
legend(h,[classNames,{'Support vectors - SVM 1',...
    'Support vectors - SVM 2','Support vectors - SVM 3'}],...
    'Location','Best')
hold off
```

You can pass `Mdl` to:

- `predict` to classify new observations
- `resubLoss` to estimate the in-sample classification error
- `crossval` to perform 10-fold cross validation.

### Cross-Validate ECOC Classifier

Load Fisher's iris data set.

```
load fisheriris
X = meas;
Y = species;
rng(1); % For reproducibility
```

Create an SVM template. It is good practice to standardize the predictors.

```
t = templateSVM('Standardize',1)
```

```
t =
```

```
Fit template for classification SVM.
```

```

                Alpha: [0x1 double]
    BoxConstraint: []
           CacheSize: []
    CachingMethod: ''
           ClipAlphas: []
DeltaGradientTolerance: []
           Epsilon: []
           GapTolerance: []
           KKTolerance: []
           IterationLimit: []
           KernelFunction: ''
           KernelScale: []
           KernelOffset: []
KernelPolynomialOrder: []
           NumPrint: []
                Nu: []
           OutlierFraction: []
RemoveDuplicates: []
           ShrinkagePeriod: []
           Solver: ''
           StandardizeData: 1
SaveSupportVectors: []
           VerbosityLevel: []
           Version: 2
           Method: 'SVM'
           Type: 'classification'

```

`t` is an SVM template. All of its properties are empty, except for `StandardizeData`, `Method`, and `Type`. When the software trains the ECOC classifier, it sets the applicable properties to their default values.

Train the ECOC classifier. It is good practice to specify the class order.

```
Mdl = fitcecoc(X,Y,'Learners',t,...
    'ClassNames',{'setosa','versicolor','virginica'});
```

Mdl is a `ClassificationECOC` classifier. You can access its properties using dot notation.

Cross-validate Mdl using 10-fold cross-validation.

```
CVMDL = crossval(Mdl);
```

CVMDL is a `ClassificationPartitionedECOC` cross-validated ECOC classifier.

Estimate the generalization error.

```
oosLoss = kfoldLoss(CVMDL)
```

```
oosLoss =  
    0.0400
```

The out-of-sample classification error is 4%, which indicates that the ECOC classifier generalizes fairly well.

- “Code Generation Support, Usage Notes, and Limitations” on page 31-24
- “Code Generation Workflows” on page 31-2
- “Predict Class Labels Using MATLAB Function Block” on page 31-18
- “C Code Generation for Image Classifier”

## Definitions

### Error-Correcting Output Codes Model

An error-correcting output codes (ECOC) model reduces the problem of classification with three or more classes to a set of binary classifiers.

ECOC classification requires a coding design, which determines the classes that the binary learners train on, and a decoding scheme, which determines how the results (predictions) of the binary classifiers are aggregated.

Suppose that:



- There are three classes.
- The coding design is one-versus-one.
- The decoding scheme uses loss  $g$ .
- The learners are SVMs.

To build this classification model, ECOC follows these steps.

- 1 A one-versus-one coding design is

	Learner 1	Learner 2	Learner 3
Class 1	1	1	0
Class 2	-1	0	1
Class 3	0	-1	-1

Learner 1 trains on observations having Class 1 and Class 2, and treats Class 1 as the positive class and Class 2 as the negative class. The other learners are trained similarly. Let  $M$  be the coding design matrix with elements  $m_{kl}$ , and  $s_l$  be the predicted classification score for the positive class of learner  $l$ .

- 2 A new observation is assigned to the class ( $\hat{k}$ ) that minimizes the aggregation of the losses for the  $L$  binary learners. That is,

$$\hat{k} = \underset{k}{\operatorname{argmin}} \frac{\sum_{l=1}^L |m_{kl}| g(m_{kl}, s_l)}{\sum_{l=1}^L |m_{kl}|}.$$

ECOC models can improve classification accuracy, even compared to other multiclass models [2].

## Coding Design

A coding design is a matrix where elements direct which classes are trained by each binary learner, that is, how the multiclass problem is reduced to a series of binary problems.

Each row of the coding design corresponds to a distinct class, and each column corresponds to a binary learner. In a ternary coding design (adopted by the software), for a particular column (or binary learner):

- Rows containing a 1 indicate to the binary learner to group all observations in the corresponding classes into a positive class.
- Rows containing a -1 indicate to the binary learner to group all observations in the corresponding classes into a negative class.
- Rows containing a 0 indicate to the binary learner to ignore all observations in the corresponding classes.

Coding matrices with large, minimal, pair-wise row distances based on the Hamming measure are desirable. For details on the pair-wise row distance measure, see “Random Coding Design Matrices” on page 32-1923 and [4].

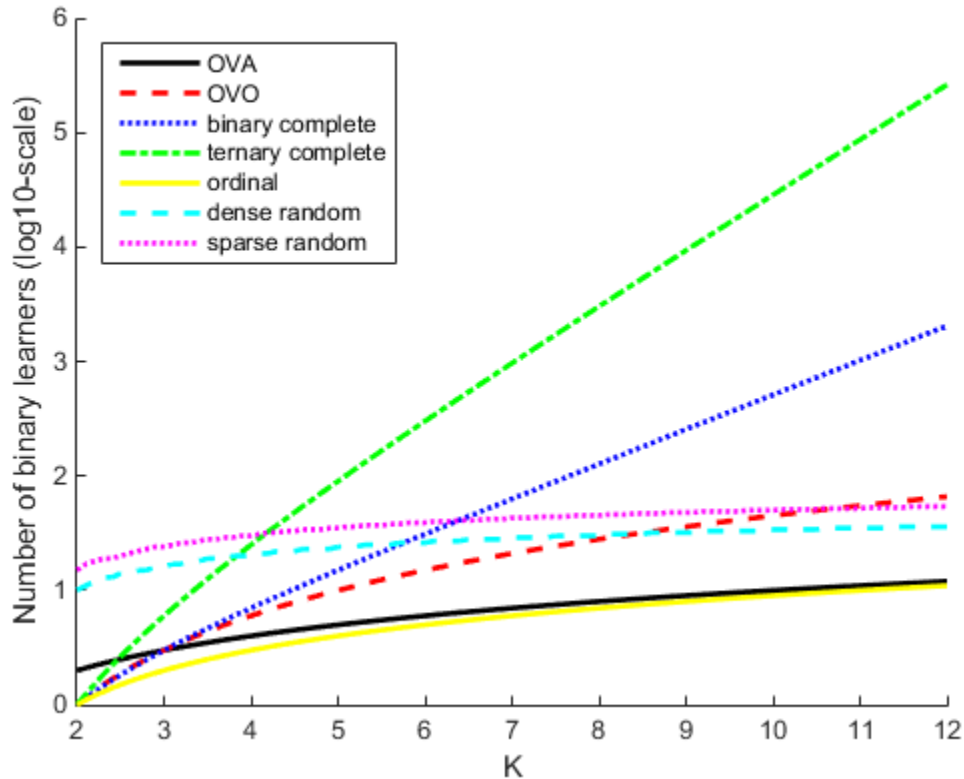
This table describes popular coding designs. For the example, suppose  $K$  (the number of distinct classes) is 3.

Coding Design	Description	Number of Learners	Minimal Pair-Wise Row Distance
one-versus-all (OVA)	For each binary learner, one class is positive and the rest are negative. This design exhausts all combinations of positive class assignments.	$K$	2
one-versus-one (OVO)	For each binary learner, one class is positive, another is negative, and the rest are ignored. This design exhausts all combinations of class pair assignments.	$K(K - 1)/2$	1

Coding Design	Description	Number of Learners	Minimal Pair-Wise Row Distance
binary complete	This design partitions the classes into all binary combinations, and does not ignore any classes. That is, all class assignments are -1 and 1 with at least one positive and negative class in the assignment for each binary learner.	$2^{K-1} - 1$	$2^{K-2}$
ternary complete	This design partitions the classes into all ternary combinations. That is, all class assignments are 0, -1, and 1 with at least one positive and negative class in the assignment for each binary learner.	$(3^K - 2^{K+1} + 1)/2$	$3^{K-2}$
ordinal	For the first binary learner, the first class is negative, and the rest positive. For the second binary learner, the first two classes are negative, and the rest positive, and so on.	$K - 1$	1

Coding Design	Description	Number of Learners	Minimal Pair-Wise Row Distance
dense random	For each binary learner, the software randomly assigns classes into positive or negative classes, with at least one of each type. For more details, see “Random Coding Design Matrices” on page 32-1923.	Random, but approximately $10 \log_2 K$	Variable
sparse random	For each binary learner, the software randomly assigns classes as positive or negative with probability 0.25 for each, and ignores classes with probability 0.5. For more details, see “Random Coding Design Matrices” on page 32-1923.	Random, but approximately $15 \log_2 K$	Variable

This plot compares the number of binary learners for the coding designs with increasing  $K$ .



## Algorithms

### Random Coding Design Matrices

For a given number of classes, for example,  $K$ , the software generates random coding design matrices as follows.

- 1 The software generates one of the following:
  - a Dense random — The software assigns a 1 or -1 with equal probability to each element of the  $K$ -by- $L_d$  coding design matrix, where  $L_d \approx \lceil 10 \log_2 K \rceil$ .

- b** Sparse random — The software assigns a 1 to each element of the  $K$ -by- $L_s$  coding design matrix with probability 0.25, a -1 with probability 0.25, and a 0 with probability 0.5, where  $L_s \approx \lceil 15 \log_2 K \rceil$ .
- 2** If a column does not contain at least one 1 and at least one -1, then the software removes that column.
- 3** For distinct columns  $u$  and  $v$ , if  $u = v$  or  $u = -v$ , then the software removes  $v$  from the coding design matrix.

The software randomly generates 10,000 matrices by default, and retains the matrix with the largest, minimal pairwise row distance based on the Hamming measure ([4]) given by

$$\Delta(k_1, k_2) = 0.5 \sum_{l=1}^L |m_{k_1 l} - m_{k_2 l}|,$$

where  $m_{k,l}$  is an element of coding design matrix  $j$ .

## Support Vector Storage

For linear, SVM binary learners, and for efficiency, `fitcecoc` empties the properties `Alpha`, `SupportVectorLabels`, and `SupportVectors`. `fitcecoc` lists `Beta`, rather than `Alpha`, in the model display.

To store `Alpha`, `SupportVectorLabels`, and `SupportVectors`, pass a linear, SVM template that specifies storing support vectors to `fitcecoc`. For example, enter:

```
t = templateSVM('SaveSupportVectors', 'on')
Mdl = fitcecoc(X, Y, 'Learners', t);
```

You can remove the support vectors and related values by passing the resulting `ClassificationECOC` model to `discardSupportVectors`.

## Alternatives

You can use these alternative algorithms to train a multiclass model:

- Classification ensembles—see `fitcensemble` and `ClassificationEnsemble`
- Classification trees—see `fitctree` and `ClassificationTree`

- Discriminant analysis classifiers—see `fitcdiscr` and `ClassificationDiscriminant`
- $k$ -nearest neighbor classifiers—see `fitcknn` and `ClassificationKNN`
- Naive Bayes classifiers—see `fitcnb` and `ClassificationNaiveBayes`

## References

- [1] Fürnkranz, Johannes. “Round Robin Classification.” *Journal of Machine Learning Research*, Vol. 2, 2002, pp. 721–747.
- [2] Escalera, S., O. Pujol, and P. Radeva. “Separability of ternary codes for sparse designs of error-correcting output codes.” *Pattern Recognition Letters*, Vol. 30, Issue 3, 2009, pp. 285–297.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Only the `predict` function supports code generation.
- When you train an ECOC model by using `fitcecoc`, the following restrictions apply.
  - You cannot fit posterior probabilities by using the 'FitPosterior' name-value pair argument.
  - All binary learners must be either SVM classifiers or linear classification models. For the 'Learners' name-value pair argument, you can specify:
    - 'svm' or 'linear'
    - An SVM template object or a cell vector of such objects (see `templateSVM`)
    - A linear classification model template object or a cell vector of such objects (see `templateLinear`)

- Code generation limitations for SVM classifiers and linear classification models also apply to ECOC classifiers, depending on the choice of binary learners. For more details, see “Code Generation” on page 32-0 of the `ClassificationSVM` class and “Code Generation” on page 32-0 of the `ClassificationLinear` class.

## See Also

`ClassificationDiscriminant` | `ClassificationEnsemble` | `ClassificationKNN`  
| `ClassificationLinear` | `ClassificationNaiveBayes` |  
`ClassificationPartitonedECOC` | `ClassificationTree` |  
`CompactClassificationECOC` | `fitcdiscr` | `fitcecoc` | `fitcensemble` | `fitcknn`  
| `fitclinear` | `fitcnb` | `fitcsvm` | `fitctree`

## Topics

“Code Generation Support, Usage Notes, and Limitations” on page 31-24

“Code Generation Workflows” on page 31-2

“Predict Class Labels Using MATLAB Function Block” on page 31-18

“C Code Generation for Image Classifier”

**Introduced in R2014b**



# ClassificationDiscriminant class

## Superclasses:

Discriminant analysis classification

## Description

A `ClassificationDiscriminant` object encapsulates a discriminant analysis classifier, which is a Gaussian mixture model for data generation. A `ClassificationDiscriminant` object can predict responses for new data using the `predict` method. The object contains the data used for training, so can compute resubstitution predictions.

## Construction

`Mdl = fitcdiscr(Tbl, ResponseVarName)` returns a fitted discriminant analysis model based on the input variables (also known as predictors, features, or attributes) contained in the table `Tbl` and output (response or labels) contained in `ResponseVarName`.

`Mdl = fitcdiscr(Tbl, formula)` returns a fitted discriminant analysis model based on the predictor data and class labels in the table `Tbl`. `formula` is an explanatory model of the response and a subset of predictor variables in `Tbl` used to fit `Mdl`.

`Mdl = fitcdiscr(Tbl, Y)` returns a fitted discriminant analysis model based on the input variables contained in the table `Tbl` and response `Y`.

`Mdl = fitcdiscr(X, Y)` returns a discriminant analysis classifier based on the input variables `X` and response `Y`.

`Mdl = fitcdiscr( ____, Name, Value)` fits a classifier with additional options specified by one or more name-value pair arguments, using any of the previous syntaxes. For example, you can specify the cost of misclassification, prior probabilities for each class, or observation weights.

## Input Arguments

### **Tbl** — Sample data

table

Sample data used to train the model, specified as a table. Each row of `Tbl` corresponds to one observation, and each column corresponds to one predictor variable. Optionally, `Tbl` can contain one additional column for the response variable. Multi-column variables and cell arrays other than cell arrays of character vectors are not allowed.

If `Tbl` contains the response variable, and you want to use all remaining variables in `Tbl` as predictors, then specify the response variable using `ResponseVarName`.

If `Tbl` contains the response variable, and you want to use only a subset of the remaining variables in `Tbl` as predictors, then specify a formula using `formula`.

If `Tbl` does not contain the response variable, then specify a response variable using `Y`. The length of response variable and the number of rows of `Tbl` must be equal.

Data Types: `table`

### **ResponseVarName** — Response variable name

name of variable in `Tbl`

Response variable name, specified as the name of a variable in `Tbl`.

You must specify `ResponseVarName` as a character vector. For example, if the response variable `Y` is stored as `Tbl.Y`, then specify it as `'Y'`. Otherwise, the software treats all columns of `Tbl`, including `Y`, as predictors when training the model.

The response variable must be a categorical or character array, logical or numeric vector, or cell array of character vectors. If `Y` is a character array, then each element must correspond to one row of the array.

It is good practice to specify the order of the classes using the `ClassNames` name-value pair argument.

Data Types: `char`

### **formula** — Explanatory model of response and subset of predictor variables

character vector

Explanatory model of the response and a subset of the predictor variables, specified as a character vector in the form of 'Y~X1+X2+X3'. In this form, Y represents the response variable, and X1, X2, and X3 represent the predictor variables. The variables must be variable names in Tbl (Tbl.Properties.VariableNames).

To specify a subset of variables in Tbl as predictors for training the model, use a formula. If you specify a formula, then the software does not use any variables in Tbl that do not appear in formula.

Data Types: char

### **Y — Class labels**

numeric vector | categorical vector | logical vector | character array | cell array of character vectors

Class labels, specified as a categorical or character array, logical or numeric vector, or cell array of character vectors. Each row of Y represents the classification of the corresponding row of X.

The software considers NaN, '' (empty character vector), and <undefined> values in Y to be missing values. Consequently, the software does not train using observations with a missing response.

Data Types: single | double | logical | char | cell

### **X — Predictor data**

numeric matrix

Predictor values, specified as a numeric matrix. Each column of X represents one variable, and each row represents one observation.

fitcdiscr considers NaN values in X as missing values. fitcdiscr does not use observations with missing values for X in the fit.

Data Types: single | double

## Properties

### **BetweenSigma**

p-by-p matrix, the between-class covariance, where p is the number of predictors.

**CategoricalPredictors**

Categorical predictor indices, which is always empty (`[]`).

**ClassNames**

List of the elements in the training data `Y` with duplicates removed. `ClassNames` can be a categorical array, cell array of character vectors, character array, logical vector, or a numeric vector. `ClassNames` has the same data type as the data in the argument `Y`.

**Coeffs**

`k`-by-`k` structure of coefficient matrices, where `k` is the number of classes. `Coeffs(i, j)` contains coefficients of the linear or quadratic boundaries between classes `i` and `j`. Fields in `Coeffs(i, j)`:

- `DiscrimType`
- `Class1` — `ClassNames(i)`
- `Class2` — `ClassNames(j)`
- `Const` — A scalar
- `Linear` — A vector with `p` components, where `p` is the number of columns in `X`
- `Quadratic` — `p`-by-`p` matrix, exists for quadratic `DiscrimType`

The equation of the boundary between class `i` and class `j` is

$$\text{Const} + \text{Linear} * x + x' * \text{Quadratic} * x = 0,$$

where `x` is a column vector of length `p`.

If `fitcdiscr` had the `FillCoeffs` name-value pair set to `'off'` when constructing the classifier, `Coeffs` is empty (`[]`).

**Cost**

Square matrix, where `Cost(i, j)` is the cost of classifying a point into class `j` if its true class is `i` (i.e., the rows correspond to the true class and the columns correspond to the predicted class). The order of the rows and columns of `Cost` corresponds to the order of the classes in `ClassNames`. The number of rows and columns in `Cost` is the number of unique classes in the response.

Change a Cost matrix using dot notation: `obj.Cost = costMatrix`.

### **Delta**

Value of the Delta threshold for a linear discriminant model, a nonnegative scalar. If a coefficient of `obj` has magnitude smaller than `Delta`, `obj` sets this coefficient to 0, and so you can eliminate the corresponding predictor from the model. Set `Delta` to a higher value to eliminate more predictors.

`Delta` must be 0 for quadratic discriminant models.

Change `Delta` using dot notation: `obj.Delta = newDelta`.

### **DeltaPredictor**

Row vector of length equal to the number of predictors in `obj`. If `DeltaPredictor(i) < Delta` then coefficient `i` of the model is 0.

If `obj` is a quadratic discriminant model, all elements of `DeltaPredictor` are 0.

### **DiscrimType**

Character vector specifying the discriminant type. One of:

- 'linear'
- 'quadratic'
- 'diagLinear'
- 'diagQuadratic'
- 'pseudoLinear'
- 'pseudoQuadratic'

Change `DiscrimType` using dot notation: `obj.DiscrimType = newDiscrimType`.

You can change between linear types, or between quadratic types, but cannot change between linear and quadratic types.

### **Gamma**

Value of the Gamma regularization parameter, a scalar from 0 to 1. Change `Gamma` using dot notation: `obj.Gamma = newGamma`.

- If you set 1 for linear discriminant, the discriminant sets its type to 'diagLinear'.
- If you set a value between `MinGamma` and 1 for linear discriminant, the discriminant sets its type to 'linear'.
- You cannot set values below the value of the `MinGamma` property.
- For quadratic discriminant, you can set either 0 (for `DiscrimType` 'quadratic') or 1 (for `DiscrimType` 'diagQuadratic').

### **HyperparameterOptimizationResults**

Description of the cross-validation optimization of hyperparameters, stored as a `BayesianOptimization` object or a table of hyperparameters and associated values. Nonempty when the `OptimizeHyperparameters` name-value pair is nonempty at creation. Value depends on the setting of the `HyperparameterOptimizationOptions` name-value pair at creation:

- 'bayesopt' (default) — Object of class `BayesianOptimization`
- 'gridsearch' or 'randomsearch' — Table of hyperparameters used, observed objective function values (cross-validation loss), and rank of observations from lowest (best) to highest (worst)

### **LogDetSigma**

Logarithm of the determinant of the within-class covariance matrix. The type of `LogDetSigma` depends on the discriminant type:

- Scalar for linear discriminant analysis
- Vector of length `K` for quadratic discriminant analysis, where `K` is the number of classes

### **MinGamma**

Nonnegative scalar, the minimal value of the `Gamma` parameter so that the correlation matrix is invertible. If the correlation matrix is not singular, `MinGamma` is 0.

### **ModelParameters**

Parameters used in training `obj`.

**Mu**

Class means, specified as a  $K$ -by- $p$  matrix of scalar values class means of size.  $K$  is the number of classes, and  $p$  is the number of predictors. Each row of `Mu` represents the mean of the multivariate normal distribution of the corresponding class. The class indices are in the `ClassNames` attribute.

**NumObservations**

Number of observations in the training data, a numeric scalar. `NumObservations` can be less than the number of rows of input data  $X$  when there are missing values in  $X$  or response  $Y$ .

**PredictorNames**

Cell array of names for the predictor variables, in the order in which they appear in the training data  $X$ .

**Prior**

Numeric vector of prior probabilities for each class. The order of the elements of `Prior` corresponds to the order of the classes in `ClassNames`.

Add or change a `Prior` vector using dot notation: `obj.Prior = priorVector`.

**ResponseName**

Character vector describing the response variable  $Y$ .

**ScoreTransform**

Function handle for transforming scores, or character vector representing a built-in transformation function. 'none' means no transformation; equivalently, 'none' means  $@(x)x$ . For a list of built-in transformation functions and the syntax of custom transformation functions, see `fitcdiscr`.

Implement dot notation to add or change a `ScoreTransform` function using one of the following:

- `cobj.ScoreTransform = 'function'`
- `cobj.ScoreTransform = @function`

**Sigma**

Within-class covariance matrix or matrices. The dimensions depend on `DiscrimType`:

- 'linear' (default) — Matrix of size  $p$ -by- $p$ , where  $p$  is the number of predictors
- 'quadratic' — Array of size  $p$ -by- $p$ -by- $K$ , where  $K$  is the number of classes
- 'diagLinear' — Row vector of length  $p$
- 'diagQuadratic' — Array of size  $1$ -by- $p$ -by- $K$
- 'pseudoLinear' — Matrix of size  $p$ -by- $p$
- 'pseudoQuadratic' — Array of size  $p$ -by- $p$ -by- $K$

**W**

Scaled weights, a vector with length  $n$ , the number of rows in  $X$ .

**X**

Matrix of predictor values. Each column of  $X$  represents one predictor (variable), and each row represents one observation.

**Xcentered**

$X$  data with class means subtracted. If  $Y(i)$  is of class  $j$ ,

$$X_{\text{centered}}(i, :) = X(i, :) - \text{Mu}(j, :),$$

where  $\text{Mu}$  is the class mean property.

**Y**

A categorical array, cell array of character vectors, character array, logical vector, or a numeric vector with the same number of rows as  $X$ . Each row of  $Y$  represents the classification of the corresponding row of  $X$ .



## Methods

compact	Compact discriminant analysis classifier
crossval	Cross-validated discriminant analysis classifier
cvshrink	Cross-validate regularization of linear discriminant
resubEdge	Classification edge by resubstitution
resubLoss	Classification error by resubstitution
resubMargin	Classification margins by resubstitution
resubPredict	Predict resubstitution response of classifier

## Inherited Methods

compareHoldout	Compare accuracies of two classification models using new data
edge	Classification edge
logP	Log unconditional probability density for discriminant analysis classifier
loss	Classification error
mahal	Mahalanobis distance to class means
margin	Classification margins
nLinearCoeffs	Number of nonzero linear coefficients
predict	Predict labels using discriminant analysis classification model

## Copy Semantics

Value. To learn how value classes affect copy operations, see [Copying Objects \(MATLAB\)](#).

## Examples

### Train Discriminant Analysis Model

Load Fisher's iris data set.

```
load fisheriris
```

Train a discriminant analysis model using the entire data set.

```
Mdl = fitcdiscr(meas, species)
```

```
Mdl =
```

```
ClassificationDiscriminant
    ResponseName: 'Y'
    CategoricalPredictors: []
        ClassNames: {'setosa' 'versicolor' 'virginica'}
    ScoreTransform: 'none'
    NumObservations: 150
    DiscrimType: 'linear'
        Mu: [3x4 double]
        Coeffs: [3x3 struct]
```

Mdl is a `ClassificationDiscriminant` model. To access its properties, use dot notation. For example, display the group means for each predictor.

```
Mdl.Mu
```

```
ans =
```

```
    5.0060    3.4280    1.4620    0.2460
    5.9360    2.7700    4.2600    1.3260
    6.5880    2.9740    5.5520    2.0260
```

To predict labels for new observations, pass Mdl and predictor data to `predict`.

## Definitions

### Discriminant Classification

The model for discriminant analysis is:

- Each class ( $Y$ ) generates data ( $X$ ) using a multivariate normal distribution. That is, the model assumes  $X$  has a Gaussian mixture distribution (`gmdistribution`).
- For linear discriminant analysis, the model has the same covariance matrix for each class, only the means vary.
- For quadratic discriminant analysis, both means and covariances of each class vary.

`predict` classifies so as to minimize the expected classification cost:

$$\hat{y} = \arg \min_{y=1, \dots, K} \sum_{k=1}^K \hat{P}(k | x) C(y | k),$$

where

- $\hat{y}$  is the predicted classification.
- $K$  is the number of classes.
- $\hat{P}(k | x)$  is the posterior probability on page 20-8 of class  $k$  for observation  $x$ .
- $C(y | k)$  is the cost on page 20-9 of classifying an observation as  $y$  when its true class is  $k$ .

For details, see “Prediction Using Discriminant Analysis Models” on page 20-8.

## Regularization

Regularization is the process of finding a small set of predictors that yield an effective predictive model. For linear discriminant analysis, there are two parameters,  $\gamma$  and  $\delta$ , that control regularization as follows. `cvshrink` helps you select appropriate values of the parameters.

Let  $\Sigma$  represent the covariance matrix of the data  $X$ , and let  $\hat{X}$  be the centered data (the data  $X$  minus the mean by class). Define

$$D = \text{diag}(\hat{X}^T * \hat{X}).$$

The regularized covariance matrix  $\tilde{\Sigma}$  is

$$\tilde{\Sigma} = (1 - \gamma)\Sigma + \gamma D.$$

Whenever  $\gamma \geq \text{MinGamma}$ ,  $\tilde{\Sigma}$  is nonsingular.

Let  $\mu_k$  be the mean vector for those elements of  $X$  in class  $k$ , and let  $\mu_0$  be the global mean vector (the mean of the rows of  $X$ ). Let  $C$  be the correlation matrix of the data  $X$ , and let

$\tilde{C}$  be the regularized correlation matrix:

$$\tilde{C} = (1 - \gamma)C + \gamma I,$$

where  $I$  is the identity matrix.

The linear term in the regularized discriminant analysis classifier for a data point  $x$  is

$$(x - \mu_0)^T \tilde{\Sigma}^{-1} (\mu_k - \mu_0) = \left[ (x - \mu_0)^T D^{-1/2} \right] \left[ \tilde{C}^{-1} D^{-1/2} (\mu_k - \mu_0) \right].$$

The parameter  $\delta$  enters into this equation as a threshold on the final term in square

brackets. Each component of the vector  $\left[ \tilde{C}^{-1} D^{-1/2} (\mu_k - \mu_0) \right]$  is set to zero if it is smaller in magnitude than the threshold  $\delta$ . Therefore, for class  $k$ , if component  $j$  is thresholded to zero, component  $j$  of  $x$  does not enter into the evaluation of the posterior probability.

The `DeltaPredictor` property is a vector related to this threshold. When  $\delta \geq \text{DeltaPredictor}(i)$ , all classes  $k$  have

$$\left| \tilde{C}^{-1} D^{-1/2} (\mu_k - \mu_0) \right| \leq \delta.$$

Therefore, when  $\delta \geq \text{DeltaPredictor}(i)$ , the regularized classifier does not use predictor  $i$ .

## References

- [1] Guo, Y., T. Hastie, and R. Tibshirani. *Regularized linear discriminant analysis and its application in microarrays*. *Biostatistics*, Vol. 8, No. 1, pp. 86–100, 2007.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Only the `predict` function supports code generation.
- When you train a discriminant analysis model by using `fitcdiscr`, the following restrictions apply.
  - The class labels input argument value (`Y`) cannot be a categorical array.
  - The value of the `'ClassNames'` name-value pair argument cannot be a categorical array.
  - The value of the `'ScoreTransform'` name-value pair argument cannot be an anonymous function.

### See Also

`CompactClassificationDiscriminant` | `fitcdiscr`

### Topics

“Discriminant Analysis Classification” on page 20-2

Introduced in R2011b

## ClassificationEnsemble class

### Superclasses:

Ensemble classifier

### Description

`ClassificationEnsemble` combines a set of trained weak learner models and data on which these learners were trained. It can predict ensemble response for new data by aggregating predictions from its weak learners. It stores data used for training, can compute resubstitution predictions, and can resume training if desired.

### Construction

Create a classification ensemble object using `fitcensemble`.

### Properties

#### **CategoricalPredictors**

Indices of categorical predictors, stored as a vector of positive integers. `CategoricalPredictors` contains index values corresponding to the columns of the predictor data that contain categorical predictors. If none of the predictors are categorical, then this property is empty (`[]`).

#### **ClassNames**

List of the elements in `Y` with duplicates removed. `ClassNames` can be a numeric vector, categorical vector, logical vector, character array, or cell array of character vectors. `ClassNames` has the same data type as the data in the argument `Y`.

#### **CombineWeights**

Character vector describing how `ens` combines weak learner weights, either `'WeightedSum'` or `'WeightedAverage'`.

**Cost**

Square matrix, where  $\text{Cost}(i, j)$  is the cost of classifying a point into class  $j$  if its true class is  $i$  (i.e., the rows correspond to the true class and the columns correspond to the predicted class). The order of the rows and columns of `Cost` corresponds to the order of the classes in `ClassNames`. The number of rows and columns in `Cost` is the number of unique classes in the response. This property is read-only.

**ExpandedPredictorNames**

Expanded predictor names, stored as a cell array of character vectors.

If the model uses encoding for categorical variables, then `ExpandedPredictorNames` includes the names that describe the expanded variables. Otherwise, `ExpandedPredictorNames` is the same as `PredictorNames`.

**FitInfo**

Numeric array of fit information. The `FitInfoDescription` property describes the content of this array.

**FitInfoDescription**

Character vector describing the meaning of the `FitInfo` array.

**HyperparameterOptimizationResults**

Description of the cross-validation optimization of hyperparameters, stored as a `BayesianOptimization` object or a table of hyperparameters and associated values. Nonempty when the `OptimizeHyperparameters` name-value pair is nonempty at creation. Value depends on the setting of the `HyperparameterOptimizationOptions` name-value pair at creation:

- 'bayesopt' (default) — Object of class `BayesianOptimization`
- 'gridsearch' or 'randomsearch' — Table of hyperparameters used, observed objective function values (cross-validation loss), and rank of observations from lowest (best) to highest (worst)

**LearnerNames**

Cell array of character vectors with names of weak learners in the ensemble. The name of each learner appears just once. For example, if you have an ensemble of 100 trees, `LearnerNames` is `{ 'Tree' }`.

**Method**

Character vector describing the method that creates `ens`.

**ModelParameters**

Parameters used in training `ens`.

**NumObservations**

Numeric scalar containing the number of observations in the training data.

**NumTrained**

Number of trained weak learners in `ens`, a scalar.

**PredictorNames**

Cell array of names for the predictor variables, in the order in which they appear in `X`.

**Prior**

Numeric vector of prior probabilities for each class. The order of the elements of `Prior` corresponds to the order of the classes in `ClassNames`. The number of elements of `Prior` is the number of unique classes in the response. This property is read-only.

**ReasonForTermination**

Character vector describing the reason `fitcensemble` stopped adding weak learners to the ensemble.

**ResponseName**

Character vector with the name of the response variable `Y`.



**ScoreTransform**

Function handle for transforming scores, or character vector representing a built-in transformation function. 'none' means no transformation; equivalently, 'none' means  $@(x) x$ . For a list of built-in transformation functions and the syntax of custom transformation functions, see `fitctree`.

Add or change a `ScoreTransform` function using dot notation:

```
ens.ScoreTransform = 'function'
```

or

```
ens.ScoreTransform = @function
```

**Trained**

A cell vector of trained classification models.

- If `Method` is 'LogitBoost' or 'GentleBoost', then `ClassificationEnsemble` stores trained learner  $j$  in the `CompactRegressionLearner` property of the object stored in `Trained{j}`. That is, to access trained learner  $j$ , use `ens.Trained{j}.CompactRegressionLearner`.
- Otherwise, cells of the cell vector contain the corresponding, compact classification models.

**TrainedWeights**

Numeric vector of trained weights for the weak learners in `ens`. `TrainedWeights` has `T` elements, where `T` is the number of weak learners in `learners`.

**UsePredForLearner**

Logical matrix of size `P`-by-`NumTrained`, where `P` is the number of predictors (columns) in the training data `X`. `UsePredForLearner(i,j)` is `true` when learner  $j$  uses predictor  $i$ , and is `false` otherwise. For each learner, the predictors have the same order as the columns in the training data `X`.

If the ensemble is not of type `Subspace`, all entries in `UsePredForLearner` are `true`.

**W**

Scaled `weights`, a vector with length `n`, the number of rows in `X`. The sum of the elements of `W` is 1.

**X**

Matrix of predictor values that trained the ensemble. Each column of `X` represents one variable, and each row represents one observation.

**Y**

Numeric vector, categorical vector, logical vector, character array, or cell array of character vectors. Each row of `Y` represents the classification of the corresponding row of `X`.

## Methods

<code>compact</code>	Compact classification ensemble
<code>crossval</code>	Cross validate ensemble
<code>resubEdge</code>	Classification edge by resubstitution
<code>resubLoss</code>	Classification error by resubstitution
<code>resubMargin</code>	Classification margins by resubstitution
<code>resubPredict</code>	Predict ensemble response by resubstitution
<code>resume</code>	Resume training ensemble

## Inherited Methods

compareHoldout	Compare accuracies of two classification models using new data
edge	Classification edge
loss	Classification error
margin	Classification margins
predict	Predict labels using ensemble of classification models
predictorImportance	Estimates of predictor importance
removeLearners	Remove members of compact classification ensemble

## Copy Semantics

Value. To learn how value classes affect copy operations, see [Copying Objects \(MATLAB\)](#).

## Examples

### Train Boosted Classification Ensemble

Load the ionosphere data set.

```
load ionosphere
```

Train a boosted ensemble of 100 classification trees using all measurements and the AdaBoostM1 method.

```
Mdl = fitensemble(X,Y,'AdaBoostM1',100,'tree')
```

```
Mdl =
```

```
classreg.learning.classif.ClassificationEnsemble
    ResponseName: 'Y'
    CategoricalPredictors: []
           ClassNames: {'b' 'g'}
    ScoreTransform: 'none'
    NumObservations: 351
```

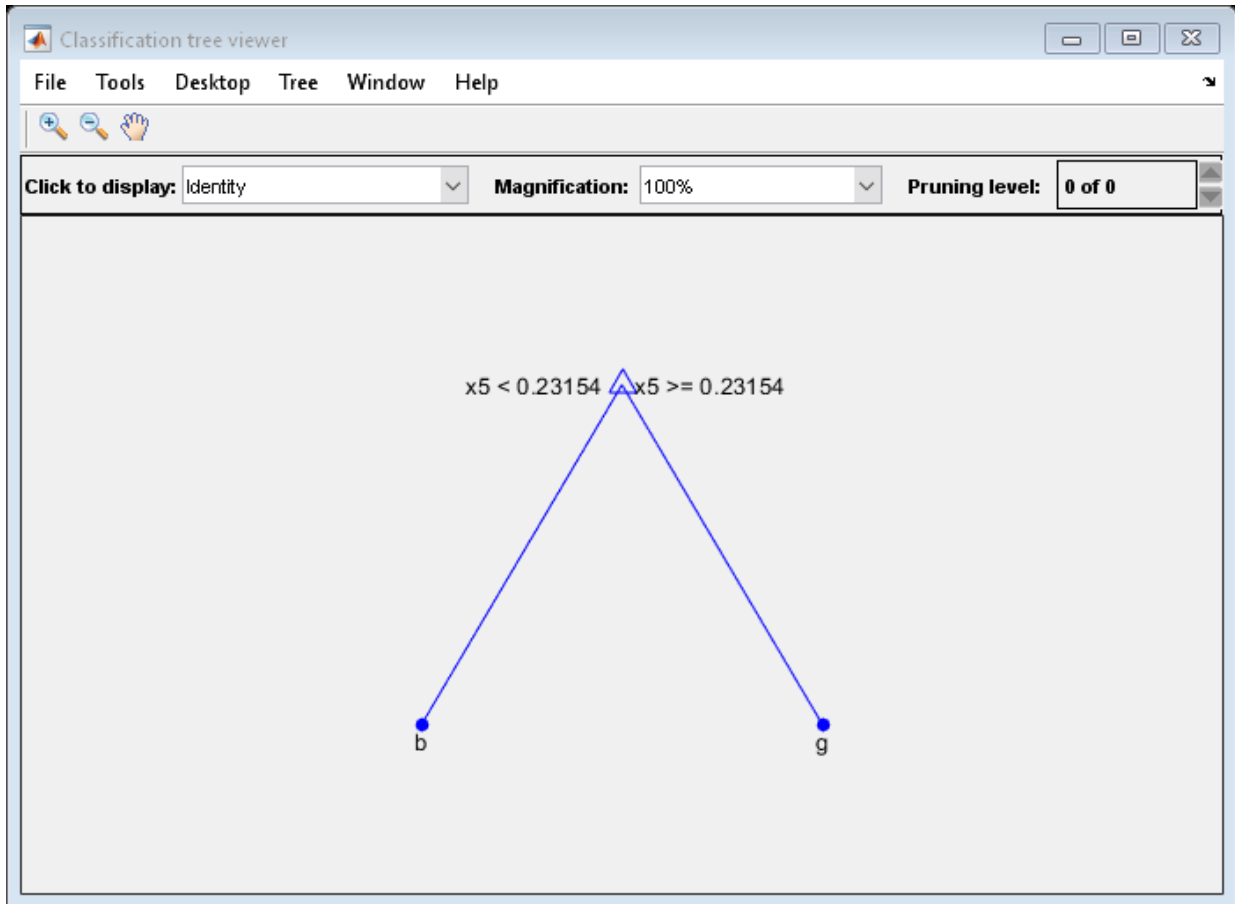
```
    NumTrained: 100
      Method: 'AdaBoostM1'
    LearnerNames: {'Tree'}
ReasonForTermination: 'Terminated normally after completing the requested number o
      FitInfo: [100x1 double]
FitInfoDescription: {2x1 cell}
```

Mdl is a `ClassificationEnsemble` model object.

`Mdl.Trained` is the property that stores a 100-by-1 cell vector of the trained classification trees (`CompactClassificationTree` model objects) that compose the ensemble.

Plot a graph of the first trained classification tree.

```
view(Mdl.Trained{1}, 'Mode', 'graph')
```



By default, `fitensemble` grows stumps for boosted ensembles of trees.

Predict the label of the mean of  $X$ .

```
predMeanX = predict(Mdl,mean(X))
```

```
predMeanX =
```

```
1x1 cell array
```

```
{'g'}
```

## Tip

For an ensemble of classification trees, the `Trained` property of `ens` stores an `ens.NumTrained-by-1` cell vector of compact classification models. For a textual or graphical display of tree `t` in the cell vector, enter:

- `view(ens.Trained{t}.CompactRegressionLearner)` for ensembles aggregated using `LogitBoost` or `GentleBoost`.
- `view(ens.Trained{t})` for all other aggregation methods.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Only the `predict` function supports code generation.
- When you train an ensemble by using `fitcensemble`, the following restrictions apply.
  - You can specify only 'Tree' or tree templates (see `templateTree`) as weak learners using the 'Learners' name-value pair argument.
  - Code generation limitations for classification trees also apply to ensembles of classification trees. For more details, see “Code Generation” on page 32-0 of the `CompactClassificationTree` class.

### See Also

`ClassificationTree` | `CompactClassificationEnsemble` | `fitcensemble` | `view`

Introduced in R2011a

## ClassificationKNN class

*k*-nearest neighbor classification

### Description

A nearest-neighbor classification object, where both distance metric (“nearest”) and number of neighbors can be altered. The object classifies new observations using the `predict` method. The object contains the data used for training, so can compute resubstitution predictions.

### Construction

`mdl = fitcknn(Tbl, ResponseVarName)` returns a classification model based on the input variables (also known as predictors, features, or attributes) in the table `Tbl` and output (response) `Tbl.ResponseVarName`.

`mdl = fitcknn(Tbl, formula)` returns a classification model based on the predictor data and class labels in the table `Tbl`. `formula` is an explanatory model of the response and a subset of predictor variables in `Tbl` used for training.

`mdl = fitcknn(Tbl, Y)` returns a classification model based on the input variables (also known as predictors, features, or attributes) in the table `Tbl` and output (response) `Y`.

`mdl = fitcknn(X, Y)` returns a classification model based on the input variables `X` and output (response) `Y`.

`mdl = fitcknn( ____, Name, Value)` fits a model with additional options specified by one or more name-value pair arguments, using any of the previous syntaxes. For example, you can specify the tie-breaking algorithm, distance metric, or observation weights.



## Input Arguments

### **Tbl** — Sample data

table

Sample data used to train the model, specified as a table. Each row of `Tbl` corresponds to one observation, and each column corresponds to one predictor variable. Optionally, `Tbl` can contain one additional column for the response variable. Multi-column variables and cell arrays other than cell arrays of character vectors are not allowed.

If `Tbl` contains the response variable, and you want to use all remaining variables in `Tbl` as predictors, then specify the response variable using `ResponseVarName`.

If `Tbl` contains the response variable, and you want to use only a subset of the remaining variables in `Tbl` as predictors, then specify a formula using `formula`.

If `Tbl` does not contain the response variable, then specify a response variable using `Y`. The length of response variable and the number of rows of `Tbl` must be equal.

Data Types: `table`

### **ResponseVarName** — Response variable name

name of variable in `Tbl`

Response variable name, specified as the name of a variable in `Tbl`.

You must specify `ResponseVarName` as a character vector. For example, if the response variable `Y` is stored as `Tbl.Y`, then specify it as `'Y'`. Otherwise, the software treats all columns of `Tbl`, including `Y`, as predictors when training the model.

The response variable must be a categorical or character array, logical or numeric vector, or cell array of character vectors. If `Y` is a character array, then each element must correspond to one row of the array.

It is good practice to specify the order of the classes using the `ClassNames` name-value pair argument.

Data Types: `char`

### **formula** — Explanatory model of response and subset of predictor variables

character vector

Explanatory model of the response and a subset of the predictor variables, specified as a character vector in the form of 'Y~X1+X2+X3'. In this form, Y represents the response variable, and X1, X2, and X3 represent the predictor variables. The variables must be variable names in Tbl (Tbl.Properties.VariableNames).

To specify a subset of variables in Tbl as predictors for training the model, use a formula. If you specify a formula, then the software does not use any variables in Tbl that do not appear in formula.

Data Types: char

### **y — Class labels**

numeric vector | categorical vector | logical vector | character array | cell array of character vectors

Class labels, specified as a categorical or character array, logical or numeric vector, or cell array of character vectors. Each row of Y represents the classification of the corresponding row of X.

The software considers NaN, '' (empty character vector), and <undefined> values in Y to be missing values. Consequently, the software does not train using observations with a missing response.

Data Types: single | double | logical | char | cell

### **x — Predictor data**

numeric matrix

Predictor data, specified as numeric matrix.

Each row corresponds to one observation (also known as an instance or example), and each column corresponds to one predictor variable (also known as a feature).

The length of Y and the number of rows of X must be equal.

To specify the names of the predictors in the order of their appearance in X, use the PredictorNames name-value pair argument.

Data Types: double | single

## Properties

### **BreakTies**

Character vector specifying the method `predict` uses to break ties if multiple classes have the same smallest cost. By default, ties occur when multiple classes have the same number of nearest points among the  $K$  nearest neighbors.

- 'nearest' — Use the class with the nearest neighbor among tied groups.
- 'random' — Use a random tiebreaker among tied groups.
- 'smallest' — Use the smallest index among tied groups.

'BreakTies' applies when 'IncludeTies' is false.

Change `BreakTies` using dot notation: `mdl.BreakTies = newBreakTies`.

### **CategoricalPredictors**

Indices of categorical predictors, stored as a vector of positive integers. `CategoricalPredictors` contains index values corresponding to the columns of the predictor data that contain categorical predictors. If none of the predictors are categorical, then this property is empty (`[]`).

### **ClassNames**

List of elements in the training data  $Y$  with duplicates removed. `ClassNames` can be a numeric vector, vector of categorical variables, logical vector, character array, or cell array of character vectors. `ClassNames` has the same data type as the data in the argument  $Y$ .

Change `ClassNames` using dot notation: `mdl.ClassNames = newClassNames`

### **Cost**

Square matrix, where `Cost(i, j)` is the cost of classifying a point into class  $j$  if its true class is  $i$  (i.e., the rows correspond to the true class and the columns correspond to the predicted class). The order of the rows and columns of `Cost` corresponds to the order of the classes in `ClassNames`. The number of rows and columns in `Cost` is the number of unique classes in the response.

Change a `Cost` matrix using dot notation: `obj.Cost = costMatrix`.

**Distance**

Character vector or function handle specifying the distance metric. The allowable character vectors depend on the `NSMethod` parameter, which you set in `fitcknn`, and which exists as a field in `ModelParameters`.

<b>NSMethod</b>	<b>Distance Metric Names</b>
exhaustive	Any distance metric of <code>ExhaustiveSearcher</code>
kdtree	'cityblock', 'chebychev', 'euclidean', or 'minkowski'

For definitions, see “Distance Metrics” on page 18-16.

The distance metrics of `ExhaustiveSearcher`:

<b>Value</b>	<b>Description</b>
'cityblock'	City block distance.
'chebychev'	Chebychev distance (maximum coordinate difference).
'correlation'	One minus the sample linear correlation between observations (treated as sequences of values).
'cosine'	One minus the cosine of the included angle between observations (treated as vectors).
'euclidean'	Euclidean distance.
'hamming'	Hamming distance, percentage of coordinates that differ.
'jaccard'	One minus the Jaccard coefficient, the percentage of nonzero coordinates that differ.
'mahalanobis'	Mahalanobis distance, computed using a positive definite covariance matrix $C$ . The default value of $C$ is the sample covariance matrix of $X$ , as computed by <code>nancov(X)</code> . To specify a different value for $C$ , set the <code>DistParameter</code> property of <code>mdl</code> using dot notation.
'minkowski'	Minkowski distance. The default exponent is 2. To specify a different exponent, set the <code>DistParameter</code> property of <code>mdl</code> using dot notation.

Value	Description
'seuclidean'	Standardized Euclidean distance. Each coordinate difference between $X$ and a query point is scaled, meaning divided by a scale value $S$ . The default value of $S$ is the standard deviation computed from $X$ , $S = \text{nanstd}(X)$ . To specify another value for $S$ , set the <code>DistParameter</code> property of <code>mdl</code> using dot notation.
'spearman'	One minus the sample Spearman's rank correlation between observations (treated as sequences of values).
@ <i>distfun</i>	Distance function handle. <i>distfun</i> has the form <pre>function D2 = DISTFUN(ZI,ZJ) % calculation of distance ...</pre> <p>where</p> <ul style="list-style-type: none"> <li>• <math>ZI</math> is a 1-by-<math>N</math> vector containing one row of <math>X</math> or <math>Y</math>.</li> <li>• <math>ZJ</math> is an <math>M2</math>-by-<math>N</math> matrix containing multiple rows of <math>X</math> or <math>Y</math>.</li> <li>• <math>D2</math> is an <math>M2</math>-by-1 vector of distances, and <math>D2(k)</math> is the distance between observations <math>ZI</math> and <math>ZJ(J, :)</math>.</li> </ul>

Change Distance using dot notation: `mdl.Distance = newDistance`.

If `NSMethod` is `kdtree`, you can use dot notation to change `Distance` only among the types `'cityblock'`, `'chebychev'`, `'euclidean'`, or `'minkowski'`.

### DistanceWeight

Character vector or function handle specifying the distance weighting function.

DistanceWeight	Meaning
'equal'	No weighting
'inverse'	Weight is 1/distance
'squaredinverse'	Weight is 1/distance <sup>2</sup>

DistanceWeight	Meaning
@fcn	fcn is a function that accepts a matrix of nonnegative distances, and returns a matrix the same size containing nonnegative distance weights. For example, 'inversesquared' is equivalent to @(d) d.^(-2).

Change DistanceWeight using dot notation: mdl.DistanceWeight = newDistanceWeight.

### DistParameter

Additional parameter for the distance metric.

Distance Metric	Parameter
'mahalanobis'	Positive definite covariance matrix C.
'minkowski'	Minkowski distance exponent, a positive scalar.
'seuclidean'	Vector of positive scale values with length equal to the number of columns of X.

For values of the distance metric other than those in the table, DistParameter must be [].

You can alter DistParameter using dot notation: mdl.DistParameter = newDistParameter. However, if Distance is mahalanobis or seuclidean, then you cannot alter DistParameter.

### ExpandedPredictorNames

Expanded predictor names, stored as a cell array of character vectors.

If the model uses encoding for categorical variables, then ExpandedPredictorNames includes the names that describe the expanded variables. Otherwise, ExpandedPredictorNames is the same as PredictorNames.

### HyperparameterOptimizationResults

Description of the cross-validation optimization of hyperparameters, stored as a BayesianOptimization object or a table of hyperparameters and associated values. Nonempty when the OptimizeHyperparameters name-value pair is nonempty at

creation. Value depends on the setting of the `HyperparameterOptimizationOptions` name-value pair at creation:

- `'bayesopt'` (default) — Object of class `BayesianOptimization`
- `'gridsearch'` or `'randomsearch'` — Table of hyperparameters used, observed objective function values (cross-validation loss), and rank of observations from lowest (best) to highest (worst)

### **IncludeTies**

Logical value indicating whether `predict` includes all the neighbors whose distance values are equal to the `K`th smallest distance. If `IncludeTies` is true, `predict` includes all these neighbors. Otherwise, `predict` uses exactly `K` neighbors (see `'BreakTies'`).

Change `IncludeTies` using dot notation: `mdl.IncludeTies = newIncludeTies`.

### **ModelParameters**

Parameters used in training `mdl`.

### **Mu**

Numeric vector of predictor means with length `numel(PredictorNames)`.

If you did not standardize `mdl` when you trained it using `fitcknn`, then `Mu` is empty (`[]`).

### **NumNeighbors**

Positive integer specifying the number of nearest neighbors in `X` to find for classifying each point when predicting. Change `NumNeighbors` using dot notation:  
`mdl.NumNeighbors = newNumNeighbors`.

### **NumObservations**

Number of observations used in training `mdl`. This can be less than the number of rows in the training data, because data rows containing `NaN` values are not part of the fit.

**PredictorNames**

Cell array of names for the predictor variables, in the order in which they appear in the training data  $X$ . Change `PredictorNames` using dot notation: `mdl.PredictorNames = newPredictorNames`.

**Prior**

Numeric vector of prior probabilities for each class. The order of the elements of `Prior` corresponds to the order of the classes in `ClassNames`.

Add or change a `Prior` vector using dot notation: `obj.Prior = priorVector`.

**ResponseName**

Character vector describing the response variable  $Y$ . Change `ResponseName` using dot notation: `mdl.ResponseName = newResponseName`.

**Sigma**

Numeric vector of predictor standard deviations with length `numel(PredictorNames)`.

If you did not standardize `mdl` when you trained it using `fitcknn`, then `Sigma` is empty (`[]`).

**W**

Numeric vector of nonnegative weights with the same number of rows as  $Y$ . Each entry in `W` specifies the relative importance of the corresponding observation in  $Y$ .

**X**

Numeric matrix of unstandardized predictor values. Each column of `X` represents one predictor (variable), and each row represents one observation.

**Y**

A numeric vector, vector of categorical variables, logical vector, character array, or cell array of character vectors, with the same number of rows as  $X$ .

$Y$  is of the same type as the passed-in  $Y$  data.



## Methods

compareHoldout	Compare accuracies of two models using new data
crossval	Cross-validated $k$ -nearest neighbor classifier
edge	Edge of $k$ -nearest neighbor classifier
loss	Loss of $k$ -nearest neighbor classifier
margin	Margin of $k$ -nearest neighbor classifier
predict	Predict labels using $k$ -nearest neighbor classification model
resubEdge	Edge of $k$ -nearest neighbor classifier by resubstitution
resubLoss	Loss of $k$ -nearest neighbor classifier by resubstitution
resubMargin	Margin of $k$ -nearest neighbor classifier by resubstitution
resubPredict	Predict resubstitution response of $k$ -nearest neighbor classifier

## Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects (MATLAB).

## Examples

### Train a $k$ -Nearest Neighbor Classifier

Construct a  $k$ -nearest neighbor classifier for Fisher's iris data, where  $k$ , the number of nearest neighbors in the predictors, is 5.

Load Fisher's iris data.

```
load fisheriris
X = meas;
Y = species;
```

X is a numeric matrix that contains four petal measurements for 150 irises. Y is a cell array of character vectors that contains the corresponding iris species.

Train a 5-nearest neighbors classifier. It is good practice to standardize noncategorical predictor data.

```
Mdl = fitcknn(X,Y,'NumNeighbors',5,'Standardize',1)
```

```
Mdl =
```

```
ClassificationKNN
    ResponseName: 'Y'
    CategoricalPredictors: []
        ClassNames: {'setosa' 'versicolor' 'virginica'}
    ScoreTransform: 'none'
    NumObservations: 150
        Distance: 'euclidean'
    NumNeighbors: 5
```

Mdl is a trained `ClassificationKNN` classifier, and some of its properties display in the Command Window.

To access the properties of Mdl, use dot notation.

```
Mdl.ClassNames
Mdl.Prior
```

```
ans =
```

```
3x1 cell array

    {'setosa'    }
    {'versicolor'}
    {'virginica' }
```

```
ans =
```

```
0.3333    0.3333    0.3333
```

Mdl.Prior contains the class prior probabilities, which are settable using the name-value pair argument 'Prior' in `fitcknn`. The order of the class prior probabilities

corresponds to the order of the classes in `Mdl.ClassNames`. By default, the prior probabilities are the respective relative frequencies of the classes in the data.

You can also reset the prior probabilities after training. For example, set the prior probabilities to 0.5, 0.2, and 0.3 respectively.

```
Mdl.Prior = [0.5 0.2 0.3];
```

You can pass `Mdl` to, for example, `ClassificationKNN.predict` to label new measurements, or `ClassificationKNN.crossval` to cross validate the classifier.

- “Construct KNN Classifier” on page 18-36
- “Examine Quality of KNN Classifier” on page 18-37
- “Predict Classification Using KNN Classifier” on page 18-38
- “Modify KNN Classifier” on page 18-39

## Definitions

### Prediction

`ClassificationKNN` predicts the classification of a point  $X_{\text{new}}$  using a procedure equivalent to this:

- 1 Find the `NumNeighbors` points in the training set  $X$  that are nearest to  $X_{\text{new}}$ .
- 2 Find the `NumNeighbors` response values  $Y$  to those nearest points.
- 3 Assign the classification label  $Y_{\text{new}}$  that has smallest expected misclassification cost among the values in  $Y$ .

For details, see “Posterior Probability” on page 32-4897 and “Expected Cost” on page 32-4898 in the `predict` documentation.

### Tips

The `compact` function reduces the size of most classification models by removing the training data properties, and any other properties that are not required to predict the label of new observations. Because  $k$ NN classification models require all of the training data to predict labels, you cannot reduce the size of a `ClassificationKNN` model.

## Alternatives

`knnsearch` finds the  $k$ -nearest neighbors of points. `rangesearch` finds all the points within a fixed distance. You can use these functions for classification, as shown in “Classify Query Data” on page 18-24. If you want to perform classification, `ClassificationKNN` can be more convenient, in that you can construct a classifier in one step and classify in other steps. Also, `ClassificationKNN` has cross-validation options.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Only the `predict` function supports code generation.
- When you train a  $k$ -nearest neighbor classification model by using `fitcknn`, the following restrictions apply.
  - The class labels input argument value ( $Y$ ) cannot be a categorical array.
  - Code generation does not support categorical predictors. You cannot supply training data in a table that contains at least one categorical predictor, and you cannot use the 'CategoricalPredictors' name-value pair argument. To dummy-code variables that you want treated as categorical, preprocess the categorical data by using `dummyvar` before fitting the model.
  - The value of the 'ClassNames' name-value pair argument cannot be a categorical array.
  - The value of the 'Distance' name-value pair argument cannot be a custom distance function.
  - The value of the 'DistanceWeight' name-value pair argument can be a custom distance weight function, but it cannot be an anonymous function.
  - The value of the 'NSMethod' name-value pair argument must be 'exhaustive'.

- The value of the 'ScoreTransform' name-value pair argument cannot be an anonymous function.

## See Also

`fitcknn` | `predict`

## Topics

- “Construct KNN Classifier” on page 18-36
- “Examine Quality of KNN Classifier” on page 18-37
- “Predict Classification Using KNN Classifier” on page 18-38
- “Modify KNN Classifier” on page 18-39
- “Classification Using Nearest Neighbors” on page 18-16

## ClassificationLinear class

Linear model for binary classification of high-dimensional data

### Description

`ClassificationLinear` is a trained linear model object for binary classification; the linear model is a support vector machine (SVM) or logistic regression model. `fitclinear` fits a `ClassificationLinear` model by minimizing the objective function using techniques that reduce computation time for high-dimensional data sets (e.g., stochastic gradient descent). The classification loss plus the regularization term compose the objective function.

Unlike other classification models, and for economical memory usage, `ClassificationLinear` model objects do not store the training data. However, they do store, for example, the estimated linear model coefficients, prior-class probabilities, and the regularization strength.

You can use trained `ClassificationLinear` models to predict labels or classification scores for new data. For details, see `predict`.

### Construction

`Mdl = fitclinear(X,Y)` returns a trained linear classification model object that contains the results of fitting a binary support vector machine to the predictor data `X` and class labels `Y`.

`Mdl = fitclinear(___,Name,Value)` returns a trained linear classification model with additional options specified by one or more `Name,Value` pair arguments, using any of the previous syntaxes. For example, you can specify to implement logistic regression, specify to cross-validate, or specify the type of regularization.

If you specify one of the following four options, then `Mdl` is a `ClassificationPartitionedLinear` model: `'CrossVal'`, `'CVPartition'`, `'Holdout'`, or `'KFold'`.

---

**Note** Unlike other classification model objects:

- For economical memory usage, `ClassificationLinear` model objects do not store the training data or training process details (for example, convergence history).
  - You cannot cross-validate by passing a trained `ClassificationLinear` model object to `crossval`. Instead, specify the appropriate cross-validation name-value pair argument when you train the model using `fitclinear`.
- 

## Input Arguments

### **x** — Predictor data

full matrix | sparse matrix

Predictor data, specified as an  $n$ -by- $p$  full or sparse matrix.

The length of  $Y$  and the number of observations in  $X$  must be equal.

---

**Note** If you orient your predictor matrix so that observations correspond to columns and specify `'ObservationsIn', 'columns'`, then you might experience a significant reduction in optimization-execution time.

---

Data Types: single | double

### **y** — Class labels

categorical array | character array | logical vector | vector of numeric values | cell array of character vectors

Class labels to which the classification model is trained, specified as a categorical or character array, logical or numeric vector, or cell array of character vectors.

- `ClassificationLinear` only supports binary classification. Either  $Y$  must contain exactly two distinct classes, or you must specify two classes for training using the `'ClassNames'` name-value pair argument. For multiclass learning, see `fitcecoc`.
- If  $Y$  is a character array, then each element must correspond to one row of the array.
- The length of  $Y$  and the number of observations in  $X$  must be equal.
- It is good practice to specify the class order using the `ClassNames` name-value pair argument.

Data Types: char | cell | categorical | logical | single | double

For more details, see `fitclinear`.

## Properties

### Linear Classification Properties

#### **Lambda** — Regularization term strength

nonnegative scalar | vector of nonnegative values

Regularization term strength, specified as a nonnegative scalar or vector of nonnegative values.

Data Types: double | single

#### **Learner** — Linear classification model type

'logistic' | 'svm'

Linear classification model type, specified as 'logistic' or 'svm'.

In this table,  $f(x) = x\beta + b$ .

- $\beta$  is a vector of  $p$  coefficients.
- $x$  is an observation from  $p$  predictor variables.
- $b$  is the scalar bias.

Value	Algorithm	Loss Function	FittedLoss Value
'logistic'	Logistic regression	Deviance (logistic): $\ell[y, f(x)] = \log\{1 + \exp[-yf(x)]\}$	'logit'
'svm'	Support vector machine	Hinge: $\ell[y, f(x)] = \max[0, 1 - yf(x)]$	'hinge'

#### **Beta** — Linear coefficient estimates

numeric vector



Linear coefficient estimates, specified as a numeric vector with length equal to the number of predictors.

Data Types: double

### **Bias — Estimated bias term**

numeric scalar

Estimated bias term or model intercept, specified as a numeric scalar.

Data Types: double

### **FittedLoss — Loss function used to fit the linear model**

'hinge' | 'logit'

Loss function used to fit the linear model, specified as 'hinge' or 'logit'.

Value	Algorithm	Loss Function	Learner Value
'hinge'	Support vector machine	Hinge: $\ell[y, f(x)] = \max[0, 1 - yf(x)]$	'svm'
'logit'	Logistic regression	Deviance (logistic): $\ell[y, f(x)] = \log\{1 + \exp[-yf(x)]\}$	'logistic'

### **Regularization — Complexity penalty type**

'lasso (L1)' | 'ridge (L2)'

Complexity penalty type, specified as 'lasso (L1)' or 'ridge (L2)'.

The software composes the objective function for minimization from the sum of the average loss function (see FittedLoss) and the regularization term in this table.

Value	Description
'lasso (L1)'	Lasso ( $L_1$ ) penalty: $\lambda \sum_{j=1}^p  \beta_j $
'ridge (L2)'	Ridge ( $L_2$ ) penalty: $\frac{\lambda}{2} \sum_{j=1}^p \beta_j^2$

$\lambda$  specifies the regularization term strength (see `Lambda`).

The software excludes the bias term ( $\beta_0$ ) from the regularization penalty.

### Other Classification Properties

#### `CategoricalPredictors` — Indices of categorical predictors

`[]`

Indices of categorical predictors, which is always empty (`[]`) because a `ClassificationLinear` model does not support categorical predictors.

#### `ClassNames` — Unique class labels

categorical array | character array | logical vector | vector of numeric values | cell array of character vectors

Unique class labels used in training, specified as a categorical or character array, logical or numeric vector, or cell array of character vectors. `ClassNames` has the same data type as class labels (`Y`). `ClassNames` also determines class order.

Data Types: `categorical` | `char` | `logical` | `single` | `double` | `cell`

#### `Cost` — Misclassification costs

square numeric matrix

This property is read-only.

Misclassification costs, specified as a square numeric matrix. `Cost` has  $K$  rows and columns, where  $K$  is the number of classes.

`Cost(i, j)` is the cost of classifying a point into class  $j$  if its true class is  $i$ . The order of the rows and columns of `Cost` corresponds to the order of the classes in `ClassNames`.

Data Types: `double`

#### `ModelParameters` — Parameters used for training model

structure

Parameters used for training the `ClassificationLinear` model, specified as a structure.

Access fields of `ModelParameters` using dot notation. For example, access the relative tolerance on linear coefficients and bias term using `Mdl.ModelParameters.BetaTolerance`.

Data Types: `struct`

#### **PredictorNames** — Predictor names

cell array of character vectors

Predictor names in their order of appearance in predictor data ( $X$ ), specified as a cell array of character vectors. `PredictorNames` has length equal to the number of columns in  $X$ .

Data Types: `cell`

#### **ExpandedPredictorNames** — Expanded predictor names

cell array of character vectors

Expanded predictor names, stored as a cell array of character vectors.

Because a `ClassificationLinear` model does not support categorical predictors, `ExpandedPredictorNames` and `PredictorNames` are equal.

Data Types: `cell`

#### **Prior** — Prior class probabilities

numeric vector

This property is read-only.

Prior class probabilities, specified as a numeric vector. `Prior` has as many elements as classes in `ClassNames`, and the order of the elements corresponds to the elements of `ClassNames`.

Data Types: `double`

#### **ScoreTransform** — Score transformation function to apply to predicted scores

function name | function handle

Score transformation function to apply to predicted scores, specified as a function name or function handle.

For linear classification models and before transformation, the predicted classification score for the observation  $x$  (row vector) is  $f(x) = x\beta + b$ , where  $\beta$  and  $b$  correspond to `Mdl.Beta` and `Mdl.Bias`, respectively.

To change the score transformation function to, e.g., *function*, use dot notation.

- For a built-in function, enter this code and replace *function* with a character vector in the table.

```
Mdl.ScoreTransform = 'function';
```

Value	Description
'doublelogit'	$1/(1 + e^{-2x})$
'invlogit'	$\log(x / (1-x))$
'ismax'	Set the score for the class with the largest score to 1, and set the scores for all other classes to 0.
'logit'	$1/(1 + e^{-x})$
'none' or 'identity'	$x$ (no transformation)
'sign'	-1 for $x < 0$ 0 for $x = 0$ 1 for $x > 0$
'symmetric'	$2x - 1$
'symmetricismax'	Set the score for the class with the largest score to 1, and set the scores for all other classes to -1.
'symmetriclogit'	$2/(1 + e^{-x}) - 1$

- For a MATLAB function, or a function that you define, enter its function handle.

```
Mdl.ScoreTransform = @function;
```

*function* must accept a matrix of the original scores for each class, and then return a matrix of the same size representing the transformed scores for each class.

Data Types: `char` | `function_handle`

**ResponseName** — Response variable name

character array

Response variable name, specified as a character array.

Data Types: `char`

## Methods

<code>edge</code>	Classification edge for linear classification models
<code>loss</code>	Classification loss for linear classification models
<code>margin</code>	Classification margins for linear classification models
<code>predict</code>	Predict labels for linear classification models
<code>selectModels</code>	Choose subset of regularized, binary linear classification models

## Copy Semantics

Value. To learn how value classes affect copy operations, see [Copying Objects \(MATLAB\)](#).

## Examples

### Train Linear Classification Model

Train a binary, linear classification model using support vector machines, dual SGD, and ridge regularization.

Load the NLP data set.

```
load nlpdata
```

`X` is a sparse matrix of predictor data, and `Y` is a categorical vector of class labels. There are more than two classes in the data.

Identify the labels that correspond to the Statistics and Machine Learning Toolbox™ documentation web pages.

```
Ystats = Y == 'stats';
```

Train a binary, linear classification model that can identify whether the word counts in a documentation web page are from the Statistics and Machine Learning Toolbox™ documentation. Train the algorithm using the entire data set. Determine how well the optimization algorithm fit the model to the data by extracting a fit summary.

```
rng(1); % For reproducibility
[Mdl,FitInfo] = fitclinear(X,Ystats)
```

```
Mdl =
```

```
ClassificationLinear
  ResponseName: 'Y'
  ClassNames: [0 1]
  ScoreTransform: 'none'
    Beta: [34023x1 double]
    Bias: -1.0059
    Lambda: 3.1674e-05
  Learner: 'svm'
```

```
FitInfo =
```

```
struct with fields:

    Lambda: 3.1674e-05
    Objective: 5.3783e-04
    PassLimit: 10
    NumPasses: 10
    BatchLimit: []
    NumIterations: 238561
    GradientNorm: NaN
    GradientTolerance: 0
    RelativeChangeInBeta: 0.0562
    BetaTolerance: 1.0000e-04
    DeltaGradient: 1.4582
    DeltaGradientTolerance: 1
    TerminationCode: 0
    TerminationStatus: {'Iteration limit exceeded.'}
    Alpha: [31572x1 double]
    History: []
    FitTime: 0.2200
```

```
Solver: {'dual'}
```

Mdl is a ClassificationLinear model. You can pass Mdl and the training or new data to loss to inspect the in-sample classification error. Or, you can pass Mdl and new predictor data to predict to predict class labels for new observations.

FitInfo is a structure array containing, among other things, the termination status (TerminationStatus) and how long the solver took to fit the model to the data (FitTime). It is good practice to use FitInfo to determine whether optimization-termination measurements are satisfactory. Because training time is small, you can try to retrain the model, but increase the number of passes through the data. This can improve measures like DeltaGradient.

## Predict Class Labels Using Linear Classification Model

Load the NLP data set.

```
load nlpdata
n = size(X,1); % Number of observations
```

Identify the labels that correspond to the Statistics and Machine Learning Toolbox™ documentation web pages.

```
Ystats = Y == 'stats';
```

Hold out 5% of the data.

```
rng(1); % For reproducibility
cvp = cvpartition(n, 'Holdout', 0.05)
```

```
cvp =
```

```
Hold-out cross validation partition
  NumObservations: 31572
  NumTestSets: 1
  TrainSize: 29994
  TestSize: 1578
```

cvp is a CVPartition object that defines the random partition of  $n$  data into training and test sets.

Train a binary, linear classification model using the training set that can identify whether the word counts in a documentation web page are from the Statistics and Machine Learning Toolbox™ documentation. For faster training time, orient the predictor data matrix so that the observations are in columns.

```
idxTrain = training(cvp); % Extract training set indices
X = X';
Mdl = fitclinear(X(:,idxTrain),Ystats(idxTrain),'ObservationsIn','columns');
```

Predict observations and classification error for the hold out sample.

```
idxTest = test(cvp); % Extract test set indices
labels = predict(Mdl,X(:,idxTest),'ObservationsIn','columns');
L = loss(Mdl,X(:,idxTest),Ystats(idxTest),'ObservationsIn','columns')
```

```
L =
```

```
7.1753e-04
```

Mdl misclassifies fewer than 1% of the out-of-sample observations.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Only the `predict` function supports code generation.
- When you train a linear classification model by using `fitclinear`, the following restrictions apply.
  - The predictor data input argument value (X) must be a full, numeric matrix.
  - The class labels input argument value (Y) cannot be a categorical array.
  - The value of the 'ClassNames' name-value pair argument or property cannot be a categorical array.



- You can specify only one regularization strength, either 'auto' or a nonnegative scalar for the 'Lambda' name-value pair argument.
- The value of the 'ScoreTransform' name-value pair argument cannot be an anonymous function.

## See Also

ClassificationECOC | ClassificationKernel |  
ClassificationPartitionedLinear | ClassificationPartitionedLinearECOC  
| fitclinear | predict

## Topics

“Code Generation Support, Usage Notes, and Limitations” on page 31-24

**Introduced in R2016a**

## ClassificationNaiveBayes class

### Superclasses:

Naive Bayes classification

### Description

`ClassificationNaiveBayes` is a naive Bayes on page 32-466 classifier for multiclass learning. Use `fitcnb` and the training data to train a `ClassificationNaiveBayes` classifier.

Trained `ClassificationNaiveBayes` classifiers store the training data, parameter values, data distribution, and prior probabilities. You can use these classifiers to:

- Estimate resubstitution predictions. For details, see `resubPredict`.
- Predict labels or posterior probabilities for new data. For details, see `predict`.

### Construction

`Mdl = fitcnb(Tbl,ResponseVarName)` returns a multiclass naive Bayes model (`Mdl`), trained by the predictors in table `Tbl` and class labels in `Tbl.ResponseVarName`.

`Mdl = fitcnb(Tbl,formula)` returns a multiclass naive Bayes model (`Mdl`), trained by the predictor data and class labels in the table `Tbl`. `formula` is an explanatory model of the response and a subset of predictor variables in `Tbl` used for training.

`Mdl = fitcnb(Tbl,Y)` returns a multiclass naive Bayes model (`Mdl`), trained by the predictors in the table `Tbl` and class labels in the vector `Y`.

`Mdl = fitcnb(X,Y)` returns a multiclass naive Bayes model (`Mdl`), trained by predictors `X` and class labels `Y`.

`Mdl = fitcnb( ____,Name,Value)` returns a naive Bayes classifier with additional options specified by one or more `Name,Value` pair arguments, using any of the previous

syntaxes. For example, you can specify a distribution to model the data, prior probabilities for the classes, or the kernel smoothing window bandwidth.

Predict labels for new data by passing the data and `Mdl` to `predict`.

If you set one of the following five options, then `Mdl` is a `ClassificationPartitionedModel` model: `'CrossVal'`, `'CVPartition'`, `'Holdout'`, `'KFold'`, or `'Leaveout'`. Otherwise, `Mdl` is a `ClassificationNaiveBayes` classifier.

## Input Arguments

### **Tbl** — Sample data

table

Sample data used to train the model, specified as a table. Each row of `Tbl` corresponds to one observation, and each column corresponds to one predictor variable. Optionally, `Tbl` can contain one additional column for the response variable. Multi-column variables and cell arrays other than cell arrays of character vectors are not allowed.

If `Tbl` contains the response variable, and you want to use all remaining variables in `Tbl` as predictors, then specify the response variable using `ResponseVarName`.

If `Tbl` contains the response variable, and you want to use only a subset of the remaining variables in `Tbl` as predictors, then specify a formula using `formula`.

If `Tbl` does not contain the response variable, then specify a response variable using `Y`. The length of response variable and the number of rows of `Tbl` must be equal.

Data Types: `table`

### **ResponseVarName** — Response variable name

name of variable in `Tbl`

Response variable name, specified as the name of a variable in `Tbl`.

You must specify `ResponseVarName` as a character vector. For example, if the response variable `Y` is stored as `Tbl.Y`, then specify it as `'Y'`. Otherwise, the software treats all columns of `Tbl`, including `Y`, as predictors when training the model.

The response variable must be a categorical or character array, logical or numeric vector, or cell array of character vectors. If  $Y$  is a character array, then each element must correspond to one row of the array.

It is good practice to specify the order of the classes using the `ClassNames` name-value pair argument.

Data Types: `char`

#### **formula** — Explanatory model of response and subset of predictor variables

character vector

Explanatory model of the response and a subset of the predictor variables, specified as a character vector in the form of ' $Y \sim X_1 + X_2 + X_3$ '. In this form,  $Y$  represents the response variable, and  $X_1$ ,  $X_2$ , and  $X_3$  represent the predictor variables. The variables must be variable names in `Tbl` (`Tbl.Properties.VariableNames`).

To specify a subset of variables in `Tbl` as predictors for training the model, use a formula. If you specify a formula, then the software does not use any variables in `Tbl` that do not appear in `formula`.

Data Types: `char`

#### **Y** — Class labels

categorical array | character array | logical vector | numeric vector | cell array of character vectors

Class labels to which the naive Bayes classifier is trained, specified as a categorical or character array, logical or numeric vector, or cell array of character vectors. Each element of  $Y$  defines the class membership of the corresponding row of  $X$ .  $Y$  supports  $K$  class levels.

If  $Y$  is a character array, then each row must correspond to one class label.

The length of  $Y$  and the number of rows of  $X$  must be equivalent.

Data Types: `cell` | `char` | `double` | `logical`

#### **x** — Predictor data

numeric matrix

Predictor data, specified as a numeric matrix.

Each row of  $X$  corresponds to one observation (also known as an instance or example), and each column corresponds to one variable (also known as a feature).

The length of  $Y$  and the number of rows of  $X$  must be equivalent.

Data Types: `double`

---

**Note:** The software treats NaN, empty character vector (' '), and <undefined> elements as missing values.

- If  $Y$  contains missing values, then the software removes them and the corresponding rows of  $X$ .
- If  $X$  contains any rows composed entirely of missing values, then the software removes those rows and the corresponding elements of  $Y$ .
- If  $X$  contains missing values and you set 'Distribution', 'mn', then the software removes those rows of  $X$  and the corresponding elements of  $Y$ .
- If a predictor is not represented in a class, that is, if all of its values are NaN within a class, then the software returns an error.

Removing rows of  $X$  and corresponding elements of  $Y$  decreases the effective training or cross-validation sample size.

---

## Properties

### **CategoricalPredictors** — Indices of categorical predictors

vector of positive integers

Indices of categorical predictors, stored as a vector of positive integers. `CategoricalPredictors` contains index values corresponding to the columns of the predictor data that contain categorical predictors. If none of the predictors are categorical, then this property is empty ( []).

Data Types: `single` | `double`

### **CategoricalLevels** — Multivariate multinomial levels

cell vector of numeric vectors

Multivariate multinomial levels, specified as a cell vector of numeric vectors. `CategoricalLevels` has length equal to the number of predictors (`size(X, 2)`).

The cells of `CategoricalLevels` correspond to predictors that you specified as 'mvnm' (i.e., having a multivariate multinomial distribution) during training. Cells that do not correspond to a multivariate multinomial distribution are empty (`[]`).

If predictor  $j$  is multivariate multinomial, then `CategoricalLevels{j}` is a list of all distinct values of predictor  $j$  in the sample (NaNs removed from `unique(X(:,j))`).

Data Types: `cell`

### **ClassNames — Distinct class names**

categorical array | character array | logical vector | numeric vector | cell array of character vectors

Distinct class names, specified as a categorical or character array, logical or numeric vector, or cell vector of character vectors.

`ClassNames` is the same data type as `Y`, and has as  $K$  elements or rows for character arrays.

### **Cost — Misclassification cost**

square matrix

Misclassification cost, specified as a  $K$ -by- $K$  square matrix.

The value of `Cost(i,j)` is the cost of classifying a point into class  $j$  if its true class is  $i$ . The order of the rows and columns of `Cost` correspond to the order of the classes in `ClassNames`.

The value of `Cost` does not influence training. You can reset `Cost` after training `Mdl` using dot notation, e.g., `Mdl.Cost = [0 0.5; 1 0];`.

Data Types: `double` | `single`

### **DistributionNames — Predictor distributions**

'normal' (default) | 'kernel' | 'mn' | 'mvnm' | cell array of character vectors

Predictor distributions `fitcnb` uses to model the predictors, specified as a character vector or cell array of character vectors.

This table summarizes the available distributions.

Value	Description
'kernel'	Kernel smoothing density estimate.
'mn'	Multinomial bag-of-tokens model on page 32-466. Indicates that all predictors have this distribution.
'mvnm'	Multivariate multinomial distribution.
'normal'	Normal (Gaussian) distribution.

If `Distribution` is a 1-by- $P$  cell array of character vectors, then the software models feature  $j$  using the distribution in element  $j$  of the cell array.

Data Types: `cell` | `char`

#### **DistributionParameters** — Distribution parameter estimates

cell array

Distribution parameter estimates, specified as a cell array. `DistributionParameters` is a  $K$ -by- $D$  cell array, where cell  $(k,d)$  contains the distribution parameter estimates for instances of predictor  $d$  in class  $k$ . The order of the rows corresponds to the order of the classes in the property `ClassNames`, and the order of the predictors corresponds to the order of the columns of `X`.

If class  $k$  has no observations for predictor  $j$ , then `DistributionParameters{k,j}` is empty (`[]`).

The elements of `DistributionParameters` depends on the distributions of the predictors. This table describes the values in `DistributionParameters{k,j}`.

Distribution of Predictor $j$	Value
kernel	A <code>KernelDistribution</code> model. Display properties using cell indexing and dot notation. For example, to display the estimated bandwidth of the kernel density for predictor 2 in the third class, use <code>Mdl.DistributionParameters{3,2}.BandWidth</code> .
mn	A scalar representing the probability that token $j$ appears in class $k$ . For details, see “Algorithms” on page 32-467.

Distribution of Predictor $j$	Value
<code>mvmn</code>	A numeric vector containing the probabilities for each possible level of predictor $j$ in class $k$ . The software orders the probabilities by the sorted order of all unique levels of predictor $j$ (stored in the property <code>CategoricalLevels</code> ). For more details, see “Algorithms” on page 32-467.
<code>normal</code>	A 2-by-1 numeric vector. The first element is the sample mean and the second element is the sample standard deviation.

### **ExpandedPredictorNames** — Expanded predictor names

cell array of character vectors

Expanded predictor names, stored as a cell array of character vectors.

If the model uses encoding for categorical variables, then `ExpandedPredictorNames` includes the names that describe the expanded variables. Otherwise, `ExpandedPredictorNames` is the same as `PredictorNames`.

### **HyperparameterOptimizationResults** — Description of the cross-validation optimization of hyperparameters

`BayesianOptimization` object | table of hyperparameters and associated values

Description of the cross-validation optimization of hyperparameters, stored as a `BayesianOptimization` object or a table of hyperparameters and associated values. Nonempty when the `OptimizeHyperparameters` name-value pair is nonempty at creation. Value depends on the setting of the `HyperparameterOptimizationOptions` name-value pair at creation:

- `'bayesopt'` (default) — Object of class `BayesianOptimization`
- `'gridsearch'` or `'randomsearch'` — Table of hyperparameters used, observed objective function values (cross-validation loss), and rank of observations from lowest (best) to highest (worst)

Data Types: `table`



**Kernel — Kernel smoother types**

'normal' (default) | 'box' | 'epanechnikov' | 'triangle' | cell array of character vectors

Kernel smoother types, specified as a character vector or cell array of character vectors. Kernel has length equal to the number of predictors (`size(X, 2)`). `Kernel{j}` corresponds to predictor  $j$ , and contains a character vector describing the type of kernel smoother. This table describes the supported kernel smoother types. Let  $I\{u\}$  denote the indicator function.

Value	Kernel	Formula
'box'	Box (uniform)	$f(x) = 0.5I\{ x  \leq 1\}$
'epanechnikov'	Epanechnikov	$f(x) = 0.75(1 - x^2)I\{ x  \leq 1\}$
'normal'	Gaussian	$f(x) = \frac{1}{\sqrt{2\pi}} \exp(-0.5x^2)$
'triangle'	Triangular	$f(x) = (1 -  x )I\{ x  \leq 1\}$

If a cell is empty (`[]`), then the software did not fit a kernel distribution to the corresponding predictor.

**ModelParameters — Parameter values used to train**

object

Parameter values used to train the classifier (such as the name-value pair argument values), specified as an object. This table summarizes the properties of `ModelParameters`. The properties correspond to the name-value pair argument values set for training the classifier.

Property	Purpose
<code>DistributionNames</code>	Data distribution or distributions. This is the same value as the property <code>DistributionNames</code> .
<code>Kernel</code>	Kernel smoother type. This is the same as the property <code>Kernel</code> .

Property	Purpose
Method	Training method. For naive Bayes, the value is 'NaiveBayes'.
Support	Kernel-smoothing density support. This is the same as the property Support.
Type	Learning type. For classification, the value is 'classification'.
Width	Kernel smoothing window width. This is the same as the property Width.

Access fields of `ModelParameters` using dot notation. For example, access the kernel support using `Mdl.ModelParameters.Support`.

### **NumObservations** — Number of training observations

numeric scalar

Number of training observations, specified as a numeric scalar.

If  $X$  or  $Y$  contain missing values, then `NumObservations` might be less than the length of  $Y$ .

Data Types: `double`

### **PredictorNames** — Predictor names

character vector

Predictor names, specified as a cell vector of character vectors. The order of the elements in `PredictorNames` corresponds to the order in  $X$ .

Data Types: `cell`

### **Prior** — Class prior probabilities

numeric vector

Class prior probabilities, specified as a numeric row vector. `Prior` is a 1-by- $K$  vector, and the order of its elements correspond to the elements of `ClassNames`.

`fitcnb` normalizes the prior probabilities you set using the name-value pair parameter '`Prior`' so that `sum(Prior) = 1`.

The value of `Prior` does not change the best-fitting model. Therefore, you can reset `Prior` after training `Mdl` using dot notation, e.g., `Mdl.Prior = [0.2 0.8];`.

Data Types: `double` | `single`

**ResponseName** — Response name

character vector

Response name, specified as a character vector.

Data Types: `char`

**ScoreTransform** — Classification score transformation function

character vector | function handle

Classification score transformation function, specified as a character vector or function handle.

To change the score transformation function to e.g., *function*, use dot notation.

- For a built-in function, enter a character vector.

```
Mdl.ScoreTransform = 'function';
```

This table lists available, built-in functions.

Value	Description
'doublelogit'	$1/(1 + e^{-2x})$
'invlogit'	$\log(x / (1-x))$
'ismax'	Set the score for the class with the largest score to 1, and set the scores for all other classes to 0.
'logit'	$1/(1 + e^{-x})$
'none' or 'identity'	$x$ (no transformation)
'sign'	-1 for $x < 0$ 0 for $x = 0$ 1 for $x > 0$
'symmetric'	$2x - 1$

Value	Description
'symmetricismax'	Set the score for the class with the largest score to 1, and set the scores for all other classes to -1.
'symmetriclogit'	$2/(1 + e^{-x}) - 1$

- For a MATLAB function, or a function that you define, enter its function handle.

```
Mdl.ScoreTransform = @function;
```

*function* should accept a matrix (the original scores) and return a matrix of the same size (the transformed scores).

Data Types: char | function\_handle

### Support — Kernel smoother density support

cell vector

Kernel smoother density support, specified as a cell vector. Support has length equal to the number of predictors (`size(X, 2)`). The cells represent the regions to apply the kernel density.

This table describes the supported options.

Value	Description
1-by-2 numeric row vector	For example, [L,U], where L and U are the finite lower and upper bounds, respectively, for the density support.
'positive'	The density support is all positive real values.
'unbounded'	The density support is all real values.

If a cell is empty (`[]`), then the software did not fit a kernel distribution to the corresponding predictor.

### w — Observation weights

numeric vector

Observation weights, specified as a numeric vector.

The length of `W` is `NumObservations`.

`fitcnb` normalizes the value you set for the name-value pair parameter 'Weights' so that the weights within a particular class sum to the prior probability for that class.

Data Types: `double`

#### **width** — Kernel smoother window width

numeric matrix

Kernel smoother window width, specified as a numeric matrix. `width` is a  $K$ -by- $P$  matrix, where  $K$  is the number of classes in the data, and  $P$  is the number of predictors (`size(X, 2)`).

`width(k, j)` is the kernel smoother window width for the kernel smoothing density of predictor  $j$  within class  $k$ . NaNs in column  $j$  indicate that the software did not fit predictor  $j$  using a kernel density.

#### **x** — Unstandardized predictor data

numeric matrix

Unstandardized predictor data, specified as a numeric matrix. `X` has `NumObservations` rows and  $P$  columns.

Each row of `X` corresponds to one observation, and each column corresponds to one variable.

The software excludes rows removed due to missing values from `X`.

Data Types: `double`

#### **y** — Observed class labels

categorical array | character array | logical vector | numeric vector | cell array of character vectors

Observed class labels, specified as a categorical or character array, logical or numeric vector, or cell array of character vectors. `Y` is the same data type as the input argument `Y` of `fitcnb`.

Each row of `Y` represents the observed classification of the corresponding row of `X`.

The software excludes elements removed due to missing values from `Y`.

## Methods

<code>compact</code>	Compact naive Bayes classifier
<code>crossval</code>	Cross-validated naive Bayes classifier
<code>resubEdge</code>	Classification edge for naive Bayes classifiers by resubstitution
<code>resubLoss</code>	Classification loss for naive Bayes classifiers by resubstitution
<code>resubMargin</code>	Classification margins for naive Bayes classifiers by resubstitution
<code>resubPredict</code>	Predict naive Bayes classifier resubstitution response

## Inherited Methods

<code>compareHoldout</code>	Compare accuracies of two classification models using new data
<code>edge</code>	Classification edge for naive Bayes classifiers
<code>logP</code>	Log unconditional probability density for naive Bayes classifier
<code>loss</code>	Classification error for naive Bayes classifier
<code>margin</code>	Classification margins for naive Bayes classifiers
<code>predict</code>	Predict labels using naive Bayes classification model

## Copy Semantics

Value. To learn how value classes affect copy operations, see [Copying Objects \(MATLAB\)](#).

## Examples

### Train a Naive Bayes Classifier

Construct a naive Bayes classifier for Fisher's iris data. Also, specify prior probabilities after training.

Load Fisher's iris data.

```
load fisheriris
X = meas;
Y = species;
```

X is a numeric matrix that contains four petal measurements for 150 irises. Y is a cell array of character vectors that contains the corresponding iris species.

Train a naive Bayes classifier.

```
Mdl = fitcnb(X,Y)
```

```
Mdl =

ClassificationNaiveBayes
    ResponseName: 'Y'
    CategoricalPredictors: []
        ClassNames: {'setosa' 'versicolor' 'virginica'}
        ScoreTransform: 'none'
        NumObservations: 150
        DistributionNames: {'normal' 'normal' 'normal' 'normal'}
        DistributionParameters: {3x4 cell}
```

Mdl is a trained ClassificationNaiveBayes classifier, and some of its properties display in the Command Window. By default, the software treats each predictor as independent, and fits them using normal distributions.

To access the properties of Mdl, use dot notation.

```
Mdl.ClassNames
Mdl.Prior
```

```
ans =

3x1 cell array

    {'setosa'    }
    {'versicolor'}
    {'virginica' }
```

```
ans =
```

```
0.3333    0.3333    0.3333
```

`Mdl.Prior` contains the class prior probabilities, which are settable using the name-value pair argument `'Prior'` in `fitcnb`. The order of the class prior probabilities corresponds to the order of the classes in `Mdl.ClassNames`. By default, the prior probabilities are the respective relative frequencies of the classes in the data.

You can also reset the prior probabilities after training. For example, set the prior probabilities to 0.5, 0.2, and 0.3 respectively.

```
Mdl.Prior = [0.5 0.2 0.3];
```

You can pass `Mdl` to e.g., `predict` to label new measurements, or `crossval` to cross validate the classifier.

## Definitions

### Bag-of-Tokens Model

In the bag-of-tokens model, the value of predictor  $j$  is the nonnegative number of occurrences of token  $j$  in this observation. The number of categories (bins) in this multinomial model is the number of distinct tokens, that is, the number of predictors.

### Naive Bayes

Naive Bayes is a classification algorithm that applies density estimation to the data.

The algorithm leverages Bayes theorem, and (naively) assumes that the predictors are conditionally independent, given the class. Though the assumption is usually violated in practice, naive Bayes classifiers tend to yield posterior distributions that are robust to biased class density estimates, particularly where the posterior is 0.5 (the decision boundary) [1].

Naive Bayes classifiers assign observations to the most probable class (in other words, the maximum a posteriori decision rule). Explicitly, the algorithm:

- 1 Estimates the densities of the predictors within each class.



- 2 Models posterior probabilities according to Bayes rule. That is, for all  $k = 1, \dots, K$ ,

$$\hat{P}(Y = k | X_1, \dots, X_P) = \frac{\pi(Y = k) \prod_{j=1}^P P(X_j | Y = k)}{\sum_{k=1}^K \pi(Y = k) \prod_{j=1}^P P(X_j | Y = k)},$$

where:

- $Y$  is the random variable corresponding to the class index of an observation.
  - $X_1, \dots, X_P$  are the random predictors of an observation.
  - $\pi(Y = k)$  is the prior probability that a class index is  $k$ .
- 3 Classifies an observation by estimating the posterior probability for each class, and then assigns the observation to the class yielding the maximum posterior probability.

If the predictors compose a multinomial distribution, then the posterior probability

$\hat{P}(Y = k | X_1, \dots, X_P) \propto \pi(Y = k) P_{mn}(X_1, \dots, X_P | Y = k)$ , where  $P_{mn}(X_1, \dots, X_P | Y = k)$  is the probability mass function of a multinomial distribution.

## Algorithms

- If you specify 'Distribution', 'mn' when training Mdl using `fitcnb`, then the software fits a multinomial distribution using the bag-of-tokens model on page 32-2082. The software stores the probability that token  $j$  appears in class  $k$  in the property `DistributionParameters{k, j}`. Using additive smoothing [2], the estimated probability is

$$P(\text{token } j | \text{class } k) = \frac{1 + c_{jk}}{P + c_k},$$

where:

•

$c_{j|k} = n_k \frac{\sum_{i:y_i \in \text{class } k} x_{ij} w_i}{\sum_{i \in \text{class } k} w_i}$ ; which is the weighted number of occurrences of token  $j$  in class  $k$ .

•  $n_k$  is the number of observations in class  $k$ .

•

$w_i$  is the weight for observation  $i$ . The software normalizes weights within a class such that they sum to the prior probability for that class.

•

$c_k = \sum_{j=1}^P c_{j|k}$ ; which is the total weighted number of occurrences of all tokens in class  $k$ .

• If you specify 'Distribution', 'mvnmn' when training Mdl using fitcnb, then:

- 1 For each predictor, the software collects a list of the unique levels, stores the sorted list in CategoricalLevels, and considers each level a bin. Each predictor/class combination is a separate, independent multinomial random variable.
- 2 For predictor  $j$  in class  $k$ , the software counts instances of each categorical level using the list stored in CategoricalLevels{ $j$ }.
- 3 The software stores the probability that predictor  $j$ , in class  $k$ , has level  $L$  in the property DistributionParameters{ $k, j$ }, for all levels in CategoricalLevels{ $j$ }. Using additive smoothing [2], the estimated probability is

$$P(\text{predictor } j = L \mid \text{class } k) = \frac{1 + m_{j|k}(L)}{m_j + m_k},$$

where:

•

$$m_{j|k}(L) = n_k \frac{\sum_{i: y_i \in \text{class } k} I\{x_{ij} = L\} w_i}{\sum_{i: y_i \in \text{class } k} w_i}$$
; which is the weighted number of observations for which predictor  $j$  equals  $L$  in class  $k$ .

•  $n_k$  is the number of observations in class  $k$ .

•

$I\{x_{ij} = L\} = 1$  if  $x_{ij} = L$ , 0 otherwise.

•

$w_i$  is the weight for observation  $i$ . The software normalizes weights within a class such that they sum to the prior probability for that class.

•  $m_j$  is the number of distinct levels in predictor  $j$ .•  $m_k$  is the weighted number of observations in class  $k$ .

## References

- [1] Hastie, T., R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*, Second Edition. NY: Springer, 2008.
- [2] Manning, C. D., P. Raghavan, and M. Schütze. *Introduction to Information Retrieval*, NY: Cambridge University Press, 2008.

## See Also

`CompactClassificationNaiveBayes` | `fitcnb` | `loss` | `predict`

## Topics

- “Naive Bayes Classification” on page 21-2  
 “Grouping Variables” on page 2-59

## ClassificationPartitionedECOC class

### Superclasses:

Cross-validated multiclass model for support vector machines or other classifiers

### Description

`ClassificationPartitionedECOC` is a set of error-correcting output codes (ECOC) models trained on cross-validated folds. Estimate the quality of classification by cross-validation using one or more “kfold” functions: `kfoldPredict`, `kfoldLoss`, `kfoldMargin`, `kfoldEdge`, and `kfoldfun`.

Every “kfold” method uses models trained on in-fold observations to predict the response for out-of-fold observations. For example, suppose you cross-validate using five folds. In this case, the software randomly assigns each observation into five roughly equal-sized groups. The training fold contains four of the groups (i.e., roughly 4/5 of the data) and the test fold contains the other group (i.e., roughly 1/5 of the data). In this case, cross-validation proceeds as follows.

- 1 The software trains the first model (stored in `CVMD1.Trained{1}`) using the observations in the last four groups and reserves the observations in the first group for validation.
- 2 The software trains the second model (stored in `CVMD1.Trained{2}`) using the observations in the first group and last three groups, and reserves the observations in the second group for validation.
- 3 The software proceeds in a similar fashion for the third, fourth, and fifth models.

If you validate by calling `kfoldPredict`, it computes predictions for the observations in group 1 using the first model, group 2 for the second model, and so on. In short, the software estimates a response for every observation using the model trained without that observation.

## Construction

`CVMdl = crossval(Mdl)` returns a cross-validated ECOC model from `Mdl`, an ECOC model. For details, see `crossval`.

Alternatively, `CVMdl = fitcecoc(X,Y,Name,Value)` returns a cross-validated ECOC model when `Name` is one of `'CrossVal'`, `'Kfold'`, `'Holdout'`, `'Leaveout'`, or `'CVPartition'`. For syntax details, see `fitcecoc`.

## Properties

### Cross-Validation Properties

#### **CrossValidatedModel** — Cross-validated model name

character vector

Cross-validated model name, specified as a character vector.

For example, `'ECOC'` specifies a cross-validated ECOC model.

Data Types: `char`

#### **KFold** — Number of cross-validated folds

positive integer

Number of cross-validated folds, specified as a positive integer.

Data Types: `double`

#### **ModelParameters** — Cross-validation parameter values

object

Cross-validation parameter values, e.g., the name-value pair argument values used to cross-validate the ECOC classifier, specified as an object. `ModelParameters` does not contain estimated parameters.

Access properties of `ModelParameters` using dot notation.

#### **NumObservations** — Number of observations

positive numeric scalar

Number of observations in the training data, specified as a positive numeric scalar.

Data Types: `double`

**Partition — Data partition**

`cvpartition` model

Data partition indicating how the software splits the data into cross-validation folds, specified as a `cvpartition` model.

**Trained — Compact classifiers trained on cross-validation folds**

cell array of `CompactClassificationECOC` models

Compact classifiers trained on cross-validation folds, specified cell as a array of `CompactClassificationECOC` models. `Trained` has  $k$  cells, where  $k$  is the number of folds.

Data Types: `cell`

**w — Observation weights**

numeric vector

Observation weights used to cross-validate the model, specified as a numeric vector.  $W$  has `NumObservations` elements.

The software normalizes the weights used for training so that `nansum(W)` is 1.

Data Types: `single` | `double`

**x — Unstandardized predictor data used to cross-validate the classifier**

numeric matrix

Unstandardized predictor data used to cross-validate the classifier, specified as a numeric matrix.  $X$  is a `NumObservations`-by- $p$  matrix, where  $p$  is the number of predictors.

Each row of  $X$  corresponds to one observation, and each column corresponds to one variable.

Data Types: `single` | `double`

**y — Observed class labels**

categorical array | character array | logical vector | vector of numeric values | cell array of character vectors

Observed class labels used to cross-validate the model, specified as a categorical or character array, logical or numeric vector, or cell array of character vectors. `Y` has `NumObservations` elements, and is the same data type as the input argument `Y` that you passed to `fitcecoc` to cross-validate the model.

Each row of `Y` represents the observed classification of the corresponding row of `X`.

Data Types: `char` | `cell` | `categorical` | `logical` | `single` | `double`

### ECOC Properties

#### **BinaryLoss** — Binary learner loss function

character vector representing the loss function name

Binary learner loss function, specified as a character vector representing the loss function name.

If you train using binary learners that use different loss functions, then the software sets `BinaryLoss` to `'hamming'`. To potentially increase accuracy, set a different binary loss function than this default during prediction or loss computation using the `BinaryLoss` name-value pair argument of `predict` or `loss`.

Data Types: `char`

#### **BinaryY** — Binary learner class labels

numeric matrix | []

Binary learner class labels, specified as a numeric matrix or [].

- If the coding matrix is the same across folds, then `BinaryY` is a `NumObservations`-by-`L` matrix, where `L` is the number of binary learners (`size(CodingMatrix,2)`).

Elements of `BinaryY` are -1, 0, or 1, and the value corresponds to a dichotomous class assignment. This table describes how learner `j` assigns observation `k` to a dichotomous class corresponding to the value of `BinaryY(k,j)`.

Value	Dichotomous Class Assignment
-1	Negative class
0	Before training, learner <code>j</code> removes observations in class <code>i</code> from the data set.
1	Positive class

- If the coding matrix varies across folds, then `BinaryY` is empty (`[]`).

Data Types: `double`

### **CodingMatrix** — Codes specifying class assignments

numeric matrix | `[]`

Codes specifying class assignments for the binary learners, specified as a numeric matrix or `[]`.

- If the coding matrix is the same across folds, then `CodingMatrix` is a  $K$ -by- $L$  matrix, where  $K$  is the number of classes and  $L$  is the number of binary learners.

Elements of `CodingMatrix` are -1, 0, or 1, and the value corresponds to a dichotomous class assignment. This table describes how learner  $j$  assigns observations in class  $k$  to a dichotomous class corresponding to the value of `CodingMatrix(k, j)`.

Value	Dichotomous Class Assignment
-1	Negative class
0	Before training, learner $j$ removes observations in class $i$ from the data set.
1	Positive class

- If the coding matrix varies across folds, then `CodingMatrix` is empty (`[]`). Obtain the coding matrix for each fold using the `Trained` property. For example, `CVMDL.Trained{1}.CodingMatrix` is the coding matrix in the first fold of the cross-validated ECOC model `CVMDL`.

Data Types: `char` | `double` | `single` | `int8` | `int16` | `int32` | `int64`

### **Other Classification Properties**

#### **CategoricalPredictors** — Indices of categorical predictors

vector of positive integers

Indices of categorical predictors, stored as a vector of positive integers. `CategoricalPredictors` contains index values corresponding to the columns of the predictor data that contain categorical predictors. If none of the predictors are categorical, then this property is empty (`[]`).

Data Types: `single` | `double`



**ClassNames — Unique class labels**

categorical array | character array | logical vector | vector of numeric values | cell array of character vectors

Unique class labels used in training, specified as a categorical or character array, logical or numeric vector, or cell array of character vectors. `ClassNames` has the same data type as class labels ( $\mathcal{Y}$ ). `ClassNames` also determines class order.

Data Types: `categorical` | `char` | `logical` | `single` | `double` | `cell`

**Cost — Misclassification costs**

square numeric matrix

This property is read-only.

Misclassification costs, specified as a square numeric matrix. `Cost` has  $K$  rows and columns, where  $K$  is the number of classes.

$\text{Cost}(i, j)$  is the cost of classifying a point into class  $j$  if its true class is  $i$ . The order of the rows and columns of `Cost` corresponds to the order of the classes in `ClassNames`.

`fitcecoc` incorporates misclassification costs differently among different types of binary learners.

Data Types: `double`

**PredictorNames — Predictor names**

cell array of character vectors

Predictor names in their order of appearance in predictor data ( $\mathbf{X}$ ), specified as a cell array of character vectors. `PredictorNames` has length equal to the number of columns in  $\mathbf{X}$ .

Data Types: `cell`

**Prior — Prior class probabilities**

numeric vector

This property is read-only.

Prior class probabilities, specified as a numeric vector. `Prior` has as many elements as classes in `ClassNames`, and the order of the elements corresponds to the elements of `ClassNames`.

`fitcecoc` incorporates misclassification costs differently among different types of binary learners.

Data Types: `double`

**ResponseName** — Response variable name

character array

Response variable name, specified as a character array.

Data Types: `char`

**ScoreTransform** — Score transformation function to apply to predicted scores

function name | function handle

Score transformation function to apply to predicted scores, specified as a function name or function handle.

For linear classification models and before transformation, the predicted classification score for the observation  $x$  (row vector) is  $f(x) = x\beta + b$ , where  $\beta$  and  $b$  correspond to `Mdl.Beta` and `Mdl.Bias`, respectively.

To change the score transformation function to, e.g., *function*, use dot notation.

- For a built-in function, enter this code and replace *function* with a character vector in the table.

```
Mdl.ScoreTransform = 'function';
```

Value	Description
'doublelogit'	$1/(1 + e^{-2x})$
'invlogit'	$\log(x / (1-x))$
'ismax'	Set the score for the class with the largest score to 1, and set the scores for all other classes to 0.
'logit'	$1/(1 + e^{-x})$
'none' or 'identity'	$x$ (no transformation)

Value	Description
'sign'	-1 for $x < 0$ 0 for $x = 0$ 1 for $x > 0$
'symmetric'	$2x - 1$
'symmetricismax'	Set the score for the class with the largest score to 1, and set the scores for all other classes to -1.
'symmetriclogit'	$2/(1 + e^{-x}) - 1$

- For a MATLAB function, or a function that you define, enter its function handle.

```
Mdl.ScoreTransform = @function;
```

*function* must accept a matrix of the original scores for each class, and then return a matrix of the same size representing the transformed scores for each class.

Data Types: char | function\_handle

## Methods

kfoldEdge	Classification edge for observations not used for training
kfoldfun	Cross validate function
kfoldLoss	Classification loss for observations not used for training
kfoldMargin	Classification margins for observations not used for training
kfoldPredict	Predict responses for observations not used for training

## Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects (MATLAB).

## Examples

### Cross-Validate ECOC Classifier

Load Fisher's iris data set.

```
load fisheriris
X = meas;
Y = species;
rng(1); % For reproducibility
```

Create an SVM template. It is good practice to standardize the predictors.

```
t = templateSVM('Standardize',1)
```

```
t =
```

Fit template for classification SVM.

```
          Alpha: [0x1 double]
      BoxConstraint: []
          CacheSize: []
      CachingMethod: ''
          ClipAlphas: []
DeltaGradientTolerance: []
          Epsilon: []
          GapTolerance: []
          KKTolerance: []
      IterationLimit: []
      KernelFunction: ''
          KernelScale: []
          KernelOffset: []
KernelPolynomialOrder: []
          NumPrint: []
              Nu: []
      OutlierFraction: []
      RemoveDuplicates: []
      ShrinkagePeriod: []
          Solver: ''
      StandardizeData: 1
      SaveSupportVectors: []
          VerbosityLevel: []
          Version: 2
          Method: 'SVM'
          Type: 'classification'
```

`t` is an SVM template. All of its properties are empty, except for `StandardizeData`, `Method`, and `Type`. When the software trains the ECOC classifier, it sets the applicable properties to their default values.

Train the ECOC classifier. It is good practice to specify the class order.

```
Mdl = fitcecoc(X,Y,'Learners',t,...
    'ClassNames',{'setosa','versicolor','virginica'});
```

`Mdl` is a `ClassificationECOC` classifier. You can access its properties using dot notation.

Cross-validate `Mdl` using 10-fold cross-validation.

```
CVMDL = crossval(Mdl);
```

`CVMDL` is a `ClassificationPartitionedECOC` cross-validated ECOC classifier.

Estimate the generalization error.

```
oosLoss = kfoldLoss(CVMDL)
```

```
oosLoss =
    0.0400
```

The out-of-sample classification error is 4%, which indicates that the ECOC classifier generalizes fairly well.

## Train ECOC Classifiers Using Ensembles and Parallel Computing

Train a one-versus-all ECOC classifier using a GentleBoost ensemble of decision trees with surrogate splits. Estimate the classification error using 10-fold cross-validation.

Load and inspect the `arrhythmia` data set.

```
load arrhythmia
[n,p] = size(X)
isLabels = unique(Y);
nLabels = numel(isLabels)
tabulate(categorical(Y))
```

```
n =  
  452  
  
p =  
  279  
  
nLabels =  
  13  
  
  Value    Count    Percent  
    1      245    54.20%  
    2       44     9.73%  
    3       15     3.32%  
    4       15     3.32%  
    5       13     2.88%  
    6       25     5.53%  
    7        3     0.66%  
    8        2     0.44%  
    9        9     1.99%  
   10       50    11.06%  
   14        4     0.88%  
   15        5     1.11%  
   16       22     4.87%
```

There are 279 predictors, and a relatively small sample size of 452. There are 16 distinct labels, but only 13 are represented in the response ( $Y$ ), and each label describes various degrees of arrhythmia. 54.20% of the observations are in class 1.

Create an ensemble template. You must specify at least three arguments: a method, a number of learners, and the type of learner. For this example, specify 'GentleBoost' for the method, 100 for the number of learners, and a decision tree template that uses surrogate splits since there are missing observations.

```
tTree = templateTree('surrogate', 'on');  
tEnsemble = templateEnsemble('GentleBoost', 100, tTree);
```

tEnsemble is a template object. Most of its properties are empty, but the software fills them with their default values during training.

Train a one-versus-all ECOC classifier using the ensembles of decision trees as binary learners. If you have a Parallel Computing Toolbox license, then you can speed up the computation by specifying to use parallel computing. This sends each binary learner to a worker in the pool (the number of workers depends on your system configuration). Also, specify that the prior probabilities are  $1/K$ , where  $K = 13$ , which is the number of distinct classes.

```
pool = parpool; % Invoke workers
options = statset('UseParallel',true);
Mdl = fitcecoc(X,Y,'Coding','onevsall','Learners',tEnsemble,...
    'Prior','uniform','Options',options);
```

Starting parallel pool (parpool) using the 'local' profile ... connected to 2 workers.

Mdl is a ClassificationECOC model.

Cross-validate the ECOC classifier using 10-fold cross-validation.

```
CVMDL = crossval(Mdl,'Options',options);
```

Warning: One or more folds do not contain points from all the groups.

CVMDL is a ClassificationPartitionedECOC model. The warning indicates that some classes are not represented while the software trains at least one fold. Therefore, those folds cannot predict labels for the missing classes. You can inspect the results of a fold using cell indexing and dot notation, e.g., access the results of the first fold by entering `CVMDL.Trained{1}`. Your results might vary.

Use the cross-validated ECOC classifier to predict out-of-fold labels. You can compute the confusion matrix using `confusionmat`. However, if you have a Neural Network Toolbox license, you can plot the confusion matrix using `plotconfusion`. The input arguments of `plotconfusion` are not vectors of the true and predicted labels like `confusionmat`, but indicator matrices of the true and predicted labels. Both start as  $K$ -by- $n$  matrices of 0s. If observation  $j$  has label index  $k$  (or has predicted label  $k$ ), then element  $(k,j)$  of the true label indicator matrix (or predicted label indicator matrix) is 1. You can convert label indices returned by `predict`, `resubPredict`, or `kfoldPredict` to label indicator matrices using linear indexing. For details on linear indexing, see `sub2ind` and `ind2sub`.

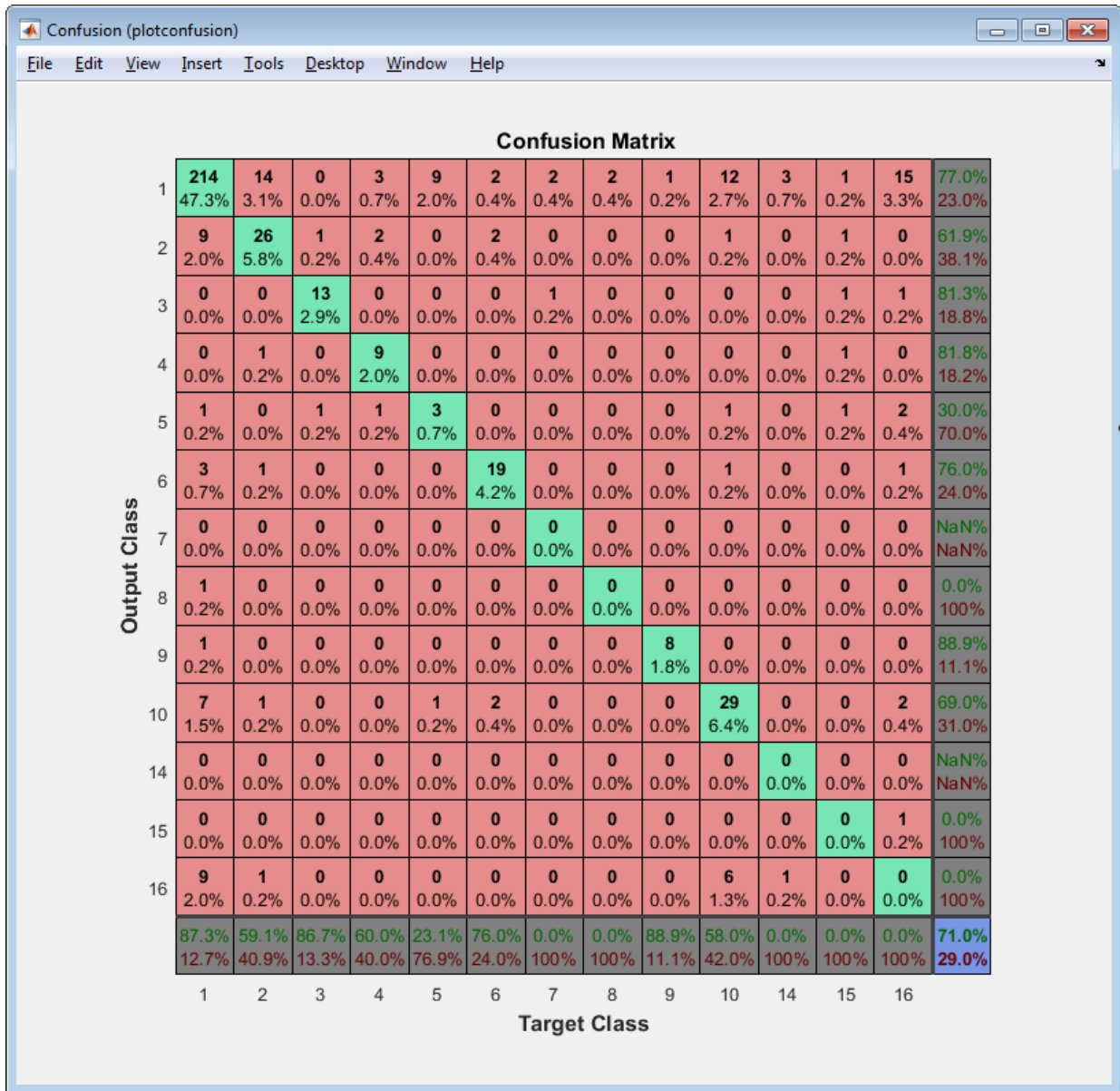
```
oofLabel = kfoldPredict(CVMDL,'Options',options);
ConfMat = confusionmat(Y,oofLabel);
```

```
% Convert the integer label vector to a class-identifier matrix.
```

```
[~,grpOOF] = ismember(oofLabel,isLabels);
oofLabelMat = zeros(nLabels,n);
idxLinear = sub2ind([nLabels n],grpOOF,(1:n)');
oofLabelMat(idxLinear) = 1; % Flags the row corresponding to the class
[~,grpY] = ismember(Y,isLabels);
YMat = zeros(nLabels,n);
idxLinearY = sub2ind([nLabels n],grpY,(1:n)');
YMat(idxLinearY) = 1;

figure;
plotconfusion(YMat,oofLabelMat);
h = gca;
h.XTickLabel = [num2cell(isLabels); {' '}];
h.YTickLabel = [num2cell(isLabels); {' '}];
```





## Algorithms

For linear, SVM binary learners, and for efficiency, `fitcecoc` empties the properties

Alpha, SupportVectorLabels, and SupportVectors. `fitcecoc` lists Beta, rather than Alpha, in the model display.

To store Alpha, SupportVectorLabels, and SupportVectors, pass a linear, SVM template that specifies storing support vectors to `fitcecoc`. For example, enter:

```
t = templateSVM('SaveSupportVectors','on')
Mdl = fitcecoc(X,Y,'Learners',t);
```

You can remove the support vectors and related values by passing the resulting ClassificationECOC model to `discardSupportVectors`.

## See Also

[ClassificationECOC](#) | [CompactClassificationECOC](#) | [cvpartiton](#) | [fitcecoc](#)

# ClassificationPartitionedEnsemble class

Cross-validated classification ensemble

## Description

`ClassificationPartitionedEnsemble` is a set of classification ensembles trained on cross-validated folds. Estimate the quality of classification by cross validation using one or more “kfold” methods: `kfoldPredict`, `kfoldLoss`, `kfoldMargin`, `kfoldEdge`, and `kfoldfun`.

Every “kfold” method uses models trained on in-fold observations to predict response for out-of-fold observations. For example, suppose you cross validate using five folds. In this case, every training fold contains roughly 4/5 of the data and every test fold contains roughly 1/5 of the data. The first model stored in `Trained{1}` was trained on `X` and `Y` with the first 1/5 excluded, the second model stored in `Trained{2}` was trained on `X` and `Y` with the second 1/5 excluded, and so on. When you call `kfoldPredict`, it computes predictions for the first 1/5 of the data using the first model, for the second 1/5 of data using the second model, and so on. In short, response for every observation is computed by `kfoldPredict` using the model trained without this observation.

## Construction

`cvens = crossval(ens)` creates a cross-validated ensemble from `ens`, a classification ensemble. For syntax details, see the `crossval` method reference page.

`cvens = fitcensemble(X,Y,Name,Value)` creates a cross-validated ensemble when `Name` is one of `'CrossVal'`, `'KFold'`, `'Holdout'`, `'Leaveout'`, or `'CVPartition'`. For syntax details, see the `fitcensemble` function reference page.

## Properties

### **CategoricalPredictors**

Indices of categorical predictors, stored as a vector of positive integers. `CategoricalPredictors` contains index values corresponding to the columns of the predictor data that contain categorical predictors. If none of the predictors are categorical, then this property is empty (`[]`).

### **ClassNames**

List of the elements in `Y` with duplicates removed. `ClassNames` can be a numeric vector, vector of categorical variables, logical vector, character array, or cell array of character vectors. `ClassNames` has the same data type as the data in the argument `Y`.

### **Combiner**

Cell array of combiners across all folds.

### **Cost**

Square matrix, where `Cost(i, j)` is the cost of classifying a point into class `j` if its true class is `i` (i.e., the rows correspond to the true class and the columns correspond to the predicted class). The order of the rows and columns of `Cost` corresponds to the order of the classes in `ClassNames`. The number of rows and columns in `Cost` is the number of unique classes in the response. This property is read-only.

### **CrossValidatedModel**

Name of the cross-validated model, a character vector.

### **Kfold**

Number of folds used in a cross-validated ensemble, a positive integer.

### **ModelParameters**

Object holding parameters of `cvens`.

### **NumObservations**

Number of data points used in training the ensemble, a positive integer.

**NTrainedPerFold**

Number of data points used in training each fold of the ensemble, a positive integer.

**Partition**

Partition of class `cvpartition` used in creating the cross-validated ensemble.

**PredictorNames**

Cell array of names for the predictor variables, in the order in which they appear in `X`.

**Prior**

Numeric vector of prior probabilities for each class. The order of the elements of `Prior` corresponds to the order of the classes in `ClassNames`. The number of elements of `Prior` is the number of unique classes in the response. This property is read-only.

**ResponseName**

Name of the response variable `Y`, a character vector.

**ScoreTransform**

Function handle for transforming scores, or character vector representing a built-in transformation function. 'none' means no transformation; equivalently, 'none' means  $@(x)x$ . For a list of built-in transformation functions and the syntax of custom transformation functions, see `fitctree`.

Add or change a `ScoreTransform` function using dot notation:

```
ens.ScoreTransform = 'function'
```

or

```
ens.ScoreTransform = @function
```

**Trainable**

Cell array of ensembles trained on cross-validation folds. Every ensemble is full, meaning it contains its training data and weights.

**Trained**

Cell array of compact ensembles trained on cross-validation folds.

**W**

Scaled `weights`, a vector with length `n`, the number of rows in `X`.

**X**

A matrix of predictor values. Each column of `X` represents one variable, and each row represents one observation.

**Y**

A numeric column vector with the same number of rows as `X`. Each entry in `Y` is the response to the data in the corresponding row of `X`.

## Methods

<code>kfoldEdge</code>	Classification edge for observations not used for training
<code>kfoldLoss</code>	Classification loss for observations not used for training
<code>resume</code>	Resume training learners on cross-validation folds

## Inherited Methods

<code>kfoldEdge</code>	Classification edge for observations not used for training
<code>kfoldfun</code>	Cross validate function
<code>kfoldLoss</code>	Classification loss for observations not used for training
<code>kfoldMargin</code>	Classification margins for observations not used for training
<code>kfoldPredict</code>	Predict response for observations not used for training

## Copy Semantics

Value. To learn how value classes affect copy operations, see [Copying Objects \(MATLAB\)](#).

## Examples

### Evaluate K-Fold Cross-Validation Error for Classification Ensemble

Evaluate the k-fold cross-validation error for a classification ensemble that models the Fisher iris data.

Load the sample data set.

```
load fisheriris
```

Train an ensemble of 100 boosted classification trees using AdaBoostM2.

```
t = templateTree('MaxNumSplits',1); % Weak learner template tree object
ens = fitcensemble(meas,species,'Method','AdaBoostM2','Learners',t);
```

Create a cross-validated ensemble from ens and find the k-fold cross-validation error.

```
rng(10,'twister') % For reproducibility
cvens = crossval(ens);
L = kfoldLoss(cvens)

L = 0.0467
```

## See Also

[ClassificationEnsemble](#) | [ClassificationPartitionedModel](#) |  
[RegressionPartitionedEnsemble](#) | [fitctree](#)

## ClassificationPartitionedLinear class

Cross-validated linear model for binary classification of high-dimensional data

### Description

`ClassificationPartitionedLinear` is a set of linear classification models trained on cross-validated folds. To obtain a cross-validated, linear classification model, use `fitclinear` and specify one of the cross-validation options. You can estimate the quality of classification, or how well the linear classification model generalizes, using one or more of these “kfold” methods: `kfoldPredict`, `kfoldLoss`, `kfoldMargin`, and `kfoldEdge`.

Every “kfold” method uses models trained on in-fold observations to predict the response for out-of-fold observations. For example, suppose that you cross-validate using five folds. In this case, the software randomly assigns each observation into five roughly equally sized groups. The training fold contains four of the groups (that is, roughly 4/5 of the data) and the test fold contains the other group (that is, roughly 1/5 of the data). In this case, cross-validation proceeds as follows:

- 1 The software trains the first model (stored in `CVMdl.Trained{1}`) using the observations in the last four groups and reserves the observations in the first group for validation.
- 2 The software trains the second model, which is stored in `CVMdl.Trained{2}`, using the observations in the first group and last three groups. The software reserves the observations in the second group for validation.
- 3 The software proceeds in a similar fashion for the third through fifth models.

If you validate by calling `kfoldPredict`, it computes predictions for the observations in group 1 using the first model, group 2 for the second model, and so on. In short, the software estimates a response for every observation using the model trained without that observation.

---

**Note** Unlike other cross-validated, classification models, `ClassificationPartitionedLinear` model objects do not store the predictor data set.

---



## Construction

`CVMdl = fitcllinear(X,Y,Name,Value)` creates a cross-validated, linear classification model when `Name` is either `'CrossVal'`, `'CVPartition'`, `'Holdout'`, or `'Kfold'`. For more details, see `fitcllinear`.

## Properties

### Cross-Validation Properties

#### **CrossValidatedModel** — Cross-validated model name

character array

Cross-validated model name, specified as a character array.

For example, `'Linear'` specifies a cross-validated linear model for binary classification or regression.

Data Types: `char`

#### **Kfold** — Number of cross-validated folds

positive integer

Number of cross-validated folds, specified as a positive integer.

Data Types: `double`

#### **ModelParameters** — Cross-validation parameter values

object

Cross-validation parameter values, e.g., the name-value pair argument values used to cross-validate the linear model, specified as an object. `ModelParameters` does not contain estimated parameters.

Access properties of `ModelParameters` using dot notation.

#### **NumObservations** — Number of observations

positive numeric scalar

Number of observations in the training data, specified as a positive numeric scalar.

Data Types: `double`

**Partition — Data partition**

`cvpartition` model

Data partition indicating how the software splits the data into cross-validation folds, specified as a `cvpartition` model.

**Trained — Linear classification models trained on cross-validation folds**

cell array of `ClassificationLinear` model objects

Linear classification models trained on cross-validation folds, specified as a cell array of `ClassificationLinear` models. `Trained` has  $k$  cells, where  $k$  is the number of folds.

Data Types: `cell`

**w — Observation weights**

numeric vector

Observation weights used to cross-validate the model, specified as a numeric vector.  $W$  has `NumObservations` elements.

The software normalizes  $W$  so that the weights for observations within a particular class sum up to the prior probability of that class.

Data Types: `single` | `double`

**Y — Observed class labels**

categorical array | character array | logical vector | vector of numeric values | cell array of character vectors

Observed class labels used to cross-validate the model, specified as a categorical or character array, logical or numeric vector, or cell array of character vectors.  $Y$  has `NumObservations` elements, and is the same data type as the input argument  $Y$  that you passed to `fitclinear` to cross-validate the model.

Each row of  $Y$  represents the observed classification of the corresponding observation in the predictor data.

Data Types: `char` | `cell` | `categorical` | `logical` | `single` | `double`

**Other Classification Properties****CategoricalPredictors — Indices of categorical predictors**

vector of positive integers

Indices of categorical predictors, stored as a vector of positive integers. `CategoricalPredictors` contains index values corresponding to the columns of the predictor data that contain categorical predictors. If none of the predictors are categorical, then this property is empty (`[]`).

Data Types: `single` | `double`**ClassNames — Unique class labels**

categorical array | character array | logical vector | vector of numeric values | cell array of character vectors

Unique class labels used in training, specified as a categorical or character array, logical or numeric vector, or cell array of character vectors. `ClassNames` has the same data type as class labels (`Y`). `ClassNames` also determines class order.

Data Types: `categorical` | `char` | `logical` | `single` | `double` | `cell`**Cost — Misclassification costs**

square numeric matrix

This property is read-only.

Misclassification costs, specified as a square numeric matrix. `Cost` has  $K$  rows and columns, where  $K$  is the number of classes.

$Cost(i, j)$  is the cost of classifying a point into class  $j$  if its true class is  $i$ . The order of the rows and columns of `Cost` corresponds to the order of the classes in `ClassNames`.

Data Types: `double`**PredictorNames — Predictor names**

cell array of character vectors

Predictor names in their order of appearance in predictor data used to train `CVMD1`, specified as a cell array of character vectors. `PredictorNames` has length equal to the number of predictor variables in the predictor data.

Data Types: `cell`

**Prior — Prior class probabilities**

numeric vector

This property is read-only.

Prior class probabilities, specified as a numeric vector. `Prior` has as many elements as classes in `ClassNames`, and the order of the elements corresponds to the elements of `ClassNames`.

`fitcecoc` incorporates misclassification costs differently among different types of binary learners.

Data Types: `double`

**ResponseName — Response variable name**

character array

Response variable name, specified as a character array.

Data Types: `char`

**ScoreTransform — Score transformation function to apply to predicted scores**

function name | function handle

Score transformation function to apply to predicted scores, specified as a function name or function handle.

For linear classification models and before transformation, the predicted classification score for the observation  $x$  (row vector) is  $f(x) = x\beta + b$ , where  $\beta$  and  $b$  correspond to `Mdl.Beta` and `Mdl.Bias`, respectively.

To change the score transformation function to, e.g., `function`, use dot notation.

- For a built-in function, enter this code and replace `function` with a character vector in the table.

```
Mdl.ScoreTransform = 'function';
```

Value	Description
'doublelogit'	$1/(1 + e^{-2x})$
'invlogit'	$\log(x / (1-x))$

Value	Description
'ismax'	Set the score for the class with the largest score to 1, and set the scores for all other classes to 0.
'logit'	$1/(1 + e^{-x})$
'none' or 'identity'	$x$ (no transformation)
'sign'	-1 for $x < 0$ 0 for $x = 0$ 1 for $x > 0$
'symmetric'	$2x - 1$
'symmetricismax'	Set the score for the class with the largest score to 1, and set the scores for all other classes to -1.
'symmetriclogit'	$2/(1 + e^{-x}) - 1$

- For a MATLAB function, or a function that you define, enter its function handle.

```
Mdl.ScoreTransform = @function;
```

*function* must accept a matrix of the original scores for each class, and then return a matrix of the same size representing the transformed scores for each class.

Data Types: char | function\_handle

## Methods

kfoldEdge      Classification edge for observations not used for training  
kfoldLoss      Classification loss for observations not used in training  
kfoldMargin    Classification margins for observations not used in training  
kfoldPredict    Predict labels for observations not used for training

## Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects (MATLAB).

## Examples

### Create Cross-Validated Binary Linear Classification Model

Load the NLP data set.

```
load nlpdata
```

X is a sparse matrix of predictor data, and Y is a categorical vector of class labels. There are more than two classes in the data.

Identify the labels that correspond to the Statistics and Machine Learning Toolbox™ documentation web pages.

```
Ystats = Y == 'stats';
```

Cross-validate a binary, linear classification model that can identify whether the word counts in a documentation web page are from the Statistics and Machine Learning Toolbox™ documentation.

```
rng(1); % For reproducibility
CVMdl = fitclinear(X, Ystats, 'CrossVal', 'on')
```

```
CVMdl =
```

```
classreg.learning.partition.ClassificationPartitionedLinear
  CrossValidatedModel: 'Linear'
  ResponseName: 'Y'
  NumObservations: 31572
  KFold: 10
  Partition: [1x1 cvpartition]
  ClassNames: [0 1]
  ScoreTransform: 'none'
```

CVMdl is a ClassificationPartitionedLinear cross-validated model. Because fitclinear implements 10-fold cross-validation by default, CVMdl.Trained contains ten ClassificationLinear models that contain the results of training linear classification models for each of the folds.

Estimate labels for out-of-fold observations and estimate the generalization error by passing `CVMDL` to `kfoldPredict` and `kfoldLoss`, respectively.

```
oofLabels = kfoldPredict(CVMDL);
ge = kfoldLoss(CVMDL)
```

```
ge =
    7.6017e-04
```

The estimated generalization error is less than 0.1% misclassified observations.

### Find Good Lasso Penalty Using Cross-Validation

To determine a good lasso-penalty strength for a linear classification model that uses a logistic regression learner, implement 5-fold cross-validation.

Load the NLP data set.

```
load nlpdata
```

`X` is a sparse matrix of predictor data, and `Y` is a categorical vector of class labels. There are more than two classes in the data.

The models should identify whether the word counts in a web page are from the Statistics and Machine Learning Toolbox™ documentation. So, identify the labels that correspond to the Statistics and Machine Learning Toolbox™ documentation web pages.

```
Ystats = Y == 'stats';
```

Create a set of 11 logarithmically-spaced regularization strengths from  $10^{-6}$  through  $10^{-0.5}$ .

```
Lambda = logspace(-6, -0.5, 11);
```

Cross-validate the models. To increase execution speed, transpose the predictor data and specify that the observations are in columns. Estimate the coefficients using `Sparse`. Lower the tolerance on the gradient of the objective function to  $1e-8$ .

```
X = X';
rng(10); % For reproducibility
```

```
CVMdl = fitclinear(X,Ystats,'ObservationsIn','columns','KFold',5,...  
    'Learner','logistic','Solver','sparsa','Regularization','lasso',...  
    'Lambda',Lambda,'GradientTolerance',1e-8)
```

```
numCLModels = numel(CVMdl.Trained)
```

```
CVMdl =
```

```
classreg.learning.partition.ClassificationPartitionedLinear  
  CrossValidatedModel: 'Linear'  
    ResponseName: 'Y'  
    NumObservations: 31572  
           KFold: 5  
           Partition: [1x1 cvpartition]  
           ClassNames: [0 1]  
           ScoreTransform: 'none'
```

```
numCLModels =
```

```
5
```

CVMdl is a `ClassificationPartitionedLinear` model. Because `fitclinear` implements 5-fold cross-validation, CVMdl contains 5 `ClassificationLinear` models that the software trains on each fold.

Display the first trained linear classification model.

```
Mdl1 = CVMdl.Trained{1}
```

```
Mdl1 =
```

```
ClassificationLinear  
  ResponseName: 'Y'  
    ClassNames: [0 1]  
  ScoreTransform: 'logit'  
           Beta: [34023x11 double]  
           Bias: [1x11 double]  
           Lambda: [1x11 double]  
           Learner: 'logistic'
```



Mdl1 is a ClassificationLinear model object. fitclinear constructed Mdl1 by training on the first four folds. Because Lambda is a sequence of regularization strengths, you can think of Mdl1 as 11 models, one for each regularization strength in Lambda.

Estimate the cross-validated classification error.

```
ce = kfoldLoss(CVMdl);
```

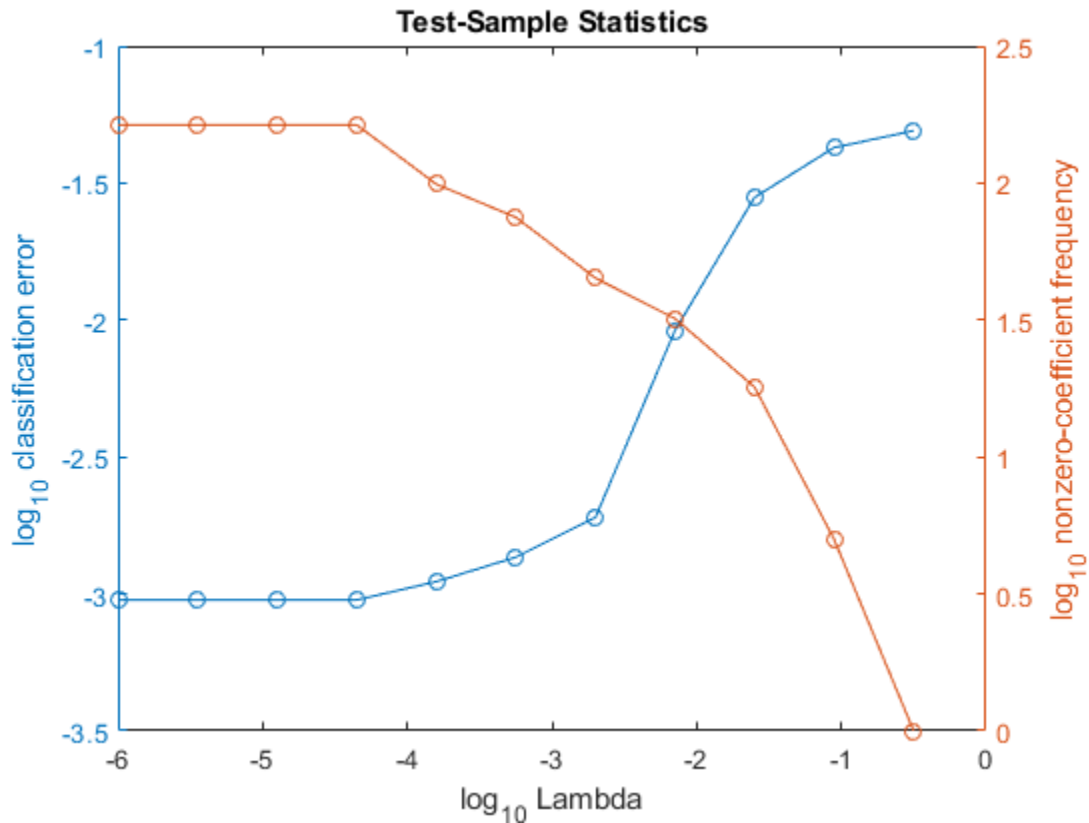
Because there are 11 regularization strengths, ce is a 1-by-11 vector of classification error rates.

Higher values of Lambda lead to predictor variable sparsity, which is a good quality of a classifier. For each regularization strength, train a linear classification model using the entire data set and the same options as when you cross-validated the models. Determine the number of nonzero coefficients per model.

```
Mdl = fitclinear(X,Ystats,'ObservationsIn','columns',...
    'Learner','logistic','Solver','sparsa','Regularization','lasso',...
    'Lambda',Lambda,'GradientTolerance',1e-8);
numNZCoeff = sum(Mdl.Beta~=0);
```

In the same figure, plot the cross-validated, classification error rates and frequency of nonzero coefficients for each regularization strength. Plot all variables on the log scale.

```
figure;
[h,hL1,hL2] = plotyy(log10(Lambda),log10(ce),...
    log10(Lambda),log10(numNZCoeff));
hL1.Marker = 'o';
hL2.Marker = 'o';
ylabel(h(1),'log_{10} classification error')
ylabel(h(2),'log_{10} nonzero-coefficient frequency')
xlabel('log_{10} Lambda')
title('Test-Sample Statistics')
hold off
```



Choose the index of the regularization strength that balances predictor variable sparsity and low classification error. In this case, a value between  $10^{-4}$  to  $10^{-1}$  should suffice.

```
idxFinal = 7;
```

Select the model from `Mdl` with the chosen regularization strength.

```
MdlFinal = selectModels(Mdl,idxFinal);
```

`MdlFinal` is a `ClassificationLinear` model containing one regularization strength. To estimate labels for new observations, pass `MdlFinal` and the new data to `predict`.

## See Also

`ClassificationLinear` | `fitlinear` | `kfoldLoss` | `kfoldPredict`

**Introduced in R2016a**

## ClassificationPartitionedLinearECOC class

Cross-validated linear error-correcting output codes model for multiclass classification of high-dimensional data

### Description

`ClassificationPartitionedLinearECOC` is a set of error-correcting output codes (ECOC) models composed of linear classification models, trained on cross-validated folds. Estimate the quality of classification by cross-validation using one or more “kfold” functions: `kfoldPredict`, `kfoldLoss`, `kfoldMargin`, and `kfoldEdge`.

Every “kfold” method uses models trained on in-fold observations to predict the response for out-of-fold observations. For example, suppose that you cross-validate using five folds. In this case, the software randomly assigns each observation into five roughly equal-sized groups. The training fold contains four of the groups (that is, roughly 4/5 of the data) and the test fold contains the other group (that is, roughly 1/5 of the data). In this case, cross-validation proceeds as follows.

- 1 The software trains the first model (stored in `CVMDL.Trained{1}`) using the observations in the last four groups and reserves the observations in the first group for validation.
- 2 The software trains the second model (stored in `CVMDL.Trained{2}`) using the observations in the first group and last three groups. The software reserves the observations in the second group for validation.
- 3 The software proceeds in a similar fashion for the third, fourth, and fifth models.

If you validate by calling `kfoldPredict`, it computes predictions for the observations in group 1 using the first model, group 2 for the second model, and so on. In short, the software estimates a response for every observation using the model trained without that observation.

---

**Note** Unlike other cross-validated ECOC models, `ClassificationPartitionedLinearECOC` model objects do not store the predictor data set.

---

## Construction

`CVMdl = fitcecoc(X,Y,'Learners',t,Name,Value)` returns a cross-validated, linear ECOC model when:

- `t` is 'Linear' or a template object returned by `templateLinear`.
- `Name` is one of 'CrossVal', 'CVPartition', 'Holdout', or 'KFold'.

For more details, see `fitcecoc`.

## Properties

### Cross-Validation Properties

#### **CrossValidatedModel** — Cross-validated model name

character vector

Cross-validated model name, specified as a character vector.

For example, 'ECOC' specifies a cross-validated ECOC model.

Data Types: `char`

#### **KFold** — Number of cross-validated folds

positive integer

Number of cross-validated folds, specified as a positive integer.

Data Types: `double`

#### **ModelParameters** — Cross-validation parameter values

object

Cross-validation parameter values, e.g., the name-value pair argument values used to cross-validate the ECOC classifier, specified as an object. `ModelParameters` does not contain estimated parameters.

Access properties of `ModelParameters` using dot notation.

#### **NumObservations** — Number of observations

positive numeric scalar

Number of observations in the training data, specified as a positive numeric scalar.

Data Types: `double`

**Partition — Data partition**

`cvpartition` model

Data partition indicating how the software splits the data into cross-validation folds, specified as a `cvpartition` model.

**Trained — Compact classifiers trained on cross-validation folds**

cell array of `CompactClassificationECOC` models

Compact classifiers trained on cross-validation folds, specified as a cell array of `CompactClassificationECOC` models. `Trained` has  $k$  cells, where  $k$  is the number of folds.

Data Types: `cell`

**w — Observation weights**

numeric vector

Observation weights used to cross-validate the model, specified as a numeric vector.  $W$  has `NumObservations` elements.

The software normalizes the weights used for training so that `nansum(W)` is 1.

Data Types: `single` | `double`

**Y — Observed class labels**

categorical array | character array | logical vector | vector of numeric values | cell array of character vectors

Observed class labels used to cross-validate the model, specified as a categorical or character array, logical or numeric vector, or cell array of character vectors.  $Y$  has `NumObservations` elements, and is the same data type as the input argument  $Y$  that you passed to `fitcecoc` to cross-validate the model.

Each row of  $Y$  represents the observed classification of the observation in the predictor data.

Data Types: `char` | `cell` | `categorical` | `logical` | `single` | `double`

**ECOC Properties****BinaryLoss** — Binary learner loss function

character vector representing the loss function name

Binary learner loss function, specified as a character vector representing the loss function name.

If you train using binary learners that use different loss functions, then the software sets `BinaryLoss` to 'hamming'. To potentially increase accuracy, set a different binary loss function than this default during prediction or loss computation using the `BinaryLoss` name-value pair argument of `predict` or `loss`.

Data Types: char

**BinaryY** — Binary learner class labels

numeric matrix | []

Binary learner class labels, specified as a numeric matrix or [].

- If the coding matrix is the same across folds, then `BinaryY` is a `NumObservations-by-L` matrix, where  $L$  is the number of binary learners (`size(CodingMatrix,2)`).

Elements of `BinaryY` are -1, 0, or 1, and the value corresponds to a dichotomous class assignment. This table describes how learner  $j$  assigns observation  $k$  to a dichotomous class corresponding to the value of `BinaryY(k,j)`.

Value	Dichotomous Class Assignment
-1	Negative class
0	Before training, learner $j$ removes observations in class $i$ from the data set.
1	Positive class

- If the coding matrix varies across folds, then `BinaryY` is empty ([]).

Data Types: double

**CodingMatrix** — Codes specifying class assignments

numeric matrix | []

Codes specifying class assignments for the binary learners, specified as a numeric matrix or [].

- If the coding matrix is the same across folds, then `CodingMatrix` is a  $K$ -by- $L$  matrix.  $K$  is the number of classes and  $L$  is the number of binary learners.

Elements of `CodingMatrix` are -1, 0, or 1, and the value corresponds to a dichotomous class assignment. This table describes how learner  $j$  assigns observations in class  $k$  to a dichotomous class corresponding to the value of `CodingMatrix(k, j)`.

Value	Dichotomous Class Assignment
-1	Negative class
0	Before training, learner $j$ removes observations in class $i$ from the data set.
1	Positive class

- If the coding matrix varies across folds, then `CodingMatrix` is empty (`[]`). Obtain the coding matrix for each fold using the `Trained` property. For example, `CVMDL.Trained{1}.CodingMatrix` is the coding matrix in the first fold of the cross-validated ECOC model `CVMDL`.

Data Types: `char` | `double` | `single` | `int8` | `int16` | `int32` | `int64`

### Other Classification Properties

#### **CategoricalPredictors** — Indices of categorical predictors

vector of positive integers

Indices of categorical predictors, stored as a vector of positive integers. `CategoricalPredictors` contains index values corresponding to the columns of the predictor data that contain categorical predictors. If none of the predictors are categorical, then this property is empty (`[]`).

Data Types: `single` | `double`

#### **ClassNames** — Unique class labels

categorical array | character array | logical vector | vector of numeric values | cell array of character vectors

Unique class labels used in training, specified as a categorical or character array, logical or numeric vector, or cell array of character vectors. `ClassNames` has the same data type as class labels (`Y`). `ClassNames` also determines class order.

Data Types: `categorical` | `char` | `logical` | `single` | `double` | `cell`



**Cost — Misclassification costs**

square numeric matrix

This property is read-only.

Misclassification costs, specified as a square numeric matrix. `Cost` has  $K$  rows and columns, where  $K$  is the number of classes.

`Cost(i, j)` is the cost of classifying a point into class  $j$  if its true class is  $i$ . The order of the rows and columns of `Cost` corresponds to the order of the classes in `ClassNames`.

`fitcecoc` incorporates misclassification costs differently among different types of binary learners.

Data Types: `double`

**PredictorNames — Predictor names**

cell array of character vectors

Predictor names in their order of appearance in predictor data used to train `CVMdl`, specified as a cell array of character vectors. `PredictorNames` has length equal to the number of predictor variables in the predictor data.

Data Types: `cell`

**Prior — Prior class probabilities**

numeric vector

This property is read-only.

Prior class probabilities, specified as a numeric vector. `Prior` has as many elements as classes in `ClassNames`, and the order of the elements corresponds to the elements of `ClassNames`.

`fitcecoc` incorporates misclassification costs differently among different types of binary learners.

Data Types: `double`

**ResponseName — Response variable name**

character array

Response variable name, specified as a character array.

Data Types: char

**ScoreTransform** — Score transformation function to apply to predicted scores

function name | function handle

Score transformation function to apply to predicted scores, specified as a function name or function handle.

For linear classification models and before transformation, the predicted classification score for the observation  $x$  (row vector) is  $f(x) = x\beta + b$ , where  $\beta$  and  $b$  correspond to `Mdl.Beta` and `Mdl.Bias`, respectively.

To change the score transformation function to, e.g., *function*, use dot notation.

- For a built-in function, enter this code and replace *function* with a character vector in the table.

```
Mdl.ScoreTransform = 'function';
```

Value	Description
'doublelogit'	$1/(1 + e^{-2x})$
'invlogit'	$\log(x / (1-x))$
'ismax'	Set the score for the class with the largest score to 1, and set the scores for all other classes to 0.
'logit'	$1/(1 + e^{-x})$
'none' or 'identity'	$x$ (no transformation)
'sign'	-1 for $x < 0$ 0 for $x = 0$ 1 for $x > 0$
'symmetric'	$2x - 1$
'symmetricismax'	Set the score for the class with the largest score to 1, and set the scores for all other classes to -1.
'symmetriclogit'	$2/(1 + e^{-x}) - 1$

- For a MATLAB function, or a function that you define, enter its function handle.

```
Mdl.ScoreTransform = @function;
```

*function* must accept a matrix of the original scores for each class, and then return a matrix of the same size representing the transformed scores for each class.

Data Types: char | function\_handle

## Methods

kfoldEdge	Classification edge for observations not used for training
kfoldLoss	Classification loss for observations not used in training
kfoldMargin	Classification margins for observations not used in training
kfoldPredict	Predict labels for observations not used for training

## Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects (MATLAB).

## Examples

### Create Cross-Validated Multiclass Linear Classification Model

Load the NLP data set.

```
load nlpdata
```

X is a sparse matrix of predictor data, and Y is a categorical vector of class labels.

Cross-validate a multiclass, linear classification model that can identify which MATLAB® toolbox a documentation web page is from based on counts of words on the page.

```
rng(1); % For reproducibility  
CVMdl = fitcecoc(X,Y,'Learners','linear','CrossVal','on')
```

```
CVMdl =
```

```
classreg.learning.partition.ClassificationPartitionedLinearECOC
  CrossValidatedModel: 'Linear'
    ResponseName: 'Y'
  NumObservations: 31572
    KFold: 10
  Partition: [1x1 cvpartition]
  ClassNames: [1x13 categorical]
  ScoreTransform: 'none'
```

`CVMD1` is a `ClassificationPartitionedLinearECOC` cross-validated model. Because `fitcecoc` implements 10-fold cross-validation by default, `CVMD1.Trained` contains a 10-by-1 cell vector of ten `CompactClassificationECOC` models that contain the results of training ECOC models composed of binary, linear classification models for each of the folds.

Estimate labels for out-of-fold observations and estimate the generalization error by passing `CVMD1` to `kfoldPredict` and `kfoldLoss`, respectively.

```
oofLabels = kfoldPredict(CVMD1);
ge = kfoldLoss(CVMD1)
```

```
ge =

    0.0958
```

The estimated generalization error is about 10% misclassified observations.

To improve generalization error, try specifying another solver, such as `LBFGS`. To change default options when training ECOC models composed of linear classification models, create a linear classification model template using `templateLinear`, and then pass the template to `fitcecoc`.

### Find Good Lasso Penalty Using Cross-Validation

To determine a good lasso-penalty strength for an ECOC model composed of linear classification models that use logistic regression learners, implement 5-fold cross-validation.

Load the NLP data set.

```
load nlpdata
```

X is a sparse matrix of predictor data, and Y is a categorical vector of class labels.

For simplicity, use the label 'others' for all observations in Y that are not 'simulink', 'dsp', or 'comm'.

```
Y(~(ismember(Y,{'simulink','dsp','comm'}))) = 'others';
```

Create a set of 11 logarithmically-spaced regularization strengths from  $10^{-7}$  through  $10^{-2}$ .

```
Lambda = logspace(-7,-2,11);
```

Create a linear classification model template that specifies to use logistic regression learners, use lasso penalties with strengths in Lambda, train using SpARSA, and lower the tolerance on the gradient of the objective function to  $1e-8$ .

```
t = templateLinear('Learner','logistic','Solver','sparsa',...
    'Regularization','lasso','Lambda',Lambda,'GradientTolerance',1e-8);
```

Cross-validate the models. To increase execution speed, transpose the predictor data and specify that the observations are in columns.

```
X = X';
rng(10); % For reproducibility
CVMdl = fitcecoc(X,Y,'Learners',t,'ObservationsIn','columns','KFold',5);
```

CVMdl is a ClassificationPartitionedLinearECOC model.

Dissect CVMdl, and each model within it.

```
numECOCModels = numel(CVMdl.Trained)
ECOCmdl1 = CVMdl.Trained{1}
numCLModels = numel(ECOCmdl1.BinaryLearners)
CLmdl1 = ECOCmdl1.BinaryLearners{1}
```

```
numECOCModels =
```

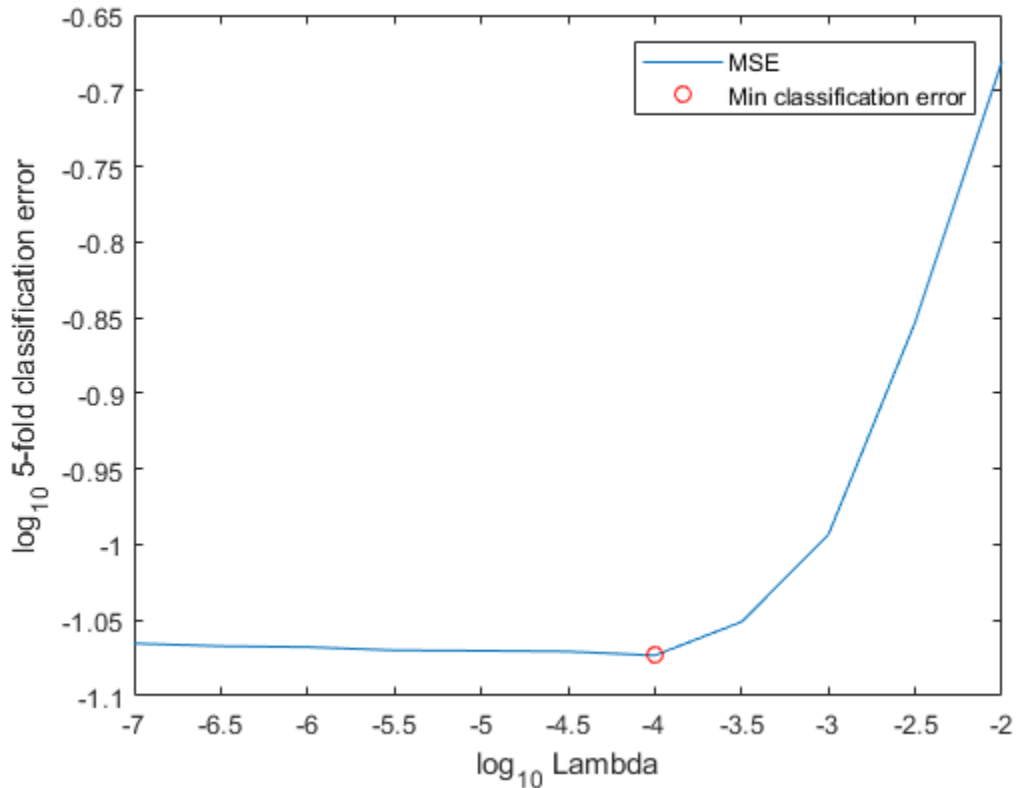
```
ECOCmdl1 =  
  
    classreg.learning.classif.CompactClassificationECOC  
        ResponseName: 'Y'  
        ClassNames: [comm    dsp    simulink    others]  
        ScoreTransform: 'none'  
        BinaryLearners: {6x1 cell}  
        CodingMatrix: [4x6 double]  
  
numCLModels =  
  
    6  
  
CLMdl1 =  
  
    ClassificationLinear  
        ResponseName: 'Y'  
        ClassNames: [-1 1]  
        ScoreTransform: 'logit'  
            Beta: [34023x11 double]  
            Bias: [1x11 double]  
            Lambda: [1x11 double]  
            Learner: 'logistic'
```

Because `fitcecoc` implements 5-fold cross-validation, `CVMD1` contains a 5-by-1 cell array of `CompactClassificationECOC` models that the software trains on each fold. The `BinaryLearners` property of each `CompactClassificationECOC` model contains the `ClassificationLinear` models. The number of `ClassificationLinear` models within each compact ECOC model depends on the number of distinct labels and coding design. Because `Lambda` is a sequence of regularization strengths, you can think of `CLMdl1` as 11 models, one for each regularization strength in `Lambda`.

Determine how well the models generalize by plotting the averages of the 5-fold classification error for each regularization strength. Identify the regularization strength that minimizes the generalization error over the grid.

```
ce = kfoldLoss(CVMD1);  
figure;
```

```
plot(log10(Lambda),log10(ce))
[~,minCEIdx] = min(ce);
minLambda = Lambda(minCEIdx);
hold on
plot(log10(minLambda),log10(ce(minCEIdx)),'ro');
ylabel('log_{10} 5-fold classification error')
xlabel('log_{10} Lambda')
legend('MSE','Min classification error')
hold off
```



Train an ECOC model composed of linear classification model using the entire data set, and specify the minimal regularization strength.

```
t = templateLinear('Learner','logistic','Solver','sparsa',...  
  'Regularization','lasso','Lambda',minLambda,'GradientTolerance',1e-8);  
MdlFinal = fitcecoc(X,Y,'Learners',t,'ObservationsIn','columns');
```

To estimate labels for new observations, pass `MdlFinal` and the new data to `predict`.

## See Also

[ClassificationECOC](#) | [ClassificationLinear](#) | [fitcecoc](#) | [fitclinear](#) |  
[kfoldLoss](#) | [kfoldPredict](#)

**Introduced in R2016a**



# ClassificationPartitionedModel class

Cross-validated classification model

## Description

`ClassificationPartitionedModel` is a set of classification models trained on cross-validated folds. Estimate the quality of classification by cross validation using one or more “kfold” methods: `kfoldPredict`, `kfoldLoss`, `kfoldMargin`, `kfoldEdge`, and `kfoldfun`.

Every “kfold” method uses models trained on in-fold observations to predict the response for out-of-fold observations. For example, suppose you cross validate using five folds. In this case, the software randomly assigns each observation into five roughly equally sized groups. The training fold contains four of the groups (i.e., roughly 4/5 of the data) and the test fold contains the other group (i.e., roughly 1/5 of the data). In this case, cross validation proceeds as follows:

- The software trains the first model (stored in `CVMD1.Trained{1}`) using the observations in the last four groups and reserves the observations in the first group for validation.
- The software trains the second model (stored in `CVMD1.Trained{2}`) using the observations in the first group and last three groups, and reserves the observations in the second group for validation.
- The software proceeds in a similar fashion for the third to fifth models.

If you validate by calling `kfoldPredict`, it computes predictions for the observations in group 1 using the first model, group 2 for the second model, and so on. In short, the software estimates a response for every observation using the model trained without that observation.

## Construction

`CVMD1 = crossval(Mdl)` creates a cross-validated classification model from a classification model (`Mdl`).

Alternatively:

- `CVDiscrMdl = fitcdiscr(X,Y,Name,Value)`
- `CVKNNMdl = fitcknn(X,Y,Name,Value)`
- `CVNBMDL = fitcnb(X,Y,Name,Value)`
- `CVSVMMDL = fitcsvm(X,Y,Name,Value)`
- `CVTreeMdl = fitctree(X,Y,Name,Value)`

create a cross-validated model when name is either 'CrossVal', 'Kfold', 'Holdout', 'Leaveout', or 'CVPartition'. For syntax details, see `fitcdiscr`, `fitcknn`, `fitcnb`, `fitcsvm`, and `fitctree`.

## Input Arguments

### **Mdl**

A classification model. `Mdl` can be any of the following:

- A classification tree trained using `fitctree`
- A discriminant analysis classifier trained using `fitcdiscr`
- A naive Bayes classifier trained using `fitcnb`
- A nearest-neighbor classifier trained using `fitcknn`
- A support vector machine classifier trained using `fitcsvm`

## Properties

### **CategoricalPredictors**

Indices of categorical predictors, stored as a vector of positive integers. `CategoricalPredictors` contains index values corresponding to the columns of the predictor data that contain categorical predictors. If none of the predictors are categorical, then this property is empty (`[]`).

If `Mdl` is a trained discriminant analysis classifier, then `CategoricalPredictors` is always empty (`[]`).

**ClassNames**

List of elements in  $Y$  with duplicates removed. `ClassNames` has the same data type as the data in the argument  $Y$ , and therefore can be a categorical or character array, logical or numeric vector, or cell array of character vectors.

**Cost**

Square matrix, where  $\text{Cost}(i, j)$  is the cost of classifying a point into class  $j$  if its true class is  $i$  (i.e., the rows correspond to the true class and the columns correspond to the predicted class). The order of the rows and columns of `Cost` corresponds to the order of the classes in `ClassNames`. The number of rows and columns in `Cost` is the number of unique classes in the response.

If `CVModel` is a cross-validated `ClassificationDiscriminant`, `ClassificationKNN`, or `ClassificationNaiveBayes` model, then you can change its cost matrix to e.g., `CostMatrix`, using dot notation.

```
CVModel.Cost = CostMatrix;
```

**CrossValidatedModel**

Name of the cross-validated model, which is a character vector.

**KFold**

Number of folds used in cross-validated model, which is a positive integer.

**ModelParameters**

Object holding parameters of `CVModel`.

**Partition**

The partition of class `CVPartition` used in creating the cross-validated model.

**PredictorNames**

Cell array of character vectors containing the predictor names, in the order that they appear in the training data.

**Prior**

Numeric vector of prior probabilities for each class. The order of the elements of `Prior` corresponds to the order of the classes in `ClassNames`.

If `CVModel` is a cross-validated `ClassificationDiscriminant` or `ClassificationNaiveBayes` model, then you can change its vector of priors to e.g., `priorVector`, using dot notation.

```
CVModel.Prior = priorVector;
```

**ResponseName**

Character vector describing the response variable  $Y$ .

**ScoreTransform**

Character vector representing a built-in transformation function, or a function handle for transforming predicted classification scores.

To change the score transformation function to, e.g., `function`, use dot notation.

- For a built-in function, enter a character vector.

```
Mdl.ScoreTransform = 'function';
```

This table contains the available, built-in functions.

Value	Description
'doublelogit'	$1/(1 + e^{-2x})$
'invlogit'	$\log(x / (1-x))$
'ismax'	Set the score for the class with the largest score to 1, and set the scores for all other classes to 0.
'logit'	$1/(1 + e^{-x})$
'none' or 'identity'	$x$ (no transformation)
'sign'	-1 for $x < 0$ 0 for $x = 0$ 1 for $x > 0$

Value	Description
'symmetric'	$2x - 1$
'symmetricismax'	Set the score for the class with the largest score to 1, and set the scores for all other classes to -1.
'symmetriclogit'	$2/(1 + e^{-x}) - 1$

- For a MATLAB function, or a function that you define, enter its function handle.

```
Mdl.ScoreTransform = @function;
```

*function* should accept a matrix (the original scores) and return a matrix of the same size (the transformed scores).

#### **Trained**

The trained learners, which is a cell array of compact classification models.

#### **W**

The scaled `weights`, which is a vector with length `n`, the number of rows in `X`.

#### **X**

Numeric matrix of predictor values. Each column of `X` represents one variable, and each row represents one observation.

#### **Y**

Categorical or character array, logical or numeric vector, or cell array of character vectors specifying the class labels for each observation. `Y` has the same number of rows as `X`, and each entry of `Y` is the response to the data in the corresponding row of `X`.

## Methods

<code>kfoldEdge</code>	Classification edge for observations not used for training
<code>kfoldfun</code>	Cross validate function
<code>kfoldLoss</code>	Classification loss for observations not used for training
<code>kfoldMargin</code>	Classification margins for observations not used for training
<code>kfoldPredict</code>	Predict response for observations not used for training

## Copy Semantics

Value. To learn how value classes affect copy operations, see [Copying Objects \(MATLAB\)](#).

## Examples

### Evaluate the Classification Error of a Classification Tree Classifier

Evaluate the  $k$ -fold cross-validation error for a classification tree model.

Load Fisher's iris data set.

```
load fisheriris
```

Train a classification tree using default options.

```
Mdl = fitctree(meas, species);
```

Cross validate the classification tree model.

```
CVMdl = crossval(Mdl);
```

Estimate the 10-fold cross-validation loss.

```
L = kfoldLoss(CVMdl)
```

```
L =
```

```
0.0533
```

### Estimate Posterior Probabilities for Test Samples

Estimate positive class posterior probabilities for the test set of an SVM algorithm.

Load the ionosphere data set.

```
load ionosphere
```

Train an SVM classifier. Specify a 20% holdout sample. It is good practice to standardize the predictors and specify the class order.

```
rng(1) % For reproducibility
CVSVMModel = fitcsvm(X,Y,'Holdout',0.2,'Standardize',true,...
    'ClassNames',{'b','g'});
```

CVSVMModel is a trained ClassificationPartitionedModel cross-validated classifier.

Estimate the optimal score function for mapping observation scores to posterior probabilities of an observation being classified as 'g'.

```
ScoreCVSVMModel = fitSVMPosterior(CVSVMModel);
```

ScoreSVMModel is a trained ClassificationPartitionedModel cross-validated classifier containing the optimal score transformation function estimated from the training data.

Estimate the out-of-sample positive class posterior probabilities. Display the results for the first 10 out-of-sample observations.

```
[~,OOSPostProbs] = kfoldPredict(ScoreCVSVMModel);
indx = ~isnan(OOSPostProbs(:,2));
hoObs = find(indx); % Holdout observation numbers
OOSPostProbs = [hoObs, OOSPostProbs(indx,2)];
table(OOSPostProbs(1:10,1),OOSPostProbs(1:10,2),...
    'VariableNames',{'ObservationIndex','PosteriorProbability'})
```

```
ans =
```

10x2 table

ObservationIndex	PosteriorProbability
6	0.17378
7	0.89637
8	0.0076609
9	0.91602
16	0.026718
22	4.6081e-06
23	0.9024
24	2.4129e-06
38	0.00042697
41	0.86427

- “Cross Validating a Discriminant Analysis Classifier” on page 20-23

## Tips

To estimate posterior probabilities of trained, cross-validated SVM classifiers, use `fitSVMPosterior`.

## See Also

`ClassificationKNN` | `ClassificationNaiveBayes` |  
`CompactClassificationDiscriminant` | `CompactClassificationSVM` |  
`CompactClassificationTree` | `fitSVMPosterior` | `fitcdiscr` | `fitcknn` |  
`fitcnb` | `fitcsvm` | `fitctree`

## Topics

“Cross Validating a Discriminant Analysis Classifier” on page 20-23



# ClassificationSVM class

## Superclasses:

Support vector machine for binary classification

## Description

`ClassificationSVM` is a support vector machine classifier on page 32-543 for one- or two-class learning. To train a `ClassificationSVM` classifier, use `fitcsvm`.

Trained `ClassificationSVM` classifiers store the training data, parameter values, prior probabilities, support vectors, and algorithmic implementation information. You can use these classifiers to:

- Estimate resubstitution predictions. For details, see `resubPredict`.
- Predict labels or posterior probabilities for new data. For details, see `predict`.

## Construction

`Mdl = fitcsvm(Tbl, ResponseVarName)` returns an SVM classifier (`Mdl`) trained using the sample data contained in the table `Tbl`. `ResponseVarName` is the name of the variable in `Tbl` that contains the class labels for one- or two-class classification. For details, see `fitcsvm`.

`Mdl = fitcsvm(Tbl, formula)` returns an SVM classifier trained using the predictor data and class labels in the table `Tbl`. `formula` is an explanatory model of the response and a subset of predictor variables in `Tbl` used for training. For details, see `fitcsvm`.

`Mdl = fitcsvm(Tbl, Y)` returns an SVM classifier trained using the predictor variables in table `Tbl` and class labels in vector `Y`. For details, see `fitcsvm`.

`Mdl = fitcsvm(X, Y)` returns an SVM classifier trained using the predictors in the matrix `X` and class labels in the vector `Y` for one- or two-class classification. For details, see `fitcsvm`.

`Mdl = fitcsvm( ____, Name, Value)` returns a trained SVM classifier with additional options specified by one or more `Name, Value` pair arguments, using any of the previous syntaxes. For example, you can specify the type of cross validation, the cost for misclassification, or the type of score transformation function. For name-value pair argument details, see `fitcsvm`.

If you set one of the following five options, then `Mdl` is a `ClassificationPartitionedModel` `model`: `'CrossVal'`, `'CVPartition'`, `'Holdout'`, `'KFold'`, or `'Leaveout'`. Otherwise, `Mdl` is a `ClassificationSVM` classifier.

## Input Arguments

### **Tbl** — Sample data

table

Sample data used to train the model, specified as a table. Each row of `Tbl` corresponds to one observation, and each column corresponds to one predictor variable. Optionally, `Tbl` can contain one additional column for the response variable. Multi-column variables and cell arrays other than cell arrays of character vectors are not allowed.

If `Tbl` contains the response variable, and you want to use all remaining variables in `Tbl` as predictors, then specify the response variable using `ResponseVarName`.

If `Tbl` contains the response variable, and you want to use only a subset of the remaining variables in `Tbl` as predictors, then specify a formula using `formula`.

If `Tbl` does not contain the response variable, then specify a response variable using `Y`. The length of response variable and the number of rows of `Tbl` must be equal.

Data Types: `table`

### **ResponseVarName** — Response variable name

name of variable in `Tbl`

Response variable name, specified as the name of a variable in `Tbl`.

You must specify `ResponseVarName` as a character vector. For example, if the response variable `Y` is stored as `Tbl.Y`, then specify it as `'Y'`. Otherwise, the software treats all columns of `Tbl`, including `Y`, as predictors when training the model.

The response variable must be a categorical or character array, logical or numeric vector, or cell array of character vectors. If  $Y$  is a character array, then each element must correspond to one row of the array.

It is good practice to specify the order of the classes using the `ClassNames` name-value pair argument.

Data Types: `char`

**formula** — Explanatory model of response and subset of predictor variables  
character vector

Explanatory model of the response and a subset of the predictor variables, specified as a character vector in the form of ' $Y \sim X_1 + X_2 + X_3$ '. In this form,  $Y$  represents the response variable, and  $X_1$ ,  $X_2$ , and  $X_3$  represent the predictor variables. The variables must be variable names in `Tbl` (`Tbl.Properties.VariableNames`).

To specify a subset of variables in `Tbl` as predictors for training the model, use a formula. If you specify a formula, then the software does not use any variables in `Tbl` that do not appear in `formula`.

Data Types: `char`

**y** — Class labels

categorical array | character array | logical vector | vector of numeric values | cell array of character vectors

Class labels to which the SVM model is trained, specified as a categorical or character array, logical or numeric vector, or cell array of character vectors.

- $Y$  must contain at most two distinct classes. For multiclass learning, see `fitcecoc`.
- If  $Y$  is a character array, then each element must correspond to one row of the array.
- The length of  $Y$  and the number of rows of `Tbl` or `X` must be equal.
- It is good practice to specify the class order using the `ClassNames` name-value pair argument.

Data Types: `char` | `cell` | `categorical` | `logical` | `single` | `double`

**x** — Predictor data

matrix of numeric values

Predictor data to which the SVM classifier is trained, specified as a matrix of numeric values.

Each row of `X` corresponds to one observation (also known as an instance or example), and each column corresponds to one predictor.

The length of `Y` and the number of rows of `X` must be equal.

To specify the names of the predictors in the order of their appearance in `X`, use the `PredictorNames` name-value pair argument.

Data Types: `double` | `single`

---

**Note** The software treats `NaN`, empty character vector (`''`), and `<undefined>` elements as missing values. If a row of `X` or an element of `Y` contains at least one `NaN`, then the software removes those rows and elements from both arguments. Such deletion decreases the effective training or cross-validation sample size.

---

## Properties

### Alpha

`s`-by-1 numeric vector of trained classifier coefficients from the dual problem, that is, the estimated Lagrange multipliers. `s` is the number of support vectors in the trained classifier, that is, `sum(Mdl.IsSupportVector)`.

If you specify removing duplicates using `RemoveDuplicates`, then for a given set of duplicate observations that are support vectors, `Alpha` contains one coefficient corresponding to the entire set. That is, MATLAB attributes a nonzero coefficient to one observation from the set of duplicates and a coefficient of 0 to all other duplicate observations in the set.

### Beta

Numeric vector of linear predictor coefficients. `Beta` has length equal to the number of predictors used to train the model.

If your predictor data contains categorical variables, then the software uses full dummy encoding for these variables. The software creates one dummy variable for each level of

each categorical variable. `Beta` stores one value for each predictor variable, including the dummy variables. For example, if there are three predictors, one of which is a categorical variable with three levels, then `Beta` is a numeric vector containing five values.

If `KernelParameters.Function` is 'linear', then the classification score for the observation  $x$  is consistent with

$$f(x) = (x / s)' \beta + b.$$

`Mdl` stores  $\beta$ ,  $b$ , and  $s$  in the properties `Beta`, `Bias`, and `KernelParameters.Scale`, respectively.

If `KernelParameters.Function` is not 'linear', then `Beta` is empty (`[]`).

### **Bias**

Scalar corresponding to the trained classifier bias term.

### **BoxConstraints**

$n$ -by-1 numeric vector of box constraints on page 32-541.  $n$  is the number of observations in the training data (see the `NumObservations` property).

If you specify removing duplicates using `RemoveDuplicates`, then for a given set of duplicate observations, MATLAB sums the box constraints, and then attributes the sum to one observation and box constraints of 0 to all other observations in the set.

### **CacheInfo**

Structure array containing:

- The cache size (in MB) that the software reserves to train the SVM classifier (`Mdl.CacheInfo.Size`). To set the cache size to `CacheSize` MB, set the `fitsvm` name-value pair argument to 'CacheSize', `CacheSize`.
- The caching algorithm that the software uses during optimization (`Mdl.CacheInfo.Algorithm`). Currently, the only available caching algorithm is `Queue`. You cannot set the caching algorithm.

### **CategoricalPredictors**

Indices of categorical predictors, stored as a vector of positive integers.

`CategoricalPredictors` contains index values corresponding to the columns of the

predictor data that contain categorical predictors. If none of the predictors are categorical, then this property is empty (`[]`).

### **ClassNames**

List of elements in `Y` with duplicates removed. `ClassNames` has the same data type as the data in the argument `Y`, and therefore can be a categorical or character array, logical or numeric vector, or cell array of character vectors.

### **ConvergenceInfo**

Structure array containing convergence information.

Field	Description
Converged	Logical flag indicating whether the algorithm converged (1 indicates convergence)
ReasonForConvergence	Character vector indicating the criterion the software uses to detect convergence
Gap	Scalar feasibility gap between the dual and primal objective functions
GapTolerance	Scalar feasibility gap tolerance. Set this tolerance to, e.g., <i>gt</i> , using the name-value pair argument 'GapTolerance', <i>gt</i> of <code>fitcsvm</code> .
DeltaGradient	Scalar-attained gradient difference between upper and lower violators
DeltaGradientTolerance	Scalar tolerance for gradient difference between upper and lower violators. Set this tolerance to, e.g., <i>dgt</i> , using the name-value pair argument 'DeltaGradientTolerance', <i>dgt</i> of <code>fitcsvm</code> .
LargestKKTViolation	Maximal, scalar Karush-Kuhn-Tucker (KKT) violation value

Field	Description
KKTTolerance	Scalar tolerance for the largest KKT violation. Set this tolerance to, e.g., <code>kkt</code> , using the name-value pair argument <code>'KKTTolerance', kkt</code> of <code>fitcsvm</code> .
History	Structure array containing convergence information at set optimization iterations. The fields are: <ul style="list-style-type: none"> <li>• <code>NumIterations</code>: numeric vector of iteration indices for which the software records convergence information</li> <li>• <code>Gap</code>: numeric vector of Gap values at the iterations</li> <li>• <code>DeltaGradient</code>: numeric vector of DeltaGradient values at the iterations</li> <li>• <code>LargestKKTViolation</code>: numeric vector of LargestKKTViolation values at the iterations</li> <li>• <code>NumSupportVectors</code>: numeric vector indicating the number of support vectors at the iterations</li> <li>• <code>Objective</code>: numeric vector of Objective values at the iterations</li> </ul>
Objective	Scalar value of the dual objective function

### Cost

Square matrix, where  $\text{Cost}(i, j)$  is the cost of classifying a point into class  $j$  if its true class is  $i$ .

During training, the software updates the prior probabilities by incorporating the penalties described in the cost matrix. Therefore:

- For two-class learning,  $\text{Cost}$  always has this form:  $\text{Cost}(i, j) = 1$  if  $i \neq j$ , and  $\text{Cost}(i, j) = 0$  if  $i = j$  (i.e., the rows correspond to the true class and the columns

correspond to the predicted class). The order of the rows and columns of `Cost` corresponds to the order of the classes in `ClassNames`.

- For one-class learning, `Cost = 0`.

This property is read-only. For more details, see Algorithms on page 32-545.

### **ExpandedPredictorNames**

Expanded predictor names, stored as a cell array of character vectors.

If the model uses encoding for categorical variables, then `ExpandedPredictorNames` includes the names that describe the expanded variables. Otherwise, `ExpandedPredictorNames` is the same as `PredictorNames`.

### **Gradient**

Numeric vector of training data gradient values. `Gradient` has length equal to the number of observations (i.e., `size(Mdl.X,1)`).

### **HyperparameterOptimizationResults**

Description of the cross-validation optimization of hyperparameters, stored as a `BayesianOptimization` object or a table of hyperparameters and associated values. Nonempty when the `OptimizeHyperparameters` name-value pair is nonempty at creation. Value depends on the setting of the `HyperparameterOptimizationOptions` name-value pair at creation:

- 'bayesopt' (default) — Object of class `BayesianOptimization`
- 'gridsearch' or 'randomsearch' — Table of hyperparameters used, observed objective function values (cross-validation loss), and rank of observations from lowest (best) to highest (worst)

### **IsSupportVector**

$n$ -by-1 logical vector indicating whether a corresponding row in the predictor data matrix is a support vector.  $n$  is the number of observations in the training data (see `NumObservations`).

If you specify removing duplicates using `RemoveDuplicates`, then for a given set of duplicate observations that are support vectors, `IsSupportVector` flags only one as a support vector.



**KernelParameters**

Structure array containing the kernel name and parameter values.

To display the values of `KernelParameters`, use dot notation, for example, `Mdl.KernelParameters.Scale` displays the scale parameter value.

The software accepts `KernelParameters` as inputs, and does not modify them. Alter `KernelParameters` by setting the appropriate name-value pair arguments when you train the SVM classifier using `fitcsvm`.

**ModelParameters**

Object containing parameter values, e.g., the name-value pair argument values, used to train the SVM classifier. `ModelParameters` does not contain estimated parameters.

Access fields of `ModelParameters` using dot notation. For example, access the initial values for estimating `Alpha` using `Mdl.ModelParameters.Alpha`.

**Mu**

Numeric vector of predictor means.

If you specify `'Standardize',1` or `'Standardize',true` when you train an SVM classifier using `fitcsvm`, then `Mu` has length equal to the number of predictors.

If your predictor data contains categorical variables, then the software uses full dummy encoding for these variables. The software creates one dummy variable for each level of each categorical variable. `Mu` stores one value for each predictor variable, including the dummy variables. However, the software does not standardize the columns that contain categorical variables.

If `'Standardize'` is `false` or `0`, then `Mu` is an empty vector (`[]`).

**NumIterations**

Positive integer indicating the number of iterations required by the optimization routine to attain convergence.

To set a limit on the number of iterations to, e.g.,  $k$ , specify the name-value pair argument `'IterationLimit',k` of `fitcsvm`.

**Nu**

Positive scalar representing the  $\nu$  parameter for one-class learning on page 32-542.

**NumObservations**

Numeric scalar representing the number of observations in the training data. If the input arguments `X` or `Y` contain missing values, then `NumObservations` is less than the length of `Y`.

**OutlierFraction**

Scalar indicating the expected proportion of outliers in the training data.

**PredictorNames**

Cell array of character vectors containing the predictor names, in the order that they appear in the training data.

**Prior**

Numeric vector of prior probabilities for each class. The order of the elements of `Prior` corresponds to the elements of `Mdl.ClassNames`.

For two-class learning, if you specify a cost matrix, then the software updates the prior probabilities by incorporating the penalties described in the cost matrix.

This property is read-only. For more details, see Algorithms on page 32-545.

**ResponseName**

Character vector describing the response variable `Y`.

**ScoreTransform**

Character vector representing a built-in transformation function, or a function handle for transforming predicted classification scores.

To change the score transformation function to, e.g., `function`, use dot notation.

- For a built-in function, enter a character vector.

```
Mdl.ScoreTransform = 'function';
```

This table contains the available, built-in functions.

Value	Description
'doublelogit'	$1/(1 + e^{-2x})$
'invlogit'	$\log(x / (1-x))$
'ismax'	Set the score for the class with the largest score to 1, and set the scores for all other classes to 0.
'logit'	$1/(1 + e^{-x})$
'none' or 'identity'	$x$ (no transformation)
'sign'	-1 for $x < 0$ 0 for $x = 0$ 1 for $x > 0$
'symmetric'	$2x - 1$
'symmetricismax'	Set the score for the class with the largest score to 1, and set the scores for all other classes to -1.
'symmetriclogit'	$2/(1 + e^{-x}) - 1$

- For a MATLAB function, or a function that you define, enter its function handle.

```
Mdl.ScoreTransform = @function;
```

*function* should accept a matrix (the original scores) and return a matrix of the same size (the transformed scores).

### ShrinkagePeriod

Nonnegative integer indicating the shrinkage period, i.e., number of iterations between reductions of the active set.

To set the shrinkage period to, e.g., *sp*, specify the name-value pair argument 'ShrinkagePeriod', *sp* of fitcsvm.

### Sigma

Numeric vector of predictor standard deviations.

If you specify 'Standardize', 1 or 'Standardize', true when you train the SVM classifier, then Sigma has length equal to the number of predictors.

If your predictor data contains categorical variables, then the software uses full dummy encoding for these variables. The software creates one dummy variable for each level of each categorical variable. `Sigma` stores one value for each predictor variable, including the dummy variables. However, the software does not standardize the columns that contain categorical variables.

If `'Standardize'` is `false` or `0`, then `Sigma` is an empty vector (`[]`).

### **Solver**

Character vector indicating the solving routine that the software used to train the SVM classifier.

To set the solver to, e.g., `solver`, specify the name-value pair argument `'Solver', solver` of `fitcsvm`.

### **SupportVectors**

$s$ -by- $p$  numeric matrix containing rows of  $X$  that MATLAB considers the support vectors.  $s$  is the number of support vectors in the trained classifier, that is, `sum(Mdl.IsSupportVector)` and  $p$  is the number of predictor variables in the predictor data.

If you specify `'Standardize', 1` or `'Standardize', true`, then `SupportVectors` are the standardized rows of  $X$ .

If you specify removing duplicates using `RemoveDuplicates`, then for a given set of duplicate observations that are support vectors, `SupportVectors` contains one unique support vector.

### **SupportVectorLabels**

$s$ -by-1 numeric vector of support vector class labels.  $s$  is the number of support vectors in the trained classifier, that is, `sum(Mdl.IsSupportVector)`.

A value of `+1` indicates that the corresponding support vector is in the positive class (`Mdl.ClassNames{2}`). A value of `-1` indicates that the corresponding support vector is in the negative class (`Mdl.ClassNames{1}`).

If you specify removing duplicates using `RemoveDuplicates`, then for a given set of duplicate observations that are support vectors, `SupportVectorLabels` contains one unique support vector label.

**W**

Numeric vector of observation weights that the software used to train the SVM classifier.

The length of `W` is `Mdl.NumObservations`.

`fitcsvm` normalizes `Weights` so that the elements of `W` within a particular class sum up to the prior probability of that class.

**X**

Numeric matrix of unstandardized predictor values that the software used to train the SVM classifier.

Each row of `X` corresponds to one observation, and each column corresponds to one variable.

The software excludes predictor data rows removed due to NaNs from `X`.

**Y**

Categorical or character array, logical or numeric vector, or cell array of character vectors representing the observed class labels used to train the SVM classifier. `Y` is the same data type as the input argument `Y` of `fitcsvm`.

Each row of `Y` represents the observed classification of the corresponding row of `X`.

The software excludes elements removed due to NaNs from `Y`.

## Methods

<code>compact</code>	Compact support vector machine classifier
<code>crossval</code>	Cross-validated support vector machine classifier
<code>fitPosterior</code>	Fit posterior probabilities
<code>resubEdge</code>	Classification edge for support vector machine classifiers by resubstitution
<code>resubLoss</code>	Classification loss for support vector machine classifiers by resubstitution
<code>resubMargin</code>	Classification margins for support vector machine classifiers by resubstitution
<code>resubPredict</code>	Predict support vector machine classifier resubstitution responses
<code>resume</code>	Resume training support vector machine classifier

## Inherited Methods

<code>compareHoldout</code>	Compare accuracies of two classification models using new data
<code>discardSupportVectors</code>	Discard support vectors for linear support vector machine models
<code>edge</code>	Classification edge for support vector machine classifiers
<code>fitPosterior</code>	Fit posterior probabilities
<code>loss</code>	Classification error for support vector machine classifiers
<code>margin</code>	Classification margins for support vector machine classifiers
<code>predict</code>	Predict labels using support vector machine classification model

## Copy Semantics

Value. To learn how value classes affect copy operations, see [Copying Objects \(MATLAB\)](#).

## Examples

## Train a Support Vector Machine Classifier

Load Fisher's iris data set. Remove the sepal lengths and widths, and all observed setosa irises.

```
load fisheriris
inds = ~strcmp(species, 'setosa');
X = meas(inds, 3:4);
y = species(inds);
```

Train an SVM classifier using the processed data set.

```
SVModel = fitcsvm(X, y)
```

```
SVModel =
```

```
ClassificationSVM
    ResponseName: 'Y'
  CategoricalPredictors: []
        ClassNames: {'versicolor' 'virginica'}
    ScoreTransform: 'none'
  NumObservations: 100
           Alpha: [24x1 double]
           Bias: -14.4149
  KernelParameters: [1x1 struct]
    BoxConstraints: [100x1 double]
  ConvergenceInfo: [1x1 struct]
  IsSupportVector: [100x1 logical]
           Solver: 'SMO'
```

The Command Window shows that `SVModel` is a trained `ClassificationSVM` classifier and a property list. Display the properties of `SVModel`, for example, to determine the class order, by using dot notation.

```
classOrder = SVModel.ClassNames
```

```
classOrder =
```

```
2x1 cell array
    {'versicolor'}
```

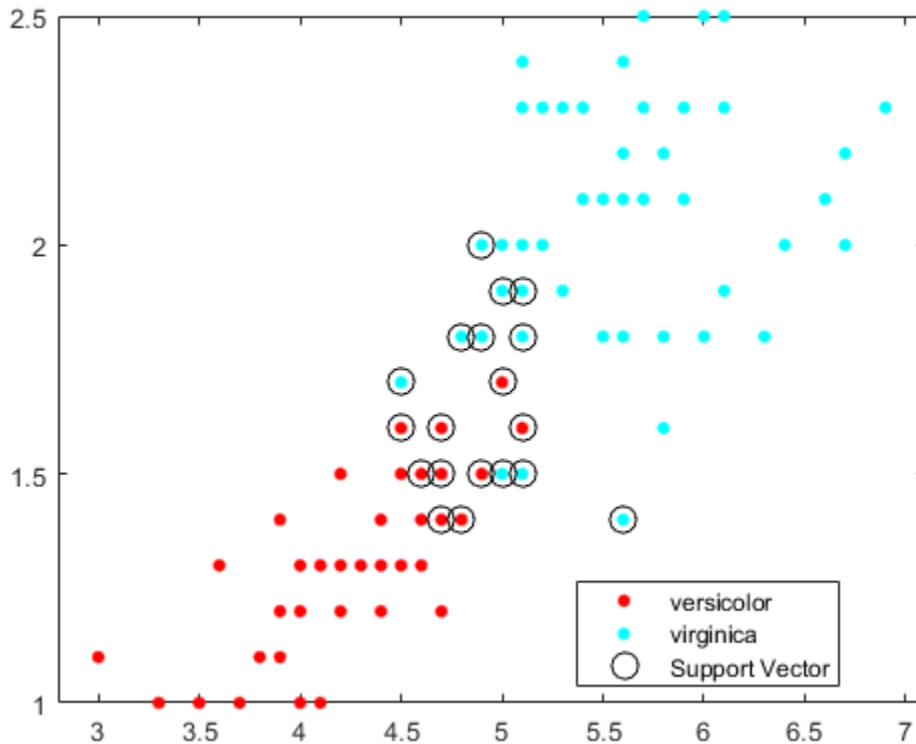
```
{'virginica' }
```

The first class ('versicolor') is the negative class, and the second ('virginica') is the positive class. You can change the class order during training by using the 'ClassNames' name-value pair argument.

Plot a scatter diagram of the data and circle the support vectors.

```
sv = SVMModel.SupportVectors;  
figure  
gscatter(X(:,1),X(:,2),y)  
hold on  
plot(sv(:,1),sv(:,2),'ko','MarkerSize',10)  
legend('versicolor','virginica','Support Vector')  
hold off
```





The support vectors are observations that occur on or beyond their estimated class boundaries.

You can adjust the boundaries (and therefore the number of support vectors) by setting a box constraint during training using the 'BoxConstraint' name-value pair argument.

### Train and Cross-Validate Support Vector Machine Classifiers

Load the `ionosphere` data set.

```
load ionosphere
```

Train and cross-validate an SVM classifier. It is good practice to standardize the predictors and specify the order of the classes.

```
rng(1); % For reproducibility
CVSVMModel = fitcsvm(X,Y,'Standardize',true,...
    'ClassNames',{'b','g'},'CrossVal','on')

CVSVMModel =

classreg.learning.partition.ClassificationPartitionedModel
  CrossValidatedModel: 'SVM'
  PredictorNames: {1x34 cell}
  ResponseName: 'Y'
  NumObservations: 351
  KFold: 10
  Partition: [1x1 cvpartition]
  ClassNames: {'b' 'g'}
  ScoreTransform: 'none'
```

CVSVMModel is not a ClassificationSVM classifier, but a ClassificationPartitionedModel cross-validated, SVM classifier. By default, the software implements 10-fold cross-validation.

Alternatively, you can cross-validate a trained ClassificationSVM classifier by passing it to crossval.

Inspect one of the trained folds using dot notation.

```
CVSVMModel.Trained{1}

ans =

classreg.learning.classif.CompactClassificationSVM
  ResponseName: 'Y'
  CategoricalPredictors: []
  ClassNames: {'b' 'g'}
  ScoreTransform: 'none'
  Alpha: [78x1 double]
  Bias: -0.2209
  KernelParameters: [1x1 struct]
```

```

                Mu: [1x34 double]
                Sigma: [1x34 double]
                SupportVectors: [78x34 double]
SupportVectorLabels: [78x1 double]

```

Each fold is a `CompactClassificationSVM` classifier trained on 90% of the data.

Estimate the generalization error.

```
genError = kfoldLoss(CVSVMModel)
```

```
genError =
    0.1168
```

On average, the generalization error is approximately 12%.

- Using Support Vector Machines on page 18-192

## Definitions

### Box Constraint

A parameter that controls the maximum penalty imposed on margin-violating observations, and aids in preventing overfitting (regularization).

If you increase the box constraint, then the SVM classifier assigns fewer support vectors. However, increasing the box constraint can lead to longer training times.

### Gram Matrix

The Gram matrix of a set of  $n$  vectors  $\{x_1, \dots, x_n; x_j \in R^p\}$  is an  $n$ -by- $n$  matrix with element  $(j, k)$  defined as  $G(x_j, x_k) = \langle \phi(x_j), \phi(x_k) \rangle$ , an inner product of the transformed predictors using the kernel function  $\phi$ .

For nonlinear SVM, the algorithm forms a Gram matrix using the predictor matrix columns. The dual formalization replaces the inner product of the predictors with

corresponding elements of the resulting Gram matrix (called the “kernel trick”). Subsequently, nonlinear SVM operates in the transformed predictor space to find a separating hyperplane.

## Karush-Kuhn-Tucker Complementarity Conditions

KKT complementarity conditions are optimization constraints required for optimal nonlinear programming solutions.

In SVM, the KKT complementarity conditions are

$$\begin{cases} \alpha_j [y_j f(x_j) - 1 + \xi_j] = 0 \\ \xi_j (C - \alpha_j) = 0 \end{cases}$$

for all  $j = 1, \dots, n$ , where  $f(x_j) = \phi(x_j)' \beta + b$ ,  $\phi$  is a kernel function (see Gram matrix on page 32-2127), and  $\xi_j$  is a slack variable. If the classes are perfectly separable, then  $\xi_j = 0$  for all  $j = 1, \dots, n$ .

## One-Class Learning

One-class learning, or unsupervised SVM, aims at separating data from the origin in the high-dimensional, predictor space (not the original predictor space), and is an algorithm used for outlier detection.

The algorithm resembles that of SVM for binary classification on page 32-2129. The objective is to minimize dual expression

$$0.5 \sum_{jk} \alpha_j \alpha_k G(x_j, x_k)$$

with respect to  $\alpha_1, \dots, \alpha_n$ , subject to

$$\sum \alpha_j = nv$$

and  $0 \leq \alpha_j \leq 1$  for all  $j = 1, \dots, n$ .  $G(x_j, x_k)$  is element  $(j, k)$  of the Gram matrix on page 32-2127.

A small value of  $v$  leads to fewer support vectors, and, therefore, a smooth, crude decision boundary. A large value of  $v$  leads to more support vectors, and therefore, a curvy,

flexible decision boundary. The optimal value of  $\nu$  should be large enough to capture the data complexity and small enough to avoid overtraining. Also,  $0 < \nu \leq 1$ .

For more details, see [5].

## Support Vector

Support vectors are observations corresponding to strictly positive estimates of  $\alpha_1, \dots, \alpha_n$ .

SVM classifiers that yield fewer support vectors for a given training set are more desirable.

## Support Vector Machines for Binary Classification

The SVM binary classification algorithm searches for an optimal hyperplane that separates the data into two classes. For separable classes, the optimal hyperplane maximizes a margin (space that does not contain any observations) surrounding itself, which creates boundaries for the positive and negative classes. For inseparable classes, the objective is the same, but the algorithm imposes a penalty on the length of the margin for every observation that is on the wrong side of its class boundary.

The linear SVM score function is

$$f(x) = x'\beta + b,$$

where:

- $x$  is an observation (corresponding to a row of  $X$ ).
- The vector  $\beta$  contains the coefficients that define an orthogonal vector to the hyperplane (corresponding to `Mdl.Beta`). For separable data, the optimal margin length is  $2 / \|\beta\|$ .
- $b$  is the bias term (corresponding to `Mdl.Bias`).

The root of  $f(x)$  for particular coefficients defines a hyperplane. For a particular hyperplane,  $f(z)$  is the distance from point  $z$  to the hyperplane.

The algorithm searches for the maximum margin length, while keeping observations in the positive ( $y = 1$ ) and negative ( $y = -1$ ) classes separate. Therefore:

- For separable classes, the objective is to minimize  $\|\beta\|$  with respect to the  $\beta$  and  $b$  subject to  $y_j f(x_j) \geq 1$ , for all  $j = 1, \dots, n$ . This is the primal formalization for separable classes.
- For inseparable classes, the algorithm uses slack variables ( $\xi_j$ ) to penalize the objective function for observations that cross the margin boundary for their class.  $\xi_j = 0$  for observations that do not cross the margin boundary for their class, otherwise  $\xi_j \geq 0$ .

The objective is to minimize  $0.5\|\beta\|^2 + C\sum\xi_j$  with respect to the  $\beta$ ,  $b$ , and  $\xi_j$  subject to  $y_j f(x_j) \geq 1 - \xi_j$  and  $\xi_j \geq 0$  for all  $j = 1, \dots, n$ , and for a positive scalar box constraint on page 32-2127 *C*. This is the primal formalization for inseparable classes.

The algorithm uses the Lagrange multipliers method to optimize the objective. This introduces  $n$  coefficients  $\alpha_1, \dots, \alpha_n$  (corresponding to `Mdl.Alpha`). The dual formalizations for linear SVM are:

- For separable classes, minimize

$$0.5 \sum_{j=1}^n \sum_{k=1}^n \alpha_j \alpha_k y_j y_k x_j' x_k - \sum_{j=1}^n \alpha_j$$

with respect to  $\alpha_1, \dots, \alpha_n$ , subject to  $\sum \alpha_j y_j = 0$ ,  $\alpha_j \geq 0$  for all  $j = 1, \dots, n$ , and Karush-Kuhn-Tucker (KKT) complementarity conditions on page 32-2128.

- For inseparable classes, the objective is the same as for separable classes, except for the additional condition  $0 \leq \alpha_j \leq C$  for all  $j = 1, \dots, n$ .

The resulting score function is

$$\hat{f}(x) = \sum_{j=1}^n \hat{\alpha}_j y_j x' x_j + \hat{b}.$$

$\hat{b}$  is the estimate of the bias and  $\hat{\alpha}_j$  is the  $j$ th estimate of the vector  $\hat{\alpha}$ ,  $j = 1, \dots, n$ . Written this way, the score function is free of the estimate of  $\beta$  as a result of the primal formalization.

The SVM algorithm classifies a new observation,  $z$  using  $\text{sign}(\hat{f}(z))$ .

In some cases, there is a nonlinear boundary separating the classes. Nonlinear SVM works in a transformed predictor space to find an optimal, separating hyperplane.

The dual formalization for nonlinear SVM is

$$0.5 \sum_{j=1}^n \sum_{k=1}^n \alpha_j \alpha_k y_j y_k G(x_j, x_k) - \sum_{j=1}^n \alpha_j$$

with respect to  $\alpha_1, \dots, \alpha_n$ , subject to  $\sum \alpha_j y_j = 0$ ,  $0 \leq \alpha_j \leq C$  for all  $j = 1, \dots, n$ , and the KKT complementarity conditions.  $G(x_k, x_j)$  are elements of the Gram matrix on page 32-2127. The resulting score function is

$$\hat{f}(x) = \sum_{j=1}^n \hat{\alpha}_j y_j G(x, x_j) + \hat{b}.$$

For more details, see Understanding Support Vector Machines on page 18-186, [1], and [3].

## Algorithms

- For the mathematical formulation of the SVM binary classification algorithm, see “Support Vector Machines for Binary Classification” on page 32-2129 and “Understanding Support Vector Machines” on page 18-186.
- NaN, <undefined>, and empty character vector ( ' ' ) values indicate missing values. `fitsvm` removes entire rows of data corresponding to a missing response. When computing total weights (see the next bullets), `fitsvm` ignores any weight corresponding to an observation with at least one missing predictor. This action can lead to unbalanced prior probabilities in balanced-class problems. Consequently, observation box constraints might not equal `BoxConstraint`.
- `fitsvm` removes observations that have zero weight or prior probability.
- For two-class learning, if you specify the cost matrix `C` (see `Cost`), then the software updates the class prior probabilities  $p$  (see `Prior`) to  $p_c$  by incorporating the penalties described in `C`.

Specifically, `fitsvm`:

- 1 Computes  $p_c^* = p'c$ .
- 2 Normalizes  $p_c^*$  so that the updated prior probabilities sum 1:

$$p_c = \frac{1}{\sum_{j=1}^K p_{c,j}^*} p_c^*.$$

$K$  is the number of classes.

- 3 Resets the cost matrix to the default:

$$C = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}.$$

- 4 Removes observations from the training data corresponding to classes with zero prior probability.
- For two-class learning, `fitcsvm` normalizes all observation weights (see `Weights`) to sum to 1. Then, renormalizes the normalized weights to sum up to the updated, prior probability of the class to which the observation belongs. That is, the total weight for observation  $j$  in class  $k$  is

$$w_j^* = \frac{w_j}{\sum_{\forall j \in \text{Class } k} w_j} p_{c,k}.$$

$w_j$  is the normalized weight for observation  $j$ ;  $p_{c,k}$  is the updated prior probability of class  $k$  (see previous bullet).

- For two-class learning, `fitcsvm` assigns a box constraint to each observation in the training data. The formula for the box constraint of observation  $j$  is

$$C_j = nC_0w_j^*.$$

$n$  is the training sample size,  $C_0$  is the initial box constraint (see `BoxConstraint`),

and  $w_j^*$  is the total weight of observation  $j$  (see previous bullet).

- If you set `'Standardize', true` and any of `'Cost'`, `'Prior'`, or `'Weights'`, then `fitcsvm` standardizes the predictors using their corresponding weighted means and weighted standard deviations. That is, `fitcsvm` standardizes predictor  $j$  ( $x_j$ ) using

$$x_j^* = \frac{x_j - \mu_j^*}{\sigma_j^*}.$$



•

$$\mu_j^* = \frac{1}{\sum_k w_k^*} \sum_k w_k^* x_{jk}.$$

•  $x_{jk}$  is observation  $k$  (row) of predictor  $j$  (column).

•

$$(\sigma_j^*)^2 = \frac{v_1}{v_1^2 - v_2} \sum_k w_k^* (x_{jk} - \mu_j^*)^2.$$

•

$$v_1 = \sum_j w_j^*.$$

•

$$v_2 = \sum_j (w_j^*)^2.$$

- Let `p` be the proportion of outliers that you expect in the training data. If you set `'OutlierFraction', p`, then:
  - For one-class learning, the software trains the bias term such that 100p% of the observations in the training data have negative scores.
  - The software implements robust learning for two-class learning. In other words, the software attempts to remove 100p% of the observations when the optimization algorithm converges. The removed observations correspond to gradients that are large in magnitude.
- If your predictor data contains categorical variables, then the software generally uses full dummy encoding for these variables. The software creates one dummy variable for each level of each categorical variable.
  - The `PredictorNames` property stores one element for each of the original predictor variable names. For example, assume that there are three predictors, one of which is a categorical variable with three levels. Then `PredictorNames` is a 1-by-3 cell array of character vectors containing the original names of the predictor variables.
  - The `ExpandedPredictorNames` property stores one element for each of the predictor variables, including the dummy variables. For example, assume that there are three predictors, one of which is a categorical variable with three levels. Then `ExpandedPredictorNames` is a 1-by-5 cell array of character vectors containing the names of the predictor variables and the new dummy variables.

- Similarly, the `Beta` property stores one beta coefficient for each predictor, including the dummy variables.
- The `SupportVectors` property stores the predictor values for the support vectors, including the dummy variables. For example, assume that there are  $m$  support vectors and three predictors, one of which is a categorical variable with three levels. Then `SupportVectors` is an  $n$ -by-5 matrix.
- The `X` property stores the training data as originally input. It does not include the dummy variables. When the input is a table, `X` contains only the columns used as predictors.
- For predictors specified in a table, if any of the variables contain ordered (ordinal) categories, the software uses ordinal encoding for these variables.
  - For a variable having  $k$  ordered levels, the software creates  $k - 1$  dummy variables. The  $j$ th dummy variable is -1 for levels up to  $j$ , and +1 for levels  $j + 1$  through  $k$ .
  - The names of the dummy variables stored in the `ExpandedPredictorNames` property indicate the first level with the value +1. The software stores  $k - 1$  additional predictor names for the dummy variables, including the names of levels 2, 3, ...,  $k$ .
- All solvers implement  $L1$  soft-margin minimization.
- `fitcsvm` and `svmtrain` use, among other algorithms, SMO for optimization. The software implements SMO differently between the two functions, but numerical studies show that there is sensible agreement in the results.
- For one-class learning, the software estimates the Lagrange multipliers,  $\alpha_1, \dots, \alpha_n$ , such that

$$\sum_{j=1}^n \alpha_j = nv.$$

## References

- [1] Hastie, T., R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*, Second Edition. NY: Springer, 2008.
- [2] Scholkopf, B., J. C. Platt, J. C. Shawe-Taylor, A. J. Smola, and R. C. Williamson. "Estimating the Support of a High-Dimensional Distribution." *Neural Comput.*, Vol. 13, Number 7, 2001, pp. 1443–1471.

- [3] Christianini, N., and J. C. Shawe-Taylor. *An Introduction to Support Vector Machines and Other Kernel-Based Learning Methods*. Cambridge, UK: Cambridge University Press, 2000.
- [4] Scholkopf, B. and A. Smola. *Learning with Kernels: Support Vector Machines, Regularization, Optimization and Beyond, Adaptive Computation and Machine Learning* Cambridge, MA: The MIT Press, 2002.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Only the `predict` function supports code generation.
- When you train an SVM model by using `fitcsvm`, the following restrictions apply.
  - The class labels input argument value ( $Y$ ) cannot be a categorical array.
  - Code generation does not support categorical predictors. You cannot supply training data in a table that contains at least one categorical predictor, and you cannot use the 'CategoricalPredictors' name-value pair argument. To dummy-code variables that you want treated as categorical, preprocess the categorical data by using `dummyvar` before fitting the model.
  - The value of the 'ClassNames' name-value pair argument cannot be a categorical array.
  - The value of the 'ScoreTransform' name-value pair argument cannot be an anonymous function. For generating code that predicts posterior probabilities given new observations, pass a trained SVM model to `fitPosterior` or `fitSVMPosterior`. The `ScoreTransform` property of the returned model contains an anonymous function that represents the score-to-posterior-probability function and is configured for code generation.

## See Also

`ClassificationPartitionedModel` | `CompactClassificationSVM` | `fitsvm`

## Topics

Using Support Vector Machines on page 18-192

Understanding Support Vector Machines on page 18-186

“Code Generation Support, Usage Notes, and Limitations” on page 31-24

**Introduced in R2014a**

# ClassificationTree class

## Superclasses:

Binary decision tree for classification

## Description

A `ClassificationTree` object represents a decision tree with binary splits for classification. An object of this class can predict responses for new data using the `predict` method. The object contains the data used for training, so it can also compute resubstitution predictions.

## Construction

`tree = fitctree(TBL, ResponseVarName)` returns a fitted binary classification decision tree based on the input variables (also known as predictors, features, or attributes) contained in the table `TBL` and output (response or labels) contained in `ResponseVarName`. The returned binary tree splits branching nodes based on the values of a column of `TBL`.

`tree = fitctree(TBL, formula)` returns a fitted binary classification decision tree based on the input variables contained in the table `TBL`. `formula` is a formula that identifies the response and predictor variables in `TBL` used to fit `tree`. The returned binary tree splits branching nodes based on the values of a column of `TBL`.

`tree = fitctree(TBL, Y)` returns a fitted binary classification decision tree based on the input variables contained in the table `TBL` and output in vector `Y`. The returned binary tree splits branching nodes based on the values of a column of `TBL`.

`tree = fitctree(X, Y)` returns a fitted binary classification decision tree based on the input variables contained in matrix `X` and output `Y`. The returned binary tree splits branching nodes based on the values of a column of `X`.

`tree = fitctree(____, Name, Value)` fits a tree with additional options specified by one or more name-value pair arguments, using any of the previous syntaxes. For

example, you can specify the algorithm used to find the best split on a categorical predictor or grow a cross-validated tree.

## Input Arguments

### **Tbl** — Sample data

table

Sample data used to train the model, specified as a table. Each row of `Tbl` corresponds to one observation, and each column corresponds to one predictor variable. Optionally, `Tbl` can contain one additional column for the response variable. Multi-column variables and cell arrays other than cell arrays of character vectors are not allowed.

If `Tbl` contains the response variable, and you want to use all remaining variables in `Tbl` as predictors, then specify the response variable using `ResponseVarName`.

If `Tbl` contains the response variable, and you want to use only a subset of the remaining variables in `Tbl` as predictors, then specify a formula using `formula`.

If `Tbl` does not contain the response variable, then specify a response variable using `Y`. The length of response variable and the number of rows of `Tbl` must be equal.

Data Types: `table`

### **x** — Predictor data

numeric matrix

Predictor data, specified as a numeric matrix. Each row of `X` corresponds to one observation, and each column corresponds to one predictor variable.

`fitctree` considers NaN values in `X` as missing values. `fitctree` does not use observations with all missing values for `X` in the fit. `fitctree` uses observations with some missing values for `X` to find splits on variables for which these observations have valid values.

Data Types: `single` | `double`

### **ResponseVarName** — Response variable name

name of variable in `Tbl`

Response variable name, specified as the name of a variable in `Tbl`.

You must specify `ResponseVarName` as a character vector. For example, if the response variable `Y` is stored as `Tbl.Y`, then specify it as `'Y'`. Otherwise, the software treats all columns of `Tbl`, including `Y`, as predictors when training the model.

The response variable must be a categorical or character array, logical or numeric vector, or cell array of character vectors. If `Y` is a character array, then each element must correspond to one row of the array.

It is good practice to specify the order of the classes using the `ClassNames` name-value pair argument.

Data Types: `char`

### **formula** — Response and predictor variables to use in model training

character vector

Response and predictor variables to use in model training, specified as a character vector in the form of `'Y~X1+X2+X3'`. In this form, `Y` represents the response variable, and `X1`, `X2`, and `X3` represent the predictor variables.

To specify a subset of variables in `Tbl` as predictors for training the model, use a formula. If you specify a formula, then any variables in `Tbl` that do not appear in `formula` are not used to train the model.

Data Types: `char`

### **Y** — Class labels

numeric vector | categorical vector | logical vector | character array | cell array of character vectors

Class labels, specified as a numeric vector, categorical vector, logical vector, character array, or cell array of character vectors. Each row of `Y` represents the classification of the corresponding row of `X`.

When fitting the tree, `fitctree` considers `NaN`, `'` (empty character vector), and `<undefined>` values in `Y` to be missing values. `fitctree` does not use observations with missing values for `Y` in the fit.

For numeric `Y`, consider fitting a regression tree using `fitrtree` instead.

Data Types: `single` | `double` | `char` | `logical` | `cell`

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

### **AlgorithmForCategorical** — Algorithm for best categorical predictor split

'Exact' | 'PullLeft' | 'PCA' | 'OVBbyClass'

Algorithm to find the best split on a categorical predictor with  $C$  categories for data and  $K \geq 3$  classes, specified as the comma-separated pair consisting of

'AlgorithmForCategorical' and one of the following values.

Value	Description
'Exact'	Consider all $2^{C-1} - 1$ combinations.
'PullLeft'	Start with all $C$ categories on the right branch. Consider moving each category to the left branch as it achieves the minimum impurity for the $K$ classes among the remaining categories. From this sequence, choose the split that has the lowest impurity.
'PCA'	Compute a score for each category using the inner product between the first principal component of a weighted covariance matrix (of the centered class probability matrix) and the vector of class probabilities for that category. Sort the scores in ascending order, and consider all $C - 1$ splits.
'OVBbyClass'	Start with all $C$ categories on the right branch. For each class, order the categories based on their probability for that class. For the first class, consider moving each category to the left branch in order, recording the impurity criterion at each move. Repeat for the remaining classes. From this sequence, choose the split that has the minimum impurity.



`fitctree` automatically selects the optimal subset of algorithms for each split using the known number of classes and levels of a categorical predictor. For  $K = 2$  classes, `fitctree` always performs the exact search. To specify a particular algorithm, use the `'AlgorithmForCategorical'` name-value pair argument.

Example: `'AlgorithmForCategorical','PCA'`

### **CategoricalPredictors** — Categorical predictors list

numeric or logical vector | cell array of character vectors | character matrix | `'all'`

Categorical predictors list, specified as the comma-separated pair consisting of `'CategoricalPredictors'` and one of the following:

- A numeric vector with indices from 1 through  $p$ , where  $p$  is the number of columns of  $X$ .
- A logical vector of length  $p$ , where a `true` entry means that the corresponding column of  $X$  is a categorical variable.
- A cell array of character vectors, where each element in the array is the name of a predictor variable. The names must match entries in `PredictorNames` values.
- A character matrix, where each row of the matrix is a name of a predictor variable. The names must match entries in `PredictorNames` values. Pad the names with extra blanks so each row of the character matrix has the same length.
- `'all'`, meaning all predictors are categorical.

By default, if the predictor data is in a matrix ( $X$ ), the software assumes that none of the predictors are categorical. If the predictor data is in a table (`Tbl`), the software assumes that a variable is categorical if it contains, logical values, values of the unordered data type `categorical`, or a cell array of character vectors.

Example: `'CategoricalPredictors','all'`

Data Types: `single` | `double` | `char` | `logical` | `cell`

### **ClassNames** — Names of classes to use for training

categorical array | cell array of character vectors | character array | logical vector | numeric vector

Names of classes to use for training, specified as the comma-separated pair consisting of `'ClassNames'` and a categorical or character array, logical or numeric vector, or cell array of character vectors. `ClassNames` must be the same data type as  $Y$ .

If `ClassNames` is a character array, then each element must correspond to one *row* of the array.

Use `ClassNames` to:

- Order the classes during training.
- Specify the order of any input or output argument dimension that corresponds to the class order. For example, use `ClassNames` to specify the order of the dimensions of `Cost` or the column order of classification scores returned by `predict`.
- Select a subset of classes for training. For example, suppose that the set of all distinct class names in `Y` is `{'a', 'b', 'c'}`. To train the model using observations from classes `'a'` and `'c'` only, specify `'ClassNames', {'a', 'c'}`.

The default is the set of all distinct class names in `Y`.

Example: `'ClassNames', {'b', 'g'}`

Data Types: `categorical` | `char` | `logical` | `single` | `double` | `cell`

### **Cost — Cost of misclassification**

`square matrix` | `structure`

Cost of misclassification of a point, specified as the comma-separated pair consisting of `'Cost'` and one of the following:

- Square matrix, where `Cost(i, j)` is the cost of classifying a point into class `j` if its true class is `i` (i.e., the rows correspond to the true class and the columns correspond to the predicted class). To specify the class order for the corresponding rows and columns of `Cost`, also specify the `ClassNames` name-value pair argument.
- Structure `S` having two fields: `S.ClassNames` containing the group names as a variable of the same data type as `Y`, and `S.ClassificationCosts` containing the cost matrix.

The default is `Cost(i, j) = 1` if `i ~ j`, and `Cost(i, j) = 0` if `i = j`.

Data Types: `single` | `double` | `struct`

### **CrossVal — Flag to grow cross-validated decision tree**

`'off'` (default) | `'on'`

Flag to grow a cross-validated decision tree, specified as the comma-separated pair consisting of `'CrossVal'` and `'on'` or `'off'`.

If 'on', `fitctree` grows a cross-validated decision tree with 10 folds. You can override this cross-validation setting using one of the 'KFold', 'Holdout', 'Leaveout', or 'CVPartition' name-value pair arguments. You can only use one of these four arguments at a time when creating a cross-validated tree.

Alternatively, cross validate tree later using the `crossval` method.

Example: 'CrossVal', 'on'

### **CVPartition** — Partition for cross-validated tree

`cvpartition` object

Partition to use in a cross-validated tree, specified as the comma-separated pair consisting of 'CVPartition' and an object created using `cvpartition`.

If you use 'CVPartition', you cannot use any of the 'KFold', 'Holdout', or 'Leaveout' name-value pair arguments.

### **Holdout** — Fraction of data for holdout validation

0 (default) | scalar value in the range [0,1]

Fraction of data used for holdout validation, specified as the comma-separated pair consisting of 'Holdout' and a scalar value in the range [0,1]. Holdout validation tests the specified fraction of the data, and uses the rest of the data for training.

If you use 'Holdout', you cannot use any of the 'CVPartition', 'KFold', or 'Leaveout' name-value pair arguments.

Example: 'Holdout', 0.1

Data Types: `single` | `double`

### **KFold** — Number of folds

10 (default) | positive integer value greater than 1

Number of folds to use in a cross-validated classifier, specified as the comma-separated pair consisting of 'KFold' and a positive integer value greater than 1. If you specify, e.g., 'KFold',  $k$ , then the software:

- 1 Randomly partitions the data into  $k$  sets
- 2 For each set, reserves the set as validation data, and trains the model using the other  $k - 1$  sets

- 3** Stores the  $k$  compact, trained models in the cells of a  $k$ -by-1 cell vector in the `Trained` property of the cross-validated model.

To create a cross-validated model, you can use one of these four options only: `CVPartition`, `Holdout`, `KFold`, or `Leaveout`.

Example: `'KFold', 8`

Data Types: `single` | `double`

#### **Leaveout — Leave-one-out cross-validation flag**

`'off'` (default) | `'on'`

Leave-one-out cross-validation flag, specified as the comma-separated pair consisting of `'Leaveout'` and `'on'` or `'off'`. Specify `'on'` to use leave-one-out cross-validation.

If you use `'Leaveout'`, you cannot use any of the `'CVPartition'`, `'Holdout'`, or `'KFold'` name-value pair arguments.

Example: `'Leaveout', 'on'`

#### **MaxNumCategories — Maximum category levels**

10 (default) | nonnegative scalar value

Maximum category levels, specified as the comma-separated pair consisting of `'MaxNumCategories'` and a nonnegative scalar value. `fitctree` splits a categorical predictor using the exact search algorithm if the predictor has at most `MaxNumCategories` levels in the split node. Otherwise, `fitctree` finds the best categorical split using one of the inexact algorithms.

Passing a small value can lead to loss of accuracy and passing a large value can increase computation time and memory overload.

Example: `'MaxNumCategories', 8`

#### **MaxNumSplits — Maximal number of decision splits**

`size(X,1) - 1` (default) | positive integer

Maximal number of decision splits (or branch nodes), specified as the comma-separated pair consisting of `'MaxNumSplits'` and a positive integer. `ClassificationTree` splits `MaxNumSplits` or fewer branch nodes. For more details on splitting behavior, see [Algorithms on page 32-2174](#).

Example: 'MaxNumSplits', 5

Data Types: single | double

### **MergeLeaves — Leaf merge flag**

'on' (default) | 'off'

Leaf merge flag, specified as the comma-separated pair consisting of 'MergeLeaves' and 'on' or 'off'.

If MergeLeaves is 'on', then ClassificationTree:

- Merges leaves that originate from the same parent node, and that yields a sum of risk values greater or equal to the risk associated with the parent node
- Estimates the optimal sequence of pruned subtrees, but does not prune the classification tree

Otherwise, ClassificationTree does not merge leaves.

Example: 'MergeLeaves', 'off'

### **MinLeafSize — Minimum number of leaf node observations**

1 (default) | positive integer value

Minimum number of leaf node observations, specified as the comma-separated pair consisting of 'MinLeafSize' and a positive integer value. Each leaf has at least MinLeafSize observations per tree leaf. If you supply both MinParentSize and MinLeafSize, fitctree uses the setting that gives larger leaves:  $\text{MinParentSize} = \max(\text{MinParentSize}, 2 * \text{MinLeafSize})$ .

Example: 'MinLeafSize', 3

Data Types: single | double

### **MinParentSize — Minimum number of branch node observations**

10 (default) | positive integer value

Minimum number of branch node observations, specified as the comma-separated pair consisting of 'MinParentSize' and a positive integer value. Each branch node in the tree has at least MinParentSize observations. If you supply both MinParentSize and MinLeafSize, fitctree uses the setting that gives larger leaves:  $\text{MinParentSize} = \max(\text{MinParentSize}, 2 * \text{MinLeafSize})$ .

Example: `'MinParentSize',8`

Data Types: `single | double`

**NumVariablesToSample** — Number of predictors to select at random for each split

`'all'` | positive integer value

Number of predictors to select at random for each split, specified as the comma-separated pair consisting of `'NumVariablesToSample'` and a positive integer value. You can also specify `'all'` to use all available predictors.

Example: `'NumVariablesToSample',3`

Data Types: `single | double`

**PredictorNames** — Predictor variable names

cell array of unique character vectors

Predictor variable names, specified as the comma-separated pair consisting of `'PredictorNames'` and a cell array of unique character vectors. The functionality of `'PredictorNames'` depends on the way you supply the training data.

- If you supply `X` and `Y`, then you can use `'PredictorNames'` to give the predictor variables in `X` names.
  - The order of the names in `PredictorNames` must correspond to the column order of `X`. That is, `PredictorNames{1}` is the name of `X(:,1)`, `PredictorNames{2}` is the name of `X(:,2)`, and so on. Also, `size(X,2)` and `numel(PredictorNames)` must be equal.
  - By default, `PredictorNames` is `{'x1','x2',...}`.
- If you supply `Tbl`, then you can use `'PredictorNames'` to choose which predictor variables to use in training. That is, `ClassificationTree` uses the predictor variables in `PredictorNames` and the response only in training.
  - `PredictorNames` must be a subset of `Tbl.Properties.VariableNames` and cannot include the name of the response variable.
  - By default, `PredictorNames` contains the names of all predictor variables.
  - It good practice to specify the predictors for training using one of `'PredictorNames'` or formula only.

```
Example: 'PredictorNames',
{'SepalLength', 'SepalWidth', 'PedalLength', 'PedalWidth'}
```

Data Types: cell

### **Prior — Prior probabilities**

```
'empirical' (default) | 'uniform' | vector of scalar values | structure
```

Prior probabilities for each class, specified as the comma-separated pair consisting of 'Prior' and one of the following.

- A character vector:
  - 'empirical' determines class probabilities from class frequencies in *Y*. If you pass observation weights, `fitctree` uses the weights to compute the class probabilities.
  - 'uniform' sets all class probabilities equal.
- A vector (one scalar value for each class). To specify the class order for the corresponding elements of `Prior`, also specify the `ClassNames` name-value pair argument.
- A structure *S* with two fields:
  - `S.ClassNames` containing the class names as a variable of the same type as *Y*
  - `S.ClassProbs` containing a vector of corresponding probabilities

If you set values for both `weights` and `prior`, the weights are renormalized to add up to the value of the prior probability in the respective class.

```
Example: 'Prior', 'uniform'
```

### **Prune — Flag to estimate optimal sequence of pruned subtrees**

```
'on' (default) | 'off'
```

Flag to estimate the optimal sequence of pruned subtrees, specified as the comma-separated pair consisting of 'Prune' and 'on' or 'off'.

If `Prune` is 'on', then `ClassificationTree` grows the classification tree without pruning it, but estimates the optimal sequence of pruned subtrees. Otherwise, `ClassificationTree` grows the classification tree without estimating the optimal sequence of pruned subtrees.

To prune a trained `ClassificationTree` model, pass it to `prune`.

Example: `'Prune', 'off'`

#### **PruneCriterion** — Pruning criterion

`'error'` (default) | `'impurity'`

Pruning criterion, specified as the comma-separated pair consisting of `'PruneCriterion'` and `'error'` or `'impurity'`.

Example: `'PruneCriterion', 'impurity'`

#### **ResponseName** — Response variable name

`'Y'` (default) | character vector

Response variable name, specified as the comma-separated pair consisting of `'ResponseName'` and a character vector representing the name of the response variable.

This name-value pair is not valid when using the `ResponseVarName` or formula input arguments.

Example: `'ResponseName', 'IrisType'`

#### **ScoreTransform** — Score transform function

`'none'` | character vector | function handle

Score transform function, specified as the comma-separated pair consisting of `'ScoreTransform'` and a function handle for transforming scores. Your function must accept a matrix (the original scores) and return a matrix of the same size (the transformed scores).

Alternatively, you can specify one of the following character vectors representing a built-in transformation function.

Value	Description
<code>'doublelogit'</code>	$1/(1 + e^{-2x})$
<code>'invlogit'</code>	$\log(x / (1-x))$
<code>'ismax'</code>	Set the score for the class with the largest score to 1, and set the scores for all other classes to 0.
<code>'logit'</code>	$1/(1 + e^{-x})$



Value	Description
'none' or 'identity'	$x$ (no transformation)
'sign'	-1 for $x < 0$ 0 for $x = 0$ 1 for $x > 0$
'symmetric'	$2x - 1$
'symmetricismax'	Set the score for the class with the largest score to 1, and set the scores for all other classes to -1.
'symmetriclogit'	$2/(1 + e^{-x}) - 1$

Example: 'ScoreTransform', 'logit'

### SplitCriterion — Split criterion

'gdi' (default) | 'twoing' | 'deviance'

Split criterion, specified as the comma-separated pair consisting of 'SplitCriterion' and 'gdi' (Gini's diversity index), 'twoing' for the twoing rule, or 'deviance' for maximum deviance reduction (also known as cross entropy).

Example: 'SplitCriterion', 'deviance'

### Surrogate — Surrogate decision splits flag

'off' (default) | 'on' | 'all' | positive integer value

Surrogate decision splits on page 32-2174 flag, specified as the comma-separated pair consisting of 'Surrogate' and 'on', 'off', 'all', or a positive integer value.

- When set to 'on', `fitctree` finds at most 10 surrogate splits at each branch node.
- When set to 'all', `fitctree` finds all surrogate splits at each branch node. The 'all' setting can use considerable time and memory.
- When set to a positive integer value, `fitctree` finds at most the specified number of surrogate splits at each branch node.

Use surrogate splits to improve the accuracy of predictions for data with missing values. The setting also lets you compute measures of predictive association between predictors. For more details, see “Node Splitting Rules” on page 32-2174.

Example: 'Surrogate', 'on'

Data Types: single | double | char

**Weights — Observation weights**`ones(size(x,1),1)` (default) | vector of scalar values

Observation weights, specified as the comma-separated pair consisting of 'Weights' and a vector of scalar values. The software weights the observations in each row of `X` or `Tbl` with the corresponding value in `Weights`. The size of `Weights` must equal the number of rows in `X` or `Tbl`.

If you specify the input data as a table `Tbl`, then `Weights` can be the name of a variable in `Tbl` that contains a numeric vector. In this case, you must specify `Weights` as a character vector. For example, if weights vector `W` is stored as `Tbl.W`, then specify it as 'W'. Otherwise, the software treats all columns of `Tbl`, including `W`, as predictors when training the model.

`fitctree` normalizes the weights in each class to add up to the value of the prior probability of the class.

Data Types: `single` | `double`

## Properties

**CategoricalPredictors**

Indices of categorical predictors, stored as a vector of positive integers. `CategoricalPredictors` contains index values corresponding to the columns of the predictor data that contain categorical predictors. If none of the predictors are categorical, then this property is empty (`[]`).

**CategoricalSplit**

An  $n$ -by-2 cell array, where  $n$  is the number of categorical splits in `tree`. Each row in `CategoricalSplits` gives left and right values for a categorical split. For each branch node with categorical split  $j$  based on a categorical predictor variable  $z$ , the left child is chosen if  $z$  is in `CategoricalSplits(j,1)` and the right child is chosen if  $z$  is in `CategoricalSplits(j,2)`. The splits are in the same order as nodes of the tree. Find the nodes for these splits by selecting 'categorical' cuts from top to bottom in the `CutType` property.

**Children**

An  $n$ -by-2 array containing the numbers of the child nodes for each node in `tree`, where  $n$  is the number of nodes. Leaf nodes have child node 0.

**ClassCount**

An  $n$ -by- $k$  array of class counts for the nodes in `tree`, where  $n$  is the number of nodes and  $k$  is the number of classes. For any node number  $i$ , the class counts `ClassCount(i, :)` are counts of observations (from the data used in fitting the tree) from each class satisfying the conditions for node  $i$ .

**ClassNames**

List of the elements in  $Y$  with duplicates removed. `ClassNames` can be a categorical array, cell array of character vectors, character array, logical vector, or a numeric vector. `ClassNames` has the same data type as the data in the argument  $Y$ .

**ClassProbability**

An  $n$ -by- $k$  array of class probabilities for the nodes in `tree`, where  $n$  is the number of nodes and  $k$  is the number of classes. For any node number  $i$ , the class probabilities `ClassProbability(i, :)` are the estimated probabilities for each class for a point satisfying the conditions for node  $i$ .

**Cost**

Square matrix, where `Cost(i, j)` is the cost of classifying a point into class  $j$  if its true class is  $i$  (i.e., the rows correspond to the true class and the columns correspond to the predicted class). The order of the rows and columns of `Cost` corresponds to the order of the classes in `ClassNames`. The number of rows and columns in `Cost` is the number of unique classes in the response. This property is read-only.

**CutCategories**

An  $n$ -by-2 cell array of the categories used at branches in `tree`, where  $n$  is the number of nodes. For each branch node  $i$  based on a categorical predictor variable  $X$ , the left child is chosen if  $X$  is among the categories listed in `CutCategories{i, 1}`, and the right child is chosen if  $X$  is among those listed in `CutCategories{i, 2}`. Both columns of `CutCategories` are empty for branch nodes based on continuous predictors and for leaf nodes.

`CutPoint` contains the cut points for 'continuous' cuts, and `CutCategories` contains the set of categories.

**CutPoint**

An  $n$ -element vector of the values used as cut points in `tree`, where  $n$  is the number of nodes. For each branch node  $i$  based on a continuous predictor variable  $X$ , the left child is chosen if  $X < \text{CutPoint}(i)$  and the right child is chosen if  $X \geq \text{CutPoint}(i)$ . `CutPoint` is `NaN` for branch nodes based on categorical predictors and for leaf nodes.

`CutPoint` contains the cut points for 'continuous' cuts, and `CutCategories` contains the set of categories.

**CutType**

An  $n$ -element cell array indicating the type of cut at each node in `tree`, where  $n$  is the number of nodes. For each node  $i$ , `CutType{i}` is:

- 'continuous' — If the cut is defined in the form  $X < v$  for a variable  $X$  and cut point  $v$ .
- 'categorical' — If the cut is defined by whether a variable  $X$  takes a value in a set of categories.
- '' — If  $i$  is a leaf node.

`CutPoint` contains the cut points for 'continuous' cuts, and `CutCategories` contains the set of categories.

**CutPredictor**

An  $n$ -element cell array of the names of the variables used for branching in each node in `tree`, where  $n$  is the number of nodes. These variables are sometimes known as *cut variables*. For leaf nodes, `CutPredictor` contains an empty character vector.

`CutPoint` contains the cut points for 'continuous' cuts, and `CutCategories` contains the set of categories.

**ExpandedPredictorNames**

Expanded predictor names, stored as a cell array of character vectors.

If the model uses encoding for categorical variables, then `ExpandedPredictorNames` includes the names that describe the expanded variables. Otherwise, `ExpandedPredictorNames` is the same as `PredictorNames`.

### **HyperparameterOptimizationResults**

Description of the cross-validation optimization of hyperparameters, stored as a `BayesianOptimization` object or a table of hyperparameters and associated values. Nonempty when the `OptimizeHyperparameters` name-value pair is nonempty at creation. Value depends on the setting of the `HyperparameterOptimizationOptions` name-value pair at creation:

- 'bayesopt' (default) — Object of class `BayesianOptimization`
- 'gridsearch' or 'randomsearch' — Table of hyperparameters used, observed objective function values (cross-validation loss), and rank of observations from lowest (best) to highest (worst)

### **IsBranchNode**

An  $n$ -element logical vector that is `true` for each branch node and `false` for each leaf node of `tree`.

### **ModelParameters**

Parameters used in training `tree`. To display all parameter values, enter `tree.ModelParameters`. To access a particular parameter, use dot notation.

### **NumObservations**

Number of observations in the training data, a numeric scalar. `NumObservations` can be less than the number of rows of input data `X` when there are missing values in `X` or response `Y`.

### **NodeClass**

An  $n$ -element cell array with the names of the most probable classes in each node of `tree`, where  $n$  is the number of nodes in the tree. Every element of this array is a character vector equal to one of the class names in `ClassNames`.

**NodeError**

An  $n$ -element vector of the errors of the nodes in `tree`, where  $n$  is the number of nodes. `NodeError(i)` is the misclassification probability for node  $i$ .

**NodeProbability**

An  $n$ -element vector of the probabilities of the nodes in `tree`, where  $n$  is the number of nodes. The probability of a node is computed as the proportion of observations from the original data that satisfy the conditions for the node. This proportion is adjusted for any prior probabilities assigned to each class.

**NodeRisk**

An  $n$ -element vector of the risk of the nodes in the tree, where  $n$  is the number of nodes. The risk for each node is the measure of impurity (Gini index or deviance) for this node weighted by the node probability. If the tree is grown by `twoing`, the risk for each node is zero.

**NodeSize**

An  $n$ -element vector of the sizes of the nodes in `tree`, where  $n$  is the number of nodes. The size of a node is defined as the number of observations from the data used to create the tree that satisfy the conditions for the node.

**NumNodes**

The number of nodes in `tree`.

**Parent**

An  $n$ -element vector containing the number of the parent node for each node in `tree`, where  $n$  is the number of nodes. The parent of the root node is 0.

**PredictorNames**

Cell array of character vectors containing the predictor names, in the order which they appear in `X`.

**Prior**

Numeric vector of prior probabilities for each class. The order of the elements of `Prior` corresponds to the order of the classes in `ClassNames`. The number of elements of `Prior` is the number of unique classes in the response. This property is read-only.

**PruneAlpha**

Numeric vector with one element per pruning level. If the pruning level ranges from 0 to  $M$ , then `PruneAlpha` has  $M + 1$  elements sorted in ascending order. `PruneAlpha(1)` is for pruning level 0 (no pruning), `PruneAlpha(2)` is for pruning level 1, and so on.

**PruneList**

An  $n$ -element numeric vector with the pruning levels in each node of `tree`, where  $n$  is the number of nodes. The pruning levels range from 0 (no pruning) to  $M$ , where  $M$  is the distance between the deepest leaf and the root node.

**ResponseName**

A character vector that specifies the name of the response variable ( $\hat{y}$ ).

**RowsUsed**

An  $n$ -element logical vector indicating which rows of the original predictor data ( $X$ ) were used in fitting. If the software uses all rows of  $X$ , then `RowsUsed` is an empty array (`[]`).

**ScoreTransform**

Function handle for transforming predicted classification scores, or character vector representing a built-in transformation function.

`none` means no transformation, or `@(x)x`.

To change the score transformation function to, e.g., `function`, use dot notation.

- For available functions (see `fitctree`), enter

```
Mdl.ScoreTransform = 'function';
```

- You can set a function handle for an available function, or a function you define yourself by entering

```
tree.ScoreTransform = @function;
```

**SurrogateCutCategories**

An  $n$ -element cell array of the categories used for surrogate splits in `tree`, where  $n$  is the number of nodes in `tree`. For each node  $k$ , `SurrogateCutCategories{k}` is a cell array. The length of `SurrogateCutCategories{k}` is equal to the number of surrogate predictors found at this node. Every element of `SurrogateCutCategories{k}` is either an empty character vector for a continuous surrogate predictor, or is a two-element cell array with categories for a categorical surrogate predictor. The first element of this two-element cell array lists categories assigned to the left child by this surrogate split, and the second element of this two-element cell array lists categories assigned to the right child by this surrogate split. The order of the surrogate split variables at each node is matched to the order of variables in `SurrogateCutPredictor`. The optimal-split variable at this node does not appear. For nonbranch (leaf) nodes, `SurrogateCutCategories` contains an empty cell.

**SurrogateCutFlip**

An  $n$ -element cell array of the numeric cut assignments used for surrogate splits in `tree`, where  $n$  is the number of nodes in `tree`. For each node  $k$ , `SurrogateCutFlip{k}` is a numeric vector. The length of `SurrogateCutFlip{k}` is equal to the number of surrogate predictors found at this node. Every element of `SurrogateCutFlip{k}` is either zero for a categorical surrogate predictor, or a numeric cut assignment for a continuous surrogate predictor. The numeric cut assignment can be either  $-1$  or  $+1$ . For every surrogate split with a numeric cut  $C$  based on a continuous predictor variable  $Z$ , the left child is chosen if  $Z < C$  and the cut assignment for this surrogate split is  $+1$ , or if  $Z \geq C$  and the cut assignment for this surrogate split is  $-1$ . Similarly, the right child is chosen if  $Z \geq C$  and the cut assignment for this surrogate split is  $+1$ , or if  $Z < C$  and the cut assignment for this surrogate split is  $-1$ . The order of the surrogate split variables at each node is matched to the order of variables in `SurrogateCutPredictor`. The optimal-split variable at this node does not appear. For nonbranch (leaf) nodes, `SurrogateCutFlip` contains an empty array.

**SurrogateCutPoint**

An  $n$ -element cell array of the numeric values used for surrogate splits in `tree`, where  $n$  is the number of nodes in `tree`. For each node  $k$ , `SurrogateCutPoint{k}` is a numeric vector. The length of `SurrogateCutPoint{k}` is equal to the number of surrogate predictors found at this node. Every element of `SurrogateCutPoint{k}` is either `NaN` for a categorical surrogate predictor, or a numeric cut for a continuous surrogate predictor. For every surrogate split with a numeric cut  $C$  based on a continuous predictor



variable  $Z$ , the left child is chosen if  $Z < C$  and `SurrogateCutFlip` for this surrogate split is +1, or if  $Z \geq C$  and `SurrogateCutFlip` for this surrogate split is -1. Similarly, the right child is chosen if  $Z \geq C$  and `SurrogateCutFlip` for this surrogate split is +1, or if  $Z < C$  and `SurrogateCutFlip` for this surrogate split is -1. The order of the surrogate split variables at each node is matched to the order of variables returned by `SurrogateCutPredictor`. The optimal-split variable at this node does not appear. For nonbranch (leaf) nodes, `SurrogateCutPoint` contains an empty cell.

### **SurrogateCutType**

An  $n$ -element cell array indicating types of surrogate splits at each node in `tree`, where  $n$  is the number of nodes in `tree`. For each node  $k$ , `SurrogateCutType{k}` is a cell array with the types of the surrogate split variables at this node. The variables are sorted by the predictive measure of association with the optimal predictor in the descending order, and only variables with the positive predictive measure are included. The order of the surrogate split variables at each node is matched to the order of variables in `SurrogateCutPredictor`. The optimal-split variable at this node does not appear. For nonbranch (leaf) nodes, `SurrogateCutType` contains an empty cell. A surrogate split type can be either 'continuous' if the cut is defined in the form  $Z < V$  for a variable  $Z$  and cut point  $V$  or 'categorical' if the cut is defined by whether  $Z$  takes a value in a set of categories.

### **SurrogateCutPredictor**

An  $n$ -element cell array of the names of the variables used for surrogate splits in each node in `tree`, where  $n$  is the number of nodes in `tree`. Every element of `SurrogateCutPredictor` is a cell array with the names of the surrogate split variables at this node. The variables are sorted by the predictive measure of association with the optimal predictor in the descending order, and only variables with the positive predictive measure are included. The optimal-split variable at this node does not appear. For nonbranch (leaf) nodes, `SurrogateCutPredictor` contains an empty cell.

### **SurrogatePredictorAssociation**

An  $n$ -element cell array of the predictive measures of association for surrogate splits in `tree`, where  $n$  is the number of nodes in `tree`. For each node  $k$ , `SurrogatePredictorAssociation{k}` is a numeric vector. The length of `SurrogatePredictorAssociation{k}` is equal to the number of surrogate predictors found at this node. Every element of `SurrogatePredictorAssociation{k}` gives the predictive measure of association between the optimal split and this surrogate split. The

order of the surrogate split variables at each node is the order of variables in `SurrogateCutPredictor`. The optimal-split variable at this node does not appear. For nonbranch (leaf) nodes, `SurrogatePredictorAssociation` contains an empty cell.

**w**

The scaled `weights`, a vector with length  $n$ , the number of rows in `x`.

**x**

A matrix of predictor values. Each column of `x` represents one variable, and each row represents one observation.

**y**

A categorical array, cell array of character vectors, character array, logical vector, or a numeric vector. Each row of `y` represents the classification of the corresponding row of `x`.

## Methods

<code>compact</code>	Compact tree
<code>crossval</code>	Cross-validated decision tree
<code>cvloss</code>	Classification error by cross validation
<code>prune</code>	Produce sequence of subtrees by pruning
<code>resubEdge</code>	Classification edge by resubstitution
<code>resubLoss</code>	Classification error by resubstitution
<code>resubMargin</code>	Classification margins by resubstitution
<code>resubPredict</code>	Predict resubstitution response of tree

## Inherited Methods

<code>compareHoldout</code>	Compare accuracies of two classification models using new data
<code>edge</code>	Classification edge
<code>loss</code>	Classification error
<code>margin</code>	Classification margins
<code>surrogateAssociation</code>	Mean predictive measure of association for surrogate splits in decision tree
<code>predict</code>	Predict labels using classification tree
<code>predictorImportance</code>	Estimates of predictor importance
<code>view</code>	View tree

## Copy Semantics

Value. To learn how value classes affect copy operations, see [Copying Objects \(MATLAB\)](#).

## Examples

### Grow a Classification Tree

Grow a classification tree using the ionosphere data set.

```
load ionosphere
tc = fitctree(X,Y)

tc =

ClassificationTree
    ResponseName: 'Y'
    CategoricalPredictors: []
        ClassNames: {'b' 'g'}
    ScoreTransform: 'none'
    NumObservations: 351
```

## Control Tree Depth

You can control the depth of the trees using the `MaxNumSplits`, `MinLeafSize`, or `MinParentSize` name-value pair parameters. `fitctree` grows deep decision trees by default. You can grow shallower trees to reduce model complexity or computation time.

Load the `ionosphere` data set.

```
load ionosphere
```

The default values of the tree depth controllers for growing classification trees are:

- `n - 1` for `MaxNumSplits`. `n` is the training sample size.
- `1` for `MinLeafSize`.
- `10` for `MinParentSize`.

These default values tend to grow deep trees for large training sample sizes.

Train a classification tree using the default values for tree depth control. Cross validate the model using 10-fold cross validation.

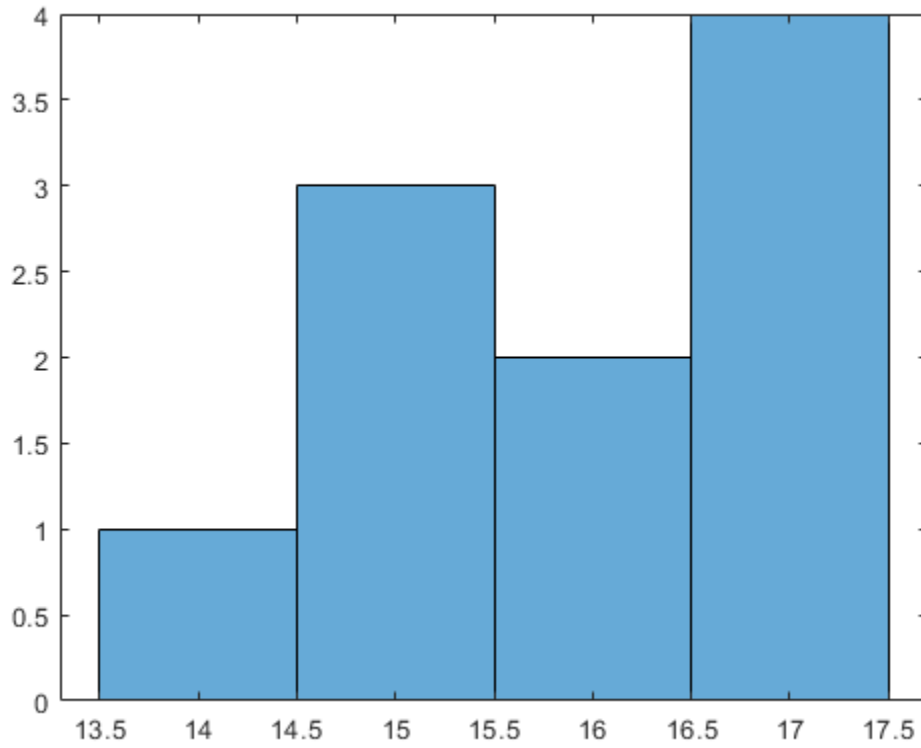
```
rng(1); % For reproducibility
MdlDefault = fitctree(X,Y,'CrossVal','on');
```

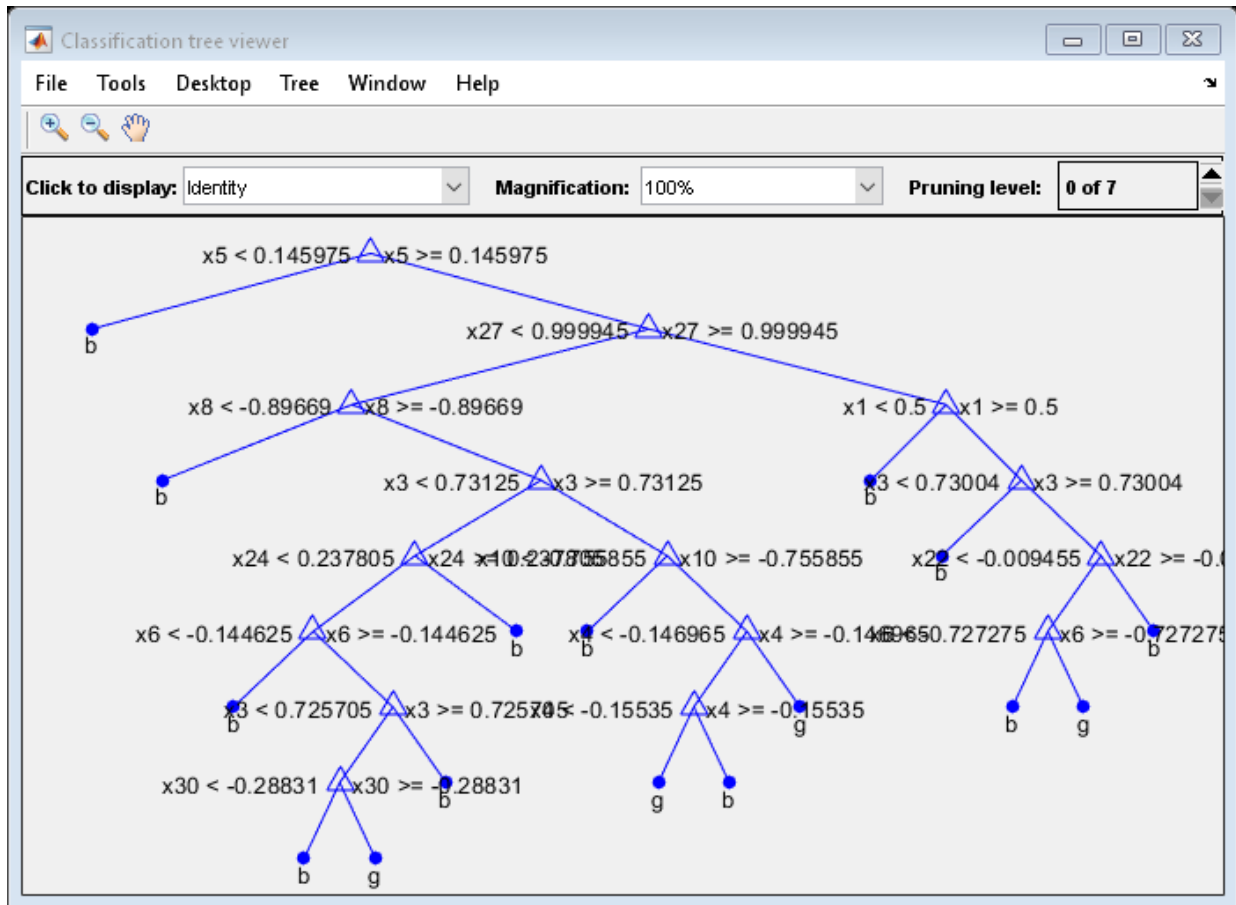
Draw a histogram of the number of imposed splits on the trees. Also, view one of the trees.

```
numBranches = @(x)sum(x.IsBranch);
mdlDefaultNumSplits = cellfun(numBranches, MdlDefault.Trained);

figure;
histogram(mdlDefaultNumSplits)

view(MdlDefault.Trained{1}, 'Mode', 'graph')
```

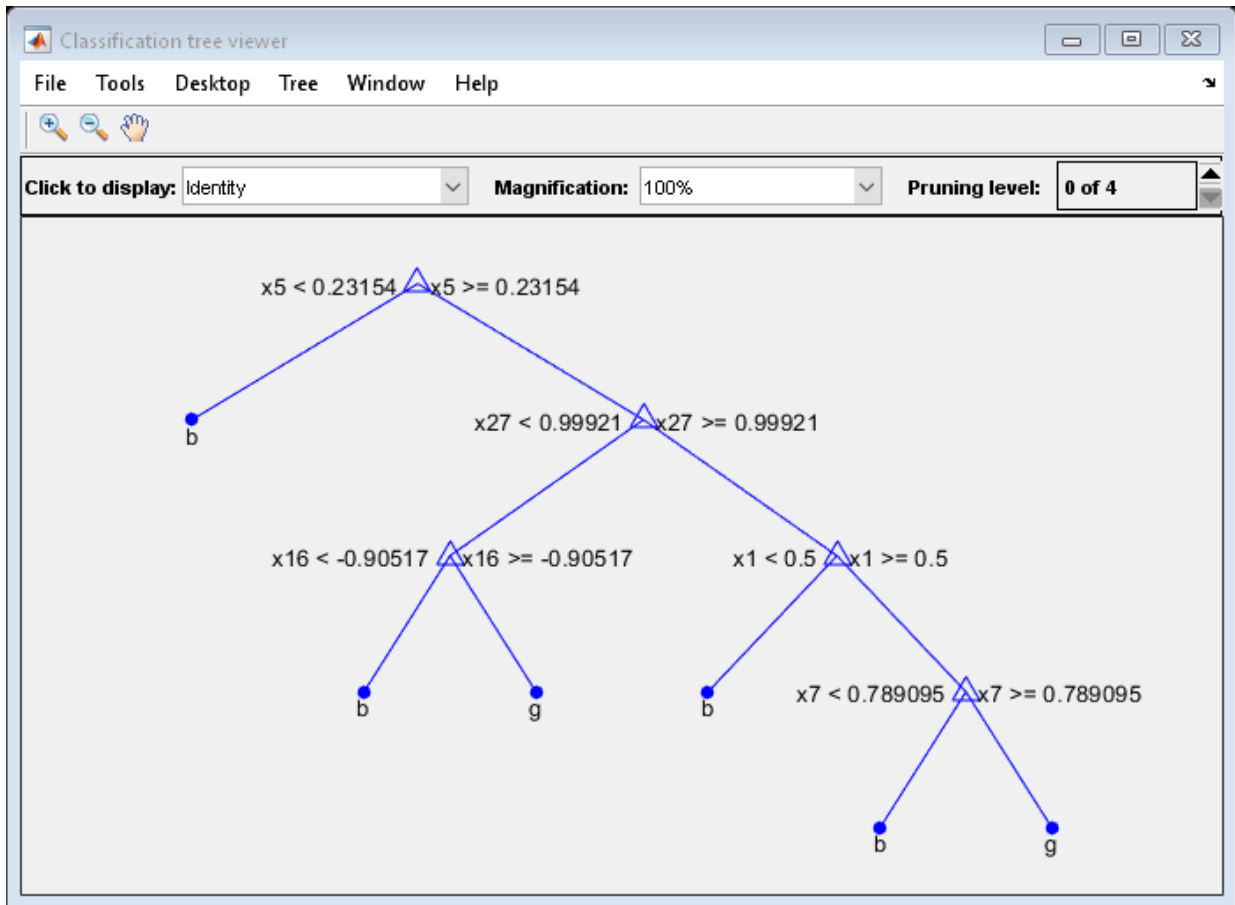




The average number of splits is around 15.

Suppose that you want a classification tree that is not as complex (deep) as the ones trained using the default number of splits. Train another classification tree, but set the maximum number of splits at 7, which is about half the mean number of splits from the default classification tree. Cross validate the model using 10-fold cross validation.

```
Mdl7 = fitctree(X,Y,'MaxNumSplits',7,'CrossVal','on');
view(Mdl7.Trained{1},'Mode','graph')
```



Compare the cross validation classification errors of the models.

```
classErrorDefault = kfoldLoss(MdlDefault)
classError7 = kfoldLoss(Mdl7)
```

```
classErrorDefault =
```

```
    0.1140
```

```
classError7 =
```

0.1254

Mdl17 is much less complex and performs only slightly worse than MdlDefault.

## Definitions

### Impurity and Node Error

ClassificationTree splits nodes based on either impurity or node error.

Impurity means one of several things, depending on your choice of the SplitCriterion name-value pair argument:

- Gini's Diversity Index (gdi) — The Gini index of a node is

$$1 - \sum_i p^2(i),$$

where the sum is over the classes  $i$  at the node, and  $p(i)$  is the observed fraction of classes with class  $i$  that reach the node. A node with just one class (a pure node) has Gini index 0; otherwise the Gini index is positive. So the Gini index is a measure of node impurity.

- Deviance ('deviance') — With  $p(i)$  defined the same as for the Gini index, the deviance of a node is

$$-\sum_i p(i) \log_2 p(i).$$

A pure node has deviance 0; otherwise, the deviance is positive.

- Twoing rule ('twoing') — Twoing is not a purity measure of a node, but is a different measure for deciding how to split a node. Let  $L(i)$  denote the fraction of members of class  $i$  in the left child node after a split, and  $R(i)$  denote the fraction of members of class  $i$  in the right child node after a split. Choose the split criterion to maximize

$$P(L)P(R) \left( \sum_i |L(i) - R(i)| \right)^2,$$



where  $P(L)$  and  $P(R)$  are the fractions of observations that split to the left and right respectively. If the expression is large, the split made each child node purer. Similarly, if the expression is small, the split made each child node similar to each other, and hence similar to the parent node, and so the split did not increase node purity.

- Node error — The node error is the fraction of misclassified classes at a node. If  $j$  is the class with the largest number of training samples at a node, the node error is

$$1 - p(j).$$

## References

- [1] Breiman, L., J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Boca Raton, FL: CRC Press, 1984.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Only the `predict` function supports code generation.
- When you train a classification tree using `fitctree`, the following restrictions apply.
  - The class labels input argument value ( $Y$ ) cannot be a categorical array.
  - Code generation does not support categorical predictors. You cannot supply training data in a table that contains at least one categorical predictor, and you cannot use the 'CategoricalPredictors' name-value pair argument. To dummy-code variables that you want treated as categorical, preprocess the categorical data by using `dummyvar` before fitting the model.
  - The value of the 'ClassNames' name-value pair argument cannot be a categorical array.

- The value of the 'ScoreTransform' name-value pair argument cannot be an anonymous function.
- You cannot use surrogate splits, that is, the value of the 'Surrogate' name-value pair argument must be 'off'.

## See Also

`ClassificationEnsemble` | `CompactClassificationTree` | `RegressionTree` | `fitctree` | `predict`

## Topics

“Decision Trees” on page 19-2

**Introduced in R2011a**

# classify

Discriminant analysis

## Syntax

```
class = classify(sample, training, group)
class = classify(sample, training, group, 'type')
class = classify(sample, training, group, 'type', prior)
[class, err] = classify(...)
[class, err, POSTERIOR] = classify(...)
[class, err, POSTERIOR, logp] = classify(...)
[class, err, POSTERIOR, logp, coeff] = classify(...)
```

## Description

`class = classify(sample, training, group)` classifies each row of the data in `sample` into one of the groups in `training`. `sample` and `training` must be matrices with the same number of columns. `group` is a grouping variable for `training`. Its unique values define groups; each element defines the group to which the corresponding row of `training` belongs. `group` can be a categorical variable, a numeric vector, a character array, or a cell array of character vectors. `training` and `group` must have the same number of rows. `classify` treats NaNs or empty character vectors in `group` as missing values, and ignores the corresponding rows of `training`. The output `class` indicates the group to which each row of `sample` has been assigned, and is of the same type as `group`.

`class = classify(sample, training, group, 'type')` allows you to specify the type of discriminant function. Specify `type` inside single quotes. `type` is one of:

- `linear` — Fits a multivariate normal density to each group, with a pooled estimate of covariance. This is the default.
- `diaglinear` — Similar to `linear`, but with a diagonal covariance matrix estimate (naive Bayes classifiers).

- `quadratic` — Fits multivariate normal densities with covariance estimates stratified by `group`.
- `diagquadratic` — Similar to `quadratic`, but with a diagonal covariance matrix estimate (naive Bayes classifiers).
- `mahalanobis` — Uses Mahalanobis distances with stratified covariance estimates.

`class = classify(sample, training, group, 'type', prior)` allows you to specify prior probabilities for the groups. `prior` is one of:

- A numeric vector the same length as the number of unique values in `group` (or the number of levels defined for `group`, if `group` is categorical). If `group` is numeric or categorical, the order of `prior` must correspond to the ordered values in `group`, or, if `group` contains character vectors, to the order of first occurrence of the values in `group`.
- A 1-by-1 structure with fields:
  - `prob` — A numeric vector.
  - `group` — Of the same type as `group`, containing unique values indicating the groups to which the elements of `prob` correspond.

As a structure, `prior` can contain groups that do not appear in `group`. This can be useful if `training` is a subset a larger training set. `classify` ignores any groups that appear in the structure but not in the `group` array.

- The character vector `'empirical'`, indicating that group prior probabilities should be estimated from the group relative frequencies in `training`.

`prior` defaults to a numeric vector of equal probabilities, i.e., a uniform distribution. `prior` is not used for discrimination by Mahalanobis distance, except for error rate calculation.

`[class, err] = classify(...)` also returns an estimate `err` of the misclassification error rate based on the training data. `classify` returns the apparent error rate, i.e., the percentage of observations in `training` that are misclassified, weighted by the prior probabilities for the groups.

`[class, err, POSTERIOR] = classify(...)` also returns a matrix `POSTERIOR` of estimates of the posterior probabilities that the  $j$ th training group was the source of the  $i$ th sample observation, i.e.,  $Pr(\text{group } j | \text{obs } i)$ . `POSTERIOR` is not computed for Mahalanobis discrimination.

`[class,err,POSTERIOR,logp] = classify(...)` also returns a vector `logp` containing estimates of the logarithms of the unconditional predictive probability density of the sample observations,  $p(\text{obs } i) = \sum p(\text{obs } i | \text{group } j)Pr(\text{group } j)$  over all groups. `logp` is not computed for Mahalanobis discrimination.

`[class,err,POSTERIOR,logp,coeff] = classify(...)` also returns a structure array `coeff` containing coefficients of the boundary curves between pairs of groups. Each element `coeff(I,J)` contains information for comparing group `I` to group `J` in the following fields:

- `type` — Type of discriminant function, from the `type` input.
- `name1` — Name of the first group.
- `name2` — Name of the second group.
- `const` — Constant term of the boundary equation (K)
- `linear` — Linear coefficients of the boundary equation (L)
- `quadratic` — Quadratic coefficient matrix of the boundary equation (Q)

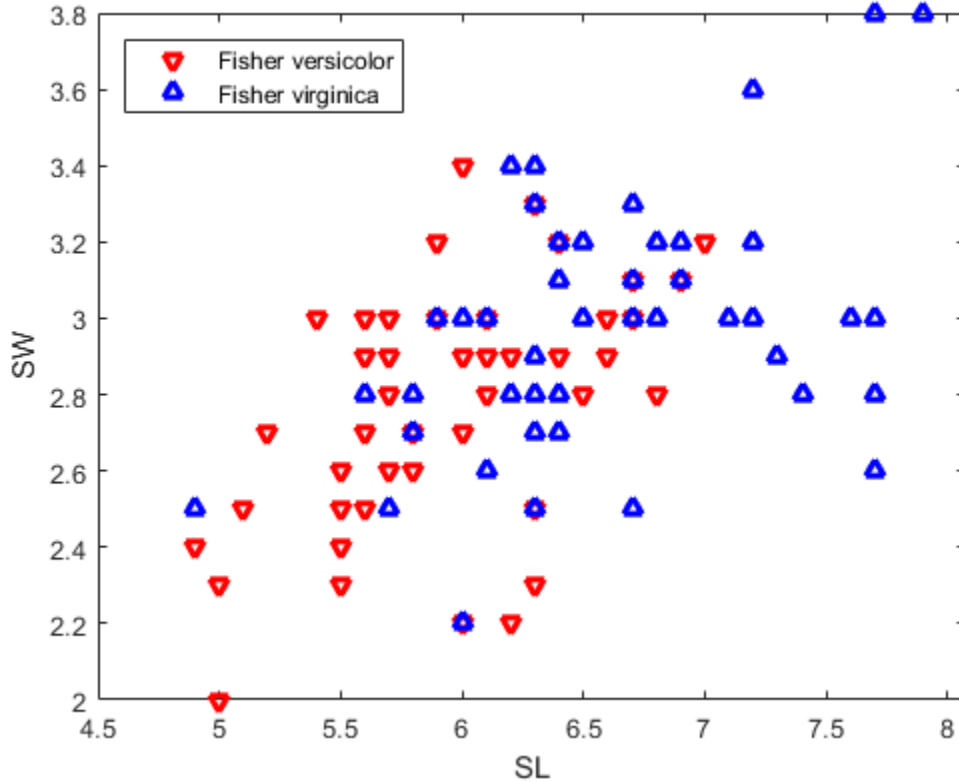
For the `linear` and `diaglinear` types, the `quadratic` field is absent, and a row `x` from the sample array is classified into group `I` rather than group `J` if  $0 < K+x*L$ . For the other types, `x` is classified into group `I` if  $0 < K+x*L+x*Q*x'$ .

## Examples

### Classify Using Discriminant Analysis

For training data, use Fisher's sepal measurements for iris versicolor and virginica:

```
load fisheriris
SL = meas(51:end,1);
SW = meas(51:end,2);
group = species(51:end);
h1 = gscatter(SL,SW,group,'rb','v^',[],'off');
set(h1,'LineWidth',2)
legend('Fisher versicolor','Fisher virginica',...
      'Location','NW')
```



Classify a grid of measurements on the same scale:

```
[X,Y] = meshgrid(linspace(4.5,8),linspace(2,4));
X = X(:); Y = Y(:);
[C,err,P,logp,coeff] = classify([X Y],[SL SW],...
                               group,'Quadratic');
```

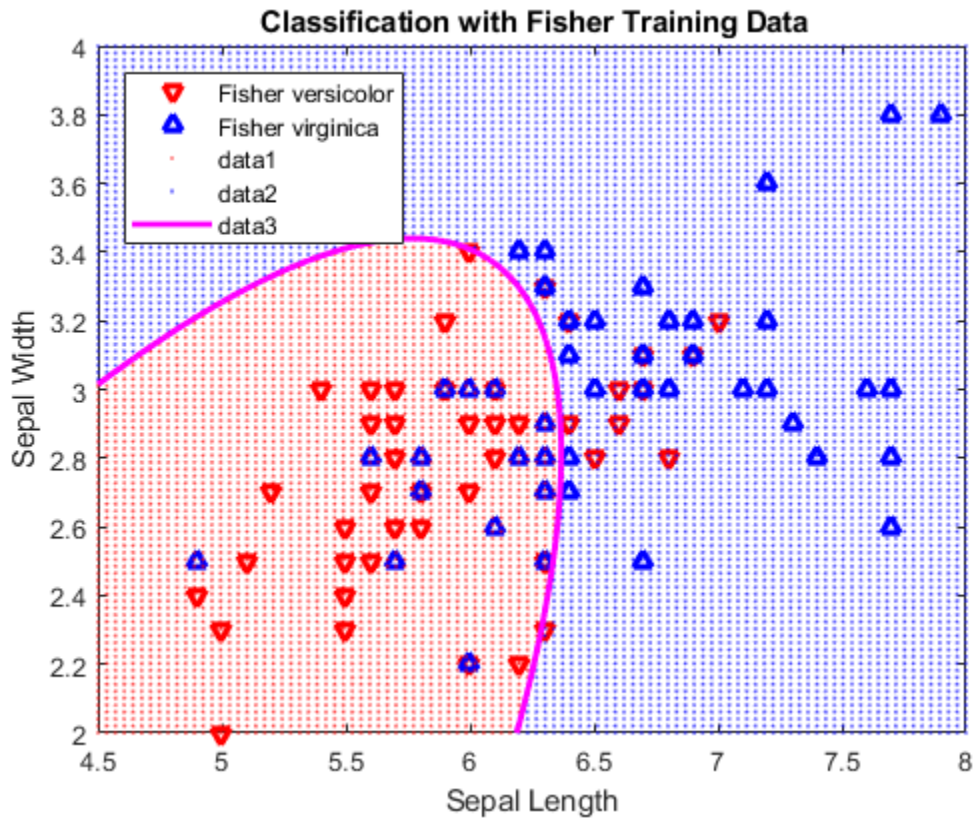
Visualize the classification:

```
hold on;
gscatter(X,Y,C,'rb','.',1,'off');
K = coeff(1,2).const;
L = coeff(1,2).linear;
Q = coeff(1,2).quadratic;
```

---

```
% Function to compute  $K + L*v + v'*Q*v$  for multiple vectors
%  $v=[x;y]$ . Accepts  $x$  and  $y$  as scalars or column vectors.
f = @(x,y) K + [x y]*L + sum(([x y]*Q) .* [x y], 2);

h2 = ezplot(f,[4.5 8 2 4]);
set(h2, 'Color', 'm', 'LineWidth', 2)
axis([4.5 8 2 4])
xlabel('Sepal Length')
ylabel('Sepal Width')
title('\bf Classification with Fisher Training Data')
```



## References

- [1] Krzanowski, W. J. *Principles of Multivariate Analysis: A User's Perspective*. New York: Oxford University Press, 1988.
- [2] Seber, G. A. F. *Multivariate Observations*. Hoboken, NJ: John Wiley & Sons, Inc., 1984.

## See Also

`fitNaiveBayes` | `fitctree` | `mahal`



## **Topics**

“Grouping Variables” on page 2-59

**Introduced before R2006a**

## classname

**Class:** classregtree

Class names for classification decision tree

---

**Note** `classregtree` will be removed in a future release. See `fitctree`, `fitrtree`, `ClassificationTree`, or `RegressionTree` instead.

---

## Syntax

CNAMES = `classname`(T)

CNAMES = `classname`(T, J)

## Description

CNAMES = `classname`(T) returns a cell array of character vectors with class names for this classification decision tree.

CNAMES = `classname`(T, J) takes an array J of class numbers and returns the class names for the specified numbers.

## See Also

`classregtree`

## ClassNames property

**Class:** TreeBagger

Names of classes

### Description

The `ClassNames` property is a cell array containing the class names for the response variable  $Y$ . This property is empty for regression trees.

## classprob

**Class:** classregtree

Class probabilities

---

**Note** `classregtree` will be removed in a future release. See `fitctree`, `fitrtree`, `ClassificationTree`, or `RegressionTree` instead.

---

## Syntax

```
P = classprob(t)
P = classprob(t,nodes)
```

## Description

`P = classprob(t)` returns an  $n$ -by- $m$  array `P` of class probabilities for the nodes in the classification tree `t`, where  $n$  is the number of nodes and  $m$  is the number of classes. For any node number  $i$ , the class probabilities `P(i, :)` are the estimated probabilities for each class for a point satisfying the conditions for node  $i$ .

`P = classprob(t,nodes)` takes a vector `nodes` of node numbers and returns the class probabilities for the specified nodes.

## Examples

Create a classification tree for Fisher's iris data:

```
load fisheriris;

t = classregtree(meas,species,...
                'names',{'SL' 'SW' 'PL' 'PW'})
t =
Decision tree for classification
1  if PL<2.45 then node 2 elseif PL>=2.45 then node 3 else setosa
2  class = setosa
3  if PW<1.75 then node 4 elseif PW>=1.75 then node 5 else versicolor
```

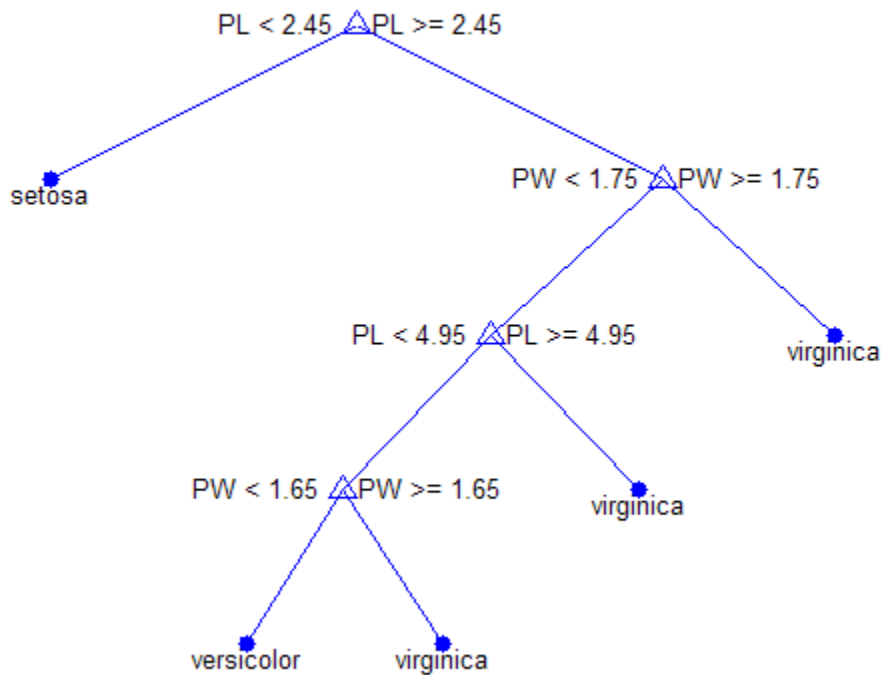
```

4 if PL<4.95 then node 6 elseif PL>=4.95 then node 7 else versicolor
5 class = virginica
6 if PW<1.65 then node 8 elseif PW>=1.65 then node 9 else versicolor
7 class = virginica
8 class = versicolor
9 class = virginica

```

```
view(t)
```

Click to display:  Magnification:  Pruning level:



```
P = classprob(t)
```

```
P =
```

```

0.3333  0.3333  0.3333
1.0000  0        0
0        0.5000  0.5000
0        0.9074  0.0926
0        0.0217  0.9783

```

0	0.9792	0.0208
0	0.3333	0.6667
0	1.0000	0
0	0	1.0000

## References

- [1] Breiman, L., J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Boca Raton, FL: CRC Press, 1984.

## See Also

`classregtree` | `numnodes`

## classregtree class

Classification and regression trees

---

**Note** Use `ClassificationTree` and `RegressionTree` classes instead. This class is superseded by `ClassificationTree` and `RegressionTree` classes and is maintained only for backwards compatibility.

---

### Construction

`classregtree`      Construct classification and regression trees

## Methods

<code>catsplit</code>	Categorical splits used for branches in decision tree
<code>children</code>	Child nodes
<code>classcount</code>	Class counts
<code>classname</code>	Class names for classification decision tree
<code>classprob</code>	Class probabilities
<code>cutcategories</code>	Cut categories
<code>cutpoint</code>	Decision tree cut point values
<code>cuttype</code>	Cut types
<code>cutvar</code>	Cut variable names
<code>disp</code>	Display <code>classregtree</code> object
<code>display</code>	Display <code>classregtree</code> object
<code>eval</code>	Predicted responses
<code>isbranch</code>	Test node for branch
<code>meansurrvarassoc</code>	Mean predictive measure of association for surrogate splits in decision tree
<code>nodeclass</code>	Class values of nodes of classification tree
<code>nodeerr</code>	Return vector of node errors
<code>nodemean</code>	Mean values of nodes of regression tree
<code>nodeprob</code>	Node probabilities
<code>nodesize</code>	Return node size
<code>numnodes</code>	Number of nodes
<code>parent</code>	Parent node
<code>prune</code>	Prune tree
<code>prunelist</code>	Pruning levels for decision tree nodes
<code>risk</code>	Node risks
<code>subsasgn</code>	Subscripted reference for <code>classregtree</code> object
<code>subsref</code>	Subscripted reference for <code>classregtree</code> object
<code>surrutcategories</code>	Categories used for surrogate splits in decision tree
<code>surrutflip</code>	Numeric cutpoint assignments used for surrogate splits in decision tree
<b>32-594</b> <code>surrutpoint</code>	Cutpoints used for surrogate splits in decision tree
<code>surruttype</code>	Types of surrogate splits used at branches in decision tree
<code>surrutvar</code>	Variables used for surrogate splits in decision tree
<code>surrvarassoc</code>	Predictive measure of association for surrogate splits in decision tree



## Properties

Objects of the `classregtree` class have no properties accessible by dot indexing, get methods, or set methods. To obtain information about a `classregtree` object, use the appropriate method.

## Copy Semantics

Value. To learn how this affects your use of the class, see [Comparing Handle and Value Classes \(MATLAB\)](#) in the MATLAB Object-Oriented Programming documentation.

## See Also

### Topics

“Framework for Ensemble Learning” on page 18-41

“Decision Trees” on page 19-2

“Grouping Variables” on page 2-59

## classregtree

**Class:** classregtree

Construct classification and regression trees

---

**Note** Use `fitctree` or `fitrtree` instead. This function is superseded by `fitctree` and `fitrtree` of `ClassificationTree` and `RegressionTree` classes. It is maintained only for backwards compatibility.

---

## Syntax

```
t = classregtree(X, y)
t = classregtree(X, y, 'Name', value)
```

## Description

`t = classregtree(X, y)` creates a decision tree `t` for predicting the response `y` as a function of the predictors in the columns of `X`. `X` is an  $n$ -by- $m$  matrix of predictor values. If `y` is a vector of  $n$  response values, `classregtree` performs regression. If `y` is a categorical variable, character array, or cell array of character vectors, `classregtree` performs classification. Either way, `t` is a binary tree where each branching node is split based on the values of a column of `X`. NaN values in `X` or `y` are taken to be missing values. Observations with all missing values for `X` or missing values for `y` are not used in the fit. Observations with some missing values for `X` are used to find splits on variables for which these observations have valid values.

`t = classregtree(X, y, 'Name', value)` specifies one or more optional parameter name/value pairs. Specify `Name` in single quotes. The following options are available:

For all trees:

- `categorical` — Vector of indices of the columns of `X` that are to be treated as unordered categorical variables

- `method` — Either 'classification' (default if `y` is text or a categorical variable) or 'regression' (default if `y` is numeric).
- `names` — A cell array of names for the predictor variables, in the order in which they appear in the `X` from which the tree was created.
- `prune` — 'on' (default) to compute the full tree and the optimal sequence of pruned subtrees, or 'off' for the full tree without pruning.
- `minparent` — A number  $k$  such that impure nodes must have  $k$  or more observations to be split (default is 10).
- `minleaf` — A minimal number of observations per tree leaf (default is 1). If you supply both 'minparent' and 'minleaf', `classregtree` uses the setting which results in larger leaves:  $\text{minparent} = \max(\text{minparent}, 2 * \text{minleaf})$
- `mergeleaves` — 'on' (default) to merge leaves that originate from the same parent node and give the sum of risk values greater or equal to the risk associated with the parent node. If 'off', `classregtree` does not merge leaves.
- `nvarsample` — Number of predictor variables randomly selected for each split. By default all variables are considered for each decision split.
- `stream` — Random number stream. Default is the MATLAB default random number stream.
- `surrogate` — 'on' to find surrogate splits at each branch node. Default is 'off'. If you set this parameter to 'on', `classregtree` can run significantly slower and consume significantly more memory.
- `weights` — Vector of observation weights. By default the weight of every observation is 1. The length of this vector must be equal to the number of rows in `X`.

For regression trees only:

- `getoler` — Defines tolerance on quadratic error per node for regression trees. Splitting nodes stops when quadratic error per node drops below  $\text{getoler} * \text{qed}$ , where  $\text{qed}$  is the quadratic error for the entire data computed before the decision tree is grown:  $\text{qed} = \text{norm}(y - \text{ybar})$  with  $\text{ybar}$  estimated as the average of the input array `Y`. Default value is 1e-6.

For classification trees only:

- `cost` — Square matrix  $C$ , where  $C(i, j)$  is the cost of classifying a point into class  $j$  if its true class is  $i$  (default has  $C(i, j) = 1$  if  $i \neq j$ , and  $C(i, j) = 0$  if  $i = j$ ). Alternatively, this value can be a structure `S` having two fields: `S.group` containing

the group names as a categorical variable, character array, or cell array of character vectors; and `S.cost` containing the cost matrix `C`.

- `splitcriterion` — Criterion for choosing a split. One of 'gdi' (default) or Gini's diversity index, 'twoing' for the twoing rule, or 'deviance' for maximum deviance reduction.
- `priorprob` — Prior probabilities for each class, specified as 'empirical' or 'equal', or as a vector (one value for each distinct group name) or as a structure `S` with two fields:
  - `S.group` containing the group names as a categorical variable, character array, or cell array of character vectors.
  - `S.prob` containing a vector of corresponding probabilities.

If the input value is 'empirical' (default), class probabilities are determined from class frequencies in `Y`. If the input value is 'equal', all class probabilities are set equal. If both observation weights and class prior probabilities are supplied, the weights are renormalized to add up to the value of the prior probability in the respective class.

## Examples

### Plot a Classification Tree

Create a classification decision tree for Fisher's iris data:

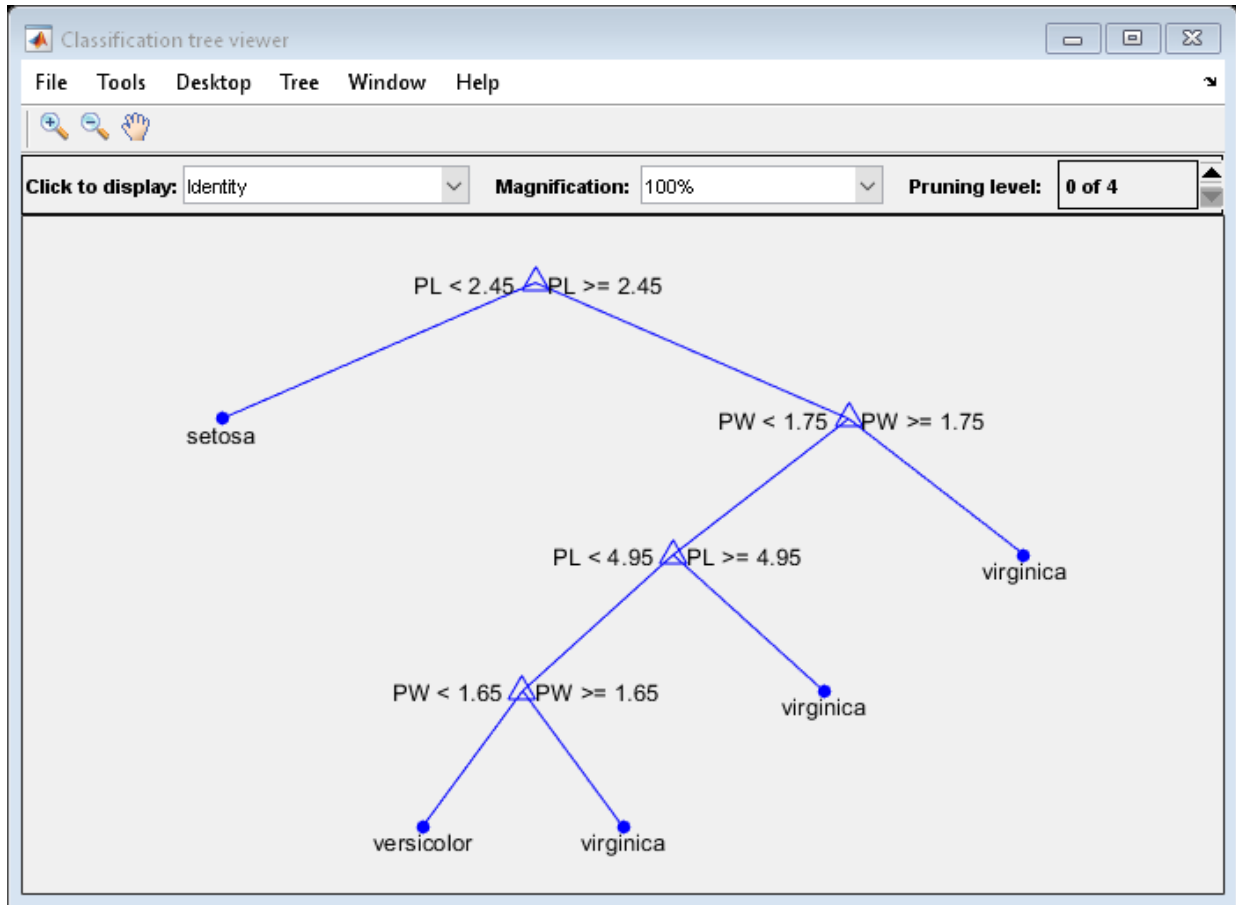
```
load fisheriris;
t = classregtree(meas, species, ...
                'names', {'SL' 'SW' 'PL' 'PW'})
view(t)
```

Warning: `classregtree` will be removed in a future release. Use `fitctree` or `fitrtree` instead.

```
t =
```

```
Decision tree for classification
1  if PL<2.45 then node 2 elseif PL>=2.45 then node 3 else setosa
2  class = setosa
3  if PW<1.75 then node 4 elseif PW>=1.75 then node 5 else versicolor
```

```
4 if PL<4.95 then node 6 elseif PL>=4.95 then node 7 else versicolor
5 class = virginica
6 if PW<1.65 then node 8 elseif PW>=1.65 then node 9 else versicolor
7 class = virginica
8 class = versicolor
9 class = virginica
```



## References

- [1] Breiman, L., J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Boca Raton, FL: CRC Press, 1984.

## See Also

`eval` | `fitctree` | `fitrtree` | `prune` | `test` | `view`

## **Topics**

“Grouping Variables” on page 2-59

“Framework for Ensemble Learning” on page 18-41

**Introduced in R2007a**

## ClassLevels property

**Class:** NaiveBayes

Class levels

### Description

The `ClassLevels` property is a vector of the same type as the grouping variable, containing the unique levels of the grouping variable.



## cluster

Construct agglomerative clusters from linkages

### Syntax

```
T = cluster(Z, 'cutoff', c)
T = cluster(Z, 'cutoff', c, 'depth', d)
T = cluster(Z, 'cutoff', c, 'criterion', criterion)
T = cluster(Z, 'maxclust', n)
```

### Description

`T = cluster(Z, 'cutoff', c)` constructs clusters from the agglomerative hierarchical cluster tree, `Z`, as generated by the `linkage` function. `Z` is a matrix of size  $(m - 1)$ -by-3, where  $m$  is the number of observations in the original data. `c` is a threshold for cutting `Z` into clusters. Clusters are formed when a node and all of its subnodes have inconsistent value less than `c`. All leaves at or below the node are grouped into a cluster. `t` is a vector of size  $m$  containing the cluster assignments of each observation.

If `c` is a vector, `T` is a matrix of cluster assignments with one column per cutoff value.

`T = cluster(Z, 'cutoff', c, 'depth', d)` evaluates inconsistent values by looking to a depth `d` below each node. The default depth is 2.

`T = cluster(Z, 'cutoff', c, 'criterion', criterion)` uses the specified criterion for forming clusters, where `criterion` is `'inconsistent'` (default) or `'distance'`. The `'distance'` criterion uses the distance between the two subnodes merged at a node to measure node height. All leaves at or below a node with height less than `c` are grouped into a cluster.

`T = cluster(Z, 'maxclust', n)` constructs a maximum of `n` clusters using the `'distance'` criterion. `cluster` finds the smallest height at which a horizontal cut through the tree leaves `n` or fewer clusters.

If  $n$  is a vector,  $T$  is a matrix of cluster assignments with one column per maximum value.

## Examples

### Compare Cluster Assignments to Clusters

Load the sample data.

```
load fisheriris
```

Compute four clusters of the Fisher iris data using Ward linkage and ignoring species information.

```
Z = linkage(meas, 'ward', 'euclidean');  
c = cluster(Z, 'maxclust', 4);
```

See how the cluster assignments correspond to the three species.

```
crosstab(c, species)
```

```
ans =
```

```
    0    25     1  
    0    24    14  
    0     1    35  
   50     0     0
```

Display the first five rows of  $Z$ .

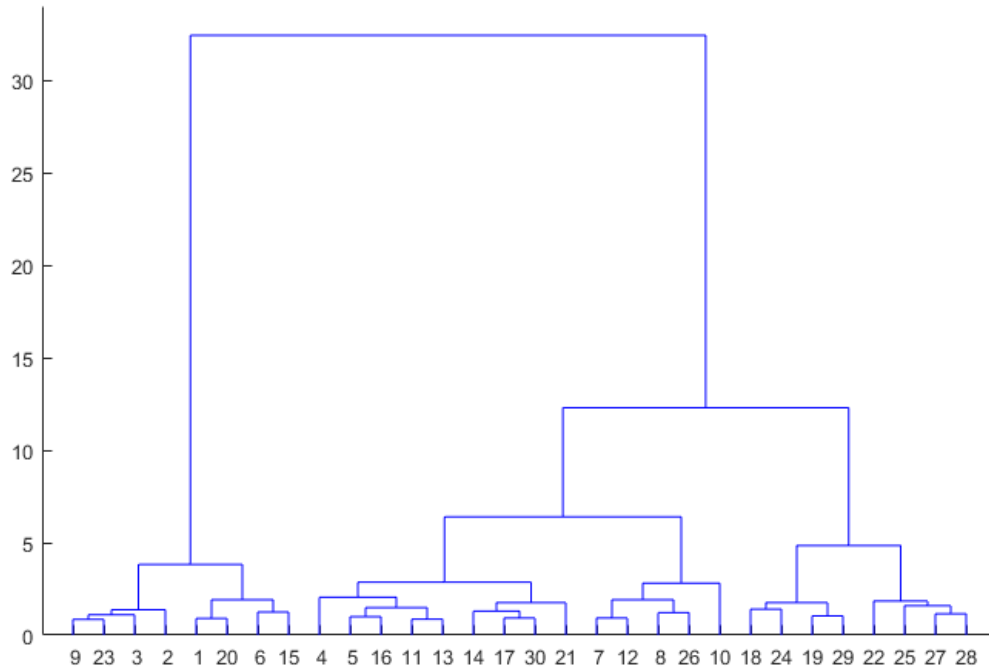
```
firstfive = Z(1:5, :)
```

```
firstfive =
```

```
102.0000 143.0000         0  
  8.0000  40.0000  0.1000  
  1.0000  18.0000  0.1000  
 10.0000  35.0000  0.1000  
129.0000 133.0000  0.1000
```

Create a dendrogram plot of  $Z$ .

```
dendrogram(Z)
```



### Cluster Data and Plot the Result

Randomly generate the sample data with 20000 observations.

```
rng default; % For reproducibility
X = rand(20000,3);
```

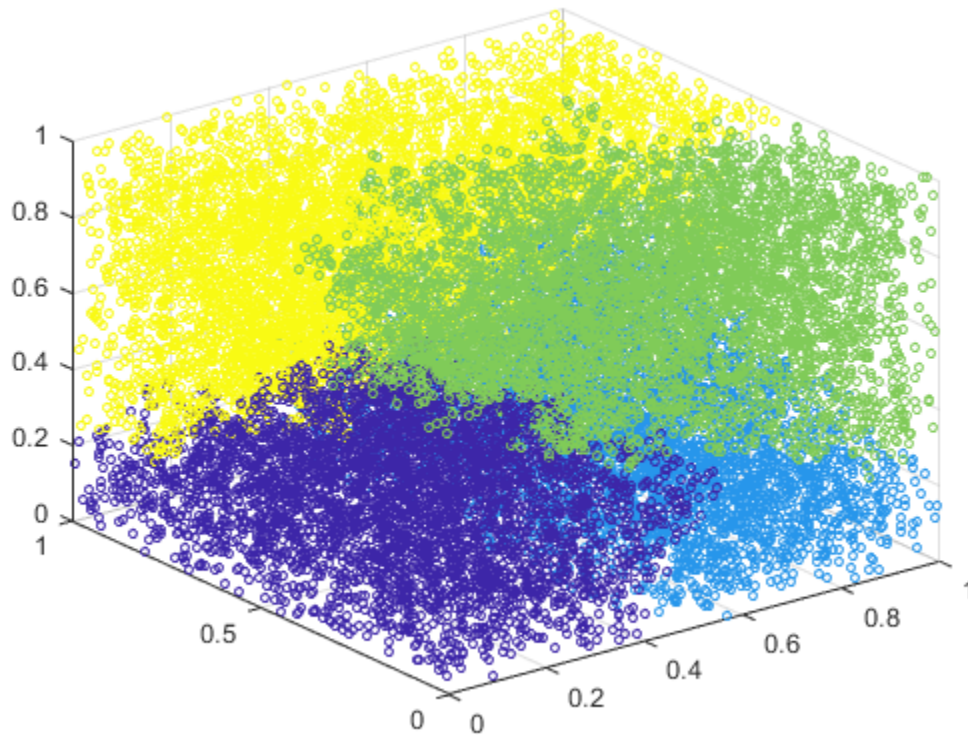
Create a hierarchical cluster tree using Ward's linkage.

```
Z = linkage(X, 'ward', 'euclidean', 'savememory', 'on');
```

If you set `savememory` to `'off'`, you can get an out-of-memory error if your machine doesn't have enough memory to hold the distance matrix.

Cluster data into four groups and plot the result.

```
c = cluster(Z, 'maxclust', 4);  
scatter3(X(:,1), X(:,2), X(:,3), 10, c)
```



## See Also

[clusterdata](#) | [cophenet](#) | [inconsistent](#) | [linkage](#) | [pdist](#)

Introduced before R2006a

---

# cluster

**Class:** `gmdistribution`

Construct clusters from Gaussian mixture distribution

## Syntax

```
idx = cluster(obj,X)
[idx,nlogl] = cluster(obj,X)
[idx,nlogl,P] = cluster(obj,X)
[idx,nlogl,P,logpdf] = cluster(obj,X)
[idx,nlogl,P,logpdf,M] = cluster(obj,X)
```

## Description

`idx = cluster(obj,X)` partitions data in the  $n$ -by- $d$  matrix  $X$ , where  $n$  is the number of observations and  $d$  is the dimension of the data, into  $k$  clusters determined by the  $k$  components of the Gaussian mixture distribution defined by `obj`. `obj` is an object created by `gmdistribution` or `fitgmdist`. `idx` is an  $n$ -by-1 vector, where `idx(I)` is the cluster index of observation  $I$ . The cluster index gives the component with the largest posterior probability for the observation, weighted by the component probability.

---

**Note** The data in  $X$  is typically the same as the data used to create the Gaussian mixture distribution defined by `obj`. Clustering with `cluster` is treated as a separate step, apart from density estimation. For `cluster` to provide meaningful clustering with new data,  $X$  should come from the same population as the data used to create `obj`.

---

`cluster` treats NaN values as missing data. Rows of  $X$  with NaN values are excluded from the partition.

`[idx,nlogl] = cluster(obj,X)` also returns `nlogl`, the negative log-likelihood of the data.

`[idx,nlogl,P] = cluster(obj,X)` also returns the posterior probabilities of each component for each observation in the  $n$ -by- $k$  matrix  $P$ .  $P(I, J)$  is the probability of component  $J$  given observation  $I$ .

`[idx,nlogl,P,logpdf]` = `cluster(obj,X)` also returns the  $n$ -by-1 vector `logpdf` containing the logarithm of the estimated probability density function for each observation. The density estimate for observation  $I$  is a sum over all components of the component density at  $I$  times the component probability.

`[idx,nlogl,P,logpdf,M]` = `cluster(obj,X)` also returns an  $n$ -by- $k$  matrix  $M$  containing Mahalanobis distances in squared units.  $M(I, J)$  is the Mahalanobis distance of observation  $I$  from the mean of component  $J$ .

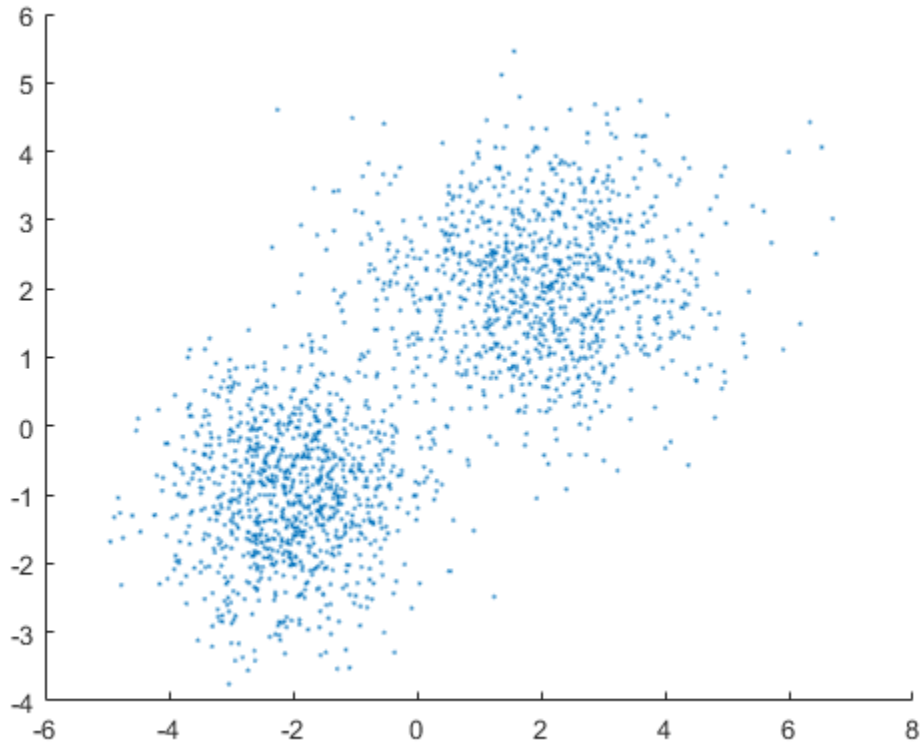
## Examples

### Cluster Data from a Gaussian Mixture Distribution

Generate data from a mixture of two bivariate Gaussian distributions using the `mvnrnd` function

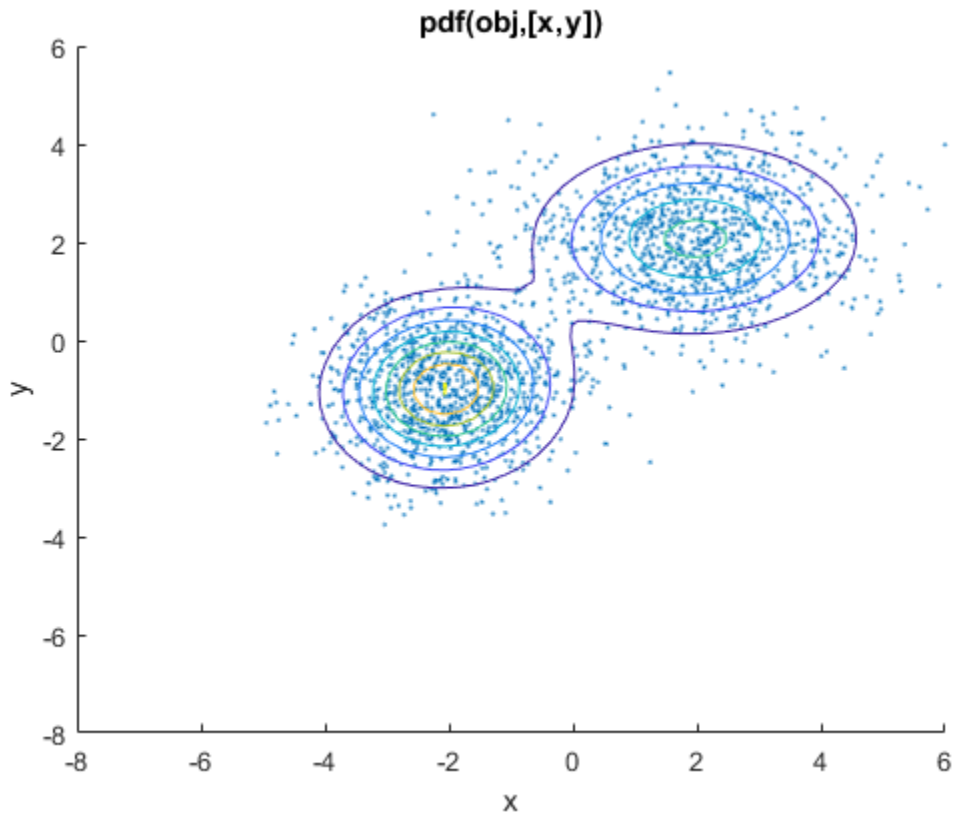
```
MU1 = [2 2];
SIGMA1 = [2 0; 0 1];
MU2 = [-2 -1];
SIGMA2 = [1 0; 0 1];
rng(1); % For reproducibility
X = [mvnrnd(MU1,SIGMA1,1000);mvnrnd(MU2,SIGMA2,1000)];

scatter(X(:,1),X(:,2),10, '.')
hold on
```



Fit a two-component Gaussian mixture model.

```
obj = fitgmdist(X,2);  
h = ezcontour(@(x,y)pdf(obj,[x y]),[-8 6],[-8 6]);
```

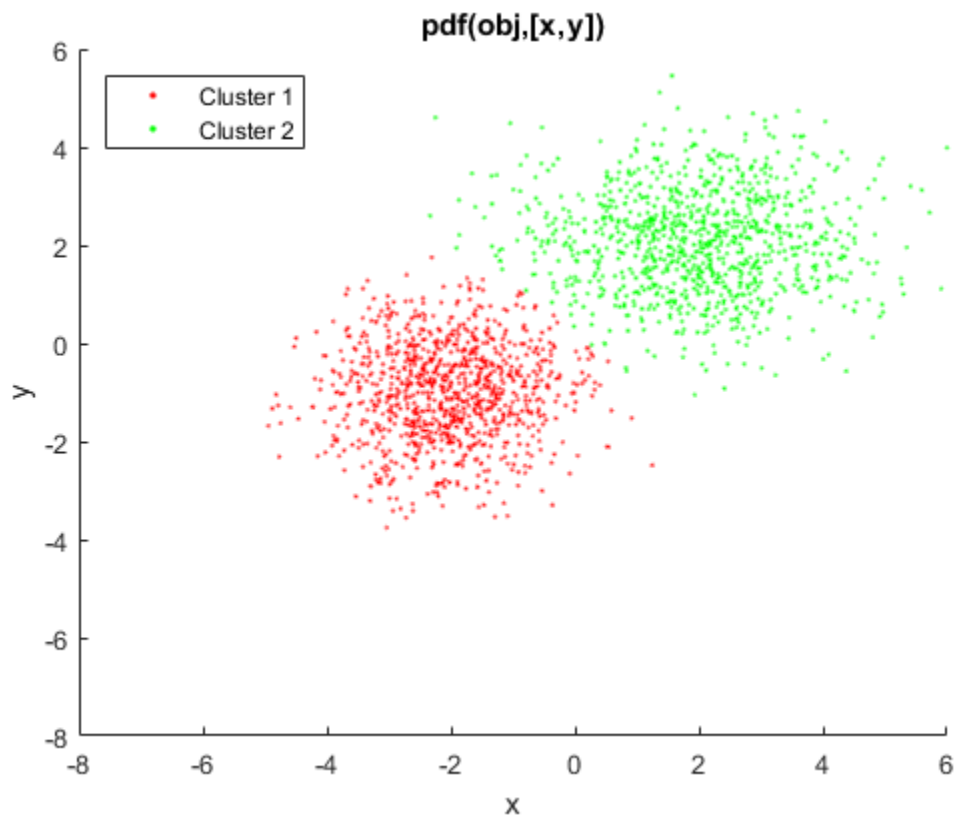


Use the fit to cluster the data.

```
idx = cluster(obj,X);
cluster1 = X(idx == 1,:);
cluster2 = X(idx == 2,:);

delete(h)
h1 = scatter(cluster1(:,1),cluster1(:,2),10,'r. ');
h2 = scatter(cluster2(:,1),cluster2(:,2),10,'g. ');
legend([h1 h2], 'Cluster 1', 'Cluster 2', 'Location', 'NW')
```





## See Also

[fitgmdist](#) | [gmdistribution](#) | [mahal](#) | [posterior](#)

## clustering.evaluation.ClusterCriterion class

**Package:** clustering.evaluation

Clustering evaluation object

### Description

Create a clustering evaluation object using `evalclusters`.

### Properties

#### **ClusteringFunction**

Clustering algorithm used to cluster the input data, stored as a valid clustering algorithm name or function handle. If the clustering solutions are provided in the input, `ClusteringFunction` is empty.

#### **CriterionName**

Name of the criterion used for clustering evaluation, stored as a valid criterion name.

#### **CriterionValues**

Criterion values corresponding to each proposed number of clusters in `InspectedK`, stored as a vector of numerical values.

#### **InspectedK**

List of the number of proposed clusters for which to compute criterion values, stored as a vector of positive integer values.

#### **Missing**

Logical flag for excluded data, stored as a column vector of logical values. If `Missing` equals `true`, then the corresponding value in the data matrix  $x$  is not used in the clustering solution.

**NumObservations**

Number of observations in the data matrix  $X$ , minus the number of missing (NaN) values in  $X$ , stored as a positive integer value.

**OptimalK**

Optimal number of clusters, stored as a positive integer value.

**OptimalY**

Optimal clustering solution corresponding to `OptimalK`, stored as a column vector of positive integer values. If the clustering solutions are provided in the input, `OptimalY` is empty.

**x**

Data used for clustering, stored as a matrix of numerical values.

## Methods

<code>addK</code>	Evaluate additional numbers of clusters
<code>plot</code>	Plot clustering evaluation object criterion values
<code>compact</code>	Compact clustering evaluation object

## See Also

`clustering.evaluation.CalinskiHarabaszEvaluation` |  
`clustering.evaluation.DaviesBouldinEvaluation` |  
`clustering.evaluation.GapEvaluation` |  
`clustering.evaluation.SilhouetteEvaluation` | `evalclusters`

## Topics

Class Attributes (MATLAB)  
Property Attributes (MATLAB)

## clusterdata

Agglomerative clusters from data

### Syntax

```
T = clusterdata(X,cutoff)
T = clusterdata(X,Name,Value)
```

### Description

`T = clusterdata(X,cutoff)` returns the cluster indices (`T`) for each observation (row) of the data (`X`) while adhering to a threshold for cutting the hierarchical tree (`cutoff`).

`T = clusterdata(X,Name,Value)` clusters with additional options specified by one or more `Name, Value` pair arguments.

### Input Arguments

#### **x**

Matrix with two or more rows. The rows represent observations, the columns represent categories or dimensions.

#### **cutoff**

When  $0 < \text{cutoff} < 2$ , `clusterdata` forms clusters when inconsistent values are greater than `cutoff` (see `inconsistent`). When `cutoff` is an integer  $\geq 2$ , `clusterdata` interprets `cutoff` as the maximum number of clusters to keep in the hierarchical tree generated by `linkage`.

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single

quotes (' '). You can specify several name and value pair arguments in any order as Name1, Value1, ..., NameN, ValueN.

### **criterion**

Either 'inconsistent' or 'distance'.

### **cutoff**

Cutoff for inconsistent or distance measure, a positive scalar. When  $0 < \text{cutoff} < 2$ , clusterdata forms clusters when inconsistent values are greater than cutoff (see inconsistent). When cutoff is an integer  $\geq 2$ , clusterdata interprets cutoff as the maximum number of clusters to keep in the hierarchical tree generated by linkage.

### **depth**

Depth for computing inconsistent values, a positive integer.

### **distance**

Any of the distance metric names allowed by pdist:

Value	Description
'euclidean'	Euclidean distance (default).
'squaredeuclidean'	Squared Euclidean distance. (This option is provided for efficiency only. It does not satisfy the triangle inequality.)
'seuclidean'	Standardized Euclidean distance. Each coordinate difference between observations is scaled by dividing by the corresponding element of the standard deviation, $S = \text{nanstd}(X)$ .
'mahalanobis'	Mahalanobis distance using the sample covariance of X, $C = \text{nancov}(X)$ .
'cityblock'	City block distance.
'minkowski'	Minkowski distance. The default exponent is 2. To use a different exponent P, specify P after 'minkowski', where P is a positive scalar value: 'minkowski', P
'chebychev'	Chebychev distance (maximum coordinate difference).

Value	Description
'cosine'	One minus the cosine of the included angle between points (treated as vectors).
'correlation'	One minus the sample correlation between points (treated as sequences of values).
'hamming'	Hamming distance, which is the percentage of coordinates that differ.
'jaccard'	One minus the Jaccard coefficient, which is the percentage of nonzero coordinates that differ.
'spearman'	One minus the sample Spearman's rank correlation between observations (treated as sequences of values).
@ <i>distfun</i>	<p>Custom distance function handle. A distance function has the form</p> <pre>function D2 = DISTFUN(ZI,ZJ) % calculation of distance ...</pre> <p>where</p> <ul style="list-style-type: none"> <li>• ZI is a 1-by-n vector containing a single observation.</li> <li>• ZJ is an m2-by-n matrix containing multiple observations. <i>distfun</i> must accept a matrix XJ with an arbitrary number of observations.</li> <li>• D2 is an m2-by-1 vector of distances, and D2(k) is the distance between observations ZI and ZJ(k, :).</li> </ul> <p>If your data is not sparse, generally it is faster to use a built-in distance than to use a function handle.</p>

For more information on these distance metrics, see “Distance Metrics” on page 32-4665.

### linkage

Any of the linkage methods allowed by the `linkage` function:

- 'average'
- 'centroid'

- 'complete'
- 'median'
- 'single'
- 'ward'
- 'weighted'

For details, see the definitions in the `linkage` function reference page.

**maxclust**

Maximum number of clusters to form, a positive integer.

**savememory**

Either 'on' or 'off'. When applicable, the 'on' setting causes `clusterdata` to construct clusters without computing the distance matrix. `savememory` is applicable when:

- 'linkage' is 'centroid', 'median', or 'ward'
- 'distance' is 'euclidean' (default)

When `savememory` is 'on', linkage run time is proportional to the number of dimensions (number of columns of  $X$ ). When `savememory` is 'off', linkage memory requirement is proportional to  $N^2$ , where  $N$  is the number of observations. So choosing the best (least-time) setting for `savememory` depends on the problem dimensions, number of observations, and available memory. The default `savememory` setting is a rough approximation of an optimal setting.

**Default:** 'on' when  $X$  has 20 columns or fewer, or the computer does not have enough memory to store the distance matrix; otherwise 'off'

## Output Arguments

**T**

$T$  is a vector of size  $m$  containing a cluster number for each observation.

- When  $0 < \text{cutoff} < 2$ ,  $T = \text{clusterdata}(X, \text{cutoff})$  is equivalent to:

```
Y = pdist(X, 'euclid');
Z = linkage(Y, 'single');
T = cluster(Z, 'cutoff', cutoff);
```

- When `cutoff` is an integer  $\geq 2$ , `T = clusterdata(X, cutoff)` is equivalent to:

```
Y = pdist(X, 'euclid');
Z = linkage(Y, 'single');
T = cluster(Z, 'maxclust', cutoff);
```

## Examples

### Create Hierarchical Cluster Tree From Sample Data

This example shows how to create a hierarchical cluster tree from sample data, and visualize the clusters using a 3-dimensional scatter plot.

Generate sample data matrices containing random numbers from the standard uniform distribution.

```
rng('default'); % For reproducibility
X = [gallery('uniformdata', [10 3], 12); ...
     gallery('uniformdata', [10 3], 13)+1.2; ...
     gallery('uniformdata', [10 3], 14)+2.5];
```

Compute the distances between items and create a hierarchical cluster tree from the sample data. List all of the items in cluster 2.

```
T = clusterdata(X, 'Maxclust', 3);
find(T==2)
```

```
ans =
```

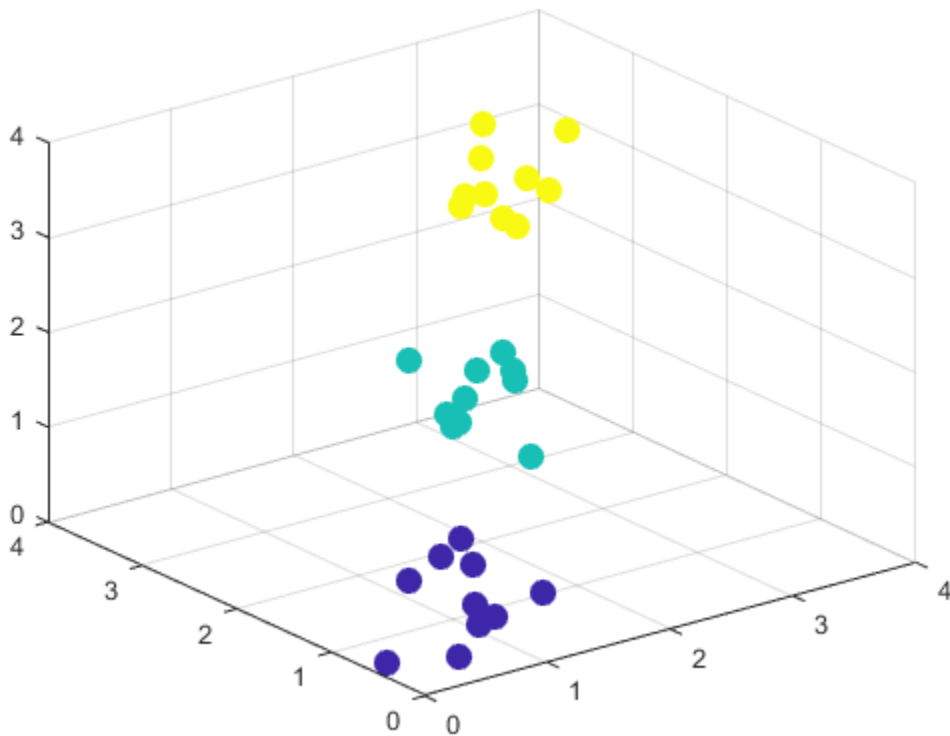
```
11
12
13
14
15
16
17
18
```



19  
20

Plot the data with each cluster shown in a different color.

```
scatter3(X(:,1),X(:,2),X(:,3),100,T,'filled')
```



### Create Hierarchical Cluster Tree Using Ward's Linkage

This example shows how to create a hierarchical cluster tree using Ward's linkage, and visualize the clusters using a 3-dimensional scatter plot.

Create a 20,000-by-3 matrix of sample data generated from the standard uniform distribution.

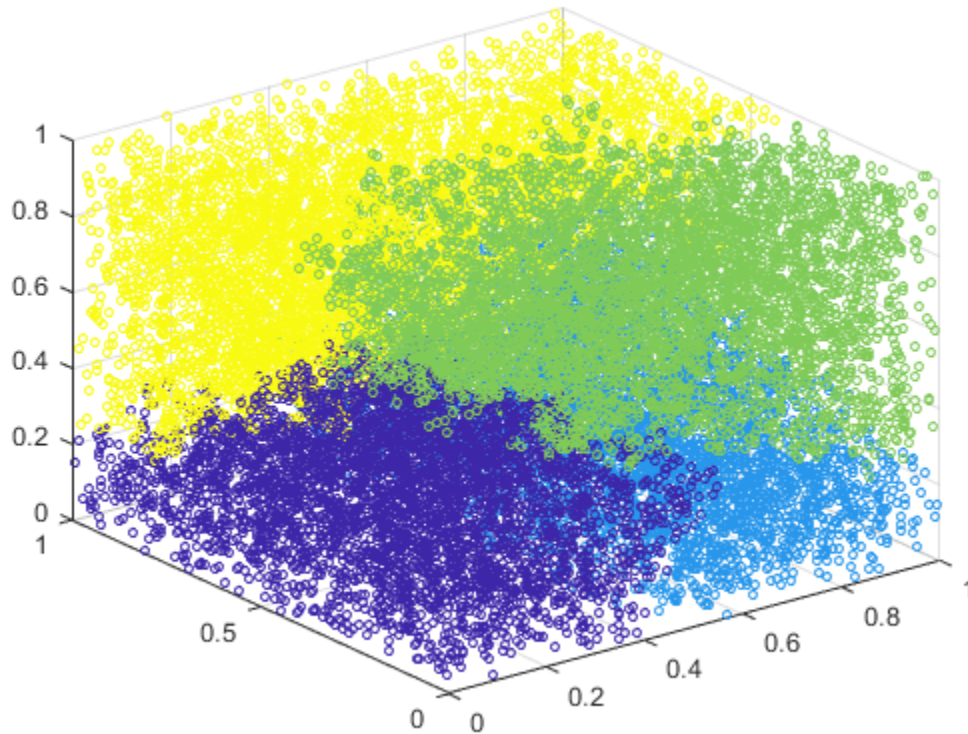
```
rng default; % For reproducibility
X = rand(20000,3);
```

Create a hierarchical cluster tree from the sample data using Ward's linkage. Set 'savememory' to 'on' to construct clusters without computing the distance matrix.

```
c = clusterdata(X,'linkage','ward','savememory','on','maxclust',4);
```

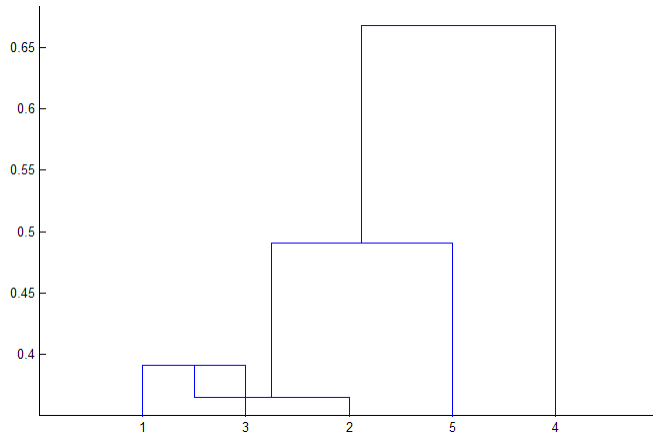
Plot the data with each cluster shown in a different color.

```
scatter3(X(:,1),X(:,2),X(:,3),10,c)
```



## Tips

- The centroid and median methods can produce a cluster tree that is not monotonic. This occurs when the distance from the union of two clusters,  $r$  and  $s$ , to a third cluster is less than the distance between  $r$  and  $s$ . In this case, in a dendrogram drawn with the default orientation, the path from a leaf to the root node takes some downward steps. To avoid this, use another method. The following image shows a nonmonotonic cluster tree.



In this case, cluster 1 and cluster 3 are joined into a new cluster, while the distance between this new cluster and cluster 2 is less than the distance between cluster 1 and cluster 3. This leads to a nonmonotonic tree.

- You can provide the output `T` to other functions including `dendrogram` to display the tree, `cluster` to assign points to clusters, `inconsistent` to compute inconsistent measures, and `cophenet` to compute the cophenetic correlation coefficient.

## See Also

`cluster` | `inconsistent` | `kmeans` | `linkage` | `pdist`

Introduced before R2006a

# cmdscale

Classical multidimensional scaling

## Syntax

```
Y = cmdscale(D)
[Y,e] = cmdscale(D)
[Y,e] = cmdscale(D,p)
```

## Description

`Y = cmdscale(D)` takes an  $n$ -by- $n$  distance matrix  $D$ , and returns an  $n$ -by- $p$  configuration matrix  $Y$ . Rows of  $Y$  are the coordinates of  $n$  points in  $p$ -dimensional space for some  $p < n$ . When  $D$  is a Euclidean distance matrix, the distances between those points are given by  $D$ .  $p$  is the dimension of the smallest space in which the  $n$  points whose inter-point distances are given by  $D$  can be embedded.

`[Y,e] = cmdscale(D)` also returns the eigenvalues of  $Y*Y'$ . When  $D$  is Euclidean, the first  $p$  elements of  $e$  are positive, the rest zero. If the first  $k$  elements of  $e$  are much larger than the remaining  $(n-k)$ , then you can use the first  $k$  columns of  $Y$  as  $k$ -dimensional points whose inter-point distances approximate  $D$ . This can provide a useful dimension reduction for visualization, e.g., for  $k = 2$ .

$D$  need not be a Euclidean distance matrix. If it is non-Euclidean or a more general dissimilarity matrix, then some elements of  $e$  are negative, and `cmdscale` chooses  $p$  as the number of positive eigenvalues. In this case, the reduction to  $p$  or fewer dimensions provides a reasonable approximation to  $D$  only if the negative elements of  $e$  are small in magnitude.

`[Y,e] = cmdscale(D,p)` also accepts a positive integer  $p$  between 1 and  $n$ .  $p$  specifies the dimensionality of the desired embedding  $Y$ . If a  $p$  dimensional embedding is possible, then  $Y$  will be of size  $n$ -by- $p$  and  $e$  will be of size  $p$ -by-1. If only a  $q$  dimensional embedding with  $q < p$  is possible, then  $Y$  will be of size  $n$ -by- $q$  and  $e$  will be of size  $p$ -by-1. Specifying  $p$  may reduce the computational burden when  $n$  is very large.

You can specify `D` as either a full dissimilarity matrix, or in upper triangle vector form such as is output by `pdist`. A full dissimilarity matrix must be real and symmetric, and have zeros along the diagonal and positive elements everywhere else. A dissimilarity matrix in upper triangle form must have real, positive entries. You can also specify `D` as a full similarity matrix, with ones along the diagonal and all other elements less than one. `cmdscale` transforms a similarity matrix to a dissimilarity matrix in such a way that distances between the points returned in `Y` equal or approximate  $\sqrt{1-D}$ . To use a different transformation, you must transform the similarities prior to calling `cmdscale`.

## Examples

Generate some points in 4-D space, but close to 3-D space, then reduce them to distances only.

```
X = [normrnd(0,1,10,3) normrnd(0,.1,10,1)];
D = pdist(X,'euclidean');
```

Find a configuration with those inter-point distances.

```
[Y,e] = cmdscale(D);

% Four, but fourth one small
dim = sum(e > eps^(3/4))

% Poor reconstruction
maxerr2 = max(abs(pdist(X)-pdist(Y(:,1:2))))

% Good reconstruction
maxerr3 = max(abs(pdist(X)-pdist(Y(:,1:3))))

% Exact reconstruction
maxerr4 = max(abs(pdist(X)-pdist(Y)))

% D is now non-Euclidean
D = pdist(X,'cityblock');
[Y,e] = cmdscale(D);

% One is large negative
min(e)

% Poor reconstruction
maxerr = max(abs(pdist(X)-pdist(Y)))
```

## References

- [1] Seber, G. A. F. *Multivariate Observations*. Hoboken, NJ: John Wiley & Sons, Inc., 1984.

## See Also

mdscale | pdist | procrustes

**Introduced before R2006a**

## coefCI

**Class:** CompactGeneralizedLinearModel

Confidence intervals of coefficient estimates of generalized linear model

## Syntax

```
ci = coefCI mdl  
ci = coefCI mdl, alpha
```

## Description

`ci = coefCI(mdl)` returns confidence intervals for the coefficients in `mdl`.

`ci = coefCI(mdl, alpha)` returns confidence intervals with confidence level  $1 - \alpha$ .

## Input Arguments

**mdl**

Generalized linear model, specified as a full `GeneralizedLinearModel` object constructed using `fitglm` or `stepwiseglm`, or a compacted `CompactGeneralizedLinearModel` object constructed using `compact`.

**alpha**

Scalar from 0 to 1, the probability that the confidence interval does not contain the true value.

**Default:** 0.05



## Output Arguments

**ci**

$k$ -by-2 matrix of confidence intervals. The  $j$ th row of `ci` is the confidence interval of coefficient  $j$  of `mdl`. The name of coefficient  $j$  of `mdl` is in `mdl.CoefficientNames`.

## Examples

### Confidence Interval for Coefficients of a Generalized Linear Model

Find confidence intervals for the coefficients of a fitted generalized nonlinear model.

Generate artificial data for the model using Poisson random numbers with two underlying predictors  $X(1)$  and  $X(2)$ .

```
rng('default') % for reproducibility
rndvars = randn(100,2);
X = [2+rndvars(:,1),rndvars(:,2)];
mu = exp(1 + X*[1;2]);
y = poissrnd(mu);
```

Create a generalized linear regression model of Poisson data.

```
mdl = fitglm(X,y,...
    'y ~ x1 + x2','distr','poisson')
```

```
mdl =
```

Generalized linear regression model:

```
log(y) ~ 1 + x1 + x2
Distribution = Poisson
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	1.0405	0.022122	47.034	0
x1	0.9968	0.003362	296.49	0

```
x2          1.987      0.0063433    313.24    0
```

```
100 observations, 97 error degrees of freedom
Dispersion: 1
Chi^2-statistic vs. constant model: 2.95e+05, p-value = 0
```

Find 95% (default) confidence intervals on the coefficients of the model.

```
ci = coefCI(mdl)
```

```
ci =
```

```
  0.9966    1.0844
  0.9901    1.0035
  1.9744    1.9996
```

Find 99% confidence intervals on the coefficients.

```
alpha = .01;
ci = coefCI(mdl, alpha)
```

```
ci =
```

```
  0.9824    1.0986
  0.9880    1.0056
  1.9703    2.0036
```

- “Generalized Linear Model Workflow” on page 12-41

## Definitions

### Confidence Interval

Assume that model assumptions hold (data comes from a generalized linear model represented by the formula `mdl`. Formula and the specified link function, and with observations that are independent conditional on the predictor values). Then row  $j$  of the confidence interval matrix `ci` gives a confidence interval  $[ci(j, 1), ci(j, 2)]$

computed such that, with repeated experimentation, a proportion  $1 - \alpha$  of the intervals will contain the true value of the coefficient.

## See Also

`CompactGeneralizedLinearModel` | `GeneralizedLinearModel`

## Topics

“Generalized Linear Model Workflow” on page 12-41

## coefCI

**Class:** GeneralizedLinearMixedModel

Confidence intervals for coefficients of generalized linear mixed-effects model

## Syntax

```
feCI = coefCI(glme)
feCI = coefCI(glme, Name, Value)
[feCI, reCI] = coefCI(____)
```

## Description

`feCI = coefCI(glme)` returns the 95% confidence intervals for the fixed-effects coefficients in the generalized linear mixed-effects model `glme`.

`feCI = coefCI(glme, Name, Value)` returns the confidence intervals using additional options specified by one or more `Name, Value` pair arguments. For example, you can specify a different confidence level or the method used to compute the approximate degrees of freedom.

`[feCI, reCI] = coefCI(____)` also returns the confidence intervals for the random-effects coefficients using any of the previous syntaxes.

## Input Arguments

**glme** — Generalized linear mixed-effects model

GeneralizedLinearMixedModel object

Generalized linear mixed-effects model, specified as a `GeneralizedLinearMixedModel` object. For properties and methods of this object, see `GeneralizedLinearMixedModel`.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### **Alpha** — Confidence level

0.05 (default) | scalar value in the range [0,1]

Confidence level, specified as the comma-separated pair consisting of 'Alpha' and a scalar value in the range [0,1]. For a value  $\alpha$ , the confidence level is  $100 \times (1 - \alpha)\%$ .

For example, for 99% confidence intervals, you can specify the confidence level as follows.

Example: 'Alpha', 0.01

Data Types: single | double

### **DFMethod** — Method for computing approximate degrees of freedom

'residual' (default) | 'none'

Method for computing approximate degrees of freedom, specified as the comma-separated pair consisting of 'DFMethod' and one of the following.

Value	Description
'residual'	The degrees of freedom are assumed to be constant and equal to $n - p$ , where $n$ is the number of observations and $p$ is the number of fixed effects.
'none'	All degrees of freedom are set to infinity.

Example: 'DFMethod', 'none'

## Output Arguments

### **feCI** — Fixed-effects confidence intervals

$p$ -by-2 matrix

Fixed-effects confidence intervals, returned as a  $p$ -by-2 matrix. `feCI` contains the confidence limits that correspond to the  $p$ -by-1 fixed-effects vector returned by the

`fixedEffects` method. The first column of `feCI` contains the lower confidence limits and the second column contains the upper confidence limits.

When fitting a GLME model using `fitglm` and one of the maximum likelihood fit methods ('Laplace' or 'ApproximateLaplace'):

- If you specify the 'CovarianceMethod' name-value pair argument as 'conditional', then the confidence intervals are conditional on the estimated covariance parameters.
- If you specify the 'CovarianceMethod' name-value pair argument as 'JointHessian', then the confidence intervals account for the uncertainty in the estimated covariance parameters.

When fitting a GLME model using `fitglm` and one of the pseudo likelihood fit methods ('MPL' or 'REMP'), `coefci` uses the fitted linear mixed effects model from the final pseudo likelihood iteration to compute confidence intervals on the fixed effects.

### **reCI — Random-effects confidence intervals**

*q*-by-2 matrix

Random-effects confidence intervals, returned as a *q*-by-2 matrix. `reCI` contains the confidence limits corresponding to the *q*-by-1 random-effects vector `B` returned by the `randomEffects` method. The first column of `reCI` contains the lower confidence limits, and the second column contains the upper confidence limits.

When fitting a GLME model using `fitglm` and one of the maximum likelihood fit methods ('Laplace' or 'ApproximateLaplace'), `coefCI` computes the confidence intervals using the conditional mean squared error of prediction (CMSEP) approach conditional on the estimated covariance parameters and the observed response. Alternatively, you can interpret the confidence intervals from `coefCI` as approximate Bayesian credible intervals conditional on the estimated covariance parameters and the observed response.

When fitting a GLME model using `fitglm` and one of the pseudo likelihood fit methods ('MPL' or 'REMP'), `coefci` uses the fitted linear mixed effects model from the final pseudo likelihood iteration to compute confidence intervals on the random effects.

## Examples

## 95% Confidence Intervals for Fixed Effects

Load the sample data.

```
load mfr
```

This simulated data is from a manufacturing company that operates 50 factories across the world, with each factory running a batch process to create a finished product. The company wants to decrease the number of defects in each batch, so it developed a new manufacturing process. To test the effectiveness of the new process, the company selected 20 of its factories at random to participate in an experiment: Ten factories implemented the new process, while the other ten continued to run the old process. In each of the 20 factories, the company ran five batches (for a total of 100 batches) and recorded the following data:

- Flag to indicate whether the batch used the new process (`newprocess`)
- Processing time for each batch, in hours (`time`)
- Temperature of the batch, in degrees Celsius (`temp`)
- Categorical variable indicating the supplier (A, B, or C) of the chemical used in the batch (`supplier`)
- Number of defects in the batch (`defects`)

The data also includes `time_dev` and `temp_dev`, which represent the absolute deviation of time and temperature, respectively, from the process standard of 3 hours at 20 degrees Celsius.

Fit a generalized linear mixed-effects model using `newprocess`, `time_dev`, `temp_dev`, and `supplier` as fixed-effects predictors. Include a random-effects term for intercept grouped by `factory`, to account for quality differences that might exist due to factory-specific variations. The response variable `defects` has a Poisson distribution, and the appropriate link function for this model is `log`. Use the Laplace fit method to estimate the coefficients. Specify the dummy variable encoding as `'effects'`, so the dummy variable coefficients sum to 0.

The number of defects can be modeled using a Poisson distribution

$$\text{defects}_{ij} \sim \text{Poisson}(\mu_{ij})$$

This corresponds to the generalized linear mixed-effects model

$$\log(\mu_{ij}) = \beta_0 + \beta_1 \text{newprocess}_{ij} + \beta_2 \text{time.dev}_{ij} + \beta_3 \text{temp.dev}_{ij} + \beta_4 \text{supplier.C}_{ij} + \beta_5 \text{supplier.B}_{ij} + b_i,$$

where

- `defectsij` is the number of defects observed in the batch produced by factory  $i$  during batch  $j$ .
- $\mu_{ij}$  is the mean number of defects corresponding to factory  $i$  (where  $i = 1, 2, \dots, 20$ ) during batch  $j$  (where  $j = 1, 2, \dots, 5$ ).
- `newprocessij`, `time_devij`, and `temp_devij` are the measurements for each variable that correspond to factory  $i$  during batch  $j$ . For example, `newprocessij` indicates whether the batch produced by factory  $i$  during batch  $j$  used the new process.
- `supplier_Cij` and `supplier_Bij` are dummy variables that use effects (sum-to-zero) coding to indicate whether company C or B, respectively, supplied the process chemicals for the batch produced by factory  $i$  during batch  $j$ .
- $b_i \sim N(0, \sigma_b^2)$  is a random-effects intercept for each factory  $i$  that accounts for factory-specific variation in quality.

```
glme = fitglme(mfr, 'defects ~ 1 + newprocess + time_dev + temp_dev + supplier + (1|fact
```

Use `fixedEffects` to display the estimates and names of the fixed-effects coefficients in `glme`.

```
[beta, betanames] = fixedEffects(glme)
```

```
beta =
```

```
    1.4689
   -0.3677
   -0.0945
   -0.2832
   -0.0719
    0.0711
```

```
betanames =
```

```
6x1 table
```

```
    Name
```

---



```
'(Intercept)'  
'newprocess'  
'time_dev'  
'temp_dev'  
'supplier_C'  
'supplier_B'
```

Each row of `beta` contains the estimated value for the coefficient named in the corresponding row of `betanames`. For example, the value  $-0.0945$  in row 3 of `beta` is the estimated coefficient for the predictor variable `time_dev`.

Compute the 95% confidence intervals for the fixed-effects coefficients.

```
feCI = coefCI(glme)
```

```
feCI =
```

```
  1.1515    1.7864  
 -0.7202   -0.0151  
 -1.7395    1.5505  
 -2.1926    1.6263  
 -0.2268    0.0831  
 -0.0826    0.2247
```

Column 1 of `feCI` contains the lower bound of the 95% confidence interval. Column 2 contains the upper bound. Row 1 corresponds to the intercept term. Rows 2, 3, and 4 correspond to `newprocess`, `time_dev`, and `temp_dev`, respectively. Rows 5 and 6 correspond to the indicator variables `supplier_C` and `supplier_B`, respectively. For example, the 95% confidence interval for the coefficient for `time_dev` is  $[-1.7395, 1.5505]$ . Some of the confidence intervals include 0, which indicates that those predictors are not significant at the 5% significance level. To obtain specific *P*-values for each fixed-effects term, use `fixedEffects`. To test significance for entire terms, use `anova`.

## 99% Confidence Intervals for Random Effects

Load the sample data.

```
load mfr
```

This simulated data is from a manufacturing company that operates 50 factories across the world, with each factory running a batch process to create a finished product. The company wants to decrease the number of defects in each batch, so it developed a new manufacturing process. To test the effectiveness of the new process, the company selected 20 of its factories at random to participate in an experiment: Ten factories implemented the new process, while the other ten continued to run the old process. In each of the 20 factories, the company ran five batches (for a total of 100 batches) and recorded the following data:

- Flag to indicate whether the batch used the new process (`newprocess`)
- Processing time for each batch, in hours (`time`)
- Temperature of the batch, in degrees Celsius (`temp`)
- Categorical variable indicating the supplier (A, B, or C) of the chemical used in the batch (`supplier`)
- Number of defects in the batch (`defects`)

The data also includes `time_dev` and `temp_dev`, which represent the absolute deviation of time and temperature, respectively, from the process standard of 3 hours at 20 degrees Celsius.

Fit a generalized linear mixed-effects model using `newprocess`, `time_dev`, `temp_dev`, and `supplier` as fixed-effects predictors. Include a random-effects intercept grouped by `factory`, to account for quality differences that might exist due to factory-specific variations. The response variable `defects` has a Poisson distribution, and the appropriate link function for this model is `log`. Use the Laplace fit method to estimate the coefficients.

The number of defects can be modeled using a Poisson distribution

$$\text{defects}_{ij} \sim \text{Poisson}(\mu_{ij})$$

This corresponds to the generalized linear mixed-effects model

$$\log(\mu_{ij}) = \beta_0 + \beta_1 \text{newprocess}_{ij} + \beta_2 \text{time\_dev}_{ij} + \beta_3 \text{temp\_dev}_{ij} + \beta_4 \text{supplier\_C}_{ij} + \beta_5 \text{supplier\_B}_{ij} + b_i,$$

where

- `defectsij` is the number of defects observed in the batch produced by factory *i* during batch *j*.

- $\mu_{ij}$  is the mean number of defects corresponding to factory  $i$  (where  $i = 1, 2, \dots, 20$ ) during batch  $j$  (where  $j = 1, 2, \dots, 5$ ).
- `newprocessij`, `time_devij`, and `temp_devij` are the measurements for each variable that correspond to factory  $i$  during batch  $j$ . For example, `newprocessij` indicates whether the batch produced by factory  $i$  during batch  $j$  used the new process.
- `supplier_Cij` and `supplier_Bij` are dummy variables that use effects (sum-to-zero) coding to indicate whether company C or B, respectively, supplied the process chemicals for the batch produced by factory  $i$  during batch  $j$ .
- $b_i \sim N(0, \sigma_b^2)$  is a random-effects intercept for each factory  $i$  that accounts for factory-specific variation in quality.

```
glme = fitglme(mfr, 'defects ~ 1 + newprocess + time_dev + temp_dev + supplier + (1|fact
```

Use `randomEffects` to compute and display the estimates of the empirical Bayes predictors (EBPs) for the random effects associated with `factory`.

```
[B, Bnames] = randomEffects(glme)
```

```
B =
```

```

    0.2913
    0.1542
   -0.2633
   -0.4257
    0.5453
   -0.1069
    0.3040
   -0.1653
   -0.1458
   -0.0816
    0.0145
    0.1771
    0.2487
    0.2115
    0.2777
   -0.2518
   -0.1351
   -0.1627
   -0.3208
```

```
0.0584
```

```
Bnames =
```

```
20x3 table
```

Group	Level	Name
'factory'	'1'	'(Intercept)'
'factory'	'2'	'(Intercept)'
'factory'	'3'	'(Intercept)'
'factory'	'4'	'(Intercept)'
'factory'	'5'	'(Intercept)'
'factory'	'6'	'(Intercept)'
'factory'	'7'	'(Intercept)'
'factory'	'8'	'(Intercept)'
'factory'	'9'	'(Intercept)'
'factory'	'10'	'(Intercept)'
'factory'	'11'	'(Intercept)'
'factory'	'12'	'(Intercept)'
'factory'	'13'	'(Intercept)'
'factory'	'14'	'(Intercept)'
'factory'	'15'	'(Intercept)'
'factory'	'16'	'(Intercept)'
'factory'	'17'	'(Intercept)'
'factory'	'18'	'(Intercept)'
'factory'	'19'	'(Intercept)'
'factory'	'20'	'(Intercept)'

Each row of `B` contains the estimated EBPs for the random-effects coefficient named in the corresponding row of `Bnames`. For example, the value `-0.2633` in row 3 of `B` is the estimated coefficient of `'(Intercept)'` for level `'3'` of `factory`.

Compute the 99% confidence intervals of the EBPs for the random effects.

```
[feCI, reCI] = coefCI(glme, 'Alpha', 0.01);  
reCI
```

```
reCI =
```

```
-0.2125    0.7951
```

-0.3510	0.6595
-0.8219	0.2954
-0.9953	0.1440
0.0730	1.0176
-0.6362	0.4224
-0.1796	0.7877
-0.7044	0.3738
-0.6795	0.3880
-0.6142	0.4509
-0.5487	0.5777
-0.3677	0.7219
-0.2908	0.7883
-0.3322	0.7551
-0.2572	0.8126
-0.8451	0.3416
-0.7214	0.4513
-0.7482	0.4228
-0.9333	0.2916
-0.5064	0.6232

Column 1 of `reCI` contains the lower bound of the 99% confidence interval. Column 2 contains the upper bound. Each row corresponds to a level of `factory`, in the order shown in `Bnames`. For example, row 3 corresponds to the coefficient of `' (Intercept) '` for level `'3'` of `factory`, which has a 99% confidence interval of `[-0.8219 , 0.2954]`. For additional statistics related to each random-effects term, use `randomEffects`.

## References

- [1] Booth, J.G., and J.P. Hobert. “Standard Errors of Prediction in Generalized Linear Mixed Models.” *Journal of the American Statistical Association*. Vol. 93, 1998, pp. 262–272.

## See Also

`GeneralizedLinearMixedModel` | `anova` | `coefTest` | `covarianceParameters` | `fixedEffects` | `randomEffects`

## coefCI

**Class:** CompactLinearModel

Confidence intervals of coefficient estimates of linear model

## Syntax

```
ci = coefCI mdl  
ci = coefCI mdl, alpha
```

## Description

`ci = coefCI(mdl)` returns confidence intervals for the coefficients in `mdl`.

`ci = coefCI(mdl, alpha)` returns confidence intervals using the confidence level  $1 - \alpha$ .

## Input Arguments

**mdl** — Linear model object

LinearModel object | CompactLinearModel object

Linear model object, specified as a full `LinearModel` object constructed using `fitlm` or `stepwiselm`, or a compacted `CompactLinearModel` object constructed using `compact`.

**alpha** — Confidence interval

0.05 (default) | numeric value in the range [0,1]

Confidence interval, specified as a numeric value in the range [0,1]. `alpha` is the probability that the confidence interval does not contain the true value.

## Output Arguments

### ci — Confidence intervals

numeric matrix

Confidence intervals, returned as a  $k$ -by-2 numeric matrix. The  $j$ th row of `ci` is the confidence interval of coefficient  $j$  of `mdl`. The name of coefficient  $j$  is stored in the `mdl` property `CoefficientNames`.

## Examples

### Fit Linear Model Using Default Confidence Intervals

Fit a linear regression model for auto mileage based on the `carbig` data. Then obtain the default 95% confidence intervals for the resulting model coefficients.

Load the data and create a table.

```
load carbig
Origin = nominal(Origin);
tbl = table(Horsepower,Weight,MPG,Origin);
```

Fit a linear regression model. Use horsepower, weight, and origin as predictor variables, and miles per gallon as the response variable.

```
modelspec = 'MPG ~ 1 + Horsepower + Weight + Origin';
mdl = fitlm(tbl,modelspec);
```

View the names of the coefficients.

```
mdl.CoefficientNames
```

```
ans =
```

```
1x9 cell array
```

```
Columns 1 through 4
```

```
{ '(Intercept)' } { 'Horsepower' } { 'Weight' } { 'Origin_France' }
```

```
Columns 5 through 7
    {'Origin_Germany'}    {'Origin_Italy'}    {'Origin_Japan'}
Columns 8 through 9
    {'Origin_Sweden'}    {'Origin_USA'}
```

Find confidence intervals for the coefficients of the model.

```
ci = coefCI mdl)
```

```
ci =
    43.3611    59.9390
   -0.0748   -0.0315
   -0.0059   -0.0037
  -17.3623   -0.3477
  -15.7503    0.7434
  -17.2091    0.0613
  -14.5106    1.8738
  -18.5820   -1.5036
  -17.3114   -0.9642
```

### Fit Linear Model with Custom Confidence Intervals

Fit a linear regression model for auto mileage based on the `carbig` data. Then obtain confidence intervals for the resulting model coefficients at the 99% level.

Load the data and create a table.

```
load carbig
Origin = nominal(Origin);
tbl = table(Horsepower,Weight,MPG,Origin);
```

Fit a linear regression model using horsepower, weight, and origin as the predictor variables, and miles per gallon as the response variable.



```
modelspec = 'MPG ~ 1 + Horsepower + Weight + Origin';
mdl = fitlm(tbl,modelspec);
```

Find 99% confidence intervals for the coefficients.

```
ci = coefCI(mdl, .01)
```

```
ci =
    40.7365    62.5635
   -0.0816   -0.0246
   -0.0062   -0.0034
  -20.0560    2.3459
  -18.3615    3.3546
  -19.9433    2.7955
  -17.1045    4.4676
  -21.2858    1.2002
  -19.8995    1.6238
```

The confidence intervals are wider than the default 5% confidence intervals.

## Definitions

### Confidence Interval

Assume that model assumptions hold (data comes from a generalized linear model represented by the formula `mdl.Formula`, and with observations that are independent conditional on the predictor values). Then row `j` of the confidence interval matrix `ci` gives a confidence interval `[ci(j,1), ci(j,2)]` computed such that, with repeated experimentation, a proportion `1 - alpha` of the intervals will contain the true value of the coefficient.

### Alternatives

You can create the intervals from the model coefficients in `mdl.Coefficients.Estimate` and an appropriate multiplier of the standard errors `sqrt(diag(mdl.CoefficientCovariance))`. The multiplier is `tinv(1-alpha/`

`2, dof`), where `level` is the confidence level, and `dof` is the degrees of freedom (number of data points minus the number of coefficients).

## See Also

`CompactLinearModel` | `LinearModel`

## Topics

“Linear Regression” on page 11-11

## coefCI

**Class:** LinearMixedModel

Confidence intervals for coefficients of linear mixed-effects model

## Syntax

```
feCI = coefCI(lme)
feCI = coefCI(lme, Name, Value)
[feCI, reCI] = coefCI(____)
```

## Description

`feCI = coefCI(lme)` returns the 95% confidence intervals for the fixed-effects coefficients in the linear mixed-effects model `lme`.

`feCI = coefCI(lme, Name, Value)` returns the 95% confidence intervals for the fixed-effects coefficients in the linear mixed-effects model `lme` with additional options specified by one or more `Name, Value` pair arguments.

For example, you can specify the confidence level or method to compute the degrees of freedom.

`[feCI, reCI] = coefCI(____)` also returns the 95% confidence intervals for the random-effects coefficients in the linear mixed-effects model `lme`.

## Input Arguments

**lme** — Linear mixed-effects model

LinearMixedModel object

Linear mixed-effects model, specified as a LinearMixedModel object constructed using `fitlme` or `fitlmematrix`.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### **Alpha** — Confidence level

0.05 (default) | scalar value in the range 0 to 1

Confidence level, specified as the comma-separated pair consisting of 'Alpha' and a scalar value in the range 0 to 1. For a value  $\alpha$ , the confidence level is  $100*(1-\alpha)\%$ .

For example, for 99% confidence intervals, you can specify the confidence level as follows.

Example: 'Alpha', 0.01

Data Types: single | double

### **DFMethod** — Method for computing approximate degrees of freedom

'residual' (default) | 'satterthwaite' | 'none'

Method for computing approximate degrees of freedom for confidence interval computation, specified as the comma-separated pair consisting of 'DFMethod' and one of the following.

'residual'

Default. The degrees of freedom are assumed to be constant and equal to  $n - p$ , where  $n$  is the number of observations and  $p$  is the number of fixed effects.

'satterthwaite'

Satterthwaite approximation.

'none'

All degrees of freedom are set to infinity.

For example, you can specify the Satterthwaite approximation as follows.

Example: 'DFMethod', 'satterthwaite'

## Output Arguments

### **feCI** — Fixed-effects confidence intervals

$p$ -by-2 matrix

Fixed-effects confidence intervals, returned as a  $p$ -by-2 matrix. `feCI` contains the confidence limits that correspond to the  $p$  fixed-effects estimates in the vector `beta` returned by the `fixedEffects` method. The first column of `feCI` has the lower confidence limits and the second column has the upper confidence limits.

### **reCI — Random-effects confidence intervals**

$q$ -by-2 matrix

Random-effects confidence intervals, returned as a  $q$ -by-2 matrix. `reCI` contains the confidence limits corresponding to the  $q$  random-effects estimates in the vector `B` returned by the `randomEffects` method. The first column of `reCI` has the lower confidence limits and the second column has the upper confidence limits.

## Examples

### 95% Confidence Intervals for Fixed-Effects Coefficients

Load the sample data.

```
load(fullfile(matlabroot, 'examples', 'stats', 'weight.mat'))
```

`weight` contains data from a longitudinal study, where 20 subjects are randomly assigned to 4 exercise programs, and their weight loss is recorded over six 2-week time periods. This is simulated data.

Store the data in a table. Define `Subject` and `Program` as categorical variables.

```
tbl = table(InitialWeight, Program, Subject, Week, y);
tbl.Subject = nominal(tbl.Subject);
tbl.Program = nominal(tbl.Program);
```

Fit a linear mixed-effects model where the initial weight, type of program, week, and the interaction between the week and type of program are the fixed effects. The intercept and week vary by subject.

```
lme = fitlme(tbl, 'y ~ InitialWeight + Program*Week + (Week|Subject)');
```

Compute the fixed-effects coefficient estimates.

```
fe = fixedEffects(lme)
```

```
fe =  
  
  0.6610  
  0.0032  
  0.3608  
 -0.0333  
  0.1132  
  0.1732  
  0.0388  
  0.0305  
  0.0331
```

The first estimate, 0.6610, corresponds to the constant term. The second row, 0.0032, and the third row, 0.3608, are estimates for the coefficient of initial weight and week, respectively. Rows four to six correspond to the indicator variables for programs B-D, and the last three rows correspond to the interaction of programs B-D and week.

Compute the 95% confidence intervals for the fixed-effects coefficients.

```
fecI = coefCI(lme)
```

```
fecI =  
  
  0.1480    1.1741  
  0.0005    0.0059  
  0.1004    0.6211  
 -0.2932    0.2267  
 -0.1471    0.3734  
  0.0395    0.3069  
 -0.1503    0.2278  
 -0.1585    0.2196  
 -0.1559    0.2221
```

Some confidence intervals include 0. To obtain specific *P*-values for each fixed-effects term, use the `fixedEffects` method. To test for entire terms use the `anova` method.

### Confidence Intervals with Specified Options

Load the sample data.

```
load carbig
```

Fit a linear mixed-effects model for miles per gallon (MPG), with fixed effects for acceleration and horsepower, and a potentially correlated random effect for intercept and acceleration grouped by model year. First, store the data in a table.

```
tbl = table(Acceleration, Horsepower, Model_Year, MPG);
```

Fit the model.

```
lme = fitlme(tbl, 'MPG ~ Acceleration + Horsepower + (Acceleration|Model_Year)');
```

Compute the fixed-effects coefficient estimates.

```
fe = fixedEffects(lme)
```

```
fe =
```

```
50.1325
-0.5833
-0.1695
```

Compute the 99% confidence intervals for fixed-effects coefficients using the residuals method to determine the degrees of freedom. This is the default method.

```
feCI = coefCI(lme, 'Alpha', 0.01)
```

```
feCI =
```

```
44.2690    55.9961
-0.9300    -0.2365
-0.1883    -0.1507
```

Compute the 99% confidence intervals for fixed-effects coefficients using the Satterthwaite approximation to compute the degrees of freedom.

```
feCI = coefCI(lme, 'Alpha', 0.01, 'DFMethod', 'satterthwaite')
```

```
feCI =
```

```
44.0949    56.1701
-0.9640    -0.2025
-0.1884    -0.1507
```

The Sattthertwaite approximation produces similar confidence intervals than the residual method.

### Compute Confidence Intervals for Random Effects

Load the sample data.

```
load(fullfile(matlabroot, 'examples', 'stats', 'shift.mat'))
```

The data shows the deviations from the target quality characteristic measured from the products that five operators manufacture during three shifts: morning, evening, and night. This is a randomized block design, where the operators are the blocks. The experiment is designed to study the impact of the time of shift on the performance. The performance measure is the deviation of the quality characteristics from the target value. This is simulated data.

Shift and Operator are nominal variables.

```
shift.Shift = nominal(shift.Shift);
shift.Operator = nominal(shift.Operator);
```

Fit a linear mixed-effects model with a random intercept grouped by operator to assess if there is significant difference in the performance according to the time of the shift.

```
lme = fitlme(shift, 'QCDev ~ Shift + (1|Operator)');
```

Compute the estimate of the BLUPs for random effects.

```
randomEffects(lme)
```

```
ans =
    0.5775
    1.1757
   -2.1715
    2.3655
```



```
-1.9472
```

Compute the 95% confidence intervals for random effects.

```
[~,reCI] = coefCI(lme)
```

```
reCI =
```

```
-1.3916    2.5467  
-0.7934    3.1449  
-4.1407   -0.2024  
 0.3964    4.3347  
-3.9164    0.0219
```

Compute the 99% confidence intervals for random effects using the residuals method to determine the degrees of freedom. This is the default method.

```
[~,reCI] = coefCI(lme, 'Alpha', 0.01)
```

```
reCI =
```

```
-2.1831    3.3382  
-1.5849    3.9364  
-4.9322    0.5891  
-0.3951    5.1261  
-4.7079    0.8134
```

Compute the 99% confidence intervals for random effects using the Satterthwaite approximation to determine the degrees of freedom.

```
[~,reCI] = coefCI(lme, 'Alpha', 0.01, 'DFMethod', 'satterthwaite')
```

```
reCI =
```

```
-2.6840    3.8390  
-2.0858    4.4372  
-5.4330    1.0900  
-0.8960    5.6270  
-5.2087    1.3142
```

The Satterthwaite approximation might produce smaller DF values than the residual method. That is why these confidence intervals are larger than the previous ones computed using the residual method.

## See Also

`LinearMixedModel` | `coefTest` | `fixedEffects` | `randomEffects`

## coefCI

**Class:** NonLinearModel

Confidence intervals of coefficient estimates of nonlinear regression model

## Syntax

```
ci = coefCI mdl  
ci = coefCI mdl, alpha
```

## Description

`ci = coefCI(mdl)` returns confidence intervals for the coefficients in `mdl`.

`ci = coefCI(mdl, alpha)` returns confidence intervals with confidence level  $1 - \alpha$ .

## Input Arguments

**mdl**

Nonlinear regression model, constructed by `fitnlm`.

**alpha**

Scalar from 0 to 1, the probability that the confidence interval does not contain the true value.

**Default:** 0.05

## Output Arguments

**ci**

k-by-2 matrix of confidence intervals. The *j*th row of *ci* is the confidence interval of coefficient *j* of *mdl*. The name of coefficient *j* of *mdl* is in *mdl*.CoefNames.

## Examples

### Default Confidence Intervals

Create a nonlinear model for auto mileage based on the `carbig` data. Then obtain confidence intervals for the resulting model coefficients.

Load the data and create a nonlinear model.

```
load carbig
ds = dataset(Horsepower,Weight,MPG);
modelfun = @(b,x)b(1) + b(2)*x(:,1) + ...
    b(3)*x(:,2) + b(4)*x(:,1).*x(:,2);
beta0 = [1 1 1 1];
mdl = fitnlm(ds,modelfun,beta0)
```

```
mdl =
```

```
Nonlinear regression model:
```

```
MPG ~ b1 + b2*Horsepower + b3*Weight + b4*Horsepower*Weight
```

```
Estimated Coefficients:
```

	Estimate	SE	tStat	pValue
b1	63.558	2.3429	27.127	1.2343e-91
b2	-0.25084	0.027279	-9.1952	2.3226e-18
b3	-0.010772	0.00077381	-13.921	5.1372e-36
b4	5.3554e-05	6.6491e-06	8.0542	9.9336e-15

```
Number of observations: 392, Error degrees of freedom: 388
```

```

Root Mean Squared Error: 3.93
R-Squared: 0.748, Adjusted R-Squared 0.746
F-statistic vs. constant model: 385, p-value = 7.26e-116

```

All the coefficients have extremely small  $P$ -values. This means a confidence interval around the coefficients will not contain the point 0, unless the confidence level is very high.

Find 95% confidence intervals for the coefficients of the model.

```
ci = coefCI mdl
```

```

ci =
    58.9515    68.1644
   -0.3045   -0.1972
   -0.0123   -0.0093
    0.0000    0.0001

```

The confidence interval for  $b_4$  seems to contain 0. Examine it in more detail.

```
ci(4, :)
```

```

ans =
    1.0e-04 *
    0.4048    0.6663

```

As expected, the confidence interval does not contain the point 0.

- “Nonlinear Regression Workflow” on page 13-15

## Definitions

### Confidence Interval

Assume that model assumptions hold (the data comes from a model represented by the formula `mdl.Formula`, with independent normally distributed errors). Then row  $j$  of the

confidence interval matrix `ci` gives a confidence interval `[ci(j,1),ci(j,2)]` that contains coefficient `j` with probability `1 - alpha`.

## Alternatives

You can create the intervals from the model coefficients in `mdl.Coefficients.Estimate` and an appropriate multiplier of the standard errors `sqrt(diag(mdl.CoefficientCovariance))`. The multiplier is `tinv(1-alpha/2,dof)`, where `level` is the confidence level, and `dof` is the degrees of freedom (number of data points minus the number of coefficients).

## See Also

`NonLinearModel`

## Topics

“Nonlinear Regression Workflow” on page 13-15

“Nonlinear Regression” on page 13-2

# coefTest

**Class:** CompactGeneralizedLinearModel

Linear hypothesis test on generalized linear regression model coefficients

## Syntax

```
p = coefTest mdl
p = coefTest mdl, H
p = coefTest mdl, H, C
[p, F] = coefTest mdl, ...
[p, F, r] = coefTest mdl, ...
```

## Description

`p = coefTest(mdl)` computes the  $p$ -value for an  $F$  test that all coefficient estimates in `mdl` are zero, except for the intercept term.

`p = coefTest(mdl, H)` performs an  $F$  test that  $H*B = 0$ , where  $B$  represents the coefficient vector.

`p = coefTest(mdl, H, C)` performs an  $F$  test that  $H*B = C$ .

`[p, F] = coefTest(mdl, ...)` returns the  $F$  test statistic.

`[p, F, r] = coefTest(mdl, ...)` returns the numerator degrees of freedom for the test.

## Input Arguments

### **mdl**

Generalized linear model, specified as a full `GeneralizedLinearModel` object constructed using `fitglm` or `stepwiseglm`, or a compacted `CompactGeneralizedLinearModel` object constructed using `compact`.

**H**

Numeric matrix having one column for each coefficient in the model. When  $H$  is an input, the output  $p$  is the  $p$ -value for an  $F$  test that  $H*B = 0$ , where  $B$  represents the coefficient vector.

**C**

Numeric vector with the same number of rows as  $H$ . When  $C$  is an input, the output  $p$  is the  $p$ -value for an  $F$  test that  $H*B = C$ , where  $B$  represents the coefficient vector.

## Output Arguments

**p**

$p$ -value of the  $F$  test (see “Definitions” on page 32-659).

**F**

Value of the test statistic for the  $F$  test (see “Definitions” on page 32-659).

**r**

Numerator degrees of freedom for the  $F$  test (see “Definitions” on page 32-659). The  $F$  statistic has  $r$  degrees of freedom in the numerator and  $mdl.DFE$  degrees of freedom in the denominator.

## Examples

### Test Generalized Linear Model Coefficients

Test a generalized linear model to see if its coefficients differ from zero.

Create a generalized linear regression model of Poisson data.

```
X = 2 + randn(100,1);  
mu = exp(1 + X/2);  
y = poissrnd(mu);  
mdl = fitglm(X,y,'y ~ x1','distr','poisson');
```



Test whether the fitted model has coefficients that differ significantly from zero.

```
p = coefTest mdl)
```

```
p =
```

```
3.1394e-36
```

There is no doubt that the coefficient of `x1` is nonzero.

- “Generalized Linear Model Workflow” on page 12-41

## Definitions

### Test Statistics

The  $p$ -value,  $F$  statistic, and numerator degrees of freedom are valid under these assumptions:

- The data comes from a model represented by the formula `mdl.Formula`.
- The observations are independent conditional on the predictor values.

Suppose these assumptions hold. Let  $\beta$  represent the (unknown) coefficient vector of the linear regression. Suppose  $H$  is a full-rank matrix of size  $r$ -by- $s$ , where  $s$  is the number of terms in  $\beta$ . Let  $v$  be a vector the same size as  $\beta$ . The following is a test statistic for the hypothesis that  $H\beta = v$ :

$$F = (H\hat{\beta} - v)'(HCH')^{-1}(H\hat{\beta} - v).$$

Here  $\hat{\beta}$  is the estimate of the coefficient vector  $\beta$  in `mdl.Coeffs`, and  $C$  is the estimated covariance of the coefficient estimates in `mdl.CoeffCov`. When the hypothesis is true, the test statistic  $F$  has an “F Distribution” on page B-44 with  $r$  and  $u$  degrees of freedom.

## Alternatives

The values of commonly used test statistics are available in the `mdl.Coefficients` table.

## See Also

`CompactGeneralizedLinearModel` | `GeneralizedLinearModel` | `linhyptest`

## Topics

“Generalized Linear Model Workflow” on page 12-41

“Generalized Linear Models” on page 12-12

## coefTest

**Class:** GeneralizedLinearMixedModel

Hypothesis test on fixed and random effects of generalized linear mixed-effects model

### Syntax

```
pVal = coefTest(glme)
pVal = coefTest(glme,H)
pVal = coefTest(glme,H,C)
pVal = coefTest(glme,H,C,Name,Value)
[pVal,F,DF1,DF2] = coefTest(____)
```

### Description

`pVal = coefTest(glme)` returns the  $p$ -value of an  $F$ -test of the null hypothesis that all fixed-effects coefficients of the generalized linear mixed-effects model `glme`, except for the intercept, are equal to 0.

`pVal = coefTest(glme,H)` returns the  $p$ -value of an  $F$ -test using a specified contrast matrix,  $H$ . The null hypothesis is  $H_0: H\beta = 0$ , where  $\beta$  is the fixed-effects vector.

`pVal = coefTest(glme,H,C)` returns the  $p$ -value for an  $F$ -test using the hypothesized value,  $C$ . The null hypothesis is  $H_0: H\beta = C$ , where  $\beta$  is the fixed-effects vector.

`pVal = coefTest(glme,H,C,Name,Value)` returns the  $p$ -value for an  $F$ -test on the fixed- and/or random-effects coefficients of the generalized linear mixed-effects model `glme`, with additional options specified by one or more name-value pair arguments. For example, you can specify the method to compute the approximate denominator degrees of freedom for the  $F$ -test.

`[pVal,F,DF1,DF2] = coefTest(____)` also returns the  $F$ -statistic,  $F$ , and the numerator and denominator degrees of freedom for  $F$ , respectively  $DF1$  and  $DF2$ , using any of the previous syntaxes.

## Input Arguments

### **g1me** — Generalized linear mixed-effects model

`GeneralizedLinearMixedModel` object

Generalized linear mixed-effects model, specified as a `GeneralizedLinearMixedModel` object. For properties and methods of this object, see `GeneralizedLinearMixedModel`.

### **H** — Fixed-effects contrasts

$m$ -by- $p$  matrix

Fixed-effects contrasts, specified as an  $m$ -by- $p$  matrix, where  $p$  is the number of fixed-effects coefficients in `g1me`. Each row of `H` represents one contrast. The columns of `H` (left to right) correspond to the rows of the  $p$ -by-1 fixed-effects vector `beta` (top to bottom) whose estimate is returned by the `fixedEffects` method.

Data Types: `single` | `double`

### **c** — Hypothesized value

$m$ -by-1 vector

Hypothesized value for testing the null hypothesis  $H\beta = C$ , specified as an  $m$ -by-1 vector. Here,  $\beta$  is the vector of fixed-effects whose estimate is returned by `fixedEffects`.

Data Types: `single` | `double`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### **DFMethod** — Method for computing approximate degrees of freedom

`'residual'` (default) | `'none'`

Method for computing approximate degrees of freedom, specified as the comma-separated pair consisting of `'DFMethod'` and one of the following.

Value	Description
'residual'	The degrees of freedom are assumed to be constant and equal to $n - p$ , where $n$ is the number of observations and $p$ is the number of fixed effects.
'none'	All degrees of freedom are set to infinity.

Example: 'DFMethod', 'none'

### REContrast — Random-effects contrasts

*m*-by-*q* matrix

Random-effects contrasts, specified as the comma-separated pair consisting of 'REContrast' and an *m*-by-*q* matrix, where *q* is the number of random effects parameters in `glme`. The columns of the matrix (left to right) correspond to the rows of the *q*-by-1 random-effects vector **B** (top to bottom), whose estimate is returned by the `randomEffects` method.

Data Types: `single` | `double`

## Output Arguments

### pVal — *p*-value

scalar value

*p*-value for the *F*-test on the fixed- and/or random-effects coefficients of the generalized linear mixed-effects model `glme`, returned as a scalar value.

When fitting a GLME model using `fitglme` and one of the maximum likelihood fit methods ('Laplace' or 'ApproximateLaplace'), `coefTest` uses an approximation of the conditional mean squared error of prediction (CMSEP) of the estimated linear combination of fixed- and random-effects to compute *p*-values. This accounts for the uncertainty in the fixed-effects estimates, but not for the uncertainty in the covariance parameter estimates. For tests on fixed effects only, if you specify the 'CovarianceMethod' name-value pair argument in `fitglme` as 'JointHessian', then `coefTest` accounts for the uncertainty in the estimation of covariance parameters.

When fitting a GLME model using `fitglme` and one of the pseudo likelihood fit methods ('MPL' or 'REMP'), `coefTest` bases the inference on the fitted linear mixed effects model from the final pseudo likelihood iteration.

**F** — *F*-statistic

scalar value

*F*-statistic, returned as a scalar value.

**DF1** — Numerator degrees of freedom for **F**

scalar value

Numerator degrees of freedom for the *F*-statistic **F**, returned as a scalar value.

- If you test the null hypothesis  $H_0: H\beta = 0$  or  $H_0: H\beta = C$ , then **DF1** is equal to the number of linearly independent rows in **H**.
- If you test the null hypothesis  $H_0: H\beta + KB = C$ , then **DF1** is equal to the number of linearly independent rows in  $[H, K]$ .

**DF2** — Denominator degrees of freedom for **F**

scalar value

Denominator degrees of freedom for the *F*-statistic **F**, returned as a scalar value. The value of **DF2** depends on the option specified by the 'DFMethod' name-value pair argument.

## Examples

**Test the Significance of Coefficients**

Load the sample data.

```
load mfr
```

This simulated data is from a manufacturing company that operates 50 factories across the world, with each factory running a batch process to create a finished product. The company wants to decrease the number of defects in each batch, so it developed a new manufacturing process. To test the effectiveness of the new process, the company selected 20 of its factories at random to participate in an experiment: Ten factories implemented

the new process, while the other ten continued to run the old process. In each of the 20 factories, the company ran five batches (for a total of 100 batches) and recorded the following data:

- Flag to indicate whether the batch used the new process (`newprocess`)
- Processing time for each batch, in hours (`time`)
- Temperature of the batch, in degrees Celsius (`temp`)
- Categorical variable indicating the supplier (A, B, or C) of the chemical used in the batch (`supplier`)
- Number of defects in the batch (`defects`)

The data also includes `time_dev` and `temp_dev`, which represent the absolute deviation of time and temperature, respectively, from the process standard of 3 hours at 20 degrees Celsius.

Fit a generalized linear mixed-effects model using `newprocess`, `time_dev`, `temp_dev`, and `supplier` as fixed-effects predictors. Include a random-effects intercept grouped by `factory`, to account for quality differences that might exist due to factory-specific variations. The response variable `defects` has a Poisson distribution, and the appropriate link function for this model is log. Use the Laplace fit method to estimate the coefficients. Specify the dummy variable encoding as 'effects', so the dummy variable coefficients sum to 0.

The number of defects can be modeled using a Poisson distribution

$$\text{defects}_{ij} \sim \text{Poisson}(\mu_{ij})$$

This corresponds to the generalized linear mixed-effects model

$$\log(\mu_{ij}) = \beta_0 + \beta_1 \text{newprocess}_{ij} + \beta_2 \text{time.dev}_{ij} + \beta_3 \text{temp.dev}_{ij} + \beta_4 \text{supplier.C}_{ij} + \beta_5 \text{supplier.B}_{ij} + b_i,$$

where

- $\text{defects}_{ij}$  is the number of defects observed in the batch produced by factory  $i$  during batch  $j$ .
- $\mu_{ij}$  is the mean number of defects corresponding to factory  $i$  (where  $i = 1, 2, \dots, 20$ ) during batch  $j$  (where  $j = 1, 2, \dots, 5$ ).

- `newprocessij`, `time_devij`, and `temp_devij` are the measurements for each variable that correspond to factory *i* during batch *j*. For example, `newprocessij` indicates whether the batch produced by factory *i* during batch *j* used the new process.
- `supplier_Cij` and `supplier_Bij` are dummy variables that use effects (sum-to-zero) coding to indicate whether company C or B, respectively, supplied the process chemicals for the batch produced by factory *i* during batch *j*.
- $b_i \sim N(0, \sigma_b^2)$  is a random-effects intercept for each factory *i* that accounts for factory-specific variation in quality.

```
glme = fitglme(mfr, 'defects ~ 1 + newprocess + time_dev + temp_dev + supplier + (1|fact
```

Test if there is any significant difference between supplier C and supplier B.

```
H = [0,0,0,0,1,-1];
```

```
[pVal,F,DF1,DF2] = coefTest(glme,H)
```

```
pVal =
```

```
0.2793
```

```
F =
```

```
1.1842
```

```
DF1 =
```

```
1
```

```
DF2 =
```

```
94
```

The large *P*-value indicates that there is no significant difference between supplier C and supplier B at the 5% significance level. Here, `coefTest` also returns the *F*-statistic, the numerator degrees of freedom, and the approximate denominator degrees of freedom.



Test if there is any significant difference between supplier A and supplier B.

If you specify the 'DummyVarCoding' name-value pair argument as 'effects' when fitting the model using `fitglme`, then

$$\beta_A + \beta_B + \beta_C = 0,$$

where  $\beta_A$ ,  $\beta_B$ , and  $\beta_C$  correspond to suppliers A, B, and C, respectively.  $\beta_A$  is the effect of A minus the average effect of A, B, and C. To determine the contrast matrix corresponding to a test between supplier A and supplier B,

$$\beta_B - \beta_A = \beta_B - (-\beta_B - \beta_C) = 2\beta_B + \beta_C.$$

From the output of `disp(glme)`, column 5 of the contrast matrix corresponds to  $\beta_C$ , and column 6 corresponds to  $\beta_B$ . Therefore, the contrast matrix for this test is specified as  $H = [0, 0, 0, 0, 1, 2]$ .

```
H = [0,0,0,0,1,2];
```

```
[pVal,F,DF1,DF2] = coefTest(glme,H)
```

```
pVal =
```

```
0.6177
```

```
F =
```

```
0.2508
```

```
DF1 =
```

```
1
```

```
DF2 =
```

```
94
```

The large  $P$ -value indicates that there is no significant difference between supplier A and supplier B at the 5% significance level.

## References

- [1] Booth, J.G., and J.P. Hobert. “Standard Errors of Prediction in Generalized Linear Mixed Models.” *Journal of the American Statistical Association*, Vol. 93, 1998, pp. 262–272.

## See Also

`GeneralizedLinearMixedModel` | `anova` | `coefCI` | `covarianceParameters` | `fixedEffects` | `randomEffects`

# coefTest

**Class:** CompactLinearModel

Linear hypothesis test on linear regression model coefficients

## Syntax

```
p = coefTest mdl
p = coefTest mdl, H
p = coefTest mdl, H, C
[p, F] = coefTest mdl, ____
[p, F, r] = coefTest mdl, ____
```

## Description

`p = coefTest(mdl)` computes the  $p$ -value for an  $F$  test that all coefficient estimates in `mdl` are zero, except for the intercept term.

`p = coefTest(mdl, H)` performs an  $F$  test that  $H \times B = 0$ , where  $B$  represents the coefficient vector.

`p = coefTest(mdl, H, C)` performs an  $F$  test that  $H \times B = C$ .

`[p, F] = coefTest(mdl, ____)` returns the  $F$  test statistic,  $F$ , using any of the previous syntaxes.

`[p, F, r] = coefTest(mdl, ____)` returns the numerator degrees of freedom,  $r$ , for the test.

## Input Arguments

**mdl** — Linear model object

LinearModel object | CompactLinearModel object

Linear model object, specified as a full `LinearModel` object constructed using `fitlm` or `stepwiselm`, or a compacted `CompactLinearModel` object constructed using `compact`.

**H — Hypothesized model coefficients**

numeric matrix

Hypothesized model coefficients, specified as a numeric matrix with one column for each coefficient in the model. If `H` is an input, then the output `p` is the  $p$ -value for an  $F$  test that  $H \times B = 0$ , where  $B$  represents the coefficient vector.

**C — Hypothesized value**

numeric vector

Hypothesized value for testing the null hypothesis, specified as a numeric vector with the same number of rows as `H`. When `C` is an input, the output `p` is the  $p$ -value for an  $F$  test that  $H \times B = C$ , where  $B$  represents the coefficient vector.

## Output Arguments

**p —  $p$ -value for  $F$  test**

numeric value in the range [0,1]

$p$ -value of the  $F$  test, returned as a numeric value in the range [0,1].

**F — Test statistic for  $F$  test**

numeric value

Value of the test statistic for the  $F$  test, returned as a numeric value.

**r — Numerator degrees of freedom for  $F$  test**

positive integer

Numerator degrees of freedom for the  $F$  test, returned as a positive integer. The  $F$  statistic has `r` degrees of freedom in the numerator and `mdl.DFE` degrees of freedom in the denominator.

## Examples

### Test Significance of Linear Regression Model

Fit a linear regression model of mileage as a function of the weight, weight squared, and model year from the `carsmall` data set. Test the coefficients to see if all should be zero.

Load the data and make a table, where the model year is an ordinal variable.

```
load carsmall
tbl = table(MPG,Weight);
tbl.Year = ordinal(Model_Year);
mdl = fitlm(tbl, 'MPG ~ Year + Weight + Weight^2');
```

Test the model for significant differences from a constant model.

```
p = coefTest(mdl)
```

```
p =
    5.5208e-41
```

The returned *P*-value indicates that the fitted model contains more than the intercept term.

### Test Significance of Linear Model Coefficient

Test the `Weight^2` coefficient in a linear model of mileage as a function of the weight, weight squared, and model year.

Load the data and make a table, where the model year is an ordinal variable.

```
load carsmall
tbl = table(MPG,Weight);
tbl.Year = ordinal(Model_Year);
mdl = fitlm(tbl, 'MPG ~ Year + Weight + Weight^2');
```

Test the significance of the `Weight^2` coefficient. To do so, find the coefficient corresponding to `Weight^2`.

```
mdl.CoefficientNames
```

```
ans =
```

```
1x5 cell array
    {'(Intercept)'}    {'Weight'}    {'Year_76'}    {'Year_82'}    {'Weight^2'}
```

`Weight^2` is the fifth (final) coefficient.

Test the significance of the `Weight^2` coefficient.

```
p = coefTest mdl, [0 0 0 0 1])
```

```
p =
```

```
0.0022
```

## Definitions

### Test Statistics

The  $p$ -value,  $F$  statistic, and numerator degrees of freedom are valid under these assumptions:

- The data comes from a model represented by the formula `mdl.Formula`.
- The observations are independent conditional on the predictor values.

Suppose these assumptions hold. Let  $\beta$  represent the (unknown) coefficient vector of the linear regression. Suppose  $H$  is a full-rank matrix of size  $r$ -by- $s$ , where  $s$  is the number of terms in  $\beta$ . Let  $v$  be a vector the same size as  $\beta$ . The following is a test statistic for the hypothesis that  $H\beta = v$ :

$$F = (H\hat{\beta} - v)' (HCH')^{-1} (H\hat{\beta} - v).$$

Here  $\hat{\beta}$  is the estimate of the coefficient vector  $\beta$  in `mdl.Coeffs`, and  $C$  is the estimated covariance of the coefficient estimates in `mdl.CoeffCov`. When the hypothesis is true, the test statistic  $F$  has an “F Distribution” on page B-44 with  $r$  and  $u$  degrees of freedom.

## Alternatives

The values of commonly used test statistics are available in the `mdl.Coefficients` table.

`anova` provides a test for each model term.

## See Also

`CompactLinearModel` | `LinearModel` | `anova` | `linhyptest`

## Topics

“Linear Regression” on page 11-11

## coefTest

**Class:** LinearMixedModel

Hypothesis test on fixed and random effects of linear mixed-effects model

### Syntax

```
pVal = coefTest(lme)
pVal = coefTest(lme, H)
pVal = coefTest(lme, H, C)
pVal = coefTest(lme, H, C, 'REContrast', K)
pVal = coefTest( ____, Name, Value)
[pVal, F, DF1, DF2] = coefTest( ____)
```

### Description

`pVal = coefTest(lme)` returns the  $p$ -value for an  $F$ -test that all fixed-effects coefficients except for the intercept are 0.

`pVal = coefTest(lme, H)` returns the  $p$ -value for an  $F$ -test on fixed-effects coefficients of linear mixed-effects model `lme`, using the contrast matrix `H`. It tests the null hypothesis that  $H_0: H\beta = 0$ , where  $\beta$  is the fixed-effects vector.

`pVal = coefTest(lme, H, C)` returns the  $p$ -value for an  $F$ -test on fixed-effects coefficients of the linear mixed-effects model `lme`, using the contrast matrix `H`. It tests the null hypothesis that  $H_0: H\beta = C$ , where  $\beta$  is the fixed-effects vector.

`pVal = coefTest(lme, H, C, 'REContrast', K)` returns the  $p$ -value for an  $F$ -test on the fixed- and random-effects coefficients of the linear mixed-effects model `lme`, using the contrast matrices `H` and `K`. It tests the null hypothesis that  $H_0: H\beta + KB = C$ , where  $\beta$  is the fixed-effects vector and  $B$  is the random-effects vector.

`pVal = coefTest( ____, Name, Value)` returns the  $p$ -value for an  $F$ -test on the fixed- and/or random-effects coefficients of the linear mixed-effects model `lme`, with additional options specified by one or more `Name, Value` pair arguments. It can accept any of the input arguments in the previous syntaxes.



For example, you can specify the method to compute the degrees of freedom.

`[pVal, F, DF1, DF2] = coefTest(____)` also returns the  $F$ -statistic  $F$ , and the numerator and denominator degrees of freedom for  $F$ , respectively  $DF1$  and  $DF2$ .

## Input Arguments

### **lme** — Linear mixed-effects model

`LinearMixedModel` object

Linear mixed-effects model, specified as a `LinearMixedModel` object constructed using `fitlme` or `fitlmematrix`.

### **H** — Fixed-effects contrasts

$m$ -by- $p$  matrix

Fixed-effects contrasts, specified as an  $m$ -by- $p$  matrix, where  $p$  is the number of fixed-effects coefficients in `lme`. Each row of `H` represents one contrast. The columns of `H` (left to right) correspond to the rows of the  $p$ -by-1 fixed-effects vector `beta` (top to bottom), returned by the `fixedEffects` method.

Data Types: `single` | `double`

### **C** — Hypothesized value

$m$ -by-1 vector

Hypothesized value for testing the null hypothesis  $H*beta = C$ , specified as an  $m$ -by-1 matrix. Here, `beta` is the vector of fixed-effects estimates returned by the `fixedEffects` method.

Data Types: `single` | `double`

### **K** — Random-effects contrasts

$m$ -by- $q$  matrix

Random-effects contrasts, specified as an  $m$ -by- $q$  matrix, where  $q$  is the number of random effects parameters in `lme`. The columns of `K` (left to right) correspond to the rows of the random-effects best linear unbiased predictor vector `B` (top to bottom), returned by the `randomEffects` method.

Data Types: `single` | `double`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### **DFMethod** — Method for computing approximate denominator degrees of freedom

`'residual'` (default) | `'satterthwaite'` | `'none'`

Method for computing the approximate denominator degrees of freedom for the  $F$ -test, specified as the comma-separated pair consisting of `'DFMethod'` and one of the following.

`'residual'`

Default. The degrees of freedom are assumed to be constant and equal to  $n - p$ , where  $n$  is the number of observations and  $p$  is the number of fixed effects.

`'satterthwaite'`

Satterthwaite approximation.

`'none'`

All degrees of freedom are set to infinity.

For example, you can specify the Satterthwaite approximation as follows.

Example: `'DFMethod', 'satterthwaite'`

## Output Arguments

### **pVal** — $p$ -value

scalar value

$p$ -value for the  $F$ -test on the fixed and/or random-effects coefficients of the linear mixed-effects model `lme`, returned as a scalar value.

### **F** — $F$ -statistic

scalar value

$F$ -statistic, returned as a scalar value.

### **DF1** — Numerator degrees of freedom for **F**

scalar value

Numerator degrees of freedom for  $F$ , returned as a scalar value.

- If you test the null hypothesis  $H_0: H\beta = 0$ , or  $H_0: H\beta = C$ , then  $DF1$  is equal to the number of linearly independent rows in  $H$ .
- If you test the null hypothesis  $H_0: H\beta + KB = C$ , then  $DF1$  is equal to the number of linearly independent rows in  $[H, K]$ .

### **DF2 — Denominator degrees of freedom for $F$**

scalar value

Denominator degrees of freedom for  $F$ , returned as a scalar value. The value of  $DF2$  depends on the option you select for `DFMethod`.

## Examples

### Test Fixed-Effects Coefficients for Categorical Data

Load the sample data.

```
load(fullfile(matlabroot, 'examples', 'stats', 'shift.mat'))
```

The data shows the absolute deviations from the target quality characteristic measured from the products that five operators manufacture during three different shifts: morning, evening, and night. This is a randomized block design, where the operators are the blocks. The experiment is designed to study the impact of the time of shift on the performance. The performance measure is the absolute deviation of the quality characteristics from the target value. This is simulated data.

`Shift` and `Operator` are nominal variables.

```
shift.Shift = nominal(shift.Shift);
shift.Operator = nominal(shift.Operator);
```

Fit a linear mixed-effects model with a random intercept grouped by operator to assess if there is significant difference in the performance according to the time of the shift.

```
lme = fitlme(shift, 'QCDev ~ Shift + (1|Operator)')
```

```
lme =
```

Linear mixed-effects model fit by ML

Model information:

Number of observations	15
Fixed effects coefficients	3
Random effects coefficients	5
Covariance parameters	2

Formula:

QCDev ~ 1 + Shift + (1 | Operator)

Model fit statistics:

AIC	BIC	LogLikelihood	Deviance
59.012	62.552	-24.506	49.012

Fixed effects coefficients (95% CIs):

Name	Estimate	SE	tStat	DF	pValue
'(Intercept)'	3.1196	0.88681	3.5178	12	0.0042407
'Shift_Morning'	-0.3868	0.48344	-0.80009	12	0.43921
'Shift_Night'	1.9856	0.48344	4.1072	12	0.0014535

Lower	Upper
1.1874	5.0518
-1.4401	0.66653
0.93227	3.0389

Random effects covariance parameters (95% CIs):

Group: Operator (5 Levels)

Name1	Name2	Type	Estimate	Lower
'(Intercept)'	'(Intercept)'	'std'	1.8297	0.94915

Upper
3.5272

Group: Error

Name	Estimate	Lower	Upper
'Res Std'	0.76439	0.49315	1.1848

Test if all fixed-effects coefficients except for the intercept are 0.

```
pVal = coefTest(lme)
```

```
pVal =
    7.5956e-04
```

The small **P**-value indicates that not all fixed-effects coefficients are 0.

Test the significance of the `Shift` term using a contrast matrix.

```
H = [0 1 0; 0 0 1];
pVal = coefTest(lme,H)
```

```
pVal =
    7.5956e-04
```

Test the significance of the `Shift` term using the anova method.

```
anova(lme)
```

```
ans =
```

```
ANOVA MARGINAL TESTS: DFMETHOD = 'RESIDUAL'
```

Term	FStat	DF1	DF2	pValue
'(Intercept)'	12.375	1	12	0.0042407
'Shift'	13.864	2	12	0.00075956

The **P**-value for `Shift`, 0.00075956, is the same as the **P**-value of the previous hypothesis test.

Test if there is any difference between the evening and morning shifts.

```
pVal = coefTest(lme,[0 1 -1])
```

```
pVal =
```

```
3.6147e-04
```

This small *P*-value indicates that the performance of the operators are not the same in the morning and the evening shifts.

### Hypothesis Tests for Fixed-Effects Coefficients

Load the sample data.

```
load(fullfile(matlabroot, 'examples', 'stats', 'weight.mat'))
```

`weight` contains data from a longitudinal study, where 20 subjects are randomly assigned to 4 exercise programs, and their weight loss is recorded over six 2-week time periods. This is simulated data.

Store the data in a table. Define `Subject` and `Program` as categorical variables.

```
tbl = table(InitialWeight, Program, Subject, Week, y);  
tbl.Subject = nominal(tbl.Subject);  
tbl.Program = nominal(tbl.Program);
```

Fit a linear mixed-effects model where the initial weight, type of program, week, and the interaction between the week and type of program are the fixed effects. The intercept and week vary by subject.

```
lme = fitlme(tbl, 'y ~ InitialWeight + Program*Week + (Week|Subject)')
```

```
lme =
```

```
Linear mixed-effects model fit by ML
```

```
Model information:
```

Number of observations	120
Fixed effects coefficients	9
Random effects coefficients	40
Covariance parameters	4

```
Formula:
```

y ~ 1 + InitialWeight + Program\*Week + (1 + Week | Subject)

Model fit statistics:

AIC	BIC	LogLikelihood	Deviance
-22.981	13.257	24.49	-48.981

Fixed effects coefficients (95% CIs):

Name	Estimate	SE	tStat	DF
'(Intercept)'	0.66105	0.25892	2.5531	111
'InitialWeight'	0.0031879	0.0013814	2.3078	111
'Program_B'	0.36079	0.13139	2.746	111
'Program_C'	-0.033263	0.13117	-0.25358	111
'Program_D'	0.11317	0.13132	0.86175	111
'Week'	0.1732	0.067454	2.5677	111
'Program_B:Week'	0.038771	0.095394	0.40644	111
'Program_C:Week'	0.030543	0.095394	0.32018	111
'Program_D:Week'	0.033114	0.095394	0.34713	111

pValue	Lower	Upper
0.012034	0.14798	1.1741
0.022863	0.00045067	0.0059252
0.0070394	0.10044	0.62113
0.80029	-0.29319	0.22666
0.39068	-0.14706	0.3734
0.011567	0.039536	0.30686
0.68521	-0.15026	0.2278
0.74944	-0.15849	0.21957
0.72915	-0.15592	0.22214

Random effects covariance parameters (95% CIs):

Group: Subject (20 Levels)

Name1	Name2	Type	Estimate	Lower
'(Intercept)'	'(Intercept)'	'std'	0.18407	0.12281
'Week'	'(Intercept)'	'corr'	0.66841	0.21077
'Week'	'Week'	'std'	0.15033	0.11004

Upper  
0.27587  
0.88573  
0.20537

Group: Error

Name	Estimate	Lower	Upper
'Res Std'	0.10261	0.087882	0.11981

Test for the significance of the interaction between Program and Week.

```
H = [0 0 0 0 0 0 1 0 0;  
      0 0 0 0 0 0 0 1 0;  
      0 0 0 0 0 0 0 0 1];  
pVal = coefTest(lme,H)
```

```
pVal =  
  
0.9775
```

The high  $P$ -value indicates that the interaction between Program and Week is not statistically significant.

Now, test whether all coefficients involving Program are 0.

```
H = [0 0 1 0 0 0 0 0 0;  
      0 0 0 1 0 0 0 0 0;  
      0 0 0 0 1 0 0 0 0;  
      0 0 0 0 0 0 1 0 0;  
      0 0 0 0 0 0 0 1 0;  
      0 0 0 0 0 0 0 0 1];  
C = [0;0;0;0;0;0];  
pVal = coefTest(lme,H,C)
```

```
pVal =  
  
0.0274
```

The  $P$ -value of 0.0274 indicates that not all coefficients involving Program are zero.

### Hypothesis Tests for Fixed- and Random-Effects Coefficients

Load the sample data.



```
load flu
```

The `flu` dataset array has a `Date` variable, and 10 variables containing estimated influenza rates (in 9 different regions, estimated from Google® searches, plus a nationwide estimate from the CDC).

To fit a linear-mixed effects model, your data must be in a properly formatted dataset array. To fit a linear mixed-effects model with the influenza rates as the responses and region as the predictor variable, combine the nine columns corresponding to the regions into an array. The new dataset array, `flu2`, must have the response variable, `FluRate`, the nominal variable, `Region`, that shows which region each estimate is from, and the grouping variable `Date`.

```
flu2 = stack(flu,2:10,'NewDataVarName','FluRate',...
    'IndVarName','Region');
flu2.Date = nominal(flu2.Date);
```

Fit a linear mixed-effects model with fixed effects for the region and a random intercept that varies by `Date`.

```
lme = fitlme(flu2,'FluRate ~ 1 + Region + (1|Date)')
```

```
lme =
```

```
Linear mixed-effects model fit by ML
```

```
Model information:
```

Number of observations	468
Fixed effects coefficients	9
Random effects coefficients	52
Covariance parameters	2

```
Formula:
```

```
FluRate ~ 1 + Region + (1 | Date)
```

```
Model fit statistics:
```

AIC	BIC	LogLikelihood	Deviance
318.71	364.35	-148.36	296.71

```
Fixed effects coefficients (95% CIs):
```

Name	Estimate	SE	tStat	DF
'(Intercept)'	1.2233	0.096678	12.654	459

'Region_MidAtl'	0.010192	0.052221	0.19518	459
'Region_ENCentral'	0.051923	0.052221	0.9943	459
'Region_WNCentral'	0.23687	0.052221	4.5359	459
'Region_SAtl'	0.075481	0.052221	1.4454	459
'Region_ESCentral'	0.33917	0.052221	6.495	459
'Region_WSCentral'	0.069	0.052221	1.3213	459
'Region_Mtn'	0.046673	0.052221	0.89377	459
'Region_Pac'	-0.16013	0.052221	-3.0665	459

pValue	Lower	Upper
1.085e-31	1.0334	1.4133
0.84534	-0.092429	0.11281
0.3206	-0.050698	0.15454
7.3324e-06	0.13424	0.33949
0.14902	-0.02714	0.1781
2.1623e-10	0.23655	0.44179
0.18705	-0.033621	0.17162
0.37191	-0.055948	0.14929
0.0022936	-0.26276	-0.057514

Random effects covariance parameters (95% CIs):

Group: Date (52 Levels)

Name1	Name2	Type	Estimate	Lower
'(Intercept)'	'(Intercept)'	'std'	0.6443	0.5297

Upper  
0.78368

Group: Error

Name	Estimate	Lower	Upper
'Res Std'	0.26627	0.24878	0.285

Test the hypothesis that the random effects-term for week 10/9/2005 is zero.

```
[~,~,STATS] = randomEffects(lme); % Compute the random-effects statistics (STATS)
STATS.Level = nominal(STATS.Level);
K = zeros(length(STATS),1);
K(STATS.Level == '10/9/2005') = 1;
pVal = coefTest(lme,[0 0 0 0 0 0 0 0 0],0,'REContrast',K')
```

pVal =

```
0.1692
```

Refit the model this time with a random intercept and slope.

```
lme = fitlme(flu2, 'FluRate ~ 1 + Region + (1 + Region|Date)');
```

Test the hypothesis that the combined coefficient of region WNCentral for week 10/9/2005 is zero.

```
[~,~,STATS] = randomEffects(lme); STATS.Level = nominal(STATS.Level);
K = zeros(length(STATS),1);
K(STATS.Level == '10/9/2005' & flu2.Region == 'WNCentral') = 1;
pVal = coefTest(lme,[0 0 0 1 0 0 0 0 0],0,'REContrast',K)
```

```
pVal =
```

```
1.3911e-12
```

Also return the  $F$ -statistic with the numerator and denominator degrees of freedom.

```
[pVal,F,DF1,DF2] = coefTest(lme,[0 0 0 1 0 0 0 0 0],0,'REContrast',K)
```

```
pVal =
```

```
1.3911e-12
```

```
F =
```

```
53.1044
```

```
DF1 =
```

```
1
```

```
DF2 =
```

```
459
```

Repeat the test using the Satterthwaite approximation for the denominator degrees of freedom.

```
[pVal,F,DF1,DF2] = coefTest(lme,[0 0 0 1 0 0 0 0 0],0,'REContrast','K',...  
    'DFMethod','satterthwaite')
```

```
pVal =
```

```
NaN
```

```
F =
```

```
53.1044
```

```
DF1 =
```

```
1
```

```
DF2 =
```

```
0
```

## See Also

`LinearMixedModel` | `anova` | `coefCI`

# coefTest

**Class:** NonLinearModel

Linear hypothesis test on nonlinear regression model coefficients

## Syntax

```
p = coefTest mdl
p = coefTest mdl, H
p = coefTest mdl, H, C
[p, F] = coefTest mdl, ...
[p, F, r] = coefTest mdl, ...
```

## Description

`p = coefTest(mdl)` computes the  $p$ -value for an  $F$  test that all coefficient estimates in `mdl` are zero.

`p = coefTest(mdl, H)` performs an  $F$  test that  $H*B = 0$ , where  $B$  represents the coefficient vector.

`p = coefTest(mdl, H, C)` performs an  $F$  test that  $H*B = C$ .

`[p, F] = coefTest(mdl, ...)` returns the  $F$  test statistic.

`[p, F, r] = coefTest(mdl, ...)` returns the numerator degrees of freedom for the test.

## Input Arguments

**mdl**

Nonlinear regression model, constructed by `fitnlm`.

**H**

Numeric matrix having one column for each coefficient in the model. When `H` is an input, the output `p` is the  $p$ -value for an  $F$  test that  $H*B = 0$ , where `B` represents the coefficient vector.

**C**

Numeric vector with the same number of rows as `H`. When `C` is an input, the output `p` is the  $p$ -value for an  $F$  test that  $H*B = C$ , where `B` represents the coefficient vector.

## Output Arguments

**P**

$p$ -value of the  $F$  test (see “Definitions” on page 32-689).

**F**

Value of the test statistic for the  $F$  test (see “Definitions” on page 32-689).

**r**

Numerator degrees of freedom for the  $F$  test (see “Definitions” on page 32-689). The  $F$  statistic has `r` degrees of freedom in the numerator and `mdl.DFE` degrees of freedom in the denominator.

## Examples

### Test Nonlinear Regression Model Coefficients

Make a nonlinear model of mileage as a function of the weight from the `carsmall` data set. Test the coefficients to see if all should be zero.

Create an exponential model of car mileage as a function of weight from the `carsmall` data. Scale the weight by a factor of 1000 so all the variables are roughly equal in size.

```
load carsmall
X = Weight;
```

```

y = MPG;
modelfun = 'y ~ b1 + b2*exp(-b3*x/1000)';
beta0 = [1 1 1];
mdl = fitnlm(X,y,modelfun,beta0);

```

Test the model for significant differences from a constant model.

```
p = coefTest(mdl)
```

```
p =
```

```
1.3708e-36
```

There is no doubt that the model contains nonzero terms.

## Definitions

### Test Statistics

The  $p$ -value,  $F$  statistic, and numerator degrees of freedom are valid under these assumptions:

- The data comes from a normal distribution.
- The entries are independent.

Suppose these assumptions hold. Let  $\beta$  represent the unknown coefficient vector of the linear regression. Suppose  $H$  is a full-rank matrix of size  $r$ -by- $s$ , where  $s$  is the number of terms in  $\beta$ . Let  $v$  be a vector the same size as  $\beta$ . The following is a test statistic for the hypothesis that  $H\beta = v$ :

$$F = (H\hat{\beta} - v)'(HCH')^{-1}(H\hat{\beta} - v).$$

Here  $\hat{\beta}$  is the estimate of the coefficient vector  $\beta$  in `mdl.Coeffs`, and  $C$  is the estimated covariance of the coefficient estimates in `mdl.CoeffCov`. When the hypothesis is true, the test statistic  $F$  has an “F Distribution” on page B-44 with  $r$  and  $u$  degrees of freedom.

## Alternatives

The values of commonly used test statistics are available in the `mdl.Coefficients` table.

## See Also

`NonLinearModel`

## Topics

“Nonlinear Regression” on page 13-2



# coefstest

**Class:** RepeatedMeasuresModel

Linear hypothesis test on coefficients of repeated measures model

## Syntax

```
tbl = coefstest(rm, A, C, D)
```

## Description

`tbl = coefstest(rm, A, C, D)` returns a table `tbl` containing the multivariate analysis of variance (manova) for the repeated measures model `rm`.

## Input Arguments

**rm** — Repeated measures model

RepeatedMeasuresModel object

Repeated measures model, returned as a RepeatedMeasuresModel object.

For properties and methods of this object, see RepeatedMeasuresModel.

**A** — Specification representing between-subjects model

$a$ -by- $p$  matrix

Specification representing the between-subjects model, specified as an  $a$ -by- $p$  numeric matrix, with rank  $a \leq p$ .

Data Types: `single` | `double`

**c** — Specification representing within-subjects hypothesis

$r$ -by- $c$  matrix

Specification representing the within-subjects (within time) hypotheses, specified as an  $r$ -by- $c$  numeric matrix, with rank  $c \leq r \leq n - p$ .

Data Types: `single` | `double`

#### **D — Hypothesized value**

0 (default) | scalar value |  $a$ -by- $c$  matrix

Hypothesized value, specified as a scalar value or an  $a$ -by- $c$  matrix.

Data Types: `single` | `double`

## Output Arguments

### **tb1 — Results of multivariate analysis of variance**

table

Results of multivariate analysis of variance for the repeated measures model `rm`, returned as a table containing the following columns.

<code>Statistic</code>	Type of test statistic used
<code>Value</code>	Value of the corresponding test statistic
<code>F</code>	$F$ -statistic value
<code>RSquare</code>	Measure of variance explained
<code>df1</code>	Numerator degrees of freedom for the $F$ -statistic
<code>df2</code>	Denominator degrees of freedom for the $F$ -statistic
<code>pValue</code>	$p$ -value associated with the test statistic value

## Examples

### **Test Coefficients for First and Last Repeated Measures**

Load the sample data.

```
load repeatedmeas
```

The table `between` includes the between-subject variables `age`, `IQ`, `group`, `gender`, and eight repeated measures `y1` through `y8` as responses. The table `within` includes the within-subject variables `w1` and `w2`. This is simulated data.

Fit a repeated measures model, where the repeated measures `y1` through `y8` are the responses, and `age`, `IQ`, `group`, `gender`, and the group-gender interaction are the predictor variables. Also specify the within-subject design matrix.

```
rm = fitrm(between, 'y1-y8 ~ Group*Gender + Age + IQ', 'WithinDesign', within);
```

Test that the coefficients of all terms in the between-subjects model are the same for the first and last repeated measurement variable.

```
coefstest(rm, eye(8), [1 0 0 0 0 0 0 -1]')
```

```
ans =
```

```
4x7 table
```

Statistic	Value	F	RSquare	df1	df2	pValue
Pillai	0.3355	1.3884	0.3355	8	22	0.25567
Wilks	0.6645	1.3884	0.3355	8	22	0.25567
Hotelling	0.50488	1.3884	0.3355	8	22	0.25567
Roy	0.50488	1.3884	0.3355	8	22	0.25567

The **P**-value of 0.25567 indicates that there is not enough statistical evidence to conclude that the coefficients of all terms in the between-subjects model for the first and last repeated measures variable are different.

## Tips

- This test is defined as  $A*B*C = D$ , where  $B$  is the matrix of coefficients in the repeated measures model.  $A$  and  $C$  are numeric matrices of the proper size for this multiplication.  $D$  is a scalar or numeric matrix of the proper size. The default is  $D = 0$ .

## See Also

`fitrm` | `manova`

# combine

**Class:** CompactTreeBagger

Combine two ensembles

## Syntax

```
B1 = combine(B1,B2)
```

## Description

`B1 = combine(B1,B2)` appends decision trees from ensemble `B2` to those stored in `B1` and returns ensemble `B1`. This method requires that the class and variable names be identical in both ensembles.

## See Also

`TreeBagger.append`

## combnk

Enumeration of combinations

### Syntax

```
C = combnk(v, k)
```

### Description

`C = combnk(v, k)` returns all combinations of the  $n$  elements in  $v$  taken  $k$  at a time.

`C = combnk(v, k)` produces a matrix  $C$  with  $k$  columns and  $n!/k!(n-k)!$  rows, where each row contains  $k$  of the elements in the vector  $v$ .

It is not practical to use this function if  $v$  has more than about 15 elements.

### Examples

Combinations of characters.

```
C = combnk('tendrill', 4);  
last5 = C(31:35, :);  
last5 =  
tedr  
tenl  
teni  
tenr  
tend
```

Combinations of elements from a numeric vector.

```
c = combnk(1:4, 2);  
c =  
 3  4  
 2  4  
 2  3
```

1 4  
1 3  
1 2

## See Also

perms | randperm

**Introduced before R2006a**

## compact

**Class:** ClassificationDiscriminant

Compact discriminant analysis classifier

## Syntax

```
cobj = compact(obj)
```

## Description

`cobj = compact(obj)` creates a compact version of `obj`.

## Input Arguments

**obj**

Discriminant analysis classifier created using `fitcdiscr`.

## Output Arguments

**cobj**

Compact classifier. `cobj` has class `CompactClassificationDiscriminant`. You can predict classifications using `cobj` exactly as you can using `obj`. However, since `cobj` does not contain training data, you cannot perform some actions, such as cross validation.

## Examples

Compare the size of the discriminant analysis classifier for Fisher's iris data to the compact version of the classifier:



```
load fisheriris
fullobj = fitcdiscr(meas,species);
cobj = compact(fullobj);
b = whos('fullobj'); % b.bytes = size of fullobj
c = whos('cobj'); % c.bytes = size of cobj
[b.bytes c.bytes] % shows cobj uses 60% of the memory

ans =
    18578    11498
```

## See Also

ClassificationDiscriminant | fitcdiscr

## Topics

“Discriminant Analysis Classification” on page 20-2

## compact

**Class:** ClassificationECOC

Compact multiclass, error-correcting output codes model

## Syntax

```
CMdl = compact(Mdl)
```

## Description

`CMdl = compact(Mdl)` returns a compact, multiclass, error-correcting output codes (ECOC) model (CMdl), which is the compact version of the trained ECOC model Mdl.

CMdl does not contain the training data, whereas Mdl contains the training data in its properties `Mdl.X` and `Mdl.Y`.

## Input Arguments

**md1** — Multiclass ECOC model

ClassificationECOC model

Multiclass ECOC model, specified as a ClassificationECOC model returned by `fitcecoc`.

## Output Arguments

**cmd1** — Compact ECOC model

CompactClassificationECOC model

Compact ECOC model, returned as a CompactClassificationECOC model.

Predict class labels using `CMdl` exactly as you would using `Mdl`. However, since `CMdl` does not contain training data, you cannot implement cross validation.

## Examples

### Reduce the Size of Full ECOC Models

Full ECOC models (i.e., `ClassificationECOC` models) hold the training data. For efficiency, you might not want to predict new labels using a large classifier.

Load Fisher's iris data set.

```
load fisheriris
X = meas;
Y = categorical(species);
classOrder = unique(Y);
```

Train an ECOC model using SVM binary classifiers. It is good practice to standardize the predictors and define the class order. Specify to standardize the predictors using an SVM template.

```
t = templateSVM('Standardize',1);
Mdl = fitcecoc(X,Y,'Learners',t,'ClassNames',classOrder);
```

`t` is an SVM template object. The software uses default values for empty options in `t` during training. `Mdl` is a `ClassificationECOC` model.

Reduce the size of the trained ECOC model.

```
CMdl = compact(Mdl)
```

```
CMdl =
```

```
classreg.learning.classif.CompactClassificationECOC
    ResponseName: 'Y'
    CategoricalPredictors: []
        ClassNames: [setosa    versicolor    virginica]
    ScoreTransform: 'none'
    BinaryLearners: {3x1 cell}
    CodingMatrix: [3x3 double]
```

CMdl is a CompactClassificationECOC model. It does not store the training data nor some of the properties that Mdl stores.

Display how much memory each classifier uses.

```
whos('Mdl', 'CMdl')
```

Name	Size	Bytes	Class
CMdl	1x1	12956	classreg.learning.classif.CompactClassificationECOC
Mdl	1x1	25784	ClassificationECOC

The full ECOC model (Mdl) is approximately double the size of the compact ECOC model (CMdl).

You can remove Mdl from the MATLAB® Workspace, and pass CMdl and new predictor values to predict to efficiently label new observations.

## See Also

ClassificationECOC | CompactClassificationECOC | fitcecoc | predict

## compact

**Class:** ClassificationEnsemble

Compact classification ensemble

## Syntax

```
cens = compact(ens)
```

## Description

`cens = compact(ens)` creates a compact version of `ens`. You can predict classifications using `cens` exactly as you can using `ens`. However, since `cens` does not contain training data, you cannot perform some actions, such as cross validation.

## Input Arguments

**ens**

A classification ensemble created with `fitcensemble`.

## Output Arguments

**cens**

A compact classification ensemble. `cens` has class `CompactClassificationEnsemble`.

## Examples

### View Size of Compact Classification Ensemble

Compare the size of a classification ensemble for the Fisher iris data to the compact version of the ensemble.

Load the Fisher iris data set.

```
load fisheriris
```

Train an ensemble of 100 boosted classification trees using AdaBoostM2.

```
t = templateTree('MaxNumSplits',1); % Weak learner template tree object
ens = fitcensemble(meas,species,'Method','AdaBoostM2','Learners',t);
```

Create a compact version of ens and compare ensemble sizes.

```
cens = compact(ens);
b = whos('ens'); % b.bytes = size of ens
c = whos('cens'); % c.bytes = size of cens
[b.bytes c.bytes] % Shows cens uses less memory
```

```
ans =
      392654      352622
```

The compact version of the ensemble uses less memory than the full ensemble. Note that the ensemble sizes can vary slightly, depending on your operating system.

## See Also

ClassificationTree | fitcensemble

## Topics

“Framework for Ensemble Learning” on page 18-41

## compact

**Class:** ClassificationNaiveBayes

Compact naive Bayes classifier

## Syntax

```
CMdl = compact(Mdl)
```

## Description

`CMdl = compact(Mdl)` returns a compact naive Bayes classifier (CMdl), which is the compact version of the trained naive Bayes classifier Mdl.

CMdl stores less than Mdl, e.g., CMdl does not store the training data.

## Input Arguments

**Mdl** — Fully trained naive Bayes classifier

ClassificationNaiveBayes model

A fully trained naive Bayes classifier, specified as a ClassificationNaiveBayes model trained by `fitcnb`.

## Output Arguments

**CMdl** — Compact naive Bayes classifier

CompactClassificationNaiveBayes model

Compact naive Bayes classifier, returned as a CompactClassificationNaiveBayes model.

Predict class labels using `CMdl` exactly as you would using `Mdl`. However, since `CMdl` does not contain training data, you cannot perform certain tasks, such as cross validation.

## Examples

### Reduce the Size of Naive Bayes Classifiers

Full naive Bayes classifiers (i.e., `ClassificationNaiveBayes` models) hold the training data. For efficiency, you might not want to predict new labels using a large classifier. This example shows how to reduce the size of a full naive Bayes classifier.

Load the ionosphere data set.

```
load ionosphere
X = X(:,3:end); % Remove two predictors for stability
```

Train a naive Bayes classifier. Assume that each predictor is conditionally, normally distributed given its label. It is good practice to specify the order of the labels.

```
Mdl = fitcnb(X,Y,'ClassNames',{'b','g'})
```

```
Mdl =
```

```
ClassificationNaiveBayes
    ResponseName: 'Y'
    CategoricalPredictors: []
        ClassNames: {'b' 'g'}
    ScoreTransform: 'none'
    NumObservations: 351
    DistributionNames: {1x32 cell}
    DistributionParameters: {2x32 cell}
```

`Mdl` is a `ClassificationNaiveBayes` model.

Reduce the size of the naive Bayes classifier.

```
CMdl = compact(Mdl)
```



```

CMdl =

classreg.learning.classif.CompactClassificationNaiveBayes
    ResponseName: 'Y'
    CategoricalPredictors: []
        ClassNames: {'b' 'g'}
        ScoreTransform: 'none'
    DistributionNames: {1x32 cell}
    DistributionParameters: {2x32 cell}

```

CMdl is a CompactClassificationNaiveBayes model.

Display how much memory each classifier uses.

```
whos('Mdl', 'CMdl')
```

Name	Size	Bytes	Class
CMdl	1x1	15244	classreg.learning.classif.CompactClassificationNaiveBayes
Mdl	1x1	111374	ClassificationNaiveBayes

The full naive Bayes classifier (Mdl) is much larger than the compact naive Bayes classifier (CMdl).

You can remove Mdl from the MATLAB® Workspace, and pass CMdl and new predictor values to CompactClassificationNaiveBayes.predict to efficiently label new observations.

## See Also

ClassificationNaiveBayes | CompactClassificationNaiveBayes | fitcnb | predict

## compact

**Class:** ClassificationSVM

Compact support vector machine classifier

## Syntax

```
CompactSVMModel = compact(SVMModel)
```

## Description

`CompactSVMModel = compact(SVMModel)` returns a compact support vector machine (SVM) classifier (`CompactSVMModel`), the compact version of the trained SVM classifier `SVMModel`.

`CompactSVMModel` does not contain the training data, whereas `SVMModel` contains the training data in its properties `SVMModel.X` and `SVMModel.Y`.

## Input Arguments

**SVMModel** — Full, trained SVM classifier

ClassificationSVM classifier

Full, trained SVM classifier, specified as a `ClassificationSVM` model trained using `fitcsvm`.

## Output Arguments

**CompactSVMModel** — Compact SVM classifier

CompactClassificationSVM classifier

Compact SVM classifier, returned as a `CompactClassificationSVM` classifier.

Predict class labels using `CompactSVMModel` exactly as you would using `SVMModel`. However, since `CompactSVMModel` does not contain training data, you cannot perform certain tasks, such as cross validation.

## Examples

### Reduce the Size of Support Vector Machine Classifiers

Full SVM classifiers (i.e., `ClassificationSVM` classifiers) hold the training data. For efficiency, you might not want to predict new labels using a large classifier. This example shows how to reduce the size of a full SVM classifier.

Load the ionosphere data set.

```
load ionosphere
```

Train an SVM classifier. It is good practice to standardize the predictors and specify the order of the classes.

```
SVMModel = fitcsvm(X,Y,'Standardize',true,...  
    'ClassNames',{'b','g'})
```

```
SVMModel =
```

```
ClassificationSVM  
    ResponseName: 'Y'  
    CategoricalPredictors: []  
           ClassNames: {'b' 'g'}  
    ScoreTransform: 'none'  
    NumObservations: 351  
           Alpha: [90x1 double]  
           Bias: -0.1343  
    KernelParameters: [1x1 struct]  
           Mu: [1x34 double]  
           Sigma: [1x34 double]  
    BoxConstraints: [351x1 double]  
    ConvergenceInfo: [1x1 struct]  
    IsSupportVector: [351x1 logical]  
           Solver: 'SMO'
```

`SVMModel` is a `ClassificationSVM` classifier.

**Reduce the size of the SVM classifier.**

```
CompactSVMModel = compact(SVMModel)
```

```
CompactSVMModel =
```

```
classreg.learning.classif.CompactClassificationSVM
  ResponseName: 'Y'
  CategoricalPredictors: []
  ClassNames: {'b' 'g'}
  ScoreTransform: 'none'
  Alpha: [90x1 double]
  Bias: -0.1343
  KernelParameters: [1x1 struct]
  Mu: [1x34 double]
  Sigma: [1x34 double]
  SupportVectors: [90x34 double]
  SupportVectorLabels: [90x1 double]
```

`CompactSVMModel` is a `CompactClassificationSVM` classifier.

**Display how much memory each classifier uses.**

```
whos('SVMModel', 'CompactSVMModel')
```

Name	Size	Bytes	Class
CompactSVMModel	1x1	30512	classreg.learning.classif.CompactClassificationSVM
SVMModel	1x1	140834	ClassificationSVM

The full SVM classifier (`SVMModel`) is more than four times the compact SVM classifier (`CompactSVMModel`).

You can remove `SVMModel` from the MATLAB® Workspace, and pass `CompactSVMModel` and new predictor values to `predict` to efficiently label new observations.

## See Also

`ClassificationSVM` | `CompactClassificationSVM` | `fitcsvm`

## compact

**Class:** ClassificationTree

Compact tree

## Syntax

```
ctree = compact(tree)
```

## Description

`ctree = compact(tree)` creates a compact version of `tree`.

## Input Arguments

**tree**

A classification tree created using `fitctree`.

## Output Arguments

**ctree**

A compact decision tree. `ctree` has class `CompactClassificationTree`. You can predict classifications using `ctree` exactly as you can using `tree`. However, since `ctree` does not contain training data, you cannot perform some actions, such as cross validation.

## Examples

## Create a Compact Classification Tree

Compare the size of the classification tree for Fisher's iris data to the compact version of the tree.

```
load fisheriris
fulltree = fitctree(meas,species);
ctree = compact(fulltree);
b = whos('fulltree'); % b.bytes = size of fulltree
c = whos('ctree'); % c.bytes = size of ctree
[b.bytes c.bytes] % shows ctree uses half the memory
```

```
ans =
```

```
11260    4572
```

## See Also

[ClassificationTree](#) | [CompactClassificationTree](#) | [fitctree](#) | [predict](#)

## compact

**Class:** LinearModel

Compact linear regression model

## Syntax

```
compactMdl = compact (mdl)
```

## Description

`compactMdl = compact (mdl)` returns a compact linear regression model, `compactMdl`, which is the compact version of the full, fitted linear regression model `mdl`.

## Input Arguments

**mdl** — Full, fitted linear regression model

LinearModel object

Full, fitted linear regression model, specified as a `LinearModel` object constructed using `fitlm` or `stepwiselm`.

## Output Arguments

**compactMdl** — Compact linear regression model

CompactLinearModel object

Compact linear regression model, returned as a `CompactLinearModel` object.

Predict response values using `compactMdl` exactly as you would using `mdl`. However, since `compactMdl` does not contain training data, you cannot perform certain tasks, such as cross-validation.



## Examples

### Compact a Linear Regression Model

This example shows how to reduce the size of a full, fitted linear regression model by discarding the sample data and some information related to the fitting process.

Load the data into the workspace.

```
load(fullfile(matlabroot, 'examples', 'stats', 'largedata4reg.mat'))
```

The simulated sample data contains 15,000 observations and 45 predictor variables.

Fit a simple linear regression model to the data.

```
mdl = fitlm(X,Y)
```

```
mdl =
```

```
Linear regression model:
```

```
    y ~ [Linear formula with 46 terms in 45 predictors]
```

```
Estimated Coefficients:
```

	Estimate	SE	tStat	pValue
	-----	-----	-----	-----
(Intercept)	3.2903	1.2333e-05	2.6679e+05	0
x1	-0.0006461	5.9019e-09	-1.0947e+05	0
x2	-0.00024739	1.0256e-08	-24121	0
x3	-9.5161e-05	1.3149e-08	-7236.9	0
x4	0.00013143	1.8311e-08	7177.3	0
x5	7.163e-05	2.3367e-08	3065.4	0
x6	4.5064e-06	2.6264e-08	171.58	0
x7	-2.6258e-05	3.006e-08	-873.51	0
x8	6.284e-05	3.0262e-08	2076.5	0
x9	-0.00014288	3.3258e-08	-4296.1	0
x10	-2.2642e-05	3.6555e-08	-619.41	0
x11	-6.0227e-05	3.7353e-08	-1612.4	0
x12	1.1665e-05	4.0048e-08	291.27	0
x13	3.8595e-05	4.203e-08	918.26	0
x14	0.00010021	4.7592e-08	2105.5	0

x15	-6.5674e-06	4.9221e-08	-133.43	0
x16	8.5598e-06	5.0296e-08	170.19	0
x17	-3.9107e-05	5.3e-08	-737.87	0
x18	-6.5841e-06	5.5355e-08	-118.94	0
x19	-1.7053e-05	5.7431e-08	-296.94	0
x20	-3.8911e-06	6.2724e-08	-62.036	0
x21	-9.7219e-06	6.3515e-08	-153.06	0
x22	-1.8749e-06	6.5388e-08	-28.673	4.6032e-176
x23	-4.7514e-06	6.6636e-08	-71.303	0
x24	-1.7756e-05	6.8495e-08	-259.23	0
x25	-9.6673e-06	7.0054e-08	-138	0
x26	7.6237e-06	7.2442e-08	105.24	0
x27	-8.4338e-07	7.7519e-08	-10.88	1.8249e-27
x28	7.0502e-06	8.1889e-08	86.094	0
x29	-1.4703e-05	8.7126e-08	-168.75	0
x30	2.7008e-05	9.0084e-08	299.8	0
x31	6.3685e-07	9.1253e-08	6.979	3.0977e-12
x32	-1.9916e-05	1.0034e-07	-198.48	0
x33	1.7369e-05	1.019e-07	170.45	0
x34	-9.931e-06	1.0706e-07	-92.764	0
x35	-1.5195e-05	1.0858e-07	-139.94	0
x36	-1.0118e-05	1.1122e-07	-90.976	0
x37	2.4595e-06	1.1254e-07	21.856	2.9315e-104
x38	-2.2928e-06	1.1493e-07	-19.95	2.0535e-87
x39	1.1397e-05	1.1855e-07	96.136	0
x40	4.0239e-06	1.2327e-07	32.643	7.75e-226
x41	-8.6667e-06	1.2535e-07	-69.142	0
x42	-8.2932e-06	1.3095e-07	-63.334	0
x43	2.7309e-06	1.3452e-07	20.301	2.0697e-90
x44	-6.9235e-06	1.3725e-07	-50.444	0
x45	1.1165e-06	1.4021e-07	7.9633	1.7956e-15

Number of observations: 15000, Error degrees of freedom: 14954

Root Mean Squared Error: 0.00151

R-squared: 1, Adjusted R-Squared 1

F-statistic vs. constant model: 2.82e+08, p-value = 0

### Compact the model.

```
compactMdl = compact(mdl)
```

```
compactMdl =
```

Compact linear regression model:

$y \sim$  [Linear formula with 46 terms in 45 predictors]

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	3.2903	1.2333e-05	2.6679e+05	0
x1	-0.0006461	5.9019e-09	-1.0947e+05	0
x2	-0.00024739	1.0256e-08	-24121	0
x3	-9.5161e-05	1.3149e-08	-7236.9	0
x4	0.00013143	1.8311e-08	7177.3	0
x5	7.163e-05	2.3367e-08	3065.4	0
x6	4.5064e-06	2.6264e-08	171.58	0
x7	-2.6258e-05	3.006e-08	-873.51	0
x8	6.284e-05	3.0262e-08	2076.5	0
x9	-0.00014288	3.3258e-08	-4296.1	0
x10	-2.2642e-05	3.6555e-08	-619.41	0
x11	-6.0227e-05	3.7353e-08	-1612.4	0
x12	1.1665e-05	4.0048e-08	291.27	0
x13	3.8595e-05	4.203e-08	918.26	0
x14	0.00010021	4.7592e-08	2105.5	0
x15	-6.5674e-06	4.9221e-08	-133.43	0
x16	8.5598e-06	5.0296e-08	170.19	0
x17	-3.9107e-05	5.3e-08	-737.87	0
x18	-6.5841e-06	5.5355e-08	-118.94	0
x19	-1.7053e-05	5.7431e-08	-296.94	0
x20	-3.8911e-06	6.2724e-08	-62.036	0
x21	-9.7219e-06	6.3515e-08	-153.06	0
x22	-1.8749e-06	6.5388e-08	-28.673	4.6032e-176
x23	-4.7514e-06	6.6636e-08	-71.303	0
x24	-1.7756e-05	6.8495e-08	-259.23	0
x25	-9.6673e-06	7.0054e-08	-138	0
x26	7.6237e-06	7.2442e-08	105.24	0
x27	-8.4338e-07	7.7519e-08	-10.88	1.8249e-27
x28	7.0502e-06	8.1889e-08	86.094	0
x29	-1.4703e-05	8.7126e-08	-168.75	0
x30	2.7008e-05	9.0084e-08	299.8	0
x31	6.3685e-07	9.1253e-08	6.979	3.0977e-12
x32	-1.9916e-05	1.0034e-07	-198.48	0
x33	1.7369e-05	1.019e-07	170.45	0
x34	-9.931e-06	1.0706e-07	-92.764	0
x35	-1.5195e-05	1.0858e-07	-139.94	0

x36	-1.0118e-05	1.1122e-07	-90.976	0
x37	2.4595e-06	1.1254e-07	21.856	2.9315e-104
x38	-2.2928e-06	1.1493e-07	-19.95	2.0535e-87
x39	1.1397e-05	1.1855e-07	96.136	0
x40	4.0239e-06	1.2327e-07	32.643	7.75e-226
x41	-8.6667e-06	1.2535e-07	-69.142	0
x42	-8.2932e-06	1.3095e-07	-63.334	0
x43	2.7309e-06	1.3452e-07	20.301	2.0697e-90
x44	-6.9235e-06	1.3725e-07	-50.444	0
x45	1.1165e-06	1.4021e-07	7.9633	1.7956e-15

```
Number of observations: 15000, Error degrees of freedom: 14954
Root Mean Squared Error: 0.00151
R-squared: 1, Adjusted R-Squared 1
F-statistic vs. constant model: 2.82e+08, p-value = 0
```

The compact model discards the original sample data and some information related to the fitting process.

Compare the size of the full model `mdl` and the compact model `compactMdl`.

```
vars = whos('compactMdl','mdl');
[vars(1).bytes,vars(2).bytes]
```

```
ans =
```

```
83329    11410264
```

The compacted model consumes less memory than the full model.

## See Also

`CompactLinearModel` | `LinearModel` | `fitlm` | `stepwiselm`

Introduced in R2016a

# compact

**Class:** GeneralizedLinearModel

Compact generalized linear regression model

## Syntax

```
compactMdl = compact(mdl)
```

## Description

`compactMdl = compact(mdl)` returns a compact generalized linear regression model, `compactMdl`, which is the compact version of the full, fitted regression model `mdl`. The compact model uses less memory than the full model because it does not include a copy of the data or anything comparable in size to the data. However, the compact model also does not support properties (such as `Residuals`) or methods (such as `addTerms`) that require the data.

## Input Arguments

**mdl** — Full, fitted generalized linear regression model

GeneralizedLinearModel object

Full, fitted generalized linear regression model, specified as a GeneralizedLinearModel object constructed using `fitglm` or `stepwiseglm`.

## Output Arguments

**compactMdl** — Compact generalized linear regression model

CompactGeneralizedLinearModel object

Compact generalized linear regression model, returned as a CompactGeneralizedLinearModel object.

Predict response values using `compactMdl` exactly as you would using `mdl`. However, since `compactMdl` does not contain training data, you cannot perform certain tasks, such as cross-validation.

## Examples

### Compact a Generalized Linear Regression Model

Reduce the size of a full, fitted generalized linear regression model by discarding the sample data and some information related to the fitting process.

Load the data into the workspace. The simulated sample data contains 15,000 observations and 45 predictor variables.

```
load(fullfile(matlabroot, 'examples', 'stats', 'largedata4reg.mat'))
```

Fit a generalized linear regression model to the data using the first 15 predictor variables.

```
mdl = fitglm(X(:,1:15),Y)
```

```
mdl =
```

```
Generalized linear regression model:
```

```
  y ~ [Linear formula with 16 terms in 15 predictors]
```

```
  Distribution = Normal
```

```
Estimated Coefficients:
```

	Estimate	SE	tStat	pValue
(Intercept)	3.2903	0.00010447	31497	0
x1	-0.0006461	4.9991e-08	-12924	0
x2	-0.00024739	8.6874e-08	-2847.7	0
x3	-9.5161e-05	1.1138e-07	-854.38	0
x4	0.00013143	1.551e-07	847.35	0
x5	7.163e-05	1.9793e-07	361.9	0
x6	4.5064e-06	2.2247e-07	20.257	4.9539e-90
x7	-2.6258e-05	2.5462e-07	-103.13	0
x8	6.284e-05	2.5633e-07	245.15	0
x9	-0.00014288	2.817e-07	-507.19	0
x10	-2.2642e-05	3.0963e-07	-73.127	0

x11	-6.0227e-05	3.1639e-07	-190.36	0
x12	1.1665e-05	3.3921e-07	34.388	1.6995e-249
x13	3.8595e-05	3.5601e-07	108.41	0
x14	0.00010021	4.0312e-07	248.57	0
x15	-6.5674e-06	4.1692e-07	-15.752	1.844e-55

15000 observations, 14984 error degrees of freedom  
 Estimated Dispersion: 0.000164  
 F-statistic vs. constant model: 1.18e+07, p-value = 0

**Compact the model. The compact model discards the original sample data and some information related to the fitting process, so it uses less memory than the full model.**

```
compactMdl = compact(mdl)
```

```
compactMdl =
```

```
Compact generalized linear regression model:
```

```
  y ~ [Linear formula with 16 terms in 15 predictors]
```

```
  Distribution = Normal
```

```
Estimated Coefficients:
```

	Estimate	SE	tStat	pValue
(Intercept)	3.2903	0.00010447	31497	0
x1	-0.0006461	4.9991e-08	-12924	0
x2	-0.00024739	8.6874e-08	-2847.7	0
x3	-9.5161e-05	1.1138e-07	-854.38	0
x4	0.00013143	1.551e-07	847.35	0
x5	7.163e-05	1.9793e-07	361.9	0
x6	4.5064e-06	2.2247e-07	20.257	4.9539e-90
x7	-2.6258e-05	2.5462e-07	-103.13	0
x8	6.284e-05	2.5633e-07	245.15	0
x9	-0.00014288	2.817e-07	-507.19	0
x10	-2.2642e-05	3.0963e-07	-73.127	0
x11	-6.0227e-05	3.1639e-07	-190.36	0
x12	1.1665e-05	3.3921e-07	34.388	1.6995e-249
x13	3.8595e-05	3.5601e-07	108.41	0
x14	0.00010021	4.0312e-07	248.57	0
x15	-6.5674e-06	4.1692e-07	-15.752	1.844e-55

15000 observations, 14984 error degrees of freedom

```
Estimated Dispersion: 0.000164  
F-statistic vs. constant model: 1.18e+07, p-value = 0
```

## See Also

```
CompactGeneralizedLinearModel | GeneralizedLinearModel | fitglm |  
stepwiseglm
```

**Introduced in R2016b**



# compact

**Class:** RegressionEnsemble

Create compact regression ensemble

## Syntax

```
cens = compact(ens)
```

## Description

`cens = compact(ens)` creates a compact version of `ens`. You can predict regressions using `cens` exactly as you can using `ens`. However, since `cens` does not contain training data, you cannot perform some actions, such as cross validation.

## Input Arguments

**ens**

A regression ensemble created with `fitensemble`.

## Output Arguments

**cens**

A compact regression ensemble. `cens` is of class `CompactRegressionEnsemble`.

## Examples

### View Size of Compact Regression Ensemble

Compare the size of a regression ensemble for the `carsmall` data to the size of the compact version of the ensemble.

Load the `carsmall` data set and select acceleration, number of cylinders, displacement, horsepower, and vehicle weight as predictors.

```
load carsmall
X = [Acceleration Cylinders Displacement Horsepower Weight];
```

Train an ensemble of regression trees.

```
ens = fitensemble(X,MPG);
```

Create a compact version of `ens` and compare ensemble sizes.

```
cens = compact(ens);
b = whos('ens');
c = whos('cens');
[b.bytes c.bytes] % b.bytes = size of ens and c.bytes = size of cens
```

```
ans =
```

```
415490    389558
```

The compact ensemble uses less memory.

## See Also

`CompactRegressionEnsemble` | `RegressionEnsemble`

# compact

**Class:** RegressionGP

Create compact Gaussian process regression model

## Syntax

```
cgprMdl = compact(gprMdl)
```

## Description

`cgprMdl = compact(gprMdl)` returns a compact version of the trained Gaussian process regression (GPR) model, `gprMdl`.

## Input Arguments

**gprMdl** — Gaussian process regression model

RegressionGP object

Gaussian process regression model, specified as a RegressionGP object.

## Output Arguments

**cgprMdl** — Compact Gaussian process regression model

CompactRegressionGP object

Compact Gaussian process regression model, returned as a CompactRegressionGP object.

## Examples

## Compute Predictions and Regression Loss for Test Data

Generate example training data.

```
rng(1) % For reproducibility
n = 100000;
X = linspace(0,1,n)';
X = [X,X.^2];
y = 1 + X*[1;2] + sin(20*X*[1;-2]) + 0.2*randn(n,1);
```

Train a GPR model using the subset of regressors ('sr') approximation method and predict using the subset of data ('sd') method. Use 50 points in the active set and sparse greedy matrix approximation ('sgma') method for active set selection. Because the scales of the first and second predictors are different, it is good practice to standardize the data.

```
gprMdl = fitrgp(X,y,'KernelFunction','squaredExponential','FitMethod',...
'sr','PredictMethod','sd','Basis','none','ActiveSetSize',50,...
'ActiveSetMethod','sgma','Standardize',1,'KernelParameters',[1;1]);
```

`fitrgp` accepts any combination of fitting, prediction, and active set selection methods. In some cases it might not be possible to compute the standard deviations of the predicted responses, hence the prediction intervals. See “Tips” on page 32-728. And, in some cases, using the exact method might be expensive because of the size of the training data.

Create a compact GPR object.

```
cgprMdl = compact(gprMdl);
```

Generate the test data.

```
n = 4000;
Xnew = linspace(0,1,n)';
Xnew = [Xnew,Xnew.^2];
ynew = 1 + Xnew*[1;2] + sin(20*Xnew*[1;-2]) + 0.2*randn(n,1);
```

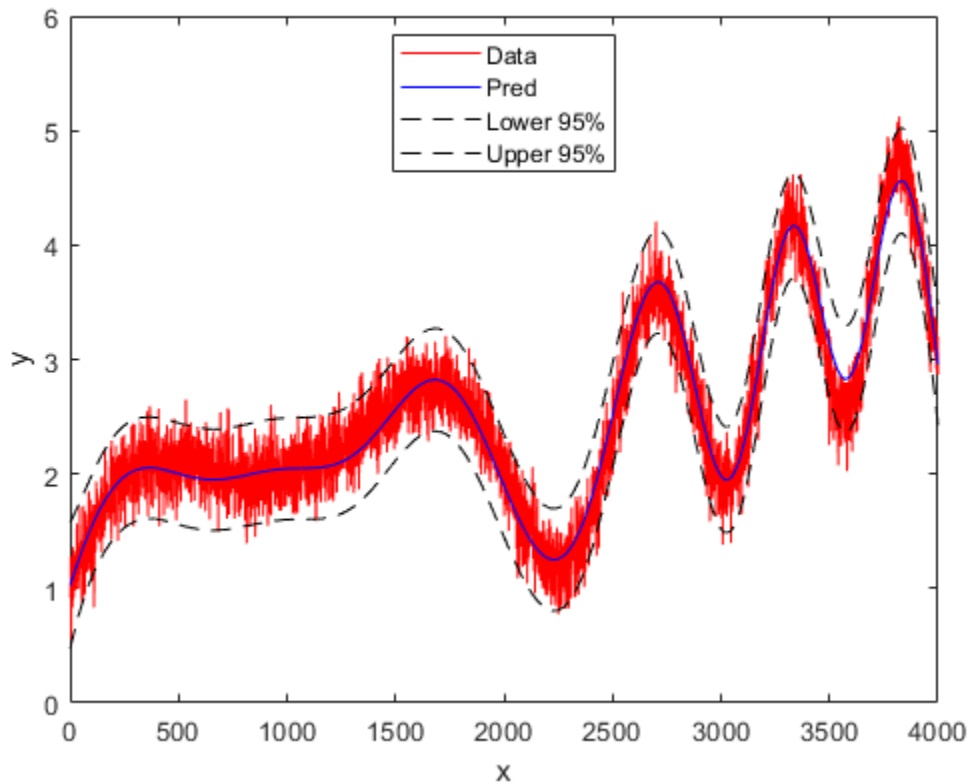
Use the compact object to predict the response in test data and the prediction intervals.

```
[ypred,~,yci] = predict(cgprMdl,Xnew);
```

Plot the true response, predicted response, and prediction intervals.

```
figure;
plot(ynew,'r');
```

```
hold on;  
plot(ypred, 'b')  
plot(yci(:,1), 'k--');  
plot(yci(:,2), 'k--');  
legend('Data', 'Pred', 'Lower 95%', 'Upper 95%', 'Location', 'Best');  
xlabel('x');  
ylabel('y');  
hold off
```



Compute the mean squared error loss on the test data using the trained GPR model.

```
L = loss(cgprMdl, Xnew, ynew)
```

```
L =
```

0.0497

## Tips

- The compact object does not contain the training data that `fitrgp` uses in training the GPR model, `gprMdl`.
- You can use the compact object to make predictions (see `predict`) or compute the regression error (see `loss`).
- Computation of standard deviations, `ysd`, and prediction intervals, `yint`, is not supported when `PredictMethod` is `'bcd'`.
- If `gprMdl` is a `CompactRegressionGP` object, you cannot compute standard deviations, `ysd`, or prediction intervals, `yint`, for `PredictMethod` equal to `'sr'` or `'fic'`. To compute `ysd` and `yint` for `PredictMethod` equal to `'sr'` or `'fic'`, use the full regression (`RegressionGP`) object.

## See Also

`CompactRegressionGP` | `fitrgp` | `loss` | `predict`

**Introduced in R2015b**

# compact

**Class:** RegressionSVM

Compact support vector machine regression model

## Syntax

```
compactMdl = compact (mdl)
```

## Description

`compactMdl = compact (mdl)` returns a compact support vector machine (SVM) regression model, `compactMdl`, which is the compact version of the full, trained SVM regression model `mdl`.

`compactMdl` does not contain the training data, whereas `mdl` contains the training data in its properties `mdl.X` and `mdl.Y`.

## Input Arguments

**mdl** — Full, trained SVM regression model

RegressionSVM model

Full, trained SVM regression model, specified as a RegressionSVM model returned by `fitrsvm`.

## Output Arguments

**compactMdl** — Compact SVM regression model

CompactRegressionSVM model

Compact SVM regression model, returned as a CompactRegressionSVM model.

Predict response values using `compactMdl` exactly as you would using `mdl`. However, since `compactMdl` does not contain training data, you cannot perform certain tasks, such as cross validation.

## Examples

### Compact an SVM Regression Model

This example shows how to reduce the size of a full, trained SVM regression model by discarding the training data and some information related to the training process.

This example uses the abalone data from the UCI Machine Learning Repository. Download the data and save it in your current directory with the name 'abalone.data'. Read the data into a table.

```
tbl = readtable('abalone.data', 'Filetype', 'text', 'ReadVariableNames', false);  
rng default % for reproducibility
```

The sample data contains 4177 observations. All of the predictor variables are continuous except for `sex`, which is a categorical variable with possible values 'M' (for males), 'F' (for females), and 'I' (for infants). The goal is to predict the number of rings on the abalone, and thereby determine its age, using physical measurements.

Train an SVM regression model using a Gaussian kernel function and an automatic kernel scale. Standardize the data.

```
mdl = fitrsvm(tbl, 'Var9', 'KernelFunction', 'gaussian', 'KernelScale', 'auto', 'Standardize'  
mdl =
```

```
RegressionSVM  
    PredictorNames: {1x8 cell}  
    ResponseName: 'Var9'  
    CategoricalPredictors: 1  
    ResponseTransform: 'none'  
        Alpha: [3635x1 double]  
        Bias: 10.8144  
    KernelParameters: [1x1 struct]  
        Mu: [1x10 double]  
        Sigma: [1x10 double]  
    NumObservations: 4177
```



```

    BoxConstraints: [4177x1 double]
    ConvergenceInfo: [1x1 struct]
    IsSupportVector: [4177x1 logical]
    Solver: 'SMO'

```

Properties, Methods

### Compact the model.

```
compactMdl = compact mdl)
```

```
compactMdl =
```

```

classreg.learning.regr.CompactRegressionSVM
    PredictorNames: {1x8 cell}
    ResponseName: 'Var9'
    CategoricalPredictors: 1
    ResponseTransform: 'none'
    Alpha: [3635x1 double]
    Bias: 10.8144
    KernelParameters: [1x1 struct]
    Mu: [1x10 double]
    Sigma: [1x10 double]
    SupportVectors: [3635x10 double]

```

Properties, Methods

The compacted model discards the training data and some information related to the training process.

Compare the size of the full model `mdl` and the compact model `compactMdl`.

```
vars = whos('compactMdl','mdl');
[vars(1).bytes,vars(2).bytes]
```

```
ans =
```

```
323793    775968
```

The compacted model consumes about half the memory of the full model.

## Reduce Memory Consumption of SVM Regression Model

This example shows how to reduce the memory consumption of a full, trained SVM regression model by compacting the model and discarding the support vectors.

Load the `carsmall` sample data.

```
load carsmall
rng default % for reproducibility
```

Train a linear SVM regression model using `Weight` as the predictor variable and `MPG` as the response variable. Standardize the data.

```
mdl = fitrsvm(Weight,MPG,'Standardize',true);
```

Note that `MPG` contains several NaN values. When training a model, `fitrsvm` will remove rows that contain NaN values from both the predictor and response data. As a result, the trained model uses only 94 of the 100 total observations contained in the sample data.

Compact the regression model to discard the training data and some information related to the training process.

```
compactMdl = compact(mdl);
```

`compactMdl` is a `CompactRegressionSVM` model that has the same parameters, support vectors, and related estimates as `mdl`, but no longer stores the training data.

Discard the support vectors and related estimates for the compacted model.

```
mdlOut = discardSupportVectors(compactMdl);
```

`mdlOut` is a `CompactRegressionSVM` model that has the same parameters as `mdl` and `compactMdl`, but no longer stores the support vectors and related estimates.

Compare the sizes of the three SVM regression models, `compactMdl`, `mdl`, and `mdlOut`.

```
vars = whos('compactMdl','mdl','mdlOut');
[vars(1).bytes,vars(2).bytes,vars(3).bytes]
```

```
ans =
```

```
3601      13727      2305
```

---

The compacted model `compactMdl` consumes 3601 bytes of memory, while the full model `mdl` consumes 13727 bytes of memory. The model `mdlOut`, which also discards the support vectors, consumes 2305 bytes of memory.

## References

- [1] Nash, W.J., T. L. Sellers, S. R. Talbot, A. J. Cawthorn, and W. B. Ford. *The Population Biology of Abalone (Haliotis species) in Tasmania. I. Blacklip Abalone (H. rubra) from the North Coast and Islands of Bass Strait*, Sea Fisheries Division, Technical Report No. 48, 1994.
- [2] Waugh, S. *Extending and benchmarking Cascade-Correlation*, Ph.D. thesis, Computer Science Department, University of Tasmania, 1995.
- [3] Clark, D., Z. Schreter, A. Adams. *A Quantitative Comparison of Dystal and Backpropagation*, submitted to the Australian Conference on Neural Networks, 1996.
- [4] Lichman, M. *UCI Machine Learning Repository*, [<http://archive.ics.uci.edu/ml>]. Irvine, CA: University of California, School of Information and Computer Science.

## See Also

`CompactRegressionSVM` | `RegressionSVM` | `fitrsvm`

**Introduced in R2015b**

## compact

**Class:** RegressionTree

Compact regression tree

## Syntax

```
ctree = compact(tree)
```

## Description

`ctree = compact(tree)` creates a compact version of `tree`.

## Input Arguments

**tree**

A regression tree created using `fitrtree`.

## Output Arguments

**ctree**

A compact regression tree. `ctree` has class `CompactRegressionTree`. You can predict regressions using `ctree` exactly as you can using `tree`. However, since `ctree` does not contain training data, you cannot perform some actions, such as cross validation.

## Examples

## Reduce Memory Consumption of Regression Tree Model

Compare the size of a full regression tree model to the compacted model.

Load the `carsmall` data set. Consider Acceleration, Displacement, Horsepower, and Weight as predictor variables.

```
load carsmall
X = [Acceleration Cylinders Displacement Horsepower Weight];
```

Grow a regression tree using the entire data set.

```
Mdl = fitrtree(X,MPG)
```

```
Mdl =
```

```
RegressionTree
    ResponseName: 'Y'
    CategoricalPredictors: []
    ResponseTransform: 'none'
    NumObservations: 94
```

`Mdl` is a `RegressionTree` model. It is a full model, that is, it stores information such as the predictor and response data `fitrtree` used in training. For a properties list of full regression tree models, see `RegressionTree`.

Create a compact version of the full regression tree. That is, one that contains enough information to make predictions only.

```
CMdl = compact(Mdl)
```

```
CMdl =
```

```
classreg.learning.regr.CompactRegressionTree
    ResponseName: 'Y'
    CategoricalPredictors: []
    ResponseTransform: 'none'
```

`CMdl` is a `CompactRegressionTree` model. For a properties list of compact regression tree models, see `CompactRegressionTree`.

Inspect the amounts of memory that the full and compact regression trees consume.

```
mdlInfo = whos('Mdl');  
cMdlInfo = whos('CMdl');  
[mdlInfo.bytes cMdlInfo.bytes]  
cMdlInfo.bytes/mdlInfo.bytes
```

```
ans =
```

```
11778      6276
```

```
ans =
```

```
0.5329
```

In this case, the compact regression tree model consumes about 25% less memory than the full model consumes.

## See Also

`CompactRegressionTree` | `RegressionTree` | `fitrtree` | `predict`

# compact

**Class:** TreeBagger

Compact ensemble of decision trees

## Description

`CMdl = compact(Mdl)` creates a compact version of `Mdl`, a `TreeBagger` model object. You can predict regressions using `CMdl` exactly as you can using `Mdl`. However, since `CMdl` does not contain training data, you cannot perform some actions, such as make out-of-bag predictions using `oobPredict`.

## Input Arguments

**Mdl**

A regression ensemble created with `TreeBagger`.

## Output Arguments

**CMdl**

A compact regression ensemble. `CMdl` is of class `CompactTreeBagger`.

## Examples

### Reduce Size of Bag of Trees

Create a compact bag of trees for efficiently making predictions on new data.

Load the `ionosphere` data set.

```
load ionosphere
```

Train a bag of 100 classification trees using all measurements and the AdaBoostM1 method.

```
Mdl = TreeBagger(100,X,Y,'Method','classification')
```

```
Mdl =
```

```
TreeBagger
Ensemble with 100 bagged decision trees:
      Training X:      [351x34]
      Training Y:      [351x1]
      Method:          classification
      NumPredictors:   34
      NumPredictorsToSample: 6
      MinLeafSize:     1
      InBagFraction:   1
      SampleWithReplacement: 1
      ComputeOOBPrediction: 0
      ComputeOOBPredictorImportance: 0
      Proximity:       []
      ClassNames:      'b'      'g'
```

Mdl is a `TreeBagger` model object that contains the training data, among other things.

Create a compact version of Mdl.

```
CMdl = compact(Mdl)
```

```
CMdl =
```

```
CompactTreeBagger
Ensemble with 100 bagged decision trees:
      Method:          classification
      NumPredictors:   34
      ClassNames:      'b' 'g'
```

CMdl is a `CompactTreeBagger` model object. CMdl is almost the same as Mdl. One exception is that it does not store the training data.



Compare the amounts of space consumed by Mdl and CMdl.

```
mdlInfo = whos('Mdl');  
cMdlInfo = whos('CMdl');  
[mdlInfo.bytes cMdlInfo.bytes]
```

```
ans =
```

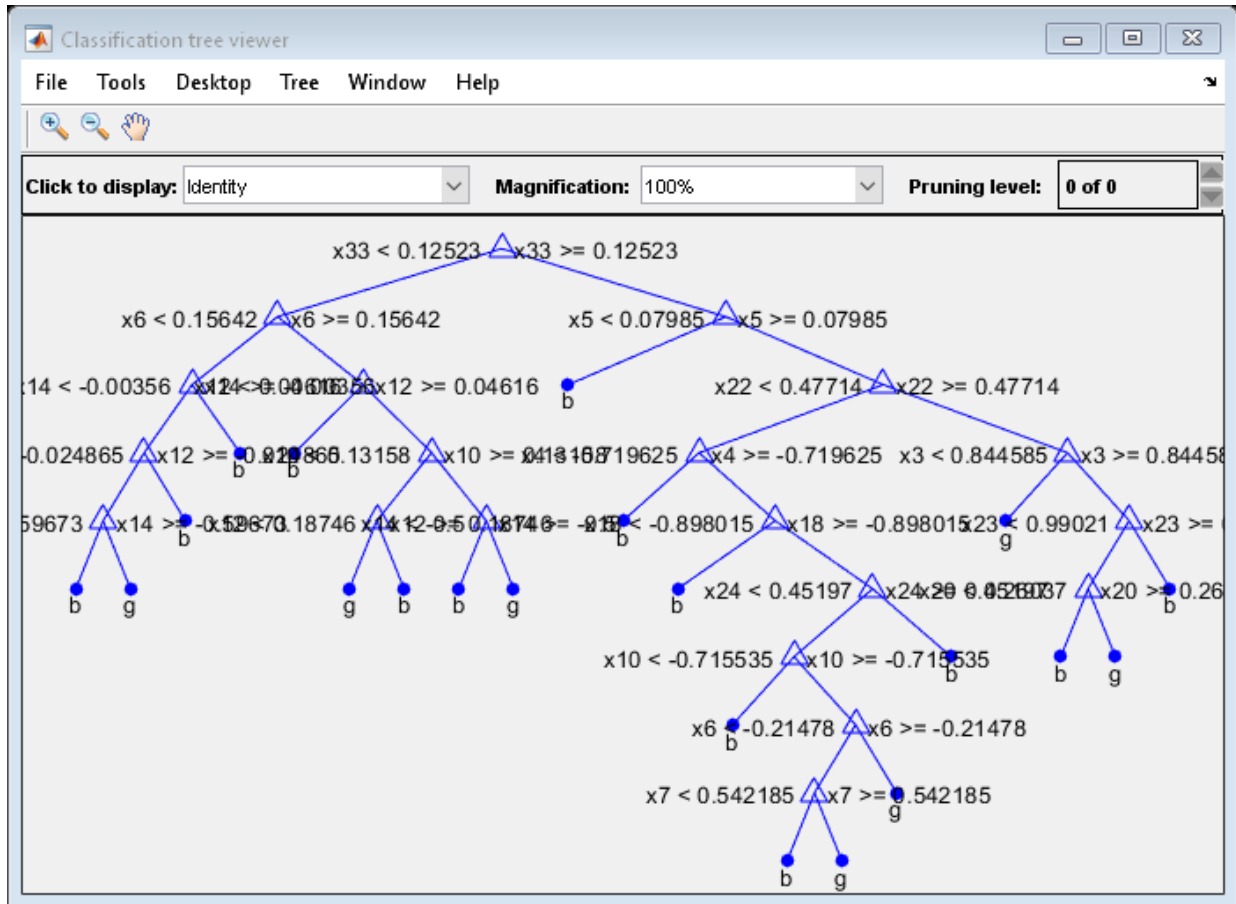
```
1031399      890204
```

Mdl consumes more space than CMdl.

CMdl.Trees stores the trained classification trees (CompactClassificationTree model objects) that compose Mdl.

Display a graph of the first tree in the compact model.

```
view(CMdl.Trees{1}, 'Mode', 'graph');
```



By default, `TreeBagger` grows deep trees.

Predict the label of the mean of  $X$  using the compact ensemble.

```
predMeanX = predict(CMdl,mean(X))
```

```
predMeanX =
```

```
1x1 cell array
```

```
{'g'}
```

- “Bootstrap Aggregation (Bagging) of Regression Trees” on page 18-141
- “Bootstrap Aggregation (Bagging) of Classification Trees” on page 18-154

## See Also

`CompactTreeBagger` | `error` | `predict`

## Topics

“Bootstrap Aggregation (Bagging) of Regression Trees” on page 18-141

“Bootstrap Aggregation (Bagging) of Classification Trees” on page 18-154

## CompactClassificationDiscriminant class

Compact discriminant analysis class

### Description

A `CompactClassificationDiscriminant` object is a compact version of a discriminant analysis classifier. The compact version does not include the data for training the classifier. Therefore, you cannot perform some tasks with a compact classifier, such as cross validation. Use a compact classifier for making predictions (classifications) of new data.

### Construction

`cobj = compact(obj)` constructs a compact classifier from a full classifier.

`cobj = makecdiscr(Mu, Sigma)` constructs a compact discriminant analysis classifier from the class means `Mu` and covariance matrix `Sigma`. For syntax details, see `makecdiscr`.

### Input Arguments

`obj`

Discriminant analysis classifier, created using `fitcdiscr`.

### Properties

**BetweenSigma**

$p$ -by- $p$  matrix, the between-class covariance, where  $p$  is the number of predictors.

**CategoricalPredictors**

Categorical predictor indices, which is always empty ( []).

**ClassNames**

List of the elements in the training data  $Y$  with duplicates removed. `ClassNames` can be a categorical array, cell array of character vectors, character array, logical vector, or a numeric vector. `ClassNames` has the same data type as the data in the argument  $Y$ .

**Coeffs**

$k$ -by- $k$  structure of coefficient matrices, where  $k$  is the number of classes. `Coeffs(i, j)` contains coefficients of the linear or quadratic boundaries between classes  $i$  and  $j$ . Fields in `Coeffs(i, j)`:

- `DiscrimType`
- `Class1` — `ClassNames(i)`
- `Class2` — `ClassNames(j)`
- `Const` — A scalar
- `Linear` — A vector with  $p$  components, where  $p$  is the number of columns in  $X$
- `Quadratic` —  $p$ -by- $p$  matrix, exists for quadratic `DiscrimType`

The equation of the boundary between class  $i$  and class  $j$  is

$$\text{Const} + \text{Linear} * x + x' * \text{Quadratic} * x = 0,$$

where  $x$  is a column vector of length  $p$ .

If `fitcdiscr` had the `FillCoeffs` name-value pair set to 'off' when constructing the classifier, `Coeffs` is empty ( []).

**Cost**

Square matrix, where `Cost(i, j)` is the cost of classifying a point into class  $j$  if its true class is  $i$  (i.e., the rows correspond to the true class and the columns correspond to the predicted class). The order of the rows and columns of `Cost` corresponds to the order of the classes in `ClassNames`. The number of rows and columns in `Cost` is the number of unique classes in the response.

Change a `Cost` matrix using dot notation: `obj.Cost = costMatrix`.

### **Delta**

Value of the `Delta` threshold for a linear discriminant model, a nonnegative scalar. If a coefficient of `obj` has magnitude smaller than `Delta`, `obj` sets this coefficient to 0, and so you can eliminate the corresponding predictor from the model. Set `Delta` to a higher value to eliminate more predictors.

`Delta` must be 0 for quadratic discriminant models.

Change `Delta` using dot notation: `obj.Delta = newDelta`.

### **DeltaPredictor**

Row vector of length equal to the number of predictors in `obj`. If `DeltaPredictor(i) < Delta` then coefficient `i` of the model is 0.

If `obj` is a quadratic discriminant model, all elements of `DeltaPredictor` are 0.

### **DiscrimType**

Character vector specifying the discriminant type. One of:

- 'linear'
- 'quadratic'
- 'diagLinear'
- 'diagQuadratic'
- 'pseudoLinear'
- 'pseudoQuadratic'

Change `DiscrimType` using dot notation: `obj.DiscrimType = newDiscrimType`.

You can change between linear types, or between quadratic types, but cannot change between linear and quadratic types.

### **Gamma**

Value of the `Gamma` regularization parameter, a scalar from 0 to 1. Change `Gamma` using dot notation: `obj.Gamma = newGamma`.

- If you set 1 for linear discriminant, the discriminant sets its type to 'diagLinear'.
- If you set a value between `MinGamma` and 1 for linear discriminant, the discriminant sets its type to 'linear'.
- You cannot set values below the value of the `MinGamma` property.
- For quadratic discriminant, you can set either 0 (for `DiscrimType` 'quadratic') or 1 (for `DiscrimType` 'diagQuadratic').

**LogDetSigma**

Logarithm of the determinant of the within-class covariance matrix. The type of `LogDetSigma` depends on the discriminant type:

- Scalar for linear discriminant analysis
- Vector of length  $K$  for quadratic discriminant analysis, where  $K$  is the number of classes

**MinGamma**

Nonnegative scalar, the minimal value of the `Gamma` parameter so that the correlation matrix is invertible. If the correlation matrix is not singular, `MinGamma` is 0.

**Mu**

Class means, specified as a  $K$ -by- $p$  matrix of scalar values class means of size.  $K$  is the number of classes, and  $p$  is the number of predictors. Each row of `Mu` represents the mean of the multivariate normal distribution of the corresponding class. The class indices are in the `ClassNames` attribute.

**PredictorNames**

Cell array of names for the predictor variables, in the order in which they appear in the training data  $X$ .

**Prior**

Numeric vector of prior probabilities for each class. The order of the elements of `Prior` corresponds to the order of the classes in `ClassNames`.

Add or change a `Prior` vector using dot notation: `obj.Prior = priorVector`.

**ResponseName**

Character vector describing the response variable  $Y$ .

**ScoreTransform**

Function handle for transforming scores, or character vector representing a built-in transformation function. 'none' means no transformation; equivalently, 'none' means  $@(x) x$ . For a list of built-in transformation functions and the syntax of custom transformation functions, see `fitcdiscr`.

Implement dot notation to add or change a `ScoreTransform` function using one of the following:

- `cobj.ScoreTransform = 'function'`
- `cobj.ScoreTransform = @function`

**Sigma**

Within-class covariance matrix or matrices. The dimensions depend on `DiscrimType`:

- 'linear' (default) — Matrix of size  $p$ -by- $p$ , where  $p$  is the number of predictors
- 'quadratic' — Array of size  $p$ -by- $p$ -by- $K$ , where  $K$  is the number of classes
- 'diagLinear' — Row vector of length  $p$
- 'diagQuadratic' — Array of size  $1$ -by- $p$ -by- $K$
- 'pseudoLinear' — Matrix of size  $p$ -by- $p$
- 'pseudoQuadratic' — Array of size  $p$ -by- $p$ -by- $K$



## Methods

compareHoldout	Compare accuracies of two classification models using new data
edge	Classification edge
logP	Log unconditional probability density for discriminant analysis classifier
loss	Classification error
mahal	Mahalanobis distance to class means
margin	Classification margins
nLinearCoeffs	Number of nonzero linear coefficients
predict	Predict labels using discriminant analysis classification model

## Copy Semantics

Value. To learn how value classes affect copy operations, see [Copying Objects \(MATLAB\)](#).

## Examples

### Construct a Compact Discriminant Analysis Classifier

Load the sample data.

```
load fisheriris
```

Construct a discriminant analysis classifier for the sample data.

```
fullobj = fitcdiscr(meas, species);
```

Construct a compact discriminant analysis classifier, and compare its size to that of the full classifier.

```
cobj = compact(fullobj);
b = whos('fullobj'); % b.bytes = size of fullobj
c = whos('cobj'); % c.bytes = size of cobj
[b.bytes c.bytes] % shows cobj uses 60% of the memory
```

```
ans =  
  
    18279    11642
```

The compact classifier is smaller than the full classifier.

### Construct Classifier Using Means and Covariances

Construct a compact discriminant analysis classifier from the means and covariances of the Fisher iris data.

```
load fisheriris  
mu(1,:) = mean(meas(1:50,:));  
mu(2,:) = mean(meas(51:100,:));  
mu(3,:) = mean(meas(101:150,:));  
  
mm1 = repmat(mu(1,:),50,1);  
mm2 = repmat(mu(2,:),50,1);  
mm3 = repmat(mu(3,:),50,1);  
cc = meas;  
cc(1:50,:) = cc(1:50,:) - mm1;  
cc(51:100,:) = cc(51:100,:) - mm2;  
cc(101:150,:) = cc(101:150,:) - mm3;  
sigstar = cc' * cc / 147;  
cpct = makecdiscr(mu,sigstar,...  
    'ClassNames',{'setosa','versicolor','virginica'});
```

## Definitions

### Discriminant Classification

The model for discriminant analysis is:

- Each class ( $Y$ ) generates data ( $X$ ) using a multivariate normal distribution. That is, the model assumes  $X$  has a Gaussian mixture distribution (`gmdistribution`).
- For linear discriminant analysis, the model has the same covariance matrix for each class, only the means vary.

- For quadratic discriminant analysis, both means and covariances of each class vary.

predict classifies so as to minimize the expected classification cost:

$$\hat{y} = \arg \min_{y=1, \dots, K} \sum_{k=1}^K \hat{P}(k | x) C(y | k),$$

where

- $\hat{y}$  is the predicted classification.
- $K$  is the number of classes.
- $\hat{P}(k | x)$  is the posterior probability on page 20-8 of class  $k$  for observation  $x$ .
- $C(y | k)$  is the cost on page 20-9 of classifying an observation as  $y$  when its true class is  $k$ .

For details, see “Prediction Using Discriminant Analysis Models” on page 20-8.

## Regularization

Regularization is the process of finding a small set of predictors that yield an effective predictive model. For linear discriminant analysis, there are two parameters,  $\gamma$  and  $\delta$ , that control regularization as follows. `cvshrink` helps you select appropriate values of the parameters.

Let  $\Sigma$  represent the covariance matrix of the data  $X$ , and let  $\hat{X}$  be the centered data (the data  $X$  minus the mean by class). Define

$$D = \text{diag}(\hat{X}^T * \hat{X}).$$

The regularized covariance matrix  $\tilde{\Sigma}$  is

$$\tilde{\Sigma} = (1 - \gamma)\Sigma + \gamma D.$$

Whenever  $\gamma \geq \text{MinGamma}$ ,  $\tilde{\Sigma}$  is nonsingular.

Let  $\mu_k$  be the mean vector for those elements of  $X$  in class  $k$ , and let  $\mu_0$  be the global mean vector (the mean of the rows of  $X$ ). Let  $C$  be the correlation matrix of the data  $X$ , and let

$\tilde{C}$  be the regularized correlation matrix:

$$\tilde{C} = (1 - \gamma)C + \gamma I,$$

where  $I$  is the identity matrix.

The linear term in the regularized discriminant analysis classifier for a data point  $x$  is

$$(x - \mu_0)^T \tilde{\Sigma}^{-1} (\mu_k - \mu_0) = \left[ (x - \mu_0)^T D^{-1/2} \right] \left[ \tilde{C}^{-1} D^{-1/2} (\mu_k - \mu_0) \right].$$

The parameter  $\delta$  enters into this equation as a threshold on the final term in square

brackets. Each component of the vector  $\left[ \tilde{C}^{-1} D^{-1/2} (\mu_k - \mu_0) \right]$  is set to zero if it is smaller in magnitude than the threshold  $\delta$ . Therefore, for class  $k$ , if component  $j$  is thresholded to zero, component  $j$  of  $x$  does not enter into the evaluation of the posterior probability.

The `DeltaPredictor` property is a vector related to this threshold. When

$\delta \geq \text{DeltaPredictor}(i)$ , all classes  $k$  have

$$\left| \tilde{C}^{-1} D^{-1/2} (\mu_k - \mu_0) \right| \leq \delta.$$

Therefore, when  $\delta \geq \text{DeltaPredictor}(i)$ , the regularized classifier does not use predictor  $i$ .

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Only the `predict` function supports code generation.
- When you train a discriminant analysis model by using `fitcdiscr`, the following restrictions apply.

- The class labels input argument value (Y) cannot be a categorical array.
- The value of the 'ClassNames' name-value pair argument cannot be a categorical array.
- The value of the 'ScoreTransform' name-value pair argument cannot be an anonymous function.

## See Also

`ClassificationDiscriminant` | `compact` | `fitcdiscr` | `makecdiscr` | `predict`

## Topics

“Discriminant Analysis Classification” on page 20-2

**Introduced in R2011b**

## CompactClassificationECOC class

Compact multiclass model for support vector machines or other classifiers

### Description

`CompactClassificationECOC` is a compact, multiclass, error-correcting output codes (ECOC) model.

The compact classifier does not include the data used for training the ECOC multiclass model. Therefore, you cannot perform tasks, such as cross-validation, using the compact classifier.

Use a compact ECOC multiclass model for labeling new data (in other words, predicting the labels of new data).

### Construction

`CompactMdl = compact(Mdl)` returns a compact, multiclass ECOC model (`CompactModel`) from a full, trained, multiclass, ECOC model (`Mdl`).

`CompactMdl = fitcecoc(X,Y,'Learners',t)` returns a compact ECOC model composed of linear classification models (`ClassificationLinear` model objects). `t` must be the character vector `'linear'` or a linear-classification-model template object returned by `templateLinear`. For more details, see `fitcecoc`.

### Input Arguments

**`mdl` — Full, trained, multiclass, ECOC model**

`ClassificationECOC` classifier

Full, trained, multiclass, ECOC model, specified as a `ClassificationECOC` classifier trained by `fitcecoc`.

**x — Predictor data**

full matrix | sparse matrix

Predictor data, specified as an  $n$ -by- $p$  full or sparse matrix.

The length of  $Y$  and the number of observations in  $X$  must be equal.

**y — Class labels**

categorical array | character array | logical vector | vector of numeric values | cell array of character vectors

Class labels to which the classification linear model is trained, specified as a categorical or character array, logical or numeric vector, or cell array of character vectors.

If  $Y$  is a character array, each element must correspond to one row of the array. The length of  $Y$  and the number of observations in  $X$  must be equal.

Data Types: char | cell | categorical | logical | single | double

## Properties

**BinaryLearners — Trained binary learners**

cell vector of model objects

Trained binary learners, specified as a cell vector of model objects. `BinaryLearners` has as many elements as classes in  $Y$ .

The software trained `BinaryLearner{j}` according to the binary problem specified by `CodingMatrix(:,j)`. For example, for multiclass learning using SVM learners, each element of `BinaryLearners` is a `CompactClassificationSVM` classifier.

Data Types: cell

**BinaryLoss — Binary learner loss function**

character vector representing the loss function name

Binary learner loss function, specified as a character vector representing the loss function name.

If you train using binary learners that use different loss functions, then the software sets `BinaryLoss` to 'hamming'. To potentially increase accuracy, set a different binary loss

function than this default during prediction or loss computation using the `BinaryLoss` name-value pair argument of `predict` or `loss`.

Data Types: `char`

### **CategoricalPredictors** — Indices of categorical predictors

vector of positive integers

Indices of categorical predictors, stored as a vector of positive integers. `CategoricalPredictors` contains index values corresponding to the columns of the predictor data that contain categorical predictors. If none of the predictors are categorical, then this property is empty (`[]`).

Data Types: `single` | `double`

### **ClassNames** — Unique class labels

categorical array | character array | logical vector | vector of numeric values | cell array of character vectors

Unique class labels used in training, specified as a categorical or character array, logical or numeric vector, or cell array of character vectors. `ClassNames` has the same data type as class labels (`Y`). `ClassNames` also determines class order.

Data Types: `categorical` | `char` | `logical` | `single` | `double` | `cell`

### **CodingMatrix** — Class-assignment codes

numeric matrix

Class-assignment codes for the binary learners, specified as a numeric matrix. `CodingMatrix` is a  $K$ -by- $L$  matrix, where  $K$  is the number of classes and  $L$  is the number of binary learners.

Elements of `CodingMatrix` are -1, 0, or 1, and the value corresponds to a dichotomous class assignment. This table describes the meaning of `CodingMatrix(i, j)`, that is, the class that learner  $j$  assigns to observations in class  $i$ .

Value	Dichotomous Class Assignment
-1	Negative class
0	Before training, learner $j$ removes observations in class $i$ from the data set.
1	Positive class



Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64`

### **Cost** — Misclassification costs

square numeric matrix

This property is read-only.

Misclassification costs, specified as a square numeric matrix. `Cost` has  $K$  rows and columns, where  $K$  is the number of classes.

`Cost(i, j)` is the cost of classifying a point into class  $j$  if its true class is  $i$ . The order of the rows and columns of `Cost` corresponds to the order of the classes in `ClassNames`.

`fitcecoc` incorporates misclassification costs differently among different types of binary learners.

Data Types: `double`

### **ExpandedPredictorNames** — Expanded predictor names

cell array of character vectors

Expanded predictor names, stored as a cell array of character vectors.

If the model uses encoding for categorical variables, then `ExpandedPredictorNames` includes the names that describe the expanded variables. Otherwise, `ExpandedPredictorNames` is the same as `PredictorNames`.

### **LearnerWeights** — Binary learner weights

numeric row vector

Binary learner weights, specified as a numeric row vector. `LearnerWeights` has length equal to the number of binary learners (`size(CodingMatrix, 2)`).

`LearnerWeights(j)` is the sum of the observation weights that binary learner  $j$  used to train its classifier.

The software uses `LearnerWeights` to fit posterior probabilities by minimizing the Kullback-Leibler divergence.

Data Types: `double` | `single`

### **PredictorNames** — Predictor names

cell array of character vectors

Predictor names in their order of appearance in predictor data ( $X$ ), specified as a cell array of character vectors. `PredictorNames` has length equal to the number of columns in  $X$ .

Data Types: `cell`

#### **Prior** — Prior class probabilities

numeric vector

This property is read-only.

Prior class probabilities, specified as a numeric vector. `Prior` has as many elements as classes in `ClassNames`, and the order of the elements corresponds to the elements of `ClassNames`.

`fitcecoc` incorporates misclassification costs differently among different types of binary learners.

Data Types: `double`

#### **ResponseName** — Response variable name

character array

Response variable name, specified as a character array.

Data Types: `char`

#### **ScoreTransform** — Score transformation function to apply to predicted scores

character vector | function handle

Score transformation function to apply to predicted scores, specified as a character vector or function handle.

To change the score transformation function to, for example, *function*, use dot notation.

- For a built-in function, enter this code and replace *function* with a character vector in the table.

```
Mdl.ScoreTransform = 'function';
```

Value	Description
'doublelogit'	$1/(1 + e^{-2x})$

Value	Description
'invlogit'	$\log(x / (1-x))$
'ismax'	Set the score for the class with the largest score to 1, and set the scores for all other classes to 0.
'logit'	$1/(1 + e^{-x})$
'none' or 'identity'	$x$ (no transformation)
'sign'	-1 for $x < 0$ 0 for $x = 0$ 1 for $x > 0$
'symmetric'	$2x - 1$
'symmetricismax'	Set the score for the class with the largest score to 1, and set the scores for all other classes to -1.
'symmetriclogit'	$2/(1 + e^{-x}) - 1$

- For a MATLAB function, or a function that you define, enter its function handle.

```
Mdl.ScoreTransform = @function;
```

*function* must accept a matrix (the original scores) and return a matrix of the same size (the transformed scores).

Data Types: char | function\_handle

## Methods

<code>compareHoldout</code>	Compare accuracies of two classification models using new data
<code>discardSupportVectors</code>	Discard support vectors of linear support vector machine binary learners
<code>edge</code>	Classification edge for multiclass, error-correcting output codes model
<code>loss</code>	Classification loss for multiclass, error-correcting output codes model
<code>margin</code>	Classification margins for multiclass, error-correcting output codes model
<code>predict</code>	Predict labels using multiclass, error-correcting output codes model
<code>selectModels</code>	Choose subset of regularized linear classification models

## Copy Semantics

Value. To learn how value classes affect copy operations, see [Copying Objects \(MATLAB\)](#).

## Examples

### Reduce the Size of Full ECOC Models

Full ECOC models (i.e., `ClassificationECOC` classifiers) hold the training data. For efficiency, you might not want to predict new labels using a large classifier.

Load Fisher's iris data set.

```
load fisheriris
X = meas;
Y = species;
```

Train an ECOC model using default SVMs as binary learners.

```
Mdl = fitcecoc(X,Y)
```

```
Mdl =

ClassificationECOC
    ResponseName: 'Y'
    CategoricalPredictors: []
        ClassNames: {'setosa' 'versicolor' 'virginica'}
    ScoreTransform: 'none'
    BinaryLearners: {3x1 cell}
    CodingName: 'onevsone'
```

Mdl is a ClassificationECOC model.

Reduce the size of the ECOC model.

```
CMdl = compact(Mdl)
```

```
CMdl =

classreg.learning.classif.CompactClassificationECOC
    ResponseName: 'Y'
    CategoricalPredictors: []
        ClassNames: {'setosa' 'versicolor' 'virginica'}
    ScoreTransform: 'none'
    BinaryLearners: {3x1 cell}
    CodingMatrix: [3x3 double]
```

CMdl is a CompactClassificationECOC model.

Display how much memory each classifier consumes.

```
whos('Mdl', 'CMdl')
```

Name	Size	Bytes	Class
CMdl	1x1	12764	classreg.learning.classif.CompactClassificationECOC
Mdl	1x1	25337	ClassificationECOC

The full ECOC model (Mdl) is almost twice the size of the compact ECOC model (CMdl).

You can remove `Mdl` from the MATLAB® Workspace, and pass `CMdl` and new predictor values to `predict` to efficiently label new observations.

### Train and Cross-Validate ECOC Classifiers

Train an ECOC classifier using different binary learners and the one-versus-all coding design. Then, cross validate the classifier.

Load Fisher's iris data set.

```
load fisheriris
X = meas;
Y = species;
classNames = unique(species(~strcmp(species, ''))) % Remove empty classes
K = numel(classNames) % Number of classes
rng(1); % For reproducibility

classNames =

3x1 cell array

    'setosa'
    'versicolor'
    'virginica'

K =

3
```

`classNames` are the unique classes in the data set, and  $K$  is the number of classes. You can use `classNames` to specify the order of the classes during training.

For a one-versus-all coding design, there are  $K = 3$  binary learners. Specify templates for the binary learners such that:

- Binary learner 1 and 2 are naive Bayes classifiers. By default, each predictor is conditionally, normally distributed given its label.
- Binary learner 3 is an SVM classifier. Specify to use the Gaussian kernel.

```
tNB = templateNaiveBayes();
tSVM = templateSVM('KernelFunction','gaussian');
tLearners = {tNB tNB tSVM};
```

tNB and tSVM are template objects for naive Bayes and SVM learning, respectively. They indicate what options to use during training. Most of their properties are empty, except for those specified using name-value pair arguments. The software fills in the empty properties with their default values during training.

Train and cross-validate an ECOC classifier using the binary learner templates and the one-versus-all coding design. Specify the order of the classes. By default, naive Bayes classifiers use posterior probabilities as scores, whereas SVM classifiers use distance from the decision boundary. Therefore, to aggregate the binary learners, you must specify fitting posterior probabilities.

```
CVMdl = fitcecoc(X,Y,'ClassNames',classNames,'CrossVal','on',...
    'Learners',tLearners,'FitPosterior',1);
```

CVMdl is not a ClassificationECOC model, but a ClassificationPartitionedECOC cross-validated model. By default, the software implements 10-fold cross-validation. The scores across the binary learners are in the same form (that is, they are posterior probabilities), so the software can aggregate the results of the binary classifications properly.

Inspect one of the trained folds using dot notation.

```
CVMdl.Trained{1}
```

```
ans =
```

```
classreg.learning.classif.CompactClassificationECOC
    ResponseName: 'Y'
    CategoricalPredictors: []
        ClassNames: {'setosa' 'versicolor' 'virginica'}
    ScoreTransform: 'none'
    BinaryLearners: {3x1 cell}
    CodingMatrix: [3x3 double]
```

Each fold is a CompactClassificationECOC model trained on 90% of the data.

You can access the results of the binary learners using dot notation and cell indexing. Display the trained SVM classifier (the third binary learner) in the first fold.

```
CVMDL.Trained{1}.BinaryLearners{3}

ans =

classreg.learning.classif.CompactClassificationSVM
    ResponseName: 'Y'
    CategoricalPredictors: []
    ClassNames: [-1 1]
    ScoreTransform: '@(S) sigmoid(S,-4.016268e+00,-3.243751e-01) '
    Alpha: [33x1 double]
    Bias: -0.1345
    KernelParameters: [1x1 struct]
    SupportVectors: [33x4 double]
    SupportVectorLabels: [33x1 double]
```

**Estimate the generalization error.**

```
genError = kfoldLoss(CVMDL)
```

```
genError =

    0.0333
```

On average, the generalization error is approximately 3%.

- “Code Generation Support, Usage Notes, and Limitations” on page 31-24
- “Code Generation Workflows” on page 31-2
- “Predict Class Labels Using MATLAB Function Block” on page 31-18
- “C Code Generation for Image Classifier”



## Algorithms

### Random Coding Design Matrices

For a given number of classes, for example,  $K$ , the software generates random coding design matrices as follows.

- 1 The software generates one of the following:
  - a Dense random — The software assigns a 1 or -1 with equal probability to each element of the  $K$ -by- $L_d$  coding design matrix, where  $L_d \approx \lceil 10 \log_2 K \rceil$ .
  - b Sparse random — The software assigns a 1 to each element of the  $K$ -by- $L_s$  coding design matrix with probability 0.25, a -1 with probability 0.25, and a 0 with probability 0.5, where  $L_s \approx \lceil 15 \log_2 K \rceil$ .
- 2 If a column does not contain at least one 1 and at least one -1, then the software removes that column.
- 3 For distinct columns  $u$  and  $v$ , if  $u = v$  or  $u = -v$ , then the software removes  $v$  from the coding design matrix.

The software randomly generates 10,000 matrices by default, and retains the matrix with the largest, minimal pairwise row distance based on the Hamming measure ([4]) given by

$$\Delta(k_1, k_2) = 0.5 \sum_{l=1}^L |m_{k_1 l} - m_{k_2 l}|,$$

where  $m_{k,l}$  is an element of coding design matrix  $j$ .

### Support Vector Storage

For linear, SVM binary learners, and for efficiency, `fitcecoc` empties the properties `Alpha`, `SupportVectorLabels`, and `SupportVectors`. `fitcecoc` lists `Beta`, rather than `Alpha`, in the model display.

To store `Alpha`, `SupportVectorLabels`, and `SupportVectors`, pass a linear, SVM template that specifies storing support vectors to `fitcecoc`. For example, enter:

```
t = templateSVM('SaveSupportVectors','on')
Mdl = fitcecoc(X,Y,'Learners',t);
```

You can remove the support vectors and related values by passing the resulting `ClassificationECOC` model to `discardSupportVectors`.

## References

- [1] Fürnkranz, Johannes. “Round Robin Classification.” *Journal of Machine Learning Research*, Vol. 2, 2002, pp. 721–747.
- [2] Escalera, S., O. Pujol, and P. Radeva. “Separability of ternary codes for sparse designs of error-correcting output codes.” *Pattern Recognition Letters*, Vol. 30, Issue 3, 2009, pp. 285–297.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Only the `predict` function supports code generation.
- When you train an ECOC model by using `fitcecoc`, the following restrictions apply.
  - You cannot fit posterior probabilities by using the `'FitPosterior'` name-value pair argument.
  - All binary learners must be either SVM classifiers or linear classification models. For the `'Learners'` name-value pair argument, you can specify:
    - `'svm'` or `'linear'`
    - An SVM template object or a cell vector of such objects (see `templateSVM`)
    - A linear classification model template object or a cell vector of such objects (see `templateLinear`)

- Code generation limitations for SVM classifiers and linear classification models also apply to ECOC classifiers, depending on the choice of binary learners. For more details, see “Code Generation” on page 32-0 of the `ClassificationSVM` class and “Code Generation” on page 32-0 of the `ClassificationLinear` class.

## See Also

`ClassificationECOC` | `compact` | `fitcecoc`

## Topics

“Code Generation Support, Usage Notes, and Limitations” on page 31-24

“Code Generation Workflows” on page 31-2

“Predict Class Labels Using MATLAB Function Block” on page 31-18

“C Code Generation for Image Classifier”

**Introduced in R2014b**

## CompactClassificationEnsemble class

Compact classification ensemble class

### Description

Compact version of a classification ensemble (of class `ClassificationEnsemble`). The compact version does not include the data for training the classification ensemble. Therefore, you cannot perform some tasks with a compact classification ensemble, such as cross validation. Use a compact classification ensemble for making predictions (classifications) of new data.

### Construction

`ens = compact(fullEns)` constructs a compact decision ensemble from a full decision ensemble.

### Input Arguments

**fullEns**

A classification ensemble created by `fitcensemble`.

### Properties

**CategoricalPredictors**

Indices of categorical predictors, stored as a vector of positive integers. `CategoricalPredictors` contains index values corresponding to the columns of the predictor data that contain categorical predictors. If none of the predictors are categorical, then this property is empty (`[]`).

**ClassNames**

List of the elements in  $Y$  with duplicates removed. `ClassNames` can be a numeric vector, vector of categorical variables, logical vector, character array, or cell array of character vectors. `ClassNames` has the same data type as the data in the argument  $Y$ .

**CombineWeights**

Character vector describing how `ens` combines weak learner weights, either 'WeightedSum' or 'WeightedAverage'.

**Cost**

Square matrix, where  $Cost(i, j)$  is the cost of classifying a point into class  $j$  if its true class is  $i$  (i.e., the rows correspond to the true class and the columns correspond to the predicted class). The order of the rows and columns of `Cost` corresponds to the order of the classes in `ClassNames`. The number of rows and columns in `Cost` is the number of unique classes in the response. This property is read-only.

**ExpandedPredictorNames**

Expanded predictor names, stored as a cell array of character vectors.

If the model uses encoding for categorical variables, then `ExpandedPredictorNames` includes the names that describe the expanded variables. Otherwise, `ExpandedPredictorNames` is the same as `PredictorNames`.

**NumTrained**

Number of trained weak learners in `ens`, a scalar.

**PredictorNames**

A cell array of names for the predictor variables, in the order in which they appear in  $X$ .

**Prior**

Numeric vector of prior probabilities for each class. The order of the elements of `Prior` corresponds to the order of the classes in `ClassNames`. The number of elements of `Prior` is the number of unique classes in the response. This property is read-only.

**ResponseName**

Character vector with the name of the response variable  $Y$ .

**ScoreTransform**

Function handle for transforming scores, or character vector representing a built-in transformation function. 'none' means no transformation; equivalently, 'none' means  $@(x)x$ . For a list of built-in transformation functions and the syntax of custom transformation functions, see `fitctree`.

Add or change a `ScoreTransform` function using dot notation:

```
ens.ScoreTransform = 'function'
```

or

```
ens.ScoreTransform = @function
```

**Trained**

A cell vector of trained classification models.

- If `Method` is 'LogitBoost' or 'GentleBoost', then `CompactClassificationEnsemble` stores trained learner  $j$  in the `CompactRegressionLearner` property of the object stored in `Trained{j}`. That is, to access trained learner  $j$ , use `ens.Trained{j}.CompactRegressionLearner`.
- Otherwise, cells of the cell vector contain the corresponding, compact classification models.

**TrainedWeights**

Numeric vector of trained weights for the weak learners in `ens`. `TrainedWeights` has  $T$  elements, where  $T$  is the number of weak learners in `learners`.

**UsePredForLearner**

Logical matrix of size  $P$ -by-`NumTrained`, where  $P$  is the number of predictors (columns) in the training data  $X$ . `UsePredForLearner(i,j)` is `true` when learner  $j$  uses predictor  $i$ , and is `false` otherwise. For each learner, the predictors have the same order as the columns in the training data  $X$ .

If the ensemble is not of type `Subspace`, all entries in `UsePredForLearner` are `true`.

## Methods

<code>compareHoldout</code>	Compare accuracies of two classification models using new data
<code>edge</code>	Classification edge
<code>loss</code>	Classification error
<code>margin</code>	Classification margins
<code>predict</code>	Predict labels using ensemble of classification models
<code>predictorImportance</code>	Estimates of predictor importance
<code>removeLearners</code>	Remove members of compact classification ensemble

## Copy Semantics

Value. To learn how value classes affect copy operations, see [Copying Objects \(MATLAB\)](#).

## Examples

### Reduce Size of Classification Ensemble

Create a compact classification ensemble for efficiently making predictions on new data.

Load the `ionosphere` data set.

```
load ionosphere
```

Train a boosted ensemble of 100 classification trees using all measurements and the `AdaBoostM1` method.

```
Mdl = fitensemble(X,Y,'AdaBoostM1',100,'tree')
```

```
Mdl =
```

```
classreg.learning.classif.ClassificationEnsemble
```

```
        ResponseName: 'Y'
CategoricalPredictors: []
        ClassNames: {'b' 'g'}
        ScoreTransform: 'none'
NumObservations: 351
NumTrained: 100
        Method: 'AdaBoostM1'
        LearnerNames: {'Tree'}
ReasonForTermination: 'Terminated normally after completing the requested number of
        FitInfo: [100x1 double]
FitInfoDescription: {2x1 cell}
```

Mdl is a `ClassificationEnsemble` model object that contains the training data, among other things.

Create a compact version of Mdl.

```
CMdl = compact(Mdl)
```

```
CMdl =
```

```
classreg.learning.classif.CompactClassificationEnsemble
    ResponseName: 'Y'
CategoricalPredictors: []
    ClassNames: {'b' 'g'}
    ScoreTransform: 'none'
    NumTrained: 100
```

CMdl is a `CompactClassificationEnsemble` model object. CMdl is almost the same as Mdl. One exception is that it does not store the training data.

Compare the amounts of space consumed by Mdl and CMdl.

```
mdlInfo = whos('Mdl');
cMdlInfo = whos('CMdl');
[mdlInfo.bytes cMdlInfo.bytes]
```

```
ans =
```



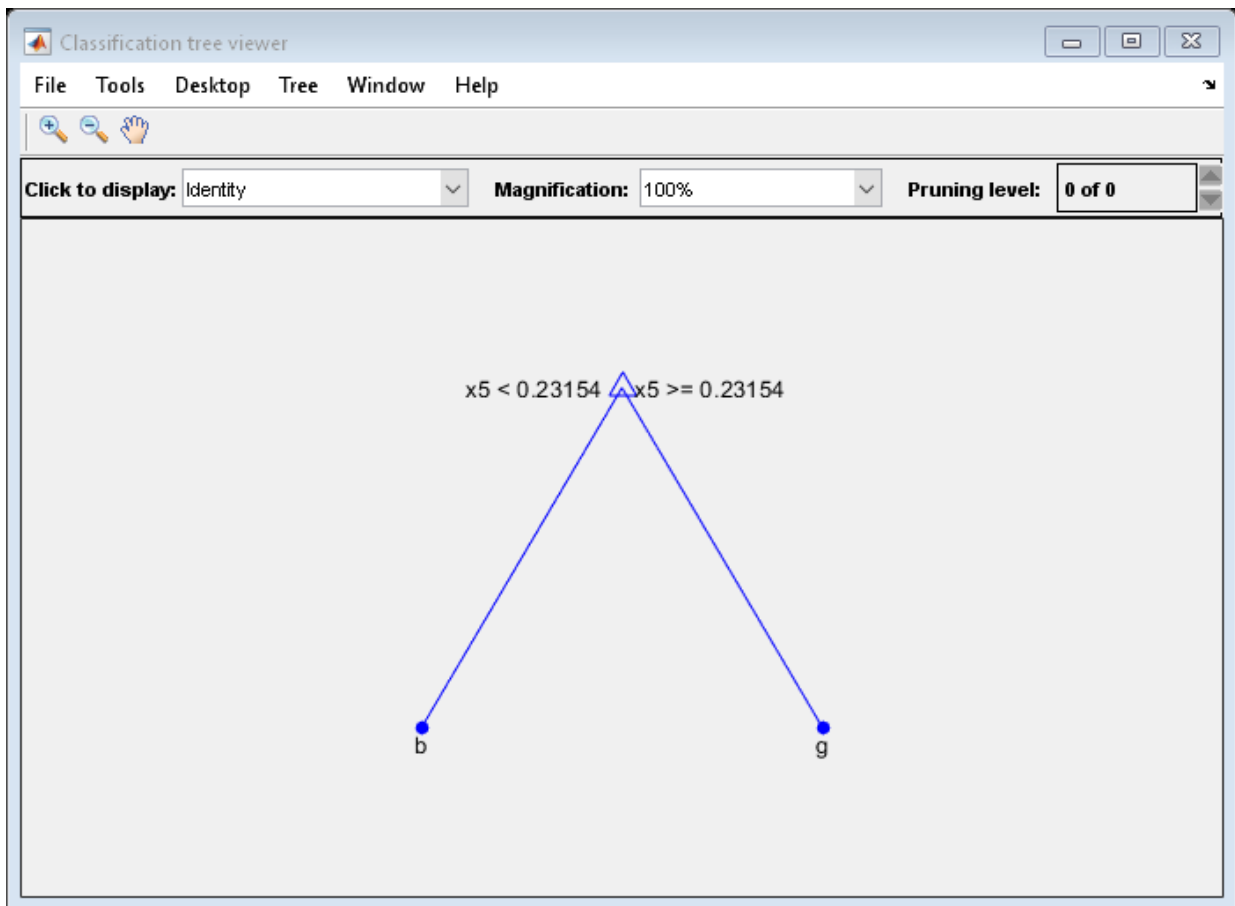
584656      339200

Mdl consumes more space than CMdl.

CMdl.Trained stores the trained classification trees (CompactClassificationTree model objects) that compose Mdl.

Display a graph of the first tree in the compact ensemble.

```
view(CMdl.Trained{1}, 'Mode', 'graph');
```



By default, `fitensemble` grows stumps for boosted ensembles of trees.

Predict the label of the mean of `X` using the compact ensemble.

```
predMeanX = predict(CMdl,mean(X))
```

```
predMeanX =  
  
    1x1 cell array  
  
    {'g'}
```

## Tip

For an ensemble of classification trees, the `Trained` property of `ens` stores an `ens.NumTrained-by-1` cell vector of compact classification models. For a textual or graphical display of tree `t` in the cell vector, enter:

- `view(ens.Trained{t}.CompactRegressionLearner)` for ensembles aggregated using `LogitBoost` or `GentleBoost`.
- `view(ens.Trained{t})` for all other aggregation methods.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Only the `predict` function supports code generation.
- When you train an ensemble by using `fitcensemble`, the following restrictions apply.
  - You can specify only 'Tree' or tree templates (see `templateTree`) as weak learners using the 'Learners' name-value pair argument.

- Code generation limitations for classification trees also apply to ensembles of classification trees. For more details, see “Code Generation” on page 32-0 of the CompactClassificationTree class.

## See Also

`ClassificationEnsemble` | `compact` | `fitcensemble` | `fitctree` | `predict` | `view`

**Introduced in R2011a**

## CompactClassificationNaiveBayes class

Compact naive Bayes classifier

### Description

`CompactClassificationNaiveBayes` is a compact naive Bayes on page 32-785 classifier.

The compact classifier does not include the data used for training the naive Bayes classifier. Therefore, you cannot perform tasks, such as cross validation, using the compact classifier.

Use a compact naive Bayes classifier to label new data (i.e., predicting the labels of new data) more efficiently.

### Construction

`CMdl = compact(Mdl)` returns a compact naive Bayes classifier (`CMdl`) from a full, trained naive Bayes classifier (`Mdl`).

### Input Arguments

**md1** — Fully trained naive Bayes classifier

`ClassificationNaiveBayes` model

A fully trained naive Bayes classifier, specified as a `ClassificationNaiveBayes` model trained by `fitcnb`.

### Properties

**CategoricalPredictors** — Indices of categorical predictors

vector of positive integers

Indices of categorical predictors, stored as a vector of positive integers. `CategoricalPredictors` contains index values corresponding to the columns of the predictor data that contain categorical predictors. If none of the predictors are categorical, then this property is empty (`[]`).

Data Types: `single` | `double`

### **CategoricalLevels — Multivariate multinomial levels**

cell vector of numeric vectors

Multivariate multinomial levels, specified as a cell vector of numeric vectors. `CategoricalLevels` has length equal to the number of predictors (`size(X, 2)`).

The cells of `CategoricalLevels` correspond to predictors that you specified as `'mvnmn'` (i.e., having a multivariate multinomial distribution) during training. Cells that do not correspond to a multivariate multinomial distribution are empty (`[]`).

If predictor  $j$  is multivariate multinomial, then `CategoricalLevels{j}` is a list of all distinct values of predictor  $j$  in the sample (NaNs removed from `unique(X(:, j))`).

Data Types: `cell`

### **ClassNames — Distinct class names**

categorical array | character array | logical vector | numeric vector | cell array of character vectors

Distinct class names, specified as a categorical or character array, logical or numeric vector, or cell vector of character vectors.

`ClassNames` is the same data type as `Y`, and has as  $K$  elements or rows for character arrays.

### **Cost — Misclassification cost**

square matrix

Misclassification cost, specified as a  $K$ -by- $K$  square matrix.

The value of `Cost(i, j)` is the cost of classifying a point into class  $j$  if its true class is  $i$ . The order of the rows and columns of `Cost` correspond to the order of the classes in `ClassNames`.

The value of `Cost` does not influence training. You can reset `Cost` after training `Mdl` using dot notation, e.g., `Mdl.Cost = [0 0.5; 1 0];`.

Data Types: `double` | `single`

### **DistributionNames** — Predictor distributions

'normal' (default) | 'kernel' | 'mn' | 'mvnm' | cell array of character vectors

Predictor distributions `fitcnb` uses to model the predictors, specified as a character vector or cell array of character vectors.

This table summarizes the available distributions.

Value	Description
'kernel'	Kernel smoothing density estimate.
'mn'	Multinomial bag-of-tokens model on page 32-466. Indicates that all predictors have this distribution.
'mvnm'	Multivariate multinomial distribution.
'normal'	Normal (Gaussian) distribution.

If `Distribution` is a 1-by- $P$  cell array of character vectors, then the software models feature  $j$  using the distribution in element  $j$  of the cell array.

Data Types: `cell` | `char`

### **DistributionParameters** — Distribution parameter estimates

cell array

Distribution parameter estimates, specified as a cell array. `DistributionParameters` is a  $K$ -by- $D$  cell array, where cell  $(k,d)$  contains the distribution parameter estimates for instances of predictor  $d$  in class  $k$ . The order of the rows corresponds to the order of the classes in the property `ClassNames`, and the order of the predictors corresponds to the order of the columns of  $X$ .

If class  $k$  has no observations for predictor  $j$ , then `Distribution{k,j}` is empty (`[]`).

The elements of `DistributionParameters` depends on the distributions of the predictors. This table describes the values in `DistributionParameters{k,j}`.

Distribution of Predictor $j$	Value
kernel	A <code>KernelDistribution</code> model. Display properties using cell indexing and dot notation. For example, to display the estimated bandwidth of the kernel density for predictor 2 in the third class, use <code>Mdl.DistributionParameters{3,2}.BandWidth</code> .
mn	A scalar representing the probability that token $j$ appears in class $k$ . For details, see “Algorithms” on page 32-467.
mvmn	A numeric vector containing the probabilities for each possible level of predictor $j$ in class $k$ . The software orders the probabilities by the sorted order of all unique levels of predictor $j$ (stored in the property <code>CategoricalLevels</code> ). For more details, see “Algorithms” on page 32-467.
normal	A 2-by-1 numeric vector. The first element is the sample mean and the second element is the sample standard deviation.

### **ExpandedPredictorNames** — Expanded predictor names

cell array of character vectors

Expanded predictor names, stored as a cell array of character vectors.

If the model uses encoding for categorical variables, then `ExpandedPredictorNames` includes the names that describe the expanded variables. Otherwise, `ExpandedPredictorNames` is the same as `PredictorNames`.

### **Kernel** — Kernel smoother types

'normal' (default) | 'box' | 'epanechnikov' | 'triangle' | cell array of character vectors

Kernel smoother types, specified as a character vector or cell array of character vectors. `Kernel` has length equal to the number of predictors (`size(X,2)`). `Kernel{j}` corresponds to predictor  $j$ , and contains a character vector describing the type of kernel

smoother. This table describes the supported kernel smoother types. Let  $I\{u\}$  denote the indicator function.

Value	Kernel	Formula
'box'	Box (uniform)	$f(x) = 0.5I\{ x  \leq 1\}$
'epanechnikov'	Epanechnikov	$f(x) = 0.75(1 - x^2)I\{ x  \leq 1\}$
'normal'	Gaussian	$f(x) = \frac{1}{\sqrt{2\pi}} \exp(-0.5x^2)$
'triangle'	Triangular	$f(x) = (1 -  x )I\{ x  \leq 1\}$

If a cell is empty (`[]`), then the software did not fit a kernel distribution to the corresponding predictor.

#### **PredictorNames** — Predictor names

character vector

Predictor names, specified as a cell vector of character vectors. The order of the elements in `PredictorNames` corresponds to the order in `X`.

Data Types: `cell`

#### **Prior** — Class prior probabilities

numeric vector

Class prior probabilities, specified as a numeric row vector. `Prior` is a 1-by- $K$  vector, and the order of its elements correspond to the elements of `ClassNames`.

`fitcnb` normalizes the prior probabilities you set using the name-value pair parameter `'Prior'` so that `sum(Prior) = 1`.

The value of `Prior` does not change the best-fitting model. Therefore, you can reset `Prior` after training `Mdl` using dot notation, e.g., `Mdl.Prior = [0.2 0.8];`.

Data Types: `double` | `single`

#### **ResponseName** — Response name

character vector



Response name, specified as a character vector.

Data Types: `char`

**ScoreTransform** — Classification score transformation function

character vector | function handle

Classification score transformation function, specified as a character vector or function handle.

To change the score transformation function to e.g., *function*, use dot notation.

- For a built-in function, enter a character vector.

```
Mdl.ScoreTransform = 'function';
```

This table lists available, built-in functions.

Value	Description
'doublelogit'	$1/(1 + e^{-2x})$
'invlogit'	$\log(x / (1-x))$
'ismax'	Set the score for the class with the largest score to 1, and set the scores for all other classes to 0.
'logit'	$1/(1 + e^{-x})$
'none' or 'identity'	$x$ (no transformation)
'sign'	-1 for $x < 0$ 0 for $x = 0$ 1 for $x > 0$
'symmetric'	$2x - 1$
'symmetricismax'	Set the score for the class with the largest score to 1, and set the scores for all other classes to -1.
'symmetriclogit'	$2/(1 + e^{-x}) - 1$

- For a MATLAB function, or a function that you define, enter its function handle.

```
Mdl.ScoreTransform = @function;
```

*function* should accept a matrix (the original scores) and return a matrix of the same size (the transformed scores).

Data Types: `char` | `function_handle`

### **Support** — Kernel smoother density support

cell vector

Kernel smoother density support, specified as a cell vector. `Support` has length equal to the number of predictors (`size(X, 2)`). The cells represent the regions to apply the kernel density.

This table describes the supported options.

Value	Description
1-by-2 numeric row vector	For example, <code>[L, U]</code> , where <code>L</code> and <code>U</code> are the finite lower and upper bounds, respectively, for the density support.
'positive'	The density support is all positive real values.
'unbounded'	The density support is all real values.

If a cell is empty (`[]`), then the software did not fit a kernel distribution to the corresponding predictor.

### **width** — Kernel smoother window width

numeric matrix

Kernel smoother window width, specified as a numeric matrix. `width` is a  $K$ -by- $P$  matrix, where  $K$  is the number of classes in the data, and  $P$  is the number of predictors (`size(X, 2)`).

`width(k, j)` is the kernel smoother window width for the kernel smoothing density of predictor  $j$  within class  $k$ . NaNs in column  $j$  indicate that the software did not fit predictor  $j$  using a kernel density.

## Methods

compareHoldout	Compare accuracies of two classification models using new data
edge	Classification edge for naive Bayes classifiers
logP	Log unconditional probability density for naive Bayes classifier
loss	Classification error for naive Bayes classifier
margin	Classification margins for naive Bayes classifiers
predict	Predict labels using naive Bayes classification model

## Copy Semantics

Value. To learn how value classes affect copy operations, see [Copying Objects \(MATLAB\)](#).

## Examples

### Reduce the Size of Naive Bayes Classifiers

Full naive Bayes classifiers (i.e., `ClassificationNaiveBayes` models) hold the training data. For efficiency, you might not want to predict new labels using a large classifier. This example shows how to reduce the size of a full naive Bayes classifier.

Load the `ionosphere` data set.

```
load ionosphere
X = X(:,3:end); % Remove two predictors for stability
```

Train a naive Bayes classifier. Assume that each predictor is conditionally, normally distributed given its label. It is good practice to specify the order of the labels.

```
Mdl = fitcnb(X,Y,'ClassNames',{'b','g'})
```

```
Mdl =
```

```
ClassificationNaiveBayes
    ResponseName: 'Y'
```

```
CategoricalPredictors: []
      ClassNames: {'b' 'g'}
      ScoreTransform: 'none'
      NumObservations: 351
      DistributionNames: {1x32 cell}
DistributionParameters: {2x32 cell}
```

Mdl is a ClassificationNaiveBayes model.

Reduce the size of the naive Bayes classifier.

```
CMdl = compact(Mdl)
```

```
CMdl =
```

```
classreg.learning.classif.CompactClassificationNaiveBayes
      ResponseName: 'Y'
      CategoricalPredictors: []
      ClassNames: {'b' 'g'}
      ScoreTransform: 'none'
      DistributionNames: {1x32 cell}
DistributionParameters: {2x32 cell}
```

CMdl is a CompactClassificationNaiveBayes model.

Display how much memory each classifier uses.

```
whos('Mdl', 'CMdl')
```

Name	Size	Bytes	Class
CMdl	1x1	15244	classreg.learning.classif.CompactClassificationNaiveBayes
Mdl	1x1	111374	ClassificationNaiveBayes

The full naive Bayes classifier (Mdl) is much larger than the compact naive Bayes classifier (CMdl).

You can remove `Mdl` from the MATLAB® Workspace, and pass `CMdl` and new predictor values to `CompactClassificationNaiveBayes.predict` to efficiently label new observations.

## Train and Cross Validate Naive Bayes Classifiers

Load the ionosphere data set.

```
load ionosphere
X = X(:,3:end); % Remove two predictors for stability
```

Train and cross validate a naive Bayes classifier. Assume that each predictor is conditionally, normally distributed given its label. It is good practice to specify the order of the classes.

```
rng(1); % For reproducibility
CVMdl = fitcnb(X,Y,'ClassNames',{'b','g'},'CrossVal','on')
```

CVMdl =

```
classreg.learning.partition.ClassificationPartitionedModel
  CrossValidatedModel: 'NaiveBayes'
  PredictorNames: {1x32 cell}
  ResponseName: 'Y'
  NumObservations: 351
  KFold: 10
  Partition: [1x1 cvpartition]
  ClassNames: {'b' 'g'}
  ScoreTransform: 'none'
```

`CVMdl` is not a `ClassificationNaiveBayes` model, but a `ClassificationPartitionedModel` cross-validated, naive Bayes model. By default, the software implements 10-fold cross validation.

Alternatively, you can cross validate a trained `ClassificationNaiveBayes` model by passing it to `ClassificationNaiveBayes.crossval`.

Inspect one of the trained folds using dot notation.

```
CVMdl.Trained{1}

ans =

    classreg.learning.classif.CompactClassificationNaiveBayes
        ResponseName: 'Y'
        CategoricalPredictors: []
            ClassNames: {'b' 'g'}
            ScoreTransform: 'none'
            DistributionNames: {1x32 cell}
            DistributionParameters: {2x32 cell}
```

Each fold is a `CompactClassificationNaiveBayes` model trained on 90% of the data.

Estimate the generalization error.

```
genError = kfoldLoss(CVMdl)
```

```
genError =

    0.1795
```

On average, the generalization error is approximately 17%.

One way to attempt reducing an unsatisfactory generalization error is to specify different conditional distributions for the predictors, or tune the parameters of the conditional distributions.

## Definitions

### Bag-of-Tokens Model

In the bag-of-tokens model, the value of predictor  $j$  is the nonnegative number of occurrences of token  $j$  in this observation. The number of categories (bins) in this multinomial model is the number of distinct tokens, that is, the number of predictors.

## Naive Bayes

Naive Bayes is a classification algorithm that applies density estimation to the data.

The algorithm leverages Bayes theorem, and (naively) assumes that the predictors are conditionally independent, given the class. Though the assumption is usually violated in practice, naive Bayes classifiers tend to yield posterior distributions that are robust to biased class density estimates, particularly where the posterior is 0.5 (the decision boundary) [1].

Naive Bayes classifiers assign observations to the most probable class (in other words, the maximum a posteriori decision rule). Explicitly, the algorithm:

- 1 Estimates the densities of the predictors within each class.
- 2 Models posterior probabilities according to Bayes rule. That is, for all  $k = 1, \dots, K$ ,

$$\hat{P}(Y = k | X_1, \dots, X_P) = \frac{\pi(Y = k) \prod_{j=1}^P P(X_j | Y = k)}{\sum_{k=1}^K \pi(Y = k) \prod_{j=1}^P P(X_j | Y = k)},$$

where:

- $Y$  is the random variable corresponding to the class index of an observation.
  - $X_1, \dots, X_P$  are the random predictors of an observation.
  - $\pi(Y = k)$  is the prior probability that a class index is  $k$ .
- 3 Classifies an observation by estimating the posterior probability for each class, and then assigns the observation to the class yielding the maximum posterior probability.

If the predictors compose a multinomial distribution, then the posterior probability

$\hat{P}(Y = k | X_1, \dots, X_P) \propto \pi(Y = k) P_{mn}(X_1, \dots, X_P | Y = k)$ , where  $P_{mn}(X_1, \dots, X_P | Y = k)$  is the probability mass function of a multinomial distribution.

## Algorithms

- If you specify 'Distribution', 'mn' when training Mdl using `fitcnb`, then the software fits a multinomial distribution using the bag-of-tokens model on page 32-2082. The software stores the probability that token  $j$  appears in class  $k$  in the property `DistributionParameters{k, j}`. Using additive smoothing [2], the estimated probability is

$$P(\text{token } j \mid \text{class } k) = \frac{1 + c_{j|k}}{P + c_k},$$

where:

•

$$c_{j|k} = n_k \frac{\sum_{i: y_i \in \text{class } k} x_{ij} w_i}{\sum w_i};$$

which is the weighted number of occurrences of token  $j$  in class  $k$ .

- $n_k$  is the number of observations in class  $k$ .

•

$w_i$  is the weight for observation  $i$ . The software normalizes weights within a class such that they sum to the prior probability for that class.

•

$$c_k = \sum_{j=1}^P c_{j|k};$$

which is the total weighted number of occurrences of all tokens in class  $k$ .

- If you specify 'Distribution', 'mvnm' when training Mdl using `fitcnb`, then:
  - 1 For each predictor, the software collects a list of the unique levels, stores the sorted list in `CategoricalLevels`, and considers each level a bin. Each predictor/class combination is a separate, independent multinomial random variable.
  - 2 For predictor  $j$  in class  $k$ , the software counts instances of each categorical level using the list stored in `CategoricalLevels{j}`.
  - 3 The software stores the probability that predictor  $j$ , in class  $k$ , has level  $L$  in the property `DistributionParameters{k, j}`, for all levels in



`CategoricalLevels{j}`. Using additive smoothing [2], the estimated probability is

$$P(\text{predictor } j = L \mid \text{class } k) = \frac{1 + m_{j|k}(L)}{m_j + m_k},$$

where:

- 

$$m_{j|k}(L) = n_k \frac{\sum_{i: y_i \in \text{class } k} I\{x_{ij} = L\} w_i}{\sum w_i};$$

which is the weighted number of observations for which predictor  $j$  equals  $L$  in class  $k$ .

- $n_k$  is the number of observations in class  $k$ .

- 

$$I\{x_{ij} = L\} = 1 \text{ if } x_{ij} = L, 0 \text{ otherwise.}$$

- 

$w_i$  is the weight for observation  $i$ . The software normalizes weights within a class such that they sum to the prior probability for that class.

- $m_j$  is the number of distinct levels in predictor  $j$ .

- $m_k$  is the weighted number of observations in class  $k$ .

## References

- [1] Hastie, T., R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*, Second Edition. NY: Springer, 2008.
- [2] Manning, C. D., P. Raghavan, and M. Schütze. *Introduction to Information Retrieval*, NY: Cambridge University Press, 2008.

## See Also

`ClassificationNaiveBayes` | `fitcnb` | `loss` | `predict`

## Topics

“Naive Bayes Classification” on page 21-2

“Grouping Variables” on page 2-59

# CompactClassificationSVM class

Compact support vector machine for binary classification

## Description

`CompactClassificationSVM` is a compact support vector machine (SVM) classifier.

The compact classifier does not include the data used for training the SVM classifier. Therefore, you cannot perform tasks, such as cross-validation, using the compact classifier.

Use a compact SVM classifier for labeling new data (i.e., predicting the labels of new data).

## Construction

`CompactSVMModel = compact(SVMModel)` returns a compact SVM classifier (`CompactSVMModel`) from a full, trained support vector machine classifier (`SVMModel`).

## Input Arguments

### **SVMModel**

A full, trained `ClassificationSVM` classifier trained by `fitcsvm`.

## Properties

### **Alpha**

$s$ -by-1 numeric vector of trained classifier coefficients from the dual problem, that is, the estimated Lagrange multipliers.  $s$  is the number of support vectors in the trained classifier, that is, `sum(Mdl.IsSupportVector)`.

If you specify removing duplicates using `RemoveDuplicates`, then for a given set of duplicate observations that are support vectors, `Alpha` contains one coefficient corresponding to the entire set. That is, MATLAB attributes a nonzero coefficient to one observation from the set of duplicates and a coefficient of 0 to all other duplicate observations in the set.

**Beta**

Numeric vector of linear predictor coefficients. `Beta` has length equal to the number of predictors used to train the model.

If your predictor data contains categorical variables, then the software uses full dummy encoding for these variables. The software creates one dummy variable for each level of each categorical variable. `Beta` stores one value for each predictor variable, including the dummy variables. For example, if there are three predictors, one of which is a categorical variable with three levels, then `Beta` is a numeric vector containing five values.

If `KernelParameters.Function` is 'linear', then the classification score for the observation  $x$  is consistent with

$$f(x) = (x/s)' \beta + b.$$

`Mdl` stores  $\beta$ ,  $b$ , and  $s$  in the properties `Beta`, `Bias`, and `KernelParameters.Scale`, respectively.

If `KernelParameters.Function` is not 'linear', then `Beta` is empty (`[]`).

**Bias**

Scalar corresponding to the trained classifier bias term.

**CategoricalPredictors**

Indices of categorical predictors, stored as a vector of positive integers. `CategoricalPredictors` contains index values corresponding to the columns of the predictor data that contain categorical predictors. If none of the predictors are categorical, then this property is empty (`[]`).

**ClassNames**

List of elements in  $Y$  with duplicates removed. `ClassNames` has the same data type as the data in the argument  $Y$ , and therefore can be a categorical or character array, logical or numeric vector, or cell array of character vectors.

**Cost**

Square matrix, where  $\text{Cost}(i, j)$  is the cost of classifying a point into class  $j$  if its true class is  $i$ .

During training, the software updates the prior probabilities by incorporating the penalties described in the cost matrix. Therefore:

- For two-class learning,  $\text{Cost}$  always has this form:  $\text{Cost}(i, j) = 1$  if  $i \neq j$ , and  $\text{Cost}(i, j) = 0$  if  $i = j$  (i.e., the rows correspond to the true class and the columns correspond to the predicted class). The order of the rows and columns of  $\text{Cost}$  corresponds to the order of the classes in  $\text{ClassNames}$ .
- For one-class learning,  $\text{Cost} = 0$ .

This property is read-only. For more details, see Algorithms on page 32-545.

**ExpandedPredictorNames**

Expanded predictor names, stored as a cell array of character vectors.

If the model uses encoding for categorical variables, then  $\text{ExpandedPredictorNames}$  includes the names that describe the expanded variables. Otherwise,  $\text{ExpandedPredictorNames}$  is the same as  $\text{PredictorNames}$ .

**KernelParameters**

Structure array containing the kernel name and parameter values.

To display the values of  $\text{KernelParameters}$ , use dot notation, for example, `Mdl.KernelParameters.Scale` displays the scale parameter value.

The software accepts  $\text{KernelParameters}$  as inputs, and does not modify them. Alter  $\text{KernelParameters}$  by setting the appropriate name-value pair arguments when you train the SVM classifier using `fitcsvm`.

**Mu**

Numeric vector of predictor means.

If you specify `'Standardize', 1` or `'Standardize', true` when you train an SVM classifier using `fitcsvm`, then  $\text{Mu}$  has length equal to the number of predictors.

If your predictor data contains categorical variables, then the software uses full dummy encoding for these variables. The software creates one dummy variable for each level of each categorical variable. `Mu` stores one value for each predictor variable, including the dummy variables. However, the software does not standardize the columns that contain categorical variables.

If `'Standardize'` is `false` or `0`, then `Mu` is an empty vector (`[]`).

**PredictorNames**

Cell array of character vectors containing the predictor names, in the order that they appear in the training data.

**Prior**

Numeric vector of prior probabilities for each class. The order of the elements of `Prior` corresponds to the elements of `Mdl.ClassNames`.

For two-class learning, if you specify a cost matrix, then the software updates the prior probabilities by incorporating the penalties described in the cost matrix.

This property is read-only. For more details, see Algorithms on page 32-545.

**ScoreTransform**

Character vector representing a built-in transformation function, or a function handle for transforming predicted classification scores.

To change the score transformation function to, e.g., *function*, use dot notation.

- For a built-in function, enter a character vector.

```
Mdl.ScoreTransform = 'function';
```

This table contains the available, built-in functions.

Value	Description
'doublelogit'	$1/(1 + e^{-2x})$
'invlogit'	$\log(x / (1-x))$

Value	Description
'ismax'	Set the score for the class with the largest score to 1, and set the scores for all other classes to 0.
'logit'	$1/(1 + e^{-x})$
'none' or 'identity'	$x$ (no transformation)
'sign'	-1 for $x < 0$ 0 for $x = 0$ 1 for $x > 0$
'symmetric'	$2x - 1$
'symmetricismax'	Set the score for the class with the largest score to 1, and set the scores for all other classes to -1.
'symmetriclogit'	$2/(1 + e^{-x}) - 1$

- For a MATLAB function, or a function that you define, enter its function handle.

```
Mdl.ScoreTransform = @function;
```

*function* should accept a matrix (the original scores) and return a matrix of the same size (the transformed scores).

### **Sigma**

Numeric vector of predictor standard deviations.

If you specify 'Standardize', 1 or 'Standardize', true when you train the SVM classifier, then Sigma has length equal to the number of predictors.

If your predictor data contains categorical variables, then the software uses full dummy encoding for these variables. The software creates one dummy variable for each level of each categorical variable. Sigma stores one value for each predictor variable, including the dummy variables. However, the software does not standardize the columns that contain categorical variables.

If 'Standardize' is false or 0, then Sigma is an empty vector ( []).

**SupportVectors**

$s$ -by- $p$  numeric matrix containing rows of  $X$  that MATLAB considers the support vectors.  $s$  is the number of support vectors in the trained classifier, that is, `sum(Mdl.IsSupportVector)` and  $p$  is the number of predictor variables in the predictor data.

If you specify `'Standardize', 1` or `'Standardize', true`, then `SupportVectors` are the standardized rows of  $X$ .

If you specify removing duplicates using `RemoveDuplicates`, then for a given set of duplicate observations that are support vectors, `SupportVectors` contains one unique support vector.

**SupportVectorLabels**

$s$ -by-1 numeric vector of support vector class labels.  $s$  is the number of support vectors in the trained classifier, that is, `sum(Mdl.IsSupportVector)`.

A value of +1 indicates that the corresponding support vector is in the positive class (`Mdl.ClassNames{2}`). A value of -1 indicates that the corresponding support vector is in the negative class (`Mdl.ClassNames{1}`).

If you specify removing duplicates using `RemoveDuplicates`, then for a given set of duplicate observations that are support vectors, `SupportVectorLabels` contains one unique support vector label.

## Methods

<code>compareHoldout</code>	Compare accuracies of two classification models using new data
<code>discardSupportVectors</code>	Discard support vectors for linear support vector machine models
<code>edge</code>	Classification edge for support vector machine classifiers
<code>fitPosterior</code>	Fit posterior probabilities
<code>loss</code>	Classification error for support vector machine classifiers
<code>margin</code>	Classification margins for support vector machine classifiers
<code>predict</code>	Predict labels using support vector machine classification model



## Copy Semantics

Value. To learn how value classes affect copy operations, see [Copying Objects \(MATLAB\)](#).

## Examples

### Reduce the Size of Support Vector Machine Classifiers

Full SVM classifiers (i.e., `ClassificationSVM` classifiers) hold the training data. For efficiency, you might not want to predict new labels using a large classifier. This example shows how to reduce the size of a full SVM classifier.

Load the `ionosphere` data set.

```
load ionosphere
```

Train an SVM classifier. It is good practice to standardize the predictors and specify the order of the classes.

```
SVMModel = fitcsvm(X,Y,'Standardize',true,...
    'ClassNames',{'b','g'})
```

```
SVMModel =
```

```
ClassificationSVM
    ResponseName: 'Y'
  CategoricalPredictors: []
        ClassNames: {'b' 'g'}
    ScoreTransform: 'none'
  NumObservations: 351
           Alpha: [90x1 double]
           Bias: -0.1343
  KernelParameters: [1x1 struct]
           Mu: [1x34 double]
           Sigma: [1x34 double]
    BoxConstraints: [351x1 double]
  ConvergenceInfo: [1x1 struct]
  IsSupportVector: [351x1 logical]
           Solver: 'SMO'
```

SVMMModel is a ClassificationSVM classifier.

Reduce the size of the SVM classifier.

```
CompactSVMMModel = compact(SVMMModel)
```

```
CompactSVMMModel =
```

```
classreg.learning.classif.CompactClassificationSVM
    ResponseName: 'Y'
    CategoricalPredictors: []
        ClassNames: {'b' 'g'}
        ScoreTransform: 'none'
            Alpha: [90x1 double]
            Bias: -0.1343
        KernelParameters: [1x1 struct]
            Mu: [1x34 double]
            Sigma: [1x34 double]
        SupportVectors: [90x34 double]
        SupportVectorLabels: [90x1 double]
```

CompactSVMMModel is a CompactClassificationSVM classifier.

Display how much memory each classifier uses.

```
whos('SVMMModel', 'CompactSVMMModel')
```

Name	Size	Bytes	Class
CompactSVMMModel	1x1	30512	classreg.learning.classif.CompactClassificationSVM
SVMMModel	1x1	140834	ClassificationSVM

The full SVM classifier (SVMMModel) is more than four times the compact SVM classifier (CompactSVMMModel).

You can remove `SVMModel` from the MATLAB® Workspace, and pass `CompactSVMModel` and new predictor values to `predict` to efficiently label new observations.

## Train and Cross-Validate Support Vector Machine Classifiers

Load the `ionosphere` data set.

```
load ionosphere
```

Train and cross-validate an SVM classifier. It is good practice to standardize the predictors and specify the order of the classes.

```
rng(1); % For reproducibility
CVSVMModel = fitcsvm(X,Y,'Standardize',true,...
    'ClassNames',{'b','g'},'CrossVal','on')
```

```
CVSVMModel =
```

```
classreg.learning.partition.ClassificationPartitionedModel
  CrossValidatedModel: 'SVM'
  PredictorNames: {1x34 cell}
  ResponseName: 'Y'
  NumObservations: 351
  KFold: 10
  Partition: [1x1 cvpartition]
  ClassNames: {'b' 'g'}
  ScoreTransform: 'none'
```

`CVSVMModel` is not a `ClassificationSVM` classifier, but a `ClassificationPartitionedModel` cross-validated, SVM classifier. By default, the software implements 10-fold cross-validation.

Alternatively, you can cross-validate a trained `ClassificationSVM` classifier by passing it to `crossval`.

Inspect one of the trained folds using dot notation.

```
CVSVMModel.Trained{1}
```

```
ans =  
  
classreg.learning.classif.CompactClassificationSVM  
    ResponseName: 'Y'  
    CategoricalPredictors: []  
        ClassNames: {'b' 'g'}  
        ScoreTransform: 'none'  
            Alpha: [78x1 double]  
            Bias: -0.2209  
    KernelParameters: [1x1 struct]  
        Mu: [1x34 double]  
        Sigma: [1x34 double]  
    SupportVectors: [78x34 double]  
    SupportVectorLabels: [78x1 double]
```

Each fold is a `CompactClassificationSVM` classifier trained on 90% of the data.

Estimate the generalization error.

```
genError = kfoldLoss(CVSVMModel)
```

```
genError =  
  
    0.1168
```

On average, the generalization error is approximately 12%.

- Using Support Vector Machines on page 18-192

## References

- [1] Hastie, T., R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*, Second Edition. NY: Springer, 2008.
- [2] Scholkopf, B., J. C. Platt, J. C. Shawe-Taylor, A. J. Smola, and R. C. Williamson. “Estimating the Support of a High-Dimensional Distribution.” *Neural Computation.*, Vol. 13, Number 7, 2001, pp. 1443–1471.

- [3] Christianini, N., and J. C. Shawe-Taylor. *An Introduction to Support Vector Machines and Other Kernel-Based Learning Methods*. Cambridge, UK: Cambridge University Press, 2000.
- [4] Scholkopf, B. and A. Smola. *Learning with Kernels: Support Vector Machines, Regularization, Optimization and Beyond, Adaptive Computation and Machine Learning* Cambridge, MA: The MIT Press, 2002.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Only the `predict` function supports code generation.
- When you train an SVM model by using `fitcsvm`, the following restrictions apply.
  - The class labels input argument value ( $Y$ ) cannot be a categorical array.
  - Code generation does not support categorical predictors. You cannot supply training data in a table that contains at least one categorical predictor, and you cannot use the 'CategoricalPredictors' name-value pair argument. To dummy-code variables that you want treated as categorical, preprocess the categorical data by using `dummyvar` before fitting the model.
  - The value of the 'ClassNames' name-value pair argument cannot be a categorical array.
  - The value of the 'ScoreTransform' name-value pair argument cannot be an anonymous function. For generating code that predicts posterior probabilities given new observations, pass a trained SVM model to `fitPosterior` or `fitSVMPosterior`. The `ScoreTransform` property of the returned model contains an anonymous function that represents the score-to-posterior-probability function and is configured for code generation.

## See Also

`ClassificationSVM` | `compact` | `fitcsvm`

## Topics

Using Support Vector Machines on page 18-192

Understanding Support Vector Machines on page 18-186

“Code Generation Support, Usage Notes, and Limitations” on page 31-24

**Introduced in R2014a**

# CompactClassificationTree class

Compact classification tree

## Description

Compact version of a classification tree (of class `ClassificationTree`). The compact version does not include the data for training the classification tree. Therefore, you cannot perform some tasks with a compact classification tree, such as cross validation. Use a compact classification tree for making predictions (classifications) of new data.

## Construction

`ctree = compact(tree)` constructs a compact decision tree from a full decision tree.

## Input Arguments

**tree**

A decision tree constructed using `fitctree`.

## Properties

### **CategoricalPredictors**

Indices of categorical predictors, stored as a vector of positive integers. `CategoricalPredictors` contains index values corresponding to the columns of the predictor data that contain categorical predictors. If none of the predictors are categorical, then this property is empty (`[]`).

### **CategoricalSplits**

An  $n$ -by-2 cell array, where  $n$  is the number of categorical splits in `tree`. Each row in `CategoricalSplits` gives left and right values for a categorical split. For each branch

node with categorical split  $j$  based on a categorical predictor variable  $z$ , the left child is chosen if  $z$  is in `CategoricalSplits(j, 1)` and the right child is chosen if  $z$  is in `CategoricalSplits(j, 2)`. The splits are in the same order as nodes of the tree. Find the nodes for these splits by selecting 'categorical' cuts from top to bottom in the `CutType` property.

**Children**

An  $n$ -by-2 array containing the numbers of the child nodes for each node in `tree`, where  $n$  is the number of nodes. Leaf nodes have child node 0.

**ClassCount**

An  $n$ -by- $k$  array of class counts for the nodes in `tree`, where  $n$  is the number of nodes and  $k$  is the number of classes. For any node number  $i$ , the class counts `ClassCount(i, :)` are counts of observations (from the data used in fitting the tree) from each class satisfying the conditions for node  $i$ .

**ClassNames**

List of the elements in  $Y$  with duplicates removed. `ClassNames` can be a numeric vector, vector of categorical variables, logical vector, character array, or cell array of character vectors. `ClassNames` has the same data type as the data in the argument  $Y$ .

If the value of a property has at least one dimension of length  $k$ , then `ClassNames` indicates the order of the elements along that dimension (e.g., `Cost` and `Prior`).

**ClassProbability**

An  $n$ -by- $k$  array of class probabilities for the nodes in `tree`, where  $n$  is the number of nodes and  $k$  is the number of classes. For any node number  $i$ , the class probabilities `ClassProbability(i, :)` are the estimated probabilities for each class for a point satisfying the conditions for node  $i$ .

**Cost**

Square matrix, where `Cost(i, j)` is the cost of classifying a point into class  $j$  if its true class is  $i$  (i.e., the rows correspond to the true class and the columns correspond to the predicted class). The order of the rows and columns of `Cost` corresponds to the order of the classes in `ClassNames`. The number of rows and columns in `Cost` is the number of unique classes in the response. This property is read-only.



**CutCategories**

An  $n$ -by-2 cell array of the categories used at branches in `tree`, where  $n$  is the number of nodes. For each branch node  $i$  based on a categorical predictor variable  $x$ , the left child is chosen if  $x$  is among the categories listed in `CutCategories{i,1}`, and the right child is chosen if  $x$  is among those listed in `CutCategories{i,2}`. Both columns of `CutCategories` are empty for branch nodes based on continuous predictors and for leaf nodes.

`CutPoint` contains the cut points for 'continuous' cuts, and `CutCategories` contains the set of categories.

**CutPoint**

An  $n$ -element vector of the values used as cut points in `tree`, where  $n$  is the number of nodes. For each branch node  $i$  based on a continuous predictor variable  $x$ , the left child is chosen if  $x < \text{CutPoint}(i)$  and the right child is chosen if  $x \geq \text{CutPoint}(i)$ . `CutPoint` is NaN for branch nodes based on categorical predictors and for leaf nodes.

`CutPoint` contains the cut points for 'continuous' cuts, and `CutCategories` contains the set of categories.

**CutType**

An  $n$ -element cell array indicating the type of cut at each node in `tree`, where  $n$  is the number of nodes. For each node  $i$ , `CutType{i}` is:

- 'continuous' — If the cut is defined in the form  $x < v$  for a variable  $x$  and cut point  $v$ .
- 'categorical' — If the cut is defined by whether a variable  $x$  takes a value in a set of categories.
- '' — If  $i$  is a leaf node.

`CutPoint` contains the cut points for 'continuous' cuts, and `CutCategories` contains the set of categories.

**CutPredictor**

An  $n$ -element cell array of the names of the variables used for branching in each node in `tree`, where  $n$  is the number of nodes. These variables are sometimes known as *cut variables*. For leaf nodes, `CutPredictor` contains an empty character vector.

`CutPoint` contains the cut points for 'continuous' cuts, and `CutCategories` contains the set of categories.

**ExpandedPredictorNames**

Expanded predictor names, stored as a cell array of character vectors.

If the model uses encoding for categorical variables, then `ExpandedPredictorNames` includes the names that describe the expanded variables. Otherwise, `ExpandedPredictorNames` is the same as `PredictorNames`.

**IsBranchNode**

An  $n$ -element logical vector that is `true` for each branch node and `false` for each leaf node of `tree`.

**NodeClass**

An  $n$ -element cell array with the names of the most probable classes in each node of `tree`, where  $n$  is the number of nodes in the tree. Every element of this array is a character vector equal to one of the class names in `ClassNames`.

**NodeError**

An  $n$ -element vector of the errors of the nodes in `tree`, where  $n$  is the number of nodes. `NodeError(i)` is the misclassification probability for node  $i$ .

**NodeProbability**

An  $n$ -element vector of the probabilities of the nodes in `tree`, where  $n$  is the number of nodes. The probability of a node is computed as the proportion of observations from the original data that satisfy the conditions for the node. This proportion is adjusted for any prior probabilities assigned to each class.

**NodeRisk**

An  $n$ -element vector of the risk of the nodes in the tree, where  $n$  is the number of nodes. The risk for each node is the measure of impurity (Gini index or deviance) for this node weighted by the node probability. If the tree is grown by twining, the risk for each node is zero.

**NodeSize**

An  $n$ -element vector of the sizes of the nodes in `tree`, where  $n$  is the number of nodes. The size of a node is defined as the number of observations from the data used to create the tree that satisfy the conditions for the node.

**NumNodes**

The number of nodes in `tree`.

**Parent**

An  $n$ -element vector containing the number of the parent node for each node in `tree`, where  $n$  is the number of nodes. The parent of the root node is 0.

**PredictorNames**

A cell array of names for the predictor variables, in the order in which they appear in `X`.

**Prior**

Numeric vector of prior probabilities for each class. The order of the elements of `Prior` corresponds to the order of the classes in `ClassNames`. The number of elements of `Prior` is the number of unique classes in the response. This property is read-only.

**PruneAlpha**

Numeric vector with one element per pruning level. If the pruning level ranges from 0 to  $M$ , then `PruneAlpha` has  $M + 1$  elements sorted in ascending order. `PruneAlpha(1)` is for pruning level 0 (no pruning), `PruneAlpha(2)` is for pruning level 1, and so on.

**PruneList**

An  $n$ -element numeric vector with the pruning levels in each node of `tree`, where  $n$  is the number of nodes. The pruning levels range from 0 (no pruning) to  $M$ , where  $M$  is the distance between the deepest leaf and the root node.

**ResponseName**

Character vector describing the response variable `Y`.

**ScoreTransform**

Function handle for transforming scores, or character vector representing a built-in transformation function. 'none' means no transformation; equivalently, 'none' means @ (x) x. For a list of built-in transformation functions and the syntax of custom transformation functions, see `fitctree`.

Add or change a `ScoreTransform` function using dot notation:

```
ctree.ScoreTransform = 'function'  
or  
ctree.ScoreTransform = @function
```

**SurrogateCutCategories**

An  $n$ -element cell array of the categories used for surrogate splits in `tree`, where  $n$  is the number of nodes in `tree`. For each node  $k$ , `SurrogateCutCategories{k}` is a cell array. The length of `SurrogateCutCategories{k}` is equal to the number of surrogate predictors found at this node. Every element of `SurrogateCutCategories{k}` is either an empty character vector for a continuous surrogate predictor, or is a two-element cell array with categories for a categorical surrogate predictor. The first element of this two-element cell array lists categories assigned to the left child by this surrogate split and the second element of this two-element cell array lists categories assigned to the right child by this surrogate split. The order of the surrogate split variables at each node is matched to the order of variables in `SurrogateCutVar`. The optimal-split variable at this node does not appear. For nonbranch (leaf) nodes, `SurrogateCutCategories` contains an empty cell.

**SurrogateCutFlip**

An  $n$ -element cell array of the numeric cut assignments used for surrogate splits in `tree`, where  $n$  is the number of nodes in `tree`. For each node  $k$ , `SurrSurrogateCutFlip{k}` is a numeric vector. The length of `SurrogateCutFlip{k}` is equal to the number of surrogate predictors found at this node. Every element of `SurrogateCutFlip{k}` is either zero for a categorical surrogate predictor, or a numeric cut assignment for a continuous surrogate predictor. The numeric cut assignment can be either  $-1$  or  $+1$ . For every surrogate split with a numeric cut  $C$  based on a continuous predictor variable  $Z$ , the left child is chosen if  $Z < C$  and the cut assignment for this surrogate split is  $+1$ , or if  $Z \geq C$  and the cut assignment for this surrogate split is  $-1$ . Similarly, the right child is chosen if  $Z \geq C$  and the cut assignment for this surrogate split is  $+1$ , or if  $Z < C$  and the cut assignment for this surrogate split is  $-1$ . The order of the surrogate split variables at

each node is matched to the order of variables in `SurrogateCutPredictor`. The optimal-split variable at this node does not appear. For nonbranch (leaf) nodes, `SurrogateCutFlip` contains an empty array.

### **SurrogateCutPoint**

An  $n$ -element cell array of the numeric values used for surrogate splits in `tree`, where  $n$  is the number of nodes in `tree`. For each node  $k$ , `SurrogateCutPoint{k}` is a numeric vector. The length of `SurrogateCutPoint{k}` is equal to the number of surrogate predictors found at this node. Every element of `SurrogateCutPoint{k}` is either NaN for a categorical surrogate predictor, or a numeric cut for a continuous surrogate predictor. For every surrogate split with a numeric cut  $C$  based on a continuous predictor variable  $Z$ , the left child is chosen if  $Z < C$  and `SurrogateCutFlip` for this surrogate split is +1, or if  $Z \geq C$  and `SurrogateCutFlip` for this surrogate split is -1. Similarly, the right child is chosen if  $Z \geq C$  and `SurrogateCutFlip` for this surrogate split is +1, or if  $Z < C$  and `SurrogateCutFlip` for this surrogate split is -1. The order of the surrogate split variables at each node is matched to the order of variables returned by `SurrogateCutPredictor`. The optimal-split variable at this node does not appear. For nonbranch (leaf) nodes, `SurrogateCutPoint` contains an empty cell.

### **SurrogateCutType**

An  $n$ -element cell array indicating types of surrogate splits at each node in `tree`, where  $n$  is the number of nodes in `tree`. For each node  $k$ , `SurrogateCutType{k}` is a cell array with the types of the surrogate split variables at this node. The variables are sorted by the predictive measure of association with the optimal predictor in the descending order, and only variables with the positive predictive measure are included. The order of the surrogate split variables at each node is matched to the order of variables in `SurrogateCutPredictor`. The optimal-split variable at this node does not appear. For nonbranch (leaf) nodes, `SurrogateCutType` contains an empty cell. A surrogate split type can be either 'continuous' if the cut is defined in the form  $Z < V$  for a variable  $Z$  and cut point  $V$  or 'categorical' if the cut is defined by whether  $Z$  takes a value in a set of categories.

### **SurrogateCutPredictor**

An  $n$ -element cell array of the names of the variables used for surrogate splits in each node in `tree`, where  $n$  is the number of nodes in `tree`. Every element of `SurrogateCutPredictor` is a cell array with the names of the surrogate split variables at this node. The variables are sorted by the predictive measure of association with the

optimal predictor in the descending order, and only variables with the positive predictive measure are included. The optimal-split variable at this node does not appear. For nonbranch (leaf) nodes, `SurrogateCutPredictor` contains an empty cell.

### **SurrogatePredictorAssociation**

An  $n$ -element cell array of the predictive measures of association for surrogate splits in `tree`, where  $n$  is the number of nodes in `tree`. For each node  $k$ , `SurrogatePredictorAssociation{k}` is a numeric vector. The length of `SurrogatePredictorAssociation{k}` is equal to the number of surrogate predictors found at this node. Every element of `SurrogatePredictorAssociation{k}` gives the predictive measure of association between the optimal split and this surrogate split. The order of the surrogate split variables at each node is the order of variables in `SurrogateCutPredictor`. The optimal-split variable at this node does not appear. For nonbranch (leaf) nodes, `SurrogatePredictorAssociation` contains an empty cell.

## **Methods**

<code>compareHoldout</code>	Compare accuracies of two classification models using new data
<code>edge</code>	Classification edge
<code>loss</code>	Classification error
<code>margin</code>	Classification margins
<code>surrogateAssociation</code>	Mean predictive measure of association for surrogate splits in decision tree
<code>predict</code>	Predict labels using classification tree
<code>predictorImportance</code>	Estimates of predictor importance
<code>view</code>	View tree

## **Copy Semantics**

Value. To learn how value classes affect copy operations, see [Copying Objects \(MATLAB\)](#).

## **Examples**

## Construct a Compact Classification Tree

Construct a compact classification tree for the Fisher iris data.

```
load fisheriris
tree = fitctree(meas, species);
ctree = compact(tree);
```

Compare the size of the resulting tree to that of the original tree.

```
t = whos('tree'); % t.bytes = size of tree in bytes
c = whos('ctree'); % c.bytes = size of ctree in bytes
[c.bytes t.bytes]
```

```
ans =
      4572      11260
```

The compact tree is smaller than the original tree.

## Definitions

### Impurity and Node Error

ClassificationTree splits nodes based on either impurity or node error.

Impurity means one of several things, depending on your choice of the SplitCriterion name-value pair argument:

- Gini's Diversity Index (gdi) — The Gini index of a node is

$$1 - \sum_i p^2(i),$$

where the sum is over the classes  $i$  at the node, and  $p(i)$  is the observed fraction of classes with class  $i$  that reach the node. A node with just one class (a pure node) has Gini index 0; otherwise the Gini index is positive. So the Gini index is a measure of node impurity.

- Deviance ('deviance') — With  $p(i)$  defined the same as for the Gini index, the deviance of a node is

$$-\sum_i p(i) \log_2 p(i).$$

A pure node has deviance 0; otherwise, the deviance is positive.

- Twoing rule ('twoing') — Twoing is not a purity measure of a node, but is a different measure for deciding how to split a node. Let  $L(i)$  denote the fraction of members of class  $i$  in the left child node after a split, and  $R(i)$  denote the fraction of members of class  $i$  in the right child node after a split. Choose the split criterion to maximize

$$P(L)P(R) \left( \sum_i |L(i) - R(i)| \right)^2,$$

where  $P(L)$  and  $P(R)$  are the fractions of observations that split to the left and right respectively. If the expression is large, the split made each child node purer.

Similarly, if the expression is small, the split made each child node similar to each other, and hence similar to the parent node, and so the split did not increase node purity.

- Node error — The node error is the fraction of misclassified classes at a node. If  $j$  is the class with the largest number of training samples at a node, the node error is

$$1 - p(j).$$

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Only the `predict` function supports code generation.
- When you train a classification tree using `fitctree`, the following restrictions apply.



- The class labels input argument value (Y) cannot be a categorical array.
- Code generation does not support categorical predictors. You cannot supply training data in a table that contains at least one categorical predictor, and you cannot use the 'CategoricalPredictors' name-value pair argument. To dummy-code variables that you want treated as categorical, preprocess the categorical data by using `dummyvar` before fitting the model.
- The value of the 'ClassNames' name-value pair argument cannot be a categorical array.
- The value of the 'ScoreTransform' name-value pair argument cannot be an anonymous function.
- You cannot use surrogate splits, that is, the value of the 'Surrogate' name-value pair argument must be 'off'.

## See Also

`ClassificationTree` | `compact` | `fitctree`

**Introduced in R2011a**

## CompactLinearModel class

Compact linear regression model class

### Description

`CompactLinearModel` is a compact linear regression model object. It consumes less memory than a full, fitted linear regression model (`LinearModel` model) because it does not store the data used to fit the model. Because the compact model does not store the input data, you cannot use it to perform certain tasks. However, you can use a compact linear regression model to predict responses using new input data.

### Construction

`compactMdl = compact(mdl)` returns a compact linear regression model `compactMdl` from the full, fitted linear regression model `mdl`. For more information, see `compact`.

### Input Arguments

**mdl** — Full, fitted linear regression model

`LinearModel` object

Full, fitted linear regression model, specified as a `LinearModel` object.

### Properties

**CoefficientCovariance** — Covariance matrix of coefficient estimates

numeric matrix

Covariance matrix of coefficient estimates, stored as a  $p$ -by- $p$  matrix of numeric values.  $p$  is the number of coefficients in the fitted model.

**CoefficientNames** — Coefficient names

cell array of character vectors

Coefficient names, stored as a cell array of character vectors containing a label for each coefficient.

### **Coefficients** — Coefficient values

table

Coefficient values, stored as a table. `Coefficients` has one row for each coefficient and the following columns:

- `Estimate` — Estimated coefficient value
- `SE` — Standard error of the estimate
- `tStat` —  $t$  statistic for a test that the coefficient is zero
- `pValue` —  $p$ -value for the  $t$  statistic

To obtain any of these columns as a vector, index into the property using dot notation. For example, in `mdl` the estimated coefficient vector is

```
beta = mdl.Coefficients.Estimate
```

Use `coefTest` to perform other tests on the coefficients.

### **DFE** — Degrees of freedom for error

positive integer value

Degrees of freedom for error (residuals), equal to the number of observations minus the number of estimated coefficients, stored as a positive integer value.

### **Formula** — Model information

`LinearFormula` object | `NonLinearFormula` object

Model information, stored as a `LinearFormula` object or `NonLinearFormula` object. If you fit a linear or generalized linear regression model, then `Formula` is a `LinearFormula` object. If you fit a nonlinear regression model, then `Formula` is a `NonLinearFormula` object.

### **LogLikelihood** — Log likelihood

numeric value

Log likelihood of the model distribution at the response values, stored as a numeric value. The mean is fitted from the model, and other parameters are estimated as part of the model fit.

**ModelCriterion** — Criterion for model comparison

structure

Criterion for model comparison, stored as a structure with the following fields:

- AIC — Akaike information criterion
- AICc — Akaike information criterion corrected for sample size
- BIC — Bayesian information criterion
- CAIC — Consistent Akaike information criterion

To obtain any of these values as a scalar, index into the property using dot notation. For example, in a model `mdl`, the AIC value `aic` is:

```
aic = mdl.ModelCriterion.AIC
```

**MSE** — Mean squared error

numeric value

Mean squared error (residuals), stored as a numeric value. Mean square error is calculated as  $MSE = SSE / DFE$ , where  $MSE$  is the mean square error,  $SSE$  is the sum of squared errors, and  $DFE$  is the degrees of freedom.

**NumCoefficients** — Number of model coefficients

positive integer

Number of model coefficients, stored as a positive integer. `NumCoefficients` includes coefficients that are set to zero when the model terms are rank deficient.

**NumEstimatedCoefficients** — Number of estimated coefficients

positive integer

Number of estimated coefficients in the model, stored as a positive integer. `NumEstimatedCoefficients` does not include coefficients that are set to zero when the model terms are rank deficient. `NumEstimatedCoefficients` is the degrees of freedom for regression.

**NumObservations** — Number of observations

positive integer

Number of observations the fitting function used in fitting, stored as a positive integer. This is the number of observations supplied in the original table, dataset, or matrix,

minus any excluded rows (set with the `Excluded` name-value pair) or rows with missing values.

**NumPredictors** — Number of predictor variables

positive integer

Number of predictor variables used to fit the model, stored as a positive integer.

**NumVariables** — Number of variables

positive integer

Number of variables in the input data, stored as a positive integer. `NumVariables` is the number of variables in the original table or dataset, or the total number of columns in the predictor matrix and response vector when the fit is based on those arrays. It includes variables, if any, that are not used as predictors or as the response.

**PredictorNames** — Names of predictors used to fit the model

cell array of character vectors

Names of predictors used to fit the model, stored as a cell array of character vectors.

**ResponseName** — Response variable name

character vector

Response variable name, stored as a character vector.

**RMSE** — Root mean squared error

numeric value

Root mean squared error (residuals), stored as a numeric value. The root mean squared error (RMSE) is equal to  $RMSE = \sqrt{MSE}$ , where  $MSE$  is the mean squared error.

**Robust** — Robust fit information

structure

Robust fit information, stored as a structure with the following fields:

Field	Description
<code>WgtFun</code>	Robust weighting function, such as 'bisquare' (see <code>robustfit</code> )
<code>Tune</code>	Value specified for tuning parameter (can be [])

Field	Description
<code>Weights</code>	Vector of weights used in final iteration of robust fit. This field is empty for compacted <code>CompactLinearModel</code> models.

This structure is empty unless `fitlm` constructed the model using robust regression.

### **Rsquared** — R-squared value for the model

structure

R-squared value for the model, stored as a structure.

For a linear or nonlinear model, `Rsquared` is a structure with two fields:

- `Ordinary` — Ordinary (unadjusted) R-squared
- `Adjusted` — R-squared adjusted for the number of coefficients

For a generalized linear model, `Rsquared` is a structure with five fields:

- `Ordinary` — Ordinary (unadjusted) R-squared
- `Adjusted` — R-squared adjusted for the number of coefficients
- `LLR` — Log-likelihood ratio
- `Deviance` — Deviance
- `AdjGeneralized` — Adjusted generalized R-squared

The R-squared value is the proportion of total sum of squares explained by the model.

The ordinary R-squared value relates to the `SSR` and `SST` properties:

$$\text{Rsquared} = \text{SSR}/\text{SST} = 1 - \text{SSE}/\text{SST}.$$

To obtain any of these values as a scalar, index into the property using dot notation. For example, the adjusted R-squared value in `mdl` is

```
r2 = mdl.Rsquared.Adjusted
```

### **SSE** — Sum of squared errors

numeric value

Sum of squared errors (residuals), stored as a numeric value.

The Pythagorean theorem implies

$$SST = SSE + SSR.$$

### **SSR — Regression sum of squares**

numeric value

Regression sum of squares, stored as a numeric value. The regression sum of squares is equal to the sum of squared deviations of the fitted values from their mean.

The Pythagorean theorem implies

$$SST = SSE + SSR.$$

### **SST — Total sum of squares**

numeric value

Total sum of squares, stored as a numeric value. The total sum of squares is equal to the sum of squared deviations of  $y$  from  $\text{mean}(y)$ .

The Pythagorean theorem implies

$$SST = SSE + SSR.$$

### **VariableInfo — Information about input variables**

table

Information about input variables contained in `Variables`, stored as a table with one row for each model term and the following columns.

Field	Description
Class	Character vector giving variable class, such as 'double'
Range	Cell array giving variable range: <ul style="list-style-type: none"> <li>• Continuous variable — Two-element vector <math>[min, max]</math>, the minimum and maximum values</li> <li>• Categorical variable — Cell array of distinct variable values</li> </ul>
InModel	Logical vector, where <code>true</code> indicates the variable is in the model
IsCategorical	Logical vector, where <code>true</code> indicates a categorical variable

### **VariableNames — Names of variables used in fit**

cell array of character vectors

Names of variables used in fit, stored as a cell array of character vectors.

- If the fit is based on a table or dataset, this property provides the names of the variables in that table or dataset.
- If the fit is based on a predictor matrix and response vector, `VariableNames` is the values in the `VarNames` name-value pair of the fitting method.
- Otherwise the variables have the default fitting names.

## Methods

<code>anova</code>	Analysis of variance for linear model
<code>coefCI</code>	Confidence intervals of coefficient estimates of linear model
<code>coefTest</code>	Linear hypothesis test on linear regression model coefficients
<code>disp</code>	Display linear regression model
<code>feval</code>	Evaluate linear regression model prediction
<code>plotEffects</code>	Plot main effects of each predictor in linear regression model
<code>plotInteraction</code>	Plot interaction effects of two predictors in linear regression model
<code>plotSlice</code>	Plot of slices through fitted linear regression surface
<code>predict</code>	Predict response of linear regression model
<code>random</code>	Simulate responses for linear regression model

## Copy Semantics

Value. To learn how value classes affect copy operations, see [Copying Objects \(MATLAB\)](#).

## Examples

### Compact a Linear Regression Model

This example shows how to reduce the size of a full, fitted linear regression model by discarding the sample data and some information related to the fitting process.



Load the data into the workspace.

```
load(fullfile(matlabroot, 'examples', 'stats', 'largedata4reg.mat'))
```

The simulated sample data contains 15,000 observations and 45 predictor variables.

Fit a simple linear regression model to the data.

```
mdl = fitlm(X,Y)
```

```
mdl =
```

Linear regression model:

```
y ~ [Linear formula with 46 terms in 45 predictors]
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
	-----	-----	-----	-----
(Intercept)	3.2903	1.2333e-05	2.6679e+05	0
x1	-0.0006461	5.9019e-09	-1.0947e+05	0
x2	-0.00024739	1.0256e-08	-24121	0
x3	-9.5161e-05	1.3149e-08	-7236.9	0
x4	0.00013143	1.8311e-08	7177.3	0
x5	7.163e-05	2.3367e-08	3065.4	0
x6	4.5064e-06	2.6264e-08	171.58	0
x7	-2.6258e-05	3.006e-08	-873.51	0
x8	6.284e-05	3.0262e-08	2076.5	0
x9	-0.00014288	3.3258e-08	-4296.1	0
x10	-2.2642e-05	3.6555e-08	-619.41	0
x11	-6.0227e-05	3.7353e-08	-1612.4	0
x12	1.1665e-05	4.0048e-08	291.27	0
x13	3.8595e-05	4.203e-08	918.26	0
x14	0.00010021	4.7592e-08	2105.5	0
x15	-6.5674e-06	4.9221e-08	-133.43	0
x16	8.5598e-06	5.0296e-08	170.19	0
x17	-3.9107e-05	5.3e-08	-737.87	0
x18	-6.5841e-06	5.5355e-08	-118.94	0
x19	-1.7053e-05	5.7431e-08	-296.94	0
x20	-3.8911e-06	6.2724e-08	-62.036	0
x21	-9.7219e-06	6.3515e-08	-153.06	0
x22	-1.8749e-06	6.5388e-08	-28.673	4.6032e-176
x23	-4.7514e-06	6.6636e-08	-71.303	0

x24	-1.7756e-05	6.8495e-08	-259.23	0
x25	-9.6673e-06	7.0054e-08	-138	0
x26	7.6237e-06	7.2442e-08	105.24	0
x27	-8.4338e-07	7.7519e-08	-10.88	1.8249e-27
x28	7.0502e-06	8.1889e-08	86.094	0
x29	-1.4703e-05	8.7126e-08	-168.75	0
x30	2.7008e-05	9.0084e-08	299.8	0
x31	6.3685e-07	9.1253e-08	6.979	3.0977e-12
x32	-1.9916e-05	1.0034e-07	-198.48	0
x33	1.7369e-05	1.019e-07	170.45	0
x34	-9.931e-06	1.0706e-07	-92.764	0
x35	-1.5195e-05	1.0858e-07	-139.94	0
x36	-1.0118e-05	1.1122e-07	-90.976	0
x37	2.4595e-06	1.1254e-07	21.856	2.9315e-104
x38	-2.2928e-06	1.1493e-07	-19.95	2.0535e-87
x39	1.1397e-05	1.1855e-07	96.136	0
x40	4.0239e-06	1.2327e-07	32.643	7.75e-226
x41	-8.6667e-06	1.2535e-07	-69.142	0
x42	-8.2932e-06	1.3095e-07	-63.334	0
x43	2.7309e-06	1.3452e-07	20.301	2.0697e-90
x44	-6.9235e-06	1.3725e-07	-50.444	0
x45	1.1165e-06	1.4021e-07	7.9633	1.7956e-15

Number of observations: 15000, Error degrees of freedom: 14954  
 Root Mean Squared Error: 0.00151  
 R-squared: 1, Adjusted R-Squared 1  
 F-statistic vs. constant model: 2.82e+08, p-value = 0

### Compact the model.

```
compactMdl = compact(mdl)
```

```
compactMdl =
```

Compact linear regression model:

```
y ~ [Linear formula with 46 terms in 45 predictors]
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	3.2903	1.2333e-05	2.6679e+05	0

x1	-0.0006461	5.9019e-09	-1.0947e+05	0
x2	-0.00024739	1.0256e-08	-24121	0
x3	-9.5161e-05	1.3149e-08	-7236.9	0
x4	0.00013143	1.8311e-08	7177.3	0
x5	7.163e-05	2.3367e-08	3065.4	0
x6	4.5064e-06	2.6264e-08	171.58	0
x7	-2.6258e-05	3.006e-08	-873.51	0
x8	6.284e-05	3.0262e-08	2076.5	0
x9	-0.00014288	3.3258e-08	-4296.1	0
x10	-2.2642e-05	3.6555e-08	-619.41	0
x11	-6.0227e-05	3.7353e-08	-1612.4	0
x12	1.1665e-05	4.0048e-08	291.27	0
x13	3.8595e-05	4.203e-08	918.26	0
x14	0.00010021	4.7592e-08	2105.5	0
x15	-6.5674e-06	4.9221e-08	-133.43	0
x16	8.5598e-06	5.0296e-08	170.19	0
x17	-3.9107e-05	5.3e-08	-737.87	0
x18	-6.5841e-06	5.5355e-08	-118.94	0
x19	-1.7053e-05	5.7431e-08	-296.94	0
x20	-3.8911e-06	6.2724e-08	-62.036	0
x21	-9.7219e-06	6.3515e-08	-153.06	0
x22	-1.8749e-06	6.5388e-08	-28.673	4.6032e-176
x23	-4.7514e-06	6.6636e-08	-71.303	0
x24	-1.7756e-05	6.8495e-08	-259.23	0
x25	-9.6673e-06	7.0054e-08	-138	0
x26	7.6237e-06	7.2442e-08	105.24	0
x27	-8.4338e-07	7.7519e-08	-10.88	1.8249e-27
x28	7.0502e-06	8.1889e-08	86.094	0
x29	-1.4703e-05	8.7126e-08	-168.75	0
x30	2.7008e-05	9.0084e-08	299.8	0
x31	6.3685e-07	9.1253e-08	6.979	3.0977e-12
x32	-1.9916e-05	1.0034e-07	-198.48	0
x33	1.7369e-05	1.019e-07	170.45	0
x34	-9.931e-06	1.0706e-07	-92.764	0
x35	-1.5195e-05	1.0858e-07	-139.94	0
x36	-1.0118e-05	1.1122e-07	-90.976	0
x37	2.4595e-06	1.1254e-07	21.856	2.9315e-104
x38	-2.2928e-06	1.1493e-07	-19.95	2.0535e-87
x39	1.1397e-05	1.1855e-07	96.136	0
x40	4.0239e-06	1.2327e-07	32.643	7.75e-226
x41	-8.6667e-06	1.2535e-07	-69.142	0
x42	-8.2932e-06	1.3095e-07	-63.334	0
x43	2.7309e-06	1.3452e-07	20.301	2.0697e-90
x44	-6.9235e-06	1.3725e-07	-50.444	0

```
x45          1.1165e-06    1.4021e-07          7.9633    1.7956e-15
```

```
Number of observations: 15000, Error degrees of freedom: 14954  
Root Mean Squared Error: 0.00151  
R-squared: 1, Adjusted R-Squared 1  
F-statistic vs. constant model: 2.82e+08, p-value = 0
```

The compact model discards the original sample data and some information related to the fitting process.

Compare the size of the full model `mdl` and the compact model `compactMdl`.

```
vars = whos('compactMdl','mdl');  
[vars(1).bytes,vars(2).bytes]
```

```
ans =
```

```
83329    11410264
```

The compacted model consumes less memory than the full model.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Only the `predict` and `random` functions support code generation.
- When you fit a model by using `fitlm` or `stepwiselm`, you cannot supply training data in a table that contains at least one categorical predictor, and you cannot use the `'CategoricalVars'` name-value pair argument. Code generation does not support categorical predictors. To dummy-code variables that you want treated as categorical, preprocess the categorical data by using `dummyvar` before fitting the model.

## See Also

`LinearModel` | `compact` | `fitlm` | `plotPartialDependence`

## Topics

“Linear Regression” on page 11-11

**Introduced in R2016a**

## CompactGeneralizedLinearModel class

Compact generalized linear regression model class

### Description

`CompactGeneralizedLinearModel` is a compact generalized linear regression model object. It consumes less memory than a full generalized linear regression model (`GeneralizedLinearModel`) because it does not store the data used to fit the model. The compact model does not store the input data, so you cannot use it to perform certain tasks. However, you can use a compact generalized linear regression model to predict responses using new input data.

Fitting operations (`fitlm`, `fitglm`, ...) automatically use compact objects when you work with tall arrays. Fitting operations with in-memory tables and arrays produce full objects. You can use the `compact` method to make them smaller.

### Construction

`compactMdl = compact(mdl)` returns a compact generalized linear regression model `compactMdl` from the full generalized linear regression model `mdl`. For more information, see `compact`.

### Input Arguments

**mdl** — Full generalized linear regression model

`GeneralizedLinearModel` object

Full generalized linear regression model, specified as a `GeneralizedLinearModel` object.

## Properties

### **CoefficientCovariance** — Covariance matrix of coefficient estimates

numeric matrix

Covariance matrix of coefficient estimates, stored as a  $p$ -by- $p$  matrix of numeric values.  $p$  is the number of coefficients in the fitted model.

### **CoefficientNames** — Coefficient names

cell array of character vectors

Coefficient names, stored as a cell array of character vectors containing a label for each coefficient.

### **Coefficients** — Coefficient values

table

Coefficient values, stored as a table. `Coefficients` has one row for each coefficient and the following columns:

- `Estimate` — Estimated coefficient value
- `SE` — Standard error of the estimate
- `tStat` —  $t$  statistic for a test that the coefficient is zero
- `pValue` —  $p$ -value for the  $t$  statistic

To obtain any of these columns as a vector, index into the property using dot notation. For example, in `mdl` the estimated coefficient vector is

```
beta = mdl.Coefficients.Estimate
```

Use `coefTest` to perform other tests on the coefficients.

### **Deviance** — Deviance of fit

numeric value

Deviance of fit, stored as a numeric value. Deviance is useful for comparing two models when one is a special case of the other. The difference between the deviance of the two models has a chi-square distribution with degrees of freedom equal to the difference in the number of estimated parameters between the two models. For more information on deviance, see “Deviance” on page 32-2845.

**DFE — Degrees of freedom for error**

positive integer value

Degrees of freedom for error (residuals), equal to the number of observations minus the number of estimated coefficients, stored as a positive integer value.

**Dispersion — Scale factor of the variance of the response**

structure

Scale factor of the variance of the response, stored as a structure. `Dispersion` multiplies the variance function for the distribution.

For example, the variance function for the binomial distribution is  $p(1-p)/n$ , where  $p$  is the probability parameter and  $n$  is the sample size parameter. If `Dispersion` is near 1, the variance of the data appears to agree with the theoretical variance of the binomial distribution. If `Dispersion` is larger than 1, the data are “overdispersed” relative to the binomial distribution.

**DispersionEstimated — Flag to indicate use of dispersion scale factor**

logical value

Flag to indicate use of dispersion scale factor, stored as a logical value. Use `DispersionEstimated` to indicate whether `fitglm` used the `Dispersion` scale factor to compute standard errors for the coefficients in `Coefficients.SE`. If `DispersionEstimated` is false, then `fitglm` used the theoretical value of the variance.

- `DispersionEstimated` can be false only for 'binomial' or 'poisson' distributions.
- To set `DispersionEstimated`, set the `DispersionFlag` name-value pair in `fitglm`.

**Distribution — Generalized distribution information**

structure

Generalized distribution information, stored as a structure with the following fields relating to the generalized distribution.



Field	Description
Name	Name of the distribution. Options are: 'normal', 'binomial', 'poisson', 'gamma', or 'inverse gamma'.
DevianceFunction	Function that computes the components of the deviance as a function of the fitted parameter values and the response values.
VarianceFunction	Function that computes the theoretical variance for the distribution as a function of the fitted parameter values. When DispersionEstimated is true, Dispersion multiplies the variance function in the computation of the coefficient standard errors.

**Formula — Model information**

LinearFormula object | NonLinearFormula object

Model information, stored as a LinearFormula object or NonLinearFormula object. If you fit a linear or generalized linear regression model, then Formula is a LinearFormula object. If you fit a nonlinear regression model, then Formula is a NonLinearFormula object.

**Link — Link function**

structure

Link function, stored as a structure with the following fields.

Field	Description
Name	Name of the link function, or '' if you specified the link as a function handle rather than a character vector
LinkFunction	The function that defines $f$ , a function handle
DevianceFunction	Derivative of $f$ , a function handle
VarianceFunction	Inverse of $f$ , a function handle

The link is a function  $f$  that links the distribution parameter  $\mu$  to the fitted linear combination  $Xb$  of the predictors:

$$f(\mu) = Xb.$$

**LogLikelihood — Log likelihood**

numeric value

Log likelihood of the model distribution at the response values, stored as a numeric value. The mean is fitted from the model, and other parameters are estimated as part of the model fit.

**ModelCriterion** — Criterion for model comparison

structure

Criterion for model comparison, stored as a structure with the following fields:

- AIC — Akaike information criterion
- AICc — Akaike information criterion corrected for sample size
- BIC — Bayesian information criterion
- CAIC — Consistent Akaike information criterion

To obtain any of these values as a scalar, index into the property using dot notation. For example, in a model `mdl`, the AIC value `aic` is:

```
aic = mdl.ModelCriterion.AIC
```

**NumCoefficients** — Number of model coefficients

positive integer

Number of model coefficients, stored as a positive integer. `NumCoefficients` includes coefficients that are set to zero when the model terms are rank deficient.

**NumEstimatedCoefficients** — Number of estimated coefficients

positive integer

Number of estimated coefficients in the model, stored as a positive integer. `NumEstimatedCoefficients` does not include coefficients that are set to zero when the model terms are rank deficient. `NumEstimatedCoefficients` is the degrees of freedom for regression.

**NumObservations** — Number of observations

positive integer

Number of observations the fitting function used in fitting, stored as a positive integer. This is the number of observations supplied in the original table, dataset, or matrix, minus any excluded rows (set with the `Excluded` name-value pair) or rows with missing values.

**NumPredictors** — Number of predictor variables

positive integer

Number of predictor variables used to fit the model, stored as a positive integer.

**NumVariables** — Number of variables

positive integer

Number of variables in the input data, stored as a positive integer. `NumVariables` is the number of variables in the original table or dataset, or the total number of columns in the predictor matrix and response vector when the fit is based on those arrays. It includes variables, if any, that are not used as predictors or as the response.

**PredictorNames** — Names of predictors used to fit the model

cell array of character vectors

Names of predictors used to fit the model, stored as a cell array of character vectors.

**ResponseName** — Response variable name

character vector

Response variable name, stored as a character vector.

**Rsquared** — R-squared value for the model

structure

R-squared value for the model, stored as a structure.

For a linear or nonlinear model, `Rsquared` is a structure with two fields:

- `Ordinary` — Ordinary (unadjusted) R-squared
- `Adjusted` — R-squared adjusted for the number of coefficients

For a generalized linear model, `Rsquared` is a structure with five fields:

- `Ordinary` — Ordinary (unadjusted) R-squared
- `Adjusted` — R-squared adjusted for the number of coefficients
- `LLR` — Log-likelihood ratio
- `Deviance` — Deviance
- `AdjGeneralized` — Adjusted generalized R-squared

The R-squared value is the proportion of total sum of squares explained by the model. The ordinary R-squared value relates to the SSR and SST properties:

$$\text{Rsquared} = \text{SSR}/\text{SST} = 1 - \text{SSE}/\text{SST}.$$

To obtain any of these values as a scalar, index into the property using dot notation. For example, the adjusted R-squared value in `mdl` is

```
r2 = mdl.Rsquared.Adjusted
```

**SSE — Sum of squared errors**

numeric value

Sum of squared errors (residuals), stored as a numeric value.

The Pythagorean theorem implies

$$\text{SST} = \text{SSE} + \text{SSR}.$$

**SSR — Regression sum of squares**

numeric value

Regression sum of squares, stored as a numeric value. The regression sum of squares is equal to the sum of squared deviations of the fitted values from their mean.

The Pythagorean theorem implies

$$\text{SST} = \text{SSE} + \text{SSR}.$$

**SST — Total sum of squares**

numeric value

Total sum of squares, stored as a numeric value. The total sum of squares is equal to the sum of squared deviations of  $\bar{y}$  from  $\text{mean}(\bar{y})$ .

The Pythagorean theorem implies

$$\text{SST} = \text{SSE} + \text{SSR}.$$

**VariableInfo — Information about input variables**

table

Information about input variables contained in `Variables`, stored as a table with one row for each model term and the following columns.

Field	Description
<code>Class</code>	Character vector giving variable class, such as 'double'
<code>Range</code>	Cell array giving variable range: <ul style="list-style-type: none"> <li>• Continuous variable — Two-element vector <math>[min, max]</math>, the minimum and maximum values</li> <li>• Categorical variable — Cell array of distinct variable values</li> </ul>
<code>InModel</code>	Logical vector, where <code>true</code> indicates the variable is in the model
<code>IsCategorical</code>	Logical vector, where <code>true</code> indicates a categorical variable

**VariableNames** — Names of variables used in fit

cell array of character vectors

Names of variables used in fit, stored as a cell array of character vectors.

- If the fit is based on a table or dataset, this property provides the names of the variables in that table or dataset.
- If the fit is based on a predictor matrix and response vector, `VariableNames` is the values in the `VarNames` name-value pair of the fitting method.
- Otherwise the variables have the default fitting names.

## Methods

<code>coefCI</code>	Confidence intervals of coefficient estimates of generalized linear model
<code>coefTest</code>	Linear hypothesis test on generalized linear regression model coefficients
<code>devianceTest</code>	Analysis of deviance
<code>disp</code>	Display generalized linear regression model
<code>feval</code>	Evaluate generalized linear regression model prediction
<code>plotSlice</code>	Plot of slices through fitted generalized linear regression surface
<code>predict</code>	Predict response of generalized linear regression model
<code>random</code>	Simulate responses for generalized linear regression model

## Copy Semantics

Value. To learn how value classes affect copy operations, see [Copying Objects \(MATLAB\)](#).

## Examples

### Compact a Generalized Linear Regression Model

Reduce the size of a full, fitted generalized linear regression model by discarding the sample data and some information related to the fitting process.

Load the data into the workspace. The simulated sample data contains 15,000 observations and 45 predictor variables.

```
load(fullfile(matlabroot, 'examples', 'stats', 'largedata4reg.mat'))
```

Fit a generalized linear regression model to the data using the first 15 predictor variables.

```
mdl = fitglm(X(:,1:15),Y)

mdl =
Generalized linear regression model:
    y ~ [Linear formula with 16 terms in 15 predictors]
```

Distribution = Normal

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	3.2903	0.00010447	31497	0
x1	-0.0006461	4.9991e-08	-12924	0
x2	-0.00024739	8.6874e-08	-2847.7	0
x3	-9.5161e-05	1.1138e-07	-854.38	0
x4	0.00013143	1.551e-07	847.35	0
x5	7.163e-05	1.9793e-07	361.9	0
x6	4.5064e-06	2.2247e-07	20.257	4.9539e-90
x7	-2.6258e-05	2.5462e-07	-103.13	0
x8	6.284e-05	2.5633e-07	245.15	0
x9	-0.00014288	2.817e-07	-507.19	0
x10	-2.2642e-05	3.0963e-07	-73.127	0
x11	-6.0227e-05	3.1639e-07	-190.36	0
x12	1.1665e-05	3.3921e-07	34.388	1.6995e-249
x13	3.8595e-05	3.5601e-07	108.41	0
x14	0.00010021	4.0312e-07	248.57	0
x15	-6.5674e-06	4.1692e-07	-15.752	1.844e-55

15000 observations, 14984 error degrees of freedom

Estimated Dispersion: 0.000164

F-statistic vs. constant model: 1.18e+07, p-value = 0

**Compact the model. The compact model discards the original sample data and some information related to the fitting process, so it uses less memory than the full model.**

```
compactMdl = compact(mdl)
```

```
compactMdl =
```

```
Compact generalized linear regression model:
```

```
  y ~ [Linear formula with 16 terms in 15 predictors]
```

```
  Distribution = Normal
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	3.2903	0.00010447	31497	0
x1	-0.0006461	4.9991e-08	-12924	0
x2	-0.00024739	8.6874e-08	-2847.7	0

x3	-9.5161e-05	1.1138e-07	-854.38	0
x4	0.00013143	1.551e-07	847.35	0
x5	7.163e-05	1.9793e-07	361.9	0
x6	4.5064e-06	2.2247e-07	20.257	4.9539e-90
x7	-2.6258e-05	2.5462e-07	-103.13	0
x8	6.284e-05	2.5633e-07	245.15	0
x9	-0.00014288	2.817e-07	-507.19	0
x10	-2.2642e-05	3.0963e-07	-73.127	0
x11	-6.0227e-05	3.1639e-07	-190.36	0
x12	1.1665e-05	3.3921e-07	34.388	1.6995e-249
x13	3.8595e-05	3.5601e-07	108.41	0
x14	0.00010021	4.0312e-07	248.57	0
x15	-6.5674e-06	4.1692e-07	-15.752	1.844e-55

15000 observations, 14984 error degrees of freedom  
Estimated Dispersion: 0.000164  
F-statistic vs. constant model: 1.18e+07, p-value = 0

- “Generalized Linear Model Workflow” on page 12-41

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Only the `predict` and `random` functions support code generation.
- When you fit a model by using `fitglm` or `stepwiseglm`, the following restrictions apply.
  - Code generation does not support categorical predictors. You cannot supply training data in a table that contains at least one categorical predictor, and you cannot use the 'CategoricalVars' name-value pair argument. To dummy-code variables that you want treated as categorical, preprocess the categorical data by using `dummyvar` before fitting the model..



- The `Link`, `Derivative`, and `Inverse` fields of the `'Link'` name-value pair argument cannot be anonymous functions. That is, you cannot generate code using a generalized linear model that was created using anonymous functions for links. Instead, declare functions for link components.

## See Also

`GeneralizedLinearModel` | `compact` | `fitglm` | `plotPartialDependence`

## Topics

“Generalized Linear Model Workflow” on page 12-41

“Generalized Linear Models” on page 12-12

**Introduced in R2016b**

## CompactRegressionEnsemble class

Compact regression ensemble class

### Description

Compact version of a regression ensemble (of class `RegressionEnsemble`). The compact version does not include the data for training the regression ensemble. Therefore, you cannot perform some tasks with a compact regression ensemble, such as cross validation. Use a compact regression ensemble for making predictions (regressions) of new data.

### Construction

`ens = compact(fullEns)` constructs a compact decision ensemble from a full decision ensemble.

### Input Arguments

#### **fullEns**

A regression ensemble created by `fitensemble`.

### Properties

#### **CategoricalPredictors**

Indices of categorical predictors, stored as a vector of positive integers. `CategoricalPredictors` contains index values corresponding to the columns of the predictor data that contain categorical predictors. If none of the predictors are categorical, then this property is empty (`[]`).

#### **CombineWeights**

A character vector describing how the ensemble combines learner predictions.

**ExpandedPredictorNames**

Expanded predictor names, stored as a cell array of character vectors.

If the model uses encoding for categorical variables, then `ExpandedPredictorNames` includes the names that describe the expanded variables. Otherwise, `ExpandedPredictorNames` is the same as `PredictorNames`.

**NumTrained**

Number of trained learners in the ensemble, a positive scalar.

**PredictorNames**

A cell array of names for the predictor variables, in the order in which they appear in  $X$ .

**ResponseName**

A character vector with the name of the response variable  $Y$ .

**ResponseTransform**

Function handle for transforming scores, or character vector representing a built-in transformation function. 'none' means no transformation; equivalently, 'none' means  $@(x)x$ .

Add or change a `ResponseTransform` function using dot notation:

```
ens.ResponseTransform = @function
```

**Trained**

The trained learners, a cell array of compact regression models.

**TrainedWeights**

A numeric vector of weights the ensemble assigns to its learners. The ensemble computes predicted response by aggregating weighted predictions from its learners.

## Methods

loss	Regression error
predict	Predict responses using ensemble of regression models
predictorImportance	Estimates of predictor importance
removeLearners	Remove members of compact regression ensemble

## Copy Semantics

Value. To learn how value classes affect copy operations, see [Copying Objects \(MATLAB\)](#).

## Examples

### Reduce Size of Regression Ensemble

Create a compact regression ensemble for efficiently making predictions on new data.

Load the `carsmall` data set. Consider a model that explains a car's fuel economy (MPG) using its weight (`Weight`) and number of cylinders (`Cylinders`).

```
load carsmall
X = [Weight Cylinders];
Y = MPG;
```

Train a boosted ensemble of 100 regression trees using the `LSBoost`. Specify that `Cylinders` is a categorical variable.

```
Mdl = fitensemble(X,Y,'LSBoost',100,'Tree','PredictorNames',{'W','C'},...
    'CategoricalPredictors',2)
```

```
Mdl =
```

```
classreg.learning.regr.RegressionEnsemble
    PredictorNames: {'W' 'C'}
    ResponseName: 'Y'
    CategoricalPredictors: 2
```

```

ResponseTransform: 'none'
  NumObservations: 94
    NumTrained: 100
      Method: 'LSBoost'
        LearnerNames: {'Tree'}
ReasonForTermination: 'Terminated normally after completing the requested number of
  FitInfo: [100x1 double]
FitInfoDescription: {2x1 cell}
Regularization: []

```

Mdl is a RegressionEnsemble model object that contains the training data, among other things.

Create a compact version of Mdl.

```
CMDl = compact(Mdl)
```

```
CMDl =
```

```

classreg.learning.regr.CompactRegressionEnsemble
  PredictorNames: {'W' 'C'}
  ResponseName: 'Y'
CategoricalPredictors: 2
  ResponseTransform: 'none'
  NumTrained: 100

```

CMDl is a CompactRegressionEnsemble model object. CMDl is almost the same as Mdl. One exception is that it does not store the training data.

Compare the amounts of space consumed by Mdl and CMDl.

```

mdlInfo = whos('Mdl');
cMdlInfo = whos('CMDl');
[mdlInfo.bytes cMdlInfo.bytes]

```

```
ans =
```

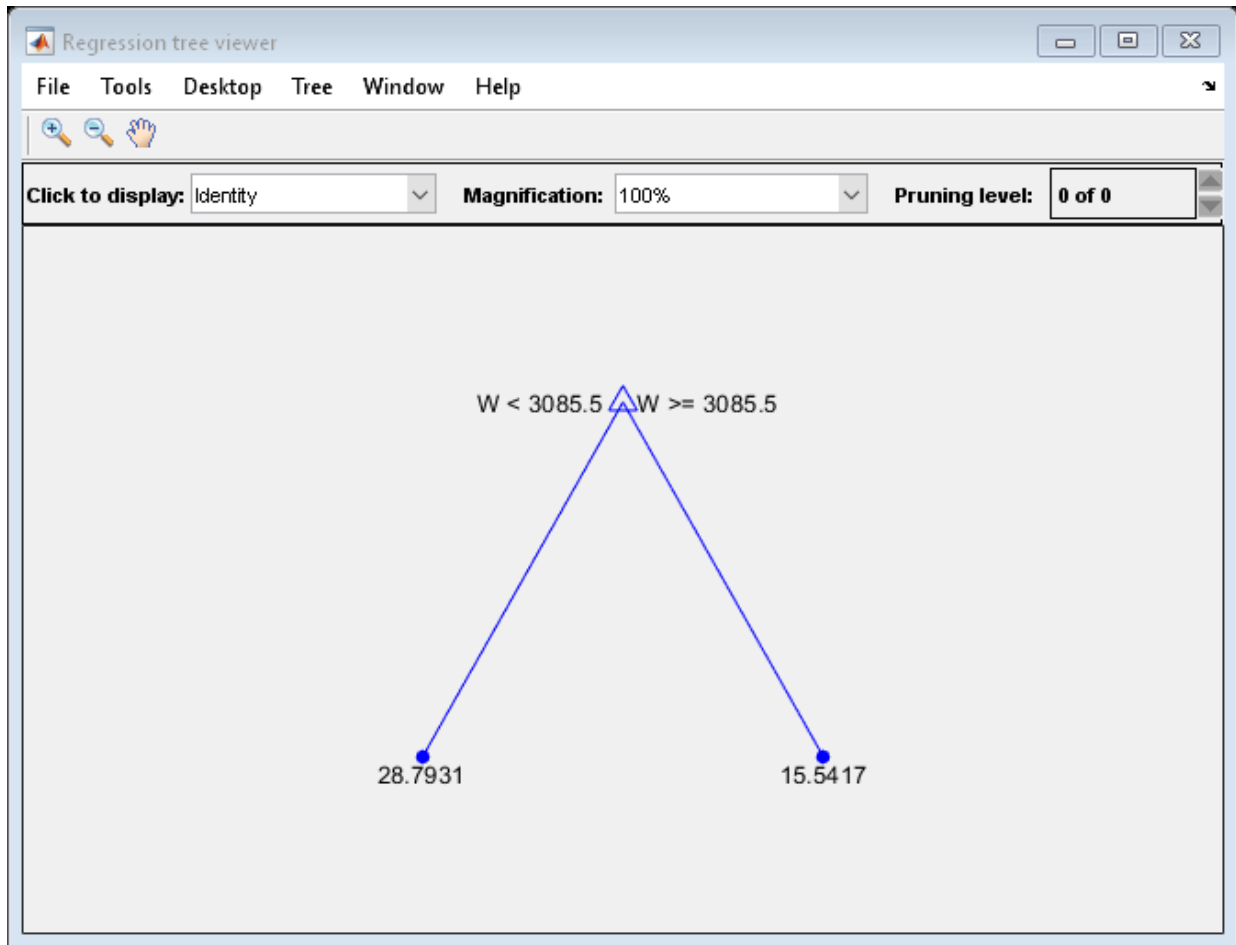
```
240350    219282
```

Mdl consumes more space than CMdl.

CMdl.Trained stores the trained regression trees (CompactRegressionTree model objects) that compose Mdl.

Display a graph of the first tree in the compact ensemble.

```
view(CMdl.Trained{1}, 'Mode', 'graph');
```



By default, `fitensemble` grows stumps for boosted ensembles of trees.

Predict the fuel economy of a typical car using the compact ensemble.

```
typicalX = [mean(X(:,1)) mode(X(:,2))];  
predMeanX = predict(CMdl,typicalX)
```

```
predMeanX =
```

```
    27.3465
```

## Tip

For a compact ensemble of regression trees, the `Trained` property of `ens` stores a cell vector of `ens.NumTrained CompactRegressionTree` model objects. For a textual or graphical display of tree `t` in the cell vector, enter

```
view(ens.Trained{t})
```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Only the `predict` function supports code generation.
- When you train an ensemble by using `fitensemble`, code generation limitations for regression trees also apply to ensembles of regression trees. For more details, see “Code Generation” on page 32-0 of the `CompactRegressionTree` class.

### See Also

`RegressionEnsemble` | `compact` | `fitensemble` | `plotPartialDependence` | `predict` | `templateTree` | `view`

**Introduced in R2011a**



# CompactRegressionGP class

Compact Gaussian process regression model class

## Description

`CompactRegressionGP` is a compact Gaussian process regression (GPR) model. The compact model consumes less memory than a full model, because it does not include the data used for training the GPR model.

Because the compact model does not include the training data, you cannot perform some tasks, such as cross-validation, using the compact model. However, you can use the compact model for making predictions or calculate regression loss for new data (use `predict` and `loss`).

## Construction

`compactMdl = compact(gprMdl)` returns a compact GPR model, `compactMdl`, from a full, trained GPR model, `gprMdl`. For more information, see `compact`.

## Input Arguments

**`gprMdl`** — Full, trained Gaussian process regression model

RegressionGP model

Full, trained Gaussian process regression model, specified as a RegressionGP model, returned by `fitrgp`.

## Properties

### Fitting

**`FitMethod`** — Method used to estimate the parameters

'none' | 'exact' | 'sd' | 'sr' | 'fic'

Method used to estimate the basis function coefficients,  $\beta$ ; noise standard deviation,  $\sigma$ ; and kernel parameters,  $\theta$ , of the GPR model, stored as a character vector. It can be one of the following.

Fit Method	Description
'none'	No estimation. <code>fitrgp</code> uses the initial parameter values as the parameter values.
'exact'	Exact Gaussian process regression.
'sd'	Subset of data points approximation.
'sr'	Subset of regressors approximation.
'fic'	Fully independent conditional approximation.

**BasisFunction — Explicit basis function**

'none' | 'constant' | 'linear' | 'pureQuadratic' | function handle

Explicit basis function used in the GPR model, stored as a character vector or a function handle. It can be one of the following. If  $n$  is the number of observations, the basis function adds the term  $\mathbf{H}^*\boldsymbol{\beta}$  to the model, where  $\mathbf{H}$  is the basis matrix and  $\boldsymbol{\beta}$  is a  $p$ -by-1 vector of basis coefficients.

Explicit Basis	Basis Matrix
'none'	Empty matrix.
'constant'	$\mathbf{H} = \mathbf{1}$ ( $n$ -by-1 vector of 1s, where $n$ is the number of observations)
'linear'	$\mathbf{H} = [\mathbf{1}, \mathbf{X}]$
'pureQuadratic'	$\mathbf{H} = [\mathbf{1}, \mathbf{X}, \mathbf{X}_2]$ , where $\mathbf{X}_2 = \begin{bmatrix} x_{11}^2 & x_{12}^2 & \cdots & x_{1d}^2 \\ x_{21}^2 & x_{22}^2 & \cdots & x_{2d}^2 \\ \vdots & \vdots & \vdots & \vdots \\ x_{n1}^2 & x_{n2}^2 & \cdots & x_{nd}^2 \end{bmatrix}.$

Explicit Basis	Basis Matrix
Function handle	Function handle, <code>hfcn</code> , that <code>fitrgp</code> calls as: $\mathbf{H} = hfcn(\mathbf{X}),$ where $\mathbf{X}$ is an $n$ -by- $d$ matrix of predictors and $\mathbf{H}$ is an $n$ -by- $p$ matrix of basis functions.

Data Types: `char` | `function_handle`

### **CategoricalPredictors** — Indices of categorical predictors

vector of positive integers

Indices of categorical predictors, stored as a vector of positive integers. `CategoricalPredictors` contains index values corresponding to the columns of the predictor data that contain categorical predictors. If none of the predictors are categorical, then this property is empty (`[]`).

Data Types: `single` | `double`

### **Beta** — Estimated coefficients

vector

Estimated coefficients for the explicit basis functions, stored as a vector. You can define the explicit basis function by using the `BasisFunction` name-value pair argument in `fitrgp`.

Data Types: `double`

### **Sigma** — Estimated noise standard deviation

scalar value

Estimated noise standard deviation of the GPR model, stored as a scalar value.

Data Types: `double`

### **ModelParameters** — Parameters used for training

`GPParams` object

Parameters used for training the GPR model, stored as a `GPParams` object.

**Kernel Function****KernelFunction** — Form of the covariance function

'squaredExponential' | 'matern32' | 'matern52' | 'ardsquaredexponential' | 'ardmatern32' | 'ardmatern52' | function handle

Form of the covariance function used in the GPR model, stored as a character vector containing the name of the built-in covariance function or a function handle. It can be one of the following.

Function	Description
'squaredexponential'	Squared exponential kernel.
'matern32'	Matern kernel with parameter 3/2.
'matern52'	Matern kernel with parameter 5/2.
'ardsquaredexponential'	Squared exponential kernel with a separate length scale per predictor.
'ardmatern32'	Matern kernel with parameter 3/2 and a separate length scale per predictor.
'ardmatern52'	Matern kernel with parameter 5/2 and a separate length scale per predictor.
Function handle	A function handle that <code>fitrgp</code> can call like this: $K_{mn} = \text{kfcn}(X_m, X_n, \theta)$ where $X_m$ is an $m$ -by- $d$ matrix, $X_n$ is an $n$ -by- $d$ matrix and $K_{mn}$ is an $m$ -by- $n$ matrix of kernel products such that $K_{mn}(i,j)$ is the kernel product between $X_m(i,:)$ and $X_n(j,:)$ . $\theta$ is the $r$ -by-1 unconstrained parameter vector for <code>kfcn</code> .

Data Types: char | function\_handle

**KernelInformation** — Information about the parameters of the kernel function structure

Information about the parameters of the kernel function used in the GPR model, stored as a structure with the following fields.

Field Name	Description
Name	Name of the kernel function
KernelParameters	Vector of the estimated kernel parameters
KernelParameterNames	Names associated with the elements of KernelParameters.

Data Types: struct

### Prediction

#### **PredictMethod** — Method used to make predictions

'exact' | 'bcd' | 'sd' | 'sr' | 'fic'

Method that `predict` uses to make predictions from the GPR model, stored as a character vector. It can be one of the following.

PredictMethod	Description
'exact'	Exact Gaussian process regression
'bcd'	Block Coordinate Descent
'sd'	Subset of Data points approximation
'sr'	Subset of Regressors approximation
'fic'	Fully Independent Conditional approximation

#### **Alpha** — Weights

numeric vector

Weights used to make predictions from the trained GPR model, stored as a numeric vector. `predict` computes the predictions for a new predictor matrix  $X_{new}$  by using the product

$$\mathbf{K}(X_{new}, \mathbf{A}) * \alpha.$$

$\mathbf{K}(X_{new}, \mathbf{A})$  is the matrix of kernel products between  $X_{new}$  and active set vector  $\mathbf{A}$  and  $\alpha$  is a vector of weights.

Data Types: double

#### **ResponseTransform** — Transformation applied to predicted response

character vector

Transformation applied to the predicted response, stored as a character vector describing how the response values the model predicts are transformed. In `RegressionGP`, `ResponseTransform` is 'none' and `RegressionGP` does not use `ResponseTransform` when making predictions.

Data Types: `char` | `function_handle`

### Active Set Selection

#### **ActiveSetVectors** — Subset of training data

matrix

Subset of training data used to make predictions from the GPR model, stored as a matrix.

`predict` computes the predictions for a new predictor matrix  $X_{new}$  by using the product  $\mathbf{K}(X_{new}, \mathbf{A}) * \alpha$ .

$\mathbf{K}(X_{new}, \mathbf{A})$  is the matrix of kernel products between  $X_{new}$  and active set vector  $\mathbf{A}$  and  $\alpha$  is a vector of weights.

`ActiveSetVectors` is equal to the training data  $X$  for exact GPR fitting and a subset of the training data  $X$  for sparse GPR methods. When there are categorical predictors in the model, `ActiveSetVectors` contains dummy variables for the corresponding predictors.

Data Types: `double`

#### **ActiveSetMethod** — Method used to select the active set

'sgma' | 'entropy' | 'likelihood' | 'random'

Method used to select the active set for sparse methods ('sd', 'sr', or 'fic'), stored as a character vector. It can be one of the following.

ActiveSetMethod	Description
'sgma'	Sparse greedy matrix approximation
'entropy'	Differential entropy-based selection
'likelihood'	Subset of regressors log likelihood-based selection
'random'	Random selection

The selected active set is used in parameter estimation or prediction, depending on the choice of `FitMethod` and `PredictMethod` in `fitrgp`.

**ActiveSetSize** — Size of the active set

integer value

Size of the active set for sparse methods ('sd', 'sr', or 'fic'), stored as an integer value.

Data Types: double

## Methods

`loss`      Regression error for Gaussian process regression model  
`predict`    Predict response of Gaussian process regression model

## Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects (MATLAB).

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Only the `predict` function supports code generation.
- When you train a Gaussian process regression model by using `fitrgp`, the following restrictions apply.
  - Code generation does not support categorical predictors. You cannot supply training data in a table that contains at least one categorical predictor, and you cannot use the 'CategoricalPredictors' name-value pair argument. To dummy-code variables that you want treated as categorical, preprocess the categorical data by using `dummyvar` before fitting the model.

- The value of the 'ResponseTransform' name-value pair argument cannot be an anonymous function.

## See Also

`RegressionGP` | `compact` | `fitrgp` | `plotPartialDependence`

## Topics

Class Attributes (MATLAB)

Property Attributes (MATLAB)

**Introduced in R2015b**



# CompactRegressionSVM class

Compact support vector machine regression model

## Description

`CompactRegressionSVM` is a compact support vector machine (SVM) regression model. It consumes less memory than a full, trained support vector machine model (`RegressionSVM` model) because it does not store the data used to train the model.

Because the compact model does not store the training data, you cannot use it to perform certain tasks, such as cross validation. However, you can use a compact SVM regression model to predict responses using new input data.

## Construction

`compactMdl = compact(mdl)` returns a compact SVM regression model `compactMdl` from a full, trained SVM regression model, `mdl`. For more information, see `compact`.

## Input Arguments

**mdl** — Full, trained SVM regression model

`RegressionSVM` model

Full, trained SVM regression model, specified as a `RegressionSVM` model returned by `fitrsvm`.

## Properties

**Alpha** — Dual problem coefficients

vector of numeric values

Dual problem coefficients, specified as a vector of numeric values. `Alpha` contains  $m$  elements, where  $m$  is the number of support vectors in the trained SVM regression

model. The dual problem introduces two Lagrange multipliers for each support vector. The values of `Alpha` are the differences between the two estimated Lagrange multipliers for the support vectors. For more details, see “Understanding Support Vector Machine Regression” on page 25-2.

If you specified to remove duplicates using `RemoveDuplicates`, then, for a particular set of duplicate observations that are support vectors, `Alpha` contains one coefficient corresponding to the entire set. That is, MATLAB attributes a nonzero coefficient to one observation from the set of duplicates and a coefficient of 0 to all other duplicate observations in the set.

Data Types: `single` | `double`

### **Beta — Primal linear problem coefficients**

vector of numeric values | ' [] '

Primal linear problem coefficients, stored as a numeric vector of length  $p$ , where  $p$  is the number of predictors in the SVM regression model.

The values in `Beta` are the linear coefficients for the primal optimization problem.

If the model is obtained using a kernel function other than `'linear'`, this property is empty (`' [] '`).

The `predict` method computes predicted response values for the model as  $YFIT = (X/S) \times Beta + Bias$ , where  $S$  is the value of the kernel scale stored in the `KernelParameters.Scale` property.

Data Types: `double`

### **Bias — Bias term**

scalar value

Bias term in the SVM regression model, stored as a scalar value.

Data Types: `double`

### **CategoricalPredictors — Indices of categorical predictors**

vector of positive integers

Indices of categorical predictors, stored as a vector of positive integers. `CategoricalPredictors` contains index values corresponding to the columns of the

predictor data that contain categorical predictors. If none of the predictors are categorical, then this property is empty (`[]`).

Data Types: `single` | `double`

### **ExpandedPredictorNames** — Expanded predictor names

cell array of character vectors

Expanded predictor names, stored as a cell array of character vectors.

If the model uses encoding for categorical variables, then `ExpandedPredictorNames` includes the names that describe the expanded variables. Otherwise, `ExpandedPredictorNames` is the same as `PredictorNames`.

### **KernelParameters** — Kernel function parameters

structure

Kernel function parameters, stored as a structure with the following fields.

Field	Description
Function	Kernel function name (a character vector).
Scale	Numeric scale factor used to divide predictor values.

You can specify values for `KernelParameters.Function` and `KernelParameters.Scale` by using the `KernelFunction` and `KernelScale` name-value pair arguments in `fitrsvm`, respectively.

Data Types: `struct`

### **Mu** — Predictor means

vector of numeric values | `'[]'`

Predictor means, stored as a vector of numeric values.

If the training data is standardized, then `Mu` is a numeric vector of length  $p$ , where  $p$  is the number of predictors used to train the model. In this case, the `predict` method centers predictor matrix  $X$  by subtracting the corresponding element of `Mu` from each column.

If the training data is not standardized, then `Mu` is empty (`'[]'`).

Data Types: `single` | `double`

**PredictorNames — Predictor names**

cell array of character vectors

Predictor names, stored as a cell array of character vectors containing the name of each predictor in the order in which they appear in  $X$ . `PredictorNames` has a length equal to the number of columns in  $X$ .

Data Types: `cell`**ResponseName — Response variable name**

character vector

Response variable name, stored as a character vector.

Data Types: `char`**ResponseTransform — Response transformation function**

'none' | function handle

Response transformation function, stored as 'none' or a function handle. `ResponseTransform` describes how the software transforms raw response values.

For a MATLAB function, or a function that you define, enter its function handle. For example, you can enter `Mdl.ResponseTransform = @function`, where `function` accepts a numeric vector of the original responses and returns a numeric vector of the same size containing the transformed responses.

Data Types: `char` | `function_handle`**Sigma — Predictor standard deviations**

vector of numeric values | ' [] '

Predictor standard deviations, stored as a vector of numeric values.

If the training data is standardized, then `Sigma` is a numeric vector of length  $p$ , where  $p$  is the number of predictors used to train the model. In this case, the `predict` method scales the predictor matrix  $X$  by dividing each column by the corresponding element of `Sigma`, after centering each element using `Mu`.

If the training data is not standardized, then `Sigma` is empty (' [] ').

Data Types: `single` | `double`

**SupportVectors** — Support vectors

matrix of numeric values

Support vectors, stored as an  $m$ -by- $p$  matrix of numeric values.  $m$  is the number of support vectors (`sum(Mdl.IsSupportVector)`), and  $p$  is the number of predictors in  $X$ .

If you specified to remove duplicates using `RemoveDuplicates`, then for a given set of duplicate observations that are support vectors, `SupportVectors` contains one unique support vector.

Data Types: `single` | `double`

## Methods

<code>discardSupportVectors</code>	Discard support vectors
<code>loss</code>	Regression error for support vector machine regression model
<code>predict</code>	Predict responses using support vector machine regression model

## Copy Semantics

Value. To learn how value classes affect copy operations, see [Copying Objects \(MATLAB\)](#).

## Examples

### Compact an SVM Regression Model

This example shows how to reduce the size of a full, trained SVM regression model by discarding the training data and some information related to the training process.

This example uses the `abalone` data from the UCI Machine Learning Repository. Download the data and save it in your current directory with the name `'abalone.data'`. Read the data into a table.

```
tbl = readtable('abalone.data', 'Filetype', 'text', 'ReadVariableNames', false);
rng default % for reproducibility
```

The sample data contains 4177 observations. All of the predictor variables are continuous except for `sex`, which is a categorical variable with possible values 'M' (for males), 'F' (for females), and 'I' (for infants). The goal is to predict the number of rings on the abalone, and thereby determine its age, using physical measurements.

Train an SVM regression model using a Gaussian kernel function and an automatic kernel scale. Standardize the data.

```
mdl = fitrsvm(tbl, 'Var9', 'KernelFunction', 'gaussian', 'KernelScale', 'auto', 'Standardize')
mdl =
```

```
RegressionSVM
  PredictorNames: {1x8 cell}
  ResponseName: 'Var9'
  CategoricalPredictors: 1
  ResponseTransform: 'none'
    Alpha: [3635x1 double]
    Bias: 10.8144
  KernelParameters: [1x1 struct]
    Mu: [1x10 double]
    Sigma: [1x10 double]
  NumObservations: 4177
  BoxConstraints: [4177x1 double]
  ConvergenceInfo: [1x1 struct]
  IsSupportVector: [4177x1 logical]
  Solver: 'SMO'
```

Properties, Methods

**Compact the model.**

```
compactMdl = compact(mdl)
```

```
compactMdl =
```

```
classreg.learning.regr.CompactRegressionSVM
  PredictorNames: {1x8 cell}
  ResponseName: 'Var9'
  CategoricalPredictors: 1
  ResponseTransform: 'none'
    Alpha: [3635x1 double]
    Bias: 10.8144
  KernelParameters: [1x1 struct]
```

```

        Mu: [1x10 double]
        Sigma: [1x10 double]
SupportVectors: [3635x10 double]

```

Properties, Methods

The compacted model discards the training data and some information related to the training process.

Compare the size of the full model `mdl` and the compact model `compactMdl`.

```
vars = whos('compactMdl','mdl');
[vars(1).bytes,vars(2).bytes]
```

```
ans =
```

```

    323793    775968

```

The compacted model consumes about half the memory of the full model.

## References

- [1] Nash, W.J., T. L. Sellers, S. R. Talbot, A. J. Cawthorn, and W. B. Ford. *The Population Biology of Abalone (Haliotis species) in Tasmania. I. Blacklip Abalone (H. rubra) from the North Coast and Islands of Bass Strait*, Sea Fisheries Division, Technical Report No. 48, 1994.
- [2] Waugh, S. *Extending and benchmarking Cascade-Correlation*, Ph.D. thesis, Computer Science Department, University of Tasmania, 1995.
- [3] Clark, D., Z. Schreter, A. Adams. *A Quantitative Comparison of Dystal and Backpropagation*, submitted to the Australian Conference on Neural Networks, 1996.
- [4] Lichman, M. *UCI Machine Learning Repository*, [<http://archive.ics.uci.edu/ml>]. Irvine, CA: University of California, School of Information and Computer Science.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Only the `predict` function supports code generation.
- When you train an SVM regression model by using `fitrsvm`, the following restrictions apply.
  - Code generation does not support categorical predictors. You cannot supply training data in a table that contains at least one categorical predictor, and you cannot use the `'CategoricalPredictors'` name-value pair argument. To dummy-code variables that you want treated as categorical, preprocess the categorical data by using `dummyvar` before fitting the model.
  - The value of the `'ResponseTransform'` name-value pair argument cannot be an anonymous function.

### See Also

`RegressionSVM` | `compact` | `fitrsvm` | `plotPartialDependence`

Introduced in R2015b



# CompactRegressionTree class

Compact regression tree

## Description

Compact version of a regression tree (of class `RegressionTree`). The compact version does not include the data for training the regression tree. Therefore, you cannot perform some tasks with a compact regression tree, such as cross validation. Use a compact regression tree for making predictions (regressions) of new data.

## Construction

`ctree = compact(tree)` constructs a compact decision tree from a full decision tree.

## Input Arguments

**tree** — Full, trained regression tree

`RegressionTree` object

Full, trained regression tree, specified as a `RegressionTree` object constructed by `fitrtree`.

## Properties

### **CategoricalPredictors**

Indices of categorical predictors, stored as a vector of positive integers. `CategoricalPredictors` contains index values corresponding to the columns of the predictor data that contain categorical predictors. If none of the predictors are categorical, then this property is empty (`[]`).

**CategoricalSplits**

An  $n$ -by-2 cell array, where  $n$  is the number of categorical splits in `tree`. Each row in `CategoricalSplits` gives left and right values for a categorical split. For each branch node with categorical split  $j$  based on a categorical predictor variable  $z$ , the left child is chosen if  $z$  is in `CategoricalSplits(j, 1)` and the right child is chosen if  $z$  is in `CategoricalSplits(j, 2)`. The splits are in the same order as nodes of the tree. Nodes for these splits can be found by running `cuttype` and selecting 'categorical' cuts from top to bottom.

**Children**

An  $n$ -by-2 array containing the numbers of the child nodes for each node in `tree`, where  $n$  is the number of nodes. Leaf nodes have child node 0.

**CutCategories**

An  $n$ -by-2 cell array of the categories used at branches in `tree`, where  $n$  is the number of nodes. For each branch node  $i$  based on a categorical predictor variable  $x$ , the left child is chosen if  $x$  is among the categories listed in `CutCategories{i, 1}`, and the right child is chosen if  $x$  is among those listed in `CutCategories{i, 2}`. Both columns of `CutCategories` are empty for branch nodes based on continuous predictors and for leaf nodes.

`CutPoint` contains the cut points for 'continuous' cuts, and `CutCategories` contains the set of categories.

**CutPoint**

An  $n$ -element vector of the values used as cut points in `tree`, where  $n$  is the number of nodes. For each branch node  $i$  based on a continuous predictor variable  $x$ , the left child is chosen if `CutPoint < v(i)` and the right child is chosen if `x >= CutPoint(i)`. `CutPoint` is NaN for branch nodes based on categorical predictors and for leaf nodes.

**CutType**

An  $n$ -element cell array indicating the type of cut at each node in `tree`, where  $n$  is the number of nodes. For each node  $i$ , `CutType{i}` is:

- 'continuous' — If the cut is defined in the form  $x < v$  for a variable  $x$  and cut point  $v$ .

- 'categorical' — If the cut is defined by whether a variable  $x$  takes a value in a set of categories.
- '' — If  $i$  is a leaf node.

CutPoint contains the cut points for 'continuous' cuts, and CutCategories contains the set of categories.

### **CutPredictor**

An  $n$ -element cell array of the names of the variables used for branching in each node in `tree`, where  $n$  is the number of nodes. These variables are sometimes known as *cut variables*. For leaf nodes, CutPredictor contains an empty character vector.

CutPoint contains the cut points for 'continuous' cuts, and CutCategories contains the set of categories.

### **ExpandedPredictorNames**

Expanded predictor names, stored as a cell array of character vectors.

If the model uses encoding for categorical variables, then ExpandedPredictorNames includes the names that describe the expanded variables. Otherwise, ExpandedPredictorNames is the same as PredictorNames.

### **IsBranchNode**

An  $n$ -element logical vector `ib` that is true for each branch node and false for each leaf node of `tree`.

### **NodeError**

An  $n$ -element vector `e` of the errors of the nodes in `tree`, where  $n$  is the number of nodes. `e(i)` is the misclassification probability for node  $i$ .

### **NodeMean**

An  $n$ -element numeric array with mean values in each node of `tree`, where  $n$  is the number of nodes in the tree. Every element in NodeMean is the average of the true  $Y$  values over all observations in the node.

**NodeProbability**

An  $n$ -element vector `p` of the probabilities of the nodes in `tree`, where  $n$  is the number of nodes. The probability of a node is computed as the proportion of observations from the original data that satisfy the conditions for the node. This proportion is adjusted for any prior probabilities assigned to each class.

**NodeRisk**

An  $n$ -element vector of the risk of the nodes in the tree, where  $n$  is the number of nodes. The risk for each node is the node error weighted by the node probability.

**NodeSize**

An  $n$ -element vector `sizes` of the sizes of the nodes in `tree`, where  $n$  is the number of nodes. The size of a node is defined as the number of observations from the data used to create the tree that satisfy the conditions for the node.

**NumNodes**

The number of nodes  $n$  in `tree`.

**Parent**

An  $n$ -element vector `p` containing the number of the parent node for each node in `tree`, where  $n$  is the number of nodes. The parent of the root node is 0.

**PredictorNames**

A cell array of names for the predictor variables, in the order in which they appear in `X`.

**PruneAlpha**

Numeric vector with one element per pruning level. If the pruning level ranges from 0 to  $M$ , then `PruneAlpha` has  $M + 1$  elements sorted in ascending order. `PruneAlpha(1)` is for pruning level 0 (no pruning), `PruneAlpha(2)` is for pruning level 1, and so on.

**PruneList**

An  $n$ -element numeric vector with the pruning levels in each node of `tree`, where  $n$  is the number of nodes. The pruning levels range from 0 (no pruning) to  $M$ , where  $M$  is the distance between the deepest leaf and the root node.

**ResponseName**

Name of the response variable  $Y$ , a character vector.

**ResponseTransform**

Function handle for transforming the raw response values (mean squared error). The function handle must accept a matrix of response values and return a matrix of the same size. The default 'none' means  $@(x) x$ , or no transformation.

Add or change a ResponseTransform function using dot notation:

```
ctree.ResponseTransform = @function
```

**SurrogateCutCategories**

An  $n$ -element cell array of the categories used for surrogate splits in `tree`, where  $n$  is the number of nodes in `tree`. For each node  $k$ , `SurrogateCutCategories{k}` is a cell array. The length of `SurrogateCutCategories{k}` is equal to the number of surrogate predictors found at this node. Every element of `SurrogateCutCategories{k}` is either an empty character vector for a continuous surrogate predictor, or is a two-element cell array with categories for a categorical surrogate predictor. The first element of this two-element cell array lists categories assigned to the left child by this surrogate split, and the second element of this two-element cell array lists categories assigned to the right child by this surrogate split. The order of the surrogate split variables at each node is matched to the order of variables in `SurrogateCutPredictor`. The optimal-split variable at this node does not appear. For nonbranch (leaf) nodes, `SurrogateCutCategories` contains an empty cell.

**SurrogateCutFlip**

An  $n$ -element cell array of the numeric cut assignments used for surrogate splits in `tree`, where  $n$  is the number of nodes in `tree`. For each node  $k$ , `SurrogateCutFlip{k}` is a numeric vector. The length of `SurrogateCutFlip{k}` is equal to the number of surrogate predictors found at this node. Every element of `SurrogateCutFlip{k}` is either zero for a categorical surrogate predictor, or a numeric cut assignment for a continuous surrogate predictor. The numeric cut assignment can be either  $-1$  or  $+1$ . For every surrogate split with a numeric cut  $C$  based on a continuous predictor variable  $Z$ , the left child is chosen if  $Z < C$  and the cut assignment for this surrogate split is  $+1$ , or if  $Z \geq C$  and the cut assignment for this surrogate split is  $-1$ . Similarly, the right child is chosen if  $Z \geq C$  and the cut assignment for this surrogate split is  $+1$ , or if  $Z < C$  and the cut

assignment for this surrogate split is  $-1$ . The order of the surrogate split variables at each node is matched to the order of variables in `SurrogateCutPredictor`. The optimal-split variable at this node does not appear. For nonbranch (leaf) nodes, `SurrogateCutFlip` contains an empty array.

### **SurrogateCutPoint**

An  $n$ -element cell array of the numeric values used for surrogate splits in `tree`, where  $n$  is the number of nodes in `tree`. For each node  $k$ , `SurrogateCutPoint{k}` is a numeric vector. The length of `SurrogateCutPoint{k}` is equal to the number of surrogate predictors found at this node. Every element of `SurrogateCutPoint{k}` is either NaN for a categorical surrogate predictor, or a numeric cut for a continuous surrogate predictor. For every surrogate split with a numeric cut  $C$  based on a continuous predictor variable  $Z$ , the left child is chosen if  $Z < C$  and `SurrogateCutFlip` for this surrogate split is  $+1$ , or if  $Z \geq C$  and `SurrogateCutFlip` for this surrogate split is  $-1$ . Similarly, the right child is chosen if  $Z \geq C$  and `SurrogateCutFlip` for this surrogate split is  $+1$ , or if  $Z < C$  and `SurrogateCutFlip` for this surrogate split is  $-1$ . The order of the surrogate split variables at each node is matched to the order of variables returned by `SurrogateCutVar`. The optimal-split variable at this node does not appear. For nonbranch (leaf) nodes, `SurrogateCutPoint` contains an empty cell.

### **SurrogateCutType**

An  $n$ -element cell array indicating types of surrogate splits at each node in `tree`, where  $n$  is the number of nodes in `tree`. For each node  $k$ , `SurrogateCutType{k}` is a cell array with the types of the surrogate split variables at this node. The variables are sorted by the predictive measure of association with the optimal predictor in the descending order, and only variables with the positive predictive measure are included. The order of the surrogate split variables at each node is matched to the order of variables in `SurrogateCutPredictor`. The optimal-split variable at this node does not appear. For nonbranch (leaf) nodes, `SurrogateCutType` contains an empty cell. A surrogate split type can be either 'continuous' if the cut is defined in the form  $Z < V$  for a variable  $Z$  and cut point  $V$  or 'categorical' if the cut is defined by whether  $Z$  takes a value in a set of categories.

### **SurrogateCutPredictor**

An  $n$ -element cell array of the names of the variables used for surrogate splits in each node in `tree`, where  $n$  is the number of nodes in `tree`. Every element of `SurrogateCutPredictor` is a cell array with the names of the surrogate split variables

at this node. The variables are sorted by the predictive measure of association with the optimal predictor in the descending order, and only variables with the positive predictive measure are included. The optimal-split variable at this node does not appear. For nonbranch (leaf) nodes, `SurrogateCutPredictor` contains an empty cell.

### **SurrogatePredictorAssociation**

An  $n$ -element cell array of the predictive measures of association for surrogate splits in `tree`, where  $n$  is the number of nodes in `tree`. For each node  $k$ , `SurrogatePredictorAssociation{k}` is a numeric vector. The length of `SurrogatePredictorAssociation{k}` is equal to the number of surrogate predictors found at this node. Every element of `SurrogatePredictorAssociation{k}` gives the predictive measure of association between the optimal split and this surrogate split. The order of the surrogate split variables at each node is the order of variables in `SurrogateCutPredictor`. The optimal-split variable at this node does not appear. For nonbranch (leaf) nodes, `SurrogatePredictorAssociation` contains an empty cell.

## Methods

<code>loss</code>	Regression error
<code>surrogateAssociation</code>	Mean predictive measure of association for surrogate splits in decision tree
<code>predict</code>	Predict responses using regression tree
<code>predictorImportance</code>	Estimates of predictor importance
<code>view</code>	View tree

## Copy Semantics

Value. To learn how value classes affect copy operations, see [Copying Objects \(MATLAB\)](#).

## Examples

### Construct and Compact a Regression Tree

Load the sample data.

```
load carsmall
```

Construct a regression tree for the sample data.

```
tree = fitrtree([Weight, Cylinders],MPG,...  
    'MinParentSize',20,...  
    'PredictorNames',{'W','C'});
```

Make a compact version of the tree.

```
ctree = compact(tree);
```

Compare the size of the compact tree to that of the full tree.

```
t = whos('tree'); % t.bytes = size of tree in bytes  
c = whos('ctree'); % c.bytes = size of ctree in bytes  
[c.bytes t.bytes]
```

```
ans =
```

```
3708    6954
```

The compact tree is smaller than the full tree.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Only the `predict` function supports code generation.
- When you train a regression tree model by using `fitrtree`, the following restrictions apply.



- Code generation does not support categorical predictors. You cannot supply training data in a table that contains at least one categorical predictor, and you cannot use the 'CategoricalPredictors' name-value pair argument. To dummy-code variables that you want treated as categorical, preprocess the categorical data by using `dummyvar` before fitting the model..
- The value of the 'ResponseTransform' name-value pair argument cannot be an anonymous function.
- You cannot use surrogate splits, that is, the value of the 'Surrogate' name-value pair argument must be 'off'.

## See Also

`RegressionTree` | `compact` | `fitrtree` | `plotPartialDependence`

**Introduced in R2011a**

## CompactTreeBagger class

Compact ensemble of decision trees grown by bootstrap aggregation

### Description

`CompactTreeBagger` class is a lightweight class that contains the trees grown using `TreeBagger`. `CompactTreeBagger` does not preserve any information about how `TreeBagger` grew the decision trees. It does not contain the input data used for growing trees, nor does it contain training parameters such as minimal leaf size or number of variables sampled for each decision split at random. You can only use `CompactTreeBagger` for predicting the response of the trained ensemble given new data  $X$ , and other related functions.

`CompactTreeBagger` lets you save the trained ensemble to disk, or use it in any other way, while discarding training data and various parameters of the training configuration irrelevant for predicting response of the fully grown ensemble. This reduces storage and memory requirements, especially for ensembles trained on large data sets.

### Construction

`CompactTreeBagger`                      Create `CompactTreeBagger` object

`CMdl = compact(Mdl)` creates a compact version of `Mdl`, a `TreeBagger` model object. You can predict regressions using `CMdl` exactly as you can using `Mdl`. However, since `CMdl` does not contain training data, you cannot perform some actions, such as make out-of-bag predictions using `oobPredict`.

## Methods

<code>combine</code>	Combine two ensembles
<code>error</code>	Error (misclassification probability or MSE)
<code>margin</code>	Classification margin
<code>mdsprox</code>	Multidimensional scaling of proximity matrix
<code>meanMargin</code>	Mean classification margin
<code>outlierMeasure</code>	Outlier measure for data
<code>predict</code>	Predict responses using ensemble of bagged decision trees
<code>proximity</code>	Proximity matrix for data
<code>setDefaultYfit</code>	Set default value for <code>predict</code>

## Properties

### **ClassNames**

The `ClassNames` property is a cell array containing the class names for the response variable `Y` supplied to `TreeBagger`. This property is empty for regression trees.

### **DefaultYfit**

The `DefaultYfit` property controls what predicted value `CompactTreeBagger` returns when no prediction is possible, for example when the `predict` method needs to predict for an observation which has only false values in the matrix supplied through `'useifort'` argument.

For classification, you can set this property to either `''` or `'MostPopular'`. If you choose `'MostPopular'` (default), the property value becomes the name of the most probable class in the training data.

For regression, you can set this property to any numeric scalar. The default is the mean of the response for the training data.

**DeltaCriterionDecisionSplit**

The `DeltaCriterionDecisionSplit` property is a numeric array of size 1-by-*Nvars* of changes in the split criterion summed over splits on each variable, averaged across the entire ensemble of grown trees.

**Method**

The `Method` property is 'classification' for classification ensembles and 'regression' for regression ensembles.

**NumPredictorSplit**

The `NumPredictorSplit` property is a numeric array of size 1-by-*Nvars*, where every element gives a number of splits on this predictor summed over all trees.

**NumTrees**

The `NumTrees` property is a scalar equal to the number of decision trees in the ensemble.

**PredictorNames**

The `PredictorNames` property is a cell array containing the names of the predictor variables (features). These names are taken from the optional 'names' parameter that supplied to `TreeBagger`. The default names are 'x1', 'x2', etc.

**SurrogateAssociation**

The `SurrogateAssociation` property is a matrix of size *Nvars*-by-*Nvars* with predictive measures of variable association, averaged across the entire ensemble of grown trees. If you grew the ensemble setting 'surrogate' to 'on', this matrix for each tree is filled with predictive measures of association averaged over the surrogate splits. If you grew the ensemble setting 'surrogate' to 'off' (default), `SurrogateAssociation` is diagonal.

**Trees**

The `Trees` property is a cell array of size `NumTrees`-by-1 containing the trees in the ensemble.

## Examples

### Reduce Size of Bag of Trees

Create a compact bag of trees for efficiently making predictions on new data.

Load the ionosphere data set.

```
load ionosphere
```

Train a bag of 100 classification trees using all measurements and the AdaBoostM1 method.

```
Mdl = TreeBagger(100,X,Y,'Method','classification')
```

```
Mdl =
```

```

TreeBagger
Ensemble with 100 bagged decision trees:
      Training X:      [351x34]
      Training Y:      [351x1]
      Method:          classification
      NumPredictors:   34
      NumPredictorsToSample: 6
      MinLeafSize:     1
      InBagFraction:   1
      SampleWithReplacement: 1
      ComputeOOBPrediction: 0
      ComputeOOBPredictorImportance: 0
      Proximity:       []
      ClassNames:      'b'      'g'
```

Mdl is a TreeBagger model object that contains the training data, among other things.

Create a compact version of Mdl.

```
CMdl = compact(Mdl)
```

```
CMdl =
```

```
CompactTreeBagger
Ensemble with 100 bagged decision trees:
      Method:      classification
NumPredictors:    34
ClassNames: 'b' 'g'
```

CMdl is a CompactTreeBagger model object. CMdl is almost the same as Mdl. One exception is that it does not store the training data.

Compare the amounts of space consumed by Mdl and CMdl.

```
mdlInfo = whos('Mdl');
cMdlInfo = whos('CMdl');
[mdlInfo.bytes cMdlInfo.bytes]
```

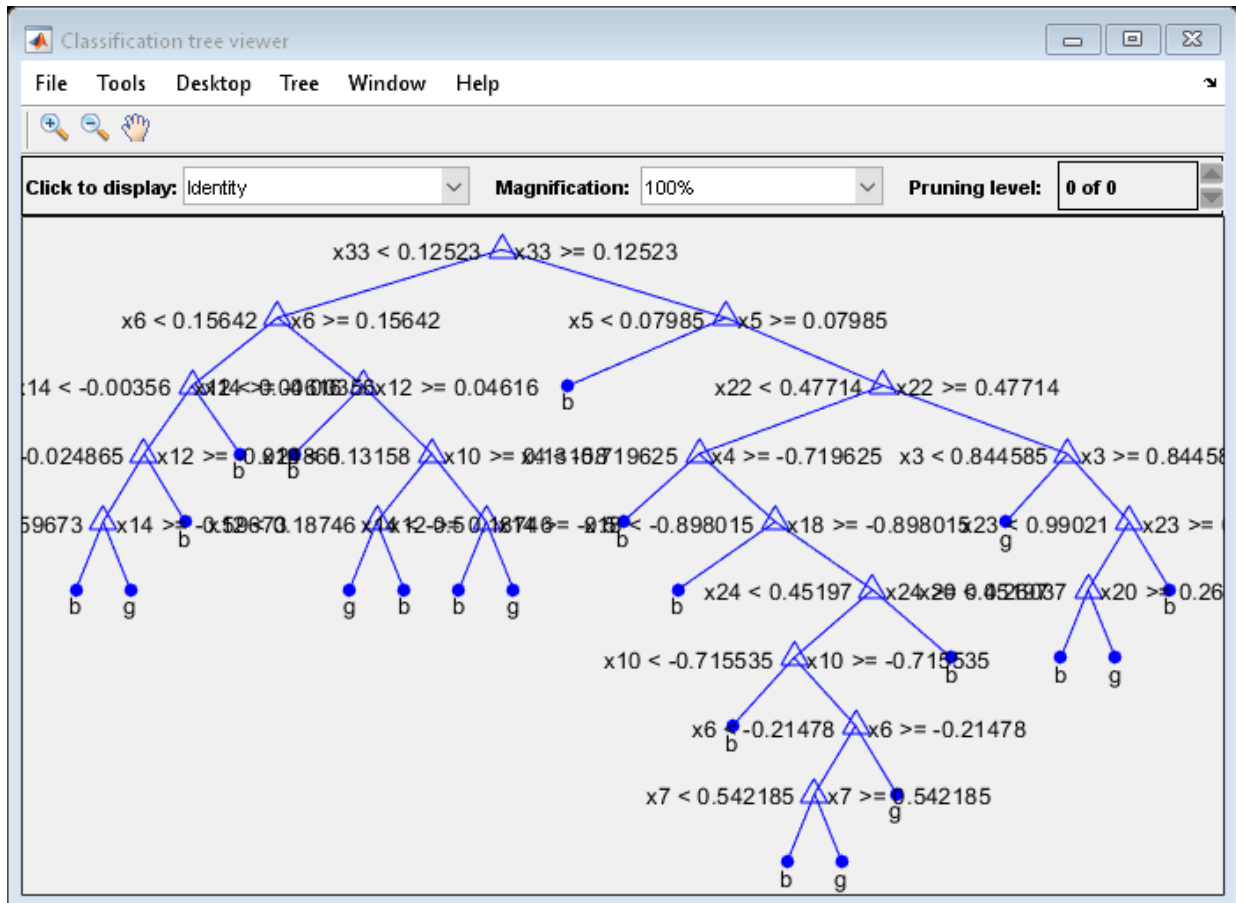
```
ans =
      1031399      890204
```

Mdl consumes more space than CMdl.

CMdl.Trees stores the trained classification trees (CompactClassificationTree model objects) that compose Mdl.

Display a graph of the first tree in the compact model.

```
view(CMdl.Trees{1}, 'Mode', 'graph');
```



By default, TreeBagger grows deep trees.

Predict the label of the mean of  $X$  using the compact ensemble.

```
predMeanX = predict(CMdl,mean(X))
```

```
predMeanX =
```

```
1x1 cell array
```

```
{'g'}
```

- “Bootstrap Aggregation (Bagging) of Regression Trees” on page 18-141
- “Bootstrap Aggregation (Bagging) of Classification Trees” on page 18-154
- “Framework for Ensemble Learning” on page 18-41
- “Decision Trees” on page 19-2
- “Grouping Variables” on page 2-59

## Copy Semantics

Value. To learn how this affects your use of the class, see [Comparing Handle and Value Classes \(MATLAB\)](#) in the MATLAB Object-Oriented Programming documentation.

## Tip

The `Trees` property of `CMdl` stores a cell vector of `CMdl.NumTrees` `CompactClassificationTree` or `CompactRegressionTree` model objects. For a textual or graphical display of tree `t` in the cell vector, enter

```
view(CMdl.Trees{t})
```

## See Also

[ClassificationTree](#) | [RegressionTree](#) | [TreeBagger](#) | [compact](#) | [error](#) | [plotPartialDependence](#) | [predict](#) | [view](#) | [view](#)

## Topics

- “Bootstrap Aggregation (Bagging) of Regression Trees” on page 18-141
- “Bootstrap Aggregation (Bagging) of Classification Trees” on page 18-154
- “Framework for Ensemble Learning” on page 18-41
- “Decision Trees” on page 19-2
- “Grouping Variables” on page 2-59



# CompactTreeBagger

**Class:** CompactTreeBagger

Create CompactTreeBagger object

## Description

When you use the `TreeBagger` constructor to grow trees, it creates a `CompactTreeBagger` object. You can obtain the compact object from the full `TreeBagger` object using the `TreeBagger/compact` method. You do not create an instance of `CompactTreeBagger` directly.

## See Also

`TreeBagger`

## Topics

“Grouping Variables” on page 2-59

“Framework for Ensemble Learning” on page 18-41

**Introduced in R2009a**

## compare

**Class:** GeneralizedLinearMixedModel

Compare generalized linear mixed-effects models

## Syntax

```
results = compare(glme, altglme)
results = compare(glme, altglme, Name, Value)
```

## Description

`results = compare(glme, altglme)` returns the results of a likelihood ratio test on page 32-881 that compares the generalized linear mixed-effects models `glme` and `altglme`. To conduct a valid likelihood ratio test, both models must use the same response vector in the fit, and `glme` must be nested in `altglme`. Always input the smaller model first, and the larger model second.

`compare` tests the following null and alternate hypotheses:

- $H_0$ : Observed response vector is generated by `glme`.
- $H_1$ : Observed response vector is generated by model `altglme`.

`results = compare(glme, altglme, Name, Value)` returns the results of a likelihood ratio test using additional options specified by one or more `Name, Value` pair arguments. For example, you can check if the first input model, `glme`, is nested in the second input model, `altglme`.

## Input Arguments

**glme** — Generalized linear mixed-effects model

GeneralizedLinearMixedModel object

Generalized linear mixed-effects model, specified as a `GeneralizedLinearMixedModel` object. For properties and methods of this object, see `GeneralizedLinearMixedModel`.

You can create a `GeneralizedLinearMixedModel` object by fitting a generalized linear mixed-effects model to your sample data using `fitglme`. To conduct a valid likelihood ratio test on two models that have response distributions other than normal, you must fit both models using the 'ApproximateLaplace' or 'Laplace' fit method. Models with response distributions other than normal that are fitted using 'MPL' or 'REML' cannot be compared using a likelihood ratio test.

### **altglme** — Alternative generalized linear mixed-effects model

`GeneralizedLinearMixedModel` object

Alternative generalized linear mixed-effects model, specified as a `GeneralizedLinearMixedModel` object. `altglme` must fit to the same response vector as `glme`, but with different model specifications. `glme` must be nested in `altglme`, such that you can obtain `glme` from `altglme` by setting some of the model parameters of `altglme` to fixed values such as 0.

You can create a `GeneralizedLinearMixedModel` object by fitting a generalized linear mixed-effects model to your sample data using `fitglme`. To conduct a valid likelihood ratio test on two models that have response distributions other than normal, you must fit both models using the 'ApproximateLaplace' or 'Laplace' fit method. Models with response distributions other than normal that are fitted using 'MPL' or 'REML' cannot be compared using a likelihood ratio test.

## **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### **CheckNesting** — Indicator to check nesting between two models

`true` (default) | `false`

Indicator to check nesting on page 32-882 between two models, specified as the comma-separated pair consisting of 'CheckNesting' and either `true` or `false`. If 'CheckNesting' is `true`, then `compare` checks if the smaller model `glme` is nested in

the larger model `altglme`. If the nesting requirements are not satisfied, then `compare` returns an error. If `'CheckNesting'` is `false`, then `compare` does not perform this check.

Example: `'CheckNesting', true`

## Output Arguments

**results** — Results of likelihood ratio test

table

Results of the likelihood ratio test, returned as a table with two rows. The first row is for `glme`, and the second row is for `altglme`. The columns of `results` contain the following.

Column Name	Description
Model	Name of the model
DF	Degrees of freedom
AIC	Akaike information criterion for the model
BIC	Bayesian information criterion for the model
LogLik	Maximized log likelihood for the model
LRStat	Likelihood ratio test statistic for comparing <code>altglme</code> and <code>glme</code>
deltaDF	DF for <code>altglme</code> minus DF for <code>glme</code>
pValue	<i>p</i> -value for the likelihood ratio test

## Examples

### Compare Mixed-Effects Models

Load the sample data.

```
load mfr
```

This simulated data is from a manufacturing company that operates 50 factories across the world, with each factory running a batch process to create a finished product. The

company wants to decrease the number of defects in each batch, so it developed a new manufacturing process. To test the effectiveness of the new process, the company selected 20 of its factories at random to participate in an experiment: Ten factories implemented the new process, while the other ten continued to run the old process. In each of the 20 factories, the company ran five batches (for a total of 100 batches) and recorded the following data:

- Flag to indicate whether the batch used the new process (`newprocess`)
- Processing time for each batch, in hours (`time`)
- Temperature of the batch, in degrees Celsius (`temp`)
- Categorical variable indicating the supplier of the chemical used in the batch (`supplier`)
- Number of defects in the batch (`defects`)

The data also includes `time_dev` and `temp_dev`, which represent the absolute deviation of time and temperature, respectively, from the process standard of 3 hours at 20 degrees Celsius.

Fit a fixed-effects-only model using `newprocess`, `time_dev`, `temp_dev`, and `supplier` as fixed-effects predictors. Specify the response distribution as Poisson, the link function as log, and the fit method as Laplace. Specify the dummy variable encoding as 'effects', so the dummy variable coefficients sum to 0.

```
FEglme = fitglm(mfr, 'defects ~ 1 + newprocess + time_dev + temp_dev + supplier', 'Distr
```

Fit a second model that uses the same fixed-effects predictors, response distribution, link function, and fit method. This time, include a random-effects intercept grouped by `factory`, to account for quality differences that might exist due to factory-specific variations.

The number of defects can be modeled using a Poisson distribution

$$\text{defects}_{ij} \sim \text{Poisson}(\mu_{ij})$$

This corresponds to the generalized linear mixed-effects model

$$\log(\mu_{ij}) = \beta_0 + \beta_1 \text{newprocess}_{ij} + \beta_2 \text{time.dev}_{ij} + \beta_3 \text{temp.dev}_{ij} + \beta_4 \text{supplier.C}_{ij} + \beta_5 \text{supplier.B}_{ij} + b_i,$$

where

- $\text{defects}_{ij}$  is the number of defects observed in the batch produced by factory  $i$  during batch  $j$ .

- $\mu_{ij}$  is the mean number of defects corresponding to factory  $i$  (where  $i = 1, 2, \dots, 20$ ) during batch  $j$  (where  $j = 1, 2, \dots, 5$ ).
- `newprocessij`, `time.devij`, and `temp.devij` are the measurements for each variable that correspond to factory  $i$  during batch  $j$ . For example, `newprocessij` indicates whether the batch produced by factory  $i$  during batch  $j$  used the new process.
- `supplier.Cij` and `supplier.Bij` are dummy variables that use effects (sum-to-zero) coding to indicate whether company C or B, respectively, supplied the process chemicals for the batch produced by factory  $i$  during batch  $j$ .
- $b_i \sim N(0, \sigma_b^2)$  is a random-effects intercept for each factory  $i$  that accounts for factory-specific variation in quality.

```
glme = fitglme(mfr, 'defects ~ 1 + newprocess + time_dev + temp_dev + supplier + (1|fact
```

Compare the two models using a theoretical likelihood ratio test. Specify 'CheckNesting' as true, so compare returns a warning if the nesting requirements are not satisfied.

```
results = compare(FEglme, glme, 'CheckNesting', true)
```

```
results =
```

THEORETICAL LIKELIHOOD RATIO TEST

Model	DF	AIC	BIC	LogLik	LRStat	deltaDF
FEglme	6	431.02	446.65	-209.51		
glme	7	416.35	434.58	-201.17	16.672	1

```
pValue
```

```
4.4435e-05
```

Since `compare` did not return an error, the nesting requirements are satisfied. The small  $P$ -value indicates that `compare` rejects the null hypothesis that the observed response vector is generated by the model `FEglme`, and instead accepts the alternate model `glme`.

The smaller AIC and BIC values for `glme` also support the conclusion that `glme` provides a better fitting model for the response.

## Definitions

### Likelihood Ratio Test

A *likelihood ratio test* compares the specifications of two nested models by assessing the significance of restrictions to an extended model with unrestricted parameters. Under the null hypothesis  $H_0$ , the likelihood ratio test statistic has an approximate chi-squared reference distribution with degrees of freedom `deltaDF`.

When comparing two models, `compare` computes the  $p$ -value for the likelihood ratio test by comparing the observed likelihood ratio test statistic with this chi-squared reference distribution. A small  $p$ -value leads to a rejection of  $H_0$  in favor of  $H_1$ , and acceptance of the alternate model `altglme`. On the other hand, a large  $p$ -value indicates that we cannot reject  $H_0$ , and reflects insufficient evidence to accept the model `altglme`.

The  $p$ -values obtained using the likelihood ratio test can be conservative when testing for the presence or absence of random-effects terms, and anti-conservative when testing for the presence or absence of fixed-effects terms. Instead, use the `fixedEffects` or `coefTest` methods to test for fixed effects.

To conduct a valid likelihood ratio test on GLME models, both models must be fitted using a Laplace or approximate Laplace fit method. Models fitted using a maximum pseudo likelihood (MPL) or restricted maximum pseudo likelihood (REMPL) method cannot be compared using a likelihood ratio test. When comparing models fitted using MPL, the maximized log likelihood of the pseudodata from the final pseudo likelihood iteration is used in the likelihood ratio test. If you compare models with non-normal distributions fitted using MPL, then `compare` gives a warning that the likelihood ratio test is using maximized log likelihood of pseudodata from the final pseudo likelihood iteration. To use the true maximized log likelihood in the likelihood ratio test, fit both `glme` and `altglme` using approximate Laplace or Laplace prior to model comparison.

## Nesting Requirements

To conduct a valid likelihood ratio test, `glme` must be nested in `altglme`. The `'CheckNesting'`, `true` name-value pair argument checks the following requirements, and returns an error if any are not satisfied:

- You must fit both models (`glme` and `altglme`) using the `'ApproximateLaplace'` or `'Laplace'` fit method. You cannot compare GLME models fitted using `'MPL'` or `'REMPPL'` using a likelihood ratio test.
- You must fit both models using the same response vector, response distribution, and link function.
- The smaller model (`glme`) must be nested within the larger model (`altglme`), such that you can obtain `glme` from `altglme` by setting some of the model parameters of `altglme` to fixed values such as 0.
- The maximized log likelihood of the larger model (`altglme`) must be greater than or equal to the maximized log likelihood of the smaller model (`glme`).
- The weight vectors used to fit `glme` and `altglme` must be identical.
- The random-effects design matrix of the larger model (`altglme`) must contain the random-effects design matrix of the smaller model (`glme`).
- The fixed-effects design matrix of the larger model (`altglme`) must contain the fixed-effects design matrix of the smaller model (`glme`).

## Akaike and Bayesian Information Criteria

The *Akaike information criterion* (AIC) is  $AIC = -2\log L_M + 2(param)$ .

$\log L_M$  depends on the method used to fit the model.

- If you use `'Laplace'` or `'ApproximateLaplace'`, then  $\log L_M$  is the maximized log likelihood.
- If you use `'MPL'`, then  $\log L_M$  is the maximized log likelihood of the pseudo data from the final pseudo likelihood iteration.
- If you use `'REMPPL'`, then  $\log L_M$  is the maximized restricted log likelihood of the pseudo data from the final pseudo likelihood iteration.

*param* is the total number of parameters estimated in the model. For most GLME models, *param* is equal to  $nc + p + 1$ , where *nc* is the total number of parameters in the



random-effects covariance, excluding the residual variance, and  $p$  is the number of fixed-effects coefficients. However, if the dispersion parameter is fixed at 1.0 for binomial or Poisson distributions, then  $param$  is equal to  $(nc + p)$ .

The *Bayesian information criterion* (BIC) is  $BIC = -2 * \log L_M + \ln(n_{eff})(param)$ .

$\log L_M$  depends on the method used to fit the model.

- If you use 'Laplace' or 'ApproximateLaplace', then  $\log L_M$  is the maximized log likelihood.
- If you use 'MPL', then  $\log L_M$  is the maximized log likelihood of the pseudo data from the final pseudo likelihood iteration.
- If you use 'REML', then  $\log L_M$  is the maximized restricted log likelihood of the pseudo data from the final pseudo likelihood iteration.

$n_{eff}$  is the effective number of observations.

- If you use 'MPL', 'Laplace', or 'ApproximateLaplace', then  $n_{eff} = n$ , where  $n$  is the number of observations.
- If you use 'REML', then  $n_{eff} = n - p$ .

$param$  is the total number of parameters estimated in the model. For most GLME models,  $param$  is equal to  $nc + p + 1$ , where  $nc$  is the total number of parameters in the random-effects covariance, excluding the residual variance, and  $p$  is the number of fixed-effects coefficients. However, if the dispersion parameter is fixed at 1.0 for binomial or Poisson distributions, then  $param$  is equal to  $(nc + p)$ .

A lower value of deviance indicates a better fit. As the value of deviance decreases, both AIC and BIC tend to decrease. Both AIC and BIC also include penalty terms based on the number of parameters estimated,  $p$ . So, when the number of parameters increase, the values of AIC and BIC tend to increase as well. When comparing different models, the model with the lowest AIC or BIC value is considered as the best fitting model.

For models fitted using 'MPL' and 'REML', AIC and BIC are based on the log likelihood (or restricted log likelihood) of pseudo data from the final pseudo likelihood iteration. Therefore, a direct comparison of AIC and BIC values between models fitted using 'MPL' and 'REML' is not appropriate.

## See Also

`GeneralizedLinearMixedModel` | `covarianceParameters` | `fixedEffects` | `randomEffects`

# compare

**Class:** LinearMixedModel

Compare linear mixed-effects models

## Syntax

```
results = compare(lme, altlme)
results = compare(____, Name, Value)

[results, siminfo] = compare(lme, altlme, 'NSim', nsim)
[results, siminfo] = compare(____, Name, Value)
```

## Description

`results = compare(lme, altlme)` returns the results of a likelihood ratio test on page 32-897 that compares the linear mixed-effects models `lme` and `altlme`. Both models must use the same response vector in the fit and `lme` must be nested in `altlme` for a valid theoretical likelihood ratio test. Always input the smaller model first, and the larger model second.

`compare` tests the following null and alternate hypotheses:

$H_0$ : Observed response vector is generated by `lme`.

$H_1$ : Observed response vector is generated by model `altlme`.

It is recommended that you fit `lme` and `altlme` using the maximum likelihood (ML) method prior to model comparison. If you use the restricted maximum likelihood (REML) method, then both models must have the same fixed-effects design matrix.

To test for fixed effects, use `compare` with the simulated likelihood ratio test on page 32-898 when `lme` and `altlme` are fit using ML or use the `fixedEffects`, `anova`, or `coefTest` methods.

`results = compare( ____, Name, Value)` also returns the results of a likelihood ratio test that compares linear mixed-effects models `lme` and `altlme` with additional options specified by one or more `Name, Value` pair arguments.

For example, you can check if the first input model is nested in the second input model.

`[results, siminfo] = compare(lme, altlme, 'NSim', nsim)` returns the results of a simulated likelihood ratio test that compares linear mixed-effects models `lme` and `altlme`.

You can fit `lme` and `altlme` using ML or REML. Also, `lme` does not have to be nested in `altlme`. If you use the restricted maximum likelihood (REML) method to fit the models, then both models must have the same fixed-effects design matrix.

`[results, siminfo] = compare( ____, Name, Value)` also returns the results of a simulated likelihood ratio test that compares linear mixed-effects models `lme` and `altlme` with additional options specified by one or more `Name, Value` pair arguments.

For example, you can change the options for performing the simulated likelihood ratio test, or change the confidence level of the confidence interval for the  $p$ -value.

## Input Arguments

### **lme** — Linear mixed-effects model

`LinearMixedModel` object

Linear mixed-effects model, specified as a `LinearMixedModel` object constructed using `fitlme` or `fitlmematrix`.

### **altlme** — Alternative linear mixed-effects model

`LinearMixedModel` object

Alternative linear mixed-effects model fit to the same response vector but with different model specifications, specified as a `LinearMixedModel` object. `lme` must be nested in `altlme`, that is, `lme` should be obtained from `altlme` by setting some parameters to fixed values, such as 0. You can create a linear mixed-effects object using `fitlme` or `fitlmematrix`.

### **nsim** — Number of replications for simulations

positive integer number

Number of replications for simulations in the simulated likelihood ratio test, specified as a positive integer number. You must specify `nsim` to do a simulated likelihood ratio test.

Example: `'NSim', 1000`

Data Types: `double` | `single`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( `' '`). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### Alpha — Confidence level

`0.05` (default) | scalar value in the range 0 to 1

Confidence level, specified as the comma-separated pair consisting of `'Alpha'` and a scalar value in the range 0 to 1. For a value  $\alpha$ , the confidence level is  $100*(1-\alpha)\%$ .

For example, for 99% confidence intervals, you can specify the confidence level as follows.

Example: `'Alpha', 0.01`

Data Types: `single` | `double`

### Options — Options for performing simulated likelihood ratio test

structure

Options for performing the simulated likelihood ratio test, specified as the comma-separated pair consisting of `'Options'`, and a structure created by `statset('LinearMixedModel')`.

`compare` uses the following fields.

- `'UseParallel'`
  - `False` for serial computation. Default.
  - `True` for parallel computation.

'UseSubstreams'

- False for not using a separate substream of the random number generator for each iteration. Default.
- True for using a separate substream of the random number generator for each iteration. You can only use this option with random stream types that support substreams.

'Streams'

- If 'UseSubstreams' is True, then 'Streams' must be a single random number stream, or a scalar cell array containing a single stream.
- If 'UseSubstreams' is False and
  - 'UseParallel' is False, then 'Streams' must be a single random number stream, or a scalar cell array containing a single stream.
  - 'UseParallel' is True, then 'Streams' must be equal to the number of processors used. If a parallel pool is open, then the 'Streams' is the same length as the size of the parallel pool. If 'UseParallel' is True, a parallel pool might open up for you. But since 'Streams' must be equal to the number of processors used, it is best to open a pool explicitly using the `parpool` command, before calling `compare` with the 'UseParallel', 'True' option.

For information on parallel statistical computing at the command line, enter

```
help parallelstats
```

Data Types: `struct`

**CheckNesting** — Indicator to check nesting between two models`false (default) | true`

Indicator to check nesting on page 32-899 between two models, specified as the comma-separated pair consisting of 'CheckNesting' and one of the following.

<code>false</code>	Default. No checks.
<code>true</code>	<code>compare</code> checks if the smaller model <code>lme</code> is nested in the bigger model <code>altlme</code> .

`lme` must be nested in the alternate model `altlme` for a valid theoretical likelihood ratio test on page 32-897. `compare` returns an error message if the nesting requirements are not satisfied.

Although valid for both tests, the nesting requirements are weaker for the simulated likelihood ratio test on page 32-898.

Example: `'CheckNesting', true`

Data Types: `single` | `double`

## Output Arguments

**results** — Results of likelihood ratio test or simulated likelihood ratio test`dataset array`

Results of the likelihood ratio test or simulated likelihood ratio test, returned as a dataset array with two rows. The first row is for `lme`, and the second row is for `altlme`. The columns of `results` depend on whether the test is a likelihood ratio or a simulated likelihood ratio test.

- If you use the likelihood ratio test on page 32-897, then `results` contains the following columns.

Model	Name of the model
DF	Degrees of freedom, that is, the number of free parameters in the model
AIC	Akaike information criterion for the model

BIC	Bayesian information criterion for the model
LogLik	Maximized log likelihood for the model
LRStat	Likelihood ratio test statistic for comparing <code>altlme</code> versus <code>lme</code>
deltaDF	DF for <code>altlme</code> minus DF for <code>lme</code>
pValue	<i>p</i> -value for the likelihood ratio test

- If you use the simulated likelihood ratio test on page 32-898, then `results` contains the following columns.

Model	Name of the model
DF	Degrees of freedom, that is, the number of free parameters in the model
LogLik	Maximized log likelihood for the model
LRStat	Likelihood ratio test statistic for comparing <code>altlme</code> versus <code>lme</code>
pValue	<i>p</i> -value for the likelihood ratio test
Lower	Lower limit of the confidence interval for <code>pValue</code>
Upper	Upper limit of the confidence interval for <code>pValue</code>

### **siminfo** — Simulation output

structure

Simulation output, returned as a structure with the following fields.

<code>nsim</code>	Value set for <code>nsim</code> .
<code>alpha</code>	Value set for 'Alpha'.
<code>pValueSim</code>	Simulation-based <i>p</i> -value.
<code>pValueSimCI</code>	Confidence interval for <code>pValueSim</code> . The first element of the vector is the lower limit and the second element of the vector contains the upper limit.



deltaDF	The number of free parameters in <code>altlme</code> minus the number of free parameters in <code>lme</code> . DF for <code>altlme</code> minus DF for <code>lme</code> .
THO	A vector of simulated likelihood ratio test statistics under the null hypothesis that the model <code>lme</code> generated the observed response vector <code>y</code> .

## Examples

### Test for Random Effects

Load the sample data.

```
load flu
```

The `flu` dataset array has a `Date` variable, and 10 variables containing estimated influenza rates (in 9 different regions, estimated from Google® searches, plus a nationwide estimate from the CDC).

To fit a linear-mixed effects model, your data must be in a properly formatted dataset array. To fit a linear mixed-effects model with the influenza rates as the responses and region as the predictor variable, combine the nine columns corresponding to the regions into an array. The new dataset array, `flu2`, must have the response variable, `FluRate`, the nominal variable, `Region`, that shows which region each estimate is from, and the grouping variable `Date`.

```
flu2 = stack(flu,2:10,'NewDataVarName','FluRate',...
            'IndVarName','Region');
flu2.Date = nominal(flu2.Date);
```

Fit a linear mixed-effects model, with a varying intercept and varying slope for each region, grouped by `Date`.

```
altlme = fitlme(flu2,'FluRate ~ 1 + Region + (1 + Region|Date)');
```

Fit a linear mixed-effects model with fixed effects for the region and a random intercept that varies by `Date`.

```
lme = fitlme(flu2, 'FluRate ~ 1 + Region + (1|Date)');
```

Compare the two models. Also check if `lme2` is nested in `lme`.

```
compare(lme, altlme, 'CheckNesting', true)
```

```
ans =
```

```
THEORETICAL LIKELIHOOD RATIO TEST
```

Model	DF	AIC	BIC	LogLik	LRStat	deltaDF	pValue
lme	11	318.71	364.35	-148.36			
altlme	55	-305.51	-77.346	207.76	712.22	44	0

The small *P*-value of 0 indicates that model `altlme` is significantly better than the simpler model `lme`.

## Test for Fixed and Random Effects

Load the sample data.

```
load(fullfile(matlabroot, 'examples', 'stats', 'fertilizer.mat'));
```

The dataset array includes data from a split-plot experiment, where soil is divided into three blocks based on the soil type: sandy, silty, and loamy. Each block is divided into five plots, where five different types of tomato plants (cherry, heirloom, grape, vine, and plum) are randomly assigned to these plots. The tomato plants in the plots are then divided into subplots, where each subplot is treated by one of four fertilizers. This is simulated data.

Store the data in a dataset array called `ds`, for practical purposes, and define `Tomato`, `Soil`, and `Fertilizer` as categorical variables.

```
ds = fertilizer;  
ds.Tomato = nominal(ds.Tomato);  
ds.Soil = nominal(ds.Soil);  
ds.Fertilizer = nominal(ds.Fertilizer);
```

Fit a linear mixed-effects model, where `Fertilizer` and `Tomato` are the fixed-effects variables, and the mean yield varies by the block (soil type) and the plots within blocks (tomato types within soil types) independently.

```
lmeBig = fitlme(ds, 'Yield ~ Fertilizer * Tomato + (1|Soil) + (1|Soil:Tomato)');
```

Refit the model after removing the interaction term `Tomato:Fertilizer` and the random-effects term `(1 | Soil)`.

```
lmeSmall = fitlme(ds, 'Yield ~ Fertilizer + Tomato + (1|Soil:Tomato)');
```

Compare the two models using the simulated likelihood ratio test with 1000 replications. You must use this test to test for both fixed- and random-effect terms. Note that both models are fit using the default fitting method, ML. That's why, there is no restriction on the fixed-effects design matrices. If you use restricted maximum likelihood (REML) method, both models must have identical fixed-effects design matrices.

```
[table, siminfo] = compare(lmeSmall, lmeBig, 'nsim', 1000)
```

```
table =
```

```
SIMULATED LIKELIHOOD RATIO TEST: NSIM = 1000, ALPHA = 0.05
```

Model	DF	AIC	BIC	LogLik	LRStat	pValue
lmeSmall	10	511.06	532	-245.53		
lmeBig	23	522.57	570.74	-238.29	14.491	0.57343

Lower	Upper
0.54211	0.60431

```
siminfo =
```

```
struct with fields:
```

```

    nsim: 1000
    alpha: 0.0500
    pvalueSim: 0.5734
    pvalueSimCI: [0.5421 0.6043]
    deltaDF: 13

```

```
TH0: [1000x1 double]
```

The high  $P$ -value suggests that the larger model, `lme` is not significantly better than the smaller model, `lme2`. The smaller values of “Akaike and Bayesian Information Criteria” on page 32-899 for `lme2` also support this.

### Models with Correlated and Uncorrelated Random Effects

Load the sample data.

```
load carbig
```

Fit a linear mixed-effects model for miles per gallon (MPG), with fixed effects for acceleration, horsepower, and the cylinders, and potentially correlated random effects for intercept and acceleration grouped by model year.

First, prepare the design matrices.

```
X = [ones(406,1) Acceleration Horsepower];  
Z = [ones(406,1) Acceleration];  
Model_Year = nominal(Model_Year);  
G = Model_Year;
```

Now, fit the model using `fitlmematrix` with the defined design matrices and grouping variables.

```
lme = fitlmematrix(X,MPG,Z,G,'FixedEffectPredictors',...  
{'Intercept','Acceleration','Horsepower'},'RandomEffectPredictors',...  
{{'Intercept','Acceleration'}},'RandomEffectGroups',{'Model_Year'});
```

Refit the model with uncorrelated random effects for intercept and acceleration. First prepare the random effects design and the random effects grouping variables.

```
Z = {ones(406,1),Acceleration};  
G = {Model_Year,Model_Year};  
  
lme2 = fitlmematrix(X,MPG,Z,G,'FixedEffectPredictors',...  
{'Intercept','Acceleration','Horsepower'},'RandomEffectPredictors',...  
{{'Intercept'},{'Acceleration'}},'RandomEffectGroups',...  
{'Model_Year','Model_Year'});
```

Compare `lme` and `lme2` using the simulated likelihood ratio test.

```
compare(lme2,lme,'CheckNesting',true,'NSim',1000)
```

```
ans =
```

```
SIMULATED LIKELIHOOD RATIO TEST: NSIM = 1000, ALPHA = 0.05
```

Model	DF	AIC	BIC	LogLik	LRStat	pValue	Lower
lme2	6	2194.5	2218.3	-1091.3			
lme	7	2193.5	2221.3	-1089.7	3.0323	0.094905	0.077462

```
Upper
```

```
0.11477
```

The high  $P$ -value indicates that lme2 is not a significantly better fit than lme.

### Simulated Likelihood Ratio Test Using Parallel Computing

Load the sample data.

```
load(fullfile(matlabroot,'examples','stats','fertilizer.mat'));
```

The dataset array includes data from a split-plot experiment, where soil is divided into three blocks based on the soil type: sandy, silty, and loamy. Each block is divided into five plots, where five different types of tomato plants (cherry, heirloom, grape, vine, and plum) are randomly assigned to these plots. The tomato plants in the plots are then divided into subplots, where each subplot is treated by one of four fertilizers. This is simulated data.

Store the data in a dataset array called ds, for practical purposes, and define Tomato, Soil, and Fertilizer as categorical variables.

```
ds = fertilizer;
ds.Tomato = nominal(ds.Tomato);
ds.Soil = nominal(ds.Soil);
ds.Fertilizer = nominal(ds.Fertilizer);
```

Fit a linear mixed-effects model, where `Fertilizer` and `Tomato` are the fixed-effects variables, and the mean yield varies by the block (soil type), and the plots within blocks (tomato types within soil types) independently.

```
lme = fitlme(ds, 'Yield ~ Fertilizer * Tomato + (1|Soil) + (1|Soil:Tomato)');
```

Refit the model after removing the interaction term `Tomato:Fertilizer` and the random-effects term `(1|Soil)`.

```
lme2 = fitlme(ds, 'Yield ~ Fertilizer + Tomato + (1|Soil:Tomato)');
```

Create the options structure for `LinearMixedModel`.

```
opt = statset('LinearMixedModel')
```

```
opt =
```

```
    struct with fields:
```

```
        Display: 'off'
    MaxFunEvals: []
        MaxIter: 10000
        TolBnd: []
        TolFun: 1.0000e-06
    TolTypeFun: []
        TolX: 1.0000e-12
    TolTypeX: []
        GradObj: []
        Jacobian: []
        DerivStep: []
    FunValCheck: []
        Robust: []
    RobustWgtFun: []
        WgtFun: []
        Tune: []
    UseParallel: []
    UseSubstreams: []
        Streams: {}
    OutputFcn: []
```

Change the options for parallel testing.

```
opt.UseParallel = true;
```

Start a parallel environment.

```
mypool = parpool();
```

```
Starting parallel pool (parpool) using the 'local' profile ...
connected to 12 workers.
```

Compare `lme2` and `lme` using the simulated likelihood ratio test with 1000 replications and parallel computing.

```
compare(lme2,lme,'nsim',1000,'Options',opt)
```

```
ans =
```

```
SIMULATED LIKELIHOOD RATIO TEST: NSIM = 1000, ALPHA = 0.05
```

Model	DF	AIC	BIC	LogLik	LRStat	pValue	Lower
lme2	10	511.06	532	-245.53			
lme	23	522.57	570.74	-238.29	14.491	0.54246	0.51101

```
Upper
```

```
0.57366
```

The high  $P$ -value suggests that the larger model, `lme` is not significantly better than the smaller model, `lme2`. The smaller values of AIC and BIC for `lme2` also support this.

## Definitions

### Likelihood Ratio Test

Under the null hypothesis  $H_0$ , the observed likelihood ratio test statistic has an approximate chi-squared reference distribution with degrees of freedom `deltaDF`. When comparing two models, `compare` computes the  $p$ -value for the likelihood ratio test by comparing the observed likelihood ratio test statistic with this chi-squared reference distribution.

The  $p$ -values obtained using the likelihood ratio test can be conservative when testing for the presence or absence of random-effects terms and anticonservative when testing for the presence or absence of fixed-effects terms. Hence, use the `fixedEffects`, `anova`, or `coefTest` method or the simulated likelihood ratio test while testing for fixed effects.

## Simulated Likelihood Ratio Test

To perform the simulated likelihood ratio test, `compare` first generates the reference distribution of the likelihood ratio test statistic under the null hypothesis. Then, it assesses the statistical significance of the alternate model by comparing the observed likelihood ratio test statistic to this reference distribution.

`compare` produces the simulated reference distribution of the likelihood ratio test statistic under the null hypothesis as follows:

- Generate random data `ysim` from the fitted model `lme`.
- Fit the model specified in `lme` and alternate model `altlme` to the simulated data `ysim`.
- Calculate the likelihood ratio test statistic using results from step 2 and store the value.
- Repeat step 1 to 3 `nsim` times.

Then, `compare` computes the  $p$ -value for the simulated likelihood ratio test by comparing the observed likelihood ratio test statistic with the simulated reference distribution. The  $p$ -value estimate is the ratio of the number of times the simulated likelihood ratio test statistic is equal to or exceeds the observed value plus one, to the number of replications plus one.

Suppose the observed likelihood ratio statistic is  $T$ , and the simulated reference distribution is stored in vector  $T_{H_0}$ . Then,

$$p\text{-value} = \frac{\left[ \sum_{j=1}^{nsim} I(T_{H_0}(j) \geq T) \right] + 1}{nsim + 1}.$$

To account for the uncertainty in the simulated reference distribution, `compare` computes a  $100*(1 - \alpha)\%$  confidence interval for the true  $p$ -value.

You can use the simulated likelihood ratio test to compare arbitrary linear mixed-effects models. That is, when you are using the simulated likelihood ratio test, `lme` does not



have to be nested within `altlme`, and you can fit `lme` and `altlme` using either maximum likelihood (ML) or restricted maximum likelihood (REML) methods. If you use the restricted maximum likelihood (REML) method to fit the models, then both models must have the same fixed-effects design matrix.

## Nesting Requirements

The `'CheckNesting', 'True'` name-value pair argument checks the following requirements.

For a simulated likelihood ratio test:

- You must use the same method to fit both models (`lme` and `altlme`). `compare` cannot compare a model fit using ML to a model fit using REML.
- You must fit both models to the same response vector.
- If you use REML to fit `lme` and `altlme`, then both models must have the same fixed-effects design matrix.
- The maximized log likelihood or restricted log likelihood of the bigger model (`altlme`) must be greater than or equal to that of the smaller model (`lme`).

For a theoretical test, `'CheckNesting', 'True'` checks all the requirements listed for a simulated likelihood ratio test and the following:

- Weight vectors you use to fit `lme` and `altlme` must be identical.
- If you use ML to fit `lme` and `altlme`, the fixed-effects design matrix of the bigger model (`altlme`) must contain that of the smaller model (`lme`).
- The random-effects design matrix of the bigger model (`altlme`) must contain that of the smaller model (`lme`).

## Akaike and Bayesian Information Criteria

Akaike information criterion (AIC) is  $AIC = -2 \log L_M + 2(nc + p + 1)$ , where  $\log L_M$  is the maximized log likelihood (or maximized restricted log likelihood) of the model, and  $nc + p + 1$  is the number of parameters estimated in the model.  $p$  is the number of fixed-effects coefficients, and  $nc$  is the total number of parameters in the random-effects covariance excluding the residual variance.

Bayesian information criterion (BIC) is  $BIC = -2 \log L_M + \ln(n_{eff}) * (nc + p + 1)$ , where  $\log L_M$  is the maximized log likelihood (or maximized restricted log likelihood) of the

model,  $n_{eff}$  is the effective number of observations, and  $(nc + p + 1)$  is the number of parameters estimated in the model.

- If the fitting method is maximum likelihood (ML), then  $n_{eff} = n$ , where  $n$  is the number of observations.
- If the fitting method is restricted maximum likelihood (REML), then  $n_{eff} = n - p$ .

A lower value of deviance indicates a better fit. As the value of deviance decreases, both AIC and BIC tend to decrease. Both AIC and BIC also include penalty terms based on the number of parameters estimated,  $p$ . So, when the number of parameters increase, the values of AIC and BIC tend to increase as well. When comparing different models, the model with the lowest AIC or BIC value is considered as the best fitting model.

## Deviance

`LinearMixedModel` computes the deviance of model  $M$  as minus two times the loglikelihood of that model. Let  $L_M$  denote the maximum value of the likelihood function for model  $M$ . Then, the deviance of model  $M$  is

$$-2 * \log L_M.$$

A lower value of deviance indicates a better fit. Suppose  $M_1$  and  $M_2$  are two different models, where  $M_1$  is nested in  $M_2$ . Then, the fit of the models can be assessed by comparing the deviances  $Dev_1$  and  $Dev_2$  of these models. The difference of the deviances is

$$Dev = Dev_1 - Dev_2 = 2(\log LM_2 - \log LM_1).$$

Usually, the asymptotic distribution of this difference has a chi-square distribution with degrees of freedom  $v$  equal to the number of parameters that are estimated in one model but fixed (typically at 0) in the other. That is, it is equal to the difference in the number of parameters estimated in  $M_1$  and  $M_2$ . You can get the  $p$ -value for this test using `1 - chi2cdf(Dev, v)`, where  $Dev = Dev_2 - Dev_1$ .

However, in mixed-effects models, when some variance components fall on the boundary of the parameter space, the asymptotic distribution of this difference is more complicated. For example, consider the hypotheses

$$H_0: D = \begin{pmatrix} D_{11} & 0 \\ 0 & 0 \end{pmatrix}, \text{ } D \text{ is a } q\text{-by-}q \text{ symmetric positive semidefinite matrix.}$$

$H_1$ :  $D$  is a  $(q+1)$ -by- $(q+1)$  symmetric positive semidefinite matrix.

That is,  $H_1$  states that the last row and column of  $D$  are different from zero. Here, the bigger model  $M_2$  has  $q + 1$  parameters and the smaller model  $M_1$  has  $q$  parameters. And  $Dev$  has a 50:50 mixture of  $\chi^2_q$  and  $\chi^2_{(q+1)}$  distributions (Stram and Lee, 1994).

## References

[1] Hox, J. *Multilevel Analysis, Techniques and Applications*. Lawrence Erlbaum Associates, Inc., 2002.

[2] Stram D. O. and J. W. Lee. "Variance components testing in the longitudinal mixed-effects model". *Biometrics*, Vol. 50, 4, 1994, pp. 1171–1177.

## See Also

`LinearMixedModel` | `anova` | `covarianceParameters` | `fitlme` | `fitlmematrix` | `fixedEffects` | `randomEffects`

## compareHoldout

**Class:** ClassificationKNN

Compare accuracies of two models using new data

`compareHoldout` statistically assesses the accuracies of two classification models. The function first compares their predicted labels against the true labels, and then it detects whether the difference between the misclassification rates is statistically significant.

You can assess whether the accuracies of the classification models are different, or whether one classification model performs better than another. `compareHoldout` can conduct several McNemar test on page 32-1055 variations, including the asymptotic test, the exact-conditional test, and the mid- $p$ -value test. For cost-sensitive assessment on page 32-1053, available tests include a chi-square test (requires an Optimization Toolbox license) and a likelihood ratio test.

## Syntax

```
h = compareHoldout(C1,C2,T1,T2,ResponseVarName)
```

```
h = compareHoldout(C1,C2,T1,T2,Y)
```

```
h = compareHoldout(C1,C2,X1,X2,Y)
```

```
h = compareHoldout(C1,C2,X1,X2,Y,Name,Value)
```

```
[h,p,e1,e2] = compareHoldout(____)
```

## Description

`h = compareHoldout(C1,C2,T1,T2,ResponseVarName)` returns the test decision from testing the null hypothesis that the trained classification models `C1` and `C2` have equal accuracy for predicting the true class labels `ResponseVarName`. The alternative hypothesis is that the labels have unequal accuracy.

The first classification model `C1` uses predictor data `T1` and `C2` uses `T2`. The software conducts the mid- $p$ -value McNemar test to compare the accuracies.

$h = 1$  indicates to reject the null hypothesis at the 5% significance level.  $h = 0$  indicates to not reject the null hypothesis at 5% level.

Examples of tests you can conduct include:

- Compare the accuracies of a simple classification model and a model that is more complex by passing the same set of predictor data (i.e.,  $T1 = T2$ ).
- Compare the accuracies of two perhaps different models using two perhaps different sets of predictors.
- Perform various types of feature selection. For example, you can compare the accuracy of a model trained using a set of predictors to the accuracy of one trained on a subset or different set of those predictors. You can arbitrarily choose the set of predictors, or use a feature selection technique like PCA or sequential feature selection (see `pca` and `sequentialfs`).

`h = compareHoldout(C1,C2,T1,T2,Y)` returns the test decision from testing the null hypothesis that the trained classification models `C1` and `C2` have equal accuracy for predicting the true class labels `Y`. The alternative hypothesis is that the labels have unequal accuracy.

The first classification model `C1` uses predictor data `T1` and `C2` uses `T2`. The software conducts the mid- $p$ -value McNemar test to compare the accuracies.

`h = compareHoldout(C1,C2,X1,X2,Y)` returns the test decision from testing the null hypothesis that the trained classification models `C1` and `C2` have equal accuracy for predicting the true class labels `Y`. The alternative hypothesis is that the labels have unequal accuracy.

The first classification model `C1` uses predictor data `X1` and `C2` uses `X2`. The software conducts the mid- $p$ -value McNemar test to compare the accuracies.

`h = compareHoldout(C1,C2,X1,X2,Y,Name,Value)` returns the result of the hypothesis test with additional options specified by one or more `Name, Value` pair arguments. For example, you can specify the type of alternative hypothesis, and the type of test, or you can supply a cost matrix.

`[h,p,e1,e2] = compareHoldout(____)` returns the  $p$ -value for the hypothesis test ( $p$ ) and the respective classification losses on page 32-925 of each set of predicted class labels ( $e1$  and  $e2$ ) using any of the input arguments in the previous syntaxes.

## Input Arguments

### **c1 — Trained *k*NN classification model**

`ClassificationKNN` model object

Trained *k*NN classification model, specified as a `ClassificationKNN` model object. That is, `C1` is a trained classification model returned by `fitcknn`.

### **c2 — Trained classification model**

trained classification model object | trained, compact classification model object

Trained classification model, specified as any trained or compact classification model object described in this table.

Trained Model Type	Model Object	Returned By
Classification tree	<code>ClassificationTree</code>	<code>fitctree</code>
Discriminant analysis	<code>ClassificationDiscriminant</code>	<code>fitcdiscr</code>
Ensemble of bagged classification models	<code>ClassificationBaggedEnsemble</code>	<code>fitcensemble</code>
Ensemble of classification models	<code>ClassificationEnsemble</code>	<code>fitcensemble</code>
Multiclass, error-correcting output codes (ECOC)	<code>ClassificationECOC</code>	<code>fitcecoc</code>
<i>k</i> NN	<code>ClassificationKNN</code>	<code>fitcknn</code>
Naive Bayes	<code>ClassificationNaiveBayes</code>	<code>fitcnb</code>
Support vector machine (SVM)	<code>ClassificationSVM</code>	<code>fitcsvm</code>
Compact discriminant analysis	<code>CompactClassificationDiscriminant</code>	<code>compact</code>
Compact ECOC	<code>CompactClassificationECOC</code>	<code>compact</code>
Compact ensemble of classification models	<code>CompactClassificationEnsemble</code>	<code>compact</code>
Compact naive Bayes	<code>CompactClassificationNaiveBayes</code>	<code>compact</code>
Compact SVM	<code>CompactClassificationSVM</code>	<code>compact</code>

Trained Model Type	Model Object	Returned By
Compact classification tree	CompactClassificationTree	compact

### **T1 — Test-set predictor table for first classification model**

table

Test-set predictor table for the first classification model, C1, specified as a table. Each row of T1 corresponds to one observation, and each column corresponds to one predictor variable. Optionally, T1 can contain an additional column for the response variable. T1 must contain all the predictors used to train C1. Multi-column variables and cell arrays other than cell arrays of character vectors are not allowed.

T1 and T2 must have the same number of rows, and must have the same response values. If T1 and T2 contain the response variable used to train C1 and C2, then, you do not need to specify `ResponseVarName` or `Y`.

Data Types: table

### **T2 — Test-set predictor table for second classification model**

table

Test-set predictor data for the second classification model, C2, specified as a table. Each row of T2 corresponds to one observation, and each column corresponds to one predictor variable. Optionally, T2 can contain an additional column for the response variable. T2 must contain all the predictors used to train C2. Multi-column variables and cell arrays other than cell arrays of character vectors are not allowed.

T1 and T2 must have the same number of rows and the same response values. If T1 and T2 contain the response variable used to train C1 and C2, then, you do not need to specify `ResponseVarName` or `Y`.

Data Types: table

### **ResponseVarName — Response variable name**

name of a variable in T1 and T2

Response variable name, specified as the name of a variable in T1 and T2. If T1 and T2 contain the response variable used to train C1 and C2, then you do not need to specify `ResponseVarName`.

If you specify `ResponseVarName`, then you must do so as a character vector. For example, if the response variable is stored as `T1.Response`, then specify it as `'Response'`. Otherwise, the software treats all columns of `T1` and `T2`, including `Response`, as predictors.

The response variable must be a categorical or character array, logical or numeric vector, or cell array of character vectors. If the response variable is a character array, then each element must correspond to one row of the array.

### **x1 — Test-set predictor data for first classification model**

numeric matrix

Test-set predictor data for the first classification model, `C1`, specified as a numeric matrix.

Each row of `X1` corresponds to one observation (also known as an instance or example), and each column corresponds to one variable (also known as a predictor or feature). The variables used to train `C1` must compose `X1`.

The number of rows in `X1` and `X2` must equal the length of `Y`.

Data Types: `double` | `single`

### **x2 — Test-set predictor data for second classification model**

numeric matrix

Test-set predictor data for the second classification model, `C2`, specified as a numeric matrix.

Each row of `X2` corresponds to one observation (also known as an instance or example), and each column corresponds to one variable (also known as a predictor or feature). The variables used to train `C2` must compose `X2`.

The number of rows in `X2` and `X1` must equal the length of `Y`.

Data Types: `double` | `single`

### **y — True class labels**

categorical array | character array | logical vector | vector of numeric values | cell array of character vectors | character vector

True class labels, specified as a categorical or character array, a logical or numeric vector, a cell array of character vectors, or a character vector.



If you specify any array except a character vector, then `ClassificationKNN.compareHoldout` treats all variables or columns in `X1` and `X2` as predictor variables.

If you specify a character vector, then `X1` and `X2` must be tables, their response variables must have the same name and values, and `Y` must be the common name. For example, if `X1.Labels` and `X2.Labels` are the response variables, then `Y` must be `'Labels'` and `X1.Labels` and `X2.Labels` must be equivalent.

If `Y` is a character array, then each element must correspond to one row of the array.

The number of rows in `X1` and `X2` must equal the length of `Y`.

Data Types: `categorical` | `char` | `logical` | `single` | `double` | `cell`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'Alternative', 'greater', 'Test', 'asymptotic', 'Cost', [0 2; 1 0]` specifies to test whether the first set of first predicted class labels is more accurate than the second set, to conduct the asymptotic McNemar test, and to penalize misclassifying observations with the true label `ClassNames{1}` twice as much as for misclassifying observations with the true label `ClassNames{2}`.

### **Alpha** — Hypothesis test significance level

`0.05` (default) | scalar value in the interval (0,1)

Hypothesis test significance level, specified as the comma-separated pair consisting of `'Alpha'` and a scalar value in the interval (0,1).

Example: `'Alpha', 0.1`

Data Types: `single` | `double`

### **Alternative** — Alternative hypothesis to assess

`'unequal'` (default) | `'greater'` | `'less'`

Alternative hypothesis to assess, specified as the comma-separated pair consisting of 'Alternative' and one of these values listed in the table.

Value	Alternative hypothesis
'unequal' (default)	For predicting $Y$ , the set of predictions resulting from $C1$ applied to $X1$ and $C2$ applied to $X2$ have unequal accuracies.
'greater'	For predicting $Y$ , the set of predictions resulting from $C1$ applied to $X1$ is more accurate than $C2$ applied to $X2$ .
'less'	For predicting $Y$ , the set of predictions resulting from $C1$ applied to $X1$ is less accurate than $C2$ applied to $X2$ .

Example: 'Alternative', 'greater'

Data Types: char

### **ClassNames** — Class names

categorical vector | cell array of character vectors | character array | logical vector | numeric vector

Class names, specified as the comma-separated pair consisting of 'ClassNames' and a categorical or character array, logical or numeric vector, or cell array of character vectors. You must set `ClassNames` using the data type of  $Y$ .

If `ClassNames` is a character array, then each element must correspond to one *row* of the array.

Use `ClassNames` to:

- Specify the order of any input argument dimension that corresponds to class order. For example, use `ClassNames` to specify the order of the dimensions of `Cost`.
- Select a subset of classes for testing. For example, suppose that the set of all distinct class names in  $Y$  is {'a', 'b', 'c'}. To train and test models using observations from classes 'a' and 'c' only, specify 'ClassNames', {'a', 'c'}.

The default is the set of all distinct class names in  $Y$ .

Example: 'ClassNames', {'b', 'g'}

Data Types: categorical | char | logical | single | double | cell

### **Cost** — Misclassification cost

square matrix | structure array

Misclassification cost, specified as the comma-separated pair consisting of 'Cost' and a square matrix or structure array. If you specify:

- If you specify the square matrix `Cost`, then `Cost(i,j)` is the cost of classifying a point into class `j` if its true class is `i`. That is, the rows correspond to the true class and the columns correspond to the predicted class. To specify the class order for the corresponding rows and columns of `Cost`, additionally specify the `ClassNames` name-value pair argument.
- If you specify the structure `S`, then `S` must have two fields:
  - `S.ClassNames`, which contains the class names as a variable of the same data type as `Y`. You can use this field to specify the order of the classes.
  - `S.ClassificationCosts`, which contains the cost matrix, with rows and columns ordered as in `S.ClassNames`

If you specify `Cost`, then `ClassificationKNN.compareHoldout` cannot conduct one-sided, exact, or mid- $p$  tests. You must also specify 'Alternative', 'unequal', 'Test', 'asymptotic'. For cost-sensitive testing options, see the `CostTest` name-value pair argument.

It is a best practice to supply the same cost matrix used to train the classification models.

The default is  $\text{Cost}(i,j) = 1$  if  $i \neq j$ , and  $\text{Cost}(i,j) = 0$  if  $i = j$ .

Example: 'Cost',[0 1 2 ; 1 0 2; 2 2 0]

Data Types: double | single | struct

### **CostTest — Cost-sensitive test type**

'likelihood' (default) | 'chisquare'

Cost-sensitive test type, specified as the comma-separated pair consisting of 'CostTest' and 'chisquare' or 'likelihood'. Unless you specify a cost matrix using the `Cost` name-value pair argument, `ClassificationKNN.compareHoldout` ignores `CostTest`.

This table summarizes the available options for cost-sensitive testing.

Value	Asymptotic test type	Requirements
'chisquare'	Chi-square test	Optimization Toolbox license to implement quadprog

Value	Asymptotic test type	Requirements
'likelihood'	Likelihood ratio test	None

For more details, see “Cost-Sensitive Testing” on page 32-6515.

Example: `'CostTest', 'chisquare'`

Data Types: char

### Test — Test to conduct

`'asymptotic' | 'exact' | 'midp'`

Test to conduct, specified as the comma-separated pair consisting of 'Test' and 'asymptotic', 'exact', and 'midp'. This table summarizes the available options for cost-insensitive testing.

Value	Description
'asymptotic'	Asymptotic McNemar test
'exact'	Exact-conditional McNemar test
'midp' (default)	Mid- <i>p</i> -value McNemar test

For more details, see “McNemar Tests” on page 32-6517.

For cost-sensitive testing, Test must be 'asymptotic'. When you specify the Cost name-value pair argument, and choose a cost-sensitive test using the CostTest name-value pair argument, 'asymptotic' is the default.

Example: `'Test', 'asymptotic'`

Data Types: char

---

**Note** NaNs, <undefined> values, and empty character vectors (' ') indicate missing values. `ClassificationKNN.compareHoldout`:

- Removes missing values in Y and the corresponding rows of X1 and X2
  - Predicts classes whether X1 and X2 have missing observations.
-

## Output Arguments

### **h** — Hypothesis test result

1 | 0

Hypothesis test result, returned as a logical value.

$h = 1$  indicates the rejection of the null hypothesis at the Alpha significance level.

$h = 0$  indicates failure to reject the null hypothesis at the Alpha significance level.

### **p** — *p*-value

scalar in the interval [0,1]

*p*-value of the test, returned as a scalar in the interval [0,1]. *p* is the probability that a random test statistic is at least as extreme as the observed test statistic, given that the null hypothesis is true.

`ClassificationKNN.compareHoldout` estimates *p* using the distribution of the test statistic, which varies with the type of test. For details on test statistics derived from the available variants of the McNemar test, see “McNemar Tests” on page 32-6517. For details on test statistics derived from cost-sensitive tests, see “Cost-Sensitive Testing” on page 32-6515.

### **e1** — Classification loss

scalar

Classification loss on page 32-1058, returned as a scalar. *e1* summarizes the accuracy of the first set of class labels predicting the true class labels (*Y*).

`ClassificationKNN.compareHoldout` applies the first test-set predictor data (*X1*) to the first classification model (*C1*) to estimate the first set of class labels. Then, the function compares the estimated labels to *Y* to obtain the classification loss.

For cost-insensitive testing, *e1* is the misclassification rate. That is, *e1* is the proportion of misclassified observations, which is a scalar in the interval [0,1].

For cost-sensitive testing, *e1* is the misclassification cost. That is, *e1* is the weighted average of the misclassification costs, in which the weights are the respective estimated proportions of misclassified observations.

**e2 — Classification loss**

scalar

Classification loss on page 32-1058, returned as a scalar. `e2` summarizes the accuracy of the second set of class labels predicting the true class labels ( $Y$ ).

`ClassificationKNN.compareHoldout` applies the second test-set predictor data ( $X_2$ ) to the second classification model ( $C_2$ ) to estimate the second set of class labels. Then the function compares the estimated labels to  $Y$  to obtain the classification loss.

For cost-insensitive testing, `e2` is the misclassification rate. That is, `e2` is the proportion of misclassified observations, which is a scalar in the interval  $[0,1]$ .

For cost-sensitive testing, `e2` is the misclassification cost. That is, `e2` is the weighted average of the misclassification costs, in which the weights are the respective estimated proportions of misclassified observations.

## Examples

### Compare Accuracies of Two Different Classification Models

Train two classification models using different algorithms. Conduct a statistical test comparing the misclassification rates of the two models on a held-out set.

Load the `ionosphere` data set.

```
load ionosphere;
```

Create a partition that evenly splits the data into training and testing sets.

```
rng(1); % For reproducibility
CVP = cvpartition(Y, 'holdout', 0.5);
idxTrain = training(CVP); % Training-set indices
idxTest = test(CVP); % Test-set indices
```

`CVP` is a cross-validation partition object that specifies the training and test sets.

Train an SVM model and an ensemble of 100 bagged classification trees. For the SVM model, specify to use the radial basis function kernel and a heuristic procedure to determine the kernel scale.

```
C1 = fitcsvm(X(idxTrain,:),Y(idxTrain),'Standardize',true,...
    'KernelFunction','RBF','KernelScale','auto');
C2 = fitensemble(X(idxTrain,:),Y(idxTrain),'Bag',100,'Tree',...
    'Type','classification');
```

C1 is a trained ClassificationSVM model. C2 is a trained ClassificationBaggedEnsemble model.

Test whether the two models have equal predictive accuracies. Use the same test-set predictor data for each model.

```
h = compareHoldout(C1,C2,X(idxTest,:),X(idxTest,:),Y(idxTest))
```

```
h =
    logical
     0
```

$h = 0$  indicates to not reject the null hypothesis that the two models have equal predictive accuracies.

### Assess Whether One Classification Model Classifies Better Than Another

Train two classification models using the same algorithm, but adjust a hyperparameter to make the algorithm more complex. Conduct a statistical test to assess whether the simpler model has better accuracy in held-out data than the more complex model.

Load the `ionosphere` data set.

```
load ionosphere;
```

Create a partition that evenly splits the data into training and testing sets.

```
rng(1); % For reproducibility
CVP = cvpartition(Y,'holdout',0.5);
idxTrain = training(CVP); % Training-set indices
idxTest = test(CVP); % Test-set indices
```

CVP is a cross-validation partition object that specifies the training and test sets.

Train two SVM models: one that uses a linear kernel (the default for binary classification) and one that uses the radial basis function kernel. Use the default kernel scale of 1.

```
C1 = fitcsvm(X(idxTrain,:),Y(idxTrain),'Standardize',true);
C2 = fitcsvm(X(idxTrain,:),Y(idxTrain),'Standardize',true,...
    'KernelFunction','RBF');
```

C1 and C2 are trained ClassificationSVM models.

Test the null hypothesis that the simpler model (C1) is at most as accurate as the more complex model (C2). Because the test-set size is large, conduct the asymptotic McNemar test, and compare the results with the mid- $p$ -value test (the cost-insensitive testing default). Request to return  $p$ -values and misclassification rates.

```
Asymp = zeros(4,1); % Preallocation
MidP = zeros(4,1);

[Asymp(1),Asymp(2),Asymp(3),Asymp(4)] = compareHoldout(C1,C2,...
    X(idxTest,:),X(idxTest,:),Y(idxTest),'Alternative','greater',...
    'Test','asymptotic');
[MidP(1),MidP(2),MidP(3),MidP(4)] = compareHoldout(C1,C2,...
    X(idxTest,:),X(idxTest,:),Y(idxTest),'Alternative','greater');
table(Asymp,MidP,'RowNames',{'h' 'p' 'e1' 'e2'})
```

ans =

4x2 table

	Asymp	MidP
h	1	1
p	7.2801e-09	2.7649e-10
e1	0.13714	0.13714
e2	0.33143	0.33143

The  $p$ -value is close to zero for both tests, which indicates strong evidence to reject the null hypothesis that the simpler model is less accurate than the more complex model. No



matter what test you specify, `compareHoldout` returns the same type of misclassification measure for both models.

### Conduct a Cost-Sensitive Comparison of Two Classification Models

For data sets with imbalanced class representations, or for data sets with imbalanced false-positive and false-negative costs, you can statistically compare the predictive performances of two classification models by including a cost matrix in the analysis.

Load the `arrhythmia` data set. Determine the class representations in the data.

```
load arrhythmia;
Y = categorical(Y);
tabulate(Y);
```

Value	Count	Percent
1	245	54.20%
2	44	9.73%
3	15	3.32%
4	15	3.32%
5	13	2.88%
6	25	5.53%
7	3	0.66%
8	2	0.44%
9	9	1.99%
10	50	11.06%
14	4	0.88%
15	5	1.11%
16	22	4.87%

There are 16 classes, however some are not represented in the data set. Most observations are classified as not having arrhythmia (class 1). To summarize, the data set is highly discrete with imbalanced classes.

Combine all observations with arrhythmia (classes 2 through 15) into one class. Remove those observations with unknown arrhythmia status from the data set.

```
Y = Y(Y ~= '16');
Y(Y ~= '1') = '2';
X = X(Y ~= '16', :);
```

Create a partition that evenly splits the data into training and testing sets.

```
rng(1); % For reproducibility
CVP = cvpartition(Y, 'holdout', 0.5);
idxTrain = training(CVP); % Training-set indices
idxTest = test(CVP); % Test-set indices
```

CVP is a cross-validation partition object that specifies the training and test sets.

Create a cost matrix such that misclassifying an arrhythmic patient into the no arrhythmia class is five times worse than misclassifying a patient without arrhythmia into the arrhythmia class. Classifying correctly incurs no cost. The rows indicate the true class and the columns indicate the predicted class. When conducting a cost-sensitive analysis, it is a good practice to specify the order of the classes.

```
cost = [0 1; 5 0];
ClassNames = categorical([2 1]);
```

Train two boosting ensembles of 50 classification trees, one that uses AdaBoostM1 and the other that uses LogitBoost. Because there are missing values, specify to use surrogate splits. Train the models using the cost matrix.

```
t = templateTree('Surrogate', 'on');
numTrees = 50;
C1 = fitensemble(X(idxTrain, :), Y(idxTrain), 'AdaBoostM1', numTrees, t, ...
    'Cost', cost);
C2 = fitensemble(X(idxTrain, :), Y(idxTrain), 'LogitBoost', numTrees, t, ...
    'Cost', cost);
```

C1 and C2 are trained ClassificationEnsemble models.

Test whether the AdaBoostM1 ensemble (C1) and the LogitBoost ensemble (C2) have equal predictive accuracy. Supply the cost matrix. Conduct the asymptotic, likelihood ratio, cost-sensitive test (the default when you pass in a cost matrix). Request to return *p*-values and misclassification costs.

```
[h, p, e1, e2] = compareHoldout(C1, C2, X(idxTest, :), X(idxTest, :), Y(idxTest), ...
    'Cost', cost)
```

```
h =
    logical
    0
```

```
p =
    0.0743
```

```
e1 =
    1.3581
```

```
e2 =
    1.6186
```

$h = 0$  indicates to not reject the null hypothesis that the two models have equal predictive accuracies.

### Select Features Using Statistical Accuracy Comparison

Reduce classification model complexity by selecting a subset of predictor variables (features) from a larger set. Then, statistically compare the out-of-sample accuracy between the two models.

Load the `ionosphere` data set.

```
load ionosphere;
```

Create a partition that evenly splits the data into training and testing sets.

```
rng(1); % For reproducibility
CVP = cvpartition(Y, 'holdout', 0.5);
idxTrain = training(CVP); % Training-set indices
idxTest = test(CVP); % Test-set indices
```

CVP is a cross-validation partition object that specifies the training and test sets.

Train an ensemble of 100 boosted classification trees using `AdaBoostM1` and the entire set of predictors. Inspect the importance measure for each predictor.

```
nTrees = 100;
C2 = fitensemble(X(idxTrain,:), Y(idxTrain), 'AdaBoostM1', nTrees, 'Tree');
```

```
predImp = predictorImportance(C2);

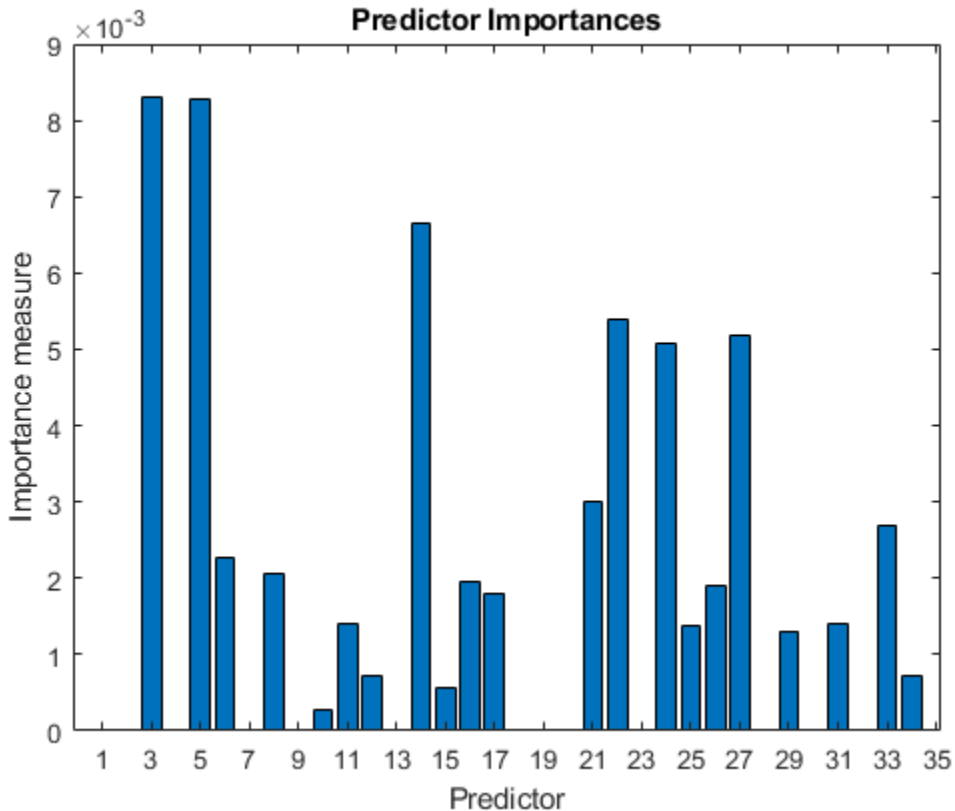
figure;
bar(predImp);
h = gca;
h.XTick = 1:2:h.XLim(2)
title('Predictor Importances');
xlabel('Predictor');
ylabel('Importance measure');

h =

Axes with properties:

    XLim: [-0.2000 35.2000]
    YLim: [0 0.0090]
    XScale: 'linear'
    YScale: 'linear'
    GridLineStyle: '-'
    Position: [0.1300 0.1100 0.7750 0.8150]
    Units: 'normalized'

Use GET to show all properties
```



Identify the top five predictors in terms of their importance.

```
[~,idxSort] = sort(predImp, 'descend');
idx5 = idxSort(1:5);
```

Train another ensemble of 100 boosted classification trees using AdaBoostM1 and the five predictors with the best importance.

```
C1 = fitensemble(X(idxTrain,idx5),Y(idxTrain), 'AdaBoostM1', nTrees, ...
    'Tree');
```

Test whether the two models have equal predictive accuracies. Specify the reduced test-set predictor data for C1 and the full test-set predictor data for C2.

```
[h,p,e1,e2] = compareHoldout(C1,C2,X(idxTest,idx5),X(idxTest,:),Y(idxTest))
```

```
h =  
  logical  
  0  
  
p =  
  0.7744  
  
e1 =  
  0.0914  
  
e2 =  
  0.0857
```

`h = 0` indicates to not reject the null hypothesis that the two models have equal predictive accuracies. This result favors the simpler ensemble, C1.

## Definitions

### Cost-Sensitive Testing

Conduct cost-sensitive testing when the cost of misclassification is imbalanced. When conducting a cost-sensitive analysis, you can account for the cost imbalance in training the classification models, and then in statistically comparing them.

If the cost of misclassification is imbalanced, then the misclassification rate tends to be a poorly performing classification loss on page 32-6520. Use misclassification cost instead to compare classification models.

Misclassification costs are often unbalanced in applications. For example, consider classifying subjects based on a set of predictors into two categories: healthy and sick. Misclassifying a sick subject as healthy poses a danger to the subject's life. However,

misclassifying a healthy subject as sick can cause some inconvenience, but does not pose any danger. In this situation, you assign misclassification costs such that misclassifying a sick subject as healthy is more costly than misclassifying a healthy subject as sick.

The definitions that follow summarize the cost-sensitive tests. In the definitions:

- $n_{ijk}$  and  $\hat{\pi}_{ijk}$  are the number and estimated proportion of test-sample observations with true class  $k$  that the first classification model assigns label  $i$ . The second classification model assigns label  $j$ . The unknown, true value of  $\hat{\pi}_{ijk}$  is  $\pi_{ijk}$ . The test-set sample size is  $\sum_{i,j,k} n_{ijk} = n_{test}$ .  $\sum_{i,j,k} \pi_{ijk} = \sum_{i,j,k} \pi_{ijk} = 1$ .
- $c_{ij}$  is the relative cost of assigning label  $j$  to an observation with true class  $i$ .  $c_{ii} = 0$ ,  $c_{ij} \geq 0$ , and, for at least one  $(i,j)$  pair,  $c_{ij} > 0$ .
- All subscripts take on integer values from 1 through  $K$ , which is the number of classes.
- The expected difference in the misclassification costs of the two classification models is

$$\delta = \sum_{i=1}^K \sum_{j=1}^K \sum_{k=1}^K (c_{ki} - c_{kj}) \pi_{ijk}.$$

- The hypothesis test is
 
$$H_0 : \delta = 0$$

$$H_1 : \delta \neq 0$$

The available cost-sensitive tests are appropriate for two-tailed testing.

Available, asymptotic tests that address imbalanced costs are a chi-square test and a likelihood ratio test.

- Chi-square test — The chi-square test statistic is based on the Pearson and Neyman chi-square test statistics, but with a Laplace correction factor to account for any  $n_{ijk} = 0$ . The test statistic is

$$t_{\chi^2}^* = \sum_{i \neq j} \sum_k \frac{\left( n_{ijk} + 1 - (n_{test} + K^3) \hat{\pi}_{ijk}^{(1)} \right)^2}{n_{ijk} + 1}.$$

If  $1 - F_{\chi^2}(t_{\chi^2}^*; 1) < \alpha$ , then reject  $H_0$ .

•

$\hat{\pi}_{ijk}^{(1)}$  are estimated by minimizing  $t_{\chi^2}^*$  under the constraint that  $\delta = 0$ .

•

$F_{\chi^2}(x; 1)$  is the  $\chi^2$  C.D.F. with one degree of freedom evaluated at  $x$ .

- Likelihood ratio test — The likelihood ratio test is based on  $N_{ijk}$ , which are binomial random variables having sample size  $n_{test}$  and success probability  $\pi_{ijk}$ . They represent the random number of observations with true class  $k$  that the first classification model assigns label  $i$ . The second classification model assigns label  $j$ . Jointly, their distribution is multinomial.

The test statistic is

$$t_{LRT}^* = 2 \log \left[ \frac{P\left(\bigcap_{i,j,k} N_{ijk} = n_{ijk}; n_{test}, \pi_{ijk} = \pi_{ijk}^{(2)}\right)}{P\left(\bigcap_{i,j,k} N_{ijk} = n_{ijk}; n_{test}, \pi_{ijk} = \pi_{ijk}^{(3)}\right)} \right].$$

If  $1 - F_{\chi^2}(t_{LRT}^*; 1) < \alpha$ , then reject  $H_0$ .

•

$\hat{\pi}_{ijk}^{(2)} = \frac{n_{ijk}}{n_{test}}$  is the unrestricted MLE of  $\pi_{ijk}$ .

•

$\hat{\pi}_{ijk}^{(3)} = \frac{n_{ijk}}{n_{test} + \lambda(c_{ki} - c_{kj})}$  is the MLE under the null hypothesis that  $\delta = 0$ .  $\lambda$  is the solution to

$$\sum_{i,j,k} \frac{n_{ijk}(c_{ki} - c_{kj})}{n_{test} + \lambda(c_{ki} - c_{kj})} = 0.$$

•

$F_{\chi^2}(x; 1)$  is the  $\chi^2$  C.D.F. with one degree of freedom evaluated at  $x$ .

## McNemar Tests

McNemar Tests are hypothesis tests that compare two population proportions while addressing the issues resulting from two dependent, matched-pair samples.



One way to compare the predictive accuracies of two classification models is:

- 1 Partition the data into training and test sets.
- 2 Train both classification models using the training set.
- 3 Predict class labels using the test set.
- 4 Summarize the results in a two-by-two table resembling this figure.

		Model 2		
		Correct	Incorrect	
Model 1	Correct	$n_{11}$	$n_{12}$	$n_{1\bullet}$
	Incorrect	$n_{21}$	$n_{22}$	$n_{2\bullet}$
		$n_{\bullet 1}$	$n_{\bullet 2}$	$n_{test}$

$n_{ii}$  are the number of concordant pairs, that is, the number of observations that both models classify the same way (correctly or incorrectly).  $n_{ij}$ ,  $i \neq j$ , are the number of discordant pairs, that is, the number of observations that models classify differently (correctly or incorrectly).

The misclassification rates for Models 1 and 2 are

$$\hat{\pi}_{2\bullet} = n_{2\bullet} / n$$

and  $\hat{\pi}_{\bullet 2} = n_{\bullet 2} / n$ , respectively. A two-sided test for comparing the accuracy of the two models is

$$H_0 : \pi_{\bullet 2} = \pi_{2\bullet}$$

$$H_1 : \pi_{\bullet 2} \neq \pi_{2\bullet}$$

The null hypothesis suggests that the population exhibits marginal homogeneity, which reduces the null hypothesis to  $H_0 : \pi_{12} = \pi_{21}$ . Also, under the null hypothesis,  $N_{12} \sim \text{Binomial}(n_{12} + n_{21}, 0.5)$  [1].

These facts are the basis for these, available McNemar test variants: the asymptotic, exact-conditional and mid-p-value McNemar tests. The definitions that follow summarize the available variants.

- Asymptotic — The asymptotic McNemar test statistics and rejection regions (for significance-level  $\alpha$ ) are:

- For one-sided tests, the test statistic

$$t_{a1}^* = \frac{n_{12} - n_{21}}{\sqrt{n_{12} + n_{21}}}.$$

If  $1 - \Phi\left(|t_1^*|\right) < \alpha$ , where  $\Phi$  is the standard Gaussian C.D.F., then reject  $H_0$ .

- For two-sided tests, the test statistic

$$t_{a2}^* = \frac{(n_{12} - n_{21})^2}{n_{12} + n_{21}}.$$

If  $1 - F_{\chi^2}\left(t_2^*; m\right) < \alpha$ , where  $F_{\chi^2}(x; m)$  is the  $\chi_m^2$  C.D.F. evaluated at  $x$ , then reject  $H_0$ .

This variant requires large-sample theory, specifically, the Gaussian approximation to the binomial distribution. Therefore:

- The total number of discordant pairs,  $n_d = n_{12} + n_{21}$  must be greater than 10 ([1], Ch. 10.1.4).
- In general, asymptotic tests do not guarantee nominal coverage. The observed probability of falsely rejecting the null hypothesis can exceed  $\alpha$ . Simulation studies in [18] suggest this, but the asymptotic McNemar test performs well in terms of statistical power.
- Exact Conditional — The exact-conditional McNemar test statistics and rejection regions (for significance-level  $\alpha$ ) are ([36], [38]):

- For one-sided tests, the test statistic

$$t_1^* = n_{12}$$

If  $F_{\text{Bin}}\left(t_1^*; n_d, 0.5\right) < \alpha$ , where  $F_{\text{Bin}}(x; n, p)$  is the binomial C.D.F. with sample size  $n$  and success probability  $p$  evaluated at  $x$ , then reject  $H_0$ .

- For two-sided tests, the test statistic

$$t_2^* = \min(n_{12}, n_{21})$$

If  $F_{\text{Bin}}(t_2^*; n_d, 0.5) < \alpha / 2$ , then reject  $H_0$ .

The exact conditional test always attains nominal coverage. Simulation studies in [18] suggest that the test is conservative, and then show that the test lacks statistical power compared to other variants. For small or highly discrete test samples, consider using the mid- $p$ -value test ([1], Ch. 3.6.3). For details, see `Test` and “McNemar Tests” on page 32-6517.

- Mid- $p$ -value test — The mid- $p$ -value McNemar test statistics and rejection regions (for significance-level  $\alpha$ ) are ([32]):

- For one-sided tests, the test statistic

$$t_1^* = n_{12}$$

If  $F_{\text{Bin}}(t_1^* - 1; n_{12} + n_{21}, 0.5) + 0.5f_{\text{Bin}}(t_1^*; n_{12} + n_{21}, 0.5) < \alpha$ , where  $F_{\text{Bin}}(x; n, p)$  and  $f_{\text{Bin}}(x; n, p)$  are the binomial C.D.F. and P.D.F, respectively, with sample size  $n$  and success probability  $p$  evaluated at  $x$ , then reject  $H_0$ .

- For two-sided tests, the test statistic

$$t_2^* = \min(n_{12}, n_{21})$$

If  $F_{\text{Bin}}(t_2^* - 1; n_{12} + n_{21} - 1, 0.5) + 0.5f_{\text{Bin}}(t_2^*; n_{12} + n_{21}, 0.5) < \alpha / 2$ , then reject  $H_0$ .

The mid- $p$ -value test addresses the over-conservative behavior of the exact conditional test. The simulation studies in [18] demonstrate that this test attains nominal coverage, and has good statistical power.

## Classification Loss

Classification losses indicate the accuracy of a classification model or set of predicted labels. Two classification losses are misclassification rate and cost.

`ClassificationKNN.compareHoldout` returns the classification losses (see `e1` and `e2`) under the alternative hypothesis (i.e., the unrestricted classification losses).  $n_{ijk}$  is the number of test-sample observations with true class  $k$  that the first classification model

assigns label  $i$  and the second classification model assigns label  $j$ , and the corresponding

estimated proportion is  $\hat{\pi}_{ijk} = \frac{n_{ijk}}{n_{test}}$ . The test-set sample size is  $\sum_{i,j,k} n_{ijk} = n_{test}$ . The indices are taken from 1 through  $K$ , the number of classes.

- Misclassification rate, or classification error, is a scalar in the interval  $[0,1]$  representing the proportion of misclassified observations. That is, the misclassification rate for the first classification model is

$$e_1 = \sum_{j=1}^K \sum_{k=1}^K \sum_{i \neq k} \hat{\pi}_{ijk}.$$

For the misclassification rate of the second classification model ( $e_2$ ), switch the indices  $i$  and  $j$  in the formula.

Classification accuracy decreases as the misclassification rate increases to 1.

- Misclassification cost is a nonnegative scalar and is a measure of classification quality relative to the values the specified cost matrix elements. Its interpretation depends on the specified costs of misclassification. Misclassification cost is the weighted average of the costs of misclassification (specified in a cost matrix,  $C$ ) in which the weights are the respective, estimated proportions of misclassified observations. The misclassification cost for the first classification model is

$$e_1 = \sum_{j=1}^K \sum_{k=1}^K \sum_{i \neq k} \hat{\pi}_{ijk} c_{ki},$$

where  $c_{kj}$  is the cost of classifying an observation into class  $j$  if its true class is  $k$ . For the misclassification cost of the second classification model ( $e_2$ ), switch the indices  $i$  and  $j$  in the formula.

In general, for a fixed cost matrix, classification accuracy decreases as misclassification cost increases.

## Tips

- One way to perform cost-insensitive feature selection is:
  - 1 Train the first classification model (C1) using the full predictor set.
  - 2 Train the second classification model (C2) using the reduced predictor set.

- 3 Specify `X1` as the full, test-set predictor data and `X2` as the reduced test-set predictor data.
- 4 Enter `compareHoldout(C1,C2,X1,X2,'Alternative','less')`. If `compareHoldout` returns 1, then there is enough evidence to suggest that the classification model that uses fewer predictors performs better than the model that uses the full predictor set.

Alternatively, you can assess whether there is a significant difference between the accuracies of the two models. To perform this assessment, remove the `'Alternative','greater'` specification in step 4. `compareHoldout` conducts a two-sided test, and `h = 0` indicates that there is not enough evidence to suggest a difference in the accuracy of the two models.

- Cost-sensitive tests perform numerical optimization, which requires additional computational resources. The likelihood ratio test conducts numerical optimization indirectly by finding the root of a Lagrange multiplier in an interval. For some data sets, if the root lies close to the boundaries of the interval, then the method can fail. Therefore, if you have an Optimization Toolbox license, consider conducting the cost-sensitive chi-square test instead. For more details, see `CostTest` and “Cost-Sensitive Testing” on page 32-1053.

## Alternatives

To directly compare the accuracy of two sets of class labels in predicting a set of true class labels, use `testcholdout`.

## References

- [1] Agresti, A. *Categorical Data Analysis*, 2nd Ed. John Wiley & Sons, Inc.: Hoboken, NJ, 2002.
- [2] Fagerlan, M.W., S Lydersen, P. Laake. “The McNemar Test for Binary Matched-Pairs Data: Mid-p and Asymptotic Are Better Than Exact Conditional.” *BMC Medical Research Methodology*. Vol. 13, 2013, pp. 1–8.
- [3] Lancaster, H.O. “Significance Tests in Discrete Distributions.” *JASA*, Vol. 56, Number 294, 1961, pp. 223–234.

- [4] McNemar, Q. “Note on the Sampling Error of the Difference Between Correlated Proportions or Percentages.” *Psychometrika*, Vol. 12, Number 2, 1947, pp. 153–157.
- [5] Mosteller, F. “Some Statistical Problems in Measuring the Subjective Response to Drugs.” *Biometrics*, Vol. 8, Number 3, 1952, pp. 220–226.

## See Also

`fitcknn` | `predict` | `testcholdout` | `testckfold`

## Topics

“Hypothesis Tests”

**Introduced in R2015a**

# compareHoldout

**Class:** CompactClassificationDiscriminant

Compare accuracies of two classification models using new data

`compareHoldout` statistically assesses the accuracies of two classification models. The function first compares their predicted labels against the true labels, and then it detects whether the difference between the misclassification rates is statistically significant.

You can assess whether the accuracies of the classification models are different, or whether one classification model performs better than another. `compareHoldout` can conduct several McNemar test on page 32-1055 variations, including the asymptotic test, the exact-conditional test, and the mid- $p$ -value test. For cost-sensitive assessment on page 32-1053, available tests include a chi-square test (requires an Optimization Toolbox license) and a likelihood ratio test.

## Syntax

```
h = compareHoldout(C1,C2,X1,X2,Y)
h = compareHoldout(C1,C2,X1,X2,Y,Name,Value)
[h,p,e1,e2] = compareHoldout(____)
```

## Description

`h = compareHoldout(C1,C2,X1,X2,Y)` returns the test decision from testing the null hypothesis that the trained classification models `C1` and `C2` have equal accuracy for predicting the true class labels `Y`. The alternative hypothesis is that the labels have unequal accuracy.

The first classification model `C1` uses predictor data `X1` and `C2` uses `X2`. The software conducts the mid- $p$ -value McNemar test to compare the accuracies.

`h = 1` indicates to reject the null hypothesis at the 5% significance level. `h = 0` indicates to not reject the null hypothesis at 5% level.

Examples of tests you can conduct include:

- Compare the accuracies of a simple classification model and a model that is more complex by passing the same set of predictor data (i.e.,  $X1 = X2$ ).
- Compare the accuracies of two perhaps different models using two perhaps different sets of predictors.
- Perform various types of feature selection. For example, you can compare the accuracy of a model trained using a set of predictors and one trained on a subset or different set of those predictors. You can arbitrarily choose the set of predictors, or use a feature selection technique like PCA or sequential feature selection (see `pca` and `sequentialfs`).

`h = compareHoldout(C1, C2, X1, X2, Y, Name, Value)` returns the result of the hypothesis test with additional options specified by one or more `Name, Value` pair arguments. For example, you can specify the type of alternative hypothesis, and the type of test, or you can supply a cost matrix.

`[h, p, e1, e2] = compareHoldout( ___ )` returns the  $p$ -value for the hypothesis test ( $p$ ) and the respective classification losses on page 32-951 of each set of predicted class labels ( $e1$  and  $e2$ ) using any of the input arguments in the previous syntaxes.

## Input Arguments

### **c1 — Trained discriminant analysis classification model**

`ClassificationDiscriminant` model object |  
`CompactClassificationDiscriminant` model object

Trained discriminant analysis classification model, specified as a `ClassificationDiscriminant` or `CompactClassificationDiscriminant` model object. That is, `C1` is a trained classification model returned by `fitcdiscr` or `compact`.

### **c2 — Trained classification model**

trained classification model object | trained, compact classification model object

Trained classification model, specified as any trained or compact classification model object described in this table.

Trained Model Type	Model Object	Returned By
Classification tree	<code>ClassificationTree</code>	<code>fitctree</code>
Discriminant analysis	<code>ClassificationDiscriminant</code>	<code>fitcdiscr</code>



Trained Model Type	Model Object	Returned By
Ensemble of bagged classification models	ClassificationBaggedEnsemble	fitcensemble
Ensemble of classification models	ClassificationEnsemble	fitcensemble
Multiclass, error-correcting output codes (ECOC)	ClassificationECOC	fitcecoc
<i>k</i> NN	ClassificationKNN	fitcknn
Naive Bayes	ClassificationNaiveBayes	fitcnb
Support vector machine (SVM)	ClassificationSVM	fitcsvm
Compact discriminant analysis	CompactClassificationDiscriminant	compact
Compact ECOC	CompactClassificationECOC	compact
Compact ensemble of classification models	CompactClassificationEnsemble	compact
Compact naive Bayes	CompactClassificationNaiveBayes	compact
Compact SVM	CompactClassificationSVM	compact
Compact classification tree	CompactClassificationTree	compact

### **x1 — Test-set predictor data for first classification model**

numeric matrix

Test-set predictor data for the first classification model, C1, specified as a numeric matrix.

Each row of X1 corresponds to one observation (also known as an instance or example), and each column corresponds to one variable (also known as a predictor or feature). The variables used to train C1 must compose X1.

The number of rows in X1 and X2 must equal the length of Y.

Data Types: double | single

**x2 — Test-set predictor data for second classification model**

numeric matrix

Test-set predictor data for the second classification model, `C2`, specified as a numeric matrix.

Each row of `X2` corresponds to one observation (also known as an instance or example), and each column corresponds to one variable (also known as a predictor or feature). The variables used to train `C2` must compose `X2`.

The number of rows in `X2` and `X1` must equal the length of `Y`.

Data Types: `double` | `single`

**y — True class labels**

categorical array | character array | logical vector | vector of numeric values | cell array of character vectors | character vector

True class labels, specified as a categorical or character array, a logical or numeric vector, a cell array of character vectors, or a character vector.

If you specify any array except a character vector, then `CompactClassificationDiscriminant.compareHoldout` treats all variables or columns in `X1` and `X2` as predictor variables.

If you specify a character vector, then `X1` and `X2` must be tables, their response variables must have the same name and values, and `Y` must be the common name. For example, if `X1.Labels` and `X2.Labels` are the response variables, then `Y` must be `'Labels'` and `X1.Labels` and `X2.Labels` must be equivalent.

If `Y` is a character array, then each element must correspond to one row of the array.

The number of rows in `X1` and `X2` must equal the length of `Y`.

Data Types: `categorical` | `char` | `logical` | `single` | `double` | `cell`

**Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: 'Alternative', 'greater', 'Test', 'asymptotic', 'Cost', [0 2;1 0] specifies to test whether the first set of first predicted class labels is more accurate than the second set, to conduct the asymptotic McNemar test, and to penalize misclassifying observations with the true label `ClassNames{1}` twice as much as for misclassifying observations with the true label `ClassNames{2}`.

### **Alpha — Hypothesis test significance level**

0.05 (default) | scalar value in the interval (0,1)

Hypothesis test significance level, specified as the comma-separated pair consisting of 'Alpha' and a scalar value in the interval (0,1).

Example: 'Alpha', 0.1

Data Types: single | double

### **Alternative — Alternative hypothesis to assess**

'unequal' (default) | 'greater' | 'less'

Alternative hypothesis to assess, specified as the comma-separated pair consisting of 'Alternative' and one of these values listed in the table.

Value	Alternative hypothesis
'unequal' (default)	For predicting Y, the set of predictions resulting from C1 applied to X1 and C2 applied to X2 have unequal accuracies.
'greater'	For predicting Y, the set of predictions resulting from C1 applied to X1 is more accurate than C2 applied to X2.
'less'	For predicting Y, the set of predictions resulting from C1 applied to X1 is less accurate than C2 applied to X2.

Example: 'Alternative', 'greater'

Data Types: char

### **ClassNames — Class names**

categorical vector | cell array of character vectors | character array | logical vector | numeric vector

Class names, specified as the comma-separated pair consisting of 'ClassNames' and a categorical or character array, logical or numeric vector, or cell array of character vectors. You must set `ClassNames` using the data type of Y.

If `ClassNames` is a character array, then each element must correspond to one *row* of the array.

Use `ClassNames` to:

- Specify the order of any input argument dimension that corresponds to class order. For example, use `ClassNames` to specify the order of the dimensions of `Cost`.
- Select a subset of classes for testing. For example, suppose that the set of all distinct class names in `Y` is `{'a', 'b', 'c'}`. To train and test models using observations from classes 'a' and 'c' only, specify `'ClassNames', {'a', 'c'}`.

The default is the set of all distinct class names in `Y`.

Example: `'ClassNames', {'b', 'g'}`

Data Types: `categorical | char | logical | single | double | cell`

### **Cost — Misclassification cost**

`square matrix | structure array`

Misclassification cost, specified as the comma-separated pair consisting of `'Cost'` and a square matrix or structure array. If you specify:

- If you specify the square matrix `Cost`, then `Cost(i, j)` is the cost of classifying a point into class `j` if its true class is `i`. That is, the rows correspond to the true class and the columns correspond to the predicted class. To specify the class order for the corresponding rows and columns of `Cost`, additionally specify the `ClassNames` name-value pair argument.
- If you specify the structure `S`, then `S` must have two fields:
  - `S.ClassNames`, which contains the class names as a variable of the same data type as `Y`. You can use this field to specify the order of the classes.
  - `S.ClassificationCosts`, which contains the cost matrix, with rows and columns ordered as in `S.ClassNames`

If you specify `Cost`, then `CompactClassificationDiscriminant.compareHoldout` cannot conduct one-sided, exact, or mid- $p$  tests. You must also specify `'Alternative', 'unequal', 'Test', 'asymptotic'`. For cost-sensitive testing options, see the `CostTest` name-value pair argument.

It is a best practice to supply the same cost matrix used to train the classification models.

The default is  $\text{Cost}(i, j) = 1$  if  $i \neq j$ , and  $\text{Cost}(i, j) = 0$  if  $i = j$ .

Example: 'Cost', [0 1 2 ; 1 0 2; 2 2 0]

Data Types: double | single | struct

### **CostTest** — Cost-sensitive test type

'likelihood' (default) | 'chisquare'

Cost-sensitive test type, specified as the comma-separated pair consisting of 'CostTest' and 'chisquare' or 'likelihood'. Unless you specify a cost matrix using the Cost name-value pair argument,

CompactClassificationDiscriminant.compareHoldout ignores CostTest.

This table summarizes the available options for cost-sensitive testing.

Value	Asymptotic test type	Requirements
'chisquare'	Chi-square test	Optimization Toolbox license to implement quadprog
'likelihood'	Likelihood ratio test	None

For more details, see “Cost-Sensitive Testing” on page 32-6515.

Example: 'CostTest', 'chisquare'

Data Types: char

### **Test** — Test to conduct

'asymptotic' | 'exact' | 'midp'

Test to conduct, specified as the comma-separated pair consisting of 'Test' and 'asymptotic', 'exact', and 'midp'. This table summarizes the available options for cost-insensitive testing.

Value	Description
'asymptotic'	Asymptotic McNemar test
'exact'	Exact-conditional McNemar test
'midp' (default)	Mid- <i>p</i> -value McNemar test

For more details, see “McNemar Tests” on page 32-6517.

For cost-sensitive testing, `Test` must be `'asymptotic'`. When you specify the `Cost` name-value pair argument, and choose a cost-sensitive test using the `CostTest` name-value pair argument, `'asymptotic'` is the default.

Example: `'Test','asymptotic'`

Data Types: `char`

---

**Note** NaNs, `<undefined>` values, and empty character vectors ( `' '` ) indicate missing values. `CompactClassificationDiscriminant.compareHoldout`:

- Removes missing values in `Y` and the corresponding rows of `X1` and `X2`
  - Predicts classes whether `X1` and `X2` have missing observations.
- 

## Output Arguments

### **h** — Hypothesis test result

1 | 0

Hypothesis test result, returned as a logical value.

`h = 1` indicates the rejection of the null hypothesis at the Alpha significance level.

`h = 0` indicates failure to reject the null hypothesis at the Alpha significance level.

### **p** — *p*-value

scalar in the interval [0,1]

*p*-value of the test, returned as a scalar in the interval [0,1]. *p* is the probability that a random test statistic is at least as extreme as the observed test statistic, given that the null hypothesis is true.

`CompactClassificationDiscriminant.compareHoldout` estimates *p* using the distribution of the test statistic, which varies with the type of test. For details on test statistics derived from the available variants of the McNemar test, see “McNemar Tests” on page 32-6517. For details on test statistics derived from cost-sensitive tests, see “Cost-Sensitive Testing” on page 32-6515.

**e1 — Classification loss**

scalar

Classification loss on page 32-1058, returned as a scalar.  $e1$  summarizes the accuracy of the first set of class labels predicting the true class labels ( $Y$ ).

`CompactClassificationDiscriminant.compareHoldout` applies the first test-set predictor data ( $X1$ ) to the first classification model ( $C1$ ) to estimate the first set of class labels. Then, the function compares the estimated labels to  $Y$  to obtain the classification loss.

For cost-insensitive testing,  $e1$  is the misclassification rate. That is,  $e1$  is the proportion of misclassified observations, which is a scalar in the interval  $[0,1]$ .

For cost-sensitive testing,  $e1$  is the misclassification cost. That is,  $e1$  is the weighted average of the misclassification costs, in which the weights are the respective estimated proportions of misclassified observations.

**e2 — Classification loss**

scalar

Classification loss on page 32-1058, returned as a scalar.  $e2$  summarizes the accuracy of the second set of class labels predicting the true class labels ( $Y$ ).

`CompactClassificationDiscriminant.compareHoldout` applies the second test-set predictor data ( $X2$ ) to the second classification model ( $C2$ ) to estimate the second set of class labels. Then the function compares the estimated labels to  $Y$  to obtain the classification loss.

For cost-insensitive testing,  $e2$  is the misclassification rate. That is,  $e2$  is the proportion of misclassified observations, which is a scalar in the interval  $[0,1]$ .

For cost-sensitive testing,  $e2$  is the misclassification cost. That is,  $e2$  is the weighted average of the misclassification costs, in which the weights are the respective estimated proportions of misclassified observations.

## Examples

### Compare Accuracies of Two Different Classification Models

Train two classification models using different algorithms. Conduct a statistical test comparing the misclassification rates of the two models on a held-out set.

Load the `ionosphere` data set.

```
load ionosphere;
```

Create a partition that evenly splits the data into training and testing sets.

```
rng(1); % For reproducibility
CVP = cvpartition(Y, 'holdout', 0.5);
idxTrain = training(CVP); % Training-set indices
idxTest = test(CVP); % Test-set indices
```

CVP is a cross-validation partition object that specifies the training and test sets.

Train an SVM model and an ensemble of 100 bagged classification trees. For the SVM model, specify to use the radial basis function kernel and a heuristic procedure to determine the kernel scale.

```
C1 = fitcsvm(X(idxTrain,:), Y(idxTrain), 'Standardize', true, ...
    'KernelFunction', 'RBF', 'KernelScale', 'auto');
C2 = fitensemble(X(idxTrain,:), Y(idxTrain), 'Bag', 100, 'Tree', ...
    'Type', 'classification');
```

C1 is a trained `ClassificationSVM` model. C2 is a trained `ClassificationBaggedEnsemble` model.

Test whether the two models have equal predictive accuracies. Use the same test-set predictor data for each model.

```
h = compareHoldout(C1, C2, X(idxTest,:), X(idxTest,:), Y(idxTest))
```

```
h =
    logical
     0
```



$h = 0$  indicates to not reject the null hypothesis that the two models have equal predictive accuracies.

### Assess Whether One Classification Model Classifies Better Than Another

Train two classification models using the same algorithm, but adjust a hyperparameter to make the algorithm more complex. Conduct a statistical test to assess whether the simpler model has better accuracy in held-out data than the more complex model.

Load the `ionosphere` data set.

```
load ionosphere;
```

Create a partition that evenly splits the data into training and testing sets.

```
rng(1); % For reproducibility
CVP = cvpartition(Y, 'holdout', 0.5);
idxTrain = training(CVP); % Training-set indices
idxTest = test(CVP); % Test-set indices
```

CVP is a cross-validation partition object that specifies the training and test sets.

Train two SVM models: one that uses a linear kernel (the default for binary classification) and one that uses the radial basis function kernel. Use the default kernel scale of 1.

```
C1 = fitcsvm(X(idxTrain,:), Y(idxTrain), 'Standardize', true);
C2 = fitcsvm(X(idxTrain,:), Y(idxTrain), 'Standardize', true, ...
    'KernelFunction', 'RBF');
```

C1 and C2 are trained `ClassificationSVM` models.

Test the null hypothesis that the simpler model (C1) is at most as accurate as the more complex model (C2). Because the test-set size is large, conduct the asymptotic McNemar test, and compare the results with the mid- $p$ -value test (the cost-insensitive testing default). Request to return  $p$ -values and misclassification rates.

```
Asymp = zeros(4,1); % Preallocation
MidP = zeros(4,1);

[Asymp(1), Asymp(2), Asymp(3), Asymp(4)] = compareHoldout(C1, C2, ...
    X(idxTest,:), X(idxTest,:), Y(idxTest), 'Alternative', 'greater', ...
```

```

    'Test', 'asymptotic');
[MidP(1), MidP(2), MidP(3), MidP(4)] = compareHoldout(C1, C2, ...
    X(idxTest, :), X(idxTest, :), Y(idxTest), 'Alternative', 'greater');
table(Asymp, MidP, 'RowNames', {'h' 'p' 'e1' 'e2'})

```

```
ans =
```

```
4x2 table
```

	Asymp	MidP
h	1	1
p	7.2801e-09	2.7649e-10
e1	0.13714	0.13714
e2	0.33143	0.33143

The  $p$ -value is close to zero for both tests, which indicates strong evidence to reject the null hypothesis that the simpler model is less accurate than the more complex model. No matter what test you specify, `compareHoldout` returns the same type of misclassification measure for both models.

### Conduct a Cost-Sensitive Comparison of Two Classification Models

For data sets with imbalanced class representations, or for data sets with imbalanced false-positive and false-negative costs, you can statistically compare the predictive performances of two classification models by including a cost matrix in the analysis.

Load the `arrhythmia` data set. Determine the class representations in the data.

```

load arrhythmia;
Y = categorical(Y);
tabulate(Y);

```

Value	Count	Percent
1	245	54.20%
2	44	9.73%
3	15	3.32%
4	15	3.32%
5	13	2.88%

6	25	5.53%
7	3	0.66%
8	2	0.44%
9	9	1.99%
10	50	11.06%
14	4	0.88%
15	5	1.11%
16	22	4.87%

There are 16 classes, however some are not represented in the data set. Most observations are classified as not having arrhythmia (class 1). To summarize, the data set is highly discrete with imbalanced classes.

Combine all observations with arrhythmia (classes 2 through 15) into one class. Remove those observations with unknown arrhythmia status from the data set.

```
Y = Y(Y ~= '16');
Y(Y ~= '1') = '2';
X = X(Y ~= '16', :);
```

Create a partition that evenly splits the data into training and testing sets.

```
rng(1); % For reproducibility
CVP = cvpartition(Y, 'holdout', 0.5);
idxTrain = training(CVP); % Training-set indices
idxTest = test(CVP); % Test-set indices
```

CVP is a cross-validation partition object that specifies the training and test sets.

Create a cost matrix such that misclassifying an arrhythmic patient into the no arrhythmia class is five times worse than misclassifying a patient without arrhythmia into the arrhythmia class. Classifying correctly incurs no cost. The rows indicate the true class and the columns indicate the predicted class. When conducting a cost-sensitive analysis, it is a good practice to specify the order of the classes.

```
cost = [0 1; 5 0];
ClassNames = categorical([2 1]);
```

Train two boosting ensembles of 50 classification trees, one that uses AdaBoostM1 and the other that uses LogitBoost. Because there are missing values, specify to use surrogate splits. Train the models using the cost matrix.

```
t = templateTree('Surrogate', 'on');
numTrees = 50;
```

```
C1 = fitensemble(X(idxTrain,:),Y(idxTrain),'AdaBoostM1',numTrees,t,...  
    'Cost',cost);  
C2 = fitensemble(X(idxTrain,:),Y(idxTrain),'LogitBoost',numTrees,t,...  
    'Cost',cost);
```

C1 and C2 are trained ClassificationEnsemble models.

Test whether the AdaBoostM1 ensemble (C1) and the LogitBoost ensemble (C2) have equal predictive accuracy. Supply the cost matrix. Conduct the asymptotic, likelihood ratio, cost-sensitive test (the default when you pass in a cost matrix). Request to return *p*-values and misclassification costs.

```
[h,p,e1,e2] = compareHoldout(C1,C2,X(idxTest,:),X(idxTest,:),Y(idxTest),...  
    'Cost',cost)
```

h =

```
logical
```

```
0
```

p =

```
0.0743
```

e1 =

```
1.3581
```

e2 =

```
1.6186
```

h = 0 indicates to not reject the null hypothesis that the two models have equal predictive accuracies.

## Select Features Using Statistical Accuracy Comparison

Reduce classification model complexity by selecting a subset of predictor variables (features) from a larger set. Then, statistically compare the out-of-sample accuracy between the two models.

Load the ionosphere data set.

```
load ionosphere;
```

Create a partition that evenly splits the data into training and testing sets.

```
rng(1); % For reproducibility
CVP = cvpartition(Y, 'holdout', 0.5);
idxTrain = training(CVP); % Training-set indices
idxTest = test(CVP); % Test-set indices
```

CVP is a cross-validation partition object that specifies the training and test sets.

Train an ensemble of 100 boosted classification trees using AdaBoostM1 and the entire set of predictors. Inspect the importance measure for each predictor.

```
nTrees = 100;
C2 = fitensemble(X(idxTrain, :), Y(idxTrain), 'AdaBoostM1', nTrees, 'Tree');
predImp = predictorImportance(C2);

figure;
bar(predImp);
h = gca;
h.XTick = 1:2:h.XLim(2);
title('Predictor Importances');
xlabel('Predictor');
ylabel('Importance measure');
```

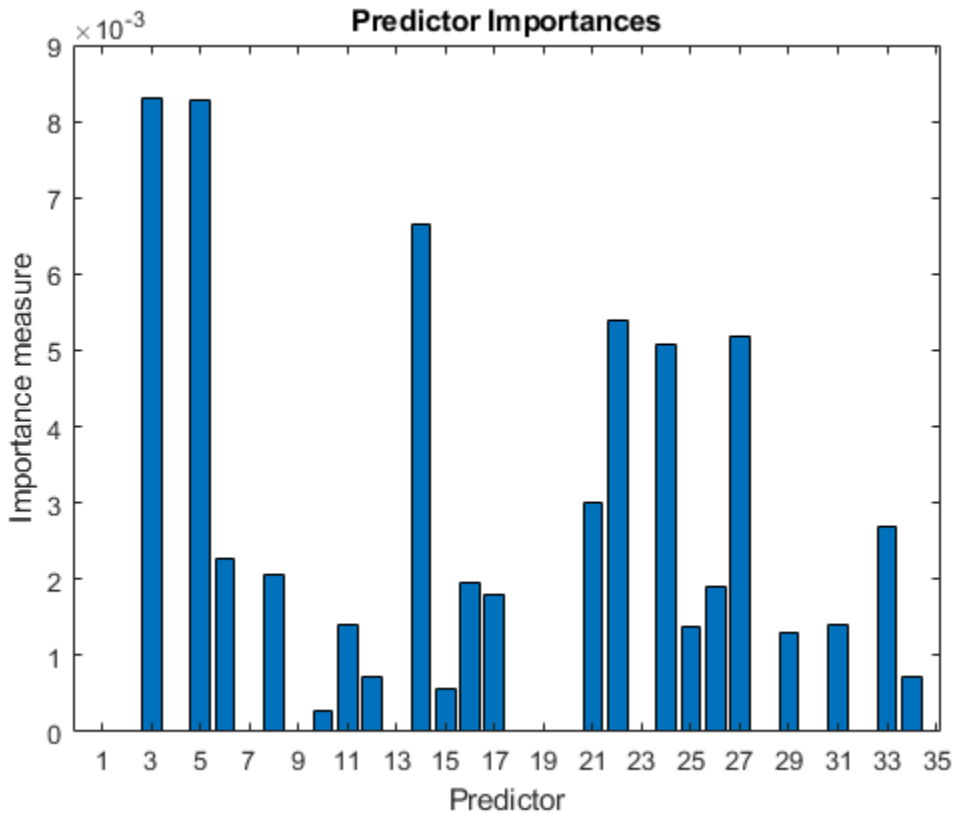
h =

Axes with properties:

```
    XLim: [-0.2000 35.2000]
    YLim: [0 0.0090]
    XScale: 'linear'
    YScale: 'linear'
  GridLineStyle: '-'
    Position: [0.1300 0.1100 0.7750 0.8150]
```

Units: 'normalized'

Use GET to show all properties



Identify the top five predictors in terms of their importance.

```
[~,idxSort] = sort(predImp, 'descend');
idx5 = idxSort(1:5);
```

Train another ensemble of 100 boosted classification trees using AdaBoostM1 and the five predictors with the best importance.

```
C1 = fitensemble(X(idxTrain,idx5),Y(idxTrain), 'AdaBoostM1', nTrees, ...
    'Tree');
```

Test whether the two models have equal predictive accuracies. Specify the reduced test-set predictor data for C1 and the full test-set predictor data for C2.

```
[h,p,e1,e2] = compareHoldout(C1,C2,X(idxTest,idx5),X(idxTest,:),Y(idxTest))
```

```
h =
```

```
    logical
```

```
    0
```

```
p =
```

```
    0.7744
```

```
e1 =
```

```
    0.0914
```

```
e2 =
```

```
    0.0857
```

`h = 0` indicates to not reject the null hypothesis that the two models have equal predictive accuracies. This result favors the simpler ensemble, C1.

## Definitions

### Cost-Sensitive Testing

Conduct cost-sensitive testing when the cost of misclassification is imbalanced. When conducting a cost-sensitive analysis, you can account for the cost imbalance in training the classification models, and then in statistically comparing them.

If the cost of misclassification is imbalanced, then the misclassification rate tends to be a poorly performing classification loss on page 32-6520. Use misclassification cost instead to compare classification models.

Misclassification costs are often unbalanced in applications. For example, consider classifying subjects based on a set of predictors into two categories: healthy and sick. Misclassifying a sick subject as healthy poses a danger to the subject's life. However, misclassifying a healthy subject as sick can cause some inconvenience, but does not pose any danger. In this situation, you assign misclassification costs such that misclassifying a sick subject as healthy is more costly than misclassifying a healthy subject as sick.

The definitions that follow summarize the cost-sensitive tests. In the definitions:

- $n_{ijk}$  and  $\hat{\pi}_{ijk}$  are the number and estimated proportion of test-sample observations with true class  $k$  that the first classification model assigns label  $i$ . The second classification model assigns label  $j$ . The unknown, true value of  $\hat{\pi}_{ijk}$  is  $\pi_{ijk}$ . The test-set sample size is  $\sum_{i,j,k} n_{ijk} = n_{test}$ .  $\sum_{i,j,k} \pi_{ijk} = \sum_{i,j,k} \pi_{ijk} = 1$ .
- $c_{ij}$  is the relative cost of assigning label  $j$  to an observation with true class  $i$ .  $c_{ii} = 0$ ,  $c_{ij} \geq 0$ , and, for at least one  $(i,j)$  pair,  $c_{ij} > 0$ .
- All subscripts take on integer values from 1 through  $K$ , which is the number of classes.
- The expected difference in the misclassification costs of the two classification models is
 
$$\delta = \sum_{i=1}^K \sum_{j=1}^K \sum_{k=1}^K (c_{ki} - c_{kj}) \pi_{ijk}.$$
- The hypothesis test is
 
$$H_0 : \delta = 0$$

$$H_1 : \delta \neq 0$$

The available cost-sensitive tests are appropriate for two-tailed testing.

Available, asymptotic tests that address imbalanced costs are a chi-square test and a likelihood ratio test.

- Chi-square test — The chi-square test statistic is based on the Pearson and Neyman chi-square test statistics, but with a Laplace correction factor to account for any  $n_{ijk} = 0$ . The test statistic is



$$t_{\chi^2}^* = \sum_{i \neq j} \sum_k \frac{(n_{ijk} + 1 - (n_{test} + K^3) \hat{\pi}_{ijk}^{(1)})^2}{n_{ijk} + 1}.$$

If  $1 - F_{\chi^2}(t_{\chi^2}^*; 1) < \alpha$ , then reject  $H_0$ .

•

$\hat{\pi}_{ijk}^{(1)}$  are estimated by minimizing  $t_{\chi^2}^*$  under the constraint that  $\delta = 0$ .

•

$F_{\chi^2}(x; 1)$  is the  $\chi^2$  C.D.F. with one degree of freedom evaluated at  $x$ .

- Likelihood ratio test — The likelihood ratio test is based on  $N_{ijk}$ , which are binomial random variables having sample size  $n_{test}$  and success probability  $\pi_{ijk}$ . They represent the random number of observations with true class  $k$  that the first classification model assigns label  $i$ . The second classification model assigns label  $j$ . Jointly, their distribution is multinomial.

The test statistic is

$$t_{LRT}^* = 2 \log \left[ \frac{P\left(\bigcap_{i,j,k} N_{ijk} = n_{ijk}; n_{test}, \pi_{ijk} = \pi_{ijk}^{(2)}\right)}{P\left(\bigcap_{i,j,k} N_{ijk} = n_{ijk}; n_{test}, \pi_{ijk} = \pi_{ijk}^{(3)}\right)} \right].$$

If  $1 - F_{\chi^2}(t_{LRT}^*; 1) < \alpha$ , then reject  $H_0$ .

•

$\hat{\pi}_{ijk}^{(2)} = \frac{n_{ijk}}{n_{test}}$  is the unrestricted MLE of  $\pi_{ijk}$ .

•

$\hat{\pi}_{ijk}^{(3)} = \frac{n_{ijk}}{n_{test} + \lambda(c_{ki} - c_{kj})}$  is the MLE under the null hypothesis that  $\delta = 0$ .  $\lambda$  is the solution to

$$\sum_{i,j,k} \frac{n_{ijk}(c_{ki} - c_{kj})}{n_{test} + \lambda(c_{ki} - c_{kj})} = 0.$$

•

$F_{\chi^2}(x; 1)$  is the  $\chi^2$  C.D.F. with one degree of freedom evaluated at  $x$ .

## McNemar Tests

McNemar Tests are hypothesis tests that compare two population proportions while addressing the issues resulting from two dependent, matched-pair samples.

One way to compare the predictive accuracies of two classification models is:

- 1 Partition the data into training and test sets.
- 2 Train both classification models using the training set.
- 3 Predict class labels using the test set.
- 4 Summarize the results in a two-by-two table resembling this figure.

		Model 2		
		Correct	Incorrect	
Model 1	Correct	$n_{11}$	$n_{12}$	$n_{1\bullet}$
	Incorrect	$n_{21}$	$n_{22}$	$n_{2\bullet}$
		$n_{\bullet 1}$	$n_{\bullet 2}$	$n_{test}$

$n_{ii}$  are the number of concordant pairs, that is, the number of observations that both models classify the same way (correctly or incorrectly).  $n_{ij}$ ,  $i \neq j$ , are the number of discordant pairs, that is, the number of observations that models classify differently (correctly or incorrectly).

The misclassification rates for Models 1 and 2 are

$$\hat{\pi}_{2\bullet} = n_{2\bullet} / n$$

and  $\hat{\pi}_{\bullet 2} = n_{\bullet 2} / n$ , respectively. A two-sided test for comparing the accuracy of the two models is

$$H_0 : \pi_{\bullet 2} = \pi_{2\bullet}$$

$$H_1 : \pi_{\bullet 2} \neq \pi_{2\bullet}$$

The null hypothesis suggests that the population exhibits marginal homogeneity, which reduces the null hypothesis to  $H_0 : \pi_{12} = \pi_{21}$ . Also, under the null hypothesis,  $N_{12} \sim \text{Binomial}(n_{12} + n_{21}, 0.5)$  [1].

These facts are the basis for these, available McNemar test variants: the asymptotic, exact-conditional and mid-p-value McNemar tests. The definitions that follow summarize the available variants.

- Asymptotic — The asymptotic McNemar test statistics and rejection regions (for significance-level  $\alpha$ ) are:

- For one-sided tests, the test statistic

$$t_{a1}^* = \frac{n_{12} - n_{21}}{\sqrt{n_{12} + n_{21}}}.$$

If  $1 - \Phi\left(\left|t_1^*\right|\right) < \alpha$ , where  $\Phi$  is the standard Gaussian C.D.F., then reject  $H_0$ .

- For two-sided tests, the test statistic

$$t_{a2}^* = \frac{(n_{12} - n_{21})^2}{n_{12} + n_{21}}.$$

If  $1 - F_{\chi^2}\left(t_2^*; m\right) < \alpha$ , where  $F_{\chi^2}(x; m)$  is the  $\chi_m^2$  C.D.F. evaluated at  $x$ , then reject  $H_0$ .

This variant requires large-sample theory, specifically, the Gaussian approximation to the binomial distribution. Therefore:

- The total number of discordant pairs,  $n_d = n_{12} + n_{21}$  must be greater than 10 ([1], Ch. 10.1.4).
- In general, asymptotic tests do not guarantee nominal coverage. The observed probability of falsely rejecting the null hypothesis can exceed  $\alpha$ . Simulation studies in [18] suggest this, but the asymptotic McNemar test performs well in terms of statistical power.
- Exact Conditional — The exact-conditional McNemar test statistics and rejection regions (for significance-level  $\alpha$ ) are ([36], [38]):

- For one-sided tests, the test statistic

$$t_1^* = n_{12}$$

If  $F_{\text{Bin}}(t_1^*; n_d, 0.5) < \alpha$ , where  $F_{\text{Bin}}(x; n, p)$  is the binomial C.D.F. with sample size  $n$  and success probability  $p$  evaluated at  $x$ , then reject  $H_0$ .

- For two-sided tests, the test statistic

$$t_2^* = \min(n_{12}, n_{21})$$

If  $F_{\text{Bin}}(t_2^*; n_d, 0.5) < \alpha / 2$ , then reject  $H_0$ .

The exact conditional test always attains nominal coverage. Simulation studies in [18] suggest that the test is conservative, and then show that the test lacks statistical power compared to other variants. For small or highly discrete test samples, consider using the mid- $p$ -value test ([1], Ch. 3.6.3). For details, see `TEST` and “McNemar Tests” on page 32-6517.

- Mid- $p$ -value test — The mid- $p$ -value McNemar test statistics and rejection regions (for significance-level  $\alpha$ ) are ([32]):

- For one-sided tests, the test statistic

$$t_1^* = n_{12}$$

If  $F_{\text{Bin}}(t_1^* - 1; n_{12} + n_{21}, 0.5) + 0.5f_{\text{Bin}}(t_1^*; n_{12} + n_{21}, 0.5) < \alpha$ , where  $F_{\text{Bin}}(x; n, p)$  and  $f_{\text{Bin}}(x; n, p)$  are the binomial C.D.F. and P.D.F, respectively, with sample size  $n$  and success probability  $p$  evaluated at  $x$ , then reject  $H_0$ .

- For two-sided tests, the test statistic

$$t_2^* = \min(n_{12}, n_{21})$$

If  $F_{\text{Bin}}(t_2^* - 1; n_{12} + n_{21} - 1, 0.5) + 0.5f_{\text{Bin}}(t_2^*; n_{12} + n_{21}, 0.5) < \alpha / 2$ , then reject  $H_0$ .

The mid- $p$ -value test addresses the over-conservative behavior of the exact conditional test. The simulation studies in [18] demonstrate that this test attains nominal coverage, and has good statistical power.

## Classification Loss

Classification losses indicate the accuracy of a classification model or set of predicted labels. Two classification losses are misclassification rate and cost.

`CompactClassificationDiscriminant.compareHoldout` returns the classification losses (see  $e_1$  and  $e_2$ ) under the alternative hypothesis (i.e., the unrestricted classification losses).  $n_{ijk}$  is the number of test-sample observations with true class  $k$  that the first classification model assigns label  $i$  and the second classification model assigns

label  $j$ , and the corresponding estimated proportion is  $\hat{\pi}_{ijk} = \frac{n_{ijk}}{n_{test}}$ . The test-set sample

size is  $\sum_{i,j,k} n_{ijk} = n_{test}$ . The indices are taken from 1 through  $K$ , the number of classes.

- Misclassification rate, or classification error, is a scalar in the interval  $[0,1]$  representing the proportion of misclassified observations. That is, the misclassification rate for the first classification model is

$$e_1 = \sum_{j=1}^K \sum_{k=1}^K \sum_{i \neq k} \hat{\pi}_{ijk}.$$

For the misclassification rate of the second classification model ( $e_2$ ), switch the indices  $i$  and  $j$  in the formula.

Classification accuracy decreases as the misclassification rate increases to 1.

- Misclassification cost is a nonnegative scalar and is a measure of classification quality relative to the values the specified cost matrix elements. Its interpretation depends on the specified costs of misclassification. Misclassification cost is the weighted average of the costs of misclassification (specified in a cost matrix,  $C$ ) in which the weights are the respective, estimated proportions of misclassified observations. The misclassification cost for the first classification model is

$$e_1 = \sum_{j=1}^K \sum_{k=1}^K \sum_{i \neq k} \hat{\pi}_{ijk} c_{ki},$$

where  $c_{kj}$  is the cost of classifying an observation into class  $j$  if its true class is  $k$ . For the misclassification cost of the second classification model ( $e_2$ ), switch the indices  $i$  and  $j$  in the formula.

In general, for a fixed cost matrix, classification accuracy decreases as misclassification cost increases.

## Tips

- One way to perform cost-insensitive feature selection is:
  - 1 Train the first classification model (C1) using the full predictor set.
  - 2 Train the second classification model (C2) using the reduced predictor set.
  - 3 Specify X1 as the full, test-set predictor data and X2 as the reduced test-set predictor data.
  - 4 Enter `compareHoldout(C1,C2,X1,X2,'Alternative','less')`. If `compareHoldout` returns 1, then there is enough evidence to suggest that the classification model that uses fewer predictors performs better than the model that uses the full predictor set.

Alternatively, you can assess whether there is a significant difference between the accuracies of the two models. To perform this assessment, remove the 'Alternative', 'greater' specification in step 4. `compareHoldout` conducts a two-sided test, and `h = 0` indicates that there is not enough evidence to suggest a difference in the accuracy of the two models.

- Cost-sensitive tests perform numerical optimization, which requires additional computational resources. The likelihood ratio test conducts numerical optimization indirectly by finding the root of a Lagrange multiplier in an interval. For some data sets, if the root lies close to the boundaries of the interval, then the method can fail. Therefore, if you have an Optimization Toolbox license, consider conducting the cost-sensitive chi-square test instead. For more details, see `CostTest` and “Cost-Sensitive Testing” on page 32-1053.

## Alternatives

To directly compare the accuracy of two sets of class labels in predicting a set of true class labels, use `testcholdout`.

## References

- [1] Agresti, A. *Categorical Data Analysis*, 2nd Ed. John Wiley & Sons, Inc.: Hoboken, NJ, 2002.
- [2] Fagerlan, M.W., S Lydersen, P. Laake. “The McNemar Test for Binary Matched-Pairs Data: Mid-p and Asymptotic Are Better Than Exact Conditional.” *BMC Medical Research Methodology*. Vol. 13, 2013, pp. 1–8.
- [3] Lancaster, H.O. “Significance Tests in Discrete Distributions.” *JASA*, Vol. 56, Number 294, 1961, pp. 223–234.
- [4] McNemar, Q. “Note on the Sampling Error of the Difference Between Correlated Proportions or Percentages.” *Psychometrika*, Vol. 12, Number 2, 1947, pp. 153–157.
- [5] Mosteller, F. “Some Statistical Problems in Measuring the Subjective Response to Drugs.” *Biometrics*, Vol. 8, Number 3, 1952, pp. 220–226.

## See Also

`fitcdiscr` | `predict` | `testcholdout` | `testckfold`

## Topics

“Hypothesis Tests”

Introduced in R2015a

## compareHoldout

**Class:** CompactClassificationECOC

Compare accuracies of two classification models using new data

`compareHoldout` statistically assesses the accuracies of two classification models. The function first compares their predicted labels against the true labels, and then it detects whether the difference between the misclassification rates is statistically significant.

You can assess whether the accuracies of the classification models are different, or whether one classification model performs better than another. `compareHoldout` can conduct several McNemar test on page 32-1055 variations, including the asymptotic test, the exact-conditional test, and the mid- $p$ -value test. For cost-sensitive assessment on page 32-1053, available tests include a chi-square test (requires an Optimization Toolbox license) and a likelihood ratio test.

### Syntax

```
h = compareHoldout(C1,C2,T1,T2,ResponseVarName)
```

```
h = compareHoldout(C1,C2,T1,T2,Y)
```

```
h = compareHoldout( ___, Name, Value)
```

```
h = compareHoldout(C1,C2,X1,X2,Y)
```

```
h = compareHoldout(C1,C2,X1,X2,Y, Name, Value)
```

```
[h,p,e1,e2] = compareHoldout( ___)
```

### Description

`h = compareHoldout(C1,C2,T1,T2,ResponseVarName)` returns the test decision from testing the null hypothesis that the trained classification models `C1` and `C2` have equal accuracy for predicting the true class labels `ResponseVarName`. The alternative hypothesis is that the labels have unequal accuracy.

The first classification model `C1` uses predictor data `T1` and `C2` uses `T2`. The software conducts the mid- $p$ -value McNemar test to compare the accuracies.



$h = 1$  indicates to reject the null hypothesis at the 5% significance level.  $h = 0$  indicates to not reject the null hypothesis at 5% level.

Examples of tests you can conduct include:

- Compare the accuracies of a simple classification model and a model that is more complex by passing the same set of predictor data (i.e.,  $X_1 = X_2$ ).
- Compare the accuracies of two perhaps different models using two perhaps different sets of predictors.
- Perform various types of feature selection. For example, you can compare the accuracy of a model trained using a set of predictors to the accuracy of one trained on a subset or different set of those predictors. You can arbitrarily choose the set of predictors, or use a feature selection technique like PCA or sequential feature selection (see `pca` and `sequentialfs`).

`h = compareHoldout(C1,C2,T1,T2,Y)` returns the test decision from testing the null hypothesis that the trained classification models `C1` and `C2` have equal accuracy for predicting the true class labels `Y`. The alternative hypothesis is that the labels have unequal accuracy.

`h = compareHoldout(____,Name,Value)` returns the result of the hypothesis test with additional options specified by one or more `Name,Value` pair arguments, using any of the previous syntaxes. For example, you can specify the type of alternative hypothesis, and the type of test, or you can supply a cost matrix.

`h = compareHoldout(C1,C2,X1,X2,Y)` returns the test decision from testing the null hypothesis that the trained classification models `C1` and `C2` have equal accuracy for predicting the true class labels `Y`. The alternative hypothesis is that the labels have unequal accuracy.

The first classification model `C1` uses predictor data `X1` and `C2` uses `X2`. The software conducts the mid- $p$ -value McNemar test to compare the accuracies.

$h = 1$  indicates to reject the null hypothesis at the 5% significance level.  $h = 0$  indicates to not reject the null hypothesis at 5% level.

`h = compareHoldout(C1,C2,X1,X2,Y,Name,Value)` returns the result of the hypothesis test with additional options specified by one or more `Name,Value` pair arguments. For example, you can specify the type of alternative hypothesis, and the type of test, or you can supply a cost matrix.

`[h,p,e1,e2] = compareHoldout(____)` returns the  $p$ -value for the hypothesis test ( $p$ ) and the respective classification losses on page 32-977 of each set of predicted class labels ( $e1$  and  $e2$ ) using any of the input arguments in the previous syntaxes.

## Limitations

- `compareHoldout` does not compare ECOC models composed of linear classification models (that is, `ClassificationLinear` model objects). To compare ECOC models composed of linear classification models, use `testcholdout` instead.

## Input Arguments

### **c1 — Trained, full or compact, multiclass error-correcting output codes model**

`ClassificationECOC` model object | `CompactClassificationECOC` model object

Trained, full or compact, multiclass, error-correcting output codes (ECOC) model, specified as a `ClassificationECOC` or `CompactClassificationECOC` model object. That is, `C1` is a trained classification model returned by `fitcecoc` or `compact`.

`C1.BinaryLearners` cannot contain `ClassificationLinear` model objects.

### **c2 — Trained classification model**

trained classification model object | trained, compact classification model object

Trained classification model, specified as any trained or compact classification model object described in this table.

Trained Model Type	Model Object	Returned By
Classification tree	<code>ClassificationTree</code>	<code>fitctree</code>
Discriminant analysis	<code>ClassificationDiscriminant</code>	<code>fitcdiscr</code>
Ensemble of bagged classification models	<code>ClassificationBaggedEnsemble</code>	<code>fitcensemble</code>
Ensemble of classification models	<code>ClassificationEnsemble</code>	<code>fitcensemble</code>

Trained Model Type	Model Object	Returned By
Multiclass, error-correcting output codes (ECOC)	ClassificationECOC	fitcecoc
$k$ NN	ClassificationKNN	fitcknn
Naive Bayes	ClassificationNaiveBayes	fitcnb
Support vector machine (SVM)	ClassificationSVM	fitcsvm
Compact discriminant analysis	CompactClassificationDiscriminant	compact
Compact ECOC	CompactClassificationECOC	compact
Compact ensemble of classification models	CompactClassificationEnsemble	compact
Compact naive Bayes	CompactClassificationNaiveBayes	compact
Compact SVM	CompactClassificationSVM	compact
Compact classification tree	CompactClassificationTree	compact

### **T1 — Test-set predictor table for first classification model**

table

Test-set predictor table for the first classification model, C1, specified as a table. Each row of T1 corresponds to one observation, and each column corresponds to one predictor variable. Optionally, T1 can contain an additional column for the response variable. T1 must contain all of the predictors used to train C1. Multi-column variables and cell arrays other than cell arrays of character vectors are not allowed.

T1 and T2 must have the same number of rows, and must have the same response values. If T1 and T2 contain the response variable used to train C1 and C2, then, you do not need to specify ResponseVarName or Y.

Data Types: table

### **T2 — Test-set predictor table for second classification model**

table

Test-set predictor data for the second classification model, `C2`, specified as a table. Each row of `T2` corresponds to one observation, and each column corresponds to one predictor variable. Optionally, `T2` can contain an additional column for the response variable. `T2` must contain all of the predictors used to train `C2`. Multi-column variables and cell arrays other than cell arrays of character vectors are not allowed.

`T1` and `T2` must have the same number of rows and the same response values. If `T1` and `T2` contain the response variable used to train `C1` and `C2`, then, you do not need to specify `ResponseVarName` or `Y`.

Data Types: `table`

#### **`ResponseVarName` — Response variable name**

name of a variable in `T1` and `T2`

Response variable name, specified as the name of a variable in `T1` and `T2`. If `T1` and `T2` contain the response variable used to train `C1` and `C2`, then you do not need to specify `ResponseVarName`.

If you specify `ResponseVarName`, then you must do so as a character vector. For example, if the response variable is stored as `T1.Response`, then specify it as `'Response'`. Otherwise, the software treats all columns of `T1` and `T2`, including `Response`, as predictors.

The response variable must be a categorical or character array, logical or numeric vector, or cell array of character vectors. If the response variable is a character array, then each element must correspond to one row of the array.

#### **`x1` — Test-set predictor data for first classification model**

numeric matrix

Test-set predictor data for the first classification model, `C1`, specified as a numeric matrix.

Each row of `X1` corresponds to one observation (also known as an instance or example), and each column corresponds to one variable (also known as a predictor or feature). The variables used to train `C1` must compose `X1`.

The number of rows in `X1` and `X2` must equal the length of `Y`.

Data Types: `double` | `single`

**x2 — Test-set predictor data for second classification model**

numeric matrix

Test-set predictor data for the second classification model, `C2`, specified as a numeric matrix.

Each row of `X2` corresponds to one observation (also known as an instance or example), and each column corresponds to one variable (also known as a predictor or feature). The variables used to train `C2` must compose `X2`.

The number of rows in `X2` and `X1` must equal the length of `Y`.

Data Types: `double` | `single`

**y — True class labels**

categorical array | character array | logical vector | vector of numeric values | cell array of character vectors | character vector

True class labels, specified as a categorical or character array, a logical or numeric vector, a cell array of character vectors, or a character vector.

If you specify any array except a character vector, then `CompactClassificationECOC.compareHoldout` treats all variables or columns in `X1` and `X2` as predictor variables.

If you specify a character vector, then `X1` and `X2` must be tables, their response variables must have the same name and values, and `Y` must be the common name. For example, if `X1.Labels` and `X2.Labels` are the response variables, then `Y` must be `'Labels'` and `X1.Labels` and `X2.Labels` must be equivalent.

If `Y` is a character array, then each element must correspond to one row of the array.

The number of rows in `X1` and `X2` must equal the length of `Y`.

Data Types: `categorical` | `char` | `logical` | `single` | `double` | `cell`

**Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'Alternative', 'greater', 'Test', 'asymptotic', 'Cost', [0 2; 1 0]` specifies to test whether the first set of first predicted class labels is more accurate than the second set, to conduct the asymptotic McNemar test, and to penalize misclassifying observations with the true label `ClassNames{1}` twice as much as for misclassifying observations with the true label `ClassNames{2}`.

### **Alpha — Hypothesis test significance level**

0.05 (default) | scalar value in the interval (0,1)

Hypothesis test significance level, specified as the comma-separated pair consisting of `'Alpha'` and a scalar value in the interval (0,1).

Example: `'Alpha', 0.1`

Data Types: `single` | `double`

### **Alternative — Alternative hypothesis to assess**

`'unequal'` (default) | `'greater'` | `'less'`

Alternative hypothesis to assess, specified as the comma-separated pair consisting of `'Alternative'` and one of these values listed in the table.

Value	Alternative hypothesis
<code>'unequal'</code> (default)	For predicting Y, the set of predictions resulting from C1 applied to X1 and C2 applied to X2 have unequal accuracies.
<code>'greater'</code>	For predicting Y, the set of predictions resulting from C1 applied to X1 is more accurate than C2 applied to X2.
<code>'less'</code>	For predicting Y, the set of predictions resulting from C1 applied to X1 is less accurate than C2 applied to X2.

Example: `'Alternative', 'greater'`

Data Types: `char`

### **ClassNames — Class names**

categorical vector | cell array of character vectors | character array | logical vector | numeric vector

Class names, specified as the comma-separated pair consisting of `'ClassNames'` and a categorical or character array, logical or numeric vector, or cell array of character vectors. You must set `ClassNames` using the data type of Y.

If `ClassNames` is a character array, then each element must correspond to one *row* of the array.

Use `ClassNames` to:

- Specify the order of any input argument dimension that corresponds to class order. For example, use `ClassNames` to specify the order of the dimensions of `Cost`.
- Select a subset of classes for testing. For example, suppose that the set of all distinct class names in  $Y$  is `{'a', 'b', 'c'}`. To train and test models using observations from classes 'a' and 'c' only, specify `'ClassNames', {'a', 'c'}`.

The default is the set of all distinct class names in  $Y$ .

Example: `'ClassNames', {'b', 'g'}`

Data Types: `categorical | char | logical | single | double | cell`

### **Cost — Misclassification cost**

`square matrix | structure array`

Misclassification cost, specified as the comma-separated pair consisting of `'Cost'` and a square matrix or structure array. If you specify:

- If you specify the square matrix `Cost`, then `Cost(i, j)` is the cost of classifying a point into class  $j$  if its true class is  $i$ . That is, the rows correspond to the true class and the columns correspond to the predicted class. To specify the class order for the corresponding rows and columns of `Cost`, additionally specify the `ClassNames` name-value pair argument.
- If you specify the structure `S`, then `S` must have two fields:
  - `S.ClassNames`, which contains the class names as a variable of the same data type as  $Y$ . You can use this field to specify the order of the classes.
  - `S.ClassificationCosts`, which contains the cost matrix, with rows and columns ordered as in `S.ClassNames`

If you specify `Cost`, then `CompactClassificationECOC.compareHoldout` cannot conduct one-sided, exact, or mid- $p$  tests. You must also specify `'Alternative', 'unequal', 'Test', 'asymptotic'`. For cost-sensitive testing options, see the `CostTest` name-value pair argument.

It is a best practice to supply the same cost matrix used to train the classification models.

The default is  $\text{Cost}(i, j) = 1$  if  $i \neq j$ , and  $\text{Cost}(i, j) = 0$  if  $i = j$ .

Example: 'Cost', [0 1 2 ; 1 0 2; 2 2 0]

Data Types: double | single | struct

### **CostTest** — Cost-sensitive test type

'likelihood' (default) | 'chisquare'

Cost-sensitive test type, specified as the comma-separated pair consisting of 'CostTest' and 'chisquare' or 'likelihood'. Unless you specify a cost matrix using the Cost name-value pair argument, CompactClassificationECOC.compareHoldout ignores CostTest.

This table summarizes the available options for cost-sensitive testing.

Value	Asymptotic test type	Requirements
'chisquare'	Chi-square test	Optimization Toolbox license to implement quadprog
'likelihood'	Likelihood ratio test	None

For more details, see “Cost-Sensitive Testing” on page 32-6515.

Example: 'CostTest', 'chisquare'

Data Types: char

### **Test** — Test to conduct

'asymptotic' | 'exact' | 'midp'

Test to conduct, specified as the comma-separated pair consisting of 'Test' and 'asymptotic', 'exact', and 'midp'. This table summarizes the available options for cost-insensitive testing.

Value	Description
'asymptotic'	Asymptotic McNemar test
'exact'	Exact-conditional McNemar test
'midp' (default)	Mid- <i>p</i> -value McNemar test

For more details, see “McNemar Tests” on page 32-6517.



For cost-sensitive testing, `Test` must be `'asymptotic'`. When you specify the `Cost` name-value pair argument, and choose a cost-sensitive test using the `CostTest` name-value pair argument, `'asymptotic'` is the default.

Example: `'Test','asymptotic'`

Data Types: `char`

---

**Note** NaNs, `<undefined>` values, and empty character vectors ( `' '` ) indicate missing values. `CompactClassificationECOC.compareHoldout`:

- Removes missing values in `Y` and the corresponding rows of `X1` and `X2`
  - Predicts classes whether `X1` and `X2` have missing observations.
- 

## Output Arguments

### **h** — Hypothesis test result

1 | 0

Hypothesis test result, returned as a logical value.

`h = 1` indicates the rejection of the null hypothesis at the Alpha significance level.

`h = 0` indicates failure to reject the null hypothesis at the Alpha significance level.

### **p** — *p*-value

scalar in the interval [0,1]

*p*-value of the test, returned as a scalar in the interval [0,1]. *p* is the probability that a random test statistic is at least as extreme as the observed test statistic, given that the null hypothesis is true.

`CompactClassificationECOC.compareHoldout` estimates *p* using the distribution of the test statistic, which varies with the type of test. For details on test statistics derived from the available variants of the McNemar test, see “McNemar Tests” on page 32-6517. For details on test statistics derived from cost-sensitive tests, see “Cost-Sensitive Testing” on page 32-6515.

**e1 — Classification loss**

scalar

Classification loss on page 32-1058, returned as a scalar. `e1` summarizes the accuracy of the first set of class labels predicting the true class labels ( $Y$ ).

`CompactClassificationECOC.compareHoldout` applies the first test-set predictor data ( $X1$ ) to the first classification model ( $C1$ ) to estimate the first set of class labels. Then, the function compares the estimated labels to  $Y$  to obtain the classification loss.

For cost-insensitive testing, `e1` is the misclassification rate. That is, `e1` is the proportion of misclassified observations, which is a scalar in the interval  $[0,1]$ .

For cost-sensitive testing, `e1` is the misclassification cost. That is, `e1` is the weighted average of the misclassification costs, in which the weights are the respective estimated proportions of misclassified observations.

**e2 — Classification loss**

scalar

Classification loss on page 32-1058, returned as a scalar. `e2` summarizes the accuracy of the second set of class labels predicting the true class labels ( $Y$ ).

`CompactClassificationECOC.compareHoldout` applies the second test-set predictor data ( $X2$ ) to the second classification model ( $C2$ ) to estimate the second set of class labels. Then the function compares the estimated labels to  $Y$  to obtain the classification loss.

For cost-insensitive testing, `e2` is the misclassification rate. That is, `e2` is the proportion of misclassified observations, which is a scalar in the interval  $[0,1]$ .

For cost-sensitive testing, `e2` is the misclassification cost. That is, `e2` is the weighted average of the misclassification costs, in which the weights are the respective estimated proportions of misclassified observations.

## Examples

**Compare Accuracies of Two Different Classification Models**

Train two classification models using different algorithms. Conduct a statistical test comparing the misclassification rates of the two models on a held-out set.

Load the ionosphere data set.

```
load ionosphere;
```

Create a partition that evenly splits the data into training and testing sets.

```
rng(1); % For reproducibility
CVP = cvpartition(Y, 'holdout', 0.5);
idxTrain = training(CVP); % Training-set indices
idxTest = test(CVP); % Test-set indices
```

CVP is a cross-validation partition object that specifies the training and test sets.

Train an SVM model and an ensemble of 100 bagged classification trees. For the SVM model, specify to use the radial basis function kernel and a heuristic procedure to determine the kernel scale.

```
C1 = fitcsvm(X(idxTrain,:), Y(idxTrain), 'Standardize', true, ...
    'KernelFunction', 'RBF', 'KernelScale', 'auto');
C2 = fitensemble(X(idxTrain,:), Y(idxTrain), 'Bag', 100, 'Tree', ...
    'Type', 'classification');
```

C1 is a trained ClassificationSVM model. C2 is a trained ClassificationBaggedEnsemble model.

Test whether the two models have equal predictive accuracies. Use the same test-set predictor data for each model.

```
h = compareHoldout(C1, C2, X(idxTest,:), X(idxTest,:), Y(idxTest))
```

```
h =
    logical
     0
```

h = 0 indicates to not reject the null hypothesis that the two models have equal predictive accuracies.

### Assess Whether One Classification Model Classifies Better Than Another

Train two classification models using the same algorithm, but adjust a hyperparameter to make the algorithm more complex. Conduct a statistical test to assess whether the simpler model has better accuracy in held-out data than the more complex model.

Load the `ionosphere` data set.

```
load ionosphere;
```

Create a partition that evenly splits the data into training and testing sets.

```
rng(1); % For reproducibility
CVP = cvpartition(Y, 'holdout', 0.5);
idxTrain = training(CVP); % Training-set indices
idxTest = test(CVP); % Test-set indices
```

CVP is a cross-validation partition object that specifies the training and test sets.

Train two SVM models: one that uses a linear kernel (the default for binary classification) and one that uses the radial basis function kernel. Use the default kernel scale of 1.

```
C1 = fitcsvm(X(idxTrain,:), Y(idxTrain), 'Standardize', true);
C2 = fitcsvm(X(idxTrain,:), Y(idxTrain), 'Standardize', true, ...
    'KernelFunction', 'RBF');
```

C1 and C2 are trained `ClassificationSVM` models.

Test the null hypothesis that the simpler model (C1) is at most as accurate as the more complex model (C2). Because the test-set size is large, conduct the asymptotic McNemar test, and compare the results with the mid- $p$ -value test (the cost-insensitive testing default). Request to return  $p$ -values and misclassification rates.

```
Asymp = zeros(4,1); % Preallocation
MidP = zeros(4,1);

[Asymp(1), Asymp(2), Asymp(3), Asymp(4)] = compareHoldout(C1, C2, ...
    X(idxTest,:), X(idxTest,:), Y(idxTest), 'Alternative', 'greater', ...
    'Test', 'asymptotic');
[MidP(1), MidP(2), MidP(3), MidP(4)] = compareHoldout(C1, C2, ...
    X(idxTest,:), X(idxTest,:), Y(idxTest), 'Alternative', 'greater');
table(Asymp, MidP, 'RowNames', {'h' 'p' 'e1' 'e2'})
```

```
ans =
```

```
4x2 table
```

	Asymp	MidP
h	1	1
p	7.2801e-09	2.7649e-10
e1	0.13714	0.13714
e2	0.33143	0.33143

The  $p$ -value is close to zero for both tests, which indicates strong evidence to reject the null hypothesis that the simpler model is less accurate than the more complex model. No matter what test you specify, `compareHoldout` returns the same type of misclassification measure for both models.

### Conduct a Cost-Sensitive Comparison of Two Classification Models

For data sets with imbalanced class representations, or for data sets with imbalanced false-positive and false-negative costs, you can statistically compare the predictive performances of two classification models by including a cost matrix in the analysis.

Load the `arrhythmia` data set. Determine the class representations in the data.

```
load arrhythmia;
Y = categorical(Y);
tabulate(Y);
```

Value	Count	Percent
1	245	54.20%
2	44	9.73%
3	15	3.32%
4	15	3.32%
5	13	2.88%
6	25	5.53%
7	3	0.66%
8	2	0.44%
9	9	1.99%
10	50	11.06%

```
14      4      0.88%
15      5      1.11%
16     22      4.87%
```

There are 16 classes, however some are not represented in the data set. Most observations are classified as not having arrhythmia (class 1). To summarize, the data set is highly discrete with imbalanced classes.

Combine all observations with arrhythmia (classes 2 through 15) into one class. Remove those observations with unknown arrhythmia status from the data set.

```
Y = Y(Y ~= '16');
Y(Y ~= '1') = '2';
X = X(Y ~= '16',:);
```

Create a partition that evenly splits the data into training and testing sets.

```
rng(1); % For reproducibility
CVP = cvpartition(Y, 'holdout', 0.5);
idxTrain = training(CVP); % Training-set indices
idxTest = test(CVP); % Test-set indices
```

CVP is a cross-validation partition object that specifies the training and test sets.

Create a cost matrix such that misclassifying an arrhythmic patient into the no arrhythmia class is five times worse than misclassifying a patient without arrhythmia into the arrhythmia class. Classifying correctly incurs no cost. The rows indicate the true class and the columns indicate the predicted class. When conducting a cost-sensitive analysis, it is a good practice to specify the order of the classes.

```
cost = [0 1; 5 0];
ClassNames = categorical([2 1]);
```

Train two boosting ensembles of 50 classification trees, one that uses AdaBoostM1 and the other that uses LogitBoost. Because there are missing values, specify to use surrogate splits. Train the models using the cost matrix.

```
t = templateTree('Surrogate', 'on');
numTrees = 50;
C1 = fitensemble(X(idxTrain,:), Y(idxTrain), 'AdaBoostM1', numTrees, t, ...
    'Cost', cost);
C2 = fitensemble(X(idxTrain,:), Y(idxTrain), 'LogitBoost', numTrees, t, ...
    'Cost', cost);
```

C1 and C2 are trained ClassificationEnsemble models.

Test whether the AdaBoostM1 ensemble (C1) and the LogitBoost ensemble (C2) have equal predictive accuracy. Supply the cost matrix. Conduct the asymptotic, likelihood ratio, cost-sensitive test (the default when you pass in a cost matrix). Request to return  $p$ -values and misclassification costs.

```
[h,p,e1,e2] = compareHoldout(C1,C2,X(idxTest,:),X(idxTest,:),Y(idxTest),...  
    'Cost',cost)
```

```
h =
```

```
    logical
```

```
    0
```

```
p =
```

```
    0.0743
```

```
e1 =
```

```
    1.3581
```

```
e2 =
```

```
    1.6186
```

$h = 0$  indicates to not reject the null hypothesis that the two models have equal predictive accuracies.

### Select Features Using Statistical Accuracy Comparison

Reduce classification model complexity by selecting a subset of predictor variables (features) from a larger set. Then, statistically compare the out-of-sample accuracy between the two models.

Load the ionosphere data set.

```
load ionosphere;
```

Create a partition that evenly splits the data into training and testing sets.

```
rng(1); % For reproducibility
CVP = cvpartition(Y, 'holdout', 0.5);
idxTrain = training(CVP); % Training-set indices
idxTest = test(CVP); % Test-set indices
```

CVP is a cross-validation partition object that specifies the training and test sets.

Train an ensemble of 100 boosted classification trees using AdaBoostM1 and the entire set of predictors. Inspect the importance measure for each predictor.

```
nTrees = 100;
C2 = fitensemble(X(idxTrain,:), Y(idxTrain), 'AdaBoostM1', nTrees, 'Tree');
predImp = predictorImportance(C2);

figure;
bar(predImp);
h = gca;
h.XTick = 1:2:h.XLim(2)
title('Predictor Importances');
xlabel('Predictor');
ylabel('Importance measure');
```

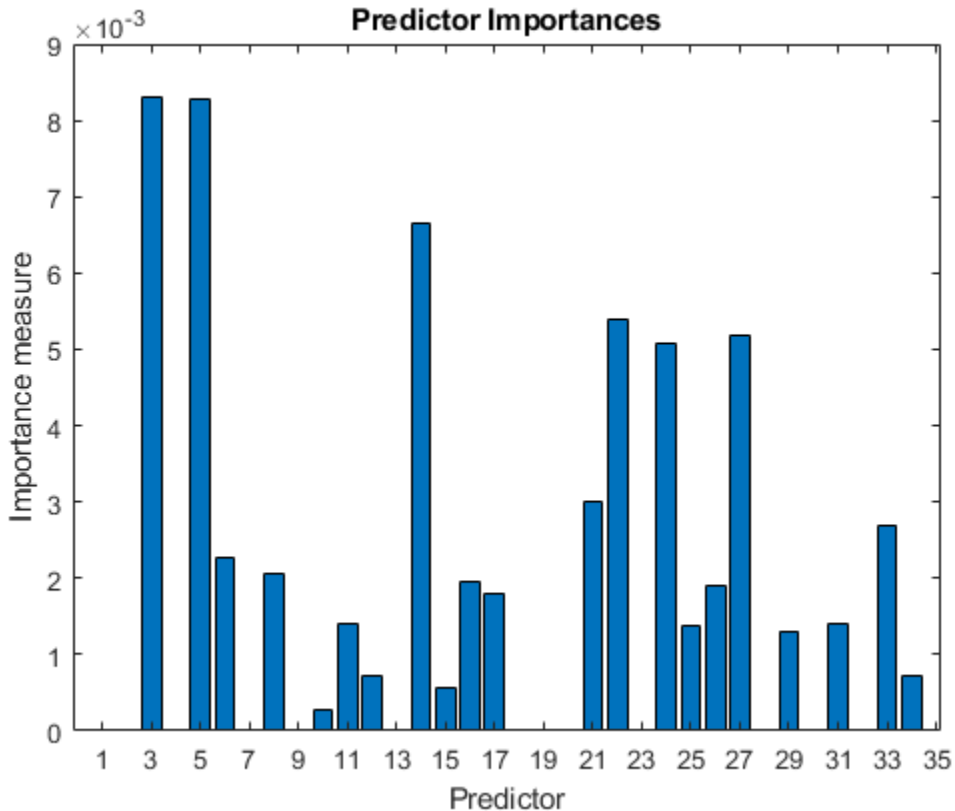
```
h =
```

```
  Axes with properties:
```

```
      XLim: [-0.2000 35.2000]
      YLim: [0 0.0090]
      XScale: 'linear'
      YScale: 'linear'
  GridLineStyle: '-'
      Position: [0.1300 0.1100 0.7750 0.8150]
      Units: 'normalized'
```

```
Use GET to show all properties
```





Identify the top five predictors in terms of their importance.

```
[~,idxSort] = sort(predImp, 'descend');
idx5 = idxSort(1:5);
```

Train another ensemble of 100 boosted classification trees using AdaBoostM1 and the five predictors with the best importance.

```
C1 = fitensemble(X(idxTrain,idx5),Y(idxTrain), 'AdaBoostM1', nTrees, ...
    'Tree');
```

Test whether the two models have equal predictive accuracies. Specify the reduced test-set predictor data for C1 and the full test-set predictor data for C2.

```
[h,p,e1,e2] = compareHoldout(C1,C2,X(idxTest,idx5),X(idxTest,:),Y(idxTest))
```

```
h =  
  logical  
  0  
  
p =  
  0.7744  
  
e1 =  
  0.0914  
  
e2 =  
  0.0857
```

$h = 0$  indicates to not reject the null hypothesis that the two models have equal predictive accuracies. This result favors the simpler ensemble, C1.

## Definitions

### Cost-Sensitive Testing

Conduct cost-sensitive testing when the cost of misclassification is imbalanced. When conducting a cost-sensitive analysis, you can account for the cost imbalance in training the classification models, and then in statistically comparing them.

If the cost of misclassification is imbalanced, then the misclassification rate tends to be a poorly performing classification loss on page 32-6520. Use misclassification cost instead to compare classification models.

Misclassification costs are often unbalanced in applications. For example, consider classifying subjects based on a set of predictors into two categories: healthy and sick. Misclassifying a sick subject as healthy poses a danger to the subject's life. However,

misclassifying a healthy subject as sick can cause some inconvenience, but does not pose any danger. In this situation, you assign misclassification costs such that misclassifying a sick subject as healthy is more costly than misclassifying a healthy subject as sick.

The definitions that follow summarize the cost-sensitive tests. In the definitions:

- $n_{ijk}$  and  $\hat{\pi}_{ijk}$  are the number and estimated proportion of test-sample observations with true class  $k$  that the first classification model assigns label  $i$ . The second classification model assigns label  $j$ . The unknown, true value of  $\hat{\pi}_{ijk}$  is  $\pi_{ijk}$ . The test-set sample size is  $\sum_{i,j,k} n_{ijk} = n_{test}$ .  $\sum_{i,j,k} \pi_{ijk} = \sum_{i,j,k} \pi_{ijk} = 1$ .
- $c_{ij}$  is the relative cost of assigning label  $j$  to an observation with true class  $i$ .  $c_{ii} = 0$ ,  $c_{ij} \geq 0$ , and, for at least one  $(i,j)$  pair,  $c_{ij} > 0$ .
- All subscripts take on integer values from 1 through  $K$ , which is the number of classes.
- The expected difference in the misclassification costs of the two classification models is

$$\delta = \sum_{i=1}^K \sum_{j=1}^K \sum_{k=1}^K (c_{ki} - c_{kj}) \pi_{ijk}.$$

- The hypothesis test is
 
$$H_0 : \delta = 0$$

$$H_1 : \delta \neq 0$$

The available cost-sensitive tests are appropriate for two-tailed testing.

Available, asymptotic tests that address imbalanced costs are a chi-square test and a likelihood ratio test.

- Chi-square test — The chi-square test statistic is based on the Pearson and Neyman chi-square test statistics, but with a Laplace correction factor to account for any  $n_{ijk} = 0$ . The test statistic is

$$t_{\chi^2}^* = \sum_{i \neq j} \sum_k \frac{\left( n_{ijk} + 1 - (n_{test} + K^3) \hat{\pi}_{ijk}^{(1)} \right)^2}{n_{ijk} + 1}.$$

If  $1 - F_{\chi^2}(t_{\chi^2}^*; 1) < \alpha$ , then reject  $H_0$ .

- 
- $\hat{\pi}_{ijk}^{(1)}$  are estimated by minimizing  $t_{\chi^2}^*$  under the constraint that  $\delta = 0$ .
- $F_{\chi^2}(x; 1)$  is the  $\chi^2$  C.D.F. with one degree of freedom evaluated at  $x$ .
- Likelihood ratio test — The likelihood ratio test is based on  $N_{ijk}$ , which are binomial random variables having sample size  $n_{test}$  and success probability  $\pi_{ijk}$ . They represent the random number of observations with true class  $k$  that the first classification model assigns label  $i$ . The second classification model assigns label  $j$ . Jointly, their distribution is multinomial.

The test statistic is

$$t_{LRT}^* = 2 \log \left[ \frac{P\left(\bigcap_{i,j,k} N_{ijk} = n_{ijk}; n_{test}, \pi_{ijk} = \pi_{ijk}^{(2)}\right)}{P\left(\bigcap_{i,j,k} N_{ijk} = n_{ijk}; n_{test}, \pi_{ijk} = \pi_{ijk}^{(3)}\right)} \right].$$

If  $1 - F_{\chi^2}(t_{LRT}^*; 1) < \alpha$ , then reject  $H_0$ .

- 
- $\hat{\pi}_{ijk}^{(2)} = \frac{n_{ijk}}{n_{test}}$  is the unrestricted MLE of  $\pi_{ijk}$ .
- $\hat{\pi}_{ijk}^{(3)} = \frac{n_{ijk}}{n_{test} + \lambda(c_{ki} - c_{kj})}$  is the MLE under the null hypothesis that  $\delta = 0$ .  $\lambda$  is the solution to
 
$$\sum_{i,j,k} \frac{n_{ijk}(c_{ki} - c_{kj})}{n_{test} + \lambda(c_{ki} - c_{kj})} = 0.$$
- $F_{\chi^2}(x; 1)$  is the  $\chi^2$  C.D.F. with one degree of freedom evaluated at  $x$ .

## McNemar Tests

McNemar Tests are hypothesis tests that compare two population proportions while addressing the issues resulting from two dependent, matched-pair samples.

One way to compare the predictive accuracies of two classification models is:

- 1 Partition the data into training and test sets.
- 2 Train both classification models using the training set.
- 3 Predict class labels using the test set.
- 4 Summarize the results in a two-by-two table resembling this figure.

		Model 2		
		Correct	Incorrect	
Model 1	Correct	$n_{11}$	$n_{12}$	$n_{1\bullet}$
	Incorrect	$n_{21}$	$n_{22}$	$n_{2\bullet}$
		$n_{\bullet 1}$	$n_{\bullet 2}$	$n_{test}$

$n_{ii}$  are the number of concordant pairs, that is, the number of observations that both models classify the same way (correctly or incorrectly).  $n_{ij}$ ,  $i \neq j$ , are the number of discordant pairs, that is, the number of observations that models classify differently (correctly or incorrectly).

The misclassification rates for Models 1 and 2 are

$$\hat{\pi}_{2\bullet} = n_{2\bullet} / n$$

and  $\hat{\pi}_{\bullet 2} = n_{\bullet 2} / n$ , respectively. A two-sided test for comparing the accuracy of the two models is

$$H_0 : \pi_{\bullet 2} = \pi_{2\bullet}$$

$$H_1 : \pi_{\bullet 2} \neq \pi_{2\bullet}$$

The null hypothesis suggests that the population exhibits marginal homogeneity, which reduces the null hypothesis to  $H_0 : \pi_{12} = \pi_{21}$ . Also, under the null hypothesis,  $N_{12} \sim \text{Binomial}(n_{12} + n_{21}, 0.5)$  [1].

These facts are the basis for these, available McNemar test variants: the asymptotic, exact-conditional and mid-p-value McNemar tests. The definitions that follow summarize the available variants.

- Asymptotic — The asymptotic McNemar test statistics and rejection regions (for significance-level  $\alpha$ ) are:

- For one-sided tests, the test statistic

$$t_{a1}^* = \frac{n_{12} - n_{21}}{\sqrt{n_{12} + n_{21}}}.$$

If  $1 - \Phi\left(|t_1^*|\right) < \alpha$ , where  $\Phi$  is the standard Gaussian C.D.F., then reject  $H_0$ .

- For two-sided tests, the test statistic

$$t_{a2}^* = \frac{(n_{12} - n_{21})^2}{n_{12} + n_{21}}.$$

If  $1 - F_{\chi^2}\left(t_2^*; m\right) < \alpha$ , where  $F_{\chi^2}(x; m)$  is the  $\chi_m^2$  C.D.F. evaluated at  $x$ , then reject  $H_0$ .

This variant requires large-sample theory, specifically, the Gaussian approximation to the binomial distribution. Therefore:

- The total number of discordant pairs,  $n_d = n_{12} + n_{21}$  must be greater than 10 ([1], Ch. 10.1.4).
- In general, asymptotic tests do not guarantee nominal coverage. The observed probability of falsely rejecting the null hypothesis can exceed  $\alpha$ . Simulation studies in [18] suggest this, but the asymptotic McNemar test performs well in terms of statistical power.
- Exact Conditional — The exact-conditional McNemar test statistics and rejection regions (for significance-level  $\alpha$ ) are ([36], [38]):

- For one-sided tests, the test statistic

$$t_1^* = n_{12}$$

If  $F_{\text{Bin}}\left(t_1^*; n_d, 0.5\right) < \alpha$ , where  $F_{\text{Bin}}(x; n, p)$  is the binomial C.D.F. with sample size  $n$  and success probability  $p$  evaluated at  $x$ , then reject  $H_0$ .

- For two-sided tests, the test statistic

$$t_2^* = \min(n_{12}, n_{21})$$

If  $F_{\text{Bin}}(t_2^*; n_d, 0.5) < \alpha / 2$ , then reject  $H_0$ .

The exact conditional test always attains nominal coverage. Simulation studies in [18] suggest that the test is conservative, and then show that the test lacks statistical power compared to other variants. For small or highly discrete test samples, consider using the mid- $p$ -value test ([1], Ch. 3.6.3). For details, see `Test` and “McNemar Tests” on page 32-6517.

- Mid- $p$ -value test — The mid- $p$ -value McNemar test statistics and rejection regions (for significance-level  $\alpha$ ) are ([32]):

- For one-sided tests, the test statistic

$$t_1^* = n_{12}$$

If  $F_{\text{Bin}}(t_1^* - 1; n_{12} + n_{21}, 0.5) + 0.5f_{\text{Bin}}(t_1^*; n_{12} + n_{21}, 0.5) < \alpha$ , where  $F_{\text{Bin}}(x; n, p)$  and  $f_{\text{Bin}}(x; n, p)$  are the binomial C.D.F. and P.D.F, respectively, with sample size  $n$  and success probability  $p$  evaluated at  $x$ , then reject  $H_0$ .

- For two-sided tests, the test statistic

$$t_2^* = \min(n_{12}, n_{21})$$

If  $F_{\text{Bin}}(t_2^* - 1; n_{12} + n_{21} - 1, 0.5) + 0.5f_{\text{Bin}}(t_2^*; n_{12} + n_{21}, 0.5) < \alpha / 2$ , then reject  $H_0$ .

The mid- $p$ -value test addresses the over-conservative behavior of the exact conditional test. The simulation studies in [18] demonstrate that this test attains nominal coverage, and has good statistical power.

## Classification Loss

Classification losses indicate the accuracy of a classification model or set of predicted labels. Two classification losses are misclassification rate and cost.

`CompactClassificationECOC.compareHoldout` returns the classification losses (see `e1` and `e2`) under the alternative hypothesis (i.e., the unrestricted classification losses).  $n_{ijk}$  is the number of test-sample observations with true class  $k$  that the first

classification model assigns label  $i$  and the second classification model assigns label  $j$ ,

and the corresponding estimated proportion is  $\hat{\pi}_{ijk} = \frac{n_{ijk}}{n_{test}}$ . . The test-set sample size is

$\sum_{i,j,k} n_{ijk} = n_{test}$ . The indices are taken from 1 through  $K$ , the number of classes.

- Misclassification rate, or classification error, is a scalar in the interval  $[0,1]$  representing the proportion of misclassified observations. That is, the misclassification rate for the first classification model is

$$e_1 = \sum_{j=1}^K \sum_{k=1}^K \sum_{i \neq k} \hat{\pi}_{ijk}.$$

For the misclassification rate of the second classification model ( $e_2$ ), switch the indices  $i$  and  $j$  in the formula.

Classification accuracy decreases as the misclassification rate increases to 1.

- Misclassification cost is a nonnegative scalar and is a measure of classification quality relative to the values the specified cost matrix elements. Its interpretation depends on the specified costs of misclassification. Misclassification cost is the weighted average of the costs of misclassification (specified in a cost matrix,  $C$ ) in which the weights are the respective, estimated proportions of misclassified observations. The misclassification cost for the first classification model is

$$e_1 = \sum_{j=1}^K \sum_{k=1}^K \sum_{i \neq k} \hat{\pi}_{ijk} c_{ki},$$

where  $c_{kj}$  is the cost of classifying an observation into class  $j$  if its true class is  $k$ . For the misclassification cost of the second classification model ( $e_2$ ), switch the indices  $i$  and  $j$  in the formula.

In general, for a fixed cost matrix, classification accuracy decreases as misclassification cost increases.

## Tips

- One way to perform cost-insensitive feature selection is:
  - 1 Train the first classification model ( $C_1$ ) using the full predictor set.



- 2 Train the second classification model (C2) using the reduced predictor set.
- 3 Specify X1 as the full, test-set predictor data and X2 as the reduced test-set predictor data.
- 4 Enter `compareHoldout(C1,C2,X1,X2,'Alternative','less')`. If `compareHoldout` returns 1, then there is enough evidence to suggest that the classification model that uses fewer predictors performs better than the model that uses the full predictor set.

Alternatively, you can assess whether there is a significant difference between the accuracies of the two models. To perform this assessment, remove the 'Alternative', 'greater' specification in step 4. `compareHoldout` conducts a two-sided test, and  $h = 0$  indicates that there is not enough evidence to suggest a difference in the accuracy of the two models.

- Cost-sensitive tests perform numerical optimization, which requires additional computational resources. The likelihood ratio test conducts numerical optimization indirectly by finding the root of a Lagrange multiplier in an interval. For some data sets, if the root lies close to the boundaries of the interval, then the method can fail. Therefore, if you have an Optimization Toolbox license, consider conducting the cost-sensitive chi-square test instead. For more details, see `CostTest` and “Cost-Sensitive Testing” on page 32-1053.

## Alternatives

To directly compare the accuracy of two sets of class labels in predicting a set of true class labels, use `testcholdout`.

## References

- [1] Agresti, A. *Categorical Data Analysis*, 2nd Ed. John Wiley & Sons, Inc.: Hoboken, NJ, 2002.
- [2] Fagerlan, M.W., S Lydersen, P. Laake. “The McNemar Test for Binary Matched-Pairs Data: Mid-p and Asymptotic Are Better Than Exact Conditional.” *BMC Medical Research Methodology*. Vol. 13, 2013, pp. 1–8.
- [3] Lancaster, H.O. “Significance Tests in Discrete Distributions.” *JASA*, Vol. 56, Number 294, 1961, pp. 223–234.

- [4] McNemar, Q. “Note on the Sampling Error of the Difference Between Correlated Proportions or Percentages.” *Psychometrika*, Vol. 12, Number 2, 1947, pp. 153–157.
- [5] Mosteller, F. “Some Statistical Problems in Measuring the Subjective Response to Drugs.” *Biometrics*, Vol. 8, Number 3, 1952, pp. 220–226.

## See Also

`fitcecoc` | `predict` | `testcholdout` | `testckfold`

## Topics

“Hypothesis Tests”

**Introduced in R2015a**

# compareHoldout

**Class:** CompactClassificationEnsemble

Compare accuracies of two classification models using new data

`compareHoldout` statistically assesses the accuracies of two classification models. The function first compares their predicted labels against the true labels, and then it detects whether the difference between the misclassification rates is statistically significant.

You can assess whether the accuracies of the classification models are different, or whether one classification model performs better than another. `compareHoldout` can conduct several McNemar test on page 32-1055 variations, including the asymptotic test, the exact-conditional test, and the mid- $p$ -value test. For cost-sensitive assessment on page 32-1053, available tests include a chi-square test (requires an Optimization Toolbox license) and a likelihood ratio test.

## Syntax

```
h = compareHoldout(C1,C2,T1,T2,ResponseVarName)
```

```
h = compareHoldout(C1,C2,T1,T2,Y)
```

```
h = compareHoldout( ___,Name,Value)
```

```
h = compareHoldout(C1,C2,X1,X2,Y)
```

```
h = compareHoldout( ___,Name,Value)
```

```
[h,p,e1,e2] = compareHoldout( ___)
```

## Description

`h = compareHoldout(C1,C2,T1,T2,ResponseVarName)` returns the test decision from testing the null hypothesis that the trained classification models `C1` and `C2` have equal accuracy for predicting the true class labels in `ResponseVarName`. The alternative hypothesis is that the labels have unequal accuracy.

The first classification model `C1` uses predictor data in table `T1` and `C2` uses `T2`. The software conducts the mid- $p$ -value McNemar test to compare the accuracies.

$h = 1$  indicates to reject the null hypothesis at the 5% significance level.  $h = 0$  indicates to not reject the null hypothesis at 5% level.

Examples of tests you can conduct include:

- Compare the accuracies of a simple classification model and a model that is more complex by passing the same set of predictor data (i.e.,  $T1 = T2$ ).
- Compare the accuracies of two perhaps different models using two perhaps different sets of predictors.
- Perform various types of feature selection. For example, compare the accuracies of a model trained using a set of predictors and one trained on a subset or different set of those predictors. You can arbitrarily choose the set of predictors, or use a feature selection technique like PCA or sequential feature selection (see `pca` and `sequentialfs`).

`h = compareHoldout(C1, C2, T1, T2, Y)` returns the test decision from testing the null hypothesis that the trained classification models `C1` and `C2` have equal accuracy for predicting the true class labels `Y`. The alternative hypothesis is that the labels have unequal accuracy.

`h = compareHoldout( ____, Name, Value)` returns the result of the hypothesis test with additional options specified by one or more `Name, Value` pair arguments, using any of the previous syntaxes. For example, specify the type of alternative hypothesis, specify the type of test, or supply a cost matrix.

`h = compareHoldout(C1, C2, X1, X2, Y)` returns the test decision from testing the null hypothesis that the trained classification models `C1` and `C2` have equal accuracy for predicting the true class labels `Y`. The alternative hypothesis is that the labels have unequal accuracy.

The first classification model `C1` uses predictor data `X1` and `C2` uses `X2`. The software conducts the mid- $p$ -value McNemar test to compare the accuracies.

`h = compareHoldout( ____, Name, Value)` returns the result of the hypothesis test with additional options specified by one or more `Name, Value` pair arguments. For example, specify the type of alternative hypothesis, specify the type of test, or supply a cost matrix.

`[h,p,e1,e2] = compareHoldout(____)` returns the  $p$ -value for the hypothesis test ( $p$ ) and the respective classification losses on page 32-1004 of each set of predicted class labels ( $e1$  and  $e2$ ).

## Input Arguments

### **c1 — Trained ensemble of classification models**

`ClassificationBaggedEnsemble` model object | `ClassificationEnsemble` model object | `CompactClassificationEnsemble` model object

Trained ensemble of classification models, specified as a `ClassificationBaggedEnsemble`, `ClassificationEnsemble`, or `CompactClassificationEnsemble` model object. That is, `C1` is a trained classification model returned by `fitcensemble` or `compact`.

### **c2 — Trained classification model**

trained classification model object | trained, compact classification model object

Trained classification model, specified as any trained or compact classification model object described in this table.

Trained Model Type	Model Object	Returned By
Classification tree	<code>ClassificationTree</code>	<code>fitctree</code>
Discriminant analysis	<code>ClassificationDiscriminant</code>	<code>fitcdiscr</code>
Ensemble of bagged classification models	<code>ClassificationBaggedEnsemble</code>	<code>fitcensemble</code>
Ensemble of classification models	<code>ClassificationEnsemble</code>	<code>fitcensemble</code>
Multiclass, error-correcting output codes (ECOC)	<code>ClassificationECOC</code>	<code>fitcecoc</code>
$k$ NN	<code>ClassificationKNN</code>	<code>fitcknn</code>
Naive Bayes	<code>ClassificationNaiveBayes</code>	<code>fitcnb</code>
Support vector machine (SVM)	<code>ClassificationSVM</code>	<code>fitcsvm</code>

Trained Model Type	Model Object	Returned By
Compact discriminant analysis	CompactClassificationDiscriminant	compact
Compact ECOC	CompactClassificationECOC	compact
Compact ensemble of classification models	CompactClassificationEnsemble	compact
Compact naive Bayes	CompactClassificationNaiveBayes	compact
Compact SVM	CompactClassificationSVM	compact
Compact classification tree	CompactClassificationTree	compact

### **T1 — Test-set predictor table for first classification model**

table

Test-set predictor table for the first classification model, C1, specified as a table. Each row of T1 corresponds to one observation, and each column corresponds to one predictor variable. Optionally, T1 can contain an additional column for the response variable. T1 must contain all of the predictors used to train C1. Multi-column variables and cell arrays other than cell arrays of character vectors are not allowed.

T1 and T2 must have the same number of rows, and must have the same response values. If T1 and T2 contain the response variable used to train C1 and C2, then, you do not need to specify ResponseVarName or Y.

Data Types: table

### **T2 — Test-set predictor table for second classification model**

table

Test-set predictor data for the second classification model, C2, specified as a table. Each row of T2 corresponds to one observation, and each column corresponds to one predictor variable. Optionally, T2 can contain an additional column for the response variable. T2 must contain all of the predictors used to train C2. Multi-column variables and cell arrays other than cell arrays of character vectors are not allowed.

T1 and T2 must have the same number of rows and the same response values. If T1 and T2 contain the response variable used to train C1 and C2, then, you do not need to specify ResponseVarName or Y.

Data Types: `table`

### **x1 — Test-set predictor data for first classification model**

numeric matrix

Test-set predictor data for the first classification model, C1, specified as a numeric matrix.

Each row of X1 corresponds to one observation (also known as an instance or example), and each column corresponds to one variable (also known as a predictor or feature). The variables used to train C1 must compose X1.

The number of rows in X1 and X2 must equal the length of Y.

Data Types: `double` | `single`

### **x2 — Test-set predictor data for second classification model**

numeric matrix

Test-set predictor data for the second classification model, C2, specified as a numeric matrix.

Each row of X2 corresponds to one observation (also known as an instance or example), and each column corresponds to one variable (also known as a predictor or feature). The variables used to train C2 must compose X2.

The number of rows in X2 and X1 must equal the length of Y.

Data Types: `double` | `single`

### **ResponseVarName — Response variable name**

name of a variable in T1 and T2

Response variable name, specified as the name of a variable in T1 and T2. If T1 and T2 contain the response variable used to train C1 and C2, then you do not need to specify ResponseVarName.

If you specify ResponseVarName, then you must do so as a character vector. For example, if the response variable is stored as T1.Response, then specify it as 'Response'. Otherwise, the software treats all columns of T1 and T2, including Response, as predictors.

The response variable must be a categorical or character array, logical or numeric vector, or cell array of character vectors. If the response variable is a character array, then each element must correspond to one row of the array.

Data Types: `char`

### **Y — True class labels**

categorical array | character array | logical vector | vector of numeric values | cell array of character vectors | character vector

True class labels, specified as a categorical or character array, a logical or numeric vector, a cell array of character vectors, or a character vector.

If you specify any array except a character vector, then

`CompactClassificationEnsemble.compareHoldout` treats all variables or columns in `X1` and `X2` as predictor variables.

If you specify a character vector, then `X1` and `X2` must be tables, their response variables must have the same name and values, and `Y` must be the common name. For example, if `X1.Labels` and `X2.Labels` are the response variables, then `Y` must be `'Labels'` and `X1.Labels` and `X2.Labels` must be equivalent.

If `Y` is a character array, then each element must correspond to one row of the array.

The number of rows in `X1` and `X2` must equal the length of `Y`.

Data Types: `categorical` | `char` | `logical` | `single` | `double` | `cell`

## **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'Alternative', 'greater', 'Test', 'asymptotic', 'Cost', [0 2; 1 0]` specifies to test whether the first set of first predicted class labels is more accurate than the second set, to conduct the asymptotic McNemar test, and to penalize misclassifying observations with the true label `ClassNames{1}` two times the penalty of misclassifying observations with the true label `ClassNames{2}`.



**Alpha — Hypothesis test significance level**

0.05 (default) | scalar value in the interval (0,1)

Hypothesis test significance level, specified as the comma-separated pair consisting of 'Alpha' and a scalar value in the interval (0,1).

Example: 'Alpha', 0.1

Data Types: single | double

**Alternative — Alternative hypothesis to assess**

'unequal' (default) | 'greater' | 'less'

Alternative hypothesis to assess, specified as the comma-separated pair consisting of 'Alternative' and one of these values listed in the table.

Value	Alternative hypothesis
'unequal' (default)	For predicting $Y$ , the set of predictions resulting from C1 applied to X1 and C2 applied to X2 have unequal accuracies.
'greater'	For predicting $Y$ , the set of predictions resulting from C1 applied to X1 is more accurate than C2 applied to X2.
'less'	For predicting $Y$ , the set of predictions resulting from C1 applied to X1 is less accurate than C2 applied to X2.

Example: 'Alternative', 'greater'

Data Types: char

**ClassNames — Class names**

categorical vector | cell array of character vectors | character array | logical vector | numeric vector

Class names, specified as the comma-separated pair consisting of 'ClassNames' and a categorical or character array, logical or numeric vector, or cell array of character vectors. You must set ClassNames using the data type of  $Y$ .

If ClassNames is a character array, then each element must correspond to one *row* of the array.

Use ClassNames to:

- Specify the order of any input argument dimension that corresponds to class order. For example, use `ClassNames` to specify the order of the dimensions of `Cost`.
- Select a subset of classes for testing. For example, suppose that the set of all distinct class names in  $Y$  is `{'a', 'b', 'c'}`. To train and test models using observations from classes 'a' and 'c' only, specify `'ClassNames', {'a', 'c'}`.

The default is the set of all distinct class names in  $Y$ .

Example: `'ClassNames', {'b', 'g'}`

Data Types: `categorical | char | logical | single | double | cell`

### **Cost — Misclassification cost**

`square matrix | structure array`

Misclassification cost, specified as the comma-separated pair consisting of `'Cost'` and a square matrix or structure array. If you specify:

- If you specify the square matrix `Cost`, then `Cost(i, j)` is the cost of classifying a point into class  $j$  if its true class is  $i$ . That is, the rows correspond to the true class and the columns correspond to the predicted class. To specify the class order for the corresponding rows and columns of `Cost`, additionally specify the `ClassNames` name-value pair argument.
- If you specify the structure `S`, then `S` must have two fields:
  - `S.ClassNames`, which contains the class names as a variable of the same data type as  $Y$ . You can use this field to specify the order of the classes.
  - `S.ClassificationCosts`, which contains the cost matrix, with rows and columns ordered as in `S.ClassNames`

If you specify `Cost`, then `CompactClassificationEnsemble.compareHoldout` cannot conduct one-sided, exact, or mid- $p$  tests. You must also specify `'Alternative', 'unequal', 'Test', 'asymptotic'`. For cost-sensitive testing options, see the `CostTest` name-value pair argument.

It is a best practice to supply the same cost matrix used to train the classification models.

The default is `Cost(i, j) = 1 if i ~= j`, and `Cost(i, j) = 0 if i = j`.

Example: `'Cost', [0 1 2 ; 1 0 2; 2 2 0]`

Data Types: `double | single | struct`

**CostTest — Cost-sensitive test type**

'likelihood' (default) | 'chisquare'

Cost-sensitive test type, specified as the comma-separated pair consisting of 'CostTest' and 'chisquare' or 'likelihood'. Unless you specify a cost matrix using the Cost name-value pair argument, CompactClassificationEnsemble.compareHoldout ignores CostTest.

This table summarizes the available options for cost-sensitive testing.

Value	Asymptotic test type	Requirements
'chisquare'	Chi-square test	Optimization Toolbox license to implement quadprog
'likelihood'	Likelihood ratio test	None

For more details, see “Cost-Sensitive Testing” on page 32-6515.

Example: 'CostTest', 'chisquare'

Data Types: char

**Test — Test to conduct**

'asymptotic' | 'exact' | 'midp'

Test to conduct, specified as the comma-separated pair consisting of 'Test' and 'asymptotic', 'exact', and 'midp'. This table summarizes the available options for cost-insensitive testing.

Value	Description
'asymptotic'	Asymptotic McNemar test
'exact'	Exact-conditional McNemar test
'midp' (default)	Mid- <i>p</i> -value McNemar test

For more details, see “McNemar Tests” on page 32-6517.

For cost-sensitive testing, Test must be 'asymptotic'. When you specify the Cost name-value pair argument, and choose a cost-sensitive test using the CostTest name-value pair argument, 'asymptotic' is the default.

Example: 'Test', 'asymptotic'

Data Types: char

---

**Note** NaNs, <undefined> values, and empty character vectors ( ' ' ) indicate missing values. `CompactClassificationEnsemble.compareHoldout`:

- Removes missing values in  $Y$  and the corresponding rows of  $X1$  and  $X2$
  - Predicts classes whether  $X1$  and  $X2$  have missing observations.
- 

## Output Arguments

### **h** — Hypothesis test result

1 | 0

Hypothesis test result, returned as a logical value.

$h = 1$  indicates the rejection of the null hypothesis at the Alpha significance level.

$h = 0$  indicates failure to reject the null hypothesis at the Alpha significance level.

### **p** — *p*-value

scalar in the interval [0,1]

*p*-value of the test, returned as a scalar in the interval [0,1]. *p* is the probability that a random test statistic is at least as extreme as the observed test statistic, given that the null hypothesis is true.

`CompactClassificationEnsemble.compareHoldout` estimates *p* using the distribution of the test statistic, which varies with the type of test. For details on test statistics derived from the available variants of the McNemar test, see “McNemar Tests” on page 32-6517. For details on test statistics derived from cost-sensitive tests, see “Cost-Sensitive Testing” on page 32-6515.

### **e1** — Classification loss

scalar

Classification loss on page 32-1058, returned as a scalar. *e1* summarizes the accuracy of the first set of class labels predicting the true class labels ( $Y$ ).

`CompactClassificationEnsemble.compareHoldout` applies the first test-set predictor data ( $X_1$ ) to the first classification model (C1) to estimate the first set of class labels. Then, the function compares the estimated labels to  $Y$  to obtain the classification loss.

For cost-insensitive testing,  $e_1$  is the misclassification rate. That is,  $e_1$  is the proportion of misclassified observations, which is a scalar in the interval  $[0,1]$ .

For cost-sensitive testing,  $e_1$  is the misclassification cost. That is,  $e_1$  is the weighted average of the misclassification costs, in which the weights are the respective estimated proportions of misclassified observations.

### **$e_2$ — Classification loss**

scalar

Classification loss on page 32-1058, returned as a scalar.  $e_2$  summarizes the accuracy of the second set of class labels predicting the true class labels ( $Y$ ).

`CompactClassificationEnsemble.compareHoldout` applies the second test-set predictor data ( $X_2$ ) to the second classification model (C2) to estimate the second set of class labels. Then the function compares the estimated labels to  $Y$  to obtain the classification loss.

For cost-insensitive testing,  $e_2$  is the misclassification rate. That is,  $e_2$  is the proportion of misclassified observations, which is a scalar in the interval  $[0,1]$ .

For cost-sensitive testing,  $e_2$  is the misclassification cost. That is,  $e_2$  is the weighted average of the misclassification costs, in which the weights are the respective estimated proportions of misclassified observations.

## **Examples**

### **Compare Accuracies of Two Different Classification Models**

Train two classification models using different algorithms. Conduct a statistical test comparing the misclassification rates of the two models on a held-out set.

Load the `ionosphere` data set.

```
load ionosphere;
```

Create a partition that evenly splits the data into training and testing sets.

```
rng(1); % For reproducibility
CVP = cvpartition(Y, 'holdout', 0.5);
idxTrain = training(CVP); % Training-set indices
idxTest = test(CVP); % Test-set indices
```

CVP is a cross-validation partition object that specifies the training and test sets.

Train an SVM model and an ensemble of 100 bagged classification trees. For the SVM model, specify to use the radial basis function kernel and a heuristic procedure to determine the kernel scale.

```
C1 = fitcsvm(X(idxTrain,:), Y(idxTrain), 'Standardize', true, ...
    'KernelFunction', 'RBF', 'KernelScale', 'auto');
C2 = fitensemble(X(idxTrain,:), Y(idxTrain), 'Bag', 100, 'Tree', ...
    'Type', 'classification');
```

C1 is a trained `ClassificationSVM` model. C2 is a trained `ClassificationBaggedEnsemble` model.

Test whether the two models have equal predictive accuracies. Use the same test-set predictor data for each model.

```
h = compareHoldout(C1, C2, X(idxTest,:), X(idxTest,:), Y(idxTest))
```

```
h =
    logical
     0
```

`h = 0` indicates to not reject the null hypothesis that the two models have equal predictive accuracies.

### Assess Whether One Classification Model Classifies Better Than Another

Train two classification models using the same algorithm, but adjust a hyperparameter to make the algorithm more complex. Conduct a statistical test to assess whether the simpler model has better accuracy in held-out data than the more complex model.

Load the ionosphere data set.

```
load ionosphere;
```

Create a partition that evenly splits the data into training and testing sets.

```
rng(1); % For reproducibility
CVP = cvpartition(Y, 'holdout', 0.5);
idxTrain = training(CVP); % Training-set indices
idxTest = test(CVP); % Test-set indices
```

CVP is a cross-validation partition object that specifies the training and test sets.

Train two SVM models: one that uses a linear kernel (the default for binary classification) and one that uses the radial basis function kernel. Use the default kernel scale of 1.

```
C1 = fitcsvm(X(idxTrain,:), Y(idxTrain), 'Standardize', true);
C2 = fitcsvm(X(idxTrain,:), Y(idxTrain), 'Standardize', true, ...
    'KernelFunction', 'RBF');
```

C1 and C2 are trained ClassificationSVM models.

Test the null hypothesis that the simpler model (C1) is at most as accurate as the more complex model (C2). Because the test-set size is large, conduct the asymptotic McNemar test, and compare the results with the mid-  $p$ -value test (the cost-insensitive testing default). Request to return  $p$ -values and misclassification rates.

```
Asymp = zeros(4,1); % Preallocation
MidP = zeros(4,1);

[Asymp(1), Asymp(2), Asymp(3), Asymp(4)] = compareHoldout(C1, C2, ...
    X(idxTest,:), X(idxTest,:), Y(idxTest), 'Alternative', 'greater', ...
    'Test', 'asymptotic');
[MidP(1), MidP(2), MidP(3), MidP(4)] = compareHoldout(C1, C2, ...
    X(idxTest,:), X(idxTest,:), Y(idxTest), 'Alternative', 'greater');
table(Asymp, MidP, 'RowNames', {'h' 'p' 'e1' 'e2'})
```

```
ans =
```

```
4x2 table
```

```
Asymp MidP
```

	-----	-----
h	1	1
p	7.2801e-09	2.7649e-10
e1	0.13714	0.13714
e2	0.33143	0.33143

The  $p$ -value is close to zero for both tests, which indicates strong evidence to reject the null hypothesis that the simpler model is less accurate than the more complex model. No matter what test you specify, `compareHoldout` returns the same type of misclassification measure for both models.

### Conduct a Cost-Sensitive Comparison of Two Classification Models

For data sets with imbalanced class representations, or for data sets with imbalanced false-positive and false-negative costs, you can statistically compare the predictive performances of two classification models by including a cost matrix in the analysis.

Load the `arrhythmia` data set. Determine the class representations in the data.

```
load arrhythmia;  
Y = categorical(Y);  
tabulate(Y);
```

Value	Count	Percent
1	245	54.20%
2	44	9.73%
3	15	3.32%
4	15	3.32%
5	13	2.88%
6	25	5.53%
7	3	0.66%
8	2	0.44%
9	9	1.99%
10	50	11.06%
14	4	0.88%
15	5	1.11%
16	22	4.87%

There are 16 classes, however some are not represented in the data set. Most observations are classified as not having arrhythmia (class 1). To summarize, the data set is highly discrete with imbalanced classes.



Combine all observations with arrhythmia (classes 2 through 15) into one class. Remove those observations with unknown arrhythmia status from the data set.

```
Y = Y(Y ~= '16');
Y(Y ~= '1') = '2';
X = X(Y ~= '16', :);
```

Create a partition that evenly splits the data into training and testing sets.

```
rng(1); % For reproducibility
CVP = cvpartition(Y, 'holdout', 0.5);
idxTrain = training(CVP); % Training-set indices
idxTest = test(CVP); % Test-set indices
```

CVP is a cross-validation partition object that specifies the training and test sets.

Create a cost matrix such that misclassifying an arrhythmic patient into the no arrhythmia class is five times worse than misclassifying a patient without arrhythmia into the arrhythmia class. Classifying correctly incurs no cost. The rows indicate the true class and the columns indicate the predicted class. When conducting a cost-sensitive analysis, it is a good practice to specify the order of the classes.

```
cost = [0 1; 5 0];
ClassNames = categorical([2 1]);
```

Train two boosting ensembles of 50 classification trees, one that uses AdaBoostM1 and the other that uses LogitBoost. Because there are missing values, specify to use surrogate splits. Train the models using the cost matrix.

```
t = templateTree('Surrogate', 'on');
numTrees = 50;
C1 = fitensemble(X(idxTrain, :), Y(idxTrain), 'AdaBoostM1', numTrees, t, ...
    'Cost', cost);
C2 = fitensemble(X(idxTrain, :), Y(idxTrain), 'LogitBoost', numTrees, t, ...
    'Cost', cost);
```

C1 and C2 are trained ClassificationEnsemble models.

Test whether the AdaBoostM1 ensemble (C1) and the LogitBoost ensemble (C2) have equal predictive accuracy. Supply the cost matrix. Conduct the asymptotic, likelihood ratio, cost-sensitive test (the default when you pass in a cost matrix). Request to return  $p$ -values and misclassification costs.

```
[h,p,e1,e2] = compareHoldout(C1,C2,X(idxTest,:),X(idxTest,:),Y(idxTest),...  
    'Cost',cost)
```

```
h =
```

```
    logical
```

```
    0
```

```
p =
```

```
    0.0743
```

```
e1 =
```

```
    1.3581
```

```
e2 =
```

```
    1.6186
```

`h = 0` indicates to not reject the null hypothesis that the two models have equal predictive accuracies.

### Select Features Using Statistical Accuracy Comparison

Reduce classification model complexity by selecting a subset of predictor variables (features) from a larger set. Then, statistically compare the out-of-sample accuracy between the two models.

Load the `ionosphere` data set.

```
load ionosphere;
```

Create a partition that evenly splits the data into training and testing sets.

```
rng(1); % For reproducibility  
CVP = cvpartition(Y,'holdout',0.5);
```

```
idxTrain = training(CVP);           % Training-set indices
idxTest = test(CVP);               % Test-set indices
```

CVP is a cross-validation partition object that specifies the training and test sets.

Train an ensemble of 100 boosted classification trees using AdaBoostM1 and the entire set of predictors. Inspect the importance measure for each predictor.

```
nTrees = 100;
C2 = fitensemble(X(idxTrain,:),Y(idxTrain),'AdaBoostM1',nTrees,'Tree');
predImp = predictorImportance(C2);
```

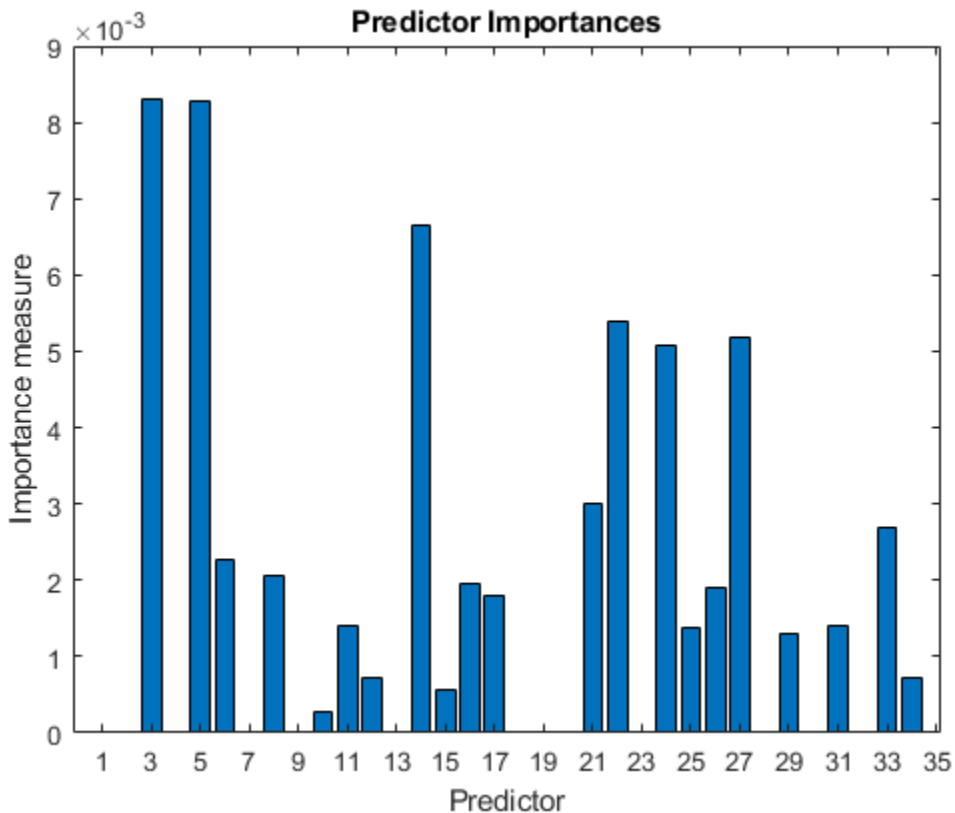
```
figure;
bar(predImp);
h = gca;
h.XTick = 1:2:h.XLim(2)
title('Predictor Importances');
xlabel('Predictor');
ylabel('Importance measure');
```

h =

Axes with properties:

```
        XLim: [-0.2000 35.2000]
        YLim: [0 0.0090]
        XScale: 'linear'
        YScale: 'linear'
GridLineStyle: '-'
        Position: [0.1300 0.1100 0.7750 0.8150]
        Units: 'normalized'
```

Use GET to show all properties



Identify the top five predictors in terms of their importance.

```
[~,idxSort] = sort(predImp, 'descend');
idx5 = idxSort(1:5);
```

Train another ensemble of 100 boosted classification trees using AdaBoostM1 and the five predictors with the best importance.

```
C1 = fitensemble(X(idxTrain,idx5),Y(idxTrain),'AdaBoostM1',nTrees,...
    'Tree');
```

Test whether the two models have equal predictive accuracies. Specify the reduced test-set predictor data for C1 and the full test-set predictor data for C2.

```
[h,p,e1,e2] = compareHoldout(C1,C2,X(idxTest,idx5),X(idxTest,:),Y(idxTest))
```

```
h =  
  logical  
  0  
  
p =  
  0.7744  
  
e1 =  
  0.0914  
  
e2 =  
  0.0857
```

$h = 0$  indicates to not reject the null hypothesis that the two models have equal predictive accuracies. This result favors the simpler ensemble, C1.

## Definitions

### Cost-Sensitive Testing

Conduct cost-sensitive testing when the cost of misclassification is imbalanced. When conducting a cost-sensitive analysis, you can account for the cost imbalance in training the classification models, and then in statistically comparing them.

If the cost of misclassification is imbalanced, then the misclassification rate tends to be a poorly performing classification loss on page 32-6520. Use misclassification cost instead to compare classification models.

Misclassification costs are often unbalanced in applications. For example, consider classifying subjects based on a set of predictors into two categories: healthy and sick. Misclassifying a sick subject as healthy poses a danger to the subject's life. However,

misclassifying a healthy subject as sick can cause some inconvenience, but does not pose any danger. In this situation, you assign misclassification costs such that misclassifying a sick subject as healthy is more costly than misclassifying a healthy subject as sick.

The definitions that follow summarize the cost-sensitive tests. In the definitions:

- $n_{ijk}$  and  $\hat{\pi}_{ijk}$  are the number and estimated proportion of test-sample observations with true class  $k$  that the first classification model assigns label  $i$ . The second classification model assigns label  $j$ . The unknown, true value of  $\hat{\pi}_{ijk}$  is  $\pi_{ijk}$ . The test-set sample size is  $\sum_{i,j,k} n_{ijk} = n_{test}$ .  $\sum_{i,j,k} \pi_{ijk} = \sum_{i,j,k} \pi_{ijk} = 1$ .
- $c_{ij}$  is the relative cost of assigning label  $j$  to an observation with true class  $i$ .  $c_{ii} = 0$ ,  $c_{ij} \geq 0$ , and, for at least one  $(i,j)$  pair,  $c_{ij} > 0$ .

- All subscripts take on integer values from 1 through  $K$ , which is the number of classes.
- The expected difference in the misclassification costs of the two classification models is

$$\delta = \sum_{i=1}^K \sum_{j=1}^K \sum_{k=1}^K (c_{ki} - c_{kj}) \pi_{ijk}.$$

- The hypothesis test is

$$H_0 : \delta = 0$$

$$H_1 : \delta \neq 0$$

The available cost-sensitive tests are appropriate for two-tailed testing.

Available, asymptotic tests that address imbalanced costs are a chi-square test and a likelihood ratio test.

- Chi-square test — The chi-square test statistic is based on the Pearson and Neyman chi-square test statistics, but with a Laplace correction factor to account for any  $n_{ijk} = 0$ . The test statistic is

$$t_{\chi^2}^* = \sum_{i \neq j} \sum_k \frac{\left( n_{ijk} + 1 - (n_{test} + K^3) \hat{\pi}_{ijk}^{(1)} \right)^2}{n_{ijk} + 1}.$$

If  $1 - F_{\chi^2}(t_{\chi^2}^*; 1) < \alpha$ , then reject  $H_0$ .

•

$\hat{\pi}_{ijk}^{(1)}$  are estimated by minimizing  $t_{\chi^2}^*$  under the constraint that  $\delta = 0$ .

•

$F_{\chi^2}(x; 1)$  is the  $\chi^2$  C.D.F. with one degree of freedom evaluated at  $x$ .

- Likelihood ratio test — The likelihood ratio test is based on  $N_{ijk}$ , which are binomial random variables having sample size  $n_{test}$  and success probability  $\pi_{ijk}$ . They represent the random number of observations with true class  $k$  that the first classification model assigns label  $i$ . The second classification model assigns label  $j$ . Jointly, their distribution is multinomial.

The test statistic is

$$t_{LRT}^* = 2 \log \left[ \frac{P\left(\bigcap_{i,j,k} N_{ijk} = n_{ijk}; n_{test}, \pi_{ijk} = \pi_{ijk}^{(2)}\right)}{P\left(\bigcap_{i,j,k} N_{ijk} = n_{ijk}; n_{test}, \pi_{ijk} = \pi_{ijk}^{(3)}\right)} \right].$$

If  $1 - F_{\chi^2}(t_{LRT}^*; 1) < \alpha$ , then reject  $H_0$ .

•

$\hat{\pi}_{ijk}^{(2)} = \frac{n_{ijk}}{n_{test}}$  is the unrestricted MLE of  $\pi_{ijk}$ .

•

$\hat{\pi}_{ijk}^{(3)} = \frac{n_{ijk}}{n_{test} + \lambda(c_{ki} - c_{kj})}$  is the MLE under the null hypothesis that  $\delta = 0$ .  $\lambda$  is the solution to

$$\sum_{i,j,k} \frac{n_{ijk}(c_{ki} - c_{kj})}{n_{test} + \lambda(c_{ki} - c_{kj})} = 0.$$

•

$F_{\chi^2}(x; 1)$  is the  $\chi^2$  C.D.F. with one degree of freedom evaluated at  $x$ .

## McNemar Tests

McNemar Tests are hypothesis tests that compare two population proportions while addressing the issues resulting from two dependent, matched-pair samples.

One way to compare the predictive accuracies of two classification models is:

- 1 Partition the data into training and test sets.
- 2 Train both classification models using the training set.
- 3 Predict class labels using the test set.
- 4 Summarize the results in a two-by-two table resembling this figure.

		Model 2		
		Correct	Incorrect	
Model 1	Correct	$n_{11}$	$n_{12}$	$n_{1\bullet}$
	Incorrect	$n_{21}$	$n_{22}$	$n_{2\bullet}$
		$n_{\bullet 1}$	$n_{\bullet 2}$	$n_{test}$

$n_{ii}$  are the number of concordant pairs, that is, the number of observations that both models classify the same way (correctly or incorrectly).  $n_{ij}$ ,  $i \neq j$ , are the number of discordant pairs, that is, the number of observations that models classify differently (correctly or incorrectly).

The misclassification rates for Models 1 and 2 are

$$\hat{\pi}_{2\bullet} = n_{2\bullet} / n$$

and  $\hat{\pi}_{\bullet 2} = n_{\bullet 2} / n$ , respectively. A two-sided test for comparing the accuracy of the two models is

$$H_0 : \pi_{\bullet 2} = \pi_{2\bullet}$$

$$H_1 : \pi_{\bullet 2} \neq \pi_{2\bullet}$$

The null hypothesis suggests that the population exhibits marginal homogeneity, which reduces the null hypothesis to  $H_0 : \pi_{12} = \pi_{21}$ . Also, under the null hypothesis,  $N_{12} \sim \text{Binomial}(n_{12} + n_{21}, 0.5)$  [1].



These facts are the basis for these, available McNemar test variants: the asymptotic, exact-conditional and mid-p-value McNemar tests. The definitions that follow summarize the available variants.

- Asymptotic — The asymptotic McNemar test statistics and rejection regions (for significance-level  $\alpha$ ) are:

- For one-sided tests, the test statistic

$$t_{a1}^* = \frac{n_{12} - n_{21}}{\sqrt{n_{12} + n_{21}}}.$$

If  $1 - \Phi\left(|t_1^*|\right) < \alpha$ , where  $\Phi$  is the standard Gaussian C.D.F., then reject  $H_0$ .

- For two-sided tests, the test statistic

$$t_{a2}^* = \frac{(n_{12} - n_{21})^2}{n_{12} + n_{21}}.$$

If  $1 - F_{\chi^2}\left(t_2^*; m\right) < \alpha$ , where  $F_{\chi^2}(x; m)$  is the  $\chi_m^2$  C.D.F. evaluated at  $x$ , then reject  $H_0$ .

This variant requires large-sample theory, specifically, the Gaussian approximation to the binomial distribution. Therefore:

- The total number of discordant pairs,  $n_d = n_{12} + n_{21}$  must be greater than 10 ([1], Ch. 10.1.4).
- In general, asymptotic tests do not guarantee nominal coverage. The observed probability of falsely rejecting the null hypothesis can exceed  $\alpha$ . Simulation studies in [18] suggest this, but the asymptotic McNemar test performs well in terms of statistical power.
- Exact Conditional — The exact-conditional McNemar test statistics and rejection regions (for significance-level  $\alpha$ ) are ([36], [38]):

- For one-sided tests, the test statistic

$$t_1^* = n_{12}$$

If  $F_{\text{Bin}}\left(t_1^*; n_d, 0.5\right) < \alpha$ , where  $F_{\text{Bin}}(x; n, p)$  is the binomial C.D.F. with sample size  $n$  and success probability  $p$  evaluated at  $x$ , then reject  $H_0$ .

- For two-sided tests, the test statistic

$$t_2^* = \min(n_{12}, n_{21})$$

If  $F_{\text{Bin}}(t_2^*; n_d, 0.5) < \alpha / 2$ , then reject  $H_0$ .

The exact conditional test always attains nominal coverage. Simulation studies in [18] suggest that the test is conservative, and then show that the test lacks statistical power compared to other variants. For small or highly discrete test samples, consider using the mid- $p$ -value test ([1], Ch. 3.6.3). For details, see `Test` and “McNemar Tests” on page 32-6517.

- Mid- $p$ -value test — The mid- $p$ -value McNemar test statistics and rejection regions (for significance-level  $\alpha$ ) are ([32]):

- For one-sided tests, the test statistic

$$t_1^* = n_{12}$$

If  $F_{\text{Bin}}(t_1^* - 1; n_{12} + n_{21}, 0.5) + 0.5f_{\text{Bin}}(t_1^*; n_{12} + n_{21}, 0.5) < \alpha$ , where  $F_{\text{Bin}}(x; n, p)$  and  $f_{\text{Bin}}(x; n, p)$  are the binomial C.D.F. and P.D.F, respectively, with sample size  $n$  and success probability  $p$  evaluated at  $x$ , then reject  $H_0$ .

- For two-sided tests, the test statistic

$$t_2^* = \min(n_{12}, n_{21})$$

If  $F_{\text{Bin}}(t_2^* - 1; n_{12} + n_{21} - 1, 0.5) + 0.5f_{\text{Bin}}(t_2^*; n_{12} + n_{21}, 0.5) < \alpha / 2$ , then reject  $H_0$ .

The mid- $p$ -value test addresses the over-conservative behavior of the exact conditional test. The simulation studies in [18] demonstrate that this test attains nominal coverage, and has good statistical power.

## Classification Loss

Classification losses indicate the accuracy of a classification model or set of predicted labels. Two classification losses are misclassification rate and cost.

`CompactClassificationEnsemble.compareHoldout` returns the classification losses (see `e1` and `e2`) under the alternative hypothesis (i.e., the unrestricted classification losses).  $n_{ijk}$  is the number of test-sample observations with true class  $k$  that the first

classification model assigns label  $i$  and the second classification model assigns label  $j$ ,

and the corresponding estimated proportion is  $\hat{\pi}_{ijk} = \frac{n_{ijk}}{n_{test}}$ . The test-set sample size is

$\sum_{i,j,k} n_{ijk} = n_{test}$ . The indices are taken from 1 through  $K$ , the number of classes.

- Misclassification rate, or classification error, is a scalar in the interval  $[0,1]$  representing the proportion of misclassified observations. That is, the misclassification rate for the first classification model is

$$e_1 = \sum_{j=1}^K \sum_{k=1}^K \sum_{i \neq k} \hat{\pi}_{ijk}.$$

For the misclassification rate of the second classification model ( $e_2$ ), switch the indices  $i$  and  $j$  in the formula.

Classification accuracy decreases as the misclassification rate increases to 1.

- Misclassification cost is a nonnegative scalar and is a measure of classification quality relative to the values the specified cost matrix elements. Its interpretation depends on the specified costs of misclassification. Misclassification cost is the weighted average of the costs of misclassification (specified in a cost matrix,  $C$ ) in which the weights are the respective, estimated proportions of misclassified observations. The misclassification cost for the first classification model is

$$e_1 = \sum_{j=1}^K \sum_{k=1}^K \sum_{i \neq k} \hat{\pi}_{ijk} c_{ki},$$

where  $c_{kj}$  is the cost of classifying an observation into class  $j$  if its true class is  $k$ . For the misclassification cost of the second classification model ( $e_2$ ), switch the indices  $i$  and  $j$  in the formula.

In general, for a fixed cost matrix, classification accuracy decreases as misclassification cost increases.

## Tips

- One way to perform cost-insensitive feature selection is:
  - 1 Train the first classification model ( $C_1$ ) using the full predictor set.

- 2 Train the second classification model (C2) using the reduced predictor set.
- 3 Specify X1 as the full, test-set predictor data and X2 as the reduced test-set predictor data.
- 4 Enter `compareHoldout(C1,C2,X1,X2,'Alternative','less')`. If `compareHoldout` returns 1, then there is enough evidence to suggest that the classification model that uses fewer predictors performs better than the model that uses the full predictor set.

Alternatively, you can assess whether there is a significant difference between the accuracies of the two models. To perform this assessment, remove the 'Alternative', 'greater' specification in step 4. `compareHoldout` conducts a two-sided test, and `h = 0` indicates that there is not enough evidence to suggest a difference in the accuracy of the two models.

- Cost-sensitive tests perform numerical optimization, which requires additional computational resources. The likelihood ratio test conducts numerical optimization indirectly by finding the root of a Lagrange multiplier in an interval. For some data sets, if the root lies close to the boundaries of the interval, then the method can fail. Therefore, if you have an Optimization Toolbox license, consider conducting the cost-sensitive chi-square test instead. For more details, see `CostTest` and “Cost-Sensitive Testing” on page 32-1053.

## Alternatives

To directly compare the accuracy of two sets of class labels in predicting a set of true class labels, use `testcholdout`.

## References

- [1] Agresti, A. *Categorical Data Analysis*, 2nd Ed. John Wiley & Sons, Inc.: Hoboken, NJ, 2002.
- [2] Fagerlan, M.W., S Lydersen, P. Laake. “The McNemar Test for Binary Matched-Pairs Data: Mid-p and Asymptotic Are Better Than Exact Conditional.” *BMC Medical Research Methodology*. Vol. 13, 2013, pp. 1–8.
- [3] Lancaster, H.O. “Significance Tests in Discrete Distributions.” *JASA*, Vol. 56, Number 294, 1961, pp. 223–234.

- [4] McNemar, Q. “Note on the Sampling Error of the Difference Between Correlated Proportions or Percentages.” *Psychometrika*, Vol. 12, Number 2, 1947, pp. 153–157.
- [5] Mosteller, F. “Some Statistical Problems in Measuring the Subjective Response to Drugs.” *Biometrics*, Vol. 8, Number 3, 1952, pp. 220–226.

## See Also

`fitcensemble` | `predict` | `testcholdout` | `testckfold`

## Topics

“Hypothesis Tests”

**Introduced in R2015a**

## compareHoldout

**Class:** CompactClassificationNaiveBayes

Compare accuracies of two classification models using new data

`compareHoldout` statistically assesses the accuracies of two classification models. The function first compares their predicted labels against the true labels, and then it detects whether the difference between the misclassification rates is statistically significant.

You can assess whether the accuracies of the classification models are different, or whether one classification model performs better than another. `compareHoldout` can conduct several McNemar test on page 32-1055 variations, including the asymptotic test, the exact-conditional test, and the mid- $p$ -value test. For cost-sensitive assessment on page 32-1053, available tests include a chi-square test (requires an Optimization Toolbox license) and a likelihood ratio test.

### Syntax

```
h = compareHoldout(C1,C2,T1,T2,ResponseVarName)
```

```
h = compareHoldout(C1,C2,T1,T2,Y)
```

```
h = compareHoldout(C1,C2,X1,X2,Y,Name,Value)
```

```
h = compareHoldout(C1,C2,X1,X2,Y)
```

```
h = compareHoldout(C1,C2,X1,X2,Y,Name,Value)
```

```
[h,p,e1,e2] = compareHoldout(____)
```

### Description

`h = compareHoldout(C1,C2,T1,T2,ResponseVarName)` returns the test decision from testing the null hypothesis that the trained classification models `C1` and `C2` have equal accuracy for predicting the true class labels stored as `ResponseVarName`. The alternative hypothesis is that the labels have unequal accuracy.

The first classification model `C1` uses predictor data in table `T1` and `C2` uses `T2`. The software conducts the mid- $p$ -value McNemar test to compare the accuracies.

$h = 1$  indicates to reject the null hypothesis at the 5% significance level.  $h = 0$  indicates to not reject the null hypothesis at 5% level.

Examples of tests you can conduct include:

- Compare the accuracies of a simple classification model and a model that is more complex by passing the same set of predictor data (i.e.,  $T1 = T2$ ).
- Compare the accuracies of two perhaps different models using two perhaps different sets of predictors.
- Perform various types of feature selection. For example, you can compare the accuracy of a model trained using a set of predictors to the accuracy of one trained on a subset or different set of those predictors. You can arbitrarily choose the set of predictors, or use a feature selection technique like PCA or sequential feature selection (see `pca` and `sequentialfs`).

`h = compareHoldout(C1,C2,T1,T2,Y)` returns the test decision from testing the null hypothesis that the trained classification models `C1` and `C2` have equal accuracy for predicting the true class labels `Y`. The alternative hypothesis is that the labels have unequal accuracy.

`h = compareHoldout(C1,C2,X1,X2,Y,Name,Value)` returns the result of the hypothesis test with additional options specified by one or more `Name,Value` pair arguments. For example, you can specify the type of alternative hypothesis, and the type of test, or you can supply a cost matrix.

`h = compareHoldout(C1,C2,X1,X2,Y)` returns the test decision from testing the null hypothesis that the trained classification models `C1` and `C2` have equal accuracy for predicting the true class labels `Y`. The alternative hypothesis is that the labels have unequal accuracy.

The first classification model `C1` uses predictor data `X1` and `C2` uses `X2`. The software conducts the mid- $p$ -value McNemar test to compare the accuracies.

`h = compareHoldout(C1,C2,X1,X2,Y,Name,Value)` returns the result of the hypothesis test with additional options specified by one or more `Name,Value` pair arguments. For example, you can specify the type of alternative hypothesis, and the type of test, or you can supply a cost matrix.

`[h,p,e1,e2] = compareHoldout(____)` returns the  $p$ -value for the hypothesis test ( $p$ ) and the respective classification losses on page 32-1031 of each set of predicted class labels ( $e1$  and  $e2$ ) using any of the input arguments in the previous syntaxes.

## Input Arguments

### **c1 — Trained naive Bayes classification model**

`ClassificationNaiveBayes` model object | `CompactClassificationNaiveBayes` model object

Trained naive Bayes classification model, specified as a `ClassificationNaiveBayes` or `CompactClassificationNaiveBayes` model object. That is, `C1` is a trained classification model returned by `fitcnb` or `compact`.

### **c2 — Trained classification model**

trained classification model object | trained, compact classification model object

Trained classification model, specified as any trained or compact classification model object described in this table.

Trained Model Type	Model Object	Returned By
Classification tree	<code>ClassificationTree</code>	<code>fitctree</code>
Discriminant analysis	<code>ClassificationDiscriminant</code>	<code>fitcdiscr</code>
Ensemble of bagged classification models	<code>ClassificationBaggedEnsemble</code>	<code>fitcensemble</code>
Ensemble of classification models	<code>ClassificationEnsemble</code>	<code>fitcensemble</code>
Multiclass, error-correcting output codes (ECOC)	<code>ClassificationECOC</code>	<code>fitcecoc</code>
$k$ NN	<code>ClassificationKNN</code>	<code>fitcknn</code>
Naive Bayes	<code>ClassificationNaiveBayes</code>	<code>fitcnb</code>
Support vector machine (SVM)	<code>ClassificationSVM</code>	<code>fitcsvm</code>
Compact discriminant analysis	<code>CompactClassificationDiscriminant</code>	<code>compact</code>



Trained Model Type	Model Object	Returned By
Compact ECOC	CompactClassificationECOC	compact
Compact ensemble of classification models	CompactClassificationEnsemble	compact
Compact naive Bayes	CompactClassificationNaiveBayes	compact
Compact SVM	CompactClassificationSVM	compact
Compact classification tree	CompactClassificationTree	compact

### **T1 — Test-set predictor table for first classification model**

table

Test-set predictor table for the first classification model, C1, specified as a table. Each row of T1 corresponds to one observation, and each column corresponds to one predictor variable. Optionally, T1 can contain an additional column for the response variable. T1 must contain all of the predictors used to train C1. Multi-column variables and cell arrays other than cell arrays of character vectors are not allowed.

T1 and T2 must have the same number of rows, and must have the same response values. If T1 and T2 contain the response variable used to train C1 and C2, then, you do not need to specify Y.

Data Types: table

### **T2 — Test-set predictor table for second classification model**

table

Test-set predictor data for the second classification model, C2, specified as a table. Each row of T2 corresponds to one observation, and each column corresponds to one predictor variable. Optionally, T2 can contain an additional column for the response variable. T2 must contain all of the predictors used to train C2. Multi-column variables and cell arrays other than cell arrays of character vectors are not allowed.

T1 and T2 must have the same number of rows and the same response values. If T1 and T2 contain the response variable used to train C1 and C2, then, you do not need to specify Y.

Data Types: table

**x1 — Test-set predictor data for first classification model**

numeric matrix

Test-set predictor data for the first classification model, C1, specified as a numeric matrix.

Each row of X1 corresponds to one observation (also known as an instance or example), and each column corresponds to one variable (also known as a predictor or feature). The variables used to train C1 must compose X1.

The number of rows in X1 and X2 must equal the length of Y.

Data Types: `double` | `single`

**x2 — Test-set predictor data for second classification model**

numeric matrix

Test-set predictor data for the second classification model, C2, specified as a numeric matrix.

Each row of X2 corresponds to one observation (also known as an instance or example), and each column corresponds to one variable (also known as a predictor or feature). The variables used to train C2 must compose X2.

The number of rows in X2 and X1 must equal the length of Y.

Data Types: `double` | `single`

**ResponseVarName — Response variable name**

name of a variable in T1 and T2

Response variable name, specified as the name of a variable in T1 and T2. If T1 and T2 contain the response variable used to train C1 and C2, then you do not need to specify ResponseVarName.

If you specify ResponseVarName, then you must do so as a character vector. For example, if the response variable is stored as T1.Response, then specify it as 'Response'. Otherwise, the software treats all columns of T1 and T2, including Response, as predictors.

The response variable must be a categorical or character array, logical or numeric vector, or cell array of character vectors. If the response variable is a character array, then each element must correspond to one row of the array.

### **y — True class labels**

categorical array | character array | logical vector | vector of numeric values | cell array of character vectors | character vector

True class labels, specified as a categorical or character array, a logical or numeric vector, a cell array of character vectors, or a character vector.

If you specify any array except a character vector, then `CompactClassificationNaiveBayes.compareHoldout` treats all variables or columns in `X1` and `X2` as predictor variables.

If you specify a character vector, then `X1` and `X2` must be tables, their response variables must have the same name and values, and `Y` must be the common name. For example, if `X1.Labels` and `X2.Labels` are the response variables, then `Y` must be `'Labels'` and `X1.Labels` and `X2.Labels` must be equivalent.

If `Y` is a character array, then each element must correspond to one row of the array.

The number of rows in `X1` and `X2` must equal the length of `Y`.

Data Types: categorical | char | logical | single | double | cell

## **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'Alternative', 'greater', 'Test', 'asymptotic', 'Cost', [0 2; 1 0]` specifies to test whether the first set of first predicted class labels is more accurate than the second set, to conduct the asymptotic McNemar test, and to penalize misclassifying observations with the true label `ClassNames{1}` twice as much as for misclassifying observations with the true label `ClassNames{2}`.

### **Alpha — Hypothesis test significance level**

0.05 (default) | scalar value in the interval (0,1)

Hypothesis test significance level, specified as the comma-separated pair consisting of 'Alpha' and a scalar value in the interval (0,1).

Example: 'Alpha',0.1

Data Types: single | double

#### **Alternative — Alternative hypothesis to assess**

'unequal' (default) | 'greater' | 'less'

Alternative hypothesis to assess, specified as the comma-separated pair consisting of 'Alternative' and one of these values listed in the table.

Value	Alternative hypothesis
'unequal' (default)	For predicting $Y$ , the set of predictions resulting from $C1$ applied to $X1$ and $C2$ applied to $X2$ have unequal accuracies.
'greater'	For predicting $Y$ , the set of predictions resulting from $C1$ applied to $X1$ is more accurate than $C2$ applied to $X2$ .
'less'	For predicting $Y$ , the set of predictions resulting from $C1$ applied to $X1$ is less accurate than $C2$ applied to $X2$ .

Example: 'Alternative', 'greater'

Data Types: char

#### **ClassNames — Class names**

categorical vector | cell array of character vectors | character array | logical vector | numeric vector

Class names, specified as the comma-separated pair consisting of 'ClassNames' and a categorical or character array, logical or numeric vector, or cell array of character vectors. You must set `ClassNames` using the data type of  $Y$ .

If `ClassNames` is a character array, then each element must correspond to one *row* of the array.

Use `ClassNames` to:

- Specify the order of any input argument dimension that corresponds to class order. For example, use `ClassNames` to specify the order of the dimensions of `Cost`.

- Select a subset of classes for testing. For example, suppose that the set of all distinct class names in  $Y$  is  $\{'a', 'b', 'c'\}$ . To train and test models using observations from classes 'a' and 'c' only, specify 'ClassNames', {'a', 'c'}.

The default is the set of all distinct class names in  $Y$ .

Example: 'ClassNames', {'b', 'g'}

Data Types: categorical | char | logical | single | double | cell

### Cost — Misclassification cost

square matrix | structure array

Misclassification cost, specified as the comma-separated pair consisting of 'Cost' and a square matrix or structure array. If you specify:

- If you specify the square matrix  $Cost$ , then  $Cost(i, j)$  is the cost of classifying a point into class  $j$  if its true class is  $i$ . That is, the rows correspond to the true class and the columns correspond to the predicted class. To specify the class order for the corresponding rows and columns of  $Cost$ , additionally specify the `ClassNames` name-value pair argument.
- If you specify the structure  $S$ , then  $S$  must have two fields:
  - $S.ClassNames$ , which contains the class names as a variable of the same data type as  $Y$ . You can use this field to specify the order of the classes.
  - $S.ClassificationCosts$ , which contains the cost matrix, with rows and columns ordered as in  $S.ClassNames$

If you specify  $Cost$ , then `CompactClassificationNaiveBayes.compareHoldout` cannot conduct one-sided, exact, or mid- $p$  tests. You must also specify 'Alternative', 'unequal', 'Test', 'asymptotic'. For cost-sensitive testing options, see the `CostTest` name-value pair argument.

It is a best practice to supply the same cost matrix used to train the classification models.

The default is  $Cost(i, j) = 1$  if  $i \neq j$ , and  $Cost(i, j) = 0$  if  $i = j$ .

Example: 'Cost', [0 1 2 ; 1 0 2; 2 2 0]

Data Types: double | single | struct

**CostTest — Cost-sensitive test type**

'likelihood' (default) | 'chisquare'

Cost-sensitive test type, specified as the comma-separated pair consisting of 'CostTest' and 'chisquare' or 'likelihood'. Unless you specify a cost matrix using the Cost name-value pair argument, CompactClassificationNaiveBayes.compareHoldout ignores CostTest.

This table summarizes the available options for cost-sensitive testing.

Value	Asymptotic test type	Requirements
'chisquare'	Chi-square test	Optimization Toolbox license to implement quadprog
'likelihood'	Likelihood ratio test	None

For more details, see “Cost-Sensitive Testing” on page 32-6515.

Example: 'CostTest', 'chisquare'

Data Types: char

**Test — Test to conduct**

'asymptotic' | 'exact' | 'midp'

Test to conduct, specified as the comma-separated pair consisting of 'Test' and 'asymptotic', 'exact', and 'midp'. This table summarizes the available options for cost-insensitive testing.

Value	Description
'asymptotic'	Asymptotic McNemar test
'exact'	Exact-conditional McNemar test
'midp' (default)	Mid- <i>p</i> -value McNemar test

For more details, see “McNemar Tests” on page 32-6517.

For cost-sensitive testing, Test must be 'asymptotic'. When you specify the Cost name-value pair argument, and choose a cost-sensitive test using the CostTest name-value pair argument, 'asymptotic' is the default.

Example: 'Test', 'asymptotic'

Data Types: char

---

**Note** NaNs, <undefined> values, and empty character vectors ( ' ' ) indicate missing values. CompactClassificationNaiveBayes.compareHoldout:

- Removes missing values in Y and the corresponding rows of X1 and X2
  - Predicts classes whether X1 and X2 have missing observations.
- 

## Output Arguments

### **h** — Hypothesis test result

1 | 0

Hypothesis test result, returned as a logical value.

$h = 1$  indicates the rejection of the null hypothesis at the Alpha significance level.

$h = 0$  indicates failure to reject the null hypothesis at the Alpha significance level.

### **p** — *p*-value

scalar in the interval [0,1]

*p*-value of the test, returned as a scalar in the interval [0,1]. *p* is the probability that a random test statistic is at least as extreme as the observed test statistic, given that the null hypothesis is true.

CompactClassificationNaiveBayes.compareHoldout estimates *p* using the distribution of the test statistic, which varies with the type of test. For details on test statistics derived from the available variants of the McNemar test, see “McNemar Tests” on page 32-6517. For details on test statistics derived from cost-sensitive tests, see “Cost-Sensitive Testing” on page 32-6515.

### **e1** — Classification loss

scalar

Classification loss on page 32-1058, returned as a scalar. *e1* summarizes the accuracy of the first set of class labels predicting the true class labels (Y).

`CompactClassificationNaiveBayes.compareHoldout` applies the first test-set predictor data ( $X_1$ ) to the first classification model ( $C_1$ ) to estimate the first set of class labels. Then, the function compares the estimated labels to  $Y$  to obtain the classification loss.

For cost-insensitive testing,  $e_1$  is the misclassification rate. That is,  $e_1$  is the proportion of misclassified observations, which is a scalar in the interval  $[0,1]$ .

For cost-sensitive testing,  $e_1$  is the misclassification cost. That is,  $e_1$  is the weighted average of the misclassification costs, in which the weights are the respective estimated proportions of misclassified observations.

### **$e_2$ — Classification loss**

scalar

Classification loss on page 32-1058, returned as a scalar.  $e_2$  summarizes the accuracy of the second set of class labels predicting the true class labels ( $Y$ ).

`CompactClassificationNaiveBayes.compareHoldout` applies the second test-set predictor data ( $X_2$ ) to the second classification model ( $C_2$ ) to estimate the second set of class labels. Then the function compares the estimated labels to  $Y$  to obtain the classification loss.

For cost-insensitive testing,  $e_2$  is the misclassification rate. That is,  $e_2$  is the proportion of misclassified observations, which is a scalar in the interval  $[0,1]$ .

For cost-sensitive testing,  $e_2$  is the misclassification cost. That is,  $e_2$  is the weighted average of the misclassification costs, in which the weights are the respective estimated proportions of misclassified observations.

## **Examples**

### **Compare Accuracies of Two Different Classification Models**

Train two classification models using different algorithms. Conduct a statistical test comparing the misclassification rates of the two models on a held-out set.

Load the `ionosphere` data set.

```
load ionosphere;
```



Create a partition that evenly splits the data into training and testing sets.

```
rng(1); % For reproducibility
CVP = cvpartition(Y, 'holdout', 0.5);
idxTrain = training(CVP); % Training-set indices
idxTest = test(CVP); % Test-set indices
```

CVP is a cross-validation partition object that specifies the training and test sets.

Train an SVM model and an ensemble of 100 bagged classification trees. For the SVM model, specify to use the radial basis function kernel and a heuristic procedure to determine the kernel scale.

```
C1 = fitcsvm(X(idxTrain,:), Y(idxTrain), 'Standardize', true, ...
    'KernelFunction', 'RBF', 'KernelScale', 'auto');
C2 = fitensemble(X(idxTrain,:), Y(idxTrain), 'Bag', 100, 'Tree', ...
    'Type', 'classification');
```

C1 is a trained ClassificationSVM model. C2 is a trained ClassificationBaggedEnsemble model.

Test whether the two models have equal predictive accuracies. Use the same test-set predictor data for each model.

```
h = compareHoldout(C1, C2, X(idxTest,:), X(idxTest,:), Y(idxTest))
```

```
h =
    logical
     0
```

h = 0 indicates to not reject the null hypothesis that the two models have equal predictive accuracies.

### Assess Whether One Classification Model Classifies Better Than Another

Train two classification models using the same algorithm, but adjust a hyperparameter to make the algorithm more complex. Conduct a statistical test to assess whether the simpler model has better accuracy in held-out data than the more complex model.

Load the ionosphere data set.

```
load ionosphere;
```

Create a partition that evenly splits the data into training and testing sets.

```
rng(1); % For reproducibility
CVP = cvpartition(Y, 'holdout', 0.5);
idxTrain = training(CVP); % Training-set indices
idxTest = test(CVP); % Test-set indices
```

CVP is a cross-validation partition object that specifies the training and test sets.

Train two SVM models: one that uses a linear kernel (the default for binary classification) and one that uses the radial basis function kernel. Use the default kernel scale of 1.

```
C1 = fitcsvm(X(idxTrain,:), Y(idxTrain), 'Standardize', true);
C2 = fitcsvm(X(idxTrain,:), Y(idxTrain), 'Standardize', true, ...
    'KernelFunction', 'RBF');
```

C1 and C2 are trained ClassificationSVM models.

Test the null hypothesis that the simpler model (C1) is at most as accurate as the more complex model (C2). Because the test-set size is large, conduct the asymptotic McNemar test, and compare the results with the mid-  $p$ -value test (the cost-insensitive testing default). Request to return  $p$ -values and misclassification rates.

```
Asymp = zeros(4,1); % Preallocation
MidP = zeros(4,1);

[Asymp(1), Asymp(2), Asymp(3), Asymp(4)] = compareHoldout(C1, C2, ...
    X(idxTest,:), X(idxTest,:), Y(idxTest), 'Alternative', 'greater', ...
    'Test', 'asymptotic');
[MidP(1), MidP(2), MidP(3), MidP(4)] = compareHoldout(C1, C2, ...
    X(idxTest,:), X(idxTest,:), Y(idxTest), 'Alternative', 'greater');
table(Asymp, MidP, 'RowNames', {'h' 'p' 'e1' 'e2'})
```

```
ans =
```

```
4x2 table
```

```
Asymp MidP
```

	1	1
h		
p	7.2801e-09	2.7649e-10
e1	0.13714	0.13714
e2	0.33143	0.33143

The  $p$ -value is close to zero for both tests, which indicates strong evidence to reject the null hypothesis that the simpler model is less accurate than the more complex model. No matter what test you specify, `compareHoldout` returns the same type of misclassification measure for both models.

### Conduct a Cost-Sensitive Comparison of Two Classification Models

For data sets with imbalanced class representations, or for data sets with imbalanced false-positive and false-negative costs, you can statistically compare the predictive performances of two classification models by including a cost matrix in the analysis.

Load the `arrhythmia` data set. Determine the class representations in the data.

```
load arrhythmia;
Y = categorical(Y);
tabulate(Y);
```

Value	Count	Percent
1	245	54.20%
2	44	9.73%
3	15	3.32%
4	15	3.32%
5	13	2.88%
6	25	5.53%
7	3	0.66%
8	2	0.44%
9	9	1.99%
10	50	11.06%
14	4	0.88%
15	5	1.11%
16	22	4.87%

There are 16 classes, however some are not represented in the data set. Most observations are classified as not having arrhythmia (class 1). To summarize, the data set is highly discrete with imbalanced classes.

Combine all observations with arrhythmia (classes 2 through 15) into one class. Remove those observations with unknown arrhythmia status from the data set.

```
Y = Y(Y ~= '16');  
Y(Y ~= '1') = '2';  
X = X(Y ~= '16', :);
```

Create a partition that evenly splits the data into training and testing sets.

```
rng(1); % For reproducibility  
CVP = cvpartition(Y, 'holdout', 0.5);  
idxTrain = training(CVP); % Training-set indices  
idxTest = test(CVP); % Test-set indices
```

CVP is a cross-validation partition object that specifies the training and test sets.

Create a cost matrix such that misclassifying an arrhythmic patient into the no arrhythmia class is five times worse than misclassifying a patient without arrhythmia into the arrhythmia class. Classifying correctly incurs no cost. The rows indicate the true class and the columns indicate the predicted class. When conducting a cost-sensitive analysis, it is a good practice to specify the order of the classes.

```
cost = [0 1; 5 0];  
ClassNames = categorical([2 1]);
```

Train two boosting ensembles of 50 classification trees, one that uses AdaBoostM1 and the other that uses LogitBoost. Because there are missing values, specify to use surrogate splits. Train the models using the cost matrix.

```
t = templateTree('Surrogate', 'on');  
numTrees = 50;  
C1 = fitensemble(X(idxTrain, :), Y(idxTrain), 'AdaBoostM1', numTrees, t, ...  
    'Cost', cost);  
C2 = fitensemble(X(idxTrain, :), Y(idxTrain), 'LogitBoost', numTrees, t, ...  
    'Cost', cost);
```

C1 and C2 are trained ClassificationEnsemble models.

Test whether the AdaBoostM1 ensemble (C1) and the LogitBoost ensemble (C2) have equal predictive accuracy. Supply the cost matrix. Conduct the asymptotic, likelihood ratio, cost-sensitive test (the default when you pass in a cost matrix). Request to return  $p$ -values and misclassification costs.

```
[h,p,e1,e2] = compareHoldout(C1,C2,X(idxTest,:),X(idxTest,:),Y(idxTest),...
    'Cost',cost)

h =

    logical

     0

p =

    0.0743

e1 =

    1.3581

e2 =

    1.6186
```

$h = 0$  indicates to not reject the null hypothesis that the two models have equal predictive accuracies.

### Select Features Using Statistical Accuracy Comparison

Reduce classification model complexity by selecting a subset of predictor variables (features) from a larger set. Then, statistically compare the out-of-sample accuracy between the two models.

Load the `ionosphere` data set.

```
load ionosphere;
```

Create a partition that evenly splits the data into training and testing sets.

```
rng(1); % For reproducibility
CVP = cvpartition(Y,'holdout',0.5);
```

```
idxTrain = training(CVP);           % Training-set indices
idxTest  = test(CVP);              % Test-set indices
```

CVP is a cross-validation partition object that specifies the training and test sets.

Train an ensemble of 100 boosted classification trees using AdaBoostM1 and the entire set of predictors. Inspect the importance measure for each predictor.

```
nTrees = 100;
C2 = fitensemble(X(idxTrain,:), Y(idxTrain), 'AdaBoostM1', nTrees, 'Tree');
predImp = predictorImportance(C2);
```

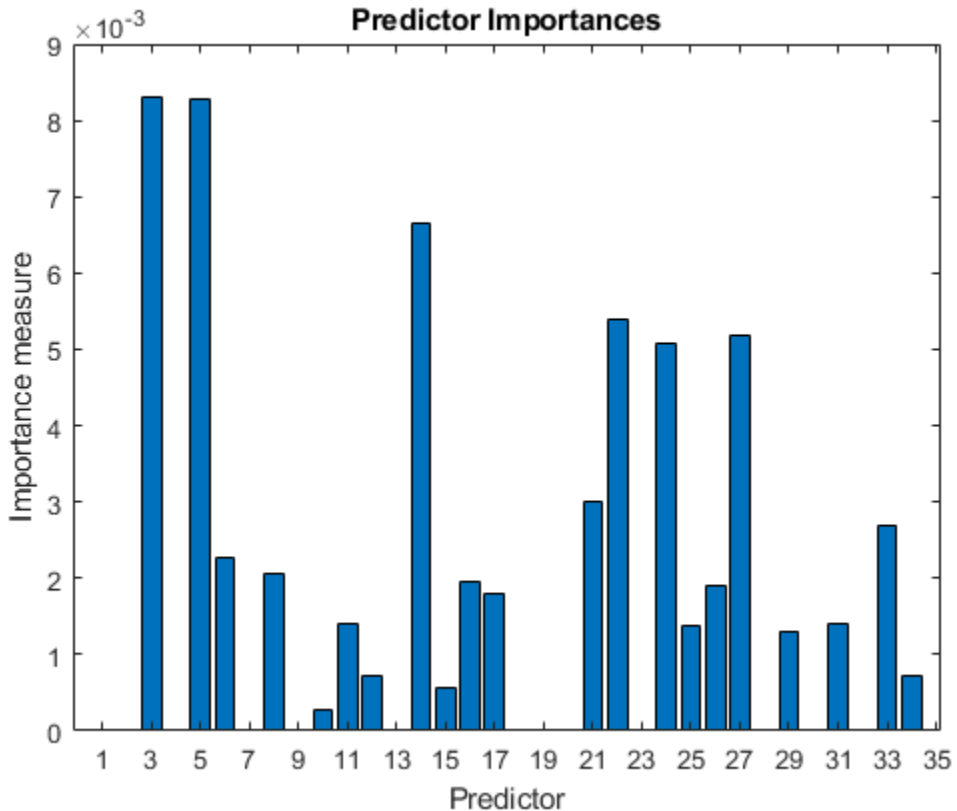
```
figure;
bar(predImp);
h = gca;
h.XTick = 1:2:h.XLim(2)
title('Predictor Importances');
xlabel('Predictor');
ylabel('Importance measure');
```

h =

Axes with properties:

```
        XLim: [-0.2000 35.2000]
        YLim: [0 0.0090]
        XScale: 'linear'
        YScale: 'linear'
GridLineStyle: '-'
        Position: [0.1300 0.1100 0.7750 0.8150]
        Units: 'normalized'
```

Use GET to show all properties



Identify the top five predictors in terms of their importance.

```
[~,idxSort] = sort(predImp, 'descend');
idx5 = idxSort(1:5);
```

Train another ensemble of 100 boosted classification trees using AdaBoostM1 and the five predictors with the best importance.

```
C1 = fitensemble(X(idxTrain,idx5),Y(idxTrain),'AdaBoostM1',nTrees,...
    'Tree');
```

Test whether the two models have equal predictive accuracies. Specify the reduced test-set predictor data for C1 and the full test-set predictor data for C2.

```
[h,p,e1,e2] = compareHoldout(C1,C2,X(idxTest,idx5),X(idxTest,:),Y(idxTest))
```

```
h =  
  logical  
  0  
  
p =  
  0.7744  
  
e1 =  
  0.0914  
  
e2 =  
  0.0857
```

$h = 0$  indicates to not reject the null hypothesis that the two models have equal predictive accuracies. This result favors the simpler ensemble, C1.

## Definitions

### Cost-Sensitive Testing

Conduct cost-sensitive testing when the cost of misclassification is imbalanced. When conducting a cost-sensitive analysis, you can account for the cost imbalance in training the classification models, and then in statistically comparing them.

If the cost of misclassification is imbalanced, then the misclassification rate tends to be a poorly performing classification loss on page 32-6520. Use misclassification cost instead to compare classification models.

Misclassification costs are often unbalanced in applications. For example, consider classifying subjects based on a set of predictors into two categories: healthy and sick. Misclassifying a sick subject as healthy poses a danger to the subject's life. However,



misclassifying a healthy subject as sick can cause some inconvenience, but does not pose any danger. In this situation, you assign misclassification costs such that misclassifying a sick subject as healthy is more costly than misclassifying a healthy subject as sick.

The definitions that follow summarize the cost-sensitive tests. In the definitions:

- $n_{ijk}$  and  $\hat{\pi}_{ijk}$  are the number and estimated proportion of test-sample observations with true class  $k$  that the first classification model assigns label  $i$ . The second classification model assigns label  $j$ . The unknown, true value of  $\hat{\pi}_{ijk}$  is  $\pi_{ijk}$ . The test-set sample size is  $\sum_{i,j,k} n_{ijk} = n_{test}$ .  $\sum_{i,j,k} \pi_{ijk} = \sum_{i,j,k} \pi_{ijk} = 1$ .
- $c_{ij}$  is the relative cost of assigning label  $j$  to an observation with true class  $i$ .  $c_{ii} = 0$ ,  $c_{ij} \geq 0$ , and, for at least one  $(i,j)$  pair,  $c_{ij} > 0$ .
- All subscripts take on integer values from 1 through  $K$ , which is the number of classes.
- The expected difference in the misclassification costs of the two classification models is

$$\delta = \sum_{i=1}^K \sum_{j=1}^K \sum_{k=1}^K (c_{ki} - c_{kj}) \pi_{ijk}.$$

- The hypothesis test is
 
$$H_0 : \delta = 0$$

$$H_1 : \delta \neq 0$$

The available cost-sensitive tests are appropriate for two-tailed testing.

Available, asymptotic tests that address imbalanced costs are a chi-square test and a likelihood ratio test.

- Chi-square test — The chi-square test statistic is based on the Pearson and Neyman chi-square test statistics, but with a Laplace correction factor to account for any  $n_{ijk} = 0$ . The test statistic is

$$t_{\chi^2}^* = \sum_{i \neq j} \sum_k \frac{\left( n_{ijk} + 1 - (n_{test} + K^3) \hat{\pi}_{ijk}^{(1)} \right)^2}{n_{ijk} + 1}.$$

If  $1 - F_{\chi^2}(t_{\chi^2}^*; 1) < \alpha$ , then reject  $H_0$ .

•

$\hat{\pi}_{ijk}^{(1)}$  are estimated by minimizing  $t_{\chi^2}^*$  under the constraint that  $\delta = 0$ .

•

$F_{\chi^2}(x; 1)$  is the  $\chi^2$  C.D.F. with one degree of freedom evaluated at  $x$ .

- Likelihood ratio test — The likelihood ratio test is based on  $N_{ijk}$ , which are binomial random variables having sample size  $n_{test}$  and success probability  $\pi_{ijk}$ . They represent the random number of observations with true class  $k$  that the first classification model assigns label  $i$ . The second classification model assigns label  $j$ . Jointly, their distribution is multinomial.

The test statistic is

$$t_{LRT}^* = 2 \log \left[ \frac{P\left(\bigcap_{i,j,k} N_{ijk} = n_{ijk}; n_{test}, \pi_{ijk} = \pi_{ijk}^{(2)}\right)}{P\left(\bigcap_{i,j,k} N_{ijk} = n_{ijk}; n_{test}, \pi_{ijk} = \pi_{ijk}^{(3)}\right)} \right].$$

If  $1 - F_{\chi^2}(t_{LRT}^*; 1) < \alpha$ , then reject  $H_0$ .

•

$\hat{\pi}_{ijk}^{(2)} = \frac{n_{ijk}}{n_{test}}$  is the unrestricted MLE of  $\pi_{ijk}$ .

•

$\hat{\pi}_{ijk}^{(3)} = \frac{n_{ijk}}{n_{test} + \lambda(c_{ki} - c_{kj})}$  is the MLE under the null hypothesis that  $\delta = 0$ .  $\lambda$  is the solution to

$$\sum_{i,j,k} \frac{n_{ijk}(c_{ki} - c_{kj})}{n_{test} + \lambda(c_{ki} - c_{kj})} = 0.$$

•

$F_{\chi^2}(x; 1)$  is the  $\chi^2$  C.D.F. with one degree of freedom evaluated at  $x$ .

## McNemar Tests

McNemar Tests are hypothesis tests that compare two population proportions while addressing the issues resulting from two dependent, matched-pair samples.

One way to compare the predictive accuracies of two classification models is:

- 1 Partition the data into training and test sets.
- 2 Train both classification models using the training set.
- 3 Predict class labels using the test set.
- 4 Summarize the results in a two-by-two table resembling this figure.

		Model 2		
		Correct	Incorrect	
Model 1	Correct	$n_{11}$	$n_{12}$	$n_{1\bullet}$
	Incorrect	$n_{21}$	$n_{22}$	$n_{2\bullet}$
		$n_{\bullet 1}$	$n_{\bullet 2}$	$n_{test}$

$n_{ii}$  are the number of concordant pairs, that is, the number of observations that both models classify the same way (correctly or incorrectly).  $n_{ij}$ ,  $i \neq j$ , are the number of discordant pairs, that is, the number of observations that models classify differently (correctly or incorrectly).

The misclassification rates for Models 1 and 2 are

$$\hat{\pi}_{2\bullet} = n_{2\bullet} / n$$

and  $\hat{\pi}_{\bullet 2} = n_{\bullet 2} / n$ , respectively. A two-sided test for comparing the accuracy of the two models is

$$H_0 : \pi_{\bullet 2} = \pi_{2\bullet}$$

$$H_1 : \pi_{\bullet 2} \neq \pi_{2\bullet}$$

The null hypothesis suggests that the population exhibits marginal homogeneity, which reduces the null hypothesis to  $H_0 : \pi_{12} = \pi_{21}$ . Also, under the null hypothesis,  $N_{12} \sim \text{Binomial}(n_{12} + n_{21}, 0.5)$  [1].

These facts are the basis for these, available McNemar test variants: the asymptotic, exact-conditional and mid-p-value McNemar tests. The definitions that follow summarize the available variants.

- Asymptotic — The asymptotic McNemar test statistics and rejection regions (for significance-level  $\alpha$ ) are:

- For one-sided tests, the test statistic

$$t_{a1}^* = \frac{n_{12} - n_{21}}{\sqrt{n_{12} + n_{21}}}.$$

If  $1 - \Phi\left(|t_1^*|\right) < \alpha$ , where  $\Phi$  is the standard Gaussian C.D.F., then reject  $H_0$ .

- For two-sided tests, the test statistic

$$t_{a2}^* = \frac{(n_{12} - n_{21})^2}{n_{12} + n_{21}}.$$

If  $1 - F_{\chi^2}\left(t_2^*; m\right) < \alpha$ , where  $F_{\chi^2}(x; m)$  is the  $\chi_m^2$  C.D.F. evaluated at  $x$ , then reject  $H_0$ .

This variant requires large-sample theory, specifically, the Gaussian approximation to the binomial distribution. Therefore:

- The total number of discordant pairs,  $n_d = n_{12} + n_{21}$  must be greater than 10 ([1], Ch. 10.1.4).
- In general, asymptotic tests do not guarantee nominal coverage. The observed probability of falsely rejecting the null hypothesis can exceed  $\alpha$ . Simulation studies in [18] suggest this, but the asymptotic McNemar test performs well in terms of statistical power.
- Exact Conditional — The exact-conditional McNemar test statistics and rejection regions (for significance-level  $\alpha$ ) are ([36], [38]):

- For one-sided tests, the test statistic

$$t_1^* = n_{12}$$

If  $F_{\text{Bin}}\left(t_1^*; n_d, 0.5\right) < \alpha$ , where  $F_{\text{Bin}}(x; n, p)$  is the binomial C.D.F. with sample size  $n$  and success probability  $p$  evaluated at  $x$ , then reject  $H_0$ .

- For two-sided tests, the test statistic

$$t_2^* = \min(n_{12}, n_{21})$$

If  $F_{\text{Bin}}(t_2^*; n_d, 0.5) < \alpha / 2$ , then reject  $H_0$ .

The exact conditional test always attains nominal coverage. Simulation studies in [18] suggest that the test is conservative, and then show that the test lacks statistical power compared to other variants. For small or highly discrete test samples, consider using the mid- $p$ -value test ([1], Ch. 3.6.3). For details, see `Test` and “McNemar Tests” on page 32-6517.

- Mid- $p$ -value test — The mid- $p$ -value McNemar test statistics and rejection regions (for significance-level  $\alpha$ ) are ([32]):

- For one-sided tests, the test statistic

$$t_1^* = n_{12}$$

If  $F_{\text{Bin}}(t_1^* - 1; n_{12} + n_{21}, 0.5) + 0.5f_{\text{Bin}}(t_1^*; n_{12} + n_{21}, 0.5) < \alpha$ , where  $F_{\text{Bin}}(x; n, p)$  and  $f_{\text{Bin}}(x; n, p)$  are the binomial C.D.F. and P.D.F, respectively, with sample size  $n$  and success probability  $p$  evaluated at  $x$ , then reject  $H_0$ .

- For two-sided tests, the test statistic

$$t_2^* = \min(n_{12}, n_{21})$$

If  $F_{\text{Bin}}(t_2^* - 1; n_{12} + n_{21} - 1, 0.5) + 0.5f_{\text{Bin}}(t_2^*; n_{12} + n_{21}, 0.5) < \alpha / 2$ , then reject  $H_0$ .

The mid- $p$ -value test addresses the over-conservative behavior of the exact conditional test. The simulation studies in [18] demonstrate that this test attains nominal coverage, and has good statistical power.

## Classification Loss

Classification losses indicate the accuracy of a classification model or set of predicted labels. Two classification losses are misclassification rate and cost.

`CompactClassificationNaiveBayes.compareHoldout` returns the classification losses (see `e1` and `e2`) under the alternative hypothesis (i.e., the unrestricted classification losses).  $n_{ijk}$  is the number of test-sample observations with true class  $k$  that

the first classification model assigns label  $i$  and the second classification model assigns

label  $j$ , and the corresponding estimated proportion is  $\hat{\pi}_{ijk} = \frac{n_{ijk}}{n_{test}}$ . The test-set sample

size is  $\sum_{i,j,k} n_{ijk} = n_{test}$ . The indices are taken from 1 through  $K$ , the number of classes.

- Misclassification rate, or classification error, is a scalar in the interval  $[0,1]$  representing the proportion of misclassified observations. That is, the misclassification rate for the first classification model is

$$e_1 = \sum_{j=1}^K \sum_{k=1}^K \sum_{i \neq k} \hat{\pi}_{ijk}.$$

For the misclassification rate of the second classification model ( $e_2$ ), switch the indices  $i$  and  $j$  in the formula.

Classification accuracy decreases as the misclassification rate increases to 1.

- Misclassification cost is a nonnegative scalar and is a measure of classification quality relative to the values the specified cost matrix elements. Its interpretation depends on the specified costs of misclassification. Misclassification cost is the weighted average of the costs of misclassification (specified in a cost matrix,  $C$ ) in which the weights are the respective, estimated proportions of misclassified observations. The misclassification cost for the first classification model is

$$e_1 = \sum_{j=1}^K \sum_{k=1}^K \sum_{i \neq k} \hat{\pi}_{ijk} c_{ki},$$

where  $c_{kj}$  is the cost of classifying an observation into class  $j$  if its true class is  $k$ . For the misclassification cost of the second classification model ( $e_2$ ), switch the indices  $i$  and  $j$  in the formula.

In general, for a fixed cost matrix, classification accuracy decreases as misclassification cost increases.

## Tips

- One way to perform cost-insensitive feature selection is:
  - 1 Train the first classification model ( $C1$ ) using the full predictor set.

- 2 Train the second classification model (C2) using the reduced predictor set.
- 3 Specify X1 as the full, test-set predictor data and X2 as the reduced test-set predictor data.
- 4 Enter `compareHoldout(C1,C2,X1,X2,'Alternative','less')`. If `compareHoldout` returns 1, then there is enough evidence to suggest that the classification model that uses fewer predictors performs better than the model that uses the full predictor set.

Alternatively, you can assess whether there is a significant difference between the accuracies of the two models. To perform this assessment, remove the 'Alternative', 'greater' specification in step 4. `compareHoldout` conducts a two-sided test, and `h = 0` indicates that there is not enough evidence to suggest a difference in the accuracy of the two models.

- Cost-sensitive tests perform numerical optimization, which requires additional computational resources. The likelihood ratio test conducts numerical optimization indirectly by finding the root of a Lagrange multiplier in an interval. For some data sets, if the root lies close to the boundaries of the interval, then the method can fail. Therefore, if you have an Optimization Toolbox license, consider conducting the cost-sensitive chi-square test instead. For more details, see `CostTest` and “Cost-Sensitive Testing” on page 32-1053.

## Alternatives

To directly compare the accuracy of two sets of class labels in predicting a set of true class labels, use `testcholdout`.

## References

- [1] Agresti, A. *Categorical Data Analysis*, 2nd Ed. John Wiley & Sons, Inc.: Hoboken, NJ, 2002.
- [2] Fagerlan, M.W., S Lydersen, P. Laake. “The McNemar Test for Binary Matched-Pairs Data: Mid-p and Asymptotic Are Better Than Exact Conditional.” *BMC Medical Research Methodology*. Vol. 13, 2013, pp. 1–8.
- [3] Lancaster, H.O. “Significance Tests in Discrete Distributions.” *JASA*, Vol. 56, Number 294, 1961, pp. 223–234.

- [4] McNemar, Q. “Note on the Sampling Error of the Difference Between Correlated Proportions or Percentages.” *Psychometrika*, Vol. 12, Number 2, 1947, pp. 153–157.
- [5] Mosteller, F. “Some Statistical Problems in Measuring the Subjective Response to Drugs.” *Biometrics*, Vol. 8, Number 3, 1952, pp. 220–226.

## See Also

`fitcnb` | `predict` | `testcholdout` | `testckfold`

## Topics

“Hypothesis Tests”

**Introduced in R2015a**



# compareHoldout

**Class:** CompactClassificationSVM

Compare accuracies of two classification models using new data

`compareHoldout` statistically assesses the accuracies of two classification models. The function first compares their predicted labels against the true labels, and then it detects whether the difference between the misclassification rates is statistically significant.

You can assess whether the accuracies of the classification models are different, or whether one classification model performs better than another. `compareHoldout` can conduct several McNemar test on page 32-1055 variations, including the asymptotic test, the exact-conditional test, and the mid- $p$ -value test. For cost-sensitive assessment on page 32-1053, available tests include a chi-square test (requires an Optimization Toolbox license) and a likelihood ratio test.

## Syntax

```
h = compareHoldout(C1,C2,T1,T2,ResponseVarName)
```

```
h = compareHoldout(C1,C2,T1,T2,Y)
```

```
h = compareHoldout( ___,Name,Value)
```

```
h = compareHoldout(C1,C2,X1,X2,Y)
```

```
h = compareHoldout( ___,Name,Value)
```

```
[h,p,e1,e2] = compareHoldout( ___)
```

## Description

`h = compareHoldout(C1,C2,T1,T2,ResponseVarName)` returns the test decision from testing the null hypothesis that the trained classification models `C1` and `C2` have equal accuracy for predicting the true class labels `TBL.ResponseVarName`. The alternative hypothesis is that the labels have unequal accuracy.

The first classification model `C1` uses the predictor data in table `T1` and `C2` uses `T2`.

The software conducts the mid- $p$ -value McNemar test to compare the accuracies.  $h = 1$  indicates to reject the null hypothesis at the 5% significance level.  $h = 0$  indicates to not reject the null hypothesis at 5% level.

Examples of tests you can conduct include:

- Compare the accuracies of a simple classification model and a model that is more complex by passing the same set of predictor data (i.e.,  $T1 = T2$ ).
- Compare the accuracies of two perhaps different models using two perhaps different sets of predictors.
- Perform various types of feature selection. For example, you can compare the accuracy of a model trained using a set of predictors to the accuracy of one trained on a subset or different set of those predictors. You can arbitrarily choose the set of predictors, or use a feature selection technique like PCA or sequential feature selection (see `pca` and `sequentialfs`).

`h = compareHoldout(C1,C2,T1,T2,Y)` returns the test decision from testing the null hypothesis that the trained classification models `C1` and `C2` have equal accuracy for predicting the true class labels `Y`. The alternative hypothesis is that the labels have unequal accuracy.

The first classification model `C1` uses the predictor data in table `T1` and `C2` uses `T2`.

`h = compareHoldout( ____,Name,Value)` returns the result of the hypothesis test with additional options specified by one or more `Name,Value` pair arguments, using any of the previous syntaxes. For example, you can specify the type of alternative hypothesis, and the type of test, or you can supply a cost matrix.

`h = compareHoldout(C1,C2,X1,X2,Y)` returns the test decision from testing the null hypothesis that the trained classification models `C1` and `C2` have equal accuracy for predicting the true class labels `Y`. The alternative hypothesis is that the labels have unequal accuracy.

The first classification model `C1` uses the predictor data in matrix `X1` and `C2` uses `X2`.

`h = compareHoldout( ____,Name,Value)` returns the result of the hypothesis test with additional options specified by one or more `Name,Value` pair arguments, using any of the previous syntaxes. For example, you can specify the type of alternative hypothesis, and the type of test, or you can supply a cost matrix.

`[h,p,e1,e2] = compareHoldout(____)` returns the  $p$ -value for the hypothesis test ( $p$ ) and the respective classification losses on page 32-1058 of each set of predicted class labels ( $e1$  and  $e2$ ), using any of the input arguments in the previous syntaxes.

## Input Arguments

### **c1 — Trained support vector machine classification model**

`ClassificationSVM` model object | `CompactClassificationSVM` model object

Trained support vector machine (SVM) classification model, specified as a `ClassificationSVM` or `CompactClassificationSVM` model object. That is, `C1` is a trained classification model returned by `fitsvm` or `compact`.

### **c2 — Trained classification model**

trained classification model object | trained, compact classification model object

Trained classification model, specified as any trained or compact classification model object described in this table.

Trained Model Type	Model Object	Returned By
Classification tree	<code>ClassificationTree</code>	<code>fitctree</code>
Discriminant analysis	<code>ClassificationDiscriminant</code>	<code>fitcdiscr</code>
Ensemble of bagged classification models	<code>ClassificationBaggedEnsemble</code>	<code>fitcensemble</code>
Ensemble of classification models	<code>ClassificationEnsemble</code>	<code>fitcensemble</code>
Multiclass, error-correcting output codes (ECOC)	<code>ClassificationECOC</code>	<code>fitcecoc</code>
$k$ NN	<code>ClassificationKNN</code>	<code>fitcknn</code>
Naive Bayes	<code>ClassificationNaiveBayes</code>	<code>fitcnb</code>
Support vector machine (SVM)	<code>ClassificationSVM</code>	<code>fitsvm</code>
Compact discriminant analysis	<code>CompactClassificationDiscriminant</code>	<code>compact</code>

Trained Model Type	Model Object	Returned By
Compact ECOC	CompactClassificationECOC	compact
Compact ensemble of classification models	CompactClassificationEnsemble	compact
Compact naive Bayes	CompactClassificationNaiveBayes	compact
Compact SVM	CompactClassificationSVM	compact
Compact classification tree	CompactClassificationTree	compact

**T1 — Test-set predictor table for first classification model**

table

Test-set predictor table for the first classification model, *C1*, specified as a table. Each row of *T1* corresponds to one observation, and each column corresponds to one predictor variable. Optionally, *T1* can contain an additional column for the response variable. *T1* must contain all of the predictors used to train *C1*. Multi-column variables and cell arrays other than cell arrays of character vectors are not allowed.

*T1* and *T2* must have the same number of rows, and must have the same response values. If *T1* and *T2* contain the response variable used to train *C1* and *C2*, then, you do not need to specify *Y*.

Data Types: table

**T2 — Test-set predictor table for second classification model**

table

Test-set predictor data for the second classification model, *C2*, specified as a table. Each row of *T2* corresponds to one observation, and each column corresponds to one predictor variable. Optionally, *T2* can contain an additional column for the response variable. *T2* must contain all of the predictors used to train *C2*. Multi-column variables and cell arrays other than cell arrays of character vectors are not allowed.

*T1* and *T2* must have the same number of rows and the same response values. If *T1* and *T2* contain the response variable used to train *C1* and *C2*, then, you do not need to specify *Y*.

Data Types: table

**x1 — Test-set predictor data for first classification model**

numeric matrix

Test-set predictor data for the first classification model, C1, specified as a numeric matrix.

Each row of X1 corresponds to one observation (also known as an instance or example), and each column corresponds to one variable (also known as a predictor or feature). The variables used to train C1 must compose X1.

The number of rows in X1 and X2 must equal the length of Y.

Data Types: `double` | `single`

**x2 — Test-set predictor data for second classification model**

numeric matrix

Test-set predictor data for the second classification model, C2, specified as a numeric matrix.

Each row of X2 corresponds to one observation (also known as an instance or example), and each column corresponds to one variable (also known as a predictor or feature). The variables used to train C2 must compose X2.

The number of rows in X2 and X1 must equal the length of Y.

Data Types: `double` | `single`

**ResponseVarName — Response variable name**

name of a variable in T1 and T2

Response variable name, specified as the name of a variable in T1 and T2. If T1 and T2 contain the response variable used to train C1 and C2, then you do not need to specify ResponseVarName.

If you specify ResponseVarName, then you must do so as a character vector. For example, if the response variable is stored as T1.Response, then specify it as 'Response'. Otherwise, the software treats all columns of T1 and T2, including Response, as predictors.

The response variable must be a categorical or character array, logical or numeric vector, or cell array of character vectors. If the response variable is a character array, then each element must correspond to one row of the array.

**y — True class labels**

categorical array | character array | logical vector | vector of numeric values | cell array of character vectors | character vector

True class labels, specified as a categorical or character array, a logical or numeric vector, a cell array of character vectors, or a character vector.

If you specify any array except a character vector, then

`CompactClassificationSVM.compareHoldout` treats all variables or columns in `X1` and `X2` as predictor variables.

If you specify a character vector, then `X1` and `X2` must be tables, their response variables must have the same name and values, and `Y` must be the common name. For example, if `X1.Labels` and `X2.Labels` are the response variables, then `Y` must be `'Labels'` and `X1.Labels` and `X2.Labels` must be equivalent.

If `Y` is a character array, then each element must correspond to one row of the array.

The number of rows in `X1` and `X2` must equal the length of `Y`.

Data Types: categorical | char | logical | single | double | cell

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'Alternative', 'greater', 'Test', 'asymptotic', 'Cost', [0 2; 1 0]` specifies to test whether the first set of first predicted class labels is more accurate than the second set, to conduct the asymptotic McNemar test, and to penalize misclassifying observations with the true label `ClassNames{1}` twice as much as for misclassifying observations with the true label `ClassNames{2}`.

**Alpha — Hypothesis test significance level**

0.05 (default) | scalar value in the interval (0,1)

Hypothesis test significance level, specified as the comma-separated pair consisting of 'Alpha' and a scalar value in the interval (0,1).

Example: 'Alpha',0.1

Data Types: single | double

#### **Alternative — Alternative hypothesis to assess**

'unequal' (default) | 'greater' | 'less'

Alternative hypothesis to assess, specified as the comma-separated pair consisting of 'Alternative' and one of these values listed in the table.

Value	Alternative hypothesis
'unequal' (default)	For predicting $Y$ , the set of predictions resulting from C1 applied to X1 and C2 applied to X2 have unequal accuracies.
'greater'	For predicting $Y$ , the set of predictions resulting from C1 applied to X1 is more accurate than C2 applied to X2.
'less'	For predicting $Y$ , the set of predictions resulting from C1 applied to X1 is less accurate than C2 applied to X2.

Example: 'Alternative', 'greater'

Data Types: char

#### **ClassNames — Class names**

categorical vector | cell array of character vectors | character array | logical vector | numeric vector

Class names, specified as the comma-separated pair consisting of 'ClassNames' and a categorical or character array, logical or numeric vector, or cell array of character vectors. You must set `ClassNames` using the data type of  $Y$ .

If `ClassNames` is a character array, then each element must correspond to one *row* of the array.

Use `ClassNames` to:

- Specify the order of any input argument dimension that corresponds to class order. For example, use `ClassNames` to specify the order of the dimensions of `Cost`.

- Select a subset of classes for testing. For example, suppose that the set of all distinct class names in  $Y$  is `{'a', 'b', 'c'}`. To train and test models using observations from classes 'a' and 'c' only, specify `'ClassNames', {'a', 'c'}`.

The default is the set of all distinct class names in  $Y$ .

Example: `'ClassNames', {'b', 'g'}`

Data Types: `categorical | char | logical | single | double | cell`

### **Cost — Misclassification cost**

`square matrix | structure array`

Misclassification cost, specified as the comma-separated pair consisting of `'Cost'` and a square matrix or structure array. If you specify:

- If you specify the square matrix `Cost`, then `Cost(i, j)` is the cost of classifying a point into class  $j$  if its true class is  $i$ . That is, the rows correspond to the true class and the columns correspond to the predicted class. To specify the class order for the corresponding rows and columns of `Cost`, additionally specify the `ClassNames` name-value pair argument.
- If you specify the structure `S`, then `S` must have two fields:
  - `S.ClassNames`, which contains the class names as a variable of the same data type as  $Y$ . You can use this field to specify the order of the classes.
  - `S.ClassificationCosts`, which contains the cost matrix, with rows and columns ordered as in `S.ClassNames`

If you specify `Cost`, then `CompactClassificationSVM.compareHoldout` cannot conduct one-sided, exact, or mid- $p$  tests. You must also specify `'Alternative', 'unequal', 'Test', 'asymptotic'`. For cost-sensitive testing options, see the `CostTest` name-value pair argument.

It is a best practice to supply the same cost matrix used to train the classification models.

The default is `Cost(i, j) = 1` if  $i \neq j$ , and `Cost(i, j) = 0` if  $i = j$ .

Example: `'Cost', [0 1 2 ; 1 0 2; 2 2 0]`

Data Types: `double | single | struct`



**CostTest — Cost-sensitive test type**

'likelihood' (default) | 'chisquare'

Cost-sensitive test type, specified as the comma-separated pair consisting of 'CostTest' and 'chisquare' or 'likelihood'. Unless you specify a cost matrix using the Cost name-value pair argument, CompactClassificationSVM.compareHoldout ignores CostTest.

This table summarizes the available options for cost-sensitive testing.

Value	Asymptotic test type	Requirements
'chisquare'	Chi-square test	Optimization Toolbox license to implement quadprog
'likelihood'	Likelihood ratio test	None

For more details, see “Cost-Sensitive Testing” on page 32-6515.

Example: 'CostTest', 'chisquare'

Data Types: char

**Test — Test to conduct**

'asymptotic' | 'exact' | 'midp'

Test to conduct, specified as the comma-separated pair consisting of 'Test' and 'asymptotic', 'exact', and 'midp'. This table summarizes the available options for cost-insensitive testing.

Value	Description
'asymptotic'	Asymptotic McNemar test
'exact'	Exact-conditional McNemar test
'midp' (default)	Mid- <i>p</i> -value McNemar test

For more details, see “McNemar Tests” on page 32-6517.

For cost-sensitive testing, Test must be 'asymptotic'. When you specify the Cost name-value pair argument, and choose a cost-sensitive test using the CostTest name-value pair argument, 'asymptotic' is the default.

Example: 'Test', 'asymptotic'

Data Types: `char`

---

**Note** NaNs, `<undefined>` values, and empty character vectors ( `' '` ) indicate missing values. `CompactClassificationSVM.compareHoldout`:

- Removes missing values in `Y` and the corresponding rows of `X1` and `X2`
  - Predicts classes whether `X1` and `X2` have missing observations.
- 

## Output Arguments

### **h** — Hypothesis test result

1 | 0

Hypothesis test result, returned as a logical value.

`h = 1` indicates the rejection of the null hypothesis at the Alpha significance level.

`h = 0` indicates failure to reject the null hypothesis at the Alpha significance level.

### **p** — *p*-value

scalar in the interval [0,1]

*p*-value of the test, returned as a scalar in the interval [0,1]. *p* is the probability that a random test statistic is at least as extreme as the observed test statistic, given that the null hypothesis is true.

`CompactClassificationSVM.compareHoldout` estimates *p* using the distribution of the test statistic, which varies with the type of test. For details on test statistics derived from the available variants of the McNemar test, see “McNemar Tests” on page 32-6517. For details on test statistics derived from cost-sensitive tests, see “Cost-Sensitive Testing” on page 32-6515.

### **e1** — Classification loss

scalar

Classification loss on page 32-1058, returned as a scalar. `e1` summarizes the accuracy of the first set of class labels predicting the true class labels (`Y`).

`CompactClassificationSVM.compareHoldout` applies the first test-set predictor data ( $X_1$ ) to the first classification model ( $C_1$ ) to estimate the first set of class labels. Then, the function compares the estimated labels to  $Y$  to obtain the classification loss.

For cost-insensitive testing,  $e_1$  is the misclassification rate. That is,  $e_1$  is the proportion of misclassified observations, which is a scalar in the interval  $[0,1]$ .

For cost-sensitive testing,  $e_1$  is the misclassification cost. That is,  $e_1$  is the weighted average of the misclassification costs, in which the weights are the respective estimated proportions of misclassified observations.

### **$e_2$ — Classification loss**

scalar

Classification loss on page 32-1058, returned as a scalar.  $e_2$  summarizes the accuracy of the second set of class labels predicting the true class labels ( $Y$ ).

`CompactClassificationSVM.compareHoldout` applies the second test-set predictor data ( $X_2$ ) to the second classification model ( $C_2$ ) to estimate the second set of class labels. Then the function compares the estimated labels to  $Y$  to obtain the classification loss.

For cost-insensitive testing,  $e_2$  is the misclassification rate. That is,  $e_2$  is the proportion of misclassified observations, which is a scalar in the interval  $[0,1]$ .

For cost-sensitive testing,  $e_2$  is the misclassification cost. That is,  $e_2$  is the weighted average of the misclassification costs, in which the weights are the respective estimated proportions of misclassified observations.

## Examples

### **Compare Accuracies of Two Different Classification Models**

Train two classification models using different algorithms. Conduct a statistical test comparing the misclassification rates of the two models on a held-out set.

Load the `ionosphere` data set.

```
load ionosphere;
```

Create a partition that evenly splits the data into training and testing sets.

```
rng(1); % For reproducibility
CVP = cvpartition(Y, 'holdout', 0.5);
idxTrain = training(CVP); % Training-set indices
idxTest = test(CVP); % Test-set indices
```

CVP is a cross-validation partition object that specifies the training and test sets.

Train an SVM model and an ensemble of 100 bagged classification trees. For the SVM model, specify to use the radial basis function kernel and a heuristic procedure to determine the kernel scale.

```
C1 = fitcsvm(X(idxTrain,:), Y(idxTrain), 'Standardize', true, ...
    'KernelFunction', 'RBF', 'KernelScale', 'auto');
C2 = fitensemble(X(idxTrain,:), Y(idxTrain), 'Bag', 100, 'Tree', ...
    'Type', 'classification');
```

C1 is a trained ClassificationSVM model. C2 is a trained ClassificationBaggedEnsemble model.

Test whether the two models have equal predictive accuracies. Use the same test-set predictor data for each model.

```
h = compareHoldout(C1, C2, X(idxTest,:), X(idxTest,:), Y(idxTest))
```

```
h =
    logical
     0
```

h = 0 indicates to not reject the null hypothesis that the two models have equal predictive accuracies.

### Assess Whether One Classification Model Classifies Better Than Another

Train two classification models using the same algorithm, but adjust a hyperparameter to make the algorithm more complex. Conduct a statistical test to assess whether the simpler model has better accuracy in held-out data than the more complex model.

Load the ionosphere data set.

```
load ionosphere;
```

Create a partition that evenly splits the data into training and testing sets.

```
rng(1); % For reproducibility
CVP = cvpartition(Y, 'holdout', 0.5);
idxTrain = training(CVP); % Training-set indices
idxTest = test(CVP); % Test-set indices
```

CVP is a cross-validation partition object that specifies the training and test sets.

Train two SVM models: one that uses a linear kernel (the default for binary classification) and one that uses the radial basis function kernel. Use the default kernel scale of 1.

```
C1 = fitcsvm(X(idxTrain,:), Y(idxTrain), 'Standardize', true);
C2 = fitcsvm(X(idxTrain,:), Y(idxTrain), 'Standardize', true, ...
    'KernelFunction', 'RBF');
```

C1 and C2 are trained ClassificationSVM models.

Test the null hypothesis that the simpler model (C1) is at most as accurate as the more complex model (C2). Because the test-set size is large, conduct the asymptotic McNemar test, and compare the results with the mid- $p$ -value test (the cost-insensitive testing default). Request to return  $p$ -values and misclassification rates.

```
Asymp = zeros(4,1); % Preallocation
MidP = zeros(4,1);

[Asymp(1), Asymp(2), Asymp(3), Asymp(4)] = compareHoldout(C1, C2, ...
    X(idxTest,:), X(idxTest,:), Y(idxTest), 'Alternative', 'greater', ...
    'Test', 'asymptotic');
[MidP(1), MidP(2), MidP(3), MidP(4)] = compareHoldout(C1, C2, ...
    X(idxTest,:), X(idxTest,:), Y(idxTest), 'Alternative', 'greater');
table(Asymp, MidP, 'RowNames', {'h' 'p' 'e1' 'e2'})
```

```
ans =
```

```
4x2 table
```

	Asymp	MidP
h		
p		
e1		
e2		

h	1	1
p	7.2801e-09	2.7649e-10
e1	0.13714	0.13714
e2	0.33143	0.33143

The  $p$ -value is close to zero for both tests, which indicates strong evidence to reject the null hypothesis that the simpler model is less accurate than the more complex model. No matter what test you specify, `compareHoldout` returns the same type of misclassification measure for both models.

### Conduct a Cost-Sensitive Comparison of Two Classification Models

For data sets with imbalanced class representations, or for data sets with imbalanced false-positive and false-negative costs, you can statistically compare the predictive performances of two classification models by including a cost matrix in the analysis.

Load the `arrhythmia` data set. Determine the class representations in the data.

```
load arrhythmia;  
Y = categorical(Y);  
tabulate(Y);
```

Value	Count	Percent
1	245	54.20%
2	44	9.73%
3	15	3.32%
4	15	3.32%
5	13	2.88%
6	25	5.53%
7	3	0.66%
8	2	0.44%
9	9	1.99%
10	50	11.06%
14	4	0.88%
15	5	1.11%
16	22	4.87%

There are 16 classes, however some are not represented in the data set. Most observations are classified as not having arrhythmia (class 1). To summarize, the data set is highly discrete with imbalanced classes.

Combine all observations with arrhythmia (classes 2 through 15) into one class. Remove those observations with unknown arrhythmia status from the data set.

```
Y = Y(Y ~= '16');
Y(Y ~= '1') = '2';
X = X(Y ~= '16', :);
```

Create a partition that evenly splits the data into training and testing sets.

```
rng(1); % For reproducibility
CVP = cvpartition(Y, 'holdout', 0.5);
idxTrain = training(CVP); % Training-set indices
idxTest = test(CVP); % Test-set indices
```

CVP is a cross-validation partition object that specifies the training and test sets.

Create a cost matrix such that misclassifying an arrhythmic patient into the no arrhythmia class is five times worse than misclassifying a patient without arrhythmia into the arrhythmia class. Classifying correctly incurs no cost. The rows indicate the true class and the columns indicate the predicted class. When conducting a cost-sensitive analysis, it is a good practice to specify the order of the classes.

```
cost = [0 1; 5 0];
ClassNames = categorical([2 1]);
```

Train two boosting ensembles of 50 classification trees, one that uses AdaBoostM1 and the other that uses LogitBoost. Because there are missing values, specify to use surrogate splits. Train the models using the cost matrix.

```
t = templateTree('Surrogate', 'on');
numTrees = 50;
C1 = fitensemble(X(idxTrain, :), Y(idxTrain), 'AdaBoostM1', numTrees, t, ...
    'Cost', cost);
C2 = fitensemble(X(idxTrain, :), Y(idxTrain), 'LogitBoost', numTrees, t, ...
    'Cost', cost);
```

C1 and C2 are trained ClassificationEnsemble models.

Test whether the AdaBoostM1 ensemble (C1) and the LogitBoost ensemble (C2) have equal predictive accuracy. Supply the cost matrix. Conduct the asymptotic, likelihood ratio, cost-sensitive test (the default when you pass in a cost matrix). Request to return  $p$ -values and misclassification costs.

```
[h,p,e1,e2] = compareHoldout(C1,C2,X(idxTest,:),X(idxTest,:),Y(idxTest),...
    'Cost',cost)

h =

    logical

     0

p =

    0.0743

e1 =

    1.3581

e2 =

    1.6186
```

`h = 0` indicates to not reject the null hypothesis that the two models have equal predictive accuracies.

### Select Features Using Statistical Accuracy Comparison

Reduce classification model complexity by selecting a subset of predictor variables (features) from a larger set. Then, statistically compare the out-of-sample accuracy between the two models.

Load the `ionosphere` data set.

```
load ionosphere;
```

Create a partition that evenly splits the data into training and testing sets.

```
rng(1); % For reproducibility
CVP = cvpartition(Y,'holdout',0.5);
```



```

idxTrain = training(CVP);           % Training-set indices
idxTest = test(CVP);               % Test-set indices

```

CVP is a cross-validation partition object that specifies the training and test sets.

Train an ensemble of 100 boosted classification trees using AdaBoostM1 and the entire set of predictors. Inspect the importance measure for each predictor.

```

nTrees = 100;
C2 = fitensemble(X(idxTrain,:),Y(idxTrain),'AdaBoostM1',nTrees,'Tree');
predImp = predictorImportance(C2);

```

```

figure;
bar(predImp);
h = gca;
h.XTick = 1:2:h.XLim(2)
title('Predictor Importances');
xlabel('Predictor');
ylabel('Importance measure');

```

h =

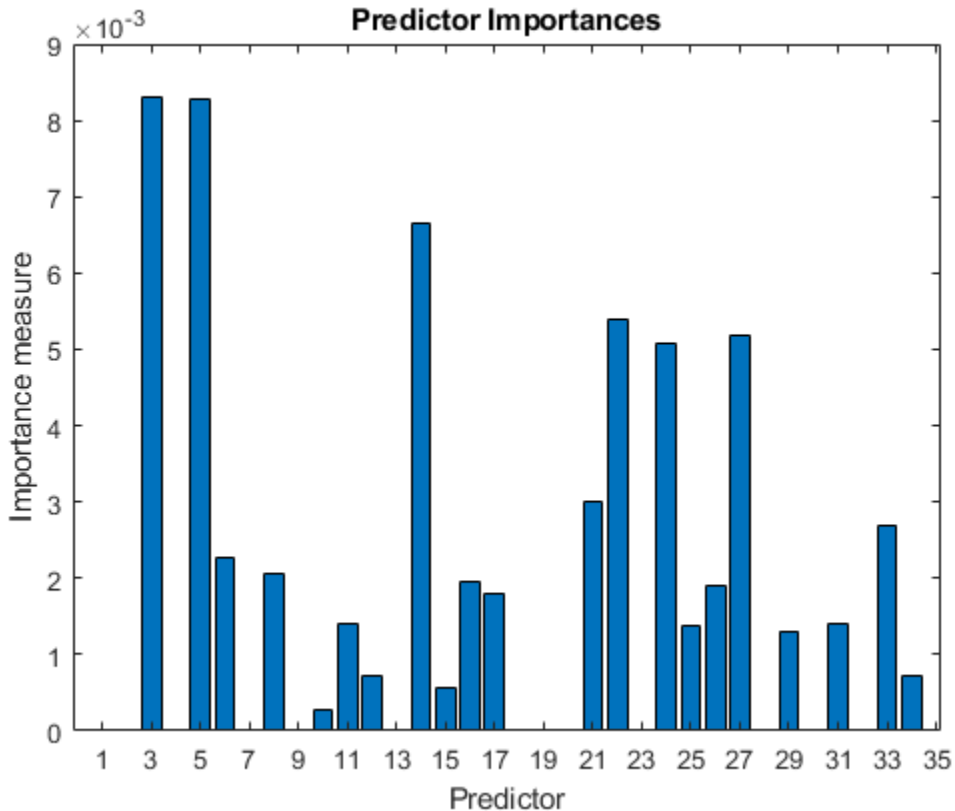
Axes with properties:

```

        XLim: [-0.2000 35.2000]
        YLim: [0 0.0090]
        XScale: 'linear'
        YScale: 'linear'
GridLineStyle: '-'
        Position: [0.1300 0.1100 0.7750 0.8150]
        Units: 'normalized'

```

Use GET to show all properties



Identify the top five predictors in terms of their importance.

```
[~,idxSort] = sort(predImp, 'descend');
idx5 = idxSort(1:5);
```

Train another ensemble of 100 boosted classification trees using AdaBoostM1 and the five predictors with the best importance.

```
C1 = fitensemble(X(idxTrain,idx5),Y(idxTrain),'AdaBoostM1',nTrees,...
    'Tree');
```

Test whether the two models have equal predictive accuracies. Specify the reduced test-set predictor data for C1 and the full test-set predictor data for C2.

```
[h,p,e1,e2] = compareHoldout(C1,C2,X(idxTest,idx5),X(idxTest,:),Y(idxTest))
```

```
h =  
  logical  
  0  
  
p =  
  0.7744  
  
e1 =  
  0.0914  
  
e2 =  
  0.0857
```

$h = 0$  indicates to not reject the null hypothesis that the two models have equal predictive accuracies. This result favors the simpler ensemble, C1.

## Definitions

### Cost-Sensitive Testing

Conduct cost-sensitive testing when the cost of misclassification is imbalanced. When conducting a cost-sensitive analysis, you can account for the cost imbalance in training the classification models, and then in statistically comparing them.

If the cost of misclassification is imbalanced, then the misclassification rate tends to be a poorly performing classification loss on page 32-6520. Use misclassification cost instead to compare classification models.

Misclassification costs are often unbalanced in applications. For example, consider classifying subjects based on a set of predictors into two categories: healthy and sick. Misclassifying a sick subject as healthy poses a danger to the subject's life. However,

misclassifying a healthy subject as sick can cause some inconvenience, but does not pose any danger. In this situation, you assign misclassification costs such that misclassifying a sick subject as healthy is more costly than misclassifying a healthy subject as sick.

The definitions that follow summarize the cost-sensitive tests. In the definitions:

- $n_{ijk}$  and  $\hat{\pi}_{ijk}$  are the number and estimated proportion of test-sample observations with true class  $k$  that the first classification model assigns label  $i$ . The second classification model assigns label  $j$ . The unknown, true value of  $\hat{\pi}_{ijk}$  is  $\pi_{ijk}$ . The test-set sample size is  $\sum_{i,j,k} n_{ijk} = n_{test}$ .  $\sum_{i,j,k} \pi_{ijk} = \sum_{i,j,k} \pi_{ijk} = 1$ .
- $c_{ij}$  is the relative cost of assigning label  $j$  to an observation with true class  $i$ .  $c_{ii} = 0$ ,  $c_{ij} \geq 0$ , and, for at least one  $(i,j)$  pair,  $c_{ij} > 0$ .

- All subscripts take on integer values from 1 through  $K$ , which is the number of classes.
- The expected difference in the misclassification costs of the two classification models is

$$\delta = \sum_{i=1}^K \sum_{j=1}^K \sum_{k=1}^K (c_{ki} - c_{kj}) \pi_{ijk}.$$

- The hypothesis test is

$$H_0 : \delta = 0$$

$$H_1 : \delta \neq 0$$

The available cost-sensitive tests are appropriate for two-tailed testing.

Available, asymptotic tests that address imbalanced costs are a chi-square test and a likelihood ratio test.

- Chi-square test — The chi-square test statistic is based on the Pearson and Neyman chi-square test statistics, but with a Laplace correction factor to account for any  $n_{ijk} = 0$ . The test statistic is

$$t_{\chi^2}^* = \sum_{i \neq j} \sum_k \frac{\left( n_{ijk} + 1 - (n_{test} + K^3) \hat{\pi}_{ijk}^{(1)} \right)^2}{n_{ijk} + 1}.$$

If  $1 - F_{\chi^2}(t_{\chi^2}^*; 1) < \alpha$ , then reject  $H_0$ .

•

$\hat{\pi}_{ijk}^{(1)}$  are estimated by minimizing  $t_{\chi^2}^*$  under the constraint that  $\delta = 0$ .

•

$F_{\chi^2}(x; 1)$  is the  $\chi^2$  C.D.F. with one degree of freedom evaluated at  $x$ .

- Likelihood ratio test — The likelihood ratio test is based on  $N_{ijk}$ , which are binomial random variables having sample size  $n_{test}$  and success probability  $\pi_{ijk}$ . They represent the random number of observations with true class  $k$  that the first classification model assigns label  $i$ . The second classification model assigns label  $j$ . Jointly, their distribution is multinomial.

The test statistic is

$$t_{LRT}^* = 2 \log \left[ \frac{P\left(\bigcap_{i,j,k} N_{ijk} = n_{ijk}; n_{test}, \pi_{ijk} = \pi_{ijk}^{(2)}\right)}{P\left(\bigcap_{i,j,k} N_{ijk} = n_{ijk}; n_{test}, \pi_{ijk} = \pi_{ijk}^{(3)}\right)} \right].$$

If  $1 - F_{\chi^2}(t_{LRT}^*; 1) < \alpha$ , then reject  $H_0$ .

•

$\hat{\pi}_{ijk}^{(2)} = \frac{n_{ijk}}{n_{test}}$  is the unrestricted MLE of  $\pi_{ijk}$ .

•

$\hat{\pi}_{ijk}^{(3)} = \frac{n_{ijk}}{n_{test} + \lambda(c_{ki} - c_{kj})}$  is the MLE under the null hypothesis that  $\delta = 0$ .  $\lambda$  is the solution to

$$\sum_{i,j,k} \frac{n_{ijk}(c_{ki} - c_{kj})}{n_{test} + \lambda(c_{ki} - c_{kj})} = 0.$$

•

$F_{\chi^2}(x; 1)$  is the  $\chi^2$  C.D.F. with one degree of freedom evaluated at  $x$ .

## McNemar Tests

McNemar Tests are hypothesis tests that compare two population proportions while addressing the issues resulting from two dependent, matched-pair samples.

One way to compare the predictive accuracies of two classification models is:

- 1 Partition the data into training and test sets.
- 2 Train both classification models using the training set.
- 3 Predict class labels using the test set.
- 4 Summarize the results in a two-by-two table resembling this figure.

		Model 2		
		Correct	Incorrect	
Model 1	Correct	$n_{11}$	$n_{12}$	$n_{1\bullet}$
	Incorrect	$n_{21}$	$n_{22}$	$n_{2\bullet}$
		$n_{\bullet 1}$	$n_{\bullet 2}$	$n_{test}$

$n_{ii}$  are the number of concordant pairs, that is, the number of observations that both models classify the same way (correctly or incorrectly).  $n_{ij}$ ,  $i \neq j$ , are the number of discordant pairs, that is, the number of observations that models classify differently (correctly or incorrectly).

The misclassification rates for Models 1 and 2 are

$$\hat{\pi}_{2\bullet} = n_{2\bullet} / n$$

and  $\hat{\pi}_{\bullet 2} = n_{\bullet 2} / n$ , respectively. A two-sided test for comparing the accuracy of the two models is

$$H_0 : \pi_{\bullet 2} = \pi_{2\bullet}$$

$$H_1 : \pi_{\bullet 2} \neq \pi_{2\bullet}$$

The null hypothesis suggests that the population exhibits marginal homogeneity, which reduces the null hypothesis to  $H_0 : \pi_{12} = \pi_{21}$ . Also, under the null hypothesis,  $N_{12} \sim \text{Binomial}(n_{12} + n_{21}, 0.5)$  [1].

These facts are the basis for these, available McNemar test variants: the asymptotic, exact-conditional and mid-p-value McNemar tests. The definitions that follow summarize the available variants.

- Asymptotic — The asymptotic McNemar test statistics and rejection regions (for significance-level  $\alpha$ ) are:

- For one-sided tests, the test statistic

$$t_{a1}^* = \frac{n_{12} - n_{21}}{\sqrt{n_{12} + n_{21}}}.$$

If  $1 - \Phi\left(|t_1^*|\right) < \alpha$ , where  $\Phi$  is the standard Gaussian C.D.F., then reject  $H_0$ .

- For two-sided tests, the test statistic

$$t_{a2}^* = \frac{(n_{12} - n_{21})^2}{n_{12} + n_{21}}.$$

If  $1 - F_{\chi^2}\left(t_2^*; m\right) < \alpha$ , where  $F_{\chi^2}(x; m)$  is the  $\chi_m^2$  C.D.F. evaluated at  $x$ , then reject  $H_0$ .

This variant requires large-sample theory, specifically, the Gaussian approximation to the binomial distribution. Therefore:

- The total number of discordant pairs,  $n_d = n_{12} + n_{21}$  must be greater than 10 ([1], Ch. 10.1.4).
- In general, asymptotic tests do not guarantee nominal coverage. The observed probability of falsely rejecting the null hypothesis can exceed  $\alpha$ . Simulation studies in [18] suggest this, but the asymptotic McNemar test performs well in terms of statistical power.
- Exact Conditional — The exact-conditional McNemar test statistics and rejection regions (for significance-level  $\alpha$ ) are ([36], [38]):

- For one-sided tests, the test statistic

$$t_1^* = n_{12}$$

If  $F_{\text{Bin}}\left(t_1^*; n_d, 0.5\right) < \alpha$ , where  $F_{\text{Bin}}(x; n, p)$  is the binomial C.D.F. with sample size  $n$  and success probability  $p$  evaluated at  $x$ , then reject  $H_0$ .

- For two-sided tests, the test statistic

$$t_2^* = \min(n_{12}, n_{21})$$

If  $F_{\text{Bin}}(t_2^*; n_d, 0.5) < \alpha / 2$ , then reject  $H_0$ .

The exact conditional test always attains nominal coverage. Simulation studies in [18] suggest that the test is conservative, and then show that the test lacks statistical power compared to other variants. For small or highly discrete test samples, consider using the mid- $p$ -value test ([1], Ch. 3.6.3). For details, see `Test` and “McNemar Tests” on page 32-6517.

- Mid- $p$ -value test — The mid- $p$ -value McNemar test statistics and rejection regions (for significance-level  $\alpha$ ) are ([32]):

- For one-sided tests, the test statistic

$$t_1^* = n_{12}$$

If  $F_{\text{Bin}}(t_1^* - 1; n_{12} + n_{21}, 0.5) + 0.5f_{\text{Bin}}(t_1^*; n_{12} + n_{21}, 0.5) < \alpha$ , where  $F_{\text{Bin}}(x; n, p)$  and  $f_{\text{Bin}}(x; n, p)$  are the binomial C.D.F. and P.D.F, respectively, with sample size  $n$  and success probability  $p$  evaluated at  $x$ , then reject  $H_0$ .

- For two-sided tests, the test statistic

$$t_2^* = \min(n_{12}, n_{21})$$

If  $F_{\text{Bin}}(t_2^* - 1; n_{12} + n_{21} - 1, 0.5) + 0.5f_{\text{Bin}}(t_2^*; n_{12} + n_{21}, 0.5) < \alpha / 2$ , then reject  $H_0$ .

The mid- $p$ -value test addresses the over-conservative behavior of the exact conditional test. The simulation studies in [18] demonstrate that this test attains nominal coverage, and has good statistical power.

## Classification Loss

Classification losses indicate the accuracy of a classification model or set of predicted labels. Two classification losses are misclassification rate and cost.

`CompactClassificationSVM.compareHoldout` returns the classification losses (see `e1` and `e2`) under the alternative hypothesis (i.e., the unrestricted classification losses).  $n_{ijk}$  is the number of test-sample observations with true class  $k$  that the first



classification model assigns label  $i$  and the second classification model assigns label  $j$ ,

and the corresponding estimated proportion is  $\hat{\pi}_{ijk} = \frac{n_{ijk}}{n_{test}}$ . The test-set sample size is

$\sum_{i,j,k} n_{ijk} = n_{test}$ . The indices are taken from 1 through  $K$ , the number of classes.

- Misclassification rate, or classification error, is a scalar in the interval  $[0,1]$  representing the proportion of misclassified observations. That is, the misclassification rate for the first classification model is

$$e_1 = \sum_{j=1}^K \sum_{k=1}^K \sum_{i \neq k} \hat{\pi}_{ijk}.$$

For the misclassification rate of the second classification model ( $e_2$ ), switch the indices  $i$  and  $j$  in the formula.

Classification accuracy decreases as the misclassification rate increases to 1.

- Misclassification cost is a nonnegative scalar and is a measure of classification quality relative to the values the specified cost matrix elements. Its interpretation depends on the specified costs of misclassification. Misclassification cost is the weighted average of the costs of misclassification (specified in a cost matrix,  $C$ ) in which the weights are the respective, estimated proportions of misclassified observations. The misclassification cost for the first classification model is

$$e_1 = \sum_{j=1}^K \sum_{k=1}^K \sum_{i \neq k} \hat{\pi}_{ijk} c_{ki},$$

where  $c_{kj}$  is the cost of classifying an observation into class  $j$  if its true class is  $k$ . For the misclassification cost of the second classification model ( $e_2$ ), switch the indices  $i$  and  $j$  in the formula.

In general, for a fixed cost matrix, classification accuracy decreases as misclassification cost increases.

## Tips

- One way to perform cost-insensitive feature selection is:
  - 1 Train the first classification model ( $C_1$ ) using the full predictor set.

- 2 Train the second classification model (C2) using the reduced predictor set.
- 3 Specify X1 as the full, test-set predictor data and X2 as the reduced test-set predictor data.
- 4 Enter `compareHoldout(C1,C2,X1,X2,'Alternative','less')`. If `compareHoldout` returns 1, then there is enough evidence to suggest that the classification model that uses fewer predictors performs better than the model that uses the full predictor set.

Alternatively, you can assess whether there is a significant difference between the accuracies of the two models. To perform this assessment, remove the 'Alternative', 'greater' specification in step 4. `compareHoldout` conducts a two-sided test, and `h = 0` indicates that there is not enough evidence to suggest a difference in the accuracy of the two models.

- Cost-sensitive tests perform numerical optimization, which requires additional computational resources. The likelihood ratio test conducts numerical optimization indirectly by finding the root of a Lagrange multiplier in an interval. For some data sets, if the root lies close to the boundaries of the interval, then the method can fail. Therefore, if you have an Optimization Toolbox license, consider conducting the cost-sensitive chi-square test instead. For more details, see `CostTest` and “Cost-Sensitive Testing” on page 32-1053.

## Alternatives

To directly compare the accuracy of two sets of class labels in predicting a set of true class labels, use `testcholdout`.

## References

- [1] Agresti, A. *Categorical Data Analysis*, 2nd Ed. John Wiley & Sons, Inc.: Hoboken, NJ, 2002.
- [2] Fagerlan, M.W., S Lydersen, P. Laake. “The McNemar Test for Binary Matched-Pairs Data: Mid-p and Asymptotic Are Better Than Exact Conditional.” *BMC Medical Research Methodology*. Vol. 13, 2013, pp. 1–8.
- [3] Lancaster, H.O. “Significance Tests in Discrete Distributions.” *JASA*, Vol. 56, Number 294, 1961, pp. 223–234.

- [4] McNemar, Q. “Note on the Sampling Error of the Difference Between Correlated Proportions or Percentages.” *Psychometrika*, Vol. 12, Number 2, 1947, pp. 153–157.
- [5] Mosteller, F. “Some Statistical Problems in Measuring the Subjective Response to Drugs.” *Biometrics*, Vol. 8, Number 3, 1952, pp. 220–226.

## See Also

`fitcsvm` | `predict` | `testcholdout` | `testckfold`

## Topics

“Hypothesis Tests”

**Introduced in R2015a**

## compareHoldout

**Class:** CompactClassificationTree

Compare accuracies of two classification models using new data

`compareHoldout` statistically assesses the accuracies of two classification models. The function first compares their predicted labels against the true labels, and then it detects whether the difference between the misclassification rates is statistically significant.

You can assess whether the accuracies of the classification models are different, or whether one classification model performs better than another. `compareHoldout` can conduct several McNemar test on page 32-1055 variations, including the asymptotic test, the exact-conditional test, and the mid- $p$ -value test. For cost-sensitive assessment on page 32-1053, available tests include a chi-square test (requires an Optimization Toolbox license) and a likelihood ratio test.

### Syntax

```
h = compareHoldout(C1,C2,T1,T2,ResponseVarName)
```

```
h = compareHoldout(C1,C2,T1,T2,Y)
```

```
h = compareHoldout( ____,Name,Value)
```

```
h = compareHoldout(C1,C2,X1,X2,Y)
```

```
h = compareHoldout(C1,C2,X1,X2,Y,Name,Value)
```

```
[h,p,e1,e2] = compareHoldout( ____,Name,Value)
```

```
[h,p,e1,e2] = compareHoldout( ____,Name,Value)
```

### Description

`h = compareHoldout(C1,C2,T1,T2,ResponseVarName)` returns the test decision from testing the null hypothesis that the trained classification models `C1` and `C2` have equal accuracy for predicting the true class labels `ResponseVarName`. The alternative hypothesis is that the labels have unequal accuracy.

$h = 1$  indicates to reject the null hypothesis at the 5% significance level.  $h = 0$  indicates to not reject the null hypothesis at 5% level.

The first classification model `C1` uses predictor data in table `T1` and `C2` uses `T2`. The software conducts the mid- $p$ -value McNemar test to compare the accuracies.

Examples of tests you can conduct include:

- Compare the accuracies of a simple classification model and a model that is more complex by passing the same set of predictor data (i.e.,  $X1 = X2$ ).
- Compare the accuracies of two perhaps different models using two perhaps different sets of predictors.
- Perform various types of feature selection. For example, you can compare the accuracy of a model trained using a set of predictors to the accuracy of one trained on a subset or different set of those predictors. You can arbitrarily choose the set of predictors, or use a feature selection technique like PCA or sequential feature selection (see `pca` and `sequentialfs`).

`h = compareHoldout(C1,C2,T1,T2,Y)` returns the test decision from testing the null hypothesis that the trained classification models `C1` and `C2` have equal accuracy for predicting the true class labels `Y`. The alternative hypothesis is that the labels have unequal accuracy.

The first classification model `C1` uses predictor data on table `T1` and `C2` uses `T2`. The software conducts the mid- $p$ -value McNemar test to compare the accuracies.

`h = compareHoldout( ____, Name, Value)` returns the result of the hypothesis test with additional options specified by one or more `Name, Value` pair arguments. For example, you can specify the type of alternative hypothesis, and the type of test, or you can supply a cost matrix.

`h = compareHoldout(C1,C2,X1,X2,Y)` returns the test decision from testing the null hypothesis that the trained classification models `C1` and `C2` have equal accuracy for predicting the true class labels `Y`. The alternative hypothesis is that the labels have unequal accuracy.

The first classification model `C1` uses predictor data `X1` and `C2` uses `X2`. The software conducts the mid- $p$ -value McNemar test to compare the accuracies.

`h = compareHoldout(C1,C2,X1,X2,Y, Name, Value)` returns the result of the hypothesis test with additional options specified by one or more `Name, Value` pair

arguments. For example, you can specify the type of alternative hypothesis, and the type of test, or you can supply a cost matrix.

`[h,p,e1,e2] = compareHoldout( ___ )` returns the  $p$ -value for the hypothesis test ( $p$ ) and the respective classification losses on page 32-1085 of each set of predicted class labels ( $e1$  and  $e2$ ) using any of the input arguments in the previous syntaxes.

`[h,p,e1,e2] = compareHoldout( ___ )` returns the  $p$ -value for the hypothesis test ( $p$ ) and the respective classification losses on page 32-1085 of each set of predicted class labels ( $e1$  and  $e2$ ) using any of the input arguments in the previous syntaxes.

## Input Arguments

### c1 — Trained classification tree

`ClassificationTree` model object | `CompactClassificationTree` model object

Trained classification tree, specified as a `ClassificationTree` or `CompactClassificationTree` model object. That is, `C1` is a trained classification model returned by `fitctree` or `compact`.

### c2 — Trained classification model

trained classification model object | trained, compact classification model object

Trained classification model, specified as any trained or compact classification model object described in this table.

Trained Model Type	Model Object	Returned By
Classification tree	<code>ClassificationTree</code>	<code>fitctree</code>
Discriminant analysis	<code>ClassificationDiscriminant</code>	<code>fitcdiscr</code>
Ensemble of bagged classification models	<code>ClassificationBaggedEnsemble</code>	<code>fitcensemble</code>
Ensemble of classification models	<code>ClassificationEnsemble</code>	<code>fitcensemble</code>
Multiclass, error-correcting output codes (ECOC)	<code>ClassificationECOC</code>	<code>fitcecoc</code>
$k$ NN	<code>ClassificationKNN</code>	<code>fitcknn</code>

Trained Model Type	Model Object	Returned By
Naive Bayes	ClassificationNaiveBayes	fitcnb
Support vector machine (SVM)	ClassificationSVM	fitcsvm
Compact discriminant analysis	CompactClassificationDiscriminant	compact
Compact ECOC	CompactClassificationECOC	compact
Compact ensemble of classification models	CompactClassificationEnsemble	compact
Compact naive Bayes	CompactClassificationNaiveBayes	compact
Compact SVM	CompactClassificationSVM	compact
Compact classification tree	CompactClassificationTree	compact

### T1 — Test-set predictor table for first classification model

table

Test-set predictor table for the first classification model, C1, specified as a table. Each row of T1 corresponds to one observation, and each column corresponds to one predictor variable. Optionally, T1 can contain an additional column for the response variable. T1 must contain all the predictors used to train C1. Multi-column variables and cell arrays other than cell arrays of character vectors are not allowed.

T1 and T2 must have the same number of rows, and must have the same response values. If T1 and T2 contain the response variable used to train C1 and C2, then, you do not need to specify Y.

Data Types: table

### T2 — Test-set predictor table for second classification model

table

Test-set predictor data for the second classification model, C2, specified as a table. Each row of T2 corresponds to one observation, and each column corresponds to one predictor variable. Optionally, T2 can contain an additional column for the response variable. T2 must contain all the predictors used to train C2. Multi-column variables and cell arrays other than cell arrays of character vectors are not allowed.

T1 and T2 must have the same number of rows and the same response values. If T1 and T2 contain the response variable used to train C1 and C2, then, you do not need to specify Y.

Data Types: `table`

**x1 — Test-set predictor data for first classification model**

numeric matrix

Test-set predictor data for the first classification model, C1, specified as a numeric matrix.

Each row of X1 corresponds to one observation (also known as an instance or example), and each column corresponds to one variable (also known as a predictor or feature). The variables used to train C1 must compose X1.

The number of rows in X1 and X2 must equal the length of Y.

Data Types: `double` | `single`

**x2 — Test-set predictor data for second classification model**

numeric matrix

Test-set predictor data for the second classification model, C2, specified as a numeric matrix.

Each row of X2 corresponds to one observation (also known as an instance or example), and each column corresponds to one variable (also known as a predictor or feature). The variables used to train C2 must compose X2.

The number of rows in X2 and X1 must equal the length of Y.

Data Types: `double` | `single`

**ResponseVarName — Response variable name**

name of a variable in T1 and T2

Response variable name, specified as the name of a variable in T1 and T2. If T1 and T2 contain the response variable used to train C1 and C2, then you do not need to specify ResponseVarName.

If you specify ResponseVarName, then you must do so as a character vector. For example, if the response variable is stored as T1.Response, then specify it as



'Response'. Otherwise, the software treats all columns of T1 and T2, including Response, as predictors.

The response variable must be a categorical or character array, logical or numeric vector, or cell array of character vectors. If the response variable is a character array, then each element must correspond to one row of the array.

### **y** — True class labels

categorical array | character array | logical vector | vector of numeric values | cell array of character vectors | character vector

True class labels, specified as a categorical or character array, a logical or numeric vector, a cell array of character vectors, or a character vector.

If you specify any array except a character vector, then

`CompactClassificationTree.compareHoldout` treats all variables or columns in X1 and X2 as predictor variables.

If you specify a character vector, then X1 and X2 must be tables, their response variables must have the same name and values, and Y must be the common name. For example, if X1.Labels and X2.Labels are the response variables, then Y must be 'Labels' and X1.Labels and X2.Labels must be equivalent.

If Y is a character array, then each element must correspond to one row of the array.

The number of rows in X1 and X2 must equal the length of Y.

Data Types: categorical | char | logical | single | double | cell

## **Name-Value Pair Arguments**

Specify optional comma-separated pairs of Name, Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as Name1, Value1, . . . , NameN, ValueN.

Example: 'Alternative', 'greater', 'Test', 'asymptotic', 'Cost', [0 2; 1 0] specifies to test whether the first set of first predicted class labels is more accurate than the second set, to conduct the asymptotic McNemar test, and to penalize misclassifying observations with the true label `ClassNames{1}` twice as much as for misclassifying observations with the true label `ClassNames{2}`.

**Alpha — Hypothesis test significance level**

0.05 (default) | scalar value in the interval (0,1)

Hypothesis test significance level, specified as the comma-separated pair consisting of 'Alpha' and a scalar value in the interval (0,1).

Example: 'Alpha', 0.1

Data Types: single | double

**Alternative — Alternative hypothesis to assess**

'unequal' (default) | 'greater' | 'less'

Alternative hypothesis to assess, specified as the comma-separated pair consisting of 'Alternative' and one of these values listed in the table.

Value	Alternative hypothesis
'unequal' (default)	For predicting $Y$ , the set of predictions resulting from $C1$ applied to $X1$ and $C2$ applied to $X2$ have unequal accuracies.
'greater'	For predicting $Y$ , the set of predictions resulting from $C1$ applied to $X1$ is more accurate than $C2$ applied to $X2$ .
'less'	For predicting $Y$ , the set of predictions resulting from $C1$ applied to $X1$ is less accurate than $C2$ applied to $X2$ .

Example: 'Alternative', 'greater'

Data Types: char

**ClassNames — Class names**

categorical vector | cell array of character vectors | character array | logical vector | numeric vector

Class names, specified as the comma-separated pair consisting of 'ClassNames' and a categorical or character array, logical or numeric vector, or cell array of character vectors. You must set `ClassNames` using the data type of  $Y$ .

If `ClassNames` is a character array, then each element must correspond to one *row* of the array.

Use `ClassNames` to:

- Specify the order of any input argument dimension that corresponds to class order. For example, use `ClassNames` to specify the order of the dimensions of `Cost`.
- Select a subset of classes for testing. For example, suppose that the set of all distinct class names in  $Y$  is `{'a', 'b', 'c'}`. To train and test models using observations from classes 'a' and 'c' only, specify `'ClassNames', {'a', 'c'}`.

The default is the set of all distinct class names in  $Y$ .

Example: `'ClassNames', {'b', 'g'}`

Data Types: `categorical | char | logical | single | double | cell`

### **Cost — Misclassification cost**

`square matrix | structure array`

Misclassification cost, specified as the comma-separated pair consisting of `'Cost'` and a square matrix or structure array. If you specify:

- If you specify the square matrix `Cost`, then `Cost(i, j)` is the cost of classifying a point into class  $j$  if its true class is  $i$ . That is, the rows correspond to the true class and the columns correspond to the predicted class. To specify the class order for the corresponding rows and columns of `Cost`, additionally specify the `ClassNames` name-value pair argument.
- If you specify the structure `S`, then `S` must have two fields:
  - `S.ClassNames`, which contains the class names as a variable of the same data type as  $Y$ . You can use this field to specify the order of the classes.
  - `S.ClassificationCosts`, which contains the cost matrix, with rows and columns ordered as in `S.ClassNames`

If you specify `Cost`, then `CompactClassificationTree.compareHoldout` cannot conduct one-sided, exact, or mid- $p$  tests. You must also specify `'Alternative', 'unequal', 'Test', 'asymptotic'`. For cost-sensitive testing options, see the `CostTest` name-value pair argument.

It is a best practice to supply the same cost matrix used to train the classification models.

The default is  $\text{Cost}(i, j) = 1$  if  $i \neq j$ , and  $\text{Cost}(i, j) = 0$  if  $i = j$ .

Example: `'Cost', [0 1 2 ; 1 0 2; 2 2 0]`

Data Types: `double | single | struct`

**CostTest — Cost-sensitive test type**

'likelihood' (default) | 'chisquare'

Cost-sensitive test type, specified as the comma-separated pair consisting of 'CostTest' and 'chisquare' or 'likelihood'. Unless you specify a cost matrix using the Cost name-value pair argument, CompactClassificationTree.compareHoldout ignores CostTest.

This table summarizes the available options for cost-sensitive testing.

Value	Asymptotic test type	Requirements
'chisquare'	Chi-square test	Optimization Toolbox license to implement quadprog
'likelihood'	Likelihood ratio test	None

For more details, see “Cost-Sensitive Testing” on page 32-6515.

Example: 'CostTest', 'chisquare'

Data Types: char

**Test — Test to conduct**

'asymptotic' | 'exact' | 'midp'

Test to conduct, specified as the comma-separated pair consisting of 'Test' and 'asymptotic', 'exact', and 'midp'. This table summarizes the available options for cost-insensitive testing.

Value	Description
'asymptotic'	Asymptotic McNemar test
'exact'	Exact-conditional McNemar test
'midp' (default)	Mid- <i>p</i> -value McNemar test

For more details, see “McNemar Tests” on page 32-6517.

For cost-sensitive testing, Test must be 'asymptotic'. When you specify the Cost name-value pair argument, and choose a cost-sensitive test using the CostTest name-value pair argument, 'asymptotic' is the default.

Example: 'Test', 'asymptotic'

Data Types: char

---

**Note** NaNs, <undefined> values, and empty character vectors ( ' ' ) indicate missing values. `CompactClassificationTree.compareHoldout`:

- Removes missing values in  $Y$  and the corresponding rows of  $X1$  and  $X2$
  - Predicts classes whether  $X1$  and  $X2$  have missing observations.
- 

## Output Arguments

### **h** — Hypothesis test result

1 | 0

Hypothesis test result, returned as a logical value.

$h = 1$  indicates the rejection of the null hypothesis at the Alpha significance level.

$h = 0$  indicates failure to reject the null hypothesis at the Alpha significance level.

### **p** — *p*-value

scalar in the interval [0,1]

*p*-value of the test, returned as a scalar in the interval [0,1]. *p* is the probability that a random test statistic is at least as extreme as the observed test statistic, given that the null hypothesis is true.

`CompactClassificationTree.compareHoldout` estimates *p* using the distribution of the test statistic, which varies with the type of test. For details on test statistics derived from the available variants of the McNemar test, see “McNemar Tests” on page 32-6517. For details on test statistics derived from cost-sensitive tests, see “Cost-Sensitive Testing” on page 32-6515.

### **e1** — Classification loss

scalar

Classification loss on page 32-1058, returned as a scalar. *e1* summarizes the accuracy of the first set of class labels predicting the true class labels ( $Y$ ).

`CompactClassificationTree.compareHoldout` applies the first test-set predictor data ( $X_1$ ) to the first classification model ( $C_1$ ) to estimate the first set of class labels. Then, the function compares the estimated labels to  $Y$  to obtain the classification loss.

For cost-insensitive testing,  $e_1$  is the misclassification rate. That is,  $e_1$  is the proportion of misclassified observations, which is a scalar in the interval  $[0,1]$ .

For cost-sensitive testing,  $e_1$  is the misclassification cost. That is,  $e_1$  is the weighted average of the misclassification costs, in which the weights are the respective estimated proportions of misclassified observations.

### **$e_2$ — Classification loss**

scalar

Classification loss on page 32-1058, returned as a scalar.  $e_2$  summarizes the accuracy of the second set of class labels predicting the true class labels ( $Y$ ).

`CompactClassificationTree.compareHoldout` applies the second test-set predictor data ( $X_2$ ) to the second classification model ( $C_2$ ) to estimate the second set of class labels. Then the function compares the estimated labels to  $Y$  to obtain the classification loss.

For cost-insensitive testing,  $e_2$  is the misclassification rate. That is,  $e_2$  is the proportion of misclassified observations, which is a scalar in the interval  $[0,1]$ .

For cost-sensitive testing,  $e_2$  is the misclassification cost. That is,  $e_2$  is the weighted average of the misclassification costs, in which the weights are the respective estimated proportions of misclassified observations.

## Examples

### **Compare Accuracies of Two Different Classification Models**

Train two classification models using different algorithms. Conduct a statistical test comparing the misclassification rates of the two models on a held-out set.

Load the `ionosphere` data set.

```
load ionosphere;
```

Create a partition that evenly splits the data into training and testing sets.

```

rng(1); % For reproducibility
CVP = cvpartition(Y, 'holdout', 0.5);
idxTrain = training(CVP); % Training-set indices
idxTest = test(CVP); % Test-set indices

```

CVP is a cross-validation partition object that specifies the training and test sets.

Train an SVM model and an ensemble of 100 bagged classification trees. For the SVM model, specify to use the radial basis function kernel and a heuristic procedure to determine the kernel scale.

```

C1 = fitcsvm(X(idxTrain,:), Y(idxTrain), 'Standardize', true, ...
    'KernelFunction', 'RBF', 'KernelScale', 'auto');
C2 = fitensemble(X(idxTrain,:), Y(idxTrain), 'Bag', 100, 'Tree', ...
    'Type', 'classification');

```

C1 is a trained ClassificationSVM model. C2 is a trained ClassificationBaggedEnsemble model.

Test whether the two models have equal predictive accuracies. Use the same test-set predictor data for each model.

```

h = compareHoldout(C1, C2, X(idxTest,:), X(idxTest,:), Y(idxTest))

```

```

h =
    logical
     0

```

h = 0 indicates to not reject the null hypothesis that the two models have equal predictive accuracies.

### Assess Whether One Classification Model Classifies Better Than Another

Train two classification models using the same algorithm, but adjust a hyperparameter to make the algorithm more complex. Conduct a statistical test to assess whether the simpler model has better accuracy in held-out data than the more complex model.

Load the ionosphere data set.

```
load ionosphere;
```

Create a partition that evenly splits the data into training and testing sets.

```
rng(1); % For reproducibility
CVP = cvpartition(Y, 'holdout', 0.5);
idxTrain = training(CVP); % Training-set indices
idxTest = test(CVP); % Test-set indices
```

CVP is a cross-validation partition object that specifies the training and test sets.

Train two SVM models: one that uses a linear kernel (the default for binary classification) and one that uses the radial basis function kernel. Use the default kernel scale of 1.

```
C1 = fitcsvm(X(idxTrain,:), Y(idxTrain), 'Standardize', true);
C2 = fitcsvm(X(idxTrain,:), Y(idxTrain), 'Standardize', true, ...
    'KernelFunction', 'RBF');
```

C1 and C2 are trained ClassificationSVM models.

Test the null hypothesis that the simpler model (C1) is at most as accurate as the more complex model (C2). Because the test-set size is large, conduct the asymptotic McNemar test, and compare the results with the mid- $p$ -value test (the cost-insensitive testing default). Request to return  $p$ -values and misclassification rates.

```
Asymp = zeros(4,1); % Preallocation
MidP = zeros(4,1);

[Asymp(1), Asymp(2), Asymp(3), Asymp(4)] = compareHoldout(C1, C2, ...
    X(idxTest,:), X(idxTest,:), Y(idxTest), 'Alternative', 'greater', ...
    'Test', 'asymptotic');
[MidP(1), MidP(2), MidP(3), MidP(4)] = compareHoldout(C1, C2, ...
    X(idxTest,:), X(idxTest,:), Y(idxTest), 'Alternative', 'greater');
table(Asymp, MidP, 'RowNames', {'h' 'p' 'e1' 'e2'})
```

```
ans =
```

```
4x2 table
```

	Asymp	MidP



h	1	1
p	7.2801e-09	2.7649e-10
e1	0.13714	0.13714
e2	0.33143	0.33143

The  $p$ -value is close to zero for both tests, which indicates strong evidence to reject the null hypothesis that the simpler model is less accurate than the more complex model. No matter what test you specify, `compareHoldout` returns the same type of misclassification measure for both models.

### Conduct a Cost-Sensitive Comparison of Two Classification Models

For data sets with imbalanced class representations, or for data sets with imbalanced false-positive and false-negative costs, you can statistically compare the predictive performances of two classification models by including a cost matrix in the analysis.

Load the `arrhythmia` data set. Determine the class representations in the data.

```
load arrhythmia;
Y = categorical(Y);
tabulate(Y);
```

Value	Count	Percent
1	245	54.20%
2	44	9.73%
3	15	3.32%
4	15	3.32%
5	13	2.88%
6	25	5.53%
7	3	0.66%
8	2	0.44%
9	9	1.99%
10	50	11.06%
14	4	0.88%
15	5	1.11%
16	22	4.87%

There are 16 classes, however some are not represented in the data set. Most observations are classified as not having arrhythmia (class 1). To summarize, the data set is highly discrete with imbalanced classes.

Combine all observations with arrhythmia (classes 2 through 15) into one class. Remove those observations with unknown arrhythmia status from the data set.

```
Y = Y(Y ~= '16');  
Y(Y ~= '1') = '2';  
X = X(Y ~= '16', :);
```

Create a partition that evenly splits the data into training and testing sets.

```
rng(1); % For reproducibility  
CVP = cvpartition(Y, 'holdout', 0.5);  
idxTrain = training(CVP); % Training-set indices  
idxTest = test(CVP); % Test-set indices
```

CVP is a cross-validation partition object that specifies the training and test sets.

Create a cost matrix such that misclassifying an arrhythmic patient into the no arrhythmia class is five times worse than misclassifying a patient without arrhythmia into the arrhythmia class. Classifying correctly incurs no cost. The rows indicate the true class and the columns indicate the predicted class. When conducting a cost-sensitive analysis, it is a good practice to specify the order of the classes.

```
cost = [0 1; 5 0];  
ClassNames = categorical([2 1]);
```

Train two boosting ensembles of 50 classification trees, one that uses AdaBoostM1 and the other that uses LogitBoost. Because there are missing values, specify to use surrogate splits. Train the models using the cost matrix.

```
t = templateTree('Surrogate', 'on');  
numTrees = 50;  
C1 = fitensemble(X(idxTrain, :), Y(idxTrain), 'AdaBoostM1', numTrees, t, ...  
    'Cost', cost);  
C2 = fitensemble(X(idxTrain, :), Y(idxTrain), 'LogitBoost', numTrees, t, ...  
    'Cost', cost);
```

C1 and C2 are trained ClassificationEnsemble models.

Test whether the AdaBoostM1 ensemble (C1) and the LogitBoost ensemble (C2) have equal predictive accuracy. Supply the cost matrix. Conduct the asymptotic, likelihood ratio, cost-sensitive test (the default when you pass in a cost matrix). Request to return  $p$ -values and misclassification costs.

```
[h,p,e1,e2] = compareHoldout(C1,C2,X(idxTest,:),X(idxTest,:),Y(idxTest),...
    'Cost',cost)

h =

    logical

     0

p =

    0.0743

e1 =

    1.3581

e2 =

    1.6186
```

$h = 0$  indicates to not reject the null hypothesis that the two models have equal predictive accuracies.

### Select Features Using Statistical Accuracy Comparison

Reduce classification model complexity by selecting a subset of predictor variables (features) from a larger set. Then, statistically compare the out-of-sample accuracy between the two models.

Load the `ionosphere` data set.

```
load ionosphere;
```

Create a partition that evenly splits the data into training and testing sets.

```
rng(1); % For reproducibility
CVP = cvpartition(Y,'holdout',0.5);
```

```
idxTrain = training(CVP);           % Training-set indices
idxTest  = test(CVP);              % Test-set indices
```

CVP is a cross-validation partition object that specifies the training and test sets.

Train an ensemble of 100 boosted classification trees using AdaBoostM1 and the entire set of predictors. Inspect the importance measure for each predictor.

```
nTrees = 100;
C2 = fitensemble(X(idxTrain,:), Y(idxTrain), 'AdaBoostM1', nTrees, 'Tree');
predImp = predictorImportance(C2);
```

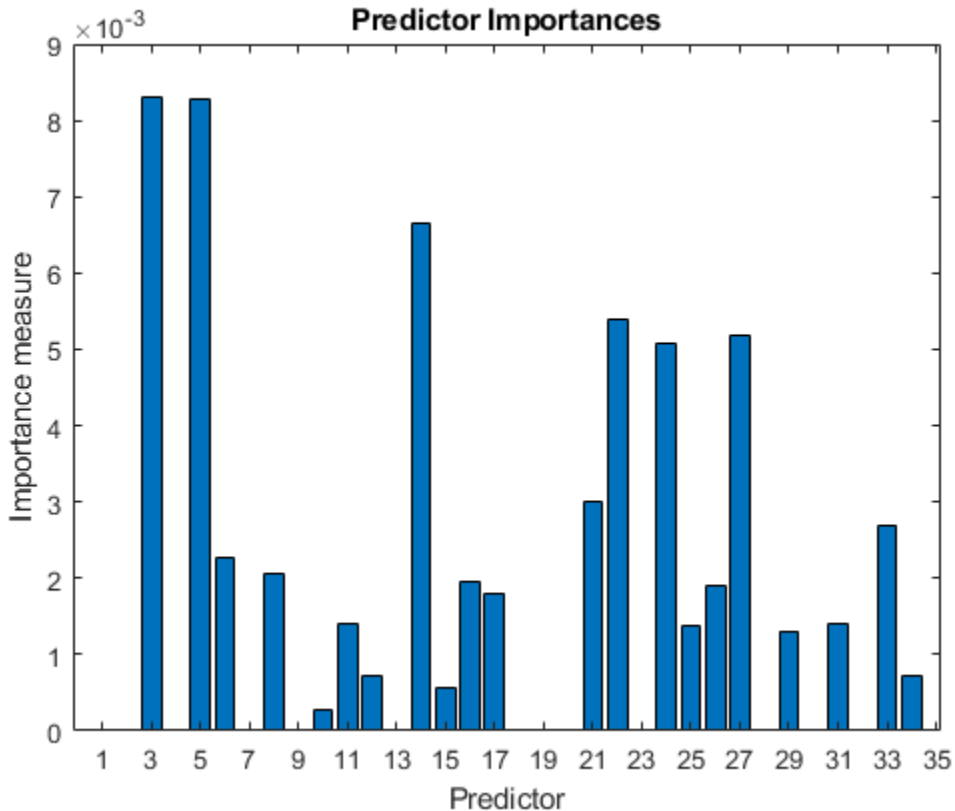
```
figure;
bar(predImp);
h = gca;
h.XTick = 1:2:h.XLim(2)
title('Predictor Importances');
xlabel('Predictor');
ylabel('Importance measure');
```

h =

Axes with properties:

```
        XLim: [-0.2000 35.2000]
        YLim: [0 0.0090]
        XScale: 'linear'
        YScale: 'linear'
GridLineStyle: '-'
        Position: [0.1300 0.1100 0.7750 0.8150]
        Units: 'normalized'
```

Use GET to show all properties



Identify the top five predictors in terms of their importance.

```
[~,idxSort] = sort(predImp, 'descend');
idx5 = idxSort(1:5);
```

Train another ensemble of 100 boosted classification trees using AdaBoostM1 and the five predictors with the best importance.

```
C1 = fitensemble(X(idxTrain,idx5),Y(idxTrain), 'AdaBoostM1', nTrees, ...
    'Tree');
```

Test whether the two models have equal predictive accuracies. Specify the reduced test-set predictor data for C1 and the full test-set predictor data for C2.

```
[h,p,e1,e2] = compareHoldout(C1,C2,X(idxTest,idx5),X(idxTest,:),Y(idxTest))
```

```
h =  
  logical  
  0  
  
p =  
  0.7744  
  
e1 =  
  0.0914  
  
e2 =  
  0.0857
```

`h = 0` indicates to not reject the null hypothesis that the two models have equal predictive accuracies. This result favors the simpler ensemble, C1.

## Definitions

### Cost-Sensitive Testing

Conduct cost-sensitive testing when the cost of misclassification is imbalanced. When conducting a cost-sensitive analysis, you can account for the cost imbalance in training the classification models, and then in statistically comparing them.

If the cost of misclassification is imbalanced, then the misclassification rate tends to be a poorly performing classification loss on page 32-6520. Use misclassification cost instead to compare classification models.

Misclassification costs are often unbalanced in applications. For example, consider classifying subjects based on a set of predictors into two categories: healthy and sick. Misclassifying a sick subject as healthy poses a danger to the subject's life. However,

misclassifying a healthy subject as sick can cause some inconvenience, but does not pose any danger. In this situation, you assign misclassification costs such that misclassifying a sick subject as healthy is more costly than misclassifying a healthy subject as sick.

The definitions that follow summarize the cost-sensitive tests. In the definitions:

- $n_{ijk}$  and  $\hat{\pi}_{ijk}$  are the number and estimated proportion of test-sample observations with true class  $k$  that the first classification model assigns label  $i$ . The second classification model assigns label  $j$ . The unknown, true value of  $\hat{\pi}_{ijk}$  is  $\pi_{ijk}$ . The test-set sample size is  $\sum_{i,j,k} n_{ijk} = n_{test}$ .  $\sum_{i,j,k} \pi_{ijk} = \sum_{i,j,k} \pi_{ijk} = 1$ .
- $c_{ij}$  is the relative cost of assigning label  $j$  to an observation with true class  $i$ .  $c_{ii} = 0$ ,  $c_{ij} \geq 0$ , and, for at least one  $(i,j)$  pair,  $c_{ij} > 0$ .
- All subscripts take on integer values from 1 through  $K$ , which is the number of classes.
- The expected difference in the misclassification costs of the two classification models is

$$\delta = \sum_{i=1}^K \sum_{j=1}^K \sum_{k=1}^K (c_{ki} - c_{kj}) \pi_{ijk}.$$

- The hypothesis test is

$$H_0 : \delta = 0$$

$$H_1 : \delta \neq 0$$

The available cost-sensitive tests are appropriate for two-tailed testing.

Available, asymptotic tests that address imbalanced costs are a chi-square test and a likelihood ratio test.

- Chi-square test — The chi-square test statistic is based on the Pearson and Neyman chi-square test statistics, but with a Laplace correction factor to account for any  $n_{ijk} = 0$ . The test statistic is

$$t_{\chi^2}^* = \sum_{i \neq j} \sum_k \frac{\left( n_{ijk} + 1 - (n_{test} + K^3) \hat{\pi}_{ijk}^{(1)} \right)^2}{n_{ijk} + 1}.$$

If  $1 - F_{\chi^2}(t_{\chi^2}^*; 1) < \alpha$ , then reject  $H_0$ .

- 
- $\hat{\pi}_{ijk}^{(1)}$  are estimated by minimizing  $t_{\chi^2}^*$  under the constraint that  $\delta = 0$ .
- $F_{\chi^2}(x; 1)$  is the  $\chi^2$  C.D.F. with one degree of freedom evaluated at  $x$ .
- Likelihood ratio test — The likelihood ratio test is based on  $N_{ijk}$ , which are binomial random variables having sample size  $n_{test}$  and success probability  $\pi_{ijk}$ . They represent the random number of observations with true class  $k$  that the first classification model assigns label  $i$ . The second classification model assigns label  $j$ . Jointly, their distribution is multinomial.

The test statistic is

$$t_{LRT}^* = 2 \log \left[ \frac{P\left(\bigcap_{i,j,k} N_{ijk} = n_{ijk}; n_{test}, \pi_{ijk} = \pi_{ijk}^{(2)}\right)}{P\left(\bigcap_{i,j,k} N_{ijk} = n_{ijk}; n_{test}, \pi_{ijk} = \pi_{ijk}^{(3)}\right)} \right].$$

If  $1 - F_{\chi^2}(t_{LRT}^*; 1) < \alpha$ , then reject  $H_0$ .

- 
- $\hat{\pi}_{ijk}^{(2)} = \frac{n_{ijk}}{n_{test}}$  is the unrestricted MLE of  $\pi_{ijk}$ .
- $\hat{\pi}_{ijk}^{(3)} = \frac{n_{ijk}}{n_{test} + \lambda(c_{ki} - c_{kj})}$  is the MLE under the null hypothesis that  $\delta = 0$ .  $\lambda$  is the solution to
 
$$\sum_{i,j,k} \frac{n_{ijk}(c_{ki} - c_{kj})}{n_{test} + \lambda(c_{ki} - c_{kj})} = 0.$$
- $F_{\chi^2}(x; 1)$  is the  $\chi^2$  C.D.F. with one degree of freedom evaluated at  $x$ .

## McNemar Tests

McNemar Tests are hypothesis tests that compare two population proportions while addressing the issues resulting from two dependent, matched-pair samples.



One way to compare the predictive accuracies of two classification models is:

- 1 Partition the data into training and test sets.
- 2 Train both classification models using the training set.
- 3 Predict class labels using the test set.
- 4 Summarize the results in a two-by-two table resembling this figure.

		Model 2		
		Correct	Incorrect	
Model 1	Correct	$n_{11}$	$n_{12}$	$n_{1\bullet}$
	Incorrect	$n_{21}$	$n_{22}$	$n_{2\bullet}$
		$n_{\bullet 1}$	$n_{\bullet 2}$	$n_{test}$

$n_{ii}$  are the number of concordant pairs, that is, the number of observations that both models classify the same way (correctly or incorrectly).  $n_{ij}$ ,  $i \neq j$ , are the number of discordant pairs, that is, the number of observations that models classify differently (correctly or incorrectly).

The misclassification rates for Models 1 and 2 are

$$\hat{\pi}_{2\bullet} = n_{2\bullet} / n$$

and  $\hat{\pi}_{\bullet 2} = n_{\bullet 2} / n$ , respectively. A two-sided test for comparing the accuracy of the two models is

$$H_0 : \pi_{\bullet 2} = \pi_{2\bullet}$$

$$H_1 : \pi_{\bullet 2} \neq \pi_{2\bullet}$$

The null hypothesis suggests that the population exhibits marginal homogeneity, which reduces the null hypothesis to  $H_0 : \pi_{12} = \pi_{21}$ . Also, under the null hypothesis,  $N_{12} \sim \text{Binomial}(n_{12} + n_{21}, 0.5)$  [1].

These facts are the basis for these, available McNemar test variants: the asymptotic, exact-conditional and mid-p-value McNemar tests. The definitions that follow summarize the available variants.

- Asymptotic — The asymptotic McNemar test statistics and rejection regions (for significance-level  $\alpha$ ) are:

- For one-sided tests, the test statistic

$$t_{a1}^* = \frac{n_{12} - n_{21}}{\sqrt{n_{12} + n_{21}}}.$$

If  $1 - \Phi\left(|t_1^*|\right) < \alpha$ , where  $\Phi$  is the standard Gaussian C.D.F., then reject  $H_0$ .

- For two-sided tests, the test statistic

$$t_{a2}^* = \frac{(n_{12} - n_{21})^2}{n_{12} + n_{21}}.$$

If  $1 - F_{\chi^2}\left(t_2^*; m\right) < \alpha$ , where  $F_{\chi^2}(x; m)$  is the  $\chi_m^2$  C.D.F. evaluated at  $x$ , then reject  $H_0$ .

This variant requires large-sample theory, specifically, the Gaussian approximation to the binomial distribution. Therefore:

- The total number of discordant pairs,  $n_d = n_{12} + n_{21}$  must be greater than 10 ([1], Ch. 10.1.4).
- In general, asymptotic tests do not guarantee nominal coverage. The observed probability of falsely rejecting the null hypothesis can exceed  $\alpha$ . Simulation studies in [18] suggest this, but the asymptotic McNemar test performs well in terms of statistical power.
- Exact Conditional — The exact-conditional McNemar test statistics and rejection regions (for significance-level  $\alpha$ ) are ([36], [38]):

- For one-sided tests, the test statistic

$$t_1^* = n_{12}$$

If  $F_{\text{Bin}}\left(t_1^*; n_d, 0.5\right) < \alpha$ , where  $F_{\text{Bin}}(x; n, p)$  is the binomial C.D.F. with sample size  $n$  and success probability  $p$  evaluated at  $x$ , then reject  $H_0$ .

- For two-sided tests, the test statistic

$$t_2^* = \min(n_{12}, n_{21})$$

If  $F_{\text{Bin}}(t_2^*; n_d, 0.5) < \alpha / 2$ , then reject  $H_0$ .

The exact conditional test always attains nominal coverage. Simulation studies in [18] suggest that the test is conservative, and then show that the test lacks statistical power compared to other variants. For small or highly discrete test samples, consider using the mid- $p$ -value test ([1], Ch. 3.6.3). For details, see `Test` and “McNemar Tests” on page 32-6517.

- Mid- $p$ -value test — The mid- $p$ -value McNemar test statistics and rejection regions (for significance-level  $\alpha$ ) are ([32]):

- For one-sided tests, the test statistic

$$t_1^* = n_{12}$$

If  $F_{\text{Bin}}(t_1^* - 1; n_{12} + n_{21}, 0.5) + 0.5f_{\text{Bin}}(t_1^*; n_{12} + n_{21}, 0.5) < \alpha$ , where  $F_{\text{Bin}}(x; n, p)$  and  $f_{\text{Bin}}(x; n, p)$  are the binomial C.D.F. and P.D.F, respectively, with sample size  $n$  and success probability  $p$  evaluated at  $x$ , then reject  $H_0$ .

- For two-sided tests, the test statistic

$$t_2^* = \min(n_{12}, n_{21})$$

If  $F_{\text{Bin}}(t_2^* - 1; n_{12} + n_{21} - 1, 0.5) + 0.5f_{\text{Bin}}(t_2^*; n_{12} + n_{21}, 0.5) < \alpha / 2$ , then reject  $H_0$ .

The mid- $p$ -value test addresses the over-conservative behavior of the exact conditional test. The simulation studies in [18] demonstrate that this test attains nominal coverage, and has good statistical power.

## Classification Loss

Classification losses indicate the accuracy of a classification model or set of predicted labels. Two classification losses are misclassification rate and cost.

`CompactClassificationTree.compareHoldout` returns the classification losses (see `e1` and `e2`) under the alternative hypothesis (i.e., the unrestricted classification losses).  $n_{ijk}$  is the number of test-sample observations with true class  $k$  that the first

classification model assigns label  $i$  and the second classification model assigns label  $j$ ,

and the corresponding estimated proportion is  $\hat{\pi}_{ijk} = \frac{n_{ijk}}{n_{test}}$ . . The test-set sample size is

$\sum_{i,j,k} n_{ijk} = n_{test}$ . The indices are taken from 1 through  $K$ , the number of classes.

- Misclassification rate, or classification error, is a scalar in the interval  $[0,1]$  representing the proportion of misclassified observations. That is, the misclassification rate for the first classification model is

$$e_1 = \sum_{j=1}^K \sum_{k=1}^K \sum_{i \neq k} \hat{\pi}_{ijk}.$$

For the misclassification rate of the second classification model ( $e_2$ ), switch the indices  $i$  and  $j$  in the formula.

Classification accuracy decreases as the misclassification rate increases to 1.

- Misclassification cost is a nonnegative scalar and is a measure of classification quality relative to the values the specified cost matrix elements. Its interpretation depends on the specified costs of misclassification. Misclassification cost is the weighted average of the costs of misclassification (specified in a cost matrix,  $C$ ) in which the weights are the respective, estimated proportions of misclassified observations. The misclassification cost for the first classification model is

$$e_1 = \sum_{j=1}^K \sum_{k=1}^K \sum_{i \neq k} \hat{\pi}_{ijk} c_{ki},$$

where  $c_{kj}$  is the cost of classifying an observation into class  $j$  if its true class is  $k$ . For the misclassification cost of the second classification model ( $e_2$ ), switch the indices  $i$  and  $j$  in the formula.

In general, for a fixed cost matrix, classification accuracy decreases as misclassification cost increases.

## Tips

- One way to perform cost-insensitive feature selection is:
  - 1 Train the first classification model ( $C_1$ ) using the full predictor set.

- 2 Train the second classification model (C2) using the reduced predictor set.
- 3 Specify X1 as the full, test-set predictor data and X2 as the reduced test-set predictor data.
- 4 Enter `compareHoldout(C1,C2,X1,X2,'Alternative','less')`. If `compareHoldout` returns 1, then there is enough evidence to suggest that the classification model that uses fewer predictors performs better than the model that uses the full predictor set.

Alternatively, you can assess whether there is a significant difference between the accuracies of the two models. To perform this assessment, remove the 'Alternative', 'greater' specification in step 4. `compareHoldout` conducts a two-sided test, and `h = 0` indicates that there is not enough evidence to suggest a difference in the accuracy of the two models.

- Cost-sensitive tests perform numerical optimization, which requires additional computational resources. The likelihood ratio test conducts numerical optimization indirectly by finding the root of a Lagrange multiplier in an interval. For some data sets, if the root lies close to the boundaries of the interval, then the method can fail. Therefore, if you have an Optimization Toolbox license, consider conducting the cost-sensitive chi-square test instead. For more details, see `CostTest` and “Cost-Sensitive Testing” on page 32-1053.

## Alternatives

To directly compare the accuracy of two sets of class labels in predicting a set of true class labels, use `testcholdout`.

## References

- [1] Agresti, A. *Categorical Data Analysis*, 2nd Ed. John Wiley & Sons, Inc.: Hoboken, NJ, 2002.
- [2] Fagerlan, M.W., S Lydersen, P. Laake. “The McNemar Test for Binary Matched-Pairs Data: Mid-p and Asymptotic Are Better Than Exact Conditional.” *BMC Medical Research Methodology*. Vol. 13, 2013, pp. 1–8.
- [3] Lancaster, H.O. “Significance Tests in Discrete Distributions.” *JASA*, Vol. 56, Number 294, 1961, pp. 223–234.

- [4] McNemar, Q. “Note on the Sampling Error of the Difference Between Correlated Proportions or Percentages.” *Psychometrika*, Vol. 12, Number 2, 1947, pp. 153–157.
- [5] Mosteller, F. “Some Statistical Problems in Measuring the Subjective Response to Drugs.” *Biometrics*, Vol. 8, Number 3, 1952, pp. 220–226.

## See Also

`fitctree` | `predict` | `testcholdout` | `testckfold`

## Topics

“Hypothesis Tests”

**Introduced in R2015a**

# ComputeOOBPrediction property

**Class:** TreeBagger

Flag to compute out-of-bag predictions

## Description

The `ComputeOOBPrediction` property is a logical flag specifying whether out-of-bag predictions for training observations should be computed. The default is `false`.

If this flag is true, the following properties are available:

- `OOBIndices`
- `OOBInstanceWeight`

If this flag is true, the following methods can be called:

- `oobError`
- `oobMargin`
- `oobMeanMargin`

## See Also

`OOBIndices` | `OOBInstanceWeight` | `oobError` | `oobMargin` | `oobMeanMargin`

## ComputeOOBVarImp property

**Class:** TreeBagger

Flag to compute out-of-bag variable importance

### Description

The `ComputeOOBVarImp` property is a logical flag specifying whether `TreeBagger` should compute out-of-bag estimates of variable importance. The default is `false`.

If this flag is true, the following properties are available:

- `OOBPermutedVarDeltaError`
- `OOBPermutedVarDeltaMeanMargin`
- `OOBPermutedVarCountRaiseMargin`

### See Also

`ComputeOOBPrediction` | `OOBPermutedVarCountRaiseMargin` | `OOBPermutedVarDeltaError` | `OOBPermutedVarDeltaMeanMargin` | `TreeBagger` | `oobMeanMargin`



# confusionmat

Confusion matrix

## Syntax

```
C = confusionmat(group,grouphat)
C = confusionmat(group,grouphat,'order',grouporder)
[C,order] = confusionmat(...)
```

## Description

`C = confusionmat(group,grouphat)` returns the confusion matrix `C` determined by the known and predicted groups in `group` and `grouphat`, respectively. `group` and `grouphat` are grouping variables with the same number of observations, as described in “Grouping Variables” on page 2-59. Input vectors must be of the same type. `C` is a square matrix with size equal to the total number of distinct elements in `group` and `grouphat`. `C(i,j)` is a count of observations known to be in group `i` but predicted to be in group `j`. Group indices and their order are the same for the rows and columns of `C`, computed by `grp2idx` using `grp2idx(group;grouphat)`. NaN, empty, or 'undefined' groups are not counted.

`C = confusionmat(group,grouphat,'order',grouporder)` uses `grouporder` to order the rows and columns of `C`. `grouporder` is a grouping variable containing all of the distinct elements in `group` and `grouphat`. If `grouporder` contains elements that are not in `group` or `grouphat`, the corresponding entries in `C` will be 0.

`[C,order] = confusionmat(...)` also returns the order of the rows and columns of `C` in a variable `order` the same type as `group` and `grouphat`.

## Examples

### Example 1

Display the confusion matrix for data with two misclassifications and one missing classification:

```
g1 = [1 1 2 2 3 3]';          % Known groups
g2 = [1 1 2 3 4 NaN]';       % Predicted groups

[C,order] = confusionmat(g1,g2)
C =
     2     0     0     0
     0     1     1     0
     0     0     0     1
     0     0     0     0
order =
     1
     2
     3
     4
```

### Example 2

Randomize the measurements and groups in Fisher's iris data:

```
load fisheriris
numObs = length(species);
p = randperm(numObs);
meas = meas(p,:);
species = species(p);
```

Use `classify` to classify measurements in the second half of the data, using the first half of the data for training:

```
half = floor(numObs/2);
training = meas(1:half,:);
trainingSpecies = species(1:half);
sample = meas(half+1:end,:);
grouphat = classify(sample,training,trainingSpecies);
```

Display the confusion matrix for the resulting classification:

```
group = species(half+1:end);
[C,order] = confusionmat(group,grouphat)
C =
    22     0     0
     2    22     0
     0     0    29
order =
    'virginica'
    'versicolor'
    'setosa'
```

## See Also

`crosstab` | `grp2idx`

## Topics

“Grouping Variables” on page 2-59

**Introduced in R2008b**

## controlchart

Shewhart control charts

### Syntax

```
controlchart(X)
controlchart(x,group)
controlchart(X,group)
[stats,plotdata] = controlchart(x,[group])
controlchart(x,group,'name',value)
```

### Description

`controlchart(X)` produces an xbar chart of the measurements in matrix `X`. Each row of `X` is considered to be a subgroup of measurements containing replicate observations taken at the same time. The rows should be in time order. If `X` is a time series object, the time samples should contain replicate observations.

The chart plots the means of the subgroups in time order, a center line (CL) at the average of the means, and upper and lower control limits (UCL, LCL) at three standard errors from the center line. The standard error is the estimated process standard deviation divided by the square root of the subgroup size. Process standard deviation is estimated from the average of the subgroup standard deviations. Out of control measurements are marked as violations and drawn with a red circle. Data cursor mode is enabled, so clicking any data point displays information about that point.

`controlchart(x,group)` accepts a grouping variable `group` for a vector of measurements `x`. `group` is a categorical variable, numeric vector, character vector, or cell array of character vectors the same length as `x`. Consecutive measurements `x(n)` sharing the same value of `group(n)` for  $1 \leq n \leq \text{length}(x)$  are defined to be a subgroup. Subgroups can have different numbers of observations.

`controlchart(X,group)` accepts a grouping variable `group` for a matrix of measurements in `X`. In this case, `group` is only used to label the time axis; it does not change the default grouping by rows.

`[stats,plotdata] = controlchart(x,[group])` returns a structure `stats` of subgroup statistics and parameter estimates, and a structure `plotdata` of plotted values. `plotdata` contains one record for each chart.

The fields in `stats` and `plotdata` depend on the chart type.

The fields in `stats` are selected from the following:

- `mean` — Subgroup means
- `std` — Subgroup standard deviations
- `range` — Subgroup ranges
- `n` — Subgroup size, or total inspection size or area
- `i` — Individual data values
- `ma` — Moving averages
- `mr` — Moving ranges
- `count` — Count of defects or defective items
- `mu` — Estimated process mean
- `sigma` — Estimated process standard deviation
- `p` — Estimated proportion defective
- `m` — Estimated mean defects per unit

The fields in `plotdata` are the following:

- `pts` — Plotted point values
- `cl` — Center line
- `lcl` — Lower control limit
- `ucl` — Upper control limit
- `se` — Standard error of plotted point
- `n` — Subgroup size
- `ooc` — Logical that is true for points that are out of control

`controlchart(x,group,'name',value)` specifies one or more of the following optional parameter name/value pairs, with `name` in single quotes:

- `charttype` — The name of a chart type chosen from among the following:
  - `'xbar'` —  $\bar{x}$  or mean
  - `'s'` — Standard deviation
  - `'r'` — Range
  - `'ewma'` — Exponentially weighted moving average
  - `'i'` — Individual observation
  - `'mr'` — Moving range of individual observations
  - `'ma'` — Moving average of individual observations
  - `'p'` — Proportion defective
  - `'np'` — Number of defectives
  - `'u'` — Defects per unit
  - `'c'` — Count of defects

Alternatively, a parameter can be a cell array listing multiple compatible chart types. There are four sets of compatible types:

- `'xbar', 's', 'r', and 'ewma'`
- `'i', 'mr', and 'ma'`
- `'p' and 'np'`
- `'u' and 'c'`
- `display` — Either `'on'` (default) to display the control chart, or `'off'` to omit the display
- `label` — A character vector or cell array of character vectors, one per subgroup. This label is displayed as part of the data cursor for a point on the plot.
- `lambda` — A parameter between 0 and 1 controlling how much the current prediction is influenced by past observations in an EWMA plot. Higher values of `'lambda'` give less weight to past observations and more weight to the current observation. The default is 0.4.
- `limits'` — A three-element vector specifying the values of the lower control limit, center line, and upper control limits. Default is to estimate the center line and to compute control limits based on the estimated value of sigma. Not permitted if there are multiple chart types.

- `mean` — Value for the process mean, or an empty value (default) to estimate the mean from  $X$ . This is the `p` parameter for `p` and `np` charts, the mean defects per unit for `u` and `c` charts, and the normal `mu` parameter for other charts.
- `nsigma` — The number of sigma multiples from the center line to a control limit. Default is 3.
- `parent` — The handle of the axes to receive the control chart plot. Default is to create axes in a new figure. Not permitted if there are multiple chart types.
- `rules` — The name of a control rule, or a cell array containing multiple control rule names. These rules, together with the control limits, determine if a point is marked as out of control. The default is to apply no control rules, and to use only the control limits to decide if a point is out of control. See `controlrules` for more information. Control rules are applied to charts that measure the process level (`xbar`, `i`, `c`, `u`, `p`, and `np`) rather than the variability (`r`, `s`), and they are not applied to charts based on moving statistics (`ma`, `mr`, `ewma`).
- `sigma` — Either a value for sigma, or a method of estimating sigma chosen from among `'std'` (the default) to use the average within-subgroup standard deviation, `'range'` to use the average subgroup range, and `'variance'` to use the square root of the pooled variance. When creating `i`, `mr`, or `ma` charts for data not in subgroups, the estimate is always based on a moving range.
- `specs` — A vector specifying specification limits. Typically this is a two-element vector of lower and upper specification limits. Since specification limits typically apply to individual measurements, this parameter is primarily suitable for `i` charts. These limits are not plotted on `r`, `s`, or `mr` charts.
- `unit` — The total number of inspected items for `p` and `np` charts, and the size of the inspected unit for `u` and `c` charts. In both cases  $X$  must be the count of the number of defects or defectives found. Default is 1 for `u` and `c` charts. This argument is required (no default) for `p` and `np` charts.
- `width` — The width of the window used for computing the moving ranges and averages in `mr` and `ma` charts, and for computing the sigma estimate in `i`, `mr`, and `ma` charts. Default is 5.

## Examples

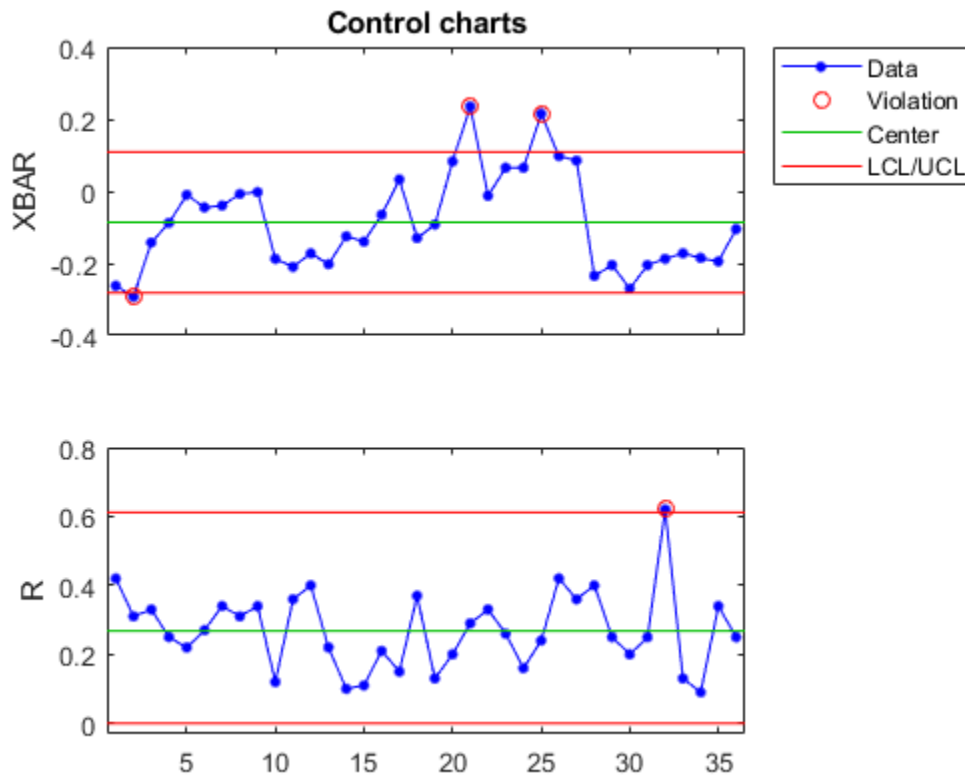
## XBar and R Charts

Load the sample data.

```
load parts
```

Create xbar and r control charts for the data.

```
st = controlchart(runout, 'charttype', {'xbar' 'r'});
```



Display the process mean and standard deviation.

```
fprintf('Parameter estimates: mu = %g, sigma = %g\n', st.mu, st.sigma);
```



Parameter estimates:  $\mu = -0.0863889$ ,  $\sigma = 0.130215$

## See Also

controlrules

## Topics

“Grouping Variables” on page 2-59

**Introduced in R2006b**

## controlrules

Western Electric and Nelson control rules

### Syntax

```
R = controlrules('rules',x,cl,se)
[R,RULES] = controlrules('rules',x,cl,se)
```

### Description

`R = controlrules('rules',x,cl,se)` determines which points in the vector `x` violate the control rules in `rules`. `cl` is a vector of center-line values. `se` is a vector of standard errors. (Typically, control limits on a control chart are at the values  $cl - 3*se$  and  $cl + 3*se$ .) `rules` is the name of a control rule, or a cell array containing multiple control rule names, from the list below. If `x` has  $n$  values and `rules` contains  $m$  rules, then `R` is an  $n$ -by- $m$  logical array, with `R(i,j)` assigned the value 1 if point `i` violates rule `j`, 0 if it does not.

The following are accepted values for `rules` (specified inside single quotes):

- `we1` — 1 point above  $cl + 3*se$
- `we2` — 2 of 3 above  $cl + 2*se$
- `we3` — 4 of 5 above  $cl + se$
- `we4` — 8 of 8 above `cl`
- `we5` — 1 below  $cl - 3*se$
- `we6` — 2 of 3 below  $cl - 2*se$
- `we7` — 4 of 5 below  $cl - se$
- `we8` — 8 of 8 below `cl`
- `we9` — 15 of 15 between  $cl - se$  and  $cl + se$
- `we10` — 8 of 8 below  $cl - se$  or above  $cl + se$

- n1 — 1 point below  $c1 - 3*se$  or above  $c1 + 3*se$
- n2 — 9 of 9 on the same side of  $c1$
- n3 — 6 of 6 increasing or decreasing
- n4 — 14 alternating up/down
- n5 — 2 of 3 below  $c1 - 2*se$  or above  $c1 + 2*se$ , same side
- n6 — 4 of 5 below  $c1 - se$  or above  $c1 + se$ , same side
- n7 — 15 of 15 between  $c1 - se$  and  $c1 + se$
- n8 — 8 of 8 below  $c1 - se$  or above  $c1 + se$ , either side
- we — All Western Electric rules
- n — All Nelson rules

For multi-point rules, a rule violation at point  $i$  indicates that the set of points ending at point  $i$  triggered the rule. Point  $i$  is considered to have violated the rule only if it is one of the points violating the rule's condition.

Any points with NaN as their  $x$ ,  $c1$ , or  $se$  values are not considered to have violated rules, and are not counted in the rules for other points.

Control rules can be specified in the `controlchart` function as values for the 'rules' parameter.

`[R,RULES] = controlrules('rules',x,c1,se)` returns a cell array of text RULES listing the rules applied.

## Examples

### Use Western Electric Control Rule

Load the sample data.

```
load parts;
```

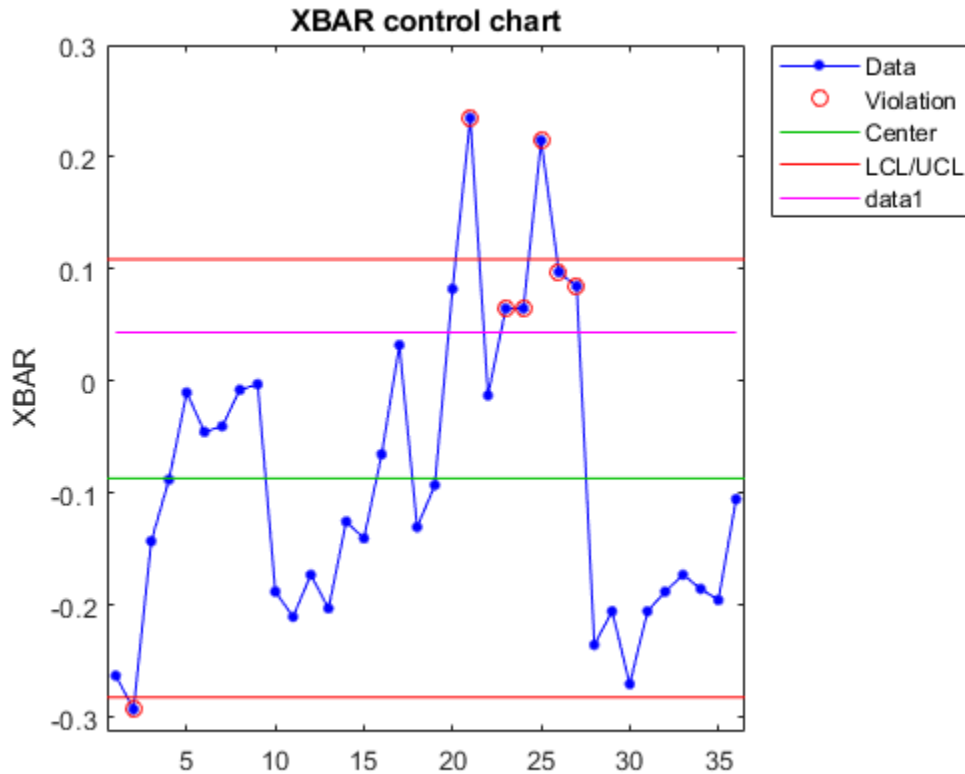
Create an Xbar chart using the we2 rule to mark out of control measurements.

```
st = controlchart(runout, 'rules', 'we2');
x = st.mean;
```

```

cl = st.mu;
se = st.sigma./sqrt(st.n);
hold on
plot(cl+2*se, 'm')

```



You can see the out of control points marked with a red circle.

Use `controlrules` to identify the measurements that violate the control rule.

```

R = controlrules('we2',x,cl,se);
I = find(R)

```

```

I =

```

21  
23  
24  
25  
26  
27

## See Also

controlchart

**Introduced in R2006b**

## Converged property

**Class:** `gmdistribution`

Determine if algorithm converged

### Description

Logical `true` if the algorithm has converged; logical `false` if the algorithm has not converged.

---

**Note** This property applies only to `gmdistribution` objects constructed with `fitgmdist`.

---

# cophenet

Cophenetic correlation coefficient

## Syntax

```
c = cophenet(Z, Y)
[c, d] = cophenet(Z, Y)
```

## Description

`c = cophenet(Z, Y)` computes the cophenetic correlation coefficient for the hierarchical cluster tree represented by `Z`. `Z` is the output of the `linkage` function. `Y` contains the distances or dissimilarities used to construct `Z`, as output by the `pdist` function. `Z` is a matrix of size  $(m-1)$ -by-3, with distance information in the third column. `Y` is a vector of size  $m*(m-1)/2$ .

`[c, d] = cophenet(Z, Y)` returns the cophenetic distances `d` in the same lower triangular distance vector format as `Y`.

The cophenetic correlation for a cluster tree is defined as the linear correlation coefficient between the cophenetic distances obtained from the tree, and the original distances (or dissimilarities) used to construct the tree. Thus, it is a measure of how faithfully the tree represents the dissimilarities among observations.

The cophenetic distance between two observations is represented in a dendrogram by the height of the link at which those two observations are first joined. That height is the distance between the two subclusters that are merged by that link.

The output value, `c`, is the cophenetic correlation coefficient. The magnitude of this value should be very close to 1 for a high-quality solution. This measure can be used to compare alternative cluster solutions obtained using different algorithms.

The cophenetic correlation between `Z(:, 3)` and `Y` is defined as

$$c = \frac{\sum_{i < j} (Y_{ij} - y)(Z_{ij} - z)}{\sqrt{\sum_{i < j} (Y_{ij} - y)^2 \sum_{i < j} (Z_{ij} - z)^2}}$$

where:

- $Y_{ij}$  is the distance between objects  $i$  and  $j$  in  $Y$ .
- $Z_{ij}$  is the cophenetic distance between objects  $i$  and  $j$ , from  $Z(:, 3)$ .
- $y$  and  $z$  are the average of  $Y$  and  $Z(:, 3)$ , respectively.

## Examples

```
X = [rand(10,3); rand(10,3)+1; rand(10,3)+2];
Y = pdist(X);
Z = linkage(Y, 'average');

% Compute Spearman's rank correlation between the
% dissimilarities and the cophenetic distances
[c,D] = cophenet(Z,Y);
r = corr(Y',D', 'type', 'spearman')
r =
    0.8279
```

## See Also

`cluster` | `dendrogram` | `inconsistent` | `linkage` | `pdist` | `squareform`

Introduced before R2006a



# copulacdf

Copula cumulative distribution function

## Syntax

```
y = copulacdf('Gaussian',u,rho)
```

```
y = copulacdf('t',u,rho,nu)
```

```
y = copulacdf(family,u,alpha)
```

## Description

`y = copulacdf('Gaussian',u,rho)` returns the cumulative probability of the Gaussian copula, with linear correlation parameters `rho` evaluated at the points in `u`.

`y = copulacdf('t',u,rho,nu)` returns the cumulative probability of the  $t$  copula, with linear correlation parameters, `rho`, and degrees of freedom parameter `nu` evaluated at the points in `u`.

`y = copulacdf(family,u,alpha)` returns the cumulative probability of the bivariate Archimedean copula of the type specified by `family`, with scalar parameter `alpha` evaluated at the points in `u`.

## Examples

### Compute the Clayton Copula cdf

Define two 10-by-10 matrices containing the values at which to compute the cdf.

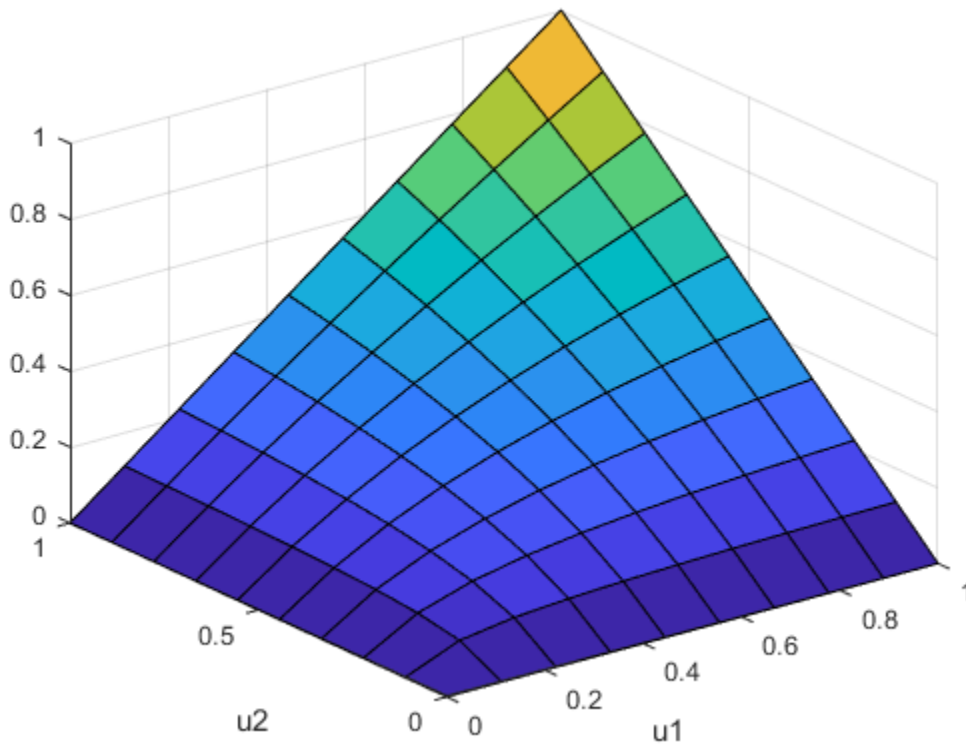
```
u = linspace(0,1,10);  
[u1,u2] = meshgrid(u,u);
```

Compute the cdf of a Clayton copula that has an alpha parameter equal to 1, at the values in `u`.

```
y = copulacdf('Clayton', [u1(:), u2(:)], 1);
```

Plot the cdf as a surface, and label the axes.

```
surf(u1, u2, reshape(y, 10, 10))  
xlabel('u1')  
ylabel('u2')
```



- “Generate Correlated Data Using Rank Correlation” on page 5-142

## Input Arguments

### **u** — Values at which to evaluate cdf

matrix of scalar values in the range [0,1]

Values at which to evaluate the cdf, specified as a matrix of scalar values in the range [0,1]. If *u* is an *n*-by-*p* matrix, then its values represent *n* points in the *p*-dimensional unit hypercube. If *u* is an *n*-by-2 matrix, then its values represent *n* points in the unit square.

If you specify a bivariate Archimedean copula type ('Clayton', 'Frank', or 'Gumbel'), then *u* must be an *n*-by-2 matrix.

Data Types: `single` | `double`

### **rho** — Linear correlation parameters

scalar values | matrix of scalar values

Linear correlation parameters for the copula, specified as a scalar value or matrix of scalar values.

- If *u* is an *n*-by-*p* matrix, then *rho* is a *p*-by-*p* correlation matrix.
- If *u* is an *n*-by-2 matrix, then *rho* can be a scalar correlation coefficient.

Data Types: `single` | `double`

### **nu** — Degrees of freedom

positive integer value

Degrees of freedom for the *t* copula, specified as a positive integer value.

Data Types: `single` | `double`

### **family** — Bivariate Archimedean copula family

'Clayton' | 'Frank' | 'Gumbel'

Bivariate Archimedean copula family, specified as one of the following.

'Clayton'	Clayton copula
'Frank'	Frank copula
'Gumbel'	Gumbel copula

Data Types: `single` | `double`

### **alpha** — Bivariate Archimedean copula parameter

scalar value

Bivariate Archimedean copula parameter, specified as a scalar value. Permitted values for `alpha` depend on the specified copula family.

Copula Family	Permitted Alpha Values
'Clayton'	$[0, \infty)$
'Frank'	$(-\infty, \infty)$
'Gumbel'	$[1, \infty)$

Data Types: `single` | `double`

## Output Arguments

### **y** — Cumulative distribution function

vector of scalar values

Cumulative distribution function of the copula, evaluated at the points in `u`, returned as a vector of scalar values.

## See Also

`copulaparam` | `copulapdf` | `copularnd` | `copulastat`

## Topics

“Generate Correlated Data Using Rank Correlation” on page 5-142

“Copulas: Generate Correlated Samples” on page 5-164

Introduced in R2006a

# copulafit

Fit copula to data

## Syntax

```
rhohat = copulafit('Gaussian',u)

[rhohat,nuhat] = copulafit('t',u)
[rhohat,nuhat,nuci] = copulafit('t',u)

paramhat = copulafit(family,u)
[paramhat,paramci] = copulafit(family,u)

___ = copulafit(___ ,Name,Value)
```

## Description

`rhohat = copulafit('Gaussian',u)` returns an estimate, `rhohat`, of the matrix of linear correlation parameters for a Gaussian copula, given the data in `u`.

`[rhohat,nuhat] = copulafit('t',u)` returns an estimate, `rhohat`, of the matrix of linear correlation parameters for a  $t$  copula, and an estimate of the degrees of freedom parameter, `nuhat`, given the data in `u`.

`[rhohat,nuhat,nuci] = copulafit('t',u)` also returns an approximate 95% confidence interval, `nuci`, for the degrees of freedom estimated in `nuhat`.

`paramhat = copulafit(family,u)` returns an estimate, `paramhat`, of the copula parameter for a bivariate Archimedean copula of the type specified by `family`, given the data in `u`.

`[paramhat,paramci] = copulafit(family,u)` also returns an approximate 95% confidence interval, `paramci`, for the copula parameter estimated in `paramhat`.

`___ = copulafit(___ ,Name,Value)` returns any of the previous syntaxes, with additional options specified by one or more `Name, Value` pair arguments. For example,

you can specify the confidence interval to compute, or specify control parameters for the iterative parameter estimation algorithm using a options structure.

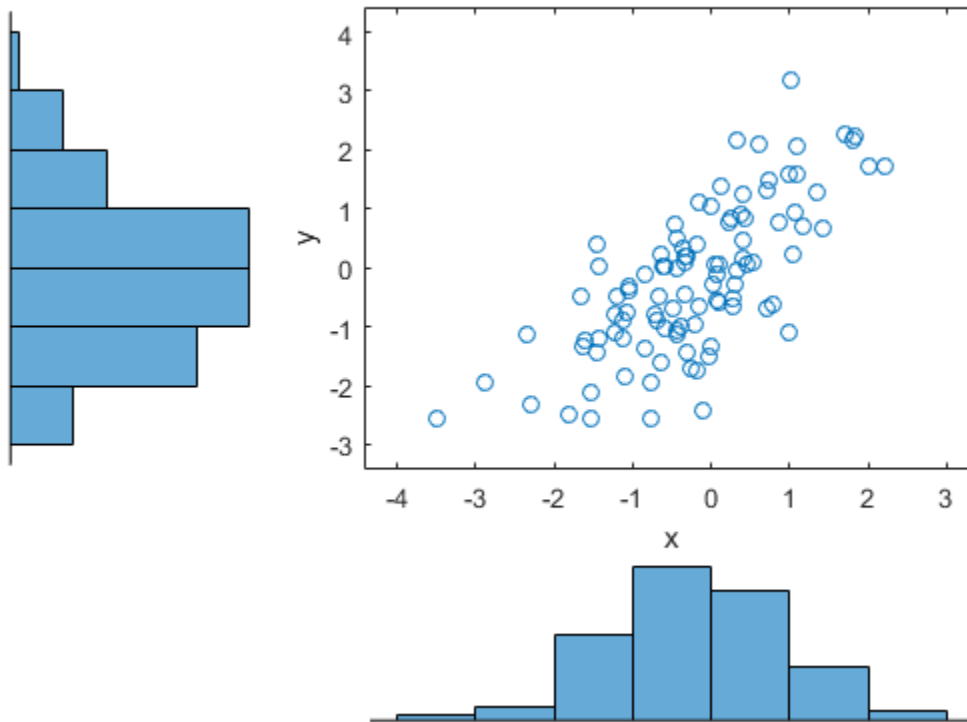
## Examples

### Fit a *t*Copula to Data

Load and plot simulated stock return data.

```
load stockreturns
x = stocks(:,1);
y = stocks(:,2);

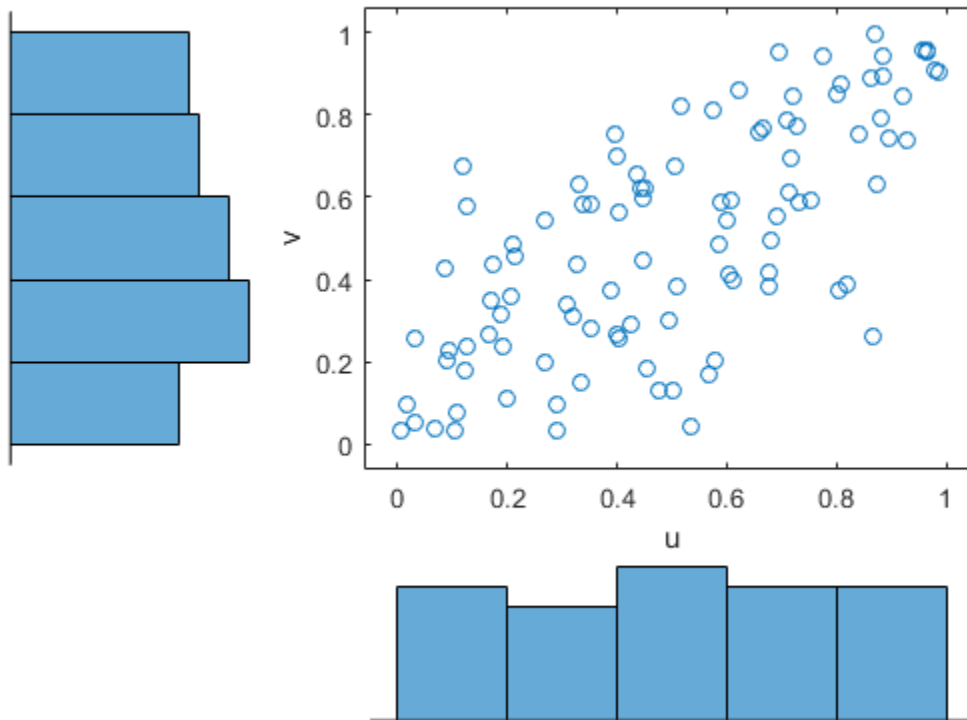
figure;
scatterhist(x,y)
```



Transform the data to the copula scale (unit square) using a kernel estimator of the cumulative distribution function.

```
u = ksdensity(x,x,'function','cdf');  
v = ksdensity(y,y,'function','cdf');
```

```
figure;  
scatterhist(u,v)  
xlabel('u')  
ylabel('v')
```



Fit a  $t$  copula to the data.

```
rng default % For reproducibility
[Rho,nu] = copulafit('t',[u v], 'Method', 'ApproximateML')
```

Rho =

```
1.0000    0.7220
0.7220    1.0000
```

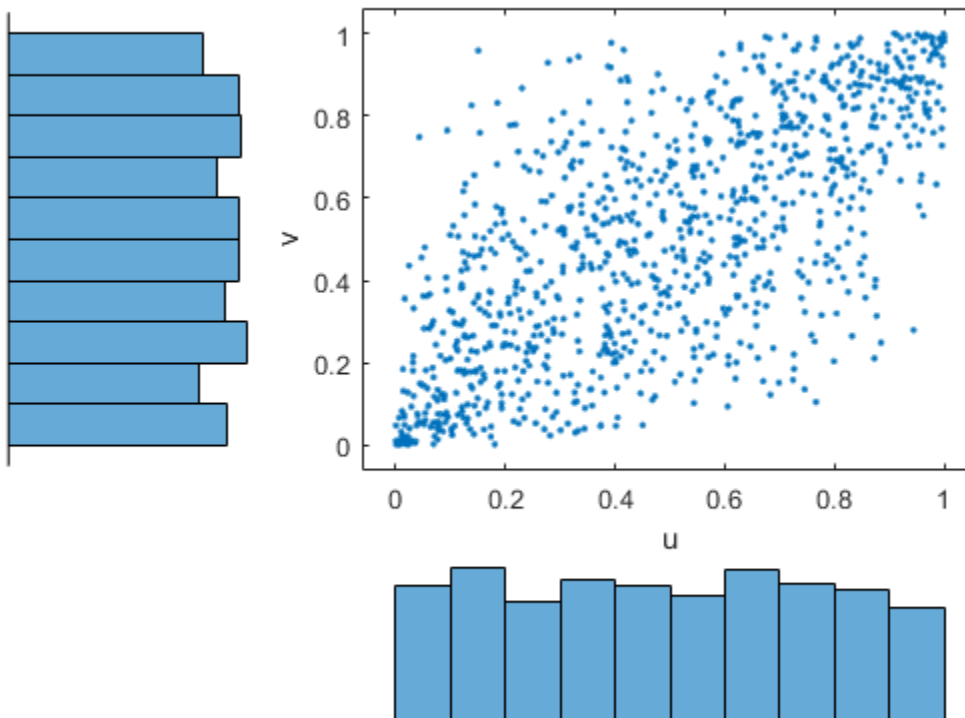
nu =



3.1800e+06

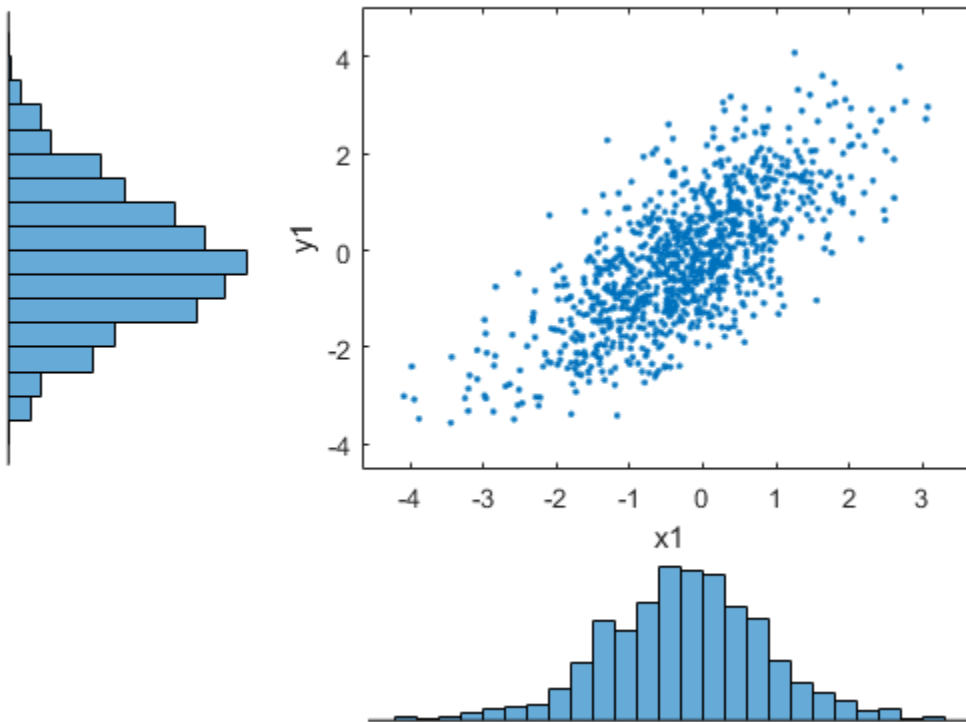
Generate a random sample from the  $t$  copula.

```
r = copularnd('t',Rho,nu,1000);  
u1 = r(:,1);  
v1 = r(:,2);  
  
figure;  
scatterhist(u1,v1)  
xlabel('u')  
ylabel('v')  
set(get(gca,'children'),'marker','.')
```



Transform the random sample back to the original scale of the data.

```
x1 = ksdensity(x,u1,'function','icdf');  
y1 = ksdensity(y,v1,'function','icdf');  
  
figure;  
scatterhist(x1,y1)  
set(get(gca,'children'),'marker','.')
```



- “Generate Correlated Data Using Rank Correlation” on page 5-142

## Input Arguments

### **u** — Copula values

matrix of scalar values in the range (0,1)

Copula values, specified as a matrix of scalar values in the range (0,1). If *u* is an *n*-by-*p* matrix, then its values represent *n* points in the *p*-dimensional unit hypercube. If *u* is an *n*-by-2 matrix, then its values represent *n* points in the unit square.

If you specify a bivariate Archimedean copula type ('Clayton', 'Frank', or 'Gumbel'), then *u* must be an *n*-by-2 matrix.

Data Types: single | double

### **family** — Bivariate Archimedean copula family

'Clayton' | 'Frank' | 'Gumbel'

Bivariate Archimedean copula family, specified as one of the following.

'Clayton'	Clayton copula
'Frank'	Frank copula
'Gumbel'	Gumbel copula

Data Types: single | double

## Name-Value Pair Arguments

Specify optional comma-separated pairs of *Name*, *Value* arguments. *Name* is the argument name and *Value* is the corresponding value. *Name* must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as *Name1*, *Value1*, ..., *NameN*, *ValueN*.

Example: 'Alpha', 0.01, 'Method', 'ApproximateML' computes 99% confidence intervals for the estimated copula parameter and uses an approximation method to fit the copula.

### **Alpha** — Significance level for confidence intervals

0.05 (default) | scalar value in the range (0,1)

Significance level for confidence intervals, specified as the comma-separated pair consisting of 'Alpha' and a scalar value in the range (0,1). `copulafit` returns approximate  $100 \times (1-\text{Alpha})\%$  confidence intervals.

Example: 'Alpha', 0.01

Data Types: single | double

#### **Method — Method for fitting $t$ copula**

'ML' (default) | 'ApproximateML'

Method for fitting  $t$  copula, specified as the comma-separated pair consisting of 'Method' and either 'ML' or 'ApproximateML'.

If you specify 'ApproximateML', then `copulafit` fits a  $t$  copula for large samples by maximizing an objective function that approximates the profile log likelihood for the degrees of freedom parameter [1]. This method can be significantly faster than maximum likelihood ('ML'), but the estimates and confidence limits may not be accurate for small to moderate sample sizes.

Example: 'Method', 'ApproximateML'

#### **Options — Control parameter specifications**

structure

Control parameter specifications, specified as the comma-separated pair consisting of 'Options' and an options structure created by `statset`. To see the fields and default values used by `copulafit`, type `statset('copulafit')` at the command prompt.

This name-value pair is not applicable when you specify the copula type as 'Gaussian'.

Data Types: struct

## Output Arguments

#### **rho\_hat — Estimated correlation parameters for the fitted Gaussian copula**

matrix of scalar values

Estimated correlation parameters for the fitted Gaussian copula, given the data in `u`, returned as a matrix of scalar values.

**nuhat** — Estimated degrees of freedom parameter for the fitted  $t$  copula  
scalar value

Estimated degrees of freedom parameter for the fitted  $t$  copula, returned as a scalar value.

**nuci** — Approximate confidence interval for the degrees of freedom parameter  
1-by-2 matrix of scalar values

Approximate confidence interval for the degrees of freedom parameter, returned as a 1-by-2 matrix of scalar values. The first column contains the lower boundary, and the second column contains the upper boundary. By default, `copulafit` returns the approximate 95% confidence interval. You can specify a different confidence interval using the 'Alpha' name-value pair.

**paramhat** — Estimated copula parameter for the fitted Archimedean copula  
scalar value

Estimated copula parameter for the fitted Archimedean copula, returned as a scalar value.

**paramci** — Approximate confidence interval for the copula parameter  
1-by-2 matrix of scalar values

Approximate confidence interval for the copula parameter, returned as a 1-by-2 matrix of scalar values. The first column contains the lower boundary, and the second column contains the upper boundary. By default, `copulafit` returns the approximate 95% confidence interval. You can specify a different confidence interval using the 'Alpha' name-value pair.

## Algorithms

By default, `copulafit` uses maximum likelihood to fit a copula to  $u$ . When  $u$  contains data transformed to the unit hypercube by parametric estimates of their marginal cumulative distribution functions, this is known as the *Inference Functions for Margins (IFM)* method. When  $u$  contains data transformed by the empirical cdf (see `ecdf`), this is known as *Canonical Maximum Likelihood (CML)*.

## References

- [1] Bouyé, E., V. Durrleman, A. Nikeghbali, G. Riboulet, and T. Roncalli. “Copulas for Finance: A Reading Guide and Some Applications.” Working Paper. Groupe de Recherche Opérationnelle, Crédit Lyonnais, Paris, 2000.

## See Also

`copulacdf` | `copulaparam` | `copulapdf` | `copularnd` | `copulastat`

## Topics

“Generate Correlated Data Using Rank Correlation” on page 5-142

“Copulas: Generate Correlated Samples” on page 5-164

**Introduced in R2007b**

# copulaparam

Copula parameters as function of rank correlation

## Syntax

```
rho = copulaparam('Gaussian', r)
rho = copulaparam('t', r, nu)
alpha = copulaparam(family, r)
___ = copulaparam( ___, Name, Value)
```

## Description

`rho = copulaparam('Gaussian', r)` returns the linear correlation parameters, `rho`, that correspond to a Gaussian copula with Kendall's rank correlation, `r`.

`rho = copulaparam('t', r, nu)` returns the linear correlation parameters, `rho`, that correspond to a *t* copula with Kendall's rank correlation, `r`, and degrees of freedom, `nu`.

`alpha = copulaparam(family, r)` returns the copula parameter, `alpha`, that corresponds to a bivariate Archimedean copula of the type specified by `family`, with Kendall's rank correlation, `r`.

`___ = copulaparam( ___, Name, Value)` returns the correlation parameter using any of the previous syntaxes, with additional options specified by one or more `Name, Value` pair arguments. For example, you can specify whether the input rank correlation value is Spearman's *rho* or Kendall's *tau*.

## Examples

### Generate Correlated Data Using the Inverse cdf

Generate correlated random data from a beta distribution using a bivariate Gaussian copula with Kendall's *tau* rank correlation equal to -0.5.

Compute the linear correlation parameter from the rank correlation value.

```
rng default % For reproducibility
tau = -0.5;
rho = copulaparam('Gaussian',tau)
```

```
rho =
    -0.7071
```

Use a Gaussian copula to generate a two-column matrix of dependent random values.

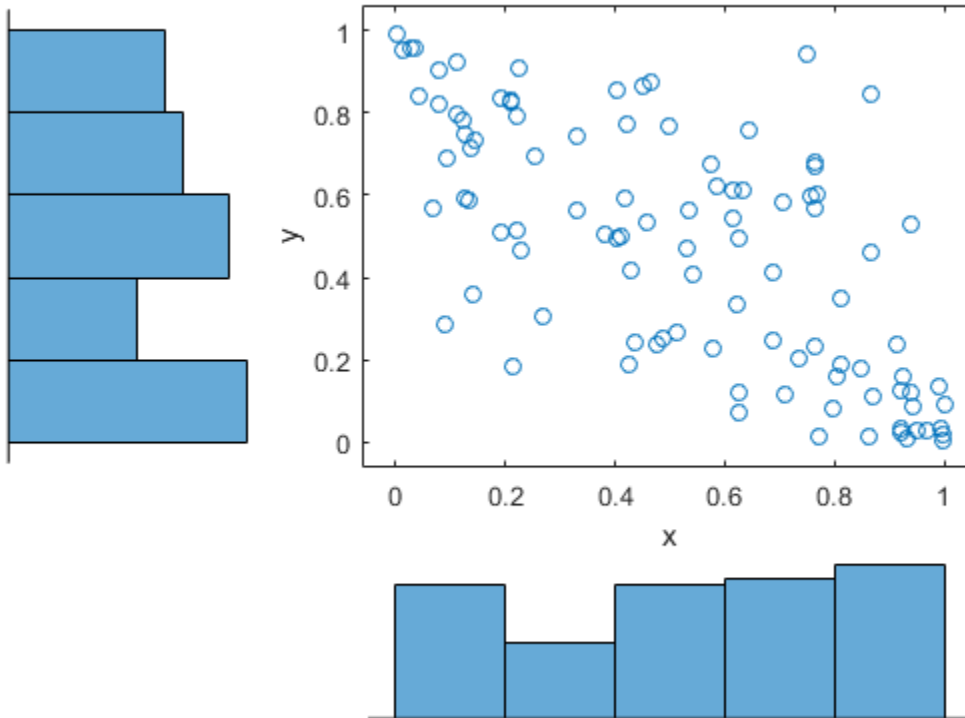
```
u = copularnd('gaussian',rho,100);
```

Each column contains 100 random values between 0 and 1, inclusive, sampled from a continuous uniform distribution.

Create a scatterhist plot to visualize the random numbers generated using the copula.

```
figure
scatterhist(u(:,1),u(:,2))
```





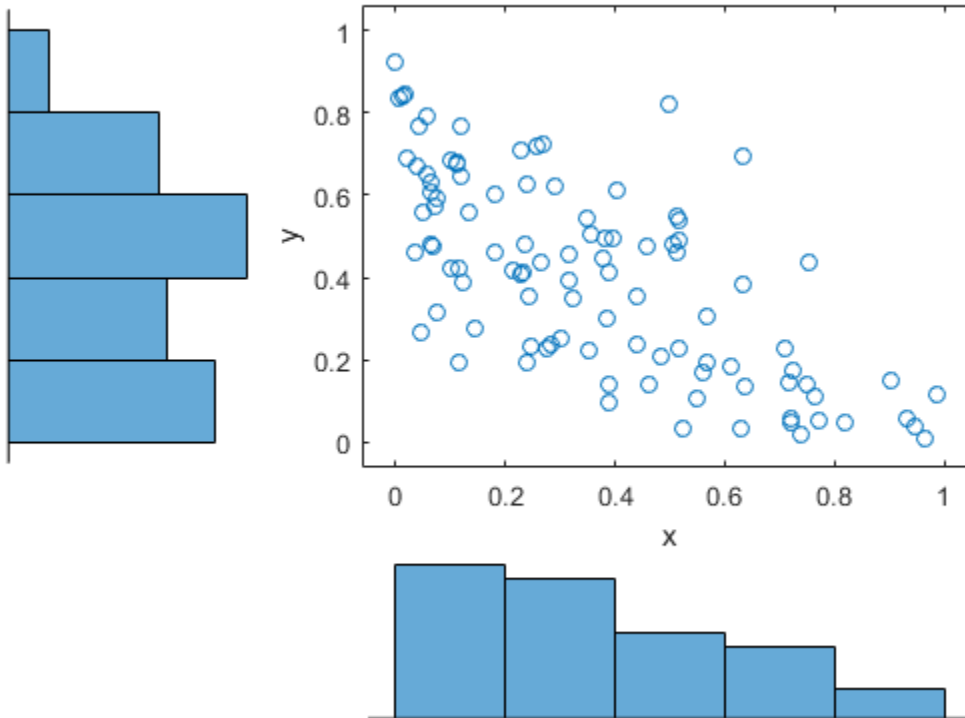
The histograms show that the data in each column of the copula has a marginal uniform distribution. The scatterplot shows that the data in the two columns is negatively correlated.

Use the inverse cdf function `betainv` to transform each column of the uniform marginal distributions into random numbers from a beta distribution. In the first column, the first shape parameter  $A$  is equal to 1, and a second shape parameter  $B$  is equal to 2. In the second column, the first shape parameter  $A$  is equal to 1.5, and a second shape parameter  $B$  is equal to 2.

```
b = [betainv(u(:,1),1,2), betainv(u(:,2),1.5,2)];
```

Create a `scatterhist` plot to visualize the correlated beta distribution data.

```
figure
scatterhist(b(:,1),b(:,2))
```



The histograms show the marginal beta distributions for each variable. The scatterplot shows the negative correlation.

Verify that the sample has a rank correlation approximately equal to the initial value for Kendall's  $\tau$ .

```
tau_sample = corr(b, 'type', 'kendall')
```

```
tau_sample =
```

```

1.0000    -0.5135
-0.5135    1.0000

```

The sample rank correlation of -0.5135 is approximately equal to the -0.5 initial value for *tau*.

- “Generate Correlated Data Using Rank Correlation” on page 5-142

## Input Arguments

### **r** — Copula rank correlation

scalar value | matrix of scalar values

Copula rank correlation, returned as a scalar value or matrix of scalar values.

- If *r* is a scalar correlation coefficient, then *rho* is a scalar correlation coefficient corresponding to a bivariate copula.
- If *r* is a *p*-by-*p* correlation matrix, then *rho* is a *p*-by-*p* correlation matrix.

If the copula is specified as one of the bivariate Archimedean copula types ('Clayton', 'Frank', or 'Gumbel'), then *r* is a scalar value.

### **nu** — Degrees of freedom

positive integer value

Degrees of freedom for the *t* copula, specified as a positive integer value.

Data Types: single | double

### **family** — Bivariate Archimedean copula family

'Clayton' | 'Frank' | 'Gumbel'

Bivariate Archimedean copula family, specified as one of the following.

'Clayton'	Clayton copula
'Frank'	Frank copula
'Gumbel'	Gumbel copula

Data Types: single | double

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'Type', 'Spearman'` computes Spearman's rank correlation.

### **Type** — Type of rank correlation

`'Kendall'` (default) | `'Spearman'`

Type of rank correlation, specified as the comma-separated pair consisting of `'Type'` and one of the following.

- `'Kendall'` — Indicates that the input value for `r` is a Kendall's *tau* correlation value
- `'Spearman'` — Indicates that the input value for `r` is a Spearman's *rho* rank correlation value

`copulaparam` uses an approximation to Spearman's rank correlation for copula families that do not have an existing analytic formula. The approximation is based on a smooth fit to values computed at discrete values of the copula parameters. For a *t* copula, the approximation is accurate for degrees of freedom larger than 0.05.

Example: `'Type', 'Spearman'`

## Output Arguments

### **rho** — Linear correlation parameter

scalar value | matrix of scalar values

Linear correlation parameter, returned as a scalar value or matrix of scalar values.

- If `r` is a scalar correlation coefficient, then `rho` is a scalar correlation coefficient corresponding to a bivariate copula.
- If `r` is a *p*-by-*p* correlation matrix, then `rho` is a *p*-by-*p* correlation matrix.

### **alpha** — Bivariate Archimedian copula parameter

scalar value

Bivariate Archimedean copula parameter, returned as a scalar value. Permitted values for alpha depend on the specified copula family.

Copula Family	Permitted Alpha Values
'Clayton'	$[0, \infty)$
'Frank'	$(-\infty, \infty)$
'Gumbel'	$[1, \infty)$

Data Types: `single` | `double`

## See Also

`copulacdf` | `copulafit` | `copulapdf` | `copularnd` | `copulastat` | `ecdf`

## Topics

“Generate Correlated Data Using Rank Correlation” on page 5-142

“Copulas: Generate Correlated Samples” on page 5-164

Introduced in R2006a

## copulapdf

Copula probability density function

### Syntax

```
y = copulapdf('Gaussian',u,rho)
```

```
y = copulapdf('t',u,rho,nu)
```

```
y = copulapdf(family,u,alpha)
```

### Description

`y = copulapdf('Gaussian',u,rho)` returns the probability density of the Gaussian copula with linear correlation parameters, `rho`, evaluated at the points in `u`.

`y = copulapdf('t',u,rho,nu)` returns the probability density of the  $t$  copula with linear correlation parameters, `rho`, and degrees of freedom parameter, `nu`, evaluated at the points in `u`.

`y = copulapdf(family,u,alpha)` returns the probability density of the bivariate Archimedean copula of the type specified by `family`, with scalar parameter, `alpha`, evaluated at the points in `u`.

### Examples

#### Compute the Clayton Copula pdf

Define two 10-by-10 matrices containing the values at which to compute the pdf.

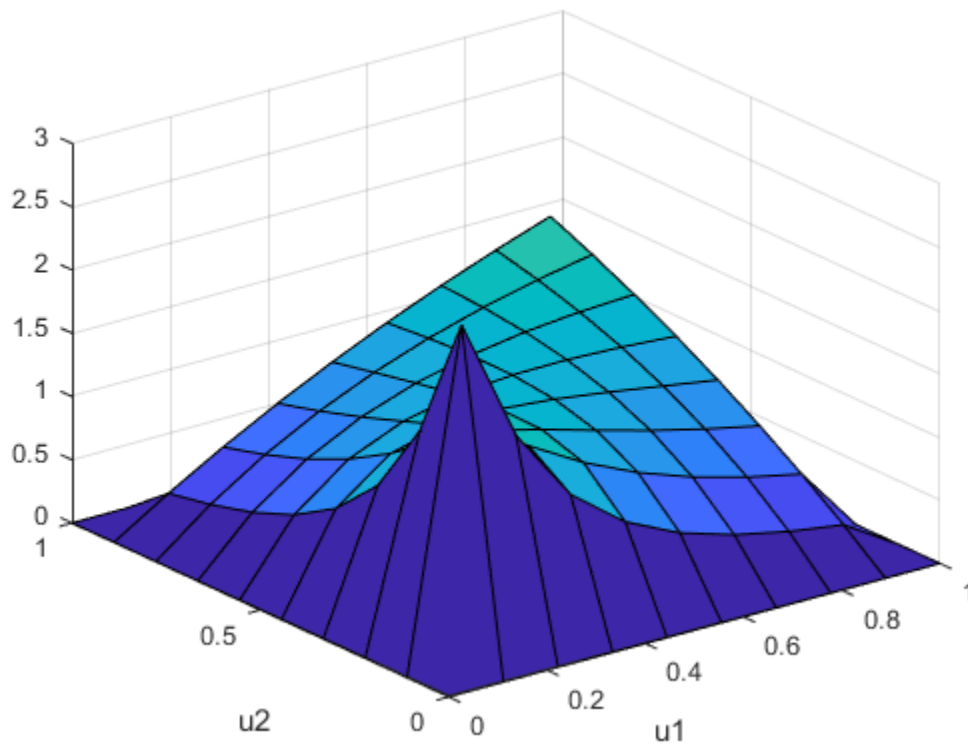
```
u = linspace(0,1,10);  
[u1,u2] = meshgrid(u,u);
```

Compute the pdf of a Clayton copula that has an alpha parameter equal to 1, at the values in u.

```
y = copulapdf('Clayton', [u1(:), u2(:)], 1);
```

Plot the pdf as a surface, and label the axes.

```
surf(u1, u2, reshape(y, 10, 10))  
xlabel('u1')  
ylabel('u2')
```



- “Generate Correlated Data Using Rank Correlation” on page 5-142

## Input Arguments

### **u** — Values at which to evaluate pdf

matrix of scalar values in the range [0,1]

Values at which to evaluate the pdf, specified as a matrix of scalar values in the range [0,1]. If *u* is an *n*-by-*p* matrix, then its values represent *n* points in the *p*-dimensional unit hypercube. If *u* is an *n*-by-2 matrix, then its values represent *n* points in the unit square.

If you specify a bivariate Archimedean copula type ('Clayton', 'Frank', or 'Gumbel'), then *u* must be an *n*-by-2 matrix.

Data Types: `single` | `double`

### **rho** — Linear correlation parameters

scalar values | matrix of scalar values

Linear correlation parameters for the copula, specified as a scalar value or matrix of scalar values.

- If *u* is an *n*-by-*p* matrix, then *rho* is a *p*-by-*p* correlation matrix.
- If *u* is an *n*-by-2 matrix, then *rho* can be a scalar correlation coefficient.

Data Types: `single` | `double`

### **nu** — Degrees of freedom

positive integer value

Degrees of freedom for the *t* copula, specified as a positive integer value.

Data Types: `single` | `double`

### **family** — Bivariate Archimedean copula family

'Clayton' | 'Frank' | 'Gumbel'

Bivariate Archimedean copula family, specified as one of the following.

'Clayton'	Clayton copula
'Frank'	Frank copula
'Gumbel'	Gumbel copula



Data Types: `single` | `double`

### **alpha** — Bivariate Archimedean copula parameter

scalar value

Bivariate Archimedean copula parameter, specified as a scalar value. Permitted values for alpha depend on the specified copula family.

Copula Family	Permitted Alpha Values
'Clayton'	$[0, \infty)$
'Frank'	$(-\infty, \infty)$
'Gumbel'	$[1, \infty)$

Data Types: `single` | `double`

## Output Arguments

### **y** — Probability density function

vector of scalar values

Probability density function, evaluated at the values in `u`, returned as a vector of scalar values.

## See Also

`copulacdf` | `copulaparam` | `copularnd` | `copulastat`

## Topics

“Generate Correlated Data Using Rank Correlation” on page 5-142

“Copulas: Generate Correlated Samples” on page 5-164

Introduced in R2006a

## copulastat

Copula rank correlation

### Syntax

```
r = copulastat('Gaussian', rho)
```

```
r = copulastat('t', rho, nu)
```

```
r = copulastat(family, alpha)
```

```
r = copulstat( ____, Name, Value)
```

### Description

`r = copulastat('Gaussian', rho)` returns the Kendall's rank correlation,  $r$ , that corresponds to a Gaussian copula with linear correlation parameters  $\rho$ .

`r = copulastat('t', rho, nu)` returns the Kendall's rank correlation,  $r$ , that corresponds to a  $t$  copula with linear correlation parameters,  $\rho$ , and degrees of freedom parameter,  $\nu$ .

`r = copulastat(family, alpha)` returns the Kendall's rank correlation,  $r$ , that corresponds to a bivariate Archimedean copula that has the type specified by `family` and scalar parameter `alpha`.

`r = copulstat( ____, Name, Value)` returns the copula rank correlation with additional options specified by one or more `Name, Value` pair arguments, using any of the previous syntaxes. For example, you can return Spearman's  $\rho$  rank correlation.

### Examples

## Compute the Gaussian Copula Rank Correlation

Compute the rank correlation for a Gaussian copula with the specified linear correlation parameter `rho`.

```
rho = -.7071;
tau = copulastat('gaussian',rho)
```

```
tau =
    -0.5000
```

Use the copula to generate dependent random values from a beta distribution that has parameters `a` and `b` equal to 2.

```
rng default % For reproducibility
u = copularnd('gaussian',rho,100);
b = betainv(u,2,2);
```

Verify that the sample has a rank correlation approximately equal to `tau`.

```
tau_sample = corr(b,'type','k')
```

```
tau_sample =
    1.0000    -0.5135
   -0.5135    1.0000
```

- “Generate Correlated Data Using Rank Correlation” on page 5-142

## Input Arguments

### **rho** — Linear correlation parameters

scalar values | matrix of scalar values

Linear correlation parameters for the copula, specified as a scalar value or matrix of scalar values.

- If `rho` is a scalar correlation coefficient, then `r` is a scalar correlation coefficient corresponding to a bivariate copula.
- If `rho` is a  $p$ -by- $p$  correlation matrix, then `r` is a  $p$ -by- $p$  correlation matrix.

Data Types: `single` | `double`

### **nu** — Degrees of freedom

positive integer value

Degrees of freedom for the  $t$  copula, specified as a positive integer value.

Data Types: `single` | `double`

### **family** — Bivariate Archimedean copula family

'Clayton' | 'Frank' | 'Gumbel'

Bivariate Archimedean copula family, specified as one of the following.

'Clayton'	Clayton copula
'Frank'	Frank copula
'Gumbel'	Gumbel copula

Data Types: `single` | `double`

### **alpha** — Bivariate Archimedean copula parameter

scalar value

Bivariate Archimedean copula parameter, specified as a scalar value. Permitted values for `alpha` depend on the specified copula family.

Copula Family	Permitted Alpha Values
'Clayton'	$[0, \infty)$
'Frank'	$(-\infty, \infty)$
'Gumbel'	$[1, \infty)$

Data Types: `single` | `double`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single

quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'Type', 'Spearman'` computes Spearman's rank correlation.

### **Type** — Type of rank correlation

`'Kendall'` (default) | `'Spearman'`

Type of rank correlation, specified as the comma-separated pair consisting of `'Type'` and one of the following.

- `'Kendall'` — Compute Kendall's  $\tau$ .
- `'Spearman'` — Compute Spearman's  $\rho$  (rank correlation).

`copulastat` uses an approximation to Spearman's rank correlation for copula families that do not have an existing analytic formula. The approximation is based on a smooth fit to values computed at discrete values of the copula parameters. For a  $t$  copula, the approximation is accurate for degrees of freedom larger than 0.05.

Example: `'Type', 'Spearman'`

## Output Arguments

### **r** — Copula rank correlation

scalar value | matrix of scalar values

Copula rank correlation, returned as a scalar value or matrix of scalar values.

- If  $\rho$  is a scalar correlation coefficient, then  $r$  is a scalar correlation coefficient corresponding to a bivariate copula.
- If  $\rho$  is a  $p$ -by- $p$  correlation matrix, then  $r$  is a  $p$ -by- $p$  correlation matrix.

## See Also

`copulacdf` | `copulaparam` | `copulapdf` | `copularnd`

## Topics

“Generate Correlated Data Using Rank Correlation” on page 5-142

“Copulas: Generate Correlated Samples” on page 5-164

**Introduced in R2006a**

# copularnd

Copula random numbers

## Syntax

```
u = copularnd('Gaussian',rho,n)
```

```
u = copularnd('t',rho,nu,n)
```

```
u = copularnd(family,alpha,n)
```

## Description

`u = copularnd('Gaussian',rho,n)` returns  $n$  random vectors generated from a Gaussian copula with linear correlation parameters `rho`.

`u = copularnd('t',rho,nu,n)` returns  $n$  random vectors generated from a  $t$  copula with linear correlation parameters `rho` and degrees of freedom `nu`.

`u = copularnd(family,alpha,n)` returns  $n$  random vectors generated from a bivariate Archimedean copula that has the type specified by `family` and the scalar parameter `alpha`.

## Examples

### Generate Correlated Data Using the Inverse cdf

Generate correlated random data from a beta distribution using a bivariate Gaussian copula with Kendall's  $\tau$  rank correlation equal to -0.5.

Compute the linear correlation parameter from the rank correlation value.

```
rng default % For reproducibility
tau = -0.5;
rho = copulaparam('Gaussian',tau)

rho =

    -0.7071
```

Use a Gaussian copula to generate a two-column matrix of dependent random values.

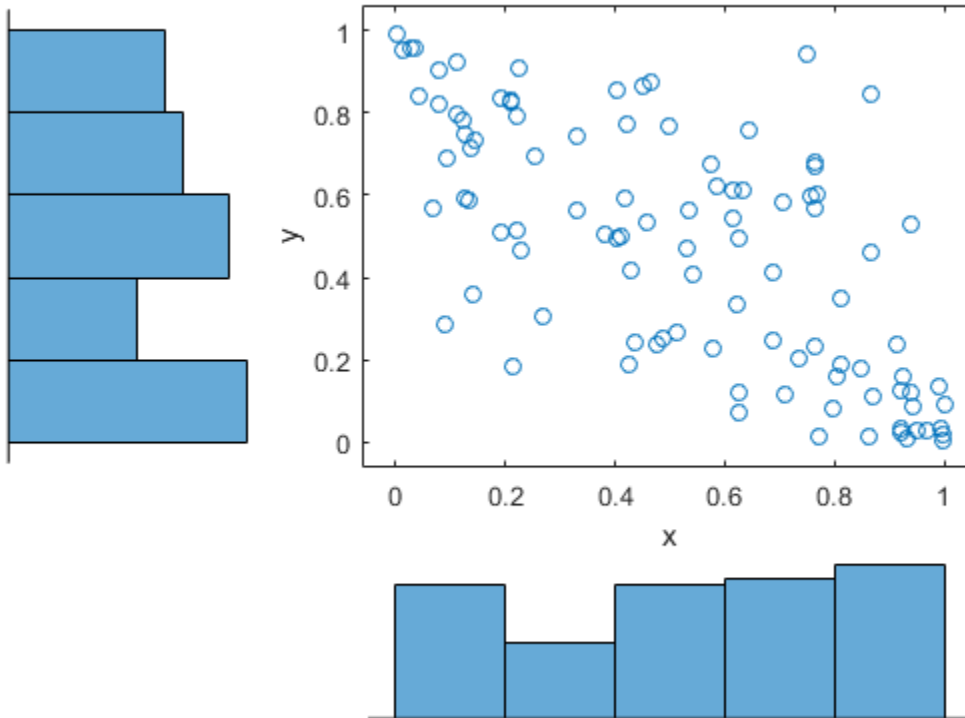
```
u = copularnd('gaussian',rho,100);
```

Each column contains 100 random values between 0 and 1, inclusive, sampled from a continuous uniform distribution.

Create a scatterhist plot to visualize the random numbers generated using the copula.

```
figure
scatterhist(u(:,1),u(:,2))
```





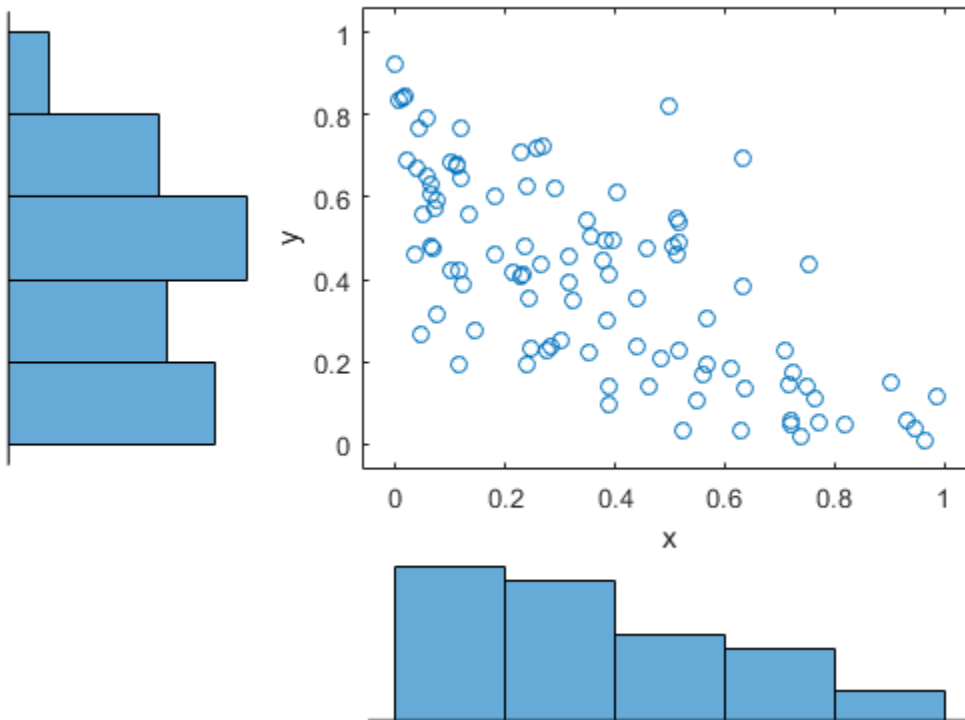
The histograms show that the data in each column of the copula has a marginal uniform distribution. The scatterplot shows that the data in the two columns is negatively correlated.

Use the inverse cdf function `betainv` to transform each column of the uniform marginal distributions into random numbers from a beta distribution. In the first column, the first shape parameter  $A$  is equal to 1, and a second shape parameter  $B$  is equal to 2. In the second column, the first shape parameter  $A$  is equal to 1.5, and a second shape parameter  $B$  is equal to 2.

```
b = [betainv(u(:,1),1,2), betainv(u(:,2),1.5,2)];
```

Create a `scatterhist` plot to visualize the correlated beta distribution data.

```
figure
scatterhist(b(:,1),b(:,2))
```



The histograms show the marginal beta distributions for each variable. The scatterplot shows the negative correlation.

Verify that the sample has a rank correlation approximately equal to the initial value for Kendall's *tau*.

```
tau_sample = corr(b, 'type', 'kendall')
```

```
tau_sample =
```

```

1.0000    -0.5135
-0.5135    1.0000

```

The sample rank correlation of -0.5135 is approximately equal to the -0.5 initial value for *tau*.

- “Generate Correlated Data Using Rank Correlation” on page 5-142

## Input Arguments

### **rho** — Linear correlation parameters

scalar values | matrix of scalar values

Linear correlation parameters for the copula, specified as a scalar value or matrix of scalar values.

- If *rho* is a *p*-by-*p* correlation matrix, then the output argument *u* is an *n*-by-*p* matrix.
- If *rho* is a scalar correlation coefficient, then the output argument *u* is an *n*-by-2 matrix.

Data Types: `single` | `double`

### **n** — Number of random vectors to return

positive scalar value

Number of random vectors to return, specified as a positive scalar value.

- If you specify the copula type as 'Gaussian' or 't', and *rho* is a *p*-by-*p* correlation matrix, then *u* is an *n*-by-*p* matrix.
- If you specify the copula type as 'Gaussian' or 't', and *rho* is a scalar correlation coefficient, then *u* is an *n*-by-2 matrix.
- If you specify the copula type as 'Clayton', 'Frank', or 'Gumbel', then *u* is an *n*-by-2 matrix.

Data Types: `single` | `double`

### **nu** — Degrees of freedom

positive integer value

Degrees of freedom for the  $t$  copula, specified as a positive integer value.

Data Types: `single` | `double`

**family** — Bivariate Archimedean copula family

'Clayton' | 'Frank' | 'Gumbel'

Bivariate Archimedean copula family, specified as one of the following.

'Clayton'	Clayton copula
'Frank'	Frank copula
'Gumbel'	Gumbel copula

Data Types: `single` | `double`

**alpha** — Bivariate Archimedean copula parameter

scalar value

Bivariate Archimedean copula parameter, specified as a scalar value. Permitted values for alpha depend on the specified copula family.

Copula Family	Permitted Alpha Values
'Clayton'	$[0, \infty)$
'Frank'	$(-\infty, \infty)$
'Gumbel'	$[1, \infty)$

Data Types: `single` | `double`

## Output Arguments

**u** — Copula random numbers

matrix of scalar values

Copula random numbers, returned as a matrix of scalar values. Each column of `u` is a sample from a `Uniform(0, 1)` marginal distribution.

- If you specify the copula type as 'Gaussian' or 't', and `rho` is a  $p$ -by- $p$  correlation matrix, then `u` is an  $n$ -by- $p$  matrix.
- If you specify the copula type as 'Gaussian' or 't', and `rho` is a scalar correlation coefficient, then `u` is an  $n$ -by-2 matrix.

- If you specify the copula type as 'Clayton', 'Frank', or 'Gumbel', then `u` is an  $n$ -by-2 matrix.

## See Also

`copulacdf` | `copulaparam` | `copulapdf` | `copulastat`

## Topics

“Generate Correlated Data Using Rank Correlation” on page 5-142

“Copulas: Generate Correlated Samples” on page 5-164

**Introduced in R2006a**

## cordexch

Coordinate exchange

### Syntax

```
dCE = cordexch(nfactors, nruns)
[dCE, X] = cordexch(nfactors, nruns)
[dCE, X] = cordexch(nfactors, nruns, 'model')
[dCE, X] = cordexch(..., 'name', value)
```

### Description

`dCE = cordexch(nfactors, nruns)` uses a coordinate-exchange algorithm to generate a  $D$ -optimal design `dCE` with `nruns` runs (the rows of `dCE`) for a linear additive model with `nfactors` factors (the columns of `dCE`). The model includes a constant term.

`[dCE, X] = cordexch(nfactors, nruns)` also returns the associated design matrix `X`, whose columns are the model terms evaluated at each treatment (row) of `dCE`.

`[dCE, X] = cordexch(nfactors, nruns, 'model')` uses the linear regression model specified in `model`. `model` is one of the following:

- 'linear' — Constant and linear terms. This is the default.
- 'interaction' — Constant, linear, and interaction terms
- 'quadratic' — Constant, linear, interaction, and squared terms
- 'purequadratic' — Constant, linear, and squared terms

The order of the columns of `X` for a full quadratic model with  $n$  terms is:

- 1 The constant term
- 2 The linear terms in order 1, 2, ...,  $n$
- 3 The interaction terms in order (1, 2), (1, 3), ..., (1,  $n$ ), (2, 3), ..., ( $n - 1$ ,  $n$ )
- 4 The squared terms in order 1, 2, ...,  $n$

Other models use a subset of these terms, in the same order.

Alternatively, *model* can be a matrix specifying polynomial terms of arbitrary order. In this case, *model* should have one column for each factor and one row for each term in the model. The entries in any row of *model* are powers for the factors in the columns. For example, if a model has factors  $X_1$ ,  $X_2$ , and  $X_3$ , then a row  $[0 \ 1 \ 2]$  in *model* specifies the term  $(X_1.^0) .* (X_2.^1) .* (X_3.^2)$ . A row of all zeros in *model* specifies a constant term, which can be omitted.

`[dCE,X] = cordexch(...,'name',value)` specifies one or more optional name/value pairs for the design. Valid parameters and their values are listed in the following table. Specify *name* inside single quotes.

name	Value
bounds	Lower and upper bounds for each factor, specified as a 2-by- <code>nfactors</code> matrix. Alternatively, this value can be a cell array containing <code>nfactors</code> elements, each element specifying the vector of allowable values for the corresponding factor.
categorical	Indices of categorical predictors.
display	Either 'on' or 'off' to control display of the iteration counter. The default is 'on'.
excldefun	Handle to a function that excludes undesirable runs. If the function is $f$ , it must support the syntax $b = f(S)$ , where $S$ is a matrix of treatments with <code>nfactors</code> columns and $b$ is a vector of Boolean values with the same number of rows as $S$ . $b(i)$ is true if the method should exclude $i$ th row $S$ .
init	Initial design as a <code>nruns</code> -by- <code>nfactors</code> matrix. The default is a randomly selected set of points.
levels	Vector of number of levels for each factor. Not used when bounds is specified as a cell array.
maxiter	Maximum number of iterations. The default is 10.
tries	Number of times to try to generate a design from a new starting point. The algorithm uses random points for each try, except possibly the first. The default is 1.

name	Value
options	<p>A structure that specifies whether to run in parallel, and specifies the random stream or streams. Create the options structure with <code>statset</code>. Option fields:</p> <ul style="list-style-type: none"> <li>• <code>UseParallel</code> — Set to <code>true</code> to compute in parallel. Default is <code>false</code>.</li> <li>• <code>UseSubstreams</code> — Set to <code>true</code> to compute in parallel in a reproducible fashion. Default is <code>false</code>. To compute reproducibly, set <code>Streams</code> to a type allowing substreams: <code>'mlfg6331_64'</code> or <code>'mrg32k3a'</code>.</li> <li>• <code>Streams</code> — A <code>RandStream</code> object or cell array of such objects. If you do not specify <code>Streams</code>, <code>cordexch</code> uses the default stream or streams. If you choose to specify <code>Streams</code>, use a single object except in the case <ul style="list-style-type: none"> <li>• You have an open Parallel pool</li> <li>• <code>UseParallel</code> is <code>true</code></li> <li>• <code>UseSubstreams</code> is <code>false</code></li> </ul> </li> </ul> <p>In that case, use a cell array the same size as the Parallel pool.</p>

## Examples

Suppose you want a design to estimate the parameters in the following three-factor, seven-term interaction model:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + \beta_{12} x_1 x_2 + \beta_{13} x_1 x_3 + \beta_{23} x_2 x_3 + \varepsilon$$

Use `cordexch` to generate a *D*-optimal design with seven runs:

```
nfactors = 3;
nruns = 7;
[dCE,X] = cordexch(nfactors,nruns,'interaction','tries',10)
dCE =
    -1     1     1
    -1    -1    -1
     1     1     1
```



```

      -1      1      -1
      1      -1      1
      1      -1      -1
      -1     -1      1
X =
      1     -1      1      1     -1     -1      1
      1     -1     -1     -1      1      1      1
      1      1      1      1      1      1      1
      1     -1      1     -1     -1      1     -1
      1      1     -1      1     -1      1     -1
      1      1     -1     -1     -1     -1      1
      1     -1     -1      1      1     -1     -1

```

Columns of the design matrix  $X$  are the model terms evaluated at each row of the design dCE. The terms appear in order from left to right: constant term, linear terms (1, 2, 3), interaction terms (12, 13, 23). Use  $X$  to fit the model, as described in “Linear Regression” on page 11-11, to response data measured at the design points in dCE.

## Algorithms

Both `cordexch` and `rowexch` use iterative search algorithms. They operate by incrementally changing an initial design matrix  $X$  to increase  $D = |X^T X|$  at each step. In both algorithms, there is randomness built into the selection of the initial design and into the choice of the incremental changes. As a result, both algorithms may return locally, but not globally,  $D$ -optimal designs. Run each algorithm multiple times and select the best result for your final design. Both functions have a `'tries'` parameter that automates this repetition and comparison.

Unlike the row-exchange algorithm used by `rowexch`, `cordexch` does not use a candidate set. (Or rather, the candidate set is the entire design space.) At each step, the coordinate-exchange algorithm exchanges a single element of  $X$  with a new element evaluated at a neighboring point in design space. The absence of a candidate set reduces demands on memory, but the smaller scale of the search means that the coordinate-exchange algorithm is more likely to become trapped in a local minimum.

## See Also

`daugment` | `dcovary` | `rowexch`

**Introduced before R2006a**

## corr

Linear or rank correlation

### Syntax

```
RHO = corr(X)
RHO = corr(X,Y)
[RHO,PVAL] = corr(X,Y)
[RHO,PVAL] = corr(X,Y,'name',value)
```

### Description

`RHO = corr(X)` returns a  $p$ -by- $p$  matrix containing the pairwise linear correlation coefficient between each pair of columns in the  $n$ -by- $p$  matrix  $X$ .

`RHO = corr(X,Y)` returns a  $p1$ -by- $p2$  matrix containing the pairwise correlation coefficient between each pair of columns in the  $n$ -by- $p1$  and  $n$ -by- $p2$  matrices  $X$  and  $Y$ .

The difference between `corr(X,Y)` and the MATLAB function `corrcoef(X,Y)` is that `corrcoef(X,Y)` returns a matrix of correlation coefficients for the two column vectors  $X$  and  $Y$ . If  $X$  and  $Y$  are not column vectors, `corrcoef(X,Y)` converts them to column vectors.

`[RHO,PVAL] = corr(X,Y)` also returns `PVAL`, a matrix of  $p$ -values for testing the hypothesis of no correlation against the alternative that there is a nonzero correlation. Each element of `PVAL` is the  $p$  value for the corresponding element of `RHO`. If `PVAL(i,j)` is small, say less than 0.05, then the correlation `RHO(i,j)` is significantly different from zero.

`[RHO,PVAL] = corr(X,Y,'name',value)` specifies one or more optional name/value pairs. Specify `name` inside single quotes. The following table lists valid parameters and their values.

Parameter	Values
type	<ul style="list-style-type: none"> <li>• 'Pearson' (the default) computes Pearson's linear correlation coefficient</li> <li>• 'Kendall' computes Kendall's tau</li> <li>• 'Spearman' computes Spearman's rho</li> </ul>
rows	<ul style="list-style-type: none"> <li>• 'all' (the default) uses all rows regardless of missing values (NaNs)</li> <li>• 'complete' uses only rows with no missing values</li> <li>• 'pairwise' computes <math>RHO(i, j)</math> using rows with no missing values in column <math>i</math> or <math>j</math></li> </ul>
tail — The alternative hypothesis against which to compute $p$ -values for testing the hypothesis of no correlation	<ul style="list-style-type: none"> <li>• 'both' — Correlation is not zero (the default)</li> <li>• 'right' — Correlation is greater than zero</li> <li>• 'left' — Correlation is less than zero</li> </ul>

Using the 'pairwise' option for the rows parameter may return a matrix that is not positive definite. The 'complete' option always returns a positive definite matrix, but in general the estimates are based on fewer observations.

`corr` computes  $p$ -values for Pearson's correlation using a Student's  $t$  distribution for a transformation of the correlation. This correlation is exact when  $X$  and  $Y$  are normal. `corr` computes  $p$ -values for Kendall's tau and Spearman's rho using either the exact permutation distributions (for small sample sizes), or large-sample approximations.

`corr` computes  $p$ -values for the two-tailed test by doubling the more significant of the two one-tailed  $p$ -values.

## Examples

## Find Correlation Between Two Matrices

Find the correlation between two matrices and compare to the correlation between two column vectors.

Generate sample data.

```
rng('default')
x = randn(30,4);
y = randn(30,4);
y(:,4) = sum(x,2); % introduce correlation
```

Calculate the correlation between columns of X and Y.

```
[r,p] = corr(x,y)
```

r =

```
-0.1686    -0.0363     0.2278     0.6901
 0.3022     0.0332    -0.0866     0.2617
-0.3632    -0.0987    -0.0200     0.3504
-0.1365    -0.1804     0.0853     0.4908
```

p =

```
 0.3731     0.8489     0.2260     0.0000
 0.1045     0.8619     0.6491     0.1624
 0.0485     0.6039     0.9166     0.0577
 0.4721     0.3400     0.6539     0.0059
```

Calculate the correlation between X and Y using `corrcoef`.

```
[r,p] = corrcoef(x,y)
```

r =

```
 1.0000     0.1252
 0.1252     1.0000
```

p =

```
1.0000    0.1729
0.1729    1.0000
```

MATLAB® function `corrcoef` converts  $X$  and  $Y$  into column vectors before computing the correlation between them.

## References

- [1] Gibbons, J.D. (1985) Nonparametric Statistical Inference, 2nd ed., M. Dekker.
- [2] Hollander, M. and D.A. Wolfe (1973) Nonparametric Statistical Methods, Wiley.
- [3] Kendall, M.G. (1970) Rank Correlation Methods, Griffin.
- [4] Best, D.J. and D.E. Roberts (1975) "Algorithm AS 89: The Upper Tail Probabilities of Spearman's rho", Applied Statistics, 24:377-379.

## Extended Capabilities

### Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function supports tall arrays for out-of-memory data with the limitation:

Only 'Pearson' type is supported.

For more information, see “Tall Arrays” (MATLAB).

### See Also

`corrcoef` | `corrcoef` | `partialcorr` | `tiedrank`

Introduced before R2006a

## corrcoef

Convert covariance matrix to correlation matrix

## Syntax

```
R = corrcoef(C)
[R,sigma] = corrcoef(C)
```

## Description

`R = corrcoef(C)` computes the correlation matrix `R` corresponding to the covariance matrix `C`. `C` must be square, symmetric, and positive semi-definite.

`[R,sigma] = corrcoef(C)` also computes the vector of standard deviations `sigma`.

## Examples

Use `cov` and `corrcoef` to compute covariances and correlations, respectively, for sample data on weight and blood pressure (systolic, diastolic) in `hospital.mat`:

```
load hospital
X = [hospital.Weight hospital.BloodPressure];
C = cov(X)
C =
    706.0404    27.7879    41.0202
    27.7879    45.0622    23.8194
    41.0202    23.8194    48.0590
R = corrcoef(X)
R =
    1.0000    0.1558    0.2227
    0.1558    1.0000    0.5118
    0.2227    0.5118    1.0000
```

Compare `R` with the correlation matrix computed from `C` by `corrcoef`:

```
corrcov(C)
ans =
    1.0000    0.1558    0.2227
    0.1558    1.0000    0.5118
    0.2227    0.5118    1.0000
```

## See Also

[cholcov](#) | [corr](#) | [corrcoef](#) | [cov](#)

**Introduced in R2007b**



## Cost property

**Class:** `TreeBagger`

Misclassification costs

## Description

The `Cost` property is a matrix with misclassification costs. This property is empty for ensembles of regression trees.

## See Also

`ClassificationTree` | `RegressionTree` | `TreeBagger` | `fitctree` | `fitrtree`

## covarianceParameters

**Class:** GeneralizedLinearMixedModel

Extract covariance parameters of generalized linear mixed-effects model

### Syntax

```
psi = covarianceParameters(glme)
[psi, dispersion] = covarianceParameters(glme)
[psi, dispersion, stats] = covarianceParameters(glme)
[ ___ ] = covarianceParameters(glme, Name, Value)
```

### Description

`psi = covarianceParameters(glme)` returns the estimated prior covariance parameters of random-effects predictors in the generalized linear mixed-effects model `glme`.

`[psi, dispersion] = covarianceParameters(glme)` also returns an estimate of the dispersion parameter.

`[psi, dispersion, stats] = covarianceParameters(glme)` also returns a cell array `stats` containing the covariance parameter estimates and related statistics.

`[ ___ ] = covarianceParameters(glme, Name, Value)` returns any of the above output arguments using additional options specified by one or more `Name, Value` pair arguments. For example, you can specify the confidence level for the confidence limits of covariance parameters.

### Input Arguments

**glme** — Generalized linear mixed-effects model

GeneralizedLinearMixedModel object

Generalized linear mixed-effects model, specified as a `GeneralizedLinearMixedModel` object. For properties and methods of this object, see `GeneralizedLinearMixedModel`.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### **Alpha** — Confidence level

0.05 (default) | scalar value in the range [0,1]

Confidence level, specified as the comma-separated pair consisting of 'Alpha' and a scalar value in the range [0,1]. For a value  $\alpha$ , the confidence level is  $100 \times (1 - \alpha)\%$ .

For example, for 99% confidence intervals, you can specify the confidence level as follows.

Example: 'Alpha', 0.01

Data Types: single | double

## Output Arguments

### **psi** — Estimated prior covariance parameters

cell array

Estimated prior covariance parameters for the random-effects predictors, returned as a cell array of length  $R$ , where  $R$  is the number of grouping variables used in the model. `psi{r}` contains the covariance matrix of random effects associated with grouping variable  $g_r$ , where  $r = 1, 2, \dots, R$ . The order of grouping variables in `psi` is the same as the order entered when fitting the model. For more information on grouping variables, see “Grouping Variables” on page 2-59.

### **dispersion** — Dispersion parameter

scalar value

Dispersion parameter, returned as a scalar value.

**stats — Covariance parameter estimates and related statistics**

cell array

Covariance parameter estimates and related statistics, returned as a cell array of length  $(R + 1)$ , where  $R$  is the number of grouping variables used in the model. The first  $R$  cells of `stats` each contain a dataset array with the following columns.

Column Name	Description
Group	Grouping variable name
Name1	Name of the first predictor variable
Name2	Name of the second predictor variable
Type	If Name1 and Name2 are the same, then Type is <code>std</code> (standard deviation).  If Name1 and Name2 are different, then Type is <code>corr</code> (correlation).
Estimate	If Name1 and Name2 are the same, then Estimate is the standard deviation of the random effect associated with predictor Name1 or Name2.  If Name1 and Name2 are different, then Estimate is the correlation between the random effects associated with predictors Name1 and Name2.
Lower	Lower limit of the confidence interval for the covariance parameter
Upper	Upper limit of the confidence interval for the covariance parameter

Cell  $R + 1$  contains related statistics for the dispersion parameter.

It is recommended that the presence or absence of covariance parameters in `glme` be tested using the `compare` method, which uses a likelihood ratio test.

When fitting a GLME model using `fitglme` and one of the maximum likelihood fit methods ('Laplace' or 'ApproximateLaplace'), `covarianceParameters` derives

the confidence intervals in `stats` based on a Laplace approximation to the log likelihood of the generalized linear mixed-effects model.

When fitting a GLME model using `fitglme` and one of the pseudo likelihood fit methods ('MPL' or 'REMP'), `covarianceParameters` derives the confidence intervals in `stats` based on the fitted linear mixed-effects model from the final pseudo likelihood iteration.

## Examples

### Obtain Estimated Covariance Parameters

Load the sample data.

```
load mfr
```

This simulated data is from a manufacturing company that operates 50 factories across the world, with each factory running a batch process to create a finished product. The company wants to decrease the number of defects in each batch, so it developed a new manufacturing process. To test the effectiveness of the new process, the company selected 20 of its factories at random to participate in an experiment: Ten factories implemented the new process, while the other ten continued to run the old process. In each of the 20 factories, the company ran five batches (for a total of 100 batches) and recorded the following data:

- Flag to indicate whether the batch used the new process (`newprocess`)
- Processing time for each batch, in hours (`time`)
- Temperature of the batch, in degrees Celsius (`temp`)
- Categorical variable indicating the supplier (A, B, or C) of the chemical used in the batch (`supplier`)
- Number of defects in the batch (`defects`)

The data also includes `time_dev` and `temp_dev`, which represent the absolute deviation of time and temperature, respectively, from the process standard of 3 hours at 20 degrees Celsius.

Fit a generalized linear mixed-effects model using `newprocess`, `time_dev`, `temp_dev`, and `supplier` as fixed-effects predictors. Include a random-effects term for intercept

grouped by `factory`, to account for quality differences that might exist due to factory-specific variations. The response variable `defects` has a Poisson distribution, and the appropriate link function for this model is `log`. Use the Laplace fit method to estimate the coefficients. Specify the dummy variable encoding as `'effects'`, so the dummy variable coefficients sum to 0.

The number of defects can be modeled using a Poisson distribution

$$\text{defects}_{ij} \sim \text{Poisson}(\mu_{ij})$$

This corresponds to the generalized linear mixed-effects model

$$\log(\mu_{ij}) = \beta_0 + \beta_1 \text{newprocess}_{ij} + \beta_2 \text{time.dev}_{ij} + \beta_3 \text{temp.dev}_{ij} + \beta_4 \text{supplier.C}_{ij} + \beta_5 \text{supplier.B}_{ij} + b_i,$$

where

- `defectsij` is the number of defects observed in the batch produced by factory *i* during batch *j*.
- `μij` is the mean number of defects corresponding to factory *i* (where *i* = 1, 2, ..., 20) during batch *j* (where *j* = 1, 2, ..., 5).
- `newprocessij`, `time.devij`, and `temp.devij` are the measurements for each variable that correspond to factory *i* during batch *j*. For example, `newprocessij` indicates whether the batch produced by factory *i* during batch *j* used the new process.
- `supplier.Cij` and `supplier.Bij` are dummy variables that use effects (sum-to-zero) coding to indicate whether company C or B, respectively, supplied the process chemicals for the batch produced by factory *i* during batch *j*.
- `bi ~ N(0, σb2)` is a random-effects intercept for each factory *i* that accounts for factory-specific variation in quality.

```
glme = fitglme(mfr, 'defects ~ 1 + newprocess + time_dev + temp_dev + supplier + (1|fact
```

Compute and display the estimate of the prior covariance parameter for the random-effects predictor.

```
[psi, dispersion, stats] = covarianceParameters(glme);
psi{1}
```

```
ans =
```

```
0.0985
```

`psi{1}` is an estimate of the prior covariance matrix of the first grouping variable. In this example, there is only one grouping variable (`factory`), so `psi{1}` is an estimate of  $\sigma_b^2$ .

Display the dispersion parameter.

```
dispersion
```

```
dispersion =
```

```
1
```

Display the estimated standard deviation of the random effect associated with the predictor. The first cell of `stats` contains statistics for `factory`, while the second cell contains statistics for the dispersion parameter.

```
stats{1}
```

```
ans =
```

```
COVARIANCE TYPE: ISOTROPIC
```

Group	Name1	Name2	Type	Estimate
factory	'(Intercept)'	'(Intercept)'	'std'	0.31381
	Lower	Upper		
	0.19253	0.51148		

The estimated standard deviation of the random effect associated with the predictor is 0.31381. The 95% confidence interval is [0.19253 , 0.51148]. Because the confidence

interval does not contain 0, the random intercept is significant at the 5% significance level.

## See Also

`GeneralizedLinearMixedModel` | `compare` | `fitglme` | `fixedEffects` | `randomEffects`



# covarianceParameters

**Class:** LinearMixedModel

Extract covariance parameters of linear mixed-effects model

## Syntax

```
psi = covarianceParameters(lme)
[psi,mse] = covarianceParameters(lme)
[psi,mse,stats] = covarianceParameters(lme)
[psi,mse,stats] = covarianceParameters(lme,Name,Value)
```

## Description

`psi = covarianceParameters(lme)` returns the estimated covariance parameters that parameterize the prior covariance of random effects.

`[psi,mse] = covarianceParameters(lme)` also returns an estimate of the residual variance.

`[psi,mse,stats] = covarianceParameters(lme)` also returns a cell array, `stats`, containing the covariance parameters and related statistics.

`[psi,mse,stats] = covarianceParameters(lme,Name,Value)` returns the covariance parameters and related statistics in `stats` with additional options specified by one or more `Name,Value` pair arguments.

For example, you can specify the confidence level for the confidence limits of covariance parameters.

## Input Arguments

**lme** — Linear mixed-effects model

LinearMixedModel object

Linear mixed-effects model, specified as a `LinearMixedModel` object constructed using `fitlme` or `fitlmematrix`.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### **Alpha** — Confidence level

0.05 (default) | scalar value in the range 0 to 1

Confidence level, specified as the comma-separated pair consisting of 'Alpha' and a scalar value in the range 0 to 1. For a value  $\alpha$ , the confidence level is  $100*(1-\alpha)\%$ .

For example, for 99% confidence intervals, you can specify the confidence level as follows.

Example: 'Alpha', 0.01

Data Types: `single` | `double`

## Output Arguments

### **psi** — Estimate of covariance parameters

cell array

Estimate of covariance parameters that parameterize the prior covariance of the random effects, returned as a cell array of length  $R$ , such that `psi{r}` contains the covariance matrix of random effects associated with grouping variable  $g_r$ ,  $r = 1, 2, \dots, R$ . The order of grouping variables is the same order you enter when you fit the model.

### **mse** — Residual variance estimate

scalar value

Residual variance estimate, returned as a scalar value.

### **stats** — Covariance parameter estimates and related statistics

cell array

Covariance parameter estimates and related statistics, returned as a cell array of length  $(R + 1)$  containing dataset arrays with the following columns.

Group	Grouping variable name
Name1	Name of the first predictor variable
Name2	Name of the second predictor variable
Type	std (standard deviation), if Name1 and Name2 are the same
	corr (correlation), if Name1 and Name2 are different
Estimate	Standard deviation of the random effect associated with predictor Name1 or Name2, if Name1 and Name2 are the same
	Correlation between the random effects associated with predictors Name1 and Name2, if Name1 and Name2 are different
Lower	Lower limit of a 95% confidence interval for the covariance parameter
Upper	Upper limit of a 95% confidence interval for the covariance parameter

`stats{r}` is a dataset array containing statistics on covariance parameters for the  $r$ th grouping variable,  $r = 1, 2, \dots, R$ . `stats{R+1}` contains statistics on the residual standard deviation. The dataset array for the residual error has the fields `Group`, `Name`, `Estimate`, `Lower`, and `Upper`.

## Examples

### Two Random-Effects Terms for Intercept

Load the sample data.

```
load(fullfile(matlabroot, 'examples', 'stats', 'fertilizer.mat'));
```

The dataset array includes data from a split-plot experiment, where soil is divided into three blocks based on the soil type: sandy, silty, and loamy. Each block is divided into

five plots, where five different types of tomato plants (cherry, heirloom, grape, vine, and plum) are randomly assigned to these plots. The tomato plants in the plots are then divided into subplots, where each subplot is treated by one of four fertilizers. This is simulated data.

Store the data in a dataset array called `ds`, for practical purposes, and define `Tomato`, `Soil`, and `Fertilizer` as categorical variables.

```
ds = fertilizer;  
ds.Tomato = nominal(ds.Tomato);  
ds.Soil = nominal(ds.Soil);  
ds.Fertilizer = nominal(ds.Fertilizer);
```

Fit a linear mixed-effects model, where `Fertilizer` is the fixed-effects variable, and the mean yield varies by the block (soil type), and the plots within blocks (tomato types within soil types) independently. This model corresponds to

$$y_{ijk} = \beta_0 + \sum_{j=2}^5 \beta_{2j} I[T]_{ij} + b_{0jk}(S * T)_{jk} + \epsilon_{ijk},$$

where  $i = 1, 2, \dots, 60$  corresponds to the observations,  $j = 2, \dots, 5$  corresponds to the tomato types, and  $k = 1, 2, 3$  corresponds to the blocks (soil).  $S_k$  represents the  $k$ th soil type, and  $(S * T)_{jk}$  represents the  $j$ th tomato type nested in the  $k$ th soil type.  $I[T]_{ij}$  is the dummy variable representing the level  $j$  of the tomato type.

The random effects and observation error have the following prior distributions:

$$b_{0k} \sim N(0, \sigma_S^2), b_{0jk} \sim N(0, \sigma_{S*T}^2), \text{ and } \epsilon_{ijk} \sim N(0, \sigma^2).$$

```
lme = fitlme(ds, 'Yield ~ Fertilizer + (1|Soil) + (1|Soil:Tomato)');
```

Compute the covariance parameter estimates (estimates of  $\sigma_S^2$  and  $\sigma_{S*T}^2$ ) of the random-effects terms.

```
psi = covarianceParameters(lme)
```

```
psi =
```

```
2x1 cell array
```

```
{[2.4628e-17]}
```

```
{[ 352.8481]}
```

Compute the residual variance ( $\sigma^2$ ).

```
[~,mse] = covarianceParameters(lme)
```

```
mse =
```

```
151.9007
```

## Potentially Correlated Random-Effects Terms

Load the sample data.

```
load(fullfile(matlabroot, 'examples', 'stats', 'weight.mat'));
```

`weight` contains data from a longitudinal study, where 20 subjects are randomly assigned to 4 exercise programs, and their weight loss is recorded over six 2-week time periods. This is simulated data.

Store the data in a dataset array. Define `Subject` and `Program` as categorical variables.

```
ds = dataset(InitialWeight, Program, Subject, Week, y);
ds.Subject = nominal(ds.Subject);
ds.Program = nominal(ds.Program);
```

Fit a linear mixed-effects model where the initial weight, type of program, week, and the interaction between the week and type of program are the fixed effects. The intercept and week vary by subject.

For 'reference' dummy variable coding, `fitlme` uses Program A as reference and creates the necessary dummy variables  $I[\cdot]$ . This model corresponds to

$$y_{im} = \beta_0 + \beta_1 IW_i + \beta_2 \text{Week}_i + \beta_3 I[PB]_i + \beta_4 I[PC]_i \\ + \beta_5 I[PD]_i + b_{0m} + b_{1m} \text{Week}_{im} + \epsilon_{im}$$

where  $i$  corresponds to the observation number,  $i = 1, 2, \dots, 120$ , and  $m$  corresponds to the subject number,  $m = 1, 2, \dots, 20$ .  $\beta_j$  are the fixed-effects coefficients,  $j = 0, 1, \dots, 8$ ,

and  $b_{0m}$  and  $b_{1m}$  are random effects.  $IW$  stands for initial weight and  $I[\cdot]$  is a dummy variable representing a type of program. For example,  $I[PB]_i$  is the dummy variable representing Program B.

The random effects and observation error have the following prior distributions:

$$\begin{pmatrix} b_{0m} \\ b_{1m} \end{pmatrix} \sim N\left(0, \begin{pmatrix} \sigma_0^2 & \sigma_{0,1} \\ \sigma_{0,1} & \sigma_1^2 \end{pmatrix}\right)$$

and

$$\epsilon_{im} \sim N(0, \sigma^2).$$

```
lme = fitlme(ds, 'y ~ InitialWeight + Program + (Week|Subject)');
```

Compute the estimates of covariance parameters for the random effects.

```
[psi, mse, stats] = covarianceParameters(lme)
```

```
psi =
```

```
1x1 cell array  
  
{2x2 double}
```

```
mse =
```

```
0.0105
```

```
stats =
```

```
2x1 cell array  
  
{3x7 classreg.regr.lmeutils.titledataset}  
{1x5 classreg.regr.lmeutils.titledataset}
```

`mse` is the estimated residual variance. It is the estimate for  $\sigma^2$ .

To see the covariance parameters estimates for the random-effects terms ( $\sigma_0^2$ ,  $\sigma_1^2$ , and  $\sigma_{0,1}$ ), index into `psi`.

```
psi{1}
```

```
ans =
```

```
    0.0572    0.0490
    0.0490    0.0624
```

The estimate of the variance of the random effects term for the intercept,  $\sigma_0^2$ , is 0.0572. The estimate of the variance of the random effects term for week,  $\sigma_1^2$ , is 0.0624. The estimate for the covariance of the random effects terms for the intercept and week,  $\sigma_{0,1}$ , is 0.0490.

`stats` is a 2-by-1 cell array. The first cell of `stats` contains the confidence intervals for the standard deviation of the random effects and the correlation between the random effects for intercept and week. To display them, index into `stats`.

```
stats{1}
```

```
ans =
```

```
COVARIANCE TYPE: FULLCHOLESKY
```

Group	Name1	Name2	Type	Estimate
Subject	'(Intercept)'	'(Intercept)'	'std'	0.23927
Subject	'Week'	'(Intercept)'	'corr'	0.81971
Subject	'Week'	'Week'	'std'	0.2497

Lower	Upper
0.14365	0.39854
0.38662	0.95658
0.18303	0.34067

The display shows the name of the grouping parameter (`Group`), the random-effects variables (`Name1`, `Name2`), the type of the covariance parameters (`Type`), the estimate (`Estimate`) for each parameter, and the 95% confidence intervals for the parameters (`Lower`, `Upper`). The estimates in this table are related to the estimates in `psi` as follows.

The standard deviation of the random-effects term for intercept is  $0.23927 = \sqrt{0.0527}$ . Likewise, the standard deviation of the random effects term for week is  $0.2497 = \sqrt{0.0624}$ . Finally, the correlation between the random-effects terms of intercept and week is  $0.81971 = 0.0490 / (0.23927 * 0.2497)$ .

Note that this display also shows which covariance pattern you use when fitting the model. In this case, the covariance pattern is `FullCholesky`. To change the covariance pattern for the random-effects terms, you must use the `'CovariancePattern'` name-value pair argument when fitting the model.

The second cell of `stats` includes similar statistics for the residual standard deviation. Display the contents of the second cell.

```
stats{2}
```

```
ans =
```

Group	Name	Estimate	Lower	Upper
Error	'Res Std'	0.10261	0.087882	0.11981

The estimate for residual standard deviation is the square root of `mse`,  $0.10261 = \sqrt{0.0105}$ .

## Two Grouping Variables

Load the sample data.

```
load carbig
```

Fit a linear mixed-effects model for miles per gallon (MPG), with fixed effects for acceleration and weight, a potentially correlated random effect for intercept and acceleration grouped by model year, and an independent random effect for weight, grouped by the origin of the car. This model corresponds to

$$MPG_{imk} = \beta_0 + \beta_1 Acc_i + \beta_2 Weight_i + b_{10m} + b_{11m} Acc_i + b_{21k} Weight_i + \epsilon_{imk}$$

where  $m = 1, 2, \dots, 13$  represents the levels for the variable `Model_Year`, and  $k = 1, 2, \dots, 8$  represents the levels for the variable `Origin`.  $MPG_{imk}$  is the miles per gallon for the  $i$ th observation,  $|m|$  th model year, and  $|k|$  th origin that correspond to the



ith observation. The random-effects terms and the observation error have the following prior distributions:

$$b_{1m} = \begin{pmatrix} b_{10m} \\ b_{11m} \end{pmatrix} \sim N \left( 0, \begin{pmatrix} \sigma_{10}^2 & \sigma_{10,11} \\ \sigma_{10,11} & \sigma_{11}^2 \end{pmatrix} \right),$$

$$b_{2k} \sim N(0, \sigma_2^2),$$

$$\epsilon_{imk} \sim N(0, \sigma^2).$$

Here, the random-effects term  $b_{1m}$  represents the first random effect at level  $m$  of the first grouping variable. The random-effects term  $b_{10m}$  corresponds to the first random effects term (1), for the intercept (0), at the  $m$ th level ( $m$ ) of the first grouping variable. Likewise  $b_{11m}$  is the level  $m$  for the first predictor (1) in the first random-effects term (1).

Similarly,  $b_{2k}$  stands for the second random effects-term at level  $k$  of the second grouping variable.

$\sigma_{10}^2$  is the variance of the random-effects term for the intercept,  $\sigma_{11}^2$  is the variance of the random effects term for the predictor acceleration, and  $\sigma_{10,11}$  is the covariance of the random-effects terms for the intercept and the predictor acceleration.  $\sigma_2^2$  is the variance of the second random-effects term, and  $\sigma^2$  is the residual variance.

First, prepare the design matrices for fitting the linear mixed-effects model.

```
X = [ones(406,1) Acceleration Weight];
Z = {[ones(406,1) Acceleration],[Weight]};
Model_Year = nominal(Model_Year);
Origin = nominal(Origin);
G = {Model_Year,Origin};
```

Fit the model using the design matrices.

```
lme = fitlmematrix(X,MPG,Z,G,'FixedEffectPredictors',...
{'Intercept','Acceleration','Weight'],'RandomEffectPredictors',...
{{'Intercept','Acceleration'},{'Weight'}},'RandomEffectGroups',{'Model_Year','Origin'})
```

Compute the estimates of covariance parameters for the random effects.

```
[psi,mse,stats] = covarianceParameters(lme)
```

```
psi =
```

```
2x1 cell array

    {2x2 double }
    {[6.7579e-08]}

mse =

    9.0802

stats =

3x1 cell array

    {3x7 classreg.regr.lmeutils.titleddataset}
    {1x7 classreg.regr.lmeutils.titleddataset}
    {1x5 classreg.regr.lmeutils.titleddataset}
```

The residual variance `mse` is 9.0755. `psi` is a 2-by-1 cell array, and `stats` is a 3-by-1 cell array. To see the contents, you must index into these cell arrays.

First, index into the first cell of `psi`.

```
psi{1}

ans =

    8.3081    -0.8359
   -0.8359     0.1093
```

The first cell of `psi` contains the covariance parameters for the correlated random effects for intercept  $\sigma_{10}^2$  as 8.5160, and for acceleration  $\sigma_{11}^2$  as 0.1087. The estimate for the covariance of the random-effects terms for the intercept and acceleration  $\sigma_{10,11}$  is -0.8387.

Now, index into the second cell of `psi`.

```
psi{2}

ans =
```

```
6.7579e-08
```

The second cell of `psi` contains the estimate for the variance of the random-effects term for weight  $\sigma_2^2$ .

Index into the first cell of `stats`.

```
stats{1}
```

```
ans =
```

```
COVARIANCE TYPE: FULLCHOLESKY
```

Group	Name1	Name2	Type
Model_Year	'Intercept'	'Intercept'	'std'
Model_Year	'Acceleration'	'Intercept'	'corr'
Model_Year	'Acceleration'	'Acceleration'	'std'

Estimate	Lower	Upper
2.8824	1.1116	7.4739
-0.87704	-0.98411	-0.30222
0.33066	0.18918	0.57796

This table shows the standard deviation estimates for the random-effects terms for intercept and acceleration. Note that the standard deviations estimates are the square roots of the diagonal elements in the first cell of `psi`. Specifically,  $2.9182 = \text{sqrt}(8.5160)$  and  $0.32968 = \text{sqrt}(0.1087)$ . The correlation is a function of the covariance of intercept and acceleration, and the standard deviations of intercept and acceleration. The covariance of intercept and acceleration is the off-diagonal value in the first cell of `psi`,  $-0.8387$ . So, the correlation is  $-0.8387 / (0.32968 * 2.92182) = -0.87$ .

The grouping variable for intercept and acceleration is `Model_Year`.

Index into the second cell of `stats`.

```
stats{2}
```

```
ans =
```

```
COVARIANCE TYPE: FULLCHOLESKY
```

Group	Name1	Name2	Type	Estimate
Origin	'Weight'	'Weight'	'std'	0.00025996

Lower	Upper
9.1721e-05	0.0007368

The second cell of `stats` has the standard deviation estimate and the 95% confidence limits for the standard deviation of the random-effects term for `Weight`. The grouping variable is `Origin`.

Index into the third cell of `stats`.

```
stats{3}
```

```
ans =
```

Group	Name	Estimate	Lower	Upper
Error	'Res Std'	3.0133	2.8034	3.239

The third cell of `stats` contains the estimate for residual standard deviation and the 95% confidence limits. The estimate for residual standard deviation is the square root of `mse`, `sqrt(9.0755) = 3.0126`.

Construct 99% confidence intervals for the covariance parameters.

```
[~,~,stats] = covarianceParameters(lme, 'Alpha', 0.01);  
stats{1}
```

```
ans =
```

```
COVARIANCE TYPE: FULLCHOLESKY
```

Group	Name1	Name2	Type
Model_Year	'Intercept'	'Intercept'	'std'

Model_Year	'Acceleration'	'Intercept'	'corr'
Model_Year	'Acceleration'	'Acceleration'	'std'

Estimate	Lower	Upper
2.8824	0.824	10.083
-0.87704	-0.99176	0.018241
0.33066	0.15873	0.68881

```
stats{2}
```

```
ans =
```

```
COVARIANCE TYPE: FULLCHOLESKY
```

Group	Name1	Name2	Type	Estimate
Origin	'Weight'	'Weight'	'std'	0.00025996

Lower	Upper
6.6115e-05	0.0010221

```
stats{3}
```

```
ans =
```

Group	Name	Estimate	Lower	Upper
Error	'Res Std'	3.0133	2.7405	3.3134

## See Also

[LinearMixedModel](#) | [compare](#) | [fixedEffects](#) | [randomEffects](#)

## CovarianceType property

**Class:** `gmdistribution`

Type of covariance matrices

### Description

'diagonal' if the covariance matrices are restricted to be diagonal; 'full' otherwise.

# coxphfit

Cox proportional hazards regression

## Syntax

```
b = coxphfit(X,T)
b = coxphfit(X,T,Name,Value)
[b,logl,H,stats] = coxphfit(____)
```

## Description

`b = coxphfit(X,T)` returns a  $p$ -by-1 vector, `b`, of coefficient estimates for a Cox proportional hazards regression on page 32-1187 of the observed responses `T` on the predictors `X`, where `T` is either an  $n$ -by-1 vector or an  $n$ -by-2 matrix, and `X` is an  $n$ -by- $p$  matrix.

The model does not include a constant term, and `X` cannot contain a column of 1s.

`b = coxphfit(X,T,Name,Value)` returns a vector of coefficient estimates, with additional options specified by one or more `Name,Value` pair arguments.

`[b,logl,H,stats] = coxphfit(____)` also returns the loglikelihood, `logl`, a structure, `stats`, that contains additional statistics, and a two-column matrix, `H`, that contains the `T` values in the first column and the estimated baseline cumulative hazard, in the second column. You can use any of the input arguments in the previous syntaxes.

## Examples

### Use Cox Proportional Hazards Regression to Model Lifetime of Light Bulbs

Load the sample data.

```
load(fullfile(matlabroot,'examples','stats','lightbulb.mat'));
```

The first column of the light bulb data has the lifetime (in hours) of two different types of bulbs. The second column has the binary variable indicating whether the bulb is fluorescent or incandescent. 0 indicates that the bulb is incandescent, and 1 indicates that it is fluorescent. The third column contains the censorship information, where 0 indicates the bulb was observed until failure, and 1 indicates the bulb was censored.

Fit a Cox proportional hazards model for the lifetime of the light bulbs, also accounting for censoring. The predictor variable is the type of bulb.

```
b = coxphfit(lightbulb(:,2),lightbulb(:,1), ...  
'Censoring',lightbulb(:,3))
```

```
b =
```

```
4.7262
```

The estimate of the hazard ratio is  $e^b = 112.8646$ . This means that the hazard for the incandescent bulbs is 112.86 times the hazard for the fluorescent bulbs.

### Change Algorithm Parameters for Cox Proportional Hazards Model

Load the sample data.

```
load(fullfile(matlabroot,'examples','stats','lightbulb.mat'));
```

The first column of the data has the lifetime (in hours) of two types of bulbs. The second column has the binary variable indicating whether the bulb is fluorescent or incandescent. 1 indicates that the bulb is fluorescent and 0 indicates that it is incandescent. The third column contains the censorship information, where 0 indicates the bulb is observed until failure, and 1 indicates the item (bulb) is censored.

Fit a Cox proportional hazards model, also accounting for censoring. The predictor variable is the type of bulb.

```
b = coxphfit(lightbulb(:,2),lightbulb(:,1), ...  
'Censoring',lightbulb(:,3))
```

```
b =
```



4.7262

Display the default control parameters for the algorithm `coxphfit` uses to estimate the coefficients.

```
statset('coxphfit')
```

```
ans =
```

```
struct with fields:

    Display: 'off'
    MaxFunEvals: 200
    MaxIter: 100
    TolBnd: 1.0000e-06
    TolFun: 1.0000e-08
    TolTypeFun: []
    TolX: 1.0000e-08
    TolTypeX: []
    GradObj: []
    Jacobian: []
    DerivStep: []
    FunValCheck: []
    Robust: []
    RobustWgtFun: []
    WgtFun: []
    Tune: []
    UseParallel: []
    UseSubstreams: []
    Streams: {}
    OutputFcn: []
```

Save the options under a different name and change how the results will be displayed and the maximum number of iterations, `Display` and `MaxIter`.

```
coxphopt = statset('coxphfit');
coxphopt.Display = 'final';
coxphopt.MaxIter = 50;
```

Run `coxphfit` with the new algorithm parameters.

```
b = coxphfit(lightbulb(:,2),lightbulb(:,1),...  
'Censoring',lightbulb(:,3),'Options',coxphopt)
```

```
Successful convergence: Norm of gradient less than OPTIONS.TolFun
```

```
b =
```

```
4.7262
```

coxphfit displays a report on the final iteration. Changing the maximum number of iterations did not affect the coefficient estimate.

### Fit and Compare Cox and Weibull Survivor Functions

Generate Weibull data depending on predictor X.

```
rng('default') % for reproducibility  
X = 4*rand(100,1);  
A = 50*exp(-0.5*X);  
B = 2;  
y = wblrnd(A,B);
```

The response values are generated from a Weibull distribution with a shape parameter depending on the predictor variable X and a scale parameter of 2.

Fit a Cox proportional hazards model.

```
[b,logL,H,stats] = coxphfit(X,y);  
[b logL]
```

```
ans =
```

```
0.9409 -331.1479
```

The coefficient estimate is 0.9409 and the log likelihood value is -331.1479.

Request the model statistics.

```
stats
```

```

stats =

  struct with fields:

    covb: 0.0158
    beta: 0.9409
    se: 0.1256
    z: 7.4889
    p: 6.9462e-14
    csres: [100x1 double]
    devres: [100x1 double]
    martres: [100x1 double]
    schres: [100x1 double]
    sschres: [100x1 double]
    scores: [100x1 double]
    sscores: [100x1 double]

```

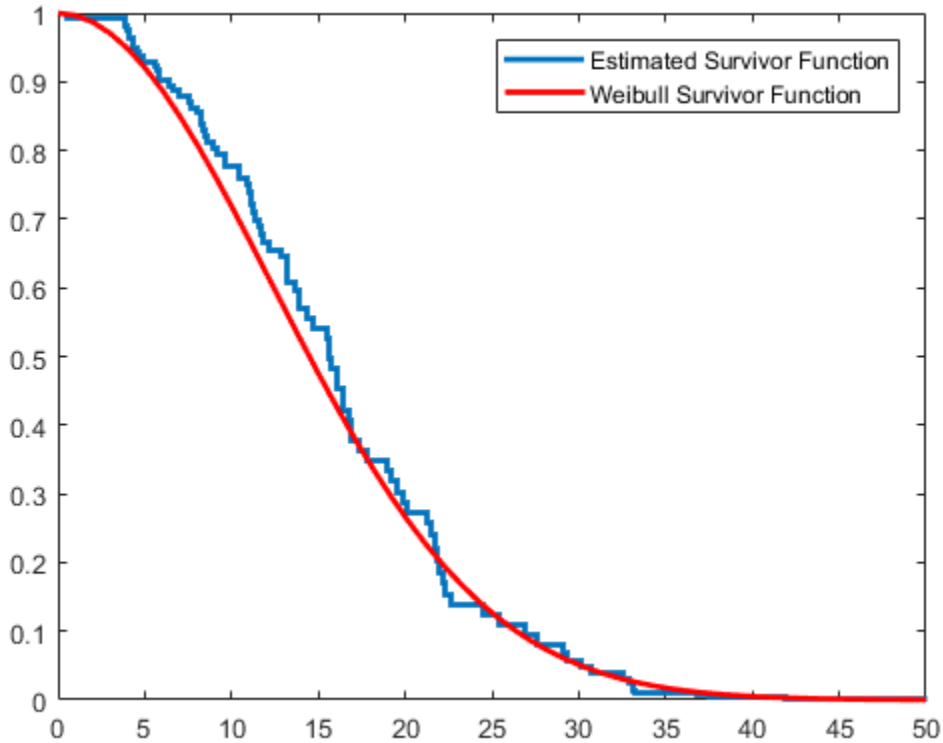
The covariance matrix of the coefficient estimates, `covb`, contains only one value, which is equal to the variance of the coefficient estimate in this example. The coefficient estimate, `beta`, is the same as `b` and is equal to 0.9409. The standard error of the coefficient estimate, `se`, is 0.1256, which is the square root of the variance 0.0158. The  $z$ -statistic, `z`, is  $\text{beta}/\text{se} = 0.9409/0.1256 = 7.4880$ . The  $p$ -value, `p`, indicates that the effect of `X` is significant.

Plot the Cox estimate of the baseline survivor function together with the known Weibull function.

```

stairs(H(:,1),exp(-H(:,2)),'LineWidth',2)
xx = linspace(0,100);
line(xx,1-wblcdf(xx,50*exp(-0.5*mean(X)),B),'color','r','LineWidth',2)
xlim([0,50])
legend('Estimated Survivor Function','Weibull Survivor Function')

```



The fitted model gives a close estimate to the survivor function of the actual distribution.

- “Hazard and Survivor Functions for Different Groups” on page 14-17
- “Survivor Functions for Two Groups” on page 14-24
- “Cox Proportional Hazards Model for Censored Data” on page 14-35
- “Cox Proportional Hazards Model with Time-Dependent Covariates” on page 14-41

## Input Arguments

**x** — Observations on predictor variables  
matrix

Observations on predictor variables, specified as an  $n$ -by- $p$  matrix of  $p$  predictors for each of  $n$  observations.

The model does not include a constant term, thus  $X$  cannot contain a column of 1s.

If  $X$ ,  $T$ , or the value of 'Frequency' or 'Strata' contain NaN values, then `coxphfit` removes rows with NaN values from all data when fitting a Cox model.

Data Types: `double`

### **T** — Time-to-event data

vector | two-column matrix

Time-to-event data, specified as an  $n$ -by-1 vector or a two-column matrix.

- When  $T$  is an  $n$ -by-1 vector, it represents the event time of right-censored time-to-event data.
- When  $T$  is an  $n$ -by-2 matrix, each row represents the risk interval (start,stop] in the counting process format for time-dependent covariates. The first column is the start time and the second column is the stop time. For an example, see “Cox Proportional Hazards Model with Time-Dependent Covariates” on page 14-41.

If  $X$ ,  $T$ , or the value of 'Frequency' or 'Strata' contain NaN values, then `coxphfit` removes rows with NaN values from all data when fitting a Cox model.

Data Types: `single` | `double`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

Example: `'Baseline', 0, 'Censoring', censoreddata, 'Frequency', freq` specifies that `coxphfit` calculates the baseline hazard rate relative to 0, considering the censoring information in the vector `censoreddata`, and the frequency of observations on  $T$  and  $X$  given in the vector `freq`.

### **B0** — Coefficient initial values

`0.01/std(X)` (default) | numeric vector

Coefficient initial values, specified as the comma-separated value consisting of 'B0' and a numeric vector.

Data Types: double

**Baseline — x values at which to compute the baseline hazard**

mean(X) (default) | scalar value

X values at which to compute the baseline hazard, specified as the comma-separated pair consisting of 'Baseline' and a scalar value.

The default is mean(X), so the hazard rate at X is  $h(t) * \exp((X - \text{mean}(X)) * b)$ . Enter 0 to compute the baseline relative to 0, so the hazard rate at X is  $h(t) * \exp(X * b)$ .

Changing the baseline does not affect the coefficient estimates, but the hazard ratio changes.

Example: 'Baseline', 0

Data Types: double

**Censoring — Indicator for censoring**

array of 0s (default) | array of 0s and 1s

Indicator for censoring, specified as the comma-separated pair consisting of 'Censoring' and a Boolean array of the same size as T. Use 1 for observations that are right censored and 0 for observations that are fully observed. The default is all observations are fully observed. For an example, see “Cox Proportional Hazards Model for Censored Data” on page 14-35.

Example: 'Censoring', cens

Data Types: logical

**Frequency — Frequency or weights of observations**

array of 1s (default) | vector of nonnegative scalar values

Frequency or weights of observations, specified as the comma-separated pair consisting of 'Frequency' and an array that is the same size as T containing nonnegative scalar values. The array can contain integer values corresponding to frequencies of observations or nonnegative values corresponding to observation weights.

If X, T, or the value of 'Frequency' or 'Strata' contain NaN values, then `coxphfit` removes rows with NaN values from all data when fitting a Cox model.

The default is 1 per row of `X` and `T`.

Example: `'Frequency',w`

Data Types: double

### **strata** — Stratification variables

`[]` (default) | matrix of real values

Stratification variables, specified as the comma-separated pair consisting of a matrix of real values. The matrix must have the same number of rows as `T`, with each row corresponding to an observation.

If `X`, `T`, or the value of `'Frequency'` or `'Strata'` contain NaN values, then `coxphfit` removes rows with NaN values from all data when fitting a Cox model.

The default, `[]`, is no stratification variable.

Example: `'Strata',Gender`

Data Types: single | double

### **ties** — Method to handle tied failure times

`'breslow'` (default) | `'efron'`

Method to handle tied failure times, specified as the comma-separated pair consisting of `'Ties'` and either `'breslow'` or `'efron'`.

Example: `'Ties','efron'`

Data Types: char

### **options** — Algorithm control parameters

structure

Algorithm control parameters for the iterative algorithm used to estimate `b`, specified as the comma-separated pair consisting of `'Options'` and a structure. A call to `statset` creates this argument. For parameter names and default values, type `statset('coxphfit')`. You can set the options under a new name and use that in the name-value pair argument.

Example: `'Options',statset('coxphfit')`

Data Types: char

## Output Arguments

### **b** — Coefficient estimates

vector

Coefficient estimates for a Cox proportional hazards regression on page 32-1187, returned as a  $p$ -by-1 vector.

### **logl** — Loglikelihood

scalar

Loglikelihood of the fitted model, returned as a scalar.

You can use log likelihood values to compare different models and assess the significance of effects of terms in the model.

### **H** — Estimated baseline cumulative hazard

two-column matrix |  $(2+k)$  column matrix

Estimated baseline cumulative hazard rate evaluated at  $T$  values, returned as one of the following.

- If the model is unstratified, then  $H$  is a two-column matrix. The first column of the matrix contains  $T$  values, and the second column contains cumulative hazard rate estimates.
- If the model is stratified, then  $H$  is a  $(2+k)$  column matrix, where the last  $k$  columns correspond to the stratification variables using the `Strata` name-value pair argument.

### **stats** — Coefficient statistics

structure

Coefficient statistics, returned as a structure that contains the following fields.

beta	Coefficient estimates (same as <code>b</code> )
se	Standard errors of coefficient estimates, <code>b</code>
z	$z$ -statistics for <code>b</code> (that is, <code>b</code> divided by standard error)
p	$p$ -values for <code>b</code>
covb	Estimated covariance matrix for <code>b</code>



csres	Cox-Snell residuals
devres	Deviance residuals
martres	Martingale residuals
schres	Schoenfeld residuals
sschres	Scaled Schoenfeld residuals
scores	Score residuals
sscores	Scaled score residuals

coxphfit returns the Cox-Snell, martingale, and deviance residuals as a column vector with one row per observation. It returns the Schoenfeld, scaled Schoenfeld, score, and scaled score residuals as matrices of the same size as X. Schoenfeld and scaled Schoenfeld residuals of censored data are NaNs.

## Definitions

### Cox Proportional Hazards Regression

Cox proportional hazards regression is a semiparametric method for adjusting survival rate estimates to remove the effect of confounding variables and to quantify the effect of predictor variables. The method represents the effects of explanatory and confounding variables as a multiplier of a common baseline hazard function,  $h_0(t)$ .

For a baseline relative to 0, this model corresponds to

$$h(X_i, t) = h_0(t) \exp \left[ \sum_{j=1}^p x_{ij} b_j \right],$$

where  $X_i = (x_{i1}, x_{i2}, \dots, x_{ip})$  is the predictor variable for the  $i$ th subject,  $h(X_i, t)$  is the hazard rate at time  $t$  for  $X_i$ , and  $h_0(t)$  is the baseline hazard rate function. The baseline hazard function is the nonparametric part of the Cox proportional hazards regression function, whereas the impact of the predictor variables is a loglinear regression. The assumption is that the baseline hazard function depends on time,  $t$ , but the predictor variables do not depend on time. See “Cox Proportional Hazards Model” on page 14-29 for details, including the extensions for stratification and time-dependent variables, tied events, and observation weights.

## References

- [1] Cox, D.R., and D. Oakes. *Analysis of Survival Data*. London: Chapman & Hall, 1984.
- [2] Lawless, J. F. *Statistical Models and Methods for Lifetime Data*. Hoboken, NJ: Wiley-Interscience, 2002.
- [3] Kleinbaum, D. G., and M. Klein. *Survival Analysis*. Statistics for Biology and Health. 2nd edition. Springer, 2005.

## See Also

`ecdf` | `statset` | `wblfit`

## Topics

- “Hazard and Survivor Functions for Different Groups” on page 14-17
- “Survivor Functions for Two Groups” on page 14-24
- “Cox Proportional Hazards Model for Censored Data” on page 14-35
- “Cox Proportional Hazards Model with Time-Dependent Covariates” on page 14-41
- “What Is Survival Analysis?” on page 14-2
- “Kaplan-Meier Method” on page 14-10
- “Cox Proportional Hazards Model” on page 14-29

**Introduced before R2006a**

## Prior property

**Class:** NaiveBayes

Class priors

### Description

The `Prior` property is a vector of length `NClasses` containing the class priors. The priors for empty classes are zero.

## createns

Create object to use in  $k$ -nearest neighbor search

### Syntax

```
NS = createns(X)
NS = createns(X, 'Name', Value)
```

### Description

`NS = createns(X)` uses the data observations in an  $m \times n$  matrix `X` to create an object `NS`. Rows of `X` correspond to observations and columns correspond to variables. `NS` is either an `ExhaustiveSearcher` or a `KDTreeSearcher` model object which you can use to find nearest neighbors in `X` for desired query points. If `NS` is an `ExhaustiveSearcher` model, `knnsearch` and `rangesearch` use the exhaustive search algorithm to find nearest neighbors. If `NS` is a `KDTreeSearcher` model, `createns` grows and saves a *Kd*-tree based on `X` in `NS`. `knnsearch` and `rangesearch` use the *Kd*-tree to find nearest neighbors. For information on these search methods, see “[k-Nearest Neighbor Search and Radius Search](#)” on page 18-18.

`NS = createns(X, 'Name', Value)` accepts one or more optional name/value pairs. Specify *Name* inside single quotes. Specify `NSMethod` to determine which type of object to create. The object's properties save the information when you specify other arguments. For more information on the objects' properties, see `ExhaustiveSearcher` or a `KDTreeSearcher`.

# Input Arguments

## Name-Value Pair Arguments

### **NSMethod**

Nearest neighbor search method, used to define the type of object created. Value is either:

- 'kdtree' — Create a `KDTreeSearcher` model. If you do not specify `NSMethod`, this is the default value when the number of columns of `X` is less than 10, `X` is not sparse, and the distance measure is one of the following measures:
  - 'euclidean' (default)
  - 'cityblock'
  - 'minkowski'
  - 'chebychev'
- 'exhaustive' — Create an `ExhaustiveSearcher` model. If you do not specify `NSMethod`, this is the default value when the default criteria for 'kdtree' do not apply.

### **Distance**

A character vector or a function handle specifying the default distance metric used when you call `knnsearch` or `rangesearch` to find nearest neighbors for future query points. If you specify a distance metric but not an `NSMethod`, this input determines the type of object `createns` creates, according to the default values described in `NSMethod`.

For both `KDTreeSearcher` and `ExhaustiveSearcher` models, the following options apply:

- 'euclidean' (default) — Euclidean distance.
- 'cityblock' — City block distance.
- 'chebychev' — Chebychev distance (maximum coordinate difference).
- 'minkowski' — Minkowski distance.

The following options apply to `ExhaustiveSearcher` models:

- 'seuclidean' — Standardized Euclidean distance. Each coordinate difference between rows in `X` and the query matrix is scaled by dividing by the corresponding element of the standard deviation computed from `X`, `S=nansd(X)`. To specify another value for `S`, use the `Scale` argument.
- 'mahalanobis' — Mahalanobis distance, which is computed using a positive definite covariance matrix `C`. The default value of `C` is the sample covariance matrix of `X`, as computed by `nancov(X)`. To change the value of `C`, use the `Cov` parameter.
- 'cosine' — One minus the cosine of the included angle between observations (treated as vectors).
- 'correlation' — One minus the sample linear correlation between observations (treated as sequences of values).
- 'spearman' — One minus the sample Spearman's rank correlation between observations (treated as sequences of values).
- 'hamming' — Hamming distance, which is percentage of coordinates that differ.
- 'jaccard' — One minus the Jaccard coefficient, which is the percentage of nonzero coordinates that differ.
- custom distance function — A distance function specified using `@` (for example, `@distfun`). A distance function must be of the form `function D2 = distfun(ZI, ZJ)`, taking as arguments a 1-by- $n$  vector `ZI` containing a single row from `X` or from the query points `Y`, and an  $m_2$ -by- $n$  matrix `ZJ` containing multiple rows of `X` or `Y`, and returning an  $m_2$ -by-1 vector of distances `D2`, whose  $j$ th element is the distance between the observations `ZI` and `ZJ(j, :)`.

**P**

A positive scalar,  $p$ , indicating the exponent of the Minkowski distance. This parameter is only valid when `Distance` is 'minkowski'. Default is 2.

**Cov**

A positive definite matrix indicating the covariance matrix when computing the Mahalanobis distance. This parameter is only valid when `Distance` is 'mahalanobis'. Default is `nancov(X)`.

**Scale**

A vector `S` with the length equal to the number of columns in `X`. Each coordinate of `X` and each query point is scaled by the corresponding element of `S` when computing the

standardized Euclidean distance. This parameter is only valid when `Distance` is `'seuclidean'`. Default is `nanstd(X)`.

### **BucketSize**

A positive integer, indicating the maximum number of data points in each leaf node of the *Kd*-tree. This argument is only meaningful when using the *Kd*-tree search method. Default is 50.

## Examples

### **Grow a *K*d-tree Using the Minkowski Distance Metric**

Grow a *K* d-tree that uses the Minkowski distance with an exponent of five.

Load Fisher's iris data set. Create a variable for the petal dimensions.

```
load fisheriris
x = meas(:,3:4);
```

Grow a *K* d-tree. Specify the Minkowski distance with an exponent of five.

```
Mdl = createns(x, 'Distance', 'minkowski', 'P', 5)
```

```
Mdl =
```

```
    KDTreeSearcher with properties:
```

```
        BucketSize: 50
           Distance: 'minkowski'
    DistParameter: 5
                X: [150x2 double]
```

Since `x` has two columns and the distance metric is Minkowski, `createns` creates a `KDTreeSearcher` model by default.

## See Also

`ExhaustiveSearcher` | `KDTreeSearcher` | `knnsearch` | `rangesearch`

## Topics

ExhaustiveSearcher

KDTreeSearcher

“k-Nearest Neighbor Search and Radius Search” on page 18-18

**Introduced in R2010a**



# crosstab

Cross-tabulation

## Syntax

```
tbl = crosstab(x1, x2)
tbl = crosstab(x1, ..., xn)
[tbl, chi2, p] = crosstab(____)
[tbl, chi2, p, labels] = crosstab(____)
```

## Description

`tbl = crosstab(x1, x2)` returns a cross-tabulation, `tbl`, of two vectors of the same length, `x1` and `x2`.

`tbl = crosstab(x1, ..., xn)` returns a multi-dimensional cross-tabulation, `tbl`, of data for multiple input vectors, `x1`, `x2`, ..., `xn`.

`[tbl, chi2, p] = crosstab(____)` also returns the chi-square statistic, `chi2`, and its *p*-value, `p`, for a test that `tbl` is independent in each dimension. You can use any of the previous syntaxes.

`[tbl, chi2, p, labels] = crosstab(____)` also returns a cell array, `labels`, which contains one column of labels for each input argument, `x1` ... `xn`.

## Examples

### Cross-Tabulate Two Data Vectors

Create two sample data vectors, containing three and four distinct values, respectively.

```
x = [1 1 2 3 1];
y = [1 2 5 3 1];
```

Cross-tabulate `x` and `y`.

```
table = crosstab(x,y)
```

```
table =  
  
     2     1     0     0  
     0     0     0     1  
     0     0     1     0
```

The rows in `table` correspond to the three distinct values in `x`, and the columns correspond to the four distinct values in `y`.

### Cross-Tabulate Independent Data Vectors

Generate two independent vectors, `x1` and `x2`, each containing 50 discrete uniform random numbers in the range 1:3.

```
rng default; % for reproducibility  
x1 = unidrnd(3,50,1);  
x2 = unidrnd(3,50,1);
```

Cross-tabulate `x1` and `x2`.

```
[table,chi2,p] = crosstab(x1,x2)
```

```
table =  
  
     1     6     7  
     5     5     2  
    11     7     6
```

```
chi2 =  
  
     7.5449
```

```
p =
```

```
0.1097
```

The returned p value of 0.1097 indicates that, at the 5% significance level, `crosstab` fails to reject the null hypothesis that `table` is independent in each dimension.

### Cross-Tabulate Grouped Data

Load the sample data, which contains measurements of large model cars during the years 1970-1982.

```
load carbig
```

Cross-tabulate the data of four-cylinder cars (`cyl4`) based on model year (`when`) and country of origin (`org`).

```
[table,chi2,p,labels] = crosstab(cyl4,when,org);
```

Use `labels` to determine the index location in `table` for the number of four-cylinder cars made in the USA during the late period of the data.

```
labels
```

```
labels =
```

```
3x3 cell array
```

```
{'Other'   }    {'Early'}    {'USA'    }
{'Four'    }    {'Mid'   }    {'Europe' }
{0x0 double}    {'Late'  }    {'Japan'  }
```

The first column of `labels` corresponds to the data in `cyl4`, and indicates that row 2 of `table` contains data on cars with four cylinders. The second column of `labels` corresponds to the data in `when`, and indicates that column 3 of `table` contains data on cars made during the late period. The third column of `labels` corresponds to the data in `org`, and indicates that location 1 of the third dimension of `table` contains data on cars made in the USA.

Therefore, `table(2,3,1)` contains the number of four-cylinder cars made in the USA during the late period.

```
table(2,3,1)
```

```
ans =
```

```
38
```

The data contains 38 four-cylinder cars made in the USA during the late period.

### Generate Contingency Table Using `crosstab`

Load the hospital data.

```
load hospital
```

The `hospital` dataset array contains data on 100 hospital patients, including last name, gender, age, weight, smoking status, and systolic and diastolic blood pressure measurements.

To determine whether smoking status is independent of gender, use `crosstab` to create a 2-by-2 contingency table of smokers and nonsmokers, grouped by gender.

```
[tbl,chi2,p,labels] = crosstab(hospital.Sex, hospital.Smoker)
```

```
tbl =
```

```
40    13  
26    21
```

```
chi2 =
```

```
4.5083
```

```
p =
```

```
0.0337
```

```
labels =
```

```
2x2 cell array

    {'Female'}    {'0'}
    {'Male'  }    {'1'}
```

The rows of the resulting contingency table `tbl` correspond to the patient's gender, with row 1 containing data for females and row 2 containing data for males. The columns correspond to the patient's smoking status, with column 1 containing data for nonsmokers and column 2 containing data for smokers. The returned result `chi2 = 4.5083` is the value of the chi-squared test statistic for a Pearson's chi-squared test of independence. The returned value `p = 0.0337` is an approximate *P*-value based on the chi-squared distribution.

## Input Arguments

### **x1** — Input vector

vector of grouping variables

Input vector, specified as a vector of grouping variables. All input vectors, including `x1`, `x2`, ..., `xn`, must be the same length.

Data Types: `single` | `double` | `char` | `logical`

### **x2** — Input vector

vector of grouping variables

Input vector, specified as a vector of grouping variables. All input vectors, including `x1`, `x2`, ..., `xn`, must be the same length.

Data Types: `single` | `double` | `char` | `logical`

### **x1, ..., xn** — Input vectors

vectors of grouping variables

Input vectors, specified as vectors of grouping variables. If you use this syntax to specify more than two input vectors, then `crosstab` generates a multi-dimensional cross-tabulation table. All input vectors, including `x1`, `x2`, ..., `xn`, must be the same length.

Data Types: `single` | `double` | `char` | `logical`

## Output Arguments

### **tbl** — Cross-tabulation table

matrix of integer values

Cross-tabulation table, returned as a matrix of integer values.

If you specify two input vectors, `x1` and `x2`, then `tbl` is an  $m$ -by- $n$  matrix, where  $m$  is the number of distinct values in `x1` and  $n$  is the number of distinct values in `x2`.

If you specify three or more input vectors, then `tbl(i, j, ..., n)` is a count of indices where `grp2idx(x1)` is  $i$ , `grp2idx(x2)` is  $j$ , `grp2idx(x3)` is  $k$ , and so on.

### **chi2** — Chi-square statistic

positive scalar value

Chi-square statistic, returned as a positive scalar value. The null hypothesis is that the proportion in any entry of `tbl` is the product of the proportions in each dimension.

### **p** — $p$ -Value

scalar value in the range  $[0, 1]$

$p$ -value for the chi-square test statistic, returned as a scalar value in the range  $[0, 1]$ . `crosstab` tests that `tbl` is independent in each dimension.

### **labels** — Data labels

cell array

Data labels, returned as a cell array. The entries in the first column are labels for the rows of `tbl`, the entries in the second column are labels for the columns, and so on, for a multi-dimensional `tbl`.

## Algorithms

`crosstab` uses `grp2idx` to assign a positive integer to each distinct value. `tbl(i, j)` is a count of indices where `grp2idx(x1)` is  $i$  and `grp2idx(x2)` is  $j$ . The numerical order of `grp2idx(x1)` and `grp2idx(x2)` order rows and columns of `tbl`, respectively.

In this case, the returned value of `tbl(i, j, ..., n)` is a count of indices where `grp2idx(x1)` is  $i$ , `grp2idx(x2)` is  $j$ , `grp2idx(x3)` is  $k$ , and so on.

## Extended Capabilities

### Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function supports tall arrays for out-of-memory data with the limitation:

The fourth output, `labels`, is returned as a cell array containing `M` unevaluated tall cell arrays, where `M` is the number of input grouping variables. Each unevaluated tall cell array, `labels{j}`, contains the labels for one grouping variable.

For more information, see “Tall Arrays” (MATLAB).

### See Also

`grp2idx` | `tabulate`

### Topics

“Grouping Variables” on page 2-59

Introduced before R2006a

## crossval

Loss estimate using cross validation

### Syntax

```
vals = crossval(fun,X)
vals = crossval(fun,X,Y,...)
mse = crossval('mse',X,y,'Predfun',predfun)
mcr = crossval('mcr',X,y,'Predfun',predfun)
val = crossval(criterion,X1,X2,...,y,'Predfun',predfun)
vals = crossval(...,'name',value)
```

### Description

`vals = crossval(fun,X)` performs 10-fold cross validation for the function `fun`, applied to the data in `X`.

`fun` is a function handle to a function with two inputs, the training subset of `X`, `XTRAIN`, and the test subset of `X`, `XTEST`, as follows:

```
testval = fun(XTRAIN,XTEST)
```

Each time it is called, `fun` should use `XTRAIN` to fit a model, then return some criterion `testval` computed on `XTEST` using that fitted model.

`X` can be a column vector or a matrix. Rows of `X` correspond to observations; columns correspond to variables or features. Each row of `vals` contains the result of applying `fun` to one test set. If `testval` is a non-scalar value, `crossval` converts it to a row vector using linear indexing and stored in one row of `vals`.

`vals = crossval(fun,X,Y,...)` is used when data are stored in separate variables `X`, `Y`, ... . All variables (column vectors, matrices, or arrays) must have the same number of rows. `fun` is called with the training subsets of `X`, `Y`, ... , followed by the test subsets of `X`, `Y`, ... , as follows:

```
testvals = fun(XTRAIN,YTRAIN,...,XTEST,YTEST,...)
```



`mse = crossval('mse', X, y, 'Predfun', predfun)` returns `mse`, a scalar containing a 10-fold cross validation estimate of mean-squared error for the function `predfun`. `X` can be a column vector, matrix, or array of predictors. `y` is a column vector of response values. `X` and `y` must have the same number of rows.

`predfun` is a function handle called with the training subset of `X`, the training subset of `y`, and the test subset of `X` as follows:

```
yfit = predfun(XTRAIN, ytrain, XTEST)
```

Each time it is called, `predfun` should use `XTRAIN` and `ytrain` to fit a regression model and then return fitted values in a column vector `yfit`. Each row of `yfit` contains the predicted values for the corresponding row of `XTEST`. `crossval` computes the squared errors between `yfit` and the corresponding response test set, and returns the overall mean across all test sets.

`mcr = crossval('mcr', X, y, 'Predfun', predfun)` returns `mcr`, a scalar containing a 10-fold cross validation estimate of misclassification rate (the proportion of misclassified samples) for the function `predfun`. The matrix `X` contains predictor values and the vector `y` contains class labels. `predfun` should use `XTRAIN` and `YTRAIN` to fit a classification model and return `yfit` as the predicted class labels for `XTEST`. `crossval` computes the number of misclassifications between `yfit` and the corresponding response test set, and returns the overall misclassification rate across all test sets.

`val = crossval(criterion, X1, X2, ..., y, 'Predfun', predfun)`, where `criterion` is 'mse' or 'mcr', returns a cross validation estimate of mean-squared error (for a regression model) or misclassification rate (for a classification model) with predictor values in `X1`, `X2`, ... and, respectively, response values or class labels in `y`. `X1`, `X2`, ... and `y` must have the same number of rows. `predfun` is a function handle called with the training subsets of `X1`, `X2`, ..., the training subset of `y`, and the test subsets of `X1`, `X2`, ..., as follows:

```
yfit=predfun(X1TRAIN, X2TRAIN, ..., ytrain, X1TEST, X2TEST, ...)
```

`yfit` should be a column vector containing the fitted values.

`vals = crossval(..., 'name', value)` specifies one or more optional parameter name/value pairs from the following table. Specify `name` inside single quotes.

Name	Value
holdout	A scalar specifying the ratio or the number of observations $p$ for holdout cross validation. When $0 < p < 1$ , approximately $p \cdot n$ observations for the test set are randomly selected. When $p$ is an integer, $p$ observations for the test set are randomly selected.
kfold	A positive integer that is greater than 1 specifying the number of folds $k$ for $k$ -fold cross validation.
leaveout	Specifies leave-one-out cross validation. The value must be 1.
mcreps	A positive integer specifying the number of Monte-Carlo repetitions for validation. If the first input of <code>crossval</code> is 'mse' or 'mcr', <code>crossval</code> returns the mean of mean-squared error or misclassification rate across all of the Monte-Carlo repetitions. Otherwise, <code>crossval</code> concatenates the values <code>vals</code> from all of the Monte-Carlo repetitions along the first dimension.
partition	An object <code>c</code> of the <code>cvpartition</code> on page 32-1292 class, specifying the cross validation type and partition.
stratify	A column vector <code>group</code> specifying groups for stratification. Both training and test sets have roughly the same class proportions as in <code>group</code> . NaNs or empty character vectors in <code>group</code> are treated as missing values, and the corresponding rows of the data are ignored.

Name	Value
options	<p>A structure that specifies whether to run in parallel, and specifies the random stream or streams. Create the options structure with <code>statset</code>. Option fields:</p> <ul style="list-style-type: none"> <li>• <code>UseParallel</code> — Set to <code>true</code> to compute in parallel. Default is <code>false</code>.</li> <li>• <code>UseSubstreams</code> — Set to <code>true</code> to compute in parallel in a reproducible fashion. Default is <code>false</code>. To compute reproducibly, set <code>Streams</code> to a type allowing substreams: <code>'mlfg6331_64'</code> or <code>'mrg32k3a'</code>.</li> <li>• <code>Streams</code> — A <code>RandStream</code> object or cell array consisting of one such object. If you do not specify <code>Streams</code>, <code>crossval</code> uses the default stream.</li> </ul>

Only one of `kfold`, `holdout`, `leaveout`, or `partition` can be specified, and `partition` cannot be specified with `stratify`. If both `partition` and `mcreps` are specified, the first Monte-Carlo repetition uses the partition information in the `cvpartition` object, and the `repartition` method is called to generate new partitions for each of the remaining repetitions. If no cross validation type is specified, the default is 10-fold cross validation.

---

**Note** When using cross validation with classification algorithms, stratification is preferred. Otherwise, some test sets may not include observations from all classes.

---

## Examples

### Example 1

Compute mean-squared error for regression using 10-fold cross validation:

```
load('fisheriris');
y = meas(:,1);
X = [ones(size(y,1),1),meas(:,2:4)];

regf=@(XTRAIN,ytrain,XTEST)(XTEST*regress(ytrain,XTRAIN));
```

```
cvMse = crossval('mse',X,y,'predfun',regf)
cvMse =
    0.1015
```

## Example 2

Compute misclassification rate using stratified 10-fold cross validation:

```
load('fisheriris');
y = species;
X = meas;
cp = cvpartition(y,'k',10); % Stratified cross-validation

classf = @(XTRAIN, ytrain,XTEST)(classify(XTEST,XTRAIN,...
ytrain));

cvMCR = crossval('mcr',X,y,'predfun',classf,'partition',cp)
cvMCR =
    0.0200
```

## Example 3

Compute the confusion matrix using stratified 10-fold cross validation:

```
load('fisheriris');
y = species;
X = meas;
order = unique(y); % Order of the group labels
cp = cvpartition(y,'k',10); % Stratified cross-validation

f = @(xtr,ytr,xte,yte) confusionmat(yte,...
classify(xte,xtr,ytr),'order',order);

cfMat = crossval(f,X,y,'partition',cp);
cfMat = reshape(sum(cfMat),3,3)
cfMat =
    50     0     0
     0    48     2
     0     1    49
```

cfMat is the summation of 10 confusion matrices from 10 test sets.

## References

- [1] Hastie, T., R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*. New York: Springer, 2001.

## See Also

`cvpartition`

## Topics

“Grouping Variables” on page 2-59

Introduced in R2008a

## crossval

**Class:** ClassificationDiscriminant

Cross-validated discriminant analysis classifier

## Syntax

```
cvmodel = crossval(obj)
cvmodel = crossval(obj,Name,Value)
```

## Description

`cvmodel = crossval(obj)` creates a partitioned model from `obj`, a fitted discriminant analysis classifier. By default, `crossval` uses 10-fold cross validation on the training data to create `cvmodel`.

`cvmodel = crossval(obj,Name,Value)` creates a partitioned model with additional options specified by one or more `Name,Value` pair arguments.

## Input Arguments

**obj**

Discriminant analysis classifier, produced using `fitcdiscr`.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1,Value1, ...,NameN,ValueN`.

**CVPartition**

Object of class `cvpartition`, created by the `cvpartition` function. `crossval` splits the data into subsets with `cvpartition`.

Use only one of these options at a time: 'CVPartition', 'Holdout', 'Kfold', or 'Leaveout'.

**Default:** []

**Holdout**

Holdout validation tests the specified fraction of the data, and uses the rest of the data for training. Specify a numeric scalar from 0 to 1. Use only one of these options at a time: 'CVPartition', 'Holdout', 'Kfold', or 'Leaveout'.

**KFold**

Number of folds to use in a cross-validated classifier, a positive integer value greater than 1.

Use only one of these options at a time: 'CVPartition', 'Holdout', 'Kfold', or 'Leaveout'.

**Default:** 10

**Leaveout**

Set to 'on' for leave-one-out cross validation.

Use only one of these options at a time: 'CVPartition', 'Holdout', 'Kfold', or 'Leaveout'.

## Examples

Create a classification model for the Fisher iris data, and then create a cross-validation model. Evaluate the quality the model using `kfoldLoss`.

```
load fisheriris
obj = fitcdiscr(meas,species);
cvmodel = crossval(obj);
```

```
L = kfoldLoss(cvmodel)
```

```
L =  
    0.0200
```

## Tips

- Assess the predictive performance of `obj` on cross-validated data using the “kfold” methods and properties of `cvmodel`, such as `kfoldLoss`.

## Alternatives

You can create a cross-validation classifier directly from the data, instead of creating a discriminant analysis classifier followed by a cross-validation classifier. To do so, include one of these options in `fitcdiscr`: `'CrossVal'`, `'CVPartition'`, `'Holdout'`, `'KFold'`, or `'Leaveout'`.

## See Also

```
crossval | fitcdiscr | kfoldEdge | kfoldLoss | kfoldMargin | kfoldPredict  
| kfoldfun
```

## Topics

“Discriminant Analysis Classification” on page 20-2



# crossval

**Class:** ClassificationECOC

Cross-validate multiclass, error-correcting output codes model

## Syntax

```
CVMdl = crossval(Mdl)
```

```
CVMdl = crossval(Mdl,Name,Value)
```

## Description

`CVMdl = crossval(Mdl)` returns a cross-validated (partitioned), multiclass, error-correcting output codes (ECOC) model (`CVMdl`) from a trained ECOC model (`Mdl`).

By default, `crossval` uses 10-fold cross validation on the training data to create `CVMdl`.

`CVMdl = crossval(Mdl,Name,Value)` returns a partitioned ECOC model with additional options specified by one or more `Name,Value` pair arguments.

For example, you can specify the number of folds or a holdout sample proportion.

## Input Arguments

**mdl** — Multiclass ECOC model

ClassificationECOC model

Multiclass ECOC model, specified as a `ClassificationECOC` model returned by `fitcecoc`.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single

quotes ( ' ' ). You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

### **CVPartition — Cross-validation partition**

`[]` (default) | `cvpartition` partition object

Cross-validation partition, specified as the comma-separated pair consisting of 'CVPartition' and a `cvpartition` partition object as created by `cvpartition`. The partition object specifies the type of cross-validation, and also the indexing for training and validation sets.

To create a cross-validated model, you can use one of these four name-value pair arguments only: `CVPartition`, `Holdout`, `KFold`, or `Leaveout`.

Example: Suppose you create a random partition for 5-fold cross-validation on 500 observations by using `cvp = cvpartition(500, 'KFold', 5)`. Then, you can specify the cross-validation partition for `fitcecoc` as 'CVPartition', `cvp`.

### **Holdout — Fraction of data for holdout validation**

scalar value in the range (0,1)

Fraction of data used for holdout validation, specified as the comma-separated pair consisting of 'Holdout' and a scalar value in the range (0,1). If you specify 'Holdout',  $p$ , then the software:

- 1 Randomly reserves  $p*100\%$  of the data as validation data, and trains the model using the rest of the data
- 2 Stores the compact, trained model in the `Trained` property of the cross-validated model.

To create a cross-validated model, you can use one of these four name-value pair arguments only: `CVPartition`, `Holdout`, `KFold`, or `Leaveout`.

Example: 'Holdout', 0.1

Data Types: `double` | `single`

### **KFold — Number of folds**

10 (default) | positive integer value greater than 1

Number of folds to use in a cross-validated model, specified as the comma-separated pair consisting of 'KFold' and a positive integer value greater than 1. If you specify, e.g., 'KFold',  $k$ , then the software:

- 1 Randomly partitions the data into  $k$  sets
- 2 For each set, reserves the set as validation data, and trains the model using the other  $k - 1$  sets
- 3 Stores the  $k$  compact, trained models in the cells of a  $k$ -by-1 cell vector in the Trained property of the cross-validated model.

To create a cross-validated model, you can use one of these four name-value pair arguments only: CVPartition, Holdout, KFold, or Leaveout.

Example: 'KFold', 5

Data Types: single | double

#### **Leaveout — Leave-one-out cross-validation flag**

'off' (default) | 'on'

Leave-one-out cross-validation flag, specified as the comma-separated pair consisting of 'Leaveout' and 'on' or 'off'. If you specify 'Leaveout', 'on', then, for each of the  $n$  observations, where  $n$  is `size(Mdl.X, 1)`, the software:

- 1 Reserves the observation as validation data, and trains the model using the other  $n - 1$  observations
- 2 Stores the  $n$  compact, trained models in the cells of an  $n$ -by-1 cell vector in the Trained property of the cross-validated model.

To create a cross-validated model, you can use one of these four name-value pair arguments only: CVPartition, Holdout, KFold, or Leaveout.

Example: 'Leaveout', 'on'

Data Types: char

#### **Options — Estimation options**

[] (default) | structure array returned by `statset`

Estimation options, specified as the comma-separated pair consisting of 'Options' and a structure array returned by `statset`.

To invoke parallel computing:

- You need a Parallel Computing Toolbox license.
- Specify 'Options', `statset('UseParallel',1)`.

## Output Arguments

### `cvmdl` — Cross-validated ECOC model

`ClassificationPartitionedECOC` model

Cross-validated ECOC model, returned as a `ClassificationPartitionedECOC` model.

## Examples

### Cross-Validate ECOC Classifier

Load Fisher's iris data set.

```
load fisheriris
X = meas;
Y = species;
rng(1); % For reproducibility
```

Create an SVM template. It is good practice to standardize the predictors.

```
t = templateSVM('Standardize',1)
```

```
t =
```

Fit template for classification SVM.

```
Alpha: [0x1 double]
BoxConstraint: []
CacheSize: []
CachingMethod: ''
ClipAlphas: []
DeltaGradientTolerance: []
Epsilon: []
```

```

        GapTolerance: []
        KKTolerance: []
        IterationLimit: []
        KernelFunction: ''
            KernelScale: []
            KernelOffset: []
        KernelPolynomialOrder: []
            NumPrint: []
            Nu: []
        OutlierFraction: []
        RemoveDuplicates: []
        ShrinkagePeriod: []
            Solver: ''
        StandardizeData: 1
        SaveSupportVectors: []
        VerbosityLevel: []
        Version: 2
        Method: 'SVM'
        Type: 'classification'

```

`t` is an SVM template. All of its properties are empty, except for `StandardizeData`, `Method`, and `Type`. When the software trains the ECOC classifier, it sets the applicable properties to their default values.

Train the ECOC classifier. It is good practice to specify the class order.

```

Mdl = fitcecoc(X,Y,'Learners',t,...
    'ClassNames',{'setosa','versicolor','virginica'});

```

`Mdl` is a `ClassificationECOC` classifier. You can access its properties using dot notation.

Cross-validate `Mdl` using 10-fold cross-validation.

```

CVMdl = crossval(Mdl);

```

`CVMdl` is a `ClassificationPartitionedECOC` cross-validated ECOC classifier.

Estimate the generalization error.

```

oosLoss = kfoldLoss(CVMdl)

```

```

oosLoss =

```

```
0.0400
```

The out-of-sample classification error is 4%, which indicates that the ECOC classifier generalizes fairly well.

### Cross Validate an ECOC Classifier Using Parallel Computing

Consider the `arrhythmia` data set. There are 16 classes in the study, 13 of which are represented in the data. The first class indicates that the subject did not have arrhythmia, and the last class indicates that the subject's arrhythmia state was not recorded. Suppose that the other classes are ordinal levels indicating the severity of arrhythmia.

Train an ECOC classifier with a custom coding design specified by the description of the classes.

Load the `arrhythmia` data set.

```
load arrhythmia
Y = categorical(Y);
K = unique(Y); % Number of distinct classes
```

Construct a coding matrix that describes the nature of the classes.

```
OrdMat = designecoc(11, 'ordinal');
nOM = size(OrdMat);
class1VSOrd = [1; -ones(11,1); 0];
class1VSClass16 = [1; zeros(11,1); -1];
OrdVSClass16 = [0; ones(11,1); -1];
Coding = [class1VSOrd class1VSClass16 OrdVSClass16, ...
          [zeros(1,nOM(2)); OrdMat; zeros(1,nOM(2))]];

```

Train an ECOC classifier using the custom coding design (`Coding`) and parallel computing. Specify to use an ensemble of 50 classification trees boosted using `GentleBoost`.

```
t = templateEnsemble('GentleBoost', 50, 'Tree');
options = statset('UseParallel', 1);
Mdl = fitcecoc(X, Y, 'Coding', Coding, 'Learners', t, 'Options', options);
```

Mdl is a ClassificationECOC model. You can access its properties using dot notation.

Cross validate Mdl using 8-fold cross validation and parallel computing.

```
rng(1); % For reproducibility
CVMdl = crossval(Mdl, 'Options', options, 'Kfold', 8);
```

Warning: One or more folds do not contain points from all the groups.

Since some of the classes have low relative frequency, some of the folds do not train using observations from those classes. CVMdl is a ClassificationPartitionedECOC cross-validated ECOC model.

Estimate the generalization error using parallel computing.

```
oosLoss = kfoldLoss(CVMdl, 'Options', options)
```

```
oosLoss =
```

```
0.3208
```

The out-of-sample classification error is 32%, which indicates that this model does not generalize well. To improve the model, try training using a different boosting method, such as RobustBoost, or a different algorithm altogether, such as SVM.

- “Quick Start Parallel Computing for Statistics and Machine Learning Toolbox” on page 30-2

## Tips

Assess the predictive performance of Mdl on cross-validated data using the kfold functions and properties of CVMdl, such as kfoldLoss.

## Alternatives

Instead of training an ECOC model and then cross validating it, you can create a cross-validated ECOC model directly using fitcecoc and by specifying any of these name-value pair arguments: CrossVal, CVPartition, Holdout, Leaveout, or KFold.

## See Also

ClassificationECOC | ClassificationPartitionedECOC |  
CompactClassificationECOC | cvpartition | fitcecoc | statset

## Topics

“Quick Start Parallel Computing for Statistics and Machine Learning Toolbox” on page 30-2

“Reproducibility in Parallel Statistical Computations” on page 30-14

“Concepts of Parallel Computing in Statistics and Machine Learning Toolbox” on page 30-8



# crossval

**Class:** ClassificationEnsemble

Cross validate ensemble

## Syntax

```
cvens = crossval(ens)
cvens = crossval(ens,Name,Value)
```

## Description

`cvens = crossval(ens)` creates a cross-validated ensemble from `ens`, a classification ensemble. Default is 10-fold cross validation.

`cvens = crossval(ens,Name,Value)` creates a cross-validated ensemble with additional options specified by one or more `Name, Value` pair arguments. You can specify several name-value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

## Input Arguments

**ens**

A classification ensemble created with `fitcensemble`.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

**cvpartition**

A partition of class `cvpartition`. Sets the partition for cross validation.

Use no more than one of the name-value pairs `cvpartition`, `holdout`, `kfold`, or `leaveout`.

**holdout**

Holdout validation tests the specified fraction of the data, and uses the rest of the data for training. Specify a numeric scalar from 0 to 1. You can only use one of these four options at a time for creating a cross-validated tree: `'kfold'`, `'holdout'`, `'leaveout'`, or `'cvpartition'`.

**kfold**

Number of folds for cross validation, a numeric positive scalar greater than 1.

Use no more than one of the name-value pairs `'kfold'`, `'holdout'`, `'leaveout'`, or `'cvpartition'`.

**leaveout**

If `'on'`, use leave-one-out cross validation.

Use no more than one of the name-value pairs `'kfold'`, `'holdout'`, `'leaveout'`, or `'cvpartition'`.

**nprint**

Printout frequency, a positive integer scalar. Use this parameter to observe the training of cross-validation folds.

**Default:** `'off'`, meaning no printout

## Output Arguments

**cvens**

A cross-validated classification ensemble of class `ClassificationPartitionedEnsemble`.

## Examples

### Cross-Validate Classification Ensemble

Create a cross-validated classification model for the Fisher iris data, and assess its quality using the `kfoldLoss` method.

Load the Fisher iris data set.

```
load fisheriris
```

Train an ensemble of 100 boosted classification trees using `AdaBoostM2`.

```
t = templateTree('MaxNumSplits',1); % Weak learner template tree object
ens = fitcensemble(meas,species,'Method','AdaBoostM2','Learners',t);
```

Create a cross-validated ensemble from `ens` and find the classification error averaged over all folds.

```
rng(10,'twister') % For reproducibility
cvens = crossval(ens);
L = kfoldLoss(cvens)
```

```
L = 0.0467
```

## Alternatives

You can create a cross-validation ensemble directly from the data, instead of creating an ensemble followed by a cross-validation ensemble. To do so, include one of these five options in `fitcensemble`: `'crossval'`, `'kfold'`, `'holdout'`, `'leaveout'`, or `'cvpartition'`.

## See Also

`ClassificationPartitionedEnsemble` | `cvpartition`

## crossval

**Class:** ClassificationKNN

Cross-validated  $k$ -nearest neighbor classifier

## Syntax

```
cvmodel = crossval mdl
cvmodel = crossval mdl, Name, Value
```

## Description

`cvmodel = crossval(mdl)` creates a partitioned model from `mdl`, a fitted KNN classification model. By default, `crossval` uses 10-fold cross validation on the training data to create `cvmodel`.

`cvmodel = crossval(mdl, Name, Value)` creates a partitioned model with additional options specified by one or more `Name, Value` pair arguments.

## Input Arguments

**mdl** — Classifier model

classifier model object

$k$ -nearest neighbor classifier model, returned as a classifier model object.

Note that using the 'CrossVal', 'Kfold', 'Holdout', 'Leaveout', or 'CVPartition' options results in a model of class `ClassificationPartitionedModel`. You cannot use a partitioned tree for prediction, so this kind of tree does not have a `predict` method.

Otherwise, `mdl` is of class `ClassificationKNN`, and you can use the `predict` method to make predictions.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### **CVPartition**

Object of class `cvpartition`, created by the `cvpartition` function. `crossval` splits the data into subsets with `cvpartition`.

Use only one of these four options at a time: 'KFold', 'Holdout', 'Leaveout', or 'CVPartition'.

### **Holdout**

Holdout validation tests the specified fraction of the data, and uses the remaining data for training. Specify a numeric scalar from 0 to 1. Use only one of these four options at a time: 'KFold', 'Holdout', 'Leaveout', or 'CVPartition'.

### **KFold**

Number of folds to use in a cross-validated tree, a positive integer greater than 1.

Use only one of these four options at a time: 'KFold', 'Holdout', 'Leaveout', or 'CVPartition'.

### **Default: 10**

### **Leaveout**

Set to 'on' for leave-one-out cross validation.

Use only one of these four options at a time: 'KFold', 'Holdout', 'Leaveout', or 'CVPartition'.

## Output Arguments

### `cvmodel`

Partitioned model of class `ClassificationPartitionedModel`.

## Examples

### Cross-Validated K-Nearest Neighbor Model

Construct a cross-validated  $k$ -nearest neighbor model, and assess classification performance using the model.

Load the data.

```
load fisheriris
X = meas;
Y = species;
```

Construct a classifier for nearest neighbors.

```
mdl = fitcknn(X,Y);
```

Construct a cross-validated classifier.

```
cvmdl = crossval(mdl)
```

```
cvmdl =
```

```
classreg.learning.partition.ClassificationPartitionedModel
  CrossValidatedModel: 'KNN'
  PredictorNames: {'x1' 'x2' 'x3' 'x4'}
  ResponseName: 'Y'
  NumObservations: 150
  KFold: 10
  Partition: [1x1 cvpartition]
  ClassNames: {'setosa' 'versicolor' 'virginica'}
  ScoreTransform: 'none'
```

Find the cross-validated loss of the classifier.

```
cvmdlloss = kfoldLoss(cvmdl)
```

```
cvmdlloss =  
    0.0467
```

The cross-validated loss is less than 5%. You can expect `mdl` to have a similar error rate.

- “Examine Quality of KNN Classifier” on page 18-37
- “Modify KNN Classifier” on page 18-39

## Tips

- Assess the predictive performance of `mdl` on cross-validated data using the “kfold” methods and properties of `cvmodel`, such as `kfoldLoss`.

## Alternatives

You can create a cross-validated model directly from the data, instead of creating a model followed by a cross-validated model. To do so, include one of these options in `fitcknn`: `'CrossVal'`, `'KFold'`, `'Holdout'`, `'Leaveout'`, or `'CVPartition'`.

## See Also

`ClassificationKNN` | `ClassificationPartitionedModel` | `crossval` | `fitcknn` | `kfoldEdge` | `kfoldLoss` | `kfoldMargin` | `kfoldPredict` | `kfoldfun`

## Topics

- “Examine Quality of KNN Classifier” on page 18-37
- “Modify KNN Classifier” on page 18-39
- “Classification Using Nearest Neighbors” on page 18-16

## crossval

**Class:** ClassificationNaiveBayes

Cross-validated naive Bayes classifier

## Syntax

```
CVMD1 = crossval(Mdl)
```

```
CVMD1 = crossval(Mdl, Name, Value)
```

## Description

`CVMD1 = crossval(Mdl)` returns a partitioned naive Bayes classifier (`CVSMD1`) from a trained naive Bayes classifier (`Mdl`).

By default, `crossval` uses 10-fold cross validation on the training data to create `CVMD1`.

`CVMD1 = crossval(Mdl, Name, Value)` returns a partitioned naive Bayes classifier with additional options specified by one or more `Name, Value` pair arguments.

For example, you can specify a holdout sample proportion.

## Input Arguments

**mdl** — Fully trained naive Bayes classifier

ClassificationNaiveBayes model

A fully trained naive Bayes classifier, specified as a `ClassificationNaiveBayes` model trained by `fitcnb`.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single



quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### **CVPartition — Cross-validation partition**

`[]` (default) | `cvpartition` partition object

Cross-validation partition, specified as the comma-separated pair consisting of 'CVPartition' and a `cvpartition` partition object as created by `cvpartition`. The partition object specifies the type of cross-validation, and also the indexing for training and validation sets.

To create a cross-validated model, you can use one of these four name-value pair arguments only: `CVPartition`, `Holdout`, `KFold`, or `Leaveout`.

Example: Suppose you create a random partition for 5-fold cross-validation on 500 observations by using `cvp = cvpartition(500, 'KFold', 5)`. Then, you can specify the cross-validation partition for `fitcecoc` as 'CVPartition', `cvp`.

### **Holdout — Fraction of data for holdout validation**

scalar value in the range (0,1)

Fraction of data used for holdout validation, specified as the comma-separated pair consisting of 'Holdout' and a scalar value in the range (0,1). If you specify 'Holdout',  $p$ , then the software:

- 1 Randomly reserves  $p*100\%$  of the data as validation data, and trains the model using the rest of the data
- 2 Stores the compact, trained model in the `Trained` property of the cross-validated model.

To create a cross-validated model, you can use one of these four name-value pair arguments only: `CVPartition`, `Holdout`, `KFold`, or `Leaveout`.

Example: 'Holdout', 0.1

Data Types: `double` | `single`

### **KFold — Number of folds**

10 (default) | positive integer value greater than 1

Number of folds to use in a cross-validated model, specified as the comma-separated pair consisting of 'KFold' and a positive integer value greater than 1. If you specify, e.g., 'KFold',  $k$ , then the software:

- 1 Randomly partitions the data into  $k$  sets
- 2 For each set, reserves the set as validation data, and trains the model using the other  $k - 1$  sets
- 3 Stores the  $k$  compact, trained models in the cells of a  $k$ -by-1 cell vector in the `Trained` property of the cross-validated model.

To create a cross-validated model, you can use one of these four name-value pair arguments only: `CVPartition`, `Holdout`, `KFold`, or `Leaveout`.

Example: `'KFold', 5`

Data Types: `single` | `double`

#### **Leaveout — Leave-one-out cross-validation flag**

`'off'` (default) | `'on'`

Leave-one-out cross-validation flag, specified as the comma-separated pair consisting of 'Leaveout' and 'on' or 'off'. If you specify 'Leaveout', 'on', then, for each of the  $n$  observations, where  $n$  is `size(Mdl.X, 1)`, the software:

- 1 Reserves the observation as validation data, and trains the model using the other  $n - 1$  observations
- 2 Stores the  $n$  compact, trained models in the cells of an  $n$ -by-1 cell vector in the `Trained` property of the cross-validated model.

To create a cross-validated model, you can use one of these four name-value pair arguments only: `CVPartition`, `Holdout`, `KFold`, or `Leaveout`.

Example: `'Leaveout', 'on'`

Data Types: `char`

## Output Arguments

### **cvmdl — Cross-validated naive Bayes classifier**

`ClassificationPartitionedModel` model

Cross-validated naive Bayes classifier, returned as a `ClassificationPartitionedModel` model.

## Examples

### Cross Validate a Naive Bayes Classifier Using `crossval`

Load the `ionosphere` data set.

```
load ionosphere
X = X(:,3:end); % Remove first two predictors for stability
rng(1);        % For reproducibility
```

Train a naive Bayes classifier. It is good practice to define the class order. Assume that each predictor is conditionally, normally distributed given its label.

```
Mdl = fitcnb(X,Y,'ClassNames',{'b','g'});
```

`Mdl` is a trained `ClassificationNaiveBayes` classifier. 'b' is the negative class and 'g' is the positive class.

Cross validate the classifier using 10-fold cross validation.

```
CVMDl = crossval(Mdl)
FirstModel = CVMDl.Trained{1}
```

```
CVMDl =
```

```
classreg.learning.partition.ClassificationPartitionedModel
  CrossValidatedModel: 'NaiveBayes'
  PredictorNames: {1x32 cell}
  ResponseName: 'Y'
  NumObservations: 351
  KFold: 10
  Partition: [1x1 cvpartition]
  ClassNames: {'b' 'g'}
  ScoreTransform: 'none'
```

```
FirstModel =  
  
classreg.learning.classif.CompactClassificationNaiveBayes  
    ResponseName: 'Y'  
    CategoricalPredictors: []  
        ClassNames: {'b' 'g'}  
        ScoreTransform: 'none'  
        DistributionNames: {1x32 cell}  
        DistributionParameters: {2x32 cell}
```

CVMD1 is a `ClassificationPartitionedModel` cross-validated classifier. The software:

- 1 Randomly partitions the data into 10, equally sized sets.
- 2 Trains a naive Bayes classifier on nine of the sets.
- 3 Repeats steps 1 and 2  $k = 10$  times. It excludes one partition each time, and trains on the other nine partitions.
- 4 Combines generalization statistics for each fold.

FirstModel is the first of the 10 trained classifiers. It is a `CompactClassificationNaiveBayes` model.

You can estimate the generalization error by passing CVMD1 to `kfoldLoss`.

### Specify a Holdout-Sample Proportion for Naive Bayes Cross Validation

By default, `crossval` uses 10-fold cross validation to cross validate a naive Bayes classifier. You have several other options, such as specifying a different number of folds or holdout-sample proportion. This example shows how to specify a holdout-sample proportion.

Load the ionosphere data set.

```
load ionosphere  
X = X(:,3:end); % Remove first two predictors for stability  
rng(1); % For reproducibility
```

Train a naive Bayes classifier. Assume that each predictor is conditionally, normally distributed given its label. It is good practice to define the class order.

```
Mdl = fitcnb(X,Y,'ClassNames',{'b','g'});
```

Mdl is a trained ClassificationNaiveBayes classifier. 'b' is the negative class and 'g' is the positive class.

Cross validate the classifier by specifying a 30% holdout sample.

```
CVMdl = crossval(Mdl,'Holdout',0.30)
TrainedModel = CVMdl.Trained{1}
```

```
CVMdl =
```

```
classreg.learning.partition.ClassificationPartitionedModel
  CrossValidatedModel: 'NaiveBayes'
  PredictorNames: {1x32 cell}
  ResponseName: 'Y'
  NumObservations: 351
  KFold: 1
  Partition: [1x1 cvpartition]
  ClassNames: {'b' 'g'}
  ScoreTransform: 'none'
```

```
TrainedModel =
```

```
classreg.learning.classif.CompactClassificationNaiveBayes
  ResponseName: 'Y'
  CategoricalPredictors: []
  ClassNames: {'b' 'g'}
  ScoreTransform: 'none'
  DistributionNames: {1x32 cell}
  DistributionParameters: {2x32 cell}
```

CVMdl is a ClassificationPartitionedModel. TrainedModel is a CompactClassificationNaiveBayes classifier trained using 70% of the data.

Estimate the generalization error.

```
kfoldLoss(CVMdl)
```

```
ans =
```

0.2571

The out-of-sample misclassification error is approximately 2.6%.

## Tips

Assess the predictive performance of `Mdl` on cross-validated data using the “`kfold`” function and properties of `CVMdl`, such as `kfoldLoss`.

## Alternatives

Instead of creating a naive Bayes classifier followed by a cross-validation classifier, create a cross-validated classifier directly using `fitcnb` and by specifying any of these name-value pair arguments: `'CrossVal'`, `'CVPartition'`, `'Holdout'`, `'Leaveout'`, or `'KFold'`.

## See Also

`ClassificationNaiveBayes` | `ClassificationPartitionedModel` |  
`CompactClassificationNaiveBayes` | `fitcnb` | `kfoldLoss`

# crossval

**Class:** ClassificationSVM

Cross-validated support vector machine classifier

## Syntax

```
CVSVMModel = crossval(SVMModel)
CVSVMModel = crossval(SVMModel, Name, Value)
```

## Description

`CVSVMModel = crossval(SVMModel)` returns a cross-validated (partitioned) support vector machine classifier (`CVSVMModel`) from a trained SVM classifier (`SVMModel`).

By default, `crossval` uses 10-fold cross validation on the training data to create `CVSVMModel`.

`CVSVMModel = crossval(SVMModel, Name, Value)` returns a partitioned SVM classifier with additional options specified by one or more `Name, Value` pair arguments.

For example, you can specify the number of folds or holdout sample proportion.

## Input Arguments

**SVMModel** — Full, trained SVM classifier

ClassificationSVM classifier

Full, trained SVM classifier, specified as a `ClassificationSVM` model trained using `fitcsvm`.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single

quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

### **CVPartition — Cross-validation partition**

`[]` (default) | `cvpartition` partition object

Cross-validation partition, specified as the comma-separated pair consisting of 'CVPartition' and a `cvpartition` partition object as created by `cvpartition`. The partition object specifies the type of cross-validation, and also the indexing for training and validation sets.

To create a cross-validated model, you can use one of these four name-value pair arguments only: `CVPartition`, `Holdout`, `KFold`, or `Leaveout`.

Example: Suppose you create a random partition for 5-fold cross-validation on 500 observations by using `cvp = cvpartition(500, 'KFold', 5)`. Then, you can specify the cross-validation partition for `fitcecoc` as 'CVPartition', `cvp`.

### **Holdout — Fraction of data for holdout validation**

scalar value in the range (0,1)

Fraction of data used for holdout validation, specified as the comma-separated pair consisting of 'Holdout' and a scalar value in the range (0,1). If you specify 'Holdout',  $p$ , then the software:

- 1 Randomly reserves  $p*100\%$  of the data as validation data, and trains the model using the rest of the data
- 2 Stores the compact, trained model in the `Trained` property of the cross-validated model.

To create a cross-validated model, you can use one of these four name-value pair arguments only: `CVPartition`, `Holdout`, `KFold`, or `Leaveout`.

Example: 'Holdout', 0.1

Data Types: `double` | `single`

### **KFold — Number of folds**

10 (default) | positive integer value greater than 1



Number of folds to use in a cross-validated model, specified as the comma-separated pair consisting of 'KFold' and a positive integer value greater than 1. If you specify, e.g., 'KFold',  $k$ , then the software:

- 1 Randomly partitions the data into  $k$  sets
- 2 For each set, reserves the set as validation data, and trains the model using the other  $k - 1$  sets
- 3 Stores the  $k$  compact, trained models in the cells of a  $k$ -by-1 cell vector in the `Trained` property of the cross-validated model.

To create a cross-validated model, you can use one of these four name-value pair arguments only: `CVPartition`, `Holdout`, `KFold`, or `Leaveout`.

Example: 'KFold', 5

Data Types: `single` | `double`

#### **Leaveout** — Leave-one-out cross-validation flag

'off' (default) | 'on'

Leave-one-out cross-validation flag, specified as the comma-separated pair consisting of 'Leaveout' and 'on' or 'off'. If you specify 'Leaveout', 'on', then, for each of the  $n$  observations, where  $n$  is `size(Mdl.X, 1)`, the software:

- 1 Reserves the observation as validation data, and trains the model using the other  $n - 1$  observations
- 2 Stores the  $n$  compact, trained models in the cells of an  $n$ -by-1 cell vector in the `Trained` property of the cross-validated model.

To create a cross-validated model, you can use one of these four name-value pair arguments only: `CVPartition`, `Holdout`, `KFold`, or `Leaveout`.

Example: 'Leaveout', 'on'

Data Types: `char`

## Output Arguments

### **CVSVMModel** — Cross-validated SVM classifier

`ClassificationPartitionedModel` classifier

Cross-validated SVM classifier, returned as a `ClassificationPartitionedModel` classifier.

## Examples

### Cross Validate an SVM Classifier Using `crossval`

Load the ionosphere data set.

```
load ionosphere
rng(1); % For reproducibility
```

Train an SVM classifier. It is good practice to standardize the predictors and define the class order.

```
SVMModel = fitcsvm(X,Y,'Standardize',true,'ClassNames',{'b','g'});
```

`SVMModel` is a trained `ClassificationSVM` classifier. 'b' is the negative class and 'g' is the positive class.

Cross validate the classifier using 10-fold cross validation.

```
CVSVMModel = crossval(SVMModel)
FirstModel = CVSVMModel.Trained{1}
```

```
CVSVMModel =
```

```
classreg.learning.partition.ClassificationPartitionedModel
  CrossValidatedModel: 'SVM'
  PredictorNames: {1x34 cell}
  ResponseName: 'Y'
  NumObservations: 351
  KFold: 10
  Partition: [1x1 cvpartition]
  ClassNames: {'b' 'g'}
  ScoreTransform: 'none'
```

```
FirstModel =
```

```

classreg.learning.classif.CompactClassificationSVM
    ResponseName: 'Y'
    CategoricalPredictors: []
        ClassNames: {'b' 'g'}
        ScoreTransform: 'none'
            Alpha: [78x1 double]
            Bias: -0.2209
        KernelParameters: [1x1 struct]
            Mu: [1x34 double]
            Sigma: [1x34 double]
        SupportVectors: [78x34 double]
        SupportVectorLabels: [78x1 double]

```

`CVSVMModel` is a `ClassificationPartitionedModel` cross-validated classifier. The software:

- 1 Randomly partitions the data into 10, equally sized sets.
- 2 Trains an SVM classifier on nine of the sets.
- 3 Repeats steps 1 and 2  $k = 10$  times. It leaves out one of the partitions each time, and trains on the other nine partitions.
- 4 Combines generalization statistics for each fold.

`FirstModel` is the first of the 10 trained classifiers. It is a `CompactClassificationSVM` classifier.

You can estimate the generalization error by passing `CVSVMModel` to `kfoldLoss`.

### Specify a Holdout-Sample Proportion for SVM Cross Validation

By default, `crossval` uses 10-fold cross validation to cross validate an SVM classifier. You have several other options, such as specifying a different number of folds or holdout sample proportion. This example shows how to specify a holdout-sample proportion.

Load the ionosphere data set.

```

load ionosphere
rng(1); % For reproducibility

```

Train an SVM classifier. It is good practice to standardize the predictors and define the class order.

```
SVModel = fitcsvm(X,Y,'Standardize',true,'ClassNames',{'b','g'});
```

SVModel is a trained ClassificationSVM classifier. 'b' is the negative class and 'g' is the positive class.

Cross validate the classifier by specifying a 15% holdout sample.

```
CVSVModel = crossval(SVModel,'Holdout',0.15)
TrainedModel = CVSVModel.Trained{1}
```

```
CVSVModel =
```

```
classreg.learning.partition.ClassificationPartitionedModel
  CrossValidatedModel: 'SVM'
  PredictorNames: {1x34 cell}
  ResponseName: 'Y'
  NumObservations: 351
  KFold: 1
  Partition: [1x1 cvpartition]
  ClassNames: {'b' 'g'}
  ScoreTransform: 'none'
```

```
TrainedModel =
```

```
classreg.learning.classif.CompactClassificationSVM
  ResponseName: 'Y'
  CategoricalPredictors: []
  ClassNames: {'b' 'g'}
  ScoreTransform: 'none'
  Alpha: [74x1 double]
  Bias: -0.2952
  KernelParameters: [1x1 struct]
  Mu: [1x34 double]
  Sigma: [1x34 double]
  SupportVectors: [74x34 double]
  SupportVectorLabels: [74x1 double]
```

`CVSVMModel` is a `ClassificationPartitionedModel`. `TrainedModel` is a `CompactClassificationSVM` classifier trained using 85% of the data.

Estimate the generalization error.

```
kfoldLoss(CVSVMModel)
```

```
ans =
```

```
    0.0769
```

The out-of-sample misclassification error is approximately 8%.

## Tips

Assess the predictive performance of `SVMModel` on cross-validated data using the “kfold” methods and properties of `CVSVMModel`, such as `kfoldLoss`.

## Alternatives

Instead of training an SVM classifier and then cross-validating it, you can create a cross-validated classifier directly using `fitcsvm` and by specifying any of these name-value pair arguments: `'CrossVal'`, `'CVPartition'`, `'Holdout'`, `'Leaveout'`, or `'Kfold'`.

## See Also

`ClassificationPartitionedModel` | `ClassificationSVM` |  
`CompactClassificationSVM` | `cvpartition` | `fitcsvm`

## crossval

**Class:** ClassificationTree

Cross-validated decision tree

### Syntax

```
cvmodel = crossval(model)
cvmodel = crossval(model,Name,Value)
```

### Description

`cvmodel = crossval(model)` creates a partitioned model from `model`, a fitted classification tree. By default, `crossval` uses 10-fold cross validation on the training data to create `cvmodel`.

`cvmodel = crossval(model,Name,Value)` creates a partitioned model with additional options specified by one or more `Name,Value` pair arguments.

### Input Arguments

#### **model**

A classification model, produced using `fitctree`.

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

**CVPartition**

Object of class `cvpartition`, created by the `cvpartition` function. `crossval` splits the data into subsets with `cvpartition`.

Use only one of these four options at a time: 'KFold', 'Holdout', 'Leaveout', or 'CVPartition'.

**Holdout**

Holdout validation tests the specified fraction of the data, and uses the remaining data for training. Specify a numeric scalar from 0 to 1. Use only one of these four options at a time: 'KFold', 'Holdout', 'Leaveout', or 'CVPartition'.

**KFold**

Number of folds to use in a cross-validated tree, a positive integer greater than 1.

Use only one of these four options at a time: 'KFold', 'Holdout', 'Leaveout', or 'CVPartition'.

**Default:** 10

**Leaveout**

Set to 'on' for leave-one-out cross validation.

Use only one of these four options at a time: 'KFold', 'Holdout', 'Leaveout', or 'CVPartition'.

## Output Arguments

**cvmodel**

Partitioned model of class `ClassificationPartitionedModel`.

## Examples

### Create a Cross-Validation Model

Create a classification model for the ionosphere data, then create a cross-validation model. Evaluate the quality the model using `kfoldLoss`.

```
load ionosphere
tree = fitctree(X,Y);
cvmodel = crossval(tree);
L = kfoldLoss(cvmodel)
```

```
L =
    0.1111
```

### Tips

- Assess the predictive performance of model on cross-validated data using the “kfold” methods and properties of `cvmodel`, such as `kfoldLoss`.

### Alternatives

You can create a cross-validation tree directly from the data, instead of creating a decision tree followed by a cross-validation tree. To do so, include one of these five options in `fitctree`: `'CrossVal'`, `'KFold'`, `'Holdout'`, `'Leaveout'`, or `'CVPartition'`.

### See Also

`crossval` | `fitctree`



# crossval

**Class:** RegressionEnsemble

Cross validate ensemble

## Syntax

```
cvens = crossval(ens)
cvens = crossval(ens,Name,Value)
```

## Description

`cvens = crossval(ens)` creates a cross-validated ensemble from `ens`, a regression ensemble. Default is 10-fold cross validation.

`cvens = crossval(ens,Name,Value)` creates a cross-validated ensemble with additional options specified by one or more `Name, Value` pair arguments. You can specify several name-value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

## Input Arguments

**ens**

A regression ensemble created with `fitrensemble`.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

**cvpartition**

A partition of class `cvpartition`. Sets the partition for cross validation.

Use no more than one of the name-value pairs `cvpartition`, `holdout`, `kfold`, and `leaveout`.

**holdout**

Holdout validation tests the specified fraction of the data, and uses the rest of the data for training. Specify a numeric scalar from 0 to 1. You can only use one of these four options at a time for creating a cross-validated tree: `'kfold'`, `'holdout'`, `'leaveout'`, or `'cvpartition'`.

**kfold**

Number of folds for cross validation, a positive integer value greater than 1.

Use no more than one of the name-value pairs `'kfold'`, `'holdout'`, `'leaveout'`, or `'cvpartition'`.

**leaveout**

If `'on'`, use leave-one-out cross-validation.

Use no more than one of the name-value pairs `'kfold'`, `'holdout'`, `'leaveout'`, or `'cvpartition'`.

**nprint**

Printout frequency, a positive integer scalar. Use this parameter to observe the training of cross-validation folds.

**Default:** `'off'`, meaning no printout

## Output Arguments

**cvens**

A cross-validated classification ensemble of class `RegressionPartitionedEnsemble`.

## Examples

### Create Cross-Validated Regression Model

Create a cross-validated regression model for the `carsmall` data, and evaluate its quality using the `kfoldLoss` method.

Load the `carsmall` data set and select acceleration, displacement, horsepower, and vehicle weight as predictors.

```
load carsmall;
X = [Acceleration Displacement Horsepower Weight];
```

Train a regression ensemble.

```
rens = fitrensemble(X,MPG);
```

Create a cross-validated ensemble from `rens` and find the cross-validation loss.

```
rng(10,'twister') % For reproducibility
cvens = crossval(rens);
L = kfoldLoss(cvens)

L = 31.8744
```

## Alternatives

You can create a cross-validation ensemble directly from the data, instead of creating an ensemble followed by a cross-validation ensemble. To do so, include one of these five options in `fitrensemble`: `'crossval'`, `'kfold'`, `'holdout'`, `'leaveout'`, or `'cvpartition'`.

## See Also

`RegressionPartitionedEnsemble` | `cvpartition` | `kfoldLoss`

## crossval

**Class:** RegressionGP

Cross-validate Gaussian process regression model

## Syntax

```
cvMdl = crossval(gprMdl)
cvmdl = crossval(gprMdl, Name, Value)
```

## Description

`cvMdl = crossval(gprMdl)` returns the partitioned model, `cvMdl`, built from the Gaussian process regression (GPR) model, `gprMdl`, using 10-fold cross validation.

`cvmdl` is a `RegressionPartitionedModel` object, and `gprMdl` is a `RegressionGP` (full) object.

`cvmdl = crossval(gprMdl, Name, Value)` returns the partitioned model, `cvmdl`, with additional options specified by one or more `Name, Value` pair arguments. For example, you can specify the number of folds or the fraction of the data to use for testing.

## Input Arguments

**gprMdl** — Gaussian process regression model

`RegressionGP` object

Gaussian process regression model, specified as a `RegressionGP` (full) object. You cannot call `crossval` on a compact regression object.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single

quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### **CVPartition** — Random partition for a $k$ -fold cross validation

`cvpartition` object

Random partition for a  $k$ -fold cross validation, specified as the comma-separated pair consisting of 'CVPartition' and a `cvpartition` object.

Example: 'CVPartition', `cvp` uses the random partition defined by `cvp`.

If you specify `CVPartition`, then you cannot specify `Holdout`, `KFold`, or `LeaveOut`.

### **Holdout** — Fraction of data to use for testing

Scalar value in the range from 0 to 1

Fraction of the data to use for testing in holdout validation, specified as the comma-separated pair consisting of 'Holdout' and a scalar value in the range from 0 to 1. If you specify 'Holdout', `p`, then `crossval`:

1. Randomly reserves  $p*100\%$  of the data as validation data, and trains the model using the rest of the data
2. Stores the compact, trained model in `cvgprMdl.Trained`.

Example: 'Holdout', 0.3 uses 30% of the data for testing and 70% of the data for training.

If you specify `Holdout`, then you cannot specify `CVPartition`, `KFold`, or `Leaveout`.

Data Types: `single` | `double`

### **KFold** — Number of folds

10 (default) | positive integer value greater than 1

Number of folds to use in cross-validated GPR model, specified as the comma-separated pair consisting of 'KFold' and a positive integer value greater than 1. `Kfold` must be greater than 1. If you specify 'Kfold', `k` then `crossval`:

1. Randomly partitions the data into  $k$  sets.
2. For each set, reserves the set as test data, and trains the model using the other  $k - 1$  sets.
3. Stores the  $k$  compact, trained models in the cells of a  $k$ -by-1 cell array in `cvgprMdl.Trained`.

Example: `'KFold', 5` uses 5 folds in cross-validation. That is, for each fold, it uses that fold as test data, and trains the model on the remaining 4 folds.

If you specify `KFold`, then you cannot specify `CVPartition`, `Holdout`, or `Leaveout`.

Data Types: `single` | `double`

### **Leaveout** — Indicator for leave-one-out cross-validation

`'off'` (default) | `'on'`

Indicator for leave-one-out cross-validation, specified as the comma-separated pair consisting of `'LeaveOut'` and either `'on'` or `'off'`. If you specify `'Leaveout', 'on'`, then, for each of the  $n$  observations, `crossval`:

1. Reserves the observation as test data, and trains the model using the other  $n - 1$  observations.
2. Stores the  $n$  compact, trained models in the cells of a  $n$ -by-1 cell array in `cvgprMdl.Trained`.

Example: `'Leaveout', 'on'`

If you specify `Leaveout`, then you cannot specify `CVPartition`, `Holdout`, or `KFold`.

## Output Arguments

### **cvgprMdl** — Partitioned Gaussian process regression model

`RegressionPartitionedModel` object

Partitioned Gaussian process regression model, returned as a `RegressionPartitionedModel` object.

## Examples

### **Partition Data for Cross-Validation**

Download the housing data [1], from the UCI Machine Learning Repository [4].

The dataset has 506 observations. The first 13 columns contain the predictor values and the last column contains the response values. The goal is to predict the median value of owner-occupied homes in suburban Boston as a function of 13 predictors.

Load the data and define the response vector and the predictor matrix.

```
load('housing.data');
X = housing(:,1:13);
y = housing(:,end);
```

Fit a GPR model using the squared exponential kernel function with separate length scale for each predictor. Standardize the predictor variables.

```
gprMdl = fitrgp(X,y,'KernelFunction','ardsquaredexponential','Standardize',1);
```

Create a cross-validation partition for data using predictor 4 as a grouping variable.

```
rng('default') % For reproducibility
cvp = cvpartition(X(:,4),'kfold',10);
```

Create a 10-fold cross-validated model using the partitioned data in `cvp`.

```
cvgprMdl = crossval(gprMdl,'CVPartition',cvp);
```

Compute the regression loss for in-fold observations using models trained on out-of-fold observations.

```
L = kfoldLoss(cvgprMdl)
```

```
L =
```

```
9.5299
```

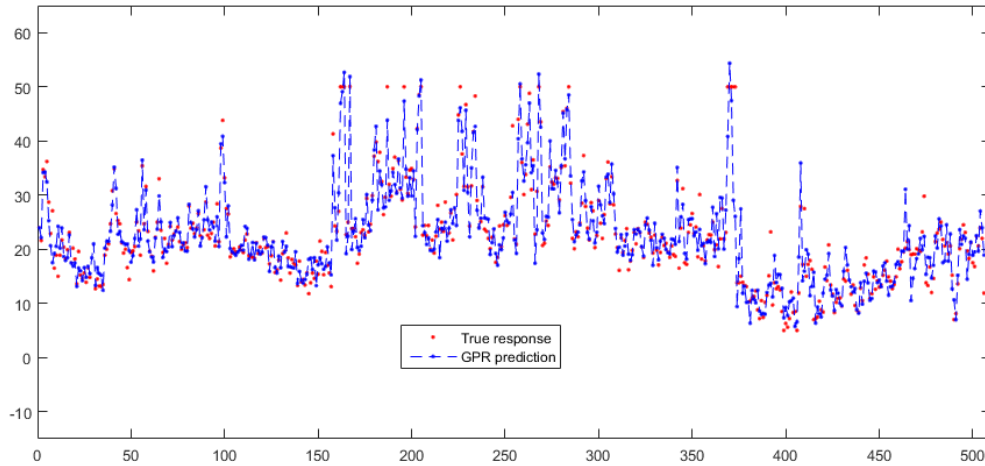
Predict the response for in-fold observations, i.e. observations not used for training.

```
ypred = kfoldPredict(cvgprMdl);
```

For every fold, `kfoldPredict` predicts responses for observations in that fold using the models trained on out-of-fold observations.

Plot the actual responses and prediction data.

```
plot(y,'r. ');
hold on;
plot(ypred,'b--. ');
axis([0 510 -15 65]);
legend('True response','GPR prediction','Location','Best');
hold off;
```



### Train GPR Model Using 4-Fold Cross Validation

Download the abalone data [2], [3], from the UCI Machine Learning Repository [4] and save it in your current directory with the name 'abalone.data'.

Read the data into a table.

```
tbl = readtable('abalone.data', 'Filetype', 'text', 'ReadVariableNames', false);
```

The dataset has 4177 observations. The goal is to predict the age of abalone from 8 physical measurements.

Fit a GPR model using the subset of regressors (`sr`) method for parameter estimation and fully independent conditional (`fic`) method for prediction. Standardize the predictors and use a squared exponential kernel function with a separate length scale for each predictor.

```
gprMdl = fitrgp(tbl, tbl(:, end), 'KernelFunction', 'ardsquaredexponential', ...
    'FitMethod', 'sr', 'PredictMethod', 'fic', 'Standardize', 1);
```

Cross-validate the model using 4-fold cross validation. This partitions the data into 4 sets. For each set, `fitrgp` uses that set (25% of the data) as the test data, and trains the model on the remaining 3 sets (75% of the data).



```
rng('default') % For reproducibility
cvgprMdl = crossval(gprMdl, 'KFold', 4);
```

Compute the loss over individual folds.

```
L = kfoldLoss(cvgprMdl, 'mode', 'individual')
```

```
L =
```

```
    4.3669
    4.6896
    4.0565
    4.3162
```

Compute the average cross-validated loss on over all folds. The default is the mean squared error.

```
L2 = kfoldLoss(cvgprMdl)
```

```
L2 =
```

```
    4.3573
```

This is equal to the mean loss over individual folds.

```
mse = mean(L)
```

```
mse =
```

```
    4.3573
```

- “Train GPR Model Using Cross-Validation” on page 32-2341

## Tips

- You can only use one of the name-value pair arguments at a time.
- You can not compute the prediction intervals for a cross-validated model.

## Alternatives

Alternatively, you can train a cross-validated model using the related name-value pair arguments in `fitrgp`.

If you supply a custom 'ActiveSet' in the call to `fitrgp`, then you cannot cross validate the GPR model.

## References

- [1] Harrison, D. and D.L., Rubinfeld. "Hedonic prices and the demand for clean air." *J. Environ. Economics & Management*. Vol.5, 1978, pp. 81-102.
- [2] Warwick J. N., T. L. Sellers, S. R. Talbot, A. J. Cawthorn, and W. B. Ford. "The Population Biology of Abalone (*Haliotis* species) in Tasmania. I. Blacklip Abalone (*H. rubra*) from the North Coast and Islands of Bass Strait." Sea Fisheries Division, Technical Report No. 48 (ISSN 1034-3288), 1994.
- [3] S. Waugh. "Extending and Benchmarking Cascade-Correlation", PhD Thesis. Computer Science Department, University of Tasmania, 1995.
- [4] Lichman, M. UCI Machine Learning Repository, Irvine, CA: University of California, School of Information and Computer Science, 2013. <http://archive.ics.uci.edu/ml>.

## See Also

`RegressionGP` | `RegressionPartitionedModel` | `fitrgp` | `kfoldLoss` | `kfoldPredict`

## Topics

"Train GPR Model Using Cross-Validation" on page 32-2341

**Introduced in R2015b**

# crossval

**Class:** RegressionSVM

Cross-validated support vector machine regression model

## Syntax

```
CVMD1 = crossval mdl
CVMD1 = crossval mdl, Name, Value
```

## Description

`CVMD1 = crossval mdl` returns a cross-validated (partitioned) support vector machine regression model, `CVMD1`, from a trained SVM regression model, `mdl`.

`CVMD1 = crossval mdl, Name, Value` returns a cross-validated model with additional options specified by one or more `Name, Value` pair arguments.

## Input Arguments

**mdl** — Full, trained SVM regression model

RegressionSVM model

Full, trained SVM regression model, specified as a `RegressionSVM` model returned by `fitrsvm`.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

**CVPartition — Cross-validation partition**[] (default) | `cvpartition` partition object

Cross-validation partition, specified as the comma-separated pair consisting of 'CVPartition' and a `cvpartition` partition object as created by `cvpartition`. The partition object specifies the type of cross-validation, and also the indexing for training and validation sets.

To create a cross-validated model, you can use one of these four name-value pair arguments only: `CVPartition`, `Holdout`, `KFold`, or `Leaveout`.

Example: Suppose you create a random partition for 5-fold cross-validation on 500 observations by using `cvp = cvpartition(500, 'KFold', 5)`. Then, you can specify the cross-validation partition for `fitcecoc` as 'CVPartition', `cvp`.

**Holdout — Fraction of data for holdout validation**

scalar value in the range (0,1)

Fraction of data used for holdout validation, specified as the comma-separated pair consisting of 'Holdout' and a scalar value in the range (0,1). If you specify 'Holdout',  $p$ , then the software:

- 1 Randomly reserves  $p*100\%$  of the data as validation data, and trains the model using the rest of the data
- 2 Stores the compact, trained model in the `Trained` property of the cross-validated model.

To create a cross-validated model, you can use one of these four name-value pair arguments only: `CVPartition`, `Holdout`, `KFold`, or `Leaveout`.

Example: 'Holdout', 0.1

Data Types: `double` | `single`

**KFold — Number of folds**

10 (default) | positive integer value greater than 1

Number of folds to use in a cross-validated model, specified as the comma-separated pair consisting of 'KFold' and a positive integer value greater than 1. If you specify, e.g., 'KFold',  $k$ , then the software:

- 1 Randomly partitions the data into  $k$  sets

- 2 For each set, reserves the set as validation data, and trains the model using the other  $k - 1$  sets
- 3 Stores the  $k$  compact, trained models in the cells of a  $k$ -by-1 cell vector in the `Trained` property of the cross-validated model.

To create a cross-validated model, you can use one of these four name-value pair arguments only: `CVPartition`, `Holdout`, `KFold`, or `Leaveout`.

Example: `'KFold', 5`

Data Types: `single` | `double`

### **Leaveout** — Leave-one-out cross-validation flag

`'off'` (default) | `'on'`

Leave-one-out cross-validation flag, specified as the comma-separated pair consisting of `'Leaveout'` and `'on'` or `'off'`. If you specify `'Leaveout', 'on'`, then, for each of the  $n$  observations, where  $n$  is `size(Mdl.X, 1)`, the software:

- 1 Reserves the observation as validation data, and trains the model using the other  $n - 1$  observations
- 2 Stores the  $n$  compact, trained models in the cells of an  $n$ -by-1 cell vector in the `Trained` property of the cross-validated model.

To create a cross-validated model, you can use one of these four name-value pair arguments only: `CVPartition`, `Holdout`, `KFold`, or `Leaveout`.

Example: `'Leaveout', 'on'`

Data Types: `char`

## Output Arguments

### **cvmdl** — Cross-validated SVM regression model

`RegressionPartitionedSVM` model

Cross-validated SVM regression model, returned as a `RegressionPartitionedSVM` model.

## Examples

### Train Cross-Validated SVM Regression Model Using `crossval`

This example shows how to train a cross-validated SVM regression model using `crossval`.

This example uses the abalone data from the UCI Machine Learning Repository. Download the data and save it in your current folder with the name 'abalone.data'. Read the data into a table.

```
tbl = readtable('abalone.data','Filetype','text','ReadVariableNames',false);  
rng default % for reproducibility
```

The sample data contains 4177 observations. All the predictor variables are continuous except for `sex`, which is a categorical variable with possible values 'M' (for males), 'F' (for females), and 'I' (for infants). The goal is to predict the number of rings on the abalone and determine its age using physical measurements.

Train an SVM regression model, using a Gaussian kernel function with a kernel scale equal to 2.2. Standardize the data.

```
mdl = fitrsvm(tbl,'Var9','KernelFunction','gaussian','KernelScale',2.2,'Standardize',true)
```

`mdl` is a trained `RegressionSVM` regression model.

Cross validate the model using 10-fold cross validation.

```
CVMDL = crossval(mdl)
```

```
CVMDL =
```

```
classreg.learning.partition.RegistrationPartitionedSVM  
  CrossValidatedModel: 'SVM'  
  PredictorNames: {1x8 cell}  
  CategoricalPredictors: 1  
  ResponseName: 'Var9'  
  NumObservations: 4177  
  KFold: 10  
  Partition: [1x1 cvpartition]  
  ResponseTransform: 'none'
```

Properties, Methods

CVMD1 is a RegressionPartitionedSVM cross-validated regression model. The software:

1. Randomly partitions the data into 10 equally sized sets.
2. Trains an SVM regression model on nine of the 10 sets.
3. Repeats steps 1 and 2  $k = 10$  times. It leaves out one of the partitions each time, and trains on the other nine partitions.
4. Combines generalization statistics for each fold.

Calculate the resubstitution loss for the cross-validated model.

```
loss = kfoldLoss(CVMD1)
```

```
loss =
```

```
    4.5712
```

### Specify Cross-Validation Holdout Proportion for SVM Regression

This example shows how to specify a holdout proportion for training a cross-validated SVM regression model.

This example uses the abalone data from the UCI Machine Learning Repository. Download the data and save it in your current folder with the name 'abalone.data'. Read the data into a table.

```
tbl = readtable('abalone.data', 'Filetype', 'text', 'ReadVariableNames', false);
rng default % for reproducibility
```

The sample data contains 4177 observations. All the predictor variables are continuous except for sex, which is a categorical variable with possible values 'M' (for males), 'F' (for females), and 'I' (for infants). The goal is to predict the number of rings on the abalone and determine its age using physical measurements.

Train an SVM regression model, using a Gaussian kernel function with an automatic kernel scale. Standardize the data.

```
mdl = fitrsvm(tbl, 'Var9', 'KernelFunction', 'gaussian', 'KernelScale', 'auto', 'Standardize')
```

mdl is a trained `RegressionSVM` regression model.

Cross validate the regression model by specifying a 10% holdout sample.

```
CVmdl = crossval(mdl, 'Holdout', 0.1)
```

```
CVmdl =
```

```
classreg.learning.partition.RegistrationPartitionedSVM
  CrossValidatedModel: 'SVM'
  PredictorNames: {1x8 cell}
  CategoricalPredictors: 1
  ResponseName: 'Var9'
  NumObservations: 4177
  KFold: 1
  Partition: [1x1 cvpartition]
  ResponseTransform: 'none'
```

```
Properties, Methods
```

CVmdl is a `RegistrationPartitionedSVM` model object.

Calculate the resubstitution loss for the cross-validated model.

```
loss = kfoldLoss(CVmdl)
```

```
loss =
```

```
5.2499
```

## Alternatives

Instead of training an SVM regression model and then cross-validating it, you can create a cross-validated model directly by using `fitrsvm` and specifying any of these name-value pair arguments: 'CrossVal', 'CVPartition', 'Holdout', 'Leaveout', or 'KFold'.



## References

- [1] Nash, W.J., T. L. Sellers, S. R. Talbot, A. J. Cawthorn, and W. B. Ford. *The Population Biology of Abalone (Haliotis species) in Tasmania. I. Blacklip Abalone (H. rubra) from the North Coast and Islands of Bass Strait*, Sea Fisheries Division, Technical Report No. 48, 1994.
- [2] Waugh, S. *Extending and benchmarking Cascade-Correlation*, Ph.D. thesis, Computer Science Department, University of Tasmania, 1995.
- [3] Clark, D., Z. Schreter, A. Adams. *A Quantitative Comparison of Dystal and Backpropagation*, submitted to the Australian Conference on Neural Networks, 1996.
- [4] Lichman, M. *UCI Machine Learning Repository*, [<http://archive.ics.uci.edu/ml>]. Irvine, CA: University of California, School of Information and Computer Science.

## See Also

`CompactRegressionSVM` | `RegressionPartitionedSVM` | `RegressionSVM` | `fitrsvm` | `kfoldLoss` | `kfoldPredict`

**Introduced in R2015b**

## crossval

**Class:** RegressionTree

Cross-validated decision tree

## Syntax

```
cvmodel = crossval(model)
cvmodel = crossval(model, Name, Value)
```

## Description

`cvmodel = crossval(model)` creates a partitioned model from `model`, a fitted regression tree. By default, `crossval` uses 10-fold cross validation on the training data to create `cvmodel`.

`cvmodel = crossval(model, Name, Value)` creates a partitioned model with additional options specified by one or more `Name, Value` pair arguments.

## Input Arguments

**model**

A regression model, produced using `fitrtree`.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

**CVPartition**

Object of class `cvpartition`, created by the `cvpartition` function. `crossval` splits the data into subsets with `cvpartition`.

Use only one of these four options at a time: 'KFold', 'Holdout', 'Leaveout', or 'CVPartition'.

**Default:** []

**Holdout**

Holdout validation tests the specified fraction of the data, and uses the rest of the data for training. Specify a numeric scalar from 0 to 1. You can only use one of these four options at a time for creating a cross-validated tree: 'KFold', 'Holdout', 'Leaveout', or 'CVPartition'.

**KFold**

Number of folds to use in a cross-validated tree, a positive integer value greater than 1.

Use only one of these four options at a time: 'KFold', 'Holdout', 'Leaveout', or 'CVPartition'.

**Default:** 10

**Leaveout**

Set to 'on' for leave-one-out cross-validation.

## Output Arguments

**cvmodel**

A partitioned model of class `RegressionPartitionedModel`.

## Examples

### Cross-Validate Regression Tree

Load the `carsmall` data set. Consider Acceleration, Displacement, Horsepower, and Weight as predictor variables.

```
load carsmall
X = [Acceleration Displacement Horsepower Weight];
```

Grow a regression tree using the entire data set.

```
Mdl = fitrtree(X,MPG);
```

Mdl is a `RegressionTree` model.

Cross-validate the regression tree using 10-fold cross-validation.

```
CVMD1 = crossval(Mdl);
```

CVMD1 is a `RegressionPartitionedModel` cross-validated model. `crossval` stores the ten trained, compact regression trees in the `Trained` property of CVMD1.

Display the compact regression tree that `crossval` trained using all observations except those in the first fold.

```
CVMD1.Trained{1}
```

```
ans =
```

```
classreg.learning.regr.CompactRegressionTree
    PredictorNames: {'x1' 'x2' 'x3' 'x4'}
    ResponseName: 'Y'
    CategoricalPredictors: []
    ResponseTransform: 'none'
```

Estimate the generalization error of Mdl by computing the 10-fold cross-validated mean-squared error.

```
L = kfoldLoss(CVMD1)
```

```
L =
```

23.8065

## Tips

- Assess the predictive performance of `model` on cross-validated data using the “kfold” methods and properties of `cvmodel`, such as `kfoldLoss`.

## Alternatives

You can create a cross-validation tree directly from the data, instead of creating a decision tree followed by a cross-validation tree. To do so, include one of these five options in `fitrtree`: `'CrossVal'`, `'KFold'`, `'Holdout'`, `'Leaveout'`, or `'CVPartition'`.

## See Also

`crossval` | `fitrtree`

## cutcategories

**Class:** classregtree

Cut categories

---

**Note** `classregtree` will be removed in a future release. See `fitctree`, `fitrtree`, `ClassificationTree`, or `RegressionTree` instead.

---

## Syntax

```
C = cutcategories(t)
C = cutcategories(t,nodes)
```

## Description

`C = cutcategories(t)` returns an  $n$ -by-2 cell array `C` of the categories used at branches in the decision tree `t`, where  $n$  is the number of nodes. For each branch node  $i$  based on a categorical predictor variable  $x$ , the left child is chosen if  $x$  is among the categories listed in `C{i,1}`, and the right child is chosen if  $x$  is among those listed in `C{i,2}`. Both columns of `C` are empty for branch nodes based on continuous predictors and for leaf nodes.

`C = cutcategories(t,nodes)` takes a vector `nodes` of node numbers and returns the categories for the specified nodes.

## Examples

Create a classification tree for car data:

```
load carsmall

t = classregtree([MPG Cylinders],Origin,...
                'names',{'MPG' 'Cyl'},'cat',2)

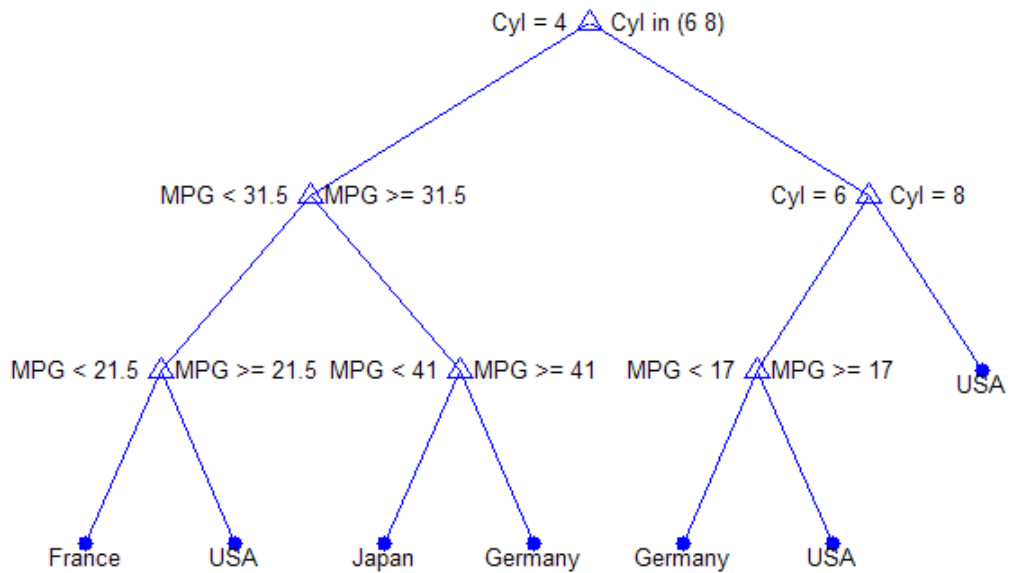
t =
Decision tree for classification
```

```

1 if Cyl=4 then node 2 elseif Cyl in {6 8} then node 3 else USA
2 if MPG<31.5 then node 4 elseif MPG>=31.5 then node 5 else USA
3 if Cyl=6 then node 6 elseif Cyl=8 then node 7 else USA
4 if MPG<21.5 then node 8 elseif MPG>=21.5 then node 9 else USA
5 if MPG<41 then node 10 elseif MPG>=41 then node 11 else Japan
6 if MPG<17 then node 12 elseif MPG>=17 then node 13 else USA
7 class = USA
8 class = France
9 class = USA
10 class = Japan
11 class = Germany
12 class = Germany
13 class = USA
view(t)

```

Click to display:  Magnification:  Pruning level:



```

C = cutcategories(t)
C =
    [4]    [1x2 double]
    []
    [6]    [      8]

```

```
      []          []
      []          []
      []          []
      []          []
      []          []
      []          []
      []          []
      []          []
      []          []
      []          []
C{1,2}
ans =
     6     8
```

## References

- [1] Breiman, L., J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Boca Raton, FL: CRC Press, 1984.

## See Also

[classregtree](#) | [cutpoint](#) | [cuttype](#) | [cutvar](#)



# cutpoint

**Class:** classregtree

Decision tree cut point values

---

**Note** `classregtree` will be removed in a future release. See `fitctree`, `fitrtree`, `ClassificationTree`, or `RegressionTree` instead.

---

## Syntax

```
v = cutpoint(t)
v = cutpoint(t,nodes)
```

## Description

`v = cutpoint(t)` returns an  $n$ -element vector  $v$  of the values used as cut points in the decision tree  $t$ , where  $n$  is the number of nodes. For each branch node  $i$  based on a continuous predictor variable  $x$ , the left child is chosen if  $x < v(i)$  and the right child is chosen if  $x \geq v(i)$ .  $v$  is NaN for branch nodes based on categorical predictors and for leaf nodes.

`v = cutpoint(t,nodes)` takes a vector `nodes` of node numbers and returns the cut points for the specified nodes.

## Examples

Create a classification tree for car data:

```
load carsmall

t = classregtree([MPG Cylinders],Origin,...
                'names',{'MPG' 'Cyl'},'cat',2)
t =
Decision tree for classification
1 if Cyl=4 then node 2 elseif Cyl in {6 8} then node 3 else USA
```

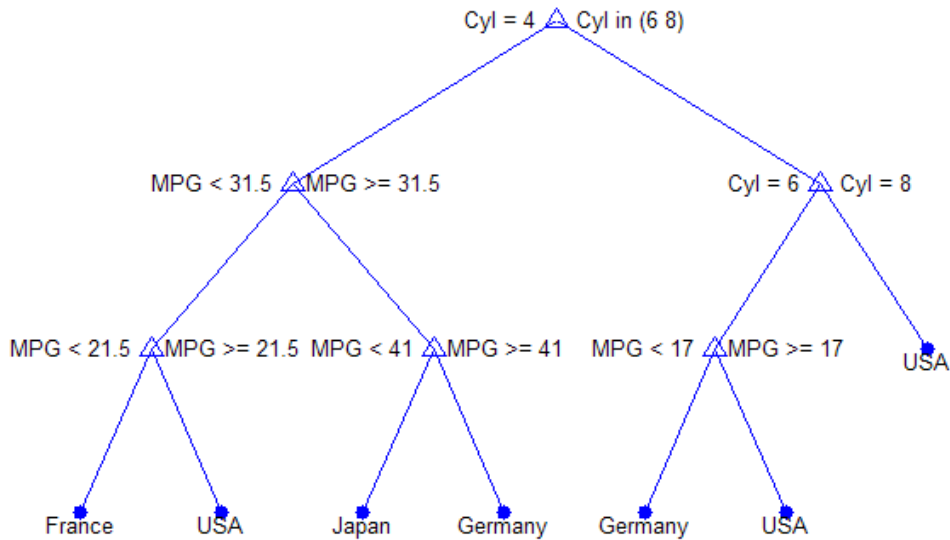
```

2 if MPG<31.5 then node 4 elseif MPG>=31.5 then node 5 else USA
3 if Cyl=6 then node 6 elseif Cyl=8 then node 7 else USA
4 if MPG<21.5 then node 8 elseif MPG>=21.5 then node 9 else USA
5 if MPG<41 then node 10 elseif MPG>=41 then node 11 else Japan
6 if MPG<17 then node 12 elseif MPG>=17 then node 13 else USA
7 class = USA
8 class = France
9 class = USA
10 class = Japan
11 class = Germany
12 class = Germany
13 class = USA

```

view(t)

Click to display:  Magnification:  Pruning level:



```

v = cutpoint(t)
v =
    NaN
    31.5000
    NaN
    21.5000
    41.0000

```

```
17.0000
NaN
NaN
NaN
NaN
NaN
NaN
NaN
```

## References

- [1] Breiman, L., J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Boca Raton, FL: CRC Press, 1984.

## See Also

`classregtree` | `cutcategories` | `cuttype` | `cutvar`

## cuttype

**Class:** classregtree

Cut types

---

**Note** `classregtree` will be removed in a future release. See `fitctree`, `fitrtree`, `ClassificationTree`, or `RegressionTree` instead.

---

## Syntax

```
c = cuttype(t)
c = cuttype(t,nodes)
```

## Description

`c = cuttype(t)` returns an  $n$ -element cell array `c` indicating the type of cut at each node in the tree `t`, where  $n$  is the number of nodes. For each node  $i$ , `c{i}` is:

- `'continuous'` — If the cut is defined in the form  $x < v$  for a variable  $x$  and cut point  $v$ .
- `'categorical'` — If the cut is defined by whether a variable  $x$  takes a value in a set of categories.
- `''` — If  $i$  is a leaf node.

`cutvar` returns the cut points for `'continuous'` cuts, and `cutcategories` returns the set of categories.

`c = cuttype(t,nodes)` takes a vector `nodes` of node numbers and returns the cut types for the specified nodes.

## Examples

Create a classification tree for car data:

```

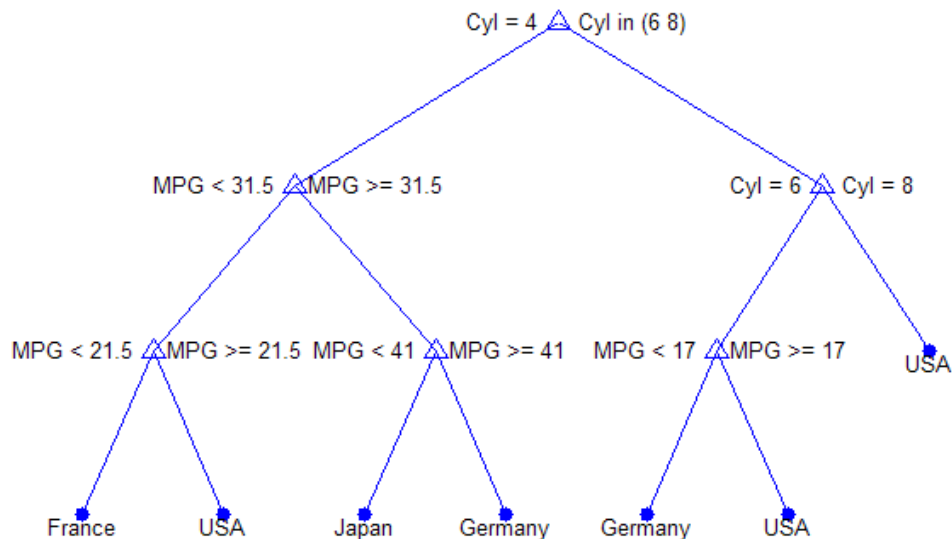
load carsmall

t = classregtree([MPG Cylinders],Origin,...
                'names',{'MPG' 'Cyl'},'cat',2)

t =
Decision tree for classification
1 if Cyl=4 then node 2 elseif Cyl in {6 8} then node 3 else USA
2 if MPG<31.5 then node 4 elseif MPG>=31.5 then node 5 else USA
3 if Cyl=6 then node 6 elseif Cyl=8 then node 7 else USA
4 if MPG<21.5 then node 8 elseif MPG>=21.5 then node 9 else USA
5 if MPG<41 then node 10 elseif MPG>=41 then node 11 else Japan
6 if MPG<17 then node 12 elseif MPG>=17 then node 13 else USA
7 class = USA
8 class = France
9 class = USA
10 class = Japan
11 class = Germany
12 class = Germany
13 class = USA
view(t)

```

Click to display:  Magnification:  Pruning level:



```

c = cuttype(t)
c =

```

```
'categorical'  
'continuous'  
'categorical'  
'continuous'  
'continuous'  
'continuous'  
''  
''  
''  
''  
''  
''  
''
```

## References

- [1] Breiman, L., J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Boca Raton, FL: CRC Press, 1984.

## See Also

`classregtree` | `cutcategories` | `cutvar` | `numnodes`

# cutvar

**Class:** classregtree

Cut variable names

---

**Note** classregtree will be removed in a future release. See fitctree, fitrtree, ClassificationTree, or RegressionTree instead.

---

## Syntax

```
v = cutvar(t)
v = cutvar(t,nodes)
[v,num] = cutvar(...)
```

## Description

`v = cutvar(t)` returns an  $n$ -element cell array `v` of the names of the variables used for branching in each node of the tree `t`, where  $n$  is the number of nodes. These variables are sometimes known as *cut variables*. For leaf nodes, `v` contains an empty character vector.

`v = cutvar(t,nodes)` takes a vector `nodes` of node numbers and returns the cut variables for the specified nodes.

`[v,num] = cutvar(...)` also returns a vector `num` containing the number of each variable.

## Examples

Create a classification tree for car data:

```
load carsmall

t = classregtree([MPG Cylinders],Origin,...
    'names',{'MPG' 'Cyl'},'cat',2)
t =
```

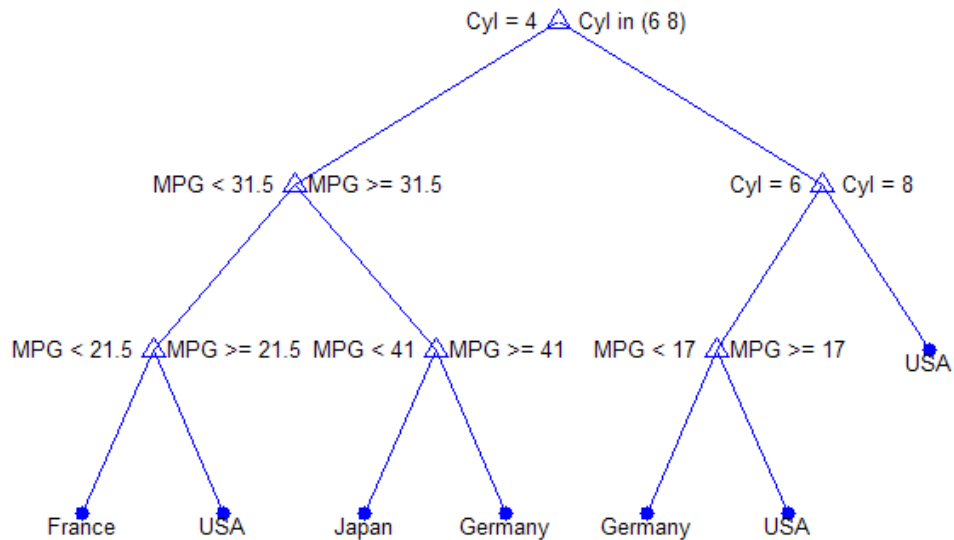
```

Decision tree for classification
1 if Cyl=4 then node 2 elseif Cyl in {6 8} then node 3 else USA
2 if MPG<31.5 then node 4 elseif MPG>=31.5 then node 5 else USA
3 if Cyl=6 then node 6 elseif Cyl=8 then node 7 else USA
4 if MPG<21.5 then node 8 elseif MPG>=21.5 then node 9 else USA
5 if MPG<41 then node 10 elseif MPG>=41 then node 11 else Japan
6 if MPG<17 then node 12 elseif MPG>=17 then node 13 else USA
7 class = USA
8 class = France
9 class = USA
10 class = Japan
11 class = Germany
12 class = Germany
13 class = USA

view(t)

```

Click to display:  Magnification:  Pruning level:



```

[v,num] = cutvar(t)
v =
    'Cyl'
    'MPG'
    'Cyl'

```



```
'MPG'  
'MPG'  
'MPG'  
' '  
' '  
' '  
' '  
' '  
' '  
' '  
' '  
num =  
 2  
 1  
 2  
 1  
 1  
 1  
 0  
 0  
 0  
 0  
 0  
 0  
 0  
 0
```

## References

- [1] Breiman, L., J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Boca Raton, FL: CRC Press, 1984.

## See Also

`children` | `classregtree` | `numnodes`

## cvloss

**Class:** ClassificationTree

Classification error by cross validation

### Syntax

```
E = cvloss(tree)
[E, SE] = cvloss(tree)
[E, SE, Nleaf] = cvloss(tree)
[E, SE, Nleaf, BestLevel] = cvloss(tree)
[ ___ ] = cvloss(tree, Name, Value)
```

### Description

`E = cvloss(tree)` returns the cross-validated classification error (loss) for `tree`, a classification tree. The `cvloss` method uses stratified partitioning to create cross-validated sets. That is, for each fold, each partition of the data has roughly the same class proportions as in the data used to train `tree`.

`[E, SE] = cvloss(tree)` returns the standard error of `E`.

`[E, SE, Nleaf] = cvloss(tree)` returns the number of leaves of `tree`.

`[E, SE, Nleaf, BestLevel] = cvloss(tree)` returns the optimal pruning level for `tree`.

`[ ___ ] = cvloss(tree, Name, Value)` cross validates with additional options specified by one or more `Name, Value` pair arguments, using any of the previous syntaxes. You can specify several name-value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

## Input Arguments

### **tree** — Trained classification tree

`ClassificationTree` model object

Trained classification tree, specified as a `ClassificationTree` model object produced by `fitctree`.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### **Subtrees** — Pruning level

0 (default) | vector of nonnegative integers | 'all'

Pruning level, specified as the comma-separated pair consisting of 'Subtrees' and a vector of nonnegative integers in ascending order or 'all'.

If you specify a vector, then all elements must be at least 0 and at most `max(tree.PruneList)`. 0 indicates the full, unpruned tree and `max(tree.PruneList)` indicates the completely pruned tree (i.e., just the root node).

If you specify 'all', then `ClassificationTree.cvloss` operates on all subtrees (i.e., the entire pruning sequence). This specification is equivalent to using `0:max(tree.PruneList)`.

`ClassificationTree.cvloss` prunes tree to each level indicated in `Subtrees`, and then estimates the corresponding output arguments. The size of `Subtrees` determines the size of some output arguments.

To invoke `Subtrees`, the properties `PruneList` and `PruneAlpha` of tree must be nonempty. In other words, grow tree by setting 'Prune', 'on', or by pruning tree using `prune`.

Example: 'Subtrees', 'all'

**TreeSize — Tree size**

'se' (default) | 'min'

Tree size, specified as the comma-separated pair consisting of 'TreeSize' and one of the following character vectors:

- 'se' — `cvloss` uses the smallest tree whose cost is within one standard error of the minimum cost.
- 'min' — `cvloss` uses the minimal cost tree.

Example: 'TreeSize', 'min'

**KFold — Number of cross-validation samples**

10 (default) | positive integer value greater than 1

Number of cross-validation samples, specified as the comma-separated pair consisting of `KFold` and a positive integer value greater than 1.

Example: 'KFold', 8

## Output Arguments

**E — Cross-validation classification error**

numeric vector | scalar value

Cross-validation classification error (loss), returned as a vector or scalar depending on the setting of the `Subtrees` name-value pair.

**SE — Standard error**

numeric vector | scalar value

Standard error of `E`, returned as a vector or scalar depending on the setting of the `Subtrees` name-value pair.

**Nleaf — Number of leaf nodes**

numeric vector | scalar value

Number of leaf nodes in `tree`, returned as a vector or scalar depending on the setting of the `Subtrees` name-value pair. Leaf nodes are terminal nodes, which give classifications, not splits.

**BestLevel** — Best pruning level

scalar value

Best pruning level, returned as a scalar value. By default, a scalar representing the largest pruning level that achieves a value of  $E$  within  $SE$  of the minimum error. If you set `TreeSize` to 'min', `BestLevel` is the smallest value in `Subtrees`.

## Examples

**Compute the Cross-Validation Error**

Compute the cross-validation error for a default classification tree.

Load the ionosphere data set.

```
load ionosphere
```

Grow a classification tree using the entire data set.

```
Mdl = fitctree(X,Y);
```

Compute the cross-validation error.

```
rng(1); % For reproducibility  
E = cvloss(Mdl)
```

```
E =
```

```
0.1111
```

$E$  is the 10-fold misclassification error.

**Find the Best Pruning Level Using Cross Validation**

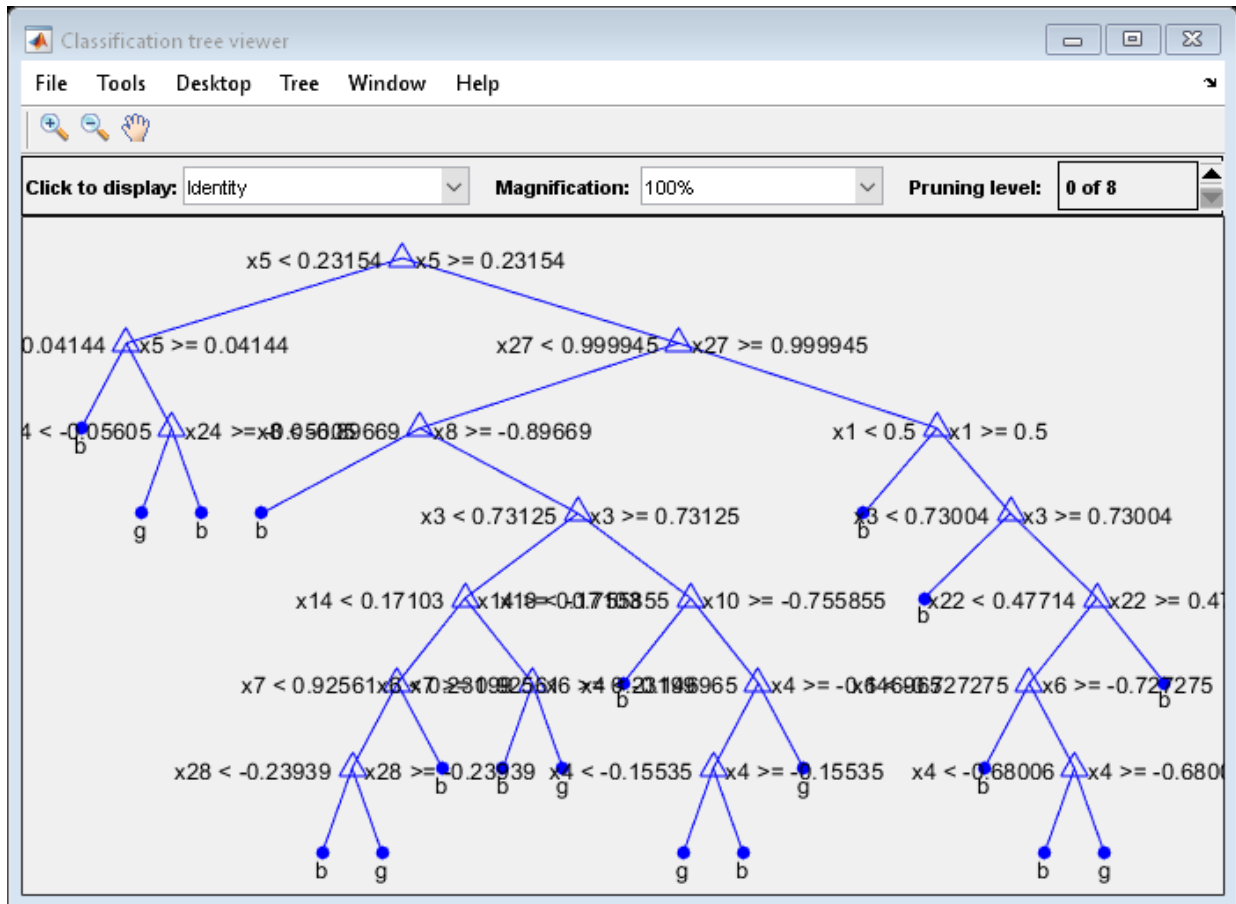
Apply  $k$ -fold cross validation to find the best level to prune a classification tree for all of its subtrees.

Load the ionosphere data set.

```
load ionosphere
```

Grow a classification tree using the entire data set. View the resulting tree.

```
Mdl = fitctree(X,Y);
view(Mdl, 'Mode', 'graph')
```



Compute the 5-fold cross-validation error for each subtree except for the highest pruning level. Specify to return the best pruning level over all subtrees.

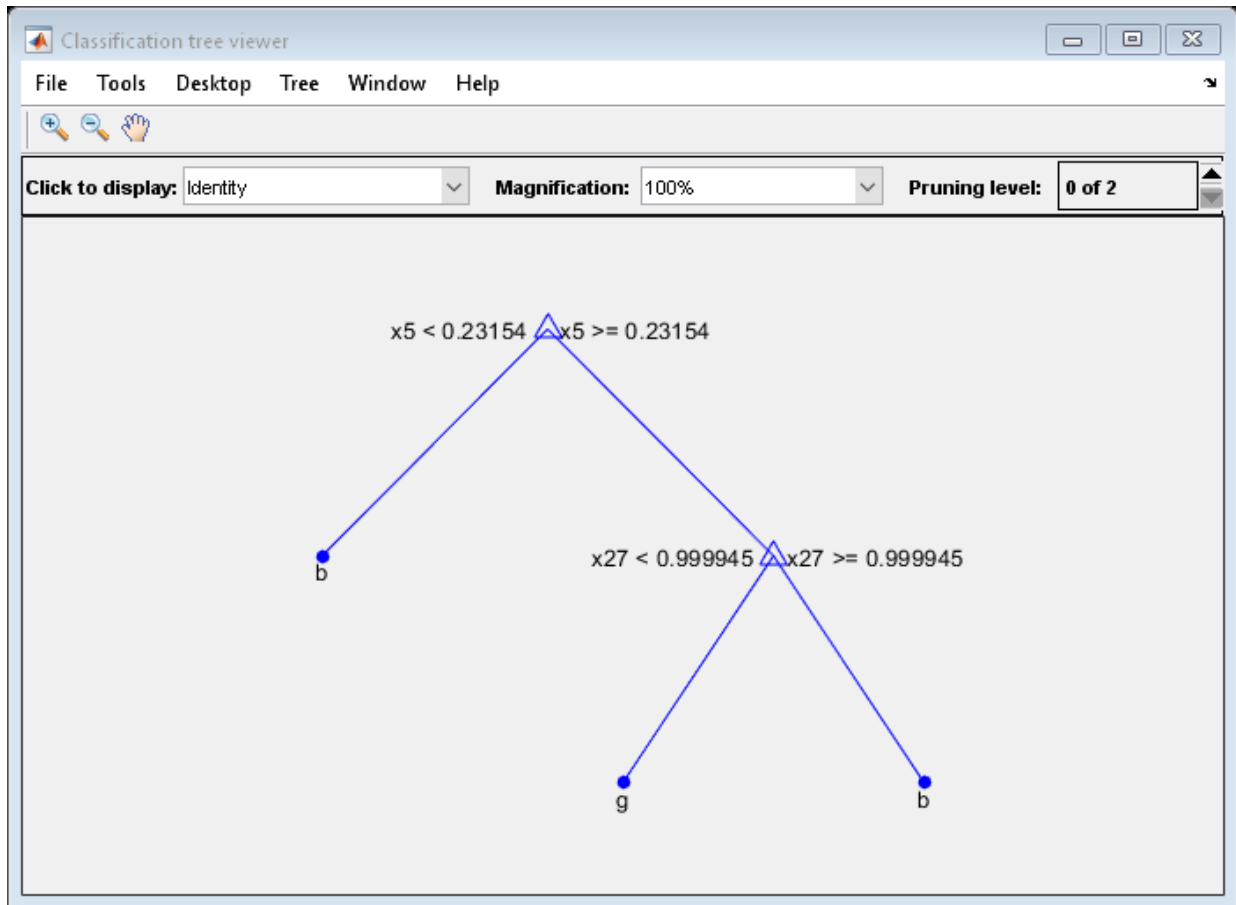
```
rng(1); % For reproducibility
m = max(Mdl.PruneList) - 1
[E,~,~,bestLevel] = cvloss(Mdl, 'SubTrees', 0:m, 'KFold', 5)
```

```
m =  
    7  
  
E =  
    0.1368  
    0.1339  
    0.1311  
    0.1339  
    0.1339  
    0.1254  
    0.0997  
    0.1738  
  
bestLevel =  
    6
```

Of the 7 pruning levels, the best pruning level is 6.

Prune the tree to the best level. View the resulting tree.

```
MdlPrune = prune(Mdl, 'Level', bestLevel);  
view(MdlPrune, 'Mode', 'graph')
```



## Alternatives

You can construct a cross-validated tree model with `crossval`, and call `kfoldLoss` instead of `cvloss`. If you are going to examine the cross-validated tree more than once, then the alternative can save time.

However, unlike `cvloss`, `kfoldLoss` does not return `SE`, `Nleaf`, or `BestLevel`. `kfoldLoss` also does not allow you to examine any error other than the classification error.



## See Also

`crossval` | `fitctree` | `kfoldLoss` | `loss`

## cvloss

**Class:** RegressionTree

Regression error by cross validation

## Syntax

```
E = cvloss(tree)
[E, SE] = cvloss(tree)
[E, SE, Nleaf] = cvloss(tree)
[E, SE, Nleaf, BestLevel] = cvloss(tree)
[E, ...] = cvloss(tree, Name, Value)
```

## Description

`E = cvloss(tree)` returns the cross-validated regression error (loss) for `tree`, a regression tree.

`[E, SE] = cvloss(tree)` returns the standard error of `E`.

`[E, SE, Nleaf] = cvloss(tree)` returns the number of leaves (terminal nodes) in `tree`.

`[E, SE, Nleaf, BestLevel] = cvloss(tree)` returns the optimal pruning level for `tree`.

`[E, ...] = cvloss(tree, Name, Value)` cross validates with additional options specified by one or more `Name, Value` pair arguments. You can specify several name-value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

## Input Arguments

**tree** — Trained regression tree

RegressionTree object

Trained regression tree, specified as a `RegressionTree` object constructed using `fitrtree`.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### **Subtrees** — Pruning level

0 (default) | vector of nonnegative integers | 'all'

Pruning level, specified as the comma-separated pair consisting of 'Subtrees' and a vector of nonnegative integers in ascending order or 'all'.

If you specify a vector, then all elements must be at least 0 and at most `max(tree.PruneList)`. 0 indicates the full, unpruned tree and `max(tree.PruneList)` indicates the completely pruned tree (i.e., just the root node).

If you specify 'all', then `RegressionTree.cvloss` operates on all subtrees (i.e., the entire pruning sequence). This specification is equivalent to using `0:max(tree.PruneList)`.

`RegressionTree.cvloss` prunes `tree` to each level indicated in `Subtrees`, and then estimates the corresponding output arguments. The size of `Subtrees` determines the size of some output arguments.

To invoke `Subtrees`, the properties `PruneList` and `PruneAlpha` of `tree` must be nonempty. In other words, grow `tree` by setting 'Prune', 'on', or by pruning `tree` using `prune`.

Example: 'Subtrees', 'all'

### **TreeSize** — Tree size

'se' (default) | 'min'

Tree size, specified as the comma-separated pair consisting of 'TreeSize' and one of the following:

- 'se' — `cvloss` uses the smallest tree whose cost is within one standard error of the minimum cost.
- 'min' — `cvloss` uses the minimal cost tree.

**KFold** — Number of folds

10 (default) | positive integer value greater than 1

Number of folds to use in a cross-validated tree, specified as the comma-separated pair consisting of 'KFold' and a positive integer value greater than 1.

Example: 'KFold', 8

## Output Arguments

**E** — Mean squared error

scalar value | numeric vector

Cross-validation mean squared error (loss), returned as a vector or scalar depending on the setting of the `Subtrees` name-value pair.

**SE** — Standard error

scalar value | numeric vector

Standard error of `E`, returned as vector or scalar depending on the setting of the `Subtrees` name-value pair.

**Nleaf** — Number of leaf nodes

vector of integer values

Number of leaf nodes in `tree`, returned as a vector or scalar depending on the setting of the `Subtrees` name-value pair. Leaf nodes are terminal nodes, which give responses, not splits.

**BestLevel** — Best pruning level

scalar value

Best pruning level as defined in the `TreeSize` name-value pair, returned as a scalar whose value depends on `TreeSize`:

- If `TreeSize` is 'se', then `BestLevel` is the largest pruning level that achieves a value of `E` within `SE` of the minimum error.
- If `TreeSize` is 'min', then `BestLevel` is the smallest value in `Subtrees`.

## Examples

### Compute the Cross-Validation Error

Compute the cross-validation error for a default regression tree.

Load the `carsmall` data set. Consider `Displacement`, `Horsepower`, and `Weight` as predictors of the response `MPG`.

```
load carsmall
X = [Displacement Horsepower Weight];
```

Grow a regression tree using the entire data set.

```
Mdl = fitrtree(X,MPG);
```

Compute the cross-validation error.

```
rng(1); % For reproducibility
E = cvloss(Mdl)
```

```
E =
```

```
25.7383
```

`E` is the 10-fold weighted, average MSE (weighted by number of test observations in the folds).

### Find the Best Pruning Level Using Cross Validation

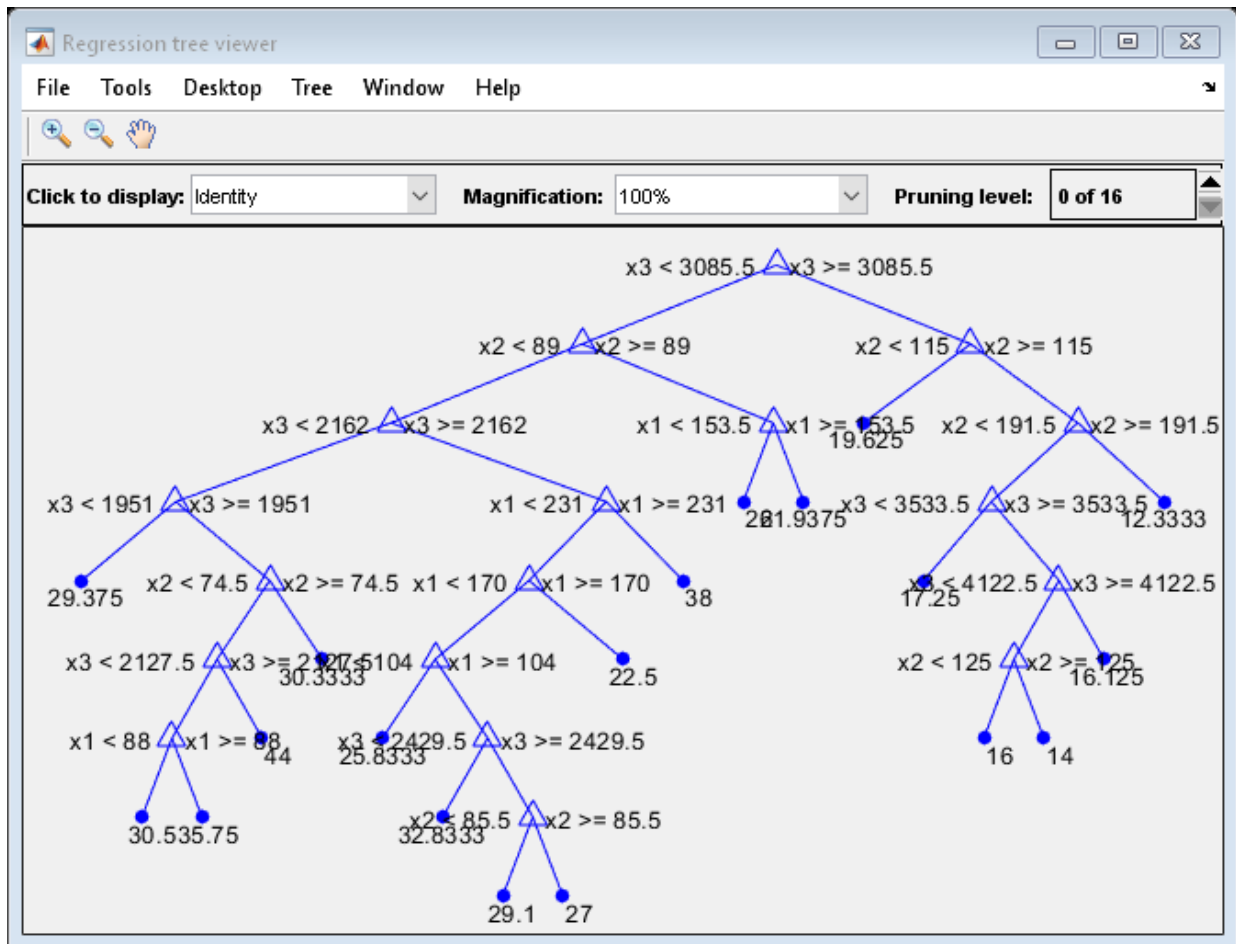
Apply  $k$ -fold cross validation to find the best level to prune a regression tree for all of its subtrees.

Load the carsmall data set. Consider Displacement, Horsepower, and Weight as predictors of the response MPG.

```
load carsmall
X = [Displacement Horsepower Weight];
```

Grow a regression tree using the entire data set. View the resulting tree.

```
Mdl = fitrtree(X,MPG);
view(Mdl,'Mode','graph')
```



Compute the 5-fold cross-validation error for each subtree except for the first two lowest and highest pruning level. Specify to return the best pruning level over all subtrees.

```
rng(1); % For reproducibility
m = max(Mdl.PruneList) - 1
[~,~,~,bestLevel] = cvloss(Mdl, 'SubTrees', 2:m, 'KFold', 5)
```

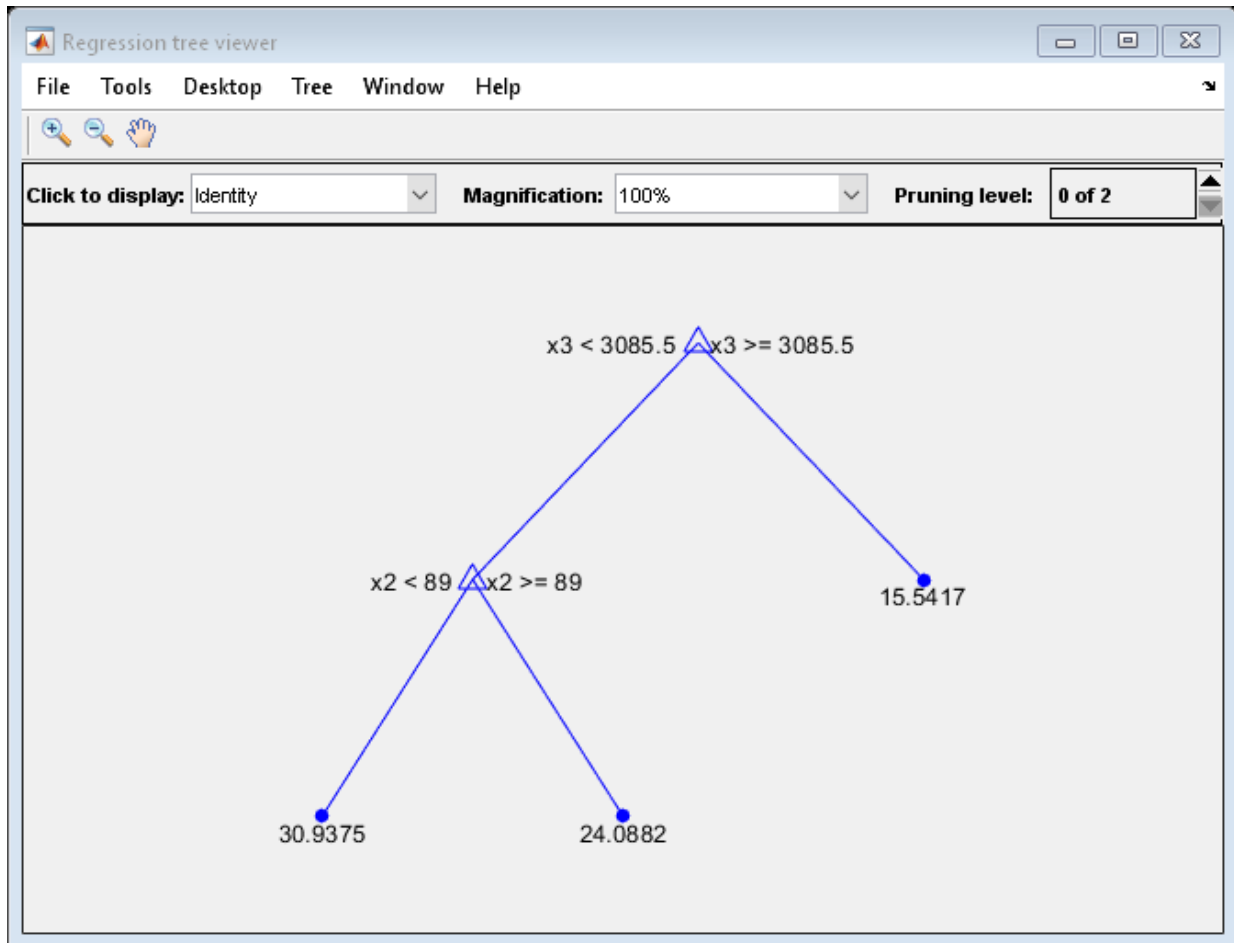
```
m =
    15
```

```
bestLevel =
    14
```

Of the 15 pruning levels, the best pruning level is 14.

Prune the tree to the best level. View the resulting tree.

```
MdlPrune = prune(Mdl, 'Level', bestLevel);
view(MdlPrune, 'Mode', 'graph')
```



## Alternatives

You can construct a cross-validated tree model with `crossval`, and call `kfoldLoss` instead of `cvloss`. If you are going to examine the cross-validated tree more than once, then the alternative can save time.

However, unlike `cvloss`, `kfoldLoss` does not return `SE`, `Nleaf`, or `BestLevel`.



## See Also

`crossval` | `fitrtree` | `kfoldLoss` | `loss`

## cvpartition class

Data partitions for cross validation

### Description

An object of the `cvpartition` class defines a random partition on a set of data of a specified size. Use this partition to define test and training sets for validating a statistical model using cross validation.

### Construction

`cvpartition`          Create cross validation partition for data

### Methods

<code>disp</code>	Display <code>cvpartition</code> object
<code>display</code>	Display <code>cvpartition</code> object
<code>repartition</code>	Repartition data for cross-validation
<code>test</code>	Test indices for cross-validation
<code>training</code>	Training indices for cross-validation

## Properties

NumObservations	Number of observations (including observations with missing group values)
NumTestSets	Number of test sets
TestSize	Size of each test set
TrainSize	Size of each training set
Type	Type of partition

## Copy Semantics

Value. To learn how this affects your use of the class, see [Comparing Handle and Value Classes \(MATLAB\)](#) in the MATLAB Object-Oriented Programming documentation.

## Examples

Use a 10-fold stratified cross validation to compute the misclassification error for `classify` on iris data.

```
load('fisheriris');
CVO = cvpartition(species, 'k', 10);
err = zeros(CVO.NumTestSets, 1);
for i = 1:CVO.NumTestSets
    trIdx = CVO.training(i);
    teIdx = CVO.test(i);
    ytest = classify(meas(teIdx, :), meas(trIdx, :), ...
        species(trIdx, :));
    err(i) = sum(~strcmp(ytest, species(teIdx)));
end
cvErr = sum(err)/sum(CVO.TestSize);
```

## See Also

`crossval`

## **Topics**

“Grouping Variables” on page 2-59

# cvpartition

**Class:** cvpartition

Create cross validation partition for data

## Syntax

```
c = cvpartition(n, 'Kfold', k)
c = cvpartition(group, 'Kfold', k)
c = cvpartition(n, 'HoldOut', p)
c = cvpartition(group, 'HoldOut', p)
c = cvpartition(n, 'LeaveOut')
c = cvpartition(n, 'resubstitution')
```

## Description

`c = cvpartition(n, 'Kfold', k)` constructs an object `c` of the `cvpartition` class on page 32-1292 defining a random partition for  $k$ -fold cross validation on  $n$  observations. The partition divides the observations into  $k$  disjoint subsamples (or folds), chosen randomly but with roughly equal size. The default value of  $k$  is 10.

`c = cvpartition(group, 'Kfold', k)` creates a random partition for a stratified  $k$ -fold cross validation. `group` is a numeric vector, categorical array, character array, or cell array of character vectors indicating the class of each observation. Each subsample has roughly equal size and roughly the same class proportions as in `group`. `cvpartition` treats NaNs or empty character vectors in `group` as missing values.

`c = cvpartition(n, 'HoldOut', p)` creates a random partition for holdout validation on  $n$  observations. This partition divides the observations into a training set and a test (or holdout) set. The parameter `p` must be a scalar. When  $0 < p < 1$ , `cvpartition` randomly selects approximately  $p \cdot n$  observations for the test set. When `p` is an integer, `cvpartition` randomly selects `p` observations for the test set. The default value of `p` is 1/10.

`c = cvpartition(group, 'HoldOut', p)` randomly partitions observations into a training set and a test set with stratification, using the class information in `group`; that is, both training and test sets have roughly the same class proportions as in `group`.

`c = cvpartition(n, 'LeaveOut')` creates a random partition for leave-one-out cross validation on `n` observations. Leave-one-out is a special case of 'Kfold', in which the number of folds equals the number of observations.

`c = cvpartition(n, 'resubstitution')` creates an object `c` that does not partition the data. Both the training set and the test set contain all of the original `n` observations.

## Examples

Use stratified 10-fold cross validation to compute misclassification rate:

```
load fisheriris;
y = species;
c = cvpartition(y, 'k', 10);

fun = @(xT, yT, xt, yt) (sum(~strcmp(yt, classify(xt, xT, yT))));

rate = sum(crossval(fun, meas, y, 'partition', c))...
      /sum(c.TestSize)
rate =
    0.0200
```

## Extended Capabilities

### Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function supports tall arrays for out-of-memory data with some limitations.

- For tall arrays only stratified-HoldOut partitions are supported.
- `c = cvpartition(group, 'HoldOut', p)` randomly partitions observations into a training set and a test set with stratification, using the class information in `group`. `P` is a scalar such that  $0 < P < 1$ .

- To obtain nonstratified partitions, set a uniform grouping variable from the data samples. For example, assuming  $X$  is a tall numeric array, you can use

```
groups = X(:,1).*0;  
C = cvpartition(groups,'HoldOut',P)
```

For more information, see “Tall Arrays” (MATLAB).

## See Also

[crossval](#) | [repartition](#)

## Topics

“Grouping Variables” on page 2-59

**Introduced in R2008a**

## cvshrink

**Class:** ClassificationDiscriminant

Cross-validate regularization of linear discriminant

### Syntax

```
err = cvshrink(obj)
[err,gamma] = cvshrink(obj)
[err,gamma,delta] = cvshrink(obj)
[err,gamma,delta,numpred] = cvshrink(obj)
[err,...] = cvshrink(obj,Name,Value)
```

### Description

`err = cvshrink(obj)` returns a vector of cross-validated classification error values for differing values of the regularization parameter Gamma.

`[err,gamma] = cvshrink(obj)` also returns the vector of Gamma values.

`[err,gamma,delta] = cvshrink(obj)` also returns the vector of Delta values.

`[err,gamma,delta,numpred] = cvshrink(obj)` returns the vector of number of nonzero predictors for each setting of the parameters Gamma and Delta.

`[err,...] = cvshrink(obj,Name,Value)` cross validates with additional options specified by one or more Name,Value pair arguments.

### Input Arguments

**obj**

Discriminant analysis classifier, produced using `fitcdiscr`.



## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### **delta**

- **Scalar delta** — `cvshrink` uses this value of `delta` with every value of `gamma` for regularization.
- **Row vector delta** — For each `i` and `j`, `cvshrink` uses `delta(j)` with `gamma(i)` for regularization.
- **Matrix delta** — The number of rows of `delta` must equal the number of elements in `gamma`. For each `i` and `j`, `cvshrink` uses `delta(i,j)` with `gamma(i)` for regularization.

**Default:** 0

### **gamma**

Vector of Gamma values for cross-validation.

**Default:** 0:0.1:1

### **NumDelta**

Number of Delta intervals for cross-validation. For every value of Gamma, `cvshrink` cross-validates the discriminant using `NumDelta + 1` values of Delta, uniformly spaced from zero to the maximal Delta at which all predictors are eliminated for this value of Gamma. If you set `delta`, `cvshrink` ignores `NumDelta`.

**Default:** 0

### **NumGamma**

Number of Gamma intervals for cross-validation. `cvshrink` cross-validates the discriminant using `NumGamma + 1` values of Gamma, uniformly spaced from `MinGamma` to 1. If you set `gamma`, `cvshrink` ignores `NumGamma`.

**Default:** 10

**verbose**

Verbosity level, an integer from 0 to 2. Higher values give more progress messages.

**Default:** 0

## Output Arguments

**err**

Numeric vector or matrix of errors. `err` is the misclassification error rate, meaning the average fraction of misclassified data over all folds.

- If `delta` is a scalar (default), `err(i)` is the misclassification error rate for `obj` regularized with `gamma(i)`.
- If `delta` is a vector, `err(i,j)` is the misclassification error rate for `obj` regularized with `gamma(i)` and `delta(j)`.
- If `delta` is a matrix, `err(i,j)` is the misclassification error rate for `obj` regularized with `gamma(i)` and `delta(i,j)`.

**gamma**

Vector of Gamma values used for regularization. See “Gamma and Delta” on page 32-1303.

**delta**

Vector or matrix of Delta values used for regularization. See “Gamma and Delta” on page 32-1303.

- If you give a scalar for the `delta` name-value pair, the output `delta` is a row vector the same size as `gamma`, with entries equal to the input scalar.
- If you give a row vector for the `delta` name-value pair, the output `delta` is a matrix with the same number of columns as the row vector, and with the number of rows equal to the number of elements of `gamma`. The output `delta(i,j)` is equal to the input `delta(j)`.
- If you give a matrix for the `delta` name-value pair, the output `delta` is the same as the input matrix. The number of rows of `delta` must equal the number of elements in `gamma`.

**numpred**

Numeric vector or matrix containing the number of predictors in the model at various regularizations. `numpred` has the same size as `err`.

- If `delta` is a scalar (default), `numpred(i)` is the number of predictors for `obj` regularized with `gamma(i)` and `delta`.
- If `delta` is a vector, `numpred(i,j)` is the number of predictors for `obj` regularized with `gamma(i)` and `delta(j)`.
- If `delta` is a matrix, `numpred(i,j)` is the number of predictors for `obj` regularized with `gamma(i)` and `delta(i,j)`.

## Examples

### Regularize Data with Many Predictors

Regularize a discriminant analysis classifier, and view the tradeoff between the number of predictors in the model and the classification accuracy.

Create a linear discriminant analysis classifier for the `ovariancancer` data. Set the `SaveMemory` and `FillCoeffs` options to keep the resulting model reasonably small.

```
load ovariancancer
obj = fitcdiscr(obs,grp,...
    'SaveMemory','on','FillCoeffs','off');
```

Use 10 levels of `Gamma` and 10 levels of `Delta` to search for good parameters. This search is time-consuming. Set `Verbose` to 1 to view the progress.

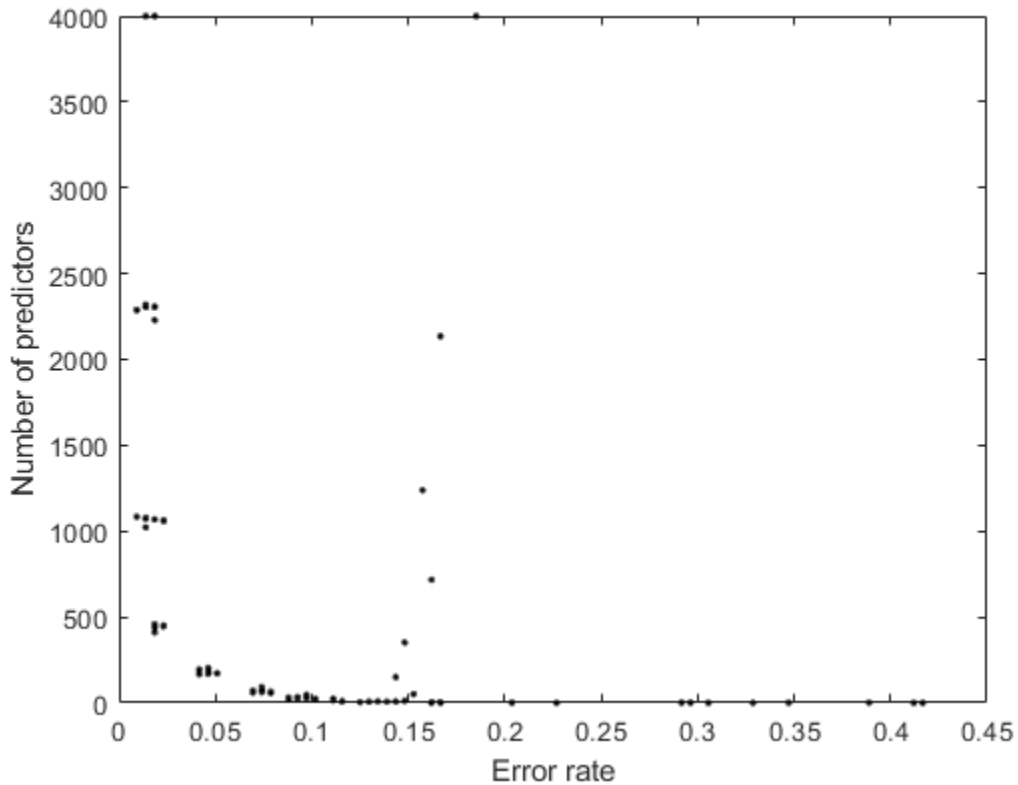
```
rng('default') % for reproducibility
[err,gamma,delta,numpred] = cvshrink(obj,...
    'NumGamma',9,'NumDelta',9,'Verbose',1);
```

```
Done building cross-validated model.
Processing Gamma step 1 out of 10.
Processing Gamma step 2 out of 10.
Processing Gamma step 3 out of 10.
Processing Gamma step 4 out of 10.
Processing Gamma step 5 out of 10.
```

```
Processing Gamma step 6 out of 10.  
Processing Gamma step 7 out of 10.  
Processing Gamma step 8 out of 10.  
Processing Gamma step 9 out of 10.  
Processing Gamma step 10 out of 10.
```

Plot the classification error rate against the number of predictors.

```
plot(err,numpred,'k.')  
xlabel('Error rate');  
ylabel('Number of predictors');
```



- “Regularize a Discriminant Analysis Classifier” on page 20-28

## Definitions

### Gamma and Delta

Regularization is the process of finding a small set of predictors that yield an effective predictive model. For linear discriminant analysis, there are two parameters,  $\gamma$  and  $\delta$ , that control regularization as follows. `cvshrink` helps you select appropriate values of the parameters.

Let  $\Sigma$  represent the covariance matrix of the data  $X$ , and let  $\hat{X}$  be the centered data (the data  $X$  minus the mean by class). Define

$$D = \text{diag}(\hat{X}^T * \hat{X}).$$

The regularized covariance matrix  $\tilde{\Sigma}$  is

$$\tilde{\Sigma} = (1 - \gamma)\Sigma + \gamma D.$$

Whenever  $\gamma \geq \text{MinGamma}$ ,  $\tilde{\Sigma}$  is nonsingular.

Let  $\mu_k$  be the mean vector for those elements of  $X$  in class  $k$ , and let  $\mu_0$  be the global mean vector (the mean of the rows of  $X$ ). Let  $C$  be the correlation matrix of the data  $X$ , and let

$\tilde{C}$  be the regularized correlation matrix:

$$\tilde{C} = (1 - \gamma)C + \gamma I,$$

where  $I$  is the identity matrix.

The linear term in the regularized discriminant analysis classifier for a data point  $x$  is

$$(x - \mu_0)^T \tilde{\Sigma}^{-1} (\mu_k - \mu_0) = \left[ (x - \mu_0)^T D^{-1/2} \right] \left[ \tilde{C}^{-1} D^{-1/2} (\mu_k - \mu_0) \right].$$

The parameter  $\delta$  enters into this equation as a threshold on the final term in square

brackets. Each component of the vector  $\left[ \tilde{C}^{-1} D^{-1/2} (\mu_k - \mu_0) \right]$  is set to zero if it is smaller in magnitude than the threshold  $\delta$ . Therefore, for class  $k$ , if component  $j$  is thresholded to zero, component  $j$  of  $x$  does not enter into the evaluation of the posterior probability.

The `DeltaPredictor` property is a vector related to this threshold. When  $\delta \geq \text{DeltaPredictor}(i)$ , all classes  $k$  have

$$\left| \tilde{C}^{-1} D^{-1/2} (\mu_k - \mu_0) \right| \leq \delta.$$

Therefore, when  $\delta \geq \text{DeltaPredictor}(i)$ , the regularized classifier does not use predictor  $i$ .

## Tips

- Examine the `err` and `numpred` outputs to see the tradeoff between cross-validated error and number of predictors. When you find a satisfactory point, set the corresponding `gamma` and `delta` properties in the model using dot notation. For example, if  $(i, j)$  is the location of the satisfactory point, set

```
obj.Gamma = gamma(i);  
obj.Delta = delta(i,j);
```

## See Also

`ClassificationDiscriminant` | `fitcdiscr`

## Topics

“Regularize a Discriminant Analysis Classifier” on page 20-28

“Discriminant Analysis Classification” on page 20-2

# cvshrink

**Class:** RegressionEnsemble

Cross validate shrinking (pruning) ensemble

## Syntax

```
vals = cvshrink(ens)
[vals,nlearn] = cvshrink(ens)
[vals,nlearn] = cvshrink(ens,Name,Value)
```

## Description

`vals = cvshrink(ens)` returns an L-by-T matrix with cross-validated values of the mean squared error. L is the number of lambda values in the `ens.Regularization` structure. T is the number of threshold values on weak learner weights. If `ens` does not have a `Regularization` property filled in by the `regularize` method, pass a lambda name-value pair.

`[vals,nlearn] = cvshrink(ens)` returns an L-by-T matrix of the mean number of learners in the cross-validated ensemble.

`[vals,nlearn] = cvshrink(ens,Name,Value)` cross validates with additional options specified by one or more `Name,Value` pair arguments. You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

## Input Arguments

### **ens**

A regression ensemble, created with `fitrensemble`.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

### **cvpartition**

A partition created with `cvpartition` to use in a cross-validated tree. You can only use one of these four options at a time: `'kfold'`, `'holdout'`, `'leaveout'`, or `'cvpartition'`.

### **holdout**

Holdout validation tests the specified fraction of the data, and uses the rest of the data for training. Specify a numeric scalar from 0 to 1. You can only use one of these four options at a time for creating a cross-validated tree: `'kfold'`, `'holdout'`, `'leaveout'`, or `'cvpartition'`.

### **kfold**

Number of folds to use in a cross-validated tree, a positive integer. If you do not supply a cross-validation method, `cvshrink` uses 10-fold cross validation. You can only use one of these four options at a time: `'kfold'`, `'holdout'`, `'leaveout'`, or `'cvpartition'`.

**Default:** 10

### **lambda**

Vector of nonnegative regularization parameter values for lasso. If empty, `cvshrink` does not perform cross validation.

**Default:** []

### **leaveout**

Use leave-one-out cross validation by setting to `'on'`. You can only use one of these four options at a time: `'kfold'`, `'holdout'`, `'leaveout'`, or `'cvpartition'`.



**threshold**

Numeric vector with lower cutoffs on weights for weak learners. `cvshrink` discards learners with weights below `threshold` in its cross-validation calculation.

**Default:** 0

## Output Arguments

**vals**

L-by-T matrix with cross-validated values of the mean squared error. L is the number of values of the regularization parameter `'lambda'`, and T is the number of `'threshold'` values on weak learner weights.

**nlearn**

L-by-T matrix with cross-validated values of the mean number of learners in the cross-validated ensemble. L is the number of values of the regularization parameter `'lambda'`, and T is the number of `'threshold'` values on weak learner weights.

## Examples

### Cross-Validate Regression Ensemble

Create a regression ensemble for predicting mileage from the `carsmall` data. Cross-validate the ensemble.

Load the `carsmall` data set and select displacement, horsepower, and vehicle weight as predictors.

```
load carsmall
X = [Displacement Horsepower Weight];
```

Train an ensemble of bagged regression trees.

```
ens = fitensemble(X,MPG, 'Method', 'Bag', 'Learners', 'Tree');
```

Specify values for `lambda` and `threshold`. Use these values to cross-validate the ensemble.

```
rng(10, 'twister') % For reproducibility
[vals, nlearn] = cvshrink(ens, 'lambda', [.01 .1 1], 'threshold', [0 .01 .1])

vals =

    17.7488    17.6537   123.2749
    17.7374    17.6304   123.4240
    17.6531    17.5540   101.2271

nlearn =

    13.1000    11.6000    3.4000
    13.2000    11.6000    3.4000
    13.1000    11.6000    3.7000
```

Clearly, setting a threshold of 0.1 leads to unacceptable errors, while a threshold of 0.01 gives similar errors to a threshold of 0. The mean number of learners with a threshold of 0.01 is 11.6, whereas the mean number is about 13.1 when the threshold is 0.

## See Also

`regularize` | `shrink`

# datasample

Randomly sample from data, with or without replacement

## Syntax

```
y = datasample(data, k)
y = datasample(data, k, dim)
y = datasample( ____, Name, Value)
y = datasample(s, ____)
[y, idx] = datasample( ____)
```

## Description

`y = datasample(data, k)` returns  $k$  observations sampled uniformly at random, with replacement, from the data in `data`.

`y = datasample(data, k, dim)` returns a sample taken along dimension `dim` of `data`.

`y = datasample( ____, Name, Value)` uses any of the input arguments in the previous syntaxes followed by one or more `Name, Value` pair arguments.

`y = datasample(s, ____)` uses the random number stream `s` to generate random numbers. The option `s` can precede any of the input arguments in the previous syntaxes.

`[y, idx] = datasample( ____)` also returns an index vector indicating which values `datasample` sampled from `data` using any of the input arguments in the previous syntaxes.

## Input Arguments

### **data**

Vector, matrix,  $N$ -dimensional array, table, or dataset array representing the data from which to sample. By default, `datasample` samples from the first nonsingleton dimension

of the `data` array. For example, if `data` is a matrix, then `datasample` samples from the rows. Change this behavior with the `dim` input argument.

**k**

Positive integer, the number of samples.

**dim**

Integer specifying the dimension to sample. For example, if `data` is a matrix and `dim` is 2, `y` contains a selection of columns in `data`. If `data` is a table or dataset array and `dim` is 2, `y` contains a selection of variables in `data`. Use `dim` to ensure sampling along a specific dimension regardless of whether `data` is a vector, matrix, or  $N$ -dimensional array.

**Default:** 1

**s**

Random number stream. Create `s` using `RandStream`. For example, `s = RandStream('mlfg6331_64')` creates a random number stream that uses the multiplicative lagged Fibonacci generator algorithm. For details, see “Creating and Controlling a Random Number Stream” (MATLAB).

**Default:** The global stream. The `rng` function provides a simple way to control the global stream. For example, `rng(seed)` seeds the random number generator using the nonnegative integer `seed`. For details, see “Managing the Global Stream” (MATLAB).

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

**Replace**

Sample with replacement if `Replace` is `true`, or without replacement if `Replace` is `false`. If `Replace` is `false`, then `k` must not be larger than the size of the dimension being sampled. For example, if `data = [1 3 Inf; 2 4 5]` and `y = datasample(data, k, 'Replace', false)`, then `k` cannot be larger than 2.

**Default:** `true`

### **Weights**

Vector with the same number of elements as the size of the dimension being sampled. The vector must have nonnegative elements and at least one positive value (NaN values are not allowed). The `datasample` function samples with probability proportional to the elements of `Weights`.

**Default:** `ones(datasize,1)`, where `datasize` is the size of the dimension being sampled

## Output Arguments

### **y**

- If `data` is a vector, `y` is a vector containing `k` elements selected from `data`.
- If `data` is a matrix, `y` is a matrix containing `k` rows selected from `data`. Or, if `dim = 2`, `y` is a matrix containing `k` columns selected from `data`.
- If `data` is an  $N$ -dimensional array, `datasample` samples along its first nonsingleton dimension. Or, if you specify a value for the `dim` name-value pair argument, `datasample` samples along the dimension `dim`.

If the input `data` contains missing observations that are represented as NaN values, `datasample` samples from the entire input, including the NaN values. For example, `y = datasample([NaN 6 14],2)` can return `y = NaN 14`.

When the sample is taken with replacement (default), `y` can contain repeated observations from `data`. Set the `Replace` name-value pair argument to `false` to sample without replacement.

### **idx**

Vector of indices indicating which elements `datasample` chose from `data` to create `y`. For example:

- If `data` is a vector, then `y = data(idx)`.
- If `data` is a matrix and `dim = 1`, then `y = data(idx,:)`.

- If data is a matrix and `dim = 2`, then `y = data(:,idx)`.

## Examples

Draw five unique values from the integers 1:10.

```
s = RandStream('mlfg6331_64'); % For reproducibility
y = datasample(s,1:10,5,'Replace',false)
```

```
y =
```

```
     9     8     3     6     2
```

Generate a random sequence of the characters ACGT, with replacement, according to specified probabilities.

```
s = RandStream('mlfg6331_64'); % For reproducibility
seq = datasample(s,'ACGT',48,'Weights',[0.15 0.35 0.35 0.15])
```

```
seq =
```

```
 'GGCGGCGCAAGGCGCCGACCTGGCTGCACCCGTTCCTGCTACTCG'
```

Select a random subset of columns from a data matrix.

```
rng(10,'twister') % For reproducibility
X = randn(10,1000);
s = RandStream('mlfg6331_64'); % For reproducibility
Y = datasample(s,X,5,2,'Replace',false)
```

```
Y =
```

```
    0.4317   -0.3327    0.9112   -2.3244    0.9559
    0.6977   -0.7422    0.4578   -1.3745   -0.8634
   -0.8543   -0.3105    0.9836   -0.6434   -0.4457
    0.1686    0.6609   -0.0553   -0.1202   -1.3699
   -1.7649   -1.1607   -0.3513   -1.5533    0.0597
   -0.3821    0.5696   -1.6264   -0.2104   -1.5486
   -1.6844    0.7148   -0.6876   -0.4447   -1.4615
   -0.4170    1.3696    1.1874   -0.9901    0.5875
   -0.2410    1.4703   -2.5003   -1.1321   -1.8451
    0.6212    1.4118   -0.4518    0.8697    0.8093
```

Resample observations from a dataset array to create a bootstrap replicate dataset. See “Bootstrap Resampling” on page 3-17 for more information about bootstrapping.

```
load hospital
y = datasample(hospital, size(hospital,1));
```

Use the second output to sample “in parallel” from two data vectors.

```
x1 = randn(100,1);
x2 = randn(100,1);
[y1,idx] = datasample(x1,10);
y2 = x2(idx);
```

## Tips

- To sample random integers with replacement from a range, use `randi`.
- To sample random integers without replacement, use `randperm` or `datasample`.
- To randomly sample from data, with or without replacement, use `datasample`.

## Algorithms

`datasample` uses `randperm`, `rand`, or `randi` to generate random values. Therefore, `datasample` changes the state of the MATLAB global random number generator. Control the random number generator using `rng`.

For selecting weighted samples without replacement, `datasample` uses the algorithm of Wong and Easton [1].

## Alternatives

You can use `randi` or `randperm` to generate indices for random sampling with or without replacement, respectively. However, `datasample` can be more convenient because it samples directly from your data. `datasample` also allows weighted sampling.

## References

- [1] Wong, C. K. and M. C. Easton. *An Efficient Method for Weighted Sampling Without Replacement*. SIAM Journal of Computing 9(1), pp. 111–113, 1980.

## Extended Capabilities

### Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function supports tall arrays for out-of-memory data with some limitations.

- `datasample` is useful as a precursor to plotting a random subset of a very large data set. Sampling a large data set preserves trends in the data without requiring that you plot all the data points.
- `datasample` supports sampling only along the first dimension of the data.
- For tall arrays, `datasample` does not support sampling with replacement. You must specify `'Replace', false`, for example, `datasample(data, k, 'Replace', false)`.
- The value of `'Weights'` must be a numeric tall array of the same height as `data`.
- For the syntax `[Y, idx] = datasample(___)`, the output `idx` is a tall logical vector of the same height as `data`. The vector indicates whether each data point is included in the sample.
- If you specify a random number stream, then the underlying generator must support multiple streams and substreams. If you do not specify a random number stream, then `datasample` uses the stream controlled by `tallrng`.

For more information, see “Tall Arrays” (MATLAB).

### See Also

`RandStream` | `rand` | `randi` | `randperm` | `rng` | `tallrng`

Introduced in R2011b



## dataset class

Arrays for statistical data

---

**Note** The `dataset` data type might be removed in a future release. To work with heterogeneous data, use the MATLAB® `table` data type instead. See MATLAB `table` documentation for more information.

---

### Description

Dataset arrays are used to collect heterogeneous data and metadata including variable and observation names into a single container variable. Dataset arrays are suitable for storing column-oriented or tabular data that are often stored as columns in a text file or in a spreadsheet, and can accommodate variables of different types, sizes, units, etc.

Dataset arrays can contain different kinds of variables, including numeric, logical, character, categorical, and cell. However, a dataset array is a different class than the variables that it contains. For example, even a dataset array that contains only variables that are double arrays cannot be operated on as if it were itself a double array. However, using dot subscripting, you can operate on variable in a dataset array as if it were a workspace variable.

You can subscript dataset arrays using parentheses much like ordinary numeric arrays, but in addition to numeric and logical indices, you can use variable and observation names as indices.

### Construction

Use the `dataset` constructor to create a dataset array from variables in the MATLAB workspace. You can also create a dataset array by reading data from a text or spreadsheet file. You can access each variable in a dataset array much like fields in a structure, using dot subscripting. See the following section for a list of operations available for dataset arrays.

dataset

Construct dataset array

## Methods

cat	Concatenate dataset arrays
cellstr	Create cell array of character vectors from dataset array
dataset2cell	Convert dataset array to cell array
dataset2struct	Convert dataset array to structure
datasetfun	Apply function to dataset array variables
disp	Display dataset array
display	Display dataset array
double	Convert dataset variables to double array
end	Last index in indexing expression for dataset array
export	Write dataset array to file
get	Access dataset array properties
horzcat	Horizontal concatenation for dataset arrays
intersect	Set intersection for dataset array observations
isempty	True for empty dataset array
ismember	Dataset array elements that are members of set
ismissing	Find dataset array elements with missing values
join	Merge observations
length	Length of dataset array
ndims	Number of dimensions of dataset array
numel	Number of elements in dataset array
replacedata	Replace dataset variables
replaceWithMissing	Insert missing data indicators into a dataset array
set	Set and display properties
setdiff	Set difference for dataset array observations
setxor	Set exclusive or for dataset array observations
single	Convert dataset variables to single array
size	Size of dataset array
sortrows	Sort rows of dataset array
stack	Stack data from multiple variables into single variable
subsasgn	Subscripted assignment to dataset array
subsref	Subscripted reference for dataset array
summary	Print summary of dataset array
union	Set union for dataset array observations

## Properties

A dataset array `D` has properties that store metadata (information about your data). Access or assign to a property using `P = D.Properties.PropName` or `D.Properties.PropName = P`, where `PropName` is one of the following:

Description	Character vector describing data set
DimNames	Two-element cell array of character vectors giving names of dimensions of data set
ObsNames	Cell array of nonempty, distinct character vectors giving names of observations in data set
Units	Units of variables in data set
UserData	Variable containing additional information associated with data set
VarDescription	Cell array of character vectors giving descriptions of variables in data set
VarNames	Cell array giving names of variables in data set

## Copy Semantics

Value. To learn how this affects your use of the class, see [Comparing Handle and Value Classes \(MATLAB\)](#) in the MATLAB Object-Oriented Programming documentation.

## Examples

Load a dataset array from a `.mat` file and create some simple subsets:

```
load hospital
h1 = hospital(1:10, :)
h2 = hospital(:, {'LastName' 'Age' 'Sex' 'Smoker'})

% Access and modify metadata
hospital.Properties.Description
hospital.Properties.VarNames{4} = 'Wgt'

% Create a new dataset variable from an existing one
hospital.AtRisk = hospital.Smoker | (hospital.Age > 40)
```

```
% Use individual variables to explore the data
boxplot(hospital.Age,hospital.Sex)
h3 = hospital(hospital.Age<30,...
    {'LastName' 'Age' 'Sex' 'Smoker'})

% Sort the observations based on two variables
h4 = sortrows(hospital,{'Sex','Age'})
```

## See Also

[tdfread](#) | [textscan](#) | [xlsread](#)

## Topics

“Dataset Arrays” on page 2-142

## dataset

**Class:** dataset

Construct dataset array

---

**Note** The `dataset` data type might be removed in a future release. To work with heterogeneous data, use the MATLAB® `table` data type instead. See MATLAB `table` documentation for more information.

---

## Syntax

```
A = dataset(varspec, 'ParamName', Value)
A = dataset('File', filename, 'ParamName', Value)
A = dataset('XLSFile', filename, 'ParamName', Value)
A = dataset('XPTFile', xptfilename, 'ParamName', Value)
```

## Description

`A = dataset(varspec, 'ParamName', Value)` creates dataset array `A` using the workspace variable input method `varspec` and one or more optional name/value pairs (see Parameter Name/Value Pairs).

The input method `varspec` can be one or more of the following:

- `VAR` — a workspace variable. `dataset` uses the workspace name for the variable name in `A`. To include multiple variables, specify `VAR_1,VAR_2,...,VAR_N`. Variables can be arrays of any size, but all variables must have the same number of rows. `VAR` can also be an expression. In this case, `dataset` creates a default name automatically.
- `{VAR,name}` — a workspace variable, `VAR` and a variable name, `name`. `dataset` uses `name` as the variable name. To include multiple variables and names, specify `{VAR_1,name_1}, {VAR_2,name_2},..., {VAR_N,name_N}`.

- $\{VAR, name_1, \dots, name_m\}$  — an  $m$ -columned workspace variable,  $VAR$ . `dataset` uses the names  $name_1, \dots, name_m$  as variable names. You must include a name for every column in  $VAR$ . Each column becomes a separate variable in  $A$ .

You can combine these input methods to include as many variables and names as needed. Names must be valid, unique MATLAB identifiers. For example input combinations, see Examples. For optional name/value pairs see Inputs.

To convert numeric arrays, cell arrays, structure arrays, or tables to dataset arrays, you can also use (respectively):

- `mat2dataset`
- `cell2dataset`
- `struct2dataset`
- `table2dataset`

---

**Note** Dataset arrays may contain built-in types or array objects as variables. Array objects must implement each of the following:

- Standard MATLAB parenthesis indexing of the form `var(i, ...)`, where  $i$  is a numeric or logical vector corresponding to rows of the variable
- A `size` method with a `dim` argument
- A `vertcat` method

---

`A = dataset('File', filename, 'ParamName', Value)` creates dataset array  $A$  from column-oriented data in the text file specified by `filename`. Variables in  $A$  are of type `double` if data in the corresponding column of the file, following the column header, are entirely numeric; otherwise the variables in  $A$  are cell arrays of character vectors. `dataset` converts empty fields to either `NaN` (for a numeric variable) or the empty character vector (for a character-valued variable). `dataset` ignores insignificant white space in the file. You cannot specify both a file and workspace variables as input. See Name/Value Pairs for more information.

`A = dataset('XLSFile', filename, 'ParamName', Value)` creates dataset array  $A$  from column-oriented data in the Excel spreadsheet specified by `filename`. Variables in  $A$  are of type `double` if data in the corresponding column of the spreadsheet, following

the column header, are entirely numeric; otherwise the variables in *A* are cell arrays of character vectors. See Name/Value Pairs for more information.

`A = dataset('XPTFile', xptfilename, 'ParamName', Value)` creates a dataset array from a SAS® XPORT format file. Variable names from the XPORT format file are preserved. Numeric data types in the XPORT format file are preserved but all other data types are converted to cell arrays of character vectors. The XPORT format allows for 28 missing data types. `dataset` represents these in the file by an upper case letter, ' .' or ' \_ '. `dataset` converts all missing data to NaN values in *A*. See Name/Value Pairs for more information.

## Parameter Name/Value Pairs

Specify one or more of the following name/value pairs when constructing a dataset:

### **VarNames**

A cell array `{name_1, ..., name_m}` naming the *m* variables in *A* with the specified variable names. Names must be valid, unique MATLAB identifiers. The number of names must equal the number of variables in *A*. You cannot use the `VarNames` parameter if you provide names for individual variables using `{VAR, name}` pairs. To specify `VarNames` when using a file as input, set `ReadVarNames` to `false`.

### **ObsNames**

A cell array `{name_1, ..., name_n}` naming the *n* observations in *A* with the specified observation names. The names need not be valid MATLAB identifiers, but must be unique. The number of names must equal the number of observations (rows) in *A*. To specify `ObsNames` when using a file as input, set `ReadObsNames` to `false`.

### **Name/value pairs available when using text files as inputs:**

#### **Delimiter**

A character vector indicating the character separating columns in the file. Values are

- `'\t'` (tab, the default when no `format` is specified)
- `' '` (space, the default when a `format` is specified)
- `','` (comma)



- ';' (semicolon)
- '|' (bar)

**Format**

A format character vector, as accepted by `textscan`. `dataset` reads the file using `textscan`, and creates variables in `A` according to the conversion specifiers in the format character vector. You may also provide any name/value pairs accepted by `textscan`. Using the `Format` parameter is much faster for large files. If `ReadObsNames` is `true`, then `format` should include a format specifier for the first column of the file.

**HeaderLines**

Numeric value indicating the number of lines to skip at the beginning of a file.

**Default:** 0

**TreatAsEmpty**

Specifies characters to treat as the empty character vector in a numeric column. Values may be a character array or a cell array of character vectors. The parameter applies only to numeric columns in the file; `dataset` does not accept numeric literals such as '-99'.

**Name/value pairs available when using text files or Excel spreadsheets as inputs:****ReadVarNames**

A logical value indicating whether (`true`) or not (`false`) to read variable names from the first row of the file. The default is `true`. If `ReadVarNames` is `true`, variable names in the column headers of the file or range (if using an Excel spreadsheet) cannot be empty.

**ReadObsNames**

A logical value indicating whether (`true`) or not (`false`) to read observation names from the first column of the file or range (if using an Excel spreadsheet). The default is `false`. If `ReadObsNames` and `ReadVarNames` are both `true`, `dataset` saves the header of the first column in the file or range as the name of the first dimension in `A.Properties.DimNames`.

When reading from an XPT format file, the `ReadObsNames` parameter name/value pair determines whether or not to try to use the first variable in the file as observation

names. Specify as a logical value (default `false`). If the contents of the first variable are not valid observation names then `dataset` reads the variable into a variable of the dataset array and does not set the observation names.

### **Name/value pairs available when using Excel spreadsheets as input:**

#### **Sheet**

A positive scalar value of type `double` indicating the sheet number, or a quoted sheet name.

#### **Range**

A character vector of the form `'C1:C2'` where `C1` and `C2` are the names of cells at opposing corners of a rectangular region to be read, as for `xlsread`. By default, the rectangular region extends to the right-most column containing data. If the spreadsheet contains empty columns between columns of data, or if the spreadsheet contains figures or other non-tabular information, specify a range that contains only data.

## **Examples**

Create a dataset array from workspace variables, including observation names:

```
load cereal
cereal = dataset(Calories, Protein, Fat, Sodium, Fiber, Carbo, ...
    Sugars, 'ObsNames', Name)
cereal.Properties.VarDescription = Variables(4:10, 2);
```

Create a dataset array from a single, multi-columned workspace variable, designating variable names for each column:

```
load cities
categories = cellstr(categories);
cities = dataset({ratings, categories{:}}, ...
    'ObsNames', cellstr(names))
```

Load data from a text or spreadsheet file

```
patients = dataset('File', 'hospital.dat', ...
    'Delimiter', ',', 'ReadObsNames', true)
patients2 = dataset('XLSFile', 'hospital.xls', ...
    'ReadObsNames', true)
```

- 1 Load patient data from the CSV file `hospital.dat` and store the information in a dataset array with observation names given by the first column in the data (patient identification):

```
patients = dataset('file','hospital.dat', ...
                  'format','%s%s%s%f%f%f%f%f%f%f%f', ...
                  'Delimiter',' ',' ','ReadObsNames',true);
```

You can also load the data without specifying a format. `dataset` will automatically create dataset variables that are either double arrays or cell arrays of character vectors, depending on the contents of the file:

```
patients = dataset('file','hospital.dat',...
                  'delimiter',' ','...', ...
                  'ReadObsNames',true);
```

- 2 Make the {0,1}-valued variable `smoke` nominal, and change the labels to 'No' and 'Yes':

```
patients.smoke = nominal(patients.smoke,{'No','Yes'});
```

- 3 Add new levels to `smoke` as placeholders for more detailed histories of smokers:

```
patients.smoke = addlevels(patients.smoke,...
                          {'0-5 Years','5-10 Years','LongTerm'});
```

- 4 Assuming the nonsmokers have never smoked, relabel the 'No' level:

```
patients.smoke = setlabels(patients.smoke,'Never','No');
```

- 5 Drop the undifferentiated 'Yes' level from `smoke`:

```
patients.smoke = droplevels(patients.smoke,'Yes');
```

Warning: OLDLEVELS contains categorical levels that were present in A, caused some array elements to have undefined levels.

Note that smokers now have an undefined level.

- 6 Set each smoker to one of the new levels, by observation name:

```
patients.smoke('YPL-320') = '5-10 Years';
```

## See Also

`cell2dataset` | `mat2dataset` | `struct2dataset` | `tdfread` | `textscan` | `xlsread`

## **Topics**

“Create a Dataset Array from Workspace Variables” on page 2-70

“Create a Dataset Array from a File” on page 2-77

“Dataset Arrays in the Variables Editor” on page 2-128

“Dataset Arrays” on page 2-142

**Introduced in R2007a**

# dataset2cell

**Class:** dataset

Convert dataset array to cell array

---

**Note** The `dataset` data type might be removed in a future release. To work with heterogeneous data, use the MATLAB® `table` data type instead. See MATLAB `table` documentation for more information.

---

## Syntax

```
C = dataset2cell(D)
```

## Description

`C = dataset2cell(D)` converts the dataset array `D` to a cell array `C`. Each variable of `D` becomes a column in `C`. If `D` is an `M`-by-`N` array, then `C` is  $(M+1)$ -by-`N`, with the variable names of `D` in the first row. If `D` contains observation names, then `C` is  $(M+1)$ -by- $(N+1)$ , with the observation names in the first column.

## See Also

`cell2dataset` | `dataset` | `dataset.export`

## Topics

“Dataset Arrays” on page 2-142

## dataset2struct

**Class:** dataset

Convert dataset array to structure

---

**Note** The `dataset` data type might be removed in a future release. To work with heterogeneous data, use the MATLAB® `table` data type instead. See MATLAB `table` documentation for more information.

---

### Syntax

```
S = dataset2struct(D)
S = dataset2struct(D, 'AsScalar', true)
```

### Description

`S = dataset2struct(D)` converts a dataset array to a structure array. Each variable of `D` becomes a field in `S`. If `D` is an  $M$ -by- $N$  dataset array, then `S` is  $M$ -by-1 and has  $N$  fields. If `D` contains observation names, then `S` contains those names in the additional field `ObsNames`.

`S = dataset2struct(D, 'AsScalar', true)` converts a dataset array to a scalar structure. Each variable of `D` becomes a field in `S`. If `D` is an  $M$ -by- $N$  dataset array, then `S` has  $N$  fields, each of which as  $M$  rows. If `D` contains observation names, then `S` contains those names in the additional field `ObsNames`.

### Input Arguments

`D`

$M$ -by- $N$  dataset array.

## Output Arguments

**S**

$M$ -by-1 structure array, with  $N$  fields. If the input dataset array contains observation names, then  $S$  has an additional field `ObsNames`.

If you specify `'AsScalar', true`, then  $S$  is a scalar structure, with  $N$  fields, each with  $M$  rows.

## Examples

### Convert Dataset Array to Structure Array

Load sample dataset array.

```
load('hospital')
```

Create a dataset array,  $D$ , that has only a subset of the observations and variables.

```
D = hospital(1:8, {'LastName', 'Sex', 'Age'});  
size(D)
```

```
ans =
```

```
      8      3
```

The dataset array  $D$  has 8 observations and 3 variables.

Convert  $D$  to a structure array.

```
S = dataset2struct(D)
```

```
S =
```

```
8x1 struct array with fields:
```

```
  ObsNames
```

```
    LastName
    Sex
    Age
```

The structure is  $8 \times 1$ , corresponding to the 8 observations in the dataset array. `S` also has the field `ObsNames`, since `D` had observation names.

Display the field data for the first element of `S`.

```
S(1)

ans =

    struct with fields:

        ObsNames: 'YPL-320'
        LastName: 'SMITH'
           Sex: Male
           Age: 38
```

This information corresponds to the first observation (row) of the dataset array.

### Convert Dataset Array to Scalar Structure

Load sample dataset array.

```
load('hospital')
```

Create a dataset array, `D`, that has only a subset of the observations and variables.

```
D = hospital(1:8, {'LastName', 'Sex', 'Age'});
size(D)
```

```
ans =

     8     3
```

The dataset array `D` has 8 observations and 3 variables.



Convert `D` to a scalar structure array.

```
S = dataset2struct(D, 'AsScalar', true)
```

```
S =
```

```
struct with fields:

  ObsNames: {8x1 cell}
  LastName: {8x1 cell}
           Sex: [8x1 nominal]
           Age: [8x1 double]
```

The data in the fields of the scalar structure is 8x1, corresponding to the 8 observations in the dataset array. `S` also has the field `ObsNames`, since `D` had observation names.

Display the data for the field `LastName`.

```
S.LastName
```

```
ans =
```

```
8x1 cell array

{'SMITH' }
{'JOHNSON' }
{'WILLIAMS' }
{'JONES' }
{'BROWN' }
{'DAVIS' }
{'MILLER' }
{'WILSON' }
```

The structure field `LastName` contains all of the data that was in the original dataset array variable, `LastName`.

## See Also

[dataset](#) | [dataset2cell](#) | [struct2dataset](#)

## **Topics**

“Dataset Arrays” on page 2-142

# dataset2table

Convert dataset array to table

## Syntax

```
t = dataset2table(ds)
```

## Description

`t = dataset2table(ds)` converts a dataset array to a table.

## Examples

### Convert Dataset Array to Table

Load the sample data, which contains nutritional information for 77 cereals.

```
load cereal;
```

Create a dataset array containing the calorie, protein, fat, and name data for the first five cereals. Label the variables.

```
Calories = Calories(1:5);  
Protein = Protein(1:5);  
Fat = Fat(1:5);  
Name = Name(1:5);  
  
cereal = dataset(Calories, Protein, Fat, 'ObsNames', Name)  
cereal.Properties.VarDescription = Variables(4:6, 2);  
  
cereal =
```

	Calories	Protein	Fat
100% Bran	70	4	1

```
100% Natural Bran      120      3      5
All-Bran               70       4      1
All-Bran with Extra Fiber  50       4      0
Almond Delight        110      2      2
```

Convert the dataset array to a table.

```
t = dataset2table(cereal)
```

```
t =
```

```
5x3 table
```

	Calories	Protein	Fat
	-----	-----	-----
100% Bran	70	4	1
100% Natural Bran	120	3	5
All-Bran	70	4	1
All-Bran with Extra Fiber	50	4	0
Almond Delight	110	2	2

## Input Arguments

### **ds** — Input dataset array

dataset array

Input dataset array to convert to a table, specified as a dataset array. Each variable in `ds` becomes a variable in the output table `t`.

## Output Arguments

### **t** — Output table

table

Output table, returned as a table. The table can store metadata such as descriptions, variable units, variable names, and row names. For more information, see “Tables” (MATLAB).

## See Also

dataset | table

## Topics

“Dataset Arrays” on page 2-142

“Tables” (MATLAB)

**Introduced in R2013b**

## datasetfun

**Class:** dataset

Apply function to dataset array variables

---

**Note** The `dataset` data type might be removed in a future release. To work with heterogeneous data, use the MATLAB® `table` data type instead. See MATLAB `table` documentation for more information.

---

## Syntax

```
b = datasetfun(fun,A)
[b,c,...] = datasetfun(fun,A)
[b,...] = datasetfun(fun,A,...,'UniformOutput',false)
[b,...] = datasetfun(fun,A,...,'DatasetOutput',true)
[b,...] = datasetfun(fun,A,...,'DataVars',vars)
[b,...] = datasetfun(fun,A,...,'ObsNames',obsnames)
[b,...] = datasetfun(fun,A,...,'ErrorHandler',efun)
```

## Description

`b = datasetfun(fun,A)` applies the function specified by `fun` to each variable of the dataset array `A`, and returns the results in the vector `b`. The  $i$ th element of `b` is equal to `fun` applied to the  $i$ th dataset variable of `A`. `fun` is a function handle to a function that takes one input argument and returns a scalar value. `fun` must return values of the same class each time it is called, and `datasetfun` concatenates them into the vector `b`. The outputs from `fun` must be one of the following types: numeric, logical, character, structure, or cell.

To apply functions that return results that are nonscalar or of different sizes and types, use the `'UniformOutput'` or `'DatasetOutput'` parameters described below.

Do not rely on the order in which `datasetfun` computes the elements of `b`, which is unspecified.

If `fun` is bound to more than one built-in function or file, (that is, if it represents a set of overloaded functions), `datasetfun` follows MATLAB dispatching rules in calling the function. (See “Function Precedence Order” (MATLAB).)

`[b,c,...] = datasetfun(fun,A)`, where `fun` is a function handle to a function that returns multiple outputs, returns vectors `b`, `c`, ..., each corresponding to one of the output arguments of `fun`. `datasetfun` calls `fun` each time with as many outputs as there are in the call to `datasetfun`. `fun` may return output arguments having different classes, but the class of each output must be the same each time `fun` is called.

`[b,...] = datasetfun(fun,A,...,'UniformOutput',false)` allows you to specify a function `fun` that returns values of different sizes or types. `datasetfun` returns a cell array (or multiple cell arrays), where the *i*th cell contains the value of `fun` applied to the *i*th dataset variable of `A`. Setting `'UniformOutput'` to `true` is equivalent to the default behavior.

`[b,...] = datasetfun(fun,A,...,'DatasetOutput',true)` specifies that the output(s) of `fun` are returned as variables in a dataset array (or multiple dataset arrays). `fun` must return values with the same number of rows each time it is called, but it may return values of any type. The variables in the output dataset array(s) have the same names as the variables in the input. Setting `'DatasetOutput'` to `false` (the default) specifies that the type of the output(s) from `datasetfun` is determined by `'UniformOutput'`.

`[b,...] = datasetfun(fun,A,...,'DataVars',vars)` allows you to apply `fun` only to the dataset variables in `A` specified by `vars`. `vars` is a positive integer, a vector of positive integers, a variable name, a cell array containing one or more variable names, or a logical vector.

`[b,...] = datasetfun(fun,A,...,'ObsNames',obsnames)` specifies observation names for the dataset output when `'DatasetOutput'` is `true`.

`[b,...] = datasetfun(fun,A,...,'ErrorHandler',efun)`, where `efun` is a function handle, specifies the MATLAB function to call if the call to `fun` fails. The error-handling function is called with the following input arguments:

- A structure with the fields `identifier`, `message`, and `index`, respectively containing the identifier of the error that occurred, the text of the error message, and the linear index into the input array(s) at which the error occurred

- The set of input arguments at which the call to the function failed

The error-handling function should either re-throw an error, or return the same number of outputs as `fun`. These outputs are then returned as the outputs of `datasetfun`. If 'UniformOutput' is true, the outputs of the error handler must also be scalars of the same type as the outputs of `fun`. For example, the following code could be saved in a file as the error-handling function:

```
function [A,B] = errorFunc(S,varargin)

warning(S.identifier,S.message);
A = NaN;
B = NaN;
```

If an error-handling function is not specified, the error from the call to `fun` is rethrown.

## Examples

### Work With Datasets Using Function Handles

Use function handles to compute the mean and plot a histogram of selected variables in a dataset array.

Load the sample data.

```
load hospital
```

Use `datasetfun` to compute the means of the `Weight` and `BloodPressure` variables, and store the results in a dataset array.

```
stats = datasetfun(@mean,hospital,...
    'DataVars',{'Weight','BloodPressure'},...
    'UniformOutput',false)

stats =

    1x2 cell array

    {[154]}    {1x2 double}
```



The variable `BloodPressure` contains two columns: One for the systolic measurement, and one for the diastolic measurement.

Display the mean of the blood pressure variable.

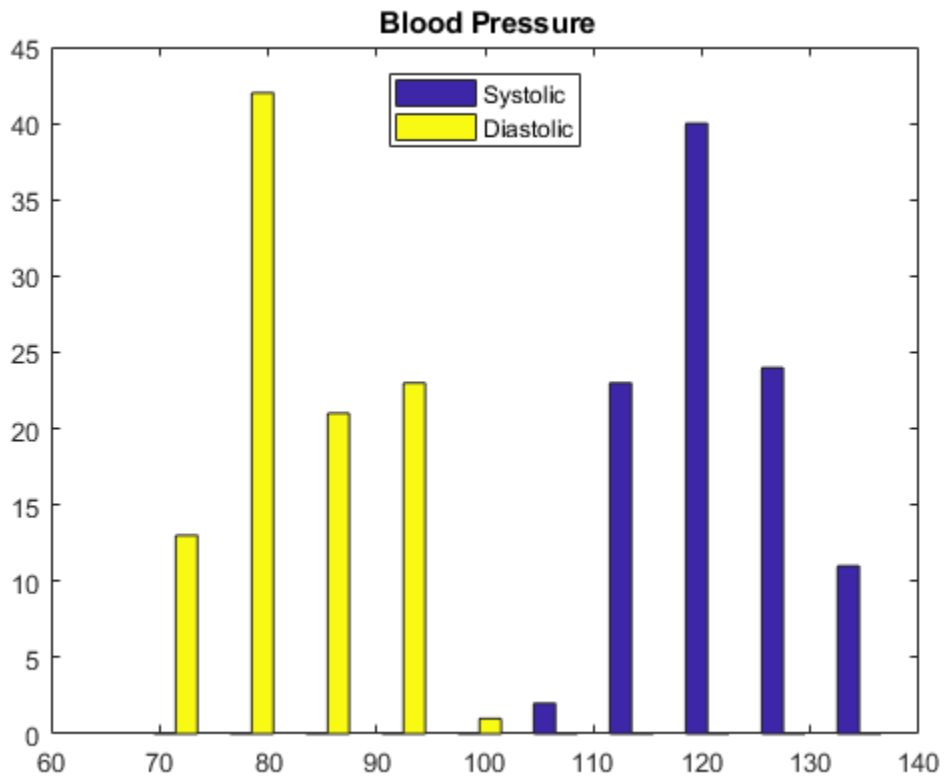
```
stats{2}
```

```
ans =
```

```
122.7800    82.9600
```

Plot a histogram of the blood pressure variable.

```
datasetfun(@hist,hospital,...  
           'DataVars','BloodPressure',...  
           'UniformOutput',false);  
title({'\bf Blood Pressure'})  
legend('Systolic','Diastolic','Location','N')
```



## See Also

`grpstats`

# daugment

*D*-optimal augmentation

## Syntax

```
dCE2 = daugment(dCE, mruns)
[dCE2, X] = daugment(dCE, mruns)
[dCE2, X] = daugment(dCE, mruns, model)
[dCE2, X] = daugment(..., param1, val1, param2, val2, ...)
```

## Description

`dCE2 = daugment(dCE, mruns)` uses a coordinate-exchange algorithm to *D*-optimally add `mruns` runs to an existing experimental design `dCE` for a linear additive model.

`[dCE2, X] = daugment(dCE, mruns)` also returns the design matrix `X` associated with the augmented design.

`[dCE2, X] = daugment(dCE, mruns, model)` uses the linear regression model specified in `model`. `model` is one of the following:

- 'linear' — Constant and linear terms. This is the default.
- 'interaction' — Constant, linear, and interaction terms
- 'quadratic' — Constant, linear, interaction, and squared terms
- 'purequadratic' — Constant, linear, and squared terms

The order of the columns of `X` for a full quadratic model with  $n$  terms is:

- 1 The constant term
- 2 The linear terms in order 1, 2, ...,  $n$
- 3 The interaction terms in order (1, 2), (1, 3), ..., (1,  $n$ ), (2, 3), ..., ( $n - 1$ ,  $n$ )
- 4 The squared terms in order 1, 2, ...,  $n$

Other models use a subset of these terms, in the same order.

Alternatively, *model* can be a matrix specifying polynomial terms of arbitrary order. In this case, *model* should have one column for each factor and one row for each term in the model. The entries in any row of *model* are powers for the factors in the columns. For example, if a model has factors *X1*, *X2*, and *X3*, then a row `[0 1 2]` in *model* specifies the term  $(X1.^0) .* (X2.^1) .* (X3.^2)$ . A row of all zeros in *model* specifies a constant term, which can be omitted.

`[dCE2,X] = daugment(..., param1, val1, param2, val2, ...)` specifies additional parameter/value pairs for the design. Valid parameters and their values are listed in the following table.

Parameter	Value
'bounds'	Lower and upper bounds for each factor, specified as a 2-by- <code>nfactors</code> matrix, where <code>nfactors</code> is the number of factors. Alternatively, this value can be a cell array containing <code>nfactors</code> elements, each element specifying the vector of allowable values for the corresponding factor.
'categorical'	Indices of categorical predictors.
'display'	Either 'on' or 'off' to control display of the iteration counter. The default is 'on'.
'excludedefun'	Handle to a function that excludes undesirable runs. If the function is <i>f</i> , it must support the syntax $b = f(S)$ , where <i>S</i> is a matrix of treatments with <code>nfactors</code> columns, where <code>nfactors</code> is the number of factors, and <i>b</i> is a vector of Boolean values with the same number of rows as <i>S</i> . <i>b</i> ( <i>i</i> ) is true if the <i>i</i> th row <i>S</i> should be excluded.
'init'	Initial design as an <code>mrns-by-nfactors</code> matrix, where <code>nfactors</code> is the number of factors. The default is a randomly selected set of points.
'levels'	Vector of number of levels for each factor.
'maxiter'	Maximum number of iterations. The default is 10.

Parameter	Value
'options'	<p>The value is a structure that contains options specifying whether to compute multiple tries in parallel, and specifying how to use random numbers when generating the starting points for the tries. Create the options structure with <code>statset</code>. Applicable <code>statset</code> parameters are:</p> <ul style="list-style-type: none"> <li>• 'UseParallel' — If <code>true</code> and if a <code>parpool</code> of the Parallel Computing Toolbox is open, compute in parallel. If the Parallel Computing Toolbox is not installed, or a <code>parpool</code> is not open, computation occurs in serial mode. Default is <code>false</code>, meaning serial computation.</li> <li>• <code>UseSubstreams</code> — Set to <code>true</code> to compute in parallel in a reproducible fashion. Default is <code>false</code>. To compute reproducibly, set <code>Streams</code> to a type allowing substreams: <code>'mlfg6331_64'</code> or <code>'mrg32k3a'</code>.</li> <li>• <code>Streams</code> — A <code>RandStream</code> object or cell array of such objects. If you do not specify <code>Streams</code>, <code>daugment</code> uses the default stream or streams. If you choose to specify <code>Streams</code>, use a single object except in the case <ul style="list-style-type: none"> <li>• You have an open Parallel pool</li> <li>• <code>UseParallel</code> is <code>true</code></li> <li>• <code>UseSubstreams</code> is <code>false</code></li> </ul> <p>In that case, use a cell array the same size as the Parallel pool.</p> </li> </ul>
'tries'	<p>Number of times to try to generate a design from a new starting point. The algorithm uses random points for each try, except possibly the first. The default is 1.</p>

---

**Note** The `daugment` function augments an existing design using a coordinate-exchange algorithm; the `'start'` parameter of the `candexch` function provides the same functionality using a row-exchange algorithm.

---

## Examples

The following eight-run design is adequate for estimating main effects in a four-factor model:

```
dCEmain = cordexch(4,8)
dCEmain =
    1   -1   -1    1
   -1   -1    1    1
   -1    1   -1    1
    1    1    1   -1
    1    1    1    1
   -1    1   -1   -1
    1   -1   -1   -1
   -1   -1    1   -1
```

To estimate the six interaction terms in the model, augment the design with eight additional runs:

```
dCEinteraction = daugment(dCEmain,8,'interaction')
dCEinteraction =
    1   -1   -1    1
   -1   -1    1    1
   -1    1   -1    1
    1    1    1   -1
    1    1    1    1
   -1    1   -1   -1
    1   -1   -1   -1
   -1   -1    1   -1
   -1    1    1    1
   -1   -1   -1   -1
    1   -1    1   -1
    1    1   -1    1
   -1    1    1   -1
    1    1   -1   -1
    1   -1    1    1
    1    1    1   -1
```

The augmented design is full factorial, with the original eight runs in the first eight rows.

## See Also

candexch | cordexch | dcovary

**Introduced before R2006a**

## dcovary

*D*-optimal design with fixed covariates

### Syntax

```
dcv = dcovary(nfactors, fixed)
[dcv, X] = dcovary(nfactors, fixed)
[dcv, X] = dcovary(nfactors, fixed, model)
[dcv, X] = daugment(..., param1, val1, param2, val2, ...)
```

### Description

`dcv = dcovary(nfactors, fixed)` uses a coordinate-exchange algorithm to generate a *D*-optimal design for a linear additive model with `nfactors` factors, subject to the constraint that the model include the fixed covariate factors in `fixed`. The number of runs in the design is the number of rows in `fixed`. The design `dcv` augments `fixed` with initial columns for treatments of the model terms.

`[dcv, X] = dcovary(nfactors, fixed)` also returns the design matrix `X` associated with the design.

`[dcv, X] = dcovary(nfactors, fixed, model)` uses the linear regression model specified in `model`. `model` is one of the following:

- 'linear' — Constant and linear terms. This is the default.
- 'interaction' — Constant, linear, and interaction terms
- 'quadratic' — Constant, linear, interaction, and squared terms
- 'purequadratic' — Constant, linear, and squared terms

The order of the columns of `X` for a full quadratic model with  $n$  terms is:

- 1 The constant term
- 2 The linear terms in order 1, 2, ...,  $n$



- 3 The interaction terms in order (1, 2), (1, 3), ..., (1,  $n$ ), (2, 3), ..., ( $n - 1$ ,  $n$ )
- 4 The squared terms in order 1, 2, ...,  $n$

Other models use a subset of these terms, in the same order.

Alternatively, *model* can be a matrix specifying polynomial terms of arbitrary order. In this case, *model* should have one column for each factor and one row for each term in the model. The entries in any row of *model* are powers for the factors in the columns. For example, if a model has factors  $X_1$ ,  $X_2$ , and  $X_3$ , then a row  $[0 \ 1 \ 2]$  in *model* specifies the term  $(X_1.^0) .* (X_2.^1) .* (X_3.^2)$ . A row of all zeros in *model* specifies a constant term, which can be omitted.

`[dCV,X] = daugment(..., param1, val1, param2, val2, ...)` specifies additional parameter/value pairs for the design. Valid parameters and their values are listed in the following table.

Parameter	Value
'bounds'	Lower and upper bounds for each factor, specified as a 2-by- <code>nfactors</code> matrix. Alternatively, this value can be a cell array containing <code>nfactors</code> elements, each element specifying the vector of allowable values for the corresponding factor.
'categorical'	Indices of categorical predictors.
'display'	Either 'on' or 'off' to control display of the iteration counter. The default is 'on'.
'excludedefun'	Handle to a function that excludes undesirable runs. If the function is $f$ , it must support the syntax $b = f(S)$ , where $S$ is a matrix of treatments with <code>nfactors</code> columns and $b$ is a vector of Boolean values with the same number of rows as $S$ . $b(i)$ is true if the $i$ th row $S$ should be excluded.
'init'	Initial design as an <code>mrns-by-nfactors</code> matrix. The default is a randomly selected set of points.
'levels'	Vector of number of levels for each factor.
'maxiter'	Maximum number of iterations. The default is 10.

Parameter	Value
'options'	<p>The value is a structure that contains options specifying whether to compute multiple tries in parallel, and specifying how to use random numbers when generating the starting points for the tries. Create the options structure with <code>statset</code>. Applicable <code>statset</code> parameters are:</p> <ul style="list-style-type: none"> <li>• 'UseParallel' — If <code>true</code> and if a <code>parpool</code> of the Parallel Computing Toolbox is open, compute in parallel. If the Parallel Computing Toolbox is not installed, or a <code>parpool</code> is not open, computation occurs in serial mode. Default is <code>false</code>, meaning serial computation.</li> <li>• <code>UseSubstreams</code> — Set to <code>true</code> to compute in parallel in a reproducible fashion. Default is <code>false</code>. To compute reproducibly, set <code>Streams</code> to a type allowing substreams: <code>'mlfg6331_64'</code> or <code>'mrg32k3a'</code>.</li> <li>• <code>Streams</code> — A <code>RandStream</code> object or cell array of such objects. If you do not specify <code>Streams</code>, <code>dcovary</code> uses the default stream or streams. If you choose to specify <code>Streams</code>, use a single object except in the case <ul style="list-style-type: none"> <li>• You have an open Parallel pool</li> <li>• <code>UseParallel</code> is <code>true</code></li> <li>• <code>UseSubstreams</code> is <code>false</code></li> </ul> <p>In that case, use a cell array the same size as the Parallel pool.</p> </li> </ul>
'tries'	<p>Number of times to try to generate a design from a new starting point. The algorithm uses random points for each try, except possibly the first. The default is 1.</p>

## Examples

### Example 1

Suppose you want a design to estimate the parameters in a three-factor linear additive model, with eight runs that necessarily occur at different times. If the process

experiences temporal linear drift, you may want to include the run time as a variable in the model. Produce the design as follows:

```
time = linspace(-1,1,8)';
[dCV1,X] = dcovary(3,time,'linear')
dCV1 =
   -1.0000    1.0000    1.0000   -1.0000
    1.0000   -1.0000   -1.0000   -0.7143
   -1.0000   -1.0000   -1.0000   -0.4286
    1.0000   -1.0000    1.0000   -0.1429
    1.0000    1.0000   -1.0000    0.1429
   -1.0000    1.0000   -1.0000    0.4286
    1.0000    1.0000    1.0000    0.7143
   -1.0000   -1.0000    1.0000    1.0000
X =
    1.0000   -1.0000    1.0000    1.0000   -1.0000
    1.0000    1.0000   -1.0000   -1.0000   -0.7143
    1.0000   -1.0000   -1.0000   -1.0000   -0.4286
    1.0000    1.0000   -1.0000    1.0000   -0.1429
    1.0000    1.0000    1.0000   -1.0000    0.1429
    1.0000   -1.0000    1.0000   -1.0000    0.4286
    1.0000    1.0000    1.0000    1.0000    0.7143
    1.0000   -1.0000   -1.0000    1.0000    1.0000
```

The column vector `time` is a fixed factor, normalized to values between  $\pm 1$ . The number of rows in the fixed factor specifies the number of runs in the design. The resulting design `dCV` gives factor settings for the three controlled model factors at each time.

## Example 2

The following example uses the `dummyvar` function to block an eight-run experiment into 4 blocks of size 2 for estimating a linear additive model with two factors:

```
fixed = dummyvar([1 1 2 2 3 3 4 4]);
dCV2 = dcovary(2, fixed(:,1:3), 'linear')
dCV2 =
    1    1    1    0    0
   -1   -1    1    0    0
   -1    1    0    1    0
    1   -1    0    1    0
    1    1    0    0    1
   -1   -1    0    0    1
   -1    1    0    0    0
    1   -1    0    0    0
```

The first two columns of `dCV2` contain the settings for the two factors; the last three columns are dummy variable codings for the four blocks.

## See Also

`cordexch` | `daugment` | `dummyvar`

**Introduced before R2006a**

## DefaultYfit property

**Class:** `TreeBagger`

Default value returned by `predict` and `oobPredict`

### Description

The `DefaultYfit` property controls what predicted value `TreeBagger` returns when no prediction is possible, for example when the `oobPredict` method needs to predict for an observation that is in-bag for all trees in the ensemble.

For classification, you can set this property to either `' '` or `'MostPopular'`. If you choose `'MostPopular'` (default), the property value becomes the name of the most probable class in the training data.

For regression, you can set this property to any numeric scalar. The default is the mean of the response for the training data.

If you set this property to `' '` for classification or `NaN` for regression, `TreeBagger` excludes the in-bag observations from computation of the out-of-bag error and margin.

### See Also

`OOBIndices` | `Predict` | `oobPredict`

## delete

**Class:** grandstream

Delete handle object

## Syntax

```
delete(h)
```

## Description

`delete(h)` deletes the handle object `h`, where `h` is a scalar handle. The `delete` method deletes a handle object but does not clear the handle from the workspace. A deleted handle is no longer valid.

## See Also

`clear` | `isvalid` | `grandstream`

## DeltaCritDecisionSplit property

**Class:** TreeBagger

Split criterion contributions for each predictor

### Description

The `DeltaCritDecisionSplit` property is a numeric array of size 1-by-Nvars of changes in the split criterion summed over splits on each variable, averaged across the entire ensemble of grown trees.

### See Also

`ClassificationTree` | `RegressionTree` | `TreeBagger` | `fitctree` | `fitrtree`

## dendrogram

Dendrogram plot

### Syntax

```
dendrogram(tree)
dendrogram(tree, Name, Value)
```

```
dendrogram(tree, P)
dendrogram(tree, P, Name, Value)
```

```
H = dendrogram(____)
[H, T, outperm] = dendrogram(____)
```

### Description

`dendrogram(tree)` generates a dendrogram plot of the hierarchical binary cluster tree. A dendrogram consists of many *U*-shaped lines that connect data points in a hierarchical tree. The height of each *U* represents the distance between the two data points being connected.

- If there are 30 or fewer data points in the original data set, then each leaf in the dendrogram corresponds to one data point.
- If there are more than 30 data points, then `dendrogram` collapses lower branches so that there are 30 leaf nodes. As a result, some leaves in the plot correspond to more than one data point.

`dendrogram(tree, Name, Value)` uses additional options specified by one or more name-value pair arguments.

`dendrogram(tree, P)` generates a dendrogram plot with no more than *P* leaf nodes. If there are more than *P* data points in the original data set, then `dendrogram` collapses the lower branches of the tree. As a result, some leaves in the plot correspond to more than one data point.



`dendrogram(tree, P, Name, Value)` uses additional options specified by one or more name-value pair arguments.

`H = dendrogram(____)` generates a dendrogram plot and returns a vector of line handles. You can use any of the input arguments from the previous syntaxes.

`[H, T, outperm] = dendrogram(____)` also returns a vector containing the leaf node number for each object in the original data set, `T`, and a vector giving the order of the node labels of the leaves as shown in the dendrogram, `outperm`.

- It is useful to return `T` when the number of leaf nodes, `P`, is less than the total number of data points, so that some leaf nodes in the display correspond to multiple data points.
- The order of the node labels given in `outperm` is from left to right for a horizontal dendrogram, and from bottom to top for a vertical dendrogram.

## Examples

### Plot Dendrogram

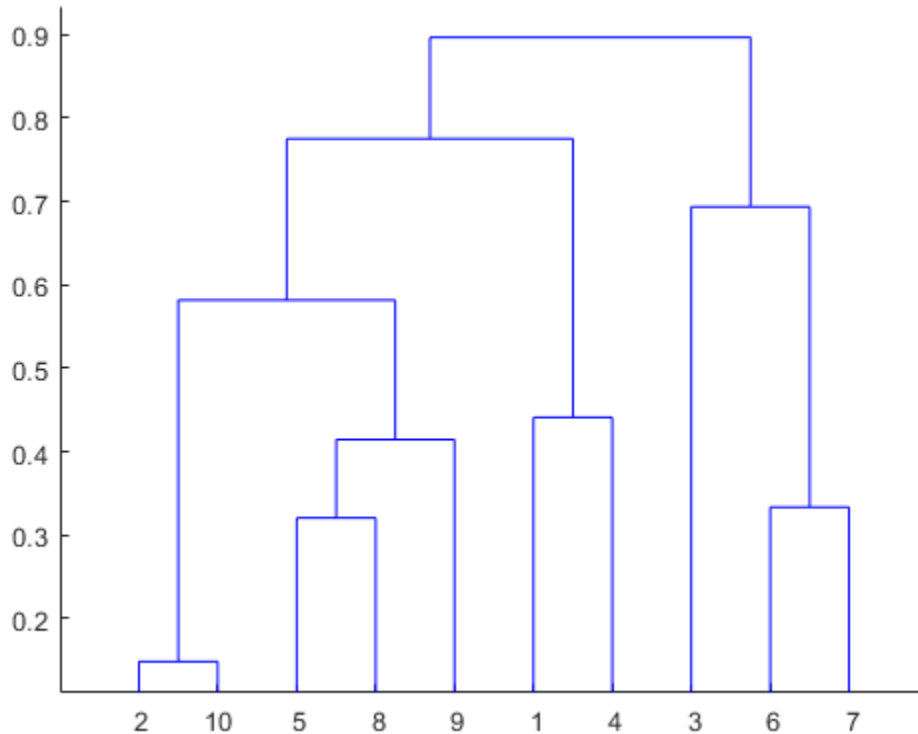
Generate sample data.

```
rng('default') % For reproducibility
X = rand(10,3);
```

Create a hierarchical binary cluster tree using `linkage`. Then, plot the dendrogram using the default options.

```
tree = linkage(X, 'average');

figure()
dendrogram(tree)
```



### Specify Dendrogram Leaf Node Order

Generate sample data.

```
rng('default') % For reproducibility  
X = rand(10,3);
```

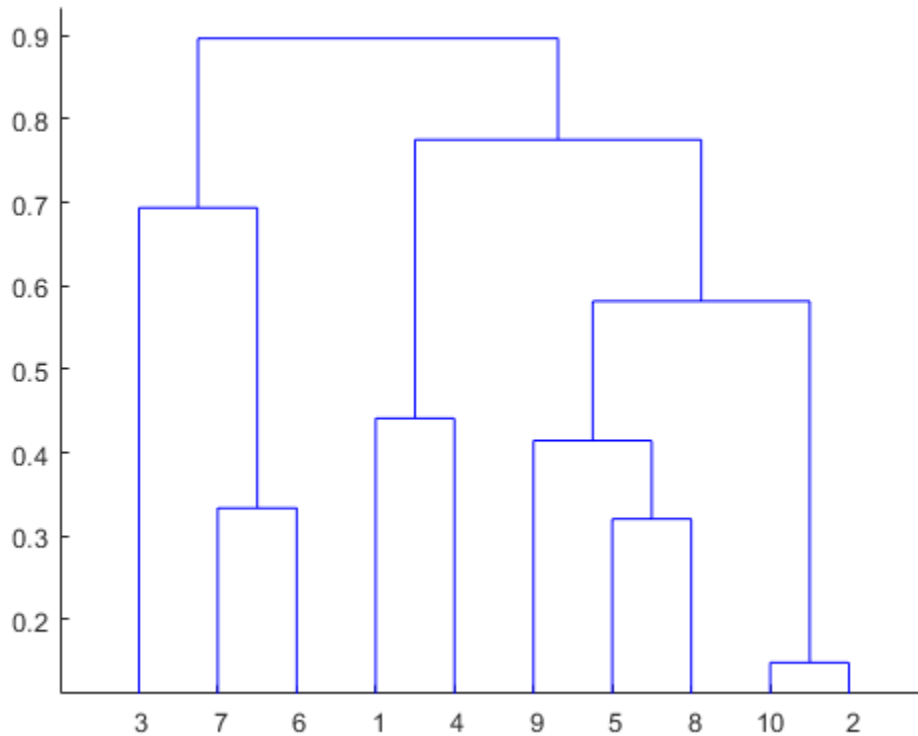
Create a hierarchical binary cluster tree using linkage.

```
tree = linkage(X, 'average');  
  
D = pdist(X);  
leafOrder = optimalleaforder(tree,D)
```

```
leafOrder =  
    3    7    6    1    4    9    5    8    10    2
```

Plot the dendrogram using an optimal leaf order.

```
figure()  
dendrogram(tree, 'Reorder', leafOrder)
```



The order of the leaf nodes in the dendrogram plot corresponds - from left to right - to the permutation in leafOrder.

### Specify Number of Nodes in Dendrogram Plot

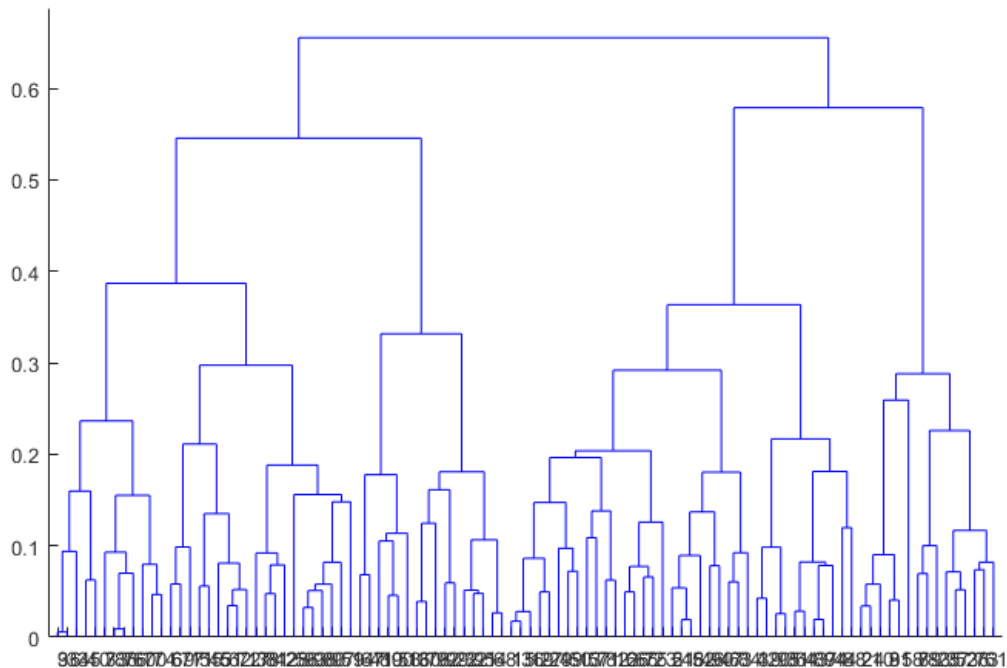
Generate sample data.

```
rng('default') % For reproducibility
X = rand(100,2);
```

There are 100 data points in the original data set, X.

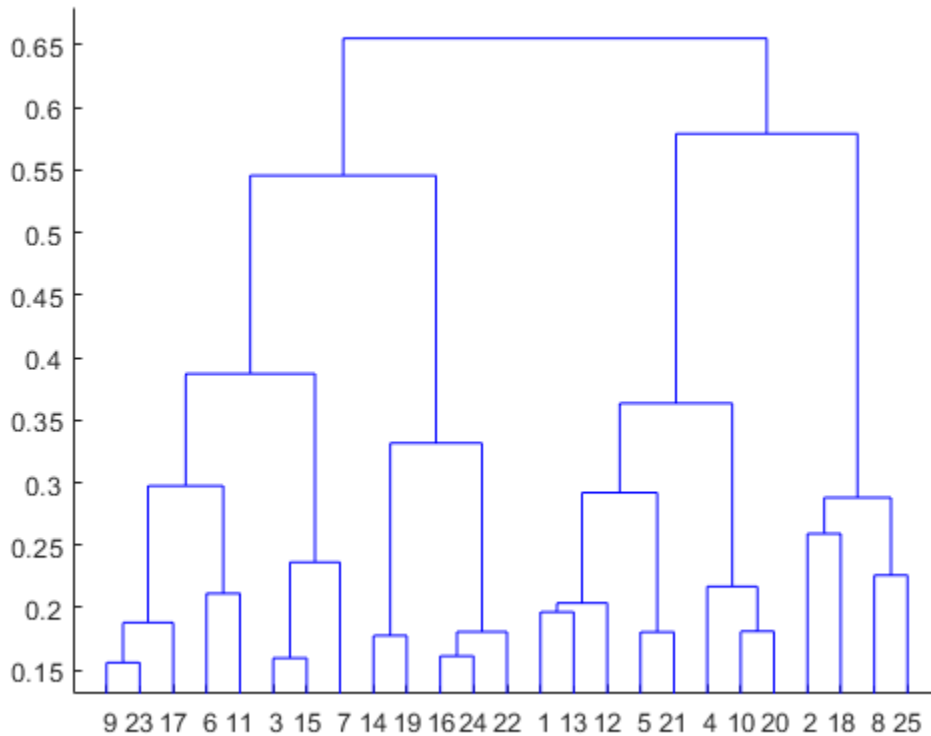
Create a hierarchical binary cluster tree using `linkage`. Then, plot the dendrogram for the complete tree (100 leaf nodes) by setting the input argument `P` equal to 0.

```
tree = linkage(X, 'average');
dendrogram(tree, 0)
```



Now, plot the dendrogram with only 25 leaf nodes. Return the mapping of the original data points to the leaf nodes shown in the plot.

```
figure
[~,T] = dendrogram(tree,25);
```



List the original data points that are in leaf node 7 of the dendrogram plot.

```
find(T==7)
```

```
ans =
```

```
7
33
60
70
74
```

76  
86

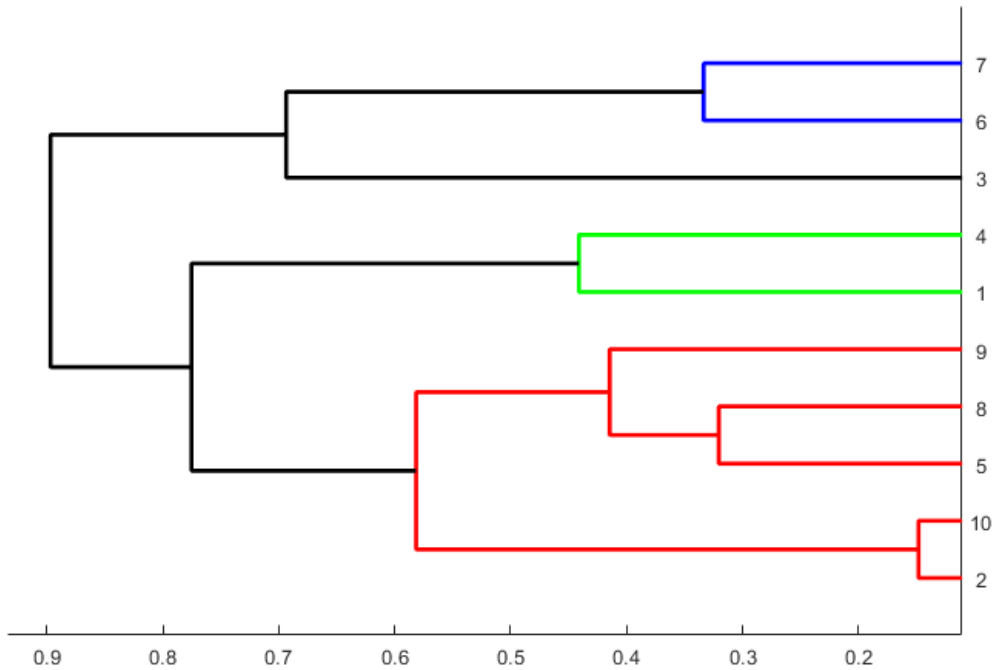
### Change Dendrogram Orientation and Line Width

Generate sample data.

```
rng('default') % For reproducibility  
X = rand(10,3);
```

Create a hierarchical binary cluster tree using `linkage`. Then, plot the dendrogram with a vertical orientation, using the default color threshold. Return handles to the lines so you can change the dendrogram line widths.

```
tree = linkage(X, 'average');  
H = dendrogram(tree, 'Orientation', 'left', 'ColorThreshold', 'default');  
set(H, 'LineWidth', 2)
```



## Input Arguments

**tree** — Hierarchical binary cluster tree  
matrix returned by `linkage`

Hierarchical binary cluster tree, specified as an  $(M - 1)$ -by-3 matrix that you generate using `linkage`, where  $M$  is the number of data points in the original data set.

**P** — Maximum number of leaf nodes  
30 (default) | positive integer value

Maximum number of leaf nodes to include in the dendrogram plot, specified as a positive integer value.

- If there are  $P$  or fewer data points in the original data set, then each leaf in the dendrogram corresponds to one data point.
- If there are more than  $P$  data points, then `dendrogram` collapses lower branches so that there are  $P$  leaf nodes. As a result, some leaves in the plot correspond to more than one data point.

If you do not specify  $P$ , then `dendrogram` uses 30 as the maximum number of leaf nodes. To display the complete tree, set  $P$  equal to 0.

Data Types: `single` | `double`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'Orientation', 'left', 'Reorder', myOrder` specifies a vertical dendrogram with leaves in the order specified by `myOrder`.

### **Reorder** — Order of leaf nodes

vector

Order of leaf nodes in the dendrogram plot, specified as the comma-separated pair consisting of `'Reorder'` and a vector giving the order of nodes in the complete tree. The order vector must be a permutation of the vector  $1:M$ , where  $M$  is the number of data points in the original data set. Specify the order from left to right for horizontal dendrograms, and from bottom to top for vertical dendrograms.

If  $M$  is greater than the number of leaf nodes in the dendrogram plot,  $P$  (by default,  $P$  is 30), then you can only specify a permutation vector that does not separate the groups of leaves that correspond to collapsed nodes.

Example:

Data Types: `single` | `double`

### **CheckCrossing** — Indicator for whether to check for crossing branches

`true` (default) | `false`



Indicator for whether to check for crossing branches in the dendrogram plot, specified as the comma-separated pair consisting of 'CheckCrossing' and either `true` or `false`. This option is only useful when you specify a value for `Reorder`.

When `CheckCrossing` has the value `true`, `dendrogram` issues a warning if the order of the leaf nodes causes crossing branches in the plot. If the dendrogram plot does not show a complete tree (because the number of data points in the original data set is greater than  $P$ ), `dendrogram` only issues a warning when the order of the leaf nodes causes branch to cross in the dendrogram as shown in the plot. That is, there is no warning if the order causes crossing branches in the complete tree but not in the dendrogram as shown in the plot.

Data Types: `logical`

#### **ColorThreshold — Threshold for unique colors**

'default' | scalar value in the range  $(0, \max(\text{tree}(:, 3)))$

Threshold for unique colors in the dendrogram plot, specified as the comma-separated pair consisting of 'ColorThreshold' and either 'default' or a scalar value in the range  $(0, \max(\text{tree}(:, 3)))$ . If `ColorThreshold` has the value  $T$ , then `dendrogram` assigns a unique color to each group of nodes in the dendrogram whose linkage is less than  $T$ .

- If `ColorThreshold` has the value 'default', then the threshold,  $T$ , is 70% of the maximum linkage,  $0.7 * \max(\text{tree}(:, 3))$ .
- If you do not specify a value for `ColorThreshold`, or if you specify a threshold outside the range  $(0, \max(\text{tree}(:, 3)))$ , then `dendrogram` uses only one color for the dendrogram plot.

#### **Orientation — Orientation of dendrogram**

'top' (default) | 'bottom' | 'left' | 'right'

Orientation of the dendrogram in the figure window, specified as the comma-separated pair consisting of 'Orientation' and one of these values:

'top'	Top to bottom
'bottom'	Bottom to top
'left'	Left to right
'right'	Right to left

Data Types: `char`

**Labels** — Label for each data point

character array | cell array of character vectors

Label for each data point in the original data set, specified as the comma-separated pair consisting of 'Labels' and a character array or cell array of character vectors. dendrogram labels any leaves in the dendrogram plot containing a single data point with that data point's label.

Data Types: char | cell

## Output Arguments

**H** — Handles to lines

vector

Handles to lines in the dendrogram plot, returned as a vector.

**T** — Leaf node numbers

column vector

Leaf node numbers for each data point in the original data set, returned as a column vector of length  $M$ , where  $M$  is the number of data points in the original data set.

When there are fewer than  $P$  data points in the original data ( $P$  is 30, by default), all data points are displayed in the dendrogram, with each node containing a single data point. In this case,  $T$  is the identity map,  $T = (1:M)'$ .

$T$  is useful when  $P$  is less than the total number of data points. That is, when some leaf nodes in the dendrogram display correspond to multiple data points. For example, to find out which data points are contained in leaf node  $k$  of the dendrogram plot, use `find(T==k)`.

**outperm** — Permutation of node labels

vector

Permutation of the node labels of the leaves of the dendrogram as shown in the plot, returned as a row vector. `outperm` gives the order from left to right for a horizontal dendrogram, and from bottom to top for a vertical dendrogram. If there are  $P$  leaves in the dendrogram plot, `outperm` is a permutation of the vector  $1:P$ .

## See Also

`cluster` | `clusterdata` | `cophenet` | `inconsistent` | `linkage` | `pdist` | `silhouette`

**Introduced before R2006a**

## Description property

**Class:** dataset

Character vector describing data set

---

**Note** The `dataset` data type might be removed in a future release. To work with heterogeneous data, use the MATLAB® `table` data type instead. See MATLAB `table` documentation for more information.

---

## Description

`Description` is a character vector describing the data set. The default is an empty character vector.

# designecoc

Coding matrix for reducing error-correcting output code to binary

## Syntax

```
M = designecoc(K, name)
M = designecoc(K, name, Name, Value)
```

## Description

`M = designecoc(K, name)` returns the coding matrix  $M$  that reduces the error-correcting output code (ECOC) design specified by `name` and  $K$  classes to a binary problem.  $M$  has  $K$  rows and  $L$  columns, with each row corresponding to a class and each column corresponding to a binary learner. `name` and  $K$  determine the value of  $L$ .

You can view or customize  $M$ , and then specify it as the coding matrix for training an ECOC multiclass classifier using `fitcecoc`.

`M = designecoc(K, name, Name, Value)` returns the coding matrix with additional options specified by one or more `Name, Value` pair arguments.

For example, you can specify the number of trials when generating a dense or sparse, random coding matrix.

## Examples

### Train ECOC Classifiers Using a Custom Coding Design

Consider the `arrhythmia` data set. There are 16 classes in the study, 13 of which are represented in the data. The first class indicates that the subject did not have arrhythmia, and the last class indicates that the subject's arrhythmia state was not recorded. Suppose that the other classes are ordinal levels indicating the severity of

arrhythmia. Train an ECOC classifier using a custom coding design specified by the description of the classes.

Load the arrhythmia data set.

```
load arrhythmia
K = 13; % Number of distinct classes
```

Construct a coding matrix that describes the nature of the classes.

```
OrdMat = designecoc(11, 'ordinal');
nOM = size(OrdMat);
class1VSOrd = [1; -ones(11,1); 0];
class1VSClass16 = [1; zeros(11,1); -1];
OrdVSClass16 = [0; ones(11,1); -1];
Coding = [class1VSOrd class1VSClass16 OrdVSClass16, ...
          [zeros(1,nOM(2)); OrdMat; zeros(1,nOM(2))]]
```

Coding =

1	1	0	0	0	0	0	0	0	0	0	0	0
-1	0	1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	0	1	1	-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	0	1	1	1	-1	-1	-1	-1	-1	-1	-1	-1
-1	0	1	1	1	1	-1	-1	-1	-1	-1	-1	-1
-1	0	1	1	1	1	1	-1	-1	-1	-1	-1	-1
-1	0	1	1	1	1	1	1	-1	-1	-1	-1	-1
-1	0	1	1	1	1	1	1	1	-1	-1	-1	-1
-1	0	1	1	1	1	1	1	1	1	-1	-1	-1
-1	0	1	1	1	1	1	1	1	1	1	-1	-1
-1	0	1	1	1	1	1	1	1	1	1	1	-1
-1	0	1	1	1	1	1	1	1	1	1	1	1
0	-1	-1	0	0	0	0	0	0	0	0	0	0

Train an ECOC classifier using the custom coding design Coding and specify that the binary learners are decision trees.

```
Mdl = fitcecoc(X,Y, 'Coding', Coding, 'Learner', 'Tree');
```

Estimate the in-sample classification error.

```
genErr = resubLoss(Mdl)
```

```
genErr =
    0.1460
```

## Choose Among Several Random Coding Designs

If you request a random coding matrix by specifying `sparserandom` or `denserandom`, then, by default, `designecoc` generates 10,000 random matrices. Then, it chooses the matrix with the largest, minimal, pair-wise row distances based on the Hamming measure. You can specify to generate more matrices to increase the chance of obtaining a better one, or you can generate several coding matrices, and then see which performs best.

Load the `arrhythmia` data set. Reserve the observations classified into class 16 (i.e., those that do not have an `arrhythmia` classification) as new data.

```
load arrhythmia
oosIdx = Y == 16;
isIdx = ~oosIdx;
Y = categorical(Y(isIdx));
tabulate(Y)
K = numel(unique(Y));
```

Value	Count	Percent
1	245	56.98%
2	44	10.23%
3	15	3.49%
4	15	3.49%
5	13	3.02%
6	25	5.81%
7	3	0.70%
8	2	0.47%
9	9	2.09%
10	50	11.63%
14	4	0.93%
15	5	1.16%

Generate four random coding design matrices such that the first two are dense and the second two are sparse. Specify to find the best out of 20,000 variates.

```
rng(1); % For reproducibility

Coding = cell(4,1); % Preallocate for coding matrices
CodingTypes = {'denserandom','denserandom','sparserandom','sparserandom'};
for j = 1:4;
    Coding{j} = designecoc(K,CodingTypes{j},'NumTrials',2e4);
end
```

Coding is a 4-by-1 cell array, where each cell is a coding design matrix. The matrices have  $K$  rows, but the number of columns (i.e., binary learners) might vary.

Train and cross validate ECOC classifiers using the 15-fold cross validation. Specify that each ECOC classifier be trained using a classification tree, and the random coding matrix stored in Coding.

```
Mdl = cell(4,1); % Preallocate for the ECOC classifiers
for j = 1:4;
    Mdl{j} = fitcecoc(X(isIdx,:),Y,'Learners','tree',...
        'Coding',Coding{j},'KFold',15);
end
```

```
Warning: One or more folds do not contain points from all the groups.
Warning: One or more folds do not contain points from all the groups.
Warning: One or more folds do not contain points from all the groups.
Warning: One or more folds do not contain points from all the groups.
```

Mdl is a 4-by-1 cell array of ClassificationPartitionedECOC models. Several classes have low relative frequency in the data, and so there is a chance that, during cross validation, some in-sample folds will not train using observations from those classes.

Estimate the 15-fold classification error for each classifier.

```
genErr = nan(4,1);
for j = 1:4;
    genErr(j) = kfoldLoss(Mdl{j});
end
```

```
genErr
```

```
genErr =
    0.2349
    0.2163
```



```
0.2140
0.2279
```

Though the generalization error is still high, the best performing model, based solely on the out-of-sample classification error, is the model that used the coding design `Coding{3}`.

You can try to improve the generalization error by tuning some parameters of the binary learners. For example, you can specify to use the twoing rule or deviance for the split criterion, rather than the default Gini's diversity index. You might also specify to use surrogate splits since there are missing values in the data.

## Input Arguments

### **$K$ — Number of classes**

positive integer

Number of classes, specified as a positive integer.

$K$  specifies the number of rows of the coding matrix  $M$ .

Data Types: `single` | `double`

### **name — Coding design name**

'binarycomplete' | 'denserandom' | 'onevsall' | 'onevsone' |  
'sparserandom' | ...

Coding design name, specified as 'binarycomplete', 'denserandom', 'onevsall', 'onevsone', or 'sparserandom'. This table summarizes the available coding schemes.

Value	Number of Binary Learners	Description
'allpairs' and 'onevsone'	$K(K-1)/2$	For each binary learner, one class is positive, another is negative, and the software ignores the rest. This design exhausts all combinations of class pair assignments.

Value	Number of Binary Learners	Description
'binarycomplete'	$2^{(K-1)} - 1$	This design partitions the classes into all binary combinations, and does not ignore any classes. For each binary learner, all class assignments are -1 and 1 with at least one positive and negative class in the assignment.
'denserandom'	Random, but approximately $10 \log_2 K$	For each binary learner, the software randomly assigns classes into positive or negative classes, with at least one of each type. For more details, see “Random Coding Design Matrices” on page 32-1923.
'onevsall'	$K$	For each binary learner, one class is positive and the rest are negative. This design exhausts all combinations of positive class assignments.
'ordinal'	$K - 1$	For the first binary learner, the first class is negative, and the rest positive. For the second binary learner, the first two classes are negative, the rest positive, and so on.

Value	Number of Binary Learners	Description
'sparsrandom'	Random, but approximately $15 \log_2 K$	For each binary learner, the software randomly assigns classes as positive or negative with probability 0.25 for each, and ignores classes with probability 0.5. For more details, see “Random Coding Design Matrices” on page 32-1923.
'ternarycomplete'	$(3^K - 2^{(K+1)} + 1) / 2$	This design partitions the classes into all ternary combinations. All class assignments are 0, -1, and 1 with at least one positive and one negative class in the assignment.

Data Types: char

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'NumTrials', 1000` specifies to generate 1000 random matrices.

### **NumTrials** — Number of random coding matrices to generate

10000 (default) | positive integer

Number of random coding matrices to generate, specified as the comma-separated pair consisting of `'NumTrials'` and a positive integer.

The software:

- Generates `NumTrials` matrices, and selects the one with the maximal, pair-wise row distance.

- Ignores `NumTrials` for all values of `name` except `'denserandom'` and `'sparserandom'`.

Example: `'NumTrials', 1000`

Data Types: `single` | `double`

## Output Arguments

### **M** — Coding matrix

numeric matrix

Coding matrix that reduces an ECOC scheme to binary, returned as a numeric matrix. `M` has `K` rows and `L` columns, where `L` is the number of binary learners. Each row corresponds to a class and each column corresponds to a binary learner.

The elements of `M` are `-1`, `0`, or `1`, and the value corresponds to a dichotomous class assignment. This table describes the meaning of  $M(i, j)$ , that is, the class that learner `j` assigns to observations in class `i`.

Value	Dichotomous Class Assignment
-1	Negative class
0	Before training, learner <code>j</code> removes observations in class <code>i</code> from the data set.
1	Positive class

The binary learners for designs `denserandom`, `binarycomplete`, and `onevsall` do not assign `0` to observations in any class.

## Tips

- The number of binary learners grows with the number of classes. For a problem with many classes, the `binarycomplete` and `ternarycomplete` coding designs are not efficient. However:
  - If  $K \leq 4$ , then use `ternarycomplete` coding design rather than `sparserandom`.
  - If  $K \leq 5$ , then use `binarycomplete` coding design rather than `denserandom`.

You can display the coding design matrix of a trained ECOC classifier by entering `Mdl.CodingMatrix` into the Command Window.

- You should form a coding matrix using intimate knowledge of the application, and taking into account computational constraints. If you have sufficient computational power and time, then try several coding matrices and choose the one with the best performance (e.g., check the confusion matrices for each model using `confusionmat`).
- Leave-one-out cross-validation (`Leaveout`) is inefficient for data sets with many observations. Instead, use  $k$ -fold cross-validation (`KFold`).

## Algorithms

### Custom Coding Design Matrices

Custom coding matrices must have a certain form. The software validates custom coding matrices by ensuring:

- Every element is -1, 0, or 1.
- Every column contains as least one -1 and one 1.
- For all distinct column vectors  $u$  and  $v$ ,  $u \neq v$  and  $u \neq -v$ .
- All rows vectors are unique.
- The matrix can separate any two classes. That is, you can travel from any row to any other row following these rules:
  - You can move vertically from 1 to -1 or -1 to 1.
  - You can move horizontally from a nonzero element to another nonzero element.
  - You can use a column of the matrix for a vertical move only once.

If it is not possible to move from row  $i$  to row  $j$  using these rules, then classes  $i$  and  $j$  cannot be separated by the design. For example, in the coding design

$$\begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{bmatrix}$$

classes 1 and 2 cannot be separated from classes 3 and 4 (that is, you cannot move horizontally from the -1 in row 2 to column 2 since there is a 0 in that position). Therefore, the software rejects this coding design.

## Random Coding Design Matrices

For a given number of classes, for example,  $K$ , the software generates random coding design matrices as follows.

- 1 The software generates one of the following:
  - a Dense random — The software assigns a 1 or -1 with equal probability to each element of the  $K$ -by- $L_d$  coding design matrix, where  $L_d \approx \lceil 10 \log_2 K \rceil$ .
  - b Sparse random — The software assigns a 1 to each element of the  $K$ -by- $L_s$  coding design matrix with probability 0.25, a -1 with probability 0.25, and a 0 with probability 0.5, where  $L_s \approx \lceil 15 \log_2 K \rceil$ .
- 2 If a column does not contain at least one 1 and at least one -1, then the software removes that column.
- 3 For distinct columns  $u$  and  $v$ , if  $u = v$  or  $u = -v$ , then the software removes  $v$  from the coding design matrix.

The software randomly generates 10,000 matrices by default, and retains the matrix with the largest, minimal pairwise row distance based on the Hamming measure ([4]) given by

$$\Delta(k_1, k_2) = 0.5 \sum_{l=1}^L |m_{k_1 l} - m_{k_2 l}|,$$

where  $m_{k,l}$  is an element of coding design matrix  $j$ .

## References

- [1] Fürnkranz, Johannes. “Round Robin Classification.” *J. Mach. Learn. Res.*, Vol. 2, 2002, pp. 721–747.
- [2] Escalera, S., O. Pujol, and P. Radeva. “Separability of ternary codes for sparse designs of error-correcting output codes.” *Pattern Recog. Lett.*, Vol. 30, Issue 3, 2009, pp. 285–297.

## See Also

ClassificationECOC | fitcecoc

**Introduced in R2014b**

## devianceTest

**Class:** CompactGeneralizedLinearModel

Analysis of deviance

### Syntax

```
tbl = devianceTest(mdl)
```

### Description

`tbl = devianceTest(mdl)` returns an analysis of deviance table for the `mdl` generalized linear model. `tbl` gives the result of a test of whether the fitted model fits significantly better than a constant model.

### Input Arguments

**mdl**

Generalized linear model, specified as a full `GeneralizedLinearModel` object constructed using `fitglm` or `stepwiseglm`, or a compacted `CompactGeneralizedLinearModel` object constructed using `compact`.

### Output Arguments

**tbl**

Table containing two rows and four columns.

- The first row relates to a constant model.
- The second row relates to the full model in `mdl`.
- The columns are:



Deviance	Deviance is twice the difference between the log likelihoods of the corresponding model ( <code>mdl</code> or <code>constant</code> ) and the saturated model. The test statistic for the deviance test is twice the difference between the log likelihoods of the tested model <code>mdl</code> and the constant model. For more information, see Deviance on page 32-1380.
DFE	Error degrees of freedom. It is the number of observations minus the number of parameters in the corresponding model.
chi2Stat	<p>F statistic or Chi-squared statistic, depending on whether the dispersion is estimated (F statistic) or not (Chi-squared statistic)</p> <ul style="list-style-type: none"> <li>• Chi-squared statistic is the difference between the deviance of the constant model and the deviance of the full model.</li> <li>• F statistic is the difference between the deviance of the constant model and the deviance of the full model, divided by the estimated dispersion.</li> </ul>
pValue	$p$ -value associated with the test. It is the Chi-squared statistic with (number of coefficients in the model minus one) degrees of freedom, or $F$ statistic with (number of coefficients in the model minus one) numerator degrees of freedom, and DFE denominator degrees of freedom.

## Examples

## Deviance Test

Perform a deviance test on a generalized linear model.

Construct a generalized linear model.

```
rng('default') % for reproducibility
X = randn(100,5);
mu = exp(X(:, [1 4 5]) * [.4; .2; .3]);
y = poissrnd(mu);
mdl = fitglm(X,y, 'linear', 'Distribution', 'poisson');
```

Test whether the model differs from a constant in a statistically significant way.

```
tbl = devianceTest(mdl)
```

```
tbl =
```

```
2x4 table
```

	Deviance	DFE	chi2Stat	pValue
log(y) ~ 1	128.58	99		
log(y) ~ 1 + x1 + x2 + x3 + x4 + x5	83.726	94	44.858	1.5502e-08

The  $P$ -value is very small, indicating that the model significantly differs from a constant.

## Definitions

### Deviance

Deviance of a model  $M_1$  is twice the difference between the loglikelihood of that model and the saturated model,  $M_S$ . The saturated model is the model with the maximum number of parameters that can be estimated. For example, if there are  $n$  observations  $y_i$ ,  $i = 1, 2, \dots, n$ , with potentially different values for  $X_i^T \beta$ , then you can define a saturated model with  $n$  parameters. Let  $L(b, y)$  denote the maximum value of the likelihood function for a model. Then the deviance of model  $M_1$  is

$$-2(\log L(b_1, y) - \log L(b_S, y)),$$

where  $b_1$  are the estimated parameters for model  $M_1$  and  $b_S$  are the estimated parameters for the saturated model. The deviance has a chi-square distribution with  $n - p$  degrees of freedom, where  $n$  is the number of parameters in the saturated model and  $p$  is the number of parameters in model  $M_1$ .

If  $M_1$  and  $M_2$  are two different generalized linear models, then the fit of the models can be assessed by comparing the deviances  $D_1$  and  $D_2$  of these models. The difference of the deviances is

$$\begin{aligned} D = D_2 - D_1 &= -2(\log L(b_2, y) - \log L(b_S, y)) + 2(\log L(b_1, y) - \log L(b_S, y)) \\ &= -2(\log L(b_2, y) - \log L(b_1, y)). \end{aligned}$$

Asymptotically, this difference has a chi-square distribution with degrees of freedom  $v$  equal to the number of parameters that are estimated in one model but fixed (typically at 0) in the other. That is, it is equal to the difference in the number of parameters estimated in  $M_1$  and  $M_2$ . You can get the  $p$ -value for this test using `1 - chi2cdf(D, v)`, where  $D = D_2 - D_1$ .

## See Also

`CompactGeneralizedLinearModel` | `GeneralizedLinearModel`

## Topics

“Generalized Linear Models” on page 12-12

## designMatrix

**Class:** GeneralizedLinearMixedModel

Fixed- and random-effects design matrices

### Syntax

```
D = designMatrix(glme)
D = designMatrix(glme, 'Fixed')

D = designMatrix(glme, 'Random')
Dsub = designMatrix(glme, 'Random', gnumbers)
[Dsub, gnames] = designMatrix(glme, 'Random', gnumbers)
```

### Description

`D = designMatrix(glme)` or `D = designMatrix(glme, 'Fixed')` returns the fixed-effects design matrix for the generalized linear mixed-effects model `glme`.

`D = designMatrix(glme, 'Random')` returns the random-effects design matrix for the generalized linear mixed-effects model `glme`.

`Dsub = designMatrix(glme, 'Random', gnumbers)` returns a subset of the random-effects design matrix for the generalized linear mixed-effects model `glme` that corresponds to the grouping variables indicated by `gnumbers`.

`[Dsub, gnames] = designMatrix(glme, 'Random', gnumbers)` also returns the grouping variable names that correspond to `gnumbers`.

### Input Arguments

**glme** — Generalized linear mixed-effects model

GeneralizedLinearMixedModel object

Generalized linear mixed-effects model, specified as a `GeneralizedLinearMixedModel` object. For properties and methods of this object, see `GeneralizedLinearMixedModel`.

### **gnumbers** — Grouping variable numbers

array of integer values

Grouping variable numbers, specified as an array of integer values containing elements in the range  $[1, R]$ , where  $R$  is the length of the cell array that contains the grouping variables for the generalized linear mixed-effects model `glme`.

For example, you can specify the grouping variables  $g_1$ ,  $g_3$ , and  $g_r$  as  $[1, 3, r]$ .

Data Types: `single` | `double`

## Output Arguments

### **D** — Design matrix

matrix

Design matrix of a generalized linear mixed-effects model `glme` returned as one of the following:

- Fixed-effects design matrix —  $n$ -by- $p$  matrix consisting of the fixed-effects design matrix of `glme`, where  $n$  is the number of observations and  $p$  is the number of fixed-effects terms.
- Random-effects design matrix —  $n$ -by- $k$  matrix, consisting of the random-effects design matrix of `glme`. Here,  $k$  is equal to `length(B)`, where `B` is the random-effects coefficients vector of generalized linear mixed-effects model `glme`. The random-effects design matrix is returned as a sparse matrix. For more information, see “Sparse Matrices” (MATLAB).

If `glme` has  $R$  grouping variables  $g_1, g_2, \dots, g_R$ , with levels  $m_1, m_2, \dots, m_R$ , respectively, and if  $q_1, q_2, \dots, q_R$  are the lengths of the random-effects vectors that are associated with  $g_1, g_2, \dots, g_R$ , respectively, then `B` is a column vector of length  $q_1 * m_1 + q_2 * m_2 + \dots + q_R * m_R$ .

`B` is made by concatenating the empirical Bayes predictors of random effects vectors corresponding to each level of each grouping variable as `[g1level1; g1level2; ...; g1levelm1; g2level1; g2level2; ...; g2levelm2; ...; gRlevel1; gRlevel2; ...; gRlevelmR]`'.

Data Types: `single` | `double`

**Dsub** — Submatrix of random-effects design matrix  
matrix

Submatrix of random-effects design matrix that corresponds to the grouping variables specified by `gnumbers`, returned as an  $n$ -by- $k$  matrix, where  $k$  is length of the column vector `Bsub`.

`Bsub` contains the concatenated empirical Bayes predictors of random-effects vectors, corresponding to each level of the grouping variables, specified by `gnumbers`.

If, for example, `gnumbers` is `[1, 3, r]`, this corresponds to the grouping variables  $g_1$ ,  $g_3$ , and  $g_r$ . Then, `Bsub` contains the empirical Bayes predictors of random-effects vectors corresponding to each level of the grouping variables  $g_1$ ,  $g_3$ , and  $g_r$ , such as

```
[g1level1; g1level2; ...; g1levelm1; g3level1; g3level2; ...; g3levelm3;  
grlevel1; grlevel2; ...; grlevelmr]'.
```

Thus, `Dsub*Bsub` represents the contribution of all random effects corresponding to grouping variables  $g_1$ ,  $g_3$ , and  $g_r$  to the response of `glme`.

If `gnumbers` is empty, then `Dsub` is the full random-effects design matrix.

Data Types: `single` | `double`

**gnames** — Names of grouping variables  
 $k$ -by-1 cell array

Names of grouping variables corresponding to the integers in `gnumbers` if the design type is 'Random', returned as a  $k$ -by-1 cell array. If the design type is 'Fixed', then `gnames` is an empty matrix `[]`.

Data Types: `cell`

## Examples

### Obtain Fixed- and Random-Effects Design Matrices

Load the sample data.

```
load mfr
```

This simulated data is from a manufacturing company that operates 50 factories across the world, with each factory running a batch process to create a finished product. The company wants to decrease the number of defects in each batch, so it developed a new manufacturing process. To test the effectiveness of the new process, the company selected 20 of its factories at random to participate in an experiment: Ten factories implemented the new process, while the other ten continued to run the old process. In each of the 20 factories, the company ran five batches (for a total of 100 batches) and recorded the following data:

- Flag to indicate whether the batch used the new process (`newprocess`)
- Processing time for each batch, in hours (`time`)
- Temperature of the batch, in degrees Celsius (`temp`)
- Categorical variable indicating the supplier (A, B, or C) of the chemical used in the batch (`supplier`)
- Number of defects in the batch (`defects`)

The data also includes `time_dev` and `temp_dev`, which represent the absolute deviation of time and temperature, respectively, from the process standard of 3 hours at 20 degrees Celsius.

Fit a generalized linear mixed-effects model using `newprocess`, `time_dev`, `temp_dev`, and `supplier` as fixed-effects predictors. Include a random-effects term for intercept grouped by `factory`, to account for quality differences that might exist due to factory-specific variations. The response variable `defects` has a Poisson distribution, and the appropriate link function for this model is log. Use the Laplace fit method to estimate the coefficients. Specify the dummy variable encoding as 'effects', so the dummy variable coefficients sum to 0.

The number of defects can be modeled using a Poisson distribution  $\text{defects}_{ij} \sim \text{Poisson}(\mu_{ij})$ .

This corresponds to the generalized linear mixed-effects model

$$\log(\mu_{ij}) = \beta_0 + \beta_1 \text{newprocess}_{ij} + \beta_2 \text{time.dev}_{ij} + \beta_3 \text{temp.dev}_{ij} + \beta_4 \text{supplier.C}_{ij} + \beta_5 \text{supplier.B}_{ij} + b_i,$$

where

- $\text{defects}_{ij}$  is the number of defects observed in the batch produced by factory  $i$  during batch  $j$ .
- $\mu_{ij}$  is the mean number of defects corresponding to factory  $i$  (where  $i = 1, 2, \dots, 20$ ) during batch  $j$  (where  $j = 1, 2, \dots, 5$ ).
- $\text{newprocess}_{ij}$ ,  $\text{time\_dev}_{ij}$ , and  $\text{temp\_dev}_{ij}$  are the measurements for each variable that correspond to factory  $i$  during batch  $j$ . For example,  $\text{newprocess}_{ij}$  indicates whether the batch produced by factory  $i$  during batch  $j$  used the new process.
- $\text{supplier\_C}_{ij}$  and  $\text{supplier\_B}_{ij}$  are dummy variables that use effects (sum-to-zero) coding to indicate whether company C or B, respectively, supplied the process chemicals for the batch produced by factory  $i$  during batch  $j$ .
- $b_i \sim N(0, \sigma_b^2)$  is a random-effects intercept for each factory  $i$  that accounts for factory-specific variation in quality.

```
glme = fitglme(mfr, 'defects ~ 1 + newprocess + time_dev + temp_dev + supplier + (1|fact
```

Extract the fixed-effects design matrix and display rows 1 through 10.

```
Dfe = designMatrix(glme, 'Fixed');
disp(Dfe(1:10, :))
```

```

1.0000    0    0.1834    0.2259    1.0000    0
1.0000    0    0.3035    0.0725    0    1.0000
1.0000    0    0.0717    0.1630    1.0000    0
1.0000    0    0.1069    0.0809   -1.0000   -1.0000
1.0000    0    0.0241    0.0319    1.0000    0
1.0000    0    0.1214    0.1114    0    1.0000
1.0000    0    0.0033    0.0553    1.0000    0
1.0000    0    0.2350    0.0616    1.0000    0
1.0000    0    0.0488    0.0177    0    1.0000
1.0000    0    0.1148    0.0105    1.0000    0
```

Column 1 of the fixed-effects design matrix `Dfe` contains the constant term. Column 2, 3, and 4 contain the `newprocess`, `time_dev`, and `temp_dev` terms, respectively. Columns 5 and 6 contain dummy variables for `supplier_C` and `supplier_B`, respectively.

Extract the random-effects design matrix and display rows 1 through 10.

```
Dre = designMatrix(glme, 'Random');
disp(Dre(1:10, :))
```



```
(1,1)      1
(2,1)      1
(3,1)      1
(4,1)      1
(5,1)      1
(6,2)      1
(7,2)      1
(8,2)      1
(9,2)      1
(10,2)     1
```

Convert the sparse matrix `Dre` to a full matrix and display rows 1 through 10.

```
full(Dre(1:10,:))
```

```
ans =
```

```
Columns 1 through 13
```

```
 1   0   0   0   0   0   0   0   0   0   0   0   0
 1   0   0   0   0   0   0   0   0   0   0   0   0
 1   0   0   0   0   0   0   0   0   0   0   0   0
 1   0   0   0   0   0   0   0   0   0   0   0   0
 1   0   0   0   0   0   0   0   0   0   0   0   0
 0   1   0   0   0   0   0   0   0   0   0   0   0
 0   1   0   0   0   0   0   0   0   0   0   0   0
 0   1   0   0   0   0   0   0   0   0   0   0   0
 0   1   0   0   0   0   0   0   0   0   0   0   0
 0   1   0   0   0   0   0   0   0   0   0   0   0
```

```
Columns 14 through 20
```

```
 0   0   0   0   0   0   0
 0   0   0   0   0   0   0
 0   0   0   0   0   0   0
 0   0   0   0   0   0   0
 0   0   0   0   0   0   0
 0   0   0   0   0   0   0
 0   0   0   0   0   0   0
 0   0   0   0   0   0   0
 0   0   0   0   0   0   0
 0   0   0   0   0   0   0
```

Each column corresponds to a level of the grouping variable factory.

## See Also

`GeneralizedLinearMixedModel` | `fitglme` | `fitted` | `residuals` | `response`

# designMatrix

**Class:** LinearMixedModel

Fixed- and random-effects design matrices

## Syntax

```
D = designMatrix(lme)
D = designMatrix(lme, 'Fixed')

D = designMatrix(lme, 'Random')
Dsub = designMatrix(lme, 'Random', gnumbers)
[Dsub, gnames] = designMatrix(lme, 'Random', gnumbers)
```

## Description

`D = designMatrix(lme)` or `D = designMatrix(lme, 'Fixed')` returns the fixed-effects design matrix for the linear mixed-effects model `lme`.

`D = designMatrix(lme, 'Random')` returns the random-effects design matrix for the linear mixed-effects model `lme`.

`Dsub = designMatrix(lme, 'Random', gnumbers)` returns a subset of the random-effects design matrix for the linear mixed-effects model `lme` corresponding to the grouping variables indicated by the integers in `gnumbers`.

`[Dsub, gnames] = designMatrix(lme, 'Random', gnumbers)` also returns the grouping variable names corresponding to the integers in `gnumbers`.

## Input Arguments

**lme** — Linear mixed-effects model

LinearMixedModel object

Linear mixed-effects model, specified as a `LinearMixedModel` object constructed using `fitlme` or `fitlmematrix`.

**gnumbers** — Grouping variable numbers

integer array

Grouping variable numbers, specified as an integer array, where  $R$  is the length of the cell array that contains the grouping variables for the linear mixed-effects model `lme`.

For example, you can specify the grouping variables  $g_1$ ,  $g_3$ , and  $g_r$  as follows.

Example: `[1, 3, r]`

Data Types: `double` | `single`

## Output Arguments

**D** — Design matrix

matrix

Design matrix of a linear mixed-effects model `lme` returned as one of the following:

- Fixed-effects design matrix —  $n$ -by- $p$  matrix consisting of the fixed-effects design of `lme`, where  $n$  is the number of observations and  $p$  is the number of fixed-effects terms.
- Random-effects design matrix —  $n$ -by- $k$  matrix, consisting of the random-effects design matrix of `lme`. Here,  $k$  is equal to `length(B)`, where `B` is the random-effects coefficients vector of linear mixed-effects model `lme`.

If `lme` has  $R$  grouping variables  $g_1, g_2, \dots, g_R$ , with levels  $m_1, m_2, \dots, m_R$ , respectively, and if  $q_1, q_2, \dots, q_R$  are the lengths of the random-effects vectors that are associated with  $g_1, g_2, \dots, g_R$ , respectively, then `B` is a column vector of length  $q_1 * m_1 + q_2 * m_2 + \dots + q_R * m_R$ .

`B` is made by concatenating the best linear unbiased predictors of random-effects vectors corresponding to each level of each grouping variable as `[g1level1; g1level2; ...; g1levelm1; g2level1; g2level2; ...; g2levelm2; ...; gRlevel1; gRlevel2; ...; gRlevelmR]`'.

Data Types: `single` | `double`

**Dsub — Submatrix of random-effects design matrix**

matrix

Submatrix of random-effects design matrix corresponding to the grouping variables indicated by the integers in `gnumbers`, returned as an  $n$ -by- $k$  matrix, where  $k$  is length of the column vector `Bsub`.

`Bsub` contains the concatenated best linear unbiased predictors (BLUPs) of random-effects vectors, corresponding to each level of the grouping variables, specified by `gnumbers`.

If, for example, `gnumbers` is `[1, 3, r]`, this corresponds to the grouping variables `g1`, `g3`, and `gr`. Then, `Bsub` contains the concatenated BLUPs of random-effects vectors corresponding to each level of the grouping variables `g1`, `g3`, and `gr`, such as

```
[g1level1; g1level2; ...; g1levelm1; g3level1; g3level2; ...; g3levelm3;
grlevel1; grlevel2; ...; grlevelmr]'
```

Thus, `Dsub*Bsub` represents the contribution of all random effects corresponding to grouping variables `g1`, `g3`, and `gr` to the response of `lme`.

If `gnumbers` is empty, then `Dsub` is the full random-effects design matrix.

Data Types: `single` | `double`

**gnames — Names of grouping variables** $k$ -by-1 cell array

Names of grouping variables corresponding to the integers in `gnumbers` if the design type is 'Random', returned as a  $k$ -by-1 cell array. If the design type is 'Fixed', then `gnames` is an empty matrix `[]`.

Data Types: `cell`

## Examples

**Display Fixed- and Random-Effects Design Matrices**

Load the sample data.

```
load(fullfile(matlabroot, 'examples', 'stats', 'shift.mat'));
```

The data shows the deviations from the target quality characteristic measured from the products that 5 operators manufacture during three different shifts, morning, evening, and night. This is a randomized block design, where the operators are the blocks. The experiment is designed to study the impact of the time of shift on the performance. The performance measure is the deviation of the quality characteristics from the target value. This is simulated data.

Shift and Operator are nominal variables.

```
shift.Shift = nominal(shift.Shift);  
shift.Operator = nominal(shift.Operator);
```

Fit a linear mixed-effects model with a random intercept grouped by operator to assess if performance significantly differs according to the time of the shift.

```
lme = fitlme(shift, 'QCDev ~ Shift + (1|Operator)');
```

Display the fixed-effects design matrix.

```
designMatrix(lme)
```

```
ans =
```

```
1     1     0  
1     0     0  
1     0     1  
1     1     0  
1     0     0  
1     0     1  
1     1     0  
1     0     0  
1     0     1  
1     1     0  
1     0     0  
1     0     1  
1     1     0  
1     0     1  
1     1     0  
1     0     0  
1     0     1
```

The column of 1s represents the constant term in the model. `fitlme` takes the evening shift as the reference group and creates two dummy variables to represent the morning and night shifts, respectively.

Display the random-effects design matrix.

```
designMatrix(lme, 'random')
```

```
ans =
```

```
(1,1)      1
(2,1)      1
(3,1)      1
(4,2)      1
(5,2)      1
(6,2)      1
(7,3)      1
(8,3)      1
(9,3)      1
(10,4)     1
(11,4)     1
(12,4)     1
(13,5)     1
(14,5)     1
(15,5)     1
```

The first number,  $i$ , in the  $(i, |j|)$  indices corresponds to the observation number, and  $|j|$  corresponds to the level of the grouping variable, `Operator`, i.e., the operator number.

Show the full display of the random-effects design matrix.

```
full(designMatrix(lme, 'random'))
```

```
ans =
```

```
  1   0   0   0   0
  1   0   0   0   0
  1   0   0   0   0
  0   1   0   0   0
  0   1   0   0   0
  0   1   0   0   0
  0   0   1   0   0
```

```
0    0    1    0    0
0    0    1    0    0
0    0    0    1    0
0    0    0    1    0
0    0    0    1    0
0    0    0    0    1
0    0    0    0    1
0    0    0    0    1
```

Each column corresponds to a level of the grouping variable, `Operator`.

### Random-Effects Design Matrix of Multiple Grouping Variables

Load the sample data.

```
load(fullfile(matlabroot, 'examples', 'stats', 'fertilizer.mat'));
```

The dataset array includes data from a split-plot experiment, where soil is divided into three blocks based on the soil type: sandy, silty, and loamy. Each block is divided into five plots, where five different types of tomato plants (cherry, heirloom, grape, vine, and plum) are randomly assigned to these plots. The tomato plants in the plots are then divided into subplots, where each subplot is treated by one of four fertilizers. This is simulated data.

Store the data in a dataset array called `ds`, for practical purposes, and define `Tomato`, `Soil`, and `Fertilizer` as categorical variables.

```
ds = fertilizer;
ds.Tomato = nominal(ds.Tomato);
ds.Soil = nominal(ds.Soil);
ds.Fertilizer = nominal(ds.Fertilizer);
```

Fit a linear mixed-effects model, where `Fertilizer` and `Tomato` are the fixed-effects variables, and the mean yield varies by the block (soil type), and the plots within blocks (tomato types within soil types) independently.

```
lme = fitlme(ds, 'Yield ~ Fertilizer * Tomato + (1|Soil) + (1|Soil:Tomato)');
```

Store and examine the full random-effects design matrix.

```
D = full(designMatrix(lme, 'random'));
```



The first three columns of matrix `D` contain the indicator variables `fitlme` creates for the three levels (Loamy, Silty, Sandy, respectively) of the first grouping variable, `Soil`. The next 15 columns contain the indicator variables created for the second grouping variable, `Tomato` nested under `Soil`. These are basically the elementwise products of the dummy variables representing the levels of `Soil` (Loamy, Silty, and Sandy, respectively) and the levels of `Tomato` (Cherry, Grape, Heirloom, Plum, Vine, respectively).

### Subset of the Random-Effects Design Matrix

Load the sample data.

```
load(fullfile(matlabroot, 'examples', 'stats', 'fertilizer.mat'));
```

The dataset array includes data from a split-plot experiment, where soil is divided into three blocks based on the soil type: sandy, silty, and loamy. Each block is divided into five plots, where five different types of tomato plants (cherry, heirloom, grape, vine, and plum) are randomly assigned to these plots. The tomato plants in the plots are then divided into subplots, where each subplot is treated by one of four fertilizers. This is simulated data.

Store the data in a dataset array called `ds`, for practical purposes, and define `Tomato`, `Soil`, and `Fertilizer` as categorical variables.

```
ds = fertilizer;
ds.Tomato = nominal(ds.Tomato);
ds.Soil = nominal(ds.Soil);
ds.Fertilizer = nominal(ds.Fertilizer);
```

Fit a linear mixed-effects model, where `Fertilizer` and `Tomato` are the fixed-effects variables, and the mean yield varies by the block (soil type), and the plots within blocks (tomato types within soil types) independently.

```
lme = fitlme(ds, 'Yield ~ Fertilizer * Tomato + (1|Soil) + (1|Soil:Tomato)');
```

Compute the random-effects design matrix for the second grouping variable, and display the first 12 rows.

```
[Dsub, gname] = designMatrix(lme, 'random', 2);
full(Dsub(1:12, :))
```

```
ans =
```

```
Columns 1 through 13
```

```
0    0    0    0    0    0    0    0    1    0    0    0    0
0    0    0    0    0    0    0    0    1    0    0    0    0
0    0    0    0    0    0    0    0    1    0    0    0    0
0    0    0    0    0    0    0    0    1    0    0    0    0
0    0    0    0    0    1    0    0    0    0    0    0    0
0    0    0    0    0    1    0    0    0    0    0    0    0
0    0    0    0    0    1    0    0    0    0    0    0    0
0    0    0    0    0    0    0    1    0    0    0    0    0
0    0    0    0    0    0    0    1    0    0    0    0    0
0    0    0    0    0    0    0    1    0    0    0    0    0
0    0    0    0    0    0    0    1    0    0    0    0    0
```

```
Columns 14 through 15
```

```
0    0
0    0
0    0
0    0
0    0
0    0
0    0
0    0
0    0
0    0
0    0
0    0
```

Dsub contains the dummy variables created for the second grouping variable, that is, tomato nested under soil. These are the elementwise products of the dummy variables representing the levels of Soil (Loamy, Silty, Sandy, respectively) and the levels of Tomato (Cherry, Grape, Heirloom, Plum, Vine, respectively).

Display the name of the grouping variable.

```
gname
```

```
gname =
```

```
1x1 cell array  
  
{'Soil:Tomato'}
```

## See Also

[LinearMixedModel](#) | [fitlmematrix](#) | [fitted](#)

## dfittool

Open Distribution Fitter app

This page contains programmatic syntax information for the Distribution Fitter app. For general usage information, see **Distribution Fitter**.

---

**Note** The `distributionFitter` function was introduced in R2017a as a replacement for the `dfittool` function. Both functions continue to work to start the Distribution Fitter app.

---

## Syntax

```
dfittool
dfittool(y)
dfittool(y,cens)
dfittool(y,cens,freq)
dfittool(y,cens,freq,dsname)
```

## Description

`dfittool` opens the Distribution Fitter app, or brings focus to the app if it is already open.

`dfittool(y)` opens the Distribution Fitter app populated with the data specified by the vector `y`.

`dfittool(y,cens)` uses the vector `cens` to specify whether each observation in `y` is censored.

`dfittool(y,cens,freq)` uses the vector `freq` to specify the frequency of each element of `y`.

`dfittool(y,cens,freq,dsname)` creates a data set with the name `dsname`, using the data vector, `y`, censoring indicator, `cens`, and frequency vector, `freq`.

## Examples

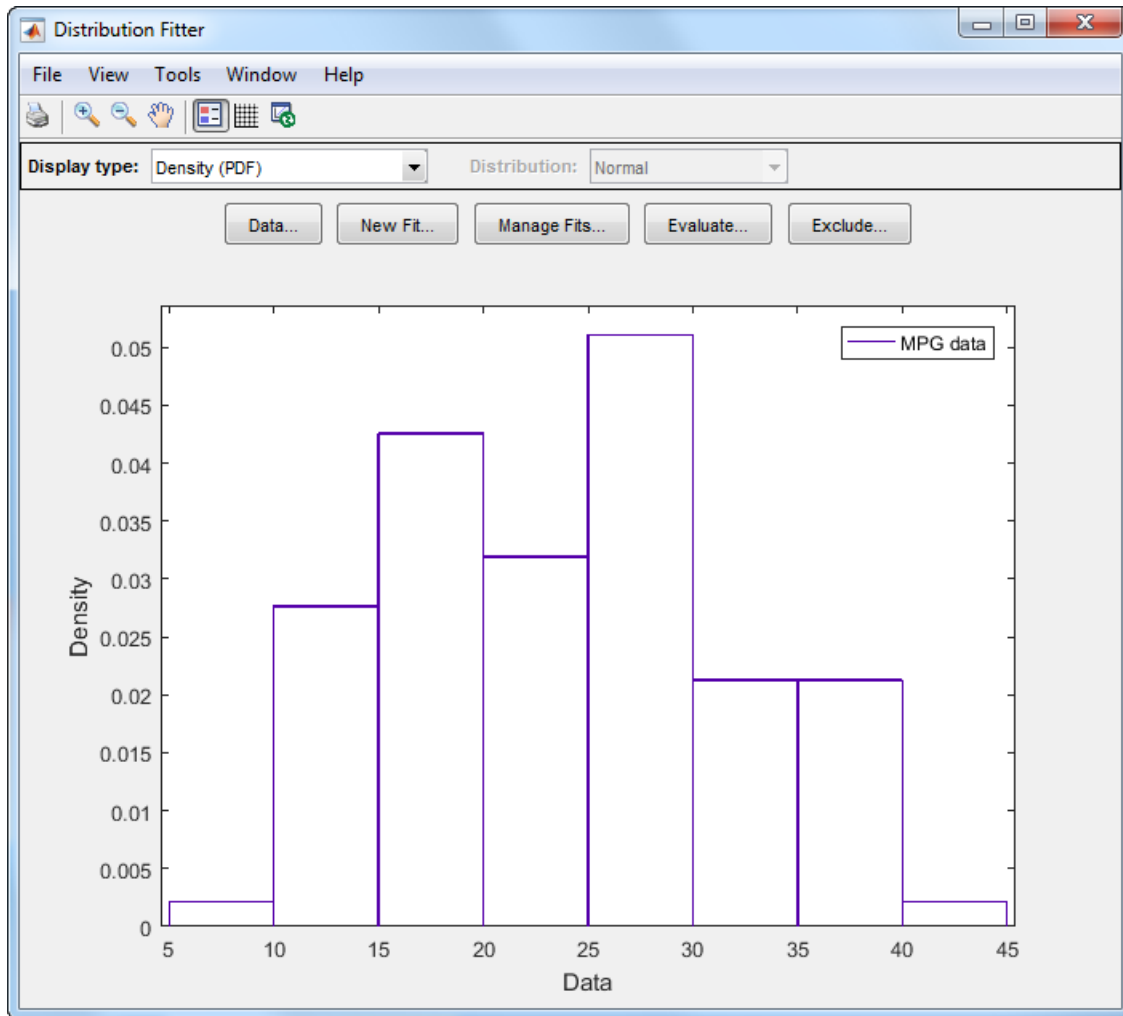
### Open Distribution Fitter App with Existing Data

Load the `carsmall` sample data.

```
load carsmall
```

Open the Distribution Fitter app using the MPG miles per gallon data.

```
dfittool(MPG)
```



The Distribution Fitter app opens, populated with the MPG data, and displays the density (PDF) plot. You can use the app to display different plots and fit distributions to this data.

## Open Distribution Fitter App with Censoring Data

Load the sample data.

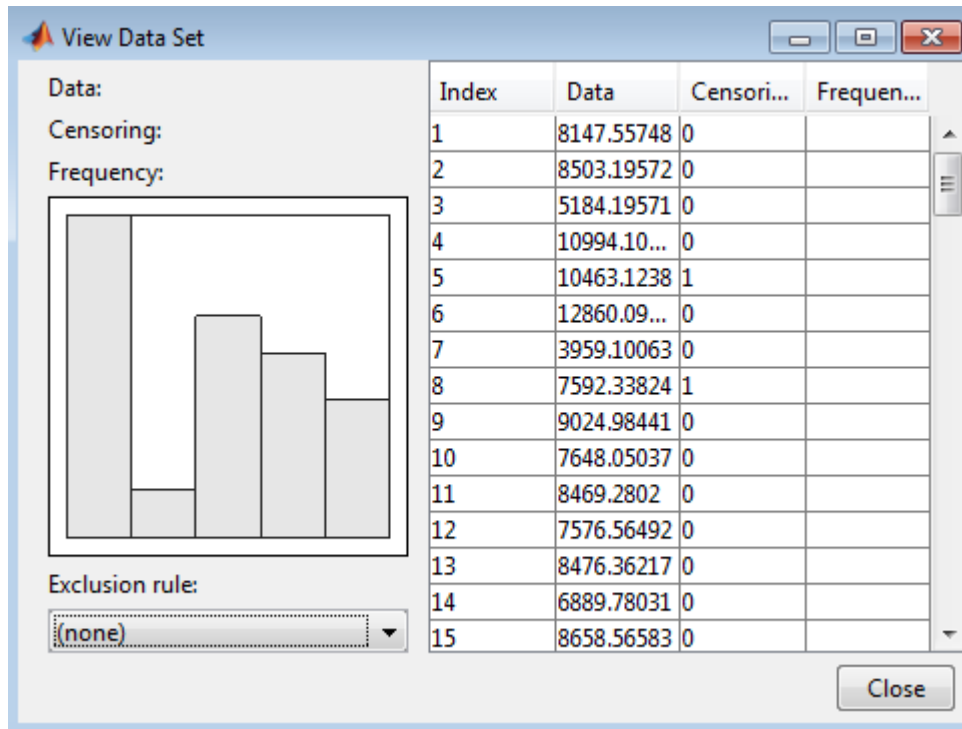
```
load(fullfile(matlabroot, 'examples', 'stats', 'lightbulb.mat'))
```

The first column of the data contains the lifetime (in hours) of two types of light bulbs. The second column contains information about the type of light bulb. 1 indicates fluorescent bulbs, and 0 indicates the incandescent bulb. The third column contains censoring information. 1 indicates censored data, and 0 indicates the exact failure time. This is simulated data.

Open the Distribution Fitter app using the first column of `lightbulb` as the input data, and the third column as the censoring data. Name the data `lifetime`.

```
dffitool(lightbulb(:,1),lightbulb(:,3),[], 'lifetime')
```

To open the Data dialog box, click **Data**. In the **Manage data sets** pane, click to highlight the `lifetime` data set row. Finally, to open the View Data Set dialog, click **View**. The lifetime data appears in the second column and the corresponding censoring indicator appears in the third column.



- “Fit a Distribution Using the Distribution Fitter App” on page 5-97

## Input Arguments

### **y** — Input data

array of scalar values | variable representing an array of scalar values

Input data, specified as an array of scalar values or a variable representing an array of such values.

Data Types: `single` | `double`

### **cens** — Censoring indicator

`zeros(n)` (default) | vector of 0 and 1 values



Censoring indicator, specified as a vector of 0 and 1 values. The length of `cens` must be equal to the length of `y`. If `y(j)` is censored, then `(cens(j)==1)`. If `y(j)` is not censored, then `(cens(j)==0)`. If `cens` is omitted or empty, then no `y` values are censored.

If you have frequency data (`freq`) but not censoring data (`cens`), then you must specify empty brackets (`[]`) for `cens`.

Data Types: `single` | `double`

### **freq** — Frequency data

`ones(n)` (default) | vector of scalar values

Frequency data, specified as a vector of scalar values. The length of `freq` must be equal to the length of `y`. If `freq` is omitted or empty, then all `y` values have a frequency of 1.

If you have frequency data (`freq`) but not censoring data (`cens`), then you must specify empty brackets (`[]`) for `cens`.

Data Types: `single` | `double`

### **dsname** — Data set name

character vector

Data set name, specified as a character vector enclosed in single quotes.

If you want to specify a data set name, but do not have censoring data (`cens`) or frequency data (`freq`), then you must specify empty brackets (`[]`) for both `freq` and `cens`.

Example: `'MyData'`

Data Types: `char`

## See Also

`distributionFitter` | `fitdist` | `makedist`

## Topics

“Fit a Distribution Using the Distribution Fitter App” on page 5-97

“Model Data Using the Distribution Fitter App” on page 5-70

“Working with Probability Distributions” on page 5-3  
“Supported Distributions” on page 5-17

**Introduced before R2006a**

## Dimensions property

**Class:** grandset

Number of dimensions

### Description

Number of dimensions in the point set. The `Dimensions` property of a point set contains a positive integer that indicates the number of dimensions for which the points have values. For example, a point set with `Dimensions=5` produces points that each have five values.

Set this property by specifying the number of dimensions when constructing a new point set. After construction, you cannot change the value. The default number of dimensions is 2.

## DimNames property

**Class:** dataset

Two-element cell array of character vectors giving names of dimensions of data set

---

**Note** The `dataset` data type might be removed in a future release. To work with heterogeneous data, use the MATLAB® `table` data type instead. See MATLAB `table` documentation for more information.

---

## Description

A two-element cell array of character vectors giving the names of the two dimensions of the data set. The default is {'Observations' 'Variables'}.

# discardSupportVectors

**Class:** CompactClassificationECOC

Discard support vectors of linear support vector machine binary learners

## Syntax

```
Mdl = discardSupportVectors(MdlSV)
```

## Description

`Mdl = discardSupportVectors(MdlSV)` returns a trained, multiclass, error-correcting output codes (ECOC) model (`Mdl`) containing at least one linear, support vector machine (SVM) binary learner. `Mdl` is similar to the trained ECOC model `MdlSV`, except:

- The `Alpha`, `SupportVectors`, and `SupportVectorLabels` properties of all linear SVM binary learners are empty (`[]`).
- If you display any linear SVM binary learners stored in the cell array of trained models `Mdl.BinaryLearners`, the software lists the `Beta` property instead of `Alpha`.

## Input Arguments

**mdlsv** — Full or compact, trained, multiclass ECOC model

ClassificationECOC model | CompactClassificationECOC model

Full or compact, trained, multiclass ECOC model containing at least one linear, SVM binary learner, specified as a `ClassificationECOC` or `CompactClassificationECOC` model.

## Output Arguments

### **Mdl** — Full or compact, trained, multiclass ECOC model

`ClassificationECOC` model | `CompactClassificationECOC` model

Full or compact, trained, multiclass ECOC model, returned as a `ClassificationECOC` or `CompactClassificationECOC` model. `Mdl` is the same type as `MdlSV`.

The properties `Alpha`, `SupportVectorLabels`, and `SupportVectors` of all linear, SVM binary learners in the cell array `Mdl.BinaryLearners` are empty. The software lists the property `Beta` when you display any linear, SVM binary learner, and does not list `Alpha`.

## Examples

### Retain and Discard Support Vectors of SVM Binary Learners

By default, `fitcecoc` empties the `Alpha`, `SupportVectorLabels`, and `SupportVectors` properties of the linear, SVM binary learners stored in the `BinaryLearners` property of the trained ECOC model. You can retain the support vectors and related values, and then discard them from the model.

Load Fisher's iris data set.

```
load fisheriris
rng(1); % For reproducibility
```

Train an ECOC model using the entire data set. Specify retaining the support vectors by passing in the appropriate SVM template.

```
t = templateSVM('SaveSupportVectors',true);
MdlSV = fitcecoc(meas,species,'Learners',t);
```

`Mdl` is a trained `ClassificationECOC` model. By default, `fitcecoc` uses linear, SVM binary learners. It implements a one-versus-one coding design, which requires three binary learners for three-class learning.

Access the estimated  $\alpha$  values using dot notation.

```
alpha = cell(3,1);
alpha{1} = Md1SV.BinaryLearners{1}.Alpha;
alpha{2} = Md1SV.BinaryLearners{2}.Alpha;
alpha{3} = Md1SV.BinaryLearners{3}.Alpha;
alpha
```

```
alpha =

    3x1 cell array

    { 3x1 double}
    { 3x1 double}
    {23x1 double}
```

alpha is a 3-by-1 cell array that stores the estimated values of  $\alpha$ .

Discard the support vectors and related values from the ECOC model.

```
Md1 = discardSupportVectors(Md1SV);
```

Md1 is similar to Md1SV, except that the Alpha, SupportVectorLabels, and SupportVectors of all linear SVM binary learners are empty ([]).

```
areAllEmpty = @(x) isempty([x.Alpha x.SupportVectors x.SupportVectorLabels]);
cellfun(areAllEmpty, Md1.BinaryLearners)
```

```
ans =

    3x1 logical array

     1
     1
     1
```

Compare the sizes of the two ECOC models.

```
vars = whos('Md1SV', 'Md1');
100*(1 - vars(1).bytes/vars(2).bytes)
```

```
ans =
```

```
5.1575
```

Mdl is about 5% smaller than MdlSV.

Reduce your memory footprint by compacting Mdl, and then clearing MdlSV and Mdl from the workspace.

```
CMdl = compact(Mdl);  
clear MdlSV Mdl;
```

Predict the label for a random row of the training data using the more efficient SVM model.

```
idx = randsample(size(meas,1),1)  
predictedLabel = predict(CMdl,meas(idx,:))  
trueLabel = species(idx)
```

```
idx =
```

```
63
```

```
predictedLabel =
```

```
1x1 cell array
```

```
{'versicolor'}
```

```
trueLabel =
```

```
1x1 cell array
```



```
{'versicolor'}
```

## Tip

For linear, SVM binary learners, and for efficiency, `fitcecoc` empties the properties `Alpha`, `SupportVectorLabels`, and `SupportVectors`. `fitcecoc` lists `Beta`, rather than `Alpha`, in the model display.

To store `Alpha`, `SupportVectorLabels`, and `SupportVectors`, pass a linear, SVM template that specifies storing support vectors to `fitcecoc`. For example, enter:

```
t = templateSVM('SaveSupportVectors','on')
Mdl = fitcecoc(X,Y,'Learners',t);
```

You can remove the support vectors and related values by passing the resulting `ClassificationECOC` model to `discardSupportVectors`.

## Algorithms

For each linear, SVM binary learner in an ECOC model, `predict` and `resubPredict` estimate SVM scores  $[f(x)]$  using

$$f(x) = \beta'x + \beta_0,$$

$\beta$  is the `Beta` property and  $\beta_0$  is the `Bias` property of the binary learners. You can access these properties for each linear, SVM binary learner in the cell array `Mdl.BinaryLearners`. For more details on the SVM score calculation, see “Support Vector Machines for Binary Classification” on page 32-2129.

## See Also

`ClassificationECOC` | `ClassificationSVM` | `CompactClassificationECOC` | `discardSupportVectors` | `fitcecoc` | `fitcsvm` | `templateSVM`

Introduced in R2015a

## discardSupportVectors

**Class:** CompactClassificationSVM

Discard support vectors for linear support vector machine models

### Syntax

```
Mdl = discardSupportVectors (MdlSV)
```

### Description

`Mdl = discardSupportVectors (MdlSV)` returns the trained, linear support vector machine (SVM) model `Mdl`, which is similar to the trained, linear SVM model `MdlSV`, except:

- The `Alpha`, `SupportVectors`, and `SupportVectorLabels` properties are empty (`[]`).
- If you display `Mdl`, the software lists the `Beta` property instead of `Alpha`.

### Input Arguments

**mdlsv** — Trained, linear SVM model

ClassificationSVM model | CompactClassificationSVM model

Trained, linear SVM model, specified as a `ClassificationSVM` or `CompactClassificationSVM` model.

If the field `MdlSV.KernelParameters.Function` is not `'linear'` (i.e., `MdlSV` is not a linear SVM model), the software returns an error.

## Output Arguments

### **Mdl** — Trained, linear SVM model

ClassificationSVM model | CompactClassificationSVM model

Trained, linear SVM model, returned as a ClassificationSVM or CompactClassificationSVM model. Mdl is the same type as MdlSV.

The properties Alpha, SupportVectorLabels, and SupportVectors of Mdl are empty. The software lists the property Beta in its display, and does not list Alpha.

## Examples

### Discard Support Vectors

To use less disk space, you can discard the support vectors and other related parameters from a trained, linear SVM.

Load the ionosphere data set.

```
load ionosphere
```

Train a linear SVM model using the entire data set.

```
MdlSV = fitcsvm(X,Y)
numSV = size(MdlSV.SupportVectors,1)
p = size(X,2)
```

```
MdlSV =
```

```
ClassificationSVM
    ResponseName: 'Y'
  CategoricalPredictors: []
        ClassNames: {'b' 'g'}
    ScoreTransform: 'none'
    NumObservations: 351
           Alpha: [103x1 double]
           Bias: -3.8828
  KernelParameters: [1x1 struct]
```

```
        BoxConstraints: [351x1 double]
        ConvergenceInfo: [1x1 struct]
        IsSupportVector: [351x1 logical]
            Solver: 'SMO'

numSV =

    103

p =

    34
```

By default, `fitcsvm` trains a linear SVM model for two-class learning. The software lists Alpha in the display. There are 103 support vectors and 34 predictors. If you discard the support vectors, the resulting model consumes less memory.

Discard the support vectors and other related parameters.

```
Mdl = discardSupportVectors(MdlSV)
Mdl.Alpha
Mdl.SupportVectors
Mdl.SupportVectorLabels

Mdl =

    ClassificationSVM
        ResponseName: 'Y'
    CategoricalPredictors: []
        ClassNames: {'b' 'g'}
        ScoreTransform: 'none'
    NumObservations: 351
        Beta: [34x1 double]
        Bias: -3.8828
    KernelParameters: [1x1 struct]
        BoxConstraints: [351x1 double]
        ConvergenceInfo: [1x1 struct]
        IsSupportVector: [351x1 logical]
        Solver: 'SMO'
```

```
ans =  
  
    []  
  
ans =  
  
    []  
  
ans =  
  
    []
```

The software lists Beta in the display instead of Alpha. The Alpha, SupportVectors, and SupportVectorLabels properties are empty.

Compare the sizes of the models.

```
vars = whos('MdlSV', 'Mdl');  
100*(1 - vars(1).bytes/vars(2).bytes)  
  
ans =  
  
    20.5951
```

Mdl is about 20% smaller than MdlSV.

Remove MdlSV from the workspace.

```
clear MdlSV
```

### Reduce Memory Consumption of SVM Models

predict accepts compacted SVM models, and, for linear SVM models, does not require the Alpha, SupportVectors, and SupportVectorLabels properties to predict labels for new observations. If your training set is large, consider compacting the SVM model, and then discarding the stored support vectors and other related estimates.

Load the ionosphere data set.

```
load ionosphere
rng(1); % For reproducibility
```

Train an SVM model using default options.

```
MdlSV = fitcsvm(X,Y);
```

MdlSV is a ClassificationSVM model containing nonempty values for its Alpha, SupportVectors, and SupportVectorLabels properties.

Reduce the size of the SVM model by discarding the training data, support vectors, and related estimates.

```
CMdlSV = compact(MdlSV); % Discard training data
CMdl = discardSupportVectors(CMdlSV); % Discard support vectors
```

CMdl is a CompactClassificationSVM model.

Compare the sizes of the SVM models MdlSV and CMdl.

```
vars = whos('MdlSV', 'CMdl');
100*(1 - vars(1).bytes/vars(2).bytes)
```

```
ans =
```

```
97.1896
```

The compacted model consumes much less memory than the full model.

Predict the label for a random row of the training data using the more efficient SVM model.

```
idx = randsample(size(X,1),1)
predictedLabel = predict(CMdl,X(idx,:))
trueLabel = Y(idx)
```

```
idx =
```

```
147
```

```

predictedLabel =
    1x1 cell array
        {'b'}

trueLabel =
    1x1 cell array
        {'b'}

```

## Tips

For a trained, linear SVM model, the `SupportVectors` property is an  $n_{sv}$ -by- $p$  matrix.  $n_{sv}$  is the number of support vectors (at most the training sample size) and  $p$  is the number of predictors or features. The `Alpha` and `SupportVectorLabels` properties are vectors with  $n_{sv}$  elements. These properties can be large for complex data sets containing many observations or examples. However, the `Beta` property is a vector with  $p$  elements.

If the trained SVM model has many support vectors, use `discardSupportVectors` to reduce the amount of disk space that the trained, linear SVM model consumes. You can display the size of the support vector matrix by entering `size(MdlSV.SupportVectors)`.

## Algorithms

`predict` and `resubPredict` estimate SVM scores  $[f(x)]$ , and subsequently labels and estimates posterior probabilities using

$$f(x) = \beta'x + \beta_0,$$

$\beta$  is `Mdl.Beta` and  $\beta_0$  is `Mdl.Bias`, that is, the `Beta` and `Bias` properties, respectively, of `Mdl`. For more details, see “Support Vector Machines for Binary Classification” on page 32-2129.

## See Also

`ClassificationECOC` | `ClassificationSVM` | `CompactClassificationSVM` | `discardSupportVectors` | `fitcsvm` | `templateSVM`

**Introduced in R2015a**



# discardSupportVectors

**Class:** CompactRegressionSVM

Discard support vectors

## Syntax

```
mdlOut = discardSupportVectors(mdl)
```

## Description

`mdlOut = discardSupportVectors(mdl)` returns the trained, linear support vector machine (SVM) regression model `mdlOut`, which is similar to the trained, linear SVM regression model `mdl`, except:

- The `Alpha` and `SupportVectors` properties are empty (`[]`).
- If you display `mdlOut`, the software lists the `Beta` property instead of the `Alpha` property.

## Input Arguments

**mdl** — Trained, linear SVM regression model

RegressionSVM model | CompactRegressionSVM model

Trained, linear SVM regression model, specified as a `RegressionSVM` or `CompactRegressionSVM` model.

If you train the model using a kernel function that is not linear (i.e., if the field `mdl.KernelFunction` is something other than `'linear'`), the software returns an error. You can only discard support vectors for linear models.

## Output Arguments

### **mdlOut** — Trained, linear SVM regression model

RegressionSVM model | CompactRegressionSVM model

Trained, linear SVM regression model, returned as a RegressionSVM or CompactRegressionSVM model. `mdlOut` is the same type as `mdl`.

After discarding the support vectors, the properties `Alpha` and `SupportVector` are empty (`[]`). The software lists the property `Beta` in its display, and does not list the property `Alpha`. The `predict` and `resubPredict` methods compute predicted responses using the coefficients stored in the `Beta` property.

## Examples

### Discard Support Vectors for SVM Regression Model

This model shows how to reduce the disk space used by a trained, linear SVM regression model by discarding the support vectors and other related parameters.

Load the `carsmall` data set. Specify `Horsepower` and `Weight` as the predictor variables (`X`), and `MPG` as the response variable (`Y`).

```
load carsmall
X = [Horsepower,Weight];
Y = MPG;
```

Train a linear SVM regression model, standardizing the data. Display the number of support vectors.

```
mdl = fitrsvm(X,Y,'Standardize',true)
numSV = size(mdl.SupportVectors,1)
```

```
mdl =
```

```
RegressionSVM
    PredictorNames: {'x1' 'x2'}
    ResponseName: 'Y'
    CategoricalPredictors: []
    ResponseTransform: 'none'
```

```

        Alpha: [77x1 double]
        Bias: 22.9131
KernelParameters: [1x1 struct]
        Mu: [109.3441 2.9625e+03]
        Sigma: [45.3545 805.9668]
NumObservations: 93
    BoxConstraints: [93x1 double]
    ConvergenceInfo: [1x1 struct]
    IsSupportVector: [93x1 logical]
    Solver: 'SMO'

```

Properties, Methods

```
numSV =
```

```
77
```

By default, `fitrsvm` trains a linear SVM regression model. The software lists Alpha in the display. The model has 77 support vectors.

Note that the predictor and response variables contain several NaN values. When training a model, `fitrsvm` will remove rows that contain NaN values from both the predictor and response data. As a result, the trained model uses only 93 of the 100 total observations contained in the sample data.

Discard the support vectors and other related parameters.

```
mdlOut = discardSupportVectors(mdl)
mdlOut.Alpha
mdlOut.SupportVectors
```

```
mdlOut =
```

```

RegressionSVM
    PredictorNames: {'x1' 'x2'}
    ResponseName: 'Y'
    CategoricalPredictors: []
    ResponseTransform: 'none'
        Beta: [2x1 double]
        Bias: 22.9131
    KernelParameters: [1x1 struct]
        Mu: [109.3441 2.9625e+03]

```

```
          Sigma: [45.3545 805.9668]
NumObservations: 93
  BoxConstraints: [93x1 double]
ConvergenceInfo: [1x1 struct]
IsSupportVector: [93x1 logical]
          Solver: 'SMO'
```

Properties, Methods

```
ans =
```

```
 []
```

```
ans =
```

```
 []
```

The software lists Beta in the display instead of Alpha. The Alpha and SupportVectors properties are empty.

Compare the sizes of the models.

```
vars = whos('mdl','mdlOut');
[vars(1).bytes,vars(2).bytes]
```

```
ans =
```

```
15004      13156
```

mdlOut consumes less memory than mdl because it does not store the support vectors.

### Reduce Memory Consumption of SVM Regression Model

This example shows how to reduce the memory consumption of a full, trained SVM regression model by compacting the model and discarding the support vectors.

Load the carsmall sample data.

```
load carsmall
rng default % for reproducibility
```

Train a linear SVM regression model using `Weight` as the predictor variable and `MPG` as the response variable. Standardize the data.

```
mdl = fitrsvm(Weight,MPG,'Standardize',true);
```

Note that `MPG` contains several `NaN` values. When training a model, `fitrsvm` will remove rows that contain `NaN` values from both the predictor and response data. As a result, the trained model uses only 94 of the 100 total observations contained in the sample data.

Compact the regression model to discard the training data and some information related to the training process.

```
compactMdl = compact(mdl);
```

`compactMdl` is a `CompactRegressionSVM` model that has the same parameters, support vectors, and related estimates as `mdl`, but no longer stores the training data.

Discard the support vectors and related estimates for the compacted model.

```
mdlOut = discardSupportVectors(compactMdl);
```

`mdlOut` is a `CompactRegressionSVM` model that has the same parameters as `mdl` and `compactMdl`, but no longer stores the support vectors and related estimates.

Compare the sizes of the three SVM regression models, `compactMdl`, `mdl`, and `mdlOut`.

```
vars = whos('compactMdl','mdl','mdlOut');
[vars(1).bytes,vars(2).bytes,vars(3).bytes]
```

```
ans =
```

```
      3601      13727      2305
```

The compacted model `compactMdl` consumes 3601 bytes of memory, while the full model `mdl` consumes 13727 bytes of memory. The model `mdlOut`, which also discards the support vectors, consumes 2305 bytes of memory.

## Tips

For a trained, linear SVM regression model, the `SupportVectors` property is an  $n_{sv}$ -by- $p$  matrix.  $n_{sv}$  is the number of support vectors (at most the training sample size) and  $p$  is

the number of predictor variables. If any of the predictors are categorical, then  $p$  includes the number of dummy variables necessary to account for all of the categorical predictor levels. The `Alpha` property is a vector with  $n_{sv}$  elements.

The `SupportVectors` and `Alpha` properties can be large for complex data sets that contain many observations or examples. However, the `Beta` property is a vector with  $p$  elements, which may be considerably smaller. You can use a trained SVM regression model to predict response values even if you discard the support vectors because the `predict` and `resubPredict` methods use `Beta` to compute the predicted responses.

If the trained, linear SVM regression model has many support vectors, use `discardSupportVectors` to reduce the amount of disk space that the trained, linear SVM regression model consumes. You can display the size of the support vector matrix by entering `size mdlIn.SupportVectors`.

## Algorithms

The `predict` and `resubPredict` estimate response values using the formula

$$f(x) = \left( \frac{X}{S} \right) \beta + \beta_0,$$

where:

- $\beta$  is the Beta value, stored as `mdl.Beta`.
- $\beta_0$  is the bias value, stored as `mdl.Bias`.
- $X$  is the training data.
- $S$  is the kernel scale value, stored as `mdl.KernelParameters.Scale`.

In this way, the software can use the value of `mdl.Beta` to make predictions even after discarding the support vectors.

## See Also

`CompactRegressionSVM` | `RegressionSVM` | `fitrsvm` | `predict` | `resubPredict`

**Introduced in R2015b**

# disp

**Class:** classregtree

Display classregtree object

---

**Note** classregtree will be removed in a future release. See fitctree, fitrtree, ClassificationTree, or RegressionTree instead.

---

## Syntax

display(t)

## Description

display(t) prints the classregtree object t.

## See Also

classregtree | view

## disp

**Class:** cvpartition

Display cvpartition object

## Syntax

`disp(c)`

## Description

`disp(c)` prints the cvpartition object `c`.

## See Also

`cvpartition`



# disp

**Class:** dataset

Display dataset array

---

**Note** The `dataset` data type might be removed in a future release. To work with heterogeneous data, use the MATLAB® `table` data type instead. See MATLAB `table` documentation for more information.

---

## Syntax

`disp(ds)`

## Description

`disp(ds)` prints the dataset array `ds`, including variable names and observation names (if present), without printing the dataset name. In all other ways it's the same as leaving the semicolon off an expression.

For numeric or categorical variables that are 2-D and have three or fewer columns, `disp` prints the actual data using either short `g`, long `g`, or bank format, depending on the current command line setting. Otherwise, `disp` prints the size and type of each dataset element.

For character variables that are 2-D and 10 or fewer characters wide, `disp` prints quoted text. Otherwise, `disp` prints the size and type of each dataset element.

For cell variables that are 2-D and have three or fewer columns, `disp` prints the contents of each cell (or its size and type if too large). Otherwise, `disp` prints the size of each dataset element.

For time series variables, `disp` prints columns for both the time and the data. If the variable is 2-D and has three or fewer columns, `disp` prints the actual data. Otherwise, `disp` prints the size and type of each dataset element.

For other types of variables, `disp` prints the size and type of each dataset element.

## See Also

`dataset` | `display` | `format`

# disp

**Class:** CompactGeneralizedLinearModel

Display generalized linear regression model

## Syntax

```
disp mdl
```

## Description

`disp(mdl)` displays the `mdl` linear model.

## Input Arguments

**mdl**

Generalized linear model, specified as a full `GeneralizedLinearModel` object constructed using `fitglm` or `stepwiseglm`, or a compacted `CompactGeneralizedLinearModel` object constructed using `compact`.

## Examples

### Display a Generalized Linear Regression Model

Create and display a generalized linear regression model.

Create a generalized linear regression model of Poisson data.

```
X = 2 + randn(100,1);  
mu = exp(1 + X/2);  
y = poissrnd(mu);
```

```
mdl = fitglm(X,y,...  
            'y ~ x1', 'distr', 'poisson');
```

Display the model.

```
disp(mdl)
```

```
Generalized linear regression model:
```

```
log(y) ~ 1 + x1  
Distribution = Poisson
```

```
Estimated Coefficients:
```

	Estimate	SE	tStat	pValue
(Intercept)	0.99516	0.080089	12.426	1.8947e-35
x1	0.50906	0.025557	19.919	2.7949e-88

```
100 observations, 98 error degrees of freedom
```

```
Dispersion: 1
```

```
Chi^2-statistic vs. constant model: 374, p-value = 2.53e-83
```

## Alternatives

Enter *mdl* at the command line to obtain a display, where *mdl* is the name of your model.

## See Also

[CompactGeneralizedLinearModel](#) | [GeneralizedLinearModel](#)

## Topics

“Generalized Linear Models” on page 12-12

# disp

**Class:** GeneralizedLinearMixedModel

Display generalized linear mixed-effects model

## Syntax

```
disp(glme)
```

## Description

`disp(glme)` displays fitted generalized linear mixed-effects model `glme`.

## Input Arguments

**glme** — Generalized linear mixed-effects model

GeneralizedLinearMixedModel object

Generalized linear mixed-effects model, specified as a GeneralizedLinearMixedModel object. For properties and methods of this object, see GeneralizedLinearMixedModel.

## Examples

### Display a Generalized Linear Mixed-Effects Model

Load the sample data.

```
load mfr
```

This simulated data is from a manufacturing company that operates 50 factories across the world, with each factory running a batch process to create a finished product. The company wants to decrease the number of defects in each batch, so it developed a new

manufacturing process. To test the effectiveness of the new process, the company selected 20 of its factories at random to participate in an experiment: Ten factories implemented the new process, while the other ten continued to run the old process. In each of the 20 factories, the company ran five batches (for a total of 100 batches) and recorded the following data:

- Flag to indicate whether the batch used the new process (`newprocess`)
- Processing time for each batch, in hours (`time`)
- Temperature of the batch, in degrees Celsius (`temp`)
- Categorical variable indicating the supplier of the chemical used in the batch (`supplier`)
- Number of defects in the batch (`defects`)

The data also includes `time_dev` and `temp_dev`, which represent the absolute deviation of time and temperature, respectively, from the process standard of 3 hours at 20 degrees Celsius.

Fit a generalized linear mixed-effects model using `newprocess`, `time_dev`, `temp_dev`, and `supplier` as fixed-effects predictors. Include a random-effects term for intercept grouped by `factory`, to account for quality differences that might exist due to factory-specific variations. The response variable `defects` has a Poisson distribution, and the appropriate link function for this model is `log`. Use the Laplace fit method to estimate the coefficients. Specify the dummy variable encoding as `'effects'`, so the dummy variable coefficients sum to 0.

The number of defects can be modeled using a Poisson distribution  
 $\text{defects}_{ij} \sim \text{Poisson}(\mu_{ij})$

This corresponds to the generalized linear mixed-effects model

$$\log(\mu_{ij}) = \beta_0 + \beta_1 \text{newprocess}_{ij} + \beta_2 \text{time.dev}_{ij} + \beta_3 \text{temp.dev}_{ij} + \beta_4 \text{supplier.C}_{ij} + \beta_5 \text{supplier.B}_{ij} + b_i,$$

where

- $\text{defects}_{ij}$  is the number of defects observed in the batch produced by factory  $i$  during batch  $j$ .
- $\mu_{ij}$  is the mean number of defects corresponding to factory  $i$  (where  $i = 1, 2, \dots, 20$ ) during batch  $j$  (where  $j = 1, 2, \dots, 5$ ).

- `newprocessij`, `time_devij`, and `temp_devij` are the measurements for each variable that correspond to factory *i* during batch *j*. For example, `newprocessij` indicates whether the batch produced by factory *i* during batch *j* used the new process.
- `supplier_Cij` and `supplier_Bij` are dummy variables that use effects (sum-to-zero) coding to indicate whether company C or B, respectively, supplied the process chemicals for the batch produced by factory *i* during batch *j*.
- $b_i \sim N(0, \sigma_b^2)$  is a random-effects intercept for each factory *i* that accounts for factory-specific variation in quality.

```
glme = fitglme(mfr, 'defects ~ 1 + newprocess + time_dev + temp_dev + supplier + (1|fact
```

Display the model.

```
disp(glme)
```

```
Generalized linear mixed-effects model fit by ML
```

```
Model information:
```

Number of observations	100
Fixed effects coefficients	6
Random effects coefficients	20
Covariance parameters	1
Distribution	Poisson
Link	Log
FitMethod	Laplace

```
Formula:
```

```
defects ~ 1 + newprocess + time_dev + temp_dev + supplier + (1 | factory)
```

```
Model fit statistics:
```

AIC	BIC	LogLikelihood	Deviance
416.35	434.58	-201.17	402.35

```
Fixed effects coefficients (95% CIs):
```

Name	Estimate	SE	tStat	DF	pValue
'(Intercept)'	1.4689	0.15988	9.1875	94	9.8194e-15
'newprocess'	-0.36766	0.17755	-2.0708	94	0.041122
'time_dev'	-0.094521	0.82849	-0.11409	94	0.90941
'temp_dev'	-0.28317	0.9617	-0.29444	94	0.76907
'supplier_C'	-0.071868	0.078024	-0.9211	94	0.35936
'supplier_B'	0.071072	0.07739	0.91836	94	0.36078

Lower	Upper
1.1515	1.7864
-0.72019	-0.015134
-1.7395	1.5505
-2.1926	1.6263
-0.22679	0.083051
-0.082588	0.22473

Random effects covariance parameters:

Group: factory (20 Levels)

Name1	Name2	Type	Estimate
'(Intercept)'	'(Intercept)'	'std'	0.31381

Group: Error

Name	Estimate
'sqrt(Dispersion)'	1

The `Model` information table displays the total number of observations in the sample data (100), the number of fixed- and random-effects coefficients (6 and 20, respectively), and the number of covariance parameters (1). It also indicates that the response variable has a Poisson distribution, the link function is Log, and the fit method is Laplace.

Formula indicates the model specification using Wilkinson's notation.

The `Model fit statistics` table displays statistics used to assess the goodness of fit of the model. This includes the Akaike information criterion (AIC), Bayesian information criterion (BIC) values, log likelihood (`LogLikelihood`), and deviance (`Deviance`) values.

The `Fixed effects coefficients` table indicates that `fitglm` returned 95% confidence intervals. It contains one row for each fixed-effects predictor, and each column contains statistics corresponding to that predictor. Column 1 (`Name`) contains the name of each fixed-effects coefficient, column 2 (`Estimate`) contains its estimated value, and column 3 (`SE`) contains the standard error of the coefficient. Column 4 (`tStat`) contains the *t*-statistic for a hypothesis test that the coefficient is equal to 0. Column 5 (`DF`) and column 6 (`pValue`) contain the degrees of freedom and *P*-value that correspond to the *t*-statistic, respectively. The last two columns (`Lower` and `Upper`) display the lower and upper limits, respectively, of the 95% confidence interval for each fixed-effects coefficient.



Random effects covariance parameters displays a table for each grouping variable (here, only `factory`), including its total number of levels (20), and the type and estimate of the covariance parameter. Here, `std` indicates that `fitglme` returns the standard deviation of the random effect associated with the `factory` predictor, which has an estimated value of 0.31381. It also displays a table containing the error parameter type (here, the square root of the dispersion parameter), and its estimated value of 1.

The standard display generated by `fitglme` does not provide confidence intervals for the random-effects parameters. To compute and display these values, use `covarianceParameters`.

## Definitions

### Akaike and Bayesian Information Criteria

The *Akaike information criterion* (AIC) is  $AIC = -2\log L_M + 2(param)$ .

$\log L_M$  depends on the method used to fit the model.

- If you use 'Laplace' or 'ApproximateLaplace', then  $\log L_M$  is the maximized log likelihood.
- If you use 'MPL', then  $\log L_M$  is the maximized log likelihood of the pseudo data from the final pseudo likelihood iteration.
- If you use 'REMPL', then  $\log L_M$  is the maximized restricted log likelihood of the pseudo data from the final pseudo likelihood iteration.

$param$  is the total number of parameters estimated in the model. For most GLME models,  $param$  is equal to  $nc + p + 1$ , where  $nc$  is the total number of parameters in the random-effects covariance, excluding the residual variance, and  $p$  is the number of fixed-effects coefficients. However, if the dispersion parameter is fixed at 1.0 for binomial or Poisson distributions, then  $param$  is equal to  $(nc + p)$ .

The *Bayesian information criterion* (BIC) is  $BIC = -2*\log L_M + \ln(n_{eff})(param)$ .

$\log L_M$  depends on the method used to fit the model.

- If you use 'Laplace' or 'ApproximateLaplace', then  $\log L_M$  is the maximized log likelihood.

- If you use 'MPL', then  $\log L_M$  is the maximized log likelihood of the pseudo data from the final pseudo likelihood iteration.
- If you use 'REML', then  $\log L_M$  is the maximized restricted log likelihood of the pseudo data from the final pseudo likelihood iteration.

$n_{eff}$  is the effective number of observations.

- If you use 'MPL', 'Laplace', or 'ApproximateLaplace', then  $n_{eff} = n$ , where  $n$  is the number of observations.
- If you use 'REML', then  $n_{eff} = n - p$ .

$param$  is the total number of parameters estimated in the model. For most GLME models,  $param$  is equal to  $nc + p + 1$ , where  $nc$  is the total number of parameters in the random-effects covariance, excluding the residual variance, and  $p$  is the number of fixed-effects coefficients. However, if the dispersion parameter is fixed at 1.0 for binomial or Poisson distributions, then  $param$  is equal to  $(nc + p)$ .

A lower value of deviance indicates a better fit. As the value of deviance decreases, both AIC and BIC tend to decrease. Both AIC and BIC also include penalty terms based on the number of parameters estimated,  $p$ . So, when the number of parameters increase, the values of AIC and BIC tend to increase as well. When comparing different models, the model with the lowest AIC or BIC value is considered as the best fitting model.

For models fitted using 'MPL' and 'REML', AIC and BIC are based on the log likelihood (or restricted log likelihood) of pseudo data from the final pseudo likelihood iteration. Therefore, a direct comparison of AIC and BIC values between models fitted using 'MPL' and 'REML' is not appropriate.

## See Also

`GeneralizedLinearMixedModel` | `covarianceParameters` | `fitglme`

# disp

**Class:** `gmdistribution`

Display Gaussian mixture distribution object

## Syntax

```
disp(obj)
```

## Description

`disp(obj)` prints a text representation of the `gmdistribution` object, `obj`, without printing the object name. In all other ways it's the same as leaving the semicolon off an expression.

## See Also

`display` | `gmdistribution`

## disp

**Class:** CompactLinearModel

Display linear regression model

## Syntax

```
display mdl
```

## Description

`display(mdl)` displays the `mdl` linear model.

## Input Arguments

**mdl** — Linear model object

LinearModel object | CompactLinearModel object

Linear model object, specified as a full `LinearModel` object constructed using `fitlm` or `stepwiselm`, or a compacted `CompactLinearModel` object constructed using `compact`.

## Examples

### Display a Linear Regression Model

Create and display a linear regression model.

Create a linear regression model.

```
X = randn(100,5);  
y = X*[1;2;3;4;5] + 6 + randn(100,1);  
mdl = fitlm(X,y);
```

Display the model.

```
disp mdl)
```

Linear regression model:

$$y \sim 1 + x1 + x2 + x3 + x4 + x5$$

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	6.0382	0.099458	60.71	3.2792e-77
x1	0.92794	0.087307	10.628	8.5494e-18
x2	1.9244	0.10044	19.16	3.1678e-34
x3	2.8965	0.099879	29	1.1117e-48
x4	4.0453	0.10832	37.346	3.4862e-58
x5	5.0029	0.11799	42.401	4.452e-63

Number of observations: 100, Error degrees of freedom: 94

Root Mean Squared Error: 0.972

R-squared: 0.976, Adjusted R-Squared 0.975

F-statistic vs. constant model: 778, p-value = 8e-75

## Alternatives

Type the name of your model (for example, mdl) at the command line to obtain a display.

## See Also

CompactLinearModel | LinearModel

## Topics

“Linear Regression” on page 11-11

“Stepwise Regression” on page 11-127

“Robust Regression — Reduce Outlier Effects” on page 11-133

## disp

**Class:** LinearMixedModel

Display linear mixed-effects model

## Syntax

```
display(lme)
```

## Description

`display(lme)` displays the fitted linear mixed-effects model `lme`.

## Input Arguments

**lme** — Linear mixed-effects model

LinearMixedModel object

Linear mixed-effects model, specified as a LinearMixedModel object constructed using `fitlme` or `fitlmematrix`.

## Examples

### Randomized Block Design

Load the sample data.

```
load(fullfile(matlabroot, 'examples', 'stats', 'shift.mat'));
```

The dataset array shows the absolute deviations from the target quality characteristic measured from the products that five operators manufacture during three shifts, morning, evening, and night. This is a randomized block design, where the operators are

the blocks. The experiment is designed to study the impact of the time of shift on the performance. The performance measure is the absolute deviation of the quality characteristics from the target value. This is simulated data.

Shift and Operator are nominal variables.

```
shift.Shift = nominal(shift.Shift);
shift.Operator = nominal(shift.Operator);
```

Fit a linear mixed-effects model with a random intercept grouped by operator to assess if performance significantly differs according to the time of the shift.

```
lme = fitlme(shift, 'QCDev ~ Shift + (1|Operator)');
```

Display the model.

```
disp(lme)
```

Linear mixed-effects model fit by ML

Model information:

Number of observations	15
Fixed effects coefficients	3
Random effects coefficients	5
Covariance parameters	2

Formula:

```
QCDev ~ 1 + Shift + (1 | Operator)
```

Model fit statistics:

AIC	BIC	LogLikelihood	Deviance
59.012	62.552	-24.506	49.012

Fixed effects coefficients (95% CIs):

Name	Estimate	SE	tStat	DF	pValue
'(Intercept)'	3.1196	0.88681	3.5178	12	0.0042407
'Shift_Morning'	-0.3868	0.48344	-0.80009	12	0.43921
'Shift_Night'	1.9856	0.48344	4.1072	12	0.0014535

Lower	Upper
1.1874	5.0518
-1.4401	0.66653
0.93227	3.0389

```
Random effects covariance parameters (95% CIs):
Group: Operator (5 Levels)
  Name1          Name2          Type          Estimate    Lower
  '(Intercept)'  '(Intercept)'  'std'        1.8297      0.94915

Upper
3.5272

Group: Error
  Name          Estimate    Lower    Upper
  'Res Std'    0.76439    0.49315  1.1848
```

This display includes the model performance statistics, “Akaike and Bayesian Information Criteria” on page 32-1442, “Akaike and Bayesian Information Criteria” on page 32-1442, loglikelihood, and “Deviance” on page 32-1443.

The fixed-effects coefficients table includes the names and estimates of the coefficients in the first two columns. The third column SE shows the standard errors of the coefficients. The column `tStat` includes the *t*-statistic values that correspond to each coefficient. `DF` is the residual degrees of freedom, and the `pValue` is the *P*-value that corresponds to the corresponding *t*-statistic value. The columns `Lower` and `Upper` display the lower and upper limits of a 95% confidence interval for each fixed-effects coefficient.

The first table for the random effects shows the types and the estimates of the random effects covariance parameters, with the lower and upper limits of a 95% confidence interval for each parameter. The display also shows the name of the grouping variable, operator, and the total number of levels, 5.

The second table for the random effects shows the estimate of the observation error, with the lower and upper limits of a 95% confidence interval.

## Definitions

### Akaike and Bayesian Information Criteria

Akaike information criterion (AIC) is  $AIC = -2 \cdot \log L_M + 2 \cdot (nc + p + 1)$ , where  $\log L_M$  is the maximized log likelihood (or maximized restricted log likelihood) of the model, and  $nc + p$



+ 1 is the number of parameters estimated in the model.  $p$  is the number of fixed-effects coefficients, and  $nc$  is the total number of parameters in the random-effects covariance excluding the residual variance.

Bayesian information criterion (BIC) is  $BIC = -2 \cdot \log L_M + \ln(n_{eff}) \cdot (nc + p + 1)$ , where  $\log L_M$  is the maximized log likelihood (or maximized restricted log likelihood) of the model,  $n_{eff}$  is the effective number of observations, and  $(nc + p + 1)$  is the number of parameters estimated in the model.

- If the fitting method is maximum likelihood (ML), then  $n_{eff} = n$ , where  $n$  is the number of observations.
- If the fitting method is restricted maximum likelihood (REML), then  $n_{eff} = n - p$ .

A lower value of deviance indicates a better fit. As the value of deviance decreases, both AIC and BIC tend to decrease. Both AIC and BIC also include penalty terms based on the number of parameters estimated,  $p$ . So, when the number of parameters increase, the values of AIC and BIC tend to increase as well. When comparing different models, the model with the lowest AIC or BIC value is considered as the best fitting model.

## Deviance

`LinearMixedModel` computes the deviance of model  $M$  as minus two times the loglikelihood of that model. Let  $L_M$  denote the maximum value of the likelihood function for model  $M$ . Then, the deviance of model  $M$  is

$$-2 \cdot \log L_M.$$

A lower value of deviance indicates a better fit. Suppose  $M_1$  and  $M_2$  are two different models, where  $M_1$  is nested in  $M_2$ . Then, the fit of the models can be assessed by comparing the deviances  $Dev_1$  and  $Dev_2$  of these models. The difference of the deviances is

$$Dev = Dev_1 - Dev_2 = 2(\log LM_2 - \log LM_1).$$

Usually, the asymptotic distribution of this difference has a chi-square distribution with degrees of freedom  $\nu$  equal to the number of parameters that are estimated in one model but fixed (typically at 0) in the other. That is, it is equal to the difference in the number of parameters estimated in  $M_1$  and  $M_2$ . You can get the  $p$ -value for this test using  $1 - \text{chi2cdf}(Dev, \nu)$ , where  $Dev = Dev_2 - Dev_1$ .

However, in mixed-effects models, when some variance components fall on the boundary of the parameter space, the asymptotic distribution of this difference is more complicated. For example, consider the hypotheses

$H_0: D = \begin{pmatrix} D_{11} & 0 \\ 0 & 0 \end{pmatrix}$ ,  $D$  is a  $q$ -by- $q$  symmetric positive semidefinite matrix.

$H_1: D$  is a  $(q+1)$ -by- $(q+1)$  symmetric positive semidefinite matrix.

That is,  $H_1$  states that the last row and column of  $D$  are different from zero. Here, the bigger model  $M_2$  has  $q + 1$  parameters and the smaller model  $M_1$  has  $q$  parameters. And  $Dev$  has a 50:50 mixture of  $\chi^2_q$  and  $\chi^2_{(q+1)}$  distributions (Stram and Lee, 1994).

## References

[1] Hox, J. *Multilevel Analysis, Techniques and Applications*. Lawrence Erlbaum Associates, Inc., 2002.

[2] Stram D. O. and J. W. Lee. “Variance components testing in the longitudinal mixed-effects model”. *Biometrics*, Vol. 50, 4, 1994, pp. 1171–1177.

## See Also

`LinearMixedModel` | `fitlme` | `fitlmematrix`

# disp

**Class:** NaiveBayes

Display NaiveBayes classifier object

## Syntax

```
disp(nb)
```

## Description

`disp(nb)` prints a text representation of the `NaiveBayes` object `nb`, without printing the object name. In all other ways it's the same as leaving the semicolon off an expression.

## See Also

`NaiveBayes` | `display`

## disp

**Class:** NonLinearModel

Display nonlinear regression model

## Syntax

```
disp mdl
```

## Description

`disp(mdl)` displays the `mdl` nonlinear model at the command line.

## Input Arguments

**mdl**

Nonlinear regression model, constructed by `fitnlm`.

## Examples

### Display a Nonlinear Regression Model

Create and display a nonlinear regression model.

Load the `reaction` data, and specify both a model function and starting values for the iterations.

```
load reaction
modelfun = 'rate~(b1*x2-x3/b5)/(1+b2*x1+b3*x2+b4*x3)';
beta0 = [1 .05 .02 .1 2];
```

Create a model of the data.

```
mdl = fitnlm(reactants,rate,modelfun,beta0);
```

Display the model.

```
disp(mdl)
```

Nonlinear regression model:

$$\text{rate} \sim (b1*x2 - x3/b5) / (1 + b2*x1 + b3*x2 + b4*x3)$$

Estimated Coefficients:

	Estimate	SE	tStat	pValue
b1	1.2526	0.86701	1.4447	0.18654
b2	0.062776	0.043561	1.4411	0.18753
b3	0.040048	0.030885	1.2967	0.23089
b4	0.11242	0.075157	1.4957	0.17309
b5	1.1914	0.83671	1.4239	0.1923

Number of observations: 13, Error degrees of freedom: 8

Root Mean Squared Error: 0.193

R-Squared: 0.999, Adjusted R-Squared 0.998

F-statistic vs. zero model: 3.91e+03, p-value = 2.54e-13

## Alternatives

Enter *mdl* at the command line to obtain a display, where *mdl* is the name of your model.

## See Also

NonLinearModel

## Topics

“Nonlinear Regression” on page 13-2

## disp

**Class:** `piecwisedistribution`

Display `piecwisedistribution` object

## Syntax

```
disp(A)
```

## Description

`disp(A)` prints a text representation of the `piecwisedistribution` object `A`, without printing the object name. In all other ways it's the same as leaving the semicolon off an expression.

## See Also

`piecwisedistribution`

# disp

**Class:** grandset

Display grandset object

## Syntax

```
disp(p)
```

## Description

`disp(p)` displays the properties of the quasi-random point set `s`, without printing the variable name. `disp` prints out the number of dimensions and points in the point-set, and follows this with the list of all property values for the object.

## See Also

`grandset`

## disp

**Class:** grandstream

Display grandstream object

## Syntax

```
disp(q)
```

## Description

`disp(q)` displays the quasi-random stream `q`, without printing the variable name. `disp` prints the type and number of dimensions in the stream, and follows it with the list of point set properties.

## See Also

grandstream



# display

**Class:** `classregtree`

Display `classregtree` object

---

**Note** `classregtree` will be removed in a future release. See `fitctree`, `fitrtree`, `ClassificationTree`, or `RegressionTree` instead.

---

## Syntax

```
display(t)
display(A)
```

## Description

`display(t)` prints the `classregtree` object `t`. `classregtree` calls `display` when a you do not use a semicolon to terminate a statement.

`display(A)` prints the categorical array `A`. `categorical` calls `display` when a you do not use a semicolon to terminate a statement.

## See Also

`classregtree` | `eval` | `prune` | `test`

## display

**Class:** `cvpartition`

Display `cvpartition` object

## Syntax

`display(c)`

## Description

`display(c)` prints the `cvpartition` object `c`. `cvpartition` calls `display` when a you do not use a semicolon to terminate a statement.

## See Also

`cvpartition`

# display

**Class:** dataset

Display dataset array

---

**Note** The `dataset` data type might be removed in a future release. To work with heterogeneous data, use the MATLAB® `table` data type instead. See MATLAB `table` documentation for more information.

---

## Syntax

`display(ds)`

## Description

`display(ds)` prints the dataset array `ds`, including variable names and observation names (if present). `dataset` calls `display` when you do not use a semicolon to terminate a statement

For numeric or categorical variables that are 2-D and have three or fewer columns, `display` prints the actual data. Otherwise, `display` prints the size and type of each dataset element.

For character variables that are 2-D and 10 or fewer characters wide, `display` prints quoted text. Otherwise, `display` prints the size and type of each dataset element.

For cell variables that are 2-D and have three or fewer columns, `display` prints the contents of each cell (or its size and type if too large). Otherwise, `display` prints the size of each dataset element.

For time series variables, `display` prints columns for both the time and the data. If the variable is 2-D and has three or fewer columns, `display` prints the actual data. Otherwise, `display` prints the size and type of each dataset element.

For other types of variables, `display` prints the size and type of each dataset element.

## See Also

`dataset` | `display` | `format`

# display

**Class:** `gmdistribution`

Display Gaussian mixture distribution object

## Syntax

```
display(obj)
```

## Description

`display(obj)` prints a text representation of the `gmdistribution` object `obj`. `gmdistribution` calls `display` when you do not use a semicolon to terminate a statement.

## See Also

`disp` | `gmdistribution`

## display

**Class:** NaiveBayes

Display NaiveBayes classifier object

## Syntax

```
display(nb)
```

## Description

`display(nb)` prints a text representation of the NaiveBayes object `nb`. NaiveBayes calls `display` when a you do not use a semicolon to terminate a statement.

## See Also

NaiveBayes | display

# display

**Class:** `piecwisedistribution`

Display `piecwisedistribution` object

## Syntax

```
display(A)
```

## Description

`display(A)` prints a text representation of the `piecwisedistribution` object `A`, without printing the object name. `piecwisedistribution` calls `display` when a you do not use a semicolon to terminate a statement.

## See Also

`piecwisedistribution`

## DistName property

**Class:** ProbDist

Read-only probability distribution name of ProbDist object

### Description

`DistName` is a read-only property of the `ProbDist` class. `DistName` is a character vector containing the type of distribution used to create the object.

### Values

Possible values are:

- 'kernel'
- 'beta'
- 'binomial'
- 'birnbaumsaunders'
- 'exponential'
- 'extreme value'
- 'gamma'
- 'generalized extreme value'
- 'generalized pareto'
- 'inversegaussian'
- 'logistic'
- 'loglogistic'
- 'lognormal'
- 'nakagami'
- 'negative binomial'
- 'normal'



- 'poisson'
- 'rayleigh'
- 'rician'
- 'tlocationscale'
- 'weibull'

Use this information to view and compare the type of distribution used to create distribution objects.

## Dist property

**Class:** NaiveBayes

Distribution names

### Description

The `Dist` property is a character vector or a 1-by-`NDims` cell array of character vectors indicating the types of distributions for all the features. If all the features use the same type of distribution, `Dist` is a single character vector. Otherwise `Dist(j)` indicates the distribution type used for the `j`th feature.

The valid character vectors for this property are the following:

'normal'	Normal distribution.
'kernel'	Kernel smoothing density estimate.
'mvnm'	Multivariate multinomial distribution.
'mn'	Multinomial bag-of-tokens model.

# distributionFitter

Open Distribution Fitter app

This page contains programmatic syntax information for the Distribution Fitter app. For general usage information, see **Distribution Fitter**.

## Syntax

```
distributionFitter
distributionFitter(y)
distributionFitter(y, cens)
distributionFitter(y, cens, freq)
distributionFitter(y, cens, freq, dsname)
```

## Description

`distributionFitter` opens the Distribution Fitter app, or brings focus to the app if it is already open.

`distributionFitter(y)` opens the Distribution Fitter app populated with the data specified by the vector `y`.

`distributionFitter(y, cens)` uses the vector `cens` to specify whether each observation in `y` is censored.

`distributionFitter(y, cens, freq)` uses the vector `freq` to specify the frequency of each element of `y`.

`distributionFitter(y, cens, freq, dsname)` creates a data set with the name `dsname`, using the data vector, `y`, censoring indicator, `cens`, and frequency vector, `freq`.

## Examples

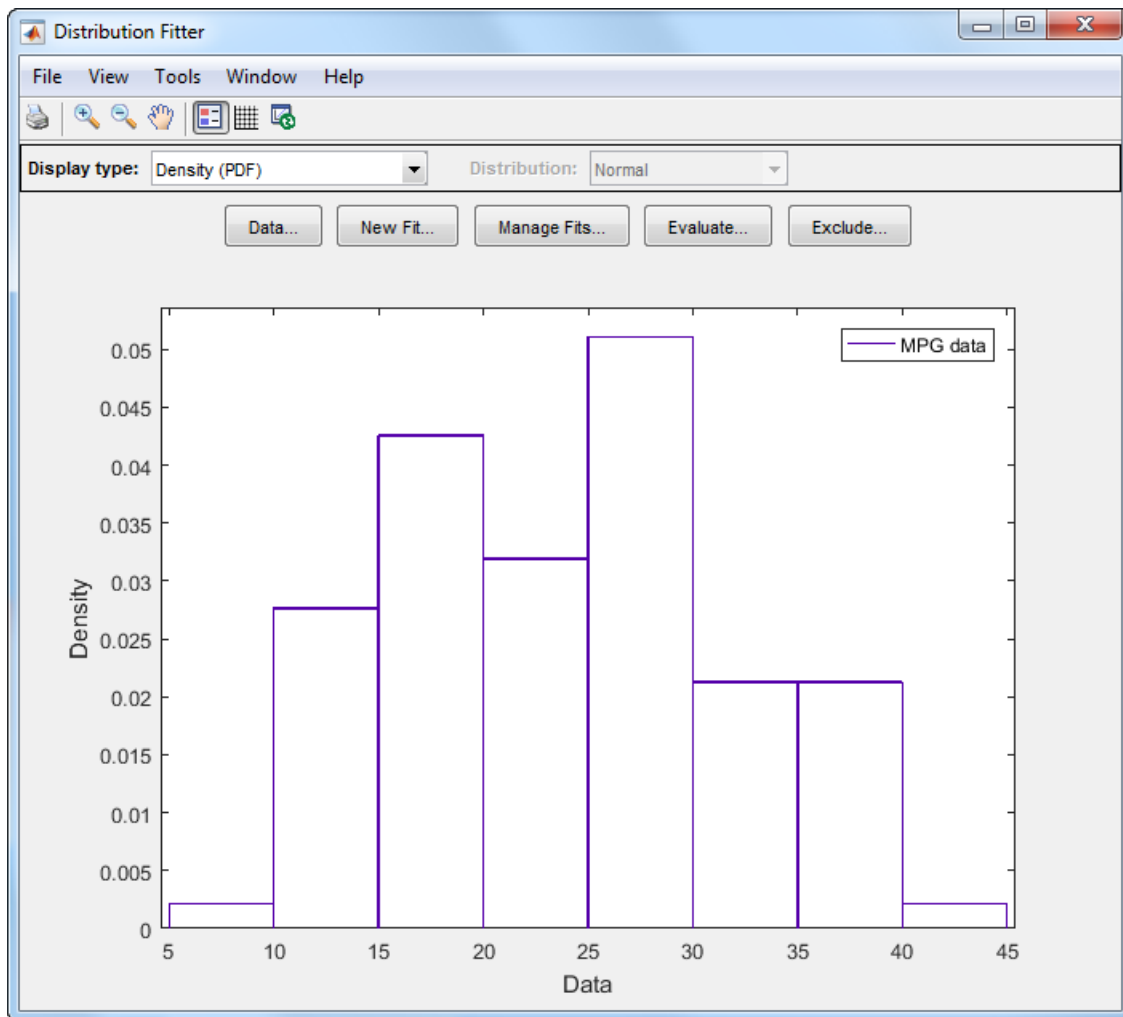
### Open Distribution Fitter App with Existing Data

Load the `carsmall` sample data.

```
load carsmall
```

Open the Distribution Fitter app using the MPG miles per gallon data.

```
distributionFitter(MPG)
```



The Distribution Fitter app opens, populated with the MPG data, and displays the density (PDF) plot. You can use the app to display different plots and fit distributions to this data.

### Open Distribution Fitter App with Censoring Data

Load the sample data.

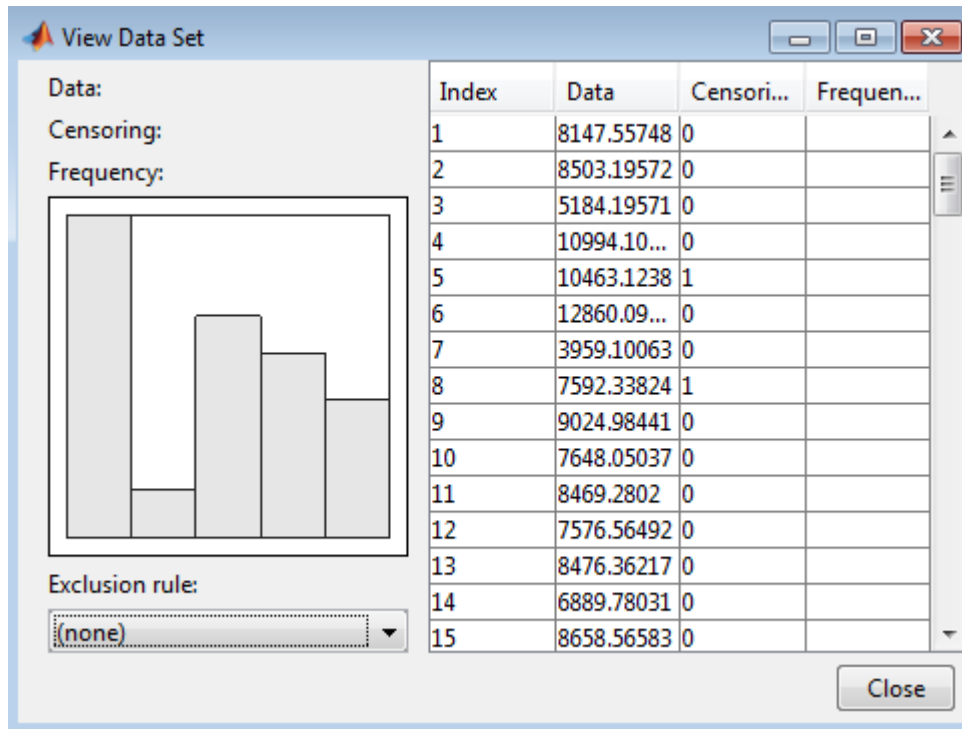
```
load(fullfile(matlabroot, 'examples', 'stats', 'lightbulb.mat'))
```

The first column of the data contains the lifetime (in hours) of two types of light bulbs. The second column contains information about the type of light bulb. 1 indicates fluorescent bulbs, and 0 indicates the incandescent bulb. The third column contains censoring information. 1 indicates censored data, and 0 indicates the exact failure time. This is simulated data.

Open the Distribution Fitter app using the first column of `lightbulb` as the input data, and the third column as the censoring data. Name the data `lifetime`.

```
distributionFitter(lightbulb(:,1),lightbulb(:,3), [], 'lifetime')
```

To open the Data dialog box, click **Data**. In the **Manage data sets** pane, click to highlight the `lifetime` data set row. Finally, to open the View Data Set dialog, click **View**. The lifetime data appears in the second column and the corresponding censoring indicator appears in the third column.



- “Fit a Distribution Using the Distribution Fitter App” on page 5-97

## Input Arguments

### **y** — Input data

array of scalar values | variable representing an array of scalar values

Input data, specified as an array of scalar values or a variable representing an array of such values.

Data Types: `single` | `double`

### **cens** — Censoring indicator

`zeros(n)` (default) | vector of 0 and 1 values

Censoring indicator, specified as a vector of 0 and 1 values. The length of `cens` must be equal to the length of `y`. If `y(j)` is censored, then `(cens(j)==1)`. If `y(j)` is not censored, then `(cens(j)==0)`. If `cens` is omitted or empty, then no `y` values are censored.

If you have frequency data (`freq`) but not censoring data (`cens`), then you must specify empty brackets (`[]`) for `cens`.

Data Types: `single` | `double`

### **freq** — Frequency data

`ones(n)` (default) | vector of scalar values

Frequency data, specified as a vector of scalar values. The length of `freq` must be equal to the length of `y`. If `freq` is omitted or empty, then all `y` values have a frequency of 1.

If you have frequency data (`freq`) but not censoring data (`cens`), then you must specify empty brackets (`[]`) for `cens`.

Data Types: `single` | `double`

### **dsname** — Data set name

character vector

Data set name, specified as a character vector enclosed in single quotes.

If you want to specify a data set name, but do not have censoring data (`cens`) or frequency data (`freq`), then you must specify empty brackets (`[]`) for both `freq` and `cens`.

Example: `'MyData'`

Data Types: `char`

## See Also

`fitdist` | `makedist`

## Topics

“Fit a Distribution Using the Distribution Fitter App” on page 5-97

“Model Data Using the Distribution Fitter App” on page 5-70

“Working with Probability Distributions” on page 5-3  
“Supported Distributions” on page 5-17

**Introduced in R2017a**



## DistributionName property

**Class:** gmdistribution

Type of distribution

### Description

The character vector 'gaussian mixture distribution'.

# Probability Distribution Function

Interactive density and distribution plots

## Description

The Probability Distribution Function user interface creates an interactive plot of the cumulative distribution function (cdf) or probability density function (pdf) for a probability distribution. Explore the effects of changing parameter values on the shape of the plot, either by specifying parameter values or using interactive sliders.

## Required Products

- MATLAB
- Statistics and Machine Learning Toolbox

## Open the Probability Distribution Function App

- At the command prompt, enter `disttool`.

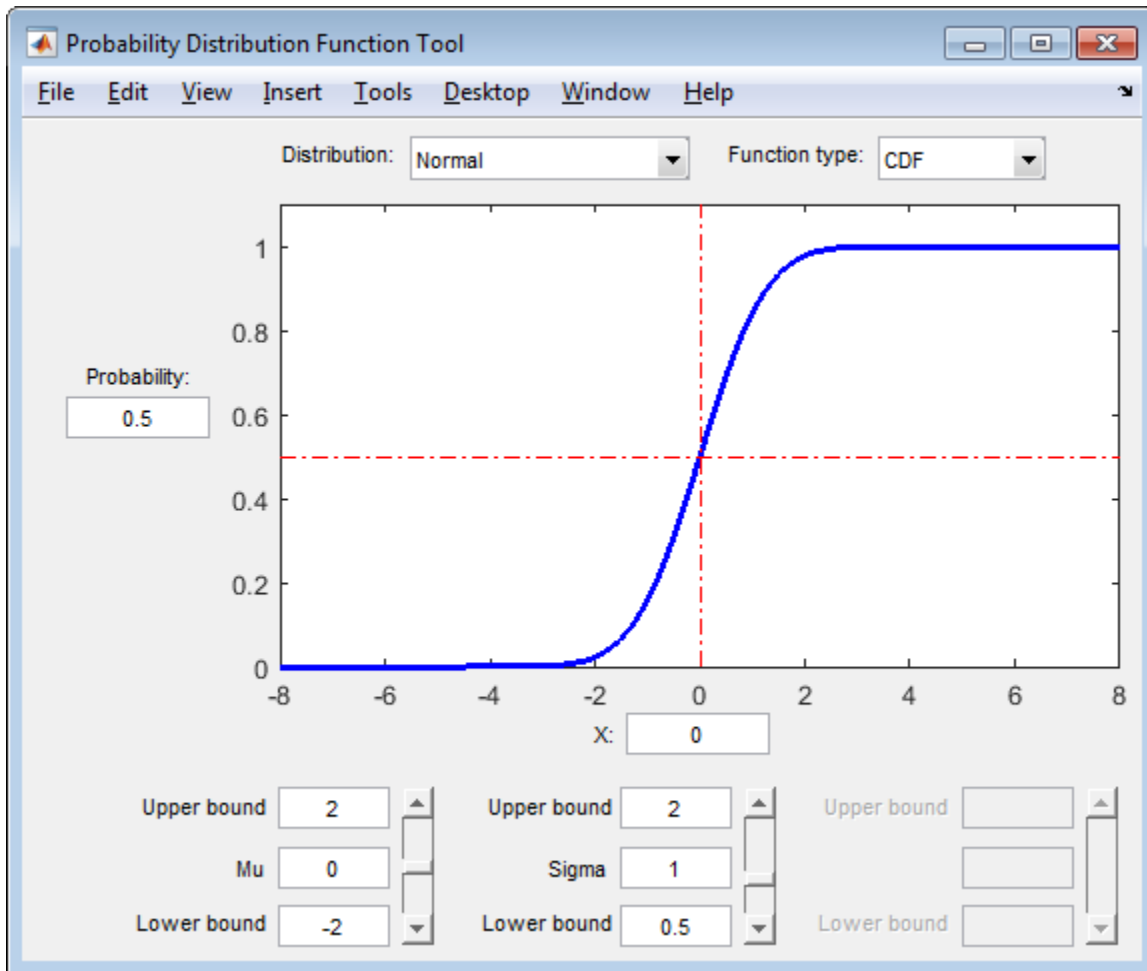
## Examples

### Explore the Probability Distribution Function User Interface

This example shows how to use the Probability Distribution Function user interface to explore the shape of cdf and pdf plots for different probability distributions and parameter values.

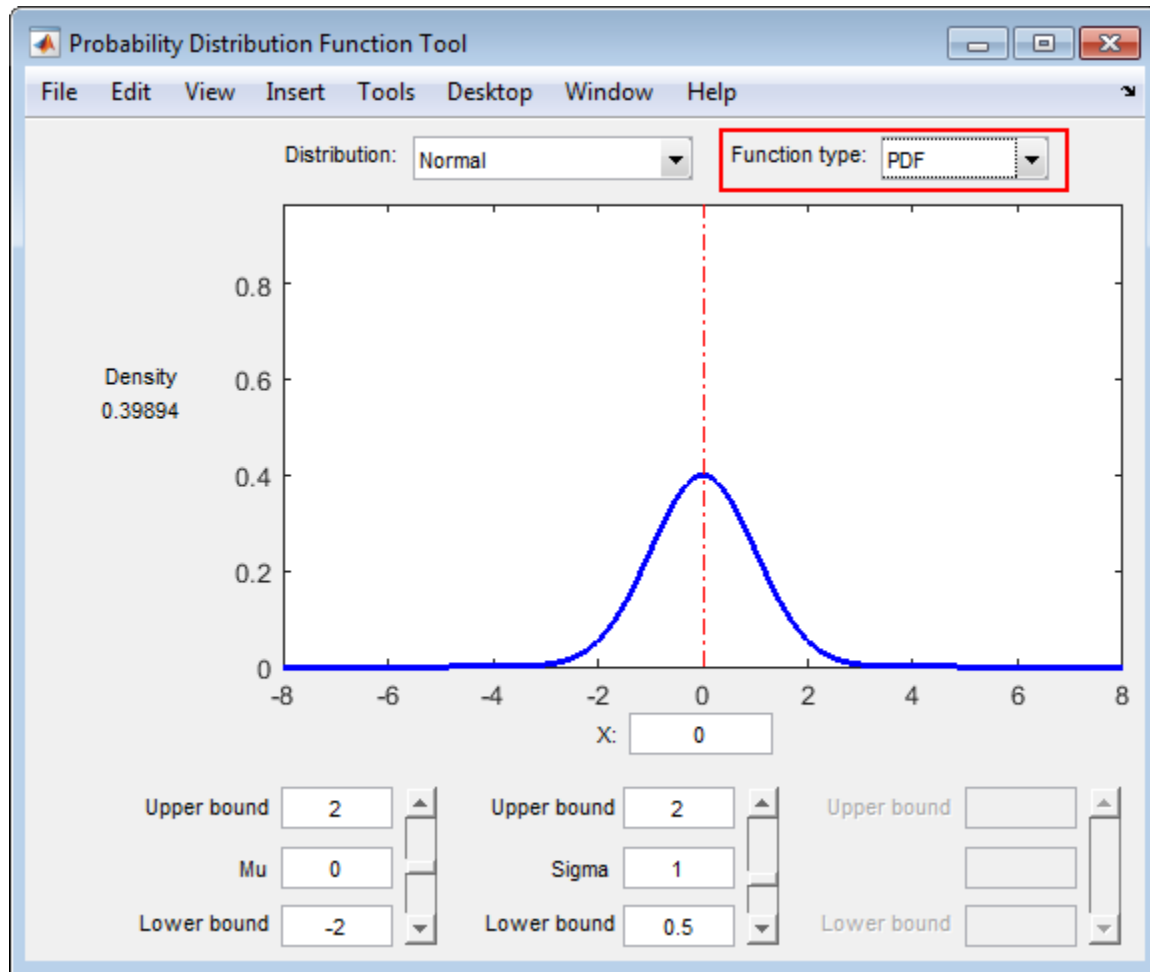
Open the Probability Distribution Function user interface.

```
disttool
```

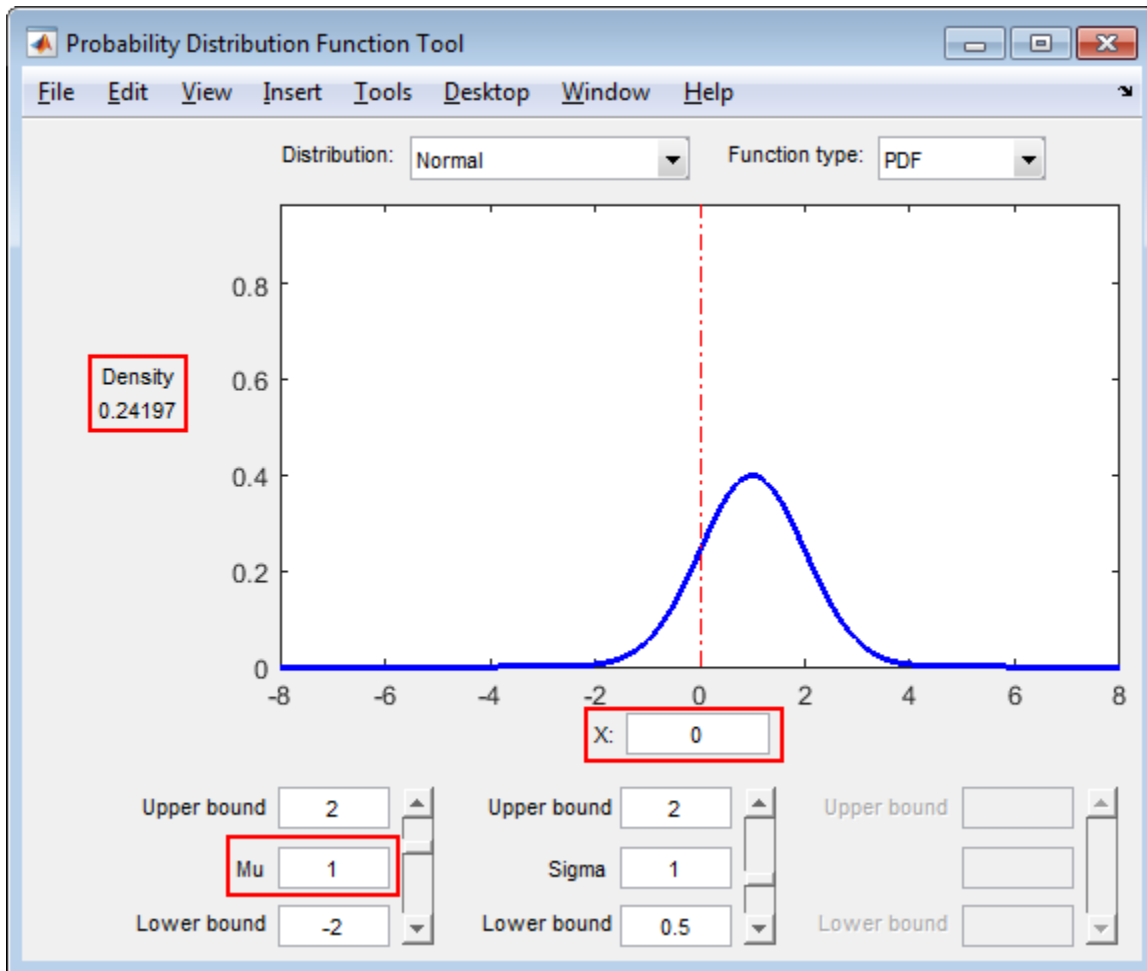


The interface opens with a plot of the cdf of the Normal distribution. The initial parameter settings are  $\mu = 0$  and  $\sigma = 1$ .

Select PDF from the **Function type** drop-down menu to plot the pdf of the Normal distribution using the same parameter values.

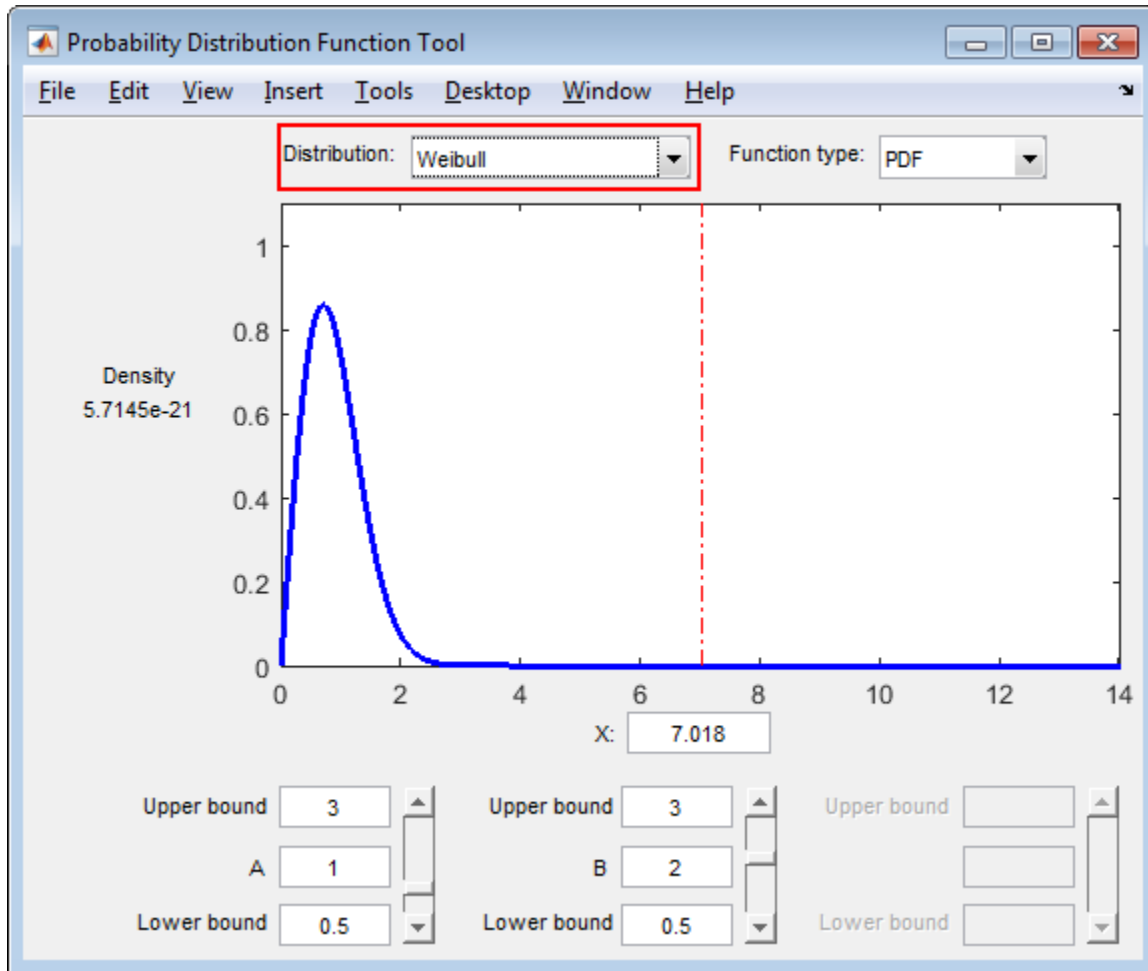


Change the value of the location parameter  $\mu$  to 1.



As the parameter values change, the shape of the plot also changes. Also, the value of  $X$  remains the same, but the density value changes because of the new parameter value.

Use the **Distribution** drop-down menu to change the distribution type from **Normal** to **Weibull**.



The shape of the plot changes, along with the names and values of the parameters.

## Parameters

### Distribution — Probability distribution

Normal (default) | Exponential | Poisson | Weibull | ...

Specify the probability distribution to explore by selecting a distribution name from the drop-down list. The drop-down list includes approximately 25 probability distribution options, including Normal, Exponential, Poisson, Weibull, and more.

#### **Function type — Probability distribution function type**

CDF (default) | PDF

Specify the probability distribution function type as CDF (cumulative distribution function) or PDF (probability density function) by selecting the function name from the drop-down list.

#### **Probability — Cumulative distribution function value**

numeric value in the range [0,1]

Specify the cumulative distribution function (cdf) value of interest as a numeric value in the range [0,1]. The corresponding random variable value appears in the **X** field below the plot. Alternatively, you can specify a value for **X**, and the **Probability** value will update automatically.

This option only appears if **Function type** is CDF. If **Function type** is PDF, then the probability density at the specified **X** value displays to the left of the plot.

#### **x — Random variable**

numeric value

Specify the random variable of interest as a numeric value. If the **Function type** is CDF, then the corresponding cumulative distribution function (cdf) value appears in the **Probability** field to the left of the plot. Alternatively, you can specify a value for **Probability**, and the **X** value will update automatically. If the **Function type** is PDF, then the corresponding probability density value appears to the left of the plot.

#### **Parameters — Parameter boundaries and values**

numeric value

Specify the parameter boundaries and values as numeric values. Each column contains a field for the upper bound, value, and lower bound of one parameter. The name and number of available parameters changes based on the distribution specified in the **Distribution** drop-down list. For example, if you select the Normal distribution, then `disttool` enables two columns: One column for the **Mu** parameter and one column for the **Sigma** parameter. If you select the **Exponential** distribution, then `disttool` enables one column for the **Mu** parameter.

## Tips

To change the value of **X** (on the  $y$ -axis), or **Probability** or **Density** (on the  $x$ -axis):

- Type the values of interest into the **Probability** or **X** fields;
- Click on the point of interest on the plot; or
- Click and drag the reference lines across the plot.

## See Also

### Functions

`distributionFitter` | `fitdist` | `makedist`



# double

**Class:** dataset

Convert dataset variables to double array

---

**Note** The `dataset` data type might be removed in a future release. To work with heterogeneous data, use the MATLAB® `table` data type instead. See MATLAB `table` documentation for more information.

---

## Syntax

```
b = double(A)
b = double(a,vars)
```

## Description

`b = double(A)` returns the contents of the dataset `A`, converted to one double array. The classes of the variables in the dataset must support the conversion.

`b = double(a,vars)` returns the contents of the dataset variables specified by `vars`. `vars` is a positive integer, a vector of positive integers, a variable name, a cell array containing one or more variable names, or a logical vector.

## See Also

`dataset` | `replacedata` | `single`

## droplevels

Drop levels from a nominal or ordinal array

---

**Note** The `nominal` and `ordinal` array data types might be removed in a future release. To represent ordered and unordered discrete, nonnumeric data, use the MATLAB® `categorical` data type instead.

---

## Syntax

```
B = droplevels(A)
B = droplevels(A,oldlevels)
```

## Description

`B = droplevels(A)` drops unused levels from the nominal or ordinal array `A`. The array `B` has the same size, type, and values as `A`, but has a list of potential levels that includes only those present in some element of `A`.

`B = droplevels(A,oldlevels)` removes the specified levels `oldlevels` from `A`.

`droplevels` removes levels, but does not remove elements. Elements of `B` that correspond to elements of `A` having levels in `oldlevels` all have an undefined level.

## Examples

### Drop Levels From an Ordinal Array

Bin patient ages into ordinal levels corresponding to 10-year intervals.

```
load hospital
edges = 0:10:100;
labels = strcat(num2str((0:10:90)', '%d'), {'s'});
```

```
A = ordinal(hospital.Age, labels, [], edges);
getlabels(A)
```

```
ans =
```

```
1x10 cell array
```

```
Columns 1 through 7
```

```
    {'0s'}    {'10s'}    {'20s'}    {'30s'}    {'40s'}    {'50s'}    {'60s'}
```

```
Columns 8 through 10
```

```
    {'70s'}    {'80s'}    {'90s'}
```

Drop any levels that have no patients in them.

```
A = droplevels(A);
getlabels(A)
```

```
ans =
```

```
1x4 cell array
```

```
    {'20s'}    {'30s'}    {'40s'}    {'50s'}
```

- “Add and Drop Category Levels” on page 2-25

## Input Arguments

### **A** — Nominal or ordinal array

nominal array | ordinal array

Nominal or ordinal array, specified as a nominal or ordinal array object created using `nominal` or `ordinal`.

### **oldlevels** — Levels to remove

cell array of character vectors | 2-D character matrix

Levels to remove from the `nominal` or `ordinal` array, specified as a cell array of character vectors or 2-D character matrix.

Data Types: `char` | `cell`

## Output Arguments

**B** — Nominal or ordinal array

`nominal` array | `ordinal` array

Nominal or ordinal array, returned as a `nominal` or `ordinal` array object.

## See Also

`addlevels` | `mergelevels` | `nominal` | `ordinal` | `reorderlevels`

## Topics

“Add and Drop Category Levels” on page 2-25

`nominal`

`ordinal`

**Introduced in R2007a**

# dummyvar

Create dummy variables

## Syntax

```
D = dummyvar(group)
```

## Description

`D = dummyvar(group)` returns a matrix `D` containing zeros and ones, whose columns are dummy variables for the grouping variable `group`. Columns of `group` represent categorical predictor variables, with values indicating categorical levels. Rows of `group` represent observations across variables.

`group` can be a numeric vector or categorical column vector representing levels within a single variable, a cell array containing one or more grouping variables, or a numeric matrix or cell array of categorical column vectors representing levels within multiple variables. If `group` is a numeric vector or matrix, values in any column must be positive integers in the range from 1 to the number of levels for the corresponding variable. In this case, `dummyvars` treats each column as a separate numeric grouping variable. With multiple grouping variables, the sets of dummy variable columns are in the same order as the grouping variables in `group`.

The order of the dummy variable columns in `D` matches the order of the groups defined by `group`. When `group` is a categorical vector, the groups and their order match the output of the `getlabels(group)` method. When `group` is a numeric vector, `dummyvar` assumes that the groups and their order are `1:max(group)`. In this respect, `dummyvars` treats a numeric grouping variable differently than `grp2idx`.

If `group` is  $n$ -by- $p$ , `D` is  $n$ -by- $S$ , where  $S$  is the sum of the number of levels in each of the columns of `group`. The number of levels  $s$  in any column of `group` is the maximum positive integer in the column or the number of categorical levels. Levels are considered distinct if they appear in different columns of `group`, even if they have the same value. Columns of `D` are, from left to right, dummy variables created from the first column of `group`, followed by dummy variables created from the second column of `group`, etc.

`dummyvar` treats NaN values or undefined categorical levels in `group` as missing data and returns NaN values in `D`.

Dummy variables are used in regression analysis and ANOVA to indicate values of categorical predictors.

---

**Note** If a column of 1s is introduced in the matrix `D`, the resulting matrix  $X = [\text{ones}(\text{size}(D,1),1) \ D]$  will be rank deficient. The matrix `D` itself will be rank deficient if `group` has multiple columns. This is because dummy variables produced from any column of `group` always sum to a column of 1s. Regression and ANOVA calculations often address this issue by eliminating one dummy variable (implicitly setting the coefficients for dropped columns to zero) from each group of dummy variables produced by a column of `group`.

---

## Examples

Suppose you are studying the effects of two machines and three operators on a process. Use `group` to organize predictor data on machine-operator combinations:

```
machine = [1 1 1 1 2 2 2 2]';
operator = [1 2 3 1 2 3 1 2]';
group = [machine operator]
group =
     1     1
     1     2
     1     3
     1     1
     2     2
     2     3
     2     1
     2     2
```

Use `dummyvar` to create dummy variables for a regression or ANOVA calculation:

```
D = dummyvar(group)
D =
     1     0     1     0     0
     1     0     0     1     0
     1     0     0     0     1
     1     0     1     0     0
```

0	1	0	1	0
0	1	0	0	1
0	1	1	0	0
0	1	0	1	0

The first two columns of `D` represent observations of machine 1 and machine 2, respectively; the remaining columns represent observations of the three operators.

## See Also

`anova1` | `regress`

## Topics

“Grouping Variables” on page 2-59

“Dummy Indicator Variables” on page 2-62

“Regression with Categorical Covariates” on page 2-65

**Introduced before R2006a**

## dwtest

Durbin-Watson test

### Syntax

```
p = dwtest(r, x)
p = dwtest(r, x, Name, Value)
[p, d] = dwtest(____)
```

### Description

`p = dwtest(r, x)` returns the  $p$ -value for the Durbin-Watson test on page 32-1486 of the null hypothesis that the residuals from a linear regression are uncorrelated. The alternative hypothesis is that there is autocorrelation among the residuals.

`p = dwtest(r, x, Name, Value)` returns the  $p$ -value for the Durbin-Watson test with additional options specified by one or more name-value pair arguments. For example, you can conduct a one-sided test or calculate the  $p$ -value using a normal approximation.

`[p, d] = dwtest(____)` also returns the Durbin-Watson test statistic,  $d$ , using any of the input arguments from the previous syntaxes.

### Examples

#### Test Residuals For Correlation

Load the sample census data.

```
load census
```

Create a design matrix using the census date (`cdate`) as the predictor. Add a column of 1 values to include a constant term.



```
n = length(cdate);  
x = [ones(n,1), cdate];
```

Fit a linear regression to the data.

```
[b,bint,r] = regress(pop,x);
```

Test the null hypothesis that there is no autocorrelation among the residuals,  $r$ .

```
[p,d] = dwtest(r,x)
```

```
p =  
  
    0
```

```
d =  
  
    0.1308
```

The returned value  $p = 0$  indicates rejection of the null hypothesis at the 5% significance level.

### One-Sided Hypothesis Test

Load the sample census data.

```
load census
```

Create a design matrix using the census date (`cdate`) as the predictor. Add a column of 1 values to include a constant term.

```
n = length(cdate);  
x = [ones(n,1), cdate];
```

Fit a linear regression to the data.

```
[b,bint,r] = regress(pop,x);
```

Test the null hypothesis that there is no autocorrelation among regression residuals, against the alternative hypothesis that the autocorrelation is greater than zero.

```
[p,d] = dwtest(r,x,'Tail','right')
```

```
p =  
    0
```

```
d =  
    0.1308
```

The returned value  $p = 0$  indicates rejection of the null hypothesis at the 5% significance level, in favor of the alternative hypothesis that the autocorrelation among residuals is greater than zero.

## Input Arguments

### **x** — Design matrix

matrix

Design matrix for a linear regression, specified as a matrix. Include a column of 1 values in the design matrix so the model contains a constant term.

Data Types: `single` | `double`

### **r** — Regression residuals

vector

Regression residuals, specified as a vector. Obtain  $r$  by performing a linear regression using a function such as `regress`, or by using the backslash operator.

Data Types: `single` | `double`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: 'Tail', 'right', 'Method', 'approximate' specifies a right-tailed hypothesis test and calculates the  $p$ -value using a normal approximation.

#### **Method** — Computation method for $p$ -value

'exact' | 'approximate'

Computation method for the  $p$ -value, specified as the comma-separated pair consisting of 'Method' and one of the following.

- 'exact'                Calculate an exact  $p$ -value using the Pan algorithm. This is the default if the sample size is less than 400.
- 'approximate'       Calculate the  $p$ -value using a normal approximation. This is the default if the sample size is 400 or larger.

Example: 'Method', 'exact'

#### **Tail** — Type of alternative hypothesis

'both' (default) | 'right' | 'left'

Type of alternative hypothesis to evaluate, specified as the comma-separated pair consisting of 'Tail' and one of the following.

- 'both'                Test the alternate hypothesis that autocorrelation among the residuals is not zero.
- 'right'               Test the alternative hypothesis that autocorrelation among the residuals is greater than zero.
- 'left'                Test the alternative hypothesis that autocorrelation among the residuals is less than zero.

Example: 'Tail', 'right'

## Output Arguments

### **p** — $p$ -value

scalar value in the range [0,1]

$p$ -value of the test, returned as a scalar value in the range [0,1].  $p$  is the probability of observing a test statistic as extreme as, or more extreme than, the observed value under the null hypothesis. Small values of  $p$  cast doubt on the validity of the null hypothesis.

**d** — Test statistic

nonnegative scalar value

Test statistic of the hypothesis test, returned as a nonnegative scalar value.

## Definitions

### Durbin-Watson Test

The Durbin-Watson test is used to test the null hypothesis that linear regression residuals are uncorrelated, against the alternative that autocorrelation exists.

The test statistic for the Durbin-Watson test is

$$d = \frac{\sum_{t=2}^T (e_t - e_{t-1})^2}{\sum_{t=1}^T e_t^2},$$

where  $T$  is the number of observations, and  $e_t$  is the residual at time  $t$ .

The  $p$ -value of the Durbin-Watson test is the probability of observing a test statistic as extreme as, or more extreme than, the observed value under the null hypothesis. A significantly small  $p$ -value casts doubt on the validity of the null hypothesis and indicates correlation among residuals. The  $p$ -value can be calculated exactly using the Pan algorithm. Alternatively, the  $p$ -value can be estimated using a normal approximation.

## See Also

`regress`

Introduced in R2006a

# dwtest

**Class:** LinearModel

Durbin-Watson test of linear model

## Syntax

```
P = dwtest mdl
[P, DW] = dwtest mdl
[P, DW] = dwtest mdl, method
[P, DW] = dwtest mdl, method, tail
```

## Description

`P = dwtest(mdl)` returns the  $p$ -value of the Durbin-Watson test on the `mdl` linear model.

`[P, DW] = dwtest(mdl)` also returns the Durbin-Watson statistic, `DW`.

`[P, DW] = dwtest(mdl, method)` specifies the method `dwtest` uses to compute the  $p$ -value.

`[P, DW] = dwtest(mdl, method, tail)` specifies the alternative hypothesis.

## Input Arguments

**mdl** — Full, fitted linear regression model

LinearModel object

Full, fitted linear regression model, specified as a `LinearModel` object constructed using `fitlm` or `stepwiselm`.

**method** — Algorithm for computing  $p$ -value

'exact' | 'approximate'

Algorithm for computing the  $p$ -value, specified as one of the following:

- 'exact' — Calculates an exact  $p$ -value using Pan's algorithm.
- 'approximate' — Calculates the  $p$ -value using a normal approximation.

The default is 'exact' when the sample size is less than 400, 'approximate' otherwise.

**tail** — Alternative hypothesis to test

'both' (default) | 'right' | 'left'

Alternative hypothesis to test, specified as one of the following:

Tail	Alternative Hypothesis
'both'	Serial correlation is not 0.
'right'	Serial correlation is greater than 0 (right-tailed test).
'left'	Serial correlation is less than 0 (left-tailed test).

`dwtest` tests whether `mdl` has no serial correlation against the specified alternative hypotheses.

## Output Arguments

**P** —  $p$ -value of the test

numeric value

$p$ -value of the test, returned as a numeric value. `dwtest` tests if the residuals are uncorrelated, against the alternative that there is autocorrelation among them. Small values of `P` indicate that the residuals are correlated.

**DW** — Durbin-Watson statistic

numeric value

Durbin-Watson statistic, returned as a numeric value.

## Examples

## Test Residuals for Autocorrelation

Examine whether the residuals from a fitted model of census data over time have autocorrelated residuals.

Load the census data and create a linear model.

```
load census
mdl = fitlm(cdate, pop);
```

Find the  $P$ -value of the Durbin-Watson autocorrelation test.

```
P = dwtest(mdl)
```

```
P =
    0
```

There is significant autocorrelation in the residuals.

## Definitions

### Durbin-Watson Statistic

Let  $r$  be the vector of residuals (in `mdl.residuals.response`). The Durbin-Watson statistic is

$$DW = \frac{\sum_{i=1}^{n-1} (r_{i+1} - r_i)^2}{\sum_{i=1}^n r_i^2}.$$

## Algorithms

Approximate calculation of the  $p$ -value uses a normal approximation [1]. Exact calculation uses Pan's algorithm [2].

## References

- [1] Durbin, J., and G. S. Watson. *Testing for Serial Correlation in Least Squares Regression I*. *Biometrika* 37, pp. 409–428, 1950.
- [2] Farebrother, R. W. *Pan's Procedure for the Tail Probabilities of the Durbin-Watson Statistic*. *Applied Statistics* 29, pp. 224–227, 1980.

## See Also

`LinearModel`

## Topics

“Linear Regression” on page 11-11



# ecdf

Empirical cumulative distribution function

## Syntax

```
[f, x] = ecdf(y)
[f, x] = ecdf(y, Name, Value)
[f, x, flo, fup] = ecdf( ___ )
```

```
ecdf( ___ )
ecdf(ax, ___ )
```

## Description

`[f, x] = ecdf(y)` returns the empirical cumulative distribution function (cdf), `f`, evaluated at the points in `x`, using the data in the vector `y`.

In survival and reliability analysis, this empirical cdf is called the Kaplan-Meier estimate. And the data might correspond to survival or failure times.

`[f, x] = ecdf(y, Name, Value)` returns the empirical function values, `f`, evaluated at the points in `x`, with additional options specified by one or more `Name, Value` pair arguments.

For example, you can specify the type of function to evaluate or which data is censored.

`[f, x, flo, fup] = ecdf( ___ )` also returns the 95% lower and upper confidence bounds for the evaluated function values. You can use any of the input arguments in the previous syntaxes.

`ecdf` computes the confidence bounds using Greenwood's formula on page 32-1503. They are not simultaneous confidence bounds.

`ecdf( ___ )` plots the evaluated function.

`ecdf(ax, ___)` plots the evaluated function using axes with the handle, `ax`, instead of the current axes returned by `gca`.

## Examples

### Compute Empirical Cumulative Distribution Function

Compute the Kaplan-Meier estimate of the cumulative distribution function (cdf) for simulated survival data.

Generate survival data from a Weibull distribution with parameters 3 and 1.

```
rng default; % for reproducibility
failuretime = random('wbl',3,1,15,1);
```

Compute the Kaplan-Meier estimate of the cdf for survival data.

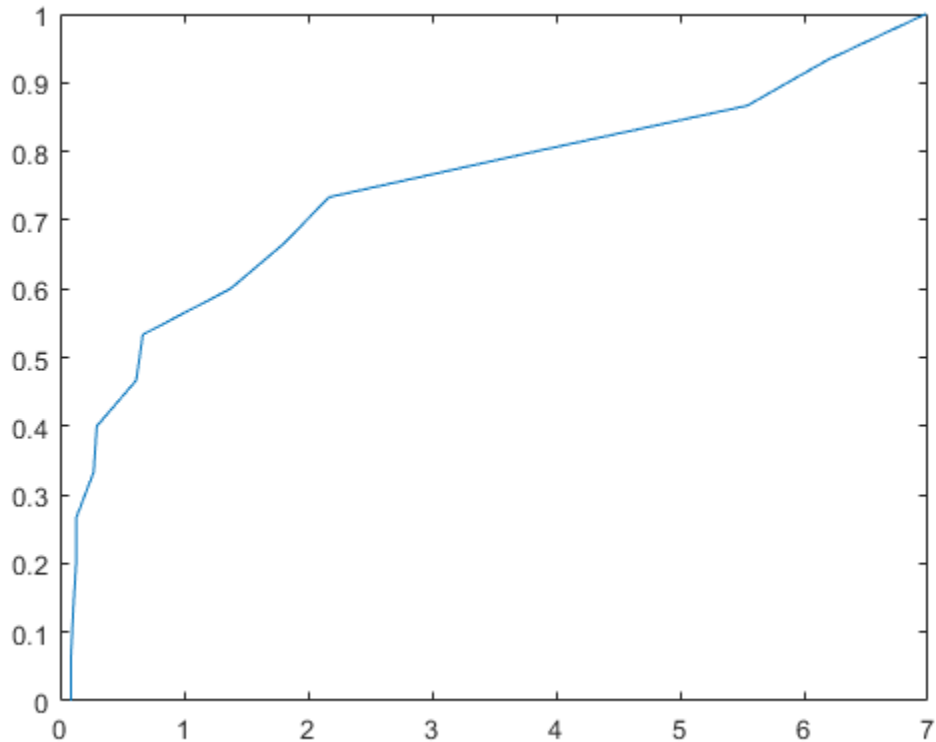
```
[f,x] = ecdf(failuretime);
[f,x]
```

```
ans =
```

```
      0      0.0895
0.0667      0.0895
0.1333      0.1072
0.2000      0.1303
0.2667      0.1313
0.3333      0.2718
0.4000      0.2968
0.4667      0.6147
0.5333      0.6684
0.6000      1.3749
0.6667      1.8106
0.7333      2.1685
0.8000      3.8350
0.8667      5.5428
0.9333      6.1910
1.0000      6.9825
```

Plot the estimated cdf.

```
figure()  
plot(x,f)
```



### Empirical Hazard Function of Right-Censored Data

Compute and plot the hazard function of simulated right-censored survival data.

Generate failure times from a Birnbaum-Saunders distribution.

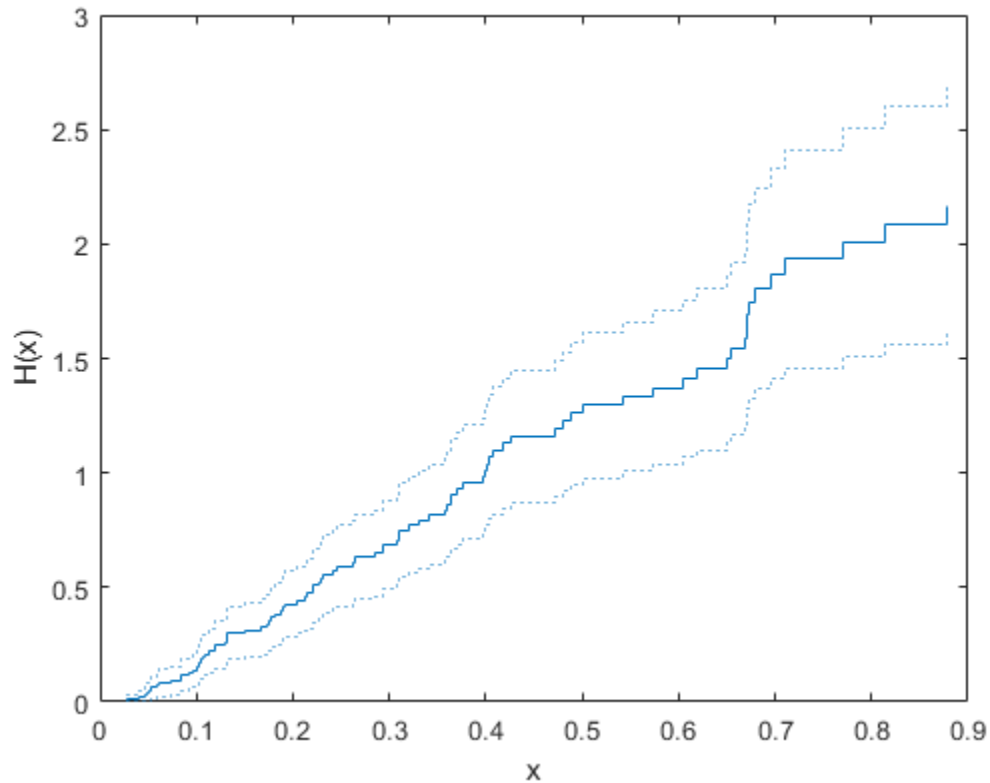
```
rng default % for reproducibility  
failuretime = random('birnbaumsaunders',0.3,1,100,1);
```

Assuming that the end of the study is at time 0.9, generate a logical array that indicates simulated failure times that are larger than 0.9 as censored data, and store this information in a vector.

```
T = 0.9;  
cens = (failuretime>T);
```

Plot the empirical hazard function for the data.

```
ecdf(failuretime, 'function', 'cumulative hazard', ...  
    'censoring', cens, 'bounds', 'on');
```



## Compare Empirical Cumulative Distribution Function (CDF) with Known CDF

Generate right-censored survival data and compare the empirical cumulative distribution function (cdf) with the known cdf.

Generate failure times from an exponential distribution with mean failure time of 15.

```
rng default % for reproducibility
y = exprnd(15,75,1);
```

Generate drop-out times from an exponential distribution with mean failure time of 30.

```
d = exprnd(30,75,1);
```

Generate the observed failure times. They are the minimum of the generated failure times and the drop-out times.

```
t = min(y,d);
```

Create a logical array that indicates generated failure times that are larger than the drop-out times. The data for which this is true are censored.

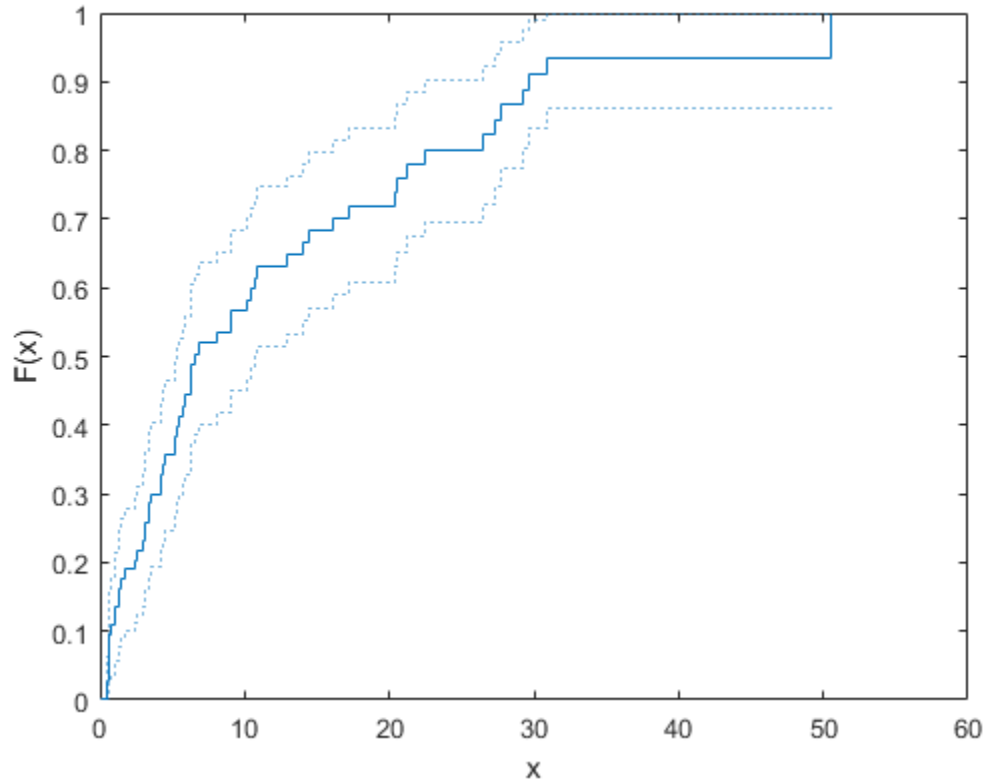
```
censored = (y>d);
```

Compute the empirical cdf and confidence bounds.

```
[f,x,flo,fup] = ecdf(t,'censoring',censored);
```

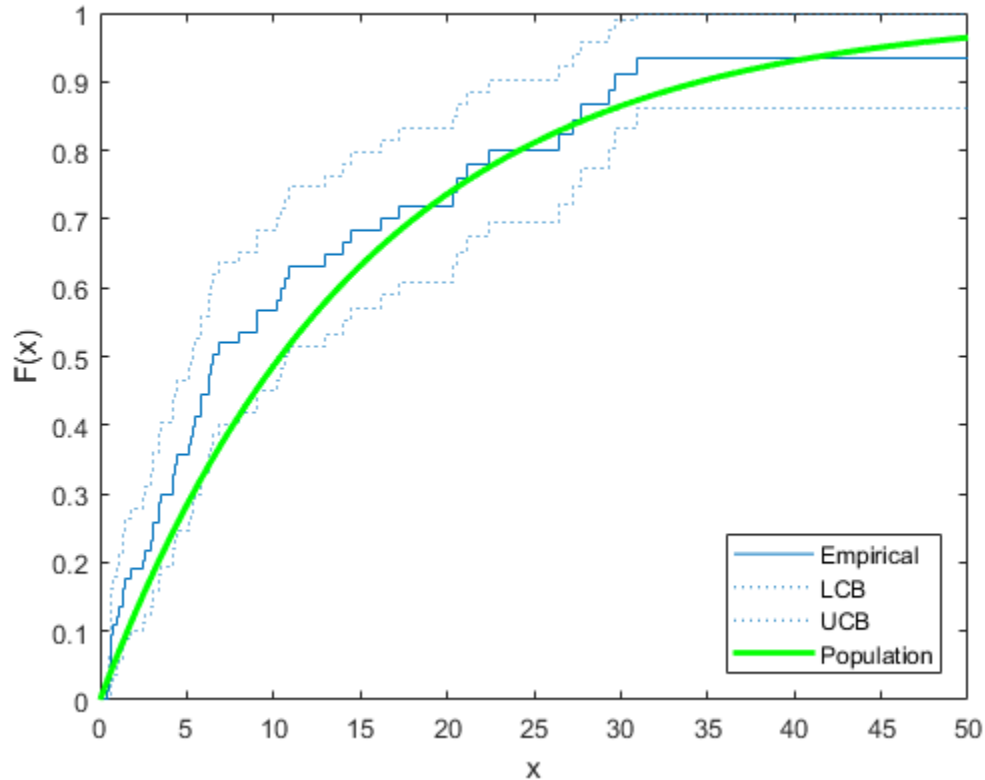
Plot the cdf and confidence bounds.

```
figure()
ecdf(t,'censoring',censored,'bounds','on');
hold on
```



Superimpose a plot of the known population cdf.

```
xx = 0:.1:max(t);  
yy = 1-exp(-xx/15);  
plot(xx,yy,'g-', 'LineWidth',2)  
axis([0 50 0 1])  
legend('Empirical','LCB','UCB','Population',...  
       'Location','SE')  
hold off
```



### Empirical Survivor Function with 99% Confidence Bounds

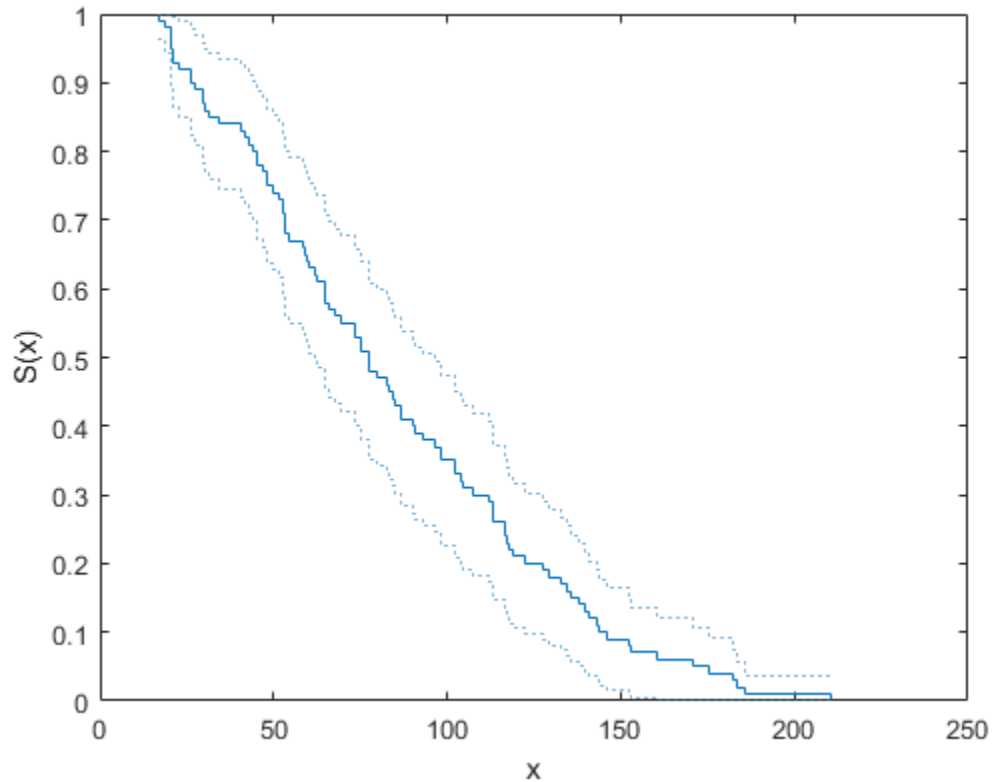
Generate survival data and plot the empirical survivor function with 99% confidence bounds.

Generate lifetime data from a Weibull distribution with parameters 100 and 2.

```
rng default % for reproducibility
R = wblrnd(100,2,100,1);
```

Plot the survivor function for the data with 99% confidence bounds.

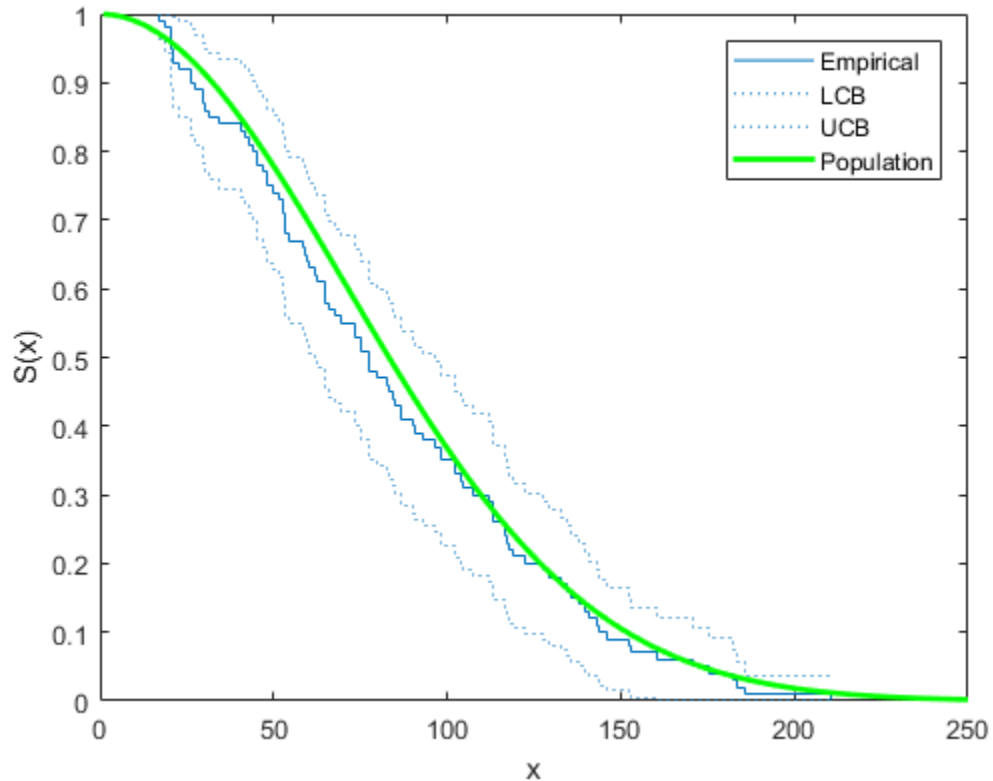
```
ecdf(R, 'function', 'survivor', 'alpha', 0.01, 'bounds', 'on')
hold on
```



Fit the Weibull survivor function.

```
x = 1:1:250;
wblsurv = 1-cdf('weibull', x, 100, 2);
plot(x, wblsurv, 'g-', 'LineWidth', 2)
legend('Empirical', 'LCB', 'UCB', 'Population', ...
'Location', 'NE')
```





The survivor function based on the actual distribution is within the confidence bounds.

- “Hazard and Survivor Functions for Different Groups” on page 14-17
- “Survivor Functions for Two Groups” on page 14-24

## Input Arguments

### **y** — Input data

column vector

Input data, specified as a column vector. For example, in survival or reliability analysis, data might be survival or failure times for each item or individual.

Data Types: `single` | `double`

### **ax** — Axes handle

handle

Axes handle for the figure `ecdf` plots to, specified as a handle.

For instance, if `h` is a handle for a figure, then `ecdf` can plot to that figure as follows.

Example: `ecdf(h,x)`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'censoring', c, 'function', 'cumulative hazard', 'alpha', 0.025, 'bounds', 'on'` specifies that `ecdf` returns the cumulative hazard function and plots the 97.5% confidence bounds, accounting for the censored data specified by vector `c`.

### **censoring** — Indicator of censored data

array of 0s (default) | vector of 0s and 1s

Indicator of censored data, specified as the comma-separated pair including `'censoring'` and a Boolean array of the same size as `x`. Enter 1 for observations that are right-censored and 0 for observations that are fully observed. Default is all observations are fully observed.

For instance, if vector `cdata` stores the censored data information, you can enter the censoring information as follows.

Example: `'censoring', cdata`

Data Types: `logical`

### **frequency** — Frequency of observations

array of 1s (default) | vector of nonnegative scalars

Frequency of observations, specified as the comma-separated pair consisting of `'frequency'` and a vector containing nonnegative integer counts. This vector is the

same size as the vector `x`. The  $j$ th element of this vector gives the number of times the  $j$ th element of `x` was observed. Default is one observation per element of `x`.

For instance, if `failurefreq` is a vector of frequencies, then you can enter it as follows.

Example: `'frequency', failurefreq`

Data Types: `single` | `double`

### **alpha** — Confidence level

`0.05` (default) | scalar value in the range (0,1)

Confidence level for the confidence interval of the evaluated function, specified as the comma-separated pair consisting of `'alpha'` and a scalar value between in the range (0,1). Default is 0.05 for 95% confidence. For a given value `alpha`, the confidence level is  $100(1-\alpha)\%$ .

For instance, for a 99% confidence interval, you can specify the `alpha` value as follows.

Example: `'alpha', 0.01`

Data Types: `single` | `double`

### **function** — Type of function returned

`'cdf'` (default) | `'survivor'` | `'cumulative hazard'`

Type of function that `ecdf` evaluates and returns, specified as the comma-separated pair consisting of `'function'` and one of the following.

<code>'cdf'</code>	Default. Cumulative distribution function.
<code>'survivor'</code>	Survivor function.
<code>'cumulative hazard'</code>	Cumulative hazard function.

Example: `'function', 'cumulative hazard'`

Data Types: `char`

### **bounds** — Indicator for including bounds

`'off'` (default) | `'on'`

Indicator for including bounds, specified as the comma-separated pair consisting of `'bounds'` and one of the following.

'off'	Default. Specify to omit bounds.
'on'	Specify to include bounds.

---

**Note** This name-value argument is used only for plotting.

---

Example: 'bounds', 'on'

Data Types: char

## Output Arguments

### **f** — Function values

column vector

Function values evaluated at the points in *x*, returned as a column vector.

### **x** — Distinct observed points

column vector

Distinct observed points in data vector *y*, returned as a column vector.

### **f<sub>lo</sub>** — Lower confidence bound

column vector

Lower confidence bound for the evaluated function, returned as a column vector. *ecdf* computes the confidence bounds using Greenwood's formula on page 32-1503. They are not simultaneous confidence bounds.

### **f<sub>up</sub>** — Upper confidence bound

column vector

Upper confidence bound for the evaluated function, returned as a column vector. *ecdf* computes the confidence bounds using Greenwood's formula on page 32-1503. They are not simultaneous confidence bounds.

## Definitions

### Greenwood's Formula

Approximation for the variance of Kaplan-Meier estimator.

The variance estimate is given by

$$V(S(t)) = S^2(t) \sum_{t_i < T} \frac{d_i}{r_i(r_i - d_i)},$$

where  $r_i$  is the number at risk at time  $t_i$ , and  $d_i$  is the number of failures at time  $t_i$ .

### References

- [1] Cox, D. R., and D. Oakes. *Analysis of Survival Data*. London: Chapman & Hall, 1984.
- [2] Lawless, J. F. *Statistical Models and Methods for Lifetime Data*. 2nd ed., Hoboken, NJ: John Wiley & Sons, Inc., 2003.

### See Also

`cdfplot` | `ecdfhist`

### Topics

- “Hazard and Survivor Functions for Different Groups” on page 14-17
- “Survivor Functions for Two Groups” on page 14-24
- “What Is Survival Analysis?” on page 14-2
- “Kaplan-Meier Method” on page 14-10

Introduced before R2006a

## ecdfhist

Histogram based on empirical cumulative distribution function

### Syntax

```
[n,c] = ecdfhist(f,x)
[n,c] = ecdfhist(f,x,m)

n = ecdfhist(f,x,centers)

ecdfhist(____)
```

### Description

`[n,c] = ecdfhist(f,x)` returns the heights,  $n$ , of histogram bars for 10 equally spaced bins and the position of the bin centers,  $c$ .

`ecdfhist` computes the bar heights from the increases in the empirical cumulative distribution function,  $f$ , at evaluation points,  $x$ . It normalizes the bar heights so that the area of the histogram is equal to 1. In contrast, `histogram` produces bars with heights representing bin counts.

`[n,c] = ecdfhist(f,x,m)` returns the histogram bars using  $m$  bins.

`n = ecdfhist(f,x,centers)` returns the heights of the histogram bars with bin centers specified by `centers`.

`ecdfhist(____)` plots the histogram bars.

### Examples

## Return Histogram Bar Heights and Bin Centers

Compute the histogram bar heights based on the empirical cumulative distribution function.

Generate failure times from a Birnbaum-Saunders distribution.

```
rng('default') % for reproducibility
failuretime = random('birnbaumsaunders',0.3,1,100,1);
```

Assuming that the end of the study is at time 0.9, mark the generated failure times that are larger than 0.9 as censored data and store that information in a vector.

```
T = 0.9;
cens = (failuretime>T);
```

Compute the empirical cumulative distribution function for the data.

```
[f,x] = ecdf(failuretime,'censoring',cens);
```

Now, find the bar heights of the histogram using the cumulative distribution function estimate.

```
[n,c] = ecdfhist(f,x);
[n' c']
```

```
ans =
```

```
2.3529    0.0715
1.7647    0.1565
1.4117    0.2415
1.5294    0.3265
1.0588    0.4115
0.4706    0.4965
0.4706    0.5815
0.9412    0.6665
0.2353    0.7515
0.2353    0.8365
```

### Return Bar Heights and Bin Centers for a Given Number of Bins

Compute the bar heights for six bins using the empirical cumulative distribution function and also return the bin centers.

Generate failure times from a Birnbaum-Saunders distribution.

```
rng('default') % for reproducibility
failuretime = random('birnbaumsaunders',0.3,1,100,1);
```

Assuming that the end of the study is at time 0.9, mark the generated failure times that are larger than 0.9 as censored data and store that information in a vector.

```
T = 0.9;
cens = (failuretime>T);
```

First, compute the empirical cumulative distribution function for the data.

```
[f,x] = ecdf(failuretime,'censoring',cens);
```

Now, estimate the histogram with six bins using the cumulative distribution function estimate.

```
[n,c] = ecdfhist(f,x,6);
[n' c']
```

```
ans =
```

```
1.9764    0.0998
1.7647    0.2415
1.1294    0.3831
0.4235    0.5248
0.7764    0.6665
0.2118    0.8081
```

### Draw Histogram for Given Bin Centers

Draw the histogram of the empirical cumulative distribution histogram for specified bin centers.

Generate failure times from a Birnbaum-Saunders distribution.



---

```
rng default; % For reproducibility
failuretime = random('birnbaumsaunders',0.3,1,100,1);
```

Assuming that the end of the study is at time 0.9, mark the generated failure times that are larger than 0.9 as censored data and store that information in a vector.

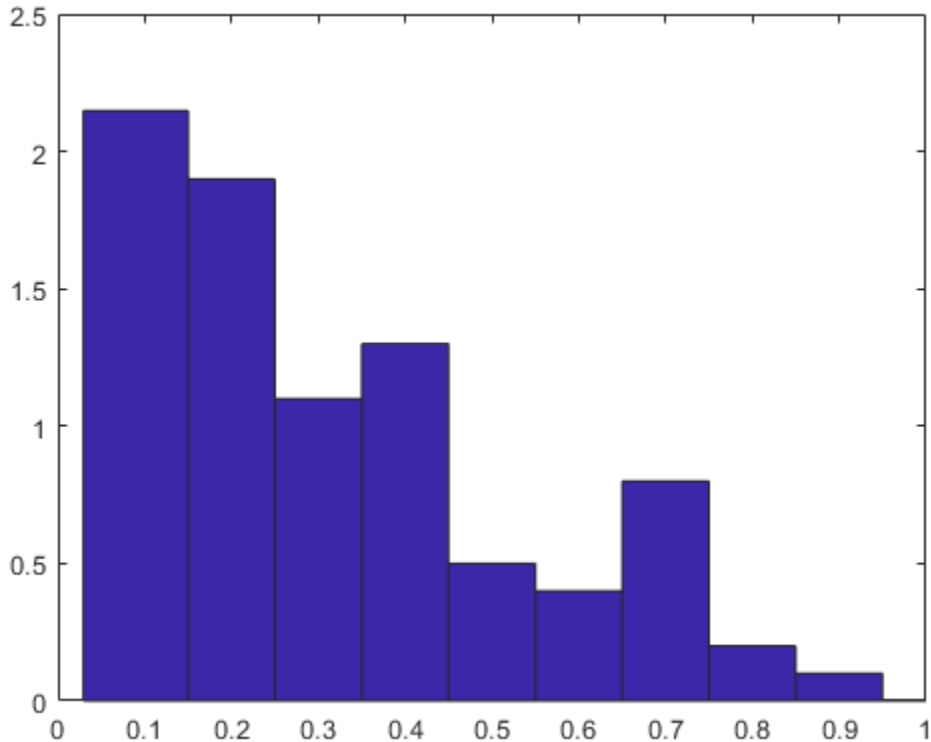
```
T = 0.9;
cens = (failuretime>T);
```

Define bin centers.

```
centers = 0.1:0.1:1;
```

Compute the empirical cumulative distribution function for the data and draw the histogram for specified bin centers.

```
[f,x] = ecdf(failuretime,'censoring',cens);
ecdfhist(f,x,centers)
axis([0 1 0 2.5])
```



### Compare Histogram with Known Probability Distribution Function

Generate right-censored survival data and compare the histogram from cumulative distribution function with the known probability distribution function.

Generate failure times from an exponential distribution with mean failure time of 15.

```
rng default; % For reproducibility  
y = exprnd(15,75,1);
```

Generate drop-out times from an exponential distribution with mean failure time of 30.

```
d = exprnd(30,75,1);
```

Record the minimum of these times as the observed failure times.

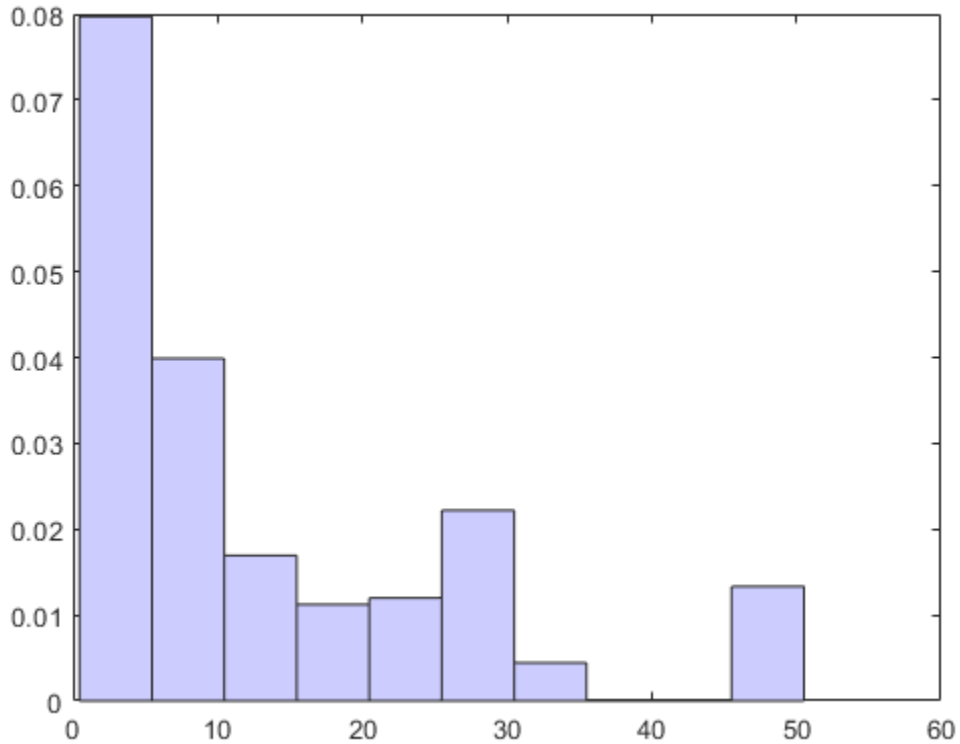
```
t = min(y,d);
```

Generate censoring by finding the generated failure times that are greater than the drop-out times.

```
censored = (y>d);
```

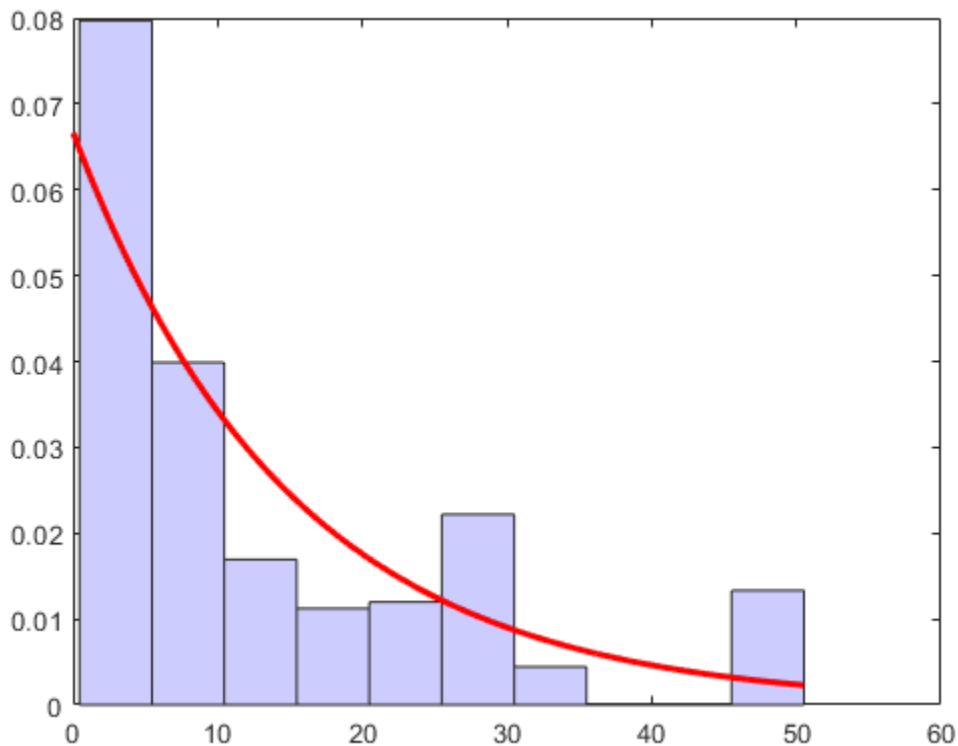
Calculate the empirical cdf and plot a histogram using the empirical cumulative distribution function.

```
[f,x] = ecdf(t, 'censoring', censored);  
ecdfhist(f,x)  
h = findobj(gca, 'Type', 'patch');  
h.FaceColor = [.8 .8 1];  
hold on
```



Superimpose a plot of the known population pdf.

```
xx = 0:.1:max(t);  
yy = exp(-xx/15)/15;  
plot(xx,yy,'r-', 'LineWidth',2)  
hold off
```



## Input Arguments

**f** — Empirical cdf values

vector

Empirical cdf values at given evaluation points,  $x$ , specified as a vector.

For instance, you can use `ecdf` to obtain the empirical cdf values and enter them in `ecdfhist` as follows.

```
Example: [f,x] = ecdf(failure); ecdfhist(f,x);
```

Data Types: `single` | `double`

**x — Evaluation points**

vector

Evaluation points at which empirical cdf values, `f`, are calculated, specified as a vector.

For instance, you can use `ecdf` to obtain the empirical cdf values and enter them in `ecdfhist` as follows.

```
Example: [f,x] = ecdf(failure); ecdfhist(f,x);
```

Data Types: `single` | `double`

**m — Number of bins**

scalar

Number of bins, specified as a scalar.

For instance, you can draw a histogram with 8 bins as follows.

```
Example: ecdfhist(f,x,8)
```

Data Types: `single` | `double`

**centers — Center points of bins**

vector

Center points of bins, specified as a vector.

```
Example: centers = 2:2:10; ecdfhist(f,x,centers);
```

Data Types: `single` | `double`

## Output Arguments

**n — Heights of histogram bars**

row vector

Heights of histogram bars `ecdfhist` calculates based on the empirical cdf values, returned as a row vector.

**c — Position of bin centers**

row vector

Position of bin centers, returned as a row vector.

## See Also

`ecdf` | `histc` | `histogram`

## Topics

“Nonparametric and Empirical Probability Distributions” on page 5-37

**Introduced before R2006a**

## edge

**Class:** ClassificationKNN

Edge of  $k$ -nearest neighbor classifier

## Syntax

```
E = edge (mdl, tbl, ResponseVarName)
E = edge (mdl, tbl, Y)
E = edge (mdl, X, Y)
E = edge ( ____, Name, Value)
```

## Description

`E = edge (mdl, tbl, ResponseVarName)` returns the classification edge for `mdl` with data `tbl` and classification `tbl.ResponseVarName`.

`E = edge (mdl, tbl, Y)` returns the classification edge for `mdl` with data `tbl` and classification `Y`.

`E = edge (mdl, X, Y)` returns the classification edge for `mdl` with data `X` and classification `Y`.

`E = edge ( ____, Name, Value)` computes the edge with additional options specified by one or more `Name, Value` pair arguments, using any of the previous syntaxes.

## Input Arguments

**mdl** — Classifier model

classifier model object

$k$ -nearest neighbor classifier model, returned as a classifier model object.

Note that using the 'CrossVal', 'Kfold', 'Holdout', 'Leaveout', or 'CVPartition' options results in a model of class



`ClassificationPartitionedModel`. You cannot use a partitioned tree for prediction, so this kind of tree does not have a `predict` method.

Otherwise, `mdl` is of class `ClassificationKNN`, and you can use the `predict` method to make predictions.

### **tbl — Sample data**

table

Sample data used to train the model, specified as a table. Each row of `tbl` corresponds to one observation, and each column corresponds to one predictor variable. Optionally, `tbl` can contain one additional column for the response variable. Multi-column variables and cell arrays other than cell arrays of character vectors are not allowed.

If `tbl` contains the response variable used to train `mdl`, then you do not need to specify `ResponseVarName` or `Y`.

If you trained `mdl` using sample data contained in a `table`, then the input data for this method must also be in a table.

Data Types: `table`

### **ResponseVarName — Response variable name**

name of a variable in `tbl`

Response variable name, specified as the name of a variable in `tbl`. If `tbl` contains the response variable used to train `mdl`, then you do not need to specify `ResponseVarName`.

If you specify `ResponseVarName`, then you must do so as a character vector. For example, if the response variable is stored as `tbl.response`, then specify it as `'response'`. Otherwise, the software treats all columns of `tbl`, including `tbl.response`, as predictors.

The response variable must be a categorical or character array, logical or numeric vector, or cell array of character vectors. If the response variable is a character array, then each element must correspond to one row of the array.

### **x — Matrix of predictor values**

matrix

Matrix of predictor values. Each column of `x` represents one variable, and each row represents one observation.

**Y — Categorical variables**

categorical array | cell array of character vectors | character array | logical vector | numeric vector

A categorical array, cell array of character vectors, character array, logical vector, or a numeric vector with the same number of rows as X. Each row of Y represents the classification of the corresponding row of X.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

**Weights — Observation weights**

`ones(size(X,1))` (default) | name of a variable in `tbl` | numeric vector

Observation weights, specified as the comma-separated pair consisting of 'Weights' and a numeric vector or the name of a variable in TBL.

If you specify `Weights` as a numeric vector, then the size of `Weights` must be equal to the number of rows in X or `tbl`.

If you specify `Weights` as the name of a variable in `tbl`, you must do so as a character vector. For example, if the weights are stored as `tbl.w`, then specify it as 'w'. Otherwise, the software treats all columns of `tbl`, including `tbl.w`, as predictors.

If you specify `Weights`, `edge` computes weighted classification edge. The software weights the observations in each row of X or `tbl` with the corresponding weight in `Weights`.

## Output Arguments

**E**

Classification edge, a scalar that is the mean classification margin (see “Margin” on page 32-1518).

## Examples

### Edge Calculation

Construct a  $k$ -nearest neighbor classifier for the Fisher iris data, where  $k = 5$ .

Load the data.

```
load fisheriris
X = meas;
Y = species;
```

Construct a classifier for five-nearest neighbors.

```
mdl = fitcknn(X,Y,'NumNeighbors',5);
```

Examine the edge of the classifier for minimum, mean, and maximum observations classified 'setosa', 'versicolor', and 'virginica' respectively.

```
NewX = [min(X);mean(X);max(X)];
Y = {'setosa';'versicolor';'virginica'};
E = edge(mdl,NewX,Y)
```

```
E =
```

```
1
```

The classifier has no doubt that the Y entries are correct classifications (all five nearest neighbors of each NewX point classify as the corresponding Y entry).

## Definitions

### Edge

The edge is the mean value of the classification margin.

## Margin

The classification margin is the difference between the classification score for the true class and maximal classification score for the false classes.

Margin is a column vector with the same number of rows as  $X$ .

## Score

The score of a classification is the posterior probability of the classification. The posterior probability is the number of neighbors that have that classification, divided by the number of neighbors. For a more detailed definition that includes weights and prior probabilities, see “Posterior Probability” on page 32-4897.

# Extended Capabilities

## Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays” (MATLAB).

## See Also

`ClassificationKNN` | `fitcknn` | `loss` | `margin`

## Topics

“Classification Using Nearest Neighbors” on page 18-16

# edge

**Class:** ClassificationLinear

Classification edge for linear classification models

## Syntax

```
e = edge(Mdl, X, Y)
```

```
e = edge( ____, Name, Value)
```

## Description

`e = edge(Mdl, X, Y)` returns the classification edges on page 32-1528 for the binary, linear classification model `Mdl` using predictor data in `X` and corresponding class labels in `Y`. `e` contains a classification edge for each regularization strength in `Mdl`.

`e = edge( ____, Name, Value)` uses any of the previous syntaxes and additional options specified by one or more `Name, Value` pair arguments. For example, you can specify that columns in the predictor data correspond to observations or supply observation weights.

## Input Arguments

**mdl** — Binary, linear classification model

ClassificationLinear model object

Binary, linear classification model, specified as a `ClassificationLinear` model object. You can create a `ClassificationLinear` model object using `fitclinear`.

**x** — Predictor data

full matrix | sparse matrix

Predictor data, specified as an  $n$ -by- $p$  full or sparse matrix. This orientation of `X` indicates that rows correspond to individual observations, and columns correspond to individual predictor variables.

---

**Note** If you orient your predictor matrix so that observations correspond to columns and specify 'ObservationsIn', 'columns', then you might experience a significant reduction in computation time.

---

The length of `Y` and the number of observations in `X` must be equal.

Data Types: `single` | `double`

### **`y` — Class labels**

categorical array | character array | logical vector | vector of numeric values | cell array of character vectors

Class labels, specified as a categorical or character array, logical or numeric vector, or cell array of character vectors.

- The data type of `Y` must be the same `Mdl.ClassNames`.
- The distinct classes in `Y` must be a subset of `Mdl.ClassNames`.
- If `Y` is a character array, then each element must correspond to one row of the array.
- The length of `Y` and the number of observations in `X` must be equal.

Data Types: `categorical` | `cell` | `char` | `double` | `logical` | `single`

## **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### **ObservationsIn — Predictor data observation dimension**

'rows' (default) | 'columns'

Predictor data observation dimension, specified as the comma-separated pair consisting of 'ObservationsIn' and 'columns' or 'rows'.

---

**Note** If you orient your predictor matrix so that observations correspond to columns and specify 'ObservationsIn', 'columns', then you might experience a significant reduction in optimization-execution time.

---

Data Types: `char`

### **Weights — Observation weights**

numeric vector of positive values

Observation weights, specified as the comma-separated pair consisting of 'Weights' and a numeric vector of positive values. If you supply weights, `edge` computes the weighted classification edge on page 32-1528.

Let  $n$  be the number of observations in  $X$ .

- `numel(Weights)` must be  $n$ .
- By default, `Weights` is `ones(n,1)`.

`ClassificationLinear.edge` normalizes `Weights` to sum up to the value of the prior probability in the respective class.

Data Types: `double` | `single`

## Output Arguments

### **e — Classification edges**

numeric scalar | numeric row vector

Classification edges on page 32-1528, returned as a numeric scalar or row vector.

`e` is the same size as `Mdl.Lambda`. `e(j)` is the classification edge of the linear classification model trained using the regularization strength `Mdl.Lambda(j)`.

## Examples

### **Estimate Test-Sample Edge**

Load the NLP data set.

```
load nlpdata
```

$X$  is a sparse matrix of predictor data, and  $Y$  is a categorical vector of class labels. There are more than two classes in the data.

The models should identify whether the word counts in a web page are from the Statistics and Machine Learning Toolbox™ documentation. So, identify the labels that correspond to the Statistics and Machine Learning Toolbox™ documentation web pages.

```
Ystats = Y == 'stats';
```

Train a binary, linear classification model that can identify whether the word counts in a documentation web page are from the Statistics and Machine Learning Toolbox™ documentation. Specify to holdout 30% of the observations. Optimize the objective function using SpARSA.

```
rng(1); % For reproducibility
CVMdl = fitclinear(X,Ystats,'Solver','sparsa','Holdout',0.30);
CMdl = CVMdl.Trained{1};
```

CVMdl is a ClassificationPartitionedLinear model. It contains the property Trained, which is a 1-by-1 cell array holding a ClassificationLinear model that the software trained using the training set.

Extract the training and test data from the partition definition.

```
trainIdx = training(CVMdl.Partition);
testIdx = test(CVMdl.Partition);
```

Estimate the training- and test-sample edges.

```
eTrain = edge(CMdl,X(trainIdx,:),Ystats(trainIdx))
eTest = edge(CMdl,X(testIdx,:),Ystats(testIdx))
```

```
eTrain =
    15.6660

eTest =
    15.4767
```



## Feature Selection Using Test-Sample Edges

One way to perform feature selection is to compare test-sample edges from multiple models. Based solely on this criterion, the classifier with the highest edge is the best classifier.

Load the NLP data set.

```
load nlpdata
```

$X$  is a sparse matrix of predictor data, and  $Y$  is a categorical vector of class labels. There are more than two classes in the data.

The models should identify whether the word counts in a web page are from the Statistics and Machine Learning Toolbox™ documentation. So, identify the labels that correspond to the Statistics and Machine Learning Toolbox™ documentation web pages. For quicker execution time, orient the predictor data so that individual observations correspond to columns.

```
Ystats = Y == 'stats';
X = X';
rng(1); % For reproducibility
```

Create a data partition which holds out 30% of the observations for testing.

```
Partition = cvpartition(Ystats, 'Holdout', 0.30);
testIdx = test(Partition); % Test-set indices
XTest = X(:, testIdx);
YTest = Ystats(testIdx);
```

Partition is a `cvpartition` object that defines the data set partition.

Randomly choose half of the predictor variables.

```
p = size(X, 1); % Number of predictors
idxPart = randsample(p, ceil(0.5*p));
```

Train two binary, linear classification models: one that uses the all of the predictors and one that uses half of the predictors. Optimize the objective function using SparSA, and indicate that observations correspond to columns.

```
CVMdl = fitlinear(X, Ystats, 'CVPartition', Partition, 'Solver', 'sparsa', ...
    'ObservationsIn', 'columns');
```

```
PCVMdl = fitclinear(X(idxPart,:), Ystats, 'CVPartition', Partition, 'Solver', 'sparsa', ...  
    'ObservationsIn', 'columns');
```

CVMDL and PCVMDL are ClassificationPartitionedLinear models.

Extract the trained ClassificationLinear models from the cross-validated models.

```
CMdl = CVMDL.Trained{1};  
PCMdl = PCVMDL.Trained{1};
```

Estimate the test sample edge for each classifier.

```
fullEdge = edge(CMdl, XTest, YTest, 'ObservationsIn', 'columns')  
partEdge = edge(PCMdl, XTest(idxPart,:), YTest, 'ObservationsIn', 'columns')
```

```
fullEdge =
```

```
    15.4767
```

```
partEdge =
```

```
    13.4458
```

Based on the test-sample edges, the classifier that uses all of the predictors is the better model.

### Find Good Lasso Penalty Using Edge

To determine a good lasso-penalty strength for a linear classification model that uses a logistic regression learner, compare test-sample edges.

Load the NLP data set. Preprocess the data as in “Feature Selection Using Test-Sample Edges” on page 32-1522.

```
load nlpdata  
Ystats = Y == 'stats';  
X = X';
```

```
Partition = cvpartition(Ystats, 'Holdout', 0.30);  
testIdx = test(Partition);
```

```
XTest = X(:,testIdx);
YTest = Ystats(testIdx);
```

Create a set of 11 logarithmically-spaced regularization strengths from  $10^{-8}$  through  $10^1$ .

```
Lambda = logspace(-8,1,11);
```

Train binary, linear classification models that use each of the regularization strengths. Optimize the objective function using SpARSA. Lower the tolerance on the gradient of the objective function to  $1e-8$ .

```
rng(10); % For reproducibility
CVMdl = fitclinear(X,Ystats,'ObservationsIn','columns',...
    'CVPartition',Partition,'Learner','logistic','Solver','sparsa',...
    'Regularization','lasso','Lambda',Lambda,'GradientTolerance',1e-8)
```

```
CVMdl =
```

```
classreg.learning.partition.ClassificationPartitionedLinear
  CrossValidatedModel: 'Linear'
  ResponseName: 'Y'
  NumObservations: 31572
  KFold: 1
  Partition: [1x1 cvpartition]
  ClassNames: [0 1]
  ScoreTransform: 'none'
```

Extract the trained linear classification model.

```
Mdl = CVMdl.Trained{1}
```

```
Mdl =
```

```
ClassificationLinear
  ResponseName: 'Y'
  ClassNames: [0 1]
  ScoreTransform: 'logit'
  Beta: [34023x11 double]
  Bias: [1x11 double]
  Lambda: [1x11 double]
  Learner: 'logistic'
```

Mdl is a `ClassificationLinear` model object. Because `Lambda` is a sequence of regularization strengths, you can think of `Mdl` as 11 models, one for each regularization strength in `Lambda`.

Estimate the test-sample edges.

```
e = edge(Mdl,X(:,testIdx),Ystats(testIdx),'ObservationsIn','columns')
```

```
e =
```

```
Columns 1 through 7
```

```
    0.9986    0.9986    0.9986    0.9986    0.9986    0.9932    0.9766
```

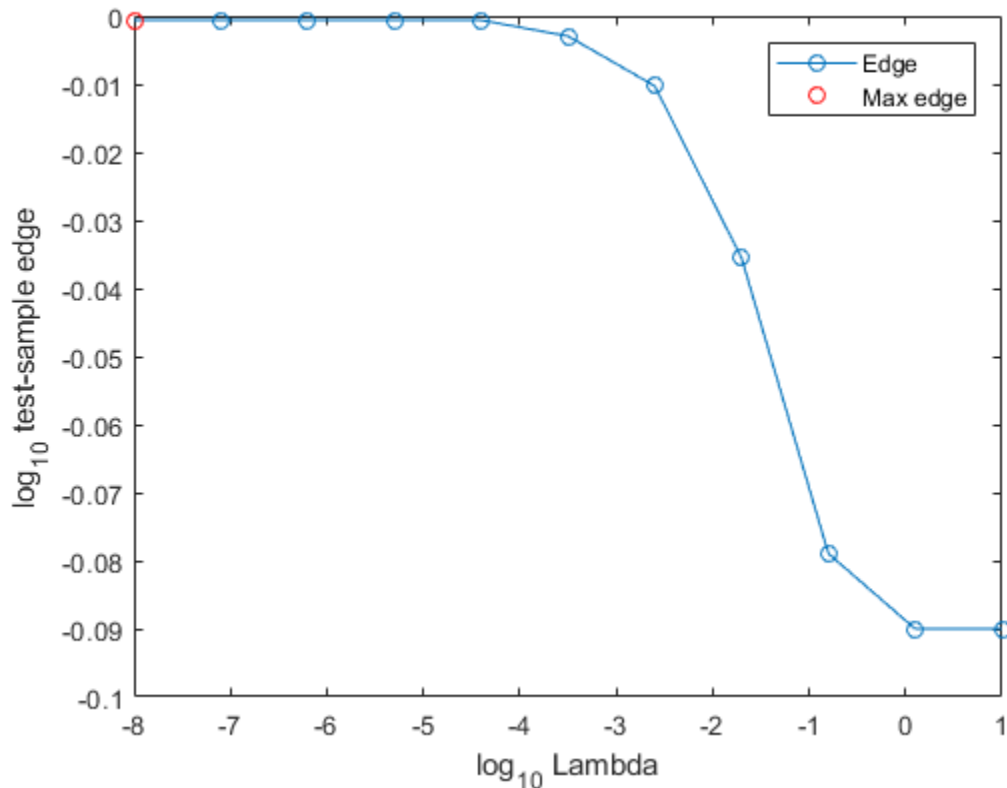
```
Columns 8 through 11
```

```
    0.9214    0.8338    0.8128    0.8128
```

Because there are 11 regularization strengths, `e` is a 1-by-11 vector of edges.

Plot the test-sample edges for each regularization strength. Identify the regularization strength that maximizes the edges over the grid.

```
figure;
plot(log10(Lambda),log10(e),'-o')
[~, maxEIdx] = max(e);
maxLambda = Lambda(maxEIdx);
hold on
plot(log10(maxLambda),log10(e(maxEIdx)),'ro');
ylabel('log_{10} test-sample edge')
xlabel('log_{10} Lambda')
legend('Edge','Max edge')
hold off
```



Several values of Lambda yield similarly high edges. Higher values of lambda lead to predictor variable sparsity, which is a good quality of a classifier.

Choose the regularization strength that occurs just before the edge starts decreasing.

```
LambdaFinal = Lambda(5);
```

Train a linear classification model using the entire data set and specify the regularization strength yielding the maximal edge.

```
MdlFinal = fitclinear(X,Ystats,'ObservationsIn','columns',...
    'Learner','logistic','Solver','sparsa','Regularization','lasso',...
    'Lambda',LambdaFinal);
```

To estimate labels for new observations, pass `MdlFinal` and the new data to `predict`.

## Definitions

### Classification Edge

The classification edge is the weighted mean of the classification margins.

One way to choose among multiple classifiers, e.g., to perform feature selection, is to choose the classifier that yields the highest edge.

### Classification Margin

The classification margins for binary classification are, for each observation, the difference between the classification score for the true class and the classification score for the false class.

The software defines the classification margin for binary classification as

$$m = 2yf(x).$$

$x$  is an observation. If the true label of  $x$  is the positive class, then  $y$  is 1, and  $-1$  otherwise.  $f(x)$  is the positive-class classification score for the observation  $x$ . The literature commonly defines the margin as  $m = yf(x)$ .

If the margins are on the same scale, then they serve as a classification confidence measure, i.e., among multiple classifiers, those that yield larger margins are better.

### Classification Score

For linear classification models, the raw classification score for classifying the observation  $x$ , a row vector, into the positive class is defined by

$$f_j(x) = x\beta_j + b_j.$$

For the model with regularization strength  $j$ ,  $\beta_j$  is the estimated column vector of coefficients (the model property `Beta(:, j)`) and  $b_j$  is the estimated, scalar bias (the model property `Bias(j)`).

The raw classification score for classifying  $x$  into the negative class is  $-f(x)$ . The software classifies observations into the class that yields the positive score.

If the linear classification model consists of logistic regression learners, then the software applies the 'logit' score transformation to the raw classification scores (see `ScoreTransform`).

## Algorithms

By default, observation weights are prior class probabilities. If you supply weights using `Weights`, then the software normalizes them to sum to the prior probabilities in the respective classes. The software uses the normalized weights to estimate the weighted edge.

## Extended Capabilities

### Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays” (MATLAB).

### See Also

`ClassificationLinear` | `fitclinear` | `margin` | `predict`

Introduced in R2016a

## edge

**Class:** CompactClassificationDiscriminant

Classification edge

## Syntax

`E = edge(obj, X, Y)`

`E = edge(obj, X, Y, Name, Value)`

## Description

`E = edge(obj, X, Y)` returns the classification edge for `obj` with data `X` and classification `Y`.

`E = edge(obj, X, Y, Name, Value)` computes the edge with additional options specified by one or more `Name, Value` pair arguments.

## Input Arguments

**obj**

Discriminant analysis classifier of class `ClassificationDiscriminant` or `CompactClassificationDiscriminant`, typically constructed with `fitcdiscr`.

**X**

Matrix where each row represents an observation, and each column represents a predictor. The number of columns in `X` must equal the number of predictors in `obj`.

**Y**

Class labels, with the same data type as exists in `obj`. The number of elements of `Y` must equal the number of rows of `X`.



## Name-Value Pair Arguments

Specify optional comma-separated pairs of *Name*, *Value* arguments. *Name* is the argument name and *Value* is the corresponding value. *Name* must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as *Name1*, *Value1*, ..., *NameN*, *ValueN*.

### **weights**

Observation weights, a numeric vector of length `size(X,1)`. If you supply weights, `edge` computes the weighted classification edge.

**Default:** `ones(size(X,1))`

## Output Arguments

### **E**

Edge, a scalar representing the weighted average value of the margin.

## Examples

Compute the classification edge and margin for the Fisher iris data, trained on its first two columns of data, and view the last 10 entries:

```
load fisheriris
X = meas(:,1:2);
obj = fitcdiscr(X,species);
E = edge(obj,X,species)

E =
    0.4980

M = margin(obj,X,species);
M(end-10:end)

ans =
    0.6551
    0.4838
```

```
0.6551
-0.5127
0.5659
0.4611
0.4949
0.1024
0.2787
-0.1439
-0.4444
```

The classifier trained on all the data is better:

```
obj = fitcdiscr(meas, species);
E = edge(obj, meas, species)
```

```
E =
    0.9454
```

```
M = margin(obj, meas, species);
M(end-10:end)
```

```
ans =
    0.9983
    1.0000
    0.9991
    0.9978
    1.0000
    1.0000
    0.9999
    0.9882
    0.9937
    1.0000
    0.9649
```

## Definitions

### Edge

The edge is the weighted mean value of the classification margin. The weights are class prior probabilities. If you supply additional weights, those weights are normalized to sum to the prior probabilities in the respective classes, and are then used to compute the weighted average.

## Margin

The classification margin is the difference between the classification score for the true class and maximal classification score for the false classes.

The classification margin is a column vector with the same number of rows as in the matrix  $X$ . A high value of margin indicates a more reliable prediction than a low value.

## Score (discriminant analysis)

For discriminant analysis, the score of a classification is the posterior probability of the classification. For the definition of posterior probability in discriminant analysis, see “Posterior Probability” on page 20-8.

# Extended Capabilities

## Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays” (MATLAB).

## See Also

`ClassificationDiscriminant` | `fitcdiscr` | `loss` | `margin` | `predict`

## Topics

“Discriminant Analysis Classification” on page 20-2

## edge

**Class:** CompactClassificationECOC

Classification edge for multiclass, error-correcting output codes model

## Syntax

```
e = edge(Mdl, tbl, ResponseVarName)
```

```
e = edge(Mdl, tbl, Y)
```

```
e = edge(Mdl, X, Y)
```

```
e = edge( ____, Name, Value)
```

## Description

`e = edge(Mdl, tbl, ResponseVarName)` returns the classification edge on page 32-1544 (e) for the error-correcting output code (ECOC) multiclass classifier `Mdl` using predictor data in table `tbl` and class labels `tbl.ResponseVarName`. Each row of `tbl` and `ResponseVarName` is an observation.

`e = edge(Mdl, tbl, Y)` returns the classification edge for the classifier `Mdl` using predictor data in table `tbl` and class labels in vector `Y`. Each row of `tbl` and `Y` is an observation.

`e = edge(Mdl, X, Y)` returns the classification edge (e) for the classifier `Mdl` using predictor data in matrix `X` and class labels in vector `Y`. Each row of `X` and `Y` is an observation.

`e = edge( ____, Name, Value)` computes the classification edge with additional options specified by one or more `Name, Value` pair arguments, using any of the previous syntaxes.

For example, specify a decoding scheme, binary learner loss function, or verbosity level.

## Input Arguments

### **mdl** — Full or compact, multiclass ECOC model

`ClassificationECOC` model object | `CompactClassificationECOC` model object

Full or compact, multiclass ECOC model, specified as a `ClassificationECOC` or `CompactClassificationECOC` model object.

To create a full or compact ECOC model, see `ClassificationECOC` or `CompactClassificationECOC`.

### **tbl** — Sample data

table

Sample data, specified as a table. Each row of `tbl` corresponds to one observation, and each column corresponds to one predictor variable. Optionally, `tbl` can contain additional columns for the response variable and observation weights. `tbl` must contain all the predictors used to train `Mdl`. Multi-column variables and cell arrays other than cell arrays of character vectors are not allowed.

If you trained `Mdl` using sample data contained in a table, then the input data for this method must also be in a table.

---

**Note** If `Mdl.BinaryLearners` contains linear classification models (that is, `ClassificationLinear` model objects), then you cannot specify sample data in a table. Instead, pass a full or sparse matrix (`X`) and class labels (`Y`).

---

Data Types: table

### **ResponseVarName** — Response variable name

name of a variable in `tbl`

Response variable name, specified as the name of a variable in `tbl`.

You must specify `ResponseVarName` as a character vector. For example, if the response variable `y` is stored as `tbl.y`, then specify it as `'y'`. Otherwise, the software treats all columns of `tbl`, including `y`, as predictors when training the model.

The response variable must be a categorical or character array, logical or numeric vector, or cell array of character vectors. If the response variable is a character array, then each element must correspond to one row of the array.

Data Types: `char`

### **x — Predictor data**

numeric matrix

Predictor data, specified as a numeric matrix.

Each row of `X` corresponds to one observation, and each column corresponds to one variable. The variables composing the columns of `X` should be the same as the variables that trained the `Mdl` classifier.

The length of `Y` and the number of rows of `X` must be equal.

Data Types: `double` | `single`

### **y — Class labels**

categorical array | character array | logical vector | vector of numeric values | cell array of character vectors

Class labels, specified as a categorical or character array, logical or numeric vector, or cell array of character vectors. `Y` must be the same as the data type of `Mdl.ClassNames`.

The length of `Y` and the number of rows of `X` must be equal.

## **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### **BinaryLoss — Binary learner loss function**

'hamming' | 'linear' | 'logit' | 'exponential' | 'binodeviance' | 'hinge' | 'quadratic' | function handle

Binary learner loss function, specified as the comma-separated pair consisting of 'BinaryLoss' and a built-in, loss-function name or function handle.

- This table contains names and descriptions of the built-in functions, where  $y_j$  is a class label for a particular binary learner (in the set  $\{-1, 1, 0\}$ ),  $s_j$  is the score for observation  $j$ , and  $g(y_j, s_j)$  is the binary loss formula.

Value	Description	Score Domain	$g(y_j, s_j)$
'binodeviance'	Binomial deviance	$(-\infty, \infty)$	$\log[1 + \exp(-2y_j s_j)] / [2\log(2)]$
'exponential'	Exponential	$(-\infty, \infty)$	$\exp(-y_j s_j) / 2$
'hamming'	Hamming	$[0, 1]$ or $(-\infty, \infty)$	$[1 - \text{sign}(y_j s_j)] / 2$
'hinge'	Hinge	$(-\infty, \infty)$	$\max(0, 1 - y_j s_j) / 2$
'linear'	Linear	$(-\infty, \infty)$	$(1 - y_j s_j) / 2$
'logit'	Logistic	$(-\infty, \infty)$	$\log[1 + \exp(-y_j s_j)] / [2\log(2)]$
'quadratic'	Quadratic	$[0, 1]$	$[1 - y_j(2s_j - 1)]^2 / 2$

The software normalizes the binary losses such that the loss is 0.5 when  $y_j = 0$ . Also, the software calculates the mean binary loss for each class.

- For a custom binary loss function, e.g., `customFunction`, specify its function handle `'BinaryLoss', @customFunction`.

`customFunction` should have this form:

```
bLoss = customFunction(M, s)
```

where:

- `M` is the  $K$ -by- $L$  coding matrix stored in `Mdl.CodingMatrix`.
- `s` is the 1-by- $L$  row vector of classification scores.
- `bLoss` is the classification loss. This scalar aggregates the binary losses for every learner in a particular class. For example, you can use the mean binary loss to aggregate the loss over the learners for each class.
- $K$  is the number of classes.
- $L$  is the number of binary learners.

For an example of passing a custom binary loss function, see “Predict Test-Sample Labels of ECOC Models Using Custom Binary Loss Function” on page 32-4933.

By default, if all binary learners are:

- SVMs or linear classification models of SVM learners, then `BinaryLoss` is 'hinge'
- Ensembles trained by `AdaboostM1` or `GentleBoost`, then `BinaryLoss` is 'exponential'
- Ensembles trained by `LogitBoost`, then `BinaryLoss` is 'binodeviance'
- Linear classification models of logistic regression learners, or you specify to predict class posterior probabilities (i.e., set 'FitPosterior', 1 in `fitcecoc`), then `BinaryLoss` is 'quadratic'

Otherwise, the default `BinaryLoss` is 'hamming'. To check the default value, use dot notation to display the `BinaryLoss` property of the trained model at the command line.

Example: `'BinaryLoss', 'binodeviance'`

Data Types: `char` | `function_handle`

#### **Decoding — Decoding scheme**

`'lossweighted'` (default) | `'lossbased'`

Decoding scheme that aggregates the binary losses on page 32-4941, specified as the comma-separated pair consisting of 'Decoding' and 'lossweighted' or 'lossbased'.

Example: `'Decoding', 'lossbased'`

Data Types: `char`

#### **ObservationsIn — Predictor data observation dimension**

`'rows'` (default) | `'columns'`

Predictor data observation dimension, specified as the comma-separated pair consisting of 'ObservationsIn' and 'columns' or 'rows'. `Mdl.BinaryLearners` must contain linear classification models.

---

**Note** If you orient your predictor matrix so that observations correspond to columns and specify 'ObservationsIn', 'columns', you can experience a significant reduction in execution time.

---

Data Types: `char`



**Options — Estimation options**

`[]` (default) | structure array returned by `statset`

Estimation options, specified as the comma-separated pair consisting of 'Options' and a structure array returned by `statset`.

To invoke parallel computing:

- You need a Parallel Computing Toolbox license.
- Specify 'Options', `statset('UseParallel',1)`.

**Verbose — Verbosity level**

`0` (default) | `1`

Verbosity level, specified as the comma-separated pair consisting of 'Verbose' and `0` or `1`. Verbose controls the amount of diagnostic messages that the software displays in the Command Window.

If `Verbose` is `0`, then the software does not display diagnostic messages. Otherwise, the software displays diagnostic messages.

Example: 'Verbose', `1`

Data Types: `single` | `double`

**Weights — Observation weights**

`ones(size(X,1))` (default) | numeric vector | name of a variable in `tbl`

Observation weights, specified as the comma-separated pair consisting of 'Weights' and a numeric vector or the name of a variable in `tbl`. If you supply weights, `edge` computes the weighted classification edge on page 32-1544.

`Weights` requires the same length as the number of observations in `X` or `tbl`. The software normalizes `Weights` to sum up to the value of the prior probability in the respective class.

If you specify `Weights` as the name of a variable in `tbl`, you must do so as a character vector. For example, if the weights are stored as `tbl.w`, then specify it as 'w'. Otherwise, the software treats all columns of `tbl`, including `tbl.w`, as predictors.

If `Mdl.BinaryLearners` contains linear classification models, then you must specify a vector.

## Output Arguments

### **e** — Classification edge

numeric scalar | numeric vector

Classification edge on page 32-1544, returned as a numeric scalar or vector. *e* represents the (weighted) mean of the classification margins on page 32-1544.

If `Mdl.BinaryLearners` contains linear classification models, then *e* is a 1-by-*L* vector, where *L* is the number of regularization strengths in the linear classification models (i.e., `numel(Mdl.BinaryLearners{1}.Lambda)`). *e*(*j*) is the edge for the model trained using regularization strength `Mdl.BinaryLearners{1}.Lambda(j)`.

Otherwise, *e* is a scalar.

## Examples

### Estimate the Test-Sample Edge of ECOC Models

Load Fisher's iris data set.

```
load fisheriris
X = meas;
Y = categorical(species);
classOrder = unique(Y); % Class order
rng(1); % For reproducibility
```

Train an ECOC model using SVM binary classifiers and specify a 30% holdout sample. It is good practice to standardize the predictors and define the class order. Specify to standardize the predictors using an SVM template.

```
t = templateSVM('Standardize',1);
CVMdl = fitcecoc(X,Y,'Holdout',0.30,'Learners',t,'ClassNames',classOrder);
CMdl = CVMdl.Trained{1}; % Extract trained, compact classifier
testInds = test(CVMdl.Partition); % Extract the test indices
XTest = X(testInds,:);
YTest = Y(testInds,:);
```

CVMdl is a ClassificationPartitionedECOC model. It contains the property Trained, which is a 1-by-1 cell array holding a CompactClassificationECOC model that the software trained using the training set.

Estimate the test-sample edge.

```
e = edge(CMdl, XTest, YTest)
```

```
e =
```

```
0.4574
```

The estimated test sample margin average is approximately 0.45.

### Estimate the Test-Sample Weighted Margin Mean of ECOC Models

Load Fisher's iris data set.

```
load fisheriris
X = meas;
Y = categorical(species);
classOrder = unique(Y); % Class order
rng(1); % For reproducibility
```

Suppose that the observations were measured sequentially, and that the last 75 observations were better quality because of a technology upgrade. One way to incorporate this advancement is to weigh the better-quality observations more than the other observations.

Define a weight vector that weights the better-quality observations two times greater than the other observations.

```
n = size(X,1);
weights = [ones(n-75,1); 2*ones(75,1)];
```

Train an ECOC model using SVM binary classifiers and specify a 30% holdout sample and the weighting scheme. It is good practice to standardize the predictors and define the class order. Specify to standardize the predictors using an SVM template.

```
t = templateSVM('Standardize',1);
CVMdl = fitcecoc(X,Y,'Holdout',0.30,'Learners',t,...
```

```
    'Weights',weights,'ClassNames',classOrder);
CVMdl = CVMdl.Trained{1};           % Extract trained, compact classifier
testInds = test(CVMdl.Partition); % Extract the test indices
XTest = X(testInds,:);
YTest = Y(testInds,:);
wTest = weights(testInds,:);
```

CVMdl is a trained ClassificationPartitionedECOC model. It contains the property Trained, which is a 1-by-1 cell array holding a CompactClassificationECOC classifier that the software trained using the training set.

Estimate the test-sample weighted edge using the weighting scheme.

```
e = edge(CMdl,XTest,YTest,'Weights',wTest)

e =

    0.4798
```

The test sample weighted average margin is approximately 0.48.

### Select ECOC Model Features by Comparing Test-Sample Edges

The classifier edge measures the average of the classifier margins. One way to perform feature selection is to compare test-sample edges from multiple models. Based solely on this criterion, the classifier with the highest edge is the best classifier.

Load Fisher's iris data set.

```
load fisheriris
X = meas;
Y = categorical(species);
classOrder = unique(Y); % Class order
rng(1); % For reproducibility
```

Partition the data set into training and test sets. Specify a 30% holdout sample for testing.

```
Partition = cvpartition(Y,'Holdout',0.30);
testInds = test(Partition); % Indices for the test set
```

```
XTest = X(testInds, :);
YTest = Y(testInds, :);
```

Partition defines the data set partition.

Define these two data sets:

- fullX contains all predictors.
- partX contains the petal dimensions.

```
fullX = X;
partX = X(:, 3:4);
```

Train an ECOC model using SVM binary classifiers for each predictor set, and specify the partition definition. It is good practice to standardize the predictors and define the class order. Specify to standardize the predictors using an SVM template.

```
t = templateSVM('Standardize', 1);
CVMdl = fitcecoc(fullX, Y, 'CVPartition', Partition, 'Learners', t, ...
    'ClassNames', classOrder);
PCVMdl = fitcecoc(partX, Y, 'CVPartition', Partition, 'Learners', t, ...
    'ClassNames', classOrder);
CMdl = CVMdl.Trained{1};
PCMdl = PCVMdl.Trained{1};
```

CVMdl and PCVMdl are ClassificationPartitionedECOC models. They contain the property Trained, which is a 1-by-1 cell array holding a CompactClassificationECOC model that the software trained using the training set.

Estimate the test sample edge for each classifier.

```
fullEdge = edge(CMdl, XTest, YTest)
partEdge = edge(PCMdl, XTest(:, 3:4), YTest)
```

```
fullEdge =
```

```
    0.4574
```

```
partEdge =
```

```
    0.4839
```

PCMD1 achieves an edge that resembles the more complex model CMD1.

- “Quick Start Parallel Computing for Statistics and Machine Learning Toolbox” on page 30-2

## Definitions

### Classification Edge

The classification edge is the weighted mean of the classification margins.

One way to choose among multiple classifiers, e.g., to perform feature selection, is to choose the classifier that yields the highest edge.

### Classification Margin

The classification margins are, for each observation, the difference between the negative loss for the true class and maximal negative loss among the false classes. If the margins are on the same scale, then they serve as a classification confidence measure, i.e., among multiple classifiers, those that yield larger margins are better.

### Binary Loss

A binary loss is a function of the class and classification score that determines how well a binary learner classifies an observation into the class.

Let:

- $m_{kj}$  be element  $(k,j)$  of the coding design matrix  $M$  (i.e., the code corresponding to class  $k$  of binary learner  $j$ )
- $s_j$  be the score of binary learner  $j$  for an observation
- $g$  be the binary loss function
- $\hat{k}$  be the predicted class for the observation

In loss-based decoding [15], the class producing the minimum sum of the binary losses over binary learners determines the predicted class of an observation, that is,

$$\hat{k} = \operatorname{argmin}_k \sum_{j=1}^L |m_{kj}| g(m_{kj}, s_j).$$

In loss-weighted decoding [15], the class producing the minimum average of the binary losses over binary learners determines the predicted class of an observation, that is,

$$\hat{k} = \operatorname{argmin}_k \frac{\sum_{j=1}^L |m_{kj}| g(m_{kj}, s_j)}{\sum_{j=1}^L |m_{kj}|}.$$

Allwein et al. [2] suggest that loss-weighted decoding improves classification accuracy by keeping loss values for all classes in the same dynamic range.

This table summarizes the supported loss functions, where  $y_j$  is a class label for a particular binary learner (in the set  $\{-1, 1, 0\}$ ),  $s_j$  is the score for observation  $j$ , and  $g(y_j, s_j)$ .

Value	Description	Score Domain	$g(y_j, s_j)$
'binodeviance'	Binomial deviance	$(-\infty, \infty)$	$\log[1 + \exp(-2y_j s_j)] / [2\log(2)]$
'exponential'	Exponential	$(-\infty, \infty)$	$\exp(-y_j s_j) / 2$
'hamming'	Hamming	$[0, 1]$ or $(-\infty, \infty)$	$[1 - \operatorname{sign}(y_j s_j)] / 2$
'hinge'	Hinge	$(-\infty, \infty)$	$\max(0, 1 - y_j s_j) / 2$
'linear'	Linear	$(-\infty, \infty)$	$(1 - y_j s_j) / 2$
'logit'	Logistic	$(-\infty, \infty)$	$\log[1 + \exp(-y_j s_j)] / [2\log(2)]$
'quadratic'	Quadratic	$[0, 1]$	$[1 - y_j(2s_j - 1)]^2 / 2$

The software normalizes the binary losses such that the loss is 0.5 when  $y_j = 0$ , and aggregates using the average of the binary learners [2].

Do not confuse the binary loss with the overall classification loss (specified by the `LossFun` name-value pair argument of `predict` and `loss`), e.g., classification error, which measures how well an ECOC classifier performs as a whole.

## Tip

To compare margins or edges of several classifiers, use template objects to specify a common score transform function among the classifiers when you train them using `fitcecoc`.

## Algorithms

If you trained `Mdl` specifying to standardize the predictor data, then the software standardizes the columns of `X` using the corresponding means and standard deviations that the software stored in `Mdl.BinaryLearner{j}.Mu` and `Mdl.BinaryLearner{j}.Sigma` for learner `j`.

## References

- [1] Allwein, E., R. Schapire, and Y. Singer. “Reducing multiclass to binary: A unifying approach for margin classifiers.” *Journal of Machine Learning Research*. Vol. 1, 2000, pp. 113–141.
- [2] Escalera, S., O. Pujol, and P. Radeva. “On the decoding process in ternary error-correcting output codes.” *IEEE Transactions on Pattern Analysis and Machine Intelligence*. Vol. 32, Issue 7, 2010, pp. 120–134.
- [3] Escalera, S., O. Pujol, and P. Radeva. “Separability of ternary codes for sparse designs of error-correcting output codes.” *Pattern Recogn.* Vol. 30, Issue 3, 2009, pp. 285–297.

## Extended Capabilities

### Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays” (MATLAB).



## See Also

`ClassificationECOC` | `ClassificationLinear` | `CompactClassificationECOC` | `fitcecoc` | `margin` | `predict` | `resubEdge`

## Topics

“Quick Start Parallel Computing for Statistics and Machine Learning Toolbox” on page 30-2

“Reproducibility in Parallel Statistical Computations” on page 30-14

“Concepts of Parallel Computing in Statistics and Machine Learning Toolbox” on page 30-8

## edge

**Class:** CompactClassificationEnsemble

Classification edge

## Syntax

```
E = edge(ens, tbl, ResponseVarName)
E = edge(ens, tbl, Y)
E = edge(ens, X, Y)
E = edge( ____, Name, Value)
```

## Description

`E = edge(ens, tbl, ResponseVarName)` returns the classification edge for `ens` with data `tbl` and classification `tbl.ResponseVarName`.

`E = edge(ens, tbl, Y)` returns the classification edge for `ens` with data `tbl` and classification `Y`.

`E = edge(ens, X, Y)` returns the classification edge for `ens` with data `X` and classification `Y`.

`E = edge( ____, Name, Value)` computes the edge with additional options specified by one or more `Name, Value` pair arguments, using any of the previous syntaxes.

## Input Arguments

### **ens**

A classification ensemble constructed with `fitcensemble`, or a compact classification ensemble constructed with `compact`.

**tbl**

Sample data, specified as a table. Each row of `tbl` corresponds to one observation, and each column corresponds to one predictor variable. `tbl` must contain all of the predictors used to train the model. Multi-column variables and cell arrays other than cell arrays of character vectors are not allowed.

If you trained `ens` using sample data contained in a table, then the input data for this method must also be in a table.

**ResponseVarName**

Response variable name, specified as the name of a variable in `tbl`. The response variable must be a numeric vector.

You must specify `ResponseVarName` as a character vector. For example, if the response variable `Y` is stored as `tbl.Y`, then specify it as `'Y'`. Otherwise, the software treats all columns of `tbl`, including `Y`, as predictors when training the model.

**X**

A matrix where each row represents an observation, and each column represents a predictor. The number of columns in `X` must equal the number of predictors in `ens`.

If you trained `ens` using sample data contained in a matrix, then the input data for this method must also be in a matrix.

**Y**

Class labels, with the same data type as exists in `ens`. The number of elements of `Y` must equal the number of rows of `tbl` or `X`.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

**learners**

Indices of weak learners in the ensemble ranging from 1 to `ens.NumTrained`. `edge` uses only these learners for calculating loss.

**Default:** `1:NumTrained`

**mode**

Meaning of the output `E`:

- `'ensemble'` — `E` is a scalar value, the edge for the entire ensemble.
- `'individual'` — `E` is a vector with one element per trained learner.
- `'cumulative'` — `E` is a vector in which element `J` is obtained by using learners `1:J` from the input list of learners.

**Default:** `'ensemble'`

**UseObsForLearner**

A logical matrix of size `N-by-T`, where:

- `N` is the number of rows of `X`.
- `T` is the number of weak learners in `ens`.

When `UseObsForLearner(i,j)` is true, learner `j` is used in predicting the class of row `i` of `X`.

**Default:** `true(N,T)`

**weights**

Observation weights, a numeric vector of length `size(X,1)`. If you supply weights, `edge` computes weighted classification edge.

**Default:** `ones(size(X,1))`

## Output Arguments

### E

The classification edge, a vector or scalar depending on the setting of the mode name-value pair. Classification edge is weighted average classification margin.

## Examples

### Find Classification Edge of Training Data

Find the classification edge for some of the data used to train a boosted ensemble classifier.

Load the ionosphere data set.

```
load ionosphere
```

Train an ensemble of 100 boosted classification trees using AdaBoostM1.

```
t = templateTree('MaxNumSplits',1); % Weak learner template tree object
ens = fitcensemble(X,Y,'Method','AdaBoostM1','Learners',t);
```

Find the classification edge for the last few rows.

```
E = edge(ens,X(end-10:end,:),Y(end-10:end))
```

```
E = 8.3310
```

## Definitions

### Margin

The classification margin is the difference between the classification score for the true class and maximal classification score for the false classes. Margin is a column vector with the same number of rows as in the matrix X.

## Score (ensemble)

For ensembles, a classification score represents the confidence of a classification into a class. The higher the score, the higher the confidence.

Different ensemble algorithms have different definitions for their scores. Furthermore, the range of scores depends on ensemble type. For example:

- `AdaBoostM1` scores range from  $-\infty$  to  $\infty$ .
- `Bag` scores range from 0 to 1.

## Edge

The edge is the weighted mean value of the classification margin. The weights are the class probabilities in `ens.Prior`. If you supply weights in the `weights` name-value pair, those weights are used instead of class probabilities.

# Extended Capabilities

## Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays” (MATLAB).

## See Also

`edge` | `margin`

## edge

**Class:** CompactClassificationNaiveBayes

Classification edge for naive Bayes classifiers

## Syntax

```
e = edge (Mdl, tbl, ResponseVarName)
```

```
e = edge (Mdl, tbl, Y)
```

```
e = edge (Mdl, X, Y)
```

```
e = edge ( ____, Name, Value)
```

## Description

`e = edge (Mdl, tbl, ResponseVarName)` returns the classification edge on page 32-1560 (e) for the naive Bayes classifier `Mdl` using the predictor data in table `tbl` and the class labels in `tbl.ResponseVarName`.

`e = edge (Mdl, tbl, Y)` returns the classification edge (e) for the naive Bayes classifier `Mdl` using predictor data in table `tbl` and the class labels in vector `Y`.

`e = edge (Mdl, X, Y)` returns the classification edge on page 32-1560 (e) for the naive Bayes classifier `Mdl` using predictor data `X` and class labels `Y`.

`e = edge ( ____, Name, Value)` computes the classification edge with additional options specified by one or more `Name, Value` pair arguments, using any of the previous syntaxes.

## Input Arguments

**mdl** — Naive Bayes classifier

ClassificationNaiveBayes model | CompactClassificationNaiveBayes model

Naive Bayes classifier, specified as a `ClassificationNaiveBayes` model or `CompactClassificationNaiveBayes` model returned by `fitcnb` or `compact`, respectively.

**tbl** — Sample data

table

Sample data, specified as a table. Each row of `tbl` corresponds to one observation, and each column corresponds to one predictor variable. Optionally, `tbl` can contain additional columns for the response variable and observation weights. `tbl` must contain all the predictors used to train `Mdl`. Multi-column variables and cell arrays other than cell arrays of character vectors are not allowed.

If you trained `Mdl` using sample data contained in a table, then the input data for this method must also be in a table.

Data Types: `table`

**ResponseVarName** — Response variable name

name of a variable in `tbl`

Response variable name, specified as the name of a variable in `tbl`.

You must specify `ResponseVarName` as a character vector. For example, if the response variable `y` is stored as `tbl.y`, then specify it as `'y'`. Otherwise, the software treats all columns of `tbl`, including `y`, as predictors when training the model.

The response variable must be a categorical or character array, logical or numeric vector, or cell array of character vectors. If the response variable is a character array, then each element must correspond to one row of the array.

**x** — Predictor data

numeric matrix

Predictor data, specified as a numeric matrix.

Each row of `X` corresponds to one observation (also known as an instance or example), and each column corresponds to one variable (also known as a feature). The variables making up the columns of `X` should be the same as the variables that trained `Mdl`.

The length of `Y` and the number of rows of `X` must be equal.



Data Types: `double` | `single`

### **Y — Class labels**

categorical array | character array | logical vector | vector of numeric values | cell array of character vectors

Class labels, specified as a categorical or character array, logical or numeric vector, or cell array of character vectors. `Y` must be the same as the data type of `Mdl.ClassNames`.

The length of `Y` and the number of rows of `tbl` or `X` must be equal.

## **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### **Weights — Observation weights**

`ones(size(X,1),1)` (default) | numeric vector | name of a variable in `tbl`

Observation weights, specified as the comma-separated pair consisting of 'Weights' and a numeric vector or the name of a variable in `tbl`. The software weighs the observations in each row of `X` or `tbl` with the corresponding weight in `Weights`.

If you specify `Weights` as a vector, then the size of `Weights` must be equal to the number of rows of `X` or `tbl`.

If you specify `Weights` as the name of a variable in `tbl`, you must do so as a character vector. For example, if the weights are stored as `tbl.w`, then specify it as 'w'. Otherwise, the software treats all columns of `tbl`, including `tbl.w`, as predictors.

If you do not specify your own loss function, then the software normalizes `Weights` to add up to 1.

Data Types: `double`

## Output Arguments

### **e** — Classification edge

scalar

Classification edge on page 32-1560, returned as a scalar. If you supply `Weights`, then `e` is the weighted classification edge.

## Examples

### Estimate the Test Sample Edge of Naive Bayes Classifiers

Load Fisher's iris data set.

```
load fisheriris
X = meas;      % Predictors
Y = species;  % Response
rng(1);       % For reproducibility
```

Train a naive Bayes classifier. Specify a 30% holdout sample for testing. It is good practice to specify the class order. Assume that each predictor is conditionally, normally distributed given its label.

```
CVMDL = fitcnb(X,Y,'Holdout',0.30,...
    'ClassNames',{'setosa','versicolor','virginica'});
CMDL = CVMDL.Trained{1};           % Extract trained, compact classifier
testInds = test(CVMDL.Partition); % Extract the test indices
XTest = X(testInds,:);
YTest = Y(testInds);
```

`CVMDL` is a `ClassificationPartitionedModel` classifier. It contains the property `Trained`, which is a 1-by-1 cell array holding a `CompactClassificationNaiveBayes` classifier that the software trained using the training set.

Estimate the test sample edge.

```
e = edge(CMDL,XTest,YTest)
```

```
e =
```

0.8244

The estimated test sample margin average is approximately 0.82. This indicates that, on average, the test sample difference between the estimated posterior probability for the predicted class and the posterior probability for the class with the next lowest posterior probability is approximately 0.82. This indicates that the classifier labels with high confidence.

### Estimate the Test Sample Weighted Margin Mean of Naive Bayes Classifiers

Load Fisher's iris data set.

```
load fisheriris
X = meas;      % Predictors
Y = species;  % Response
rng(1);
```

Suppose that the setosa iris measurements are lower quality because they were measured with an older technology. One way to incorporate this is to weigh the setosa iris measurements less than the other observations.

Define a weight vector that weighs the better quality observations twice the other observations.

```
n = size(X,1);
idx = strcmp(Y,'setosa');
weights = ones(size(X,1),1);
weights(idx) = 0.5;
```

Train a naive Bayes classifier. Specify the weighting scheme and a 30% holdout sample for testing. It is good practice to specify the class order. Assume that each predictor is conditionally, normally distributed given its label.

```
CVMDL = fitcnb(X,Y,'Weights',weights,'Holdout',0.30,...
    'ClassNames',{'setosa','versicolor','virginica'});
CMDL = CVMDL.Trained{1}; % Extract trained, compact classifier
testInds = test(CVMDL.Partition); % Extract the test indices
XTest = X(testInds,:);
YTest = Y(testInds);
wTest = weights(testInds);
```

`CVMDL` is a `ClassificationPartitionedModel` classifier. It contains the property `Trained`, which is a 1-by-1 cell array holding a `CompactClassificationNaiveBayes` classifier that the software trained using the training set.

Estimate the test sample weighted edge using the weighting scheme.

```
e = edge(CMDL, XTest, YTest, 'Weights', wTest)
```

```
e =
```

```
0.7893
```

The test sample weighted average margin is approximately 0.79. This indicates that, on average, the test sample difference between the estimated posterior probability for the predicted class and the posterior probability for the class with the next lowest posterior probability is approximately 0.79. This indicates that the classifier labels with high confidence.

### Select Naive Bayes Classifier Features by Comparing Test Sample Edges

The classifier edge measures the average of the classifier margins. One way to perform feature selection is to compare test sample edges from multiple models. Based solely on this criterion, the classifier with the highest edge is the best classifier.

Load Fisher's iris data set.

```
load fisheriris
X = meas;      % Predictors
Y = species;  % Response
rng(1);
```

Partition the data set into training and test sets. Specify a 30% holdout sample for testing.

```
Partition = cvpartition(Y, 'Holdout', 0.30);
testInds = test(Partition); % Indices for the test set
XTest = X(testInds, :);
YTest = Y(testInds, :);
```

`Partition` defines the data set partition.

Define these two data sets:

- `fullX` contains all predictors.
- `partX` contains the last two predictors.

```
fullX = X;  
partX = X(:,3:4);
```

Train naive Bayes classifiers for each predictor set. Specify the partition definition.

```
FCVMdl = fitcnb(fullX,Y,'CVPartition',Partition);  
PCVMdl = fitcnb(partX,Y,'CVPartition',Partition);  
FCMdl = FCVMdl.Trained{1};  
PCMdl = PCVMdl.Trained{1};
```

`FCVMdl` and `PCVMdl` are `ClassificationPartitionedModel` classifiers. They contain the property `Trained`, which is a 1-by-1 cell array holding a `CompactClassificationNaiveBayes` classifier that the software trained using the training set.

Estimate the test sample edge for each classifier.

```
fullEdge = edge(FCMdl,XTest,YTest)  
partEdge = edge(PCMdl,XTest(:,3:4),YTest)
```

```
fullEdge =
```

```
    0.8244
```

```
partEdge =
```

```
    0.8420
```

The test-sample edges of the classifiers are nearly the same. However, the model trained using two predictors (`PCmdl`) is less complex.

## Definitions

### Classification Edge

The classification edge is the weighted mean of the classification margins.

If you supply weights, then the software normalizes them to sum to the prior probability of their respective class. The software uses the normalized weights to compute the weighted mean.

One way to choose among multiple classifiers, e.g., to perform feature selection, is to choose the classifier that yields the highest edge.

### Classification Margins

The classification margins are, for each observation, the difference between the score for the true class and maximal score for the false classes. Provided that they are on the same scale, margins serve as a classification confidence measure, i.e., among multiple classifiers, those that yield larger margins are better.

### Posterior Probability

The posterior probability is the probability that an observation belongs in a particular class, given the data.

For naive Bayes, the posterior probability that a classification is  $k$  for a given observation  $(x_1, \dots, x_p)$  is

$$\hat{P}(Y = k | x_1, \dots, x_p) = \frac{P(X_1, \dots, X_p | y = k)\pi(Y = k)}{P(X_1, \dots, X_p)},$$

where:

- $P(X_1, \dots, X_p | y = k)$  is the conditional joint density of the predictors given they are in class  $k$ . `Mdl.DistributionNames` stores the distribution names of the predictors.

- $\pi(Y = k)$  is the class prior probability distribution. `Mdl.Prior` stores the prior distribution.
- $P(X_1, \dots, X_P)$  is the joint density of the predictors. The classes are discrete, so

$$P(X_1, \dots, X_P) = \sum_{k=1}^K P(X_1, \dots, X_P | y = k) \pi(Y = k).$$

## Prior Probability

The prior probability is the believed relative frequency that observations from a class occur in the population for each class.

## Score

The naive Bayes score is the class posterior probability given the observation.

# Extended Capabilities

## Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays” (MATLAB).

## See Also

`ClassificationNaiveBayes` | `CompactClassificationNaiveBayes` | `fitcnb` | `loss` | `margin` | `predict` | `resubEdge` | `resubLoss`

## Topics

“Naive Bayes Classification” on page 21-2

## edge

**Class:** CompactClassificationSVM

Classification edge for support vector machine classifiers

## Syntax

```
e = edge (SVMModel, TBL, ResponseVarName)
e = edge (SVMModel, TBL, Y)
e = edge ( ____, Name, Value)

e = edge (SVMModel, X, Y)
e = edge ( ____, Name, Value)
```

## Description

`e = edge (SVMModel, TBL, ResponseVarName)` returns the classification edge on page 32-1568 (e) for the support vector machine (SVM) classifier `SVMModel` using the predictor data in table `TBL` and class labels in `TBL.ResponseVarName`.

`e = edge (SVMModel, TBL, Y)` returns the classification edge on page 32-1568 (e) for the support vector machine (SVM) classifier `SVMModel` using the predictor data in table `TBL` and class labels in `Y`.

`e = edge ( ____, Name, Value)` computes the classification edge with additional options specified by one or more `Name, Value` pair arguments, using any of the previous syntaxes. For example, you can specify observation weights.

`e = edge (SVMModel, X, Y)` returns the classification edge for `SVMModel` using the predictor data in matrix `X` and class labels in `Y`.

`e = edge ( ____, Name, Value)` computes the classification edge with additional options specified by one or more `Name, Value` pair arguments, using any of the previous syntaxes. For example, you can specify observation weights.



## Input Arguments

### **SVMMoDel** — SVM classification model

ClassificationSVM model object | CompactClassificationSVM model object

SVM classification model, specified as a ClassificationSVM model object or CompactClassificationSVM model object returned by `fitcsvm` or `compact`, respectively.

### **TBL** — Sample data

table

Sample data, specified as a table. Each row of TBL corresponds to one observation, and each column corresponds to one predictor variable. Optionally, TBL can contain additional columns for the response variable and observation weights. TBL must contain all of the predictors used to train SVMMoDel. Multi-column variables and cell arrays other than cell arrays of character vectors are not allowed.

If TBL contains the response variable used to train SVMMoDel, then you do not need to specify ResponseVarName or Y.

If you trained SVMMoDel using sample data contained in a table, then the input data for this method must also be in a table.

Data Types: table

### **x** — Predictor data

numeric matrix

Predictor data, specified as a numeric matrix.

Each row of X corresponds to one observation (also known as an instance or example), and each column corresponds to one variable (also known as a feature). The variables making up the columns of X must be the same as the variables that trained the SVMMoDel classifier.

The length of Y and the number of rows of X must be equal.

If you set 'Standardize', true in `fitcsvm` to train SVMMoDel, then the software standardizes the columns of X using the corresponding means in SVMMoDel.Mu and standard deviations in SVMMoDel.Sigma.

Data Types: `double` | `single`

**ResponseVarName** — Response variable name

name of a variable in `TBL`

Response variable name, specified as the name of a variable in `TBL`. If `TBL` contains the response variable used to train `SVMModel`, then you do not need to specify `ResponseVarName`.

If you specify `ResponseVarName`, then you must do so as a character vector. For example, if the response variable is stored as `TBL.Response`, then specify it as `'Response'`. Otherwise, the software treats all columns of `TBL`, including `TBL.Response`, as predictors.

The response variable must be a categorical or character array, logical or numeric vector, or cell array of character vectors. If the response variable is a character array, then each element must correspond to one row of the array.

**Y** — Class labels

categorical array | character array | logical vector | vector of numeric values | cell array of character vectors

Class labels, specified as a categorical or character array, logical or numeric vector, or cell array of character vectors. `Y` must be the same as the data type of `SVMModel.ClassNames`.

The length of `Y` must equal the number of rows of `TBL` or `X` must be equal.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

**Weights** — Observation weights

`ones(size(X,1))` (default) | name of a variable in `TBL` | numeric vector

Observation weights, specified as the comma-separated pair consisting of `'Weights'` and a numeric vector or the name of a variable in `TBL`.

If you specify `Weights` as a numeric vector, then the size of `Weights` must be equal to the number of rows in `X` or `TBL`.

If you specify `Weights` as the name of a variable in `TBL`, you must do so as a character vector. For example, if the weights are stored as `TBL.W`, then specify it as `'W'`. Otherwise, the software treats all columns of `TBL`, including `TBL.W`, as predictors.

If you supply weights, `edge` computes the weighted classification edge on page 32-1568. The software weights the observations in each row of `X` or `TBL` with the corresponding weight in `Weights`.

Data Types: `single` | `double`

## Output Arguments

### **e** — Classification edge

scalar

Classification edge, returned as a scalar. `e` represents the (weighted) mean of the classification margins on page 32-1569.

## Examples

### Estimate the Test Sample Edge of SVM Classifiers

Load the ionosphere data set.

```
load ionosphere
rng(1); % For reproducibility
```

Train an SVM classifier. Specify a 15% holdout sample for testing. It is good practice to specify the class order and standardize the data.

```
CVSVMMModel = fitsvm(X,Y,'Holdout',0.15,'ClassNames',{'b','g'},...
    'Standardize',true);
CompactSVMMModel = CVSVMMModel.Trained{1}; % Extract trained, compact classifier
testInds = test(CVSVMMModel.Partition); % Extract the test indices
XTest = X(testInds,:);
YTest = Y(testInds,:);
```

CVSVMModel is a ClassificationPartitionedModel classifier. It contains the property Trained, which is a 1-by-1 cell array holding a CompactClassificationSVM classifier that the software trained using the training set.

Estimate the test sample edge.

```
e = edge(CompactSVMModel,XTest,YTest)
```

```
e =
```

```
5.0765
```

The estimated test sample margin average is approximately 5.

### Estimate the Test Sample Weighted Margin Mean of SVM Classifiers

Load the ionosphere data set.

```
load ionosphere
rng(1); % For reproducibility
```

Suppose that the observations were measured sequentially, and that the last 150 observations were better quality due to a technology upgrade. One way to incorporate this advancement is to weigh the better quality observations more than the other observations.

Define a weight vector that weighs the better quality observations two times the other observations.

```
n = size(X,1);
weights = [ones(n-150,1);2*ones(150,1)];
```

Train an SVM classifier. Specify the weighting scheme and a 15% holdout sample for testing. It is good practice to specify the class order and standardize the data.

```
CVSVMModel = fitcsvm(X,Y,'Weights',weights,'Holdout',0.15,...
    'ClassNames',{'b','g'],'Standardize',true);
CompactSVMModel = CVSVMModel.Trained{1};
testInds = test(CVSVMModel.Partition); % Extract the test indices
XTest = X(testInds,:);
```

```
YTest = Y(testInds, :);
wTest = weights(testInds, :);
```

CVSVMModel is a trained ClassificationPartitionedModel classifier. It contains the property Trained, which is a 1-by-1 cell array holding a CompactClassificationSVM classifier that the software trained using the training set.

Estimate the test sample weighted edge using the weighting scheme.

```
e = edge(CompactSVMModel, XTest, YTest, 'Weights', wTest)
```

```
e =
```

```
4.8341
```

The test sample weighted average margin is approximately 5.

### Select SVM Classifier Features by Comparing Test Sample Edges

The classifier edge measures the average of the classifier margins. One way to perform feature selection is to compare test sample edges from multiple models. Based solely on this criterion, the classifier with the highest edge is the best classifier.

Load the ionosphere data set.

```
load ionosphere
rng(1); % For reproducibility
```

Partition the data set into training and test sets. Specify a 15% holdout sample for testing.

```
Partition = cvpartition(Y, 'Holdout', 0.15);
testInds = test(Partition); % Indices for the test set
XTest = X(testInds, :);
YTest = Y(testInds, :);
```

Partition defines the data set partition.

Define these two data sets:

- `fullX` contains all predictors (except the removed column of 0s).
- `partX` contains the last 20 predictors.

```
fullX = X;  
partX = X(:,end-20:end);
```

Train SVM classifiers for each predictor set. Specify the partition definition.

```
FullCVSVMModel = fitcsvm(fullX,Y,'CVPartition',Partition);  
PartCVSVMModel = fitcsvm(partX,Y,'CVPartition',Partition);  
FCSVMModel = FullCVSVMModel.Trained{1};  
PCSVMModel = PartCVSVMModel.Trained{1};
```

`FullCVSVMModel` and `PartCVSVMModel` are `ClassificationPartitionedModel` classifiers. They contain the property `Trained`, which is a 1-by-1 cell array holding a `CompactClassificationSVM` classifier that the software trained using the training set.

Estimate the test sample edge for each classifier.

```
fullEdge = edge(FCSVMModel,XTest,YTest)  
partEdge = edge(PCSVMModel,XTest(:,end-20:end),YTest)
```

```
fullEdge =
```

```
    2.8319
```

```
partEdge =
```

```
    1.5541
```

The edge for the classifier trained on the complete data set is greater, suggesting that the classifier trained using all of the predictors is better.

## Definitions

### Classification Edge

The edge is the weighted mean of the classification margins.

The weights are the prior class probabilities. If you supply weights, then the software normalizes them to sum to the prior probabilities in the respective classes. The software uses the renormalized weights to compute the weighted mean.

One way to choose among multiple classifiers, e.g., to perform feature selection, is to choose the classifier that yields the highest edge.

## Classification Margin

The classification margins for binary classification are, for each observation, the difference between the classification score for the true class and the classification score for the false class.

The software defines the classification margin for binary classification as

$$m = 2yf(x).$$

$x$  is an observation. If the true label of  $x$  is the positive class, then  $y$  is 1, and  $-1$  otherwise.  $f(x)$  is the positive-class classification score for the observation  $x$ . The literature commonly defines the margin as  $m = yf(x)$ .

If the margins are on the same scale, then they serve as a classification confidence measure, i.e., among multiple classifiers, those that yield larger margins are better.

## Classification Score

The SVM classification score for classifying observation  $x$  is the signed distance from  $x$  to the decision boundary ranging from  $-\infty$  to  $+\infty$ . A positive score for a class indicates that  $x$  is predicted to be in that class. A negative score indicates otherwise.

The score for predicting  $x$  into the positive class, also the numerical, predicted response for  $x$ ,  $f(x)$ , is the trained SVM classification function

$$f(x) = \sum_{j=1}^n \alpha_j y_j G(x_j, x) + b,$$

where  $(\alpha_1, \dots, \alpha_n, b)$  are the estimated SVM parameters,  $G(x_j, x)$  is the dot product in the predictor space between  $x$  and the support vectors, and the sum includes the training set observations. The score for predicting  $x$  into the negative class is  $-f(x)$ .

If  $G(x_j, x) = x_j'x$  (the linear kernel), then the score function reduces to

$$f(x) = (x / s)' \beta + b.$$

$s$  is the kernel scale and  $\beta$  is the vector of fitted linear coefficients.

## Algorithms

For binary classification, the software defines the margin for observation  $j$ ,  $m_j$ , as

$$m_j = 2y_j f(x_j),$$

where  $y_j \in \{-1, 1\}$ , and  $f(x_j)$  is the predicted score of observation  $j$  for the positive class.

However, the literature commonly uses  $m_j = y_j f(x_j)$  to define the margin.

## References

- [1] Christianini, N., and J. C. Shawe-Taylor. *An Introduction to Support Vector Machines and Other Kernel-Based Learning Methods*. Cambridge, UK: Cambridge University Press, 2000.

## Extended Capabilities

### Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays” (MATLAB).

### See Also

ClassificationSVM | CompactClassificationSVM | fitcsvm | kfoldedge | loss  
| margin | predict | resubEdge



# edge

**Class:** CompactClassificationTree

Classification edge

## Syntax

```
E = edge(tree, TBL, ResponseVarName)
```

```
E = edge(tree, X, Y)
```

```
E = edge( ____, Name, Value)
```

## Description

`E = edge(tree, TBL, ResponseVarName)` returns the classification edge for `tree` with data `TBL` and classification `TBL.ResponseVarName`.

`E = edge(tree, X, Y)` returns the classification edge for `tree` with data `X` and classification `Y`.

`E = edge( ____, Name, Value)` computes the edge with additional options specified by one or more `Name, Value` pair arguments, using any of the previous syntaxes. For example, you can specify observation weights.

## Input Arguments

**tree** — Trained classification tree

ClassificationTree model object | CompactClassificationTree model object

Trained classification tree, specified as a `ClassificationTree` or `CompactClassificationTree` model object. That is, `tree` is a trained classification model returned by `fitctree` or `compact`.

**TBL** — Sample data

table

Sample data, specified as a table. Each row of `TBL` corresponds to one observation, and each column corresponds to one predictor variable. Optionally, `TBL` can contain additional columns for the response variable and observation weights. `TBL` must contain all the predictors used to train `tree`. Multi-column variables and cell arrays other than cell arrays of character vectors are not allowed.

If `TBL` contains the response variable used to train `tree`, then you do not need to specify `ResponseVarName` or `Y`.

If you train `tree` using sample data contained in a table, then the input data for this method must also be in a table.

Data Types: `table`

### **x — Data to classify**

numeric matrix

Data to classify, specified as a numeric matrix. Each row of `X` represents one observation, and each column represents one predictor. `X` must have the same number of columns as the data used to train `tree`. `X` must have the same number of rows as the number of elements in `Y`.

Data Types: `single` | `double`

### **ResponseVarName — Response variable name**

name of a variable in `TBL`

Response variable name, specified as the name of a variable in `TBL`. If `TBL` contains the response variable used to train `tree`, then you do not need to specify `ResponseVarName`.

If you specify `ResponseVarName`, then you must do so as a character vector. For example, if the response variable is stored as `TBL.Response`, then specify it as `'Response'`. Otherwise, the software treats all columns of `TBL`, including `TBL.ResponseVarName`, as predictors.

The response variable must be a categorical or character array, logical or numeric vector, or cell array of character vectors. If the response variable is a character array, then each element must correspond to one row of the array.

### **y — Class labels**

categorical array | character array | logical vector | vector of numeric values | cell array of character vectors

Class labels, specified as a categorical or character array, a logical or numeric vector, or a cell array of character vectors. `Y` must be of the same type as the classification used to train `tree`, and its number of elements must equal the number of rows of `X`.

Data Types: `single` | `double` | `categorical` | `char` | `logical` | `cell`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### **Weights** — Observation weights

`ones(size(X,1))` (default) | name of a variable in TBL | numeric vector

Observation weights, specified as the comma-separated pair consisting of `'Weights'` and a numeric vector or the name of a variable in TBL.

If you specify `Weights` as a numeric vector, then the size of `Weights` must be equal to the number of rows in `X` or TBL.

If you specify `Weights` as the name of a variable in TBL, you must do so as a character vector. For example, if the weights are stored as `TBL.W`, then specify it as `'W'`. Otherwise, the software treats all columns of TBL, including `TBL.W`, as predictors.

If you supply weights, `edge` computes the weighted classification `edge` on page 32-1568. The software weights the observations in each row of `X` or TBL with the corresponding weight in `Weights`.

## Output Arguments

### **E** — Classification edge

scalar value

Classification edge, returned as a scalar representing the weighted average value of the margin.

## Examples

Compute the classification margin and edge for the Fisher iris data, trained on its first two columns of data, and view the last 10 entries:

```
load fisheriris
X = meas(:,1:2);
tree = fitctree(X,species);
E = edge(tree,X,species)

E =
    0.6299

M = margin(tree,X,species);
M(end-10:end)

ans =
    0.1111
    0.1111
    0.1111
   -0.2857
    0.6364
    0.6364
    0.1111
    0.7500
    1.0000
    0.6364
    0.2000
```

The classification tree trained on all the data is better.

```
tree = fitctree(meas,species);
E = edge(tree,meas,species)

E =
    0.9384

M = margin(tree,meas,species);
M(end-10:end)

ans =
    0.9565
    0.9565
    0.9565
```

```
0.9565
0.9565
0.9565
0.9565
0.9565
0.9565
0.9565
0.9565
```

## Definitions

### Margin

The classification margin is the difference between the classification score for the true class and maximal classification score for the false classes. Margin is a column vector with the same number of rows as the matrix  $X$ .

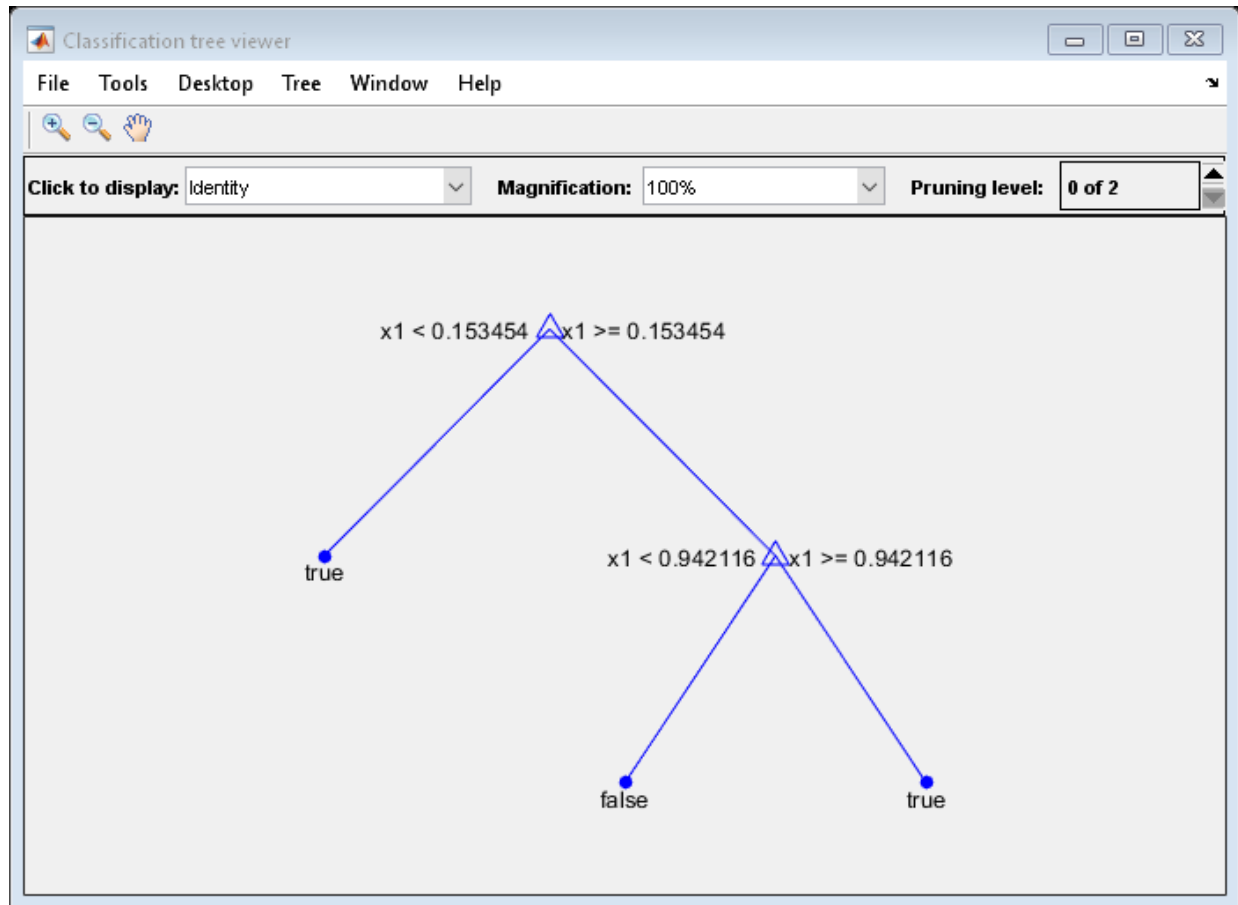
### Score (tree)

For trees, the score of a classification of a leaf node is the posterior probability of the classification at that node. The posterior probability of the classification at a node is the number of training sequences that lead to that node with the classification, divided by the number of training sequences that lead to that node.

For example, consider classifying a predictor  $X$  as true when  $X < 0.15$  or  $X > 0.95$ , and  $X$  is false otherwise.

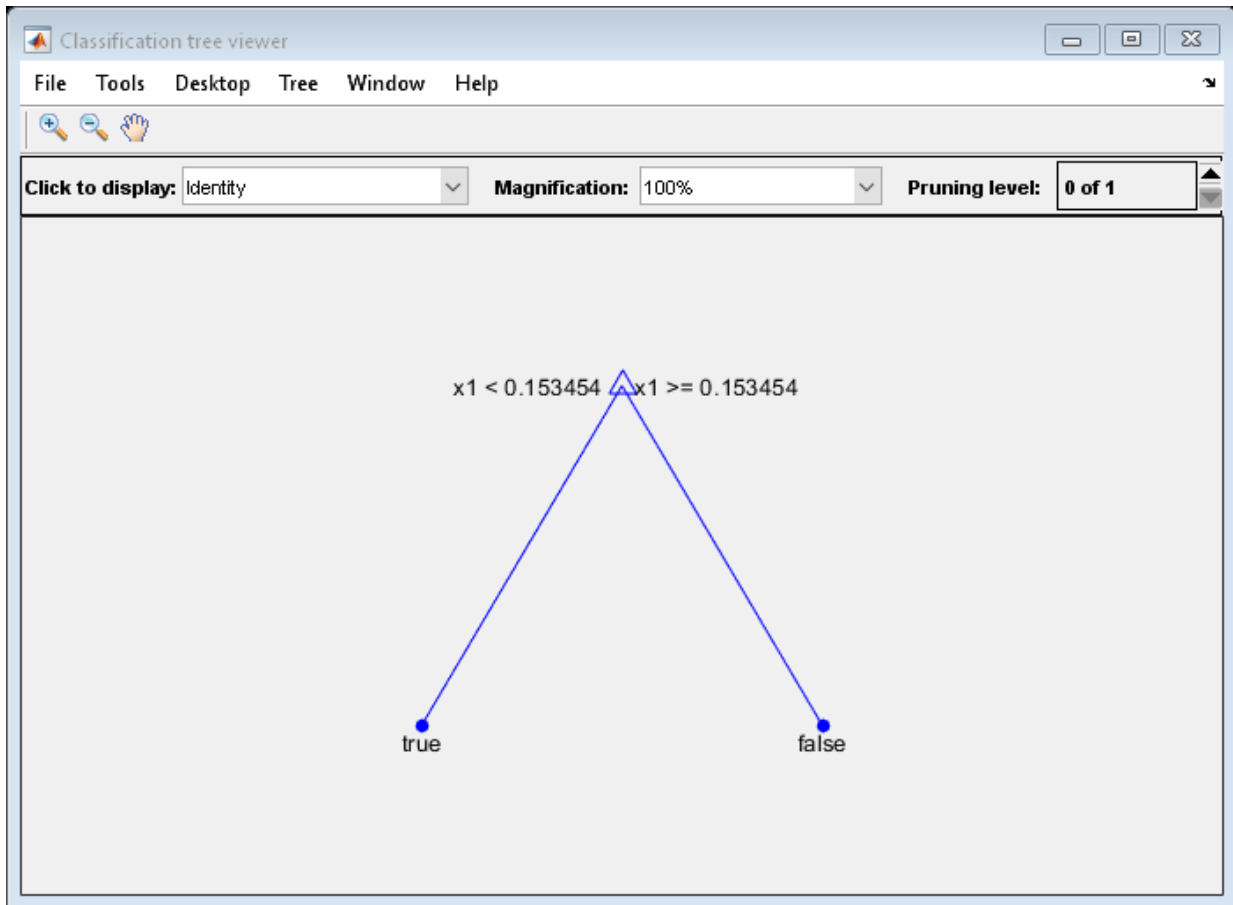
Generate 100 random points and classify them:

```
rng(0, 'twister') % for reproducibility
X = rand(100,1);
Y = (abs(X - .55) > .4);
tree = fitctree(X,Y);
view(tree, 'Mode', 'Graph')
```



Prune the tree:

```
tree1 = prune(tree, 'Level', 1);  
view(tree1, 'Mode', 'Graph')
```



The pruned tree correctly classifies observations that are less than 0.15 as `true`. It also correctly classifies observations from .15 to .94 as `false`. However, it incorrectly classifies observations that are greater than .94 as `false`. Therefore, the score for observations that are greater than .15 should be about  $.05/.85=.06$  for `true`, and about  $.8/.85=.94$  for `false`.

Compute the prediction scores for the first 10 rows of `X`:

```
[~,score] = predict(tree1,X(1:10));
[score X(1:10,:)]
```

```
ans =  
  
    0.9059    0.0941    0.8147  
    0.9059    0.0941    0.9058  
         0    1.0000    0.1270  
    0.9059    0.0941    0.9134  
    0.9059    0.0941    0.6324  
         0    1.0000    0.0975  
    0.9059    0.0941    0.2785  
    0.9059    0.0941    0.5469  
    0.9059    0.0941    0.9575  
    0.9059    0.0941    0.9649
```

Indeed, every value of `X` (the right-most column) that is less than 0.15 has associated scores (the left and center columns) of 0 and 1, while the other values of `X` have associated scores of 0.91 and 0.09. The difference (score 0.09 instead of the expected .06) is due to a statistical fluctuation: there are 8 observations in `X` in the range `(.95, 1)` instead of the expected 5 observations.

## Edge

The edge is the weighted mean value of the classification margin. The weights are the class probabilities in `tree.Prior`. If you supply weights in the `weights` name-value pair, those weights are normalized to sum to the prior probabilities in the respective classes, and are then used to compute the weighted average.

## Extended Capabilities

### Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays” (MATLAB).



## See Also

`fitctree` | `loss` | `margin` | `predict`

## end

**Class:** dataset

Last index in indexing expression for dataset array

---

**Note** The `dataset` data type might be removed in a future release. To work with heterogeneous data, use the MATLAB® `table` data type instead. See MATLAB `table` documentation for more information.

---

## Syntax

`end(A, k, n)`

## Description

`end(A, k, n)` is called for indexing expressions involving the dataset `A` when `end` is part of the `k`-th index out of `n` indices. For example, the expression `A(end-1, :)` calls `A`'s `end` method with `end(A, 1, 2)`.

## See Also

`size`

# end

**Class:** grandset

Last index in indexing expression for point set

## Syntax

`end(p, k, n)`

## Description

`end(p, k, n)` is called for indexing expressions involving the point set `p` when `end` is part of the `k`-th index out of `n` indices. For example, the expression `p(end-1, :)` calls `p`'s `end` method with `end(p, 1, 2)`.

## See Also

`grandset`

## epsilon

**Class:** RepeatedMeasuresModel

Epsilon adjustment for repeated measures anova

## Syntax

```
tbl = epsilon(rm)
tbl = epsilon(rm,C)
```

## Description

`tbl = epsilon(rm)` returns the epsilon adjustment factors for repeated measures model `rm`.

`tbl = epsilon(rm,C)` returns the epsilon adjustment factors for the test based on the contrast matrix `C`.

## Input Arguments

**rm — Repeated measures model**

RepeatedMeasuresModel object

Repeated measures model, returned as a RepeatedMeasuresModel object.

For properties and methods of this object, see RepeatedMeasuresModel.

**c — Contrasts**

matrix

Contrasts, specified as a matrix. The default value of `C` is the `Q` factor in a QR decomposition of the matrix `M`, where `M` is defined so that `Y*M` is the difference between all successive pairs of columns of the repeated measures matrix `Y`.

Data Types: `single` | `double`

## Output Arguments

### `tbl` — Epsilon adjustment factors

table

Epsilon adjustment factors for the repeated measures model `rm`, returned as a table. `tbl` contains four different adjustments for `epsilon`.

Correction	Definition
Uncorrected	No adjustments, <code>epsilon = 1</code>
Greenhouse-Geiser	Greenhouse-Geiser approximation
Huynh-Feldt	Huynh-Feldt approximation
Lower bound	Lower bound on the <i>p</i> -value

For details, see “Compound Symmetry Assumption and Epsilon Corrections” on page 9-81.

Data Types: table

## Examples

### Epsilon Corrections for Repeated Measures ANOVA

Load the sample data.

```
load fisheriris
```

The column vector, `species` consists of iris flowers of three different species: `setosa`, `versicolor`, `virginica`. The double matrix `meas` consists of four types of measurements on the flowers: the length and width of sepals and petals in centimeters, respectively.

Store the data in a table array.

```
t = table(species,meas(:,1),meas(:,2),meas(:,3),meas(:,4),...
'VariableNames',{'species','meas1','meas2','meas3','meas4'});
Meas = dataset([1 2 3 4'],'VarNames',{'Measurements'});
```

Fit a repeated measures model, where the measurements are the responses and the `species` is the predictor variable.

```
rm = fitrm(t, 'meas1-meas4~species', 'WithinDesign', Meas);
```

Perform repeated measures analysis of variance.

```
ranovatbl = ranova(rm)
```

```
ranovatbl =
```

```
3x8 table
```

	SumSq	DF	MeanSq	F	pValue
(Intercept):Measurements	1656.3	3	552.09	6873.3	0
species:Measurements	282.47	6	47.078	586.1	1.4271e-206
Error(Measurements)	35.423	441	0.080324		

`ranova` computes the last three *P*-values using Greenhouse-Geisser, Huynh-Feldt, and lower bound corrections, respectively.

Display the epsilon correction values.

```
epsilon(rm)
```

```
ans =
```

```
1x4 table
```

Uncorrected	GreenhouseGeisser	HuynhFeldt	LowerBound
1	0.75179	0.76409	0.33333

You can check the compound symmetry (sphericity) assumption using the `mauchly` method.

## Tips

- The `mauchly` method tests for sphericity.

- The `ranova` method contains  $p$ -values based on each epsilon value.

## Algorithms

`ranova` computes the regular  $p$ -value (in the `pValue` column of the `rmanova` table) using the  $F$ -statistic cumulative distribution function:

$$p\text{-value} = 1 - \text{fcdf}(F, v_1, v_2).$$

When the compound symmetry assumption is not satisfied, `ranova` uses a correction factor epsilon,  $\varepsilon$ , to compute the corrected  $p$ -values as follows:

$$p\text{-value\_corrected} = 1 - \text{fcdf}(F, \varepsilon^* v_1, \varepsilon^* v_2).$$

The `epsilon` method returns the epsilon adjustment values.

## See Also

`fitrm` | `mauchly` | `ranova`

## Topics

“Compound Symmetry Assumption and Epsilon Corrections” on page 9-81

“Mauchly’s Test of Sphericity” on page 9-83

## evcdf

Extreme value cumulative distribution function

### Syntax

```
p = evcdf(x,mu,sigma)
[p,plo,pup] = evcdf(x,mu,sigma,pcov,alpha)
[p,plo,pup] = evcdf( ___, 'upper')
```

### Description

`p = evcdf(x,mu,sigma)` returns the cumulative distribution function (cdf) for the type 1 extreme value distribution, with location parameter `mu` and scale parameter `sigma`, at each of the values in `x`. `x`, `mu`, and `sigma` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array of the same size as the other inputs. The default values for `mu` and `sigma` are 0 and 1, respectively.

`[p,plo,pup] = evcdf(x,mu,sigma,pcov,alpha)` returns confidence bounds for `p` when the input parameters `mu` and `sigma` are estimates. `pcov` is a 2-by-2 covariance matrix of the estimated parameters. `alpha` has a default value of 0.05, and specifies 100(1 - `alpha`)% confidence bounds. `plo` and `pup` are arrays of the same size as `p`, containing the lower and upper confidence bounds.

`[p,plo,pup] = evcdf( ___, 'upper')` returns the complement of the type 1 extreme value distribution cdf at each value in `x`, using an algorithm that more accurately computes the extreme upper tail probabilities. You can use the 'upper' argument with any of the previous syntaxes.

The function `evcdf` computes confidence bounds for `P` using a normal approximation to the distribution of the estimate

$$\frac{X - \hat{\mu}}{\hat{\sigma}}$$



and then transforming those bounds to the scale of the output  $\mathcal{P}$ . The computed bounds give approximately the desired confidence level when you estimate  $\mu$ ,  $\sigma$ , and  $\rho_{cov}$  from large samples, but in smaller samples other methods of computing the confidence bounds might be more accurate.

The type 1 extreme value distribution is also known as the Gumbel distribution. The version used here is suitable for modeling minima; the mirror image of this distribution can be used to model maxima by negating  $x$  and subtracting the resulting distribution values from 1. See “Extreme Value Distribution” on page B-38 for more details. If  $x$  has a Weibull distribution, then  $X = \log(x)$  has the type 1 extreme value distribution.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`cdf` | `evfit` | `evinv` | `evlike` | `evpdf` | `evrnd` | `evstat`

### Topics

“Extreme Value Distribution” on page B-38

Introduced before R2006a

## evfit

Extreme value parameter estimates

### Syntax

```
parmhat = evfit(data)
[parmhat,parmci] = evfit(data)
[parmhat,parmci] = evfit(data,alpha)
[...] = evfit(data,alpha,censoring)
[...] = evfit(data,alpha,censoring,freq)
[...] = evfit(data,alpha,censoring,freq,options)
```

### Description

`parmhat = evfit(data)` returns maximum likelihood estimates of the parameters of the type 1 extreme value distribution given the data in the vector `data`. `parmhat(1)` is the location parameter,  $\mu$ , and `parmhat(2)` is the scale parameter,  $\sigma$ .

`[parmhat,parmci] = evfit(data)` returns 95% confidence intervals for the parameter estimates on the  $\mu$  and  $\sigma$  parameters in the 2-by-2 matrix `parmci`. The first column of the matrix of the extreme value fit contains the lower and upper confidence bounds for the parameter  $\mu$ , and the second column contains the confidence bounds for the parameter  $\sigma$ .

`[parmhat,parmci] = evfit(data,alpha)` returns  $100(1 - \text{alpha})\%$  confidence intervals for the parameter estimates, where `alpha` is a value in the range `[0 1]` specifying the width of the confidence intervals. By default, `alpha` is `0.05`, which corresponds to 95% confidence intervals.

`[...] = evfit(data,alpha,censoring)` accepts a Boolean vector, `censoring`, of the same size as `data`, which is 1 for observations that are right-censored and 0 for observations that are observed exactly.

`[...] = evfit(data,alpha,censoring,freq)` accepts a frequency vector, `freq` of the same size as `data`. Typically, `freq` contains integer frequencies for the corresponding

elements in `data`, but can contain any nonnegative values. Pass in `[]` for `alpha`, `censoring`, or `freq` to use their default values.

`[...] = evfit(data, alpha, censoring, freq, options)` accepts a structure, `options`, that specifies control parameters for the iterative algorithm the function uses to compute maximum likelihood estimates. You can create `options` using the function `statset`. Enter `statset('evfit')` to see the names and default values of the parameters that `evfit` accepts in the `options` structure. See the reference page for `statset` for more information about these options.

The type 1 extreme value distribution is also known as the Gumbel distribution. The version used here is suitable for modeling minima; the mirror image of this distribution can be used to model maxima by negating  $x$ . See “Extreme Value Distribution” on page B-38 for more details. If  $x$  has a Weibull distribution, then  $X = \log(x)$  has the type 1 extreme value distribution.

## See Also

`evcdf` | `evinv` | `evlike` | `evpdf` | `evrnd` | `evstat` | `mle`

## Topics

“Extreme Value Distribution” on page B-38

Introduced before R2006a

## evinv

Extreme value inverse cumulative distribution function

### Syntax

```
X = evinv(P,mu,sigma)
[X,XLO,XUP] = evinv(P,mu,sigma,pcov,alpha)
```

### Description

`X = evinv(P,mu,sigma)` returns the inverse cumulative distribution function (cdf) for a type 1 extreme value distribution with location parameter `mu` and scale parameter `sigma`, evaluated at the values in `P`. `P`, `mu`, and `sigma` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array of the same size as the other inputs. The default values for `mu` and `sigma` are 0 and 1, respectively.

`[X,XLO,XUP] = evinv(P,mu,sigma,pcov,alpha)` produces confidence bounds for `X` when the input parameters `mu` and `sigma` are estimates. `pcov` is the covariance matrix of the estimated parameters. `alpha` is a scalar that specifies  $100(1 - \text{alpha})\%$  confidence bounds for the estimated parameters, and has a default value of 0.05. `XLO` and `XUP` are arrays of the same size as `X` containing the lower and upper confidence bounds.

The function `evinv` computes confidence bounds for `P` using a normal approximation to the distribution of the estimate

$$\hat{\mu} + \hat{\sigma}q$$

where  $q$  is the  $P$ th quantile from an extreme value distribution with parameters  $\mu = 0$  and  $\sigma = 1$ . The computed bounds give approximately the desired confidence level when you estimate `mu`, `sigma`, and `pcov` from large samples, but in smaller samples other methods of computing the confidence bounds might be more accurate.

The type 1 extreme value distribution is also known as the Gumbel distribution. The version used here is suitable for modeling minima; the mirror image of this distribution

can be used to model maxima by negating  $x$ . See “Extreme Value Distribution” on page B-38 for more details. If  $x$  has a Weibull distribution, then  $X = \log(x)$  has the type 1 extreme value distribution.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`evcdf` | `evfit` | `evlike` | `evpdf` | `evrnd` | `evstat` | `icdf`

Introduced before R2006a

## eq

**Class:** grandstream

Test handle equality

## Syntax

```
h1 == h2
tf = eq(h1, h2)
```

## Description

`h1 == h2` performs element-wise comparisons between handle arrays `h1` and `h2`. `h1` and `h2` must be of the same dimensions unless one is a scalar. The result is a logical array of the same dimensions, where each element is an element-wise equality result. If one of `h1` or `h2` is scalar, scalar expansion is performed and the result will match the dimensions of the array that is not scalar.

`tf = eq(h1, h2)` stores the result in a logical array of the same dimensions.

## See Also

`ge` | `gt` | `le` | `lt` | `ne` | `grandstream`

## error

**Class:** CompactTreeBagger

Error (misclassification probability or MSE)

## Syntax

```
err = error(B, TBLnew, Ynew)
err = error(B, Xnew, Ynew)
err = error(B, TBLnew, Ynew, 'param1', val1, 'param2', val2, ...)
err = error(B, Xnew, Ynew, 'param1', val1, 'param2', val2, ...)
```

## Description

`err = error(B, TBLnew, Ynew)` computes the misclassification probability for classification trees or mean squared error (MSE) for regression trees for each tree, for the predictors contained in the table `TBLnew` given true response `Ynew`. You can omit `Ynew` if `TBLnew` contains the response variable. If you trained `B` using sample data contained in a table, then the input data for this method must also be in a table.

`err = error(B, Xnew, Ynew)` computes the misclassification probability for classification trees or mean squared error (MSE) for regression trees for each tree, the for predictors contained in the matrix `Xnew` given true response `Ynew`. If you trained `B` using sample data contained in a matrix, then the input data for this method must also be in a matrix.

For classification, `Ynew` can be either a numeric vector, character matrix, cell array of character vectors, categorical vector or logical vector. For regression, `Y` must be a numeric vector. `err` is a vector with one error measure for each of the `NTrees` trees in the ensemble `B`.

`err = error(B, TBLnew, Ynew, 'param1', val1, 'param2', val2, ...)` or `err = error(B, Xnew, Ynew, 'param1', val1, 'param2', val2, ...)` specifies optional parameter name-value pairs:

'Mode'	Character vector indicating how the method computes errors. If set to 'cumulative' (default), <code>error</code> computes cumulative errors and <code>err</code> is a vector of length <code>NTrees</code> , where the first element gives error from <code>trees(1)</code> , second element gives error from <code>trees(1:2)</code> etc., up to <code>trees(1:NTrees)</code> . If set to 'individual', <code>err</code> is a vector of length <code>NTrees</code> , where each element is an error from each tree in the ensemble. If set to 'ensemble', <code>err</code> is a scalar showing the cumulative error for the entire ensemble.
'Weights'	Vector of observation weights to use for error averaging. By default the weight of every observation is 1. The length of this vector must be equal to the number of rows in <code>X</code> .
'Trees'	Vector of indices indicating what trees to include in this calculation. By default, this argument is set to 'all' and the method uses all trees. If 'Trees' is a numeric vector, the method returns a vector of length <code>NTrees</code> for 'cumulative' and 'individual' modes, where <code>NTrees</code> is the number of elements in the input vector, and a scalar for 'ensemble' mode. For example, in the 'cumulative' mode, the first element gives error from <code>trees(1)</code> , the second element gives error from <code>trees(1:2)</code> etc.
'TreeWeights'	Vector of tree weights. This vector must have the same length as the 'Trees' vector. The method uses these weights to combine output from the specified trees by taking a weighted average instead of the simple non-weighted majority vote. You cannot use this argument in the 'individual' mode.
'UseInstanceForTree'	Logical matrix of size <code>Nobs-by-NTrees</code> indicating which trees should be used to make predictions for each observation. By default the method uses all trees for all observations.

## Algorithms

When estimating the ensemble error:

- Using the 'Mode' name-value pair argument, you can specify to return the error any of these three ways:
  - The error for individual trees in the ensemble



- The cumulative error over all trees
- The error for the entire ensemble
- Using the 'Trees' name-value pair argument, you can specify which trees to use in the ensemble error calculations.
- Using the 'UseInstanceForTree' name-value pair argument, you can specify which observations in the input data (X and Y) to use in the ensemble error calculation for each selected tree.
- Using the 'Weights' name-value pair argument, you can attribute each *observation* with a weight. For the formulae that follow,  $w_j$  is the weight of observation  $j$ .
- Using the 'TreeWeights' name-value pair argument, you can attribute each *tree* with a weight.

For regression problems, `error` estimates the weighted MSE of the ensemble of bagged regression trees for predicting Y given X using selected trees and observations.

- 1 `error` predicts responses for selected observations in X using the selected regression trees in the ensemble.
- 2 The MSE estimate depends on the value of 'Mode'.
  - If you specify 'Mode', 'Individual', then the weighted MSE for tree  $t$  is

$$\text{MSE}_t = \frac{1}{\sum_{j=1}^n w_j} \sum_{j=1}^n w_j (y_j - \hat{y}_{tj})^2.$$

$\hat{y}_{tj}$  is the predicted response of observation  $j$  from selected regression tree  $t$ . `error` sets any unselected observations within a selected tree to the weighted sample average of the observed, training data responses.

- If you specify 'Mode', 'Cumulative', then the weighted MSE is a vector of size  $T^*$  containing cumulative, weighted MSEs over the  $T^* \leq T$  selected trees. `error` follows these steps to estimate  $\text{MSE}_t^*$ , the cumulative, weighted MSE using the first  $t$  selected trees.

**a**

For selected observation  $j$ ,  $j = 1, \dots, n$ , `error` estimates  $\hat{y}_{\text{bag},tj}$ , the weighted average of the predictions among the first  $t$  selected trees (for details, see `predict`). For this computation, `error` uses the tree weights.

- b** `error` estimates the cumulative, weighted MSE through tree  $t$ .

$$\text{MSE}_t^* = \frac{1}{\sum_{j=1}^n w_j} \sum_{j=1}^n w_j (y_j - \hat{y}_{\text{bag},tj})^2.$$

`error` sets observations that are unselected for all selected trees to the weighted sample average of the observed, training data responses.

- If you specify 'Mode', 'Ensemble', then the weighted MSE is the last element of the cumulative, weighted MSE vector.

For classification problems, `error` estimates the weighted misclassification rate of the ensemble of bagged classification trees for predicting  $Y$  given  $X$  using selected trees and observations.

- If you specify 'Mode', 'Individual', then the weighted misclassification rate for tree  $t$  is

$$e_t = \frac{1}{\sum_{j=1}^n w_j} \sum_{j=1}^n w_j I(y_j \neq \hat{y}_{tj}).$$

$\hat{y}_{tj}$  is the predicted class for selected observation  $j$  using from selected classification tree  $t$ . `error` sets any unselected observations within a selected tree to the predicted, weighted, most popular class over all training responses. If there are multiple most popular classes, `error` considers the one listed first in the `ClassNames` property of the `TreeBagger` model the most popular.

- If you specify 'Mode', 'Cumulative' then the weighted misclassification rate is a vector of size  $T^*$  containing cumulative, weighted misclassification rates over the  $T^* \leq T$  selected trees. `error` follows these steps to estimate  $e_t^*$ , the cumulative, weighted misclassification rate using the first  $t$  selected trees.

**1**

For selected observation  $j$ ,  $j = 1, \dots, n$ , `error` estimates  $\hat{y}_{\text{bag},tj}$ , the weighted, most popular class among the first  $t$  selected trees (for details, see `predict`). For this computation, `error` uses the tree weights.

- 2** `error` estimates the cumulative, weighted misclassification rate through tree  $t$ .

$$e_t^* = \frac{1}{\sum_{j=1}^n w_j} \sum_{j=1}^n w_j I(y_j \neq \hat{y}_{\text{bag},tj}).$$

`error` sets any observations that are unselected for all selected trees to the predicted, weighted, most popular class over all training responses. If there are multiple most popular classes, `error` considers the one listed first in the `ClassNames` property of the `TreeBagger` model the most popular.

- If you specify `'Mode', 'Ensemble'`, then the weighted misclassification rate is the last element of the cumulative, weighted misclassification rate vector.

## See Also

`CompactTreeBagger` | `predict`

## Topics

“Bootstrap Aggregation (Bagging) of Regression Trees” on page 18-141

“Bootstrap Aggregation (Bagging) of Classification Trees” on page 18-154

## error

**Class:** TreeBagger

Error (misclassification probability or MSE)

## Syntax

```
err = error(B, TBLnew, Ynew)
err = error(B, Xnew, Ynew)
err = error(B, TBLnew, Ynew, 'param1', val1, 'param2', val2, ...)
err = error(B, Xnew, Ynew, 'param1', val1, 'param2', val2, ...)
```

## Description

`err = error(B, TBLnew, Ynew)` computes the misclassification probability for classification trees or mean squared error (MSE) for regression trees for each tree, for the predictors contained in the table `TBLnew` given true response `Ynew`. You can omit `Ynew` if `TBLnew` contains the response variable. If you trained `B` using sample data contained in a table, then the input data for this method must also be in a table.

`err = error(B, Xnew, Ynew)` computes the misclassification probability for classification trees or mean squared error (MSE) for regression trees for each tree, the for predictors contained in the matrix `Xnew` given true response `Ynew`. If you trained `B` using sample data contained in a matrix, then the input data for this method must also be in a matrix.

For classification, `Ynew` can be either a numeric vector, character matrix, cell array of character vectors, categorical vector or logical vector. For regression, `Y` must be a numeric vector. `err` is a vector with one error measure for each of the `NTrees` trees in the ensemble `B`.

`err = error(B, TBLnew, Ynew, 'param1', val1, 'param2', val2, ...)` or `err = error(B, Xnew, Ynew, 'param1', val1, 'param2', val2, ...)` specifies optional parameter name-value pairs:

'Mode'	Character vector indicating how the method computes errors. If set to 'cumulative' (default), <code>error</code> computes cumulative errors and <code>err</code> is a vector of length <code>NTrees</code> , where the first element gives error from <code>trees(1)</code> , second element gives error from <code>trees(1:2)</code> etc., up to <code>trees(1:NTrees)</code> . If set to 'individual', <code>err</code> is a vector of length <code>NTrees</code> , where each element is an error from each tree in the ensemble. If set to 'ensemble', <code>err</code> is a scalar showing the cumulative error for the entire ensemble.
'Weights'	Vector of observation weights to use for error averaging. By default the weight of every observation is 1. The length of this vector must be equal to the number of rows in <code>X</code> .
'Trees'	Vector of indices indicating what trees to include in this calculation. By default, this argument is set to 'all' and the method uses all trees. If 'Trees' is a numeric vector, the method returns a vector of length <code>NTrees</code> for 'cumulative' and 'individual' modes, where <code>NTrees</code> is the number of elements in the input vector, and a scalar for 'ensemble' mode. For example, in the 'cumulative' mode, the first element gives error from <code>trees(1)</code> , the second element gives error from <code>trees(1:2)</code> etc.
'TreeWeights'	Vector of tree weights. This vector must have the same length as the 'Trees' vector. The method uses these weights to combine output from the specified trees by taking a weighted average instead of the simple non-weighted majority vote. You cannot use this argument in the 'individual' mode.
'UseInstanceForTree'	Logical matrix of size <code>Nobs-by-NTrees</code> indicating which trees should be used to make predictions for each observation. By default the method uses all trees for all observations.

## Algorithms

When estimating the ensemble error:

- Using the 'Mode' name-value pair argument, you can specify to return the error any of these three ways:
  - The error for individual trees in the ensemble

- The cumulative error over all trees
- The error for the entire ensemble
- Using the 'Trees' name-value pair argument, you can specify which trees to use in the ensemble error calculations.
- Using the 'UseInstanceForTree' name-value pair argument, you can specify which observations in the input data (X and Y) to use in the ensemble error calculation for each selected tree.
- Using the 'Weights' name-value pair argument, you can attribute each *observation* with a weight. For the formulae that follow,  $w_j$  is the weight of observation  $j$ .
- Using the 'TreeWeights' name-value pair argument, you can attribute each *tree* with a weight.

For regression problems, `error` estimates the weighted MSE of the ensemble of bagged regression trees for predicting Y given X using selected trees and observations.

- 1 `error` predicts responses for selected observations in X using the selected regression trees in the ensemble.
- 2 The MSE estimate depends on the value of 'Mode'.
  - If you specify 'Mode', 'Individual', then the weighted MSE for tree  $t$  is

$$\text{MSE}_t = \frac{1}{\sum_{j=1}^n w_j} \sum_{j=1}^n w_j (y_j - \hat{y}_{tj})^2.$$

$\hat{y}_{tj}$  is the predicted response of observation  $j$  from selected regression tree  $t$ . `error` sets any unselected observations within a selected tree to the weighted sample average of the observed, training data responses.

- If you specify 'Mode', 'Cumulative', then the weighted MSE is a vector of size  $T^*$  containing cumulative, weighted MSEs over the  $T^* \leq T$  selected trees. `error` follows these steps to estimate  $\text{MSE}_t^*$ , the cumulative, weighted MSE using the first  $t$  selected trees.

**a**

For selected observation  $j$ ,  $j = 1, \dots, n$ , `error` estimates  $\hat{y}_{\text{bag},tj}$ , the weighted average of the predictions among the first  $t$  selected trees (for details, see `predict`). For this computation, `error` uses the tree weights.

- b** `error` estimates the cumulative, weighted MSE through tree  $t$ .

$$\text{MSE}_t^* = \frac{1}{\sum_{j=1}^n w_j} \sum_{j=1}^n w_j (y_j - \hat{y}_{\text{bag},tj})^2.$$

`error` sets observations that are unselected for all selected trees to the weighted sample average of the observed, training data responses.

- If you specify 'Mode', 'Ensemble', then the weighted MSE is the last element of the cumulative, weighted MSE vector.

For classification problems, `error` estimates the weighted misclassification rate of the ensemble of bagged classification trees for predicting  $Y$  given  $X$  using selected trees and observations.

- If you specify 'Mode', 'Individual', then the weighted misclassification rate for tree  $t$  is

$$e_t = \frac{1}{\sum_{j=1}^n w_j} \sum_{j=1}^n w_j I(y_j \neq \hat{y}_{tj}).$$

$\hat{y}_{tj}$  is the predicted class for selected observation  $j$  using from selected classification tree  $t$ . `error` sets any unselected observations within a selected tree to the predicted, weighted, most popular class over all training responses. If there are multiple most popular classes, `error` considers the one listed first in the `ClassNames` property of the `TreeBagger` model the most popular.

- If you specify 'Mode', 'Cumulative' then the weighted misclassification rate is a vector of size  $T^*$  containing cumulative, weighted misclassification rates over the  $T^* \leq T$  selected trees. `error` follows these steps to estimate  $e_t^*$ , the cumulative, weighted misclassification rate using the first  $t$  selected trees.

**1**

For selected observation  $j$ ,  $j = 1, \dots, n$ , `error` estimates  $\hat{y}_{\text{bag},tj}$ , the weighted, most popular class among the first  $t$  selected trees (for details, see `predict`). For this computation, `error` uses the tree weights.

- 2** `error` estimates the cumulative, weighted misclassification rate through tree  $t$ .

$$e_t^* = \frac{1}{\sum_{j=1}^n w_j} \sum_{j=1}^n w_j I(y_j \neq \hat{y}_{\text{bag},tj}).$$

`error` sets any observations that are unselected for all selected trees to the predicted, weighted, most popular class over all training responses. If there are multiple most popular classes, `error` considers the one listed first in the `ClassNames` property of the `TreeBagger` model the most popular.

- If you specify `'Mode', 'Ensemble'`, then the weighted misclassification rate is the last element of the cumulative, weighted misclassification rate vector.

## See Also

`TreeBagger` | `compact` | `error` | `oobError` | `oobPredict` | `predict` | `quantileError`

## Topics

“Bootstrap Aggregation (Bagging) of Regression Trees” on page 18-141

“Bootstrap Aggregation (Bagging) of Classification Trees” on page 18-154



## eval

**Class:** `classregtree`

Predicted responses

---

**Note** `classregtree` will be removed in a future release. See `fitctree`, `fitrtree`, `ClassificationTree`, or `RegressionTree` instead.

---

## Syntax

```
yfit = eval(t,X)
yfit = eval(t,X,s)
[yfit,nodes] = eval(...)
[yfit,nodes,cnums] = eval(...)
[...] = t(X)
[...] = t(X,s)
```

## Description

`yfit = eval(t,X)` takes a classification or regression tree `t` and a matrix `X` of predictors, and produces a vector `yfit` of predicted response values. For a regression tree, `yfit(i)` is the fitted response value for a point having the predictor values `X(i,:)`. For a classification tree, `yfit(i)` is the class into which the tree assigns the point with data `X(i,:)`.

`yfit = eval(t,X,s)` takes an additional vector `s` of pruning levels, with 0 representing the full, unpruned tree. `t` must include a pruning sequence as created by `classregtree` or by `prune`. If `s` has  $k$  elements and `X` has  $n$  rows, the output `yfit` is an  $n$ -by- $k$  matrix, with the  $j$ th column containing the fitted values produced by the  $s(j)$  subtree. `s` must be sorted in ascending order.

To compute fitted values for a tree that is not part of the optimal pruning sequence, first use `prune` to prune the tree.

`[yfit,nodes] = eval(...)` also returns a vector `nodes` the same size as `yfit` containing the node number assigned to each row of `X`. Use `view` to display the node numbers for any node you select.

`[yfit,nodes,cnums] = eval(...)` is valid only for classification trees. It returns a vector `cnum` containing the predicted class numbers.

NaN values in `X` are treated as missing. If `eval` encounters a missing value when it attempts to evaluate the split rule at a branch node, it cannot determine whether to proceed to the left or right child node. Instead, it sets the corresponding fitted value equal to the fitted value assigned to the branch node.

`[...] = t(X)` or `[...] = t(X,s)` also invoke `eval`.

## Examples

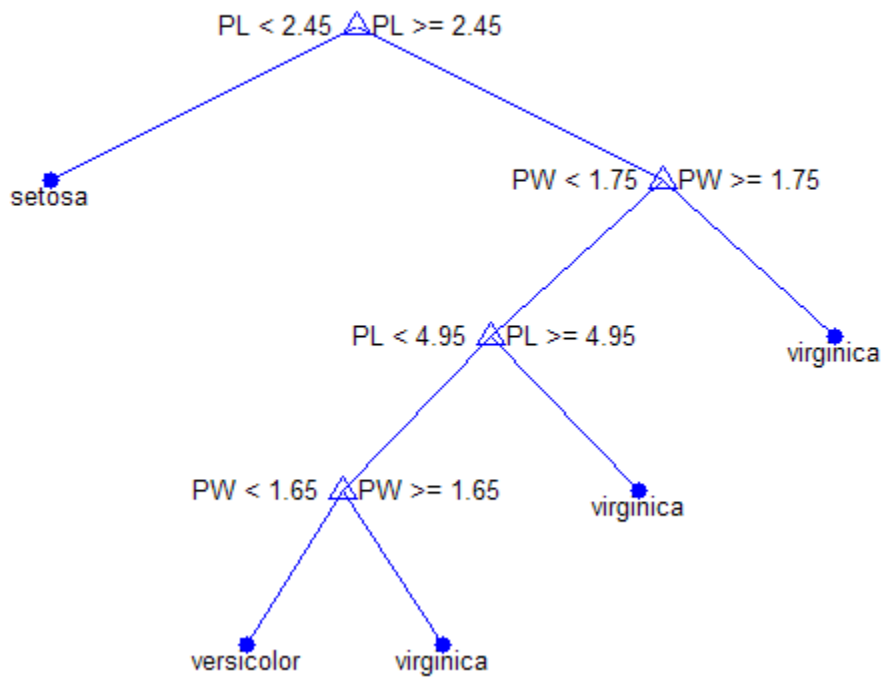
Create a classification tree for Fisher's iris data:

```
load fisheriris;

t = classregtree(meas,species,...
                'names',{'SL' 'SW' 'PL' 'PW'})
t =
Decision tree for classification
1  if PL<2.45 then node 2 elseif PL>=2.45 then node 3 else setosa
2  class = setosa
3  if PW<1.75 then node 4 elseif PW>=1.75 then node 5 else versicolor
4  if PL<4.95 then node 6 elseif PL>=4.95 then node 7 else versicolor
5  class = virginica
6  if PW<1.65 then node 8 elseif PW>=1.65 then node 9 else versicolor
7  class = virginica
8  class = versicolor
9  class = virginica

view(t)
```

Click to display:  Magnification:  Pruning level:



Find assigned class names:

```
sfit = eval(t,meas);
```

Compute that proportion is correctly classified:

```
pct = mean(strcmp(sfit,species))
pct =
    0.9800
```

## References

- [1] Breiman, L., J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Boca Raton, FL: CRC Press, 1984.

## See Also

`classregtree` | `prune` | `test` | `view`

# evalclusters

Evaluate clustering solutions

## Syntax

```
eva = evalclusters(x, clust, criterion)
eva = evalclusters(x, clust, criterion, Name, Value)
```

## Description

`eva = evalclusters(x, clust, criterion)` creates a clustering evaluation object containing data used to evaluate the optimal number of data clusters.

`eva = evalclusters(x, clust, criterion, Name, Value)` creates a clustering evaluation object using additional options specified by one or more name-value pair arguments.

## Examples

### Evaluate the Clustering Solution Using Calinski-Harabasz Criterion

Evaluate the optimal number of clusters using the Calinski-Harabasz clustering evaluation criterion.

Load the sample data.

```
load fisheriris;
```

The data contains length and width measurements from the sepals and petals of three species of iris flowers.

Evaluate the optimal number of clusters using the Calinski-Harabasz criterion. Cluster the data using `kmeans`.

```
rng('default'); % For reproducibility
eva = evalclusters(meas, 'kmeans', 'CalinskiHarabasz', 'KList', [1:6])

eva =

CalinskiHarabaszEvaluation with properties:

    NumObservations: 150
      InspectedK: [1 2 3 4 5 6]
  CriterionValues: [NaN 513.9245 561.6278 530.4871 456.1279 469.5068]
      OptimalK: 3
```

The `OptimalK` value indicates that, based on the Calinski-Harabasz criterion, the optimal number of clusters is three.

### Evaluate a Matrix of Clustering Solutions

Use an input matrix of proposed clustering solutions to evaluate the optimal number of clusters.

Load the sample data.

```
load fisheriris;
```

The data contains length and width measurements from the sepals and petals of three species of iris flowers.

Use `kmeans` to create an input matrix of proposed clustering solutions for the sepal length measurements, using 1, 2, 3, 4, 5, and 6 clusters.

```
clust = zeros(size(meas,1),6);
for i=1:6
    clust(:,i) = kmeans(meas,i,'emptyaction','singleton',...
        'replicate',5);
end
```

Each row of `clust` corresponds to one sepal length measurement. Each of the six columns corresponds to a clustering solution containing 1 to 6 clusters.

Evaluate the optimal number of clusters using the Calinski-Harabasz criterion.

```

eva = evalclusters(meas, clust, 'CalinskiHarabasz')

eva =

    CalinskiHarabaszEvaluation with properties:

        NumObservations: 150
        InspectedK: [1 2 3 4 5 6]
        CriterionValues: [NaN 513.9245 561.6278 530.4871 456.1279 469.5068]
        OptimalK: 3

```

The OptimalK value indicates that, based on the Calinski-Harabasz criterion, the optimal number of clusters is three.

### Specify Clustering Algorithm with a Function Handle

Use a function handle to specify the clustering algorithm, then evaluate the optimal number of clusters.

Load the sample data.

```
load fisheriris;
```

The data contains length and width measurements from the sepals and petals of three species of iris flowers.

Use a function handle to specify the clustering algorithm.

```
myfunc = @(X,K) (kmeans(X, K, 'emptyaction', 'singleton', ...
    'replicate', 5));
```

Evaluate the optimal number of clusters for the sepal length data using the Calinski-Harabasz criterion.

```
eva = evalclusters(meas, myfunc, 'CalinskiHarabasz', ...
    'klist', [1:6])
```

```

eva =

    CalinskiHarabaszEvaluation with properties:

```

```
NumObservations: 150
  InspectedK: [1 2 3 4 5 6]
CriterionValues: [NaN 513.9245 561.6278 530.4871 456.1279 469.5068]
  OptimalK: 3
```

The `OptimalK` value indicates that, based on the Calinski-Harabasz criterion, the optimal number of clusters is three.

- “Clustering Using Gaussian Mixture Models” on page 16-28

## Input Arguments

### **x** — Input data

matrix

Input data, specified as an  $N$ -by- $P$  matrix.  $N$  is the number of observations, and  $P$  is the number of variables.

Data Types: `single` | `double`

### **c**l`ust` — Clustering algorithm

'kmeans' | 'linkage' | 'gmdistribution' | matrix of clustering solutions | function handle

Clustering algorithm, specified as one of the following.

'kmeans'	Cluster the data in <code>x</code> using the <code>kmeans</code> clustering algorithm, with <code>'EmptyAction'</code> set to <code>'singleton'</code> and <code>'Replicates'</code> set to 5.
'linkage'	Cluster the data in <code>x</code> using the <code>clusterdata</code> agglomerative clustering algorithm, with <code>'Linkage'</code> set to <code>'ward'</code> .
'gmdistribution'	Cluster the data in <code>x</code> using the <code>gmdistribution</code> Gaussian mixture distribution algorithm, with <code>'SharedCov'</code> set to <code>true</code> and <code>'Replicates'</code> set to 5.

If `Criterion` is `'CalinskiHarabasz'`, `'DaviesBouldin'`, or `'silhouette'`, you can specify a clustering algorithm using a function handle (MATLAB). The function must be



of the form `C = clustfun(DATA, K)`, where `DATA` is the data to be clustered, and `K` is the number of clusters. The output of `clustfun` must be one of the following:

- A vector of integers representing the cluster index for each observation in `DATA`. There must be `K` unique values in this vector.
- A numeric  $n$ -by- $K$  matrix of score for  $n$  observations and  $K$  classes. In this case, the cluster index for each observation is determined by taking the largest score value in each row.

If `Criterion` is `'CalinskiHarabasz'`, `'DaviesBouldin'`, or `'silhouette'`, you can also specify `clust` as a  $n$ -by- $K$  matrix containing the proposed clustering solutions.  $n$  is the number of observations in the sample data, and  $K$  is the number of proposed clustering solutions. Column  $j$  contains the cluster indices for each of the  $N$  points in the  $j$ th clustering solution.

#### **criterion** — Clustering evaluation criterion

`'CalinskiHarabasz'` | `'DaviesBouldin'` | `'gap'` | `'silhouette'`

Clustering evaluation criterion, specified as one of the following.

<code>'CalinskiHarabasz'</code>	Create a <code>CalinskiHarabaszEvaluation</code> clustering evaluation object containing Calinski-Harabasz index values.
<code>'DaviesBouldin'</code>	Create a <code>DaviesBouldinEvaluation</code> cluster evaluation object containing Davies-Bouldin index values.
<code>'gap'</code>	Create a <code>GapEvaluation</code> cluster evaluation object containing gap criterion values.
<code>'silhouette'</code>	Create a <code>SilhouetteEvaluation</code> cluster evaluation object containing silhouette values.

## **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'KList', [1:5], 'Distance', 'cityblock'` specifies to test 1, 2, 3, 4, and 5 clusters using the sum of absolute differences distance measure.

#### For All Criteria

#### **KList** — List of number of clusters to evaluate

vector

List of number of clusters to evaluate, specified as the comma-separated pair consisting of `'KList'` and a vector of positive integer values. You must specify `KList` when `clust` is a clustering algorithm name or a function handle. When `criterion` is `'gap'`, `clust` must be a character vector or a function handle, and you must specify `KList`.

Example: `'KList', [1:6]`

#### For Silhouette and Gap

#### **Distance** — Distance metric

`'sqEuclidean'` (default) | `'Euclidean'` | `'cityblock'` | vector | function | ...

Distance metric used for computing the criterion values, specified as the comma-separated pair consisting of `'Distance'` and one of the following.

<code>'sqEuclidean'</code>	Squared Euclidean distance
<code>'Euclidean'</code>	Euclidean distance. This option is not valid for the <code>kmeans</code> clustering algorithm.
<code>'cityblock'</code>	Sum of absolute differences
<code>'cosine'</code>	One minus the cosine of the included angle between points (treated as vectors)
<code>'correlation'</code>	One minus the sample correlation between points (treated as sequences of values)
<code>'Hamming'</code>	Percentage of coordinates that differ. This option is only valid for the <code>Silhouette</code> criterion.
<code>'Jaccard'</code>	Percentage of nonzero coordinates that differ. This option is only valid for the <code>Silhouette</code> criterion.

For detailed information about each distance metric, see `pdist`.

You can also specify a function for the distance metric using a function handle (MATLAB). The distance function must be of the form `d2 = distfun(XI, XJ)`, where

$XI$  is a 1-by- $n$  vector corresponding to a single row of the input matrix  $X$ , and  $XJ$  is an  $m_2$ -by- $n$  matrix corresponding to multiple rows of  $X$ . `distfun` must return an  $m_2$ -by-1 vector of distances `d2`, whose  $k$ th element is the distance between  $XI$  and  $XJ(k, :)$ .

If `Criterion` is 'silhouette', you can also specify `Distance` as the output vector output created by the function `pdist`.

When `Clust` a character vector representing a built-in clustering algorithm, `evalclusters` uses the distance metric specified for `Distance` to cluster the data, except for the following:

- If `Clust` is 'linkage', and `Distance` is either 'sqEuclidean' or 'Euclidean', then the clustering algorithm uses Euclidean distance and Ward linkage.
- If `Clust` is 'linkage' and `Distance` is any other metric, then the clustering algorithm uses the specified distance metric and average linkage.

In all other cases, the distance metric specified for `Distance` must match the distance metric used in the clustering algorithm to obtain meaningful results.

`Distance` only accepts a function handle if the clustering algorithm `clust` accepts a function handle as the distance metric. For example, the `kmeans` clustering algorithm does not accept a function handle as the distance metric. Therefore, if you use the `kmeansalgorithm` and then specify a function handle for `Distance`, the software errors.

Example: 'Distance', 'Euclidean'

### For Silhouette Only

#### ClusterPriors — Prior probabilities for each cluster

'empirical' (default) | 'equal'

Prior probabilities for each cluster, specified as the comma-separated pair consisting of 'ClusterPriors' and one of the following.

'empirical'	Compute the overall silhouette value for the clustering solution by averaging the silhouette values for all points. Each cluster contributes to the overall silhouette value proportionally to its size.
-------------	--

`'equal'` Compute the overall silhouette value for the clustering solution by averaging the silhouette values for all points within each cluster, and then averaging those values across all clusters. Each cluster contributes equally to the overall silhouette value, regardless of its size.

Example: `'ClusterPriors', 'empirical'`

### For Gap Only

#### **B — Number of reference data sets**

100 (default) | positive integer value

Number of reference data sets generated from the reference distribution `ReferenceDistribution`, specified as the comma-separated pair consisting of `'B'` and a positive integer value.

Example: `'B', 150`

#### **ReferenceDistribution — Reference data generation method**

`'PCA'` (default) | `'uniform'`

Reference data generation method, specified as the comma-separated pair consisting of `'ReferenceDistributions'` and one of the following.

`'PCA'` Generate reference data from a uniform distribution over a box aligned with the principal components of the data matrix  $x$ .

`'uniform'` Generate reference data uniformly over the range of each feature in the data matrix  $x$ .

Example: `'ReferenceDistribution', 'uniform'`

#### **SearchMethod — Method for selecting optimal number of clusters**

`'globalMaxSE'` (default) | `'firstMaxSE'`

Method for selecting the optimal number of clusters, specified as the comma-separated pair consisting of `'SearchMethod'` and one of the following.

'globalMaxSE'	Evaluate each proposed number of clusters in <code>KList</code> and select the smallest number of clusters satisfying
	$\text{Gap}(K) \geq \text{GAPMAX} - \text{SE}(\text{GAPMAX}),$
	where $K$ is the number of clusters, $\text{Gap}(K)$ is the gap value for the clustering solution with $K$ clusters, $\text{GAPMAX}$ is the largest gap value, and $\text{SE}(\text{GAPMAX})$ is the standard error corresponding to the largest gap value.
'firstMaxSE'	Evaluate each proposed number of clusters in <code>KList</code> and select the smallest number of clusters satisfying
	$\text{Gap}(K) \geq \text{Gap}(K + 1) - \text{SE}(K + 1),$
	where $K$ is the number of clusters, $\text{Gap}(K)$ is the gap value for the clustering solution with $K$ clusters, and $\text{SE}(K + 1)$ is the standard error of the clustering solution with $K + 1$ clusters.

Example: `'SearchMethod', 'globalMaxSE'`

## Output Arguments

### **eva** — Clustering evaluation data

clustering evaluation object

Clustering evaluation data, returned as a clustering evaluation object.

## See Also

`CalinskiHarabaszEvaluation` | `DaviesBouldinEvaluation` | `GapEvaluation` | `SilhouetteEvaluation`

## Topics

“Clustering Using Gaussian Mixture Models” on page 16-28

“k-Means Clustering” on page 16-21

“Hierarchical Clustering” on page 16-3

“Gaussian Mixture Models” on page 5-148

**Introduced in R2013b**

## evlike

Extreme value negative log-likelihood

### Syntax

```
nlogL = evlike(params,data)
[nlogL,AVAR] = evlike(params,data)
[...] = evlike(params,data,censoring)
[...] = evlike(params,data,censoring,freq)
```

### Description

`nlogL = evlike(params,data)` returns the negative of the log-likelihood for the type 1 extreme value distribution. `params(1)` is the tail location parameter, `mu`, and `params(2)` is the scale parameter, `sigma`. `nlogL` is a scalar.

`[nlogL,AVAR] = evlike(params,data)` returns the inverse of Fisher's information matrix, `AVAR`. If the input parameter values in `params` are the maximum likelihood estimates, the diagonal elements of `AVAR` are their asymptotic variances. `AVAR` is based on the observed Fisher's information, not the expected information.

`[...] = evlike(params,data,censoring)` accepts a Boolean vector of the same size as `data`, which is 1 for observations that are right-censored and 0 for observations that are observed exactly.

`[...] = evlike(params,data,censoring,freq)` accepts a frequency vector of the same size as `data`. `freq` typically contains integer frequencies for the corresponding elements in `data`, but can contain any nonnegative values. Pass in `[]` for `censoring` to use its default value.

The type 1 extreme value distribution is also known as the Gumbel distribution. The version used here is suitable for modeling minima; the mirror image of this distribution can be used to model maxima by negating `data`. See “Extreme Value Distribution” on page B-38 for more details. If  $x$  has a Weibull distribution, then  $X = \log(x)$  has the type 1 extreme value distribution.

## See Also

`evcdf` | `evfit` | `evinv` | `evpdf` | `evrnd` | `evstat`

## Topics

“Extreme Value Distribution” on page B-38

**Introduced before R2006a**



## evpdf

Extreme value probability density function

### Syntax

```
Y = evpdf(X,mu,sigma)
```

### Description

`Y = evpdf(X,mu,sigma)` returns the pdf of the type 1 extreme value distribution with location parameter `mu` and scale parameter `sigma`, evaluated at the values in `X`. `X`, `mu`, and `sigma` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array of the same size as the other inputs. The default values for `mu` and `sigma` are 0 and 1, respectively.

The type 1 extreme value distribution is also known as the Gumbel distribution. The version used here is suitable for modeling minima; the mirror image of this distribution can be used to model maxima by negating `x`. See “Extreme Value Distribution” on page B-38 for more details. If `x` has a Weibull distribution, then `X = log(x)` has the type 1 extreme value distribution.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`evcdf` | `evfit` | `evinv` | `evlike` | `evrnd` | `evstat` | `pdf`

## **Topics**

“Extreme Value Distribution” on page B-38

**Introduced before R2006a**

## evrnd

Extreme value random numbers

### Syntax

```
R = evrnd(mu, sigma)
R = evrnd(mu, sigma, m, n, ...)
R = evrnd(mu, sigma, [m, n, ...])
```

### Description

`R = evrnd(mu, sigma)` generates random numbers from the extreme value distribution with parameters specified by location parameter `mu` and scale parameter `sigma`. `mu` and `sigma` can be vectors, matrices, or multidimensional arrays that have the same size, which is also the size of `R`. A scalar input for `mu` or `sigma` is expanded to a constant array with the same dimensions as the other input.

`R = evrnd(mu, sigma, m, n, ...)` or `R = evrnd(mu, sigma, [m, n, ...])` generates an `m`-by-`n`-by-... array containing random numbers from the extreme value distribution with parameters `mu` and `sigma`. `mu` and `sigma` can each be scalars or arrays of the same size as `R`.

The type 1 extreme value distribution is also known as the Gumbel distribution. The version used here is suitable for modeling minima; the mirror image of this distribution can be used to model maxima by negating `R`. See “Extreme Value Distribution” on page B-38 for more details. If  $x$  has a Weibull distribution, then  $X = \log(x)$  has the type 1 extreme value distribution.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

The generated code can return a different sequence of numbers than MATLAB if either of the following is true:

- The output is nonscalar.
- An input parameter is invalid for the distribution.

## See Also

`evcdf` | `evfit` | `evinv` | `evlike` | `evpdf` | `evstat` | `random`

## Topics

“Extreme Value Distribution” on page B-38

**Introduced before R2006a**

## evstat

Extreme value mean and variance

### Syntax

```
[M,V] = evstat(mu,sigma)
```

### Description

`[M,V] = evstat(mu,sigma)` returns the mean of and variance for the type 1 extreme value distribution with location parameter `mu` and scale parameter `sigma`. `mu` and `sigma` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array of the same size as the other input. The default values for `mu` and `sigma` are 0 and 1, respectively.

The type 1 extreme value distribution is also known as the Gumbel distribution. The version used here is suitable for modeling minima; the mirror image of this distribution can be used to model maxima. See “Extreme Value Distribution” on page B-38 for more details. If  $x$  has a Weibull distribution, then  $X = \log(x)$  has the type 1 extreme value distribution.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`evcdf` | `evfit` | `evinv` | `evlike` | `evpdf` | `evrnd`

## **Topics**

“Extreme Value Distribution” on page B-38

**Introduced before R2006a**

# expcdf

Exponential cumulative distribution function

## Syntax

```
p = expcdf(x,mu)
[p,plo,pup] = expcdf(x,mu,pcov,alpha)
[p,plo,pup] = expcdf( ____, 'upper')
```

## Description

`p = expcdf(x,mu)` computes the exponential cdf at each of the values in `x` using the corresponding mean parameter `mu`. `x` and `mu` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array with the same dimensions as the other input. The parameters in `mu` must be positive.

`[p,plo,pup] = expcdf(x,mu,pcov,alpha)` produces confidence bounds for `Pp` when the input mean parameter `mu` is an estimate. `pcov` is the variance of the estimated `mu`. `alpha` specifies `100(1 - alpha)%` confidence bounds. The default value of `alpha` is 0.05. `plo` and `pup` are arrays of the same size as `p` containing the lower and upper confidence bounds. The bounds are based on a normal approximation for the distribution of the log of the estimate of `mu`. If you estimate `mu` from a set of data, you can get a more accurate set of bounds by applying `expfit` to the data to get a confidence interval for `mu`, and then evaluating `expinv` at the lower and upper endpoints of that interval.

`[p,plo,pup] = expcdf( ____, 'upper')` returns the complement of the exponential cdf at each value in `x`, using an algorithm that more accurately computes the extreme upper tail probabilities. You can use the 'upper' argument with any of the prior syntaxes.

The exponential cdf is

$$p = F(x | u) = \int_0^x \frac{1}{\mu} e^{-\frac{t}{\mu}} dt = 1 - e^{-\frac{x}{\mu}}$$

The result,  $p$ , is the probability that a single observation from an exponential distribution will fall in the interval  $[0 x]$ .

## Examples

### Compute the Exponential CDF

The following code shows that the median of the exponential distribution is  $\mu \log(2)$ .

```
mu = 10:10:60;  
p = expcdf(log(2)*mu,mu)
```

```
p =
```

```
    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000
```

What is the probability that an exponential random variable is less than or equal to the mean,  $\mu$ ?

```
mu = 1:6;  
x = mu;  
p = expcdf(x,mu)
```

```
p =
```

```
    0.6321    0.6321    0.6321    0.6321    0.6321    0.6321
```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.



## See Also

`cdf` | `expfit` | `expinv` | `explike` | `exppdf` | `exprnd` | `expstat`

## Topics

“Exponential Distribution” on page B-34

**Introduced before R2006a**

## expfit

Exponential parameter estimates

### Syntax

```
muhat = expfit(data)
[muhat,muci] = expfit(data)
[muhat,muci] = expfit(data,alpha)
[...] = expfit(data,alpha,censoring)
[...] = expfit(data,alpha,censoring,freq)
```

### Description

`muhat = expfit(data)` estimates the mean of an exponentially distributed sample data. Each entry of `muhat` corresponds to the data in a column of `data`.

`[muhat,muci] = expfit(data)` returns 95% confidence intervals for the mean parameter estimates in matrix `muci`. The first row of `muci` contains the lower bounds of the confidence intervals, and the second row contains the upper bounds.

`[muhat,muci] = expfit(data,alpha)` returns  $100(1 - \alpha)\%$  confidence intervals for the parameter estimates, where `alpha` is a value in the range `[0 1]` specifying the width of the confidence intervals. By default, `alpha` is `0.05`, which corresponds to 95% confidence intervals.

`[...] = expfit(data,alpha,censoring)` accepts a Boolean vector, `censoring`, of the same size as `data`, which is 1 for observations that are right-censored and 0 for observations that are observed exactly. `data` must be a vector in order to pass in the argument `censoring`.

`[...] = expfit(data,alpha,censoring,freq)` accepts a frequency vector, `freq` of the same size as `data`. Typically, `freq` contains integer frequencies for the corresponding elements in `data`, but can contain any nonnegative values. Pass in `[]` for `alpha`, `censoring`, or `freq` to use their default values.

## Examples

The following estimates the mean  $\mu$  of exponentially distributed data, and returns a 95% confidence interval for the estimate:

```
mu = 3;
data = exprnd(mu,100,1); % Simulated data

[muhat,muci] = expfit(data)
muhat =
    2.7511
muci =
    2.2826
    3.3813
```

## See Also

[expcdf](#) | [expinv](#) | [explike](#) | [exppdf](#) | [exprnd](#) | [expstat](#) | [mle](#)

**Introduced before R2006a**

## ExhaustiveSearcher

Prepare exhaustive nearest neighbor searcher

### Syntax

```
Mdl = ExhaustiveSearcher(X)
Mdl = ExhaustiveSearcher(X,Name,Value)
```

### Description

`Mdl = ExhaustiveSearcher(X)` prepares an exhaustive nearest neighbor searcher (`Mdl`) to find the nearest neighbors of query data using the  $n$ -by- $K$  numeric matrix of training data (`X`). `Mdl` is an `ExhaustiveSearcher` model object that stores the statistics and options required for an exhaustive nearest neighbor search. You can use `Mdl` to search the training data (`X`) for the nearest neighbors to the query data.

`Mdl = ExhaustiveSearcher(X,Name,Value)` prepares an exhaustive nearest neighbor searcher with additional options specified by one or more `Name,Value` pair arguments. For example, you can specify a distance metric or distance metric parameters values.

### Examples

#### Train Default Exhaustive Nearest Neighbors Searcher

Load Fisher's iris data set.

```
load fisheriris
X = meas;
[n,k] = size(X)
```

```
n =
```

```

150

k =

    4

```

X has 150 observations and 4 predictors.

Prepare an exhaustive nearest neighbors searcher using the entire data set as training data.

```
Mdl = ExhaustiveSearcher(X)
```

```

Mdl =

    ExhaustiveSearcher with properties:

        Distance: 'euclidean'
    DistParameter: []
           X: [150x4 double]

```

Mdl is an ExhaustiveSearcher model object, and its properties appear in the Command Window. It contains information about the trained algorithm, such as the distance metric. You can alter property values using dot notation.

To search X for the nearest neighbors to a batch of query data, pass Mdl and the query data to knnsearch or rangesearch.

### Specify the Mahalanobis Distance for Nearest Neighbor Search

Load Fisher's iris data. Focus on the petal dimensions.

```

load fisheriris
X = meas(:, [3 4]); % Predictors

```

Prepare an exhaustive, nearest neighbors searcher. Specify to use the Mahalanobis distance metric.

```
Mdl = createns(X, 'NSMethod', 'exhaustive', 'Distance', 'mahalanobis')
```

```
Mdl =  
  
ExhaustiveSearcher with properties:  
  
    Distance: 'mahalanobis'  
    DistParameter: [2x2 double]  
    X: [150x2 double]
```

Mdl is an `ExhaustiveSearcher` model object. Access properties of Mdl using dot notation. For example, use `Mdl.DistParameter` to access the Mahalanobis covariance parameter.

```
Mdl.DistParameter
```

```
ans =  
  
    3.1163    1.2956  
    1.2956    0.5810
```

You can pass query data and Mdl to:

- `knnsearch` to find indices and distances of nearest neighbors.
- `rangesearch` to find indices of all nearest neighbors within a distance that you specify.

### Search for Nearest Neighbors of Query Data Using the Mahalanobis Distance

Load Fisher's iris data set.

```
load fisheriris
```

Remove five irises randomly from the predictor data to use as a query set.

```
rng('default');           % For reproducibility  
n = size(meas,1);         % Sample size  
qIdx = randsample(n,5);   % Indices of query data  
X = meas(~ismember(1:n,qIdx),:);  
Y = meas(qIdx,:);
```

Prepare an exhaustive nearest neighbors searcher using the training data. Specify to use the Mahalanobis distance for finding nearest neighbors later.

```
Mdl = createns(X, 'NSMethod', 'exhaustive', 'Distance', 'mahalanobis')
```

```
Mdl =
```

```
ExhaustiveSearcher with properties:
```

```
    Distance: 'mahalanobis'
  DistParameter: [4x4 double]
           X: [145x4 double]
```

Mdl is an ExhaustiveSearcher model object. By default, the Mahalanobis metric parameter value is the estimated covariance matrix of the predictors (columns) in the training data. To display this value, use Mdl.DistParameter.

```
Mdl.DistParameter
```

```
ans =
```

```
    0.6547    -0.0368     1.2320     0.5026
   -0.0368     0.1914    -0.3227    -0.1193
    1.2320    -0.3227     3.0671     1.2842
    0.5026    -0.1193     1.2842     0.5800
```

Find the indices of the training data (Mdl.X) that are the two nearest neighbors of each point in the query data (Y).

```
IdxNN = knnsearch(Mdl, Y, 'K', 2)
```

```
IdxNN =
```

```
     5     6
    98    95
   104   128
   135    65
   102   115
```

Each row of `NN` corresponds to a query data observation. The column order corresponds to the order of the nearest neighbors with respect to ascending distance. For example, using the Mahalanobis metric, the second nearest neighbor of `Y(3, :)` is `X(34, :)`.

## Input Arguments

### **x** — Training data

numeric matrix

Training data for an exhaustive nearest neighbor search, specified as a numeric matrix. `X` has  $n$  rows, each corresponding to an observation (i.e. an instance or example), and  $K$  columns, each corresponding to a predictor or feature.

Data Types: `single` | `double`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'Distance', 'mahalanobis', 'Cov', eye(3)` specifies to use the Mahalanobis distance when searching for nearest neighbors, and a 3-by-3 identity matrix for the covariance matrix in the Mahalanobis distance metric.

### **Cov** — Covariance matrix for Mahalanobis distance metric

`nancov(X)` (default) | positive definite matrix

Covariance matrix for the Mahalanobis distance metric, specified as the comma-separated pair consisting of `'Cov'` and a positive definite matrix. `Cov` is a  $K$ -by- $K$  matrix, where  $K$  is the number of columns of `X`. If you specify `Cov` and do not specify `'Distance', 'mahalanobis'`, then `ExhaustiveSearcher` throws an error.

Example: `'Cov', eye(3)`

Data Types: `double` | `single`



**Distance — Distance metric**

'euclidean' (default) | 'chebychev' | 'cityblock' | 'correlation' | 'cosine' | 'hamming' | 'jaccard' | 'mahalanobis' | 'minkowski' | 'seuclidean' | 'spearman' | custom distance function

Distance metric used to find nearest neighbors of query points, specified as a comma-separated pair consisting of 'Distance' and a character vector or function handle.

This table describes the supported distance metrics specified by character vectors.

Value	Description
'chebychev'	Chebychev distance (maximum coordinate difference).
'cityblock'	City block distance
'correlation'	One minus the sample linear correlation between observations (treated as sequences of values)
'cosine'	One minus the cosine of the included angle between observations (row vectors)
'euclidean'	Euclidean distance
'hamming'	Hamming distance, which is the percentage of coordinates that differ
'jaccard'	One minus the Jaccard coefficient, which is the percentage of nonzero coordinates that differ
'minkowski'	Minkowski distance
'mahalanobis'	Mahalanobis distance
'seuclidean'	Standardized Euclidean distance
'spearman'	One minus the sample Spearman's rank correlation between observations (treated as sequences of values)

For more details, see “Distance Metrics” on page 18-16.

You can specify a function handle for a custom distance metric using @ (for example, @distfun). The custom distance function must:

- Have the form `function D2 = distfun(ZI, ZJ)`.
- Take as arguments:
  - A 1-by- $K$  vector  $ZI$  containing a single row from  $X$  or from the query points  $Y$
  - An  $m$ -by- $K$  matrix  $ZJ$  containing multiple rows of  $X$  or  $Y$
- Return an  $m$ -by-1 vector of distances  $D2$ .  $D2(j)$  is the distance between the observations  $ZI$  and  $ZJ(j, :)$ .

The software does not use the distance metric for training the exhaustive searcher algorithm. Therefore, you can alter it after training by specifying a supported character vector or function handle for a custom function using dot notation.

Example: `'Distance', 'mahalanobis'`

Data Types: `char` | `function_handle`

### **P** — Exponent for Minkowski distance metric

2 (default) | positive scalar

Exponent for the Minkowski distance metric, specified as the comma-separated pair consisting of `'P'` and a positive scalar. If you specify `P` and do not specify `'Distance', 'minkowski'`, then the software throws an error.

Example: `'P', 3`

Data Types: `double` | `single`

### **scale** — Scale parameter value for standard Euclidean distance metric

`nanstd(X)` (default) | nonnegative numeric vector

Scale parameter value for the standard Euclidean distance metric, specified as the comma-separated pair consisting of `'Scale'` and a nonnegative numeric vector. `Scale` has length  $K$ , where  $K$  is the number of columns of  $X$ .

The software scales each difference between the training and query data using the corresponding element of `Scale`. If you specify `Scale` and do not specify `'Distance', 'seuclidean'`, then `ExhaustiveSearcher` throws an error.

Example: `'Scale', quantile(X,0.75) - quantile(X,0.25)`

Data Types: `double` | `single`

## Output Arguments

### **mdl** — Prepared exhaustive nearest neighbor searcher

ExhaustiveSearcher model object

Prepared exhaustive nearest neighbor searcher, returned as an ExhaustiveSearcher model object. To search the training data for the nearest neighbors of the query data, pass the query data and Mdl to `knnsearch` or `rangesearch`.

## See Also

`createns` | `knnsearch` | `rangesearch`

## Topics

ExhaustiveSearcher

“k-Nearest Neighbor Search and Radius Search” on page 18-18

“Distance Metrics” on page 18-16

**Introduced in R2010a**

# ExhaustiveSearcher

Exhaustive nearest neighbor searcher

## Description

`ExhaustiveSearcher` model objects store statistics and options for an exhaustive, nearest neighbor search. Statistics and options that you can store include the training data, the distance metric, and the parameter values of the distance metric. The exhaustive search algorithm finds the distance from each query observation to all  $n$  observations in the training data, which is an  $n$ -by- $K$  numeric matrix.

Once you create an `ExhaustiveSearcher` model object, find neighboring points in the training data to the query data by performing a nearest neighbor search using `knnsearch` or a radius search using `rangesearch`. The exhaustive search algorithm is more efficient than the *Kd*-tree algorithm when  $K$  is large (i.e.,  $K \geq 10$ ), and it is more flexible than the *Kd*-tree algorithm with respect to distance metric choices. The algorithm also supports sparse data.

## Creation

Create an `ExhaustiveSearcher` model object using `ExhaustiveSearcher` or `createns`.

## Properties

### **Distance — Distance metric**

character vector | custom distance function

Distance metric used to find nearest neighbors of query points, specified as a character vector or function handle.

This table describes the supported distance metrics specified by character vectors.

Value	Description
'chebychev'	Chebychev distance (maximum coordinate difference)
'cityblock'	City block distance
'correlation'	One minus the sample linear correlation between observations (treated as sequences of values)
'cosine'	One minus the cosine of the included angle between observations (row vectors)
'euclidean'	Euclidean distance
'hamming'	Hamming distance, which is the percentage of coordinates that differ
'jaccard'	One minus the Jaccard coefficient, which is the percentage of nonzero coordinates that differ
'minkowski'	Minkowski distance
'mahalanobis'	Mahalanobis distance
'seuclidean'	Standardized Euclidean distance
'spearman'	One minus the sample Spearman's rank correlation between observations (treated as sequences of values)

For more details, see “Distance Metrics” on page 18-16.

You can specify a function handle for a custom distance metric using @ (for example, @distfun). A custom distance function must:

- Have the form `function D2 = distfun(ZI, ZJ)`.
- Take as arguments:
  - A 1-by- $n$  vector  $ZI$  containing a single row from  $X$  or from the query points  $Y$
  - An  $m$ -by- $n$  matrix  $ZJ$  containing multiple rows of  $X$  or  $Y$
- Return an  $m$ -by-1 vector of distances  $D2$ .  $D2(j)$  is the distance between the observations  $ZI$  and  $ZJ(j, :)$ .

The software does not use the distance metric for training the exhaustive searcher algorithm. Therefore, you can alter it after training by specifying a supported character vector or function handle for a custom function using dot notation. For example, to specify the Mahalanobis distance, enter `Mdl.Distance = 'mahalanobis'`.

Data Types: `char` | `function_handle`

#### **DistParameter** — Distance metric parameter values

`[]` | positive scalar

Distance metric parameter values, specified as empty (`[]`) or as a positive scalar.

This table describes the distance parameters of the supported distance metrics.

Distance Metric	Parameter Description
'mahalanobis'	A positive definite matrix representing the covariance matrix used for computing the Mahalanobis distances. By default, the software sets the covariance using <code>nancov(Mdl.X)</code> . You can alter the scale parameter using dot notation, e.g., <code>Mdl.DistParameter = CovNew</code> , where <code>CovNew</code> is a $K$ -by- $K$ positive definite numeric matrix.
'minkowski'	A positive scalar indicating the exponent of the Minkowski distances. By default, the exponent is 2.

Distance Metric	Parameter Description
'seuclidean'	<p>A positive, numeric vector indicating the values that the software uses to scale the predictors when computing the standardized Euclidean distances. By default, the software:</p> <ol style="list-style-type: none"> <li><b>1</b> Estimates the standard deviation of each predictor (column) of <math>X</math> using <code>scale = nanstd(Mdl.X)</code>.</li> <li><b>2</b> Scales each coordinate difference between the rows in <math>X</math> and the query matrix by dividing by the corresponding element of <code>scale</code>.</li> </ol> <p>You can alter the scale parameter using dot notation, e.g., <code>Mdl.DistanceParameter = sNew</code>, where <code>sNew</code> is a <math>K</math>-dimensional positive numeric vector.</p>

If `Mdl.Distance` is not one of the parameters listed in this table, then `Mdl.DistanceParameter` is `[]`, which means that the specified distance metric formula has no parameters.

Data Types: `single` | `double`

### **x** — Training data

numeric matrix

This property is read-only.

Training data that prepares the exhaustive searcher algorithm, specified as a numeric matrix.  $X$  has  $n$  rows, each corresponding to an observation (i.e., instance or example), and  $K$  columns, each corresponding to a predictor or feature.

Data Types: `single` | `double`

## Object Functions

`knnsearch`     `k`-nearest neighbor search using Kd-tree or exhaustive search

`rangesearch` Find all neighbors within specified distance using exhaustive search or Kd-tree

## Examples

### Train Default Exhaustive Nearest Neighbors Searcher

Load Fisher's iris data set.

```
load fisheriris
X = meas;
[n,k] = size(X)
```

```
n =
    150
```

```
k =
     4
```

X has 150 observations and 4 predictors.

Prepare an exhaustive nearest neighbors searcher using the entire data set as training data.

```
Mdl = ExhaustiveSearcher(X)
```

```
Mdl =
    ExhaustiveSearcher with properties:
        Distance: 'euclidean'
        DistParameter: []
        X: [150x4 double]
```



Mdl is an ExhaustiveSearcher model object, and its properties appear in the Command Window. It contains information about the trained algorithm, such as the distance metric. You can alter property values using dot notation.

To search X for the nearest neighbors to a batch of query data, pass Mdl and the query data to knnsearch or rangesearch.

### Alter Properties of ExhaustiveSearcher Model

Load Fisher's iris data set.

```
load fisheriris
X = meas;
```

Train a default exhaustive searcher algorithm using the entire data set as training data.

```
Mdl = ExhaustiveSearcher(X)
```

```
Mdl =
```

```
ExhaustiveSearcher with properties:
```

```
    Distance: 'euclidean'
  DistParameter: []
           X: [150x4 double]
```

Specify that the neighbor searcher use the Mahalanobis metric to compute the distances between the training and query data.

```
Mdl.Distance = 'mahalanobis'
```

```
Mdl =
```

```
ExhaustiveSearcher with properties:
```

```
    Distance: 'mahalanobis'
  DistParameter: [4x4 double]
           X: [150x4 double]
```

Pass `Mdl` and the query data to either `knnsearch` or `rangesearch` to find the nearest neighbors to the points in the query data using the Mahalanobis distance.

### Search for Nearest Neighbors of Query Data Using the Mahalanobis Distance

Load Fisher's iris data set.

```
load fisheriris
```

Remove five irises randomly from the predictor data to use as a query set.

```
rng('default');           % For reproducibility
n = size(meas,1);         % Sample size
qIdx = randsample(n,5);   % Indices of query data
X = meas(~ismember(1:n,qIdx),:);
Y = meas(qIdx,:);
```

Prepare an exhaustive nearest neighbors searcher using the training data. Specify to use the Mahalanobis distance for finding nearest neighbors later.

```
Mdl = createns(X,'NSMethod','exhaustive','Distance','mahalanobis')
```

```
Mdl =
```

```
ExhaustiveSearcher with properties:
```

```
    Distance: 'mahalanobis'
  DistParameter: [4x4 double]
           X: [145x4 double]
```

`Mdl` is an `ExhaustiveSearcher` model object. By default, the Mahalanobis metric parameter value is the estimated covariance matrix of the predictors (columns) in the training data. To display this value, use `Mdl.DistParameter`.

```
Mdl.DistParameter
```

```
ans =
```

```
    0.6547    -0.0368    1.2320    0.5026
   -0.0368    0.1914   -0.3227   -0.1193
```

```

1.2320    -0.3227    3.0671    1.2842
0.5026    -0.1193    1.2842    0.5800

```

Find the indices of the training data (`Mdl.X`) that are the two nearest neighbors of each point in the query data (`Y`).

```
IdxNN = knnsearch(Mdl,Y,'K',2)
```

```
IdxNN =
```

```

     5     6
    98    95
   104   128
   135    65
   102   115

```

Each row of `NN` corresponds to a query data observation. The column order corresponds to the order of the nearest neighbors with respect to ascending distance. For example, using the Mahalanobis metric, the second nearest neighbor of `Y(3, :)` is `X(34, :)`.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Only the `knnsearch` and `rangesearch` functions support code generation.
- When you train an `ExhaustiveSearcher` model object using `ExhaustiveSearcher` or `createns`, the value of the `'Distance'` name-value pair argument cannot be a custom distance function.

### See Also

`KDTreeSearcher`

## **Topics**

“k-Nearest Neighbor Search and Radius Search” on page 18-18

“Distance Metrics” on page 18-16

“Code Generation Workflow for Exhaustive Searcher” on page 31-47

**Introduced in R2010a**

## expinv

Exponential inverse cumulative distribution function

### Syntax

```
X = expinv(P,mu)
[X,XLO,XUP] = expinv(X,mu,pcov,alpha)
```

### Description

`X = expinv(P,mu)` computes the inverse of the exponential cdf with parameters specified by mean parameter `mu` for the corresponding probabilities in `P`. `P` and `mu` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array with the same dimensions as the other input. The parameters in `mu` must be positive and the values in `P` must lie on the interval `[0 1]`.

`[X,XLO,XUP] = expinv(X,mu,pcov,alpha)` produces confidence bounds for `X` when the input mean parameter `mu` is an estimate. `pcov` is the variance of the estimated `mu`. `alpha` specifies `100(1 - alpha)%` confidence bounds. The default value of `alpha` is `0.05`. `XLO` and `XUP` are arrays of the same size as `X` containing the lower and upper confidence bounds. The bounds are based on a normal approximation for the distribution of the log of the estimate of `mu`. If you estimate `mu` from a set of data, you can get a more accurate set of bounds by applying `expfit` to the data to get a confidence interval for `mu`, and then evaluating `expinv` at the lower and upper end points of that interval.

The inverse of the exponential cdf is

$$x = F^{-1}(p | \mu) = -\mu \ln(1 - p)$$

The result, `x`, is the value such that an observation from an exponential distribution with parameter `μ` will fall in the range `[0 x]` with probability `p`.

## Examples

Let the lifetime of light bulbs be exponentially distributed with  $\mu = 700$  hours. What is the median lifetime of a bulb?

```
expinv(0.50,700)
ans =
    485.2030
```

Suppose you buy a box of “700 hour” light bulbs. If 700 hours is the mean life of the bulbs, half of them will burn out in less than 500 hours.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`expcdf` | `expfit` | `explike` | `exppdf` | `exprnd` | `expstat` | `icdf`

### Topics

“Exponential Distribution” on page B-34

**Introduced before R2006a**

# explike

Exponential negative log-likelihood

## Syntax

```
nlogL = explike(param, data)
[nlogL, avar] = explike(param, data)
[...] = explike(param, data, censoring)
[...] = explike(param, data, censoring, freq)
```

## Description

`nlogL = explike(param, data)` returns the negative of the log-likelihood for the exponential distribution. `param` is the mean parameter, `mu`. `nlogL` is a scalar.

`[nlogL, avar] = explike(param, data)` returns the inverse of Fisher's information, `avar`, a scalar. If the input parameter value in `param` is the maximum likelihood estimate, `avar` is its asymptotic variance. `avar` is based on the observed Fisher's information, not the expected information.

`[...] = explike(param, data, censoring)` accepts a Boolean vector, `censoring`, of the same size as `data`, which is 1 for observations that are right-censored and 0 for observations that are observed exactly.

`[...] = explike(param, data, censoring, freq)` accepts a frequency vector, `freq`, of the same size as `data`. The vector `freq` typically contains integer frequencies for the corresponding elements in `data`, but can contain any nonnegative values. Pass in `[]` for `censoring` to use its default value.

## See Also

`expcdf` | `expfit` | `expinv` | `exppdf` | `exprnd` | `expstat`

## **Topics**

“Exponential Distribution” on page B-34

**Introduced before R2006a**



# export

**Class:** dataset

Write dataset array to file

---

**Note** The `dataset` data type might be removed in a future release. To work with heterogeneous data, use the MATLAB® `table` data type instead. See MATLAB `table` documentation for more information.

---

## Syntax

```
export(DS,'file',filename)
export(DS)
export(DS,'file',filename,'Delimiter',delim)
export(DS,'XLSfile',filename)
export(DS,'XPTfile',filename)
export(DS,...,'WriteVarNames',false)
export(DS,...,'WriteObsNames',false)
```

## Description

`export(DS,'file',filename)` writes the dataset array `DS` to a tab-delimited text file, including variable names and observation names, if present. If the observation names exist, the name in the first column of the first line of the file is the first dimension name for the dataset (by default, 'Observations'). `export` overwrites any existing file named `filename`.

`export(DS)` writes to a text file whose default name is the name of the dataset array `DS` appended by `'.txt'`. If `export` cannot construct the file name from the dataset array input, it writes to the file `'dataset.txt'`. `export` overwrites any existing file.

`export(DS,'file',filename,'Delimiter',delim)` writes the dataset array `DS` to a text file using the delimiter `delim`. `delim` must be one of the following:

- ' ' or 'space'
- '\t' or 'tab'
- ',' or 'comma'
- ';' or 'semi'
- '|' or 'bar'

`export(DS, 'XLSfile', filename)` writes the dataset array `DS` to a Microsoft® Excel spreadsheet file, including variable names and observation names (if present). You can specify the 'Sheet' and 'Range' parameter name/value pairs, with parameter values as accepted by the `xlsread` function. Since `export` uses the `xlswrite` function internally, this syntax is only compatible with Microsoft Excel for Windows®, and does not work on a Mac. For more information, see `xlswrite`.

`export(DS, 'XPTfile', filename)` writes the dataset array `DS` to a SAS XPORT format file. When writing to an XPORT format file, variables must be scalar valued. `export` saves observation names to a variable called `obsnames`, unless the `WriteObsNames` parameter described below is `false`. The XPORT format restricts the length of variable names to eight characters; longer variable names are truncated.

`export(DS, ..., 'WriteVarNames', false)` does not write the variable names to the text file. `export(DS, ..., 'WriteVarNames', true)` is the default, writing the names as column headings in the first line of the file.

`export(DS, ..., 'WriteObsNames', false)` does not write the observation names to the text file. `export(DS, ..., 'WriteObsNames', true)` is the default, writing the names as the first column of the file.

In some cases, `export` creates a text file that does not represent `A` exactly, as described below. If you use `dataset` to read the file back into MATLAB, the new dataset array may not have exactly the same contents as the original dataset array. Save `A` as a MAT-file if you need to import it again as a dataset array.

`export` writes out numeric variables using long `g` format, and categorical or character variables as unquoted text.

For non-character variables with more than one column, `export` writes out multiple delimiter-separated fields on each line, and constructs suitable column headings for the first line of the file.

`export` writes out variables that have more than two dimensions as a single empty field in each line of the file.

For cell-valued variables, `export` writes out the contents of each cell only when the cell contains a single row, and writes out a single empty field otherwise.

In some cases, `export` creates a file that cannot be read back into MATLAB using `dataset`. Writing a dataset array that contains a cell-valued variable whose cell contents are not scalars results in a mismatch in the file between the number of fields on each line and the number of column headings on the first line. Writing a dataset array that contains a cell-valued variable whose cell contents are not all the same length results in a different number of fields on each line in the file. Therefore, if you might need to import a dataset array again, save it as a `.mat` file.

## Examples

Move data between external text files and dataset arrays in the MATLAB workspace:

```
A = dataset('file','sat2.dat','delimiter',' ','')
A =
    Test          Gender          Score
    'Verbal'      'Male'          470
    'Verbal'      'Female'        530
    'Quantitative' 'Male'          520
    'Quantitative' 'Female'        480

export(A(A.Score > 500,:), 'file', 'HighScores.txt')

B = dataset('file', 'HighScores.txt', 'delimiter', '\t')
B =
    Test          Gender          Score
    'Verbal'      'Female'        530
    'Quantitative' 'Male'          520
```

## See Also

`dataset`

## exppdf

Exponential probability density function

### Syntax

```
Y = exppdf(X,mu)
```

### Description

`Y = exppdf(X,mu)` returns the pdf of the exponential distribution with mean parameter `mu`, evaluated at the values in `X`. `X` and `mu` can be vectors, matrices, or multidimensional arrays that have the same size. A scalar input is expanded to a constant array with the same dimensions as the other input. The parameters in `mu` must be positive.

The exponential pdf is

$$y = f(x | \mu) = \frac{1}{\mu} e^{-\frac{x}{\mu}}$$

The exponential pdf is the gamma pdf with its first parameter equal to 1.

The exponential distribution is appropriate for modeling waiting times when the probability of waiting an additional period of time is independent of how long you have already waited. For example, the probability that a light bulb will burn out in its next minute of use is relatively independent of how many minutes it has already burned.

### Examples

```
y = exppdf(5,1:5)
y =
    0.0067    0.0410    0.0630    0.0716    0.0736

y = exppdf(1:5,1:5)
```

y =  
0.3679 0.1839 0.1226 0.0920 0.0736

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

[expcdf](#) | [expfit](#) | [expinv](#) | [explike](#) | [exprnd](#) | [expstat](#) | [pdf](#)

### Topics

“Exponential Distribution” on page B-34

**Introduced before R2006a**

## exprnd

Exponential random numbers

### Syntax

```
R = exprnd(mu)
R = exprnd(mu,m,n,...)
R = exprnd(mu,[m,n,...])
```

### Description

`R = exprnd(mu)` generates random numbers from the exponential distribution with mean parameter `mu`. `mu` can be a vector, a matrix, or a multidimensional array. The size of `R` is the size of `mu`.

`R = exprnd(mu,m,n,...)` or `R = exprnd(mu,[m,n,...])` generates an `m`-by-`n`-by-... array containing random numbers from the exponential distribution with mean parameter `mu`. `mu` can be a scalar or an array of the same size as `R`.

### Examples

```
n1 = exprnd(5:10)
n1 =
    7.5943    18.3400    2.7113    3.0936    0.6078    9.5841

n2 = exprnd(5:10,[1 6])
n2 =
    3.2752    1.1110    23.5530    23.4303    5.7190    3.9876

n3 = exprnd(5,2,3)
n3 =
    24.3339    13.5271    1.8788
     4.7932     4.3675    2.6468
```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

The generated code can return a different sequence of numbers than MATLAB if either of the following is true:

- The output is nonscalar.
- An input parameter is invalid for the distribution.

### See Also

`expcdf` | `expfit` | `expinv` | `explike` | `exppdf` | `expstat` | `random`

### Topics

“Exponential Distribution” on page B-34

Introduced before R2006a

## expstat

Exponential mean and variance

### Syntax

```
[m,v] = expstat(mu)
```

### Description

`[m,v] = expstat(mu)` returns the mean of and variance for the exponential distribution with parameters `mu`. `mu` can be a vectors, matrix, or multidimensional array. The mean of the exponential distribution is  $\mu$ , and the variance is  $\mu^2$ .

### Examples

```
[m,v] = expstat([1 10 100 1000])  
m =  
     1     10     100     1000  
v =  
     1     100    10000   1000000
```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`expcdf` | `expfit` | `expinv` | `explike` | `exppdf` | `exprnd` | `expstat`



## **Topics**

“Exponential Distribution” on page B-34

**Introduced before R2006a**

## factoran

Factor analysis

### Syntax

```
lambda = factoran(X,m)
[lambda,psi] = factoran(X,m)
[lambda,psi,T] = factoran(X,m)
[lambda,psi,T,stats] = factoran(X,m)
[lambda,psi,T,stats,F] = factoran(X,m)
[...] = factoran(...,param1, val1,param2, val2,...)
```

### Description

`lambda = factoran(X,m)` returns the maximum likelihood estimate, `lambda`, of the factor loadings matrix, in a common factor analysis model with `m` common factors. `X` is an `n`-by-`d` matrix where each row is an observation of `d` variables. The  $(i, j)$ th element of the `d`-by-`m` matrix `lambda` is the coefficient, or loading, of the  $j$ th factor for the  $i$ th variable. By default, `factoran` calls the function `rotatefactors` to rotate the estimated factor loadings using the 'varimax' option.

`[lambda,psi] = factoran(X,m)` also returns maximum likelihood estimates of the specific variances as a column vector `psi` of length `d`.

`[lambda,psi,T] = factoran(X,m)` also returns the `m`-by-`m` factor loadings rotation matrix `T`.

`[lambda,psi,T,stats] = factoran(X,m)` also returns a structure `stats` containing information relating to the null hypothesis,  $H_0$ , that the number of common factors is `m`. `stats` includes the following fields:

Field	Description
<code>loglike</code>	Maximized log-likelihood value
<code>dfe</code>	Error degrees of freedom = $((d-m)^2 - (d+m))/2$

Field	Description
chisq	Approximate chi-squared statistic for the null hypothesis
p	Right-tail significance level for the null hypothesis

factoran does not compute the `chisq` and `p` fields unless `dfe` is positive and all the specific variance estimates in `psi` are positive (see “Heywood Case” on page 32-1672 below). If `X` is a covariance matrix, then you must also specify the `'nobs'` parameter if you want factoran to compute the `chisq` and `p` fields.

`[lambda,psi,T,stats,F] = factoran(X,m)` also returns, in `F`, predictions of the common factors, known as factor scores. `F` is an `n`-by-`m` matrix where each row is a prediction of `m` common factors. If `X` is a covariance matrix, factoran cannot compute `F`. factoran rotates `F` using the same criterion as for `lambda`.

`[...] = factoran(...,param1,val1,param2,val2,...)` enables you to specify optional parameter name/value pairs to control the model fit and the outputs. The following are the valid parameter/value pairs.

Parameter	Value	
'xtype'	Type of input in the matrix <code>X</code> . 'xtype' can be one of:	
	'data'	Raw data (default)
	'covariance'	Positive definite covariance or correlation matrix
'scores'	Method for predicting factor scores. 'scores' is ignored if <code>X</code> is not raw data.	
	'wls' 'Bartlett'	Synonyms for a weighted least-squares estimate that treats <code>F</code> as fixed (default)
	'regression' 'Thomson'	Synonyms for a minimum mean squared error prediction that is equivalent to a ridge regression
'start'	Starting point for the specific variances <code>psi</code> in the maximum likelihood optimization. Can be specified as:	
	'random'	Chooses <code>d</code> uniformly distributed values on the interval <code>[0,1]</code> .

Parameter	Value	
	'Rsquared'	Chooses the starting vector as a scale factor times $\text{diag}(\text{inv}(\text{corrcoef}(X)))$ (default). For examples, see Jöreskog [2].
	Positive integer	Performs the given number of maximum likelihood fits, each initialized as with 'random'. <code>factoran</code> returns the fit with the highest likelihood.
	Matrix	Performs one maximum likelihood fit for each column of the specified matrix. The <i>i</i> th optimization is initialized with the values from the <i>i</i> th column. The matrix must have <i>d</i> rows.
'rotate'	Method used to rotate factor loadings and scores. 'rotate' can have the same values as the 'Method' parameter of <code>rotatefactors</code> . See the reference page for <code>rotatefactors</code> for a full description of the available methods.	
	'none'	Performs no rotation.
	'equamax'	Special case of the orthomax rotation. Use the 'normalize' on page 32-0 , 'reltol' on page 32-0 , and 'maxit' on page 32-0 parameters to control the details of the rotation.
	'orthomax'	Orthogonal rotation that maximizes a criterion based on the variance of the loadings.  Use the 'coeff' on page 32-0 , 'normalize' on page 32-0 , 'reltol' on page 32-0 , and 'maxit' on page 32-0 parameters to control the details of the rotation.
	'parsimax'	Special case of the orthomax rotation (default). Use the 'normalize' on page 32-0 , 'reltol' on page 32-0 , and 'maxit' parameters to control the details of the rotation.

Parameter	Value	
	'pattern'	Performs either an oblique rotation (the default) or an orthogonal rotation to best match a specified pattern matrix. Use the 'type' parameter to choose the type of rotation. Use the 'target' parameter to specify the pattern matrix.
	'procrustes'	Performs either an oblique (the default) or an orthogonal rotation to best match a specified target matrix in the least squares sense.  Use the 'type' on page 32-0 parameter to choose the type of rotation. Use 'target' on page 32-0 to specify the target matrix.
	'promax'	Performs an oblique procrustes rotation to a target matrix determined by factoran as a function of an orthomax solution.  Use the 'power' on page 32-0 parameter to specify the exponent for creating the target matrix. Because 'promax' uses 'orthomax' internally, you can also specify the parameters that apply to 'orthomax'.
	'quartimax'	Special case of the orthomax rotation (default). Use the 'normalize' on page 32-0 , 'reitol' on page 32-0 , and 'maxit' parameters to control the details of the rotation.
	'varimax'	Special case of the orthomax rotation (default). Use the 'normalize' on page 32-0 , 'reitol' on page 32-0 , and 'maxit' parameters to control the details of the rotation.

Parameter	Value
	<p>Function</p> <p>Function handle to rotation function of the form</p> <pre>[B,T] = myrotation(A,...)</pre> <p>where A is a d-by-m matrix of unrotated factor loadings, B is a d-by-m matrix of rotated loadings, and T is the corresponding m-by-m rotation matrix.</p> <p>Use the factoran parameter 'userargs' on page 32-0 to pass additional arguments to this rotation function. See “User-Defined Rotation Function” on page 32-1670.</p>
'coeff'	Coefficient, often denoted as $\gamma$ , defining the specific 'orthomax' criterion. Must be from 0 to 1. The value 0 corresponds to quartimax, and 1 corresponds to varimax. Default is 1.
'normalize'	Flag indicating whether the loading matrix should be row-normalized (1) or left unnormalized (0) for 'orthomax' or 'varimax' rotation. Default is 1.
'reltol'	Relative convergence tolerance for 'orthomax' or 'varimax' rotation. Default is $\sqrt{\text{eps}}$ .
'maxit'	Iteration limit for 'orthomax' or 'varimax' rotation. Default is 250.
'target'	Target factor loading matrix for 'procrustes' rotation. Required for 'procrustes' rotation. No default value.
'type'	Type of 'procrustes' rotation. Can be 'oblique' (default) or 'orthogonal'.
'power'	Exponent for creating the target matrix in the 'promax' rotation. Must be $\geq 1$ . Default is 4.

Parameter	Value
'userargs'	Denotes the beginning of additional input values for a user-defined rotation function. <code>factoran</code> appends all subsequent values, in order and without processing, to the rotation function argument list, following the unrotated factor loadings matrix $A$ . See “User-Defined Rotation Function” on page 32-1670.
'nobs'	If $X$ is a covariance or correlation matrix, indicates the number of observations that were used in its estimation. This allows calculation of significance for the null hypothesis even when the original data are not available. There is no default. 'nobs' is ignored if $X$ is raw data.
'delta'	Lower bound for the specific variances $\psi_i$ during the maximum likelihood optimization. Default is 0.005.
'optimopts'	Structure that specifies control parameters for the iterative algorithm the function uses to compute maximum likelihood estimates. Create this structure with the function <code>statset</code> . Enter <code>statset('factoran')</code> to see the names and default values of the parameters that <code>factoran</code> accepts in the options structure. See the reference page for <code>statset</code> for more information about these options.

## Examples

### Estimate and Plot Factor Loadings

Load the sample data.

```
load carbig
```

Define the variable matrix.

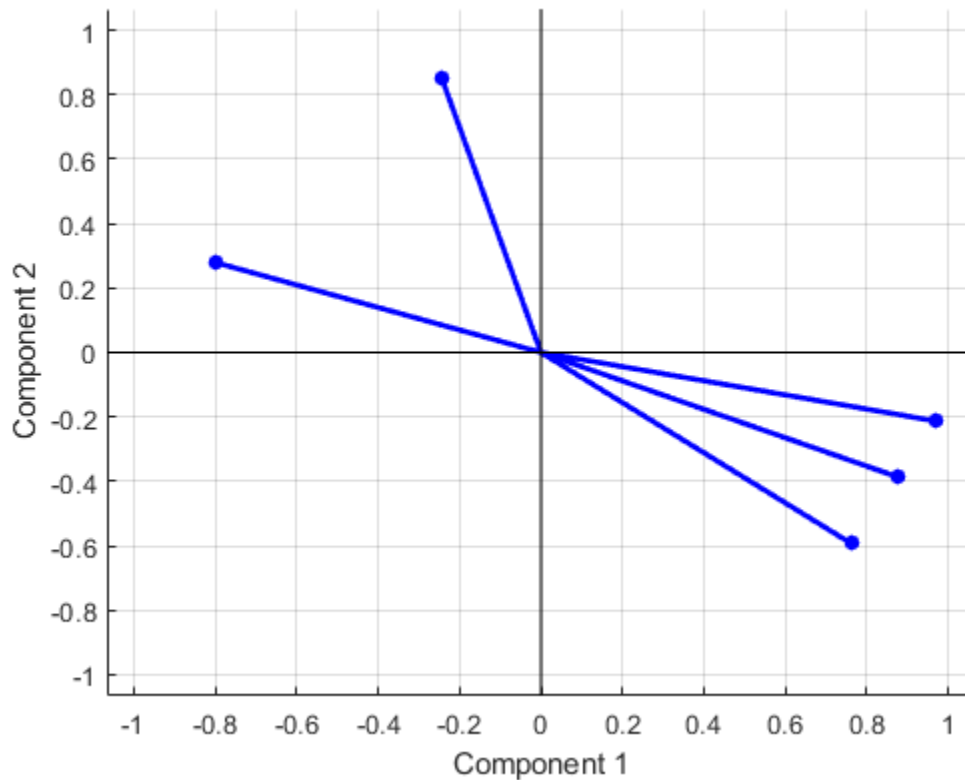
```
X = [Acceleration Displacement Horsepower MPG Weight];
X = X(all(~isnan(X),2),:);
```

Estimate the factor loadings using a minimum mean squared error prediction for a factor analysis with two common factors.

```
[Lambda,Psi,T,stats,F] = factoran(X,2,'scores','regression');
inv(T'*T); % Estimated correlation matrix of F, == eye(2)
Lambda*Lambda' + diag(Psi); % Estimated correlation matrix
Lambda*inv(T); % Unrotate the loadings
F*T'; % Unrotate the factor scores
```

Create biplot of two factors.

```
biplot(Lambda,'LineWidth',2,'MarkerSize',20)
```



Estimate the factor loadings using the covariance (or correlation) matrix.

```
[Lambda,Psi,T] = factoran(cov(X),2,'xtype','cov')
% [Lambda,Psi,T] = factoran(corrcoef(X),2,'xtype','cov')
```



Lambda =

```
-0.2432  -0.8500
 0.8773   0.3871
 0.7618   0.5930
-0.7978  -0.2786
 0.9692   0.2129
```

Psi =

```
0.2184
0.0804
0.0680
0.2859
0.0152
```

T =

```
0.9476  0.3195
0.3195 -0.9476
```

Although the estimates are the same, the use of a covariance matrix rather than raw data doesn't let you request scores or significance level.

Use promax rotation.

```
[Lambda,Psi,T,stats,F] = factoran(X,2,'rotate','promax',...
                                   'powerpm',4);
inv(T'*T)                          % Estimated correlation of F,
                                   % no longer eye(2)
Lambda*inv(T'*T)*Lambda'+diag(Psi) % Estimated correlation of X
```

ans =

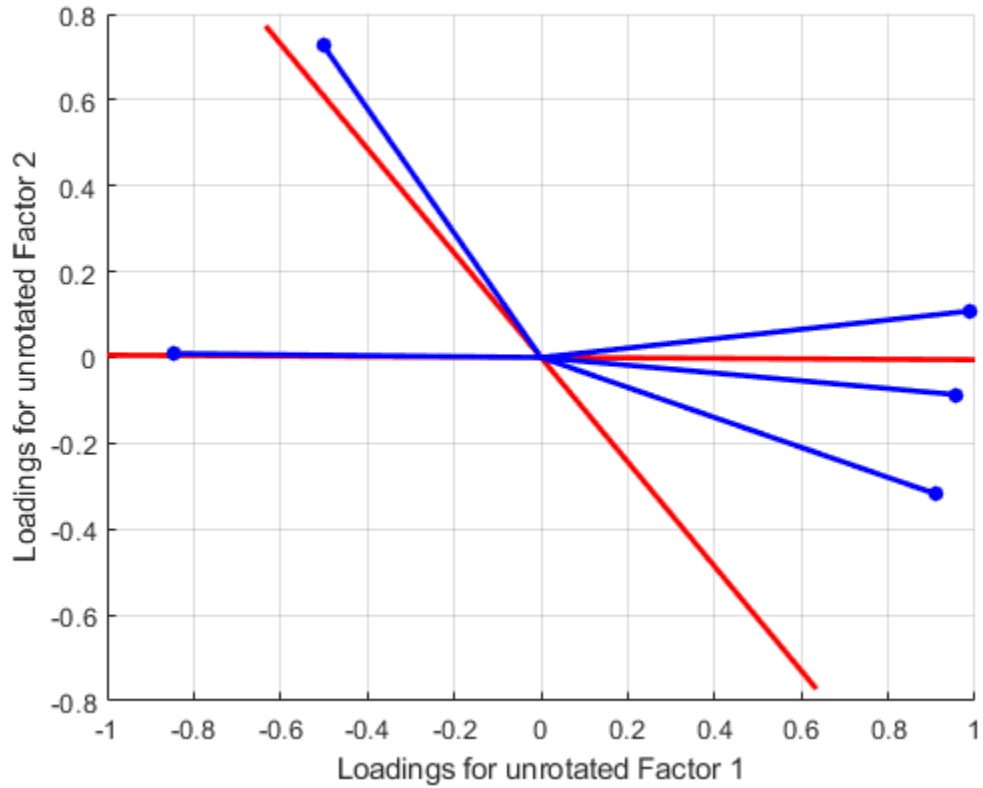
```
1.0000  -0.6391
-0.6391  1.0000
```

ans =

```
1.0000    -0.5424    -0.6893     0.4309    -0.4167
-0.5424     1.0000     0.8979    -0.8078     0.9328
-0.6893     0.8979     1.0000    -0.7730     0.8647
 0.4309    -0.8078    -0.7730     1.0000    -0.8326
-0.4167     0.9328     0.8647    -0.8326     1.0000
```

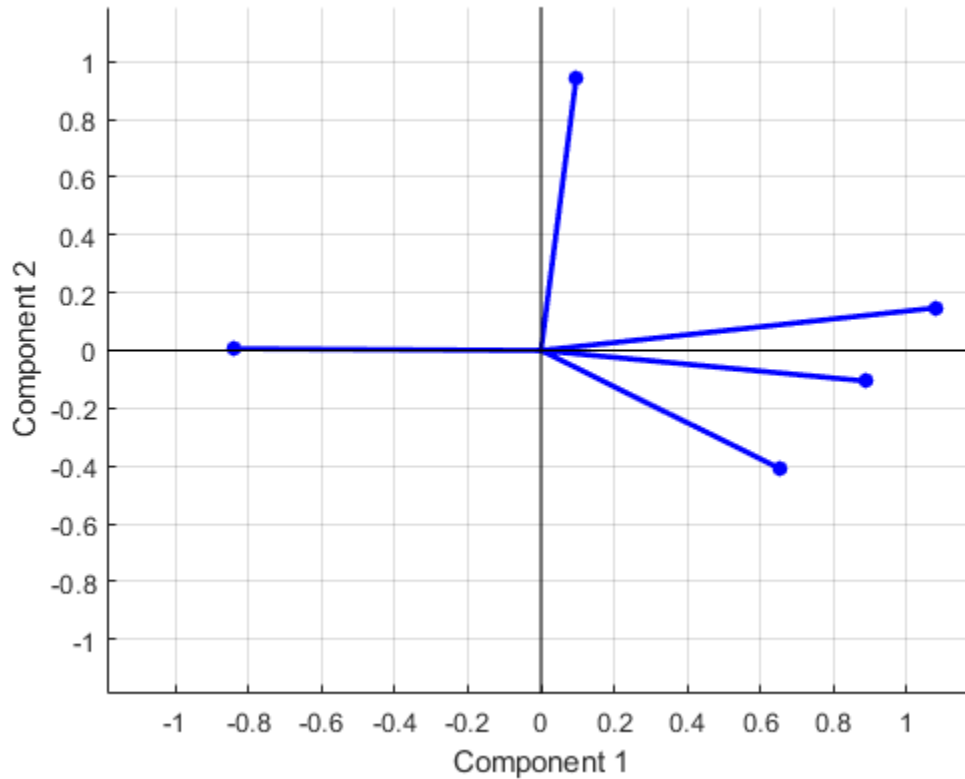
Plot the unrotated variables with oblique axes superimposed.

```
invT = inv(T);
Lambda0 = Lambda*invT;
figure()
line([-invT(1,1) invT(1,1) NaN -invT(2,1) invT(2,1)], ...
      [-invT(1,2) invT(1,2) NaN -invT(2,2) invT(2,2)], ...
      'Color','r','linewidth',2)
grid on
hold on
biplot(Lambda0,'LineWidth',2,'MarkerSize',20)
xlabel('Loadings for unrotated Factor 1')
ylabel('Loadings for unrotated Factor 2')
```



Plot the rotated variables against the oblique axes.

```
figure()  
biplot(Lambda, 'LineWidth', 2, 'MarkerSize', 20)
```



### User-Defined Rotation Function

Syntax for passing additional arguments to a user-defined rotation function:

```
[Lambda, Psi, T] = ...
    factoran(X, 2, 'rotate', @myrotation, 'userargs', 1, 'two');
```

## Definitions

### Factor Analysis Model

`factoran` computes the maximum likelihood estimate (MLE) of the factor loadings matrix  $\Lambda$  in the factor analysis model

$$x = \mu + \Lambda f + e$$

where  $x$  is a vector of observed variables,  $\mu$  is a constant vector of means,  $\Lambda$  is a constant  $d$ -by- $m$  matrix of factor loadings,  $f$  is a vector of independent, standardized common factors, and  $e$  is a vector of independent specific factors.  $x$ ,  $\mu$ , and  $e$  are of length  $d$ .  $f$  is of length  $m$ .

Alternatively, the factor analysis model can be specified as

$$\text{cov}(x) = \Lambda\Lambda^T + \Psi$$

where  $\Psi = \text{cov}(e)$  is a  $d$ -by- $d$  diagonal matrix of specific variances.

## Tips

### Observed Data Variables

The variables in the observed data matrix  $X$  must be linearly independent, i.e.,  $\text{cov}(X)$  must have full rank, for maximum likelihood estimation to succeed. `factoran` reduces both raw data and a covariance matrix to a correlation matrix before performing the fit.

`factoran` standardizes the observed data  $X$  to zero mean and unit variance before estimating the loadings `lambda`. This does not affect the model fit, because MLEs in this model are invariant to scale. However, `lambda` and `psi` are returned in terms of the standardized variables, i.e., `lambda*lambda'+diag(psi)` is an estimate of the correlation matrix of the original data  $X$  (although not after an oblique rotation). See

“Estimate and Plot Factor Loadings” on page 32-1665 and “User-Defined Rotation Function” on page 32-1670.

## Heywood Case

If elements of  $\psi$  are equal to the value of the 'delta' parameter (i.e., they are essentially zero), the fit is known as a Heywood case, and interpretation of the resulting estimates is problematic. In particular, there can be multiple local maxima of the likelihood, each with different estimates of the loadings and the specific variances. Heywood cases can indicate overfitting (i.e.,  $m$  is too large), but can also be the result of underfitting.

## Rotation of Factor Loadings and Scores

Unless you explicitly specify no rotation using the 'rotate' parameter, `factoran` rotates the estimated factor loadings,  $\lambda$ , and the factor scores,  $F$ . The output matrix  $T$  is used to rotate the loadings, i.e.,  $\lambda = \lambda_0 * T$ , where  $\lambda_0$  is the initial (unrotated) MLE of the loadings.  $T$  is an orthogonal matrix for orthogonal rotations, and the identity matrix for no rotation. The inverse of  $T$  is known as the primary axis rotation matrix, while  $T$  itself is related to the reference axis rotation matrix. For orthogonal rotations, the two are identical.

`factoran` computes factor scores that have been rotated by  $\text{inv}(T')$ , i.e.,  $F = F_0 * \text{inv}(T')$ , where  $F_0$  contains the unrotated predictions. The estimated covariance of  $F$  is  $\text{inv}(T' * T)$ , which, for orthogonal or no rotation, is the identity matrix. Rotation of factor loadings and scores is an attempt to create a more easily interpretable structure in the loadings matrix after maximum likelihood estimation.

## References

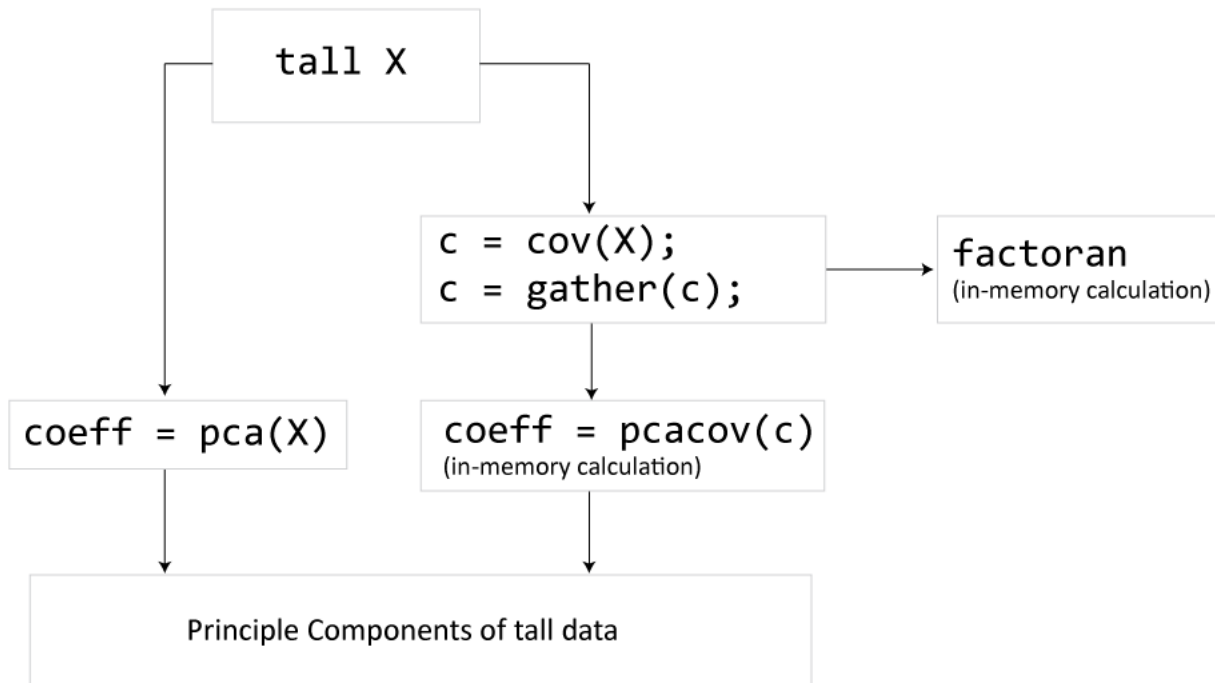
- [1] Harman, H. H. *Modern Factor Analysis*. 3rd Ed. Chicago: University of Chicago Press, 1976.
- [2] Jöreskog, K. G. “Some Contributions to Maximum Likelihood Factor Analysis.” *Psychometrika*. Vol. 32, Issue 4, 1967, pp. 443–482.
- [3] Lawley, D. N., and A. E. Maxwell. *Factor Analysis as a Statistical Method*. 2nd Ed. New York: American Elsevier Publishing Co., 1971.

## Extended Capabilities

### Tall Arrays

Calculate with arrays that have more rows than fit in memory.

`pcacov` and `factoran` do *not* work directly on tall arrays. Instead, use `C = gather(cov(X))` to compute the covariance matrix of a tall array. Then, you can use `pcacov` or `factoran` on the in-memory covariance matrix. Alternatively, you can use `pca` directly on a tall array.



For more information, see “Tall Arrays” (MATLAB).

### See Also

`biplot` | `pca` | `pcacov` | `procrustes` | `rotatefactors` | `statset`

**Introduced before R2006a**



## FBoot property

**Class:** TreeBagger

Fraction of in-bag observations

### Description

The `FBoot` property is the fraction of observations to be randomly selected with replacement for each bootstrap replica. The size of each replica is given by  $n * F_{\text{Boot}}$ , where  $n$  is the number of observations in the training set. The default value is 1.

## fcdf

*F* cumulative distribution function

## Syntax

```
p = fcdf(x, v1, v2)
p = fcdf(x, v1, v2, 'upper')
```

## Description

`p = fcdf(x, v1, v2)` computes the *F* cdf at each of the values in `x` using the corresponding numerator degrees of freedom `v1` and denominator degrees of freedom `v2`. `x`, `v1`, and `v2` can be vectors, matrices, or multidimensional arrays that are all the same size. A scalar input is expanded to a constant matrix with the same dimensions as the other inputs. `v1` and `v2` parameters must contain real positive values, and the values in `x` must lie on the interval `[0 Inf]`.

`p = fcdf(x, v1, v2, 'upper')` returns the complement of the *F* cdf at each value in `x`, using an algorithm that more accurately computes the extreme upper tail probabilities.

The *F* cdf is

$$p = F(x | v_1, v_2) = \int_0^x \frac{\Gamma\left[\frac{(v_1 + v_2)}{2}\right]}{\Gamma\left(\frac{v_1}{2}\right)\Gamma\left(\frac{v_2}{2}\right)} \left(\frac{v_1}{v_2}\right)^{\frac{v_1}{2}} \frac{t^{\frac{v_1-2}{2}}}{\left[1 + \left(\frac{v_1}{v_2}\right)t\right]^{\frac{v_1+v_2}{2}}} dt$$

The result, `p`, is the probability that a single observation from an *F* distribution with parameters `v1` and `v2` will fall in the interval `[0 x]`.

## Examples

## Compute F Distribution CDF

The following illustrates a useful mathematical identity for the  $F$  distribution.

```
nu1 = 1:5;  
nu2 = 6:10;  
x = 2:6;
```

```
F1 = fcdf(x, nu1, nu2)
```

```
F1 =
```

```
    0.7930    0.8854    0.9481    0.9788    0.9919
```

```
F2 = 1 - fcdf(1./x, nu2, nu1)
```

```
F2 =
```

```
    0.7930    0.8854    0.9481    0.9788    0.9919
```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

[cdf](#) | [finv](#) | [fpdf](#) | [frnd](#) | [fstat](#)

### Topics

“F Distribution” on page B-44

Introduced before R2006a

## feval

**Class:** CompactGeneralizedLinearModel

Evaluate generalized linear regression model prediction

## Syntax

```
ypred = feval mdl, Xnew1, Xnew2, ..., Xnewn
```

## Description

`ypred = feval(mdl, Xnew1, Xnew2, ..., Xnewn)` returns the predicted response of `mdl` to the input `[Xnew1, Xnew_2, ..., Xnewn]`.

## Input Arguments

**mdl**

Generalized linear model, specified as a full `GeneralizedLinearModel` object constructed using `fitglm` or `stepwiseglm`, or a compacted `CompactGeneralizedLinearModel` object constructed using `compact`.

**Xnew1, Xnew2, ..., Xnewn**

Predictor components. `Xnewi` can be one of:

- Scalar
- Vector
- Array

Each nonscalar component must have the same size (number of elements in each dimension).

If you pass just one `Xnew` array, `Xnew` can be a table, dataset array, or an array of doubles, where each column of the array represents one predictor.

## Output Arguments

### `ypred`

Predicted mean values at `Xnew`. `ypred` is the same size as each component of `Xnew`.

For binomial models, `feval` uses 1 as the `BinomialSize` parameter, so `ypred` is predicted probabilities.

For models with an offset, `feval` uses 0 as the offset value.

## Examples

### Predict Responses Using `feval`

Generate a generalized linear model, and plot its responses to a range of input data.

Generate artificial data for the model, Poisson random numbers with two underlying predictors `X(1)` and `X(2)`.

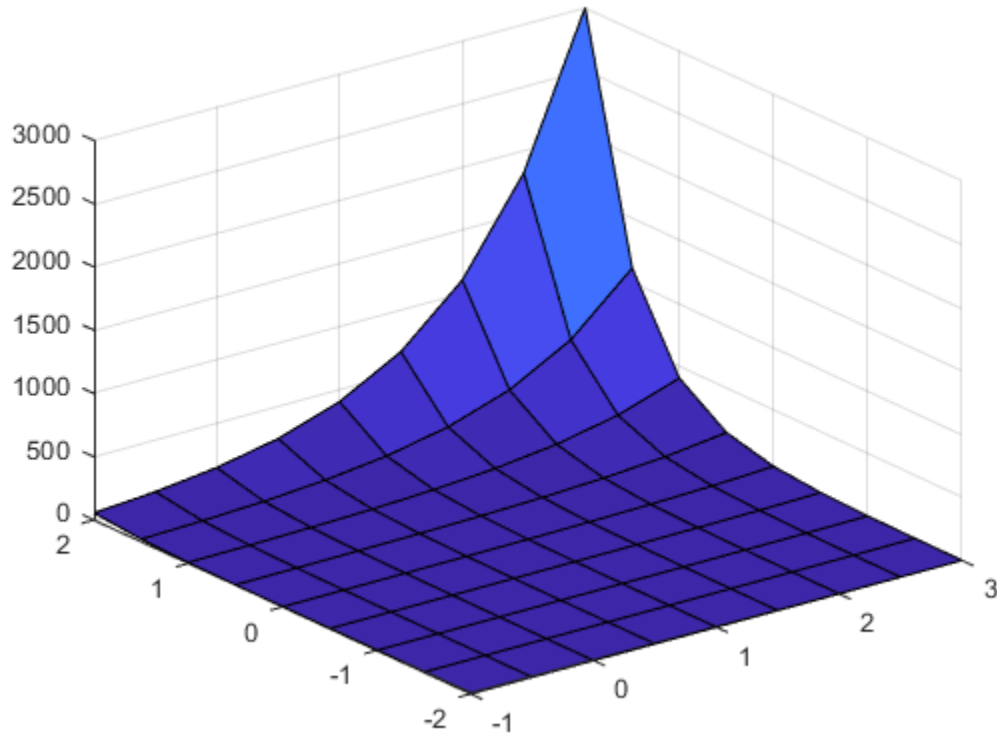
```
rng('default') % for reproducibility
rndvars = randn(100,2);
X = [2+rndvars(:,1), rndvars(:,2)];
mu = exp(1 + X*[1;2]);
y = poissrnd(mu);
```

Create a generalized linear regression model of Poisson data.

```
mdl = fitglm(X,y,'y ~ x1 + x2','distr','poisson');
```

Generate a range of values for `X(1)` and `X(2)`, and plot the model predictions at those values.

```
[Xtest1 Xtest2] = meshgrid(-1:.5:3,-2:.5:2);
Z = feval(mdl,Xtest1,Xtest2);
surf(Xtest1,Xtest2,Z)
```



- “feval” on page 12-36

## Tips

- `feval` allows you to easily evaluate predictions of a model when the model was fitted using a table or dataset array. `predict` requires a table or dataset array with the same predictor names, but you can use simple arrays of scalars with `feval`.

## Alternatives

`predict` gives the same predictions, but uses a single input array with one observation in each row, rather than one component in each input argument.

`random` predicts with added noise.

## See Also

`CompactGeneralizedLinearModel` | `GeneralizedLinearModel` | `predict` | `random`

## Topics

“feval” on page 12-36

“Generalized Linear Models” on page 12-12

## feval

**Class:** CompactLinearModel

Evaluate linear regression model prediction

## Syntax

```
ypred = feval mdl, Xnew1, Xnew2, . . . , Xnewn
```

## Description

`ypred = feval(mdl, Xnew1, Xnew2, . . . , Xnewn)` returns the predicted response of `mdl` to the new input predictors `[Xnew1, Xnew2, . . . , Xnewn]`.

## Input Arguments

**mdl** — Linear model object

LinearModel object | CompactLinearModel object

Linear model object, specified as a full `LinearModel` object constructed using `fitlm` or `stepwiselm`, or a compacted `CompactLinearModel` object constructed using `compact`.

**Xnew1, Xnew2, . . . , Xnewn** — New predictor values

scalar value | numeric vector

New predictor values, specified as a scalar value, numeric vector, or numeric array. You can specify multiple components for this argument. Each vector component must have the same number of elements in each dimension (in other words, each vector must be same size).

If you pass just one `Xnew` array, then `Xnew` can be a table, dataset array, or an array of doubles, where each column of the array represents one predictor.



## Output Arguments

### **ypred** — Predicted mean values

scalar value | numeric vector

Predicted mean values at  $X_{new1}, X_{new2}, \dots, X_{newn}$ , returned as a scalar value or numeric vector. `ypred` is the same size as each component of  $X_{new1}, X_{new2}, \dots, X_{newn}$ .

For models with an offset, `feval` uses 0 as the offset value.

## Examples

### **Plot Different Categorical Levels for Linear Model**

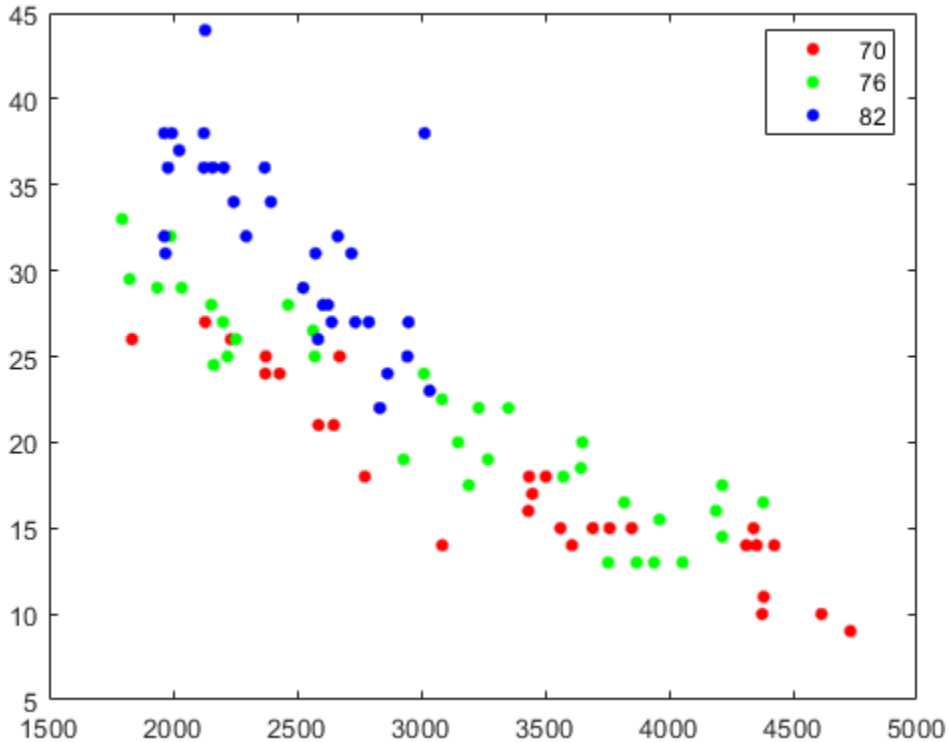
Fit a mileage model to the `carsmall` data, including the `Year` categorical predictor. Superimpose fitted curves on a scatter plot of the data.

Load the data and fit a model.

```
load carsmall
tbl = table(MPG,Weight);
tbl.Year = ordinal(Model_Year);
mdl = fitlm(tbl, 'MPG ~ Year + Weight^2');
```

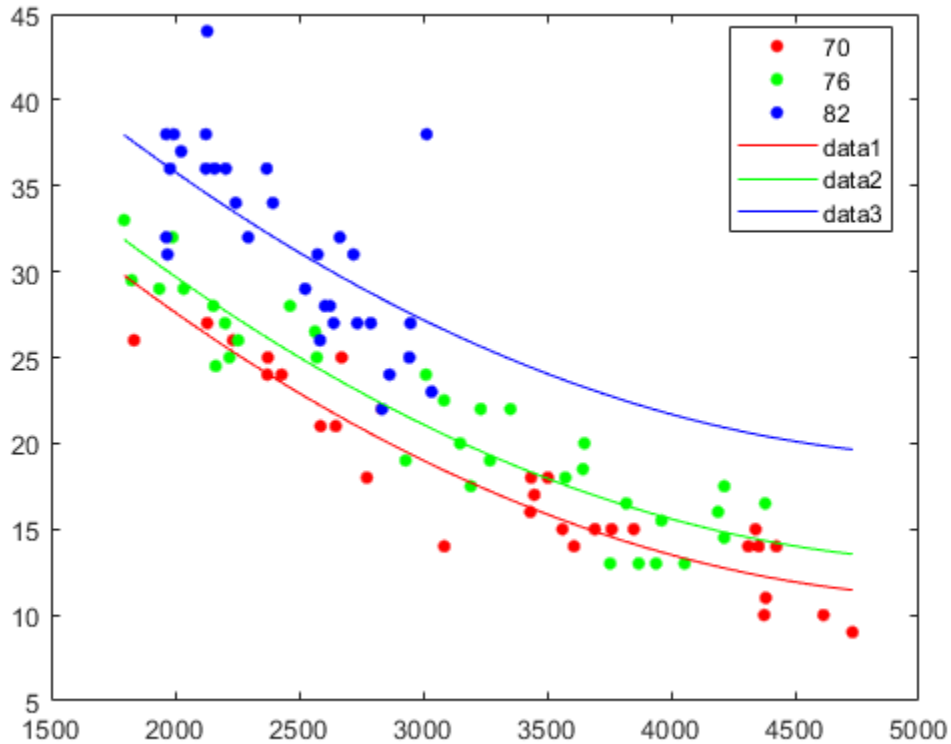
Create a scatter plot of the mileage versus weight.

```
gscatter(tbl.Weight, tbl.MPG, tbl.Year);
```



Use `feval` to plot curves of the model predictions for the various years and weights.

```
w = linspace(min(tbl.Weight),max(tbl.Weight))';  
line(w,feval(mdl,w,'70'),'Color','r')  
line(w,feval(mdl,w,'76'),'Color','g')  
line(w,feval(mdl,w,'82'),'Color','b')
```



- “feval” on page 11-39
- “Linear Regression Workflow” on page 11-43

## Alternatives

`predict` gives the same predictions, but uses a single input array with one observation in each row, rather than one component in each input argument. `predict` also gives confidence intervals on its predictions.

`random` predicts with added noise.

## See Also

`CompactLinearModel` | `LinearModel` | `predict` | `random`

## Topics

“feval” on page 11-39

“Linear Regression Workflow” on page 11-43

“Linear Regression” on page 11-11

# feval

**Class:** NonLinearModel

Evaluate nonlinear regression model prediction

## Syntax

```
ypred = feval mdl, Xnew1, Xnew2, ..., Xnewn
```

## Description

`ypred = feval(mdl, Xnew1, Xnew2, ..., Xnewn)` returns the predicted response of `mdl` to the input `[Xnew1, Xnew2, ..., Xnewn]`.

## Input Arguments

**mdl**

Nonlinear regression model, constructed by `fitnlm`.

**Xnew1, Xnew2, ..., Xnewn**

Predictor components. `Xnewi` can be one of:

- Scalar
- Vector
- Array

Each nonscalar component must have the same size (number of elements in each dimension).

If you pass just one `Xnew` array, `Xnew` can be a table, dataset array, or an array of doubles, where each column of the array represents one predictor.

## Output Arguments

### **ypred**

Predicted mean values at *Xnew*. *ypred* is the same size as each component of *Xnew*.

## Examples

### **Predict a Nonlinear Model from a Table**

Create a nonlinear model for auto mileage based on the `carbig` data. Predict the mileage of an average automobile.

Load the data and create a nonlinear model.

```
load carbig
tbl = table(Horsepower,Weight,MPG);
modelfun = @(b,x)b(1) + b(2)*x(:,1).^b(3) + ...
    b(4)*x(:,2).^b(5);
beta0 = [-50 500 -1 500 -1];
mdl = fitnlm(tbl,modelfun,beta0);
```

Find the predicted mileage of an average auto. The data contain some observations with NaN, so compute the mean using `nanmean`.

```
Xnew = nanmean([Horsepower Weight]);
MPGnew = feval(mdl,Xnew)
```

```
MPGnew =
    21.8073
```

- “Predict or Simulate Responses Using a Nonlinear Model” on page 13-10

## Alternatives

`predict` gives the same predictions, but uses a single input array with one observation in each row, rather than one component in each input argument. `predict` also gives confidence intervals on its predictions.

`random` predicts with added noise.

## See Also

`NonLinearModel` | `predict` | `random`

## Topics

“Predict or Simulate Responses Using a Nonlinear Model” on page 13-10

“Nonlinear Regression” on page 13-2

## ff2n

Two-level full factorial design

## Syntax

```
dFF2 = ff2n(n)
```

## Description

`dFF2 = ff2n(n)` gives factor settings `dFF2` for a two-level full factorial design with `n` factors. `dFF2` is  $m$ -by- $n$ , where  $m$  is the number of treatments in the full-factorial design. Each row of `dFF2` corresponds to a single treatment. Each column contains the settings for a single factor, with values of 0 and 1 for the two levels.

## Examples

```
dFF2 = ff2n(3)
dFF2 =
  0  0  0
  0  0  1
  0  1  0
  0  1  1
  1  0  0
  1  0  1
  1  1  0
  1  1  1
```

## See Also

`fullfact`

Introduced before R2006a



# fillprox

**Class:** TreeBagger

Proximity matrix for training data

## Syntax

```
B = fillprox(B)
B = fillprox(B, 'param1', val1, 'param2', val2, ...)
```

## Description

`B = fillprox(B)` computes a proximity matrix for the training data and stores it in the `Properties` field of `B`.

`B = fillprox(B, 'param1', val1, 'param2', val2, ...)` specifies optional parameter name/value pairs:

'Trees'	Either 'all' or a vector of indices of the trees in the ensemble to be used in computing the proximity matrix. Default is 'all'.
'NumPrint'	Number of training cycles (grown trees) after which <code>TreeBagger</code> displays a diagnostic message showing training progress. Default is no diagnostic messages.

## See Also

`CompactTreeBagger.outlierMeasure` | `CompactTreeBagger.proximity`

## findobj

**Class:** grandstream

Find objects matching specified conditions

### Syntax

```
hm = findobj(h, 'conditions')
```

### Description

The `findobj` method of the `handle` class follows the same syntax as the MATLAB `findobj` command, except that the first argument must be an array of handles to objects.

`hm = findobj(h, 'conditions')` searches the handle object array `h` and returns an array of handle objects matching the specified conditions. Only the public members of the objects of `h` are considered when evaluating the conditions.

### See Also

`findobj` | `grandstream`

# findprop

**Class:** grandstream

Find property of MATLAB handle object

## Syntax

```
p = findprop(h, 'propname')
```

## Description

`p = findprop(h, 'propname')` finds and returns the `meta.property` object associated with property name `propname` of scalar handle object `h`. `propname` must be a character vector. It can be the name of a property defined by the class of `h` or a dynamic property added to scalar object `h`.

If no property named `propname` exists for object `h`, an empty `meta.property` array is returned.

## See Also

`dynamicprops` | `findobj` | `meta.property` | `grandstream`

## finv

*F* inverse cumulative distribution function

## Syntax

`X = finv(P, V1, V2)`

## Description

`X = finv(P, V1, V2)` computes the inverse of the *F* cdf with numerator degrees of freedom `V1` and denominator degrees of freedom `V2` for the corresponding probabilities in `P`. `P`, `V1`, and `V2` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array with the same dimensions as the other inputs. `V1` and `V2` parameters must contain real positive values, and the values in `P` must lie on the interval `[0 1]`.

The *F* inverse function is defined in terms of the *F* cdf as

$$x = F^{-1}(p | v_1, v_2) = \{x : F(x | v_1, v_2) = p\}$$

where

$$p = F(x | v_1, v_2) = \int_0^x \frac{\Gamma\left[\frac{(v_1 + v_2)}{2}\right]}{\Gamma\left(\frac{v_1}{2}\right)\Gamma\left(\frac{v_2}{2}\right)} \left(\frac{v_1}{v_2}\right)^{\frac{v_1}{2}} \frac{t^{\frac{v_1-2}{2}}}{\left[1 + \left(\frac{v_1}{v_2}\right)t\right]^{\frac{v_1+v_2}{2}}} dt$$

## Examples

Find a value that should exceed 95% of the samples from an *F* distribution with 5 degrees of freedom in the numerator and 10 degrees of freedom in the denominator.

```
x = finv(0.95,5,10)
x =
    3.3258
```

You would observe values greater than 3.3258 only 5% of the time by chance.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`fcdf` | `fpdf` | `frnd` | `fstat` | `icdf`

### Topics

“F Distribution” on page B-44

Introduced before R2006a

## fishertest

Fisher's exact test

### Syntax

```
h = fishertest(x)
[h,p,stats] = fishertest(x)
[ ___ ] = fishertest(x,Name,Value)
```

### Description

`h = fishertest(x)` returns a test decision for Fisher's exact test of the null hypothesis that there are no nonrandom associations between the two categorical variables in `x`, against the alternative that there is a nonrandom association. The result `h` is 1 if the test rejects the null hypothesis at the 5% significance level, or 0 otherwise.

`[h,p,stats] = fishertest(x)` also returns the significance level `p` of the test and a structure `stats` containing additional test results, including the odds ratio and its asymptotic confidence interval.

`[ ___ ] = fishertest(x,Name,Value)` returns a test decision using additional options specified by one or more name-value pair arguments. For example, you can change the significance level of the test or conduct a one-sided test.

### Examples

#### Conduct Fisher's Exact Test

In a small survey, a researcher asked 17 individuals if they received a flu shot this year, and whether they caught the flu this winter. The results indicate that, of the nine people who did not receive a flu shot, three got the flu and six did not. Of the eight people who received a flu shot, one got the flu and seven did not.

Create a 2-by-2 contingency table containing the survey data. Row 1 contains data for the individuals who did not receive a flu shot, and row 2 contains data for the individuals who received a flu shot. Column 1 contains the number of individuals who got the flu, and column 2 contains the number of individuals who did not.

```
x = table([3;1],[6;7], 'VariableNames', {'Flu', 'NoFlu'}, 'RowNames', {'NoShot', 'Shot'})
```

```
x =
```

```
2x2 table
```

	Flu	NoFlu
NoShot	3	6
Shot	1	7

Use Fisher's exact test to determine if there is a nonrandom association between receiving a flu shot and getting the flu.

```
h = fishertest(x)
```

```
h =
```

```
logical
```

```
0
```

The returned test decision  $h = 0$  indicates that `fishertest` does not reject the null hypothesis of no nonrandom association between the categorical variables at the default 5% significance level. Therefore, based on the test results, individuals who do not get a flu shot do not have different odds of getting the flu than those who got the flu shot.

### Conduct a One-Sided Fisher's Exact Test

In a small survey, a researcher asked 17 individuals if they received a flu shot this year, and whether they caught the flu. The results indicate that, of the nine people who did not receive a flu shot, three got the flu and six did not. Of the eight people who received a flu shot, one got the flu and seven did not.

```
x = [3,6;1,7];
```

Use a right-tailed Fisher's exact test to determine if the odds of getting the flu is higher for individuals who did not receive a flu shot than for individuals who did. Conduct the test at the 1% significance level.

```
[h,p,stats] = fishertest(x,'Tail','right','Alpha',0.01)
```

```
h =
```

```
logical
```

```
0
```

```
p =
```

```
0.3353
```

```
stats =
```

```
struct with fields:
```

```
OddsRatio: 3.5000
```

```
ConfidenceInterval: [0.1289 95.0408]
```

The returned test decision `h = 0` indicates that `fishertest` does not reject the null hypothesis of no nonrandom association between the categorical variables at the 1% significance level. Since this is a right-tailed hypothesis test, the conclusion is that individuals who do not get a flu shot do not have greater odds of getting the flu than those who got the flu shot.

### Generate a Contingency Table Using `crosstab`

Load the hospital data.

```
load hospital
```



The `hospital` dataset array contains data on 100 hospital patients, including last name, gender, age, weight, smoking status, and systolic and diastolic blood pressure measurements.

To determine if smoking status is independent of gender, use `crosstab` to create a 2-by-2 contingency table of smokers and nonsmokers, grouped by gender.

```
[tbl,chi2,p,labels] = crosstab(hospital.Sex,hospital.Smoker)
```

```
tbl =
```

```
    40    13
    26    21
```

```
chi2 =
```

```
    4.5083
```

```
p =
```

```
    0.0337
```

```
labels =
```

```
    2x2 cell array
```

```
    {'Female'}    {'0'}
    {'Male'  }    {'1' }
```

The rows of the resulting contingency table `tbl` correspond to the patient's gender, with row 1 containing data for females and row 2 containing data for males. The columns correspond to the patient's smoking status, with column 1 containing data for nonsmokers and column 2 containing data for smokers. The returned result `chi2 = 4.5083` is the value of the chi-squared test statistic for a chi-squared test of independence. The returned value `p = 0.0337` is an approximate  $P$ -value based on the chi-squared distribution.

Use the contingency table generated by `crosstab` to perform Fisher's exact test on the data.

```
[h,p,stats] = fishertest(tbl)

h =

    logical

     1

p =

    0.0375

stats =

    struct with fields:

                OddsRatio: 2.4852
    ConfidenceInterval: [1.0624 5.8135]
```

The result `h = 1` indicates that `fishertest` rejects the null hypothesis of nonassociation between smoking status and gender at the 5% significance level. In other words, there is an association between gender and smoking status. The odds ratio indicates that the male patients have about 2.5 times greater odds of being smokers than the female patients.

The returned *P*-value of the test, `p = 0.0375`, is close to, but not exactly the same as, the result obtained by `crosstab`. This is because `fishertest` computes an exact *P*-value using the sample data, while `crosstab` uses a chi-squared approximation to compute the *P*-value.

## Input Arguments

### **x** — Contingency table

2-by-2 matrix of nonnegative integer values | 2-by-2 table of nonnegative integer values

Contingency table, specified as a 2-by-2 matrix or table containing nonnegative integer values. A contingency table contains the frequency distribution of the variables in the sample data. You can use `crosstab` to generate a contingency table from sample data.

Example: [4,0;0,4]

Data Types: single | double

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: 'Alpha', 0.01, 'Tail', 'right' specifies a right-tailed hypothesis test at the 1% significance level.

### Alpha — Significance level

0.05 (default) | scalar value in the range (0,1)

Significance level of the hypothesis test, specified as the comma-separated pair consisting of 'Alpha' and a scalar value in the range (0,1).

Example: 'Alpha', 0.01

Data Types: single | double

### Tail — Type of alternative hypothesis

'both' (default) | 'right' | 'left'

Type of alternative hypothesis, specified as the comma-separated pair consisting of 'Tail' and one of the following.

'both'

Two-tailed test. The alternative hypothesis is that there is a nonrandom association between the two variables in  $x$ , and the odds ratio is not equal to 1.

'right'

Right-tailed test. The alternative hypothesis is that the odds ratio is greater than 1.

'left'

Left-tailed test. The alternative hypothesis is that the odds ratio is less than 1.

Example: 'Tail', 'right'

## Output Arguments

### **h** — Hypothesis test result

1 | 0

Hypothesis test result, returned as a logical value.

- If `h` is 1, then `fishertest` rejects the null hypothesis at the Alpha significance level.
- If `h` is 0, then `fishertest` fails to reject the null hypothesis at the Alpha significance level.

### **p** — *p*-value

scalar value in the range [0,1]

*p*-value of the test, returned as a scalar value in the range [0,1]. *p* is the probability of observing a test statistic as extreme as, or more extreme than, the observed value under the null hypothesis. Small values of *p* cast doubt on the validity of the null hypothesis.

### **stats** — Test data

structure

Test data, returned as a structure with the following fields:

- `OddsRatio` — A measure of association between the two variables.
- `ConfidenceInterval` — Asymptotic confidence interval for the odds ratio. If any of the cell frequencies in `x` are 0, then `fishertest` does not compute a confidence interval and instead displays `[-Inf Inf]`.

## Definitions

### Fisher's Exact Test

Fisher's exact test is a nonparametric statistical test used to test the null hypothesis that no nonrandom associations exist between two categorical variables, against the alternative that there is a nonrandom association between the variables.

Fisher's exact test provides an alternative to the chi-squared test for small samples, or samples with very uneven marginal distributions. Unlike the chi-squared test, Fisher's

exact test does not depend on large-sample distribution assumptions, and instead calculates an exact  $p$ -value based on the sample data. Although Fisher's exact test is valid for samples of any size, it is not recommended for large samples because it is computationally intensive. If all of the frequency counts in the contingency table are greater than or equal to  $1e7$ , then `fishertest` errors. For contingency tables that contain large count values or are well-balanced, use `crosstab` or `chi2gof` instead.

`fishertest` accepts a 2-by-2 contingency table as input, and computes the  $p$ -value of the test as follows:

- 1 Calculate the sums for each row, column, and total number of observations in the contingency table.
- 2 Using a multivariate generalization of the hypergeometric probability function, calculate the conditional probability of observing the exact result in the contingency table if the null hypothesis were true, given its row and column sums. The conditional probability is

$$P_{cutoff} = \frac{(R_1!R_2!)(C_1!C_2!)}{N! \prod_{i,j} n_{ij}!},$$

where  $R_1$  and  $R_2$  are the row sums,  $C_1$  and  $C_2$  are the column sums,  $N$  is the total number of observations in the contingency table, and  $n_{ij}$  is the value in the  $i$ th row and  $j$ th column of the table.

- 3 Find all possible matrices of nonnegative integers consistent with the row and column sums. For each matrix, calculate the associated conditional probability using the equation for  $P_{cutoff}$ .
- 4 Use these values to calculate the  $p$ -value of the test, based on the alternative hypothesis of interest.
  - For a two-sided test, sum all of the conditional probabilities less than or equal to  $P_{cutoff}$  for the observed contingency table. This represents the probability of observing a result as extreme as, or more extreme than, the actual outcome if the null hypothesis were true. Small  $p$ -values cast doubt on the validity of the null hypothesis, in favor of the alternative hypothesis of association between the variables.
  - For a left-sided test, sum the conditional probabilities of all the matrices with a (1,1) cell frequency less than or equal to  $n_{11}$ .
  - For a right-sided test, sum the conditional probabilities of all the matrices with a (1,1) cell frequency greater than or equal to  $n_{11}$  in the observed contingency table.

The odds ratio is

$$OR = \frac{n_{11}n_{22}}{n_{21}n_{12}}.$$

The null hypothesis of conditional independence is equivalent to the hypothesis that the odds ratio equals 1. The left-sided alternative is equivalent to an odds ratio less than 1, and the right-sided alternative is equivalent to an odds ratio greater than 1.

The asymptotic  $100(1 - \alpha)\%$  confidence interval for the odds ratio is

$$CI = \left[ \exp\left(L - \Phi^{-1}\left(\frac{1 - \alpha}{2}\right)SE\right), \exp\left(L + \Phi^{-1}\left(\frac{1 - \alpha}{2}\right)SE\right) \right],$$

where  $L$  is the log odds ratio,  $\Phi^{-1}(\cdot)$  is the inverse of the normal inverse cumulative distribution function, and  $SE$  is the standard error for the log odds ratio. If the  $100(1 - \alpha)\%$  confidence interval does not contain the value 1, then the association is significant at the  $\alpha$  significance level. If any of the four cell frequencies are 0, then `fishertest` does not compute the confidence interval and instead displays `[-Inf Inf]`.

`fishertest` only accepts 2-by-2 contingency tables as input. To test the independence of categorical variables with more than two levels, use the chi-squared test provided by `crosstab`.

## See Also

`chi2gof` | `crosstab`

**Introduced in R2014b**

# ClassificationDiscriminant.fit

**Class:** ClassificationDiscriminant

Fit discriminant analysis classifier (to be removed)

---

**Note** `ClassificationDiscriminant.fit` will be removed in a future release. Use `fitcdiscr` instead.

---

## Syntax

```
obj = ClassificationDiscriminant.fit(x, y)
obj = ClassificationDiscriminant.fit(x, y, Name, Value)
```

## Description

`obj = ClassificationDiscriminant.fit(x, y)` returns a discriminant analysis classifier based on the input variables (also known as predictors, features, or attributes) `x` and output (response) `y`.

`obj = ClassificationDiscriminant.fit(x, y, Name, Value)` fits a classifier with additional options specified by one or more `Name, Value` pair arguments. If you use one of the following five options, `obj` is of class `ClassificationPartitionedModel`: `'CrossVal'`, `'KFold'`, `'Holdout'`, `'Leaveout'`, or `'CVPartition'`. Otherwise, `obj` is of class `ClassificationDiscriminant`.

## Input Arguments

**x** — Predictor values

matrix of numeric values

Predictor values, specified as a matrix of numeric values. Each column of `x` represents one variable, and each row represents one observation.

`ClassificationDiscriminant.fit` considers NaN values in `x` as missing values. `ClassificationDiscriminant.fit` does not use observations with missing values for `x` in the fit.

Example:

Data Types: `single` | `double`

### **y** — Classification values

numeric vector | categorical vector | logical vector | character array | cell array of character vectors

Classification values, specified as a numeric vector, categorical vector (nominal or ordinal), logical vector, character array, or cell array of character vectors. Each row of `y` represents the classification of the corresponding row of `x`.

`ClassificationDiscriminant.fit` considers NaN values in `y` to be missing values. `ClassificationDiscriminant.fit` does not use observations with missing values for `y` in the fit.

Data Types: `single` | `double` | `logical` | `char` | `cell`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### **ClassNames** — Class names

array

Class names, specified as the comma-separated pair consisting of 'ClassNames' and an array. Use the data type that exists in `y`. The default is the class names that exist in `y`. Use `ClassNames` to order the classes or to select a subset of classes for training.

Data Types: `single` | `double` | `logical` | `char`

### **Cost** — Cost of misclassification

square matrix | structure

Cost of misclassification, specified as the comma-separated pair consisting of 'Cost' and a square matrix, where `Cost(i, j)` is the cost of classifying a point into class `j` if its true



class is  $i$ . Alternatively, `Cost` can be a structure `S` having two fields: `S.ClassNames` containing the group names as a variable of the same type as `y`, and `S.ClassificationCosts` containing the cost matrix.

The default is  $\text{Cost}(i, j) = 1$  if  $i \neq j$ , and  $\text{Cost}(i, j) = 0$  if  $i = j$ .

Data Types: `single` | `double` | `struct`

### **CrossVal** — Cross-validation flag

`'off'` (default) | `'on'`

Cross-validation flag, specified as the comma-separated pair consisting of `'Crossval'` and `'on'` or `'off'`.

If you specify `'on'`, then the software implements 10-fold cross-validation.

To override this cross-validation setting, use one of these name-value pair arguments: `CVPartition`, `Holdout`, `KFold`, or `Leaveout`. To create a cross-validated model, you can use one cross-validation name-value pair argument at a time only.

Alternatively, cross validate later by passing `Mdl` to `crossval`.

Example: `'CrossVal', 'on'`

### **CVPartition** — Cross-validated model partition

`cvpartition` object

Cross-validated model partition, specified as the comma-separated pair consisting of `'CVPartition'` and an object created using `cvpartition`.

To create a cross-validated model, you can use one of these four name-value pair arguments only: `CVPartition`, `Holdout`, `KFold`, or `Leaveout`.

### **Delta** — Linear coefficient threshold

0 (default) | nonnegative scalar value

Linear coefficient threshold, specified as the comma-separated pair consisting of `'Delta'` and a nonnegative scalar value. If a coefficient of `Mdl` has magnitude smaller than `Delta`, `Mdl` sets this coefficient to 0, and you can eliminate the corresponding predictor from the model. Set `Delta` to a higher value to eliminate more predictors.

`Delta` must be 0 for quadratic discriminant models.

Data Types: `single` | `double`

**DiscrimType** — Discriminant type

'linear' (default) | 'quadratic' | 'diaglinear' | 'diagquadratic' |  
'pseudolinear' | 'pseudoquadratic'

Discriminant type, specified as the comma-separated pair consisting of 'DiscrimType' and a character vector in this table.

Value	Description	Predictor Covariance Treatment
'linear'	Regularized linear discriminant analysis (LDA)	<ul style="list-style-type: none"> <li>All classes have the same covariance matrix.</li> <li><math>\hat{\Sigma}_\gamma = (1 - \gamma)\hat{\Sigma} + \gamma \text{diag}(\hat{\Sigma})</math>.</li> </ul> <p><math>\hat{\Sigma}</math> is the empirical, pooled covariance matrix and <math>\gamma</math> is the amount of regularization.</p>
'diaglinear'	LDA	All classes have the same, diagonal covariance matrix.
'pseudolinear'	LDA	All classes have the same covariance matrix. The software inverts the covariance matrix using the pseudo inverse.
'quadratic'	Quadratic discriminant analysis (QDA)	The covariance matrices can vary among classes.
'diagquadratic'	QDA	The covariance matrices are diagonal and can vary among classes.
'pseudoquadratic'	QDA	The covariance matrices can vary among classes. The software inverts the covariance matrix using the pseudo inverse.

---

**Note** To use regularization, you must specify 'linear'. To specify the amount of regularization, use the Gamma name-value pair argument.

---

Example: 'DiscrimType', 'quadratic'

### **FillCoeffs — Coeffs property flag**

'on' | 'off'

Coeffs property flag, specified as the comma-separated pair consisting of 'FillCoeffs' and 'on' or 'off'. Setting the flag to 'on' populates the Coeffs property in the classifier object. This can be computationally intensive, especially when cross validating. The default is 'on', unless you specify a cross validation name-value pair, in which case the flag is set to 'off' by default.

Example: 'FillCoeffs', 'off'

### **Gamma — Amount of regularization**

scalar value in the interval [0,1]

Amount of regularization to apply when estimating the covariance matrix of the predictors, specified as the comma-separated pair consisting of 'Gamma' and a scalar value in the interval [0,1]. Gamma provides finer control over the covariance matrix structure than DiscrimType.

- If you specify 0, then the software does not use regularization to adjust the covariance matrix. That is, the software estimates and uses the unrestricted, empirical covariance matrix.
  - For linear discriminant analysis, if the empirical covariance matrix is singular, then the software automatically applies the minimal regularization required to invert the covariance matrix. You can display the chosen regularization amount by entering `Mdl.Gamma` at the command line.
  - For quadratic discriminant analysis, if at least one class has an empirical covariance matrix that is singular, then the software throws an error.
- If you specify a value in the interval (0,1), then you must implement linear discriminant analysis, otherwise the software throws an error. Consequently, the software sets `DiscrimType` to 'linear'.
- If you specify 1, then the software uses maximum regularization for covariance matrix estimation. That is, the software restricts the covariance matrix to be

diagonal. Alternatively, you can set `DiscrimType` to `'diagLinear'` or `'diagQuadratic'` for diagonal covariance matrices.

Example: `'Gamma', 1`

Data Types: `single` | `double`

### **Holdout** — Fraction of data for holdout validation

scalar value in the range (0,1)

Fraction of data used for holdout validation, specified as the comma-separated pair consisting of `'Holdout'` and a scalar value in the range (0,1). If you specify `'Holdout', p`, then the software:

- 1 Randomly reserves  $p \times 100\%$  of the data as validation data, and trains the model using the rest of the data
- 2 Stores the compact, trained model in the `Trained` property of the cross-validated model.

To create a cross-validated model, you can use one of these four name-value pair arguments only: `CVPartition`, `Holdout`, `KFold`, or `Leaveout`.

Example: `'Holdout', 0.1`

Data Types: `double` | `single`

### **KFold** — Number of folds

10 (default) | positive integer value greater than 1

Number of folds to use in a cross-validated classifier, specified as the comma-separated pair consisting of `'KFold'` and a positive integer value greater than 1. If you specify, e.g., `'KFold', k`, then the software:

- 1 Randomly partitions the data into  $k$  sets
- 2 For each set, reserves the set as validation data, and trains the model using the other  $k - 1$  sets
- 3 Stores the  $k$  compact, trained models in the cells of a  $k$ -by-1 cell vector in the `Trained` property of the cross-validated model.

To create a cross-validated model, you can use one of these four options only: `CVPartition`, `Holdout`, `KFold`, or `Leaveout`.

Example: `'KFold', 8`

Data Types: `single | double`

### **Leaveout** — Leave-one-out cross-validation flag

`'off'` (default) | `'on'`

Leave-one-out cross-validation flag, specified as the comma-separated pair consisting of `'Leaveout'` and `'on'` or `'off'`. If you specify `'Leaveout', 'on'`, then, for each of the  $n$  observations, where  $n$  is `size(Mdl.X, 1)`, the software:

- 1 Reserves the observation as validation data, and trains the model using the other  $n - 1$  observations
- 2 Stores the  $n$  compact, trained models in the cells of a  $n$ -by-1 cell vector in the `Trained` property of the cross-validated model.

To create a cross-validated model, you can use one of these four options only:

`'CVPartition', 'Holdout', 'KFold', or 'Leaveout'`.

Example: `'Leaveout', 'on'`

Data Types: `char`

### **PredictorNames** — Predictor variable names

`{'X1', 'X2', ...}` (default) | cell array of character vectors

Predictor variable names, specified as the comma-separated pair consisting of `'PredictorNames'` and a cell array of character vectors containing the names for the predictor variables, in the order in which they appear in  $X$ .

Data Types: `cell`

### **Prior** — Prior probabilities

`'empirical'` (default) | `'uniform'` | vector of scalar values | structure

Prior probabilities for each class, specified as the comma-separated pair consisting of `'Prior'` and one of the following.

- `'empirical'` determines class probabilities from class frequencies in  $y$ . If you pass observation weights, they are used to compute the class probabilities.
- `'uniform'` sets all class probabilities equal.

- A vector containing one scalar value for each class.
- A structure *S* with two fields:
  - *S.ClassNames* containing the class names as a variable of the same type as *y*
  - *S.ClassProbs* containing a vector of corresponding probabilities

Example: 'Prior', 'uniform'

Data Types: single | double | char | struct

### **ResponseName** — Response variable name

'Y' (default) | character vector

Response variable name, specified as the comma-separated pair consisting of 'ResponseName' and a character vector containing the name of the response variable *y*.

Example: 'ResponseName', 'Response'

Data Types: char

### **SaveMemory** — Flag to save covariance matrix

'off' (default) | 'on'

Flag to save covariance matrix, specified as the comma-separated pair consisting of 'SaveMemory' and either 'on' or 'off'. If you specify 'on', then `fitcdiscr` does not store the full covariance matrix, but instead stores enough information to compute the matrix. The `predict` method computes the full covariance matrix for prediction, and does not store the matrix. If you specify 'off', then `fitcdiscr` computes and stores the full covariance matrix in `Mdl`.

Specify `SaveMemory` as 'on' when the input matrix contains thousands of predictors.

Example: 'SaveMemory', 'on'

### **ScoreTransform** — Score transformation

'none' (default) | character vector | function handle

Score transformation, specified as the comma-separated pair consisting of 'ScoreTransform' and either a character vector or a function handle.

This table summarizes the available character vectors.

Value	Description
'doublelogit'	$1/(1 + e^{-2x})$
'invlogit'	$\log(x / (1-x))$
'ismax'	Set the score for the class with the largest score to 1, and set the scores for all other classes to 0.
'logit'	$1/(1 + e^{-x})$
'none' or 'identity'	$x$ (no transformation)
'sign'	-1 for $x < 0$ 0 for $x = 0$ 1 for $x > 0$
'symmetric'	$2x - 1$
'symmetricismax'	Set the score for the class with the largest score to 1, and set the scores for all other classes to -1.
'symmetriclogit'	$2/(1 + e^{-x}) - 1$

For a MATLAB function or a function you define, use its function handle. The function handle must accept a matrix (the original scores) and return a matrix of the same size (the transformed scores).

Example: 'ScoreTransform', 'logit'

Data Types: char | function\_handle

### Weights — Observation weights

ones(size(X,1),1) (default) | vector of scalar values

Observation weights, specified as the comma-separated pair consisting of 'Weights' and a vector of scalar values. The length of Weights is the number of rows in X. fitcdiscr normalizes the weights to sum to 1.

Data Types: single | double

## Output Arguments

### obj — Discriminant analysis classifier

classifier object

Discriminant analysis classifier, returned as a classifier object.

Note that using the 'CrossVal', 'KFold', 'Holdout', 'Leaveout', or 'CVPartition' options results in a tree of class `ClassificationPartitionedModel`. You cannot use a partitioned tree for prediction, so this kind of tree does not have a `predict` method.

Otherwise, `obj` is of class `ClassificationDiscriminant`, and you can use the `predict` method to predict the response of new data.

## Examples

### Construct a Discriminant Analysis Classifier

Load the sample data.

```
load fisheriris
```

Construct a discriminant analysis classifier using the sample data.

```
obj = ClassificationDiscriminant.fit(meas, species)
```

```
obj =
```

```
ClassificationDiscriminant
    ResponseName: 'Y'
  CategoricalPredictors: []
        ClassNames: {'setosa' 'versicolor' 'virginica'}
    ScoreTransform: 'none'
  NumObservations: 150
        DiscrimType: 'linear'
                Mu: [3x4 double]
                Coeffs: [3x3 struct]
```



## Definitions

### Discriminant Classification

The model for discriminant analysis is:

- Each class ( $Y$ ) generates data ( $X$ ) using a multivariate normal distribution. That is, the model assumes  $X$  has a Gaussian mixture distribution (`gmdistribution`).
- For linear discriminant analysis, the model has the same covariance matrix for each class, only the means vary.
- For quadratic discriminant analysis, both means and covariances of each class vary.

`predict` classifies so as to minimize the expected classification cost:

$$\hat{y} = \arg \min_{y=1, \dots, K} \sum_{k=1}^K \hat{P}(k | x) C(y | k),$$

where

- $\hat{y}$  is the predicted classification.
- $K$  is the number of classes.
- $\hat{P}(k | x)$  is the posterior probability on page 20-8 of class  $k$  for observation  $x$ .
- $C(y | k)$  is the cost on page 20-9 of classifying an observation as  $y$  when its true class is  $k$ .

For details, see “Prediction Using Discriminant Analysis Models” on page 20-8.

## Alternatives

The `classify` function also performs discriminant analysis. `classify` is usually more awkward to use:

- `classify` requires you to fit the classifier every time you make a new prediction.
- `classify` does not perform cross validation.
- `classify` requires you to fit the classifier when changing prior probabilities.

## See Also

`ClassificationDiscriminant`

## Topics

“Discriminant Analysis Classification” on page 20-2

# ClassificationKNN.fit

**Class:** ClassificationKNN

Fit  $k$ -nearest neighbor classifier (to be removed)

---

**Note** ClassificationKNN.fit will be removed in a future release. Use fitcknn instead.

---

## Syntax

```
mdl = ClassificationKNN.fit(X,y)
mdl = ClassificationKNN.fit(X,y,Name,Value)
```

## Description

mdl = ClassificationKNN.fit(X,y) returns a classification model based on the input variables (also known as predictors, features, or attributes) X and output (response) y.

mdl = ClassificationKNN.fit(X,y,Name,Value) fits a model with additional options specified by one or more Name,Value pair arguments.

If you use one of these options, mdl is of class ClassificationPartitionedModel: 'CrossVal', 'KFold', 'Holdout', 'Leaveout', or 'CVPartition'. Otherwise, mdl is of class ClassificationKNN.

## Input Arguments

### **x** — Predictor data

numeric matrix

Predictor data, specified as a numeric matrix. Each column of X represents one variable, and each row represents one observation.

The length of  $Y$  and the number of observations in  $X$  must be equal.

To specify the names of the predictors in the order of their appearance in  $X$ , use the `PredictorNames` name-value pair argument.

Data Types: `single` | `double`

### **$y$ — Classification values**

numeric vector | categorical vector | logical vector | character array | cell array of character vectors

Classification values, specified as a numeric vector, categorical vector, logical vector, character array, or cell array of character vectors, with the same number of rows as  $X$ . Each row of  $Y$  represents the classification of the corresponding row of  $X$ .

Data Types: `single` | `double` | `cell` | `logical` | `char`

## **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### **`BreakTies` — Tie-breaking algorithm**

'smallest' (default) | 'nearest' | 'random'

Tie-breaking algorithm used by the `predict` method if multiple classes have the same smallest cost, specified as the comma-separated pair consisting of '`BreakTies`' and one of the following:

- '`smallest`' — Use the smallest index among tied groups.
- '`nearest`' — Use the class with the nearest neighbor among tied groups.
- '`random`' — Use a random tiebreaker among tied groups.

By default, ties occur when multiple classes have the same number of nearest points among the  $K$  nearest neighbors.

Example: '`BreakTies`', '`nearest`'

**BucketSize — Maximum data points in node**

50 (default) | positive integer value

Maximum number of data points in the leaf node of the *kd*-tree, specified as the comma-separated pair consisting of 'BucketSize' and a positive integer value. This argument is meaningful only when NSMethod is 'kdtree'.

Example: 'BucketSize',40

Data Types: single | double

**CategoricalPredictors — Categorical predictor flag**

[] (default) | 'all'

Categorical predictor flag, specified as the comma-separated pair consisting of 'CategoricalPredictors' and one of the following:

- 'all' — All predictors are categorical.
- [] — No predictors are categorical.

When you set CategoricalPredictors to 'all', the default Distance is 'hamming'.

Example: 'CategoricalPredictors','all'

Data Types: char

**ClassNames — Class names**

numeric vector | categorical vector | logical vector | character array | cell array of character vectors

Class names, specified as the comma-separated pair consisting of 'ClassNames' and an array representing the class names. Use the same data type as the values that exist in *y*.

Use ClassNames to order the classes or to select a subset of classes for training. The default is the class names in *y*.

Data Types: single | double | char | logical | cell

**Cost — Cost of misclassification**

square matrix | structure

Cost of misclassification of a point, specified as the comma-separated pair consisting of 'Cost' and one of the following:

- Square matrix, where  $\text{Cost}(i, j)$  is the cost of classifying a point into class  $j$  if its true class is  $i$  (i.e., the rows correspond to the true class and the columns correspond to the predicted class). To specify the class order for the corresponding rows and columns of `Cost`, additionally specify the `ClassNames` name-value pair argument.
- Structure `S` having two fields: `S.ClassNames` containing the group names as a variable of the same type as `Y`, and `S.ClassificationCosts` containing the cost matrix.

The default is  $\text{Cost}(i, j) = 1$  if  $i \neq j$ , and  $\text{Cost}(i, j) = 0$  if  $i = j$ .

Data Types: `single` | `double` | `struct`

### **Cov** — Covariance matrix

`nancov(X)` (default) | positive definite matrix of scalar values

Covariance matrix, specified as the comma-separated pair consisting of `'Cov'` and a positive definite matrix of scalar values representing the covariance matrix when computing the Mahalanobis distance. This argument is only valid when `'Distance'` is `'mahalanobis'`.

You cannot simultaneously specify `'Standardize'` and either of `'Scale'` or `'Cov'`.

Data Types: `single` | `double`

### **CrossVal** — Cross-validation flag

`'off'` (default) | `'on'`

Cross-validation flag, specified as the comma-separated pair consisting of `'CrossVal'` and either `'on'` or `'off'`. If `'on'`, `fitcknn` creates a cross-validated model with 10 folds. Use the `'Kfold'`, `'Holdout'`, `'Leaveout'`, or `'CVPartition'` parameters to override this cross-validation setting. You can only use one cross-validation name-value pair argument at a time to create a cross-validated model.

Alternatively, cross validate `mdl` later using the `crossval` method.

Example: `'Crossval', 'on'`

### **CVPartition** — Cross-validated model partition

`cvpartition` object

Cross-validated model partition, specified as the comma-separated pair consisting of `'CVPartition'` and an object created using `cvpartition`. You can only use one of

these four options at a time to create a cross-validated model: 'Kfold', 'Holdout', 'Leaveout', or 'CVPartition'.

### Distance — Distance metric

character vector | function handle

Distance metric, specified as the comma-separated pair consisting of 'Distance' and a valid distance metric name or function handle. The allowable distance metric names depend on your choice of a neighbor-searcher method (see NSMethod).

NSMethod	Distance Metric Names
exhaustive	Any distance metric of ExhaustiveSearcher
kdtree	'cityblock', 'chebychev', 'euclidean', or 'minkowski'

This table includes valid distance metrics of ExhaustiveSearcher.

Distance Metric Names	Description
'cityblock'	City block distance.
'chebychev'	Chebychev distance (maximum coordinate difference).
'correlation'	One minus the sample linear correlation between observations (treated as sequences of values).
'cosine'	One minus the cosine of the included angle between observations (treated as vectors).
'euclidean'	Euclidean distance.
'hamming'	Hamming distance, percentage of coordinates that differ.
'jaccard'	One minus the Jaccard coefficient, the percentage of nonzero coordinates that differ.
'mahalanobis'	Mahalanobis distance, computed using a positive definite covariance matrix $C$ . The default value of $C$ is the sample covariance matrix of $X$ , as computed by <code>nancov(X)</code> . To specify a different value for $C$ , use the 'Cov' name-value pair argument.
'minkowski'	Minkowski distance. The default exponent is 2. To specify a different exponent, use the 'Exponent' name-value pair argument.

Distance Metric Names	Description
'seuclidean'	Standardized Euclidean distance. Each coordinate difference between $X$ and a query point is scaled, meaning divided by a scale value $S$ . The default value of $S$ is the standard deviation computed from $X$ , $S = \text{nanstd}(X)$ . To specify another value for $S$ , use the <code>Scale</code> name-value pair argument.
'spearman'	One minus the sample Spearman's rank correlation between observations (treated as sequences of values).
@ <i>distfun</i>	Distance function handle. <i>distfun</i> has the form <pre>function D2 = DISTFUN(ZI,ZJ) % calculation of distance ...</pre> <p>where</p> <ul style="list-style-type: none"> <li>• <math>ZI</math> is a 1-by-<math>N</math> vector containing one row of <math>X</math> or <math>Y</math>.</li> <li>• <math>ZJ</math> is an <math>M2</math>-by-<math>N</math> matrix containing multiple rows of <math>X</math> or <math>Y</math>.</li> <li>• <math>D2</math> is an <math>M2</math>-by-1 vector of distances, and <math>D2(k)</math> is the distance between observations <math>ZI</math> and <math>ZJ(k, :)</math>.</li> </ul>

If you specify `CategoricalPredictors` as 'all', then the default distance metric is 'hamming'. Otherwise, the default distance metric is 'euclidean'.

For definitions, see “Distance Metrics” on page 18-16.

Example: 'Distance', 'minkowski'

Data Types: `function_handle` | `char`

### DistanceWeight — Distance weighting function

'equal' (default) | 'inverse' | 'squaredinverse' | function handle

Distance weighting function, specified as the comma-separated pair consisting of 'DistanceWeight' and either a function handle or one of the values in this table.

Value	Description
'equal'	No weighting



Value	Description
'inverse'	Weight is 1/distance
'squaredinverse'	Weight is 1/distance <sup>2</sup>
@fcn	fcn is a function that accepts a matrix of nonnegative distances, and returns a matrix the same size containing nonnegative distance weights. For example, 'squaredinverse' is equivalent to @ (d) d.^(-2).

Example: 'DistanceWeight', 'inverse'

Data Types: function\_handle | char

#### **Exponent — Minkowski distance exponent**

2 (default) | positive scalar value

Minkowski distance exponent, specified as the comma-separated pair consisting of 'Exponent' and a positive scalar value. This argument is only valid when 'Distance' is 'minkowski'.

Example: 'Exponent', 3

Data Types: single | double

#### **Holdout — Fraction of data for holdout validation**

0 (default) | scalar value in the range [0,1]

Fraction of data used for holdout validation, specified as the comma-separated pair consisting of 'Holdout' and a scalar value in the range [0,1]. Holdout validation tests the specified fraction of the data, and uses the remaining data for training.

If you use Holdout, you cannot use any of the 'CVPartition', 'KFold', or 'Leaveout' name-value pair arguments.

Example: 'Holdout', 0.1

Data Types: single | double

#### **IncludeTies — Tie inclusion flag**

false (default) | true

Tie inclusion flag, specified as the comma-separated pair consisting of 'IncludeTies' and a logical value indicating whether predict includes all the neighbors whose

distance values are equal to the  $K$ th smallest distance. If `IncludeTies` is `true`, `predict` includes all these neighbors. Otherwise, `predict` uses exactly  $K$  neighbors.

Example: `'IncludeTies', true`

Data Types: `logical`

### **KFold** — Number of folds

10 (default) | positive integer value

Number of folds to use in a cross-validated model, specified as the comma-separated pair consisting of `'KFold'` and a positive integer value.

If you use `'KFold'`, you cannot use any of the `'CVPartition'`, `'Holdout'`, or `'Leaveout'` name-value pair arguments.

Example: `'KFold', 8`

Data Types: `single` | `double`

### **Leaveout** — Leave-one-out cross-validation flag

'off' (default) | 'on'

Leave-one-out cross-validation flag, specified as the comma-separated pair consisting of `'Leaveout'` and either `'on'` or `'off'`. Specify `'on'` to use leave-one-out cross validation.

If you use `'Leaveout'`, you cannot use any of the `'CVPartition'`, `'Holdout'`, or `'KFold'` name-value pair arguments.

Example: `'Leaveout', 'on'`

### **NSMethod** — Nearest neighbor search method

'kdtree' | 'exhaustive'

Nearest neighbor search method, specified as the comma-separated pair consisting of `'NSMethod'` and `'kdtree'` or `'exhaustive'`.

- `'kdtree'` — Create and use a *kd*-tree to find nearest neighbors. `'kdtree'` is valid when the distance metric is one of the following:
  - `'euclidean'`
  - `'cityblock'`

- 'minkowski'
- 'chebychev'
- 'exhaustive' — Use the exhaustive search algorithm. The distance values from all points in X to each point in Y are computed to find nearest neighbors.

The default is 'kdtree' when X has 10 or fewer columns, X is not sparse, and the distance metric is a 'kdtree' type; otherwise, 'exhaustive'.

Example: 'NSMethod', 'exhaustive'

### **NumNeighbors** — Number of nearest neighbors to find

1 (default) | positive integer value

Number of nearest neighbors in X to find for classifying each point when predicting, specified as the comma-separated pair consisting of 'NumNeighbors' and a positive integer value.

Example: 'NumNeighbors', 3

Data Types: single | double

### **PredictorNames** — Predictor variable names

{ 'x1', 'x2', ... } (default) | cell array of character vectors

Predictor variable names, specified as the comma-separated pair consisting of 'PredictorNames' and a cell array of character vectors containing the names for the predictor variables, in the order in which they appear in X.

If you specify the predictors as a table (tbl), then PredictorNames must be a subset of the variable names in tbl. In this case, the software uses only the variables in PredictorNames to train the model. If you use formula to specify the model, then you cannot use the PredictorNames name-value pair.

Example: 'PredictorNames', {'PedalWidth', 'PedalLength'}

Data Types: cell

### **Prior** — Prior probabilities

'empirical' (default) | 'uniform' | vector of scalar values | structure

Prior probabilities for each class, specified as the comma-separated pair consisting of 'Prior' and a value in this table.

Value	Description
'empirical'	The class prior probabilities are the class relative frequencies in $Y$ .
'uniform'	All class prior probabilities are equal to $1/K$ , where $K$ is the number of classes.
numeric vector	Each element is a class prior probability. Order the elements according to <code>Mdl.ClassNames</code> or specify the order using the <code>ClassNames</code> name-value pair argument. The software normalizes the elements such that they sum to 1.
structure	A structure $S$ with two fields: <ul style="list-style-type: none"> <li><code>S.ClassNames</code> contains the class names as a variable of the same type as <math>Y</math>.</li> <li><code>S.ClassProbs</code> contains a vector of corresponding prior probabilities. The software normalizes the elements such that they sum to 1.</li> </ul>

If you set values for both `Weights` and `Prior`, the weights are renormalized to add up to the value of the prior probability in the respective class.

Example: `'Prior', 'uniform'`

Data Types: `single` | `double` | `struct`

#### **ResponseName** — Response variable name

`'Y'` (default) | character vector

Response variable name, specified as the comma-separated pair consisting of `'ResponseName'` and a character vector containing the name of the response variable  $Y$ .

This name-value pair is not valid when using the `ResponseVarName` or `formula` input arguments.

Example: `'ResponseName', 'Response'`

Data Types: `char`

**Scale — Distance scale**

`nanstd(X)` (default) | vector of nonnegative scalar values

Distance scale, specified as the comma-separated pair consisting of 'Scale' and a vector containing nonnegative scalar values with length equal to the number of columns in X. Each coordinate difference between X and a query point is scaled by the corresponding element of Scale. This argument is only valid when 'Distance' is 'seuclidean'.

You cannot simultaneously specify 'Standardize' and either of 'Scale' or 'Cov'.

Data Types: `single` | `double`

**Weights — Observation weights**

`ones(size(X,1),1)` (default) | vector of scalar values

Observation weights, specified as the comma-separated pair consisting of 'Weights' and a vector of scalar values. The software weighs the observations in each row of X or `tbl` with the corresponding value in Weights. The size of Weights must equal the number of rows of X or `tbl`.

If you specify the input data as a table `tbl`, then Weights can be the name of a variable in `tbl` that contains a numeric vector. In this case, you must specify Weights as a character vector containing the variable name. For example, if the weights vector `w` is stored as `tbl.w`, then specify it as 'w'. Otherwise, the software treats all columns of `tbl`, including `w`, as predictors when training the model.

The software normalizes the weights in each class to add up to the value of the prior probability of the class.

Data Types: `single` | `double`

## Output Arguments

**mdl — Classifier model**

classifier model object

*k*-nearest neighbor classifier model, returned as a classifier model object.

Note that using the 'CrossVal', 'Kfold', 'Holdout', 'Leaveout', or 'CVPartition' options results in a model of class

`ClassificationPartitionedModel`. You cannot use a partitioned tree for prediction, so this kind of tree does not have a `predict` method.

Otherwise, `mdl` is of class `ClassificationKNN`, and you can use the `predict` method to make predictions.

## Examples

### Train a $k$ -Nearest Neighbor Classifier

Construct a  $k$ -nearest neighbor classifier for Fisher's iris data, where  $k$ , the number of nearest neighbors in the predictors, is 5.

Load Fisher's iris data.

```
load fisheriris
X = meas;
Y = species;
```

`X` is a numeric matrix that contains four petal measurements for 150 irises. `Y` is a cell array of character vectors that contains the corresponding iris species.

Train a 5-nearest neighbors classifier. It is good practice to standardize noncategorical predictor data.

```
Mdl = fitcknn(X,Y,'NumNeighbors',5,'Standardize',1)
```

```
Mdl =
```

```
ClassificationKNN
    ResponseName: 'Y'
  CategoricalPredictors: []
        ClassNames: {'setosa' 'versicolor' 'virginica'}
    ScoreTransform: 'none'
  NumObservations: 150
           Distance: 'euclidean'
    NumNeighbors: 5
```

Mdl is a trained ClassificationKNN classifier, and some of its properties display in the Command Window.

To access the properties of Mdl, use dot notation.

```
Mdl.ClassNames
Mdl.Prior

ans =

    3x1 cell array

    {'setosa'      }
    {'versicolor'}
    {'virginica'   }

ans =

    0.3333    0.3333    0.3333
```

Mdl.Prior contains the class prior probabilities, which are settable using the name-value pair argument 'Prior' in fitcknn. The order of the class prior probabilities corresponds to the order of the classes in Mdl.ClassNames. By default, the prior probabilities are the respective relative frequencies of the classes in the data.

You can also reset the prior probabilities after training. For example, set the prior probabilities to 0.5, 0.2, and 0.3 respectively.

```
Mdl.Prior = [0.5 0.2 0.3];
```

You can pass Mdl to, for example, ClassificationKNN.predict to label new measurements, or ClassificationKNN.crossval to cross validate the classifier.

## Train a $k$ -Nearest Neighbor Classifier Using the Minkowski Metric

Load Fisher's iris data set.

```
load fisheriris
X = meas;
Y = species;
```

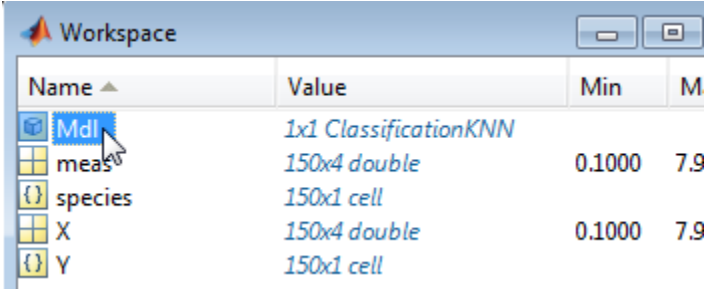
`X` is a numeric matrix that contains four petal measurements for 150 irises. `Y` is a cell array of character vectors that contains the corresponding iris species.

Train a 3-nearest neighbors classifier using the Minkowski metric. To use the Minkowski metric, you must use an exhaustive searcher. It is good practice to standardize noncategorical predictor data.

```
Mdl = fitcknn(X,Y,'NumNeighbors',3,...
    'NSMethod','exhaustive','Distance','minkowski',...
    'Standardize',1);
```

`Mdl` is a `ClassificationKNN` classifier.

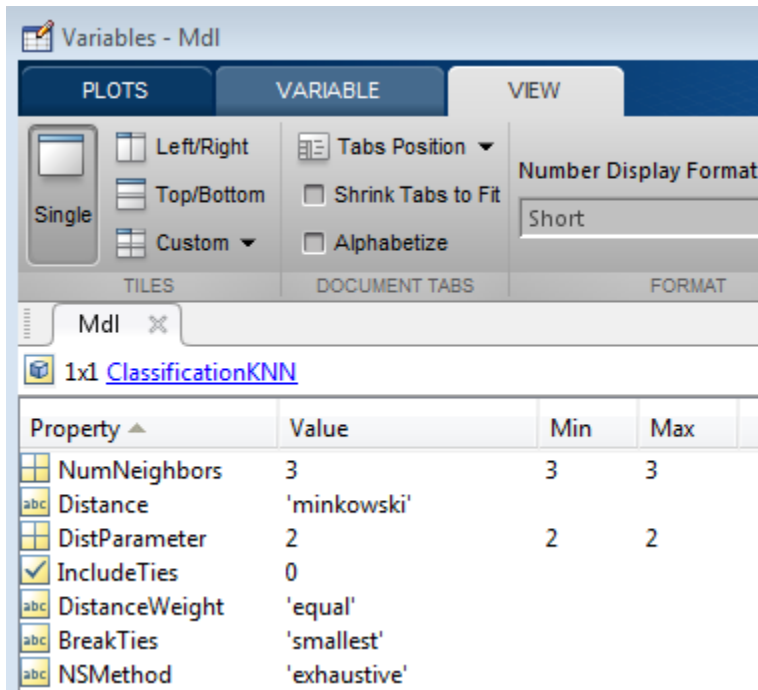
You can examine the properties of `Mdl` by double-clicking `Mdl` in the Workspace window. This opens the Variable Editor.



The screenshot shows the MATLAB Workspace window with a table of variables. The table has four columns: Name, Value, Min, and M. The variable `Mdl` is highlighted with a mouse cursor. The other variables are `meas`, `species`, `X`, and `Y`.

Name	Value	Min	M
Mdl	1x1 ClassificationKNN		
meas	150x4 double	0.1000	7.9
species	150x1 cell		
X	150x4 double	0.1000	7.9
Y	150x1 cell		





- “Construct KNN Classifier” on page 18-36
- “Modify KNN Classifier” on page 18-39

## Definitions

### Prediction

`ClassificationKNN` predicts the classification of a point  $X_{new}$  using a procedure equivalent to this:

- 1 Find the `NumNeighbors` points in the training set  $X$  that are nearest to  $X_{new}$ .
- 2 Find the `NumNeighbors` response values  $Y$  to those nearest points.
- 3 Assign the classification label  $Y_{new}$  that has the largest posterior probability among the values in  $Y$ .

For details, see “Posterior Probability” on page 32-4897 in the `predict` documentation.

## See Also

`ClassificationKNN` | `fitcknn` | `predict`

## Topics

“Construct KNN Classifier” on page 18-36

“Modify KNN Classifier” on page 18-39

“Classification Using Nearest Neighbors” on page 18-16

# ClassificationTree.fit

**Class:** ClassificationTree

Fit classification tree (to be removed)

---

**Note** `ClassificationTree.fit` will be removed in a future release. Use `fitctree` instead.

---

## Syntax

```
tree = ClassificationTree.fit(x, y)
tree = ClassificationTree.fit(x, y, Name, Value)
```

## Description

`tree = ClassificationTree.fit(x, y)` returns a classification tree based on the input variables (also known as predictors, features, or attributes) `x` and output (response) `y`. `tree` is a binary tree, where each branching node is split based on the values of a column of `x`.

`tree = ClassificationTree.fit(x, y, Name, Value)` fits a tree with additional options specified by one or more `Name, Value` pair arguments. You can specify several name-value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Note that using the `'CrossVal'`, `'KFold'`, `'Holdout'`, `'Leaveout'`, or `'CVPartition'` options results in a tree of class `ClassificationPartitionedModel`. You cannot use a partitioned tree for prediction, so this kind of tree does not have a `predict` method.

Otherwise, `tree` is of class `ClassificationTree`, and you can use the `predict` method to make predictions.

## Input Arguments

### **x** — Predictor values

matrix of floating point values

Predictor values, specified as a matrix of floating point values.

`ClassificationTree.fit` considers NaN values in `x` as missing values.

`ClassificationTree.fit` does not use observations with all missing values for `x` in the fit. `ClassificationTree.fit` uses observations with some missing values for `x` to find splits on variables for which these observations have valid values.

Data Types: `single` | `double`

### **y** — predictor values

numeric vector | categorical vector | logical vector | character array | cell array of character vectors

Predictor values, specified as a numeric vector, categorical vector, logical vector, character array, or cell array of character vectors.

Each row of `y` represents the classification of the corresponding row of `x`. For numeric `y`, consider using `fitrtree` instead. `ClassificationTree.fit` considers NaN values in `y` to be missing values.

`ClassificationTree.fit` does not use observations with missing values for `y` in the fit.

Data Types: `single` | `double` | `char` | `logical` | `cell`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### **AlgorithmForCategorical** — Algorithm for best split on categorical predictor

'Exact' | 'PullLeft' | 'PCA' | 'OVAbyClass'

Algorithm to find the best split on a categorical predictor with  $L$  levels for data with  $K \geq 3$  classes, specified as the comma-separated pair consisting of

'AlgorithmForCategorical' and one of the following.

'Exact'	Consider all $2^{L-1} - 1$ combinations
'PullLeft'	Pull left by purity
'PCA'	Principal component-based partition
'OVABYClass'	One versus all by class

ClassificationTree.fit selects the optimal subset of algorithms for each split using the known number of classes and levels of a categorical predictor. For  $K = 2$  classes, ClassificationTree.fit always performs the exact search.

Example: 'AlgorithmForCategorical', 'PCA'

### **CategoricalPredictors** — Categorical predictors list

numeric or logical vector | cell array of character vectors | character matrix | 'all'

Categorical predictors list, specified as the comma-separated pair consisting of 'CategoricalPredictors' and one of the following.

- A numeric vector with indices from 1 to  $p$ , where  $p$  is the number of columns of  $x$ .
- A logical vector of length  $p$ , where a true entry means that the corresponding column of  $x$  is a categorical variable.
- A cell array of character vectors, where each element in the array is the name of a predictor variable. The names must match entries in PredictorNames values.
- A character matrix, where each row of the matrix is a name of a predictor variable. The names must match entries in PredictorNames values. Pad the names with extra blanks so each row of the character matrix has the same length.
- 'all', meaning all predictors are categorical.

Example: 'CategoricalPredictors', 'all'

Data Types: single | double | char

### **ClassNames** — Class names

numeric vector | categorical vector | logical vector | character array | cell array of character vectors

Class names, specified as the comma-separated pair consisting of 'ClassNames' and an array representing the class names. Use the same data type as the values that exist in `y`.

Use `ClassNames` to order the classes or to select a subset of classes for training. The default is the class names that exist in `y`.

Data Types: `single` | `double` | `char` | `logical` | `cell`

### **Cost** — Cost of misclassification

`square matrix` | `structure`

Cost of misclassification a point, specified as the comma-separated pair consisting of 'Cost' and one of the following.

- Square matrix, where  $\text{Cost}(i, j)$  is the cost of classifying a point into class `j` if its true class is `i`.
- Structure `S` having two fields: `S.ClassNames` containing the group names as a variable of the same type as `y`, and `S.ClassificationCosts` containing the cost matrix.

The default is  $\text{Cost}(i, j) = 1$  if  $i \neq j$ , and  $\text{Cost}(i, j) = 0$  if  $i = j$

Data Types: `single` | `double` | `struct`

### **CrossVal** — Flag to grow cross-validated tree

'off' (default) | 'on'

Flag to grow a cross-validated decision tree, specified as the comma-separated pair consisting of 'CrossVal' and either 'on' or 'off'.

If 'on', `ClassificationTree.fit` grows a cross-validated decision tree with 10 folds. You can override this cross-validation setting using one of the 'KFold', 'Holdout', 'Leaveout', or 'CVPartition' name-value pair arguments. Note that you can only use one of these four options ('KFold', 'Holdout', 'Leaveout', or 'CVPartition') at a time when creating a cross-validated tree.

Alternatively, cross-validate `tree` later using the `crossval` method.

Example: 'CrossVal', 'on'

### **CVPartition** — Partition for cross-validation tree

`cvpartition` object

Partition to use in a cross-validated tree, specified as the comma-separated pair consisting of 'CVPartition' and an object of the `cvpartition` class created using `cvpartition`.

Note that if you use 'CVPartition', you cannot use any of the 'KFold', 'Holdout', or 'Leaveout' name-value pair arguments.

#### **Holdout — Fraction of data for holdout validation**

1 (default) | scalar value in the range (0,1]

Fraction of data used for holdout validation, specified as the comma-separated pair consisting of 'Holdout' and a scalar value in the range [0,1]. Holdout validation tests the specified fraction of the data, and uses the rest of the data for training.

Note that if you use 'Holdout', you cannot use any of the 'CVPartition', 'KFold', or 'Leaveout' name-value pair arguments.

Example: 'Holdout',0.1

Data Types: `single` | `double`

#### **KFold — Number of folds**

10 (default) | positive integer value

Number of folds to use in a cross-validated tree, specified as the comma-separated pair consisting of 'KFold' and a positive integer value.

Note that if you use 'KFold', you cannot use any of the 'CVPartition', 'Holdout', or 'Leaveout' name-value pair arguments.

Example: 'KFold',8

Data Types: `single` | `double`

#### **Leaveout — Leave-one-out cross validation flag**

'off' (default) | 'on'

Leave-one-out cross validation flag, specified as the comma-separated pair consisting of 'Leaveout' and either 'on' or 'off'. Use leave-one-out cross validation by setting to 'on'.

Note that if you use 'Leaveout', you cannot use any of the 'CVPartition', 'Holdout', or 'KFold' name-value pair arguments.

Example: 'Leaveout', 'on'

**MaxCat — Maximum category levels**

10 (default) | nonnegative scalar value

Maximum category levels, specified as the comma-separated pair consisting of 'MaxCat' and a nonnegative scalar value. `ClassificationTree.fit` splits a categorical predictor using the exact search algorithm if the predictor has at most MaxCat levels in the split node. Otherwise, `ClassificationTree.fit` finds the best categorical split using one of the inexact algorithms.

Note that passing a small value can lead to loss of accuracy and passing a large value can lead to long computation time and memory overload.

Example: 'MaxCat', 8

**MergeLeaves — Leaf merge flag**

'on' (default) | 'off'

Leaf merge flag, specified as the comma-separated pair consisting of 'MergeLeaves' and either 'on' or 'off'. When 'on', `ClassificationTree.fit` merges leaves that originate from the same parent node, and that give a sum of risk values greater or equal to the risk associated with the parent node. When 'off', `ClassificationTree.fit` does not merge leaves.

Example: 'MergeLeaves', 'off'

**MinLeaf — Minimum number of leaf node observations**

1 (default) | positive integer value

Minimum number of leaf node observations, specified as the comma-separated pair consisting of 'MinLeaf' and a positive integer value. Each leaf has at least MinLeaf observations per tree leaf. If you supply both MinParent and MinLeaf, `ClassificationTree.fit` uses the setting that gives larger leaves:  $\text{MinParent} = \max(\text{MinParent}, 2 * \text{MinLeaf})$ .

Example: 'MinLeaf', 3

Data Types: single | double

**MinParent — Minimum number of branch node observations**

10 (default) | positive integer value



Minimum number of branch node observations, specified as the comma-separated pair consisting of 'MinParent' and a positive integer value. Each branch node in the tree has at least MinParent observations. If you supply both MinParent and MinLeaf, ClassificationTree.fit uses the setting that gives larger leaves:  
`MinParent=max(MinParent,2*MinLeaf)`.

Example: 'MinParent',8

Data Types: single | double

### **NVarToSample** — Number of predictors for split

'all' | positive integer value

Number of predictors to select at random for each split, specified as the comma-separated pair consisting of 'NVarToSample' and a positive integer value. You can also specify 'all' to use all available predictors.

Example: 'NVarToSample',3

Data Types: single | double

### **PredictorNames** — Predictor variable names

{'x1','x2',...} (default) | cell array of character vectors

Predictor variable names, specified as the comma-separated pair consisting of 'PredictorNames' and a cell array of character vectors containing the names for the predictor variables, in the order in which they appear in `x`.

### **Prior** — Prior probabilities

'empirical' (default) | 'uniform' | vector of scalar values | structure

Prior probabilities for each class, specified as the comma-separated pair consisting of 'Prior' and one of the following.

- 'empirical' determines class probabilities from class frequencies in `y`. If you pass observation weights, they are used to compute the class probabilities.
- 'uniform' sets all class probabilities equal.
- A vector (one scalar value for each class)
- A structure `S` with two fields:
  - `S.ClassNames` containing the class names as a variable of the same type as `y`

- `S.ClassProbs` containing a vector of corresponding probabilities

If you set values for both `weights` and `prior`, the weights are renormalized to add up to the value of the prior probability in the respective class.

Example: `'Prior', 'uniform'`

### **Prune — Pruning flag**

`'on'` (default) | `'off'`

Pruning flag, specified as the comma-separated pair consisting of `'Prune'` and either `'on'` or `'off'`. When `'on'`, `ClassificationTree.fit` grows the classification tree, and computes the optimal sequence of pruned subtrees. When `'off'` `ClassificationTree.fit` grows the classification tree without pruning.

Example: `'Prune', 'off'`

### **PruneCriterion — Pruning criterion**

`'error'` (default) | `'impurity'`

Pruning criterion, specified as the comma-separated pair consisting of `'PruneCriterion'` and either `'error'` or `'impurity'`.

Example: `'PruneCriterion', 'impurity'`

### **ResponseName — Response variable name**

`'Y'` (default) | character vector

Response variable name, specified as the comma-separated pair consisting of `'ResponseName'` and a character vector representing the name of the response variable `Y`.

Example: `'ResponseName', 'Response'`

### **ScoreTransform — Score transform function**

`'none'` | `'symmetric'` | `'invlogit'` | `'ismax'` | function handle | ...

Score transform function, specified as the comma-separated pair consisting of `'ScoreTransform'` and a function handle for transforming scores. Your function should accept a matrix (the original scores) and return a matrix of the same size (the transformed scores).

Alternatively, you can specify one of the following values representing a built-in transformation function.

Value	Description
'doublelogit'	$1/(1 + e^{-2x})$
'invlogit'	$\log(x / (1-x))$
'ismax'	Set the score for the class with the largest score to 1, and set the scores for all other classes to 0.
'logit'	$1/(1 + e^{-x})$
'none' or 'identity'	$x$ (no transformation)
'sign'	-1 for $x < 0$ 0 for $x = 0$ 1 for $x > 0$
'symmetric'	$2x - 1$
'symmetricismax'	Set the score for the class with the largest score to 1, and set the scores for all other classes to -1.
'symmetriclogit'	$2/(1 + e^{-x}) - 1$

Example: 'ScoreTransform', 'logit'

#### SplitCriterion — Split criterion

'gdi' (default) | 'twoing' | 'deviance'

Split criterion, specified as the comma-separated pair consisting of 'SplitCriterion' and 'gdi' (Gini's diversity index), 'twoing' for the twoing rule, or 'deviance' for maximum deviance reduction (also known as cross entropy).

Example: 'SplitCriterion', 'deviance'

#### Surrogate — Surrogate decision splits flag

'off' | 'on' | 'all' | positive integer value

Surrogate decision splits flag, specified as the comma-separated pair consisting of 'Surrogate' and 'on', 'off', 'all', or a positive integer value.

- When set to 'on', ClassificationTree.fit finds at most 10 surrogate splits at each branch node.
- When set to a positive integer value, ClassificationTree.fit finds at most the specified number of surrogate splits at each branch node.

- When set to 'all', `ClassificationTree.fit` finds all surrogate splits at each branch node. The 'all' setting can use much time and memory.

Use surrogate splits to improve the accuracy of predictions for data with missing values. The setting also enables you to compute measures of predictive association between predictors.

Example: `'Surrogate', 'on'`

### **Weights — Observation weights**

`ones(size(x,1),1)` (default) | vector of scalar values

Vector of observation weights, specified as the comma-separated pair consisting of 'Weights' and a vector of scalar values. The length of `Weights` is equal to the number of rows in `x`. `ClassificationTree.fit` normalizes the weights in each class to add up to the value of the prior probability of the class.

Data Types: `single` | `double`

## Output Arguments

### **tree — Classification tree**

classification tree object

Classification tree object, returned as a classification tree object.

Note that using the 'CrossVal', 'KFold', 'Holdout', 'Leaveout', or 'CVPartition' options results in a tree of class `ClassificationPartitionedModel`. You cannot use a partitioned tree for prediction, so this kind of tree does not have a `predict` method. Instead, use `kfoldpredict` to predict responses for observations not used for training.

Otherwise, `tree` is of class `ClassificationTree`, and you can use the `predict` method to make predictions.

## Examples

## Construct a Classification Tree

Construct a classification tree for the data in `ionosphere.mat`.

```
load ionosphere
tc = ClassificationTree.fit(X,Y)

tc =

    ClassificationTree
      ResponseName: 'Y'
    CategoricalPredictors: []
      ClassNames: {'b' 'g'}
      ScoreTransform: 'none'
    NumObservations: 351
```

## Definitions

### Impurity and Node Error

`ClassificationTree` splits nodes based on either impurity or node error.

Impurity means one of several things, depending on your choice of the `SplitCriterion` name-value pair argument:

- Gini's Diversity Index (`gdi`) — The Gini index of a node is

$$1 - \sum_i p^2(i),$$

where the sum is over the classes  $i$  at the node, and  $p(i)$  is the observed fraction of classes with class  $i$  that reach the node. A node with just one class (a pure node) has Gini index 0; otherwise the Gini index is positive. So the Gini index is a measure of node impurity.

- Deviance (`'deviance'`) — With  $p(i)$  defined the same as for the Gini index, the deviance of a node is

$$-\sum_i p(i) \log_2 p(i).$$

A pure node has deviance 0; otherwise, the deviance is positive.

- Twoing rule ('twoing') — Twoing is not a purity measure of a node, but is a different measure for deciding how to split a node. Let  $L(i)$  denote the fraction of members of class  $i$  in the left child node after a split, and  $R(i)$  denote the fraction of members of class  $i$  in the right child node after a split. Choose the split criterion to maximize

$$P(L)P(R)\left(\sum_i |L(i) - R(i)|\right)^2,$$

where  $P(L)$  and  $P(R)$  are the fractions of observations that split to the left and right respectively. If the expression is large, the split made each child node purer.

Similarly, if the expression is small, the split made each child node similar to each other, and hence similar to the parent node, and so the split did not increase node purity.

- Node error — The node error is the fraction of misclassified classes at a node. If  $j$  is the class with the largest number of training samples at a node, the node error is  $1 - p(j)$ .

## References

- [1] Coppersmith, D., S. J. Hong, and J. R. M. Hosking. "Partitioning Nominal Attributes in Decision Trees." *Data Mining and Knowledge Discovery*, Vol. 3, 1999, pp. 197–217.
- [2] Breiman, L., J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Boca Raton, FL: CRC Press, 1984.

## See Also

`ClassificationTree` | `fitctree` | `kfoldpredict` | `predict`

# GeneralizedLinearModel.fit

**Class:** GeneralizedLinearModel

Create generalized linear regression model

---

**Note** GeneralizedLinearModel.fit will be removed in a future release. Use fitglm instead.

---

## Syntax

```
mdl = GeneralizedLinearModel.fit(tbl)
mdl = GeneralizedLinearModel.fit(X,y)
mdl = GeneralizedLinearModel.fit(...,modelspec)
mdl = GeneralizedLinearModel.fit(...,Name,Value)
mdl = GeneralizedLinearModel.fit(...,modelspec,Name,Value)
```

## Description

`mdl = GeneralizedLinearModel.fit(tbl)` creates a generalized linear model of a table or dataset array `tbl`.

`mdl = GeneralizedLinearModel.fit(X,y)` creates a generalized linear model of the responses `y` to a data matrix `X`.

`mdl = GeneralizedLinearModel.fit(...,modelspec)` creates a generalized linear model as specified by `modelspec`.

`mdl = GeneralizedLinearModel.fit(...,Name,Value)` or `mdl = GeneralizedLinearModel.fit(...,modelspec,Name,Value)` creates a generalized linear model with additional options specified by one or more `Name, Value` pair arguments.

## Input Arguments

### **tbl** — Input data

table | dataset array

Input data, specified as a table or dataset array. When `modelspec` is a formula, it specifies the variables to be used as the predictors and response. Otherwise, if you do not specify the predictor and response variables, the last variable is the response variable and the others are the predictor variables by default.

Predictor variables and response variables can be numeric, or any grouping variable type, such as logical or categorical (see “Grouping Variables” on page 2-59).

To set a different column as the response variable, use the `ResponseVar` name-value pair argument. To use a subset of the columns as predictors, use the `PredictorVars` name-value pair argument.

Data Types: `single` | `double` | `logical`

### **x** — Predictor variables

matrix

Predictor variables, specified as an  $n$ -by- $p$  matrix, where  $n$  is the number of observations and  $p$  is the number of predictor variables. Each column of `x` represents one variable, and each row represents one observation.

By default, there is a constant term in the model, unless you explicitly remove it, so do not include a column of 1s in `x`.

Data Types: `single` | `double` | `logical`

### **y** — Response variable

vector

Response variable, specified as an  $n$ -by-1 vector, where  $n$  is the number of observations. Each entry in `y` is the response for the corresponding row of `x`.

### **modelspec** — Model specification

character vector specifying the model |  $t$ -by- $(p+1)$  terms matrix | character vector of the form `'Y ~ terms'`



Model specification, which is the starting model for `stepwiseglm`, specified as one of the following:

- Character vector specifying the type of model.

Character Vector	Model Type
'constant'	Model contains only a constant (intercept) term.
'linear'	Model contains an intercept and linear terms for each predictor.
'interactions'	Model contains an intercept, linear terms, and all products of pairs of distinct predictors (no squared terms).
'purequadratic'	Model contains an intercept, linear terms, and squared terms.
'quadratic'	Model contains an intercept, linear terms, interactions, and squared terms.
'polyijk'	Model is a polynomial with all terms up to degree $i$ in the first predictor, degree $j$ in the second predictor, etc. Use numerals 0 through 9. For example, 'poly2111' has a constant plus all linear and product terms, and also contains terms with predictor 1 squared.

- $t$ -by- $(p+1)$  matrix, namely terms matrix on page 32-1755, specifying terms to include in model, where  $t$  is the number of terms and  $p$  is the number of predictor variables, and plus one is for the response variable.
- Character vector representing a formula on page 32-1758 in the form

`'Y ~ terms'`,

where the `terms` are in “Wilkinson Notation” on page 32-1759.

Example: `'quadratic'`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

**BinomialSize — Number of trials for binomial distribution**

1 (default) | scalar value | vector

Number of trials for binomial distribution, that is the sample size, specified as the comma-separated pair consisting of a scalar value or a vector of the same length as the response. This is the parameter `n` for the fitted binomial distribution. `BinomialSize` applies only when the `Distribution` parameter is `'binomial'`.

If `BinomialSize` is a scalar value, that means all observations have the same number of trials.

As an alternative to `BinomialSize`, you can specify the response as a two-column vector with counts in column 1 and `BinomialSize` in column 2.

Data Types: `single` | `double`

**CategoricalVars — Categorical variable list**

cell array of character vectors | logical or numeric index vector

Categorical variable list, specified as the comma-separated pair consisting of `'CategoricalVars'` and either a cell array of character vectors containing categorical variable names in the table or dataset array `tbl`, or a logical or numeric index vector indicating which columns are categorical.

- If data is in a table or dataset array `tbl`, then, by default, `GeneralizedLinearModel.fit` treats all categorical values, logical values, character arrays, and cell arrays of character vectors as categorical variables.
- If data is in matrix `X`, then the default value of `'CategoricalVars'` is an empty matrix `[]`. That is, no variable is categorical unless you specify it as categorical.

For example, you can specify the observations 2 and 3 out of 6 as categorical using either of the following examples.

Example: `'CategoricalVars', [2,3]`

Example: `'CategoricalVars', logical([0 1 1 0 0 0])`

Data Types: `single` | `double` | `logical` | `cell`

**DispersionFlag — Indicator to compute dispersion parameter**`false` for `'binomial'` and `'poisson'` distributions (default) | `true`

Indicator to compute dispersion parameter for 'binomial' and 'poisson' distributions, specified as the comma-separated pair consisting of 'DispersionFlag' and one of the following.

true	Estimate a dispersion parameter when computing standard errors
false	Default. Use the theoretical value when computing standard errors

The fitting function always estimates the dispersion for other distributions.

Example: 'DispersionFlag', true

#### **Distribution — Distribution of the response variable**

'normal' (default) | 'binomial' | 'poisson' | 'gamma' | 'inverse gaussian'

Distribution of the response variable, specified as the comma-separated pair consisting of 'Distribution' and one of the following.

'normal'	Normal distribution
'binomial'	Binomial distribution
'poisson'	Poisson distribution
'gamma'	Gamma distribution
'inverse gaussian'	Inverse Gaussian distribution

Example: 'Distribution', 'gamma'

#### **Exclude — Observations to exclude**

logical or numeric index vector

Observations to exclude from the fit, specified as the comma-separated pair consisting of 'Exclude' and a logical or numeric index vector indicating which observations to exclude from the fit.

For example, you can exclude observations 2 and 3 out of 6 using either of the following examples.

Example: 'Exclude', [2,3]

Example: 'Exclude', logical([0 1 1 0 0 0])

Data Types: single | double | logical

**Intercept — Indicator for constant term**

true (default) | false

Indicator for the constant term (intercept) in the fit, specified as the comma-separated pair consisting of 'Intercept' and either true to include or false to remove the constant term from the model.

Use 'Intercept' only when specifying the model using a character vector, not a formula or matrix.

Example: 'Intercept', false

**Link — Link function**

The canonical link function (default) | scalar value | structure

Link function to use in place of the canonical link function, specified as the comma-separated pair consisting of 'Link' and one of the following.

Link Function Name	Link Function	Mean (Inverse) Function
'identity'	$f(\mu) = \mu$	$\mu = Xb$
'log'	$f(\mu) = \log(\mu)$	$\mu = \exp(Xb)$
'logit'	$f(\mu) = \log(\mu/(1-\mu))$	$\mu = \exp(Xb) / (1 + \exp(Xb))$
'probit'	$f(\mu) = \Phi^{-1}(\mu)$	$\mu = \Phi(Xb)$
'comloglog'	$f(\mu) = \log(-\log(1 - \mu))$	$\mu = 1 - \exp(-\exp(Xb))$
'reciprocal'	$f(\mu) = 1/\mu$	$\mu = 1/(Xb)$
p (a number)	$f(\mu) = \mu^p$	$\mu = Xb^{1/p}$

Link Function Name	Link Function	Mean (Inverse) Function
<p>S (a structure) with three fields. Each field holds a function handle that accepts a vector of inputs and returns a vector of the same size:</p> <ul style="list-style-type: none"> <li>• <code>S.Link</code> — The link function</li> <li>• <code>S.Inverse</code> — The inverse link function</li> <li>• <code>S.Derivative</code> — The derivative of the link function</li> </ul>	$f(\mu) = S.Link(\mu)$	$\mu = S.Inverse(Xb)$

The link function defines the relationship  $f(\mu) = X*b$  between the mean response  $\mu$  and the linear combination of predictors  $X*b$ .

For more information on the canonical link functions, see [Definitions](#).

Example: `'Link', 'probit'`

#### **Offset — Offset variable**

`[]` (default) | `vector` | character vector

Offset variable in the fit, specified as the comma-separated pair consisting of `'Offset'` and a vector or name of a variable with the same length as the response.

`fitglm` and `stepwiseglm` use `Offset` as an additional predictor, with a coefficient value fixed at 1.0. In other words, the formula for fitting is

$$\mu \sim \text{Offset} + (\text{terms involving real predictors})$$

with the `Offset` predictor having coefficient 1.

For example, consider a Poisson regression model. Suppose the number of counts is known for theoretical reasons to be proportional to a predictor `A`. By using the log link function and by specifying `log(A)` as an offset, you can force the model to satisfy this theoretical constraint.

Data Types: `single` | `double` | `char`

### **PredictorVars** — Predictor variables

cell array of character vectors | logical or numeric index vector

Predictor variables to use in the fit, specified as the comma-separated pair consisting of 'PredictorVars' and either a cell array of character vectors of the variable names in the table or dataset array `tbl`, or a logical or numeric index vector indicating which columns are predictor variables.

The character vectors should be among the names in `tbl`, or the names you specify using the 'VarNames' name-value pair argument.

The default is all variables in `X`, or all variables in `tbl` except for `ResponseVar`.

For example, you can specify the second and third variables as the predictor variables using either of the following examples.

Example: `'PredictorVars', [2,3]`

Example: `'PredictorVars', logical([0 1 1 0 0 0])`

Data Types: `single` | `double` | `logical` | `cell`

### **ResponseVar** — Response variable

last column in `tbl` (default) | character vector containing variable name | logical or numeric index vector

Response variable to use in the fit, specified as the comma-separated pair consisting of 'ResponseVar' and either a character vector containing the variable name in the table or dataset array `tbl`, or a logical or numeric index vector indicating which column is the response variable. You typically need to use 'ResponseVar' when fitting a table or dataset array `tbl`.

For example, you can specify the fourth variable, say `yield`, as the response out of six variables, in one of the following ways.

Example: `'ResponseVar', 'yield'`

Example: `'ResponseVar', [4]`

Example: `'ResponseVar', logical([0 0 0 1 0 0])`

Data Types: `single` | `double` | `logical` | `char`

**VarNames — Names of variables in fit**

`{'x1', 'x2', ..., 'xn', 'y'}` (default) | cell array of character vectors

Names of variables in fit, specified as the comma-separated pair consisting of 'VarNames' and a cell array of character vectors including the names for the columns of  $X$  first, and the name for the response variable  $y$  last.

'VarNames' is not applicable to variables in a table or dataset array, because those variables already have names.

For example, if in your data, horsepower, acceleration, and model year of the cars are the predictor variables, and miles per gallon (MPG) is the response variable, then you can name the variables as follows.

Example: 'VarNames', {'Horsepower', 'Acceleration', 'Model\_Year', 'MPG'}

Data Types: cell

**Weights — Observation weights**

`ones(n,1)` (default) |  $n$ -by-1 vector of nonnegative scalar values

Observation weights, specified as the comma-separated pair consisting of 'Weights' and an  $n$ -by-1 vector of nonnegative scalar values, where  $n$  is the number of observations.

Data Types: single | double

## Output Arguments

**mdl — Generalized linear model**

GeneralizedLinearModel object

Generalized linear model representing a least-squares fit of the link of the response to the data, returned as a GeneralizedLinearModel object.

For properties and methods of the generalized linear model object, mdl, see the GeneralizedLinearModel class page.

## Examples

### Fit a Generalized Linear Model

Fit a logistic regression model of probability of smoking as a function of age, weight, and sex, using a two-way interactions model.

Load the hospital dataset array.

```
load hospital
ds = hospital; % just to use the ds name
```

Specify the model using a formula that allows up to two-way interactions.

```
modelspec = 'Smoker ~ Age*Weight*Sex - Age:Weight:Sex';
```

Create the generalized linear model.

```
mdl = fitglm(ds,modelspec,'Distribution','binomial')
```

```
mdl =
```

Generalized linear regression model:

```
logit(Smoker) ~ 1 + Sex*Age + Sex*Weight + Age*Weight
Distribution = Binomial
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	-6.0492	19.749	-0.3063	0.75938
Sex_Male	-2.2859	12.424	-0.18399	0.85402
Age	0.11691	0.50977	0.22934	0.81861
Weight	0.031109	0.15208	0.20455	0.83792
Sex_Male:Age	0.020734	0.20681	0.10025	0.92014
Sex_Male:Weight	0.01216	0.053168	0.22871	0.8191
Age:Weight	-0.00071959	0.0038964	-0.18468	0.85348

100 observations, 93 error degrees of freedom

Dispersion: 1

Chi^2-statistic vs. constant model: 5.07, p-value = 0.535



The large  $P$ -value indicates the model might not differ statistically from a constant.

- “Generalized Linear Model Workflow” on page 12-41

## Definitions

### Terms Matrix

A terms matrix is a  $t$ -by- $(p + 1)$  matrix specifying terms in a model, where  $t$  is the number of terms,  $p$  is the number of predictor variables, and plus one is for the response variable.

The value of  $T(i, j)$  is the exponent of variable  $j$  in term  $i$ . Suppose there are three predictor variables  $A$ ,  $B$ , and  $C$ :

```
[0 0 0 0] % Constant term or intercept
[0 1 0 0] % B; equivalently, A^0 * B^1 * C^0
[1 0 1 0] % A*C
[2 0 0 0] % A^2
[0 1 2 0] % B*(C^2)
```

The 0 at the end of each term represents the response variable. In general,

- If you have the variables in a table or dataset array, then 0 must represent the response variable depending on the position of the response variable. The following example illustrates this.

Load the sample data and define the dataset array.

```
load hospital
dsa = dataset(hospital.Sex,hospital.BloodPressure(:,1),hospital.Age,...
hospital.Smoker, 'VarNames', {'Sex', 'BloodPressure', 'Age', 'Smoker'});
```

Represent the linear model 'BloodPressure ~ 1 + Sex + Age + Smoker' in a terms matrix. The response variable is in the second column of the dataset array, so there must be a column of 0s for the response variable in the second column of the terms matrix.

```
T = [0 0 0 0;1 0 0 0;0 0 1 0;0 0 0 1]
```

```
T =
```

```
0    0    0    0
1    0    0    0
0    0    1    0
0    0    0    1
```

Redefine the dataset array.

```
dsa = dataset(hospital.BloodPressure(:,1),hospital.Sex,hospital.Age,...
hospital.Smoker, 'VarNames',{'BloodPressure','Sex','Age','Smoker'});
```

Now, the response variable is the first term in the dataset array. Specify the same linear model, 'BloodPressure ~ 1 + Sex + Age + Smoker', using a terms matrix.

```
T = [0 0 0 0;0 1 0 0;0 0 1 0;0 0 0 1]
```

```
T =
```

```
0    0    0    0
0    1    0    0
0    0    1    0
0    0    0    1
```

- If you have the predictor and response variables in a matrix and column vector, then you must include 0 for the response variable at the end of each term. The following example illustrates this.

Load the sample data and define the matrix of predictors.

```
load carsmall
X = [Acceleration,Weight];
```

Specify the model 'MPG ~ Acceleration + Weight + Acceleration:Weight + Weight^2' using a term matrix and fit the model to the data. This model includes the main effect and two-way interaction terms for the variables, Acceleration and Weight, and a second-order term for the variable, Weight.

```
T = [0 0 0;1 0 0;0 1 0;1 1 0;0 2 0]
```

```
T =
```

```
0    0    0
1    0    0
0    1    0
```

```

1      1      0
0      2      0

```

### Fit a linear model.

```
mdl = fitlm(X,MPG,T)
```

```
mdl =
```

```
Linear regression model:
```

```
y ~ 1 + x1*x2 + x2^2
```

```
Estimated Coefficients:
```

	Estimate	SE	tStat	pValue
(Intercept)	48.906	12.589	3.8847	0.00019665
x1	0.54418	0.57125	0.95261	0.34337
x2	-0.012781	0.0060312	-2.1192	0.036857
x1:x2	-0.00010892	0.00017925	-0.6076	0.545
x2^2	9.7518e-07	7.5389e-07	1.2935	0.19917

```
Number of observations: 94, Error degrees of freedom: 89
```

```
Root Mean Squared Error: 4.1
```

```
R-squared: 0.751, Adjusted R-Squared 0.739
```

```
F-statistic vs. constant model: 67, p-value = 4.99e-26
```

Only the intercept and x2 term, which correspond to the Weight variable, are significant at the 5% significance level.

Now, perform a stepwise regression with a constant model as the starting model and a linear model with interactions as the upper model.

```
T = [0 0 0;1 0 0;0 1 0;1 1 0];
```

```
mdl = stepwiselm(X,MPG,[0 0 0], 'upper', T)
```

```
1. Adding x2, FStat = 259.3087, pValue = 1.643351e-28
```

```
mdl =
```

```
Linear regression model:
```

```
y ~ 1 + x2
```

```
Estimated Coefficients:
```

	Estimate	SE	tStat	pValue
(Intercept)	49.238	1.6411	30.002	2.7015e-49
x2	-0.0086119	0.0005348	-16.103	1.6434e-28

```
Number of observations: 94, Error degrees of freedom: 92
Root Mean Squared Error: 4.13
R-squared: 0.738, Adjusted R-Squared 0.735
F-statistic vs. constant model: 259, p-value = 1.64e-28
```

The results of the stepwise regression are consistent with the results of `fitlm` in the previous step.

## Formula

A formula for model specification is a character vector of the form `'Y ~ terms'`

where

- `Y` is the response name.
- `terms` contains
  - Variable names
  - `+` means include the next variable
  - `-` means do not include the next variable
  - `:` defines an interaction, a product of terms
  - `*` defines an interaction **and all lower-order terms**
  - `^` raises the predictor to a power, exactly as in `*` repeated, so `^` includes lower order terms as well
  - `()` groups terms

---

**Note** Formulas include a constant (intercept) term by default. To exclude a constant term from the model, include `-1` in the formula.

---

For example,

`'Y ~ A + B + C'` means a three-variable linear model with intercept.

`'Y ~ A + B + C - 1'` is a three-variable linear model without intercept.

`'Y ~ A + B + C + B^2'` is a three-variable model with intercept and a  $B^2$  term.

`'Y ~ A + B^2 + C'` is the same as the previous example because  $B^2$  includes a  $B$  term.

'Y ~ A + B + C + A:B' includes an A\*B term.

'Y ~ A\*B + C' is the same as the previous example because  $A*B = A + B + A:B$ .

'Y ~ A\*B\*C - A:B:C' has all interactions among A, B, and C, except the three-way interaction.

'Y ~ A\*(B + C + D)' has all linear terms, plus products of A with each of the other variables.

## Wilkinson Notation

Wilkinson notation describes the factors present in models. The notation relates to factors present in models, not to the multipliers (coefficients) of those factors.

Wilkinson Notation	Factors in Standard Notation
1	Constant (intercept) term
$A^k$ , where k is a positive integer	A, $A^2$ , ..., $A^k$
A + B	A, B
A*B	A, B, A*B
A:B	A*B only
-B	Do not include B
A*B + C	A, B, C, A*B
A + B + C + A:B	A, B, C, A*B
A*B*C - A:B:C	A, B, C, A*B, A*C, B*C
A*(B + C)	A, B, C, A*B, A*C

Statistics and Machine Learning Toolbox notation always includes a constant term unless you explicitly remove the term using -1.

## Canonical Link Function

The default link function for a generalized linear model is the canonical link function.

### Canonical Link Functions for Generalized Linear Models

Distribution	Link Function Name	Link Function	Mean (Inverse) Function
'normal'	'identity'	$f(\mu) = \mu$	$\mu = Xb$
'binomial'	'logit'	$f(\mu) = \log(\mu/(1-\mu))$	$\mu = \exp(Xb) / (1 + \exp(Xb))$
'poisson'	'log'	$f(\mu) = \log(\mu)$	$\mu = \exp(Xb)$
'gamma'	-1	$f(\mu) = 1/\mu$	$\mu = 1/(Xb)$
'inverse gaussian'	-2	$f(\mu) = 1/\mu^2$	$\mu = (Xb)^{-1/2}$

## Tips

- The generalized linear model `mdl` is a standard linear model unless you specify otherwise with the `Distribution` name-value pair.
- For other methods such as `devianceTest`, or properties of the `GeneralizedLinearModel` object, see `GeneralizedLinearModel`.

## Alternatives

You can also construct a generalized linear model using `fitglm`.

Use `stepwiseglm` to select a model specification automatically. Use `step`, `addTerms`, or `removeTerms` to adjust a fitted model.

## References

- [1] Collett, D. *Modeling Binary Data*. New York: Chapman & Hall, 2002.
- [2] Dobson, A. J. *An Introduction to Generalized Linear Models*. New York: Chapman & Hall, 1990.
- [3] McCullagh, P., and J. A. Nelder. *Generalized Linear Models*. New York: Chapman & Hall, 1990.

## See Also

GeneralizedLinearModel | stepwiseglm

## Topics

“Generalized Linear Model Workflow” on page 12-41

“Generalized Linear Models” on page 12-12

## gmdistribution.fit

**Class:** gmdistribution

Gaussian mixture parameter estimates

---

**Note** `fit` will be removed in a future release. Use `fitgmdist` instead.

---

## Syntax

```
obj = gmdistribution.fit(X,k)
obj = gmdistribution.fit(...,param1, val1, param2, val2, ...)
```

## Description

`obj = gmdistribution.fit(X,k)` uses an Expectation Maximization (EM) algorithm to construct an object `obj` of the `gmdistribution` class containing maximum likelihood estimates of the parameters in a Gaussian mixture model with `k` components for data in the  $n$ -by- $d$  matrix `X`, where  $n$  is the number of observations and  $d$  is the dimension of the data.

`gmdistribution` treats NaN values as missing data. Rows of `X` with NaN values are excluded from the fit.

`obj = gmdistribution.fit(...,param1, val1, param2, val2, ...)` provides control over the iterative EM algorithm. Parameters and values are listed below.



Parameter	Value
'Start'	<p>Method used to choose initial component parameters. One of the following:</p> <ul style="list-style-type: none"> <li>'randSample' — To select <math>k</math> observations from <math>X</math> at random as initial component means. The mixing proportions are uniform. The initial covariance matrices for all components are diagonal, where the element <math>j</math> on the diagonal is the variance of <math>X(:, j)</math>. This is the default.</li> <li>'plus' — The software selects <math>k</math> observations from <math>X</math> using the <code>kmeans++</code> algorithm. The initial mixing proportions are uniform. The initial covariance matrices for all components are diagonal, where the element <math>j</math> on the diagonal is the variance of <math>X(:, j)</math>.</li> <li><math>S</math> — A structure array with fields <code>mu</code>, <code>Sigma</code>, and <code>PComponents</code>. See <code>gmdistribution</code> for descriptions of values.</li> <li><math>s</math> — A vector of length <math>n</math> containing an initial guess of the component index for each point.</li> </ul>
'Replicates'	A positive integer giving the number of times to repeat the EM algorithm, each time with a new set of parameters. The solution with the largest likelihood is returned. A value larger than 1 requires the 'randSample' start method. The default is 1.
'CovType'	'diagonal' if the covariance matrices are restricted to be diagonal; 'full' otherwise. The default is 'full'.
'SharedCov'	Logical <code>true</code> if all the covariance matrices are restricted to be the same (pooled estimate); logical <code>false</code> otherwise.
'Regularize'	A nonnegative regularization number added to the diagonal of covariance matrices to make them positive-definite. The default is 0.
'Options'	Options structure for the iterative EM algorithm, as created by <code>statset.gmdistribution.fit</code> uses the parameters 'Display' with a default value of 'off', 'MaxIter' with a default value of 100, and 'TolFun' with a default value of $1e-6$ .

In some cases, `gmdistribution` may converge to a solution where one or more of the components has an ill-conditioned or singular covariance matrix.

The following issues may result in an ill-conditioned covariance matrix:

- The number of dimension of your data is relatively high and there are not enough observations.
- Some of the features (variables) of your data are highly correlated.
- Some or all the features are discrete.
- You tried to fit the data to too many components.

In general, you can avoid getting ill-conditioned covariance matrices by using one of the following precautions:

- Pre-process your data to remove correlated features.
- Set `'SharedCov'` to `true` to use an equal covariance matrix for every component.
- Set `'CovType'` to `'diagonal'`.
- Use `'Regularize'` to add a very small positive number to the diagonal of every covariance matrix.
- Try another set of initial values.

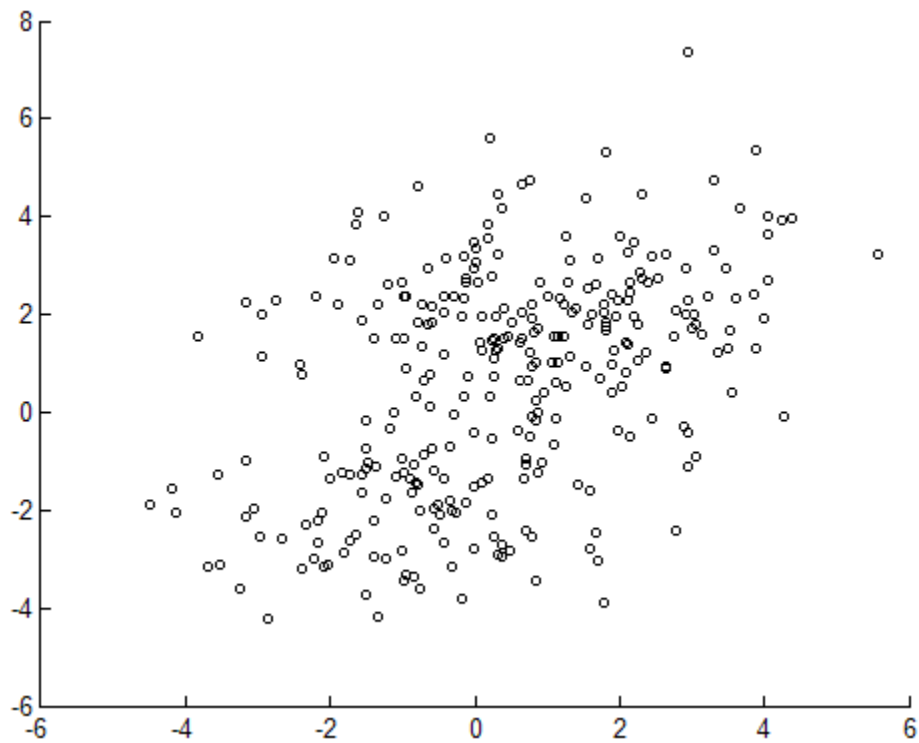
In other cases `gmdistribution` may pass through an intermediate step where one or more of the components has an ill-conditioned covariance matrix. Trying another set of initial values may avoid this issue without altering your data or model.

## Examples

Generate data from a mixture of two bivariate Gaussian distributions using the `mvnrnd` function:

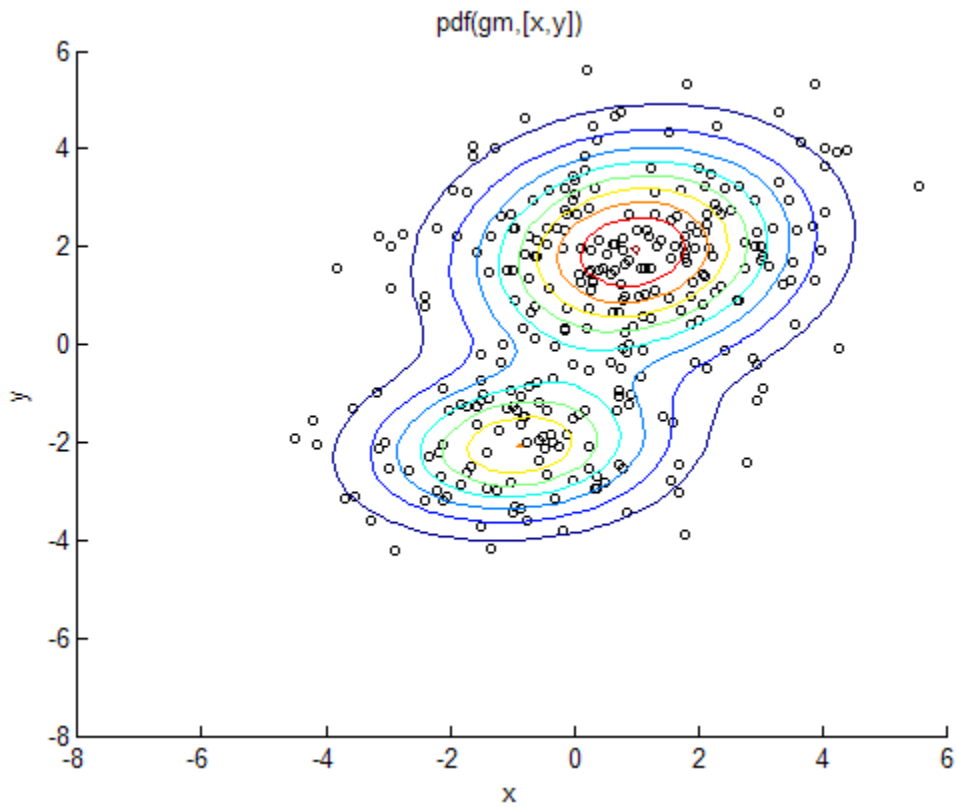
```
MU1 = [1 2];
SIGMA1 = [2 0; 0 .5];
MU2 = [-3 -5];
SIGMA2 = [1 0; 0 1];
X = [mvnrnd(MU1,SIGMA1,1000);mvnrnd(MU2,SIGMA2,1000)];

scatter(X(:,1),X(:,2),10,'.')
hold on
```



Next, fit a two-component Gaussian mixture model:

```
options = statset('Display','final');  
obj = gmdistribution.fit(X,2,'Options',options);  
10 iterations, log-likelihood = -7046.78  
  
h = ezcontour(@(x,y)pdf(obj,[x y]),[-8 6],[-8 6]);
```



Among the properties of the fit are the parameter estimates:

```
ComponentMeans = obj.mu
ComponentMeans =
    0.9391    2.0322
   -2.9823   -4.9737

ComponentCovariances = obj.Sigma
ComponentCovariances(:, :, 1) =
    1.7786   -0.0528
   -0.0528    0.5312
ComponentCovariances(:, :, 2) =
    1.0491   -0.0150
   -0.0150    0.9816
```

```
MixtureProportions = obj.PComponents
MixtureProportions =
    0.5000    0.5000
```

The Akaike information is minimized by the two-component model:

```
AIC = zeros(1,4);
obj = cell(1,4);
for k = 1:4
    obj{k} = gmdistribution.fit(X,k);
    AIC(k) = obj{k}.AIC;
end
```

```
[minAIC,numComponents] = min(AIC);
numComponents
numComponents =
    2
```

```
model = obj{2}
model =
Gaussian mixture distribution
with 2 components in 2 dimensions
Component 1:
Mixing proportion: 0.500000
Mean:      0.9391    2.0322
Component 2:
Mixing proportion: 0.500000
Mean:     -2.9823   -4.9737
```

Both the Akaike and Bayes information are negative log-likelihoods for the data with penalty terms for the number of estimated parameters. They are often used to determine an appropriate number of components for a model when the number of components is unspecified.

## References

- [1] McLachlan, G., and D. Peel. *Finite Mixture Models*. Hoboken, NJ: John Wiley & Sons, Inc., 2000.

## See Also

`cluster` | `gmdistribution`

# LinearModel.fit

**Class:** LinearModel

Create linear regression model

---

**Note** `LinearModel.fit` will be removed in a future release. Use `fitlm` instead.

---

## Syntax

```
mdl = LinearModel.fit(tbl)
mdl = LinearModel.fit(X,y)
mdl = LinearModel.fit( ____,modelspec)
mdl = LinearModel.fit( ____,Name,Value)
mdl = LinearModel.fit( ____,modelspec,Name,Value)
```

## Description

`mdl = LinearModel.fit(tbl)` creates a linear model of a table or dataset array `tbl`.

`mdl = LinearModel.fit(X,y)` creates a linear model of the responses `y` to a data matrix `X`.

`mdl = LinearModel.fit( ____,modelspec)` creates a linear model of the type specified by `modelspec`, using any of the previous syntaxes.

`mdl = LinearModel.fit( ____,Name,Value)` or `mdl = LinearModel.fit( ____,modelspec,Name,Value)` creates a linear model with additional options specified by one or more `Name,Value` pair arguments. For example, you can specify which predictor variables to include in the fit or include observation weights.

## Input Arguments

**tbl** — Input data

table | dataset array

Input data, specified as a table or dataset array. When `modelspec` is a formula, it specifies the variables to be used as the predictors and response. Otherwise, if you do not specify the predictor and response variables, the last variable is the response variable and the others are the predictor variables by default.

Predictor variables can be numeric, or any grouping variable type, such as logical or categorical (see “Grouping Variables” on page 2-59). The response must be numeric or logical.

To set a different column as the response variable, use the `ResponseVar` name-value pair argument. To use a subset of the columns as predictors, use the `PredictorVars` name-value pair argument.

### **x — Predictor variables**

matrix

Predictor variables, specified as an  $n$ -by- $p$  matrix, where  $n$  is the number of observations and  $p$  is the number of predictor variables. Each column of `X` represents one variable, and each row represents one observation.

By default, there is a constant term in the model, unless you explicitly remove it, so do not include a column of 1s in `X`.

Data Types: `single` | `double` | `logical`

### **y — Response variable**

vector

Response variable, specified as an  $n$ -by-1 vector, where  $n$  is the number of observations. Each entry in `y` is the response for the corresponding row of `X`.

Data Types: `single` | `double` | `logical`

### **modelspec — Model specification**

character vector naming the model |  $t$ -by- $(p + 1)$  terms matrix | character vector of the form `'Y ~ terms'`

Model specification, specified as one of the following.

- A character vector naming the model.



Character Vector	Model Type
'constant'	Model contains only a constant (intercept) term.
'linear'	Model contains an intercept and linear terms for each predictor.
'interactions'	Model contains an intercept, linear terms, and all products of pairs of distinct predictors (no squared terms).
'purequadratic'	Model contains an intercept, linear terms, and squared terms.
'quadratic'	Model contains an intercept, linear terms, interactions, and squared terms.
'polyijk'	Model is a polynomial with all terms up to degree $i$ in the first predictor, degree $j$ in the second predictor, etc. Use numerals 0 through 9. For example, 'poly2111' has a constant plus all linear and product terms, and also contains terms with predictor 1 squared.

- $t$ -by- $(p + 1)$  matrix, namely terms matrix on page 32-1781, specifying terms to include in the model, where  $t$  is the number of terms and  $p$  is the number of predictor variables, and plus 1 is for the response variable.
- A character vector representing a formula on page 32-1784 in the form  
`'Y ~ terms'`,  
where the terms are specified using Wilkinson Notation on page 32-1785.

Example: `'quadratic'`

Example: `'y ~ X1 + X2^2 + X1:X2'`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### **CategoricalVars** — Categorical variable list

cell array of character vectors | logical or numeric index vector

Categorical variable list, specified as the comma-separated pair consisting of 'CategoricalVars' and either a cell array of character vectors containing categorical variable names in the table or dataset array `tbl`, or a logical or numeric index vector indicating which columns are categorical.

- If data is in a table or dataset array `tbl`, then, by default, `LinearModel.fit` treats all categorical values, logical values, character arrays, and cell arrays of character vectors as categorical variables.
- If data is in matrix `X`, then the default value of 'CategoricalVars' is an empty matrix `[]`. That is, no variable is categorical unless you specify it as categorical.

For example, you can specify the observations 2 and 3 out of 6 as categorical using either of the following examples.

```
Example: 'CategoricalVars', [2,3]
```

```
Example: 'CategoricalVars', logical([0 1 1 0 0 0])
```

Data Types: single | double | logical | cell

#### **Exclude — Observations to exclude**

logical or numeric index vector

Observations to exclude from the fit, specified as the comma-separated pair consisting of 'Exclude' and a logical or numeric index vector indicating which observations to exclude from the fit.

For example, you can exclude observations 2 and 3 out of 6 using either of the following examples.

```
Example: 'Exclude', [2,3]
```

```
Example: 'Exclude', logical([0 1 1 0 0 0])
```

Data Types: single | double | logical

#### **Intercept — Indicator for constant term**

true (default) | false

Indicator for the constant term (intercept) in the fit, specified as the comma-separated pair consisting of 'Intercept' and either `true` to include or `false` to remove the constant term from the model.

Use 'Intercept' only when specifying the model using a character vector, not a formula or matrix.

Example: 'Intercept', false

### **PredictorVars — Predictor variables**

cell array of character vectors | logical or numeric index vector

Predictor variables to use in the fit, specified as the comma-separated pair consisting of 'PredictorVars' and either a cell array of character vectors of the variable names in the table or dataset array `tbl`, or a logical or numeric index vector indicating which columns are predictor variables.

The character vectors should be among the names in `tbl`, or the names you specify using the 'VarNames' name-value pair argument.

The default is all variables in `X`, or all variables in `tbl` except for `ResponseVar`.

For example, you can specify the second and third variables as the predictor variables using either of the following examples.

Example: 'PredictorVars', [2, 3]

Example: 'PredictorVars', logical([0 1 1 0 0 0])

Data Types: single | double | logical | cell

### **ResponseVar — Response variable**

last column in `tbl` (default) | character vector containing variable name | logical or numeric index vector

Response variable to use in the fit, specified as the comma-separated pair consisting of 'ResponseVar' and either a character vector containing the variable name in the table or dataset array `tbl`, or a logical or numeric index vector indicating which column is the response variable. You typically need to use 'ResponseVar' when fitting a table or dataset array `tbl`.

For example, you can specify the fourth variable, say `yield`, as the response out of six variables, in one of the following ways.

Example: 'ResponseVar', 'yield'

Example: 'ResponseVar', [4]

Example: `'ResponseVar', logical([0 0 0 1 0 0])`

Data Types: `single` | `double` | `logical` | `char`

### **RobustOpts** — Indicator of robust fitting type

`'off'` (default) | `'on'` | character vector | structure with character vector or function handle

Indicator of the robust fitting type to use, specified as the comma-separated pair consisting of `'RobustOpts'` and one of the following.

- `'off'` — No robust fitting. `fitlm` uses ordinary least squares.
- `'on'` — Robust fitting. When you use robust fitting, `'bisquare'` weight function is the default.
- Character vector — Name of the robust fitting weight function from the following table. `fitlm` uses the corresponding default tuning constant in the table.
- Structure with the character vector `RobustWgtFun` containing the name of the robust fitting weight function from the following table and optional scalar `Tune` fields — `fitlm` uses the `RobustWgtFun` weight function and `Tune` tuning constant from the structure. You can choose the name of the robust fitting weight function from this table. If you do not supply a `Tune` field, the fitting function uses the corresponding default tuning constant.

Weight Function	Equation	Default Tuning Constant
<code>'andrews'</code>	$w = (\text{abs}(r) < \pi) .* \sin(r) ./ r$	1.339
<code>'bisquare'</code> (default)	$w = (\text{abs}(r) < 1) .* (1 - r.^2).^2$	4.685
<code>'cauchy'</code>	$w = 1 ./ (1 + r.^2)$	2.385
<code>'fair'</code>	$w = 1 ./ (1 + \text{abs}(r))$	1.400
<code>'huber'</code>	$w = 1 ./ \max(1, \text{abs}(r))$	1.345
<code>'logistic'</code>	$w = \tanh(r) ./ r$	1.205
<code>'ols'</code>	Ordinary least squares (no weighting function)	None
<code>'talwar'</code>	$w = 1 * (\text{abs}(r) < 1)$	2.795
<code>'welsch'</code>	$w = \exp(-r.^2)$	2.985

The value  $r$  in the weight functions is

$$r = \text{resid} / (\text{tune} * s * \sqrt{1-h}),$$

where `resid` is the vector of residuals from the previous iteration, `h` is the vector of leverage values from a least-squares fit, and `s` is an estimate of the standard deviation of the error term given by

$$s = \text{MAD} / 0.6745.$$

MAD is the median absolute deviation of the residuals from their median. The constant 0.6745 makes the estimate unbiased for the normal distribution. If there are  $p$  columns in  $X$ , the smallest  $p$  absolute deviations are excluded when computing the median.

Default tuning constants give coefficient estimates that are approximately 95% as statistically efficient as the ordinary least-squares estimates, provided the response has a normal distribution with no outliers. Decreasing the tuning constant increases the downweight assigned to large residuals; increasing the tuning constant decreases the downweight assigned to large residuals.

- Structure with the function handle `RobustWgtFun` and optional scalar `Tune` fields — You can specify a custom weight function. `fitlm` uses the `RobustWgtFun` weight function and `Tune` tuning constant from the structure. Specify `RobustWgtFun` as a function handle that accepts a vector of residuals, and returns a vector of weights the same size. The fitting function scales the residuals, dividing by the tuning constant (default 1) and by an estimate of the error standard deviation before it calls the weight function.

Example: `'RobustOpts', 'andrews'`

#### **VarNames** — Names of variables in fit

`{'x1', 'x2', ..., 'xn', 'y'}` (default) | cell array of character vectors

Names of variables in fit, specified as the comma-separated pair consisting of `'VarNames'` and a cell array of character vectors including the names for the columns of  $X$  first, and the name for the response variable  $y$  last.

`'VarNames'` is not applicable to variables in a table or dataset array, because those variables already have names.

For example, if in your data, horsepower, acceleration, and model year of the cars are the predictor variables, and miles per gallon (MPG) is the response variable, then you can name the variables as follows.

```
Example: 'VarNames', {'Horsepower', 'Acceleration', 'Model_Year', 'MPG'}
```

```
Data Types: cell
```

### **Weights — Observation weights**

```
ones(n,1) (default) | n-by-1 vector of nonnegative scalar values
```

Observation weights, specified as the comma-separated pair consisting of 'Weights' and an *n*-by-1 vector of nonnegative scalar values, where *n* is the number of observations.

```
Data Types: single | double
```

## Output Arguments

### **mdl — Linear model**

```
LinearModel object
```

Linear model representing a least-squares fit of the response to the data, returned as a `LinearModel` object.

If the value of the 'RobustOpts' name-value pair is not [] or 'ols', the model is not a least-squares fit, but uses the robust fitting function.

For properties and methods of the linear model object, see the `LinearModel` class page.

## Examples

### **Linear Regression Model of Matrix Data**

Fit a linear model of the Hald data.

Load the data.

```
load hald
X = ingredients; % Predictor variables
y = heat; % Response
```

Fit a default linear model to the data.

```
mdl = fitlm(X,y)
```

```
mdl =
```

```
Linear regression model:
  y ~ 1 + x1 + x2 + x3 + x4
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	62.405	70.071	0.8906	0.39913
x1	1.5511	0.74477	2.0827	0.070822
x2	0.51017	0.72379	0.70486	0.5009
x3	0.10191	0.75471	0.13503	0.89592
x4	-0.14406	0.70905	-0.20317	0.84407

```
Number of observations: 13, Error degrees of freedom: 8
Root Mean Squared Error: 2.45
R-squared: 0.982, Adjusted R-Squared 0.974
F-statistic vs. constant model: 111, p-value = 4.76e-07
```

## Linear Regression with Categorical Predictor

Fit a model of a table that contains a categorical predictor.

Load the `carsmall` data.

```
load carsmall
```

Construct a table containing continuous predictor variable `Weight`, nominal predictor variable `Year`, and response variable `MPG`.

```
tbl = table(MPG,Weight);
tbl.Year = nominal(Model_Year);
```

Create a fitted model of MPG as a function of Year, Weight, and Weight<sup>2</sup>. (You don't have to include Weight explicitly in your formula because it is a lower-order term of Weight<sup>2</sup>) and is included automatically.

```
mdl = fitlm(tbl, 'MPG ~ Year + Weight^2')
```

```
mdl =
```

```
Linear regression model:
```

```
MPG ~ 1 + Weight + Year + Weight^2
```

```
Estimated Coefficients:
```

	Estimate	SE	tStat	pValue
(Intercept)	54.206	4.7117	11.505	2.6648e-19
Weight	-0.016404	0.0031249	-5.2493	1.0283e-06
Year_76	2.0887	0.71491	2.9215	0.0044137
Year_82	8.1864	0.81531	10.041	2.6364e-16
Weight^2	1.5573e-06	4.9454e-07	3.149	0.0022303

```
Number of observations: 94, Error degrees of freedom: 89
```

```
Root Mean Squared Error: 2.78
```

```
R-squared: 0.885, Adjusted R-Squared 0.88
```

```
F-statistic vs. constant model: 172, p-value = 5.52e-41
```

fitlm creates two dummy (indicator) variables for the nominal variate, Year. The dummy variable Year\_76 takes the value 1 if model year is 1976 and takes the value 0 if it is not. The dummy variable Year\_82 takes the value 1 if model year is 1982 and takes the value 0 if it is not. And the year 1970 is the reference year. The corresponding model is

$$\hat{MPG} = 54.206 - 0.0164(\text{Weight}) + 2.0887(\text{Year}_76) + 8.1864(\text{Year}_82) + 1.557e - 06(\text{Weight}^2)$$

### Specify Response and Predictor Variables for Linear Model

Fit a linear regression model to sample data. Specify the response and predictor variables, and include only pairwise interaction terms in the model.

Load sample data.



```
load hospital
```

Fit a linear model with interaction terms to the data. Specify weight as the response variable, and sex, age, and smoking status as the predictor variables. Also, specify that sex and smoking status are categorical variables.

```
mdl = fitlm(hospital, 'interactions', 'ResponseVar', 'Weight', ...
    'PredictorVars', {'Sex', 'Age', 'Smoker'}, ...
    'CategoricalVar', {'Sex', 'Smoker'})
```

```
mdl =
```

Linear regression model:

```
Weight ~ 1 + Sex*Age + Sex*Smoker + Age*Smoker
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	118.7	7.0718	16.785	6.821e-30
Sex_Male	68.336	9.7153	7.0339	3.3386e-10
Age	0.31068	0.18531	1.6765	0.096991
Smoker_1	3.0425	10.446	0.29127	0.77149
Sex_Male:Age	-0.49094	0.24764	-1.9825	0.050377
Sex_Male:Smoker_1	0.9509	3.8031	0.25003	0.80312
Age:Smoker_1	-0.07288	0.26275	-0.27737	0.78211

Number of observations: 100, Error degrees of freedom: 93

Root Mean Squared Error: 8.75

R-squared: 0.898, Adjusted R-Squared 0.892

F-statistic vs. constant model: 137, p-value = 6.91e-44

The weight of the patients do not seem to differ significantly according to age, or the status of smoking, or interaction of these factors with patient sex at the 5% significance level.

### Fit a Robust Linear Regression Model

Fit a linear regression model using a robust fitting method.

Load the sample data.

```
load hald
```

The `hald` data measures the effect of cement composition on its hardening heat. The matrix `ingredients` contains the percent composition of four chemicals present in the cement. The array `heat` contains the heat of hardening after 180 days for each cement sample.

Fit a robust linear model to the data.

```
mdl = fitlm(ingredients,heat,'linear','RobustOpts','on')
```

```
mdl =
```

```
Linear regression model (robust fit):  
y ~ 1 + x1 + x2 + x3 + x4
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	60.09	75.818	0.79256	0.4509
x1	1.5753	0.80585	1.9548	0.086346
x2	0.5322	0.78315	0.67957	0.51596
x3	0.13346	0.8166	0.16343	0.87424
x4	-0.12052	0.7672	-0.15709	0.87906

```
Number of observations: 13, Error degrees of freedom: 8  
Root Mean Squared Error: 2.65  
R-squared: 0.979, Adjusted R-Squared 0.969  
F-statistic vs. constant model: 94.6, p-value = 9.03e-07
```

- “Examine Quality and Adjust the Fitted Model” on page 11-20
- “Predict or Simulate Responses to New Data” on page 11-38
- “Linear Regression Workflow” on page 11-43
- “Regression with Categorical Covariates” on page 2-65

## Definitions

### Terms Matrix

A terms matrix is a  $t$ -by- $(p + 1)$  matrix specifying terms in a model, where  $t$  is the number of terms,  $p$  is the number of predictor variables, and plus one is for the response variable.

The value of  $T(i, j)$  is the exponent of variable  $j$  in term  $i$ . Suppose there are three predictor variables A, B, and C:

```
[0 0 0 0] % Constant term or intercept
[0 1 0 0] % B; equivalently, A^0 * B^1 * C^0
[1 0 1 0] % A*C
[2 0 0 0] % A^2
[0 1 2 0] % B*(C^2)
```

The 0 at the end of each term represents the response variable. In general,

- If you have the variables in a table or dataset array, then 0 must represent the response variable depending on the position of the response variable. The following example illustrates this.

Load the sample data and define the dataset array.

```
load hospital
dsa = dataset(hospital.Sex,hospital.BloodPressure(:,1),hospital.Age,...
hospital.Smoker, 'VarNames',{'Sex','BloodPressure','Age','Smoker'});
```

Represent the linear model 'BloodPressure ~ 1 + Sex + Age + Smoker' in a terms matrix. The response variable is in the second column of the dataset array, so there must be a column of 0s for the response variable in the second column of the terms matrix.

```
T = [0 0 0 0;1 0 0 0;0 0 1 0;0 0 0 1]
```

```
T =
```

```

0     0     0     0
1     0     0     0
0     0     1     0
0     0     0     1
```

Redefine the dataset array.

```
dsa = dataset(hospital.BloodPressure(:,1),hospital.Sex,hospital.Age,...  
hospital.Smoker,'VarNames',{'BloodPressure','Sex','Age','Smoker'});
```

Now, the response variable is the first term in the dataset array. Specify the same linear model, 'BloodPressure ~ 1 + Sex + Age + Smoker', using a terms matrix.

```
T = [0 0 0 0;0 1 0 0;0 0 1 0;0 0 0 1]
```

```
T =
```

```
    0    0    0    0  
    0    1    0    0  
    0    0    1    0  
    0    0    0    1
```

- If you have the predictor and response variables in a matrix and column vector, then you must include 0 for the response variable at the end of each term. The following example illustrates this.

Load the sample data and define the matrix of predictors.

```
load carsmall  
X = [Acceleration,Weight];
```

Specify the model 'MPG ~ Acceleration + Weight + Acceleration:Weight + Weight^2' using a term matrix and fit the model to the data. This model includes the main effect and two-way interaction terms for the variables, Acceleration and Weight, and a second-order term for the variable, Weight.

```
T = [0 0 0;1 0 0;0 1 0;1 1 0;0 2 0]
```

```
T =
```

```
    0    0    0  
    1    0    0  
    0    1    0  
    1    1    0  
    0    2    0
```

Fit a linear model.

```
mdl = fitlm(X,MPG,T)
```

```
mdl =
```

```
Linear regression model:
```

```
y ~ 1 + x1*x2 + x2^2
```

```
Estimated Coefficients:
```

	Estimate	SE	tStat	pValue
(Intercept)	48.906	12.589	3.8847	0.00019665
x1	0.54418	0.57125	0.95261	0.34337
x2	-0.012781	0.0060312	-2.1192	0.036857
x1:x2	-0.00010892	0.00017925	-0.6076	0.545
x2^2	9.7518e-07	7.5389e-07	1.2935	0.19917

```
Number of observations: 94, Error degrees of freedom: 89
```

```
Root Mean Squared Error: 4.1
```

```
R-squared: 0.751, Adjusted R-Squared 0.739
```

```
F-statistic vs. constant model: 67, p-value = 4.99e-26
```

Only the intercept and x2 term, which correspond to the weight variable, are significant at the 5% significance level.

Now, perform a stepwise regression with a constant model as the starting model and a linear model with interactions as the upper model.

```
T = [0 0 0;1 0 0;0 1 0;1 1 0];
```

```
mdl = stepwiselm(X,MPG,[0 0 0], 'upper', T)
```

```
1. Adding x2, FStat = 259.3087, pValue = 1.643351e-28
```

```
mdl =
```

```
Linear regression model:
```

```
y ~ 1 + x2
```

```
Estimated Coefficients:
```

	Estimate	SE	tStat	pValue
(Intercept)	49.238	1.6411	30.002	2.7015e-49
x2	-0.0086119	0.0005348	-16.103	1.6434e-28

```
Number of observations: 94, Error degrees of freedom: 92
```

```
Root Mean Squared Error: 4.13
```

```
R-squared: 0.738, Adjusted R-Squared 0.735
```

```
F-statistic vs. constant model: 259, p-value = 1.64e-28
```

The results of the stepwise regression are consistent with the results of `fitlm` in the previous step.

## Formula

A formula for model specification is a character vector of the form `'Y ~ terms'`

where

- `Y` is the response name.
- `terms` contains
  - Variable names
  - `+` means include the next variable
  - `-` means do not include the next variable
  - `:` defines an interaction, a product of terms
  - `*` defines an interaction **and all lower-order terms**
  - `^` raises the predictor to a power, exactly as in `*` repeated, so `^` includes lower order terms as well
  - `()` groups terms

---

**Note** Formulas include a constant (intercept) term by default. To exclude a constant term from the model, include `-1` in the formula.

---

For example,

`'Y ~ A + B + C'` means a three-variable linear model with intercept.

`'Y ~ A + B + C - 1'` is a three-variable linear model without intercept.

`'Y ~ A + B + C + B^2'` is a three-variable model with intercept and a `B^2` term.

`'Y ~ A + B^2 + C'` is the same as the previous example because `B^2` includes a `B` term.

`'Y ~ A + B + C + A:B'` includes an `A*B` term.

`'Y ~ A*B + C'` is the same as the previous example because `A*B = A + B + A:B`.

`'Y ~ A*B*C - A:B:C'` has all interactions among `A`, `B`, and `C`, except the three-way interaction.

' $Y \sim A*(B + C + D)$ ' has all linear terms, plus products of A with each of the other variables.

## Wilkinson Notation

Wilkinson notation describes the factors present in models. The notation relates to factors present in models, not to the multipliers (coefficients) of those factors.

Wilkinson Notation	Factors in Standard Notation
1	Constant (intercept) term
$A^k$ , where k is a positive integer	A, $A^2$ , ..., $A^k$
$A + B$	A, B
$A*B$	A, B, $A*B$
$A:B$	$A*B$ only
$-B$	Do not include B
$A*B + C$	A, B, C, $A*B$
$A + B + C + A:B$	A, B, C, $A*B$
$A*B*C - A:B:C$	A, B, C, $A*B$ , $A*C$ , $B*C$
$A*(B + C)$	A, B, C, $A*B$ , $A*C$

Statistics and Machine Learning Toolbox notation always includes a constant term unless you explicitly remove the term using `-1`.

## Tips

- Use robust fitting (`RobustOpts` name-value pair) to reduce the effect of outliers automatically.
- Do not use robust fitting when you want to subsequently adjust a model using `step`.
- For other methods or properties of the `LinearModel` object, see `LinearModel`.

## Algorithms

The main fitting algorithm is QR decomposition. For robust fitting, the algorithm is `robustfit`.

## Alternatives

You can also construct a linear model using `fitlm`.

You can construct a model in a range of possible models using `stepwiselm`. However, you cannot use robust regression and stepwise regression together.

## See Also

`LinearModel` | `predict` | `stepwiselm`

## Topics

“Examine Quality and Adjust the Fitted Model” on page 11-20

“Predict or Simulate Responses to New Data” on page 11-38

“Linear Regression Workflow” on page 11-43

“Regression with Categorical Covariates” on page 2-65

“Linear Regression” on page 11-11



# LinearMixedModel.fit

**Class:** LinearMixedModel

Fit linear mixed-effects model using tables

---

**Note** `LinearMixedModel.fit` will be removed in a future release. Use `fitlme` instead.

---

## Syntax

```
lme = LinearMixedModel.fit(tbl, formula)
lme = LinearMixedModel.fit(tbl, formula, Name, Value)
```

## Description

`lme = LinearMixedModel.fit(tbl, formula)` returns a linear mixed-effects model, specified by `formula`, fitted to the variables in the table or dataset array `tbl`.

`lme = LinearMixedModel.fit(tbl, formula, Name, Value)` returns a linear mixed-effects model with additional options specified by one or more `Name, Value` pair arguments.

For example, you can specify the covariance pattern of the random-effects terms, the method to use in estimating the parameters, or options for the optimization algorithm.

## Input Arguments

**tbl** — Input data

table | dataset array

Input data, which includes the response variable, predictor variables, and grouping variables, specified as a table or dataset array. The predictor variables can be continuous or grouping variables (see “Grouping Variables” on page 2-59). You must specify the model for the variables using `formula`.

Data Types: `table`

### **formula** — Formula for model specification

character vector of the form `'y ~ fixed + (random1|grouping1) + ... + (randomR|groupingR)'`

Formula for model specification, specified as a character vector of the form `'y ~ fixed + (random1|grouping1) + ... + (randomR|groupingR)'`. For full description, see “Formula” on page 32-1808.

Example: `'y ~ treatment + (1|block)'`

Data Types: `char`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### **CovariancePattern** — Pattern of covariance matrix

`'FullCholesky'` (default) | character vector | square symmetric logical matrix | cell array of character vectors or logical matrices

Pattern of the covariance matrix of the random effects, specified as the comma-separated pair consisting of `'CovariancePattern'` and a character vector, a square symmetric logical matrix, or a cell array of character vectors or logical matrices.

If there are  $R$  random-effects terms, then the value of `'CovariancePattern'` must be a cell array of length  $R$ , where each element  $r$  of this cell array specifies the pattern of the covariance matrix of the random-effects vector associated with the  $r$ th random-effects term. The options for each element follow.

`'FullCholesky'`

Default. Full covariance matrix using the Cholesky parameterization. `fitlme` estimates all elements of the covariance matrix.

'Full'

Full covariance matrix, using the log-Cholesky parameterization. `fitlme` estimates all elements of the covariance matrix.

'Diagonal'

Diagonal covariance matrix. That is, off-diagonal elements of the covariance matrix are constrained to be 0.

$$\begin{pmatrix} \sigma_{b1}^2 & 0 & 0 \\ 0 & \sigma_{b2}^2 & 0 \\ 0 & 0 & \sigma_{b3}^2 \end{pmatrix}$$

'Isotropic'

Diagonal covariance matrix with equal variances. That is, off-diagonal elements of the covariance matrix are constrained to be 0, and the diagonal elements are constrained to be equal. For example, if there are three random-effects terms with an isotropic covariance structure, this covariance matrix looks like

$$\begin{pmatrix} \sigma_b^2 & 0 & 0 \\ 0 & \sigma_b^2 & 0 \\ 0 & 0 & \sigma_b^2 \end{pmatrix}$$

where  $\sigma_b^2$  is the common variance of the random-effects terms.

`'CompSymm'`

Compound symmetry structure. That is, common variance along diagonals and equal correlation between all random effects. For example, if there are three random-effects terms with a covariance matrix having a compound symmetry structure, this covariance matrix looks like

$$\begin{pmatrix} \sigma_{b1}^2 & \sigma_{b1,b2} & \sigma_{b1,b2} \\ \sigma_{b1,b2} & \sigma_{b1}^2 & \sigma_{b1,b2} \\ \sigma_{b1,b2} & \sigma_{b1,b2} & \sigma_{b1}^2 \end{pmatrix}$$

where  $\sigma_{b1}^2$  is the common variance of the random-effects terms and  $\sigma_{b1,b2}$  is the common covariance between any two random-effects term .

`PAT`

Square symmetric logical matrix. If `'CovariancePattern'` is defined by the matrix `PAT`, and if `PAT(a,b) = false`, then the `(a,b)` element of the corresponding covariance matrix is constrained to be 0.

Example: `'CovariancePattern', 'Diagonal'`

Example: `'CovariancePattern', {'Full', 'Diagonal'}`

### **FitMethod** — Method for estimating parameters

`'ML'` (default) | `'REML'`

Method for estimating parameters of the linear mixed-effects model, specified as the comma-separated pair consisting of `'FitMethod'` and either of the following.

`'ML'`

Default. Maximum likelihood estimation

`'REML'`

Restricted maximum likelihood estimation

Example: `'FitMethod', 'REML'`

### **weights** — Observation weights

vector of scalar values

Observation weights, specified as the comma-separated pair consisting of 'Weights' and a vector of length  $n$ , where  $n$  is the number of observations.

Data Types: `single` | `double`

**Exclude** — Indices for rows to exclude

use all rows without NaNs (default) | vector of integer or logical values

Indices for rows to exclude from the linear mixed-effects model in the data, specified as the comma-separated pair consisting of 'Exclude' and a vector of integer or logical values.

For example, you can exclude the 13th and 67th rows from the fit as follows.

Example: `'Exclude', [13, 67]`

Data Types: `single` | `double` | `logical`

**DummyVarCoding** — Coding to use for dummy variables

'reference' (default) | 'effects' | 'full'

Coding to use for dummy variables created from the categorical variables, specified as the comma-separated pair consisting of 'DummyVarCoding' and one of the following.

Value	Description
'reference'	Default. Coefficient for first category set to 0.
'effects'	Coefficients sum to 0.
'full'	One dummy variable for each category.

Example: `'DummyVarCoding', 'effects'`

**Optimizer** — Optimization algorithm

'quasinevton' (default) | 'fminunc'

Optimization algorithm, specified as the comma-separated pair consisting of 'Optimizer' and either of the following.

<code>'quasinevton'</code>	Default. Uses a trust region based quasi-Newton optimizer. Change the options of the algorithm using <code>statset('LinearMixedModel')</code> . If you don't specify the options, then <code>LinearMixedModel</code> uses the default options of <code>statset('LinearMixedModel')</code> .
<code>'fminunc'</code>	You must have Optimization Toolbox to specify this option. Change the options of the algorithm using <code>optimoptions('fminunc')</code> . If you don't specify the options, then <code>LinearMixedModel</code> uses the default options of <code>optimoptions('fminunc')</code> with <code>'Algorithm'</code> set to <code>'quasi-newton'</code> .

Example: `'Optimizer','fminunc'`

### **OptimizerOptions — Options for optimization algorithm**

structure returned by `statset` | object returned by `optimoptions`

Options for the optimization algorithm, specified as the comma-separated pair consisting of `'OptimizerOptions'` and a structure returned by `statset('LinearMixedModel')` or an object returned by `optimoptions('fminunc')`.

- If `'Optimizer'` is `'fminunc'`, then use `optimoptions('fminunc')` to change the options of the optimization algorithm. See `optimoptions` for the options `'fminunc'` uses. If `'Optimizer'` is `'fminunc'` and you do not supply `'OptimizerOptions'`, then the default for `LinearMixedModel` is the default options created by `optimoptions('fminunc')` with `'Algorithm'` set to `'quasi-newton'`.
- If `'Optimizer'` is `'quasinevton'`, then use `statset('LinearMixedModel')` to change the optimization parameters. If you don't change the optimization parameters, then `LinearMixedModel` uses the default options created by `statset('LinearMixedModel')`:

The 'quasinevton' optimizer uses the following fields in the structure created by `statset('LinearMixedModel')`.

**TolFun** — Relative tolerance on gradient of objective function

1e-6 (default) | positive scalar value

Relative tolerance on the gradient of the objective function, specified as a positive scalar value.

**TolX** — Absolute tolerance on step size

1e-12 (default) | positive scalar value

Absolute tolerance on the step size, specified as a positive scalar value.

**MaxIter** — Maximum number of iterations allowed

10000 (default) | positive scalar value

Maximum number of iterations allowed, specified as a positive scalar value.

**Display** — Level of display

'off' (default) | 'iter' | 'final'

Level of display, specified as one of 'off', 'iter', or 'final'.

**StartMethod** — Method to start iterative optimization

'default' (default) | 'random'

Method to start iterative optimization, specified as the comma-separated pair consisting of 'StartMethod' and either of the following.

Value	Description
'default'	An internally defined default value
'random'	A random initial value

Example: 'StartMethod', 'random'

**Verbose** — Indicator to display optimization process on screen

false (default) | true

Indicator to display the optimization process on screen, specified as the comma-separated pair consisting of 'Verbose' and either false or true. Default is false.

The setting for `'Verbose'` overrides the field `'Display'` in `'OptimizerOptions'`.

Example: `'Verbose', true`

### **CheckHessian** — Indicator to check positive definiteness of Hessian

`false` (default) | `true`

Indicator to check the positive definiteness of the Hessian of the objective function with respect to unconstrained parameters at convergence, specified as the comma-separated pair consisting of `'CheckHessian'` and either `false` or `true`. Default is `false`.

Specify `'CheckHessian'` as `true` to verify optimality of the solution or to determine if the model is overparameterized in the number of covariance parameters.

Example: `'CheckHessian', true`

## Output Arguments

### **lme** — Linear mixed-effects model

`LinearMixedModel` object

Linear mixed-effects model, returned as a `LinearMixedModel` object.

## Examples

### **Random-Intercept Model**

Load the sample data.

```
load flu
```

The `flu` dataset array has a `Date` variable, and 10 variables containing estimated influenza rates (in 9 different regions, estimated from Google® searches, plus a nationwide estimate from the CDC).

To fit a linear-mixed effects model, your data must be in a properly formatted dataset array. To fit a linear mixed-effects model with the influenza rates as the responses, combine the nine columns corresponding to the regions into an array. The new dataset



array, flu2, must have the response variable FluRate, the nominal variable Region that shows which region each estimate is from, the nationwide estimate WtdILI, and the grouping variable Date.

```
flu2 = stack(flu, 2:10, 'NewDataVarName', 'FluRate', ...
            'IndVarName', 'Region');
flu2.Date = nominal(flu2.Date);
```

Fit a linear mixed-effects model with the nationwide a random intercept that varies by Date. The model corresponds to

$$y_{im} = \beta_0 + \beta_1 \text{WtdILI}_{im} + b_{0m} + \varepsilon_{im}, \quad i = 1, 2, \dots, 468, \quad m = 1, 2, \dots, 52,$$

where  $y_{im}$  is the observation  $i$  for level  $m$  of grouping variable Date,  $b_{0m}$  is the random effect for level  $m$  of the grouping variable Date, and  $\varepsilon_{im}$  is the observation error for observation  $i$ . The random effect has the prior distribution,

$$b_{0m} \sim N(0, \sigma_b^2),$$

and the error term has the distribution,

$$\varepsilon_{im} \sim N(0, \sigma^2).$$

```
lme = LinearMixedModel.fit(flu2, 'FluRate ~ 1 + WtdILI + (1|Date)')
```

```
lme =
```

```
Linear mixed-effects model fit by ML
```

```
Model information:
```

Number of observations	468
Fixed effects coefficients	2
Random effects coefficients	52
Covariance parameters	2

```
Formula:
```

```
FluRate ~ 1 + WtdILI + (1 | Date)
```

```
Model fit statistics:
```

AIC	BIC	LogLikelihood	Deviance
286.24	302.83	-139.12	278.24

```
Fixed effects coefficients (95% CIs):
```

Name	Estimate	SE	tStat	DF	pValue
'(Intercept)'	0.16385	0.057525	2.8484	466	0.0045885
'WtdILI'	0.7236	0.032219	22.459	466	3.0502e-76

Lower	Upper
0.050813	0.27689
0.66028	0.78691

Random effects covariance parameters (95% CIs):

Group: Date (52 Levels)

Name1	Name2	Type	Estimate	Lower
'(Intercept)'	'(Intercept)'	'std'	0.17146	0.13227

Upper  
0.22226

Group: Error

Name	Estimate	Lower	Upper
'Res Std'	0.30201	0.28217	0.32324

The confidence limits for the standard deviation of the random-effects term,  $\sigma_b$ , do not include 0 (0.13227, 0.22226), which indicates that the random-effects term is significant. You can also test the significance of the random-effects terms using the `LinearMixedModel.compare` method.

The estimated value of an observation is the sum of the fixed effects and the random-effect value at the grouping variable level corresponding to that observation. For example, the estimated flu rate for observation 28 is

$$\begin{aligned}\hat{y}_{28} &= \hat{\beta}_0 + \hat{\beta}_1 \text{WtdILI}_{28} + \hat{b}_{10/30/2005} \\ &= 0.1639 + 0.7236 * (1.343) + 0.3318 \\ &= 1.46749,\end{aligned}$$

where  $\hat{b}$  is the BLUP of the random effects for the intercept. You can compute this value in the following way.

```
beta = fixedEffects(lme);
[~,~,STATS] = randomEffects(lme); % Compute the random-effects statistics (STATS)
STATS.Level = nominal(STATS.Level);
y_hat = beta(1) + beta(2)*flu2.WtdILI(28) + STATS.Estimate(STATS.Level=='10/30/2005')
```

```
y_hat =
    1.4674
```

You can display the fitted value using the `fitted` method.

```
F = fitted(lme);
F(28)

ans =

    1.4674
```

## Randomized-Block Design

Load the sample data.

```
load(fullfile(matlabroot, 'examples', 'stats', 'shift.mat'));
```

The dataset array shows the absolute deviations from the target quality characteristic measured from the products each of five operators manufacture over three different shifts, morning, evening, and night. This is a randomized block design, where the operators are the blocks. The experiment is designed to study the impact of the time of shift on the performance. The performance measure is the absolute deviations of the quality characteristics from the target value. This is simulated data.

Fit a linear mixed-effects model with a random intercept grouped by operator, to assess if there is significant difference in the performance according to the time of the shift. Use the restricted maximum likelihood method and 'effects' contrasts.

'effects' contrasts mean that the coefficients sum to 0, and `LinearMixedModel.fit` creates a matrix called a *fixed effects design matrix* to describe the effect of Shift. This matrix has two columns, `Shift_Evening` and `Shift_Morning`, where

$$\text{Shift\_Evening} = \begin{cases} 0, & \text{if Morning} \\ 1, & \text{if Evening} \\ -1, & \text{if Night} \end{cases}$$

$$\text{Shift\_Morning} = \begin{cases} 1, & \text{if Morning} \\ 0, & \text{if Evening} \\ -1, & \text{if Night} \end{cases}$$

The model corresponds to

$$\text{Morning Shift: } \text{QCDev}_{im} = \beta_0 + \beta_2 \text{Shift\_Morning}_i + b_{0m} + \epsilon_{im},$$

$$\text{Evening Shift: } \text{QCDev}_{im} = \beta_0 + \beta_1 \text{Shift\_Evening}_i + b_{0m} + \epsilon_{im},$$

$$\text{Night Shift: } \text{QCDev}_{im} = \beta_0 - \beta_1 \text{Shift\_Evening}_i - \beta_2 \text{Shift\_Morning}_i + b_{0m} + \epsilon_{im},$$

where  $i$  represents the observations, and  $m$  represents the operators,  $i = 1, 2, \dots, 15$ , and  $m = 1, 2, \dots, 5$ . The random effects and the observation error have the following distributions:

$$b_{0m} \sim N(0, \sigma_b^2)$$

and

$$\epsilon_{im} \sim N(0, \sigma^2).$$

```
lme = LinearMixedModel.fit(shift, 'QCDev ~ Shift + (1|Operator)', ...
'FitMethod', 'REML', 'DummyVarCoding', 'effects')
```

```
lme =
```

```
Linear mixed-effects model fit by REML
```

```
Model information:
```

Number of observations	15
Fixed effects coefficients	3
Random effects coefficients	5
Covariance parameters	2

```
Formula:
```

```
QCDev ~ 1 + Shift + (1 | Operator)
```

```
Model fit statistics:
```

AIC	BIC	LogLikelihood	Deviance
58.913	61.337	-24.456	48.913

```
Fixed effects coefficients (95% CIs):
```

Name	Estimate	SE	tStat	DF	pValue
'(Intercept)'	3.6525	0.94109	3.8812	12	0.0021832

'Shift_Evening'	-0.53293	0.31206	-1.7078	12	0.11339
'Shift_Morning'	-0.91973	0.31206	-2.9473	12	0.012206

Lower	Upper
1.6021	5.703
-1.2129	0.14699
-1.5997	-0.23981

Random effects covariance parameters (95% CIs):

Group: Operator (5 Levels)

Name1	Name2	Type	Estimate	Lower
'(Intercept)'	'(Intercept)'	'std'	2.0457	0.98207

Upper
4.2612

Group: Error

Name	Estimate	Lower	Upper
'Res Std'	0.85462	0.52357	1.395

Compute the best linear unbiased predictor (BLUP) estimates of random effects.

```
B = randomEffects(lme)
```

```
B =
```

```
0.5775
1.1757
-2.1715
2.3655
-1.9472
```

The estimated absolute deviation from the target quality characteristics for the third operator working in the evening shift is

$$\begin{aligned}\hat{y}_{\text{Evening.Operator3}} &= \hat{\beta}_0 + \hat{\beta}_1 \text{Shift.Evening} + \hat{b}_{03} \\ &= 3.6525 - 0.53293 - 2.1715 \\ &= 0.94807.\end{aligned}$$

You can also display this value as follows.

```
F = fitted(lme);  
F(shift.Shift=='Evening' & shift.Operator=='3')  
  
ans =  
  
    0.9481
```

Similarly, you can calculate the estimated absolute deviation from the target quality characteristics for the third operator working in the morning shift is

$$\begin{aligned}\hat{y}_{\text{Morning,Operator3}} &= \hat{\beta}_0 + \hat{\beta}_2 \text{Shift\_Morning} + \hat{b}_{03} \\ &= 3.6525 - 0.91973 - 2.1715 \\ &= 0.56127.\end{aligned}$$

You can also display this value in the following way.

```
F(shift.Shift=='Morning' & shift.Operator=='3')  
  
ans =  
  
    0.5613
```

The operator tends to make a smaller magnitude of error in the morning shift.

### Split-Plot Experiment

Load the sample data.

```
load(fullfile(matlabroot, 'examples', 'stats', 'fertilizer.mat'));
```

The dataset array includes data from a split-plot experiment, where soil is divided into three blocks based on the soil type: sandy, silty, and loamy. Each block is divided into five plots, where five types of tomato plants, (cherry, heirloom, grape, vine, and plum) are randomly assigned to these plots. Then, the tomato plants in the plots are divided into subplots, where each subplot is treated by one of the four fertilizers. This is simulated data.

Store the data in a dataset array called `ds`, for practical purposes, and define `Tomato`, `Soil`, and `Fertilizer` as categorical variables.

```

ds = fertilizer;
ds.Tomato = nominal(ds.Tomato);
ds.Soil = nominal(ds.Soil);
ds.Fertilizer = nominal(ds.Fertilizer);

```

Fit a linear mixed-effects model, where `Fertilizer` and `Tomato` are the fixed-effects variables, and the mean yield varies by the block (soil type) and the plots within blocks (tomato types within soil types) independently.

This model corresponds to

$$y_{imjk} = \beta_0 + \sum_{m=2}^4 \beta_{1m} I[F]_{im} + \sum_{j=2}^5 \beta_{2j} I[T]_{ij} + \sum_{j=2}^5 \sum_{m=2}^4 \beta_{3mj} I[F]_{im} I[T]_{ij} + b_{0k} S_k + b_{0jk} (S * T)_{jk} + \epsilon_{imjk},$$

where  $i = 1, 2, \dots, 60$ , the index  $m$  corresponds to the fertilizer types,  $j$  corresponds to the tomato types, and  $k = 1, 2, 3$  corresponds to the blocks (soil).  $S_k$  represents the  $k$ th soil type, and  $(S * T)_{jk}$  represents the  $j$ th tomato type nested in the  $k$ th soil type.  $I[F]_{im}$  is the dummy variable representing level  $m$  of the fertilizer. Similarly,  $I[T]_{ij}$  is the dummy variable representing the level  $j$  of the tomato type.

The random effects and observation error have the following prior distributions:

$$b_{0k} \sim N(0, \sigma_S^2)$$

$$b_{0jk} \sim N(0, \sigma_{S*T}^2)$$

$$\epsilon_{imjk} \sim N(0, \sigma^2)$$

```
lme = LinearMixedModel.fit(ds, 'Yield ~ Fertilizer * Tomato + (1|Soil) + (1|Soil:Tomato)
```

```
lme =
```

```
Linear mixed-effects model fit by ML
```

```
Model information:
```

Number of observations	60
Fixed effects coefficients	20
Random effects coefficients	18
Covariance parameters	3

```
Formula:
```

```
Yield ~ 1 + Tomato*Fertilizer + (1 | Soil) + (1 | Soil:Tomato)
```

Model fit statistics:

AIC	BIC	LogLikelihood	Deviance
522.57	570.74	-238.29	476.57

Fixed effects coefficients (95% CIs):

Name	Estimate	SE	tStat	DF
'(Intercept)'	77	8.5836	8.9706	40
'Tomato_Grape'	-16	11.966	-1.3371	40
'Tomato_Heirloom'	-6.6667	11.966	-0.55714	40
'Tomato_Plum'	32.333	11.966	2.7022	40
'Tomato_Vine'	-13	11.966	-1.0864	40
'Fertilizer_2'	34.667	8.572	4.0442	40
'Fertilizer_3'	33.667	8.572	3.9275	40
'Fertilizer_4'	47.667	8.572	5.5607	40
'Tomato_Grape:Fertilizer_2'	-2.6667	12.123	-0.21997	40
'Tomato_Heirloom:Fertilizer_2'	-8	12.123	-0.65992	40
'Tomato_Plum:Fertilizer_2'	-15	12.123	-1.2374	40
'Tomato_Vine:Fertilizer_2'	-16	12.123	-1.3198	40
'Tomato_Grape:Fertilizer_3'	16.667	12.123	1.3748	40
'Tomato_Heirloom:Fertilizer_3'	3.3333	12.123	0.27497	40
'Tomato_Plum:Fertilizer_3'	3.6667	12.123	0.30246	40
'Tomato_Vine:Fertilizer_3'	3	12.123	0.24747	40
'Tomato_Grape:Fertilizer_4'	13.333	12.123	1.0999	40
'Tomato_Heirloom:Fertilizer_4'	-19	12.123	-1.5673	40
'Tomato_Plum:Fertilizer_4'	-2.6667	12.123	-0.21997	40
'Tomato_Vine:Fertilizer_4'	8.6667	12.123	0.71492	40

pValue	Lower	Upper
4.0206e-11	59.652	94.348
0.18873	-40.184	8.1837
0.58053	-30.85	17.517
0.010059	8.1496	56.517
0.28379	-37.184	11.184
0.00023272	17.342	51.991
0.00033057	16.342	50.991
1.9567e-06	30.342	64.991
0.82701	-27.167	21.834
0.51309	-32.501	16.501
0.22317	-39.501	9.5007
0.19439	-40.501	8.5007
0.17683	-7.8341	41.167
0.78476	-21.167	27.834



```

0.76387    -20.834    28.167
0.80581    -21.501    27.501
0.27796    -11.167    37.834
0.12492    -43.501    5.5007
0.82701    -27.167    21.834
0.47881    -15.834    33.167

```

Random effects covariance parameters (95% CIs):

Group: Soil (3 Levels)

Name1	Name2	Type	Estimate	Lower
'(Intercept)'	'(Intercept)'	'std'	2.5028	0.02771

Upper  
226.05

Group: Soil:Tomato (15 Levels)

Name1	Name2	Type	Estimate	Lower
'(Intercept)'	'(Intercept)'	'std'	10.225	6.1497

Upper  
17.001

Group: Error

Name	Estimate	Lower	Upper
'Res Std'	10.499	8.5389	12.908

The  $P$ -values corresponding to the last 12 rows in the fixed-effects coefficients display (0.82701 to 0.47881) indicate that interaction coefficients between the tomato and fertilizer types are not significant. To test for the overall interaction between tomato and fertilizer, use the `LinearMixedModel.anova` method after refitting the model using `'effects'` contrasts.

The confidence interval for the standard deviations of the random-effects terms ( $\sigma_s^2$ ), where the intercept is grouped by soil is very large. This term does not appear significant.

Refit the model after removing the interaction term `Tomato:Fertilizer` and the random-effects term `(1 | Soil)`.

```
lme = LinearMixedModel.fit(ds, 'Yield ~ Fertilizer + Tomato + (1|Soil:Tomato)')
```

lme =

Linear mixed-effects model fit by ML

Model information:

Number of observations	60
Fixed effects coefficients	8
Random effects coefficients	15
Covariance parameters	2

Formula:

Yield ~ 1 + Tomato + Fertilizer + (1 | Soil:Tomato)

Model fit statistics:

AIC	BIC	LogLikelihood	Deviance
511.06	532	-245.53	491.06

Fixed effects coefficients (95% CIs):

Name	Estimate	SE	tStat	DF	pValue
'(Intercept)'	77.733	7.3293	10.606	52	1.3108e-14
'Tomato_Grape'	-9.1667	9.6045	-0.95441	52	0.34429
'Tomato_Heirloom'	-12.583	9.6045	-1.3102	52	0.1959
'Tomato_Plum'	28.833	9.6045	3.0021	52	0.0041138
'Tomato_Vine'	-14.083	9.6045	-1.4663	52	0.14858
'Fertilizer_2'	26.333	4.5004	5.8514	52	3.3024e-07
'Fertilizer_3'	39	4.5004	8.6659	52	1.1459e-11
'Fertilizer_4'	47.733	4.5004	10.607	52	1.308e-14

Lower	Upper
63.026	92.441
-28.439	10.106
-31.856	6.6895
9.5605	48.106
-33.356	5.1895
17.303	35.364
29.969	48.031
38.703	56.764

Random effects covariance parameters (95% CIs):

Group: Soil:Tomato (15 Levels)

Name1	Name2	Type	Estimate	Lower
-------	-------	------	----------	-------

```

      '(Intercept)'      '(Intercept)'      'std'      10.02      6.0812

Upper
16.509

Group: Error
Name      Estimate      Lower      Upper
'Res Std' 12.325      10.024    15.153

```

You can compare the two models using the `LinearMixedModel.compare` method with the simulated likelihood ratio test since both a fixed-effect and a random-effect term will be tested.

### Longitudinal Study with a Covariate

Load the sample data.

```
load(fullfile(matlabroot, 'examples', 'stats', 'weight.mat'));
```

`weight` contains data from a longitudinal study, where 20 subjects are randomly assigned to 4 exercise programs (A, B, C, D), and their weight loss is recorded over six two-week time periods. This is simulated data.

Store the data in a table. Define `Subject` and `Program` as categorical variables.

```
tbl = table(InitialWeight, Program, Subject, Week, y);
tbl.Subject = nominal(tbl.Subject);
tbl.Program = nominal(tbl.Program);
```

Fit a linear mixed-effects model where the initial weight, type of program, week, and the interaction between the week and type of program are the fixed effects. The intercept and week vary by subject.

`LinearMixedModel.fit` uses Program A as a reference and creates the necessary dummy variables  $I[.]$ . Since the model already has an intercept, `LinearMixedModel.fit` only creates dummy variables for program types B, C, D. This is also known as the 'reference' method of coding dummy variables.

This model corresponds to

$$\begin{aligned}
 y_{im} = & \beta_0 + \beta_1 IW_i + \beta_2 \text{Week}_i + \beta_3 I[\text{PB}]_i + \beta_4 I[\text{PC}]_i + \beta_5 I[\text{PD}]_i \\
 & + \beta_6 (\text{Week}_i * I[\text{PB}]_i) + \beta_7 (\text{Week}_i * I[\text{PC}]_i) + \beta_8 (\text{Week}_i * I[\text{PD}]_i) \\
 & + b_{0m} + b_{1m} \text{Week}_{im} + \varepsilon_{im},
 \end{aligned}$$

where  $i = 1, 2, \dots, 120$ , and  $m = 1, 2, \dots, 20$ .  $\beta_j$  are the fixed-effects coefficients,  $j = 0, 1, \dots, 8$ , and  $b_{0m}$  and  $b_{1m}$  are random effects.  $IW$  stands for initial weight and  $I[\cdot]$  is a dummy variable representing a type of program. For example,  $I[\text{PB}]_i$  is the dummy variable representing program type B. The random effects and observation error have the following prior distributions:

$$b_{0m} \sim N(0, \sigma_0^2)$$

$$b_{1m} \sim N(0, \sigma_1^2)$$

$$\varepsilon_{im} \sim N(0, \sigma^2).$$

```
lme = LinearMixedModel.fit(tbl, 'y ~ InitialWeight + Program*Week + (Week|Subject)')
```

```
lme =
```

```
Linear mixed-effects model fit by ML
```

```
Model information:
```

Number of observations	120
Fixed effects coefficients	9
Random effects coefficients	40
Covariance parameters	4

```
Formula:
```

```
y ~ 1 + InitialWeight + Program*Week + (1 + Week | Subject)
```

```
Model fit statistics:
```

AIC	BIC	LogLikelihood	Deviance
-22.981	13.257	24.49	-48.981

```
Fixed effects coefficients (95% CIs):
```

Name	Estimate	SE	tStat	DF
'(Intercept)'	0.66105	0.25892	2.5531	111
'InitialWeight'	0.0031879	0.0013814	2.3078	111
'Program_B'	0.36079	0.13139	2.746	111
'Program_C'	-0.033263	0.13117	-0.25358	111
'Program_D'	0.11317	0.13132	0.86175	111

'Week'	0.1732	0.067454	2.5677	111
'Program_B:Week'	0.038771	0.095394	0.40644	111
'Program_C:Week'	0.030543	0.095394	0.32018	111
'Program_D:Week'	0.033114	0.095394	0.34713	111

pValue	Lower	Upper
0.012034	0.14798	1.1741
0.022863	0.00045067	0.0059252
0.0070394	0.10044	0.62113
0.80029	-0.29319	0.22666
0.39068	-0.14706	0.3734
0.011567	0.039536	0.30686
0.68521	-0.15026	0.2278
0.74944	-0.15849	0.21957
0.72915	-0.15592	0.22214

Random effects covariance parameters (95% CIs):

Group: Subject (20 Levels)

Name1	Name2	Type	Estimate	Lower
'(Intercept)'	'(Intercept)'	'std'	0.18407	0.12281
'Week'	'(Intercept)'	'corr'	0.66841	0.21077
'Week'	'Week'	'std'	0.15033	0.11004

Upper  
0.27587  
0.88573  
0.20537

Group: Error

Name	Estimate	Lower	Upper
'Res Std'	0.10261	0.087882	0.11981

The *P*-values 0.022863 and 0.011567 indicate significant effects of subject initial weights and time in the amount of weight lost. The weight loss of subjects who are in Program B is significantly different relative to the weight loss of subjects who are in Program A. The lower and upper limits of the covariance parameters for the random effects do not include

0, thus they are significant. You can also test the significance of the random effects using the `compare` method.

## Definitions

### Formula

In general, a formula for model specification is a character vector of the form `'y ~ terms'`. For the linear mixed-effects models, this formula is in the form `'y ~ fixed + (random1|grouping1) + ... + (randomR|groupingR)'`, where `fixed` and `random` contain the fixed-effects and the random-effects terms.

Suppose a table `tbl` contains the following:

- A response variable,  $y$
- Predictor variables,  $X_j$ , which can be continuous or grouping variables
- Grouping variables,  $g_1, g_2, \dots, g_R$ ,

where the grouping variables in  $X_j$  and  $g_r$  can be categorical, logical, character arrays, or cell arrays of character vectors.

Then, in a formula of the form, `'y ~ fixed + (random1|g1) + ... + (randomR|gR)'`, the term `fixed` corresponds to a specification of the fixed-effects design matrix  $X$ , `random1` is a specification of the random-effects design matrix  $Z_1$  corresponding to grouping variable  $g_1$ , and similarly `randomR` is a specification of the random-effects design matrix  $Z_R$  corresponding to grouping variable  $g_R$ . You can express the `fixed` and `random` terms using Wilkinson notation.

Wilkinson notation describes the factors present in models. The notation relates to factors present in models, not to the multipliers (coefficients) of those factors.

Wilkinson Notation	Factors in Standard Notation
1	Constant (intercept) term
$X^k$ , where $k$ is a positive integer	$X, X^2, \dots, X^k$
$X1 + X2$	$X1, X2$

Wilkinson Notation	Factors in Standard Notation
$X1 * X2$	$X1, X2, X1.*X2$ (elementwise multiplication of $X1$ and $X2$ )
$X1 : X2$	$X1.*X2$ only
$- X2$	Do not include $X2$
$X1 * X2 + X3$	$X1, X2, X3, X1 * X2$
$X1 + X2 + X3 + X1 : X2$	$X1, X2, X3, X1 * X2$
$X1 * X2 * X3 - X1 : X2 : X3$	$X1, X2, X3, X1 * X2, X1 * X3, X2 * X3$
$X1 * (X2 + X3)$	$X1, X2, X3, X1 * X2, X1 * X3$

Statistics and Machine Learning Toolbox notation always includes a constant term unless you explicitly remove the term using  $-1$ . Here are some examples for linear mixed-effects model specification.

#### Examples:

Formula	Description
' $y \sim X1 + X2$ '	Fixed effects for the intercept, $X1$ and $X2$ . This is equivalent to ' $y \sim 1 + X1 + X2$ '.
' $y \sim -1 + X1 + X2$ '	No intercept and fixed effects for $X1$ and $X2$ . The implicit intercept term is suppressed by including $-1$ .
' $y \sim 1 + (1   g1)$ '	Fixed effects for the intercept plus random effect for the intercept for each level of the grouping variable $g1$ .
' $y \sim X1 + (1   g1)$ '	Random intercept model with a fixed slope.
' $y \sim X1 + (X1   g1)$ '	Random intercept and slope, with possible correlation between them. This is equivalent to ' $y \sim 1 + X1 + (1 + X1   g1)$ '.
' $y \sim X1 + (1   g1) + (-1 + X1   g1)$ '	Independent random effects terms for intercept and slope.

Formula	Description
'y ~ 1 + (1   g1) + (1   g2) + (1   g1:g2)'	Random intercept model with independent main effects for g1 and g2, plus an independent interaction effect.

## Cholesky Parameterization

One of the assumptions of linear mixed-effects models is that the random effects have the following prior distribution.

$$b \sim N(0, \sigma^2 D(\theta)),$$

where  $D$  is a  $q$ -by- $q$  symmetric and positive semidefinite matrix, parameterized by a variance component vector  $\theta$ ,  $q$  is the number of variables in the random-effects term, and  $\sigma^2$  is the observation error variance. Since the covariance matrix of the random effects,  $D$ , is symmetric, it has  $q(q+1)/2$  free parameters. Suppose  $L$  is the lower triangular Cholesky factor of  $D(\theta)$  such that

$$D(\theta) = L(\theta)L(\theta)^T,$$

then the  $q^*(q+1)/2$ -by-1 unconstrained parameter vector  $\theta$  is formed from elements in the lower triangular part of  $L$ .

For example, if

$$L = \begin{bmatrix} L_{11} & 0 & 0 \\ L_{21} & L_{22} & 0 \\ L_{31} & L_{32} & L_{33} \end{bmatrix},$$

then

$$\theta = \begin{bmatrix} L_{11} \\ L_{21} \\ L_{31} \\ L_{22} \\ L_{32} \\ L_{33} \end{bmatrix}.$$



## Log-Cholesky Parameterization

When the diagonal elements of  $L$  in Cholesky parameterization are constrained to be positive, then the solution for  $L$  is unique. Log-Cholesky parameterization is the same as Cholesky parameterization except that the logarithm of the diagonal elements of  $L$  are used to guarantee unique parameterization.

For example, for the 3-by-3 example in Cholesky parameterization, enforcing  $L_{ii} \geq 0$ ,

$$\theta = \begin{bmatrix} \log(L_{11}) \\ L_{21} \\ L_{31} \\ \log(L_{22}) \\ L_{32} \\ \log(L_{33}) \end{bmatrix}.$$

## Tips

- If your model is not easily described using a formula, you can create matrices to define the fixed and random effects, and fit the model using `fitlmematrix`.

## Alternatives

You can also construct a linear mixed-effects model using `fitlme`. If your data is in matrix format, then use `fitlmematrix`.

## See Also

`LinearMixedModel` | `anova` | `compare` | `fitlme` | `fitlmematrix`

## fit

**Class:** NaiveBayes

Create Naive Bayes classifier object by fitting training data

## Syntax

```
nb = NaiveBayes.fit(training, class)
nb = NaiveBayes.fit(..., 'param1',val1, 'param2',val2, ...)
```

---

**Note** `fit` will be removed in a future release. Use `fitcnb` instead.

---

## Description

`nb = NaiveBayes.fit(training, class)` builds a NaiveBayes classifier object `nb`. `training` is an N-by-D numeric matrix of training data. Rows of `training` correspond to observations; columns correspond to features. `class` is a classing variable for `training` taking K distinct levels. Each element of `class` defines which class the corresponding row of `training` belongs to. `training` and `class` must have the same number of rows.

`nb = NaiveBayes.fit(..., 'param1',val1, 'param2',val2, ...)` specifies one or more of the following name/value pairs:

- `'Distribution'` – a character vector or a 1-by-D cell array of character vectors, specifying which distributions `fit` uses to model the data. If the value is a `fit` models all the features using one type of distribution. `fit` can also model different features using different types of distributions. If the value is a cell array, its `j`th element specifies the distribution `fit` uses for the `j`th feature. The available types of distributions are:

<code>'normal'</code> (default)	Normal (Gaussian) distribution.
<code>'kernel'</code>	Kernel smoothing density estimate.

- 
- |   |   |
|---|---|
| 'mvmn'  | Multivariate multinomial distribution for discrete data. <code>fit</code> assumes each individual feature follows a multinomial model within a class. The parameters for a feature include the probabilities of all possible values that the corresponding feature can take.  |
| 'mn'  | Multinomial distribution for classifying the count-based data such as the bag-of-tokens model. In the bag-of-tokens model, the value of the $j$ th feature is the number of occurrences of the $j$ th token in this observation, so it must be a nonnegative integer. When 'mn' is used, <code>fit</code> considers each observation as multiple trials of a multinomial distribution, and considers each occurrence of a token as one trial. The number of categories (bins) in this multinomial model is the number of distinct tokens, i.e., the number of columns of <code>training</code> .  |
|   | If you specify <code>mn</code> , then all features are components of a multinomial distribution. Therefore, you cannot include 'mn' as an element of a cell array of character vectors.   |
| • 'Prior' – The prior probabilities for the classes, specified as one of the following: |   |
| 'empirical'<br>(default)  | <code>fit</code> estimates the prior probabilities from the relative frequencies of the classes in <code>training</code> .  |
| 'uniform'   | The prior probabilities are equal for all classes.  |
| vector  | A numeric vector of length $K$ specifying the prior probabilities in the class order of <code>class</code> .  |
| structure   | A structure <code>S</code> containing class levels and their prior probabilities. <code>S</code> must have two fields: <ul style="list-style-type: none"> <li>• <code>S.prob</code>: A numeric vector of prior probabilities.</li> <li>• <code>S.class</code>: A vector of the same type as <code>class</code>, containing unique class levels indicating the class for the corresponding element of <code>prob</code>. <code>S.class</code> must contain all the <math>K</math> levels in <code>class</code>. It can also contain classes that do not appear in <code>class</code>. This can be useful if <code>training</code> is a subset of a larger training set. <code>fit</code> ignores any classes that appear in <code>S.class</code> but not in <code>class</code>.</li> </ul> |

If the prior probabilities don't sum to one, `fit` will normalize them.

- `'KSWidth'` – The bandwidth of the kernel smoothing window. The default is to select a default bandwidth automatically for each combination of feature and class, using a value that is optimal for a Gaussian distribution. You can specify the value as one of the following:
  - scalar      Width for all features in all classes.
  - row vector    1-by-D vector where the  $j$ th element is the bandwidth for the  $j$ th feature in all classes.
  - column vector     $K$ -by-1 vector where the  $i$ th element specifies the bandwidth for all features in the  $i$ th class.  $K$  represents the number of class levels.
  - matrix       $K$ -by-D matrix  $M$  where  $M(i, j)$  specifies the bandwidth for the  $j$ th feature in the  $i$ th class.
  - structure    A structure  $S$  containing class levels and their bandwidths.  $S$  must have two fields:
    - $S.width$  – A numeric array of bandwidths specified as a row vector, or a matrix with  $D$  columns.
    - $S.class$  – A vector of the same type as `class`, containing unique class levels indicating the class for the corresponding row of width.
- `'KSSupport'` – The regions where the density can be applied. It can be `'unbounded'` (default), `'positive'`, a two-element vector as shown below, or a 1-by-D cell array of these values:
  - `'unbounded'` (default)    The density can extend over the whole real line.
  - `'positive'`                The density is restricted to positive values.
  - `[L,U]`                      A two-element vector specifying the finite lower bound  $L$  and upper bound  $U$  for the support of the density.
- `'KSType'` – The type of kernel smoother to use. It can be `'normal'` (default), `'box'`, `'triangle'`, `'epanechnikov'`, or a 1-by-D cell array of these values.

## See Also

### Topics

“Naive Bayes Classification” on page 21-2

“Grouping Variables” on page 2-59

# NonLinearModel.fit

**Class:** NonLinearModel

Fit nonlinear regression model

---

**Note** `NonLinearModel.fit` will be removed in a future release. Use `fitnlm` instead.

---

## Syntax

```
mdl = NonLinearModel.fit(tbl,modelfun,beta0)
mdl = NonLinearModel.fit(X,y,modelfun,beta0)
mdl = NonLinearModel.fit(...,modelfun,beta0,Name,Value)
```

## Description

`mdl = NonLinearModel.fit(tbl,modelfun,beta0)` fits the model specified by `modelfun` to variables in the table or dataset array `tbl`, and returns the nonlinear model `mdl`. `NonLinearModel.fit` estimates model coefficients using an iterative procedure starting from the initial values in `beta0`.

`mdl = NonLinearModel.fit(X,y,modelfun,beta0)` fits a nonlinear regression model using the column vector `y` as a response variable and the columns of the matrix `X` as predictor variables.

`mdl = NonLinearModel.fit(...,modelfun,beta0,Name,Value)` fits a nonlinear regression model with additional options specified by one or more `Name, Value` pair arguments.

## Input Arguments

**tbl** — Input data  
table | dataset array

Input data, specified as a table or dataset array. If you do not specify the predictor and response variables, the last variable is the response variable and the others are the predictor variables by default.

Predictor variables and response variable must be numeric.

To set a different column as the response variable, use the `ResponseVar` name-value pair argument. To use a subset of the columns as predictors, use the `PredictorVars` name-value pair argument.

Data Types: `single` | `double` | `logical`

### **x — Predictor variables**

matrix

Predictor variables, specified as an  $n$ -by- $p$  matrix, where  $n$  is the number of observations and  $p$  is the number of predictor variables. Each column of `X` represents one variable, and each row represents one observation.

By default, there is a constant term in the model, unless you explicitly remove it, so do not include a column of 1s in `X`.

Data Types: `single` | `double` | `logical`

### **y — Response variable**

vector

Response variable, specified as an  $n$ -by-1 vector, where  $n$  is the number of observations. Each entry in `y` is the response for the corresponding row of `X`.

Data Types: `single` | `double` | `logical`

### **modelfun — Functional form of the model**

function handle | character vector of the form `'y ~ f(b1,b2,...,bj,x1,x2,...,xk)'`

Functional form of the model, specified as either of the following.

- Function handle `@modelfun` or `@(b,x) modelfun`, where
  - `b` is a coefficient vector with the same number of elements as `beta0`.
  - `x` is a matrix with the same number of columns as `X` or the number of predictor variable columns of `tbl`.

`modelfun(b, x)` returns a column vector that contains the same number of rows as `x`. Each row of the vector is the result of evaluating `modelfun` on the corresponding row of `x`. In other words, `modelfun` is a vectorized function, one that operates on all data rows and returns all evaluations in one function call. `modelfun` should return real numbers to obtain meaningful coefficients.

- Character vector of the form `'y ~ f(b1,b2,...,bj,x1,x2,...,xk)'`, where `f` represents a scalar function of the scalar coefficient variables `b1,...,bj` and the scalar data variables `x1,...,xk`.

### **beta0 — Coefficients**

numeric vector

Coefficients for the nonlinear model, specified as a numeric vector. `NonLinearModel` starts its search for optimal coefficients from `beta0`.

Data Types: `single` | `double`

## **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### **CoefficientNames — Names of the model coefficients**

`{'b1','b2',..., 'bk'}` (default) | cell array of character vectors

Names of the model coefficients, specified as a cell array of character vectors.

Data Types: `char`

### **ErrorModel — Form of the error variance model**

`'constant'` (default) | `'proportional'` | `'combined'`

Form of the error variance model, specified as one of the following. Each model defines the error using a standard mean-zero and unit-variance variable  $e$  in combination with independent components: the function value  $f$ , and one or two parameters  $a$  and  $b$

<code>'constant'</code> (default)	$y = f + ae$
-----------------------------------	--------------

'proportional'	$y = f + bfe$
'combined'	$y = f + (a + b f )e$

The only allowed error model when using `Weights` is 'constant'.

---

**Note** `options.RobustWgtFun` must have value `[]` when using an error model other than 'constant'.

---

Example: `'ErrorModel', 'proportional'`

### **ErrorParameters** — Initial estimates of the error model parameters

numeric array

Initial estimates of the error model parameters for the chosen `ErrorModel`, specified as a numeric array.

Error Model	Parameters	Default Values
'constant'	$a$	1
'proportional'	$b$	1
'combined'	$a, b$	[1, 1]

You can only use the 'constant' error model when using `Weights`.

---

**Note** `options.RobustWgtFun` must have value `[]` when using an error model other than 'constant'.

---

For example, if 'ErrorModel' has the value 'combined', you can specify the starting value 1 for  $a$  and the starting value 2 for  $b$  as follows.

Example: `'ErrorParameters', [1, 2]`

Data Types: char

### **Exclude** — Observations to exclude

logical or numeric index vector



Observations to exclude from the fit, specified as the comma-separated pair consisting of 'Exclude' and a logical or numeric index vector indicating which observations to exclude from the fit.

For example, you can exclude observations 2 and 3 out of 6 using either of the following examples.

Example: 'Exclude', [2,3]

Example: 'Exclude', logical([0 1 1 0 0 0])

Data Types: single | double | logical

### options — Options for controlling the iterative fitting procedure

[] (default) | structure

Options for controlling the iterative fitting procedure, specified as a structure created by `statset`. The relevant fields are the nonempty fields in the structure returned by the call `statset('nlinfit')`.

Option	Meaning	Default
DerivStep	Relative difference used in finite difference derivative calculations. A positive scalar, or a vector of positive scalars the same size as the vector of parameters estimated by the Statistics and Machine Learning Toolbox function using the options structure.	$\text{eps}^{(1/3)}$
Display	Amount of information displayed by the fitting algorithm. <ul style="list-style-type: none"> <li>'off' — Displays no information.</li> <li>'final' — Displays the final output.</li> <li>'iter' — Displays iterative output to the Command Window.</li> </ul>	'off'
FunValCheck	Character vector indicating to check for invalid values, such as NaN or Inf, from the model function.	'on'
MaxIter	Maximum number of iterations allowed. Positive integer.	200

Option	Meaning	Default
RobustWgtFun	Weight function for robust fitting. Can also be a function handle that accepts a normalized residual as input and returns the robust weights as output. If you use a function handle, give a Tune constant. See “Robust Options” on page 32-2500	[]
Tune	Tuning constant used in robust fitting to normalize the residuals before applying the weight function. A positive scalar. Required if the weight function is specified as a function handle.	See “Robust Options” on page 32-2500 for the default, which depends on RobustWgtFun.
TolFun	Termination tolerance for the objective function value. Positive scalar.	1e-8
TolX	Termination tolerance for the parameters. Positive scalar.	1e-8

Data Types: `struct`

### **PredictorVars** — Predictor variables

cell array of character vectors | logical or numeric index vector

Predictor variables to use in the fit, specified as the comma-separated pair consisting of 'PredictorVars' and either a cell array of character vectors of the variable names in the table or dataset array `tbl`, or a logical or numeric index vector indicating which columns are predictor variables.

The character vectors should be among the names in `tbl`, or the names you specify using the 'VarNames' name-value pair argument.

The default is all variables in `X`, or all variables in `tbl` except for `ResponseVar`.

For example, you can specify the second and third variables as the predictor variables using either of the following examples.

Example: `'PredictorVars', [2,3]`

Example: `'PredictorVars', logical([0 1 1 0 0 0])`

Data Types: `single` | `double` | `logical` | `cell`

**ResponseVar — Response variable**

last column in `tbl` (default) | character vector containing variable name | logical or numeric index vector

Response variable to use in the fit, specified as the comma-separated pair consisting of 'ResponseVar' and either a character vector containing the variable name in the table or dataset array `tbl`, or a logical or numeric index vector indicating which column is the response variable. You typically need to use 'ResponseVar' when fitting a table or dataset array `tbl`.

For example, you can specify the fourth variable, say `yield`, as the response out of six variables, in one of the following ways.

Example: `'ResponseVar', 'yield'`

Example: `'ResponseVar', [4]`

Example: `'ResponseVar', logical([0 0 0 1 0 0])`

Data Types: `single` | `double` | `logical` | `char`

**VarNames — Names of variables in fit**

{'x1', 'x2', ..., 'xn', 'y'} (default) | cell array of character vectors

Names of variables in fit, specified as the comma-separated pair consisting of 'VarNames' and a cell array of character vectors including the names for the columns of `X` first, and the name for the response variable `y` last.

'VarNames' is not applicable to variables in a table or dataset array, because those variables already have names.

For example, if in your data, horsepower, acceleration, and model year of the cars are the predictor variables, and miles per gallon (MPG) is the response variable, then you can name the variables as follows.

Example: `'VarNames', {'Horsepower', 'Acceleration', 'Model_Year', 'MPG'}`

Data Types: `cell`

**Weights — Observation weights**

`ones(n, 1)` (default) | vector of nonnegative scalar values | function handle

Observation weights, specified as a vector of nonnegative scalar values or function handle.

- If you specify a vector, then it must have  $n$  elements, where  $n$  is the number of rows in `tbl` or `y`.
- If you specify a function handle, then the function must accept a vector of predicted response values as input, and return a vector of real positive weights as output.

Given weights, `W`, `NonLinearModel` estimates the error variance at observation `i` by  $\text{MSE} * (1/W(i))$ , where `MSE` is the mean squared error.

Data Types: `single` | `double` | `function_handle`

## Output Arguments

### `mdl` — Nonlinear model

`NonLinearModel` object

Nonlinear model representing a least-squares fit of the response to the data, returned as a `NonLinearModel` object.

If the `Options` structure contains a nonempty `RobustWgtFun` field, the model is not a least-squares fit, but uses the `RobustWgtFun` robust fitting function.

For properties and methods of the nonlinear model object, `mdl`, see the `NonLinearModel` class page.

## Examples

### Nonlinear Model from a Table

Create a nonlinear model for auto mileage based on the `carbig` data.

Load the data and create a nonlinear model.

```
load carbig
tbl = table(Horsepower,Weight,MPG);
modelfun = @(b,x)b(1) + b(2)*x(:,1).^b(3) + ...
    b(4)*x(:,2).^b(5);
beta0 = [-50 500 -1 500 -1];
mdl = NonLinearModel.fit(tbl,modelfun,beta0)
```

```
mdl =
```

```
Nonlinear regression model:
```

```
MPG ~ b1 + b2*Horsepower^b3 + b4*Weight^b5
```

```
Estimated Coefficients:
```

	Estimate	SE	tStat	pValue
b1	-49.383	119.97	-0.41164	0.68083
b2	376.43	567.05	0.66384	0.50719
b3	-0.78193	0.47168	-1.6578	0.098177
b4	422.37	776.02	0.54428	0.58656
b5	-0.24127	0.48325	-0.49926	0.61788

```
Number of observations: 392, Error degrees of freedom: 387
```

```
Root Mean Squared Error: 3.96
```

```
R-Squared: 0.745, Adjusted R-Squared 0.743
```

```
F-statistic vs. constant model: 283, p-value = 1.79e-113
```

## Nonlinear Model from Matrix Data

Create a nonlinear model for auto mileage based on the carbig data.

Load the data and create a nonlinear model.

```
load carbig
X = [Horsepower,Weight];
y = MPG;
modelfun = @(b,x)b(1) + b(2)*x(:,1).^b(3) + ...
    b(4)*x(:,2).^b(5);
beta0 = [-50 500 -1 500 -1];
mdl = NonLinearModel.fit(X,y,modelfun,beta0)
```

```
mdl =
```

```
Nonlinear regression model:
```

```
y ~ b1 + b2*x1^b3 + b4*x2^b5
```

Estimated Coefficients:				
	Estimate	SE	tStat	pValue
	-----	-----	-----	-----
b1	-49.383	119.97	-0.41164	0.68083
b2	376.43	567.05	0.66384	0.50719
b3	-0.78193	0.47168	-1.6578	0.098177
b4	422.37	776.02	0.54428	0.58656
b5	-0.24127	0.48325	-0.49926	0.61788

Number of observations: 392, Error degrees of freedom: 387  
 Root Mean Squared Error: 3.96  
 R-Squared: 0.745, Adjusted R-Squared 0.743  
 F-statistic vs. constant model: 283, p-value = 1.79e-113

### Adjust Fitting Options in the Nonlinear Model

Create a nonlinear model for auto mileage based on the `carbig` data. Strive for more accuracy by lowering the `TolFun` option, and observe the iterations by setting the `Display` option.

Load the data and create a nonlinear model.

```
load carbig
X = [Horsepower,Weight];
y = MPG;
modelfun = @(b,x)b(1) + b(2)*x(:,1).^b(3) + ...
    b(4)*x(:,2).^b(5);
beta0 = [-50 500 -1 500 -1];
```

Create options to lower `TolFun` and to report iterative display, and create a model using the options.

```
opts = statset('Display','iter','TolFun',1e-10);
mdl = NonLinearModel.fit(X,y,modelfun,beta0,'Options',opts);
```

Iteration	SSE	Norm of Gradient	Norm of Step
-----	-----	-----	-----

---

0	1.82248e+06		
1	678600	788810	1691.07
2	616716	6.12739e+06	45.4738
3	249831	3.9532e+06	293.557
4	17675	361544	369.284
5	11746.6	69670.5	169.079
6	7242.22	343738	394.822
7	6250.32	159719	452.941
8	6172.87	91622.9	268.674
9	6077	6957.44	100.208
10	6076.34	6370.39	88.1905
11	6075.75	5199.08	77.9694
12	6075.3	4646.61	69.764
13	6074.91	4235.96	62.9114
14	6074.55	3885.28	57.0647
15	6074.23	3571.1	52.0036
16	6073.93	3286.48	47.5795
17	6073.66	3028.34	43.6844
18	6073.4	2794.31	40.2352
19	6073.17	2582.15	37.1663
20	6072.95	2389.68	34.4243
21	6072.74	2214.84	31.965
22	6072.55	2055.78	29.7516
23	6072.37	1910.83	27.753
24	6072.21	1778.51	25.9428
25	6072.05	1657.5	24.2986
26	6071.9	1546.65	22.8011
27	6071.76	1444.93	21.4338
28	6071.63	1351.44	20.1822
29	6071.51	1265.39	19.0339
30	6071.39	1186.06	17.978
31	6071.28	1112.83	17.0052
32	6071.17	1045.13	16.107
33	6071.07	982.465	15.2762
34	6070.98	924.389	14.5063
35	6070.89	870.498	13.7916
36	6070.8	820.434	13.127
37	6070.72	773.872	12.5081
38	6070.64	730.521	11.9307
39	6070.57	690.117	11.3914
40	6070.5	652.422	10.887
41	6070.43	617.219	10.4144
42	6070.37	584.315	9.97115
43	6070.31	553.53	9.55489

44	6070.25	524.703	9.1635
45	6070.19	497.686	8.79506
46	6070.14	472.345	8.44785
47	6070.08	448.557	8.12028
48	6070.03	426.21	7.81091
49	6069.99	405.201	7.51845
50	6069.94	385.435	7.2417
51	6069.9	366.825	6.97956
52	6069.85	349.293	6.73104
53	6069.81	332.764	6.49523
54	6069.77	317.171	6.27127
55	6069.74	302.452	6.0584
56	6069.7	288.55	5.85591
57	6069.66	275.411	5.66315
58	6069.63	262.986	5.47949
59	6069.6	251.23	5.3044
60	6069.57	240.1	5.13734
61	6069.54	229.558	4.97784
62	6069.51	219.567	4.82545
63	6069.48	210.094	4.67977
64	6069.45	201.108	4.5404
65	6069.43	192.578	4.407
66	6069.4	184.479	4.27923
67	6069.38	176.785	4.15677
68	6069.35	169.472	4.03935
69	6069.33	162.518	3.9267
70	6069.31	155.903	3.81855
71	6069.29	149.608	3.71468
72	6069.26	143.615	3.61486
73	6069.24	137.907	3.5189
74	6069.22	132.468	3.42658
75	6069.21	127.283	3.33774
76	6069.19	122.339	3.25221
77	6069.17	117.623	3.16981
78	6069.15	113.123	3.09041
79	6069.14	108.827	3.01386
80	6069.12	104.725	2.94002
81	6069.1	100.806	2.86877
82	6069.09	97.0611	2.8
83	6069.07	93.4814	2.73358
84	6069.06	90.0584	2.66942
85	6069.05	86.7841	2.60741
86	6069.03	83.6513	2.54745
87	6069.02	80.6528	2.48947



88	6069.01	77.7821	2.43338
89	6068.99	75.0327	2.37908
90	6068.98	72.399	2.32652
91	6068.97	69.8752	2.27561
92	6068.96	67.4561	2.22629
93	6068.95	65.1366	2.17849
94	6068.94	62.9123	2.13216
95	6068.93	60.7784	2.08723
96	6068.92	58.7308	2.04364
97	6068.91	56.7655	2.00135
98	6068.9	54.8787	1.9603
99	6068.89	4349.28	18.1917
100	6068.77	2416.27	14.4439
101	6068.71	1721.26	12.1305
102	6068.66	1228.78	10.289
103	6068.63	884.002	8.82019
104	6068.6	639.615	7.62744
105	6068.58	464.84	6.64627
106	6068.56	338.878	5.82964
107	6068.55	247.508	5.14296
108	6068.54	180.878	4.56032
109	6068.53	132.084	4.06194
110	6068.52	96.2341	3.63255
111	6068.51	69.8362	3.26019
112	6068.51	50.3735	2.93541
113	6068.5	36.0205	2.65062
114	6068.5	25.4452	2.39969
115	6068.49	17.6693	2.17764
116	6068.49	1027.39	14.0164
117	6068.48	544.039	5.3137
118	6068.48	94.0576	2.86664
119	6068.48	113.636	3.73502
120	6068.48	0.518567	1.3705
121	6068.48	4.5944	0.91284
122	6068.48	1.56389	0.629322
123	6068.48	1.13809	0.432547
124	6068.48	0.295936	0.297509

Iterations terminated: relative change in SSE less than OPTIONS.TolFun

### Specify Nonlinear Regression Using Model Character Vector

Specify a nonlinear regression model for estimation using a function handle or character vector.

Load sample data.

```
S = load('reaction');
X = S.reactants;
y = S.rate;
beta0 = S.beta;
```

Use a function handle to specify the Hougen-Watson model for the rate data.

```
mdl = NonLinearModel.fit(X,y,@hougen,beta0)
```

```
mdl =
```

```
Nonlinear regression model:
```

```
  y ~ hougen(b,X)
```

```
Estimated Coefficients:
```

	Estimate	SE	tStat	pValue
b1	1.2526	0.86701	1.4447	0.18654
b2	0.062776	0.043561	1.4411	0.18753
b3	0.040048	0.030885	1.2967	0.23089
b4	0.11242	0.075157	1.4957	0.17309
b5	1.1914	0.83671	1.4239	0.1923

```
Number of observations: 13, Error degrees of freedom: 8
```

```
Root Mean Squared Error: 0.193
```

```
R-Squared: 0.999, Adjusted R-Squared 0.998
```

```
F-statistic vs. zero model: 3.91e+03, p-value = 2.54e-13
```

Alternatively, you can use a character vector to specify the Hougen-Watson model for the rate data.

```
myfun = 'y~(b1*x2-x3/b5)/(1+b2*x1+b3*x2+b4*x3)';
mdl2 = NonLinearModel.fit(X,y,myfun,beta0)
```

```
mdl2 =
```

```
Nonlinear regression model:
```

$$y \sim (b1*x2 - x3/b5)/(1 + b2*x1 + b3*x2 + b4*x3)$$

Estimated Coefficients:

	Estimate	SE	tStat	pValue
b1	1.2526	0.86701	1.4447	0.18654
b2	0.062776	0.043561	1.4411	0.18753
b3	0.040048	0.030885	1.2967	0.23089
b4	0.11242	0.075157	1.4957	0.17309
b5	1.1914	0.83671	1.4239	0.1923

Number of observations: 13, Error degrees of freedom: 8

Root Mean Squared Error: 0.193

R-Squared: 0.999, Adjusted R-Squared 0.998

F-statistic vs. zero model: 3.91e+03, p-value = 2.54e-13

## Estimate Nonlinear Regression Using Robust Fitting Options

Generate sample data from the nonlinear regression model

$$y = b_1 + b_2 \exp[-b_3 x] + \epsilon,$$

where  $b_1$ ,  $b_2$ , and  $b_3$  are coefficients, and the error term is normally distributed with mean 0 and standard deviation 0.5.

```
modelfun = @(b,x) (b(1)+b(2)*exp(-b(3)*x));
```

```
rng('default') % for reproducibility
```

```
b = [1;3;2];
```

```
x = exprnd(2,100,1);
```

```
y = modelfun(b,x) + normrnd(0,0.5,100,1);
```

Set robust fitting options.

```
opts = statset('nlinfit');
```

```
opts.RobustWgtFun = 'bisquare';
```

Fit the nonlinear model using the robust fitting options. Here, use a character vector to specify the model.

```
b0 = [2;2;2];
```

```
modelstr = 'y ~ b1 + b2*exp(-b3*x)';
```

```
mdl = NonLinearModel.fit(x,y,modelstr,b0,'Options',opts)
```

```
mdl =
```

```
Nonlinear regression model (robust fit):
```

```
  y ~ b1 + b2*exp( - b3*x)
```

```
Estimated Coefficients:
```

	Estimate	SE	tStat	pValue
b1	1.0218	0.07202	14.188	2.1344e-25
b2	3.6619	0.25429	14.401	7.974e-26
b3	2.9732	0.38496	7.7232	1.0346e-11

```
Number of observations: 100, Error degrees of freedom: 97
```

```
Root Mean Squared Error: 0.501
```

```
R-Squared: 0.807, Adjusted R-Squared 0.803
```

```
F-statistic vs. constant model: 203, p-value = 2.34e-35
```

### Fit Nonlinear Regression Model Using Weights Function Handle

Load sample data.

```
S = load('reaction');  
X = S.reactants;  
y = S.rate;  
beta0 = S.beta;
```

Specify a function handle for observation weights. The function accepts the model fitted values as input, and returns a vector of weights.

```
a = 1; b = 1;  
weights = @(yhat) 1./((a + b*abs(yhat)).^2);
```

Fit the Hougen-Watson model to the rate data using the specified observation weights function.

```
mdl = NonLinearModel.fit(X,y,@hougen,beta0,'Weights',weights)
```

```
mdl =
```

```
Nonlinear regression model:
  y ~ hougen(b,X)
```

```
Estimated Coefficients:
              Estimate          SE          tStat          pValue
-----
b1      0.83085      0.58224      1.427      0.19142
b2      0.04095      0.029663     1.3805     0.20477
b3      0.025063     0.019673     1.274     0.23842
b4      0.080053     0.057812     1.3847     0.20353
b5      1.8261      1.281      1.4256     0.19183
```

```
Number of observations: 13, Error degrees of freedom: 8
Root Mean Squared Error: 0.037
R-Squared: 0.998, Adjusted R-Squared 0.998
F-statistic vs. zero model: 1.14e+03, p-value = 3.49e-11
```

## Nonlinear Regression Model Using Nonconstant Error Model

Load sample data.

```
S = load('reaction');
X = S.reactants;
y = S.rate;
beta0 = S.beta;
```

Fit the Hougen-Watson model to the rate data using the combined error variance model.

```
mdl = NonLinearModel.fit(X,y,@hougen,beta0,'ErrorModel','combined')
```

```
mdl =
```

```
Nonlinear regression model:
  y ~ hougen(b,X)
```

Estimated Coefficients:				
	Estimate	SE	tStat	pValue
	_____	_____	_____	_____
b1	1.2526	0.86702	1.4447	0.18654
b2	0.062776	0.043561	1.4411	0.18753
b3	0.040048	0.030885	1.2967	0.23089
b4	0.11242	0.075158	1.4957	0.17309
b5	1.1914	0.83671	1.4239	0.1923

Number of observations: 13, Error degrees of freedom: 8  
 Root Mean Squared Error: 1.27  
 R-Squared: 0.999, Adjusted R-Squared 0.998  
 F-statistic vs. zero model: 3.91e+03, p-value = 2.54e-13

- “Examine Quality and Adjust the Fitted Nonlinear Model” on page 13-7
- “Predict or Simulate Responses Using a Nonlinear Model” on page 13-10
- “Nonlinear Regression Workflow” on page 13-15

## Definitions

### Robust Options

Weight Function	Equation	Default Tuning Constant
'andrews'	$w = (\text{abs}(r) < \pi) .* \sin(r) ./ r$	1.339
'bisquare' (default)	$w = (\text{abs}(r) < 1) .* (1 - r.^2).^2$	4.685
'cauchy'	$w = 1 ./ (1 + r.^2)$	2.385
'fair'	$w = 1 ./ (1 + \text{abs}(r))$	1.400
'huber'	$w = 1 ./ \max(1, \text{abs}(r))$	1.345
'logistic'	$w = \tanh(r) ./ r$	1.205
'talwar'	$w = 1 * (\text{abs}(r) < 1)$	2.795
'welsch'	$w = \exp(-r.^2)$	2.985
[]	No robust fitting	—

## Algorithms

`NonLinearModel.fit` uses the same fitting algorithm as `nlinfit`.

## Alternatives

You can also construct a nonlinear model using `fitnlm`.

## References

- [1] Seber, G. A. F., and C. J. Wild. *Nonlinear Regression*. Hoboken, NJ: Wiley-Interscience, 2003.
- [2] DuMouchel, W. H., and F. L. O'Brien. "Integrating a Robust Option into a Multiple Regression Computing Environment." *Computer Science and Statistics: Proceedings of the 21st Symposium on the Interface*. Alexandria, VA: American Statistical Association, 1989.
- [3] Holland, P. W., and R. E. Welsch. "Robust Regression Using Iteratively Reweighted Least-Squares." *Communications in Statistics: Theory and Methods, A6*, 1977, pp. 813–827.

## See Also

`NonLinearModel` | `nlinfit`

## Topics

- "Examine Quality and Adjust the Fitted Nonlinear Model" on page 13-7
- "Predict or Simulate Responses Using a Nonlinear Model" on page 13-10
- "Nonlinear Regression Workflow" on page 13-15
- "Nonlinear Regression" on page 13-2

## RegressionTree.fit

**Class:** RegressionTree

Binary decision tree for regression (to be removed)

---

**Note** `RegressionTree.fit` will be removed in a future release. Use `fitrtree` instead.

---

### Syntax

```
tree = RegressionTree.fit(x, y)
tree = RegressionTree.fit(x, y, Name, Value)
```

### Description

`tree = RegressionTree.fit(x, y)` returns a regression tree based on the input variables (also known as predictors, features, or attributes) `x` and output (response) `y`. `tree` is a binary tree where each branching node is split based on the values of a column of `x`.

`tree = RegressionTree.fit(x, y, Name, Value)` fits a tree with additional options specified by one or more `Name, Value` pair arguments. You can specify several name-value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Note that using the `'CrossVal'`, `'KFold'`, `'Holdout'`, `'Leaveout'`, or `'CVPartition'` options results in a tree of class `RegressionPartitionedModel`. You cannot use a partitioned tree for prediction, so this kind of tree does not have a `predict` method.

Otherwise, `tree` is of class `RegressionTree`, and you can use the `predict` method to make predictions.



## Input Arguments

### **x** — Predictor values

matrix of scalar values

Predictor values, specified as a matrix of scalar values. Each column of `x` represents one variable, and each row represents one observation.

`RegressionTree.fit` considers NaN values in `x` as missing values.

`RegressionTree.fit` does not use observations with all missing values for `x` the fit.

`RegressionTree.fit` uses observations with some missing values for `x` to find splits on variables for which these observations have valid values.

Data Types: `single` | `double`

### **y** — Response values

vector of scalar values

Response values, specified as a vector of scalar values with the same number of rows as `x`. Each entry in `y` is the response to the data in the corresponding row of `x`.

`RegressionTree.fit` considers NaN values in `y` to be missing values.

`RegressionTree.fit` does not use observations with missing values for `y` in the fit.

Data Types: `single` | `double`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### **CategoricalPredictors** — Categorical predictors list

numeric or logical vector | cell array of character vectors | character matrix | 'all'

Categorical predictors list, specified as the comma-separated pair consisting of 'CategoricalPredictors' and one of the following.

- A numeric vector with indices from 1 to `p`, where `p` is the number of columns of `x`.

- A logical vector of length `p`, where a `true` entry means that the corresponding column of `x` is a categorical variable.
- A cell array of character vectors, where each element in the array is the name of a predictor variable. The names must match entries in the `PredictorNames` property.
- A character matrix, where each row of the matrix is a name of a predictor variable. Pad the names with extra blanks so each row of the character matrix has the same length.
- `'all'`, meaning all predictors are categorical.

Example:

```
Data Types: single | double | logical | char | struct | cell
```

### **CrossVal** — Cross-validation flag

```
'off' (default) | 'on'
```

Cross-validation flag, specified as the comma-separated pair consisting of `'CrossVal'` and either `'on'` or `'off'`.

If `'on'`, `RegressionTree.fit` grows a cross-validated decision tree with 10 folds. You can override this cross-validation setting using one of the `'KFold'`, `'Holdout'`, `'Leaveout'`, or `'CVPartition'` name-value pair arguments. Note that you can only use one of these four options (`'KFold'`, `'Holdout'`, `'Leaveout'`, or `'CVPartition'`) at a time when creating a cross-validated tree.

Alternatively, cross-validate `tree` later using the `crossval` method.

Example: `'CrossVal', 'on'`

### **CVPartition** — Partition for cross-validation tree

```
cvpartition object
```

Partition for cross-validated tree, specified as the comma-separated pair consisting of `'CVPartition'` and an object of the `cvpartition` class created using `cvpartition`.

Note that if you use `'CVPartition'`, you cannot use any of the `'KFold'`, `'Holdout'`, or `'Leaveout'` name-value pair arguments.

Example:

### **Holdout** — Fraction of data for holdout validation

```
scalar value in the range [0,1]
```

Fraction of data used for holdout validation, specified as the comma-separated pair consisting of 'Holdout' and a scalar value in the range  $[0, 1]$ . Holdout validation tests the specified fraction of the data, and uses the rest of the data for training.

Note that if you use 'Holdout', you cannot use any of the 'CVPartition', 'KFold', or 'Leaveout' name-value pair arguments.

Example: 'Holdout', 0.1

Data Types: single | double

### **KFold** — Number of folds

10 (default) | positive integer value

Number of folds to use in a cross-validated tree, specified as the comma-separated pair consisting of 'KFold' and a positive integer value.

Note that if you use 'KFold', you cannot use any of the 'CVPartition', 'Holdout', or 'Leaveout' name-value pair arguments.

Example: 'KFold', 8

Data Types: single | double

### **Leaveout** — Leave-one-out cross-validation flag

'off' (default) | 'on'

Leave-one-out cross-validation flag, specified as the comma-separated pair consisting of 'Leaveout' and either 'on' or 'off'. Use leave-one-out cross validation by setting to 'on'.

Note that if you use 'Leaveout', you cannot use any of the 'CVPartition', 'Holdout', or 'KFold' name-value pair arguments.

Example: 'Leaveout', 'on'

### **MergeLeaves** — Leaf merge flag

'on' (default) | 'off'

Leaf merge flag, specified as the comma-separated pair consisting of 'MergeLeaves' and either 'on' or 'off'. When 'on', RegressionTree.fit merges leaves that originate from the same parent node, and that give a sum of risk values greater or equal

to the risk associated with the parent node. When 'off', `RegressionTree.fit` does not merge leaves.

Example: `'MergeLeaves', 'off'`

**MinLeaf** — Minimum number of leaf node observations

1 (default) | positive integer value

Minimum number of leaf node observations, specified as the comma-separated pair consisting of 'MinLeaf' and a positive integer value. Each leaf has at least `MinLeaf` observations per tree leaf. If you supply both `MinParent` and `MinLeaf`, `RegressionTree.fit` uses the setting that gives larger leaves: `MinParent=max(MinParent,2*MinLeaf)`.

Example: `'MinLeaf',3`

Data Types: `single` | `double`

**MinParent** — Minimum number of branch node observations

10 (default) | positive integer value

Minimum number of branch node observations, specified as the comma-separated pair consisting of 'MinParent' and a positive integer value. Each branch node in the tree has at least `MinParent` observations. If you supply both `MinParent` and `MinLeaf`, `RegressionTree.fit` uses the setting that gives larger leaves: `MinParent=max(MinParent,2*MinLeaf)`.

Example: `'MinParent',8`

Data Types: `single` | `double`

**NVarToSample** — Number of predictors for split

'all' (default) | positive integer value

Number of predictors to select at random for each split, specified as the comma-separated pair consisting of 'NVarToSample' and a positive integer value. You can also specify 'all' to use all available predictors.

Example: `'NVarToSample',3`

Data Types: `single` | `double`

**PredictorNames** — Predictor variable names

{ 'x1', 'x2', ... } (default) | cell array of character vectors

Predictor variable names, specified as the comma-separated pair consisting of 'PredictorNames' and a cell array of character vectors containing the names for the predictor variables, in the order in which they appear in  $x$ .

Example:

Data Types: cell

### **Prune** — Pruning flag

'on' (default) | 'off'

Pruning flag, specified as the comma-separated pair consisting of 'Prune' and either 'on' or 'off'. When 'on', `RegressionTree.fit` computes the full tree and the optimal sequence of pruned subtrees. When 'off', `RegressionTree.fit` computes the full tree without pruning.

Example: 'Prune', 'off'

### **PruneCriterion** — Pruning criterion

'mse' (default)

Pruning criterion, specified as the comma-separated pair consisting of 'PruneCriterion' and 'mse'.

Example: 'PruneCriterion', 'mse'

### **QEToler** — Quadratic error tolerance

1e-6 (default) | positive scalar value

Quadratic error tolerance per node, specified as the comma-separated pair consisting of 'QEToler' and a positive scalar value. Splitting nodes stops when quadratic error per node drops below  $QEToler * QED$ , where  $QED$  is the quadratic error for the entire data computed before the decision tree is grown.

Example: 'QEToler', 1e4

### **ResponseName** — Response variable name

'Y' (default) | character vector

Response variable name, specified as the comma-separated pair consisting of 'ResponseName' and a character vector containing the name of the response variable in  $y$ .

Example: 'ResponseName', 'Response'

Data Types: char

### **ResponseTransform** — Response transform function

'none' (default) | function handle

Response transform function for transforming the raw response values, specified as the comma-separated pair consisting of 'ResponseTransform' and either a function handle or 'none'. The function handle should accept a matrix of response values and return a matrix of the same size. The default character vector 'none' means  $@(x) x$ , or no transformation.

Add or change a ResponseTransform function using dot notation:

```
tree.ResponseTransform = @function
```

Example:

Data Types: function\_handle

### **SplitCriterion** — Split criterion

'MSE' (default)

Split criterion, specified as the comma-separated pair consisting of 'SplitCriterion' and 'MSE', meaning mean squared error.

Example: 'SplitCriterion', 'MSE'

### **Surrogate** — Surrogate decision splits flag

'off' | 'on' | 'all' | positive integer value

Surrogate decision splits flag, specified as the comma-separated pair consisting of 'Surrogate' and 'on', 'off', 'all', or a positive integer value.

- When 'on', `RegressionTree.fit` finds at most 10 surrogate splits at each branch node.
- When set to a positive integer value, `RegressionTree.fit` finds at most the specified number of surrogate splits at each branch node.
- When set to 'all', `RegressionTree.fit` finds all surrogate splits at each branch node. The 'all' setting can use much time and memory.

Use surrogate splits to improve the accuracy of predictions for data with missing values. The setting also enables you to compute measures of predictive association between predictors.

Example: `'Surrogate', 'on'`

Data Types: `single | double`

### **Weights — Observation weights**

`ones(size(X,1),1)` (default) | vector of scalar values

Observation weights, specified as the comma-separated pair consisting of `'Weights'` and a vector of scalar values. The length of `Weights` is the number of rows in `x`.

Example:

Data Types: `single | double`

## Output Arguments

### **tree — Regression tree**

regression tree object

Regression tree, returned as a regression tree object. Note that using the `'Crossval'`, `'KFold'`, `'Holdout'`, `'Leaveout'`, or `'CVPartition'` options results in a tree of class `RegressionPartitionedModel`. You cannot use a partitioned tree for prediction, so this kind of tree does not have a `predict` method.

Otherwise, `tree` is of class `RegressionTree`, and you can use the `predict` method to make predictions.

## Examples

### **Predict Values Using a Regression Tree**

Load the sample data.

```
load carsmall
```

Construct a regression tree to predict the mileage of cars based on their weights and numbers of cylinders.

```
tree = RegressionTree.fit([Weight, Cylinders],MPG,...
                          'MinParent',20,...
                          'PredictorNames',{'W','C'})
```

```
tree =
  RegressionTree
    PredictorNames: {'W' 'C'}
    ResponseName: 'Y'
    CategoricalPredictors: []
    ResponseTransform: 'none'
    NumObservations: 94
```

Predict the mileage of a car that weighs 2200 lbs and has four cylinders.

```
predict(tree,[2200,4])
```

```
ans =
    29.6111
```

## References

[1] Breiman, L., J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Boca Raton, FL: CRC Press, 1984.

## See Also

[fitrtree](#) | [predict](#) | [view](#)

## Topics

“Splitting Categorical Predictors” on page 19-42



## fitcdiscr

Fit discriminant analysis classifier

### Syntax

```
Mdl = fitcdiscr(Tbl,ResponseVarName)
```

```
Mdl = fitcdiscr(Tbl,formula)
```

```
Mdl = fitcdiscr(Tbl,Y)
```

```
Mdl = fitcdiscr(X,Y)
```

```
Mdl = fitcdiscr( ____,Name,Value)
```

### Description

`Mdl = fitcdiscr(Tbl,ResponseVarName)` returns a fitted discriminant analysis model based on the input variables (also known as predictors, features, or attributes) contained in the table `Tbl` and output (response or labels) contained in `ResponseVarName`.

`Mdl = fitcdiscr(Tbl,formula)` returns a fitted discriminant analysis model based on the input variables contained in the table `Tbl`. `formula` is an explanatory model of the response and a subset of predictor variables in `Tbl` used to fit `Mdl`.

`Mdl = fitcdiscr(Tbl,Y)` returns a fitted discriminant analysis model based on the input variables contained in the table `Tbl` and response `Y`.

`Mdl = fitcdiscr(X,Y)` returns a discriminant analysis classifier based on the input variables `X` and response `Y`.

`Mdl = fitcdiscr( ____,Name,Value)` fits a classifier with additional options specified by one or more name-value pair arguments, using any of the previous syntaxes. For example, you can optimize hyperparameters to minimize the model's cross-validation loss, or specify the cost of misclassification, the prior probabilities for each class, or the observation weights.

## Examples

### Train Discriminant Analysis Model

Load Fisher's iris data set.

```
load fisheriris
```

Train a discriminant analysis model using the entire data set.

```
Mdl = fitcdiscr(meas, species)
```

```
Mdl =
```

```
ClassificationDiscriminant
    ResponseName: 'Y'
  CategoricalPredictors: []
        ClassNames: {'setosa' 'versicolor' 'virginica'}
    ScoreTransform: 'none'
  NumObservations: 150
    DiscrimType: 'linear'
           Mu: [3x4 double]
        Coeffs: [3x3 struct]
```

`Mdl` is a `ClassificationDiscriminant` model. To access its properties, use dot notation. For example, display the group means for each predictor.

```
Mdl.Mu
```

```
ans =
```

```
5.0060    3.4280    1.4620    0.2460
5.9360    2.7700    4.2600    1.3260
6.5880    2.9740    5.5520    2.0260
```

To predict labels for new observations, pass `Mdl` and predictor data to `predict`.

## Optimize Discriminant Analysis Model

This example shows how to optimize hyperparameters automatically using `fitcdiscr`. The example uses Fisher's iris data.

Load the data.

```
load fisheriris
```

Find hyperparameters that minimize five-fold cross-validation loss by using automatic hyperparameter optimization.

For reproducibility, set the random seed and use the 'expected-improvement-plus' acquisition function.

```
rng(1)
Mdl = fitcdiscr(meas,species,'OptimizeHyperparameters','auto',...
    'HyperparameterOptimizationOptions',...
    struct('AcquisitionFunctionName','expected-improvement-plus'))
```

```
=====
| Iter | Eval  | Objective | Objective | BestSoFar | BestSoFar |      Delta
|      | result|           | runtime   | (observed) | (estim.)  |
|=====|=====|=====|=====|=====|=====|=====
|  1  | Best  |  0.66667 |  1.2498  |  0.66667  |  0.66667  |  13.261
|  2  | Best  |  0.02    |  0.5271  |  0.02     |  0.064227 |  2.7404e-05
|  3  | Accept|  0.04    |  0.4912  |  0.02     |  0.020084 |  3.2455e-06
|  4  | Accept|  0.66667 |  0.28689 |  0.02     |  0.020118 |  14.879
|  5  | Accept|  0.046667|  0.33569 |  0.02     |  0.019907 |  0.00031449
|  6  | Accept|  0.04    |  0.26072 |  0.02     |  0.028438 |  4.5092e-05
|  7  | Accept|  0.046667|  0.24172 |  0.02     |  0.031424 |  2.0973e-05
|  8  | Accept|  0.02    |  0.24976 |  0.02     |  0.022424 |  1.0554e-06
|  9  | Accept|  0.02    |  0.2436  |  0.02     |  0.021105 |  1.1232e-06
| 10  | Accept|  0.02    |  0.21579 |  0.02     |  0.020948 |  0.00011837
| 11  | Accept|  0.02    |  0.25163 |  0.02     |  0.020172 |  1.0292e-06
| 12  | Accept|  0.02    |  0.28238 |  0.02     |  0.020105 |  9.7792e-05
| 13  | Accept|  0.02    |  0.24057 |  0.02     |  0.020038 |  0.00036014
| 14  | Accept|  0.02    |  0.23928 |  0.02     |  0.019597 |  0.00021059
| 15  | Accept|  0.02    |  0.24841 |  0.02     |  0.019461 |  1.1911e-05
| 16  | Accept|  0.02    |  0.25794 |  0.02     |  0.01993  |  0.0017896
| 17  | Accept|  0.02    |  0.22344 |  0.02     |  0.019551 |  0.00073745
| 18  | Accept|  0.02    |  0.25561 |  0.02     |  0.019776 |  0.00079304
| 19  | Accept|  0.02    |  0.21702 |  0.02     |  0.019678 |  0.007292
| 20  | Accept|  0.046667|  0.21343 |  0.02     |  0.019785 |  0.0074408
```

```

=====
| Iter | Eval | Objective | Objective | BestSoFar | BestSoFar | Delta
|   | result |   | runtime | (observed) | (estim.) |
=====
| 21 | Accept | 0.02 | 0.2104 | 0.02 | 0.019043 | 0.0036004
| 22 | Accept | 0.02 | 0.22595 | 0.02 | 0.019755 | 2.5238e-05
| 23 | Accept | 0.02 | 0.22557 | 0.02 | 0.0191 | 1.5478e-05
| 24 | Accept | 0.02 | 0.22469 | 0.02 | 0.019081 | 0.0040557
| 25 | Accept | 0.02 | 0.24789 | 0.02 | 0.019333 | 2.959e-05
| 26 | Accept | 0.02 | 0.24593 | 0.02 | 0.019369 | 2.3111e-06
| 27 | Accept | 0.02 | 0.21678 | 0.02 | 0.019455 | 3.8898e-05
| 28 | Accept | 0.02 | 0.26941 | 0.02 | 0.019449 | 0.0035925
| 29 | Accept | 0.66667 | 0.24639 | 0.02 | 0.019479 | 998.93
| 30 | Accept | 0.02 | 0.22971 | 0.02 | 0.01947 | 8.1557e-06
=====

```

---

Optimization completed.  
MaxObjectiveEvaluations of 30 reached.  
Total function evaluations: 30  
Total elapsed time: 72.8275 seconds.  
Total objective function evaluation time: 8.8747

Best observed feasible point:

Delta	Gamma
2.7404e-05	0.073264

Observed objective function value = 0.02  
Estimated objective function value = 0.01947  
Function evaluation time = 0.5271

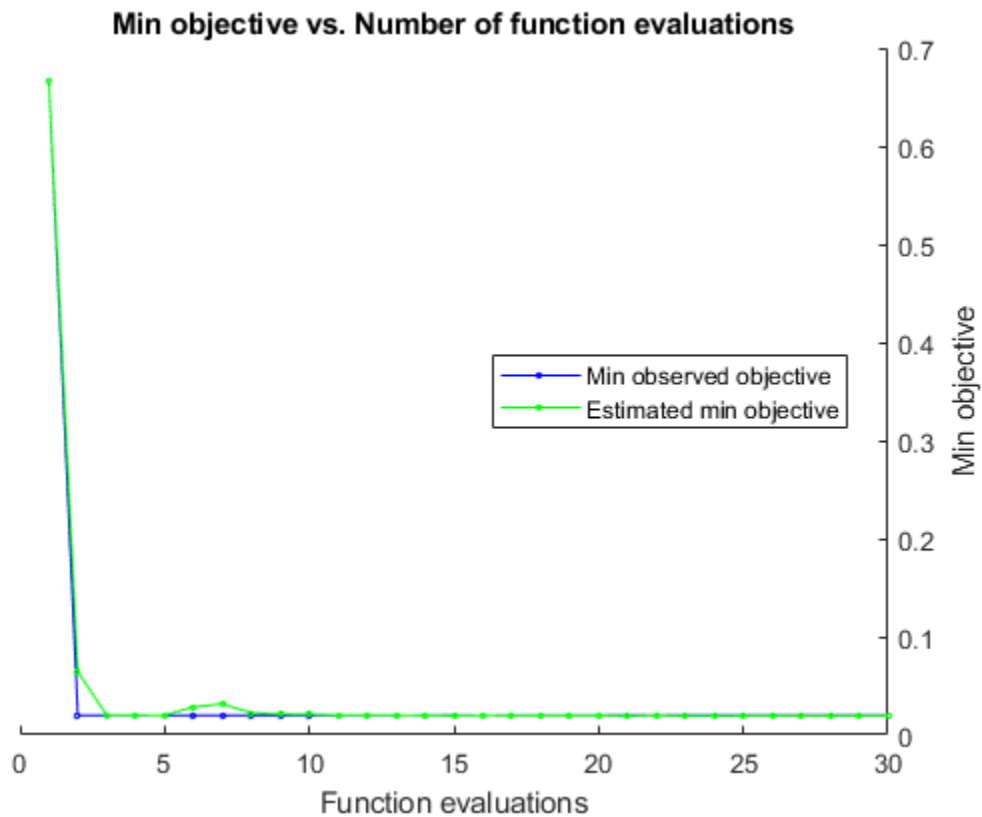
Best estimated feasible point (according to models):

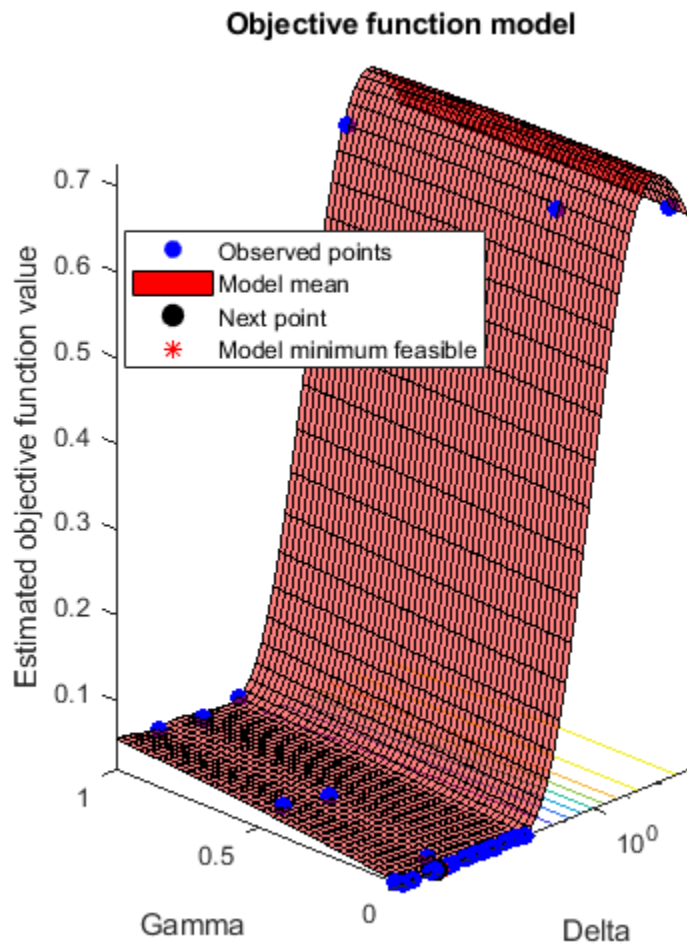
Delta	Gamma
2.5238e-05	0.0015542

Estimated objective function value = 0.01947  
Estimated function evaluation time = 0.23961

Mdl =

```
ClassificationDiscriminant
  ResponseName: 'Y'
  CategoricalPredictors: []
  ClassNames: {'setosa' 'versicolor' 'virginica'}
  ScoreTransform: 'none'
  NumObservations: 150
  HyperparameterOptimizationResults: [1x1 BayesianOptimization]
  DiscrimType: 'linear'
  Mu: [3x4 double]
  Coeffs: [3x3 struct]
```





The fit achieved about 2% loss for the default 5-fold cross validation.

## Input Arguments

**tbl** — Sample data  
table

Sample data used to train the model, specified as a table. Each row of `Tbl` corresponds to one observation, and each column corresponds to one predictor variable. Optionally, `Tbl` can contain one additional column for the response variable. Multi-column variables and cell arrays other than cell arrays of character vectors are not allowed.

If `Tbl` contains the response variable, and you want to use all remaining variables in `Tbl` as predictors, then specify the response variable using `ResponseVarName`.

If `Tbl` contains the response variable, and you want to use only a subset of the remaining variables in `Tbl` as predictors, then specify a formula using `formula`.

If `Tbl` does not contain the response variable, then specify a response variable using `Y`. The length of response variable and the number of rows of `Tbl` must be equal.

Data Types: `table`

#### **`ResponseVarName` — Response variable name**

name of variable in `Tbl`

Response variable name, specified as the name of a variable in `Tbl`.

You must specify `ResponseVarName` as a character vector. For example, if the response variable `Y` is stored as `Tbl.Y`, then specify it as `'Y'`. Otherwise, the software treats all columns of `Tbl`, including `Y`, as predictors when training the model.

The response variable must be a categorical or character array, logical or numeric vector, or cell array of character vectors. If `Y` is a character array, then each element must correspond to one row of the array.

It is good practice to specify the order of the classes using the `ClassNames` name-value pair argument.

Data Types: `char`

#### **`formula` — Explanatory model of response and subset of predictor variables**

character vector

Explanatory model of the response and a subset of the predictor variables, specified as a character vector in the form of `'Y~X1+X2+X3'`. In this form, `Y` represents the response variable, and `X1`, `X2`, and `X3` represent the predictor variables. The variables must be variable names in `Tbl` (`Tbl.Properties.VariableNames`).

To specify a subset of variables in `Tbl` as predictors for training the model, use a formula. If you specify a formula, then the software does not use any variables in `Tbl` that do not appear in `formula`.

Data Types: `char`

### **y** — Class labels

numeric vector | categorical vector | logical vector | character array | cell array of character vectors

Class labels, specified as a categorical or character array, logical or numeric vector, or cell array of character vectors. Each row of `Y` represents the classification of the corresponding row of `X`.

The software considers `NaN`, `' '` (empty character vector), and `<undefined>` values in `Y` to be missing values. Consequently, the software does not train using observations with a missing response.

Data Types: `single` | `double` | `logical` | `char` | `cell`

### **x** — Predictor data

numeric matrix

Predictor values, specified as a numeric matrix. Each column of `X` represents one variable, and each row represents one observation.

`fitcdiscr` considers `NaN` values in `X` as missing values. `fitcdiscr` does not use observations with missing values for `X` in the fit.

Data Types: `single` | `double`

## **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

---

**Note** You cannot use any cross-validation name-value pair along with `OptimizeHyperparameters`. You can modify the cross-validation for



OptimizeHyperparameters only by using the HyperparameterOptimizationOptions name-value pair.

---

Example: 'DiscrimType', 'quadratic', 'SaveMemory', 'on' specifies a quadratic discriminant classifier and does not store the covariance matrix in the output object.

### Model Parameters

#### ClassNames — Names of classes to use for training

categorical array | cell array of character vectors | character array | logical vector | numeric vector

Names of classes to use for training, specified as the comma-separated pair consisting of 'ClassNames' and a categorical or character array, logical or numeric vector, or cell array of character vectors. ClassNames must be the same data type as Y.

If ClassNames is a character array, then each element must correspond to one *row* of the array.

Use ClassNames to:

- Order the classes during training.
- Specify the order of any input or output argument dimension that corresponds to the class order. For example, use ClassNames to specify the order of the dimensions of Cost or the column order of classification scores returned by predict.
- Select a subset of classes for training. For example, suppose that the set of all distinct class names in Y is {'a', 'b', 'c'}. To train the model using observations from classes 'a' and 'c' only, specify 'ClassNames', {'a', 'c'}.

The default is the set of all distinct class names in Y.

Example: 'ClassNames', {'b', 'g'}

Data Types: categorical | char | logical | single | double | cell

#### Cost — Cost of misclassification

square matrix | structure

Cost of misclassification of a point, specified as the comma-separated pair consisting of 'Cost' and one of the following:

- Square matrix, where `Cost(i,j)` is the cost of classifying a point into class `j` if its true class is `i` (i.e., the rows correspond to the true class and the columns correspond to the predicted class). To specify the class order for the corresponding rows and columns of `Cost`, additionally specify the `ClassNames` name-value pair argument.
- Structure `S` having two fields: `S.ClassNames` containing the group names as a variable of the same type as `Y`, and `S.ClassificationCosts` containing the cost matrix.

The default is `Cost(i,j)=1` if `i~=j`, and `Cost(i,j)=0` if `i=j`.

Data Types: `single` | `double` | `struct`

### **FillCoeffs** — Coeffs property flag

'on' | 'off'

`Coeffs` property flag, specified as the comma-separated pair consisting of 'FillCoeffs' and 'on' or 'off'. Setting the flag to 'on' populates the `Coeffs` property in the classifier object. This can be computationally intensive, especially when cross validating. The default is 'on', unless you specify a cross validation name-value pair, in which case the flag is set to 'off' by default.

Example: 'FillCoeffs', 'off'

### **PredictorNames** — Predictor variable names

cell array of unique character vectors

Predictor variable names, specified as the comma-separated pair consisting of 'PredictorNames' and a cell array of unique character vectors. The functionality of 'PredictorNames' depends on the way you supply the training data.

- If you supply `X` and `Y`, then you can use 'PredictorNames' to give the predictor variables in `X` names.
  - The order of the names in `PredictorNames` must correspond to the column order of `X`. That is, `PredictorNames{1}` is the name of `X(:,1)`, `PredictorNames{2}` is the name of `X(:,2)`, and so on. Also, `size(X,2)` and `numel(PredictorNames)` must be equal.
  - By default, `PredictorNames` is {'x1', 'x2', ...}.
- If you supply `Tbl`, then you can use 'PredictorNames' to choose which predictor variables to use in training. That is, `fitcdiscr` uses the predictor variables in `PredictorNames` and the response only in training.

- `PredictorNames` must be a subset of `Tbl.Properties.VariableNames` and cannot include the name of the response variable.
- By default, `PredictorNames` contains the names of all predictor variables.
- It is good practice to specify the predictors for training using one of `'PredictorNames'` or `formula` only.

```
Example: 'PredictorNames',
{'SepalLength', 'SepalWidth', 'PetalLength', 'PetalWidth'}
```

Data Types: cell

### **Prior — Prior probabilities**

'empirical' (default) | 'uniform' | vector of scalar values | structure

Prior probabilities for each class, specified as the comma-separated pair consisting of `'Prior'` and a value in this table.

Value	Description
'empirical'	The class prior probabilities are the class relative frequencies in $Y$ .
'uniform'	All class prior probabilities are equal to $1/K$ , where $K$ is the number of classes.
numeric vector	Each element is a class prior probability. Order the elements according to <code>Mdl.ClassNames</code> or specify the order using the <code>ClassNames</code> name-value pair argument. The software normalizes the elements such that they sum to 1.
structure	A structure $S$ with two fields: <ul style="list-style-type: none"> <li>• <code>S.ClassNames</code> contains the class names as a variable of the same type as <math>Y</math>.</li> <li>• <code>S.ClassProbs</code> contains a vector of corresponding prior probabilities. The software normalizes the elements such that they sum to 1.</li> </ul>

If you set values for both `Weights` and `Prior`, the weights are renormalized to add up to the value of the prior probability in the respective class.

Example: `'Prior', 'uniform'`

Data Types: `single` | `double` | `struct`

### **ResponseName** — Response variable name

`'Y'` (default) | character vector

Response variable name, specified as the comma-separated pair consisting of `'ResponseName'` and a character vector.

- If you supply `Y`, then you can use `'ResponseName'` to specify a name for the response variable.
- If you supply `ResponseVarName` or `formula`, then you cannot use `'ResponseName'`.

Example: `'ResponseName', 'response'`

Data Types: `char`

### **SaveMemory** — Flag to save covariance matrix

`'off'` (default) | `'on'`

Flag to save covariance matrix, specified as the comma-separated pair consisting of `'SaveMemory'` and either `'on'` or `'off'`. If you specify `'on'`, then `fitcdiscr` does not store the full covariance matrix, but instead stores enough information to compute the matrix. The `predict` method computes the full covariance matrix for prediction, and does not store the matrix. If you specify `'off'`, then `fitcdiscr` computes and stores the full covariance matrix in `Mdl`.

Specify `SaveMemory` as `'on'` when the input matrix contains thousands of predictors.

Example: `'SaveMemory', 'on'`

### **ScoreTransform** — Score transformation

`'none'` (default) | character vector | function handle

Score transformation, specified as the comma-separated pair consisting of `'ScoreTransform'` and either a character vector or a function handle.

This table summarizes the available character vectors.

Value	Description
'doublelogit'	$1/(1 + e^{-2x})$
'invlogit'	$\log(x / (1-x))$
'ismax'	Set the score for the class with the largest score to 1, and set the scores for all other classes to 0.
'logit'	$1/(1 + e^{-x})$
'none' or 'identity'	$x$ (no transformation)
'sign'	-1 for $x < 0$ 0 for $x = 0$ 1 for $x > 0$
'symmetric'	$2x - 1$
'symmetricismax'	Set the score for the class with the largest score to 1, and set the scores for all other classes to -1.
'symmetriclogit'	$2/(1 + e^{-x}) - 1$

For a MATLAB function or a function you define, use its function handle. The function handle must accept a matrix (the original scores) and return a matrix of the same size (the transformed scores).

Example: 'ScoreTransform', 'logit'

Data Types: char | function\_handle

### Weights — Observation weights

numeric vector of positive values | character vector

Observation weights, specified as the comma-separated pair consisting of 'Weights' and a numeric vector of positive values or name of a variable in Tbl. The software weighs the observations in each row of X or Tbl with the corresponding value in Weights. The size of Weights must equal the number of rows of X or Tbl.

If you specify the input data as a table Tbl, then Weights can be the name of a variable in Tbl that contains a numeric vector. In this case, you must specify Weights as a character vector. For example, if the weights vector W is stored as Tbl.W, then specify it as 'W'. Otherwise, the software treats all columns of Tbl, including W, as predictors or the response when training the model.

The software normalizes `Weights` to sum up to the value of the prior probability in the respective class.

By default, `Weights` is ones ( $n, 1$ ), where  $n$  is the number of observations in `X` or `Tbl`.

Data Types: `double` | `single` | `char`

### **Cross-Validation**

#### **CrossVal** — Cross-validation flag

`'off'` (default) | `'on'`

Cross-validation flag, specified as the comma-separated pair consisting of `'Crossval'` and `'on'` or `'off'`.

If you specify `'on'`, then the software implements 10-fold cross-validation.

To override this cross-validation setting, use one of these name-value pair arguments: `CVPartition`, `Holdout`, `KFold`, or `Leaveout`. To create a cross-validated model, you can use one cross-validation name-value pair argument at a time only.

Alternatively, cross validate later by passing `Mdl` to `crossval`.

Example: `'CrossVal', 'on'`

#### **CVPartition** — Cross-validation partition

`[]` (default) | `cvpartition` partition object

Cross-validation partition, specified as the comma-separated pair consisting of `'CVPartition'` and a `cvpartition` partition object as created by `cvpartition`. The partition object specifies the type of cross-validation, and also the indexing for training and validation sets.

To create a cross-validated model, you can use one of these four name-value pair arguments only: `CVPartition`, `Holdout`, `KFold`, or `Leaveout`.

Example: Suppose you create a random partition for 5-fold cross-validation on 500 observations by using `cvp = cvpartition(500, 'KFold', 5)`. Then, you can specify the cross-validation partition for `fitcecoc` as `'CVPartition', cvp`.

#### **Holdout** — Fraction of data for holdout validation

scalar value in the range (0,1)

Fraction of data used for holdout validation, specified as the comma-separated pair consisting of 'Holdout' and a scalar value in the range (0,1). If you specify 'Holdout',  $p$ , then the software:

- 1 Randomly reserves  $p*100\%$  of the data as validation data, and trains the model using the rest of the data
- 2 Stores the compact, trained model in the `Trained` property of the cross-validated model.

To create a cross-validated model, you can use one of these four name-value pair arguments only: `CVPartition`, `Holdout`, `KFold`, or `Leaveout`.

Example: 'Holdout', 0.1

Data Types: double | single

### **KFold** — Number of folds

10 (default) | positive integer value greater than 1

Number of folds to use in a cross-validated model, specified as the comma-separated pair consisting of 'KFold' and a positive integer value greater than 1. If you specify, e.g., 'KFold',  $k$ , then the software:

- 1 Randomly partitions the data into  $k$  sets
- 2 For each set, reserves the set as validation data, and trains the model using the other  $k - 1$  sets
- 3 Stores the  $k$  compact, trained models in the cells of a  $k$ -by-1 cell vector in the `Trained` property of the cross-validated model.

To create a cross-validated model, you can use one of these four name-value pair arguments only: `CVPartition`, `Holdout`, `KFold`, or `Leaveout`.

Example: 'KFold', 5

Data Types: single | double

### **Leaveout** — Leave-one-out cross-validation flag

'off' (default) | 'on'

Leave-one-out cross-validation flag, specified as the comma-separated pair consisting of 'Leaveout' and 'on' or 'off'. If you specify 'Leaveout', 'on', then, for each of the  $n$  observations, where  $n$  is `size(Mdl.X, 1)`, the software:

- 1 Reserves the observation as validation data, and trains the model using the other  $n - 1$  observations
- 2 Stores the  $n$  compact, trained models in the cells of an  $n$ -by-1 cell vector in the `Trained` property of the cross-validated model.

To create a cross-validated model, you can use one of these four name-value pair arguments only: `CVPartition`, `Holdout`, `KFold`, or `Leaveout`.

Example: `'Leaveout', 'on'`

Data Types: `char`

### Hyperparameter Optimization

#### **Delta** — Linear coefficient threshold

0 (default) | nonnegative scalar value

Linear coefficient threshold, specified as the comma-separated pair consisting of `'Delta'` and a nonnegative scalar value. If a coefficient of `Mdl` has magnitude smaller than `Delta`, `Mdl` sets this coefficient to 0, and you can eliminate the corresponding predictor from the model. Set `Delta` to a higher value to eliminate more predictors.

`Delta` must be 0 for quadratic discriminant models.

Data Types: `single` | `double`

#### **DiscrimType** — Discriminant type

`'linear'` (default) | `'quadratic'` | `'diaglinear'` | `'diagquadratic'` | `'pseudolinear'` | `'pseudoquadratic'`

Discriminant type, specified as the comma-separated pair consisting of `'DiscrimType'` and a character vector in this table.



Value	Description	Predictor Covariance Treatment
'linear'	Regularized linear discriminant analysis (LDA)	<ul style="list-style-type: none"> <li>All classes have the same covariance matrix.</li> <li><math>\hat{\Sigma}_\gamma = (1 - \gamma)\hat{\Sigma} + \gamma \text{diag}(\hat{\Sigma})</math>.</li> </ul> <p><math>\hat{\Sigma}</math> is the empirical, pooled covariance matrix and <math>\gamma</math> is the amount of regularization.</p>
'diaglinear'	LDA	All classes have the same, diagonal covariance matrix.
'pseudolinear'	LDA	All classes have the same covariance matrix. The software inverts the covariance matrix using the pseudo inverse.
'quadratic'	Quadratic discriminant analysis (QDA)	The covariance matrices can vary among classes.
'diagquadratic'	QDA	The covariance matrices are diagonal and can vary among classes.
'pseudoquadratic'	QDA	The covariance matrices can vary among classes. The software inverts the covariance matrix using the pseudo inverse.

---

**Note** To use regularization, you must specify 'linear'. To specify the amount of regularization, use the Gamma name-value pair argument.

---

Example: 'DiscrimType', 'quadratic'

**Gamma — Amount of regularization**

scalar value in the interval [0,1]

Amount of regularization to apply when estimating the covariance matrix of the predictors, specified as the comma-separated pair consisting of 'Gamma' and a scalar value in the interval [0,1]. Gamma provides finer control over the covariance matrix structure than `DiscrimType`.

- If you specify 0, then the software does not use regularization to adjust the covariance matrix. That is, the software estimates and uses the unrestricted, empirical covariance matrix.
  - For linear discriminant analysis, if the empirical covariance matrix is singular, then the software automatically applies the minimal regularization required to invert the covariance matrix. You can display the chosen regularization amount by entering `Mdl.Gamma` at the command line.
  - For quadratic discriminant analysis, if at least one class has an empirical covariance matrix that is singular, then the software throws an error.
- If you specify a value in the interval (0,1), then you must implement linear discriminant analysis, otherwise the software throws an error. Consequently, the software sets `DiscrimType` to 'linear'.
- If you specify 1, then the software uses maximum regularization for covariance matrix estimation. That is, the software restricts the covariance matrix to be diagonal. Alternatively, you can set `DiscrimType` to 'diagLinear' or 'diagQuadratic' for diagonal covariance matrices.

Example: `'Gamma',1`

Data Types: `single` | `double`

### Hyperparameter Optimization

#### **OptimizeHyperparameters** — Parameters to optimize

'none' (default) | 'auto' | 'all' | cell array of eligible parameter names | vector of `optimizableVariable` objects

Parameters to optimize, specified as:

- 'none' — Do not optimize.
- 'auto' — Use {'Delta', 'Gamma'}
- 'all' — Optimize all eligible parameters.
- Cell array of eligible parameter names

- Vector of `optimizableVariable` objects, typically the output of hyperparameters

The optimization attempts to minimize the cross-validation loss (error) for `fitcdiscr` by varying the parameters. For information about cross-validation loss (albeit in a different context), see “Classification Loss” on page 32-3250. To control the cross-validation type and other aspects of the optimization, use the `HyperparameterOptimizationOptions` name-value pair.

---

**Note** `OptimizeHyperparameters` values override any values you set using other name-value pairs. For example, setting `OptimizeHyperparameters` to 'auto' causes the 'auto' values to apply.

---

The eligible parameters for `fitcdiscr` are:

- `Delta` — `fitcdiscr` searches among positive values, by default log-scaled in the range `[1e-6, 1e3]`.
- `DiscrimType` — `fitcdiscr` searches among 'linear', 'quadratic', 'diagLinear', 'diagQuadratic', 'pseudoLinear', and 'pseudoQuadratic'.
- `Gamma` — `fitcdiscr` searches among real values in the range `[0, 1]`.

Set nondefault parameters by passing a vector of `optimizableVariable` objects that have nondefault values. For example,

```
load fisheriris
params = hyperparameters('fitcdiscr', meas, species);
params(1).Range = [1e-4, 1e6];
```

Pass `params` as the value of `OptimizeHyperparameters`.

By default, iterative display appears at the command line, and plots appear according to the number of hyperparameters in the optimization. For the optimization and plots, the objective function is  $\log(1 + \text{cross-validation loss})$  for regression, and the misclassification rate for classification. To control the iterative display, set the `HyperparameterOptimizationOptions` name-value pair, `Verbose` field. To control the plots, set the `HyperparameterOptimizationOptions` name-value pair, `ShowPlots` field.

For an example, see “Optimize Discriminant Analysis Model” on page 32-1844.

Example: 'auto'

Data Types: char | cell

### HyperparameterOptimizationOptions — Options for optimization structure

Options for optimization, specified as a structure. Modifies the effect of the OptimizeHyperparameters name-value pair. All fields in the structure are optional.

Field Name	Values	Default
Optimizer	<ul style="list-style-type: none"> <li>'bayesopt' — Use Bayesian optimization. Internally, this setting calls bayesopt.</li> <li>'gridsearch' — Use grid search with NumGridDivisions values per dimension.</li> <li>'randomsearch' — Search at random among MaxObjectiveEvaluations points.</li> </ul> <p>'gridsearch' searches in a random order, using uniform sampling without replacement from the grid. After optimization, you can get a table in grid order by using the command sortrows(Mdl.ParameterOptimizationResults).</p>	'bayesopt'

Field Name	Values	Default
AcquisitionFunctionName	<ul style="list-style-type: none"> <li>• 'expected-improvement-per-second-plus'</li> <li>• 'expected-improvement'</li> <li>• 'expected-improvement-plus'</li> <li>• 'expected-improvement-per-second'</li> <li>• 'lower-confidence-bound'</li> <li>• 'probability-of-improvement'</li> </ul> <p>For details, see the bayesopt AcquisitionFunctionName name-value pair, or “Acquisition Function Types” on page 10-4.</p>	'expected-improvement-per-second-plus'
MaxObjectiveEvaluations	Maximum number of objective function evaluations.	30 for 'bayesopt' or 'randomsearch', and the entire grid for 'gridsearch'
MaxTime	Time limit, specified as a positive real. The time limit is in seconds, as measured by tic and toc. Run time can exceed MaxTime because MaxTime does not interrupt function evaluations.	Inf
NumGridDivisions	For 'gridsearch', the number of values in each dimension. Can be a vector of positive integers giving the number of values for each dimension, or a scalar that applies to all dimensions. Ignored for categorical variables.	10

Field Name	Values	Default
ShowPlots	Logical value indicating whether to show plots. If <code>true</code> , plots the best objective function value against iteration number. If there are one or two optimization parameters, and if <code>Optimizer</code> is <code>'bayesopt'</code> , then <code>ShowPlots</code> also plots a model of the objective function against the parameters.	<code>true</code>
SaveIntermediateResults	Logical value indicating whether to save results when <code>Optimizer</code> is <code>'bayesopt'</code> . If <code>true</code> , overwrites a workspace variable named <code>'BayesoptResults'</code> at each iteration. The variable is a <code>BayesianOptimization</code> object.	<code>false</code>
Verbose	Display to the command line. <ul style="list-style-type: none"> <li>• 0 — No iterative display</li> <li>• 1 — Iterative display</li> <li>• 2 — Iterative display with extra information</li> </ul> For details, see the <code>bayesopt</code> <code>Verbose</code> name-value pair.	1
UseParallel	Logical value indicating whether to run Bayesian optimization in parallel, which requires Parallel Computing Toolbox. For details, see “Parallel Bayesian Optimization” on page 10-9.	<code>false</code>

Field Name	Values	Default
Repartition	Logical value indicating whether to repartition the cross-validation at every iteration. If <code>false</code> , the optimizer uses a single partition for the optimization.  true usually gives the most robust results because this setting takes partitioning noise into account. However, for good results, true requires at least twice as many function evaluations.	false
Use no more than one of the following three field names.		
CVPartition	A <code>cvpartition</code> object, as created by <code>cvpartition</code>	Kfold = 5
Holdout	A scalar in the range (0,1) representing the holdout fraction.	
Kfold	An integer greater than 1.	

Example: `struct('MaxObjectiveEvaluations',60)`

Data Types: `struct`

## Output Arguments

### **mdl** — Trained discriminant analysis classification model

`ClassificationDiscriminant` model object | `ClassificationPartitionedModel` cross-validated model object

Trained discriminant analysis classification model, returned as a `ClassificationDiscriminant` model object or a `ClassificationPartitionedModel` cross-validated model object.

If you set any of the name-value pair arguments `KFold`, `Holdout`, `CrossVal`, or `CVPartition`, then `Mdl` is a `ClassificationPartitionedModel` cross-validated model object. Otherwise, `Mdl` is a `ClassificationDiscriminant` model object.

To reference properties of `Mdl`, use dot notation. For example, to display the estimated component means at the Command Window, enter `Mdl.Mu`.

## Definitions

### Discriminant Classification

The model for discriminant analysis is:

- Each class ( $Y$ ) generates data ( $X$ ) using a multivariate normal distribution. That is, the model assumes  $X$  has a Gaussian mixture distribution (`gmdistribution`).
- For linear discriminant analysis, the model has the same covariance matrix for each class, only the means vary.
- For quadratic discriminant analysis, both means and covariances of each class vary.

`predict` classifies so as to minimize the expected classification cost:

$$\hat{y} = \arg \min_{y=1, \dots, K} \sum_{k=1}^K \hat{P}(k | x) C(y | k),$$

where

- $\hat{y}$  is the predicted classification.
- $K$  is the number of classes.
- $\hat{P}(k | x)$  is the posterior probability on page 20-8 of class  $k$  for observation  $x$ .
- $C(y | k)$  is the cost on page 20-9 of classifying an observation as  $y$  when its true class is  $k$ .

For details, see “Prediction Using Discriminant Analysis Models” on page 20-8.

## Tips

- After training a `ClassificationDiscriminant` model object by using `fitcdiscr`, use the function `predict` and the trained model object to generate C code that predicts labels for new data. For details, see “Code Generation” on page 32-0 .



---

## Alternative Functionality

### Functions

The `classify` function also performs discriminant analysis. `classify` is usually more awkward to use.

- `classify` requires you to fit the classifier every time you make a new prediction.
- `classify` does not perform cross validation or hyperparameter optimization.
- `classify` requires you to fit the classifier when changing prior probabilities.

## Extended Capabilities

### Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function supports tall arrays for out-of-memory data with some limitations.

- Supported name-value pairs are:
  - `'ClassNames'`
  - `'Cost'`
  - `'DiscrimType'`
  - `'PredictorNames'`
  - `'Prior'`
  - `'ResponseName'`
  - `'ScoreTransform'`
  - `'Weights'`
- For tall arrays and tall tables, `fitcdiscr` returns a `CompactClassificationDiscriminant` object, which contains most of the same properties as a `ClassificationDiscriminant` object. The main difference is that the compact object is sensitive to memory requirements. The compact object does not include properties that include the data, or that include an array of the same size as

the data. The compact object does not contain these `ClassificationDiscriminant` properties:

- `ModelParameters`
- `NumObservations`
- `ParameterOptimizationResults`
- `RowsUsed`
- `XCentered`
- `W`
- `X`
- `Y`

Additionally, the compact object does not support these `ClassificationDiscriminant` methods:

- `compact`
- `crossval`
- `cvshrink`
- `resubEdge`
- `resubLoss`
- `resubMargin`
- `resubPredict`

For more information, see “Tall Arrays” (MATLAB).

## See Also

`ClassificationDiscriminant` | `ClassificationPartitionedModel` | `classify`  
| `crossval` | `predict`

## Topics

“Discriminant Analysis Classification” on page 20-2

**Introduced in R2014a**

## fitcecoc

Fit multiclass models for support vector machines or other classifiers

### Syntax

```
Mdl = fitcecoc(Tbl, ResponseVarName)
```

```
Mdl = fitcecoc(Tbl, formula)
```

```
Mdl = fitcecoc(Tbl, Y)
```

```
Mdl = fitcecoc(X, Y)
```

```
Mdl = fitcecoc( ____, Name, Value)
```

```
[Mdl, HyperparameterOptimizationResults] = fitcecoc( ____, Name, Value)
```

### Description

`Mdl = fitcecoc(Tbl, ResponseVarName)` returns a full, trained, multiclass, error-correcting output codes (ECOC) model on page 32-1919 using the predictors in table `Tbl` and the class labels in `Tbl.ResponseVarName`. `fitcecoc` uses  $K(K-1)/2$  binary support vector machine (SVM) models using the one-versus-one coding design on page 32-1915, where  $K$  is the number of unique class labels (levels). `Mdl` is a `ClassificationECOC` model.

`Mdl = fitcecoc(Tbl, formula)` returns an ECOC model using the predictors in table `Tbl` and the class labels. `formula` is an explanatory model of the response and a subset of predictor variables in `Tbl` used for training.

`Mdl = fitcecoc(Tbl, Y)` returns an ECOC model using the predictors in table `Tbl` and the class labels in vector `Y`.

`Mdl = fitcecoc(X, Y)` returns a full, trained ECOC model using the predictors `X` and the class labels `Y`.

`Mdl = fitcecoc( ____, Name, Value)` returns an ECOC model with additional options specified by one or more `Name, Value` pair arguments, using any of the previous syntaxes.

For example, specify different binary learners, a different coding design, or to cross-validate. It is good practice to cross-validate using the `Kfold` `Name, Value` pair argument. The cross-validation results determine how well the model generalizes.

`[Mdl,HyperparameterOptimizationResults] = fitcecoc( ____,Name,Value)` also returns hyperparameter optimization details when you pass an `OptimizeHyperparameters` `name-value` pair with `Learners = 'linear'`. For other `Learners`, the `HyperparameterOptimizationResults` property of `Mdl` contains the results.

## Examples

### Train Multiclass Model Using SVM Learners

Train an error-correcting output codes (ECOC) multiclass model using support vector machine (SVM) binary learners.

Load Fisher's iris data set.

```
load fisheriris
X = meas;
Y = species;
```

Train an ECOC multiclass model using the default options.

```
Mdl = fitcecoc(X,Y)
```

```
Mdl =
```

```
ClassificationECOC
    ResponseName: 'Y'
    CategoricalPredictors: []
        ClassNames: {'setosa' 'versicolor' 'virginica'}
    ScoreTransform: 'none'
    BinaryLearners: {3x1 cell}
    CodingName: 'onevsone'
```

Mdl is a ClassificationECOC model. By default, fitcecoc uses SVM binary learners, and uses a one-versus-one coding design. You can access Mdl properties using dot notation.

Display the coding design matrix.

```
Mdl.ClassNames
CodingMat = Mdl.CodingMatrix
```

```
ans =
```

```
3x1 cell array

{'setosa'   }
{'versicolor'}
{'virginica' }
```

```
CodingMat =
```

```
 1     1     0
-1     0     1
 0    -1    -1
```

A one-versus-one coding design on three classes yields three binary learners. Columns of CodingMat correspond to learners, and rows correspond to classes. The class order corresponds to the order in Mdl.ClassNames. For example, CodingMat(:,1) is [1; -1; 0] and indicates that the software trains the first SVM binary learner using all observations classified as 'setosa' and 'versicolor'. Since 'setosa' corresponds to 1, it is the positive class, and since 'versicolor' corresponds to -1, it is the negative class.

You can access each binary learner using cell indexing and dot notation.

```
Mdl.BinaryLearners{1}           % The first binary learner
```

```
ans =
```

```
classreg.learning.classif.CompactClassificationSVM
    ResponseName: 'Y'
    CategoricalPredictors: []
```

```
      ClassNames: [-1 1]
    ScoreTransform: 'none'
           Beta: [4x1 double]
           Bias: 1.4505
KernelParameters: [1x1 struct]
```

Compute the in-sample classification error.

```
isLoss = resubLoss(Mdl)
```

```
isLoss =
    0.0067
```

The classification error is small, but the classifier might have been overfit. You can cross-validate the classifier using `crossval`.

### Cross-Validate ECOC Classifier

Load Fisher's iris data set.

```
load fisheriris
X = meas;
Y = species;
rng(1); % For reproducibility
```

Create an SVM template. It is good practice to standardize the predictors.

```
t = templateSVM('Standardize',1)
```

```
t =
```

Fit template for classification SVM.

```
      Alpha: [0x1 double]
BoxConstraint: []
      CacheSize: []
CachingMethod: ''
```

```

        ClipAlphas: []
DeltaGradientTolerance: []
        Epsilon: []
        GapTolerance: []
        KKTolerance: []
        IterationLimit: []
        KernelFunction: ''
        KernelScale: []
        KernelOffset: []
        KernelPolynomialOrder: []
        NumPrint: []
        Nu: []
        OutlierFraction: []
        RemoveDuplicates: []
        ShrinkagePeriod: []
        Solver: ''
        StandardizeData: 1
        SaveSupportVectors: []
        VerbosityLevel: []
        Version: 2
        Method: 'SVM'
        Type: 'classification'

```

`t` is an SVM template. All of its properties are empty, except for `StandardizeData`, `Method`, and `Type`. When the software trains the ECOC classifier, it sets the applicable properties to their default values.

Train the ECOC classifier. It is good practice to specify the class order.

```
Mdl = fitcecoc(X,Y,'Learners',t,...
    'ClassNames',{'setosa','versicolor','virginica'});
```

`Mdl` is a `ClassificationECOC` classifier. You can access its properties using dot notation.

Cross-validate `Mdl` using 10-fold cross-validation.

```
CVMDL = crossval(Mdl);
```

`CVMDL` is a `ClassificationPartitionedECOC` cross-validated ECOC classifier.

Estimate the generalization error.

```
oosLoss = kfoldLoss(CVMDL)
```

```
oosLoss =  
  
    0.0400
```

The out-of-sample classification error is 4%, which indicates that the ECOC classifier generalizes fairly well.

### Estimate Posterior Probabilities Using ECOC Classifiers

Load Fisher's iris data set. Train the classifier using the petal dimensions as predictors.

```
load fisheriris  
X = meas(:,3:4);  
Y = species;  
rng(1); % For reproducibility
```

Create an SVM template, and specify the Gaussian kernel. It is good practice to standardize the predictors.

```
t = templateSVM('Standardize',1,'KernelFunction','gaussian');
```

`t` is an SVM template. Most of its properties are empty. When the software trains the ECOC classifier, it sets the applicable properties to their default values.

Train the ECOC classifier using the SVM template. Transform classification scores to class posterior probabilities (which are returned by `predict` or `resubPredict`) using the 'FitPosterior' name-value pair argument. Display diagnostic messages during the training using the 'Verbose' name-value pair argument. It is good practice to specify the class order.

```
Mdl = fitcecoc(X,Y,'Learners',t,'FitPosterior',1,...  
    'ClassNames',{'setosa','versicolor','virginica'},...  
    'Verbose',2);
```

```
Training binary learner 1 (SVM) out of 3 with 50 negative and 50 positive observations.  
Negative class indices: 2  
Positive class indices: 1
```

```
Fitting posterior probabilities for learner 1 (SVM).  
Training binary learner 2 (SVM) out of 3 with 50 negative and 50 positive observations.
```



```
Negative class indices: 3
Positive class indices: 1
```

```
Fitting posterior probabilities for learner 2 (SVM).
Training binary learner 3 (SVM) out of 3 with 50 negative and 50 positive observations.
Negative class indices: 3
Positive class indices: 2
```

```
Fitting posterior probabilities for learner 3 (SVM).
```

Mdl is a ClassificationECOC model. The same SVM template applies to each binary learner, but you can adjust options for each binary learner by passing in a cell vector of templates.

Predict the in-sample labels and class posterior probabilities. Display diagnostic messages during the computation of labels and class posterior probabilities using the 'Verbose' name-value pair argument.

```
[label,~,~,Posterior] = resubPredict(Mdl,'Verbose',1);
Mdl.BinaryLoss
```

```
Predictions from all learners have been computed.
Loss for all observations has been computed.
Computing posterior probabilities...
```

```
ans =

    'quadratic'
```

The software assigns an observation to the class that yields the smallest average binary loss. Since all binary learners are computing posterior probabilities, the binary loss function is quadratic.

Display a random set of results.

```
idx = randsample(size(X,1),10,1);
Mdl.ClassNames
table(Y(idx),label(idx),Posterior(idx,:),...
    'VariableNames',{'TrueLabel','PredLabel','Posterior'})
```

```
ans =
```

```

3x1 cell array

    {'setosa'      }
    {'versicolor'}
    {'virginica'  }

ans =

10x3 table

    TrueLabel      PredLabel      Posterior
    _____    _____    _____
    'virginica'    'virginica'    0.0039321      0.0039869      0.99208
    'virginica'    'virginica'    0.017067       0.018263       0.96467
    'virginica'    'virginica'    0.014948       0.015856       0.9692
    'versicolor'  'versicolor'  2.2197e-14     0.87317        0.12683
    'setosa'       'setosa'       0.999          0.00025091     0.00074639
    'versicolor'  'virginica'    2.2195e-14     0.059429       0.94057
    'versicolor'  'versicolor'  2.2194e-14     0.97001        0.029986
    'setosa'       'setosa'       0.999          0.0002499      0.00074741
    'versicolor'  'versicolor'  0.0085646     0.98259        0.008849
    'setosa'       'setosa'       0.999          0.00025013     0.00074718

```

The columns of Posterior correspond to the class order of `Mdl.ClassNames`.

Define a grid of values in the observed predictor space. Predict the posterior probabilities for each instance in the grid.

```

xMax = max(X);
xMin = min(X);

x1Pts = linspace(xMin(1), xMax(1));
x2Pts = linspace(xMin(2), xMax(2));
[x1Grid, x2Grid] = meshgrid(x1Pts, x2Pts);

[~,~,~,PosteriorRegion] = predict(Mdl, [x1Grid(:), x2Grid(:)]);

```

For each coordinate on the grid, plot the maximum class posterior probability among all classes.

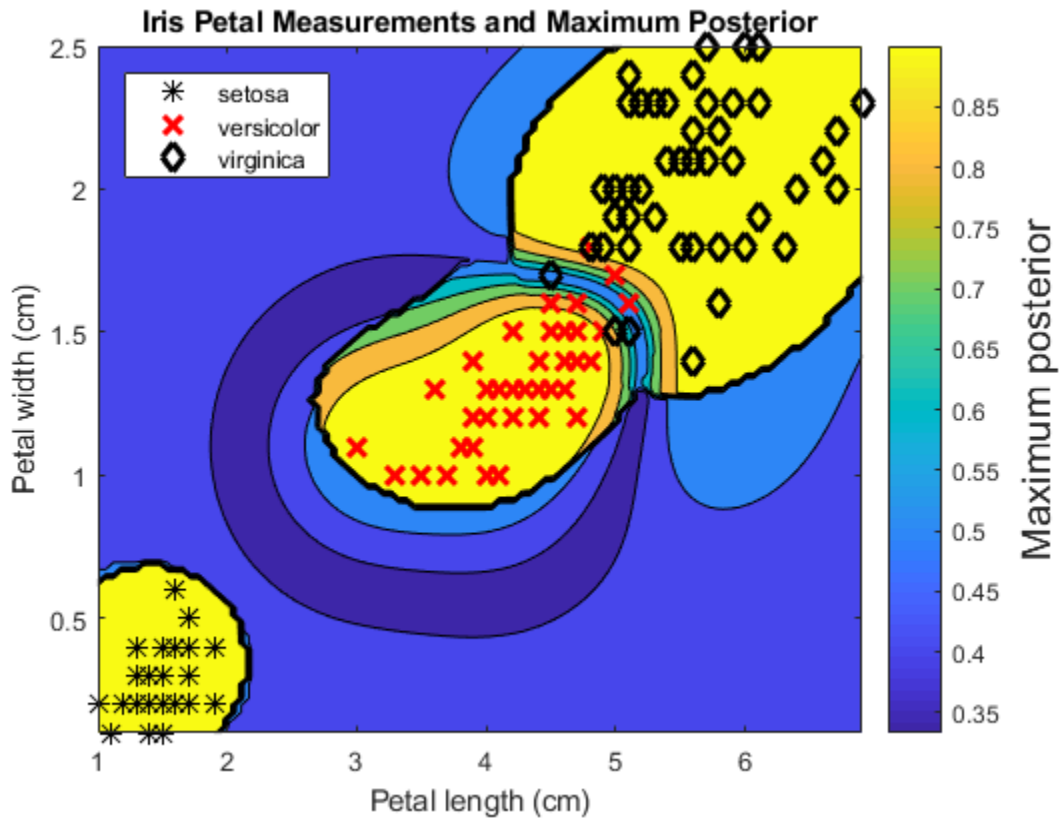
```

figure;
contourf(x1Grid, x2Grid, ...

```

```
        reshape(max(PosteriorRegion,[],2),size(x1Grid,1),size(x1Grid,2));
h = colorbar;
h.YLabel.String = 'Maximum posterior';
h.YLabel.FontSize = 15;
hold on
gh = gscatter(X(:,1),X(:,2),Y,'krk','*xd',8);
gh(2).LineWidth = 2;
gh(3).LineWidth = 2;

title 'Iris Petal Measurements and Maximum Posterior';
xlabel 'Petal length (cm)';
ylabel 'Petal width (cm)';
axis tight
legend(gh,'Location','NorthWest')
hold off
```



### Train Multiclass Linear Classification Model

Train an ECOC model composed of multiple binary, linear classification models.

Load the NLP data set.

```
load nlpdata
```

$X$  is a sparse matrix of predictor data, and  $Y$  is a categorical vector of class labels. There are more than two classes in the data.

Create a default linear-classification-model template.

```
t = templateLinear();
```

To adjust the default values, see `templateLinear`.

Train an ECOC model composed of multiple binary, linear classification models that can identify the product given the frequency distribution of words on a documentation web page. For faster training time, transpose the predictor data, and specify that observations correspond to columns.

```
X = X';
rng(1); % For reproducibility
Mdl = fitcecoc(X,Y,'Learners',t,'ObservationsIn','columns')
```

```
Mdl =
```

```
classreg.learning.classif.CompactClassificationECOC
  ResponseName: 'Y'
  ClassNames: [1x13 categorical]
  ScoreTransform: 'none'
  BinaryLearners: {78x1 cell}
  CodingMatrix: [13x78 double]
```

Alternatively, you can train an ECOC model composed of default linear classification models using `'Learners','Linear'`.

To conserve memory, `fitcecoc` returns trained ECOC models composed of linear classification learners in `CompactClassificationECOC` model objects.

## Train ECOC Classifiers Using Ensembles and Parallel Computing

Train a one-versus-all ECOC classifier using a GentleBoost ensemble of decision trees with surrogate splits. Estimate the classification error using 10-fold cross-validation.

Load and inspect the arrhythmia data set.

```
load arrhythmia
[n,p] = size(X)
isLabels = unique(Y);
```

```
nLabels = numel(isLabels)
tabulate(categorical(Y))

n =

    452

p =

    279

nLabels =

    13

    Value    Count    Percent
    1         245     54.20%
    2          44      9.73%
    3          15      3.32%
    4          15      3.32%
    5          13      2.88%
    6          25      5.53%
    7           3      0.66%
    8           2      0.44%
    9           9      1.99%
   10         50     11.06%
   14           4      0.88%
   15           5      1.11%
   16          22      4.87%
```

There are 279 predictors, and a relatively small sample size of 452. There are 16 distinct labels, but only 13 are represented in the response ( $Y$ ), and each label describes various degrees of arrhythmia. 54.20% of the observations are in class 1.

Create an ensemble template. You must specify at least three arguments: a method, a number of learners, and the type of learner. For this example, specify 'GentleBoost' for the method, 100 for the number of learners, and a decision tree template that uses surrogate splits since there are missing observations.

```
tTree = templateTree('surrogate','on');
tEnsemble = templateEnsemble('GentleBoost',100,tTree);
```

tEnsemble is a template object. Most of its properties are empty, but the software fills them with their default values during training.

Train a one-versus-all ECOC classifier using the ensembles of decision trees as binary learners. If you have a Parallel Computing Toolbox license, then you can speed up the computation by specifying to use parallel computing. This sends each binary learner to a worker in the pool (the number of workers depends on your system configuration). Also, specify that the prior probabilities are  $1/K$ , where  $K = 13$ , which is the number of distinct classes.

```
pool = parpool; % Invoke workers
options = statset('UseParallel',true);
Mdl = fitcecoc(X,Y,'Coding','onevsall','Learners',tEnsemble,...
    'Prior','uniform','Options',options);
```

```
Starting parallel pool (parpool) using the 'local' profile ... connected to 2 workers.
```

Mdl is a ClassificationECOC model.

Cross-validate the ECOC classifier using 10-fold cross-validation.

```
CVMdl = crossval(Mdl,'Options',options);
```

```
Warning: One or more folds do not contain points from all the groups.
```

CVMdl is a ClassificationPartitionedECOC model. The warning indicates that some classes are not represented while the software trains at least one fold. Therefore, those folds cannot predict labels for the missing classes. You can inspect the results of a fold using cell indexing and dot notation, e.g., access the results of the first fold by entering CVMdl.Trained{1}. Your results might vary.

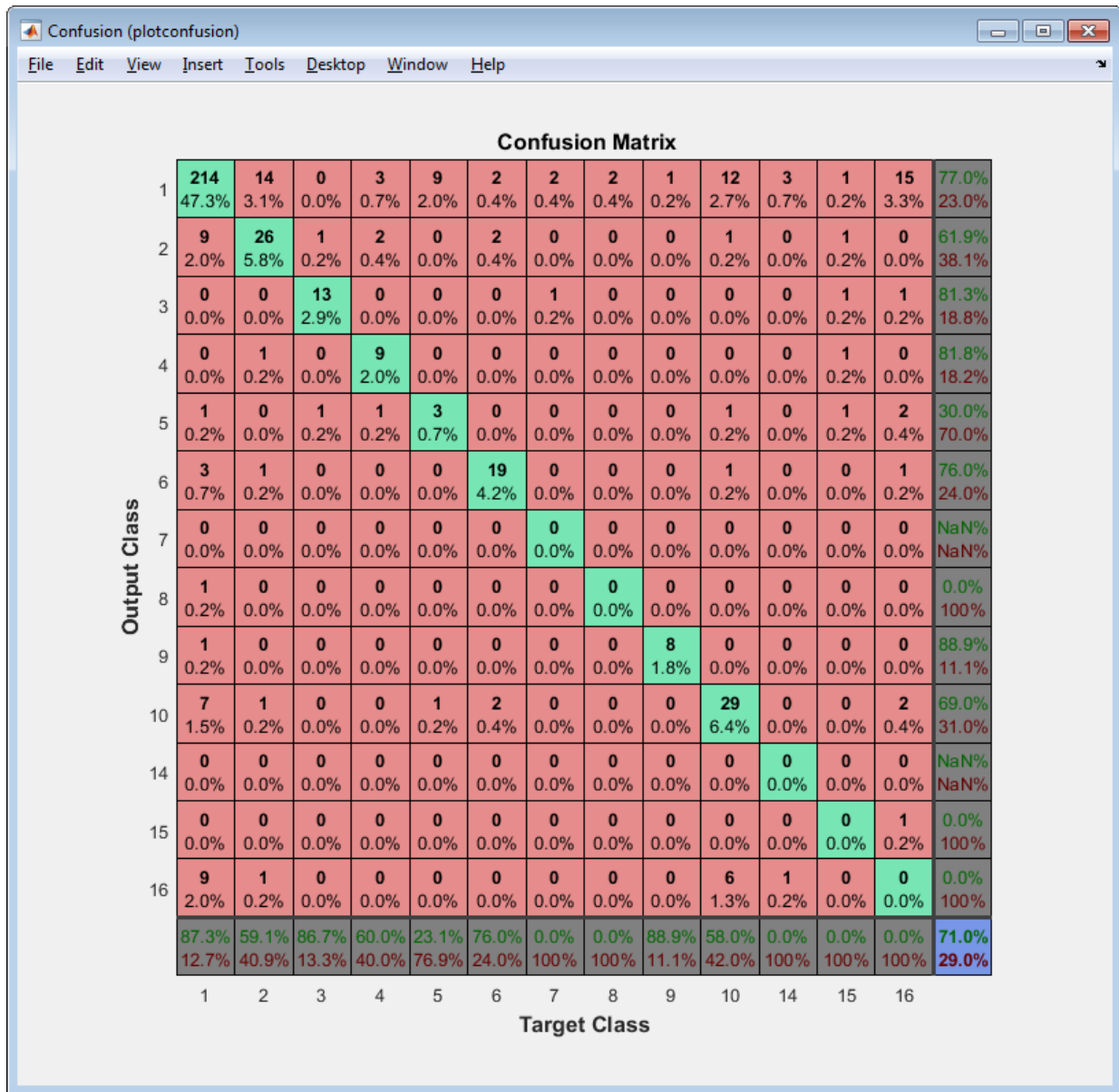
Use the cross-validated ECOC classifier to predict out-of-fold labels. You can compute the confusion matrix using confusionmat. However, if you have a Neural Network Toolbox license, you can plot the confusion matrix using plotconfusion. The input arguments of plotconfusion are not vectors of the true and predicted labels like confusionmat, but indicator matrices of the true and predicted labels. Both start as  $K$ -by- $n$  matrices of 0s. If observation  $j$  has label index  $k$  (or has predicted label  $k$ ), then element  $(k,j)$  of the true label indicator matrix (or predicted label indicator matrix) is 1. You can convert label indices returned by predict, resubPredict, or kfoldPredict to label indicator matrices using linear indexing. For details on linear indexing, see sub2ind and ind2sub.

```
oofLabel = kfoldPredict(CVMdl,'Options',options);
ConfMat = confusionmat(Y,oofLabel);

% Convert the integer label vector to a class-identifier matrix.
[~,grpOOF] = ismember(oofLabel,isLabels);
oofLabelMat = zeros(nLabels,n);
idxLinear = sub2ind([nLabels n],grpOOF,(1:n)');
oofLabelMat(idxLinear) = 1; % Flags the row corresponding to the class
[~,grpY] = ismember(Y,isLabels);
YMat = zeros(nLabels,n);
idxLinearY = sub2ind([nLabels n],grpY,(1:n)');
YMat(idxLinearY) = 1;

figure;
plotconfusion(YMat,oofLabelMat);
h = gca;
h.XTickLabel = [num2cell(isLabels); {' '}];
h.YTickLabel = [num2cell(isLabels); {' '}];
```





## Optimize ECOC Classifier

This example shows how to optimize hyperparameters automatically using `fitcecoc`. The example uses Fisher's iris data.

Load the data.

```
load fisheriris
X = meas;
Y = species;
```

Find hyperparameters that minimize five-fold cross-validation loss by using automatic hyperparameter optimization.

For reproducibility, set the random seed and use the 'expected-improvement-plus' acquisition function.

```
rng default
Mdl = fitcecoc(X,Y,'OptimizeHyperparameters','auto',...
    'HyperparameterOptimizationOptions',struct('AcquisitionFunctionName',...
    'expected-improvement-plus'))
```

```
=====
| Iter | Eval  | Objective | Objective | BestSoFar | BestSoFar | Coding
|      | result|           | runtime   | (observed) | (estim.)  |
|=====|=====|=====|=====|=====|=====|=====
|  1  | Best  | 0.10667  | 2.0786   | 0.10667   | 0.10667   | onevsone
|  2  | Best  | 0.073333 | 6.5899   | 0.073333  | 0.075057  | onevsone
|  3  | Accept| 0.08     | 0.56494  | 0.073333  | 0.073412  | onevsall
|  4  | Accept| 0.08     | 0.39391  | 0.073333  | 0.073336  | onevsall
|  5  | Best  | 0.04     | 0.46382  | 0.04       | 0.042912  | onevsone
|  6  | Accept| 0.04     | 0.53714  | 0.04       | 0.040003  | onevsone
|  7  | Accept| 0.04     | 0.47212  | 0.04       | 0.039991  | onevsone
|  8  | Best  | 0.02     | 0.49258  | 0.02       | 0.020017  | onevsone
|  9  | Accept| 0.026667 | 0.50568  | 0.02       | 0.020027  | onevsone
| 10  | Accept| 0.033333 | 0.50768  | 0.02       | 0.027303  | onevsone
| 11  | Accept| 0.026667 | 0.46401  | 0.02       | 0.02639   | onevsone
| 12  | Accept| 0.026667 | 0.4489   | 0.02       | 0.026345  | onevsone
| 13  | Accept| 0.42     | 29.722   | 0.02       | 0.026282  | onevsall
| 14  | Accept| 0.33333  | 0.54279  | 0.02       | 0.025874  | onevsall
| 15  | Accept| 0.33333  | 0.64556  | 0.02       | 0.020621  | onevsall
| 16  | Accept| 0.066667 | 0.5183   | 0.02       | 0.025808  | onevsone
| 17  | Accept| 0.10667  | 0.50443  | 0.02       | 0.025774  | onevsone
| 18  | Accept| 0.10667  | 0.49538  | 0.02       | 0.025765  | onevsone
```

	19	Accept		0.10667		0.52785		0.02		0.025772		onevsone
	20	Accept		0.04		0.55651		0.02		0.025873		onevsone
=====												
	Iter	Eval		Objective		Objective		BestSoFar		BestSoFar		Coding
		result				runtime		(observed)		(estim.)		
=====												
	21	Accept		0.34667		34.39		0.02		0.025156		onevsall
	22	Accept		0.046667		0.70889		0.02		0.025384		onevsall
	23	Accept		0.046667		0.68488		0.02		0.025456		onevsall
	24	Accept		0.04		0.59382		0.02		0.025509		onevsone
	25	Accept		0.046667		2.0358		0.02		0.025568		onevsone
	26	Accept		0.033333		0.62602		0.02		0.025665		onevsone
	27	Best		0.013333		0.61439		0.013333		0.013612		onevsone
	28	Accept		0.046667		0.59635		0.013333		0.013832		onevsone
	29	Accept		0.033333		0.4818		0.013333		0.015413		onevsone
	30	Accept		0.093333		0.44611		0.013333		0.015398		onevsone

---

Optimization completed.

MaxObjectiveEvaluations of 30 reached.

Total function evaluations: 30

Total elapsed time: 146.6451 seconds.

Total objective function evaluation time: 88.2096

Best observed feasible point:

Coding	BoxConstraint	KernelScale
onevsone	12.137	5.3747

Observed objective function value = 0.013333

Estimated objective function value = 0.015398

Function evaluation time = 0.61439

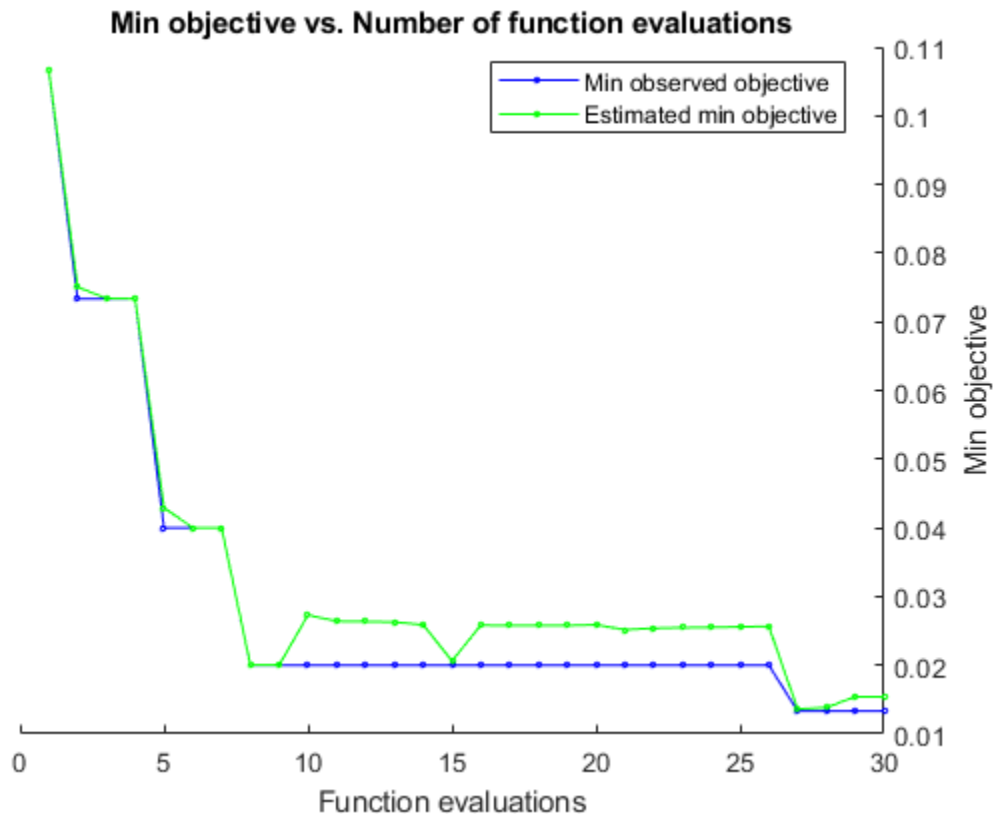
Best estimated feasible point (according to models):

Coding	BoxConstraint	KernelScale
onevsone	12.137	5.3747

Estimated objective function value = 0.015398

Estimated function evaluation time = 0.61469

```
Mdl =  
  
ClassificationECOC  
    ResponseName: 'Y'  
    CategoricalPredictors: []  
        ClassNames: {'setosa' 'versicolor' 'virginica'}  
    ScoreTransform: 'none'  
    BinaryLearners: {3x1 cell}  
        CodingName: 'onevsone'  
    HyperparameterOptimizationResults: [1x1 BayesianOptimization]
```



- “Quick Start Parallel Computing for Statistics and Machine Learning Toolbox” on page 30-2

## Input Arguments

### **Tbl** — Sample data

table

Sample data, specified as a table. Each row of **Tbl** corresponds to one observation, and each column corresponds to one predictor. Optionally, **Tbl** can contain one additional column for the response variable. Multicolumn variables and cell arrays other than cell arrays of character vectors are not accepted.

If `Tbl` contains the response variable, and you want to use all remaining variables in `Tbl` as predictors, then specify the response variable using `ResponseVarName`.

If `Tbl` contains the response variable, and you want to use only a subset of the remaining variables in `Tbl` as predictors, specify a formula using `formula`.

If `Tbl` does not contain the response variable, specify a response variable using `Y`. The length of response variable and the number of `Tbl` rows must be equal.

---

**Note** For training linear classification models, `fitcecoc` does not support tables. That is, if `Learners` is `'linear'` or contains a linear classification model learner template (see `templateLinear`), you cannot supply `Tbl`, `ResponseVarName`, or `formula`. Supply a matrix of predictor data (`X`) and an array of responses (`Y`) instead.

---

Data Types: `table`

#### **`ResponseVarName` — Response variable name**

name of variable in `Tbl`

Response variable name, specified as the name of a variable in `Tbl`.

You must specify `ResponseVarName` as a character vector. For example, if the response variable `Y` is stored as `Tbl.Y`, then specify it as `'Y'`. Otherwise, the software treats all columns of `Tbl`, including `Y`, as predictors when training the model.

The response variable must be a categorical or character array, logical or numeric vector, or cell array of character vectors. If `Y` is a character array, then each element must correspond to one row of the array.

It is good practice to specify the order of the classes using the `ClassNames` name-value pair argument.

Data Types: `char`

#### **`formula` — Explanatory model of response and subset of predictor variables**

character vector

Explanatory model of the response and a subset of the predictor variables, specified as a character vector in the form of `'Y~X1+X2+X3'`. In this form, `Y` represents the response

variable, and `X1`, `X2`, and `X3` represent the predictor variables. The variables must be variable names in `Tbl` (`Tbl.Properties.VariableNames`).

To specify a subset of variables in `Tbl` as predictors for training the model, use a formula. If you specify a formula, then the software does not use any variables in `Tbl` that do not appear in `formula`.

Data Types: `char`

### **y** — Class labels

categorical array | character array | logical vector | vector of numeric values | cell array of character vectors

Class labels to which the ECOC model is trained, specified as a categorical or character array, logical or numeric vector, or cell array of character vectors.

If `Y` is a character array, then each element must correspond to one row of the array.

The length of `Y` and the number of rows of `Tbl` or `X` must be equal.

It is good practice to specify the class order using the `ClassNames` name-value pair argument.

Data Types: `char` | `cell` | `categorical` | `logical` | `single` | `double`

### **x** — Predictor data

full matrix | sparse matrix

Predictor data, specified as a full or sparse matrix.

The length of `Y` and the number of observations in `X` must be equal.

To specify the names of the predictors in the order of their appearance in `X`, use the `PredictorNames` name-value pair argument.

---

### Note

- For linear classification learners, if you orient `X` so that observations correspond to columns and specify `'ObservationsIn'`, `'columns'`, then you can experience a significant reduction in optimization-execution time.

- For all other learners, orient  $X$  so that observations correspond to rows.
  - `fitcecoc` supports sparse matrices for training linear classification models only.
- 

Data Types: `double` | `single`

---

**Note** The software treats NaN, empty character vector (' '), and `<undefined>` elements as missing data. The software removes rows of  $X$  corresponding to missing values in  $Y$ . However, the treatment of missing values in  $X$  varies among binary learners. For details, see the training functions for your binary learners: `fitcdiscr`, `fitcknn`, `fitclinear`, `fitcnb`, `fitcsvm`, `fitctree`, or `fitcensemble`. Removing observations decreases the effective training or cross-validation sample size.

---

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

---

**Note** You cannot use any cross-validation name-value pair along with `OptimizeHyperparameters`. You can modify the cross-validation for `OptimizeHyperparameters` only by using the `HyperparameterOptimizationOptions` name-value pair.

---

Example: `'Learners', 'tree', 'Coding', 'onevsone', 'CrossVal', 'on'` specifies to use decision trees for all binary learners, a one-versus-one coding design, and to implement 10-fold cross-validation.

### ECOC Classifier Options

#### Coding — Coding design

`'onevsone'` (default) | `'allpairs'` | `'binarycomplete'` | `'denserandom'` | `'onevsall'` | `'ordinal'` | `'sparserandom'` | `'ternarycomplete'` | numeric matrix



Coding design name, specified as the comma-separated pair consisting of 'Coding' and a numeric matrix or a value in this table.

Value	Number of Binary Learners	Description
'allpairs' and 'onevsone'	$K(K-1)/2$	For each binary learner, one class is positive, another is negative, and the software ignores the rest. This design exhausts all combinations of class pair assignments.
'binarycomplete'	$2^{(K-1)} - 1$	This design partitions the classes into all binary combinations, and does not ignore any classes. For each binary learner, all class assignments are -1 and 1 with at least one positive and negative class in the assignment.
'denserandom'	Random, but approximately $10 \log_2 K$	For each binary learner, the software randomly assigns classes into positive or negative classes, with at least one of each type. For more details, see “Random Coding Design Matrices” on page 32-1923.
'onevsall'	$K$	For each binary learner, one class is positive and the rest are negative. This design exhausts all combinations of positive class assignments.

Value	Number of Binary Learners	Description
'ordinal'	$K - 1$	For the first binary learner, the first class is negative, and the rest positive. For the second binary learner, the first two classes are negative, the rest positive, and so on.
'sparsrandom'	Random, but approximately $15 \log_2 K$	For each binary learner, the software randomly assigns classes as positive or negative with probability 0.25 for each, and ignores classes with probability 0.5. For more details, see “Random Coding Design Matrices” on page 32-1923.
'ternarycomplete'	$(3^K - 2^{(K+1)} + 1) / 2$	This design partitions the classes into all ternary combinations. All class assignments are 0, -1, and 1 with at least one positive and one negative class in the assignment.

You can also specify a coding design using a custom coding matrix. The custom coding matrix is a  $K$ -by- $L$  matrix. Each row corresponds to a class and each column corresponds to a binary learner. The class order (rows) corresponds to the order in `ClassNames`. Compose the matrix by following these guidelines:

- Every element of the custom coding matrix must be -1, 0, or 1, and the value must correspond to a dichotomous class assignment. This table describes the meaning of `Coding(i, j)`, that is, the class that learner  $j$  assigns to observations in class  $i$ .

Value	Dichotomous Class Assignment
-1	Negative class
0	Before training, learner $j$ removes observations in class $i$ from the data set.

Value	Dichotomous Class Assignment
1	Positive class

- Every column must contain at least one  $-1$  or  $1$ .
- For all column indices  $i, j$  such that  $i \neq j$ ,  $\text{Coding}(:, i)$  cannot equal  $\text{Coding}(:, j)$  and  $\text{Coding}(:, i)$  cannot equal  $-\text{Coding}(:, j)$ .
- All rows of the custom coding matrix must be different.

For more details on the form of custom coding design matrices, see “Custom Coding Design Matrices” on page 32-1921.

Example: `'Coding', 'ternarycomplete'`

Data Types: `char | double | single | int16 | int32 | int64 | int8`

#### **FitPosterior** — Flag indicating whether to transform scores to posterior probabilities

`false` or `0` (default) | `true` or `1`

Flag indicating whether to transform scores to posterior probabilities, specified as the comma-separated pair consisting of `'FitPosterior'` and a true (1) or false (0).

If `FitPosterior` is true, then the software transforms binary-learner classification scores to posterior probabilities. You can obtain posterior probabilities by using `kfoldPredict`, `predict`, or `resubPredict`.

`fitcecoc` does not support fitting posterior probabilities:

- For the ensemble methods `AdaBoostM2`, `LPBoost`, `RUSBoost`, `RobustBoost`, and `TotalBoost`.
- If the binary learners (`Learners`) are linear classification models that implement SVM. To obtain posterior probabilities for linear classification models, implement logistic regression instead.

Example: `'FitPosterior', true`

Data Types: `logical`

#### **Learners** — Binary learner templates

`'svm'` (default) | `'discriminant'` | `'knn'` | `'linear'` | `'naivebayes'` | `'tree'` | template object | cell vector of template objects

Binary learner templates, specified as the comma-separated pair consisting of 'Learners' and a character vector, template object, or cell vector of template objects. Specifically, you can specify binary classifiers such as SVM, and the ensembles that use GentleBoost, LogitBoost, and RobustBoost, to solve multiclass problems. However, `fitcecoc` also supports multiclass models as binary classifiers.

- If `Learners` is a character vector, then the software trains each binary learner using the default values of the specified algorithm. This table summarizes the available algorithms.

Value	Description
'discriminant'	Discriminant analysis. For default options, see <code>templateDiscriminant</code> .
'knn'	$k$ -nearest neighbors. For default options, see <code>templateKNN</code> .
'linear'	Linear classification model. For default options, see <code>templateLinear</code> .
'naivebayes'	Naive Bayes. For default options, see <code>templateNaiveBayes</code> .
'svm'	SVM. For default options, see <code>templateSVM</code> .
'tree'	Classification trees. For default options, see <code>templateTree</code> .

- If `Learners` is a template object, then each binary learner trains according to the stored options. You can create a template object using:
  - `templateDiscriminant`, for discriminant analysis.
  - `templateEnsemble`, for ensemble learning. You must at least specify the learning method (`Method`), the number of learners (`NLearn`), and the type of learner (`Learners`). You cannot use the `AdaBoostM2` ensemble method for binary learning.
  - `templateKNN`, for  $k$ -nearest neighbors.
  - `templateLinear`, for linear classification.
  - `templateNaiveBayes`, for naive Bayes.
  - `templateSVM`, for SVM.

- `templateTree`, for classification trees.
- If `Learners` is cell vector of template objects, then:
  - Cell  $j$  corresponds to binary learner  $j$  (in other words, column  $j$  of the coding design matrix), and the cell vector must have length  $L$ .  $L$  is the number of columns in the coding design matrix. For details, see `Coding`.
  - To use one of the built-in loss functions for prediction, then all binary learners must return a score in the same range. For example, you cannot include default SVM binary learners with default naive Bayes binary learners. The former returns a score in the range  $(-\infty, \infty)$ , and the latter returns a posterior probability as a score. Otherwise, you must provide a custom loss as a function handle to functions such as `predict` and `loss`.
  - You cannot specify linear classification model learner templates with any other template in the cell array.

By default, the software trains learners using default SVM templates.

Example: `'Learners', 'tree'`

Data Types: `char` | `cell`

### **ObservationsIn — Predictor data observation dimension**

`'rows'` (default) | `'columns'`

Predictor data observation dimension, specified as the comma-separated pair consisting of `'ObservationsIn'` and `'columns'` or `'rows'`.

---

#### **Note**

- For linear classification learners, if you orient `X` so that observations correspond to columns and specify `'ObservationsIn', 'columns'`, then you can experience a significant reduction in optimization-execution time.
  - For all other learners, orient `X` so that observations correspond to rows.
- 

Example: `'ObservationsIn', 'columns'`

Data Types: `char`

**Verbose — Verbosity level**

0 (default) | 1 | 2

Verbosity level, specified as the comma-separated pair consisting of 'Verbose' and 0, 1, or 2. Verbose controls the amount of diagnostic information per binary learner that the software displays in the Command Window.

This table summarizes the available verbosity level options.

Value	Description
0	The software does not display diagnostic information.
1	The software displays diagnostic messages every time it trains a new binary learner.
2	The software displays extra diagnostic messages every time it trains a new binary learner.

Example: 'Verbose', 1

Data Types: double | single

**Cross-Validation Options****CrossVal — Flag to train cross-validated classifier**

'off' (default) | 'on'

Flag to train a cross-validated classifier, specified as the comma-separated pair consisting of 'Crossval' and 'on' or 'off'.

If you specify 'on', then the software trains a cross-validated classifier with 10 folds.

You can override this cross-validation setting using one of the `CVPartition`, `Holdout`, `KFold`, or `Leaveout` name-value pair arguments. You can only use one cross-validation name-value pair argument at a time to create a cross-validated model.

Alternatively, cross-validate later by passing `Mdl` to `crossval`.

Example: 'Crossval', 'on'

Data Types: char

**CVPartition — Cross-validation partition**

[] (default) | `cvpartition` partition object

Cross-validation partition, specified as the comma-separated pair consisting of 'CVPartition' and a `cvpartition` partition object as created by `cvpartition`. The partition object specifies the type of cross-validation, and also the indexing for training and validation sets.

To create a cross-validated model, you can use one of these four name-value pair arguments only: `CVPartition`, `Holdout`, `KFold`, or `Leaveout`.

Example: Suppose you create a random partition for 5-fold cross-validation on 500 observations by using `cvp = cvpartition(500, 'KFold', 5)`. Then, you can specify the cross-validation partition for `fitcecoc` as 'CVPartition', `cvp`.

**Holdout — Fraction of data for holdout validation**

scalar value in the range (0,1)

Fraction of data used for holdout validation, specified as the comma-separated pair consisting of 'Holdout' and a scalar value in the range (0,1). If you specify 'Holdout',  $p$ , then the software:

- 1 Randomly reserves  $p*100\%$  of the data as validation data, and trains the model using the rest of the data
- 2 Stores the compact, trained model in the `Trained` property of the cross-validated model.

To create a cross-validated model, you can use one of these four name-value pair arguments only: `CVPartition`, `Holdout`, `KFold`, or `Leaveout`.

Example: 'Holdout', 0.1

Data Types: `double` | `single`

**KFold — Number of folds**

10 (default) | positive integer value greater than 1

Number of folds to use in a cross-validated model, specified as the comma-separated pair consisting of 'KFold' and a positive integer value greater than 1. If you specify, e.g., 'KFold',  $k$ , then the software:

- 1 Randomly partitions the data into  $k$  sets

- 2 For each set, reserves the set as validation data, and trains the model using the other  $k - 1$  sets
- 3 Stores the  $k$  compact, trained models in the cells of a  $k$ -by-1 cell vector in the `Trained` property of the cross-validated model.

To create a cross-validated model, you can use one of these four name-value pair arguments only: `CVPartition`, `Holdout`, `KFold`, or `Leaveout`.

Example: `'KFold', 5`

Data Types: `single` | `double`

### **Leaveout** — Leave-one-out cross-validation flag

`'off'` (default) | `'on'`

Leave-one-out cross-validation flag, specified as the comma-separated pair consisting of `'Leaveout'` and `'on'` or `'off'`. If you specify `'Leaveout', 'on'`, then, for each of the  $n$  observations, where  $n$  is `size(Mdl.X, 1)`, the software:

- 1 Reserves the observation as validation data, and trains the model using the other  $n - 1$  observations
- 2 Stores the  $n$  compact, trained models in the cells of a  $n$ -by-1 cell vector in the `Trained` property of the cross-validated model.

To create a cross-validated model, you can use one of these four options only: `CVPartition`, `Holdout`, `KFold`, or `Leaveout`.

---

**Note** Leave-one-out is not recommended for cross-validating ECOC models composed of linear classification model learners.

---

Example: `'Leaveout', 'on'`

Data Types: `char`

### **Other Classification Options**

#### **CategoricalPredictors** — Categorical predictors list

vector of positive integers | logical vector | character matrix | cell array of character vectors | `'all'`



Categorical predictors list, specified as the comma-separated pair consisting of 'CategoricalPredictors' and one of these values:

Value	Description
Vector of positive integers	An entry in the vector is the index value corresponding to the column of the predictor data (X or Tbl) that contains a categorical variable.
Logical vector	A true entry means that the corresponding column of predictor data (X or Tbl) is a categorical variable.
Character matrix	Each row of the matrix is the name of a predictor variable. The names must match the entries in PredictorNames. Pad the names with extra blanks so each row of the character matrix has the same length.
Cell array of character vectors	Each element in the array is the name of a predictor variable. The names must match the entries in PredictorNames.
'all'	All predictors are categorical.

Specification of CategoricalPredictors is appropriate if:

- At least one predictor is categorical and all binary learners are classification trees, naive Bayes learners, SVM, or ensembles of classification trees.
- All predictors are categorical and at least one binary learner is  $k$ NN.

If you specify CategoricalPredictors for any other learner, then the software warns that it cannot train that binary learner. For example, the software cannot train linear classification model learners using categorical predictors.

By default, if the predictor data is in a table (Tbl), fitcecoc assumes that a variable is categorical if it contains logical values, categorical values, or a cell array of character vectors. If the predictor data is a matrix (X), fitcecoc assumes all predictors are continuous. To identify any categorical predictors when the data is a matrix, use the 'CategoricalPredictors' name-value pair argument.

Example: 'CategoricalPredictors', 'all'

Data Types: single | double | logical | char | cell

#### **ClassNames** — Names of classes to use for training

categorical array | cell array of character vectors | character array | logical vector | numeric vector

Names of classes to use for training, specified as the comma-separated pair consisting of 'ClassNames' and a categorical or character array, logical or numeric vector, or cell array of character vectors. `ClassNames` must be the same data type as `Y`.

If `ClassNames` is a character array, then each element must correspond to one *row* of the array.

Use `ClassNames` to:

- Order the classes during training.
- Specify the order of any input or output argument dimension that corresponds to the class order. For example, use `ClassNames` to specify the order of the dimensions of `Cost` or the column order of classification scores returned by `predict`.
- Select a subset of classes for training. For example, suppose that the set of all distinct class names in `Y` is {'a', 'b', 'c'}. To train the model using observations from classes 'a' and 'c' only, specify 'ClassNames', {'a', 'c'}.

The default is the set of all distinct class names in `Y`.

Example: 'ClassNames', {'b', 'g'}

Data Types: categorical | char | logical | single | double | cell

### **Cost — Misclassification cost**

square matrix | structure array

Misclassification cost, specified as the comma-separated pair consisting of 'Cost' and a square matrix or structure. If you specify:

- The square matrix `Cost`, then `Cost(i, j)` is the cost of classifying a point into class `j` if its true class is `i`. That is, the rows correspond to the true class and the columns correspond to the predicted class. To specify the class order for the corresponding rows and columns of `Cost`, additionally specify the `ClassNames` name-value pair argument.
- The structure `S`, then it must have two fields:
  - `S.ClassNames`, which contains the class names as a variable of the same data type as `Y`
  - `S.ClassificationCosts`, which contains the cost matrix with rows and columns ordered as in `S.ClassNames`

The default is  $\text{ones}(K) - \text{eye}(K)$ , where  $K$  is the number of distinct classes.

Example: 'Cost', [0 1 2 ; 1 0 2; 2 2 0]

Data Types: double | single | struct

### Options — Parallel computing options

[] (default) | structure array returned by `statset`

Parallel computing options, specified as the comma-separated pair consisting of 'Options' and a structure array returned by `statset`. These options require Parallel Computing Toolbox. `fitcecoc` uses 'Streams', 'UseParallel', and 'UseSubstreams' fields.

This table summarizes the available options.

Option	Description
'Streams'	<p>A <code>RandStream</code> object or cell array of such objects. If you do not specify <code>Streams</code>, the software uses the default stream or streams. If you specify <code>Streams</code>, use a single object except when the following are true:</p> <ul style="list-style-type: none"> <li>• You have an open parallel pool.</li> <li>• <code>UseParallel</code> is true.</li> <li>• <code>UseSubstreams</code> is false.</li> </ul> <p>In that case, use a cell array of the same size as the parallel pool. If a parallel pool is not open, then the software tries to open one (depending on your preferences), and <code>Streams</code> must supply a single random number stream.</p>
'UseParallel'	<p>If you have Parallel Computing Toolbox, then you can invoke a pool of workers by setting 'UseParallel', true.</p>

Option	Description
'UseSubstreams'	Set to true to compute in parallel using the stream specified by 'Streams'. Default is false. For example, set Streams to a type allowing substreams, such as 'mlfg6331_64' or 'mrg32k3a'.

A best practice to ensure more predictable results is to use `parpool` and explicitly create a parallel pool before you invoke parallel computing using `fitcecoc`.

Example: `'Options', statset('UseParallel', true)`

Data Types: `struct`

### **PredictorNames** — Predictor variable names

cell array of unique character vectors

Predictor variable names, specified as the comma-separated pair consisting of 'PredictorNames' and a cell array of unique character vectors. The functionality of 'PredictorNames' depends on the way you supply the training data.

- If you supply `X` and `Y`, then you can use 'PredictorNames' to give the predictor variables in `X` names.
  - The order of the names in `PredictorNames` must correspond to the column order of `X`. That is, `PredictorNames{1}` is the name of `X(:, 1)`, `PredictorNames{2}` is the name of `X(:, 2)`, and so on. Also, `size(X, 2)` and `numel(PredictorNames)` must be equal.
  - By default, `PredictorNames` is `{'x1', 'x2', ...}`.
- If you supply `Tbl`, then you can use 'PredictorNames' to choose which predictor variables to use in training. That is, `fitcecoc` uses the predictor variables in `PredictorNames` and the response only in training.
  - `PredictorNames` must be a subset of `Tbl.Properties.VariableNames` and cannot include the name of the response variable.
  - By default, `PredictorNames` contains the names of all predictor variables.
  - It good practice to specify the predictors for training using one of 'PredictorNames' or formula only.

```
Example: 'PredictorNames',
{'SepalLength', 'SepalWidth', 'PedalLength', 'PedalWidth'}
```

Data Types: cell

### **Prior — Prior probabilities**

'empirical' (default) | 'uniform' | numeric vector | structure array

Prior probabilities for each class, specified as the comma-separated pair consisting of 'Prior' and a value in this table.

Value	Description
'empirical'	The class prior probabilities are the class relative frequencies in $Y$ .
'uniform'	All class prior probabilities are equal to $1/K$ , where $K$ is the number of classes.
numeric vector	Each element is a class prior probability. Order the elements according to <code>Mdl.ClassNames</code> or specify the order using the <code>ClassNames</code> name-value pair argument. The software normalizes the elements such that they sum to 1.
structure	A structure $S$ with two fields: <ul style="list-style-type: none"> <li><math>S.ClassNames</math> contains the class names as a variable of the same type as <math>Y</math>.</li> <li><math>S.ClassProbs</math> contains a vector of corresponding prior probabilities. The software normalizes the elements such that they sum to 1.</li> </ul>

For more details on how the software incorporates class prior probabilities, see “Prior Probabilities and Cost” on page 32-1922.

```
Example: struct('ClassNames',
{'setosa', 'versicolor', 'virginica'}), 'ClassProbs', 1:3)
```

Data Types: single | double | char | struct

**ResponseName** — Response variable name

'Y' (default) | character vector

Response variable name, specified as the comma-separated pair consisting of 'ResponseName' and a character vector.

- If you supply Y, then you can use 'ResponseName' to specify a name for the response variable.
- If you supply ResponseVarName or formula, then you cannot use 'ResponseName'.

Example: 'ResponseName', 'response'

Data Types: char

**ScoreTransform** — Score transformation

'none' (default) | character vector | function handle

Score transformation, specified as the comma-separated pair consisting of 'ScoreTransform' and either a character vector or a function handle.

This table summarizes the available character vectors.

Value	Description
'doublelogit'	$1/(1 + e^{-2x})$
'invlogit'	$\log(x / (1-x))$
'ismax'	Set the score for the class with the largest score to 1, and set the scores for all other classes to 0.
'logit'	$1/(1 + e^{-x})$
'none' or 'identity'	$x$ (no transformation)
'sign'	-1 for $x < 0$ 0 for $x = 0$ 1 for $x > 0$
'symmetric'	$2x - 1$
'symmetricismax'	Set the score for the class with the largest score to 1, and set the scores for all other classes to -1.
'symmetriclogit'	$2/(1 + e^{-x}) - 1$

For a MATLAB function or a function you define, use its function handle. The function handle must accept a matrix (the original scores) and return a matrix of the same size (the transformed scores).

Example: `'ScoreTransform', 'logit'`

Data Types: `char` | `function_handle`

### **Weights — Observation weights**

numeric vector of positive values | character vector

Observation weights, specified as the comma-separated pair consisting of `'Weights'` and a numeric vector of positive values or name of a variable in `Tbl`. The software weighs the observations in each row of `X` or `Tbl` with the corresponding value in `Weights`. The size of `Weights` must equal the number of rows of `X` or `Tbl`.

If you specify the input data as a table `Tbl`, then `Weights` can be the name of a variable in `Tbl` that contains a numeric vector. In this case, you must specify `Weights` as a character vector. For example, if the weights vector `W` is stored as `Tbl.W`, then specify it as `'W'`. Otherwise, the software treats all columns of `Tbl`, including `W`, as predictors or the response when training the model.

The software normalizes `Weights` to sum up to the value of the prior probability in the respective class.

By default, `Weights` is ones ( $n, 1$ ), where  $n$  is the number of observations in `X` or `Tbl`.

Data Types: `double` | `single` | `char`

### **Hyperparameter Optimization**

#### **OptimizeHyperparameters — Parameters to optimize**

`'none'` (default) | `'auto'` | `'all'` | cell array of eligible parameter names | vector of `optimizableVariable` objects

Parameters to optimize, specified as:

- `'none'` — Do not optimize.
- `'auto'` — Use `{'Coding'}` along with the default parameters for the specified Learners:
  - `Learners = 'svm'` (default) — `{'BoxConstraint', 'KernelScale'}`

- `Learners = 'discriminant' — {'Delta', 'Gamma'}`
- `Learners = 'knn' — {'Distance', 'NumNeighbors'}`
- `Learners = 'linear' — {'Lambda', 'Learner'}`
- `Learners = 'naivebayes' — {'DistributionNames', 'Width'}`
- `Learners = 'tree' — {'MinLeafSize'}`
- `'all'` — Optimize all eligible parameters.
- Cell array of eligible parameter names
- Vector of `optimizableVariable` objects, typically the output of hyperparameters

The optimization attempts to minimize the cross-validation loss (error) for `fitcecoc` by varying the parameters. For information about cross-validation loss in a different context, see “Classification Loss” on page 32-3250. To control the cross-validation type and other aspects of the optimization, use the `HyperparameterOptimizationOptions` name-value pair.

---

**Note** `OptimizeHyperparameters` values override any values you set using other name-value pairs. For example, setting `OptimizeHyperparameters` to `'auto'` causes the `'auto'` values to apply.

---

The eligible parameters for `fitcecoc` are:

- Coding — `fitcecoc` searches among `'onevsall'` and `'onevsone'`.
- The eligible hyperparameters for the chosen `Learners`:

Learners	Eligible Hyperparameters (Bold = Default)	Default Range
'discriminant'	<b>Delta</b>	Log-scaled in the range [1e-6, 1e3]
	DiscrimType	'linear', 'quadratic', 'diagLinear', 'diagQuadratic', 'pseudoLinear', and 'pseudoQuadratic'
	<b>Gamma</b>	Real values in [0, 1]



Learners	Eligible Hyperparameters (Bold = Default)	Default Range
'knn'	<b>Distance</b>	'cityblock', 'chebychev', 'correlation', 'cosine', 'euclidean', 'hamming', 'jaccard', 'mahalanobis', 'minkowski', 'seuclidean', and 'spearman'
	DistanceWeight	'equal', 'inverse', and 'squaredinverse'
	Exponent	Positive values in [0.5, 3]
	<b>NumNeighbors</b>	Positive integer values log-scaled in the range [1, max(2, round(NumObservations/2))]
	Standardize	'true' and 'false'
'linear'	<b>Lambda</b>	Positive values log-scaled in the range [1e-5/NumObservations, 1e5/NumObservations]
	<b>Learner</b>	'svm' and 'logistic'
	Regularization	'ridge' and 'lasso'
'naivebayes'	<b>DistributionNames</b>	'normal' and 'kernel'
	<b>Width</b>	Positive values log-scaled in the range [MinPredictorDiff/4, max(MaxPredictorRange, MinPredictorDiff)]
	Kernel	'normal', 'box', 'epanechnikov', and 'triangle'
'svm'	<b>BoxConstraint</b>	Positive values log-scaled in the range [1e-3, 1e3]
	<b>KernelScale</b>	Positive values log-scaled in the range [1e-3, 1e3]

Learners	Eligible Hyperparameters (Bold = Default)	Default Range
	KernelFunction	'gaussian', 'linear', and 'polynomial'
	PolynomialOrder	Integers in the range [2, 4]
	Standardize	'true' and 'false'
'tree'	MaxNumSplits	Integers log-scaled in the range [1, max(2, NumObservations-1)]
	<b>MinLeafSize</b>	Integers log-scaled in the range [1, max(2, floor(NumObservations/2))]
	NumVariablesToSample	Integers in the range [1, max(2, NumPredictors)]
	SplitCriterion	'gdi', 'deviance', and 'twoing'

Alternatively, use hyperparameters with your chosen Learners, such as

```
load fisheriris % hyperparameters requires data and learner
params = hyperparameters('fitcecoc', meas, species, 'svm');
```

To see the eligible and default hyperparameters, examine `params`.

Set nondefault parameters by passing a vector of `optimizableVariable` objects that have nondefault values. For example,

```
load fisheriris
params = hyperparameters('fitcecoc', meas, species, 'svm');
params(2).Range = [1e-4, 1e6];
```

Pass `params` as the value of `OptimizeHyperparameters`.

By default, iterative display appears at the command line, and plots appear according to the number of hyperparameters in the optimization. For the optimization and plots, the objective function is  $\log(1 + \text{cross-validation loss})$  for regression, and the misclassification rate for classification. To control the iterative display, set the `HyperparameterOptimizationOptions` name-value pair, `Verbose` field. To control the plots, set the `HyperparameterOptimizationOptions` name-value pair, `ShowPlots` field.

For an example, see “Optimize ECOC Classifier” on page 32-1883.

Example: 'auto'

Data Types: char | cell

### HyperparameterOptimizationOptions — Options for optimization structure

Options for optimization, specified as a structure. Modifies the effect of the OptimizeHyperparameters name-value pair. All fields in the structure are optional.

Field Name	Values	Default
Optimizer	<ul style="list-style-type: none"> <li>'bayesopt' — Use Bayesian optimization. Internally, this setting calls bayesopt.</li> <li>'gridsearch' — Use grid search with NumGridDivisions values per dimension.</li> <li>'randomsearch' — Search at random among MaxObjectiveEvaluations points.</li> </ul> <p>'gridsearch' searches in a random order, using uniform sampling without replacement from the grid. After optimization, you can get a table in grid order by using the command sortrows(Mdl.ParameterOptimizationResults).</p>	'bayesopt'

Field Name	Values	Default
AcquisitionFunctionName	<ul style="list-style-type: none"> <li>• 'expected-improvement-per-second-plus'</li> <li>• 'expected-improvement'</li> <li>• 'expected-improvement-plus'</li> <li>• 'expected-improvement-per-second'</li> <li>• 'lower-confidence-bound'</li> <li>• 'probability-of-improvement'</li> </ul> <p>For details, see the bayesopt AcquisitionFunctionName name-value pair, or “Acquisition Function Types” on page 10-4.</p>	'expected-improvement-per-second-plus'
MaxObjectiveEvaluations	Maximum number of objective function evaluations.	30 for 'bayesopt' or 'randomsearch', and the entire grid for 'gridsearch'
MaxTime	Time limit, specified as a positive real. The time limit is in seconds, as measured by tic and toc. Run time can exceed MaxTime because MaxTime does not interrupt function evaluations.	Inf
NumGridDivisions	For 'gridsearch', the number of values in each dimension. Can be a vector of positive integers giving the number of values for each dimension, or a scalar that applies to all dimensions. Ignored for categorical variables.	10

Field Name	Values	Default
ShowPlots	Logical value indicating whether to show plots. If <code>true</code> , plots the best objective function value against iteration number. If there are one or two optimization parameters, and if <code>Optimizer</code> is <code>'bayesopt'</code> , then <code>ShowPlots</code> also plots a model of the objective function against the parameters.	<code>true</code>
SaveIntermediateResults	Logical value indicating whether to save results when <code>Optimizer</code> is <code>'bayesopt'</code> . If <code>true</code> , overwrites a workspace variable named <code>'BayesoptResults'</code> at each iteration. The variable is a <code>BayesianOptimization</code> object.	<code>false</code>
Verbose	Display to the command line. <ul style="list-style-type: none"> <li>• 0 — No iterative display</li> <li>• 1 — Iterative display</li> <li>• 2 — Iterative display with extra information</li> </ul> For details, see the <code>bayesopt</code> <code>Verbose</code> name-value pair.	1
UseParallel	Logical value indicating whether to run Bayesian optimization in parallel, which requires Parallel Computing Toolbox. For details, see “Parallel Bayesian Optimization” on page 10-9.	<code>false</code>

Field Name	Values	Default
Repartition	Logical value indicating whether to repartition the cross-validation at every iteration. If <code>false</code> , the optimizer uses a single partition for the optimization.  <code>true</code> usually gives the most robust results because this setting takes partitioning noise into account. However, for good results, <code>true</code> requires at least twice as many function evaluations.	<code>false</code>
Use no more than one of the following three field names.		
CVPartition	A <code>cvpartition</code> object, as created by <code>cvpartition</code>	<code>Kfold = 5</code>
Holdout	A scalar in the range $(0, 1)$ representing the holdout fraction.	
Kfold	An integer greater than 1.	

Example: `struct('MaxObjectiveEvaluations', 60)`

Data Types: `struct`

## Output Arguments

### **mdl** — Trained ECOC model

`ClassificationECOC` model object | `CompactClassificationECOC` model object | `ClassificationPartitionedECOC` cross-validated model object | `ClassificationPartitionedLinearECOC` cross-validated model object

Trained ECOC classifier, returned as a `ClassificationECOC` or `CompactClassificationECOC` model object, or a `ClassificationPartitionedECOC` or `ClassificationPartitionedLinearECOC` cross-validated model object.

This table describes which types of model objects `fitcecoc` returns.

Linear Classification Model Learners?	Cross-Validation?	Returned model object
No	No	ClassificationECOC
No	Yes	ClassificationPartitionedECOC
Yes	No	CompactClassificationECOC
Yes	Yes	ClassificationPartitionedLinearECOC

### HyperparameterOptimizationResults — Description of the cross-validation optimization of hyperparameters

BayesianOptimization object | table of hyperparameters and associated values

Description of the cross-validation optimization of hyperparameters, returned as a BayesianOptimization object or a table of hyperparameters and associated values. Nonempty when the OptimizeHyperparameters name-value pair was nonempty and Learners was 'linear'. Value depends on the setting of the HyperparameterOptimizationOptions name-value pair:

- 'bayesopt' (default) — Object of class BayesianOptimization.
- 'gridsearch' or 'randomsearch' — Table of hyperparameters used, observed objective function values (cross-validation loss), and rank of observation from smallest (best) to highest (worst).

Data Types: table

## Limitations

- For training linear classification models, fitcecoc does not support tables. That is, if Learners is 'linear' or contains a linear classification model learner template (see templateLinear), then you cannot supply Tbl, ResponseVarName, or formula. Supply a full or sparse matrix of predictor data ( $X$ ) and an array of responses ( $Y$ ) instead.
- fitcecoc supports sparse matrices for training linear classification models only. For all other models, supply a full matrix of predictor data instead.

## Definitions

### Binary Loss

A binary loss is a function of the class and classification score that determines how well a binary learner classifies an observation into the class.

Let:

- $m_{kj}$  be element  $(k,j)$  of the coding design matrix  $M$  (i.e., the code corresponding to class  $k$  of binary learner  $j$ )
- $s_j$  be the score of binary learner  $j$  for an observation
- $g$  be the binary loss function
- $\hat{k}$  be the predicted class for the observation

In loss-based decoding [15], the class producing the minimum sum of the binary losses over binary learners determines the predicted class of an observation, that is,

$$\hat{k} = \operatorname{argmin}_k \sum_{j=1}^L |m_{kj}| g(m_{kj}, s_j).$$

In loss-weighted decoding [15], the class producing the minimum average of the binary losses over binary learners determines the predicted class of an observation, that is,

$$\hat{k} = \operatorname{argmin}_k \frac{\sum_{j=1}^L |m_{kj}| g(m_{kj}, s_j)}{\sum_{j=1}^L |m_{kj}|}.$$

Allwein et al. [2] suggest that loss-weighted decoding improves classification accuracy by keeping loss values for all classes in the same dynamic range.

This table summarizes the supported loss functions, where  $y_j$  is a class label for a particular binary learner (in the set  $\{-1,1,0\}$ ),  $s_j$  is the score for observation  $j$ , and  $g(y_j, s_j)$ .

Value	Description	Score Domain	$g(y_j, s_j)$
'binodeviance'	Binomial deviance	$(-\infty, \infty)$	$\log[1 + \exp(-2y_j s_j)] / [2\log(2)]$



Value	Description	Score Domain	$g(y_j, s_j)$
'exponential'	Exponential	$(-\infty, \infty)$	$\exp(-y_j s_j)/2$
'hamming'	Hamming	$[0, 1]$ or $(-\infty, \infty)$	$[1 - \text{sign}(y_j s_j)]/2$
'hinge'	Hinge	$(-\infty, \infty)$	$\max(0, 1 - y_j s_j)/2$
'linear'	Linear	$(-\infty, \infty)$	$(1 - y_j s_j)/2$
'logit'	Logistic	$(-\infty, \infty)$	$\log[1 + \exp(-y_j s_j)]/[2\log(2)]$
'quadratic'	Quadratic	$[0, 1]$	$[1 - y_j(2s_j - 1)]^2/2$

The software normalizes the binary losses such that the loss is 0.5 when  $y_j = 0$ , and aggregates using the average of the binary learners [2].

Do not confuse the binary loss with the overall classification loss (specified by the `LossFun` name-value pair argument of `predict` and `loss`), e.g., classification error, which measures how well an ECOC classifier performs as a whole.

## Coding Design

A coding design is a matrix where elements direct which classes are trained by each binary learner, that is, how the multiclass problem is reduced to a series of binary problems.

Each row of the coding design corresponds to a distinct class, and each column corresponds to a binary learner. In a ternary coding design (adopted by the software), for a particular column (or binary learner):

- Rows containing a 1 indicate to the binary learner to group all observations in the corresponding classes into a positive class.
- Rows containing a -1 indicate to the binary learner to group all observations in the corresponding classes into a negative class.
- Rows containing a 0 indicate to the binary learner to ignore all observations in the corresponding classes.

Coding matrices with large, minimal, pair-wise row distances based on the Hamming measure are desirable. For details on the pair-wise row distance measure, see “Random Coding Design Matrices” on page 32-1923 and [4].

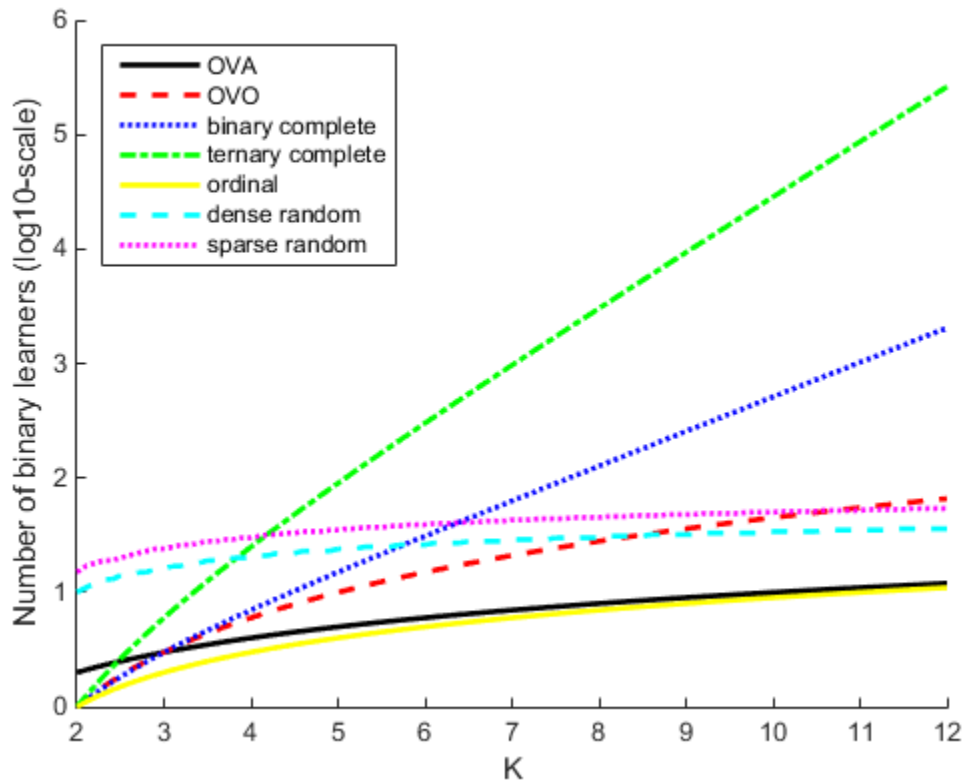
This table describes popular coding designs. For the example, suppose  $K$  (the number of distinct classes) is 3.

Coding Design	Description	Number of Learners	Minimal Pair-Wise Row Distance
one-versus-all (OVA)	For each binary learner, one class is positive and the rest are negative. This design exhausts all combinations of positive class assignments.	$K$	2
one-versus-one (OVO)	For each binary learner, one class is positive, another is negative, and the rest are ignored. This design exhausts all combinations of class pair assignments.	$K(K - 1)/2$	1
binary complete	This design partitions the classes into all binary combinations, and does not ignore any classes. That is, all class assignments are $-1$ and $1$ with at least one positive and negative class in the assignment for each binary learner.	$2^{K-1} - 1$	$2^{K-2}$

Coding Design	Description	Number of Learners	Minimal Pair-Wise Row Distance
ternary complete	This design partitions the classes into all ternary combinations. That is, all class assignments are 0, -1, and 1 with at least one positive and negative class in the assignment for each binary learner.	$(3^K - 2^{K+1} + 1)/2$	$3^{K-2}$
ordinal	For the first binary learner, the first class is negative, and the rest positive. For the second binary learner, the first two classes are negative, and the rest positive, and so on.	$K - 1$	1
dense random	For each binary learner, the software randomly assigns classes into positive or negative classes, with at least one of each type. For more details, see “Random Coding Design Matrices” on page 32-1923.	Random, but approximately $10 \log_2 K$	Variable

Coding Design	Description	Number of Learners	Minimal Pair-Wise Row Distance
sparse random	For each binary learner, the software randomly assigns classes as positive or negative with probability 0.25 for each, and ignores classes with probability 0.5. For more details, see “Random Coding Design Matrices” on page 32-1923.	Random, but approximately $15 \log_2 K$	Variable

This plot compares the number of binary learners for the coding designs with increasing  $K$ .



## Error-Correcting Output Codes Model

An error-correcting output codes (ECOC) model reduces the problem of classification with three or more classes to a set of binary classifiers.

ECOC classification requires a coding design, which determines the classes that the binary learners train on, and a decoding scheme, which determines how the results (predictions) of the binary classifiers are aggregated.

Suppose that:

- There are three classes.

- The coding design is one-versus-one.
- The decoding scheme uses loss  $g$ .
- The learners are SVMs.

To build this classification model, ECOC follows these steps.

- 1 A one-versus-one coding design is

	Learner 1	Learner 2	Learner 3
Class 1	1	1	0
Class 2	-1	0	1
Class 3	0	-1	-1

Learner 1 trains on observations having Class 1 and Class 2, and treats Class 1 as the positive class and Class 2 as the negative class. The other learners are trained similarly. Let  $M$  be the coding design matrix with elements  $m_{kl}$ , and  $s_l$  be the predicted classification score for the positive class of learner  $l$ .

- 2 A new observation is assigned to the class ( $\hat{k}$ ) that minimizes the aggregation of the losses for the  $L$  binary learners. That is,

$$\hat{k} = \underset{k}{\operatorname{argmin}} \frac{\sum_{l=1}^L |m_{kl}| g(m_{kl}, s_l)}{\sum_{l=1}^L |m_{kl}|}.$$

ECOC models can improve classification accuracy, even compared to other multiclass models [2].

## Tips

- The number of binary learners grows with the number of classes. For a problem with many classes, the `binarycomplete` and `ternarycomplete` coding designs are not efficient. However:
  - If  $K \leq 4$ , then use `ternarycomplete` coding design rather than `sparserandom`.
  - If  $K \leq 5$ , then use `binarycomplete` coding design rather than `denserandom`.

You can display the coding design matrix of a trained ECOC classifier by entering `Mdl.CodingMatrix` into the Command Window.

- You should form a coding matrix using intimate knowledge of the application, and taking into account computational constraints. If you have sufficient computational power and time, then try several coding matrices and choose the one with the best performance (e.g., check the confusion matrices for each model using `confusionmat`).
- Leave-one-out cross-validation (`Leaveout`) is inefficient for data sets with many observations. Instead, use  $k$ -fold cross-validation (`KFold`).
- After training a `ClassificationECOC` or `CompactClassificationECOC` model object by using `fitcecoc`, use the function `predict` and the trained model object to generate C code that predicts labels for new data. For details, see “Code Generation” on page 32-0 .

## Algorithms

### Custom Coding Design Matrices

Custom coding matrices must have a certain form. The software validates custom coding matrices by ensuring:

- Every element is -1, 0, or 1.
- Every column contains at least one -1 and one 1.
- For all distinct column vectors  $u$  and  $v$ ,  $u \neq v$  and  $u \neq -v$ .
- All rows vectors are unique.
- The matrix can separate any two classes. That is, you can travel from any row to any other row following these rules:
  - You can move vertically from 1 to -1 or -1 to 1.
  - You can move horizontally from a nonzero element to another nonzero element.
  - You can use a column of the matrix for a vertical move only once.

If it is not possible to move from row  $i$  to row  $j$  using these rules, then classes  $i$  and  $j$  cannot be separated by the design. For example, in the coding design

$$\begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{bmatrix}$$

classes 1 and 2 cannot be separated from classes 3 and 4 (that is, you cannot move horizontally from the -1 in row 2 to column 2 since there is a 0 in that position).

Therefore, the software rejects this coding design.

## Parallel Computing

If you use parallel computing (see `Options`), then `fitcecoc` trains binary learners in parallel.

## Prior Probabilities and Cost

- **Prior probabilities** — The software normalizes the specified class prior probabilities (`Prior`) for each binary learner. Let  $M$  be the coding design matrix and  $I(A,c)$  be an indicator matrix. The indicator matrix has the same dimensions as  $A$ . If the corresponding element of  $A$  is  $c$ , then the indicator matrix has elements equaling one, and zero otherwise. Let  $M_{+1}$  and  $M_{-1}$  be  $K$ -by- $L$  matrices such that:
  - $M_{+1} = M \circ I(M,1)$ , where  $\circ$  is element-wise multiplication (that is, `Mplus = M.* (M == 1)`). Also, let  $m_l^{(+1)}$  be column vector  $l$  of  $M_{+1}$ .
  - $M_{-1} = -M \circ I(M,-1)$  (that is, `Mminus = -M.* (M == -1)`). Also, let  $m_l^{(-1)}$  be column vector  $l$  of  $M_{-1}$ .

Let  $\pi_l^{+1} = m_l^{(+1)} \circ \pi$  and  $\pi_l^{-1} = m_l^{(-1)} \circ \pi$ , where  $\pi$  is the vector of specified, class prior probabilities (`Prior`).

Then, the positive and negative, scalar class prior probabilities for binary learner  $l$  are

$$\hat{\pi}_l^{(j)} = \frac{\|\pi_l^{(j)}\|_1}{\|\pi_l^{(+1)}\|_1 + \|\pi_l^{(-1)}\|_1},$$



where  $j = \{-1, 1\}$  and  $\|a\|_1$  is the one-norm of  $a$ .

- Cost — The software normalizes the  $K$ -by- $K$  cost matrix  $C$  (Cost) for each binary learner. For binary learner  $l$ , the cost of classifying a negative-class observation into the positive class is

$$c_l^{-+} = \left( \pi_l^{(-1)} \right) C \pi_l^{(+1)}.$$

Similarly, the cost of classifying a positive-class observation into the negative class is

$$c_l^{+-} = \left( \pi_l^{(+1)} \right) C \pi_l^{(-1)}.$$

The cost matrix for binary learner  $l$  is

$$C_l = \begin{bmatrix} 0 & c_l^{-+} \\ c_l^{+-} & 0 \end{bmatrix}.$$

ECOC models accommodate misclassification costs by incorporating them with class prior probabilities. If you specify `Prior` and `Cost`, then the software adjusts the class prior probabilities as follows:

$$\bar{\pi}_l^{-1} = \frac{c_l^{-+} \hat{\pi}_l^{-1}}{c_l^{-+} \hat{\pi}_l^{-1} + c_l^{+-} \hat{\pi}_l^{+1}}$$

$$\bar{\pi}_l^{+1} = \frac{c_l^{+-} \hat{\pi}_l^{+1}}{c_l^{-+} \hat{\pi}_l^{-1} + c_l^{+-} \hat{\pi}_l^{+1}}.$$

## Random Coding Design Matrices

For a given number of classes, for example,  $K$ , the software generates random coding design matrices as follows.

- 1 The software generates one of the following:
  - a Dense random — The software assigns a 1 or -1 with equal probability to each element of the  $K$ -by- $L_d$  coding design matrix, where  $L_d \approx \lceil 10 \log_2 K \rceil$ .
  - b Sparse random — The software assigns a 1 to each element of the  $K$ -by- $L_s$  coding design matrix with probability 0.25, a -1 with probability 0.25, and a 0 with probability 0.5, where  $L_s \approx \lceil 15 \log_2 K \rceil$ .

- 2 If a column does not contain at least one 1 and at least one -1, then the software removes that column.
- 3 For distinct columns  $u$  and  $v$ , if  $u = v$  or  $u = -v$ , then the software removes  $v$  from the coding design matrix.

The software randomly generates 10,000 matrices by default, and retains the matrix with the largest, minimal pairwise row distance based on the Hamming measure ([4]) given by

$$\Delta(k_1, k_2) = 0.5 \sum_{l=1}^L |m_{k_1 l} - m_{k_2 l}|,$$

where  $m_{k,l}$  is an element of coding design matrix  $j$ .

## Support Vector Storage

For linear, SVM binary learners, and for efficiency, `fitcecoc` empties the properties `Alpha`, `SupportVectorLabels`, and `SupportVectors`. `fitcecoc` lists `Beta`, rather than `Alpha`, in the model display.

To store `Alpha`, `SupportVectorLabels`, and `SupportVectors`, pass a linear, SVM template that specifies storing support vectors to `fitcecoc`. For example, enter:

```
t = templateSVM('SaveSupportVectors', 'on')
Mdl = fitcecoc(X, Y, 'Learners', t);
```

You can remove the support vectors and related values by passing the resulting `ClassificationECOC` model to `discardSupportVectors`.

## References

- [1] Allwein, E., R. Schapire, and Y. Singer. “Reducing multiclass to binary: A unifying approach for margin classifiers.” *Journal of Machine Learning Research*. Vol. 1, 2000, pp. 113–141.
- [2] Fürnkranz, Johannes, “Round Robin Classification.” *J. Mach. Learn. Res.*, Vol. 2, 2002, pp. 721–747.
- [3] Escalera, S., O. Pujol, and P. Radeva. “On the decoding process in ternary error-correcting output codes.” *IEEE Transactions on Pattern Analysis and Machine Intelligence*. Vol. 32, Issue 7, 2010, pp. 120–134.

- [4] Escalera, S., O. Pujol, and P. Radeva. “Separability of ternary codes for sparse designs of error-correcting output codes.” *Pattern Recog. Lett.*, Vol. 30, Issue 3, 2009, pp. 285–297.

## See Also

`ClassificationECOC` | `ClassificationLinear` |  
`ClassificationPartitionedECOC` | `ClassificationPartitionedLinearECOC` |  
`CompactClassificationECOC` | `designecoc` | `fitclinear` | `loss` | `predict` |  
`statset` | `templateDiscriminant` | `templateEnsemble` | `templateKNN` |  
`templateSVM` | `templateTree`

## Topics

“Quick Start Parallel Computing for Statistics and Machine Learning Toolbox” on page 30-2

“Reproducibility in Parallel Statistical Computations” on page 30-14

“Concepts of Parallel Computing in Statistics and Machine Learning Toolbox” on page 30-8

**Introduced in R2014b**

## fitcensemble

Fit ensemble of learners for classification

### Syntax

```
Mdl = fitcensemble(Tbl, ResponseVarName)
```

```
Mdl = fitcensemble(Tbl, formula)
```

```
Mdl = fitcensemble(Tbl, Y)
```

```
Mdl = fitcensemble(X, Y)
```

```
Mdl = fitcensemble( ____, Name, Value)
```

### Description

`Mdl = fitcensemble(Tbl, ResponseVarName)` returns the trained classification ensemble model object (`Mdl`) that contains the results of boosting 100 classification trees and the predictor and response data in the table `Tbl`. `ResponseVarName` is the name of the response variable in `Tbl`. By default, `fitcensemble` uses `LogitBoost` for binary classification and `AdaBoostM2` for multiclass classification.

`Mdl = fitcensemble(Tbl, formula)` applies `formula` to fit the model to the predictor and response data in the table `Tbl`. `formula` is an explanatory model of the response and a subset of predictor variables in `Tbl` used to fit `Mdl`. For example, `'Y~X1+X2+X3'` fits the response variable `Tbl.Y` as a function of the predictor variables `Tbl.X1`, `Tbl.X2`, and `Tbl.X3`.

`Mdl = fitcensemble(Tbl, Y)` treats all variables in the table `Tbl` as predictor variables. `Y` is the array of class labels that is not in `Tbl`.

`Mdl = fitcensemble(X, Y)` uses the predictor data in the matrix `X` and the array of class labels in `Y`.

`Mdl = fitcensemble( ____, Name, Value)` uses additional options specified by one or more `Name, Value` pair arguments and any of the input arguments in the previous

syntaxes. For example, you can specify the number of learning cycles, the ensemble-aggregation method, or to implement 10-fold cross-validation.

## Examples

### Train Classification Ensemble

Create a predictive classification ensemble using all available predictor variables in the data. Then, train another ensemble using fewer predictors. Compare the in-sample predictive accuracies of the ensembles.

Load the census1994 data set.

```
load census1994
```

Train an ensemble of classification models using the entire data set and default options.

```
Mdl1 = fitcensemble(adultdata, 'salary')
```

```
Mdl1 =
```

```
classreg.learning.classif.ClassificationEnsemble
    PredictorNames: {1x14 cell}
    ResponseName: 'salary'
    CategoricalPredictors: [2 4 6 7 8 9 10 14]
    ClassNames: [<=50K >50K]
    ScoreTransform: 'none'
    NumObservations: 32561
    NumTrained: 100
    Method: 'LogitBoost'
    LearnerNames: {'Tree'}
    ReasonForTermination: 'Terminated normally after completing the requested number of
    FitInfo: [100x1 double]
    FitInfoDescription: {2x1 cell}
```

Mdl is a `ClassificationEnsemble` model. Some notable characteristics of Mdl are:

- Because two classes are represented in the data, `LogitBoost` is the ensemble-aggregation algorithm.

- Because the ensemble-aggregation method is a boosting algorithm, classification trees that allow a maximum of 10 splits compose the ensemble.
- One hundred trees compose the ensemble.

Use the classification ensemble to predict the labels of a random set of five observations from the data. Compare the predicted labels with their true values.

```
rng(1) % For reproducibility
[pX,pIdx] = datasample(adultdata,5);
label = predict(Mdl1,pX);
table(label,adultdata.salary(pIdx),'VariableNames',{'Predicted','Truth'})
```

```
ans =
```

```
5x2 table
```

Predicted	Truth
<=50K	<=50K
<=50K	<=50K
<=50K	<=50K
<=50K	<=50K
<=50K	<=50K

Train a new ensemble using age and education only.

```
Mdl2 = fitcensemble(adultdata,'salary ~ age + education');
```

Compare the resubstitution losses between Mdl1 and Mdl2.

```
rsLoss1 = resubLoss(Mdl1)
rsLoss2 = resubLoss(Mdl2)
```

```
rsLoss1 =
```

```
0.1077
```

```
rsLoss2 =
```

0.2037

The in-sample misclassification rate for the ensemble that uses all predictors is lower.

### Estimate Generalization Error of Boosting Ensemble

Estimate the generalization error of ensemble of boosted classification trees.

Load the ionosphere data set.

```
load ionosphere
```

Cross-validate an ensemble of classification trees using AdaBoostM1 and 10-fold cross-validation. Specify that each tree should be split a maximum of five times using a decision tree template.

```
rng(5); % For reproducibility
t = templateTree('MaxNumSplits',5);
Mdl = fitcensemble(X,Y,'Method','AdaBoostM1','Learners',t,'CrossVal','on');
```

Mdl is a ClassificationPartitionedEnsemble model.

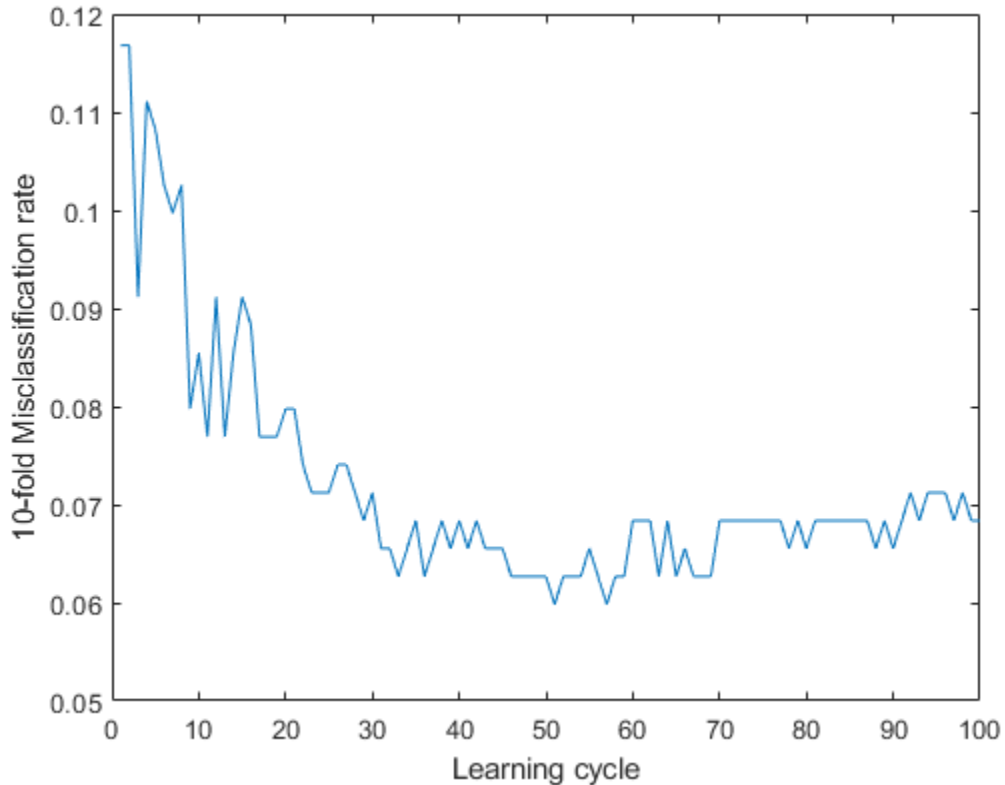
Plot the cumulative, 10-fold cross-validated, misclassification rate. Display the estimated generalization error of the ensemble.

```
kflc = kfoldLoss(Mdl,'Mode','cumulative');
figure;
plot(kflc);
ylabel('10-fold Misclassification rate');
xlabel('Learning cycle');
```

```
estGenError = kflc(end)
```

```
estGenError =
```

0.0684



`kfoldLoss` returns the generalization error by default. However, plotting the cumulative loss allows you to monitor how the loss changes as weak learners accumulate in the ensemble.

The ensemble achieves a misclassification rate of around 0.06 after accumulating about 50 weak learners. Then, the misclassification rate increase slightly as more weak learners enter the ensemble.

If you are satisfied with the generalization error of the ensemble, then, to create a predictive model, train the ensemble again using all of the settings except cross-



validation. However, it is good practice to tune hyperparameters, such as the maximum number of decision splits per tree and the number of learning cycles.

## Optimize Classification Ensemble

This example shows how to optimize hyperparameters automatically using `fitcensemble`. The example uses the `ionosphere` data.

Load the data.

```
load ionosphere
```

Find hyperparameters that minimize five-fold cross-validation loss by using automatic hyperparameter optimization.

For reproducibility, set the random seed and use the 'expected-improvement-plus' acquisition function.

```
rng default
Mdl = fitcensemble(X,Y,'OptimizeHyperparameters','auto',...
    'HyperparameterOptimizationOptions',struct('AcquisitionFunctionName',...
    'expected-improvement-plus'))
```

Iter	Eval result	Objective	Objective runtime	BestSoFar (observed)	BestSoFar (estim.)	Method
1	Best	0.10256	3.0672	0.10256	0.10256	RUSBoost
2	Best	0.062678	24.409	0.062678	0.064264	LogitBoost
3	Accept	0.099715	25.261	0.062678	0.062688	AdaBoostM1
4	Accept	0.065527	4.4553	0.062678	0.062681	Bag
5	Accept	0.065527	21.433	0.062678	0.062695	LogitBoost
6	Accept	0.068376	18.777	0.062678	0.062693	GentleBoost
7	Accept	0.076923	43.173	0.062678	0.063613	GentleBoost
8	Accept	0.068376	43.073	0.062678	0.063878	LogitBoost
9	Accept	0.068376	28.311	0.062678	0.065468	LogitBoost
10	Accept	0.17379	1.123	0.062678	0.064692	LogitBoost
11	Accept	0.065527	12.133	0.062678	0.064854	Bag
12	Accept	0.076923	2.0973	0.062678	0.062571	GentleBoost
13	Accept	0.082621	1.7708	0.062678	0.064919	GentleBoost
14	Accept	0.065527	56.102	0.062678	0.06557	Bag
15	Accept	0.079772	41.041	0.062678	0.064962	GentleBoost

16	Accept	0.35897	1.7323	0.062678	0.062491	Bag
17	Accept	0.35897	2.8461	0.062678	0.062483	AdaBoostM1
18	Accept	0.10826	96.11	0.062678	0.062484	RUSBoost
19	Accept	0.64103	3.5226	0.062678	0.062469	RUSBoost
20	Accept	0.088319	42.915	0.062678	0.062473	RUSBoost
=====						
Iter	Eval	Objective	Objective	BestSoFar	BestSoFar	Method
	result		runtime	(observed)	(estim.)	
=====						
21	Accept	0.074074	13.765	0.062678	0.062473	GentleBoost
22	Accept	0.10256	1.6694	0.062678	0.062472	AdaBoostM1
23	Accept	0.16524	5.7288	0.062678	0.062371	LogitBoost
24	Accept	0.17379	1.4733	0.062678	0.062586	LogitBoost
25	Accept	0.065527	22.941	0.062678	0.062582	LogitBoost
26	Accept	0.11396	58.009	0.062678	0.062585	RUSBoost
27	Accept	0.079772	2.2318	0.062678	0.063407	GentleBoost
28	Accept	0.099715	1.6986	0.062678	0.062569	RUSBoost
29	Accept	0.068376	20.056	0.062678	0.063914	LogitBoost
30	Accept	0.065527	68.07	0.062678	0.0629	Bag

Optimization completed.

MaxObjectiveEvaluations of 30 reached.

Total function evaluations: 30

Total elapsed time: 765.6832 seconds.

Total objective function evaluation time: 668.9945

Best observed feasible point:

Method	NumLearningCycles	LearnRate	MinLeafSize
LogitBoost	206	0.96537	33

Observed objective function value = 0.062678

Estimated objective function value = 0.0629

Function evaluation time = 24.4087

Best estimated feasible point (according to models):

Method	NumLearningCycles	LearnRate	MinLeafSize
LogitBoost	206	0.96537	33

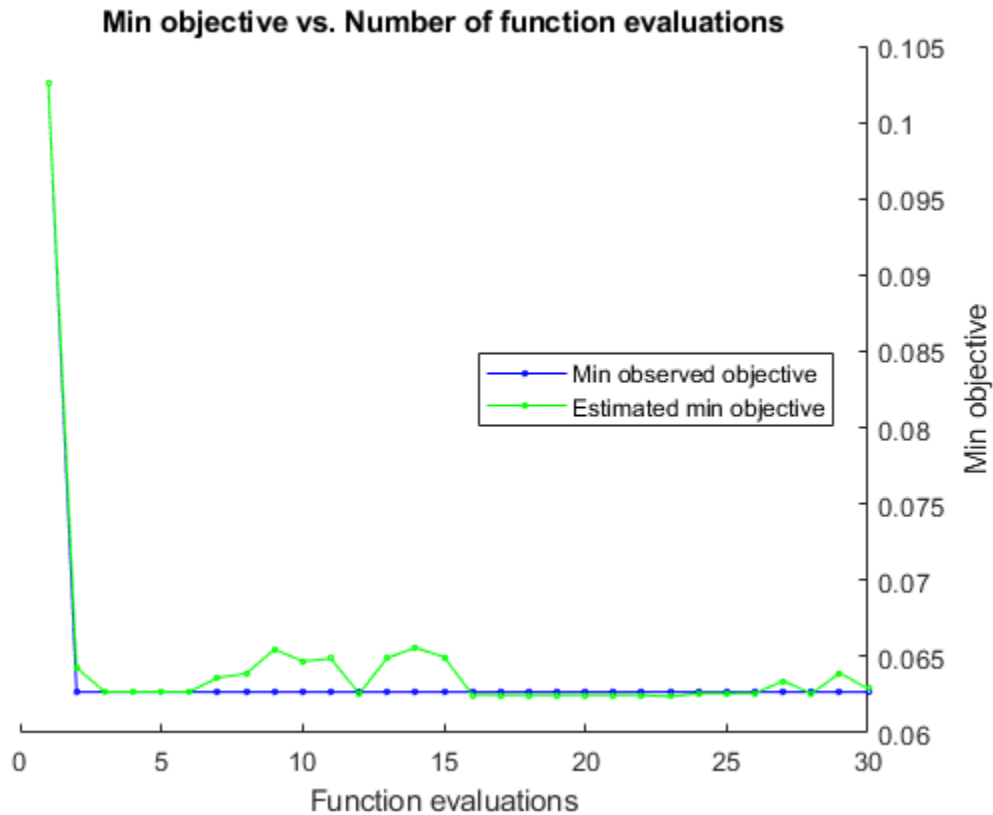
Estimated objective function value = 0.0629

---

Estimated function evaluation time = 24.232

Mdl =

```
classreg.learning.classif.ClassificationEnsemble
    ResponseName: 'Y'
    CategoricalPredictors: []
    ClassNames: {'b' 'g'}
    ScoreTransform: 'none'
    NumObservations: 351
    HyperparameterOptimizationResults: [1x1 BayesianOptimization]
    NumTrained: 206
    Method: 'LogitBoost'
    LearnerNames: {'Tree'}
    ReasonForTermination: 'Terminated normally after completing the request'
    FitInfo: [206x1 double]
    FitInfoDescription: {2x1 cell}
```



The optimization searched over the methods for binary classification, over `NumLearningCycles`, over the `LearnRate` for applicable methods, and over the tree learner `MinLeafSize`. The output is the ensemble classifier with the minimum estimated cross-validation loss.

### Create Predictive Ensemble Using Cross-Validation

One way to create an ensemble of boosted classification trees that has satisfactory predictive performance is by tuning the decision tree-complexity level using cross-validation. While searching for an optimal complexity level, tune the learning rate to minimize the number of learning cycles.

Load the ionosphere data set.

```
load ionosphere
```

To search for the optimal tree-complexity level:

- 1 Cross-validate a set of ensembles. Exponentially increase the tree-complexity level for subsequent ensembles from decision stump (one split) to at most  $n - 1$  splits.  $n$  is the sample size. Also, vary the learning rate for each ensemble between 0.1 to 1.
- 2 Estimate the cross-validated misclassification rate of each ensemble.
- 3 For tree-complexity level  $j$ ,  $j = 1 \dots J$ , compare the cumulative, cross-validated misclassification rate of the ensembles by plotting them against number of learning cycles. Plot separate curves for each learning rate on the same figure.
- 4 Choose the curve that achieves the minimal misclassification rate, and note the corresponding learning cycle and learning rate.

Cross-validate a deep classification tree and a stump. These classification trees serve as benchmarks.

```
rng(1); % For reproducibility
MdlDeep = fitctree(X,Y,'CrossVal','on','MergeLeaves','off',...
    'MinParentSize',1);
MdlStump = fitctree(X,Y,'MaxNumSplits',1,'CrossVal','on');
```

Cross-validate an ensemble of 150 boosted classification trees using 5-fold cross-validation. Using a tree template, vary the maximum number of splits using the values in the sequence  $\{3^0, 3^1, \dots, 3^m\}$ .  $m$  is such that  $2^m$  is no greater than  $n - 1$ . For each variant, adjust the learning rate using each value in the set  $\{0.1, 0.25, 0.5, 1\}$ ;

```
n = size(X,1);
m = floor(log(n - 1)/log(3));
learnRate = [0.1 0.25 0.5 1];
numLR = numel(learnRate);
maxNumSplits = 3.^(0:m);
numMNS = numel(maxNumSplits);
numTrees = 150;
Mdl = cell(numMNS,numLR);

for k = 1:numLR;
    for j = 1:numMNS;
        t = templateTree('MaxNumSplits',maxNumSplits(j));
        Mdl{j,k} = fitcensemble(X,Y,'NumLearningCycles',numTrees,...
```

```

        'Learners',t,'KFold',5,'LearnRate',learnRate(k));
    end;
end;

```

Estimate the cumulative, cross-validated misclassification rate for each ensemble and the classification trees serving as benchmarks.

```

kflAll = @(x) kfoldLoss(x, 'Mode', 'cumulative');
errorCell = cellfun(kflAll,Mdl, 'Uniform', false);
error = reshape(cell2mat(errorCell), [numTrees numel(maxNumSplits) numel(learnRate)]);
errorDeep = kfoldLoss(MdlDeep);
errorStump = kfoldLoss(MdlStump);

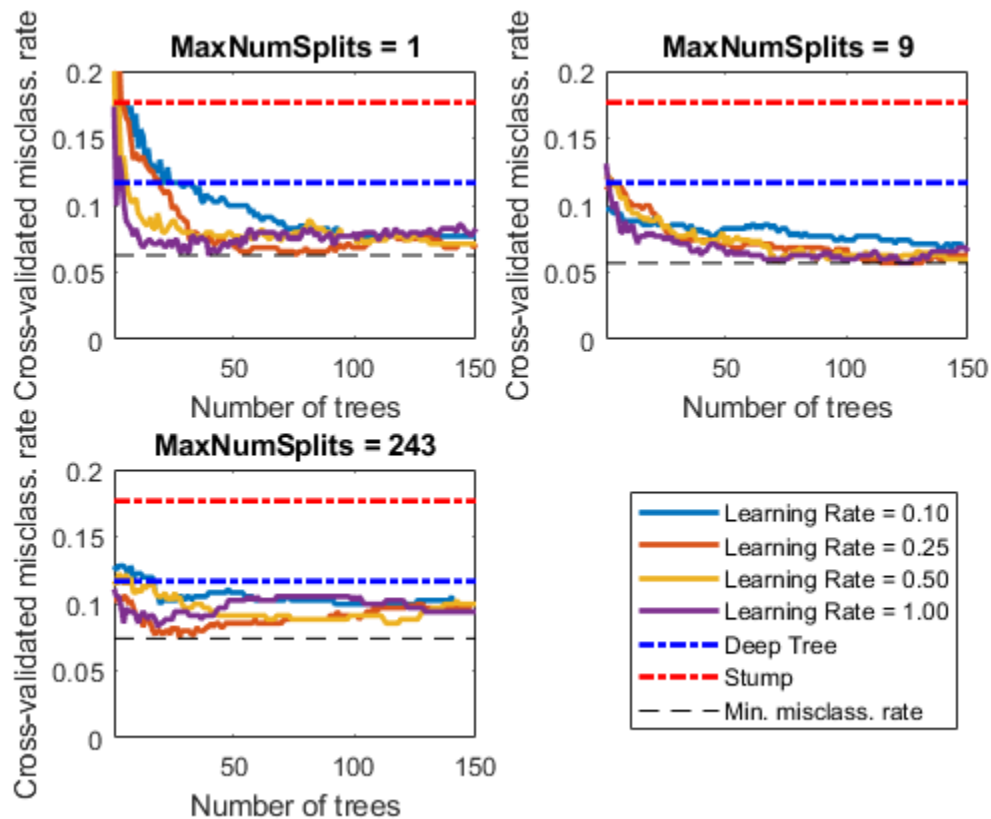
```

Plot how the cross-validated misclassification rate behaves as the number of trees in the ensemble increases. Plot the curves with respect to learning rate on the same plot, and plot separate plots for varying tree-complexity levels. Choose a subset of tree complexity levels to plot.

```

mnsPlot = [1 round(numel(maxNumSplits)/2) numel(maxNumSplits)];
figure;
for k = 1:3;
    subplot(2,2,k);
    plot(squeeze(error(:,mnsPlot(k),:)), 'LineWidth',2);
    axis tight;
    hold on;
    h = gca;
    plot(h.XLim,[errorDeep errorDeep], '-.b', 'LineWidth',2);
    plot(h.XLim,[errorStump errorStump], '-.r', 'LineWidth',2);
    plot(h.XLim,min(min(error(:,mnsPlot(k),:)))*[1 1], '--k');
    h.YLim = [0 0.2];
    xlabel 'Number of trees';
    ylabel 'Cross-validated misclass. rate';
    title(sprintf('MaxNumSplits = %0.3g', maxNumSplits(mnsPlot(k))));
    hold off;
end;
hL = legend([cellstr(num2str(learnRate),'Learning Rate = %0.2f');...
            'Deep Tree';'Stump';'Min. misclass. rate']);
hL.Position(1) = 0.6;

```



Each curve contains a minimum cross-validated misclassification rate occurring at the optimal number of trees in the ensemble.

Identify the maximum number of splits, number of trees, and learning rate that yields the lowest misclassification rate overall.

```
[minErr,minErrIdxLin] = min(error(:));
[idxNumTrees,idxMNS,idxLR] = ind2sub(size(error),minErrIdxLin);

fprintf('\nMin. misclass. rate = %0.5f',minErr)
fprintf('\nOptimal Parameter Values:\nNum. Trees = %d',idxNumTrees);
fprintf('\nMaxNumSplits = %d\nLearning Rate = %0.2f\n',...
    maxNumSplits(idxMNS),learnRate(idxLR))
```

```
Min. misclass. rate = 0.05413
Optimal Parameter Values:
Num. Trees = 47
MaxNumSplits = 3
Learning Rate = 0.25
```

Create a predictive ensemble based on the optimal hyperparameters and the entire training set.

```
tFinal = templateTree('MaxNumSplits',maxNumSplits(idxMNS));
MdlFinal = fitcensemble(X,Y,'NumLearningCycles',idxNumTrees,...
    'Learners',tFinal,'LearnRate',learnRate(idxLR))
```

```
MdlFinal =
```

```
classreg.learning.classif.ClassificationEnsemble
    ResponseName: 'Y'
    CategoricalPredictors: []
        ClassNames: {'b' 'g'}
        ScoreTransform: 'none'
        NumObservations: 351
        NumTrained: 47
            Method: 'LogitBoost'
            LearnerNames: {'Tree'}
    ReasonForTermination: 'Terminated normally after completing the requested number o
        FitInfo: [47x1 double]
    FitInfoDescription: {2x1 cell}
```

MdlFinal is a ClassificationEnsemble. To predict the whether a radar return is good given predictor data, pass the predictor data and MdlFinal to predict.

- “Supervised Learning Workflow and Algorithms” on page 18-2
- “Framework for Ensemble Learning” on page 18-41

## Input Arguments

**Tb1** — Sample data

table



Sample data used to train the model, specified as a table. Each row of `Tbl` corresponds to one observation, and each column corresponds to one predictor variable. `Tbl` can contain one additional column for the response variable. Multi-column variables and cell arrays other than cell arrays of character vectors are not allowed.

- If `Tbl` contains the response variable and you want to use all remaining variables as predictors, then specify the response variable using `ResponseVarName`.
- If `Tbl` contains the response variable, and you want to use a subset of the remaining variables only as predictors, then specify a formula using `formula`.
- If `Tbl` does not contain the response variable, then specify the response data using `Y`. The length of response variable and the number of rows of `Tbl` must be equal.

---

**Note** To save memory and execution time, supply `X` and `Y` instead of `Tbl`.

---

Data Types: `table`

#### **ResponseVarName** — Response variable name

name of response variable in `Tbl`

Response variable name, specified as the name of the response variable in `Tbl`.

You must specify `ResponseVarName` as a character vector. For example, if `Tbl.Y` is the response variable, then specify `ResponseVarName` as `'Y'`. Otherwise, `fitcensemble` treats all columns of `Tbl` as predictor variables.

The response variable must be a categorical or character array, logical or numeric vector, or cell array of character vectors. If the response variable is a character array, then each element must correspond to one row of the array.

For classification, you can specify the order of the classes using the `ClassNames` name-value pair argument. Otherwise, `fitcensemble` determines the class order, and stores it in the `Mdl.ClassNames`.

Data Types: `char`

#### **formula** — Explanatory model of response and subset of predictor variables

character vector

Explanatory model of the response and a subset of the predictor variables, specified as a character vector in the form of `'Y~X1+X2+X3'`. In this form, `Y` represents the response

variable, and `X1`, `X2`, and `X3` represent the predictor variables. The variables must be variable names in `Tbl` (`Tbl.Properties.VariableNames`).

To specify a subset of variables in `Tbl` as predictors for training the model, use a formula. If you specify a formula, then the software does not use any variables in `Tbl` that do not appear in `formula`.

Data Types: `char`

### **x** — Predictor data

numeric matrix

Predictor data, specified as numeric matrix.

Each row corresponds to one observation, and each column corresponds to one predictor variable.

The length of `Y` and the number of rows of `X` must be equal.

To specify the names of the predictors in the order of their appearance in `X`, use the `PredictorNames` name-value pair argument.

Data Types: `single` | `double`

### **y** — Response data

categorical array | character array | logical vector | numeric vector | cell array of character vectors

Response data, specified as a categorical or character array, logical or numeric vector, or cell array of character vectors. Each entry in `Y` is the response to or label for the observation in the corresponding row of `X` or `Tbl`. The length of `Y` and the number of rows of `X` or `Tbl` must be equal. If the response variable is a character array, then each element must correspond to one row of the array.

You can specify the order of the classes using the `ClassNames` name-value pair argument. Otherwise, `fitcensemble` determines the class order, and stores it in the `Mdl.ClassNames`.

Data Types: `char` | `cell` | `categorical` | `logical` | `single` | `double`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

---

**Note** You cannot use any cross-validation name-value pair along with `OptimizeHyperparameters`. You can modify the cross-validation for `OptimizeHyperparameters` only by using the `HyperparameterOptimizationOptions` name-value pair.

---

Example: `'CrossVal', 'on', 'LearningRate', 0.05` specifies to implement 10-fold cross-validation and to use 0.05 as the learning rate.

### General Ensemble Options

#### Method — Ensemble-aggregation method

`'AdaBoostM1' | 'LogitBoost' | 'GentleBoost' | 'RobustBoost' | 'LPBoost' | 'TotalBoost' | 'RUSBoost' | 'Subspace' | 'Bag' | 'AdaBoostM2' | ...`

Ensemble-aggregation method, specified as the comma separated pair consisting of `'Method'` and a character vector such as `'RUSBoost'`.

Value	Description	Classification Problem Support	Additional Requirements
<code>'AdaBoostM1'</code>	Adaptive boosting	Binary only	-
<code>'AdaBoostM2'</code>	Adaptive boosting	Multiclass only	-
<code>'Bag'</code>	Bootstrap aggregating (e.g., random forest)	Binary and multiclass	-
<code>'GentleBoost'</code>	Gentle adaptive boosting	Binary only	-
<code>'LogitBoost'</code>	Adaptive logistic regression	Binary only	-

Value	Description	Classification Problem Support	Additional Requirements
'LPBoost'	Linear programming boosting	Binary and multiclass	Optimization Toolbox license
'RobustBoost'	Robust boosting	Binary only	Optimization Toolbox license
'RUSBoost'	Random undersampling boosting	Binary and multiclass	-
'Subspace'	Random subspace	Binary and multiclass	-
'TotalBoost'	Totally corrective boosting	Binary and multiclass	Optimization Toolbox license

For binary classification problems, the default is 'LogitBoost'. Otherwise, the default is 'AdaBoostM2'.

Example: 'Method', 'Bag'

Data Types: char

### **NumLearningCycles** — Number of ensemble learning cycles

100 (default) | positive integer | 'AllPredictorCombinations'

Number of ensemble learning cycles, specified as a positive integer or 'AllPredictorCombinations'.

- If you specify a positive integer, then, at every learning cycle, the software trains one weak learner for every template object in `Learners`. Consequently, the software trains `NumLearningCycles* numel(Learners)` learners.
- If you specify 'AllPredictorCombinations', then set `Method` to 'Subspace' and specify one learner only for `Learners`. With these settings, the software trains learners for all possible combinations of predictors taken `NPredToSample` at a time. Consequently, the software trains `nchoosek(size(X,2), NPredToSample)` learners.

The software composes the ensemble using all trained learners and stores them in `Mdl.Trained`.

For more details, see “Tips” on page 32-1963.

Example: `'NumLearningCycles', 500`

Data Types: `single` | `double` | `char`

### Learners — Weak learners to use in ensemble

weak-learner name | weak-learner template object | cell vector of weak-learner template objects

Weak learners to use in the ensemble, specified as a weak-learner name, weak-learner template object, or cell vector of weak-learner template objects.

Weak Learner	Weak-Learner Name	Template Object Creation Function	Method Settings
Discriminant analysis	'discriminant'	templateDiscriminant	Recommended for 'Subspace'
$k$ nearest neighbors	'knn'	templateKNN	For 'Subspace' only
Decision tree	'tree'	templateTree	All methods except 'Subspace'

The defaults are:

- 'knn' if 'Method' is 'Subspace'
- 'tree' if 'Method' is 'Bag'
- `templateTree('MaxNumSplits', 10)` if 'Method' is any boosting method

`fitcensemble` sets the maximum number of splits for decision tree weak learners to 10 when both of these conditions occur:

- You do not set a value for the 'MaxNumSplits' property of the decision tree weak learners.
- You use a boosting algorithm as the ensemble-aggregation method for `fitcensemble`.

For example, if you specify `'Learners', templateTree()` and `'Method', 'AdaBoostM1'`, then `fitcensemble` sets the maximum number of splits of the decision tree weak learners to 10.

Otherwise, `fitcensemble` defers to the corresponding template object to choose any required default values for the 'Learners' name-value pair argument.

For details on the number of learners to train, see `NumLearningCycles` and “Tips” on page 32-1963.

Example: For an ensemble composed of two types of classification trees, supply `{t1 t2}`, where `t1` and `t2` are classification tree templates.

Data Types: `char`

### **NPrint** — Printout frequency

`'off'` (default) | positive integer

Printout frequency, specified as the comma-separated pair consisting of `'NPrint'` and a positive integer or `'off'`.

To track the number of *weak learners* or *folds* that `fitcensemble` trained so far, specify a positive integer. That is, if you specify the positive integer *m*:

- Without also specifying any cross-validation option (for example, `CrossVal`), then `fitcensemble` displays a message to the command line every time it completes training *m* weak learners.
- And a cross-validation option, then `fitcensemble` displays a message to the command line every time it finishes training *m* folds.

If you specify `'off'`, then `fitcensemble` does not display a message when it completes training weak learners.

---

**Tip** When training an ensemble of many weak learners on a large data set, specify a positive integer for `NPrint`.

---

Example: `'NPrint',5`

Data Types: `single` | `double` | `char`

### **CategoricalPredictors** — Categorical predictors list

vector of positive integers | logical vector | character matrix | cell array of character vectors | `'all'`

Categorical predictors list, specified as the comma-separated pair consisting of `'CategoricalPredictors'` and one of these values:

Value	Description
Vector of positive integers	An entry in the vector is the index value corresponding to the column of the predictor data (X or Tbl) that contains a categorical variable.
Logical vector	A true entry means that the corresponding column of predictor data (X or Tbl) is a categorical variable.
Character matrix	Each row of the matrix is the name of a predictor variable. The names must match the entries in PredictorNames. Pad the names with extra blanks so each row of the character matrix has the same length.
Cell array of character vectors	Each element in the array is the name of a predictor variable. The names must match the entries in PredictorNames.
'all'	All predictors are categorical.

Specification of `CategoricalPredictors` is appropriate if:

- 'Learners' is 'tree'.
- 'Learners' is 'knn' when all predictors are categorical.

By default, if the predictor data is in a table (Tbl), `fitcensemble` assumes that a variable is categorical if it contains logical values, categorical values, or a cell array of character vectors. If the predictor data is a matrix (X), `fitcensemble` assumes all predictors are continuous. To identify any categorical predictors when the data is a matrix, use the '`CategoricalPredictors`' name-value pair argument.

Example: '`CategoricalPredictors`', 'all'

Data Types: `single` | `double` | `logical` | `char` | `cell`

### **PredictorNames** — Predictor variable names

cell array of unique character vectors

Predictor variable names, specified as the comma-separated pair consisting of '`PredictorNames`' and a cell array of unique character vectors. The functionality of '`PredictorNames`' depends on the way you supply the training data.

- If you supply X and Y, then you can use '`PredictorNames`' to give the predictor variables in X names.

- The order of the names in `PredictorNames` must correspond to the column order of `X`. That is, `PredictorNames{1}` is the name of `X(:,1)`, `PredictorNames{2}` is the name of `X(:,2)`, and so on. Also, `size(X,2)` and `numel(PredictorNames)` must be equal.
- By default, `PredictorNames` is `{'x1','x2',...}`.
- If you supply `Tbl`, then you can use `'PredictorNames'` to choose which predictor variables to use in training. That is, `fitcensemble` uses the predictor variables in `PredictorNames` and the response only in training.
  - `PredictorNames` must be a subset of `Tbl.Properties.VariableNames` and cannot include the name of the response variable.
  - By default, `PredictorNames` contains the names of all predictor variables.
  - It good practice to specify the predictors for training using one of `'PredictorNames'` or `formula` only.

Example: `'PredictorNames',`  
`{'SepalLength','SepalWidth','PedalLength','PedalWidth'}`

Data Types: `cell`

### **ResponseName — Response variable name**

`'Y'` (default) | character vector

Response variable name, specified as the comma-separated pair consisting of `'ResponseName'` and a character vector.

- If you supply `Y`, then you can use `'ResponseName'` to specify a name for the response variable.
- If you supply `ResponseVarName` or `formula`, then you cannot use `'ResponseName'`.

Example: `'ResponseName','response'`

Data Types: `char`

### **Cross-Validation Options**

#### **CrossVal — Cross-validation flag**

`'off'` (default) | `'on'`

Cross-validation flag, specified as the comma-separated pair consisting of `'Crossval'` and `'on'` or `'off'`.



If you specify 'on', then the software implements 10-fold cross-validation.

To override this cross-validation setting, use one of these name-value pair arguments: `CVPartition`, `Holdout`, `KFold`, or `Leaveout`. To create a cross-validated model, you can use one cross-validation name-value pair argument at a time only.

Alternatively, cross-validate later by passing `Mdl` to `crossval` or `crossval`.

Example: 'Crossval', 'on'

Data Types: char

### **CVPartition — Cross-validation partition**

[ ] (default) | `cvpartition` partition object

Cross-validation partition, specified as the comma-separated pair consisting of 'CVPartition' and a `cvpartition` partition object as created by `cvpartition`. The partition object specifies the type of cross-validation, and also the indexing for training and validation sets.

To create a cross-validated model, you can use one of these four name-value pair arguments only: `CVPartition`, `Holdout`, `KFold`, or `Leaveout`.

Example: Suppose you create a random partition for 5-fold cross-validation on 500 observations by using `cvp = cvpartition(500, 'KFold', 5)`. Then, you can specify the cross-validation partition for `fitcecoc` as 'CVPartition', `cvp`.

### **Holdout — Fraction of data for holdout validation**

scalar value in the range (0,1)

Fraction of data used for holdout validation, specified as the comma-separated pair consisting of 'Holdout' and a scalar value in the range (0,1). If you specify 'Holdout',  $p$ , then the software:

- 1 Randomly reserves  $p*100\%$  of the data as validation data, and trains the model using the rest of the data
- 2 Stores the compact, trained model in the `Trained` property of the cross-validated model.

To create a cross-validated model, you can use one of these four name-value pair arguments only: `CVPartition`, `Holdout`, `KFold`, or `Leaveout`.

Example: 'Holdout', 0.1

Data Types: `double` | `single`

### **KFold** — Number of folds

10 (default) | positive integer value greater than 1

Number of folds to use in a cross-validated model, specified as the comma-separated pair consisting of 'KFold' and a positive integer value greater than 1. If you specify, e.g., 'KFold',  $k$ , then the software:

- 1 Randomly partitions the data into  $k$  sets
- 2 For each set, reserves the set as validation data, and trains the model using the other  $k - 1$  sets
- 3 Stores the  $k$  compact, trained models in the cells of a  $k$ -by-1 cell vector in the `Trained` property of the cross-validated model.

To create a cross-validated model, you can use one of these four name-value pair arguments only: `CVPartition`, `Holdout`, `KFold`, or `Leaveout`.

Example: 'KFold', 5

Data Types: `single` | `double`

### **Leaveout** — Leave-one-out cross-validation flag

'off' (default) | 'on'

Leave-one-out cross-validation flag, specified as the comma-separated pair consisting of 'Leaveout' and 'on' or 'off'. If you specify 'Leaveout', 'on', then, for each of the  $n$  observations, where  $n$  is `size(Mdl.X, 1)`, the software:

- 1 Reserves the observation as validation data, and trains the model using the other  $n - 1$  observations
- 2 Stores the  $n$  compact, trained models in the cells of an  $n$ -by-1 cell vector in the `Trained` property of the cross-validated model.

To create a cross-validated model, you can use one of these four name-value pair arguments only: `CVPartition`, `Holdout`, `KFold`, or `Leaveout`.

Example: 'Leaveout', 'on'

Data Types: `char`

## Other Classification Options

### **ClassNames** — Names of classes to use for training

categorical array | cell array of character vectors | character array | logical vector | numeric vector

Names of classes to use for training, specified as the comma-separated pair consisting of 'ClassNames' and a categorical or character array, logical or numeric vector, or cell array of character vectors. ClassNames must be the same data type as *Y*.

If ClassNames is a character array, then each element must correspond to one *row* of the array.

Use ClassNames to:

- Order the classes during training.
- Specify the order of any input or output argument dimension that corresponds to the class order. For example, use ClassNames to specify the order of the dimensions of *Cost* or the column order of classification scores returned by *predict*.
- Select a subset of classes for training. For example, suppose that the set of all distinct class names in *Y* is {'a', 'b', 'c'}. To train the model using observations from classes 'a' and 'c' only, specify 'ClassNames', {'a', 'c'}.

The default is the set of all distinct class names in *Y*.

Example: 'ClassNames', {'b', 'g'}

Data Types: categorical | char | logical | single | double | cell

### **Cost** — Misclassification cost

square matrix | structure array

Misclassification cost, specified as the comma-separated pair consisting of 'Cost' and a square matrix or structure. If you specify:

- The square matrix *Cost*, then  $Cost(i, j)$  is the cost of classifying a point into class *j* if its true class is *i*. That is, the rows correspond to the true class and the columns correspond to the predicted class. To specify the class order for the corresponding rows and columns of *Cost*, also specify the ClassNames name-value pair argument.
- The structure *S*, then it must have two fields:

- `S.ClassNames`, which contains the class names as a variable of the same data type as `Y`
- `S.ClassificationCosts`, which contains the cost matrix with rows and columns ordered as in `S.ClassNames`

The default is  $\text{ones}(K) - \text{eye}(K)$ , where  $K$  is the number of distinct classes.

---

**Note** `fitcensemble` uses `Cost` to adjust the prior class probabilities specified in `Prior`. Then, `fitcensemble` uses the adjusted prior probabilities for training and resets the cost matrix to its default.

---

Example: `'Cost',[0 1 2 ; 1 0 2; 2 2 0]`

Data Types: `double` | `single` | `struct`

#### **Prior** — Prior probabilities

`'empirical'` (default) | `'uniform'` | numeric vector | structure array

Prior probabilities for each class, specified as the comma-separated pair consisting of `'Prior'` and a value in this table.

Value	Description
<code>'empirical'</code>	The class prior probabilities are the class relative frequencies in <code>Y</code> .
<code>'uniform'</code>	All class prior probabilities are equal to $1/K$ , where $K$ is the number of classes.
numeric vector	Each element is a class prior probability. Order the elements according to <code>SVMModel.ClassNames</code> or specify the order using the <code>ClassNames</code> name-value pair argument. The software normalizes the elements such that they sum to 1.

Value	Description
structure array	<p>A structure <i>S</i> with two fields:</p> <ul style="list-style-type: none"> <li>• <i>S.ClassNames</i> contains the class names as a variable of the same type as <i>Y</i>.</li> <li>• <i>S.ClassProbs</i> contains a vector of corresponding prior probabilities. The software normalizes the elements such that they sum to 1.</li> </ul>

fitcensemble normalizes the prior probabilities in *Prior* to sum to 1.

Example: `struct('ClassNames',  
{'setosa','versicolor','virginica'}), 'ClassProbs',1:3)`

Data Types: char | double | single | struct

### ScoreTransform — Score transformation

'none' (default) | character vector | function handle

Score transformation, specified as the comma-separated pair consisting of 'ScoreTransform' and either a character vector or a function handle.

This table summarizes the available character vectors.

Value	Description
'doublelogit'	$1/(1 + e^{-2x})$
'invlogit'	$\log(x / (1-x))$
'ismax'	Set the score for the class with the largest score to 1, and set the scores for all other classes to 0.
'logit'	$1/(1 + e^{-x})$
'none' or 'identity'	$x$ (no transformation)
'sign'	-1 for $x < 0$ 0 for $x = 0$ 1 for $x > 0$
'symmetric'	$2x - 1$

Value	Description
'symmetricismax'	Set the score for the class with the largest score to 1, and set the scores for all other classes to -1.
'symmetriclogit'	$2/(1 + e^{-x}) - 1$

For a MATLAB function or a function you define, use its function handle. The function handle must accept a matrix (the original scores) and return a matrix of the same size (the transformed scores).

Example: 'ScoreTransform', 'logit'

Data Types: char | function\_handle

### Weights — Observation weights

numeric vector of positive values | character vector

Observation weights, specified as the comma-separated pair consisting of 'Weights' and a numeric vector of positive values or name of a variable in Tbl. The software weighs the observations in each row of X or Tbl with the corresponding value in Weights. The size of Weights must equal the number of rows of X or Tbl.

If you specify the input data as a table Tbl, then Weights can be the name of a variable in Tbl that contains a numeric vector. In this case, you must specify Weights as a character vector. For example, if the weights vector W is stored as Tbl.W, then specify it as 'W'. Otherwise, the software treats all columns of Tbl, including W, as predictors or the response when training the model.

The software normalizes Weights to sum up to the value of the prior probability in the respective class.

By default, Weights is ones ( $n, 1$ ), where  $n$  is the number of observations in X or Tbl.

Data Types: double | single | char

### Sampling Options for Boosting Methods and Bagging

#### FResample — Fraction of training set to resample

1 (default) | positive scalar in (0,1]

Fraction of the training set to resample for every weak learner, specified as the comma-separated pair consisting of 'FResample' and a positive scalar in (0,1].

To use 'FResample', specify 'bag' for Method or set Resample to 'on'.

Example: 'FResample', 0.75

Data Types: single | double

### **Replace — Flag indicating to sample with replacement**

'on' (default) | 'off'

Flag indicating sampling with replacement, specified as the comma-separated pair consisting of 'Replace' and 'off' or 'on'.

- For 'on', the software samples the training observations with replacement.
- For 'off', the software samples the training observations without replacement. If you set Resample to 'on', then the software samples training observations assuming uniform weights. If you also specify a boosting method, then the software boosts by reweighting observations.

Unless you set Method to 'bag' or set Resample to 'on', Replace has no effect.

Example: 'Replace', 'off'

Data Types: char

### **Resample — Flag indicating to resample**

'off' | 'on'

Flag indicating to resample, specified as the comma-separated pair consisting of 'Resample' and 'off' or 'on'.

- If Method is any boosting method, then:
  - 'Resample', 'on' specifies to sample training observations using updated weights as the multinomial sampling probabilities.
  - 'Resample', 'off' specifies to reweight observations at every learning iteration. This setting is the default.
- If Method is 'bag', then 'Resample' must be 'on'. The software resamples a fraction of the training observations (see FResample) with or without replacement (see Replace).

**AdaBoostM1, AdaBoostM2, LogitBoost, and GentleBoost Method Options****LearnRate** — Learning rate for shrinkage

1 (default) | numeric scalar in (0,1]

Learning rate for shrinkage, specified as the comma-separated pair consisting of a numeric scalar in the interval (0,1].

To train an ensemble using shrinkage, set `LearnRate` to a value less than 1, for example, 0.1 is a popular choice. Training an ensemble using shrinkage requires more learning iterations, but often achieves better accuracy.

Example: `'LearnRate', 0.1`

Data Types: `single` | `double`

**RUSBoost Method Options****LearnRate** — Learning rate for shrinkage

1 (default) | numeric scalar in (0,1]

Learning rate for shrinkage, specified as the comma-separated pair consisting of a numeric scalar in the interval (0,1].

To train an ensemble using shrinkage, set `LearnRate` to a value less than 1, for example, 0.1 is a popular choice. Training an ensemble using shrinkage requires more learning iterations, but often achieves better accuracy.

Example: `'LearnRate', 0.1`

Data Types: `single` | `double`

**RatioToSmallest** — Sampling proportion with respect to lowest-represented class

positive numeric scalar | numeric vector of positive values

Sampling proportion with respect to the lowest-represented class, specified as the comma-separated pair consisting of `'RatioToSmallest'` and a numeric scalar or numeric vector of positive values with length equal to the number of distinct classes in the training data.

Suppose that there are  $K$  classes in the training data and the lowest-represented class has  $m$  observations in the training data.



- If you specify the positive numeric scalar  $s$ , then `fitcensemble` samples  $s*m$  observations from each class, that is, it uses the same sampling proportion for each class. For more details, see “Algorithms” on page 32-2479.
- If you specify the numeric vector  $[s_1, s_2, \dots, s_K]$ , then `fitcensemble` samples  $s_i*m$  observations from class  $i$ ,  $i = 1, \dots, K$ . The elements of `RatioToSmallest` correspond to the order of the class names specified using `ClassNames` (see “Tips” on page 32-2478).

The default value is `ones(K,1)`, which specifies to sample  $m$  observations from each class.

Example: `'RatioToSmallest',[2,1]`

Data Types: `single` | `double`

#### **LPBoost and TotalBoost Method Options**

##### **MarginPrecision** — Margin precision to control convergence speed

0.1 (default) | numeric scalar in [0,1]

Margin precision to control convergence speed, specified as the comma-separated pair consisting of `'MarginPrecision'` and a numeric scalar in the interval [0,1].

`MarginPrecision` affects the number of boosting iterations required for convergence.

---

**Tip** To train an ensemble using many learners, specify a small value for `MarginPrecision`. For training using a few learners, specify a large value.

---

Example: `'MarginPrecision',0.5`

Data Types: `single` | `double`

#### **RobustBoost Method Options**

##### **RobustErrorGoal** — Target classification error

0.1 (default) | nonnegative numeric scalar

Target classification error, specified as the comma-separated pair consisting of `'RobustErrorGoal'` and a nonnegative numeric scalar. The upper bound on possible values depends on the values of `RobustMarginSigma` and `RobustMaxMargin`. However, the upper bound cannot exceed 1.

---

**Tip** For a particular training set, usually there is an optimal range for `RobustErrorGoal`. If you set it too low or too high, then the software can produce a model with poor classification accuracy. Try cross-validating to search for the appropriate value.

---

Example: `'RobustErrorGoal', 0.05`

Data Types: `single` | `double`

### **RobustMarginSigma** — Classification margin distribution spread

0.1 (default) | positive numeric scalar

Classification margin distribution spread over the training data, specified as the comma-separated pair consisting of `'RobustMarginSigma'` and a positive numeric scalar. Before specifying `RobustMarginSigma`, consult the literature on `RobustBoost`, for example, [19].

Example: `'RobustMarginSigma', 0.5`

Data Types: `single` | `double`

### **RobustMaxMargin** — Maximal classification margin

0 (default) | nonnegative numeric scalar

Maximal classification margin in the training data, specified as the comma-separated pair consisting of `'RobustMaxMargin'` and a nonnegative numeric scalar. The software minimizes the number of observations in the training data having classification margins below `RobustMaxMargin`.

Example: `'RobustMaxMargin', 1`

Data Types: `single` | `double`

### **Random Subspace Method Options**

#### **NPredToSample** — Number of predictors to sample

1 (default) | positive integer

Number of predictors to sample for each random subspace learner, specified as the comma-separated pair consisting of `'NPredToSample'` and a positive integer in the interval  $1, \dots, p$ , where  $p$  is the number of predictor variables (`size(X, 2)` or `size(Tbl, 2)`).

Data Types: `single` | `double`

## Hyperparameter Optimization

### **OptimizeHyperparameters** — Parameters to optimize

'none' (default) | 'auto' | 'all' | cell array of eligible parameter names | vector of `optimizableVariable` objects

Parameters to optimize, specified as:

- 'none' — Do not optimize.
- 'auto' — Use {'Method', 'NumLearningCycles', 'LearnRate'} along with the default parameters for the specified Learners:
  - Learners = 'tree' (default) — {'MinLeafSize'}
  - Learners = 'discriminant' — {'Delta', 'Gamma'}
  - Learners = 'knn' — {'Distance', 'NumNeighbors'}

---

**Note** For hyperparameter optimization, Learners must be a single argument, not a cell array.

---

- 'all' — Optimize all eligible parameters.
- Cell array of eligible parameter names
- Vector of `optimizableVariable` objects, typically the output of `hyperparameters`

The optimization attempts to minimize the cross-validation loss (error) for `fitcensemble` by varying the parameters. For information about cross-validation loss (albeit in a different context), see “Classification Loss” on page 32-3250. To control the cross-validation type and other aspects of the optimization, use the `HyperparameterOptimizationOptions` name-value pair.

---

**Note** `OptimizeHyperparameters` values override any values you set using other name-value pairs. For example, setting `OptimizeHyperparameters` to 'auto' causes the 'auto' values to apply.

---

The eligible parameters for `fitcensemble` are:

- Method — Depends on the number of classes.
  - Two classes — Eligible methods are 'Bag', 'GentleBoost', 'LogitBoost', 'AdaBoostM1', and 'RUSBoost'.
  - Three or more classes — Eligible methods are 'Bag', 'AdaBoostM2', and 'RUSBoost'.
- NumLearningCycles — fitcensemble searches among positive integers, by default log-scaled with range [10, 500].
- LearnRate — fitcensemble searches among positive reals, by default log-scaled with range [1e-3, 1].
- The eligible hyperparameters for the chosen Learners:

Learners	Eligible Hyperparameters Bold = Used By Default	Default Range
'discriminant'	<b>Delta</b>	Log-scaled in the range [1e-6, 1e3]
	DiscrimType	'linear', 'quadratic', 'diagLinear', 'diagQuadratic', 'pseudoLinear', and 'pseudoQuadratic'
	<b>Gamma</b>	Real values in [0, 1]
'knn'	<b>Distance</b>	'cityblock', 'chebychev', 'correlation', 'cosine', 'euclidean', 'hamming', 'jaccard', 'mahalanobis', 'minkowski', 'seuclidean', and 'spearman'
	DistanceWeight	'equal', 'inverse', and 'squaredinverse'
	Exponent	Positive values in [0.5, 3]
	<b>NumNeighbors</b>	Positive integer values log-scaled in the range [1, max(2, round(NumObservations/2))]
	Standardize	'true' and 'false'

Learners	Eligible Hyperparameters Bold = Used By Default	Default Range
'tree'	MaxNumSplits	Integers log-scaled in the range [1,max(2,NumObservations-1)]
	<b>MinLeafSize</b>	Integers log-scaled in the range [1,max(2,floor(NumObservations/2))]
	NumVariablesToSample	Integers in the range [1,max(2,NumPredictors)]
	SplitCriterion	'gdi', 'deviance', and 'twoing'

Alternatively, use hyperparameters with your chosen Learners.

```
load fisheriris % hyperparameters requires data and learner
params = hyperparameters('fitcensemble',meas,species,'Tree');
```

To see the eligible and default hyperparameters, examine params.

Set nondefault parameters by passing a vector of optimizableVariable objects that have nondefault values. For example,

```
load fisheriris
params = hyperparameters('fitcensemble',meas,species,'Tree');
params(4).Range = [1,30];
```

Pass params as the value of OptimizeHyperparameters.

By default, iterative display appears at the command line, and plots appear according to the number of hyperparameters in the optimization. For the optimization and plots, the objective function is  $\log(1 + \text{cross-validation loss})$  for regression, and the misclassification rate for classification. To control the iterative display, set the HyperparameterOptimizationOptions name-value pair, Verbose field. To control the plots, set the HyperparameterOptimizationOptions name-value pair, ShowPlots field.

For an example, see “Optimize Classification Ensemble” on page 32-1931.

Example: 'auto'

Data Types: char | cell

### HyperparameterOptimizationOptions — Options for optimization structure

Options for optimization, specified as a structure. Modifies the effect of the `OptimizeHyperparameters` name-value pair. All fields in the structure are optional.

Field Name	Values	Default
Optimizer	<ul style="list-style-type: none"> <li>'bayesopt' — Use Bayesian optimization. Internally, this setting calls <code>bayesopt</code>.</li> <li>'gridsearch' — Use grid search with <code>NumGridDivisions</code> values per dimension.</li> <li>'randomsearch' — Search at random among <code>MaxObjectiveEvaluations</code> points.</li> </ul> <p>'gridsearch' searches in a random order, using uniform sampling without replacement from the grid. After optimization, you can get a table in grid order by using the command <code>sortrows(Mdl.ParameterOptimizationResults)</code>.</p>	'bayesopt'
AcquisitionFunctionName	<ul style="list-style-type: none"> <li>'expected-improvement-per-second-plus'</li> <li>'expected-improvement'</li> <li>'expected-improvement-plus'</li> <li>'expected-improvement-per-second'</li> <li>'lower-confidence-bound'</li> <li>'probability-of-improvement'</li> </ul> <p>For details, see the <code>bayesopt</code> <code>AcquisitionFunctionName</code> name-value pair, or “Acquisition Function Types” on page 10-4.</p>	'expected-improvement-per-second-plus'

Field Name	Values	Default
MaxObjectiveEvaluations	Maximum number of objective function evaluations.	30 for 'bayesopt' or 'randomsearch', and the entire grid for 'gridsearch'
MaxTime	Time limit, specified as a positive real. The time limit is in seconds, as measured by <code>tic</code> and <code>toc</code> . Run time can exceed <code>MaxTime</code> because <code>MaxTime</code> does not interrupt function evaluations.	Inf
NumGridDivisions	For 'gridsearch', the number of values in each dimension. Can be a vector of positive integers giving the number of values for each dimension, or a scalar that applies to all dimensions. Ignored for categorical variables.	10
ShowPlots	Logical value indicating whether to show plots. If <code>true</code> , plots the best objective function value against iteration number. If there are one or two optimization parameters, and if <code>Optimizer</code> is 'bayesopt', then <code>ShowPlots</code> also plots a model of the objective function against the parameters.	<code>true</code>
SaveIntermediateResults	Logical value indicating whether to save results when <code>Optimizer</code> is 'bayesopt'. If <code>true</code> , overwrites a workspace variable named 'BayesoptResults' at each iteration. The variable is a <code>BayesianOptimization</code> object.	<code>false</code>

Field Name	Values	Default
Verbose	Display to the command line. <ul style="list-style-type: none"> <li>• 0 — No iterative display</li> <li>• 1 — Iterative display</li> <li>• 2 — Iterative display with extra information</li> </ul> For details, see the <code>bayesopt</code> Verbose name-value pair.	1
UseParallel	Logical value indicating whether to run Bayesian optimization in parallel, which requires Parallel Computing Toolbox. For details, see “Parallel Bayesian Optimization” on page 10-9.	false
Repartition	Logical value indicating whether to repartition the cross-validation at every iteration. If <code>false</code> , the optimizer uses a single partition for the optimization.  <code>true</code> usually gives the most robust results because this setting takes partitioning noise into account. However, for good results, <code>true</code> requires at least twice as many function evaluations.	false
Use no more than one of the following three field names.		
CVPartition	A <code>cvpartition</code> object, as created by <code>cvpartition</code>	Kfold = 5
Holdout	A scalar in the range (0,1) representing the holdout fraction.	
Kfold	An integer greater than 1.	

Example: `struct('MaxObjectiveEvaluations',60)`

Data Types: `struct`



## Output Arguments

### **mdl** — Trained classification ensemble model

ClassificationBaggedEnsemble model object | ClassificationEnsemble model object | ClassificationPartitionedEnsemble cross-validated model object

Trained ensemble model, returned as one of the model objects in this table.

Model Object	Specify Any Cross-Validation Options?	Method Setting	Resample Setting
ClassificationBaggedEnsemble	No	'Bag'	'on'
ClassificationEnsemble	No	Any ensemble-aggregation method for classification	'off'
ClassificationPartitionedEnsemble	Yes	Any classification ensemble-aggregation method	'off' or 'on'

The name-value pair arguments that control cross-validation are `CrossVal`, `Holdout`, `KFold`, `Leaveout`, and `CVPartition`.

To reference properties of `Mdl`, use dot notation. For example, to access or display the cell vector of weak learner model objects for an ensemble that has not been cross-validated, enter `Mdl.Trained` at the command line.

## Tips

- `NumLearningCycles` can vary from a few dozen to a few thousand. Usually, an ensemble with good predictive power requires from a few hundred to a few thousand weak learners. However, you do not have to train an ensemble for that many cycles at once. You can start by growing a few dozen learners, inspect the ensemble performance and then, if necessary, train more weak learners using `resume` for classification problems.
- Ensemble performance depends on the ensemble setting and the setting of the weak learners. That is, if you specify weak learners with default parameters, then the ensemble can perform poorly. Therefore, like ensemble settings, it is good practice to adjust the parameters of the weak learners using templates, and to choose values that minimize generalization error.

- If you specify to resample using `Resample`, then it is good practice to resample to entire data set. That is, use the default setting of 1 for `FResample`.
- If the ensemble-aggregation method (`Method`) is 'bag' and:
  - The misclassification cost (`Cost`) is highly imbalanced, then, for in-bag samples, the software oversamples unique observations from the class that has a large penalty.
  - The class prior probabilities (`Prior`) are highly skewed, the software oversamples unique observations from the class that has a large prior probability.

For smaller sample sizes, these combinations can result in a low relative frequency of out-of-bag observations from the class that has a large penalty or prior probability. Consequently, the estimated out-of-bag error is highly variable and it can be difficult to interpret. To avoid large estimated out-of-bag error variances, particularly for small sample sizes, set a more balanced misclassification cost matrix using `Cost` or a less skewed prior probability vector using `Prior`.

- Because the order of some input and output arguments correspond to the distinct classes in the training data, it is good practice to specify the class order using the `ClassNames` name-value pair argument.
- To determine the class order quickly, remove all observations from the training data that are unclassified (that is, have a missing label), obtain and display an array of all the distinct classes, and then specify the array for `ClassNames`. For example, suppose the response variable (`Y`) is a cell array of labels. This code specifies the class order in the variable `classNames`.

```
Ycat = categorical(Y);
classNames = categories(Ycat)
```

`categorical` assigns <undefined> to unclassified observations and `categories` excludes <undefined> from its output. Therefore, if you use this code for cell arrays of labels or similar code for categorical arrays, then you do not have to remove observations with missing labels to obtain a list of the distinct classes.
- To specify that the class order from lowest-represented label to most-represented, then quickly determine the class order (as in the previous bullet), but arrange the classes in the list by frequency before passing the list to `ClassNames`. Following from the previous example, this code specifies the class order from lowest- to most-represented in `classNamesLH`.

```

Ycat = categorical(Y);
classNames = categories(Ycat);
freq = countcats(Ycat);
[~,idx] = sort(freq);
classNamesLH = classNames(idx);

```

- After training a `ClassificationEnsemble` or `ClassificationBaggedEnsemble` model object by using `fitcensemble`, use the function `predict` and the trained model object to generate C code that predicts labels for new data. For details, see “Code Generation” on page 32-0 .

## Algorithms

- For details of ensemble-aggregation algorithms, see “Ensemble Algorithms” on page 18-51.
- If you specify `Method` to be a boosting algorithm and `Learners` to be decision trees, then the software grows shallow decision trees by default. You can adjust tree depth by specifying the `MaxNumSplits`, `MinLeafSize`, and `MinParentSize` name-value pair arguments using `templateTree`.
- `fitcensemble` generates in-bag samples by oversampling classes with large misclassification costs and undersampling classes with small misclassification costs. Consequently, out-of-bag samples have fewer observations from classes with large misclassification costs and more observations from classes with small misclassification costs. If you train a classification ensemble using a small data set and a highly skewed cost matrix, then the number of out-of-bag observations per class can be low. Therefore, the estimated out-of-bag error can have a large variance and can be difficult to interpret. The same phenomenon can occur for classes with large prior probabilities.
- For the `RUSBoost` ensemble-aggregation method (`Method`), the name-value pair argument `RatioToSmallest` specifies the sampling proportion for each class with respect to the lowest-represented class. For example, suppose that there are two classes in the training data: *A* and *B*. *A* have 100 observations and *B* have 10 observations. and that the lowest-represented class has *m* observations in the training data.
  - If you set `'RatioToSmallest', 2`, then  $s*m = 2*10 = 20$ . Consequently, `fitcensemble` trains every learner using 20 observations from class *A* and 20 observations from class *B*. If you set `'RatioToSmallest', [2 2]`, then you obtain the same result.

- If you set 'RatioToSmallest', [2, 1], then  $s1*m = 2*10 = 20$  and  $s2*m = 1*10 = 10$ . Consequently, `fitcensemble` trains every learner using 20 observations from class *A* and 10 observations from class *B*.
- For dual-core systems and above, `fitcensemble` parallelizes training using Intel Threading Building Blocks (TBB). For details on Intel TBB, see <https://software.intel.com/en-us/intel-tbb>.

## References

- [1] Breiman, L. “Bagging Predictors.” *Machine Learning*. Vol. 26, pp. 123–140, 1996.
- [2] Breiman, L. “Random Forests.” *Machine Learning*. Vol. 45, pp. 5–32, 2001.
- [3] Freund, Y. “A more robust boosting algorithm.” *arXiv:0905.2138v1*, 2009.
- [4] Freund, Y. and R. E. Schapire. “A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting.” *J. of Computer and System Sciences*, Vol. 55, pp. 119–139, 1997.
- [5] Friedman, J. “Greedy function approximation: A gradient boosting machine.” *Annals of Statistics*, Vol. 29, No. 5, pp. 1189–1232, 2001.
- [6] Friedman, J., T. Hastie, and R. Tibshirani. “Additive logistic regression: A statistical view of boosting.” *Annals of Statistics*, Vol. 28, No. 2, pp. 337–407, 2000.
- [7] Hastie, T., R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning* section edition, Springer, New York, 2008.
- [8] Ho, T. K. “The random subspace method for constructing decision forests.” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 20, No. 8, pp. 832–844, 1998.
- [9] Schapire, R. E., Y. Freund, P. Bartlett, and W.S. Lee. “Boosting the margin: A new explanation for the effectiveness of voting methods.” *Annals of Statistics*, Vol. 26, No. 5, pp. 1651–1686, 1998.
- [10] Seiffert, C., T. Khoshgoftaar, J. Hulse, and A. Napolitano. “RUSBoost: Improving classification performance when training data is skewed.” *19th International Conference on Pattern Recognition*, pp. 1–4, 2008.

- [11] Warmuth, M., J. Liao, and G. Ratsch. “Totally corrective boosting algorithms that maximize the margin.” *Proc. 23rd Int’l. Conf. on Machine Learning, ACM*, New York, pp. 1001–1008, 2006.

## See Also

`ClassificationBaggedEnsemble` | `ClassificationEnsemble` |  
`ClassificationPartitionedEnsemble` | `predict` | `templateDiscriminant` |  
`templateKNN` | `templateTree`

## Topics

“Supervised Learning Workflow and Algorithms” on page 18-2  
“Framework for Ensemble Learning” on page 18-41

**Introduced in R2016b**

## fitcknn

Fit  $k$ -nearest neighbor classifier

### Syntax

```
Mdl = fitcknn(Tbl, ResponseVarName)
```

```
Mdl = fitcknn(Tbl, formula)
```

```
Mdl = fitcknn(Tbl, Y)
```

```
Mdl = fitcknn(X, Y)
```

```
Mdl = fitcknn( ____, Name, Value)
```

### Description

`Mdl = fitcknn(Tbl, ResponseVarName)` returns a  $k$ -nearest neighbor classification model based on the input variables (also known as predictors, features, or attributes) in the table `Tbl` and output (response) `Tbl.ResponseVarName`.

`Mdl = fitcknn(Tbl, formula)` returns a  $k$ -nearest neighbor classification model based on the input variables in the table `Tbl`. `formula` is an explanatory model of the response and a subset of predictor variables in `Tbl`.

`Mdl = fitcknn(Tbl, Y)` returns a  $k$ -nearest neighbor classification model based on the predictor variables in the table `Tbl` and response array `Y`.

`Mdl = fitcknn(X, Y)` returns a  $k$ -nearest neighbor classification model based on the predictor data `X` and response `Y`.

`Mdl = fitcknn( ____, Name, Value)` fits a model with additional options specified by one or more name-value pair arguments, using any of the previous syntaxes. For example, you can specify the tie-breaking algorithm, distance metric, or observation weights.

## Examples

### Train a $k$ -Nearest Neighbor Classifier

Construct a  $k$ -nearest neighbor classifier for Fisher's iris data, where  $k$ , the number of nearest neighbors in the predictors, is 5.

Load Fisher's iris data.

```
load fisheriris
X = meas;
Y = species;
```

$X$  is a numeric matrix that contains four petal measurements for 150 irises.  $Y$  is a cell array of character vectors that contains the corresponding iris species.

Train a 5-nearest neighbors classifier. It is good practice to standardize noncategorical predictor data.

```
Mdl = fitcknn(X,Y,'NumNeighbors',5,'Standardize',1)
```

```
Mdl =
```

```
ClassificationKNN
    ResponseName: 'Y'
    CategoricalPredictors: []
           ClassNames: {'setosa' 'versicolor' 'virginica'}
    ScoreTransform: 'none'
    NumObservations: 150
           Distance: 'euclidean'
    NumNeighbors: 5
```

$Mdl$  is a trained `ClassificationKNN` classifier, and some of its properties display in the Command Window.

To access the properties of  $Mdl$ , use dot notation.

```
Mdl.ClassNames
Mdl.Prior
```

```
ans =  
  
    3x1 cell array  
  
    {'setosa'      }  
    {'versicolor'}  
    {'virginica'  }  
  
ans =  
  
    0.3333    0.3333    0.3333
```

`Mdl.Prior` contains the class prior probabilities, which are settable using the name-value pair argument `'Prior'` in `fitcknn`. The order of the class prior probabilities corresponds to the order of the classes in `Mdl.ClassNames`. By default, the prior probabilities are the respective relative frequencies of the classes in the data.

You can also reset the prior probabilities after training. For example, set the prior probabilities to 0.5, 0.2, and 0.3 respectively.

```
Mdl.Prior = [0.5 0.2 0.3];
```

You can pass `Mdl` to, for example, `ClassificationKNN.predict` to label new measurements, or `ClassificationKNN.crossval` to cross validate the classifier.

### Train a *k*-Nearest Neighbor Classifier Using the Minkowski Metric

Load Fisher's iris data set.

```
load fisheriris  
X = meas;  
Y = species;
```

`X` is a numeric matrix that contains four petal measurements for 150 irises. `Y` is a cell array of character vectors that contains the corresponding iris species.

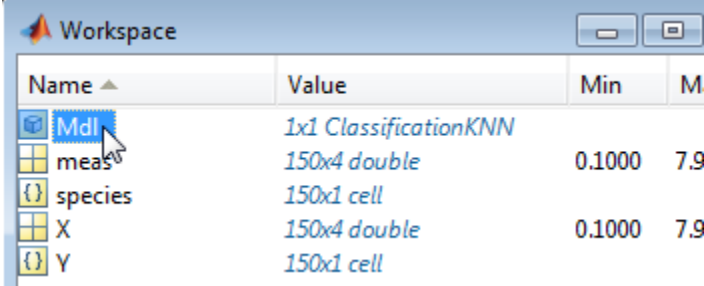
Train a 3-nearest neighbors classifier using the Minkowski metric. To use the Minkowski metric, you must use an exhaustive searcher. It is good practice to standardize noncategorical predictor data.



```
Mdl = fitcknn(X,Y,'NumNeighbors',3,...  
             'NSMethod','exhaustive','Distance','minkowski',...  
             'Standardize',1);
```

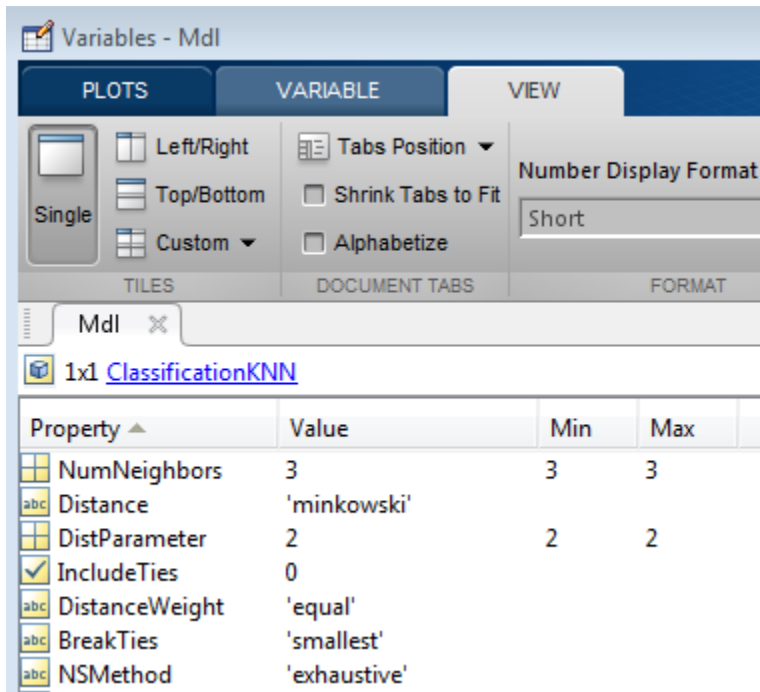
Mdl is a ClassificationKNN classifier.

You can examine the properties of Mdl by double-clicking Mdl in the Workspace window. This opens the Variable Editor.



The screenshot shows the MATLAB Workspace window with a table of variables. The table has columns for Name, Value, Min, and M. The variable Mdl is highlighted with a mouse cursor.

Name	Value	Min	M
Mdl	1x1 ClassificationKNN		
meas	150x4 double	0.1000	7.9
species	150x1 cell		
X	150x4 double	0.1000	7.9
Y	150x1 cell		



### Train a $k$ -Nearest Neighbor Classifier Using a Custom Distance Metric

Train a  $k$ -nearest neighbor classifier using the chi-square distance.

Load Fisher's iris data set.

```
load fisheriris
X = meas; % Predictors
Y = species; % Response
```

The chi-square distance between  $j$ -dimensional points  $x$  and  $z$  is

$$\chi(x, z) = \sqrt{\sum_{j=1}^J w_j (x_j - z_j)^2},$$

where  $w_j$  is a weight associated with dimension  $j$ .

Specify the chi-square distance function. The distance function must:

- Take one row of  $X$ , e.g.,  $x$ , and the matrix  $Z$ .
- Compare  $x$  to each row of  $Z$ .
- Return a vector  $D$  of length  $n_z$ , where  $n_z$  is the number of rows of  $Z$ . Each element of  $D$  is the distance between the observation corresponding to  $x$  and the observations corresponding to each row of  $Z$ .

```
chiSqrDist = @(x,Z,wt) sqrt((bsxfun(@minus,x,Z).^2)*wt);
```

This example uses arbitrary weights for illustration.

Train a 3-nearest neighbor classifier. It is good practice to standardize noncategorical predictor data.

```
k = 3;
w = [0.3; 0.3; 0.2; 0.2];
KNNMdl = fitcknn(X,Y,'Distance',@(x,Z)chiSqrDist(x,Z,w),...
    'NumNeighbors',k,'Standardize',1);
```

KNNMdl is a ClassificationKNN classifier.

Cross validate the KNN classifier using the default 10-fold cross validation. Examine the classification error.

```
rng(1); % For reproducibility
CVKNNMdl = crossval(KNNMdl);
classError = kfoldLoss(CVKNNMdl)
```

```
classError =
```

```
0.0600
```

CVKNNMdl is a ClassificationPartitionedModel classifier. The 10-fold classification error is 4%.

Compare the classifier with one that uses a different weighting scheme.

```
w2 = [0.2; 0.2; 0.3; 0.3];
CVKNNMdl2 = fitcknn(X,Y,'Distance',@(x,Z)chiSqrDist(x,Z,w2),...
    'NumNeighbors',k,'KFold',10,'Standardize',1);
classError2 = kfoldLoss(CVKNNMdl2)
```

```
classError2 =
    0.0400
```

The second weighting scheme yields a classifier that has better out-of-sample performance.

### Optimize Fitted KNN Classifier

This example shows how to optimize hyperparameters automatically using `fitcknn`. The example uses the Fisher iris data.

Load the data.

```
load fisheriris
X = meas;
Y = species;
```

Find hyperparameters that minimize five-fold cross-validation loss by using automatic hyperparameter optimization.

For reproducibility, set the random seed and use the 'expected-improvement-plus' acquisition function.

```
rng(1)
Mdl = fitcknn(X,Y,'OptimizeHyperparameters','auto',...
    'HyperparameterOptimizationOptions',...
    struct('AcquisitionFunctionName','expected-improvement-plus'))
```

```
=====
| Iter | Eval  | Objective | Objective | BestSoFar | BestSoFar | NumNeighbors
|      | result |           | runtime   | (observed) | (estim.)  |
|=====|=====|=====|=====|=====|=====|=====
|  1  | Best  | 0.026667 | 0.30693  | 0.026667 | 0.026667 | 30
|  2  | Accept | 0.04    | 0.28519  | 0.026667 | 0.027197 | 2
|  3  | Accept | 0.19333 | 0.13717  | 0.026667 | 0.030324 | 1
|  4  | Accept | 0.33333 | 0.20389  | 0.026667 | 0.033313 | 31
|  5  | Best  | 0.02    | 0.16076  | 0.02    | 0.020648 | 6
|  6  | Accept | 0.07333 | 0.15733  | 0.02    | 0.023082 | 1
|  7  | Accept | 0.06    | 0.12982  | 0.02    | 0.020875 | 2
```

	8	Accept		0.04		0.13275		0.02		0.020622		1
	9	Accept		0.24		0.18618		0.02		0.020562		74
	10	Accept		0.04		0.14594		0.02		0.020649		1
	11	Accept		0.053333		0.20403		0.02		0.020722		1
	12	Accept		0.19333		0.20054		0.02		0.020701		1
	13	Accept		0.04		0.18045		0.02		0.029203		1
	14	Accept		0.04		0.17467		0.02		0.031888		75
	15	Accept		0.04		0.17522		0.02		0.020076		1
	16	Accept		0.093333		0.18827		0.02		0.020073		75
	17	Accept		0.093333		0.13931		0.02		0.02007		75
	18	Accept		0.1		0.165		0.02		0.020061		75
	19	Accept		0.15333		0.14444		0.02		0.020044		75
	20	Accept		0.1		0.16392		0.02		0.020044		75
=====												
	Iter	Eval		Objective		Objective		BestSoFar		BestSoFar		NumNeighbors
		result				runtime		(observed)		(estim.)		
=====												
	21	Accept		0.033333		0.13962		0.02		0.020046		75
	22	Accept		0.033333		0.15279		0.02		0.02656		9
	23	Accept		0.033333		0.15605		0.02		0.02854		9
	24	Accept		0.02		0.15333		0.02		0.028607		1
	25	Accept		0.02		0.14989		0.02		0.022264		1
	26	Accept		0.02		0.18665		0.02		0.021439		1
	27	Accept		0.02		0.16238		0.02		0.020999		1
	28	Accept		0.66667		0.19424		0.02		0.020008		75
	29	Accept		0.04		0.18571		0.02		0.020008		12
	30	Best		0.013333		0.1594		0.013333		0.013351		6

---

Optimization completed.

MaxObjectiveEvaluations of 30 reached.

Total function evaluations: 30

Total elapsed time: 59.9994 seconds.

Total objective function evaluation time: 5.2219

Best observed feasible point:

NumNeighbors	Distance
-----	-----
6	euclidean

Observed objective function value = 0.013333

Estimated objective function value = 0.013351

Function evaluation time = 0.1594

Best estimated feasible point (according to models):

NumNeighbors	Distance
6	euclidean

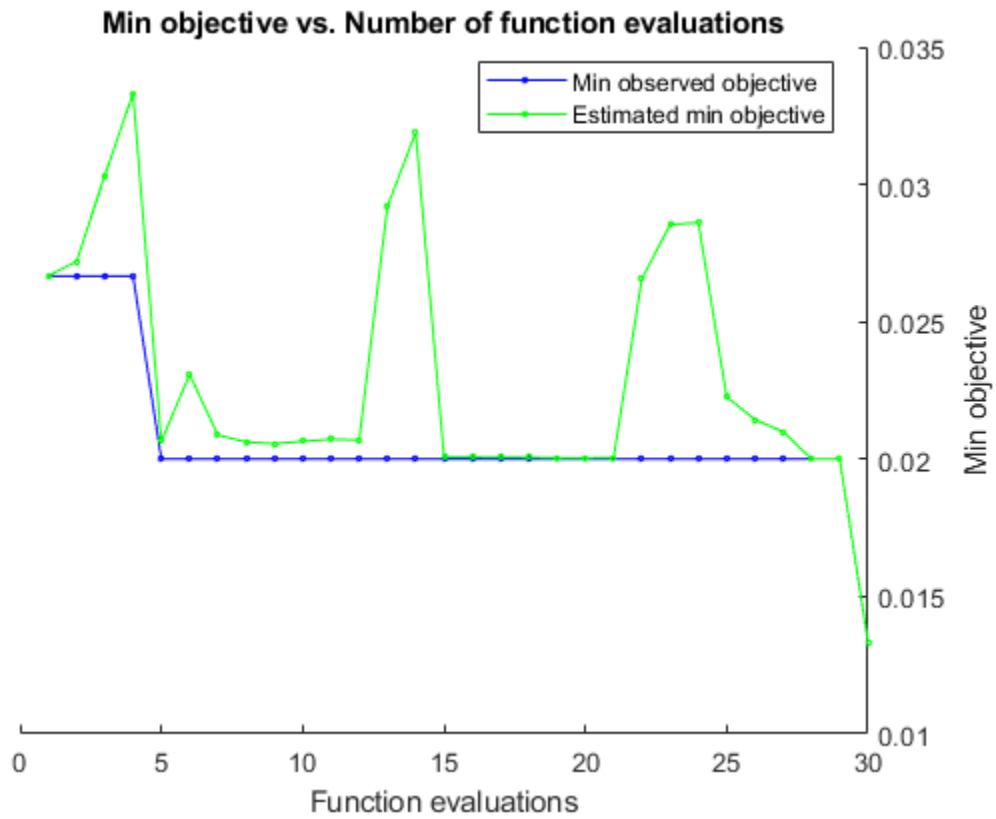
Estimated objective function value = 0.013351

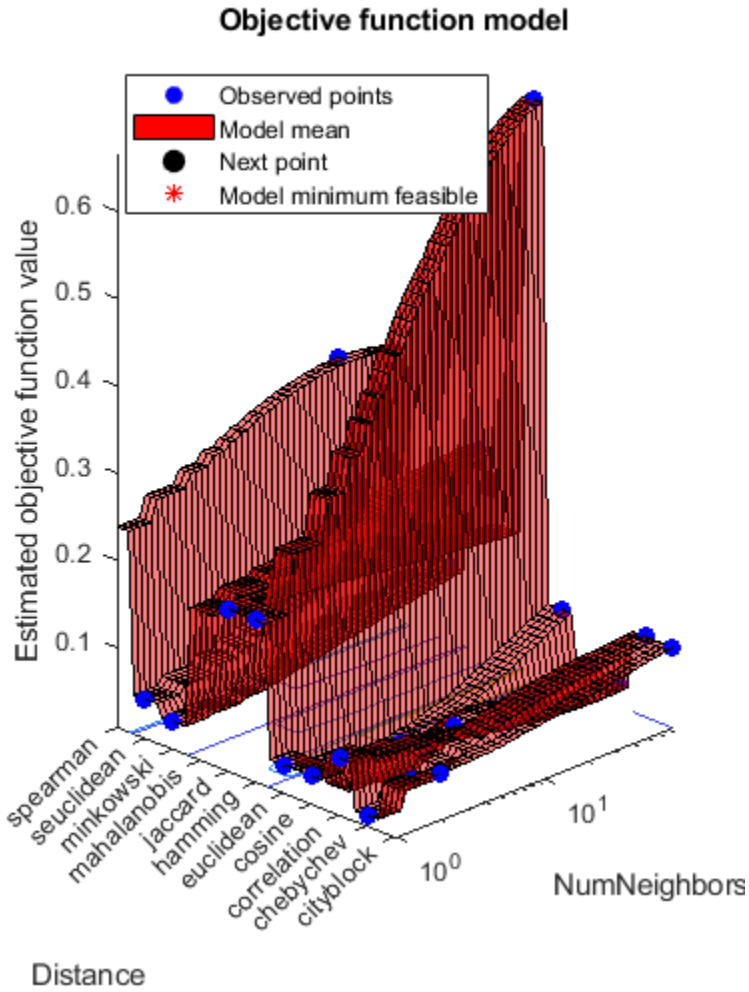
Estimated function evaluation time = 0.1705

Mdl =

ClassificationKNN

```
      ResponseName: 'Y'  
      CategoricalPredictors: []  
          ClassNames: {'setosa' 'versicolor' 'virginica'}  
      ScoreTransform: 'none'  
      NumObservations: 150  
HyperparameterOptimizationResults: [1x1 BayesianOptimization]  
          Distance: 'euclidean'  
          NumNeighbors: 6
```





- “Construct KNN Classifier” on page 18-36
- “Modify KNN Classifier” on page 18-39



## Input Arguments

### **Tbl** — Sample data

table

Sample data used to train the model, specified as a table. Each row of `Tbl` corresponds to one observation, and each column corresponds to one predictor variable. Optionally, `Tbl` can contain one additional column for the response variable. Multi-column variables and cell arrays other than cell arrays of character vectors are not allowed.

If `Tbl` contains the response variable, and you want to use all remaining variables in `Tbl` as predictors, then specify the response variable using `ResponseVarName`.

If `Tbl` contains the response variable, and you want to use only a subset of the remaining variables in `Tbl` as predictors, then specify a formula using `formula`.

If `Tbl` does not contain the response variable, then specify a response variable using `Y`. The length of response variable and the number of rows of `Tbl` must be equal.

Data Types: `table`

### **ResponseVarName** — Response variable name

name of variable in `Tbl`

Response variable name, specified as the name of a variable in `Tbl`.

You must specify `ResponseVarName` as a character vector. For example, if the response variable `Y` is stored as `Tbl.Y`, then specify it as `'Y'`. Otherwise, the software treats all columns of `Tbl`, including `Y`, as predictors when training the model.

The response variable must be a categorical or character array, logical or numeric vector, or cell array of character vectors. If `Y` is a character array, then each element must correspond to one row of the array.

It is good practice to specify the order of the classes using the `ClassNames` name-value pair argument.

Data Types: `char`

### **formula** — Explanatory model of response and subset of predictor variables

character vector

Explanatory model of the response and a subset of the predictor variables, specified as a character vector in the form of 'Y~X1+X2+X3'. In this form, Y represents the response variable, and X1, X2, and X3 represent the predictor variables. The variables must be variable names in Tbl (Tbl.Properties.VariableNames).

To specify a subset of variables in Tbl as predictors for training the model, use a formula. If you specify a formula, then the software does not use any variables in Tbl that do not appear in formula.

Data Types: char

### **Y — Class labels**

numeric vector | categorical vector | logical vector | character array | cell array of character vectors

Class labels, specified as a categorical or character array, logical or numeric vector, or cell array of character vectors. Each row of Y represents the classification of the corresponding row of X.

The software considers NaN, '' (empty character vector), and <undefined> values in Y to be missing values. Consequently, the software does not train using observations with a missing response.

Data Types: single | double | logical | char | cell

### **X — Predictor data**

numeric matrix

Predictor data, specified as numeric matrix.

Each row corresponds to one observation (also known as an instance or example), and each column corresponds to one predictor variable (also known as a feature).

The length of Y and the number of rows of X must be equal.

To specify the names of the predictors in the order of their appearance in X, use the PredictorNames name-value pair argument.

Data Types: double | single

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

---

**Note** You cannot use any cross-validation name-value pair along with `OptimizeHyperparameters`. You can modify the cross-validation for `OptimizeHyperparameters` only by using the `HyperparameterOptimizationOptions` name-value pair.

---

Example: `'NumNeighbors', 3, 'NSMethod', 'exhaustive', 'Distance', 'minkowski'` specifies a classifier for three-nearest neighbors using the nearest neighbor search method and the Minkowski metric.

### Model Parameters

#### **BreakTies** — Tie-breaking algorithm

`'smallest'` (default) | `'nearest'` | `'random'`

Tie-breaking algorithm used by the `predict` method if multiple classes have the same smallest cost, specified as the comma-separated pair consisting of `'BreakTies'` and one of the following:

- `'smallest'` — Use the smallest index among tied groups.
- `'nearest'` — Use the class with the nearest neighbor among tied groups.
- `'random'` — Use a random tiebreaker among tied groups.

By default, ties occur when multiple classes have the same number of nearest points among the `K` nearest neighbors.

Example: `'BreakTies', 'nearest'`

#### **BucketSize** — Maximum data points in node

50 (default) | positive integer value

Maximum number of data points in the leaf node of the *kd*-tree, specified as the comma-separated pair consisting of 'BucketSize' and a positive integer value. This argument is meaningful only when NSMethod is 'kdtree'.

Example: 'BucketSize',40

Data Types: single | double

### **CategoricalPredictors** — Categorical predictor flag

[] (default) | 'all'

Categorical predictor flag, specified as the comma-separated pair consisting of 'CategoricalPredictors' and one of the following:

- 'all' — All predictors are categorical.
- [] — No predictors are categorical.

When you set CategoricalPredictors to 'all', the default Distance is 'hamming'.

Example: 'CategoricalPredictors','all'

Data Types: char

### **ClassNames** — Names of classes to use for training

categorical array | cell array of character vectors | character array | logical vector | numeric vector

Names of classes to use for training, specified as the comma-separated pair consisting of 'ClassNames' and a categorical or character array, logical or numeric vector, or cell array of character vectors. ClassNames must be the same data type as Y.

If ClassNames is a character array, then each element must correspond to one *row* of the array.

Use ClassNames to:

- Order the classes during training.
- Specify the order of any input or output argument dimension that corresponds to the class order. For example, use ClassNames to specify the order of the dimensions of Cost or the column order of classification scores returned by predict.

- Select a subset of classes for training. For example, suppose that the set of all distinct class names in  $Y$  is  $\{'a', 'b', 'c'\}$ . To train the model using observations from classes 'a' and 'c' only, specify 'ClassNames', {'a', 'c'}.

The default is the set of all distinct class names in  $Y$ .

Example: 'ClassNames', {'b', 'g'}

Data Types: categorical | char | logical | single | double | cell

### **Cost — Cost of misclassification**

square matrix | structure

Cost of misclassification of a point, specified as the comma-separated pair consisting of 'Cost' and one of the following:

- Square matrix, where  $\text{Cost}(i, j)$  is the cost of classifying a point into class  $j$  if its true class is  $i$  (i.e., the rows correspond to the true class and the columns correspond to the predicted class). To specify the class order for the corresponding rows and columns of  $\text{Cost}$ , additionally specify the `ClassNames` name-value pair argument.
- Structure  $S$  having two fields:  $S.\text{ClassNames}$  containing the group names as a variable of the same type as  $Y$ , and  $S.\text{ClassificationCosts}$  containing the cost matrix.

The default is  $\text{Cost}(i, j) = 1$  if  $i \neq j$ , and  $\text{Cost}(i, j) = 0$  if  $i = j$ .

Data Types: single | double | struct

### **Cov — Covariance matrix**

nancov(X) (default) | positive definite matrix of scalar values

Covariance matrix, specified as the comma-separated pair consisting of 'Cov' and a positive definite matrix of scalar values representing the covariance matrix when computing the Mahalanobis distance. This argument is only valid when 'Distance' is 'mahalanobis'.

You cannot simultaneously specify 'Standardize' and either of 'Scale' or 'Cov'.

Data Types: single | double

### **IncludeTies — Tie inclusion flag**

false (default) | true

Tie inclusion flag, specified as the comma-separated pair consisting of `'IncludeTies'` and a logical value indicating whether `predict` includes all the neighbors whose distance values are equal to the *K*th smallest distance. If `IncludeTies` is `true`, `predict` includes all these neighbors. Otherwise, `predict` uses exactly *K* neighbors.

Example: `'IncludeTies', true`

Data Types: logical

### **NSMethod — Nearest neighbor search method**

`'kdtree' | 'exhaustive'`

Nearest neighbor search method, specified as the comma-separated pair consisting of `'NSMethod'` and `'kdtree'` or `'exhaustive'`.

- `'kdtree'` — Create and use a *kd*-tree to find nearest neighbors. `'kdtree'` is valid when the distance metric is one of the following:
  - `'euclidean'`
  - `'cityblock'`
  - `'minkowski'`
  - `'chebychev'`
- `'exhaustive'` — Use the exhaustive search algorithm. The distance values from all points in *X* to each point in *Y* are computed to find nearest neighbors.

The default is `'kdtree'` when *X* has 10 or fewer columns, *X* is not sparse, and the distance metric is a `'kdtree'` type; otherwise, `'exhaustive'`.

Example: `'NSMethod', 'exhaustive'`

### **PredictorNames — Predictor variable names**

cell array of unique character vectors

Predictor variable names, specified as the comma-separated pair consisting of `'PredictorNames'` and a cell array of unique character vectors. The functionality of `'PredictorNames'` depends on the way you supply the training data.

- If you supply *X* and *Y*, then you can use `'PredictorNames'` to give the predictor variables in *X* names.
  - The order of the names in `PredictorNames` must correspond to the column order of *X*. That is, `PredictorNames{1}` is the name of `X(:, 1)`, `PredictorNames{2}`

is the name of  $X(:, 2)$ , and so on. Also, `size(X, 2)` and `numel(PredictorNames)` must be equal.

- By default, `PredictorNames` is `{'x1', 'x2', ...}`.
- If you supply `Tbl`, then you can use `'PredictorNames'` to choose which predictor variables to use in training. That is, `fitcknn` uses the predictor variables in `PredictorNames` and the response only in training.
  - `PredictorNames` must be a subset of `Tbl.Properties.VariableNames` and cannot include the name of the response variable.
  - By default, `PredictorNames` contains the names of all predictor variables.
  - It good practice to specify the predictors for training using one of `'PredictorNames'` or formula only.

Example: `'PredictorNames',`  
`{'SepalLength', 'SepalWidth', 'PedalLength', 'PedalWidth'}`

Data Types: `cell`

### **Prior — Prior probabilities**

`'empirical'` (default) | `'uniform'` | vector of scalar values | structure

Prior probabilities for each class, specified as the comma-separated pair consisting of `'Prior'` and a value in this table.

Value	Description
<code>'empirical'</code>	The class prior probabilities are the class relative frequencies in $Y$ .
<code>'uniform'</code>	All class prior probabilities are equal to $1/K$ , where $K$ is the number of classes.
numeric vector	Each element is a class prior probability. Order the elements according to <code>Mdl.ClassNames</code> or specify the order using the <code>ClassNames</code> name-value pair argument. The software normalizes the elements such that they sum to 1.

Value	Description
structure	<p>A structure <i>S</i> with two fields:</p> <ul style="list-style-type: none"> <li>• <i>S.ClassNames</i> contains the class names as a variable of the same type as <i>Y</i>.</li> <li>• <i>S.ClassProbs</i> contains a vector of corresponding prior probabilities. The software normalizes the elements such that they sum to 1.</li> </ul>

If you set values for both `Weights` and `Prior`, the weights are renormalized to add up to the value of the prior probability in the respective class.

Example: `'Prior', 'uniform'`

Data Types: `single` | `double` | `struct`

#### **ResponseName** — Response variable name

`'Y'` (default) | character vector

Response variable name, specified as the comma-separated pair consisting of `'ResponseName'` and a character vector.

- If you supply *Y*, then you can use `'ResponseName'` to specify a name for the response variable.
- If you supply `ResponseVarName` or `formula`, then you cannot use `'ResponseName'`.

Example: `'ResponseName', 'response'`

Data Types: `char`

#### **Scale** — Distance scale

`nanstd(X)` (default) | vector of nonnegative scalar values

Distance scale, specified as the comma-separated pair consisting of `'Scale'` and a vector containing nonnegative scalar values with length equal to the number of columns in *X*. Each coordinate difference between *X* and a query point is scaled by the corresponding element of `Scale`. This argument is only valid when `'Distance'` is `'seuclidean'`.

You cannot simultaneously specify `'Standardize'` and either of `'Scale'` or `'Cov'`.



Data Types: `single` | `double`

### ScoreTransform — Score transformation

'none' (default) | character vector | function handle

Score transformation, specified as the comma-separated pair consisting of 'ScoreTransform' and either a character vector or a function handle.

This table summarizes the available character vectors.

Value	Description
'doublelogit'	$1/(1 + e^{-2x})$
'invlogit'	$\log(x / (1-x))$
'ismax'	Set the score for the class with the largest score to 1, and set the scores for all other classes to 0.
'logit'	$1/(1 + e^{-x})$
'none' or 'identity'	$x$ (no transformation)
'sign'	-1 for $x < 0$ 0 for $x = 0$ 1 for $x > 0$
'symmetric'	$2x - 1$
'symmetricismax'	Set the score for the class with the largest score to 1, and set the scores for all other classes to -1.
'symmetriclogit'	$2/(1 + e^{-x}) - 1$

For a MATLAB function or a function you define, use its function handle. The function handle must accept a matrix (the original scores) and return a matrix of the same size (the transformed scores).

Example: 'ScoreTransform', 'logit'

Data Types: `char` | `function_handle`

### Weights — Observation weights

numeric vector of positive values | character vector

Observation weights, specified as the comma-separated pair consisting of 'Weights' and a numeric vector of positive values or name of a variable in `Tbl`. The software weighs

the observations in each row of `X` or `Tbl` with the corresponding value in `Weights`. The size of `Weights` must equal the number of rows of `X` or `Tbl`.

If you specify the input data as a table `Tbl`, then `Weights` can be the name of a variable in `Tbl` that contains a numeric vector. In this case, you must specify `Weights` as a character vector. For example, if the weights vector `W` is stored as `Tbl.W`, then specify it as `'W'`. Otherwise, the software treats all columns of `Tbl`, including `W`, as predictors or the response when training the model.

The software normalizes `Weights` to sum up to the value of the prior probability in the respective class.

By default, `Weights` is `ones(n, 1)`, where  $n$  is the number of observations in `X` or `Tbl`.

Data Types: `double` | `single` | `char`

### Cross Validation

#### **CrossVal** — Cross-validation flag

`'off'` (default) | `'on'`

Cross-validation flag, specified as the comma-separated pair consisting of `'Crossval'` and `'on'` or `'off'`.

If you specify `'on'`, then the software implements 10-fold cross-validation.

To override this cross-validation setting, use one of these name-value pair arguments: `CVPartition`, `Holdout`, `KFold`, or `Leaveout`. To create a cross-validated model, you can use one cross-validation name-value pair argument at a time only.

Alternatively, cross validate later by passing `Mdl` to `crossval`.

Example: `'CrossVal', 'on'`

#### **CVPartition** — Cross-validation partition

`[]` (default) | `cvpartition` partition object

Cross-validation partition, specified as the comma-separated pair consisting of `'CVPartition'` and a `cvpartition` partition object as created by `cvpartition`. The partition object specifies the type of cross-validation, and also the indexing for training and validation sets.

To create a cross-validated model, you can use one of these four name-value pair arguments only: `CVPartition`, `Holdout`, `KFold`, or `Leaveout`.

Example: Suppose you create a random partition for 5-fold cross-validation on 500 observations by using `cvp = cvpartition(500, 'KFold', 5)`. Then, you can specify the cross-validation partition for `fitcecoc` as `'CVPartition', cvp`.

### **Holdout** — Fraction of data for holdout validation

scalar value in the range (0,1)

Fraction of data used for holdout validation, specified as the comma-separated pair consisting of `'Holdout'` and a scalar value in the range (0,1). If you specify `'Holdout', p`, then the software:

- 1 Randomly reserves  $p \times 100\%$  of the data as validation data, and trains the model using the rest of the data
- 2 Stores the compact, trained model in the `Trained` property of the cross-validated model.

To create a cross-validated model, you can use one of these four name-value pair arguments only: `CVPartition`, `Holdout`, `KFold`, or `Leaveout`.

Example: `'Holdout', 0.1`

Data Types: `double` | `single`

### **KFold** — Number of folds

10 (default) | positive integer value greater than 1

Number of folds to use in a cross-validated model, specified as the comma-separated pair consisting of `'KFold'` and a positive integer value greater than 1. If you specify, e.g., `'KFold', k`, then the software:

- 1 Randomly partitions the data into  $k$  sets
- 2 For each set, reserves the set as validation data, and trains the model using the other  $k - 1$  sets
- 3 Stores the  $k$  compact, trained models in the cells of a  $k$ -by-1 cell vector in the `Trained` property of the cross-validated model.

To create a cross-validated model, you can use one of these four name-value pair arguments only: `CVPartition`, `Holdout`, `KFold`, or `Leaveout`.

Example: 'kFold', 5

Data Types: single | double

### Leaveout — Leave-one-out cross-validation flag

'off' (default) | 'on'

Leave-one-out cross-validation flag, specified as the comma-separated pair consisting of 'Leaveout' and 'on' or 'off'. If you specify 'Leaveout', 'on', then, for each of the  $n$  observations, where  $n$  is `size(Mdl.X, 1)`, the software:

- 1 Reserves the observation as validation data, and trains the model using the other  $n - 1$  observations
- 2 Stores the  $n$  compact, trained models in the cells of an  $n$ -by-1 cell vector in the `Trained` property of the cross-validated model.

To create a cross-validated model, you can use one of these four name-value pair arguments only: `CVPartition`, `Holdout`, `kFold`, or `Leaveout`.

Example: 'Leaveout', 'on'

Data Types: char

### Hyperparameter Optimization

#### Distance — Distance metric

character vector | function handle

Distance metric, specified as the comma-separated pair consisting of 'Distance' and a valid distance metric name or function handle. The allowable distance metric names depend on your choice of a neighbor-searcher method (see `NSMethod`).

NSMethod	Distance Metric Names
exhaustive	Any distance metric of <code>ExhaustiveSearcher</code>
kdtree	'cityblock', 'chebychev', 'euclidean', or 'minkowski'

This table includes valid distance metrics of `ExhaustiveSearcher`.

Distance Metric Names	Description
'cityblock'	City block distance.
'chebychev'	Chebychev distance (maximum coordinate difference).

Distance Metric Names	Description
'correlation'	One minus the sample linear correlation between observations (treated as sequences of values).
'cosine'	One minus the cosine of the included angle between observations (treated as vectors).
'euclidean'	Euclidean distance.
'hamming'	Hamming distance, percentage of coordinates that differ.
'jaccard'	One minus the Jaccard coefficient, the percentage of nonzero coordinates that differ.
'mahalanobis'	Mahalanobis distance, computed using a positive definite covariance matrix $C$ . The default value of $C$ is the sample covariance matrix of $X$ , as computed by <code>nancov(X)</code> . To specify a different value for $C$ , use the 'Cov' name-value pair argument.
'minkowski'	Minkowski distance. The default exponent is 2. To specify a different exponent, use the 'Exponent' name-value pair argument.
'seuclidean'	Standardized Euclidean distance. Each coordinate difference between $X$ and a query point is scaled, meaning divided by a scale value $S$ . The default value of $S$ is the standard deviation computed from $X$ , $S = \text{nanstd}(X)$ . To specify another value for $S$ , use the <code>Scale</code> name-value pair argument.
'spearman'	One minus the sample Spearman's rank correlation between observations (treated as sequences of values).

Distance Metric Names	Description
@ <i>distfun</i>	<p>Distance function handle. <i>distfun</i> has the form</p> <pre>function D2 = DISTFUN(ZI,ZJ) % calculation of distance ...</pre> <p>where</p> <ul style="list-style-type: none"> <li>• ZI is a 1-by-N vector containing one row of X or Y.</li> <li>• ZJ is an M2-by-N matrix containing multiple rows of X or Y.</li> <li>• D2 is an M2-by-1 vector of distances, and D2(k) is the distance between observations ZI and ZJ(k, :).</li> </ul>

If you specify `CategoricalPredictors` as 'all', then the default distance metric is 'hamming'. Otherwise, the default distance metric is 'euclidean'.

For definitions, see “Distance Metrics” on page 18-16.

Example: 'Distance', 'minkowski'

Data Types: `function_handle` | `char`

### DistanceWeight — Distance weighting function

'equal' (default) | 'inverse' | 'squaredinverse' | function handle

Distance weighting function, specified as the comma-separated pair consisting of 'DistanceWeight' and either a function handle or one of the values in this table.

Value	Description
'equal'	No weighting
'inverse'	Weight is 1/distance
'squaredinverse'	Weight is 1/distance <sup>2</sup>
@ <i>fcn</i>	<i>fcn</i> is a function that accepts a matrix of nonnegative distances, and returns a matrix the same size containing nonnegative distance weights. For example, 'squaredinverse' is equivalent to @ (d) d.^(-2).

Example: 'DistanceWeight', 'inverse'

Data Types: `function_handle` | `char`

### **Exponent** — Minkowski distance exponent

2 (default) | positive scalar value

Minkowski distance exponent, specified as the comma-separated pair consisting of 'Exponent' and a positive scalar value. This argument is only valid when 'Distance' is 'minkowski'.

Example: 'Exponent', 3

Data Types: `single` | `double`

### **NumNeighbors** — Number of nearest neighbors to find

1 (default) | positive integer value

Number of nearest neighbors in X to find for classifying each point when predicting, specified as the comma-separated pair consisting of 'NumNeighbors' and a positive integer value.

Example: 'NumNeighbors', 3

Data Types: `single` | `double`

### **Standardize** — Flag to standardize predictors

false (default) | true

Flag to standardize the predictors, specified as the comma-separated pair consisting of 'Standardize' and true (1) or false (0).

If you set 'Standardize', true, then the software centers and scales each column of the predictor data (X) by the column mean and standard deviation, respectively.

The software does not standardize categorical predictors, and throws an error if all predictors are categorical.

You cannot simultaneously specify 'Standardize', 1 and either of 'Scale' or 'Cov'.

It is good practice to standardize the predictor data.

Example: 'Standardize', true

Data Types: `logical`

## Hyperparameter Optimization

### `OptimizeHyperparameters` — Parameters to optimize

'none' (default) | 'auto' | 'all' | cell array of eligible parameter names | vector of `optimizableVariable` objects

Parameters to optimize, specified as:

- 'none' — Do not optimize.
- 'auto' — Use {'Distance', 'NumNeighbors'}
- 'all' — Optimize all eligible parameters.
- Cell array of eligible parameter names
- Vector of `optimizableVariable` objects, typically the output of hyperparameters

The optimization attempts to minimize the cross-validation loss (error) for `fitcknn` by varying the parameters. For information about cross-validation loss (albeit in a different context), see “Classification Loss” on page 32-3250. To control the cross-validation type and other aspects of the optimization, use the `HyperparameterOptimizationOptions` name-value pair.

---

**Note** `OptimizeHyperparameters` values override any values you set using other name-value pairs. For example, setting `OptimizeHyperparameters` to 'auto' causes the 'auto' values to apply.

---

The eligible parameters for `fitcknn` are:

- `Distance` — `fitcknn` searches among 'cityblock', 'chebychev', 'correlation', 'cosine', 'euclidean', 'hamming', 'jaccard', 'mahalanobis', 'minkowski', 'seuclidean', and 'spearman'.
- `DistanceWeight` — `fitcknn` searches among 'equal', 'inverse', and 'squaredinverse'.
- `Exponent` — `fitcknn` searches among positive real values, by default in the range [0.5, 3].
- `NumNeighbors` — `fitcknn` searches among positive integer values, by default log-scaled in the range [1, max(2, round(NumObservations/2))].



- Standardize — fitcknn searches among the values 'true' and 'false'.

Set nondefault parameters by passing a vector of `optimizableVariable` objects that have nondefault values. For example,

```
load fisheriris
params = hyperparameters('fitcknn',meas,species);
params(1).Range = [1,20];
```

Pass `params` as the value of `OptimizeHyperparameters`.

By default, iterative display appears at the command line, and plots appear according to the number of hyperparameters in the optimization. For the optimization and plots, the objective function is  $\log(1 + \text{cross-validation loss})$  for regression, and the misclassification rate for classification. To control the iterative display, set the `HyperparameterOptimizationOptions` name-value pair, `Verbose` field. To control the plots, set the `HyperparameterOptimizationOptions` name-value pair, `ShowPlots` field.

For an example, see “Optimize Fitted KNN Classifier” on page 32-1974.

Example: 'auto'

Data Types: char | cell

### **HyperparameterOptimizationOptions** — Options for optimization structure

Options for optimization, specified as a structure. Modifies the effect of the `OptimizeHyperparameters` name-value pair. All fields in the structure are optional.

Field Name	Values	Default
Optimizer	<ul style="list-style-type: none"> <li>'bayesopt' — Use Bayesian optimization. Internally, this setting calls bayesopt.</li> <li>'gridsearch' — Use grid search with NumGridDivisions values per dimension.</li> <li>'randomsearch' — Search at random among MaxObjectiveEvaluations points.</li> </ul> <p>'gridsearch' searches in a random order, using uniform sampling without replacement from the grid. After optimization, you can get a table in grid order by using the command sortrows (Mdl.ParameterOptimizationResults).</p>	'bayesopt'
AcquisitionFunctionName	<ul style="list-style-type: none"> <li>'expected-improvement-per-second-plus'</li> <li>'expected-improvement'</li> <li>'expected-improvement-plus'</li> <li>'expected-improvement-per-second'</li> <li>'lower-confidence-bound'</li> <li>'probability-of-improvement'</li> </ul> <p>For details, see the bayesopt AcquisitionFunctionName name-value pair, or “Acquisition Function Types” on page 10-4.</p>	'expected-improvement-per-second-plus'
MaxObjectiveEvaluations	Maximum number of objective function evaluations.	30 for 'bayesopt' or 'randomsearch', and the entire grid for 'gridsearch'

Field Name	Values	Default
MaxTime	Time limit, specified as a positive real. The time limit is in seconds, as measured by <code>tic</code> and <code>toc</code> . Run time can exceed <code>MaxTime</code> because <code>MaxTime</code> does not interrupt function evaluations.	Inf
NumGridDivisions	For 'gridsearch', the number of values in each dimension. Can be a vector of positive integers giving the number of values for each dimension, or a scalar that applies to all dimensions. Ignored for categorical variables.	10
ShowPlots	Logical value indicating whether to show plots. If <code>true</code> , plots the best objective function value against iteration number. If there are one or two optimization parameters, and if <code>Optimizer</code> is 'bayesopt', then <code>ShowPlots</code> also plots a model of the objective function against the parameters.	<code>true</code>
SaveIntermediateResults	Logical value indicating whether to save results when <code>Optimizer</code> is 'bayesopt'. If <code>true</code> , overwrites a workspace variable named 'BayesoptResults' at each iteration. The variable is a <code>BayesianOptimization</code> object.	<code>false</code>
Verbose	Display to the command line. <ul style="list-style-type: none"> <li>• 0 — No iterative display</li> <li>• 1 — Iterative display</li> <li>• 2 — Iterative display with extra information</li> </ul> For details, see the <code>bayesopt</code> <code>Verbose</code> name-value pair.	1

Field Name	Values	Default
UseParallel	Logical value indicating whether to run Bayesian optimization in parallel, which requires Parallel Computing Toolbox. For details, see “Parallel Bayesian Optimization” on page 10-9.	false
Repartition	Logical value indicating whether to repartition the cross-validation at every iteration. If <code>false</code> , the optimizer uses a single partition for the optimization.  <code>true</code> usually gives the most robust results because this setting takes partitioning noise into account. However, for good results, <code>true</code> requires at least twice as many function evaluations.	false
Use no more than one of the following three field names.		
CVPartition	A <code>cvpartition</code> object, as created by <code>cvpartition</code>	Kfold = 5
Holdout	A scalar in the range (0, 1) representing the holdout fraction.	
Kfold	An integer greater than 1.	

Example: `struct('MaxObjectiveEvaluations', 60)`

Data Types: `struct`

## Output Arguments

### **mdl** — Trained $k$ -nearest neighbor classification model

ClassificationKNN model object | ClassificationPartitionedModel cross-validated model object

Trained  $k$ -nearest neighbor classification model, returned as a `ClassificationKNN` model object or a `ClassificationPartitionedModel` cross-validated model object.

If you set any of the name-value pair arguments `KFold`, `Holdout`, `CrossVal`, or `CVPartition`, then `Mdl` is a `ClassificationPartitionedModel` cross-validated model object. Otherwise, `Mdl` is a `ClassificationKNN` model object.

To reference properties of `Mdl`, use dot notation. For example, to display the distance metric at the Command Window, enter `Mdl.Distance`.

## Definitions

### Prediction

`ClassificationKNN` predicts the classification of a point  $X_{new}$  using a procedure equivalent to this:

- 1 Find the `NumNeighbors` points in the training set  $X$  that are nearest to  $X_{new}$ .
- 2 Find the `NumNeighbors` response values  $Y$  to those nearest points.
- 3 Assign the classification label  $Y_{new}$  that has the largest posterior probability among the values in  $Y$ .

For details, see “Posterior Probability” on page 32-4897 in the `predict` documentation.

### Tips

- After training a `ClassificationKNN` model object by using `fitcknn`, use the function `predict` and the trained model object to generate C code that predicts labels for new data. For details, see “Code Generation” on page 32-0 .

### Algorithms

- `NaNs` or `<undefined>s` indicate missing observations. The following describes the behavior of `fitcknn` when the data set or weights contain missing observations.
  - If any value of  $Y$  or any weight is missing, then `fitcknn` removes those values from  $Y$ , the weights, and the corresponding rows of  $X$  from the data. The software renormalizes the weights to sum to 1.

- If you specify to standardize predictors ('Standardize', 1) or the standardized Euclidean distance ('Distance', 'seuclidean') without a scale, then `fitcknn` removes missing observations from individual predictors before computing the mean and standard deviation. In other words, the software implements `nanmean` and `nanstd` on each predictor.
- If you specify the Mahalanobis distance ('Distance', 'mahalanobis') without its covariance matrix, then `fitcknn` removes rows of `X` that contain at least one missing value. In other words, the software implements `nancov` on the predictor matrix `X`.
- Suppose that you set 'Standardize', 1.

- If you also specify `Prior` or `Weights`, then the software takes the observation weights into account. Specifically, the weighted mean of predictor  $j$  is

$$\bar{x}_j = \sum_{B_j} w_k x_{jk}$$

and the weighted standard deviation is

$$s_j = \sum_{B_j} w_k (x_{jk} - \bar{x}_j),$$

where  $B_j$  is the set of indices  $k$  for which  $x_{jk}$  and  $w_k$  are not missing.

- If you also set 'Distance', 'mahalanobis' or 'Distance', 'seuclidean', then you cannot specify `Scale` or `Cov`. Instead, the software:
  - 1 Computes the means and standard deviations of each predictor
  - 2 Standardizes the data using the results of step 1
  - 3 Computes the distance parameter values using their respective default.
- If you specify `Scale` and either of `Prior` or `Weights`, then the software scales observed distances by the weighted standard deviations.
- If you specify `Cov` and either of `Prior` or `Weights`, then the software applies the weighted covariance matrix to the distances. In other words,

$$\text{Cov} = \frac{\sum_B w_j}{\left(\sum_B w_j\right)^2 - \sum_B w_j^2} \sum_B w_j (x_j - \bar{x})' (x_j - \bar{x}),$$

where  $B$  is the set of indices  $j$  for which the observation  $x_j$  does not have any missing values and  $w_j$  is not missing.

## Alternatives

Although `fitcknn` can train a multiclass KNN classifier, you can reduce a multiclass learning problem to a series of KNN binary learners using `fitcecoc`.

## See Also

`ClassificationKNN` | `ClassificationPartitionedModel` | `fitcecoc` | `fitcensemble` | `predict` | `templateKNN`

## Topics

“Construct KNN Classifier” on page 18-36

“Modify KNN Classifier” on page 18-39

“Classification Using Nearest Neighbors” on page 18-16

**Introduced in R2014a**

## fitclinear

Fit linear classification model to high-dimensional data

`fitclinear` trains linear classification models for two-class (binary) learning with high-dimensional, full or sparse predictor data. Available linear classification models include regularized support vector machines (SVM) and logistic regression models. `fitclinear` minimizes the objective function using techniques that reduce computing time (e.g., stochastic gradient descent).

A high-dimensional data set includes many predictor variables. Although such a data set can consume a significant fraction of memory, it must fit in the MATLAB Workspace. For low- through medium-dimensional predictor data sets, see “Alternatives for Lower-Dimensional Data” on page 22-2.

To train a linear classification model for multiclass learning by combining SVM or logistic regression binary classifiers using error-correcting output codes, see `fitcecoc`.

## Syntax

```
Mdl = fitclinear(X,Y)
Mdl = fitclinear(X,Y,Name,Value)
[Mdl,FitInfo] = fitclinear(____)
[Mdl,FitInfo,HyperparameterOptimizationResults] = fitclinear(____)
```

## Description

`Mdl = fitclinear(X,Y)` returns a trained linear classification model object that contains the results of fitting a binary support vector machine to the predictors `X` and class labels `Y`.

`Mdl = fitclinear(X,Y,Name,Value)` returns a trained linear classification model with additional options specified by one or more `Name,Value` pair arguments. For example, you can specify that the columns of the predictor matrix correspond to observations, implement logistic regression, or specify to cross-validate. It is good practice to cross-validate using the `KfoldName,Value` pair argument. The cross-validation results determine how well the model generalizes.



`[Mdl,FitInfo] = fitclinear(____)` also returns optimization details using any of the previous syntaxes. You cannot request `FitInfo` for cross-validated models.

`[Mdl,FitInfo,HyperparameterOptimizationResults] = fitclinear(____)` also returns hyperparameter optimization details when you pass an `OptimizeHyperparameters` name-value pair.

## Examples

### Train Linear Classification Model

Train a binary, linear classification model using support vector machines, dual SGD, and ridge regularization.

Load the NLP data set.

```
load nlpdata
```

`X` is a sparse matrix of predictor data, and `Y` is a categorical vector of class labels. There are more than two classes in the data.

Identify the labels that correspond to the Statistics and Machine Learning Toolbox™ documentation web pages.

```
Ystats = Y == 'stats';
```

Train a binary, linear classification model that can identify whether the word counts in a documentation web page are from the Statistics and Machine Learning Toolbox™ documentation. Train the algorithm using the entire data set. Determine how well the optimization algorithm fit the model to the data by extracting a fit summary.

```
rng(1); % For reproducibility
[Mdl,FitInfo] = fitclinear(X,Ystats)
```

```
Mdl =
```

```
ClassificationLinear
  ResponseName: 'Y'
   ClassNames: [0 1]
  ScoreTransform: 'none'
```

```
Beta: [34023x1 double]
Bias: -1.0059
Lambda: 3.1674e-05
Learner: 'svm'
```

```
FitInfo =
```

```
struct with fields:
```

```
    Lambda: 3.1674e-05
    Objective: 5.3783e-04
    PassLimit: 10
    NumPasses: 10
    BatchLimit: []
    NumIterations: 238561
    GradientNorm: NaN
    GradientTolerance: 0
    RelativeChangeInBeta: 0.0562
    BetaTolerance: 1.0000e-04
    DeltaGradient: 1.4582
    DeltaGradientTolerance: 1
    TerminationCode: 0
    TerminationStatus: {'Iteration limit exceeded.'}
    Alpha: [31572x1 double]
    History: []
    FitTime: 0.2200
    Solver: {'dual'}
```

`Mdl` is a `ClassificationLinear` model. You can pass `Mdl` and the training or new data to `loss` to inspect the in-sample classification error. Or, you can pass `Mdl` and new predictor data to `predict` to predict class labels for new observations.

`FitInfo` is a structure array containing, among other things, the termination status (`TerminationStatus`) and how long the solver took to fit the model to the data (`FitTime`). It is good practice to use `FitInfo` to determine whether optimization-termination measurements are satisfactory. Because training time is small, you can try to retrain the model, but increase the number of passes through the data. This can improve measures like `DeltaGradient`.

## Find Good Lasso Penalty Using Cross-Validation

To determine a good lasso-penalty strength for a linear classification model that uses a logistic regression learner, implement 5-fold cross-validation.

Load the NLP data set.

```
load nlpdata
```

X is a sparse matrix of predictor data, and Y is a categorical vector of class labels. There are more than two classes in the data.

The models should identify whether the word counts in a web page are from the Statistics and Machine Learning Toolbox™ documentation. So, identify the labels that correspond to the Statistics and Machine Learning Toolbox™ documentation web pages.

```
Ystats = Y == 'stats';
```

Create a set of 11 logarithmically-spaced regularization strengths from  $10^{-6}$  through  $10^{-0.5}$ .

```
Lambda = logspace(-6,-0.5,11);
```

Cross-validate the models. To increase execution speed, transpose the predictor data and specify that the observations are in columns. Estimate the coefficients using SpaRSA. Lower the tolerance on the gradient of the objective function to  $1e-8$ .

```
X = X';
rng(10); % For reproducibility
CVMdl = fitlinear(X,Ystats,'ObservationsIn','columns','KFold',5,...
    'Learner','logistic','Solver','sparsa','Regularization','lasso',...
    'Lambda',Lambda,'GradientTolerance',1e-8)
```

```
numCLModels = numel(CVMdl.Trained)
```

```
CVMdl =
```

```
classreg.learning.partition.ClassificationPartitionedLinear
  CrossValidatedModel: 'Linear'
    ResponseName: 'Y'
  NumObservations: 31572
        KFold: 5
    Partition: [1x1 cvpartition]
```

```
ClassNames: [0 1]
ScoreTransform: 'none'
```

```
numCLModels =
    5
```

`CVMD1` is a `ClassificationPartitionedLinear` model. Because `fitclinear` implements 5-fold cross-validation, `CVMD1` contains 5 `ClassificationLinear` models that the software trains on each fold.

Display the first trained linear classification model.

```
Mdl1 = CVMD1.Trained{1}
```

```
Mdl1 =
    ClassificationLinear
      ResponseName: 'Y'
      ClassNames: [0 1]
      ScoreTransform: 'logit'
      Beta: [34023x11 double]
      Bias: [1x11 double]
      Lambda: [1x11 double]
      Learner: 'logistic'
```

`Mdl1` is a `ClassificationLinear` model object. `fitclinear` constructed `Mdl1` by training on the first four folds. Because `Lambda` is a sequence of regularization strengths, you can think of `Mdl1` as 11 models, one for each regularization strength in `Lambda`.

Estimate the cross-validated classification error.

```
ce = kfoldLoss(CVMD1);
```

Because there are 11 regularization strengths, `ce` is a 1-by-11 vector of classification error rates.

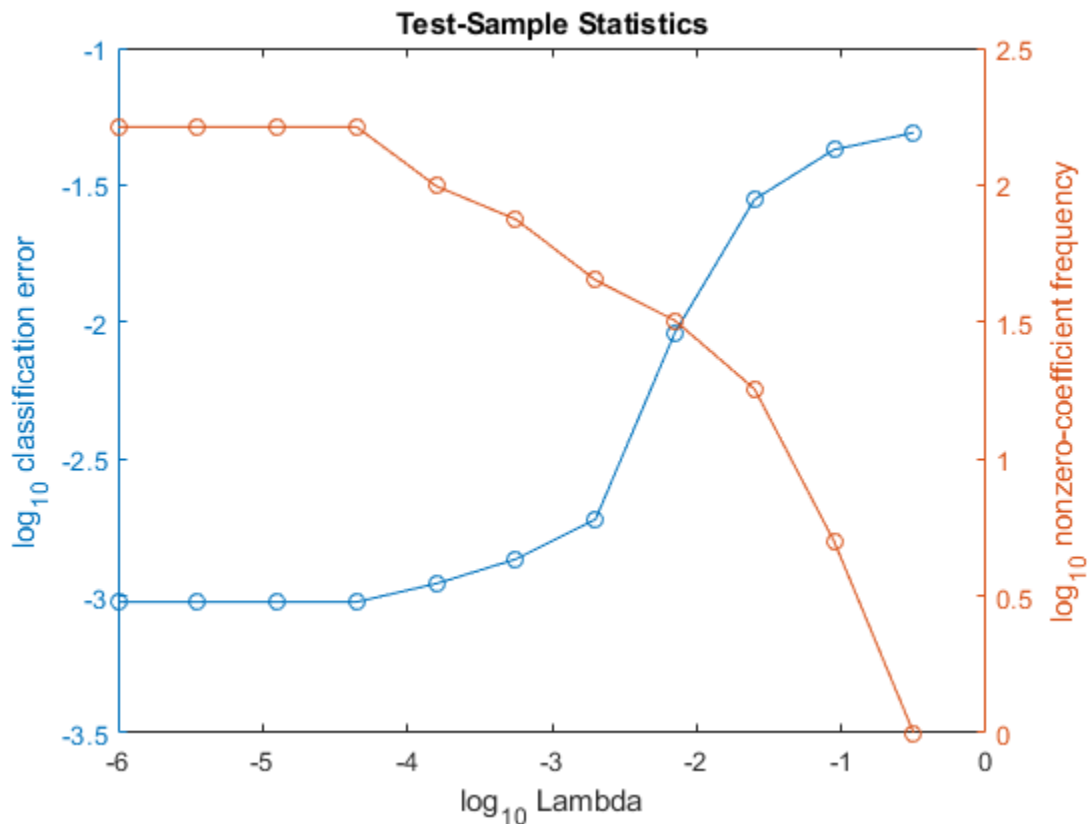
Higher values of `Lambda` lead to predictor variable sparsity, which is a good quality of a classifier. For each regularization strength, train a linear classification model using the

entire data set and the same options as when you cross-validated the models. Determine the number of nonzero coefficients per model.

```
Mdl = fitlinear(X,Ystats,'ObservationsIn','columns',...  
    'Learner','logistic','Solver','sparsa','Regularization','lasso',...  
    'Lambda',Lambda,'GradientTolerance',1e-8);  
numNZCoeff = sum(Mdl.Beta~=0);
```

In the same figure, plot the cross-validated, classification error rates and frequency of nonzero coefficients for each regularization strength. Plot all variables on the log scale.

```
figure;  
[h,hL1,hL2] = plotyy(log10(Lambda),log10(ce),...  
    log10(Lambda),log10(numNZCoeff));  
hL1.Marker = 'o';  
hL2.Marker = 'o';  
ylabel(h(1),'log_{10} classification error')  
ylabel(h(2),'log_{10} nonzero-coefficient frequency')  
xlabel('log_{10} Lambda')  
title('Test-Sample Statistics')  
hold off
```



Choose the index of the regularization strength that balances predictor variable sparsity and low classification error. In this case, a value between  $10^{-4}$  to  $10^{-1}$  should suffice.

```
idxFinal = 7;
```

Select the model from `Mdl` with the chosen regularization strength.

```
MdlFinal = selectModels(Mdl,idxFinal);
```

`MdlFinal` is a `ClassificationLinear` model containing one regularization strength. To estimate labels for new observations, pass `MdlFinal` and the new data to `predict`.

## Optimize Linear Classifier

This example shows how to minimize the cross-validation error in a linear classifier using `fitlinear`. The example uses the NLP data set.

Load the NLP data set.

```
load nlpdata
```

`X` is a sparse matrix of predictor data, and `Y` is a categorical vector of class labels. There are more than two classes in the data.

The models should identify whether the word counts in a web page are from the Statistics and Machine Learning Toolbox™ documentation. Identify the relevant labels.

```
X = X';
Ystats = Y == 'stats';
```

Optimize the classification using the 'auto' parameters.

For reproducibility, set the random seed and use the 'expected-improvement-plus' acquisition function.

```
rng default
Mdl = fitlinear(X,Ystats,'ObservationsIn','columns','Solver','sparsa',...
    'OptimizeHyperparameters','auto','HyperparameterOptimizationOptions',...
    struct('AcquisitionFunctionName','expected-improvement-plus'))
```

```
=====
| Iter | Eval  | Objective | Objective | BestSoFar | BestSoFar | Lambda
|      | result|           | runtime   | (observed) | (estim.)  |
=====
|  1  | Best  | 0.041619 | 13.584    | 0.041619 | 0.041619 | 0.077903
|  2  | Best  | 0.00079184 | 8.5253    | 0.00079184 | 0.0029367 | 2.1405e-09
|  3  | Accept| 0.049221 | 10.163    | 0.00079184 | 0.00082068 | 0.72101
|  4  | Accept| 0.00079184 | 10.252    | 0.00079184 | 0.000815 | 3.4734e-07
|  5  | Accept| 0.00079184 | 11.231    | 0.00079184 | 0.00079162 | 6.3377e-08
|  6  | Best  | 0.00076017 | 8.3655    | 0.00076017 | 0.00075995 | 3.1802e-10
|  7  | Accept| 0.00088686 | 9.6865    | 0.00076017 | 0.00075831 | 3.1843e-10
|  8  | Accept| 0.00076017 | 11.263    | 0.00076017 | 0.00076866 | 1.9789e-07
|  9  | Best  | 0.00069682 | 9.0186    | 0.00069682 | 0.00066731 | 3.3305e-08
| 10  | Accept| 0.00085519 | 7.2647    | 0.00069682 | 0.00065996 | 2.6602e-09
| 11  | Accept| 0.00076017 | 9.3069    | 0.00069682 | 0.00068921 | 6.441e-08
| 12  | Accept| 0.00079184 | 8.4294    | 0.00069682 | 0.0006901 | 1.3131e-07
=====
```

13	Accept	0.00069682	8.9486	0.00069682	0.00069274	6.7517e-08
14	Accept	0.00072849	8.2791	0.00069682	0.00069056	1.2829e-08
15	Accept	0.00069682	8.686	0.00069682	0.00069256	6.4082e-08
16	Accept	0.00072849	7.1697	0.00069682	0.00069301	3.1954e-10
17	Accept	0.0011719	9.18	0.00069682	0.00070318	0.00082749
18	Accept	0.00076017	7.0506	0.00069682	0.00070568	5.1136e-05
19	Accept	0.00095021	5.6945	0.00069682	0.00070718	0.00022517
20	Accept	0.00085519	14.355	0.00069682	0.0007067	4.9783e-06
=====						
Iter	Eval	Objective	Objective	BestSoFar	BestSoFar	Lambda
	result		runtime	(observed)	(estim.)	
=====						
21	Accept	0.0010452	49.509	0.00069682	0.00072119	4.7908e-06
22	Accept	0.00095021	39.127	0.00069682	0.00071239	1.1954e-06
23	Accept	0.00076017	7.9483	0.00069682	0.00071249	1.3654e-08
24	Accept	0.00069682	7.4718	0.00069682	0.000712	7.2517e-10
25	Accept	0.00082351	12.22	0.00069682	0.00071206	1.8265e-05
26	Accept	0.00069682	7.7773	0.00069682	0.00071193	6.916e-10
27	Accept	0.00079184	11.073	0.00069682	0.00071197	1.1111e-06
28	Accept	0.00076017	8.1148	0.00069682	0.00071585	2.5561e-08
29	Accept	0.00076017	9.378	0.00069682	0.000716	6.7349e-10
30	Accept	0.00076017	8.5543	0.00069682	0.00071635	6.5876e-09

Optimization completed.

MaxObjectiveEvaluations of 30 reached.

Total function evaluations: 30

Total elapsed time: 409.9645 seconds.

Total objective function evaluation time: 347.6249

Best observed feasible point:

Lambda	Learner
-----	-----
3.3305e-08	logistic

Observed objective function value = 0.00069682

Estimated objective function value = 0.00071635

Function evaluation time = 9.0186

Best estimated feasible point (according to models):

Lambda	Learner
-----	-----



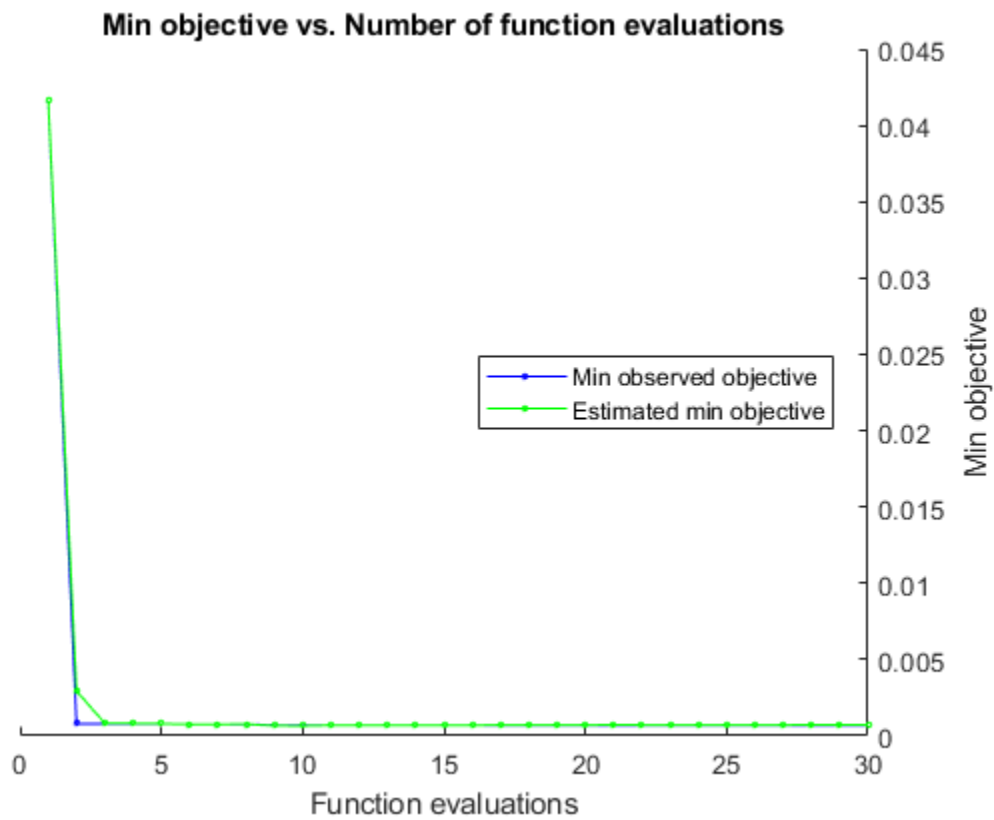
6.4082e-08    logistic

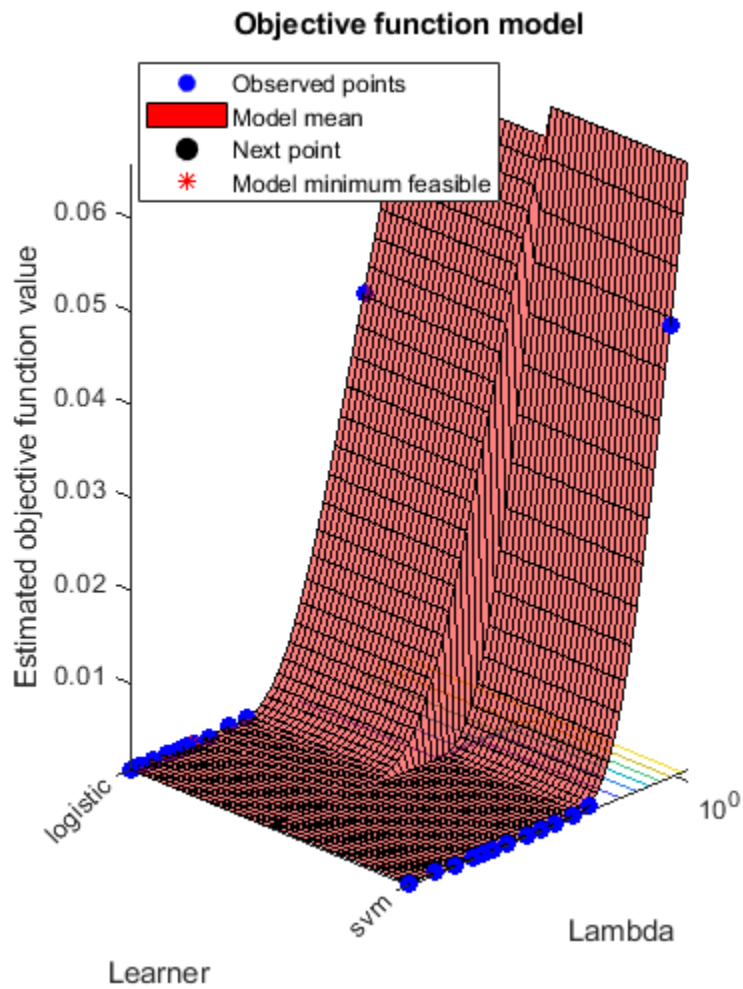
Estimated objective function value = 0.00071635

Estimated function evaluation time = 8.9397

Mdl =

```
ClassificationLinear
  ResponseName: 'Y'
  ClassNames: [0 1]
  ScoreTransform: 'logit'
    Beta: [34023x1 double]
    Bias: -10.1207
    Lambda: 6.4082e-08
  Learner: 'logistic'
```





## Input Arguments

**x** — Predictor data

full matrix | sparse matrix

Predictor data, specified as an  $n$ -by- $p$  full or sparse matrix.

The length of  $Y$  and the number of observations in  $X$  must be equal.

---

**Note** If you orient your predictor matrix so that observations correspond to columns and specify 'ObservationsIn', 'columns', then you might experience a significant reduction in optimization-execution time.

---

Data Types: `single` | `double`

### $Y$ — Class labels

categorical array | character array | logical vector | vector of numeric values | cell array of character vectors

Class labels to which the classification model is trained, specified as a categorical or character array, logical or numeric vector, or cell array of character vectors.

- `fitclinear` only supports binary classification. Either  $Y$  must contain exactly two distinct classes, or you must specify two classes for training using the 'ClassNames' name-value pair argument. For multiclass learning, see `fitcecoc`.
- If  $Y$  is a character array, then each element must correspond to one row of the array.
- The length of  $Y$  and the number of observations in  $X$  must be equal.
- It is good practice to specify the class order using the `ClassNames` name-value pair argument.

Data Types: `char` | `cell` | `categorical` | `logical` | `single` | `double`

---

**Note** `fitclinear` removes missing observations, that is, observations with any of these characteristics:

- NaN, empty character vector (' '), and `<undefined>` elements in the response ( $Y$  or `ValidationData{2}`)
- At least one NaN value in a predictor observation (row in  $X$  or `ValidationData{1}`)
- NaN value or 0 weight (`Weights` or `ValidationData{3}`)

For memory-usage economy, it is best practice to remove observations containing missing values from your training data manually before training.

---

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

---

**Note** You cannot use any cross-validation name-value pair along with `OptimizeHyperparameters`. You can modify the cross-validation for `OptimizeHyperparameters` only by using the `HyperparameterOptimizationOptions` name-value pair.

---

Example:

`'ObservationsIn', 'columns', 'Learner', 'logistic', 'CrossVal', 'on'` specifies that the columns of the predictor matrix corresponds to observations, to implement logistic regression, to implement 10-fold cross-validation.

### Linear Classification Options

#### **Lambda** — Regularization term strength

'auto' (default) | nonnegative scalar | vector of nonnegative values

Regularization term strength, specified as the comma-separated pair consisting of 'Lambda' and 'auto', a nonnegative scalar, or a vector of nonnegative values.

- For 'auto',  $\text{Lambda} = 1/n$ .
  - If you specify a cross-validation, name-value pair argument (e.g., `CrossVal`), then  $n$  is the number of in-fold observations.
  - Otherwise,  $n$  is the training sample size.
- For a vector of nonnegative values, the software sequentially optimizes the objective function for each distinct value in `Lambda` in ascending order.
  - If `Solver` is 'sgd' or 'asgd' and `Regularization` is 'lasso', then the software does not use the previous coefficient estimates as a warm start on page 32-2043 for the next optimization iteration. Otherwise, the software uses warm starts.
  - If `Regularization` is 'lasso', then any coefficient estimate of 0 retains its value when the software optimizes using subsequent values in `Lambda`.

Returns coefficient estimates for all optimization iterations.

Example: `'Lambda', 10.^(-(10:-2:2))`

Data Types: `char` | `double` | `single`

### Learner — Linear classification model type

`'svm'` (default) | `'logistic'`

Linear classification model type, specified as the comma-separated pair consisting of `'Learner'` and `'svm'` or `'logistic'`.

In this table,  $f(x) = x\beta + b$ .

- $\beta$  is a vector of  $p$  coefficients.
- $x$  is an observation from  $p$  predictor variables.
- $b$  is the scalar bias.

Value	Algorithm	Response Range	Loss Function
<code>'svm'</code>	Support vector machine	$y \in \{-1, 1\}$ ; 1 for the positive class, and -1 otherwise.	Hinge: $\ell[y, f(x)] = \max[0, 1 - yf(x)]$
<code>'logistic'</code>	Logistic regression	Same as <code>'svm'</code>	Deviance (logistic): $\ell[y, f(x)] = \log\{1 + \exp[-yf(x)]\}$

Example: `'Learner', 'logistic'`

Data Types: `char`

### ObservationsIn — Predictor data observation dimension

`'rows'` (default) | `'columns'`

Predictor data observation dimension, specified as the comma-separated pair consisting of `'ObservationsIn'` and `'columns'` or `'rows'`.

---

**Note** If you orient your predictor matrix so that observations correspond to columns and specify `'ObservationsIn', 'columns'`, then you might experience a significant reduction in optimization-execution time.

---

Data Types: char

### Regularization — Complexity penalty type

'lasso' | 'ridge'

Complexity penalty type, specified as the comma-separated pair consisting of 'Regularization' and 'lasso' or 'ridge'.

The software composes the objective function for minimization from the sum of the average loss function (see `Learner`) and the regularization term in this table.

Value	Description
'lasso'	Lasso (L1) penalty: $\lambda \sum_{j=1}^p  \beta_j $
'ridge'	Ridge (L2) penalty: $\frac{\lambda}{2} \sum_{j=1}^p \beta_j^2$

To specify the regularization term strength, which is  $\lambda$  in the expressions, use `Lambda`.

The software excludes the bias term ( $\beta_0$ ) from the regularization penalty.

If `Solver` is 'sparsa', then the default value of `Regularization` is 'lasso'. Otherwise, the default is 'ridge'.

---

### Tip

- For predictor variable selection, specify 'lasso'.
  - For optimization accuracy, specify 'ridge'.
- 

Example: 'Regularization', 'lasso'

Data Types: char

### solver — Objective function minimization technique

'sgd' | 'asgd' | 'dual' | 'bfgs' | 'lbfgs' | 'sparsa' | cell vector of character vectors

Objective function minimization technique, specified as the comma-separated pair consisting of 'Solver' and a value or cell vector of values in this table.

Value	Description	Restrictions
'sgd'	Stochastic gradient descent (SGD) [5][3]	
'asgd'	Average stochastic gradient descent (ASGD) [8]	
'dual'	Dual SGD for SVM [2][7]	Regularization must be 'ridge' and Learner must be 'svm'.
'bfgs'	Broyden-Fletcher-Goldfarb-Shanno quasi-Newton algorithm (BFGS) [4]	Inefficient if X is very high-dimensional.
'lbfgs'	Limited-memory BFGS (LBFGS) [4]	Regularization must be 'ridge'.
'sparsa'	Sparse Reconstruction by Separable Approximation (SpaRSA) [6]	Regularization must be 'lasso'.

If you specify:

- A ridge penalty (see Regularization) and X contains 100 or fewer predictor variables, then the default solver is 'bfgs'.
- An SVM model (see Learner), a ridge penalty, and X contains more than 100 predictor variables, then the default solver is 'dual'.
- A lasso penalty and X contains 100 or fewer predictor variables, then the default solver is 'sparsa'.

Otherwise, the default solver is 'sgd'.

If you specify a cell vector of solver names, then the software uses all solvers in the specified order for each Lambda.

For more details on which solver to choose, see “Tips” on page 32-2043.

Example: 'Solver', {'sgd', 'lbfgs'}

Data Types: char | cell



**Beta — Initial linear coefficient estimates**

`zeros(p, 1)` (default) | numeric vector | numeric matrix

Initial linear coefficient estimates ( $\beta$ ), specified as the comma-separated pair consisting of 'Beta' and a  $p$ -dimensional numeric vector or a  $p$ -by- $L$  numeric matrix.  $p$  is the number of predictor variables in  $X$  and  $L$  is the number of regularization-strength values (for more details, see `Lambda`).

- If you specify a  $p$ -dimensional vector, then the software optimizes the objective function  $L$  times using this process.
  - 1 The software optimizes using `Beta` as the initial value and the minimum value of `Lambda` as the regularization strength.
  - 2 The software optimizes again using the resulting estimate from the previous optimization as a warm start on page 32-2043, and the next smallest value in `Lambda` as the regularization strength.
  - 3 The software implements step 2 until it exhausts all values in `Lambda`.
- If you specify a  $p$ -by- $L$  matrix, then the software optimizes the objective function  $L$  times. At iteration  $j$ , the software uses `Beta(:, j)` as the initial value and, after it sorts `Lambda` in ascending order, uses `Lambda(j)` as the regularization strength.

If you set 'Solver', 'dual', then the software ignores `Beta`.

Data Types: `single` | `double`

**Bias — Initial intercept estimate**

numeric scalar | numeric vector

Initial intercept estimate ( $b$ ), specified as the comma-separated pair consisting of 'Bias' and a numeric scalar or an  $L$ -dimensional numeric vector.  $L$  is the number of regularization-strength values (for more details, see `Lambda`).

- If you specify a scalar, then the software optimizes the objective function  $L$  times using this process.
  - 1 The software optimizes using `Bias` as the initial value and the minimum value of `Lambda` as the regularization strength.
  - 2 The uses the resulting estimate as a warm start to the next optimization iteration, and uses the next smallest value in `Lambda` as the regularization strength.

**3** The software implements step 2 until it exhausts all values in `Lambda`.

- If you specify an  $L$ -dimensional vector, then the software optimizes the objective function  $L$  times. At iteration  $j$ , the software uses `Bias(j)` as the initial value and, after it sorts `Lambda` in ascending order, uses `Lambda(j)` as the regularization strength.
- By default:
  - If `Learner` is 'logistic', then let  $g_j$  be 1 if  $Y(j)$  is the positive class, and -1 otherwise. `Bias` is the weighted average of the  $g$  for training or, for cross-validation, in-fold observations.
  - If `Learner` is 'svm', then `Bias` is 0.

Data Types: `single` | `double`

#### **FitBias** — Linear model intercept inclusion flag

`true` (default) | `false`

Linear model intercept inclusion flag, specified as the comma-separated pair consisting of 'FitBias' and `true` or `false`.

Value	Description
<code>true</code>	The software includes the bias term $b$ in the linear model, and then estimates it.
<code>false</code>	The software sets $b = 0$ during estimation.

Example: 'FitBias', `false`

Data Types: `logical`

#### **PostFitBias** — Flag to fit linear model intercept after optimization

`false` (default) | `true`

Flag to fit the linear model intercept after optimization, specified as the comma-separated pair consisting of 'PostFitBias' and `true` or `false`.

Value	Description
<code>false</code>	The software estimates the bias term $b$ and the coefficients $\beta$ during optimization.

Value	Description
true	To estimate $b$ , the software: <ol style="list-style-type: none"> <li><b>1</b> Estimates <math>\beta</math> and <math>b</math> using the model</li> <li><b>2</b> Estimates classification scores</li> <li><b>3</b> Refits <math>b</math> by placing the threshold on the classification scores that attains maximum accuracy</li> </ol>

If you specify true, then FitBias must be true.

Example: 'PostFitBias', true

Data Types: logical

### Verbose — Verbosity level

0 (default) | nonnegative integer

Verbosity level, specified as the comma-separated pair consisting of 'Verbose' and a nonnegative integer. Verbose controls the amount of diagnostic information fitcllinear displays at the command line.

Value	Description
0	fitcllinear does not display diagnostic information.
1	fitcllinear periodically displays and stores the value of the objective function, gradient magnitude, and other diagnostic information. FitInfo.History contains the diagnostic information.
Any other positive integer	fitcllinear displays and stores diagnostic information at each optimization iteration. FitInfo.History contains the diagnostic information.

Example: 'Verbose', 1

Data Types: double | single

### SGD and ASGD Solver Options

#### **BatchSize** — Mini-batch size

positive integer

Mini-batch size, specified as the comma-separated pair consisting of 'BatchSize' and a positive integer. At each iteration, the software estimates the subgradient using BatchSize observations from the training data.

- If  $X$  is a numeric matrix, then the default value is 10.
- If  $X$  is a sparse matrix, then the default value is  $\max([10, \text{ceil}(\text{sqrt}(\text{ff}))])$ , where  $\text{ff} = \text{numel}(X) / \text{nnz}(X)$  (the fullness factor of  $X$ ).

Example: 'BatchSize', 100

Data Types: single | double

#### **LearnRate** — Learning rate

positive scalar

Learning rate, specified as the comma-separated pair consisting of 'LearnRate' and a positive scalar. LearnRate specifies how many steps to take per iteration. At each iteration, the gradient specifies the direction and magnitude of each step.

- If Regularization is 'ridge', then LearnRate specifies the initial learning rate  $\gamma_0$ . The software determines the learning rate for iteration  $t$ ,  $\gamma_t$ , using

$$\gamma_t = \frac{\gamma_0}{(1 + \lambda \gamma_0 t)^c}.$$

- $\lambda$  is the value of Lambda.
- If Solver is 'sgd', then  $c = 1$ .
- If Solver is 'asgd', then  $c$  is 0.75 [7].
- If Regularization is 'lasso', then, for all iterations, LearnRate is constant.

By default, LearnRate is  $1/\text{sqrt}(1 + \max(\text{sum}(X.^2, \text{obsDim})))$ , where obsDim is 1 if the observations compose the columns of the predictor data  $X$ , and 2 otherwise.

Example: 'LearnRate', 0.01

Data Types: single | double

**OptimizeLearnRate** — Flag to decrease learning rate

true (default) | false

Flag to decrease the learning rate when the software detects divergence (that is, overstepping the minimum), specified as the comma-separated pair consisting of 'OptimizeLearnRate' and true or false.

If `OptimizeLearnRate` is 'true', then:

- 1 For the few optimization iterations, the software starts optimization using `LearnRate` as the learning rate.
- 2 If the value of the objective function increases, then the software restarts and uses half of the current value of the learning rate.
- 3 The software iterates step 2 until the objective function decreases.

Example: 'OptimizeLearnRate', true

Data Types: logical

**TruncationPeriod** — Number of mini-batches between lasso truncation runs

10 (default) | positive integer

Number of mini-batches between lasso truncation runs, specified as the comma-separated pair consisting of 'TruncationPeriod' and a positive integer.

After a truncation run, the software applies a soft threshold to the linear coefficients. That is, after processing  $k = \text{TruncationPeriod}$  mini-batches, the software truncates the estimated coefficient  $j$  using

$$\hat{\beta}_j^* = \begin{cases} \hat{\beta}_j - u_t & \text{if } \hat{\beta}_j > u_t, \\ 0 & \text{if } |\hat{\beta}_j| \leq u_t, \\ \hat{\beta}_j + u_t & \text{if } \hat{\beta}_j < -u_t. \end{cases}$$

- For SGD,  $\hat{\beta}_j$  is the estimate of coefficient  $j$  after processing  $k$  mini-batches.  $u_t = k\gamma_t\lambda$ .  $\gamma_t$  is the learning rate at iteration  $t$ .  $\lambda$  is the value of Lambda.
- For ASGD,  $\hat{\beta}_j$  is the averaged estimate coefficient  $j$  after processing  $k$  mini-batches,  $u_t = k\lambda$ .

If `Regularization` is `'ridge'`, then the software ignores `TruncationPeriod`.

Example: `'TruncationPeriod', 100`

Data Types: `single` | `double`

### Other Classification Options

#### **ClassNames** — Names of classes to use for training

categorical array | cell array of character vectors | character array | logical vector | numeric vector

Names of classes to use for training, specified as the comma-separated pair consisting of `'ClassNames'` and a categorical or character array, logical or numeric vector, or cell array of character vectors. `ClassNames` must be the same data type as `Y`.

If `ClassNames` is a character array, then each element must correspond to one *row* of the array.

Use `ClassNames` to:

- Order the classes during training.
- Specify the order of any input or output argument dimension that corresponds to the class order. For example, use `ClassNames` to specify the order of the dimensions of `Cost` or the column order of classification scores returned by `predict`.
- Select a subset of classes for training. For example, suppose that the set of all distinct class names in `Y` is `{'a', 'b', 'c'}`. To train the model using observations from classes `'a'` and `'c'` only, specify `'ClassNames', {'a', 'c'}`.

The default is the set of all distinct class names in `Y`.

Example: `'ClassNames', {'b', 'g'}`

Data Types: `categorical` | `char` | `logical` | `single` | `double` | `cell`

#### **Cost** — Misclassification cost

square matrix | structure array

Misclassification cost, specified as the comma-separated pair consisting of `'Cost'` and a square matrix or structure. If you specify:

- The square matrix `Cost`, then `Cost(i, j)` is the cost of classifying a point into class `j` if its true class is `i`. That is, the rows correspond to the true class and the columns

correspond to the predicted class. To specify the class order for the corresponding rows and columns of `Cost`, also specify the `ClassNames` name-value pair argument.

- The structure `S`, then it must have two fields:
  - `S.ClassNames`, which contains the class names as a variable of the same data type as `Y`
  - `S.ClassificationCosts`, which contains the cost matrix with rows and columns ordered as in `S.ClassNames`

The default is  $\text{ones}(K) - \text{eye}(K)$ , where  $K$  is the number of distinct classes.

`fitclinear` uses `Cost` to adjust the prior class probabilities specified in `Prior`. Then, `fitclinear` uses the adjusted prior probabilities for training and resets the cost matrix to its default.

Example: `'Cost',[0 2; 1 0]`

Data Types: `single` | `double` | `struct`

### **Prior** — Prior probabilities

`'empirical'` (default) | `'uniform'` | numeric vector | structure array

Prior probabilities for each class, specified as the comma-separated pair consisting of `'Prior'` and a character vector, numeric vector, or a structure.

This table summarizes the available options for setting prior probabilities.

Value	Description
<code>'empirical'</code>	The class prior probabilities are the class relative frequencies in <code>Y</code> .
<code>'uniform'</code>	All class prior probabilities are equal to $1/K$ , where $K$ is the number of classes.
numeric vector	Each element is a class prior probability. Order the elements according to the order that appears in <code>Y</code> . If you specify the order using the <code>'ClassNames'</code> name-value pair argument, then order the elements accordingly.

Value	Description
structure	<p>A structure <math>S</math> with two fields:</p> <ul style="list-style-type: none"> <li>• <math>S.ClassNames</math> contains the class names as a variable of the same type as <math>Y</math>.</li> <li>• <math>S.ClassProbs</math> contains a vector of corresponding prior probabilities.</li> </ul>

`fitclinear` normalizes the prior probabilities in `Prior` to sum to 1.

Example: `struct('ClassNames', {'setosa', 'versicolor'}, 'ClassProbs', 1:2)`

Data Types: `char` | `double` | `single` | `struct`

### ScoreTransform — Score transformation

'none' (default) | character vector | function handle

Score transformation, specified as the comma-separated pair consisting of 'ScoreTransform' and either a character vector or a function handle.

This table summarizes the available character vectors.

Value	Description
'doublelogit'	$1/(1 + e^{-2x})$
'invlogit'	$\log(x / (1-x))$
'ismax'	Set the score for the class with the largest score to 1, and set the scores for all other classes to 0.
'logit'	$1/(1 + e^{-x})$
'none' or 'identity'	$x$ (no transformation)
'sign'	-1 for $x < 0$ 0 for $x = 0$ 1 for $x > 0$
'symmetric'	$2x - 1$
'symmetricismax'	Set the score for the class with the largest score to 1, and set the scores for all other classes to -1.
'symmetriclogit'	$2/(1 + e^{-x}) - 1$



For a MATLAB function or a function you define, use its function handle. The function handle must accept a matrix (the original scores) and return a matrix of the same size (the transformed scores).

Example: 'ScoreTransform', 'logit'

Data Types: char | function\_handle

### **Weights** — Observation weights

numeric vector of positive values

Observation weights, specified as the comma-separated pair consisting of 'Weights' and a numeric vector of positive values. `fitclinear` weighs the observations in `X` with the corresponding value in `Weights`. The size of `Weights` must equal the number of observations in `X`.

`fitclinear` normalizes `Weights` to sum up to the value of the prior probability in the respective class.

By default, `Weights` is ones ( $n, 1$ ), where  $n$  is the number of observations in `X`.

Data Types: double | single

### **Cross-Validation Options**

#### **CrossVal** — Cross-validation flag

'off' (default) | 'on'

Cross-validation flag, specified as the comma-separated pair consisting of 'Crossval' and 'on' or 'off'.

If you specify 'on', then the software implements 10-fold cross-validation.

To override this cross-validation setting, use one of these name-value pair arguments: `CVPartition`, `Holdout`, or `KFold`. To create a cross-validated model, you can use one cross-validation name-value pair argument at a time only.

Example: 'Crossval', 'on'

Data Types: char

#### **CVPartition** — Cross-validation partition

[] (default) | `cvpartition` partition object

Cross-validation partition, specified as the comma-separated pair consisting of 'CVPartition' and a `cvpartition` partition object as created by `cvpartition`. The partition object specifies the type of cross-validation, and also the indexing for training and validation sets.

To create a cross-validated model, you can use one of these four options only: 'CVPartition', 'Holdout', or 'KFold'.

#### **Holdout — Fraction of data for holdout validation**

scalar value in the range (0,1)

Fraction of data used for holdout validation, specified as the comma-separated pair consisting of 'Holdout' and a scalar value in the range (0,1). If you specify 'Holdout',  $p$ , then the software:

- 1 Randomly reserves  $p*100\%$  of the data as validation data, and trains the model using the rest of the data
- 2 Stores the compact, trained model in the `Trained` property of the cross-validated model.

To create a cross-validated model, you can use one of these four options only: 'CVPartition', 'Holdout', or 'KFold'.

Example: 'Holdout', 0.1

Data Types: double | single

#### **KFold — Number of folds**

10 (default) | positive integer value greater than 1

Number of folds to use in a cross-validated classifier, specified as the comma-separated pair consisting of 'KFold' and a positive integer value greater than 1. If you specify, e.g., 'KFold',  $k$ , then the software:

- 1 Randomly partitions the data into  $k$  sets
- 2 For each set, reserves the set as validation data, and trains the model using the other  $k - 1$  sets
- 3 Stores the  $k$  compact, trained models in the cells of a  $k$ -by-1 cell vector in the `Trained` property of the cross-validated model.

To create a cross-validated model, you can use one of these four options only: 'CVPartition', 'Holdout', or 'KFold'.

Example: 'KFold', 8

Data Types: single | double

### SGD and ASGD Convergence Controls

#### **BatchLimit** — Maximal number of batches

positive integer

Maximal number of batches to process, specified as the comma-separated pair consisting of 'BatchLimit' and a positive integer. When the software processes `BatchLimit` batches, it terminates optimization.

- By default:
  - The software passes through the data `PassLimit` times.
  - If you specify multiple solvers, and use (A)SGD to get an initial approximation for the next solver, then the default value is `ceil(1e6/BatchSize)`. `BatchSize` is the value of the 'BatchSize' name-value pair argument.
- If you specify 'BatchLimit' and 'PassLimit', then the software chooses the argument that results in processing the fewest observations.
- If you specify 'BatchLimit' but not 'PassLimit', then the software processes enough batches to complete up to one entire pass through the data.

Example: 'BatchLimit', 100

Data Types: single | double

#### **BetaTolerance** — Relative tolerance on linear coefficients and bias term

1e-4 (default) | nonnegative scalar

Relative tolerance on the linear coefficients and the bias term (intercept), specified as the comma-separated pair consisting of 'BetaTolerance' and a nonnegative scalar.

Let  $B_t = \begin{bmatrix} \beta_t' & b_t \end{bmatrix}$ , that is, the vector of the coefficients and the bias term at optimization

iteration  $t$ . If  $\left\| \frac{B_t - B_{t-1}}{B_t} \right\|_2 < \text{BetaTolerance}$ , then optimization terminates.

If the software converges for the last solver specified in `Solver`, then optimization terminates. Otherwise, the software uses the next solver specified in `Solver`.

Example: `'BetaTolerance', 1e-6`

Data Types: `single` | `double`

**NumCheckConvergence** — Number of batches to process before next convergence check  
positive integer

Number of batches to process before next convergence check, specified as the comma-separated pair consisting of `'NumCheckConvergence'` and a positive integer.

To specify the batch size, see `BatchSize`.

The software checks for convergence about 10 times per pass through the entire data set by default.

Example: `'NumCheckConvergence', 100`

Data Types: `single` | `double`

**PassLimit** — Maximal number of passes

1 (default) | positive integer

Maximal number of passes through the data, specified as the comma-separated pair consisting of `'PassLimit'` and a positive integer.

`fitcllinear` processes all observations when it completes one pass through the data.

When `fitcllinear` passes through the data `PassLimit` times, it terminates optimization.

If you specify `'BatchLimit'` and `'PassLimit'`, then `fitcllinear` chooses the argument that results in processing the fewest observations.

Example: `'PassLimit', 5`

Data Types: `single` | `double`

**ValidationData** — Validation data for optimization convergence detection

cell array

Data for optimization convergence detection, specified as the comma-separated pair consisting of `'ValidationData'` and a cell array.

During optimization, the software periodically estimates the loss of `ValidationData`. If the validation-data loss increases, then the software terminates optimization. For more details, see “Algorithms” on page 32-2044. To optimize hyperparameters using cross-validation, see cross-validation options such as `CrossVal`.

- `ValidationData{1}` must contain an  $m$ -by- $p$  or  $p$ -by- $m$  full or sparse matrix of predictor data that has the same orientation as  $X$ . The predictor variables in the training data  $X$  and `ValidationData{1}` must correspond. The number of observations in both sets can vary.
- `ValidationData{2}` and  $Y$  must be the same data type. The set of all distinct labels of `ValidationData{2}` must be a subset of all distinct labels of  $Y$ .
- Optionally, `ValidationData{3}` can contain an  $m$ -dimensional numeric vector of observation weights. The software normalizes the weights with the validation data so that they sum to 1.

If you specify `ValidationData`, then, to display validation loss at the command line, specify a value larger than 0 for `Verbose`.

If the software converges for the last solver specified in `Solver`, then optimization terminates. Otherwise, the software uses the next solver specified in `Solver`.

By default, the software does not detect convergence by monitoring validation-data loss.

### Dual SGD Convergence Controls

#### **BetaTolerance** — Relative tolerance on linear coefficients and bias term

1e-4 (default) | nonnegative scalar

Relative tolerance on the linear coefficients and the bias term (intercept), specified as the comma-separated pair consisting of 'BetaTolerance' and a nonnegative scalar.

Let  $B_t = \begin{bmatrix} \beta_t' & b_t \end{bmatrix}$ , that is, the vector of the coefficients and the bias term at optimization

iteration  $t$ . If  $\left\| \frac{B_t - B_{t-1}}{B_t} \right\|_2 < \text{BetaTolerance}$ , then optimization terminates.

If you also specify `DeltaGradientTolerance`, then optimization terminates when the software satisfies either stopping criterion.

If the software converges for the last solver specified in `Solver`, then optimization terminates. Otherwise, the software uses the next solver specified in `Solver`.

Example: `'BetaTolerance', 1e-6`

Data Types: `single` | `double`

### **DeltaGradientTolerance — Gradient-difference tolerance**

1 (default) | nonnegative scalar

Gradient-difference tolerance between upper and lower pool Karush-Kuhn-Tucker (KKT) complementarity conditions on page 32-2128 violators, specified as the comma-separated pair consisting of `'DeltaGradientTolerance'` and a nonnegative scalar.

- If the magnitude of the KKT violators is less than `DeltaGradientTolerance`, then the software terminates optimization.
- If the software converges for the last solver specified in `Solver`, then optimization terminates. Otherwise, the software uses the next solver specified in `Solver`.

Example: `'DeltaGapTolerance', 1e-2`

Data Types: `double` | `single`

### **NumCheckConvergence — Number of passes through entire data set to process before next convergence check**

5 (default) | positive integer

Number of passes through entire data set to process before next convergence check, specified as the comma-separated pair consisting of `'NumCheckConvergence'` and a positive integer.

Example: `'NumCheckConvergence', 100`

Data Types: `single` | `double`

### **PassLimit — Maximal number of passes**

10 (default) | positive integer

Maximal number of passes through the data, specified as the comma-separated pair consisting of `'PassLimit'` and a positive integer.

When the software completes one pass through the data, it has processed all observations.

When the software passes through the data `PassLimit` times, it terminates optimization.

Example: `'PassLimit', 5`

Data Types: `single` | `double`

### **ValidationData** — Validation data for optimization convergence detection

cell array

Data for optimization convergence detection, specified as the comma-separated pair consisting of `'ValidationData'` and a cell array.

During optimization, the software periodically estimates the loss of `ValidationData`. If the validation-data loss increases, then the software terminates optimization. For more details, see “Algorithms” on page 32-2044. To optimize hyperparameters using cross-validation, see cross-validation options such as `CrossVal`.

- `ValidationData{1}` must contain an  $m$ -by- $p$  or  $p$ -by- $m$  full or sparse matrix of predictor data that has the same orientation as `X`. The predictor variables in the training data `X` and `ValidationData{1}` must correspond. The number of observations in both sets can vary.
- `ValidationData{2}` and `Y` must be the same data type. The set of all distinct labels of `ValidationData{2}` must be a subset of all distinct labels of `Y`.
- Optionally, `ValidationData{3}` can contain an  $m$ -dimensional numeric vector of observation weights. The software normalizes the weights with the validation data so that they sum to 1.

If you specify `ValidationData`, then, to display validation loss at the command line, specify a value larger than 0 for `Verbose`.

If the software converges for the last solver specified in `Solver`, then optimization terminates. Otherwise, the software uses the next solver specified in `Solver`.

By default, the software does not detect convergence by monitoring validation-data loss.

### **BFGS, LBFGS, and SpARSA Convergence Controls**

#### **BetaTolerance** — Relative tolerance on linear coefficients and bias term

`1e-4` (default) | nonnegative scalar

Relative tolerance on the linear coefficients and the bias term (intercept), specified as the comma-separated pair consisting of 'BetaTolerance' and a nonnegative scalar.

Let  $B_t = \begin{bmatrix} \beta_t' & b_t \end{bmatrix}$ , that is, the vector of the coefficients and the bias term at optimization

iteration  $t$ . If  $\left\| \frac{B_t - B_{t-1}}{B_t} \right\|_2 < \text{BetaTolerance}$ , then optimization terminates.

If you also specify GradientTolerance, then optimization terminates when the software satisfies either stopping criterion.

If the software converges for the last solver specified in Solver, then optimization terminates. Otherwise, the software uses the next solver specified in Solver.

Example: 'BetaTolerance', 1e-6

Data Types: single | double

### **GradientTolerance — Absolute gradient tolerance**

1e-6 (default) | nonnegative scalar

Absolute gradient tolerance, specified as the comma-separated pair consisting of 'GradientTolerance' and a nonnegative scalar.

Let  $\nabla L_t$  be the gradient vector of the objective function with respect to the coefficients and bias term at optimization iteration  $t$ . If  $\|\nabla L_t\|_\infty = \max |\nabla L_t| < \text{GradientTolerance}$ , then optimization terminates.

If you also specify BetaTolerance, then optimization terminates when the software satisfies either stopping criterion.

If the software converges for the last solver specified in the software, then optimization terminates. Otherwise, the software uses the next solver specified in Solver.

Example: 'GradientTolerance', 1e-5

Data Types: single | double

### **HessianHistorySize — Size of history buffer for Hessian approximation**

15 (default) | positive integer



Size of history buffer for Hessian approximation, specified as the comma-separated pair consisting of 'HessianHistorySize' and a positive integer. That is, at each iteration, the software composes the Hessian using statistics from the latest HessianHistorySize iterations.

The software does not support 'HessianHistorySize' for SpaRSA.

Example: 'HessianHistorySize',10

Data Types: single | double

### **IterationLimit — Maximal number of optimization iterations**

1000 (default) | positive integer

Maximal number of optimization iterations, specified as the comma-separated pair consisting of 'IterationLimit' and a positive integer. IterationLimit applies to these values of Solver: 'bfgs', 'lbfgs', and 'sparsa'.

Example: 'IterationLimit',500

Data Types: single | double

### **ValidationData — Validation data for optimization convergence detection**

cell array

Data for optimization convergence detection, specified as the comma-separated pair consisting of 'ValidationData' and a cell array.

During optimization, the software periodically estimates the loss of ValidationData. If the validation-data loss increases, then the software terminates optimization. For more details, see “Algorithms” on page 32-2044. To optimize hyperparameters using cross-validation, see cross-validation options such as CrossVal.

- ValidationData{1} must contain an  $m$ -by- $p$  or  $p$ -by- $m$  full or sparse matrix of predictor data that has the same orientation as  $X$ . The predictor variables in the training data  $X$  and ValidationData{1} must correspond. The number of observations in both sets can vary.
- ValidationData{2} and  $Y$  must be the same data type. The set of all distinct labels of ValidationData{2} must be a subset of all distinct labels of  $Y$ .
- Optionally, ValidationData{3} can contain an  $m$ -dimensional numeric vector of observation weights. The software normalizes the weights with the validation data so that they sum to 1.

If you specify `ValidationData`, then, to display validation loss at the command line, specify a value larger than 0 for `Verbose`.

If the software converges for the last solver specified in `Solver`, then optimization terminates. Otherwise, the software uses the next solver specified in `Solver`.

By default, the software does not detect convergence by monitoring validation-data loss.

### Hyperparameter Optimization

#### **OptimizeHyperparameters** — Parameters to optimize

'none' (default) | 'auto' | 'all' | cell array of eligible parameter names | vector of `optimizableVariable` objects

Parameters to optimize, specified as:

- 'none' — Do not optimize.
- 'auto' — Use {'Lambda', 'Learner'}
- 'all' — Optimize all eligible parameters.
- Cell array of eligible parameter names
- Vector of `optimizableVariable` objects, typically the output of hyperparameters

The optimization attempts to minimize the cross-validation loss (error) for `fitclinear` by varying the parameters. For information about cross-validation loss (albeit in a different context), see “Classification Loss” on page 32-3250. To control the cross-validation type and other aspects of the optimization, use the `HyperparameterOptimizationOptions` name-value pair.

---

**Note** `OptimizeHyperparameters` values override any values you set using other name-value pairs. For example, setting `OptimizeHyperparameters` to 'auto' causes the 'auto' values to apply.

---

The eligible parameters for `fitclinear` are:

- `Lambda` — `fitclinear` searches among positive values, by default log-scaled in the range `[1e-5/NumObservations, 1e5/NumObservations]`.
- `Learner` — `fitclinear` searches among 'svm' and 'logistic'.

- Regularization — fitclinear searches among 'ridge' and 'lasso'.

Set nondefault parameters by passing a vector of `optimizableVariable` objects that have nondefault values. For example,

```
load fisheriris
params = hyperparameters('fitclinear', meas, species);
params(1).Range = [1e-4, 1e6];
```

Pass `params` as the value of `OptimizeHyperparameters`.

By default, iterative display appears at the command line, and plots appear according to the number of hyperparameters in the optimization. For the optimization and plots, the objective function is  $\log(1 + \text{cross-validation loss})$  for regression, and the misclassification rate for classification. To control the iterative display, set the `HyperparameterOptimizationOptions` name-value pair, `Verbose` field. To control the plots, set the `HyperparameterOptimizationOptions` name-value pair, `ShowPlots` field.

For an example, see “Optimize Linear Classifier” on page 32-2008.

Example: 'auto'

Data Types: char | cell

### **HyperparameterOptimizationOptions** — Options for optimization structure

Options for optimization, specified as a structure. Modifies the effect of the `OptimizeHyperparameters` name-value pair. All fields in the structure are optional.

Field Name	Values	Default
Optimizer	<ul style="list-style-type: none"> <li>'bayesopt' — Use Bayesian optimization. Internally, this setting calls bayesopt.</li> <li>'gridsearch' — Use grid search with NumGridDivisions values per dimension.</li> <li>'randomsearch' — Search at random among MaxObjectiveEvaluations points.</li> </ul> <p>'gridsearch' searches in a random order, using uniform sampling without replacement from the grid. After optimization, you can get a table in grid order by using the command sortrows (Mdl.ParameterOptimizationResults).</p>	'bayesopt'
AcquisitionFunctionName	<ul style="list-style-type: none"> <li>'expected-improvement-per-second-plus'</li> <li>'expected-improvement'</li> <li>'expected-improvement-plus'</li> <li>'expected-improvement-per-second'</li> <li>'lower-confidence-bound'</li> <li>'probability-of-improvement'</li> </ul> <p>For details, see the bayesopt AcquisitionFunctionName name-value pair, or “Acquisition Function Types” on page 10-4.</p>	'expected-improvement-per-second-plus'
MaxObjectiveEvaluations	Maximum number of objective function evaluations.	30 for 'bayesopt' or 'randomsearch', and the entire grid for 'gridsearch'

Field Name	Values	Default
MaxTime	Time limit, specified as a positive real. The time limit is in seconds, as measured by <code>tic</code> and <code>toc</code> . Run time can exceed <code>MaxTime</code> because <code>MaxTime</code> does not interrupt function evaluations.	Inf
NumGridDivisions	For 'gridsearch', the number of values in each dimension. Can be a vector of positive integers giving the number of values for each dimension, or a scalar that applies to all dimensions. Ignored for categorical variables.	10
ShowPlots	Logical value indicating whether to show plots. If <code>true</code> , plots the best objective function value against iteration number. If there are one or two optimization parameters, and if <code>Optimizer</code> is 'bayesopt', then <code>ShowPlots</code> also plots a model of the objective function against the parameters.	true
SaveIntermediateResults	Logical value indicating whether to save results when <code>Optimizer</code> is 'bayesopt'. If <code>true</code> , overwrites a workspace variable named 'BayesoptResults' at each iteration. The variable is a <code>BayesianOptimization</code> object.	false
Verbose	Display to the command line. <ul style="list-style-type: none"> <li>• 0 — No iterative display</li> <li>• 1 — Iterative display</li> <li>• 2 — Iterative display with extra information</li> </ul> For details, see the <code>bayesopt</code> <code>Verbose</code> name-value pair.	1

Field Name	Values	Default
UseParallel	Logical value indicating whether to run Bayesian optimization in parallel, which requires Parallel Computing Toolbox. For details, see “Parallel Bayesian Optimization” on page 10-9.	false
Repartition	Logical value indicating whether to repartition the cross-validation at every iteration. If false, the optimizer uses a single partition for the optimization.  true usually gives the most robust results because this setting takes partitioning noise into account. However, for good results, true requires at least twice as many function evaluations.	false
Use no more than one of the following three field names.		
CVPartition	A cvpartition object, as created by cvpartition	Kfold = 5
Holdout	A scalar in the range (0, 1) representing the holdout fraction.	
Kfold	An integer greater than 1.	

Example: `struct('MaxObjectiveEvaluations', 60)`

Data Types: `struct`

## Output Arguments

### **mdl** — Trained linear classification model

ClassificationLinear model object | ClassificationPartitionedLinear cross-validated model object

Trained linear classification model, returned as a ClassificationLinear model object or ClassificationPartitionedLinear cross-validated model object.

If you set any of the name-value pair arguments `KFold`, `Holdout`, `CrossVal`, or `CVPartition`, then `Mdl` is a `ClassificationPartitionedLinear` cross-validated model object. Otherwise, `Mdl` is a `ClassificationLinear` model object.

To reference properties of `Mdl`, use dot notation. For example, enter `Mdl.Beta` in the Command Window to display the vector or matrix of estimated coefficients.

---

**Note** Unlike other classification models, and for economical memory usage, `ClassificationLinear` and `ClassificationPartitionedLinear` model objects do not store the training data or training process details (for example, convergence history).

---

### **FitInfo** — Optimization details

structure array

Optimization details, returned as a structure array.

Fields specify final values or name-value pair argument specifications, for example, `Objective` is the value of the objective function when optimization terminates. Rows of multidimensional fields correspond to values of `Lambda` and columns correspond to values of `Solver`.

This table describes some notable fields.

Field	Description
<code>TerminationStatus</code>	<ul style="list-style-type: none"> <li>Reason for optimization termination</li> <li>Corresponds to a value in <code>TerminationCode</code></li> </ul>
<code>FitTime</code>	Elapsed, wall-clock time in seconds

Field	Description	
History	A structure array of optimization information for each iteration. The field <code>Solver</code> stores solver types using integer coding.	
	Integer	Solver
	1	SGD
	2	ASGD
	3	Dual SGD for SVM
	4	LBFGS
	5	BFGS
6	SpaRSA	

To access fields, use dot notation. For example, to access the vector of objective function values for each iteration, enter `FitInfo.History.Objective`.

It is good practice to examine `FitInfo` to assess whether convergence is satisfactory.

### **HyperparameterOptimizationResults — Description of the cross-validation optimization of hyperparameters**

`BayesianOptimization` object | table of hyperparameters and associated values

Description of the cross-validation optimization of hyperparameters, stored as a `BayesianOptimization` object or a table of hyperparameters and associated values. Nonempty when the `OptimizeHyperparameters` name-value pair is nonempty at creation. Value depends on the setting of the `HyperparameterOptimizationOptions` name-value pair at creation:

- 'bayesopt' (default) — Object of class `BayesianOptimization`
- 'gridsearch' or 'randomsearch' — Table of hyperparameters used, observed objective function values (cross-validation loss), and rank of observations from lowest (best) to highest (worst)

Data Types: `table`



## Definitions

### Warm Start

A warm start is initial estimates of the beta coefficients and bias term supplied to an optimization routine for quicker convergence.

### Tips

- It is a best practice to orient your predictor matrix so that observations correspond to columns and to specify 'ObservationsIn', 'columns'. As a result, you can experience a significant reduction in optimization-execution time.
- For better optimization accuracy if X is high-dimensional and Regularization is 'ridge', set any of these combinations for Solver:
  - 'sgd'
  - 'asgd'
  - 'dual' if Learner is 'svm'
  - {'sgd', 'lbfgs'}
  - {'asgd', 'lbfgs'}
  - {'dual', 'lbfgs'} if Learner is 'svm'

Other combinations can result in poor optimization accuracy.

- For better optimization accuracy if X is moderate- through low-dimensional and Regularization is 'ridge', set Solver to 'bfgs'.
- If Regularization is 'lasso', set any of these combinations for Solver:
  - 'sgd'
  - 'asgd'
  - 'sparsa'
  - {'sgd', 'sparsa'}
  - {'asgd', 'sparsa'}
- When choosing between SGD and ASGD, consider that:

- SGD takes less time per iteration, but requires more iterations to converge.
- ASGD requires fewer iterations to converge, but takes more time per iteration.
- If `X` has few observations, but many predictor variables, then:
  - Specify `'PostFitBias', true`.
  - For SGD or ASGD solvers, set `PassLimit` to a positive integer that is greater than 1, for example, 5 or 10. This setting often results in better accuracy.
- For SGD and ASGD solvers, `BatchSize` affects the rate of convergence.
  - If `BatchSize` is too small, then `fitclinear` achieves the minimum in many iterations, but computes the gradient per iteration quickly.
  - If `BatchSize` is too large, then `fitclinear` achieves the minimum in fewer iterations, but computes the gradient per iteration slowly.
- Large learning rates (see `LearnRate`) speed up convergence to the minimum, but can lead to divergence (that is, over-stepping the minimum). Small learning rates ensure convergence to the minimum, but can lead to slow termination.
- When using lasso penalties, experiment with various values of `TruncationPeriod`. For example, set `TruncationPeriod` to 1, 10, and then 100.
- For efficiency, `fitclinear` does not standardize predictor data. To standardize `X`, enter

```
X = bsxfun(@rdivide, bsxfun(@minus, X, mean(X, 2)), std(X, 0, 2));
```

The code requires that you orient the predictors and observations as the rows and columns of `X`, respectively. Also, for memory-usage economy, the code replaces the original predictor data the standardized data.

- After training a `ClassificationLinear` model object by using `fitclinear`, use the function `predict` and the trained model object to generate C code that predicts labels for new data. For details, see “Code Generation” on page 32-0 .

## Algorithms

- If you specify `ValidationData`, then, during objective-function optimization:
  - `fitclinear` estimates the validation loss of `ValidationData` periodically using the current model, and tracks the minimal estimate.

- When `fitlinear` estimates a validation loss, it compares the estimate to the minimal estimate.
- When subsequent, validation loss estimates exceed the minimal estimate five times, `fitlinear` terminates optimization.
- If you specify `ValidationData` and to implement a cross-validation routine (`CrossVal`, `CVPartition`, `Holdout`, or `KFold`), then:
  - 1 `fitlinear` randomly partitions  $X$  and  $Y$  according to the cross-validation routine that you choose.
  - 2 `fitlinear` trains the model using the training-data partition. During objective-function optimization, `fitlinear` uses `ValidationData` as another possible way to terminate optimization (for details, see the previous bullet).
  - 3 Once `fitlinear` satisfies a stopping criterion, it constructs a trained model based on the optimized linear coefficients and intercept.
    - a If you implement  $k$ -fold cross-validation, and `fitlinear` has not exhausted all training-set folds, then `fitlinear` returns to Step 2 to train using the next training-set fold.
    - b Otherwise, `fitlinear` terminates training, and then returns the cross-validated model.
  - 4 You can determine the quality of the cross-validated model. For example:
    - To determine the validation loss using the holdout or out-of-fold data from step 1, pass the cross-validated model to `kfoldLoss`.
    - To predict observations on the holdout or out-of-fold data from step 1, pass the cross-validated model to `kfoldPredict`.

## References

- [1] Hsieh, C. J., K. W. Chang, C. J. Lin, S. S. Keerthi, and S. Sundararajan. “A Dual Coordinate Descent Method for Large-Scale Linear SVM.” *Proceedings of the 25th International Conference on Machine Learning, ICML '08*, 2001, pp. 408–415.
- [2] Langford, J., L. Li, and T. Zhang. “Sparse Online Learning Via Truncated Gradient.” *J. Mach. Learn. Res.*, Vol. 10, 2009, pp. 777–801.
- [3] Nocedal, J. and S. J. Wright. *Numerical Optimization*, 2nd ed., New York: Springer, 2006.

- [4] Shalev-Shwartz, S., Y. Singer, and N. Srebro. “Pegasos: Primal Estimated Sub-Gradient Solver for SVM.” *Proceedings of the 24th International Conference on Machine Learning, ICML '07*, 2007, pp. 807–814.
- [5] Wright, S. J., R. D. Nowak, and M. A. T. Figueiredo. “Sparse Reconstruction by Separable Approximation.” *Trans. Sig. Proc.*, Vol. 57, No 7, 2009, pp. 2479–2493.
- [6] Xiao, Lin. “Dual Averaging Methods for Regularized Stochastic Learning and Online Optimization.” *J. Mach. Learn. Res.*, Vol. 11, 2010, pp. 2543–2596.
- [7] Xu, Wei. “Towards Optimal One Pass Large Scale Learning with Averaged Stochastic Gradient Descent.” *CoRR*, abs/1107.2490, 2011.

## Extended Capabilities

### Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function supports tall arrays for out-of-memory data with the limitations:

- Supported syntaxes for tall arrays  $X$  and  $Y$  are:
  - `obj = fitclinear(X,Y)`
  - `obj = fitclinear(___,Name,Value)`
- Some name-value pairs have different defaults compared to the in-memory `fitclinear` function. Supported name-value pairs, and any differences, are:
  - `'ObservationsIn'` — Supports only `'rows'`.
  - `'Lambda'` — Can be `'auto'` (default) or a scalar.
  - `'Learner'`
  - `'Regularization'` — Supports only `'ridge'`.
  - `'Solver'` — Supports only `'lbfgs'`.
  - `'FitBias'` — Supports only `true`.
  - `'Verbose'` — Default value is 1.

- 'Beta'
  - 'Bias'
  - 'ClassNames'
  - 'Cost'
  - 'Prior'
  - 'Weights' — Value must be a tall array.
  - 'HessianHistorySize'
  - 'BetaTolerance' — Default value is relaxed to  $1e-3$ .
  - 'GradientTolerance' — Default value is relaxed to  $1e-3$ .
  - 'IterationLimit' — Default value is relaxed to 20.
- For tall arrays, `fitclinear` implements LBFGS by distributing the calculation of the loss and gradient among different parts of the tall array at each iteration. Other solvers are not available for tall arrays.

When initial values for `Beta` and `Bias` are not given, `fitclinear` refines the initial estimates of the parameters by fitting the model locally to parts of the data and combining the coefficients by averaging.

For more information, see “Tall Arrays” (MATLAB).

## See Also

`ClassificationLinear` | `ClassificationPartitionedLinear` | `fitcecoc` | `fitckernel` | `fitcsvm` | `fitglm` | `fitrlinear` | `kfoldLoss` | `kfoldPredict` | `lassoglm` | `predict` | `templateLinear` | `testcholdout`

**Introduced in R2016a**

## fitcnb

Train multiclass naive Bayes model

### Syntax

```
Mdl = fitcnb(Tbl,ResponseVarName)
```

```
Mdl = fitcnb(Tbl,formula)
```

```
Mdl = fitcnb(Tbl,Y)
```

```
Mdl = fitcnb(X,Y)
```

```
Mdl = fitcnb( ____,Name,Value)
```

### Description

`Mdl = fitcnb(Tbl,ResponseVarName)` returns a multiclass naive Bayes model (`Mdl`), trained by the predictors in table `Tbl` and class labels in the variable `Tbl.ResponseVarName`.

`Mdl = fitcnb(Tbl,formula)` returns a multiclass naive Bayes model (`Mdl`), trained by the predictors in table `Tbl`. `formula` is an explanatory model of the response and a subset of predictor variables in `Tbl` used to fit `Mdl`.

`Mdl = fitcnb(Tbl,Y)` returns a multiclass naive Bayes model (`Mdl`), trained by the predictors in the table `Tbl` and class labels in the array `Y`.

`Mdl = fitcnb(X,Y)` returns a multiclass naive Bayes model (`Mdl`), trained by predictors `X` and class labels `Y`.

`Mdl = fitcnb( ____,Name,Value)` returns a naive Bayes classifier with additional options specified by one or more `Name,Value` pair arguments, using any of the previous syntaxes. For example, you can specify a distribution to model the data, prior probabilities for the classes, or the kernel smoothing window bandwidth.

## Examples

### Train a Naive Bayes Classifier

Load Fisher's iris data set.

```
load fisheriris
X = meas(:,3:4);
Y = species;
tabulate(Y)
```

Value	Count	Percent
setosa	50	33.33%
versicolor	50	33.33%
virginica	50	33.33%

The software can classify data with more than two classes using naive Bayes methods.

Train a naive Bayes classifier. It is good practice to specify the class order.

```
Mdl = fitcnb(X,Y,...
    'ClassNames',{'setosa','versicolor','virginica'})
```

```
Mdl =
```

```
ClassificationNaiveBayes
    ResponseName: 'Y'
  CategoricalPredictors: []
        ClassNames: {'setosa' 'versicolor' 'virginica'}
        ScoreTransform: 'none'
    NumObservations: 150
    DistributionNames: {'normal' 'normal'}
  DistributionParameters: {3x2 cell}
```

Mdl is a trained ClassificationNaiveBayes classifier.

By default, the software models the predictor distribution within each class using a Gaussian distribution having some mean and standard deviation. Use dot notation to display the parameters of a particular Gaussian fit, e.g., display the fit for the first feature within *setosa*.

```
setosaIndex = strcmp(Mdl.ClassNames, 'setosa');
estimates = Mdl.DistributionParameters{setosaIndex,1}

estimates =

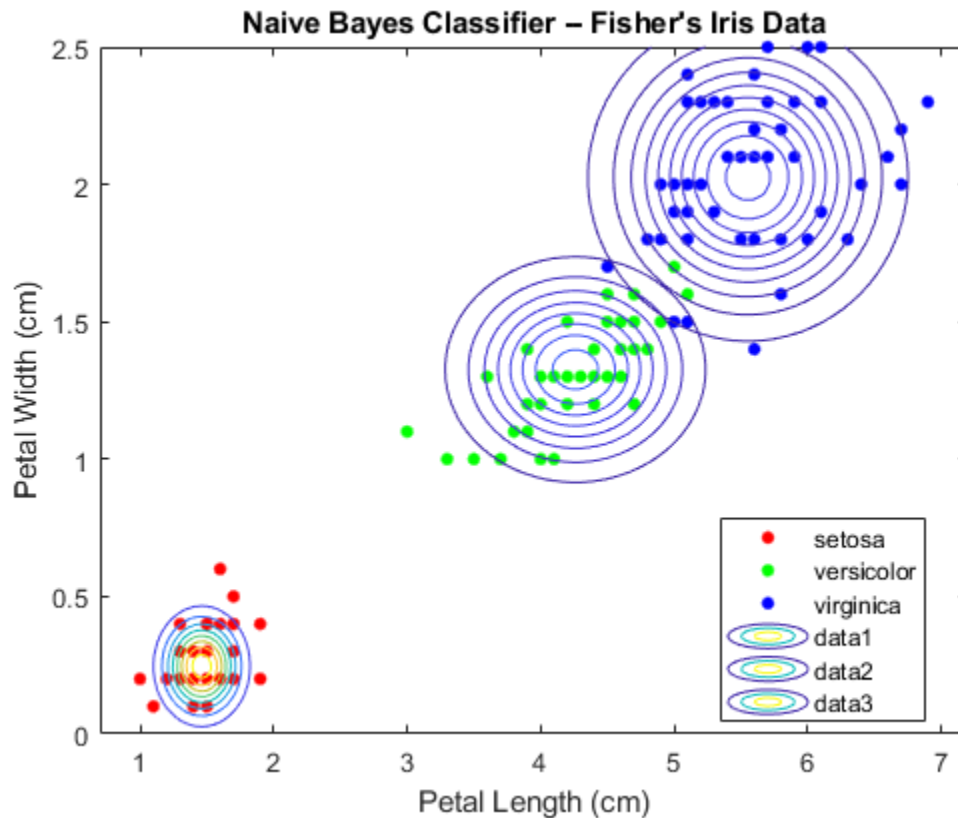
    1.4620
    0.1737
```

The mean is 1.4620 and the standard deviation is 0.1737.

Plot the Gaussian contours.

```
figure
gscatter(X(:,1),X(:,2),Y);
h = gca;
cxlim = h.XLim;
cylim = h.YLim;
hold on
Params = cell2mat(Mdl.DistributionParameters);
Mu = Params(2*(1:3)-1,1:2); % Extract the means
Sigma = zeros(2,2,3);
for j = 1:3
    Sigma(:,:,j) = diag(Params(2*j,:)).^2; % Create diagonal covariance matrix
    xlim = Mu(j,1) + 4*[1 -1]*sqrt(Sigma(1,1,j));
    ylim = Mu(j,2) + 4*[1 -1]*sqrt(Sigma(2,2,j));
    ezcontour(@(x1,x2)mvnpdf([x1,x2],Mu(j,:),Sigma(:,:,j)),[xlim ylim])
    % Draw contours for the multivariate normal distributions
end
h.XLim = cxlim;
h.YLim = cylim;
title('Naive Bayes Classifier -- Fisher''s Iris Data')
xlabel('Petal Length (cm)')
ylabel('Petal Width (cm)')
hold off
```





You can change the default distribution using the name-value pair argument `'DistributionNames'`. For example, if some predictors are categorical, then you can specify that they are multivariate, multinomial random variables using `'DistributionNames','mvmn'`.

### Specify Prior Probabilities When Training Naive Bayes Classifiers

Construct a naive Bayes classifier for Fisher's iris data set. Also, specify prior probabilities during training.

Load Fisher's iris data set.

```
load fisheriris
X = meas;
Y = species;
classNames = {'setosa','versicolor','virginica'}; % Class order
```

X is a numeric matrix that contains four petal measurements for 150 irises. Y is a cell array of character vectors that contains the corresponding iris species.

By default, the prior class probability distribution is the relative frequency distribution of the classes in the data set, which in this case is 33% for each species. However, suppose you know that in the population 50% of the irises are setosa, 20% are versicolor, and 30% are virginica. You can incorporate this information by specifying this distribution as a prior probability during training.

Train a naive Bayes classifier. Specify the class order and prior class probability distribution.

```
prior = [0.5 0.2 0.3];
Mdl = fitcnb(X,Y,'ClassNames',classNames,'Prior',prior)
```

```
Mdl =
```

```
ClassificationNaiveBayes
    ResponseName: 'Y'
    CategoricalPredictors: []
           ClassNames: {'setosa' 'versicolor' 'virginica'}
    ScoreTransform: 'none'
    NumObservations: 150
    DistributionNames: {'normal' 'normal' 'normal' 'normal'}
    DistributionParameters: {3x4 cell}
```

Mdl is a trained `ClassificationNaiveBayes` classifier, and some of its properties appear in the Command Window. The software treats the predictors as independent given a class, and, by default, fits them using normal distributions.

The naive Bayes algorithm does not use the prior class probabilities during training. Therefore, you can specify prior class probabilities after training using dot notation. For example, suppose that you want to see the difference in performance between a model that uses the default prior class probabilities and a model that uses `prior`.

Create a new naive Bayes model based on `Mdl`, and specify that the prior class probability distribution is an empirical class distribution.

```
defaultPriorMdl = Mdl;
FreqDist = cell2table(tabulate(Y));
defaultPriorMdl.Prior = FreqDist(:,3);
```

The software normalizes the prior class probabilities to sum to 1.

Estimate the cross-validation error for both models using 10-fold cross validation.

```
rng(1); % For reproducibility
defaultCVMdl = crossval(defaultPriorMdl);
defaultLoss = kfoldLoss(defaultCVMdl)
CVMdl = crossval(Mdl);
Loss = kfoldLoss(CVMdl)
```

```
defaultLoss =
```

```
    0.0533
```

```
Loss =
```

```
    0.0340
```

`Mdl` performs better than `defaultPriorMdl`.

## Specify Predictor Distributions for Naive Bayes Classifiers

Load Fisher's iris data set.

```
load fisheriris
X = meas;
Y = species;
```

Train a naive Bayes classifier using every predictor. It is good practice to specify the class order.

```
Mdl1 = fitcnb(X,Y,...
    'ClassNames',{'setosa','versicolor','virginica'})
```

```
Mdl1.DistributionParameters
Mdl1.DistributionParameters{1,2}

Mdl1 =

ClassificationNaiveBayes
    ResponseName: 'Y'
    CategoricalPredictors: []
        ClassNames: {'setosa' 'versicolor' 'virginica'}
        ScoreTransform: 'none'
        NumObservations: 150
        DistributionNames: {'normal' 'normal' 'normal' 'normal'}
        DistributionParameters: {3x4 cell}

ans =

3x4 cell array

    {2x1 double}    {2x1 double}    {2x1 double}    {2x1 double}
    {2x1 double}    {2x1 double}    {2x1 double}    {2x1 double}
    {2x1 double}    {2x1 double}    {2x1 double}    {2x1 double}

ans =

    3.4280
    0.3791
```

By default, the software models the predictor distribution within each class as a Gaussian with some mean and standard deviation. There are four predictors and three class levels. Each cell in `Mdl1.DistributionParameters` corresponds to a numeric vector containing the mean and standard deviation of each distribution, e.g., the mean and standard deviation for setosa iris sepal widths are 3.4280 and 0.3791, respectively.

Estimate the confusion matrix for `Mdl1`.

```
isLabels1 = resubPredict(Mdl1);
ConfusionMat1 = confusionmat(Y,isLabels1)

ConfusionMat1 =
```

```

50    0    0
 0   47    3
 0    3   47

```

Element  $(j, k)$  of `ConfusionMat1` represents the number of observations that the software classifies as  $k$ , but are truly in class  $j$  according to the data.

Retrain the classifier using the Gaussian distribution for predictors 1 and 2 (the sepal lengths and widths), and the default normal kernel density for predictors 3 and 4 (the petal lengths and widths).

```

Mdl2 = fitcnb(X,Y,...
    'Distribution',{ 'normal','normal','kernel','kernel'},...
    'ClassNames',{ 'setosa','versicolor','virginica'});
Mdl2.DistributionParameters{1,2}

```

```

ans =

    3.4280
    0.3791

```

The software does not train parameters to the kernel density. Rather, the software chooses an optimal width. However, you can specify a width using the `'Width'` name-value pair argument.

Estimate the confusion matrix for `Mdl2`.

```

isLabels2 = resubPredict(Mdl2);
ConfusionMat2 = confusionmat(Y,isLabels2)

```

```

ConfusionMat2 =

    50    0    0
     0   47    3
     0    3   47

```

Based on the confusion matrices, the two classifiers perform similarly in the training sample.

### Compare Classifiers Using Cross Validation

Load Fisher's iris data set.

```
load fisheriris
X = meas;
Y = species;
rng(1); % For reproducibility
```

Train and cross validate a naive Bayes classifier using the default options and  $k$ -fold cross validation. It is good practice to specify the class order.

```
CVMd11 = fitcnb(X,Y,...
    'ClassNames',{'setosa','versicolor','virginica'},...
    'CrossVal','on');
```

By default, the software models the predictor distribution within each class as a Gaussian with some mean and standard deviation. CVMd11 is a `ClassificationPartitionedModel` model.

Create a default naive Bayes binary classifier template, and train an error-correcting, output codes multiclass model.

```
t = templateNaiveBayes();
CVMd12 = fitcecoc(X,Y,'CrossVal','on','Learners',t);
```

CVMd12 is a `ClassificationPartitionedECOC` model. You can specify options for the naive Bayes binary learners using the same name-value pair arguments as for `fitcnb`.

Compare the out-of-sample  $k$ -fold classification error (proportion of misclassified observations).

```
classErr1 = kfoldLoss(CVMd11,'LossFun','ClassifErr')
classErr2 = kfoldLoss(CVMd12,'LossFun','ClassifErr')

classErr1 =

    0.0533
```

```
classErr2 =
    0.0467
```

Mdl2 has a lower generalization error.

## Train Naive Bayes Classifiers Using Multinomial Predictors

Some spam filters classify an incoming email as spam based on how many times a word or punctuation (called tokens) occurs in an email. The predictors are the frequencies of particular words or punctuations in an email. Therefore, the predictors compose multinomial random variables.

This example illustrates classification using naive Bayes and multinomial predictors.

### Create Training Data

Suppose you observed 1000 emails and classified them as spam or not spam. Do this by randomly assigning -1 or 1 to  $y$  for each email.

```
n = 1000; % Sample size
rng(1); % For reproducibility
Y = randsample([-1 1],n,true); % Random labels
```

To build the predictor data, suppose that there are five tokens in the vocabulary, and 20 observed tokens per email. Generate predictor data from the five tokens by drawing random, multinomial deviates. The relative frequencies for tokens corresponding to spam emails should differ from emails that are not spam.

```
tokenProbs = [0.2 0.3 0.1 0.15 0.25;...
    0.4 0.1 0.3 0.05 0.15]; % Token relative frequencies
tokensPerEmail = 20; % Fixed for convenience
X = zeros(n,5);
X(Y == 1,:) = mnrnd(tokensPerEmail,tokenProbs(1,:),sum(Y == 1));
X(Y == -1,:) = mnrnd(tokensPerEmail,tokenProbs(2,:),sum(Y == -1));
```

### Train the Classifier

Train a naive Bayes classifier. Specify that the predictors are multinomial.

```
Mdl = fitcnb(X,Y,'Distribution','mn');
```

Mdl is a trained `ClassificationNaiveBayes` classifier.

Assess the in-sample performance of Mdl by estimating the misclassification error.

```
isGenRate = resubLoss(Mdl,'LossFun','ClassifErr')
```

```
isGenRate =
```

```
0.0200
```

The in-sample misclassification rate is 2%.

### Create New Data

Randomly generate deviates that represent a new batch of emails.

```
newN = 500;  
newY = randsample([-1 1],newN,true);  
newX = zeros(newN,5);  
newX(newY == 1,:) = mnrnd(tokensPerEmail,tokenProbs(1,:),...  
    sum(newY == 1));  
newX(newY == -1,:) = mnrnd(tokensPerEmail,tokenProbs(2,:),...  
    sum(newY == -1));
```

### Assess Classifier Performance

Classify the new emails using the trained naive Bayes classifier Mdl, and determine whether the algorithm generalizes.

```
oosGenRate = loss(Mdl,newX,newY)
```

```
oosGenRate =
```

```
0.0261
```

The out-of-sample misclassification rate is 2.6% indicating that the classifier generalizes fairly well.



## Optimize Naive Bayes Classifier

This example shows how to use the `OptimizeHyperparameters` name-value pair to minimize cross-validation loss in a naive Bayes classifier using `fitcnb`. The example uses Fisher's iris data.

Load Fisher's iris data.

```
load fisheriris
X = meas;
Y = species;
classNames = {'setosa', 'versicolor', 'virginica'};
```

Optimize the classification using the 'auto' parameters.

For reproducibility, set the random seed and use the 'expected-improvement-plus' acquisition function.

```
rng default
Mdl = fitcnb(X,Y, 'ClassNames', classNames, 'OptimizeHyperparameters', 'auto', ...
    'HyperparameterOptimizationOptions', struct('AcquisitionFunctionName', ...
    'expected-improvement-plus'))
```

Warning: It is recommended that you first standardize all numeric predictors when optimizing the Naive Bayes 'Width' parameter. Ignore this warning if you have done that.

Iter	Eval result	Objective	Objective runtime	BestSoFar (observed)	BestSoFar (estim.)	Distribution Names
1	Best	0.053333	0.85542	0.053333	0.053333	normal
2	Best	0.046667	1.164	0.046667	0.049998	kernel
3	Accept	0.053333	0.31605	0.046667	0.046667	normal
4	Accept	0.086667	0.61229	0.046667	0.046668	kernel
5	Accept	0.046667	0.71963	0.046667	0.046663	kernel
6	Accept	0.073333	0.75656	0.046667	0.046665	kernel
7	Accept	0.046667	0.64887	0.046667	0.046655	kernel
8	Accept	0.046667	0.67	0.046667	0.046647	kernel
9	Accept	0.06	0.62215	0.046667	0.046658	kernel
10	Accept	0.046667	0.62609	0.046667	0.046618	kernel
11	Accept	0.046667	0.61731	0.046667	0.046619	kernel
12	Accept	0.046667	0.45606	0.046667	0.046612	kernel
13	Accept	0.046667	0.40349	0.046667	0.046603	kernel
14	Accept	0.046667	0.50068	0.046667	0.046607	kernel

	15		Accept		0.046667		0.53732		0.046667		0.046606		kernel
	16		Accept		0.046667		0.49949		0.046667		0.046606		kernel
	17		Accept		0.046667		0.53665		0.046667		0.046606		kernel
	18		Accept		0.046667		0.49234		0.046667		0.046626		kernel
	19		Accept		0.046667		0.46814		0.046667		0.046626		kernel
	20		Accept		0.046667		0.6009		0.046667		0.046627		kernel
=====													
	Iter		Eval		Objective		Objective		BestSoFar		BestSoFar		Distribution-
			result				runtime		(observed)		(estim.)		Names
=====													
	21		Accept		0.046667		0.53038		0.046667		0.046627		kernel
	22		Accept		0.093333		0.48279		0.046667		0.046618		kernel
	23		Accept		0.046667		0.58435		0.046667		0.046619		kernel
	24		Accept		0.046667		0.60253		0.046667		0.04663		kernel
	25		Accept		0.046667		0.45703		0.046667		0.04663		kernel
	26		Accept		0.046667		0.42059		0.046667		0.046631		kernel
	27		Accept		0.046667		0.49448		0.046667		0.046631		kernel
	28		Accept		0.046667		0.43556		0.046667		0.046631		kernel
	29		Accept		0.046667		0.46054		0.046667		0.046631		kernel
	30		Accept		0.08		0.45189		0.046667		0.046628		kernel

---

Optimization completed.  
 MaxObjectiveEvaluations of 30 reached.  
 Total function evaluations: 30  
 Total elapsed time: 77.0159 seconds.  
 Total objective function evaluation time: 17.0236

Best observed feasible point:

DistributionNames	Width
kernel	0.11903

Observed objective function value = 0.046667  
 Estimated objective function value = 0.046628  
 Function evaluation time = 1.164

Best estimated feasible point (according to models):

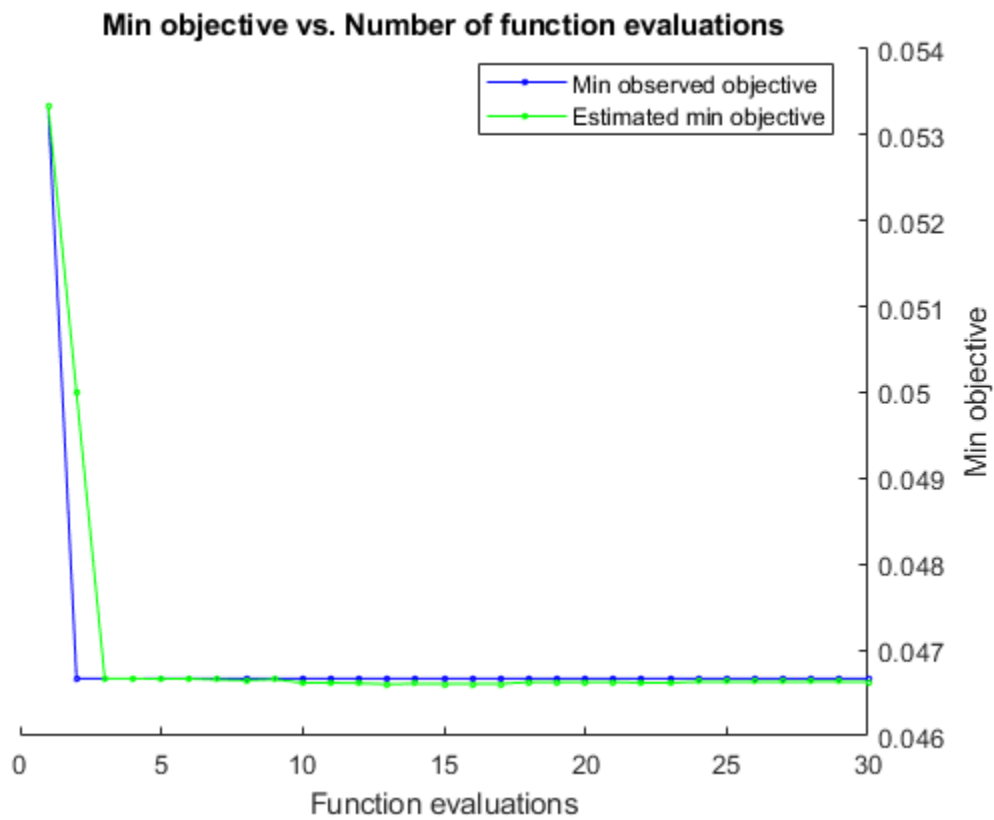
DistributionNames	Width
kernel	0.25613

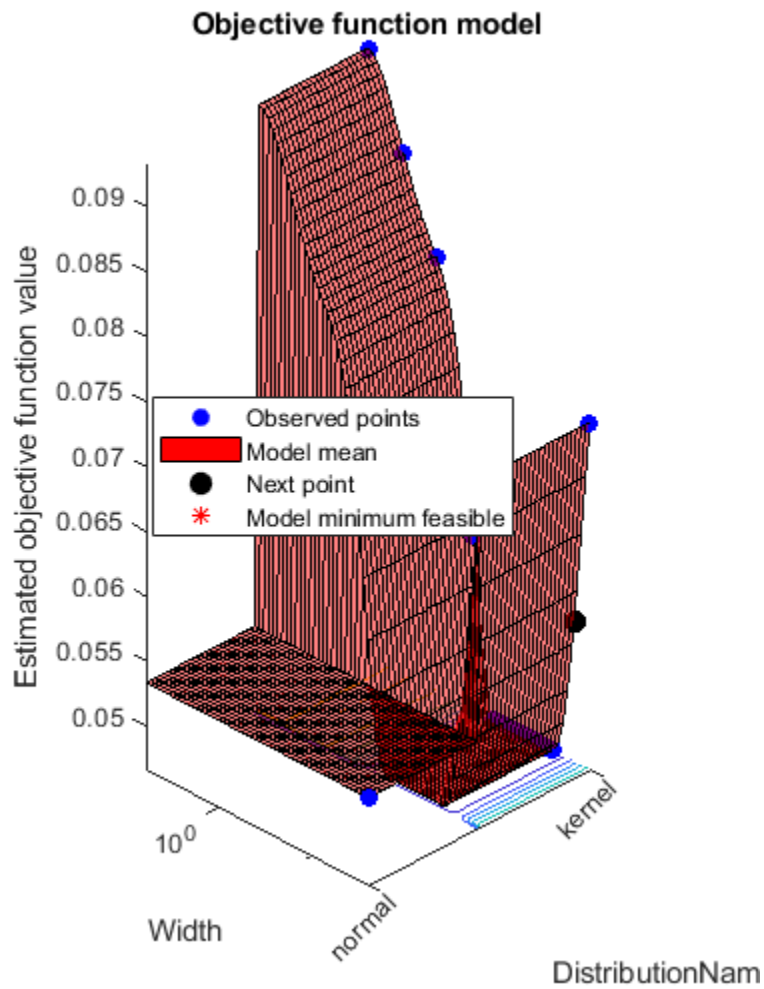
---

Estimated objective function value = 0.046628  
Estimated function evaluation time = 0.54932

Mdl =

```
ClassificationNaiveBayes
    ResponseName: 'Y'
    CategoricalPredictors: []
    ClassNames: {'setosa' 'versicolor' 'virginica'}
    ScoreTransform: 'none'
    NumObservations: 150
    HyperparameterOptimizationResults: [1x1 BayesianOptimization]
    DistributionNames: {1x4 cell}
    DistributionParameters: {3x4 cell}
    Kernel: {1x4 cell}
    Support: {1x4 cell}
    Width: [3x4 double]
```





## Input Arguments

**Tb1** — Sample data  
table

Sample data used to train the model, specified as a table. Each row of `Tbl` corresponds to one observation, and each column corresponds to one predictor variable. Optionally, `Tbl` can contain one additional column for the response variable. Multi-column variables and cell arrays other than cell arrays of character vectors are not allowed.

If `Tbl` contains the response variable, and you want to use all remaining variables in `Tbl` as predictors, then specify the response variable using `ResponseVarName`.

If `Tbl` contains the response variable, and you want to use only a subset of the remaining variables in `Tbl` as predictors, then specify a formula using `formula`.

If `Tbl` does not contain the response variable, then specify a response variable using `Y`. The length of response variable and the number of rows of `Tbl` must be equal.

Data Types: `table`

**`ResponseVarName` — Response variable name**

name of variable in `Tbl`

Response variable name, specified as the name of a variable in `Tbl`.

You must specify `ResponseVarName` as a character vector. For example, if the response variable `Y` is stored as `Tbl.Y`, then specify it as `'Y'`. Otherwise, the software treats all columns of `Tbl`, including `Y`, as predictors when training the model.

The response variable must be a categorical or character array, logical or numeric vector, or cell array of character vectors. If `Y` is a character array, then each element must correspond to one row of the array.

It is good practice to specify the order of the classes using the `ClassNames` name-value pair argument.

Data Types: `char`

**`formula` — Explanatory model of response and subset of predictor variables**

character vector

Explanatory model of the response and a subset of the predictor variables, specified as a character vector in the form of `'Y~X1+X2+X3'`. In this form, `Y` represents the response variable, and `X1`, `X2`, and `X3` represent the predictor variables. The variables must be variable names in `Tbl` (`Tbl.Properties.VariableNames`).

To specify a subset of variables in `Tbl` as predictors for training the model, use a formula. If you specify a formula, then the software does not use any variables in `Tbl` that do not appear in `formula`.

Data Types: `char`

### **Y — Class labels**

categorical array | character array | logical vector | numeric vector | cell array of character vectors

Class labels to which the naive Bayes classifier is trained, specified as a categorical or character array, logical or numeric vector, or cell array of character vectors. Each element of `Y` defines the class membership of the corresponding row of `X`. `Y` supports  $K$  class levels.

If `Y` is a character array, then each row must correspond to one class label.

The length of `Y` and the number of rows of `X` must be equivalent.

Data Types: `cell` | `char` | `double` | `logical`

### **x — Predictor data**

numeric matrix

Predictor data, specified as a numeric matrix.

Each row of `X` corresponds to one observation (also known as an instance or example), and each column corresponds to one variable (also known as a feature).

The length of `Y` and the number of rows of `X` must be equivalent.

Data Types: `double`

---

**Note:** The software treats NaN, empty character vector (' '), and `<undefined>` elements as missing values.

- If `Y` contains missing values, then the software removes them and the corresponding rows of `X`.
- If `X` contains any rows composed entirely of missing values, then the software removes those rows and the corresponding elements of `Y`.

- If  $X$  contains missing values and you set `'Distribution', 'mn'`, then the software removes those rows of  $X$  and the corresponding elements of  $Y$ .
- If a predictor is not represented in a class, that is, if all of its values are NaN within a class, then the software returns an error.

Removing rows of  $X$  and corresponding elements of  $Y$  decreases the effective training or cross-validation sample size.

---

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

---

**Note** You cannot use any cross-validation name-value pair along with `OptimizeHyperparameters`. You can modify the cross-validation for `OptimizeHyperparameters` only by using the `HyperparameterOptimizationOptions` name-value pair.

---

Example: `'Distribution', 'mn', 'Prior', 'uniform', 'KSWidth', 0.5` specifies that the data distribution is multinomial, the prior probabilities for all classes are equal, and the kernel smoothing window bandwidth for all classes is 0.5 units.

### Naive Bayes Options

#### **DistributionNames** — Data distributions

`'kernel' | 'mn' | 'mvmn' | 'normal'` | cell array of character vectors

Data distributions `fitcnb` uses to model the data, specified as the comma-separated pair consisting of `'DistributionNames'` and a value in this table.

Value	Description
<code>'kernel'</code>	Kernel smoothing density estimate.



Value	Description
'mn'	Multinomial distribution. If you specify mn, then all features are components of a multinomial distribution. Therefore, you cannot include 'mn' as an element of a cell array of character vectors. For details, see “Algorithms” on page 32-2083.
'mvmn'	Multivariate multinomial distribution. For details, see “Algorithms” on page 32-2083.
'normal'	Normal (Gaussian) distribution.

If you specify a character vector, then the software models all the features using that distribution. If you specify a 1-by- $P$  cell array of character vectors, then the software models feature  $j$  using the distribution in element  $j$  of the cell array.

By default, the software sets all predictors specified as categorical predictors (using the `CategoricalPredictors` name-value pair argument) to 'mvmn'. Otherwise, the default distribution is 'normal'.

You must specify that at least one predictor has distribution 'kernel' to additionally specify `Kernel`, `Support`, or `Width`.

Example: 'Distribution', 'mn'

Data Types: cell | char

### kernel — Kernel smoother type

'normal' (default) | 'box' | 'epanechnikov' | 'triangle' | cell array of character vectors

Kernel smoother type, specified as the comma-separated pair consisting of 'kernel' and a character vector or cell array of character vectors.

This table summarizes the available options for setting the kernel smoothing density region. Let  $I\{u\}$  denote the indicator function.

Value	Kernel	Formula
'box'	Box (uniform)	$f(x) = 0.5I\{ x  \leq 1\}$

Value	Kernel	Formula
'epanechnikov'	Epanechnikov	$f(x) = 0.75(1 - x^2)I\{ x  \leq 1\}$
'normal'	Gaussian	$f(x) = \frac{1}{\sqrt{2\pi}} \exp(-0.5x^2)$
'triangle'	Triangular	$f(x) = (1 -  x )I\{ x  \leq 1\}$

If you specify a 1-by- $P$  cell array, with each cell containing any value in the table, then the software trains the classifier using the kernel smoother type in cell  $j$  for feature  $j$  in  $X$ . The software ignores cells of `Kernel` not corresponding to a predictor whose distribution is 'kernel'.

You must specify that at least one predictor has distribution 'kernel' to additionally specify `Kernel`, `Support`, or `Width`.

Example: 'kernel', {'epanechnikov', 'normal'}

Data Types: cell | char

### Support — Kernel smoothing density support

'unbounded' (default) | 'positive' | cell array | numeric row vector

Kernel smoothing density support, specified as the comma-separated pair consisting of 'Support' and 'positive', 'unbounded', a cell array, or a numeric row vector. The software applies the kernel smoothing density to the specified region.

This table summarizes the available options for setting the kernel smoothing density region.

Value	Description
1-by-2 numeric row vector	For example, [L, U], where L and U are the finite lower and upper bounds, respectively, for the density support.
'positive'	The density support is all positive real values.
'unbounded'	The density support is all real values.

If you specify a 1-by- $P$  cell array, with each cell containing any value in the table, then the software trains the classifier using the kernel support in cell  $j$  for feature  $j$  in  $X$ . The software ignores cells of `Kernel` not corresponding to a predictor whose distribution is 'kernel'.

You must specify that at least one predictor has distribution 'kernel' to additionally specify Kernel, Support, or Width.

Example: 'KSSupport', {[-10,20], 'unbounded'}

Data Types: cell | char | double

### **width** — Kernel smoothing window width

matrix of numeric values | numeric column vector | numeric row vector | scalar

Kernel smoothing window width, specified as the comma-separated pair consisting of 'width' and a matrix of numeric values, numeric column vector, numeric row vector, or scalar.

Suppose there are  $K$  class levels and  $P$  predictors. This table summarizes the available options for setting the kernel smoothing window width.

Value	Description
$K$ -by- $P$ matrix of numeric values	Element $(k,j)$ specifies the width for predictor $j$ in class $k$ .
$K$ -by-1 numeric column vector	Element $k$ specifies the width for all predictors in class $k$ .
1-by- $P$ numeric row vector	Element $j$ specifies the width in all class levels for predictor $j$ .
scalar	Specifies the bandwidth for all features in all classes.

By default, the software selects a default width automatically for each combination of predictor and class by using a value that is optimal for a Gaussian distribution. If you specify Width and it contains NaNs, then the software selects widths for the elements containing NaNs.

You must specify that at least one predictor has distribution 'kernel' to additionally specify Kernel, Support, or Width.

Example: 'width', [NaN NaN]

Data Types: double | struct

### **Cross-Validation Options**

#### **CrossVal** — Cross-validation flag

'off' (default) | 'on'

Cross-validation flag, specified as the comma-separated pair consisting of 'Crossval' and 'on' or 'off'.

If you specify 'on', then the software implements 10-fold cross-validation.

To override this cross-validation setting, use one of these name-value pair arguments: `CVPartition`, `Holdout`, `KFold`, or `Leaveout`. To create a cross-validated model, you can use one cross-validation name-value pair argument at a time only.

Alternatively, cross-validate later by passing `Mdl` to `crossval`.

Example: 'CrossVal', 'on'

### **CVPartition — Cross-validation partition**

[] (default) | `cvpartition` partition object

Cross-validation partition, specified as the comma-separated pair consisting of 'CVPartition' and a `cvpartition` partition object as created by `cvpartition`. The partition object specifies the type of cross-validation, and also the indexing for training and validation sets.

To create a cross-validated model, you can use one of these four name-value pair arguments only: `CVPartition`, `Holdout`, `KFold`, or `Leaveout`.

Example: Suppose you create a random partition for 5-fold cross-validation on 500 observations by using `cvp = cvpartition(500, 'KFold', 5)`. Then, you can specify the cross-validation partition for `fitcecoc` as 'CVPartition', `cvp`.

### **Holdout — Fraction of data for holdout validation**

scalar value in the range (0,1)

Fraction of data used for holdout validation, specified as the comma-separated pair consisting of 'Holdout' and a scalar value in the range (0,1). If you specify 'Holdout',  $p$ , then the software:

- 1 Randomly reserves  $p*100\%$  of the data as validation data, and trains the model using the rest of the data
- 2 Stores the compact, trained model in the `Trained` property of the cross-validated model.

To create a cross-validated model, you can use one of these four name-value pair arguments only: `CVPartition`, `Holdout`, `KFold`, or `Leaveout`.

Example: 'Holdout', 0.1

Data Types: double | single

### **KFold** — Number of folds

10 (default) | positive integer value greater than 1

Number of folds to use in a cross-validated model, specified as the comma-separated pair consisting of 'KFold' and a positive integer value greater than 1. If you specify, e.g., 'KFold',  $k$ , then the software:

- 1 Randomly partitions the data into  $k$  sets
- 2 For each set, reserves the set as validation data, and trains the model using the other  $k - 1$  sets
- 3 Stores the  $k$  compact, trained models in the cells of a  $k$ -by-1 cell vector in the Trained property of the cross-validated model.

To create a cross-validated model, you can use one of these four name-value pair arguments only: CVPartition, Holdout, KFold, or Leaveout.

Example: 'KFold', 5

Data Types: single | double

### **Leaveout** — Leave-one-out cross-validation flag

'off' (default) | 'on'

Leave-one-out cross-validation flag, specified as the comma-separated pair consisting of 'Leaveout' and 'on' or 'off'. If you specify 'Leaveout', 'on', then, for each of the  $n$  observations, where  $n$  is `size(Mdl.X, 1)`, the software:

- 1 Reserves the observation as validation data, and trains the model using the other  $n - 1$  observations
- 2 Stores the  $n$  compact, trained models in the cells of an  $n$ -by-1 cell vector in the Trained property of the cross-validated model.

To create a cross-validated model, you can use one of these four name-value pair arguments only: CVPartition, Holdout, KFold, or Leaveout.

Example: 'Leaveout', 'on'

Data Types: char

**Other Classification Options****CategoricalPredictors** — Categorical predictors list

vector of positive integers | logical vector | character matrix | cell array of character vectors | 'all'

Categorical predictors list, specified as the comma-separated pair consisting of 'CategoricalPredictors' and one of these values:

Value	Description
Vector of positive integers	An entry in the vector is the index value corresponding to the column of the predictor data (X or Tbl) that contains a categorical variable.
Logical vector	A true entry means that the corresponding column of predictor data (X or Tbl) is a categorical variable.
Character matrix	Each row of the matrix is the name of a predictor variable. The names must match the entries in PredictorNames. Pad the names with extra blanks so each row of the character matrix has the same length.
Cell array of character vectors	Each element in the array is the name of a predictor variable. The names must match the entries in PredictorNames.
'all'	All predictors are categorical.

By default, if the predictor data is in a table (Tbl), `fitcnb` assumes that a variable is categorical if it contains logical values, categorical values, or a cell array of character vectors. If the predictor data is a matrix (X), `fitcnb` assumes all predictors are continuous. To identify any categorical predictors when the data is a matrix, use the 'CategoricalPredictors' name-value pair argument.

Example: 'CategoricalPredictors', 'all'

Data Types: single | double | logical | char | cell

**ClassNames** — Names of classes to use for training

categorical array | cell array of character vectors | character array | logical vector | numeric vector

Names of classes to use for training, specified as the comma-separated pair consisting of 'ClassNames' and a categorical or character array, logical or numeric vector, or cell array of character vectors. `ClassNames` must be the same data type as `Y`.

If `ClassNames` is a character array, then each element must correspond to one *row* of the array.

Use `ClassNames` to:

- Order the classes during training.
- Specify the order of any input or output argument dimension that corresponds to the class order. For example, use `ClassNames` to specify the order of the dimensions of `Cost` or the column order of classification scores returned by `predict`.
- Select a subset of classes for training. For example, suppose that the set of all distinct class names in `Y` is `{'a', 'b', 'c'}`. To train the model using observations from classes 'a' and 'c' only, specify `'ClassNames', {'a', 'c'}`.

The default is the set of all distinct class names in `Y`.

Example: `'ClassNames', {'b', 'g'}`

Data Types: categorical | char | logical | single | double | cell

### **Cost — Cost of misclassification**

square matrix | structure

Cost of misclassification of a point, specified as the comma-separated pair consisting of `'Cost'` and one of the following:

- Square matrix, where `Cost(i, j)` is the cost of classifying a point into class `j` if its true class is `i` (i.e., the rows correspond to the true class and the columns correspond to the predicted class). To specify the class order for the corresponding rows and columns of `Cost`, additionally specify the `ClassNames` name-value pair argument.
- Structure `S` having two fields: `S.ClassNames` containing the group names as a variable of the same type as `Y`, and `S.ClassificationCosts` containing the cost matrix.

The default is `Cost(i, j)=1` if `i~j`, and `Cost(i, j)=0` if `i=j`.

Example: `'Cost', struct('ClassNames', {'b', 'g'}, 'ClassificationCosts', [0 0.5; 1 0])`

Data Types: single | double | struct

### **PredictorNames — Predictor variable names**

cell array of unique character vectors

Predictor variable names, specified as the comma-separated pair consisting of 'PredictorNames' and a cell array of unique character vectors. The functionality of 'PredictorNames' depends on the way you supply the training data.

- If you supply  $X$  and  $Y$ , then you can use 'PredictorNames' to give the predictor variables in  $X$  names.
  - The order of the names in PredictorNames must correspond to the column order of  $X$ . That is, PredictorNames{1} is the name of  $X(:, 1)$ , PredictorNames{2} is the name of  $X(:, 2)$ , and so on. Also,  $\text{size}(X, 2)$  and  $\text{numel}(\text{PredictorNames})$  must be equal.
  - By default, PredictorNames is {'x1', 'x2', ...}.
- If you supply Tbl, then you can use 'PredictorNames' to choose which predictor variables to use in training. That is, fitcnb uses the predictor variables in PredictorNames and the response only in training.
  - PredictorNames must be a subset of Tbl.Properties.VariableNames and cannot include the name of the response variable.
  - By default, PredictorNames contains the names of all predictor variables.
  - It good practice to specify the predictors for training using one of 'PredictorNames' or formula only.

Example: 'PredictorNames',  
{'SepalLength', 'SepalWidth', 'PedalLength', 'PedalWidth'}

Data Types: cell

### Prior — Prior probabilities

'empirical' (default) | 'uniform' | vector of scalar values | structure

Prior probabilities for each class, specified as the comma-separated pair consisting of 'Prior' and a value in this table.

Value	Description
'empirical'	The class prior probabilities are the class relative frequencies in $Y$ .
'uniform'	All class prior probabilities are equal to $1/K$ , where $K$ is the number of classes.



Value	Description
numeric vector	Each element is a class prior probability. Order the elements according to <code>Mdl.ClassNames</code> or specify the order using the <code>ClassNames</code> name-value pair argument. The software normalizes the elements such that they sum to 1.
structure	A structure <code>S</code> with two fields: <ul style="list-style-type: none"> <li>• <code>S.ClassNames</code> contains the class names as a variable of the same type as <code>Y</code>.</li> <li>• <code>S.ClassProbs</code> contains a vector of corresponding prior probabilities. The software normalizes the elements such that they sum to 1.</li> </ul>

If you set values for both `Weights` and `Prior`, the weights are renormalized to add up to the value of the prior probability in the respective class.

Example: `'Prior','uniform'`

Data Types: `single` | `double` | `struct`

### **ResponseName** — Response variable name

`'Y'` (default) | character vector

Response variable name, specified as the comma-separated pair consisting of `'ResponseName'` and a character vector.

- If you supply `Y`, then you can use `'ResponseName'` to specify a name for the response variable.
- If you supply `ResponseVarName` or `formula`, then you cannot use `'ResponseName'`.

Example: `'ResponseName','response'`

Data Types: `char`

### **ScoreTransform** — Score transformation

`'none'` (default) | character vector | function handle

Score transformation, specified as the comma-separated pair consisting of 'ScoreTransform' and either a character vector or a function handle.

This table summarizes the available character vectors.

Value	Description
'doublelogit'	$1/(1 + e^{-2x})$
'invlogit'	$\log(x / (1-x))$
'ismax'	Set the score for the class with the largest score to 1, and set the scores for all other classes to 0.
'logit'	$1/(1 + e^{-x})$
'none' or 'identity'	$x$ (no transformation)
'sign'	-1 for $x < 0$ 0 for $x = 0$ 1 for $x > 0$
'symmetric'	$2x - 1$
'symmetricismax'	Set the score for the class with the largest score to 1, and set the scores for all other classes to -1.
'symmetriclogit'	$2/(1 + e^{-x}) - 1$

For a MATLAB function or a function you define, use its function handle. The function handle must accept a matrix (the original scores) and return a matrix of the same size (the transformed scores).

Example: 'ScoreTransform', 'logit'

Data Types: char | function\_handle

### Weights — Observation weights

numeric vector of positive values | character vector

Observation weights, specified as the comma-separated pair consisting of 'Weights' and a numeric vector of positive values or name of a variable in Tbl. The software weighs the observations in each row of X or Tbl with the corresponding value in Weights. The size of Weights must equal the number of rows of X or Tbl.

If you specify the input data as a table Tbl, then Weights can be the name of a variable in Tbl that contains a numeric vector. In this case, you must specify Weights as a

character vector. For example, if the weights vector  $W$  is stored as `Tbl.W`, then specify it as `'W'`. Otherwise, the software treats all columns of `Tbl`, including  $W$ , as predictors or the response when training the model.

The software normalizes `Weights` to sum up to the value of the prior probability in the respective class.

By default, `Weights` is ones  $(n, 1)$ , where  $n$  is the number of observations in `X` or `Tbl`.

Data Types: `double` | `single` | `char`

### Hyperparameter Optimization

#### OptimizeHyperparameters — Parameters to optimize

'none' (default) | 'auto' | 'all' | cell array of eligible parameter names | vector of `optimizableVariable` objects

Parameters to optimize, specified as:

- 'none' — Do not optimize.
- 'auto' — Use `{'DistributionNames', 'Width'}`
- 'all' — Optimize all eligible parameters.
- Cell array of eligible parameter names
- Vector of `optimizableVariable` objects, typically the output of `hyperparameters`

The optimization attempts to minimize the cross-validation loss (error) for `fitcnb` by varying the parameters. For information about cross-validation loss (albeit in a different context), see “Classification Loss” on page 32-3250. To control the cross-validation type and other aspects of the optimization, use the `HyperparameterOptimizationOptions` name-value pair.

---

**Note** `OptimizeHyperparameters` values override any values you set using other name-value pairs. For example, setting `OptimizeHyperparameters` to 'auto' causes the 'auto' values to apply.

---

The eligible parameters for `fitcnb` are:

- `DistributionNames` — `fitcnb` searches among 'normal' and 'kernel'.

- **Width** — `fitcnb` searches among real values, by default log-scaled in the range `[MinPredictorDiff/4,max(MaxPredictorRange,MinPredictorDiff)]`.
- **Kernel** — `fitcnb` searches among `'normal'`, `'box'`, `'epanechnikov'`, and `'triangle'`.

Set nondefault parameters by passing a vector of `optimizableVariable` objects that have nondefault values. For example,

```
load fisheriris
params = hyperparameters('fitcnb',meas,species);
params(2).Range = [1e-2,1e2];
```

Pass `params` as the value of `OptimizeHyperparameters`.

By default, iterative display appears at the command line, and plots appear according to the number of hyperparameters in the optimization. For the optimization and plots, the objective function is `log(1 + cross-validation loss)` for regression, and the misclassification rate for classification. To control the iterative display, set the `HyperparameterOptimizationOptions` name-value pair, `Verbose` field. To control the plots, set the `HyperparameterOptimizationOptions` name-value pair, `ShowPlots` field.

For an example, see “Optimize Naive Bayes Classifier” on page 32-2058.

Example: `'auto'`

Data Types: `char` | `cell`

### **HyperparameterOptimizationOptions** — Options for optimization

structure

Options for optimization, specified as a structure. Modifies the effect of the `OptimizeHyperparameters` name-value pair. All fields in the structure are optional.

Field Name	Values	Default
Optimizer	<ul style="list-style-type: none"> <li>'bayesopt' — Use Bayesian optimization. Internally, this setting calls bayesopt.</li> <li>'gridsearch' — Use grid search with NumGridDivisions values per dimension.</li> <li>'randomsearch' — Search at random among MaxObjectiveEvaluations points.</li> </ul> <p>'gridsearch' searches in a random order, using uniform sampling without replacement from the grid. After optimization, you can get a table in grid order by using the command sortrows (Mdl.ParameterOptimizationResults).</p>	'bayesopt'
AcquisitionFunctionName	<ul style="list-style-type: none"> <li>'expected-improvement-per-second-plus'</li> <li>'expected-improvement'</li> <li>'expected-improvement-plus'</li> <li>'expected-improvement-per-second'</li> <li>'lower-confidence-bound'</li> <li>'probability-of-improvement'</li> </ul> <p>For details, see the bayesopt AcquisitionFunctionName name-value pair, or “Acquisition Function Types” on page 10-4.</p>	'expected-improvement-per-second-plus'
MaxObjectiveEvaluations	Maximum number of objective function evaluations.	30 for 'bayesopt' or 'randomsearch', and the entire grid for 'gridsearch'

Field Name	Values	Default
MaxTime	Time limit, specified as a positive real. The time limit is in seconds, as measured by <code>tic</code> and <code>toc</code> . Run time can exceed <code>MaxTime</code> because <code>MaxTime</code> does not interrupt function evaluations.	Inf
NumGridDivisions	For 'gridsearch', the number of values in each dimension. Can be a vector of positive integers giving the number of values for each dimension, or a scalar that applies to all dimensions. Ignored for categorical variables.	10
ShowPlots	Logical value indicating whether to show plots. If <code>true</code> , plots the best objective function value against iteration number. If there are one or two optimization parameters, and if <code>Optimizer</code> is 'bayesopt', then <code>ShowPlots</code> also plots a model of the objective function against the parameters.	<code>true</code>
SaveIntermediateResults	Logical value indicating whether to save results when <code>Optimizer</code> is 'bayesopt'. If <code>true</code> , overwrites a workspace variable named 'BayesoptResults' at each iteration. The variable is a <code>BayesianOptimization</code> object.	<code>false</code>
Verbose	Display to the command line. <ul style="list-style-type: none"> <li>• 0 — No iterative display</li> <li>• 1 — Iterative display</li> <li>• 2 — Iterative display with extra information</li> </ul> For details, see the <code>bayesopt</code> <code>Verbose</code> name-value pair.	1

Field Name	Values	Default
UseParallel	Logical value indicating whether to run Bayesian optimization in parallel, which requires Parallel Computing Toolbox. For details, see “Parallel Bayesian Optimization” on page 10-9.	false
Repartition	Logical value indicating whether to repartition the cross-validation at every iteration. If <code>false</code> , the optimizer uses a single partition for the optimization.  <code>true</code> usually gives the most robust results because this setting takes partitioning noise into account. However, for good results, <code>true</code> requires at least twice as many function evaluations.	false
Use no more than one of the following three field names.		
CVPartition	A <code>cvpartition</code> object, as created by <code>cvpartition</code>	Kfold = 5
Holdout	A scalar in the range $(0, 1)$ representing the holdout fraction.	
Kfold	An integer greater than 1.	

Example: `struct('MaxObjectiveEvaluations', 60)`

Data Types: `struct`

## Output Arguments

### **mdl** — Trained naive Bayes classification model

`ClassificationNaiveBayes` model object | `ClassificationPartitionedModel` cross-validated model object

Trained naive Bayes classification model, returned as a `ClassificationNaiveBayes` model object or a `ClassificationPartitionedModel` cross-validated model object.

If you set any of the name-value pair arguments `KFold`, `Holdout`, `CrossVal`, or `CVPartition`, then `Mdl` is a `ClassificationPartitionedModel` cross-validated model object. Otherwise, `Mdl` is a `ClassificationNaiveBayes` model object.

To reference properties of `Mdl`, use dot notation. For example, to access the estimated distribution parameters, enter `Mdl.DistributionParameters`.

## Definitions

### Bag-of-Tokens Model

In the bag-of-tokens model, the value of predictor  $j$  is the nonnegative number of occurrences of token  $j$  in this observation. The number of categories (bins) in this multinomial model is the number of distinct tokens, that is, the number of predictors.

### Naive Bayes

Naive Bayes is a classification algorithm that applies density estimation to the data.

The algorithm leverages Bayes theorem, and (naively) assumes that the predictors are conditionally independent, given the class. Though the assumption is usually violated in practice, naive Bayes classifiers tend to yield posterior distributions that are robust to biased class density estimates, particularly where the posterior is 0.5 (the decision boundary) [1].

Naive Bayes classifiers assign observations to the most probable class (in other words, the maximum a posteriori decision rule). Explicitly, the algorithm:

- 1 Estimates the densities of the predictors within each class.
- 2 Models posterior probabilities according to Bayes rule. That is, for all  $k = 1, \dots, K$ ,

$$\hat{P}(Y = k | X_1, \dots, X_P) = \frac{\pi(Y = k) \prod_{j=1}^P P(X_j | Y = k)}{\sum_{k=1}^K \pi(Y = k) \prod_{j=1}^P P(X_j | Y = k)},$$

where:



- $Y$  is the random variable corresponding to the class index of an observation.
  - $X_1, \dots, X_P$  are the random predictors of an observation.
  - $\pi(Y = k)$  is the prior probability that a class index is  $k$ .
- 3 Classifies an observation by estimating the posterior probability for each class, and then assigns the observation to the class yielding the maximum posterior probability.

If the predictors compose a multinomial distribution, then the posterior probability

$\hat{P}(Y = k | X_1, \dots, X_P) \propto \pi(Y = k) P_{mn}(X_1, \dots, X_P | Y = k)$ , where  $P_{mn}(X_1, \dots, X_P | Y = k)$  is the probability mass function of a multinomial distribution.

## Tips

For classifying count-based data, such as the bag-of-tokens model on page 32-2082, use the multinomial distribution (e.g., set 'Distribution', 'mn').

## Algorithms

- If you specify 'Distribution', 'mn' when training Mdl using fitcnb, then the software fits a multinomial distribution using the bag-of-tokens model on page 32-2082. The software stores the probability that token  $j$  appears in class  $k$  in the property `DistributionParameters{k, j}`. Using additive smoothing [2], the estimated probability is

$$P(\text{token } j | \text{class } k) = \frac{1 + c_{j|k}}{P + c_k},$$

where:

- 

$$c_{j|k} = n_k \frac{\sum_{i: y_i \in \text{class } k} x_{ij} w_i}{\sum_{i: y_i \in \text{class } k} w_i}; \text{ which is the weighted number of occurrences of token } j \text{ in class } k.$$

- $n_k$  is the number of observations in class  $k$ .
- $w_i$  is the weight for observation  $i$ . The software normalizes weights within a class such that they sum to the prior probability for that class.
- 

$$c_k = \sum_{j \neq 1}^P c_{j|k}; \text{ which is the total weighted number of occurrences of all tokens in class } k$$

- If you specify 'Distribution', 'mvnm' when training Mdl using fitcnb, then:
  - 1 For each predictor, the software collects a list of the unique levels, stores the sorted list in CategoricalLevels, and considers each level a bin. Each predictor/class combination is a separate, independent multinomial random variable.
  - 2 For predictor  $j$  in class  $k$ , the software counts instances of each categorical level using the list stored in CategoricalLevels{ $j$ }.
  - 3 The software stores the probability that predictor  $j$ , in class  $k$ , has level  $L$  in the property DistributionParameters{ $k, j$ }, for all levels in CategoricalLevels{ $j$ }. Using additive smoothing [2], the estimated probability is

$$P(\text{predictor } j = L \mid \text{class } k) = \frac{1 + m_{j|k}(L)}{m_j + m_k},$$

where:

- 

$$m_{j|k}(L) = n_k \frac{\sum_{i: y_i \in \text{class } k} I\{x_{ij} = L\} w_i}{\sum_{i: y_i \in \text{class } k} w_i}; \text{ which is the weighted number of observations for which predictor } j \text{ equals } L \text{ in class } k.$$

- $n_k$  is the number of observations in class  $k$ .
- 

$$I\{x_{ij} = L\} = 1 \text{ if } x_{ij} = L, 0 \text{ otherwise.}$$

- $w_i$  is the weight for observation  $i$ . The software normalizes weights within a class such that they sum to the prior probability for that class.
- $m_j$  is the number of distinct levels in predictor  $j$ .
- $m_k$  is the weighted number of observations in class  $k$ .

## References

- [1] Hastie, T., R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*, Second Edition. NY: Springer, 2008.
- [2] Manning, C. D., P. Raghavan, and M. Schütze. *Introduction to Information Retrieval*, NY: Cambridge University Press, 2008.

## Extended Capabilities

### Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function supports tall arrays for out-of-memory data with the limitations:

- Supported syntaxes are:
  - `discr = fitcnb(Tbl, Y)`
  - `discr = fitcnb(X, Y)`
  - `discr = fitcnb(___, Name, Value)`
- Options related to kernel densities, cross-validation, and hyperparameter optimization are not supported. The supported name-value pairs are:
  - 'DistributionNames' — 'kernel' value is not supported.
  - 'CategoricalPredictors'
  - 'PredictorNames'
  - 'ResponseName'
  - 'ScoreTransform'

- 'Weights' — Value must be a tall array.

For more information, see “Tall Arrays” (MATLAB).

## See Also

`ClassificationNaiveBayes` | `ClassificationPartitionedModel` | `predict` | `templateNaiveBayes`

## Topics

“Naive Bayes Classification” on page 21-2

“Grouping Variables” on page 2-59

**Introduced in R2014b**

## fitcsvm

Train binary support vector machine classifier

`fitcsvm` trains or cross-validates a support vector machine (SVM) model for two-class (binary) classification on a low- through moderate-dimensional predictor data set. `fitcsvm` supports mapping the predictor data using kernel functions, and supports SMO, ISDA, or  $L1$  soft-margin minimization via quadratic programming for objective-function minimization.

To train a linear SVM model for binary classification on a high-dimensional data set, that is, data sets that include many predictor variables, use `fitclinear` instead.

For multiclass learning by combining binary SVM models, use error-correcting output codes (ECOC). For more details, see `fitcecoc`.

To train an SVM regression model, see `fitrsvm` for low- through moderate-dimensional predictor data sets, or `fitrlinear` for high-dimensional data sets.

## Syntax

```
Mdl = fitcsvm(Tbl,ResponseVarName)
```

```
Mdl = fitcsvm(Tbl,formula)
```

```
Mdl = fitcsvm(Tbl,Y)
```

```
Mdl = fitcsvm(X,Y)
```

```
Mdl = fitcsvm(____,Name,Value)
```

## Description

`Mdl = fitcsvm(Tbl,ResponseVarName)` returns a support vector machine classifier on page 32-2129 `Mdl` trained using the sample data contained in a table (`Tbl`).

`ResponseVarName` is the name of the variable in `Tbl` that contains the class labels for one- or two-class classification.

`Mdl = fitcsvm(Tbl, formula)` returns an SVM classifier trained using the sample data contained in a table (`Tbl`). `formula` is an explanatory model of the response and a subset of predictor variables in `Tbl` used to fit `Mdl`.

`Mdl = fitcsvm(Tbl, Y)` returns an SVM classifier trained using the predictor variables in table `Tbl` and class labels in vector `Y`.

`Mdl = fitcsvm(X, Y)` returns an SVM classifier trained using the predictors in the matrix `X` and class labels in vector `Y` for one- or two-class classification.

`Mdl = fitcsvm( ____, Name, Value)` returns a support vector machine classifier with additional options specified by one or more `Name, Value` pair arguments, using any of the previous syntaxes. For example, you can specify the type of cross-validation, the cost for misclassification, or the type of score transformation function.

## Examples

### Train a Support Vector Machine Classifier

Load Fisher's iris data set. Remove the sepal lengths and widths, and all observed setosa irises.

```
load fisheriris
inds = ~strcmp(species, 'setosa');
X = meas(inds, 3:4);
y = species(inds);
```

Train an SVM classifier using the processed data set.

```
SVModel = fitcsvm(X, y)
```

```
SVModel =
```

```
ClassificationSVM
    ResponseName: 'Y'
    CategoricalPredictors: []
           ClassNames: {'versicolor' 'virginica'}
    ScoreTransform: 'none'
    NumObservations: 100
```

```

        Alpha: [24x1 double]
        Bias: -14.4149
KernelParameters: [1x1 struct]
  BoxConstraints: [100x1 double]
  ConvergenceInfo: [1x1 struct]
  IsSupportVector: [100x1 logical]
  Solver: 'SMO'

```

The Command Window shows that `SVModel` is a trained `ClassificationSVM` classifier and a property list. Display the properties of `SVModel`, for example, to determine the class order, by using dot notation.

```
classOrder = SVModel.ClassNames
```

```

classOrder =

    2x1 cell array

    {'versicolor'}
    {'virginica' }

```

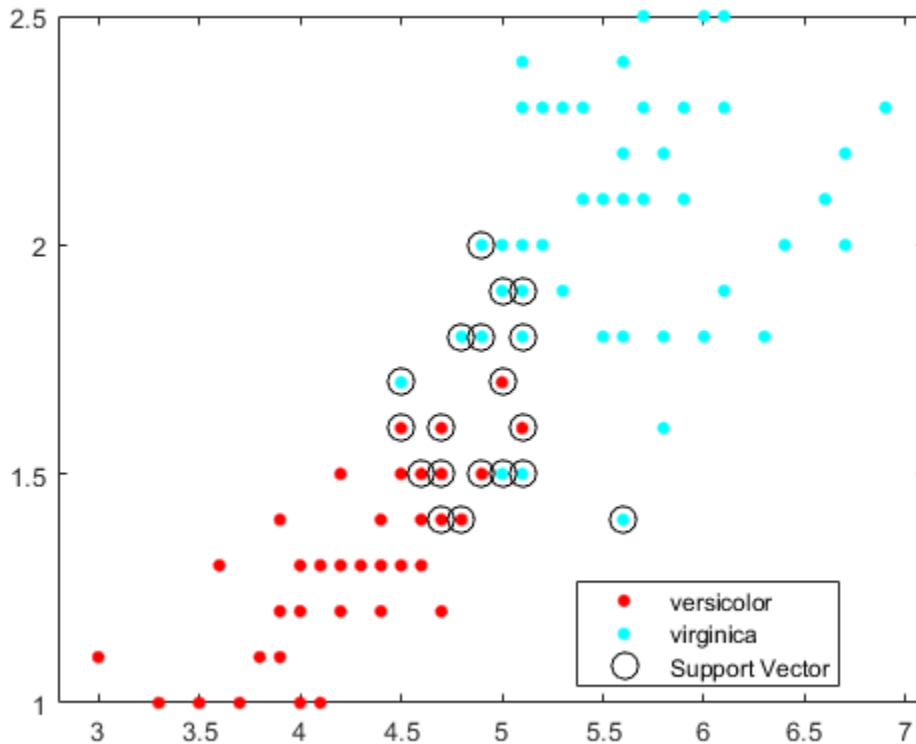
The first class ('versicolor') is the negative class, and the second ('virginica') is the positive class. You can change the class order during training by using the 'ClassNames' name-value pair argument.

Plot a scatter diagram of the data and circle the support vectors.

```

sv = SVModel.SupportVectors;
figure
gscatter(X(:,1),X(:,2),y)
hold on
plot(sv(:,1),sv(:,2),'ko','MarkerSize',10)
legend('versicolor','virginica','Support Vector')
hold off

```



The support vectors are observations that occur on or beyond their estimated class boundaries.

You can adjust the boundaries (and therefore the number of support vectors) by setting a box constraint during training using the 'BoxConstraint' name-value pair argument.

### Train and Cross Validate an SVM Classifier

Load the ionosphere data set.

```
load ionosphere
rng(1); % For reproducibility
```



Train an SVM classifier using the radial basis kernel. Let the software find a scale value for the kernel function. It is good practice to standardize the predictors.

```
SVMMModel = fitcsvm(X,Y,'Standardize',true,'KernelFunction','RBF',...  
    'KernelScale','auto');
```

SVMMModel is a trained ClassificationSVM classifier.

Cross validate the SVM classifier. By default, the software uses 10-fold cross validation.

```
CVSVMMModel = crossval(SVMMModel);
```

CVSVMMModel is a ClassificationPartitionedModel cross-validated classifier.

Estimate the out-of-sample misclassification rate.

```
classLoss = kfoldLoss(CVSVMMModel)
```

```
classLoss =  
  
    0.0484
```

The generalization rate is approximately 5%.

## Detect Outliers Using SVM and One-Class Learning

Load Fisher's iris data set. Remove the petal lengths and widths. Treat all irises as coming from the same class.

```
load fisheriris  
X = meas(:,1:2);  
y = ones(size(X,1),1);
```

Train an SVM classifier using the processed data set. Assume that 5% of the observations are outliers. It is good practice to standardize the predictors.

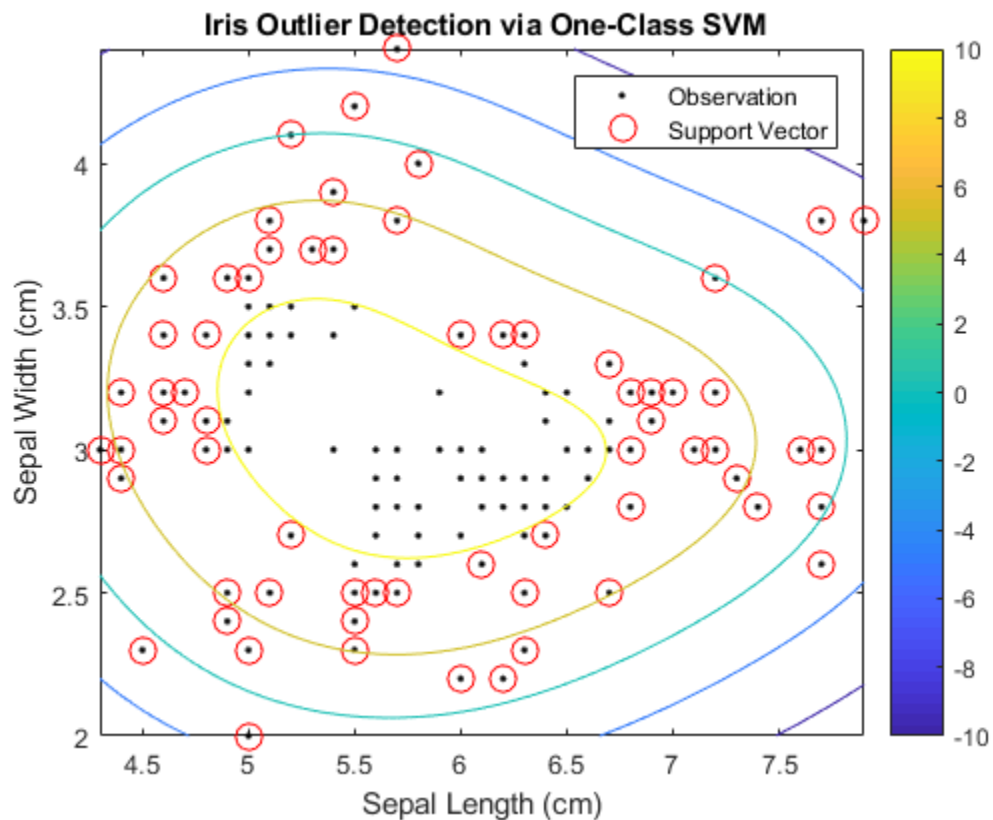
```
rng(1);  
SVMMModel = fitcsvm(X,y,'KernelScale','auto','Standardize',true,...  
    'OutlierFraction',0.05);
```

`SVMMModel` is a trained `ClassificationSVM` classifier. By default, the software uses the Gaussian kernel for one-class learning.

Plot the observations and the decision boundary. Flag the support vectors and potential outliers.

```
svInd = SVMMModel.IsSupportVector;
h = 0.02; % Mesh grid step size
[X1,X2] = meshgrid(min(X(:,1)):h:max(X(:,1)), ...
    min(X(:,2)):h:max(X(:,2)));
[~,score] = predict(SVMMModel,[X1(:),X2(:)]);
scoreGrid = reshape(score,size(X1,1),size(X2,2));

figure
plot(X(:,1),X(:,2),'k.')
hold on
plot(X(svInd,1),X(svInd,2),'ro','MarkerSize',10)
contour(X1,X2,scoreGrid)
colorbar;
title('\bf Iris Outlier Detection via One-Class SVM')
xlabel('Sepal Length (cm)')
ylabel('Sepal Width (cm)')
legend('Observation','Support Vector')
hold off
```



The boundary separating the outliers from the rest of the data occurs where the contour value is 0.

Verify that the fraction of observations with negative scores in the cross-validated data is close to 5%.

```
CVSVMModel = crossval(SVMModel);
[~,scorePred] = kfoldPredict(CVSVMModel);
outlierRate = mean(scorePred<0)
```

```
outlierRate =
```

```
0.0467
```

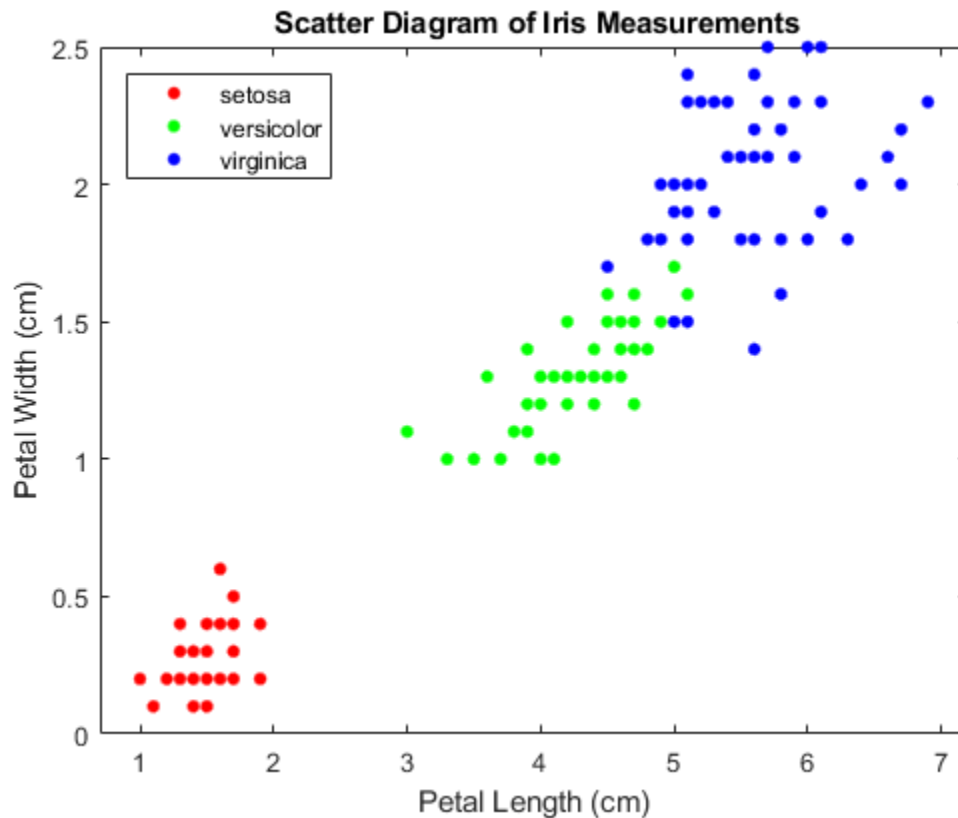
### Find Multiple Class Boundaries Using Binary SVM

Load Fisher's iris data set. Use the petal lengths and widths.

```
load fisheriris
X = meas(:,3:4);
Y = species;
```

Examine a scatter plot of the data.

```
figure
gscatter(X(:,1),X(:,2),Y);
h = gca;
lims = [h.XLim h.YLim]; % Extract the x and y axis limits
title('\bf Scatter Diagram of Iris Measurements');
xlabel('Petal Length (cm)');
ylabel('Petal Width (cm)');
legend('Location', 'Northwest');
```



There are three classes, one of which is linearly separable from the others.

For each class:

- 1 Create a logical vector (`indx`) indicating whether an observation is a member of the class.
- 2 Train an SVM classifier using the predictor data and `indx`.
- 3 Store the classifier in a cell of a cell array.

It is good practice to define the class order.

```
SVMModels = cell(3,1);
classes = unique(Y);
```

```
rng(1); % For reproducibility

for j = 1:numel(classes);
    indx = strcmp(Y,classes(j)); % Create binary classes for each classifier
    SVMModels{j} = fitcsvm(X,indx,'ClassNames',[false true],'Standardize',true,...
        'KernelFunction','rbf','BoxConstraint',1);
end
```

SVMModels is a 3-by-1 cell array, with each cell containing a ClassificationSVM classifier. For each cell, the positive class is setosa, versicolor, and virginica, respectively.

Define a fine grid within the plot, and treat the coordinates as new observations from the distribution of the training data. Estimate the score of the new observations using each classifier.

```
d = 0.02;
[x1Grid,x2Grid] = meshgrid(min(X(:,1)):d:max(X(:,1)),...
    min(X(:,2)):d:max(X(:,2)));
xGrid = [x1Grid(:),x2Grid(:)];
N = size(xGrid,1);
Scores = zeros(N,numel(classes));

for j = 1:numel(classes);
    [~,score] = predict(SVMModels{j},xGrid);
    Scores(:,j) = score(:,2); % Second column contains positive-class scores
end
```

Each row of Scores contains three scores. The index of the element with the largest score is the index of the class to which the new class observation most likely belongs.

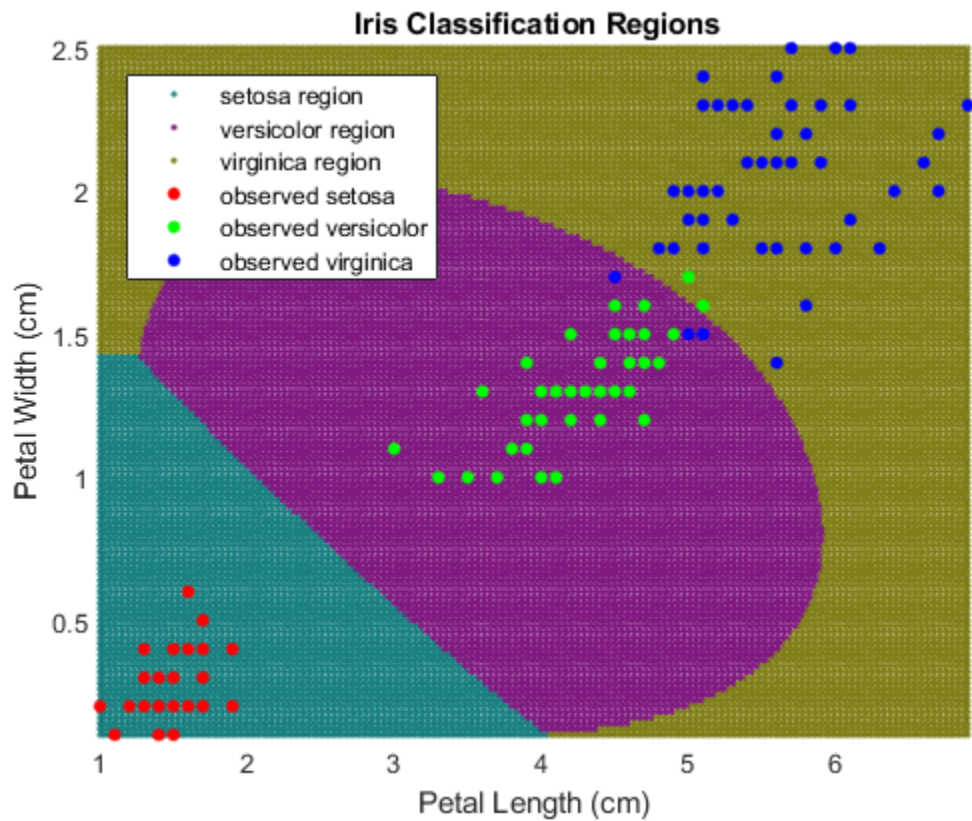
Associate each new observation with the classifier that gives it the maximum score.

```
[~,maxScore] = max(Scores,[],2);
```

Color in the regions of the plot based on which class the corresponding new observation belongs.

```
figure
h(1:3) = gscatter(xGrid(:,1),xGrid(:,2),maxScore,...
    [0.1 0.5 0.5; 0.5 0.1 0.5; 0.5 0.5 0.1]);
hold on
h(4:6) = gscatter(X(:,1),X(:,2),Y);
title('\bf Iris Classification Regions');
xlabel('Petal Length (cm)');
```

```
ylabel('Petal Width (cm)');  
legend(h,{'setosa region','versicolor region','virginica region',...  
         'observed setosa','observed versicolor','observed virginica'},...  
        'Location','Northwest');  
axis tight  
hold off
```



### Optimize SVM Classifier

This example shows how to optimize hyperparameters automatically using `fitcsvm`. The example uses the `ionosphere` data.

Load the data.

```
load ionosphere
```

Find hyperparameters that minimize five-fold cross-validation loss by using automatic hyperparameter optimization.

For reproducibility, set the random seed and use the 'expected-improvement-plus' acquisition function.

```
rng default
Mdl = fitcsvm(X,Y,'OptimizeHyperparameters','auto',...
    'HyperparameterOptimizationOptions',struct('AcquisitionFunctionName',...
    'expected-improvement-plus'))
```

```
=====
| Iter | Eval  | Objective | Objective | BestSoFar | BestSoFar | BoxConstrain-
|      | result|           | runtime   | (observed) | (estim.)  | t
|=====|
|  1  | Best  | 0.21937  | 74.402   | 0.21937   | 0.21937   | 64.836
|  2  | Accept| 0.35897  | 2.8334   | 0.21937   | 0.22807   | 0.036335
|  3  | Best  | 0.13105  | 29.382   | 0.13105   | 0.14149   | 0.0022147
|  4  | Accept| 0.35897  | 3.6978   | 0.13105   | 0.13108   | 5.1259
|  5  | Accept| 0.1339   | 54.823   | 0.13105   | 0.13111   | 0.0011599
|  6  | Accept| 0.13105  | 14.353   | 0.13105   | 0.13106   | 0.0010151
|  7  | Best  | 0.12821  | 33.711   | 0.12821   | 0.12819   | 0.0010563
|  8  | Accept| 0.1339   | 31.209   | 0.12821   | 0.13014   | 0.0010113
|  9  | Accept| 0.12821  | 25.41    | 0.12821   | 0.12977   | 0.0010934
| 10  | Accept| 0.12821  | 18.768   | 0.12821   | 0.12934   | 0.0010315
| 11  | Accept| 0.12821  | 71.538   | 0.12821   | 0.12954   | 994.04
| 12  | Accept| 0.15385  | 54.074   | 0.12821   | 0.12961   | 959.35
| 13  | Accept| 0.1339   | 19.832   | 0.12821   | 0.12964   | 990.56
| 14  | Accept| 0.13675  | 26.302   | 0.12821   | 0.12965   | 83.935
| 15  | Accept| 0.13675  | 32.217   | 0.12821   | 0.12971   | 945.18
| 16  | Accept| 0.1339   | 17.235   | 0.12821   | 0.12985   | 0.13199
| 17  | Accept| 0.1339   | 12.57    | 0.12821   | 0.12975   | 0.0095538
| 18  | Accept| 0.35897  | 0.21069  | 0.12821   | 0.12973   | 0.0010157
| 19  | Accept| 0.1339   | 19.647   | 0.12821   | 0.12971   | 2.0106
| 20  | Accept| 0.1339   | 2.3136   | 0.12821   | 0.12978   | 0.018418
|=====|
| Iter | Eval  | Objective | Objective | BestSoFar | BestSoFar | BoxConstrain-
|      | result|           | runtime   | (observed) | (estim.)  | t
|=====|
|  21 | Accept| 0.13675  | 1.8495   | 0.12821   | 0.12988   | 0.0010427
|  22 | Accept| 0.12821  | 16.043   | 0.12821   | 0.12978   | 0.20189
```



	23	Accept		0.12821		15.674		0.12821		0.12966		0.19783
	24	Accept		0.1339		17.895		0.12821		0.12976		0.19762
	25	Accept		0.12821		44.021		0.12821		0.12967		0.22514
	26	Accept		0.12821		37.321		0.12821		0.12877		0.1686
	27	Accept		0.12821		31.133		0.12821		0.12848		0.15755
	28	Accept		0.35897		3.6136		0.12821		0.12848		990.01
	29	Accept		0.13105		19.558		0.12821		0.12898		0.24825
	30	Best		0.12251		0.44609		0.12251		0.12277		998.58

---

Optimization completed.

MaxObjectiveEvaluations of 30 reached.

Total function evaluations: 30

Total elapsed time: 837.0454 seconds.

Total objective function evaluation time: 732.084

Best observed feasible point:

BoxConstraint	KernelScale
998.58	23.666

Observed objective function value = 0.12251

Estimated objective function value = 0.12277

Function evaluation time = 0.44609

Best estimated feasible point (according to models):

BoxConstraint	KernelScale
998.58	23.666

Estimated objective function value = 0.12277

Estimated function evaluation time = 0.53667

Mdl =

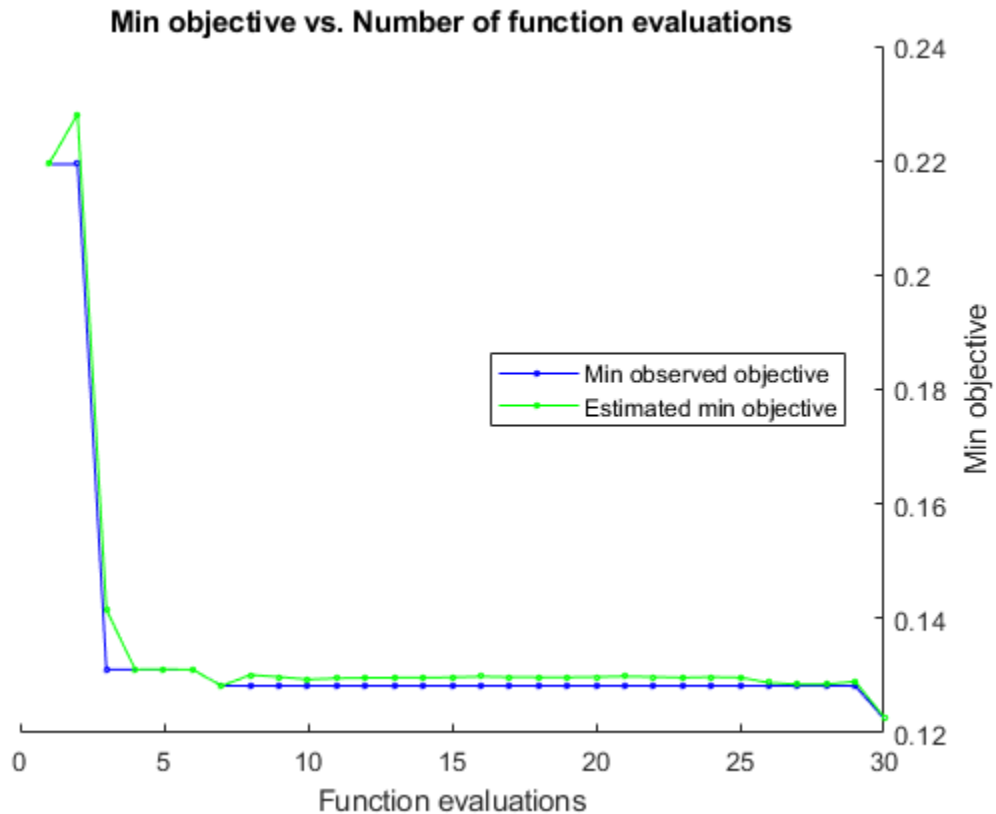
ClassificationSVM

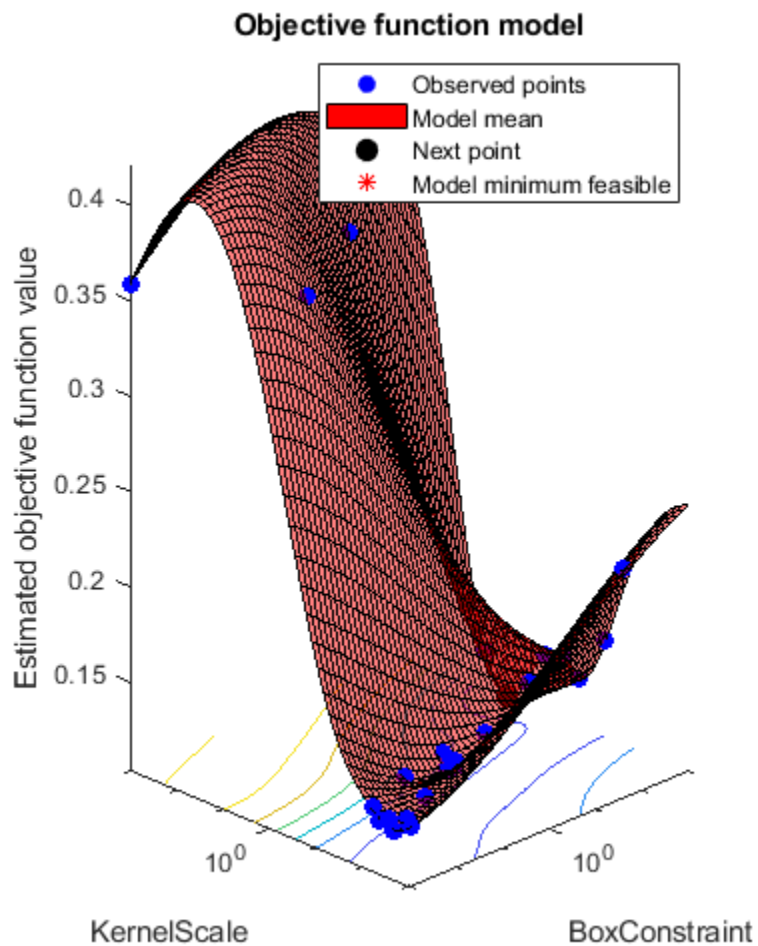
```

      ResponseName: 'Y'
  CategoricalPredictors: []
      ClassNames: {'b' 'g'}
      ScoreTransform: 'none'
      NumObservations: 351

```

```
HyperparameterOptimizationResults: [1x1 BayesianOptimization]
      Alpha: [98x1 double]
      Bias: -4.8231
KernelParameters: [1x1 struct]
      BoxConstraints: [351x1 double]
      ConvergenceInfo: [1x1 struct]
      IsSupportVector: [351x1 logical]
      Solver: 'SMO'
```





- “Train SVM Classifiers Using a Gaussian Kernel” on page 18-195
- “Train SVM Classifier Using Custom Kernel” on page 18-199
- “Optimize a Cross-Validated SVM Classifier Using Bayesian Optimization” on page 18-205
- “Optimize an SVM Classifier Fit Using Bayesian Optimization” on page 10-66

## Input Arguments

### **Tbl** — Sample data

table

Sample data used to train the model, specified as a table. Each row of `Tbl` corresponds to one observation, and each column corresponds to one predictor variable. Optionally, `Tbl` can contain one additional column for the response variable. Multi-column variables and cell arrays other than cell arrays of character vectors are not allowed.

If `Tbl` contains the response variable, and you want to use all remaining variables in `Tbl` as predictors, then specify the response variable using `ResponseVarName`.

If `Tbl` contains the response variable, and you want to use only a subset of the remaining variables in `Tbl` as predictors, then specify a formula using `formula`.

If `Tbl` does not contain the response variable, then specify a response variable using `Y`. The length of response variable and the number of rows of `Tbl` must be equal.

Data Types: `table`

### **ResponseVarName** — Response variable name

name of variable in `Tbl`

Response variable name, specified as the name of a variable in `Tbl`.

You must specify `ResponseVarName` as a character vector. For example, if the response variable `Y` is stored as `Tbl.Y`, then specify it as `'Y'`. Otherwise, the software treats all columns of `Tbl`, including `Y`, as predictors when training the model.

The response variable must be a categorical or character array, logical or numeric vector, or cell array of character vectors. If `Y` is a character array, then each element must correspond to one row of the array.

It is good practice to specify the order of the classes using the `ClassNames` name-value pair argument.

Data Types: `char`

### **formula** — Explanatory model of response and subset of predictor variables

character vector

Explanatory model of the response and a subset of the predictor variables, specified as a character vector in the form of 'Y~X1+X2+X3'. In this form, Y represents the response variable, and X1, X2, and X3 represent the predictor variables. The variables must be variable names in Tbl (Tbl.Properties.VariableNames).

To specify a subset of variables in Tbl as predictors for training the model, use a formula. If you specify a formula, then the software does not use any variables in Tbl that do not appear in formula.

Data Types: char

### **y — Class labels**

categorical array | character array | logical vector | vector of numeric values | cell array of character vectors

Class labels to which the SVM model is trained, specified as a categorical or character array, logical or numeric vector, or cell array of character vectors.

- Y must contain at most two distinct classes. For multiclass learning, see `fitcecoc`.
- If Y is a character array, then each element must correspond to one row of the array.
- The length of Y and the number of rows of Tbl or X must be equal.
- It is good practice to specify the class order using the `ClassNames` name-value pair argument.

Data Types: char | cell | categorical | logical | single | double

### **x — Predictor data**

matrix of numeric values

Predictor data to which the SVM classifier is trained, specified as a matrix of numeric values.

Each row of X corresponds to one observation (also known as an instance or example), and each column corresponds to one predictor.

The length of Y and the number of rows of X must be equal.

To specify the names of the predictors in the order of their appearance in X, use the `PredictorNames` name-value pair argument.

Data Types: double | single

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

---

**Note** You cannot use any cross-validation name-value pair along with `OptimizeHyperparameters`. You can modify the cross-validation for `OptimizeHyperparameters` only by using the `HyperparameterOptimizationOptions` name-value pair.

---

Example: `'kFold', 10, 'Cost', [0 2; 1 0], 'ScoreTransform', 'sign'` specifies to perform 10-fold cross-validation, apply double the penalty to false positives compared to false negatives, and transform the scores using the sign function.

### Support Vector Machine Options

#### **BoxConstraint** — Box constraint

1 (default) | positive scalar

Box constraint on page 32-2127, specified as the comma-separated pair consisting of `'BoxConstraint'` and a positive scalar.

For one-class learning, the software always sets the box constraint to 1.

For more details on the relationships and algorithmic behavior of `BoxConstraint`, `Cost`, `Prior`, `Standardize`, and `Weights`, see “Algorithms” on page 32-2132.

Example: `'BoxConstraint', 100`

Data Types: `double` | `single`

#### **KernelFunction** — Kernel function

`'linear'` (default) | `'gaussian'` | `'rbf'` | `'polynomial'` | function name

Kernel function used to compute the elements of the Gram matrix on page 32-2127, specified as the comma-separated pair consisting of `'KernelFunction'` and a kernel function name. Suppose  $G(x_j, x_k)$  is element  $(j, k)$  of the Gram matrix, where  $x_j$  and  $x_k$  are

$p$ -dimensional vectors representing observations  $j$  and  $k$  in  $X$ . This table contains supported kernel function names and their functional forms.

Kernel Function Name	Description	Formula
'gaussian' or 'rbf'	Gaussian or Radial Basis Function (RBF) kernel, default for one-class learning	$G(x_j, x_k) = \exp\left(-\ x_j - x_k\ ^2\right)$
'linear'	Linear kernel, default for two-class learning	$G(x_j, x_k) = x_j'x_k$
'polynomial'	Polynomial kernel. Use 'PolynomialOrder', $q$ to specify a polynomial kernel of order $q$ .	$G(x_j, x_k) = (1 + x_j'x_k)^q$

You can set your own kernel function, for example, `kernel`, by setting 'KernelFunction', 'kernel'. `kernel` must have the following form:

```
function G = kernel(U,V)
```

where:

- $U$  is an  $m$ -by- $p$  matrix. Columns correspond to predictor variables and rows correspond to observations.
- $V$  is an  $n$ -by- $p$  matrix. Columns correspond to predictor variables and rows correspond to observations.
- $G$  is an  $m$ -by- $n$  Gram matrix on page 32-2127 of the rows of  $U$  and  $V$ .

`kernel.m` must be on the MATLAB path.

It is good practice to avoid using generic names for kernel functions. For example, call a sigmoid kernel function 'mysigmoid' rather than 'sigmoid'.

Example: 'KernelFunction', 'gaussian'

Data Types: char

#### **kernelScale** — Kernel scale parameter

1 (default) | 'auto' | positive scalar

Kernel scale parameter, specified as the comma-separated pair consisting of 'KernelScale' and 'auto' or a positive scalar. The software divides all elements of

the predictor matrix  $X$  by the value of `KernelScale`. Then, the software applies the appropriate kernel norm to compute the Gram matrix.

- If you specify `'auto'`, then the software selects an appropriate scale factor using a heuristic procedure. This heuristic procedure uses subsampling, so estimates can vary from one call to another. Therefore, to reproduce results, set a random number seed using `rng` before training.
- If you specify `KernelScale` and your own kernel function, for example, `kernel`, using `'KernelFunction', 'kernel'`, then the software throws an error. You must apply scaling within `kernel`.

Example: `'KernelScale', 'auto'`

Data Types: `double` | `single` | `char`

### **PolynomialOrder** — Polynomial kernel function order

3 (default) | positive integer

Polynomial kernel function order, specified as the comma-separated pair consisting of `'PolynomialOrder'` and a positive integer.

If you set `'PolynomialOrder'` and `KernelFunction` is not `'polynomial'`, then the software throws an error.

Example: `'PolynomialOrder', 2`

Data Types: `double` | `single`

### **KernelOffset** — Kernel offset parameter

nonnegative scalar

Kernel offset parameter, specified as the comma-separated pair consisting of `'KernelOffset'` and a nonnegative scalar.

The software adds `KernelOffset` to each element of the Gram matrix.

The defaults are:

- 0 if the solver is SMO (that is, you set `'Solver', 'SMO'`)
- 0.1 if the solver is ISDA (that is, you set `'Solver', 'ISDA'`)

Example: `'KernelOffset', 0`



Data Types: `double` | `single`

### **Standardize** — Flag to standardize predictor data

`false` (default) | `true`

Flag to standardize the predictor data, specified as the comma-separated pair consisting of `'Standardize'` and `true` (1) or `false` (0).

If you set `'Standardize', true`:

- The software centers and scales each column of the predictor data ( $X$ ) by the weighted column mean and standard deviation, respectively (for details on weighted standardizing, see “Algorithms” on page 32-2132). MATLAB does not standardize the data contained in the dummy variable columns generated for categorical predictors.
- The software trains the classifier using the standardized predictor matrix, but stores the unstandardized data in the classifier property `X`.

Example: `'Standardize', true`

Data Types: `logical`

### **Solver** — Optimization routine

`'ISDA'` | `'L1QP'` | `'SMO'`

Optimization routine, specified as the comma-separated pair consisting of `'Solver'` and a value in this table.

Value	Description
<code>'ISDA'</code>	Iterative Single Data Algorithm (see [30])
<code>'L1QP'</code>	Uses <code>quadprog</code> to implement $L1$ soft-margin minimization by quadratic programming. This option requires an Optimization Toolbox license. For more details, see “Quadratic Programming Definition” (Optimization Toolbox).
<code>'SMO'</code>	Sequential Minimal Optimization (see [17])

The defaults are:

- `'ISDA'` if you set `'OutlierFraction'` to a positive value and for two-class learning

- 'SMO' otherwise

Example: 'Solver', 'ISDA'

Data Types: char

### **Alpha** — Initial estimates of alpha coefficients

numeric vector of nonnegative values

Initial estimates of alpha coefficients, specified as the comma-separated pair consisting of 'Alpha' and a numeric vector of nonnegative values. The length of Alpha must be equal to the number of rows of X.

- Each element of Alpha corresponds to an observation in X.
- Alpha cannot contain any NaNs.
- If you specify Alpha and any one of the cross-validations name-value pair arguments ('CrossVal', 'CVPartition', 'Holdout', 'KFold', or 'Leaveout'), then the software returns an error.

If Y contains any missing values, then remove all rows of Y, X, and Alpha that correspond to the missing values. That is, enter:

```
idx = ~isundefined(categorical(Y));  
Y = Y(idx, :);  
X = X(idx, :);  
alpha = alpha(idx);
```

Then, pass Y, X, and alpha as the response, predictors, and initial alpha estimates, respectively.

The defaults are:

- `0.5*ones(size(X,1),1)` for one-class learning
- `zeros(size(X,1),1)` for two-class learning

Example: 'Alpha', `0.1*ones(size(X,1),1)`

Data Types: double | single

### **CacheSize** — Cache size

1000 (default) | 'maximal' | positive scalar

Cache size, specified as the comma-separated pair consisting of 'CacheSize' and 'maximal' or a positive scalar.

If CacheSize is 'maximal', then the software reserves enough memory to hold the entire  $n$ -by- $n$  Gram matrix on page 32-2127.

If CacheSize is a positive scalar, then the software reserves CacheSize megabytes of memory for training the model.

Example: 'CacheSize', 'maximal'

Data Types: double | char | single

### ClipAlphas — Flag to clip alpha coefficients

true (default) | false

Flag to clip alpha coefficients, specified as the comma-separated pair consisting of 'ClipAlphas' and either true or false.

Suppose that the alpha coefficient for observation  $j$  is  $\alpha_j$  and the box constraint of observation  $j$  is  $C_j$ ,  $j = 1, \dots, n$ .  $n$  is the training sample size.

Value	Description
true	At each iteration, if $\alpha_j$ is near 0 or near $C_j$ , then MATLAB sets $\alpha_j$ to 0 or to $C_j$ , respectively.
false	MATLAB does not change the alpha coefficients during optimization.

MATLAB stores the final values of  $\alpha$  in the Alpha property of the trained SVM model object.

ClipAlphas can affect SMO and ISDA convergence.

Example: 'ClipAlphas', false

Data Types: logical

### Nu — $\nu$ parameter for one-class learning

0.5 (default) | positive scalar

$\nu$  parameter for one-class learning on page 32-2128, specified as the comma-separated pair consisting of 'Nu' and a positive scalar. Nu must be greater than 0 and at most 1.

Set `Nu` to control the tradeoff between ensuring most training examples are in the positive class and minimizing the weights in the score function.

Example: `'Nu',0.25`

Data Types: `double` | `single`

**NumPrint** — Number of iterations between optimization diagnostic message output

1000 (default) | nonnegative integer

Number of iterations between optimization diagnostic message output, specified as the comma-separated pair consisting of `'NumPrint'` and a nonnegative integer.

If you use `'Verbose',1` and `'NumPrint',numprint`, then the software displays all optimization diagnostic messages from SMO and ISDA every `numprint` iterations in the Command Window.

Example: `'NumPrint',500`

Data Types: `double` | `single`

**OutlierFraction** — Expected proportion of outliers in training data

0 (default) | numeric scalar in the interval [0,1)

Expected proportion of outliers in the training data, specified as the comma-separated pair consisting of `'OutlierFraction'` and a numeric scalar in the interval [0,1).

If you set `'OutlierFraction',outlierfraction`, where `outlierfraction` is a value greater than 0, then:

- For two-class learning, the software implements robust learning. In other words, the software attempts to remove  $100 \times \text{outlierfraction}\%$  of the observations when the optimization algorithm converges. The removed observations correspond to gradients that are large in magnitude.
- For one-class learning, the software finds an appropriate bias term such that `outlierfraction` of the observations in the training set have negative scores.

Example: `'OutlierFraction',0.01`

Data Types: `double` | `single`

**RemoveDuplicates** — Flag to replace duplicate observations with single observations in training data`false (default) | true`

Flag to replace duplicate observations with single observations in the training data, specified as the comma-separated pair consisting of 'RemoveDuplicates' and `true` or `false`.

If `RemoveDuplicates` is `true`, then `fitcsvm` replaces duplicate observations in the training data with a single observation of the same value. The weight of the single observation is equal to the sum of the weights of the corresponding removed duplicates (see `Weights`).

---

**Tip** If your data set contains many duplicate observations, then specifying 'RemoveDuplicates', `true` can decrease convergence time considerably.

---

Data Types: `logical`

**verbose** — Verbosity level`0 (default) | 1 | 2`

Verbosity level, specified as the comma-separated pair consisting of 'Verbose' and either 0, 1, or 2. `Verbose` controls the amount of optimization information that the software displays in the Command Window and saves as a structure to `Mdl.ConvergenceInfo.History`.

This table summarizes the available verbosity level options.

Value	Description
0	The software does not display or save convergence information.
1	The software displays diagnostic messages and saves convergence criteria every <code>numprint</code> iterations, where <code>numprint</code> is the value of the name-value pair argument 'NumPrint'.

Value	Description
2	The software displays diagnostic messages and saves convergence criteria at every iteration.

Example: `'Verbose', 1`

Data Types: `double` | `single`

### Other Classification Options

#### **CategoricalPredictors** — Categorical predictors list

vector of positive integers | logical vector | character matrix | cell array of character vectors | `'all'`

Categorical predictors list, specified as the comma-separated pair consisting of `'CategoricalPredictors'` and one of these values:

Value	Description
Vector of positive integers	An entry in the vector is the index value corresponding to the column of the predictor data ( <code>X</code> or <code>Tbl</code> ) that contains a categorical variable.
Logical vector	A <code>true</code> entry means that the corresponding column of predictor data ( <code>X</code> or <code>Tbl</code> ) is a categorical variable.
Character matrix	Each row of the matrix is the name of a predictor variable. The names must match the entries in <code>PredictorNames</code> . Pad the names with extra blanks so each row of the character matrix has the same length.
Cell array of character vectors	Each element in the array is the name of a predictor variable. The names must match the entries in <code>PredictorNames</code> .
<code>'all'</code>	All predictors are categorical.

By default, if the predictor data is in a table (`Tbl`), `fitcsvm` assumes that a variable is categorical if it contains logical values, categorical values, or a cell array of character vectors. If the predictor data is a matrix (`X`), `fitcsvm` assumes all predictors are continuous. To identify any categorical predictors when the data is a matrix, use the `'CategoricalPredictors'` name-value pair argument.

Example: `'CategoricalPredictors', 'all'`

Data Types: `single` | `double` | `logical` | `char` | `cell`

### **ClassNames** — Names of classes to use for training

categorical array | cell array of character vectors | character array | logical vector | numeric vector

Names of classes to use for training, specified as the comma-separated pair consisting of 'ClassNames' and a categorical or character array, logical or numeric vector, or cell array of character vectors. ClassNames must be the same data type as Y.

If ClassNames is a character array, then each element must correspond to one *row* of the array.

Use ClassNames to:

- Order the classes during training.
- Specify the order of any input or output argument dimension that corresponds to the class order. For example, use ClassNames to specify the order of the dimensions of Cost or the column order of classification scores returned by predict.
- Select a subset of classes for training. For example, suppose that the set of all distinct class names in Y is {'a', 'b', 'c'}. To train the model using observations from classes 'a' and 'c' only, specify 'ClassNames', {'a', 'c'}.

The default is the set of all distinct class names in Y.

Example: 'ClassNames', {'b', 'g'}

Data Types: `categorical` | `char` | `logical` | `single` | `double` | `cell`

### **Cost** — Misclassification cost

square matrix | structure array

Misclassification cost, specified as the comma-separated pair consisting of 'Cost' and a square matrix or structure. If you specify:

- The square matrix Cost, then, if true class of an observation is  $i$ ,  $\text{Cost}(i, j)$  is the cost of classifying a point into class  $j$ . That is, rows correspond to the true classes and the columns correspond to predicted classes. To specify the class order for the corresponding rows and columns of Cost, also specify the ClassNames name-value pair argument.

- The structure `S`, then it must have two fields:
  - `S.ClassNames`, which contains the class names as a variable of the same data type as `Y`
  - `S.ClassificationCosts`, which contains the cost matrix with rows and columns ordered as in `S.ClassNames`

For two-class learning, if you specify a cost matrix, then the software updates the prior probabilities by incorporating the penalties described in the cost matrix. Consequently, the cost matrix resets to the default. For more details on the relationships and algorithmic behavior of `BoxConstraint`, `Cost`, `Prior`, `Standardize`, and `Weights`, see “Algorithms” on page 32-2132.

The defaults are:

- For one-class learning, `Cost = 0`.
- For two-class learning, `Cost(i,j) = 1` if `i ~= j`, and `Cost(i,j) = 0` if `i = j`.

Example: `'Cost', [0,1;2,0]`

Data Types: `double` | `single` | `struct`

### **PredictorNames** — Predictor variable names

cell array of unique character vectors

Predictor variable names, specified as the comma-separated pair consisting of `'PredictorNames'` and a cell array of unique character vectors. The functionality of `'PredictorNames'` depends on the way you supply the training data.

- If you supply `X` and `Y`, then you can use `'PredictorNames'` to give the predictor variables in `X` names.
  - The order of the names in `PredictorNames` must correspond to the column order of `X`. That is, `PredictorNames{1}` is the name of `X(:,1)`, `PredictorNames{2}` is the name of `X(:,2)`, and so on. Also, `size(X,2)` and `numel(PredictorNames)` must be equal.
  - By default, `PredictorNames` is `{'x1','x2',...}`.
- If you supply `Tbl`, then you can use `'PredictorNames'` to choose which predictor variables to use in training. That is, `fitcsvm` uses the predictor variables in `PredictorNames` and the response only in training.



- `PredictorNames` must be a subset of `Tbl.Properties.VariableNames` and cannot include the name of the response variable.
- By default, `PredictorNames` contains the names of all predictor variables.
- It is good practice to specify the predictors for training using one of `'PredictorNames'` or `formula` only.

```
Example: 'PredictorNames',
{'SepalLength', 'SepalWidth', 'PetalLength', 'PetalWidth'}
```

Data Types: cell

### **Prior — Prior probabilities**

'empirical' (default) | 'uniform' | numeric vector | structure array

Prior probabilities for each class, specified as the comma-separated pair consisting of `'Prior'` and a value in this table.

Value	Description
'empirical'	The class prior probabilities are the class relative frequencies in $Y$ .
'uniform'	All class prior probabilities are equal to $1/K$ , where $K$ is the number of classes.
numeric vector	Each element is a class prior probability. Order the elements according to <code>Mdl.ClassNames</code> or specify the order using the <code>ClassNames</code> name-value pair argument. The software normalizes the elements such that they sum to 1.
structure	A structure $S$ with two fields: <ul style="list-style-type: none"> <li>• <code>S.ClassNames</code> contains the class names as a variable of the same type as <math>Y</math>.</li> <li>• <code>S.ClassProbs</code> contains a vector of corresponding prior probabilities. The software normalizes the elements such that they sum to 1.</li> </ul>

For two-class learning, if you specify a cost matrix, then the software updates the prior probabilities by incorporating the penalties described in the cost matrix. For more details on the relationships and algorithmic behavior of `BoxConstraint`, `Cost`, `Prior`, `Standardize`, and `Weights`, see “Algorithms” on page 32-2132.

Example: `struct('ClassNames',  
{'setosa','versicolor','virginica'}), 'ClassProbs', 1:3)`

Data Types: `char` | `double` | `single` | `struct`

### **ResponseName** — Response variable name

'Y' (default) | character vector

Response variable name, specified as the comma-separated pair consisting of 'ResponseName' and a character vector.

- If you supply Y, then you can use 'ResponseName' to specify a name for the response variable.
- If you supply `ResponseVarName` or `formula`, then you cannot use 'ResponseName'.

Example: `'ResponseName', 'response'`

Data Types: `char`

### **ScoreTransform** — Score transformation

'none' (default) | character vector | function handle

Score transformation, specified as the comma-separated pair consisting of 'ScoreTransform' and either a character vector or a function handle.

This table summarizes the available character vectors.

Value	Description
'doublelogit'	$1/(1 + e^{-2x})$
'invlogit'	$\log(x / (1-x))$
'ismax'	Set the score for the class with the largest score to 1, and set the scores for all other classes to 0.
'logit'	$1/(1 + e^{-x})$
'none' or 'identity'	$x$ (no transformation)

Value	Description
'sign'	-1 for $x < 0$ 0 for $x = 0$ 1 for $x > 0$
'symmetric'	$2x - 1$
'symmetricismax'	Set the score for the class with the largest score to 1, and set the scores for all other classes to -1.
'symmetriclogit'	$2/(1 + e^{-x}) - 1$

For a MATLAB function or a function you define, use its function handle. The function handle must accept a matrix (the original scores) and return a matrix of the same size (the transformed scores).

Example: 'ScoreTransform', 'logit'

Data Types: char | function\_handle

### Weights — Observation weights

numeric vector | name of variable in Tbl

Observation weights, specified as the comma-separated pair consisting of 'Weights' and a numeric vector of positive values or name of a variable in Tbl. The software weighs the observations in each row of X or Tbl with the corresponding value in Weights. The size of Weights must equal the number of rows of X or Tbl.

If you specify the input data as a table Tbl, then Weights can be the name of a variable in Tbl that contains a numeric vector. In this case, you must specify Weights as a character vector. For example, if the weights vector W is stored as Tbl.W, then specify it as 'W'. Otherwise, the software treats all columns of Tbl, including W, as predictors or the response when training the model.

By default, Weights is ones ( $n, 1$ ), where  $n$  is the number of observations in X or Tbl.

The software normalizes Weights to sum up to the value of the prior probability in the respective class. For more details on the relationships and algorithmic behavior of BoxConstraint, Cost, Prior, Standardize, and Weights, see “Algorithms” on page 32-2132.

Data Types: double | single

### Cross-Validation Options

#### **CrossVal** — Flag to train cross-validated classifier

'off' (default) | 'on'

Flag to train a cross-validated classifier, specified as the comma-separated pair consisting of 'Crossval' and 'on' or 'off'.

If you specify 'on', then the software trains a cross-validated classifier with 10 folds.

You can override this cross-validation setting using one of the `CVPartition`, `Holdout`, `KFold`, or `Leaveout` name-value pair arguments. You can only use one cross-validation name-value pair argument at a time to create a cross-validated model.

Alternatively, cross-validate later by passing `Mdl` to `crossval`.

Example: 'Crossval', 'on'

Data Types: char

#### **CVPartition** — Cross-validation partition

[] (default) | `cvpartition` partition object

Cross-validation partition, specified as the comma-separated pair consisting of 'CVPartition' and a `cvpartition` partition object as created by `cvpartition`. The partition object specifies the type of cross-validation, and also the indexing for training and validation sets.

To create a cross-validated model, you can use one of these four name-value pair arguments only: `CVPartition`, `Holdout`, `KFold`, or `Leaveout`.

Example: Suppose you create a random partition for 5-fold cross-validation on 500 observations by using `cvp = cvpartition(500, 'KFold', 5)`. Then, you can specify the cross-validation partition for `fitcecoc` as 'CVPartition', `cvp`.

#### **Holdout** — Fraction of data for holdout validation

scalar value in the range (0,1)

Fraction of data used for holdout validation, specified as the comma-separated pair consisting of 'Holdout' and a scalar value in the range (0,1). If you specify 'Holdout',  $p$ , then the software:

- 1 Randomly reserves  $p \times 100\%$  of the data as validation data, and trains the model using the rest of the data
- 2 Stores the compact, trained model in the `Trained` property of the cross-validated model.

To create a cross-validated model, you can use one of these four name-value pair arguments only: `CVPartition`, `Holdout`, `KFold`, or `Leaveout`.

Example: `'Holdout', 0.1`

Data Types: `double` | `single`

### **KFold** — Number of folds

10 (default) | positive integer value greater than 1

Number of folds to use in a cross-validated model, specified as the comma-separated pair consisting of `'KFold'` and a positive integer value greater than 1. If you specify, e.g., `'KFold', k`, then the software:

- 1 Randomly partitions the data into  $k$  sets
- 2 For each set, reserves the set as validation data, and trains the model using the other  $k - 1$  sets
- 3 Stores the  $k$  compact, trained models in the cells of a  $k$ -by-1 cell vector in the `Trained` property of the cross-validated model.

To create a cross-validated model, you can use one of these four name-value pair arguments only: `CVPartition`, `Holdout`, `KFold`, or `Leaveout`.

Example: `'KFold', 5`

Data Types: `single` | `double`

### **Leaveout** — Leave-one-out cross-validation flag

'off' (default) | 'on'

Leave-one-out cross-validation flag, specified as the comma-separated pair consisting of `'Leaveout'` and `'on'` or `'off'`. If you specify `'Leaveout', 'on'`, then, for each of the  $n$  observations, where  $n$  is `size(Mdl.X, 1)`, the software:

- 1 Reserves the observation as validation data, and trains the model using the other  $n - 1$  observations

- 2 Stores the  $n$  compact, trained models in the cells of an  $n$ -by-1 cell vector in the `Trained` property of the cross-validated model.

To create a cross-validated model, you can use one of these four name-value pair arguments only: `CVPartition`, `Holdout`, `KFold`, or `Leaveout`.

Example: `'Leaveout', 'on'`

Data Types: `char`

### Convergence Controls

#### **DeltaGradientTolerance** — Tolerance for gradient difference

nonnegative scalar

Tolerance for the gradient difference between upper and lower violators obtained by Sequential Minimal Optimization (SMO) or Iterative Single Data Algorithm (ISDA), specified as the comma-separated pair consisting of `'DeltaGradientTolerance'` and a nonnegative scalar.

If `DeltaGradientTolerance` is 0, then the software does not use the tolerance for the gradient difference to check for optimization convergence.

The defaults are:

- $1e-3$  if the solver is SMO (for example, you set `'Solver', 'SMO'`)
- 0 if the solver is ISDA (for example, you set `'Solver', 'ISDA'`)

Example: `'DeltaGapTolerance', 1e-2`

Data Types: `double` | `single`

#### **GapTolerance** — Feasibility gap tolerance

0 (default) | nonnegative scalar

Feasibility gap tolerance obtained by SMO or ISDA, specified as the comma-separated pair consisting of `'GapTolerance'` and a nonnegative scalar.

If `GapTolerance` is 0, then the software does not use the feasibility gap tolerance to check for optimization convergence.

Example: `'GapTolerance', 1e-2`

Data Types: `double` | `single`

**IterationLimit** — Maximal number of numerical optimization iterations`1e6 (default) | positive integer`

Maximal number of numerical optimization iterations, specified as the comma-separated pair consisting of `'IterationLimit'` and a positive integer.

The software returns a trained model regardless of whether the optimization routine successfully converges. `Mdl.ConvergenceInfo` contains convergence information.

Example: `'IterationLimit',1e8`

Data Types: `double | single`

**KKTTolerance** — Karush-Kuhn-Tucker complementarity conditions violation tolerance`nonnegative scalar`

Karush-Kuhn-Tucker (KKT) complementarity conditions on page 32-2128 violation tolerance, specified as the comma-separated pair consisting of `'KKTTolerance'` and a nonnegative scalar.

If `KKTTolerance` is 0, then the software does not use the KKT complementarity conditions violation tolerance to check for optimization convergence.

The defaults are:

- 0 if the solver is SMO (for example, you set `'Solver','SMO'`)
- $1e-3$  if the solver is ISDA (for example, you set `'Solver','ISDA'`)

Example: `'KKTTolerance',1e-2`

Data Types: `double | single`

**ShrinkagePeriod** — Number of iterations between movement of observations from active to inactive set`0 (default) | nonnegative integer`

Number of iterations between the movement of observations from the active to inactive set, specified as the comma-separated pair consisting of `'ShrinkagePeriod'` and a nonnegative integer.

If you set `'ShrinkagePeriod',0`, then the software does not shrink the active set.

Example: `'ShrinkagePeriod',1000`

Data Types: `double` | `single`

### Hyperparameter Optimization

#### **OptimizeHyperparameters** — Parameters to optimize

'none' (default) | 'auto' | 'all' | cell array of eligible parameter names | vector of `optimizableVariable` objects

Parameters to optimize, specified as:

- 'none' — Do not optimize.
- 'auto' — Use {'BoxConstraint', 'KernelScale'}
- 'all' — Optimize all eligible parameters.
- Cell array of eligible parameter names
- Vector of `optimizableVariable` objects, typically the output of `hyperparameters`

The optimization attempts to minimize the cross-validation loss (error) for `fitcsvm` by varying the parameters. For information about cross-validation loss (albeit in a different context), see “Classification Loss” on page 32-3250. To control the cross-validation type and other aspects of the optimization, use the `HyperparameterOptimizationOptions` name-value pair.

---

**Note** `OptimizeHyperparameters` values override any values you set using other name-value pairs. For example, setting `OptimizeHyperparameters` to 'auto' causes the 'auto' values to apply.

---

The eligible parameters for `fitcsvm` are:

- `BoxConstraint` — `fitcsvm` searches among positive values, by default log-scaled in the range `[1e-3, 1e3]`.
- `KernelScale` — `fitcsvm` searches among positive values, by default log-scaled in the range `[1e-3, 1e3]`.
- `KernelFunction` — `fitcsvm` searches among 'gaussian', 'linear', and 'polynomial'.
- `PolynomialOrder` — `fitcsvm` searches among integers in the range `[2, 4]`.
- `Standardize` — `fitcsvm` searches among 'true' and 'false'.



Set nondefault parameters by passing a vector of `optimizableVariable` objects that have nondefault values. For example,

```
load fisheriris
params = hyperparameters('fitcsvm',meas,species);
params(1).Range = [1e-4,1e6];
```

Pass `params` as the value of `OptimizeHyperparameters`.

By default, iterative display appears at the command line, and plots appear according to the number of hyperparameters in the optimization. For the optimization and plots, the objective function is  $\log(1 + \text{cross-validation loss})$  for regression, and the misclassification rate for classification. To control the iterative display, set the `HyperparameterOptimizationOptions` name-value pair, `Verbose` field. To control the plots, set the `HyperparameterOptimizationOptions` name-value pair, `ShowPlots` field.

For an example, see “Optimize SVM Classifier” on page 32-2097.

Example: 'auto'

Data Types: char | cell

### **HyperparameterOptimizationOptions — Options for optimization structure**

Options for optimization, specified as a structure. Modifies the effect of the `OptimizeHyperparameters` name-value pair. All fields in the structure are optional.

Field Name	Values	Default
Optimizer	<ul style="list-style-type: none"> <li>'bayesopt' — Use Bayesian optimization. Internally, this setting calls bayesopt.</li> <li>'gridsearch' — Use grid search with NumGridDivisions values per dimension.</li> <li>'randomsearch' — Search at random among MaxObjectiveEvaluations points.</li> </ul> <p>'gridsearch' searches in a random order, using uniform sampling without replacement from the grid. After optimization, you can get a table in grid order by using the command sortrows (Mdl.ParameterOptimizationResults).</p>	'bayesopt'
AcquisitionFunctionName	<ul style="list-style-type: none"> <li>'expected-improvement-per-second-plus'</li> <li>'expected-improvement'</li> <li>'expected-improvement-plus'</li> <li>'expected-improvement-per-second'</li> <li>'lower-confidence-bound'</li> <li>'probability-of-improvement'</li> </ul> <p>For details, see the bayesopt AcquisitionFunctionName name-value pair, or “Acquisition Function Types” on page 10-4.</p>	'expected-improvement-per-second-plus'
MaxObjectiveEvaluations	Maximum number of objective function evaluations.	30 for 'bayesopt' or 'randomsearch', and the entire grid for 'gridsearch'

Field Name	Values	Default
MaxTime	Time limit, specified as a positive real. The time limit is in seconds, as measured by <code>tic</code> and <code>toc</code> . Run time can exceed <code>MaxTime</code> because <code>MaxTime</code> does not interrupt function evaluations.	Inf
NumGridDivisions	For 'gridsearch', the number of values in each dimension. Can be a vector of positive integers giving the number of values for each dimension, or a scalar that applies to all dimensions. Ignored for categorical variables.	10
ShowPlots	Logical value indicating whether to show plots. If <code>true</code> , plots the best objective function value against iteration number. If there are one or two optimization parameters, and if <code>Optimizer</code> is 'bayesopt', then <code>ShowPlots</code> also plots a model of the objective function against the parameters.	true
SaveIntermediateResults	Logical value indicating whether to save results when <code>Optimizer</code> is 'bayesopt'. If <code>true</code> , overwrites a workspace variable named 'BayesoptResults' at each iteration. The variable is a <code>BayesianOptimization</code> object.	false
Verbose	Display to the command line. <ul style="list-style-type: none"> <li>• 0 — No iterative display</li> <li>• 1 — Iterative display</li> <li>• 2 — Iterative display with extra information</li> </ul> For details, see the <code>bayesopt</code> <code>Verbose</code> name-value pair.	1

Field Name	Values	Default
UseParallel	Logical value indicating whether to run Bayesian optimization in parallel, which requires Parallel Computing Toolbox. For details, see “Parallel Bayesian Optimization” on page 10-9.	false
Repartition	Logical value indicating whether to repartition the cross-validation at every iteration. If <code>false</code> , the optimizer uses a single partition for the optimization.  <code>true</code> usually gives the most robust results because this setting takes partitioning noise into account. However, for good results, <code>true</code> requires at least twice as many function evaluations.	false
Use no more than one of the following three field names.		
CVPartition	A <code>cvpartition</code> object, as created by <code>cvpartition</code>	Kfold = 5
Holdout	A scalar in the range (0, 1) representing the holdout fraction.	
Kfold	An integer greater than 1.	

Example: `struct('MaxObjectiveEvaluations', 60)`

Data Types: `struct`

## Output Arguments

### **mdl** — Trained SVM classification model

ClassificationSVM model object | ClassificationPartitionedModel cross-validated model object

Trained SVM classification model, returned as a `ClassificationSVM` model object or `ClassificationPartitionedModel` cross-validated model object.

If you set any of the name-value pair arguments `KFold`, `Holdout`, `Leaveout`, `CrossVal`, or `CVPartition`, then `Mdl` is a `ClassificationPartitionedModel` cross-validated classifier. Otherwise, `Mdl` is a `ClassificationSVM` classifier.

To reference properties of `Mdl`, use dot notation. For example, enter `Mdl.Alpha` in the Command Window to display the trained Lagrange multipliers.

## Limitations

- `fitcsvm` trains SVM classifiers for one- or two-class learning applications. To train SVM classifiers using data with more than two classes, use `fitcecoc`.
- `fitcsvm` supports low- through moderate-dimensional data sets. For high-dimensional data set, use `fitclinear` instead.

## Definitions

### Box Constraint

A parameter that controls the maximum penalty imposed on margin-violating observations, and aids in preventing overfitting (regularization).

If you increase the box constraint, then the SVM classifier assigns fewer support vectors. However, increasing the box constraint can lead to longer training times.

### Gram Matrix

The Gram matrix of a set of  $n$  vectors  $\{x_1, \dots, x_n; x_j \in R^p\}$  is an  $n$ -by- $n$  matrix with element  $(j, k)$  defined as  $G(x_j, x_k) = \langle \phi(x_j), \phi(x_k) \rangle$ , an inner product of the transformed predictors using the kernel function  $\phi$ .

For nonlinear SVM, the algorithm forms a Gram matrix using the predictor matrix columns. The dual formalization replaces the inner product of the predictors with corresponding elements of the resulting Gram matrix (called the “kernel trick”). Subsequently, nonlinear SVM operates in the transformed predictor space to find a separating hyperplane.

## Karush-Kuhn-Tucker Complementarity Conditions

KKT complementarity conditions are optimization constraints required for optimal nonlinear programming solutions.

In SVM, the KKT complementarity conditions are

$$\begin{cases} \alpha_j [y_j f(x_j) - 1 + \xi_j] = 0 \\ \xi_j (C - \alpha_j) = 0 \end{cases}$$

for all  $j = 1, \dots, n$ , where  $f(x_j) = \phi(x_j)' \beta + b$ ,  $\phi$  is a kernel function (see Gram matrix on page 32-2127), and  $\xi_j$  is a slack variable. If the classes are perfectly separable, then  $\xi_j = 0$  for all  $j = 1, \dots, n$ .

## One-Class Learning

One-class learning, or unsupervised SVM, aims at separating data from the origin in the high-dimensional, predictor space (not the original predictor space), and is an algorithm used for outlier detection.

The algorithm resembles that of SVM for binary classification on page 32-2129. The objective is to minimize dual expression

$$0.5 \sum_{jk} \alpha_j \alpha_k G(x_j, x_k)$$

with respect to  $\alpha_1, \dots, \alpha_n$ , subject to

$$\sum \alpha_j = nv$$

and  $0 \leq \alpha_j \leq 1$  for all  $j = 1, \dots, n$ .  $G(x_j, x_k)$  is element  $(j, k)$  of the Gram matrix on page 32-2127.

A small value of  $v$  leads to fewer support vectors, and, therefore, a smooth, crude decision boundary. A large value of  $v$  leads to more support vectors, and therefore, a curvy, flexible decision boundary. The optimal value of  $v$  should be large enough to capture the data complexity and small enough to avoid overtraining. Also,  $0 < v \leq 1$ .

For more details, see [5].

## Support Vector

Support vectors are observations corresponding to strictly positive estimates of  $a_1, \dots, a_n$ .

SVM classifiers that yield fewer support vectors for a given training set are more desirable.

## Support Vector Machines for Binary Classification

The SVM binary classification algorithm searches for an optimal hyperplane that separates the data into two classes. For separable classes, the optimal hyperplane maximizes a margin (space that does not contain any observations) surrounding itself, which creates boundaries for the positive and negative classes. For inseparable classes, the objective is the same, but the algorithm imposes a penalty on the length of the margin for every observation that is on the wrong side of its class boundary.

The linear SVM score function is

$$f(x) = x'\beta + b,$$

where:

- $x$  is an observation (corresponding to a row of  $X$ ).
- The vector  $\beta$  contains the coefficients that define an orthogonal vector to the hyperplane (corresponding to `Mdl.Beta`). For separable data, the optimal margin length is  $2 / \|\beta\|$ .
- $b$  is the bias term (corresponding to `Mdl.Bias`).

The root of  $f(x)$  for particular coefficients defines a hyperplane. For a particular hyperplane,  $f(z)$  is the distance from point  $z$  to the hyperplane.

The algorithm searches for the maximum margin length, while keeping observations in the positive ( $y = 1$ ) and negative ( $y = -1$ ) classes separate. Therefore:

- For separable classes, the objective is to minimize  $\|\beta\|$  with respect to the  $\beta$  and  $b$  subject to  $y_j f(x_j) \geq 1$ , for all  $j = 1, \dots, n$ . This is the primal formalization for separable classes.
- For inseparable classes, the algorithm uses slack variables ( $\xi_j$ ) to penalize the objective function for observations that cross the margin boundary for their class.  $\xi_j =$

0 for observations that do not cross the margin boundary for their class, otherwise  $\xi_j \geq 0$ .

The objective is to minimize  $0.5\|\beta\|^2 + C\sum\xi_j$  with respect to the  $\beta$ ,  $b$ , and  $\xi_j$  subject to  $y_j f(x_j) \geq 1 - \xi_j$  and  $\xi_j \geq 0$  for all  $j = 1, \dots, n$ , and for a positive scalar box constraint on page 32-2127 *C*. This is the primal formalization for inseparable classes.

The algorithm uses the Lagrange multipliers method to optimize the objective. This introduces  $n$  coefficients  $\alpha_1, \dots, \alpha_n$  (corresponding to `Mdl.Alpha`). The dual formalizations for linear SVM are:

- For separable classes, minimize

$$0.5 \sum_{j=1}^n \sum_{k=1}^n \alpha_j \alpha_k y_j y_k x_j' x_k - \sum_{j=1}^n \alpha_j$$

with respect to  $\alpha_1, \dots, \alpha_n$ , subject to  $\sum \alpha_j y_j = 0$ ,  $\alpha_j \geq 0$  for all  $j = 1, \dots, n$ , and Karush-Kuhn-Tucker (KKT) complementarity conditions on page 32-2128.

- For inseparable classes, the objective is the same as for separable classes, except for the additional condition  $0 \leq \alpha_j \leq C$  for all  $j = 1, \dots, n$ .

The resulting score function is

$$\hat{f}(x) = \sum_{j=1}^n \hat{\alpha}_j y_j x' x_j + \hat{b}.$$

$\hat{b}$  is the estimate of the bias and  $\hat{\alpha}_j$  is the  $j$ th estimate of the vector  $\hat{\alpha}$ ,  $j = 1, \dots, n$ . Written this way, the score function is free of the estimate of  $\beta$  as a result of the primal formalization.

The SVM algorithm classifies a new observation,  $z$  using  $\text{sign}(\hat{f}(z))$ .

In some cases, there is a nonlinear boundary separating the classes. Nonlinear SVM works in a transformed predictor space to find an optimal, separating hyperplane.

The dual formalization for nonlinear SVM is



$$0.5 \sum_{j=1}^n \sum_{k=1}^n \alpha_j \alpha_k y_j y_k G(x_j, x_k) - \sum_{j=1}^n \alpha_j$$

with respect to  $\alpha_1, \dots, \alpha_n$ , subject to  $\sum \alpha_j y_j = 0$ ,  $0 \leq \alpha_j \leq C$  for all  $j = 1, \dots, n$ , and the KKT complementarity conditions.  $G(x_k, x_j)$  are elements of the Gram matrix on page 32-2127. The resulting score function is

$$\hat{f}(x) = \sum_{j=1}^n \hat{\alpha}_j y_j G(x, x_j) + \hat{b}.$$

For more details, see Understanding Support Vector Machines on page 18-186, [1], and [3].

## Tips

- Unless your data set is large, always try to standardize the predictors (see Standardize). Standardization makes predictors insensitive to the scales on which they are measured.
- It is good practice to cross-validate using the `KFold` name-value pair argument. The cross-validation results determine how well the SVM classifier generalizes.
- For one-class learning:
  - The default setting for the name-value pair argument `Alpha` can lead to long training times. To speed up training, set `Alpha` to a vector mostly composed of 0s.
  - Set the name-value pair argument `Nu` to a value closer to 0 to yield fewer support vectors, and, therefore, a smoother, but crude decision boundary.
- Sparsity in support vectors is a desirable property of an SVM classifier. To decrease the number of support vectors, set `BoxConstraint` to a large value. This action increases the training time.
- For optimal training time, set `CacheSize` as high as the memory limit on your computer allows.
- If you expect many fewer support vectors than observations in the training set, then you can significantly speed up convergence by shrinking the active set using the name-value pair argument `'ShrinkagePeriod'`. It is good practice to use `'ShrinkagePeriod', 1000`.

- Duplicate observations that are far from the decision boundary do not affect convergence. However, just a few duplicate observations that occur near the decision boundary can slow down convergence considerably. To speed up convergence, specify `'RemoveDuplicates', true` if:
  - Your data set contains many duplicate observations.
  - You suspect that a few duplicate observations fall near the decision boundary.

However, to maintain the original data set during training, `fitcsvm` must temporarily store separate data sets: the original and one without the duplicate observations. Therefore, if you specify `true` for data sets containing few duplicates, then `fitcsvm` consumes close to double the memory of the original data.

- After training a `ClassificationSVM` model object by using `fitcsvm`, use the function `predict` and the trained model object to generate C code that predicts labels for new data. For details, see “Code Generation” on page 32-0 .

## Algorithms

- For the mathematical formulation of the SVM binary classification algorithm, see “Support Vector Machines for Binary Classification” on page 32-2129 and “Understanding Support Vector Machines” on page 18-186.
- NaN, `<undefined>`, and empty character vector ( `' '` ) values indicate missing values. `fitcsvm` removes entire rows of data corresponding to a missing response. When computing total weights (see the next bullets), `fitcsvm` ignores any weight corresponding to an observation with at least one missing predictor. This action can lead to unbalanced prior probabilities in balanced-class problems. Consequently, observation box constraints might not equal `BoxConstraint`.
- `fitcsvm` removes observations that have zero weight or prior probability.
- For two-class learning, if you specify the cost matrix `C` (see `Cost`), then the software updates the class prior probabilities  $p$  (see `Prior`) to  $p_c$  by incorporating the penalties described in `C` .

Specifically, `fitcsvm`:

- 1 Computes  $p_c^* = p' C$ .

- 2 Normalizes  $p_c^*$  so that the updated prior probabilities sum 1:

$$p_c = \frac{1}{\sum_{j=1}^K p_{c,j}^*} p_c^*.$$

$K$  is the number of classes.

- 3 Resets the cost matrix to the default:

$$C = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}.$$

- 4 Removes observations from the training data corresponding to classes with zero prior probability.
- For two-class learning, `fitcsvm` normalizes all observation weights (see `Weights`) to sum to 1. Then, renormalizes the normalized weights to sum up to the updated, prior probability of the class to which the observation belongs. That is, the total weight for observation  $j$  in class  $k$  is

$$w_j^* = \frac{w_j}{\sum_{\forall j \in \text{Class } k} w_j} p_{c,k}.$$

$w_j$  is the normalized weight for observation  $j$ ;  $p_{c,k}$  is the updated prior probability of class  $k$  (see previous bullet).

- For two-class learning, `fitcsvm` assigns a box constraint to each observation in the training data. The formula for the box constraint of observation  $j$  is

$$C_j = nC_0w_j^*.$$

$n$  is the training sample size,  $C_0$  is the initial box constraint (see `BoxConstraint`),

and  $w_j^*$  is the total weight of observation  $j$  (see previous bullet).

- If you set `'Standardize', true` and any of `'Cost'`, `'Prior'`, or `'Weights'`, then `fitcsvm` standardizes the predictors using their corresponding weighted means and weighted standard deviations. That is, `fitcsvm` standardizes predictor  $j$  ( $x_j$ ) using

$$x_j^* = \frac{x_j - \mu_j^*}{\sigma_j^*}.$$

•

$$\mu_j^* = \frac{1}{\sum_k w_k^*} \sum_k w_k^* x_{jk}.$$

- $x_{jk}$  is observation  $k$  (row) of predictor  $j$  (column).

•

$$(\sigma_j^*)^2 = \frac{v_1}{v_1^2 - v_2} \sum_k w_k^* (x_{jk} - \mu_j^*)^2.$$

•

$$v_1 = \sum_j w_j^*.$$

•

$$v_2 = \sum_j (w_j^*)^2.$$

- Let `p` be the proportion of outliers that you expect in the training data. If you set `'OutlierFraction', p`, then:
  - For one-class learning, the software trains the bias term such that 100p% of the observations in the training data have negative scores.
  - The software implements robust learning for two-class learning. In other words, the software attempts to remove 100p% of the observations when the optimization algorithm converges. The removed observations correspond to gradients that are large in magnitude.
- If your predictor data contains categorical variables, then the software generally uses full dummy encoding for these variables. The software creates one dummy variable for each level of each categorical variable.
  - The `PredictorNames` property stores one element for each of the original predictor variable names. For example, assume that there are three predictors, one of which is a categorical variable with three levels. Then `PredictorNames` is a 1-by-3 cell array of character vectors containing the original names of the predictor variables.
  - The `ExpandedPredictorNames` property stores one element for each of the predictor variables, including the dummy variables. For example, assume that there are three predictors, one of which is a categorical variable with three levels. Then `ExpandedPredictorNames` is a 1-by-5 cell array of character vectors containing the names of the predictor variables and the new dummy variables.

- Similarly, the `Beta` property stores one beta coefficient for each predictor, including the dummy variables.
- The `SupportVectors` property stores the predictor values for the support vectors, including the dummy variables. For example, assume that there are  $m$  support vectors and three predictors, one of which is a categorical variable with three levels. Then `SupportVectors` is an  $n$ -by-5 matrix.
- The `X` property stores the training data as originally input. It does not include the dummy variables. When the input is a table, `X` contains only the columns used as predictors.
- For predictors specified in a table, if any of the variables contain ordered (ordinal) categories, the software uses ordinal encoding for these variables.
  - For a variable having  $k$  ordered levels, the software creates  $k - 1$  dummy variables. The  $j$ th dummy variable is -1 for levels up to  $j$ , and +1 for levels  $j + 1$  through  $k$ .
  - The names of the dummy variables stored in the `ExpandedPredictorNames` property indicate the first level with the value +1. The software stores  $k - 1$  additional predictor names for the dummy variables, including the names of levels 2, 3, ...,  $k$ .
- All solvers implement  $L1$  soft-margin minimization.
- `fitcsvm` and `svmtrain` use, among other algorithms, SMO for optimization. The software implements SMO differently between the two functions, but numerical studies show that there is sensible agreement in the results.
- For one-class learning, the software estimates the Lagrange multipliers,  $\alpha_1, \dots, \alpha_n$ , such that

$$\sum_{j=1}^n \alpha_j = nv.$$

## References

- [1] Christianini, N., and J. C. Shawe-Taylor. *An Introduction to Support Vector Machines and Other Kernel-Based Learning Methods*. Cambridge, UK: Cambridge University Press, 2000.
- [2] Fan, R.-E., P.-H. Chen, and C.-J. Lin. "Working set selection using second order information for training support vector machines." *Journal of Machine Learning Research*, Vol 6, 2005, pp. 1889–1918.

- [3] Hastie, T., R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*, Second Edition. NY: Springer, 2008.
- [4] Kecman V., T. -M. Huang, and M. Vogt. “Iterative Single Data Algorithm for Training Kernel Machines from Huge Data Sets: Theory and Performance.” In *Support Vector Machines: Theory and Applications*. Edited by Lipo Wang, 255–274. Berlin: Springer-Verlag, 2005.
- [5] Scholkopf, B., J. C. Platt, J. C. Shawe-Taylor, A. J. Smola, and R. C. Williamson. “Estimating the Support of a High-Dimensional Distribution.” *Neural Comput.*, Vol. 13, Number 7, 2001, pp. 1443–1471.
- [6] Scholkopf, B., and A. Smola. *Learning with Kernels: Support Vector Machines, Regularization, Optimization and Beyond, Adaptive Computation and Machine Learning*. Cambridge, MA: The MIT Press, 2002.

## See Also

`ClassificationPartitionedModel` | `ClassificationSVM` |  
`CompactClassificationSVM` | `fitSVMPosterior` | `fitcecoc` | `fitclinear` |  
`predict` | `quadprog` | `rng`

## Topics

- “Train SVM Classifiers Using a Gaussian Kernel” on page 18-195
- “Train SVM Classifier Using Custom Kernel” on page 18-199
- “Optimize a Cross-Validated SVM Classifier Using Bayesian Optimization” on page 18-205
- “Optimize an SVM Classifier Fit Using Bayesian Optimization” on page 10-66
- “Understanding Support Vector Machines” on page 18-186

## Introduced in R2014a

# fitctree

Fit binary classification decision tree for multiclass classification

## Syntax

```
tree = fitctree(Tbl,ResponseVarName)
tree = fitctree(Tbl,formula)
tree = fitctree(Tbl,Y)

tree = fitctree(X,Y)

tree = fitctree( ____,Name,Value)
```

## Description

`tree = fitctree(Tbl,ResponseVarName)` returns a fitted binary classification decision tree based on the input variables (also known as predictors, features, or attributes) contained in the table `Tbl` and output (response or labels) contained in `ResponseVarName`. The returned binary tree splits branching nodes based on the values of a column of `Tbl`.

`tree = fitctree(Tbl,formula)` returns a fitted binary classification decision tree based on the input variables contained in the table `Tbl`. `formula` is an explanatory model of the response and a subset of predictor variables in `Tbl` used to fit `tree`.

`tree = fitctree(Tbl,Y)` returns a fitted binary classification decision tree based on the input variables contained in the table `Tbl` and output in vector `Y`.

`tree = fitctree(X,Y)` returns a fitted binary classification decision tree based on the input variables contained in matrix `X` and output `Y`. The returned binary tree splits branching nodes based on the values of a column of `X`.

`tree = fitctree( ____,Name,Value)` fits a tree with additional options specified by one or more name-value pair arguments, using any of the previous syntaxes. For example, you can specify the algorithm used to find the best split on a categorical

predictor, grow a cross-validated tree, or hold out a fraction of the input data for validation.

## Examples

### Grow a Classification Tree

Grow a classification tree using the `ionosphere` data set.

```
load ionosphere
tc = fitctree(X,Y)

tc =

ClassificationTree
    ResponseName: 'Y'
    CategoricalPredictors: []
    ClassNames: {'b' 'g'}
    ScoreTransform: 'none'
    NumObservations: 351
```

### Control Tree Depth

You can control the depth of the trees using the `MaxNumSplits`, `MinLeafSize`, or `MinParentSize` name-value pair parameters. `fitctree` grows deep decision trees by default. You can grow shallower trees to reduce model complexity or computation time.

Load the `ionosphere` data set.

```
load ionosphere
```

The default values of the tree depth controllers for growing classification trees are:

- `n - 1` for `MaxNumSplits`. `n` is the training sample size.
- `1` for `MinLeafSize`.



- 10 for MinParentSize.

These default values tend to grow deep trees for large training sample sizes.

Train a classification tree using the default values for tree depth control. Cross validate the model using 10-fold cross validation.

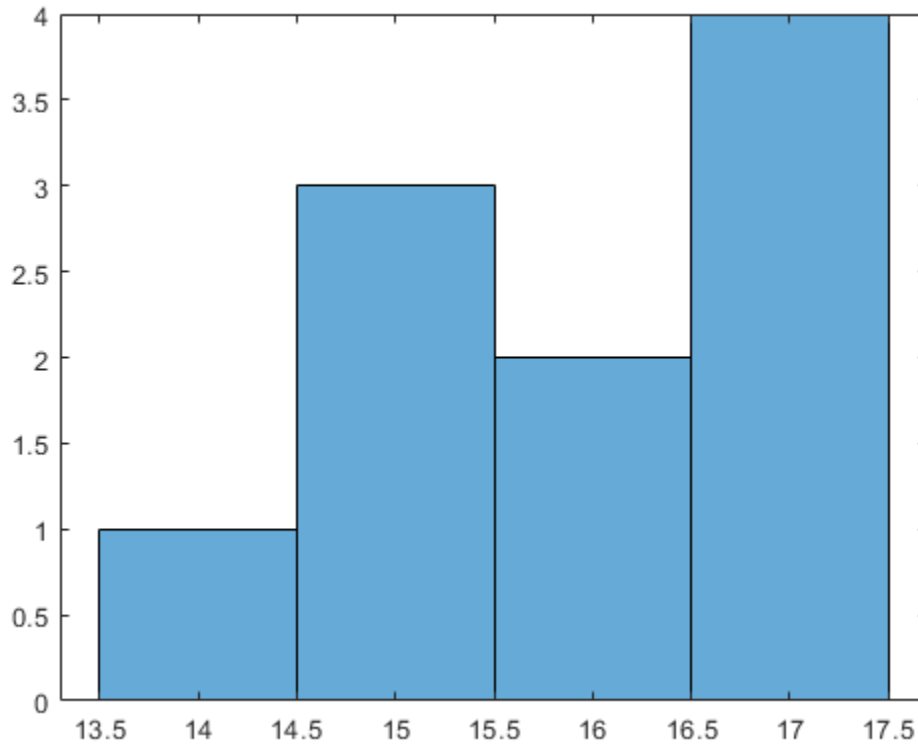
```
rng(1); % For reproducibility
MdlDefault = fitctree(X,Y, 'CrossVal', 'on');
```

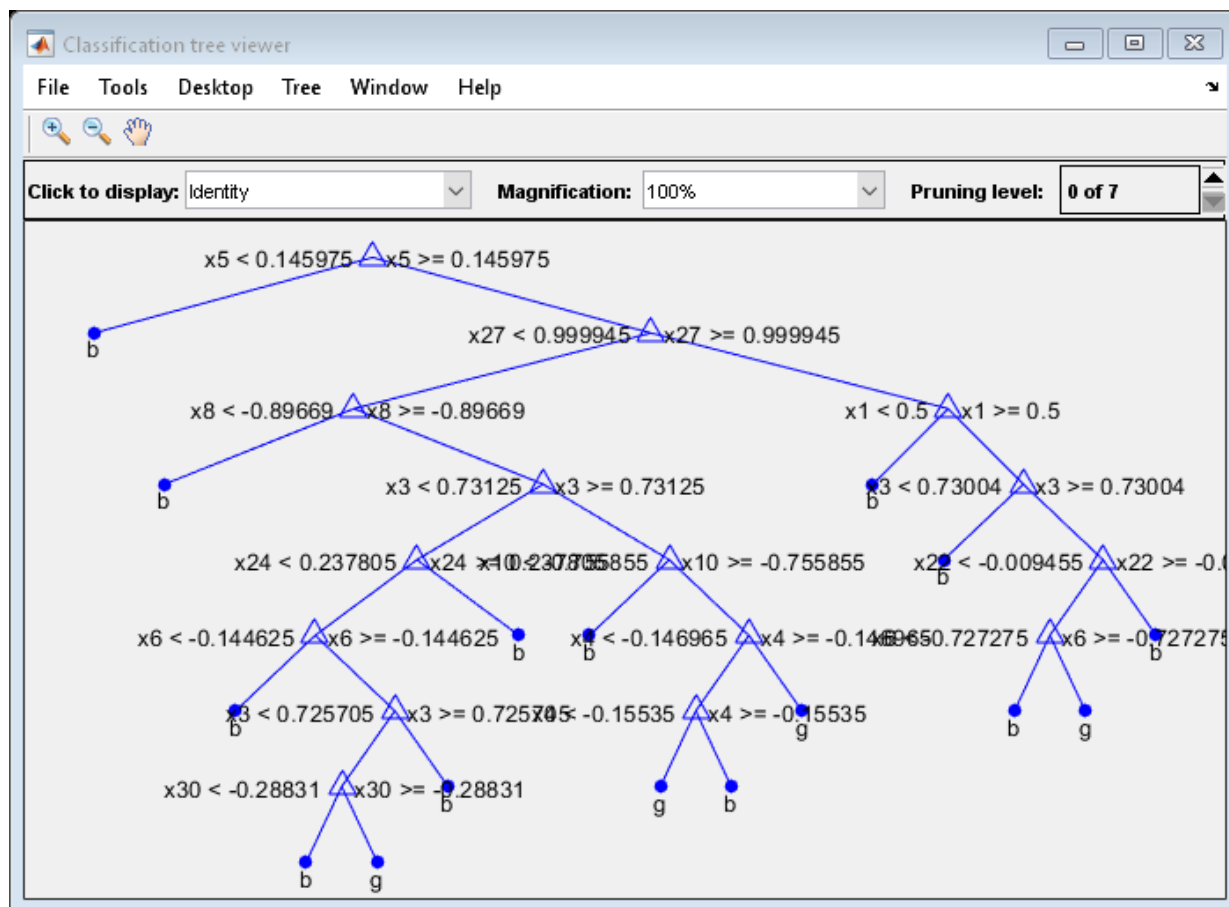
Draw a histogram of the number of imposed splits on the trees. Also, view one of the trees.

```
numBranches = @(x) sum(x.IsBranch);
mdlDefaultNumSplits = cellfun(numBranches, MdlDefault.Trained);

figure;
histogram(mdlDefaultNumSplits)

view(MdlDefault.Trained{1}, 'Mode', 'graph')
```

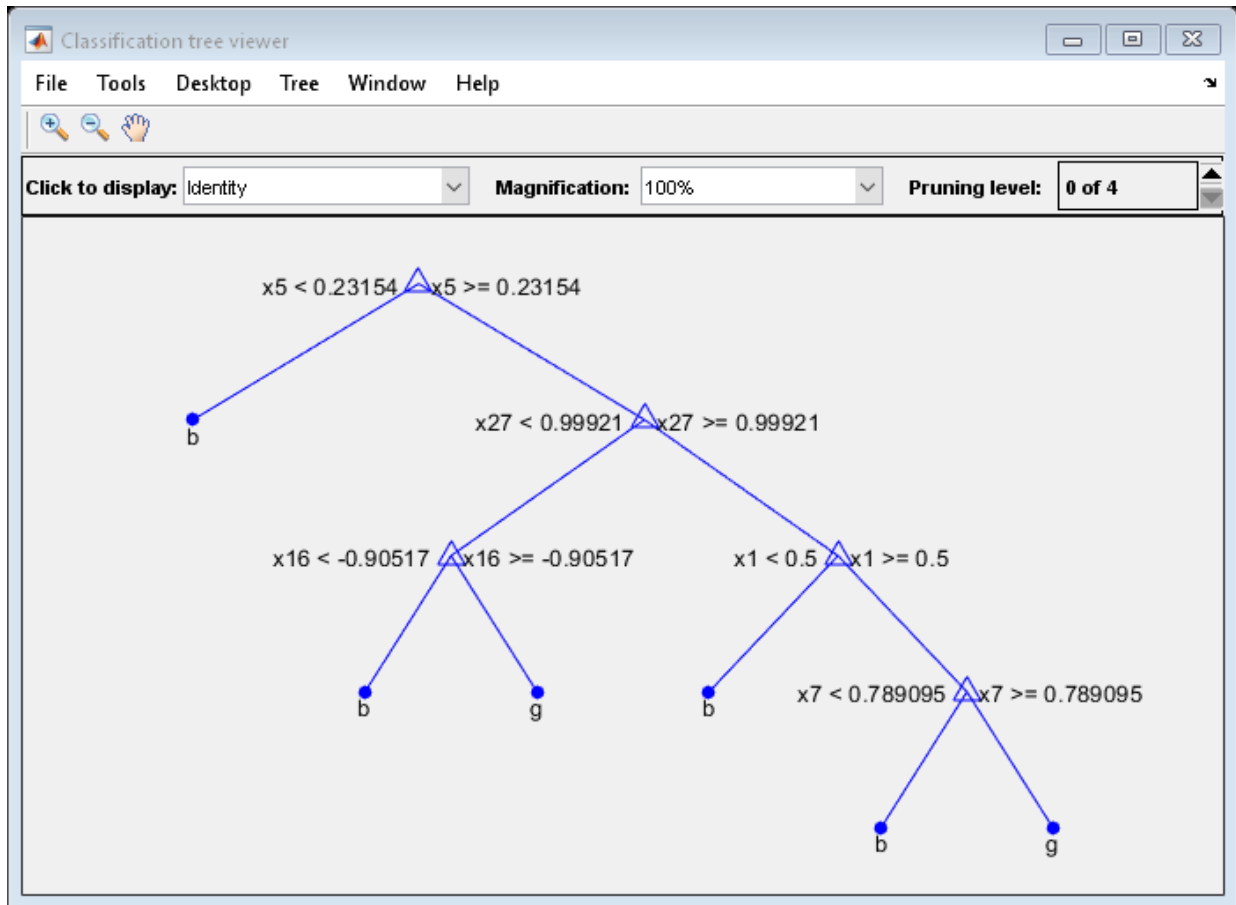




The average number of splits is around 15.

Suppose that you want a classification tree that is not as complex (deep) as the ones trained using the default number of splits. Train another classification tree, but set the maximum number of splits at 7, which is about half the mean number of splits from the default classification tree. Cross validate the model using 10-fold cross validation.

```
Mdl7 = fitctree(X,Y,'MaxNumSplits',7,'CrossVal','on');
view(Mdl7.Trained{1},'Mode','graph')
```



Compare the cross validation classification errors of the models.

```
classErrorDefault = kfoldLoss(MdlDefault)
classError7 = kfoldLoss(Mdl7)
```

```
classErrorDefault =
```

```
    0.1140
```

```
classError7 =
```

0.1254

Mdl17 is much less complex and performs only slightly worse than MdlDefault.

## Optimize Classification Tree

This example shows how to optimize hyperparameters automatically using `fitctree`. The example uses Fisher's iris data.

Load Fisher's iris data.

```
load fisheriris
```

Optimize the cross-validation loss of the classifier, using the data in `meas` to predict the response in `species`.

```
X = meas;
Y = species;
Mdl = fitctree(X,Y,'OptimizeHyperparameters','auto')
```

```
=====
| Iter | Eval  | Objective  | Objective  | BestSoFar  | BestSoFar  | MinLeafSize
|      | result|            | runtime    | (observed) | (estim.)   |
|=====|=====|=====|=====|=====|=====|=====
|  1  | Best  | 0.066667  | 0.41014   | 0.066667  | 0.066667  | 31
|  2  | Accept| 0.066667  | 0.20374   | 0.066667  | 0.066667  | 12
|  3  | Best  | 0.04       | 0.2963    | 0.04       | 0.040003  | 2
|  4  | Accept| 0.666667  | 0.20584   | 0.04       | 0.15796   | 73
|  5  | Accept| 0.04       | 0.23139   | 0.04       | 0.040009  | 2
|  6  | Accept| 0.04       | 0.24647   | 0.04       | 0.040007  | 4
|  7  | Accept| 0.046667  | 0.26569   | 0.04       | 0.040006  | 5
|  8  | Best  | 0.033333  | 0.20998   | 0.033333  | 0.033344  | 1
|  9  | Accept| 0.066667  | 0.18094   | 0.033333  | 0.033343  | 21
| 10  | Accept| 0.066667  | 0.27873   | 0.033333  | 0.033342  | 8
| 11  | Accept| 0.04       | 0.20902   | 0.033333  | 0.03334   | 3
| 12  | Accept| 0.033333  | 0.23535   | 0.033333  | 0.033337  | 1
| 13  | Accept| 0.033333  | 0.21125   | 0.033333  | 0.033335  | 1
| 14  | Accept| 0.033333  | 0.2127    | 0.033333  | 0.033335  | 1
| 15  | Accept| 0.333333  | 0.17823   | 0.033333  | 0.033335  | 45
| 16  | Accept| 0.066667  | 0.19057   | 0.033333  | 0.033335  | 26
| 17  | Accept| 0.066667  | 0.24232   | 0.033333  | 0.033335  | 16
```

	18		Accept		0.066667		0.21773		0.033333		0.033335		6
	19		Accept		0.066667		0.21611		0.033333		0.033335		10
	20		Accept		0.066667		0.27248		0.033333		0.033335		36
=====													
	Iter		Eval		Objective		Objective		BestSoFar		BestSoFar		MinLeafSize
			result				runtime		(observed)		(estim.)		
=====													
	21		Accept		0.333333		0.33311		0.033333		0.034407		58
	22		Accept		0.04		0.17992		0.033333		0.034358		2
	23		Accept		0.04		0.31005		0.033333		0.034296		3
	24		Accept		0.04		0.35821		0.033333		0.034248		4
	25		Accept		0.066667		0.27757		0.033333		0.034193		14
	26		Accept		0.046667		0.31072		0.033333		0.03414		5
	27		Accept		0.066667		0.32194		0.033333		0.034104		7
	28		Accept		0.066667		0.3001		0.033333		0.033335		18
	29		Accept		0.066667		0.29124		0.033333		0.034029		23
	30		Accept		0.066667		0.23296		0.033333		0.033997		9

---

Optimization completed.

MaxObjectiveEvaluations of 30 reached.

Total function evaluations: 30

Total elapsed time: 79.5742 seconds.

Total objective function evaluation time: 7.6308

Best observed feasible point:

MinLeafSize

\_\_\_\_\_

1

Observed objective function value = 0.033333

Estimated objective function value = 0.033997

Function evaluation time = 0.20998

Best estimated feasible point (according to models):

MinLeafSize

\_\_\_\_\_

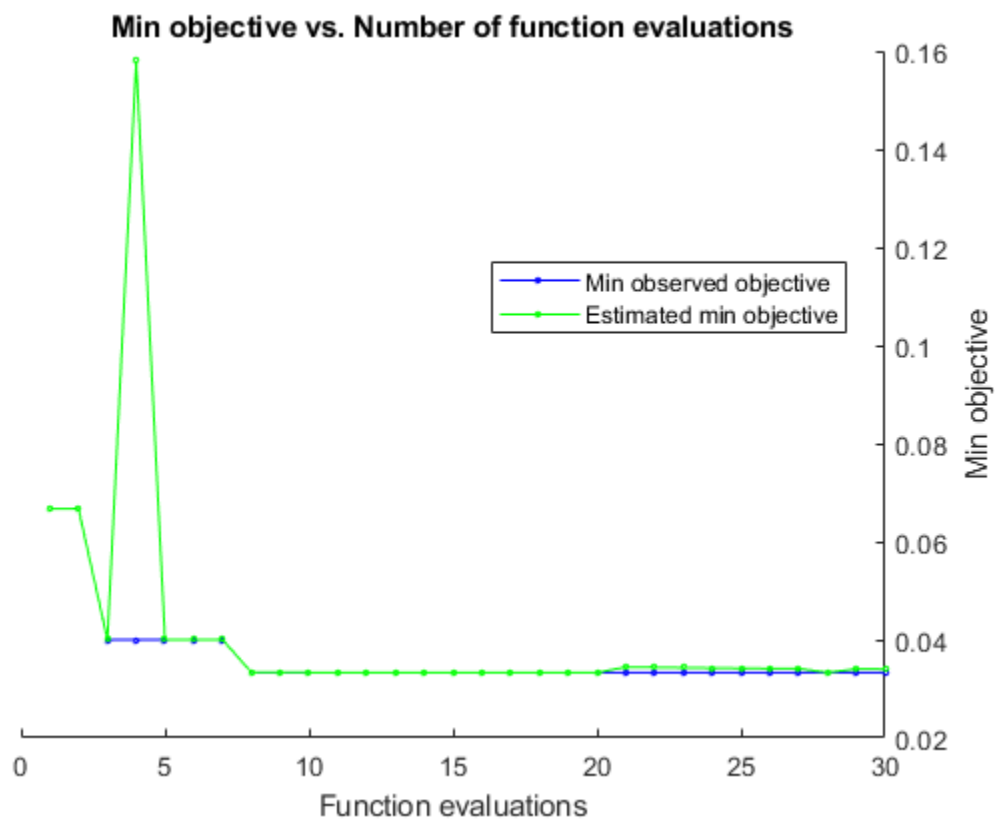
1

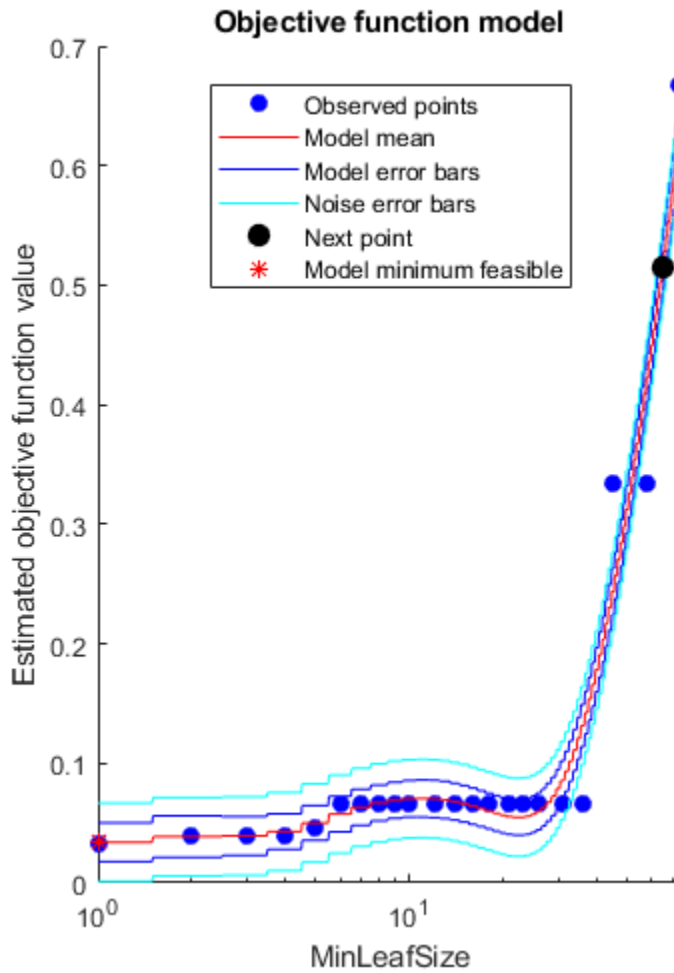
Estimated objective function value = 0.033997

Estimated function evaluation time = 0.24843

```
Mdl =
```

```
ClassificationTree  
    ResponseName: 'Y'  
    CategoricalPredictors: []  
    ClassNames: {'setosa' 'versicolor' 'virginica'}  
    ScoreTransform: 'none'  
    NumObservations: 150  
    HyperparameterOptimizationResults: [1x1 BayesianOptimization]
```





### Unbiased Predictor Importance Estimates

Load the `census1994` data set. Consider a model that predicts a person's salary category given their age, working class, education level, marital status, race, sex, capital gain and loss, and number of working hours per week.



```
load census1994
X = adultdata(:, {'age', 'workClass', 'education_num', 'marital_status', 'race', ...
                 'sex', 'capital_gain', 'capital_loss', 'hours_per_week', 'salary'});
```

Display the number of categories represented in the categorical variables using summary.

```
summary(X)
```

Variables:

```
age: 32561x1 double
```

```
Values:
```

Min	17
Median	37
Max	90

```
workClass: 32561x1 categorical
```

```
Values:
```

Federal-gov	960
Local-gov	2093
Never-worked	7
Private	22696
Self-emp-inc	1116
Self-emp-not-inc	2541
State-gov	1298
Without-pay	14
NumMissing	1836

```
education_num: 32561x1 double
```

```
Values:
```

Min	1
Median	10
Max	16

```
marital_status: 32561x1 categorical
```

```
Values:
```

Divorced	4443
Married-AF-spouse	23
Married-civ-spouse	14976
Married-spouse-absent	418
Never-married	10683
Separated	1025
Widowed	993

race: 32561x1 categorical

Values:

Amer-Indian-Eskimo	311
Asian-Pac-Islander	1039
Black	3124
Other	271
White	27816

sex: 32561x1 categorical

Values:

Female	10771
Male	21790

capital\_gain: 32561x1 double

Values:

Min	0
Median	0
Max	99999

capital\_loss: 32561x1 double

Values:

Min	0
Median	0
Max	4356

hours\_per\_week: 32561x1 double

Values:

```

        Min         1
        Median      40
        Max         99

salary: 32561x1 categorical

Values:

        <=50K      24720
        >50K       7841

```

Because there are few categories represented in the categorical variables compared to levels in the continuous variables, the standard CART, predictor-splitting algorithm prefers splitting a continuous predictor over the categorical variables.

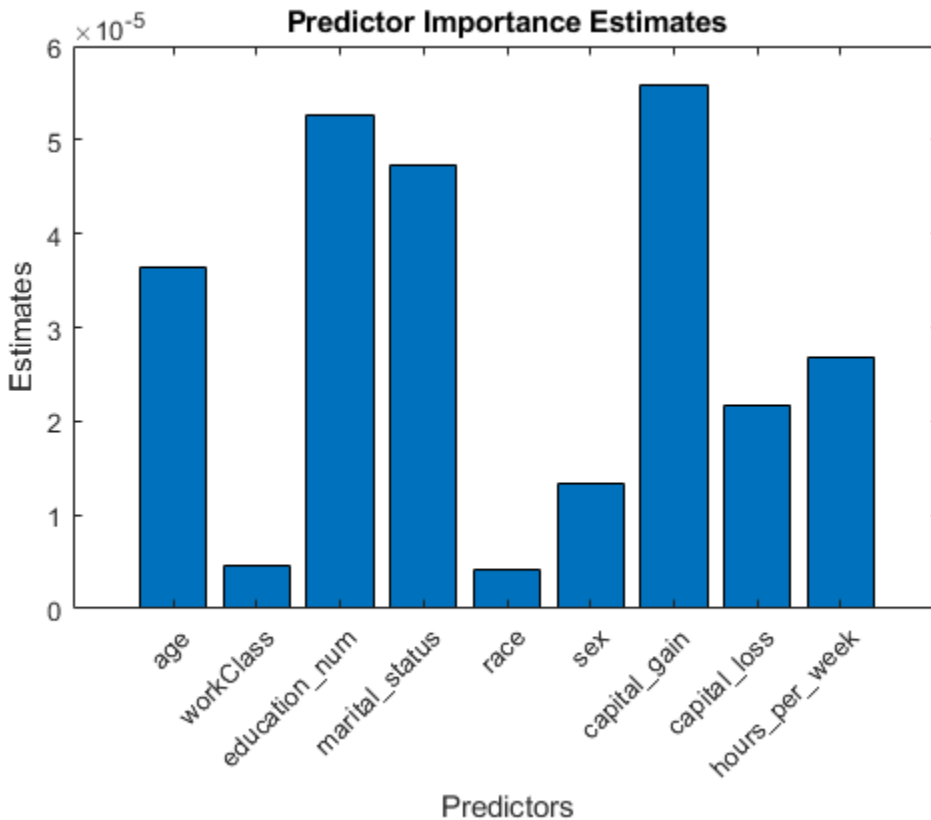
Train a classification tree using the entire data set. To grow unbiased trees, specify usage of the curvature test for splitting predictors. Because there are missing observations in the data, specify usage of surrogate splits.

```
Mdl = fitctree(X, 'salary', 'PredictorSelection', 'curvature', ...
    'Surrogate', 'on');
```

Estimate predictor importance values by summing changes in the risk due to splits on every predictor and dividing the sum by the number of branch nodes. Compare the estimates using a bar graph.

```
imp = predictorImportance(Mdl);

figure;
bar(imp);
title('Predictor Importance Estimates');
ylabel('Estimates');
xlabel('Predictors');
h = gca;
h.XTickLabel = Mdl.PredictorNames;
h.XTickLabelRotation = 45;
h.TickLabelInterpreter = 'none';
```



In this case, `capital_gain` is the most important predictor, followed by `education_num`.

## Input Arguments

### **Tbl** — Sample data

table

Sample data used to train the model, specified as a table. Each row of `Tbl` corresponds to one observation, and each column corresponds to one predictor variable. Optionally, `Tbl` can contain one additional column for the response variable. Multi-column variables and cell arrays other than cell arrays of character vectors are not allowed.

If `Tbl` contains the response variable, and you want to use all remaining variables in `Tbl` as predictors, then specify the response variable using `ResponseVarName`.

If `Tbl` contains the response variable, and you want to use only a subset of the remaining variables in `Tbl` as predictors, then specify a formula using `formula`.

If `Tbl` does not contain the response variable, then specify a response variable using `Y`. The length of response variable and the number of rows of `Tbl` must be equal.

Data Types: `table`

**ResponseVarName — Response variable name**

name of variable in `Tbl`

Response variable name, specified as the name of a variable in `Tbl`.

You must specify `ResponseVarName` as a character vector. For example, if the response variable `Y` is stored as `Tbl.Y`, then specify it as `'Y'`. Otherwise, the software treats all columns of `Tbl`, including `Y`, as predictors when training the model.

The response variable must be a categorical or character array, logical or numeric vector, or cell array of character vectors. If `Y` is a character array, then each element must correspond to one row of the array.

It is good practice to specify the order of the classes using the `ClassNames` name-value pair argument.

Data Types: `char`

**formula — Explanatory model of response and subset of predictor variables**

character vector

Explanatory model of the response and a subset of the predictor variables, specified as a character vector in the form of `'Y~X1+X2+X3'`. In this form, `Y` represents the response variable, and `X1`, `X2`, and `X3` represent the predictor variables. The variables must be variable names in `Tbl` (`Tbl.Properties.VariableNames`).

To specify a subset of variables in `Tbl` as predictors for training the model, use a formula. If you specify a formula, then the software does not use any variables in `Tbl` that do not appear in `formula`.

Data Types: `char`

**Y — Class labels**

numeric vector | categorical vector | logical vector | character array | cell array of character vectors

Class labels, specified as a numeric vector, categorical vector, logical vector, character array, or cell array of character vectors. Each row of  $Y$  represents the classification of the corresponding row of  $X$ .

When fitting the tree, `fitctree` considers NaN, '' (empty character vector), and <undefined> values in  $Y$  to be missing values. `fitctree` does not use observations with missing values for  $Y$  in the fit.

For numeric  $Y$ , consider fitting a regression tree using `fitrtree` instead.

Data Types: `single` | `double` | `char` | `logical` | `cell`

**X — Predictor data**

numeric matrix

Predictor data, specified as a numeric matrix. Each row of  $X$  corresponds to one observation, and each column corresponds to one predictor variable.

`fitctree` considers NaN values in  $X$  as missing values. `fitctree` does not use observations with all missing values for  $X$  in the fit. `fitctree` uses observations with some missing values for  $X$  to find splits on variables for which these observations have valid values.

Data Types: `single` | `double`

**Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

---

**Note** You cannot use any cross-validation name-value pair along with `OptimizeHyperparameters`. You can modify the cross-validation for `OptimizeHyperparameters` only by using the `HyperparameterOptimizationOptions` name-value pair.

---

Example: `'CrossVal', 'on', 'MinLeafSize', 40` specifies a cross-validated classification tree with a minimum of 40 observations per leaf.

### Model Parameters

#### AlgorithmForCategorical — Algorithm for best categorical predictor split

`'Exact' | 'PullLeft' | 'PCA' | 'OVabyClass'`

Algorithm to find the best split on a categorical predictor with  $C$  categories for data and  $K \geq 3$  classes, specified as the comma-separated pair consisting of `'AlgorithmForCategorical'` and one of the following values.

Value	Description
<code>'Exact'</code>	Consider all $2^{C-1} - 1$ combinations.
<code>'PullLeft'</code>	Start with all $C$ categories on the right branch. Consider moving each category to the left branch as it achieves the minimum impurity for the $K$ classes among the remaining categories. From this sequence, choose the split that has the lowest impurity.
<code>'PCA'</code>	Compute a score for each category using the inner product between the first principal component of a weighted covariance matrix (of the centered class probability matrix) and the vector of class probabilities for that category. Sort the scores in ascending order, and consider all $C - 1$ splits.
<code>'OVabyClass'</code>	Start with all $C$ categories on the right branch. For each class, order the categories based on their probability for that class. For the first class, consider moving each category to the left branch in order, recording the impurity criterion at each move. Repeat for the remaining classes. From this sequence, choose the split that has the minimum impurity.

`fitctree` automatically selects the optimal subset of algorithms for each split using the known number of classes and levels of a categorical predictor. For  $K = 2$  classes, `fitctree` always performs the exact search. To specify a particular algorithm, use the `'AlgorithmForCategorical'` name-value pair argument.

Example: `'AlgorithmForCategorical','PCA'`

### **CategoricalPredictors** — Categorical predictors list

vector of positive integers | logical vector | character matrix | cell array of character vectors | `'all'`

Categorical predictors list, specified as the comma-separated pair consisting of `'CategoricalPredictors'` and one of these values:

Value	Description
Vector of positive integers	An entry in the vector is the index value corresponding to the column of the predictor data (X or Tbl) that contains a categorical variable.
Logical vector	A true entry means that the corresponding column of predictor data (X or Tbl) is a categorical variable.
Character matrix	Each row of the matrix is the name of a predictor variable. The names must match the entries in <code>PredictorNames</code> . Pad the names with extra blanks so each row of the character matrix has the same length.
Cell array of character vectors	Each element in the array is the name of a predictor variable. The names must match the entries in <code>PredictorNames</code> .
<code>'all'</code>	All predictors are categorical.

By default, if the predictor data is in a table (Tbl), `fitctree` assumes that a variable is categorical if it contains logical values, categorical values, or a cell array of character vectors. If the predictor data is a matrix (X), `fitctree` assumes all predictors are continuous. To identify any categorical predictors when the data is a matrix, use the `'CategoricalPredictors'` name-value pair argument.

Example: `'CategoricalPredictors','all'`

Data Types: `single` | `double` | `logical` | `char` | `cell`



**ClassNames — Names of classes to use for training**

categorical array | cell array of character vectors | character array | logical vector | numeric vector

Names of classes to use for training, specified as the comma-separated pair consisting of 'ClassNames' and a categorical or character array, logical or numeric vector, or cell array of character vectors. ClassNames must be the same data type as *Y*.

If ClassNames is a character array, then each element must correspond to one *row* of the array.

Use ClassNames to:

- Order the classes during training.
- Specify the order of any input or output argument dimension that corresponds to the class order. For example, use ClassNames to specify the order of the dimensions of *Cost* or the column order of classification scores returned by *predict*.
- Select a subset of classes for training. For example, suppose that the set of all distinct class names in *Y* is {'a', 'b', 'c'}. To train the model using observations from classes 'a' and 'c' only, specify 'ClassNames', {'a', 'c'}.

The default is the set of all distinct class names in *Y*.

Example: 'ClassNames', {'b', 'g'}

Data Types: categorical | char | logical | single | double | cell

**Cost — Cost of misclassification**

square matrix | structure

Cost of misclassification of a point, specified as the comma-separated pair consisting of 'Cost' and one of the following:

- Square matrix, where *Cost*(*i*, *j*) is the cost of classifying a point into class *j* if its true class is *i* (i.e., the rows correspond to the true class and the columns correspond to the predicted class). To specify the class order for the corresponding rows and columns of *Cost*, also specify the ClassNames name-value pair argument.
- Structure *S* having two fields: *S*.ClassNames containing the group names as a variable of the same data type as *Y*, and *S*.ClassificationCosts containing the cost matrix.

The default is  $\text{Cost}(i, j) = 1$  if  $i \neq j$ , and  $\text{Cost}(i, j) = 0$  if  $i = j$ .

Data Types: `single` | `double` | `struct`

### **MaxDepth** — Maximum tree depth

positive integer

Maximum tree depth, specified as a positive integer. Specify a value for this parameter to return a tree with fewer levels that requires fewer passes through the tall array to compute. Generally the algorithm of `fitctree` takes one pass through the data and an additional pass for each tree level. There is no maximum tree depth by default.

---

**Note** This option only applies when using `fitctree` on tall arrays. See [Extended Capabilities on page 32-0](#) for more information.

---

### **MaxNumCategories** — Maximum category levels

10 (default) | nonnegative scalar value

Maximum category levels, specified as the comma-separated pair consisting of `'MaxNumCategories'` and a nonnegative scalar value. `fitctree` splits a categorical predictor using the exact search algorithm if the predictor has at most `MaxNumCategories` levels in the split node. Otherwise, `fitctree` finds the best categorical split using one of the inexact algorithms.

Passing a small value can lead to loss of accuracy and passing a large value can increase computation time and memory overload.

Example: `'MaxNumCategories', 8`

### **MergeLeaves** — Leaf merge flag

`'on'` (default) | `'off'`

Leaf merge flag, specified as the comma-separated pair consisting of `'MergeLeaves'` and `'on'` or `'off'`.

If `MergeLeaves` is `'on'`, then `fitctree`:

- Merges leaves that originate from the same parent node, and that yields a sum of risk values greater or equal to the risk associated with the parent node
- Estimates the optimal sequence of pruned subtrees, but does not prune the classification tree

Otherwise, `fitctree` does not merge leaves.

Example: `'MergeLeaves','off'`

### **MinParentSize** — Minimum number of branch node observations

10 (default) | positive integer value

Minimum number of branch node observations, specified as the comma-separated pair consisting of `'MinParentSize'` and a positive integer value. Each branch node in the tree has at least `MinParentSize` observations. If you supply both `MinParentSize` and `MinLeafSize`, `fitctree` uses the setting that gives larger leaves: `MinParentSize = max(MinParentSize, 2*MinLeafSize)`.

Example: `'MinParentSize',8`

Data Types: `single` | `double`

### **PredictorNames** — Predictor variable names

cell array of unique character vectors

Predictor variable names, specified as the comma-separated pair consisting of `'PredictorNames'` and a cell array of unique character vectors. The functionality of `'PredictorNames'` depends on the way you supply the training data.

- If you supply `X` and `Y`, then you can use `'PredictorNames'` to give the predictor variables in `X` names.
  - The order of the names in `PredictorNames` must correspond to the column order of `X`. That is, `PredictorNames{1}` is the name of `X(:,1)`, `PredictorNames{2}` is the name of `X(:,2)`, and so on. Also, `size(X,2)` and `numel(PredictorNames)` must be equal.
  - By default, `PredictorNames` is `{'x1','x2',...}`.
- If you supply `Tbl`, then you can use `'PredictorNames'` to choose which predictor variables to use in training. That is, `fitctree` uses the predictor variables in `PredictorNames` and the response only in training.
  - `PredictorNames` must be a subset of `Tbl.Properties.VariableNames` and cannot include the name of the response variable.
  - By default, `PredictorNames` contains the names of all predictor variables.
  - It good practice to specify the predictors for training using one of `'PredictorNames'` or formula only.

```
Example: 'PredictorNames',
{'SepalLength', 'SepalWidth', 'PedalLength', 'PedalWidth'}
```

Data Types: cell

**PredictorSelection** — Algorithm used to select the best split predictor

'allsplits' (default) | 'curvature' | 'interaction-curvature'

Algorithm used to select the best split predictor at each node, specified as the comma-separated pair consisting of 'PredictorSelection' and a value in this table.

Value	Description
'allsplits'	Standard CART — Selects the split predictor that maximizes the split-criterion gain over all possible splits of all predictors [1].
'curvature'	Curvature test on page 32-2170 — Selects the split predictor that minimizes the $p$ -value of chi-square tests of independence between each predictor and the response [4]. Training speed is similar to standard CART.
'interaction-curvature'	Interaction test on page 32-2172 — Chooses the split predictor that minimizes the $p$ -value of chi-square tests of independence between each predictor and the response, and that minimizes the $p$ -value of a chi-square test of independence between each pair of predictors and response [3]. Training speed can be slower than standard CART.

For 'curvature' and 'interaction-curvature', if all tests yield  $p$ -values greater than 0.05, then `fitctree` stops splitting nodes.

---

**Tip**

- Standard CART tends to select split predictors containing many distinct values, e.g., continuous variables, over those containing few distinct values, e.g., categorical variables [4]. Consider specifying the curvature or interaction test if any of the following are true:
  - If there are predictors that have relatively fewer distinct values than other predictors, for example, if the predictor data set is heterogeneous.
  - If an analysis of predictor importance is your goal. For more on predictor importance estimation, see `predictorImportance`.

- Trees grown using standard CART are not sensitive to predictor variable interactions. Also, such trees are less likely to identify important variables in the presence of many irrelevant predictors than the application of the interaction test. Therefore, to account for predictor interactions and identify importance variables in the presence of many irrelevant variables, specify the interaction test [3].
- Prediction speed is unaffected by the value of 'PredictorSelection'.

---

For details on how `fitctree` selects split predictors, see “Node Splitting Rules” on page 32-2174.

Example: `'PredictorSelection', 'curvature'`

Data Types: char

### **Prior — Prior probabilities**

`'empirical'` (default) | `'uniform'` | vector of scalar values | structure

Prior probabilities for each class, specified as the comma-separated pair consisting of `'Prior'` and one of the following.

- A character vector:
  - `'empirical'` determines class probabilities from class frequencies in `Y`. If you pass observation weights, `fitctree` uses the weights to compute the class probabilities.
  - `'uniform'` sets all class probabilities equal.
- A vector (one scalar value for each class). To specify the class order for the corresponding elements of `Prior`, also specify the `ClassNames` name-value pair argument.
- A structure `S` with two fields:
  - `S.ClassNames` containing the class names as a variable of the same type as `Y`
  - `S.ClassProbs` containing a vector of corresponding probabilities

If you set values for both `weights` and `prior`, the weights are renormalized to add up to the value of the prior probability in the respective class.

Example: `'Prior', 'uniform'`

**Prune — Flag to estimate optimal sequence of pruned subtrees**`'on' (default) | 'off'`

Flag to estimate the optimal sequence of pruned subtrees, specified as the comma-separated pair consisting of 'Prune' and 'on' or 'off'.

If `Prune` is 'on', then `fitctree` grows the classification tree without pruning it, but estimates the optimal sequence of pruned subtrees. Otherwise, `fitctree` grows the classification tree without estimating the optimal sequence of pruned subtrees.

To prune a trained `ClassificationTree` model, pass it to `prune`.

Example: `'Prune', 'off'`

**PruneCriterion — Pruning criterion**`'error' (default) | 'impurity'`

Pruning criterion, specified as the comma-separated pair consisting of 'PruneCriterion' and 'error' or 'impurity'.

Example: `'PruneCriterion', 'impurity'`

**ResponseName — Response variable name**`'Y' (default) | character vector`

Response variable name, specified as the comma-separated pair consisting of 'ResponseName' and a character vector representing the name of the response variable.

This name-value pair is not valid when using the `ResponseVarName` or formula input arguments.

Example: `'ResponseName', 'IrisType'`

**ScoreTransform — Score transformation**`'none' (default) | character vector | function handle`

Score transformation, specified as the comma-separated pair consisting of 'ScoreTransform' and either a character vector or a function handle.

This table summarizes the available character vectors.

Value	Description
'doublelogit'	$1/(1 + e^{-2x})$
'invlogit'	$\log(x / (1-x))$
'ismax'	Set the score for the class with the largest score to 1, and set the scores for all other classes to 0.
'logit'	$1/(1 + e^{-x})$
'none' or 'identity'	$x$ (no transformation)
'sign'	-1 for $x < 0$ 0 for $x = 0$ 1 for $x > 0$
'symmetric'	$2x - 1$
'symmetricismax'	Set the score for the class with the largest score to 1, and set the scores for all other classes to -1.
'symmetriclogit'	$2/(1 + e^{-x}) - 1$

For a MATLAB function or a function you define, use its function handle. The function handle must accept a matrix (the original scores) and return a matrix of the same size (the transformed scores).

Example: 'ScoreTransform', 'logit'

Data Types: char | function\_handle

#### Surrogate — Surrogate decision splits flag

'off' (default) | 'on' | 'all' | positive integer value

Surrogate decision splits on page 32-2174 flag, specified as the comma-separated pair consisting of 'Surrogate' and 'on', 'off', 'all', or a positive integer value.

- When set to 'on', `fitctree` finds at most 10 surrogate splits at each branch node.
- When set to 'all', `fitctree` finds all surrogate splits at each branch node. The 'all' setting can use considerable time and memory.
- When set to a positive integer value, `fitctree` finds at most the specified number of surrogate splits at each branch node.

Use surrogate splits to improve the accuracy of predictions for data with missing values. The setting also lets you compute measures of predictive association between predictors. For more details, see “Node Splitting Rules” on page 32-2174.

Example: `'Surrogate', 'on'`

Data Types: `single | double | char`

### **weights** — Observation weights

`ones(size(x,1),1)` (default) | vector of scalar values

Observation weights, specified as the comma-separated pair consisting of `'Weights'` and a vector of scalar values. The software weights the observations in each row of `X` or `Tbl` with the corresponding value in `Weights`. The size of `Weights` must equal the number of rows in `X` or `Tbl`.

If you specify the input data as a table `Tbl`, then `Weights` can be the name of a variable in `Tbl` that contains a numeric vector. In this case, you must specify `Weights` as a character vector. For example, if weights vector `W` is stored as `Tbl.W`, then specify it as `'W'`. Otherwise, the software treats all columns of `Tbl`, including `W`, as predictors when training the model.

`fitctree` normalizes the weights in each class to add up to the value of the prior probability of the class.

Data Types: `single | double`

### **Cross Validation**

#### **CrossVal** — Flag to grow cross-validated decision tree

`'off'` (default) | `'on'`

Flag to grow a cross-validated decision tree, specified as the comma-separated pair consisting of `'CrossVal'` and `'on'` or `'off'`.

If `'on'`, `fitctree` grows a cross-validated decision tree with 10 folds. You can override this cross-validation setting using one of the `'Kfold'`, `'Holdout'`, `'Leaveout'`, or `'CVPartition'` name-value pair arguments. You can only use one of these four arguments at a time when creating a cross-validated tree.

Alternatively, cross validate tree later using the `crossval` method.

Example: `'CrossVal', 'on'`



**CVPartition — Partition for cross-validated tree**

cvpartition object

Partition to use in a cross-validated tree, specified as the comma-separated pair consisting of 'CVPartition' and an object created using cvpartition.

If you use 'CVPartition', you cannot use any of the 'KFold', 'Holdout', or 'Leaveout' name-value pair arguments.

**Holdout — Fraction of data for holdout validation**

0 (default) | scalar value in the range [0,1]

Fraction of data used for holdout validation, specified as the comma-separated pair consisting of 'Holdout' and a scalar value in the range [0,1]. Holdout validation tests the specified fraction of the data, and uses the rest of the data for training.

If you use 'Holdout', you cannot use any of the 'CVPartition', 'KFold', or 'Leaveout' name-value pair arguments.

Example: 'Holdout',0.1

Data Types: single | double

**KFold — Number of folds**

10 (default) | positive integer value greater than 1

Number of folds to use in a cross-validated classifier, specified as the comma-separated pair consisting of 'KFold' and a positive integer value greater than 1. If you specify, e.g., 'KFold',  $k$ , then the software:

- 1 Randomly partitions the data into  $k$  sets
- 2 For each set, reserves the set as validation data, and trains the model using the other  $k - 1$  sets
- 3 Stores the  $k$  compact, trained models in the cells of a  $k$ -by-1 cell vector in the Trained property of the cross-validated model.

To create a cross-validated model, you can use one of these four options only: CVPartition, Holdout, KFold, or Leaveout.

Example: 'KFold',8

Data Types: single | double

**Leaveout — Leave-one-out cross-validation flag**`'off'` (default) | `'on'`

Leave-one-out cross-validation flag, specified as the comma-separated pair consisting of `'Leaveout'` and `'on'` or `'off'`. Specify `'on'` to use leave-one-out cross-validation.

If you use `'Leaveout'`, you cannot use any of the `'CVPartition'`, `'Holdout'`, or `'Kfold'` name-value pair arguments.

Example: `'Leaveout','on'`

**Hyperparameter Optimization****MaxNumSplits — Maximal number of decision splits**`size(X,1) - 1` (default) | positive integer

Maximal number of decision splits (or branch nodes), specified as the comma-separated pair consisting of `'MaxNumSplits'` and a positive integer. `fitctree` splits `MaxNumSplits` or fewer branch nodes. For more details on splitting behavior, see Algorithms on page 32-2174.

Example: `'MaxNumSplits',5`

Data Types: `single` | `double`

**MinLeafSize — Minimum number of leaf node observations**`1` (default) | positive integer value

Minimum number of leaf node observations, specified as the comma-separated pair consisting of `'MinLeafSize'` and a positive integer value. Each leaf has at least `MinLeafSize` observations per tree leaf. If you supply both `MinParentSize` and `MinLeafSize`, `fitctree` uses the setting that gives larger leaves: `MinParentSize = max(MinParentSize,2*MinLeafSize)`.

Example: `'MinLeafSize',3`

Data Types: `single` | `double`

**NumVariablesToSample — Number of predictors to select at random for each split**`'all'` | positive integer value

Number of predictors to select at random for each split, specified as the comma-separated pair consisting of `'NumVariablesToSample'` and a positive integer value. You can also specify `'all'` to use all available predictors.

Example: `'NumVariablesToSample', 3`

Data Types: `single` | `double`

### **SplitCriterion** — Split criterion

`'gdi'` (default) | `'twoing'` | `'deviance'`

Split criterion, specified as the comma-separated pair consisting of `'SplitCriterion'` and `'gdi'` (Gini's diversity index), `'twoing'` for the twoing rule, or `'deviance'` for maximum deviance reduction (also known as cross entropy).

Example: `'SplitCriterion', 'deviance'`

### **Hyperparameter Optimization**

#### **OptimizeHyperparameters** — Parameters to optimize

`'none'` (default) | `'auto'` | `'all'` | cell array of eligible parameter names | vector of `optimizableVariable` objects

Parameters to optimize, specified as:

- `'none'` — Do not optimize.
- `'auto'` — Use `{'MinLeafSize'}`
- `'all'` — Optimize all eligible parameters.
- Cell array of eligible parameter names
- Vector of `optimizableVariable` objects, typically the output of hyperparameters

The optimization attempts to minimize the cross-validation loss (error) for `fitctree` by varying the parameters. For information about cross-validation loss (albeit in a different context), see “Classification Loss” on page 32-3250. To control the cross-validation type and other aspects of the optimization, use the `HyperparameterOptimizationOptions` name-value pair.

---

**Note** `OptimizeHyperparameters` values override any values you set using other name-value pairs. For example, setting `OptimizeHyperparameters` to `'auto'` causes the `'auto'` values to apply.

---

The eligible parameters for `fitctree` are:

- `MaxNumSplits` — `fitctree` searches among integers, by default log-scaled in the range `[1, max(2, NumObservations-1)]`.
- `MinLeafSize` — `fitctree` searches among integers, by default log-scaled in the range `[1, max(2, floor(NumObservations/2))]`.
- `SplitCriterion` — For two classes, `fitctree` searches among 'gdi' and 'deviance'. For three or more classes, `fitctree` also searches among 'twoing'.
- `NumVariablesToSample` — `fitctree` does not optimize over this hyperparameter. If you pass `NumVariablesToSample` as a parameter name, `fitctree` simply uses the full number of predictors. However, `fitcensemble` does optimize over this hyperparameter.

Set nondefault parameters by passing a vector of `optimizableVariable` objects that have nondefault values. For example,

```
load fisheriris
params = hyperparameters('fitctree', meas, species);
params(1).Range = [1, 30];
```

Pass `params` as the value of `OptimizeHyperparameters`.

By default, iterative display appears at the command line, and plots appear according to the number of hyperparameters in the optimization. For the optimization and plots, the objective function is  $\log(1 + \text{cross-validation loss})$  for regression, and the misclassification rate for classification. To control the iterative display, set the `HyperparameterOptimizationOptions` name-value pair, `Verbose` field. To control the plots, set the `HyperparameterOptimizationOptions` name-value pair, `ShowPlots` field.

For an example, see “Optimize Classification Tree” on page 32-2143.

Example: 'auto'

Data Types: char | cell

### **HyperparameterOptimizationOptions** — Options for optimization structure

Options for optimization, specified as a structure. Modifies the effect of the `OptimizeHyperparameters` name-value pair. All fields in the structure are optional.

Field Name	Values	Default
Optimizer	<ul style="list-style-type: none"> <li>'bayesopt' — Use Bayesian optimization. Internally, this setting calls bayesopt.</li> <li>'gridsearch' — Use grid search with NumGridDivisions values per dimension.</li> <li>'randomsearch' — Search at random among MaxObjectiveEvaluations points.</li> </ul> <p>'gridsearch' searches in a random order, using uniform sampling without replacement from the grid. After optimization, you can get a table in grid order by using the command sortrows (Mdl.ParameterOptimizationResults).</p>	'bayesopt'
AcquisitionFunctionName	<ul style="list-style-type: none"> <li>'expected-improvement-per-second-plus'</li> <li>'expected-improvement'</li> <li>'expected-improvement-plus'</li> <li>'expected-improvement-per-second'</li> <li>'lower-confidence-bound'</li> <li>'probability-of-improvement'</li> </ul> <p>For details, see the bayesopt AcquisitionFunctionName name-value pair, or “Acquisition Function Types” on page 10-4.</p>	'expected-improvement-per-second-plus'
MaxObjectiveEvaluations	Maximum number of objective function evaluations.	30 for 'bayesopt' or 'randomsearch', and the entire grid for 'gridsearch'

Field Name	Values	Default
MaxTime	Time limit, specified as a positive real. The time limit is in seconds, as measured by <code>tic</code> and <code>toc</code> . Run time can exceed <code>MaxTime</code> because <code>MaxTime</code> does not interrupt function evaluations.	Inf
NumGridDivisions	For 'gridsearch', the number of values in each dimension. Can be a vector of positive integers giving the number of values for each dimension, or a scalar that applies to all dimensions. Ignored for categorical variables.	10
ShowPlots	Logical value indicating whether to show plots. If <code>true</code> , plots the best objective function value against iteration number. If there are one or two optimization parameters, and if <code>Optimizer</code> is 'bayesopt', then <code>ShowPlots</code> also plots a model of the objective function against the parameters.	true
SaveIntermediateResults	Logical value indicating whether to save results when <code>Optimizer</code> is 'bayesopt'. If <code>true</code> , overwrites a workspace variable named 'BayesoptResults' at each iteration. The variable is a <code>BayesianOptimization</code> object.	false
Verbose	Display to the command line. <ul style="list-style-type: none"> <li>• 0 — No iterative display</li> <li>• 1 — Iterative display</li> <li>• 2 — Iterative display with extra information</li> </ul> For details, see the <code>bayesopt</code> <code>Verbose</code> name-value pair.	1

Field Name	Values	Default
UseParallel	Logical value indicating whether to run Bayesian optimization in parallel, which requires Parallel Computing Toolbox. For details, see “Parallel Bayesian Optimization” on page 10-9.	false
Repartition	Logical value indicating whether to repartition the cross-validation at every iteration. If <code>false</code> , the optimizer uses a single partition for the optimization.  <code>true</code> usually gives the most robust results because this setting takes partitioning noise into account. However, for good results, <code>true</code> requires at least twice as many function evaluations.	false
Use no more than one of the following three field names.		
CVPartition	A <code>cvpartition</code> object, as created by <code>cvpartition</code>	Kfold = 5
Holdout	A scalar in the range (0,1) representing the holdout fraction.	
Kfold	An integer greater than 1.	

Example: `struct('MaxObjectiveEvaluations',60)`

Data Types: `struct`

## Output Arguments

**tree** — Classification tree

classification tree object

Classification tree, returned as a classification tree object.

Using the `'CrossVal'`, `'KFold'`, `'Holdout'`, `'Leaveout'`, or `'CVPartition'` options results in a tree of class `ClassificationPartitionedModel`. You cannot use a

partitioned tree for prediction, so this kind of tree does not have a `predict` method. Instead, use `kfoldpredict` to predict responses for observations not used for training.

Otherwise, `tree` is of class `ClassificationTree`, and you can use the `predict` method to make predictions.

## Definitions

### Curvature Test

The curvature test is a statistical test assessing the null hypothesis that two variables are unassociated.

The curvature test between predictor variable  $x$  and  $y$  is conducted using this process.

- 1 If  $x$  is continuous, then partition it into its quartiles. Create a nominal variable that bins observations according to which section of the partition they occupy. If there are missing values, then create an extra bin for them.
- 2 For each level in the partitioned predictor  $j = 1 \dots J$  and class in the response  $k = 1, \dots, K$ , compute the weighted proportion of observations in class  $k$

$$\hat{\pi}_{jk} = \sum_{i=1}^n I\{y_i = k\} w_i.$$

$w_i$  is the weight of observation  $i$ ,  $\sum w_i = 1$ ,  $I$  is the indicator function, and  $n$  is the

sample size. If all observations have the same weight, then  $\hat{\pi}_{jk} = \frac{n_{jk}}{n_j}$ , where  $n_{jk}$  is the number of observations in level  $j$  of the predictor that are in class  $k$ .

- 3 Compute the test statistic

$$t = n \sum_{k=1}^K \sum_{j=1}^J \frac{(\hat{\pi}_{jk} - \hat{\pi}_j \hat{\pi}_{+k})^2}{\hat{\pi}_j \hat{\pi}_{+k}}$$



$\hat{\pi}_{j+} = \sum_k \hat{\pi}_{jk}$ , that is, the marginal probability of observing the predictor at level  $j$ .

$\hat{\pi}_{+k} = \sum_j \hat{\pi}_{jk}$ , that is the marginal probability of observing class  $k$ . If  $n$  is large enough $_j$  then  $t$  is distributed as a  $\chi^2$  with  $(K - 1)(J - 1)$  degrees of freedom.

- 4 If the  $p$ -value for the test is less than 0.05, then reject the null hypothesis that there is no association between  $x$  and  $y$ .

When determining the best split predictor at each node, the standard CART algorithm prefers to select continuous predictors that have many levels. Sometimes, such a selection can be spurious and can also mask more important predictors that have fewer levels, such as categorical predictors.

The curvature test can be applied instead of standard CART to determine the best split predictor at each node. In that case, the best split predictor variable is the one that minimizes the significant  $p$ -values (those less than 0.05) of curvature tests between each predictor and the response variable. Such a selection is robust to the number of levels in individual predictors.

---

**Note** If levels of a predictor are pure for a particular class, then `fitctree` merges those levels. Therefore, in step 3 of the algorithm,  $J$  can be less than the actual number of levels in the predictor. For example, if  $x$  has 4 levels, and all observations in bins 1 and 2 belong to class 1, then those levels are pure for class 1. Consequently, `fitctree` merges the observations in bins 1 and 2, and  $J$  reduces to 3.

---

For more details on how the curvature test applies to growing classification trees, see “Node Splitting Rules” on page 32-2174 and [4].

## Impurity and Node Error

`ClassificationTree` splits nodes based on either impurity or node error.

Impurity means one of several things, depending on your choice of the `SplitCriterion` name-value pair argument:

- Gini's Diversity Index (`gdi`) — The Gini index of a node is

$$1 - \sum_i p^2(i),$$

where the sum is over the classes  $i$  at the node, and  $p(i)$  is the observed fraction of classes with class  $i$  that reach the node. A node with just one class (a pure node) has Gini index 0; otherwise the Gini index is positive. So the Gini index is a measure of node impurity.

- Deviance ('deviance') — With  $p(i)$  defined the same as for the Gini index, the deviance of a node is

$$-\sum_i p(i) \log_2 p(i).$$

A pure node has deviance 0; otherwise, the deviance is positive.

- Twoing rule ('twoing') — Twoing is not a purity measure of a node, but is a different measure for deciding how to split a node. Let  $L(i)$  denote the fraction of members of class  $i$  in the left child node after a split, and  $R(i)$  denote the fraction of members of class  $i$  in the right child node after a split. Choose the split criterion to maximize

$$P(L)P(R) \left( \sum_i |L(i) - R(i)| \right)^2,$$

where  $P(L)$  and  $P(R)$  are the fractions of observations that split to the left and right respectively. If the expression is large, the split made each child node purer.

Similarly, if the expression is small, the split made each child node similar to each other, and hence similar to the parent node, and so the split did not increase node purity.

- Node error — The node error is the fraction of misclassified classes at a node. If  $j$  is the class with the largest number of training samples at a node, the node error is

$$1 - p(j).$$

## Interaction Test

The interaction test is a statistical test that assesses the null hypothesis that there is no interaction between a pair of predictor variables and the response variable.

The interaction test assessing the association between predictor variables  $x_1$  and  $x_2$  with respect to  $y$  is conducted using this process.

- 1 If  $x_1$  or  $x_2$  is continuous, then partition that variable into its quartiles. Create a nominal variable that bins observations according to which section of the partition they occupy. If there are missing values, then create an extra bin for them.
- 2 Create the nominal variable  $z$  with  $J = J_1 J_2$  levels that assigns an index to observation  $i$  according to which levels of  $x_1$  and  $x_2$  it belongs. Remove any levels of  $z$  that do not correspond to any observations.
- 3 Conduct a curvature test on page 32-2170 between  $z$  and  $y$ .

When growing decision trees, if there are important interactions between pairs of predictors, but there are also many other less important predictors in the data, then standard CART tends to miss the important interactions. However, conducting curvature and interaction tests for predictor selection instead can improve detection of important interactions, which can yield more accurate decision trees.

For more details on how the interaction test applies to growing decision trees, see “Curvature Test” on page 32-2170, “Node Splitting Rules” on page 32-2174 and [3].

## Predictive Measure of Association

The predictive measure of association is a value that indicates the similarity between decision rules that split observations. Among all possible decision splits that are compared to the optimal split (found by growing the tree), the best surrogate decision split on page 32-2174 yields the maximum predictive measure of association. The second-best surrogate split has the second-largest predictive measure of association.

Suppose  $x_j$  and  $x_k$  are predictor variables  $j$  and  $k$ , respectively, and  $j \neq k$ . At node  $t$ , the predictive measure of association between the optimal split  $x_j < u$  and a surrogate split  $x_k < v$  is

$$\lambda_{jk} = \frac{\min(P_L, P_R) - (1 - P_{L_j L_k} - P_{R_j R_k})}{\min(P_L, P_R)}.$$

- $P_L$  is the proportion of observations in node  $t$ , such that  $x_j < u$ . The subscript  $L$  stands for the left child of node  $t$ .
- $P_R$  is the proportion of observations in node  $t$ , such that  $x_j \geq u$ . The subscript  $R$  stands for the right child of node  $t$ .

- $P_{L_j L_k}$  is the proportion of observations at node  $t$ , such that  $x_j < u$  and  $x_k < v$ .
- $P_{R_j R_k}$  is the proportion of observations at node  $t$ , such that  $x_j \geq u$  and  $x_k \geq v$ .
- Observations with missing values for  $x_j$  or  $x_k$  do not contribute to the proportion calculations.

$\lambda_{jk}$  is a value in  $(-\infty, 1]$ . If  $\lambda_{jk} > 0$ , then  $x_k < v$  is a worthwhile surrogate split for  $x_j < u$ .

## Surrogate Decision Splits

A surrogate decision split is an alternative to the optimal decision split at a given node in a decision tree. The optimal split is found by growing the tree; the surrogate split uses a similar or correlated predictor variable and split criterion.

When the value of the optimal split predictor for an observation is missing, the observation is sent to the left or right child node using the best surrogate predictor. When the value of the best surrogate split predictor for the observation is also missing, the observation is sent to the left or right child node using the second-best surrogate predictor, and so on. Candidate splits are sorted in descending order by their predictive measure of association on page 32-2664.

## Tip

- By default, `Prune` is 'on'. However, this specification does not prune the classification tree. To prune a trained classification tree, pass the classification tree to `prune`.
- After training a `ClassificationTree` model object by using `fitctree`, use the function `predict` and the trained model object to generate C code that predicts labels for new data. For details, see “Code Generation” on page 32-0 .

## Algorithms

### Node Splitting Rules

`fitctree` uses these processes to determine how to split node  $t$ .

- For standard CART (that is, if PredictorSelection is 'allpairs') and for all predictors  $x_i$ ,  $i = 1, \dots, p$ :
  - 1 fitctree computes the weighted impurity of node  $t$ ,  $i_t$ . For supported impurity measures, see SplitCriterion.
  - 2 fitctree estimates the probability that an observation is in node  $t$  using
 
$$P(T) = \sum_{j \in T} w_j.$$

$w_j$  is the weight of observation  $j$ , and  $T$  is the set of all observation indices in node  $t$ . If you do not specify Prior or Weights, then  $w_j = 1/n$ , where  $n$  is the sample size.
  - 3 fitctree sorts  $x_i$  in ascending order. Each element of the sorted predictor is a splitting candidate or cut point. fitctree stores any indices corresponding to missing values in the set  $T_U$ , which is the unsplit set.
  - 4 fitctree determines the best way to split node  $t$  using  $x_i$  by maximizing the impurity gain ( $\Delta I$ ) over all splitting candidates. That is, for all splitting candidates in  $x_i$ :
    - a fitctree splits the observations in node  $t$  into left and right child nodes ( $t_L$  and  $t_R$ , respectively).
    - b fitctree computes  $\Delta I$ . Suppose that for a particular splitting candidate,  $t_L$  and  $t_R$  contain observation indices in the sets  $T_L$  and  $T_R$ , respectively.
      - If  $x_i$  does not contain any missing values, then the impurity gain for the current splitting candidate is
 
$$\Delta I = P(T)i_t - P(T_L)i_{t_L} - P(T_R)i_{t_R}.$$
      - If  $x_i$  contains missing values then, assuming that the observations are missing at random, the impurity gain is
 
$$\Delta I_U = P(T - T_U)i_t - P(T_L)i_{t_L} - P(T_R)i_{t_R}.$$

$T - T_U$  is the set of all observation indices in node  $t$  that are not missing.
      - If you use surrogate decision splits on page 32-2174, then:
        - i fitctree computes the predictive measures of association on page 32-2173 between the decision split  $x_j < u$  and all possible decision splits  $x_k < v$ ,  $j \neq k$ .

- ii `fitctree` sorts the possible alternative decision splits in descending order by their predictive measure of association with the optimal split. The surrogate split is the decision split yielding the largest measure.
  - iii `fitctree` decides the child node assignments for observations with a missing value for  $x_i$  using the surrogate split. If the surrogate predictor also contains a missing value, then `fitctree` uses the decision split with the second largest measure, and so on, until there are no other surrogates. It is possible for `fitctree` to split two different observations at node  $t$  using two different surrogate splits. For example, suppose the predictors  $x_1$  and  $x_2$  are the best and second best surrogates, respectively, for the predictor  $x_i$ ,  $i \in \{1,2\}$ , at node  $t$ . If observation  $m$  of predictor  $x_i$  is missing (i.e.,  $x_{mi}$  is missing), but  $x_{m1}$  is not missing, then  $x_1$  is the surrogate predictor for observation  $x_{mi}$ . If observations  $x_{(m+1),i}$  and  $x_{(m+1),1}$  are missing, but  $x_{(m+1),2}$  is not missing, then  $x_2$  is the surrogate predictor for observation  $m+1$ .
  - iv `fitctree` uses the appropriate impurity gain formula. That is, if `fitctree` fails to assign all missing observations in node  $t$  to children nodes using surrogate splits, then the impurity gain is  $\Delta I_U$ . Otherwise, `fitctree` uses  $\Delta I$  for the impurity gain.
- c** `fitctree` chooses the candidate that yields the largest impurity gain.

`fitctree` splits the predictor variable at the cut point that maximizes the impurity gain.

- For the curvature test (that is, if `PredictorSelection` is 'curvature'):
  - 1** `fitctree` conducts curvature tests on page 32-2170 between each predictor and the response for observations in node  $t$ .
    - If all  $p$ -values are at least 0.05, then `fitctree` does not split node  $t$ .
    - If there is a minimal  $p$ -value, then `fitctree` chooses the corresponding predictor to split node  $t$ .
    - If more than one  $p$ -value is zero due to underflow, then `fitctree` applies standard CART to the corresponding predictors to choose the split predictor.
  - 2** If `fitctree` chooses a split predictor, then it uses standard CART to choose the cut point (see step 4 in the standard CART process).
- For the interaction test (that is, if `PredictorSelection` is 'interaction-curvature'):

- 1 For observations in node  $t$ , `fitctree` conducts curvature tests on page 32-2170 between each predictor and the response and interaction tests on page 32-2172 between each pair of predictors and the response.
  - If all  $p$ -values are at least 0.05, then `fitctree` does not split node  $t$ .
  - If there is a minimal  $p$ -value and it is the result of a curvature test, then `fitctree` chooses the corresponding predictor to split node  $t$ .
  - If there is a minimal  $p$ -value and it is the result of an interaction test, then `fitctree` chooses the split predictor using standard CART on the corresponding pair of predictors.
  - If more than one  $p$ -value is zero due to underflow, then `fitctree` applies standard CART to the corresponding predictors to choose the split predictor.
- 2 If `fitctree` chooses a split predictor, then it uses standard CART to choose the cut point (see step 4 in the standard CART process).

## Tree Depth Control

- If `MergeLeaves` is 'on' and `PruneCriterion` is 'error' (which are the default values for these name-value pair arguments), then the software applies pruning only to the leaves and by using classification error. This specification amounts to merging leaves that share the most popular class per leaf.
- To accommodate `MaxNumSplits`, `fitctree` splits all nodes in the current layer, and then counts the number of branch nodes. A layer is the set of nodes that are equidistant from the root node. If the number of branch nodes exceeds `MaxNumSplits`, `fitctree` follows this procedure:
  - 1 Determine how many branch nodes in the current layer must be unsplit so that there are at most `MaxNumSplits` branch nodes.
  - 2 Sort the branch nodes by their impurity gains.
  - 3 Unsplit the number of least successful branches.
  - 4 Return the decision tree grown so far.

This procedure produces maximally balanced trees.

- The software splits branch nodes layer by layer until at least one of these events occurs:
  - There are `MaxNumSplits` branch nodes.

- A proposed split causes the number of observations in at least one branch node to be fewer than `MinParentSize`.
- A proposed split causes the number of observations in at least one leaf node to be fewer than `MinLeafSize`.
- The algorithm cannot find a good split within a layer (i.e., the pruning criterion (see `PruneCriterion`), does not improve for all proposed splits in a layer). A special case is when all nodes are pure (i.e., all observations in the node have the same class).
- For values `'curvature'` or `'interaction-curvature'` of `PredictorSelection`, all tests yield  $p$ -values greater than 0.05.

`MaxNumSplits` and `MinLeafSize` do not affect splitting at their default values. Therefore, if you set `'MaxNumSplits'`, splitting might stop due to the value of `MinParentSize`, before `MaxNumSplits` splits occur.

## Parallelization

For dual-core systems and above, `fitctree` parallelizes training decision trees using Intel Threading Building Blocks (TBB). For details on Intel TBB, see <https://software.intel.com/en-us/intel-tbb>.

## References

- [1] Breiman, L., J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Boca Raton, FL: CRC Press, 1984.
- [2] Coppersmith, D., S. J. Hong, and J. R. M. Hosking. “Partitioning Nominal Attributes in Decision Trees.” *Data Mining and Knowledge Discovery*, Vol. 3, 1999, pp. 197–217.
- [3] Loh, W.Y. “Regression Trees with Unbiased Variable Selection and Interaction Detection.” *Statistica Sinica*, Vol. 12, 2002, pp. 361–386.
- [4] Loh, W.Y. and Y.S. Shih. “Split Selection Methods for Classification Trees.” *Statistica Sinica*, Vol. 7, 1997, pp. 815–840.



---

## Extended Capabilities

### Tall Arrays

Calculate with arrays that have more rows than fit in memory.

Usage notes and limitations:

- Supported syntaxes for tall arrays are:
  - `tree = fitctree(Tbl,Y)`
  - `tree = fitctree(X,Y)`
  - `tree = fitctree(___,Name,Value)`
- Supported name-value pairs are:
  - `'AlgorithmForCategorical'`
  - `'CategoricalPredictors'`
  - `'ClassNames'`
  - `'MaxNumCategories'`
  - `'MaxNumSplits'`
  - `'MergeLeaves'`
  - `'MinLeafSize'`
  - `'MinParentSize'`
  - `'NumVariablesToSample'`
  - `'PredictorNames'`
  - `'ResponseName'`
  - `'ScoreTransform'`
  - `'SplitCriterion'`
  - `'Weights'`
- There is an additional name-value pair specific to tall arrays:
  - `'MaxDepth'` — A positive integer specifying the maximum depth of the output tree. Specify a value for this parameter to return a tree with fewer levels that requires fewer passes through the tall array to compute. Generally the algorithm

of `fitctree` takes one pass through the data and an additional pass for each tree level. There is no maximum tree depth by default.

For more information, see “Tall Arrays” (MATLAB).

## See Also

`ClassificationPartitionedModel` | `ClassificationTree` | `kfoldpredict` | `predict` | `prune`

## Topics

“Splitting Categorical Predictors” on page 19-42

**Introduced in R2014a**

# fitglm

Create generalized linear regression model

## Syntax

```
mdl = fitglm(tbl)
mdl = fitglm(X,y)
mdl = fitglm( ____,modelspec)
mdl = fitglm( ____,Name,Value)
```

## Description

`mdl = fitglm(tbl)` returns a generalized linear model fit to variables in the table or dataset array `tbl`. By default, `fitglm` takes the last variable as the response variable.

`mdl = fitglm(X,y)` returns a generalized linear model of the responses `y`, fit to the data matrix `X`.

`mdl = fitglm( ____,modelspec)` returns a generalized linear model of the type you specify in `modelspec`.

`mdl = fitglm( ____,Name,Value)` returns a generalized linear model with additional options specified by one or more `Name,Value` pair arguments.

For example, you can specify which variables are categorical, the distribution of the response variable, and the link function to use.

## Examples

### Fit a Logistic Regression Model

Make a logistic binomial model of the probability of smoking as a function of age, weight, and sex, using a two-way interactions model.

Load the hospital dataset array.

```
load hospital
dsa = hospital;
```

Specify the model using a formula that allows up to two-way interactions between the variables age, weight, and sex. Smoker is the response variable.

```
modelspec = 'Smoker ~ Age*Weight*Sex - Age:Weight:Sex';
```

Fit a logistic binomial model.

```
mdl = fitglm(dsa,modelspec,'Distribution','binomial')
```

```
mdl =
```

Generalized linear regression model:

```
logit(Smoker) ~ 1 + Sex*Age + Sex*Weight + Age*Weight
Distribution = Binomial
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	-6.0492	19.749	-0.3063	0.75938
Sex_Male	-2.2859	12.424	-0.18399	0.85402
Age	0.11691	0.50977	0.22934	0.81861
Weight	0.031109	0.15208	0.20455	0.83792
Sex_Male:Age	0.020734	0.20681	0.10025	0.92014
Sex_Male:Weight	0.01216	0.053168	0.22871	0.8191
Age:Weight	-0.00071959	0.0038964	-0.18468	0.85348

100 observations, 93 error degrees of freedom

Dispersion: 1

Chi^2-statistic vs. constant model: 5.07, p-value = 0.535

All of the p-values (under pValue) are large. This means none of the coefficients are significant. The large *P*-value for the test of the model, 0.535, indicates that this model might not differ statistically from a constant model.

## GLM for Poisson Response

Create sample data with 20 predictors, and Poisson response using just three of the predictors, plus a constant.

```
rng('default') % for reproducibility
X = randn(100,7);
mu = exp(X(:, [1 3 6]) * [.4; .2; .3] + 1);
y = poissrnd(mu);
```

Fit a generalized linear model using the Poisson distribution.

```
mdl = fitglm(X,y,'linear','Distribution','poisson')
```

```
mdl =
```

Generalized linear regression model:

```
log(y) ~ 1 + x1 + x2 + x3 + x4 + x5 + x6 + x7
Distribution = Poisson
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	0.88723	0.070969	12.502	7.3149e-36
x1	0.44413	0.052337	8.4858	2.1416e-17
x2	0.0083388	0.056527	0.14752	0.88272
x3	0.21518	0.063416	3.3932	0.00069087
x4	-0.058386	0.065503	-0.89135	0.37274
x5	-0.060824	0.073441	-0.8282	0.40756
x6	0.34267	0.056778	6.0352	1.5878e-09
x7	0.04316	0.06146	0.70225	0.48252

100 observations, 92 error degrees of freedom

Dispersion: 1

Chi^2-statistic vs. constant model: 119, p-value = 1.55e-22

The p-values of 2.14e-17, 0.00069, and 1.58e-09 indicate that the coefficients of the variables x1, x3, and |x6| are statistically significant.

- “Generalized Linear Model Workflow” on page 12-41

## Input Arguments

### **tbl** — Input data

table | dataset array

Input data, specified as a table or dataset array. When `modelspec` is a formula, it specifies the variables to be used as the predictors and response. Otherwise, if you do not specify the predictor and response variables, the last variable is the response variable and the others are the predictor variables by default.

Predictor variables and response variables can be numeric, or any grouping variable type, such as logical or categorical (see “Grouping Variables” on page 2-59).

To set a different column as the response variable, use the `ResponseVar` name-value pair argument. To use a subset of the columns as predictors, use the `PredictorVars` name-value pair argument.

Data Types: `single` | `double` | `logical`

### **x** — Predictor variables

matrix

Predictor variables, specified as an  $n$ -by- $p$  matrix, where  $n$  is the number of observations and  $p$  is the number of predictor variables. Each column of `x` represents one variable, and each row represents one observation.

By default, there is a constant term in the model, unless you explicitly remove it, so do not include a column of 1s in `x`.

Data Types: `single` | `double` | `logical`

### **y** — Response variable

vector

Response variable, specified as an  $n$ -by-1 vector, where  $n$  is the number of observations. Each entry in `y` is the response for the corresponding row of `x`.

### **modelspec** — Model specification

character vector specifying the model |  $t$ -by- $(p+1)$  terms matrix | character vector of the form `'Y ~ terms'`

Model specification, which is the starting model for `stepwiseglm`, specified as one of the following:

- Character vector specifying the type of model.

Character Vector	Model Type
'constant'	Model contains only a constant (intercept) term.
'linear'	Model contains an intercept and linear terms for each predictor.
'interactions'	Model contains an intercept, linear terms, and all products of pairs of distinct predictors (no squared terms).
'purequadratic'	Model contains an intercept, linear terms, and squared terms.
'quadratic'	Model contains an intercept, linear terms, interactions, and squared terms.
'polyijk'	Model is a polynomial with all terms up to degree $i$ in the first predictor, degree $j$ in the second predictor, etc. Use numerals 0 through 9. For example, 'poly2111' has a constant plus all linear and product terms, and also contains terms with predictor 1 squared.

- $t$ -by- $(p+1)$  matrix, namely “Terms Matrix” on page 32-2192, specifying terms to include in model, where  $t$  is the number of terms and  $p$  is the number of predictor variables, and plus one is for the response variable.
- Character vector representing a “Formula” on page 32-2195 in the form

`'Y ~ terms'`,

where the `terms` are in “Wilkinson Notation” on page 32-2196.

Example: `'quadratic'`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'Distribution', 'normal', 'link', 'probit', 'Exclude', [23, 59]` specifies that the distribution of the response is normal, and instructs `fitglm` to use the probit link function and exclude the 23rd and 59th observations from the fit.

### **BinomialSize** — Number of trials for binomial distribution

1 (default) | scalar value | vector

Number of trials for binomial distribution, that is the sample size, specified as the comma-separated pair consisting of a scalar value or a vector of the same length as the response. This is the parameter `n` for the fitted binomial distribution. `BinomialSize` applies only when the `Distribution` parameter is `'binomial'`.

If `BinomialSize` is a scalar value, that means all observations have the same number of trials.

As an alternative to `BinomialSize`, you can specify the response as a two-column vector with counts in column 1 and `BinomialSize` in column 2.

Data Types: `single` | `double`

### **CategoricalVars** — Categorical variable list

cell array of character vectors | logical or numeric index vector

Categorical variable list, specified as the comma-separated pair consisting of `'CategoricalVars'` and either a cell array of character vectors containing categorical variable names in the table or dataset array `tbl`, or a logical or numeric index vector indicating which columns are categorical.

- If data is in a table or dataset array `tbl`, then, by default, `fitglm` treats all categorical values, logical values, character arrays, and cell arrays of character vectors as categorical variables.
- If data is in matrix `X`, then the default value of `'CategoricalVars'` is an empty matrix `[]`. That is, no variable is categorical unless you specify it as categorical.

For example, you can specify the observations 2 and 3 out of 6 as categorical using either of the following examples.

Example: `'CategoricalVars', [2, 3]`

Example: `'CategoricalVars', logical([0 1 1 0 0 0])`

Data Types: `single` | `double` | `logical` | `cell`



**DispersionFlag** — Indicator to compute dispersion parameter

false for 'binomial' and 'poisson' distributions (default) | true

Indicator to compute dispersion parameter for 'binomial' and 'poisson' distributions, specified as the comma-separated pair consisting of 'DispersionFlag' and one of the following.

true	Estimate a dispersion parameter when computing standard errors
false	Default. Use the theoretical value when computing standard errors

The fitting function always estimates the dispersion for other distributions.

Example: 'DispersionFlag', true

**Distribution** — Distribution of the response variable

'normal' (default) | 'binomial' | 'poisson' | 'gamma' | 'inverse gaussian'

Distribution of the response variable, specified as the comma-separated pair consisting of 'Distribution' and one of the following.

'normal'	Normal distribution
'binomial'	Binomial distribution
'poisson'	Poisson distribution
'gamma'	Gamma distribution
'inverse gaussian'	Inverse Gaussian distribution

Example: 'Distribution', 'gamma'

**Exclude** — Observations to exclude

logical or numeric index vector

Observations to exclude from the fit, specified as the comma-separated pair consisting of 'Exclude' and a logical or numeric index vector indicating which observations to exclude from the fit.

For example, you can exclude observations 2 and 3 out of 6 using either of the following examples.

Example: 'Exclude', [2, 3]

Example: `'Exclude', logical([0 1 1 0 0 0])`

Data Types: `single` | `double` | `logical`

### **Intercept** — Indicator for constant term

`true` (default) | `false`

Indicator for the constant term (intercept) in the fit, specified as the comma-separated pair consisting of `'Intercept'` and either `true` to include or `false` to remove the constant term from the model.

Use `'Intercept'` only when specifying the model using a character vector, not a formula or matrix.

Example: `'Intercept', false`

### **Link** — Link function

The canonical link function (default) | scalar value | structure

Link function to use in place of the canonical link function, specified as the comma-separated pair consisting of `'Link'` and one of the following.

Link Function Name	Link Function	Mean (Inverse) Function
<code>'identity'</code>	$f(\mu) = \mu$	$\mu = Xb$
<code>'log'</code>	$f(\mu) = \log(\mu)$	$\mu = \exp(Xb)$
<code>'logit'</code>	$f(\mu) = \log(\mu/(1-\mu))$	$\mu = \exp(Xb) / (1 + \exp(Xb))$
<code>'probit'</code>	$f(\mu) = \Phi^{-1}(\mu)$	$\mu = \Phi(Xb)$
<code>'comprologlog'</code>	$f(\mu) = \log(-\log(1 - \mu))$	$\mu = 1 - \exp(-\exp(Xb))$
<code>'reciprocal'</code>	$f(\mu) = 1/\mu$	$\mu = 1/(Xb)$
<code>p</code> (a number)	$f(\mu) = \mu^p$	$\mu = Xb^{1/p}$

Link Function Name	Link Function	Mean (Inverse) Function
<p>S (a structure) with three fields. Each field holds a function handle that accepts a vector of inputs and returns a vector of the same size:</p> <ul style="list-style-type: none"> <li>• <code>S.Link</code> — The link function</li> <li>• <code>S.Inverse</code> — The inverse link function</li> <li>• <code>S.Derivative</code> — The derivative of the link function</li> </ul>	$f(\mu) = S.Link(\mu)$	$\mu = S.Inverse(Xb)$

The link function defines the relationship  $f(\mu) = X*b$  between the mean response  $\mu$  and the linear combination of predictors  $X*b$ .

For more information on the canonical link functions, see [Definitions](#).

Example: `'Link', 'probit'`

#### **Offset — Offset variable**

`[]` (default) | `vector` | character vector

Offset variable in the fit, specified as the comma-separated pair consisting of `'Offset'` and a vector or name of a variable with the same length as the response.

`fitglm` and `stepwiseglm` use `Offset` as an additional predictor, with a coefficient value fixed at 1.0. In other words, the formula for fitting is

$$\mu \sim \text{Offset} + (\text{terms involving real predictors})$$

with the `Offset` predictor having coefficient 1.

For example, consider a Poisson regression model. Suppose the number of counts is known for theoretical reasons to be proportional to a predictor `A`. By using the log link function and by specifying `log(A)` as an offset, you can force the model to satisfy this theoretical constraint.

Data Types: `single` | `double` | `char`

### **PredictorVars** — Predictor variables

cell array of character vectors | logical or numeric index vector

Predictor variables to use in the fit, specified as the comma-separated pair consisting of 'PredictorVars' and either a cell array of character vectors of the variable names in the table or dataset array `tbl`, or a logical or numeric index vector indicating which columns are predictor variables.

The character vectors should be among the names in `tbl`, or the names you specify using the 'VarNames' name-value pair argument.

The default is all variables in `X`, or all variables in `tbl` except for `ResponseVar`.

For example, you can specify the second and third variables as the predictor variables using either of the following examples.

Example: `'PredictorVars', [2,3]`

Example: `'PredictorVars', logical([0 1 1 0 0 0])`

Data Types: `single` | `double` | `logical` | `cell`

### **ResponseVar** — Response variable

last column in `tbl` (default) | character vector containing variable name | logical or numeric index vector

Response variable to use in the fit, specified as the comma-separated pair consisting of 'ResponseVar' and either a character vector containing the variable name in the table or dataset array `tbl`, or a logical or numeric index vector indicating which column is the response variable. You typically need to use 'ResponseVar' when fitting a table or dataset array `tbl`.

For example, you can specify the fourth variable, say `yield`, as the response out of six variables, in one of the following ways.

Example: `'ResponseVar', 'yield'`

Example: `'ResponseVar', [4]`

Example: `'ResponseVar', logical([0 0 0 1 0 0])`

Data Types: `single` | `double` | `logical` | `char`

**VarNames — Names of variables in fit**

`{'x1', 'x2', ..., 'xn', 'y'}` (default) | cell array of character vectors

Names of variables in fit, specified as the comma-separated pair consisting of 'VarNames' and a cell array of character vectors including the names for the columns of  $X$  first, and the name for the response variable  $y$  last.

'VarNames' is not applicable to variables in a table or dataset array, because those variables already have names.

For example, if in your data, horsepower, acceleration, and model year of the cars are the predictor variables, and miles per gallon (MPG) is the response variable, then you can name the variables as follows.

Example: `'VarNames', {'Horsepower', 'Acceleration', 'Model_Year', 'MPG'}`

Data Types: `cell`

**Weights — Observation weights**

`ones(n,1)` (default) |  $n$ -by-1 vector of nonnegative scalar values

Observation weights, specified as the comma-separated pair consisting of 'Weights' and an  $n$ -by-1 vector of nonnegative scalar values, where  $n$  is the number of observations.

Data Types: `single` | `double`

## Output Arguments

**mdl — Generalized linear model**

`GeneralizedLinearModel` object

Generalized linear model representing a least-squares fit of the link of the response to the data, returned as a `GeneralizedLinearModel` object.

For properties and methods of the generalized linear model object, `mdl`, see the `GeneralizedLinearModel` class page.

## Definitions

### Terms Matrix

A terms matrix is a  $t$ -by- $(p + 1)$  matrix specifying terms in a model, where  $t$  is the number of terms,  $p$  is the number of predictor variables, and plus one is for the response variable.

The value of  $T(i, j)$  is the exponent of variable  $j$  in term  $i$ . Suppose there are three predictor variables A, B, and C:

```
[0 0 0 0] % Constant term or intercept
[0 1 0 0] % B; equivalently, A^0 * B^1 * C^0
[1 0 1 0] % A*C
[2 0 0 0] % A^2
[0 1 2 0] % B*(C^2)
```

The 0 at the end of each term represents the response variable. In general,

- If you have the variables in a table or dataset array, then 0 must represent the response variable depending on the position of the response variable. The following example illustrates this.

Load the sample data and define the dataset array.

```
load hospital
dsa = dataset(hospital.Sex,hospital.BloodPressure(:,1),hospital.Age,...
hospital.Smoker,'VarNames',{'Sex','BloodPressure','Age','Smoker'});
```

Represent the linear model 'BloodPressure ~ 1 + Sex + Age + Smoker' in a terms matrix. The response variable is in the second column of the dataset array, so there must be a column of 0s for the response variable in the second column of the terms matrix.

```
T = [0 0 0 0;1 0 0 0;0 0 1 0;0 0 0 1]
```

```
T =
```

```
0    0    0    0
1    0    0    0
0    0    1    0
0    0    0    1
```

Redefine the dataset array.

```
dsa = dataset(hospital.BloodPressure(:,1),hospital.Sex,hospital.Age,...
hospital.Smoker,'VarNames',{'BloodPressure','Sex','Age','Smoker'});
```

Now, the response variable is the first term in the dataset array. Specify the same linear model, 'BloodPressure ~ 1 + Sex + Age + Smoker', using a terms matrix.

```
T = [0 0 0 0;0 1 0 0;0 0 1 0;0 0 0 1]
```

T =

```

0     0     0     0
0     1     0     0
0     0     1     0
0     0     0     1
```

- If you have the predictor and response variables in a matrix and column vector, then you must include 0 for the response variable at the end of each term. The following example illustrates this.

Load the sample data and define the matrix of predictors.

```
load carsmall
X = [Acceleration,Weight];
```

Specify the model 'MPG ~ Acceleration + Weight + Acceleration:Weight + Weight^2' using a term matrix and fit the model to the data. This model includes the main effect and two-way interaction terms for the variables, Acceleration and Weight, and a second-order term for the variable, Weight.

```
T = [0 0 0;1 0 0;0 1 0;1 1 0;0 2 0]
```

T =

```

0     0     0
1     0     0
0     1     0
1     1     0
0     2     0
```

Fit a linear model.

```
mdl = fitlm(X,MPG,T)
```

```
mdl =
```

```
Linear regression model:
```

```
y ~ 1 + x1*x2 + x2^2
```

```
Estimated Coefficients:
```

	Estimate	SE	tStat	pValue
(Intercept)	48.906	12.589	3.8847	0.00019665
x1	0.54418	0.57125	0.95261	0.34337
x2	-0.012781	0.0060312	-2.1192	0.036857
x1:x2	-0.00010892	0.00017925	-0.6076	0.545
x2^2	9.7518e-07	7.5389e-07	1.2935	0.19917

```
Number of observations: 94, Error degrees of freedom: 89
```

```
Root Mean Squared Error: 4.1
```

```
R-squared: 0.751, Adjusted R-Squared 0.739
```

```
F-statistic vs. constant model: 67, p-value = 4.99e-26
```

Only the intercept and x2 term, which correspond to the weight variable, are significant at the 5% significance level.

Now, perform a stepwise regression with a constant model as the starting model and a linear model with interactions as the upper model.

```
T = [0 0 0;1 0 0;0 1 0;1 1 0];
```

```
mdl = stepwiselm(X,MPG,[0 0 0], 'upper', T)
```

```
1. Adding x2, FStat = 259.3087, pValue = 1.643351e-28
```

```
mdl =
```

```
Linear regression model:
```

```
y ~ 1 + x2
```

```
Estimated Coefficients:
```

	Estimate	SE	tStat	pValue
(Intercept)	49.238	1.6411	30.002	2.7015e-49
x2	-0.0086119	0.0005348	-16.103	1.6434e-28

```
Number of observations: 94, Error degrees of freedom: 92
```

```
Root Mean Squared Error: 4.13
```

```
R-squared: 0.738, Adjusted R-Squared 0.735
```

```
F-statistic vs. constant model: 259, p-value = 1.64e-28
```



The results of the stepwise regression are consistent with the results of `fitlm` in the previous step.

## Formula

A formula for model specification is a character vector of the form `'Y ~ terms'`

where

- `Y` is the response name.
- `terms` contains
  - Variable names
  - `+` means include the next variable
  - `-` means do not include the next variable
  - `:` defines an interaction, a product of terms
  - `*` defines an interaction **and all lower-order terms**
  - `^` raises the predictor to a power, exactly as in `*` repeated, so `^` includes lower order terms as well
  - `()` groups terms

---

**Note** Formulas include a constant (intercept) term by default. To exclude a constant term from the model, include `-1` in the formula.

---

For example,

`'Y ~ A + B + C'` means a three-variable linear model with intercept.

`'Y ~ A + B + C - 1'` is a three-variable linear model without intercept.

`'Y ~ A + B + C + B^2'` is a three-variable model with intercept and a  $B^2$  term.

`'Y ~ A + B^2 + C'` is the same as the previous example because  $B^2$  includes a  $B$  term.

`'Y ~ A + B + C + A:B'` includes an  $A*B$  term.

`'Y ~ A*B + C'` is the same as the previous example because  $A*B = A + B + A:B$ .

`'Y ~ A*B*C - A:B:C'` has all interactions among  $A$ ,  $B$ , and  $C$ , except the three-way interaction.

' $Y \sim A*(B + C + D)$ ' has all linear terms, plus products of A with each of the other variables.

### Wilkinson Notation

Wilkinson notation describes the factors present in models. The notation relates to factors present in models, not to the multipliers (coefficients) of those factors.

Wilkinson Notation	Factors in Standard Notation
1	Constant (intercept) term
$A^k$ , where k is a positive integer	A, $A^2$ , ..., $A^k$
$A + B$	A, B
$A*B$	A, B, $A*B$
$A:B$	$A*B$ only
$-B$	Do not include B
$A*B + C$	A, B, C, $A*B$
$A + B + C + A:B$	A, B, C, $A*B$
$A*B*C - A:B:C$	A, B, C, $A*B$ , $A*C$ , $B*C$
$A*(B + C)$	A, B, C, $A*B$ , $A*C$

Statistics and Machine Learning Toolbox notation always includes a constant term unless you explicitly remove the term using  $-1$ .

### Canonical Link Function

The default link function for a generalized linear model is the canonical link function.

## Canonical Link Functions for Generalized Linear Models

Distribution	Link Function Name	Link Function	Mean (Inverse) Function
'normal'	'identity'	$f(\mu) = \mu$	$\mu = Xb$
'binomial'	'logit'	$f(\mu) = \log(\mu/(1-\mu))$	$\mu = \exp(Xb) / (1 + \exp(Xb))$
'poisson'	'log'	$f(\mu) = \log(\mu)$	$\mu = \exp(Xb)$
'gamma'	-1	$f(\mu) = 1/\mu$	$\mu = 1/(Xb)$
'inverse gaussian'	-2	$f(\mu) = 1/\mu^2$	$\mu = (Xb)^{-1/2}$

## Tips

- The generalized linear model `mdl` is a standard linear model unless you specify otherwise with the `Distribution` name-value pair.
- For methods such as `plotResiduals` or `devianceTest`, or properties of the `GeneralizedLinearModel` object, see `GeneralizedLinearModel`.
- After training a `GeneralizedLinearModel` model object by using `fitglm`, use the function `predict` and the trained model object to generate C code that predicts responses for new data. For details, see “Code Generation” on page 32-0 .

## Alternative Functionality

Use `stepwiseglm` to select a model specification automatically. Use `step`, `addTerms`, or `removeTerms` to adjust a fitted model.

## References

- [1] Collett, D. *Modeling Binary Data*. New York: Chapman & Hall, 2002.
- [2] Dobson, A. J. *An Introduction to Generalized Linear Models*. New York: Chapman & Hall, 1990.
- [3] McCullagh, P., and J. A. Nelder. *Generalized Linear Models*. New York: Chapman & Hall, 1990.

## Extended Capabilities

### Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function supports tall arrays for out-of-memory data with some limitations.

- If any input argument to `fitglm` is a tall array, then all of the other inputs must be tall arrays as well. This includes nonempty variables supplied with the 'Weights', 'Exclude', 'Offset', and 'BinomialSize' name-value pairs.
- The default number of iterations is 5. You can change the number of iterations using the 'Options' name-value pair to pass in an options structure. Create an options structure using `statset` to specify a different value for `MaxIter`.
- For tall data, `fitglm` returns a `CompactGeneralizedLinearModel` object that contains most of the same properties as a `GeneralizedLinearModel` object. The main difference is that the compact object is sensitive to memory requirements. The compact object does not include properties that include the data, or that include an array of the same size as the data. The compact object does not contain these `GeneralizedLinearModel` properties:
  - `Diagnostics`
  - `Fitted`
  - `Offset`
  - `ObservationInfo`
  - `ObservationNames`
  - `Residuals`
  - `Steps`
  - `Variables`

You can compute the residuals directly from the compact object returned by `GLM = fitglm(X,Y)` using

```
RES = Y - predict(GLM,X);  
S = sqrt(GLM.SSE/GLM.DFE);  
histogram(RES, linspace(-3*S, 3*S, 51))
```

For more information, see “Tall Arrays” (MATLAB).

## See Also

`GeneralizedLinearModel` | `predict` | `stepwiseglm`

## Topics

“Generalized Linear Model Workflow” on page 12-41

“Generalized Linear Models” on page 12-12

**Introduced in R2013b**

## fitglme

Fit generalized linear mixed-effects model

### Syntax

```
glme = fitglme(tbl, formula)
glme = fitglme(tbl, formula, Name, Value)
```

### Description

`glme = fitglme(tbl, formula)` returns a generalized linear mixed-effects model, `glme`. The model is specified by `formula` and fitted to the predictor variables in the table or dataset array, `tbl`.

`glme = fitglme(tbl, formula, Name, Value)` returns a generalized linear mixed-effects model using additional options specified by one or more `Name, Value` pair arguments. For example, you can specify the distribution of the response, the link function, or the covariance pattern of the random-effects terms.

### Examples

#### Fit a Generalized Linear Mixed-Effects Model

Load the sample data.

```
load mfr
```

This simulated data is from a manufacturing company that operates 50 factories across the world, with each factory running a batch process to create a finished product. The company wants to decrease the number of defects in each batch, so it developed a new manufacturing process. To test the effectiveness of the new process, the company selected 20 of its factories at random to participate in an experiment: Ten factories implemented the new process, while the other ten continued to run the old process. In each of the 20

factories, the company ran five batches (for a total of 100 batches) and recorded the following data:

- Flag to indicate whether the batch used the new process (`newprocess`)
- Processing time for each batch, in hours (`time`)
- Temperature of the batch, in degrees Celsius (`temp`)
- Categorical variable indicating the supplier of the chemical used in the batch (`supplier`)
- Number of defects in the batch (`defects`)

The data also includes `time_dev` and `temp_dev`, which represent the absolute deviation of time and temperature, respectively, from the process standard of 3 hours at 20 degrees Celsius.

Fit a generalized linear mixed-effects model using `newprocess`, `time_dev`, `temp_dev`, and `supplier` as fixed-effects predictors. Include a random-effects term for intercept grouped by `factory`, to account for quality differences that might exist due to factory-specific variations. The response variable `defects` has a Poisson distribution, and the appropriate link function for this model is `log`. Use the Laplace fit method to estimate the coefficients. Specify the dummy variable encoding as `'effects'`, so the dummy variable coefficients sum to 0.

The number of defects can be modeled using a Poisson distribution  $\text{defects}_{ij} \sim \text{Poisson}(\mu_{ij})$ .

This corresponds to the generalized linear mixed-effects model

$$\log(\mu_{ij}) = \beta_0 + \beta_1 \text{newprocess}_{ij} + \beta_2 \text{time\_dev}_{ij} + \beta_3 \text{temp\_dev}_{ij} + \beta_4 \text{supplier\_C}_{ij} + \beta_5 \text{supplier\_B}_{ij} + b_i,$$

where

- $\text{defects}_{ij}$  is the number of defects observed in the batch produced by factory  $i$  during batch  $j$ .
- $\mu_{ij}$  is the mean number of defects corresponding to factory  $i$  (where  $i = 1, 2, \dots, 20$ ) during batch  $j$  (where  $j = 1, 2, \dots, 5$ ).
- $\text{newprocess}_{ij}$ ,  $\text{time\_dev}_{ij}$ , and  $\text{temp\_dev}_{ij}$  are the measurements for each variable that correspond to factory  $i$  during batch  $j$ . For example,  $\text{newprocess}_{ij}$  indicates whether the batch produced by factory  $i$  during batch  $j$  used the new process.

- `supplier_Cij` and `supplier_Bij` are dummy variables that use effects (sum-to-zero) coding to indicate whether company C or B, respectively, supplied the process chemicals for the batch produced by factory *i* during batch *j*.
- $b_i \sim N(0, \sigma_b^2)$  is a random-effects intercept for each factory *i* that accounts for factory-specific variation in quality.

```
glme = fitglme(mfr, 'defects ~ 1 + newprocess + time_dev + temp_dev + supplier + (1|factory)
  'Distribution', 'Poisson', 'Link', 'log', 'FitMethod', 'Laplace', ...
  'DummyVarCoding', 'effects');
```

Display the model.

```
disp(glme)
```

```
Generalized linear mixed-effects model fit by ML
```

```
Model information:
```

Number of observations	100
Fixed effects coefficients	6
Random effects coefficients	20
Covariance parameters	1
Distribution	Poisson
Link	Log
FitMethod	Laplace

```
Formula:
```

```
defects ~ 1 + newprocess + time_dev + temp_dev + supplier + (1 | factory)
```

```
Model fit statistics:
```

AIC	BIC	LogLikelihood	Deviance
416.35	434.58	-201.17	402.35

```
Fixed effects coefficients (95% CIs):
```

Name	Estimate	SE	tStat	DF	pValue
'(Intercept)'	1.4689	0.15988	9.1875	94	9.8194e-15
'newprocess'	-0.36766	0.17755	-2.0708	94	0.041122
'time_dev'	-0.094521	0.82849	-0.11409	94	0.90941
'temp_dev'	-0.28317	0.9617	-0.29444	94	0.76907
'supplier_C'	-0.071868	0.078024	-0.9211	94	0.35936
'supplier_B'	0.071072	0.07739	0.91836	94	0.36078



Lower	Upper
1.1515	1.7864
-0.72019	-0.015134
-1.7395	1.5505
-2.1926	1.6263
-0.22679	0.083051
-0.082588	0.22473

Random effects covariance parameters:

Group: factory (20 Levels)

Name1	Name2	Type	Estimate
'(Intercept)'	'(Intercept)'	'std'	0.31381

Group: Error

Name	Estimate
'sqrt(Dispersion)'	1

The `Model information` table displays the total number of observations in the sample data (100), the number of fixed- and random-effects coefficients (6 and 20, respectively), and the number of covariance parameters (1). It also indicates that the response variable has a Poisson distribution, the link function is Log, and the fit method is Laplace.

Formula indicates the model specification using Wilkinson's notation.

The `Model fit statistics` table displays statistics used to assess the goodness of fit of the model. This includes the Akaike information criterion (AIC), Bayesian information criterion (BIC) values, log likelihood (`LogLikelihood`), and deviance (`Deviance`) values.

The `Fixed effects coefficients` table indicates that `fitglm` returned 95% confidence intervals. It contains one row for each fixed-effects predictor, and each column contains statistics corresponding to that predictor. Column 1 (`Name`) contains the name of each fixed-effects coefficient, column 2 (`Estimate`) contains its estimated value, and column 3 (`SE`) contains the standard error of the coefficient. Column 4 (`tStat`) contains the *t*-statistic for a hypothesis test that the coefficient is equal to 0. Column 5 (`DF`) and column 6 (`pValue`) contain the degrees of freedom and *P*-value that correspond to the *t*-statistic, respectively. The last two columns (`Lower` and `Upper`) display the lower and upper limits, respectively, of the 95% confidence interval for each fixed-effects coefficient.

`Random effects covariance parameters` displays a table for each grouping variable (here, only `factory`), including its total number of levels (20), and the type and estimate of the covariance parameter. Here, `std` indicates that `fitglm` returns the

standard deviation of the random effect associated with the factory predictor, which has an estimated value of 0.31381. It also displays a table containing the error parameter type (here, the square root of the dispersion parameter), and its estimated value of 1.

The standard display generated by `fitglme` does not provide confidence intervals for the random-effects parameters. To compute and display these values, use `covarianceParameters`.

- “Fit a Generalized Linear Mixed-Effects Model” on page 12-79

## Input Arguments

### **tbl** — Input data

table | dataset array

Input data, which includes the response variable, predictor variables, and grouping variables, specified as a table or dataset array. The predictor variables can be continuous or grouping variables (see “Grouping Variables” on page 2-59). You must specify the model for the variables using `formula`.

Data Types: `single` | `double` | `char` | `cell`

### **formula** — Formula for model specification

character vector of the form `'y ~ fixed + (random1|grouping1) + ... + (randomR|groupingR)'`

Formula for model specification, specified as a character vector of the form `'y ~ fixed + (random1|grouping1) + ... + (randomR|groupingR)'`. The character vector is case sensitive. For a full description, see “Formula” on page 32-2218.

Example: `'y ~ treatment + (1|block)'`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example:

```
'Distribution', 'Poisson', 'Link', 'log', 'FitMethod', 'Laplace', 'DummyVa
```

`rCoding`, `'effects'` specifies the response variable distribution as Poisson, the link function as log, the fit method as Laplace, and dummy variable coding where the coefficients sum to 0.

### **BinomialSize** — Number of trials for binomial distribution

1 (default) | scalar value | vector | variable name

Number of trials for binomial distribution, that is the sample size, specified as the comma-separated pair consisting of a scalar value, a vector of the same length as the response, or the name of a variable in the input table. If you specify the name of a variable, then the variable must be of the same length as the response. `BinomialSize` applies only when the `Distribution` parameter is `'binomial'`.

If `BinomialSize` is a scalar value, that means all observations have the same number of trials.

Data Types: `single` | `double`

### **CheckHessian** — Indicator to check positive definiteness of Hessian

`false` (default) | `true`

Indicator to check the positive definiteness of the Hessian of the objective function with respect to unconstrained parameters at convergence, specified as the comma-separated pair consisting of `'CheckHessian'` and either `false` or `true`. Default is `false`.

Specify `'CheckHessian'` as `true` to verify optimality of the solution or to determine if the model is overparameterized in the number of covariance parameters.

If you specify `'FitMethod'` as `'MPL'` or `'REML'`, then the covariance of the fixed effects and the covariance parameters is based on the fitted linear mixed-effects model from the final pseudo likelihood iteration.

Example: `'CheckHessian', true`

### **CovarianceMethod** — Method to compute covariance of estimated parameters

`'conditional'` (default) | `'JointHessian'`

Method to compute covariance of estimated parameters, specified as the comma-separated pair consisting of `'CovarianceMethod'` and either `'conditional'` or `'JointHessian'`. If you specify `'conditional'`, then `fitglm` computes a fast approximation to the covariance of fixed effects given the estimated covariance parameters. It does not compute the covariance of covariance parameters. If you specify

'JointHessian', then `fitglme` computes the joint covariance of fixed effects and covariance parameters via the observed information matrix using the Laplacian loglikelihood.

If you specify 'FitMethod' as 'MPL' or 'REML', then the covariance of the fixed effects and the covariance parameters is based on the fitted linear mixed-effects model from the final pseudo likelihood iteration.

Example: 'CovarianceMethod', 'JointHessian'

### **CovariancePattern** — Pattern of covariance matrix

'FullCholesky' | 'Isotropic' | 'Full' | 'Diagonal' | 'CompSymm' | square symmetric logical matrix | cell array of character vectors or logical matrices

Pattern of the covariance matrix of the random effects, specified as the comma-separated pair consisting of 'CovariancePattern' and 'FullCholesky', 'Isotropic', 'Full', 'Diagonal', 'CompSymm', a square symmetric logical matrix, or a cell array containing character vectors or logical matrices.

If there are  $R$  random-effects terms, then the value of 'CovariancePattern' must be a cell array of length  $R$ , where each element  $r$  of this cell array specifies the pattern of the covariance matrix of the random-effects vector associated with the  $r$ th random-effects term. The options for each element follow.

Value	Description
'FullCholesky'	Full covariance matrix using the Cholesky parameterization. <code>fitglme</code> estimates all elements of the covariance matrix.

Value	Description
'Isotropic'	<p>Diagonal covariance matrix with equal variances. That is, off-diagonal elements of the covariance matrix are constrained to be 0, and the diagonal elements are constrained to be equal. For example, if there are three random-effects terms with an isotropic covariance structure, this covariance matrix looks like</p> $\begin{pmatrix} \sigma_b^2 & 0 & 0 \\ 0 & \sigma_b^2 & 0 \\ 0 & 0 & \sigma_b^2 \end{pmatrix}$ <p>where <math>\sigma_b^2</math> is the common variance of the random-effects terms.</p>
'Full'	<p>Full covariance matrix, using the log-Cholesky parameterization. <code>fitlme</code> estimates all elements of the covariance matrix.</p>
'Diagonal'	<p>Diagonal covariance matrix. That is, off-diagonal elements of the covariance matrix are constrained to be 0.</p> $\begin{pmatrix} \sigma_{b1}^2 & 0 & 0 \\ 0 & \sigma_{b2}^2 & 0 \\ 0 & 0 & \sigma_{b3}^2 \end{pmatrix}$

Value	Description
'CompSymm'	<p>Compound symmetry structure. That is, common variance along diagonals and equal correlation between all random effects. For example, if there are three random-effects terms with a covariance matrix having a compound symmetry structure, this covariance matrix looks like</p> $\begin{pmatrix} \sigma_{b1}^2 & \sigma_{b1,b2} & \sigma_{b1,b2} \\ \sigma_{b1,b2} & \sigma_{b1}^2 & \sigma_{b1,b2} \\ \sigma_{b1,b2} & \sigma_{b1,b2} & \sigma_{b1}^2 \end{pmatrix}$ <p>where <math>\sigma_{b1}^2</math> is the common variance of the random-effects terms and <math>\sigma_{b1,b2}</math> is the common covariance between any two random-effects term .</p>
PAT	<p>Square symmetric logical matrix. If 'CovariancePattern' is defined by the matrix PAT, and if <math>PAT(a,b) = \text{false}</math>, then the (a,b) element of the corresponding covariance matrix is constrained to be 0.</p>

For scalar random-effects terms, the default is 'Isotropic'. Otherwise, the default is 'FullCholesky'.

Example: 'CovariancePattern', 'Diagonal'

Example: 'CovariancePattern', {'Full', 'Diagonal'}

#### **DispersionFlag** — Indicator to compute dispersion parameter

false for 'binomial' and 'poisson' distributions (default) | true

Indicator to compute dispersion parameter for 'binomial' and 'poisson' distributions, specified as the comma-separated pair consisting of 'DispersionFlag' and one of the following.

Value	Description
true	Estimate a dispersion parameter when computing standard errors
false	Use the theoretical value of 1.0 when computing standard errors

'DispersionFlag' only applies if 'FitMethod' is 'MPL' or 'REMP'.

The fitting function always estimates the dispersion for other distributions.

Example: 'DispersionFlag', true

#### Distribution — Distribution of the response variable

'Normal' (default) | 'Binomial' | 'Poisson' | 'Gamma' | 'InverseGaussian'

Distribution of the response variable, specified as the comma-separated pair consisting of 'Distribution' and one of the following.

Value	Description
'Normal'	Normal distribution
'Binomial'	Binomial distribution
'Poisson'	Poisson distribution
'Gamma'	Gamma distribution
'InverseGaussian'	Inverse Gaussian distribution

Example: 'Distribution', 'Binomial'

#### DummyVarCoding — Coding to use for dummy variables

'reference' (default) | 'effects' | 'full'

Coding to use for dummy variables created from the categorical variables, specified as the comma-separated pair consisting of 'DummyVarCoding' and one of the following.

Value	Description
'reference'	Default. Coefficient for first category set to 0.
'effects'	Coefficients sum to 0.
'full'	One dummy variable for each category.

Example: 'DummyVarCoding', 'effects'

### **EBMethod** — Method used to approximate empirical Bayes estimates of random effects

'Auto' (default) | 'LineSearchNewton' | 'TrustRegion2D' | 'fsolve'

Method used to approximate empirical Bayes estimates of random effects, specified as the comma-separated pair consisting of 'EBMethod' and one of the following.

- 'Auto'
- 'LineSearchNewton'
- 'TrustRegion2D'
- 'fsolve'

'Auto' is similar to 'LineSearchNewton' but uses a different convergence criterion and does not display iterative progress. 'Auto' and 'LineSearchNewton' may fail for non-canonical link functions. For non-canonical link functions, 'TrustRegion2D' or 'fsolve' are recommended. You must have Optimization Toolbox to use 'fsolve'.

Example: 'EBMethod', 'LineSearchNewton'

### **EBOptions** — Options for empirical Bayes optimization

structure

Options for empirical Bayes optimization, specified as the comma-separated pair consisting of 'EBOptions' and a structure containing the following.

Value	Description
'TolFun'	Relative tolerance on the gradient norm. Default is 1e-6.
'TolX'	Absolute tolerance on the step size. Default is 1e-8.
'MaxIter'	Maximum number of iterations. Default is 100.
'Display'	'off', 'iter', or 'final'. Default is 'off'.

If EBMethod is 'Auto' and 'FitMethod' is 'Laplace', TolFun is the relative tolerance on the linear predictor of the model, and the 'Display' option does not apply.



If 'EBMethod' is 'fsolve', then 'EBOptions' must be specified as an object created by `optimoptions('fsolve')`.

Data Types: `struct`

#### **Exclude** — Indices for rows to exclude

use all rows without NaNs (default) | vector of integer or logical values

Indices for rows to exclude from the generalized linear mixed-effects model in the data, specified as the comma-separated pair consisting of 'Exclude' and a vector of integer or logical values.

For example, you can exclude the 13th and 67th rows from the fit as follows.

Example: 'Exclude', [13,67]

Data Types: `single` | `double` | `logical`

#### **FitMethod** — Method for estimating model parameters

'MPL' (default) | 'REML' | 'Laplace' | 'ApproximateLaplace'

Method for estimating model parameters, specified as the comma-separated pair consisting of 'FitMethod' and one of the following.

- 'MPL' — Maximum pseudo likelihood
- 'REML' — Restricted maximum pseudo likelihood
- 'Laplace' — Maximum likelihood using Laplace approximation
- 'ApproximateLaplace' — Maximum likelihood using approximate Laplace approximation with fixed effects profiled out

Example: 'FitMethod', 'REML'

#### **InitPLIterations** — Initial number of pseudo likelihood iterations

10 (default) | integer value in the range [1,∞)

Initial number of pseudo likelihood iterations used to initialize parameters for `ApproximateLaplace` and `Laplace` fit methods, specified as the comma-separated pair consisting of 'InitPLIterations' and an integer value greater than or equal to 1.

Data Types: `single` | `double`

**Link — Link function**

'identity' | 'log' | 'logit' | 'probit' | 'comploglog' | 'reciprocal' | scalar value | structure

Link function, specified as the comma-separated pair consisting of 'Link' and one of the following.

Value	Description
'identity'	$g(\mu) = \mu$  This is the default for the normal distribution.
'log'	$g(\mu) = \log(\mu)$  This is the default for the Poisson distribution.
'logit'	$g(\mu) = \log(\mu/(1-\mu))$  This is the default for the binomial distribution.
'loglog'	$g(\mu) = \log(-\log(\mu))$
'probit'	$g(\mu) = \text{norminv}(\mu)$
'comploglog'	$g(\mu) = \log(-\log(1-\mu))$
'reciprocal'	$g(\mu) = \mu.^{-1}$
Scalar value P	$g(\mu) = \mu.^P$

Value	Description
Structure S	<p>A structure containing four fields whose values are function handles with the following names:</p> <ul style="list-style-type: none"> <li>• S.Link — Link function</li> <li>• S.Derivative — Derivative</li> <li>• S.SecondDerivative — Second derivative</li> <li>• S.Inverse — Inverse of link</li> </ul> <p>Specification of S.SecondDerivative can be omitted if FitMethod is MPL or REMPL, or if S is the canonical link for the specified distribution.</p>

The default link function used by `fitglm` is the canonical link that depends on the distribution of the response.

Response Distribution	Canonical Link Function
'Normal'	'identity'
'Binomial'	'logit'
'Poisson'	'log'
'Gamma'	-1
'InverseGaussian'	-2

Example: 'Link', 'log'

Data Types: single | double | struct

**MuStart** — Starting value for conditional mean  
scalar value

Starting value for conditional mean, specified as the comma-separated pair consisting of 'MuStart' and a scalar value. Valid values are as follows.

Response Distribution	Valid Values
'Normal'	(-Inf, Inf)

Response Distribution	Valid Values
'Binomial'	(0,1)
'Poisson'	(0,Inf)
'Gamma'	(0,Inf)
'InverseGaussian'	(0,Inf)

Data Types: single | double

### offset — Offset

`zeros(n,1)` (default) |  $n$ -by-1 vector of scalar values

Offset, specified as the comma-separated pair consisting of 'Offset' and an  $n$ -by-1 vector of scalar values, where  $n$  is the length of the response vector. You can also specify the variable name of an  $n$ -by-1 vector of scalar values. 'Offset' is used as an additional predictor that has a coefficient value fixed at 1.0.

Data Types: single | double

### optimizer — Optimization algorithm

'quasinevton' (default) | 'fminsearch' | 'fminunc'

Optimization algorithm, specified as the comma-separated pair consisting of 'Optimizer' and either of the following.

Value	Description
'quasinevton'	Uses a trust region based quasi-Newton optimizer. You can change the options of the algorithm using <code>statset('fitglm')</code> . If you do not specify the options, then <code>fitglm</code> uses the default options of <code>statset('fitglm')</code> .
'fminsearch'	Uses a derivative-free Nelder-Mead method. You can change the options of the algorithm using <code>optimset('fminsearch')</code> . If you do not specify the options, then <code>fitglm</code> uses the default options of <code>optimset('fminsearch')</code> .

Value	Description
'fminunc'	Uses a line search-based quasi-Newton method. You must have Optimization Toolbox to specify this option. You can change the options of the algorithm using <code>optimoptions('fminunc')</code> . If you do not specify the options, then <code>fitglm</code> uses the default options of <code>optimoptions('fminunc')</code> with 'Algorithm' set to 'quasi-newton'.

Example: `'Optimizer','fminsearch'`

### OptimizerOptions — Options for optimization algorithm

structure returned by `statset` | structure returned by `optimset` | object returned by `optimoptions`

Options for the optimization algorithm, specified as the comma-separated pair consisting of 'OptimizerOptions' and a structure returned by `statset('fitglm')`, a structure created by `optimset('fminsearch')`, or an object returned by `optimoptions('fminunc')`.

- If 'Optimizer' is 'fminsearch', then use `optimset('fminsearch')` to change the options of the algorithm. If 'Optimizer' is 'fminsearch' and you do not supply 'OptimizerOptions', then the defaults used in `fitglm` are the default options created by `optimset('fminsearch')`.
- If 'Optimizer' is 'fminunc', then use `optimoptions('fminunc')` to change the options of the optimization algorithm. See `optimoptions` for the options 'fminunc' uses. If 'Optimizer' is 'fminunc' and you do not supply 'OptimizerOptions', then the defaults used in `fitglm` are the default options created by `optimoptions('fminunc')` with 'Algorithm' set to 'quasi-newton'.
- If 'Optimizer' is 'quasnewton', then use `statset('fitglm')` to change the optimization parameters. If 'Optimizer' is 'quasnewton' and you do not change the optimization parameters using `statset`, then `fitglm` uses the default options created by `statset('fitglm')`.

The 'quasnewton' optimizer uses the following fields in the structure created by `statset('fitglm')`.

**tolFun** — Relative tolerance on gradient of objective function`1e-6` (default) | positive scalar value

Relative tolerance on the gradient of the objective function, specified as a positive scalar value.

**tolX** — Absolute tolerance on step size`1e-12` (default) | positive scalar value

Absolute tolerance on the step size, specified as a positive scalar value.

**maxIter** — Maximum number of iterations allowed`10000` (default) | positive scalar value

Maximum number of iterations allowed, specified as a positive scalar value.

**display** — Level of display`'off'` (default) | `'iter'` | `'final'`

Level of display, specified as one of `'off'`, `'iter'`, or `'final'`.

**PLIterations** — Maximum number of pseudo likelihood iterations`100` (default) | positive integer value

Maximum number of pseudo likelihood (PL) iterations, specified as the comma-separated pair consisting of `'PLIterations'` and a positive integer value. PL is used for fitting the model if `'FitMethod'` is `'MPL'` or `'REMPPL'`. For other `'FitMethod'` values, PL iterations are used to initialize parameters for subsequent optimization.

Example: `'PLIterations', 200`

Data Types: `single` | `double`

**PLTolerance** — Relative tolerance factor for pseudo likelihood iterations`1e-08` (default) | positive scalar value

Relative tolerance factor for pseudo likelihood iterations, specified as the comma-separated pair consisting of `'PLTolerance'` and a positive scalar value.

Example: `'PLTolerance', 1e-06`

Data Types: `single` | `double`

**StartMethod — Method to start iterative optimization**`'default'` (default) | `'random'`

Method to start iterative optimization, specified as the comma-separated pair consisting of `'StartMethod'` and either of the following.

Value	Description
<code>'default'</code>	An internally defined default value
<code>'random'</code>	A random initial value

Example: `'StartMethod','random'`

**UseSequentialFitting — Initial fitting type**`false` (default) | `true`

, specified as the comma-separated pair consisting of `'UseSequentialFitting'` and either `false` or `true`. If `'UseSequentialFitting'` is `false`, all maximum likelihood methods are initialized using one or more pseudo likelihood iterations. If `'UseSequentialFitting'` is `true`, the initial values from pseudo likelihood iterations are refined using `'ApproximateLaplace'` for `'Laplace'` fitting.

Example: `'UseSequentialFitting',true`

**Verbose — Indicator to display optimization process on screen**`0` (default) | `1` | `2`

Indicator to display the optimization process on screen, specified as the comma-separated pair consisting of `'Verbose'` and 0, 1, or 2. If `'Verbose'` is specified as 1 or 2, then `fitglm` displays the progress of the iterative model-fitting process. Specifying `'Verbose'` as 2 displays iterative optimization information from the individual pseudo likelihood iterations. Specifying `'Verbose'` as 1 omits this display.

The setting for `'Verbose'` overrides the field `'Display'` in `'OptimizerOptions'`.

Example: `'Verbose',1`

**weights — Observation weights**

vector of nonnegative scalar values

Observation weights, specified as the comma-separated pair consisting of `'Weights'` and an  $n$ -by-1 vector of nonnegative scalar values, where  $n$  is the number of observations.

If the response distribution is binomial or Poisson, then `'weights'` must be a vector of positive integers.

Data Types: `single` | `double`

## Output Arguments

### `glme` — Generalized linear mixed-effects model

`GeneralizedLinearMixedModel` object

Generalized linear mixed-effects model, specified as a `GeneralizedLinearMixedModel` object. For properties and methods of this object, see `GeneralizedLinearMixedModel`.

## Definitions

### Formula

In general, a formula for model specification is a character vector of the form `'y ~ terms'`. For the generalized linear mixed-effects models, this formula is in the form `'y ~ fixed + (random1|grouping1) + ... + (randomR|groupingR)'`, where `fixed` and `random` contain the fixed-effects and the random-effects terms.

Suppose a table `tbl` contains the following:

- A response variable,  $y$
- Predictor variables,  $X_j$ , which can be continuous or grouping variables
- Grouping variables,  $g_1, g_2, \dots, g_R$

where the grouping variables in  $X_j$  and  $g_r$  can be categorical, logical, character arrays, or cell arrays of character vectors.

Then, in a formula of the form, `'y ~ fixed + (random1|g1) + ... + (randomR|gR)'`, the term `fixed` corresponds to a specification of the fixed-effects design matrix  $X$ , `random1` is a specification of the random-effects design matrix  $Z_1$  corresponding to grouping variable  $g_1$ , and similarly `randomR` is a specification of the random-effects design matrix  $Z_R$  corresponding to grouping variable  $g_R$ . You can express the `fixed` and `random` terms using Wilkinson notation.



Wilkinson notation describes the factors present in models. The notation relates to factors present in models, not to the multipliers (coefficients) of those factors.

Wilkinson Notation	Factors in Standard Notation
1	Constant (intercept) term
$X^k$ , where $k$ is a positive integer	$X, X^2, \dots, X^k$
$X1 + X2$	$X1, X2$
$X1 * X2$	$X1, X2, X1.*X2$ (elementwise multiplication of $X1$ and $X2$ )
$X1 : X2$	$X1.*X2$ only
$- X2$	Do not include $X2$
$X1 * X2 + X3$	$X1, X2, X3, X1 * X2$
$X1 + X2 + X3 + X1 : X2$	$X1, X2, X3, X1 * X2$
$X1 * X2 * X3 - X1 : X2 : X3$	$X1, X2, X3, X1 * X2, X1 * X3, X2 * X3$
$X1 * (X2 + X3)$	$X1, X2, X3, X1 * X2, X1 * X3$

Statistics and Machine Learning Toolbox notation always includes a constant term unless you explicitly remove the term using  $-1$ . Here are some examples for generalized linear mixed-effects model specification.

#### Examples:

Formula	Description
'y ~ X1 + X2'	Fixed effects for the intercept, $X1$ and $X2$ . This is equivalent to 'y ~ 1 + X1 + X2'.
'y ~ -1 + X1 + X2'	No intercept and fixed effects for $X1$ and $X2$ . The implicit intercept term is suppressed by including $-1$ .
'y ~ 1 + (1   g1)'	Fixed effects for the intercept plus random effect for the intercept for each level of the grouping variable $g1$ .
'y ~ X1 + (1   g1)'	Random intercept model with a fixed slope.

Formula	Description
'y ~ X1 + (X1   g1) '	Random intercept and slope, with possible correlation between them. This is equivalent to 'y ~ 1 + X1 + (1 + X1   g1) '.
'y ~ X1 + (1   g1) + (-1 + X1   g1) '	Independent random effects terms for intercept and slope.
'y ~ 1 + (1   g1) + (1   g2) + (1   g1:g2) '	Random intercept model with independent main effects for g1 and g2, plus an independent interaction effect.

## See Also

GeneralizedLinearMixedModel

## Topics

“Fit a Generalized Linear Mixed-Effects Model” on page 12-79

“Generalized Linear Mixed-Effects Models” on page 12-67

**Introduced in R2014b**

# fitgmdist

Fit Gaussian mixture distribution to data

## Syntax

```
GMMModel = fitgmdist(X,k)
GMMModel = fitgmdist(X,k,Name,Value)
```

## Description

`GMMModel = fitgmdist(X,k)` returns a Gaussian mixture distribution model (`GMMModel`) with `k` components fitted to data (`X`).

`GMMModel = fitgmdist(X,k,Name,Value)` returns a Gaussian mixture distribution model with additional options specified by one or more `Name, Value` pair arguments.

For example, you can specify a regularization value or the covariance type.

## Examples

### Cluster Data Using a Gaussian Mixture Model

Generate data from a mixture of two bivariate Gaussian distributions.

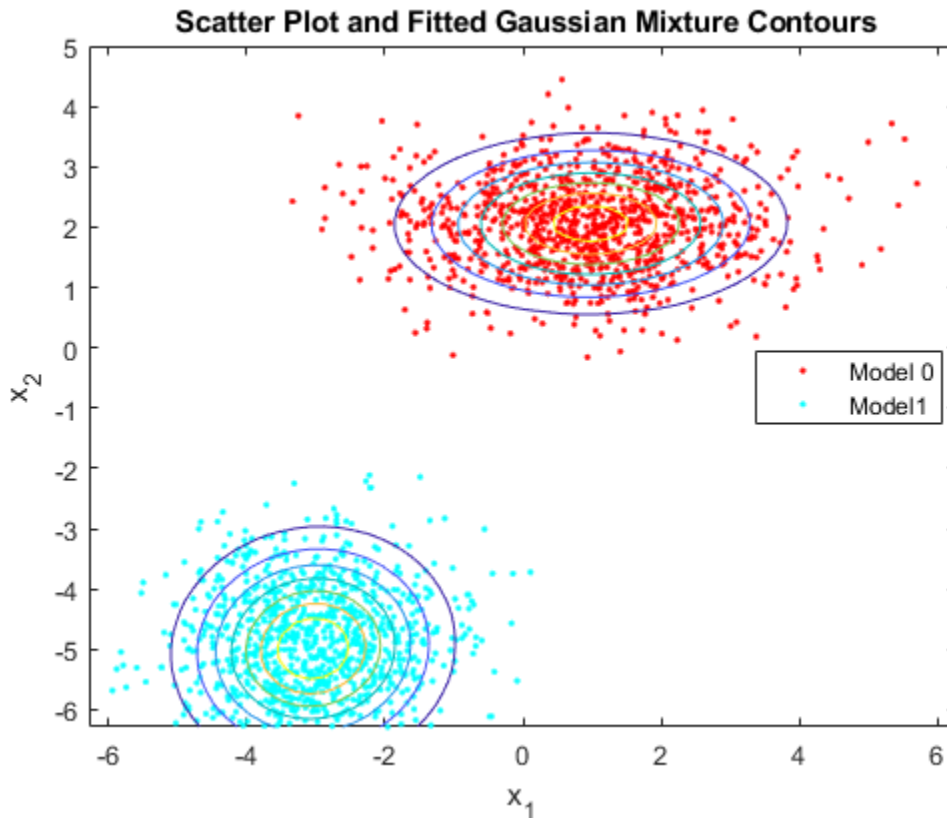
```
mu1 = [1 2];
Sigma1 = [2 0; 0 0.5];
mu2 = [-3 -5];
Sigma2 = [1 0; 0 1];
rng(1); % For reproducibility
X = [mvnrnd(mu1, Sigma1, 1000); mvnrnd(mu2, Sigma2, 1000)];
```

Fit a Gaussian mixture model. Specify that there are two components.

```
GMMModel = fitgmdist(X,2);
```

Plot the data over the fitted Gaussian mixture model contours.

```
figure
y = [zeros(1000,1);ones(1000,1)];
h = gscatter(X(:,1),X(:,2),y);
hold on
ezcontour(@ (x1,x2)pdf(GMModel,[x1 x2]),get(gca,{'XLim','YLim'}))
title('\bf Scatter Plot and Fitted Gaussian Mixture Contours')
legend(h,'Model 0','Model1')
hold off
```



## Regularize Gaussian Mixture Model Estimation

Generate data from a mixture of two bivariate Gaussian distributions. Create a third predictor that is the sum of the first and second predictors.

```
mu1 = [1 2];
Sigma1 = [1 0; 0 1];
mu2 = [3 4];
Sigma2 = [0.5 0; 0 0.5];
rng(3); % For reproducibility
X1 = [mvnrnd(mu1, Sigma1, 100); mvnrnd(mu2, Sigma2, 100)];
X = [X1, X1(:,1)+X1(:,2)];
```

The columns of  $X$  are linearly dependent. This can cause ill-conditioned covariance estimates.

Fit a Gaussian mixture model to the data. You can use `try/catch` statements to help manage error messages.

```
rng(1); % Reset seed for common start values
try
    GMMModel = fitgmdist(X,2)
catch exception
    disp('There was an error fitting the Gaussian mixture model')
    error = exception.message
end
```

```
There was an error fitting the Gaussian mixture model
```

```
error =
```

```
    'Ill-conditioned covariance created at iteration 2.'
```

The covariance estimates are ill-conditioned. Consequently, optimization stops and an error appears.

Fit a Gaussian mixture model again, but use regularization.

```
rng(3); % Reset seed for common start values
GMMModel = fitgmdist(X,2, 'RegularizationValue', 0.1)
```

```
GMMModel =
```

```
Gaussian mixture distribution with 2 components in 3 dimensions
Component 1:
Mixing proportion: 0.536725
Mean:      2.8831    3.9506    6.8338

Component 2:
Mixing proportion: 0.463275
Mean:      0.8813    1.9758    2.8571
```

In this case, the algorithm converges to a solution due to regularization.

### Select the Number of Gaussian Mixture Model Components Using PCA

Gaussian mixture models require that you specify a number of components before being fit to data. For many applications, it might be difficult to know the appropriate number of components. This example shows how to explore the data, and try to get an initial guess at the number of components using principal component analysis.

Load Fisher's iris data set.

```
load fisheriris
classes = unique(species)

classes =

    3x1 cell array

    {'setosa'   }
    {'versicolor'}
    {'virginica' }
```

The data set contains three classes of iris species. The analysis proceeds as if this is unknown.

Use principal component analysis to reduce the dimension of the data to two dimensions for visualization.

```
[~,score] = pca(meas, 'NumComponents',2);
```

Fit three Gaussian mixture models to the data by specifying 1, 2, and 3 components. Increase the number of optimization iterations to 1000. Use dot notation to store the final parameter estimates. By default, the software fits full and different covariances for each component.

```
GMMModels = cell(3,1); % Preallocation
options = statset('MaxIter',1000);
rng(1); % For reproducibility

for j = 1:3
    GMMModels{j} = fitgmdist(score,j,'Options',options);
    fprintf('\n GM Mean for %i Component(s)\n',j)
    Mu = GMMModels{j}.mu
end
```

GM Mean for 1 Component(s)

Mu =

```
1.0e-14 *
-0.2743  -0.0870
```

GM Mean for 2 Component(s)

Mu =

```
1.3212  -0.0954
-2.6424  0.1909
```

GM Mean for 3 Component(s)

Mu =

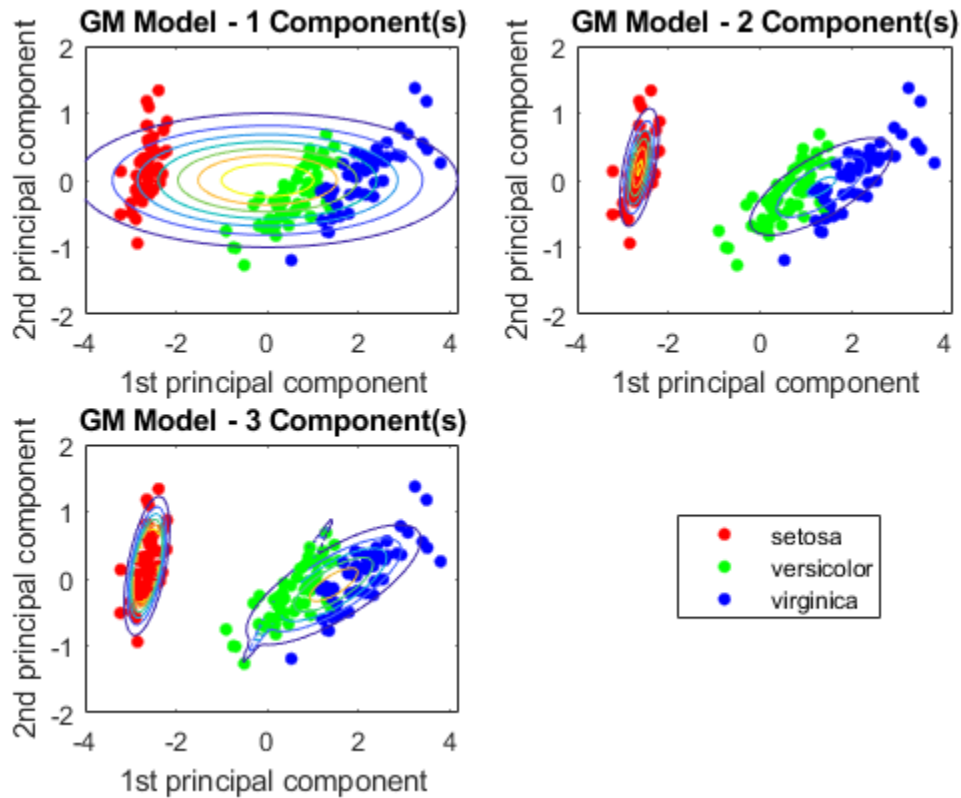
```
0.4856  -0.1287
1.4484  -0.0904
-2.6424  0.1909
```

GMModels is a cell array containing three, fitted `gmdistribution` models. The means in the three component models are different, suggesting that the model distinguishes among the three iris species.

Plot the scores over the fitted Gaussian mixture model contours. Since the data set includes labels, use `gscatter` to distinguish between the true number of components.

```
figure
for j = 1:3
    subplot(2,2,j)
    h1 = gscatter(score(:,1),score(:,2),species);
    h = gca;
    hold on
    ezcontour(@(x1,x2)pdf(GMModels{j},[x1 x2]),...
        [h.XLim h.YLim],100)
    title(sprintf('GM Model - %i Component(s)',j));
    xlabel('1st principal component');
    ylabel('2nd principal component');
    if(j ~= 3)
        legend off;
    end
    hold off
end
g = legend(h1);
g.Position = [0.7 0.25 0.1 0.1];
```





The three-component Gaussian mixture model, in conjunction with PCA, looks like it distinguishes between the three iris species.

There are other options you can use to help select the appropriate number of components for a Gaussian mixture model. For example,

- Compare multiple models with varying numbers of components using information criteria, e.g., AIC or BIC.
- Estimate the number of clusters using `evalclusters`, which supports, the Calinski-Harabasz criterion and the gap statistic, or other criteria.

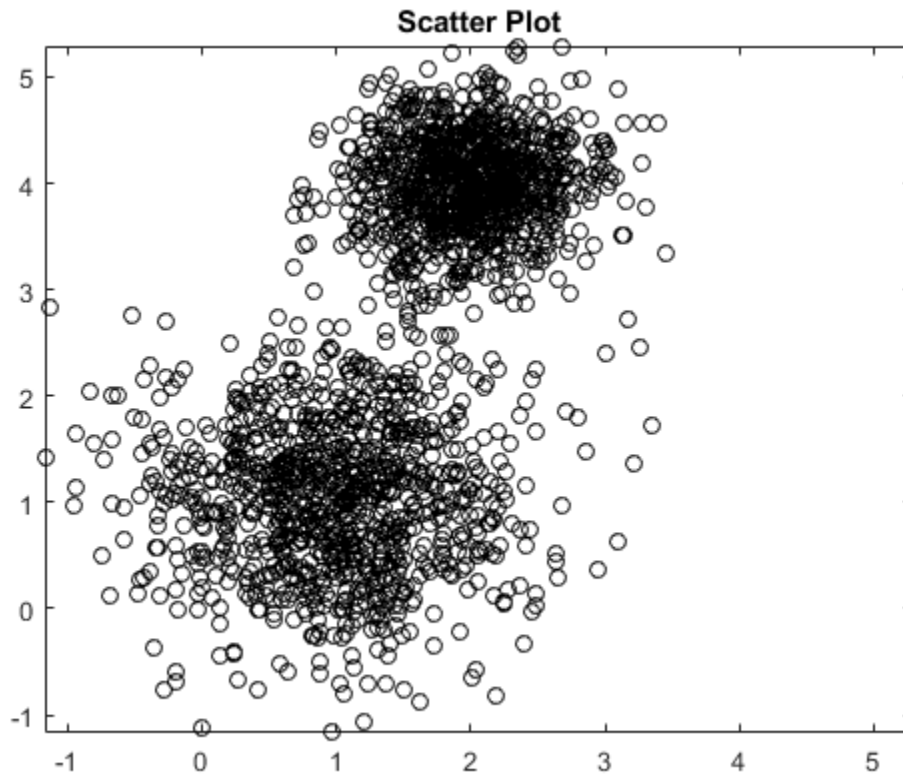
### Determine the Best Gaussian Mixture Fit Using AIC

Gaussian mixture models require that you specify a number of components before being fit to data. For many applications, it might be difficult to know the appropriate number of components. This example uses the AIC fit statistic to help you choose the best fitting Gaussian mixture model over varying numbers of components.

Generate data from a mixture of two bivariate Gaussian distributions.

```
mu1 = [1 1];
Sigma1 = [0.5 0; 0 0.5];
mu2 = [2 4];
Sigma2 = [0.2 0; 0 0.2];
rng(1);
X = [mvnrnd(mu1, Sigma1, 1000); mvnrnd(mu2, Sigma2, 1000)];

plot(X(:,1), X(:,2), 'ko')
title('Scatter Plot')
xlim([min(X(:)) max(X(:))]) % Make axes have the same scale
ylim([min(X(:)) max(X(:))])
```



Supposing that you do not know the underlying parameter values, the scatter plots suggests:

- There are two components.
- The variances between the clusters are different.
- The variance within the clusters is the same.
- There is no covariance within the clusters.

Fit a two-component Gaussian mixture model. Based on the scatter plot inspection, specify that the covariance matrices are diagonal. Print the final iteration and loglikelihood statistic to the Command Window by passing a `statset` structure as the value of the `Options` name-value pair argument.

```
options = statset('Display','final');
GMMModel = fitgmdist(X,2,'CovarianceType','diagonal','Options',options);

11 iterations, log-likelihood = -4787.38
```

GMMModel is a fitted gmdistribution model.

Examine the AIC over varying numbers of components.

```
AIC = zeros(1,4);
GMMModels = cell(1,4);
options = statset('MaxIter',500);
for k = 1:4
    GMMModels{k} = fitgmdist(X,k,'Options',options,'CovarianceType','diagonal');
    AIC(k)= GMMModels{k}.AIC;
end

[minAIC,numComponents] = min(AIC);
numComponents

BestModel = GMMModels{numComponents}
```

```
numComponents =
```

```
2
```

```
BestModel =
```

```
Gaussian mixture distribution with 2 components in 2 dimensions
```

```
Component 1:
```

```
Mixing proportion: 0.501719
```

```
Mean:    1.9824    4.0013
```

```
Component 2:
```

```
Mixing proportion: 0.498281
```

```
Mean:    0.9880    1.0511
```

The smallest AIC occurs when the software fits the two-component Gaussian mixture model.

## Set Initial Values When Fitting Gaussian Mixture Models

Gaussian mixture model parameter estimates might vary with different initial values. This example shows how to control initial values when you fit Gaussian mixture models using `fitgmdist`.

Load Fisher's iris data set. Use the petal lengths and widths as predictors.

```
load fisheriris
X = meas(:,3:4);
```

Fit a Gaussian mixture model to the data using default initial values. There are three iris species, so specify  $k = 3$  components.

```
rng(10); % For reproducibility
GMMModel1 = fitgmdist(X,3);
```

By default, the software:

- 1 Implements the “k-means++ Algorithm for Initialization” on page 32-2240 to choose  $k = 3$  initial cluster centers.
- 2 Sets the initial covariance matrices as diagonal, where element  $(j, j)$  is the variance of  $X(:, j)$ .
- 3 Treats the initial mixing proportions as uniform.

Fit a Gaussian mixture model by connecting each observation to its label.

```
y = ones(size(X,1),1);
y(strcmp(species,'setosa')) = 2;
y(strcmp(species,'virginica')) = 3;

GMMModel2 = fitgmdist(X,3,'Start',y);
```

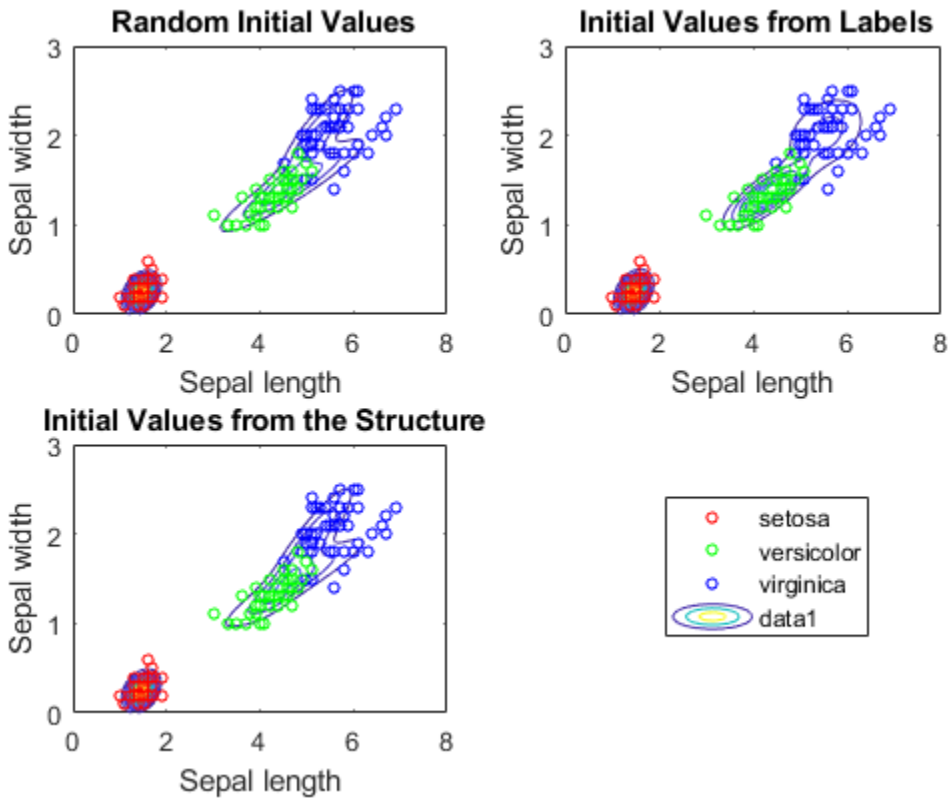
Fit a Gaussian mixture model by explicitly specifying the initial means, covariance matrices, and mixing proportions.

```
Mu = [1 1; 2 2; 3 3];
Sigma(:,:,1) = [1 1; 1 2];
Sigma(:,:,2) = 2*[1 1; 1 2];
Sigma(:,:,3) = 3*[1 1; 1 2];
PComponents = [1/2,1/4,1/4];
S = struct('mu',Mu,'Sigma',Sigma,'ComponentProportion',PComponents);

GMMModel3 = fitgmdist(X,3,'Start',S);
```

Use `gscatter` to plot a scatter diagram that distinguishes between the iris species. For each model, plot the fitted Gaussian mixture model contours.

```
figure
subplot(2,2,1)
h = gscatter(X(:,1),X(:,2),species,[],'o',4);
haxis = gca;
xlim = haxis.XLim;
ylim = haxis.YLim;
d = (max([xlim ylim])-min([xlim ylim]))/1000;
[X1Grid,X2Grid] = meshgrid(xlim(1):d:xlim(2),ylim(1):d:ylim(2));
hold on
contour(X1Grid,X2Grid,reshape(pdf(GMMModel1,[X1Grid(:) X2Grid(:)]),...
    size(X1Grid,1),size(X1Grid,2)),20)
uistack(h,'top')
title('{\bf Random Initial Values}');
xlabel('Sepal length');
ylabel('Sepal width');
legend off;
hold off
subplot(2,2,2)
h = gscatter(X(:,1),X(:,2),species,[],'o',4);
hold on
contour(X1Grid,X2Grid,reshape(pdf(GMMModel2,[X1Grid(:) X2Grid(:)]),...
    size(X1Grid,1),size(X1Grid,2)),20)
uistack(h,'top')
title('{\bf Initial Values from Labels}');
xlabel('Sepal length');
ylabel('Sepal width');
legend off
hold off
subplot(2,2,3)
h = gscatter(X(:,1),X(:,2),species,[],'o',4);
hold on
contour(X1Grid,X2Grid,reshape(pdf(GMMModel3,[X1Grid(:) X2Grid(:)]),...
    size(X1Grid,1),size(X1Grid,2)),20)
uistack(h,'top')
title('{\bf Initial Values from the Structure}');
xlabel('Sepal length');
ylabel('Sepal width');
legend('Location',[0.7,0.25,0.1,0.1]);
hold off
```



According to the counters, GMMModel2 seems to suggest a slight trimodality, while the others suggest bimodal distributions.

Display the estimated component means.

```
table(GMMModel1.mu, GMMModel2.mu, GMMModel3.mu, 'VariableNames', ...
      {'Model1', 'Model2', 'Model3'})
```

ans =

3x3 table

Model1

Model2

Model3

5.2115	2.0119	4.2857	1.3339	1.4604	0.2429
1.461	0.24423	1.462	0.246	4.7509	1.4629
4.6829	1.4429	5.5507	2.0316	5.0158	1.8592

GMMoel2 seems to distinguish between the iris species the best.

- “Clustering Using Gaussian Mixture Models” on page 16-28

## Input Arguments

### **x** — Data

numeric matrix

Data to which the Gaussian mixture model is fit, specified as a numeric matrix.

The rows of  $X$  correspond to observations, and columns correspond to variables.

NaNs indicate missing values. The software removes rows of  $X$  containing at least one NaN before fitting, which decreases the effective sample size.

Data Types: double

### **k** — Number of components

positive integer

Number of components to use when fitting Gaussian mixture model, specified as a positive integer. For example, if you specify  $k = 3$ , then the software fits a Gaussian mixture model with three distinct means, covariances matrices, and component proportions to the data ( $X$ ).

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.



Example: `'RegularizationValue',0.1,'CovarianceType','diagonal'` specifies a regularization parameter value of 0.1 and to fit diagonal covariance matrices.

### **CovarianceType** — Type of covariance matrix

`'full'` (default) | `'diagonal'`

Type of covariance matrix to fit to the data, specified as the comma-separated pair consisting of `'CovarianceType'` and either `'diagonal'` or `'full'`.

If you set `'diagonal'`, then the software fits diagonal covariance matrices. In this case, the software estimates  $k \cdot d$  covariance parameters, where  $d$  is the number of columns in  $X$  (i.e.,  $d = \text{size}(X, 2)$ ).

Otherwise, the software fits full covariance matrices. In this case, the software estimates  $k \cdot d \cdot (d+1) / 2$  covariance parameters.

Example: `'CovarianceType','diagonal'`

Data Types: char

### **Options** — Iterative EM algorithm optimization options

statset options structure

Iterative EM algorithm optimization options, specified as the comma-separated pair consisting of `'Options'` and a statset options structure.

This table describes the available name-value pair arguments.

Name	Value
<code>'Display'</code>	<p><code>'final'</code>: Display the final output.</p> <p><code>'iter'</code>: Display iterative output to the Command Window for some functions; otherwise display the final output.</p> <p><code>'off'</code>: Do not display optimization information.</p>
<code>'MaxIter'</code>	Positive integer indicating the maximum number of iterations allowed. The default is 100

Name	Value
'TolFun'	Positive scalar indicating the termination tolerance for the loglikelihood function value. The default is $1e-6$ .

Example: `'Options', statset('Display', 'final', 'MaxIter', 1500, 'TolFun', 1e-5)`

### **ProbabilityTolerance — Tolerance for posterior probabilities**

$1e-8$  (default) | nonnegative scalar value in range  $[0, 1e-6]$

Tolerance for posterior probabilities, specified as the comma-separated pair consisting of `ProbabilityTolerance` and a nonnegative scalar value in the range  $[0, 1e-6]$ .

In each iteration, after the estimation of posterior probabilities, `fitgmdist` sets any posterior probability that is not larger than the tolerance value to zero. Using a nonzero tolerance might speed up `fitgmdist`.

Example: `'ProbabilityTolerance', 0.0000025`

### **RegularizationValue — Regularization parameter value**

0 (default) | nonnegative scalar

Regularization parameter value, specified as the comma-separated pair consisting of `'RegularizationValue'` and a nonnegative scalar.

Set `RegularizationValue` to a small positive scalar to ensure that the estimated covariance matrices are positive definite.

Example: `'RegularizationValue', 0.01`

Data Types: `double`

### **Replicates — Number of times to repeat EM algorithm**

1 (default) | positive integer

Number of times to repeat the EM algorithm using a new set of initial values, specified as the comma-separated pair consisting of `'Replicates'` and a positive integer.

If `Replicates` is greater than 1, then:

- The name-value pair argument `Start` must be `plus` (the default) or `randSample`.

- `GMMModel` is the fit with the largest loglikelihood.

Example: `'Replicates', 10`

Data Types: `double`

**SharedCovariance** — Flag indicating whether all covariance matrices are identical

logical `false` (default) | logical `true`

Flag indicating whether all covariance matrices are identical (i.e., fit a pooled estimate), specified as the comma-separated pair consisting of `'SharedCovariance'` and either logical value `false` or `true`.

If `SharedCovariance` is `true`, then all  $k$  covariance matrices are equal, and the number of covariance parameters is scaled down by a factor of  $k$ .

**Start** — Initial value setting method

`'plus'` (default) | `'randSample'` | vector of integers | structure array

Initial value setting method, specified as the comma-separated pair consisting of `'Start'` and `'randSample'`, `'plus'`, a vector of integers, or a structure array.

The value of `Start` determines the initial values required by the optimization routine for each Gaussian component parameter — mean, covariance, and mixing proportion. This table summarizes the available options.

Value	Description
<code>'randSample'</code>	The software selects $k$ observations from $X$ at random as initial component means. The mixing proportions are uniform. The initial covariance matrices for all components are diagonal, where the element $j$ on the diagonal is the variance of $X(:, j)$ .
<code>'plus'</code>	The software selects $k$ observations from $X$ using the <code>kmeans++</code> algorithm on page 32-2240. The initial mixing proportions are uniform. The initial covariance matrices for all components are diagonal, where the element $j$ on the diagonal is the variance of $X(:, j)$ .

Value	Description
Vector of integers	A vector of length $n$ (the number of observations) containing an initial guess of the component index for each point. That is, each element is an integer from 1 to $k$ , which corresponds to a component. The software collects all observations corresponding to the same component, computes means, covariances, and mixing proportions for each, and sets the initial values to these statistics.
Structure array	<p>Suppose that there are <math>d</math> variables (i.e., <math>d = \text{size}(X, 2)</math>). The structure array, e.g., <math>S</math>, must have three fields:</p> <ul style="list-style-type: none"> <li>• <math>S.mu</math>: A <math>k</math>-by-<math>d</math> matrix specifying the initial mean of each component</li> <li>• <math>S.Sigma</math>: A numeric array specifying the covariance matrix of each component. <math>Sigma</math> is one of the following: <ul style="list-style-type: none"> <li>• A <math>d</math>-by-<math>d</math>-by-<math>k</math> array. <math>Sigma(:, :, j)</math> is the initial covariance matrix of component <math>j</math>.</li> <li>• A 1-by-<math>d</math>-by-<math>k</math> array. <math>\text{diag}(Sigma(:, :, j))</math> is the initial covariance matrix of component <math>j</math>.</li> <li>• A <math>d</math>-by-<math>d</math> matrix. <math>Sigma</math> is the initial covariance matrix for all components.</li> <li>• A 1-by-<math>d</math> vector. <math>\text{diag}(Sigma)</math> is the initial covariance matrix for all components.</li> </ul> </li> <li>• <math>S.ComponentProportion</math>: A 1-by-<math>k</math> vector of scalars specifying the initial mixing proportions of each component. The default is uniform.</li> </ul>

Example: `'Start', ones(n, 1)`

Data Types: `char` | `double` | `struct`

## Output Arguments

### **GMModel** — Fitted Gaussian mixture model

`gmdistribution` model

Fitted Gaussian mixture model, returned as a `gmdistribution` model.

Access properties of `GMMModel` using dot notation. For example, display the AIC by entering `GMMModel.AIC`.

## Tips

`gmdistribution` might:

- Converge to a solution where one or more of the components has an ill-conditioned or singular covariance matrix.

The following issues might result in an ill-conditioned covariance matrix:

- The number of dimensions of your data is relatively high and there are not enough observations.
- Some of the predictors (variables) of your data are highly correlated.
- Some or all the features are discrete.
- You tried to fit the data to too many components.

In general, you can avoid getting ill-conditioned covariance matrices by using one of the following precautions:

- Preprocess your data to remove correlated features.
  - Set `'SharedCovariance'` to `true` to use an equal covariance matrix for every component.
  - Set `'CovarianceType'` to `'diagonal'`.
  - Use `'RegularizationValue'` to add a very small positive number to the diagonal of every covariance matrix.
  - Try another set of initial values.
- Pass through an intermediate step where one or more of the components has an ill-conditioned covariance matrix. Try another set of initial values to avoid this issue without altering your data or model.

## Algorithms

### Gaussian Mixture Model Likelihood Optimization

The software optimizes the Gaussian mixture model likelihood using the iterative Expectation-Maximization (EM) algorithm.

### *k*-means++ Algorithm for Initialization

The *k*-means++ algorithm uses an heuristic to find centroid seeds for *k*-means clustering. `fitgmdist` can apply the same principle to initialize the EM algorithm by using the *k*-means++ algorithm to select the initial parameter values for a fitted Gaussian mixture model.

The *k*-means++ algorithm assumes the number of clusters is *k* and chooses the initial parameter values as follows.

1

Select the component mixture probability to be the uniform probability  $p_i = \frac{1}{k}$ , where  $i = 1, \dots, k$ .

2 Select the covariance matrices to be diagonal and identical, where

$$\sigma_i = \text{diag}(a_1, a_2, \dots, a_k) \text{ and } a_j = \text{var}(X_j).$$

3 Select the first initial component center  $\mu_1$  uniformly from all data points in  $X$ .

4 To choose center *j*:

**a** Compute the Mahalanobis distances from each observation to each centroid, and assign each observation to its closest centroid.

**b** For  $m = 1, \dots, n$  and  $p = 1, \dots, j - 1$ , select centroid *j* at random from  $X$  with probability

$$\frac{d^2(x_m, \mu_p)}{\sum_{h: x_h \in M_p} d^2(x_h, \mu_p)}$$

where  $d(x_m, \mu_p)$  is the distance between observation *m* and  $\mu_p$ , and  $M_p$  is the set of all observations closest to centroid  $\mu_p$  and  $x_m$  belongs to  $M_p$ .

That is, select each subsequent center with a probability proportional to the distance from itself to the closest center that you already chose.

- 5 Repeat step 4 until  $k$  centroids are chosen.

## References

- [1] McLachlan, G., and D. Peel. *Finite Mixture Models*. Hoboken, NJ: John Wiley & Sons, Inc., 2000.

## See Also

`cluster` | `gmdistribution`

## Topics

“Clustering Using Gaussian Mixture Models” on page 16-28

“Gaussian Mixture Models” on page 5-148

**Introduced in R2014a**

## fitlm

Create linear regression model

`fitlm` creates a `LinearModel` object. Once you create the object, you can see it in the workspace. You can see all the properties on page 32-3557 the object contains by clicking on it. You can create plots and do further diagnostic analysis by using methods such as `plot`, `plotResiduals`, and `plotDiagnostics`. For a full list of methods for `LinearModel`, see methods on page 32-3568.

## Syntax

```
mdl = fitlm(tbl)
mdl = fitlm(tbl,modelspec)

mdl = fitlm(X,y)
mdl = fitlm(X,y,modelspec)

mdl = fitlm( ____,Name,Value)
```

## Description

`mdl = fitlm(tbl)` returns a linear model fit to variables in the table or dataset array `tbl`. By default, `fitlm` takes the last variable as the response variable.

`mdl = fitlm(tbl,modelspec)` returns a linear model of the type you specify in `modelspec` fit to variables in the table or dataset array `tbl`.

`mdl = fitlm(X,y)` returns a linear model of the responses `y`, fit to the data matrix `X`.

`mdl = fitlm(X,y,modelspec)` returns a linear model of the type you specify in `modelspec` for the responses `y`, fit to the data matrix `X`.

`mdl = fitlm( ____,Name,Value)` returns a linear model with additional options specified by one or more `Name,Value` pair arguments.



For example, you can specify which variables are categorical, perform robust regression, or use observation weights.

## Examples

### Fit Linear Regression Using Data in Table

Load the sample data.

```
load carsmall
```

Store the variables in a table.

```
tbl = table(Weight,Acceleration,MPG,'VariableNames',{'Weight','Acceleration','MPG'});
```

Display the first five rows of the table.

```
tbl(1:5,:)
```

```
ans =
```

```
5x3 table
```

Weight	Acceleration	MPG
3504	12	18
3693	11.5	15
3436	11	18
3433	12	16
3449	10.5	17

Fit a linear regression model for miles per gallon (MPG).

```
lm = fitlm(tbl,'MPG~Weight+Acceleration')
```

```
lm =
```

```
Linear regression model:  
MPG ~ 1 + Weight + Acceleration
```

```
Estimated Coefficients:
```

	Estimate	SE	tStat	pValue
(Intercept)	45.155	3.4659	13.028	1.6266e-22
Weight	-0.0082475	0.00059836	-13.783	5.3165e-24
Acceleration	0.19694	0.14743	1.3359	0.18493

```
Number of observations: 94, Error degrees of freedom: 91  
Root Mean Squared Error: 4.12  
R-squared: 0.743, Adjusted R-Squared 0.738  
F-statistic vs. constant model: 132, p-value = 1.38e-27
```

This syntax uses Wilkinson notation to specify the `modelspec`.

The model `'MPG~Weight+Acceleration'` in this example is equivalent to fitting the model using `'linear'` as `modelspec`. For example,

```
lm2 = fitlm(tbl, 'linear');
```

When you use a character vector as `modelspec` and do not specify the response variable, `fitlm` by default accepts the last variable in `tbl` as the response variable and the other variables as the predictor variables. If there are any categorical variables and you use `'linear'` as the `modelspec`, then you must explicitly specify those variables as categorical variables using the `CategoricalVars` name-value pair argument.

### Fit Linear Regression Using Specified Model Formula

Fit a linear regression model using a model formula specified by Wilkinson notation.

Load the sample data.

```
load carsmall
```

Store the variables in a table.

```
tbl = table(Weight, Acceleration, Model_Year, MPG, 'VariableNames', {'Weight', 'Acceleration'})
```

Fit a linear regression model for miles per gallon (MPG) with weight and acceleration as the predictor variables.

```
lm = fitlm(tbl, 'MPG~Weight+Acceleration')
```

```
lm =
```

```
Linear regression model:
```

```
MPG ~ 1 + Weight + Acceleration
```

```
Estimated Coefficients:
```

	Estimate	SE	tStat	pValue
(Intercept)	45.155	3.4659	13.028	1.6266e-22
Weight	-0.0082475	0.00059836	-13.783	5.3165e-24
Acceleration	0.19694	0.14743	1.3359	0.18493

```
Number of observations: 94, Error degrees of freedom: 91
```

```
Root Mean Squared Error: 4.12
```

```
R-squared: 0.743, Adjusted R-Squared 0.738
```

```
F-statistic vs. constant model: 132, p-value = 1.38e-27
```

The **P**-value of 0.18493 indicates that Acceleration does not have a significant impact on MPG.

Remove Acceleration from the model, and try improving the model by adding the predictor variable Model\_Year. First define Model\_Year as a nominal variable.

```
tbl.Model_Year = categorical(tbl.Model_Year);
```

```
lm = fitlm(tbl, 'MPG~Weight+Model_Year')
```

```
lm =
```

```
Linear regression model:
```

```
MPG ~ 1 + Weight + Model_Year
```

```
Estimated Coefficients:
```

	Estimate	SE	tStat	pValue
--	----------	----	-------	--------

(Intercept)	40.11	1.5418	26.016	1.2024e-43
Weight	-0.0066475	0.00042802	-15.531	3.3639e-27
Model_Year_76	1.9291	0.74761	2.5804	0.011488
Model_Year_82	7.9093	0.84975	9.3078	7.8681e-15

Number of observations: 94, Error degrees of freedom: 90  
Root Mean Squared Error: 2.92  
R-squared: 0.873, Adjusted R-Squared 0.868  
F-statistic vs. constant model: 206, p-value = 3.83e-40

Specifying `modelspec` using Wilkinson notation enables you to update the model without having to change the design matrix. `fitlm` uses only the variables that are specified in the formula. It also creates the necessary two dummy indicator variables for the categorical variable `Model_Year`.

### Linear Regression with Categorical Predictor

Fit a model of a table that contains a categorical predictor.

Load the `carsmall` data.

```
load carsmall
```

Construct a table containing continuous predictor variable `Weight`, nominal predictor variable `Year`, and response variable `MPG`.

```
tbl = table(MPG,Weight);  
tbl.Year = nominal(Model_Year);
```

Create a fitted model of `MPG` as a function of `Year`, `Weight`, and `Weight^2`. (You don't have to include `Weight` explicitly in your formula because it is a lower-order term of `Weight^2`) and is included automatically.

```
mdl = fitlm(tbl,'MPG ~ Year + Weight^2')
```

```
mdl =
```

Linear regression model:

```
MPG ~ 1 + Weight + Year + Weight^2
```

```
Estimated Coefficients:
```

	Estimate	SE	tStat	pValue
(Intercept)	54.206	4.7117	11.505	2.6648e-19
Weight	-0.016404	0.0031249	-5.2493	1.0283e-06
Year_76	2.0887	0.71491	2.9215	0.0044137
Year_82	8.1864	0.81531	10.041	2.6364e-16
Weight^2	1.5573e-06	4.9454e-07	3.149	0.0022303

```
Number of observations: 94, Error degrees of freedom: 89
```

```
Root Mean Squared Error: 2.78
```

```
R-squared: 0.885, Adjusted R-Squared 0.88
```

```
F-statistic vs. constant model: 172, p-value = 5.52e-41
```

fitlm creates two dummy (indicator) variables for the nominal variate, Year. The dummy variable Year\_76 takes the value 1 if model year is 1976 and takes the value 0 if it is not. The dummy variable Year\_82 takes the value 1 if model year is 1982 and takes the value 0 if it is not. And the year 1970 is the reference year. The corresponding model is

$$\hat{MPG} = 54.206 - 0.0164(\text{Weight}) + 2.0887(\text{Year}_76) + 8.1864(\text{Year}_82) + 1.557e - 06(\text{Weight}^2)$$

## Specify Response and Predictor Variables for Linear Model

Fit a linear regression model to sample data. Specify the response and predictor variables, and include only pairwise interaction terms in the model.

Load sample data.

```
load hospital
```

Fit a linear model with interaction terms to the data. Specify weight as the response variable, and sex, age, and smoking status as the predictor variables. Also, specify that sex and smoking status are categorical variables.

```
mdl = fitlm(hospital, 'interactions', 'ResponseVar', 'Weight', ...
    'PredictorVars', {'Sex', 'Age', 'Smoker'}, ...
    'CategoricalVar', {'Sex', 'Smoker'})
```

```
mdl =
```

```
Linear regression model:
```

```
Weight ~ 1 + Sex*Age + Sex*Smoker + Age*Smoker
```

```
Estimated Coefficients:
```

	Estimate	SE	tStat	pValue
(Intercept)	118.7	7.0718	16.785	6.821e-30
Sex_Male	68.336	9.7153	7.0339	3.3386e-10
Age	0.31068	0.18531	1.6765	0.096991
Smoker_1	3.0425	10.446	0.29127	0.77149
Sex_Male:Age	-0.49094	0.24764	-1.9825	0.050377
Sex_Male:Smoker_1	0.9509	3.8031	0.25003	0.80312
Age:Smoker_1	-0.07288	0.26275	-0.27737	0.78211

```
Number of observations: 100, Error degrees of freedom: 93
```

```
Root Mean Squared Error: 8.75
```

```
R-squared: 0.898, Adjusted R-Squared 0.892
```

```
F-statistic vs. constant model: 137, p-value = 6.91e-44
```

The weight of the patients do not seem to differ significantly according to age, or the status of smoking, or interaction of these factors with patient sex at the 5% significance level.

### Fit a Robust Linear Regression Model

Fit a linear regression model using a robust fitting method.

Load the sample data.

```
load hald
```

The `hald` data measures the effect of cement composition on its hardening heat. The matrix `ingredients` contains the percent composition of four chemicals present in the cement. The array `heat` contains the heat of hardening after 180 days for each cement sample.

Fit a robust linear model to the data.

```
mdl = fitlm(ingredients,heat,'linear','RobustOpts','on')
```

```
mdl =
```

```
Linear regression model (robust fit):
  y ~ 1 + x1 + x2 + x3 + x4
```

```
Estimated Coefficients:
```

	Estimate	SE	tStat	pValue
(Intercept)	60.09	75.818	0.79256	0.4509
x1	1.5753	0.80585	1.9548	0.086346
x2	0.5322	0.78315	0.67957	0.51596
x3	0.13346	0.8166	0.16343	0.87424
x4	-0.12052	0.7672	-0.15709	0.87906

```
Number of observations: 13, Error degrees of freedom: 8
```

```
Root Mean Squared Error: 2.65
```

```
R-squared: 0.979, Adjusted R-Squared 0.969
```

```
F-statistic vs. constant model: 94.6, p-value = 9.03e-07
```

- “Examine Quality and Adjust the Fitted Model” on page 11-20
- “Predict or Simulate Responses to New Data” on page 11-38
- “Linear Regression Workflow” on page 11-43
- “Regression with Categorical Covariates” on page 2-65

## Input Arguments

### **tbl** — Input data

table | dataset array

Input data, specified as a table or dataset array. When `modelspec` is a formula, it specifies the variables to be used as the predictors and response. Otherwise, if you do not specify the predictor and response variables, the last variable is the response variable and the others are the predictor variables by default.

Predictor variables can be numeric, or any grouping variable type, such as logical or categorical (see “Grouping Variables” on page 2-59). The response must be numeric or logical.

To set a different column as the response variable, use the `ResponseVar` name-value pair argument. To use a subset of the columns as predictors, use the `PredictorVars` name-value pair argument.

### **x — Predictor variables**

matrix

Predictor variables, specified as an  $n$ -by- $p$  matrix, where  $n$  is the number of observations and  $p$  is the number of predictor variables. Each column of  $X$  represents one variable, and each row represents one observation.

By default, there is a constant term in the model, unless you explicitly remove it, so do not include a column of 1s in  $X$ .

Data Types: `single` | `double` | `logical`

### **y — Response variable**

vector

Response variable, specified as an  $n$ -by-1 vector, where  $n$  is the number of observations. Each entry in  $y$  is the response for the corresponding row of  $X$ .

Data Types: `single` | `double` | `logical`

### **modelspec — Model specification**

'linear' (default) | character vector naming the model |  $t$ -by- $(p + 1)$  terms matrix | character vector of the form 'Y ~ terms'

Model specification, specified as one of the following.

- A character vector naming the model.

Character Vector	Model Type
'constant'	Model contains only a constant (intercept) term.
'linear'	Model contains an intercept and linear terms for each predictor.



Character Vector	Model Type
'interactions'	Model contains an intercept, linear terms, and all products of pairs of distinct predictors (no squared terms).
'purequadratic'	Model contains an intercept, linear terms, and squared terms.
'quadratic'	Model contains an intercept, linear terms, interactions, and squared terms.
'polyijk'	Model is a polynomial with all terms up to degree $i$ in the first predictor, degree $j$ in the second predictor, etc. Use numerals 0 through 9. For example, 'poly2111' has a constant plus all linear and product terms, and also contains terms with predictor 1 squared.

- $t$ -by- $(p + 1)$  matrix, namely terms matrix on page 32-2257, specifying terms to include in the model, where  $t$  is the number of terms and  $p$  is the number of predictor variables, and plus 1 is for the response variable.
- A character vector representing a formula on page 32-2260 in the form

'Y ~ terms',

where the terms are in Wilkinson Notation on page 32-2261.

Example: 'quadratic'

Example: 'y ~ X1 + X2^2 + X1:X2'

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: 'Intercept', false, 'PredictorVars', [1,3], 'ResponseVar', 5, 'RobustOpts', 'logistic' specifies a robust regression model with no constant term, where the algorithm uses the logistic weighting function with the default tuning constant, first and third variables are the predictor variables, and fifth variable is the response variable.

**CategoricalVars — Categorical variable list**

cell array of character vectors | logical or numeric index vector

Categorical variable list, specified as the comma-separated pair consisting of 'CategoricalVars' and either a cell array of character vectors containing categorical variable names in the table or dataset array `tbl`, or a logical or numeric index vector indicating which columns are categorical.

- If data is in a table or dataset array `tbl`, then, by default, `fitlm` treats all categorical values, logical values, character arrays, and cell arrays of character vectors as categorical variables.
- If data is in matrix `X`, then the default value of 'CategoricalVars' is an empty matrix `[]`. That is, no variable is categorical unless you specify it as categorical.

For example, you can specify the observations 2 and 3 out of 6 as categorical using either of the following examples.

```
Example: 'CategoricalVars', [2,3]
```

```
Example: 'CategoricalVars', logical([0 1 1 0 0 0])
```

Data Types: single | double | logical | cell

**Exclude — Observations to exclude**

logical or numeric index vector

Observations to exclude from the fit, specified as the comma-separated pair consisting of 'Exclude' and a logical or numeric index vector indicating which observations to exclude from the fit.

For example, you can exclude observations 2 and 3 out of 6 using either of the following examples.

```
Example: 'Exclude', [2,3]
```

```
Example: 'Exclude', logical([0 1 1 0 0 0])
```

Data Types: single | double | logical

**Intercept — Indicator for constant term**

true (default) | false

Indicator for the constant term (intercept) in the fit, specified as the comma-separated pair consisting of 'Intercept' and either `true` to include or `false` to remove the constant term from the model.

Use 'Intercept' only when specifying the model using a character vector, not a formula or matrix.

Example: 'Intercept', `false`

### **PredictorVars** — Predictor variables

cell array of character vectors | logical or numeric index vector

Predictor variables to use in the fit, specified as the comma-separated pair consisting of 'PredictorVars' and either a cell array of character vectors of the variable names in the table or dataset array `tbl`, or a logical or numeric index vector indicating which columns are predictor variables.

The character vectors should be among the names in `tbl`, or the names you specify using the 'VarNames' name-value pair argument.

The default is all variables in `X`, or all variables in `tbl` except for `ResponseVar`.

For example, you can specify the second and third variables as the predictor variables using either of the following examples.

Example: 'PredictorVars', `[2,3]`

Example: 'PredictorVars', `logical([0 1 1 0 0 0])`

Data Types: `single` | `double` | `logical` | `cell`

### **ResponseVar** — Response variable

last column in `tbl` (default) | character vector containing variable name | logical or numeric index vector

Response variable to use in the fit, specified as the comma-separated pair consisting of 'ResponseVar' and either a character vector containing the variable name in the table or dataset array `tbl`, or a logical or numeric index vector indicating which column is the response variable. You typically need to use 'ResponseVar' when fitting a table or dataset array `tbl`.

For example, you can specify the fourth variable, say `yield`, as the response out of six variables, in one of the following ways.

Example: `'ResponseVar', 'yield'`

Example: `'ResponseVar', [4]`

Example: `'ResponseVar', logical([0 0 0 1 0 0])`

Data Types: `single` | `double` | `logical` | `char`

### **RobustOpts** — Indicator of robust fitting type

`'off'` (default) | `'on'` | character vector | structure with character vector or function handle

Indicator of the robust fitting type to use, specified as the comma-separated pair consisting of `'RobustOpts'` and one of the following.

- `'off'` — No robust fitting. `fitlm` uses ordinary least squares.
- `'on'` — Robust fitting. When you use robust fitting, `'bisquare'` weight function is the default.
- Character vector — Name of the robust fitting weight function from the following table. `fitlm` uses the corresponding default tuning constant in the table.
- Structure with the character vector `RobustWgtFun` containing the name of the robust fitting weight function from the following table and optional scalar `Tune` fields — `fitlm` uses the `RobustWgtFun` weight function and `Tune` tuning constant from the structure. You can choose the name of the robust fitting weight function from this table. If you do not supply a `Tune` field, the fitting function uses the corresponding default tuning constant.

Weight Function	Equation	Default Tuning Constant
<code>'andrews'</code>	$w = (\text{abs}(r) < \pi) .* \sin(r) ./ r$	1.339
<code>'bisquare'</code> (default)	$w = (\text{abs}(r) < 1) .* (1 - r.^2).^2$	4.685
<code>'cauchy'</code>	$w = 1 ./ (1 + r.^2)$	2.385
<code>'fair'</code>	$w = 1 ./ (1 + \text{abs}(r))$	1.400
<code>'huber'</code>	$w = 1 ./ \max(1, \text{abs}(r))$	1.345
<code>'logistic'</code>	$w = \tanh(r) ./ r$	1.205
<code>'ols'</code>	Ordinary least squares (no weighting function)	None

Weight Function	Equation	Default Tuning Constant
'talwar'	$w = 1 * (\text{abs}(r) < 1)$	2.795
'welsch'	$w = \exp(- (r.^2))$	2.985

The value  $r$  in the weight functions is

$$r = \text{resid} / (\text{tune} * s * \sqrt{1-h}),$$

where `resid` is the vector of residuals from the previous iteration, `h` is the vector of leverage values from a least-squares fit, and `s` is an estimate of the standard deviation of the error term given by

$$s = \text{MAD} / 0.6745.$$

MAD is the median absolute deviation of the residuals from their median. The constant 0.6745 makes the estimate unbiased for the normal distribution. If there are  $p$  columns in  $X$ , the smallest  $p$  absolute deviations are excluded when computing the median.

Default tuning constants give coefficient estimates that are approximately 95% as statistically efficient as the ordinary least-squares estimates, provided the response has a normal distribution with no outliers. Decreasing the tuning constant increases the downweight assigned to large residuals; increasing the tuning constant decreases the downweight assigned to large residuals.

- Structure with the function handle `RobustWgtFun` and optional scalar `Tune` fields — You can specify a custom weight function. `fitlm` uses the `RobustWgtFun` weight function and `Tune` tuning constant from the structure. Specify `RobustWgtFun` as a function handle that accepts a vector of residuals, and returns a vector of weights the same size. The fitting function scales the residuals, dividing by the tuning constant (default 1) and by an estimate of the error standard deviation before it calls the weight function.

Example: `'RobustOpts', 'andrews'`

#### **VarNames** — Names of variables in fit

`{'x1', 'x2', ..., 'xn', 'y'}` (default) | cell array of character vectors

Names of variables in fit, specified as the comma-separated pair consisting of 'VarNames' and a cell array of character vectors including the names for the columns of  $X$  first, and the name for the response variable  $y$  last.

'VarNames' is not applicable to variables in a table or dataset array, because those variables already have names.

For example, if in your data, horsepower, acceleration, and model year of the cars are the predictor variables, and miles per gallon (MPG) is the response variable, then you can name the variables as follows.

Example: 'VarNames', {'Horsepower', 'Acceleration', 'Model\_Year', 'MPG'}

Data Types: cell

### **Weights — Observation weights**

ones( $n, 1$ ) (default) |  $n$ -by-1 vector of nonnegative scalar values

Observation weights, specified as the comma-separated pair consisting of 'Weights' and an  $n$ -by-1 vector of nonnegative scalar values, where  $n$  is the number of observations.

Data Types: single | double

## **Output Arguments**

### **mdl — Linear model**

LinearModel object

Linear model representing a least-squares fit of the response to the data, returned as a LinearModel object.

If the value of the 'RobustOpts' name-value pair is not [] or 'ols', the model is not a least-squares fit, but uses the robust fitting function.

For properties and methods of the linear model object, mdl, see the LinearModel class page.

## Definitions

### Terms Matrix

A terms matrix is a  $t$ -by- $(p + 1)$  matrix specifying terms in a model, where  $t$  is the number of terms,  $p$  is the number of predictor variables, and plus one is for the response variable.

The value of  $T(i, j)$  is the exponent of variable  $j$  in term  $i$ . Suppose there are three predictor variables A, B, and C:

```
[0 0 0 0] % Constant term or intercept
[0 1 0 0] % B; equivalently, A^0 * B^1 * C^0
[1 0 1 0] % A*C
[2 0 0 0] % A^2
[0 1 2 0] % B*(C^2)
```

The 0 at the end of each term represents the response variable. In general,

- If you have the variables in a table or dataset array, then 0 must represent the response variable depending on the position of the response variable. The following example illustrates this.

Load the sample data and define the dataset array.

```
load hospital
dsa = dataset(hospital.Sex,hospital.BloodPressure(:,1),hospital.Age,...
hospital.Smoker, 'VarNames', {'Sex', 'BloodPressure', 'Age', 'Smoker'});
```

Represent the linear model 'BloodPressure ~ 1 + Sex + Age + Smoker' in a terms matrix. The response variable is in the second column of the dataset array, so there must be a column of 0s for the response variable in the second column of the terms matrix.

```
T = [0 0 0 0;1 0 0 0;0 0 1 0;0 0 0 1]
```

```
T =
```

```

0     0     0     0
1     0     0     0
0     0     1     0
0     0     0     1
```

Redefine the dataset array.

```
dsa = dataset(hospital.BloodPressure(:,1),hospital.Sex,hospital.Age,...  
hospital.Smoker,'VarNames',{'BloodPressure','Sex','Age','Smoker'});
```

Now, the response variable is the first term in the dataset array. Specify the same linear model, 'BloodPressure ~ 1 + Sex + Age + Smoker', using a terms matrix.

```
T = [0 0 0 0;0 1 0 0;0 0 1 0;0 0 0 1]
```

```
T =
```

```
    0    0    0    0  
    0    1    0    0  
    0    0    1    0  
    0    0    0    1
```

- If you have the predictor and response variables in a matrix and column vector, then you must include 0 for the response variable at the end of each term. The following example illustrates this.

Load the sample data and define the matrix of predictors.

```
load carsmall  
X = [Acceleration,Weight];
```

Specify the model 'MPG ~ Acceleration + Weight + Acceleration:Weight + Weight^2' using a term matrix and fit the model to the data. This model includes the main effect and two-way interaction terms for the variables, Acceleration and Weight, and a second-order term for the variable, Weight.

```
T = [0 0 0;1 0 0;0 1 0;1 1 0;0 2 0]
```

```
T =
```

```
    0    0    0  
    1    0    0  
    0    1    0  
    1    1    0  
    0    2    0
```

Fit a linear model.

```
mdl = fitlm(X,MPG,T)
```



```
mdl =
```

```
Linear regression model:
```

```
y ~ 1 + x1*x2 + x2^2
```

```
Estimated Coefficients:
```

	Estimate	SE	tStat	pValue
(Intercept)	48.906	12.589	3.8847	0.00019665
x1	0.54418	0.57125	0.95261	0.34337
x2	-0.012781	0.0060312	-2.1192	0.036857
x1:x2	-0.00010892	0.00017925	-0.6076	0.545
x2^2	9.7518e-07	7.5389e-07	1.2935	0.19917

```
Number of observations: 94, Error degrees of freedom: 89
```

```
Root Mean Squared Error: 4.1
```

```
R-squared: 0.751, Adjusted R-Squared 0.739
```

```
F-statistic vs. constant model: 67, p-value = 4.99e-26
```

Only the intercept and x2 term, which correspond to the Weight variable, are significant at the 5% significance level.

Now, perform a stepwise regression with a constant model as the starting model and a linear model with interactions as the upper model.

```
T = [0 0 0;1 0 0;0 1 0;1 1 0];
```

```
mdl = stepwiselm(X,MPG,[0 0 0], 'upper', T)
```

```
1. Adding x2, FStat = 259.3087, pValue = 1.643351e-28
```

```
mdl =
```

```
Linear regression model:
```

```
y ~ 1 + x2
```

```
Estimated Coefficients:
```

	Estimate	SE	tStat	pValue
(Intercept)	49.238	1.6411	30.002	2.7015e-49
x2	-0.0086119	0.0005348	-16.103	1.6434e-28

```
Number of observations: 94, Error degrees of freedom: 92
```

```
Root Mean Squared Error: 4.13
```

```
R-squared: 0.738, Adjusted R-Squared 0.735
```

```
F-statistic vs. constant model: 259, p-value = 1.64e-28
```

The results of the stepwise regression are consistent with the results of `fitlm` in the previous step.

## Formula

A formula for model specification is a character vector of the form `'Y ~ terms'`

where

- `Y` is the response name.
- `terms` contains
  - Variable names
  - `+` means include the next variable
  - `-` means do not include the next variable
  - `:` defines an interaction, a product of terms
  - `*` defines an interaction **and all lower-order terms**
  - `^` raises the predictor to a power, exactly as in `*` repeated, so `^` includes lower order terms as well
  - `()` groups terms

---

**Note** Formulas include a constant (intercept) term by default. To exclude a constant term from the model, include `-1` in the formula.

---

For example,

`'Y ~ A + B + C'` means a three-variable linear model with intercept.

`'Y ~ A + B + C - 1'` is a three-variable linear model without intercept.

`'Y ~ A + B + C + B^2'` is a three-variable model with intercept and a `B^2` term.

`'Y ~ A + B^2 + C'` is the same as the previous example because `B^2` includes a `B` term.

`'Y ~ A + B + C + A:B'` includes an `A*B` term.

`'Y ~ A*B + C'` is the same as the previous example because `A*B = A + B + A:B`.

`'Y ~ A*B*C - A:B:C'` has all interactions among `A`, `B`, and `C`, except the three-way interaction.

' $Y \sim A*(B + C + D)$ ' has all linear terms, plus products of A with each of the other variables.

## Wilkinson Notation

Wilkinson notation describes the factors present in models. The notation relates to factors present in models, not to the multipliers (coefficients) of those factors.

Wilkinson Notation	Factors in Standard Notation
1	Constant (intercept) term
$A^k$ , where k is a positive integer	$A, A^2, \dots, A^k$
$A + B$	$A, B$
$A*B$	$A, B, A*B$
$A:B$	$A*B$ only
$-B$	Do not include B
$A*B + C$	$A, B, C, A*B$
$A + B + C + A:B$	$A, B, C, A*B$
$A*B*C - A:B:C$	$A, B, C, A*B, A*C, B*C$
$A*(B + C)$	$A, B, C, A*B, A*C$

Statistics and Machine Learning Toolbox notation always includes a constant term unless you explicitly remove the term using `-1`.

## Tips

- After training a `LinearModel` model object by using `fitlm`, use the function `predict` and the trained model object to generate C code that predicts responses for new data. For details, see “Code Generation” on page 32-0 .

## Extended Capabilities

### Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function supports tall arrays for out-of-memory data with some limitations.

- If any input argument to `fitlm` is a tall array, then all of the other inputs must be tall arrays as well. This includes nonempty variables supplied with the 'Weights' and 'Exclude' name-value pairs.
- The 'RobustOpts' name-value pair is not supported with tall arrays.
- For tall data, `fitlm` returns a `CompactLinearModel` object that contains most of the same properties as a `LinearModel` object. The main difference is that the compact object is sensitive to memory requirements. The compact object does not include properties that include the data, or that include an array of the same size as the data. The compact object does not contain these `LinearModel` properties:
  - Diagnostics
  - Fitted
  - ObservationInfo
  - ObservationNames
  - Residuals
  - Steps
  - Variables

You can compute the residuals directly from the compact object returned by `LM = fitlm(X,Y)` using

```
RES = Y - predict(LM,X);  
S = LM.RMSE;  
histogram(RES,linspace(-3*S,3*S,51))
```

- If the `CompactLinearModel` object is missing lower order terms that include categorical factors:
  - The `plotEffects` and `plotInteraction` methods are not supported.

- The anova method with the 'components' option is not supported.

For more information, see “Tall Arrays” (MATLAB).

## See Also

`LinearModel` | `predict` | `stepwiselm`

## Topics

“Examine Quality and Adjust the Fitted Model” on page 11-20

“Predict or Simulate Responses to New Data” on page 11-38

“Linear Regression Workflow” on page 11-43

“Regression with Categorical Covariates” on page 2-65

“Linear Regression” on page 11-11

**Introduced in R2013b**

## fitlme

Fit linear mixed-effects model

### Syntax

```
lme = fitlme(tbl,formula)
lme = fitlme(tbl,formula,Name,Value)
```

### Description

`lme = fitlme(tbl,formula)` returns a linear mixed-effects model, specified by `formula`, fitted to the variables in the table or dataset array `tbl`.

`lme = fitlme(tbl,formula,Name,Value)` returns a linear mixed-effects model with additional options specified by one or more `Name,Value` pair arguments.

For example, you can specify the covariance pattern of the random-effects terms, the method to use in estimating the parameters, or options for the optimization algorithm.

### Examples

#### Fit Linear Mixed-Effects Model

Load the sample data.

```
load imports-85
```

Store the variables in a table.

```
tbl = table(X(:,12),X(:,14),X(:,24),'VariableNames',{'Horsepower','CityMPG','EngineType
```

Display the first five rows of the table.

```
tbl(1:5,:)
```

```
ans =
```

```
5x3 table
```

Horsepower	CityMPG	EngineType
111	21	13
111	21	13
154	19	37
102	24	35
115	18	35

Fit a linear mixed-effects model for miles per gallon in the city, with fixed effects for horsepower, and uncorrelated random effect for intercept and horsepower grouped by the engine type.

```
lme = fitlme(tbl, 'CityMPG~Horsepower+(1|EngineType)+(Horsepower-1|EngineType)');
```

In this model, CityMPG is the response variable, horsepower is the predictor variable, and engine type is the grouping variable. The fixed-effects portion of the model corresponds to  $1 + \text{Horsepower}$ , because the intercept is included by default.

Since the random-effect terms for intercept and horsepower are uncorrelated, these terms are specified separately. Because the second random-effect term is only for horsepower, you must include a  $-1$  to eliminate the intercept from the second random-effect term.

Display the model.

```
lme
```

```
lme =
```

```
Linear mixed-effects model fit by ML
```

```
Model information:
```

Number of observations	203
Fixed effects coefficients	2
Random effects coefficients	14

Covariance parameters 3

Formula:

CityMPG ~ 1 + Horsepower + (1 | EngineType) + (Horsepower | EngineType)

Model fit statistics:

AIC	BIC	LogLikelihood	Deviance
1099.5	1116	-544.73	1089.5

Fixed effects coefficients (95% CIs):

Name	Estimate	SE	tStat	DF	pValue
'(Intercept)'	37.276	2.8556	13.054	201	1.3147e-28
'Horsepower'	-0.12631	0.02284	-5.53	201	9.8848e-08

Lower	Upper
31.645	42.906
-0.17134	-0.081269

Random effects covariance parameters (95% CIs):

Group: EngineType (7 Levels)

Name1	Name2	Type	Estimate	Lower
'(Intercept)'	'(Intercept)'	'std'	5.7338	2.3773

Upper
13.829

Group: EngineType (7 Levels)

Name1	Name2	Type	Estimate	Lower
'Horsepower'	'Horsepower'	'std'	0.050357	0.02307

Upper
0.10992

Group: Error

Name	Estimate	Lower	Upper
'Res Std'	3.226	2.9078	3.5789

Note that the random-effects covariance parameters for intercept and horsepower are separate in the display.



Now, fit a linear mixed-effects model for miles per gallon in the city, with the same fixed-effects term and potentially correlated random effect for intercept and horsepower grouped by the engine type.

```
lme2 = fitlme(tbl, 'CityMPG~Horsepower+(Horsepower|EngineType)');
```

Because the random-effect term includes the intercept by default, you do not have to add 1, the random effect term is equivalent to (1 + Horsepower|EngineType).

Display the model.

```
lme2
```

```
lme2 =
```

```
Linear mixed-effects model fit by ML
```

```
Model information:
```

Number of observations	203
Fixed effects coefficients	2
Random effects coefficients	14
Covariance parameters	4

```
Formula:
```

```
CityMPG ~ 1 + Horsepower + (1 + Horsepower | EngineType)
```

```
Model fit statistics:
```

AIC	BIC	LogLikelihood	Deviance
1089	1108.9	-538.52	1077

```
Fixed effects coefficients (95% CIs):
```

Name	Estimate	SE	tStat	DF	pValue
'(Intercept)'	33.824	4.0181	8.4178	201	7.1678e-15
'Horsepower'	-0.1087	0.032912	-3.3029	201	0.0011328

Lower	Upper
25.901	41.747
-0.1736	-0.043806

```
Random effects covariance parameters (95% CIs):
```

```
Group: EngineType (7 Levels)
```

Name1	Name2	Type	Estimate	Lower
-------	-------	------	----------	-------

```
' (Intercept) '      ' (Intercept) '      'std'      9.4952      4.7022
'Horsepower'        ' (Intercept) '      'corr'     -0.96843    -0.99568
'Horsepower'        'Horsepower'        'std'      0.078874    0.039917
```

```
Upper
  19.174
 -0.78738
  0.15585
```

Group: Error

Name	Estimate	Lower	Upper
'Res Std'	3.1845	2.8774	3.5243

Note that the random effects covariance parameters for intercept and horsepower are together in the display, and it includes the correlation ('corr') between the intercept and horsepower.

### Fit Random Intercept LME Model

Load the sample data.

```
load flu
```

The `flu` dataset array has a `Date` variable, and 10 variables containing estimated influenza rates (in 9 different regions, estimated from Google® searches, plus a nationwide estimate from the Centers for Disease Control and Prevention, CDC).

To fit a linear-mixed effects model, your data must be in a properly formatted dataset array. To fit a linear mixed-effects model with the influenza rates as the responses, combine the nine columns corresponding to the regions into an array. The new dataset array, `flu2`, must have the new response variable `FluRate`, the nominal variable `Region` that shows which region each estimate is from, the nationwide estimate `WtdILLI`, and the grouping variable `Date`.

```
flu2 = stack(flu,2:10,'NewDataVarName','FluRate',...
            'IndVarName','Region');
flu2.Date = nominal(flu2.Date);
```

Display the first six rows of `flu2`.

```
flu2(1:6,:)

```

```
ans =

```

Date	WtdILI	Region	FluRate
10/9/2005	1.182	NE	0.97
10/9/2005	1.182	MidAtl	1.025
10/9/2005	1.182	ENCentral	1.232
10/9/2005	1.182	WNCentral	1.286
10/9/2005	1.182	SAtl	1.082
10/9/2005	1.182	ESCentral	1.457

Fit a linear mixed-effects model with a fixed-effects term for the nationwide estimate,  $\text{WtdILI}$ , and a random intercept that varies by  $\text{Date}$ . The model corresponds to

$$y_{im} = \beta_0 + \beta_1 \text{WtdILI}_{im} + b_{0m} + \varepsilon_{im}, \quad i = 1, 2, \dots, 468, \quad m = 1, 2, \dots, 52,$$

where  $y_{im}$  is the observation  $i$  for level  $m$  of grouping variable  $\text{Date}$ ,  $b_{0m}$  is the random effect for level  $m$  of the grouping variable  $\text{Date}$ , and  $\varepsilon_{im}$  is the observation error for observation  $i$ . The random effect has the prior distribution,

$$b_{0m} \sim N(0, \sigma_b^2),$$

and the error term has the distribution,

$$\varepsilon_{im} \sim N(0, \sigma^2).$$

```
lme = fitlme(flu2, 'FluRate ~ 1 + WtdILI + (1|Date)')

```

```
lme =

```

```
Linear mixed-effects model fit by ML

```

```
Model information:

```

Number of observations	468
Fixed effects coefficients	2
Random effects coefficients	52
Covariance parameters	2

```
Formula:

```

```
FluRate ~ 1 + WtdILI + (1 | Date)

```

Model fit statistics:

AIC	BIC	LogLikelihood	Deviance
286.24	302.83	-139.12	278.24

Fixed effects coefficients (95% CIs):

Name	Estimate	SE	tStat	DF	pValue
'(Intercept)'	0.16385	0.057525	2.8484	466	0.0045885
'WtdILI'	0.7236	0.032219	22.459	466	3.0502e-76

Lower	Upper
0.050813	0.27689
0.66028	0.78691

Random effects covariance parameters (95% CIs):

Group: Date (52 Levels)

Name1	Name2	Type	Estimate	Lower
'(Intercept)'	'(Intercept)'	'std'	0.17146	0.13227

Upper
0.22226

Group: Error

Name	Estimate	Lower	Upper
'Res Std'	0.30201	0.28217	0.32324

Estimated covariance parameters are displayed in the section titled "Random effects covariance parameters". The estimated value of  $\sigma_b$  is 0.17146 and its 95% confidence interval is [0.13227, 0.22226]. Since this interval does not include 0, the random-effects term is significant. You can formally test the significance of any random-effects term using a likelihood ratio test via the `LinearMixedModel.compare` method.

The estimated response at an observation is the sum of the fixed effects and the random-effect value at the grouping variable level corresponding to that observation. For example, the estimated flu rate for observation 28 is

$$\begin{aligned}\hat{y}_{28} &= \hat{\beta}_0 + \hat{\beta}_1 \text{WtdILI}_{28} + \hat{b}_{10/30/2005} \\ &= 0.1639 + 0.7236 * (1.343) + 0.3318 \\ &= 1.46749,\end{aligned}$$

where  $\hat{b}$  is the estimated best linear unbiased predictor (BLUP) of the random effects for the intercept. You can compute this value as follows.

```
beta = fixedEffects(lme);
[~,~,STATS] = randomEffects(lme); % Compute the random-effects statistics (STATS)
STATS.Level = nominal(STATS.Level);
y_hat = beta(1) + beta(2)*flu2.WtdILI(28) + STATS.Estimate(STATS.Level=='10/30/2005')

y_hat =

    1.4674
```

You can display the fitted value using the `fitted` method.

```
F = fitted(lme);
F(28)

ans =

    1.4674
```

## LME Model for Randomized Block Design

Load the sample data.

```
load(fullfile(matlabroot, 'examples', 'stats', 'shift.mat'))
```

The data shows the absolute deviations from the target quality characteristic measured from the products each of five operators manufacture during three shifts: morning, evening, and night. This is a randomized block design, where the operators are the blocks. The experiment is designed to study the impact of the time of shift on the performance. The performance measure is the absolute deviations of the quality characteristics from the target value. This is simulated data.

Fit a linear mixed-effects model with a random intercept grouped by operator to assess if performance significantly differs according to the time of the shift. Use the restricted maximum likelihood method and `'effects'` contrasts.

'effects' contrasts mean that the coefficients sum to 0, and `fitlme` creates a matrix called a *fixed effects design matrix* to describe the effect of shift. This matrix has two columns, `Shift_Evening` and `Shift_Morning`, where

$$\text{Shift\_Evening} = \begin{cases} 0, & \text{if Morning} \\ 1, & \text{if Evening} \\ -1, & \text{if Night} \end{cases}$$

$$\text{Shift\_Morning} = \begin{cases} 1, & \text{if Morning} \\ 0, & \text{if Evening} \\ -1, & \text{if Night} \end{cases}$$

The model corresponds to

$$\text{Morning Shift: } \text{QCDev}_{im} = \beta_0 + \beta_2 \text{Shift\_Morning}_i + b_{0m} + \epsilon_{im},$$

$$\text{Evening Shift: } \text{QCDev}_{im} = \beta_0 + \beta_1 \text{Shift\_Evening}_i + b_{0m} + \epsilon_{im},$$

$$\text{Night Shift: } \text{QCDev}_{im} = \beta_0 - \beta_1 \text{Shift\_Evening}_i - \beta_2 \text{Shift\_Morning}_i + b_{0m} + \epsilon_{im},$$

where  $i$  represents the observations, and  $m$  represents the operators,  $i = 1, 2, \dots, 15$ , and  $m = 1, 2, \dots, 5$ . The random effects and the observation error have the following distributions:

$$b_{0m} \sim N(0, \sigma_b^2)$$

and

$$\epsilon_{im} \sim N(0, \sigma^2).$$

```
lme = fitlme(shift, 'QCDev ~ Shift + (1|Operator)', ...
'FitMethod', 'REML', 'DummyVarCoding', 'effects')
```

```
lme =
```

```
Linear mixed-effects model fit by REML
```

```
Model information:
```

Number of observations	15
Fixed effects coefficients	3
Random effects coefficients	5
Covariance parameters	2

```
Formula:
```

```
QCDev ~ 1 + Shift + (1 | Operator)
```

Model fit statistics:

AIC	BIC	LogLikelihood	Deviance
58.913	61.337	-24.456	48.913

Fixed effects coefficients (95% CIs):

Name	Estimate	SE	tStat	DF	pValue
'(Intercept)'	3.6525	0.94109	3.8812	12	0.0021832
'Shift_Evening'	-0.53293	0.31206	-1.7078	12	0.11339
'Shift_Morning'	-0.91973	0.31206	-2.9473	12	0.012206

Lower	Upper
1.6021	5.703
-1.2129	0.14699
-1.5997	-0.23981

Random effects covariance parameters (95% CIs):

Group: Operator (5 Levels)

Name1	Name2	Type	Estimate	Lower
'(Intercept)'	'(Intercept)'	'std'	2.0457	0.98207

Upper
4.2612

Group: Error

Name	Estimate	Lower	Upper
'Res Std'	0.85462	0.52357	1.395

Compute the best linear unbiased predictor (BLUP) estimates of random effects.

```
B = randomEffects(lme)
```

```
B =
```

```

0.5775
1.1757
-2.1715
2.3655
-1.9472
```

The estimated absolute deviation from the target quality characteristics for the third operator working the evening shift is

$$\begin{aligned}\hat{y}_{\text{Evening,Operator}_3} &= \hat{\beta}_0 + \hat{\beta}_1 \text{Shift\_Evening} + \hat{b}_{03} \\ &= 3.6525 - 0.53293 - 2.1715 \\ &= 0.94807.\end{aligned}$$

You can also display this value as follows.

```
F = fitted(lme);  
F(shift.Shift=='Evening' & shift.Operator=='3')
```

```
ans =  
  
    0.9481
```

Similarly, you can calculate the estimated absolute deviation from the target quality characteristics for the third operator working the morning shift as

$$\begin{aligned}\hat{y}_{\text{Morning,Operator}_3} &= \hat{\beta}_0 + \hat{\beta}_2 \text{Shift\_Morning} + \hat{b}_{03} \\ &= 3.6525 - 0.91973 - 2.1715 \\ &= 0.56127.\end{aligned}$$

You can also display this value as follows.

```
F(shift.Shift=='Morning' & shift.Operator=='3')
```

```
ans =  
  
    0.5613
```

The operator tends to make a smaller magnitude of error during the morning shift.

### LME Model for Split-Plot Experiment

Load the sample data.

```
load(fullfile(matlabroot, 'examples', 'stats', 'fertilizer.mat'))
```



The dataset array includes data from a split-plot experiment, where soil is divided into three blocks based on the soil type: sandy, silty, and loamy. Each block is divided into five plots, where five types of tomato plants (cherry, heirloom, grape, vine, and plum) are randomly assigned to these plots. The tomato plants in the plots are then divided into subplots, where each subplot is treated by one of four fertilizers. This is simulated data.

Store the data in a dataset array called `ds`, and define `Tomato`, `Soil`, and `Fertilizer` as categorical variables.

```
ds = fertilizer;
ds.Tomato = nominal(ds.Tomato);
ds.Soil = nominal(ds.Soil);
ds.Fertilizer = nominal(ds.Fertilizer);
```

Fit a linear mixed-effects model, where `Fertilizer` and `Tomato` are the fixed-effects variables, and the mean yield varies by the block (soil type) and the plots within blocks (tomato types within soil types) independently.

This model corresponds to

$$y_{imjk} = \beta_0 + \sum_{m=2}^4 \beta_{1m} I[F]_{im} + \sum_{j=2}^5 \beta_{2j} I[T]_{ij} + \sum_{j=2}^5 \sum_{m=2}^4 \beta_{3mj} I[F]_{im} I[T]_{ij} + b_{0k} S_k + b_{0jk} (S * T)_{jk} + \epsilon_{imjk},$$

where  $i = 1, 2, \dots, 60$ , index  $m$  corresponds to the fertilizer types,  $j$  corresponds to the tomato types, and  $k = 1, 2, 3$  corresponds to the blocks (soil).  $S_k$  represents the  $k$ th soil type, and  $(S * T)_{jk}$  represents the  $j$ th tomato type nested in the  $k$ th soil type.  $I[F]_{im}$  is the dummy variable representing level  $m$  of the fertilizer. Similarly,  $I[T]_{ij}$  is the dummy variable representing level  $j$  of the tomato type.

The random effects and observation error have these prior distributions:  $b_{0k} \sim N(0, \sigma_S^2)$ ,  $b_{0jk} \sim N(0, \sigma_{S*T}^2)$ , and  $\epsilon_{imjk} \sim N(0, \sigma^2)$ .

```
lme = fitlme(ds, 'Yield ~ Fertilizer * Tomato + (1|Soil) + (1|Soil:Tomato)')
```

```
lme =
```

```
Linear mixed-effects model fit by ML
```

Model information:

Number of observations	60
Fixed effects coefficients	20
Random effects coefficients	18
Covariance parameters	3

Formula:

Yield ~ 1 + Tomato\*Fertilizer + (1 | Soil) + (1 | Soil:Tomato)

Model fit statistics:

AIC	BIC	LogLikelihood	Deviance
522.57	570.74	-238.29	476.57

Fixed effects coefficients (95% CIs):

Name	Estimate	SE	tStat	DF
'(Intercept)'	77	8.5836	8.9706	40
'Tomato_Grape'	-16	11.966	-1.3371	40
'Tomato_Heirloom'	-6.6667	11.966	-0.55714	40
'Tomato_Plum'	32.333	11.966	2.7022	40
'Tomato_Vine'	-13	11.966	-1.0864	40
'Fertilizer_2'	34.667	8.572	4.0442	40
'Fertilizer_3'	33.667	8.572	3.9275	40
'Fertilizer_4'	47.667	8.572	5.5607	40
'Tomato_Grape:Fertilizer_2'	-2.6667	12.123	-0.21997	40
'Tomato_Heirloom:Fertilizer_2'	-8	12.123	-0.65992	40
'Tomato_Plum:Fertilizer_2'	-15	12.123	-1.2374	40
'Tomato_Vine:Fertilizer_2'	-16	12.123	-1.3198	40
'Tomato_Grape:Fertilizer_3'	16.667	12.123	1.3748	40
'Tomato_Heirloom:Fertilizer_3'	3.3333	12.123	0.27497	40
'Tomato_Plum:Fertilizer_3'	3.6667	12.123	0.30246	40
'Tomato_Vine:Fertilizer_3'	3	12.123	0.24747	40
'Tomato_Grape:Fertilizer_4'	13.333	12.123	1.0999	40
'Tomato_Heirloom:Fertilizer_4'	-19	12.123	-1.5673	40
'Tomato_Plum:Fertilizer_4'	-2.6667	12.123	-0.21997	40
'Tomato_Vine:Fertilizer_4'	8.6667	12.123	0.71492	40

pValue	Lower	Upper
4.0206e-11	59.652	94.348
0.18873	-40.184	8.1837
0.58053	-30.85	17.517
0.010059	8.1496	56.517
0.28379	-37.184	11.184
0.00023272	17.342	51.991

0.00033057	16.342	50.991
1.9567e-06	30.342	64.991
0.82701	-27.167	21.834
0.51309	-32.501	16.501
0.22317	-39.501	9.5007
0.19439	-40.501	8.5007
0.17683	-7.8341	41.167
0.78476	-21.167	27.834
0.76387	-20.834	28.167
0.80581	-21.501	27.501
0.27796	-11.167	37.834
0.12492	-43.501	5.5007
0.82701	-27.167	21.834
0.47881	-15.834	33.167

Random effects covariance parameters (95% CIs):

Group: Soil (3 Levels)

Name1	Name2	Type	Estimate	Lower
'(Intercept)'	'(Intercept)'	'std'	2.5028	0.02771

Upper  
226.05

Group: Soil:Tomato (15 Levels)

Name1	Name2	Type	Estimate	Lower
'(Intercept)'	'(Intercept)'	'std'	10.225	6.1497

Upper  
17.001

Group: Error

Name	Estimate	Lower	Upper
'Res Std'	10.499	8.5389	12.908

The *P*-values corresponding to the last 12 rows in the fixed-effects coefficients display (0.82701 to 0.47881) indicate that interaction coefficients between the tomato and fertilizer types are not significant. To test for the overall interaction between tomato and fertilizer, use the `LinearMixedModel.anova` method after refitting the model using 'effects' contrasts.

The confidence interval for the standard deviations of the random-effects terms ( $\sigma_s^2$ ), where the intercept is grouped by soil, is very large. This term does not appear significant.

Refit the model after removing the interaction term `Tomato:Fertilizer` and the random-effects term `(1 | Soil)`.

```
lme = fitlme(ds, 'Yield ~ Fertilizer + Tomato + (1|Soil:Tomato)')
```

```
lme =
```

```
Linear mixed-effects model fit by ML
```

```
Model information:
```

Number of observations	60
Fixed effects coefficients	8
Random effects coefficients	15
Covariance parameters	2

```
Formula:
```

```
Yield ~ 1 + Tomato + Fertilizer + (1 | Soil:Tomato)
```

```
Model fit statistics:
```

AIC	BIC	LogLikelihood	Deviance
511.06	532	-245.53	491.06

```
Fixed effects coefficients (95% CIs):
```

Name	Estimate	SE	tStat	DF	pValue
'(Intercept)'	77.733	7.3293	10.606	52	1.3108e-14
'Tomato_Grape'	-9.1667	9.6045	-0.95441	52	0.34429
'Tomato_Heirloom'	-12.583	9.6045	-1.3102	52	0.1959
'Tomato_Plum'	28.833	9.6045	3.0021	52	0.0041138
'Tomato_Vine'	-14.083	9.6045	-1.4663	52	0.14858
'Fertilizer_2'	26.333	4.5004	5.8514	52	3.3024e-07
'Fertilizer_3'	39	4.5004	8.6659	52	1.1459e-11
'Fertilizer_4'	47.733	4.5004	10.607	52	1.308e-14

Lower	Upper
63.026	92.441
-28.439	10.106
-31.856	6.6895

```

9.5605    48.106
-33.356    5.1895
17.303    35.364
29.969    48.031
38.703    56.764

```

Random effects covariance parameters (95% CIs):

Group: Soil:Tomato (15 Levels)

Name1	Name2	Type	Estimate	Lower
'(Intercept)'	'(Intercept)'	'std'	10.02	6.0812

```

Upper
16.509

```

Group: Error

Name	Estimate	Lower	Upper
'Res Std'	12.325	10.024	15.153

You can compare the two models using the `LinearMixedModel.compare` method with the simulated likelihood ratio test since both a fixed-effect and a random-effect term are tested.

## Longitudinal Study with a Covariate

Load the sample data.

```
load(fullfile(matlabroot, 'examples', 'stats', 'weight.mat'))
```

`weight` contains data from a longitudinal study, where 20 subjects are randomly assigned to 4 exercise programs (A, B, C, D), and their weight loss is recorded over six 2-week time periods. This is simulated data.

Store the data in a table. Define `Subject` and `Program` as categorical variables.

```
tbl = table(InitialWeight, Program, Subject, Week, y);
tbl.Subject = nominal(tbl.Subject);
tbl.Program = nominal(tbl.Program);
```

Fit a linear mixed-effects model where the initial weight, type of program, week, and the interaction between the week and type of program are the fixed effects. The intercept and week vary by subject.

`fitlme` uses program A as a reference and creates the necessary dummy variables  $I[\cdot]$ . Since the model already has an intercept, `fitlme` only creates dummy variables for programs B, C, and D. This is also known as the 'reference' method of coding dummy variables.

This model corresponds to

$$y_{im} = \beta_0 + \beta_1 IW_i + \beta_2 \text{Week}_i + \beta_3 I[\text{PB}]_i + \beta_4 I[\text{PC}]_i + \beta_5 I[\text{PD}]_i \\ + \beta_6 (\text{Week}_i * I[\text{PB}]_i) + \beta_7 (\text{Week}_i * I[\text{PC}]_i) + \beta_8 (\text{Week}_i * I[\text{PD}]_i) \\ + b_{0m} + b_{1m} \text{Week}_{im} + \varepsilon_{im},$$

where  $i = 1, 2, \dots, 120$ , and  $m = 1, 2, \dots, 20$ .  $\beta_j$  are the fixed-effects coefficients,  $j = 0, 1, \dots, 8$ , and  $b_{0m}$  and  $b_{1m}$  are random effects.  $IW$  stands for initial weight and  $I[\cdot]$  is a dummy variable representing a type of program. For example,  $I[\text{PB}]_i$  is the dummy variable representing program type B. The random effects and observation error have the following prior distributions:

$$b_{0m} \sim N(0, \sigma_0^2)$$

$$b_{1m} \sim N(0, \sigma_1^2)$$

$$\varepsilon_{im} \sim N(0, \sigma^2).$$

```
lme = fitlme(tbl, 'y ~ InitialWeight + Program*Week + (Week|Subject)')
```

```
lme =
```

```
Linear mixed-effects model fit by ML
```

```
Model information:
```

Number of observations	120
Fixed effects coefficients	9
Random effects coefficients	40
Covariance parameters	4

```
Formula:
```

```
y ~ 1 + InitialWeight + Program*Week + (1 + Week | Subject)
```

```
Model fit statistics:
```

AIC	BIC	LogLikelihood	Deviance
-22.981	13.257	24.49	-48.981

Fixed effects coefficients (95% CIs):

Name	Estimate	SE	tStat	DF
'(Intercept)'	0.66105	0.25892	2.5531	111
'InitialWeight'	0.0031879	0.0013814	2.3078	111
'Program_B'	0.36079	0.13139	2.746	111
'Program_C'	-0.033263	0.13117	-0.25358	111
'Program_D'	0.11317	0.13132	0.86175	111
'Week'	0.1732	0.067454	2.5677	111
'Program_B:Week'	0.038771	0.095394	0.40644	111
'Program_C:Week'	0.030543	0.095394	0.32018	111
'Program_D:Week'	0.033114	0.095394	0.34713	111

pValue	Lower	Upper
0.012034	0.14798	1.1741
0.022863	0.00045067	0.0059252
0.0070394	0.10044	0.62113
0.80029	-0.29319	0.22666
0.39068	-0.14706	0.3734
0.011567	0.039536	0.30686
0.68521	-0.15026	0.2278
0.74944	-0.15849	0.21957
0.72915	-0.15592	0.22214

Random effects covariance parameters (95% CIs):

Group: Subject (20 Levels)

Name1	Name2	Type	Estimate	Lower
'(Intercept)'	'(Intercept)'	'std'	0.18407	0.12281
'Week'	'(Intercept)'	'corr'	0.66841	0.21077
'Week'	'Week'	'std'	0.15033	0.11004

Upper  
0.27587  
0.88573  
0.20537

Group: Error

Name	Estimate	Lower	Upper
'Res Std'	0.10261	0.087882	0.11981

The *P*-values 0.022863 and 0.011567 indicate significant effects of subject initial weights and time in the amount of weight lost. The weight loss of subjects who are in program B

is significantly different relative to the weight loss of subjects who are in program A. The lower and upper limits of the covariance parameters for the random effects do not include 0, thus they are significant. You can also test the significance of the random effects using the `compare` method.

## Input Arguments

### **tbl** — Input data

table | dataset array

Input data, which includes the response variable, predictor variables, and grouping variables, specified as a table or dataset array. The predictor variables can be continuous or grouping variables (see “Grouping Variables” on page 2-59). You must specify the model for the variables using `formula`.

Data Types: table

### **formula** — Formula for model specification

character vector of the form `'y ~ fixed + (random1|grouping1) + ... + (randomR|groupingR)'`

Formula for model specification, specified as a character vector of the form `'y ~ fixed + (random1|grouping1) + ... + (randomR|groupingR)'`. The formula is case sensitive. For a full description, see “Formula” on page 32-2289.

Example: `'y ~ treatment + (1|block)'`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of Name, Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as Name1, Value1, ..., NameN, ValueN.

Example:

```
'CovariancePattern', 'Diagonal', 'Optimizer', 'fminunc', 'OptimizerOptions', opt
```

specifies a model, where the random-effects terms have a diagonal covariance matrix structure, and `fitlme` uses the `fminunc` optimization algorithm with the custom optimization parameters defined in variable `opt`.



**CovariancePattern — Pattern of covariance matrix**

'FullCholesky' (default) | character vector | square symmetric logical matrix | cell array of character vectors or logical matrices

Pattern of the covariance matrix of the random effects, specified as the comma-separated pair consisting of 'CovariancePattern' and a character vector, a square symmetric logical matrix, or a cell array of character vectors or logical matrices.

If there are  $R$  random-effects terms, then the value of 'CovariancePattern' must be a cell array of length  $R$ , where each element  $r$  of this cell array specifies the pattern of the covariance matrix of the random-effects vector associated with the  $r$ th random-effects term. The options for each element follow.

'FullCholesky'	Default. Full covariance matrix using the Cholesky parameterization. <code>fitlme</code> estimates all elements of the covariance matrix.
'Full'	Full covariance matrix, using the log-Cholesky parameterization. <code>fitlme</code> estimates all elements of the covariance matrix.
'Diagonal'	Diagonal covariance matrix. That is, off-diagonal elements of the covariance matrix are constrained to be 0.

$$\begin{pmatrix} \sigma_{b1}^2 & 0 & 0 \\ 0 & \sigma_{b2}^2 & 0 \\ 0 & 0 & \sigma_{b3}^2 \end{pmatrix}$$

'Isotropic'

Diagonal covariance matrix with equal variances. That is, off-diagonal elements of the covariance matrix are constrained to be 0, and the diagonal elements are constrained to be equal. For example, if there are three random-effects terms with an isotropic covariance structure, this covariance matrix looks like

$$\begin{pmatrix} \sigma_b^2 & 0 & 0 \\ 0 & \sigma_b^2 & 0 \\ 0 & 0 & \sigma_b^2 \end{pmatrix}$$

where  $\sigma_b^2$  is the common variance of the random-effects terms.

'CompSymm'

Compound symmetry structure. That is, common variance along diagonals and equal correlation between all random effects. For example, if there are three random-effects terms with a covariance matrix having a compound symmetry structure, this covariance matrix looks like

$$\begin{pmatrix} \sigma_{b1}^2 & \sigma_{b1,b2} & \sigma_{b1,b2} \\ \sigma_{b1,b2} & \sigma_{b1}^2 & \sigma_{b1,b2} \\ \sigma_{b1,b2} & \sigma_{b1,b2} & \sigma_{b1}^2 \end{pmatrix}$$

where  $\sigma_{b1}^2$  is the common variance of the random-effects terms and  $\sigma_{b1,b2}$  is the common covariance between any two random-effects term .

**PAT** Square symmetric logical matrix. If 'CovariancePattern' is defined by the matrix PAT, and if PAT(a,b) = false, then the (a,b) element of the corresponding covariance matrix is constrained to be 0.

Example: 'CovariancePattern', 'Diagonal'

Example: 'CovariancePattern', {'Full', 'Diagonal'}

### **FitMethod** — Method for estimating parameters

'ML' (default) | 'REML'

Method for estimating parameters of the linear mixed-effects model, specified as the comma-separated pair consisting of 'FitMethod' and either of the following.

'ML'	Default. Maximum likelihood estimation
'REML'	Restricted maximum likelihood estimation

Example: 'FitMethod', 'REML'

### **Weights** — Observation weights

vector of scalar values

Observation weights, specified as the comma-separated pair consisting of 'Weights' and a vector of length  $n$ , where  $n$  is the number of observations.

Data Types: single | double

### **Exclude** — Indices for rows to exclude

use all rows without NaNs (default) | vector of integer or logical values

Indices for rows to exclude from the linear mixed-effects model in the data, specified as the comma-separated pair consisting of 'Exclude' and a vector of integer or logical values.

For example, you can exclude the 13th and 67th rows from the fit as follows.

Example: 'Exclude', [13, 67]

Data Types: single | double | logical

**DummyVarCoding** — Coding to use for dummy variables`'reference'` (default) | `'effects'` | `'full'`

Coding to use for dummy variables created from the categorical variables, specified as the comma-separated pair consisting of `'DummyVarCoding'` and one of the following.

Value	Description
<code>'reference'</code>	Default. Coefficient for first category set to 0.
<code>'effects'</code>	Coefficients sum to 0.
<code>'full'</code>	One dummy variable for each category.

Example: `'DummyVarCoding','effects'`

**Optimizer** — Optimization algorithm`'quasinevton'` (default) | `'fminunc'`

Optimization algorithm, specified as the comma-separated pair consisting of `'Optimizer'` and either of the following.

`'quasinevton'`

Default. Uses a trust region based quasi-Newton optimizer. Change the options of the algorithm using `statset('LinearMixedModel')`. If you don't specify the options, then `LinearMixedModel` uses the default options of `statset('LinearMixedModel')`.

`'fminunc'`

You must have Optimization Toolbox to specify this option. Change the options of the algorithm using `optimoptions('fminunc')`. If you don't specify the options, then `LinearMixedModel` uses the default options of `optimoptions('fminunc')` with `'Algorithm'` set to `'quasinevton'`.

Example: `'Optimizer','fminunc'`

**OptimizerOptions — Options for optimization algorithm**

structure returned by `statset` | object returned by `optimoptions`

Options for the optimization algorithm, specified as the comma-separated pair consisting of 'OptimizerOptions' and a structure returned by `statset('LinearMixedModel')` or an object returned by `optimoptions('fminunc')`.

- If 'Optimizer' is 'fminunc', then use `optimoptions('fminunc')` to change the options of the optimization algorithm. See `optimoptions` for the options 'fminunc' uses. If 'Optimizer' is 'fminunc' and you do not supply 'OptimizerOptions', then the default for `LinearMixedModel` is the default options created by `optimoptions('fminunc')` with 'Algorithm' set to 'quasi-newton'.
- If 'Optimizer' is 'quasnewton', then use `statset('LinearMixedModel')` to change the optimization parameters. If you don't change the optimization parameters, then `LinearMixedModel` uses the default options created by `statset('LinearMixedModel')`:

The 'quasnewton' optimizer uses the following fields in the structure created by `statset('LinearMixedModel')`.

**TolFun — Relative tolerance on gradient of objective function**

1e-6 (default) | positive scalar value

Relative tolerance on the gradient of the objective function, specified as a positive scalar value.

**TolX — Absolute tolerance on step size**

1e-12 (default) | positive scalar value

Absolute tolerance on the step size, specified as a positive scalar value.

**MaxIter — Maximum number of iterations allowed**

10000 (default) | positive scalar value

Maximum number of iterations allowed, specified as a positive scalar value.

**Display — Level of display**

'off' (default) | 'iter' | 'final'

Level of display, specified as one of 'off', 'iter', or 'final'.

**StartMethod** — Method to start iterative optimization`'default'` (default) | `'random'`

Method to start iterative optimization, specified as the comma-separated pair consisting of `'StartMethod'` and either of the following.

Value	Description
<code>'default'</code>	An internally defined default value
<code>'random'</code>	A random initial value

Example: `'StartMethod', 'random'`

**Verbose** — Indicator to display optimization process on screen`false` (default) | `true`

Indicator to display the optimization process on screen, specified as the comma-separated pair consisting of `'Verbose'` and either `false` or `true`. Default is `false`.

The setting for `'Verbose'` overrides the field `'Display'` in `'OptimizerOptions'`.

Example: `'Verbose', true`

**CheckHessian** — Indicator to check positive definiteness of Hessian`false` (default) | `true`

Indicator to check the positive definiteness of the Hessian of the objective function with respect to unconstrained parameters at convergence, specified as the comma-separated pair consisting of `'CheckHessian'` and either `false` or `true`. Default is `false`.

Specify `'CheckHessian'` as `true` to verify optimality of the solution or to determine if the model is overparameterized in the number of covariance parameters.

Example: `'CheckHessian', true`

## Output Arguments

**lme** — Linear mixed-effects model`LinearMixedModel` object

Linear mixed-effects model, returned as a `LinearMixedModel` object.

## Definitions

### Formula

In general, a formula for model specification is a character vector of the form 'y ~ terms'. For the linear mixed-effects models, this formula is in the form 'y ~ fixed + (random1|grouping1) + ... + (randomR|groupingR)', where `fixed` and `random` contain the fixed-effects and the random-effects terms.

Suppose a table `tbl` contains the following:

- A response variable, `y`
- Predictor variables, `Xj`, which can be continuous or grouping variables
- Grouping variables, `g1`, `g2`, ..., `gR`,

where the grouping variables in `Xj` and `gr` can be categorical, logical, character arrays, or cell arrays of character vectors.

Then, in a formula of the form, 'y ~ fixed + (random<sub>1</sub>|g<sub>1</sub>) + ... + (random<sub>R</sub>|g<sub>R</sub>)', the term `fixed` corresponds to a specification of the fixed-effects design matrix  $X$ , `random1` is a specification of the random-effects design matrix  $Z_1$  corresponding to grouping variable `g1`, and similarly `randomR` is a specification of the random-effects design matrix  $Z_R$  corresponding to grouping variable `gR`. You can express the `fixed` and `random` terms using Wilkinson notation.

Wilkinson notation describes the factors present in models. The notation relates to factors present in models, not to the multipliers (coefficients) of those factors.

Wilkinson Notation	Factors in Standard Notation
1	Constant (intercept) term
X <sup>k</sup> , where k is a positive integer	X, X <sup>2</sup> , ..., X <sup>k</sup>
X1 + X2	X1, X2
X1*X2	X1, X2, X1.*X2 (elementwise multiplication of X1 and X2)
X1:X2	X1.*X2 only
- X2	Do not include X2

Wilkinson Notation	Factors in Standard Notation
$X1 * X2 + X3$	X1, X2, X3, X1 * X2
$X1 + X2 + X3 + X1 : X2$	X1, X2, X3, X1 * X2
$X1 * X2 * X3 - X1 : X2 : X3$	X1, X2, X3, X1 * X2, X1 * X3, X2 * X3
$X1 * (X2 + X3)$	X1, X2, X3, X1 * X2, X1 * X3

Statistics and Machine Learning Toolbox notation always includes a constant term unless you explicitly remove the term using `-1`. Here are some examples for linear mixed-effects model specification.

### Examples:

Formula	Description
'y ~ X1 + X2'	Fixed effects for the intercept, X1 and X2. This is equivalent to 'y ~ 1 + X1 + X2'.
'y ~ -1 + X1 + X2'	No intercept and fixed effects for X1 and X2. The implicit intercept term is suppressed by including <code>-1</code> .
'y ~ 1 + (1   g1)'	Fixed effects for the intercept plus random effect for the intercept for each level of the grouping variable <code>g1</code> .
'y ~ X1 + (1   g1)'	Random intercept model with a fixed slope.
'y ~ X1 + (X1   g1)'	Random intercept and slope, with possible correlation between them. This is equivalent to 'y ~ 1 + X1 + (1 + X1   g1)'.
'y ~ X1 + (1   g1) + (-1 + X1   g1)'	Independent random effects terms for intercept and slope.
'y ~ 1 + (1   g1) + (1   g2) + (1   g1:g2)'	Random intercept model with independent main effects for <code>g1</code> and <code>g2</code> , plus an independent interaction effect.

## Cholesky Parameterization

One of the assumptions of linear mixed-effects models is that the random effects have the following prior distribution.



$$b \sim N(0, \sigma^2 D(\theta)),$$

where  $D$  is a  $q$ -by- $q$  symmetric and positive semidefinite matrix, parameterized by a variance component vector  $\theta$ ,  $q$  is the number of variables in the random-effects term, and  $\sigma^2$  is the observation error variance. Since the covariance matrix of the random effects,  $D$ , is symmetric, it has  $q(q+1)/2$  free parameters. Suppose  $L$  is the lower triangular Cholesky factor of  $D(\theta)$  such that

$$D(\theta) = L(\theta)L(\theta)^T,$$

then the  $q^*(q+1)/2$ -by-1 unconstrained parameter vector  $\theta$  is formed from elements in the lower triangular part of  $L$ .

For example, if

$$L = \begin{bmatrix} L_{11} & 0 & 0 \\ L_{21} & L_{22} & 0 \\ L_{31} & L_{32} & L_{33} \end{bmatrix},$$

then

$$\theta = \begin{bmatrix} L_{11} \\ L_{21} \\ L_{31} \\ L_{22} \\ L_{32} \\ L_{33} \end{bmatrix}.$$

## Log-Cholesky Parameterization

When the diagonal elements of  $L$  in Cholesky parameterization are constrained to be positive, then the solution for  $L$  is unique. Log-Cholesky parameterization is the same as Cholesky parameterization except that the logarithm of the diagonal elements of  $L$  are used to guarantee unique parameterization.

For example, for the 3-by-3 example in Cholesky parameterization, enforcing  $L_{ii} \geq 0$ ,

$$\theta = \begin{bmatrix} \log(L_{11}) \\ L_{21} \\ L_{31} \\ \log(L_{22}) \\ L_{32} \\ \log(L_{33}) \end{bmatrix}.$$

## Alternatives

If your model is not easily described using a formula, you can create matrices to define the fixed and random effects, and fit the model using `fitlmematrix(X, y, Z, G)`.

## References

- [1] Pinheiro, J. C., and D. M. Bates. “Unconstrained Parametrizations for Variance-Covariance Matrices”. *Statistics and Computing*, Vol. 6, 1996, pp. 289–296.

## See Also

`LinearMixedModel` | `fitlmematrix`

Introduced in R2013b

# fitlmematrix

Fit linear mixed-effects model

## Syntax

```
lme = fitlmematrix(X,y,Z,[])  
lme = fitlmematrix(X,y,Z,G)  
lme = fitlmematrix(____,Name,Value)
```

## Description

`lme = fitlmematrix(X,y,Z,[])` creates a linear mixed-effects model of the responses `y` using the fixed-effects design matrix `X` and random-effects design matrix or matrices in `Z`.

`[]` implies that there is one group. That is, the grouping variable `G` is ones ( $n, 1$ ), where  $n$  is the number of observations. Using `fitlmematrix(X,Y,Z,[])` without a specified covariance pattern most likely results in a nonidentifiable model. This syntax is recommended only if you build the grouping information into the random effects design `Z` and specify a covariance pattern for the random effects using the 'CovariancePattern' name-value pair argument.

`lme = fitlmematrix(X,y,Z,G)` creates a linear mixed-effects model of the responses `y` using the fixed-effects design matrix `X` and random-effects design matrix `Z` or matrices in `Z`, and the grouping variable or variables in `G`.

`lme = fitlmematrix(____,Name,Value)` also creates a linear mixed-effects model with additional options specified by one or more `Name, Value` pair arguments, using any of the previous input arguments.

For example, you can specify the names of the response, predictor, and grouping variables. You can also specify the covariance pattern, fitting method, or the optimization algorithm.

## Examples

### No Grouping Variable Specified

Load the sample data.

```
load carsmall
```

Fit a linear mixed-effects model, where miles per gallon (MPG) is the response, weight is the predictor variable, and the intercept varies by model year. First, define the design matrices. Then, fit the model using the specified design matrices.

```
y = MPG;
X = [ones(size(Weight)), Weight];
Z = ones(size(y));
lme = fitlmematrix(X,y,Z,Model_Year)
```

```
lme =
```

```
Linear mixed-effects model fit by ML
```

```
Model information:
```

Number of observations	94
Fixed effects coefficients	2
Random effects coefficients	3
Covariance parameters	2

```
Formula:
```

```
y ~ x1 + x2 + (z11 | g1)
```

```
Model fit statistics:
```

AIC	BIC	LogLikelihood	Deviance
486.09	496.26	-239.04	478.09

```
Fixed effects coefficients (95% CIs):
```

Name	Estimate	SE	tStat	DF	pValue
'x1'	43.575	2.3038	18.915	92	1.8371e-33
'x2'	-0.0067097	0.0004242	-15.817	92	5.5373e-28

```
Lower Upper
```

```

          39          48.151
-0.0075522    -0.0058672

```

Random effects covariance parameters (95% CIs):

Group: g1 (3 Levels)

Name1	Name2	Type	Estimate	Lower	Upper
'z11'	'z11'	'std'	3.301	1.4448	7.5421

Group: Error

Name	Estimate	Lower	Upper
'Res Std'	2.8997	2.5075	3.3532

Now, fit the same model by building the grouping into the Z matrix.

```

Z = double([Model_Year==70, Model_Year==76, Model_Year==82]);
lme = fitlmematrix(X,y,Z,[],'Covariancepattern','Isotropic')

```

lme =

Linear mixed-effects model fit by ML

Model information:

Number of observations	94
Fixed effects coefficients	2
Random effects coefficients	3
Covariance parameters	2

Formula:

```

y ~ x1 + x2 + (z11 + z12 + z13 | g1)

```

Model fit statistics:

AIC	BIC	LogLikelihood	Deviance
486.09	496.26	-239.04	478.09

Fixed effects coefficients (95% CIs):

Name	Estimate	SE	tStat	DF	pValue
'x1'	43.575	2.3038	18.915	92	1.8371e-33
'x2'	-0.0067097	0.0004242	-15.817	92	5.5373e-28

Lower	Upper
39	48.151

```
-0.0075522    -0.0058672
```

```
Random effects covariance parameters (95% CIs):
```

```
Group: g1 (1 Levels)
```

Name1	Name2	Type	Estimate	Lower	Upper
'z11'	'z11'	'std'	3.301	1.4448	7.5421

```
Group: Error
```

Name	Estimate	Lower	Upper
'Res Std'	2.8997	2.5075	3.3532

### Longitudinal Study with a Covariate

Load the sample data.

```
load(fullfile(matlabroot, 'examples', 'stats', 'weight.mat'));
```

`weight` contains data from a longitudinal study, where 20 subjects are randomly assigned 4 exercise programs (A, B, C, D) and their weight loss is recorded over six 2-week time periods. This is simulated data.

Define `Subject` and `Program` as categorical variables. Create the design matrices for a linear mixed-effects model, with the initial weight, type of program, week, and the interaction between the week and type of program as the fixed effects. The intercept and coefficient of week vary by subject.

This model corresponds to

$$y_{im} = \beta_0 + \beta_1 IW_i + \beta_2 \text{Week}_i + \beta_3 I[\text{PB}]_i + \beta_4 I[\text{PC}]_i + \beta_5 I[\text{PD}]_i \\ + \beta_6 (\text{Week}_i * I[\text{PB}]_i) + \beta_7 (\text{Week}_i * I[\text{PC}]_i) + \beta_8 (\text{Week}_i * I[\text{PD}]_i) \\ + b_{0m} + b_{1m} \text{Week}_{im} + \varepsilon_{im},$$

where  $i = 1, 2, \dots, 120$ , and  $m = 1, 2, \dots, 20$ .  $\beta_j$  are the fixed-effects coefficients,  $j = 0, 1, \dots, 8$ , and  $b_{0m}$  and  $b_{1m}$  are random effects. `IW` stands for initial weight and  $I[\cdot]$  is a dummy variable representing a type of program. For example,  $I[\text{PB}]_i$  is the dummy variable representing program type B. The random effects and observation error have the following prior distributions:

$$b_{0m} \sim N(0, \sigma_0^2)$$

$$b_{1m} \sim N(0, \sigma_1^2)$$

$$\varepsilon_{im} \sim N(0, \sigma^2).$$

```
Subject = nominal(Subject);
Program = nominal(Program);
D = dummyvar(Program); % Create dummy variables for Program
X = [ones(120,1), InitialWeight, D(:,2:4), Week, ...
     D(:,2).*Week, D(:,3).*Week, D(:,4).*Week];
Z = [ones(120,1), Week];
G = Subject;
```

Since the model has an intercept, you only need the dummy variables for programs B, C, and D. This is also known as the 'reference' method of coding dummy variables.

Fit the model using `fitlmematrix` with the defined design matrices and grouping variables.

```
lme = fitlmematrix(X,y,Z,G,'FixedEffectPredictors',...
{'Intercept','InitWeight','PrgB','PrgC','PrgD','Week','Week_PrgB','Week_PrgC','Week_PrgD',
'RandomEffectPredictors',{'Intercept','Week'}},'RandomEffectGroups',{'Subject'})
```

```
lme =
```

```
Linear mixed-effects model fit by ML
```

```
Model information:
```

Number of observations	120
Fixed effects coefficients	9
Random effects coefficients	40
Covariance parameters	4

```
Formula:
```

```
Linear Mixed Formula with 10 predictors.
```

```
Model fit statistics:
```

AIC	BIC	LogLikelihood	Deviance
-22.981	13.257	24.49	-48.981

```
Fixed effects coefficients (95% CIs):
```

Name	Estimate	SE	tStat	DF	pValue
'Intercept'	0.66105	0.25892	2.5531	111	0.012034
'InitWeight'	0.0031879	0.0013814	2.3078	111	0.022863
'PrgB'	0.36079	0.13139	2.746	111	0.0070394

'PrgC'	-0.033263	0.13117	-0.25358	111	0.80029
'PrgD'	0.11317	0.13132	0.86175	111	0.39068
'Week'	0.1732	0.067454	2.5677	111	0.011567
'Week_PrgB'	0.038771	0.095394	0.40644	111	0.68521
'Week_PrgC'	0.030543	0.095394	0.32018	111	0.74944
'Week_PrgD'	0.033114	0.095394	0.34713	111	0.72915

Lower	Upper
0.14798	1.1741
0.00045067	0.0059252
0.10044	0.62113
-0.29319	0.22666
-0.14706	0.3734
0.039536	0.30686
-0.15026	0.2278
-0.15849	0.21957
-0.15592	0.22214

Random effects covariance parameters (95% CIs):

Group: Subject (20 Levels)

Name1	Name2	Type	Estimate	Lower
'Intercept'	'Intercept'	'std'	0.18407	0.12281
'Week'	'Intercept'	'corr'	0.66841	0.21077
'Week'	'Week'	'std'	0.15033	0.11004

Upper
0.27587
0.88573
0.20537

Group: Error

Name	Estimate	Lower	Upper
'Res Std'	0.10261	0.087882	0.11981

Examine the fixed effects coefficients table. The row labeled 'InitWeight' has a *P*-value of 0.0228, and the row labeled 'Week' has a *P*-value of 0.0115. These *P*-values indicate significant effects of the initial weights of the subjects and the time factor in the amount of weight lost. The weight loss of subjects who are in program B is significantly different relative to the weight loss of subjects who are in program A. The lower and upper limits of the covariance parameters for the random effects do not include zero, thus



they seem significant. You can also test the significance of the random-effects using the `compare` method.

### Random Intercept Model

Load the sample data.

```
load flu
```

The `flu` dataset array has a `Date` variable, and 10 variables for estimated influenza rates (in 9 different regions, estimated from Google® searches, plus a nationwide estimate from the Centers for Disease Control and Prevention, CDC).

To fit a linear-mixed effects model, where the influenza rates are the responses, combine the nine columns corresponding to the regions into an array that has a single response variable, `FluRate`, and a nominal variable, `Region`, the nationwide estimate `WtdILI`, that shows which region each estimate is from, and the grouping variable `Date`.

```
flu2 = stack(flu,2:10, 'NewDataVarName', 'FluRate', ...
            'IndVarName', 'Region');
flu2.Date = nominal(flu2.Date);
```

Define the design matrices for a random-intercept linear mixed-effects model, where the intercept varies by `Date`. The corresponding model is

$$y_{im} = \beta_0 + \beta_1 \text{WtdILI}_{im} + b_{0m} + \varepsilon_{im}, \quad i = 1, 2, \dots, 468, \quad m = 1, 2, \dots, 52,$$

where  $y_{im}$  is the observation  $i$  for level  $m$  of grouping variable `Date`,  $b_{0m}$  is the random effect for level  $m$  of the grouping variable `Date`, and  $\varepsilon_{im}$  is the observation error for observation  $i$ . The random effect has the prior distribution,

$$b_{0m} \sim N(0, \sigma_b^2),$$

and the error term has the distribution,

$$\varepsilon_{im} \sim N(0, \sigma^2).$$

```
y = flu2.FluRate;
X = [ones(468,1) flu2.WtdILI];
Z = [ones(468,1)];
G = flu2.Date;
```

Fit the linear mixed-effects model.

```
lme = fitlmematrix(X,y,Z,G,'FixedEffectPredictors',{'Intercept','NationalRate'},...
'RandomEffectPredictors',{'Intercept'}),'RandomEffectGroups',{'Date'})
```

```
lme =
```

Linear mixed-effects model fit by ML

Model information:

Number of observations	468
Fixed effects coefficients	2
Random effects coefficients	52
Covariance parameters	2

Formula:

```
y ~ Intercept + NationalRate + (Intercept | Date)
```

Model fit statistics:

AIC	BIC	LogLikelihood	Deviance
286.24	302.83	-139.12	278.24

Fixed effects coefficients (95% CIs):

Name	Estimate	SE	tStat	DF	pValue
'Intercept'	0.16385	0.057525	2.8484	466	0.0045885
'NationalRate'	0.7236	0.032219	22.459	466	3.0502e-76

Lower	Upper
0.050813	0.27689
0.66028	0.78691

Random effects covariance parameters (95% CIs):

Group: Date (52 Levels)

Name1	Name2	Type	Estimate	Lower
'Intercept'	'Intercept'	'std'	0.17146	0.13227

Upper
0.22226

Group: Error

Name	Estimate	Lower	Upper
'Res Std'	0.30201	0.28217	0.32324

The confidence limits of the standard deviation of the random-effects term  $\sigma_b$ , do not include zero (0.13227, 0.22226), which indicates that the random-effects term is significant. You can also test the significance of the random-effects using `compare` method.

The estimated value of an observation is the sum of the fixed-effects values and value of the random effect at the grouping variable level corresponding to that observation. For example, the estimated flu rate for observation 28

$$\begin{aligned}\hat{y}_{28} &= \hat{\beta}_0 + \hat{\beta}_1 \text{WtdILI}_{28} + \hat{b}_{10/30/2005} \\ &= 0.1639 + 0.7236 * (1.343) + 0.3318 \\ &= 1.46749,\end{aligned}$$

where  $\hat{b}$  is the best linear unbiased predictor (BLUP) of the random effects for the intercept. You can compute this value as follows.

```
beta = fixedEffects(lme);
[~,~,STATS] = randomEffects(lme); % compute the random effects statistics STATS
STATS.Level = nominal(STATS.Level);
y_hat = beta(1) + beta(2)*flu2.WtdILI(28) + STATS.Estimate(STATS.Level=='10/30/2005')

y_hat =

    1.4674
```

You can simply display the fitted value using the `fitted(lme)` method.

```
F = fitted(lme);
F(28)

ans =

    1.4674
```

## Randomized Block Design

Load the sample data.

```
load(fullfile(matlabroot, 'examples', 'stats', 'shift.mat'));
```

The data shows the deviations from the target quality characteristic measured from the products that five operators manufacture during three shifts: morning, evening, and night. This is a randomized block design, where the operators are the blocks. The experiment is designed to study the impact of the time of shift on the performance. The performance measure is the deviations of the quality characteristics from the target value. This is simulated data.

Define the design matrices for a linear mixed-effects model with a random intercept grouped by operator, and shift as the fixed effects. Use the 'effects' contrasts. 'effects' contrasts mean that the coefficients sum to 0. You need to create two contrast coded variables in the fixed-effects design matrix, X1 and X2, where

$$\text{Shift\_Evening} = \begin{cases} 0, & \text{if Morning} \\ 1, & \text{if Evening} \\ -1, & \text{if Night} \end{cases}$$

$$\text{Shift\_Morning} = \begin{cases} 1, & \text{if Morning} \\ 0, & \text{if Evening} \\ -1, & \text{if Night} \end{cases}$$

The model corresponds to

$$\text{Morning Shift: } \text{QCDev}_{im} = \beta_0 + \beta_2 \text{Shift\_Morning}_i + b_{0m} + \epsilon_{im},$$

$$\text{Evening Shift: } \text{QCDev}_{im} = \beta_0 + \beta_1 \text{Shift\_Evening}_i + b_{0m} + \epsilon_{im},$$

$$\text{Night Shift: } \text{QCDev}_{im} = \beta_0 - \beta_1 \text{Shift\_Evening}_i - \beta_2 \text{Shift\_Morning}_i + b_{0m} + \epsilon_{im},$$

where  $i$  represents the observations, and  $m$  represents the operators,  $i = 1, 2, \dots, 15$ , and  $m = 1, 2, \dots, 5$ . The random effects and the observation error have the following distributions:

$$b_{0m} \sim N(0, \sigma_b^2)$$

and

$$\epsilon_{im} \sim N(0, \sigma^2).$$

```
S = shift.Shift;
X1 = (S=='Morning') - (S=='Night');
X2 = (S=='Evening') - (S=='Night');
X = [ones(15,1), X1, X2];
y = shift.QCDev;
Z = ones(15,1);
G = shift.Operator;
```

Fit a linear mixed-effects model using the specified design matrices and restricted maximum likelihood method.

```
lme = fitlmematrix(X,y,Z,G,'FitMethod','REML','FixedEffectPredictors',...
{'Intercept','S_Morning','S_Evening'},'RandomEffectPredictors',{{'Intercept'}},...
'RandomEffectGroups',{'Operator'},'DummyVarCoding','effects')
```

```
lme =
```

```
Linear mixed-effects model fit by REML
```

```
Model information:
```

Number of observations	15
Fixed effects coefficients	3
Random effects coefficients	5
Covariance parameters	2

```
Formula:
```

```
y ~ Intercept + S_Morning + S_Evening + (Intercept | Operator)
```

```
Model fit statistics:
```

AIC	BIC	LogLikelihood	Deviance
58.913	61.337	-24.456	48.913

```
Fixed effects coefficients (95% CIs):
```

Name	Estimate	SE	tStat	DF	pValue
'Intercept'	3.6525	0.94109	3.8812	12	0.0021832
'S_Morning'	-0.91973	0.31206	-2.9473	12	0.012206
'S_Evening'	-0.53293	0.31206	-1.7078	12	0.11339

Lower	Upper
1.6021	5.703
-1.5997	-0.23981
-1.2129	0.14699

```
Random effects covariance parameters (95% CIs):
```

```
Group: Operator (5 Levels)
```

Name1	Name2	Type	Estimate	Lower
'Intercept'	'Intercept'	'std'	2.0457	0.98207

```
Upper
```

```
4.2612

Group: Error
  Name          Estimate    Lower    Upper
  'Res Std'     0.85462    0.52357  1.395
```

Compute the best linear unbiased predictor (BLUP) estimates of random effects.

```
B = randomEffects(lme)
```

```
B =
0.5775
1.1757
-2.1715
2.3655
-1.9472
```

The estimated deviation from the target quality characteristics for the third operator working the evening shift is

$$\begin{aligned}\hat{y}_{\text{Evening Operator}_3} &= \hat{\beta}_0 + \hat{\beta}_1 \text{Shift\_Evening} + \hat{b}_{03} \\ &= 3.6525 - 0.53293 - 2.1715 \\ &= 0.94807.\end{aligned}$$

You can also display this value as follows.

```
F = fitted(lme);
F(shift.Shift=='Evening' & shift.Operator=='3')
```

```
ans =
0.9481
```

### Correlated and Uncorrelated Random-Effects Terms

Load the sample data.

```
load carbig
```

Fit a linear mixed-effects model for miles per gallon (MPG), with fixed effects for acceleration and horsepower, and uncorrelated random effect for intercept and acceleration grouped by the model year. This model corresponds to

$$\text{MPG}_{im} = \beta_0 + \beta_1 \text{Acc}_i + \beta_2 \text{HP} + b_{0m} + b_{1m} \text{Acc}_{im} + \varepsilon_{im}, \quad m = 1, 2, 3,$$

with the random-effects terms having the following prior distributions:

$$b_{0m} \sim N(0, \sigma_0^2),$$

$$b_{1m} \sim N(0, \sigma_1^2),$$

where  $m$  represents the model year.

First, prepare the design matrices for fitting the linear mixed-effects model.

```
X = [ones(406,1) Acceleration Horsepower];
Z = {ones(406,1), Acceleration};
G = {Model_Year, Model_Year};
Model_Year = nominal(Model_Year);
```

Now, fit the model using `fitlmematrix` with the defined design matrices and grouping variables.

```
lme = fitlmematrix(X, MPG, Z, G, 'FixedEffectPredictors', ...
    {'Intercept', 'Acceleration', 'Horsepower'}, 'RandomEffectPredictors', ...
    {'Intercept'}, {'Acceleration'}), 'RandomEffectGroups', {'Model_Year', 'Model_Year'})
```

```
lme =
```

```
Linear mixed-effects model fit by ML
```

```
Model information:
```

Number of observations	392
Fixed effects coefficients	3
Random effects coefficients	26
Covariance parameters	3

```
Formula:
```

```
Linear Mixed Formula with 4 predictors.
```

```
Model fit statistics:
```

AIC	BIC	LogLikelihood	Deviance
2194.5	2218.3	-1091.3	2182.5

Fixed effects coefficients (95% CIs):

Name	Estimate	SE	tStat	DF	pValue
'Intercept'	49.839	2.0518	24.291	389	5.6168e-80
'Acceleration'	-0.58565	0.10846	-5.3995	389	1.1652e-07
'Horsepower'	-0.16534	0.0071227	-23.213	389	1.9755e-75

Lower	Upper
45.806	53.873
-0.7989	-0.3724
-0.17934	-0.15133

Random effects covariance parameters (95% CIs):

Group: Model\_Year (13 Levels)

Name1	Name2	Type	Estimate	Lower
'Intercept'	'Intercept'	'std'	8.6657e-07	NaN

Upper  
NaN

Group: Model\_Year (13 Levels)

Name1	Name2	Type	Estimate	Lower
'Acceleration'	'Acceleration'	'std'	0.18783	0.12523

Upper  
0.28172

Group: Error

Name	Estimate	Lower	Upper
'Res Std'	3.7258	3.4698	4.0007

Note that the random effects covariance parameters for intercept and acceleration are separate in the display. The standard deviation of the random effect for the intercept does not seem significant.

Refit the model with potentially correlated random effects for intercept and acceleration. In this case, the random-effects terms has this prior distribution

$$b_m = \begin{pmatrix} b_{0m} \\ b_{1m} \end{pmatrix} \sim N \left( 0, \begin{pmatrix} \sigma_0^2 & \sigma_{0,1} \\ \sigma_{0,1} & \sigma_1^2 \end{pmatrix} \right),$$



where  $m$  represents the model year.

First, prepare the random-effects design matrix and grouping variable.

```
Z = [ones(406,1) Acceleration];
G = Model_Year;

lme = fitlmematrix(X,MPG,Z,G,'FixedEffectPredictors',...
{'Intercept','Acceleration','Horsepower'},'RandomEffectPredictors',...
{{'Intercept','Acceleration'}},'RandomEffectGroups',{'Model_Year'})

lme =
```

Linear mixed-effects model fit by ML

Model information:

Number of observations	392
Fixed effects coefficients	3
Random effects coefficients	26
Covariance parameters	4

Formula:

Linear Mixed Formula with 4 predictors.

Model fit statistics:

AIC	BIC	LogLikelihood	Deviance
2193.5	2221.3	-1089.7	2179.5

Fixed effects coefficients (95% CIs):

Name	Estimate	SE	tStat	DF	pValue
'Intercept'	50.133	2.2652	22.132	389	7.7727e-71
'Acceleration'	-0.58327	0.13394	-4.3545	389	1.7075e-05
'Horsepower'	-0.16954	0.0072609	-23.35	389	5.188e-76

Lower	Upper
45.679	54.586
-0.84661	-0.31992
-0.18382	-0.15527

Random effects covariance parameters (95% CIs):

Group: Model\_Year (13 Levels)

Name1	Name2	Type	Estimate
-------	-------	------	----------

```
'Intercept'      'Intercept'      'std'      3.3475
'Acceleration'   'Intercept'      'corr'     -0.87971
'Acceleration'   'Acceleration'   'std'      0.33789
```

```
Lower      Upper
  1.2862    8.7119
-0.98501   -0.29675
  0.1825    0.62558
```

Group: Error

```
Name      Estimate      Lower      Upper
'Res Std'  3.6874      3.4298    3.9644
```

Note that the random effects covariance parameters for intercept and acceleration are together in the display, with an addition of the correlation between the intercept and acceleration. The confidence intervals for the standard deviations and the correlation between the random effects for intercept and acceleration do not include 0s, hence they seem significant. You can compare these two models using the `LinearMixedModel.compare` method.

## Specify the Covariance Pattern

Load the sample data.

```
load(fullfile(matlabroot, 'examples', 'stats', 'weight.mat'));
```

`weight` contains data from a longitudinal study, where 20 subjects are randomly assigned 4 exercise programs, and their weight loss is recorded over six 2-week time periods. This is simulated data.

Define `Subject` and `Program` as categorical variables.

```
Subject = nominal(Subject);
Program = nominal(Program);
```

Create the design matrices for a linear mixed-effects model, with the initial weight, type of program, and week as the fixed effects.

```
D = dummyvar(Program);
X = [ones(120,1), InitialWeight, D(:,2:4), Week];
```

```
Z = [ones(120,1) Week];
G = Subject;
```

This model corresponds to

$$y_{im} = \beta_0 + \beta_1 IW_i + \beta_2 \text{Week}_i + \beta_3 I[\text{PB}]_i + \beta_4 I[\text{PC}]_i + \beta_5 I[\text{PD}]_i \\ + b_{0m} + b_{1m} \text{Week}_{2im} + b_{2m} \text{Week}_{4im} + b_{3m} \text{Week}_{6im} + b_{4m} \text{Week}_{8im} \\ + b_{5m} \text{Week}_{10im} + b_{6m} \text{Week}_{12im} + \varepsilon_{im},$$

where  $i = 1, 2, \dots, 120$ , and  $m = 1, 2, \dots, 20$ .

$\beta_j$  are the fixed-effects coefficients,  $j = 0, 1, \dots, 8$ , and  $b_{0m}$  and  $b_{1m}$  are random effects.  $IW$  stands for initial weight and  $I[\cdot]$  is a dummy variable representing a type of program. For example,  $I[\text{PB}]_i$  is the dummy variable representing program type B. The random effects and observation error have the following prior distributions:

$$b_{0m} \sim N(0, \sigma_0^2)$$

$$b_{1m} \sim N(0, \sigma_1^2)$$

$$\varepsilon_{im} \sim N(0, \sigma^2).$$

Fit the model using `fitlmematrix` with the defined design matrices and grouping variables. Assume the repeated observations collected on a subject have common variance along diagonals.

```
lme = fitlmematrix(X,y,Z,G,'FixedEffectPredictors',...
{'Intercept','InitWeight','PrgB','PrgC','PrgD','Week'},...
'RandomEffectPredictors',{{'Intercept','Week'}},...
'RandomEffectGroups',{'Subject'},'CovariancePattern','Isotropic')
```

```
lme =
```

```
Linear mixed-effects model fit by ML
```

```
Model information:
```

Number of observations	120
Fixed effects coefficients	6
Random effects coefficients	40
Covariance parameters	2

```
Formula:
```

```
Linear Mixed Formula with 7 predictors.
```

Model fit statistics:

AIC	BIC	LogLikelihood	Deviance
-24.783	-2.483	20.391	-40.783

Fixed effects coefficients (95% CIs):

Name	Estimate	SE	tStat	DF	pValue
'Intercept'	0.4208	0.28169	1.4938	114	0.13799
'InitWeight'	0.0045552	0.0015338	2.9699	114	0.0036324
'PrgB'	0.36993	0.12119	3.0525	114	0.0028242
'PrgC'	-0.034009	0.1209	-0.28129	114	0.77899
'PrgD'	0.121	0.12111	0.99911	114	0.31986
'Week'	0.19881	0.037134	5.3538	114	4.5191e-07

Lower	Upper
-0.13723	0.97883
0.0015168	0.0075935
0.12986	0.61
-0.27351	0.2055
-0.11891	0.36091
0.12525	0.27237

Random effects covariance parameters (95% CIs):

Group: Subject (20 Levels)

Name1	Name2	Type	Estimate	Lower
'Intercept'	'Intercept'	'std'	0.16561	0.12896

Upper  
0.21269

Group: Error

Name	Estimate	Lower	Upper
'Res Std'	0.10272	0.088014	0.11987

## Input Arguments

**x** — Fixed-effects design matrix

*n*-by-*p* matrix

Fixed-effects design matrix, specified as an  $n$ -by- $p$  matrix, where  $n$  is the number of observations, and  $p$  is the number of fixed-effects predictor variables. Each row of  $X$  corresponds to one observation, and each column of  $X$  corresponds to one variable.

Data Types: `single` | `double`

### **y** — Response values

$n$ -by-1 vector

Response values, specified as an  $n$ -by-1 vector, where  $n$  is the number of observations.

Data Types: `single` | `double`

### **z** — Random-effects design

$n$ -by- $q$  matrix | cell array of  $R$   $n$ -by- $q(r)$  matrices,  $r = 1, 2, \dots, R$

Random-effects design, specified as either of the following.

- If there is one random-effects term in the model, then  $Z$  must be an  $n$ -by- $q$  matrix, where  $n$  is the number of observations and  $q$  is the number of variables in the random-effects term.
- If there are  $R$  random-effects terms, then  $Z$  must be a cell array of length  $R$ . Each cell of  $Z$  contains an  $n$ -by- $q(r)$  design matrix  $Z\{r\}$ ,  $r = 1, 2, \dots, R$ , corresponding to each random-effects term. Here,  $q(r)$  is the number of random effects term in the  $r$ th random effects design matrix,  $Z\{r\}$ .

Data Types: `single` | `double` | `cell`

### **G** — Grouping variable or variables

$n$ -by-1 vector | cell array of  $R$   $n$ -by-1 vectors

Grouping variable or variables on page 2-59, specified as either of the following.

- If there is one random-effects term, then  $G$  must be an  $n$ -by-1 vector corresponding to a single grouping variable with  $M$  levels or groups.

$G$  can be a categorical vector, numeric vector, character array, or cell array of character vectors.

- If there are multiple random-effects terms, then  $G$  must be a cell array of length  $R$ . Each cell of  $G$  contains a grouping variable  $G\{r\}$ ,  $r = 1, 2, \dots, R$ , with  $M(r)$  levels.

$G\{r\}$  can be a categorical vector, numeric vector, character array, or cell array of character vectors.

Data Types: `single` | `double` | `char` | `cell`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1`, `Value1`, ..., `NameN`, `ValueN`.

Example:

`'CovariancePattern', 'Diagonal', 'DummyVarCoding', 'full', 'Optimizer', 'fminunc'` specifies a random-effects covariance pattern with zero off-diagonal elements, creates a dummy variable for each level of a categorical variable, and uses the `fminunc` optimization algorithm.

### **FixedEffectPredictors** — Names of columns in fixed-effects design matrix

`{'x1', 'x2', ..., 'xP'}` (default) | cell array of length  $p$

Names of columns in the fixed-effects design matrix  $X$ , specified as the comma-separated pair consisting of `'FixedEffectPredictors'` and a cell array of length  $p$ .

For example, if you have a constant term and two predictors, say `TimeSpent` and `Gender`, where `Female` is the reference level for `Gender`, as the fixed effects, then you can specify the names of your fixed effects in the following way. `Gender_Male` represents the dummy variable you must create for category `Male`. You can choose different names for these variables.

Example: `'FixedEffectPredictors',`  
`{'Intercept', 'TimeSpent', 'Gender_Male'},`

Data Types: `cell`

### **RandomEffectPredictors** — Names of columns in random-effects design matrix or cell array

cell array of length  $q$  | cell array of length  $R$  with elements of length  $q(r)$ ,  $r = 1, 2, \dots, R$

Names of columns in the random-effects design matrix or cell array  $Z$ , specified as the comma-separated pair consisting of `'RandomEffectPredictors'` and either of the following:

- A cell array of length  $q$  when  $Z$  is an  $n$ -by- $q$  design matrix. In this case, the default is `{'z1', 'z2', ..., 'zQ'}`.
- A cell array of length  $R$ , when  $Z$  is a cell array of length  $R$  with each element  $Z\{r\}$  of length  $q(r)$ ,  $r = 1, 2, \dots, R$ . In this case, the default is `{'z11', 'z12', ..., 'z1Q(1)'}, ..., {'zr1', 'zr2', ..., 'zrQ(r)'}`.

For example, suppose you have correlated random effects for intercept and a variable named `Acceleration`. Then, you can specify the random-effects predictor names as follows.

Example: `'RandomEffectPredictors', {'Intercept', 'Acceleration'}`

If you have two random effects terms, one for the intercept and the variable `Acceleration` grouped by variable `g1`, and the second for the intercept, grouped by the variable `g2`, then you specify the random-effects predictor names as follows.

Example: `'RandomEffectPredictors', {'Intercept', 'Acceleration'}, {'Intercept'}`

Data Types: `cell`

### **ResponseVarName** — Name of response variable

`'y'` (default) | character vector

Name of response variable, specified as the comma-separated pair consisting of `'ResponseVarName'` and a character vector.

For example, if your response variable name is `score`, then you can specify it as follows.

Example: `'ResponseVarName', 'score'`

Data Types: `char`

### **RandomEffectGroups** — Names of random effects grouping variables

`'g'` or `{'g1', 'g2', ..., 'gR'}` (default) | character vector | cell array of character vectors

Names of random effects grouping variables, specified as the comma-separated pair `'RandomEffectGroups'` and either of the following:

- Character vector — If there is only one random-effects term, that is, if `G` is a vector, then the value of `'RandomEffectGroups'` is a character vector containing the name for the grouping variable `G`. The default is `'g'`.

- Cell array of character vectors — If there are multiple random-effects terms, that is, if  $G$  is a cell array of length  $R$ , then the value of 'RandomEffectGroups' is a cell array of length  $R$ , where each cell contains the name for the grouping variable  $G\{r\}$ . The default is {'g1', 'g2', ..., 'gR'}.

For example, if you have two random-effects terms,  $z_1$  and  $z_2$ , grouped by the grouping variables `sex` and `subject`, then you can specify the names of your grouping variables as follows.

Example: 'RandomEffectGroups', {'sex', 'subject'}

Data Types: char | cell

### **CovariancePattern** — Pattern of covariance matrix

'FullCholesky' (default) | character vector | square symmetric logical matrix | cell array of character vectors or logical matrices

Pattern of the covariance matrix of the random effects, specified as the comma-separated pair consisting of 'CovariancePattern' and a character vector, a square symmetric logical matrix, or a cell array of character vectors or logical matrices.

If there are  $R$  random-effects terms, then the value of 'CovariancePattern' must be a cell array of length  $R$ , where each element  $r$  of this cell array specifies the pattern of the covariance matrix of the random-effects vector associated with the  $r$ th random-effects term. The options for each element follow.

'FullCholesky'

Default. Full covariance matrix using the Cholesky parameterization. `fitlme` estimates all elements of the covariance matrix.

'Full'

Full covariance matrix, using the log-Cholesky parameterization. `fitlme` estimates all elements of the covariance matrix.



'Diagonal'

Diagonal covariance matrix. That is, off-diagonal elements of the covariance matrix are constrained to be 0.

$$\begin{pmatrix} \sigma_{b1}^2 & 0 & 0 \\ 0 & \sigma_{b2}^2 & 0 \\ 0 & 0 & \sigma_{b3}^2 \end{pmatrix}$$

'Isotropic'

Diagonal covariance matrix with equal variances. That is, off-diagonal elements of the covariance matrix are constrained to be 0, and the diagonal elements are constrained to be equal. For example, if there are three random-effects terms with an isotropic covariance structure, this covariance matrix looks like

$$\begin{pmatrix} \sigma_b^2 & 0 & 0 \\ 0 & \sigma_b^2 & 0 \\ 0 & 0 & \sigma_b^2 \end{pmatrix}$$

where  $\sigma_b^2$  is the common variance of the random-effects terms.

`'CompSymm'`

Compound symmetry structure. That is, common variance along diagonals and equal correlation between all random effects. For example, if there are three random-effects terms with a covariance matrix having a compound symmetry structure, this covariance matrix looks like

$$\begin{pmatrix} \sigma_{b1}^2 & \sigma_{b1,b2} & \sigma_{b1,b2} \\ \sigma_{b1,b2} & \sigma_{b1}^2 & \sigma_{b1,b2} \\ \sigma_{b1,b2} & \sigma_{b1,b2} & \sigma_{b1}^2 \end{pmatrix}$$

where  $\sigma_{b1}^2$  is the common variance of the random-effects terms and  $\sigma_{b1,b2}$  is the common covariance between any two random-effects term .

`PAT`

Square symmetric logical matrix. If `'CovariancePattern'` is defined by the matrix `PAT`, and if `PAT(a,b) = false`, then the `(a,b)` element of the corresponding covariance matrix is constrained to be 0.

Example: `'CovariancePattern', 'Diagonal'`

Example: `'CovariancePattern', {'Full', 'Diagonal'}`

### **FitMethod** — Method for estimating parameters

`'ML'` (default) | `'REML'`

Method for estimating parameters of the linear mixed-effects model, specified as the comma-separated pair consisting of `'FitMethod'` and either of the following.

`'ML'`

Default. Maximum likelihood estimation

`'REML'`

Restricted maximum likelihood estimation

Example: `'FitMethod', 'REML'`

### **weights** — Observation weights

vector of scalar values

Observation weights, specified as the comma-separated pair consisting of 'Weights' and a vector of length  $n$ , where  $n$  is the number of observations.

Data Types: `single` | `double`

**Exclude — Indices for rows to exclude**

use all rows without NaNs (default) | vector of integer or logical values

Indices for rows to exclude from the linear mixed-effects model in the data, specified as the comma-separated pair consisting of 'Exclude' and a vector of integer or logical values.

For example, you can exclude the 13th and 67th rows from the fit as follows.

Example: `'Exclude', [13, 67]`

Data Types: `single` | `double` | `logical`

**DummyVarCoding — Coding to use for dummy variables**

'reference' (default) | 'effects' | 'full'

Coding to use for dummy variables created from the categorical variables, specified as the comma-separated pair consisting of 'DummyVarCoding' and one of the following.

Value	Description
'reference'	Default. Coefficient for first category set to 0.
'effects'	Coefficients sum to 0.
'full'	One dummy variable for each category.

Example: `'DummyVarCoding', 'effects'`

**Optimizer — Optimization algorithm**

'quasinevton' (default) | 'fminunc'

Optimization algorithm, specified as the comma-separated pair consisting of 'Optimizer' and either of the following.

<code>'quasinevton'</code>	Default. Uses a trust region based quasi-Newton optimizer. Change the options of the algorithm using <code>statset('LinearMixedModel')</code> . If you don't specify the options, then <code>LinearMixedModel</code> uses the default options of <code>statset('LinearMixedModel')</code> .
<code>'fminunc'</code>	You must have Optimization Toolbox to specify this option. Change the options of the algorithm using <code>optimoptions('fminunc')</code> . If you don't specify the options, then <code>LinearMixedModel</code> uses the default options of <code>optimoptions('fminunc')</code> with <code>'Algorithm'</code> set to <code>'quasi-newton'</code> .

Example: `'Optimizer','fminunc'`

### **OptimizerOptions — Options for optimization algorithm**

structure returned by `statset` | object returned by `optimoptions`

Options for the optimization algorithm, specified as the comma-separated pair consisting of `'OptimizerOptions'` and a structure returned by `statset('LinearMixedModel')` or an object returned by `optimoptions('fminunc')`.

- If `'Optimizer'` is `'fminunc'`, then use `optimoptions('fminunc')` to change the options of the optimization algorithm. See `optimoptions` for the options `'fminunc'` uses. If `'Optimizer'` is `'fminunc'` and you do not supply `'OptimizerOptions'`, then the default for `LinearMixedModel` is the default options created by `optimoptions('fminunc')` with `'Algorithm'` set to `'quasi-newton'`.
- If `'Optimizer'` is `'quasinevton'`, then use `statset('LinearMixedModel')` to change the optimization parameters. If you don't change the optimization parameters, then `LinearMixedModel` uses the default options created by `statset('LinearMixedModel')`:

The 'quasinevton' optimizer uses the following fields in the structure created by `statset('LinearMixedModel')`.

**TolFun** — Relative tolerance on gradient of objective function

1e-6 (default) | positive scalar value

Relative tolerance on the gradient of the objective function, specified as a positive scalar value.

**TolX** — Absolute tolerance on step size

1e-12 (default) | positive scalar value

Absolute tolerance on the step size, specified as a positive scalar value.

**MaxIter** — Maximum number of iterations allowed

10000 (default) | positive scalar value

Maximum number of iterations allowed, specified as a positive scalar value.

**Display** — Level of display

'off' (default) | 'iter' | 'final'

Level of display, specified as one of 'off', 'iter', or 'final'.

**StartMethod** — Method to start iterative optimization

'default' (default) | 'random'

Method to start iterative optimization, specified as the comma-separated pair consisting of 'StartMethod' and either of the following.

Value	Description
'default'	An internally defined default value
'random'	A random initial value

Example: 'StartMethod', 'random'

**Verbose** — Indicator to display optimization process on screen

false (default) | true

Indicator to display the optimization process on screen, specified as the comma-separated pair consisting of 'Verbose' and either false or true. Default is false.

The setting for 'Verbose' overrides the field 'Display' in 'OptimizerOptions'.

Example: 'Verbose', true

### **CheckHessian** — Indicator to check positive definiteness of Hessian

false (default) | true

Indicator to check the positive definiteness of the Hessian of the objective function with respect to unconstrained parameters at convergence, specified as the comma-separated pair consisting of 'CheckHessian' and either false or true. Default is false.

Specify 'CheckHessian' as true to verify optimality of the solution or to determine if the model is overparameterized in the number of covariance parameters.

Example: 'CheckHessian', true

## Output Arguments

### **lme** — Linear mixed-effects model

LinearMixedModel object

Linear mixed-effects model, returned as a LinearMixedModel object.

## Definitions

### Cholesky Parameterization

One of the assumptions of linear mixed-effects models is that the random effects have the following prior distribution.

$$b \sim N(0, \sigma^2 D(\theta)),$$

where  $D$  is a  $q$ -by- $q$  symmetric and positive semidefinite matrix, parameterized by a variance component vector  $\theta$ ,  $q$  is the number of variables in the random-effects term, and  $\sigma^2$  is the observation error variance. Since the covariance matrix of the random effects,  $D$ , is symmetric, it has  $q(q+1)/2$  free parameters. Suppose  $L$  is the lower triangular Cholesky factor of  $D(\theta)$  such that

$$D(\theta) = L(\theta)L(\theta)^T,$$

then the  $q^*(q+1)/2$ -by-1 unconstrained parameter vector  $\theta$  is formed from elements in the lower triangular part of  $L$ .

For example, if

$$L = \begin{bmatrix} L_{11} & 0 & 0 \\ L_{21} & L_{22} & 0 \\ L_{31} & L_{32} & L_{33} \end{bmatrix},$$

then

$$\theta = \begin{bmatrix} L_{11} \\ L_{21} \\ L_{31} \\ L_{22} \\ L_{32} \\ L_{33} \end{bmatrix}.$$

## Log-Cholesky Parameterization

When the diagonal elements of  $L$  in Cholesky parameterization are constrained to be positive, then the solution for  $L$  is unique. Log-Cholesky parameterization is the same as Cholesky parameterization except that the logarithm of the diagonal elements of  $L$  are used to guarantee unique parameterization.

For example, for the 3-by-3 example in Cholesky parameterization, enforcing  $L_{ii} \geq 0$ ,

$$\theta = \begin{bmatrix} \log(L_{11}) \\ L_{21} \\ L_{31} \\ \log(L_{22}) \\ L_{32} \\ \log(L_{33}) \end{bmatrix}.$$

## Alternative Functionality

You can also fit a linear mixed-effects model using `fitlme(tbl, formula)`, where `tbl` is a table or dataset array containing the response  $y$ , the predictor variables  $X$ , and the

grouping variables, and formula is of the form `'y ~ fixed + (random1|g1) + ... + (randomR|gR)'`.

## See Also

`LinearMixedModel` | `compare` | `fitlme`

**Introduced in R2013b**



# fitrgp

Fit a Gaussian process regression (GPR) model

## Syntax

```
gprMdl = fitrgp(tbl, ResponseVarName)
gprMdl = fitrgp(tbl, formula)
gprMdl = fitrgp(tbl, y)

gprMdl = fitrgp(X, y)

gprMdl = fitrgp( ____, Name, Value)
```

## Description

`gprMdl = fitrgp(tbl, ResponseVarName)` returns a Gaussian process regression (GPR) model trained using the sample data in `tbl`, where `ResponseVarName` is the name of the response variable in `tbl`.

`gprMdl = fitrgp(tbl, formula)` returns a Gaussian process regression (GPR) model, trained using the sample data in `tbl`, for the predictor variables and response variables identified by `formula`.

`gprMdl = fitrgp(tbl, y)` returns a GPR model for the predictors in table `tbl` and continuous response vector `y`.

`gprMdl = fitrgp(X, y)` returns a GPR model for predictors `X` and continuous response vector `y`.

`gprMdl = fitrgp( ____, Name, Value)` returns a GPR model for any of the input arguments in the previous syntaxes, with additional options specified by one or more `Name, Value` pair arguments.

For example, you can specify the fitting method, the prediction method, the covariance function, or the active set selection method. You can also train a cross-validated model.

`gprMdl` is a `RegressionGP` object. For methods and properties of this class, see [RegressionGP class page](#).

If you train a cross-validated model, then `gprMdl` is a `RegressionPartitionedModel` object. For further analysis on the cross-validated object, use the methods of `RegressionPartitionedModel` class. For the methods of this class, see the [RegressionPartitionedModel class page](#).

## Examples

### Train GPR Model Using Data in Table

This example uses the abalone data [1], [2], from the UCI Machine Learning Repository [3]. Download the data and save it in your current folder with the name 'abalone.data'.

Store the data into a table. Display the first seven rows.

```
tbl = readtable('abalone.data', 'Filetype', 'text', ...
    'ReadVariableNames', false);
tbl.Properties.VariableNames = {'Sex', 'Length', 'Diameter', 'Height', ...
    'WWeight', 'SWeight', 'VWeight', 'ShWeight', 'NoShellRings'};
tbl(1:7, :)
```

ans =

Sex	Length	Diameter	Height	WWeight	SWeight	VWeight	ShWeight
'M'	0.455	0.365	0.095	0.514	0.2245	0.101	0.15
'M'	0.35	0.265	0.09	0.2255	0.0995	0.0485	0.07
'F'	0.53	0.42	0.135	0.677	0.2565	0.1415	0.21
'M'	0.44	0.365	0.125	0.516	0.2155	0.114	0.155
'I'	0.33	0.255	0.08	0.205	0.0895	0.0395	0.055
'I'	0.425	0.3	0.095	0.3515	0.141	0.0775	0.12
'F'	0.53	0.415	0.15	0.7775	0.237	0.1415	0.33

The dataset has 4177 observations. The goal is to predict the age of abalone from eight physical measurements. The last variable, number of shell rings shows the age of the abalone. The first predictor is a categorical variable. The last variable in the table is the response variable.

Fit a GPR model using the subset of regressors method for parameter estimation and fully independent conditional method for prediction. Standardize the predictors.

```
gprMdl = fitrgp(tbl, 'NoShellRings', 'KernelFunction', 'ardsquaredexponential', ...
    'FitMethod', 'sr', 'PredictMethod', 'fic', 'Standardize', 1)
```

```
grMdl =
```

```
RegressionGP
  PredictorNames: {1x8 cell}
  ResponseName: 'Var9'
  ResponseTransform: 'none'
  NumObservations: 4177
  KernelFunction: 'ARDSquaredExponential'
  KernelInformation: [1x1 struct]
    BasisFunction: 'Constant'
    Beta: 10.9148
    Sigma: 2.0243
  PredictorLocation: [10x1 double]
  PredictorScale: [10x1 double]
    Alpha: [1000x1 double]
  ActiveSetVectors: [1000x10 double]
  PredictMethod: 'FIC'
  ActiveSetSize: 1000
  FitMethod: 'SR'
  ActiveSetMethod: 'Random'
  IsActiveSetVector: [4177x1 logical]
  LogLikelihood: -9.0013e+03
  ActiveSetHistory: [1x1 struct]
  BCDInformation: []
```

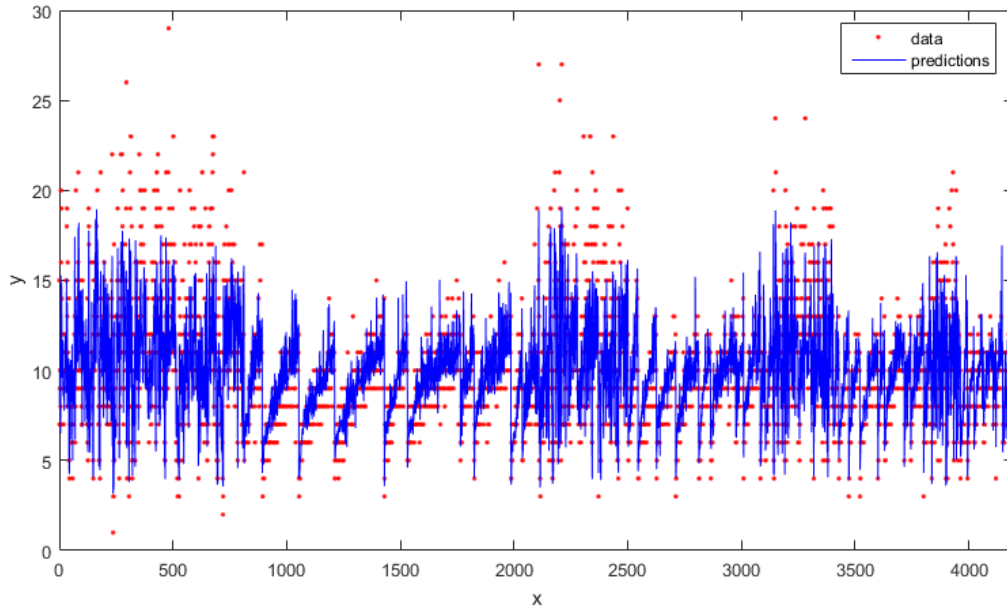
Predict the responses using the trained model.

```
ypred = resubPredict(gprMdl);
```

Plot the true response and the predicted responses.

```
figure();
plot(tbl.NoShellRings, 'r. ');
hold on
plot(ypred, 'b');
xlabel('x');
ylabel('y');
legend({'data', 'predictions'}, 'Location', 'Best');
```

```
axis([0 4300 0 30]);
hold off;
```



Compute the regression loss on the training data (resubstitution loss) for the trained model.

```
L = resubLoss(gprMdl)
```

```
L =
```

```
4.0064
```

### Train GPR Model and Plot Predictions

Generate sample data.

```
rng(0,'twister'); % For reproducibility
n = 1000;
```

```
x = linspace(-10,10,n)';  
y = 1 + x*5e-2 + sin(x)./x + 0.2*randn(n,1);
```

Fit a GPR model using a linear basis function and the exact fitting method to estimate the parameters. Also use the exact prediction method.

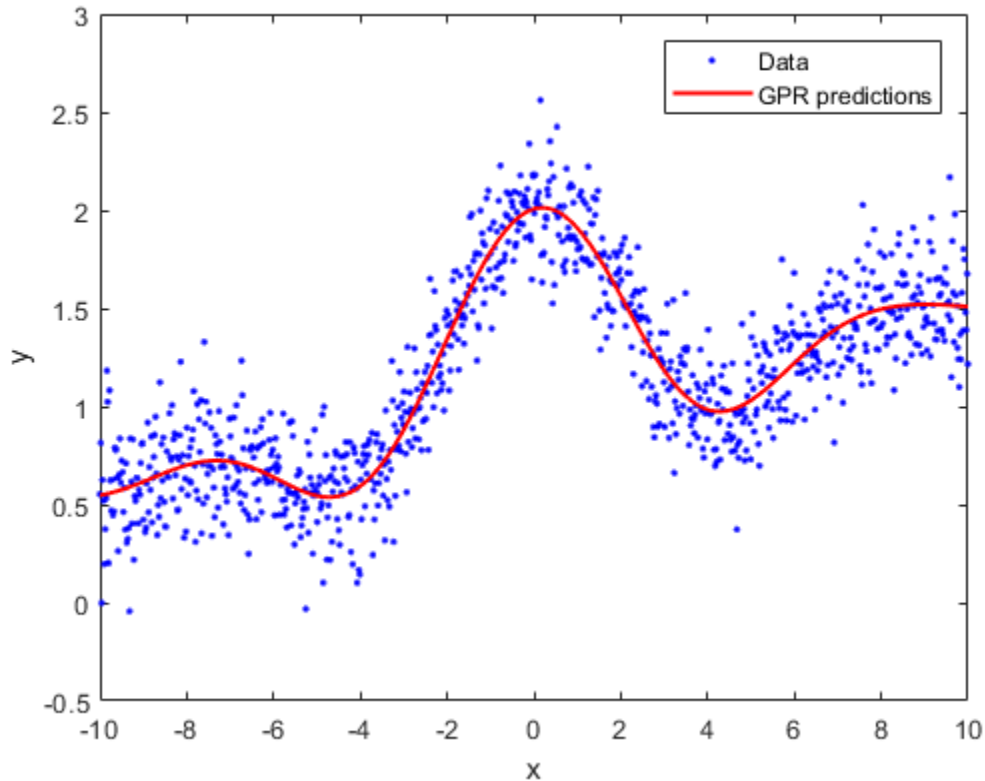
```
gprMdl = fitrgp(x,y,'Basis','linear',...  
               'FitMethod','exact','PredictMethod','exact');
```

Predict the response corresponding to the rows of  $x$  (resubstitution predictions) using the trained model.

```
ypred = resubPredict(gprMdl);
```

Plot the true response with the predicted values.

```
plot(x,y,'b. ');  
hold on;  
plot(x,ypred,'r','LineWidth',1.5);  
xlabel('x');  
ylabel('y');  
legend('Data','GPR predictions');  
hold off
```



### Impact of Specifying Initial Kernel Parameter Values

Load the sample data.

```
load(fullfile(matlabroot, 'examples', 'stats', 'gprdata2.mat'))
```

The data has one predictor variable and continuous response. This is simulated data.

Fit a GPR model using the squared exponential kernel function with default kernel parameters.

```
gprMdl1 = fitrgp(x, y, 'KernelFunction', 'squareexponential');
```

Now, fit a second model, where you specify the initial values for the kernel parameters.

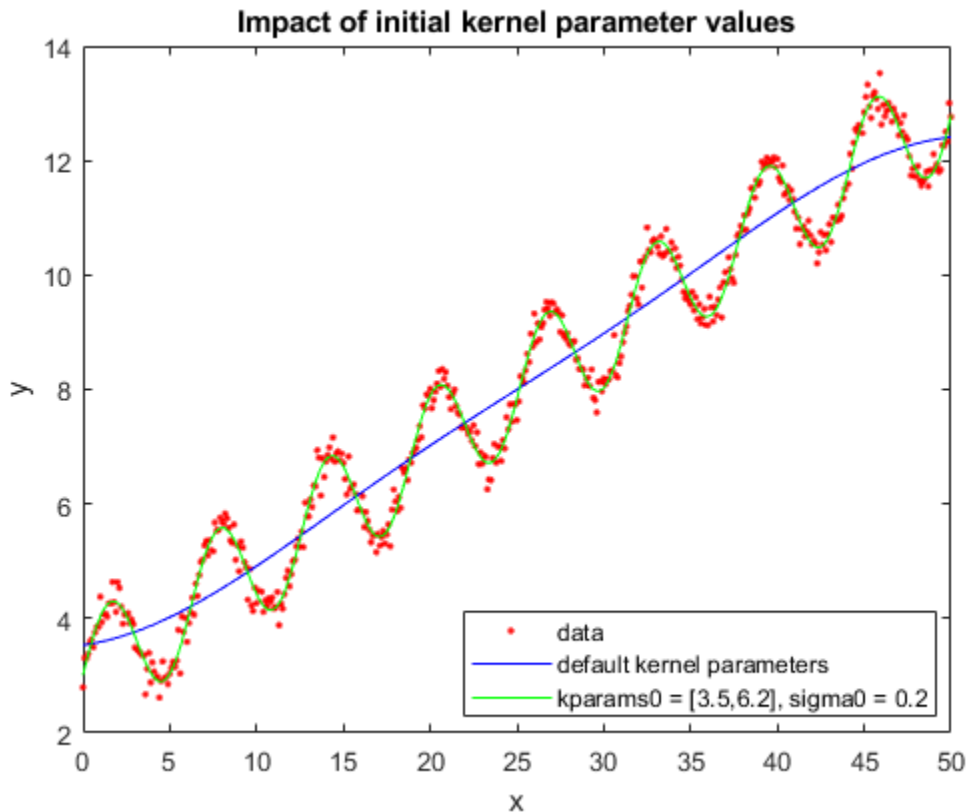
```
sigma0 = 0.2;
kparams0 = [3.5, 6.2];
gprMdl2 = fitrgp(x,y,'KernelFunction','squaredexponential',...
    'KernelParameters',kparams0,'Sigma',sigma0);
```

Compute the resubstitution predictions from both models.

```
ypred1 = resubPredict(gprMdl1);
ypred2 = resubPredict(gprMdl2);
```

Plot the response predictions from both models and the responses in training data.

```
figure();
plot(x,y,'r. ');
hold on
plot(x,ypred1,'b');
plot(x,ypred2,'g');
xlabel('x');
ylabel('y');
legend({'data','default kernel parameters',...
    'kparams0 = [3.5,6.2], sigma0 = 0.2'},...
    'Location','Best');
title('Impact of initial kernel parameter values');
hold off
```



The marginal log likelihood that `fitrgp` maximizes to estimate GPR parameters has multiple local solutions; the solution that it converges to depends on the initial point. Each local solution corresponds to a particular interpretation of the data. In this example, the solution with the default initial kernel parameters corresponds to a low frequency signal with high noise whereas the second solution with custom initial kernel parameters corresponds to a high frequency signal with low noise.

### Use Separate Length Scales for Predictors

Load the sample data.

```
load(fullfile(matlabroot, 'examples', 'stats', 'gprdata.mat'))
```



There are six continuous predictor variables. There are 500 observations in the training data set and 100 observations in the test data set. This is simulated data.

Fit a GPR model using the squared exponential kernel function with a separate length scale for each predictor. This covariance function is defined as:

$$k(x_i, x_j | \theta) = \sigma_f^2 \exp \left[ -\frac{1}{2} \sum_{m=1}^d \frac{(x_{im} - x_{jm})^2}{\sigma_m^2} \right].$$

where  $\sigma_m$  represents the length scale for predictor  $m$ ,  $m = 1, 2, \dots, d$  and  $\sigma_f$  is the signal standard deviation. The unconstrained parametrization  $\theta$  is

$$\begin{aligned} \theta_m &= \log \sigma_m, \quad \text{for } m = 1, 2, \dots, d \\ \theta_{d+1} &= \log \sigma_f. \end{aligned}$$

Initialize length scales of the kernel function at 10 and signal and noise standard deviations at the standard deviation of the response.

```
sigma0 = std(ytrain);
sigmaF0 = sigma0;
d = size(Xtrain,2);
sigmaM0 = 10*ones(d,1);
```

Fit the GPR model using the initial kernel parameter values. Standardize the predictors in the training data. Use the exact fitting and prediction methods.

```
gprMdl = fitrgp(Xtrain,ytrain,'Basis','constant','FitMethod','exact',...
'PredictMethod','exact','KernelFunction','ardsquaredexponential',...
'KernelParameters',[sigmaM0;sigmaF0],'Sigma',sigma0,'Standardize',1);
```

Compute the regression loss on the test data.

```
L = loss(gprMdl,Xtest,ytest)
```

```
L =
```

```
0.6919
```

Access the kernel information.

```
gprMdl.KernelInformation
```

```
ans =
```

```
struct with fields:

                Name: 'ARDSquaredExponential'
    KernelParameters: [7x1 double]
    KernelParameterNames: {7x1 cell}
```

Display the kernel parameter names.

```
gprMdl.KernelInformation.KernelParameterNames
```

```
ans =

7x1 cell array

    {'LengthScale1'}
    {'LengthScale2'}
    {'LengthScale3'}
    {'LengthScale4'}
    {'LengthScale5'}
    {'LengthScale6'}
    {'SigmaF'       }
```

Display the kernel parameters.

```
sigmaM = gprMdl.KernelInformation.KernelParameters(1:end-1,1)
sigmaF = gprMdl.KernelInformation.KernelParameters(end)
sigma  = gprMdl.Sigma
```

```
sigmaM =

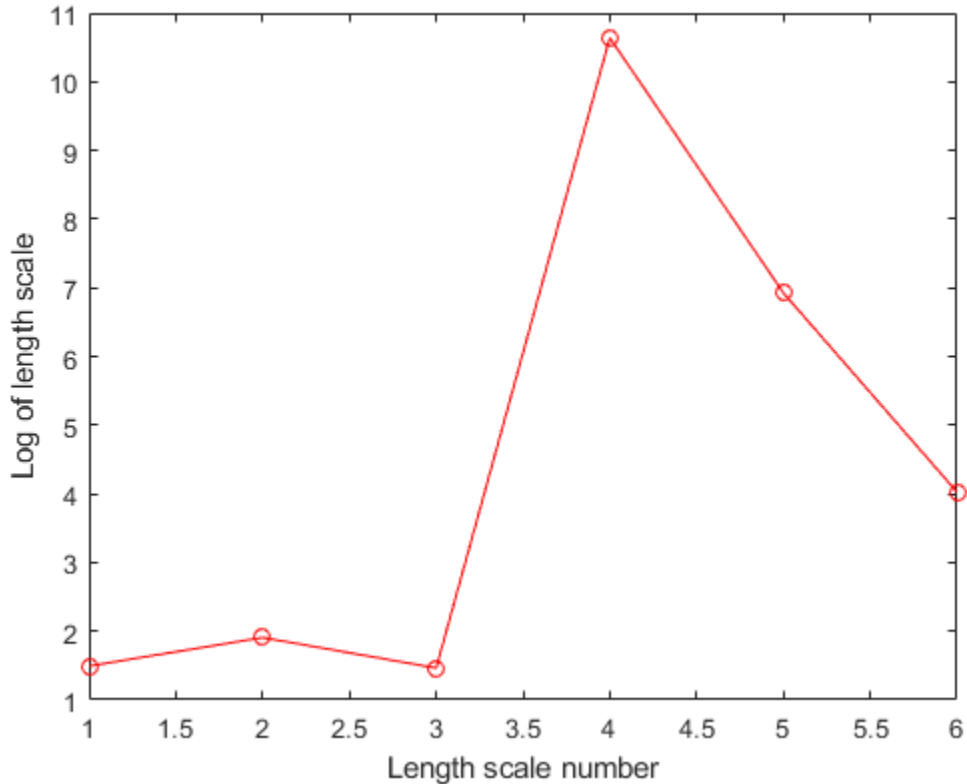
1.0e+04 *

    0.0004
    0.0007
    0.0004
    4.1729
    0.1018
    0.0056
```

```
sigmaF =  
    28.1718  
  
sigma =  
    0.8162
```

Plot the log of learned length scales.

```
figure()  
plot(1:d', log(sigmaM), 'ro-');  
xlabel('Length scale number');  
ylabel('Log of length scale');
```



The log of length scale for the 4th and 5th predictor variables are high relative to the others. These predictor variables do not seem to be as influential on the response as the other predictor variables.

Fit the GPR model without using the 4th and 5th variables as the predictor variables.

```
X = [Xtrain(:,1:3) Xtrain(:,6)];
sigma0 = std(ytrain);
sigmaF0 = sigma0;
d = size(X,2);
sigmaM0 = 10*ones(d,1);

gprMdl = fitrgp(X,ytrain,'Basis','constant','FitMethod','exact',...
```

```
'PredictMethod', 'exact', 'KernelFunction', 'ardsquaredexponential', ...  
'KernelParameters', [sigmaM0;sigmaF0], 'Sigma', sigma0, 'Standardize', 1);
```

Compute the regression error on the test data.

```
xtest = [Xtest(:,1:3) Xtest(:,6)];  
L = loss(gprMdl,xtest,ytest)
```

```
L =
```

```
0.6928
```

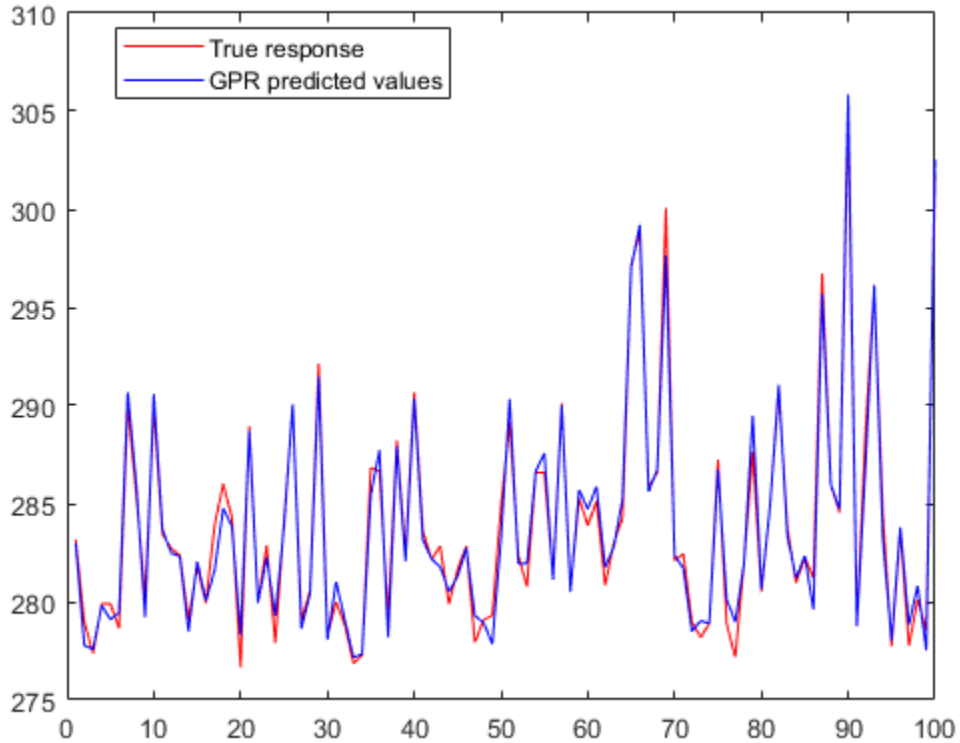
The loss is similar to the one when all variables are used as predictor variables.

Compute the predicted response for the test data.

```
ypred = predict(gprMdl,xtest);
```

Plot the original response along with the fitted values.

```
figure;  
plot(ytest, 'r');  
hold on;  
plot(ypred, 'b');  
legend('True response', 'GPR predicted values', 'Location', 'Best');  
hold off
```



### Optimize GPR Regression

This example shows how to optimize hyperparameters automatically using `fitrgp`. The example uses the `gprdata2` data that ships with your software.

Load the data.

```
load(fullfile(matlabroot, 'examples', 'stats', 'gprdata2.mat'))
```

The data has one predictor variable and continuous response. This is simulated data.

Fit a GPR model using the squared exponential kernel function with default kernel parameters.

```
gprMdl1 = fitrgp(x,y,'KernelFunction','squaredexponential');
```

Find hyperparameters that minimize five-fold cross-validation loss by using automatic hyperparameter optimization.

For reproducibility, set the random seed and use the 'expected-improvement-plus' acquisition function.

```
rng default
gprMdl2 = fitrgp(x,y,'KernelFunction','squaredexponential',...
    'OptimizeHyperparameters','auto','HyperparameterOptimizationOptions',...
    struct('AcquisitionFunctionName','expected-improvement-plus'));
```

Iter	Eval result	Objective	Objective runtime	BestSoFar (observed)	BestSoFar (estim.)	Sigma
1	Best	0.29417	3.6357	0.29417	0.29417	0.0015045
2	Best	0.037898	3.1743	0.037898	0.060792	0.14147
3	Accept	1.5693	1.7566	0.037898	0.040633	25.279
4	Accept	0.29417	3.4991	0.037898	0.037984	0.0001091
5	Accept	0.29393	3.5311	0.037898	0.038029	0.029932
6	Accept	0.13152	2.7134	0.037898	0.038127	0.37127
7	Best	0.037785	3.2644	0.037785	0.037728	0.18116
8	Accept	0.03783	3.0217	0.037785	0.036524	0.16251
9	Accept	0.037833	3.0756	0.037785	0.036854	0.16159
10	Accept	0.037835	3.3135	0.037785	0.037052	0.16072
11	Accept	0.29417	3.2393	0.037785	0.03705	0.00038214
12	Accept	0.42256	2.6638	0.037785	0.03696	3.2067
13	Accept	0.03786	3.5368	0.037785	0.037087	0.15245
14	Accept	0.29417	3.6541	0.037785	0.037043	0.0063584
15	Accept	0.42302	2.4271	0.037785	0.03725	1.2221
16	Accept	0.039486	2.4104	0.037785	0.037672	0.10069
17	Accept	0.038591	2.684	0.037785	0.037687	0.12077
18	Accept	0.038513	2.5715	0.037785	0.037696	0.1227
19	Best	0.037757	2.7562	0.037757	0.037572	0.19621
20	Accept	0.037787	2.867	0.037757	0.037601	0.18068
Iter	Eval result	Objective	Objective runtime	BestSoFar (observed)	BestSoFar (estim.)	Sigma
21	Accept	0.44917	1.9624	0.037757	0.03766	8.7818

	22		Accept		0.040201		3.2789		0.037757		0.037601		0.075414
	23		Accept		0.040142		2.7361		0.037757		0.037607		0.087198
	24		Accept		0.29417		3.3064		0.037757		0.03758		0.0031018
	25		Accept		0.29417		3.1828		0.037757		0.037555		0.00019545
	26		Accept		0.29417		2.9986		0.037757		0.037582		0.013608
	27		Accept		0.29417		3.217		0.037757		0.037556		0.00076147
	28		Accept		0.42162		2.7789		0.037757		0.037854		0.6791
	29		Best		0.037704		2.6446		0.037704		0.037908		0.2367
	30		Accept		0.037725		3.0212		0.037704		0.037881		0.21743

---

Optimization completed.

MaxObjectiveEvaluations of 30 reached.

Total function evaluations: 30

Total elapsed time: 126.9812 seconds.

Total objective function evaluation time: 88.9227

Best observed feasible point:

Sigma

---

0.2367

Observed objective function value = 0.037704

Estimated objective function value = 0.037881

Function evaluation time = 2.6446

Best estimated feasible point (according to models):

Sigma

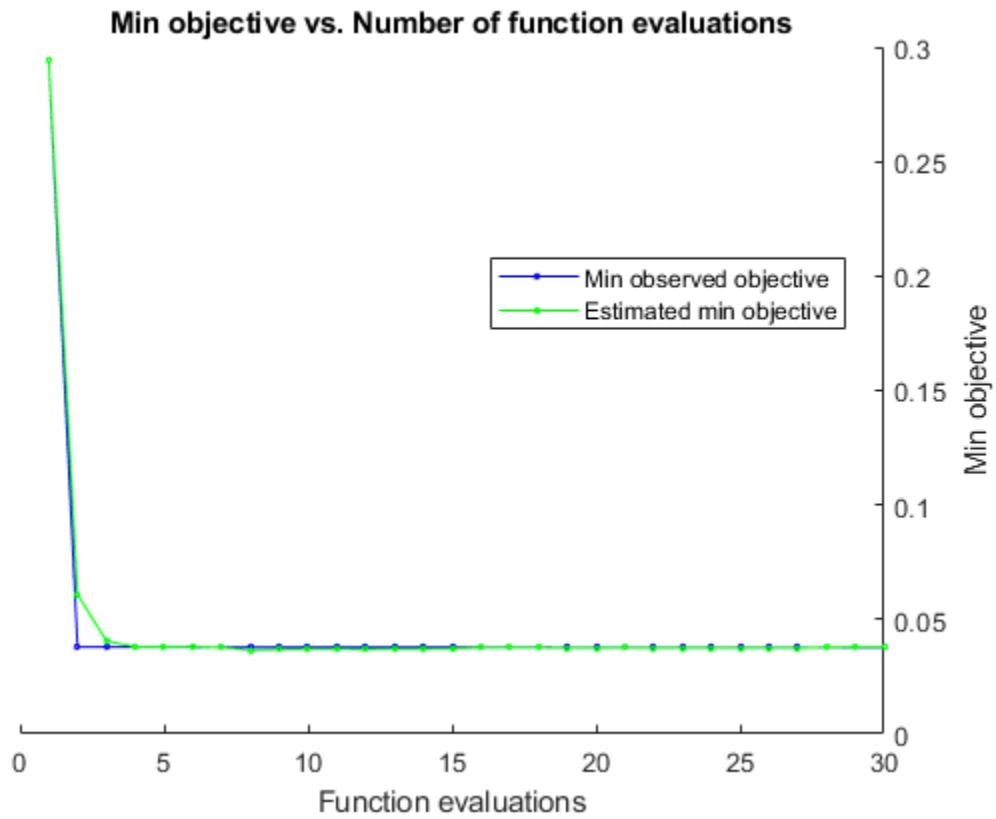
---

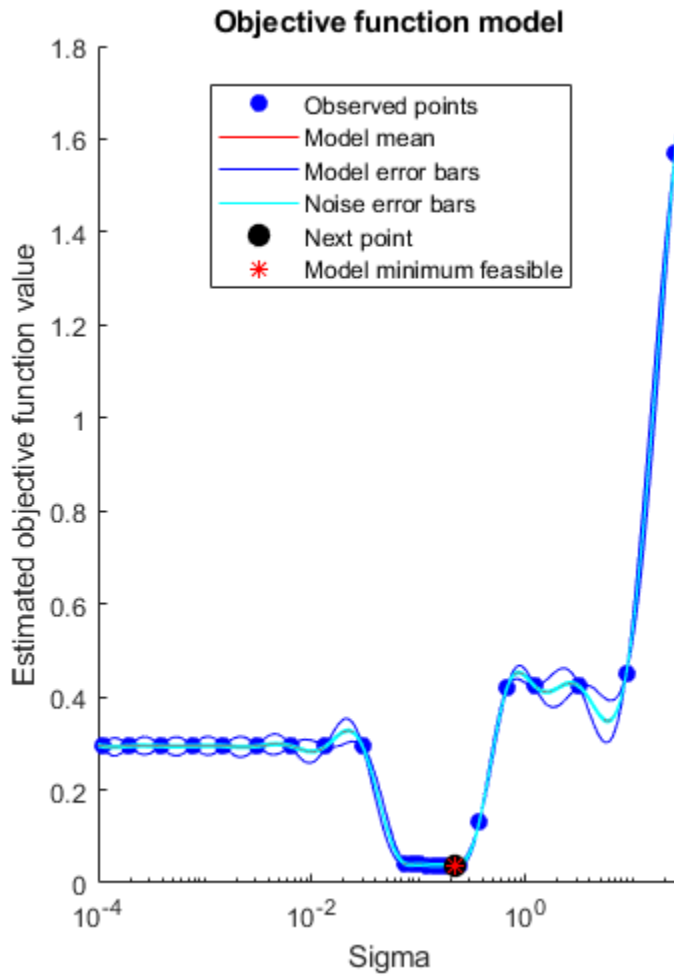
0.16159

Estimated objective function value = 0.037881

Estimated function evaluation time = 2.9327





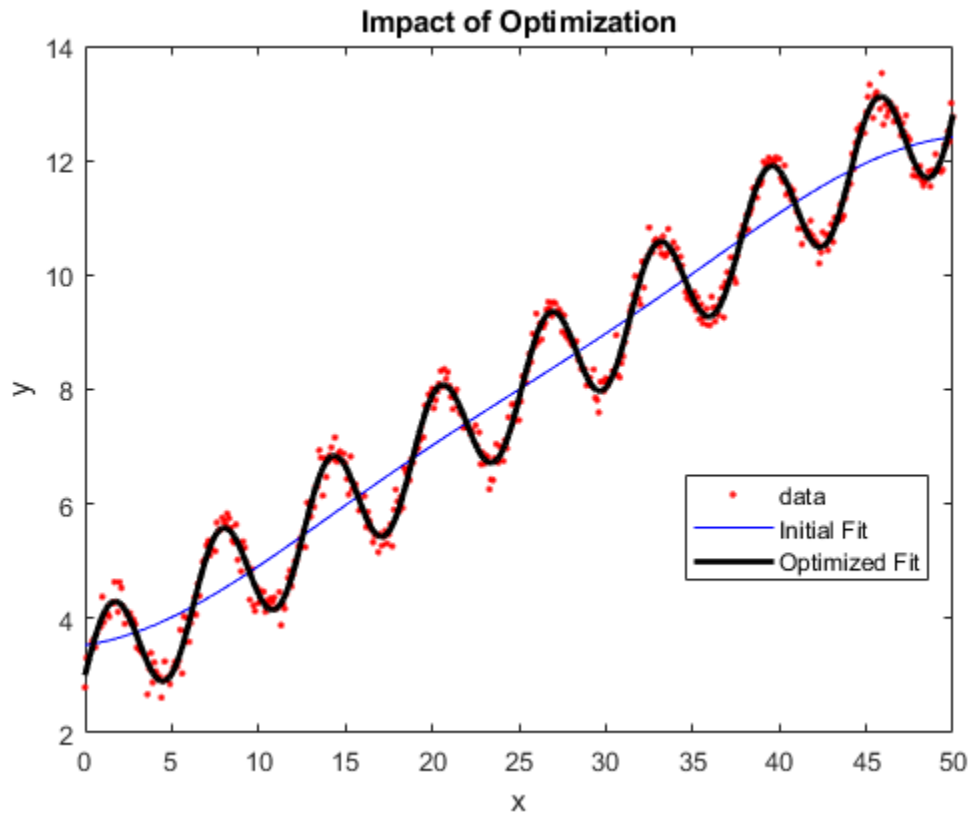


Compare the pre- and post-optimization fits.

```
ypred1 = resubPredict(gprMd11);
ypred2 = resubPredict(gprMd12);
```

```
figure();
plot(x, y, 'r. ');
hold on
```

```
plot(x,ypred1,'b');  
plot(x,ypred2,'k','LineWidth',2);  
xlabel('x');  
ylabel('y');  
legend({'data','Initial Fit','Optimized Fit'},'Location','Best');  
title('Impact of Optimization');  
hold off
```



### Train GPR Model Using Cross-Validation

This example uses the abalone data [1], [2], from the UCI Machine Learning Repository [3]. Download the data and save it in your current folder with the name 'abalone.data'.

Store the data into a table. Display the first seven rows.

```
tbl = readtable('abalone.data', 'Filetype', 'text', 'ReadVariableNames', false); tbl.Properties
tbl(1:7, :)
```

```
ans =
```

Sex	Length	Diameter	Height	WWeight	SWeight	VWeight	ShWeight
'M'	0.455	0.365	0.095	0.514	0.2245	0.101	0.15
'M'	0.35	0.265	0.09	0.2255	0.0995	0.0485	0.07
'F'	0.53	0.42	0.135	0.677	0.2565	0.1415	0.21
'M'	0.44	0.365	0.125	0.516	0.2155	0.114	0.155
'I'	0.33	0.255	0.08	0.205	0.0895	0.0395	0.055
'I'	0.425	0.3	0.095	0.3515	0.141	0.0775	0.12
'F'	0.53	0.415	0.15	0.7775	0.237	0.1415	0.33

The dataset has 4177 observations. The goal is to predict the age of abalone from eight physical measurements. The last variable, number of shell rings shows the age of the abalone. The first predictor is a categorical variable. The last variable in the table is the response variable.

Train a cross-validated GPR model using the 25% of the data for validation.

```
rng('default') % For reproducibility
cvgprMdl = fitrgp(tbl, 'NoShellRings', 'Standardize', 1, 'Holdout', 0.25);
```

Compute the average loss on folds using models trained on out-of-fold observations.

```
kfoldLoss(cvgprMdl)
```

```
ans =
    4.6409
```

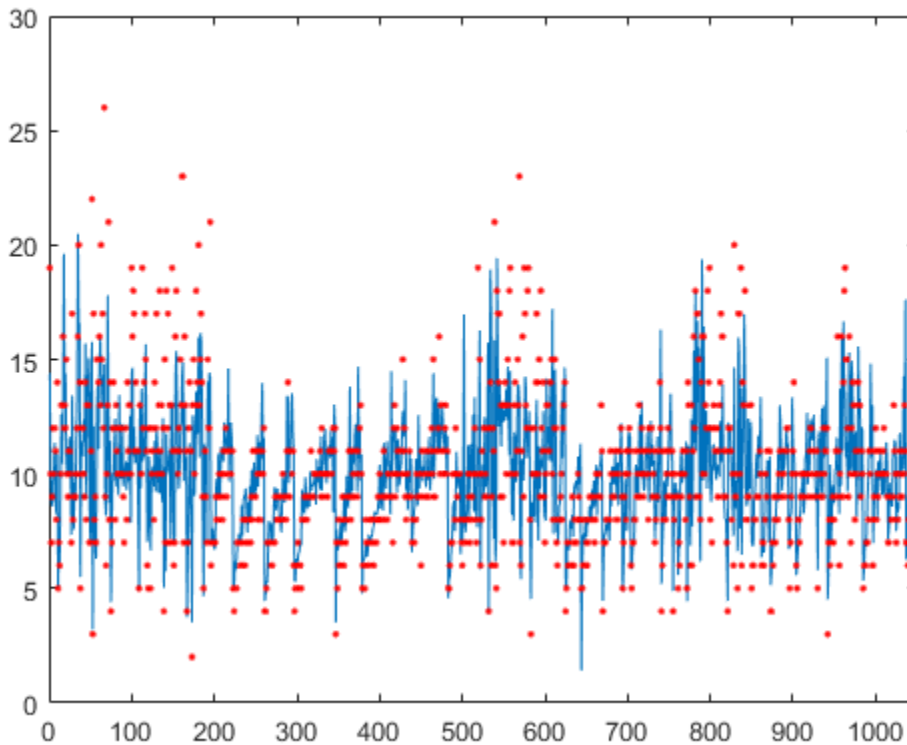
Predict the responses for out-of-fold data.

```
ypred = kfoldPredict(cvgprMdl);
```

Plot the true responses used for testing and the predictions.

```
figure();
plot(ypred(cvgprMdl.Partition.test));
hold on;
```

```
y = table2array(tbl(:,end));  
plot(y(cvgprMdl.Partition.test), 'r. ');  
axis([0 1050 0 30]);  
xlabel('x')  
ylabel('y')  
hold off;
```



### Fit GPR Model Using Custom Kernel Function

Generate the sample data.

```
rng(0, 'twister'); % For reproducibility  
n = 1000;
```

```
x = linspace(-10,10,n)';
y = 1 + x*5e-2 + sin(x)./x + 0.2*randn(n,1);
```

Define the squared exponential kernel function as a custom kernel function.

You can compute the squared exponential kernel function as

$$k(x_i, x_j | \theta) = \sigma_f^2 \exp\left(-\frac{1}{2} \frac{(x_i - x_j)^T (x_i - x_j)}{\sigma_l^2}\right),$$

where  $\sigma_f$  is the signal standard deviation,  $\sigma_l$  is the length scale. Both  $\sigma_f$  and  $\sigma_l$  must be greater than zero. This condition can be enforced by the unconstrained parametrization,  $\sigma_l = \exp(\theta(1))$  and  $\sigma_f = \exp(\theta(2))$ , for some unconstrained parametrization vector  $\theta$ .

Hence, you can define the squared exponential kernel function as a custom kernel function as follows:

```
kfcn = @(XN, XM, theta) (exp(theta(2))^2) * exp(-(pdist2(XN, XM).^2) / (2*exp(theta(1))^2));
```

Here `pdist2(XN, XM).^2` computes the distance matrix.

Fit a GPR model using the custom kernel function, `kfcn`. Specify the initial values of the kernel parameters (Because you use a custom kernel function, you must provide initial values for the unconstrained parametrization vector, `theta`).

```
theta0 = [1.5, 0.2];
gprMdl = fitrgp(x, y, 'KernelFunction', kfcn, 'KernelParameters', theta0);
```

`fitrgp` uses analytical derivatives to estimate parameters when using a built-in kernel function, whereas when using a custom kernel function it uses numerical derivatives.

Compute the resubstitution loss for this model.

```
L = resubLoss(gprMdl)
```

```
L =
    0.0391
```

Fit the GPR model using the built-in squared exponential kernel function option. Specify the initial values of the kernel parameters (Because you use the built-in custom kernel

function and specifying initial parameter values, you must provide the initial values for the signal standard deviation and length scale(s) directly).

```
sigmaL0 = exp(1.5);
sigmaF0 = exp(0.2);
gprMdl2 = fitrgp(x,y,'KernelFunction','squareexponential','KernelParameters',[sigmaL0,
```

Compute the resubstitution loss for this model.

```
L2 = resubLoss(gprMdl2)
```

```
L2 =
    0.0391
```

The two loss values are the same as expected.

### Specify Initial Step Size for LBFGS Optimization

Train a GPR model on generated data with many predictors. Specify the initial step size for the LBFGS optimizer.

Set the seed and type of the random number generator for reproducibility of the results.

```
rng(0,'twister'); % For reproducibility
```

Generate sample data with 300 observations and 3000 predictors, where the response variable depends on the 4th, 7th, and 13th predictors.

```
N = 300;
P = 3000;
X = rand(N,P);
y = cos(X(:,7)) + sin(X(:,4).*X(:,13)) + 0.1*randn(N,1);
```

Set initial values for the kernel parameters.

```
sigmaL0 = sqrt(P)*ones(P,1); % Length scale for predictors
sigmaF0 = 1; % Signal standard deviation
```

Set initial noise standard deviation to 1.

```
sigmaN0 = 1;
```

Specify  $1e-2$  as the termination tolerance for the relative gradient norm.

```
opts = statset('fitrgp');
opts.TolFun = 1e-2;
```

Fit a GPR model using the initial kernel parameter values, initial noise standard deviation, and an automatic relevance determination (ARD) squared exponential kernel function.

Specify the initial step size as 1 for determining the initial Hessian approximation for an LBFGS optimizer.

```
tic;
gpr = fitrgp(X,y,'KernelFunction','ardsquaredexponential','Verbose',1, ...
    'Optimizer','lbfgs','OptimizerOptions',opts, ...
    'KernelParameters',[sigmaL0;sigmaF0],'Sigma',sigmaN0,'InitialStepSize',1);
```

```
o Parameter estimation: FitMethod = Exact, Optimizer = lbfgs
```

```
o Solver = LBFGS, HessianHistorySize = 15, LineSearchMethod = weakwolfe
```

```
=====
| ITER | FUN VALUE | NORM GRAD | NORM STEP | CURV | GAMMA | ALPH
|=====|
| 0 | 3.004966e+02 | 2.569e+02 | 0.000e+00 | | 3.893e-03 | 0.000
| 1 | 9.525779e+01 | 1.281e+02 | 1.003e+00 | OK | 6.913e-03 | 1.000
| 2 | 3.972026e+01 | 1.647e+01 | 7.639e-01 | OK | 4.718e-03 | 5.000
| 3 | 3.893873e+01 | 1.073e+01 | 1.057e-01 | OK | 3.243e-03 | 1.000
| 4 | 3.859904e+01 | 5.659e+00 | 3.282e-02 | OK | 3.346e-03 | 1.000
| 5 | 3.748912e+01 | 1.030e+01 | 1.395e-01 | OK | 1.460e-03 | 1.000
| 6 | 2.028104e+01 | 1.380e+02 | 2.010e+00 | OK | 2.326e-03 | 1.000
| 7 | 2.001849e+01 | 1.510e+01 | 9.685e-01 | OK | 2.344e-03 | 1.000
| 8 | -7.706109e+00 | 8.340e+01 | 1.125e+00 | OK | 5.771e-04 | 1.000
| 9 | -1.786074e+01 | 2.323e+02 | 2.647e+00 | OK | 4.217e-03 | 1.250
| 10 | -4.058422e+01 | 1.972e+02 | 6.796e-01 | OK | 7.035e-03 | 1.000
| 11 | -7.850209e+01 | 4.432e+01 | 8.335e-01 | OK | 3.099e-03 | 1.000
| 12 | -1.312162e+02 | 3.334e+01 | 1.277e+00 | OK | 5.432e-02 | 1.000
| 13 | -2.005064e+02 | 9.519e+01 | 2.828e+00 | OK | 5.292e-03 | 1.000
| 14 | -2.070150e+02 | 1.898e+01 | 1.641e+00 | OK | 6.817e-03 | 1.000
| 15 | -2.108086e+02 | 3.793e+01 | 7.685e-01 | OK | 3.479e-03 | 1.000
| 16 | -2.122920e+02 | 7.057e+00 | 1.591e-01 | OK | 2.055e-03 | 1.000
| 17 | -2.125610e+02 | 4.337e+00 | 4.818e-02 | OK | 1.974e-03 | 1.000
| 18 | -2.130162e+02 | 1.178e+01 | 8.891e-02 | OK | 2.856e-03 | 1.000
| 19 | -2.139378e+02 | 1.933e+01 | 2.371e-01 | OK | 1.029e-02 | 1.000
|=====|
| ITER | FUN VALUE | NORM GRAD | NORM STEP | CURV | GAMMA | ALPH
```



```

=====
| 20 | -2.151111e+02 | 1.550e+01 | 3.015e-01 | OK | 2.765e-02 | 1.000
| 21 | -2.173046e+02 | 5.856e+00 | 6.537e-01 | OK | 1.414e-02 | 1.000
| 22 | -2.201781e+02 | 8.918e+00 | 8.484e-01 | OK | 6.381e-03 | 1.000
| 23 | -2.288858e+02 | 4.846e+01 | 2.311e+00 | OK | 2.661e-03 | 1.000
| 24 | -2.392171e+02 | 1.190e+02 | 6.283e+00 | OK | 8.113e-03 | 1.000
| 25 | -2.511145e+02 | 1.008e+02 | 1.198e+00 | OK | 1.605e-02 | 1.000
| 26 | -2.742547e+02 | 2.207e+01 | 1.231e+00 | OK | 3.191e-03 | 1.000
| 27 | -2.849931e+02 | 5.067e+01 | 3.660e+00 | OK | 5.184e-03 | 1.000
| 28 | -2.899797e+02 | 2.068e+01 | 1.162e+00 | OK | 6.270e-03 | 1.000
| 29 | -2.916723e+02 | 1.816e+01 | 3.213e-01 | OK | 1.415e-02 | 1.000
| 30 | -2.947674e+02 | 6.965e+00 | 1.126e+00 | OK | 6.339e-03 | 1.000
| 31 | -2.962491e+02 | 1.349e+01 | 2.352e-01 | OK | 8.999e-03 | 1.000
| 32 | -3.004921e+02 | 1.586e+01 | 9.880e-01 | OK | 3.940e-02 | 1.000
| 33 | -3.118906e+02 | 1.889e+01 | 3.318e+00 | OK | 1.213e-01 | 1.000
| 34 | -3.189215e+02 | 7.086e+01 | 3.070e+00 | OK | 8.095e-03 | 1.000
| 35 | -3.245557e+02 | 4.366e+00 | 1.397e+00 | OK | 2.718e-03 | 1.000
| 36 | -3.254613e+02 | 3.751e+00 | 6.546e-01 | OK | 1.004e-02 | 1.000
| 37 | -3.262823e+02 | 4.011e+00 | 2.026e-01 | OK | 2.441e-02 | 1.000
| 38 | -3.325606e+02 | 1.773e+01 | 2.427e+00 | OK | 5.234e-02 | 1.000
| 39 | -3.350374e+02 | 1.201e+01 | 1.603e+00 | OK | 2.674e-02 | 1.000
=====

```

```

=====
| ITER | FUN VALUE | NORM GRAD | NORM STEP | CURV | GAMMA | ALPH
=====
| 40 | -3.379112e+02 | 5.280e+00 | 1.393e+00 | OK | 1.177e-02 | 1.000
| 41 | -3.389136e+02 | 3.061e+00 | 7.121e-01 | OK | 2.935e-02 | 1.000
| 42 | -3.401070e+02 | 4.094e+00 | 6.224e-01 | OK | 3.399e-02 | 1.000
| 43 | -3.436291e+02 | 8.833e+00 | 1.707e+00 | OK | 5.231e-02 | 1.000
| 44 | -3.456295e+02 | 5.891e+00 | 1.424e+00 | OK | 3.772e-02 | 1.000
| 45 | -3.460069e+02 | 1.126e+01 | 2.580e+00 | OK | 3.907e-02 | 1.000
| 46 | -3.481756e+02 | 1.546e+00 | 8.142e-01 | OK | 1.565e-02 | 1.000
=====

```

```

Infinity norm of the final gradient = 1.546e+00
Two norm of the final step = 8.142e-01, TolX = 1.000e-12
Relative infinity norm of the final gradient = 6.016e-03, TolFun = 1.000e-02
EXIT: Local minimum found.

```

o Alpha estimation: PredictMethod = Exact

toc;

Elapsed time is 642.772860 seconds.

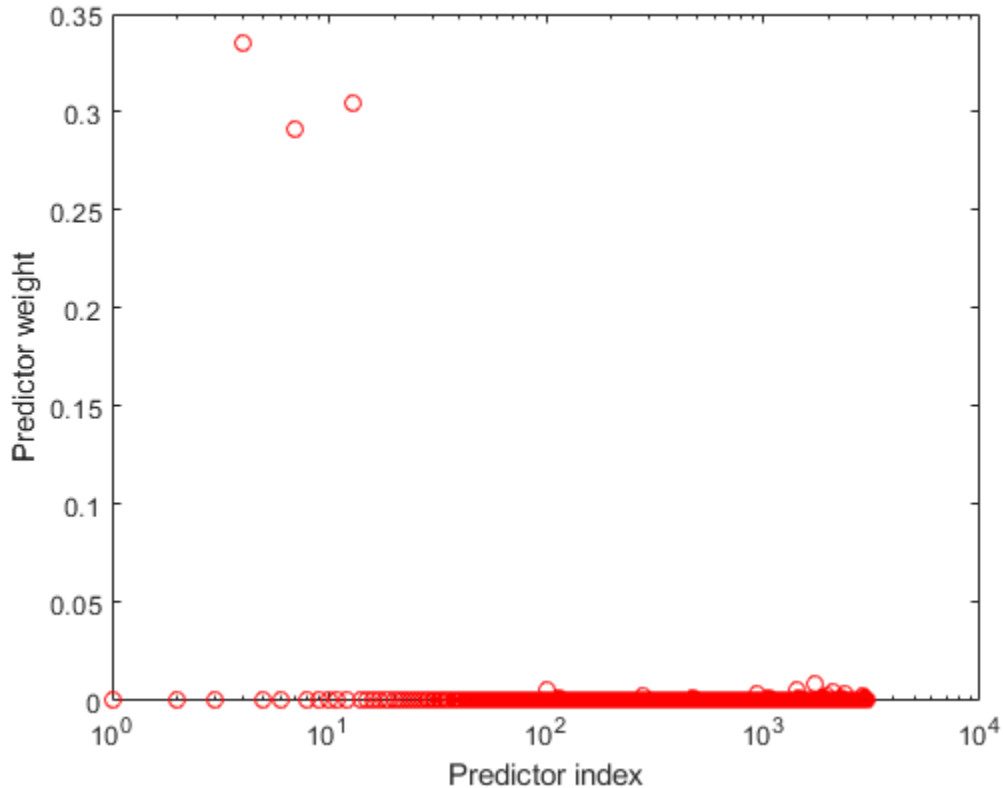
Because the GPR model uses an ARD kernel with many predictors, using an LBFGS approximation to the Hessian is more memory efficient than storing the full Hessian matrix. Also, using the initial step size to determine the initial Hessian approximation may help speed up optimization.

Find the predictor weights by taking the exponential of the negative learned length scales. Normalize the weights.

```
sigmaL = gpr.KernelInformation.KernelParameters(1:end-1); % Learned length scales
weights = exp(-sigmaL); % Predictor weights
weights = weights/sum(weights); % Normalized predictor weights
```

Plot the normalized predictor weights.

```
figure;
semilogx(weights, 'ro');
xlabel('Predictor index');
ylabel('Predictor weight');
```



The trained GPR model assigns the largest weights to the 4th, 7th, and 13th predictors. The irrelevant predictors have weights close to zero.

## Input Arguments

### `tbl` — Sample data

table

Sample data used to train the model, specified as a `table`. Each row of `tbl` corresponds to one observation, and each column corresponds to one variable. `tbl` contains the predictor variables, and optionally it can also contain one column for the response

variable. Multi-column variables and cell arrays other than cell arrays of character vectors are not allowed.

- If `tbl` contains the response variable, and you want to use all the remaining variables as predictors, then specify the response variable using `ResponseVarName`.
- If `tbl` contains the response variable, and you want to use only a subset of the predictors in training the model, then specify the response variable and the predictor variables using `formula`.
- If `tbl` does not contain the response variable, then specify a response variable using `y`. The length of the response variable and the number of rows in `tbl` must be equal.

For more information on the `table` data type, please see `table`.

If your predictor data contains categorical variables, then the software uses full dummy coding for these variables. The software creates one dummy variable for each level of the categorical variable.

Data Types: `table`

### **ResponseVarName** — Response variable name

name of a variable in `tbl`

Response variable name, specified as the name of a variable in `tbl`. You must specify `ResponseVarName` as a character vector. For example, if the response variable `y` is stored in `tbl` (as `tbl.y`), then specify it as `'y'`. Otherwise, the software treats all the columns of `tbl`, including `y`, as predictors when training the model.

Data Types: `char`

### **formula** — Response and predictor variables to use in model training

character vector in the form of `'y~x1+x2+x3'`

Response and predictor variables to use in model training, specified as a character vector in the form of `'y~x1+x2+x3'`. In this form, `y` represents the response variable; `x1`, `x2`, `x3` represent the predictor variables to use in training the model.

Use `formula` character vector if you want to specify a subset of variables in `tbl` as predictors to use when training the model. If you specify a `formula` character vector, then any variables that do not appear in `formula` are not used to train the model.

The `formula` does not indicate the form of the `BasisFunction`.

Example: `'PetalLength~PetalWidth+Species'` identifies the variable `PetalLength` as the response variable, and `PetalWidth` and `Species` as the predictor variables.

### **x** — Predictor data for the GPR model

*n*-by-*d* matrix

Predictor data for the GPR model, specified as an *n*-by-*d* matrix. *n* is the number of observations (rows), and *d* is the number of predictors (columns).

The length of *y* and the number of rows of *X* must be equal.

To specify the names of the predictors in the order of their appearance in *X*, use the `PredictorNames` name-value pair argument.

Data Types: `double`

### **y** — Response data for the GPR model

*n*-by-1 vector

Response data for the GPR model, specified as an *n*-by-1 vector. You can omit *y* if you provide the `tbl` training data that also includes *y*. In that case, use `ResponseVarName` to identify the response variable or use `formula` to identify the response and predictor variables.

Data Types: `double` | `char`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

---

**Note** You cannot use any cross-validation name-value pair along with `OptimizeHyperparameters`. You can modify the cross-validation for `OptimizeHyperparameters` only by using the `HyperparameterOptimizationOptions` name-value pair.

---

Example:

```
'FitMethod','sr','BasisFunction','linear','ActiveSetMethod','sgma','
```

PredictMethod', fic' trains the GPR model using the subset of regressors approximation method for parameter estimation, uses a linear basis function, uses sparse greedy matrix approximation for active selection, and fully independent conditional approximation method for prediction.

### Fitting

#### FitMethod — Method to estimate parameters of the GPR model

'none' | 'exact' | 'sd' | 'sr' | 'fic'

Method to estimate parameters of the GPR model, specified as the comma-separated pair consisting of 'FitMethod' and one of the following.

Fit Method	Description
'none'	No estimation, use the initial parameter values as the known parameter values.
'exact'	Exact Gaussian process regression. Default if $n \leq 2000$ , where $n$ is the number of observations.
'sd'	Subset of data points approximation. Default if $n > 2000$ , where $n$ is the number of observations.
'sr'	Subset of regressors approximation.
'fic'	Fully independent conditional approximation.

Example: 'FitMethod', 'fic'

#### BasisFunction — Explicit basis in the GPR model

'constant' (default) | 'none' | 'linear' | 'pureQuadratic' | function handle

Explicit basis in the GPR model, specified as the comma-separated pair consisting of 'BasisFunction' and one of the following. If  $n$  is the number of observations, the basis function adds the term  $\mathbf{H}^*\boldsymbol{\beta}$  to the model, where  $\mathbf{H}$  is the basis matrix and  $\boldsymbol{\beta}$  is a  $p$ -by-1 vector of basis coefficients.

Explicit Basis	Basis Matrix
'none'	Empty matrix.

Explicit Basis	Basis Matrix
'constant'	$\mathbf{H} = \mathbf{1}$ ( $n$ -by-1 vector of 1s, where $n$ is the number of observations)
'linear'	$\mathbf{H} = [\mathbf{1}, \mathbf{X}]$
'pureQuadratic'	$\mathbf{H} = [\mathbf{1}, \mathbf{X}, \mathbf{X}_2]$ , where $\mathbf{X}_2 = \begin{bmatrix} x_{11}^2 & x_{12}^2 & \cdots & x_{1d}^2 \\ x_{21}^2 & x_{22}^2 & \cdots & x_{2d}^2 \\ \vdots & \vdots & \vdots & \vdots \\ x_{n1}^2 & x_{n2}^2 & \cdots & x_{nd}^2 \end{bmatrix}.$
Function handle	Function handle, <i>hfcn</i> , that <code>fitrgp</code> calls as: $\mathbf{H} = \mathit{hfcn}(\mathbf{X})$ , where $\mathbf{X}$ is an $n$ -by- $d$ matrix of predictors and $\mathbf{H}$ is an $n$ -by- $p$ matrix of basis functions.

Example: 'BasisFunction', 'pureQuadratic'

Data Types: char | function\_handle

### Beta — Initial value of the coefficients

$p$ -by-1 vector

Initial value of the coefficients for the explicit basis, specified as the comma-separated pair consisting of 'Beta' and  $p$ -by-1 vector, where  $p$  is the number of columns in the basis matrix  $\mathbf{H}$ .

The basis matrix depends on the choice of the explicit basis function as follows (also see `BasisFunction`).

`fitrgp` uses the coefficient initial values as the known coefficient values, only when `FitMethod` is 'none'.

Data Types: double

**Sigma** — Initial value for the noise standard deviation of the Gaussian process model

`std(y)/sqrt(2)` (default) | positive scalar value

Initial value for the noise standard deviation of the Gaussian process model, specified as the comma-separated pair consisting of 'Sigma' and a positive scalar value.

Example: 'Sigma',2

Data Types: double

**ConstantSigma** — Constant value of sigma for the noise standard deviation of the Gaussian process model

`false` (default) | `true`

Constant value of Sigma for the noise standard deviation of the Gaussian process model, specified as a logical scalar. When ConstantSigma is true, `fitrgp` does not optimize the value of Sigma, but instead takes the initial value as the value throughout its computations.

Example: 'ConstantSigma',true

Data Types: logical

**SigmaLowerBound** — Lower bound on the noise standard deviation

`1e-2*std(y)` (default) | positive scalar value

Lower bound on the noise standard deviation, specified as the comma-separated pair consisting of 'SigmaLowerBound' and a positive scalar value.

Example: 'SigmaLowerBound',0.02

Data Types: double

**CategoricalPredictors** — Categorical predictors list

vector of positive integers | logical vector | character matrix | cell array of character vectors | 'all'

Categorical predictors list, specified as the comma-separated pair consisting of 'CategoricalPredictors' and one of these values:



Value	Description
Vector of positive integers	An entry in the vector is the index value corresponding to the column of the predictor data ( <code>X</code> or <code>tbl</code> ) that contains a categorical variable.
Logical vector	A <code>true</code> entry means that the corresponding column of predictor data ( <code>X</code> or <code>tbl</code> ) is a categorical variable.
Character matrix	Each row of the matrix is the name of a predictor variable. The names must match the entries in <code>PredictorNames</code> . Pad the names with extra blanks so each row of the character matrix has the same length.
Cell array of character vectors	Each element in the array is the name of a predictor variable. The names must match the entries in <code>PredictorNames</code> .
'all'	All predictors are categorical.

By default, if the predictor data is in a table (`tbl`), `fitrgp` assumes that a variable is categorical if it contains logical values, categorical values, or a cell array of character vectors. If the predictor data is a matrix (`X`), `fitrgp` assumes all predictors are continuous. To identify any categorical predictors when the data is a matrix, use the `'CategoricalPredictors'` name-value pair argument.

Example: `'CategoricalPredictors','all'`

Data Types: `single` | `double` | `logical` | `char` | `cell`

### **Standardize** — Indicator to standardize data

0 (`false`) (default) | logical value

Indicator to standardize data, specified as the comma-separated pair consisting of `'Standardize'` and a logical value.

If you set `'Standardize',1`, then the software centers and scales each column of the predictor data, by the column mean and standard deviation, respectively. The software does not standardize the data contained in the dummy variable columns that it generates for categorical predictors.

Example: `'Standardize',1`

Example: `'Standardize',true`

Data Types: `logical`

**Regularization — Regularization standard deviation**

`1e-2*std(y)` (default) | positive scalar value

Regularization standard deviation for sparse methods subset of regressors ('sr') and fully independent conditional ('fic'), specified as the comma-separated pair consisting of 'Regularization' and a positive scalar value.

Example: 'Regularization',0.2

Data Types: double

**ComputationMethod — Method for computing log likelihood and gradient**

'qr' (default) | 'v'

Method for computing the log likelihood and gradient for parameter estimation using subset of regressors ('sr') and fully independent conditional ('fic') approximation methods, specified as the comma-separated pair consisting of 'ComputationMethod' and one of the following.

- 'qr' — Use QR factorization based approach, this option provides better accuracy.
- 'v' — Use V-method-based approach. This option provides faster computation of log likelihood gradients.

Example: 'ComputationMethod','v'

**Kernel (Covariance) Function****KernelFunction — Form of the covariance function**

'squarexponential' (default) | 'exponential' | 'matern32' | 'matern52' | 'rationalquadratic' | 'ardsquarexponential' | 'ardexponential' | 'ardmatern32' | 'ardmatern52' | 'ardrationalquadratic' | function handle

Form of the covariance function, specified as the comma-separated pair consisting of 'KernelFunction' and one of the following.

Function	Description
'exponential'	Exponential kernel.
'squarexponential'	Squared exponential kernel.
'matern32'	Matern kernel with parameter 3/2.
'matern52'	Matern kernel with parameter 5/2.

Function	Description
'rationalquadratic'	Rational quadratic kernel.
'ardexponential'	Exponential kernel with a separate length scale per predictor.
'ardsquaredexponential'	Squared exponential kernel with a separate length scale per predictor.
'ardmatern32'	Matern kernel with parameter 3/2 and a separate length scale per predictor.
'ardmatern52'	Matern kernel with parameter 5/2 and a separate length scale per predictor.
'ardrationalquadratic'	Rational quadratic kernel with a separate length scale per predictor.
Function handle	A function handle that can be called like this: $K_{mn} = \text{kfcn}(X_m, X_n, \theta)$ where $X_m$ is an $m$ -by- $d$ matrix, $X_n$ is an $n$ -by- $d$ matrix and $K_{mn}$ is an $m$ -by- $n$ matrix of kernel products such that $K_{mn}(i,j)$ is the kernel product between $X_m(i,:)$ and $X_n(j,:)$ . $\theta$ is the $r$ -by-1 unconstrained parameter vector for $\text{kfcn}$ .

For more information on the kernel functions, see “Kernel (Covariance) Function Options” on page 6-6.

Example: 'KernelFunction', 'matern32'

Data Types: char | function\_handle

**KernelParameters** — Initial values for the kernel parameters vector

Initial values for the kernel parameters, specified as the comma-separated pair consisting of 'KernelParameters' and a vector. The size of the vector and the values depend on the form of the covariance function, specified by the KernelFunction name-value pair argument.

'KernelFunction'	'KernelParameters'
'exponential' or 'squarexponential' or 'matern32' or 'matern52'	2-by-1 vector phi, where phi (1) contains the length scale and phi (2) contains the signal standard deviation. Default initial value of the length scale parameter is the mean of standard deviations of the predictors, and the signal standard deviation is the standard deviation of the responses divided by square root of 2. That is, phi = [mean(std(X));std(y)/sqrt(2)]
'rationalquadratic'	3-by-1 vector phi, where phi (1) contains the length scale, phi (2) contains the scale-mixture parameter, and phi (3) contains the signal standard deviation. Default initial value of the length scale parameter is the mean of standard deviations of the predictors, and the signal standard deviation is the standard deviation of the responses divided by square root of 2. Default initial value for the scale-mixture parameter is 1. That is, phi = [mean(std(X));1;std(y)/sqrt(2)]
'ardexponential' or 'ardsquarexponential' or 'ardmatern32' or 'ardmatern52'	$(d+1)$ -by-1 vector phi, where phi (i) contains the length scale for predictor $i$ and phi ( $d+1$ ) contains the signal standard deviation. $d$ is the number of predictor variables. Default initial value of the length scale parameters are the standard deviations of the predictors and the signal standard deviation is the standard deviation of the responses divided by square root of 2. That is, phi = [std(X)';std(y)/sqrt(2)]

'KernelFunction'	'KernelParameters'
'ardrationalquadratic'	<p><math>(d+2)</math>-by-1 vector <math>\phi</math>, where <math>\phi(i)</math> contains the length scale for predictor <math>i</math>, <math>\phi(d+1)</math> contains the scale-mixture parameter, and <math>\phi(d+2)</math> contains signal standard deviation.</p> <p>Default initial value of the length scale parameters are the standard deviations of the predictors and the signal standard deviation is the standard deviation of the responses divided by square root of 2.</p> <p>Default initial value for the scale-mixture parameter is 1. That is,</p> $\phi = [\text{std}(X)'; 1; \text{std}(y)/\text{sqrt}(2)]$
Function handle	<p><math>r</math>-by-1 vector as the initial value of the unconstrained parameter vector <math>\phi</math> for the custom kernel function <math>kfcn</math>.</p> <p>When <code>KernelFunction</code> is a function handle, you must supply initial values for the kernel parameters.</p>

For more information on the kernel functions, see “Kernel (Covariance) Function Options” on page 6-6.

Example: `'KernelParameters', theta`

Data Types: double

#### **DistanceMethod — Method for computing inter-point distances**

`'fast'` (default) | `'accurate'`

Method for computing inter-point distances to evaluate built-in kernel functions, specified as the comma-separated pair consisting of `'DistanceMethod'` and either

`'fast'` or `'accurate'`. `fitrgp` computes  $(x-y)^2$  as  $x^2 + y^2 - 2*x*y$  when you choose the `fast` option and as  $(x-y)^2$  when you choose the `accurate` option.

Example: `'DistanceMethod', 'accurate'`

### Active Set Selection

#### ActiveSet — Observations in the active set

[] (default) |  $m$ -by-1 vector of integers ranging from 1 to  $n$  ( $m \leq n$ ) | logical vector of length  $n$

Observations in the active set, specified as the comma-separated pair consisting of 'ActiveSet' and an  $m$ -by-1 vector of integers ranging from 1 to  $n$  ( $m \leq n$ ) or a logical vector of length  $n$  with at least one true element.  $n$  is the total number of observations in the training data.

fitrgp uses the observations indicated by ActiveSet to train the GPR model. The active set cannot have duplicate elements.

If you supply ActiveSet, then:

- fitrgp does not use ActiveSetSize and ActiveSetMethod.
- You cannot perform cross-validation on this model.

Data Types: double | logical

#### ActiveSetSize — Size of the active set for sparse methods

an integer  $m$  ( $1 \leq m \leq n$ )

Size of the active set for sparse methods ('sd', 'sr', 'fic'), specified as the comma-separated pair consisting of 'ActiveSetSize' and an integer  $m$ ,  $1 \leq m \leq n$ , where  $n$  is the number of observations.

Default is  $\min(1000, n)$  when FitMethod is 'sr' or 'fic', and  $\min(2000, n)$ , otherwise.

Example: 'ActiveSetSize', 100

Data Types: double

#### ActiveSetMethod — Active set selection method

'random' (default) | 'sgma' | 'entropy' | 'likelihood'

Active set selection method, specified as the comma-separated pair consisting of 'ActiveSetMethod' and one of the following.

Method	Description
'random'	Random selection

Method	Description
'sgma'	Sparse greedy matrix approximation
'entropy'	Differential entropy-based selection
'likelihood'	Subset of regressors log likelihood-based selection

All active set selection methods (except 'random') require the storage of an  $n$ -by- $m$  matrix, where  $m$  is the size of the active set and  $n$  is the number of observations.

Example: 'ActiveSetMethod', 'entropy'

#### **RandomSearchSetSize — Random search set size**

59 (default) | integer value

Random search set size per greedy inclusion for active set selection, specified as the comma-separated pair consisting of 'RandomSearchSetSize' and an integer value.

Example: 'RandomSearchSetSize', 30

Data Types: double

#### **ToleranceActiveSet — Relative tolerance for terminating active set selection**

1e-06 (default) | positive scalar

Relative tolerance for terminating active set selection, specified as the comma-separated pair consisting of 'ToleranceActiveSet' and a positive scalar value.

Example: 'ToleranceActiveSet', 0.0002

Data Types: double

#### **NumActiveSetRepeats — Number of repetitions**

3 (default) | integer value

Number of repetitions for interleaved active set selection and parameter estimation on page 32-2375 when ActiveSetMethod is not 'random', specified as the comma-separated pair consisting of 'NumActiveSetRepeats' and an integer value.

Example: 'NumActiveSetRepeats', 5

Data Types: double

**Prediction****PredictMethod — Method used to make predictions**`'exact' | 'bcd' | 'sd' | 'sr' | 'fic'`

Method used to make predictions from a Gaussian process model given the parameters, specified as the comma-separated pair consisting of 'PredictMethod' and one of the following.

Method	Description
'exact'	Exact Gaussian process regression method. Default, if $n \leq 10000$ .
'bcd'	Block Coordinate Descent. Default, if $n > 10000$ .
'sd'	Subset of Datapoints approximation.
'sr'	Subset of Regressors approximation.
'fic'	Fully Independent Conditional approximation.

Example: 'PredictMethod', 'bcd'

**BlockSizeBCD — Block size**

minimum of 1000 or  $n$  (default) | integer in the range from 1 to  $n$

Block size for block coordinate descent method ('bcd'), specified as the comma-separated pair consisting of 'BlockSizeBCD' and an integer in the range from 1 to  $n$ , where  $n$  is the number of observations.

Example: 'BlockSizeBCD', 1500

Data Types: double

**NumGreedyBCD — Number of greedy selections**

minimum of 100 and BlockSizeBCD (default) | integer value in the range from 1 to BlockSizeBCD

Number of greedy selections for block coordinate descent method ('bcd'), specified as the comma-separated pair consisting of 'NumGreedyBCD' and an integer in the range from 1 to BlockSizeBCD.

Example: 'NumGreedyBCD', 150



Data Types: double

**ToleranceBCD — Relative tolerance on gradient norm**

1e-3 (default) | positive scalar

Relative tolerance on gradient norm for terminating block coordinate descent method ('bcd') iterations, specified as the comma-separated pair consisting of 'ToleranceBCD' and a positive scalar.

Example: 'ToleranceBCD',0.002

Data Types: double

**StepToleranceBCD — Absolute tolerance on step size**

1e-3 (default) | positive scalar

Absolute tolerance on step size for terminating block coordinate descent method ('bcd') iterations, specified as the comma-separated pair consisting of 'StepToleranceBCD' and a positive scalar.

Example: 'StepToleranceBCD',0.002

Data Types: double

**IterationLimitBCD — Maximum number of BCD iterations**

1000000 (default) | integer value

Maximum number of block coordinate descent method ('bcd') iterations, specified as the comma-separated pair consisting of 'IterationLimitBCD' and an integer value.

Example: 'IterationLimitBCD',10000

Data Types: double

**Optimization**

**Optimizer — Optimizer to use for parameter estimation**

'quasinevton' (default) | 'lbfgs' | 'fminsearch' | 'fminunc' | 'fmincon'

Optimizer to use for parameter estimation, specified as the comma-separated pair consisting of 'Optimizer' and one of the values in this table.

Value	Description
'quasnewton'	Dense, symmetric rank-1-based, quasi-Newton approximation to the Hessian
'lbfgs'	LBFGS-based quasi-Newton approximation to the Hessian
'fminsearch'	Unconstrained nonlinear optimization using the simplex search method of Lagarias et al. [5]
'fminunc'	Unconstrained nonlinear optimization (requires an Optimization Toolbox license)
'fmincon'	Constrained nonlinear optimization (requires an Optimization Toolbox license)

For more information on the optimizers, see Algorithms on page 32-2376.

Example: 'Optimizer', 'fmincon'

### OptimizerOptions — Options for the optimizer

structure | object

Options for the optimizer you choose using the `Optimizer` name-value pair argument, specified as the comma-separated pair consisting of 'OptimizerOptions' and a structure or object created by `optimset`, `statset('fitrgp')`, or `optimoptions`.

Optimizer	Create Optimizer Options Using
'fminsearch'	<code>optimset</code> (structure)
'quasnewton' or 'lbfgs'	<code>statset('fitrgp')</code> (structure)
'fminunc' or 'fmincon'	<code>optimoptions</code> (object)

The default options depend on the type of optimizer.

Example: 'OptimizerOptions', `opt`

### InitialStepSize — Initial step size

[] (default) | real positive scalar | 'auto'

Initial step size, specified as the comma-separated pair consisting of 'InitialStepSize' and a real positive scalar or 'auto'.

'InitialStepSize' is the approximate maximum absolute value of the first optimization step when the optimizer is 'quasinewton' or 'lbfgs'. The initial step size can determine the initial Hessian approximation during optimization.

By default, `fitrgp` does not use the initial step size to determine the initial Hessian approximation. To use the initial step size, set a value for the 'InitialStepSize' name-value pair argument, or specify 'InitialStepSize', 'auto' to have `fitrgp` determine a value automatically. For more information on 'auto', see Algorithms on page 32-2376.

Example: 'InitialStepSize', 'auto'

### Cross-Validation

#### CrossVal — Indicator for cross-validation

'off' (default) | 'on'

Indicator for cross-validation, specified as the comma-separated pair consisting of 'CrossVal' and either 'off' or 'on'. If it is 'on', then `fitrgp` returns a GPR model cross-validated with 10 folds.

You can use one of the `KFold`, `Holdout`, `Leaveout` or `CVPartition` name-value pair arguments to change the default cross-validation settings. You can use only one of these name-value pairs at a time.

As an alternative, you can use the `crossval` method for your model.

Example: 'CrossVal', 'on'

Data Types: double

#### CVPartition — Random partition for a stratified $k$ -fold cross-validation

`cvpartition` object

Random partition for a stratified  $k$ -fold cross-validation, specified as the comma-separated pair consisting of 'CVPartition' and a `cvpartition` object.

Example: 'CVPartition', `cvp` uses the random partition defined by `cvp`.

If you specify `CVPartition`, then you cannot specify `Holdout`, `KFold`, or `Leaveout`.

#### Holdout — Fraction of data to use for testing

scalar value in the range from 0 to 1

Fraction of the data to use for testing in holdout validation, specified as the comma-separated pair consisting of 'Holdout' and a scalar value in the range from 0 to 1. If you specify 'Holdout',  $p$ , then the software:

1. Randomly reserves around  $p*100\%$  of the data as validation data, and trains the model using the rest of the data
2. Stores the compact, trained model in `cvgprMdl.Trained`.

Example: 'Holdout', 0.3 uses 30% of the data for testing and 70% of the data for training.

If you specify `Holdout`, then you cannot specify `CVPartition`, `KFold`, or `Leaveout`.

Data Types: double

### **KFold** — Number of folds

10 (default) | positive integer value

Number of folds to use in cross-validated GPR model, specified as the comma-separated pair consisting of 'KFold' and a positive integer value. `KFold` must be greater than 1. If you specify 'KFold',  $k$  then the software:

1. Randomly partitions the data into  $k$  sets.
2. For each set, reserves the set as test data, and trains the model using the other  $k - 1$  sets.
3. Stores the  $k$  compact, trained models in the cells of a  $k$ -by-1 cell array in `cvgprMdl.Trained`.

Example: 'KFold', 5 uses 5 folds in cross-validation. That is, for each fold, uses that fold as test data, and trains the model on the remaining 4 folds.

If you specify `KFold`, then you cannot specify `CVPartition`, `Holdout`, or `Leaveout`.

Data Types: double

### **Leaveout** — Indicator for leave-one-out cross-validation

'off' (default) | 'on'

Indicator for leave-one-out cross-validation, specified as the comma-separated pair consisting of 'Leaveout' and either 'off' or 'on'.

If you specify 'Leaveout', 'on', then, for each of the  $n$  observations, the software:

1. Reserves the observation as test data, and trains the model using the other  $n - 1$  observations.

2. Stores the compact, trained model in a cell in the  $n$ -by-1 cell array `cvgprMdl.Trained`.

Example: `'Leaveout', 'on'`

If you specify `Leaveout`, then you cannot specify `CVPartition`, `Holdout`, or `KFold`.

## Hyperparameter Optimization

### OptimizeHyperparameters — Parameters to optimize

`'none'` (default) | `'auto'` | `'all'` | cell array of eligible parameter names | vector of `optimizableVariable` objects

Parameters to optimize, specified as:

- `'none'` — Do not optimize.
- `'auto'` — Use `{'Sigma'}`
- `'all'` — Optimize all eligible parameters, equivalent to `{'BasisFunction', 'KernelFunction', 'KernelScale', 'Sigma', 'Standardize'}`
- Cell array of eligible parameter names
- Vector of `optimizableVariable` objects, typically the output of hyperparameters

The optimization attempts to minimize the cross-validation loss (error) for `fitrgp` by varying the parameters. For information about cross-validation loss (albeit in a different context), see “Classification Loss” on page 32-3250. To control the cross-validation type and other aspects of the optimization, use the `HyperparameterOptimizationOptions` name-value pair.

---

**Note** `OptimizeHyperparameters` values override any values you set using other name-value pairs. For example, setting `OptimizeHyperparameters` to `'auto'` causes the `'auto'` values to apply.

---

The eligible parameters for `fitrgp` are:

- `BasisFunction` — `fitrgp` searches among `'constant'`, `'none'`, `'linear'`, and `'pureQuadratic'`.
- `KernelFunction` — `fitrgp` searches among `'ardexponential'`, `'ardmatern32'`, `'ardmatern52'`, `'ardrationalquadratic'`, `'ardsquaredexponential'`,

'exponential', 'matern32', 'matern52', 'rationalquadratic', and 'squarexponential'.

- `KernelScale` — `fitrgp` uses the `KernelParameters` argument to specify the value of the kernel scale parameter, which is held constant during fitting. In this case, all input dimensions are constrained to have the same `KernelScale` value. `fitrgp` searches among real value in the range  $[1e-3 * \text{MaxPredictorRange}, \text{MaxPredictorRange}]$ , where

$\text{MaxPredictorRange} = \max(\max(X) - \min(X))$ .

`KernelScale` cannot be optimized for any of the ARD kernels.

- `Sigma` — `fitrgp` searches among real value in the range  $[1e-4, \max(1e-3, 10 * \text{ResponseStd})]$ , where

$\text{ResponseStd} = \text{std}(y)$ .

Internally, `fitrgp` sets the `ConstantSigma` name-value pair to `true` so the value of `Sigma` is constant during the fitting.

- `Standardize` — `fitrgp` searches among `true` and `false`.

Set nondefault parameters by passing a vector of `optimizableVariable` objects that have nondefault values. For example,

```
load fisheriris
params = hyperparameters('fitrgp', meas, species);
params(1).Range = [1e-4, 1e6];
```

Pass `params` as the value of `OptimizeHyperparameters`.

By default, iterative display appears at the command line, and plots appear according to the number of hyperparameters in the optimization. For the optimization and plots, the objective function is  $\log(1 + \text{cross-validation loss})$  for regression, and the misclassification rate for classification. To control the iterative display, set the `HyperparameterOptimizationOptions` name-value pair, `Verbose` field. To control the plots, set the `HyperparameterOptimizationOptions` name-value pair, `ShowPlots` field.

For an example, see “Optimize GPR Regression” on page 32-2336.

Example: 'auto'

Data Types: char | cell

### HyperparameterOptimizationOptions — Options for optimization structure

Options for optimization, specified as a structure. Modifies the effect of the `OptimizeHyperparameters` name-value pair. All fields in the structure are optional.

Field Name	Values	Default
Optimizer	<ul style="list-style-type: none"> <li>'bayesopt' — Use Bayesian optimization. Internally, this setting calls <code>bayesopt</code>.</li> <li>'gridsearch' — Use grid search with <code>NumGridDivisions</code> values per dimension.</li> <li>'randomsearch' — Search at random among <code>MaxObjectiveEvaluations</code> points.</li> </ul> <p>'gridsearch' searches in a random order, using uniform sampling without replacement from the grid. After optimization, you can get a table in grid order by using the command <code>sortrows(Mdl.ParameterOptimizationResults)</code>.</p>	'bayesopt'

Field Name	Values	Default
AcquisitionFunctionName	<ul style="list-style-type: none"> <li>• 'expected-improvement-per-second-plus'</li> <li>• 'expected-improvement'</li> <li>• 'expected-improvement-plus'</li> <li>• 'expected-improvement-per-second'</li> <li>• 'lower-confidence-bound'</li> <li>• 'probability-of-improvement'</li> </ul> <p>For details, see the bayesopt AcquisitionFunctionName name-value pair, or “Acquisition Function Types” on page 10-4.</p>	'expected-improvement-per-second-plus'
MaxObjectiveEvaluations	Maximum number of objective function evaluations.	30 for 'bayesopt' or 'randomsearch', and the entire grid for 'gridsearch'
MaxTime	Time limit, specified as a positive real. The time limit is in seconds, as measured by tic and toc. Run time can exceed MaxTime because MaxTime does not interrupt function evaluations.	Inf
NumGridDivisions	For 'gridsearch', the number of values in each dimension. Can be a vector of positive integers giving the number of values for each dimension, or a scalar that applies to all dimensions. Ignored for categorical variables.	10



Field Name	Values	Default
ShowPlots	Logical value indicating whether to show plots. If <code>true</code> , plots the best objective function value against iteration number. If there are one or two optimization parameters, and if <code>Optimizer</code> is <code>'bayesopt'</code> , then <code>ShowPlots</code> also plots a model of the objective function against the parameters.	<code>true</code>
SaveIntermediateResults	Logical value indicating whether to save results when <code>Optimizer</code> is <code>'bayesopt'</code> . If <code>true</code> , overwrites a workspace variable named <code>'BayesoptResults'</code> at each iteration. The variable is a <code>BayesianOptimization</code> object.	<code>false</code>
Verbose	Display to the command line. <ul style="list-style-type: none"> <li>• 0 — No iterative display</li> <li>• 1 — Iterative display</li> <li>• 2 — Iterative display with extra information</li> </ul> For details, see the <code>bayesopt Verbose</code> name-value pair.	1
UseParallel	Logical value indicating whether to run Bayesian optimization in parallel, which requires Parallel Computing Toolbox. For details, see “Parallel Bayesian Optimization” on page 10-9.	<code>false</code>

Field Name	Values	Default
Repartition	Logical value indicating whether to repartition the cross-validation at every iteration. If <code>false</code> , the optimizer uses a single partition for the optimization.  <code>true</code> usually gives the most robust results because this setting takes partitioning noise into account. However, for good results, <code>true</code> requires at least twice as many function evaluations.	<code>false</code>
Use no more than one of the following three field names.		
CVPartition	A <code>cvpartition</code> object, as created by <code>cvpartition</code>	<code>Kfold = 5</code>
Holdout	A scalar in the range $(0, 1)$ representing the holdout fraction.	
Kfold	An integer greater than 1.	

Example: `struct('MaxObjectiveEvaluations', 60)`

Data Types: `struct`

### Other

#### **PredictorNames** — Predictor variable names

cell array of unique character vectors

Predictor variable names, specified as the comma-separated pair consisting of 'PredictorNames' and a cell array of unique character vectors. The functionality of 'PredictorNames' depends on the way you supply the training data.

- If you supply `X` and `y`, then you can use 'PredictorNames' to give the predictor variables in `X` names.
  - The order of the names in `PredictorNames` must correspond to the column order of `X`. That is, `PredictorNames{1}` is the name of `X(:, 1)`, `PredictorNames{2}` is the name of `X(:, 2)`, and so on. Also, `size(X, 2)` and `numel(PredictorNames)` must be equal.
  - By default, `PredictorNames` is `{'x1', 'x2', ...}`.

- If you supply `tbl`, then you can use `'PredictorNames'` to choose which predictor variables to use in training. That is, `fitrgp` uses the predictor variables in `PredictorNames` and the response only in training.
  - `PredictorNames` must be a subset of `tbl.Properties.VariableNames` and cannot include the name of the response variable.
  - By default, `PredictorNames` contains the names of all predictor variables.
  - It good practice to specify the predictors for training using one of `'PredictorNames'` or `formula` only.

Example: `'PredictorNames', {'PedalLength', 'PedalWidth'}`

Data Types: `cell`

### **ResponseName — Response variable name**

`'Y'` (default) | character vector

Response variable name, specified as the comma-separated pair consisting of `'ResponseName'` and a character vector.

- If you supply `Y`, then you can use `'ResponseName'` to specify a name for the response variable.
- If you supply `ResponseVarName` or `formula`, then you cannot use `'ResponseName'`.

Example: `'ResponseName', 'response'`

Data Types: `char`

### **ResponseTransform — Response transformation**

`'none'` (default) | function handle

Response transformation, specified as the comma-separated pair consisting of `'ResponseTransform'` and either `'none'` or a function handle. The default is `'none'`, which means  $@(x)x$ , or no transformation. For a MATLAB function or a function you define, use its function handle. The function handle must accept a vector (the original response values) and return a vector of the same size (the transformed response values).

Example: `'ResponseTransform', 'none'`

Data Types: `char` | `function_handle`

**Verbose — Verbosity level**

0 (default) | 1

Verbosity level, specified as the comma-separated pair consisting of 'Verbose' and one of the following.

- 0 — `fitrgp` suppresses diagnostic messages related to active set selection and block coordinate descent but displays the messages related to parameter estimation, depending on the value of 'Display' in `OptimizerOptions`.
- 1 — `fitrgp` displays the iterative diagnostic messages related to parameter estimation, active set selection, and block coordinate descent.

Example: 'Verbose',1

**CacheSize — Cache size in megabytes**

1000 MB (default) | positive scalar

Cache size in megabytes (MB), specified as the comma-separated pair consisting of 'CacheSize' and a positive scalar. Cache size is the extra memory that is available in addition to that required for fitting and active set selection. `fitrgp` uses `CacheSize` to:

- Decide whether interpoint distances should be cached when estimating parameters.
- Decide how matrix vector products should be computed for block coordinate descent method and for making predictions.

Example: 'CacheSize',2000

Data Types: double

## Output Arguments

**gprMdl — Gaussian process regression model**

RegressionGP object | RegressionPartitionedModel object

Gaussian process regression model, returned as a `RegressionGP` or a `RegressionPartitionedModel` object.

- If you cross-validate, that is, if you use one of the 'Crossval', 'KFold', 'Holdout', 'Leaveout', or 'CVPartition' name-value pairs, then `gprMdl` is a `RegressionPartitionedModel` object. You cannot use a

RegressionPartitionedModel object to make predictions using predict. For more information on the methods and properties of this object, see RegressionPartitionedModel.

- If you do not cross-validate, then gprMdl is a RegressionGP object. You can use this object for predictions using the predict method. For more information on the methods and properties of this object, see RegressionGP.

## Definitions

### Active Set Selection and Parameter Estimation

For subset of data, subset of regressors, or fully independent conditional approximation fitting methods (FitMethod equal to 'sd', 'sr', or 'fic'), if you do not provide the active set, fitrgp selects the active set and computes the parameter estimates in a series of iterations.

In the first iteration, the software uses the initial parameter values in vector  $\eta_0 = [\beta_0, \sigma_0, \theta_0]$  to select an active set  $A_1$ . It maximizes the GPR marginal log likelihood or its approximation using  $\eta_0$  as the initial values and  $A_1$  to compute the new parameter estimates  $\eta_1$ . Next, it computes the new log likelihood  $L_1$  using  $\eta_1$  and  $A_1$ .

In the second iteration, the software selects the active set  $A_2$  using the parameter values in  $\eta_1$ . Then, using  $\eta_1$  as the initial values and  $A_2$ , it maximizes the GPR marginal log likelihood or its approximation and estimates the new parameter values  $\eta_2$ . Then using  $\eta_2$  and  $A_2$ , computes the new log likelihood value  $L_2$ .

The following table summarizes the iterations and what is computed at each iteration.

Iteration Number	Active Set	Parameter Vector	Log Likelihood
1	$A_1$	$\eta_1$	$L_1$
2	$A_2$	$\eta_2$	$L_2$
3	$A_3$	$\eta_3$	$L_3$
...	...	...	...

The software iterates similarly for a specified number of repetitions. You can specify the number of replications for active set selection using the NumActiveSetRepeats name-value pair argument.

## Tips

- `fitrgp` accepts any combination of fitting, prediction, and active set selection methods. In some cases it might not be possible to compute the standard deviations of the predicted responses, hence the prediction intervals. See `predict`. And in some cases, using the exact method might be expensive due to the size of the training data.
- The `PredictorNames` property stores one element for each of the original predictor variable names. For example, if there are three predictors, one of which is a categorical variable with three levels, `PredictorNames` is a 1-by-3 cell array of character vectors.
- The `ExpandedPredictorNames` property stores one element for each of the predictor variables, including the dummy variables. For example, if there are three predictors, one of which is a categorical variable with three levels, then `ExpandedPredictorNames` is a 1-by-5 cell array of character vectors.
- Similarly, the `Beta` property stores one beta coefficient for each predictor, including the dummy variables.
- The `X` property stores the training data as originally input. It does not include the dummy variables.
- The default approach to initializing the Hessian approximation in `fitrgp` can be slow when you have a GPR model with many kernel parameters, such as when using an ARD kernel with many predictors. In this case, consider specifying `'auto'` or a value for the initial step size.

You can set `'Verbose', 1` for display of iterative diagnostic messages, and begin training a GPR model using an LBFGS or quasi-Newton optimizer with the default `fitrgp` optimization. If the iterative diagnostic messages are not displayed after a few seconds, it is possible that initialization of the Hessian approximation is taking too long. In this case, consider restarting training and using the initial step size to speed up optimization.

- After training a `RegressionGP` model object by using `fitrgp`, use the `predict` function and the trained model object to generate C code that predicts responses for new data. For details, see “Code Generation” on page 32-0 .

## Algorithms

- Fitting a GPR model involves estimating the following model parameters from the data:

- Covariance function  $k(\mathbf{x}_i, \mathbf{x}_j | \boldsymbol{\theta})$  parameterized in terms of kernel parameters in vector  $\boldsymbol{\theta}$  (see “Kernel (Covariance) Function Options” on page 6-6)
- Noise variance,  $\sigma^2$
- Coefficient vector of fixed basis functions,  $\boldsymbol{\beta}$

The value of the 'KernelParameters' name-value pair argument is a vector that consists of initial values for the signal standard deviation  $\sigma_f$  and the characteristic length scales  $\sigma_l$ . The `fitrgp` function uses these values to determine the kernel parameters. Similarly, the 'Sigma' name-value pair argument contains the initial value for the noise standard deviation  $\sigma$ .

- During optimization, `fitrgp` creates a vector of unconstrained initial parameter values  $\boldsymbol{\eta}_0$  by using the initial values for the noise standard deviation and the kernel parameters.
- `fitrgp` analytically determines the explicit basis coefficients  $\boldsymbol{\beta}$ , specified by the 'Beta' name-value pair argument, from estimated values of  $\boldsymbol{\theta}$  and  $\sigma^2$ . Therefore,  $\boldsymbol{\beta}$  does not appear in the  $\boldsymbol{\eta}_0$  vector when `fitrgp` initializes numerical optimization.

---

**Note** If you specify no estimation of parameters for the GPR model, `fitrgp` uses the value of the 'Beta' name-value pair argument and other initial parameter values as the known GPR parameter values (see `Beta`). In all other cases, the value of the 'Beta' argument is optimized analytically from the objective function.

---

- The quasi-Newton optimizer uses a trust-region method with a dense, symmetric rank-1-based (SR1), quasi-Newton approximation to the Hessian, while the LBFGS optimizer uses a standard line-search method with a limited-memory Broyden-Fletcher-Goldfarb-Shanno (LBFGS) quasi-Newton approximation to the Hessian. See Nocedal and Wright [6].
- If you set the 'InitialStepSize' name-value pair argument to 'auto', `fitrgp` determines the initial step size,  $\|s_0\|_\infty$ , by using  $\|s_0\|_\infty = 0.5\|\boldsymbol{\eta}_0\|_\infty + 0.1$ .

$s_0$  is the initial step vector, and  $\eta_0$  is the vector of unconstrained initial parameter values.

• During optimization, `fitrpg` uses the initial step size,  $\|s_0\|_\infty$ , as follows:

If you use 'Optimizer', 'quasinevton' with the initial step size, then the initial

Hessian approximation is  $\frac{\|g_0\|_\infty}{\|s_0\|_\infty} I$ .

If you use 'Optimizer', 'lbfgs' with the initial step size, then the initial inverse-

Hessian approximation is  $\frac{\|s_0\|_\infty}{\|g_0\|_\infty} I$ .

$g_0$  is the initial gradient vector, and  $I$  is the identity matrix.

## References

- [1] Warwick J. N., T. L. Sellers, S. R. Talbot, A. J. Cawthorn, and W. B. Ford. "The Population Biology of Abalone (*Haliotis* species) in Tasmania. I. Blacklip Abalone (*H. rubra*) from the North Coast and Islands of Bass Strait." Sea Fisheries Division, Technical Report No. 48 (ISSN 1034-3288), 1994.
- [2] S. Waugh. "Extending and Benchmarking Cascade-Correlation", PhD Thesis. Computer Science Department, University of Tasmania, 1995.
- [3] Lichman, M. UCI Machine Learning Repository, Irvine, CA: University of California, School of Information and Computer Science, 2013. <http://archive.ics.uci.edu/ml>.
- [4] Rasmussen, C. E. and C. K. I. Williams. *Gaussian Processes for Machine Learning*. MIT Press. Cambridge, Massachusetts, 2006.
- [5] Lagarias, J. C., J. A. Reeds, M. H. Wright, and P. E. Wright. "Convergence Properties of the Nelder-Mead Simplex Method in Low Dimensions." *SIAM Journal of Optimization*. Vol. 9, Number 1, 1998, pp. 112–147.



[6] Nocedal, J. and S. J. Wright. *Numerical Optimization*, Second Edition. Springer Series in Operations Research, Springer Verlag, 2006.

## See Also

RegressionGP | compact | predict

## Topics

“Gaussian Process Regression Models” on page 6-2

“Kernel (Covariance) Function Options” on page 6-6

**Introduced in R2015b**

## fitrlinear

Fit linear regression model to high-dimensional data

`fitrlinear` efficiently trains linear regression models with high-dimensional, full or sparse predictor data. Available linear regression models include regularized support vector machines (SVM) and least-squares regression methods. `fitrlinear` minimizes the objective function using techniques that reduce computing time (e.g., stochastic gradient descent).

A high-dimensional data set includes many predictor variables. Although such a data set can consume a significant fraction of memory, it must fit in the MATLAB Workspace. For low- through medium-dimensional predictor data sets, see “Alternatives for Lower-Dimensional Data” on page 22-2.

## Syntax

```
Mdl = fitrlinear(X,Y)
Mdl = fitrlinear(X,Y,Name,Value)
[Mdl,FitInfo] = fitrlinear(____)
[Mdl,FitInfo,HyperparameterOptimizationResults] = fitrlinear(____)
```

## Description

`Mdl = fitrlinear(X,Y)` returns a trained regression model object that contains the results of fitting a support vector machine regression model to the predictors `X` and response `Y`.

`Mdl = fitrlinear(X,Y,Name,Value)` returns a trained linear regression model with additional options specified by one or more `Name, Value` pair arguments. For example, you can specify implement least-squares regression, specify to cross-validate, or specify the type of regularization. It is good practice to cross-validate using the `Kfold` `Name, Value` pair argument. The cross-validation results determine how well the model generalizes.

`[Mdl,FitInfo] = fitrlinear(____)` also returns optimization details using any of the previous syntaxes. You cannot request `FitInfo` for cross-validated models.

[Mdl,FitInfo,HyperparameterOptimizationResults] = fitrlinear(\_\_\_\_) also returns hyperparameter optimization details when you pass an OptimizeHyperparameters name-value pair.

## Examples

### Train Linear Regression Model

Train a linear regression model using SVM, dual SGD, and ridge regularization.

Simulate 10000 observations from this model

$$y = x_{100} + 2x_{200} + e.$$

- $X = x_1, \dots, x_{1000}$  is a 10000-by-1000 sparse matrix with 10% nonzero standard normal elements.
- $e$  is random normal error with mean 0 and standard deviation 0.3.

```
rng(1) % For reproducibility
n = 1e4;
d = 1e3;
nz = 0.1;
X = sprandn(n,d,nz);
Y = X(:,100) + 2*X(:,200) + 0.3*randn(n,1);
```

Train a linear regression model. By default, fitrlinear uses support vector machines with a ridge penalty, and optimizes using dual SGD for SVM. Determine how well the optimization algorithm fit the model to the data by extracting a fit summary.

```
[Mdl,FitInfo] = fitrlinear(X,Y)
```

```
Mdl =
```

```
RegressionLinear
  ResponseName: 'Y'
  ResponseTransform: 'none'
           Beta: [1000x1 double]
           Bias: -0.0056
           Lambda: 1.0000e-04
           Learner: 'svm'
```

```
FitInfo =  
  
    struct with fields:  
  
        Lambda: 1.0000e-04  
        Objective: 0.2726  
        PassLimit: 10  
        NumPasses: 10  
        BatchLimit: []  
        NumIterations: 100000  
        GradientNorm: NaN  
        GradientTolerance: 0  
        RelativeChangeInBeta: 0.4908  
        BetaTolerance: 1.0000e-04  
        DeltaGradient: 1.5818  
        DeltaGradientTolerance: 0.1000  
        TerminationCode: 0  
        TerminationStatus: {'Iteration limit exceeded.'}  
        Alpha: [10000x1 double]  
        History: []  
        FitTime: 0.2463  
        Solver: {'dual'}
```

`Mdl` is a `RegressionLinear` model. You can pass `Mdl` and the training or new data to `loss` to inspect the in-sample mean-squared error. Or, you can pass `Mdl` and new predictor data to `predict` to predict responses for new observations.

`FitInfo` is a structure array containing, among other things, the termination status (`TerminationStatus`) and how long the solver took to fit the model to the data (`FitTime`). It is good practice to use `FitInfo` to determine whether optimization-termination measurements are satisfactory. In this case, `fitrlinear` reached the maximum number of iterations. Because training time is fast, you can retrain the model, but increase the number of passes through the data. Or, try another solver, such as `LBFGS`.

## Find Good Lasso Penalty Using Cross-Validation

To determine a good lasso-penalty strength for a linear regression model that uses least squares, implement 5-fold cross-validation.

Simulate 10000 observations from this model

$$y = x_{100} + 2x_{200} + e.$$

- $X = \{x_1, \dots, x_{1000}\}$  is a 10000-by-1000 sparse matrix with 10% nonzero standard normal elements.
- $e$  is random normal error with mean 0 and standard deviation 0.3.

```
rng(1) % For reproducibility
n = 1e4;
d = 1e3;
nz = 0.1;
X = sprandn(n,d,nz);
Y = X(:,100) + 2*X(:,200) + 0.3*randn(n,1);
```

Create a set of 15 logarithmically-spaced regularization strengths from  $10^{-5}$  through  $10^{-1}$ .

```
Lambda = logspace(-5,-1,15);
```

Cross-validate the models. To increase execution speed, transpose the predictor data and specify that the observations are in columns. Optimize the objective function using SpARSA.

```
X = X';
CVMdl = fitrlinear(X,Y,'ObservationsIn','columns','KFold',5,'Lambda',Lambda,...
    'Learner','leastsquares','Solver','sparsa','Regularization','lasso');

numCLModels = numel(CVMdl.Trained)
```

```
numCLModels =
```

```
5
```

CVMdl is a `RegressionPartitionedLinear` model. Because `fitrlinear` implements 5-fold cross-validation, CVMdl contains 5 `RegressionLinear` models that the software trains on each fold.

Display the first trained linear regression model.

```
Mdl1 = CVMdl.Trained{1}

Mdl1 =

  RegressionLinear
    ResponseName: 'Y'
    ResponseTransform: 'none'
        Beta: [1000x15 double]
        Bias: [1x15 double]
        Lambda: [1x15 double]
    Learner: 'leastsquares'
```

Mdl1 is a `RegressionLinear` model object. `fitrlinear` constructed Mdl1 by training on the first four folds. Because `Lambda` is a sequence of regularization strengths, you can think of Mdl1 as 11 models, one for each regularization strength in `Lambda`.

Estimate the cross-validated MSE.

```
mse = kfoldLoss(CVMdl);
```

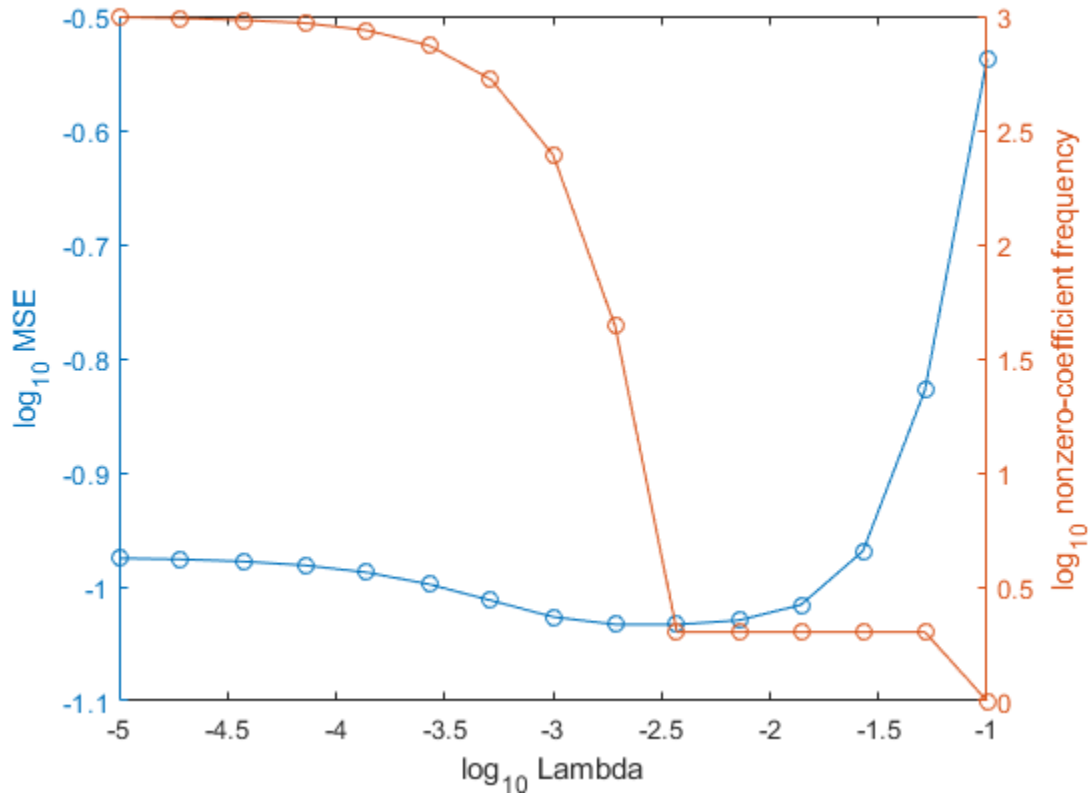
Higher values of `Lambda` lead to predictor variable sparsity, which is a good quality of a regression model. For each regularization strength, train a linear regression model using the entire data set and the same options as when you cross-validated the models. Determine the number of nonzero coefficients per model.

```
Mdl = fitrlinear(X,Y,'ObservationsIn','columns','Lambda',Lambda,...
    'Learner','leastsquares','Solver','sparsa','Regularization','lasso');
numNZCoeff = sum(Mdl.Beta~=0);
```

In the same figure, plot the cross-validated MSE and frequency of nonzero coefficients for each regularization strength. Plot all variables on the log scale.

```
figure;
[h,hL1,hL2] = plotyy(log10(Lambda),log10(mse),...
    log10(Lambda),log10(numNZCoeff));
hL1.Marker = 'o';
hL2.Marker = 'o';
ylabel(h(1),'log_{10} MSE')
ylabel(h(2),'log_{10} nonzero-coefficient frequency')
```

```
xlabel('log_{10} Lambda')
hold off
```



Choose the index of the regularization strength that balances predictor variable sparsity and low MSE (for example, `Lambda(10)`).

```
idxFinal = 10;
```

Extract the model with corresponding to the minimal MSE.

```
MdlFinal = selectModels(Mdl,idxFinal)
idxNZCoeff = find(MdlFinal.Beta~=0)
EstCoeff = Mdl.Beta(idxNZCoeff)
```

```
MdlFinal =  
  
    RegressionLinear  
        ResponseName: 'Y'  
        ResponseTransform: 'none'  
            Beta: [1000x1 double]  
            Bias: -0.0050  
            Lambda: 0.0037  
            Learner: 'leastsquares'  
  
idxNZCoeff =  
  
    100  
    200  
  
EstCoeff =  
  
    1.0051  
    1.9965
```

`MdlFinal` is a `RegressionLinear` model with one regularization strength. The nonzero coefficients `EstCoeff` are close to the coefficients that simulated the data.

### Optimize a Linear Regression

This example shows how to optimize hyperparameters automatically using `fitrlinear`. The example uses artificial (simulated) data for the model

$$y = x_{100} + 2x_{200} + e.$$

- $X = \{x_1, \dots, x_{1000}\}$  is a 10000-by-1000 sparse matrix with 10% nonzero standard normal elements.
- $e$  is random normal error with mean 0 and standard deviation 0.3.

```
rng(1) % For reproducibility  
n = 1e4;  
d = 1e3;
```



```

nz = 0.1;
X = sprandn(n,d,nz);
Y = X(:,100) + 2*X(:,200) + 0.3*randn(n,1);

```

Find hyperparameters that minimize five-fold cross validation loss by using automatic hyperparameter optimization.

For reproducibility, use the 'expected-improvement-plus' acquisition function.

```

hyperopts = struct('AcquisitionFunctionName','expected-improvement-plus');
[Mdl,FitInfo,HyperparameterOptimizationResults] = fitrlinear(X,Y,...
    'OptimizeHyperparameters','auto',...
    'HyperparameterOptimizationOptions',hyperopts)

```

```

=====
| Iter | Eval | Objective | Objective | BestSoFar | BestSoFar | Lambda
|      | result |           | runtime   | (observed) | (estim.)  |
=====
|  1  | Best | 0.16027 | 1.1822 | 0.16027 | 0.16027 | 2.4206e-09
|  2  | Best | 0.1467  | 1.0017 | 0.1467  | 0.14762 | 0.001807
|  3  | Best | 0.13388 | 0.77576 | 0.13388 | 0.14695 | 2.4681e-09
|  4  | Best | 0.1146  | 0.75519 | 0.1146  | 0.11461 | 0.021027
|  5  | Accept | 0.45011 | 0.6935 | 0.1146  | 0.11512 | 9.9943
|  6  | Best | 0.11004 | 0.71275 | 0.11004 | 0.11004 | 0.0095096
|  7  | Accept | 0.13287 | 0.63877 | 0.11004 | 0.11003 | 1.241e-07
|  8  | Accept | 0.13425 | 0.70596 | 0.11004 | 0.11002 | 3.5916e-05
|  9  | Accept | 0.15915 | 0.80448 | 0.11004 | 0.11002 | 5.0825e-06
| 10  | Accept | 0.41072 | 0.8382 | 0.11004 | 0.11002 | 0.54825
| 11  | Accept | 0.1603  | 0.99317 | 0.11004 | 0.11001 | 0.00013759
| 12  | Accept | 0.16309 | 0.78252 | 0.11004 | 0.11    | 9.8154e-08
| 13  | Accept | 0.13105 | 0.66706 | 0.11004 | 0.10999 | 0.00062585
| 14  | Accept | 0.13568 | 0.65818 | 0.11004 | 0.10999 | 1.9655e-06
| 15  | Accept | 0.13503 | 0.61507 | 0.11004 | 0.10999 | 1.4466e-08
| 16  | Accept | 0.11027 | 0.63254 | 0.11004 | 0.11011 | 0.008924
| 17  | Best | 0.10929 | 0.65466 | 0.10929 | 0.10984 | 0.0098136
| 18  | Accept | 0.11084 | 0.69351 | 0.10929 | 0.11008 | 0.0099479
| 19  | Accept | 0.48019 | 0.63683 | 0.10929 | 0.11008 | 9.9966
| 20  | Accept | 0.11044 | 0.70531 | 0.10929 | 0.11014 | 0.009283
=====
| Iter | Eval | Objective | Objective | BestSoFar | BestSoFar | Lambda
|      | result |           | runtime   | (observed) | (estim.)  |
=====
|  21  | Accept | 0.36724 | 0.65381 | 0.10929 | 0.1099  | 0.54138
|  22  | Accept | 0.11117 | 0.72045 | 0.10929 | 0.11017 | 0.012665
|  23  | Accept | 0.11681 | 0.73487 | 0.10929 | 0.11016 | 0.0037864

```

	24		Accept		0.13444		0.6767		0.10929		0.11016		1.0006e-09
	25		Accept		0.10974		0.66828		0.10929		0.11011		0.011669
	26		Best		0.10926		0.8624		0.10926		0.11014		0.026669
	27		Best		0.1065		0.94742		0.1065		0.10641		0.011227
	28		Accept		0.10721		0.98398		0.1065		0.1063		0.016855
	29		Accept		0.15392		0.87125		0.1065		0.10627		1.3979e-08
	30		Accept		0.10669		0.92861		0.1065		0.10644		0.015752

---

Optimization completed.  
 MaxObjectiveEvaluations of 30 reached.  
 Total function evaluations: 30  
 Total elapsed time: 74.1489 seconds.  
 Total objective function evaluation time: 23.1951

Best observed feasible point:

Lambda	Learner
0.011227	svm

Observed objective function value = 0.1065  
 Estimated objective function value = 0.10644  
 Function evaluation time = 0.94742

Best estimated feasible point (according to models):

Lambda	Learner
0.015752	svm

Estimated objective function value = 0.10644  
 Estimated function evaluation time = 0.93584

Mdl =

```

RegressionLinear
  ResponseName: 'Y'
  ResponseTransform: 'none'
    Beta: [1000x1 double]
    Bias: -8.8665e-04
    Lambda: 0.0158
    Learner: 'svm'
  
```

```
FitInfo =
```

```
struct with fields:
```

```
        Lambda: 0.0158
        Objective: 0.2308
        PassLimit: 10
        NumPasses: 10
        BatchLimit: []
        NumIterations: 99995
        GradientNorm: NaN
        GradientTolerance: 0
        RelativeChangeInBeta: 0.0645
        BetaTolerance: 1.0000e-04
        DeltaGradient: 1.1319
        DeltaGradientTolerance: 0.1000
        TerminationCode: 0
        TerminationStatus: {'Iteration limit exceeded.'}
        Alpha: [10000x1 double]
        History: []
        FitTime: 0.1503
        Solver: {'dual'}
```

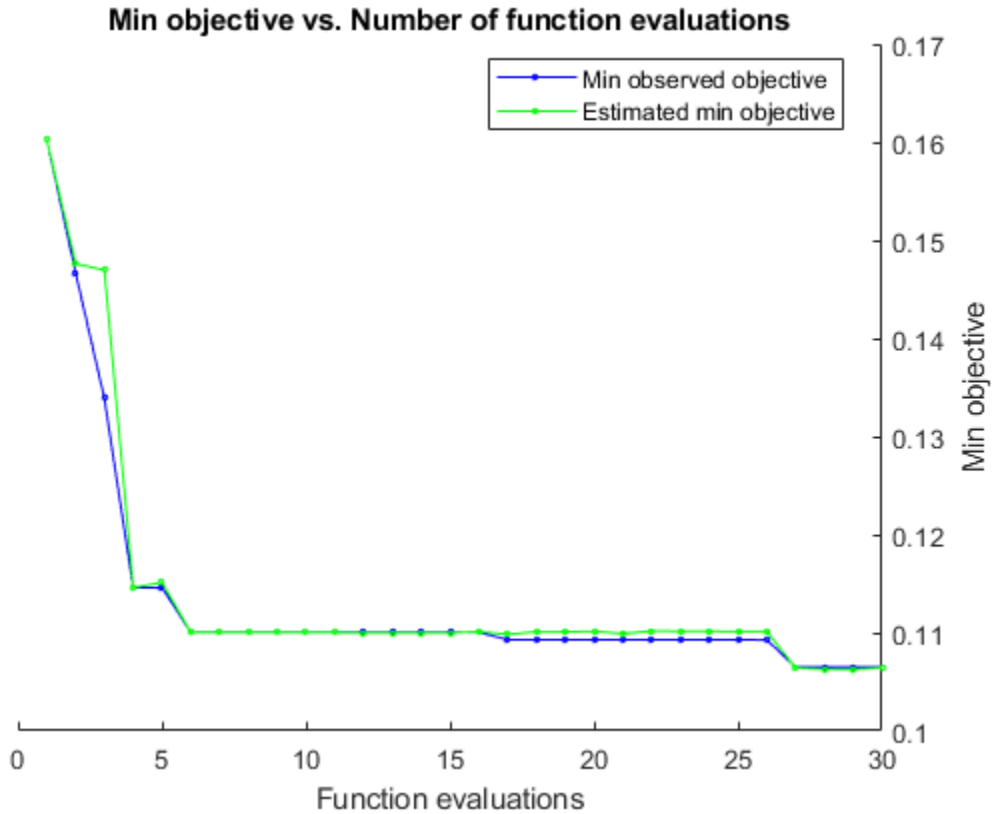
```
HyperparameterOptimizationResults =
```

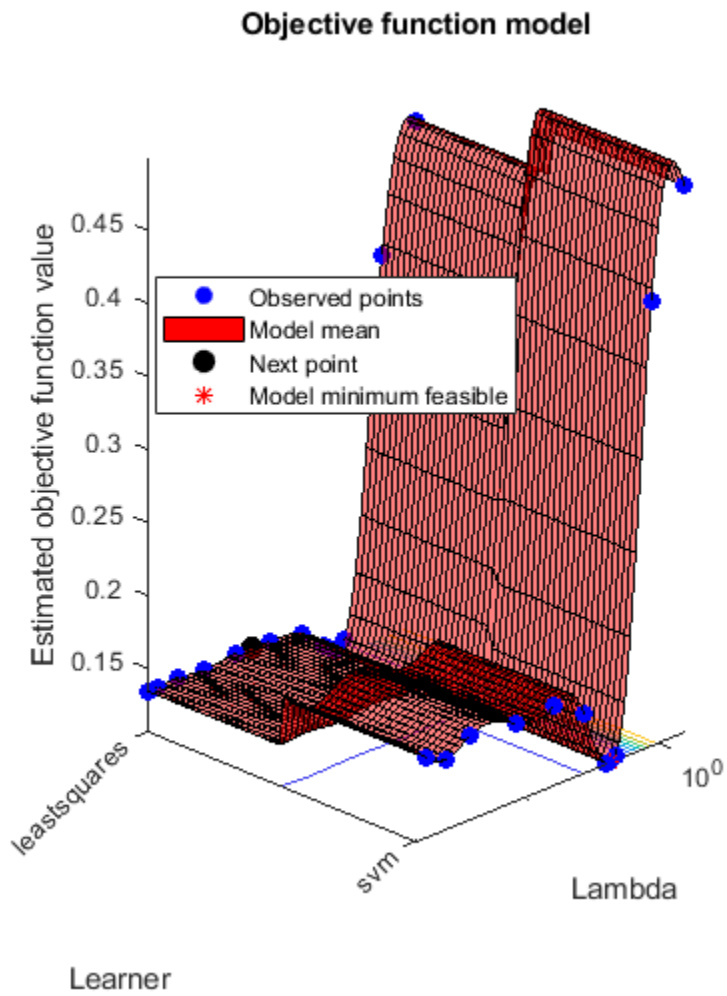
```
BayesianOptimization with properties:
```

```
        ObjectiveFcn: @createObjFcn/theObjFcn
        VariableDescriptions: [3x1 optimizableVariable]
        Options: [1x1 struct]
        MinObjective: 0.1065
        XAtMinObjective: [1x2 table]
        MinEstimatedObjective: 0.1064
        XAtMinEstimatedObjective: [1x2 table]
        NumObjectiveEvaluations: 30
        TotalElapsedTime: 74.1489
        NextPoint: [1x2 table]
        XTrace: [30x2 table]
        ObjectiveTrace: [30x1 double]
        ConstraintsTrace: []
```

```

        UserDataTrace: {30x1 cell}
    ObjectiveEvaluationTimeTrace: [30x1 double]
        IterationTimeTrace: [30x1 double]
            ErrorTrace: [30x1 double]
        FeasibilityTrace: [30x1 logical]
    FeasibilityProbabilityTrace: [30x1 double]
        IndexOfMinimumTrace: [30x1 double]
        ObjectiveMinimumTrace: [30x1 double]
    EstimatedObjectiveMinimumTrace: [30x1 double]
    
```





This optimization technique is simpler than that shown in “Find Good Lasso Penalty Using Cross-Validation” on page 32-2382, but does not allow you to trade off model complexity and cross-validation loss.

## Input Arguments

### **x** — Predictor data

full matrix | sparse matrix

Predictor data, specified as an  $n$ -by- $p$  full or sparse matrix.

The length of  $Y$  and the number of observations in  $X$  must be equal.

---

**Note** If you orient your predictor matrix so that observations correspond to columns and specify 'ObservationsIn', 'columns', then you might experience a significant reduction in optimization-execution time.

---

Data Types: single | double

### **y** — Response data

numeric vector

Response data, specified as an  $n$ -dimensional numeric vector. The length of  $Y$  and the number of observations in  $X$  must be equal.

Data Types: single | double

---

**Note** `fitrlinear` removes missing observations, that is, observations with any of these characteristics:

- NaN elements in the response ( $Y$  or `ValidationData{2}`)
- At least one NaN value in a predictor observation (row in  $X$  or `ValidationData{1}`)
- NaN value or 0 weight (`Weights` or `ValidationData{3}`)

For memory-usage economy, it is best practice to remove observations containing missing values from your training data manually before training.

---

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

---

**Note** You cannot use any cross-validation name-value pair along with `OptimizeHyperparameters`. You can modify the cross-validation for `OptimizeHyperparameters` only by using the `HyperparameterOptimizationOptions` name-value pair.

---

Example:

`'Learner', 'regression', 'CrossVal', 'on', 'Regularization', 'lasso'` specifies to implement least-squares regression, implement 10-fold cross-validation, and specifies to include a lasso regularization term.

### Linear Regression Options

#### **Epsilon** — Half the width of epsilon-insensitive band

`iqr(Y)/1.349` (default) | nonnegative scalar value

Half the width of the epsilon-insensitive band, specified as the comma-separated pair consisting of `'Epsilon'` and a nonnegative scalar value. `'Epsilon'` applies to SVM learners only.

The default `Epsilon` value is `iqr(Y)/1.349`, which is an estimate of standard deviation using the interquartile range of the response variable `Y`. If `iqr(Y)` is equal to zero, then the default `Epsilon` value is 0.1.

Example: `'Epsilon', 0.3`

Data Types: `single` | `double`

#### **Lambda** — Regularization term strength

`'auto'` (default) | nonnegative scalar | vector of nonnegative values

Regularization term strength, specified as the comma-separated pair consisting of `'Lambda'` and `'auto'`, a nonnegative scalar, or a vector of nonnegative values.

- For 'auto',  $\text{Lambda} = 1/n$ .
  - If you specify a cross-validation, name-value pair argument (e.g., `CrossVal`), then  $n$  is the number of in-fold observations.
  - Otherwise,  $n$  is the training sample size.
- For a vector of nonnegative values, the software sequentially optimizes the objective function for each distinct value in `Lambda` in ascending order.
  - If `Solver` is 'sgd' or 'asgd' and `Regularization` is 'lasso', then the software does not use the previous coefficient estimates as a warm start on page 32-2043 for the next optimization iteration. Otherwise, the software uses warm starts.
  - If `Regularization` is 'lasso', then any coefficient estimate of 0 retains its value when the software optimizes using subsequent values in `Lambda`.

Returns coefficient estimates for all optimization iterations.

Example: `'Lambda', 10.^(-(10:-2:2))`

Data Types: char | double | single

### Learner — Linear regression model type

'svm' (default) | 'leastsquares'

Linear regression model type, specified as the comma-separated pair consisting of 'Learner' and 'svm' or 'leastsquares'.

In this table,  $f(x) = x\beta + b$ .

- $\beta$  is a vector of  $p$  coefficients.
- $x$  is an observation from  $p$  predictor variables.
- $b$  is the scalar bias.

Value	Algorithm	Response range	Loss function
'leastsquares'	Linear regression via ordinary least squares	$y \in (-\infty, \infty)$	Mean squared error (MSE): $\ell[y, f(x)] = \frac{1}{2}[y - f(x)]^2$



Value	Algorithm	Response range	Loss function
'svm'	Support vector machine regression	Same as 'leastquares'	Epsilon-insensitive: $\ell[y, f(x)] = \max[0,  y - f(x)  - \epsilon]$

Example: 'Learner', 'leastquares'

Data Types: char

#### ObservationsIn — Predictor data observation dimension

'rows' (default) | 'columns'

Predictor data observation dimension, specified as the comma-separated pair consisting of 'ObservationsIn' and 'columns' or 'rows'.

---

**Note** If you orient your predictor matrix so that observations correspond to columns and specify 'ObservationsIn', 'columns', then you might experience a significant reduction in optimization-execution time.

---

Data Types: char

#### Regularization — Complexity penalty type

'lasso' | 'ridge'

Complexity penalty type, specified as the comma-separated pair consisting of 'Regularization' and 'lasso' or 'ridge'.

The software composes the objective function for minimization from the sum of the average loss function (see Learner) and the regularization term in this table.

Value	Description
'lasso'	Lasso (L1) penalty: $\lambda \sum_{j=1}^p  \beta_j $
'ridge'	Ridge (L2) penalty: $\frac{\lambda}{2} \sum_{j=1}^p \beta_j^2$

To specify the regularization term strength, which is  $\lambda$  in the expressions, use Lambda.

The software excludes the bias term ( $\beta_0$ ) from the regularization penalty.

If `Solver` is 'sparsa', then the default value of `Regularization` is 'lasso'. Otherwise, the default is 'ridge'.

---

### Tip

- For predictor variable selection, specify 'lasso'.
  - For optimization accuracy, specify 'ridge'.
- 

Example: 'Regularization', 'lasso'

Data Types: char

### **solver** — Objective function minimization technique

'sgd' | 'asgd' | 'dual' | 'bfgs' | 'lbfgs' | 'sparsa' | cell vector of character vectors

Objective function minimization technique, specified as the comma-separated pair consisting of 'Solver' and a value or cell vector of values in this table.

Value	Description	Restrictions
'sgd'	Stochastic gradient descent (SGD) [5][3]	
'asgd'	Average stochastic gradient descent (ASGD) [8]	
'dual'	Dual SGD for SVM [2][7]	Regularization must be 'ridge' and Learner must be 'svm'.
'bfgs'	Broyden-Fletcher-Goldfarb-Shanno quasi-Newton algorithm (BFGS) [4]	Inefficient if X is very high-dimensional.
'lbfgs'	Limited-memory BFGS (LBFGS) [4]	Regularization must be 'ridge'.

Value	Description	Restrictions
'sparsa'	Sparse Reconstruction by Separable Approximation (SpaRSA) [6]	Regularization must be 'lasso'.

If you specify:

- A ridge penalty (see Regularization) and  $\text{size}(X,1) \leq 100$  (100 or fewer predictor variables), then the default solver is 'bfgs'.
- An SVM regression model (see Learner), a ridge penalty, and  $\text{size}(X,1) > 100$  (more than 100 predictor variables), then the default solver is 'dual'.
- A lasso penalty and  $X$  contains 100 or fewer predictor variables, then the default solver is 'sparsa'.

Otherwise, the default solver is 'sgd'.

If you specify a cell vector of solver names, then the software uses all solvers in the specified order for each `Lambda`.

For more details on which solver to choose, see “Tips” on page 32-2420.

Example: 'Solver', {'sgd', 'lbfgs'}

Data Types: char | cell

### Beta — Initial linear coefficient estimates

`zeros(p,1)` (default) | numeric vector | numeric matrix

Initial linear coefficient estimates ( $\beta$ ), specified as the comma-separated pair consisting of 'Beta' and a  $p$ -dimensional numeric vector or a  $p$ -by- $L$  numeric matrix.  $p$  is the number of predictor variables in  $X$  and  $L$  is the number of regularization-strength values (for more details, see `Lambda`).

- If you specify a  $p$ -dimensional vector, then the software optimizes the objective function  $L$  times using this process.
  - 1 The software optimizes using `Beta` as the initial value and the minimum value of `Lambda` as the regularization strength.
  - 2 The software optimizes again using the resulting estimate from the previous optimization as a warm start on page 32-2043, and the next smallest value in `Lambda` as the regularization strength.

**3** The software implements step 2 until it exhausts all values in `Lambda`.

- If you specify a  $p$ -by- $L$  matrix, then the software optimizes the objective function  $L$  times. At iteration  $j$ , the software uses `Beta(:, j)` as the initial value and, after it sorts `Lambda` in ascending order, uses `Lambda(j)` as the regularization strength.

If you set `'Solver', 'dual'`, then the software ignores `Beta`.

Data Types: `single` | `double`

### **Bias** — Initial intercept estimate

numeric scalar | numeric vector

Initial intercept estimate ( $b$ ), specified as the comma-separated pair consisting of `'Bias'` and a numeric scalar or an  $L$ -dimensional numeric vector.  $L$  is the number of regularization-strength values (for more details, see `Lambda`).

- If you specify a scalar, then the software optimizes the objective function  $L$  times using this process.
  - 1** The software optimizes using `Bias` as the initial value and the minimum value of `Lambda` as the regularization strength.
  - 2** The uses the resulting estimate as a warm start on page 32-2420 to the next optimization iteration, and uses the next smallest value in `Lambda` as the regularization strength.
  - 3** The software implements step 2 until it exhausts all values in `Lambda`.
- If you specify an  $L$ -dimensional vector, then the software optimizes the objective function  $L$  times. At iteration  $j$ , the software uses `Bias(j)` as the initial value and, after it sorts `Lambda` in ascending order, uses `Lambda(j)` as the regularization strength.
- By default:
  - If `Learner` is `'leastsquares'`, then `Bias` is the weighted average of  $Y$  for training or, for cross-validation, in-fold responses.
  - If `Learner` is `'svm'`, then `Bias` is the weighted median of  $Y$  for all training or, for cross-validation, in-fold observations that are greater than `Epsilon`.

Data Types: `single` | `double`

### **FitBias** — Linear model intercept inclusion flag

`true` (default) | `false`

Linear model intercept inclusion flag, specified as the comma-separated pair consisting of 'FitBias' and true or false.

Value	Description
true	The software includes the bias term $b$ in the linear model, and then estimates it.
false	The software sets $b = 0$ during estimation.

Example: 'FitBias', false

Data Types: logical

### PostFitBias — Flag to fit linear model intercept after optimization

false (default) | true

Flag to fit the linear model intercept after optimization, specified as the comma-separated pair consisting of 'PostFitBias' and true or false.

Value	Description
false	The software estimates the bias term $b$ and the coefficients $\beta$ during optimization.
true	To estimate $b$ , the software: <ol style="list-style-type: none"> <li>1 Estimates <math>\beta</math> and <math>b</math> using the model.</li> <li>2 Computes residuals.</li> <li>3 Refits <math>b</math>. For least squares, <math>b</math> is the weighted average of the residuals. For SVM regression, <math>b</math> is the weighted median between all residuals with magnitude greater than Epsilon.</li> </ol>

If you specify true, then FitBias must be true.

Example: 'PostFitBias', true

Data Types: logical

### verbose — Verbosity level

0 (default) | nonnegative integer

Verbosity level, specified as the comma-separated pair consisting of 'Verbose' and a nonnegative integer. Verbose controls the amount of diagnostic information `fitrlinear` displays at the command line.

Value	Description
0	<code>fitrlinear</code> does not display diagnostic information.
1	<code>fitrlinear</code> periodically displays and stores the value of the objective function, gradient magnitude, and other diagnostic information. <code>FitInfo.History</code> contains the diagnostic information.
Any other positive integer	<code>fitrlinear</code> displays and stores diagnostic information at each optimization iteration. <code>FitInfo.History</code> contains the diagnostic information.

Example: 'Verbose',1

Data Types: double | single

### SGD and ASGD Solver Options

#### BatchSize — Mini-batch size

positive integer

Mini-batch size, specified as the comma-separated pair consisting of 'BatchSize' and a positive integer. At each iteration, the software estimates the subgradient using `BatchSize` observations from the training data.

- If `X` is a numeric matrix, then the default value is 10.
- If `X` is a sparse matrix, then the default value is  $\max([10, \text{ceil}(\text{sqrt}(\text{ff}))])$ , where  $\text{ff} = \text{numel}(X) / \text{nnz}(X)$  (the fullness factor of `X`).

Example: 'BatchSize',100

Data Types: single | double

#### LearnRate — Learning rate

positive scalar

Learning rate, specified as the comma-separated pair consisting of 'LearnRate' and a positive scalar. LearnRate specifies how many steps to take per iteration. At each iteration, the gradient specifies the direction and magnitude of each step.

- If Regularization is 'ridge', then LearnRate specifies the initial learning rate  $\gamma_0$ . The software determines the learning rate for iteration  $t$ ,  $\gamma_t$ , using

$$\gamma_t = \frac{\gamma_0}{(1 + \lambda \gamma_0 t)^c}.$$

- $\lambda$  is the value of Lambda.
- If Solver is 'sgd', then  $c = 1$ .
- If Solver is 'asgd', then  $c$  is:
  - 2/3 if Learner is 'leastsquares'
  - 3/4 if Learner is 'svm' [8]
- If Regularization is 'lasso', then, for all iterations, LearnRate is constant.

By default, LearnRate is  $1/\sqrt{1+\max(\text{sum}(X.^2, \text{obsDim}))}$ , where obsDim is 1 if the observations compose the columns of X, and 2 otherwise.

Example: 'LearnRate', 0.01

Data Types: single | double

### OptimizeLearnRate — Flag to decrease learning rate

true (default) | false

Flag to decrease the learning rate when the software detects divergence (that is, overstepping the minimum), specified as the comma-separated pair consisting of 'OptimizeLearnRate' and true or false.

If OptimizeLearnRate is 'true', then:

- 1 For the few optimization iterations, the software starts optimization using LearnRate as the learning rate.
- 2 If the value of the objective function increases, then the software restarts and uses half of the current value of the learning rate.
- 3 The software iterates step 2 until the objective function decreases.

Example: `'OptimizeLearnRate', true`

Data Types: `logical`

### **TruncationPeriod** — Number of mini-batches between lasso truncation runs

10 (default) | positive integer

Number of mini-batches between lasso truncation runs, specified as the comma-separated pair consisting of `'TruncationPeriod'` and a positive integer.

After a truncation run, the software applies a soft threshold to the linear coefficients. That is, after processing  $k = \text{TruncationPeriod}$  mini-batches, the software truncates the estimated coefficient  $j$  using

$$\hat{\beta}_j^* = \begin{cases} \hat{\beta}_j - u_t & \text{if } \hat{\beta}_j > u_t, \\ 0 & \text{if } |\hat{\beta}_j| \leq u_t, \\ \hat{\beta}_j + u_t & \text{if } \hat{\beta}_j < -u_t. \end{cases}$$

- For SGD,  $\hat{\beta}_j$  is the estimate of coefficient  $j$  after processing  $k$  mini-batches.  $u_t = k\gamma_t\lambda$ .  $\gamma_t$  is the learning rate at iteration  $t$ .  $\lambda$  is the value of `Lambda`.
- For ASGD,  $\hat{\beta}_j$  is the averaged estimate coefficient  $j$  after processing  $k$  mini-batches,  $u_t = k\lambda$ .

If `Regularization` is `'ridge'`, then the software ignores `TruncationPeriod`.

Example: `'TruncationPeriod', 100`

Data Types: `single` | `double`

### **Other Regression Options**

#### **Weights** — Observation weights

`ones(size(X,2),1)/size(X,2)` (default) | numeric vector of positive values

Observation weights, specified as the comma-separated pair consisting of `'Weights'` and a numeric vector of positive values. `fitrlinear` weighs the observations in `X` with the corresponding value in `Weights`. The size of `Weights` must equal the number of observations in `X`.



fitrlinear normalizes Weights to sum to 1.

Data Types: double | single

### **ResponseName — Response variable name**

'Y' (default) | character vector

Response variable name, specified as the comma-separated pair consisting of 'ResponseName' and a character vector.

- If you supply Y, then you can use 'ResponseName' to specify a name for the response variable.
- If you supply ResponseVarName or formula, then you cannot use 'ResponseName'.

Example: 'ResponseName', 'response'

Data Types: char

### **ResponseTransform — Response transformation**

'none' (default) | function handle

Response transformation, specified as the comma-separated pair consisting of 'ResponseTransform' and either 'none' or a function handle. The default is 'none', which means  $@(x)x$ , or no transformation. For a MATLAB function or a function you define, use its function handle. The function handle must accept a vector (the original response values) and return a vector of the same size (the transformed response values).

Example: 'ResponseTransform', 'none'

Data Types: char | function\_handle

### **Cross-Validation Options**

#### **CrossVal — Cross-validation flag**

'off' (default) | 'on'

Cross-validation flag, specified as the comma-separated pair consisting of 'Crossval' and 'on' or 'off'.

If you specify 'on', then the software implements 10-fold cross-validation.

To override this cross-validation setting, use one of these name-value pair arguments: `CVPartition`, `Holdout`, or `KFold`. To create a cross-validated model, you can use one cross-validation name-value pair argument at a time only.

Example: `'Crossval', 'on'`

Data Types: `char`

### **CVPartition — Cross-validation partition**

`[]` (default) | `cvpartition` partition object

Cross-validation partition, specified as the comma-separated pair consisting of `'CVPartition'` and a `cvpartition` partition object as created by `cvpartition`. The partition object specifies the type of cross-validation, and also the indexing for training and validation sets.

To create a cross-validated model, you can use one of these four options only: `'CVPartition'`, `'Holdout'`, or `'KFold'`.

### **Holdout — Fraction of data for holdout validation**

scalar value in the range (0,1)

Fraction of data used for holdout validation, specified as the comma-separated pair consisting of `'Holdout'` and a scalar value in the range (0,1). If you specify `'Holdout', p`, then the software:

- 1 Randomly reserves  $p*100\%$  of the data as validation data, and trains the model using the rest of the data
- 2 Stores the compact, trained model in the `Trained` property of the cross-validated model.

To create a cross-validated model, you can use one of these four options only: `'CVPartition'`, `'Holdout'`, or `'KFold'`.

Example: `'Holdout', 0.1`

Data Types: `double` | `single`

### **KFold — Number of folds**

10 (default) | positive integer value greater than 1

Number of folds to use in a cross-validated classifier, specified as the comma-separated pair consisting of 'KFold' and a positive integer value greater than 1. If you specify, e.g., 'KFold',  $k$ , then the software:

- 1 Randomly partitions the data into  $k$  sets
- 2 For each set, reserves the set as validation data, and trains the model using the other  $k - 1$  sets
- 3 Stores the  $k$  compact, trained models in the cells of a  $k$ -by-1 cell vector in the Trained property of the cross-validated model.

To create a cross-validated model, you can use one of these four options only: 'CVPartition', 'Holdout', or 'KFold'.

Example: 'KFold', 8

Data Types: single | double

### SGD and ASGD Convergence Controls

#### **BatchLimit** — Maximal number of batches

positive integer

Maximal number of batches to process, specified as the comma-separated pair consisting of 'BatchLimit' and a positive integer. When the software processes BatchLimit batches, it terminates optimization.

- By default:
  - The software passes through the data PassLimit times.
  - If you specify multiple solvers, and use (A)SGD to get an initial approximation for the next solver, then the default value is `ceil(1e6/BatchSize)`. BatchSize is the value of the 'BatchSize' name-value pair argument.
- If you specify 'BatchLimit' and 'PassLimit', then the software chooses the argument that results in processing the fewest observations.
- If you specify 'BatchLimit' but not 'PassLimit', then the software processes enough batches to complete up to one entire pass through the data.

Example: 'BatchLimit', 100

Data Types: single | double

**BetaTolerance** — Relative tolerance on linear coefficients and bias term

1e-4 (default) | nonnegative scalar

Relative tolerance on the linear coefficients and the bias term (intercept), specified as the comma-separated pair consisting of 'BetaTolerance' and a nonnegative scalar.

Let  $B_t = [\beta_t' \ b_t]$ , that is, the vector of the coefficients and the bias term at optimization

iteration  $t$ . If  $\left\| \frac{B_t - B_{t-1}}{B_t} \right\|_2 < \text{BetaTolerance}$ , then optimization terminates.

If the software converges for the last solver specified in `Solver`, then optimization terminates. Otherwise, the software uses the next solver specified in `Solver`.

Example: 'BetaTolerance', 1e-6

Data Types: single | double

**NumCheckConvergence** — Number of batches to process before next convergence check

positive integer

Number of batches to process before next convergence check, specified as the comma-separated pair consisting of 'NumCheckConvergence' and a positive integer.

To specify the batch size, see `BatchSize`.

The software checks for convergence about 10 times per pass through the entire data set by default.

Example: 'NumCheckConvergence', 100

Data Types: single | double

**PassLimit** — Maximal number of passes

1 (default) | positive integer

Maximal number of passes through the data, specified as the comma-separated pair consisting of 'PassLimit' and a positive integer.

`fitrlinear` processes all observations when it completes one pass through the data.

When `fitrlinear` passes through the data `PassLimit` times, it terminates optimization.

If you specify `'BatchLimit'` and `PassLimit`, then `fitrlinear` chooses the argument that results in processing the fewest observations. For more details, see “Algorithms” on page 32-2422.

Example: `'PassLimit',5`

Data Types: `single` | `double`

### **ValidationData — Validation data for optimization convergence detection**

cell array

Data for optimization convergence detection, specified as the comma-separated pair consisting of `'ValidationData'` and a cell array.

During optimization, the software periodically estimates the loss of `ValidationData`. If the validation-data loss increases, then the software terminates optimization. For more details, see “Algorithms” on page 32-2422. To optimize hyperparameters using cross-validation, see cross-validation options such as `CrossVal`.

- `ValidationData(1)` must contain an  $m$ -by- $p$  or  $p$ -by- $m$  full or sparse matrix of predictor data that has the same orientation as `X`. The predictor variables in the training data `X` and `ValidationData{1}` must correspond. The number of observations in both sets can vary.
- `ValidationData(2)` must contain an array of  $m$  responses with length corresponding to the number of observations in `ValidationData{1}`.
- Optionally, `ValidationData(3)` can contain an  $m$ -dimensional numeric vector of observation weights. The software normalizes the weights with the validation data so that they sum to 1.

If you specify `ValidationData`, then, to display validation loss at the command line, specify a value larger than 0 for `Verbose`.

If the software converges for the last solver specified in `Solver`, then optimization terminates. Otherwise, the software uses the next solver specified in `Solver`.

By default, the software does not detect convergence by monitoring validation-data loss.

**GradientTolerance — Absolute gradient tolerance**`1e-6 (default) | nonnegative scalar`

Absolute gradient tolerance, specified as the comma-separated pair consisting of 'GradientTolerance' and a nonnegative scalar. GradientTolerance applies to these values of Solver: 'bfgs', 'lbfgs', and 'sparsa'.

Let  $\nabla L_t$  be the gradient vector of the objective function with respect to the coefficients and bias term at optimization iteration  $t$ . If  $\|\nabla L_t\|_\infty = \max|\nabla L_t| < \text{GradientTolerance}$ , then optimization terminates.

If you also specify BetaTolerance, then optimization terminates when fitrlinear satisfies either stopping criterion.

If fitrlinear converges for the last solver specified in Solver, then optimization terminates. Otherwise, fitrlinear uses the next solver specified in Solver.

Example: 'GradientTolerance', eps

Data Types: single | double

**IterationLimit — Maximal number of optimization iterations**`1000 (default) | positive integer`

Maximal number of optimization iterations, specified as the comma-separated pair consisting of 'IterationLimit' and a positive integer. IterationLimit applies to these values of Solver: 'bfgs', 'lbfgs', and 'sparsa'.

Example: 'IterationLimit', 1e7

Data Types: single | double

**Dual SGD Optimization Convergence Controls****BetaTolerance — Relative tolerance on linear coefficients and bias term**`1e-4 (default) | nonnegative scalar`

Relative tolerance on the linear coefficients and the bias term (intercept), specified as the comma-separated pair consisting of 'BetaTolerance' and a nonnegative scalar.

Let  $B_t = \begin{bmatrix} \beta_t' & b_t \end{bmatrix}$ , that is, the vector of the coefficients and the bias term at optimization

iteration  $t$ . If  $\left\| \frac{B_t - B_{t-1}}{B_t} \right\|_2 < \text{BetaTolerance}$ , then optimization terminates.

If you also specify `DeltaGradientTolerance`, then optimization terminates when the software satisfies either stopping criterion.

If the software converges for the last solver specified in `Solver`, then optimization terminates. Otherwise, the software uses the next solver specified in `Solver`.

Example: `'BetaTolerance', 1e-6`

Data Types: `single` | `double`

### **DeltaGradientTolerance — Gradient-difference tolerance**

0.1 (default) | nonnegative scalar

Gradient-difference tolerance between upper and lower pool Karush-Kuhn-Tucker (KKT) complementarity conditions on page 32-2128 violators, specified as the comma-separated pair consisting of `'DeltaGradientTolerance'` and a nonnegative scalar.

`DeltaGradientTolerance` applies to the `'dual'` value of `Solver` only.

- If the magnitude of the KKT violators is less than `DeltaGradientTolerance`, then `fitrlinear` terminates optimization.
- If `fitrlinear` converges for the last solver specified in `Solver`, then optimization terminates. Otherwise, `fitrlinear` uses the next solver specified in `Solver`.

Example: `'DeltaGapTolerance', 1e-2`

Data Types: `double` | `single`

### **NumCheckConvergence — Number of passes through entire data set to process before next convergence check**

5 (default) | positive integer

Number of passes through entire data set to process before next convergence check, specified as the comma-separated pair consisting of `'NumCheckConvergence'` and a positive integer.

Example: `'NumCheckConvergence', 100`

Data Types: `single` | `double`

**PassLimit** — Maximal number of passes

10 (default) | positive integer

Maximal number of passes through the data, specified as the comma-separated pair consisting of `'PassLimit'` and a positive integer.

When the software completes one pass through the data, it has processed all observations.

When the software passes through the data `PassLimit` times, it terminates optimization.

Example: `'PassLimit',5`

Data Types: `single` | `double`

**ValidationData** — Validation data for optimization convergence detection

cell array

Data for optimization convergence detection, specified as the comma-separated pair consisting of `'ValidationData'` and a cell array.

During optimization, the software periodically estimates the loss of `ValidationData`. If the validation-data loss increases, then the software terminates optimization. For more details, see “Algorithms” on page 32-2422. To optimize hyperparameters using cross-validation, see cross-validation options such as `CrossVal`.

- `ValidationData(1)` must contain an  $m$ -by- $p$  or  $p$ -by- $m$  full or sparse matrix of predictor data that has the same orientation as `X`. The predictor variables in the training data `X` and `ValidationData{1}` must correspond. The number of observations in both sets can vary.
- `ValidationData(2)` must contain an array of  $m$  responses with length corresponding to the number of observations in `ValidationData{1}`.
- Optionally, `ValidationData(3)` can contain an  $m$ -dimensional numeric vector of observation weights. The software normalizes the weights with the validation data so that they sum to 1.

If you specify `ValidationData`, then, to display validation loss at the command line, specify a value larger than 0 for `Verbose`.



If the software converges for the last solver specified in `Solver`, then optimization terminates. Otherwise, the software uses the next solver specified in `Solver`.

By default, the software does not detect convergence by monitoring validation-data loss.

### **BFGS, LBFGS, and SpARSA Convergence Controls**

#### **BetaTolerance** — Relative tolerance on linear coefficients and bias term

1e-4 (default) | nonnegative scalar

Relative tolerance on the linear coefficients and the bias term (intercept), specified as the comma-separated pair consisting of 'BetaTolerance' and a nonnegative scalar.

Let  $B_t = \begin{bmatrix} \beta_t' & b_t \end{bmatrix}$ , that is, the vector of the coefficients and the bias term at optimization

iteration  $t$ . If  $\left\| \frac{B_t - B_{t-1}}{B_t} \right\|_2 < \text{BetaTolerance}$ , then optimization terminates.

If you also specify `GradientTolerance`, then optimization terminates when the software satisfies either stopping criterion.

If the software converges for the last solver specified in `Solver`, then optimization terminates. Otherwise, the software uses the next solver specified in `Solver`.

Example: 'BetaTolerance', 1e-6

Data Types: single | double

#### **GradientTolerance** — Absolute gradient tolerance

1e-6 (default) | nonnegative scalar

Absolute gradient tolerance, specified as the comma-separated pair consisting of 'GradientTolerance' and a nonnegative scalar.

Let  $\nabla_{L_t}$  be the gradient vector of the objective function with respect to the coefficients and bias term at optimization iteration  $t$ . If  $\|\nabla_{L_t}\|_\infty = \max |\nabla_{L_t}| < \text{GradientTolerance}$ , then optimization terminates.

If you also specify `BetaTolerance`, then optimization terminates when the software satisfies either stopping criterion.

If the software converges for the last solver specified in the software, then optimization terminates. Otherwise, the software uses the next solver specified in `Solver`.

Example: `'GradientTolerance', 1e-5`

Data Types: `single` | `double`

### **HessianHistorySize** — Size of history buffer for Hessian approximation

15 (default) | positive integer

Size of history buffer for Hessian approximation, specified as the comma-separated pair consisting of `'HessianHistorySize'` and a positive integer. That is, at each iteration, the software composes the Hessian using statistics from the latest `HessianHistorySize` iterations.

The software does not support `'HessianHistorySize'` for `SpaRSA`.

Example: `'HessianHistorySize', 10`

Data Types: `single` | `double`

### **IterationLimit** — Maximal number of optimization iterations

1000 (default) | positive integer

Maximal number of optimization iterations, specified as the comma-separated pair consisting of `'IterationLimit'` and a positive integer. `IterationLimit` applies to these values of `Solver`: `'bfgs'`, `'lbfgs'`, and `'sparsa'`.

Example: `'IterationLimit', 500`

Data Types: `single` | `double`

### **ValidationData** — Validation data for optimization convergence detection

cell array

Data for optimization convergence detection, specified as the comma-separated pair consisting of `'ValidationData'` and a cell array.

During optimization, the software periodically estimates the loss of `ValidationData`. If the validation-data loss increases, then the software terminates optimization. For more details, see “Algorithms” on page 32-2422. To optimize hyperparameters using cross-validation, see cross-validation options such as `CrossVal`.

- `ValidationData(1)` must contain an  $m$ -by- $p$  or  $p$ -by- $m$  full or sparse matrix of predictor data that has the same orientation as `X`. The predictor variables in the training data `X` and `ValidationData{1}` must correspond. The number of observations in both sets can vary.
- `ValidationData(2)` must contain an array of  $m$  responses with length corresponding to the number of observations in `ValidationData{1}`.
- Optionally, `ValidationData(3)` can contain an  $m$ -dimensional numeric vector of observation weights. The software normalizes the weights with the validation data so that they sum to 1.

If you specify `ValidationData`, then, to display validation loss at the command line, specify a value larger than 0 for `Verbose`.

If the software converges for the last solver specified in `Solver`, then optimization terminates. Otherwise, the software uses the next solver specified in `Solver`.

By default, the software does not detect convergence by monitoring validation-data loss.

### Hyperparameter Optimization

#### **OptimizeHyperparameters** — Parameters to optimize

'none' (default) | 'auto' | 'all' | cell array of eligible parameter names | vector of `optimizableVariable` objects

Parameters to optimize, specified as:

- 'none' — Do not optimize
- 'auto' — Use {'Lambda', 'Learner'}
- 'all' — Optimize all eligible parameters
- cell array of eligible parameter names
- vector of `optimizableVariable` objects, typically the output of `hyperparameters`

The optimization attempts to minimize the cross-validation loss (error) for `fitrlinear` by varying the parameters. To control the cross-validation type and other aspects of the optimization, use the `HyperparameterOptimizationOptions` name-value pair.

---

**Note** `OptimizeHyperparameters` values override any values you set using other name-value pairs. For example, setting `OptimizeHyperparameters` to 'auto' causes the 'auto' values to apply.

---

The eligible parameters for `fitrlinear` are:

- `Lambda` — `fitrlinear` searches among positive values, by default log-scaled in the range `[1e-5/NumObservations, 1e5/NumObservations]`.
- `Learner` — `fitrlinear` searches among 'svm' and 'leastsquares'.
- `Regularization` — `fitrlinear` searches among 'ridge' and 'lasso'.

Set nondefault parameters by passing a vector of `optimizableVariable` objects that have nondefault values. For example,

```
load carsmall
params = hyperparameters('fitrlinear', [Horsepower, Weight], MPG);
params(1).Range = [1e-3, 2e4];
```

Pass `params` as the value of `OptimizeHyperparameters`.

By default, iterative display appears at the command line, and plots appear according to the number of hyperparameters in the optimization. For the optimization and plots, the objective function is  $\log(1 + \text{cross-validation loss})$  for regression, and the misclassification rate for classification. To control the iterative display, set the `HyperparameterOptimizationOptions` name-value pair, `Verbose` field. To control the plots, set the `HyperparameterOptimizationOptions` name-value pair, `ShowPlots` field.

For an example, see “Optimize a Linear Regression” on page 32-2386.

Example: 'auto'

Data Types: char | cell

**HyperparameterOptimizationOptions** — Options for optimization structure

Options for optimization, specified as a structure. Modifies the effect of the `OptimizeHyperparameters` name-value pair. All fields in the structure are optional.

Field Name	Values	Default
Optimizer	<ul style="list-style-type: none"> <li>'bayesopt' — Use Bayesian optimization. Internally, this setting calls bayesopt.</li> <li>'gridsearch' — Use grid search with NumGridDivisions values per dimension.</li> <li>'randomsearch' — Search at random among MaxObjectiveEvaluations points.</li> </ul> <p>'gridsearch' searches in a random order, using uniform sampling without replacement from the grid. After optimization, you can get a table in grid order by using the command sortrows (Mdl.ParameterOptimizationResults).</p>	'bayesopt'
AcquisitionFunctionName	<ul style="list-style-type: none"> <li>'expected-improvement-per-second-plus'</li> <li>'expected-improvement'</li> <li>'expected-improvement-plus'</li> <li>'expected-improvement-per-second'</li> <li>'lower-confidence-bound'</li> <li>'probability-of-improvement'</li> </ul> <p>For details, see the bayesopt AcquisitionFunctionName name-value pair, or “Acquisition Function Types” on page 10-4.</p>	'expected-improvement-per-second-plus'
MaxObjectiveEvaluations	Maximum number of objective function evaluations.	30 for 'bayesopt' or 'randomsearch', and the entire grid for 'gridsearch'

Field Name	Values	Default
MaxTime	Time limit, specified as a positive real. The time limit is in seconds, as measured by <code>tic</code> and <code>toc</code> . Run time can exceed <code>MaxTime</code> because <code>MaxTime</code> does not interrupt function evaluations.	Inf
NumGridDivisions	For 'gridsearch', the number of values in each dimension. Can be a vector of positive integers giving the number of values for each dimension, or a scalar that applies to all dimensions. Ignored for categorical variables.	10
ShowPlots	Logical value indicating whether to show plots. If <code>true</code> , plots the best objective function value against iteration number. If there are one or two optimization parameters, and if <code>Optimizer</code> is 'bayesopt', then <code>ShowPlots</code> also plots a model of the objective function against the parameters.	<code>true</code>
SaveIntermediateResults	Logical value indicating whether to save results when <code>Optimizer</code> is 'bayesopt'. If <code>true</code> , overwrites a workspace variable named 'BayesoptResults' at each iteration. The variable is a <code>BayesianOptimization</code> object.	<code>false</code>
Verbose	Display to the command line. <ul style="list-style-type: none"> <li>• 0 — No iterative display</li> <li>• 1 — Iterative display</li> <li>• 2 — Iterative display with extra information</li> </ul> For details, see the <code>bayesopt</code> <code>Verbose</code> name-value pair.	1

Field Name	Values	Default
UseParallel	Logical value indicating whether to run Bayesian optimization in parallel, which requires Parallel Computing Toolbox. For details, see “Parallel Bayesian Optimization” on page 10-9.	false
Repartition	Logical value indicating whether to repartition the cross-validation at every iteration. If <code>false</code> , the optimizer uses a single partition for the optimization.  <code>true</code> usually gives the most robust results because this setting takes partitioning noise into account. However, for good results, <code>true</code> requires at least twice as many function evaluations.	false
Use no more than one of the following three field names.		
CVPartition	A <code>cvpartition</code> object, as created by <code>cvpartition</code>	Kfold = 5
Holdout	A scalar in the range $(0, 1)$ representing the holdout fraction.	
Kfold	An integer greater than 1.	

Example: `struct('MaxObjectiveEvaluations', 60)`

Data Types: `struct`

## Output Arguments

### **mdl** — Trained linear regression model

RegressionLinear model object | RegressionPartitionedLinear cross-validated model object

Trained linear regression model, returned as a `RegressionLinear` model object or `RegressionPartitionedLinear` cross-validated model object.

If you set any of the name-value pair arguments `KFold`, `Holdout`, `CrossVal`, or `CVPartition`, then `Mdl` is a `RegressionPartitionedLinear` cross-validated model object. Otherwise, `Mdl` is a `RegressionLinear` model object.

To reference properties of `Mdl`, use dot notation. For example, enter `Mdl.Beta` in the Command Window to display the vector or matrix of estimated coefficients.

---

**Note** Unlike other regression models, and for economical memory usage, `RegressionLinear` and `RegressionPartitionedLinear` model objects do not store the training data or optimization details (for example, convergence history).

---

### **FitInfo** — Optimization details

structure array

Optimization details, returned as a structure array.

Fields specify final values or name-value pair argument specifications, for example, `Objective` is the value of the objective function when optimization terminates. Rows of multidimensional fields correspond to values of `Lambda` and columns correspond to values of `Solver`.

This table describes some notable fields.

Field	Description
<code>TerminationStatus</code>	<ul style="list-style-type: none"><li>Reason for optimization termination</li><li>Corresponds to a value in <code>TerminationCode</code></li></ul>
<code>FitTime</code>	Elapsed, wall-clock time in seconds



Field	Description	
History	A structure array of optimization information for each iteration. The field <code>Solver</code> stores solver types using integer coding.	
	<b>Integer</b>	<b>Solver</b>
	1	SGD
	2	ASGD
	3	Dual SGD for SVM
	4	LBFGS
	5	BFGS
6	SpaRSA	

To access fields, use dot notation. For example, to access the vector of objective function values for each iteration, enter `FitInfo.History.Objective`.

It is good practice to examine `FitInfo` to assess whether convergence is satisfactory.

### **HyperparameterOptimizationResults — Description of the cross-validation optimization of hyperparameters**

`BayesianOptimization` object | table of hyperparameters and associated values

Description of the cross-validation optimization of hyperparameters, stored as a `BayesianOptimization` object or a table of hyperparameters and associated values. Nonempty when the `OptimizeHyperparameters` name-value pair is nonempty at creation. Value depends on the setting of the `HyperparameterOptimizationOptions` name-value pair at creation:

- 'bayesopt' (default) — Object of class `BayesianOptimization`
- 'gridsearch' or 'randomsearch' — Table of hyperparameters used, observed objective function values (cross-validation loss), and rank of observations from lowest (best) to highest (worst)

Data Types: `table`

**Note** If `Learner` is 'least-squares', then the loss term in the objective function is half of the MSE. `loss` returns the MSE by default. Therefore, if you use `loss` to check the

resubstitution, or training, error then there is a discrepancy between the MSE returned by `loss` and optimization results in `FitInfo` or returned to the command line by setting a positive verbosity level using `Verbose`.

---

## Definitions

### Warm Start

A warm start is initial estimates of the beta coefficients and bias term supplied to an optimization routine for quicker convergence.

### Tips

- It is a best practice to orient your predictor matrix so that observations correspond to columns and to specify `'ObservationsIn', 'columns'`. As a result, you can experience a significant reduction in optimization-execution time.
- For better optimization accuracy if `X` is high-dimensional and `Regularization` is `'ridge'`, set any of these combinations for `Solver`:
  - `'sgd'`
  - `'asgd'`
  - `'dual'` if `Learner` is `'svm'`
  - `{'sgd', 'lbfgs'}`
  - `{'asgd', 'lbfgs'}`
  - `{'dual', 'lbfgs'}` if `Learner` is `'svm'`

Other combinations can result in poor optimization accuracy.

- For better optimization accuracy if `X` is moderate- through low-dimensional and `Regularization` is `'ridge'`, set `Solver` to `'bfgs'`.
- If `Regularization` is `'lasso'`, set any of these combinations for `Solver`:
  - `'sgd'`
  - `'asgd'`

- 'sparsa'
- {'sgd','sparsa'}
- {'asgd','sparsa'}
- When choosing between SGD and ASGD, consider that:
  - SGD takes less time per iteration, but requires more iterations to converge.
  - ASGD requires fewer iterations to converge, but takes more time per iteration.
- If  $X$  has few observations, but many predictor variables, then:
  - Specify 'PostFitBias', true.
  - For SGD or ASGD solvers, set PassLimit to a positive integer that is greater than 1, for example, 5 or 10. This setting often results in better accuracy.
- For SGD and ASGD solvers, BatchSize affects the rate of convergence.
  - If BatchSize is too small, then fitrlinear achieves the minimum in many iterations, but computes the gradient per iteration quickly.
  - If BatchSize is too large, then fitrlinear achieves the minimum in fewer iterations, but computes the gradient per iteration slowly.
- Large learning rates (see LearnRate) speed up convergence to the minimum, but can lead to divergence (that is, over-stepping the minimum). Small learning rates ensure convergence to the minimum, but can lead to slow termination.
- When using lasso penalties, experiment with various values of TruncationPeriod. For example, set TruncationPeriod to 1, 10, and then 100.
- For efficiency, fitrlinear does not standardize predictor data. To standardize  $X$ , enter

```
X = bsxfun(@rdivide,bsxfun(@minus,X,mean(X,2)),std(X,0,2));
```

The code requires that you orient the predictors and observations as the rows and columns of  $X$ , respectively. Also, for memory-usage economy, the code replaces the original predictor data the standardized data.

- After training a RegressionLinear model object by using fitrlinear, use the function predict and the trained model object to generate C code that predicts responses for new data. For details, see “Code Generation” on page 32-0 .

## Algorithms

- If you specify `ValidationData`, then, during objective-function optimization:
  - `fitrlinear` estimates the validation loss of `ValidationData` periodically using the current model, and tracks the minimal estimate.
  - When `fitrlinear` estimates a validation loss, it compares the estimate to the minimal estimate.
  - When subsequent, validation loss estimates exceed the minimal estimate five times, `fitrlinear` terminates optimization.
- If you specify `ValidationData` and to implement a cross-validation routine (`CrossVal`, `CVPartition`, `Holdout`, or `KFold`), then:
  - 1 `fitrlinear` randomly partitions  $X$  and  $Y$  according to the cross-validation routine that you choose.
  - 2 `fitrlinear` trains the model using the training-data partition. During objective-function optimization, `fitrlinear` uses `ValidationData` as another possible way to terminate optimization (for details, see the previous bullet).
  - 3 Once `fitrlinear` satisfies a stopping criterion, it constructs a trained model based on the optimized linear coefficients and intercept.
    - a If you implement  $k$ -fold cross-validation, and `fitrlinear` has not exhausted all training-set folds, then `fitrlinear` returns to Step 2 to train using the next training-set fold.
    - b Otherwise, `fitrlinear` terminates training, and then returns the cross-validated model.
  - 4 You can determine the quality of the cross-validated model. For example:
    - To determine the validation loss using the holdout or out-of-fold data from step 1, pass the cross-validated model to `kfoldLoss`.
    - To predict observations on the holdout or out-of-fold data from step 1, pass the cross-validated model to `kfoldPredict`.

## References

- [1] Ho, C. H. and C. J. Lin. “Large-Scale Linear Support Vector Regression.” *Journal of Machine Learning Research*, Vol. 13, 2012, pp. 3323–3348.

- [2] Hsieh, C. J., K. W. Chang, C. J. Lin, S. S. Keerthi, and S. Sundararajan. “A Dual Coordinate Descent Method for Large-Scale Linear SVM.” *Proceedings of the 25th International Conference on Machine Learning, ICML '08*, 2001, pp. 408–415.
- [3] Langford, J., L. Li, and T. Zhang. “Sparse Online Learning Via Truncated Gradient.” *J. Mach. Learn. Res.*, Vol. 10, 2009, pp. 777–801.
- [4] Nocedal, J. and S. J. Wright. *Numerical Optimization*, 2nd ed., New York: Springer, 2006.
- [5] Shalev-Shwartz, S., Y. Singer, and N. Srebro. “Pegasos: Primal Estimated Sub-Gradient Solver for SVM.” *Proceedings of the 24th International Conference on Machine Learning, ICML '07*, 2007, pp. 807–814.
- [6] Wright, S. J., R. D. Nowak, and M. A. T. Figueiredo. “Sparse Reconstruction by Separable Approximation.” *Trans. Sig. Proc.*, Vol. 57, No 7, 2009, pp. 2479–2493.
- [7] Xiao, Lin. “Dual Averaging Methods for Regularized Stochastic Learning and Online Optimization.” *J. Mach. Learn. Res.*, Vol. 11, 2010, pp. 2543–2596.
- [8] Xu, Wei. “Towards Optimal One Pass Large Scale Learning with Averaged Stochastic Gradient Descent.” *CoRR*, abs/1107.2490, 2011.

## Extended Capabilities

### Tall Arrays

Calculate with arrays that have more rows than fit in memory.

Usage notes and limitations:

- Supported syntaxes for tall arrays  $X$  and  $Y$  are:
  - `obj = fitrlinear(X, Y)`
  - `obj = fitrlinear(X, Y, Name, Value)`
- Some name-value pairs have different defaults and values compared to the in-memory `fitrlinear` function. Supported name-value pairs, and any differences, are:

- 'Epsilon'
- 'ObservationsIn' — Supports only 'rows'.
- 'Lambda' — Can be 'auto' (default) or a scalar.
- 'Learner'
- 'Regularization' — Supports only 'ridge'.
- 'Solver' — Supports only 'lbfgs'.
- 'Verbose' — Default value is 1
- 'Beta'
- 'Bias'
- 'FitBias' — Supports only true.
- 'Weights' — Value must be a tall array.
- 'HessianHistorySize'
- 'BetaTolerance' — Default value is relaxed to 1e-3.
- 'GradientTolerance' — Default value is relaxed to 1e-3.
- 'IterationLimit' — Default value is relaxed to 20.
- For tall arrays `fitrlinear` implements LBFGS by distributing the calculation of the loss and the gradient among different parts of the tall array at each iteration. Other solvers are not available for tall arrays.

When initial values for `Beta` and `Bias` are not given, `fitrlinear` first refines the initial estimates of the parameters by fitting the model locally to parts of the data and combining the coefficients by averaging.

For more information, see “Tall Arrays” (MATLAB).

## See Also

`RegressionLinear` | `RegressionPartitionedLinear` | `fitclinear` | `fitlm` | `fitrsvm` | `kfoldLoss` | `kfoldPredict` | `lasso` | `predict` | `ridge`

**Introduced in R2016a**

## fitrm

Fit repeated measures model

### Syntax

```
rm = fitrm(t,modelspec)
rm = fitrm(t,modelspec,Name,Value)
```

### Description

`rm = fitrm(t,modelspec)` returns a repeated measures model, specified by `modelspec`, fitted to the variables in the table or dataset array `t`.

`rm = fitrm(t,modelspec,Name,Value)` returns a repeated measures model, with additional options specified by one or more `Name, Value` pair arguments.

For example, you can specify the hypothesis for the within-subject factors.

### Examples

#### Fit a Repeated Measures Model

Load the sample data.

```
load fisheriris
```

The column vector `species` consists of iris flowers of three different species: `setosa`, `versicolor`, and `virginica`. The double matrix `meas` consists of four types of measurements on the flowers: the length and width of sepals and petals in centimeters, respectively.

Store the data in a table array.

```
t = table(species,meas(:,1),meas(:,2),meas(:,3),meas(:,4),...
'VariableNames',{'species','meas1','meas2','meas3','meas4'});
Meas = table([1 2 3 4'],'VariableNames',{'Measurements'});
```

Fit a repeated measures model, where the measurements are the responses and the species is the predictor variable.

```
rm = fitrm(t,'meas1-meas4~species','WithinDesign',Meas)
```

```
rm =
```

```
RepeatedMeasuresModel with properties:
```

```
Between Subjects:
```

```
    BetweenDesign: [150x5 table]
```

```
    ResponseNames: {'meas1' 'meas2' 'meas3' 'meas4'}
```

```
    BetweenFactorNames: {'species'}
```

```
    BetweenModel: '1 + species'
```

```
Within Subjects:
```

```
    WithinDesign: [4x1 table]
```

```
    WithinFactorNames: {'Measurements'}
```

```
    WithinModel: 'separatemeans'
```

```
Estimates:
```

```
    Coefficients: [3x4 table]
```

```
    Covariance: [4x4 table]
```

Display the coefficients.

```
rm.Coefficients
```

```
ans =
```

```
3x4 table
```

	meas1	meas2	meas3	meas4
(Intercept)	5.8433	3.0573	3.758	1.1993
species_setosa	-0.83733	0.37067	-2.296	-0.95333
species_versicolor	0.092667	-0.28733	0.502	0.12667



fitrm uses the 'effects' contrasts which means that the coefficients sum to 0. The `rm.DesignMatrix` has one column of 1s for the intercept, and two other columns `species_setosa` and `species_versicolor`, which are as follows:

$$\text{species\_setosa} = \begin{cases} 1 & \text{if setosa} \\ 0 & \text{if versicolor} \\ -1 & \text{if virginica} \end{cases} \quad \text{and} \quad \text{species\_versicolor} = \begin{cases} 0 & \text{if setosa} \\ 1 & \text{if versicolor} \\ -1 & \text{if virginica} \end{cases}$$

Display the covariance matrix.

```
rm.Covariance
```

```
ans =
```

```
4x4 table
```

	meas1	meas2	meas3	meas4
meas1	0.26501	0.092721	0.16751	0.038401
meas2	0.092721	0.11539	0.055244	0.03271
meas3	0.16751	0.055244	0.18519	0.042665
meas4	0.038401	0.03271	0.042665	0.041882

## Specify the Within-Subject Hypothesis

Load the sample data.

```
load(fullfile(matlabroot, 'examples', 'stats', 'longitudinalData.mat'));
```

The matrix `Y` contains response data for 16 individuals. The response is the blood level of a drug measured at five time points (time = 0, 2, 4, 6, and 8). Each row of `Y` corresponds to an individual, and each column corresponds to a time point. The first eight subjects are female, and the second eight subjects are male. This is simulated data.

Define a variable that stores gender information.

```
Gender = ['F' 'F' 'F' 'F' 'F' 'F' 'F' 'F' 'M' 'M' 'M' 'M' 'M' 'M' 'M' 'M'];
```

Store the data in a proper table array format to conduct repeated measures analysis.

```
t = table(Gender, Y(:,1), Y(:,2), Y(:,3), Y(:,4), Y(:,5), ...  
'VariableNames', {'Gender', 't0', 't2', 't4', 't6', 't8'});
```

Define the within-subjects variable.

```
Time = [0 2 4 6 8]';
```

Fit a repeated measures model, where blood levels are the responses and gender is the predictor variable. Also define the hypothesis for within-subject factors.

```
rm = fitrm(t, 't0-t8 ~ Gender', 'WithinDesign', Time, 'WithinModel', 'orthogonalcontrasts')
```

```
rm =
```

```
RepeatedMeasuresModel with properties:
```

```
Between Subjects:
```

```
    BetweenDesign: [16x6 table]
```

```
    ResponseNames: {'t0' 't2' 't4' 't6' 't8'}
```

```
BetweenFactorNames: {'Gender'}
```

```
    BetweenModel: '1 + Gender'
```

```
Within Subjects:
```

```
    WithinDesign: [5x1 table]
```

```
WithinFactorNames: {'Time'}
```

```
    WithinModel: 'orthogonalcontrasts'
```

```
Estimates:
```

```
    Coefficients: [2x5 table]
```

```
    Covariance: [5x5 table]
```

## Fit a Model with Covariates

Load the sample data.

```
load repeatedmeas
```

The table `between` includes the eight repeated measurements, `y1` through `y8`, as responses and the between-subject factors `Group`, `Gender`, `IQ`, and `Age`. `IQ` and `Age` as continuous variables. The table `within` includes the within-subject factors `w1` and `w2`.

Fit a repeated measures model, where age, IQ, and group, gender are the predictor variables, and the model includes the interaction effect of group and gender. Also define the within-subject factors.

```
rm = fitrm(between, 'y1-y8 ~ Group*Gender+Age+IQ', 'WithinDesign', within)
```

```
rm =
```

```
RepeatedMeasuresModel with properties:
```

```
Between Subjects:
```

```
    BetweenDesign: [30x12 table]
```

```
    ResponseNames: {'y1' 'y2' 'y3' 'y4' 'y5' 'y6' 'y7' 'y8'}
```

```
    BetweenFactorNames: {'Age' 'IQ' 'Group' 'Gender'}
```

```
    BetweenModel: '1 + Age + IQ + Group*Gender'
```

```
Within Subjects:
```

```
    WithinDesign: [8x2 table]
```

```
    WithinFactorNames: {'w1' 'w2'}
```

```
    WithinModel: 'separatemeans'
```

```
Estimates:
```

```
    Coefficients: [8x8 table]
```

```
    Covariance: [8x8 table]
```

Display the coefficients.

```
rm.Coefficients
```

```
ans =
```

```
8x8 table
```

	y1	y2	y3	y4	y5
(Intercept)	141.38	195.25	9.8663	-49.154	157.77
Age	0.32042	-4.7672	-1.2748	0.6216	-1.0621
IQ	-1.2671	-1.1653	0.05862	0.4288	-1.4518
Group_A	-1.2195	-9.6186	22.532	15.303	12.602
Group_B	2.5186	1.417	-2.2501	0.50181	8.0907
Gender_Female	5.3957	-3.9719	8.5225	9.3403	6.0909

Group_A:Gender_Female	4.1046	10.064	-7.3053	-3.3085	4.6751
Group_B:Gender_Female	-0.48486	-2.9202	1.1222	0.69715	-0.065945

The display shows the coefficients for fitting the repeated measures as a function of the terms in the between-subjects model.

## Input Arguments

### **`t`** — Input data

table

Input data, which includes the values of the response variables and the between-subject factors to use as predictors in the repeated measures model, specified as a table.

Data Types: table

### **`modelspec`** — Formula for model specification

character vector of the form `'y1-yk ~ terms'`

Formula for model specification, specified as a character vector of the form `'y1-yk ~ terms'`. The responses and terms are specified using Wilkinson notation on page 32-2432. `fitrm` treats the variables used in model terms as categorical if they are categorical (nominal or ordinal), logical, char arrays, or a cell arrays of character vectors.

For example, if you have four repeated measures as responses and the factors `x1`, `x2`, and `x3` as the predictor variables, then you can define a repeated measures model as follows.

Example: `'y1-y4 ~ x1 + x2 * x3'`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of Name, Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'WithinDesign', 'W', 'WithinModel', 'w1+w2'` specifies the matrix `w` as the design matrix for within-subject factors, and the model for within-subject factors `w1` and `w2` is `'w1+w2'`.

**WithinDesign** — Design for within-subject factors

numeric vector of length  $r$  (default) |  $r$ -by- $k$  numeric matrix |  $r$ -by- $k$  table

Design for within-subject factors, specified as the comma-separated pair consisting of 'WithinDesign' and one of the following:

- Numeric vector of length  $r$ , where  $r$  is the number of repeated measures.

In this case, `fitrm` treats the values in the vector as continuous, and these are typically time values.

- $r$ -by- $k$  numeric matrix of the values of the  $k$  within-subject factors,  $w_1, w_2, \dots, w_k$ .

In this case, `fitrm` treats all  $k$  variables as continuous.

- $r$ -by- $k$  table that contains the values of the  $k$  within-subject factors.

In this case, `fitrm` treats all numeric variables as continuous, and all categorical variables as categorical.

For example, if the table `weeks` contains the values of the within-subject factors, then you can define the design table as follows.

Example: 'WithinDesign', weeks

Data Types: single | double | table

**WithinModel** — Model specifying within-subject hypothesis test

'separatemeans' (default) | 'orthogonalcontrasts' | character vector that defines a model

Model specifying the within-subject hypothesis test, specified as the comma-separated pair consisting of 'WithinModel' and one of the following:

- 'separatemeans' — Compute a separate mean for each group.
- 'orthogonalcontrasts' — This is valid only when the within-subject model has a single numeric factor  $T$ . Responses are the average, the slope of centered  $T$ , and, in general, all orthogonal contrasts for a polynomial up to  $T^{(p-1)}$ , where  $p$  is the number of rows in the within-subject model.
- A character vector that defines a model specification in the within-subject factors. You can define the model based on the rules for the terms in `modelspec`.

For example, if there are three within-subject factors  $w_1$ ,  $w_2$ , and  $w_3$ , then you can specify a model for the within-subject factors as follows.

Example: `'WithinModel', 'w1+w2+w2*w3'`

Data Types: `char`

## Output Arguments

**rm** — Repeated measures model

`RepeatedMeasuresModel` object

Repeated measures model, returned as a `RepeatedMeasuresModel` object.

For properties and methods of this object, see `RepeatedMeasuresModel`.

## Definitions

### Model Specification Using Wilkinson Notation

Wilkinson notation describes the factors present in models. It does not describe the multipliers (coefficients) of those factors.

The following rules specify the responses in `modelspec`.

Wilkinson Notation	Meaning
$Y_1, Y_2, Y_3$	Specific list of variables
$Y_1-Y_5$	All table variables from $Y_1$ through $Y_5$

The following rules specify terms in `modelspec`.

Wilkinson notation	Factors in Standard Notation
1	Constant (intercept) term
$X^k$ , where $k$ is a positive integer	$X, X^2, \dots, X^k$
$X_1 + X_2$	$X_1, X_2$
$X_1 * X_2$	$X_1, X_2, X_1 * X_2$

Wilkinson notation	Factors in Standard Notation
$X1 : X2$	$X1 * X2$ only
$-X2$	Do not include $X2$
$X1 * X2 + X3$	$X1, X2, X3, X1 * X2$
$X1 + X2 + X3 + X1 : X2$	$X1, X2, X3, X1 * X2$
$X1 * X2 * X3 - X1 : X2 : X3$	$X1, X2, X3, X1 * X2, X1 * X3, X2 * X3$
$X1 * (X2 + X3)$	$X1, X2, X3, X1 * X2, X1 * X3$

Statistics and Machine Learning Toolbox notation always includes a constant term unless you explicitly remove the term using  $-1$ .

## See Also

`RepeatedMeasuresModel`

Introduced in R2014a

## fitdist

Fit probability distribution object to data

### Syntax

```
pd = fitdist(x,distname)
pd = fitdist(x,distname,Name,Value)

[pdca,gn,gl] = fitdist(x,distname,'By',groupvar)
[pdca,gn,gl] = fitdist(x,distname,'By',groupvar,Name,Value)
```

### Description

`pd = fitdist(x,distname)` creates a probability distribution object by fitting the distribution specified by `distname` to the data in column vector `x`.

`pd = fitdist(x,distname,Name,Value)` creates the probability distribution object with additional options specified by one or more name-value pair arguments. For example, you can indicate censored data or specify control parameters for the iterative fitting algorithm.

`[pdca,gn,gl] = fitdist(x,distname,'By',groupvar)` creates probability distribution objects by fitting the distribution specified by `distname` to the data in `x` based on the grouping variable `groupvar`. It returns a cell array of fitted probability distribution objects, `pdca`, a cell array of group labels, `gn`, and a cell array of grouping variable levels, `gl`.

`[pdca,gn,gl] = fitdist(x,distname,'By',groupvar,Name,Value)` returns the above output arguments using additional options specified by one or more name-value pair arguments. For example, you can indicate censored data or specify control parameters for the iterative fitting algorithm.

### Examples



## Fit a Normal Distribution to Data

Load the sample data. Create a vector containing the patients' weight data.

```
load hospital
x = hospital.Weight;
```

Create a normal distribution object by fitting it to the data.

```
pd = fitdist(x, 'Normal')

pd =

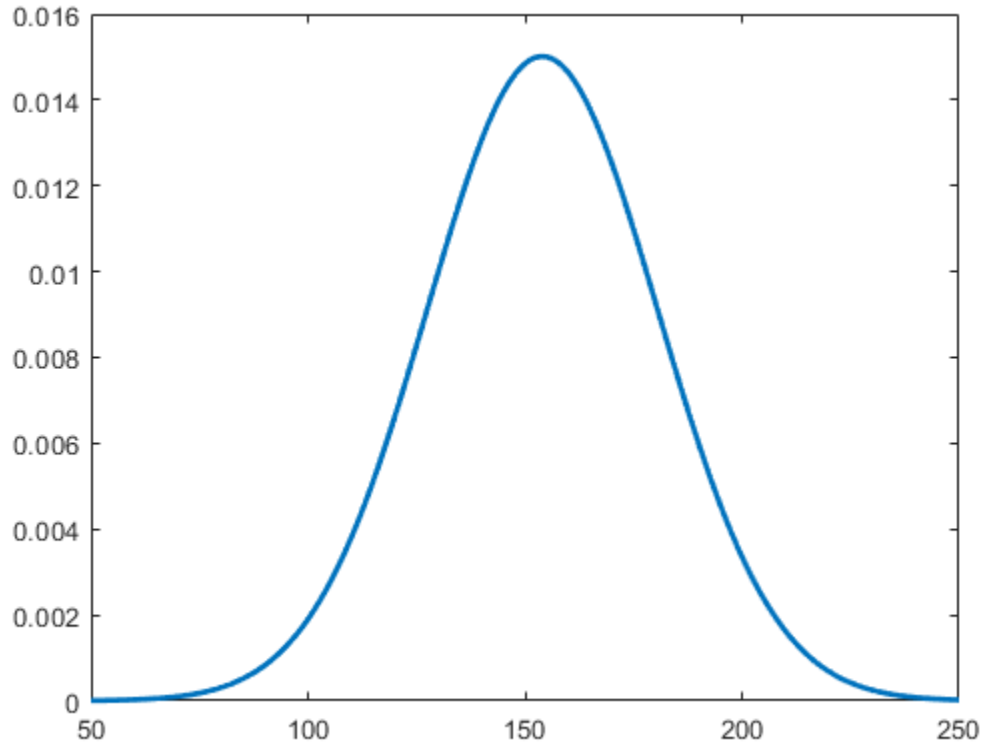
NormalDistribution

Normal distribution
    mu =      154    [148.728, 159.272]
    sigma = 26.5714 [23.3299, 30.8674]
```

The intervals next to the parameter estimates are the 95% confidence intervals for the distribution parameters.

Plot the pdf of the distribution.

```
x_values = 50:1:250;
y = pdf(pd, x_values);
plot(x_values, y, 'LineWidth', 2)
```



### Fit a Kernel Distribution to Data

Load the sample data. Create a vector containing the patients' weight data.

```
load hospital
x = hospital.Weight;
```

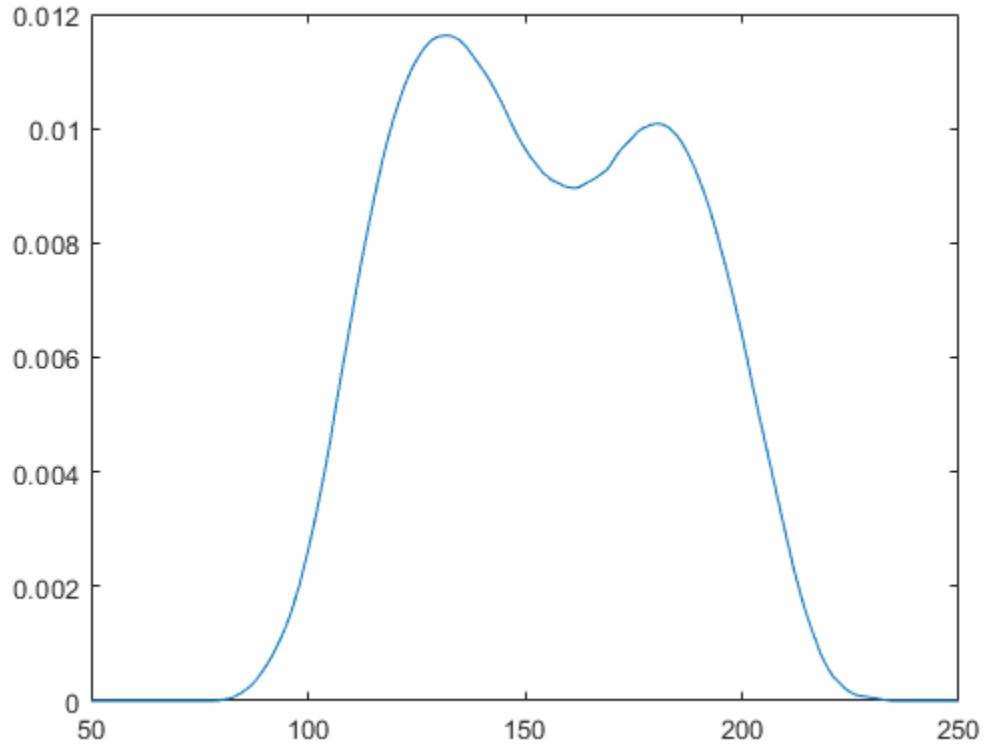
Create a kernel distribution object by fitting it to the data. Use the Epanechnikov kernel function.

```
pd = fitdist(x, 'Kernel', 'Kernel', 'epanechnikov')
```

```
pd =  
  
KernelDistribution  
  
Kernel = epanechnikov  
Bandwidth = 14.3792  
Support = unbounded
```

**Plot the pdf of the distribution.**

```
x_values = 50:1:250;  
y = pdf(pd,x_values);  
plot(x_values,y)
```



### Fit Normal Distributions to Grouped Data

Load the sample data. Create a vector containing the patients' weight data.

```
load hospital
x = hospital.Weight;
```

Create normal distribution objects by fitting them to the data, grouped by patient gender.

```
gender = hospital.Sex;
[pdca,gn,gl] = fitdist(x, 'Normal', 'By', gender)
```

```
pdca =  
  
    1x2 cell array  
  
    {1x1 prob.NormalDistribution}    {1x1 prob.NormalDistribution}  
  
gn =  
  
    2x1 cell array  
  
    {'Female'}  
    {'Male'  }  
  
gl =  
  
    2x1 cell array  
  
    {'Female'}  
    {'Male'  }
```

The cell array `pdca` contains two probability distribution objects, one for each gender group. The cell array `gn` contains two group labels. The cell array `gl` contains two group levels.

View each distribution in the cell array `pdca` to compare the mean, `mu`, and the standard deviation, `sigma`, grouped by patient gender.

```
female = pdca{1} % Distribution for females
```

```
female =  
  
NormalDistribution  
  
Normal distribution  
    mu = 130.472    [128.183, 132.76]  
    sigma = 8.30339    [6.96947, 10.2736]
```

```
male = pdca{2} % Distribution for males
```

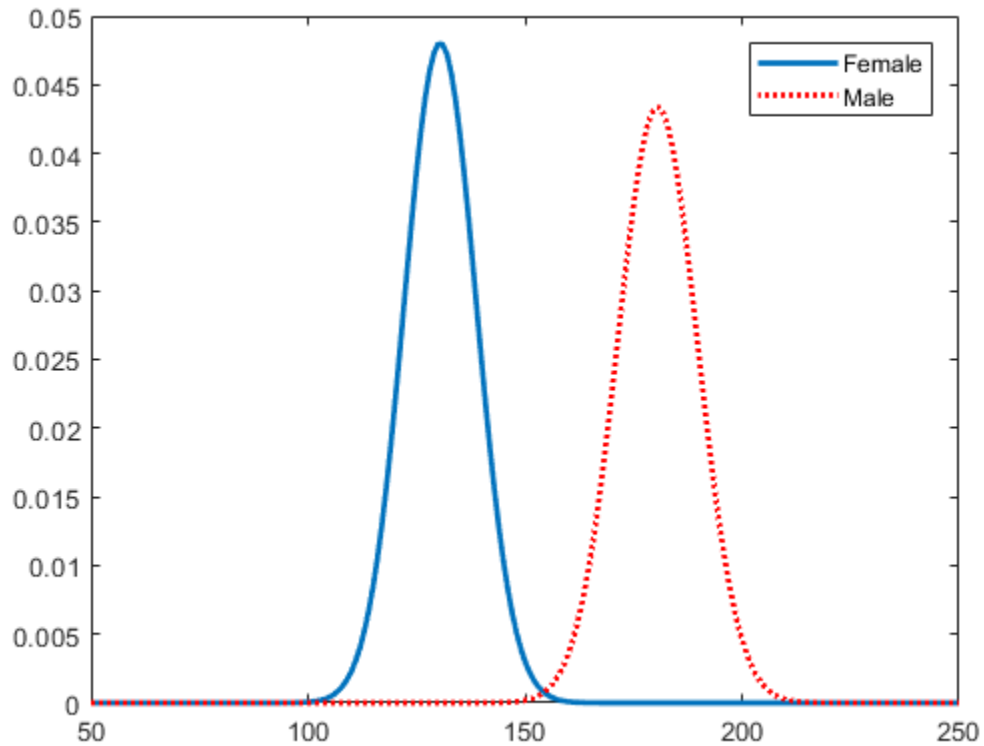
```
male =  
  
    NormalDistribution  
  
Normal distribution  
    mu = 180.532    [177.833, 183.231]  
    sigma = 9.19322    [7.63933, 11.5466]
```

Compute the pdf of each distribution.

```
x_values = 50:1:250;  
femalepdf = pdf(female,x_values);  
malepdf = pdf(male,x_values);
```

Plot the pdfs for a visual comparison of weight distribution by gender.

```
figure  
plot(x_values,femalepdf,'LineWidth',2)  
hold on  
plot(x_values,malepdf,'Color','r','LineStyle',':','LineWidth',2)  
legend(gn,'Location','NorthEast')  
hold off
```



### Fit Kernel Distributions to Grouped Data

Load the sample data. Create a vector containing the patients' weight data.

```
load hospital
x = hospital.Weight;
```

Create kernel distribution objects by fitting them to the data, grouped by patient gender. Use a triangular kernel function.

```
gender = hospital.Sex;
[pdca,gn,gl] = fitdist(x,'Kernel','By',gender,'Kernel','triangle');
```

View each distribution in the cell array `pdca` to see the kernel distributions for each gender.

```
female = pdca{1} % Distribution for females
```

```
female =
```

```
KernelDistribution  
  
Kernel = triangle  
Bandwidth = 4.25894  
Support = unbounded
```

```
male = pdca{2} % Distribution for males
```

```
male =
```

```
KernelDistribution  
  
Kernel = triangle  
Bandwidth = 5.08961  
Support = unbounded
```

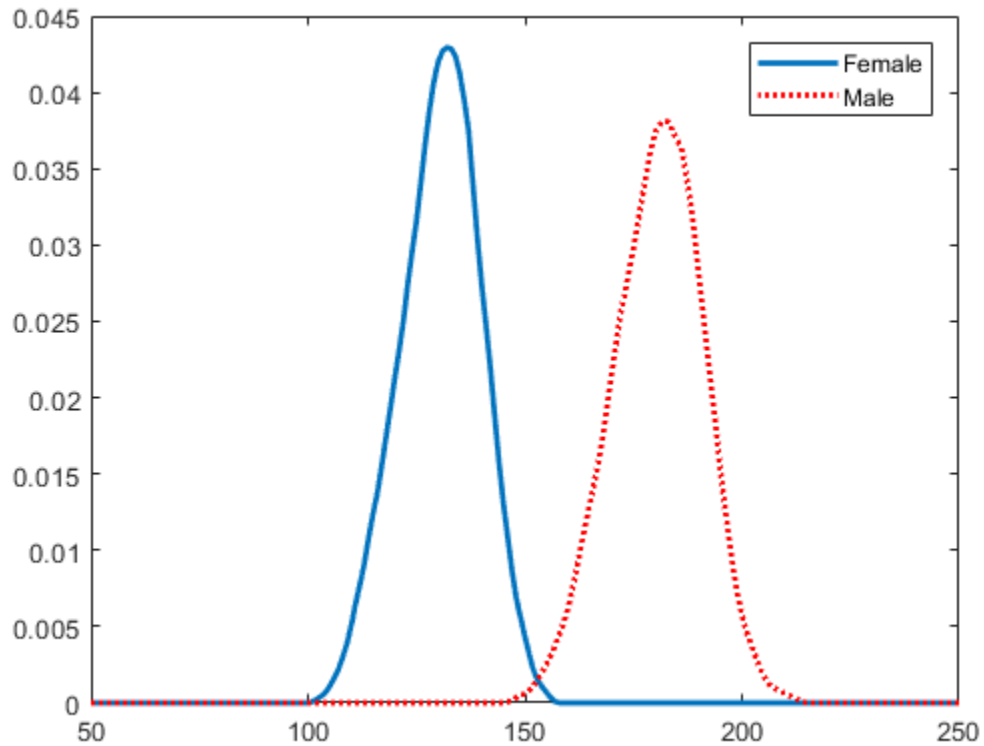
Compute the pdf of each distribution.

```
x_values = 50:1:250;  
femalepdf = pdf(female,x_values);  
malepdf = pdf(male,x_values);
```

Plot the pdfs for a visual comparison of weight distribution by gender.

```
figure  
plot(x_values,femalepdf,'LineWidth',2)  
hold on  
plot(x_values,malepdf,'Color','r','LineStyle',':','LineWidth',2)  
legend(gn,'Location','NorthEast')  
hold off
```





## Input Arguments

### **x** — Input data

column vector

Input data, specified as a column vector. `fitdist` ignores NaN values in `x`. Additionally, any NaN values in the censoring vector or frequency vector causes `fitdist` to ignore the corresponding values in `x`.

Data Types: `single` | `double`

**distname** — Distribution name

character vector

Distribution name, specified as one of the following character vectors. The distribution specified by `distname` determines the type of the returned probability distribution object.

Distribution Name	Description	Distribution Object
'Beta'	Beta distribution	BetaDistribution
'Binomial'	Binomial distribution	BinomialDistribution
'BirnbbaumSaunders'	Birnbbaum-Saunders distribution	BirnbbaumSaundersDistribution
'Burr'	Burr distribution	BurrDistribution
'Exponential'	Exponential distribution	ExponentialDistribution
'ExtremeValue'	Extreme Value distribution	ExtremeValueDistribution
'Gamma'	Gamma distribution	GammaDistribution
'GeneralizedExtremeValue'	Generalized Extreme Value distribution	GeneralizedExtremeValueDistribution
'GeneralizedPareto'	Generalized Pareto distribution	GeneralizedParetoDistribution
'HalfNormal'	Half-normal distribution	HalfNormalDistribution
'InverseGaussian'	Inverse Gaussian distribution	InverseGaussianDistribution
'Kernel'	Kernel distribution	KernelDistribution
'Logistic'	Logistic distribution	LogisticDistribution
'Loglogistic'	Loglogistic distribution	LoglogisticDistribution
'Lognormal'	Lognormal distribution	LognormalDistribution
'Nakagami'	Nakagami distribution	NakagamiDistribution
'NegativeBinomial'	Negative Binomial distribution	NegativeBinomialDistribution
'Normal'	Normal distribution	NormalDistribution

Distribution Name	Description	Distribution Object
'Poisson'	Poisson distribution	PoissonDistribution
'Rayleigh'	Rayleigh distribution	RayleighDistribution
'Rician'	Rician distribution	RicianDistribution
'Stable'	Stable distribution	StableDistribution
'tLocationScale'	$t$ Location-Scale distribution	tLocationScaleDistribution
'Weibull'	Weibull distribution	WeibullDistribution

### **groupvar** — Grouping variable

categorical array | logical or numeric vector | cell array of character vectors

Grouping variable, specified as a categorical array, logical or numeric vector, or cell array of character vectors. Each unique value in a grouping variable defines a group.

For example, if `Gender` is a cell array of character vectors with values 'Male' and 'Female', you can use `Gender` as a grouping variable to fit a distribution to your data by gender.

More than one grouping variable can be used by specifying a cell array of grouping variable names. Observations are placed in the same group if they have common values of all specified grouping variables.

For example, if `Smoker` is a logical vector with values 0 for nonsmokers and 1 for smokers, then specifying the cell array `{Gender, Smoker}` divides observations into four groups: Male Smoker, Male Nonsmoker, Female Smoker, and Female Nonsmoker.

Example: `{Gender, Smoker}`

Data Types: `single` | `double` | `logical` | `cell` | `char`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `fitdist(x, 'Kernel', 'Kernel', 'triangle')` fits a kernel distribution object to the data in `x` using a triangular kernel function.

**Censoring — Logical flag for censored data**

0 (default) | vector of logical values

Logical flag for censored data, specified as the comma-separated pair consisting of 'Censoring' and a vector of logical values that is the same size as input vector `x`. The value is 1 when the corresponding element in `x` is a right-censored observation and 0 when the corresponding elements is an exact observation. The default is a vector of 0s, indicating that all observations are exact.

`fitdist` ignores any NaN values in this censoring vector. Additionally, any NaN values in `x` or the frequency vector causes `fitdist` to ignore the corresponding values in the censoring vector.

Data Types: `logical`

**Frequency — Observation frequency**

1 (default) | vector of nonnegative integer values

Observation frequency, specified as the comma-separated pair consisting of 'Frequency' and a vector of nonnegative integer values that is the same size as input vector `x`. Each element of the frequency vector specifies the frequencies for the corresponding elements in `x`. The default is a vector of 1s, indicating that each value in `x` only appears once.

`fitdist` ignores any NaN values in this frequency vector are ignored by the fitting calculations. Additionally, any NaN values in `x` or the censoring vector causes `fitdist` to ignore the corresponding values in the frequency vector.

Data Types: `logical`

**Options — Control parameters**

structure

Control parameters for the iterative fitting algorithm, specified as the comma-separated pair consisting of 'Options' and a structure you create using `statset`.

Data Types: `struct`

**NTrials — Number of trials**

positive integer value

Number of trials for the binomial distribution, specified as the comma-separated pair consisting of 'NTrials' and a positive integer value. You must specify `distname` as 'Binomial' to use this option.

Data Types: `single` | `double`

**Theta — Threshold parameter**

0 (default) | scalar value

Threshold parameter for the generalized Pareto distribution, specified as the comma-separated pair consisting of 'Theta' and a scalar value. You must specify `distname` as 'GeneralizedPareto' to use this option.

Data Types: `single` | `double`

**mu — Location parameter**

0 (default) | scalar value

Location parameter for the half-normal distribution, specified as the comma-separated pair consisting of 'mu' and a scalar value. You must specify `distname` as 'HalfNormal' to use this option.

Data Types: `single` | `double`

**kernel — Kernel smoother type**

'normal' (default) | 'box' | 'triangle' | 'epanechnikov'

Kernel smoother type, specified as the comma-separated pair consisting of 'Kernel' and one of the following:

- 'normal'
- 'box'
- 'triangle'
- 'epanechnikov'

You must specify `distname` as 'Kernel' to use this option.

**Support — Kernel density support**

'unbounded' (default) | 'positive' | two-element vector

Kernel density support, specified as the comma-separated pair consisting of 'Support' and 'unbounded', 'positive', or a two-element vector.

'unbounded'                                      Density can extend over the whole real line.

'positive'                                        Density is restricted to positive values.

Alternatively, you can specify a two-element vector giving finite lower and upper limits for the support of the density.

You must specify `distname` as 'Kernel' to use this option.

Data Types: `single` | `double`

### **width** — Bandwidth of kernel smoothing window

scalar value

Bandwidth of the kernel smoothing window, specified as the comma-separated pair consisting of 'width' and a scalar value. The default value used by `fitdist` is optimal for estimating normal densities, but you might want to choose a smaller value to reveal features such as multiple modes. You must specify `distname` as 'Kernel' to use this option.

Data Types: `single` | `double`

## Output Arguments

### **pd** — Probability distribution

probability distribution object

Probability distribution, returned as a probability distribution object. The distribution specified by `distname` determines the class type of the returned probability distribution object.

### **pdca** — Probability distribution objects

cell array

Probability distribution objects of the type specified by `distname`, returned as a cell array.

### **gn** — Group labels

cell array of character vectors

Group labels, returned as a cell array of character vectors.

### **g1 — Grouping variable levels**

cell array of character vectors

Grouping variable levels, returned as a cell array of character vectors containing one column for each grouping variable.

## Algorithms

The `fitdist` function fits most distributions using maximum likelihood estimation. Two exceptions are the normal and lognormal distributions with uncensored data.

- For the uncensored normal distribution, the estimated value of the sigma parameter is the square root of the unbiased estimate of the variance.
- For the uncensored lognormal distribution, the estimated value of the sigma parameter is the square root of the unbiased estimate of the variance of the log of the data.

## Alternative Functionality

### App

The Distribution Fitter app opens a graphical user interface for you to import data from the workspace and interactively fit a probability distribution to that data. You can then save the distribution to the workspace as a probability distribution object. Open the Distribution Fitter app using `distributionFitter`, or click Distribution Fitter on the Apps tab.

### References

- [1] Johnson, N. L., S. Kotz, and N. Balakrishnan. *Continuous Univariate Distributions*. Vol. 1, Hoboken, NJ: Wiley-Interscience, 1993.
- [2] Johnson, N. L., S. Kotz, and N. Balakrishnan. *Continuous Univariate Distributions*. Vol. 2, Hoboken, NJ: Wiley-Interscience, 1994.

[3] Bowman, A. W., and A. Azzalini. *Applied Smoothing Techniques for Data Analysis*. New York: Oxford University Press, 1997.

## See Also

`distributionFitter` | `histfit` | `makedist` | `mle` | `paramci`

**Introduced in R2009a**



# fitensemble

Fit ensemble of learners for classification and regression

`fitensemble` can boost or bag decision tree learners or discriminant analysis classifiers. The function can also train random subspace ensembles of KNN or discriminant analysis classifiers.

For simpler interfaces that fit classification and regression ensembles, instead use `fitcensemble` and `fitrensemble`, respectively. Also, `fitcensemble` and `fitrensemble` provide options for Bayesian optimization.

## Syntax

```
Mdl = fitensemble(Tbl, ResponseVarName, Method, NLearn, Learners)
```

```
Mdl = fitensemble(Tbl, formula, Method, NLearn, Learners)
```

```
Mdl = fitensemble(Tbl, Y, Method, NLearn, Learners)
```

```
Mdl = fitensemble(X, Y, Method, NLearn, Learners)
```

```
Mdl = fitensemble(___, Name, Value)
```

## Description

`Mdl = fitensemble(Tbl, ResponseVarName, Method, NLearn, Learners)` returns a trained ensemble model object that contains the results of fitting an ensemble of `NLearn` classification or regression learners (`Learners`) to all variables in the table `Tbl`. `ResponseVarName` is the name of the response variable in `Tbl`. `Method` is the ensemble-aggregation method.

`Mdl = fitensemble(Tbl, formula, Method, NLearn, Learners)` fits the model specified by `formula`.

`Mdl = fitensemble(Tbl, Y, Method, NLearn, Learners)` treats all variables in `Tbl` as predictor variables. `Y` is the response variable that is not in `Tbl`.

`Mdl = fitensemble(X, Y, Method, NLearn, Learners)` trains an ensemble using the predictor data in `X` and response data in `Y`.

`Mdl = fitensemble( ____, Name, Value)` trains an ensemble using additional options specified by one or more `Name, Value` pair arguments and any of the previous syntaxes. For example, you can specify the class order, to implement 10-fold cross-validation, or the learning rate.

## Examples

### Estimate the Resubstitution Loss of a Boosting Ensemble

Estimate the resubstitution loss of a trained, boosting classification ensemble of decision trees.

Load the ionosphere data set.

```
load ionosphere;
```

Train a decision tree ensemble using AdaBoost, 100 learning cycles, and the entire data set.

```
ClassTreeEns = fitensemble(X, Y, 'AdaBoostM1', 100, 'Tree');
```

`ClassTreeEns` is a trained `ClassificationEnsemble` ensemble classifier.

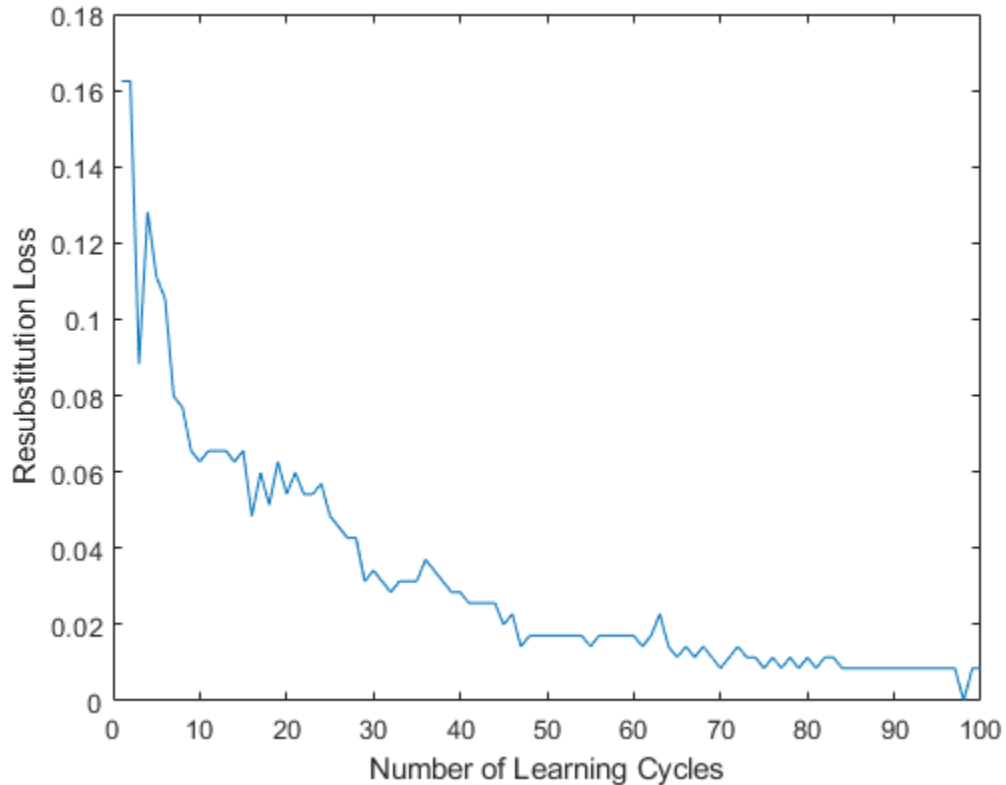
Determine the cumulative resubstitution losses (i.e., the cumulative misclassification error of the labels in the training data).

```
rsLoss = resubLoss(ClassTreeEns, 'Mode', 'Cumulative');
```

`rsLoss` is a 100-by-1 vector, where element  $k$  contains the resubstitution loss after the first  $k$  learning cycles.

Plot the cumulative resubstitution loss over the number of learning cycles.

```
plot(rsLoss);  
xlabel('Number of Learning Cycles');  
ylabel('Resubstitution Loss');
```



In general, as the number of decision trees in the trained classification ensemble increases, the resubstitution loss decreases.

A decrease in resubstitution loss might indicate that the software trained the ensemble sensibly. However, you cannot infer the predictive power of the ensemble by this decrease. To measure the predictive power of an ensemble, estimate the generalization error by:

- 1 Randomly partitioning the data into training and cross-validation sets. Do this by specifying `'holdout'`, `holdoutProportion` when you train the ensemble using `fitensemble`.

- 2 Passing the trained ensemble to `kfoldLoss`, which estimates the generalization error.

### Train Regression Ensemble

Use a trained, boosted regression tree ensemble to predict the fuel economy of a car. Choose the number of cylinders, volume displaced by the cylinders, horsepower, and weight as predictors. Then, train an ensemble using fewer predictors and compare its in-sample predictive accuracy against the first ensemble.

Load the `carsmall` data set. Store the training data in a table.

```
load carsmall
Tbl = table(Cylinders,Displacement,Horsepower,Weight,MPG);
```

Specify a regression tree template that uses surrogate splits to improve predictive accuracy in the presence of NaN values.

```
t = templateTree('Surrogate','On');
```

Train the regression tree ensemble using `LSBoost` and 100 learning cycles.

```
Mdl1 = fitensemble(Tbl,'MPG','LSBoost',100,t);
```

`Mdl1` is a trained `RegressionEnsemble` regression ensemble. Because `MPG` is a variable in the MATLAB® Workspace, you can obtain the same result by entering

```
Mdl1 = fitensemble(Tbl,MPG,'LSBoost',100,t);
```

Use the trained regression ensemble to predict the fuel economy for a four-cylinder car with a 200-cubic inch displacement, 150 horsepower, and weighing 3000 lbs.

```
predMPG = predict(Mdl1,[4 200 150 3000])
```

```
predMPG =  
  
    22.8462
```

The average fuel economy of a car with these specifications is 21.78 mpg.

Train a new ensemble using all predictors in `Tbl` except `Displacement`.

```
formula = 'MPG ~ Cylinders + Horsepower + Weight';
Mdl1 = fitensemble(Tbl, formula, 'LSBoost', 100, t);
```

Compare the resubstitution MSEs between Mdl1 and Mdl2.

```
mse1 = resubLoss(Mdl1)
mse2 = resubLoss(Mdl2)
```

```
mse1 =
    6.4721
```

```
mse2 =
    7.8599
```

The in-sample MSE for the ensemble that trains on all predictors is lower.

### Estimate the Generalization Error of a Boosting Ensemble

Estimate the generalization error of a trained, boosting classification ensemble of decision trees.

Load the ionosphere data set.

```
load ionosphere;
```

Train a decision tree ensemble using AdaBoostM1, 100 learning cycles, and half of the data chosen randomly. The software validates the algorithm using the remaining half.

```
rng(2); % For reproducibility
ClassTreeEns = fitensemble(X, Y, 'AdaBoostM1', 100, 'Tree', ...
    'Holdout', 0.5);
```

ClassTreeEns is a trained ClassificationEnsemble ensemble classifier.

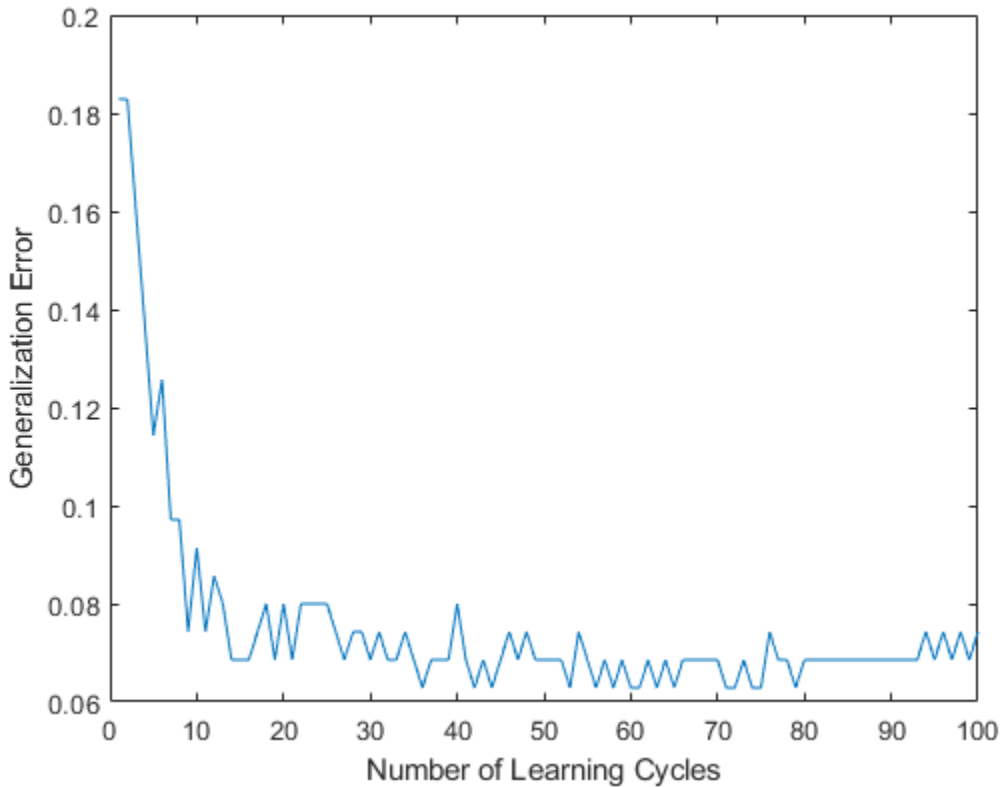
Determine the cumulative generalization error, i.e., the cumulative misclassification error of the labels in the validation data).

```
genError = kfoldLoss(ClassTreeEns, 'Mode', 'Cumulative');
```

`genError` is a 100-by-1 vector, where element  $k$  contains the generalization error after the first  $k$  learning cycles.

Plot the generalization error over the number of learning cycles.

```
plot(genError);  
xlabel('Number of Learning Cycles');  
ylabel('Generalization Error');
```



The cumulative generalization error decreases to approximately 7% when 25 weak learners compose the ensemble classifier.

## Find the Optimal Number of Splits and Trees for an Ensemble

You can control the depth of the trees in an ensemble of decision trees. You can also control the tree depth in an ECOC model containing decision tree binary learners using the `MaxNumSplits`, `MinLeafSize`, or `MinParentSize` name-value pair parameters.

- When bagging decision trees, `fitensemble` grows deep decision trees by default. You can grow shallower trees to reduce model complexity or computation time.
- When boosting decision trees, `fitensemble` grows stumps (a tree with one split) by default. You can grow deeper trees for better accuracy.

Load the `carsmall` data set. Specify the variables `Acceleration`, `Displacement`, `Horsepower`, and `Weight` as predictors, and `MPG` as the response.

```
load carsmall
X = [Acceleration Displacement Horsepower Weight];
Y = MPG;
```

The default values of the tree depth controllers for boosting regression trees are:

- 1 for `MaxNumSplits`. This option grows stumps.
- 5 for `MinLeafSize`
- 10 for `MinParentSize`

To search for the optimal number of splits:

- 1 Train a set of ensembles. Exponentially increase the maximum number of splits for subsequent ensembles from stump to at most  $n - 1$  splits, where  $n$  is the training sample size. Also, decrease the learning rate for each ensemble from 1 to 0.1.
- 2 Cross validate the ensembles.
- 3 Estimate the cross-validated mean-squared error (MSE) for each ensemble.
- 4 Compare the cross-validated MSEs. The ensemble with the lowest one performs the best, and indicates the optimal maximum number of splits, number of trees, and learning rate for the data set.

Grow and cross validate a deep regression tree and a stump. Specify to use surrogate splits because the data contain missing values. These serve as benchmarks.

```
MdlDeep = fitrtree(X,Y,'CrossVal','on','MergeLeaves','off',...
    'MinParentSize',1,'Surrogate','on');
MdlStump = fitrtree(X,Y,'MaxNumSplits',1,'CrossVal','on','Surrogate','on');
```

Train the boosting ensembles using 150 regression trees. Cross validate the ensemble using 5-fold cross validation. Vary the maximum number of splits using the values in the sequence  $\{2^0, 2^1, \dots, 2^m\}$ , where  $m$  is such that  $2^m$  is no greater than  $n - 1$ , where  $n$  is the training sample size. For each variant, adjust the learning rate to each value in the set  $\{0.1, 0.25, 0.5, 1\}$ ;

```
n = size(X,1);
m = floor(log2(n - 1));
lr = [0.1 0.25 0.5 1];
maxNumSplits = 2.^(0:m);
numTrees = 150;
Mdl = cell(numel(maxNumSplits),numel(lr));
rng(1); % For reproducibility
for k = 1:numel(lr);
    for j = 1:numel(maxNumSplits);
        t = templateTree('MaxNumSplits',maxNumSplits(j),'Surrogate','on');
        Mdl{j,k} = fitensemble(X,Y,'LSBoost',numTrees,t,...
            'Type','regression','KFold',5,'LearnRate',lr(k));
    end;
end;
```

Compute the cross-validated MSE for each ensemble.

```
kf1All = @(x) kfoldLoss(x,'Mode','cumulative');
errorCell = cellfun(kf1All,Mdl,'Uniform',false);
error = reshape(cell2mat(errorCell),[numTrees numel(maxNumSplits) numel(lr)]);
errorDeep = kfoldLoss(MdlDeep);
errorStump = kfoldLoss(MdlStump);
```

Plot how the cross-validated MSE behaves as the number of trees in the ensemble increases for a few of the ensembles, the deep tree, and the stump. Plot the curves with respect to learning rate in the same plot, and plot separate plots for varying tree complexities. Choose a subset of tree complexity levels.

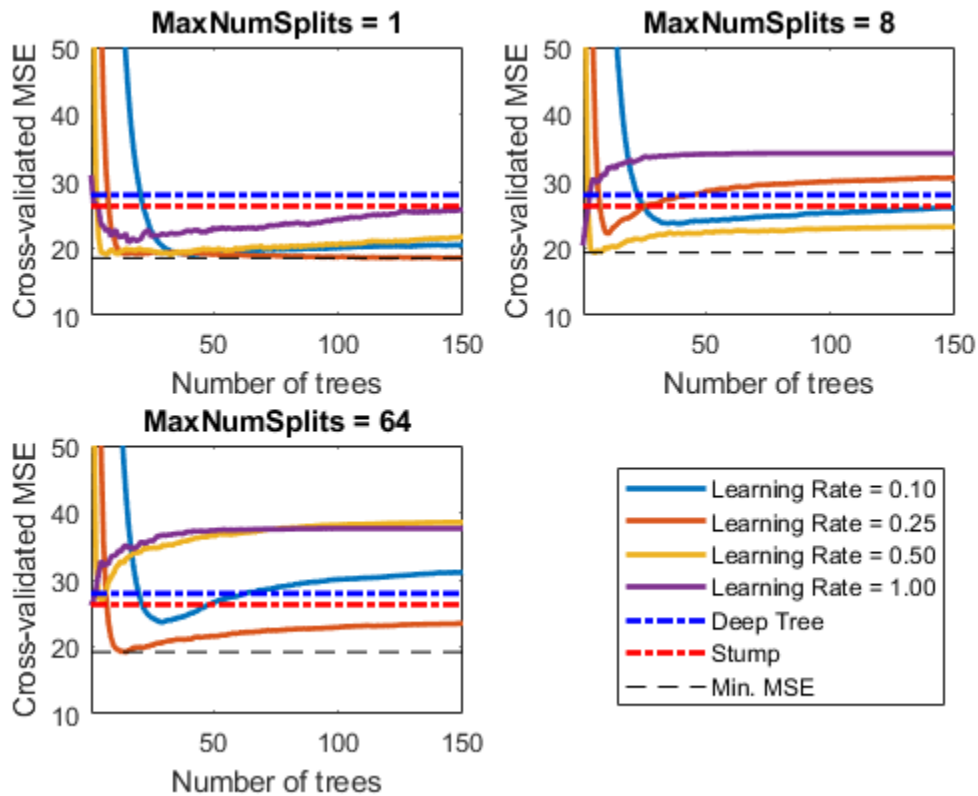
```
mnsPlot = [1 round(numel(maxNumSplits)/2) numel(maxNumSplits)];
figure;
for k = 1:3;
    subplot(2,2,k);
    plot(squeeze(error(:,mnsPlot(k),:)), 'LineWidth',2);
    axis tight;
    hold on;
    h = gca;
    plot(h.XLim,[errorDeep errorDeep], '-.b', 'LineWidth',2);
    plot(h.XLim,[errorStump errorStump], '-.r', 'LineWidth',2);
```



```

plot(h.XLim,min(min(error(:,mnsPlot(k),:)).*[1 1], '--k');
h.YLim = [10 50];
xlabel 'Number of trees';
ylabel 'Cross-validated MSE';
title(sprintf('MaxNumSplits = %0.3g', maxNumSplits(mnsPlot(k))));
hold off;
end;
hL = legend([cellstr(num2str(lr','Learning Rate = %0.2f'));...
            'Deep Tree'; 'Stump'; 'Min. MSE']);
hL.Position(1) = 0.6;

```



Each curve contains a minimum cross-validated MSE occurring at the optimal number of trees in the ensemble.

Identify the maximum number of splits, number of trees, and learning rate that yields the lowest MSE overall.

```
[minErr,minErrIdxLin] = min(error(:));  
[idxNumTrees,idxMNS,idxLR] = ind2sub(size(error),minErrIdxLin);  
  
fprintf('\nMin. MSE = %0.5f',minErr)  
fprintf('\nOptimal Parameter Values:\nNum. Trees = %d',idxNumTrees);  
fprintf('\nMaxNumSplits = %d\nLearning Rate = %0.2f\n',...  
        maxNumSplits(idxMNS),lr(idxLR))
```

```
Min. MSE = 18.42979  
Optimal Parameter Values:  
Num. Trees = 1  
MaxNumSplits = 4  
Learning Rate = 1.00
```

For a different approach to optimizing this ensemble, see “Optimize a Boosted Regression Ensemble” on page 10-75.

- “Supervised Learning Workflow and Algorithms” on page 18-2
- “Framework for Ensemble Learning” on page 18-41

## Input Arguments

### **Tbl** — Sample data

table

Sample data used to train the model, specified as a table. Each row of `Tbl` corresponds to one observation, and each column corresponds to one predictor variable. `Tbl` can contain one additional column for the response variable. Multi-column variables and cell arrays other than cell arrays of character vectors are not allowed.

- If `Tbl` contains the response variable and you want to use all remaining variables as predictors, then specify the response variable using `ResponseVarName`.
- If `Tbl` contains the response variable, and you want to use a subset of the remaining variables only as predictors, then specify a formula using `formula`.
- If `Tbl` does not contain the response variable, then specify the response data using `Y`. The length of response variable and the number of rows of `Tbl` must be equal.

---

**Note** To save memory and execution time, supply  $X$  and  $Y$  instead of `Tbl`.

---

Data Types: `table`

**ResponseVarName** — Response variable name

name of response variable in `Tbl`

Response variable name, specified as the name of the response variable in `Tbl`.

You must specify `ResponseVarName` as a character vector. For example, if `Tbl.Y` is the response variable, then specify `ResponseVarName` as `'Y'`. Otherwise, `fitensemble` treats all columns of `Tbl` as predictor variables.

The response variable must be a categorical or character array, logical or numeric vector, or cell array of character vectors. If the response variable is a character array, then each element must correspond to one row of the array.

For classification, you can specify the order of the classes using the `ClassNames` name-value pair argument. Otherwise, `fitensemble` determines the class order, and stores it in the `Mdl.ClassNames`.

Data Types: `char`

**formula** — Explanatory model of response and subset of predictor variables

character vector

Explanatory model of the response and a subset of the predictor variables, specified as a character vector in the form of `'Y~X1+X2+X3'`. In this form,  $Y$  represents the response variable, and  $X_1$ ,  $X_2$ , and  $X_3$  represent the predictor variables. The variables must be variable names in `Tbl` (`Tbl.Properties.VariableNames`).

To specify a subset of variables in `Tbl` as predictors for training the model, use a formula. If you specify a formula, then the software does not use any variables in `Tbl` that do not appear in `formula`.

Data Types: `char`

**x** — Predictor data

numeric matrix

Predictor data, specified as numeric matrix.

Each row corresponds to one observation, and each column corresponds to one predictor variable.

The length of  $Y$  and the number of rows of  $X$  must be equal.

To specify the names of the predictors in the order of their appearance in  $X$ , use the `PredictorNames` name-value pair argument.

Data Types: `single` | `double`

### **$Y$ — Response data**

categorical array | character array | logical vector | numeric vector | cell array of character vectors

Response data, specified as a categorical or character array, logical or numeric vector, or cell array of character vectors. Each entry in  $Y$  is the response to or label for the observation in the corresponding row of  $X$  or `Tbl`. The length of  $Y$  and the number of rows of  $X$  or `Tbl` must be equal. If the response variable is a character array, then each element must correspond to one row of the array.

- For classification,  $Y$  can be any of the supported data types. You can specify the order of the classes using the `ClassNames` name-value pair argument. Otherwise, `fitensemble` determines the class order, and stores it in the `Mdl.ClassNames`.
- For regression,  $Y$  must be a numeric column vector.

Data Types: `char` | `cell` | `categorical` | `logical` | `single` | `double`

### **Method — Ensemble-aggregation method**

character vector

Ensemble-aggregation method, specified as one of the method names in this list.

- For classification with two classes:
  - `'AdaBoostM1'`
  - `'LogitBoost'`
  - `'GentleBoost'`
  - `'RobustBoost'` (requires Optimization Toolbox)
  - `'LPBoost'` (requires Optimization Toolbox)

- 'TotalBoost' (requires Optimization Toolbox)
- 'RUSBoost'
- 'Subspace'
- 'Bag'
- For classification with three or more classes:
  - 'AdaBoostM2'
  - 'LPBoost' (requires Optimization Toolbox)
  - 'TotalBoost' (requires Optimization Toolbox)
  - 'RUSBoost'
  - 'Subspace'
  - 'Bag'
- For regression:
  - 'LSBoost'
  - 'Bag'

Because you can specify 'Bag' for classification and regression problems, specify the problem type using the `Type` name-value pair argument.

Data Types: `char`

### **NLearn** — Number of ensemble learning cycles

positive integer | 'AllPredictorCombinations'

Number of ensemble learning cycles, specified as a positive integer or 'AllPredictorCombinations'.

- If you specify a positive integer, then, at every learning cycle, the software trains one weak learner for every template object in `Learners`. Consequently, the software trains `NLearn* numel(Learners)` learners.
- If you specify 'AllPredictorCombinations', then set `Method` to 'Subspace' and specify one learner only in `Learners`. With these settings, the software trains learners for all possible combinations of predictors taken `NPredToSample` at a time. Consequently, the software trains `nchoosek(size(X,2), NPredToSample)` learners.

The software composes the ensemble using all trained learners and stores them in `Mdl.Trained`.

For more details, see “Tips” on page 32-2478.

Data Types: `single` | `double` | `char`

### Learners — Weak learners to use in ensemble

weak-learner name | weak-learner template object | cell vector of weak-learner template objects

Weak learners to use in the ensemble, specified as a weak-learner name, weak-learner template object, or cell array of weak-learner template objects.

Weak Learner	Weak-Learner Name	Template Object Creation Function	Method Settings
Discriminant analysis	'Discriminant'	templateDiscriminant	Recommended for 'Subspace'
$k$ nearest neighbors	'KNN'	templateKNN	For 'Subspace' only
Decision tree	'Tree'	templateTree	All methods except 'Subspace'

For more details, see `NLearn` and “Tips” on page 32-2478.

Example: For an ensemble composed of two types of classification trees, supply `{t1 t2}`, where `t1` and `t2` are classification tree templates.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'CrossVal', 'on', 'LearningRate', 0.05` specifies to implement 10-fold cross-validation and to use 0.05 as the learning rate.

### General Ensemble Options

#### CategoricalPredictors — Categorical predictors list

vector of positive integers | logical vector | character matrix | cell array of character vectors | 'all'

Categorical predictors list, specified as the comma-separated pair consisting of 'CategoricalPredictors' and one of these values:

Value	Description
Vector of positive integers	An entry in the vector is the index value corresponding to the column of the predictor data (X or Tbl) that contains a categorical variable.
Logical vector	A true entry means that the corresponding column of predictor data (X or Tbl) is a categorical variable.
Character matrix	Each row of the matrix is the name of a predictor variable. The names must match the entries in PredictorNames. Pad the names with extra blanks so each row of the character matrix has the same length.
Cell array of character vectors	Each element in the array is the name of a predictor variable. The names must match the entries in PredictorNames.
'all'	All predictors are categorical.

Specification of CategoricalPredictors is appropriate if:

- 'Learners' is 'tree'.
- 'Learners' is 'knn' when all predictors are categorical.

By default, if the predictor data is in a table (Tbl), fitensemble assumes that a variable is categorical if it contains logical values, categorical values, or a cell array of character vectors. If the predictor data is a matrix (X), fitensemble assumes all predictors are continuous. To identify any categorical predictors when the data is a matrix, use the 'CategoricalPredictors' name-value pair argument.

Example: 'CategoricalPredictors', 'all'

Data Types: single | double | logical | char | cell

#### **NPrint** — Printout frequency

'off' (default) | positive integer

Printout frequency, specified as the comma-separated pair consisting of 'NPrint' and a positive integer or 'off'.

To track the number of *weak learners* or *folds* that fitensemble trained so far, specify a positive integer. That is, if you specify the positive integer *m*:

- Without also specifying any cross-validation option (for example, `CrossVal`), then `fitensemble` displays a message to the command line every time it completes training  $m$  weak learners.
- And a cross-validation option, then `fitensemble` displays a message to the command line every time it finishes training  $m$  folds.

If you specify `'off'`, then `fitensemble` does not display a message when it completes training weak learners.

---

**Tip** When training an ensemble of many weak learners on a large data set, specify a positive integer for `NPrint`.

---

Example: `'NPrint', 5`

Data Types: `single` | `double` | `char`

### **PredictorNames** — Predictor variable names

cell array of unique character vectors

Predictor variable names, specified as the comma-separated pair consisting of `'PredictorNames'` and a cell array of unique character vectors. The functionality of `'PredictorNames'` depends on the way you supply the training data.

- If you supply `X` and `Y`, then you can use `'PredictorNames'` to give the predictor variables in `X` names.
  - The order of the names in `PredictorNames` must correspond to the column order of `X`. That is, `PredictorNames{1}` is the name of `X(:, 1)`, `PredictorNames{2}` is the name of `X(:, 2)`, and so on. Also, `size(X, 2)` and `numel(PredictorNames)` must be equal.
  - By default, `PredictorNames` is `{'x1', 'x2', ...}`.
- If you supply `Tbl`, then you can use `'PredictorNames'` to choose which predictor variables to use in training. That is, `fitensemble` uses the predictor variables in `PredictorNames` and the response only in training.
  - `PredictorNames` must be a subset of `Tbl.Properties.VariableNames` and cannot include the name of the response variable.



- By default, `PredictorNames` contains the names of all predictor variables.
- It good practice to specify the predictors for training using one of `'PredictorNames'` or formula only.

Example: `'PredictorNames',`  
`{'SepalLength', 'SepalWidth', 'PedalLength', 'PedalWidth'}`

Data Types: `cell`

### **ResponseName — Response variable name**

`'Y'` (default) | character vector

Response variable name, specified as the comma-separated pair consisting of `'ResponseName'` and a character vector.

- If you supply `Y`, then you can use `'ResponseName'` to specify a name for the response variable.
- If you supply `ResponseVarName` or formula, then you cannot use `'ResponseName'`.

Example: `'ResponseName', 'response'`

Data Types: `char`

### **Type — Supervised learning type**

`'classification'` | `'regression'`

Supervised learning type, specified as the comma-separated pair consisting of `'Type'` and `'classification'` or `'regression'`.

- If `Method` is `'bag'`, then the supervised learning type is ambiguous. Therefore, specify `Type` when bagging.
- Otherwise, the value of `Method` determines the supervised learning type.

Example: `'Type', 'classification'`

Data Types: `char`

### **Cross-Validation Options**

#### **CrossVal — Cross-validation flag**

`'off'` (default) | `'on'`

Cross-validation flag, specified as the comma-separated pair consisting of 'Crossval' and 'on' or 'off'.

If you specify 'on', then the software implements 10-fold cross-validation.

To override this cross-validation setting, use one of these name-value pair arguments: `CVPartition`, `Holdout`, `KFold`, or `Leaveout`. To create a cross-validated model, you can use one cross-validation name-value pair argument at a time only.

Alternatively, cross-validate later by passing `Mdl` to `crossval` or `crossval`.

Example: 'Crossval', 'on'

Data Types: char

### **CVPartition — Cross-validation partition**

[] (default) | `cvpartition` partition object

Cross-validation partition, specified as the comma-separated pair consisting of 'CVPartition' and a `cvpartition` partition object as created by `cvpartition`. The partition object specifies the type of cross-validation, and also the indexing for training and validation sets.

To create a cross-validated model, you can use one of these four name-value pair arguments only: `CVPartition`, `Holdout`, `KFold`, or `Leaveout`.

Example: Suppose you create a random partition for 5-fold cross-validation on 500 observations by using `cvp = cvpartition(500, 'KFold', 5)`. Then, you can specify the cross-validation partition for `fitcecoc` as 'CVPartition', `cvp`.

### **Holdout — Fraction of data for holdout validation**

scalar value in the range (0,1)

Fraction of data used for holdout validation, specified as the comma-separated pair consisting of 'Holdout' and a scalar value in the range (0,1). If you specify 'Holdout',  $p$ , then the software:

- 1 Randomly reserves  $p*100\%$  of the data as validation data, and trains the model using the rest of the data
- 2 Stores the compact, trained model in the `Trained` property of the cross-validated model.

To create a cross-validated model, you can use one of these four name-value pair arguments only: `CVPartition`, `Holdout`, `KFold`, or `Leaveout`.

Example: `'Holdout', 0.1`

Data Types: `double` | `single`

### **KFold** — Number of folds

10 (default) | positive integer value greater than 1

Number of folds to use in a cross-validated model, specified as the comma-separated pair consisting of `'KFold'` and a positive integer value greater than 1. If you specify, e.g., `'KFold', k`, then the software:

- 1 Randomly partitions the data into  $k$  sets
- 2 For each set, reserves the set as validation data, and trains the model using the other  $k - 1$  sets
- 3 Stores the  $k$  compact, trained models in the cells of a  $k$ -by-1 cell vector in the `Trained` property of the cross-validated model.

To create a cross-validated model, you can use one of these four name-value pair arguments only: `CVPartition`, `Holdout`, `KFold`, or `Leaveout`.

Example: `'KFold', 5`

Data Types: `single` | `double`

### **Leaveout** — Leave-one-out cross-validation flag

`'off'` (default) | `'on'`

Leave-one-out cross-validation flag, specified as the comma-separated pair consisting of `'Leaveout'` and `'on'` or `'off'`. If you specify `'Leaveout', 'on'`, then, for each of the  $n$  observations, where  $n$  is `size(Mdl.X, 1)`, the software:

- 1 Reserves the observation as validation data, and trains the model using the other  $n - 1$  observations
- 2 Stores the  $n$  compact, trained models in the cells of an  $n$ -by-1 cell vector in the `Trained` property of the cross-validated model.

To create a cross-validated model, you can use one of these four name-value pair arguments only: `CVPartition`, `Holdout`, `KFold`, or `Leaveout`.

Example: 'Leaveout', 'on'

Data Types: char

### Other Classification or Regression Options

#### **ClassNames** — Names of classes to use for training

categorical array | cell array of character vectors | character array | logical vector | numeric vector

Names of classes to use for training, specified as the comma-separated pair consisting of 'ClassNames' and a categorical or character array, logical or numeric vector, or cell array of character vectors. `ClassNames` must be the same data type as `Y`.

If `ClassNames` is a character array, then each element must correspond to one *row* of the array.

Use `ClassNames` to:

- Order the classes during training.
- Specify the order of any input or output argument dimension that corresponds to the class order. For example, use `ClassNames` to specify the order of the dimensions of `Cost` or the column order of classification scores returned by `predict`.
- Select a subset of classes for training. For example, suppose that the set of all distinct class names in `Y` is {'a', 'b', 'c'}. To train the model using observations from classes 'a' and 'c' only, specify 'ClassNames', {'a', 'c'}.

The default is the set of all distinct class names in `Y`.

Example: 'ClassNames', {'b', 'g'}

Data Types: categorical | char | logical | single | double | cell

#### **Cost** — Misclassification cost

square matrix | structure array

Misclassification cost, specified as the comma-separated pair consisting of 'Cost' and a square matrix or structure. If you specify:

- The square matrix `Cost`, then `Cost(i, j)` is the cost of classifying a point into class `j` if its true class is `i`. That is, the rows correspond to the true class and the columns

correspond to the predicted class. To specify the class order for the corresponding rows and columns of `Cost`, also specify the `ClassNames` name-value pair argument.

- The structure `S`, then it must have two fields:
  - `S.ClassNames`, which contains the class names as a variable of the same data type as `Y`
  - `S.ClassificationCosts`, which contains the cost matrix with rows and columns ordered as in `S.ClassNames`

The default is  $\text{ones}(K) - \text{eye}(K)$ , where  $K$  is the number of distinct classes.

---

**Note** `fitensemble` uses `Cost` to adjust the prior class probabilities specified in `Prior`. Then, `fitensemble` uses the adjusted prior probabilities for training and resets the cost matrix to its default.

---

Example: `'Cost',[0 1 2 ; 1 0 2; 2 2 0]`

Data Types: `double` | `single` | `struct`

### **Prior — Prior probabilities**

`'empirical'` (default) | `'uniform'` | numeric vector | structure array

Prior probabilities for each class, specified as the comma-separated pair consisting of `'Prior'` and a value in this table.

Value	Description
<code>'empirical'</code>	The class prior probabilities are the class relative frequencies in <code>Y</code> .
<code>'uniform'</code>	All class prior probabilities are equal to $1/K$ , where $K$ is the number of classes.
numeric vector	Each element is a class prior probability. Order the elements according to <code>SVMModel.ClassNames</code> or specify the order using the <code>ClassNames</code> name-value pair argument. The software normalizes the elements such that they sum to 1.

Value	Description
structure array	<p>A structure <i>S</i> with two fields:</p> <ul style="list-style-type: none"> <li>• <i>S.ClassNames</i> contains the class names as a variable of the same type as <i>Y</i>.</li> <li>• <i>S.ClassProbs</i> contains a vector of corresponding prior probabilities. The software normalizes the elements such that they sum to 1.</li> </ul>

`fitensemble` normalizes the prior probabilities in `Prior` to sum to 1.

Example: `struct('ClassNames',  
{'setosa','versicolor','virginica'}), 'ClassProbs', 1:3)`

Data Types: `char` | `double` | `single` | `struct`

### **Weights — Observation weights**

numeric vector of positive values | character vector

Observation weights, specified as the comma-separated pair consisting of `'Weights'` and a numeric vector of positive values or name of a variable in `Tbl`. The software weighs the observations in each row of `X` or `Tbl` with the corresponding value in `Weights`. The size of `Weights` must equal the number of rows of `X` or `Tbl`.

If you specify the input data as a table `Tbl`, then `Weights` can be the name of a variable in `Tbl` that contains a numeric vector. In this case, you must specify `Weights` as a character vector. For example, if the weights vector `W` is stored as `Tbl.W`, then specify it as `'W'`. Otherwise, the software treats all columns of `Tbl`, including `W`, as predictors or the response when training the model.

The software normalizes `Weights` to sum up to the value of the prior probability in the respective class.

By default, `Weights` is `ones(n, 1)`, where  $n$  is the number of observations in `X` or `Tbl`.

Data Types: `double` | `single` | `char`

## Sampling Options for Boosting Methods and Bagging

### **FResample** — Fraction of training set to resample

1 (default) | positive scalar in (0,1]

Fraction of the training set to resample for every weak learner, specified as the comma-separated pair consisting of 'FResample' and a positive scalar in (0,1].

To use 'FResample', specify 'bag' for Method or set Resample to 'on'.

Example: 'FResample', 0.75

Data Types: single | double

### **Replace** — Flag indicating to sample with replacement

'on' (default) | 'off'

Flag indicating sampling with replacement, specified as the comma-separated pair consisting of 'Replace' and 'off' or 'on'.

- For 'on', the software samples the training observations with replacement.
- For 'off', the software samples the training observations without replacement. If you set Resample to 'on', then the software samples training observations assuming uniform weights. If you also specify a boosting method, then the software boosts by reweighting observations.

Unless you set Method to 'bag' or set Resample to 'on', Replace has no effect.

Example: 'Replace', 'off'

Data Types: char

### **Resample** — Flag indicating to resample

'off' | 'on'

Flag indicating to resample, specified as the comma-separated pair consisting of 'Resample' and 'off' or 'on'.

- If Method is any boosting method, then:
  - 'Resample', 'on' specifies to sample training observations using updated weights as the multinomial sampling probabilities.

- 'Resample', 'off' specifies to reweight observations at every learning iteration. This setting is the default.
- If Method is 'bag', then 'Resample' must be 'on'. The software resamples a fraction of the training observations (see `FResample`) with or without replacement (see `Replace`).

### **AdaBoostM1, AdaBoostM2, LogitBoost, GentleBoost, and LSBoost Method Options**

#### **LearnRate — Learning rate for shrinkage**

1 (default) | numeric scalar in (0,1]

Learning rate for shrinkage, specified as the comma-separated pair consisting of a numeric scalar in the interval (0,1].

To train an ensemble using shrinkage, set `LearnRate` to a value less than 1, for example, 0.1 is a popular choice. Training an ensemble using shrinkage requires more learning iterations, but often achieves better accuracy.

Example: 'LearnRate', 0.1

Data Types: single | double

### **RUSBoost Method Options**

#### **LearnRate — Learning rate for shrinkage**

1 (default) | numeric scalar in (0,1]

Learning rate for shrinkage, specified as the comma-separated pair consisting of a numeric scalar in the interval (0,1].

To train an ensemble using shrinkage, set `LearnRate` to a value less than 1, for example, 0.1 is a popular choice. Training an ensemble using shrinkage requires more learning iterations, but often achieves better accuracy.

Example: 'LearnRate', 0.1

Data Types: single | double

#### **RatioToSmallest — Sampling proportion with respect to lowest-represented class**

positive numeric scalar | numeric vector of positive values

Sampling proportion with respect to the lowest-represented class, specified as the comma-separated pair consisting of 'RatioToSmallest' and a numeric scalar or



numeric vector of positive values with length equal to the number of distinct classes in the training data.

Suppose that there are  $K$  classes in the training data and the lowest-represented class has  $m$  observations in the training data.

- If you specify the positive numeric scalar  $s$ , then `fitensemble` samples  $s*m$  observations from each class, that is, it uses the same sampling proportion for each class. For more details, see “Algorithms” on page 32-2479.
- If you specify the numeric vector  $[s_1, s_2, \dots, s_K]$ , then `fitensemble` samples  $s_i*m$  observations from class  $i$ ,  $i = 1, \dots, K$ . The elements of `RatioToSmallest` correspond to the order of the class names specified using `ClassNames` (see “Tips” on page 32-2478).

The default value is `ones(K, 1)`, which specifies to sample  $m$  observations from each class.

Example: `'RatioToSmallest', [2, 1]`

Data Types: `single` | `double`

### **LPBoost and TotalBoost Method Options**

#### **MarginPrecision** — Margin precision to control convergence speed

0.1 (default) | numeric scalar in [0,1]

Margin precision to control convergence speed, specified as the comma-separated pair consisting of `'MarginPrecision'` and a numeric scalar in the interval [0,1].

`MarginPrecision` affects the number of boosting iterations required for convergence.

---

**Tip** To train an ensemble using many learners, specify a small value for `MarginPrecision`. For training using a few learners, specify a large value.

---

Example: `'MarginPrecision', 0.5`

Data Types: `single` | `double`

### **RobustBoost Method Options**

#### **RobustErrorGoal** — Target classification error

0.1 (default) | nonnegative numeric scalar

Target classification error, specified as the comma-separated pair consisting of 'RobustErrorGoal' and a nonnegative numeric scalar. The upper bound on possible values depends on the values of RobustMarginSigma and RobustMaxMargin. However, the upper bound cannot exceed 1.

---

**Tip** For a particular training set, usually there is an optimal range for RobustErrorGoal. If you set it too low or too high, then the software can produce a model with poor classification accuracy. Try cross-validating to search for the appropriate value.

---

Example: 'RobustErrorGoal',0.05

Data Types: single | double

**RobustMarginSigma — Classification margin distribution spread**

0.1 (default) | positive numeric scalar

Classification margin distribution spread over the training data, specified as the comma-separated pair consisting of 'RobustMarginSigma' and a positive numeric scalar. Before specifying RobustMarginSigma, consult the literature on RobustBoost, for example, [19].

Example: 'RobustMarginSigma',0.5

Data Types: single | double

**RobustMaxMargin — Maximal classification margin**

0 (default) | nonnegative numeric scalar

Maximal classification margin in the training data, specified as the comma-separated pair consisting of 'RobustMaxMargin' and a nonnegative numeric scalar. The software minimizes the number of observations in the training data having classification margins below RobustMaxMargin.

Example: 'RobustMaxMargin',1

Data Types: single | double

**Random Subspace Method Options**

**NPredToSample — Number of predictors to sample**

1 (default) | positive integer

Number of predictors to sample for each random subspace learner, specified as the comma-separated pair consisting of 'NPredToSample' and a positive integer in the interval  $1, \dots, p$ , where  $p$  is the number of predictor variables (`size(X, 2)` or `size(Tbl, 2)`).

Data Types: single | double

## Output Arguments

### **mdl** — Trained ensemble model

ClassificationBaggedEnsemble model object | ClassificationEnsemble model object | ClassificationPartitionedEnsemble cross-validated model object | RegressionBaggedEnsemble model object | RegressionEnsemble model object | RegressionPartitionedEnsemble cross-validated model object

Trained ensemble model, returned as one of the model objects in this table.

Model Object	Type Setting	Specify Any Cross-Validation Options?	Method Setting	Resample Setting
ClassificationBaggedEnsemble	'classification'	No	'Bag'	'on'
ClassificationEnsemble	'classification'	No	Any ensemble-aggregation method for classification	'off'
ClassificationPartitionedEnsemble	'classification'	Yes	Any classification ensemble-aggregation method	'off' or 'on'
RegressionBaggedEnsemble	'regression'	No	'Bag'	'on'
RegressionEnsemble	'regression'	No	'LSBoost'	'off'
RegressionPartitionedEnsemble	'regression'	Yes	'LSBoost' or 'Bag'	'off' or 'on'

The name-value pair arguments that control cross-validation are `CrossVal`, `Holdout`, `KFold`, `Leaveout`, and `CVPartition`.

To reference properties of `Mdl`, use dot notation. For example, to access or display the cell vector of weak learner model objects for an ensemble that has not been cross-validated, enter `Mdl.Trained` at the command line.

## Tips

- `NLearn` can vary from a few dozen to a few thousand. Usually, an ensemble with good predictive power requires from a few hundred to a few thousand weak learners. However, you do not have to train an ensemble for that many cycles at once. You can start by growing a few dozen learners, inspect the ensemble performance and then, if necessary, train more weak learners using `resume` for classification problems, or `resume` for regression problems.
- Ensemble performance depends on the ensemble setting and the setting of the weak learners. That is, if you specify weak learners with default parameters, then the ensemble can perform poorly. Therefore, like ensemble settings, it is good practice to adjust the parameters of the weak learners using templates, and to choose values that minimize generalization error.
- If you specify to resample using `Resample`, then it is good practice to resample to entire data set. That is, use the default setting of 1 for `FResample`.
- In classification problems (that is, `Type` is 'classification'):
  - If the ensemble-aggregation method (`Method`) is 'bag' and:
    - The misclassification cost (`Cost`) is highly imbalanced, then, for in-bag samples, the software oversamples unique observations from the class that has a large penalty.
    - The class prior probabilities (`Prior`) are highly skewed, the software oversamples unique observations from the class that has a large prior probability.

For smaller sample sizes, these combinations can result in a low relative frequency of out-of-bag observations from the class that has a large penalty or prior probability. Consequently, the estimated out-of-bag error is highly variable and it can be difficult to interpret. To avoid large estimated out-of-bag error variances,

particularly for small sample sizes, set a more balanced misclassification cost matrix using `Cost` or a less skewed prior probability vector using `Prior`.

- Because the order of some input and output arguments correspond to the distinct classes in the training data, it is good practice to specify the class order using the `ClassNames` name-value pair argument.
- To determine the class order quickly, remove all observations from the training data that are unclassified (that is, have a missing label), obtain and display an array of all the distinct classes, and then specify the array for `ClassNames`. For example, suppose the response variable ( $Y$ ) is a cell array of labels. This code specifies the class order in the variable `classNames`.

```
Ycat = categorical(Y);
classNames = categories(Ycat)
```

`categorical` assigns `<undefined>` to unclassified observations and `categories` excludes `<undefined>` from its output. Therefore, if you use this code for cell arrays of labels or similar code for categorical arrays, then you do not have to remove observations with missing labels to obtain a list of the distinct classes.

- To specify that the class order from lowest-represented label to most-represented, then quickly determine the class order (as in the previous bullet), but arrange the classes in the list by frequency before passing the list to `ClassNames`. Following from the previous example, this code specifies the class order from lowest- to most-represented in `classNamesLH`.

```
Ycat = categorical(Y);
classNames = categories(Ycat);
freq = countcats(Ycat);
[~,idx] = sort(freq);
classNamesLH = classNames(idx);
```

## Algorithms

- For details of ensemble-aggregation algorithms, see “Ensemble Algorithms” on page 18-51.
- If you specify `Method` to be a boosting algorithm and `Learners` to be decision trees, then the software grows stumps by default. A decision stump is one root node connected to two terminal, leaf nodes. You can adjust tree depth by specifying the

MaxNumSplits, MinLeafSize, and MinParentSize name-value pair arguments using `templateTree`.

- `fitensemble` generates in-bag samples by oversampling classes with large misclassification costs and undersampling classes with small misclassification costs. Consequently, out-of-bag samples have fewer observations from classes with large misclassification costs and more observations from classes with small misclassification costs. If you train a classification ensemble using a small data set and a highly skewed cost matrix, then the number of out-of-bag observations per class can be low. Therefore, the estimated out-of-bag error can have a large variance and can be difficult to interpret. The same phenomenon can occur for classes with large prior probabilities.
- For the RUSBoost ensemble-aggregation method (Method), the name-value pair argument `RatioToSmallest` specifies the sampling proportion for each class with respect to the lowest-represented class. For example, suppose that there are two classes in the training data: *A* and *B*. *A* have 100 observations and *B* have 10 observations. and that the lowest-represented class has *m* observations in the training data.
  - If you set `'RatioToSmallest', 2`, then  $s*m = 2*10 = 20$ . Consequently, `fitensemble` trains every learner using 20 observations from class *A* and 20 observations from class *B*. If you set `'RatioToSmallest', [2 2]`, then you obtain the same result.
  - If you set `'RatioToSmallest', [2, 1]`, then  $s1*m = 2*10 = 20$  and  $s2*m = 1*10 = 10$ . Consequently, `fitensemble` trains every learner using 20 observations from class *A* and 10 observations from class *B*.
- For ensembles of decision trees, and for dual-core systems and above, `fitensemble` parallelizes training using Intel Threading Building Blocks (TBB). For details on Intel TBB, see <https://software.intel.com/en-us/intel-tbb>.

## References

- [1] Breiman, L. “Bagging Predictors.” *Machine Learning*. Vol. 26, pp. 123–140, 1996.
- [2] Breiman, L. “Random Forests.” *Machine Learning*. Vol. 45, pp. 5–32, 2001.
- [3] Freund, Y. “A more robust boosting algorithm.” *arXiv:0905.2138v1*, 2009.

- [4] Freund, Y. and R. E. Schapire. “A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting.” *J. of Computer and System Sciences*, Vol. 55, pp. 119–139, 1997.
- [5] Friedman, J. “Greedy function approximation: A gradient boosting machine.” *Annals of Statistics*, Vol. 29, No. 5, pp. 1189–1232, 2001.
- [6] Friedman, J., T. Hastie, and R. Tibshirani. “Additive logistic regression: A statistical view of boosting.” *Annals of Statistics*, Vol. 28, No. 2, pp. 337–407, 2000.
- [7] Hastie, T., R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning* section edition, Springer, New York, 2008.
- [8] Ho, T. K. “The random subspace method for constructing decision forests.” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 20, No. 8, pp. 832–844, 1998.
- [9] Schapire, R. E., Y. Freund, P. Bartlett, and W.S. Lee. “Boosting the margin: A new explanation for the effectiveness of voting methods.” *Annals of Statistics*, Vol. 26, No. 5, pp. 1651–1686, 1998.
- [10] Seiffert, C., T. Khoshgoftaar, J. Hulse, and A. Napolitano. “RUSBoost: Improving classification performance when training data is skewed.” *19th International Conference on Pattern Recognition*, pp. 1–4, 2008.
- [11] Warmuth, M., J. Liao, and G. Ratsch. “Totally corrective boosting algorithms that maximize the margin.” *Proc. 23rd Int’l. Conf. on Machine Learning, ACM*, New York, pp. 1001–1008, 2006.

## See Also

ClassificationBaggedEnsemble | ClassificationEnsemble |  
ClassificationPartitionedEnsemble | RegressionBaggedEnsemble |  
RegressionEnsemble | RegressionPartitionedEnsemble |  
templateDiscriminant | templateKNN | templateTree

## Topics

“Supervised Learning Workflow and Algorithms” on page 18-2  
“Framework for Ensemble Learning” on page 18-41

**Introduced in R2011a**



# fitnlm

Fit nonlinear regression model

## Syntax

```
mdl = fitnlm(tbl,modelfun,beta0)
mdl = fitnlm(X,y,modelfun,beta0)
mdl = fitnlm( ____,modelfun,beta0,Name,Value)
```

## Description

`mdl = fitnlm(tbl,modelfun,beta0)` fits the model specified by `modelfun` to variables in the table or dataset array `tbl`, and returns the nonlinear model `mdl`.

`fitnlm` estimates model coefficients using an iterative procedure starting from the initial values in `beta0`.

`mdl = fitnlm(X,y,modelfun,beta0)` fits a nonlinear regression model using the column vector `y` as a response variable and the columns of the matrix `X` as predictor variables.

`mdl = fitnlm( ____,modelfun,beta0,Name,Value)` fits a nonlinear regression model with additional options specified by one or more `Name, Value` pair arguments.

## Examples

### Nonlinear Model from Table

Create a nonlinear model for auto mileage based on the `carbig` data.

Load the data and create a nonlinear model.

```
load carbig
tbl = table(Horsepower,Weight,MPG);
```

```
modelfun = @(b,x)b(1) + b(2)*x(:,1).^b(3) + ...  
    b(4)*x(:,2).^b(5);  
beta0 = [-50 500 -1 500 -1];  
mdl = fitnlm(tbl,modelfun,beta0)
```

```
mdl =
```

```
Nonlinear regression model:  
MPG ~ b1 + b2*Horsepower^b3 + b4*Weight^b5
```

```
Estimated Coefficients:  
            Estimate          SE          tStat          pValue  
            _____          _____          _____          _____  
b1          -49.383          119.97          -0.41164          0.68083  
b2           376.43          567.05           0.66384          0.50719  
b3          -0.78193          0.47168          -1.6578          0.098177  
b4           422.37          776.02           0.54428          0.58656  
b5          -0.24127          0.48325          -0.49926          0.61788
```

```
Number of observations: 392, Error degrees of freedom: 387  
Root Mean Squared Error: 3.96  
R-Squared: 0.745, Adjusted R-Squared 0.743  
F-statistic vs. constant model: 283, p-value = 1.79e-113
```

## Nonlinear Model from Matrix Data

Create a nonlinear model for auto mileage based on the carbig data.

Load the data and create a nonlinear model.

```
load carbig  
X = [Horsepower,Weight];  
y = MPG;  
modelfun = @(b,x)b(1) + b(2)*x(:,1).^b(3) + ...  
    b(4)*x(:,2).^b(5);  
beta0 = [-50 500 -1 500 -1];  
mdl = fitnlm(X,y,modelfun,beta0)
```

```
mdl =
```

Nonlinear regression model:  
 $y \sim b_1 + b_2 * x_1^{b_3} + b_4 * x_2^{b_5}$

Estimated	Coefficients:			
	Estimate	SE	tStat	pValue
b1	-49.383	119.97	-0.41164	0.68083
b2	376.43	567.05	0.66384	0.50719
b3	-0.78193	0.47168	-1.6578	0.098177
b4	422.37	776.02	0.54428	0.58656
b5	-0.24127	0.48325	-0.49926	0.61788

Number of observations: 392, Error degrees of freedom: 387  
 Root Mean Squared Error: 3.96  
 R-Squared: 0.745, Adjusted R-Squared 0.743  
 F-statistic vs. constant model: 283, p-value = 1.79e-113

## Adjust Fitting Options in Nonlinear Model

Create a nonlinear model for auto mileage based on the `carbig` data. Strive for more accuracy by lowering the `TolFun` option, and observe the iterations by setting the `Display` option.

Load the data and create a nonlinear model.

```
load carbig
X = [Horsepower,Weight];
y = MPG;
modelfun = @(b,x)b(1) + b(2)*x(:,1).^b(3) + ...
    b(4)*x(:,2).^b(5);
beta0 = [-50 500 -1 500 -1];
```

Create options to lower `TolFun` and to report iterative display, and create a model using the options.

```
opts = statset('Display','iter','TolFun',1e-10);
mdl = fitnlm(X,y,modelfun,beta0,'Options',opts);
```

---

Iteration	SSE	Norm of Gradient	Norm of Step
0	1.82248e+06		
1	678600	788810	1691.07
2	616716	6.12739e+06	45.4738
3	249831	3.9532e+06	293.557
4	17675	361544	369.284
5	11746.6	69670.5	169.079
6	7242.22	343738	394.822
7	6250.32	159719	452.941
8	6172.87	91622.9	268.674
9	6077	6957.44	100.208
10	6076.34	6370.39	88.1905
11	6075.75	5199.08	77.9694
12	6075.3	4646.61	69.764
13	6074.91	4235.96	62.9114
14	6074.55	3885.28	57.0647
15	6074.23	3571.1	52.0036
16	6073.93	3286.48	47.5795
17	6073.66	3028.34	43.6844
18	6073.4	2794.31	40.2352
19	6073.17	2582.15	37.1663
20	6072.95	2389.68	34.4243
21	6072.74	2214.84	31.965
22	6072.55	2055.78	29.7516
23	6072.37	1910.83	27.753
24	6072.21	1778.51	25.9428
25	6072.05	1657.5	24.2986
26	6071.9	1546.65	22.8011
27	6071.76	1444.93	21.4338
28	6071.63	1351.44	20.1822
29	6071.51	1265.39	19.0339
30	6071.39	1186.06	17.978
31	6071.28	1112.83	17.0052
32	6071.17	1045.13	16.107
33	6071.07	982.465	15.2762
34	6070.98	924.389	14.5063
35	6070.89	870.498	13.7916
36	6070.8	820.434	13.127
37	6070.72	773.872	12.5081
38	6070.64	730.521	11.9307
39	6070.57	690.117	11.3914

---

40	6070.5	652.422	10.887
41	6070.43	617.219	10.4144
42	6070.37	584.315	9.97115
43	6070.31	553.53	9.55489
44	6070.25	524.703	9.1635
45	6070.19	497.686	8.79506
46	6070.14	472.345	8.44785
47	6070.08	448.557	8.12028
48	6070.03	426.21	7.81091
49	6069.99	405.201	7.51845
50	6069.94	385.435	7.2417
51	6069.9	366.825	6.97956
52	6069.85	349.293	6.73104
53	6069.81	332.764	6.49523
54	6069.77	317.171	6.27127
55	6069.74	302.452	6.0584
56	6069.7	288.55	5.85591
57	6069.66	275.411	5.66315
58	6069.63	262.986	5.47949
59	6069.6	251.23	5.3044
60	6069.57	240.1	5.13734
61	6069.54	229.558	4.97784
62	6069.51	219.567	4.82545
63	6069.48	210.094	4.67977
64	6069.45	201.108	4.5404
65	6069.43	192.578	4.407
66	6069.4	184.479	4.27923
67	6069.38	176.785	4.15677
68	6069.35	169.472	4.03935
69	6069.33	162.518	3.9267
70	6069.31	155.903	3.81855
71	6069.29	149.608	3.71468
72	6069.26	143.615	3.61486
73	6069.24	137.907	3.5189
74	6069.22	132.468	3.42658
75	6069.21	127.283	3.33774
76	6069.19	122.339	3.25221
77	6069.17	117.623	3.16981
78	6069.15	113.123	3.09041
79	6069.14	108.827	3.01386
80	6069.12	104.725	2.94002
81	6069.1	100.806	2.86877
82	6069.09	97.0611	2.8
83	6069.07	93.4814	2.73358

84	6069.06	90.0584	2.66942
85	6069.05	86.7841	2.60741
86	6069.03	83.6513	2.54745
87	6069.02	80.6528	2.48947
88	6069.01	77.7821	2.43338
89	6068.99	75.0327	2.37908
90	6068.98	72.399	2.32652
91	6068.97	69.8752	2.27561
92	6068.96	67.4561	2.22629
93	6068.95	65.1366	2.17849
94	6068.94	62.9123	2.13216
95	6068.93	60.7784	2.08723
96	6068.92	58.7308	2.04364
97	6068.91	56.7655	2.00135
98	6068.9	54.8787	1.9603
99	6068.89	4349.28	18.1917
100	6068.77	2416.27	14.4439
101	6068.71	1721.26	12.1305
102	6068.66	1228.78	10.289
103	6068.63	884.002	8.82019
104	6068.6	639.615	7.62744
105	6068.58	464.84	6.64627
106	6068.56	338.878	5.82964
107	6068.55	247.508	5.14296
108	6068.54	180.878	4.56032
109	6068.53	132.084	4.06194
110	6068.52	96.2341	3.63255
111	6068.51	69.8362	3.26019
112	6068.51	50.3735	2.93541
113	6068.5	36.0205	2.65062
114	6068.5	25.4452	2.39969
115	6068.49	17.6693	2.17764
116	6068.49	1027.39	14.0164
117	6068.48	544.039	5.3137
118	6068.48	94.0576	2.86664
119	6068.48	113.636	3.73502
120	6068.48	0.518567	1.3705
121	6068.48	4.5944	0.91284
122	6068.48	1.56389	0.629322
123	6068.48	1.13809	0.432547
124	6068.48	0.295936	0.297509

Iterations terminated: relative change in SSE less than OPTIONS.TolFun

## Specify Nonlinear Regression Using Model Name Syntax

Specify a nonlinear regression model for estimation using a function handle or model syntax.

Load sample data.

```
S = load('reaction');
X = S.reactants;
y = S.rate;
beta0 = S.beta;
```

Use a function handle to specify the Hougen-Watson model for the rate data.

```
mdl = fitnlm(X,y,@hougen,beta0)
```

```
mdl =
```

```
Nonlinear regression model:
    y ~ hougen(b,X)
```

```
Estimated Coefficients:
```

	Estimate	SE	tStat	pValue
b1	1.2526	0.86701	1.4447	0.18654
b2	0.062776	0.043561	1.4411	0.18753
b3	0.040048	0.030885	1.2967	0.23089
b4	0.11242	0.075157	1.4957	0.17309
b5	1.1914	0.83671	1.4239	0.1923

```
Number of observations: 13, Error degrees of freedom: 8
```

```
Root Mean Squared Error: 0.193
```

```
R-Squared: 0.999, Adjusted R-Squared 0.998
```

```
F-statistic vs. zero model: 3.91e+03, p-value = 2.54e-13
```

Alternatively, you can use an expression to specify the Hougen-Watson model for the rate data.

```
myfun = 'y~(b1*x2-x3/b5)/(1+b2*x1+b3*x2+b4*x3)';
mdl2 = fitnlm(X,y,myfun,beta0)
```

```
mdl2 =
```

```
Nonlinear regression model:
```

```
y ~ (b1*x2 - x3/b5)/(1 + b2*x1 + b3*x2 + b4*x3)
```

```
Estimated Coefficients:
```

	Estimate	SE	tStat	pValue
b1	1.2526	0.86701	1.4447	0.18654
b2	0.062776	0.043561	1.4411	0.18753
b3	0.040048	0.030885	1.2967	0.23089
b4	0.11242	0.075157	1.4957	0.17309
b5	1.1914	0.83671	1.4239	0.1923

```
Number of observations: 13, Error degrees of freedom: 8
```

```
Root Mean Squared Error: 0.193
```

```
R-Squared: 0.999, Adjusted R-Squared 0.998
```

```
F-statistic vs. zero model: 3.91e+03, p-value = 2.54e-13
```

### Estimate Nonlinear Regression Using Robust Fitting Options

Generate sample data from the nonlinear regression model

$$y = b_1 + b_2 \exp(-b_3 x) + \varepsilon,$$

where  $b_1$ ,  $b_2$ , and  $b_3$  are coefficients, and the error term is normally distributed with mean 0 and standard deviation 0.5.

```
modelfun = @(b,x) (b(1)+b(2)*exp(-b(3)*x));
```

```
rng('default') % for reproducibility
```

```
b = [1;3;2];
```

```
x = exprnd(2,100,1);
```

```
y = modelfun(b,x) + normrnd(0,0.5,100,1);
```

Set robust fitting options.

```
opts = statset('nlinfit');
```

```
opts.RobustWgtFun = 'bisquare';
```



Fit the nonlinear model using the robust fitting options. Here, use an expression to specify the model.

```
b0 = [2;2;2];
modelstr = 'y ~ b1 + b2*exp(-b3*x)';

mdl = fitnlm(x,y,modelstr,b0,'Options',opts)
```

```
mdl =
```

```
Nonlinear regression model (robust fit):
    y ~ b1 + b2*exp( - b3*x)
```

```
Estimated Coefficients:
```

	Estimate	SE	tStat	pValue
b1	1.0218	0.07202	14.188	2.1344e-25
b2	3.6619	0.25429	14.401	7.974e-26
b3	2.9732	0.38496	7.7232	1.0346e-11

```
Number of observations: 100, Error degrees of freedom: 97
Root Mean Squared Error: 0.501
R-Squared: 0.807, Adjusted R-Squared 0.803
F-statistic vs. constant model: 203, p-value = 2.34e-35
```

## Fit Nonlinear Regression Model Using Weights Function Handle

Load sample data.

```
S = load('reaction');
X = S.reactants;
y = S.rate;
beta0 = S.beta;
```

Specify a function handle for observation weights. The function accepts the model fitted values as input, and returns a vector of weights.

```
a = 1; b = 1;
weights = @(yhat) 1./((a + b*abs(yhat)).^2);
```

Fit the Hougen-Watson model to the rate data using the specified observation weights function.

```
mdl = fitnlm(X,y,@hougen,beta0,'Weights',weights)
```

```
mdl =
```

```
Nonlinear regression model:
```

```
  y ~ hougen(b,X)
```

```
Estimated Coefficients:
```

	Estimate	SE	tStat	pValue
b1	0.83085	0.58224	1.427	0.19142
b2	0.04095	0.029663	1.3805	0.20477
b3	0.025063	0.019673	1.274	0.23842
b4	0.080053	0.057812	1.3847	0.20353
b5	1.8261	1.281	1.4256	0.19183

```
Number of observations: 13, Error degrees of freedom: 8
```

```
Root Mean Squared Error: 0.037
```

```
R-Squared: 0.998, Adjusted R-Squared 0.998
```

```
F-statistic vs. zero model: 1.14e+03, p-value = 3.49e-11
```

### Nonlinear Regression Model Using Nonconstant Error Model

Load sample data.

```
S = load('reaction');  
X = S.reactants;  
y = S.rate;  
beta0 = S.beta;
```

Fit the Hougen-Watson model to the rate data using the combined error variance model.

```
mdl = fitnlm(X,y,@hougen,beta0,'ErrorModel','combined')
```

```
mdl =
```

Nonlinear regression model:  
 $y \sim \text{hougen}(b, X)$

Estimated Coefficients:

	Estimate	SE	tStat	pValue
b1	1.2526	0.86702	1.4447	0.18654
b2	0.062776	0.043561	1.4411	0.18753
b3	0.040048	0.030885	1.2967	0.23089
b4	0.11242	0.075158	1.4957	0.17309
b5	1.1914	0.83671	1.4239	0.1923

Number of observations: 13, Error degrees of freedom: 8  
 Root Mean Squared Error: 1.27  
 R-Squared: 0.999, Adjusted R-Squared 0.998  
 F-statistic vs. zero model: 3.91e+03, p-value = 2.54e-13

- “Examine Quality and Adjust the Fitted Nonlinear Model” on page 13-7
- “Predict or Simulate Responses Using a Nonlinear Model” on page 13-10
- “Nonlinear Regression Workflow” on page 13-15

## Input Arguments

### **tbl** — Input data

table | dataset array

Input data, specified as a table or dataset array. If you do not specify the predictor and response variables, the last variable is the response variable and the others are the predictor variables by default.

Predictor variables and response variable must be numeric.

You specify the response and predictor names in your model specification. If you do not specify a model, you can set a different column as the response variable by using the `ResponseVar` name-value pair argument. You can select a subset of the columns as predictors by using the `PredictorVars` name-value pair argument.

Data Types: table

**x — Predictor variables**

matrix

Predictor variables, specified as an  $n$ -by- $p$  matrix, where  $n$  is the number of observations and  $p$  is the number of predictor variables. Each column of  $X$  represents one variable, and each row represents one observation.

Data Types: `single` | `double` | `logical`**y — Response variable**

vector

Response variable, specified as an  $n$ -by-1 vector, where  $n$  is the number of observations. Each entry in  $y$  is the response for the corresponding row of  $X$ .

Data Types: `single` | `double` | `logical`**modelfun — Functional form of the model**function handle | character vector of the form ' $y \sim f(b_1, b_2, \dots, b_j, x_1, x_2, \dots, x_k)$ '

Functional form of the model, specified as either of the following.

- Function handle `@modelfun` or `@(b,x) modelfun`, where
  - $b$  is a coefficient vector with the same number of elements as `beta0`.
  - $x$  is a matrix with the same number of columns as  $X$  or the number of predictor variable columns of `tbl`.

`modelfun(b,x)` returns a column vector that contains the same number of rows as  $x$ . Each row of the vector is the result of evaluating `modelfun` on the corresponding row of  $x$ . In other words, `modelfun` is a vectorized function, one that operates on all data rows and returns all evaluations in one function call. `modelfun` should return real numbers to obtain meaningful coefficients.

- Character vector of the form ' $y \sim f(b_1, b_2, \dots, b_j, x_1, x_2, \dots, x_k)$ ', where  $f$  represents a scalar function of the scalar coefficient variables  $b_1, \dots, b_j$  and the scalar data variables  $x_1, \dots, x_k$ .

**beta0 — Coefficients**

numeric vector

Coefficients for the nonlinear model, specified as a numeric vector. `NonLinearModel` starts its search for optimal coefficients from `beta0`.

Data Types: `single` | `double`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'ErrorModel', 'combined', 'Exclude', 2, 'Options', opt` specifies the error model as the combined model, excludes the second observation from the fit, and uses the options defined in the structure `opt` to control the iterative fitting procedure.

### CoefficientNames — Names of the model coefficients

`{'b1', 'b2', ..., 'bk'}` (default) | cell array of character vectors

Names of the model coefficients, specified as a cell array of character vectors.

Data Types: `char`

### ErrorModel — Form of the error variance model

`'constant'` (default) | `'proportional'` | `'combined'`

Form of the error variance model, specified as one of the following. Each model defines the error using a standard mean-zero and unit-variance variable  $e$  in combination with independent components: the function value  $f$ , and one or two parameters  $a$  and  $b$

<code>'constant'</code> (default)	$y = f + ae$
<code>'proportional'</code>	$y = f + bfe$
<code>'combined'</code>	$y = f + (a + b f )e$

The only allowed error model when using `Weights` is `'constant'`.

---

**Note** `options.RobustWgtFun` must have value `[]` when using an error model other than `'constant'`.

---

Example: `'ErrorModel', 'proportional'`

**ErrorParameters** — Initial estimates of the error model parameters

numeric array

Initial estimates of the error model parameters for the chosen `ErrorModel`, specified as a numeric array.

Error Model	Parameters	Default Values
'constant'	$a$	1
'proportional'	$b$	1
'combined'	$a, b$	[1, 1]

You can only use the 'constant' error model when using `Weights`.

---

**Note** `options.RobustWgtFun` must have value `[]` when using an error model other than 'constant'.

---

For example, if 'ErrorModel' has the value 'combined', you can specify the starting value 1 for  $a$  and the starting value 2 for  $b$  as follows.

Example: `'ErrorParameters', [1, 2]`

Data Types: `single` | `double`

**Exclude** — Observations to exclude

logical or numeric index vector

Observations to exclude from the fit, specified as the comma-separated pair consisting of 'Exclude' and a logical or numeric index vector indicating which observations to exclude from the fit.

For example, you can exclude observations 2 and 3 out of 6 using either of the following examples.

Example: `'Exclude', [2, 3]`

Example: `'Exclude', logical([0 1 1 0 0 0])`

Data Types: `single` | `double` | `logical`

**Options** — Options for controlling the iterative fitting procedure

`[]` (default) | `structure`

Options for controlling the iterative fitting procedure, specified as a structure created by `statset`. The relevant fields are the nonempty fields in the structure returned by the call `statset('fitnlm')`.

Option	Meaning	Default
DerivStep	Relative difference used in finite difference derivative calculations. A positive scalar, or a vector of positive scalars the same size as the vector of parameters estimated by the Statistics and Machine Learning Toolbox function using the options structure.	$\text{eps}^{(1/3)}$
Display	Amount of information displayed by the fitting algorithm. <ul style="list-style-type: none"> <li>• 'off' — Displays no information.</li> <li>• 'final' — Displays the final output.</li> <li>• 'iter' — Displays iterative output to the Command Window.</li> </ul>	'off'
FunValCheck	Character vector indicating to check for invalid values, such as NaN or Inf, from the model function.	'on'
MaxIter	Maximum number of iterations allowed. Positive integer.	200
RobustWgtFun	Weight function for robust fitting. Can also be a function handle that accepts a normalized residual as input and returns the robust weights as output. If you use a function handle, give a Tune constant. See “Robust Options” on page 32-2500	[]
Tune	Tuning constant used in robust fitting to normalize the residuals before applying the weight function. A positive scalar. Required if the weight function is specified as a function handle.	See “Robust Options” on page 32-2500 for the default, which depends on RobustWgtFun.
TolFun	Termination tolerance for the objective function value. Positive scalar.	1e-8

Option	Meaning	Default
TolX	Termination tolerance for the parameters. Positive scalar.	1e-8

Data Types: struct

### **PredictorVars** — Predictor variables

cell array of character vectors | logical or numeric index vector

Predictor variables to use in the fit, specified as the comma-separated pair consisting of 'PredictorVars' and either a cell array of character vectors of the variable names in the table or dataset array `tbl`, or a logical or numeric index vector indicating which columns are predictor variables.

The character vectors should be among the names in `tbl`, or the names you specify using the 'VarNames' name-value pair argument.

The default is all variables in `X`, or all variables in `tbl` except for `ResponseVar`.

For example, you can specify the second and third variables as the predictor variables using either of the following examples.

Example: 'PredictorVars', [2,3]

Example: 'PredictorVars', logical([0 1 1 0 0 0])

Data Types: single | double | logical | cell

### **ResponseVar** — Response variable

last column of `tbl` (default) | variable name | logical or numeric index vector

Response variable to use in the fit, specified as the comma-separated pair consisting of 'ResponseVar' and either a variable name in the table or dataset array `tbl`, or a logical or numeric index vector indicating which column is the response variable.

If you specify a model, it specifies the response variable. Otherwise, when fitting a table or dataset array, 'ResponseVar' indicates which variable `fitnlm` should use as the response.

For example, you can specify the fourth variable, say `yield`, as the response out of six variables, in one of the following ways.

Example: 'ResponseVar', 'yield'



Example: `'ResponseVar',[4]`

Example: `'ResponseVar',logical([0 0 0 1 0 0])`

Data Types: `single | double | logical | char`

### **VarNames** — Names of variables in fit

`{'x1','x2',..., 'xn','y'}` (default) | cell array of character vectors

Names of variables in fit, specified as the comma-separated pair consisting of `'VarNames'` and a cell array of character vectors including the names for the columns of  $X$  first, and the name for the response variable  $y$  last.

`'VarNames'` is not applicable to variables in a table or dataset array, because those variables already have names.

For example, if in your data, horsepower, acceleration, and model year of the cars are the predictor variables, and miles per gallon (MPG) is the response variable, then you can name the variables as follows.

Example: `'VarNames',{'Horsepower','Acceleration','Model_Year','MPG'}`

Data Types: `cell`

### **Weights** — Observation weights

`ones(n,1)` (default) | vector of nonnegative scalar values | function handle

Observation weights, specified as a vector of nonnegative scalar values or function handle.

- If you specify a vector, then it must have  $n$  elements, where  $n$  is the number of rows in `tbl` or `y`.
- If you specify a function handle, then the function must accept a vector of predicted response values as input, and return a vector of real positive weights as output.

Given weights, `W`, `NonLinearModel` estimates the error variance at observation  $i$  by  $MSE \cdot (1/W(i))$ , where `MSE` is the mean squared error.

Data Types: `single | double | function_handle`

## Output Arguments

### `mdl` — Nonlinear model

`NonLinearModel` object

Nonlinear model representing a least-squares fit of the response to the data, returned as a `NonLinearModel` object.

If the `Options` structure contains a nonempty `RobustWgtFun` field, the model is not a least-squares fit, but uses the `RobustWgtFun` robust fitting function.

For properties and methods of the nonlinear model object, `mdl`, see the `NonLinearModel` class page.

## Definitions

### Robust Options

Weight Function	Equation	Default Tuning Constant
'andrews'	$w = (\text{abs}(r) < \pi) .* \sin(r) ./ r$	1.339
'bisquare' (default)	$w = (\text{abs}(r) < 1) .* (1 - r.^2).^2$	4.685
'cauchy'	$w = 1 ./ (1 + r.^2)$	2.385
'fair'	$w = 1 ./ (1 + \text{abs}(r))$	1.400
'huber'	$w = 1 ./ \max(1, \text{abs}(r))$	1.345
'logistic'	$w = \tanh(r) ./ r$	1.205
'talwar'	$w = 1 * (\text{abs}(r) < 1)$	2.795
'welsch'	$w = \exp(-(r.^2))$	2.985
[]	No robust fitting	—

## Algorithms

`fitnlm` uses the same fitting algorithm as `nlinfit`.

## References

- [1] Seber, G. A. F., and C. J. Wild. *Nonlinear Regression*. Hoboken, NJ: Wiley-Interscience, 2003.
- [2] DuMouchel, W. H., and F. L. O'Brien. "Integrating a Robust Option into a Multiple Regression Computing Environment." *Computer Science and Statistics: Proceedings of the 21st Symposium on the Interface*. Alexandria, VA: American Statistical Association, 1989.
- [3] Holland, P. W., and R. E. Welsch. "Robust Regression Using Iteratively Reweighted Least-Squares." *Communications in Statistics: Theory and Methods*, A6, 1977, pp. 813–827.

## See Also

NonLinearModel | nlinfit

## Topics

- "Examine Quality and Adjust the Fitted Nonlinear Model" on page 13-7
- "Predict or Simulate Responses Using a Nonlinear Model" on page 13-10
- "Nonlinear Regression Workflow" on page 13-15
- "Nonlinear Regression" on page 13-2

**Introduced in R2013b**

## LinearMixedModel.fitmatrix

**Class:** LinearMixedModel

Fit linear mixed-effects model using design matrices

---

**Note** `LinearMixedModel.fitmatrix` will be removed in a future release. Use `fitlmematrix` instead.

---

### Syntax

```
lme = LinearMixedModel.fitmatrix(X, y, Z, [])  
lme = LinearMixedModel.fitmatrix(X, y, Z, G)  
lme = LinearMixedModel.fitmatrix( ____, Name, Value)
```

### Description

`lme = LinearMixedModel.fitmatrix(X, y, Z, [])` creates a linear mixed-effects model of the responses `y` using the fixed-effects design matrix `X` and random-effects design matrix or matrices in `Z`.

`[]` implies that there is one group. That is, the grouping variable `G` is `ones(n, 1)`, where `n` is the number of observations. Using `LinearMixedModel.fitmatrix(X, Y, Z, [])` without a specified covariance pattern most likely will result in a non-identifiable model. This syntax is recommended only if you build the grouping information into the random effects design `Z` and specify a covariance pattern for the random effects using `'CovariancePattern'` name-value pair argument.

`lme = LinearMixedModel.fitmatrix(X, y, Z, G)` creates a linear mixed-effects model of the responses `y` using the fixed-effects design matrix `X` and random-effects design matrix `Z` or matrices in `Z`, and the grouping variable or variables in `G`.

`lme = LinearMixedModel.fitmatrix( ____, Name, Value)` also creates a linear mixed-effects model with additional options specified by one or more `Name, Value` pair arguments, using any of the previous input arguments.

For example, you can specify the names of the response, predictor, and grouping variables. You can also specify the covariance pattern, fitting method, or the optimization algorithm.

## Input Arguments

### **x** — Fixed-effects design matrix

*n*-by-*p* matrix

Fixed-effects design matrix, specified as an *n*-by-*p* matrix, where *n* is the number of observations, and *p* is the number of fixed-effects predictor variables. Each row of **X** corresponds to one observation, and each column of **X** corresponds to one variable.

Data Types: `single` | `double`

### **y** — Response values

*n*-by-1 vector

Response values, specified as an *n*-by-1 vector, where *n* is the number of observations.

Data Types: `single` | `double`

### **z** — Random-effects design

*n*-by-*q* matrix | cell array of *R* *n*-by-*q*(*r*) matrices, *r* = 1, 2, ..., *R*

Random-effects design, specified as either of the following.

- If there is one random-effects term in the model, then **Z** must be an *n*-by-*q* matrix, where *n* is the number of observations and *q* is the number of variables in the random-effects term.
- If there are *R* random-effects terms, then **Z** must be a cell array of length *R*. Each cell of **Z** contains an *n*-by-*q*(*r*) design matrix  $Z\{r\}$ , *r* = 1, 2, ..., *R*, corresponding to each random-effects term. Here, *q*(*r*) is the number of random effects term in the *r*th random effects design matrix,  $Z\{r\}$ .

Data Types: `single` | `double` | `cell`

### **g** — Grouping variable or variables

*n*-by-1 vector | cell array of *R* *n*-by-1 vectors

Grouping variable or variables on page 2-59, specified as either of the following.

- If there is one random-effects term, then  $G$  must be an  $n$ -by-1 vector corresponding to a single grouping variable with  $M$  levels or groups.

$G$  can be a categorical vector, numeric vector, character array, or cell array of character vectors.

- If there are multiple random-effects terms, then  $G$  must be a cell array of length  $R$ . Each cell of  $G$  contains a grouping variable  $G\{r\}$ ,  $r = 1, 2, \dots, R$ , with  $M(r)$  levels.

$G\{r\}$  can be a categorical vector, numeric vector, character array, or cell array of character vectors.

Data Types: `single` | `double` | `char` | `cell`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### **FixedEffectPredictors** — Names of columns in fixed-effects design matrix

`{'x1', 'x2', ..., 'xP'}` (default) | cell array of length  $p$

Names of columns in the fixed-effects design matrix  $X$ , specified as the comma-separated pair consisting of `'FixedEffectPredictors'` and a cell array of length  $p$ .

For example, if you have a constant term and two predictors, say `TimeSpent` and `Gender`, where `Female` is the reference level for `Gender`, as the fixed effects, then you can specify the names of your fixed effects in the following way. `Gender_Male` represents the dummy variable you must create for category `Male`. You can choose different names for these variables.

Example: `'FixedEffectPredictors',`  
`{'Intercept', 'TimeSpent', 'Gender_Male'},`

Data Types: `cell`

### **RandomEffectPredictors** — Names of columns in random-effects design matrix or cell array

cell array of length  $q$  | cell array of length  $R$  with elements of length  $q(r)$ ,  $r = 1, 2, \dots, R$

Names of columns in the random-effects design matrix or cell array `Z`, specified as the comma-separated pair consisting of `'RandomEffectPredictors'` and either of the following:

- A cell array of length  $q$  when `Z` is an  $n$ -by- $q$  design matrix. In this case, the default is `{'z1', 'z2', ..., 'zQ'}`.
- A cell array of length  $R$ , when `Z` is a cell array of length  $R$  with each element `Z{r}` of length  $q(r)$ ,  $r = 1, 2, \dots, R$ . In this case, the default is `{'z11', 'z12', ..., 'z1Q(1)'}, ..., {'zr1', 'zr2', ..., 'zrQ(r)'}`.

For example, suppose you have correlated random effects for intercept and a variable named `Acceleration`. Then, you can specify the random-effects predictor names as follows.

Example: `'RandomEffectPredictors', {'Intercept', 'Acceleration'}`

If you have two random effects terms, one for the intercept and the variable `Acceleration` grouped by variable `g1`, and the second for the intercept, grouped by the variable `g2`, then you specify the random-effects predictor names as follows.

Example: `'RandomEffectPredictors', {'Intercept', 'Acceleration'}, {'Intercept'}`

Data Types: `cell`

### **ResponseVarName** — Name of response variable

`'y'` (default) | character vector

Name of response variable, specified as the comma-separated pair consisting of `'ResponseVarName'` and a character vector.

For example, if your response variable name is `score`, then you can specify it as follows.

Example: `'ResponseVarName', 'score'`

Data Types: `char`

### **RandomEffectGroups** — Names of random effects grouping variables

`'g'` or `{'g1', 'g2', ..., 'gR'}` (default) | character vector | cell array of character vectors

Names of random effects grouping variables, specified as the comma-separated pair `'RandomEffectGroups'` and either of the following:

- Character vector — If there is only one random-effects term, that is, if  $G$  is a vector, then the value of 'RandomEffectGroups' is a character vector containing the name for the grouping variable  $G$ . The default is 'g'.
- Cell array of character vectors — If there are multiple random-effects terms, that is, if  $G$  is a cell array of length  $R$ , then the value of 'RandomEffectGroups' is a cell array of length  $R$ , where each cell contains the name for the grouping variable  $G\{r\}$ . The default is {'g1', 'g2', ..., 'gR'}.

For example, if you have two random-effects terms,  $z_1$  and  $z_2$ , grouped by the grouping variables `sex` and `subject`, then you can specify the names of your grouping variables as follows.

```
Example: 'RandomEffectGroups', {'sex', 'subject'}
```

Data Types: char | cell

### **CovariancePattern** — Pattern of covariance matrix

'FullCholesky' (default) | character vector | square symmetric logical matrix | cell array of character vectors or logical matrices

Pattern of the covariance matrix of the random effects, specified as the comma-separated pair consisting of 'CovariancePattern' and a character vector, a square symmetric logical matrix, or a cell array of character vectors or logical matrices.

If there are  $R$  random-effects terms, then the value of 'CovariancePattern' must be a cell array of length  $R$ , where each element  $r$  of this cell array specifies the pattern of the covariance matrix of the random-effects vector associated with the  $r$ th random-effects term. The options for each element follow.

'FullCholesky'

Default. Full covariance matrix using the Cholesky parameterization. `fitlme` estimates all elements of the covariance matrix.

'Full'

Full covariance matrix, using the log-Cholesky parameterization. `fitlme` estimates all elements of the covariance matrix.



'Diagonal'

Diagonal covariance matrix. That is, off-diagonal elements of the covariance matrix are constrained to be 0.

$$\begin{pmatrix} \sigma_{b1}^2 & 0 & 0 \\ 0 & \sigma_{b2}^2 & 0 \\ 0 & 0 & \sigma_{b3}^2 \end{pmatrix}$$

'Isotropic'

Diagonal covariance matrix with equal variances. That is, off-diagonal elements of the covariance matrix are constrained to be 0, and the diagonal elements are constrained to be equal. For example, if there are three random-effects terms with an isotropic covariance structure, this covariance matrix looks like

$$\begin{pmatrix} \sigma_b^2 & 0 & 0 \\ 0 & \sigma_b^2 & 0 \\ 0 & 0 & \sigma_b^2 \end{pmatrix}$$

where  $\sigma_b^2$  is the common variance of the random-effects terms.

`'CompSymm'`

Compound symmetry structure. That is, common variance along diagonals and equal correlation between all random effects. For example, if there are three random-effects terms with a covariance matrix having a compound symmetry structure, this covariance matrix looks like

$$\begin{pmatrix} \sigma_{b1}^2 & \sigma_{b1,b2} & \sigma_{b1,b2} \\ \sigma_{b1,b2} & \sigma_{b1}^2 & \sigma_{b1,b2} \\ \sigma_{b1,b2} & \sigma_{b1,b2} & \sigma_{b1}^2 \end{pmatrix}$$

where  $\sigma_{b1}^2$  is the common variance of the random-effects terms and  $\sigma_{b1,b2}$  is the common covariance between any two random-effects term .

`PAT`

Square symmetric logical matrix. If `'CovariancePattern'` is defined by the matrix `PAT`, and if `PAT(a,b) = false`, then the `(a,b)` element of the corresponding covariance matrix is constrained to be 0.

Example: `'CovariancePattern', 'Diagonal'`

Example: `'CovariancePattern', {'Full', 'Diagonal'}`

### **FitMethod** — Method for estimating parameters

`'ML'` (default) | `'REML'`

Method for estimating parameters of the linear mixed-effects model, specified as the comma-separated pair consisting of `'FitMethod'` and either of the following.

`'ML'`

Default. Maximum likelihood estimation

`'REML'`

Restricted maximum likelihood estimation

Example: `'FitMethod', 'REML'`

### **weights** — Observation weights

vector of scalar values

Observation weights, specified as the comma-separated pair consisting of 'Weights' and a vector of length  $n$ , where  $n$  is the number of observations.

Data Types: single | double

**Exclude — Indices for rows to exclude**

use all rows without NaNs (default) | vector of integer or logical values

Indices for rows to exclude from the linear mixed-effects model in the data, specified as the comma-separated pair consisting of 'Exclude' and a vector of integer or logical values.

For example, you can exclude the 13th and 67th rows from the fit as follows.

Example: 'Exclude', [13, 67]

Data Types: single | double | logical

**DummyVarCoding — Coding to use for dummy variables**

'reference' (default) | 'effects' | 'full'

Coding to use for dummy variables created from the categorical variables, specified as the comma-separated pair consisting of 'DummyVarCoding' and one of the following.

Value	Description
'reference'	Default. Coefficient for first category set to 0.
'effects'	Coefficients sum to 0.
'full'	One dummy variable for each category.

Example: 'DummyVarCoding', 'effects'

**Optimizer — Optimization algorithm**

'quasinevton' (default) | 'fminunc'

Optimization algorithm, specified as the comma-separated pair consisting of 'Optimizer' and either of the following.

<code>'quasinevton'</code>	Default. Uses a trust region based quasi-Newton optimizer. Change the options of the algorithm using <code>statset('LinearMixedModel')</code> . If you don't specify the options, then <code>LinearMixedModel</code> uses the default options of <code>statset('LinearMixedModel')</code> .
<code>'fminunc'</code>	You must have Optimization Toolbox to specify this option. Change the options of the algorithm using <code>optimoptions('fminunc')</code> . If you don't specify the options, then <code>LinearMixedModel</code> uses the default options of <code>optimoptions('fminunc')</code> with <code>'Algorithm'</code> set to <code>'quasi-newton'</code> .

Example: `'Optimizer','fminunc'`

### **OptimizerOptions — Options for optimization algorithm**

structure returned by `statset` | object returned by `optimoptions`

Options for the optimization algorithm, specified as the comma-separated pair consisting of `'OptimizerOptions'` and a structure returned by `statset('LinearMixedModel')` or an object returned by `optimoptions('fminunc')`.

- If `'Optimizer'` is `'fminunc'`, then use `optimoptions('fminunc')` to change the options of the optimization algorithm. See `optimoptions` for the options `'fminunc'` uses. If `'Optimizer'` is `'fminunc'` and you do not supply `'OptimizerOptions'`, then the default for `LinearMixedModel` is the default options created by `optimoptions('fminunc')` with `'Algorithm'` set to `'quasi-newton'`.
- If `'Optimizer'` is `'quasinevton'`, then use `statset('LinearMixedModel')` to change the optimization parameters. If you don't change the optimization parameters, then `LinearMixedModel` uses the default options created by `statset('LinearMixedModel')`:

The 'quasinevton' optimizer uses the following fields in the structure created by `statset('LinearMixedModel')`.

**TolFun** — Relative tolerance on gradient of objective function

1e-6 (default) | positive scalar value

Relative tolerance on the gradient of the objective function, specified as a positive scalar value.

**TolX** — Absolute tolerance on step size

1e-12 (default) | positive scalar value

Absolute tolerance on the step size, specified as a positive scalar value.

**MaxIter** — Maximum number of iterations allowed

10000 (default) | positive scalar value

Maximum number of iterations allowed, specified as a positive scalar value.

**Display** — Level of display

'off' (default) | 'iter' | 'final'

Level of display, specified as one of 'off', 'iter', or 'final'.

**StartMethod** — Method to start iterative optimization

'default' (default) | 'random'

Method to start iterative optimization, specified as the comma-separated pair consisting of 'StartMethod' and either of the following.

Value	Description
'default'	An internally defined default value
'random'	A random initial value

Example: 'StartMethod', 'random'

**Verbose** — Indicator to display optimization process on screen

false (default) | true

Indicator to display the optimization process on screen, specified as the comma-separated pair consisting of 'Verbose' and either false or true. Default is false.

The setting for 'Verbose' overrides the field 'Display' in 'OptimizerOptions'.

Example: 'Verbose', true

### **CheckHessian** — Indicator to check positive definiteness of Hessian

false (default) | true

Indicator to check the positive definiteness of the Hessian of the objective function with respect to unconstrained parameters at convergence, specified as the comma-separated pair consisting of 'CheckHessian' and either false or true. Default is false.

Specify 'CheckHessian' as true to verify optimality of the solution or to determine if the model is overparameterized in the number of covariance parameters.

Example: 'CheckHessian', true

## Output Arguments

### **lme** — Linear mixed-effects model

LinearMixedModel object

Linear mixed-effects model, returned as a LinearMixedModel object.

## Examples

### **No Grouping Variable Specified**

Load the sample data.

```
load carsmall
```

Fit a linear mixed-effects model, where MPG is the response, weight is the predictor variable, and the intercept varies by model year. First, define the design matrices. Then, fit the model using the specified design matrices.

```
y = MPG;  
X = [ones(size(Weight)), Weight];  
Z = ones(size(y));  
lme = LinearMixedModel.fitmatrix(X,y,Z,Model_Year)
```

```
lme =
```

```
Linear mixed-effects model fit by ML
```

```
Model information:
```

Number of observations	94
Fixed effects coefficients	2
Random effects coefficients	3
Covariance parameters	2

```
Formula:
```

```
y ~ x1 + x2 + (z11 | g1)
```

```
Model fit statistics:
```

AIC	BIC	LogLikelihood	Deviance
486.09	496.26	-239.04	478.09

```
Fixed effects coefficients (95% CIs):
```

Name	Estimate	SE	tStat	DF	pValue
'x1'	43.575	2.3038	18.915	92	1.8371e-33
'x2'	-0.0067097	0.0004242	-15.817	92	5.5373e-28

Lower	Upper
39	48.151
-0.0075522	-0.0058672

```
Random effects covariance parameters (95% CIs):
```

```
Group: g1 (3 Levels)
```

Name1	Name2	Type	Estimate	Lower	Upper
'z11'	'z11'	'std'	3.301	1.4448	7.5421

```
Group: Error
```

Name	Estimate	Lower	Upper
'Res Std'	2.8997	2.5075	3.3532

Now fit the same model by building the grouping into the Z matrix.

```
Z = double([Model_Year==70, Model_Year==76, Model_Year==82]);
lme = LinearMixedModel.fitmatrix(X,y,Z,[],...
'Covariancepattern','Isotropic')
```

lme =

Linear mixed-effects model fit by ML

Model information:

Number of observations	94
Fixed effects coefficients	2
Random effects coefficients	3
Covariance parameters	2

Formula:

y ~ x1 + x2 + (z11 + z12 + z13 | g1)

Model fit statistics:

AIC	BIC	LogLikelihood	Deviance
486.09	496.26	-239.04	478.09

Fixed effects coefficients (95% CIs):

Name	Estimate	SE	tStat	DF	pValue
'x1'	43.575	2.3038	18.915	92	1.8371e-33
'x2'	-0.0067097	0.0004242	-15.817	92	5.5373e-28

Lower	Upper
39	48.151
-0.0075522	-0.0058672

Random effects covariance parameters (95% CIs):

Group: g1 (1 Levels)

Name1	Name2	Type	Estimate	Lower	Upper
'z11'	'z11'	'std'	3.301	1.4448	7.5421

Group: Error

Name	Estimate	Lower	Upper
'Res Std'	2.8997	2.5075	3.3532

### Longitudinal Study with Covariate

Load the sample data.



```
load(fullfile(matlabroot, 'examples', 'stats', 'weight.mat'));
```

weight contains data from a longitudinal study, where 20 subjects are randomly assigned 4 exercise programs (A, B, C, D) and their weight loss is recorded over six two-week time periods. This is simulated data.

Define Subject and Program as categorical variables. Create the design matrices for a linear mixed-effects model, with the initial weight, type of program, week, and the interaction between the week and type of program as the fixed effects. The intercept and coefficient of week vary by subject.

This model corresponds to

$$y_{im} = \beta_0 + \beta_1 IW_i + \beta_2 \text{Week}_i + \beta_3 I[\text{PB}]_i + \beta_4 I[\text{PC}]_i + \beta_5 I[\text{PD}]_i \\ + \beta_6 (\text{Week}_i * I[\text{PB}]_i) + \beta_7 (\text{Week}_i * I[\text{PC}]_i) + \beta_8 (\text{Week}_i * I[\text{PD}]_i) \\ + b_{0m} + b_{1m} \text{Week}_{im} + \varepsilon_{im},$$

where  $i = 1, 2, \dots, 120$ , and  $m = 1, 2, \dots, 20$ .  $\beta_j$  are the fixed-effects coefficients,  $j = 0, 1, \dots, 8$ , and  $b_{0m}$  and  $b_{1m}$  are random effects. IW stands for initial weight and  $I[\cdot]$  is a dummy variable representing a type of program. For example,  $I[\text{PB}]_i$  is the dummy variable representing program type B. The random effects and observation error have the following prior distributions:

$$b_{0m} \sim N(0, \sigma_0^2)$$

$$b_{1m} \sim N(0, \sigma_1^2)$$

$$\varepsilon_{im} \sim N(0, \sigma^2).$$

```
Subject = nominal(Subject);
Program = nominal(Program);
D = dummyvar(Program); % Create dummy variables for Program
X = [ones(120,1), InitialWeight, D(:,2:4), Week, ...
     D(:,2).*Week, D(:,3).*Week, D(:,4).*Week];
Z = [ones(120,1), Week];
G = Subject;
```

Since the model has an intercept, you only need the dummy variables for programs B, C, and D. This is also known as the 'reference' method of coding dummy variables.

Fit the model using LinearMixedModel.fitmatrix with the defined design matrices and grouping variables.

```
lme = LinearMixedModel.fitmatrix(X,y,Z,G,'FixedEffectPredictors',...
{'Intercept','InitWeight','PrgB','PrgC','PrgD','Week','Week_PrgB','Week_PrgC','Week_PrgD',
'RandomEffectPredictors',{{'Intercept','Week'}},'RandomEffectGroups',{ 'Subject'})
```

```
lme =
```

```
Linear mixed-effects model fit by ML
```

```
Model information:
```

Number of observations	120
Fixed effects coefficients	9
Random effects coefficients	40
Covariance parameters	4

```
Formula:
```

```
Linear Mixed Formula with 10 predictors.
```

```
Model fit statistics:
```

AIC	BIC	LogLikelihood	Deviance
-22.981	13.257	24.49	-48.981

```
Fixed effects coefficients (95% CIs):
```

Name	Estimate	SE	tStat	DF	pValue
'Intercept'	0.66105	0.25892	2.5531	111	0.012034
'InitWeight'	0.0031879	0.0013814	2.3078	111	0.022863
'PrgB'	0.36079	0.13139	2.746	111	0.0070394
'PrgC'	-0.033263	0.13117	-0.25358	111	0.80029
'PrgD'	0.11317	0.13132	0.86175	111	0.39068
'Week'	0.1732	0.067454	2.5677	111	0.011567
'Week_PrgB'	0.038771	0.095394	0.40644	111	0.68521
'Week_PrgC'	0.030543	0.095394	0.32018	111	0.74944
'Week_PrgD'	0.033114	0.095394	0.34713	111	0.72915

Lower	Upper
0.14798	1.1741
0.00045067	0.0059252
0.10044	0.62113
-0.29319	0.22666
-0.14706	0.3734
0.039536	0.30686
-0.15026	0.2278
-0.15849	0.21957

-0.15592      0.22214

Random effects covariance parameters (95% CIs):

Group: Subject (20 Levels)

Name1	Name2	Type	Estimate	Lower
'Intercept'	'Intercept'	'std'	0.18407	0.12281
'Week'	'Intercept'	'corr'	0.66841	0.21077
'Week'	'Week'	'std'	0.15033	0.11004

Upper  
0.27587  
0.88573  
0.20537

Group: Error

Name	Estimate	Lower	Upper
'Res Std'	0.10261	0.087882	0.11981

The *P*-values 0.0228 and 0.0115 indicate significant effects of the initial weights of the subjects and the time factor in the amount of weight lost. The weight loss of subjects that are in program B is significantly different relative to the weight loss of subjects that are in program A. The lower and upper limits of the covariance parameters for the random effects do not include zero, thus they seem significant. You can also test the significance of the random-effects using the `compare` method.

## Random-Intercept Model

Load the sample data.

```
load flu
```

`flu` dataset array has a `Date` variable, and 10 variables for estimated influenza rates (in 9 different regions, estimated from Google® searches, plus a nationwide estimate from the CDC).

To fit a linear-mixed effects model, where the influenza rates are the responses, combine the nine columns corresponding to the regions into an array that has a single response variable, `FluRate`, and a nominal variable, `Region`, the nationwide estimate `WtdILI`, that shows which region each estimate is from, and the grouping variable `Date`.

```
flu2 = stack(flu, 2:10, 'NewDataVarName', 'FluRate', ...
            'IndVarName', 'Region');
flu2.Date = nominal(flu2.Date);
```

Define the design matrices for a random-intercept linear mixed-effects model, where the intercept varies by Date. The corresponding model is

$$y_{im} = \beta_0 + \beta_1 \text{WtdILI}_{im} + b_{0m} + \varepsilon_{im}, \quad i = 1, 2, \dots, 468, \quad m = 1, 2, \dots, 52,$$

where  $y_{im}$  is the observation  $i$  for level  $m$  of grouping variable Date,  $b_{0m}$  is the random effect for level  $m$  of the grouping variable Date, and  $\varepsilon_{im}$  is the observation error for observation  $i$ . The random effect has the prior distribution,

$$b_{0m} \sim N(0, \sigma_b^2),$$

and the error term has the distribution,

$$\varepsilon_{im} \sim N(0, \sigma^2).$$

```
y = flu2.FluRate;
X = [ones(468,1) flu2.WtdILI];
Z = [ones(468,1)];
G = flu2.Date;
```

Fit the linear mixed-effects model.

```
lme = LinearMixedModel.fitmatrix(X, y, Z, G, 'FixedEffectPredictors', {'Intercept', 'NationalRate'},
                                'RandomEffectPredictors', {'Intercept'}, 'RandomEffectGroups', {'Date'})
```

```
lme =
```

```
Linear mixed-effects model fit by ML
```

```
Model information:
```

Number of observations	468
Fixed effects coefficients	2
Random effects coefficients	52
Covariance parameters	2

```
Formula:
```

```
y ~ Intercept + NationalRate + (Intercept | Date)
```

```
Model fit statistics:
```

AIC	BIC	LogLikelihood	Deviance
-----	-----	---------------	----------

286.24    302.83    -139.12            278.24

Fixed effects coefficients (95% CIs):

Name	Estimate	SE	tStat	DF	pValue
'Intercept'	0.16385	0.057525	2.8484	466	0.0045885
'NationalRate'	0.7236	0.032219	22.459	466	3.0502e-76

Lower	Upper
0.050813	0.27689
0.66028	0.78691

Random effects covariance parameters (95% CIs):

Group: Date (52 Levels)

Name1	Name2	Type	Estimate	Lower
'Intercept'	'Intercept'	'std'	0.17146	0.13227

Upper  
0.22226

Group: Error

Name	Estimate	Lower	Upper
'Res Std'	0.30201	0.28217	0.32324

The confidence limits of the standard deviation of the random-effects term  $\sigma_b$  do not include zero (0.13227, 0.22226), which indicates that the random-effects term is significant. You can also test the significance of the random-effects using `compare` method.

The estimated value of an observation is the sum of the fixed-effects values and value of the random effect at the grouping variable level corresponding to that observation. For example, the estimated flu rate for observation 28

$$\begin{aligned}\hat{y}_{28} &= \hat{\beta}_0 + \hat{\beta}_1 \text{WtdILI}_{28} + \hat{b}_{10/30/2005} \\ &= 0.1639 + 0.7236 * (1.343) + 0.3318 \\ &= 1.46749,\end{aligned}$$

where  $\hat{b}$  is the BLUP of the random effects for the intercept. You can compute this value in the following way.

```
beta = fixedEffects(lme);
[~,~,STATS] = randomEffects(lme); % compute the random effects statistics STATS
```

```
STATS.Level = nominal(STATS.Level);  
y_hat = beta(1) + beta(2)*flu2.WtdILI(28) + STATS.Estimate(STATS.Level=='10/30/2005')  
  
y_hat =  
  
    1.4674
```

You can simply display the fitted value using the `fitted(lme)` method.

```
F = fitted(lme);  
F(28)  
  
ans =  
  
    1.4674
```

## Randomized-Block Design

Load the sample data.

```
load(fullfile(matlabroot, 'examples', 'stats', 'shift.mat'));
```

The data set shows the deviations from the target quality characteristic measured from the products each of five operators manufacture over three different shifts, morning, evening, and night. This is a randomized block design, where the operators are the blocks. The experiment is designed to study the impact of the time of shift on the performance. The performance measure is the deviations of the quality characteristics from the target value. This is simulated data.

Define the design matrices for a linear mixed-effects model with a random intercept grouped by operator, and shift as the fixed effects. Use the 'effects' contrasts. 'effects' contrasts mean that the coefficients sum to zero. You need to create two contrast coded variables in the fixed-effects design matrix, `X1` and `X2`, where

$$\text{Shift\_Evening} = \begin{cases} 0, & \text{if Morning} \\ 1, & \text{if Evening} \\ -1, & \text{if Night} \end{cases}$$

$$\text{Shift\_Morning} = \begin{cases} 1, & \text{if Morning} \\ 0, & \text{if Evening} \\ -1, & \text{if Night} \end{cases}$$

The model corresponds to

$$\text{Morning Shift: } \text{QCDev}_{im} = \beta_0 + \beta_2 \text{Shift\_Morning}_i + b_{0m} + \epsilon_{im},$$

$$\text{Evening Shift: } \text{QCDev}_{im} = \beta_0 + \beta_1 \text{Shift\_Evening}_i + b_{0m} + \epsilon_{im},$$

$$\text{Night Shift: } \text{QCDev}_{im} = \beta_0 - \beta_1 \text{Shift\_Evening}_i - \beta_2 \text{Shift\_Morning}_i + b_{0m} + \epsilon_{im},$$

where  $i$  represents the observations, and  $m$  represents the operators,  $i = 1, 2, \dots, 15$ , and  $m = 1, 2, \dots, 5$ . The random effects and the observation error have the following distributions:

$$b_{0m} \sim N(0, \sigma_b^2)$$

and

$$\epsilon_{im} \sim N(0, \sigma^2).$$

```
S = shift.Shift;
X1 = (S=='Morning') - (S=='Night');
X2 = (S=='Evening') - (S=='Night');
X = [ones(15,1), X1, X2];
y = shift.QCDev;
Z = ones(15,1);
G = shift.Operator;
```

Fit a linear mixed-effects model using the specified design matrices and restricted maximum likelihood method.

```
lme = LinearMixedModel.fitmatrix(X,y,Z,G,'FitMethod','REML','FixedEffectPredictors',...
{'Intercept','S_Morning','S_Evening'},'RandomEffectPredictors',{'Intercept'},...
'RandomEffectGroups',{'Operator'},'DummyVarCoding','effects')
```

```
lme =
```

```
Linear mixed-effects model fit by REML
```

```
Model information:
```

Number of observations	15
Fixed effects coefficients	3
Random effects coefficients	5

Covariance parameters 2

Formula:

y ~ Intercept + S\_Morning + S\_Evening + (Intercept | Operator)

Model fit statistics:

AIC	BIC	LogLikelihood	Deviance
58.913	61.337	-24.456	48.913

Fixed effects coefficients (95% CIs):

Name	Estimate	SE	tStat	DF	pValue
'Intercept'	3.6525	0.94109	3.8812	12	0.0021832
'S_Morning'	-0.91973	0.31206	-2.9473	12	0.012206
'S_Evening'	-0.53293	0.31206	-1.7078	12	0.11339

Lower	Upper
1.6021	5.703
-1.5997	-0.23981
-1.2129	0.14699

Random effects covariance parameters (95% CIs):

Group: Operator (5 Levels)

Name1	Name2	Type	Estimate	Lower
'Intercept'	'Intercept'	'std'	2.0457	0.98207

Upper  
4.2612

Group: Error

Name	Estimate	Lower	Upper
'Res Std'	0.85462	0.52357	1.395

Compute the best linear unbiased predictor (BLUP) estimates of random effects.

B = randomEffects(lme)

B =

0.5775  
1.1757  
-2.1715



```
2.3655
-1.9472
```

The estimated deviation from the target quality characteristics for the third operator working in the evening shift is

$$\begin{aligned}\hat{y}_{\text{Evening,Operator3}} &= \hat{\beta}_0 + \hat{\beta}_1 \text{Shift\_Evening} + \hat{b}_{03} \\ &= 3.6525 - 0.53293 - 2.1715 \\ &= 0.94807.\end{aligned}$$

You can also display this value in the following way.

```
F = fitted(lme);
F(shift.Shift=='Evening' & shift.Operator=='3')

ans =

    0.9481
```

### Correlated and Uncorrelated Random-Effects Terms

Load the sample data.

```
load carbig
```

Fit a linear mixed-effects model for miles per gallon (MPG), with fixed effects for acceleration, horsepower and the cylinders, and uncorrelated random-effect for intercept and acceleration grouped by the model year. This model corresponds to  $\text{MPG}_{im} = \beta_0 + \beta_1 \text{Acc}_i + \beta_2 \text{HP} + b_{0m} + b_{1m} \text{Acc}_{im} + \varepsilon_{im}$ ,  $m = 1, 2, 3$ ,

with the random-effects terms having the following prior distributions:

$$\begin{aligned}b_{0m} &\sim N(0, \sigma_0^2), \\ b_{1m} &\sim N(0, \sigma_1^2),\end{aligned}$$

where  $m$  represents the model year.

First, prepare the design matrices for fitting the linear mixed-effects model.

```
X = [ones(406,1) Acceleration Horsepower];
Z = {ones(406,1),Acceleration};
G = {Model_Year,Model_Year};
Model_Year = nominal(Model_Year);
```

Now, fit the model using `LinearMixedModel.fitmatrix` with the defined design matrices and grouping variables.

```
lme = LinearMixedModel.fitmatrix(X,MPG,Z,G,'FixedEffectPredictors',...
{'Intercept','Acceleration','Horsepower'},'RandomEffectPredictors',...
{{'Intercept'},{'Acceleration'}},'RandomEffectGroups',{'Model_Year','Model_Year'})
```

```
lme =
```

```
Linear mixed-effects model fit by ML
```

```
Model information:
```

Number of observations	392
Fixed effects coefficients	3
Random effects coefficients	26
Covariance parameters	3

```
Formula:
```

```
Linear Mixed Formula with 4 predictors.
```

```
Model fit statistics:
```

AIC	BIC	LogLikelihood	Deviance
2194.5	2218.3	-1091.3	2182.5

```
Fixed effects coefficients (95% CIs):
```

Name	Estimate	SE	tStat	DF	pValue
'Intercept'	49.839	2.0518	24.291	389	5.6168e-80
'Acceleration'	-0.58565	0.10846	-5.3995	389	1.1652e-07
'Horsepower'	-0.16534	0.0071227	-23.213	389	1.9755e-75

Lower	Upper
45.806	53.873
-0.7989	-0.3724
-0.17934	-0.15133

```
Random effects covariance parameters (95% CIs):
```

```
Group: Model_Year (13 Levels)
```

```

Name1          Name2          Type          Estimate      Lower
'Intercept'    'Intercept'    'std'        8.6657e-07    NaN

Upper
NaN

Group: Model_Year (13 Levels)
Name1          Name2          Type          Estimate      Lower
'Acceleration' 'Acceleration' 'std'        0.18783      0.12523

Upper
0.28172

Group: Error
Name          Estimate      Lower      Upper
'Res Std'    3.7258      3.4698    4.0007

```

The standard deviation of the random effect for the intercept does not seem significant.

Refit the model with potentially correlated random effects for intercept and acceleration. In this case the random-effects terms have the following prior distribution

$$b_m = \begin{pmatrix} b_{0m} \\ b_{1m} \end{pmatrix} \sim N \left( 0, \begin{pmatrix} \sigma_0^2 & \sigma_{0,1} \\ \sigma_{0,1} & \sigma_1^2 \end{pmatrix} \right),$$

where  $m$  represents the model year.

First prepare the random-effects design matrix and grouping variable.

```

Z = [ones(406,1) Acceleration];
G = Model_Year;

lme = LinearMixedModel.fitmatrix(X,MPG,Z,G,'FixedEffectPredictors',...
{'Intercept','Acceleration','Horsepower'},'RandomEffectPredictors',...
{{'Intercept','Acceleration'}},'RandomEffectGroups',{'Model_Year'})

lme =

Linear mixed-effects model fit by ML

```

Model information:

Number of observations	392
Fixed effects coefficients	3
Random effects coefficients	26
Covariance parameters	4

Formula:

Linear Mixed Formula with 4 predictors.

Model fit statistics:

AIC	BIC	LogLikelihood	Deviance
2193.5	2221.3	-1089.7	2179.5

Fixed effects coefficients (95% CIs):

Name	Estimate	SE	tStat	DF	pValue
'Intercept'	50.133	2.2652	22.132	389	7.7727e-71
'Acceleration'	-0.58327	0.13394	-4.3545	389	1.7075e-05
'Horsepower'	-0.16954	0.0072609	-23.35	389	5.188e-76

Lower	Upper
45.679	54.586
-0.84661	-0.31992
-0.18382	-0.15527

Random effects covariance parameters (95% CIs):

Group: Model\_Year (13 Levels)

Name1	Name2	Type	Estimate
'Intercept'	'Intercept'	'std'	3.3475
'Acceleration'	'Intercept'	'corr'	-0.87971
'Acceleration'	'Acceleration'	'std'	0.33789

Lower	Upper
1.2862	8.7119
-0.98501	-0.29675
0.1825	0.62558

Group: Error

Name	Estimate	Lower	Upper
'Res Std'	3.6874	3.4298	3.9644

The confidence intervals for the standard deviations and the correlation between the random effects for intercept and acceleration do not include zeros, hence they seem significant. You can compare these two models using the `LinearMixedModel.compare` method.

### Specify the Covariance Pattern

Load the sample data.

```
load(fullfile(matlabroot, 'examples', 'stats', 'weight.mat'));
```

`weight` contains data from a longitudinal study, where 20 subjects are randomly assigned 4 exercise programs, and their weight loss is recorded over six two-week time periods. This is simulated data.

Define `Subject` and `Program` as categorical variables.

```
Subject = nominal(Subject);
Program = nominal(Program);
```

Create the design matrices for a linear mixed-effects model, with the initial weight, type of program, and week are the fixed effects.

```
D = dummyvar(Program);
X = [ones(120,1), InitialWeight, D(:,2:4), Week];
Z = [ones(120,1) Week];
G = Subject;
```

This model corresponds to

$$y_{im} = \beta_0 + \beta_1 IW_i + \beta_2 \text{Week}_i + \beta_3 I[\text{PB}]_i + \beta_4 I[\text{PC}]_i + \beta_5 I[\text{PD}]_i \\ + b_{0m} + b_{1m} \text{Week}_{2im} + b_{2m} \text{Week}_{4im} + b_{3m} \text{Week}_{6im} + b_{4m} \text{Week}_{8im} \\ + b_{5m} \text{Week}_{10im} + b_{6m} \text{Week}_{12im} + \varepsilon_{im},$$

where  $i = 1, 2, \dots, 120$ , and  $m = 1, 2, \dots, 20$ .

$\beta_j$  are the fixed-effects coefficients,  $j = 0, 1, \dots, 8$ , and  $b_{0m}$  and  $b_{1m}$  are random effects. `IW` stands for initial weight and  $I[\cdot]$  is a dummy variable representing a type of program. For example,  $I[\text{PB}]_i$  is the dummy variable representing program type B. The random effects and observation error have the following prior distributions:

$$b_{0m} \sim N(0, \sigma_0^2)$$

$$b_{1m} \sim N(0, \sigma_1^2)$$

$$\varepsilon_{im} \sim N(0, \sigma^2).$$

Fit the model using `LinearMixedModel.fitmatrix` with the defined design matrices and grouping variables. Assume the repeated observations collected on a subject have common variance along diagonals.

```
lme = LinearMixedModel.fitmatrix(X, y, Z, G, 'FixedEffectPredictors', ...
    {'Intercept', 'InitWeight', 'PrgB', 'PrgC', 'PrgD', 'Week'}, ...
    'RandomEffectPredictors', {'Intercept', 'Week'}, ...
    'RandomEffectGroups', {'Subject'}, 'CovariancePattern', 'Isotropic')
```

```
lme =
```

```
Linear mixed-effects model fit by ML
```

```
Model information:
```

Number of observations	120
Fixed effects coefficients	6
Random effects coefficients	40
Covariance parameters	2

```
Formula:
```

```
Linear Mixed Formula with 7 predictors.
```

```
Model fit statistics:
```

AIC	BIC	LogLikelihood	Deviance
-24.783	-2.483	20.391	-40.783

```
Fixed effects coefficients (95% CIs):
```

Name	Estimate	SE	tStat	DF	pValue
'Intercept'	0.4208	0.28169	1.4938	114	0.13799
'InitWeight'	0.0045552	0.0015338	2.9699	114	0.0036324
'PrgB'	0.36993	0.12119	3.0525	114	0.0028242
'PrgC'	-0.034009	0.1209	-0.28129	114	0.77899
'PrgD'	0.121	0.12111	0.99911	114	0.31986
'Week'	0.19881	0.037134	5.3538	114	4.5191e-07

Lower	Upper
-0.13723	0.97883
0.0015168	0.0075935

```

    0.12986      0.61
   -0.27351      0.2055
   -0.11891      0.36091
    0.12525      0.27237

```

Random effects covariance parameters (95% CIs):

Group: Subject (20 Levels)

Name1	Name2	Type	Estimate	Lower
'Intercept'	'Intercept'	'std'	0.16561	0.12896

```

Upper
0.21269

```

Group: Error

Name	Estimate	Lower	Upper
'Res Std'	0.10272	0.088014	0.11987

## Definitions

### Cholesky Parameterization

One of the assumptions of linear mixed-effects models is that the random effects have the following prior distribution.

$$b \sim N(0, \sigma^2 D(\theta)),$$

where  $D$  is a  $q$ -by- $q$  symmetric and positive semidefinite matrix, parameterized by a variance component vector  $\theta$ ,  $q$  is the number of variables in the random-effects term, and  $\sigma^2$  is the observation error variance. Since the covariance matrix of the random effects,  $D$ , is symmetric, it has  $q(q+1)/2$  free parameters. Suppose  $L$  is the lower triangular Cholesky factor of  $D(\theta)$  such that

$$D(\theta) = L(\theta)L(\theta)^T,$$

then the  $q^*(q+1)/2$ -by-1 unconstrained parameter vector  $\theta$  is formed from elements in the lower triangular part of  $L$ .

For example, if

$$L = \begin{bmatrix} L_{11} & 0 & 0 \\ L_{21} & L_{22} & 0 \\ L_{31} & L_{32} & L_{33} \end{bmatrix},$$

then

$$\theta = \begin{bmatrix} L_{11} \\ L_{21} \\ L_{31} \\ L_{22} \\ L_{32} \\ L_{33} \end{bmatrix}.$$

## Log-Cholesky Parameterization

When the diagonal elements of  $L$  in Cholesky parameterization are constrained to be positive, then the solution for  $L$  is unique. Log-Cholesky parameterization is the same as Cholesky parameterization except that the logarithm of the diagonal elements of  $L$  are used to guarantee unique parameterization.

For example, for the 3-by-3 example in Cholesky parameterization, enforcing  $L_{ii} \geq 0$ ,

$$\theta = \begin{bmatrix} \log(L_{11}) \\ L_{21} \\ L_{31} \\ \log(L_{22}) \\ L_{32} \\ \log(L_{33}) \end{bmatrix}.$$

## Tips

- If your model is not easily described using a formula, you can create matrices to define the fixed and random effects, and fit the model using `fitlmematrix`.



## Alternatives

You can also fit a linear mixed-effects model using `fitlme(tbl, formula)`, where `tbl` is a table or dataset array containing the response `y`, the predictor variables `X`, and the grouping variables, and `formula` is of the form `'y ~ fixed + (random1|g1) + ... + (randomR|gR)'`.

If your model is not easily described using a formula, you can create matrices to define the fixed and random effects, and fit the model using `fitlmematrix(X, y, Z, G)`.

## See Also

[LinearMixedModel](#) | [fitlme](#)

## fitNaiveBayes

(to be removed) Train naive Bayes classifier

---

**Note** `fitNaiveBayes` will be removed in a future release. Use `fitcnb` instead.

---

### Syntax

```
NBModel = fitNaiveBayes(X, Y)
NBModel = fitNaiveBayes(X, Y, Name, Value)
```

### Description

`NBModel = fitNaiveBayes(X, Y)` returns a naive Bayes classifier `NBModel`, trained by predictors `X` and class labels `Y` for  $K$ -level classification.

Predict labels for new data by passing the data and `NBModel` to `predict`.

`NBModel = fitNaiveBayes(X, Y, Name, Value)` returns a naive Bayes classifier with additional options specified by one or more `Name, Value` pair arguments.

For example, you can specify a distribution to model the data, prior probabilities for the classes, or the kernel smoothing window bandwidth.

### Input Arguments

#### **x** — Predictor data

numeric matrix

Predictor data, specified as a numeric matrix.

Each row of `X` corresponds to one observation (also known as an instance or example), and each column corresponds to one variable (also known as a feature).

The length of `Y` and the number of rows of `X` must be equivalent.

Data Types: `double`

### **$Y$ — Class labels**

categorical array | character array | logical vector | numeric vector | cell array of character vectors

Class labels to which the naive Bayes classifier is trained, specified as a categorical or character array, logical or numeric vector, or cell array of character vectors. Each element of  $Y$  defines the class membership of the corresponding row of  $X$ .  $Y$  supports  $K$  class levels.

If  $Y$  is a character array, then each row must correspond to one class label.

The length of  $Y$  and the number of rows of  $X$  must be equivalent.

Data Types: `cell` | `char` | `double` | `logical`

---

**Note:** The software treats NaN, empty character vector (' '), and <undefined> elements as missing values.

- If  $Y$  contains missing values, then the software removes them and the corresponding rows of  $X$ .
- If  $X$  contains any rows composed entirely of missing values, then the software removes those rows and the corresponding elements of  $Y$ .
- If  $X$  contains missing values and you set 'Distribution', 'mn', then the software removes those rows of  $X$  and the corresponding elements of  $Y$ .
- If a predictor is not represented in a class, that is, if all of its values are NaN within a class, then the software returns an error.

Removing rows of  $X$  and corresponding elements of  $Y$  decreases the effective training or cross-validation sample size.

---

## **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'Distribution', 'mn', 'Prior', 'uniform', 'KSWidth', 0.5` specifies the following: the data distribution is multinomial, the prior probabilities for all classes are equal, and the kernel smoothing window bandwidth for all classes is 0.5 units.

### Distribution — Data distributions

`'normal'` (default) | `'kernel'` | `'mn'` | `'mvnm'` | cell array of character vectors

Data distributions `fitNaiveBayes` uses to model the data, specified as the comma-separated pair consisting of `'Distribution'` and `'kernel'`, `'mn'`, `'mvnm'`, `'normal'`, or a cell array of character vectors.

This table summarizes the available distributions.

Value	Description
<code>'kernel'</code>	Kernel smoothing density estimate.
<code>'mn'</code>	Multinomial distribution. If you specify <code>mn</code> , then all features are components of a multinomial distribution. Therefore, you cannot include <code>'mn'</code> as an element of a cell array of character vectors. For details, see “Algorithms” on page 32-2538.
<code>'mvnm'</code>	Multivariate multinomial distribution. For details, see “Algorithms” on page 32-2538.
<code>'normal'</code>	Normal (Gaussian) distribution.

If you specify a character vector, then the software models all the features using that distribution. If you specify a 1-by- $D$  cell array of character vectors, then the software models feature  $j$  using the distribution in element  $j$  of the cell array.

Example: `'Distribution', {'kernel', 'normal'}`

Data Types: `cell` | `char`

### kSSupport — Kernel smoothing density support

`'unbounded'` (default) | `'positive'` | cell array | numeric row vector

Kernel smoothing density support, specified as the comma-separated pair consisting of `'kSSupport'` and a numeric row vector, `'positive'`, `'unbounded'`, or a cell array. The software applies the kernel smoothing density to this region.

If you do not specify `'Distribution'`, `'kernel'`, then the software ignores the values of `'kSSupport'`, `'KSType'`, and `'KSWidth'`.

This table summarizes the available options for setting the kernel smoothing density region.

Value	Description
1-by-2 numeric row vector	For example, $[L, U]$ , where $L$ and $U$ are the finite lower and upper bounds, respectively, for the density support.
'positive'	The density support is all positive real values.
'unbounded'	The density support is all real values.

If you specify a 1-by- $D$  cell array, with each cell containing any value in the table, then the software trains the classifier using the kernel support in cell  $j$  for feature  $j$  in  $X$ .

Example: `'KSSupport', {[ -10, 20], 'unbounded'}`

Data Types: `cell` | `char` | `double`

#### **KSType** — Kernel smoother type

'normal' (default) | 'box' | 'epanechnikov' | 'triangle' | cell array of character vectors

Kernel smoother type, specified as the comma-separated pair consisting of 'KSType' and 'box', 'epanechnikov', 'normal', 'triangle', or a cell array of character vectors.

If you do not specify 'Distribution', 'kernel', then the software ignores the values of 'KSSupport', 'KSType', and 'KSWidth'.

This table summarizes the available options for setting the kernel smoothing density region. Let  $I\{u\}$  denote the indicator function.

Value	Kernel	Formula
'box'	Box (uniform)	$f(x) = 0.5I\{ x  \leq 1\}$
'epanechnikov'	Epanechnikov	$f(x) = 0.75(1 - x^2)I\{ x  \leq 1\}$
'normal'	Gaussian	$f(x) = \frac{1}{\sqrt{2\pi}} \exp(-0.5x^2)$
'triangle'	Triangular	$f(x) = (1 -  x )I\{ x  \leq 1\}$

If you specify a 1-by- $D$  cell array, with each cell containing any value in the table, then the software trains the classifier using the kernel smoother type in cell  $j$  for feature  $j$  in  $X$ .

Example: 'KSType', {'epanechnikov', 'normal'}

Data Types: cell | char

### **kswidth** — Kernel smoothing window bandwidth

matrix of numeric values (default) | numeric column vector | numeric row vector | scalar | structure array

Kernel smoothing window bandwidth, specified as the comma-separated pair consisting of 'KSWidth' and a matrix of numeric values, numeric row vector, numeric column vector, scalar, or structure array.

If you do not specify 'Distribution', 'kernel', then the software ignores the values of 'KSSupport', 'KSType', and 'KSWidth'.

Suppose there are  $K$  class levels and  $D$  predictors. This table summarizes the available options for setting the kernel smoothing window bandwidth.

Value	Description
$K$ -by- $D$ matrix of numeric values	Element $(k,d)$ specifies the bandwidth for predictor $d$ in class $k$ .
$K$ -by-1 numeric column vector	Element $k$ specifies the bandwidth for all predictors in class $k$ .
1-by- $D$ numeric row vector	Element $d$ specifies the bandwidth in all class levels for predictor $d$ .
scalar	Specifies the bandwidth for all features in all classes.
structure array	A structure array $S$ containing class levels and their bandwidths. $S$ must have two fields: <ul style="list-style-type: none"> <li><math>S.width</math>: A numeric row vector of bandwidths, or a matrix of numeric values with <math>D</math> columns.</li> <li><math>S.group</math>: A vector of the same type as <math>Y</math>, containing unique class levels indicating the class for the corresponding element of <math>S.width</math>.</li> </ul>

By default, the software selects a default bandwidth automatically for each combination of feature and class by using a value that is optimal for a Gaussian distribution.

Example: `'KSWidth', struct('width', [0.5, 0.25], 'group', {'b'; 'g'})`

Data Types: `double` | `struct`

### **Prior — Class prior probabilities**

`'empirical'` (default) | `'uniform'` | numeric vector | structure array

Class prior probabilities, specified as the comma-separated pair consisting of `'Prior'` and a numeric vector, structure array, `'uniform'`, or `'empirical'`.

This table summarizes the available options for setting prior probabilities.

Value	Description
<code>'empirical'</code>	The software uses the class relative frequencies distribution for the prior probabilities.
numeric vector	A numeric vector of length $K$ specifying the prior probabilities for each class. The order of the elements of <code>Prior</code> should correspond to the order of the class levels. For details on the order of the classes, see “Algorithms” on page 32-2538.  The software normalizes prior probabilities to sum to 1.
structure array	A structure array <code>S</code> containing class levels and their prior probabilities. <code>S</code> must have two fields: <ul style="list-style-type: none"> <li><code>S.prob</code>: A numeric vector of prior probabilities. The software normalizes prior probabilities to sum to 1.</li> <li><code>S.group</code>: A vector of the same type as <code>Y</code> containing unique class levels indicating the class for the corresponding element of <code>S.prob</code>. <code>S.class</code> must contain all the <math>K</math> levels in <code>Y</code>. It can also contain classes that do not appear in <code>Y</code>. This can be useful if <code>X</code> is a subset of a larger training set. The software ignores any classes that appear in <code>S.group</code> but not in <code>Y</code>.</li> </ul>
<code>'uniform'</code>	The prior probabilities are equal for all classes.

Example: `'Prior', struct('prob', [1, 2], 'group', {'b'; 'g'})`

Data Types: `char` | `double` | `struct`

## Output Arguments

### **NBModel** — Trained naive Bayes classifier

NaiveBayes classifier

Trained naive Bayes classifier, returned as a NaiveBayes classifier.

## Definitions

### Bag-of-Tokens Model

In the bag-of-tokens model, the value of predictor  $j$  is the nonnegative number of occurrences of token  $j$  in this observation. The number of categories (bins) in this multinomial model is the number of distinct tokens, that is, the number of predictors.

## Tips

- For classifying count-based data, such as the bag-of-tokens model on page 32-2538, use the multinomial distribution (e.g., set 'Distribution', 'mn').
- This list defines the order of the classes. It is useful when you specify prior probabilities by setting 'Prior', prior, where prior is a numeric vector.
  - If  $Y$  is a categorical array, then the order of the class levels matches the output of `categories(Y)`.
  - If  $Y$  is a numeric or logical vector, then the order of the class levels matches the output of `sort(unique(Y))`.
  - For cell arrays of character vectors and character arrays, the order of the class labels is the order which each label appears in  $Y$ .

## Algorithms

- If you specify 'Distribution', 'mn', then the software considers each observation as multiple trials of a multinomial distribution, and considers each occurrence of a token as one trial (see “Bag-of-Tokens Model” on page 32-2538).



- If you specify 'Distribution', 'mvmn', then the software assumes each individual predictor follows a multinomial model within a class. The parameters for a predictor include the probabilities of all possible values that the corresponding feature can take.

## See Also

NaiveBayes | fitcnb | posterior | predict

## Topics

“Naive Bayes Classification” on page 21-2

“Grouping Variables” on page 2-59

**Introduced in R2014a**

## fitPosterior

**Class:** ClassificationSVM

Fit posterior probabilities

### Syntax

```
ScoreSVMModel = fitPosterior(SVMModel)
[ScoreSVMModel,ScoreTransform] = fitPosterior(SVMModel)
[ScoreSVMModel,ScoreTransform] = fitPosterior(SVMModel,Name,Value)
```

### Description

`ScoreSVMModel = fitPosterior(SVMModel)` returns a trained support vector machine (SVM) classifier `ScoreSVMModel` containing the optimal score-to-posterior-probability transformation function for two-class learning.

The software fits the appropriate score-to-posterior-probability transformation function using the SVM classifier `SVMModel`, and by conducting 10-fold cross validation using the stored predictor data (`SVMModel.X`) and the class labels (`SVMModel.Y`) as outlined in [1]. The transformation function computes the posterior probability that an observation is classified into the positive class (`SVMModel.Classnames(2)`).

- If the classes are inseparable, then the transformation function is the sigmoid function on page 32-2564.
- If the classes are perfectly separable, then the transformation function is the step function on page 32-2564.
- In two-class learning, if one of the two classes has a relative frequency of 0, then the transformation function is the constant function on page 32-2683. `fitPosterior` is not appropriate for one-class learning.
- The software stores the optimal score transformation function in `ScoreSVMModel.ScoreTransform`.

`[ScoreSVMModel, ScoreTransform] = fitPosterior(SVMModel)` additionally returns the optimal score-to-posterior-probability transformation function parameters (`ScoreTransform`).

`[ScoreSVMModel, ScoreTransform] = fitPosterior(SVMModel, Name, Value)` returns the optimal score-to-posterior-probability transformation function and its parameters with additional options specified by one or more `Name, Value` pair arguments.

## Input Arguments

### **SVMModel** — Trained SVM classifier

`ClassificationSVM` classifier

Trained SVM classifier, specified as a `ClassificationSVM`.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### **CVPartition** — Cross-validation partition

`[]` (default) | `cvpartition` partition

Cross-validation partition used to compute the transformation function, specified as the comma-separated pair consisting of '`CVPartition`' and a `cvpartition` partition as created by `cvpartition`. You can use only one of these four options at a time for creating a cross-validated model: '`KFold`', '`Holdout`', '`Leaveout`', or '`CVPartition`'.

`crossval` splits the data into subsets using `cvpartition`.

### **Holdout** — Fraction of data for holdout validation

scalar value in the range (0,1)

Fraction of data for holdout validation used to compute the transformation function, specified as the comma-separated pair consisting of '`Holdout`' and a scalar value in the

range (0,1). Holdout validation tests the specified fraction of the data, and uses the remaining data for training.

You can use only one of these four options at a time for creating a cross-validated model: 'KFold', 'Holdout', 'Leaveout', or 'CVPartition'.

Example: 'Holdout', 0.1

Data Types: double | single

### **KFold** — Number of folds

10 (default) | positive integer value greater than 1

Number of folds to use when computing the transformation function, specified as the comma-separated pair consisting of 'KFold' and a positive integer value greater than 1.

You can use only one of these four options at a time for creating a cross-validated model: 'KFold', 'Holdout', 'Leaveout', or 'CVPartition'.

Example: 'KFold', 8

Data Types: single | double

### **Leaveout** — Leave-one-out cross-validation flag

'off' (default) | 'on'

Leave-one-out cross-validation flag indicating whether to use leave-one-out cross validation to compute the transformation function, specified as the comma-separated pair consisting of 'Leaveout' and 'on' or 'off'. Use leave-one-out cross validation by using 'on'.

You can use only one of these four options at a time for creating a cross-validated model: 'KFold', 'Holdout', 'Leaveout', or 'CVPartition'.

Example: 'Leaveout', 'on'

## Output Arguments

### **ScoreSVMModel** — Trained SVM classifier

ClassificationSVM classifier

Trained SVM classifier containing the estimated score-to-posterior-probability transformation function, returned as a `ClassificationSVM` classifier.

To estimate posterior probabilities for the training set observations, pass `ScoreSVMModel` to `resubPredict`.

To estimate posterior probabilities for new observations, then pass them and `ScoreSVMModel` to `predict`. If you set `'Standardize', true` in `fitcsvm` to train `SVMModel`, then `predict` standardizes the columns of `X` using the corresponding means in `SVMModel.Mu` and standard deviations in `SVMModel.Sigma`.

### **ScoreTransform** — Optimal score transformation function parameters

structure array

Optimal score-to-posterior-probability transformation function parameters, returned as a structure array.

- If field `Type` is `sigmoid`, then `ScoreTransform` has the following other fields:
  - `Slope`: The value of  $A$  in the sigmoid function on page 32-2564
  - `Intercept`: The value of  $B$  in the sigmoid function
- If field `Type` is `step`, then `ScoreTransform` has the following other fields:
  - `PositiveClassProbability`: The value of  $\pi$  in the step function on page 32-2564. It represents the probability that an observation is in the positive class. Also, the posterior probability that an observation is in the positive class given that its score is in the interval  $(\text{LowerBound}, \text{UpperBound})$ .
  - `LowerBound`: The value  $\max_{y_i = -1} s_n$  in the step function. It represents the lower bound of the score interval that assigns observations with scores in the interval  $(\text{LowerBound}, \text{UpperBound})$  the posterior probability of being in the positive class `PositiveClassProbability`. Any observation with a score less than `LowerBound` has the posterior probability of being the positive class 0.
  - `UpperBound`: The value  $\min_{y_i = 1} s_n$  in the step function. It represents the upper bound of the score interval that assigns observations with scores in the interval  $(\text{LowerBound}, \text{UpperBound})$  the posterior probability of being in the positive class `PositiveClassProbability`. Any observation with a score greater than `UpperBound` has the posterior probability of being the positive class 1.

- If field `Type` is constant, then `ScoreTransform.PredictedClass` contains the name of the class prediction.

This result is the same as `SVMModel.ClassNames`. The posterior probability of an observation being in `ScoreTransform.PredictedClass` is always 1.

## Examples

### Estimate In-Sample Posterior Probabilities of SVM Classifiers

Load the `ionosphere` data set.

```
load ionosphere
```

Train an SVM classifier. It is good practice to specify the class order and standardize the data.

```
SVMModel = fitcsvm(X,Y,'ClassNames',{'b','g'},'Standardize',true);
```

`SVMModel` is a `ClassificationSVM` classifier. The positive class is `'g'`.

Fit the optimal score-to-posterior-probability transformation function.

```
rng(1); % For reproducibility
ScoreSVMModel = fitPosterior(SVMModel)
```

```
ScoreSVMModel =

ClassificationSVM
    ResponseName: 'Y'
    CategoricalPredictors: []
         ClassNames: {'b' 'g'}
    ScoreTransform: '@(S) sigmoid(S,-9.481802e-01,-1.218745e-01) '
    NumObservations: 351
           Alpha: [90x1 double]
           Bias: -0.1343
    KernelParameters: [1x1 struct]
                Mu: [1x34 double]
                Sigma: [1x34 double]
    BoxConstraints: [351x1 double]
```

```

ConvergenceInfo: [1x1 struct]
IsSupportVector: [351x1 logical]
Solver: 'SMO'

```

Since the classes are inseparable, the score transformation function (`ScoreSVMModel.ScoreTransform`) is the sigmoid function.

Estimate scores and positive class posterior probabilities for the training data. Display the results for the first 10 observations.

```

[label,scores] = resubPredict(SVMModel);
[~,postProbs] = resubPredict(ScoreSVMModel);
table(Y(1:10),label(1:10),scores(1:10,2),postProbs(1:10,2),'VariableNames',...
      {'TrueLabel','PredictedLabel','Score','PosteriorProbability'})

```

ans =

10x4 table

TrueLabel	PredictedLabel	Score	PosteriorProbability
'g'	'g'	1.4861	0.82215
'b'	'b'	-1.0004	0.30436
'g'	'g'	1.8685	0.86916
'b'	'b'	-2.6458	0.084183
'g'	'g'	1.2805	0.79184
'b'	'b'	-1.4617	0.22028
'g'	'g'	2.1672	0.89814
'b'	'b'	-5.7085	0.0050122
'g'	'g'	2.4797	0.92223
'b'	'b'	-2.7811	0.074805

## Plot Posterior Probability Contours for Multiple Classes Using SVM

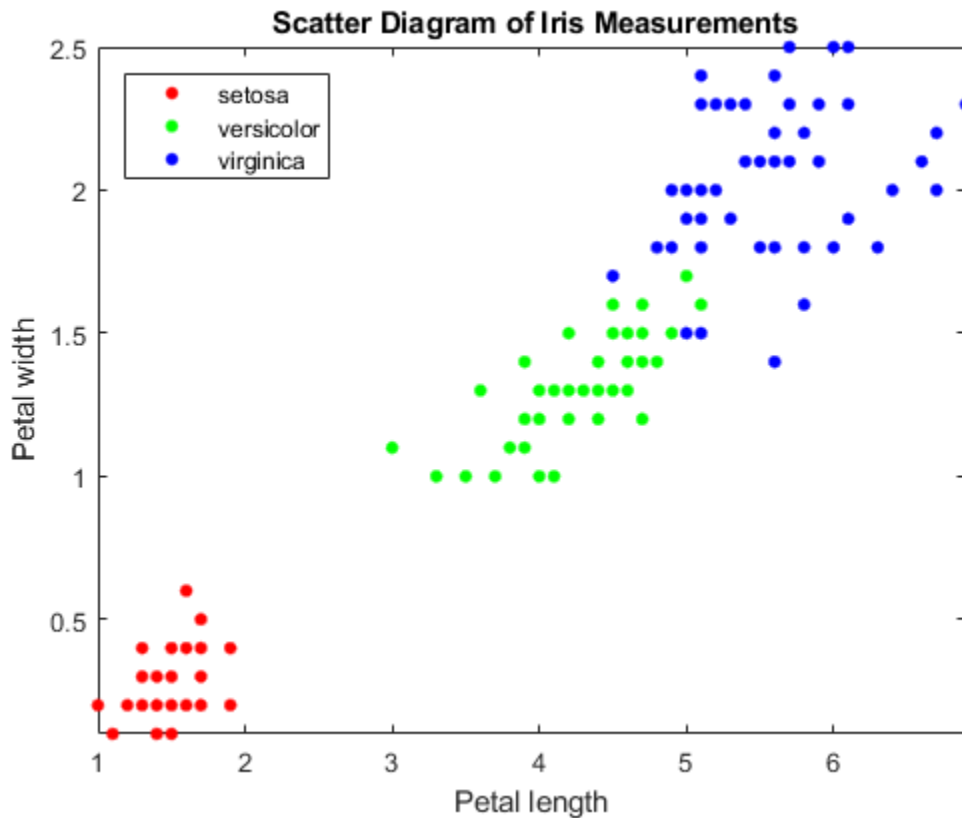
This example steps through the process of one-versus-all (OVA) classification to train a multiclass SVM classifier, and then plots probability contours for each class. To implement OVA directly, see `fitcecoc`.

Load Fisher's iris data set. Use the petal lengths and widths.

```
load fisheriris
X = meas(:,3:4);
Y = species;
```

Examine a scatter plot of the data.

```
figure
gscatter(X(:,1),X(:,2),Y);
title('\bf Scatter Diagram of Iris Measurements');
xlabel('Petal length');
ylabel('Petal width');
legend('Location','Northwest');
axis tight
```





Train three binary SVM classifiers that separate each type of iris from the others. Assume that a radial basis function is an appropriate kernel for each, and allow the algorithm to choose a kernel scale. It is good practice to define the class order.

```

classNames = {'setosa'; 'virginica'; 'versicolor'};
numClasses = size(classNames,1);
inds = cell(3,1); % Preallocation
SVMModel = cell(3,1);

rng(1); % For reproducibility
for j = 1:numClasses
    inds{j} = strcmp(Y,classNames{j}); % OVA classification
    SVMModel{j} = fitcsvm(X,inds{j},'ClassNames',[false true],...
        'Standardize',true,'KernelFunction','rbf','KernelScale','auto');
end

```

fitcsvm uses a heuristic procedure that involves subsampling to compute the value of the kernel scale.

Fit the optimal score-to-posterior-probability transformation function for each classifier.

```

for j = 1:numClasses
    SVMModel{j} = fitPosterior(SVMModel{j});
end

```

Warning: Classes are perfectly separated. The optimal score-to-posterior transformation is a step function.

Define a grid to plot the posterior probability contours. Estimate the posterior probabilities over the grid for each classifier.

```

d = 0.02;
[x1Grid,x2Grid] = meshgrid(min(X(:,1)):d:max(X(:,1)),...
    min(X(:,2)):d:max(X(:,2)));
xGrid = [x1Grid(:),x2Grid(:)];

posterior = cell(3,1);
for j = 1:numClasses
    [~,posterior{j}] = predict(SVMModel{j},xGrid);
end

```

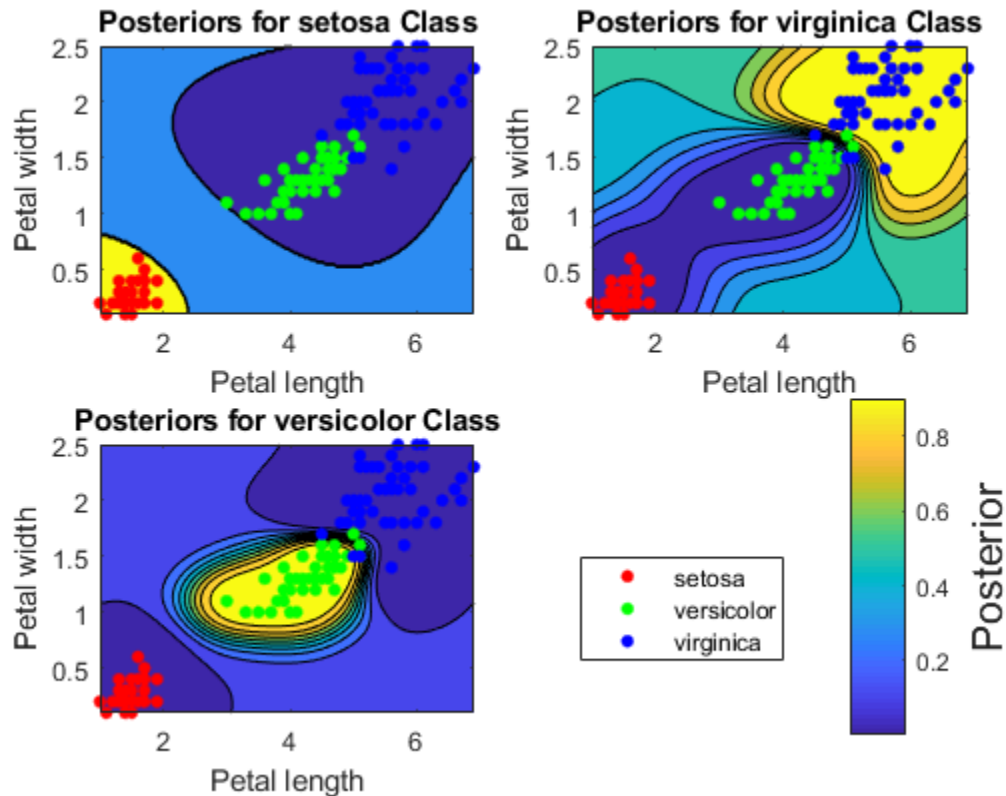
For each SVM classifier, plot the posterior probability contour under the scatter plot of the data.

```

figure
h = zeros(numClasses + 1,1); % Preallocation for graphics handles

```

```
for j = 1:numClasses
    subplot(2,2,j)
    contourf(x1Grid,x2Grid,reshape(posterior{j}(:,2),size(x1Grid,1),size(x1Grid,2)));
    hold on
    h(1:numClasses) = gscatter(X(:,1),X(:,2),Y);
    title(sprintf('Posteriors for %s Class',classNames{j}));
    xlabel('Petal length');
    ylabel('Petal width');
    legend off
    axis tight
    hold off
end
h(numClasses + 1) = colorbar('Location','EastOutside',...
    'Position',[0.8,0.1,0.05,0.4]);
set(get(h(numClasses + 1),'YLabel'),'String','Posterior','FontSize',16);
legend(h(1:numClasses),'Location',[0.6,0.2,0.1,0.1]);
```



### Fit Optimal Posterior Probability Function Using Holdout Cross Validation

Platt (2000) outlines a bias-reducing method of estimating the score-to-posterior-probability transformation function. This method estimates the transformation function after the SVM classifier is trained, and uses cross validation to reduce bias. By default, `fitPosterior` and `fitSVMPosterior` use 10-fold cross validation when they estimate the transformation function. To reduce run time for larger data sets, you can specify to use holdout cross validation instead.

Load the `ionosphere` data set.

```
load ionosphere
```

Train an SVM classifier. It is good practice to specify the class order and standardize the data.

```
SVMMModel = fitcsvm(X,Y,'ClassNames',{'b','g'},'Standardize',true);
```

SVMMModel is a ClassificationSVM classifier. The positive class is 'g'.

Fit the optimal score-to-posterior-probability transformation function. For comparison, use 10-fold cross validation (default) and specify a 10% holdout test sample.

```
rng(1); % For reproducibility

tic; % Start the stopwatch
SVMMModel_10FCV = fitPosterior(SVMMModel);
toc % Stop the stopwatch and display the run time

tic;
SVMMModel_HO = fitPosterior(SVMMModel,'Holdout',0.10);
toc

Elapsed time is 1.410929 seconds.
Elapsed time is 0.306305 seconds.
```

Though both runtimes are short because the data set is relatively small, SVMMModel\_HO fitted the score transformation function much faster than SVMMModel\_10FCV.

## Definitions

### Sigmoid Function

The sigmoid function that maps score  $s_j$  corresponding to observation  $j$  to the positive class posterior probability is

$$P(s_j) = \frac{1}{1 + \exp(As_j + B)}.$$

If the output argument ScoreTransform.Type is sigmoid, then parameters  $A$  and  $B$  correspond to the fields Scale and Intercept of ScoreTransform, respectively.

## Step Function

The step function that maps score  $s_j$  corresponding to observation  $j$  to the positive class posterior probability is

$$P(s_j) = \begin{cases} 0; & s < \max_{y_k=-1} s_k \\ \pi; & \max_{y_k=-1} s_k \leq s_j \leq \min_{y_k=+1} s_k, \\ 1; & s_j > \min_{y_k=+1} s_k \end{cases}$$

where:

- $s_j$  the score of observation  $j$ .
- $+1$  and  $-1$  denote the positive and negative classes, respectively.
- $\pi$  is the prior probability that an observation is in the positive class.

If the output argument `ScoreTransform.Type` is `step`, then the quantities  $\max_{y_k=-1} s_k$

and  $\min_{y_k=+1} s_k$  correspond to the fields `LowerBound` and `UpperBound` of `ScoreTransform`, respectively.

## Constant Function

The constant function maps all scores in a sample to posterior probabilities 1 or 0.

If all observations have posterior probability 1, then they are expected to come from the positive class.

If all observations have posterior probability 0, then they are not expected to come from the positive class.

## Tips

- Here is one way to predict positive class posterior probabilities.
  - 1 Train an SVM classifier by passing the data to `fitcsvm`. The result is a trained SVM classifier, such as, `SVModel`, that stores the data. The software sets the

score transformation function property (`SVMMModel.ScoreTransformation`) to `none`.

- 2 Pass the trained SVM classifier `SVMMModel` to `fitSVMPosterior` or `fitPosterior`. The result, for example, `ScoreSVMMModel`, is the same, trained SVM classifier as `SVMMModel`, except the software sets `ScoreSVMMModel.ScoreTransformation` to the optimal score transformation function.

If you skip step 2, then `predict` returns the positive class score rather than the positive class posterior probability.

- 3 Pass the trained SVM classifier containing the optimal score transformation function (`ScoreSVMMModel`) and predictor data matrix to `predict`. The second column of the second output argument stores the positive class posterior probabilities corresponding to each row of the predictor data matrix.
- After fitting posterior probabilities, use the function `predict` and the trained model object to generate C code that predicts labels for new data. For details, see “Code Generation” on page 32-0 .

## Algorithms

If you reestimate the score-to-posterior-probability transformation function, that is, if you pass an SVM classifier to `fitPosterior` or `fitSVMPosterior` and its `ScoreTransform` property is not `none`, then the software:

- Displays a warning
- Resets the original transformation function to 'none' before estimating the new one

## Alternatives

You can also fit the posterior probability function using `fitSVMPosterior`. This function is similar to `fitPosterior`, except it is more broad since it accepts a wider range of SVM classifier types.

## References

- [1] Platt, J. “Probabilistic outputs for support vector machines and comparisons to regularized likelihood methods”. In: *Advances in Large Margin Classifiers*. Cambridge, MA: The MIT Press, 2000, pp. 61–74.

## See Also

`ClassificationSVM` | `fitSVMPosterior` | `fitcsvm` | `predict`

## fitPosterior

**Class:** CompactClassificationSVM

Fit posterior probabilities

### Syntax

```
ScoreSVMModel = fitPosterior(SVMModel, TBL, Y)
ScoreSVMModel = fitPosterior(SVMModel, X, Y)
[ScoreSVMModel, ScoreTransform] = fitPosterior(SVMModel, X, Y)
```

### Description

`ScoreSVMModel = fitPosterior(SVMModel, TBL, Y)` returns a trained support vector machine (SVM) classifier `ScoreSVMModel` containing the optimal score-to-posterior-probability transformation function for two-class learning. If you trained `SVMModel` using a table, then you must use a table as input for `fitPosterior`.

The software fits the appropriate score-to-posterior-probability transformation function using the SVM classifier `SVMModel`, and by conducting 10-fold cross validation using the stored predictor data (`SVMModel.X`) and the class labels (`SVMModel.Y`) as outlined in [1]. The transformation function computes the posterior probability that an observation is classified into the positive class (`SVMModel.Classnames(2)`).

- If the classes are inseparable, then the transformation function is the sigmoid function on page 32-2564.
- If the classes are perfectly separable, the transformation function is the step function on page 32-2564.
- In two-class learning, if one of the two classes has a relative frequency of 0, then the transformation function is the constant function on page 32-2683. `fitPosterior` is not appropriate for one-class learning.
- The software stores the optimal score-to-posterior-probability transformation function in `ScoreSVMModel.ScoreTransform`.



`ScoreSVMModel = fitPosterior(SVMModel, X, Y)` returns a trained support vector machine (SVM) classifier `ScoreSVMModel` containing the optimal score-to-posterior-probability transformation function for two-class learning. If you trained `SVMModel` using a matrix, then you must use a matrix as input for `fitPosterior`.

`[ScoreSVMModel, ScoreTransform] = fitPosterior(SVMModel, X, Y)` additionally returns the optimal score-to-posterior-probability transformation function parameters (`ScoreTransform`)

## Input Arguments

### **SVMModel** — Trained, compact SVM classifier

`CompactClassificationSVM` classifier

Trained, compact SVM classifier, specified as a `CompactClassificationSVM`.

### **TBL** — Sample data

table

Sample data, specified as a table. Each row of `TBL` corresponds to one observation, and each column corresponds to one predictor variable. `TBL` must contain all of the predictors used to train `SVMModel`. Optionally, `TBL` can contain an additional column for the response variable. Multi-column variables and cell arrays other than cell arrays of character vectors are not allowed.

If `TBL` contains the response variable, you do not need to specify `Y`. If `TBL` does not include the response variable, then the length of `Y` must be equal to the number of rows of `TBL`.

If the sample data used to train `SVMModel` is a table, then you must specify the input data for `fitPosterior` as a table.

Data Types: table

### **x** — Predictor data

matrix

Predictor data used to estimate the score-to-posterior-probability transformation function, specified as a matrix.

Each row of  $X$  corresponds to one observation (also known as an instance or example), and each column corresponds to one variable (also known as a feature).

The length of  $Y$  and the number of rows of  $X$  must be equal.

If you set `'Standardize', true` in `fitcsvm` to train `SVMMModel`, then the software standardizes the columns of  $X$  using the corresponding means in `SVMMModel.Mu` and standard deviations in `SVMMModel.Sigma`. If the software fits the transformation-function parameter estimates using standardized data, then the estimates might differ from estimation without standardized data.

Data Types: `double` | `single`

### **$y$ — Class labels**

categorical array | character array | logical vector | vector of numeric values | cell array of character vectors

Class labels used to estimate the score-to-posterior-probability transformation function, specified as a categorical or character array, logical or numeric vector, or cell array of character vectors.

If  $Y$  is a character array, then each element must correspond to one class label.

The length of  $Y$  and the number of rows of  $X$  must be equal.

## Output Arguments

### **ScoreSVMMModel — Trained, compact SVM classifier**

`CompactClassificationSVM` classifier

Trained, compact SVM classifier containing the estimated score-to-posterior-probability transformation function, returned as a `CompactClassificationSVM` classifier.

To estimate posterior probabilities, pass `ScoreSVMMModel` and predictor data to `predict`. If you set `'Standardize', true` in `fitcsvm` to train `SVMMModel`, then `predict` standardizes the columns of  $X$  using the corresponding means in `SVMMModel.Mu` and standard deviations in `SVMMModel.Sigma`.

### **ScoreTransform — Optimal score transformation function parameters**

structure array

Optimal score-to-posterior-probability transformation function parameters, returned as a structure array.

- If field `Type` is `sigmoid`, then `ScoreTransform` has the following other fields:
  - `Slope`: The value of  $A$  in the sigmoid function on page 32-2564
  - `Intercept`: The value of  $B$  in the sigmoid function
- If field `Type` is `step`, then `ScoreTransform` has the following other fields:
  - `PositiveClassProbability`: The value of  $\pi$  in the step function on page 32-2564. It represents the probability that an observation is in the positive class. Also, the posterior probability that an observation is in the positive class given that its score is in the interval  $(\text{LowerBound}, \text{UpperBound})$ .
  - `LowerBound`: The value  $\max_{y_i = 1} s_n$  in the step function. It represents the lower bound of the score interval that assigns observations with scores in the interval the posterior probability of being in the positive class `PositiveClassProbability`. Any observation with a score less than `LowerBound` has the posterior probability of being the positive class 0.
  - `UpperBound`: The value  $\min_{y_i = 1} s_n$  in the step function. It represents the upper bound of the score interval that assigns observations with scores in the interval the posterior probability of being in the positive class `PositiveClassProbability`. Any observation with a score greater than `UpperBound` has the posterior probability of being the positive class 1.
- If field `Type` is `constant`, then `ScoreTransform.PredictedClass` contains the name of the class prediction.

This result is the same as `SVMMModel.ClassNames`. The posterior probability of an observation being in `ScoreTransform.PredictedClass` is always 1.

## Examples

### Estimate Posterior Probabilities for New Data When Classes Are Inseparable

Load the `ionosphere` data set. Reserve 20 random observations of the data, and consider this set new data.

```
load ionosphere
n = size(X,1);
rng(1); % For reproducibility

indx = ~ismember([1:n],randsample(n,20)); % Indices for the training data
```

The classes of this data set are inseparable.

Train an SVM classifier using the training data. It is good practice to specify the class order and standardize the data.

```
SVMModel = fitcsvm(X(indx,:),Y(indx),'ClassNames',{'b','g'},...
    'Standardize',true);
```

`SVMModel` is a `ClassificationSVM` classifier.

Use the new data set to estimate the optimal score-to-posterior-probability transformation function for mapping scores to the posterior probability of an observation being classified as `g`. For efficiency, make a compact version of the SVM classifier `SVMModel`, and pass it and the new data to `fitPosterior`.

```
CompactSVMModel = compact(SVMModel);
[ScoreCSVMModel,ScoreParameters] = fitPosterior(CompactSVMModel,...
    X(~indx,:),Y(~indx));
```

```
ScoreTransform = ScoreCSVMModel.ScoreTransform
ScoreParameters
```

```
ScoreTransform =
```

```
    '@(S) sigmoid(S,-1.098922e+00,4.519963e-01)'
```

```
ScoreParameters =
```

```
    struct with fields:
```

```
        Type: 'sigmoid'
```

```
Slope: -1.0989
Intercept: 0.4520
```

ScoreTransform is the optimal score transform function. ScoreParameters is a structure array having three fields: the score transformation function name (Type), the sigmoid slope (Slope) and sigmoid intercept (Intercept) estimates.

Alternatively, you can pass SVMModel and the new data to fitSVMPosterior, but this does not have the benefit of efficiency.

Estimate the posterior probabilities that the observations in the new data are in class g.

```
[labels,postProbs] = predict(ScoreCSVMModel,X(~indx,:));
table(Y(~indx),labels,postProbs(:,2),...
      'VariableNames',{'TrueLabel','PredictedLabel','PosteriorProbability'})
```

```
ans =
```

```
20x3 table
```

TrueLabel	PredictedLabel	PosteriorProbability
'g'	'g'	0.78437
'b'	'b'	0.024588
'g'	'g'	0.82399
'b'	'b'	0.0061637
'b'	'b'	3.6118e-06
'b'	'b'	0.15688
'b'	'g'	0.9622
'b'	'b'	6.1348e-09
'b'	'b'	0.0019646
'g'	'g'	0.7251
'g'	'g'	0.70263
'b'	'b'	0.075298
'g'	'g'	0.90691
'g'	'g'	0.82849
'b'	'b'	0.051193
'g'	'g'	0.95331
'b'	'b'	0.0031877
'b'	'b'	0.16015
'g'	'g'	0.92008

```
'g'          'g'          0.91349
```

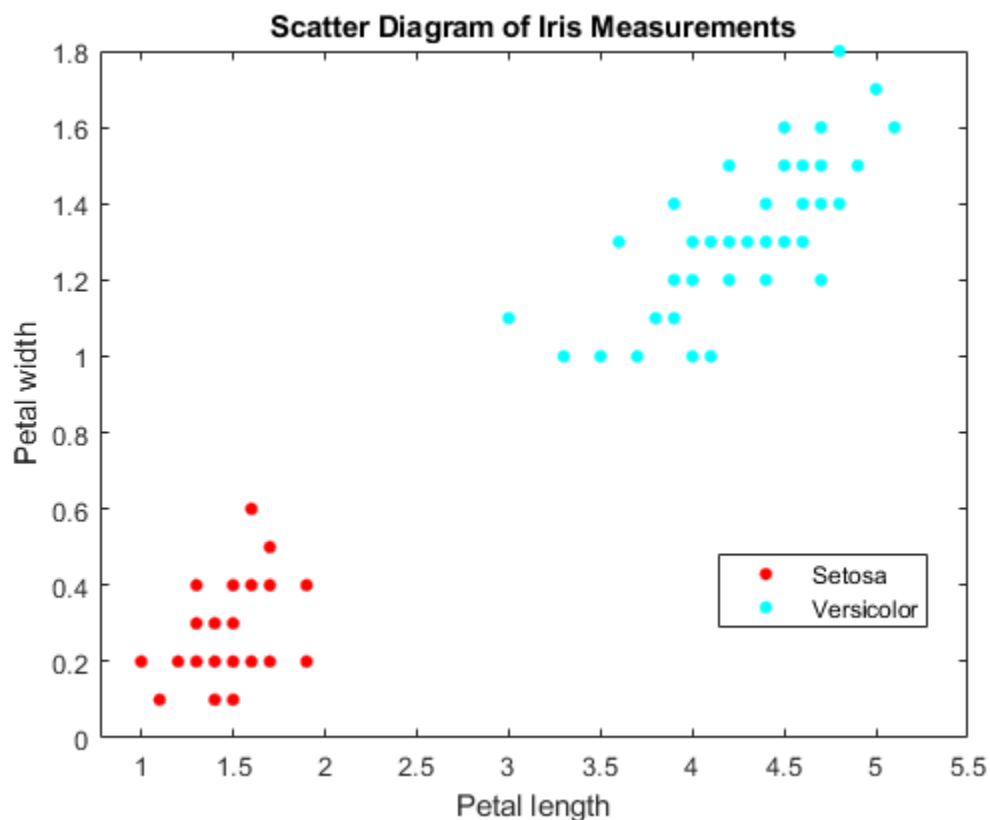
### Estimate Posterior Probabilities for New Data When Classes Are Separable

Load Fisher's iris data set. Use the petal lengths and widths, and remove the virginica species from the data. Reserve 10 random observations of the data, and consider this set new data.

```
load fisheriris
classKeep = ~strcmp(species, 'virginica');
X = meas(classKeep, 3:4);
Y = species(classKeep);

rng(1); % For reproducibility
indx1 = 1:numel(species);
indx2 = indx1(classKeep);
indx = ~ismember(indx2, randsample(indx2, 10)); % Indices for the training data

gscatter(X(indx, 1), X(indx, 2), Y(indx));
title('Scatter Diagram of Iris Measurements')
xlabel('Petal length')
ylabel('Petal width')
legend('Setosa', 'Versicolor')
```



The classes are perfectly separable. Therefore, the score-to-posterior-probability transformation function is a step function.

Train an SVM classifier. It is good practice to specify the class order and standardize the data.

```
SVMModel = fitcsvm(X(indx,:),Y(indx),...
    'ClassNames',{'setosa','versicolor'},'Standardize',true);
```

SVMModel is a ClassificationSVM classifier.

Use the new data set to estimate the optimal score-to-posterior-probability transformation function for mapping scores to the posterior probability of an observation

being classified as `versicolor`. For efficiency, make a compact version of the SVM classifier `SVMMModel`, and pass it and the new data to `fitPosterior`.

```
CompactSVMMModel = compact(SVMMModel);
[ScoreCSVMMModel, ScoreParameters] = fitPosterior(CompactSVMMModel, ...
    X(~indx, :), Y(~indx));

ScoreTransform = ScoreCSVMMModel.ScoreTransform

Warning: Classes are perfectly separated. The optimal score-to-posterior
transformation is a step function.

ScoreTransform =

    '@(S) step(S, -1.338450e+00, 2.012495e+00, 5.333333e-01)'
```

`fitPosterior` displays a warning whenever the classes are separable, and stores the step function in `ScoreSVMMModel.ScoreTransform`.

Display the score function type and its estimated values.

```
ScoreParameters

ScoreParameters =

    struct with fields:

        Type: 'step'
        LowerBound: -1.3385
        UpperBound: 2.0125
        PositiveClassProbability: 0.5333
```

`ScoreParameters` is a structure array having four fields:

- The score transformation function type (`Type`)
- The score corresponding to negative class boundary (`LowerBound`)
- The score corresponding to positive class boundary (`UpperBound`)
- The positive class probability (`PositiveClassProbability`)



Alternatively, you can pass `SVModel` and the new data to `fitSVPosterior`, but this does not have the benefit of efficiency.

Estimate the posterior probabilities that the observations in the new data are versicolor irises.

```
[labels,postProbs] = predict(ScoreCSVMModel,X(~indx,:));
table(Y(~indx),labels,postProbs(:,2),...
      'VariableNames',{'TrueLabel','PredictedLabel','PosteriorProbability'})
```

```
ans =
```

```
10x3 table
```

TrueLabel	PredictedLabel	PosteriorProbability
'setosa'	'setosa'	0
'setosa'	'setosa'	0
'setosa'	'setosa'	0
'setosa'	'setosa'	0
'setosa'	'setosa'	0
'setosa'	'setosa'	0
'setosa'	'setosa'	0
'setosa'	'setosa'	0
'versicolor'	'versicolor'	1
'versicolor'	'versicolor'	1

Since the classes are separable, the step function transforms the positive-class score to:

- 0, if the score is less than `ScoreParameters.LowerBound`
- 1, if the score is greater than `ScoreParameters.UpperBound`

- `ScoreParameters.PositiveClassProbability`, if the score is in the interval `[ScoreParameters.LowerBound, ScoreParameters.LowerBound]`

## Definitions

### Sigmoid Function

The sigmoid function that maps score  $s_j$  corresponding to observation  $j$  to the positive class posterior probability is

$$P(s_j) = \frac{1}{1 + \exp(As_j + B)}.$$

If the output argument `ScoreTransform.Type` is `sigmoid`, then parameters  $A$  and  $B$  correspond to the fields `Scale` and `Intercept` of `ScoreTransform`, respectively.

### Step Function

The step function that maps score  $s_j$  corresponding to observation  $j$  to the positive class posterior probability is

$$P(s_j) = \begin{cases} 0; & s < \max_{y_k=-1} s_k \\ \pi; & \max_{y_k=-1} s_k \leq s_j \leq \min_{y_k=+1} s_k, \\ 1; & s_j > \min_{y_k=+1} s_k \end{cases}$$

where:

- $s_j$  the score of observation  $j$ .
- $+1$  and  $-1$  denote the positive and negative classes, respectively.
- $\pi$  is the prior probability that an observation is in the positive class.

If the output argument `ScoreTransform.Type` is `step`, then the quantities  $\max_{y_k=-1} s_k$

and  $\min_{y_k=+1} s_k$  correspond to the fields `LowerBound` and `UpperBound` of `ScoreTransform`, respectively.

## Constant Function

The constant function maps all scores in a sample to posterior probabilities 1 or 0.

If all observations have posterior probability 1, then they are expected to come from the positive class.

If all observations have posterior probability 0, then they are not expected to come from the positive class.

## Tips

- Here is one way to predict positive class posterior probabilities.
  - 1 Train an SVM classifier by passing the data to `fitcsvm`. The result is a trained SVM classifier, such as, `SVMMODEL`, that stores the data. The software sets the score transformation function property (`SVMMODEL.ScoreTransformation`) to `none`.
  - 2 Pass the trained SVM classifier `SVMMODEL` to `fitSVMPosterior` or `fitPosterior`. The result, for example, `ScoreSVMMODEL`, is the same, trained SVM classifier as `SVMMODEL`, except the software sets `ScoreSVMMODEL.ScoreTransformation` to the optimal score transformation function.

If you skip step 2, then `predict` returns the positive class score rather than the positive class posterior probability.
  - 3 Pass the trained SVM classifier containing the optimal score transformation function (`ScoreSVMMODEL`) and predictor data matrix to `predict`. The second column of the second output argument stores the positive class posterior probabilities corresponding to each row of the predictor data matrix.
- After fitting posterior probabilities, use the function `predict` and the trained model object to generate C code that predicts labels for new data. For details, see “Code Generation” on page 32-0 .

## Algorithms

If you reestimate the score-to-posterior-probability transformation function, that is, if you pass an SVM classifier to `fitPosterior` or `fitSVMPosterior` and its `ScoreTransform` property is not `none`, then the software:

- Displays a warning
- Resets the original transformation function to `'none'` before estimating the new one

## Alternatives

You can also estimate the optimal score-to-posterior-probability function using `fitSVMPosterior`. This function is similar to `fitPosterior`, except it is more broad since it accepts a wider range of SVM classifier types.

## References

- [1] Platt, J. “Probabilistic outputs for support vector machines and comparisons to regularized likelihood methods”. In: *Advances in Large Margin Classifiers*. Cambridge, MA: The MIT Press, 2000, pp. 61–74.

## See Also

`CompactClassificationSVM` | `fitSVMPosterior` | `fitcsvm` | `predict`

# fitrensemble

Fit ensemble of learners for regression

## Syntax

```
Mdl = fitrensemble(Tbl, ResponseVarName)
```

```
Mdl = fitrensemble(Tbl, formula)
```

```
Mdl = fitrensemble(Tbl, Y)
```

```
Mdl = fitrensemble(X, Y)
```

```
Mdl = fitrensemble( ____, Name, Value)
```

## Description

`Mdl = fitrensemble(Tbl, ResponseVarName)` returns the trained regression ensemble model object (`Mdl`) that contains the results of boosting 100 regression trees using LSBoost and the predictor and response data in the table `Tbl`. `ResponseVarName` is the name of the response variable in `Tbl`.

`Mdl = fitrensemble(Tbl, formula)` applies `formula` to fit the model to the predictor and response data in the table `Tbl`. `formula` is an explanatory model of the response and a subset of predictor variables in `Tbl` used to fit `Mdl`. For example, `'Y~X1+X2+X3'` fits the response variable `Tbl.Y` as a function of the predictor variables `Tbl.X1`, `Tbl.X2`, and `Tbl.X3`.

`Mdl = fitrensemble(Tbl, Y)` treats all variables in the table `Tbl` as predictor variables. `Y` is the vector of responses that is not in `Tbl`.

`Mdl = fitrensemble(X, Y)` uses the predictor data in the matrix `X` and response data in the vector `Y`.

`Mdl = fitrensemble( ____, Name, Value)` uses additional options specified by one or more `Name, Value` pair arguments and any of the input arguments in the previous syntaxes. For example, you can specify the number of learning cycles, the ensemble-aggregation method, or to implement 10-fold cross-validation.

## Examples

### Train Regression Ensemble

Create a regression ensemble that predicts the fuel economy of a car given the number of cylinders, volume displaced by the cylinders, horsepower, and weight. Then, train another ensemble using fewer predictors. Compare the in-sample predictive accuracies of the ensembles.

Load the `carsmall` data set. Store the variables to be used in training in a table.

```
load carsmall
Tbl = table(Cylinders,Displacement,Horsepower,Weight,MPG);
```

Train a regression ensemble.

```
Mdl1 = fitensemble(Tbl,'MPG');
```

`Mdl1` is a `RegressionEnsemble` model. Some notable characteristics of `Mdl1` are:

- The ensemble-aggregation algorithm is `'LSBoost'`.
- Because the ensemble-aggregation method is a boosting algorithm, regression trees that allow a maximum of 10 splits compose the ensemble.
- One hundred trees compose the ensemble.

Because `MPG` is a variable in the MATLAB® Workspace, you can obtain the same result by entering

```
Mdl1 = fitensemble(Tbl,MPG);
```

Use the trained regression ensemble to predict the fuel economy for a four-cylinder car with a 200-cubic inch displacement, 150 horsepower, and weighing 3000 lbs.

```
pMPG = predict(Mdl1,[4 200 150 3000])
```

```
pMPG =  
25.6467
```

Train a new ensemble using all predictors in `Tbl` except `Displacement`.

```
formula = 'MPG ~ Cylinders + Horsepower + Weight';
Mdl2 = fitrensemble(Tbl, formula);
```

Compare the resubstitution MSEs between Mdl1 and Mdl2.

```
mse1 = resubLoss(Mdl1)
mse2 = resubLoss(Mdl2)
```

```
mse1 =
    0.3096
```

```
mse2 =
    0.5861
```

The in-sample MSE for the ensemble that trains on all predictors is lower.

### Estimate Generalization Error of Boosting Ensemble

Estimate the generalization error of an ensemble of boosted regression trees.

Load the `carsmall` data set. Choose the number of cylinders, volume displaced by the cylinders, horsepower, and weight as predictors of fuel economy.

```
load carsmall
X = [Cylinders Displacement Horsepower Weight];
```

Cross-validate an ensemble of regression trees using 10-fold cross-validation. Using a decision tree template, specify that each tree should be a split once only.

```
rng(1); % For reproducibility
t = templateTree('MaxNumSplits',1);
Mdl = fitrensemble(X,MPG, 'Learners',t, 'CrossVal','on');
```

Mdl is a `RegressionPartitionedEnsemble` model.

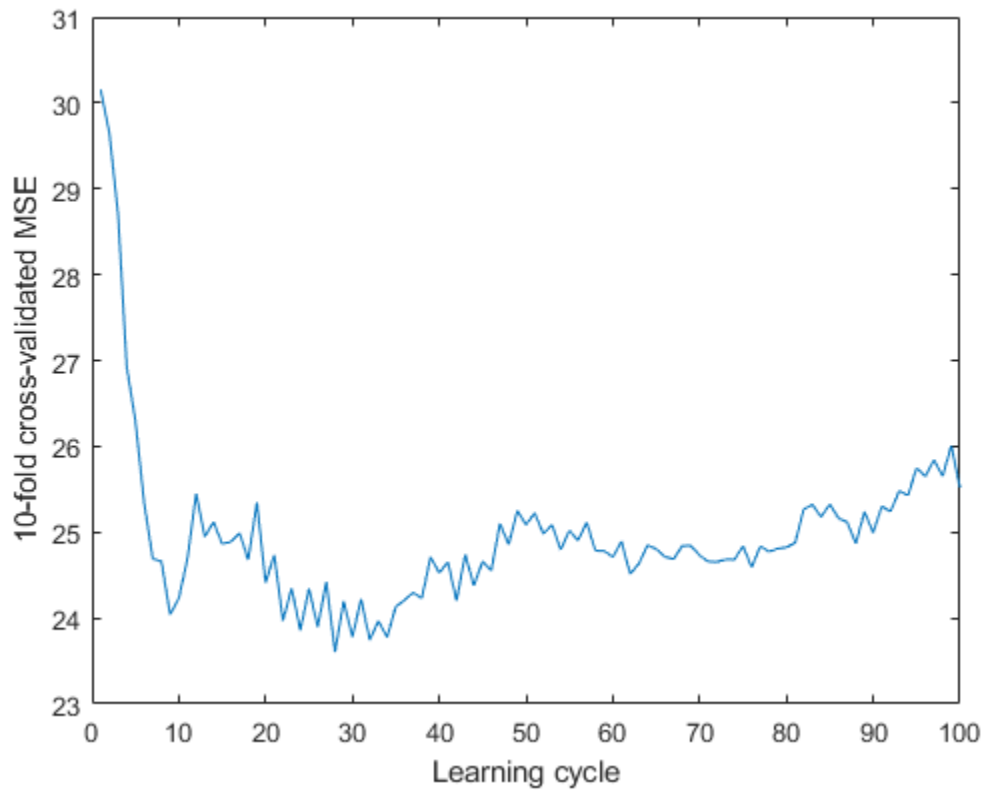
Plot the cumulative, 10-fold cross-validated, mean-squared error (MSE). Display the estimated generalization error of the ensemble.

```
kflc = kfoldLoss(Mdl,'Mode','cumulative');  
figure;  
plot(kflc);  
ylabel('10-fold cross-validated MSE');  
xlabel('Learning cycle');
```

```
estGenError = kflc(end)
```

```
estGenError =
```

```
25.5184
```





kfoldLoss returns the generalization error by default. However, plotting the cumulative loss allows you to monitor how the loss changes as weak learners accumulate in the ensemble.

The ensemble achieves an MSE of around 23.5 after accumulating about 30 weak learners.

If you are satisfied with the generalization error of the ensemble, then, to create a predictive model, train the ensemble again using all of the settings except cross-validation. However, it is good practice to tune hyperparameters such as the maximum number of decision splits per tree and the number of learning cycles..

## Optimize Regression Ensemble

This example shows how to optimize hyperparameters automatically using fitrensemble. The example uses the carsmall data.

Load the data.

```
load carsmall
```

Find hyperparameters that minimize five-fold cross-validation loss by using automatic hyperparameter optimization.

For reproducibility, set the random seed and use the 'expected-improvement-plus' acquisition function.

```
rng(1)
Mdl = fitrensemble([Horsepower,Weight],MPG,'OptimizeHyperparameters','auto',...
    'HyperparameterOptimizationOptions',struct('AcquisitionFunctionName',...
    'expected-improvement-plus'))
```

Iter	Eval result	Objective	Objective runtime	BestSoFar (observed)	BestSoFar (estim.)	Method
1	Best	3.0003	3.9637	3.0003	3.0003	LSBoost
2	Accept	3.1411	1.572	3.0003	3.0109	Bag
3	Accept	4.8671	0.8406	3.0003	3.0008	LSBoost
4	Accept	4.1814	13.986	3.0003	3.0826	Bag
5	Best	2.9281	1.7303	2.9281	2.9342	LSBoost
6	Accept	6.3689	0.69449	2.9281	2.9284	LSBoost

7	Accept	2.931	0.72856	2.9281	2.9286	LSBoost
8	Accept	4.1823	6.6151	2.9281	2.9283	LSBoost
9	Best	2.9093	0.53895	2.9093	2.9094	Bag
10	Accept	3.0763	0.42103	2.9093	2.9096	LSBoost
11	Accept	2.9689	1.9635	2.9093	2.9328	LSBoost
12	Accept	4.1823	0.5758	2.9093	2.9279	LSBoost
13	Best	2.8562	26.155	2.8562	2.8368	Bag
14	Accept	3.0078	0.48217	2.8562	2.8459	Bag
15	Accept	2.9582	25.074	2.8562	2.8483	Bag
16	Accept	3.2591	1.3213	2.8562	2.8559	LSBoost
17	Accept	3.0629	2.4848	2.8562	2.8559	Bag
18	Accept	3.6183	2.0179	2.8562	2.8559	LSBoost
19	Accept	3.6616	12.441	2.8562	2.856	LSBoost
20	Accept	6.0321	12.715	2.8562	2.8558	LSBoost
=====						
Iter	Eval	Objective	Objective	BestSoFar	BestSoFar	Method
	result		runtime	(observed)	(estim.)	
=====						
21	Accept	2.9087	0.66871	2.8562	2.856	Bag
22	Accept	3.3691	23.482	2.8562	2.856	LSBoost
23	Accept	2.9046	0.84975	2.8562	2.8705	Bag
24	Accept	2.9699	1.8478	2.8562	2.8699	LSBoost
25	Best	2.8554	28.213	2.8554	2.8558	Bag
26	Accept	2.8708	29.434	2.8554	2.8577	Bag
27	Best	2.8421	26.374	2.8421	2.8508	Bag
28	Accept	2.8425	32.783	2.8421	2.8464	Bag
29	Accept	2.9523	34.215	2.8421	2.8463	LSBoost
30	Accept	2.8525	32.975	2.8421	2.8468	Bag

Optimization completed.

MaxObjectiveEvaluations of 30 reached.

Total function evaluations: 30

Total elapsed time: 466.1839 seconds.

Total objective function evaluation time: 327.1634

Best observed feasible point:

Method	NumLearningCycles	LearnRate	MinLeafSize
Bag	487	NaN	13

Observed objective function value = 2.8421

Estimated objective function value = 2.8468

Function evaluation time = 26.3736

Best estimated feasible point (according to models):

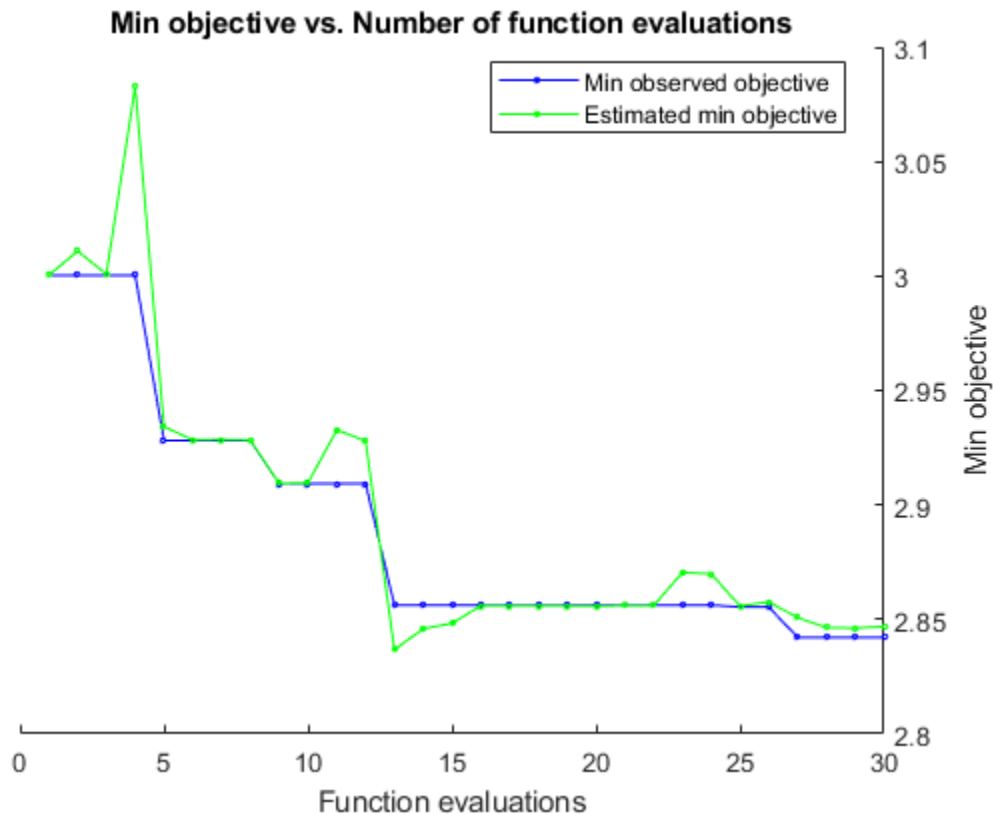
Method	NumLearningCycles	LearnRate	MinLeafSize
Bag	490	NaN	13

Estimated objective function value = 2.8468

Estimated function evaluation time = 28.0134

Mdl =

```
classreg.learning.regr.RegressionBaggedEnsemble
  ResponseName: 'Y'
  CategoricalPredictors: []
  ResponseTransform: 'none'
  NumObservations: 94
  HyperparameterOptimizationResults: [1x1 BayesianOptimization]
  NumTrained: 490
  Method: 'Bag'
  LearnerNames: {'Tree'}
  ReasonForTermination: 'Terminated normally after completing the request'
  FitInfo: []
  FitInfoDescription: 'None'
  Regularization: []
  FResample: 1
  Replace: 1
  UseObsForLearner: [94x490 logical]
```



The optimization searched over the methods for regression (`Bag` and `LSBoost`), over `NumLearningCycles`, over the `LearnRate` for `LSBoost`, and over the tree learner `MinLeafSize`. The output is the ensemble regression with the minimum estimated cross-validation loss.

### Create Predictive Ensemble Using Cross-Validation

One way to create an ensemble of boosted regression trees that has satisfactory predictive performance is to tune the decision tree-complexity level using cross-validation. While searching for an optimal complexity level, tune the learning rate to minimize the number of learning cycles.

Load the `carsmall` data set. Choose the number of cylinders, volume displaced by the cylinders, horsepower, and weight as predictors of fuel economy.

```
load carsmall
Tbl = table(Cylinders,Displacement,Horsepower,Weight,MPG);
```

To search for the optimal tree-complexity level:

- 1 Cross-validate a set of ensembles. Exponentially increase the tree-complexity level for subsequent ensembles from decision stump (one split) to at most  $n - 1$  splits.  $n$  is the sample size. Also, vary the learning rate for each ensemble between 0.1 to 1.
- 2 Estimate the cross-validated mean-squared error (MSE) for each ensemble.
- 3 For tree-complexity level  $j$ ,  $j = 1 \dots J$ , compare the cumulative, cross-validated MSE of the ensembles by plotting them against number of learning cycles. Plot separate curves for each learning rate on the same figure.
- 4 Choose the curve that achieves the minimal MSE, and note the corresponding learning cycle and learning rate.

Cross-validate a deep regression tree and a stump. Because the data contain missing values, use surrogate splits. These regression trees serve as benchmarks.

```
rng(1); % For reproducibility
MdlDeep = fitrtree(Tbl, 'MPG', 'CrossVal', 'on', 'MergeLeaves', 'off', ...
    'MinParentSize', 1, 'Surrogate', 'on');
MdlStump = fitrtree(Tbl, 'MPG', 'MaxNumSplits', 1, 'CrossVal', 'on', ...
    'Surrogate', 'on');
```

Cross-validate an ensemble of 150 boosted regression trees using 5-fold cross-validation. Using a tree template:

- Vary the maximum number of splits using the values in the sequence  $\{2^0, 2^1, \dots, 2^m\}$ .  $m$  is such that  $2^m$  is no greater than  $n - 1$ .
- Turn on surrogate splits.

For each variant, adjust the learning rate using each value in the set  $\{0.1, 0.25, 0.5, 1\}$ ;

```
n = size(Tbl,1);
m = floor(log2(n - 1));
learnRate = [0.1 0.25 0.5 1];
numLR = numel(learnRate);
maxNumSplits = 2.^(0:m);
```

```

numMNS = numel(maxNumSplits);
numTrees = 150;
Mdl = cell(numMNS,numLR);

for k = 1:numLR;
    for j = 1:numMNS;
        t = templateTree('MaxNumSplits',maxNumSplits(j),'Surrogate','on');
        Mdl{j,k} = fitrensemble(Tbl,'MPG','NumLearningCycles',numTrees,...
            'Learners',t,'KFold',5,'LearnRate',learnRate(k));
    end;
end;

```

Estimate the cumulative, cross-validated MSE of each ensemble.

```

kflAll = @(x)kfoldLoss(x,'Mode','cumulative');
errorCell = cellfun(kflAll,Mdl,'Uniform',false);
error = reshape(cell2mat(errorCell),[numTrees numel(maxNumSplits) numel(learnRate)]);
errorDeep = kfoldLoss(MdlDeep);
errorStump = kfoldLoss(MdlStump);

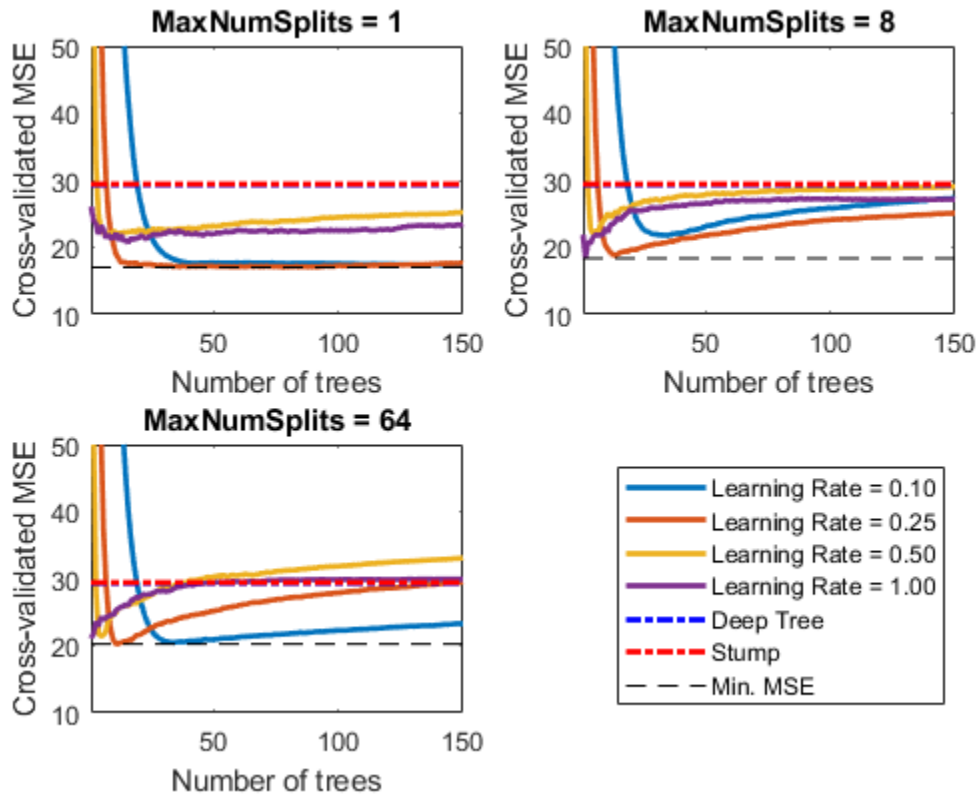
```

Plot how the cross-validated MSE behaves as the number of trees in the ensemble increases. Plot the curves with respect to learning rate on the same plot, and plot separate plots for varying tree-complexity levels. Choose a subset of tree complexity levels to plot.

```

mnsPlot = [1 round(numel(maxNumSplits)/2) numel(maxNumSplits)];
figure;
for k = 1:3;
    subplot(2,2,k);
    plot(squeeze(error(:,mnsPlot(k),:)), 'LineWidth',2);
    axis tight;
    hold on;
    h = gca;
    plot(h.XLim,[errorDeep errorDeep], '-.b', 'LineWidth',2);
    plot(h.XLim,[errorStump errorStump], '-.r', 'LineWidth',2);
    plot(h.XLim,min(min(error(:,mnsPlot(k),:)).*[1 1], '--k'));
    h.YLim = [10 50];
    xlabel 'Number of trees';
    ylabel 'Cross-validated MSE';
    title(sprintf('MaxNumSplits = %0.3g', maxNumSplits(mnsPlot(k))));
    hold off;
end;
hL = legend([cellstr(num2str(learnRate),'Learning Rate = %0.2f');...
    'Deep Tree'; 'Stump'; 'Min. MSE']);
hL.Position(1) = 0.6;

```



Each curve contains a minimum cross-validated MSE occurring at the optimal number of trees in the ensemble.

Identify the maximum number of splits, number of trees, and learning rate that yields the lowest MSE overall.

```
[minErr,minErrIdxLin] = min(error(:));
[idxNumTrees,idxMNS,idxLR] = ind2sub(size(error),minErrIdxLin);

fprintf('\nMin. MSE = %0.5f',minErr)
fprintf('\nOptimal Parameter Values:\nNum. Trees = %d',idxNumTrees);
fprintf('\nMaxNumSplits = %d\nLearning Rate = %0.2f\n',...
        maxNumSplits(idxMNS),learnRate(idxLR))
```

```
Min. MSE = 17.01148
Optimal Parameter Values:
Num. Trees = 38
MaxNumSplits = 4
Learning Rate = 0.10
```

Create a predictive ensemble based on the optimal hyperparameters and the entire training set.

```
tFinal = templateTree('MaxNumSplits',maxNumSplits(idxMNS),'Surrogate','on');
MdlFinal = fitrensemble(Tbl,'MPG','NumLearningCycles',idxNumTrees,...
    'Learners',tFinal,'LearnRate',learnRate(idxLR))
```

```
MdlFinal =
```

```
classreg.learning.regr.RegressionEnsemble
    PredictorNames: {1x4 cell}
    ResponseName: 'MPG'
    CategoricalPredictors: []
    ResponseTransform: 'none'
    NumObservations: 94
    NumTrained: 38
    Method: 'LSBoost'
    LearnerNames: {'Tree'}
    ReasonForTermination: 'Terminated normally after completing the requested number o
    FitInfo: [38x1 double]
    FitInfoDescription: {2x1 cell}
    Regularization: []
```

MdlFinal is a RegressionEnsemble. To predict the fuel economy of a car given its number of cylinders, volume displaced by the cylinders, horsepower, and weight, pass the predictor data and MdlFinal to predict.

- “Supervised Learning Workflow and Algorithms” on page 18-2
- “Framework for Ensemble Learning” on page 18-41



## Input Arguments

### **Tbl** — Sample data

table

Sample data used to train the model, specified as a table. Each row of `Tbl` corresponds to one observation, and each column corresponds to one predictor variable. `Tbl` can contain one additional column for the response variable. Multi-column variables and cell arrays other than cell arrays of character vectors are not allowed.

- If `Tbl` contains the response variable and you want to use all remaining variables as predictors, then specify the response variable using `ResponseVarName`.
- If `Tbl` contains the response variable, and you want to use a subset of the remaining variables only as predictors, then specify a formula using `formula`.
- If `Tbl` does not contain the response variable, then specify the response data using `Y`. The length of response variable and the number of rows of `Tbl` must be equal.

---

**Note** To save memory and execution time, supply `X` and `Y` instead of `Tbl`.

---

Data Types: `table`

### **ResponseVarName** — Response variable name

name of response variable in `Tbl`

Response variable name, specified as the name of the response variable in `Tbl`.

You must specify `ResponseVarName` as a character vector. For example, if `Tbl.Y` is the response variable, then specify `ResponseVarName` as `'Y'`. Otherwise, `fitrensemble` treats all columns of `Tbl` as predictor variables.

Data Types: `char`

### **formula** — Explanatory model of response and subset of predictor variables

character vector

Explanatory model of the response and a subset of the predictor variables, specified as a character vector in the form of `'Y~X1+X2+X3'`. In this form, `Y` represents the response variable, and `X1`, `X2`, and `X3` represent the predictor variables. The variables must be variable names in `Tbl` (`Tbl.Properties.VariableNames`).

To specify a subset of variables in `Tbl` as predictors for training the model, use a formula. If you specify a formula, then the software does not use any variables in `Tbl` that do not appear in `formula`.

Data Types: `char`

### **x** — Predictor data

numeric matrix

Predictor data, specified as numeric matrix.

Each row corresponds to one observation, and each column corresponds to one predictor variable.

The length of `Y` and the number of rows of `X` must be equal.

To specify the names of the predictors in the order of their appearance in `X`, use the `PredictorNames` name-value pair argument.

Data Types: `single` | `double`

### **y** — Response

numeric vector

Response, specified as a numeric vector. Each element in `Y` is the response to the observation in the corresponding row of `X` or `Tbl`. The length of `Y` and the number of rows of `X` or `Tbl` must be equal.

Data Types: `single` | `double`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

---

**Note** You cannot use any cross-validation name-value pair along with `OptimizeHyperparameters`. You can modify the cross-validation for `OptimizeHyperparameters` only by using the `HyperparameterOptimizationOptions` name-value pair.

---

Example: `'NumLearningCycles', 500, 'Method', 'Bag', 'Learners', templateTree(), 'CrossVal', 'on'` cross-validates an ensemble of 500 bagged regression trees using 10-fold cross-validation.

### General Ensemble Options

#### Method — Ensemble-aggregation method

'LSBoost' (default) | 'Bag'

Ensemble-aggregation method, specified as the comma separated pair consisting of 'Method' and 'Bag' or 'LSBoost'.

Value	Description
'Bag'	Bootstrap aggregating or bagging (for example, random forest)
'LSBoost'	Least-squares boosting or LSBoost

Example: `'Method', 'Bag'`

Data Types: char

#### NumLearningCycles — Number of ensemble learning cycles

100 (default) | positive integer

Number of ensemble learning cycles, specified as a positive integer. At every learning cycle, the software trains one weak learner for every template object in `Learners`. Consequently, the software trains `NumLearningCycles* numel(Learners)` learners.

The software composes the ensemble using all trained learners and stores them in `Mdl.Trained`.

For more details, see “Tips” on page 32-2595.

Example: `'NumLearningCycles', 500`

Data Types: single | double

#### Learners — Weak learners to use in ensemble

'tree' (default) | regression tree template object | cell vector of regression tree template objects

Weak learners to use in the ensemble, specified as 'tree', a regression tree template object, or a cell vector of regression tree template objects.

- 'tree' specifies to train an ensemble of regression trees using default regression tree options.
  - For bagging, `fitensemble` grows deep trees by default. For a complete list of defaults, see `templateTree`.
  - For LSBoost, `fitensemble` grows shallow trees by default. That is, 'Learners' is `templateTree('MaxNumSplits',10)`.
- A regression tree template object specifies how to grow all trees in the ensemble. You can create a regression tree template using `templateTree`.
- A cell vector of  $m$  regression tree templates specifies to grow  $m$  regression trees per learning cycle (see `NumLearningCycles`). For example, for an ensemble composed of two types of regression trees, supply `{t1 t2}`, where `t1` and `t2` are regression tree templates returned by `templateTree`.

`fitensemble` sets the maximum number of splits for regression tree weak learners to 10 when both of these conditions occur:

- You do not set a value for the 'MaxNumSplits' property of the weak learners.
- You use a boosting algorithm as the ensemble-aggregation method for `fitensemble`.

For example, if you specify 'Learners', `templateTree()` and 'Method', 'LSBoost', then `fitensemble` sets the maximum number of splits of the tree weak learners to 10.

Otherwise, `fitensemble` defers to the regression tree template object to choose any required default values for the 'Learners' name-value pair argument.

For details on the number of learners to train, see `NumLearningCycles` and “Tips” on page 32-2595.

Example: `'Learners',templateTree('MaxNumSplits',5)`

Data Types: char

### **NPrint** — Printout frequency

'off' (default) | positive integer

Printout frequency, specified as the comma-separated pair consisting of 'NPrint' and a positive integer or 'off'.

To track the number of *weak learners* or  *folds* that `fitrensemble` trained so far, specify a positive integer. That is, if you specify the positive integer  $m$ :

- Without also specifying any cross-validation option (for example, `CrossVal`), then `fitrensemble` displays a message to the command line every time it completes training  $m$  weak learners.
- And a cross-validation option, then `fitrensemble` displays a message to the command line every time it finishes training  $m$  folds.

If you specify 'off', then `fitrensemble` does not display a message when it completes training weak learners.

---

**Tip** When training an ensemble of many weak learners on a large data set, specify a positive integer for `NPrint`.

---

Example: `'NPrint',5`

Data Types: `single` | `double` | `char`

### **CategoricalPredictors** — Categorical predictors list

vector of positive integers | logical vector | character matrix | cell array of character vectors | 'all'

Categorical predictors list, specified as the comma-separated pair consisting of 'CategoricalPredictors' and one of these values:

Value	Description
Vector of positive integers	An entry in the vector is the index value corresponding to the column of the predictor data (X or Tbl) that contains a categorical variable.
Logical vector	A true entry means that the corresponding column of predictor data (X or Tbl) is a categorical variable.
Character matrix	Each row of the matrix is the name of a predictor variable. The names must match the entries in <code>PredictorNames</code> . Pad the names with extra blanks so each row of the character matrix has the same length.

Value	Description
Cell array of character vectors	Each element in the array is the name of a predictor variable. The names must match the entries in <code>PredictorNames</code> .
'all'	All predictors are categorical.

By default, if the predictor data is in a table (`Tbl`), `fitrensemble` assumes that a variable is categorical if it contains logical values, categorical values, or a cell array of character vectors. If the predictor data is a matrix (`X`), `fitrensemble` assumes all predictors are continuous. To identify any categorical predictors when the data is a matrix, use the `'CategoricalPredictors'` name-value pair argument.

Example: `'CategoricalPredictors', 'all'`

Data Types: `single` | `double` | `logical` | `char` | `cell`

### **PredictorNames** — Predictor variable names

cell array of unique character vectors

Predictor variable names, specified as the comma-separated pair consisting of `'PredictorNames'` and a cell array of unique character vectors. The functionality of `'PredictorNames'` depends on the way you supply the training data.

- If you supply `X` and `Y`, then you can use `'PredictorNames'` to give the predictor variables in `X` names.
  - The order of the names in `PredictorNames` must correspond to the column order of `X`. That is, `PredictorNames{1}` is the name of `X(:, 1)`, `PredictorNames{2}` is the name of `X(:, 2)`, and so on. Also, `size(X, 2)` and `numel(PredictorNames)` must be equal.
  - By default, `PredictorNames` is `{'x1', 'x2', ...}`.
- If you supply `Tbl`, then you can use `'PredictorNames'` to choose which predictor variables to use in training. That is, `fitrensemble` uses the predictor variables in `PredictorNames` and the response only in training.
  - `PredictorNames` must be a subset of `Tbl.Properties.VariableNames` and cannot include the name of the response variable.
  - By default, `PredictorNames` contains the names of all predictor variables.
  - It good practice to specify the predictors for training using one of `'PredictorNames'` or formula only.

Example: 'PredictorNames',  
{ 'SepalLength', 'SepalWidth', 'PedalLength', 'PedalWidth' }

Data Types: cell

### **ResponseName** — Response variable name

'Y' (default) | character vector

Response variable name, specified as the comma-separated pair consisting of 'ResponseName' and a character vector.

- If you supply Y, then you can use 'ResponseName' to specify a name for the response variable.
- If you supply ResponseVarName or formula, then you cannot use 'ResponseName'.

Example: 'ResponseName', 'response'

Data Types: char

### **ResponseTransform** — Response transformation

'none' (default) | function handle

Response transformation, specified as the comma-separated pair consisting of 'ResponseTransform' and either 'none' or a function handle. The default is 'none', which means  $@(x)x$ , or no transformation. For a MATLAB function or a function you define, use its function handle. The function handle must accept a vector (the original response values) and return a vector of the same size (the transformed response values).

Example: 'ResponseTransform', 'none'

Data Types: char | function\_handle

## **Cross-Validation Options**

### **CrossVal** — Cross-validation flag

'off' (default) | 'on'

Cross-validation flag, specified as the comma-separated pair consisting of 'Crossval' and 'on' or 'off'.

If you specify 'on', then the software implements 10-fold cross-validation.

To override this cross-validation setting, use one of these name-value pair arguments: `CVPartition`, `Holdout`, `KFold`, or `Leaveout`. To create a cross-validated model, you can use one cross-validation name-value pair argument at a time only.

Alternatively, cross-validate later by passing `Mdl` to `crossval` or `crossval`.

Example: `'Crossval', 'on'`

Data Types: `char`

### **CVPartition — Cross-validation partition**

`[]` (default) | `cvpartition` partition object

Cross-validation partition, specified as the comma-separated pair consisting of `'CVPartition'` and a `cvpartition` partition object as created by `cvpartition`. The partition object specifies the type of cross-validation, and also the indexing for training and validation sets.

To create a cross-validated model, you can use one of these four name-value pair arguments only: `CVPartition`, `Holdout`, `KFold`, or `Leaveout`.

Example: Suppose you create a random partition for 5-fold cross-validation on 500 observations by using `cvp = cvpartition(500, 'KFold', 5)`. Then, you can specify the cross-validation partition for `fitcecoc` as `'CVPartition', cvp`.

### **Holdout — Fraction of data for holdout validation**

scalar value in the range (0,1)

Fraction of data used for holdout validation, specified as the comma-separated pair consisting of `'Holdout'` and a scalar value in the range (0,1). If you specify `'Holdout', p`, then the software:

- 1 Randomly reserves  $p*100\%$  of the data as validation data, and trains the model using the rest of the data
- 2 Stores the compact, trained model in the `Trained` property of the cross-validated model.

To create a cross-validated model, you can use one of these four name-value pair arguments only: `CVPartition`, `Holdout`, `KFold`, or `Leaveout`.

Example: `'Holdout', 0.1`

Data Types: `double` | `single`



**KFold — Number of folds**

10 (default) | positive integer value greater than 1

Number of folds to use in a cross-validated model, specified as the comma-separated pair consisting of 'KFold' and a positive integer value greater than 1. If you specify, e.g., 'KFold',  $k$ , then the software:

- 1 Randomly partitions the data into  $k$  sets
- 2 For each set, reserves the set as validation data, and trains the model using the other  $k - 1$  sets
- 3 Stores the  $k$  compact, trained models in the cells of a  $k$ -by-1 cell vector in the `Trained` property of the cross-validated model.

To create a cross-validated model, you can use one of these four name-value pair arguments only: `CVPartition`, `Holdout`, `KFold`, or `Leaveout`.

Example: 'KFold', 5

Data Types: `single` | `double`

**Leaveout — Leave-one-out cross-validation flag**

'off' (default) | 'on'

Leave-one-out cross-validation flag, specified as the comma-separated pair consisting of 'Leaveout' and 'on' or 'off'. If you specify 'Leaveout', 'on', then, for each of the  $n$  observations, where  $n$  is `size(Mdl.X, 1)`, the software:

- 1 Reserves the observation as validation data, and trains the model using the other  $n - 1$  observations
- 2 Stores the  $n$  compact, trained models in the cells of an  $n$ -by-1 cell vector in the `Trained` property of the cross-validated model.

To create a cross-validated model, you can use one of these four name-value pair arguments only: `CVPartition`, `Holdout`, `KFold`, or `Leaveout`.

Example: 'Leaveout', 'on'

Data Types: `char`

**Other Regression Options****weights — Observation weights**

numeric vector of positive values | character vector

Observation weights, specified as the comma-separated pair consisting of `'Weights'` and a numeric vector of positive values or name of a variable in `Tbl`. The software weighs the observations in each row of `X` or `Tbl` with the corresponding value in `Weights`. The size of `Weights` must equal the number of rows of `X` or `Tbl`.

If you specify the input data as a table `Tbl`, then `Weights` can be the name of a variable in `Tbl` that contains a numeric vector. In this case, you must specify `Weights` as a character vector. For example, if the weights vector `W` is stored as `Tbl.W`, then specify it as `'W'`. Otherwise, the software treats all columns of `Tbl`, including `W`, as predictors or the response when training the model.

The software normalizes `Weights` to sum up to the value of the prior probability in the respective class.

By default, `Weights` is ones  $(n, 1)$ , where  $n$  is the number of observations in `X` or `Tbl`.

Data Types: `double` | `single` | `char`

### Sampling Options

#### **FResample** — Fraction of training set to resample

1 (default) | positive scalar in  $(0,1]$

Fraction of the training set to resample for every weak learner, specified as the comma-separated pair consisting of `'FResample'` and a positive scalar in  $(0,1]$ .

To use `'FResample'`, specify `'bag'` for `Method` or set `Resample` to `'on'`.

Example: `'FResample', 0.75`

Data Types: `single` | `double`

#### **Replace** — Flag indicating to sample with replacement

`'on'` (default) | `'off'`

Flag indicating sampling with replacement, specified as the comma-separated pair consisting of `'Replace'` and `'off'` or `'on'`.

- For `'on'`, the software samples the training observations with replacement.
- For `'off'`, the software samples the training observations without replacement. If you set `Resample` to `'on'`, then the software samples training observations assuming

uniform weights. If you also specify a boosting method, then the software boosts by reweighting observations.

Unless you set `Method` to `'bag'` or set `Resample` to `'on'`, `Replace` has no effect.

Example: `'Replace', 'off'`

Data Types: `char`

### **Resample** — Flag indicating to resample

`'off' | 'on'`

Flag indicating to resample, specified as the comma-separated pair consisting of `'Resample'` and `'off'` or `'on'`.

- If `Method` is any boosting method, then:
  - `'Resample', 'on'` specifies to sample training observations using updated weights as the multinomial sampling probabilities.
  - `'Resample', 'off'` specifies to reweight observations at every learning iteration. This setting is the default.
- If `Method` is `'bag'`, then `'Resample'` must be `'on'`. The software resamples a fraction of the training observations (see `FResample`) with or without replacement (see `Replace`).

### **LSBoost Method Options**

#### **LearnRate** — Learning rate for shrinkage

`1` (default) | numeric scalar in  $(0,1]$

Learning rate for shrinkage, specified as the comma-separated pair consisting of a numeric scalar in the interval  $(0,1]$ .

To train an ensemble using shrinkage, set `LearnRate` to a value less than 1, for example, `0.1` is a popular choice. Training an ensemble using shrinkage requires more learning iterations, but often achieves better accuracy.

Example: `'LearnRate', 0.1`

Data Types: `single | double`

## Hyperparameter Optimization

### OptimizeHyperparameters — Parameters to optimize

'none' (default) | 'auto' | 'all' | cell array of eligible parameter names | vector of `optimizableVariable` objects

Parameters to optimize, specified as:

- 'none' — Do not optimize.
- 'auto' — Use {'Method', 'NumLearningCycles', 'LearnRate'} along with the default parameters for the specified Learners:
  - Learners = 'tree' (default) — {'MinLeafSize'}

---

**Note** For hyperparameter optimization, Learners must be a single argument, not a cell array.

---

- 'all' — Optimize all eligible parameters.
- Cell array of eligible parameter names
- Vector of `optimizableVariable` objects, typically the output of hyperparameters

The optimization attempts to minimize the cross-validation loss (error) for `fitrensemble` by varying the parameters. To control the cross-validation type and other aspects of the optimization, use the `HyperparameterOptimizationOptions` name-value pair.

---

**Note** `OptimizeHyperparameters` values override any values you set using other name-value pairs. For example, setting `OptimizeHyperparameters` to 'auto' causes the 'auto' values to apply.

---

The eligible parameters for `fitrensemble` are:

- Method — Eligible methods are 'Bag' or 'LSBoost'.
- NumLearningCycles — `fitrensemble` searches among positive integers, by default log-scaled with range [10, 500].
- LearnRate — `fitrensemble` searches among positive reals, by default log-scaled with range [1e-3, 1].

- `MinLeafSize` — `fitrensemble` searches among integers log-scaled in the range `[1,max(2,floor(NumObservations/2))]`.
- `MaxNumSplits` — `fitrensemble` searches among integers log-scaled in the range `[1,max(2,NumObservations-1)]`.
- `NumVariablesToSample` — `fitrensemble` searches among integers in the range `[1,max(2,NumPredictors)]`.

Set nondefault parameters by passing a vector of `optimizableVariable` objects that have nondefault values. For example,

```
load carsmall
params = hyperparameters('fitrensemble',[Horsepower,Weight],MPG,'Tree');
params(4).Range = [1,20];
```

Pass `params` as the value of `OptimizeHyperparameters`.

By default, iterative display appears at the command line, and plots appear according to the number of hyperparameters in the optimization. For the optimization and plots, the objective function is  $\log(1 + \text{cross-validation loss})$  for regression, and the misclassification rate for classification. To control the iterative display, set the `HyperparameterOptimizationOptions` name-value pair, `Verbose` field. To control the plots, set the `HyperparameterOptimizationOptions` name-value pair, `ShowPlots` field.

For an example, see “Optimize Regression Ensemble” on page 32-2571.

Example: 'auto'

Data Types: char | cell

### **HyperparameterOptimizationOptions** — Options for optimization structure

Options for optimization, specified as a structure. Modifies the effect of the `OptimizeHyperparameters` name-value pair. All fields in the structure are optional.

Field Name	Values	Default
Optimizer	<ul style="list-style-type: none"> <li>'bayesopt' — Use Bayesian optimization. Internally, this setting calls bayesopt.</li> <li>'gridsearch' — Use grid search with NumGridDivisions values per dimension.</li> <li>'randomsearch' — Search at random among MaxObjectiveEvaluations points.</li> </ul> <p>'gridsearch' searches in a random order, using uniform sampling without replacement from the grid. After optimization, you can get a table in grid order by using the command sortrows (Mdl.ParameterOptimizationResults).</p>	'bayesopt'
AcquisitionFunctionName	<ul style="list-style-type: none"> <li>'expected-improvement-per-second-plus'</li> <li>'expected-improvement'</li> <li>'expected-improvement-plus'</li> <li>'expected-improvement-per-second'</li> <li>'lower-confidence-bound'</li> <li>'probability-of-improvement'</li> </ul> <p>For details, see the bayesopt AcquisitionFunctionName name-value pair, or “Acquisition Function Types” on page 10-4.</p>	'expected-improvement-per-second-plus'
MaxObjectiveEvaluations	Maximum number of objective function evaluations.	30 for 'bayesopt' or 'randomsearch', and the entire grid for 'gridsearch'

Field Name	Values	Default
MaxTime	Time limit, specified as a positive real. The time limit is in seconds, as measured by <code>tic</code> and <code>toc</code> . Run time can exceed <code>MaxTime</code> because <code>MaxTime</code> does not interrupt function evaluations.	Inf
NumGridDivisions	For 'gridsearch', the number of values in each dimension. Can be a vector of positive integers giving the number of values for each dimension, or a scalar that applies to all dimensions. Ignored for categorical variables.	10
ShowPlots	Logical value indicating whether to show plots. If <code>true</code> , plots the best objective function value against iteration number. If there are one or two optimization parameters, and if <code>Optimizer</code> is 'bayesopt', then <code>ShowPlots</code> also plots a model of the objective function against the parameters.	true
SaveIntermediateResults	Logical value indicating whether to save results when <code>Optimizer</code> is 'bayesopt'. If <code>true</code> , overwrites a workspace variable named 'BayesoptResults' at each iteration. The variable is a <code>BayesianOptimization</code> object.	false
Verbose	Display to the command line. <ul style="list-style-type: none"> <li>• 0 — No iterative display</li> <li>• 1 — Iterative display</li> <li>• 2 — Iterative display with extra information</li> </ul> For details, see the <code>bayesopt</code> <code>Verbose</code> name-value pair.	1

Field Name	Values	Default
UseParallel	Logical value indicating whether to run Bayesian optimization in parallel, which requires Parallel Computing Toolbox. For details, see “Parallel Bayesian Optimization” on page 10-9.	false
Repartition	Logical value indicating whether to repartition the cross-validation at every iteration. If <code>false</code> , the optimizer uses a single partition for the optimization.  <code>true</code> usually gives the most robust results because this setting takes partitioning noise into account. However, for good results, <code>true</code> requires at least twice as many function evaluations.	false
Use no more than one of the following three field names.		
CVPartition	A <code>cvpartition</code> object, as created by <code>cvpartition</code>	Kfold = 5
Holdout	A scalar in the range $(0, 1)$ representing the holdout fraction.	
Kfold	An integer greater than 1.	

Example: `struct('MaxObjectiveEvaluations', 60)`

Data Types: `struct`

## Output Arguments

### **mdl** — Trained regression ensemble model

RegressionBaggedEnsemble model object | RegressionEnsemble model object |  
RegressionPartitionedEnsemble cross-validated model object

Trained ensemble model, returned as one of the model objects in this table.



Model Object	Specify Any Cross-Validation Options?	Method Setting	Resample Setting
RegressionBaggedEnsemble	No	'Bag'	'on'
RegressionEnsemble	No	'LSBoost'	'off'
RegressionPartitionedEnsemble	Yes	'LSBoost' or 'Bag'	'off' or 'on'

The name-value pair arguments that control cross-validation are `CrossVal`, `Holdout`, `KFold`, `Leaveout`, and `CVPartition`.

To reference properties of `Mdl`, use dot notation. For example, to access or display the cell vector of weak learner model objects for an ensemble that has not been cross-validated, enter `Mdl.Trained` at the command line.

## Tips

- `NumLearningCycles` can vary from a few dozen to a few thousand. Usually, an ensemble with good predictive power requires from a few hundred to a few thousand weak learners. However, you do not have to train an ensemble for that many cycles at once. You can start by growing a few dozen learners, inspect the ensemble performance and then, if necessary, train more weak learners using `resume`.
- Ensemble performance depends on the ensemble setting and the setting of the weak learners. That is, if you specify weak learners with default parameters, then the ensemble can perform poorly. Therefore, like ensemble settings, it is good practice to adjust the parameters of the weak learners using templates, and to choose values that minimize generalization error.
- If you specify to resample using `Resample`, then it is good practice to resample to entire data set. That is, use the default setting of 1 for `FResample`.
- After training a `RegressionEnsemble` or `RegressionBaggedEnsemble` model object by using `fitrensemble`, use the function `predict` and the trained model object to generate C code that predicts responses for new data. For details, see “Code Generation” on page 32-0 .

## Algorithms

- For details of ensemble-aggregation algorithms, see “Ensemble Algorithms” on page 18-51.
- If you specify 'Method', 'LSBoost', then the software grows shallow decision trees by default. You can adjust tree depth by specifying the `MaxNumSplits`, `MinLeafSize`, and `MinParentSize` name-value pair arguments using `templateTree`.
- For dual-core systems and above, `fitrensemble` parallelizes training using Intel Threading Building Blocks (TBB). For details on Intel TBB, see <https://software.intel.com/en-us/intel-tbb>.

## References

- [1] Breiman, L. “Bagging Predictors.” *Machine Learning*. Vol. 26, pp. 123–140, 1996.
- [2] Breiman, L. “Random Forests.” *Machine Learning*. Vol. 45, pp. 5–32, 2001.
- [3] Freund, Y. and R. E. Schapire. “A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting.” *J. of Computer and System Sciences*, Vol. 55, pp. 119–139, 1997.
- [4] Friedman, J. “Greedy function approximation: A gradient boosting machine.” *Annals of Statistics*, Vol. 29, No. 5, pp. 1189–1232, 2001.
- [5] Hastie, T., R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning* section edition, Springer, New York, 2008.

## See Also

`RegressionBaggedEnsemble` | `RegressionEnsemble` |  
`RegressionPartitionedEnsemble` | `predict` | `templateTree`

## Topics

“Supervised Learning Workflow and Algorithms” on page 18-2  
“Framework for Ensemble Learning” on page 18-41

**Introduced in R2016b**

## fitrsvm

Fit a support vector machine regression model

`fitrsvm` trains or cross-validates a support vector machine (SVM) regression model on a low- through moderate-dimensional predictor data set. `fitrsvm` supports mapping the predictor data using kernel functions, and supports SMO, ISDA, or  $L1$  soft-margin minimization via quadratic programming for objective-function minimization.

To train a linear SVM regression model on a high-dimensional data set, that is, data sets that include many predictor variables, use `fitrlinear` instead.

To train an SVM model for binary classification, see `fitcsvm` for low- through moderate-dimensional predictor data sets, or `fitclinear` for high-dimensional data sets.

## Syntax

```
Mdl = fitrsvm(Tbl,ResponseVarName)
```

```
Mdl = fitrsvm(Tbl,formula)
```

```
Mdl = fitrsvm(Tbl,Y)
```

```
Mdl = fitrsvm(X,Y)
```

```
Mdl = fitrsvm(____,Name,Value)
```

## Description

`Mdl = fitrsvm(Tbl,ResponseVarName)` returns a full, trained support vector machine (SVM) regression model `Mdl` trained using the predictors values in the table `Tbl` and the response values in `Tbl.ResponseVarName`.

`Mdl = fitrsvm(Tbl,formula)` returns a full SVM regression model trained using the predictors values in the table `Tbl`. `formula` is an explanatory model of the response and a subset of predictor variables in `Tbl` used to fit `Mdl`.

`Mdl = fitrsvm(Tbl,Y)` returns a full, trained SVM regression model trained using the predictors values in the table `Tbl` and the response values in the vector `Y`.

`Mdl = fitrsvm(X,Y)` returns a full, trained SVM regression model trained using the predictors values in the matrix `X` and the response values in the vector `Y`.

`Mdl = fitrsvm( ____,Name,Value)` returns an SVM regression model with additional options specified by one or more name-value pair arguments, using any of the previous syntaxes. For example, you can specify the kernel function or train a cross-validated model.

## Examples

### Train Linear Support Vector Machine Regression Model

Train a support vector machine (SVM) regression model using sample data stored in matrices.

Load the `carsmall` data set.

```
load carsmall
rng 'default' % For reproducibility
```

Specify `Horsepower` and `Weight` as the predictor variables (`X`) and `MPG` as the response variable (`Y`).

```
X = [Horsepower,Weight];
Y = MPG;
```

Train a default SVM regression model.

```
Mdl = fitrsvm(X,Y)
```

```
Mdl =
```

```
RegressionSVM
    ResponseName: 'Y'
  CategoricalPredictors: []
    ResponseTransform: 'none'
                Alpha: [76x1 double]
                Bias: 43.1920
    KernelParameters: [1x1 struct]
    NumObservations: 93
```

```
BoxConstraints: [93x1 double]
ConvergenceInfo: [1x1 struct]
IsSupportVector: [93x1 logical]
Solver: 'SMO'
```

Mdl is a trained RegressionSVM model.

Check the model for convergence.

```
Mdl.ConvergenceInfo.Converged
```

```
ans =
```

```
logical
```

```
0
```

0 indicates that the model did not converge.

Retrain the model using standardized data.

```
MdlStd = fitrsvm(X,Y,'Standardize',true)
```

```
MdlStd =
```

```
RegressionSVM
  ResponseName: 'Y'
CategoricalPredictors: []
  ResponseTransform: 'none'
      Alpha: [77x1 double]
      Bias: 22.9131
  KernelParameters: [1x1 struct]
      Mu: [109.3441 2.9625e+03]
      Sigma: [45.3545 805.9668]
  NumObservations: 93
  BoxConstraints: [93x1 double]
  ConvergenceInfo: [1x1 struct]
  IsSupportVector: [93x1 logical]
  Solver: 'SMO'
```

Check the model for convergence.

```
MdlStd.ConvergenceInfo.Converged
```

```
ans =
```

```
    logical
```

```
         1
```

1 indicates that the model did converge.

Compute the resubstitution (in-sample) mean-squared error for the new model.

```
lStd = resubLoss(MdlStd)
```

```
lStd =
```

```
    17.0256
```

## Train Support Vector Machine Regression Model

Train a support vector machine regression model using the abalone data from the UCI Machine Learning Repository.

Download the data and save it in your current folder with the name 'abalone.csv'.

```
url = 'https://archive.ics.uci.edu/ml/machine-learning-databases/abalone/abalone.data';
websave('abalone.csv',url);
```

Read the data into a table. Specify the variable names.

```
varnames = {'Sex'; 'Length'; 'Diameter'; 'Height'; 'Whole_weight';...
            'Shucked_weight'; 'Viscera_weight'; 'Shell_weight'; 'Rings'};
Tbl = readtable('abalone.csv','Filetype','text','ReadVariableNames',false);
Tbl.Properties.VariableNames = varnames;
```

The sample data contains 4177 observations. All the predictor variables are continuous except for Sex, which is a categorical variable with possible values 'M' (for males), 'F'

(for females), and 'I' (for infants). The goal is to predict the number of rings (stored in Rings) on the abalone and determine its age using physical measurements.

Train an SVM regression model, using a Gaussian kernel function with an automatic kernel scale. Standardize the data.

```
rng default % For reproducibility
Mdl = fitrsvm(Tbl,'Rings','KernelFunction','gaussian','KernelScale','auto',...
    'Standardize',true)
```

```
Mdl =
```

```
RegressionSVM
    PredictorNames: {1×8 cell}
    ResponseName: 'Rings'
    CategoricalPredictors: 1
    ResponseTransform: 'none'
        Alpha: [3635×1 double]
        Bias: 10.8144
    KernelParameters: [1×1 struct]
        Mu: [1×10 double]
        Sigma: [1×10 double]
    NumObservations: 4177
    BoxConstraints: [4177×1 double]
    ConvergenceInfo: [1×1 struct]
    IsSupportVector: [4177×1 logical]
    Solver: 'SMO'
```

The Command Window shows that Mdl is a trained RegressionSVM model and displays a property list.

Display the properties of Mdl using dot notation. For example, check to confirm whether the model converged and how many iterations it completed.

```
conv = Mdl.ConvergenceInfo.Converged
iter = Mdl.NumIterations
```

```
conv =
```

```
logical
```



```

1

iter =

    2759

```

The returned results indicate that the model converged after 2759 iterations.

### Cross-Validate SVM Regression Model

Load the `carsmall` data set.

```

load carsmall
rng 'default' % For reproducibility

```

Specify Horsepower and Weight as the predictor variables (X) and MPG as the response variable (Y).

```

X = [Horsepower Weight];
Y = MPG;

```

Cross-validate two SVM regression models using 5-fold cross-validation. For both models, specify to standardize the predictors. For one of the models, specify to train using the default linear kernel, and the Gaussian kernel for the other model.

```

MdlLin = fitrsvm(X,Y,'Standardize',true,'KFold',5)
MdlGau = fitrsvm(X,Y,'Standardize',true,'KFold',5,'KernelFunction','gaussian')
MdlLin.Trained

```

```

MdlLin =

classreg.learning.partition.RegistrationPartitionedSVM
  CrossValidatedModel: 'SVM'
  PredictorNames: {'x1' 'x2'}
  ResponseName: 'Y'
  NumObservations: 94
  KFold: 5
  Partition: [1x1 cvpartition]
  ResponseTransform: 'none'

```

```
MdlGau =  
  
classreg.learning.partition.RegressionPartitionedSVM  
  CrossValidatedModel: 'SVM'  
  PredictorNames: {'x1' 'x2'}  
  ResponseName: 'Y'  
  NumObservations: 94  
  KFold: 5  
  Partition: [1x1 cvpartition]  
  ResponseTransform: 'none'  
  
ans =  
  
5x1 cell array  
  
{1x1 classreg.learning.regr.CompactRegressionSVM}  
{1x1 classreg.learning.regr.CompactRegressionSVM}  
{1x1 classreg.learning.regr.CompactRegressionSVM}  
{1x1 classreg.learning.regr.CompactRegressionSVM}  
{1x1 classreg.learning.regr.CompactRegressionSVM}
```

**MdlLin** and **MdlGau** are **RegressionPartitionedSVM** cross-validated models. The **Trained** property of each model is a 5-by-1 cell array of **CompactRegressionSVM** models. The models in the cell store the results of training on 4 folds of observations, and leaving one fold of observations out.

Compare the generalization error of the models. In this case, the generalization error is the out-of-sample mean-squared error.

```
mseLin = kfoldLoss(MdlLin)  
mseGau = kfoldLoss(MdlGau)
```

```
mseLin =  
  
17.4417
```

```
mseGau =  
  
    16.7333
```

The SVM regression model using the Gaussian kernel performs better than the one using the linear kernel.

Create a model suitable for making predictions by passing the entire data set to `fittersvm`, and specify all name-value pair arguments that yielded the better-performing model. However, do not specify any cross-validation options.

```
MdlGau = fittersvm(X,Y,'Standardize',true,'KernelFunction','gaussian');
```

To predict the MPG of a set of cars, pass `Mdl` and a table containing the horsepower and weight measurements of the cars to `predict`.

## Optimize SVM Regression

This example shows how to optimize hyperparameters automatically using `fittersvm`. The example uses the `carsmall` data.

Load the `carsmall` data set.

```
load carsmall
```

Specify `Horsepower` and `Weight` as the predictor variables (X) and `MPG` as the response variable (Y).

```
X = [Horsepower Weight];  
Y = MPG;
```

Find hyperparameters that minimize five-fold cross-validation loss by using automatic hyperparameter optimization.

For reproducibility, set the random seed and use the `'expected-improvement-plus'` acquisition function.

```
rng default  
Mdl = fittersvm(X,Y,'OptimizeHyperparameters','auto',...  
    'HyperparameterOptimizationOptions',struct('AcquisitionFunctionName',...  
    'expected-improvement-plus'))
```

Iter	Eval result	Objective	Objective runtime	BestSoFar (observed)	BestSoFar (estim.)	BoxConstrain- t
1	Best	7.3639	26.367	7.3639	7.3639	0.35664
2	Best	2.9107	0.23765	2.9107	3.1565	70.67
3	Accept	4.1884	0.12157	2.9107	3.13	14.367
4	Accept	4.159	0.13403	2.9107	3.0861	0.0030879
5	Accept	2.9226	0.18215	2.9107	2.9096	21.785
6	Accept	4.1884	0.10459	2.9107	2.9101	978.59
7	Best	2.9049	0.10705	2.9049	2.9049	53.698
8	Accept	8.2935	32.118	2.9049	2.9049	158.83
9	Accept	2.9392	6.168	2.9049	2.9049	993.05
10	Accept	2.9328	0.2379	2.9049	2.9053	877.93
11	Accept	4.1884	0.11617	2.9049	2.9053	932.68
12	Accept	4.1692	0.10331	2.9049	2.9056	0.0043392
13	Accept	2.9472	1.2644	2.9049	2.9057	991.29
14	Accept	4.1884	0.11693	2.9049	2.9058	0.0011719
15	Accept	2.9337	0.25162	2.9049	2.9067	108.03
16	Best	2.8964	2.4494	2.8964	2.8964	338.91
17	Accept	2.9337	0.31077	2.8964	2.8968	972.75
18	Accept	2.9026	0.67861	2.8964	2.8964	320.4
19	Accept	2.9076	0.16118	2.8964	2.8952	283.64
20	Accept	4.1884	0.11946	2.8964	2.895	0.0024693
Iter	Eval result	Objective	Objective runtime	BestSoFar (observed)	BestSoFar (estim.)	BoxConstrain- t
21	Accept	2.9026	0.47351	2.8964	2.8944	80.77
22	Accept	2.9057	0.13331	2.8964	2.8941	87.879
23	Accept	2.903	0.57279	2.8964	2.8977	260.58
24	Accept	2.9084	0.16758	2.8964	2.8978	105.7
25	Accept	2.9027	7.1288	2.8964	2.9	302.48
26	Accept	2.9012	1.822	2.8964	2.8948	253.84
27	Accept	2.9004	0.14062	2.8964	2.8948	99.856
28	Best	2.8591	2.5175	2.8591	2.8666	222.31
29	Accept	2.9055	1.1581	2.8591	2.8679	63.417
30	Accept	2.9084	20.003	2.8591	2.869	841.41

Optimization completed.  
 MaxObjectiveEvaluations of 30 reached.  
 Total function evaluations: 30  
 Total elapsed time: 183.3097 seconds.

Total objective function evaluation time: 105.4672

Best observed feasible point:

BoxConstraint	KernelScale	Epsilon
222.31	86.654	2.8081

Observed objective function value = 2.8591

Estimated objective function value = 2.869

Function evaluation time = 2.5175

Best estimated feasible point (according to models):

BoxConstraint	KernelScale	Epsilon
222.31	86.654	2.8081

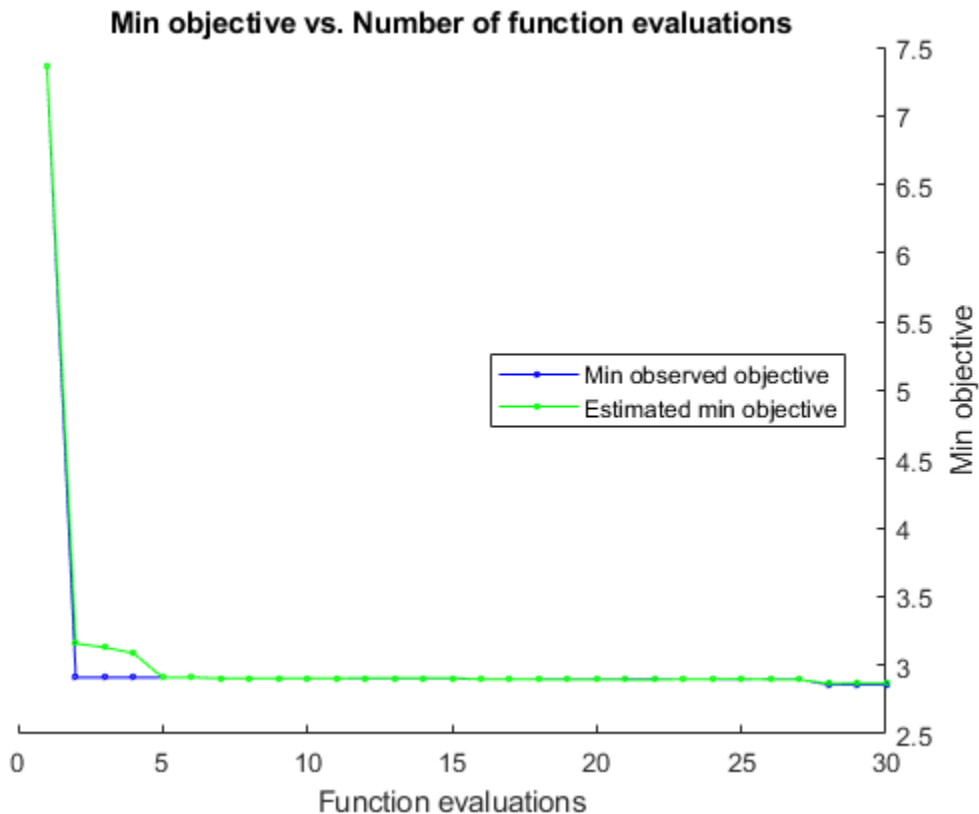
Estimated objective function value = 2.869

Estimated function evaluation time = 3.125

Mdl =

RegressionSVM

```
      ResponseName: 'Y'
      CategoricalPredictors: []
      ResponseTransform: 'none'
      Alpha: [49x1 double]
      Bias: 46.8549
      KernelParameters: [1x1 struct]
      NumObservations: 93
      HyperparameterOptimizationResults: [1x1 BayesianOptimization]
      BoxConstraints: [93x1 double]
      ConvergenceInfo: [1x1 struct]
      IsSupportVector: [93x1 logical]
      Solver: 'SMO'
```



The optimization searched over `BoxConstraint`, `KernelScale`, and `Epsilon`. The output is the regression with the minimum estimated cross-validation loss.

## Input Arguments

### `Tbl` — Predictor data

table

Sample data used to train the model, specified as a table. Each row of `Tbl` corresponds to one observation, and each column corresponds to one predictor variable. Optionally, `Tbl` can contain one additional column for the response variable. Multi-column variables and cell arrays other than cell arrays of character vectors are not allowed.

If `Tbl` contains the response variable, and you want to use all remaining variables in `Tbl` as predictors, then specify the response variable using `ResponseVarName`.

If `Tbl` contains the response variable, and you want to use only a subset of the remaining variables in `Tbl` as predictors, then specify a formula using `formula`.

If `Tbl` does not contain the response variable, then specify a response variable using `Y`. The length of response variable and the number of rows of `Tbl` must be equal.

If a row of `Tbl` or an element of `Y` contains at least one `NaN`, then `fitrsvm` removes those rows and elements from both arguments when training the model.

To specify the names of the predictors in the order of their appearance in `Tbl`, use the `PredictorNames` name-value pair argument.

Data Types: `table`

#### **ResponseVarName** — Response variable name

name of variable in `Tbl`

Response variable name, specified as the name of a variable in `Tbl`. The response variable must be a numeric vector.

You must specify `ResponseVarName` as a character vector. For example, if `Tbl` stores the response variable `Y` as `Tbl.Y`, then specify it as `'Y'`. Otherwise, the software treats all columns of `Tbl`, including `Y`, as predictors when training the model.

#### **formula** — Explanatory model of response and subset of predictor variables

character vector

Explanatory model of the response and a subset of the predictor variables, specified as a character vector in the form of `'Y~X1+X2+X3'`. In this form, `Y` represents the response variable, and `X1`, `X2`, and `X3` represent the predictor variables. The variables must be variable names in `Tbl` (`Tbl.Properties.VariableNames`).

To specify a subset of variables in `Tbl` as predictors for training the model, use a formula. If you specify a formula, then the software does not use any variables in `Tbl` that do not appear in `formula`.

Data Types: `char`

**y — Response data**

numeric vector

Response data, specified as an  $n$ -by-1 numeric vector. The length of `Y` and the number of rows of `Tbl` or `X` must be equal.

If a row of `Tbl` or `X`, or an element of `Y`, contains at least one NaN, then `fitrsvm` removes those rows and elements from both arguments when training the model.

To specify the response variable name, use the `ResponseName` name-value pair argument.

Data Types: `single` | `double`**x — Predictor data**

numeric matrix

Predictor data to which the SVM regression model is fit, specified as an  $n$ -by- $p$  numeric matrix.  $n$  is the number of observations and  $p$  is the number of predictor variables.

The length of `Y` and the number of rows of `X` must be equal.

If a row of `X` or an element of `Y` contains at least one NaN, then `fitrsvm` removes those rows and elements from both arguments.

To specify the names of the predictors in the order of their appearance in `X`, use the `PredictorNames` name-value pair argument.

Data Types: `single` | `double`**Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

---

**Note** You cannot use any cross-validation name-value pair along with `OptimizeHyperparameters`. You can modify the cross-validation for `OptimizeHyperparameters` only by using the `HyperparameterOptimizationOptions` name-value pair.

---



Example:

'KernelFunction', 'gaussian', 'Standardize', true, 'CrossVal', 'on' trains a 10-fold cross-validated SVM regression model using a Gaussian kernel and standardized training data.

### Support Vector Machine Options

#### BoxConstraint — Box constraint

positive scalar value

Box constraint for the alpha coefficients, specified as the comma-separated pair consisting of 'BoxConstraint' and a positive scalar value.

The absolute value of the Alpha coefficients cannot exceed the value of BoxConstraint.

The default BoxConstraint value for the 'gaussian' or 'rbf' kernel function is  $\text{iqr}(Y)/1.349$ , where  $\text{iqr}(Y)$  is the interquartile range of response variable  $Y$ . For all other kernels, the default BoxConstraint value is 1.

Example: BoxConstraint, 10

Data Types: single | double

#### KernelFunction — Kernel function

'linear' (default) | 'gaussian' | 'rbf' | 'polynomial' | function name

Kernel function used to compute the Gram matrix on page 32-2127, specified as the comma-separated pair consisting of 'KernelFunction' and a value in this table.

Value	Description	Formula
'gaussian' or 'rbf'	Gaussian or Radial Basis Function (RBF) kernel	$G(x_j, x_k) = \exp\left(-\ x_j - x_k\ ^2\right)$
'linear'	Linear kernel	$G(x_j, x_k) = x_j'x_k$
'polynomial'	Polynomial kernel. Use 'PolynomialOrder', $p$ to specify a polynomial kernel of order $p$ .	$G(x_j, x_k) = (1 + x_j'x_k)^p$

You can set your own kernel function, for example, kernel, by setting 'KernelFunction', 'kernel'. kernel must have the following form:

```
function G = kernel(U,V)
```

where:

- $U$  is an  $m$ -by- $p$  matrix.
- $V$  is an  $n$ -by- $p$  matrix.
- $G$  is an  $m$ -by- $n$  Gram matrix of the rows of  $U$  and  $V$ .

And `kernel.m` must be on the MATLAB path.

It is good practice to avoid using generic names for kernel functions. For example, call a sigmoid kernel function `'mysigmoid'` rather than `'sigmoid'`.

Example: `'KernelFunction', 'gaussian'`

Data Types: `char`

### **KernelScale** — Kernel scale parameter

1 (default) | `'auto'` | positive scalar

Kernel scale parameter, specified as the comma-separated pair consisting of `'KernelScale'` and `'auto'` or a positive scalar. The software divides all elements of the predictor matrix  $X$  by the value of `KernelScale`. Then, the software applies the appropriate kernel norm to compute the Gram matrix.

- If you specify `'auto'`, then the software selects an appropriate scale factor using a heuristic procedure. This heuristic procedure uses subsampling, so estimates can vary from one call to another. Therefore, to reproduce results, set a random number seed using `rng` before training.
- If you specify `KernelScale` and your own kernel function, for example, `kernel`, using `'KernelFunction', 'kernel'`, then the software throws an error. You must apply scaling within `kernel`.

Example: `'KernelScale', 'auto'`

Data Types: `double` | `single` | `char`

### **PolynomialOrder** — Polynomial kernel function order

3 (default) | positive integer

Polynomial kernel function order, specified as the comma-separated pair consisting of `'PolynomialOrder'` and a positive integer.

If you set 'PolynomialOrder' and KernelFunction is not 'polynomial', then the software throws an error.

Example: 'PolynomialOrder',2

Data Types: double | single

### **KernelOffset** — Kernel offset parameter

nonnegative scalar

Kernel offset parameter, specified as the comma-separated pair consisting of 'KernelOffset' and a nonnegative scalar.

The software adds KernelOffset to each element of the Gram matrix.

The defaults are:

- 0 if the solver is SMO (that is, you set 'Solver', 'SMO')
- 0.1 if the solver is ISDA (that is, you set 'Solver', 'ISDA')

Example: 'KernelOffset',0

Data Types: double | single

### **Epsilon** — Half the width of epsilon-insensitive band

$iqr(Y)/13.49$  (default) | nonnegative scalar value

Half the width of the epsilon-insensitive band, specified as the comma-separated pair consisting of 'Epsilon' and a nonnegative scalar value.

The default Epsilon value is  $iqr(Y)/13.49$ , which is an estimate of a tenth of the standard deviation using the interquartile range of the response variable  $Y$ . If  $iqr(Y)$  is equal to zero, then the default Epsilon value is 0.1.

Example: 'Epsilon',0.3

Data Types: single | double

### **Standardize** — Flag to standardize predictor data

false (default) | true

Flag to standardize the predictor data, specified as the comma-separated pair consisting of 'Standardize' and true (1) or false (0).

If you set `'Standardize', true`:

- The software centers and scales each column of the predictor data ( $X$ ) by the weighted column mean and standard deviation, respectively (for details on weighted standardizing, see “Algorithms” on page 32-2132). MATLAB does not standardize the data contained in the dummy variable columns generated for categorical predictors.
- The software trains the model using the standardized predictor matrix, but stores the unstandardized data in the model property `X`.

Example: `'Standardize', true`

Data Types: `logical`

### **solver** — Optimization routine

`'ISDA' | 'L1QP' | 'SMO'`

Optimization routine, specified as the comma-separated pair consisting of `'Solver'` and a value in this table.

Value	Description
<code>'ISDA'</code>	Iterative Single Data Algorithm (see [30])
<code>'L1QP'</code>	Uses <code>quadprog</code> to implement $L1$ soft-margin minimization by quadratic programming. This option requires an Optimization Toolbox license. For more details, see “Quadratic Programming Definition” (Optimization Toolbox).
<code>'SMO'</code>	Sequential Minimal Optimization (see [17])

The defaults are:

- `'ISDA'` if you set `'OutlierFraction'` to a positive value
- `'SMO'` otherwise

Example: `'Solver', 'ISDA'`

Data Types: `char`

### **Alpha** — Initial estimates of alpha coefficients

numeric vector

Initial estimates of alpha coefficients, specified as the comma-separated pair consisting of 'Alpha' and a numeric vector. The length of Alpha must be equal to the number of rows of X.

- Each element of Alpha corresponds to an observation in X.
- Alpha cannot contain any NaNs.
- If you specify Alpha and any one of the cross-validation name-value pair arguments ('CrossVal', 'CVPartition', 'Holdout', 'KFold', or 'Leaveout'), then the software returns an error.

If Y contains any missing values, then remove all rows of Y, X, and Alpha that correspond to the missing values. That is, enter:

```
idx = ~isnan(Y);
Y = Y(idx);
X = X(idx, :);
alpha = alpha(idx);
```

Then, pass Y, X, and alpha as the response, predictors, and initial alpha estimates, respectively.

The default is `zeros(size(Y,1),1)`.

Example: `'Alpha',0.1*ones(size(X,1),1)`

Data Types: `single` | `double`

### **CacheSize** — Cache size

1000 (default) | 'maximal' | positive scalar

Cache size, specified as the comma-separated pair consisting of 'CacheSize' and 'maximal' or a positive scalar.

If CacheSize is 'maximal', then the software reserves enough memory to hold the entire  $n$ -by- $n$  Gram matrix on page 32-2127.

If CacheSize is a positive scalar, then the software reserves CacheSize megabytes of memory for training the model.

Example: `'CacheSize','maximal'`

Data Types: `double` | `char` | `single`

**ClipAlphas — Flag to clip alpha coefficients**

true (default) | false

Flag to clip alpha coefficients, specified as the comma-separated pair consisting of 'ClipAlphas' and either true or false.

Suppose that the alpha coefficient for observation  $j$  is  $a_j$  and the box constraint of observation  $j$  is  $C_j$ ,  $j = 1, \dots, n$ .  $n$  is the training sample size.

Value	Description
true	At each iteration, if $a_j$ is near 0 or near $C_j$ , then MATLAB sets $a_j$ to 0 or to $C_j$ , respectively.
false	MATLAB does not change the alpha coefficients during optimization.

MATLAB stores the final values of  $a$  in the Alpha property of the trained SVM model object.

ClipAlphas can affect SMO and ISDA convergence.

Example: 'ClipAlphas', false

Data Types: logical

**NumPrint — Number of iterations between optimization diagnostic message output**

1000 (default) | nonnegative integer

Number of iterations between optimization diagnostic message output, specified as the comma-separated pair consisting of 'NumPrint' and a nonnegative integer.

If you use 'Verbose', 1 and 'NumPrint', numprint, then the software displays all optimization diagnostic messages from SMO and ISDA every numprint iterations in the Command Window.

Example: 'NumPrint', 500

Data Types: double | single

**OutlierFraction — Expected proportion of outliers in training data**

0 (default) | numeric scalar in the interval [0,1)

Expected proportion of outliers in training data, specified as the comma-separated pair consisting of 'OutlierFraction' and a numeric scalar in the interval [0,1). fitrsvm

removes observations with large gradients, ensuring that `fitrsvm` removes the fraction of observations specified by `OutlierFraction` by the time convergence is reached. This name-value pair is only valid when `'Solver'` is `'ISDA'`.

Example: `'OutlierFraction',0.1`

Data Types: `single` | `double`

**RemoveDuplicates** — Flag to replace duplicate observations with single observations in training data

`false` (default) | `true`

Flag to replace duplicate observations with single observations in the training data, specified as the comma-separated pair consisting of `'RemoveDuplicates'` and `true` or `false`.

If `RemoveDuplicates` is `true`, then `fitrsvm` replaces duplicate observations in the training data with a single observation of the same value. The weight of the single observation is equal to the sum of the weights of the corresponding removed duplicates (see `Weights`).

---

**Tip** If your data set contains many duplicate observations, then specifying `'RemoveDuplicates',true` can decrease convergence time considerably.

---

Data Types: `logical`

**Verbose** — Verbosity level

0 (default) | 1 | 2

Verbosity level, specified as the comma-separated pair consisting of `'Verbose'` and either 0, 1, or 2. `Verbose` controls the amount of optimization information that the software displays in the Command Window and saves as a structure to `Mdl.ConvergenceInfo.History`.

This table summarizes the available verbosity level options.

Value	Description
0	The software does not display or save convergence information.

Value	Description
1	The software displays diagnostic messages and saves convergence criteria every <code>numprint</code> iterations, where <code>numprint</code> is the value of the name-value pair argument <code>'NumPrint'</code> .
2	The software displays diagnostic messages and saves convergence criteria at every iteration.

Example: `'Verbose', 1`

Data Types: `double` | `single`

### Other Regression Options

#### `CategoricalPredictors` — Categorical predictors list

vector of positive integers | logical vector | character matrix | cell array of character vectors | `'all'`

Categorical predictors list, specified as the comma-separated pair consisting of `'CategoricalPredictors'` and one of these values:

Value	Description
Vector of positive integers	An entry in the vector is the index value corresponding to the column of the predictor data ( <code>X</code> or <code>Tbl</code> ) that contains a categorical variable.
Logical vector	A <code>true</code> entry means that the corresponding column of predictor data ( <code>X</code> or <code>Tbl</code> ) is a categorical variable.
Character matrix	Each row of the matrix is the name of a predictor variable. The names must match the entries in <code>PredictorNames</code> . Pad the names with extra blanks so each row of the character matrix has the same length.
Cell array of character vectors	Each element in the array is the name of a predictor variable. The names must match the entries in <code>PredictorNames</code> .
<code>'all'</code>	All predictors are categorical.

By default, if the predictor data is in a table (`Tbl`), `fitrsvm` assumes that a variable is categorical if it contains logical values, categorical values, or a cell array of character



vectors. If the predictor data is a matrix ( $X$ ), `fitrsvm` assumes all predictors are continuous. To identify any categorical predictors when the data is a matrix, use the `'CategoricalPredictors'` name-value pair argument.

Example: `'CategoricalPredictors','all'`

Data Types: `single` | `double` | `logical` | `char` | `cell`

### **PredictorNames** — Predictor variable names

cell array of unique character vectors

Predictor variable names, specified as the comma-separated pair consisting of `'PredictorNames'` and a cell array of unique character vectors. The functionality of `'PredictorNames'` depends on the way you supply the training data.

- If you supply  $X$  and  $Y$ , then you can use `'PredictorNames'` to give the predictor variables in  $X$  names.
  - The order of the names in `PredictorNames` must correspond to the column order of  $X$ . That is, `PredictorNames{1}` is the name of  $X(:,1)$ , `PredictorNames{2}` is the name of  $X(:,2)$ , and so on. Also, `size(X,2)` and `numel(PredictorNames)` must be equal.
  - By default, `PredictorNames` is `{'x1','x2',...}`.
- If you supply `Tbl`, then you can use `'PredictorNames'` to choose which predictor variables to use in training. That is, `fitrsvm` uses the predictor variables in `PredictorNames` and the response only in training.
  - `PredictorNames` must be a subset of `Tbl.Properties.VariableNames` and cannot include the name of the response variable.
  - By default, `PredictorNames` contains the names of all predictor variables.
  - It good practice to specify the predictors for training using one of `'PredictorNames'` or formula only.

Example: `'PredictorNames',`  
`{'SepalLength','SepalWidth','PedalLength','PedalWidth'}`

Data Types: `cell`

### **ResponseName** — Response variable name

`'Y'` (default) | character vector

Response variable name, specified as the comma-separated pair consisting of 'ResponseName' and a character vector.

- If you supply  $Y$ , then you can use 'ResponseName' to specify a name for the response variable.
- If you supply ResponseVarName or formula, then you cannot use 'ResponseName'.

Example: 'ResponseName', 'response'

Data Types: char

### **ResponseTransform** — Response transformation

'none' (default) | function handle

Response transformation, specified as the comma-separated pair consisting of 'ResponseTransform' and either 'none' or a function handle. The default is 'none', which means  $@(x)x$ , or no transformation. For a MATLAB function or a function you define, use its function handle. The function handle must accept a vector (the original response values) and return a vector of the same size (the transformed response values).

Example: 'ResponseTransform', 'none'

Data Types: char | function\_handle

### **Weights** — Observation weights

ones(size(X,1),1) (default) | vector of numeric values

Observation weights, specified as the comma-separated pair consisting of 'Weights' and a vector of numeric values. The size of Weights must equal the number of rows in  $X$ . fitrsvm normalizes the values of Weights to sum to 1.

Data Types: single | double

### **Cross-Validation Options**

#### **CrossVal** — Cross-validation flag

'off' (default) | 'on'

Cross-validation flag, specified as the comma-separated pair consisting of 'CrossVal' and either 'on' or 'off'.

If you specify 'on', then the software implements 10-fold cross-validation.

To override this cross-validation setting, use one of these name-value pair arguments: `CVPartition`, `Holdout`, `KFold`, or `Leaveout`. To create a cross-validated model, you can use one cross-validation name-value pair argument at a time only.

Alternatively, you can cross-validate the model later using the `crossval` method.

Example: `'CrossVal', 'on'`

### **CVPartition** — Cross-validation partition

`[]` (default) | `cvpartition` partition object

Cross-validation partition, specified as the comma-separated pair consisting of `'CVPartition'` and a `cvpartition` partition object as created by `cvpartition`. The partition object specifies the type of cross-validation, and also the indexing for training and validation sets.

To create a cross-validated model, you can use one of these four name-value pair arguments only: `CVPartition`, `Holdout`, `KFold`, or `Leaveout`.

Example: Suppose you create a random partition for 5-fold cross-validation on 500 observations by using `cvp = cvpartition(500, 'KFold', 5)`. Then, you can specify the cross-validation partition for `fitcecoc` as `'CVPartition', cvp`.

### **Holdout** — Fraction of data for holdout validation

scalar value in the range (0,1)

Fraction of data used for holdout validation, specified as the comma-separated pair consisting of `'Holdout'` and a scalar value in the range (0,1). If you specify `'Holdout', p`, then the software:

- 1 Randomly reserves  $p \times 100\%$  of the data as validation data, and trains the model using the rest of the data
- 2 Stores the compact, trained model in the `Trained` property of the cross-validated model.

To create a cross-validated model, you can use one of these four name-value pair arguments only: `CVPartition`, `Holdout`, `KFold`, or `Leaveout`.

Example: `'Holdout', 0.1`

Data Types: `double` | `single`

**KFold** — Number of folds

10 (default) | positive integer value greater than 1

Number of folds to use in a cross-validated model, specified as the comma-separated pair consisting of 'KFold' and a positive integer value greater than 1. If you specify, e.g., 'KFold',  $k$ , then the software:

- 1 Randomly partitions the data into  $k$  sets
- 2 For each set, reserves the set as validation data, and trains the model using the other  $k - 1$  sets
- 3 Stores the  $k$  compact, trained models in the cells of a  $k$ -by-1 cell vector in the `Trained` property of the cross-validated model.

To create a cross-validated model, you can use one of these four name-value pair arguments only: `CVPartition`, `Holdout`, `KFold`, or `Leaveout`.

Example: 'KFold', 5

Data Types: `single` | `double`

**Leaveout** — Leave-one-out cross-validation flag

'off' (default) | 'on'

Leave-one-out cross-validation flag, specified as the comma-separated pair consisting of 'Leaveout' and 'on' or 'off'. If you specify 'Leaveout', 'on', then, for each of the  $n$  observations, where  $n$  is `size(Mdl.X, 1)`, the software:

- 1 Reserves the observation as validation data, and trains the model using the other  $n - 1$  observations
- 2 Stores the  $n$  compact, trained models in the cells of an  $n$ -by-1 cell vector in the `Trained` property of the cross-validated model.

To create a cross-validated model, you can use one of these four name-value pair arguments only: `CVPartition`, `Holdout`, `KFold`, or `Leaveout`.

Example: 'Leaveout', 'on'

Data Types: `char`

**Convergence Controls****DeltaGradientTolerance** — Tolerance for gradient difference

0 (default) | nonnegative scalar

Tolerance for gradient difference between upper and lower violators obtained by SMO or ISDA, specified as the comma-separated pair consisting of 'DeltaGradientTolerance' and a nonnegative scalar.

Example: 'DeltaGradientTolerance',1e-4

Data Types: single | double

### **GapTolerance** — Feasibility gap tolerance

1e-3 (default) | nonnegative scalar

Feasibility gap tolerance obtained by SMO or ISDA, specified as the comma-separated pair consisting of 'GapTolerance' and a nonnegative scalar.

If GapTolerance is 0, then fitrsvm does not use this parameter to check convergence.

Example: 'GapTolerance',1e-4

Data Types: single | double

### **IterationLimit** — Maximal number of numerical optimization iterations

1e6 (default) | positive integer

Maximal number of numerical optimization iterations, specified as the comma-separated pair consisting of 'IterationLimit' and a positive integer.

The software returns a trained model regardless of whether the optimization routine successfully converges. `Mdl.ConvergenceInfo` contains convergence information.

Example: 'IterationLimit',1e8

Data Types: double | single

### **KKTTolerance** — Tolerance for KKT violation

0 | nonnegative scalar value

Tolerance for Karush-Kuhn-Tucker (KKT) violation, specified as the comma-separated pair consisting of 'KKTTolerance' and a nonnegative scalar value.

This name-value pair applies only if 'Solver' is 'SMO' or 'ISDA'.

If KKTTolerance is 0, then fitrsvm does not use this parameter to check convergence.

Example: 'KKTTolerance',1e-4

Data Types: `single` | `double`

**ShrinkagePeriod** — Number of iterations between movement of observations from active to inactive set

0 (default) | nonnegative integer

Number of iterations between the movement of observations from the active to inactive set, specified as the comma-separated pair consisting of 'ShrinkagePeriod' and a nonnegative integer.

If you set 'ShrinkagePeriod', 0, then the software does not shrink the active set.

Example: 'ShrinkagePeriod', 1000

Data Types: `double` | `single`

**Hyperparameter Optimization**

**OptimizeHyperparameters** — Parameters to optimize

'none' (default) | 'auto' | 'all' | cell array of eligible parameter names | vector of `optimizableVariable` objects

Parameters to optimize, specified as:

- 'none' — Do not optimize.
- 'auto' — Use {'BoxConstraint', 'KernelScale', 'Epsilon'}
- 'all' — Optimize all eligible parameters.
- Cell array of eligible parameter names
- Vector of `optimizableVariable` objects, typically the output of `hyperparameters`

The optimization attempts to minimize the cross-validation loss (error) for `fitrsvm` by varying the parameters. To control the cross-validation type and other aspects of the optimization, use the `HyperparameterOptimizationOptions` name-value pair.

---

**Note** `OptimizeHyperparameters` values override any values you set using other name-value pairs. For example, setting `OptimizeHyperparameters` to 'auto' causes the 'auto' values to apply.

---

The eligible parameters for `fitrsvm` are:

- `BoxConstraint` — `fittersvm` searches among positive values, by default log-scaled in the range `[1e-3, 1e3]`.
- `KernelScale` — `fittersvm` searches among positive values, by default log-scaled in the range `[1e-3, 1e3]`.
- `Epsilon` — `fittersvm` searches among positive values, by default log-scaled in the range `[1e-3*ResponseIqr/1.349, 1e2*ResponseIqr/1.349]`.
- `KernelFunction` — `fittersvm` searches among 'gaussian', 'linear', and 'polynomial'.
- `PolynomialOrder` — `fittersvm` searches among integers in the range `[2, 4]`.
- `Standardize` — `fittersvm` searches among 'true' and 'false'.

Set nondefault parameters by passing a vector of `optimizableVariable` objects that have nondefault values. For example,

```
load carsmall
params = hyperparameters('fittersvm', [Horsepower, Weight], MPG);
params(1).Range = [1e-4, 1e6];
```

Pass `params` as the value of `OptimizeHyperparameters`.

By default, iterative display appears at the command line, and plots appear according to the number of hyperparameters in the optimization. For the optimization and plots, the objective function is  $\log(1 + \text{cross-validation loss})$  for regression, and the misclassification rate for classification. To control the iterative display, set the `HyperparameterOptimizationOptions` name-value pair, `Verbose` field. To control the plots, set the `HyperparameterOptimizationOptions` name-value pair, `ShowPlots` field.

For an example, see “Optimize SVM Regression” on page 32-2605.

Example: 'auto'

Data Types: char | cell

### **HyperparameterOptimizationOptions** — Options for optimization structure

Options for optimization, specified as a structure. Modifies the effect of the `OptimizeHyperparameters` name-value pair. All fields in the structure are optional.

Field Name	Values	Default
Optimizer	<ul style="list-style-type: none"> <li>'bayesopt' — Use Bayesian optimization. Internally, this setting calls bayesopt.</li> <li>'gridsearch' — Use grid search with NumGridDivisions values per dimension.</li> <li>'randomsearch' — Search at random among MaxObjectiveEvaluations points.</li> </ul> <p>'gridsearch' searches in a random order, using uniform sampling without replacement from the grid. After optimization, you can get a table in grid order by using the command sortrows (Mdl.ParameterOptimizationResults).</p>	'bayesopt'
AcquisitionFunctionName	<ul style="list-style-type: none"> <li>'expected-improvement-per-second-plus'</li> <li>'expected-improvement'</li> <li>'expected-improvement-plus'</li> <li>'expected-improvement-per-second'</li> <li>'lower-confidence-bound'</li> <li>'probability-of-improvement'</li> </ul> <p>For details, see the bayesopt AcquisitionFunctionName name-value pair, or “Acquisition Function Types” on page 10-4.</p>	'expected-improvement-per-second-plus'
MaxObjectiveEvaluations	Maximum number of objective function evaluations.	30 for 'bayesopt' or 'randomsearch', and the entire grid for 'gridsearch'



Field Name	Values	Default
MaxTime	Time limit, specified as a positive real. The time limit is in seconds, as measured by <code>tic</code> and <code>toc</code> . Run time can exceed <code>MaxTime</code> because <code>MaxTime</code> does not interrupt function evaluations.	Inf
NumGridDivisions	For 'gridsearch', the number of values in each dimension. Can be a vector of positive integers giving the number of values for each dimension, or a scalar that applies to all dimensions. Ignored for categorical variables.	10
ShowPlots	Logical value indicating whether to show plots. If <code>true</code> , plots the best objective function value against iteration number. If there are one or two optimization parameters, and if <code>Optimizer</code> is 'bayesopt', then <code>ShowPlots</code> also plots a model of the objective function against the parameters.	true
SaveIntermediateResults	Logical value indicating whether to save results when <code>Optimizer</code> is 'bayesopt'. If <code>true</code> , overwrites a workspace variable named 'BayesoptResults' at each iteration. The variable is a <code>BayesianOptimization</code> object.	false
Verbose	Display to the command line. <ul style="list-style-type: none"> <li>• 0 — No iterative display</li> <li>• 1 — Iterative display</li> <li>• 2 — Iterative display with extra information</li> </ul> For details, see the <code>bayesopt</code> <code>Verbose</code> name-value pair.	1

Field Name	Values	Default
UseParallel	Logical value indicating whether to run Bayesian optimization in parallel, which requires Parallel Computing Toolbox. For details, see “Parallel Bayesian Optimization” on page 10-9.	false
Repartition	Logical value indicating whether to repartition the cross-validation at every iteration. If <code>false</code> , the optimizer uses a single partition for the optimization.  <code>true</code> usually gives the most robust results because this setting takes partitioning noise into account. However, for good results, <code>true</code> requires at least twice as many function evaluations.	false
Use no more than one of the following three field names.		
CVPartition	A <code>cvpartition</code> object, as created by <code>cvpartition</code>	Kfold = 5
Holdout	A scalar in the range (0, 1) representing the holdout fraction.	
Kfold	An integer greater than 1.	

Example: `struct('MaxObjectiveEvaluations', 60)`

Data Types: `struct`

## Output Arguments

### **mdl** — Trained SVM regression model

RegressionSVM model | RegressionPartitionedSVM cross-validated model

Trained SVM regression model, returned as a `RegressionSVM` model or `RegressionPartitionedSVM` cross-validated model.

If you set any of the name-value pair arguments `KFold`, `Holdout`, `Leaveout`, `CrossVal`, or `CVPartition`, then `Mdl` is a `RegressionPartitionedSVM` cross-validated model. Otherwise, `Mdl` is a `RegressionSVM` model.

## Limitations

`fitrsvm` supports low- through moderate-dimensional data sets. For high-dimensional data set, use `fitrlinear` instead.

## Tips

- Unless your data set is large, always try to standardize the predictors (see `Standardize`). Standardization makes predictors insensitive to the scales on which they are measured.
- It is good practice to cross-validate using the `KFold` name-value pair argument. The cross-validation results determine how well the SVM model generalizes.
- Sparsity in support vectors is a desirable property of an SVM model. To decrease the number of support vectors, set the `BoxConstraint` name-value pair argument to a large value. This action also increases the training time.
- For optimal training time, set `CacheSize` as high as the memory limit on your computer allows.
- If you expect many fewer support vectors than observations in the training set, then you can significantly speed up convergence by shrinking the active-set using the name-value pair argument `'ShrinkagePeriod'`. It is good practice to use `'ShrinkagePeriod',1000`.
- Duplicate observations that are far from the regression line do not affect convergence. However, just a few duplicate observations that occur near the regression line can slow down convergence considerably. To speed up convergence, specify `'RemoveDuplicates',true` if:
  - Your data set contains many duplicate observations.
  - You suspect that a few duplicate observations can fall near the regression line.

However, to maintain the original data set during training, `fitrsvm` must temporarily store separate data sets: the original and one without the duplicate

observations. Therefore, if you specify `true` for data sets containing few duplicates, then `fitrsvm` consumes close to double the memory of the original data.

- After training a `RegressionSVM` model object by using `fitrsvm`, use the function `predict` and the trained model object to generate C code that predicts responses for new data. For details, see “Code Generation” on page 32-0 .

## Algorithms

- For the mathematical formulation of linear and nonlinear SVM regression problems and the solver algorithms, see “Understanding Support Vector Machine Regression” on page 25-2.
- NaN, <undefined>, and empty character vector ( ' ' ) values indicate missing values. `fitrsvm` removes entire rows of data corresponding to a missing response. When normalizing weights, `fitrsvm` ignores any weight corresponding to an observation with at least one missing predictor. Consequently, observation box constraints might not equal `BoxConstraint`.
- `fitrsvm` removes observations that have zero weight.
- If you set 'Standardize', `true` and 'Weights', then `fitrsvm` standardizes the predictors using their corresponding weighted means and weighted standard deviations. That is, `fitrsvm` standardizes predictor  $j$  ( $x_j$ ) using

$$x_j^* = \frac{x_j - \mu_j^*}{\sigma_j^*}.$$

- $$\mu_j^* = \frac{1}{\sum_k w_k} \sum_k w_k x_{jk}.$$

- $x_{jk}$  is observation  $k$  (row) of predictor  $j$  (column).

- 

- $$(\sigma_j^*)^2 = \frac{v_1}{v_1^2 - v_2} \sum_k w_k (x_{jk} - \mu_j^*)^2.$$

- 

$$v_1 = \sum_j w_j.$$

•

$$v_2 = \sum (w_j)^2.$$

- If your predictor data contains categorical variables, then the software generally uses full dummy encoding for these variables. The software creates one dummy variable for each level of each categorical variable.
  - The `PredictorNames` property stores one element for each of the original predictor variable names. For example, assume that there are three predictors, one of which is a categorical variable with three levels. Then `PredictorNames` is a 1-by-3 cell array of character vectors containing the original names of the predictor variables.
  - The `ExpandedPredictorNames` property stores one element for each of the predictor variables, including the dummy variables. For example, assume that there are three predictors, one of which is a categorical variable with three levels. Then `ExpandedPredictorNames` is a 1-by-5 cell array of character vectors containing the names of the predictor variables and the new dummy variables.
  - Similarly, the `Beta` property stores one beta coefficient for each predictor, including the dummy variables.
  - The `SupportVectors` property stores the predictor values for the support vectors, including the dummy variables. For example, assume that there are  $m$  support vectors and three predictors, one of which is a categorical variable with three levels. Then `SupportVectors` is an  $m$ -by-5 matrix.
  - The `X` property stores the training data as originally input. It does not include the dummy variables. When the input is a table, `X` contains only the columns used as predictors.
- For predictors specified in a table, if any of the variables contain ordered (ordinal) categories, the software uses ordinal encoding for these variables.
  - For a variable having  $k$  ordered levels, the software creates  $k - 1$  dummy variables. The  $j$ th dummy variable is -1 for levels up to  $j$ , and +1 for levels  $j + 1$  through  $k$ .
  - The names of the dummy variables stored in the `ExpandedPredictorNames` property indicate the first level with the value +1. The software stores  $k - 1$  additional predictor names for the dummy variables, including the names of levels 2, 3, ...,  $k$ .
- All solvers implement  $L1$  soft-margin minimization.

- Let  $p$  be the proportion of outliers that you expect in the training data. If you set 'OutlierFraction',  $p$ , then the software implements robust learning. In other words, the software attempts to remove  $100p\%$  of the observations when the optimization algorithm converges. The removed observations correspond to gradients that are large in magnitude.

## References

- [1] Clark, D., Z. Schreter, A. Adams. *A Quantitative Comparison of Dystal and Backpropagation*, submitted to the Australian Conference on Neural Networks, 1996.
- [2] Fan, R.-E., P.-H. Chen, and C.-J. Lin. “Working set selection using second order information for training support vector machines.” *Journal of Machine Learning Research*, Vol 6, 2005, pp. 1889–1918.
- [3] Kecman V., T. -M. Huang, and M. Vogt. “Iterative Single Data Algorithm for Training Kernel Machines from Huge Data Sets: Theory and Performance.” In *Support Vector Machines: Theory and Applications*. Edited by Lipo Wang, 255–274. Berlin: Springer-Verlag, 2005.
- [4] Lichman, M. *UCI Machine Learning Repository*, [<http://archive.ics.uci.edu/ml>]. Irvine, CA: University of California, School of Information and Computer Science.
- [5] Nash, W.J., T. L. Sellers, S. R. Talbot, A. J. Cawthorn, and W. B. Ford. *The Population Biology of Abalone (Haliotis species) in Tasmania. I. Blacklip Abalone (H. rubra) from the North Coast and Islands of Bass Strait*, Sea Fisheries Division, Technical Report No. 48, 1994.
- [6] Waugh, S. *Extending and benchmarking Cascade-Correlation*, Ph.D. thesis, Computer Science Department, University of Tasmania, 1995.

## See Also

CompactRegressionSVM | RegressionPartitionedSVM | RegressionSVM | predict

## Topics

“Understanding Support Vector Machine Regression” on page 25-2

Introduced in R2015b

## fitrtree

Fit binary regression decision tree

### Syntax

```
tree = fitrtree(Tbl,ResponseVarName)
tree = fitrtree(Tbl,formula)
tree = fitrtree(Tbl,Y)

tree = fitrtree(X,Y)

tree = fitrtree( ____,Name,Value)
```

### Description

`tree = fitrtree(Tbl,ResponseVarName)` returns a regression tree based on the input variables (also known as predictors, features, or attributes) in the table `Tbl` and output (response) contained in `Tbl.ResponseVarName`. `tree` is a binary tree where each branching node is split based on the values of a column of `Tbl`.

`tree = fitrtree(Tbl,formula)` returns a regression tree based on the input variables contained in the table `Tbl`. `formula` is an explanatory model of the response and a subset of predictor variables in `Tbl` used to fit `tree`.

`tree = fitrtree(Tbl,Y)` returns a regression tree based on the input variables contained in the table `Tbl` and output in vector `Y`.

`tree = fitrtree(X,Y)` returns a regression tree based on the input variables `X` and output `Y`. `tree` is a binary tree where each branching node is split based on the values of a column of `X`.

`tree = fitrtree( ____,Name,Value)` fits a tree with additional options specified by one or more `Name,Value` pair arguments. For example, you can specify observation weights or train a cross-validated model.



## Examples

### Construct a Regression Tree

Load the sample data.

```
load carsmall;
```

Construct a regression tree using the sample data.

```
tree = fitrtree([Weight, Cylinders],MPG,...  
               'categoricalpredictors',2,'MinParentSize',20,...  
               'PredictorNames',{'W','C'})
```

```
tree =
```

```
RegressionTree  
    PredictorNames: {'W' 'C'}  
    ResponseName: 'Y'  
    CategoricalPredictors: 2  
    ResponseTransform: 'none'  
    NumObservations: 94
```

Predict the mileage of 4,000-pound cars with 4, 6, and 8 cylinders.

```
mileage4K = predict(tree,[4000 4; 4000 6; 4000 8])
```

```
mileage4K =
```

```
19.2778  
19.2778  
14.3889
```

## Control Regression Tree Depth

You can control the depth of trees using the `MaxNumSplits`, `MinLeafSize`, or `MinParentSize` name-value pair parameters. `fitrtree` grows deep decision trees by default. You can grow shallower trees to reduce model complexity or computation time.

Load the `carsmall` data set. Consider `Displacement`, `Horsepower`, and `Weight` as predictors of the response `MPG`.

```
load carsmall
X = [Displacement Horsepower Weight];
```

The default values of the tree-depth controllers for growing regression trees are:

- `n - 1` for `MaxNumSplits`. `n` is the training sample size.
- `1` for `MinLeafSize`.
- `10` for `MinParentSize`.

These default values tend to grow deep trees for large training sample sizes.

Train a regression tree using the default values for tree-depth control. Cross validate the model using 10-fold cross validation.

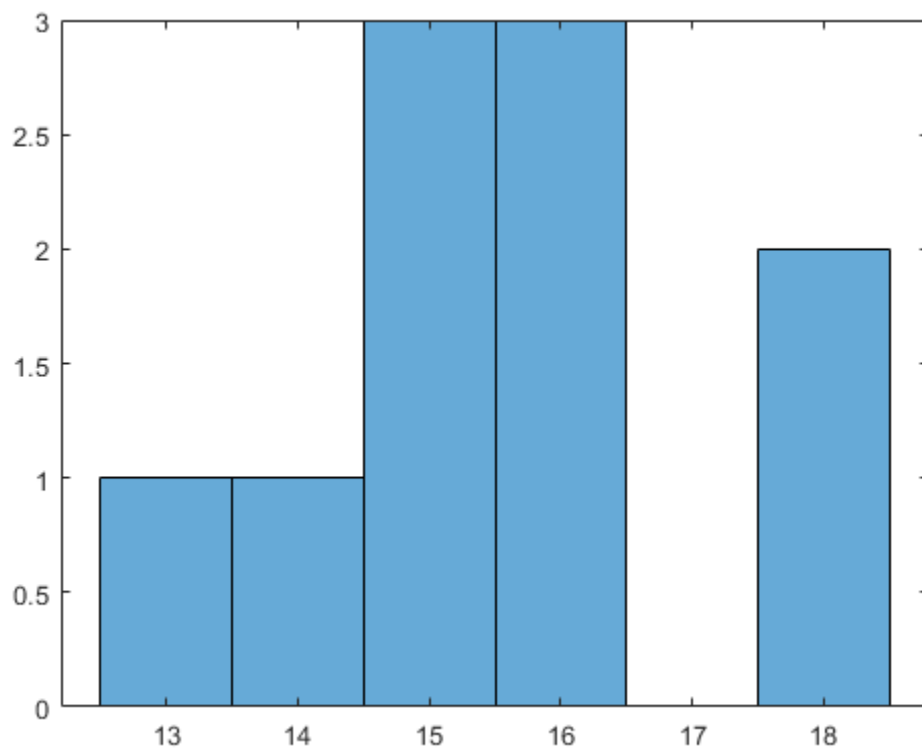
```
rng(1); % For reproducibility
MdlDefault = fitrtree(X,MPG,'CrossVal','on');
```

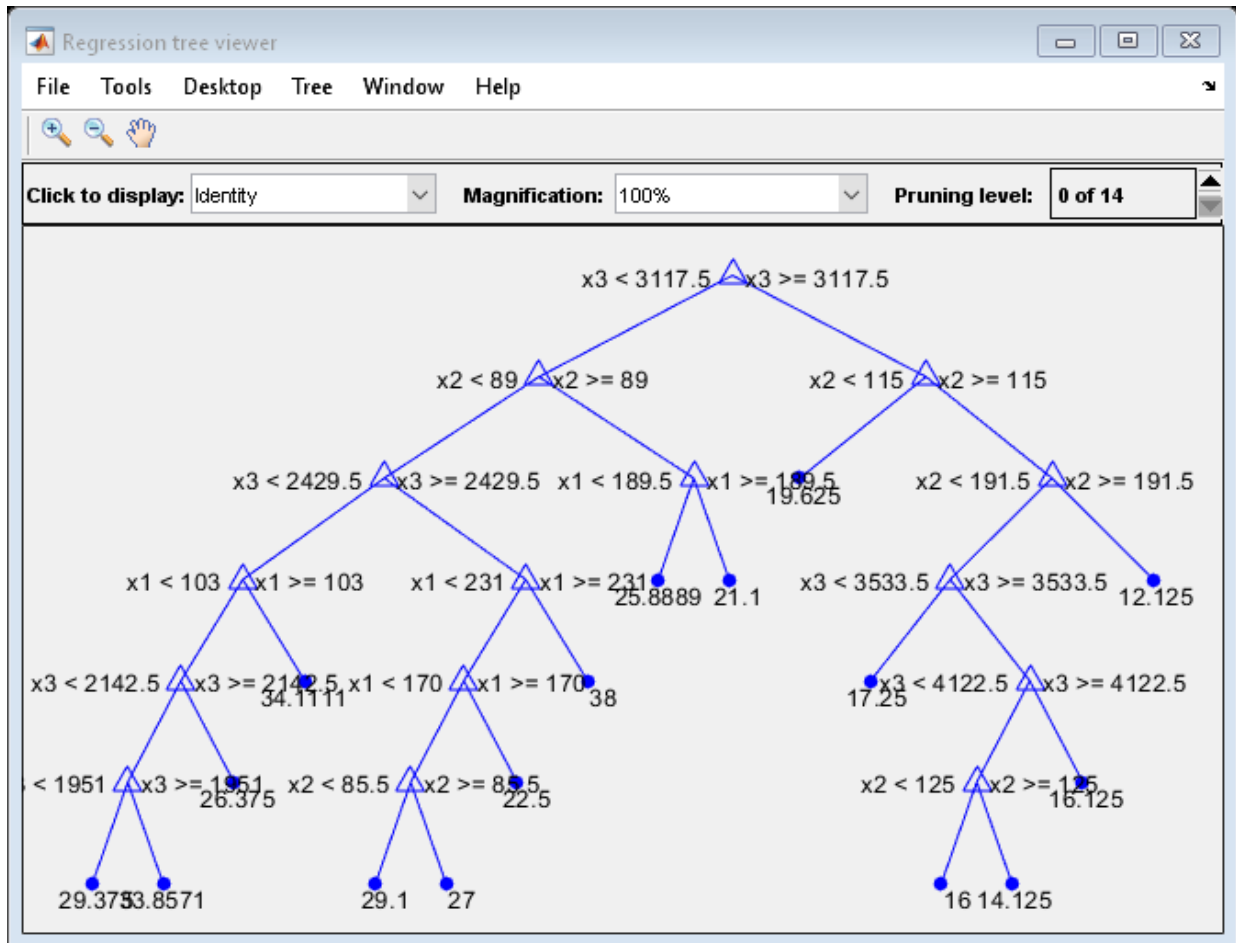
Draw a histogram of the number of imposed splits on the trees. The number of imposed splits is one less than the number of leaves. Also, view one of the trees.

```
numBranches = @(x) sum(x.IsBranch);
mdlDefaultNumSplits = cellfun(numBranches, MdlDefault.Trained);

figure;
histogram(mdlDefaultNumSplits)

view(MdlDefault.Trained{1}, 'Mode', 'graph')
```

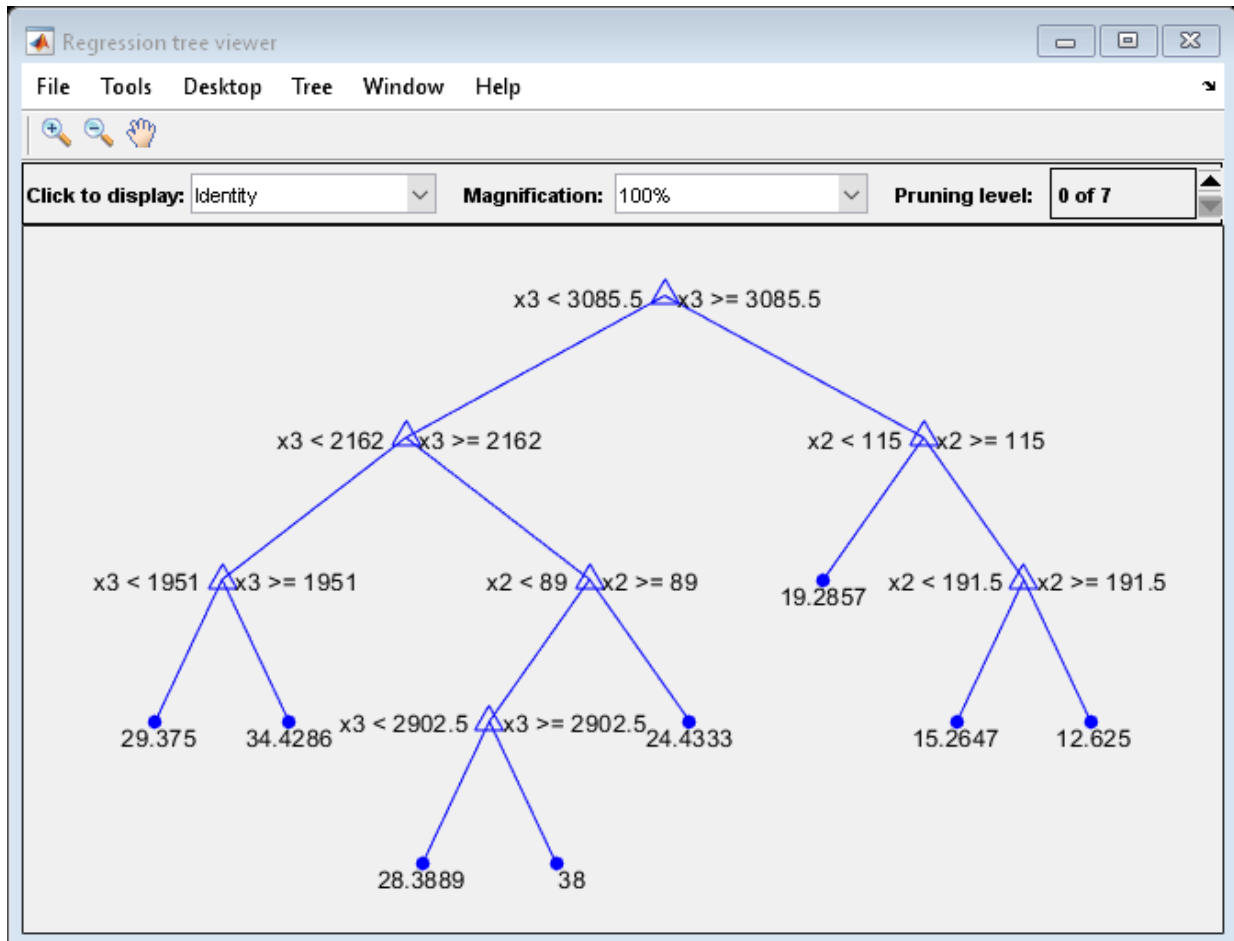




The average number of splits is between 14 and 15.

Suppose that you want a regression tree that is not as complex (deep) as the ones trained using the default number of splits. Train another regression tree, but set the maximum number of splits at 7, which is about half the mean number of splits from the default regression tree. Cross validate the model using 10-fold cross validation.

```
Mdl7 = fitrtree(X,MPG,'MaxNumSplits',7,'CrossVal','on');
view(Mdl7.Trained{1},'Mode','graph')
```



Compare the cross validation MSEs of the models.

```
mseDefault = kfoldLoss(MdlDefault)
mse7 = kfoldLoss(Mdl7)
```

```
mseDefault =
    27.7277
```

```
mse7 =
    28.3833
```

Mdl17 is much less complex and performs only slightly worse than MdlDefault.

### Optimize Regression Tree

This example shows how to optimize hyperparameters automatically using `fitrtree`. The example uses the `carsmall` data.

Load the `carsmall` data.

```
load carsmall
```

Use `Weight` and `Horsepower` as predictors for `MPG`. Find hyperparameters that minimize five-fold cross-validation loss by using automatic hyperparameter optimization.

For reproducibility, set the random seed and use the 'expected-improvement-plus' acquisition function.

```
X = [Weight,Horsepower];
Y = MPG;
rng default
Mdl = fitrtree(X,Y,'OptimizeHyperparameters','auto',...
    'HyperparameterOptimizationOptions',struct('AcquisitionFunctionName',...
    'expected-improvement-plus'))
```

```
=====
| Iter | Eval  | Objective | Objective | BestSoFar | BestSoFar | MinLeafSize
|      | result|           | runtime   | (observed)| (estim.)  |
|=====|=====|=====|=====|=====|=====|=====
|  1  | Best  | 3.2818 | 0.42715 | 3.2818 | 3.2818 | 28
|  2  | Accept| 3.4183 | 0.12596 | 3.2818 | 3.2888 | 1
|  3  | Best  | 3.1491 | 0.11258 | 3.1491 | 3.166  | 4
|  4  | Best  | 2.9885 | 0.12289 | 2.9885 | 2.9885 | 9
|  5  | Accept| 2.9978 | 0.094495| 2.9885 | 2.9885 | 7
|  6  | Accept| 3.0203 | 0.09076 | 2.9885 | 3.0013 | 8
|  7  | Accept| 2.9885 | 0.12047 | 2.9885 | 2.9981 | 9
|  8  | Best  | 2.9589 | 0.081065| 2.9589 | 2.985  | 10
|  9  | Accept| 3.0459 | 0.11128 | 2.9589 | 2.9895 | 12
```

	10	Accept		4.1881		0.11744		2.9589		2.9594		50
	11	Accept		3.4182		0.078192		2.9589		2.9594		2
	12	Accept		3.0376		0.096051		2.9589		2.9592		6
	13	Accept		3.1453		0.088487		2.9589		2.9856		19
	14	Accept		2.9589		0.10087		2.9589		2.9591		10
	15	Accept		2.9589		0.11013		2.9589		2.959		10
	16	Accept		2.9589		0.14526		2.9589		2.959		10
	17	Accept		3.3055		0.1037		2.9589		2.959		3
	18	Accept		3.4577		0.085801		2.9589		2.9589		37
	19	Accept		3.1584		0.075969		2.9589		2.9589		15
	20	Accept		3.107		0.10569		2.9589		2.9589		5
=====												
	Iter	Eval		Objective		Objective		BestSoFar		BestSoFar		MinLeafSize
		result				runtime		(observed)		(estim.)		
=====												
	21	Accept		3.0398		0.10707		2.9589		2.9589		23
	22	Accept		3.3226		0.16452		2.9589		2.9589		32
	23	Accept		3.1883		0.13647		2.9589		2.9589		17
	24	Accept		4.1881		0.15153		2.9589		2.9589		43
	25	Accept		3.0123		0.1429		2.9589		2.9589		11
	26	Accept		3.0932		0.11948		2.9589		2.9589		21
	27	Accept		3.078		0.12237		2.9589		2.9589		13
	28	Accept		3.2818		0.10059		2.9589		2.9589		25
	29	Accept		3.0992		0.15593		2.9589		2.9589		14
	30	Accept		3.4361		0.08788		2.9589		2.9589		34

---

Optimization completed.

MaxObjectiveEvaluations of 30 reached.

Total function evaluations: 30

Total elapsed time: 70.3173 seconds.

Total objective function evaluation time: 3.683

Best observed feasible point:

MinLeafSize

---

10

Observed objective function value = 2.9589

Estimated objective function value = 2.9589

Function evaluation time = 0.081065

Best estimated feasible point (according to models):

```
MinLeafSize
```

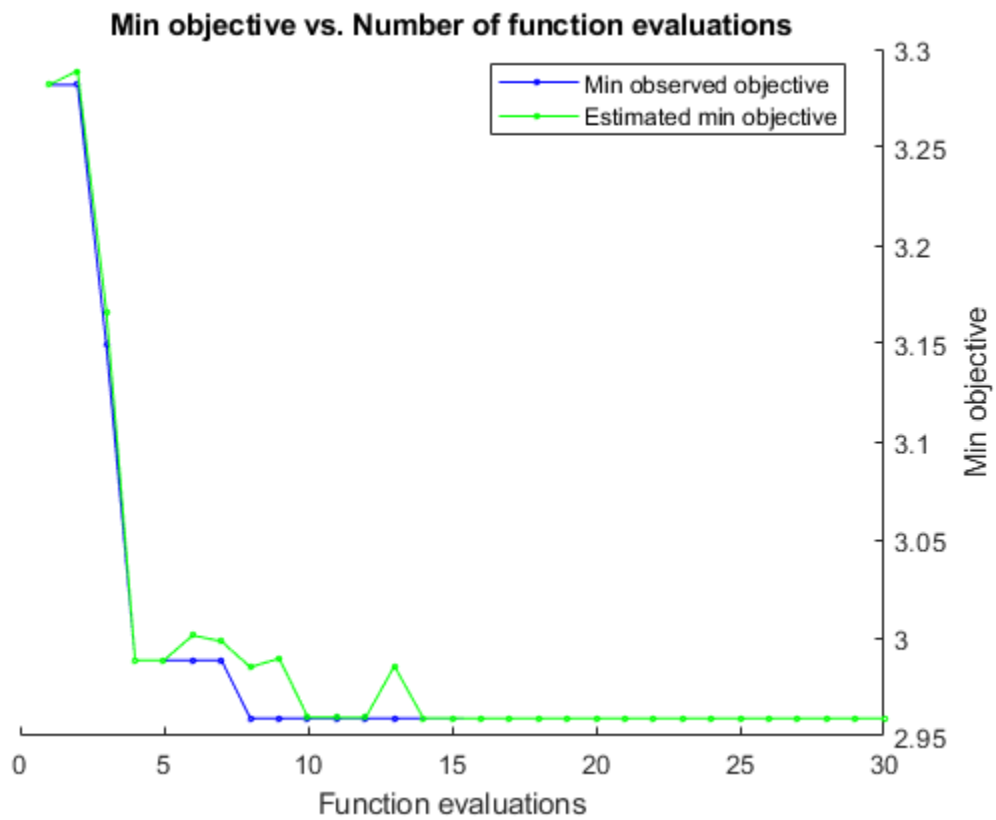
```
-----  
10
```

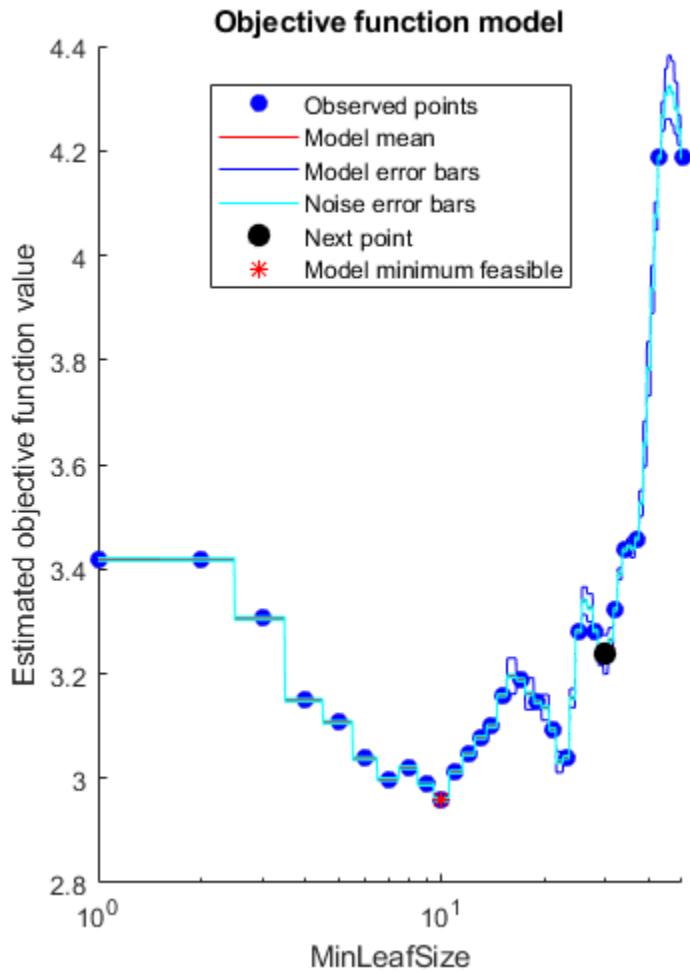
```
Estimated objective function value = 2.9589  
Estimated function evaluation time = 0.11496
```

```
Mdl =
```

```
RegressionTree  
      ResponseName: 'Y'  
      CategoricalPredictors: []  
      ResponseTransform: 'none'  
      NumObservations: 94  
      HyperparameterOptimizationResults: [1x1 BayesianOptimization]
```







### Unbiased Predictor Importance Estimates

Load the `carsmall` data set. Consider a model that predicts the mean fuel economy of a car given its acceleration, number of cylinders, engine displacement, horsepower,

manufacturer, model year, and weight. Consider `Cylinders`, `Mfg`, and `Model_Year` as categorical variables.

```
load carsmall
Cylinders = categorical(Cylinders);
Mfg = categorical(cellstr(Mfg));
Model_Year = categorical(Model_Year);
X = table(Acceleration,Cylinders,Displacement,Horsepower,Mfg,...
         Model_Year,Weight,MPG);
```

Display the number of categories represented in the categorical variables.

```
numCylinders = numel(categories(Cylinders))
numMfg = numel(categories(Mfg))
numModelYear = numel(categories(Model_Year))
```

```
numCylinders =
```

```
3
```

```
numMfg =
```

```
28
```

```
numModelYear =
```

```
3
```

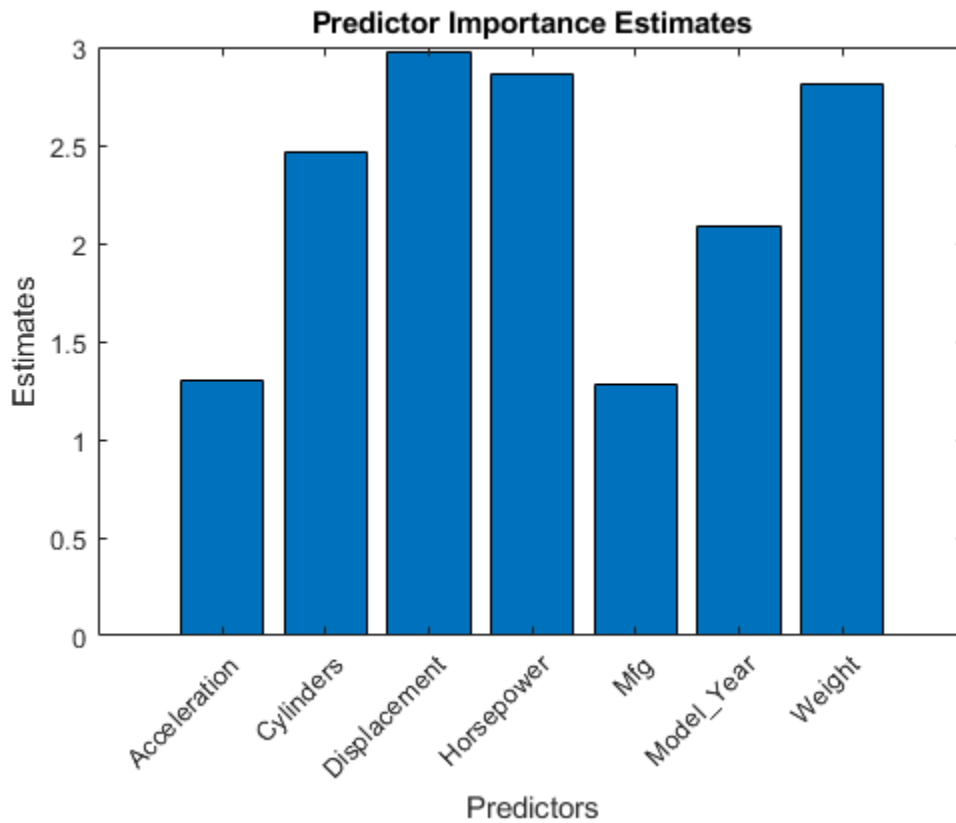
Because there are 3 categories only in `Cylinders` and `Model_Year`, the standard CART, predictor-splitting algorithm prefers splitting a continuous predictor over these two variables.

Train a regression tree using the entire data set. To grow unbiased trees, specify usage of the curvature test for splitting predictors. Because there are missing values in the data, specify usage of surrogate splits.

```
Mdl = fitrtree(X, 'MPG', 'PredictorSelection', 'curvature', 'Surrogate', 'on');
```

Estimate predictor importance values by summing changes in the risk due to splits on every predictor and dividing the sum by the number of branch nodes. Compare the estimates using a bar graph.

```
imp = predictorImportance(Mdl);  
  
figure;  
bar(imp);  
title('Predictor Importance Estimates');  
ylabel('Estimates');  
xlabel('Predictors');  
h = gca;  
h.XTickLabel = Mdl.PredictorNames;  
h.XTickLabelRotation = 45;  
h.TickLabelInterpreter = 'none';
```



In this case, `Displacement` is the most important predictor, followed by `Horsepower`.

## Input Arguments

### **Tbl** — Sample data

table

Sample data used to train the model, specified as a table. Each row of `Tbl` corresponds to one observation, and each column corresponds to one predictor variable. Optionally, `Tbl` can contain one additional column for the response variable. Multi-column variables and cell arrays other than cell arrays of character vectors are not allowed.

If `Tbl` contains the response variable, and you want to use all remaining variables in `Tbl` as predictors, then specify the response variable using `ResponseVarName`.

If `Tbl` contains the response variable, and you want to use only a subset of the remaining variables in `Tbl` as predictors, then specify a formula using `formula`.

If `Tbl` does not contain the response variable, then specify a response variable using `Y`. The length of response variable and the number of rows of `Tbl` must be equal.

Data Types: `table`

### **ResponseVarName** — Response variable name

name of variable in `Tbl`

Response variable name, specified as the name of a variable in `Tbl`. The response variable must be a numeric vector.

You must specify `ResponseVarName` as a character vector. For example, if `Tbl` stores the response variable `Y` as `Tbl.Y`, then specify it as `'Y'`. Otherwise, the software treats all columns of `Tbl`, including `Y`, as predictors when training the model.

### **formula** — Explanatory model of response and subset of predictor variables

character vector

Explanatory model of the response and a subset of the predictor variables, specified as a character vector in the form of `'Y~X1+X2+X3'`. In this form, `Y` represents the response variable, and `X1`, `X2`, and `X3` represent the predictor variables. The variables must be variable names in `Tbl` (`Tbl.Properties.VariableNames`).

To specify a subset of variables in `Tbl` as predictors for training the model, use a formula. If you specify a formula, then the software does not use any variables in `Tbl` that do not appear in `formula`.

Data Types: `char`

### **`y` — Response data**

numeric column vector

Response data, specified as a numeric column vector with the same number of rows as `X`. Each entry in `Y` is the response to the data in the corresponding row of `X`.

The software considers NaN values in `Y` to be missing values. `fitrtree` does not use observations with missing values for `Y` in the fit.

Data Types: `single` | `double`

### **`x` — Predictor data**

numeric matrix

Predictor data, specified as numeric matrix. Each column of `X` represents one variable, and each row represents one observation.

`fitrtree` considers NaN values in `X` as missing values. `fitrtree` does not use observations with all missing values for `X` the fit. `fitrtree` uses observations with some missing values for `X` to find splits on variables for which these observations have valid values.

Data Types: `single` | `double`

## **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

---

**Note** You cannot use any cross-validation name-value pair along with `OptimizeHyperparameters`. You can modify the cross-validation for `OptimizeHyperparameters` only by using the `HyperparameterOptimizationOptions` name-value pair.

---

Example: `'CrossVal', 'on', 'MinParentSize', 30` specifies a cross-validated regression tree with a minimum of 30 observations per branch node.

### Model Parameters

#### **CategoricalPredictors** — Categorical predictors list

vector of positive integers | logical vector | character matrix | cell array of character vectors | `'all'`

Categorical predictors list, specified as the comma-separated pair consisting of `'CategoricalPredictors'` and one of these values:

Value	Description
Vector of positive integers	An entry in the vector is the index value corresponding to the column of the predictor data (X or Tbl) that contains a categorical variable.
Logical vector	A true entry means that the corresponding column of predictor data (X or Tbl) is a categorical variable.
Character matrix	Each row of the matrix is the name of a predictor variable. The names must match the entries in <code>PredictorNames</code> . Pad the names with extra blanks so each row of the character matrix has the same length.
Cell array of character vectors	Each element in the array is the name of a predictor variable. The names must match the entries in <code>PredictorNames</code> .
<code>'all'</code>	All predictors are categorical.

By default, if the predictor data is in a table (Tbl), `fitrtree` assumes that a variable is categorical if it contains logical values, categorical values, or a cell array of character vectors. If the predictor data is a matrix (X), `fitrtree` assumes all predictors are continuous. To identify any categorical predictors when the data is a matrix, use the `'CategoricalPredictors'` name-value pair argument.

Example: `'CategoricalPredictors', 'all'`

Data Types: single | double | logical | char | cell

#### **MergeLeaves** — Leaf merge flag

`'on'` (default) | `'off'`

Leaf merge flag, specified as the comma-separated pair consisting of 'MergeLeaves' and 'on' or 'off'.

If MergeLeaves is 'on', then fitrtree:

- Merges leaves that originate from the same parent node, and that yields a sum of risk values greater or equal to the risk associated with the parent node
- Estimates the optimal sequence of pruned subtrees, but does not prune the regression tree

Otherwise, fitrtree does not merge leaves.

Example: 'MergeLeaves', 'off'

### **MinParentSize** — Minimum number of branch node observations

10 (default) | positive integer value

Minimum number of branch node observations, specified as the comma-separated pair consisting of 'MinParentSize' and a positive integer value. Each branch node in the tree has at least MinParentSize observations. If you supply both MinParentSize and MinLeafSize, fitrtree uses the setting that gives larger leaves: `MinParentSize = max(MinParentSize, 2*MinLeafSize)`.

Example: 'MinParentSize', 8

Data Types: single | double

### **PredictorNames** — Predictor variable names

cell array of unique character vectors

Predictor variable names, specified as the comma-separated pair consisting of 'PredictorNames' and a cell array of unique character vectors. The functionality of 'PredictorNames' depends on the way you supply the training data.

- If you supply X and Y, then you can use 'PredictorNames' to give the predictor variables in X names.
  - The order of the names in PredictorNames must correspond to the column order of X. That is, `PredictorNames{1}` is the name of `X(:, 1)`, `PredictorNames{2}` is the name of `X(:, 2)`, and so on. Also, `size(X, 2)` and `numel(PredictorNames)` must be equal.



- By default, `PredictorNames` is `{'x1', 'x2', ...}`.
- If you supply `Tbl`, then you can use `'PredictorNames'` to choose which predictor variables to use in training. That is, `fitrtree` uses the predictor variables in `PredictorNames` and the response only in training.
  - `PredictorNames` must be a subset of `Tbl.Properties.VariableNames` and cannot include the name of the response variable.
  - By default, `PredictorNames` contains the names of all predictor variables.
  - It good practice to specify the predictors for training using one of `'PredictorNames'` or formula only.

Example: `'PredictorNames'`,  
`{'SepalLength', 'SepalWidth', 'PedalLength', 'PedalWidth'}`

Data Types: `cell`

#### **PredictorSelection** — Algorithm used to select the best split predictor

`'allsplits'` (default) | `'curvature'` | `'interaction-curvature'`

Algorithm used to select the best split predictor at each node, specified as the comma-separated pair consisting of `'PredictorSelection'` and a value in this table.

Value	Description
<code>'allsplits'</code>	Standard CART — Selects the split predictor that maximizes the split-criterion gain over all possible splits of all predictors [1].
<code>'curvature'</code>	Curvature test on page 32-2662 — Selects the split predictor that minimizes the $p$ -value of chi-square tests of independence between each predictor and the response [2]. Training speed is similar to standard CART.
<code>'interaction-curvature'</code>	Interaction test on page 32-2663 — Chooses the split predictor that minimizes the $p$ -value of chi-square tests of independence between each predictor and the response (that is, conducts curvature tests), and that minimizes the $p$ -value of a chi-square test of independence between each pair of predictors and response [2]. Training speed can be slower than standard CART.

For `'curvature'` and `'interaction-curvature'`, if all tests yield  $p$ -values greater than 0.05, then `fitrtree` stops splitting nodes.

---

**Tip**

- Standard CART tends to select split predictors containing many distinct values, e.g., continuous variables, over those containing few distinct values, e.g., categorical variables [3]. Consider specifying the curvature or interaction test if any of the following are true:
    - If there are predictors that have relatively fewer distinct values than other predictors, for example, if the predictor data set is heterogeneous.
    - If an analysis of predictor importance is your goal. For more on predictor importance estimation, see `predictorImportance`.
  - Trees grown using standard CART are not sensitive to predictor variable interactions. Also, such trees are less likely to identify important variables in the presence of many irrelevant predictors than the application of the interaction test. Therefore, to account for predictor interactions and identify importance variables in the presence of many irrelevant variables, specify the interaction test .
  - Prediction speed is unaffected by the value of `'PredictorSelection'`.
- 

For details on how `fitrtree` selects split predictors, see “Node Splitting Rules” on page 32-2665.

Example: `'PredictorSelection', 'curvature'`

Data Types: `char`

**Prune — Flag to estimate optimal sequence of pruned subtrees**

`'on'` (default) | `'off'`

Flag to estimate the optimal sequence of pruned subtrees, specified as the comma-separated pair consisting of `'Prune'` and `'on'` or `'off'`.

If `Prune` is `'on'`, then `fitrtree` grows the regression tree and estimates the optimal sequence of pruned subtrees, but does not prune the regression tree. Otherwise, `fitrtree` grows the regression tree without estimating the optimal sequence of pruned subtrees.

To prune a trained regression tree, pass the regression tree to `prune`.

Example: `'Prune', 'off'`

**PruneCriterion — Pruning criterion**`'mse'` (default)

Pruning criterion, specified as the comma-separated pair consisting of `'PruneCriterion'` and `'mse'`.

**QuadraticErrorTolerance — Quadratic error tolerance**`1e-6` (default) | positive scalar value

Quadratic error tolerance per node, specified as the comma-separated pair consisting of `'QuadraticErrorTolerance'` and a positive scalar value. Splitting nodes stops when quadratic error per node drops below `QuadraticErrorTolerance*QED`, where `QED` is the quadratic error for the entire data computed before the decision tree is grown.

Example: `'QuadraticErrorTolerance', 1e-4`

**ResponseName — Response variable name**`'Y'` (default) | character vector

Response variable name, specified as the comma-separated pair consisting of `'ResponseName'` and a character vector.

- If you supply `Y`, then you can use `'ResponseName'` to specify a name for the response variable.
- If you supply `ResponseVarName` or formula, then you cannot use `'ResponseName'`.

Example: `'ResponseName', 'response'`

Data Types: char

**ResponseTransform — Response transformation**`'none'` (default) | function handle

Response transformation, specified as the comma-separated pair consisting of `'ResponseTransform'` and either `'none'` or a function handle. The default is `'none'`, which means  $@(x) x$ , or no transformation. For a MATLAB function or a function you define, use its function handle. The function handle must accept a vector (the original response values) and return a vector of the same size (the transformed response values).

Example: `'ResponseTransform', 'none'`

Data Types: char | function\_handle

**SplitCriterion — Split criterion**`'MSE'` (default)

Split criterion, specified as the comma-separated pair consisting of `'SplitCriterion'` and `'MSE'`, meaning mean squared error.

Example: `'SplitCriterion', 'MSE'`

**Surrogate — Surrogate decision splits flag**`'off'` (default) | `'on'` | `'all'` | positive integer

Surrogate decision splits flag, specified as the comma-separated pair consisting of `'Surrogate'` and `'on'`, `'off'`, `'all'`, or a positive integer.

- When `'on'`, `fitrtree` finds at most 10 surrogate splits at each branch node.
- When set to a positive integer, `fitrtree` finds at most the specified number of surrogate splits at each branch node.
- When set to `'all'`, `fitrtree` finds all surrogate splits at each branch node. The `'all'` setting can use much time and memory.

Use surrogate splits to improve the accuracy of predictions for data with missing values. The setting also enables you to compute measures of predictive association between predictors.

Example: `'Surrogate', 'on'`

Data Types: `single` | `double` | `char`

**Weights — Observation weights**`ones(size(X,1),1)` (default) | vector of scalar values

Observation weights, specified as the comma-separated pair consisting of `'Weights'` and a vector of scalar values. The software weights the observations in each row of `X` or `Tbl` with the corresponding value in `Weights`. The size of `Weights` must equal the number of rows in `X` or `Tbl`.

If you specify the input data as a table `Tbl`, then `Weights` can be the name of a variable in `Tbl` that contains a numeric vector. In this case, you must specify `Weights` as a character vector. For example, if weights vector `W` is stored as `Tbl.W`, then specify it as `'W'`. Otherwise, the software treats all columns of `Tbl`, including `W`, as predictors when training the model.

fitrtree normalizes the weights in each class to add up to 1.

Data Types: `single` | `double`

### Cross Validation

#### **CrossVal** — Cross-validation flag

`'off'` (default) | `'on'`

Cross-validation flag, specified as the comma-separated pair consisting of `'CrossVal'` and either `'on'` or `'off'`.

If `'on'`, fitrtree grows a cross-validated decision tree with 10 folds. You can override this cross-validation setting using one of the `'KFold'`, `'Holdout'`, `'Leaveout'`, or `'CVPartition'` name-value pair arguments. You can only use one of these four options (`'KFold'`, `'Holdout'`, `'Leaveout'`, or `'CVPartition'`) at a time when creating a cross-validated tree.

Alternatively, cross-validate tree later using the `crossval` method.

Example: `'CrossVal','on'`

#### **CVPartition** — Partition for cross-validation tree

`cvpartition` object

Partition for cross-validated tree, specified as the comma-separated pair consisting of `'CVPartition'` and an object created using `cvpartition`.

If you use `'CVPartition'`, you cannot use any of the `'KFold'`, `'Holdout'`, or `'Leaveout'` name-value pair arguments.

#### **Holdout** — Fraction of data for holdout validation

0 (default) | scalar value in the range `[0,1]`

Fraction of data used for holdout validation, specified as the comma-separated pair consisting of `'Holdout'` and a scalar value in the range `[0,1]`. Holdout validation tests the specified fraction of the data, and uses the rest of the data for training.

If you use `'Holdout'`, you cannot use any of the `'CVPartition'`, `'KFold'`, or `'Leaveout'` name-value pair arguments.

Example: `'Holdout',0.1`

Data Types: `single` | `double`

**KFold** — Number of folds

10 (default) | positive integer greater than 1

Number of folds to use in a cross-validated tree, specified as the comma-separated pair consisting of 'KFold' and a positive integer value greater than 1.

If you use 'KFold', you cannot use any of the 'CVPartition', 'Holdout', or 'Leaveout' name-value pair arguments.

Example: 'KFold', 8

Data Types: `single` | `double`

**Leaveout** — Leave-one-out cross-validation flag

'off' (default) | 'on'

Leave-one-out cross-validation flag, specified as the comma-separated pair consisting of 'Leaveout' and either 'on' or 'off'. Use leave-one-out cross validation by setting to 'on'.

If you use 'Leaveout', you cannot use any of the 'CVPartition', 'Holdout', or 'KFold' name-value pair arguments.

Example: 'Leaveout', 'on'

**Hyperparameters****MaxNumSplits** — Maximal number of decision splits

$\text{size}(X,1) - 1$  (default) | positive integer

Maximal number of decision splits (or branch nodes), specified as the comma-separated pair consisting of 'MaxNumSplits' and a positive integer. `fitrtree` splits `MaxNumSplits` or fewer branch nodes. For more details on splitting behavior, see “Tree Depth Control” on page 32-2668.

Example: 'MaxNumSplits', 5

Data Types: `single` | `double`

**MinLeafSize** — Minimum number of leaf node observations

1 (default) | positive integer value

Minimum number of leaf node observations, specified as the comma-separated pair consisting of `'MinLeafSize'` and a positive integer value. Each leaf has at least `MinLeafSize` observations per tree leaf. If you supply both `MinParentSize` and `MinLeafSize`, `fitree` uses the setting that gives larger leaves: `MinParentSize = max(MinParentSize, 2*MinLeafSize)`.

Example: `'MinLeafSize', 3`

Data Types: `single` | `double`

### **NumVariablesToSample** — Number of predictors for split

`'all'` (default) | positive integer value

Number of predictors to select at random for each split, specified as the comma-separated pair consisting of `'NumVariablesToSample'` and a positive integer value. You can also specify `'all'` to use all available predictors.

Example: `'NumVariablesToSample', 3`

Data Types: `single` | `double`

### **Hyperparameter Optimization**

#### **OptimizeHyperparameters** — Parameters to optimize

`'none'` (default) | `'auto'` | `'all'` | cell array of eligible parameter names | vector of `optimizableVariable` objects

Parameters to optimize, specified as:

- `'none'` — Do not optimize.
- `'auto'` — Use `{'MinLeafSize'}`.
- `'all'` — Optimize all eligible parameters.
- Cell array of eligible parameter names
- Vector of `optimizableVariable` objects, typically the output of `hyperparameters`

The optimization attempts to minimize the cross-validation loss (error) for `fitree` by varying the parameters. To control the cross-validation type and other aspects of the optimization, use the `HyperparameterOptimizationOptions` name-value pair.

---

**Note** `OptimizeHyperparameters` values override any values you set using other name-value pairs. For example, setting `OptimizeHyperparameters` to 'auto' causes the 'auto' values to apply.

---

The eligible parameters for `fitrtree` are:

- `MaxNumSplits` — `fitrtree` searches among integers, by default log-scaled in the range `[1, max(2, NumObservations-1)]`.
- `MinLeafSize` — `fitrtree` searches among integers, by default log-scaled in the range `[1, max(2, floor(NumObservations/2))]`.
- `NumVariablesToSample` — `fitrtree` does not optimize over this hyperparameter. If you pass `NumVariablesToSample` as a parameter name, `fitrtree` simply uses the full number of predictors. However, `fitrensemble` does optimize over this hyperparameter.

Set nondefault parameters by passing a vector of `optimizableVariable` objects that have nondefault values. For example,

```
load carsmall
params = hyperparameters('fitrtree', [Horsepower, Weight], MPG);
params(1).Range = [1, 30];
```

Pass `params` as the value of `OptimizeHyperparameters`.

By default, iterative display appears at the command line, and plots appear according to the number of hyperparameters in the optimization. For the optimization and plots, the objective function is  $\log(1 + \text{cross-validation loss})$  for regression, and the misclassification rate for classification. To control the iterative display, set the `HyperparameterOptimizationOptions` name-value pair, `Verbose` field. To control the plots, set the `HyperparameterOptimizationOptions` name-value pair, `ShowPlots` field.

For an example, see “Optimize Regression Tree” on page 32-2640.

Example: 'auto'

Data Types: char | cell

**HyperparameterOptimizationOptions** — Options for optimization structure



Options for optimization, specified as a structure. Modifies the effect of the `OptimizeHyperparameters` name-value pair. All fields in the structure are optional.

Field Name	Values	Default
Optimizer	<ul style="list-style-type: none"> <li>• 'bayesopt' — Use Bayesian optimization. Internally, this setting calls <code>bayesopt</code>.</li> <li>• 'gridsearch' — Use grid search with <code>NumGridDivisions</code> values per dimension.</li> <li>• 'randomsearch' — Search at random among <code>MaxObjectiveEvaluations</code> points.</li> </ul> <p>'gridsearch' searches in a random order, using uniform sampling without replacement from the grid. After optimization, you can get a table in grid order by using the command <code>sortrows(Mdl.ParameterOptimizationResults)</code>.</p>	'bayesopt'
AcquisitionFunctionName	<ul style="list-style-type: none"> <li>• 'expected-improvement-per-second-plus'</li> <li>• 'expected-improvement'</li> <li>• 'expected-improvement-plus'</li> <li>• 'expected-improvement-per-second'</li> <li>• 'lower-confidence-bound'</li> <li>• 'probability-of-improvement'</li> </ul> <p>For details, see the <code>bayesopt</code> <code>AcquisitionFunctionName</code> name-value pair, or “Acquisition Function Types” on page 10-4.</p>	'expected-improvement-per-second-plus'

Field Name	Values	Default
MaxObjectiveEvaluations	Maximum number of objective function evaluations.	30 for 'bayesopt' or 'randomsearch', and the entire grid for 'gridsearch'
MaxTime	Time limit, specified as a positive real. The time limit is in seconds, as measured by <code>tic</code> and <code>toc</code> . Run time can exceed <code>MaxTime</code> because <code>MaxTime</code> does not interrupt function evaluations.	Inf
NumGridDivisions	For 'gridsearch', the number of values in each dimension. Can be a vector of positive integers giving the number of values for each dimension, or a scalar that applies to all dimensions. Ignored for categorical variables.	10
ShowPlots	Logical value indicating whether to show plots. If <code>true</code> , plots the best objective function value against iteration number. If there are one or two optimization parameters, and if <code>Optimizer</code> is 'bayesopt', then <code>ShowPlots</code> also plots a model of the objective function against the parameters.	<code>true</code>
SaveIntermediateResults	Logical value indicating whether to save results when <code>Optimizer</code> is 'bayesopt'. If <code>true</code> , overwrites a workspace variable named 'BayesoptResults' at each iteration. The variable is a <code>BayesianOptimization</code> object.	<code>false</code>

Field Name	Values	Default
Verbose	Display to the command line. <ul style="list-style-type: none"> <li>• 0 — No iterative display</li> <li>• 1 — Iterative display</li> <li>• 2 — Iterative display with extra information</li> </ul> For details, see the <code>bayesopt</code> Verbose name-value pair.	1
UseParallel	Logical value indicating whether to run Bayesian optimization in parallel, which requires Parallel Computing Toolbox. For details, see “Parallel Bayesian Optimization” on page 10-9.	false
Repartition	Logical value indicating whether to repartition the cross-validation at every iteration. If <code>false</code> , the optimizer uses a single partition for the optimization.  <code>true</code> usually gives the most robust results because this setting takes partitioning noise into account. However, for good results, <code>true</code> requires at least twice as many function evaluations.	false
Use no more than one of the following three field names.		
CVPartition	A <code>cvpartition</code> object, as created by <code>cvpartition</code>	Kfold = 5
Holdout	A scalar in the range (0,1) representing the holdout fraction.	
Kfold	An integer greater than 1.	

Example: `struct('MaxObjectiveEvaluations',60)`

Data Types: `struct`

## Output Arguments

### **tree** — Regression tree

regression tree object

Regression tree, returned as a regression tree object. Using the 'Crossval', 'KFold', 'Holdout', 'Leaveout', or 'CVPartition' options results in a tree of class `RegressionPartitionedModel`. You cannot use a partitioned tree for prediction, so this kind of tree does not have a `predict` method.

Otherwise, `tree` is of class `RegressionTree`, and you can use the `predict` method to make predictions.

## Definitions

### Curvature Test

The curvature test is a statistical test assessing the null hypothesis that two variables are unassociated.

The curvature test between predictor variable  $x$  and  $y$  is conducted using this process.

- 1 If  $x$  is continuous, then partition it into its quartiles. Create a nominal variable that bins observations according to which section of the partition they occupy. If there are missing values, then create an extra bin for them.
- 2 For each level in the partitioned predictor  $j = 1 \dots J$  and class in the response  $k = 1, \dots, K$ , compute the weighted proportion of observations in class  $k$

$$\hat{\pi}_{jk} = \sum_{i=1}^n I\{y_i = k\} w_i.$$

$w_i$  is the weight of observation  $i$ ,  $\sum w_i = 1$ ,  $I$  is the indicator function, and  $n$  is the

sample size. If all observations have the same weight, then  $\hat{\pi}_{jk} = \frac{n_{jk}}{n}$ , where  $n_{jk}$  is the number of observations in level  $j$  of the predictor that are in class  $k$ .

- 3 Compute the test statistic

$$t = n \sum_{k=1}^K \sum_{j=1}^J \frac{(\hat{\pi}_{jk} - \hat{\pi}_{j+} \hat{\pi}_{+k})^2}{\hat{\pi}_{j+} \hat{\pi}_{+k}}$$

$\hat{\pi}_{j+} = \sum_k \hat{\pi}_{jk}$ , that is, the marginal probability of observing the predictor at level  $j$ .

$\hat{\pi}_{+k} = \sum_j \hat{\pi}_{jk}$ , that is the marginal probability of observing class  $k$ . If  $n$  is large enough then  $t$  is distributed as a  $\chi^2$  with  $(K-1)(J-1)$  degrees of freedom.

- 4 If the  $p$ -value for the test is less than 0.05, then reject the null hypothesis that there is no association between  $x$  and  $y$ .

When determining the best split predictor at each node, the standard CART algorithm prefers to select continuous predictors that have many levels. Sometimes, such a selection can be spurious and can also mask more important predictors that have fewer levels, such as categorical predictors.

The curvature test can be applied instead of standard CART to determine the best split predictor at each node. In that case, the best split predictor variable is the one that minimizes the significant  $p$ -values (those less than 0.05) of curvature tests between each predictor and the response variable. Such a selection is robust to the number of levels in individual predictors.

For more details on how the curvature test applies to growing regression trees, see “Node Splitting Rules” on page 32-2665 and [3].

## Interaction Test

The interaction test is a statistical test that assesses the null hypothesis that there is no interaction between a pair of predictor variables and the response variable.

The interaction test assessing the association between predictor variables  $x_1$  and  $x_2$  with respect to  $y$  is conducted using this process.

- 1 If  $x_1$  or  $x_2$  is continuous, then partition that variable into its quartiles. Create a nominal variable that bins observations according to which section of the partition they occupy. If there are missing values, then create an extra bin for them.

- 2 Create the nominal variable  $z$  with  $J = J_1 J_2$  levels that assigns an index to observation  $i$  according to which levels of  $x_1$  and  $x_2$  it belongs. Remove any levels of  $z$  that do not correspond to any observations.
- 3 Conduct a curvature test on page 32-2662 between  $z$  and  $y$ .

When growing decision trees, if there are important interactions between pairs of predictors, but there are also many other less important predictors in the data, then standard CART tends to miss the important interactions. However, conducting curvature and interaction tests for predictor selection instead can improve detection of important interactions, which can yield more accurate decision trees.

For more details on how the interaction test applies to growing decision trees, see “Curvature Test” on page 32-2662, “Node Splitting Rules” on page 32-2665 and [2].

## Predictive Measure of Association

The predictive measure of association is a value that indicates the similarity between decision rules that split observations. Among all possible decision splits that are compared to the optimal split (found by growing the tree), the best surrogate decision split on page 32-2174 yields the maximum predictive measure of association. The second-best surrogate split has the second-largest predictive measure of association.

Suppose  $x_j$  and  $x_k$  are predictor variables  $j$  and  $k$ , respectively, and  $j \neq k$ . At node  $t$ , the predictive measure of association between the optimal split  $x_j < u$  and a surrogate split  $x_k < v$  is

$$\lambda_{jk} = \frac{\min(P_L, P_R) - (1 - P_{L_j L_k} - P_{R_j R_k})}{\min(P_L, P_R)}.$$

- $P_L$  is the proportion of observations in node  $t$ , such that  $x_j < u$ . The subscript  $L$  stands for the left child of node  $t$ .
- $P_R$  is the proportion of observations in node  $t$ , such that  $x_j \geq u$ . The subscript  $R$  stands for the right child of node  $t$ .
- $P_{L_j L_k}$  is the proportion of observations at node  $t$ , such that  $x_j < u$  and  $x_k < v$ .
- $P_{R_j R_k}$  is the proportion of observations at node  $t$ , such that  $x_j \geq u$  and  $x_k \geq v$ .
- Observations with missing values for  $x_j$  or  $x_k$  do not contribute to the proportion calculations.

$\lambda_{jk}$  is a value in  $(-\infty, 1]$ . If  $\lambda_{jk} > 0$ , then  $x_k < v$  is a worthwhile surrogate split for  $x_j < u$ .

## Surrogate Decision Splits

A surrogate decision split is an alternative to the optimal decision split at a given node in a decision tree. The optimal split is found by growing the tree; the surrogate split uses a similar or correlated predictor variable and split criterion.

When the value of the optimal split predictor for an observation is missing, the observation is sent to the left or right child node using the best surrogate predictor. When the value of the best surrogate split predictor for the observation is also missing, the observation is sent to the left or right child node using the second-best surrogate predictor, and so on. Candidate splits are sorted in descending order by their predictive measure of association on page 32-2664.

## Tip

- By default, `Prune` is 'on'. However, this specification does not prune the regression tree. To prune a trained regression tree, pass the regression tree to `prune`.
- After training a `RegressionTree` model object by using `fitrtree`, use the function `predict` and the trained model object to generate C code that predicts responses for new data. For details, see “Code Generation” on page 32-0 .

## Algorithms

### Node Splitting Rules

`fitrtree` uses these processes to determine how to split node  $t$ .

- For standard CART (that is, if `PredictorSelection` is 'allpairs') and for all predictors  $x_i$ ,  $i = 1, \dots, p$ :

- 1 `fitrtree` computes the weighted, mean-square error (MSE) of the responses in node  $t$  using

$$\varepsilon_t = \sum_{j \in T} w_j (y_j - \bar{y}_t)^2 .$$

$w_j$  is the weight of observation  $j$ , and  $T$  is the set of all observation indices in node  $t$ . If you do not specify `Weights`, then  $w_j = 1/n$ , where  $n$  is the sample size.

- 2 `fitrtree` estimates the probability that an observation is in node  $t$  using

$$P(T) = \sum_{j \in T} w_j.$$

- 3 `fitrtree` sorts  $x_i$  in ascending order. Each element of the sorted predictor is a splitting candidate or cut point. `fitrtree` records any indices corresponding to missing values in the set  $T_U$ , which is the unsplit set.

- 4 `fitrtree` determines the best way to split node  $t$  using  $x_i$  by maximizing the reduction in MSE ( $\Delta I$ ) over all splitting candidates. That is, for all splitting candidates in  $x_i$ :

- a `fitrtree` splits the observations in node  $t$  into left and right child nodes ( $t_L$  and  $t_R$ , respectively).

- b `fitrtree` computes  $\Delta I$ . Suppose that for a particular splitting candidate,  $t_L$  and  $t_R$  contain observation indices in the sets  $T_L$  and  $T_R$ , respectively.

- If  $x_i$  does not contain any missing values, then the reduction in MSE for the current splitting candidate is

$$\Delta I = P(T)\varepsilon_t - P(T_L)\varepsilon_{t_L} - P(T_R)\varepsilon_{t_R}.$$

- If  $x_i$  contains missing values, then, assuming that the observations are missing at random, the reduction in MSE is

$$\Delta I_U = P(T - T_U)\varepsilon_t - P(T_L)\varepsilon_{t_L} - P(T_R)\varepsilon_{t_R}.$$

$T - T_U$  is the set of all observation indices in node  $t$  that are not missing.

- If you use surrogate decision splits on page 32-2665, then:

- i `fitrtree` computes the predictive measures of association on page 32-2664 between the decision split  $x_i < u$  and all possible decision splits  $x_k < v$ ,  $j \neq k$ .

- ii `fitrtree` sorts the possible alternative decision splits in descending order by their predictive measure of association with the optimal split. The surrogate split is the decision split yielding the largest measure.

- iii `fitrtree` decides the child node assignments for observations with a missing value for  $x_i$  using the surrogate split. If the surrogate



predictor also contains a missing value, then `fitrtree` uses the decision split with the second largest measure, and so on, until there are no other surrogates. It is possible for `fitrtree` to split two different observations at node  $t$  using two different surrogate splits. For example, suppose the predictors  $x_1$  and  $x_2$  are the best and second best surrogates, respectively, for the predictor  $x_i$ ,  $i \in \{1,2\}$ , at node  $t$ . If observation  $m$  of predictor  $x_i$  is missing (i.e.,  $x_{mi}$  is missing), but  $x_{m1}$  is not missing, then  $x_1$  is the surrogate predictor for observation  $x_{mi}$ . If observations  $x_{(m+1),i}$  and  $x_{(m+1),1}$  are missing, but  $x_{(m+1),2}$  is not missing, then  $x_2$  is the surrogate predictor for observation  $m+1$ .

- iv `fitrtree` uses the appropriate MSE reduction formula. That is, if `fitrtree` fails to assign all missing observations in node  $t$  to children nodes using surrogate splits, then the MSE reduction is  $\Delta I_U$ . Otherwise, `fitrtree` uses  $\Delta I$  for the MSE reduction.

- c `fitrtree` chooses the candidate that yields the largest MSE reduction.

`fitrtree` splits the predictor variable at the cut point that maximizes the MSE reduction.

- For the curvature test (that is, if `PredictorSelection` is 'curvature'):
  - 1 `fitrtree` computes the residuals  $r_{ti} = y_{ti} - \bar{y}_t$  for all observations in node  $t$ .

$\bar{y}_t = \frac{1}{\sum_i w_i} \sum_i w_i y_{ti}$ , which is the weighted average of the responses in node  $t$ . The weights are the observation weights in `Weights`.

- 2 `fitrtree` assigns observations to one of two bins according to the sign of the corresponding residuals. Let  $z_i$  be a nominal variable that contains the bin assignments for the observations in node  $t$ .
- 3 `fitrtree` conducts curvature tests on page 32-2662 between each predictor and  $z_i$ . For regression trees,  $K = 2$ .
  - If all  $p$ -values are at least 0.05, then `fitrtree` does not split node  $t$ .
  - If there is a minimal  $p$ -value, then `fitrtree` chooses the corresponding predictor to split node  $t$ .
  - If more than one  $p$ -value is zero due to underflow, then `fitrtree` applies standard CART to the corresponding predictors to choose the split predictor.

- 4 If `fitrtree` chooses a split predictor, then it uses standard CART to choose the cut point (see step 4 in the standard CART process).

For the interaction test (that is, if `PredictorSelection` is 'interaction-curvature'):

- 1 For observations in node  $t$ , `fitrtree` conducts curvature tests on page 32-2662 between each predictor and the response and interaction tests on page 32-2663 between each pair of predictors and the response.
  - If all  $p$ -values are at least 0.05, then `fitrtree` does not split node  $t$ .
  - If there is a minimal  $p$ -value and it is the result of a curvature test, then `fitrtree` chooses the corresponding predictor to split node  $t$ .
  - If there is a minimal  $p$ -value and it is the result of an interaction test, then `fitrtree` chooses the split predictor using standard CART on the corresponding pair of predictors.
  - If more than one  $p$ -value is zero due to underflow, then `fitrtree` applies standard CART to the corresponding predictors to choose the split predictor.
- 2 If `fitrtree` chooses a split predictor, then it uses standard CART to choose the cut point (see step 4 in the standard CART process).

## Tree Depth Control

- If `MergeLeaves` is 'on' and `PruneCriterion` is 'mse' (which are the default values for these name-value pair arguments), then the software applies pruning only to the leaves and by using MSE. This specification amounts to merging leaves coming from the same parent node whose MSE is at most the sum of the MSE of its two leaves.
- To accommodate `MaxNumSplits`, `fitrtree` splits all nodes in the current layer, and then counts the number of branch nodes. A layer is the set of nodes that are equidistant from the root node. If the number of branch nodes exceeds `MaxNumSplits`, `fitrtree` follows this procedure:
  - 1 Determine how many branch nodes in the current layer must be unsplit so that there are at most `MaxNumSplits` branch nodes.
  - 2 Sort the branch nodes by their impurity gains.
  - 3 Unsplit the number of least successful branches.

#### 4 Return the decision tree grown so far.

This procedure produces maximally balanced trees.

- The software splits branch nodes layer by layer until at least one of these events occurs:
  - There are `MaxNumSplits` branch nodes.
  - A proposed split causes the number of observations in at least one branch node to be fewer than `MinParentSize`.
  - A proposed split causes the number of observations in at least one leaf node to be fewer than `MinLeafSize`.
  - The algorithm cannot find a good split within a layer (i.e., the pruning criterion (see `PruneCriterion`), does not improve for all proposed splits in a layer). A special case is when all nodes are pure (i.e., all observations in the node have the same class).
  - For values 'curvature' or 'interaction-curvature' of `PredictorSelection`, all tests yield  $p$ -values greater than 0.05.

`MaxNumSplits` and `MinLeafSize` do not affect splitting at their default values. Therefore, if you set 'MaxNumSplits', splitting might stop due to the value of `MinParentSize`, before `MaxNumSplits` splits occur.

## Parallelization

For dual-core systems and above, `fitrtree` parallelizes training decision trees using Intel Threading Building Blocks (TBB). For details on Intel TBB, see <https://software.intel.com/en-us/intel-tbb>.

## References

- [1] Breiman, L., J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Boca Raton, FL: CRC Press, 1984.
- [2] Loh, W.Y. "Regression Trees with Unbiased Variable Selection and Interaction Detection." *Statistica Sinica*, Vol. 12, 2002, pp. 361–386.
- [3] Loh, W.Y. and Y.S. Shih. "Split Selection Methods for Classification Trees." *Statistica Sinica*, Vol. 7, 1997, pp. 815–840.

## See Also

`RegressionPartitionedModel` | `RegressionTree` | `predict` | `prune` | `surrogateAssociation`

## Topics

“Splitting Categorical Predictors” on page 19-42

**Introduced in R2014a**

# fitSVMPosterior

Fit posterior probabilities

## Syntax

```
ScoreSVMModel = fitSVMPosterior(SVMModel)

ScoreSVMModel = fitSVMPosterior(SVMModel, TBL, ResponseVarName)
ScoreSVMModel = fitSVMPosterior(SVMModel, TBL, Y)

ScoreSVMModel = fitSVMPosterior(SVMModel, X, Y)

ScoreSVMModel = fitSVMPosterior(____, Name, Value)

[ScoreSVMModel, ScoreTransform] = fitSVMPosterior(____)
```

## Description

`ScoreSVMModel = fitSVMPosterior(SVMModel)` returns `ScoreSVMModel`, which is a trained, support vector machine (SVM) classifier containing the optimal score-to-posterior-probability transformation function for two-class learning.

The software fits the appropriate score-to-posterior-probability transformation function using the SVM classifier `SVMModel`, and by cross validation using the stored predictor data (`SVMModel.X`) and the class labels (`SVMModel.Y`). The transformation function computes the posterior probability that an observation is classified into the positive class (`SVMModel.Classnames(2)`).

- If the classes are inseparable, then the transformation function is the sigmoid function on page 32-2682.
- If the classes are perfectly separable, the transformation function is the step function on page 32-2682.
- In two-class learning, if one of the two classes has a relative frequency of 0, then the transformation function is the constant function on page 32-2683. `fitSVMPosterior` is not appropriate for one-class learning.

- If `SVMMModel` is a `ClassificationSVM` classifier, then the software estimates the optimal transformation function by 10-fold cross validation as outlined in [1]. Otherwise, `SVMMModel` must be a `ClassificationPartitionedModel` classifier. `SVMMModel` specifies the cross-validation method.
- The software stores the optimal transformation function in `ScoreSVMMModel.ScoreTransform`.

`ScoreSVMMModel = fitSVMPosterior(SVMMModel, TBL, ResponseVarName)` returns a trained support vector classifier containing the transformation function from the trained, compact SVM classifier `SVMMModel`. The software estimates the score transformation function using predictor data in the table `TBL` and class labels `TBL.ResponseVarName`.

`ScoreSVMMModel = fitSVMPosterior(SVMMModel, TBL, Y)` returns a trained support vector classifier containing the transformation function from the trained, compact SVM classifier `SVMMModel`. The software estimates the score transformation function using predictor data in the table `TBL` and class labels `Y`.

`ScoreSVMMModel = fitSVMPosterior(SVMMModel, X, Y)` returns a trained support vector classifier containing the transformation function from the trained, compact SVM classifier `SVMMModel`. The software estimates the score transformation function using predictor data `X` and class labels `Y`.

`ScoreSVMMModel = fitSVMPosterior(____, Name, Value)` uses additional options specified by one or more `Name, Value` pair arguments provided `SVMMModel` is a `ClassificationSVM` classifier. For example, you can specify the number of folds to use in  $k$ -fold cross validation.

`[ScoreSVMMModel, ScoreTransform] = fitSVMPosterior(____)` additionally returns the transformation function parameters (`ScoreTransform`) using any of the input arguments in the previous syntaxes.

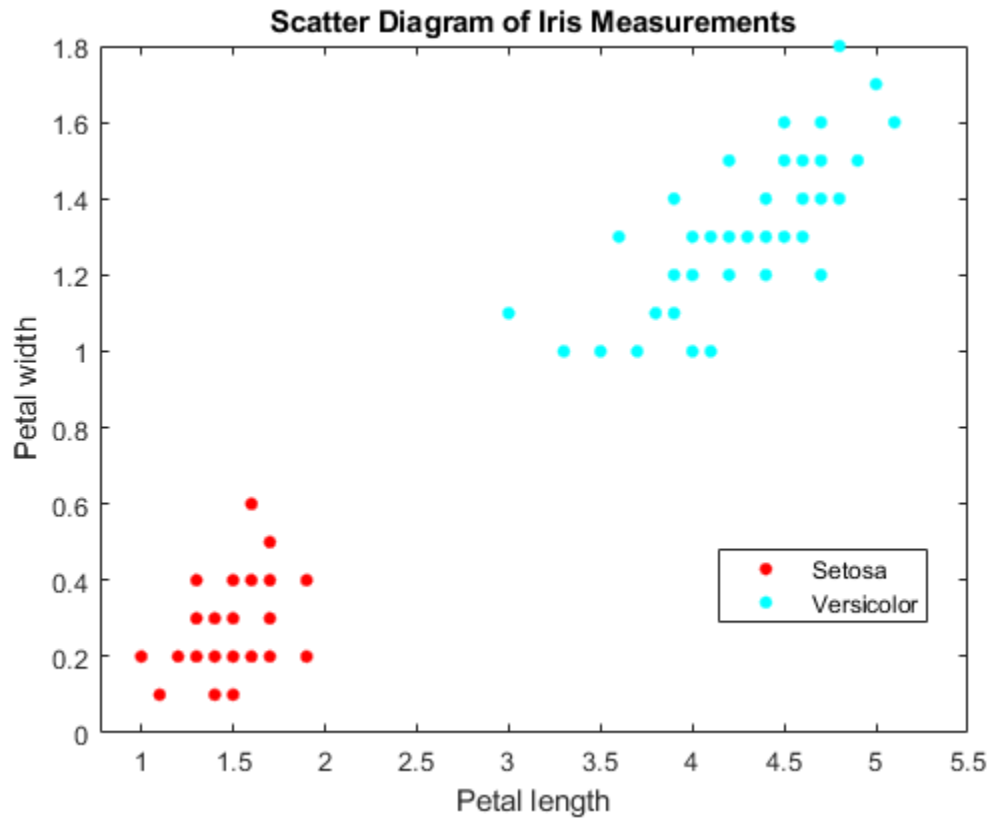
## Examples

### Fit the Score-to-Posterior Probability Function for Separable Classes

Load Fisher's iris data set. Train the classifier using the petal lengths and widths, and remove the virginica species from the data.

```
load fisheriris
classKeep = ~strcmp(species, 'virginica');
X = meas(classKeep, 3:4);
y = species(classKeep);

gscatter(X(:,1), X(:,2), y);
title('Scatter Diagram of Iris Measurements')
xlabel('Petal length')
ylabel('Petal width')
legend('Setosa', 'Versicolor')
```



The classes are perfectly separable. Therefore, the score transformation function is a step function.

Train an SVM classifier using the data. Cross validate the classifier using 10-fold cross validation (the default).

```
rng(1);  
CVSVMModel = fitcsvm(X,y,'CrossVal','on');
```

CVSVMModel is a trained `ClassificationPartitionedModel` SVM classifier.

Estimate the step function that transforms scores to posterior probabilities.

```
[ScoreCVSVMModel,ScoreParameters] = fitSVMPosterior(CVSVMModel);
```

```
Warning: Classes are perfectly separated. The optimal score-to-posterior  
transformation is a step function.
```

`fitSVMPosterior` does the following:

- Uses the data that the software stored in `CVSVMModel` to fit the transformation function
- Warns whenever the classes are separable
- Stores the step function in `ScoreCVSVMModel.ScoreTransform`

Display the score function type and its parameter values.

```
ScoreParameters
```

```
ScoreParameters =
```

```
struct with fields:
```

```
                Type: 'step'  
        LowerBound: -0.8431  
        UpperBound: 0.6897  
PositiveClassProbability: 0.5000
```

`ScoreParameters` is a structure array with four fields:

- The score transformation function type (`Type`)
- The score corresponding to the negative class boundary (`LowerBound`)
- The score corresponding to the positive class boundary (`UpperBound`)



- The positive class probability (`PositiveClassProbability`)

Since the classes are separable, the step function transforms the score to either 0 or 1, which is the posterior probability that an observation is a versicolor iris.

### Fit the Score-to-Posterior Probability Function for Inseparable Classes

Load the ionosphere data set.

```
load ionosphere
```

The classes of this data set are not separable.

Train an SVM classifier. Cross validate using 10-fold cross validation (the default). It is good practice to standardize the predictors and specify the class order.

```
rng(1) % For reproducibility
CVSVMModel = fitcsvm(X,Y,'ClassNames',{'b','g'},'Standardize',true,...
    'CrossVal','on');
ScoreTransform = CVSVMModel.ScoreTransform
```

```
ScoreTransform =
    'none'
```

`CVSVMModel` is a trained `ClassificationPartitionedModel` SVM classifier. The positive class is 'g'. The `ScoreTransform` property is none.

Estimate the optimal score function for mapping observation scores to posterior probabilities of an observation being classified as 'g'.

```
[ScoreCVSVMModel,ScoreParameters] = fitSVMPosterior(CVSVMModel);
ScoreTransform = ScoreCVSVMModel.ScoreTransform
ScoreParameters
```

```
ScoreTransform =
    '@(S) sigmoid(S,-9.481576e-01,-1.218300e-01)'
```

```
ScoreParameters =  
  
    struct with fields:  
  
        Type: 'sigmoid'  
        Slope: -0.9482  
        Intercept: -0.1218
```

ScoreTransform is the optimal score transform function. ScoreParameters contains the score transformation function, slope estimate, and the intercept estimate.

You can estimate test-sample, posterior probabilities by passing ScoreCVSVMModel to kfoldPredict.

### Estimate Posterior Probabilities for Test Samples

Estimate positive class posterior probabilities for the test set of an SVM algorithm.

Load the ionosphere data set.

```
load ionosphere
```

Train an SVM classifier. Specify a 20% holdout sample. It is good practice to standardize the predictors and specify the class order.

```
rng(1) % For reproducibility  
CVSVMModel = fitcsvm(X,Y,'Holdout',0.2,'Standardize',true,...  
    'ClassNames',{'b','g'});
```

CVSVMModel is a trained ClassificationPartitionedModel cross-validated classifier.

Estimate the optimal score function for mapping observation scores to posterior probabilities of an observation being classified as 'g'.

```
ScoreCVSVMModel = fitSVMPosterior(CVSVMModel);
```

ScoreSVMModel is a trained ClassificationPartitionedModel cross-validated classifier containing the optimal score transformation function estimated from the training data.

Estimate the out-of-sample positive class posterior probabilities. Display the results for the first 10 out-of-sample observations.

```
[~,OOSPostProbs] = kfoldPredict(ScoreCVSVMModel);
indx = ~isnan(OOSPostProbs(:,2));
hoObs = find(indx); % Holdout observation numbers
OOSPostProbs = [hoObs, OOSPostProbs(indx,2)];
table(OOSPostProbs(1:10,1),OOSPostProbs(1:10,2),...
      'VariableNames',{'ObservationIndex','PosteriorProbability'})
```

```
ans =
```

```
10x2 table
```

ObservationIndex	PosteriorProbability
6	0.17378
7	0.89637
8	0.0076609
9	0.91602
16	0.026718
22	4.6081e-06
23	0.9024
24	2.4129e-06
38	0.00042697
41	0.86427

## Input Arguments

### **svmmode1** — Trained SVM classifier

ClassificationSVM classifier | CompactClassificationSVM classifier | ClassificationPartitionedModel classifier

Trained SVM classifier, specified as a ClassificationSVM, CompactClassificationSVM, or ClassificationPartitionedModel classifier.

If svmmode1 is a ClassificationSVM classifier, then you can set optional name-value pair arguments.

If `SVMMODEL` is a `CompactClassificationSVM` classifier, then you must input predictor data `X` and class labels `Y`.

**TBL — Sample data**

table

Sample data, specified as a table. Each row of `TBL` corresponds to one observation, and each column corresponds to one predictor variable. Optionally, `TBL` can contain additional columns for the response variable and observation weights. `TBL` must contain all of the predictors used to train `SVMMODEL`. Multi-column variables and cell arrays other than cell arrays of character vectors are not allowed.

If `TBL` contains the response variable used to train `SVMMODEL`, then you do not need to specify `ResponseVarName` or `Y`.

If you trained `SVMMODEL` using sample data contained in a table, then the input data for this method must also be in a table.

Data Types: `table`

**x — Predictor data**

matrix

Predictor data used to estimate the score-to-posterior-probability transformation function, specified as a matrix.

Each row of `X` corresponds to one observation (also known as an instance or example), and each column corresponds to one variable (also known as a feature).

The length of `Y` and the number of rows of `X` must be equal.

If you set `'Standardize', true` in `fitcsvm` to train `SVMMODEL`, then the software standardizes the columns of `X` using the corresponding means in `SVMMODEL.Mu` and standard deviations in `SVMMODEL.Sigma`. If the software fits the transformation-function parameter estimates using standardized data, then the estimates might differ from estimation without standardized data.

Data Types: `double` | `single`

**ResponseVarName — Response variable name**

name of a variable in `TBL`

Response variable name, specified as the name of a variable in `TBL`. If `TBL` contains the response variable used to train `SVMModel`, then you do not need to specify `ResponseVarName`.

If you specify `ResponseVarName`, then you must do so as a character vector. For example, if the response variable is stored as `TBL.Response`, then specify it as `'Response'`. Otherwise, the software treats all columns of `TBL`, including `TBL.Response`, as predictors.

The response variable must be a categorical or character array, logical or numeric vector, or cell array of character vectors. If the response variable is a character array, then each element must correspond to one row of the array.

### **Y — Class labels**

categorical array | character array | logical vector | vector of numeric values | cell array of character vectors

Class labels used to estimate the score-to-posterior-probability transformation function, specified as a categorical or character array, logical or numeric vector, or cell array of character vectors.

If `Y` is a character array, then each element must correspond to one class label.

The length of `Y` and the number of rows of `X` must be equal.

## **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'kFold', 8` performs 8-fold cross validation when `SVMModel` is a `ClassificationSVM` classifier.

### **CVPartition — Cross-validation partition**

[] (default) | `cvpartition` partition

Cross-validation partition used to compute the transformation function, specified as the comma-separated pair consisting of `'CVPartition'` and a `cvpartition` partition as

created by `cvpartition`. You can use only one of these four options at a time for creating a cross-validated model: `'KFold'`, `'Holdout'`, `'Leaveout'`, or `'CVPARTITION'`.

`crossval` splits the data into subsets using `cvpartition`.

#### **Holdout — Fraction of data for holdout validation**

scalar value in the range (0,1)

Fraction of data for holdout validation used to compute the transformation function, specified as the comma-separated pair consisting of `'Holdout'` and a scalar value in the range (0,1). Holdout validation tests the specified fraction of the data, and uses the remaining data for training.

You can use only one of these four options at a time for creating a cross-validated model: `'KFold'`, `'Holdout'`, `'Leaveout'`, or `'CVPARTITION'`.

Example: `'Holdout',0.1`

Data Types: `double` | `single`

#### **KFold — Number of folds**

10 (default) | positive integer value greater than 1

Number of folds to use when computing the transformation function, specified as the comma-separated pair consisting of `'KFold'` and a positive integer value greater than 1.

You can use only one of these four options at a time for creating a cross-validated model: `'KFold'`, `'Holdout'`, `'Leaveout'`, or `'CVPARTITION'`.

Example: `'KFold',8`

Data Types: `single` | `double`

#### **Leaveout — Leave-one-out cross-validation flag**

`'off'` (default) | `'on'`

Leave-one-out cross-validation flag indicating whether to use leave-one-out cross validation to compute the transformation function, specified as the comma-separated pair consisting of `'Leaveout'` and `'on'` or `'off'`. Use leave-one-out cross validation by using `'on'`.

You can use only one of these four options at a time for creating a cross-validated model: 'KFold', 'Holdout', 'Leaveout', or 'CVPartition'.

Example: 'Leaveout', 'on'

## Output Arguments

### **ScoreSVMModel** — Trained SVM classifier

ClassificationSVM classifier | CompactClassificationSVM classifier | ClassificationPartitionedModel classifier

Trained SVM classifier containing the estimated score transformation function, returned as a ClassificationSVM, CompactClassificationSVM, or ClassificationPartitionedModel classifier.

The ScoreSVMModel classifier type is the same as the SVMModel classifier type.

To estimate posterior probabilities, pass ScoreSVMModel and predictor data to predict. If you set 'Standardize', true in fitcsvm to train SVMModel, then predict standardizes the columns of X using the corresponding means in SVMModel.Mu and standard deviations in SVMModel.Sigma.

### **ScoreTransform** — Optimal score-to-posterior-probability transformation function parameters

structure array

Optimal score-to-posterior-probability transformation function parameters, specified as a structure array. If field Type is:

- sigmoid, then ScoreTransform has these fields:
  - Slope — The value of  $A$  in the sigmoid function on page 32-2682
  - Intercept — The value of  $B$  in the sigmoid function
- step, then ScoreTransform has these fields:
  - PositiveClassProbability: the value of  $\pi$  in the step function on page 32-2682.  $\pi$  represents:
    - The probability that an observation is in the positive class.

- The posterior probability that a score is in the interval  $(\text{LowerBound}, \text{UpperBound})$ .
- LowerBound: the value  $\max_n s_n$  in the step function. It represents the lower bound of the interval that assigns the posterior probability of being in the positive class `PositiveClassProbability` to scores. Any observation with a score less than LowerBound has posterior probability of being the positive class 0.
- UpperBound: the value  $\min_n s_n$  in the step function. It represents the upper bound of the interval that assigns the posterior probability of being in the positive class `PositiveClassProbability`. Any observation with a score greater than UpperBound has posterior probability of being the positive class 1.
- constant, then `ScoreTransform.PredictedClass` contains the name of the class prediction.

This result is the same as `SVMModel.ClassNames`. The posterior probability of an observation being in `ScoreTransform.PredictedClass` is always 1.

## Definitions

### Sigmoid Function

The sigmoid function that maps score  $s_j$  corresponding to observation  $j$  to the positive class posterior probability is

$$P(s_j) = \frac{1}{1 + \exp(As_j + B)}.$$

If the output argument `ScoreTransform.Type` is `sigmoid`, then parameters  $A$  and  $B$  correspond to the fields `Scale` and `Intercept` of `ScoreTransform`, respectively.

### Step Function

The step function that maps score  $s_j$  corresponding to observation  $j$  to the positive class posterior probability is



$$P(s_j) = \begin{cases} 0; & s_j < \max_{y_k=-1} s_k \\ \pi; & \max_{y_k=-1} s_k \leq s_j \leq \min_{y_k=+1} s_k, \\ 1; & s_j > \min_{y_k=+1} s_k \end{cases}$$

where:

- $s_j$  the score of observation  $j$ .
- $+1$  and  $-1$  denote the positive and negative classes, respectively.
- $\pi$  is the prior probability that an observation is in the positive class.

If the output argument `ScoreTransform.Type` is `step`, then the quantities  $\max_{y_k=-1} s_k$

and  $\min_{y_k=+1} s_k$  correspond to the fields `LowerBound` and `UpperBound` of `ScoreTransform`, respectively.

## Constant Function

The constant function maps all scores in a sample to posterior probabilities 1 or 0.

If all observations have posterior probability 1, then they are expected to come from the positive class.

If all observations have posterior probability 0, then they are not expected to come from the positive class.

## Tips

- Here is one way to predict positive class posterior probabilities.
  - 1 Train an SVM classifier by passing the data to `fitcsvm`. The result is a trained SVM classifier, such as, `SVModel`, that stores the data. The software sets the score transformation function property (`SVModel.ScoreTransformation`) to `none`.

- 2 Pass the trained SVM classifier `SVModel` to `fitSVMPosterior` or `fitPosterior`. The result, for example, `ScoreSVModel`, is the same, trained SVM classifier as `SVModel`, except the software sets `ScoreSVModel.ScoreTransformation` to the optimal score transformation function.

If you skip step 2, then `predict` returns the positive class score rather than the positive class posterior probability.

- 3 Pass the trained SVM classifier containing the optimal score transformation function (`ScoreSVModel`) and predictor data matrix to `predict`. The second column of the second output argument stores the positive class posterior probabilities corresponding to each row of the predictor data matrix.
- After fitting posterior probabilities, use the function `predict` and the trained model object (`ClassificationSVM` or `CompactClassificationSVM` classifier) to generate C code that predicts labels for new data. For details, see “Code Generation” on page 32-0 .

## Algorithms

If you reestimate the score-to-posterior-probability transformation function, that is, if you pass an SVM classifier to `fitPosterior` or `fitSVMPosterior` and its `ScoreTransform` property is not none, then the software:

- Displays a warning
- Resets the original transformation function to 'none' before estimating the new one

## References

- [1] Platt, J. “Probabilistic outputs for support vector machines and comparisons to regularized likelihood methods”. In: *Advances in Large Margin Classifiers*. Cambridge, MA: The MIT Press, 2000, pp. 61–74.

## See Also

`ClassificationPartitionedModel` | `ClassificationSVM` |  
`CompactClassificationSVM` | `fitPosterior` | `fitPosterior` | `fitcsvm` |  
`kfoldPredict` | `predict`

Introduced in R2014a

## fitted

**Class:** `GeneralizedLinearMixedModel`

Fitted responses from generalized linear mixed-effects model

## Syntax

```
mufit = fitted(glme)
mufit = fitted(glme, Name, Value)
```

## Description

`mufit = fitted(glme)` returns the fitted conditional response of the generalized linear mixed-effects model `glme`.

`mufit = fitted(glme, Name, Value)` returns the fitted response with additional options specified by one or more name-value pair arguments. For example, you can specify to compute the marginal fitted response.

## Input Arguments

**glme** — Generalized linear mixed-effects model

`GeneralizedLinearMixedModel` object

Generalized linear mixed-effects model, specified as a `GeneralizedLinearMixedModel` object. For properties and methods of this object, see `GeneralizedLinearMixedModel`.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

**Conditional** — Indicator for conditional response`true` (default) | `false`

Indicator for conditional response, specified as the comma-separated pair consisting of 'Conditional' and one of the following.

Value	Description
<code>true</code>	Contributions from both fixed effects and random effects (conditional)
<code>false</code>	Contribution from only fixed effects (marginal)

To obtain fitted marginal response values, `fitted` computes the conditional mean of the response with the empirical Bayes predictor vector of random effects  $b$  set equal to 0. For more information, see “Conditional and Marginal Response” on page 32-2690

Example: 'Conditional', `false`

## Output Arguments

**`mfit`** — Fitted response values $n$ -by-1 vector

Fitted response values, returned as an  $n$ -by-1 vector, where  $n$  is the number of observations.

## Examples

**Plot Observed Versus Fitted Values**

Load the sample data.

```
load mfr
```

This simulated data is from a manufacturing company that operates 50 factories across the world, with each factory running a batch process to create a finished product. The company wants to decrease the number of defects in each batch, so it developed a new manufacturing process. To test the effectiveness of the new process, the company selected

20 of its factories at random to participate in an experiment: Ten factories implemented the new process, while the other ten continued to run the old process. In each of the 20 factories, the company ran five batches (for a total of 100 batches) and recorded the following data:

- Flag to indicate whether the batch used the new process (`newprocess`)
- Processing time for each batch, in hours (`time`)
- Temperature of the batch, in degrees Celsius (`temp`)
- Categorical variable indicating the supplier (A, B, or C) of the chemical used in the batch (`supplier`)
- Number of defects in the batch (`defects`)

The data also includes `time_dev` and `temp_dev`, which represent the absolute deviation of time and temperature, respectively, from the process standard of 3 hours at 20 degrees Celsius.

Fit a generalized linear mixed-effects model using `newprocess`, `time_dev`, `temp_dev`, and `supplier` as fixed-effects predictors. Include a random-effects term for intercept grouped by `factory`, to account for quality differences that might exist due to factory-specific variations. The response variable `defects` has a Poisson distribution, and the appropriate link function for this model is log. Use the Laplace fit method to estimate the coefficients. Specify the dummy variable encoding as `'effects'`, so the dummy variable coefficients sum to 0.

The number of defects can be modeled using a Poisson distribution  
 $\text{defects}_{ij} \sim \text{Poisson}(\mu_{ij})$

This corresponds to the generalized linear mixed-effects model

$$\log(\mu_{ij}) = \beta_0 + \beta_1 \text{newprocess}_{ij} + \beta_2 \text{time\_dev}_{ij} + \beta_3 \text{temp\_dev}_{ij} + \beta_4 \text{supplier\_C}_{ij} + \beta_5 \text{supplier\_B}_{ij} + b_i,$$

where

- $\text{defects}_{ij}$  is the number of defects observed in the batch produced by factory  $i$  during batch  $j$ .
- $\mu_{ij}$  is the mean number of defects corresponding to factory  $i$  (where  $i = 1, 2, \dots, 20$ ) during batch  $j$  (where  $j = 1, 2, \dots, 5$ ).

- `newprocessij`, `time_devij`, and `temp_devij` are the measurements for each variable that correspond to factory *i* during batch *j*. For example, `newprocessij` indicates whether the batch produced by factory *i* during batch *j* used the new process.
- `supplier_Cij` and `supplier_Bij` are dummy variables that use effects (sum-to-zero) coding to indicate whether company C or B, respectively, supplied the process chemicals for the batch produced by factory *i* during batch *j*.
- $b_i \sim N(0, \sigma_b^2)$  is a random-effects intercept for each factory *i* that accounts for factory-specific variation in quality.

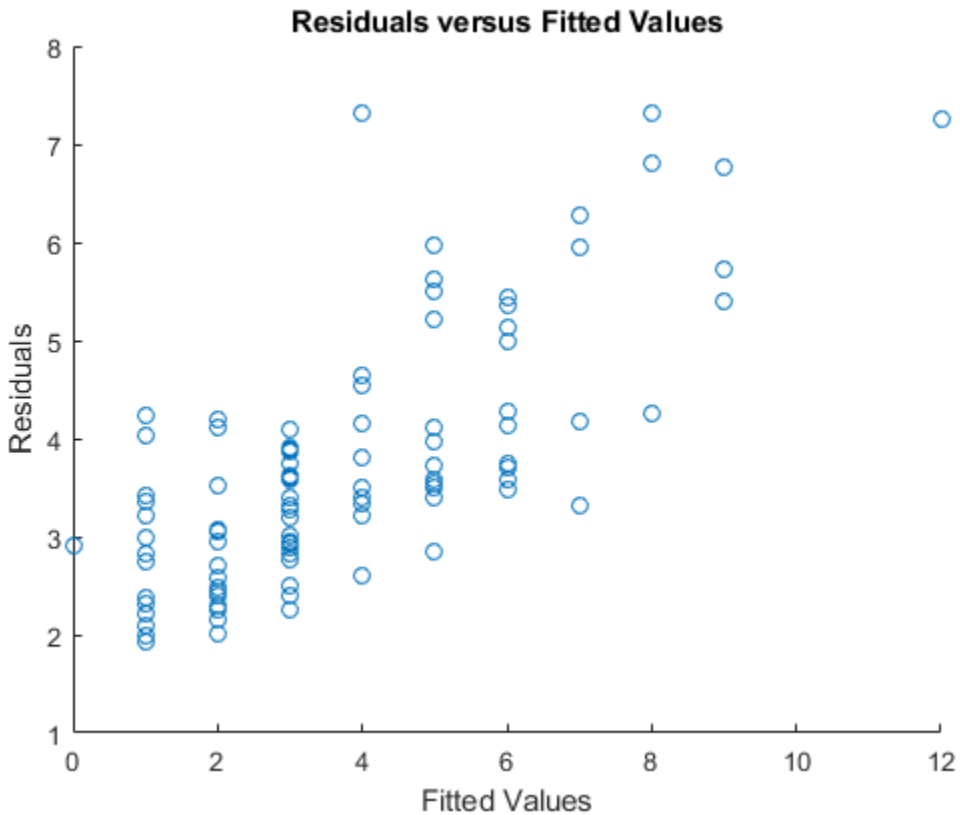
```
glme = fitglme(mfr, 'defects ~ 1 + newprocess + time_dev + temp_dev + supplier + (1|factory)
  Distribution', 'Poisson', 'Link', 'log', 'FitMethod', 'Laplace', 'DummyVarCoding', 'effects')
```

Generate the fitted conditional mean values for the model.

```
mufit = fitted(glme);
```

Create a scatterplot of the observed values versus fitted values.

```
figure
scatter(mfr.defects, mufit)
title('Residuals versus Fitted Values')
xlabel('Fitted Values')
ylabel('Residuals')
```



## Definitions

### Conditional and Marginal Response

A *conditional response* includes contributions from both fixed- and random-effects predictors. A *marginal response* includes contribution from only fixed effects.

Suppose the generalized linear mixed-effects model `glme` has an  $n$ -by- $p$  fixed-effects design matrix  $X$  and an  $n$ -by- $q$  random-effects design matrix  $Z$ . Also, suppose the



estimated  $p$ -by-1 fixed-effects vector is  $\hat{\beta}$ , and the  $q$ -by-1 empirical Bayes predictor vector of random effects is  $\hat{b}$ .

The fitted conditional response corresponds to the 'Conditional', true name-value pair argument, and is defined as

$$\hat{\mu}_{cond} = g^{-1}(\hat{\eta}_{ME}),$$

where  $\hat{\eta}_{ME}$  is the linear predictor including the fixed- and random-effects of the generalized linear mixed-effects model

$$\hat{\eta}_{ME} = X\hat{\beta} + Z\hat{b} + \delta.$$

The fitted marginal response corresponds to the 'Conditional', false name-value pair argument, and is defined as

$$\hat{\mu}_{mar} = g^{-1}(\hat{\eta}_{FE}),$$

where  $\hat{\eta}_{FE}$  is the linear predictor including only the fixed-effects portion of the generalized linear mixed-effects model

$$\hat{\eta}_{FE} = X\hat{\beta} + \delta.$$

## See Also

GeneralizedLinearMixedModel | designMatrix | fitglme | residuals | response

## fitted

**Class:** LinearMixedModel

Fitted responses from a linear mixed-effects model

## Syntax

```
yfit = fitted(lme)
yfit = fitted(lme, Name, Value)
```

## Description

`yfit = fitted(lme)` returns the fitted conditional response on page 32-2699 from the linear mixed-effects model `lme`.

`yfit = fitted(lme, Name, Value)` returns the fitted response from the linear mixed-effects model `lme` with additional options specified by one or more `Name, Value` pair arguments.

For example, you can specify if you want to compute the fitted marginal response on page 32-2699.

## Input Arguments

**lme** — Linear mixed-effects model

LinearMixedModel object

Linear mixed-effects model, specified as a LinearMixedModel object constructed using `fitlme` or `fitlmematrix`.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single

quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### **Conditional** — Indicator for conditional response

True (default) | False

Indicator for conditional response, specified as the comma-separated pair consisting of 'Conditional' and either of the following.

True	Contribution from both fixed effects and random effects (conditional)
False	Contribution from only fixed effects (marginal)

Example: `'Conditional, 'False'`

## Output Arguments

### **yfit** — Fitted response values

*n*-by-1 vector

Fitted response values, returned as an *n*-by-1 vector, where *n* is the number of observations.

## Examples

### **Compute Fitted Conditional and Marginal Responses**

Load the sample data.

```
load flu
```

The `flu` dataset array has a `Date` variable, and 10 variables containing estimated influenza rates (in 9 different regions, estimated from Google® searches, plus a nationwide estimate from the Center for Disease Control and Prevention, CDC).

To fit a linear-mixed effects model, your data must be in a properly formatted dataset array. To fit a linear mixed-effects model with the influenza rates as the responses and

region as the predictor variable, combine the nine columns corresponding to the regions into an array. The new dataset array, `flu2`, must have the response variable, `FluRate`, the nominal variable, `Region`, that shows which region each estimate is from, and the grouping variable `Date`.

```
flu2 = stack(flu, 2:10, 'NewDataVarName', 'FluRate', 'IndVarName', 'Region');
flu2.Date = nominal(flu2.Date);
```

Fit a linear mixed-effects model with fixed effects for region and a random intercept that varies by `Date`.

`Region` is a categorical variable. You can specify the contrasts for categorical variables using the `DummyVarCoding` name-value pair argument when fitting the model. When you do not specify the contrasts, `fitlme` uses the 'reference' contrast by default. Because the model has an intercept, `fitlme` takes the first region, NE, as the reference and creates eight dummy variables representing the other eight regions. For example, `I[MidAtl]` is the dummy variable representing the region `MidAtl`. For details, see “Dummy Indicator Variables” on page 2-62.

The corresponding model is

$$y_{im} = \beta_0 + \beta_1 I[\text{MidAtl}]_i + \beta_2 I[\text{ENCentral}]_i + \beta_3 I[\text{WNCentral}]_i + \beta_4 I[\text{SATl}]_i + \beta_5 I[\text{ESCentral}]_i + \beta_6 I[\text{WSCentral}]_i + \beta_7 I[\text{Mtn}]_i + \beta_8 I[\text{Pac}]_i + b_{0m} + \varepsilon_{im}, \quad m = 1, 2, \dots, 52,$$

where  $y_{im}$  is the observation  $i$  for level  $m$  of grouping variable `Date`,  $\beta_j$ ,  $j = 0, 1, \dots, 8$ , are the fixed-effects coefficients, with  $\beta_0$  being the coefficient for region NE.  $b_{0m}$  is the random effect for level  $m$  of the grouping variable `Date`, and  $\varepsilon_{im}$  is the observation error for observation  $i$ . The random effect has the prior distribution,  $b_{0m} \sim N(0, \sigma_b^2)$  and the error term has the distribution,  $\varepsilon_{im} \sim N(0, \sigma^2)$ .

```
lme = fitlme(flu2, 'FluRate ~ 1 + Region + (1|Date)')
```

```
lme =
```

```
Linear mixed-effects model fit by ML
```

```
Model information:
```

Number of observations	468
Fixed effects coefficients	9

Random effects coefficients           52  
Covariance parameters                2

Formula:

FluRate ~ 1 + Region + (1 | Date)

Model fit statistics:

AIC	BIC	LogLikelihood	Deviance
318.71	364.35	-148.36	296.71

Fixed effects coefficients (95% CIs):

Name	Estimate	SE	tStat	DF
'(Intercept)'	1.2233	0.096678	12.654	459
'Region_MidAtl'	0.010192	0.052221	0.19518	459
'Region_ENCentral'	0.051923	0.052221	0.9943	459
'Region_WNCentral'	0.23687	0.052221	4.5359	459
'Region_SAtl'	0.075481	0.052221	1.4454	459
'Region_ESCentral'	0.33917	0.052221	6.495	459
'Region_WSCentral'	0.069	0.052221	1.3213	459
'Region_Mtn'	0.046673	0.052221	0.89377	459
'Region_Pac'	-0.16013	0.052221	-3.0665	459

pValue	Lower	Upper
1.085e-31	1.0334	1.4133
0.84534	-0.092429	0.11281
0.3206	-0.050698	0.15454
7.3324e-06	0.13424	0.33949
0.14902	-0.02714	0.1781
2.1623e-10	0.23655	0.44179
0.18705	-0.033621	0.17162
0.37191	-0.055948	0.14929
0.0022936	-0.26276	-0.057514

Random effects covariance parameters (95% CIs):

Group: Date (52 Levels)

Name1	Name2	Type	Estimate	Lower
'(Intercept)'	'(Intercept)'	'std'	0.6443	0.5297

Upper  
0.78368

Group: Error

Name	Estimate	Lower	Upper
'Res Std'	0.26627	0.24878	0.285

The  $P$ -values 7.3324e-06 and 2.1623e-10 respectively show that the fixed effects of the flu rates in regions `WNCentral` and `ESCentral` are significantly different relative to the flu rates in region `NE`.

The confidence limits for the standard deviation of the random-effects term,  $\sigma_b$ , do not include 0 (0.5297, 0.78368), which indicates that the random-effects term is significant. You can also test the significance of the random-effects terms using the `compare` method.

The conditional fitted response from the model at a given observation includes contributions from fixed and random effects. For example, the estimated best linear unbiased predictor (BLUP) of the flu rate for region `WNCentral` in week 10/9/2005 is

$$\begin{aligned}\hat{y}_{\text{WNCentral},10/9/2005} &= \hat{\beta}_0 + \hat{\beta}_3 I[\text{WNCentral}] + \hat{b}_{10/9/2005} \\ &= 1.2233 + 0.23687 - 0.1718 \\ &= 1.28837.\end{aligned}$$

This is the fitted conditional response, since it includes contributions to the estimate from both the fixed and random effects. You can compute this value as follows.

```
beta = fixedEffects(lme);
[~,~,STATS] = randomEffects(lme); % Compute the random-effects statistics (STATS)
STATS.Level = nominal(STATS.Level);
y_hat = beta(1) + beta(4) + STATS.Estimate(STATS.Level=='10/9/2005')

y_hat =

    1.2884
```

In the previous calculation, `beta(1)` corresponds to the estimate for  $\beta_0$  and `beta(4)` corresponds to the estimate for  $\beta_3$ . You can simply display the fitted value using the `fitted` method.

```
F = fitted(lme);
F(flu2.Date == '10/9/2005' & flu2.Region == 'WNCentral')

ans =
```

1.2884

The estimated marginal response for region `WNCentral` in week 10/9/2005 is

$$\begin{aligned}\hat{y}_{\text{WNCentral},10/9/2005}^{(\text{marginal})} &= \hat{\beta}_0 + \hat{\beta}_3 I[\text{WNCentral}] \\ &= 1.2233 + 0.23687 \\ &= 1.46017.\end{aligned}$$

Compute the fitted marginal response.

```
F = fitted(lme, 'Conditional', false);
F(flu2.Date == '10/9/2005' & flu2.Region == 'WNCentral')
```

ans =

1.4602

## Plot Residuals vs. Fitted Values

Load the sample data.

```
load(fullfile(matlabroot, 'examples', 'stats', 'weight.mat'));
```

`weight` contains data from a longitudinal study, where 20 subjects are randomly assigned to 4 exercise programs, and their weight loss is recorded over six 2-week time periods. This is simulated data.

Store the data in a table. Define `Subject` and `Program` as categorical variables.

```
tbl = table(InitialWeight, Program, Subject, Week, y);
tbl.Subject = nominal(tbl.Subject);
tbl.Program = nominal(tbl.Program);
```

Fit a linear mixed-effects model where the initial weight, type of program, week, and the interaction between the week and type of program are the fixed effects. The intercept and week vary by subject.

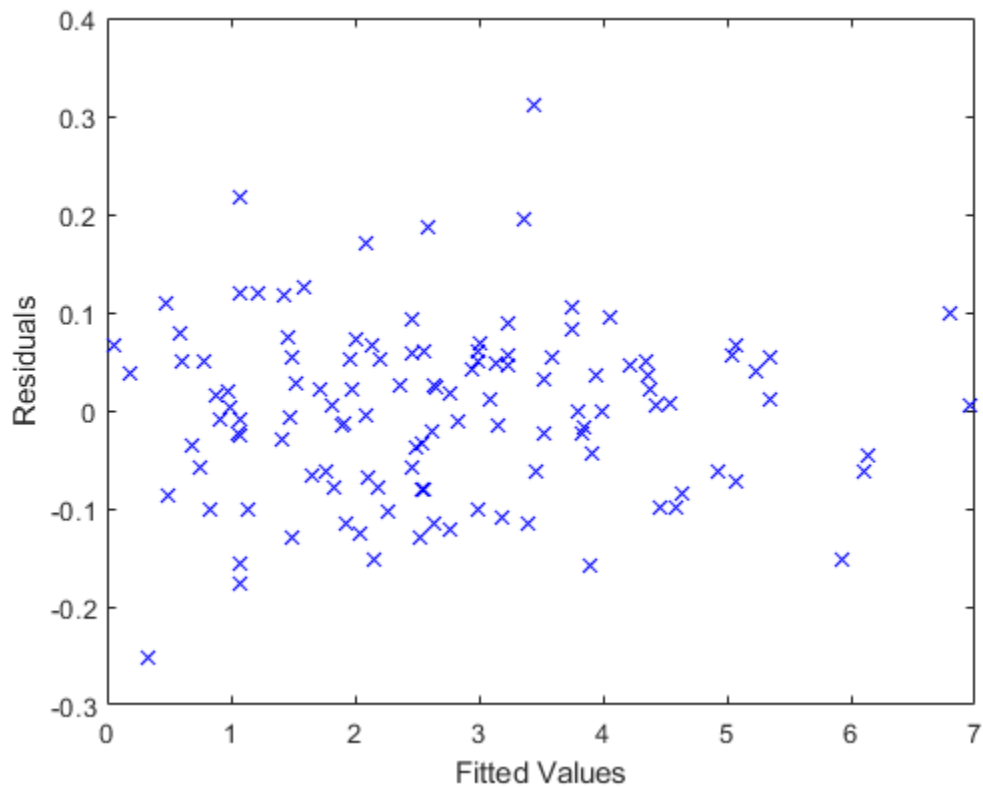
```
lme = fitlme(tbl, 'y ~ InitialWeight + Program*Week + (Week|Subject)');
```

Compute the fitted values and raw residuals.

```
F = fitted(lme);  
R = residuals(lme);
```

Plot the residuals versus the fitted values.

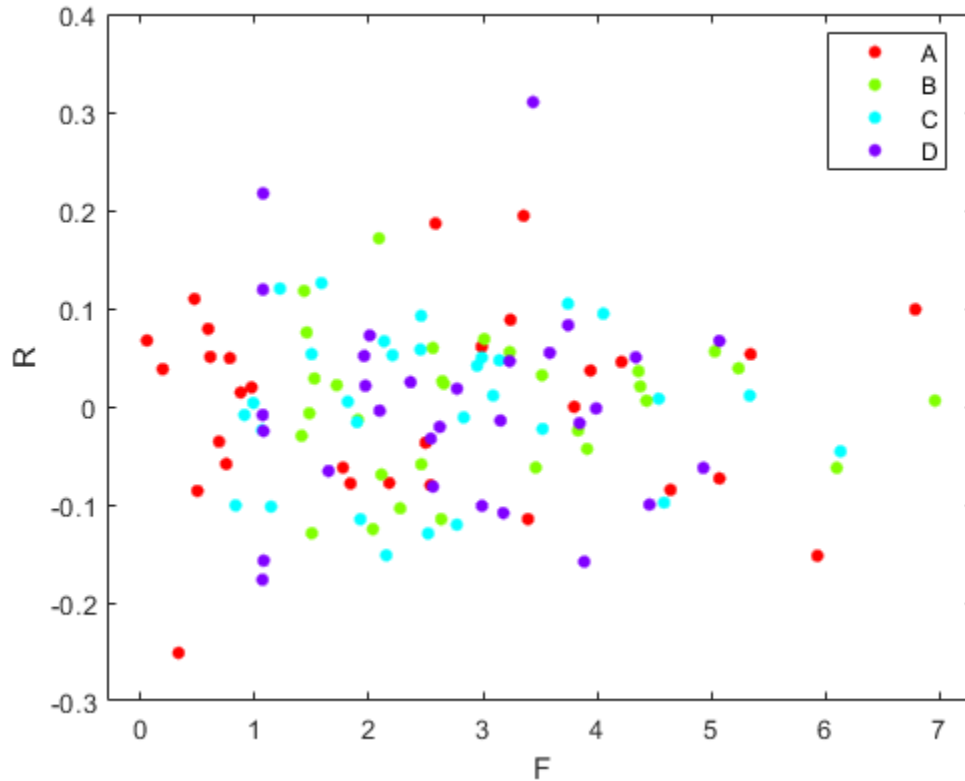
```
plot(F,R,'bx')  
xlabel('Fitted Values')  
ylabel('Residuals')
```



Now, plot the residuals versus the fitted values, grouped by program.

```
figure()  
gscatter(F,R,Program)
```





## Definitions

### Fitted Conditional and Marginal Response

A conditional response includes contributions from both fixed and random effects, whereas a marginal response includes contribution from only fixed effects.

Suppose the linear mixed-effects model,  $lme$ , has an  $n$ -by- $p$  fixed-effects design matrix  $X$  and an  $n$ -by- $q$  random-effects design matrix  $Z$ . Also, suppose the  $p$ -by-1 estimated fixed-

effects vector is  $\hat{\beta}$ , and the  $q$ -by-1 estimated best linear unbiased predictor (BLUP) vector of random effects is  $\hat{b}$ . The fitted conditional response is

$$\hat{y}_{Cond} = X\hat{\beta} + Z\hat{b},$$

and the fitted marginal response is

$$\hat{y}_{Mar} = X\hat{\beta},$$

## See Also

`LinearMixedModel` | `residuals` | `response`

# fixedEffects

**Class:** GeneralizedLinearMixedModel

Estimates of fixed effects and related statistics

## Syntax

```
beta = fixedEffects(glme)
[beta,betaname] = fixedEffects(glme)
[beta,betaname,stats] = fixedEffects(glme)
[ ___ ] = fixedEffects(glme,Name,Value)
```

## Description

`beta = fixedEffects(glme)` returns the estimated fixed-effects coefficients, `beta`, of the generalized linear mixed-effects model `glme`.

`[beta,betaname] = fixedEffects(glme)` also returns the names of estimated fixed-effects coefficients in `betaname`. Each name corresponds to a fixed-effects coefficient in `beta`.

`[beta,betaname,stats] = fixedEffects(glme)` also returns a table of statistics, `stats`, related to the estimated fixed-effects coefficients of `glme`.

`[ ___ ] = fixedEffects(glme,Name,Value)` returns any of the output arguments in previous syntaxes using additional options specified by one or more `Name,Value` pair arguments. For example, you can specify the confidence level, or the method for computing the approximate degrees of freedom for the *t*-statistic.

## Input Arguments

**glme** — Generalized linear mixed-effects model

GeneralizedLinearMixedModel object

Generalized linear mixed-effects model, specified as a `GeneralizedLinearMixedModel` object. For properties and methods of this object, see `GeneralizedLinearMixedModel`.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### **Alpha** — Confidence level

0.05 (default) | scalar value in the range [0,1]

Confidence level, specified as the comma-separated pair consisting of 'Alpha' and a scalar value in the range [0,1]. For a value  $\alpha$ , the confidence level is  $100 \times (1 - \alpha)\%$ .

For example, for 99% confidence intervals, you can specify the confidence level as follows.

Example: 'Alpha', 0.01

Data Types: `single` | `double`

### **DFMethod** — Method for computing approximate degrees of freedom

'residual' (default) | 'none'

Method for computing approximate degrees of freedom, specified as the comma-separated pair consisting of 'DFMethod' and one of the following.

Value	Description
'residual'	The degrees of freedom are assumed to be constant and equal to $n - p$ , where $n$ is the number of observations and $p$ is the number of fixed effects.
'none'	All degrees of freedom are set to infinity.

Example: 'DFMethod', 'none'

## Output Arguments

### **beta** — Estimated fixed-effects coefficients

vector

Estimated fixed-effects coefficients of the fitted generalized linear mixed-effects model `glme`, returned as a vector.

### **betanames** — Names of fixed-effects coefficients

table

Names of fixed-effects coefficients in `beta`, returned as a table.

### **stats** — Fixed-effects estimates and related statistics

dataset array

Fixed-effects estimates and related statistics, returned as a dataset array that has one row for each of the fixed effects and one column for each of the following statistics.

Column Name	Description
Name	Name of the fixed-effects coefficient
Estimate	Estimated coefficient value
SE	Standard error of the estimate
tStat	<i>t</i> -statistic for a test that the coefficient is 0
DF	Estimated degrees of freedom for the <i>t</i> -statistic
pValue	<i>p</i> -value for the <i>t</i> -statistic
Lower	Lower limit of a 95% confidence interval for the fixed-effects coefficient
Upper	Upper limit of a 95% confidence interval for the fixed-effects coefficient

When fitting a model using `fitglme` and one of the maximum likelihood fit methods ('Laplace' or 'ApproximateLaplace'), if you specify the 'CovarianceMethod' name-value pair argument as 'conditional', then SE does not account for the uncertainty in estimating the covariance parameters. To account for this uncertainty, specify 'CovarianceMethod' as 'JointHessian'.

When fitting a GLME model using `fitglme` and one of the pseudo likelihood fit methods ('MPL' or 'REMP'), `fixedEffects` bases the fixed effects estimates and related statistics on the fitted linear mixed-effects model from the final pseudo likelihood iteration.

## Examples

### Estimate Fixed-Effects Coefficients

Load the sample data.

```
load mfr
```

This simulated data is from a manufacturing company that operates 50 factories across the world, with each factory running a batch process to create a finished product. The company wants to decrease the number of defects in each batch, so it developed a new manufacturing process. To test the effectiveness of the new process, the company selected 20 of its factories at random to participate in an experiment: Ten factories implemented the new process, while the other ten continued to run the old process. In each of the 20 factories, the company ran five batches (for a total of 100 batches) and recorded the following data:

- Flag to indicate whether the batch used the new process (`newprocess`)
- Processing time for each batch, in hours (`time`)
- Temperature of the batch, in degrees Celsius (`temp`)
- Categorical variable indicating the supplier (A, B, or C) of the chemical used in the batch (`supplier`)
- Number of defects in the batch (`defects`)

The data also includes `time_dev` and `temp_dev`, which represent the absolute deviation of time and temperature, respectively, from the process standard of 3 hours at 20 degrees Celsius.

Fit a generalized linear mixed-effects model using `newprocess`, `time_dev`, `temp_dev`, and `supplier` as fixed-effects predictors. Include a random-effects term for intercept grouped by `factory`, to account for quality differences that might exist due to factory-specific variations. The response variable `defects` has a Poisson distribution, and the

appropriate link function for this model is log. Use the Laplace fit method to estimate the coefficients. Specify the dummy variable encoding as 'effects', so the dummy variable coefficients sum to 0.

The number of defects can be modeled using a Poisson distribution  
 $\text{defects}_{ij} \sim \text{Poisson}(\mu_{ij})$

This corresponds to the generalized linear mixed-effects model

$$\log(\mu_{ij}) = \beta_0 + \beta_1 \text{newprocess}_{ij} + \beta_2 \text{time.dev}_{ij} + \beta_3 \text{temp.dev}_{ij} + \beta_4 \text{supplier.C}_{ij} + \beta_5 \text{supplier.B}_{ij} + b_i,$$

where

- $\text{defects}_{ij}$  is the number of defects observed in the batch produced by factory  $i$  during batch  $j$ .
- $\mu_{ij}$  is the mean number of defects corresponding to factory  $i$  (where  $i = 1, 2, \dots, 20$ ) during batch  $j$  (where  $j = 1, 2, \dots, 5$ ).
- $\text{newprocess}_{ij}$ ,  $\text{time.dev}_{ij}$ , and  $\text{temp.dev}_{ij}$  are the measurements for each variable that correspond to factory  $i$  during batch  $j$ . For example,  $\text{newprocess}_{ij}$  indicates whether the batch produced by factory  $i$  during batch  $j$  used the new process.
- $\text{supplier.C}_{ij}$  and  $\text{supplier.B}_{ij}$  are dummy variables that use effects (sum-to-zero) coding to indicate whether company C or B, respectively, supplied the process chemicals for the batch produced by factory  $i$  during batch  $j$ .
- $b_i \sim N(0, \sigma_b^2)$  is a random-effects intercept for each factory  $i$  that accounts for factory-specific variation in quality.

```
glme = fitglme(mfr, 'defects ~ 1 + newprocess + time_dev + temp_dev + supplier + (1|factory)
  'Distribution', 'Poisson', 'Link', 'log', 'FitMethod', 'Laplace', 'DummyVarCoding', 'effects')
```

Compute and display the estimated fixed-effects coefficient values and related statistics.

```
[beta, betanames, stats] = fixedEffects(glme);
stats
```

```
stats =
```

```
FIXED EFFECT COEFFICIENTS: DFMETHOD = 'RESIDUAL', ALPHA = 0.05
```

Name	Estimate	SE	tStat	DF	pValue
' (Intercept) '	1.4689	0.15988	9.1875	94	9.8194e-15
'newprocess'	-0.36766	0.17755	-2.0708	94	0.041122
'time_dev'	-0.094521	0.82849	-0.11409	94	0.90941
'temp_dev'	-0.28317	0.9617	-0.29444	94	0.76907
'supplier_C'	-0.071868	0.078024	-0.9211	94	0.35936
'supplier_B'	0.071072	0.07739	0.91836	94	0.36078

Lower	Upper
1.1515	1.7864
-0.72019	-0.015134
-1.7395	1.5505
-2.1926	1.6263
-0.22679	0.083051
-0.082588	0.22473

The returned results indicate, for example, that the estimated coefficient for `temp_dev` is  $-0.28317$ . Its large *P*-value,  $0.76907$ , indicates that it is not a statistically significant predictor at the 5% significance level. Additionally, the confidence interval boundaries `Lower` and `Upper` indicate that the 95% confidence interval for the coefficient for `temp_dev` is  $[-2.1926, 1.6263]$ . This interval contains 0, which supports the conclusion that `temp_dev` is not statistically significant at the 5% significance level.

## See Also

`GeneralizedLinearMixedModel` | `coefCI` | `coefTest` | `fitglme` | `randomEffects`



# fixedEffects

**Class:** LinearMixedModel

Estimates of fixed effects and related statistics

## Syntax

```
beta = fixedEffects(lme)
[beta, betanames] = fixedEffects(lme)
[beta, betanames, stats] = fixedEffects(lme)
[beta, betanames, stats] = fixedEffects(lme, Name, Value)
```

## Description

`beta = fixedEffects(lme)` returns the estimated fixed-effects coefficients, `beta`, of the linear mixed-effects model `lme`.

`[beta, betanames] = fixedEffects(lme)` also returns the names of estimated fixed-effects coefficients in `betanames`. Each name corresponds to a fixed-effects coefficient in `beta`.

`[beta, betanames, stats] = fixedEffects(lme)` also returns the estimated fixed-effects coefficients of the linear mixed-effects model `lme` and related statistics in `stats`.

`[beta, betanames, stats] = fixedEffects(lme, Name, Value)` also returns the estimated fixed-effects coefficients of the linear mixed-effects model `lme` and related statistics with additional options specified by one or more `Name, Value` pair arguments.

## Input Arguments

**lme** — Linear mixed-effects model

LinearMixedModel object

Linear mixed-effects model, specified as a `LinearMixedModel` object constructed using `fitlme` or `fitlmematrix`.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### **Alpha** — Confidence level

0.05 (default) | scalar value in the range 0 to 1

Confidence level, specified as the comma-separated pair consisting of 'Alpha' and a scalar value in the range 0 to 1. For a value  $\alpha$ , the confidence level is  $100 \cdot (1 - \alpha)\%$ .

For example, for 99% confidence intervals, you can specify the confidence level as follows.

Example: 'Alpha', 0.01

Data Types: single | double

### **DFMethod** — Method for computing approximate degrees of freedom

'residual' (default) | 'satterthwaite' | 'none'

Method for computing approximate degrees of freedom for the  $t$ -statistic that tests the fixed-effects coefficients against 0, specified as the comma-separated pair consisting of 'DFMethod' and one of the following.

'residual'

Default. The degrees of freedom are assumed to be constant and equal to  $n - p$ , where  $n$  is the number of observations and  $p$  is the number of fixed effects.

'satterthwaite'

Satterthwaite approximation.

'none'

All degrees of freedom are set to infinity.

For example, you can specify the Satterthwaite approximation as follows.

Example: 'DFMethod', 'satterthwaite'

## Output Arguments

### **beta** — Fixed-effects coefficients estimates

vector

Fixed-effects coefficients estimates of the fitted linear mixed-effects model `lme`, returned as a vector.

### **betanames** — Names of fixed-effects coefficients

table

Names of fixed-effects coefficients in `beta`, returned as a table.

### **stats** — Fixed-effects estimates and related statistics

dataset array

Fixed-effects estimates and related statistics, returned as a dataset array that has one row for each of the fixed effects and one column for each of the following statistics.

Name	Name of the fixed effect coefficient
Estimate	Estimated coefficient value
SE	Standard error of the estimate
tStat	<i>t</i> -statistic for a test that the coefficient is zero
DF	Estimated degrees of freedom for the <i>t</i> -statistic
pValue	<i>p</i> -value for the <i>t</i> -statistic
Lower	Lower limit of a 95% confidence interval for the fixed-effect coefficient
Upper	Upper limit of a 95% confidence interval for the fixed-effect coefficient

## Examples

### Display Fixed-Effects Coefficient Estimates and Names

Load the sample data.

```
load(fullfile(matlabroot, 'examples', 'stats', 'weight.mat'));
```

The data set `weight` contains data from a longitudinal study, where 20 subjects are randomly assigned to 4 exercise programs, and their weight loss is recorded over six 2-week time periods. This is simulated data.

Store the data in a table. Define `Subject` and `Program` as categorical variables.

```
tbl = table(InitialWeight, Program, Subject, Week, y);  
tbl.Subject = nominal(tbl.Subject);  
tbl.Program = nominal(tbl.Program);
```

Fit a linear mixed-effects model where the initial weight, type of program, week, and the interaction between week and program are the fixed effects. The intercept and week vary by subject.

```
lme = fitlme(tbl, 'y ~ InitialWeight + Program*Week + (Week|Subject)');
```

Display the fixed-effects coefficient estimates and corresponding fixed-effects names.

```
[beta, betanames] = fixedEffects(lme)
```

```
beta =
```

```
    0.6610  
    0.0032  
    0.3608  
   -0.0333  
    0.1132  
    0.1732  
    0.0388  
    0.0305  
    0.0331
```

```
betanames =
```

```
9x1 table
```

```
      Name
```

```
-----  
'(Intercept)'  
'InitialWeight'
```

```
'Program_B'
'Program_C'
'Program_D'
'Week'
'Program_B:Week'
'Program_C:Week'
'Program_D:Week'
```

### Compute Coefficient Estimates and Related Statistics

Load the sample data.

```
load carbig
```

Fit a linear mixed-effects model for miles per gallon (MPG), with fixed effects for acceleration and horsepower, and potentially correlated random effects for intercept and acceleration grouped by model year. First, store the data in a table.

```
tbl = table(Acceleration,Horsepower,Model_Year,MPG);
```

Fit the model.

```
lme = fitlme(tbl, 'MPG ~ Acceleration + Horsepower + (Acceleration|Model_Year)');
```

Compute the fixed-effects coefficients estimates and related statistics.

```
[~,~,stats] = fixedEffects(lme)
```

```
stats =
```

```
FIXED EFFECT COEFFICIENTS: DFMETHOD = 'RESIDUAL', ALPHA = 0.05
```

Name	Estimate	SE	tStat	DF	pValue
'(Intercept)'	50.133	2.2652	22.132	389	7.7727e-71
'Acceleration'	-0.58327	0.13394	-4.3545	389	1.7075e-05
'Horsepower'	-0.16954	0.0072609	-23.35	389	5.188e-76

Lower	Upper
45.679	54.586

```
-0.84661    -0.31992  
-0.18382    -0.15527
```

The small  $P$ -values (under `pValue`) indicate that all fixed-effects coefficients are significant.

### Compute Confidence Intervals with Specified Options

Load the sample data.

```
load(fullfile(matlabroot, 'examples', 'stats', 'shift.mat'));
```

The data shows the deviations from the target quality characteristic measured from the products that five operators manufacture during three shifts: morning, evening, and night. This is a randomized block design, where the operators are the blocks. The experiment is designed to study the impact of the time of shift on the performance. The performance measure is the deviation of the quality characteristics from the target value. This is simulated data.

`Shift` and `Operator` are nominal variables.

```
shift.Shift = nominal(shift.Shift);  
shift.Operator = nominal(shift.Operator);
```

Fit a linear mixed-effects model with a random intercept grouped by operator to assess if performance significantly differs according to the time of the shift.

```
lme = fitlme(shift, 'QCDev ~ Shift + (1|Operator)');
```

Compute the 99% confidence intervals for fixed-effects coefficients, using the residual method to compute the degrees of freedom. This is the default method.

```
[~,~,stats] = fixedEffects(lme, 'alpha', 0.01)
```

```
stats =
```

```
FIXED EFFECT COEFFICIENTS: DFMETHOD = 'RESIDUAL', ALPHA = 0.01
```

Name	Estimate	SE	tStat	DF	pValue
------	----------	----	-------	----	--------

'(Intercept)'	3.1196	0.88681	3.5178	12	0.0042407
'Shift_Morning'	-0.3868	0.48344	-0.80009	12	0.43921
'Shift_Night'	1.9856	0.48344	4.1072	12	0.0014535

Lower	Upper
0.41081	5.8284
-1.8635	1.0899
0.5089	3.4623

Compute the 99% confidence intervals for fixed-effects coefficients, using the Satterthwaite approximation to compute the degrees of freedom.

```
[~,~,stats] = fixedEffects(lme,'DFMethod','satterthwaite','alpha',0.01)
```

```
stats =
```

```
FIXED EFFECT COEFFICIENTS: DFMETHOD = 'SATERTHWAITE', ALPHA = 0.01
```

Name	Estimate	SE	tStat	DF	pValue
'(Intercept)'	3.1196	0.88681	3.5178	6.123	0.01214
'Shift_Morning'	-0.3868	0.48344	-0.80009	10	0.44225
'Shift_Night'	1.9856	0.48344	4.1072	10	0.00212

Lower	Upper
-0.14122	6.3804
-1.919	1.1454
0.45343	3.5178

The Satterthwaite approximation usually produces smaller DF values than the residual method. That is why it produces larger  $P$ -values (pValue) and larger confidence intervals (see Lower and Upper).

## See Also

LinearMixedModel | coefCI | coefTest | fitlme | randomEffects

## fpdf

*F* probability density function

## Syntax

```
Y = fpdf(X, V1, V2)
```

## Description

`Y = fpdf(X, V1, V2)` computes the *F* pdf at each of the values in `X` using the corresponding numerator degrees of freedom `V1` and denominator degrees of freedom `V2`. `X`, `V1`, and `V2` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array with the same dimensions as the other inputs. `V1` and `V2` parameters must contain real positive values, and the values in `X` must lie on the interval `[0 Inf]`.

The probability density function for the *F* distribution is

$$y = f(x | v_1, v_2) = \frac{\Gamma\left[\frac{(v_1 + v_2)}{2}\right] \left(\frac{v_1}{v_2}\right)^{\frac{v_1}{2}} \frac{x^{\frac{v_1 - 2}{2}}}{\left[1 + \left(\frac{v_1}{v_2}\right)x\right]^{\frac{v_1 + v_2}{2}}}{\Gamma\left(\frac{v_1}{2}\right)\Gamma\left(\frac{v_2}{2}\right)}$$

## Examples

```
y = fpdf(1:6, 2, 2)
y =
    0.2500    0.1111    0.0625    0.0400    0.0278    0.0204

z = fpdf(3, 5:10, 5:10)
z =
    0.0689    0.0659    0.0620    0.0577    0.0532    0.0487
```



## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`fcdf` | `finv` | `frnd` | `fstat` | `pdf`

### Topics

“F Distribution” on page B-44

Introduced before R2006a

## fracfact

Fractional factorial design

### Syntax

```
X = fracfact(gen)
[X, conf] = fracfact(gen)
[X, conf] = fracfact(gen, Name, Value)
```

### Description

`X = fracfact(gen)` creates the two-level fractional factorial design defined by the generator `gen`.

`[X, conf] = fracfact(gen)` returns a cell array of character vectors containing the confounding pattern for the design.

`[X, conf] = fracfact(gen, Name, Value)` creates a fractional factorial designs with additional options specified by one or more `Name, Value` pair arguments.

### Input Arguments

#### **gen**

Either a cell array of character vectors where each cell contains one “word,” or a character array consisting of “words” separated by spaces. “Words” consist of case-sensitive letters or groups of letters, where 'a' represents value 1, 'b' represents value 2, ..., 'A' represents value 27, ..., 'Z' represents value 52.

Each word defines how the corresponding factor’s levels are defined as products of generators from a  $2^K$  full-factorial design.  $K$  is the number of letters of the alphabet in `gen`.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### **FactorNames**

Cell array specifying the name for each factor.

**Default:** {'X1', 'X2', ...}

### **MaxInt**

Positive integer setting the maximum level of interaction to include in the confounding output.

**Default:** 2

## Output Arguments

### **x**

The two-level fractional factorial design. `x` is a matrix of size `N`-by-`P`, where

- $N = 2^K$ , where  $K$  is the number of letters of the alphabet in `gen`.
- $P$  is the number of words in `gen`.

Because `x` is a two-level design, the components of `x` are  $\pm 1$ . For the meaning of `x`, see “Fractional Factorial Designs” on page 27-5.

### **conf**

Cell array of character vectors containing the confounding pattern for the design.

## Examples

Generate a fractional factorial design for four variables, where the fourth variable is the product of the first three:

```
x = fracfact('a b c abc')
```

```
x =  
  -1  -1  -1  -1  
  -1  -1   1   1  
  -1   1  -1   1  
  -1   1   1  -1  
   1  -1  -1   1  
   1  -1   1  -1  
   1   1  -1  -1  
   1   1   1   1
```

Find generators for a six-factor design that uses four factors and achieves resolution IV using `fracfactgen`. Use the result to specify the design:

```
generators = fracfactgen('a b c d e f',4, ... % 4 factors  
 4) % resolution 4
```

```
generators =  
  'a'  
  'b'  
  'c'  
  'd'  
  'bcd'  
  'acd'
```

```
x = fracfact(generators)
```

```
x =  
  -1  -1  -1  -1  -1  -1  
  -1  -1  -1   1   1   1  
  -1  -1   1  -1   1   1  
  -1  -1   1   1  -1  -1  
  -1   1  -1  -1   1  -1  
  -1   1  -1   1  -1   1  
  -1   1   1  -1  -1   1  
  -1   1   1   1   1  -1  
   1  -1  -1  -1  -1   1  
   1  -1  -1   1   1  -1
```

1	-1	1	-1	1	-1
1	-1	1	1	-1	1
1	1	-1	-1	1	1
1	1	-1	1	-1	-1
1	1	1	-1	-1	-1
1	1	1	1	1	1

## References

- [1] Box, G. E. P., W. G. Hunter, and J. S. Hunter. *Statistics for Experimenters*. Hoboken, NJ: Wiley-Interscience, 1978.

## See Also

`ff2n` | `fracfactgen` | `fullfact` | `hadamard`

## Topics

“Fractional Factorial Designs” on page 27-5

Introduced before R2006a

## fracfactgen

Fractional factorial design generators

### Syntax

```
generators = fracfactgen(terms)
generators = fracfactgen(terms, k)
generators = fracfactgen(terms, k, R)
generators = fracfactgen(terms, k, R, basic)
```

### Description

`generators = fracfactgen(terms)` uses the Franklin-Bailey algorithm to find generators for the smallest two-level fractional-factorial design for estimating linear model terms specified by `terms`. `terms` is a character vector consisting of words formed from the 52 case-sensitive letters a-z, separated by spaces. Use 'a'-'z' for the first 26 factors, and, if necessary, 'A'-'Z' for the remaining factors. For example, `terms = 'a b c ab ac'`. Single-character words indicate main effects to be estimated; multiple-character words indicate interactions. Alternatively, `terms` is an  $m$ -by- $n$  matrix of 0s and 1s where  $m$  is the number of model terms to be estimated and  $n$  is the number of factors. For example, if `terms` contains rows `[0 1 0 0]` and `[1 0 0 1]`, then the factor `b` and the interaction between factors `a` and `d` are included in the model. `generators` is a cell array of character vectors with one generator per cell. Pass `generators` to `fracfact` to produce the fractional-factorial design and corresponding confounding pattern.

`generators = fracfactgen(terms, k)` returns generators for a two-level fractional-factorial design with  $2^k$ -runs, if possible. If `k` is `[]`, `fracfactgen` finds the smallest design.

`generators = fracfactgen(terms, k, R)` finds a design with resolution `R`, if possible. The default resolution is 3.

A design of resolution  $R$  is one in which no  $n$ -factor interaction is confounded with any other effect containing less than  $R - n$  factors. Thus a resolution III design does not confound main effects with one another but may confound them with two-way

interactions, while a resolution IV design does not confound either main effects or two-way interactions but may confound two-way interactions with each other.

If `fracfactgen` is unable to find a design at the requested resolution, it tries to find a lower-resolution design sufficient to calibrate the model. If it is successful, it returns the generators for the lower-resolution design along with a warning. If it fails, it returns an error.

`generators = fracfactgen(terms, k, R, basic)` also accepts a vector `basic` specifying the indices of factors that are to be treated as basic. These factors receive full-factorial treatments in the design. The default includes factors that are part of the highest-order interaction in `terms`.

## Examples

Suppose you wish to determine the effects of four two-level factors, for which there may be two-way interactions. A full-factorial design would require  $2^4 = 16$  runs. The `fracfactgen` function finds generators for a resolution IV (separating main effects) fractional-factorial design that requires only  $2^3 = 8$  runs:

```
generators = fracfactgen('a b c d', 3, 4)
generators =
  'a'
  'b'
  'c'
  'abc'
```

The more economical design and the corresponding confounding pattern are returned by `fracfact`:

```
[dfF, confounding] = fracfact(generators)
dfF =
  -1  -1  -1  -1
  -1  -1   1   1
  -1   1  -1   1
  -1   1   1  -1
   1  -1  -1   1
   1  -1   1  -1
   1   1  -1  -1
   1   1   1   1
confounding =
```

'Term'	'Generator'	'Confounding'
'X1'	'a'	'X1'
'X2'	'b'	'X2'
'X3'	'c'	'X3'
'X4'	'abc'	'X4'
'X1*X2'	'ab'	'X1*X2 + X3*X4'
'X1*X3'	'ac'	'X1*X3 + X2*X4'
'X1*X4'	'bc'	'X1*X4 + X2*X3'
'X2*X3'	'bc'	'X1*X4 + X2*X3'
'X2*X4'	'ac'	'X1*X3 + X2*X4'
'X3*X4'	'ab'	'X1*X2 + X3*X4'

The confounding pattern shows, for example, that the two-way interaction between X1 and X2 is confounded by the two-way interaction between X3 and X4.

## References

- [1] Box, G. E. P., W. G. Hunter, and J. S. Hunter. *Statistics for Experimenters*. Hoboken, NJ: Wiley-Interscience, 1978.

## See Also

`fracfact` | `hadamard`

## Topics

“Fractional Factorial Designs” on page 27-5

Introduced in R2006a



# **friedman**

Friedman's test

## **Syntax**

```
p = friedman(x, reps)
p = friedman(x, reps, displayopt)
[p, tbl] = friedman( ___ )
[p, tbl, stats] = friedman( ___ )
```

## **Description**

`p = friedman(x, reps)` returns the  $p$ -value for the nonparametric Friedman's test to compare column effects in a two-way layout. `friedman` tests the null hypothesis that the column effects are all the same against the alternative that they are not all the same.

`p = friedman(x, reps, displayopt)` enables the ANOVA table display when `displayopt` is 'on' (default) and suppresses the display when `displayopt` is 'off'.

`[p, tbl] = friedman( ___ )` returns the ANOVA table (including column and row labels) in cell array `tbl`.

`[p, tbl, stats] = friedman( ___ )` also returns a structure `stats` that you can use to perform a follow-up multiple comparison test.

## **Examples**

### **Test For Column Effects Using Friedman's Test**

This example shows how to test for column effects in a two-way layout using Friedman's test.

Load the sample data.

```
load popcorn
popcorn
```

```
popcorn =
  5.5000  4.5000  3.5000
  5.5000  4.5000  4.0000
  6.0000  4.0000  3.0000
  6.5000  5.0000  4.0000
  7.0000  5.5000  5.0000
  7.0000  5.0000  4.5000
```

This data comes from a study of popcorn brands and popper type (Hogg 1987). The columns of the matrix `popcorn` are brands (Gourmet, National, and Generic). The rows are popper type (Oil and Air). The study popped a batch of each brand three times with each popper. The values are the yield in cups of popped popcorn.

Use Friedman's test to determine whether the popcorn brand affects the yield of popcorn.

```
p = friedman(popcorn,3)
```

```
p =
  0.0010
```

Friedman's ANOVA Table					
Source	SS	df	MS	Chi-sq	Prob>Chi-sq
Columns	99.75	2	49.875	13.76	0.001
Interaction	0.0833	2	0.0417		
Error	16.1667	12	1.3472		
Total	116	17			

Test for column effects after row effects are removed

The small value of  $p = 0.001$  indicates the popcorn brand affects the yield of popcorn.

## Input Arguments

### **x** — Sample data

matrix

Sample data for the hypothesis test, specified as a matrix. The columns of  $x$  represent changes in a factor A. The rows represent changes in a blocking factor B. If there is more than one observation for each combination of factors, input `reps` indicates the number of replicates in each “cell,” which must be constant.

Data Types: `single` | `double`

**reps** — Number of replicates per cell

1 (default) | positive integer value

Number of replicates per cell, specified as a positive integer value.

Data Types: `single` | `double`

**displayopt** — ANOVA table display option

'off' (default) | 'on'

ANOVA table display option, specified as 'off' or 'on'.

If `displayopt` is 'on', then `friedman` displays a figure showing an ANOVA table, which divides the variability of the ranks into two or three parts:

- The variability due to the differences among the column effects
- The variability due to the interaction between rows and columns (if `reps` is greater than its default value of 1)
- The remaining variability not explained by any systematic source

The ANOVA table has six columns:

- The first shows the source of the variability.
- The second shows the Sum of Squares (SS) due to each source.
- The third shows the degrees of freedom (df) associated with each source.
- The fourth shows the Mean Squares (MS), which is the ratio SS/df.
- The fifth shows Friedman's chi-square statistic.
- The sixth shows the  $p$  value for the chi-square statistic.

You can copy a text version of the ANOVA table to the clipboard by selecting `Copy Text` from the **Edit** menu.

## Output Arguments

**p** —  $p$ -value

scalar value in the range `[0, 1]`

$p$ -value of the test, returned as a scalar value in the range  $[0, 1]$ .  $p$  is the probability of observing a test statistic as extreme as, or more extreme than, the observed value under the null hypothesis. Small values of  $p$  cast doubt on the validity of the null hypothesis.

### **tbl** — ANOVA table

cell array

ANOVA table, including column and row labels, returned as a cell array. The ANOVA table has six columns:

- The first shows the source of the variability.
- The second shows the Sum of Squares (SS) due to each source.
- The third shows the degrees of freedom (df) associated with each source.
- The fourth shows the Mean Squares (MS), which is the ratio SS/df.
- The fifth shows Friedman's chi-square statistic.
- The sixth shows the  $p$  value for the chi-square statistic.

You can copy a text version of the ANOVA table to the clipboard by selecting `Copy Text` from the **Edit** menu.

### **stats** — Test data

structure

Test data, returned as a structure. `friedman` evaluates the hypothesis that the column effects are all the same against the alternative that they are not all the same. However, sometimes it is preferable to perform a test to determine which pairs of column effects are significantly different, and which are not. You can use the `multcompare` function to perform such tests by supplying `stats` as the input value.

## Definitions

### Friedman's Test

Friedman's test is similar to classical balanced two-way ANOVA, but it tests only for column effects after adjusting for possible row effects. It does not test for row effects or interaction effects. Friedman's test is appropriate when columns represent treatments that are under study, and rows represent nuisance effects (blocks) that need to be taken into account but are not of any interest.

The different columns of  $x$  represent changes in a factor A. The different rows represent changes in a blocking factor B. If there is more than one observation for each combination of factors, input `reps` indicates the number of replicates in each “cell,” which must be constant.

The matrix below illustrates the format for a set-up where column factor A has three levels, row factor B has two levels, and there are two replicates (`reps=2`). The subscripts indicate row, column, and replicate, respectively.

$$\begin{bmatrix} x_{111} & x_{121} & x_{131} \\ x_{112} & x_{122} & x_{132} \\ x_{211} & x_{221} & x_{231} \\ x_{212} & x_{222} & x_{232} \end{bmatrix}$$

Friedman's test assumes a model of the form

$$x_{ijk} = \mu + \alpha_i + \beta_j + \varepsilon_{ijk}$$

where  $\mu$  is an overall location parameter,  $\alpha_i$  represents the column effect,  $\beta_j$  represents the row effect, and  $\varepsilon_{ijk}$  represents the error. This test ranks the data within each level of B, and tests for a difference across levels of A. The  $p$  that `friedman` returns is the  $p$  value for the null hypothesis that  $\alpha_i = 0$ . If the  $p$  value is near zero, this casts doubt on the null hypothesis. A sufficiently small  $p$  value suggests that at least one column-sample median is significantly different than the others; i.e., there is a main effect due to factor A. The choice of a critical  $p$  value to determine whether a result is “statistically significant” is left to the researcher. It is common to declare a result significant if the  $p$  value is less than 0.05 or 0.01.

Friedman's test makes the following assumptions about the data in  $x$ :

- All data come from populations having the same continuous distribution, apart from possibly different locations due to column and row effects.
- All observations are mutually independent.

The classical two-way ANOVA replaces the first assumption with the stronger assumption that data come from normal distributions.

## References

- [1] Hogg, R. V., and J. Ledolter. *Engineering Statistics*. New York: MacMillan, 1987.
- [2] Hollander, M., and D. A. Wolfe. *Nonparametric Statistical Methods*. Hoboken, NJ: John Wiley & Sons, Inc., 1999.

## See Also

`anova2` | `kruskalwallis` | `multcompare`

**Introduced before R2006a**

## frnd

*F* random numbers

### Syntax

```
R = frnd(V1,V2)
R = frnd(V1,V2,m,n,...)
R = frnd(V1,V2,[m,n,...])
```

### Description

`R = frnd(V1,V2)` generates random numbers from the *F* distribution with numerator degrees of freedom `V1` and denominator degrees of freedom `V2`. `V1` and `V2` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input for `V1` or `V2` is expanded to a constant array with the same dimensions as the other input. `V1` and `V2` parameters must contain real positive values.

`R = frnd(V1,V2,m,n,...)` or `R = frnd(V1,V2,[m,n,...])` generates an `m`-by-`n`-by-... array containing random numbers from the *F* distribution with parameters `V1` and `V2`. `V1` and `V2` can each be scalars or arrays of the same size as `R`.

### Examples

```
n1 = frnd(1:6,1:6)
n1 =
    0.0022    0.3121    3.0528    0.3189    0.2715    0.9539
```

```
n2 = frnd(2,2,[2 3])
n2 =
    0.3186    0.9727    3.0268
    0.2052  148.5816    0.2191
```

```
n3 = frnd([1 2 3;4 5 6],1,2,3)
n3 =
```



```
0.6233 0.2322 31.5458  
2.5848 0.2121 4.4955
```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

The generated code can return a different sequence of numbers than MATLAB if either of the following is true:

- The output is nonscalar.
- An input parameter is invalid for the distribution.

### See Also

`fcdf` | `finv` | `fpdf` | `fstat` | `random`

### Topics

“F Distribution” on page B-44

Introduced before R2006a

## fscnca

Feature selection using neighborhood component analysis for classification

### Syntax

```
mdl = fscnca(X,Y)
mdl = fscnca(X,Y,Name,Value)
```

### Description

`mdl = fscnca(X,Y)` performs feature selection for classification using the predictors in `X` and responses in `Y`.

`fscnca` learns the feature weights by using a diagonal adaptation of neighborhood component analysis (NCA) with regularization.

`mdl = fscnca(X,Y,Name,Value)` performs feature selection for classification with additional options specified by one or more name-value pair arguments.

### Examples

#### Detect Relevant Features in Data Using NCA for Classification

Generate toy data where the response variable depends on the 3rd, 9th, and 15th predictors.

```
rng(0,'twister'); % For reproducibility
N = 100;
X = rand(N,20);
y = -ones(N,1);
y(X(:,3).*X(:,9)./X(:,15) < 0.4) = 1;
```

Fit the neighborhood component analysis model for classification.

```
mdl = fscnca(X,y,'Solver','sgd','Verbose',1);
```

o Tuning initial learning rate: NumTuningIterations = 20, TuningSubsetSize = 100

TUNING ITER	TUNING SUBSET FUN VALUE	LEARNING RATE
1	-3.755936e-01	2.000000e-01
2	-3.950971e-01	4.000000e-01
3	-4.311848e-01	8.000000e-01
4	-4.903195e-01	1.600000e+00
5	-5.630190e-01	3.200000e+00
6	-6.166993e-01	6.400000e+00
7	-6.255669e-01	1.280000e+01
8	-6.255669e-01	1.280000e+01
9	-6.255669e-01	1.280000e+01
10	-6.255669e-01	1.280000e+01
11	-6.255669e-01	1.280000e+01
12	-6.255669e-01	1.280000e+01
13	-6.255669e-01	1.280000e+01
14	-6.279210e-01	2.560000e+01
15	-6.279210e-01	2.560000e+01
16	-6.279210e-01	2.560000e+01
17	-6.279210e-01	2.560000e+01
18	-6.279210e-01	2.560000e+01
19	-6.279210e-01	2.560000e+01
20	-6.279210e-01	2.560000e+01

o Solver = SGD, MiniBatchSize = 10, PassLimit = 5

PASS	ITER	AVG MINIBATCH FUN VALUE	AVG MINIBATCH NORM GRAD	NORM STEP	LEARNING RATE
0	9	-5.658450e-01	4.492407e-02	9.290605e-01	2.560000e
1	19	-6.131382e-01	4.923625e-02	7.421541e-01	1.280000e
2	29	-6.225056e-01	3.738784e-02	3.277588e-01	8.533333e
3	39	-6.233366e-01	4.947901e-02	5.431133e-01	6.400000e
4	49	-6.238576e-01	3.445763e-02	2.946188e-01	5.120000e

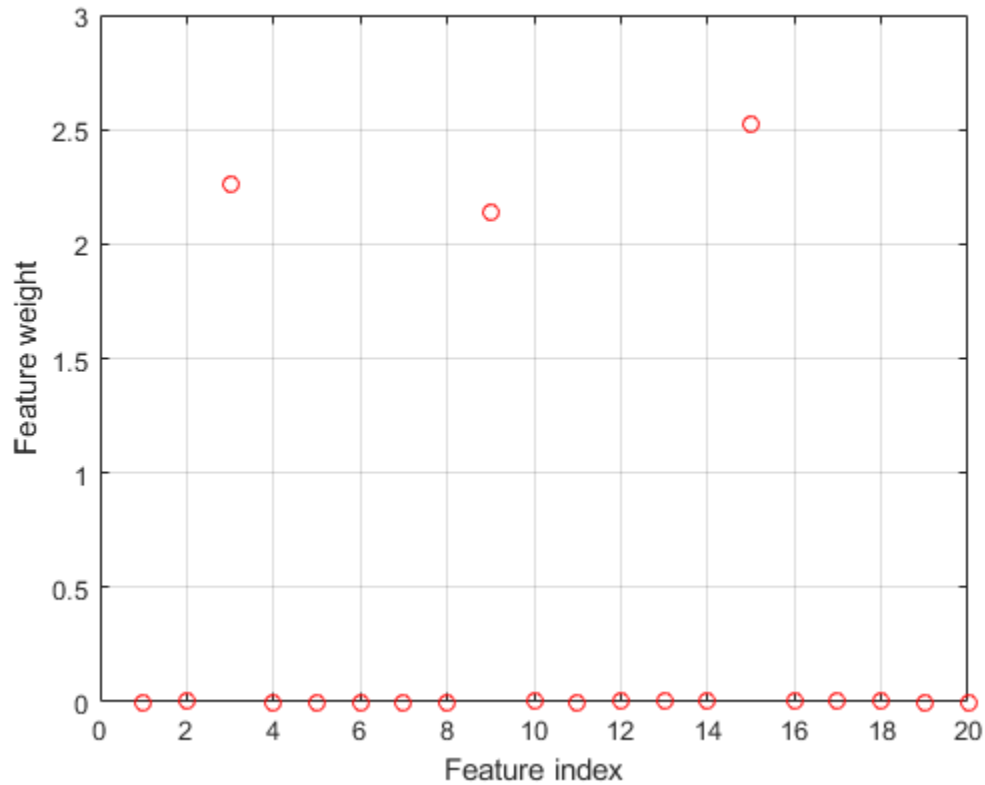
Two norm of the final step = 2.946e-01

Relative two norm of the final step = 6.588e-02, TolX = 1.000e-06

EXIT: Iteration or pass limit reached.

Plot the selected features. The weights of the irrelevant features should be close to zero.

```
figure()
plot mdl.FeatureWeights, 'ro'
grid on
xlabel('Feature index')
ylabel('Feature weight')
```



fscnca correctly detects the relevant features.

## Identify Relevant Features for Classification

### Load sample data

```
load ovariancancer;
whos
```

Name	Size	Bytes	Class	Attributes
grp	216x1	26784	cell	
obs	216x4000	3456000	single	

This example uses the high-resolution ovarian cancer data set that was generated using the WCX2 protein array. The data is from the FDA-NCI Clinical Proteomics Program Databank. After some preprocessing steps, the data set has two variables: `obs` and `grp`. The `obs` variable consists 216 observations with 4000 features. Each element in `grp` defines the group to which the corresponding row of `obs` belongs.

### Divide data into training and test sets

Use `cvpartition` to divide data into a training set of size 160 and a test set of size 56. Both the training set and the test set have roughly the same group proportions as in `grp`.

```
rng(1); % For reproducibility
cvp = cvpartition(grp, 'holdout', 56)
Xtrain = obs(cvp.training, :);
ytrain = grp(cvp.training, :);
Xtest = obs(cvp.test, :);
ytest = grp(cvp.test, :);
```

```
cvp =
```

```
Hold-out cross validation partition
  NumObservations: 216
    NumTestSets: 1
     TrainSize: 160
     TestSize: 56
```

### Determine if feature selection is necessary

Compute generalization error without fitting.

```
nca = fscnca(Xtrain,ytrain,'FitMethod','none');  
L = loss(nca,Xtest,ytest)
```

```
L =  
  
0.0893
```

This option computes the generalization error of the neighborhood component analysis (NCA) feature selection model using the initial feature weights (in this case the default feature weights) provided in `fscnca`.

Fit NCA without regularization parameter (Lambda = 0)

```
nca = fscnca(Xtrain,ytrain,'FitMethod','exact','Lambda',0,...  
            'Solver','sgd','Standardize',true);  
L = loss(nca,Xtest,ytest)
```

```
L =  
  
0.0714
```

The improvement on the loss value suggests that feature selection is a good idea. Tuning the  $\lambda$  value usually improves the results.

### Tune the regularization parameter for NCA using five-fold cross-validation

Tuning  $\lambda$  means finding the  $\lambda$  value that produces produce the minimum classification loss. To tune  $\lambda$  using cross-validation:

1. Partition the training data into five folds and extract the number of validation (test) sets. For each fold, `cvpartition` assigns four-fifths of the data as a training set, and one-fifth of the data as a test set.

```
cvp = cvpartition(ytrain,'kfold',5);  
numvalidsets = cvp.NumTestSets;
```

Assign  $\lambda$  values and create an array to store the loss function values.

```
n = length(ytrain);  
lambdaval = linspace(0,20,20)/n;  
lossvals = zeros(length(lambdaval),numvalidsets);
```

2. Train the NCA model for each  $\lambda$  value, using the training set in each fold.
3. Compute the classification loss for the corresponding test set in the fold using the NCA model. Record the loss value.
4. Repeat this process for all folds and all  $\lambda$  values.

```

for i = 1:length(lambdaval)
    for k = 1:numvalidsets
        X = Xtrain(cvp.training(k),:);
        y = ytrain(cvp.training(k),:);
        Xvalid = Xtrain(cvp.test(k),:);
        yvalid = ytrain(cvp.test(k),:);

        nca = fscnca(X,y,'FitMethod','exact', ...
                    'Solver','sgd','Lambda',lambdaval(i), ...
                    'IterationLimit',30,'GradientTolerance',1e-4, ...
                    'Standardize',true);

        lossvals(i,k) = loss(nca,Xvalid,yvalid,'LossFunction','classiferror');
    end
end

```

Compute the average loss obtained from the folds for each  $\lambda$  value.

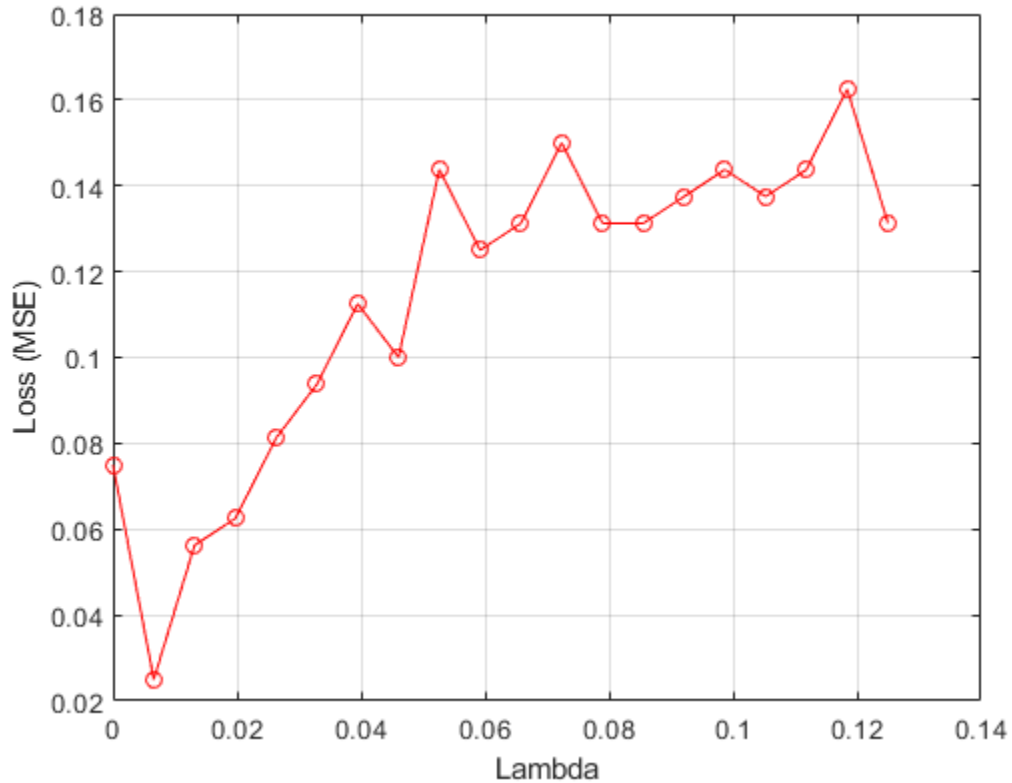
```
meanloss = mean(lossvals,2);
```

Plot the average loss values versus the  $\lambda$  values.

```

figure()
plot(lambdaval,meanloss,'ro-')
xlabel('Lambda')
ylabel('Loss (MSE)')
grid on

```



Find the best lambda value that corresponds to the minimum average loss.

```
[~,idx] = min(meanloss) % Find the index  
bestlambda = lambdaval(idx) % Find the best lambda value  
bestloss = meanloss(idx)
```

```
idx =
```

```
2
```

```
bestlambda =
```



```
0.0066
```

```
bestloss =
```

```
0.0250
```

**Fit the nca model on all data using best  $\lambda$  and plot the feature weights**

Use the solver lbfgs and standardize the predictor values.

```
nca = fscnca(Xtrain,ytrain,'FitMethod','exact','Solver','sgd',...
            'Lambda',bestlambda,'Standardize',true,'Verbose',1);
```

```
o Tuning initial learning rate: NumTuningIterations = 20, TuningSubsetSize = 100
```

```
=====|
| TUNING      | TUNING SUBSET | LEARNING  |
|  ITER      | FUN VALUE    | RATE      |
|=====|
|           1 | 2.403497e+01 | 2.000000e-01 |
|           2 | 2.275050e+01 | 4.000000e-01 |
|           3 | 2.036845e+01 | 8.000000e-01 |
|           4 | 1.627647e+01 | 1.600000e+00 |
|           5 | 1.023512e+01 | 3.200000e+00 |
|           6 | 3.864283e+00 | 6.400000e+00 |
|           7 | 4.743816e-01 | 1.280000e+01 |
|           8 | -7.260138e-01 | 2.560000e+01 |
|           9 | -7.260138e-01 | 2.560000e+01 |
|          10 | -7.260138e-01 | 2.560000e+01 |
|          11 | -7.260138e-01 | 2.560000e+01 |
|          12 | -7.260138e-01 | 2.560000e+01 |
|          13 | -7.260138e-01 | 2.560000e+01 |
|          14 | -7.260138e-01 | 2.560000e+01 |
|          15 | -7.260138e-01 | 2.560000e+01 |
|          16 | -7.260138e-01 | 2.560000e+01 |
|          17 | -7.260138e-01 | 2.560000e+01 |
|          18 | -7.260138e-01 | 2.560000e+01 |
|          19 | -7.260138e-01 | 2.560000e+01 |
|          20 | -7.260138e-01 | 2.560000e+01 |
```

```
o Solver = SGD, MiniBatchSize = 10, PassLimit = 5
```

```

=====
| PASS | ITER | AVG MINIBATCH | AVG MINIBATCH | NORM STEP | LEARNING
|      |      | FUN VALUE     | NORM GRAD     |           | RATE
=====
| 0 | 9 | 4.016078e+00 | 2.835465e-02 | 5.395984e+00 | 2.560000e
| 1 | 19 | -6.726156e-01 | 6.111354e-02 | 5.021138e-01 | 1.280000e
| 1 | 29 | -8.316555e-01 | 4.024185e-02 | 1.196030e+00 | 1.280000e
| 2 | 39 | -8.838656e-01 | 2.333418e-02 | 1.225839e-01 | 8.533333e
| 3 | 49 | -8.669035e-01 | 3.413150e-02 | 3.421881e-01 | 6.400000e
| 3 | 59 | -8.906935e-01 | 1.946293e-02 | 2.232510e-01 | 6.400000e
| 4 | 69 | -8.778630e-01 | 3.561283e-02 | 3.290643e-01 | 5.120000e
| 4 | 79 | -8.857136e-01 | 2.516633e-02 | 3.902977e-01 | 5.120000e
=====

```

Two norm of the final step = 3.903e-01

Relative two norm of the final step = 6.171e-03, TolX = 1.000e-06

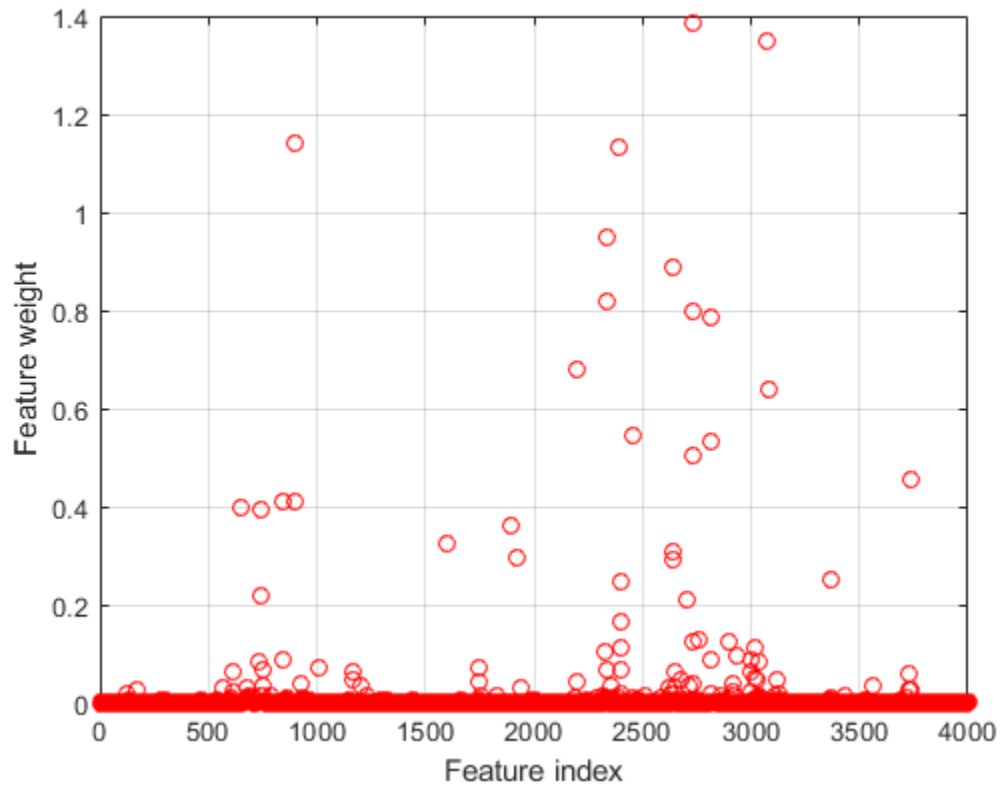
EXIT: Iteration or pass limit reached.

**Plot the feature weights.**

```

figure()
plot(nca.FeatureWeights, 'ro')
xlabel('Feature index')
ylabel('Feature weight')
grid on

```



Select features using the feature weights and a relative threshold.

```
tol = 0.02;  
selidx = find(nca.FeatureWeights > tol*max(1,max(nca.FeatureWeights)))
```

```
selidx =
```

```
565  
611  
654  
681  
737  
743
```

744  
750  
754  
839  
840  
897  
899  
925  
1010  
1166  
1167  
1199  
1601  
1744  
1751  
1894  
1923  
1936  
2195  
2202  
2327  
2333  
2336  
2337  
2357  
2391  
2397  
2399  
2400  
2401  
2452  
2621  
2642  
2643  
2644  
2645  
2649  
2650  
2677  
2709  
2710  
2730  
2731  
2734

```
2735
2736
2762
2813
2814
2820
2897
2916
2931
2998
2999
3014
3022
3032
3033
3078
3080
3118
3367
3563
3727
3733
```

Compute the classification loss using the test set.

```
L = loss(nca,Xtest,ytest)
```

```
L =
```

```
0.0179
```

### **Classify observations using the selected features**

Extract the features with feature weights greater than 0 from the training data.

```
features = Xtrain(:,selidx);
```

Apply a support vector machine classifier using the selected features to the reduced training set.

```
svmMdl = fitcsvm(features,ytrain);
```

Evaluate the accuracy of the trained classifier on the test data which has not been used for selecting features.

```
L = loss(svmMdl,Xtest(:,selidx),ytest)
```

```
L =
```

```
single
```

```
0
```

- “Tune Regularization Parameter to Detect Features Using NCA for Classification”

## Input Arguments

### **x** — Predictor variable values

*n*-by-*p* matrix

Predictor variable values, specified as an *n*-by-*p* matrix, where *n* is the number of observations and *p* is the number of predictor variables.

Data Types: `single` | `double`

### **y** — Class labels

categorical vector | logical vector | numeric vector | cell array of character vectors of length *n* | character matrix with *n* rows

Class labels, specified as a categorical vector, logical vector, numeric vector, cell array of character vectors of length *n*, or character matrix with *n* rows, where *n* is the number of observations. Element *i* or row *i* of **Y** is the class label corresponding to row *i* of **X** (observation *i*).

Data Types: `single` | `double` | `logical` | `char` | `cell` | `categorical`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of **Name**, **Value** arguments. **Name** is the argument name and **Value** is the corresponding value. **Name** must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as **Name1**, **Value1**, ..., **NameN**, **ValueN**.

Example: `'Solver', 'sgd', 'Weights', W, 'Lambda', 0.0003` specifies the solver as the stochastic gradient descent, the observation weights as the values in the vector `W`, and sets the regularization parameter at 0.0003.

### Fitting Options

#### **FitMethod** — Method for fitting the model

`'exact'` (default) | `'none'` | `'average'`

Method for fitting the model, specified as the comma-separated pair consisting of `'FitMethod'` and one of the following:

- `'exact'` — Performs fitting using all of the data.
- `'none'` — No fitting. Use this option to evaluate the generalization error of the NCA model using the initial feature weights supplied in the call to `fscnca`.
- `'average'` — Divides the data into partitions (subsets), fits each partition using the exact method, and returns the average of the feature weights. You can specify the number of partitions using the `NumPartitions` name-value pair argument.

Example: `'FitMethod', 'none'`

Data Types: `char`

#### **NumPartitions** — Number of partitions

`max(2, min(10, n))` (default) | integer between 2 and  $n$

Number of partitions to split the data for using with `'FitMethod', 'average'` option, specified as the comma-separated pair consisting of `'NumPartitions'` and an integer value between 2 and  $n$ , where  $n$  is the number of observations.

Example: `'NumPartitions', 15`

Data Types: `double` | `single`

#### **Lambda** — Regularization parameter

`1/n` (default) | nonnegative scalar

Regularization parameter, specified as the comma-separated pair consisting of `'Lambda'` and a nonnegative scalar.

For  $n$  observations, the best `Lambda` value that minimizes the generalization error of the NCA model is expected to be a multiple of  $1/n$ .

Example: `'Lambda',0.002`

Data Types: `double` | `single`

### **LengthScale** — Width of the kernel

1 (default) | positive real scalar

Width of the kernel, specified as the comma-separated pair consisting of `'LengthScale'` and a positive real scalar.

A length scale value of 1 is sensible when all predictors are on the same scale. If the predictors in  $X$  are of very different magnitudes, then consider standardizing the predictor values using `'Standardize',true` and setting `'LengthScale',1`.

Example: `'LengthScale',1.5`

Data Types: `double` | `single`

### **InitialFeatureWeights** — Initial feature weights

`ones(p,1)` (default) |  $p$ -by-1 vector of real positive scalars

Initial feature weights, specified as the comma-separated pair consisting of `'InitialFeatureWeights'` and a  $p$ -by-1 vector of real positive scalars, where  $p$  is the number of predictors in the training data.

The regularized objective function for optimizing feature weights is nonconvex. As a result, using different initial feature weights can give different results. Setting all initial feature weights to 1 generally works well, but in some cases, random initialization using `rand(p,1)` can give better quality solutions.

Data Types: `double` | `single`

### **Weights** — Observation weights

$n$ -by-1 vector of 1s (default) |  $n$ -by-1 vector of real positive scalars

Observation weights, specified as the comma-separated pair consisting of `'ObservationWeights'` and an  $n$ -by-1 vector of real positive scalars. Use observation weights to specify higher importance of some observations compared to others. The default weights assign equal importance to all observations.

Data Types: `double` | `single`

### **Prior** — Prior probabilities for each class

`'empirical'` (default) | `'uniform'` | structure



Prior probabilities for each class, specified as the comma-separated pair consisting of 'Prior' and one of the following:

- 'empirical' — fscnca obtains the prior class probabilities from class frequencies.
- 'uniform' — fscnca sets all class probabilities equal.
- Structure with two fields:
  - ClassProbs — Vector of class probabilities. If these are numeric values with a total greater than 1, fscnca normalizes them to add up to 1.
  - ClassNames — Class names corresponding to the class probabilities in ClassProbs.

Example: 'Prior', 'uniform'

Data Types: char | struct

### **Standardize** — Indicator for standardizing predictor data

false (default) | true

Indicator for standardizing the predictor data, specified as the comma-separated pair consisting of 'Standardize' and either false or true. For more information, see “Impact of Standardization” on page 15-130.

Example: 'Standardize', true

Data Types: logical

### **Verbose** — Verbosity level indicator

0 (default) | 1 | >1

Verbosity level indicator for the convergence summary display, specified as the comma-separated pair consisting of 'Verbose' and one of the following:

- 0 — No convergence summary
- 1 — Convergence summary, including norm of gradient and objective function values
- > 1 — More convergence information, depending on the fitting algorithm

When using 'minibatch-lbfgs' solver and verbosity level > 1, the convergence information includes iteration the log from intermediate mini-batch LBFGS fits.

Example: 'Verbose', 1

Data Types: `double` | `single`

### **Solver** — Solver type

`'lbfgs'` | `'sgd'` | `'minibatch-lbfgs'`

Solver type for estimating feature weights, specified as the comma-separated pair consisting of `'Solver'` and one of the following:

- `'lbfgs'` — Limited memory Broyden-Fletcher-Goldfarb-Shanno (LBFGS) algorithm
- `'sgd'` — Stochastic gradient descent (SGD) algorithm
- `'minibatch-lbfgs'` — Stochastic gradient descent with LBFGS algorithm applied to mini-batches

Default is `'lbfgs'` for  $n \leq 1000$ , and `'sgd'` for  $n > 1000$ .

Example: `'solver', 'minibatch-lbfgs'`

Data Types: `char`

### **LossFunction** — Loss function

`'classiferror'` (default) | function handle

Loss function, specified as the comma-separated pair consisting of `'LossFunction'` and one of the following.

- `'classiferror'` — Misclassification error, defined as

$$\frac{1}{n} \sum_{i=1}^n I(k_i \neq t_i),$$

where  $k_i$  is the predicted class and  $t_i$  is the true class for observation  $i$ .  $I(k_i \neq t_i)$  is the indicator for when the  $k_i$  is not the same as  $t_i$ .

- Function handle — Custom loss function. For example, for the `'LossFunction', L` name-value pair, `L` must have the following form:

```
L = lossFcn(Yn, Ym)
```

`Yn` is an  $n$ -by-1 vector and `Ym` is an  $m$ -by-1 vector. `L` is an  $n$ -by- $m$  matrix of loss values such that `L(i, j)` is the loss value for `Yn(i)` and `Ym(j)`.

Example: `'LossFunction', L`

Data Types: char | function\_handle

### **CacheSize — Memory size**

1000MB (default) | integer

Memory size, in MB, to use for objective function and gradient computation, specified as the comma-separated pair consisting of 'CacheSize' and an integer.

Example: 'CacheSize',1500MB

Data Types: double | single

### **LBFGS Options**

#### **HessianHistorySize — Size of history buffer for Hessian approximation**

15 (default) | positive integer

Size of history buffer for Hessian approximation for the 'lbfgs' solver, specified as the comma-separated pair consisting of 'HessianHistorySize' and a positive integer. At each iteration the function uses the most recent HessianHistorySize iterations to build an approximation to the inverse Hessian.

Example: 'HessianHistorySize',20

Data Types: double | single

#### **InitialStepSize — Initial step size**

'auto' (default) | positive real scalar

Initial step size for the 'lbfgs' solver, specified as the comma-separated pair consisting of 'InitialStepSize' and a positive real scalar. By default, the function determines the initial step size automatically.

Data Types: double | single

#### **LineSearchMethod — Line search method**

'weakwolfe' (default) | 'strongwolfe' | 'backtracking'

Line search method, specified as the comma-separated pair consisting of 'LineSearchMethod' and one of the following:

- 'weakwolfe' — Weak Wolfe line search
- 'strongwolfe' — Strong Wolfe line search

- `'backtracking'` — Backtracking line search

Example: `'LineSearchMethod', 'backtracking'`

Data Types: `char`

### **MaxLineSearchIterations** — Maximum number of line search iterations

20 (default) | positive integer

Maximum number of line search iterations, specified as the comma-separated pair consisting of `'MaxLineSearchIterations'` and a positive integer.

Example: `'MaxLineSearchIterations', 25`

Data Types: `double` | `single`

### **GradientTolerance** — Relative convergence tolerance

1e-6 (default) | positive real scalar

Relative convergence tolerance on the gradient norm for solver `lbfgs`, specified as the comma-separated pair consisting of `'GradientTolerance'` and a positive real scalar.

Example: `'GradientTolerance', 0.000002`

Data Types: `double` | `single`

## **SGD Options**

### **InitialLearningRate** — Initial learning rate for `'sgd'` solver

`'auto'` (default) | positive real scalar

Initial learning rate for the `'sgd'` solver, specified as the comma-separated pair consisting of `'InitialLearningRate'` and a positive real scalar.

When using solver type `'sgd'`, the learning rate decays over iterations starting with the value specified for `'InitialLearningRate'`.

The default `'auto'` means that the initial learning rate is determined using experiments on small subsets of data. Use the `NumTuningIterations` name-value pair argument to specify the number of iterations for automatically tuning the initial learning rate. Use the `TuningSubsetSize` name-value pair argument to specify the number of observations to use for automatically tuning the initial learning rate.

For solver type 'minibatch-lbfgs', you can set 'InitialLearningRate' to a very high value. In this case, the function applies LBFGS to each mini-batch separately with initial feature weights from the previous mini-batch.

To make sure the chosen initial learning rate decreases the objective value with each iteration, plot the Iteration versus the Objective values saved in the mdl.FitInfo property.

You can use the refit method with 'InitialFeatureWeights' equal to mdl.FeatureWeights to start from the current solution and run additional iterations

Example: 'InitialLearningRate',0.9

Data Types: double | single

**MiniBatchSize** — Number of observations to use in each batch for the 'sgd' solver

min(10, $n$ ) (default) | positive integer value from 1 to  $n$

Number of observations to use in each batch for the 'sgd' solver, specified as the comma-separated pair consisting of 'MiniBatchSize' and a positive integer from 1 to  $n$ .

Example: 'MiniBatchSize',25

Data Types: double | single

**PassLimit** — Maximum number of passes for solver 'sgd'

5 (default) | positive integer

Maximum number of passes through all  $n$  observations for solver 'sgd', specified as the comma-separated pair consisting of 'PassLimit' and a positive integer. Each pass through all of the data is called an epoch.

Example: 'PassLimit',10

Data Types: double | single

**NumPrint** — Frequency of batches for displaying convergence summary

10 (default) | positive integer value

Frequency of batches for displaying convergence summary for the 'sgd' solver, specified as the comma-separated pair consisting of 'NumPrint' and a positive integer. This argument applies when the 'Verbose' value is greater than 0. NumPrint mini-

batches are processed for each line of the convergence summary that is displayed on the command line.

Example: `'NumPrint',5`

Data Types: `double | single`

### **NumTuningIterations — Number of tuning iterations**

20 (default) | positive integer

Number of tuning iterations for the 'sgd' solver, specified as the comma-separated pair consisting of 'NumTuningIterations' and a positive integer. This option is valid only for 'InitialLearningRate', 'auto'.

Example: `'NumTuningIterations',15`

Data Types: `double | single`

### **TuningSubsetSize — Number of observations to use for tuning initial learning rate**

$\min(100,n)$  (default) | positive integer value from 1 to  $n$

Number of observations to use for tuning the initial learning rate, specified as the comma-separated pair consisting of 'TuningSubsetSize' and a positive integer value from 1 to  $n$ . This option is valid only for 'InitialLearningRate', 'auto'.

Example: `'TuningSubsetSize',25`

Data Types: `double | single`

### **SGD or LBFGS Options**

#### **IterationLimit — Maximum number of iterations**

positive integer

Maximum number of iterations, specified as the comma-separated pair consisting of 'IterationLimit' and a positive integer. The default is 10000 for SGD and 1000 for LBFGS and mini-batch LBFGS.

Each pass through a batch is an iteration. Each pass through all of the data is an epoch. If the data is divided into  $k$  mini-batches, then every epoch is equivalent to  $k$  iterations.

Example: `'IterationLimit',250`

Data Types: `double | single`

**StepTolerance — Convergence tolerance on the step size**

1e-6 (default) | positive real scalar

Convergence tolerance on the step size, specified as the comma-separated pair consisting of 'StepTolerance' and a positive real scalar. The 'lbfgs' solver uses an absolute step tolerance, and the 'sgd' solver uses a relative step tolerance.

Example: 'StepTolerance',0.000005

Data Types: double | single

**Mini-Batch LBFGS Options****MiniBatchLBFGSIterations — Maximum number of iterations per mini-batch LBFGS step**

10 (default) | positive integer

Maximum number of iterations per mini-batch LBFGS step, specified as the comma-separated pair consisting of 'MiniBatchLBFGSIterations' and a positive integer.

Example: 'MiniBatchLBFGSIterations',15

**Mini-batch LBFGS algorithm is a combination of SGD and LBFGS methods. Therefore, all of the name-value pair arguments that apply to SGD and LBFGS solvers also apply to the mini-batch LBFGS algorithm.**

Data Types: double | single

## Output Arguments

**mdl — Neighborhood component analysis model for classification**

FeatureSelectionNCAClassification object

Neighborhood component analysis model for classification, returned as a FeatureSelectionNCAClassification object.

## See Also

FeatureSelectionNCAClassification | loss | predict | refit

## **Topics**

“Tune Regularization Parameter to Detect Features Using NCA for Classification”

“Neighborhood Component Analysis (NCA) Feature Selection” on page 15-126

**Introduced in R2016b**



# fsrnca

Feature selection using neighborhood component analysis for regression

## Syntax

```
mdl = fsrnca(X,Y)
mdl = fsrnca(X,Y,Name,Value)
```

## Description

`mdl = fsrnca(X,Y)` performs feature selection for regression using the predictors in `X` and responses in `Y`.

`fsrnca` learns the feature weights by a diagonal adaptation of neighborhood component analysis (NCA) with regularization.

`mdl = fsrnca(X,Y,Name,Value)` performs feature selection for regression with additional options specified by one or more name-value pair arguments.

## Examples

### Detect Relevant Features in Data Using NCA for Regression

Generate toy data where the response variable depends on the 3rd, 9th, and 15th predictors.

```
rng(0,'twister'); % For reproducibility
N = 100;
X = rand(N,20);
y = 1 + X(:,3)*5 + sin(X(:,9)./X(:,15) + 0.25*randn(N,1));
```

Fit the neighborhood component analysis model for regression.

```
mdl = fsrnca(X,y,'Verbose',1,'Lambda',0.5/N);
```

o Solver = LBFGS, HessianHistorySize = 15, LineSearchMethod = weakwolfe

```

=====
| ITER | FUN VALUE | NORM GRAD | NORM STEP | CURV | GAMMA | ALPH
=====
| 0 | 1.636932e+00 | 3.688e-01 | 0.000e+00 | | 1.627e+00 | 0.000
| 1 | 8.304833e-01 | 1.083e-01 | 2.449e+00 | OK | 9.194e+00 | 4.000
| 2 | 7.548105e-01 | 1.341e-02 | 1.164e+00 | OK | 1.095e+01 | 1.000
| 3 | 7.346997e-01 | 9.752e-03 | 6.383e-01 | OK | 2.979e+01 | 1.000
| 4 | 7.053407e-01 | 1.605e-02 | 1.712e+00 | OK | 5.809e+01 | 1.000
| 5 | 6.970502e-01 | 9.106e-03 | 8.818e-01 | OK | 6.223e+01 | 1.000
| 6 | 6.952347e-01 | 5.522e-03 | 6.382e-01 | OK | 3.280e+01 | 1.000
| 7 | 6.946302e-01 | 9.102e-04 | 1.952e-01 | OK | 3.380e+01 | 1.000
| 8 | 6.945037e-01 | 6.557e-04 | 9.942e-02 | OK | 8.490e+01 | 1.000
| 9 | 6.943908e-01 | 1.997e-04 | 1.756e-01 | OK | 1.124e+02 | 1.000
| 10 | 6.943785e-01 | 3.478e-04 | 7.755e-02 | OK | 7.621e+01 | 1.000
| 11 | 6.943728e-01 | 1.428e-04 | 3.416e-02 | OK | 3.649e+01 | 1.000
| 12 | 6.943711e-01 | 1.128e-04 | 1.231e-02 | OK | 6.092e+01 | 1.000
| 13 | 6.943688e-01 | 1.066e-04 | 2.326e-02 | OK | 9.319e+01 | 1.000
| 14 | 6.943655e-01 | 9.324e-05 | 4.399e-02 | OK | 1.810e+02 | 1.000
| 15 | 6.943603e-01 | 1.206e-04 | 8.823e-02 | OK | 4.609e+02 | 1.000
| 16 | 6.943582e-01 | 1.701e-04 | 6.669e-02 | OK | 8.425e+01 | 5.000
| 17 | 6.943552e-01 | 5.160e-05 | 6.473e-02 | OK | 8.832e+01 | 1.000
| 18 | 6.943546e-01 | 2.477e-05 | 1.215e-02 | OK | 7.925e+01 | 1.000
| 19 | 6.943546e-01 | 1.077e-05 | 6.086e-03 | OK | 1.378e+02 | 1.000
=====
| ITER | FUN VALUE | NORM GRAD | NORM STEP | CURV | GAMMA | ALPH
=====
| 20 | 6.943545e-01 | 2.260e-05 | 4.071e-03 | OK | 5.856e+01 | 1.000
| 21 | 6.943545e-01 | 4.250e-06 | 1.109e-03 | OK | 2.964e+01 | 1.000
| 22 | 6.943545e-01 | 1.916e-06 | 8.356e-04 | OK | 8.649e+01 | 1.000
| 23 | 6.943545e-01 | 1.083e-06 | 5.270e-04 | OK | 1.168e+02 | 1.000
| 24 | 6.943545e-01 | 1.791e-06 | 2.673e-04 | OK | 4.016e+01 | 1.000
| 25 | 6.943545e-01 | 2.596e-07 | 1.111e-04 | OK | 3.154e+01 | 1.000

```

Infinity norm of the final gradient = 2.596e-07

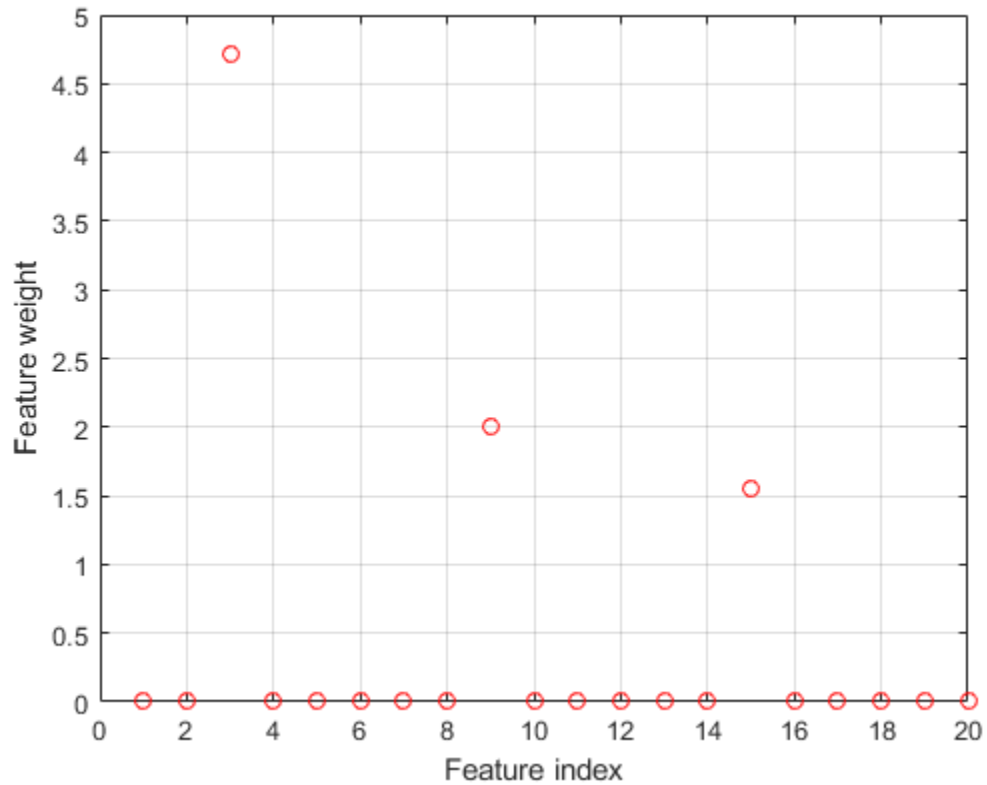
Two norm of the final step = 1.111e-04, TolX = 1.000e-06

Relative infinity norm of the final gradient = 2.596e-07, TolFun = 1.000e-06

EXIT: Local minimum found.

Plot the selected features. The weights of the irrelevant features should be close to zero.

```
figure()
plot mdl.FeatureWeights, 'ro'
grid on
xlabel('Feature index')
ylabel('Feature weight')
```



fsrnca correctly detects the relevant predictors for this response.

### Tune Regularization Parameter in NCA for Regression

Load the sample data.

```
load(fullfile(matlabroot, 'examples', 'stats', 'robotarm.mat'))
```

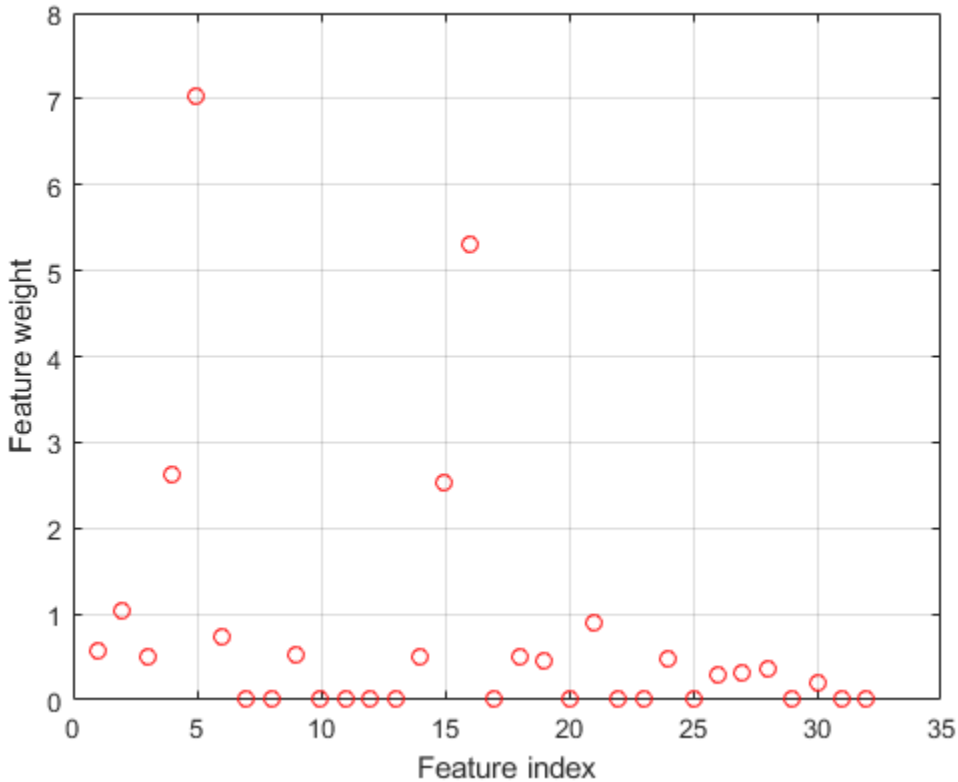
The `robotarm` (`pumadyn32nm`) dataset is created using a robot arm simulator with 7168 training observations and 1024 test observations with 32 features [1][2]. This is a preprocessed version of the original data set. The data are preprocessed by subtracting off a linear regression fit, followed by normalization of all features to unit variance.

Perform neighborhood component analysis (NCA) feature selection for regression with the default  $\lambda$  (regularization parameter) value.

```
nca = fsrnca(Xtrain,ytrain,'FitMethod','exact', ...  
            'Solver','lbfgs');
```

Plot the selected values.

```
figure  
plot(nca.FeatureWeights,'ro')  
xlabel('Feature index')  
ylabel('Feature weight')  
grid on
```



More than half of the feature weights are nonzero. Compute the loss using the test set as a measure of the performance by using the selected features.

```
L = loss(nca, Xtest, ytest)
```

```
L =
```

```
0.0837
```

Try improving the performance. Tune the regularization parameter  $\lambda$  for feature selection using five-fold cross-validation. Tuning  $\lambda$  means finding the  $\lambda$  value that produces the minimum regression loss. To tune  $\lambda$  using cross-validation:

1. Partition the data into five folds. For each fold, `cvpartition` assigns 4/5th of the data as a training set, and 1/5th of the data as a test set.

```
rng(1) % For reproducibility
n = length(ytrain);
cvp = cvpartition(length(ytrain), 'kfold', 5);
numvalidsets = cvp.NumTestSets;
```

Assign the  $\lambda$  values for the search. Create an array to store the loss values.

```
lambdaval = linspace(0, 50, 20) * std(ytrain) / n;
lossvals = zeros(length(lambdaval), numvalidsets);
```

2. Train the NCA model for each  $\lambda$  value, using the training set in each fold.

3. Compute the regression loss for the corresponding test set in the fold using the NCA model. Record the loss value.

4. Repeat this for each  $\lambda$  value and each fold.

```
for i = 1:length(lambdaval)
    for k = 1:numvalidsets
        X = Xtrain(cvp.training(k), :);
        y = ytrain(cvp.training(k), :);
        Xvalid = Xtrain(cvp.test(k), :);
        yvalid = ytrain(cvp.test(k), :);

        nca = fsrnca(X, y, 'FitMethod', 'exact', ...
                    'Solver', 'minibatch-lbfgs', 'Lambda', lambdaval(i), ...
                    'GradientTolerance', 1e-4, 'IterationLimit', 30);

        lossvals(i, k) = loss(nca, Xvalid, yvalid, 'LossFunction', 'mse');
    end
end
```

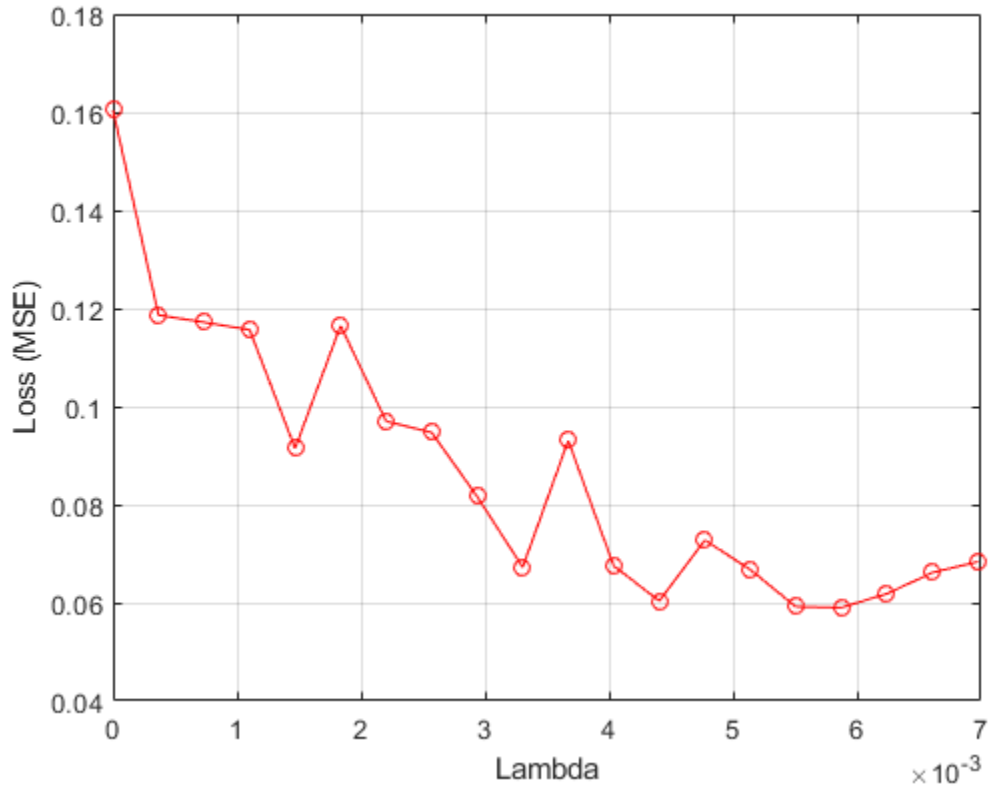
Compute the average loss obtained from the folds for each  $\lambda$  value.

```
meanloss = mean(lossvals, 2);
```

Plot the mean loss versus the  $\lambda$  values.

```
figure
plot(lambdaval, meanloss, 'ro-')
xlabel('Lambda')
```

```
ylabel('Loss (MSE)')  
grid on
```



Find the  $\lambda$  value that gives the minimum loss value.

```
[~,idx] = min(meanloss)  
  
bestlambda = lambdaval(idx)  
bestloss = meanloss(idx)
```

```
idx =
```

```
17
```

```
bestlambda =  
  
    0.0059  
  
bestloss =  
  
    0.0590
```

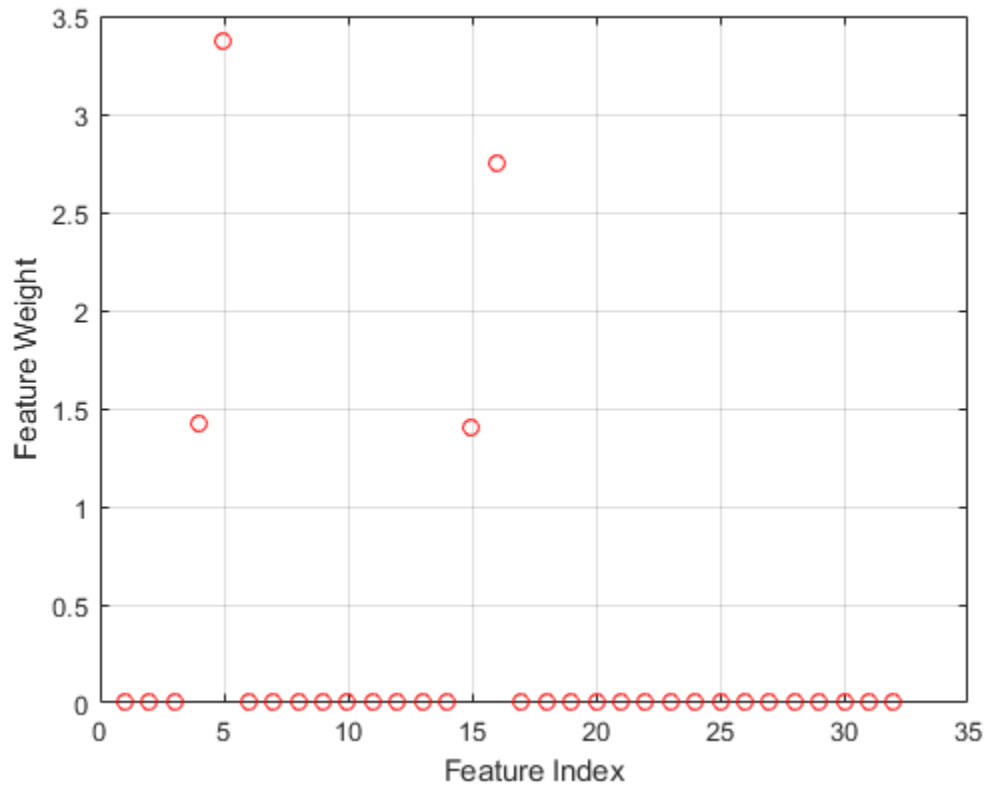
Fit the NCA feature selection model for regression using the best  $\lambda$  value.

```
nca = fsrnca(Xtrain,ytrain,'FitMethod','exact', ...  
            'Solver','lbfgs','Lambda',bestlambda);
```

Plot the selected features.

```
figure  
plot(nca.FeatureWeights,'ro')  
xlabel('Feature Index')  
ylabel('Feature Weight')  
grid on
```





Most of the feature weights are zero. `fsrnca` identifies the four most relevant features.

Compute the loss for the test set.

```
L = loss(nca,Xtest,ytest)
```

```
L =
```

```
0.0571
```

Tuning the regularization parameter,  $\lambda$ , eliminated more of the irrelevant features and improved the performance.

### Compare NCA and ARD Feature Selection

This example uses the Abalone data [3][4] from the UCI Machine Learning Repository [5]. Download the data and save it in your current folder with the name 'abalone.data'.

Store the data into a table. Display the first seven rows.

```
tbl = readtable('abalone.data', 'Filetype', 'text', 'ReadVariableNames', false);
tbl.Properties.VariableNames = {'Sex', 'Length', 'Diameter', 'Height', ...
                               'WWeight', 'SWeight', 'VWeight', 'ShWeight', 'NoShellRings'};
tbl(1:7, :)
```

ans =

Sex	Length	Diameter	Height	WWeight	SWeight	VWeight	ShWeight
'M'	0.455	0.365	0.095	0.514	0.2245	0.101	0.15
'M'	0.35	0.265	0.09	0.2255	0.0995	0.0485	0.07
'F'	0.53	0.42	0.135	0.677	0.2565	0.1415	0.21
'M'	0.44	0.365	0.125	0.516	0.2155	0.114	0.155
'I'	0.33	0.255	0.08	0.205	0.0895	0.0395	0.055
'I'	0.425	0.3	0.095	0.3515	0.141	0.0775	0.12
'F'	0.53	0.415	0.15	0.7775	0.237	0.1415	0.33

The dataset has 4177 observations. The goal is to predict the age of abalone from eight physical measurements. The last variable, the number of shell rings, shows the age of the abalone. The first predictor is a categorical variable. The last variable in the table is the response variable.

Prepare the predictor and response variables for `fsrnca`. The last column of `tbl` contains the number of shell rings, which is the response variable. The first predictor variable, `sex`, is categorical. You must create dummy variables.

```
y = table2array(tbl(:,end));
```

```
X(:,1:3) = dummyvar(categorical(tbl.Sex));
X = [X,table2array(tbl(:,2:end-1))];
```

Use four-fold cross-validation to tune the regularization parameter in the NCA model. First partition the data into four folds.

```
n = length(y);
cvp = cvpartition(n, 'kfold', 4);
numtestsets = cvp.NumTestSets;
```

`cvpartition` divides the data into four partitions (folds). In each fold, about three-fourths of the data is assigned as a training set and one-fourth is assigned as a test set.

Generate a variety of  $\lambda$  (regularization parameter) values for fitting the model to determine the best  $\lambda$  value. Create a vector to collect the loss values from each fit.

```
lambdaval = linspace(0,25,20)*std(y)/n;
lossvals = zeros(length(lambdaval),numtestsets);
```

The rows of `lossvals` corresponds to the  $\lambda$  values and the columns correspond to the folds.

Fit the NCA model for regression using `fsrnca` to the data from each fold using each  $\lambda$  value. Compute the loss for each model using the test data from each fold.

```
for i = 1:length(lambdaval)
    for k = 1:numtestsets
        Xtrain = X(cvp.training(k),:);
        ytrain = y(cvp.training(k),:);
        Xtest = X(cvp.test(k),:);
        ytest = y(cvp.test(k),:);

        nca = fsrnca(Xtrain,ytrain,'FitMethod','exact', ...
                    'Solver','lbfgs','Lambda',lambdaval(i),'Standardize',true);

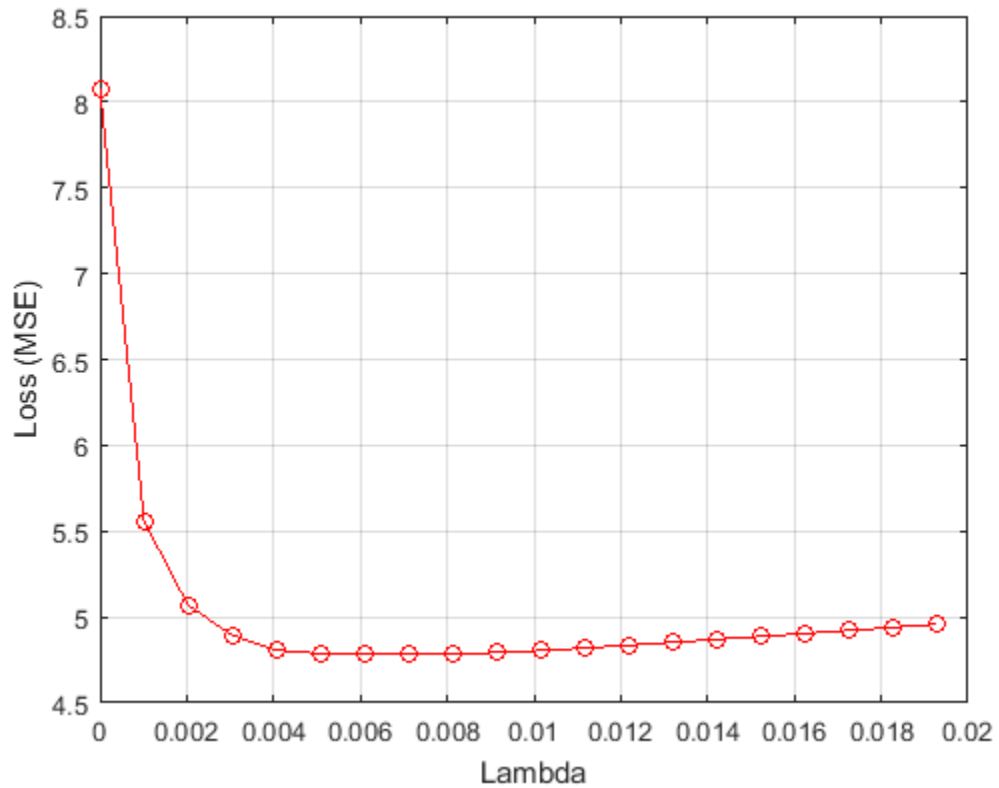
        lossvals(i,k) = loss(nca,Xtest,ytest,'LossFunction','mse');
    end
end
```

Compute the average loss for the folds, that is, compute the mean in the second dimension of `lossvals`).

```
meanloss = mean(lossvals,2);
```

Plot the  $\lambda$  values versus the mean loss from the four folds.

```
figure
plot(lambdaval,meanloss,'ro-')
xlabel('Lambda')
ylabel('Loss (MSE)')
grid on
```



Find the  $\lambda$  value that minimizes the mean loss.

```
[~,idx] = min(meanloss);
bestlambda = lambdaval(idx)
```

```
bestlambda =
```

```
0.0071
```

Compute the best loss value.

```
bestloss = meanloss(idx)
```

```
bestloss =
```

```
4.7799
```

Fit the NCA model on all of the data using the best  $\lambda$  value.

```
nca = fsrnca(X,y,'FitMethod','exact','Solver','lbfgs', ...
             'Verbose',1,'Lambda',bestlambda,'Standardize',true);
```

```
o Solver = LBFGS, HessianHistorySize = 15, LineSearchMethod = weakwolfe
```

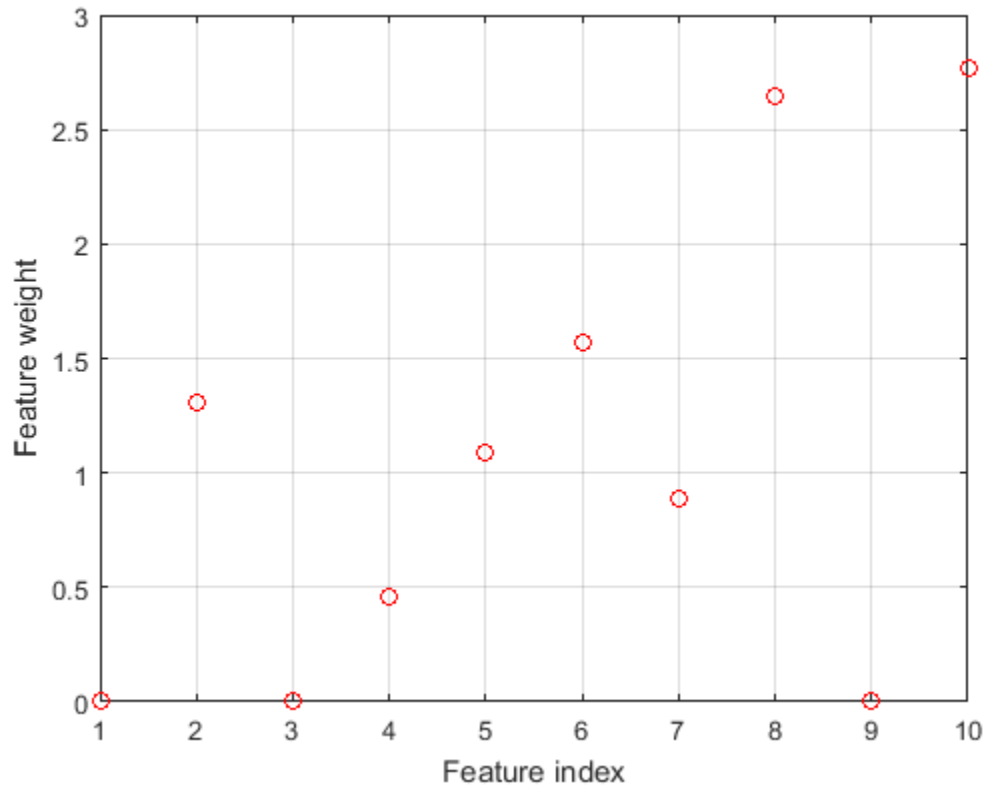
```
=====
| ITER | FUN VALUE | NORM GRAD | NORM STEP | CURV | GAMMA | ALPH |
|=====|=====|=====|=====|=====|=====|=====|
| 0 | 2.469168e+00 | 1.266e-01 | 0.000e+00 | | 4.741e+00 | 0.000 |
| 1 | 2.375166e+00 | 8.265e-02 | 7.268e-01 | OK | 1.054e+01 | 1.000 |
| 2 | 2.293528e+00 | 2.067e-02 | 2.034e+00 | OK | 1.569e+01 | 1.000 |
| 3 | 2.286703e+00 | 1.031e-02 | 3.158e-01 | OK | 2.213e+01 | 1.000 |
| 4 | 2.279928e+00 | 2.023e-02 | 9.374e-01 | OK | 1.953e+01 | 1.000 |
| 5 | 2.276258e+00 | 6.884e-03 | 2.497e-01 | OK | 1.439e+01 | 1.000 |
| 6 | 2.274358e+00 | 1.792e-03 | 4.010e-01 | OK | 3.109e+01 | 1.000 |
| 7 | 2.274105e+00 | 2.412e-03 | 2.399e-01 | OK | 3.557e+01 | 1.000 |
| 8 | 2.274073e+00 | 1.459e-03 | 7.684e-02 | OK | 1.356e+01 | 1.000 |
| 9 | 2.274050e+00 | 3.733e-04 | 3.797e-02 | OK | 1.725e+01 | 1.000 |
| 10 | 2.274043e+00 | 2.750e-04 | 1.379e-02 | OK | 2.445e+01 | 1.000 |
| 11 | 2.274027e+00 | 2.682e-04 | 5.701e-02 | OK | 7.386e+01 | 1.000 |
| 12 | 2.274020e+00 | 1.712e-04 | 4.107e-02 | OK | 9.461e+01 | 1.000 |
| 13 | 2.274014e+00 | 2.633e-04 | 6.720e-02 | OK | 7.469e+01 | 1.000 |
| 14 | 2.274012e+00 | 9.818e-05 | 2.263e-02 | OK | 3.275e+01 | 1.000 |
| 15 | 2.274012e+00 | 4.220e-05 | 6.188e-03 | OK | 2.799e+01 | 1.000 |
| 16 | 2.274012e+00 | 2.859e-05 | 4.979e-03 | OK | 6.628e+01 | 1.000 |
| 17 | 2.274011e+00 | 1.582e-05 | 6.767e-03 | OK | 1.439e+02 | 1.000 |
| 18 | 2.274011e+00 | 7.623e-06 | 4.311e-03 | OK | 1.211e+02 | 1.000 |
| 19 | 2.274011e+00 | 3.038e-06 | 2.528e-04 | OK | 1.798e+01 | 5.000 |
|=====|=====|=====|=====|=====|=====|=====|
| ITER | FUN VALUE | NORM GRAD | NORM STEP | CURV | GAMMA | ALPH |
|=====|=====|=====|=====|=====|=====|=====|
| 20 | 2.274011e+00 | 6.710e-07 | 2.325e-04 | OK | 2.721e+01 | 1.000
```

```
Infinity norm of the final gradient = 6.710e-07
```

```
Two norm of the final step = 2.325e-04, TolX = 1.000e-06  
Relative infinity norm of the final gradient = 6.710e-07, TolFun = 1.000e-06  
EXIT: Local minimum found.
```

Plot the selected features.

```
figure  
plot(nca.FeatureWeights,'ro')  
xlabel('Feature Index')  
ylabel('Feature Weight')  
grid on
```



The irrelevant features have zero weights. According to this figure, the features 1, 3, and 9 are not selected.

Fit a Gaussian process regression (GPR) model using the subset of regressors method for parameter estimation and the fully independent conditional method for prediction. Use the ARD squared exponential kernel function, which assigns an individual weight to each predictor. Standardize the predictors.

```
gprMdl = fitrgp(tbl, 'NoShellRings', 'KernelFunction', 'ardsquaredexponential', ...
    'FitMethod', 'sr', 'PredictMethod', 'fic', 'Standardize', true)
```

```
gprMdl =
```

```
RegressionGP
    PredictorNames: {1×8 cell}
    ResponseName: 'NoShellRings'
CategoricalPredictors: 1
    ResponseTransform: 'none'
    NumObservations: 4177
    KernelFunction: 'ARDSquaredExponential'
KernelInformation: [1×1 struct]
    BasisFunction: 'Constant'
        Beta: 10.6974
        Sigma: 2.0240
    PredictorLocation: [10×1 double]
    PredictorScale: [10×1 double]
        Alpha: [1000×1 double]
    ActiveSetVectors: [1000×10 double]
    PredictMethod: 'FIC'
    ActiveSetSize: 1000
    FitMethod: 'SR'
    ActiveSetMethod: 'Random'
IsActiveSetVector: [4177×1 logical]
    LogLikelihood: -8.9984e+03
    ActiveSetHistory: [1×1 struct]
    BCDInformation: []
```

Compute the regression loss on the training data (resubstitution loss) for the trained model.

```
L = resubLoss(gprMdl)
```

```
L =
```

```
4.0064
```

The smallest cross-validated loss using `fsrcna` is comparable to the loss obtained using a GPR model with an ARD kernel.

- “Robust Feature Selection Using NCA for Regression” on page 15-107

## Input Arguments

### **x** — Predictor variable values

*n*-by-*p* matrix

Predictor variable values, specified as an *n*-by-*p* matrix, where *n* is the number of observations and *p* is the number of predictor variables.

Data Types: `single` | `double`

### **y** — Response values

numeric real vector of length *n*

Response values, specified as a numeric real vector of length *n*, where *n* is the number of observations.

Data Types: `single` | `double`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'Solver', 'sgd', 'Weights', W, 'Lambda', 0.0003` specifies the solver as the stochastic gradient descent, the observation weights as the values in the vector `W`, and sets the regularization parameter at 0.0003.

### Fitting Options

#### **FitMethod** — Method for fitting the model

`'exact'` (default) | `'none'` | `'average'`

Method for fitting the model, specified as the comma-separated pair consisting of `'FitMethod'` and one of the following:



- 'exact' — Performs fitting using all of the data.
- 'none' — No fitting. Use this option to evaluate the generalization error of the NCA model using the initial feature weights supplied in the call to `fsrnca`.
- 'average' — Divides the data into partitions (subsets), fits each partition using the `exact` method, and returns the average of the feature weights. You can specify the number of partitions using the `NumPartitions` name-value pair argument.

Example: `'FitMethod','none'`

Data Types: `char`

#### **NumPartitions** — Number of partitions

`max(2,min(10,n))` (default) | integer between 2 and  $n$

Number of partitions to split the data for using with `'FitMethod','average'` option, specified as the comma-separated pair consisting of `'NumPartitions'` and an integer value between 2 and  $n$ , where  $n$  is the number of observations.

Example: `'NumPartitions',15`

Data Types: `double` | `single`

#### **Lambda** — Regularization parameter

$1/n$  (default) | nonnegative scalar

Regularization parameter, specified as the comma-separated pair consisting of `'Lambda'` and a nonnegative scalar.

For  $n$  observations, the best `Lambda` value that minimizes the generalization error of the NCA model is expected to be a multiple of  $1/n$ .

Example: `'Lambda',0.002`

Data Types: `double` | `single`

#### **LengthScale** — Width of the kernel

1 (default) | positive real scalar

Width of the kernel, specified as the comma-separated pair consisting of `'LengthScale'` and a positive real scalar.

A length scale value of 1 is sensible when all predictors are on the same scale. If the predictors in  $X$  are of very different magnitudes, then consider standardizing the predictor values using 'Standardize', true and setting 'LengthScale', 1.

Example: 'LengthScale', 1.5

Data Types: double | single

### **InitialFeatureWeights** — Initial feature weights

ones( $p$ , 1) (default) |  $p$ -by-1 vector of real positive scalars

Initial feature weights, specified as the comma-separated pair consisting of 'InitialFeatureWeights' and a  $p$ -by-1 vector of real positive scalars, where  $p$  is the number of predictors in the training data.

The regularized objective function for optimizing feature weights is nonconvex. As a result, using different initial feature weights can give different results. Setting all initial feature weights to 1 generally works well, but in some cases, random initialization using rand( $p$ , 1) can give better quality solutions.

Data Types: double | single

### **Weights** — Observation weights

$n$ -by-1 vector of 1s (default) |  $n$ -by-1 vector of real positive scalars

Observation weights, specified as the comma-separated pair consisting of 'ObservationWeights' and an  $n$ -by-1 vector of real positive scalars. Use observation weights to specify higher importance of some observations compared to others. The default weights assign equal importance to all observations.

Data Types: double | single

### **Standardize** — Indicator for standardizing predictor data

false (default) | true

Indicator for standardizing the predictor data, specified as the comma-separated pair consisting of 'Standardize' and either false or true. For more information, see “Impact of Standardization” on page 15-130.

Example: 'Standardize', true

Data Types: logical

**Verbose** — Verbosity level indicator

0 (default) | 1 | &gt;1

Verbosity level indicator for the convergence summary display, specified as the comma-separated pair consisting of 'Verbose' and one of the following:

- 0 — No convergence summary
- 1 — Convergence summary, including norm of gradient and objective function values
- > 1 — More convergence information, depending on the fitting algorithm

When using 'minibatch-lbfgs' solver and verbosity level > 1, the convergence information includes iteration the log from intermediate mini-batch LBFGS fits.

Example: 'Verbose',1

Data Types: double | single

**Solver** — Solver type

'lbfgs' | 'sgd' | 'minibatch-lbfgs'

Solver type for estimating feature weights, specified as the comma-separated pair consisting of 'Solver' and one of the following:

- 'lbfgs' — Limited memory Broyden-Fletcher-Goldfarb-Shanno (LBFGS) algorithm
- 'sgd' — Stochastic gradient descent (SGD) algorithm
- 'minibatch-lbfgs' — Stochastic gradient descent with LBFGS algorithm applied to mini-batches

Default is 'lbfgs' for  $n \leq 1000$ , and 'sgd' for  $n > 1000$ .

Example: 'solver', 'minibatch-lbfgs'

Data Types: char

**LossFunction** — Loss function

'mad' (default) | 'mse' | 'epsiloninsensitive' | function handle

Loss function, specified as the comma-separated pair consisting of 'LossFunction' and one of the following:

- 'mad' — Mean absolute deviation, defined as

$$\frac{1}{n} \sum_{i=1}^n |y_i - y_j|,$$

where  $y_i$  is the true response for  $\mathbf{x}_i$ , and  $t_i$  is the predicted response using the NCA regression model.

- 'mse' — Mean squared error, defined as

$$\frac{1}{n} \sum_{i=1}^n (y_i - y_j)^2.$$

- 'epsiloninsensitive' —  $\epsilon$ -insensitive loss function, defined as

$$l(y_i, y_j) = \max(0, |y_i - y_j| - \epsilon).$$

This loss function is more robust to outliers than mean squared error or mean absolute deviation.

- Function handle — Custom loss function. For example for the 'LossFunction', L name-value pair, L must have the following form:

```
L = lossFcn(Yn, Ym)
```

$Y_n$  is an  $n$ -by-1 vector and  $Y_m$  is an  $m$ -by-1 vector.  $L$  is an  $n$ -by- $m$  matrix of loss values such that  $L(i, j)$  is the loss value for  $Y_n(i)$  and  $Y_m(j)$ .

Example: 'LossFunction', 'mse'

Data Types: char | function\_handle

### **Epsilon** — Epsilon value

iqr(Y)/13.49 (default) | nonnegative real scalar

Epsilon value for the 'LossFunction', 'epsiloninsensitive' option, specified as the comma-separated pair consisting of 'LossFunction' and a nonnegative real scalar. The default value is an estimate of the sample standard deviation using the interquartile range of the response variable.

Example: 'Epsilon', 0.1

Data Types: double | single

### **CacheSize** — Memory size

1000MB (default) | integer

Memory size, in MB, to use for objective function and gradient computation, specified as the comma-separated pair consisting of 'CacheSize' and an integer.

Example: 'CacheSize',1500MB

Data Types: double | single

### **LBFGS Options**

#### **HessianHistorySize — Size of history buffer for Hessian approximation**

15 (default) | positive integer

Size of history buffer for Hessian approximation for the 'lbfgs' solver, specified as the comma-separated pair consisting of 'HessianHistorySize' and a positive integer. At each iteration the function uses the most recent HessianHistorySize iterations to build an approximation to the inverse Hessian.

Example: 'HessianHistorySize',20

Data Types: double | single

#### **InitialStepSize — Initial step size**

'auto' (default) | positive real scalar

Initial step size for the 'lbfgs' solver, specified as the comma-separated pair consisting of 'InitialStepSize' and a positive real scalar. By default, the function determines the initial step size automatically.

Data Types: double | single

#### **LineSearchMethod — Line search method**

'weakwolfe' (default) | 'strongwolfe' | 'backtracking'

Line search method, specified as the comma-separated pair consisting of 'LineSearchMethod' and one of the following:

- 'weakwolfe' — Weak Wolfe line search
- 'strongwolfe' — Strong Wolfe line search
- 'backtracking' — Backtracking line search

Example: 'LineSearchMethod','backtracking'

Data Types: char

**MaxLineSearchIterations** — Maximum number of line search iterations

20 (default) | positive integer

Maximum number of line search iterations, specified as the comma-separated pair consisting of 'MaxLineSearchIterations' and a positive integer.

Example: 'MaxLineSearchIterations',25

Data Types: double | single

**GradientTolerance** — Relative convergence tolerance

1e-6 (default) | positive real scalar

Relative convergence tolerance on the gradient norm for solver `lbfgs`, specified as the comma-separated pair consisting of 'GradientTolerance' and a positive real scalar.

Example: 'GradientTolerance',0.000002

Data Types: double | single

**SGD Options****InitialLearningRate** — Initial learning rate for 'sgd' solver

'auto' (default) | positive real scalar

Initial learning rate for the 'sgd' solver, specified as the comma-separated pair consisting of 'InitialLearningRate' and a positive real scalar.

When using solver type 'sgd', the learning rate decays over iterations starting with the value specified for 'InitialLearningRate'.

The default 'auto' means that the initial learning rate is determined using experiments on small subsets of data. Use the `NumTuningIterations` name-value pair argument to specify the number of iterations for automatically tuning the initial learning rate. Use the `TuningSubsetSize` name-value pair argument to specify the number of observations to use for automatically tuning the initial learning rate.

For solver type 'minibatch-lbfgs', you can set 'InitialLearningRate' to a very high value. In this case, the function applies LBFGS to each mini-batch separately with initial feature weights from the previous mini-batch.

To make sure the chosen initial learning rate decreases the objective value with each iteration, plot the `Iteration` versus the `Objective` values saved in the `mdl.FitInfo` property.

You can use the `refit` method with `'InitialFeatureWeights'` equal to `mdl.FeatureWeights` to start from the current solution and run additional iterations

Example: `'InitialLearningRate',0.9`

Data Types: `double` | `single`

**MiniBatchSize** — Number of observations to use in each batch for the `'sgd'` solver

`min(10,n)` (default) | positive integer value from 1 to  $n$

Number of observations to use in each batch for the `'sgd'` solver, specified as the comma-separated pair consisting of `'MiniBatchSize'` and a positive integer from 1 to  $n$ .

Example: `'MiniBatchSize',25`

Data Types: `double` | `single`

**PassLimit** — Maximum number of passes for solver `'sgd'`

5 (default) | positive integer

Maximum number of passes through all  $n$  observations for solver `'sgd'`, specified as the comma-separated pair consisting of `'PassLimit'` and a positive integer. Each pass through all of the data is called an epoch.

Example: `'PassLimit',10`

Data Types: `double` | `single`

**NumPrint** — Frequency of batches for displaying convergence summary

10 (default) | positive integer value

Frequency of batches for displaying convergence summary for the `'sgd'` solver, specified as the comma-separated pair consisting of `'NumPrint'` and a positive integer. This argument applies when the `'Verbose'` value is greater than 0. `NumPrint` mini-batches are processed for each line of the convergence summary that is displayed on the command line.

Example: `'NumPrint',5`

Data Types: double | single

**NumTuningIterations — Number of tuning iterations**

20 (default) | positive integer

Number of tuning iterations for the 'sgd' solver, specified as the comma-separated pair consisting of 'NumTuningIterations' and a positive integer. This option is valid only for 'InitialLearningRate', 'auto'.

Example: 'NumTuningIterations',15

Data Types: double | single

**TuningSubsetSize — Number of observations to use for tuning initial learning rate**

min(100, $n$ ) (default) | positive integer value from 1 to  $n$

Number of observations to use for tuning the initial learning rate, specified as the comma-separated pair consisting of 'TuningSubsetSize' and a positive integer value from 1 to  $n$ . This option is valid only for 'InitialLearningRate', 'auto'.

Example: 'TuningSubsetSize',25

Data Types: double | single

**SGD or LBFGS Options****IterationLimit — Maximum number of iterations**

positive integer

Maximum number of iterations, specified as the comma-separated pair consisting of 'IterationLimit' and a positive integer. The default is 10000 for SGD and 1000 for LBFGS and mini-batch LBFGS.

Each pass through a batch is an iteration. Each pass through all of the data is an epoch. If the data is divided into  $k$  mini-batches, then every epoch is equivalent to  $k$  iterations.

Example: 'IterationLimit',250

Data Types: double | single

**StepTolerance — Convergence tolerance on the step size**

1e-6 (default) | positive real scalar



Convergence tolerance on the step size, specified as the comma-separated pair consisting of 'StepTolerance' and a positive real scalar. The 'lbfgs' solver uses an absolute step tolerance, and the 'sgd' solver uses a relative step tolerance.

Example: 'StepTolerance',0.000005

Data Types: double | single

### Mini-batch LBFGS Options

**MiniBatchLBFGSIterations** — Maximum number of iterations per mini-batch LBFGS step

10 (default) | positive integer

Maximum number of iterations per mini-batch LBFGS step, specified as the comma-separated pair consisting of 'MiniBatchLBFGSIterations' and a positive integer.

Example: 'MiniBatchLBFGSIterations',15

**Mini-batch LBFGS algorithm is a combination of SGD and LBFGS methods. Therefore, all of the name-value pair arguments that apply to SGD and LBFGS solvers also apply to the mini-batch LBFGS algorithm.**

Data Types: double | single

## Output Arguments

**mdl** — Neighborhood component analysis model for regression

FeatureSelectionNCARegression object

Neighborhood component analysis model for regression, returned as a FeatureSelectionNCARegression object.

## References

- [1] Rasmussen, C. E., R. M. Neal, G. E. Hinton, D. van Campand, M. Revow, Z. Ghahramani, R. Kustra, R. Tibshirani. The DELVE Manual, 1996, <http://mlg.eng.cam.ac.uk/pub/pdf/RasNeaHinetal96.pdf>.
- [2] University of Toronto, Computer Science Department. Delve Datasets. <http://www.cs.toronto.edu/~delve/data/datasets.html>.

- [3] Warwick J. N., T. L. Sellers, S. R. Talbot, A. J. Cawthorn, and W. B. Ford. "The Population Biology of Abalone (*Haliotis* species) in Tasmania. I. Blacklip Abalone (*H. rubra*) from the North Coast and Islands of Bass Strait." Sea Fisheries Division, Technical Report No. 48 (ISSN 1034-3288), 1994.
- [4] S. Waugh. "Extending and Benchmarking Cascade-Correlation", PhD Thesis. Computer Science Department, University of Tasmania, 1995.
- [5] Lichman, M. UCI Machine Learning Repository. Irvine, CA: University of California, School of Information and Computer Science, 2013. <http://archive.ics.uci.edu/ml>.

## See Also

`FeatureSelectionNCARegression` | `loss` | `predict` | `refit`

## Topics

“Robust Feature Selection Using NCA for Regression” on page 15-107

“Neighborhood Component Analysis (NCA) Feature Selection” on page 15-126

**Introduced in R2016b**

# fstat

*F* mean and variance

## Syntax

`[M,V] = fstat(V1,V2)`

## Description

`[M,V] = fstat(V1,V2)` returns the mean of and variance for the *F* distribution with numerator degrees of freedom `V1` and denominator degrees of freedom `V2`. `V1` and `V2` can be vectors, matrices, or multidimensional arrays that all have the same size, which is also the size of `M` and `V`. A scalar input for `V1` or `V2` is expanded to a constant arrays with the same dimensions as the other input. `V1` and `V2` parameters must contain real positive values.

The mean of the *F* distribution for values of  $\nu_2$  greater than 2 is

$$\frac{\nu_2}{\nu_2 - 2}$$

The variance of the *F* distribution for values of  $\nu_2$  greater than 4 is

$$\frac{2\nu_2^2(\nu_1 + \nu_2 - 2)}{\nu_1(\nu_2 - 2)^2(\nu_2 - 4)}$$

The mean of the *F* distribution is undefined if  $\nu_2$  is less than 3. The variance is undefined for  $\nu_2$  less than 5.

## Examples

`fstat` returns NaN when the mean and variance are undefined.

```
[m,v] = fstat(1:5,1:5)
m =
```

```
NaN NaN 3.0000 2.0000 1.6667
v =
NaN NaN NaN NaN 8.8889
```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`fcdf` | `finv` | `fpdf` | `frnd`

### Topics

“F Distribution” on page B-44

Introduced before R2006a

# fsurfht

Interactive contour plot

## Syntax

```
fsurfht(fun,xlims,ylim)  
fsurfht(fun,xlims,ylim,p1,p2,p3,p4,p5)
```

## Description

`fsurfht(fun,xlims,ylim)` is an interactive contour plot of the function specified by the text variable `fun`. The  $x$ -axis limits are specified by `xlims` in the form `[xmin xmax]`, and the  $y$ -axis limits are specified by `ylim` in the form `[ymin ymax]`.

`fsurfht(fun,xlims,ylim,p1,p2,p3,p4,p5)` allows for five optional parameters that you can supply to the function `fun`.

The intersection of the vertical and horizontal reference lines on the plot defines the current  $x$  value and  $y$  value. You can drag these reference lines and watch the calculated  $z$ -values (at the top of the plot) update simultaneously. Alternatively, you can type the  $x$  value and  $y$  value into editable text fields on the  $x$ -axis and  $y$ -axis.

## Examples

Plot the Gaussian likelihood function for the `gas.mat` data.

```
load gas
```

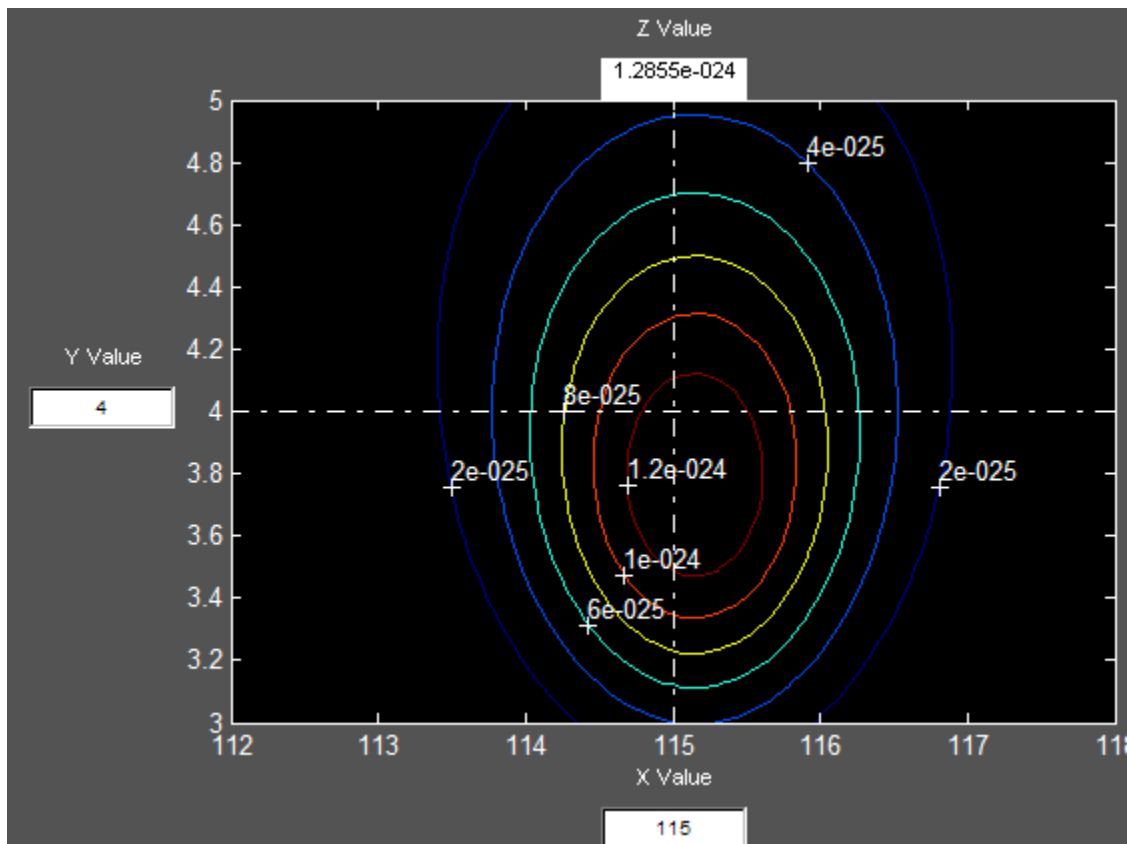
Create a function containing the following commands, and name it `gauslike.m`.

```
function z = gauslike(mu,sigma,p1)  
n = length(p1);  
z = ones(size(mu));  
for i = 1:n
```

```
z = z .* (normpdf(p1(i),mu,sigma));
end
```

The `gauslike` function calls `normpdf`, treating the data sample as fixed and the parameters  $\mu$  and  $\sigma$  as variables. Assume that the gas prices are normally distributed, and plot the likelihood surface of the sample.

```
fsurfht('gauslike',[112 118],[3 5],price1)
```



The sample mean is the  $x$  value at the maximum, but the sample standard deviation is *not* the  $y$  value at the maximum.

```
mumax = mean(price1)
mumax =
```

```
115.1500
sigmamax = std(pricel)*sqrt(19/20)
sigmamax =
  3.7719
```

**Introduced before R2006a**

## fullfact

Full factorial design

### Syntax

```
dFF = fullfact(levels)
```

### Description

`dFF = fullfact(levels)` gives factor settings `dFF` for a full factorial design with  $n$  factors, where the number of levels for each factor is given by the vector `levels` of length  $n$ . `dFF` is  $m$ -by- $n$ , where  $m$  is the number of treatments in the full-factorial design. Each row of `dFF` corresponds to a single treatment. Each column contains the settings for a single factor, with integer values from one to the number of levels.

### Examples

The following generates an eight-run full-factorial design with two levels in the first factor and four levels in the second factor:

```
dFF = fullfact([2 4])  
dFF =  
  1  1  
  2  1  
  1  2  
  2  2  
  1  3  
  2  3  
  1  4  
  2  4
```

### See Also

`ff2n`



**Introduced before R2006a**

## gagerr

Gage repeatability and reproducibility study

### Syntax

```
gagerr(y, {part, operator})  
gagerr(y, GROUP)  
gagerr(y, part)  
gagerr(..., param1, val1, param2, val2, ...)  
[TABLE, stats] = gagerr(...)
```

### Description

`gagerr(y, {part, operator})` performs a gage repeatability and reproducibility study on measurements in `y` collected by `operator` on `part`. `y` is a column vector containing the measurements on different parts. `part` and `operator` are categorical variables, numeric vectors, character matrices, or cell arrays of character vectors. The number of elements in `part` and `operator` should be the same as in `y`.

`gagerr` prints a table in the command window in which the decomposition of variance, standard deviation, study var (5.15 × standard deviation) are listed with respective percentages for different sources. Summary statistics are printed below the table giving the number of distinct categories (NDC) and the percentage of Gage R&R of total variations (PRR).

`gagerr` also plots a bar graph showing the percentage of different components of variations. Gage R&R, repeatability, reproducibility, and part-to-part variations are plotted as four vertical bars. Variance and study var are plotted as two groups.

To determine the capability of a measurement system using NDC, use the following guidelines:

- If  $NDC > 5$ , the measurement system is capable.
- If  $NDC < 2$ , the measurement system is not capable.

- Otherwise, the measurement system may be acceptable.

To determine the capability of a measurement system using PRR, use the following guidelines:

- If  $PRR < 10\%$ , the measurement system is capable.
- If  $PRR > 30\%$ , the measurement system is not capable.
- Otherwise, the measurement system may be acceptable.

`gagerr(y, GROUP)` performs a gage R&R study on measurements in `y` with part and operator represented in `GROUP`. `GROUP` is a numeric matrix whose first and second columns specify different parts and operators, respectively. The number of rows in `GROUP` should be the same as the number of elements in `y`.

`gagerr(y, part)` performs a gage R&R study on measurements in `y` without operator information. The assumption is that all variability is contributed by `part`.

`gagerr(..., param1, val1, param2, val2, ...)` performs a gage R&R study using one or more of the following parameter name/value pairs:

- `'spec'` — A two-element vector that defines the lower and upper limit of the process, respectively. In this case, summary statistics printed in the command window include Precision-to-Tolerance Ratio (PTR). Also, the bar graph includes an additional group, the percentage of tolerance.

To determine the capability of a measurement system using PTR, use the following guidelines:

- If  $PTR < 0.1$ , the measurement system is capable.
- If  $PTR > 0.3$ , the measurement system is not capable.
- Otherwise, the measurement system may be acceptable.
- `'printtable'` — A value `'on'` or `'off'` that indicates whether the tabular output should be printed in the command window or not. The default value is `'on'`.
- `'printgraph'` — A value `'on'` or `'off'` that indicates whether the bar graph should be plotted or not. The default value is `'on'`.
- `'randomoperator'` — A logical value, `true` or `false`, that indicates whether the effect of operator is random or not. The default value is `true`.
- `'model'` — The model to use, specified by one of:

- 'linear' — Main effects only (default)
- 'interaction' — Main effects plus two-factor interactions
- 'nested' — Nest operator in part

The default value is 'linear'.

[TABLE, stats] = gagerr(...) returns a 6-by-5 matrix TABLE and a structure stats. The columns of TABLE, from left to right, represent variance, percentage of variance, standard deviations, study var, and percentage of study var. The rows of TABLE, from top to bottom, represent different sources of variations: gage R&R, repeatability, reproducibility, operator, operator and part interactions, and part. stats is a structure containing summary statistics for the performance of the measurement system. The fields of stats are:

- ndc — Number of distinct categories
- prr — Percentage of gage R&R of total variations
- ptr — Precision-to-tolerance ratio. The value is NaN if the parameter 'spec' is not given.

## Examples

### Gage R&R Study

Simulate a measurement system by randomly generating the operators, parts, and the measurements, `y`, operators do on the parts.

```
rng(1234, 'twister')           % for reproducibility
y = randn(100,1);              % measurements
part = ceil(3*rand(100,1));     % parts
operator = ceil(4*rand(100,1)); % operators
```

Conduct a gage R&R study for this system using a mixed ANOVA model without interactions.

```
gagerr(y, {part, operator}, 'randomoperator', true)
```

```
Columns 1 through 4
```

'Source'	'Variance'	'% Variance'	'sigma'
'Gage R&R'	[ 0.9715]	[ 99.2653]	[0.9857]
' Repeatability'	[ 0.9535]	[ 97.4201]	[0.9765]
' Reproducibility'	[ 0.0181]	[ 1.8452]	[0.1344]
' Operator'	[ 0.0181]	[ 1.8452]	[0.1344]
'Part'	[ 0.0072]	[ 0.7347]	[0.0848]
'Total'	[ 0.9787]	[ 100]	[0.9893]

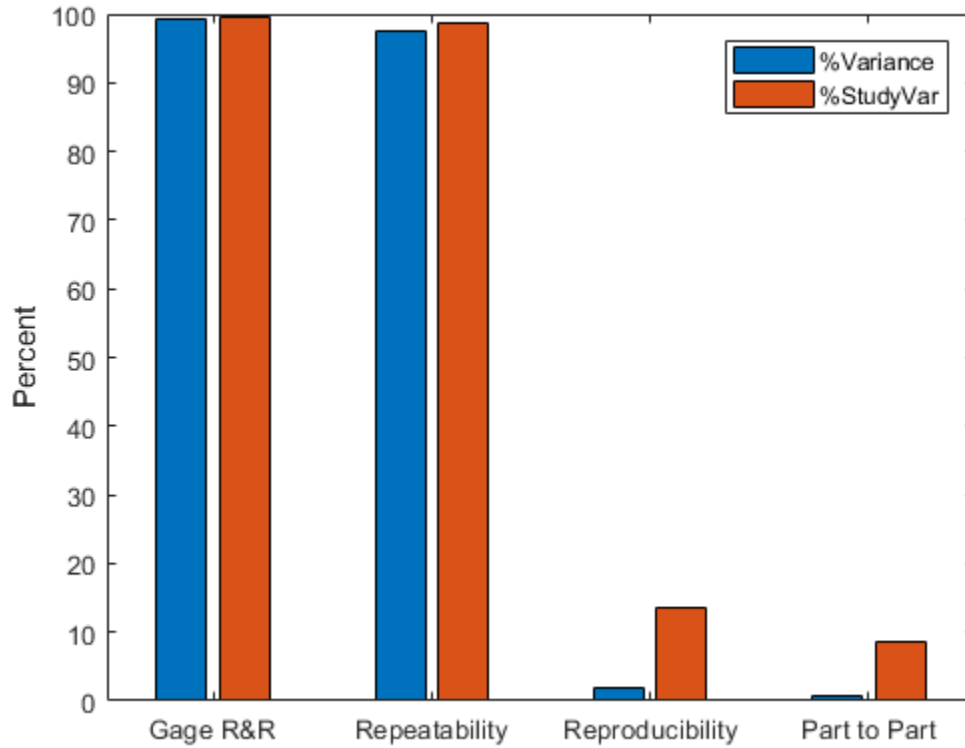
Columns 5 through 6

'5.15*sigma'	'% 5.15*sigma'
[ 5.0762]	[ 99.6320]
[ 5.0288]	[ 98.7016]
[ 0.6921]	[ 13.5838]
[ 0.6921]	[ 13.5838]
[ 0.4367]	[ 8.5716]
[ 5.0949]	''

Number of distinct categories (NDC):0

% of Gage R&R of total variations (PRR): 99.63

Note: The last column of the above table does not have to sum to 100%



## See Also

### Topics

“Grouping Variables” on page 2-59

Introduced in R2006b

# gamcdf

Gamma cumulative distribution function

## Syntax

```
gamcdf(x, a, b)
[p, plo, pup] = gamcdf(x, a, b, pcov, alpha)
[p, plo, pup] = gamcdf( ____, 'upper')
```

## Description

`gamcdf(x, a, b)` returns the gamma cdf at each of the values in `x` using the corresponding shape parameters in `a` and scale parameters in `b`. `x`, `a`, and `b` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array with the same dimensions as the other inputs. The parameters in `a` and `b` must be positive, and the values in `x` must lie on the interval `[0 Inf]`.

`[p, plo, pup] = gamcdf(x, a, b, pcov, alpha)` produces confidence bounds for `p` when the input parameters `a` and `b` are estimates. `pcov` is a 2-by-2 matrix containing the covariance matrix of the estimated parameters. `alpha` has a default value of 0.05, and specifies 100(1-alpha)% confidence bounds. `plo` and `pup` are arrays of the same size as `p` containing the lower and upper confidence bounds.

`[p, plo, pup] = gamcdf( ____, 'upper')` returns the complement of the gamma cdf at each value in `x`, using an algorithm that more accurately computes the extreme upper tail probabilities. You can use the 'upper' argument with any of the previous syntaxes.

The gamma cdf is

$$p = F(x | a, b) = \frac{1}{b^a \Gamma(a)} \int_0^x t^{a-1} e^{-\frac{t}{b}} dt$$

The result, `p`, is the probability that a single observation from a gamma distribution with parameters `a` and `b` will fall in the interval `[0 x]`.

`gammainc` is the gamma distribution with  $b$  fixed at 1.

## Examples

### Compute Gamma Distribution CDF

The mean of the gamma distribution is the product of the parameters,  $ab$ . In this example, the mean approaches the median as it increases (i.e., the distribution becomes more symmetric).

```
a = 1:6;  
b = 5:10;  
prob = gamcdf(a.*b,a,b)
```

```
prob =
```

```
    0.6321    0.5940    0.5768    0.5665    0.5595    0.5543
```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`cdf` | `gamfit` | `gaminv` | `gamlike` | `gamma` | `gampdf` | `gamrnd` | `gamstat`

### Topics

“Gamma Distribution” on page B-47

Introduced before R2006a



# gamfit

Gamma parameter estimates

## Syntax

```
phat = gamfit(data)
[phat,pci] = gamfit(data)
[phat,pci] = gamfit(data,alpha)
[...] = gamfit(data,alpha,censoring,freq,options)
```

## Description

`phat = gamfit(data)` returns the maximum likelihood estimates (MLEs) for the parameters of the gamma distribution given the data in vector `data`.

`[phat,pci] = gamfit(data)` returns MLEs and 95% percent confidence intervals. The first row of `pci` is the lower bound of the confidence intervals; the last row is the upper bound.

`[phat,pci] = gamfit(data,alpha)` returns 100(1 - alpha)% confidence intervals. For example, `alpha = 0.01` yields 99% confidence intervals.

`[...] = gamfit(data,alpha,censoring)` accepts a Boolean vector of the same size as `data` that is 1 for observations that are right-censored and 0 for observations that are observed exactly.

`[...] = gamfit(data,alpha,censoring,freq)` accepts a frequency vector of the same size as `data`. `freq` typically contains integer frequencies for the corresponding elements in `data`, but may contain any nonnegative values.

`[...] = gamfit(data,alpha,censoring,freq,options)` accepts a structure, `options`, that specifies control parameters for the iterative algorithm the function uses to compute maximum likelihood estimates. The gamma fit function accepts an `options` structure which can be created using the function `statset`. Enter `statset('gamfit')`

to see the names and default values of the parameters that `gamfit` accepts in the `options` structure.

## Examples

Fit a gamma distribution to random data generated from a specified gamma distribution:

```
a = 2; b = 4;
data = gamrnd(a,b,100,1);

[p,ci] = gamfit(data)
p =
    2.1990    3.7426
ci =
    1.6840    2.8298
    2.7141    4.6554
```

## References

- [1] Hahn, Gerald J., and S. S. Shapiro. *Statistical Models in Engineering*. Hoboken, NJ: John Wiley & Sons, Inc., 1994, p. 88.

## See Also

`gamcdf` | `gaminv` | `gamlike` | `gampdf` | `gamrnd` | `gamstat` | `mle`

## Topics

“Gamma Distribution” on page B-47

Introduced before R2006a

# gaminv

Gamma inverse cumulative distribution function

## Syntax

```
X = gaminv(P,A,B)
[X,XLO,XUP] = gamcdf(P,A,B,pcov,alpha)
```

## Description

`X = gaminv(P,A,B)` computes the inverse of the gamma cdf with shape parameters in `A` and scale parameters in `B` for the corresponding probabilities in `P`. `P`, `A`, and `B` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array with the same dimensions as the other inputs. The parameters in `A` and `B` must all be positive, and the values in `P` must lie on the interval `[0 1]`.

The gamma inverse function in terms of the gamma cdf is

$$x = F^{-1}(p | a, b) = \{x : F(x | a, b) = p\}$$

where

$$p = F(x | a, b) = \frac{1}{b^a \Gamma(a)} \int_0^x t^{a-1} e^{-\frac{t}{b}} dt$$

`[X,XLO,XUP] = gamcdf(P,A,B,pcov,alpha)` produces confidence bounds for `P` when the input parameters `A` and `B` are estimates. `pcov` is a 2-by-2 matrix containing the covariance matrix of the estimated parameters. `alpha` has a default value of 0.05, and specifies 100(1-alpha)% confidence bounds. `PLO` and `PUP` are arrays of the same size as `P` containing the lower and upper confidence bounds.

## Examples

This example shows the relationship between the gamma cdf and its inverse function.

```
a = 1:5;
b = 6:10;
x = gaminv(gamcdf(1:5, a, b), a, b)
x =
    1.0000    2.0000    3.0000    4.0000    5.0000
```

## Algorithms

There is no known analytical solution to the integral equation above. `gaminv` uses an iterative approach (Newton's method) to converge on the solution.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`gamcdf` | `gamfit` | `gamlike` | `gampdf` | `gamrnd` | `gamstat` | `icdf`

### Topics

“Gamma Distribution” on page B-47

Introduced before R2006a

# gamlike

Gamma negative log-likelihood

## Syntax

```
nlogL = gamlike(params,data)
[nlogL,AVAR] = gamlike(params,data)
```

## Description

`nlogL = gamlike(params,data)` returns the negative of the gamma log-likelihood of the parameters, `params`, given `data`. `params(1)=A`, shape parameters, and `params(2)=B`, scale parameters. The parameters in `params` must all be positive

`[nlogL,AVAR] = gamlike(params,data)` also returns `AVAR`, which is the asymptotic variance-covariance matrix of the parameter estimates when the values in `params` are the maximum likelihood estimates. `AVAR` is the inverse of Fisher's information matrix. The diagonal elements of `AVAR` are the asymptotic variances of their respective parameters.

`[...] = gamlike(params,data,censoring)` accepts a Boolean vector of the same size as `data` that is 1 for observations that are right-censored and 0 for observations that are observed exactly.

`[...] = gamfit(params,data,censoring,freq)` accepts a frequency vector of the same size as `data`. `freq` typically contains integer frequencies for the corresponding elements in `data`, but may contain any non-negative values.

`gamlike` is a utility function for maximum likelihood estimation of the gamma distribution. Since `gamlike` returns the negative gamma log-likelihood function, minimizing `gamlike` using `fminsearch` is the same as maximizing the likelihood.

## Examples

Compute the negative log-likelihood of parameter estimates computed by the `gamfit` function:

```
a = 2; b = 3;
r = gamrnd(a,b,100,1);

[nlogL,AVAR] = gamlike(gamfit(r),r)
nlogL =
    267.5648
AVAR =
    0.0788   -0.1104
   -0.1104    0.1955
```

## See Also

`gamcdf` | `gamfit` | `gaminv` | `gampdf` | `gamrnd` | `gamstat`

## Topics

“Gamma Distribution” on page B-47

**Introduced before R2006a**

# gampdf

Gamma probability density function

## Syntax

```
Y = gampdf(X,A,B)
```

## Description

`Y = gampdf(X,A,B)` computes the gamma pdf at each of the values in `X` using the corresponding shape parameters in `A` and scale parameters in `B`. `X`, `A`, and `B` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array with the same dimensions as the other inputs. The parameters in `A` and `B` must all be positive, and the values in `X` must lie on the interval  $[0 \text{ Inf}]$ .

The gamma pdf is

$$y = f(x | a, b) = \frac{1}{b^a \Gamma(a)} x^{a-1} e^{-\frac{x}{b}}$$

The gamma probability density function is useful in reliability models of lifetimes. The gamma distribution is more flexible than the exponential distribution in that the probability of a product surviving an additional period may depend on its current age. The exponential and  $\chi^2$  functions are special cases of the gamma function.

## Examples

The exponential distribution is a special case of the gamma distribution.

```
mu = 1:5;
```

```
y = gampdf(1,1,mu)  
y =
```

```
0.3679 0.3033 0.2388 0.1947 0.1637  
y1 = exppdf(1,mu)  
y1 =  
0.3679 0.3033 0.2388 0.1947 0.1637
```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

[gamcdf](#) | [gamfit](#) | [gaminv](#) | [gamlike](#) | [gamrnd](#) | [gamstat](#) | [pdf](#)

### Topics

“Gamma Distribution” on page B-47

Introduced before R2006a



# gamrnd

Gamma random numbers

## Syntax

```
R = gamrnd(A,B)
R = gamrnd(A,B,m,n,...)
R = gamrnd(A,B,[m,n,...])
```

## Description

`R = gamrnd(A,B)` generates random numbers from the gamma distribution with shape parameters in `A` and scale parameters in `B`. `A` and `B` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input for `A` or `B` is expanded to a constant array with the same dimensions as the other input. The parameters in `A` and `B` must be positive.

`R = gamrnd(A,B,m,n,...)` or `R = gamrnd(A,B,[m,n,...])` generates an `m`-by-`n`-by-... array containing random numbers from the gamma distribution with parameters `A` and `B`. `A` and `B` can each be scalars or arrays of the same size as `R`.

## Examples

```
n1 = gamrnd(1:5,6:10)
n1 =
    9.1132  12.8431  24.8025  38.5960 106.4164

n2 = gamrnd(5,10,[1 5])
n2 =
    30.9486  33.5667  33.6837  55.2014  46.8265

n3 = gamrnd(2:6,3,1,5)
n3 =
    12.8715  11.3068  3.0982  15.6012  21.6739
```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

The generated code can return a different sequence of numbers than MATLAB if either of the following is true:

- The output is nonscalar.
- An input parameter is invalid for the distribution.

### See Also

`gamcdf` | `gamfit` | `gaminv` | `gamlike` | `gampdf` | `gamstat` | `randg` | `random`

### Topics

“Gamma Distribution” on page B-47

**Introduced before R2006a**

# gamstat

Gamma mean and variance

## Syntax

```
[M,V] = gamstat(A,B)
```

## Description

`[M,V] = gamstat(A,B)` returns the mean of and variance for the gamma distribution with shape parameters in *A* and scale parameters in *B*. *A* and *B* can be vectors, matrices, or multidimensional arrays that have the same size, which is also the size of *M* and *V*. A scalar input for *A* or *B* is expanded to a constant array with the same dimensions as the other input. The parameters in *A* and *B* must be positive.

The mean of the gamma distribution with parameters *A* and *B* is  $AB$ . The variance is  $AB^2$ .

## Examples

```
[m,v] = gamstat(1:5,1:5)
```

```
m =
```

```
1 4 9 16 25
```

```
v =
```

```
1 8 27 64 125
```

```
[m,v] = gamstat(1:5,1./(1:5))
```

```
m =
```

```
1 1 1 1 1
```

```
v =
```

```
1.0000 0.5000 0.3333 0.2500 0.2000
```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`gamcdf` | `gamfit` | `gaminv` | `gamlike` | `gampdf` | `gamrnd`

### Topics

“Gamma Distribution” on page B-47

Introduced before R2006a

## ge

**Class:** grandstream

Greater than or equal relation for handles

## Syntax

```
h1 >= h2
```

## Description

`h1 >= h2` performs element-wise comparisons between handle arrays `h1` and `h2`. `h1` and `h2` must be of the same dimensions unless one is a scalar. The result is a logical array of the same dimensions, where each element is an element-wise `>=` result.

If one of `h1` or `h2` is scalar, scalar expansion is performed and the result will match the dimensions of the array that is not scalar.

`tf = ge(h1, h2)` stores the result in a logical array of the same dimensions.

## See Also

`eq` | `gt` | `le` | `lt` | `ne` | `grandstream`

## GeneralizedLinearMixedModel class

Generalized linear mixed-effects model class

### Description

A `GeneralizedLinearMixedModel` object represents a regression model of a response variable that contains both fixed and random effects. The object comprises data, a model description, fitted coefficients, covariance parameters, design matrices, residuals, residual plots, and other diagnostic information for a generalized linear mixed-effects (GLME) model. You can predict model responses with the `predict` function and generate random data at new design points using the `random` function.

### Construction

You can fit a generalized linear mixed-effects (GLME) model to sample data using `fitglme(tbl, formula)`. For more information, see `fitglme`.

### Input Arguments

#### **tbl** — Input data

table | dataset array

Input data, which includes the response variable, predictor variables, and grouping variables, specified as a table or dataset array. The predictor variables can be continuous or grouping variables (see “Grouping Variables” on page 2-59). You must specify the model for the variables using `formula`.

Data Types: `table`

#### **formula** — Formula for model specification

character vector of the form `'y ~ fixed + (random1|grouping1) + ... + (randomR|groupingR)'`

Formula for model specification, specified as a character vector of the form `'y ~ fixed + (random1|grouping1) + ... + (randomR|groupingR)'`. For a full description, see [Formula](#) on page 32-2824.

Example: `'y ~ treatment +(1|block)'`

## Properties

### **Coefficients** — Estimates of fixed-effects coefficients

dataset array

Estimates of fixed-effects coefficients and related statistics, stored as a dataset array that has one row for each coefficient and the following columns:

- `Name` — Name of the coefficient
- `Estimate` — Estimated coefficient value
- `SE` — Standard error of the estimate
- `tStat` — *t*-statistic for a test that the coefficient is equal to 0
- `DF` — Degrees of freedom associated with the *t* statistic
- `pValue` — *p*-value for the *t*-statistic
- `Lower` — Lower confidence limit
- `Upper` — Upper confidence limit

To obtain any of these columns as a vector, index into the property using dot notation.

Use the `coefTest` method to perform other tests on the coefficients.

### **CoefficientCovariance** — Covariance of estimated fixed-effects vector

matrix

Covariance of estimated fixed-effects vector, stored as a matrix.

Data Types: `single` | `double`

### **CoefficientNames** — Names of fixed-effects coefficients

cell array of character vectors

Names of fixed-effects coefficients, stored as a cell array of character vectors. The label for the coefficient of the constant term is `(Intercept)`. The labels for other coefficients

indicate the terms that they multiply. When the term includes a categorical predictor, the label also indicates the level of that predictor.

Data Types: `cell`

### **DFE — Degrees of freedom for error**

positive integer value

Degrees of freedom for error, stored as a positive integer value. DFE is the number of observations minus the number of estimated coefficients.

DFE contains the degrees of freedom corresponding to the 'Residual' method of calculating denominator degrees of freedom for hypothesis tests on fixed-effects coefficients. If  $n$  is the number of observations and  $p$  is the number of fixed-effects coefficients, then DFE is equal to  $n - p$ .

Data Types: `double`

### **Dispersion — Model dispersion parameter**

scalar value

Model dispersion parameter, stored as a scalar value. The dispersion parameter defines the conditional variance of the response.

For observation  $i$ , the conditional variance of the response  $y_i$ , given the conditional mean  $\mu_i$  and the dispersion parameter  $\sigma^2$ , in a generalized linear mixed-effects model is

$$\text{var}(y_i | \mu_i, \sigma^2) = \frac{\sigma^2}{w_i} v(\mu_i),$$

where  $w_i$  is the  $i$ th observation weight and  $v$  is the variance function for the specified conditional distribution of the response. The `Dispersion` property contains an estimate of  $\sigma^2$  for the specified GLME model. The value of `Dispersion` depends on the specified conditional distribution of the response. For binomial and Poisson distributions, the theoretical value of `Dispersion` is equal to  $\sigma^2 = 1.0$ .

- If `FitMethod` is `MPL` or `REMPL` and the 'DispersionFlag' name-value pair argument in `fitglme` is `true`, then a dispersion parameter is estimated from data for all distributions, including binomial and Poisson distributions.
- If `FitMethod` is `ApproximateLaplace` or `Laplace`, then the 'DispersionFlag' name-value pair argument in `fitglme` does not apply, and the dispersion parameter



is fixed at 1.0 for binomial and Poisson distributions. For all other distributions, Dispersion is estimated from data.

Data Types: double

**DispersionEstimated** — Flag indicating if dispersion parameter was estimated

true | false

Flag indicating estimated dispersion parameter, stored as a logical value.

- If FitMethod is ApproximateLaplace or Laplace, then the dispersion parameter is fixed at its theoretical value of 1.0 for binomial and Poisson distributions, and DispersionEstimated is false. For other distributions, the dispersion parameter is estimated from the data, and DispersionEstimated is true.
- If FitMethod is MPL or REMPL, and the 'DispersionFlag' name-value pair argument in fitglm is specified as true, then the dispersion parameter is estimated for all distributions, including binomial and Poisson distributions, and DispersionEstimated is true.
- If FitMethod is MPL or REMPL, and the 'DispersionFlag' name-value pair argument in fitglm is specified as false, then the dispersion parameter is fixed at its theoretical value for binomial and Poisson distributions, and DispersionEstimated is false. For distributions other than binomial and Poisson, the dispersion parameter is estimated from the data, and DispersionEstimated is true.

Data Types: logical

**Distribution** — Response distribution name

'Normal' | 'Binomial' | 'Poisson' | 'Gamma' | 'InverseGaussian'

Response distribution name, stored as one of the following:

- 'Normal' — Normal distribution
- 'Binomial' — Binomial distribution
- 'Poisson' — Poisson distribution
- 'Gamma' — Gamma distribution
- 'InverseGaussian' — Inverse Gaussian distribution

**FitMethod** — Method used to fit the model`'MPL' | 'REML' | 'ApproximateLaplace' | 'Laplace'`

Method used to fit the model, stored as one of the following.

- `'MPL'` — Maximum pseudo likelihood
- `'REML'` — Restricted maximum pseudo likelihood
- `'ApproximateLaplace'` — Maximum likelihood using the approximate Laplace method, with fixed effects profiled out
- `'Laplace'` — Maximum likelihood using the Laplace method

**Formula** — Model specification formula`object`

Model specification formula, stored as an object. The model specification formula uses Wilkinson's notation to describe the relationship between the fixed-effects terms, random-effects terms, and grouping variables in the GLME model. For more information see Formula on page 32-2824.

**Link** — Link function characteristics`structure`

Link function characteristics, stored as a structure containing the following fields. The link is a function  $G$  that links the distribution parameter  $\mu$  to the linear predictor  $\eta$  as follows:  $G(\mu) = \eta$ .

Field	Description
Name	Name of the link function
Link	Function that defines $G$
Derivative	Derivative of $G$
SecondDerivative	Second derivative of $G$
Inverse	Inverse of $G$

Data Types: `struct`

**LogLikelihood** — Log of likelihood function`scalar value`

Log of likelihood function evaluated at the estimated coefficient values, stored as a scalar value. `LogLikelihood` depends on the method used to fit the model.

- If you use 'Laplace' or 'ApproximateLaplace', then `LogLikelihood` is the maximized log likelihood.
- If you use 'MPL', then `LogLikelihood` is the maximized log likelihood of the pseudo data from the final pseudo likelihood iteration.
- If you use 'REML', then `LogLikelihood` is the maximized restricted log likelihood of the pseudo data from the final pseudo likelihood iteration.

Data Types: double

### **ModelCriterion** — Model criterion

table

Model criterion to compare fitted generalized linear mixed-effects models, stored as a table with the following fields.

Field	Description
AIC	Akaike information criterion
BIC	Bayesian information criterion
LogLikelihood	<ul style="list-style-type: none"> <li>• For a model fit using 'Laplace' or 'ApproximateLaplace', <code>LogLikelihood</code> is the maximized log likelihood.</li> <li>• For a model fit using 'MPL', <code>LogLikelihood</code> is the maximized log likelihood of the pseudo data from the final pseudo likelihood iteration.</li> <li>• For a model fit using 'REML', <code>LogLikelihood</code> is the maximized restricted log likelihood of the pseudo data from the final pseudo likelihood iteration.</li> </ul>
Deviance	-2 times <code>LogLikelihood</code>

### **NumCoefficients** — Number of fixed-effects coefficients

positive integer value

Number of fixed-effects coefficients in the fitted generalized linear mixed-effects model, stored as a positive integer value.

Data Types: `double`

**NumEstimatedCoefficients** — Number of estimated fixed-effects coefficients

positive integer value

Number of estimated fixed-effects coefficients in the fitted generalized linear mixed-effects model, stored as a positive integer value.

Data Types: `double`

**NumObservations** — Number of observations

positive integer value

Number of observations used in the fit, stored as a positive integer value. `NumObservations` is the number of rows in the table or dataset array `tbl`, minus rows excluded using the `'Exclude'` name-value pair of `fitglm` or rows containing NaN values.

Data Types: `double`

**NumPredictors** — Number of predictors

positive integer value

Number of variables used as predictors in the generalized linear mixed-effects model, stored as a positive integer value.

Data Types: `double`

**NumVariables** — Total number of variables

positive integer value

Total number of variables, including the response and predictors, stored as a positive integer value. If the sample data is in a table or dataset array `tbl`, then `NumVariables` is the total number of variables in `tbl`, including the response variable. `NumVariables` includes variables, if any, that are not used as predictors or as the response.

Data Types: `double`

**ObservationInfo** — Information about the observations

table

Information about the observations used in the fit, stored as a table.

ObservationInfo has one row for each observation and the following columns.

Name	Description
Weights	The weight value for the observation. The default value is 1.
Excluded	If the observation was excluded from the fit using the 'Exclude' name-value pair argument in fitglme, then Excluded is true, or 1. Otherwise, Excluded is false, or 0.
Missing	If the observation was excluded from the fit because any response or predictor value is missing, then Missing is true. Otherwise, Missing is false.  Missing values include NaN for numeric variables, empty cells for cell arrays, blank rows for character arrays, and the <undefined> value for categorical arrays.
Subset	If the observation was used in the fit, then Subset is true. If the observation was not used in the fit because it is missing or excluded, then Subset is false.
BinomSize	Binomial size for each observation. This column only applies when fitting a binomial distribution.

Data Types: table

### ObservationNames — Names of observations

cell array of character vectors

Names of observations used in the fit, stored as a cell array of character vectors.

- If the data is in a table or dataset array tbl that contains observation names, then ObservationNames uses those names.

- If the data is provided in matrices, or in a table or dataset array without observation names, then `ObservationNames` is an empty cell array.

Data Types: `cell`

#### **PredictorNames** — Names of predictors

cell array of character vectors

Names of the variables used as predictors in the fit, stored as a cell array of character vectors that has the same length as `NumPredictors`.

Data Types: `cell`

#### **ResponseName** — Name of response variable

character vector

Name of the variable used as the response variable in the fit, stored as a character vector.

Data Types: `char`

#### **Rsquared** — Proportion of variability in the response explained by the fitted model

structure

Proportion of variability in the response explained by the fitted model, stored as a structure. `Rsquared` contains the *R*-squared value of the fitted model, also known as the multiple correlation coefficient. `Rsquared` contains the following fields.

Field	Description
Ordinary	R-squared value, stored as a scalar value in a structure. $Rsquared.Ordinary = 1 - SSE./SST$
Adjusted	R-squared value adjusted for the number of fixed-effects coefficients, stored as a scalar value in a structure. $Rsquared.Adjusted = 1 - (SSE./SST) * (DFT./DFE)$ , where $DFE = n - p$ , $DFT = n - 1$ , $n$ is the total number of observations, and $p$ is the number of fixed-effects coefficients.

Data Types: `struct`

**SSE — Error sum of squares**

positive scalar value

Error sum of squares, stored as a positive scalar value. SSE is the weighted sum of the squared conditional residuals, and is calculated as

$$SSE = \sum_{i=1}^n w_i^{eff} (y_i - f_i)^2,$$

where  $n$  is the number of observations,  $w_i^{eff}$  is the  $i$ th effective weight,  $y_i$  is the  $i$ th response, and  $f_i$  is the  $i$ th fitted value.

The  $i$ th effective weight is calculated as

$$w_i^{eff} = \left\{ \frac{w_i}{v_i(\mu_i(\hat{\beta}, \hat{b}))} \right\},$$

where  $v_i$  is the variance term for the  $i$ th observation,  $\hat{\beta}$  and  $\hat{b}$  are estimated values of  $\beta$  and  $b$ , respectively.

The  $i$ th fitted value is calculated as

$$f_i = g^{-1}(x_i^T \hat{\beta} + z_i^T \hat{b} + \delta_i),$$

where  $x_i^T$  is the  $i$ th row of the fixed-effects design matrix  $X$ , and  $z_i^T$  is the  $i$ th row of the random-effects design matrix  $Z$ .  $\delta_i$  is the  $i$ th offset value.

Data Types: `double`

**SSR — Regression sum of squares**

positive scalar value

Regression sum of squares, stored as a positive scalar value. SSR is the sum of squares explained by the generalized linear mixed-effects regression, or equivalently the weighted sum of the squared deviations of the conditional fitted values from their weighted mean. SSR is calculated as

$$SSR = \sum_{i=1}^N w_i^{eff} (f_i - \bar{f})^2,$$

where  $n$  is the number of observations,  $w_i^{\text{eff}}$  is the  $i$ th effective weight,  $f_i$  is the  $i$ th fitted value, and  $\bar{f}$  is a weighted average of the fitted values.

The  $i$ th effective weight is calculated as

$$w_i^{\text{eff}} = \left\{ \frac{w_i}{v_i(\mu_i(\hat{\beta}, \hat{b}))} \right\},$$

where  $\hat{\beta}$  and  $\hat{b}$  are estimated values of  $\beta$  and  $b$ , respectively.

The  $i$ th fitted value is calculated as

$$f_i = g^{-1}(x_i^T \hat{\beta} + z_i^T \hat{b} + \delta_i),$$

where  $x_i^T$  is the  $i$ th row of the fixed-effects design matrix  $X$ , and  $z_i^T$  is the  $i$ th row of the random-effects design matrix  $Z$ .  $\delta_i$  is the  $i$ th offset value.

The weighted average of fitted values is calculated as

$$\bar{f} = \frac{\left[ \sum_{i=1}^n w_i^{\text{eff}} f_i \right]}{\sum_{i=1}^n w_i^{\text{eff}}}.$$

Data Types: double

### **SST** — Total sum of squares

positive scalar value

Total sum of squares, stored as a positive scalar value. For a GLME model, SST is defined as  $\text{SST} = \text{SSE} + \text{SSR}$ .

Data Types: double

### **VariableInfo** — Information about the variables

table

Information about the variables used in the fit, stored as a table. `VariableInfo` has one row for each variable and contains the following columns.



Column Name	Description
Class	Class of the variable ('double', 'cell', 'nominal', and so on).
Range	Value range of the variable. <ul style="list-style-type: none"> <li>For a numerical variable, Range is a two-element vector of the form [min,max].</li> <li>For a cell or categorical variable, Range is a cell or categorical array containing all unique values of the variable.</li> </ul>
InModel	If the variable is a predictor in the fitted model, InModel is true.  If the variable is not in the fitted model, InModel is false.
IsCategorical	If the variable type is treated as a categorical predictor (such as cell, logical, or categorical), then IsCategorical is true.  If the variable is a continuous predictor, then IsCategorical is false

Data Types: table

#### **VariableNames** — Names of the variables

cell array of character vectors

Names of all the variables contained in the table or dataset array `tbl`, stored as a cell array of character vectors.

Data Types: cell

#### **Variables** — Variables

table

Variables, stored as a table. If the fit is based on a table or dataset array `tbl`, then `Variables` is identical to `tbl`.

Data Types: `table`

## Methods

<code>anova</code>	Analysis of variance for generalized linear mixed-effects model
<code>coefCI</code>	Confidence intervals for coefficients of generalized linear mixed-effects model
<code>coefTest</code>	Hypothesis test on fixed and random effects of generalized linear mixed-effects model
<code>compare</code>	Compare generalized linear mixed-effects models
<code>covarianceParameters</code>	Extract covariance parameters of generalized linear mixed-effects model
<code>designMatrix</code>	Fixed- and random-effects design matrices
<code>fitted</code>	Fitted responses from generalized linear mixed-effects model
<code>fixedEffects</code>	Estimates of fixed effects and related statistics
<code>plotResiduals</code>	Plot residuals of generalized linear mixed-effects model
<code>predict</code>	Predict response of generalized linear mixed-effects model
<code>random</code>	Generate random responses from fitted generalized linear mixed-effects model
<code>randomEffects</code>	Estimates of random effects and related statistics
<code>refit</code>	Refit generalized linear mixed-effects model
<code>residuals</code>	Residuals of fitted generalized linear mixed-effects model
<code>response</code>	Response vector of generalized linear mixed-effects model

## Examples

### Fit a Generalized Linear Mixed-Effects Model

Load the sample data.

```
load mfr
```

This simulated data is from a manufacturing company that operates 50 factories across the world, with each factory running a batch process to create a finished product. The company wants to decrease the number of defects in each batch, so it developed a new manufacturing process. To test the effectiveness of the new process, the company selected 20 of its factories at random to participate in an experiment: Ten factories implemented the new process, while the other ten continued to run the old process. In each of the 20 factories, the company ran five batches (for a total of 100 batches) and recorded the following data:

- Flag to indicate whether the batch used the new process (`newprocess`)
- Processing time for each batch, in hours (`time`)
- Temperature of the batch, in degrees Celsius (`temp`)
- Categorical variable indicating the supplier (A, B, or C) of the chemical used in the batch (`supplier`)
- Number of defects in the batch (`defects`)

The data also includes `time_dev` and `temp_dev`, which represent the absolute deviation of time and temperature, respectively, from the process standard of 3 hours at 20 degrees Celsius.

Fit a generalized linear mixed-effects model using `newprocess`, `time_dev`, `temp_dev`, and `supplier` as fixed-effects predictors. Include a random-effects term for intercept grouped by `factory`, to account for quality differences that might exist due to factory-specific variations. The response variable `defects` has a Poisson distribution, and the appropriate link function for this model is `log`. Use the Laplace fit method to estimate the coefficients. Specify the dummy variable encoding as `'effects'`, so the dummy variable coefficients sum to 0.

The number of defects can be modeled using a Poisson distribution

$$\text{defects}_{ij} \sim \text{Poisson}(\mu_{ij})$$

This corresponds to the generalized linear mixed-effects model

$$\log(\mu_{ij}) = \beta_0 + \beta_1 \text{newprocess}_{ij} + \beta_2 \text{time\_dev}_{ij} + \beta_3 \text{temp\_dev}_{ij} + \beta_4 \text{supplier\_C}_{ij} + \beta_5 \text{supplier\_B}_{ij} + b_i,$$

where

- `defectsij` is the number of defects observed in the batch produced by factory  $i$  during batch  $j$ .

- $\mu_{ij}$  is the mean number of defects corresponding to factory  $i$  (where  $i = 1, 2, \dots, 20$ ) during batch  $j$  (where  $j = 1, 2, \dots, 5$ ).
- `newprocessij`, `time_devij`, and `temp_devij` are the measurements for each variable that correspond to factory  $i$  during batch  $j$ . For example, `newprocessij` indicates whether the batch produced by factory  $i$  during batch  $j$  used the new process.
- `supplier_Cij` and `supplier_Bij` are dummy variables that use effects (sum-to-zero) coding to indicate whether company C or B, respectively, supplied the process chemicals for the batch produced by factory  $i$  during batch  $j$ .
- $b_i \sim N(0, \sigma_b^2)$  is a random-effects intercept for each factory  $i$  that accounts for factory-specific variation in quality.

```
glme = fitglme(mfr, 'defects ~ 1 + newprocess + time_dev + temp_dev + supplier + (1|factory)
  Distribution', 'Poisson', 'Link', 'log', 'FitMethod', 'Laplace', 'DummyVarCoding', 'effects')
```

Display the model.

```
disp(glme)
```

Generalized linear mixed-effects model fit by ML

Model information:

Number of observations	100
Fixed effects coefficients	6
Random effects coefficients	20
Covariance parameters	1
Distribution	Poisson
Link	Log
FitMethod	Laplace

Formula:

```
defects ~ 1 + newprocess + time_dev + temp_dev + supplier + (1 | factory)
```

Model fit statistics:

AIC	BIC	LogLikelihood	Deviance
416.35	434.58	-201.17	402.35

Fixed effects coefficients (95% CIs):

Name	Estimate	SE	tStat	DF	pValue
'(Intercept)'	1.4689	0.15988	9.1875	94	9.8194e-15
'newprocess'	-0.36766	0.17755	-2.0708	94	0.041122

'time_dev'	-0.094521	0.82849	-0.11409	94	0.90941
'temp_dev'	-0.28317	0.9617	-0.29444	94	0.76907
'supplier_C'	-0.071868	0.078024	-0.9211	94	0.35936
'supplier_B'	0.071072	0.07739	0.91836	94	0.36078

Lower	Upper
1.1515	1.7864
-0.72019	-0.015134
-1.7395	1.5505
-2.1926	1.6263
-0.22679	0.083051
-0.082588	0.22473

Random effects covariance parameters:

Group: factory (20 Levels)

Name1	Name2	Type	Estimate
'(Intercept)'	'(Intercept)'	'std'	0.31381

Group: Error

Name	Estimate
'sqrt(Dispersion)'	1

The `Model` information table displays the total number of observations in the sample data (100), the number of fixed- and random-effects coefficients (6 and 20, respectively), and the number of covariance parameters (1). It also indicates that the response variable has a Poisson distribution, the link function is `Log`, and the fit method is `Laplace`.

Formula indicates the model specification using Wilkinson's notation.

The `Model fit statistics` table displays statistics used to assess the goodness of fit of the model. This includes the Akaike information criterion (AIC), Bayesian information criterion (BIC) values, log likelihood (`LogLikelihood`), and deviance (`Deviance`) values.

The `Fixed effects coefficients` table indicates that `fitglm` returned 95% confidence intervals. It contains one row for each fixed-effects predictor, and each column contains statistics corresponding to that predictor. Column 1 (`Name`) contains the name of each fixed-effects coefficient, column 2 (`Estimate`) contains its estimated value, and column 3 (`SE`) contains the standard error of the coefficient. Column 4 (`tStat`) contains the *t*-statistic for a hypothesis test that the coefficient is equal to 0. Column 5 (`DF`) and column 6 (`pValue`) contain the degrees of freedom and *P*-value that correspond to the *t*-

statistic, respectively. The last two columns (`Lower` and `Upper`) display the lower and upper limits, respectively, of the 95% confidence interval for each fixed-effects coefficient.

`Random effects covariance parameters` displays a table for each grouping variable (here, only `factory`), including its total number of levels (20), and the type and estimate of the covariance parameter. Here, `std` indicates that `fitglme` returns the standard deviation of the random effect associated with the `factory` predictor, which has an estimated value of 0.31381. It also displays a table containing the error parameter type (here, the square root of the dispersion parameter), and its estimated value of 1.

The standard display generated by `fitglme` does not provide confidence intervals for the random-effects parameters. To compute and display these values, use `covarianceParameters`.

- “Fit a Generalized Linear Mixed-Effects Model” on page 12-79

## Definitions

### Formula

In general, a formula for model specification is a character vector of the form `'y ~ terms'`. For generalized linear mixed-effects models, this formula is in the form `'y ~ fixed + (random1|grouping1) + ... + (randomR|groupingR)'`, where `fixed` and `random` contain the fixed-effects and the random-effects terms, respectively, and `R` is the number of grouping variables in the model.

Suppose a table `tbl` contains the following:

- A response variable, `y`
- Predictor variables, `Xj`, which can be continuous or grouping variables
- Grouping variables, `g1, g2, ..., gR`,

where the grouping variables in `Xj` and `gr` can be categorical, logical, character arrays, or cell arrays of character vectors.

Then, in a formula of the form, `'y ~ fixed + (random1|g1) + ... + (randomR|gR)'`, the term `fixed` corresponds to a specification of the fixed-effects design matrix `X`, `random1` is a specification of the random-effects design matrix `Z1` corresponding to

grouping variable  $g_1$ , and similarly  $\text{random}_R$  is a specification of the random-effects design matrix  $Z_R$  corresponding to grouping variable  $g_R$ . You can express the `fixed` and `random` terms using Wilkinson notation.

Wilkinson notation describes the factors present in models. The notation relates to factors present in models, not to the multipliers (coefficients) of those factors.

Wilkinson Notation	Factors in Standard Notation
1	Constant (intercept) term
$X^k$ , where $k$ is a positive integer	$X, X^2, \dots, X^k$
$X1 + X2$	$X1, X2$
$X1 * X2$	$X1, X2, X1.*X2$ (elementwise multiplication of $X1$ and $X2$ )
$X1 : X2$	$X1.*X2$ only
$- X2$	Do not include $X2$
$X1 * X2 + X3$	$X1, X2, X3, X1 * X2$
$X1 + X2 + X3 + X1 : X2$	$X1, X2, X3, X1 * X2$
$X1 * X2 * X3 - X1 : X2 : X3$	$X1, X2, X3, X1 * X2, X1 * X3, X2 * X3$
$X1 * (X2 + X3)$	$X1, X2, X3, X1 * X2, X1 * X3$

Statistics and Machine Learning Toolbox notation always includes a constant term unless you explicitly remove the term using `-1`. Here are some examples for linear mixed-effects model specification.

#### Examples:

Formula	Description
'y ~ X1 + X2'	Fixed effects for the intercept, $X1$ and $X2$ . This is equivalent to 'y ~ 1 + X1 + X2'.
'y ~ -1 + X1 + X2'	No intercept and fixed effects for $X1$ and $X2$ . The implicit intercept term is suppressed by including <code>-1</code> .
'y ~ 1 + (1   g1)'	Fixed effects for the intercept plus random effect for the intercept for each level of the grouping variable $g1$ .

Formula	Description
'y ~ X1 + (1   g1) '	Random intercept model with a fixed slope.
'y ~ X1 + (X1   g1) '	Random intercept and slope, with possible correlation between them. This is equivalent to 'y ~ 1 + X1 + (1 + X1   g1) '.
'y ~ X1 + (1   g1) + (-1 + X1   g1) '	Independent random effects terms for intercept and slope.
'y ~ 1 + (1   g1) + (1   g2) + (1   g1:g2) '	Random intercept model with independent main effects for g1 and g2, plus an independent interaction effect.

## See Also

`fitglme` | `plotPartialDependence`

## Topics

“Fit a Generalized Linear Mixed-Effects Model” on page 12-79

“Generalized Linear Mixed-Effects Models” on page 12-67



# GeneralizedLinearModel class

## Superclasses:

Generalized linear regression model class

## Description

An object comprising training data, model description, diagnostic information, and fitted coefficients for a generalized linear regression. Predict model responses with the `predict` or `feval` methods.

## Construction

`mdl = fitglm(tbl)` or `mdl = fitglm(X,y)` creates a generalized linear model of a table or dataset array `tbl`, or of the responses `y` to a data matrix `X`. For details, see `fitglm`.

`mdl = stepwiseglm(tbl)` or `mdl = stepwiseglm(X,y)` creates a generalized linear model of a table or dataset array `tbl`, or of the responses `y` to a data matrix `X`, with unimportant predictors excluded. For details, see `stepwiseglm`.

## Input Arguments

### **tbl** — Input data

table | dataset array

Input data, specified as a table or dataset array. When `modelspec` is a formula, it specifies the variables to be used as the predictors and response. Otherwise, if you do not specify the predictor and response variables, the last variable is the response variable and the others are the predictor variables by default.

Predictor variables can be numeric, or any grouping variable type, such as logical or categorical (see “Grouping Variables” on page 2-59). The response must be numeric or logical.

To set a different column as the response variable, use the `ResponseVar` name-value pair argument. To use a subset of the columns as predictors, use the `PredictorVars` name-value pair argument.

**x — Predictor variables**

matrix

Predictor variables, specified as an  $n$ -by- $p$  matrix, where  $n$  is the number of observations and  $p$  is the number of predictor variables. Each column of `X` represents one variable, and each row represents one observation.

By default, there is a constant term in the model, unless you explicitly remove it, so do not include a column of 1s in `X`.

Data Types: `single` | `double` | `logical`

**y — Response variable**

vector

Response variable, specified as an  $n$ -by-1 vector, where  $n$  is the number of observations. Each entry in `y` is the response for the corresponding row of `X`.

Data Types: `single` | `double` | `logical`

## Properties

**CoefficientCovariance — Covariance matrix of coefficient estimates**

numeric matrix

Covariance matrix of coefficient estimates, stored as a  $p$ -by- $p$  matrix of numeric values.  $p$  is the number of coefficients in the fitted model.

**CoefficientNames — Coefficient names**

cell array of character vectors

Coefficient names, stored as a cell array of character vectors containing a label for each coefficient.

**Coefficients — Coefficient values**

table

Coefficient values, stored as a table. `Coefficients` has one row for each coefficient and the following columns:

- `Estimate` — Estimated coefficient value
- `SE` — Standard error of the estimate
- `tStat` —  $t$  statistic for a test that the coefficient is zero
- `pValue` —  $p$ -value for the  $t$  statistic

To obtain any of these columns as a vector, index into the property using dot notation. For example, in `mdl` the estimated coefficient vector is

```
beta = mdl.Coefficients.Estimate
```

Use `coefTest` to perform other tests on the coefficients.

#### **Deviance — Deviance of the fit**

numeric value

Deviance of the fit, stored as a numeric value. Deviance is useful for comparing two models when one is a special case of the other. The difference between the deviance of the two models has a chi-square distribution with degrees of freedom equal to the difference in the number of estimated parameters between the two models. For more information on deviance, see “Deviance” on page 32-2845.

#### **DFE — Degrees of freedom for error**

positive integer value

Degrees of freedom for error (residuals), equal to the number of observations minus the number of estimated coefficients, stored as a positive integer value.

#### **Diagnostics — Diagnostic information**

table

Diagnostic information for the model, stored as a table. Diagnostics can help identify outliers and influential observations. `Diagnostics` contains the following fields:

Field	Meaning	Utility
Leverage	Diagonal elements of HatMatrix	Leverage indicates to what extent the predicted value for an observation is determined by the observed value for that observation. A value close to 1 indicates that the prediction is largely determined by that observation, with little contribution from the other observations. A value close to 0 indicates the fit is largely determined by the other observations. For a model with $p$ coefficients and $n$ observations, the average value of Leverage is $p/n$ . An observation with Leverage larger than $2*p/n$ can be an outlier.
CooksDistance	Cook's measure of scaled change in fitted values	CooksDistance is a measure of scaled change in fitted values. An observation with Cook'sDistance larger than three times the mean Cook's distance can be an outlier.
HatMatrix	Projection matrix to compute fitted from observed responses	HatMatrix is an $n$ -by- $n$ matrix such that $\text{Fitted} = \text{HatMatrix} * Y$ , where $Y$ is the response vector and Fitted is the vector of fitted response values.

All of these quantities are computed on the scale of the linear predictor. So, for example, in the equation that defines the hat matrix,

```
Yfit = glm.Fitted.LinearPredictor
Y = glm.Fitted.LinearPredictor + glm.Residuals.LinearPredictor
```

### **Dispersion — Scale factor of the variance of the response**

structure

Scale factor of the variance of the response, stored as a structure. Dispersion multiplies the variance function for the distribution.

For example, the variance function for the binomial distribution is  $p(1-p)/n$ , where  $p$  is the probability parameter and  $n$  is the sample size parameter. If Dispersion is near 1, the variance of the data appears to agree with the theoretical variance of the binomial distribution. If Dispersion is larger than 1, the data are “overdispersed” relative to the binomial distribution.

### **DispersionEstimated — Flag to indicate use of dispersion scale factor**

logical value

Flag to indicate whether `fitglm` used the `Dispersion` scale factor to compute standard errors for the coefficients in `Coefficients.SE`, stored as a logical value. If `DispersionEstimated` is false, `fitglm` used the theoretical value of the variance.

- `DispersionEstimated` can be false only for 'binomial' or 'poisson' distributions.
- Set `DispersionEstimated` by setting the `DispersionFlag` name-value pair in `fitglm`.

### **Distribution** — Generalized distribution information structure

Generalized distribution information, stored as a structure with the following fields relating to the generalized distribution:

Field	Description
Name	Name of the distribution, one of 'normal', 'binomial', 'poisson', 'gamma', or 'inverse gamma'.
DevianceFunction	Function that computes the components of the deviance as a function of the fitted parameter values and the response values.
VarianceFunction	Function that computes the theoretical variance for the distribution as a function of the fitted parameter values. When <code>DispersionEstimated</code> is true, <code>Dispersion</code> multiplies the variance function in the computation of the coefficient standard errors.

### **Fitted** — Fitted response values based on input data table

Fitted (predicted) values based on the input data, stored as a table with one row for each observation and the following columns.

Field	Description
Response	Predicted values on the scale of the response.
LinearPredictor	Predicted values on the scale of the linear predictor. These are the same as the link function applied to the <code>Response</code> fitted values.
Probability	Fitted probabilities (this column is included only with the binomial distribution).

To obtain any of the columns as a vector, index into the property using dot notation. For example, in the model `mdl`, the vector `f` of fitted values on the response scale is

```
f = mdl.Fitted.Response
```

Use `predict` to compute predictions for other predictor values, or to compute confidence bounds on `Fitted`.

### Formula — Model information

`LinearFormula` object | `NonLinearFormula` object

Model information, stored as a `LinearFormula` object or `NonLinearFormula` object. If you fit a linear or generalized linear regression model, then `Formula` is a `LinearFormula` object. If you fit a nonlinear regression model, then `Formula` is a `NonLinearFormula` object.

### Link — Link function

structure

Link function, stored as a structure with the following fields:

Field	Description
Name	Name of the link function, or ' ' if you specified the link as a function handle rather than a character vector.
LinkFunction	The function that defines $f$ , a function handle.
DevianceFunction	Derivative of $f$ , a function handle.
VarianceFunction	Inverse of $f$ , a function handle.

The link is a function  $f$  that links the distribution parameter  $\mu$  to the fitted linear combination  $Xb$  of the predictors:

$$f(\mu) = Xb.$$

### LogLikelihood — Log likelihood

numeric value

Log likelihood of the model distribution at the response values, stored as a numeric value. The mean is fitted from the model, and other parameters are estimated as part of the model fit.

### ModelCriterion — Criterion for model comparison

structure

Criterion for model comparison, stored as a structure with the following fields:

- AIC — Akaike information criterion
- AIC<sub>c</sub> — Akaike information criterion corrected for sample size
- BIC — Bayesian information criterion
- CAIC — Consistent Akaike information criterion

To obtain any of these values as a scalar, index into the property using dot notation. For example, in a model `mdl`, the AIC value `aic` is:

```
aic = mdl.ModelCriterion.AIC
```

**NumCoefficients — Number of model coefficients**

positive integer

Number of model coefficients, stored as a positive integer. `NumCoefficients` includes coefficients that are set to zero when the model terms are rank deficient.

**NumEstimatedCoefficients — Number of estimated coefficients**

positive integer

Number of estimated coefficients in the model, stored as a positive integer. `NumEstimatedCoefficients` does not include coefficients that are set to zero when the model terms are rank deficient. `NumEstimatedCoefficients` is the degrees of freedom for regression.

**NumObservations — Number of observations**

positive integer

Number of observations the fitting function used in fitting, stored as a positive integer. This is the number of observations supplied in the original table, dataset, or matrix, minus any excluded rows (set with the `Excluded` name-value pair) or rows with missing values.

**NumPredictors — Number of predictor variables**

positive integer

Number of predictor variables used to fit the model, stored as a positive integer.

**NumVariables — Number of variables**

positive integer

Number of variables in the input data, stored as a positive integer. `NumVariables` is the number of variables in the original table or dataset, or the total number of columns in

the predictor matrix and response vector when the fit is based on those arrays. It includes variables, if any, that are not used as predictors or as the response.

### **ObservationInfo** — Observation information

table

Observation information, stored as a  $n$ -by-4 table, where  $n$  is equal to the number of rows of input data. The four columns of `ObservationInfo` contain the following:

Field	Description
Weights	Observation weights. Default is all 1.
Excluded	Logical value, 1 indicates an observation that you excluded from the fit with the <code>Exclude</code> name-value pair.
Missing	Logical value, 1 indicates a missing value in the input. Missing values are not used in the fit.
Subset	Logical value, 1 indicates the observation is not excluded or missing, so is used in the fit.

### **ObservationNames** — Observation names

cell array of character vectors

Observation names, stored as a cell array of character vectors containing the names of the observations used in the fit.

- If the fit is based on a table or dataset containing observation names, `ObservationNames` uses those names.
- Otherwise, `ObservationNames` is an empty cell array

### **offset** — Offset variable

numeric vector

Offset variable, stored as a numeric vector with the same length as the number of rows in the data. `Offset` is passed from `fitglm` or `stepwiseglm` in the `Offset` name-value pair. The fitting function used `Offset` as a predictor variable, but with the coefficient set to exactly 1. In other words, the formula for fitting was

$$\mu \sim \text{Offset} + (\text{terms involving real predictors})$$

with the `Offset` predictor having coefficient 1.



For example, consider a Poisson regression model. Suppose the number of counts is known for theoretical reasons to be proportional to a predictor  $A$ . By using the log link function and by specifying  $\log(A)$  as an offset, you can force the model to satisfy this theoretical constraint.

**PredictorNames** — Names of predictors used to fit the model

cell array of character vectors

Names of predictors used to fit the model, stored as a cell array of character vectors.

**Residuals** — Residuals for fitted model

table

Residuals for the fitted model, stored as a table with one row for each observation and the following columns.

Field	Description
Raw	Observed minus fitted values.
LinearPredictor	Residuals on the linear predictor scale, equal to the adjusted response value minus the fitted linear combination of the predictors.
Pearson	Raw residuals divided by the estimated standard deviation of the response.
Anscombe	Residuals defined on transformed data with the transformation chosen to remove skewness.
Deviance	Residuals based on the contribution of each observation to the deviance.

To obtain any of these columns as a vector, index into the property using dot notation. For example, in a model `mdl`, the ordinary raw residual vector `r` is:

```
r = mdl.Residuals.Raw
```

Rows not used in the fit because of missing values (in `ObservationInfo.Missing`) contain NaN values.

Rows not used in the fit because of excluded values (in `ObservationInfo.Excluded`) contain NaN values, with the following exceptions:

- `raw` contains the difference between the observed and predicted values.

- `standardized` is the residual, standardized in the usual way.
- `studentized` matches the standardized values because this residual is not used in the estimate of the residual standard deviation.

**ResponseName — Response variable name**

character vector

Response variable name, stored as a character vector.

**Rsquared — R-squared value for the model**

structure

R-squared value for the model, stored as a structure.

For a linear or nonlinear model, `Rsquared` is a structure with two fields:

- `Ordinary` — Ordinary (unadjusted) R-squared
- `Adjusted` — R-squared adjusted for the number of coefficients

For a generalized linear model, `Rsquared` is a structure with five fields:

- `Ordinary` — Ordinary (unadjusted) R-squared
- `Adjusted` — R-squared adjusted for the number of coefficients
- `LLR` — Log-likelihood ratio
- `Deviance` — Deviance
- `AdjGeneralized` — Adjusted generalized R-squared

The R-squared value is the proportion of total sum of squares explained by the model.

The ordinary R-squared value relates to the `SSR` and `SST` properties:

$$\text{Rsquared} = \text{SSR}/\text{SST} = 1 - \text{SSE}/\text{SST}.$$

To obtain any of these values as a scalar, index into the property using dot notation. For example, the adjusted R-squared value in `mdl` is

```
r2 = mdl.Rsquared.Adjusted
```

**sse — Sum of squared errors**

numeric value

Sum of squared errors (residuals), stored as a numeric value.

The Pythagorean theorem implies

$$SST = SSE + SSR.$$

### **ssr** — Regression sum of squares

numeric value

Regression sum of squares, stored as a numeric value. The regression sum of squares is equal to the sum of squared deviations of the fitted values from their mean.

The Pythagorean theorem implies

$$SST = SSE + SSR.$$

### **sst** — Total sum of squares

numeric value

Total sum of squares, stored as a numeric value. The total sum of squares is equal to the sum of squared deviations of  $y$  from  $\text{mean}(y)$ .

The Pythagorean theorem implies

$$SST = SSE + SSR.$$

### **steps** — Stepwise fitting information

structure

Stepwise fitting information, stored as a structure with the following fields.

Field	Description
Start	Formula representing the starting model
Lower	Formula representing the lower bound model, these terms that must remain in the model
Upper	Formula representing the upper bound model, model cannot contain more terms than Upper
Criterion	Criterion used for the stepwise algorithm, such as 'sse'
PEnter	Value of the parameter, such as 0.05

Field	Description
PRemove	Value of the parameter, such as 0.10
History	Table representing the steps taken in the fit

The `History` table has one row for each step including the initial fit, and the following variables (columns).

Field	Description
Action	Action taken during this step, one of: <ul style="list-style-type: none"> <li>• 'Start' — First step</li> <li>• 'Add' — A term is added</li> <li>• 'Remove' — A term is removed</li> </ul>
TermName	<ul style="list-style-type: none"> <li>• 'Start' step: The starting model specification</li> <li>• 'Add' or 'Remove' steps: The term moved in that step</li> </ul>
Terms	Terms matrix (see <code>modelspec</code> of <code>fitlm</code> )
DF	Regression degrees of freedom after this step
delDF	Change in regression degrees of freedom from previous step (negative for steps that remove a term)
Deviance	Deviance (residual sum of squares) at that step
FStat	$F$ statistic that led to this step
PValue	$p$ -value of the $F$ statistic

The structure is empty unless you use `stepwiselm` or `stepwiseglm` to fit the model.

### **VariableInfo** — Information about input variables

table

Information about input variables contained in `Variables`, stored as a table with one row for each model term and the following columns.

Field	Description
Class	Character vector giving variable class, such as 'double'

Field	Description
Range	Cell array giving variable range: <ul style="list-style-type: none"> <li>• Continuous variable — Two-element vector <math>[min, max]</math>, the minimum and maximum values</li> <li>• Categorical variable — Cell array of distinct variable values</li> </ul>
InModel	Logical vector, where <code>true</code> indicates the variable is in the model
IsCategorical	Logical vector, where <code>true</code> indicates a categorical variable

**VariableNames — Names of variables used in fit**

cell array of character vectors

Names of variables used in fit, stored as a cell array of character vectors.

- If the fit is based on a table or dataset, this property provides the names of the variables in that table or dataset.
- If the fit is based on a predictor matrix and response vector, `VariableNames` is the values in the `VarNames` name-value pair of the fitting method.
- Otherwise the variables have the default fitting names.

**Variables — Data used to fit the model**

table

Data used to fit the model, stored as a table. `Variables` contains both observation and response values. If the fit is based on a table or dataset array, `Variables` contains all of the data from that table or dataset array. Otherwise, `Variables` is a table created from the input data matrix `X` and response vector `y`.

## Methods

<code>addTerms</code>	Add terms to generalized linear model
<code>compact</code>	Compact generalized linear regression model
<code>fit</code>	Create generalized linear regression model
<code>plotDiagnostics</code>	Plot diagnostics of generalized linear regression model
<code>plotResiduals</code>	Plot residuals of generalized linear regression model
<code>removeTerms</code>	Remove terms from generalized linear model
<code>step</code>	Improve generalized linear regression model by adding or removing terms
<code>stepwise</code>	Create generalized linear regression model by stepwise regression

## Inherited Methods

<code>coefCI</code>	Confidence intervals of coefficient estimates of generalized linear model
<code>coefTest</code>	Linear hypothesis test on generalized linear regression model coefficients
<code>devianceTest</code>	Analysis of deviance
<code>disp</code>	Display generalized linear regression model
<code>feval</code>	Evaluate generalized linear regression model prediction
<code>plotSlice</code>	Plot of slices through fitted generalized linear regression surface
<code>predict</code>	Predict response of generalized linear regression model
<code>random</code>	Simulate responses for generalized linear regression model

## Copy Semantics

Value. To learn how value classes affect copy operations, see [Copying Objects \(MATLAB\)](#).

## Examples

## Fit a Generalized Linear Model

Fit a logistic regression model of probability of smoking as a function of age, weight, and sex, using a two-way interactions model.

Load the hospital dataset array.

```
load hospital
ds = hospital; % just to use the ds name
```

Specify the model using a formula that allows up to two-way interactions.

```
modelspec = 'Smoker ~ Age*Weight*Sex - Age:Weight:Sex';
```

Create the generalized linear model.

```
mdl = fitglm(ds,modelspec,'Distribution','binomial')
```

```
mdl =
```

Generalized linear regression model:

```
logit(Smoker) ~ 1 + Sex*Age + Sex*Weight + Age*Weight
Distribution = Binomial
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	-6.0492	19.749	-0.3063	0.75938
Sex_Male	-2.2859	12.424	-0.18399	0.85402
Age	0.11691	0.50977	0.22934	0.81861
Weight	0.031109	0.15208	0.20455	0.83792
Sex_Male:Age	0.020734	0.20681	0.10025	0.92014
Sex_Male:Weight	0.01216	0.053168	0.22871	0.8191
Age:Weight	-0.00071959	0.0038964	-0.18468	0.85348

100 observations, 93 error degrees of freedom

Dispersion: 1

Chi^2-statistic vs. constant model: 5.07, p-value = 0.535

The large **P**-value indicates the model might not differ statistically from a constant.

### Create a Generalized Linear Model Stepwise

Create response data using just three of 20 predictors, and create a generalized linear model stepwise to see if it uses just the correct predictors.

Create data with 20 predictors, and Poisson response using just three of the predictors, plus a constant.

```
rng default % for reproducibility
X = randn(100,20);
mu = exp(X(:, [5 10 15]) * [.4; .2; .3] + 1);
y = poissrnd(mu);
```

Fit a generalized linear model using the Poisson distribution.

```
mdl = stepwiseglm(X,y,...
    'constant', 'upper', 'linear', 'Distribution', 'poisson')
```

1. Adding x5, Deviance = 134.439, Chi2Stat = 52.24814, PValue = 4.891229e-13
2. Adding x15, Deviance = 106.285, Chi2Stat = 28.15393, PValue = 1.1204e-07
3. Adding x10, Deviance = 95.0207, Chi2Stat = 11.2644, PValue = 0.000790094

```
mdl =
```

Generalized linear regression model:

```
log(y) ~ 1 + x5 + x10 + x15
Distribution = Poisson
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	1.0115	0.064275	15.737	8.4217e-56
x5	0.39508	0.066665	5.9263	3.0977e-09
x10	0.18863	0.05534	3.4085	0.0006532
x15	0.29295	0.053269	5.4995	3.8089e-08

100 observations, 96 error degrees of freedom

Dispersion: 1

Chi^2-statistic vs. constant model: 91.7, p-value = 9.61e-20

- “Generalized Linear Model Workflow” on page 12-41



## Definitions

### Canonical Link Function

The default link function for a generalized linear model is the canonical link function.

#### Canonical Link Functions for Generalized Linear Models

Distribution	Link Function Name	Link Function	Mean (Inverse) Function
'normal'	'identity'	$f(\mu) = \mu$	$\mu = Xb$
'binomial'	'logit'	$f(\mu) = \log(\mu/(1-\mu))$	$\mu = \exp(Xb) / (1 + \exp(Xb))$
'poisson'	'log'	$f(\mu) = \log(\mu)$	$\mu = \exp(Xb)$
'gamma'	-1	$f(\mu) = 1/\mu$	$\mu = 1/(Xb)$
'inverse gaussian'	-2	$f(\mu) = 1/\mu^2$	$\mu = (Xb)^{-1/2}$

### Hat Matrix

The hat matrix  $H$  is defined in terms of the data matrix  $X$  and a diagonal weight matrix  $W$ :

$$H = X(X^T W X)^{-1} X^T W^T.$$

$W$  has diagonal elements  $w_i$ :

$$w_i = \frac{g'(\mu_i)}{\sqrt{V(\mu_i)}},$$

where

- $g$  is the link function mapping  $y_i$  to  $x_i b$ .
- $g'$  is the derivative of the link function  $g$ .
- $V$  is the variance function.
- $\mu_i$  is the  $i$ th mean.

The diagonal elements  $H_{ii}$  satisfy

$$0 \leq h_{ii} \leq 1$$

$$\sum_{i=1}^n h_{ii} = p,$$

where  $n$  is the number of observations (rows of  $X$ ), and  $p$  is the number of coefficients in the regression model.

## Leverage

The leverage of observation  $i$  is the value of the  $i$ th diagonal term,  $h_{ii}$ , of the hat matrix  $H$ . Because the sum of the leverage values is  $p$  (the number of coefficients in the regression model), an observation  $i$  can be considered to be an outlier if its leverage substantially exceeds  $p/n$ , where  $n$  is the number of observations.

## Cook's Distance

The Cook's distance  $D_i$  of observation  $i$  is

$$D_i = w_i \frac{e_i^2}{p\hat{\phi}} \frac{h_{ii}}{(1-h_{ii})^2},$$

where

- $\hat{\phi}$  is the dispersion parameter (estimated or theoretical).
- $e_i$  is the linear predictor residual,  $g(y_i) - x_i\hat{\beta}$ , where
  - $g$  is the link function.
  - $y_i$  is the observed response.
  - $x_i$  is the observation.
- $\hat{\beta}$  is the estimated coefficient vector.
- $p$  is the number of coefficients in the regression model.
- $h_{ii}$  is the  $i$ th diagonal element of the Hat Matrix on page 32-2843  $H$ .

## Deviance

Deviance of a model  $M_1$  is twice the difference between the loglikelihood of that model and the saturated model,  $M_S$ . The saturated model is the model with the maximum number of parameters that can be estimated. For example, if there are  $n$  observations  $y_i$ ,  $i = 1, 2, \dots, n$ , with potentially different values for  $X_i^T\beta$ , then you can define a saturated model with  $n$  parameters. Let  $L(b,y)$  denote the maximum value of the likelihood function for a model. Then the deviance of model  $M_1$  is

$$-2(\log L(b_1, y) - \log L(b_S, y)),$$

where  $b_1$  are the estimated parameters for model  $M_1$  and  $b_S$  are the estimated parameters for the saturated model. The deviance has a chi-square distribution with  $n - p$  degrees of freedom, where  $n$  is the number of parameters in the saturated model and  $p$  is the number of parameters in model  $M_1$ .

If  $M_1$  and  $M_2$  are two different generalized linear models, then the fit of the models can be assessed by comparing the deviances  $D_1$  and  $D_2$  of these models. The difference of the deviances is

$$\begin{aligned} D &= D_2 - D_1 = -2(\log L(b_2, y) - \log L(b_S, y)) + 2(\log L(b_1, y) - \log L(b_S, y)) \\ &= -2(\log L(b_2, y) - \log L(b_1, y)). \end{aligned}$$

Asymptotically, this difference has a chi-square distribution with degrees of freedom  $v$  equal to the number of parameters that are estimated in one model but fixed (typically at 0) in the other. That is, it is equal to the difference in the number of parameters estimated in  $M_1$  and  $M_2$ . You can get the  $p$ -value for this test using `1 - chi2cdf(D, V)`, where  $D = D_2 - D_1$ .

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Only the `predict` and `random` functions support code generation.
- When you fit a model by using `fitglm` or `stepwiseglm`, the following restrictions apply.
  - Code generation does not support categorical predictors. You cannot supply training data in a table that contains at least one categorical predictor, and you cannot use the `'CategoricalVars'` name-value pair argument. To dummy-code variables that you want treated as categorical, preprocess the categorical data by using `dummyvar` before fitting the model..
  - The `Link`, `Derivative`, and `Inverse` fields of the `'Link'` name-value pair argument cannot be anonymous functions. That is, you cannot generate code using a generalized linear model that was created using anonymous functions for links. Instead, declare functions for link components.

## See Also

`LinearModel` | `NonLinearModel` | `fitglm` | `plotPartialDependence` | `stepwiseglm`

## Topics

“Generalized Linear Model Workflow” on page 12-41

“Generalized Linear Models” on page 12-12

**Introduced in R2012a**

# geocdf

Geometric cumulative distribution function

## Syntax

```
y = geocdf(x,p)
y = geocdf(x,p,'upper')
```

## Description

`y = geocdf(x,p)` returns the cumulative distribution function (cdf) of the geometric distribution at each value in `x` using the corresponding probabilities in `p`. `x` and `p` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array with the same dimensions as the other input. The parameters in `p` must lie on the interval  $[0, 1]$ .

`y = geocdf(x,p,'upper')` returns the complement of the geometric distribution cdf at each value in `x`, using an algorithm that more accurately computes the extreme upper tail probabilities.

## Examples

### Compute Geometric Distribution cdf

Suppose you toss a fair coin repeatedly, and a "success" occurs when the coin lands with heads facing up. What is the probability of observing three or fewer tails ("failures") before tossing a heads?

To solve, determine the value of the cumulative distribution function (cdf) for the geometric distribution at  $x$  equal to 3. The probability of success (tossing a heads)  $p$  in any given trial is 0.5.

```
x = 3;  
p = 0.5;  
y = geocdf(x,p)
```

```
y =  
  
    0.9375
```

The returned value of  $y$  indicates that the probability of observing three or fewer tails before tossing a heads is 0.9375.

## Definitions

### Geometric Distribution cdf

The cumulative distribution function (cdf) of the geometric distribution is

$$y = F(x | p) = 1 - (1 - p)^{x+1}; x = 0, 1, 2, \dots,$$

where  $p$  is the probability of success, and  $x$  is the number of failures before the first success. The result  $y$  is the probability of observing up to  $x$  trials before a success, when the probability of success in any given trial is  $p$ .

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

[cdf](#) | [geoinv](#) | [geopdf](#) | [geornd](#) | [geostat](#) | [mle](#)

## **Topics**

“Geometric Distribution” on page B-64

**Introduced before R2006a**

## geoinv

Geometric inverse cumulative distribution function

### Syntax

```
x = geoinv(y,p)
```

### Description

`x = geoinv(y,p)` returns the inverse cumulative distribution function (icdf) of the geometric distribution at each value in `y` using the corresponding probabilities in `p`.

`geoinv` returns the smallest positive integer `x` such that the geometric cdf evaluated at `x` is equal to or exceeds `y`. You can think of `y` as the probability of observing `x` successes in a row in independent trials, where `p` is the probability of success in each trial.

`y` and `p` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input for `p` or `y` is expanded to a constant array with the same dimensions as the other input. The values in `p` and `y` must lie on the interval  $[0, 1]$ .

### Examples

#### Compute Geometric Distribution icdf

Suppose the probability of a five-year-old car battery not starting in cold weather is 0.03. If we want no more than a ten percent chance that the car does not start, what is the maximum number of days in a row that we should try to start the car?

To solve, compute the inverse cdf of the geometric distribution. In this example, a "success" means the car does not start, while a "failure" means the car does start. The probability of success for each trial  $p$  equals 0.03, while the probability of observing  $x$  failures in a row before observing a success  $y$  equals 0.1.



```
y = 0.1;  
p = 0.03;  
x = geoinv(y,p)
```

```
x =  
  
    3
```

The returned result indicates that if we start the car three times, there is at least a ten percent chance that it will not start on one of those tries. Therefore, if we want no greater than a ten percent chance that the car will not start, we should only attempt to start it for a maximum of two days in a row.

We can confirm this result by evaluating the cdf at values of  $x$  equal to 2 and 3, given the probability of success for each trial  $p$  equal to 0.03.

```
y2 = geocdf(2,p) % cdf for x = 2  
y3 = geocdf(3,p) % cdf for x = 3
```

```
y2 =  
  
    0.0873
```

```
y3 =  
  
    0.1147
```

The returned results indicate an 8.7% chance of the car not starting if we try two days in a row, and an 11.5% chance of not starting if we try three days in a row.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

## See Also

`geocdf` | `geopdf` | `geornd` | `geostat` | `icdf`

## Topics

“Geometric Distribution” on page B-64

**Introduced before R2006a**

## geomean

Geometric mean

### Syntax

```
m = geomean(x)
geomean(X, dim)
```

### Description

`m = geomean(x)` calculates the geometric mean of a sample. For vectors, `geomean(x)` is the geometric mean of the elements in `x`. For matrices, `geomean(X)` is a row vector containing the geometric means of each column. For N-dimensional arrays, `geomean` operates along the first nonsingleton dimension of `X`.

`geomean(X, dim)` takes the geometric mean along the dimension `dim` of `X`.

The geometric mean is

$$m = \left[ \prod_{i=1}^n x_i \right]^{\frac{1}{n}}$$

### Examples

The arithmetic mean is greater than or equal to the geometric mean.

```
x = exprnd(1,10,6);
```

```
geometric = geomean(x)
```

```
geometric =
```

```
    0.7466    0.6061    0.6038    0.2569    0.7539    0.3478
```

```
average = mean(x)
```

```
average =  
    1.3509    1.1583    0.9741    0.5319    1.0088    0.8122
```

## Extended Capabilities

### Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays” (MATLAB).

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`harmmean` | `mean` | `median` | `trimmean`

### Topics

“Geometric Distribution” on page B-64

**Introduced before R2006a**

# geopdf

Geometric probability density function

## Syntax

```
y = geopdf(x,p)
```

## Description

`y = geopdf(x,p)` returns the probability density function (pdf) of the geometric distribution at each value in `x` using the corresponding probabilities in `p`. `x` and `p` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array with the same dimensions as the other input. The parameters in `p` must lie on the interval  $[0, 1]$ .

## Examples

### Compute Geometric Distribution pdf

Suppose you toss a fair coin repeatedly, and a "success" occurs when the coin lands with heads facing up. What is the probability of observing exactly three tails ("failures") before tossing a heads?

To solve, determine the value of the probability density function (pdf) for the geometric distributon at  $x$  equal to 3. The probability of success (tossing a heads)  $p$  in any given trial is 0.5.

```
x = 3;  
p = 0.5;  
y = geopdf(x,p)
```

```
y =
```

0.0625

The returned value of  $y$  indicates that the probability of observing exactly three tails before tossing a heads is 0.0625.

## Definitions

### Geometric Distribution pdf

The probability distribution function (pdf) of the geometric distribution is

$$y = f(x | p) = p(1 - p)^x \quad ; \quad x = 0, 1, 2, \dots,$$

where  $p$  is the probability of success, and  $x$  is the number of failures before the first success. The result  $y$  is the probability of observing exactly  $x$  trials before a success, when the probability of success in any given trial is  $p$ . For discrete distributions, the probability distribution function is also known as the probability mass function (pmf).

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

[geocdf](#) | [geoinv](#) | [geornd](#) | [geostat](#) | [mle](#) | [pdf](#)

### Topics

“Geometric Distribution” on page B-64

Introduced before R2006a

# geornd

Geometric random numbers

## Syntax

```
r = geornd(p)
r = geornd(p,m,n,...)
r = geornd(p,[m,n,...])
```

## Description

`r = geornd(p)` generates random numbers from a geometric distribution with probability parameter  $p$ .  $p$  can be a vector, a matrix, or a multidimensional array. The size of  $r$  is equal to the size of  $p$ . The parameters in  $p$  must lie in the interval  $[0, 1]$ .

`r = geornd(p,m,n,...)` or `r = geornd(p,[m,n,...])` generates a multidimensional  $m$ -by- $n$ -by-... array containing random numbers from the geometric distribution with probability parameter  $p$ .  $p$  can be a scalar or an array of the same size as  $r$ .

The geometric distribution is useful to model the number of failures before one success in a series of independent trials, where each trial results in either success or failure, and the probability of success in any individual trial is the constant  $p$ .

## Examples

### Generate Random Numbers from Geometric Distribution

Generate a single random number from a geometric distribution with probability parameter  $p$  equal to 0.01.

```
rng default % For reproducibility
p = 0.01;
r1 = geornd(0.01)
```

```
r1 =  
    20
```

The returned random number represents a single experiment in which 20 failures were observed before a success, where each independent trial has a probability of success  $p$  equal to 0.01.

Generate a 1-by-5 array of random numbers from a geometric distribution with probability parameter  $p$  equal to 0.01.

```
r2 = geornd(p,1,5)  
  
r2 =  
     9    205     9    45    231
```

Each random number in the returned array represents the result of an experiment to determine the number of failures observed before a success, where each independent trial has a probability of success  $p$  equal to 0.01.

Generate a 1-by-3 array containing one random number from each of the three geometric distributions corresponding to the parameters in the 1-by-3 array of probabilities  $p$ .

```
p = [0.01 0.1 0.5];  
r3 = geornd(p,[1 3])  
  
r3 =  
    127     5     0
```

Each element of the returned 1-by-3 array `r3` contains one random number generated from the geometric distribution described by the corresponding parameter in `P`. For example, the first element in `r3` represents an experiment in which 127 failures were observed before a success, where each independent trial has a probability of success  $p$  equal to 0.01. The second element in `r3` represents an experiment in which 5 failures were observed before a success, where each independent trial has a probability of success  $p$  equal to 0.1. The third element in `r3` represents an experiment in which zero failures



were observed before a success - in other words, the first attempt was a success - where each independent trial has a probability of success  $p$  equal to 0.5.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

The generated code can return a different sequence of numbers than MATLAB if either of the following is true:

- The output is nonscalar.
- An input parameter is invalid for the distribution.

### See Also

[geocdf](#) | [geoinv](#) | [geopdf](#) | [geostat](#) | [random](#)

### Topics

“Geometric Distribution” on page B-64

**Introduced before R2006a**

## geostat

Geometric mean and variance

### Syntax

```
[m,v] = geostat(p)
```

### Description

`[m,v] = geostat(p)` returns the mean  $m$  and variance  $v$  of a geometric distribution with corresponding probability parameters in  $p$ .  $p$  can be a vector, a matrix, or a multidimensional array. The parameters in  $p$  must lie in the interval  $[0, 1]$ .

### Examples

#### Compute Mean and Variance of Geometric Distribution

Define a probability vector that contains six different parameter values.

```
p = 1./(1:6)
```

```
p =
```

```
    1.0000    0.5000    0.3333    0.2500    0.2000    0.1667
```

Compute the mean and variance of the geometric distribution that corresponds to each value contained in probability vector.

```
[m,v] = geostat(1./(1:6))
```

```
m =
```

```

0      1.0000      2.0000      3.0000      4.0000      5.0000

v =
0      2.0000      6.0000     12.0000     20.0000     30.0000

```

The returned values indicate that, for example, the mean of a geometric distribution with probability parameter  $p$  equal to 1/3 is 2, and its variance is 6.

## Definitions

### Geometric Distribution Mean and Variance

The mean of the geometric distribution is

$$\text{mean} = \frac{1-p}{p},$$

and the variance of the geometric distribution is

$$\text{var} = \frac{1-p}{p^2},$$

where  $p$  is the probability of success.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`geocdf` | `geoinv` | `geopdf` | `geornd`

## **Topics**

“Geometric Distribution” on page B-64

**Introduced before R2006a**

# clustering.evaluation.GapEvaluation class

**Package:** clustering.evaluation

**Superclasses:**

Gap criterion clustering evaluation object

## Description

`clustering.evaluation.GapEvaluation` is an object consisting of sample data, clustering data, and gap criterion values used to evaluate the optimal number of clusters. Create a gap criterion clustering evaluation object using `evalclusters`.

## Construction

`eva = evalclusters(x, clust, 'Gap')` creates a gap criterion clustering evaluation object.

`eva = evalclusters(x, clust, 'Gap', Name, Value)` creates a gap criterion clustering evaluation object using additional options specified by one or more name-value pair arguments.

## Input Arguments

**x** — Input data

matrix

Input data, specified as an  $N$ -by- $P$  matrix.  $N$  is the number of observations, and  $P$  is the number of variables.

Data Types: `single` | `double`

**clust** — Clustering algorithm

'kmeans' | 'linkage' | 'gmdistribution' | matrix of clustering solutions | function handle

Clustering algorithm, specified as one of the following.

'kmeans'	Cluster the data in <code>x</code> using the <code>kmeans</code> clustering algorithm, with 'EmptyAction' set to 'singleton' and 'Replicates' set to 5.
'linkage'	Cluster the data in <code>x</code> using the <code>clusterdata</code> agglomerative clustering algorithm, with 'Linkage' set to 'ward'.
'gmdistribution'	Cluster the data in <code>x</code> using the <code>gmdistribution</code> Gaussian mixture distribution algorithm, with 'SharedCov' set to true and 'Replicates' set to 5.

If `Criterion` is 'CalinskiHarabasz', 'DaviesBouldin', or 'silhouette', you can specify a clustering algorithm using a function handle (MATLAB). The function must be of the form `C = clustfun(DATA, K)`, where `DATA` is the data to be clustered, and `K` is the number of clusters. The output of `clustfun` must be one of the following:

- A vector of integers representing the cluster index for each observation in `DATA`. There must be `K` unique values in this vector.
- A numeric  $n$ -by- $K$  matrix of score for  $n$  observations and  $K$  classes. In this case, the cluster index for each observation is determined by taking the largest score value in each row.

If `Criterion` is 'CalinskiHarabasz', 'DaviesBouldin', or 'silhouette', you can also specify `clust` as a  $n$ -by- $K$  matrix containing the proposed clustering solutions.  $n$  is the number of observations in the sample data, and  $K$  is the number of proposed clustering solutions. Column  $j$  contains the cluster indices for each of the  $N$  points in the  $j$ th clustering solution.

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: 'KList', [1:5], 'Distance', 'cityblock' specifies to test 1, 2, 3, 4, and 5 clusters using the sum of absolute differences distance measure.

### **B — Number of reference data sets**

100 (default) | positive integer value

Number of reference data sets generated from the reference distribution ReferenceDistribution, specified as the comma-separated pair consisting of 'B' and a positive integer value.

Example: 'B',150

### Distance — Distance metric

'sqEuclidean' (default) | 'Euclidean' | 'cityblock' | function | ...

Distance metric used for computing the criterion values, specified as the comma-separated pair consisting of 'Distance' and one of the following.

'sqEuclidean'	Squared Euclidean distance
'Euclidean'	Euclidean distance
'cityblock'	Sum of absolute differences
'cosine'	One minus the cosine of the included angle between points (treated as vectors)
'correlation'	One minus the sample correlation between points (treated as sequences of values)

For detailed information about each distance metric, see `pdist`.

You can also specify a function for the distance metric by using a function handle (MATLAB). The distance function must be of the form

```
d2 = distfun(XI,XJ),
```

where `XI` is a 1-by- $n$  vector corresponding to a single row of the input matrix `X`, and `XJ` is an  $m_2$ -by- $n$  matrix corresponding to multiple rows of `X`. `distfun` must return an  $m_2$ -by-1 vector of distances `d2`, whose  $k$ th element is the distance between `XI` and `XJ(k, :)`.

If `Criterion` is 'silhouette', you can also specify `Distance` as the output vector output created by the function `pdist`.

When `Clust` a character vector representing a built-in clustering algorithm, `evalclusters` uses the distance metric specified for `Distance` to cluster the data, except for the following:

- If `Clust` is 'linkage', and `Distance` is either 'sqEuclidean' or 'Euclidean', then the clustering algorithm uses Euclidean distance and Ward linkage.

- If `Clust` is `'linkage'` and `Distance` is any other metric, then the clustering algorithm uses the specified distance metric and average linkage.

In all other cases, the distance metric specified for `Distance` must match the distance metric used in the clustering algorithm to obtain meaningful results.

Example: `'Distance', 'Euclidean'`

### **KList — List of number of clusters to evaluate**

vector

List of number of clusters to evaluate, specified as the comma-separated pair consisting of `'KList'` and a vector of positive integer values. You must specify `KList` when `clust` is a clustering algorithm name or a function handle. When `criterion` is `'gap'`, `clust` must be a character vector or a function handle, and you must specify `KList`.

Example: `'KList', [1:6]`

### **ReferenceDistribution — Reference data generation method**

`'PCA'` (default) | `'uniform'`

Reference data generation method, specified as the comma-separated pair consisting of `'ReferenceDistributions'` and one of the following.

<code>'PCA'</code>	Generate reference data from a uniform distribution over a box aligned with the principal components of the data matrix <code>x</code> .
<code>'uniform'</code>	Generate reference data uniformly over the range of each feature in the data matrix <code>x</code> .

Example: `'ReferenceDistribution', 'uniform'`

### **SearchMethod — Method for selecting optimal number of clusters**

`'globalMaxSE'` (default) | `'firstMaxSE'`

Method for selecting the optimal number of clusters, specified as the comma-separated pair consisting of `'SearchMethod'` and one of the following.



'globalMaxSE'	Evaluate each proposed number of clusters in <code>KList</code> and select the smallest number of clusters satisfying
	$\text{Gap}(K) \geq \text{GAPMAX} - \text{SE}(\text{GAPMAX}),$
	where $K$ is the number of clusters, $\text{Gap}(K)$ is the gap value for the clustering solution with $K$ clusters, $\text{GAPMAX}$ is the largest gap value, and $\text{SE}(\text{GAPMAX})$ is the standard error corresponding to the largest gap value.
'firstMaxSE'	Evaluate each proposed number of clusters in <code>KList</code> and select the smallest number of clusters satisfying
	$\text{Gap}(K) \geq \text{Gap}(K + 1) - \text{SE}(K + 1),$
	where $K$ is the number of clusters, $\text{Gap}(K)$ is the gap value for the clustering solution with $K$ clusters, and $\text{SE}(K + 1)$ is the standard error of the clustering solution with $K + 1$ clusters.

Example: 'SearchMethod', 'globalMaxSE'

## Properties

### **B**

Number of data sets generated from the reference distribution, stored as a positive integer value.

### **ClusteringFunction**

Clustering algorithm used to cluster the input data, stored as a valid clustering algorithm name or function handle. If the clustering solutions are provided in the input, `ClusteringFunction` is empty.

### **CriterionName**

Name of the criterion used for clustering evaluation, stored as a valid criterion name.

**CriterionValues**

Criterion values corresponding to each proposed number of clusters in `InspectedK`, stored as a vector of numerical values.

**Distance**

Distance measure used for clustering data, stored as a valid distance measure name.

**ExpectedLogW**

Expectation of the natural logarithm of  $W$  based on the generated reference data, stored as a vector of scalar values.  $W$  is the within-cluster dispersion computed using the distance measurement `Distance`.

**InspectedK**

List of the number of proposed clusters for which to compute criterion values, stored as a vector of positive integer values.

**LogW**

Natural logarithm of  $W$  based on the input data, stored as a vector of scalar values.  $W$  is the within-cluster dispersion computed using the distance measurement `Distance`.

**Missing**

Logical flag for excluded data, stored as a column vector of logical values. If `Missing` equals `true`, then the corresponding value in the data matrix `x` is not used in the clustering solution.

**NumObservations**

Number of observations in the data matrix `X`, minus the number of missing (NaN) values in `X`, stored as a positive integer value.

**OptimalK**

Optimal number of clusters, stored as a positive integer value.

**OptimalY**

Optimal clustering solution corresponding to `OptimalK`, stored as a column vector of positive integer values. If the clustering solutions are provided in the input, `OptimalY` is empty.

**ReferenceDistribution**

Reference data generation method, stored as a valid reference distribution name.

**SE**

Standard error of the natural logarithm of  $W$  with respect to the reference data for each number of clusters in `InspectedK`, stored as a vector of scalar values.  $W$  is the within-cluster dispersion computed using the distance measurement `Distance`.

**SearchMethod**

Method for determining the optimal number of clusters, stored as a valid search method name.

**StdLogW**

Standard deviation of the natural logarithm of  $W$  with respect to the reference data for each number of clusters in `InspectedK`.  $W$  is the within-cluster dispersion computed using the distance measurement `Distance`.

**X**

Data used for clustering, stored as a matrix of numerical values.

## Methods

`increaseB`                      Increase reference data sets

## Inherited Methods

addK	Evaluate additional numbers of clusters
plot	Plot clustering evaluation object criterion values
compact	Compact clustering evaluation object

## Examples

### Evaluate the Clustering Solution Using Gap Criterion

Evaluate the optimal number of clusters using the gap clustering evaluation criterion.

Load the sample data.

```
load fisheriris
```

The data contains sepal and petal measurements from three species of iris flowers.

Evaluate the number of clusters based on the gap criterion values. Cluster the data using `kmeans`.

```
rng('default'); % For reproducibility
eva = evalclusters(meas, 'kmeans', 'gap', 'KList', [1:6])
```

```
eva =
```

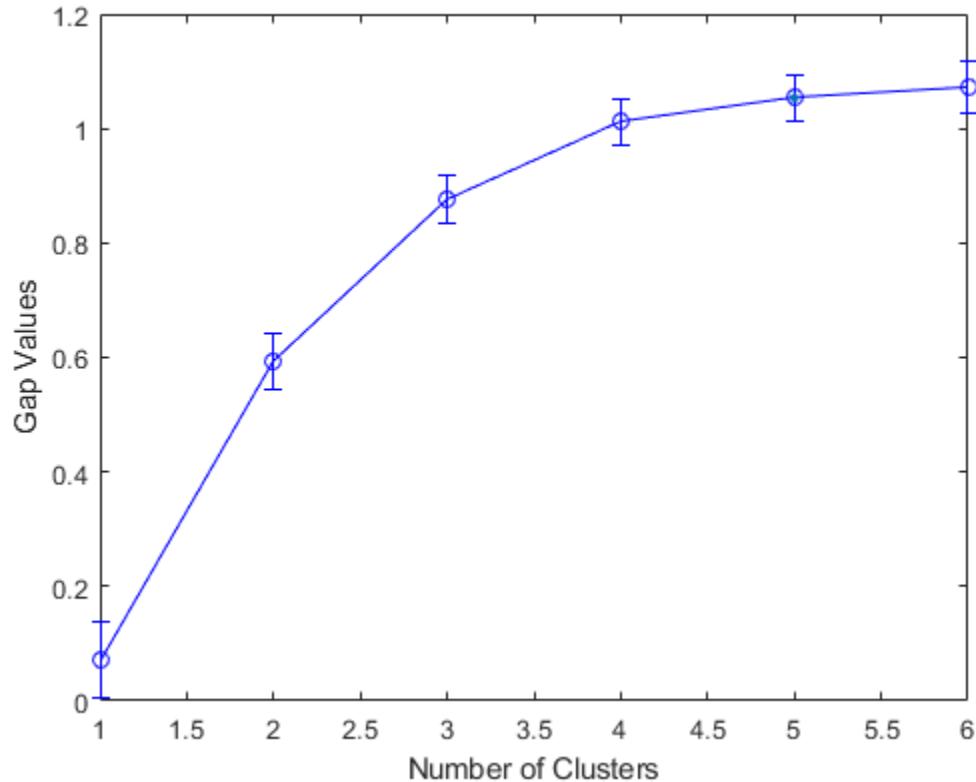
```
    GapEvaluation with properties:
```

```
    NumObservations: 150
    InspectedK: [1 2 3 4 5 6]
    CriterionValues: [0.0720 0.5928 0.8762 1.0114 1.0534 1.0720]
    OptimalK: 5
```

The `OptimalK` value indicates that, based on the gap criterion, the optimal number of clusters is four.

Plot the gap criterion values for each number of clusters tested.

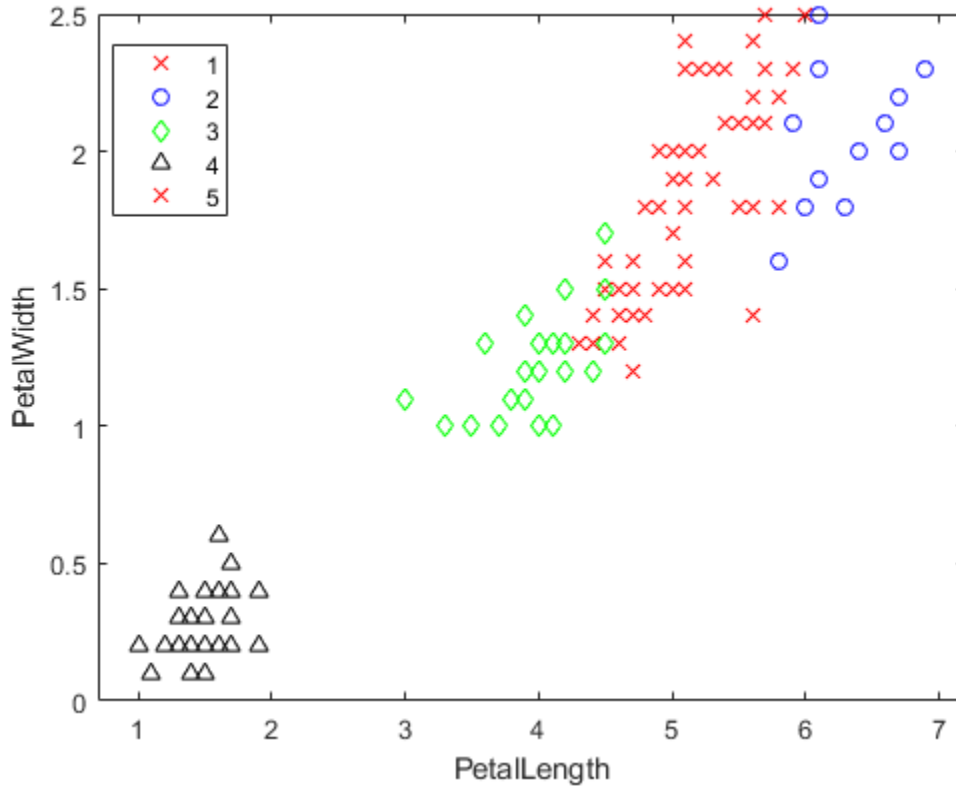
```
plot(eva)
```



Based on the plot, the maximum value of the gap criterion occurs at five clusters. However, the value at four clusters is within one standard error of the maximum, so the suggested optimal number of clusters is four.

Create a grouped scatter plot to examine the relationship between petal length and width. Group the data by suggested clusters.

```
figure
PetalLength = meas(:,3);
PetalWidth = meas(:,4);
ClusterGroup = eva.OptimalY;
gscatter(PetalLength,PetalWidth,ClusterGroup,'rbgk','xod^')
```



The plot shows cluster 1 in the lower-left corner, completely separated from the other three clusters. Cluster 1 contains flowers with the smallest petal widths and lengths. Cluster 4 is in the upper-right corner, and contains flowers with the largest petal widths

and lengths. Clusters 2 and 3 are near the center of the plot, and contain flowers with measurements between the two extremes.

## Definitions

### Gap Value

A common graphical approach to cluster evaluation involves plotting an error measurement versus several proposed numbers of clusters, and locating the “elbow” of this plot. The “elbow” occurs at the most dramatic decrease in error measurement. The gap criterion formalizes this approach by estimating the “elbow” location as the number of clusters with the largest gap value. Therefore, under the gap criterion, the optimal number of clusters occurs at the solution with the largest local or global gap value within a tolerance range.

The gap value is defined as

$$Gap_n(k) = E_n^* \{ \log(W_k) \} - \log(W_k),$$

where  $n$  is the sample size,  $k$  is the number of clusters being evaluated, and  $W_k$  is the pooled within-cluster dispersion measurement

$$W_k = \sum_{r=1}^k \frac{1}{2n_r} D_r,$$

where  $n_r$  is the number of data points in cluster  $r$ , and  $D_r$  is the sum of the pairwise distances for all points in cluster  $r$ .

The expected value  $E_n^* \{ \log(W_k) \}$  is determined by Monte Carlo sampling from a reference distribution, and  $\log(W_k)$  is computed from the sample data.

The gap value is defined even for clustering solutions that contain only one cluster, and can be used with any distance metric. However, the gap criterion is more computationally expensive than other cluster evaluation criteria, because the clustering algorithm must be applied to the reference data for each proposed clustering solution.

## References

- [1] Tibshirani, R., G. Walther, and T. Hastie. “Estimating the number of clusters in a data set via the gap statistic.” *Journal of the Royal Statistical Society: Series B*. Vol. 63, Part 2, 2001, pp. 411–423.

## See Also

`clustering.evaluation.CalinskiHarabaszEvaluation` |  
`clustering.evaluation.DaviesBouldinEvaluation` |  
`clustering.evaluation.SilhouetteEvaluation` | `evalclusters` |  
`evalclusters`

## Topics

Class Attributes (MATLAB)  
Property Attributes (MATLAB)



# get

**Class:** dataset

Access dataset array properties

---

**Note** The `dataset` data type might be removed in a future release. To work with heterogeneous data, use the MATLAB® `table` data type instead. See MATLAB `table` documentation for more information.

---

## Syntax

```
get(A)
s = get(A)
p = get(A,PropertyName)
p = get(A,{PropertyName1,PropertyName2,...})
```

## Description

`get(A)` displays a list of property/value pairs for the dataset array `A`.

`s = get(A)` returns the values in a scalar structure `s` with field names given by the properties.

`p = get(A,PropertyName)` returns the value of the property specified by `PropertyName`.

`p = get(A,{PropertyName1,PropertyName2,...})` allows multiple property names to be specified and returns their values in a cell array.

## Examples

Create a dataset array from Fisher's iris data and access the information.

```
load fisheriris
NumObs = size(meas,1);
NameObs = strcat({'Obs'}, num2str((1:NumObs)', '%-d'));
iris = dataset({nominal(species), 'species'}, ...
              {meas, 'SL', 'SW', 'PL', 'PW'}, ...
              'ObsNames', NameObs);

get(iris)
  Description: ''
  Units: {}
  DimNames: {'Observations' 'Variables'}
  UserData: []
  ObsNames: {150x1 cell}
  VarNames: {'species' 'SL' 'SW' 'PL' 'PW'}

ON = get(iris, 'ObsNames');
ON(1:3)
ans =
  'Obs1'
  'Obs2'
  'Obs3'
```

## See Also

[set](#) | [summary](#)

# getlabels

Access categorical array labels

---

**Note** The `nominal` and `ordinal` array data types might be removed in a future release. To represent ordered and unordered discrete, nonnumeric data, use the MATLAB® `categorical` data type instead.

---

## Syntax

```
labels = getlabels(A)
```

## Description

`labels = getlabels(A)` returns the labels of the levels in the nominal or ordinal array `A` as a cell array of character vectors, `labels`. If `A` is an ordinal array, `getlabels` returns the labels in the order of the levels.

## Examples

- “Change Category Labels” on page 2-10

## Input Arguments

**A** — Nominal or ordinal array

nominal array | ordinal array

Nominal or ordinal array, specified as a `nominal` or `ordinal` array object created using `nominal` or `ordinal`.

## See Also

`getlevels` | `nominal` | `ordinal`

## Topics

“Change Category Labels” on page 2-10

`nominal`

`ordinal`

**Introduced in R2007a**

# getlevels

Access categorical array levels

---

**Note** The `nominal` and `ordinal` array data types might be removed in a future release. To represent ordered and unordered discrete, nonnumeric data, use the MATLAB® `categorical` data type instead.

---

## Syntax

```
L = getlevels(A)
```

## Description

`L = getlevels(A)` returns the levels in the nominal or ordinal array `A` in `L`, a vector with the same type as `A`.

## Examples

- “Add and Drop Category Levels” on page 2-25
- “Merge Category Levels” on page 2-22
- “Reorder Category Levels” on page 2-12

## Input Arguments

**A** — Nominal or ordinal array

nominal array | ordinal array

Nominal or ordinal array, specified as a `nominal` or `ordinal` array object created using `nominal` or `ordinal`.

## See Also

`getlabels` | `nominal` | `ordinal`

## Topics

“Add and Drop Category Levels” on page 2-25

“Merge Category Levels” on page 2-22

“Reorder Category Levels” on page 2-12

`nominal`

`ordinal`

**Introduced in R2014b**

## gevcdf

Generalized extreme value cumulative distribution function

### Syntax

```
p = gevcdf(x,k,sigma,mu)
p = gevcdf(x,k,sigma,mu,'upper')
```

### Description

`p = gevcdf(x,k,sigma,mu)` returns the cdf of the generalized extreme value (GEV) distribution with shape parameter `k`, scale parameter `sigma`, and location parameter, `mu`, evaluated at the values in `x`. The size of `p` is the common size of the input arguments. A scalar input functions as a constant matrix of the same size as the other inputs.

`p = gevcdf(x,k,sigma,mu,'upper')` returns the complement of the cdf of the GEV distribution, using an algorithm that more accurately computes the extreme upper tail probabilities.

Default values for `k`, `sigma`, and `mu` are 0, 1, and 0, respectively.

When  $k < 0$ , the GEV is the type III extreme value distribution. When  $k > 0$ , the GEV distribution is the type II, or Frechet, extreme value distribution. If  $w$  has a Weibull distribution as computed by the `wblcdf` function, then  $-w$  has a type III extreme value distribution and  $1/w$  has a type II extreme value distribution. In the limit as  $k$  approaches 0, the GEV is the mirror image of the type I extreme value distribution as computed by the `evcdf` function.

The mean of the GEV distribution is not finite when  $k \geq 1$ , and the variance is not finite when  $k \geq 1/2$ . The GEV distribution has positive density only for values of  $x$  such that  $k*(X-mu)/sigma > -1$ .

## References

- [1] Embrechts, P., C. Klüppelberg, and T. Mikosch. *Modelling Extremal Events for Insurance and Finance*. New York: Springer, 1997.
- [2] Kotz, S., and S. Nadarajah. *Extreme Value Distributions: Theory and Applications*. London: Imperial College Press, 2000.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`cdf` | `gevfit` | `gevinv` | `gevlike` | `gevpdf` | `gevrnd` | `gevstat`

### Topics

“Generalized Extreme Value Distribution” on page B-53

Introduced before R2006a



# gevfit

Generalized extreme value parameter estimates

## Syntax

```
parmhat = gevfit(X)
[parmhat,parmci] = gevfit(X)
[parmhat,parmci] = gevfit(X,alpha)
[...] = gevfit(X,alpha,options)
```

## Description

`parmhat = gevfit(X)` returns maximum likelihood estimates of the parameters for the generalized extreme value (GEV) distribution given the data in `X`. `parmhat(1)` is the shape parameter, `k`, `parmhat(2)` is the scale parameter, `sigma`, and `parmhat(3)` is the location parameter, `mu`.

`[parmhat,parmci] = gevfit(X)` returns 95% confidence intervals for the parameter estimates.

`[parmhat,parmci] = gevfit(X,alpha)` returns 100(1-alpha)% confidence intervals for the parameter estimates.

`[...] = gevfit(X,alpha,options)` specifies control parameters for the iterative algorithm used to compute ML estimates. This argument can be created by a call to `statset`. See `statset('gevfit')` for parameter names and default values. Pass in `[]` for `alpha` to use the default values.

When  $k < 0$ , the GEV is the type III extreme value distribution. When  $k > 0$ , the GEV distribution is the type II, or Frechet, extreme value distribution. If  $w$  has a Weibull distribution as computed by the `wblfit` function, then  $-w$  has a type III extreme value distribution and  $1/w$  has a type II extreme value distribution. In the limit as  $k$  approaches 0, the GEV is the mirror image of the type I extreme value distribution as computed by the `evfit` function.

The mean of the GEV distribution is not finite when  $k \geq 1$ , and the variance is not finite when  $k \geq 1/2$ . The GEV distribution is defined for  $k * (X - \mu) / \sigma > -1$ .

## References

- [1] Embrechts, P., C. Klüppelberg, and T. Mikosch. *Modelling Extremal Events for Insurance and Finance*. New York: Springer, 1997.
- [2] Kotz, S., and S. Nadarajah. *Extreme Value Distributions: Theory and Applications*. London: Imperial College Press, 2000.

## See Also

`gevcdf` | `gevinv` | `gevlike` | `gevpdf` | `gevrnd` | `gevstat` | `mle`

## Topics

“Generalized Extreme Value Distribution” on page B-53

Introduced before R2006a

## gevinv

Generalized extreme value inverse cumulative distribution function

### Syntax

```
X = gevinv(P, k, sigma, mu)
```

### Description

`X = gevinv(P, k, sigma, mu)` returns the inverse cdf of the generalized extreme value (GEV) distribution with shape parameter `k`, scale parameter `sigma`, and location parameter `mu`, evaluated at the values in `P`. The size of `X` is the common size of the input arguments. A scalar input functions as a constant matrix of the same size as the other inputs.

Default values for `k`, `sigma`, and `mu` are 0, 1, and 0, respectively.

When  $k < 0$ , the GEV is the type III extreme value distribution. When  $k > 0$ , the GEV distribution is the type II, or Frechet, extreme value distribution. If  $w$  has a Weibull distribution as computed by the `wblinv` function, then  $-w$  has a type III extreme value distribution and  $1/w$  has a type II extreme value distribution. In the limit as  $k$  approaches 0, the GEV is the mirror image of the type I extreme value distribution as computed by the `evinv` function.

The mean of the GEV distribution is not finite when  $k \geq 1$ , and the variance is not finite when  $k \geq 1/2$ . The GEV distribution has positive density only for values of  $X$  such that  $k*(X-\mu)/\sigma > -1$ .

### References

- [1] Embrechts, P., C. Klüppelberg, and T. Mikosch. *Modelling Extremal Events for Insurance and Finance*. New York: Springer, 1997.

[2] Kotz, S., and S. Nadarajah. *Extreme Value Distributions: Theory and Applications*. London: Imperial College Press, 2000.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`gevcdf` | `gevfit` | `gevlake` | `gevpdf` | `gevrnd` | `gevstat` | `icdf`

### Topics

“Generalized Extreme Value Distribution” on page B-53

Introduced before R2006a

## gevlike

Generalized extreme value negative log-likelihood

### Syntax

```
nlogL = gevlike(params,data)
[nlogL,ACOV] = gevlike(params,data)
```

### Description

`nlogL = gevlike(params,data)` returns the negative of the log-likelihood `nlogL` for the generalized extreme value (GEV) distribution, evaluated at parameters `params`. `params(1)` is the shape parameter, `k`, `params(2)` is the scale parameter, `sigma`, and `params(3)` is the location parameter, `mu`.

`[nlogL,ACOV] = gevlike(params,data)` returns the inverse of Fisher's information matrix, `ACOV`. If the input parameter values in `params` are the maximum likelihood estimates, the diagonal elements of `ACOV` are their asymptotic variances. `ACOV` is based on the observed Fisher's information, not the expected information.

When  $k < 0$ , the GEV is the type III extreme value distribution. When  $k > 0$ , the GEV distribution is the type II, or Frechet, extreme value distribution. If  $w$  has a Weibull distribution as computed by the `wbllike` function, then  $-w$  has a type III extreme value distribution and  $1/w$  has a type II extreme value distribution. In the limit as  $k$  approaches 0, the GEV is the mirror image of the type I extreme value distribution as computed by the `evlike` function.

The mean of the GEV distribution is not finite when  $k \geq 1$ , and the variance is not finite when  $k \geq 1/2$ . The GEV distribution has positive density only for values of  $X$  such that  $k*(X-\mu)/\sigma > -1$ .

## References

- [1] Embrechts, P., C. Klüppelberg, and T. Mikosch. *Modelling Extremal Events for Insurance and Finance*. New York: Springer, 1997.
- [2] Kotz, S., and S. Nadarajah. *Extreme Value Distributions: Theory and Applications*. London: Imperial College Press, 2000.

## See Also

gevcdf | gevfit | gevinv | gevpdf | gevrnd | gevstat

## Topics

“Generalized Extreme Value Distribution” on page B-53

Introduced before R2006a

# gevpdf

Generalized extreme value probability density function

## Syntax

```
Y = gevpdf(X, k, sigma, mu)
```

## Description

`Y = gevpdf(X, k, sigma, mu)` returns the pdf of the generalized extreme value (GEV) distribution with shape parameter `k`, scale parameter `sigma`, and location parameter, `mu`, evaluated at the values in `X`. The size of `Y` is the common size of the input arguments. A scalar input functions as a constant matrix of the same size as the other inputs.

Default values for `k`, `sigma`, and `mu` are 0, 1, and 0, respectively.

When  $k < 0$ , the GEV is the type III extreme value distribution. When  $k > 0$ , the GEV distribution is the type II, or Frechet, extreme value distribution. If  $w$  has a Weibull distribution as computed by the `wblpdf` function, then  $-w$  has a type III extreme value distribution and  $1/w$  has a type II extreme value distribution. In the limit as  $k$  approaches 0, the GEV is the mirror image of the type I extreme value distribution as computed by the `evcdf` function.

The mean of the GEV distribution is not finite when  $k \geq 1$ , and the variance is not finite when  $k \geq 1/2$ . The GEV distribution has positive density only for values of  $x$  such that  $k*(X-\mu)/\sigma > -1$ .

## References

- [1] Embrechts, P., C. Klüppelberg, and T. Mikosch. *Modelling Extremal Events for Insurance and Finance*. New York: Springer, 1997.
- [2] Kotz, S., and S. Nadarajah. *Extreme Value Distributions: Theory and Applications*. London: Imperial College Press, 2000.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`gevcdf` | `gevfit` | `gevinv` | `gevlike` | `gevrnd` | `gevstat` | `pdf`

### Topics

“Generalized Extreme Value Distribution” on page B-53

Introduced before R2006a



# gevrnd

Generalized extreme value random numbers

## Syntax

```
R = gevrnd(k, sigma, mu)
R = gevrnd(k, sigma, mu, m, n, ...)
R = gevrnd(k, sigma, mu, [m, n, ...])
```

## Description

`R = gevrnd(k, sigma, mu)` returns an array of random numbers chosen from the generalized extreme value (GEV) distribution with shape parameter  $k$ , scale parameter  $\sigma$ , and location parameter,  $\mu$ . The size of  $R$  is the common size of the input arguments if all are arrays. If any parameter is a scalar, the size of  $R$  is the size of the other parameters.

`R = gevrnd(k, sigma, mu, m, n, ...)` or `R = gevrnd(k, sigma, mu, [m, n, ...])` generates an  $m$ -by- $n$ -by-... array containing random numbers from the GEV distribution with parameters  $k$ ,  $\sigma$ , and  $\mu$ . The  $k$ ,  $\sigma$ ,  $\mu$  parameters can each be scalars or arrays of the same size as  $R$ .

When  $k < 0$ , the GEV is the type III extreme value distribution. When  $k > 0$ , the GEV distribution is the type II, or Frechet, extreme value distribution. If  $w$  has a Weibull distribution as computed by the `wblrnd` function, then  $-w$  has a type III extreme value distribution and  $1/w$  has a type II extreme value distribution. In the limit as  $k$  approaches 0, the GEV is the mirror image of the type I extreme value distribution as computed by the `evrnd` function.

The mean of the GEV distribution is not finite when  $k \geq 1$ , and the variance is not finite when  $k \geq 1/2$ . The GEV distribution has positive density only for values of  $X$  such that  $k*(X-\mu)/\sigma > -1$ .

## References

- [1] Embrechts, P., C. Klüppelberg, and T. Mikosch. *Modelling Extremal Events for Insurance and Finance*. New York: Springer, 1997.
- [2] Kotz, S., and S. Nadarajah. *Extreme Value Distributions: Theory and Applications*. London: Imperial College Press, 2000.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

The generated code can return a different sequence of numbers than MATLAB if either of the following is true:

- The output is nonscalar.
- An input parameter is invalid for the distribution.

### See Also

`gevcdf` | `gevfit` | `gevinv` | `gevlike` | `gevpdf` | `gevstat` | `random`

### Topics

“Generalized Extreme Value Distribution” on page B-53

**Introduced before R2006a**

## gevstat

Generalized extreme value mean and variance

### Syntax

```
[M,V] = gevstat(k,sigma,mu)
```

### Description

`[M,V] = gevstat(k,sigma,mu)` returns the mean of and variance for the generalized extreme value (GEV) distribution with shape parameter  $k$ , scale parameter  $\sigma$ , and location parameter,  $\mu$ . The sizes of  $M$  and  $V$  are the common size of the input arguments. A scalar input functions as a constant matrix of the same size as the other inputs.

Default values for  $k$ ,  $\sigma$ , and  $\mu$  are 0, 1, and 0, respectively.

When  $k < 0$ , the GEV is the type III extreme value distribution. When  $k > 0$ , the GEV distribution is the type II, or Frechet, extreme value distribution. If  $w$  has a Weibull distribution as computed by the `wblstat` function, then  $-w$  has a type III extreme value distribution and  $1/w$  has a type II extreme value distribution. In the limit as  $k$  approaches 0, the GEV is the mirror image of the type I extreme value distribution as computed by the `evstat` function.

The mean of the GEV distribution is not finite when  $k \geq 1$ , and the variance is not finite when  $k \geq 1/2$ . The GEV distribution has positive density only for values of  $x$  such that  $k*(X-\mu)/\sigma > -1$ .

### References

- [1] Embrechts, P., C. Klüppelberg, and T. Mikosch. *Modelling Extremal Events for Insurance and Finance*. New York: Springer, 1997.
- [2] Kotz, S., and S. Nadarajah. *Extreme Value Distributions: Theory and Applications*. London: Imperial College Press, 2000.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`gevpdf` | `gevfit` | `gevinv` | `gevlike` | `gevpdf` | `gevrnd`

### Topics

“Generalized Extreme Value Distribution” on page B-53

Introduced before R2006a

## gline

Interactively add line to plot

### Syntax

```
gline(h)
gline
hline = gline(...)
```

### Description

`gline(h)` allows you to draw a line segment in the figure with handle `h` by clicking the pointer at the two endpoints. A rubber-band line tracks the pointer movement.

`gline` with no input arguments defaults to `h = gcf` and draws in the current figure.

`hline = gline(...)` returns the handle `hline` to the line.

### Examples

Use `gline` to connect two points in a plot:

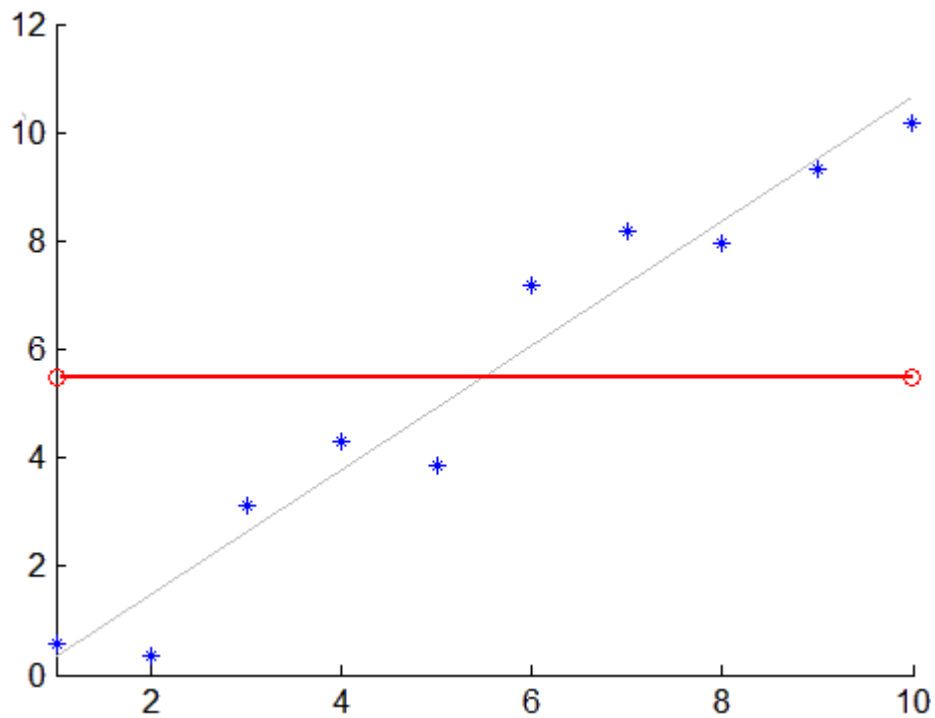
```
x = 1:10;

y = x + randn(1,10);
scatter(x,y,25,'b','*')

lsline

mu = mean(y);
hold on
plot([1 10],[mu mu],'ro')

hline = gline; % Connect circles
set(hline,'Color','r')
```



## See Also

`lsline` | `refcurve` | `refline`

Introduced before R2006a

# glmfit

Generalized linear model regression

## Syntax

```
b = glmfit(X,y,distr)
b = glmfit(X,y,distr,param1, val1,param2, val2,...)
[b,dev] = glmfit(...)
[b,dev,stats] = glmfit(...)
```

## Description

`b = glmfit(X,y,distr)` returns a  $(p + 1)$ -by-1 vector `b` of coefficient estimates for a generalized linear regression of the responses in `y` on the predictors in `X`, using the distribution `distr`. `X` is an  $n$ -by- $p$  matrix of  $p$  predictors at each of  $n$  observations. `distr` can be any of the following: 'binomial', 'gamma', 'inverse gaussian', 'normal' (the default), and 'poisson'.

In most cases, `y` is an  $n$ -by-1 vector of observed responses. For the binomial distribution, `y` can be a binary vector indicating success or failure at each observation, or a two column matrix with the first column indicating the number of successes for each observation and the second column indicating the number of trials for each observation.

This syntax uses the canonical link (see below) to relate the distribution to the predictors.

---

**Note** By default, `glmfit` adds a first column of 1s to `X`, corresponding to a constant term in the model. Do not enter a column of 1s directly into `X`. You can change the default behavior of `glmfit` using the 'constant' parameter, below.

---

`glmfit` treats NaNs in either `X` or `y` as missing values, and ignores them.

`b = glmfit(X, y, distr, param1, val1, param2, val2, ...)` additionally allows you to specify optional parameter name/value pairs to control the model fit. Acceptable parameters are as follows.

Parameter	Value	Description
'link'	'identity', default for the distribution 'normal'	$\mu = Xb$
	'log', default for the distribution 'poisson'	$\log(\mu) = Xb$
	'logit', default for the distribution 'binomial'	$\log(\mu/(1 - \mu)) = Xb$
	'probit'	$\text{norminv}(\mu) = Xb$
	'comploglog'	$\log(-\log(1 - \mu)) = Xb$
	'reciprocal', default for the distribution 'gamma'	$1/\mu = Xb$
	'loglog'	$\log(-\log(\mu)) = Xb$
	p (a number), default for the distribution 'inverse gaussian' (with $p = -2$ )	$\mu^p = Xb$
	cell array of the form {FL FD FI}, containing three function handles, created using @, that define the link (FL), the derivative of the link (FD), and the inverse link (FI).	Custom-defined link function. You must provide <ul style="list-style-type: none"> <li>• FL(mu)</li> <li>• FD = dFL(mu) / dmu</li> <li>• FI = FL^(-1)</li> </ul>



Parameter	Value	Description
	structure array having these fields: <ul style="list-style-type: none"> <li>• 'Link' — Link function</li> <li>• 'Derivative' — Derivative of the link function</li> <li>• 'Inverse' — Inverse of the link function</li> </ul> The value of each field is a character vector corresponding to a function that is on the path or a function handle (created using @).	Custom-defined link function, its derivative, and its inverse.
'estdisp'	'on'	Estimates a dispersion parameter for the binomial or Poisson distribution.
	'off' (Default for binomial or Poisson distribution)	Uses the theoretical value of 1.0 for those distributions.
'offset'	Vector	Used as an additional predictor variable, but with a coefficient value fixed at 1.0.
'weights'	Vector of prior weights, such as the inverses of the relative variance of each observation	
'constant'	'on' (default)	Includes a constant term in the model. The coefficient of the constant term is the first element of $\mathbf{b}$ .
	'off'	Omit the constant term.

`[b, dev] = glmfit(...)` returns `dev`, the deviance of the fit at the solution vector. The deviance is a generalization of the residual sum of squares. It is possible to perform an analysis of deviance to compare several models, each a subset of the other, and to test whether the model with more terms is significantly better than the model with fewer terms.

`[b, dev, stats] = glmfit(...)` returns `dev` and `stats`.

`stats` is a structure with the following fields:

- `beta` — Coefficient estimates `b`
- `dfe` — Degrees of freedom for error
- `sfit` — Estimated dispersion parameter
- `s` — Theoretical or estimated dispersion parameter
- `estdisp` — 0 when the `'estdisp'` name-value pair argument value is `'off'` and 1 when the `'estdisp'` name-value pair argument value is `'on'`.
- `covb` — Estimated covariance matrix for `B`
- `se` — Vector of standard errors of the coefficient estimates `b`
- `coeffcorr` — Correlation matrix for `b`
- `t` —  $t$  statistics for `b`
- `p` —  $p$ -values for `b`
- `resid` — Vector of residuals
- `residp` — Vector of Pearson residuals
- `residd` — Vector of deviance residuals
- `resida` — Vector of Anscombe residuals

If you estimate a dispersion parameter for the binomial or Poisson distribution, then `stats.s` is set equal to `stats.sfit`. Also, the elements of `stats.se` differ by the factor `stats.s` from their theoretical values.

## Examples

## Fit Generalized Linear Model with Probit Link

Enter the sample data.

```
x = [2100 2300 2500 2700 2900 3100 ...  
     3300 3500 3700 3900 4100 4300]';  
n = [48 42 31 34 31 21 23 23 21 16 17 21]';  
y = [1 2 0 3 8 8 14 17 19 15 17 21]';
```

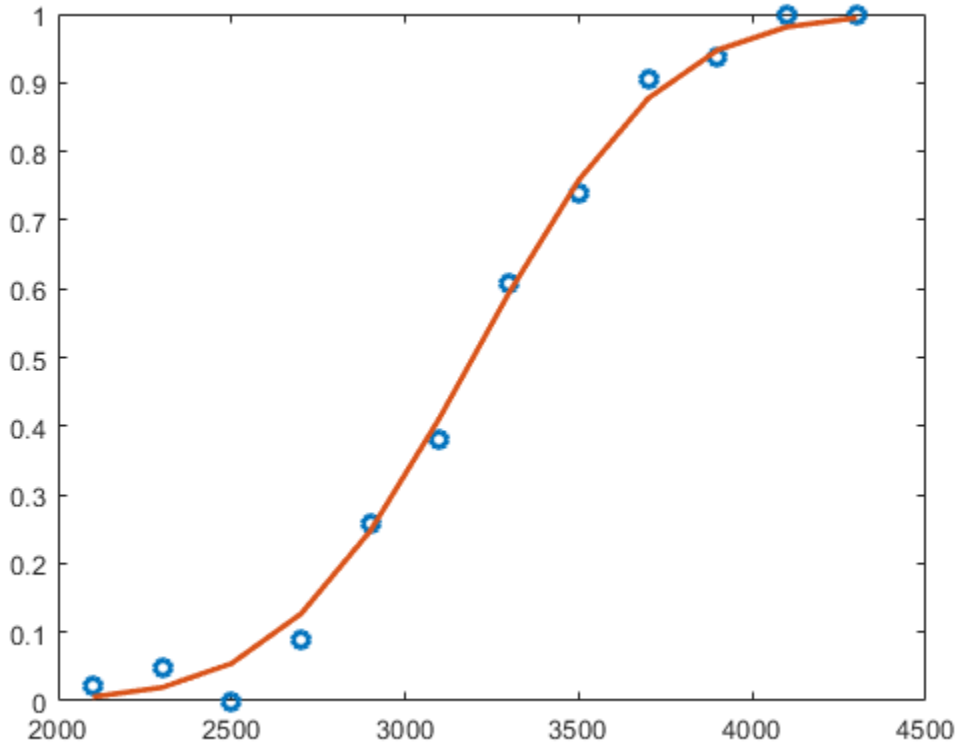
Each  $y$  value is the number of successes in the corresponding number of trials in  $n$ , and  $x$  contains the predictor variable values.

Fit a probit regression model for  $y$  on  $x$ .

```
b = glmfit(x, [y n], 'binomial', 'link', 'probit');
```

Compute the estimated number of successes and plot the percent observed and estimated percent success versus the  $x$  values.

```
yfit = glmval(b, x, 'probit', 'size', n);  
plot(x, y./n, 'o', x, yfit./n, '-', 'LineWidth', 2)
```



### Use Custom-Defined Link Function

Load the sample data.

```
load fisheriris
```

The column vector, `species`, consists of iris flowers of three different species, `setosa`, `versicolor`, `virginica`. The double matrix `meas` consists of four types of measurements on the flowers, the length and width of sepals and petals in centimeters, respectively.

Define the response and predictor variables.

```
X = meas(51:end, :);  
y = strcmp('versicolor', species(51:end));
```

Define three function handles, created using @, that define the link, the derivative of the link, and the inverse link for a logit link function, and store them in a cell array.

```
link = @(mu) log(mu ./ (1-mu));  
derlink = @(mu) 1 ./ (mu .* (1-mu));  
invlink = @(resp) 1 ./ (1 + exp(-resp));  
F = {link, derlink, invlink};
```

Fit a logistic regression using glmfit with the link function you defined.

```
b = glmfit(X,y, 'binomial', 'link', F)
```

```
b =  
  
    42.6378  
     2.4652  
     6.6809  
    -9.4294  
   -18.2861
```

Now, fit a generalized linear model using the logit link function and compare the results.

```
b = glmfit(X,y, 'binomial', 'link', 'logit')
```

```
b =  
  
    42.6378  
     2.4652  
     6.6809  
    -9.4294
```

-18.2861

## References

- [1] Dobson, A. J. *An Introduction to Generalized Linear Models*. New York: Chapman & Hall, 1990.
- [2] McCullagh, P., and J. A. Nelder. *Generalized Linear Models*. New York: Chapman & Hall, 1990.
- [3] Collett, D. *Modeling Binary Data*. New York: Chapman & Hall, 2002.

## See Also

`GeneralizedLinearModel` | `fitglm` | `glmval` | `regress` | `regstats` | `stepwiseglm`

**Introduced before R2006a**

# glmval

Generalized linear model values

## Syntax

```
yhat = glmval(b,X,link)
[yhat,dylo,dyhi] = glmval(b,X,link,stats)
[...] = glmval(...,param1, val1,param2, val2, ...)
```

## Description

`yhat = glmval(b,X,link)` computes predicted values for the generalized linear model with link function `link` and predictors `X`. Distinct predictor variables should appear in different columns of `X`. `b` is a vector of coefficient estimates as returned by the `glmfit` function. `link` can be any of the character vectors or custom-defined link functions used as values for the 'link' name-value pair argument in the `glmfit` function.

---

**Note** By default, `glmval` adds a first column of 1s to `X`, corresponding to a constant term in the model. Do not enter a column of 1s directly into `X`. You can change the default behavior of `glmval` using the 'constant' parameter, below.

---

`[yhat,dylo,dyhi] = glmval(b,X,link,stats)` also computes 95% confidence bounds for the predicted values. When the `stats` structure output of the `glmfit` function is specified, `dylo` and `dyhi` are also returned. `dylo` and `dyhi` define a lower confidence bound of `yhat-dylo`, and an upper confidence bound of `yhat+dyhi`. Confidence bounds are nonsimultaneous, and apply to the fitted curve, not to a new observation.

`[...] = glmval(...,param1, val1,param2, val2, ...)` specifies optional parameter name/value pairs to control the predicted values. Acceptable parameters are:

Parameter	Value
'confidence' — the confidence level for the confidence bounds	A scalar between 0 and 1
'size' — the size parameter (N) for a binomial model	A scalar, or a vector with one value for each row of X
'offset' — used as an additional predictor variable, but with a coefficient value fixed at 1.0	A vector
'constant'	<ul style="list-style-type: none"> <li>'on' — Includes a constant term in the model. The coefficient of the constant term is the first element of b.</li> <li>'off' — Omit the constant term</li> </ul>
'simultaneous' — Compute simultaneous confidence intervals (true), or compute non-simultaneous confidence intervals (default false)	true or false

## Examples

### Fit Generalized Linear Model with Probit Link

Enter the sample data.

```
x = [2100 2300 2500 2700 2900 3100 ...
      3300 3500 3700 3900 4100 4300]';
n = [48 42 31 34 31 21 23 23 21 16 17 21]';
y = [1 2 0 3 8 8 14 17 19 15 17 21]';
```

Each  $y$  value is the number of successes in the corresponding number of trials in  $n$ , and  $x$  contains the predictor variable values.

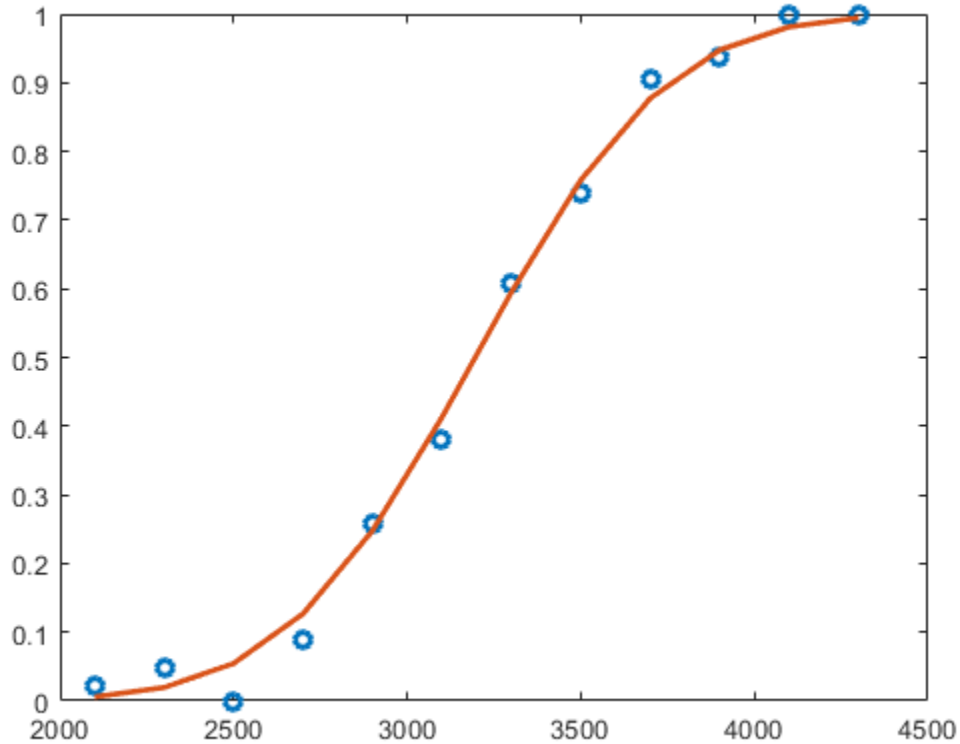
Fit a probit regression model for  $y$  on  $x$ .

```
b = glmfit(x,[y n], 'binomial', 'link', 'probit');
```

Compute the estimated number of successes and plot the percent observed and estimated percent success versus the  $x$  values.



```
yfit = glmval(b,x,'probit','size',n);
plot(x, y./n,'o',x,yfit./n,'-', 'LineWidth',2)
```



### Use Custom-Defined Link Function

Enter sample data.

```
x = [2100 2300 2500 2700 2900 3100 ...
      3300 3500 3700 3900 4100 4300]';
n = [48 42 31 34 31 21 23 23 21 16 17 21]';
y = [1 2 0 3 8 8 14 17 19 15 17 21]';
```

Each  $y$  value is the number of successes in corresponding number of trials in  $n$ , and  $x$  contains the predictor variable values.

Now define three function handles, created using `@`, that define the link, the derivative of the link, and the inverse link for a probit link function, and store them in a cell array.

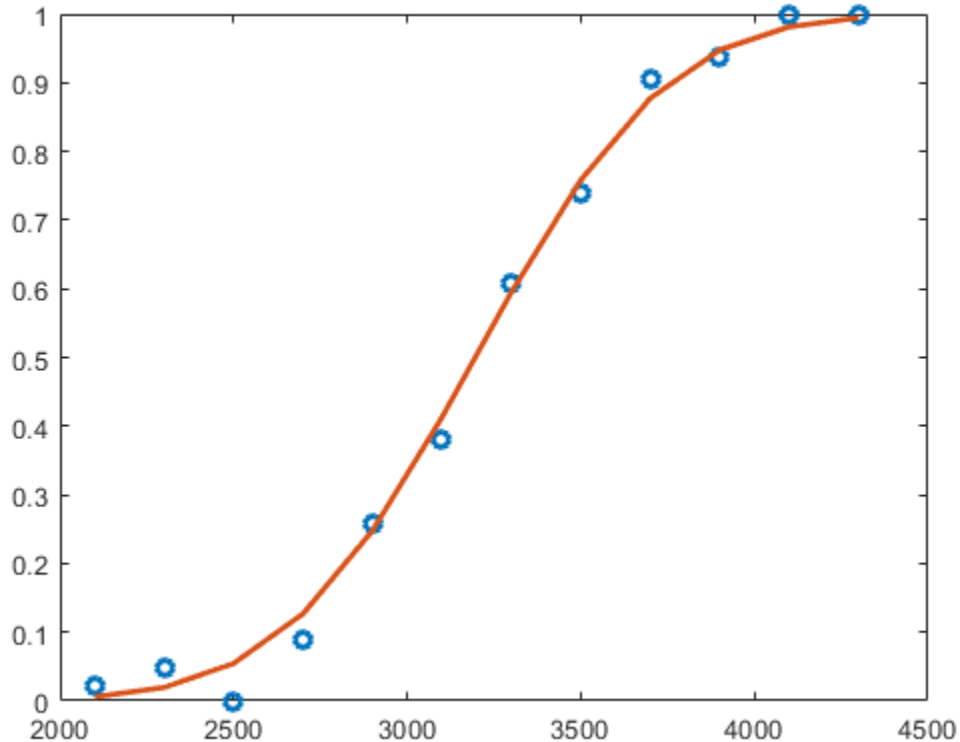
```
link = @(mu) norminv(mu);  
derlink = @(mu) 1 ./ normpdf(norminv(mu));  
invlink = @(resp) normcdf(resp);  
F = {link, derlink, invlink};
```

Fit a generalized linear model for  $y$  on  $x$  using the link function you defined.

```
b = glmfit(x, [y n], 'binomial', 'link', F);
```

Compute the estimated number of successes and plot the observed and estimated percent success versus the  $x$  values.

```
yfit = glmval(b, x, F, 'size', n);  
plot(x, y./n, 'o', x, yfit./n, '-', 'LineWidth', 2)
```



### Generate Code from Function That Predicts Responses Given New Data

Train a generalized linear model, and then generate code from a function that classifies new observations based on the model. This example is based on the “Use Custom-Defined Link Function” on page 32-2907 example.

Enter the sample data.

```
x = [2100 2300 2500 2700 2900 3100 ...  
     3300 3500 3700 3900 4100 4300]';  
n = [48 42 31 34 31 21 23 23 21 16 17 21]';  
y = [1 2 0 3 8 8 14 17 19 15 17 21]';
```

Suppose that the inverse normal pdf is an appropriate link function for the problem.

Declare a function named `myInvNorm.m` that accepts values of  $X\beta$  and returns corresponding values of the inverse of the standard normal cdf.

```
function in = myInvNorm(mu) %#codegen
%myInvNorm Inverse of standard normal cdf for code generation
% myInvNorm is a GLM link function that accepts a numeric vector mu, and
% returns in, which is a numeric vector of corresponding values of the
% inverse of the standard normal cdf.
%
in = norminv(mu);
end
```

Declare another function named `myDInvNorm.m` that accepts values of  $X\beta$  and returns corresponding values of the derivative of the link function.

```
function din = myDInvNorm(mu) %#codegen
%myDInvNorm Derivative of inverse of standard normal cdf for code
%generation
% myDInvNorm corresponds to the derivative of the GLM link function
% myInvNorm. myDInvNorm accepts a numeric vector mu, and returns din,
% which is a numeric vector of corresponding derivatives of the inverse
% of the standard normal cdf.
%
din = 1./normpdf(norminv(mu));
end
```

Declare another function named `myInvInvNorm.m` that accepts values of  $X\beta$  and returns corresponding values of the inverse of the link function.

```
function iin = myInvInvNorm(mu) %#codegen
%myInvInvNorm Standard normal cdf for code generation
% myInvInvNorm is the inverse of the GLM link function myInvNorm.
% myInvInvNorm accepts a numeric vector mu, and returns iin, which is a
% numeric vector of corresponding values of the standard normal cdf.
%
```

```
iin = normcdf(mu);
end
```

Create a structure array that specifies each of the link functions. Specifically, the structure array contains fields named 'Link', 'Derivative', and 'Inverse'. The corresponding values are the names of the functions.

```
link = struct('Link','myInvNorm','Derivative','myDInvNorm',...
             'Inverse','myInvInvNorm')
```

```
link =
    struct with fields:
        Link: 'myInvNorm'
        Derivative: 'myDInvNorm'
        Inverse: 'myInvInvNorm'
```

Fit a GLM for  $y$  on  $x$  using the link function `link`. Additionally, return the structure array of statistics.

```
[b,~,stats] = glmfit(x,[y n],'binomial','link',link);
```

`b` is a 2-by-1 vector of regression coefficients.

In your current working folder, declare a function called `classifyGLM.m` that:

- Accepts measurements with columns corresponding to those in `x`, regression coefficients whose dimensions correspond to `b`, a link function, the structure of GLM statistics, and any valid `glmval` name-value pair argument
- Returns predictions and confidence interval margins of error

```
function [yhat,lo,hi] = classifyGLM(b,x,link,varargin) %#codegen
%CLASSIFYGLM Classify measurements using GLM model
% CLASSIFYGLM classifies the n observations in the n-by-1 vector x using
% the GLM model with regression coefficients b and link function link,
% and then returns the n-by-1 vector of predicted values in yhat.
% CLASSIFYGLM also returns margins of error for the predictions using
% additional information in the GLM statistics structure stats.
```

```
narginchk(3, Inf);
if(isstruct(varargin{1}))
    stats = varargin{1};
    [yhat, lo, hi] = glmval(b, x, link, stats, varargin{2:end});
else
    yhat = glmval(b, x, link, varargin{:});
end
end
```

Generate a MEX function from `classifyGLM.m`. Because C uses static typing, `codegen` must determine the properties of all variables in MATLAB® files at compile time. To ensure that the MEX function can use the same inputs, use the `-args` argument to specify the following in the order given:

- Regression coefficients `b` as a compile-time constant
- In-sample observations `x`
- Link function as a compile-time constant
- Resulting GLM statistics as a compile-time constant
- Name 'Confidence' as a compile-time constant
- Confidence level 0.9

To designate arguments as compile-time constants, use `coder.Constant`.

```
codegen -config:mex classifyGLM -args {coder.Constant(b), x, coder.Constant(link), coder.C
```

`codegen` generates the MEX file `classifyGLM_mex.mexw64` in your current folder. The file extension depends on your platform.

Compare predictions using `glmval` and `classifyGLM_mex`. Specify name-value pair arguments in the same order as in the `-args` argument in the call to `codegen`.

```
[yhat1, melo1, mehi1] = glmval(b, x, link, stats, 'Confidence', 0.9);
[yhat2, melo2, mehi2] = classifyGLM_mex(b, x, link, stats, 'Confidence', 0.9);

comp1 = (yhat1 - yhat2)'*(yhat1 - yhat2);
agree1 = comp1 < eps
comp2 = (melo1 - melo2)'*(melo1 - melo2);
agree2 = comp2 < eps
comp3 = (mehi1 - mehi2)'*(mehi1 - mehi2);
agree3 = comp3 < eps
```

```
agree1 =  
    logical  
    1  
  
agree2 =  
    logical  
    1  
  
agree3 =  
    logical  
    1
```

The generated MEX function produces the same results as `predict`.

## References

- [1] Dobson, A. J. *An Introduction to Generalized Linear Models*. New York: Chapman & Hall, 1990.
- [2] McCullagh, P., and J. A. Nelder. *Generalized Linear Models*. New York: Chapman & Hall, 1990.
- [3] Collett, D. *Modeling Binary Data*. New York: Chapman & Hall, 2002.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- `glmval` generates reference C code.
- Call `glmval` within a function that you declare (that is, you cannot call `glmval` at the top level).
- This table contains additional input-and-output-argument notes and limitations.

Argument	Notes and Limitations
<code>b</code>	<code>b</code> must be a compile-time constant. That is, its value cannot change while <code>codegen</code> generates the C code. You can designate variables as compile-time constants using <code>coder.Constant</code> .
<code>X</code>	Must be a double- or single-precision numeric matrix
<code>link</code>	<ul style="list-style-type: none"> <li>• Does not support function handles, that is, functions created using <code>@</code>.</li> <li>• Must be a compile-time constant.</li> </ul>
<code>stats</code>	Must be a compile-time constant
'Constant ' name- value pair argument	Must be a compile-time constant
Name-value pair arguments names	Name-value pair argument names must be compile-time constants. For example, to specify a confidence level of 0.9, include <code>coder.Constant('Confidence')</code> , <code>coder.Constant(0.9)</code> in the <code>-args</code> value of <code>codegen</code> .

## See Also

`GeneralizedLinearModel` | `fitglm` | `glmfit` | `stepwiseglm`

Introduced before R2006a



# glyphplot

Glyph plot

## Syntax

```
glyphplot(X)
glyphplot(X, 'glyph', 'face')
glyphplot(X, 'glyph', 'face', 'features', f)
glyphplot(X, ..., 'grid', [rows, cols])
glyphplot(X, ..., 'grid', [rows, cols], 'page', p)
glyphplot(X, ..., 'centers', C)
glyphplot(X, ..., 'centers', C, 'radius', r)
glyphplot(X, ..., 'obslabels', labels)
glyphplot(X, ..., 'standardize', method)
glyphplot(X, ..., prop1, vall, ...)
h = glyphplot(X, ...)
```

## Description

`glyphplot(X)` creates a star plot from the multivariate data in the  $n$ -by- $p$  matrix  $X$ . Rows of  $X$  correspond to observations, columns to variables. A star plot represents each observation as a “star” whose  $i$ th spoke is proportional in length to the  $i$ th coordinate of that observation. `glyphplot` standardizes  $X$  by shifting and scaling each column separately onto the interval  $[0, 1]$  before making the plot, and centers the glyphs on a rectangular grid that is as close to square as possible. `glyphplot` treats NaNs in  $X$  as missing values, and does not plot the corresponding rows of  $X$ .

`glyphplot(X, 'glyph', 'star')` is a synonym for `glyphplot(X)`.

`glyphplot(X, 'glyph', 'face')` creates a face plot from  $X$ . A face plot represents each observation as a “face,” whose  $i$ th facial feature is drawn with a characteristic proportional to the  $i$ th coordinate of that observation. The features are described in “Face Features” on page 32-2917.

`glyphplot(X, 'glyph', 'face', 'features', f)` creates a face plot where the  $i$ th element of the index vector  $f$  defines which facial feature will represent the  $i$ th column of

X. *f* must contain integers from 0 to 17, where 0 indicate that the corresponding column of X should not be plotted. See “Face Features” on page 32-2917 for more information.

`glyphplot(X, ..., 'grid', [rows, cols])` organizes the glyphs into a rows-by-cols grid.

`glyphplot(X, ..., 'grid', [rows, cols], 'page', p)` organizes the glyph into one or more pages of a rows-by-cols grid, and displays the page *p*. If *p* is a vector, `glyphplot` displays multiple pages in succession. If *p* is 'all', `glyphplot` displays all pages. If *p* is 'scroll', `glyphplot` displays a single plot with a scrollbar.

`glyphplot(X, ..., 'centers', C)` creates a plot with each glyph centered at the locations in the *n*-by-2 matrix *C*.

`glyphplot(X, ..., 'centers', C, 'radius', r)` creates a plot with glyphs positioned using *C*, and scale the glyphs so the largest has radius *r*.

`glyphplot(X, ..., 'obslabels', labels)` labels each glyph with the text in *labels*. By default, the glyphs are labelled 1:N. Use ' ' for blank labels.

`glyphplot(X, ..., 'standardize', method)` standardizes X before making the plot. Choices for *method* are

- 'column' — Maps each column of X separately onto the interval [0,1]. This is the default.
- 'matrix' — Maps the entire matrix X onto the interval [0,1].
- 'PCA' — Transforms X to its principal component scores, in order of decreasing eigenvalue, and maps each one onto the interval [0,1].
- 'off' — No standardization. Negative values in X may make a star plot uninterpretable.

`glyphplot(X, ..., prop1, val1, ...)` sets properties to the specified property values for all line graphics objects created by `glyphplot`.

`h = glyphplot(X, ...)` returns a matrix of handles to the graphics objects created by `glyphplot`. For a star plot, `h(:, 1)` and `h(:, 2)` contain handles to the line objects for each star's perimeter and spokes, respectively. For a face plot, `h(:, 1)` and `h(:, 2)` contain object handles to the lines making up each face and to the pupils, respectively. `h(:, 3)` contains handles to the text objects for the labels, if present.

## Face Features

The following table describes the correspondence between the columns of the vector  $f$ , the value of the 'Features' input parameter, and the facial features of the glyph plot. If  $x$  has fewer than 17 columns, unused features are displayed at their default value.

Column	Facial Feature
1	Size of face
2	Forehead/jaw relative arc length
3	Shape of forehead
4	Shape of jaw
5	Width between eyes
6	Vertical position of eyes
7	Height of eyes
8	Width of eyes (this also affects eyebrow width)
9	Angle of eyes (this also affects eyebrow angle)
10	Vertical position of eyebrows
11	Width of eyebrows (relative to eyes)
12	Angle of eyebrows (relative to eyes)
13	Direction of pupils
14	Length of nose
15	Vertical position of mouth
16	Shape of mouth
17	Mouth arc length

## Examples

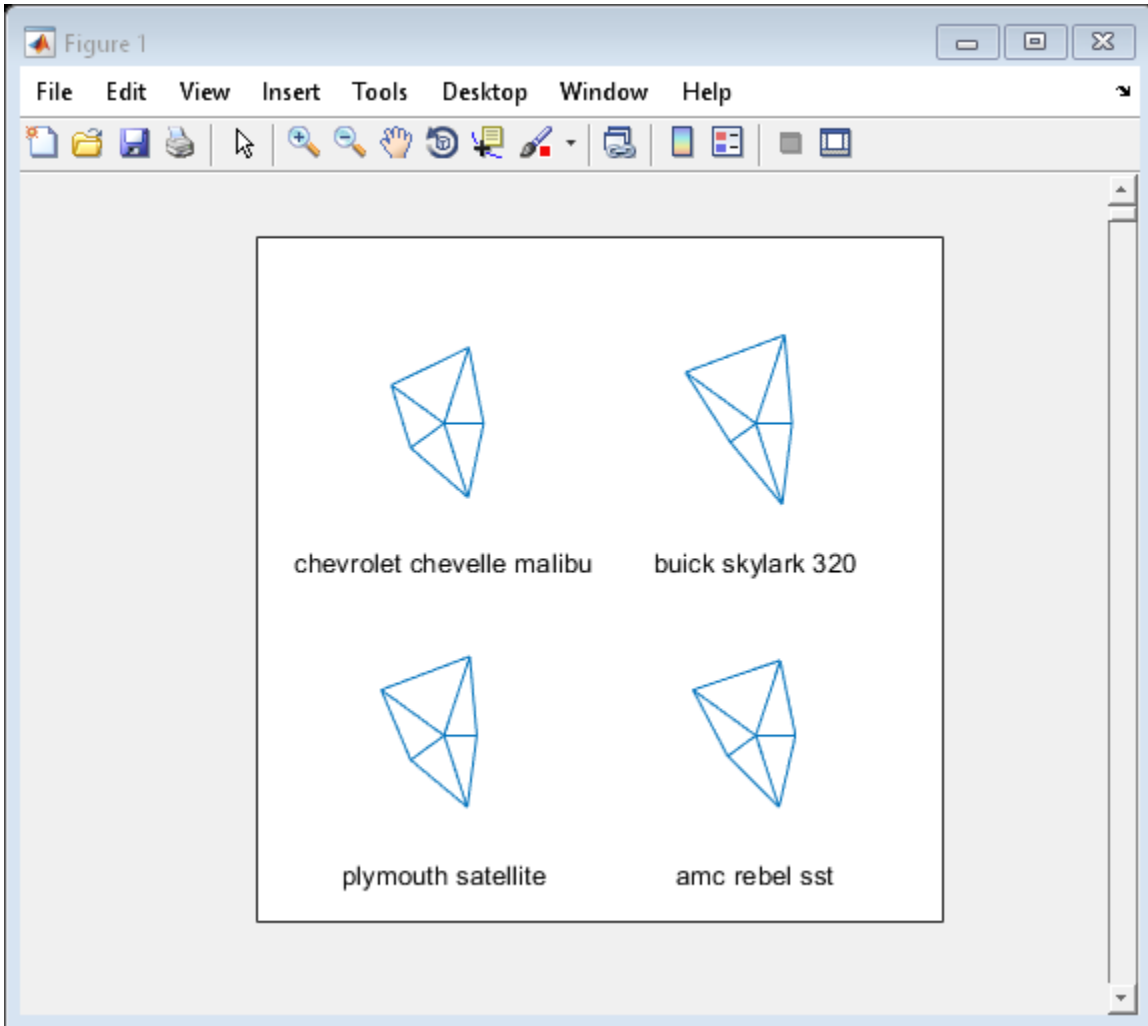
### Star and Face Plots of Multivariate Data

Load the sample data.

```
load carsmall
X = [Acceleration Displacement Horsepower MPG Weight];
```

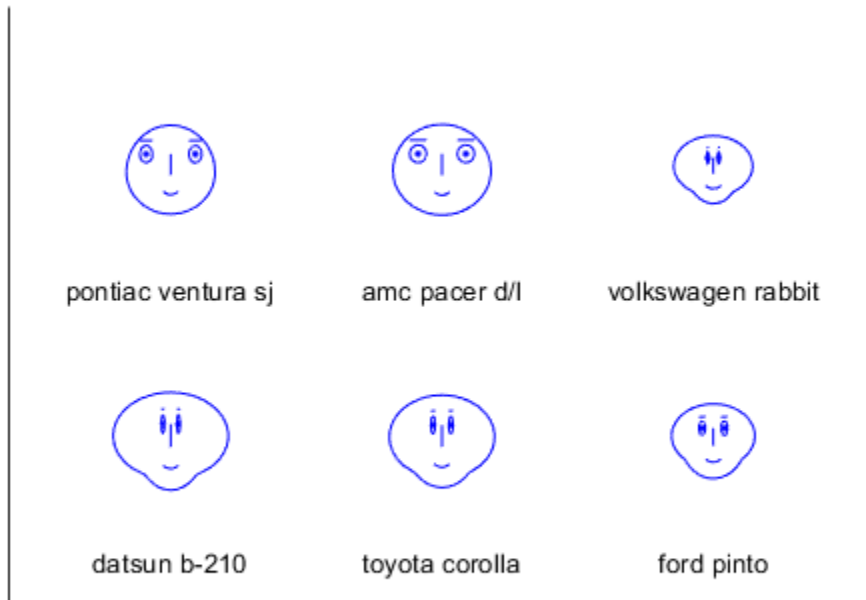
Create a star plot of the data in `X`. Standardize the data before plotting.

```
glyphplot(X, 'standardize', 'column', 'obslabels', Model, 'grid', [2 2], ...  
          'page', 'scroll');
```



Create a faceplot of the data in `X`.

```
glyphplot(X, 'glyph', 'face', 'obslabels', Model, 'grid', [2 3], 'page', 9);
```



## See Also

[andrewsplot](#) | [parallelcoords](#)

Introduced before R2006a

## gmdistribution class

Gaussian mixture models

### Description

An object of the `gmdistribution` class defines a Gaussian mixture distribution, which is a multivariate distribution that consists of a mixture of one or more multivariate Gaussian distribution components. The number of components for a given `gmdistribution` object is fixed. Each multivariate Gaussian component is defined by its mean and covariance, and the mixture is defined by a vector of mixing proportions.

### Construction

To create a Gaussian mixture distribution by specifying the distribution parameters, use the `gmdistribution` constructor. To fit a Gaussian mixture distribution model to data, use `fitgmdist`.

<code>fit</code>	Gaussian mixture parameter estimates
<code>gmdistribution</code>	Construct Gaussian mixture distribution

### Properties

All objects of the class have the properties listed in the following table.

CovarianceType	Type of covariance matrices
DistributionName	Type of distribution
mu	Input matrix of means $\mu$
NumComponents	Number $k$ of mixture components
NumVariables	Dimension $d$ of multivariate Gaussian distributions
ComponentProportion	Input vector of mixing proportions
ProbabilityTolerance	Tolerance for posterior probabilities
SharedCovariance	true if all covariance matrices are restricted to be the same
Sigma	Input array of covariances

Objects constructed with `fitgmdist` have the additional properties listed in the following table.

AIC	Akaike Information Criterion
BIC	Bayes Information Criterion
Converged	Determine if algorithm converged
NumIterations	Number of iterations
NegativeLogLikelihood	Negative of log-likelihood
RegularizationValue	Value of 'Regularize' parameter

## Methods

<code>cdf</code>	Cumulative distribution function for Gaussian mixture distribution
<code>cluster</code>	Construct clusters from Gaussian mixture distribution
<code>disp</code>	Display Gaussian mixture distribution object
<code>display</code>	Display Gaussian mixture distribution object
<code>fit</code>	Gaussian mixture parameter estimates
<code>mahal</code>	Mahalanobis distance to component means
<code>pdf</code>	Probability density function for Gaussian mixture distribution
<code>posterior</code>	Posterior probabilities of components
<code>random</code>	Random numbers from Gaussian mixture distribution
<code>subsasgn</code>	Subscripted reference for Gaussian mixture distribution object
<code>subsref</code>	Subscripted reference for Gaussian mixture distribution object

## Copy Semantics

Value. To learn how value classes affect copy operations, see [Copying Objects \(MATLAB\)](#).

## Examples

### Fit a Gaussian Mixture Model

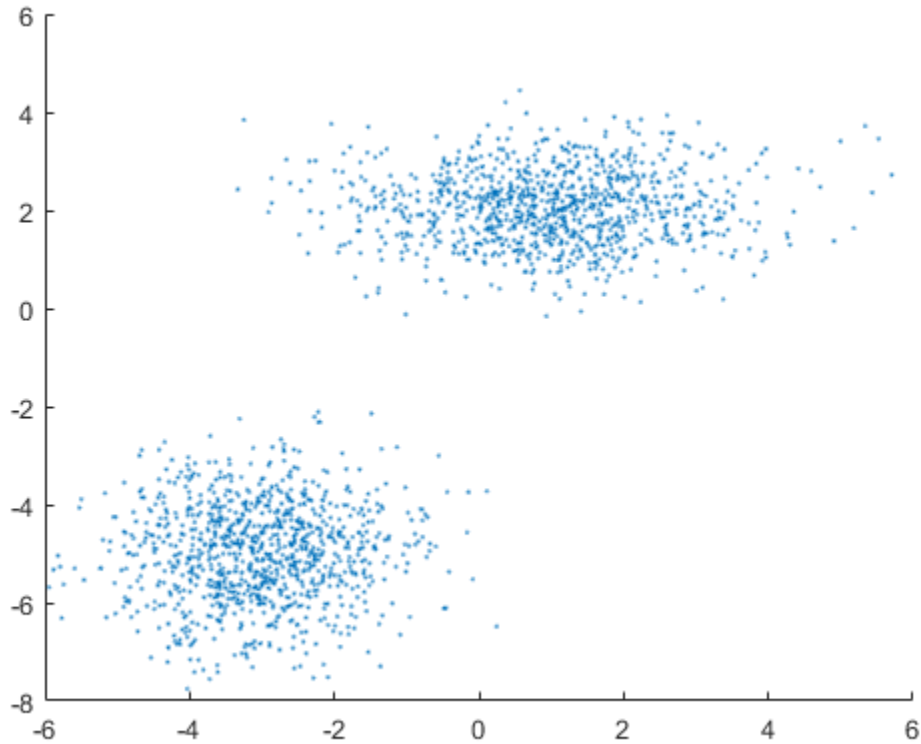
Generate data from a mixture of two bivariate Gaussian distributions using the `mvnrnd` function. Fit to the resulting data.

Generate the data using 1000 points from each distribution.

```
rng(1); % For reproducibility
MU1 = [1 2];
SIGMA1 = [2 0; 0 .5];
MU2 = [-3 -5];
SIGMA2 = [1 0; 0 1];
X = [mvnrnd(MU1, SIGMA1, 1000); mvnrnd(MU2, SIGMA2, 1000)];
```



```
scatter(X(:,1),X(:,2),10, '.')  
hold on
```



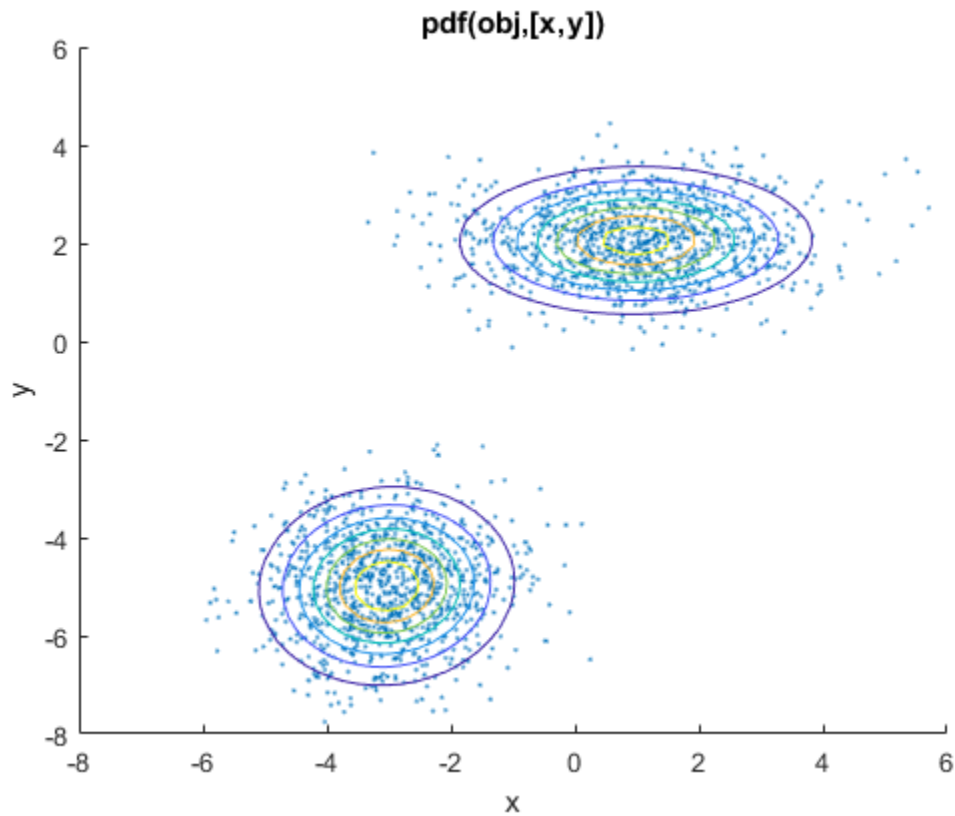
Fit a two-component Gaussian mixture model.

```
options = statset('Display','final');  
obj = fitgmdist(X,2,'Options',options);
```

```
5 iterations, log-likelihood = -7105.71
```

Plot the fit.

```
h = ezcontour(@(x,y)pdf(obj,[x y]),[-8 6],[-8 6]);
```



- “Normal Distribution” on page B-134

## References

- [1] McLachlan, G., and D. Peel. *Finite Mixture Models*. Hoboken, NJ: John Wiley & Sons, Inc., 2000.

## See Also

`fitgmdist`

## Topics

“Normal Distribution” on page B-134

## gmdistribution

**Class:** gmdistribution

Construct Gaussian mixture distribution

### Syntax

```
obj = gmdistribution(mu, sigma, p)
```

### Description

`obj = gmdistribution(mu, sigma, p)` constructs an object `obj` of the `gmdistribution` class defining a Gaussian mixture distribution.

`mu` is a  $k$ -by- $d$  matrix specifying the  $d$ -dimensional mean of each of the  $k$  components.

`sigma` specifies the covariance of each component. The size of `sigma` is:

- $d$ -by- $d$ -by- $k$  if there are no restrictions on the form of the covariance. In this case, `sigma(:, :, I)` is the covariance of component  $I$ .
- 1-by- $d$ -by- $k$  if the covariance matrices are restricted to be diagonal, but not restricted to be same across components. In this case, `sigma(:, :, I)` contains the diagonal elements of the covariance of component  $I$ .
- $d$ -by- $d$  matrix if the covariance matrices are restricted to be the same across components, but not restricted to be diagonal. In this case, `sigma` is the pooled estimate of covariance.
- 1-by- $d$  if the covariance matrices are restricted to be diagonal and the same across components. In this case, `sigma` contains the diagonal elements of the pooled estimate of covariance.

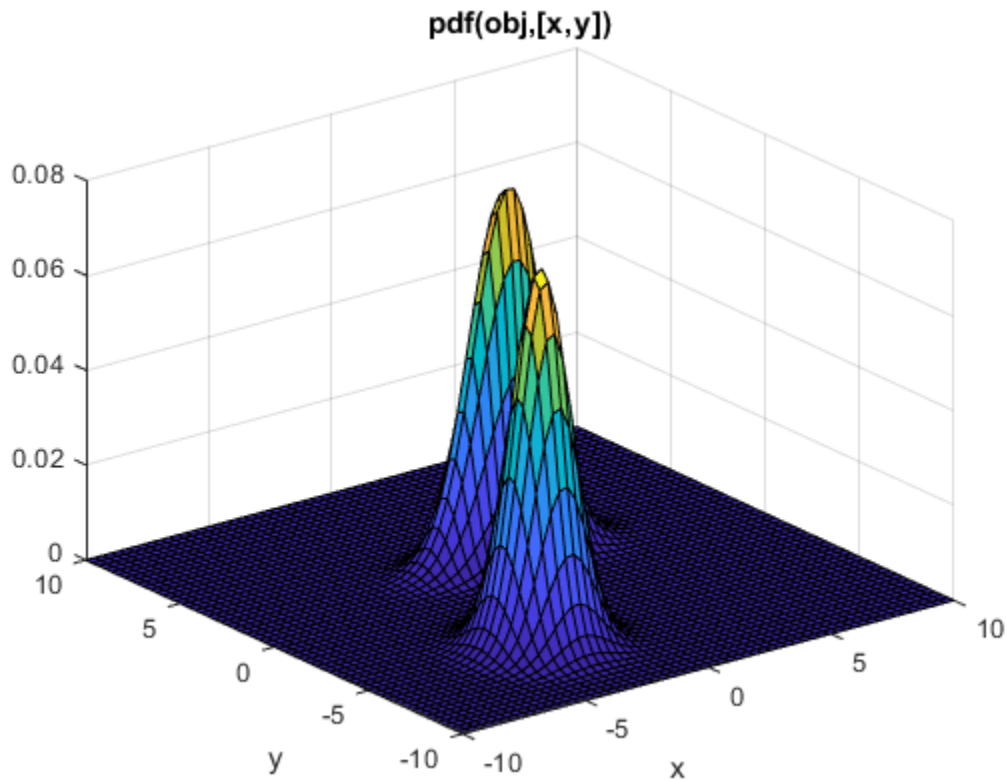
`p` is an optional 1-by- $k$  vector specifying the mixing proportions of each component. If `p` does not sum to 1, `gmdistribution` normalizes it. The default is equal proportions.

## Examples

### Construct a Gaussian Mixture Distribution

Create a `gmdistribution` distribution defining a two-component mixture of bivariate Gaussian distributions.

```
mu = [1 2;-3 -5];  
sigma = cat(3,[2 0;0 .5],[1 0;0 1]);  
p = ones(1,2)/2;  
obj = gmdistribution(mu,sigma,p);  
  
ezsurf(@(x,y)pdf(obj,[x y]),[-10 10],[-10 10])
```



## References

- [1] McLachlan, G., and D. Peel. *Finite Mixture Models*. Hoboken, NJ: John Wiley & Sons, Inc., 2000.

## See Also

`cdf` | `cluster` | `fitgmdist` | `mahal` | `pdf` | `posterior` | `random`

Introduced in R2007b

# gname

Add case names to plot

## Syntax

```
gname(cases)
gname
h = gname(cases, line_handle)
```

## Description

`gname(cases)` displays a figure window and waits for you to press a mouse button or a keyboard key. The input argument `cases` is a character array or a cell array of character vectors, in which each row of the character array or each element of the cell array contains the case name of a point. Moving the mouse over the graph displays a pair of cross-hairs. If you position the cross-hairs near a point with the mouse and click once, the graph displays the label corresponding to that point. Alternatively, you can click and drag the mouse to create a rectangle around several points. When you release the mouse button, the graph displays the labels for all points in the rectangle. Right-click a point to remove its label. When you are done labelling points, press the **Enter** or **Escape** key to stop labeling.

`gname` with no arguments labels each case with its case number.

`cases` typically contains unique case names for each point, and is a cell array of character vectors or a character matrix with each row representing a name. `cases` can also be any grouping variable, which `gname` converts to labels.

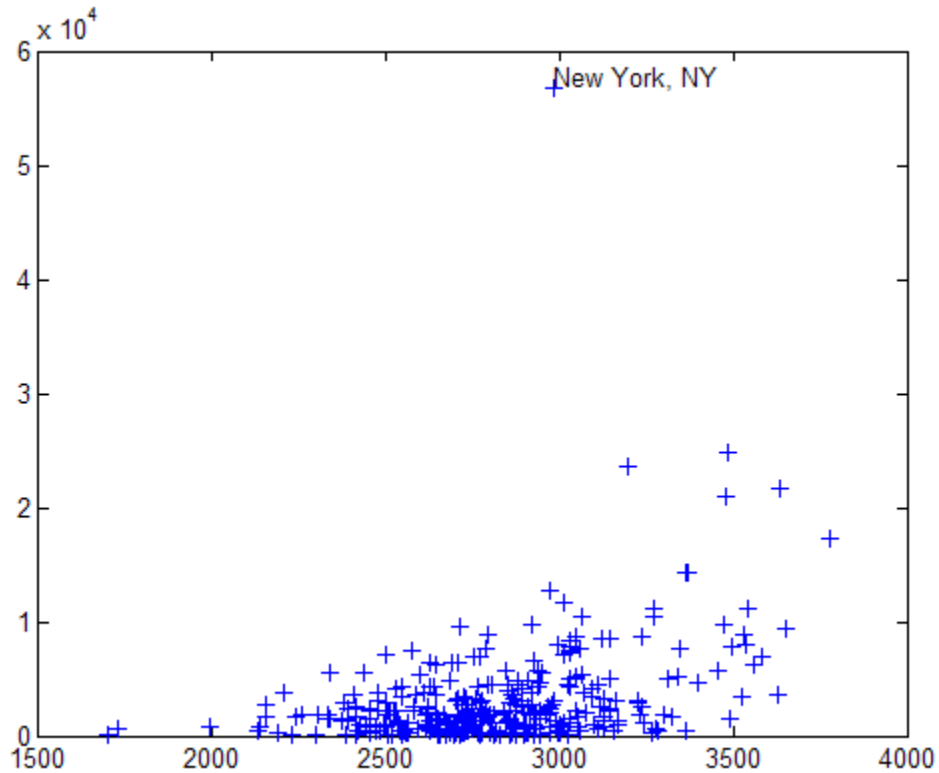
`h = gname(cases, line_handle)` returns a vector of handles to the text objects on the plot. Use the scalar `line_handle` to identify the correct line if there is more than one line object on the plot.

You can use `gname` to label plots created by the `plot`, `scatter`, `gscatter`, `plotmatrix`, and `gplotmatrix` functions.

## Examples

This example uses the city ratings data sets to find out which cities are the best and worst for education and the arts.

```
load cities
education = ratings(:,6);
arts = ratings(:,7);
plot(education,arts,'+')
gname(names)
```



Click the point at the top of the graph to display its label, “New York.”



## See Also

`gplotmatrix` | `gscatter` | `gtext`

**Introduced before R2006a**

## gpcdf

Generalized Pareto cumulative distribution function

### Syntax

```
p = gpcdf(x, k, sigma, theta)
p = gpcdf(x, k, sigma, theta, 'upper')
```

### Description

`p = gpcdf(x, k, sigma, theta)` returns the cdf of the generalized Pareto (GP) distribution with the tail index (shape) parameter `k`, scale parameter `sigma`, and threshold (location) parameter, `theta`, evaluated at the values in `x`. The size of `p` is the common size of the input arguments. A scalar input functions as a constant matrix of the same size as the other inputs.

`p = gpcdf(x, k, sigma, theta, 'upper')` returns the complement of the cdf of the generalized Pareto (GP) distribution, using an algorithm that more accurately computes the extreme upper tail probabilities.

Default values for `k`, `sigma`, and `theta` are 0, 1, and 0, respectively.

When `k = 0` and `theta = 0`, the GP is equivalent to the exponential distribution. When `k > 0` and `theta = sigma/k`, the GP is equivalent to a Pareto distribution with a scale parameter equal to `sigma/k` and a shape parameter equal to `1/k`. The mean of the GP is not finite when `k ≥ 1`, and the variance is not finite when `k ≥ 1/2`. When `k ≥ 0`, the GP has positive density for

`x > theta`, or, when

$$k < 0, 0 \leq \frac{x - \theta}{\sigma} \leq -\frac{1}{k}.$$

## References

- [1] Embrechts, P., C. Klüppelberg, and T. Mikosch. *Modelling Extremal Events for Insurance and Finance*. New York: Springer, 1997.
- [2] Kotz, S., and S. Nadarajah. *Extreme Value Distributions: Theory and Applications*. London: Imperial College Press, 2000.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`cdf` | `gpcfit` | `gpcinv` | `gpclike` | `gpcpdf` | `gpcrnd` | `gpcstat`

### Topics

- “Generalized Pareto Distribution” on page B-58
- “Working with Probability Distributions” on page 5-3
- “Nonparametric and Empirical Probability Distributions” on page 5-37
- “Fit a Nonparametric Distribution with Pareto Tails” on page 5-57
- “Supported Distributions” on page 5-17

Introduced before R2006a

## gpfif

Generalized Pareto parameter estimates

### Syntax

```
parmhat = gpfif(x)
[parmhat,parmci] = gpfif(x)
[parmhat,parmci] = gpfif(x,alpha)
[...] = gpfif(x,alpha,options)
```

### Description

`parmhat = gpfif(x)` returns maximum likelihood estimates of the parameters for the two-parameter generalized Pareto (GP) distribution given the data in `x`. `parmhat(1)` is the tail index (shape) parameter, `k` and `parmhat(2)` is the scale parameter, `sigma`. `gpfif` does not fit a threshold (location) parameter.

`[parmhat,parmci] = gpfif(x)` returns 95% confidence intervals for the parameter estimates.

`[parmhat,parmci] = gpfif(x,alpha)` returns 100(1-alpha)% confidence intervals for the parameter estimates.

`[...] = gpfif(x,alpha,options)` specifies control parameters for the iterative algorithm used to compute ML estimates. This argument can be created by a call to `statset`. See `statset('gpfif')` for parameter names and default values.

Other functions for the generalized Pareto, such as `gpcdf` allow a threshold parameter, `theta`. However, `gpfif` does not estimate `theta`. It is assumed to be known, and subtracted from `x` before calling `gpfif`.

When `k = 0` and `theta = 0`, the GP is equivalent to the exponential distribution. When `k > 0` and `theta = sigma/k`, the GP is equivalent to a Pareto distribution with a scale parameter equal to `sigma/k` and a shape parameter equal to `1/k`. The mean of the GP is

not finite when  $k \geq 1$ , and the variance is not finite when  $k \geq 1/2$ . When  $k \geq 0$ , the GP has positive density for

$k > \theta$ , or, when  $k < 0$ , for

$$0 \leq \frac{x - \theta}{\sigma} \leq -\frac{1}{k}$$

## References

- [1] Embrechts, P., C. Klüppelberg, and T. Mikosch. *Modelling Extremal Events for Insurance and Finance*. New York: Springer, 1997.
- [2] Kotz, S., and S. Nadarajah. *Extreme Value Distributions: Theory and Applications*. London: Imperial College Press, 2000.

## See Also

gpcdf | gpinv | gplike | gppdf | gprnd | gpstat | mle

## Topics

- “Generalized Pareto Distribution” on page B-58
- “Working with Probability Distributions” on page 5-3
- “Nonparametric and Empirical Probability Distributions” on page 5-37
- “Fit a Nonparametric Distribution with Pareto Tails” on page 5-57
- “Supported Distributions” on page 5-17

Introduced before R2006a

## gpinv

Generalized Pareto inverse cumulative distribution function

### Syntax

```
x = gpinv(p, k, sigma, theta)
```

### Description

`x = gpinv(p, k, sigma, theta)` returns the inverse cdf for a generalized Pareto (GP) distribution with tail index (shape) parameter `k`, scale parameter `sigma`, and threshold (location) parameter `theta`, evaluated at the values in `p`. The size of `x` is the common size of the input arguments. A scalar input functions as a constant matrix of the same size as the other inputs.

Default values for `k`, `sigma`, and `theta` are 0, 1, and 0, respectively.

When `k = 0` and `theta = 0`, the GP is equivalent to the exponential distribution. When `k > 0` and `theta = sigma/k`, the GP is equivalent to a Pareto distribution with a scale parameter equal to `sigma/k` and a shape parameter equal to `1/k`. The mean of the GP is not finite when `k ≥ 1`, and the variance is not finite when `k ≥ 1/2`. When `k ≥ 0`, the GP has positive density for

`x > theta`, or, when

$$k < 0, 0 \leq \frac{x - \theta}{\sigma} \leq -\frac{1}{k}.$$

### References

- [1] Embrechts, P., C. Klüppelberg, and T. Mikosch. *Modelling Extremal Events for Insurance and Finance*. New York: Springer, 1997.

[2] Kotz, S., and S. Nadarajah. *Extreme Value Distributions: Theory and Applications*. London: Imperial College Press, 2000.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`gpcdf` | `gpfit` | `gplike` | `gppdf` | `gprnd` | `gpstat` | `icdf`

### Topics

“Generalized Pareto Distribution” on page B-58

“Working with Probability Distributions” on page 5-3

“Nonparametric and Empirical Probability Distributions” on page 5-37

“Fit a Nonparametric Distribution with Pareto Tails” on page 5-57

“Supported Distributions” on page 5-17

**Introduced before R2006a**

## gplike

Generalized Pareto negative log-likelihood

### Syntax

```
nlogL = gplike(params, data)
[nlogL, acov] = gplike(params, data)
```

### Description

`nlogL = gplike(params, data)` returns the negative of the log-likelihood `nlogL` for the two-parameter generalized Pareto (GP) distribution, evaluated at parameters `params`. `params(1)` is the tail index (shape) parameter, `k`, and `params(2)` is the scale parameter. `gplike` does not allow a threshold (location) parameter.

`[nlogL, acov] = gplike(params, data)` returns the inverse of Fisher's information matrix, `acov`. If the input parameter values in `params` are the maximum likelihood estimates, the diagonal elements of `acov` are their asymptotic variances. `acov` is based on the observed Fisher's information, not the expected information.

When `k = 0` and `theta = 0`, the GP is equivalent to the exponential distribution. When `k > 0` and `theta = sigma/k`, the GP is equivalent to a Pareto distribution with a scale parameter equal to `sigma/k` and a shape parameter equal to `1/k`. The mean of the GP is not finite when `k ≥ 1`, and the variance is not finite when `k ≥ 1/2`. When `k ≥ 0`, the GP has positive density for

`x > theta`, or, when

$$k < 0, 0 \leq \frac{x - \theta}{\sigma} \leq -\frac{1}{k}.$$



## References

- [1] Embrechts, P., C. Klüppelberg, and T. Mikosch. *Modelling Extremal Events for Insurance and Finance*. New York: Springer, 1997.
- [2] Kotz, S., and S. Nadarajah. *Extreme Value Distributions: Theory and Applications*. London: Imperial College Press, 2000.

## See Also

gpcdf | gpfit | gpinv | gppdf | gprnd | gpstat

## Topics

- “Generalized Pareto Distribution” on page B-58
- “Working with Probability Distributions” on page 5-3
- “Nonparametric and Empirical Probability Distributions” on page 5-37
- “Fit a Nonparametric Distribution with Pareto Tails” on page 5-57
- “Supported Distributions” on page 5-17

**Introduced before R2006a**

## gppdf

Generalized Pareto probability density function

### Syntax

```
p = gppdf(x, k, sigma, theta)
```

### Description

`p = gppdf(x, k, sigma, theta)` returns the pdf of the generalized Pareto (GP) distribution with the tail index (shape) parameter `k`, scale parameter `sigma`, and threshold (location) parameter, `theta`, evaluated at the values in `x`. The size of `p` is the common size of the input arguments. A scalar input functions as a constant matrix of the same size as the other inputs.

Default values for `k`, `sigma`, and `theta` are 0, 1, and 0, respectively.

When `k = 0` and `theta = 0`, the GP is equivalent to the exponential distribution. When `k > 0` and `theta = sigma/k`, the GP is equivalent to a Pareto distribution with a scale parameter equal to `sigma/k` and a shape parameter equal to `1/k`. The mean of the GP is not finite when `k ≥ 1`, and the variance is not finite when `k ≥ 1/2`. When `k ≥ 0`, the GP has positive density for

`x > theta`, or, when

$$k < 0, 0 \leq \frac{x - \theta}{\sigma} \leq -\frac{1}{k}.$$

### References

- [1] Embrechts, P., C. Klüppelberg, and T. Mikosch. *Modelling Extremal Events for Insurance and Finance*. New York: Springer, 1997.

[2] Kotz, S., and S. Nadarajah. *Extreme Value Distributions: Theory and Applications*. London: Imperial College Press, 2000.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

[gpcdf](#) | [gpfit](#) | [gpinv](#) | [gplike](#) | [gprnd](#) | [gpstat](#) | [pdf](#)

### Topics

“Generalized Pareto Distribution” on page B-58

“Working with Probability Distributions” on page 5-3

“Nonparametric and Empirical Probability Distributions” on page 5-37

“Fit a Nonparametric Distribution with Pareto Tails” on page 5-57

“Supported Distributions” on page 5-17

**Introduced before R2006a**

## gplotmatrix

Matrix of scatter plots by group

### Syntax

```
gplotmatrix(x, [], group)
gplotmatrix(x, y, group)
gplotmatrix(x, y, group, clr, sym, siz)
gplotmatrix(x, y, group, clr, sym, siz, doleg)
gplotmatrix(x, [], group, clr, sym, siz, doleg, dispopt)
gplotmatrix(x, [], group, clr, sym, siz, doleg, 'variable', xnam)
gplotmatrix(x, y, group, clr, sym, siz, doleg, [], xnam, ynam)
[h, ax, bigax] = gplotmatrix( ___ )
```

### Description

`gplotmatrix(x, [], group)` creates a matrix of scatter plots of the data in `x`, grouped by the grouping variable in `group`. Each individual set of axes in the resulting figure contains a scatter plot of a column of `x` against another column of `x`. It also plots the outlines of the grouped histograms in the diagonals.

`gplotmatrix(x, y, group)` creates a matrix of scatter plots. Each individual set of axes in the resulting figure contains a scatter plot of a column of `x` against a column of `y`. All plots are grouped by the grouping variable `group`.

`x` and `y` are matrices with the same number of rows. If `x` has  $p$  columns and `y` has  $q$  columns, the figure contains a  $p$ -by- $q$  matrix of scatter plots. If you omit `y` or specify it as the empty matrix, `[]`, `gplotmatrix` creates a square matrix of scatter plots of columns of `x` against each other.

`group` is a grouping variable that can be a categorical variable, vector, character array, or cell array of character vectors. `group` must have the same number of rows as `x` and `y`. Points with the same value of `group` are placed in the same group, and appear on the graph with the same marker and color. Alternatively, `group` can be a cell array

containing several grouping variables (such as {g1 g2 g3}); in that case, observations are in the same group if they have common values of all grouping variables.

`gplotmatrix(x,y,group,clr,sym,siz)` specifies the color, marker type, and size for each group. `clr` is a character array of colors recognized by the `plot` function. `sym` is a character array of symbols recognized by the `plot` command, with the default value `'.'`. `siz` is a vector of sizes, with the default value determined based on the number of observations. If you do not specify enough values for all groups, `gplotmatrix` cycles through the specified values as needed.

`gplotmatrix(x,y,group,clr,sym,siz,doleg)` controls whether a legend is displayed on the graph (`doleg` is `'on'`, the default) or not (`doleg` is `'off'`).

`gplotmatrix(x,[],group,clr,sym,siz,doleg,dispopt)` controls what appears along the diagonal of a plot matrix of `x`. Allowable values are `'none'`, to leave the diagonals blank, `'hist'`, to plot histograms, `'stairs'` to display the outlines of grouped histograms (default if there is more than one group), `'grpbars'` to plot grouped histogram bars.

`gplotmatrix(x,[],group,clr,sym,siz,doleg,'variable',xnam)` displays the variable names along the diagonal of the plot matrix of `x`. It also uses these column names to label the `x`- and `y`-axes.

`gplotmatrix(x,y,group,clr,sym,siz,doleg,[],xnam,ynam)` labels the `x`- and `y`-axes using the column names specified in `xnam` and `ynam`.

`xnam` and `ynam` must be character arrays or cell arrays of character vectors, with one name for each column of `x` and `y`, respectively.

`[h,ax,bigax] = gplotmatrix(____)` returns three arrays of handles for any of the previous input arguments. `h` is an array of handles to the lines on the graphs. The array's third dimension corresponds to groups in the input argument `group`. `ax` is a matrix of handles to the axes of the individual plots. If `dispopt` is `'hist'`, `'stairs'`, or `'grpbars'`, `ax` contains one extra row of handles to invisible axes in which the histograms are plotted. `bigax` is a handle to big (invisible) axes framing the entire plot matrix. `bigax` is fixed to point to the current axes, so a subsequent `title`, `xlabel`, or `ylabel` command will produce labels that are centered with respect to the entire plot matrix.

## Examples

### Create Grouped Data Scatter Plot Matrix

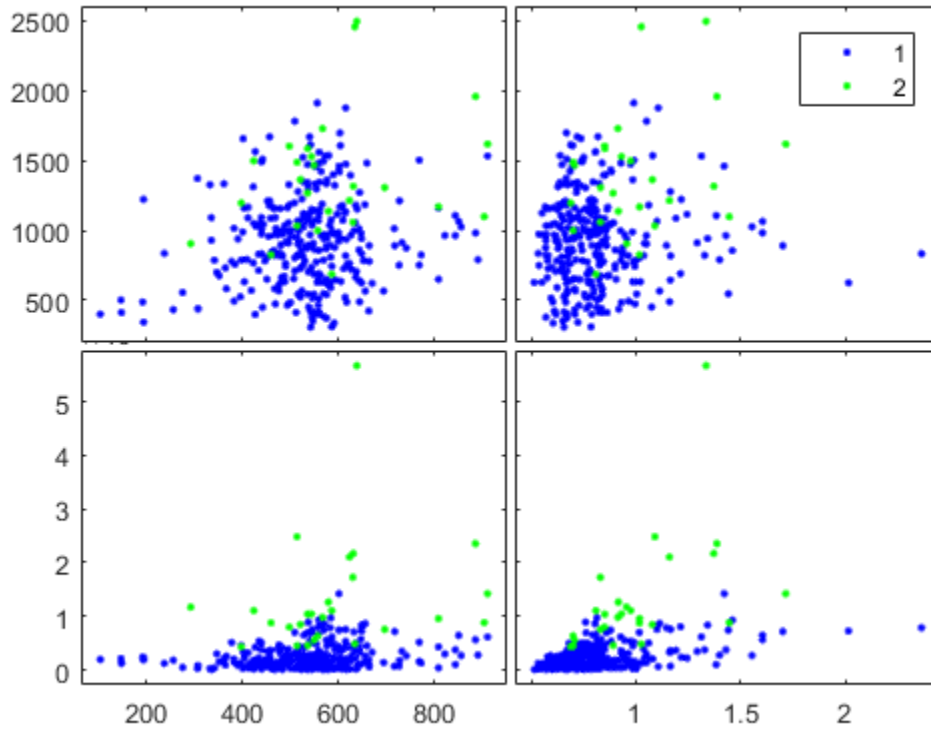
Load the sample data.

```
load discrim;
```

The `ratings` array contains rating values for 329 U.S. cities in the nine different categories listed in the `categories` array. The `group` array contains a city size code that is equal to 2 for the 26 largest cities, and 1 otherwise.

Create a matrix of scatter plots to compare the first two categories, `climate` and `housing`, with categories 4 (`crime`) and 7 (`arts`). Specify `group` as the grouping variable to visually distinguish the data for large and small cities.

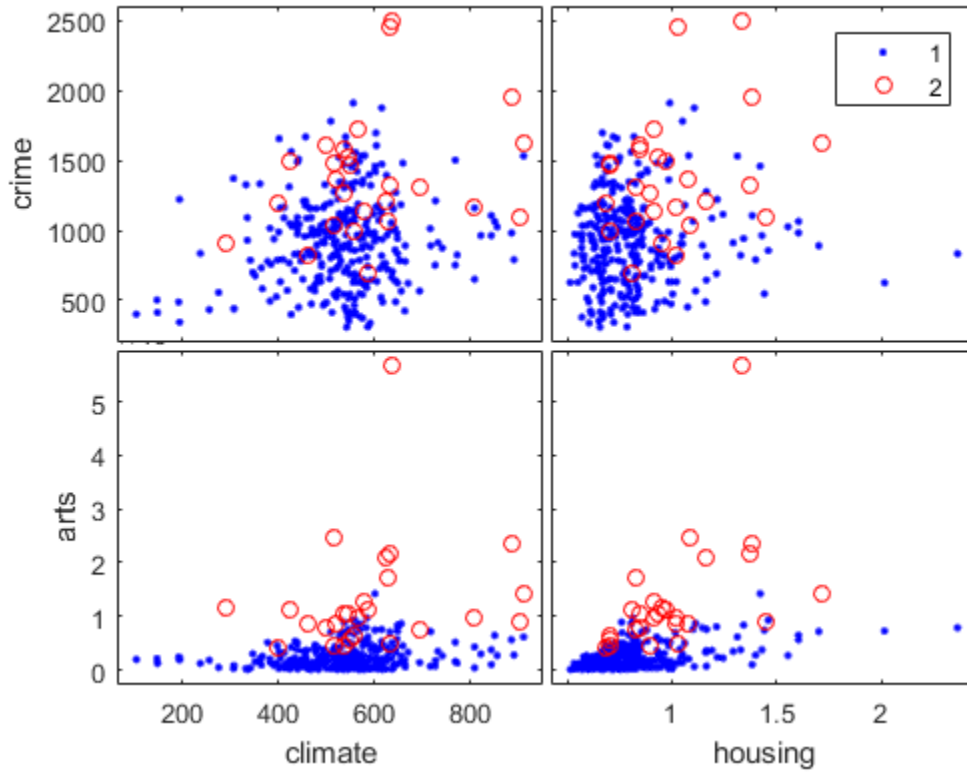
```
figure;  
gplotmatrix(ratings(:,1:2), ratings(:, [4 7]), group);
```



The figure displays a matrix of scatter plots for the specified comparisons, with each city size group represented by a different color.

For better clarity, you can adjust the appearance of the graphs by specifying colors and plotting symbols, and labeling the axes with the rating categories.

```
figure;
gplotmatrix(ratings(:,1:2),ratings(:,[4 7]),group,...
    'br','.o',[],'on','',categories(1:2,:),...
    categories([4 7],:));
```



## See Also

`grpstats` | `gscatter` | `plotmatrix`

## Topics

“Grouping Variables” on page 2-59

Introduced before R2006a



## gprnd

Generalized Pareto random numbers

### Syntax

```
r = gprnd(k, sigma, theta)
r = gprnd(k, sigma, theta, m, n, ...)
R = gprnd(K, sigma, theta, [m, n, ...])
```

### Description

`r = gprnd(k, sigma, theta)` returns an array of random numbers chosen from the generalized Pareto (GP) distribution with tail index (shape) parameter `k`, scale parameter `sigma`, and threshold (location) parameter, `theta`. The size of `r` is the common size of the input arguments if all are arrays. If any parameter is a scalar, the size of `r` is the size of the other parameters.

`r = gprnd(k, sigma, theta, m, n, ...)` or `R = gprnd(K, sigma, theta, [m, n, ...])` generates an `m`-by-`n`-by-... array. The `k`, `sigma`, `theta` parameters can each be scalars or arrays of the same size as `r`.

When `k = 0` and `theta = 0`, the GP is equivalent to the exponential distribution. When `k > 0` and `theta = sigma/k`, the GP is equivalent to a Pareto distribution with a scale parameter equal to `sigma/k` and a shape parameter equal to `1/k`. The mean of the GP is not finite when `k ≥ 1`, and the variance is not finite when `k ≥ 1/2`. When `k ≥ 0`, the GP has positive density for

`x > theta`, or, when

$$0 \leq \frac{x - \theta}{\sigma} \leq -\frac{1}{k}$$

## References

- [1] Embrechts, P., C. Klüppelberg, and T. Mikosch. *Modelling Extremal Events for Insurance and Finance*. New York: Springer, 1997.
- [2] Kotz, S., and S. Nadarajah. *Extreme Value Distributions: Theory and Applications*. London: Imperial College Press, 2000.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

The generated code can return a different sequence of numbers than MATLAB if either of the following is true:

- The output is nonscalar.
- An input parameter is invalid for the distribution.

### See Also

`gpcdf` | `gpfit` | `gpinv` | `gplike` | `gppdf` | `gpstat` | `random`

### Topics

“Generalized Pareto Distribution” on page B-58

“Working with Probability Distributions” on page 5-3

“Nonparametric and Empirical Probability Distributions” on page 5-37

“Fit a Nonparametric Distribution with Pareto Tails” on page 5-57

“Supported Distributions” on page 5-17

**Introduced before R2006a**

# gpstat

Generalized Pareto mean and variance

## Syntax

```
[m,v] = gpstat(k,sigma,theta)
```

## Description

`[m,v] = gpstat(k,sigma,theta)` returns the mean of and variance for the generalized Pareto (GP) distribution with the tail index (shape) parameter `k`, scale parameter `sigma`, and threshold (location) parameter, `theta`.

The default value for `theta` is 0.

When `k = 0` and `theta = 0`, the GP is equivalent to the exponential distribution. When `k > 0` and `theta = sigma/k`, the GP is equivalent to a Pareto distribution with a scale parameter equal to `sigma/k` and a shape parameter equal to `1/k`. The mean of the GP is not finite when `k ≥ 1`, and the variance is not finite when `k ≥ 1/2`. When `k ≥ 0`, the GP has positive density for `x > theta`, or when

$$k < 0, 0 \leq \frac{x-\theta}{\sigma} \leq -\frac{1}{k}.$$

## References

- [1] Embrechts, P., C. Klüppelberg, and T. Mikosch. *Modelling Extremal Events for Insurance and Finance*. New York: Springer, 1997.
- [2] Kotz, S., and S. Nadarajah. *Extreme Value Distributions: Theory and Applications*. London: Imperial College Press, 2000.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`gpcdf` | `gpfit` | `gpinv` | `gplike` | `gppdf` | `gprnd`

### Topics

“Generalized Pareto Distribution” on page B-58

“Working with Probability Distributions” on page 5-3

“Nonparametric and Empirical Probability Distributions” on page 5-37

“Fit a Nonparametric Distribution with Pareto Tails” on page 5-57

“Supported Distributions” on page 5-17

**Introduced before R2006a**

# growTrees

**Class:** TreeBagger

Train additional trees and add to ensemble

## Syntax

```
B = growTrees(B,ntrees)
```

```
B = growTrees(B,ntrees, 'param1', val1, 'param2', val2, ...)
```

## Description

`B = growTrees(B,ntrees)` grows `ntrees` new trees and appends them to those trees already stored in the ensemble `B`.

`B = growTrees(B,ntrees, 'param1', val1, 'param2', val2, ...)` specifies optional parameter name/value pairs:

<code>'NumPrint'</code>	Specifies that a diagnostic message showing training progress should display after every <code>value</code> training cycles (grown trees). Default is no diagnostic messages.
-------------------------	---

**'Options'**

A struct that specifies options that govern computation when growing the ensemble of decision trees. One option requests that the computation of decision trees on multiple bootstrap replicates uses multiple processors, if the Parallel Computing Toolbox is available. Two options specify the random number streams to use in selecting bootstrap replicates. You can create this argument with a call to `statset`. You can retrieve values of the individual fields with a call to `statget`. Applicable `statset` parameters are:

- `'UseParallel'` — If `true` and if a parpool of the Parallel Computing Toolbox is open, compute decision trees drawn on separate bootstrap replicates in parallel. If the Parallel Computing Toolbox is not installed, or a parpool is not open, computation occurs in serial mode. Default is `false`, or serial computation.
- `UseSubstreams` — Set to `true` to compute in parallel in a reproducible fashion. Default is `false`. To compute reproducibly, set `Streams` to a type allowing substreams: `'mlfg6331_64'` or `'mrg32k3a'`.
- `Streams` — A `RandStream` object or cell array of such objects. If you do not specify `Streams`, `growTrees` uses the default stream or streams. If you choose to specify `Streams`, use a single object except in the case
  - You have an open Parallel pool
  - `UseParallel` is `true`
  - `UseSubstreams` is `false`

In that case, use a cell array the same size as the Parallel pool.

## See Also

`TreeBagger` | `TreeBagger` | `fitctree` | `fitrtree` | `statget` | `statset`

# grp2idx

Create index vector from grouping variable

## Syntax

```
[G,GN]=grp2idx(S)  
[G,GN,GL] = grp2idx(S)
```

## Description

`[G,GN]=grp2idx(S)` creates an index vector `G` from the grouping variable `S`. `S` can be a categorical, numeric, logical, datetime, or duration vector; a cell array of character vectors; or a character array with each row representing a group label. The result `G` is a vector taking integer values from 1 up to the number `K` of distinct groups. `GN` is a cell array of character vectors representing group labels. `GN(G)` reproduces `S` (aside from any differences in type).

The order of `GN` depends on the grouping variable:

- For numeric and logical grouping variables, the order is the sorted order of `S`.
- For categorical grouping variables, the order is the order of `getlabels(S)`.
- For grouping variables, the order is the order of first appearance in `S`.

`[G,GN,GL] = grp2idx(S)` returns a column vector `GL` representing the group levels. The set of groups and their order in `GL` and `GN` are the same, except that `GL` has the same type as `S`. If `S` is a character matrix, `GL(G, :)` reproduces `S`, otherwise `GL(G)` reproduces `S`.

`grp2idx` treats NaNs (numeric, duration, or logical), ' ' (empty character arrays or cell arrays of character vectors), `<undefined>` values (categorical), and NaTs (datetime) in `S` as missing values and returns NaNs in the corresponding rows of `G`. `GN` and `GL` don't include entries for missing values.

## Examples

Load the data in `hospital.mat` and create a categorical grouping variable:

```
load hospital
edges = 0:10:100;
labels = strcat(num2str((0:10:90)', '%d'), {'s'});
AgeGroup = ordinal(hospital.Age, labels, [], edges);
```

```
ages = hospital.Age(1:5)
```

```
ages =
    38
    43
    38
    40
    49
```

```
group = AgeGroup(1:5)
```

```
group =
    30s
    40s
    30s
    40s
    40s
```

```
indices = grp2idx(group)
```

```
indices =
     4
     5
     4
     5
     5
```

## See Also

[crosstab](#) | [getlabels](#) | [grpstats](#) | [gscatter](#)

## Topics

“Grouping Variables” on page 2-59



**Introduced before R2006a**

## grpstats

Summary statistics organized by group

### Syntax

```
statarray = grpstats(tbl,groupvar)
statarray = grpstats(tbl,groupvar,whichstats)
statarray = grpstats(tbl,groupvar,whichstats,Name,Value)

means = grpstats(X,group)
[stats1,...,statsN] = grpstats(X,group,whichstats)
[stats1,...,statsN] = grpstats(X,group,whichstats,'Alpha',alpha)

grpstats(X,group,alpha)
```

### Description

`statarray = grpstats(tbl,groupvar)` returns a table or dataset array with the means for the data groups specified in `tbl` determined by the values of the grouping variable or variables specified in `groupvar`.

- If there is a single grouping variable, then there is a row in `statarray` for each value of the grouping variable. `grpstats` sorts the groups by order of appearance (if the grouping variable is a character array), in ascending numeric order (if the grouping variable is numeric), or in order of the levels (if the grouping variable is categorical).
- If `groupvar` is a cell array of character vectors containing multiple grouping variable names, or a vector of column numbers, then there is a row in `statarray` for each observed unique combination of values of the grouping variables. `grpstats` sorts the groups by the values of the first grouping variable, then the second grouping variable, and so on.
- If any variables in `tbl` (other than those specified in `groupvar`) are not numeric or logical arrays, then you must specify the names or column numbers of the numeric and logical variables for which you want to calculate means using the name-value pair argument, `DataVars`.

`statarray = grpstats(tbl,groupvar,whichstats)` returns the group values for the summary statistics types specified in `whichstats`.

`statarray = grpstats(tbl,groupvar,whichstats,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

`means = grpstats(X,group)` returns a column vector or matrix with the means of the groups of the data in the matrix or vector `X` determined by the values of the grouping variable or variables, `group`. The rows of `means` correspond to the grouping variable values.

- If there is a single grouping variable, then there is a row in `means` for each value of the grouping variable. `grpstats` sorts the groups by order of appearance (if the grouping variable is a character array), in ascending numeric order (if the grouping variable is numeric), or in order of the levels (if the grouping variable is categorical).
- If `group` is a cell array of grouping variables, then there is a row in `means` for each observed unique combination of values of the grouping variables. `grpstats` sorts the groups by the values of the first grouping variable, then the second grouping variable, and so on.
- If `X` is a matrix, then `means` is a matrix with the same number of columns as `X`. Each column of `means` has the group means for the corresponding column of `X`.

`[stats1,...,statsN] = grpstats(X,group,whichstats)` returns column vectors or arrays with group values for the summary statistic types specified in `whichstats`.

`[stats1,...,statsN] = grpstats(X,group,whichstats,'Alpha',alpha)` specifies the significance level for confidence and prediction intervals.

`grpstats(X,group,alpha)` plots the means of the groups of data in the vector or matrix `X` determined by the values of the grouping variable, `group`. The grouping variable values are on the horizontal plot axis. Each group mean has  $100 \times (1 - \alpha)\%$  confidence intervals.

- If `X` is a matrix, then `grpstats` plots the means and confidence intervals for each column of `X`.
- If `group` is a cell array of grouping variables, then `grpstats` plots the means and confidence intervals for the groups of data in `X` determined by the unique combinations of values of the grouping variables. For example, if there are two grouping variables, each with two values, there are four possible combinations of

grouping variable values. The plot includes only the combinations of values that exist in the input grouping variables (not all possible combinations).

## Examples

### Dataset Array Summary Statistics Organized by Group

Load the sample data.

```
load('hospital')
```

The dataset array `hospital` has 100 observations and 7 variables.

Create a dataset array with only the variables `Sex`, `Age`, `Weight`, and `Smoker`.

```
dsa = hospital(:, {'Sex', 'Age', 'Weight', 'Smoker'});
```

`Sex` is a nominal array, with levels `Male` and `Female`. The variables `Age` and `Weight` have numeric values, and `Smoker` has logical values.

Compute the mean for the numeric and logical arrays, `Age`, `Weight`, and `Smoker`, grouped by the levels in `Sex`.

```
statarray = grpstats(dsa, 'Sex')
```

```
statarray =
```

	Sex	GroupCount	mean_Age	mean_Weight	mean_Smoker
Female	Female	53	37.717	130.47	0.24528
Male	Male	47	38.915	180.53	0.44681

`statarray` is a dataset array with two rows, corresponding to the levels in `Sex`. `GroupCount` is the number of observations in each group. The means of `Age`, `Weight`, and `Smoker`, grouped by `Sex`, are given in `mean_Age`, `mean_Weight`, and `mean_Smoker`.

Compute the mean for `Age` and `Weight`, grouped by the values in `Smoker`.

```
statarray = grpstats(dsa, 'Smoker', 'mean', 'DataVars', {'Age', 'Weight'})
```

```

statarray =

      Smoker  GroupCount  mean_Age  mean_Weight
0  false     66          37.97    149.91
1  true      34          38.882   161.94

```

In this case, not all variables in `dsa` (excluding the grouping variable, `Smoker`) are numeric or logical arrays; the variable `Sex` is a nominal array. When not all variables in the input dataset array are numeric or logical arrays, you must specify the variables for which you want to calculate summary statistics using `DataVars`.

Compute the minimum and maximum weight, grouped by the combinations of values in `Sex` and `Smoker`.

```

statarray = grpstats(dsa, {'Sex', 'Smoker'}, {'min', 'max'}, ...
                    'DataVars', 'Weight')

```

```

statarray =

      Sex      Smoker  GroupCount  min_Weight  max_Weight
Female_0  Female    false      40          111         147
Female_1  Female     true       13          115         146
Male_0    Male       false      26          158         194
Male_1    Male       true       21          164         202

```

There are two unique values in `Smoker` and two levels in `Sex`, for a total of four possible combinations of values: Female Nonsmoker (`Female_0`), Female Smoker (`Female_1`), Male Nonsmoker (`Male_0`), and Male Smoker (`Male_1`).

Specify the names for the columns in the output.

```

statarray = grpstats(dsa, {'Sex', 'Smoker'}, {'min', 'max'}, ...
                    'DataVars', 'Weight', 'VarNames', {'Gender', 'Smoker', ...
                    'GroupCount', 'LowestWeight', 'HighestWeight'})

```

```

statarray =

      Gender  Smoker  GroupCount  LowestWeight  HighestWeight
Female_0    Female    false      40           111           147

```

```
Female_1    Female    true     13          115          146
Male_0      Male      false    26          158          194
Male_1      Male      true     21          164          202
```

### Summary Statistics for a Dataset Array Without Grouping

Load the sample data.

```
load('hospital')
```

The dataset array `hospital` has 100 observations and 7 variables.

Create a dataset array with only the variables `Age`, `Weight`, and `Smoker`.

```
dsa = hospital(:, {'Age', 'Weight', 'Smoker'});
```

The variables `Age` and `Weight` have numeric values, and `Smoker` has logical values.

Compute the mean, minimum, and maximum for the numeric and logical arrays, `Age`, `Weight`, and `Smoker`, with no grouping.

```
statarray = grpstats(dsa, [], {'mean', 'min', 'max'})
```

```
statarray =
```

```
      GroupCount    mean_Age    min_Age    max_Age    mean_Weight
All         100         38.28         25         50         154

      min_Weight    max_Weight    mean_Smoker    min_Smoker    max_Smoker
All         111         202         0.34         false         true
```

The observation name `All` indicates that all observations in `dsa` were used to compute the summary statistics.

### Group Means for a Matrix Using One or More Grouping Variables

Load the sample data.

```
load('carsmall')
```

All variables are measured for 100 cars. `Origin` is the country of origin for each car (France, Germany, Italy, Japan, Sweden, or USA). `Cylinders` has three unique values, 4, 6, and 8, indicating the number of cylinders in each car.

Calculate the mean acceleration, grouped by country of origin.

```
means = grpstats(Acceleration,Origin)
```

```
means =  
    14.4377  
    18.0500  
    15.8867  
    16.3778  
    16.6000  
    15.5000
```

`means` is a 6-by-1 vector of mean accelerations, where each value corresponds to a country of origin.

Calculate the mean acceleration, grouped by both country of origin and number of cylinders.

```
means = grpstats(Acceleration,{Origin,Cylinders})
```

```
means =  
    17.0818  
    16.5267  
    11.6406  
    18.0500  
    15.9143  
    15.5000  
    16.3375  
    16.7000  
    16.6000  
    15.5000
```

There are 18 possible combinations of grouping variable values because `Origin` has 6 unique values and `Cylinders` has 3 unique values. Only 10 of the possible combinations appear in the data, so `means` is a 10-by-1 vector of group means corresponding to the observed combinations of values.

Return the group names along with the mean acceleration for each group.

```
[means,grps] = grpstats(Acceleration,{Origin,Cylinders},{'mean','gname'})
```

```
means =
```

```
17.0818
16.5267
11.6406
18.0500
15.9143
15.5000
16.3375
16.7000
16.6000
15.5000
```

```
grps =
```

```
10x2 cell array
```

```
{'USA'   } {'4'}
{'USA'   } {'6'}
{'USA'   } {'8'}
{'France'} {'4'}
{'Japan' } {'4'}
{'Japan' } {'6'}
{'Germany'} {'4'}
{'Germany'} {'6'}
{'Sweden' } {'4'}
{'Italy'  } {'4'}
```

The output `grps` shows the 10 observed combinations of grouping variable values. For example, the mean acceleration of 4-cylinder cars made in France is 18.05.



## Multiple Summary Statistics for a Matrix Organized by Group

Load the sample data.

```
load carsmall
```

The variable `Acceleration` was measured for 100 cars. The variable `Origin` is the country of origin for each car (France, Germany, Italy, Japan, Sweden, or USA).

Return the minimum and maximum acceleration grouped by country of origin.

```
[grpMin,grpMax,grp] = grpstats(Acceleration,Origin,{'min','max','gname'})
```

```
grpMin =
```

```
8.0000
15.3000
13.9000
12.2000
15.7000
15.5000
```

```
grpMax =
```

```
22.2000
21.9000
18.2000
24.6000
17.5000
15.5000
```

```
grp =
```

```
6x1 cell array
```

```
{'USA' }
{'France' }
{'Japan' }
{'Germany' }
{'Sweden' }
{'Italy' }
```

The sample car with the lowest acceleration is made in the USA, and the sample car with the highest acceleration is made in Germany.

### Plot Prediction Intervals for a New Observation in Each Group

Load the sample data.

```
load('carsmall')
```

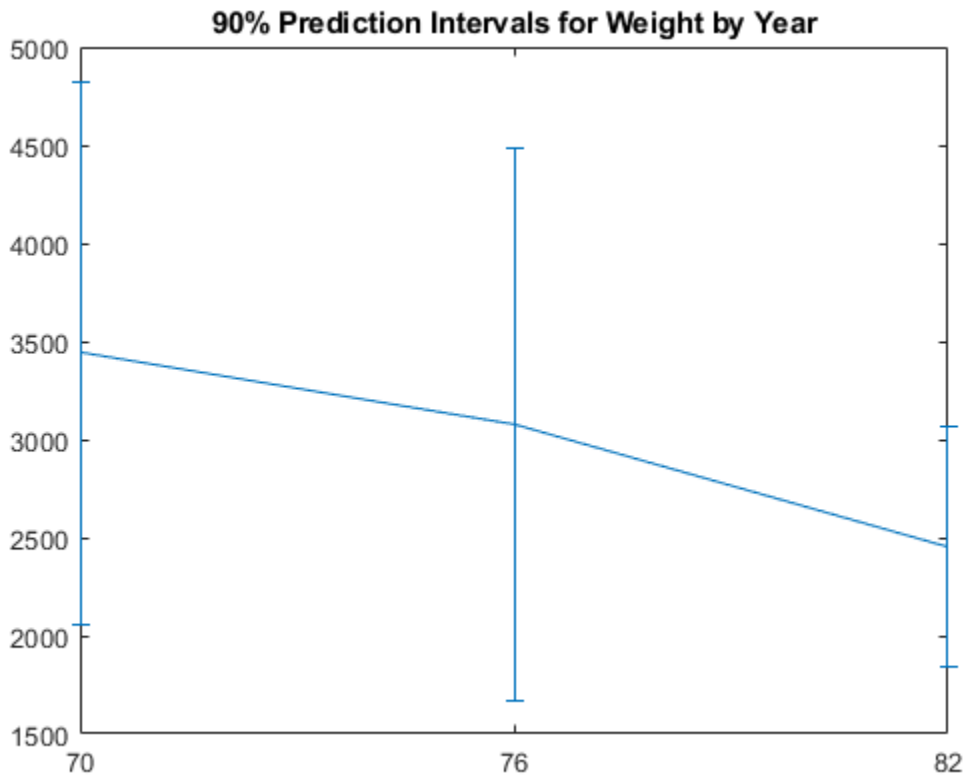
The variable `Weight` was measured for 100 cars. The variable `Model_Year` has three unique values, 70, 76, and 82, which correspond to model years 1970, 1976, and 1982.

Calculate the mean weight and 90% prediction intervals for each model year.

```
[means,pred,grp] = grpstats(Weight,Model_Year,...  
                           {'mean','predci','gname'},'Alpha',0.1);
```

Plot error bars showing the mean weight and 90% prediction intervals, grouped by model year. Label the horizontal axis with the group names.

```
ngrps = length(grp); % Number of groups  
  
errorbar(1:ngrps)',means,pred(:,2)-means)  
set(gca,'xtick',1:ngrps,'xticklabel',grp)  
title('90% Prediction Intervals for Weight by Year')
```



### Plot Group Means and Confidence Intervals

Load the sample data.

```
load('carsmall')
```

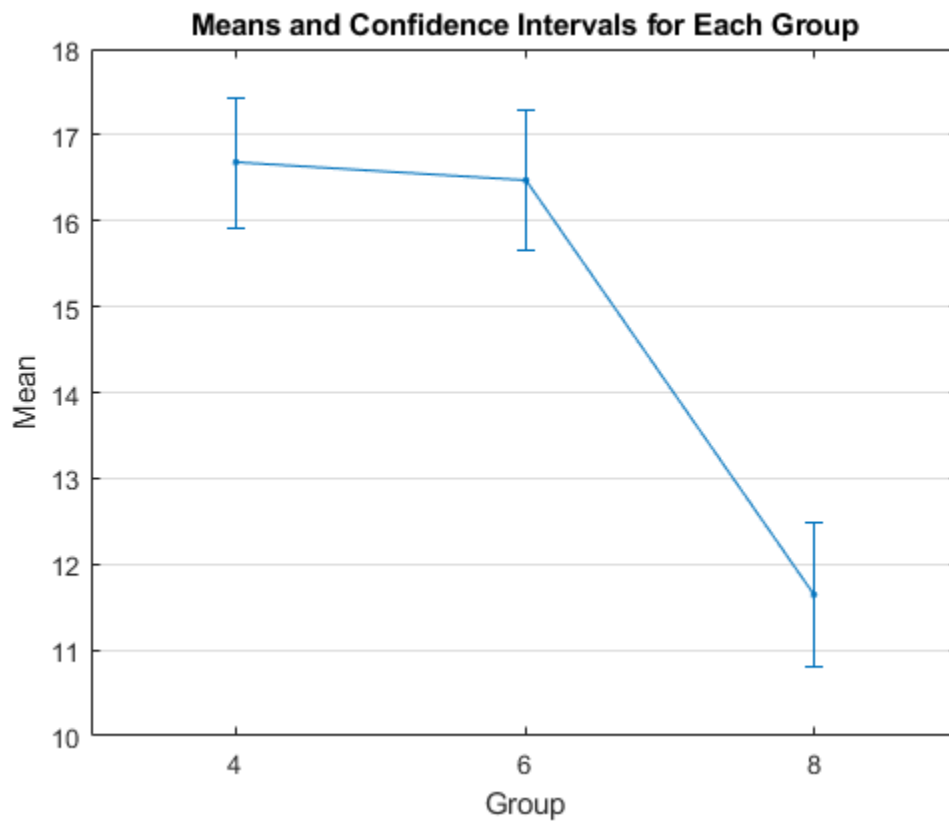
The variables `Acceleration` and `Weight` are the acceleration and weight values measured for 100 cars. The variable `Cylinders` is the number of cylinders in each car. The variable `Model_Year` has three unique values, 70, 76, and 82, which correspond to model years 1970, 1976, and 1982.

Plot mean acceleration, grouped by `Cylinders`, with 95% confidence intervals.

```
grpstats(Acceleration,Cylinders,0.05)
```

```
ans =
```

```
16.6706  
16.4765  
11.6406
```



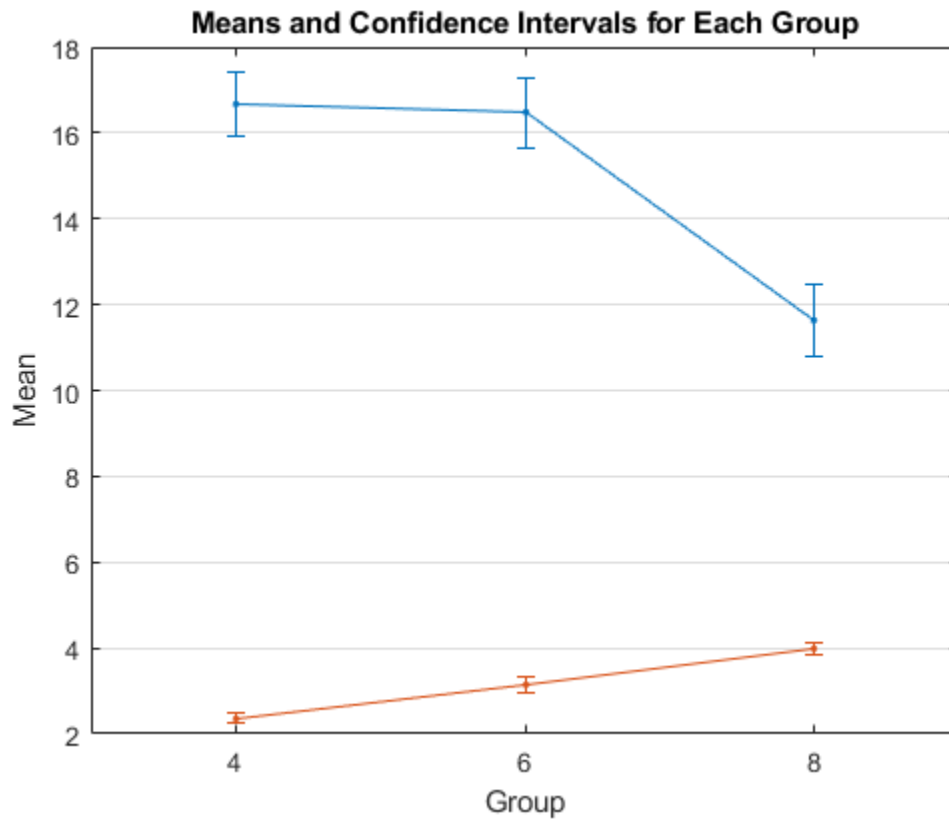
The mean acceleration for cars with 8 cylinders is significantly lower than for cars with 4 or 6 cylinders.

Plot mean acceleration and weight, grouped by Cylinders, and 95% confidence intervals. Scale the Weight values by 1000 so the means of Weight and Acceleration are the same order of magnitude.

```
grpstats([Acceleration,Weight/1000],Cylinders,0.05)
```

ans =

16.6706	2.3726
16.4765	3.1255
11.6406	3.9703



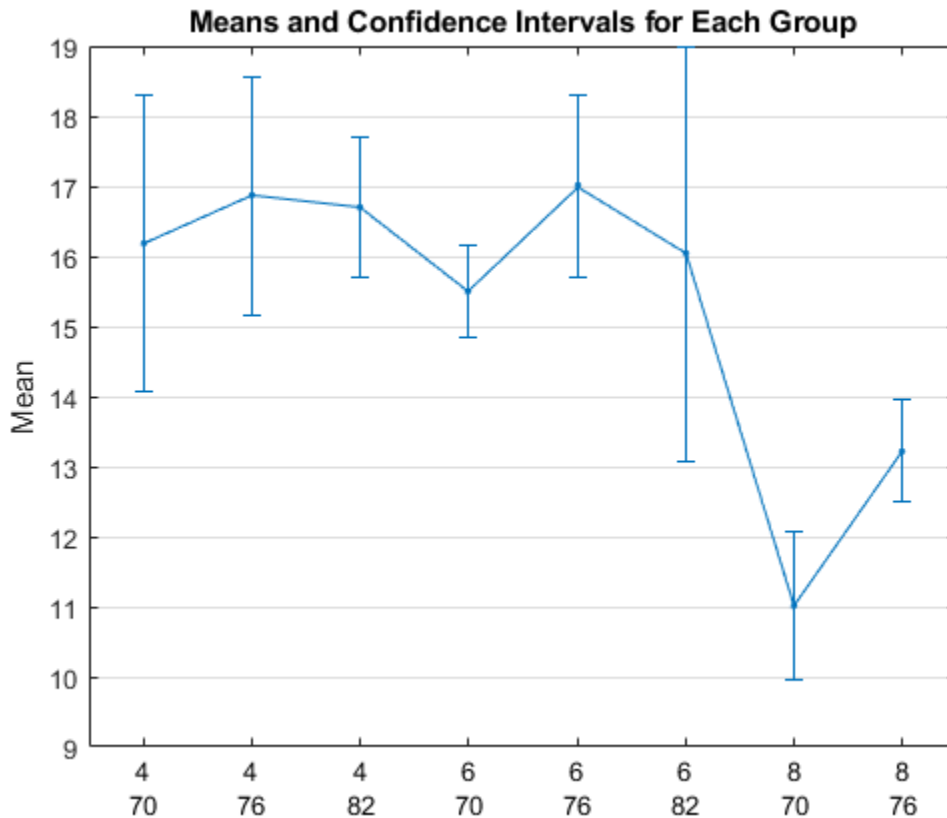
The average weight of cars increases with the number of cylinders, and the average acceleration decreases with the number of cylinders.

Plot mean acceleration, grouped by both `Cylinders` and `Model_Year`. Specify 95% confidence intervals.

```
grpstats(Acceleration, {Cylinders, Model_Year}, 0.05)
```

```
ans =
```

```
16.1875  
16.8667  
16.7036  
15.5000  
17.0000  
16.0333  
11.0217  
13.2222
```



There are nine possible combinations of grouping variable values because there are three unique values in `Cylinders` and three unique values in `Model_Year`. The plot does not show 8-cylinder cars with model year 1982 because the data did not include this combination.

The mean acceleration of 8-cylinder cars made in 1976 is significantly larger than the mean acceleration of 8-cylinder cars made in 1970.

- “Summary Statistics Grouped by Category” on page 2-44
- “Test Differences Between Category Means” on page 2-34
- “Plot Data Grouped by Category” on page 2-29
- “Calculations on Dataset Arrays” on page 2-117

## Input Arguments

### **tbl** — Input data

table | dataset array

Input data, specified as a table or dataset array. `tbl` must include at least one variable that is a grouping variable.

Summary statistics can only be calculated for variables that have a numeric or logical data type. If any variables in `tbl` (other than the grouping variables) are not numeric or logical arrays, then use the name-value pair argument `DataVars` to specify the names or column numbers of the numeric and logical variables for which to calculate summary statistics.

### **groupvar** — Identifiers for the grouping variables

cell array of character vectors | vector of positive integers | logical vector | []

Identifiers for the grouping variables in the input data, `tbl`, specified as one of the following:

Character vector or cell array of character vectors	Names of the grouping variables
Positive integer or vector of positive integers	Variable numbers of the grouping variables
Vector of logical values with number of elements equal to the number of variables in <code>tbl</code>	Logical indicator with value <code>true</code> for grouping variables and <code>false</code> otherwise
[]	No groups (returns summary statistics for all data)

Any variable that is identified by `groupvar` as a grouping variable must have a valid grouping variable data type: categorical array, logical or numeric vector, datetime or duration vector, or cell array of character vectors.

For example, consider an input table, `tbl`, with six variables. The fourth variable is named `Gender`. To be a valid grouping variable, the data type of `Gender` might be a cell array of character arrays or a nominal array, with the unique values `Male` and `Female`. To specify the variable `Gender` as the grouping variable, you can use any of these syntaxes:



- `statarray = grpstats(tbl, 'Gender')`
- `statarray = grpstats(tbl, 4)`
- `statarray = grpstats(tbl, logical([0 0 0 1 0 0]))`

Data Types: `double` | `logical` | `cell` | `char`

### **whichstats** — Types of summary statistics

character vector | function handle

Types of summary statistics to compute, specified as a character vector or function handle, or a cell array of character vectors and function handles. Use a cell array to specify multiple types of summary statistics.

Values include:

'mean'	Mean
'sem'	Standard error of the mean
'numel'	Count, or number, of non-NaN elements
'gname'	Group name
'std'	Standard deviation
'var'	Variance
'min'	Minimum
'max'	Maximum
'range'	Range
'meanci'	95% confidence interval for the mean
'predci'	95% prediction interval for a new observation

Example: `[stat1, stat2] = grpstats(X, group, {'mean', 'sem'})`

You can specify different significance levels for the 'meanci' and 'predci' options using the name-value pair argument, `Alpha`.

To specify other types of summary statistics, you can use function handles. You can use the handle to any function that accepts a column or matrix of data, and returns the same size output each time `grpstats` calls it (even if the output for some groups is empty).

If the function accepts a column of data, then the function can return either a scalar value, or an *nvals*-by-1 column vector for descriptive statistics of length *nvals* (for

example, confidence intervals have length two). If the function accepts a matrix, it must either return a 1-by-*ncols* row vector, or an *nvals*-by-*ncols* matrix, where *ncols* is the number of columns in the input data matrix.

```
Example: [stat1,stat2,stat3] = grpstats(X,group,  
{'mean','std',@skewness})
```

For functions that do not compute column-wise statistics, specify the computation direction while specifying the function.

```
Example: stat1 = grpstats(X,group,@(x)sum(x,1))
```

Data Types: char | function\_handle

### **alpha** — Significance level

scalar value in the range (0,1)

Significance level, specified as a scalar value in the range (0,1).

- When you specify 'meanci' or 'predci' in *whichstats*, you can use *alpha* to specify the significance level for the confidence or prediction intervals. If you specify *alpha*, then *grpstats* returns  $100 \times (1 - \text{alpha})\%$  confidence or prediction intervals. If you do not specify *alpha*, then *grpstats* returns 95% intervals ( $\text{alpha} = 0.05$ ).
- Use *alpha* with the syntax on page 32-0 to plot group means and corresponding  $100 \times (1 - \text{alpha})\%$  confidence intervals.

Data Types: double

### **x** — Input data

vector | matrix

Input data, specified as a vector or a matrix. If *x* is a matrix, then *grpstats* returns summary statistics for each column of *x*.

Data Types: double | single

### **group** — Grouping variable

categorical array | logical or numeric vector | datetime or duration vector | cell array of character vectors | []

Grouping variable, specified as a categorical array, logical or numeric vector, or cell array of character vectors. Each unique value in a grouping variable defines a group. *grpstats* groups data for summary statistics using the grouping variable values.

There must be a grouping variable value for each row of the input data  $X$ . Observations (rows) with the same value of the grouping variable are in the same group. Use `[]` to compute summary statistics for all data, without using groups.

For example, if `Gender` is a cell array of character vectors with values `'Male'` and `'Female'`, you can use `Gender` as a grouping variable to summarize your data by gender.

You can also use more than one grouping variable to group data for summary statistics. In this case, specify a cell array of grouping variables.

For example, if `Smoker` is a logical vector with values 0 for nonsmokers and 1 for smokers, then specifying the cell array `{Gender, Smoker}` divides observations into four groups: Male Smoker, Male Nonsmoker, Female Smoker, and Female Nonsmoker. `grpstats` returns summary statistics only for the combinations of values that exist in the input grouping variables (not all possible combinations).

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'DataVars', [1, 3, 4], 'Alpha', 0.01` specifies that summary statistics be calculated for the 1st, 3rd, and 4th variables in a dataset array, with 99% confidence intervals.

### **Alpha** — Significance level

0.05 (default) | scalar value in the range (0,1)

Significance level for confidence and prediction intervals, specified as the comma-separated pair consisting of `'Alpha'` and a scalar value in the range (0,1).

When you include `'meanci'` or `'predci'` in `whichstats`, you can use `Alpha` to specify the significance level for confidence or prediction intervals. If you specify the value  $a$ , then `grpstats` returns  $100 \times (1 - a)\%$  confidence or prediction intervals.

If you do not specify a value for `Alpha`, then `grpstats` returns 95% intervals ( $a = 0.05$ ).

Example: `'Alpha', 0.1`

Data Types: `double`

**DataVars** — Variable names or columns

cell array of character vectors | vector of positive integers | logical vector

Variable names or columns indicating which variables in the input data `tbl` you want to compute summary statistics for, specified as the comma-separated pair consisting of 'DataVars' and a cell array of character vectors, vector of positive integers, or a logical vector. Use a character vector to specify a variable name, a positive integer to specify a variable column number, or logical values to indicate which variables to include (`true` if you want to compute summary statistics, `false` otherwise).

You must specify `DataVars` if there are any variables in `tbl` (other than the grouping variables specified in `groupvar`) that are not numeric or logical arrays. Summary statistics can only be calculated for variables that have a numeric or logical data type.

Example: `'DataVars', {'Height', 'Weight'}`

Data Types: `double` | `cell` | `char`

**VarNames** — Variable names for output

cell array of character vectors

Variable names for the output `statarray`, specified as the comma-separated pair consisting of 'VarNames' and a cell array of character vectors. By default, `grpstats` constructs output variable names by appending a prefix to the variable names from the input data `tbl`. This prefix corresponds to the summary statistic name.

Example: `'VarNames', {'Gender', 'GroupCount', 'MaleMean', 'FemaleMean'}`

Data Types: `cell`

## Output Arguments

**statarray** — Group summary statistics

table | dataset array

Group summary statistics, returned as a table or a dataset array. If `tbl` is a table, `grpstats` returns `statarray` as a table. If `tbl` is a dataset array, `grpstats` returns `statarray` as a dataset array.

`statarray` contains summary statistic values for the groups of data in `tbl` determined by the levels of the grouping variables specified by `groupvar`. There is a row in `statarray` for each observed value or combination of values in the variables specified by `groupvar`. The output `statarray` contains:

- All grouping variables specified by `groupvar`.
- The variable `GroupCount`, containing the number of observations in each group.
- Group summary statistic values for all variables in `tbl` (other than those specified by `groupvar`), or for only the variables specified using `DataVars`.

The total number of variables in `statarray` is  $n\text{groupvars} + 1 + n\text{datavars} \times n\text{stats}$ , where  $n\text{groupvars}$  is the number of variables in `groupvar`,  $n\text{datavars}$  is the number of variables for which summary statistics are computed, and  $n\text{stats}$  is the number of summary statistic types specified in `whichstats`.

`grpstats` assigns default names to the variables in `statarray`, unless you specify variable names using the name-value pair argument `VarNames`.

### **means** — Group means

column vector | array

Group means for the groups of data in the vector or matrix `X` determined by the levels of `group`, returned as an  $n\text{groups}$ -by- $n\text{cols}$  array. Here,  $n\text{groups}$  is the number of unique values in the grouping variable, and  $n\text{cols}$  is the number of columns in `X`. If `X` is a vector, then `means` is a column vector.

### **stats1, ..., statsN** — Group summary statistics

column vectors | arrays

Group summary statistics for the groups of data in the vector or matrix `X` determined by the levels of `group`, returned as  $n\text{groups}$ -by- $n\text{cols}$  arrays. Here,  $n\text{groups}$  is the number of unique values in the grouping variable, and  $n\text{cols}$  is the number of columns in `X`. You must specify an output argument for each type of summary statistic specified in `whichstats`.

If a summary statistic type in `whichstats` returns a value of length  $n\text{vals}$  (for example, a confidence interval is a descriptive statistic of length two), then the corresponding output argument is an  $n\text{groups}$ -by- $n\text{cols}$ -by- $n\text{vals}$  array.

## Algorithms

- `grpstats` treats NaNs as missing values, and removes them from the input data before calculating summary statistics.
- `grpstats` ignores empty group names.

## Extended Capabilities

### Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function supports tall arrays for out-of-memory data with some limitations.

- If the input data is a tall array, then all grouping variables must also be tall and have the same number of rows as the data.
- The `whichstats` option cannot be specified as a function handle. In addition to the current built-in options, `whichstats` can also be:
  - `'Count'` — Number of non-NaN.
  - `'NNZ'` — Number of nonzeros and non-NaN.
  - `'Kurtosis'` — Compute kurtosis.
  - `'Skewness'` — Compute skewness.
  - `'all-stats'` — Compute all summary statistics.
- Group order is not guaranteed to be the same as the in-memory `grpstats` computation.
- Summary statistics for *nonnumeric* variables return NaNs.
- `grpstats` always operates on the first dimension.
- If the input is a tall table, then the output is also a tall table. However, rather than including row names, the output tall table contains an extra variable `GroupLabel` that contains the same information.

For more information, see “Tall Arrays” (MATLAB).

## See Also

dataset | table

## Topics

“Summary Statistics Grouped by Category” on page 2-44

“Test Differences Between Category Means” on page 2-34

“Plot Data Grouped by Category” on page 2-29

“Calculations on Dataset Arrays” on page 2-117

“Dataset Arrays” on page 2-142

“Grouping Variables” on page 2-59

“Categorical Arrays” on page 2-48

**Introduced before R2006a**

## grpstats

**Class:** RepeatedMeasuresModel

Compute descriptive statistics of repeated measures data by group

## Syntax

```
statstbl = grpstats(rm,g)
statstbl = grpstats(rm,g,stats)
```

## Description

`statstbl = grpstats(rm,g)` returns the count, mean, and variance for the data used to fit the repeated measures model `rm`, grouped by the factors, `g`.

`statstbl = grpstats(rm,g,stats)` returns the statistics specified by `stats` for the data used to fit the repeated measures model `rm`, grouped by the factors, `g`.

## Input Arguments

**rm** — Repeated measures model

RepeatedMeasuresModel object

Repeated measures model, returned as a RepeatedMeasuresModel object.

For properties and methods of this object, see RepeatedMeasuresModel.

**g** — Name of grouping factor or factors

character vector | cell array of character vectors

Name of grouping factor or factors, specified as a character vector or cell array of character vectors.

Example: 'Drug'



Example: {'Drug', 'Sex'}

Data Types: char

### **stats** — Statistics to compute

character vector | function handle | cell array of multiple character vectors and function handles

Statistics to compute, specified as one of the following:

- Character vector specifying the name of the statistics to compute. Names can be one of the following.

Name	Description
'mean'	Mean
'sem'	Standard error of the mean
'numel'	Count or number of elements
'gname'	Group name
'std'	Standard deviation
'var'	Variance
'min'	Minimum
'max'	Maximum
'range'	Maximum minus minimum
'meanci'	95% confidence interval for the mean
'predci'	95% prediction interval for a new observation

- Function handle — The function you specify must accept a vector of response values for a single group, and compute descriptive statistics for it. A function should typically return a value that has one row. A function must return the same size output each time `grpstats` calls it, even if the input for some groups is empty.
- A cell array of character vectors and function handles.

Example: @median

Example: @skewness

Example: 'gname'

Example: {'gname', 'range', 'predci'}

Data Types: `char` | `function_handle` | `cell`

## Output Arguments

**statstbl** — Statistics values for each group  
table

Statistics values for each group, returned as a table.

## Examples

### Compute Group Statistics

Load the sample data.

```
load fisheriris
```

The column vector, `species` consists of iris flowers of three different species: `setosa`, `versicolor`, and `virginica`. The double matrix `meas` consists of four types of measurements on the flowers: the length and width of sepals and petals in centimeters, respectively.

Store the data in a table array.

```
t = table(species,meas(:,1),meas(:,2),meas(:,3),meas(:,4),...  
'VariableNames',{'species','meas1','meas2','meas3','meas4'});  
Meas = dataset([1 2 3 4]','VarNames',{'Measurements'});
```

Fit a repeated measures model, where the measurements are the responses and the species is the predictor variable.

```
rm = fitrm(t,'meas1-meas4~species','WithinDesign',Meas);
```

Compute group counts, mean, and standard deviation with respect to species.

```
grpstats(rm,'species')
```

```
ans =
```

```
3x4 table
```

species	GroupCount	mean	std
'setosa'	200	2.5355	1.8483
'versicolor'	200	3.573	1.7624
'virginica'	200	4.285	1.9154

Now, compute the range of data and 95% confidence intervals for the group means for the factor species. Also display the group name.

```
grpstats(rm, 'species', {'gname', 'range', 'predci'})
```

```
ans =
```

```
3x5 table
```

species	gname	GroupCount	range	predci	
'setosa'	'setosa'	200	5.7	-1.1185	6.1895
'versicolor'	'versicolor'	200	6	0.088976	7.057
'virginica'	'virginica'	200	6.5	0.4985	8.0715

## Statistics for Data Grouped by Two Factors

Load the sample data.

```
load repeatedmeas
```

The table `between` includes the between-subject variables `age`, `IQ`, `group`, `gender`, and eight repeated measures `y1` through `y8` as responses. The table `within` includes the within-subject variables `w1` and `w2`. This is simulated data.

Fit a repeated measures model, where the repeated measures `y1` through `y8` are the responses, and `age`, `IQ`, `group`, `gender`, and the group-gender interaction are the predictor variables. Also specify the within-subject design matrix.

```
rm = fitrm(between, 'y1-y8 ~ Group*Gender + Age + IQ', 'WithinDesign', within);
```

Compute group counts, mean, standard deviation, skewness, and kurtosis of data grouped by the factors Group and Gender.

```
GS = grpstats(rm, {'Group', 'Gender'}, {'mean', 'std', @skewness, @kurtosis})
```

```
GS =
```

```
6x7 table
```

Group	Gender	GroupCount	mean	std	skewness	kurtosis
A	Female	40	16.554	21.498	0.35324	3.7807
A	Male	40	9.8335	20.602	-0.38722	2.7834
B	Female	40	11.261	25.779	-0.49177	4.1484
B	Male	40	3.6078	24.646	0.55447	2.7966
C	Female	40	-11.335	27.186	1.7499	6.1429
C	Male	40	-14.028	31.984	1.7362	5.141

## Tips

- `grpstats` computes results separately for each group. The results do not depend on the fitted repeated measures model. It computes the results on all available data, without omitting entire rows that contain NaNs.

## See Also

`fitrm` | `plot`

## gscatter

Scatter plot by group

### Syntax

```
gscatter(x,y,group)
gscatter(x,y,group,clr,sym,siz)
gscatter(x,y,group,clr,sym,siz,doleg)
gscatter(x,y,group,clr,sym,siz,doleg,xnam,ynam)
h = gscatter(...)
```

### Description

`gscatter(x,y,group)` creates a scatter plot of `x` and `y`, grouped by `group`. `x` and `y` are vectors of the same size. `group` is a grouping variable in the form of a categorical variable, vector, character array, or cell array of character vectors. Alternatively, `group` can be a cell array containing several grouping variables (such as `{g1 g2 g3}`), in which case observations are in the same group if they have common values of all grouping variables. Points in the same group appear on the graph with the same marker and color.

`gscatter(x,y,group,clr,sym,siz)` specifies the color, marker type, and size for each group. `clr` is either a character array of colors recognized by the `plot` function or a three-column matrix of color specifications. `sym` is a character array of symbols recognized by the `plot` command, with the default value `'.'`. `siz` is a vector of sizes, with the default value determined based on the number of observations. If you do not specify enough values for all groups, `gscatter` cycles through the specified values as needed.

`gscatter(x,y,group,clr,sym,siz,doleg)` controls whether a legend is displayed on the graph (`doleg` is `'on'`, the default) or not (`doleg` is `'off'`).

`gscatter(x,y,group,clr,sym,siz,doleg,xnam,ynam)` specifies the name to use for the `x`-axis and `y`-axis labels. If the `x` and `y` inputs are simple variable names and `xnam` and `ynam` are omitted, `gscatter` labels the axes with the variable names.

`h = gscatter(...)` returns an array of handles to the lines on the graph.

## Examples

### Scatter Plot of Climate and Housing Ratings

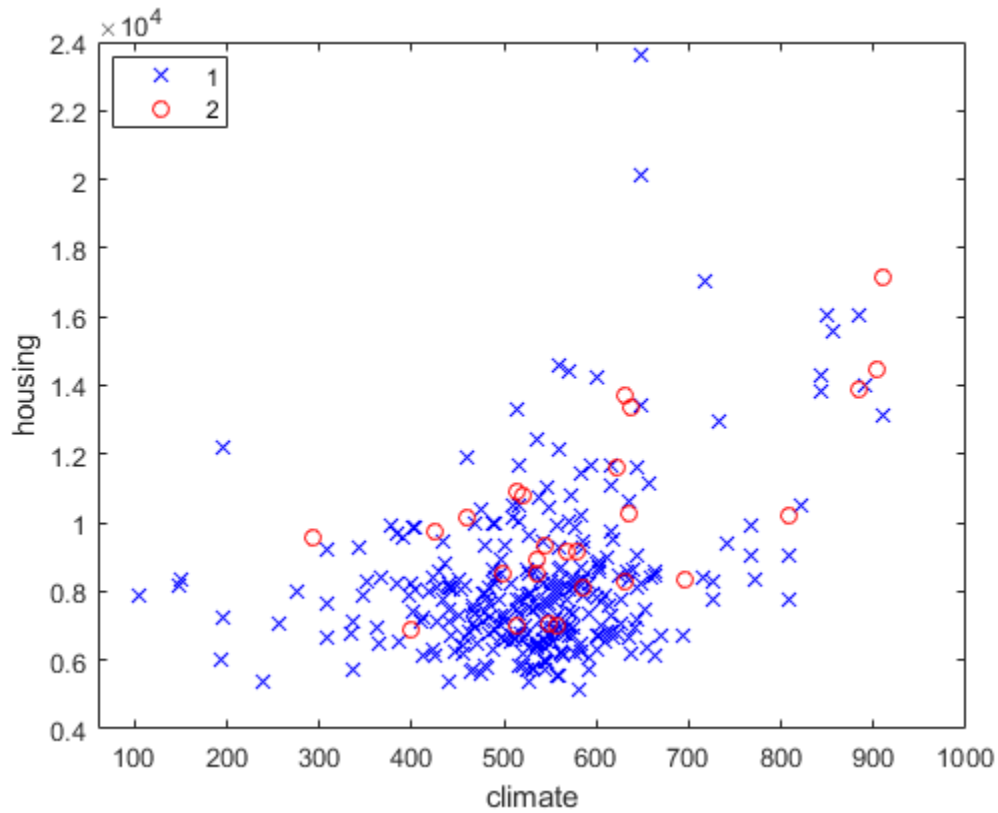
Load the sample data.

```
load discrim
```

The sample data contains ratings of cities according to nine factors such as climate, housing, education, and health in the matrix `ratings`.

Plot the relationship between the ratings for climate (first column) and housing (second column) grouped by city size in the matrix `group`. Choose different colors and plotting symbols for each group.

```
figure;  
gscatter(ratings(:,1), ratings(:,2), group, 'br', 'xo')  
xlabel('climate');  
ylabel('housing');
```



## See Also

`gplotmatrix` | `grpstats` | `scatter`

## Topics

“Grouping Variables” on page 2-59

Introduced before R2006a

## gt

**Class:** grandstream

Greater than relation for handles

## Syntax

```
h1 > h2
```

## Description

`h1 > h2` performs element-wise comparisons between handle arrays `h1` and `h2`. `h1` and `h2` must be of the same dimensions unless one is a scalar. The result is a logical array of the same dimensions, where each element is an element-wise `>` result.

If one of `h1` or `h2` is scalar, scalar expansion is performed and the result will match the dimensions of the array that is not scalar.

`tf = gt(h1, h2)` stores the result in a logical array of the same dimensions.

## See Also

`eq` | `ge` | `le` | `lt` | `ne` | `grandstream`



# haltonset class

## Superclasses:

Halton quasi-random point sets

## Description

`haltonset` is a quasi-random point set class that produces points from the Halton sequence.

## Construction

`haltonset`      Construct Halton quasi-random point set

## Methods

### Inherited Methods

Methods in the following table are inherited from `grandset` on page 32-5245.

<code>disp</code>	Display <code>grandset</code> object
<code>end</code>	Last index in indexing expression for point set
<code>length</code>	Length of point set
<code>ndims</code>	Number of dimensions in matrix
<code>net</code>	Generate quasi-random point set
<code>scramble</code>	Scramble quasi-random point set
<code>size</code>	Number of dimensions in matrix
<code>suboref</code>	Subscripted reference for <code>grandset</code>

## Properties

### Inherited Properties

Properties in the following table are inherited from `qrandset` on page 32-5245.

Dimensions	Number of dimensions
Leap	Interval between points
ScrambleMethod	Settings that control scrambling
Skip	Number of initial points to omit from sequence
Type	Name of sequence on which point set $P$ is based

### Copy Semantics

`Handle`. To learn how this affects your use of the class, see [Comparing Handle and Value Classes \(MATLAB\)](#) in the MATLAB Object-Oriented Programming documentation.

### References

[1] Kocis, L., and W. J. Whiten, "Computational Investigations of Low-Discrepancy Sequences," *ACM Transactions on Mathematical Software*, Vol. 23, No. 2, pp. 266-294, 1997.

### See Also

`sobolset`

### Topics

"Quasi-Random Point Sets" on page 7-19

# haltonset

**Class:** haltonset

Construct Halton quasi-random point set

## Syntax

```
p = haltonset(d)
p = haltonset(d,prop1, val1, prop2, val2, ...)
```

## Description

`p = haltonset(d)` constructs a  $d$ -dimensional point set `p` of the haltonset on page 32-2987 class, with default property settings.

`p = haltonset(d,prop1, val1, prop2, val2, ...)` specifies property name/value pairs used to construct `p`.

The object `p` returned by `haltonset` encapsulates properties of a specified quasi-random sequence. The point set is finite, with a length determined by the `Skip` and `Leap` properties and by limits on the size of point set indices (maximum value of  $2^{53}$ ). Values of the point set are not generated and stored in memory until you access `p` using `net` or parenthesis indexing.

## Examples

Generate a 3-D Halton point set, skip the first 1000 values, and then retain every 101st point:

```
p = haltonset(3, 'Skip', 1e3, 'Leap', 1e2)
p =
    Halton point set in 3 dimensions (8.918019e+013 points)
    Properties:
        Skip : 1000
```

```
Leap : 100
ScrambleMethod : none
```

Use `scramble` to apply reverse-radix scrambling:

```
p = scramble(p, 'RR2')
p =
  Halton point set in 3 dimensions (8.918019e+013 points)
  Properties:
    Skip : 1000
    Leap : 100
    ScrambleMethod : RR2
```

Use `net` to generate the first four points:

```
X0 = net(p, 4)
X0 =
  0.0928    0.6950    0.0029
  0.6958    0.2958    0.8269
  0.3013    0.6497    0.4141
  0.9087    0.7883    0.2166
```

Use parenthesis indexing to generate every third point, up to the 11th point:

```
X = p(1:3:11, :)
X =
  0.0928    0.6950    0.0029
  0.9087    0.7883    0.2166
  0.3843    0.9840    0.9878
  0.6831    0.7357    0.7923
```

## References

- [1] Kocis, L., and W. J. Whiten. “Computational Investigations of Low-Discrepancy Sequences.” *ACM Transactions on Mathematical Software*. Vol. 23, No. 2, 1997, pp. 266–294.

## See Also

`net` | `scramble` | `sobolset`

**Introduced in R2008a**

## harmmean

Harmonic mean

### Syntax

```
m = harmmean(X)
harmmean(X, dim)
```

### Description

`m = harmmean(X)` calculates the harmonic mean of a sample. For vectors, `harmmean(x)` is the harmonic mean of the elements in `x`. For matrices, `harmmean(X)` is a row vector containing the harmonic means of each column. For  $N$ -dimensional arrays, `harmmean` operates along the first nonsingleton dimension of `X`.

`harmmean(X, dim)` takes the harmonic mean along dimension `dim` of `X`.

The harmonic mean is

$$m = \frac{n}{\sum_{i=1}^n \frac{1}{x_i}}$$

### Examples

The arithmetic mean is greater than or equal to the harmonic mean.

```
x = exprnd(1,10,6);
```

```
harmonic = harmmean(x)
```

```
harmonic =
```

```
    0.3382    0.3200    0.3710    0.0540    0.4936    0.0907
```

```
average = mean(x)
```

```
average =  
    1.3509    1.1583    0.9741    0.5319    1.0088    0.8122
```

## Extended Capabilities

### Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays” (MATLAB).

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`geomean` | `mean` | `median` | `trimmean`

Introduced before R2006a

## hist3

Bivariate histogram

### Syntax

```
hist3(X)
hist3(X,nbins)
hist3(X,ctr)
hist3(X,'Edges',edges)
N = hist3(X,...)
[N,C] = hist3(X,...)
hist3(...,param1,val1,param2,val2,...)
```

### Description

`hist3(X)` bins the elements of the  $m$ -by-2 matrix  $X$  into a 10-by-10 grid of equally spaced containers, and plots a histogram. Each column of  $X$  corresponds to one dimension in the bin grid.

`hist3(X,nbins)` plots a histogram using an `nbins(1)`-by-`nbins(2)` grid of bins. `hist3(X,'Nbins',nbins)` is equivalent to `hist3(X,nbins)`.

`hist3(X,ctr)`, where `ctr` is a two-element cell array of numeric vectors with monotonically non-decreasing values, uses a 2-D grid of bins centered on `ctr{1}` in the first dimension and on `ctr{2}` in the second. `hist3` assigns rows of  $X$  falling outside the range of that grid to the bins along the outer edges of the grid, and ignores rows of  $X$  containing NaNs. `hist3(X,'Ctrs',ctr)` is equivalent to `hist3(X,ctr)`.

`hist3(X,'Edges',edges)`, where `edges` is a two-element cell array of numeric vectors with monotonically non-decreasing values, uses a 2-D grid of bins with edges at `edges{1}` in the first dimension and at `edges{2}` in the second. The  $(i,j)$ th bin includes the value  $X(k,:)$  if

```
edges{1}(i) <= X(k,1) < edges{1}(i+1)
edges{2}(j) <= X(k,2) < edges{2}(j+1)
```



Rows of  $X$  that fall on the upper edges of the grid, `edges{1}(end)` or `edges{2}(end)`, are counted in the  $(I, j)$ th or  $(i, J)$ th bins, where  $I$  and  $J$  are the lengths of `edges{1}` and `edges{2}`. `hist3` does not count rows of  $X$  falling outside the range of the grid. Use `-Inf` and `Inf` in `edges` to include all non-`NaN` values.

`N = hist3(X, ...)` returns a matrix containing the number of elements of  $X$  that fall in each bin of the grid, and does not plot the histogram.

`[N,C] = hist3(X, ...)` returns the positions of the bin centers in a 1-by-2 cell array of numeric vectors, and does not plot the histogram. `hist3(ax, X, ...)` plots onto an axes with handle `ax` instead of the current axes. See the axes reference page for more information about handles to plots.

`hist3(..., param1, val1, param2, val2, ...)` allows you to specify graphics parameter name/value pairs to fine-tune the plot.

## Examples

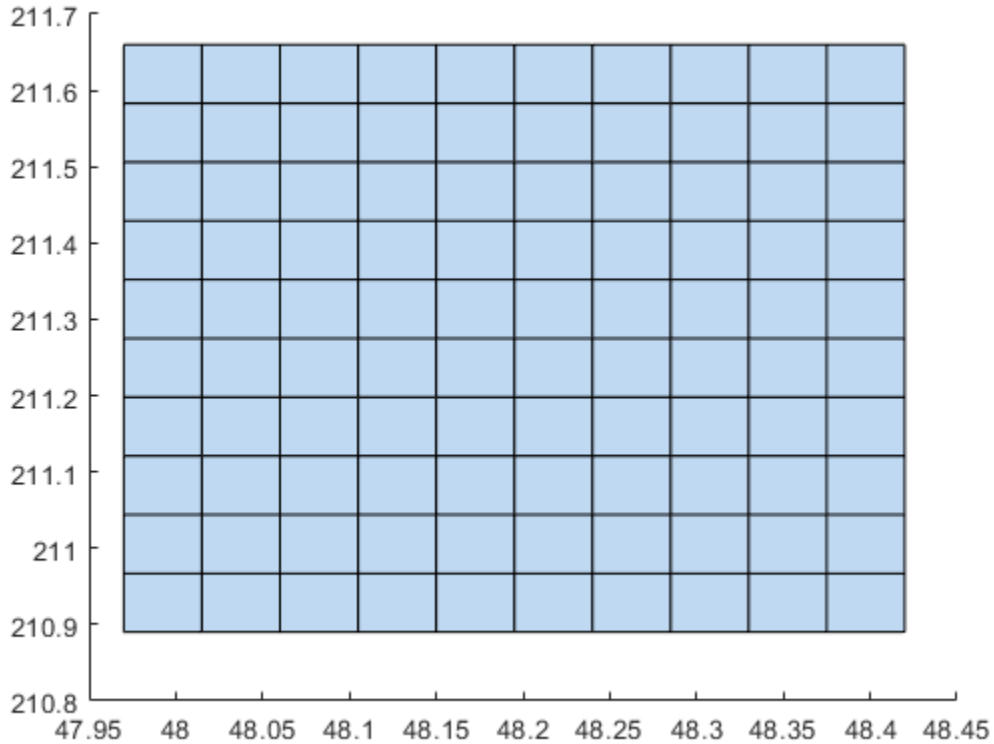
### Plot Density Histogram with Intensity Map

Load the sample data.

```
load seamount
```

Correct grid for negative y-values and draw histogram in 2D.

```
hold on  
dat = [-y,x];  
hist3(dat)
```



Extract histogram data.

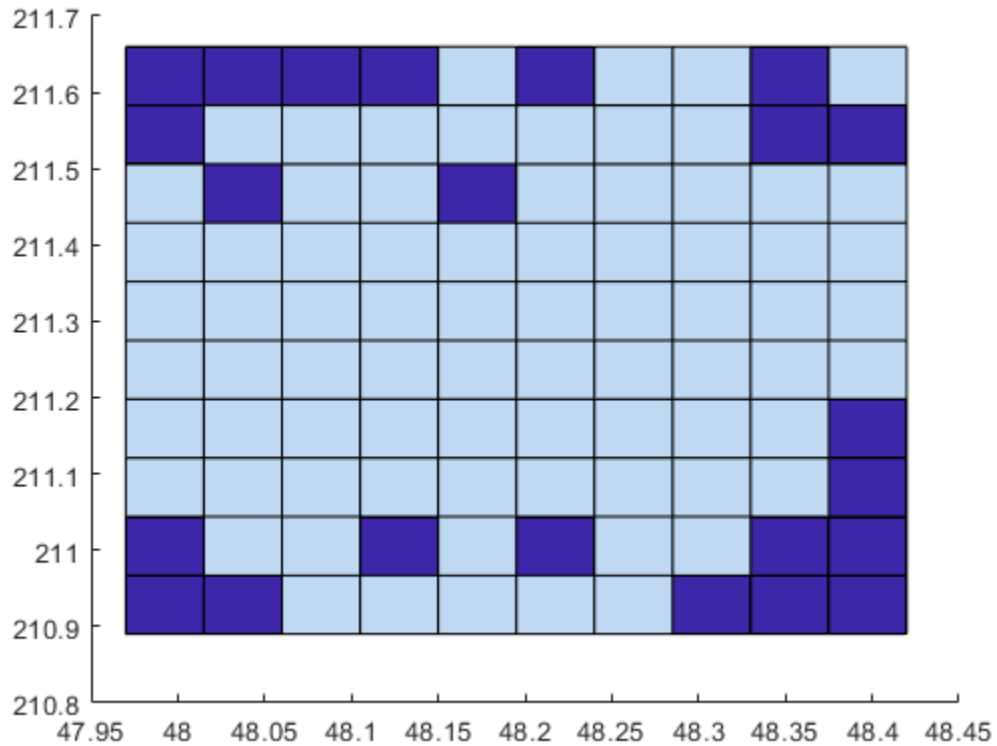
```
n = hist3(dat); % default is to 10x10 bins
n1 = n';
n1(size(n,1) + 1, size(n,2) + 1) = 0;
```

Generate grid for 2-D projected view of intensities.

```
xb = linspace(min(dat(:,1)),max(dat(:,1)),size(n,1)+1);
yb = linspace(min(dat(:,2)),max(dat(:,2)),size(n,1)+1);
```

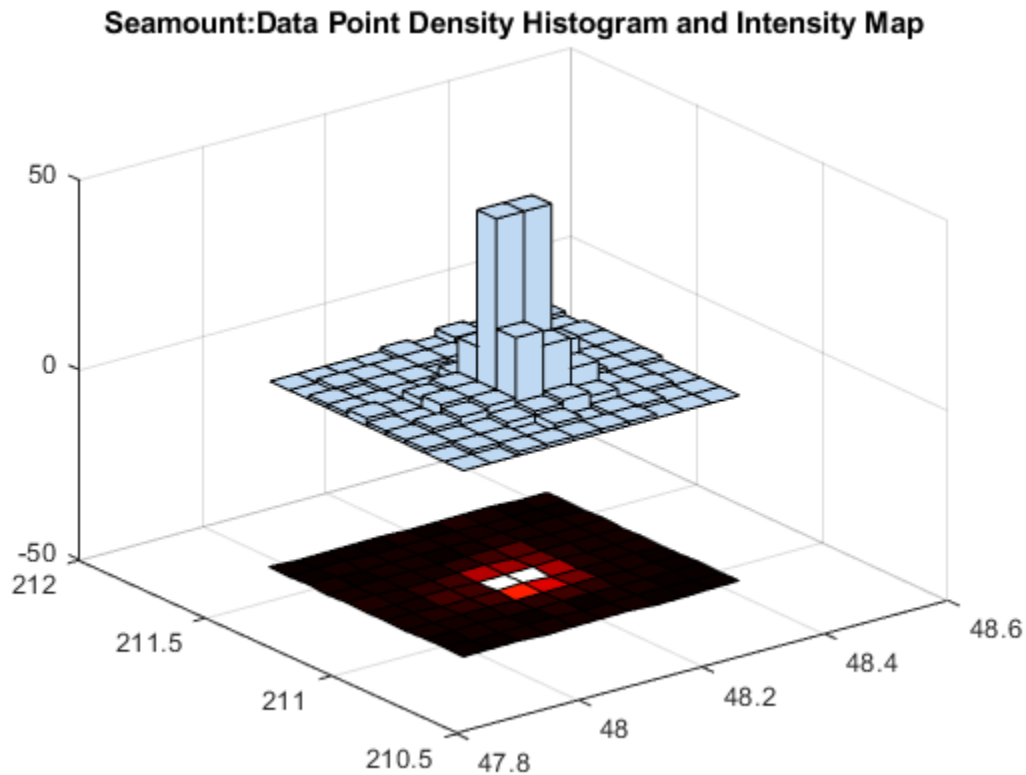
Make a pseudocolor plot.

```
h = pcolor(xb,yb,n1);
```



Set the z-level and colormap of the displayed grid, and display the default 3-D perspective view.

```
h.ZData = ones(size(n1)) * -max(max(n));
colormap(hot) % heat map
title('Seamount:Data Point Density Histogram and Intensity Map');
grid on
view(3);
```



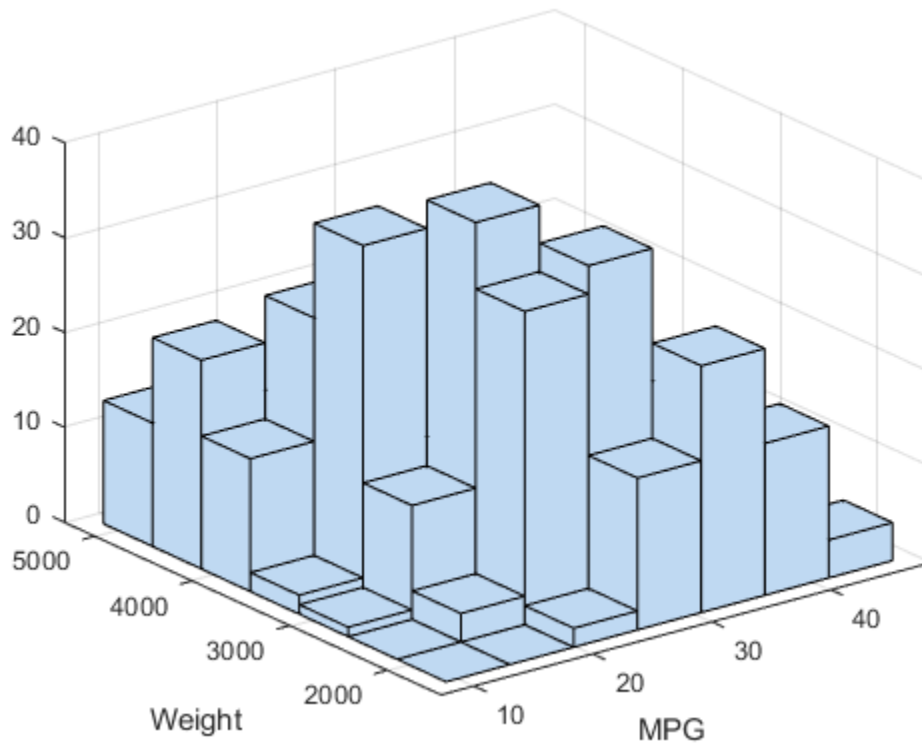
### Histogram with Semi-Transparent Bars

Load the sample data.

```
load carbig
```

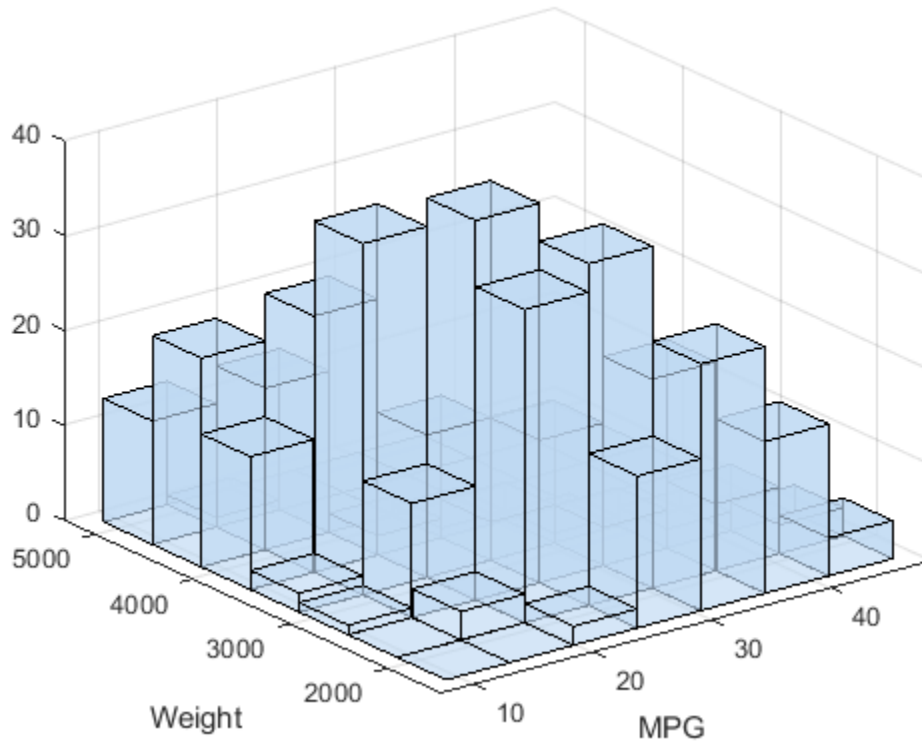
Use the data to make histogram on a 7-by-7 grid of bins.

```
X = [MPG,Weight];  
hist3(X,[7 7]);  
xlabel('MPG'); ylabel('Weight');
```



Make a histogram with semi-transparent bars.

```
hist3(X,[7 7], 'FaceAlpha', .65);  
xlabel('MPG'); ylabel('Weight');  
set(gcf, 'renderer', 'opengl');
```



Specify bin centers, different in each direction; get back counts, but don't make the plot.

```
cnt = hist3(X, {0:10:50 2000:500:5000});
```

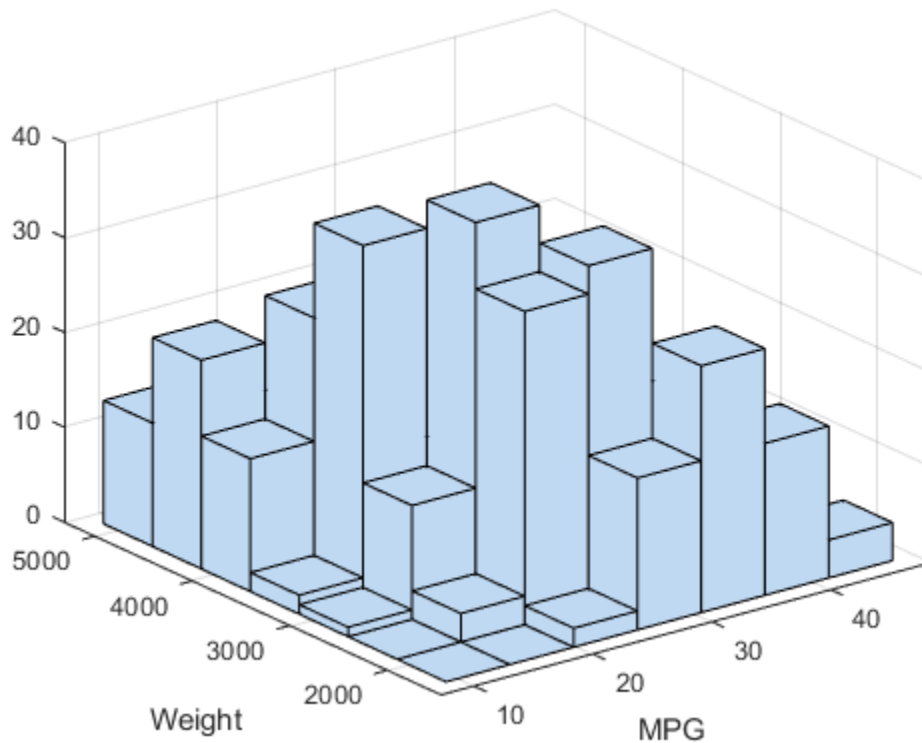
### Histogram Bars Colored According to Height

Load the sample data.

```
load carbig
```

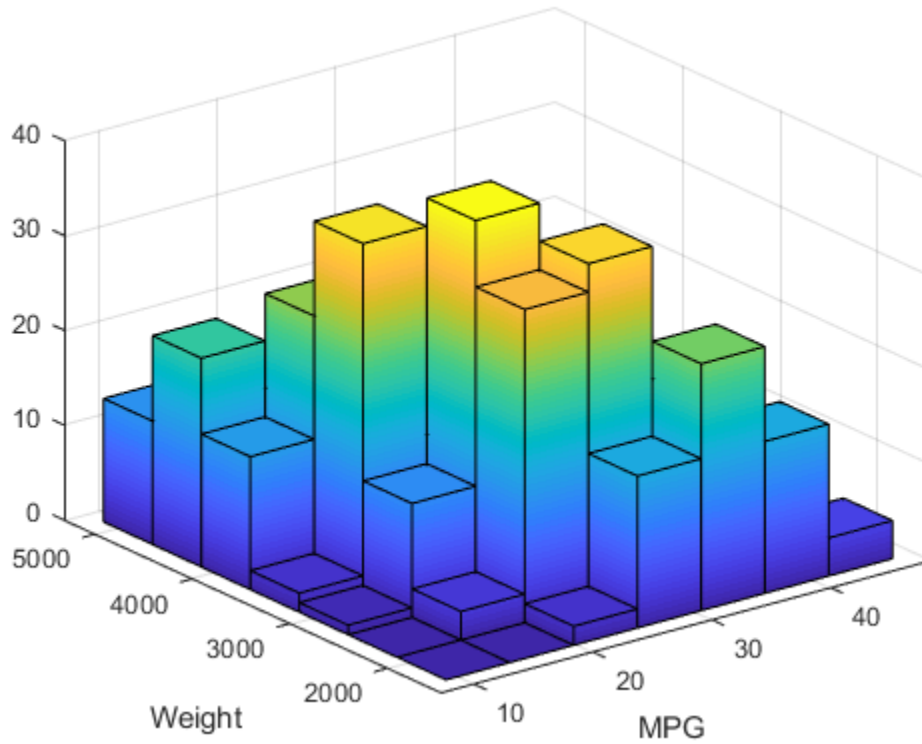
Make a histogram on a 7-by-7 grid of bins.

```
X = [MPG,Weight];  
hist3(X,[7 7]);  
xlabel('MPG'); ylabel('Weight');  
set(gcf,'renderer','opengl');
```



Color the bars based on the frequency of the observations, i.e. according to the height of the bars.

```
set(get(gca,'child'),'FaceColor','interp','CDataMode','auto');
```



## See Also

`accumarray` | `bar` | `bar3` | `histc` | `histogram`

Introduced before R2006a



# histfit

Histogram with a distribution fit

## Syntax

```
histfit(data)
histfit(data,nbins)
histfit(data,nbins,dist)

h = histfit( ___ )
```

## Description

`histfit(data)` plots a histogram of values in `data` using the number of bins equal to the square root of the number of elements in `data` and fits a normal density function.

`histfit(data,nbins)` plots a histogram using `nbins` bins and fits a normal density function.

`histfit(data,nbins,dist)` plots a histogram with `nbins` bins and fits a density function from the distribution specified by `dist`.

`h = histfit( ___ )` returns a vector of handles `h`, where `h(1)` is the handle to the histogram and `h(2)` is the handle to the density curve. It can include any of the input arguments in previous syntaxes.

## Examples

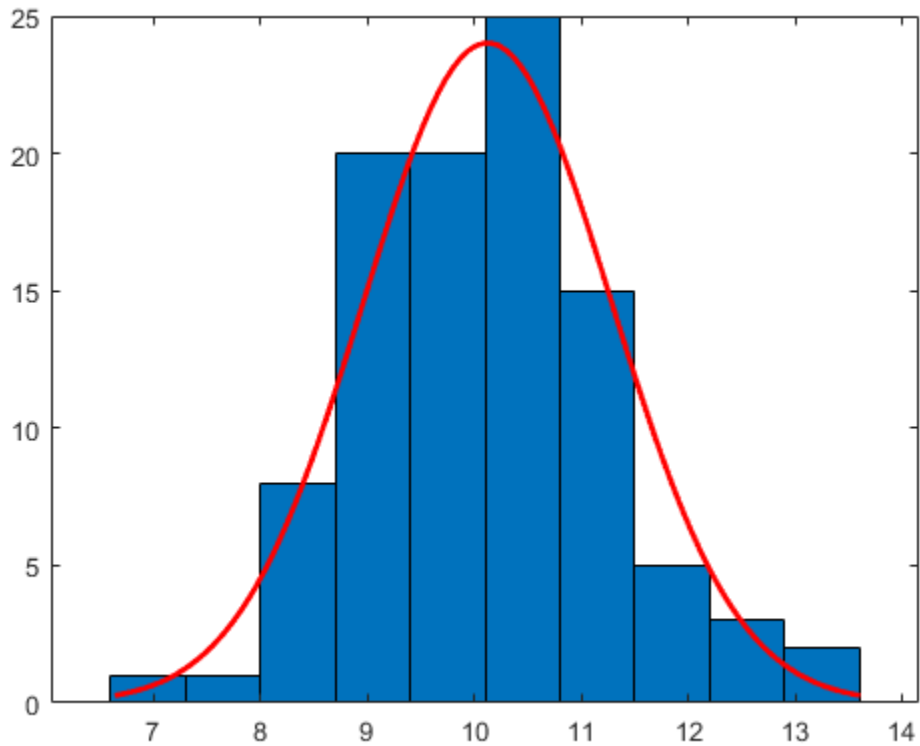
### Histogram with a Normal Distribution Fit

Generate a sample of size 100 from a normal distribution with mean 10 and variance 1.

```
rng default; % For reproducibility
r = normrnd(10,1,100,1);
```

Construct a histogram with a normal distribution fit.

```
histfit(r)
```



histfit uses fitdist to fit a distribution to data. Use fitdist to obtain parameters used in fitting.

```
pd = fitdist(r, 'Normal')
```

```
pd =
```

```
NormalDistribution
```

```
Normal distribution
mu = 10.1231    [9.89244, 10.3537]
sigma = 1.1624 [1.02059, 1.35033]
```

The intervals next to the parameter estimates are the 95% confidence intervals for the distribution parameters.

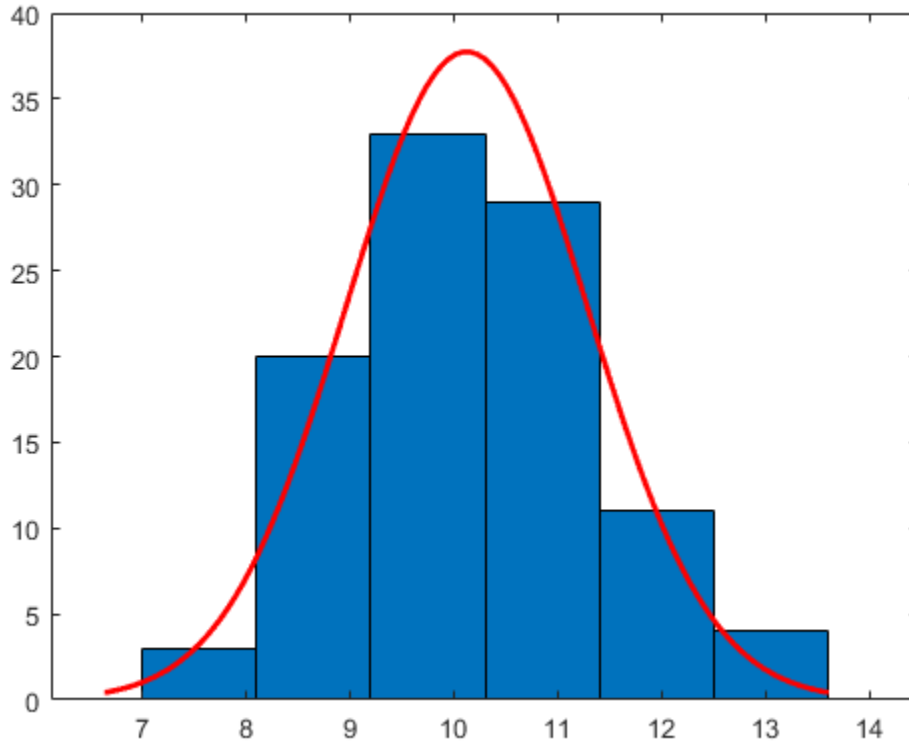
### **Histogram for a Given Number of Bins**

Generate a sample of size 100 from a normal distribution with mean 10 and variance 1.

```
rng default; % For reproducibility
r = normrnd(10,1,100,1);
```

Construct a histogram using six bins with a normal distribution fit.

```
histfit(r,6)
```



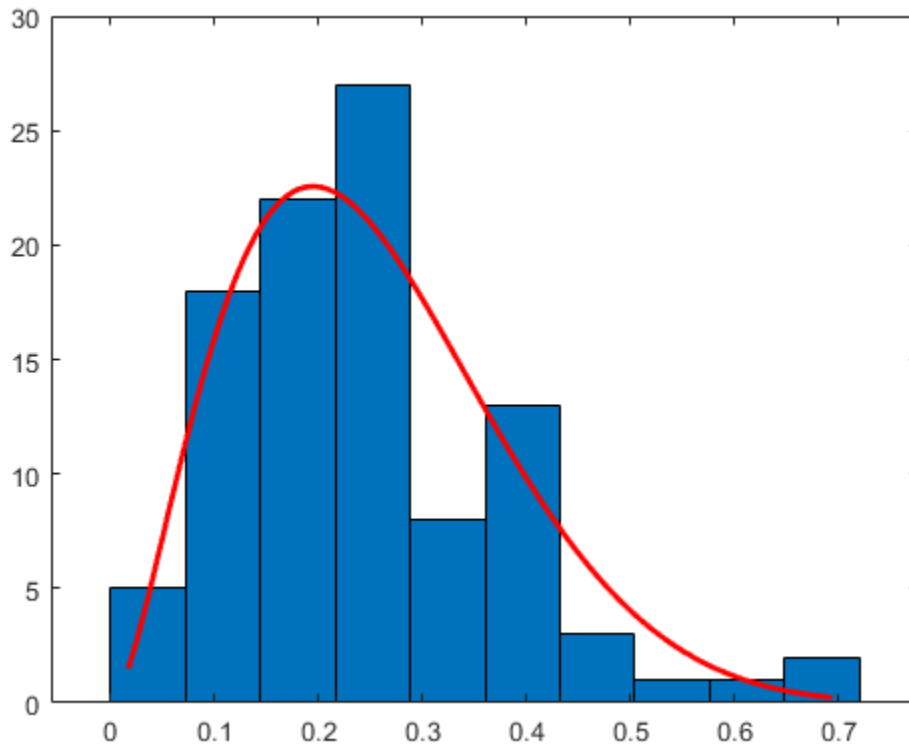
### Histogram with a Specified Distribution Fit

Generate a sample of size 100 from a beta distribution with parameters (3,10).

```
rng default; % For reproducibility  
b = betarnd(3,10,100,1);
```

Construct a histogram using 10 bins with a beta distribution fit.

```
histfit(b,10,'beta')
```



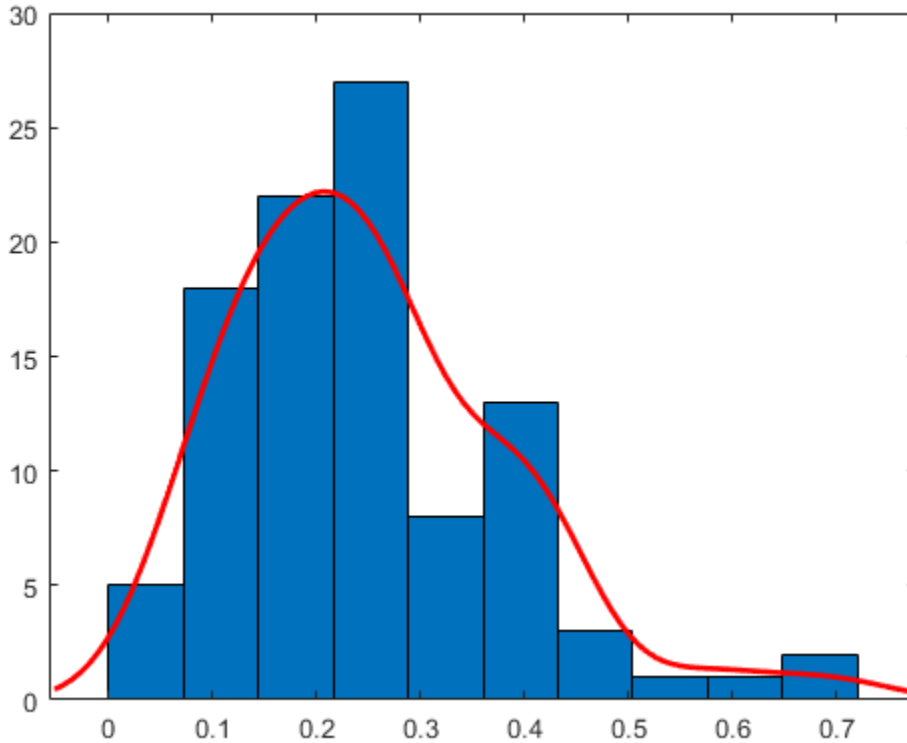
### Histogram with a Kernel Smoothing Function Fit

Generate a sample of size 100 from a beta distribution with parameters (3,10).

```
rng default; % For reproducibility  
b = betarnd(3,10,[100,1]);
```

Construct a histogram using 10 bins with a smoothing function fit.

```
histfit(b,10,'kernel')
```



### Handle for a Histogram with a Distribution Fit

Generate a sample of size 100 from a normal distribution with mean 10 and variance 1.

```
rng default % for reproducibility
r = normrnd(10,1,100,1);
```

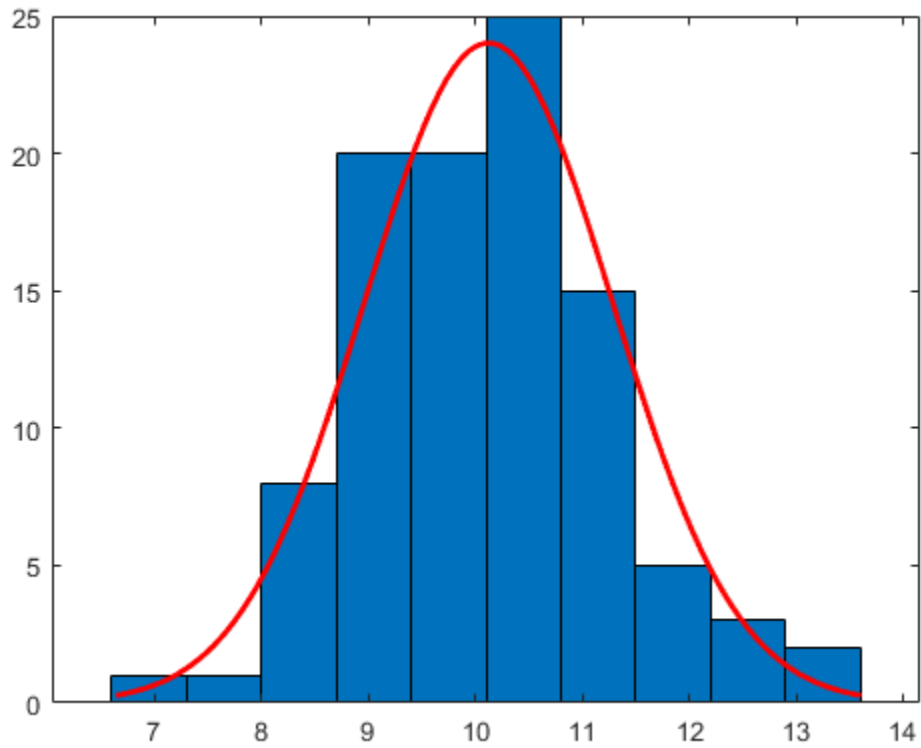
Construct a histogram with a normal distribution fit.

```
h = histfit(r,10,'normal')
```

```
h =
```

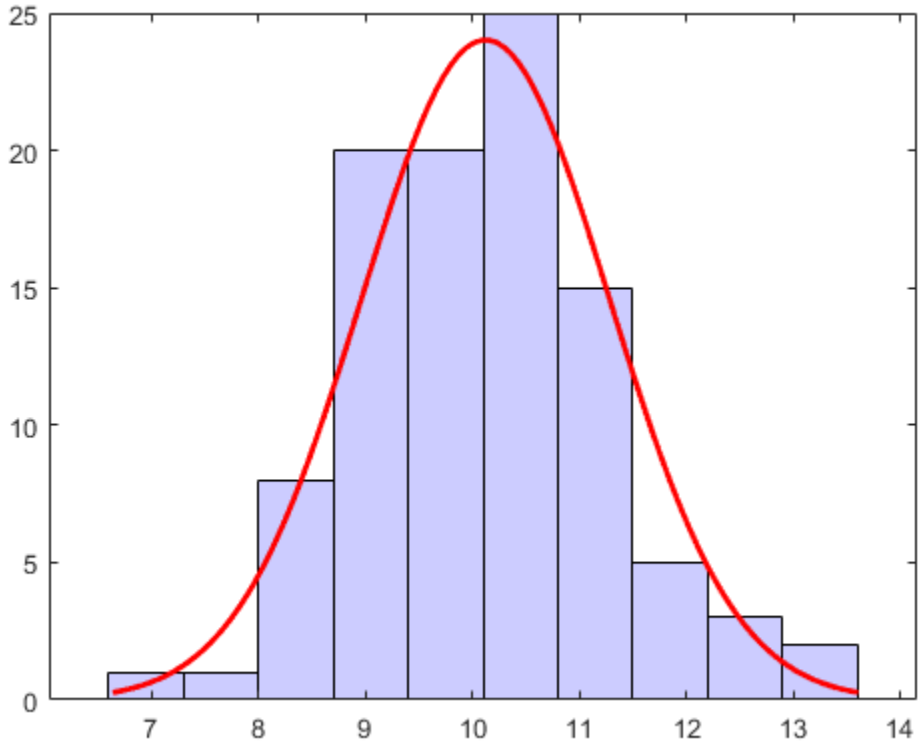
```
2x1 graphics array:
```

```
Bar  
Line
```



Change the bar colors of the histogram.

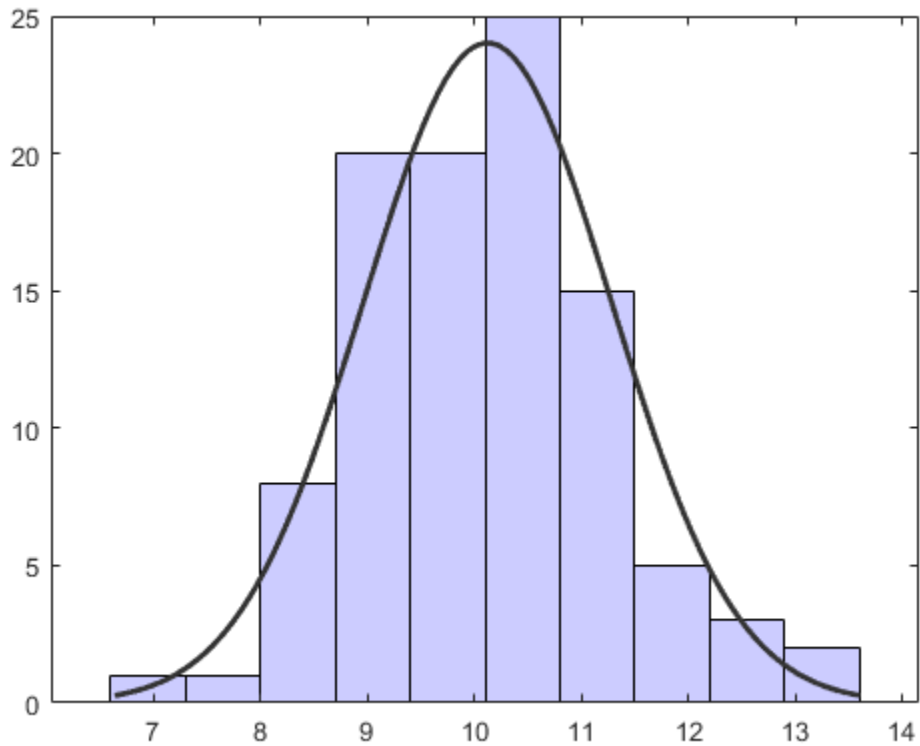
```
h(1).FaceColor = [.8 .8 1];
```



Change the color of the density curve.

```
h(2).Color = [.2 .2 .2];
```





## Input Arguments

**data** — Input data

vector

Input data, specified as a vector.

Example: `data = [1.5 2.5 4.6 1.2 3.4]`

Example: `data = [1.5 2.5 4.6 1.2 3.4]'`

Data Types: `double` | `single`

**nbins** — Number of bins

positive integer | []

Number of bins for the histogram, specified as a positive integer. Default value is the square root of the number of elements in `data`, rounded up. Use [] for the default number of bins when fitting a distribution.

Example: `y = histfit(x,8)`Example: `y = histfit(x,10,'gamma')`Example: `y = histfit(x,[],'weibull')`Data Types: `double` | `single`**dist** — Distribution to fit

'normal' (default) | character vector

Distribution to fit to the histogram, specified as a character vector. The following table shows the supported distributions.

<b>dist</b>	<b>Description</b>
'beta'	Beta
'birnbaumsaunders'	Birnbaum-Saunders
'burr'	Burr Type XII
'exponential'	Exponential
'extreme value' or 'ev'	Extreme value
'gamma'	Gamma
'generalized extreme value' or 'gev'	Generalized extreme value
'generalized pareto' or 'gp'	Generalized Pareto (threshold 0)
'inversegaussian'	Inverse Gaussian
'logistic'	Logistic
'loglogistic'	Loglogistic
'lognormal'	Lognormal
'nakagami'	Nakagami
'negative binomial' or 'nbin'	Negative binomial

<b>dist</b>	<b>Description</b>
'normal'	Normal
'poisson'	Poisson
'rayleigh'	Rayleigh
'rician'	Rician
'tlocationscale'	t location-scale
'weibull' or 'wbl'	Weibull
'kernel'	Nonparametric kernel-smoothing distribution. The density is evaluated at 100 equally spaced points that cover the range of the data in <code>data</code> . It works best with continuously distributed samples.

Data Types: char

## Output Arguments

**h** — Handles for the plot

plot handle

Handles for the plot, returned as a vector, where `h(1)` is the handle to the histogram, and `h(2)` is the handle to the density curve. `histfit` normalizes the density to match the total area under the curve with that of the histogram.

## Algorithms

`histfit` uses `fitdist` to fit a distribution to data. Use `fitdist` to obtain parameters used in fitting.

## See Also

`distributionFitter` | `fitdist` | `histogram` | `normfit` | `paramci`

Introduced before R2006a

## hmmdecode

Hidden Markov model posterior state probabilities

### Syntax

```
PSTATES = hmmdecode(seq, TRANS, EMIS)
[PSTATES, logpseq] = hmmdecode(...)
[PSTATES, logpseq, FORWARD, BACKWARD, S] = hmmdecode(...)
hmmdecode(..., 'Symbols', SYMBOLS)
```

### Description

`PSTATES = hmmdecode(seq, TRANS, EMIS)` calculates the posterior state probabilities, `PSTATES`, of the sequence `seq`, from a hidden Markov model. The posterior state probabilities are the conditional probabilities of being at state  $k$  at step  $i$ , given the observed sequence of symbols, `sym`. You specify the model by a transition probability matrix, `TRANS`, and an emissions probability matrix, `EMIS`. `TRANS(i, j)` is the probability of transition from state  $i$  to state  $j$ . `EMIS(k, seq)` is the probability that symbol `seq` is emitted from state  $k$ .

`PSTATES` is an array with the same length as `seq` and one row for each state in the model. The  $(i, j)$ th element of `PSTATES` gives the probability that the model is in state  $i$  at the  $j$ th step, given the sequence `seq`.

---

**Note** The function `hmmdecode` begins with the model in state 1 at step 0, prior to the first emission. `hmmdecode` computes the probabilities in `PSTATES` based on the fact that the model begins in state 1.

---

`[PSTATES, logpseq] = hmmdecode(...)` returns `logpseq`, the logarithm of the probability of sequence `seq`, given transition matrix `TRANS` and emission matrix `EMIS`.

`[PSTATES, logpseq, FORWARD, BACKWARD, S] = hmmdecode(...)` returns the forward and backward probabilities of the sequence scaled by `S`.

`hmmdecode(..., 'Symbols', SYMBOLS)` specifies the symbols that are emitted. `SYMBOLS` can be a numeric array or a cell array of the names of the symbols. The default symbols are integers 1 through `N`, where `N` is the number of possible emissions.

## Examples

```
trans = [0.95,0.05;  
         0.10,0.90];  
emis = [1/6 1/6 1/6 1/6 1/6 1/6;  
        1/10 1/10 1/10 1/10 1/10 1/2];  
  
[seq,states] = hmmgenerate(100,trans,emis);  
pStates = hmmdecode(seq,trans,emis);  
[seq,states] = hmmgenerate(100,trans,emis,...  
    'Symbols',{'one','two','three','four','five','six'})  
pStates = hmmdecode(seq,trans,emis,...  
    'Symbols',{'one','two','three','four','five','six'});
```

## References

- [1] Durbin, R., S. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis*. Cambridge, UK: Cambridge University Press, 1998.

## See Also

`hmmestimate` | `hmmgenerate` | `hmmtrain` | `hmmviterbi`

Introduced before R2006a

## hmmestimate

Hidden Markov model parameter estimates from emissions and states

### Syntax

```
[TRANS,EMIS] = hmmestimate(seq,states)
hmmestimate(...,'Symbols',SYMBOLS)
hmmestimate(...,'Statenames',STATENAMES)
hmmestimate(...,'Pseudoemissions',PSEUDOE)
hmmestimate(...,'Pseudotransitions',PSEUDOTR)
```

### Description

`[TRANS,EMIS] = hmmestimate(seq,states)` calculates the maximum likelihood estimate of the transition, TRANS, and emission, EMIS, probabilities of a hidden Markov model for sequence, seq, with known states, states.

`hmmestimate(...,'Symbols',SYMBOLS)` specifies the symbols that are emitted. SYMBOLS can be a numeric array or a cell array of the names of the symbols. The default symbols are integers 1 through N, where N is the number of possible emissions.

`hmmestimate(...,'Statenames',STATENAMES)` specifies the names of the states. STATENAMES can be a numeric array or a cell array of the names of the states. The default state names are 1 through M, where M is the number of states.

`hmmestimate(...,'Pseudoemissions',PSEUDOE)` specifies pseudocount emission values in the matrix PSEUDOE. Use this argument to avoid zero probability estimates for emissions with very low probability that might not be represented in the sample sequence. PSEUDOE should be a matrix of size  $m$ -by- $n$ , where  $m$  is the number of states in the hidden Markov model and  $n$  is the number of possible emissions. If the  $i \rightarrow k$  emission does not occur in seq, you can set PSEUDOE( $i$ ,  $k$ ) to be a positive number representing an estimate of the expected number of such emissions in the sequence seq.

`hmmestimate(...,'Pseudotransitions',PSEUDOTR)` specifies pseudocount transition values. You can use this argument to avoid zero probability estimates for

transitions with very low probability that might not be represented in the sample sequence. PSEUDOTR should be a matrix of size  $m$ -by- $m$ , where  $m$  is the number of states in the hidden Markov model. If the  $i \rightarrow j$  transition does not occur in `states`, you can set PSEUDOTR( $i, j$ ) to be a positive number representing an estimate of the expected number of such transitions in the sequence `states`.

## Pseudotransitions and Pseudoemissions

If the probability of a specific transition or emission is very low, the transition might never occur in the sequence `states`, or the emission might never occur in the sequence `seq`. In either case, the algorithm returns a probability of 0 for the given transition or emission in TRANS or EMIS. You can compensate for the absence of transition with the 'Pseudotransitions' and 'Pseudoemissions' arguments. The simplest way to do this is to set the corresponding entry of PSEUDOE or PSEUDOTR to 1. For example, if the transition  $i \rightarrow j$  does not occur in `states`, set PSEUDOTR( $i, j$ ) = 1. This forces TRANS( $i, j$ ) to be positive. If you have an estimate for the expected number of transitions  $i \rightarrow j$  in a sequence of the same length as `states`, and the actual number of transitions  $i \rightarrow j$  that occur in `seq` is substantially less than what you expect, you can set PSEUDOTR( $i, j$ ) to the expected number. This increases the value of TRANS( $i, j$ ). For transitions that do occur in `states` with the frequency you expect, set the corresponding entry of PSEUDOTR to 0, which does not increase the corresponding entry of TRANS.

If you do not know the sequence of states, use `hmmtrain` to estimate the model parameters.

## Examples

```
trans = [0.95,0.05; 0.10,0.90];
emis = [1/6 1/6 1/6 1/6 1/6 1/6;
        1/10 1/10 1/10 1/10 1/10 1/2];

[seq,states] = hmmgenerate(1000,trans,emis);
[estimateTR,estimateE] = hmmestimate(seq,states);
```

## References

- [1] Durbin, R., S. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis*. Cambridge, UK: Cambridge University Press, 1998.

## See Also

`hmmdecode` | `hmmgenerate` | `hmmtrain` | `hmmviterbi`

**Introduced before R2006a**



# hmmgenerate

Hidden Markov model states and emissions

## Syntax

```
[seq, states] = hmmgenerate(len, TRANS, EMIS)
hmmgenerate(..., 'Symbols', SYMBOLS)
hmmgenerate(..., 'Statenames', STATENAMES)
```

## Description

`[seq, states] = hmmgenerate(len, TRANS, EMIS)` takes a known Markov model, specified by transition probability matrix `TRANS` and emission probability matrix `EMIS`, and uses it to generate

- A random sequence `seq` of emission symbols
- A random sequence `states` of states

The length of both `seq` and `states` is `len`. `TRANS(i, j)` is the probability of transition from state `i` to state `j`. `EMIS(k, l)` is the probability that symbol `l` is emitted from state `k`.

---

**Note** The function `hmmgenerate` begins with the model in state 1 at step 0, prior to the first emission. The model then makes a transition to state  $i_1$ , with probability  $T_{1i_1}$ , and generates an emission  $a_{k_1}$  with probability  $E_{i_1, k_1}$ . `hmmgenerate` returns  $i_1$  as the first entry of `states`, and  $a_{k_1}$  as the first entry of `seq`.

---

`hmmgenerate(..., 'Symbols', SYMBOLS)` specifies the symbols that are emitted. `SYMBOLS` can be a numeric array or a cell array of the names of the symbols. The default symbols are integers 1 through `N`, where `N` is the number of possible emissions.

`hmmgenerate(..., 'Statenames', STATENAMES)` specifies the names of the states. `STATENAMES` can be a numeric array or a cell array of the names of the states. The default state names are 1 through `M`, where `M` is the number of states.

Since the model always begins at state 1, whose transition probabilities are in the first row of `TRANS`, in the following example, the first entry of the output states is be 1 with probability 0.95 and 2 with probability 0.05.

## Examples

```
trans = [0.95,0.05;
         0.10,0.90];
emis = [1/6 1/6 1/6 1/6 1/6 1/6;
       1/10 1/10 1/10 1/10 1/10 1/2];

[seq,states] = hmmgenerate(100,trans,emis)
[seq,states] = hmmgenerate(100,trans,emis,...
    'Symbols',{'one','two','three','four','five','six'},...
    'Statenames',{'fair';'loaded'})
```

## See Also

`hmmdecode` | `hmmestimate` | `hmmtrain` | `hmmviterbi`

Introduced before R2006a

# hmmtrain

Hidden Markov model parameter estimates from emissions

## Syntax

```
[ESTTR,ESTEMIT] = hmmtrain(seq,TRGUESS,EMITGUESS)
hmmtrain(...,'Algorithm',algorithm)
hmmtrain(...,'Symbols',SYMBOLS)
hmmtrain(...,'Tolerance',tol)
hmmtrain(...,'Maxiterations',maxiter)
hmmtrain(...,'Verbose',true)
hmmtrain(...,'Pseudoemissions',PSEUDOE)
hmmtrain(...,'Pseudotransitions',PSEUDOTR)
```

## Description

`[ESTTR,ESTEMIT] = hmmtrain(seq,TRGUESS,EMITGUESS)` estimates the transition and emission probabilities for a hidden Markov model using the Baum-Welch algorithm. `seq` can be a row vector containing a single sequence, a matrix with one row per sequence, or a cell array with each cell containing a sequence. `TRGUESS` and `EMITGUESS` are initial estimates of the transition and emission probability matrices. `TRGUESS(i,j)` is the estimated probability of transition from state `i` to state `j`. `EMITGUESS(i,k)` is the estimated probability that symbol `k` is emitted from state `i`.

`hmmtrain(...,'Algorithm',algorithm)` specifies the training algorithm. `algorithm` can be either 'BaumWelch' or 'Viterbi'. The default algorithm is 'BaumWelch'.

`hmmtrain(...,'Symbols',SYMBOLS)` specifies the symbols that are emitted. `SYMBOLS` can be a numeric array or a cell array of the names of the symbols. The default symbols are integers 1 through `N`, where `N` is the number of possible emissions.

`hmmtrain(...,'Tolerance',tol)` specifies the tolerance used for testing convergence of the iterative estimation process. The default tolerance is  $1e-4$ .

`hmmtrain(..., 'Maxiterations', maxiter)` specifies the maximum number of iterations for the estimation process. The default maximum is 100.

`hmmtrain(..., 'Verbose', true)` returns the status of the algorithm at each iteration.

`hmmtrain(..., 'Pseudoemissions', PSEUDO_E)` specifies pseudocount emission values for the Viterbi training algorithm. Use this argument to avoid zero probability estimates for emissions with very low probability that might not be represented in the sample sequence. PSEUDO\_E should be a matrix of size  $m$ -by- $n$ , where  $m$  is the number of states in the hidden Markov model and  $n$  is the number of possible emissions. If the  $i \rightarrow k$  emission does not occur in `seq`, you can set `PSEUDO_E(i, k)` to be a positive number representing an estimate of the expected number of such emissions in the sequence `seq`.

`hmmtrain(..., 'Pseudotransitions', PSEUDO_T)` specifies pseudocount transition values for the Viterbi training algorithm. Use this argument to avoid zero probability estimates for transitions with very low probability that might not be represented in the sample sequence. PSEUDO\_T should be a matrix of size  $m$ -by- $m$ , where  $m$  is the number of states in the hidden Markov model. If the  $i \rightarrow j$  transition does not occur in `states`, you can set `PSEUDO_T(i, j)` to be a positive number representing an estimate of the expected number of such transitions in the sequence `states`.

If you know the states corresponding to the sequences, use `hmmestimate` to estimate the model parameters.

## Tolerance

The input argument `'tolerance'` controls how many steps the `hmmtrain` algorithm executes before the function returns an answer. The algorithm terminates when all of the following three quantities are less than the value that you specify for `tolerance`:

- The log likelihood that the input sequence `seq` is generated by the currently estimated values of the transition and emission matrices
- The change in the norm of the transition matrix, normalized by the size of the matrix
- The change in the norm of the emission matrix, normalized by the size of the matrix

The default value of `'tolerance'` is  $1e-6$ . Increasing the tolerance decreases the number of steps the `hmmtrain` algorithm executes before it terminates.

## maxiterations

The maximum number of iterations, 'maxiterations', controls the maximum number of steps the algorithm executes before it terminates. If the algorithm executes `maxiterations` before reaching the specified tolerance, the algorithm terminates and the function returns a warning. If this occurs, you can increase the value of 'maxiterations' to make the algorithm reach the desired tolerance before terminating.

## Examples

```
trans = [0.95,0.05;
         0.10,0.90];
emis = [1/6, 1/6, 1/6, 1/6, 1/6, 1/6;
        1/10, 1/10, 1/10, 1/10, 1/10, 1/2];

seq1 = hmmgenerate(100,trans,emis);
seq2 = hmmgenerate(200,trans,emis);
seqs = {seq1,seq2};
[estTR,estE] = hmmtrain(seqs,trans,emis);
```

## References

- [1] Durbin, R., S. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis*. Cambridge, UK: Cambridge University Press, 1998.

## See Also

`hmmdecode` | `hmmestimate` | `hmmgenerate` | `hmmviterbi`

Introduced before R2006a

## hmmviterbi

Hidden Markov model most probable state path

### Syntax

```
STATES = hmmviterbi(seq,TRANS,EMIS)
hmmviterbi(...,'Symbols',SYMBOLS)
hmmviterbi(...,'Statenames',STATENAMES)
```

### Description

`STATES = hmmviterbi(seq,TRANS,EMIS)` given a sequence, `seq`, calculates the most likely path through the hidden Markov model specified by transition probability matrix, `TRANS`, and emission probability matrix `EMIS`. `TRANS(i,j)` is the probability of transition from state `i` to state `j`. `EMIS(i,k)` is the probability that symbol `k` is emitted from state `i`.

---

**Note** The function `hmmviterbi` begins with the model in state 1 at step 0, prior to the first emission. `hmmviterbi` computes the most likely path based on the fact that the model begins in state 1.

---

`hmmviterbi(...,'Symbols',SYMBOLS)` specifies the symbols that are emitted. `SYMBOLS` can be a numeric array or a cell array of the names of the symbols. The default symbols are integers 1 through `N`, where `N` is the number of possible emissions.

`hmmviterbi(...,'Statenames',STATENAMES)` specifies the names of the states. `STATENAMES` can be a numeric array or a cell array of the names of the states. The default state names are 1 through `M`, where `M` is the number of states.

### Examples

```
trans = [0.95,0.05;
         0.10,0.90];
```

```
emis = [1/6 1/6 1/6 1/6 1/6 1/6;  
        1/10 1/10 1/10 1/10 1/10 1/2];  
  
[seq,states] = hmmgenerate(100,trans,emis);  
estimatedStates = hmmviterbi(seq,trans,emis);  
  
[seq,states] = ...  
    hmmgenerate(100,trans,emis,...  
                'Statenames',{'fair';'loaded'});  
estimatedStates = ...  
    hmmviterbi(seq,trans,emis,...  
                'Statenames',{'fair';'loaded'});
```

## References

- [1] Durbin, R., S. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis*. Cambridge, UK: Cambridge University Press, 1998.

## See Also

[hmmdecode](#) | [hmmestimate](#) | [hmmgenerate](#) | [hmmtrain](#)

**Introduced before R2006a**

## horzcat

**Class:** dataset

Horizontal concatenation for dataset arrays

---

**Note** The `dataset` data type might be removed in a future release. To work with heterogeneous data, use the MATLAB® `table` data type instead. See MATLAB `table` documentation for more information.

---

## Syntax

```
ds = horzcat(ds1, ds2, ...)
```

## Description

`ds = horzcat(ds1, ds2, ...)` horizontally concatenates the dataset arrays `ds1`, `ds2`, ... . You may concatenate dataset arrays that have duplicate variable names, however, the variables must contain identical data, and `horzcat` includes only one copy of the variable in the output dataset.

Observation names for all dataset arrays that have them must be identical except for order. `horzcat` concatenates by matching observation names when present, or by position for datasets that do not have observation names.

## See Also

`cat` | `vertcat`



# hougen

Hougen-Watson model

## Syntax

```
yhat = hougen(beta, x)
```

## Description

`yhat = hougen(beta, x)` returns the predicted values of the reaction rate, `yhat`, as a function of the vector of parameters, `beta`, and the matrix of data, `x`. `beta` must have 5 elements and `x` must have three columns.

`hougen` is a utility function for `rsmdemo`.

The model form is:

$$\hat{y} = \frac{\beta_1 x_2 - x_3 / \beta_5}{1 + \beta_2 x_1 + \beta_3 x_2 + \beta_4 x_3}$$

## References

- [1] Bates, D. M., and D. G. Watts. *Nonlinear Regression Analysis and Its Applications*. Hoboken, NJ: John Wiley & Sons, Inc., 1988.

## See Also

`rsmdemo`

Introduced before R2006a

## hygecdf

Hypergeometric cumulative distribution function

### Syntax

```
hygecdf(x, M, K, N)  
hygecdf(x, M, K, N, 'upper')
```

### Description

`hygecdf(x, M, K, N)` computes the hypergeometric cdf at each of the values in `x` using the corresponding size of the population, `M`, number of items with the desired characteristic in the population, `K`, and number of samples drawn, `N`. Vector or matrix inputs for `x`, `M`, `K`, and `N` must all have the same size. A scalar input is expanded to a constant matrix with the same dimensions as the other inputs.

`hygecdf(x, M, K, N, 'upper')` returns the complement of the hypergeometric cdf at each value in `x`, using an algorithm that more accurately computes the extreme upper tail probabilities.

The hypergeometric cdf is

$$p = F(x | M, K, N) = \sum_{i=0}^x \frac{\binom{K}{i} \binom{M-K}{N-i}}{\binom{M}{N}}$$

The result,  $p$ , is the probability of drawing up to  $x$  of a possible  $K$  items in  $N$  drawings without replacement from a group of  $M$  objects.

### Examples

## Compute Hypergeometric Distribution CDF

Suppose you have a lot of 100 floppy disks and you know that 20 of them are defective. What is the probability of drawing zero to two defective floppies if you select 10 at random?

```
p = hygecdf(2,100,20,10)
```

```
p =
```

```
0.6812
```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

[cdf](#) | [hygeinv](#) | [hygepdf](#) | [hygernd](#) | [hygestat](#)

Introduced before R2006a

## hygeinv

Hypergeometric inverse cumulative distribution function

### Syntax

```
hygeinv(P, M, K, N)
```

### Description

`hygeinv(P, M, K, N)` returns the smallest integer  $X$  such that the hypergeometric cdf evaluated at  $X$  equals or exceeds  $P$ . You can think of  $P$  as the probability of observing  $X$  defective items in  $N$  drawings without replacement from a group of  $M$  items where  $K$  are defective.

### Examples

Suppose you are the Quality Assurance manager for a floppy disk manufacturer. The production line turns out floppy disks in batches of 1,000. You want to sample 50 disks from each batch to see if they have defects. You want to accept 99% of the batches if there are no more than 10 defective disks in the batch. What is the maximum number of defective disks should you allow in your sample of 50?

```
x = hygeinv(0.99, 1000, 10, 50)
x =
    3
```

What is the median number of defective floppy disks in samples of 50 disks from batches with 10 defective disks?

```
x = hygeinv(0.50, 1000, 10, 50)
x =
    0
```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`hygecdf` | `hygepdf` | `hygernd` | `hygestat` | `icdf`

Introduced before R2006a

## hygepdf

Hypergeometric probability density function

### Syntax

`Y = hygepdf(X, M, K, N)`

### Description

`Y = hygepdf(X, M, K, N)` computes the hypergeometric pdf at each of the values in `X` using the corresponding size of the population, `M`, number of items with the desired characteristic in the population, `K`, and number of samples drawn, `N`. `X`, `M`, `K`, and `N` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array with the same dimensions as the other inputs.

The parameters in `M`, `K`, and `N` must all be positive integers, with  $N \leq M$ . The values in `X` must be less than or equal to all the parameter values.

The hypergeometric pdf is

$$y = f(x | M, K, N) = \frac{\binom{K}{x} \binom{M-K}{N-x}}{\binom{M}{N}}$$

The result, `y`, is the probability of drawing exactly `x` of a possible `K` items in `n` drawings without replacement from a group of `M` objects.

### Examples

Suppose you have a lot of 100 floppy disks and you know that 20 of them are defective. What is the probability of drawing 0 through 5 defective floppy disks if you select 10 at random?

```
p = hygepdf(0:5,100,20,10)
p =
    0.0951    0.2679    0.3182    0.2092    0.0841    0.0215
```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

[hygecdf](#) | [hygeinv](#) | [hygernd](#) | [hygestat](#) | [pdf](#)

**Introduced before R2006a**

## hygernd

Hypergeometric random numbers

### Syntax

```
R = hygernd(M, K, N)
R = hygernd(M, K, N, m, n, ...)
R = hygernd(M, K, N, [m, n, ...])
```

### Description

`R = hygernd(M, K, N)` generates random numbers from the hypergeometric distribution with corresponding size of the population, `M`, number of items with the desired characteristic in the population, `K`, and number of samples drawn, `N`. `M`, `K`, and `N` can be vectors, matrices, or multidimensional arrays that all have the same size, which is also the size of `R`. A scalar input for `M`, `K`, or `N` is expanded to a constant array with the same dimensions as the other inputs.

`R = hygernd(M, K, N, m, n, ...)` or `R = hygernd(M, K, N, [m, n, ...])` generates an `m`-by-`n`-by-... array. The `M`, `K`, `N` parameters can each be scalars or arrays of the same size as `R`.

### Examples

```
numbers = hygernd(1000, 40, 50)
numbers =
    1
```



## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

The generated code can return a different sequence of numbers than MATLAB if either of the following is true:

- The output is nonscalar.
- An input parameter is invalid for the distribution.

### See Also

`hygecdf` | `hygeinv` | `hygepdf` | `hygestat` | `random`

**Introduced before R2006a**

## hygestat

Hypergeometric mean and variance

### Syntax

```
[MN,V] = hygestat(M,K,N)
```

### Description

`[MN,V] = hygestat(M,K,N)` returns the mean and variance for the hypergeometric distribution with corresponding size of the population,  $M$ , number of items with the desired characteristic in the population,  $K$ , and number of samples drawn,  $N$ . Vector or matrix inputs for  $M$ ,  $K$ , and  $N$  must have the same size, which is also the size of  $MN$  and  $V$ . A scalar input for  $M$ ,  $K$ , or  $N$  is expanded to a constant matrix with the same dimensions as the other inputs.

The mean of the hypergeometric distribution with parameters  $M$ ,  $K$ , and  $N$  is  $NK/M$ , and the variance is  $NK(M-K)(M-N) / [M^2(M-1)]$ .

### Examples

The hypergeometric distribution approaches the binomial distribution, where  $p = K/M$ , as  $M$  goes to infinity.

```
[m,v] = hygestat(10.^(1:4),10.^(0:3),9)
m =
    0.9000    0.9000    0.9000    0.9000
v =
    0.0900    0.7445    0.8035    0.8094
```

```
[m,v] = binostat(9,0.1)
m =
    0.9000
v =
    0.8100
```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`hygecdf` | `hygeinv` | `hygepdf` | `hygernd`

Introduced before R2006a

## hyperparameters

Variable descriptions for optimizing a fit function

### Syntax

```
VariableDescriptions = hyperparameters(FitFcnName, predictors,  
response)  
VariableDescriptions = hyperparameters(FitFcnName, predictors,  
response, LearnerType)
```

### Description

`VariableDescriptions = hyperparameters(FitFcnName, predictors, response)` returns the default variables for the given fit function. These are the variables that apply when you set the `OptimizeHyperparameters` name-value pair to `'auto'`.

`VariableDescriptions = hyperparameters(FitFcnName, predictors, response, LearnerType)` returns the variables for an ensemble fit with specified learner type. This syntax applies when `FitFcnName` is `'fitcecoc'`, `'fitcensemble'`, or `'fitrensemble'`.

### Examples

#### Obtain Default Hyperparameters

Obtain the default hyperparameters for the `fitcsvm` classifier.

Load the ionosphere data.

```
load ionosphere
```

Obtain the hyperparameters.

```
VariableDescriptions = hyperparameters('fitcsvm',X,Y);
```

**Examine all the hyperparameters.**

```
for ii = 1:length(VariableDescriptions)
    disp(ii), disp(VariableDescriptions(ii))
end
```

1

optimizableVariable with properties:

```
    Name: 'BoxConstraint'
    Range: [1.0000e-03 1000]
    Type: 'real'
    Transform: 'log'
    Optimize: 1
```

2

optimizableVariable with properties:

```
    Name: 'KernelScale'
    Range: [1.0000e-03 1000]
    Type: 'real'
    Transform: 'log'
    Optimize: 1
```

3

optimizableVariable with properties:

```
    Name: 'KernelFunction'
    Range: {'gaussian' 'linear' 'polynomial'}
    Type: 'categorical'
    Transform: 'none'
    Optimize: 0
```

4

optimizableVariable with properties:

```
    Name: 'PolynomialOrder'
    Range: [2 4]
    Type: 'integer'
```

```
Transform: 'none'  
Optimize: 0
```

```
5
```

```
optimizableVariable with properties:
```

```
    Name: 'Standardize'  
    Range: {'true' 'false'}  
    Type: 'categorical'  
Transform: 'none'  
Optimize: 0
```

Change the `PolynomialOrder` hyperparameter to have a wider range and to be used in an optimization.

```
VariableDescriptions(4).Range = [2,5];  
VariableDescriptions(4).Optimize = true;  
disp(VariableDescriptions(4))
```

```
optimizableVariable with properties:
```

```
    Name: 'PolynomialOrder'  
    Range: [2 5]  
    Type: 'integer'  
Transform: 'none'  
Optimize: 1
```

### Obtain Ensemble Hyperparameters

Obtain the default hyperparameters for the `fitensemble` ensemble regression function.

Load the `carsmall` data.

```
load carsmall
```

Use `Horsepower` and `Weight` as predictor variables, and `MPG` as the response variable.

```
X = [Horsepower Weight];  
Y = MPG;
```

Obtain the default hyperparameters for a Tree learner.

```
VariableDescriptions = hyperparameters('fitrensemble',X,Y,'Tree');
```

Examine all the hyperparameters.

```
for ii = 1:length(VariableDescriptions)
    disp(ii),disp(VariableDescriptions(ii))
end
```

1

optimizableVariable with properties:

```
    Name: 'Method'
    Range: {'Bag' 'LSBoost'}
    Type: 'categorical'
    Transform: 'none'
    Optimize: 1
```

2

optimizableVariable with properties:

```
    Name: 'NumLearningCycles'
    Range: [10 500]
    Type: 'integer'
    Transform: 'log'
    Optimize: 1
```

3

optimizableVariable with properties:

```
    Name: 'LearnRate'
    Range: [1.0000e-03 1]
    Type: 'real'
    Transform: 'log'
    Optimize: 1
```

4

optimizableVariable with properties:

```
    Name: 'MinLeafSize'
```

```
    Range: [1 50]
    Type: 'integer'
Transform: 'log'
Optimize: 1
```

5

optimizableVariable with properties:

```
    Name: 'MaxNumSplits'
    Range: [1 99]
    Type: 'integer'
Transform: 'log'
Optimize: 0
```

6

optimizableVariable with properties:

```
    Name: 'NumVariablesToSample'
    Range: [1 2]
    Type: 'integer'
Transform: 'none'
Optimize: 0
```

**Change the MaxNumSplits hyperparameter to have a wider range and to be used in an optimization.**

```
VariableDescriptions(5).Range = [1,200];
VariableDescriptions(5).Optimize = true;
disp(VariableDescriptions(5))
```

optimizableVariable with properties:

```
    Name: 'MaxNumSplits'
    Range: [1 200]
    Type: 'integer'
Transform: 'log'
```



Optimize: 1

## Input Arguments

### **FitFcnName** — Name of fitting function

'fitcdiscr' | 'fitcecoc' | 'fitcknn' | 'fitcensemble' | 'fitclinear' |  
'fitcnb' | 'fitcsvm' | 'fitctree' | 'fitrensemble' | 'fitrgp' |  
'fitrlinear' | 'fitrsvm' | 'fitrtree'

Name of fitting function, specified as one of the listed classification or regression fit function names.

If `FitFcnName` is 'fitcecoc', 'fitcensemble', or 'fitrensemble', then also specify the learner type in the `LearnerType` argument.

Example: 'fitctree'

Data Types: char

### **predictors** — Predictor data

matrix with D predictor columns | table with D predictor columns

Predictor data, specified as a matrix with D predictor columns or as a table with D predictor columns, where D is the number of predictors.

Example: X

Data Types: double | logical | char | table | cell | categorical | datetime

### **response** — Class labels or numeric response

grouping variable | scalar

Class labels or numeric response, specified as a grouping variable (see “Grouping Variables” on page 2-59) or as a scalar.

Example: Y

Data Types: single | double | logical | char | cell

### **LearnerType** — Learner type for ensemble fit

'Discriminant' | 'KNN' | 'Linear' | 'SVM' | 'Tree' | template of a listed learner

Learner type for ensemble fit, specified as 'Discriminant', 'KNN', 'Linear', 'SVM', 'Tree', or a template of a listed learner. Use this argument when `FitFcnName` is 'fitcecoc', 'fitcensemble', or 'fitrensemble'.

For 'fitcensemble' you can use only 'Discriminant', 'KNN', 'Tree', or an associated template.

For 'fitrensemble', you can use only 'Tree' or `templateTree`.

Example: 'Tree'

Data Types: char

## Output Arguments

### **VariableDescriptions** — Variable descriptions

vector of `optimizableVariable` objects

Variable descriptions, returned as a vector of `optimizableVariable` objects. The variables have their default parameters set, such as range and variable type. All eligible variables exist in the descriptions, but the variables unused in the 'auto' setting have their `Optimize` property set to `false`. You can update the variables by using dot notation, as shown in “Examples” on page 32-0 .

## See Also

`fitcdiscr` | `fitcecoc` | `fitcensemble` | `fitcknn` | `fitcllinear` | `fitcnb` | `fitcsvm` | `fitctree` | `fitrensemble` | `fitrgp` | `fitrlinear` | `fitrsvm` | `fitrtree`

Introduced in R2016b

# icdf

Inverse cumulative distribution functions

## Syntax

```
x = icdf('name', y, A)
x = icdf('name', y, A, B)
x = icdf('name', y, A, B, C)
x = icdf('name', y, A, B, C, D)

x = icdf(pd, y)
```

## Description

`x = icdf('name', y, A)` returns the inverse cumulative distribution function (icdf) for the one-parameter distribution family specified by 'name', evaluated at the probability values in `y`. `A` contains the parameter value for the distribution.

`x = icdf('name', y, A, B)` returns the icdf for the two-parameter distribution family specified by 'name', evaluated at the probability values in `y`. `A` and `B` contain the parameter values for the distribution.

`x = icdf('name', y, A, B, C)` returns the icdf for the three-parameter distribution family specified by 'name', evaluated at the probability values in `y`. `A`, `B`, and `C` contain the parameter values for the distribution.

`x = icdf('name', y, A, B, C, D)` returns the icdf for the four-parameter distribution family specified by 'name', evaluated at the probability values in `y`. `A`, `B`, `C`, and `D` contain the parameter values for the distribution.

`x = icdf(pd, y)` returns the inverse cumulative distribution function of the probability distribution object, `pd`, evaluated at the probability values in `y`.

## Examples

### Compute the Normal Distribution icdf

Create a standard normal distribution object with the mean,  $\mu$ , equal to 0 and the standard deviation,  $\sigma$ , equal to 1.

```
mu = 0;  
sigma = 1;  
pd = makedist('Normal',mu,sigma);
```

Define the input vector  $y$  to contain the probability values at which to calculate the icdf.

```
y = [0.1,0.25,0.5,0.75,0.9];
```

Compute the icdf values for the standard normal distribution at the values in  $y$ .

```
x = icdf(pd,y)
```

```
x =
```

```
-1.2816   -0.6745         0    0.6745    1.2816
```

Each value in  $x$  corresponds to a value in the input vector  $y$ . For example, at the value  $y$  equal to 0.9, the corresponding icdf value  $x$  is equal to 1.2816.

Alternatively, you can compute the same icdf values without creating a probability distribution object. Use the `icdf` function and specify a standard normal distribution using the same parameter values for  $\mu$  and  $\sigma$ .

```
x2 = icdf('Normal',y,mu,sigma)
```

```
x2 =
```

```
-1.2816   -0.6745         0    0.6745    1.2816
```

The icdf values are the same as those computed using the probability distribution object.

## Compute the Poisson Distribution icdf

Create a Poisson distribution object with the rate parameter,  $\lambda$ , equal to 2.

```
lambda = 2;
pd = makedist('Poisson',lambda);
```

Define the input vector  $y$  to contain the probability values at which to calculate the icdf.

```
y = [0.1,0.25,0.5,0.75,0.9];
```

Compute the icdf values for the Poisson distribution at the values in  $y$ .

```
x = icdf(pd,y)
```

```
x =
```

```
    0    1    2    3    4
```

Each value in  $x$  corresponds to a value in the input vector  $y$ . For example, at the value  $y$  equal to 0.9, the corresponding icdf value  $x$  is equal to 4.

Alternatively, you can compute the same icdf values without creating a probability distribution object. Use the `icdf` function and specify a Poisson distribution using the same value for the rate parameter  $\lambda$ .

```
x2 = icdf('Poisson',y,lambda)
```

```
x2 =
```

```
    0    1    2    3    4
```

The icdf values are the same as those computed using the probability distribution object.

## Input Arguments

**'name'** — Probability distribution name

valid probability distribution name

Probability distribution name, specified as one of the following.

name	Distribution	Input Parameter A	Input Parameter B	Input Parameter C	Input Parameter D
'Beta '	“Beta Distribution” on page B-4	$\alpha$ : first shape parameter	$b$ : second shape parameter	—	—
'Binomial '	“Binomial Distribution” on page B-9	$n$ : number of trials	$p$ : probability of success for each trial	—	—
'BirnbaumSaunders '	“Birnbaum-Saunders Distribution” on page B-13	$\beta$ : scale parameter	$\gamma$ : shape parameter	—	—
'Burr '	“Burr Type XII Distribution” on page B-15	$\alpha$ : scale parameter	$c$ : first shape parameter	$k$ : second shape parameter	—
'Chisquare '	“Chi-Square Distribution” on page B-28	$\nu$ : degrees of freedom	—	—	—
'Exponential '	“Exponential Distribution” on page B-34	$\mu$ : mean	—	—	—
'Extreme Value '	“Extreme Value Distribution” on page B-38	$\mu$ : location parameter	$\sigma$ : scale parameter	—	—
'F '	“F Distribution” on page B-44	$\nu_1$ : numerator degrees of freedom	$\nu_2$ : denominator degrees of freedom	—	—
'Gamma '	“Gamma Distribution” on page B-47	$\alpha$ : shape parameter	$b$ : scale parameter	—	—
'Generalized Extreme Value '	“Generalized Extreme Value Distribution” on page B-53	$k$ : shape parameter	$\sigma$ : scale parameter	$\mu$ : location parameter	—

name	Distribution	Input Parameter A	Input Parameter B	Input Parameter C	Input Parameter D
'Generalized Pareto'	“Generalized Pareto Distribution” on page B-58	$k$ : tail index (shape) parameter	$\sigma$ : scale parameter	$\mu$ : threshold (location) parameter	—
'Geometric'	“Geometric Distribution” on page B-64	$p$ : probability parameter	—	—	—
'HalfNormal'	“Half-Normal Distribution” on page B-73	$\mu$ : location parameter	$\sigma$ : scale parameter	—	—
'Hypergeometric'	“Hypergeometric Distribution” on page B-79	$m$ : size of the population	$k$ : number of items with the desired characteristic in the population	$n$ : number of samples drawn	—
'InverseGaussian'	“Inverse Gaussian Distribution” on page B-82	$\mu$ : scale parameter	$\lambda$ : shape parameter	—	—
'Logistic'	“Logistic Distribution” on page B-96	$\mu$ : mean	$\sigma$ : scale parameter	—	—
'LogLogistic'	“Loglogistic Distribution” on page B-98	$\mu$ : log mean	$\sigma$ : log scale parameter	—	—
'Lognormal'	“Lognormal Distribution” on page B-100	$\mu$ : log mean	$\sigma$ : log standard deviation	—	—
'Nakagami'	“Nakagami Distribution” on page B-118	$\mu$ : shape parameter	$\omega$ : scale parameter	—	—
'Negative Binomial'	“Negative Binomial Distribution” on page B-120	$r$ : number of successes	$p$ : probability of success in a single trial	—	—

name	Distribution	Input Parameter A	Input Parameter B	Input Parameter C	Input Parameter D
'Noncentral F'	“Noncentral F Distribution” on page B-128	$\nu_1$ : numerator degrees of freedom	$\nu_2$ : denominator degrees of freedom	$\delta$ : noncentrality parameter	—
'Noncentral t'	“Noncentral t Distribution” on page B-131	$\nu$ : degrees of freedom	$\delta$ : noncentrality parameter	—	—
'Noncentral Chi-square'	“Noncentral Chi-Square Distribution” on page B-125	$\nu$ : degrees of freedom	$\delta$ : noncentrality parameter	—	—
'Normal'	“Normal Distribution” on page B-134	$\mu$ : mean	$\sigma$ : standard deviation	—	—
'Poisson'	“Poisson Distribution” on page B-143	$\lambda$ : mean	—	—	—
'Rayleigh'	“Rayleigh Distribution” on page B-146	$b$ : scale parameter	—	—	—
'Rician'	“Rician Distribution” on page B-149	$s$ : noncentrality parameter	$\sigma$ : scale parameter	—	—
'Stable'	“Stable Distribution” on page B-151	$\alpha$ : first shape parameter	$\beta$ : second shape parameter	$\gamma$ : scale parameter	$\delta$ : location parameter
'T'	“Student's t Distribution” on page B-166	$\nu$ : degrees of freedom	—	—	—
'tLocationScale'	“t Location-Scale Distribution” on page B-175	$\mu$ : location parameter	$\sigma$ : scale parameter	$\nu$ : shape parameter	—
'Uniform'	“Uniform Distribution (Continuous)” on page B-184	$a$ : lower endpoint (minimum)	$b$ : upper endpoint (maximum)	—	—



name	Distribution	Input Parameter A	Input Parameter B	Input Parameter C	Input Parameter D
'Discrete Uniform'	“Uniform Distribution (Discrete)” on page B-190	$n$ : maximum observable value	—	—	—
'Weibull'	“Weibull Distribution” on page B-193	$a$ : scale parameter	$b$ : shape parameter	—	—

### **y** — Probability values at which to evaluate icdf

scalar value | array of scalar values

Probability values at which to evaluate the icdf, specified as a scalar value, or an array of scalar values.

- If  $x$  is a scalar value, and if you specify distribution parameters A, B, C, or D as arrays, then `cdf` expands  $x$  into a constant array of the same size as the parameters.
- If  $x$  is an array, and if you specify distribution parameters A, B, C, or D as arrays, then  $x$ , A, B, C, and D must all be the same size.

Example: `[0.1,0.25,0.5,0.75,0.9]`

Data Types: `single` | `double`

### **A** — First probability distribution parameter

scalar value | array of scalar values

First probability distribution parameter, specified as a scalar value, or an array of scalar values.

If  $x$  and A are arrays, they must be the same size. If  $x$  is a scalar, then `cdf` expands it into a constant matrix the same size as A. If A is a scalar, then `cdf` expands it into a constant matrix the same size as  $x$ .

Data Types: `single` | `double`

### **B** — Second probability distribution parameter

scalar value | array of scalar values

Second probability distribution parameter, specified as a scalar value, or an array of scalar values.

If  $x$ ,  $A$ , and  $B$  are arrays, they must be the same size. If  $x$  is a scalar, then `cdf` expands it into a constant matrix the same size as  $A$  and  $B$ . If  $A$  or  $B$  are scalars, then `cdf` expands them into constant matrices the same size as  $x$ .

Data Types: `single` | `double`

### **C** — Third probability distribution parameter

scalar value | array of scalar values

Third probability distribution parameter, specified as a scalar value, or an array of scalar values.

If  $x$ ,  $A$ ,  $B$ , and  $C$  are arrays, they must be the same size. If  $x$  is a scalar, then `cdf` expands it into a constant matrix the same size as  $A$ ,  $B$ , and  $C$ . If any of  $A$ ,  $B$  or  $C$  are scalars, then `cdf` expands them into constant matrices the same size as  $x$ .

Data Types: `single` | `double`

### **D** — Fourth probability distribution parameter

scalar value | array of scalar values

Fourth probability distribution parameter, specified as a scalar value, or an array of scalar values.

If  $x$ ,  $A$ ,  $B$ ,  $C$ , and  $D$  are arrays, they must be the same size. If  $x$  is a scalar, then `cdf` expands it into a constant array the same size as  $A$ ,  $B$ ,  $C$ , and  $D$ . If any of  $A$ ,  $B$ ,  $C$ , or  $D$  are scalars, then `cdf` expands them into constant matrices the same size as  $x$ .

Data Types: `single` | `double`

### **pd** — Probability distribution

probability distribution object

Probability distribution, specified as a probability distribution object created using one of the following.

<code>makedist</code>	Create a probability distribution object using specified parameter values.
-----------------------	--

<code>fitdist</code>	Fit a probability distribution object to sample data.
<code>distributionFitter</code>	Fit a probability distribution object to sample data using the interactive Distribution Fitting app.
<code>paretotails</code>	Create a Pareto tails object.

## Output Arguments

### **x** — Inverse cumulative distribution function

array

Inverse cumulative distribution function of the specified probability distribution, returned as an array.

- If you specify distribution parameters *A*, *B*, *C*, or *D*, then *x* is the common size of *y*, *A*, *B*, *C*, and *D*, after any necessary scalar expansion.
- If you specify a probability distribution object, *pd*, then *x* has the same dimensions as *y*.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`cdf` | `mle` | `pdf` | `random`

### Topics

“Working with Probability Distributions” on page 5-3

**Introduced before R2006a**

# icdf

**Class:** `piecewisedistribution`

Inverse cumulative distribution function for piecewise distribution

## Syntax

```
X = icdf(obj,P)
```

## Description

`X = icdf(obj,P)` returns an array `X` of values of the inverse cumulative distribution function for the piecewise distribution object `obj`, evaluated at the values in the array `P`.

## Examples

Fit Pareto tails to a  $t$  distribution at cumulative probabilities 0.1 and 0.9:

```
t = trnd(3,100,1);
obj = paretotails(t,0.1,0.9);
[p,q] = boundary(obj)
p =
    0.1000
    0.9000
q =
   -1.7766
    1.8432

icdf(obj,p)
ans =
   -1.7766
    1.8432
```

## See Also

`cdf` | `paretotails`

# icdf

**Class:** ProbDistUnivKernel

Return inverse cumulative distribution function (ICDF) for ProbDistUnivKernel object

## Syntax

```
Y = icdf(PD, P)
```

## Description

$Y = \text{icdf}(PD, P)$  returns  $Y$ , an array containing the inverse cumulative distribution function (ICDF) for the ProbDistUnivKernel object  $PD$ , evaluated at values in  $P$ .

## Input Arguments

<i>PD</i>	An object of the class ProbDistUnivKernel.
<i>P</i>	A numeric array of values from 0 to 1 where you want to evaluate the ICDF.

## Output Arguments

<i>Y</i>	An array containing the inverse cumulative distribution function (ICDF) for the ProbDistUnivKernel object $PD$ .
----------	--

## See Also

icdf

## icdf

**Class:** ProbDistUnivParam

Return inverse cumulative distribution function (ICDF) for ProbDistUnivParam object

## Syntax

```
Y = icdf(PD, P)
```

## Description

$Y = \text{icdf}(PD, P)$  returns  $Y$ , an array containing the inverse cumulative distribution function (ICDF) for the ProbDistUnivParam object  $PD$ , evaluated at values in  $P$ .

## Input Arguments

<i>PD</i>	An object of the class ProbDistUnivParam.
<i>P</i>	A numeric array of values from 0 to 1 where you want to evaluate the ICDF.

## Output Arguments

<i>Y</i>	An array containing the inverse cumulative distribution function (ICDF) for the ProbDistUnivParam object $PD$ .
----------	---

## See Also

icdf



# icdf

**Class:** prob.TruncatableDistribution

**Package:** prob

Inverse cumulative distribution function of probability distribution object

## Syntax

```
y = icdf(pd,prob)
```

## Description

`y = icdf(pd,prob)` returns the inverse cumulative distribution function (icdf) values of the probability distribution `pd` at the probabilities in `prob`.

## Input Arguments

**pd — Probability distribution**

probability distribution object

Probability distribution, specified as a probability distribution object. Create a probability distribution object with specified parameter values using `makedist`. Alternatively, for fittable distributions, create a probability distribution object by fitting it to data using `fitdist` or the Distribution Fitting app.

**prob — Probabilities**

array of scalar values in the range [0,1]

Probabilities at which to compute the icdf, specified as an array of scalar values in the range [0,1]. For example, specifying `[.25 .5 .75]` returns a vector containing three icdf values corresponding to these probabilities.

Data Types: `single` | `double`

## Output Arguments

**y** — Inverse cumulative distribution function

array

Inverse cumulative distribution function (icdf) values of the specified probability distribution, evaluated at the probabilities in `prob`, returned as an array. `y` has the same dimensions as `x`.

## Examples

### Compute Standard Normal Critical Values

Create a standard normal distribution object.

```
pd = makedist('Normal')
```

```
pd =
```

```
NormalDistribution  
  
Normal distribution  
    mu = 0  
    sigma = 1
```

Determine the critical values at the 5% significance level for a test statistic with a standard normal distribution, by computing the upper and lower 2.5% values.

```
y = icdf(pd, [.025, .975])
```

```
y =
```

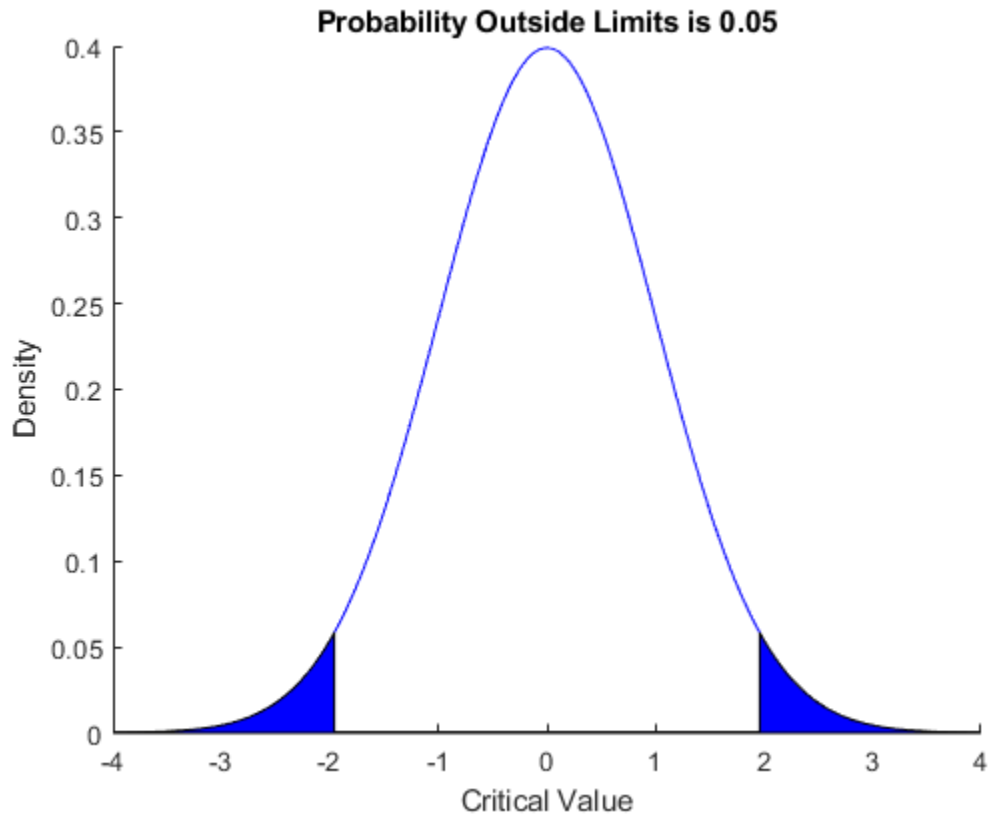
```
-1.9600    1.9600
```

Plot the cdf and shade the critical regions.

```
p = normspec(y, 0, 1, 'outside')
```

p =

0.0500



## See Also

[cdf](#) | [distributionFitter](#) | [fitdist](#) | [makedist](#) | [normspec](#) | [pdf](#)

## inconsistent

Inconsistency coefficient

### Syntax

```
Y = inconsistent(Z)
Y = inconsistent(Z, d)
```

### Description

`Y = inconsistent(Z)` computes the inconsistency coefficient for each link of the hierarchical cluster tree `Z`, where `Z` is an  $(m-1)$ -by-3 matrix generated by the `linkage` function. The inconsistency coefficient characterizes each link in a cluster tree by comparing its height with the average height of other links at the same level of the hierarchy. The higher the value of this coefficient, the less similar the objects connected by the link.

`Y = inconsistent(Z, d)` computes the inconsistency coefficient for each link in the hierarchical cluster tree `Z` to depth `d`, where `d` is an integer denoting the number of levels of the cluster tree that are included in the calculation. By default, `d=2`.

The output, `Y`, is an  $(m-1)$ -by-4 matrix formatted as follows.

Column	Description
1	Mean of the heights of all the links included in the calculation.
2	Standard deviation of the heights of all the links included in the calculation.
3	Number of links included in the calculation.
4	Inconsistency coefficient.

For each link,  $k$ , the inconsistency coefficient is calculated as:

$$Y(k,4) = (z(k,3) - Y(k,1)) / Y(k,2)$$

For leaf nodes, nodes that have no further nodes under them, the inconsistency coefficient is set to 0.

## Examples

### Compute Inconsistency Coefficient

Create the sample data.

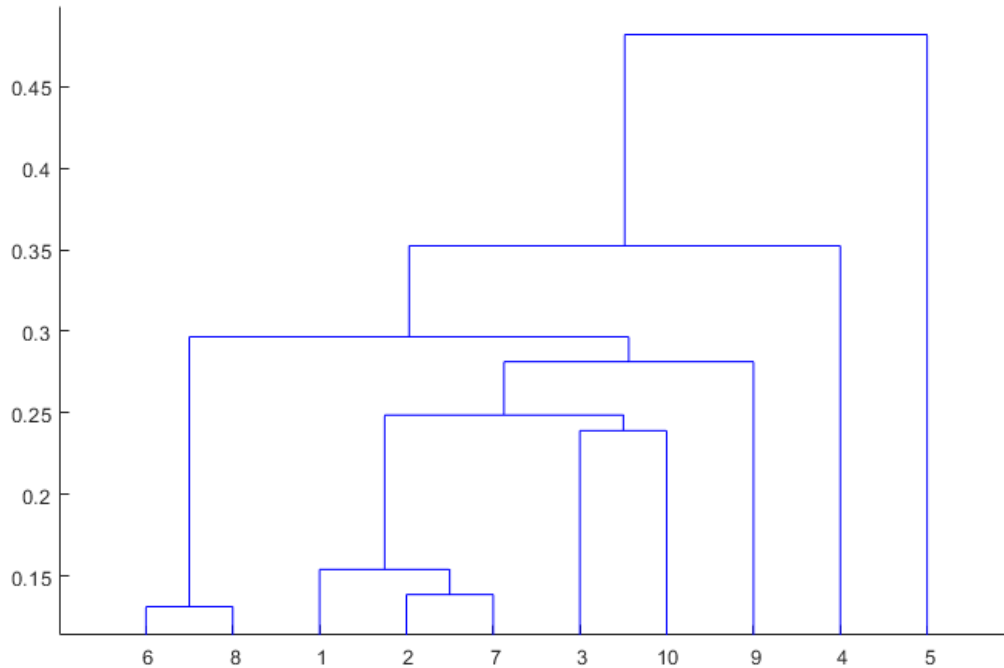
```
X = gallery('uniformdata',[10 2],12);  
Y = pdist(X);
```

Generate the hierarchical cluster tree.

```
Z = linkage(Y,'single');
```

Generate a dendrogram plot of the hierarchical cluster tree.

```
dendrogram(Z)
```



Compute the inconsistency coefficient for each link in the cluster tree  $Z$  to depth 3.

`W = inconsistent(Z,3)`

W =

0.1313	0	1.0000	0
0.1386	0	1.0000	0
0.1463	0.0109	2.0000	0.7071
0.2391	0	1.0000	0
0.1951	0.0568	4.0000	0.9425
0.2308	0.0543	4.0000	0.9320
0.2395	0.0748	4.0000	0.7636
0.2654	0.0945	4.0000	0.9203

0.3769    0.0950    3.0000    1.1040

## References

- [1] Jain, A., and R. Dubes. *Algorithms for Clustering Data*. Upper Saddle River, NJ: Prentice-Hall, 1988.
- [2] Zahn, C. T. “Graph-theoretical methods for detecting and describing Gestalt clusters.” *IEEE Transactions on Computers*. Vol. C-20, Issue 1, 1971, pp. 68–86.

## See Also

`cluster` | `clusterdata` | `cophenet` | `dendrogram` | `linkage` | `pdist` | `squareform`

**Introduced before R2006a**

## increaseB

**Class:** clustering.evaluation.GapEvaluation

**Package:** clustering.evaluation

Increase reference data sets

## Syntax

```
eva_out = increaseB(eva, nref)
```

## Description

`eva_out = increaseB(eva, nref)` returns a gap criterion clustering evaluation object `eva_out` that uses the same evaluation criteria as the input object `eva` and an additional number of reference data sets as specified by `nref`.

## Input Arguments

**eva** — Clustering evaluation data

clustering evaluation object

Clustering evaluation data, specified as a clustering evaluation object. Create a clustering evaluation object using `evalclusters`.

**nref** — Number of additional reference data sets

positive integer value

Number of additional reference data sets, specified as a positive integer value.

## Output Arguments

**eva\_out** — Updated clustering evaluation data

clustering evaluation object



Updated clustering evaluation data, returned as a gap criterion clustering evaluation object. `eva_out` contains evaluation data obtained using the reference data sets from the input object `eva` plus a number of additional reference data sets as specified in `nref`.

`increaseB` updates the `B` property of the input object `eva` to reflect the increase in the number of reference data sets used to compute the gap criterion values. `increaseB` also updates the `CriterionValues` property with gap criterion values computed using the total number of reference data sets. `increaseB` might also update the `OptimalK` and `OptimalY` properties to reflect the optimal number of clusters and optimal clustering solution as determined using the total number of reference data sets. Additionally, `increaseB` might also update the `LogW`, `ExpectedLogW`, `StdLogW`, and `SE` properties.

## Examples

### Evaluate Clustering Solutions Using Additional Reference Data

Create a gap clustering evaluation object using `evalclusters`, then use `increaseB` to increase the number of reference data sets used to compute the gap criterion values.

Load the sample data.

```
load fisheriris
```

The data contains length and width measurements from the sepals and petals of three species of iris flowers.

Cluster the flower measurement data using `kmeans`, and use the gap criterion to evaluate proposed solutions of one through five clusters. Use 50 reference data sets.

```
eva = evalclusters(meas, 'kmeans', 'gap', 'klist', 1:5, 'B', 50)
```

```
eva =
```

```
GapEvaluation with properties:
```

```
NumObservations: 150
    InspectedK: [1 2 3 4 5]
CriterionValues: [0.0870 0.5822 0.8766 1.0007 1.0465]
```

```
OptimalK: 4
```

The clustering evaluation object `eva` contains data on each proposed clustering solution. The returned results indicate that the optimal number of clusters is five.

The value of the `B` property of `eva` shows 50 reference data sets.

```
eva.B
```

```
ans =
```

```
50
```

Increase the number of reference data sets by 50, for a total of 100 sets.

```
eva = increaseB(eva,50)
```

```
eva =
```

```
GapEvaluation with properties:
```

```
NumObservations: 150
```

```
InspectedK: [1 2 3 4 5]
```

```
CriterionValues: [0.0814 0.5845 0.8736 1.0034 1.0506]
```

```
OptimalK: 4
```

The returned results now indicate that the optimal number of clusters is four.

The value of the `B` property of `eva` now shows 100 reference data sets.

```
eva.B
```

```
ans =
```

100

## See Also

evalclusters

## InputData property

**Class:** ProbDist

Read-only structure containing information about input data to ProbDist object

### Description

InputData is a read-only property of the ProbDist class. InputData is a structure containing information about input data to a ProbDist object. It includes the following fields:

- data
- cens
- freq

### Values

Possible values for the three fields in the structure are any data supplied to the `fitdist` function:

- `data` — Data passed to the `fitdist` function when creating the ProbDist object. This field is empty if the ProbDist object was created without fitting to data, that is by using the `ProbDistUnivParam` constructor.
- `cens` — The vector supplied with the 'censoring' parameter when creating the ProbDist object using the `fitdist` function. This field is empty if the ProbDist object was created without fitting to data, that is by using the `ProbDistUnivParam` constructor.
- `freq` — The vector supplied with the 'frequency' parameter when creating the ProbDist object using the `fitdist` function. This field is empty if the ProbDist object was created without fitting to data, that is by using the `ProbDistUnivParam` constructor.

Use this information to view and compare the data supplied to create distributions.

# interactionplot

Interaction plot for grouped data

## Syntax

```
interactionplot(Y,GROUP)
interactionplot(Y,GROUP,'varnames',VARNAMES)
[h,AX,bigax] = interactionplot(...)
```

## Description

`interactionplot(Y,GROUP)` displays the two-factor interaction plot for the group means of matrix `Y` with groups defined by entries in the cell array `GROUP`. `Y` is a numeric matrix or vector. If `Y` is a matrix, the rows represent different observations and the columns represent replications of each observation. If `Y` is a vector, the rows give the means of each entry in the cell array `GROUP`. Each cell of `GROUP` must contain a grouping variable that can be a categorical variable, numeric vector, character matrix, or a single-column cell array of character vectors. `GROUP` can also be a matrix whose columns represent different grouping variables. Each grouping variable must have the same number of rows as `Y`. The number of grouping variables must be greater than 1.

The interaction plot is a matrix plot, with the number of rows and columns both equal to the number of grouping variables. The grouping variable names are printed on the diagonal of the plot matrix. The plot at off-diagonal position  $(i,j)$  is the interaction of the two variables whose names are given at row diagonal  $(i,i)$  and column diagonal  $(j,j)$ , respectively.

`interactionplot(Y,GROUP,'varnames',VARNAMES)` displays the interaction plot with user-specified grouping variable names `VARNAMES`. `VARNAMES` is a character matrix or a cell array of character vectors, one per grouping variable. Default names are 'X1', 'X2', ...

`[h,AX,bigax] = interactionplot(...)` returns a handle `h` to the figure window, a matrix `AX` of handles to the subplot axes, and a handle `bigax` to the big (invisible) axes framing the subplots.

## Examples

### Display Interaction Plots

Randomly generate data for a response variable  $y$ .

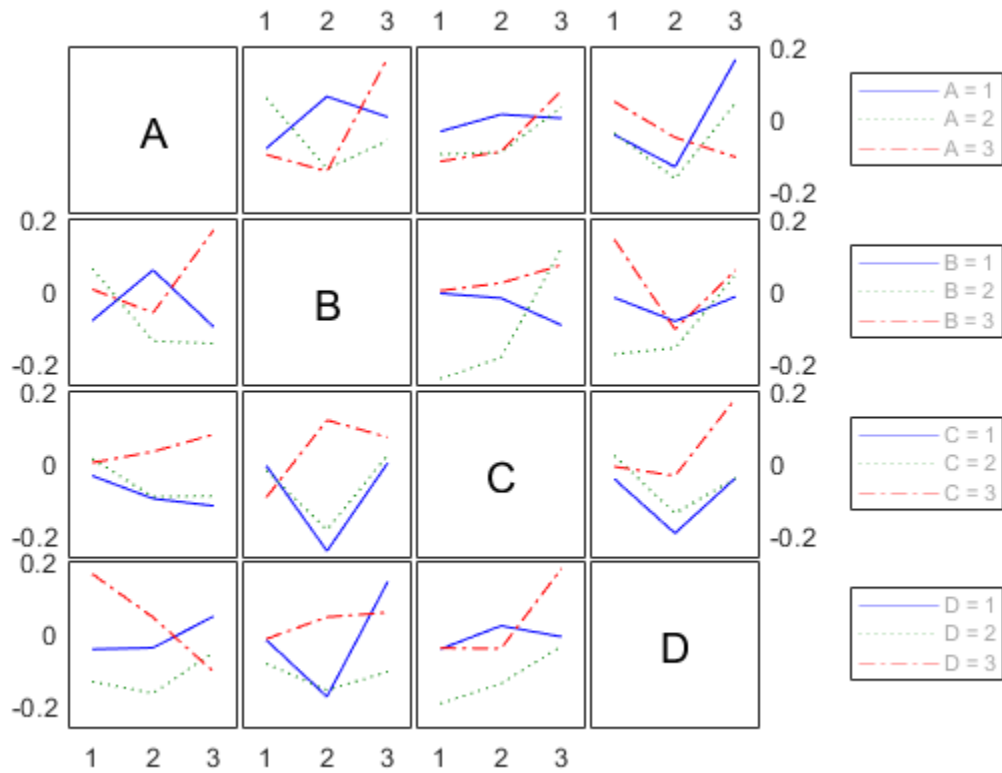
```
rng default; % For reproducibility  
y = randn(1000,1);
```

Randomly generate data for four three-level factors.

```
group = ceil(3*rand(1000,4));
```

Display the interaction plots for the factors and name the factors 'A', 'B', 'C', 'D'.

```
interactionplot(y,group,'varnames',{'A','B','C','D'})
```



## See Also

[maineffectsplot](#) | [multivarichart](#)

Introduced in R2006b

## intersect

**Class:** dataset

Set intersection for dataset array observations

---

**Note** The `dataset` data type might be removed in a future release. To work with heterogeneous data, use the MATLAB® `table` data type instead. See MATLAB `table` documentation for more information.

---

## Syntax

```
C = intersect(A,B)
C = intersect(A,B,vars)
C = intersect(A,B,vars,setOrder)
[C,iA,iB] = intersect(____)
```

## Description

`C = intersect(A,B)` for dataset arrays `A` and `B` returns the common set of observations from the two arrays, with repetitions removed. The observations in the dataset array `C` are in sorted order.

`C = intersect(A,B,vars)` returns the set of common observations from the two arrays, considering only the variables specified in `vars`, with repetitions removed. The observations in the dataset array `C` are sorted by those variables.

The values for variables not specified in `vars` for each observation in `C` are taken from the corresponding observations in `A`. If there are multiple observations in `A` that correspond to an observation in `C`, then those values are taken from the first occurrence.

`C = intersect(A,B,vars,setOrder)` returns the observations in `C` in the order specified by `setOrder`.

`[C,iA,iB] = intersect(____)` also returns index vectors `iA` and `iB` such that `C = A(iA,:)` and `C = B(iB,:)`. If there are repeated observations in `A` or `B`, then



`intersect` returns the index of the first occurrence. You can use any of the previous input arguments.

## Input Arguments

**A,B**

Input dataset arrays.

**vars**

Cell array of character vectors containing variable names or a vector of integers containing variable column numbers, indicating the variables in A and B that `intersect` considers.

Specify `vars` as `[]` to use its default value of all variables.

**setOrder**

Flag indicating the sorting order for the observations in C. The possible values of `setOrder` are:

'sorted'	Observations in C are in sorted order (default).
'stable'	Observations in C are in the same order that they appear in A.

## Output Arguments

**C**

Dataset array with the common set of observations in A and B, with repetitions removed. C is in sorted order (by default), or the order specified by `setOrder`.

**iA**

Index vector, indicating the observations in A that are common to B. The vector `iA` contains the index to the first occurrence of any repeated observations in A.

**iB**

Index vector, indicating the observations in B that are common to A. The vector `iB` contains the index to the first occurrence of any repeated observations in B.

## Examples

### Intersection of Two Dataset Arrays

Navigate to the folder containing sample data, and load sample data.

```
cd(matlabroot)
cd('help/toolbox/stats/examples')

A = dataset('XLSFile','hospitalSmall.xlsx');
B = dataset('XLSFile','hospitalSmall.xlsx','Sheet',2);
```

Return the intersection and index vectors.

```
[C,iA,iB] = intersect(A,B);
```

C =

id	name	sex	age	wgt	smoke
'TRW-072'	'WHITE'	'm'	39	202	1

There is one observation in common between A and B.

Find the observation in the original dataset arrays.

```
A(iA,:)
```

ans =

id	name	sex	age	wgt	smoke
'TRW-072'	'WHITE'	'm'	39	202	1

```
B(iB,:)
```

ans =

id	name	sex	age	wgt	smoke
'TRW-072'	'WHITE'	'm'	39	202	1

- “Merge Dataset Arrays” on page 2-107

## See Also

dataset | ismember | setdiff | setxor | sortrows | union | unique

## Topics

“Merge Dataset Arrays” on page 2-107

“Dataset Arrays” on page 2-142

## invpred

Inverse prediction

### Syntax

```
X0 = invpred(X, Y, Y0)
[X0, DXLO, DXUP] = invpred(X, Y, Y0)
[X0, DXLO, DXUP] = invpred(X, Y, Y0, name1, val1, name2, val2, ...)
```

### Description

`X0 = invpred(X, Y, Y0)` accepts vectors `X` and `Y` of the same length, fits a simple regression, and returns the estimated value `X0` for which the height of the line is equal to `Y0`. The output, `X0`, has the same size as `Y0`, and `Y0` can be an array of any size.

`[X0, DXLO, DXUP] = invpred(X, Y, Y0)` also computes 95% inverse prediction intervals. `DXLO` and `DXUP` define intervals with lower bound `X0-DXLO` and upper bound `X0+DXUP`. Both `DXLO` and `DXUP` have the same size as `Y0`.

The intervals are not simultaneous and are not necessarily finite. Some intervals may extend from a finite value to `-Inf` or `+Inf`, and some may extend over the entire real line.

`[X0, DXLO, DXUP] = invpred(X, Y, Y0, name1, val1, name2, val2, ...)` specifies optional argument name/value pairs chosen from the following list. Argument names are case insensitive and partial matches are allowed.

Name	Value
'alpha'	A value between 0 and 1 specifying a confidence level of $100 * (1 - \text{alpha})\%$ . Default is <code>alpha=0.05</code> for 95% confidence.

Name	Value
'predopt'	Either 'observation', the default value to compute the intervals for X0 at which a new observation could equal Y0, or 'curve' to compute intervals for the X0 value at which the curve is equal to Y0.

## Examples

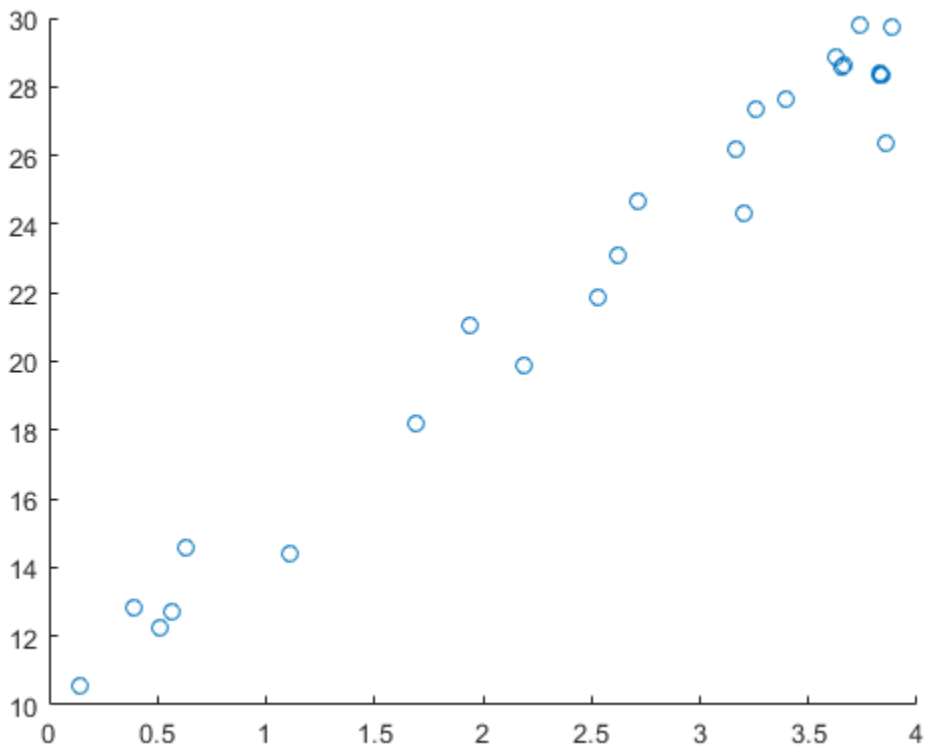
### Inverse Prediction

Generate sample data.

```
x = 4*rand(25,1);  
y = 10 + 5*x + randn(size(x));
```

Make a scatterplot of the data.

```
scatter(x,y)
```



Predict the  $x$  value for a given  $y$  value of 20.

```
x0 = invpred(x,y,20)
```

```
x0 =
```

```
1.9967
```

## See Also

`polyconf` | `polyfit` | `polytool` | `polyval`

**Introduced before R2006a**

## iqr

Interquartile range

### Syntax

```
r = iqr(x)
r = iqr(x,dim)

r = iqr(pd)
```

### Description

`r = iqr(x)` returns the interquartile range of the values in `x`.

`r = iqr(x,dim)` returns the interquartile range along the dimension of `x` specified by `dim`.

`r = iqr(pd)` returns the interquartile range of the probability distribution, `pd`.

### Examples

#### Compute the Interquartile Range

Generate a 4-by-4 matrix of random data from a normal distribution with parameter values  $\mu$  equal to 10 and  $\sigma$  equal to 1.

```
rng default % For reproducibility
x = normrnd(10,1,4)
```

`x =`

```
    10.5377    10.3188    13.5784    10.7254
    11.8339     8.6923    12.7694     9.9369
```



```

7.7412    9.5664    8.6501    10.7147
10.8622   10.3426   13.0349    9.7950

```

Compute the interquartile range for each column of data.

```
r = iqr(x)
```

```
r =
```

```

2.2086    1.2013    2.5969    0.8541

```

Compute the interquartile range for each row of data.

```
r2 = iqr(x,2)
```

```
r2 =
```

```

1.7237
2.9870
1.9449
1.8797

```

### Compute the Normal Distribution Interquartile Range

Create a standard normal distribution object with the mean,  $\mu$ , equal to 0 and the standard deviation,  $\sigma$ , equal to 1.

```
pd = makedist('Normal',0,1);
```

Compute the interquartile range of the standard normal distribution.

```
r = iqr(pd)
```

```
r =
```

```

1.3490

```

The returned value is the difference between the 75th and the 25th percentile values for the distribution. This is equivalent to computing the difference between the inverse cumulative distribution function (icdf) values at the probabilities  $y$  equal to 0.75 and 0.25.

```
r2 = icdf(pd,0.75) - icdf(pd,0.25)
```

```
r2 =  
    1.3490
```

## Input Arguments

### **x** — Input array

vector | matrix | multidimensional array

Input array, specified as a vector, matrix, or multidimensional array.

Data Types: `single` | `double`

### **dim** — Dimension

1 (default) | positive integer value

Dimension along which the interquartile range is calculated, specified as a positive integer. For example, for a matrix  $x$ , when `dim` is equal to 1, `iqr` returns the interquartile range for the columns of  $x$ . When `dim` is equal to 2, `iqr` returns the interquartile range for the rows of  $x$ . For  $n$ -dimensional arrays, `iqr` operates along the first nonsingleton dimension of  $x$ .

Data Types: `single` | `double`

### **pd** — Probability distribution

probability distribution object

Probability distribution, specified as a probability distribution object created using one of the following.

<code>makedist</code>	Create a probability distribution object using specified parameter values.
-----------------------	--

<code>fitdist</code>	Fit a probability distribution object to sample data.
<code>distributionFitter</code>	Fit a probability distribution object to sample data using the interactive Distribution Fitter app.

## Output Arguments

### **`r`** — Interquartile range

scalar value

Interquartile range, returned as a scalar value.

- If you input a vector for `x`, then `r` is the difference between the 75th and the 25th percentiles of the sample data contained in `x`.
- If you input a matrix for `x`, then `r` is a row vector containing the difference between the 75th and the 25th percentiles of the sample data contained each column of `x`.
- If you input a probability distribution, `pd`, then the value of `r` is the difference between the values of the 75th and 25th percentile of the probability distribution.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`icdf` | `mad` | `range` | `std`

### Topics

“Working with Probability Distributions” on page 5-3

**Introduced before R2006a**

## iqr

**Class:** ProbDistUnivKernel

Return interquartile range (IQR) for ProbDistUnivKernel object

## Syntax

$Y = \text{iqr}(PD)$

## Description

$Y = \text{iqr}(PD)$  returns  $Y$ , the interquartile range for the ProbDistUnivKernel object  $PD$ . The interquartile range is the distance between the 75th and 25th percentiles.

## Input Arguments

$PD$  An object of the class ProbDistUnivKernel.

## Output Arguments

$Y$  The value of the interquartile range for the ProbDistUnivKernel object  $PD$ .

## See Also

ProbDistUnivKernel.icdf | iqr

## iqr

**Class:** ProbDistUnivParam

Return interquartile range (IQR) for ProbDistUnivParam object

## Syntax

$Y = \text{iqr}(PD)$

## Description

$Y = \text{iqr}(PD)$  returns  $Y$ , the interquartile range for the ProbDistUnivParam object  $PD$ . The interquartile range is the distance between the 75th and 25th percentiles.

## Input Arguments

$PD$  An object of the class ProbDistUnivParam.

## Output Arguments

$Y$  The value of the interquartile range for the ProbDistUnivParam object  $PD$ .

## See Also

ProbDistUnivParam.icdf | iqr

# iqr

**Class:** prob.TruncatableDistribution

**Package:** prob

Interquartile range of probability distribution object

## Syntax

```
r = iqr(pd)
```

## Description

`r = iqr(pd)` returns the interquartile range `r` of the probability distribution `pd`.

## Input Arguments

**pd** — Probability distribution

probability distribution object

Probability distribution, specified as a probability distribution object. Create a probability distribution object with specified parameter values using `makedist`. Alternatively, for fittable distributions, create a probability distribution object by fitting it to data using `fitdist` or the Distribution Fitting app.

## Output Arguments

**r** — Interquartile range

scalar value

Interquartile range of the probability distribution, returned as a scalar value. The value of `r` is the difference between the values of the 75th and 25th percentile of the probability distribution.

## Examples

### Interquartile Range of a Fitted Distribution

Load the sample data. Create a vector containing the first column of students' exam grade data.

```
load examgrades;  
x = grades(:,1);
```

Create a normal distribution object by fitting it to the data.

```
pd = fitdist(x, 'Normal')
```

```
pd =
```

```
NormalDistribution
```

```
Normal distribution
```

```
mu = 75.0083 [73.4321, 76.5846]
```

```
sigma = 8.7202 [7.7391, 9.98843]
```

Compute the interquartile range of the fitted distribution.

```
r = iqr(pd)
```

```
r =
```

```
11.7634
```

The returned result indicates that the difference between the 75th and 25th percentile of the students' grades is 11.7634.

Use `icdf` to determine the 75th and 25th percentiles of the students' grades.

```
y = icdf(pd, [0.25, 0.75])
```

```
y =
```



```
69.1266 80.8900
```

Calculate the difference between the 75th and 25th percentiles. This yields the same result as `iqr`.

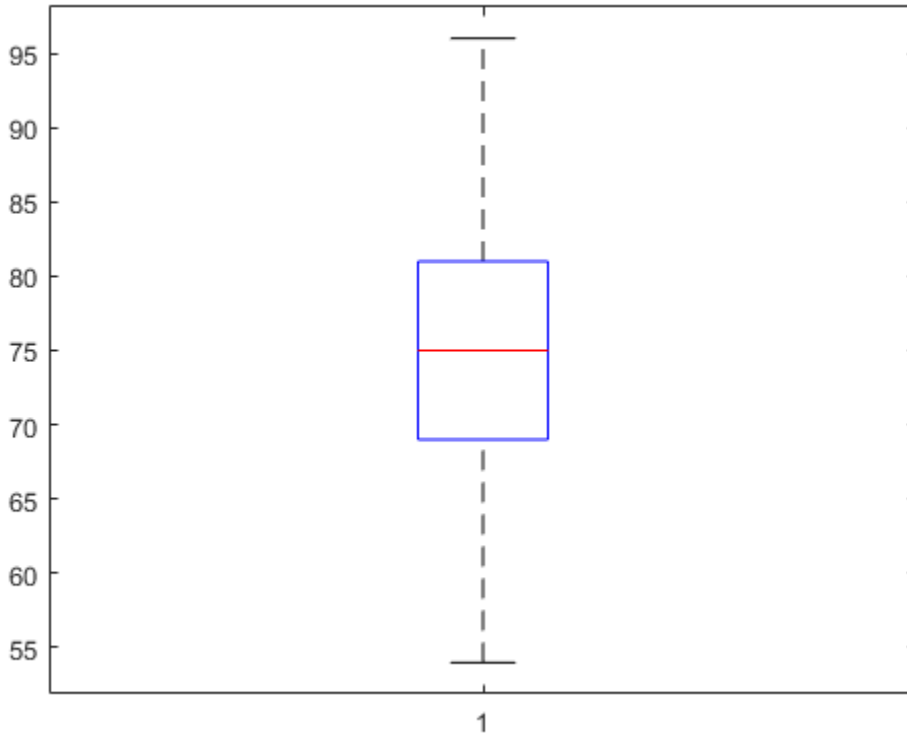
```
y(2) - y(1)
```

```
ans =
```

```
11.7634
```

Use `boxplot` to visualize the interquartile range.

```
boxplot(x)
```



The top line of the box shows the 75th percentile, and the bottom line shows the 25th percentile. The center line shows the median, which is the 50th percentile.

## See Also

`boxplot` | `distributionFitter` | `fitdist` | `makedist`

# isbranch

**Class:** classregtree

Test node for branch

---

**Note** classregtree will be removed in a future release. See fitctree, fitrtree, ClassificationTree, or RegressionTree instead.

---

## Syntax

```
ib = isbranch(t)
ib = isbranch(t,nodes)
```

## Description

`ib = isbranch(t)` returns an  $n$ -element logical vector `ib` that is true for each branch node and false for each leaf node.

`ib = isbranch(t,nodes)` takes a vector `nodes` of node numbers and returns a vector of logical values for the specified nodes.

## Examples

Create a classification tree for Fisher's iris data:

```
load fisheriris;

t = classregtree(meas,species,...
                'names',{'SL' 'SW' 'PL' 'PW'})
t =
Decision tree for classification
1  if PL<2.45 then node 2 elseif PL>=2.45 then node 3 else setosa
2  class = setosa
3  if PW<1.75 then node 4 elseif PW>=1.75 then node 5 else versicolor
4  if PL<4.95 then node 6 elseif PL>=4.95 then node 7 else versicolor
5  class = virginica
6  if PW<1.65 then node 8 elseif PW>=1.65 then node 9 else versicolor
```

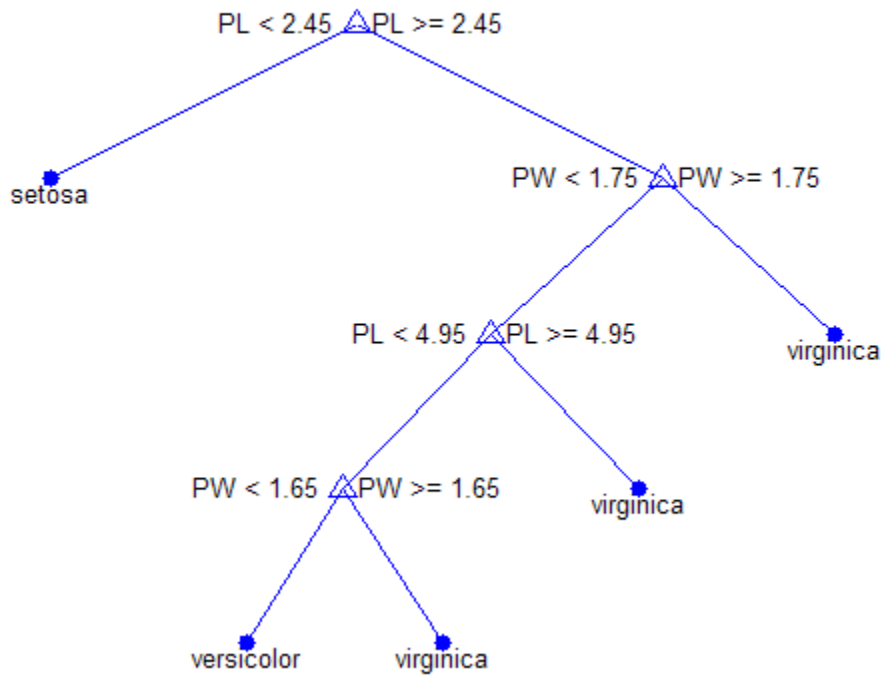
```

7 class = virginica
8 class = versicolor
9 class = virginica

```

```
view(t)
```

Click to display:  Magnification:  Pruning level:



```
ib = isbranch(t)
```

```
ib =
```

```

1
0
1
1
0
1
1
0

```

0  
0

## References

- [1] Breiman, L., J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Boca Raton, FL: CRC Press, 1984.

## See Also

`classregtree` | `cutvar` | `numnodes`

## isempty

**Class:** dataset

True for empty dataset array

---

**Note** The `dataset` data type might be removed in a future release. To work with heterogeneous data, use the MATLAB® `table` data type instead. See MATLAB `table` documentation for more information.

---

## Syntax

```
tf = isempty(A)
```

## Description

`tf = isempty(A)` returns true (1) if A is an empty dataset and false (0) otherwise. An empty array has no elements, that is `prod(size(A))==0`.

## See Also

`size`

# islevel

Determine if levels are in nominal or ordinal array

---

**Note** The `nominal` and `ordinal` array data types might be removed in a future release. To represent ordered and unordered discrete, nonnumeric data, use the MATLAB® `categorical` data type instead.

---

## Syntax

```
tf = islevel(levels,A)
```

## Description

`tf = islevel(levels,A)` returns a logical array indicating which of the levels in `levels` correspond to a level in the nominal or ordinal array `A`.

## Input Arguments

### **A** — Nominal or ordinal array

nominal array | ordinal array

Nominal or ordinal array, specified as a `nominal` or `ordinal` array object created using `nominal` or `ordinal`.

### **levels** — Levels to test

character vector | cell array of character vectors | 2-D character matrix

Levels to test, specified as a character vector, cell array of character vectors, or 2-D character matrix.

Data Types: `char` | `cell`

## Output Arguments

### **tf** — Logical array

array the same size as `levels`

Logical array, returned as an array the same size as `levels`. `tf` has value 1 (true) where the corresponding element of `levels` is the label of a level in the nominal or ordinal array `A`, even if the level contains no elements. `tf` has value 0 (false) otherwise.

## See Also

`isequal` | `ismember` | `nominal` | `ordinal`

## Topics

`nominal`

`ordinal`

**Introduced in R2007a**



# ismember

**Class:** dataset

Dataset array elements that are members of set

---

**Note** The `dataset` data type might be removed in a future release. To work with heterogeneous data, use the MATLAB® `table` data type instead. See MATLAB `table` documentation for more information.

---

## Syntax

```
LiA = ismember(A,B)  
LiA = ismember(A,B,vars)  
[LiA,LocB] = ismember(____)
```

## Description

`LiA = ismember(A,B)` for dataset arrays `A` and `B` returns a vector of logical values the same length as `A`. The output vector, `LiA`, has value 1 (true) in the elements that correspond to observations in `A` that are also present in `B`, and 0 (false) otherwise.

`LiA = ismember(A,B,vars)` returns a vector of logical values the same length as `A`. The output vector, `LiA`, has value 1 (true) in the elements that correspond to observations in `A` that are also present in `B` for the variables specified in `vars` only, and 0 (false) otherwise.

`[LiA,LocB] = ismember(____)` also returns a vector the same length as `A` containing the index to the first observation in `B` that corresponds to each observation in `A`, or 0 if there is no such observation. You can use any of the previous input arguments.

## Input Arguments

### **A**

Query dataset array, containing the observations to be found in B.

### **B**

Set dataset array. When an observation in A is found in B, for all variables or only those variables specified in `vars`, the corresponding element of `LiA` is 1.

### **vars**

Cell array of character vectors containing variable names or a vector of integers containing variable column numbers, indicating which variables to match observations on in A and B.

## Output Arguments

### **LiA**

Vector of logical values the same length as A. `LiA` has value 1 (true) when the corresponding observation in A is present in B. Otherwise, `LiA` has value 0 (false).

If you specify `vars`, `LiA` has value 1 when the corresponding observation in A is present in B for the variables in `vars` only.

### **LocB**

Vector the same length as A containing the index to the first observation in B that corresponds to each observation in A, for all variables or only those variables specified in `vars`.

## Examples

### **Find Observations That Are Members of a Dataset Array**

Load sample data.

```
load('hospital')
B = hospital(1:50,1:5);
```

This set dataset array, B, has 50 observations on 5 variables.

Specify a query dataset array.

```
rng('default')
rIx = randsample(100,10);
A = hospital(rIx,1:5)
```

A =

	LastName	Sex	Age	Weight	Smoker
YLN-495	'COLEMAN'	Male	39	188	false
LQW-768	'TAYLOR'	Female	31	132	false
DGC-290	'BUTLER'	Male	38	184	true
DAU-529	'REED'	Male	50	186	true
REV-997	'ALEXANDER'	Male	25	171	true
QEQ-082	'COX'	Female	28	111	false
AGR-528	'SIMMONS'	Male	45	181	false
PUE-347	'YOUNG'	Female	25	114	false
HVR-372	'RUSSELL'	Male	44	188	true
XUE-826	'JACKSON'	Male	25	174	false

Check which observations in A are present in B.

```
LiA = ismember(A,B)
```

LiA =

10x1 logical array

```
0
1
0
0
0
0
0
0
1
0
```

```
1
```

Display the observations in A that are present in B.

```
A(LiA, :)
```

```
ans =
```

	LastName	Sex	Age	Weight	Smoker
LQW-768	'TAYLOR'	Female	31	132	false
PUE-347	'YOUNG'	Female	25	114	false
XUE-826	'JACKSON'	Male	25	174	false

Find the location of the observations in B.

```
[~, LocB] = ismember(A, B)
```

```
LocB =
```

```
0
10
0
0
0
0
0
0
28
0
13
```

Display the observations in B that match observations in A.

```
B(LocB(LocB>0), :)
```

```
ans =
```

	LastName	Sex	Age	Weight	Smoker
LQW-768	'TAYLOR'	Female	31	132	false
PUE-347	'YOUNG'	Female	25	114	false

```
XUE-826      'JACKSON'      Male      25      174      false
```

## See Also

`dataset` | `intersect` | `setdiff` | `setxor` | `sortrows` | `union` | `unique`

## Topics

“Dataset Arrays” on page 2-142

## ismissing

**Class:** dataset

Find dataset array elements with missing values

---

**Note** The `dataset` data type might be removed in a future release. To work with heterogeneous data, use the MATLAB® `table` data type instead. See MATLAB `table` documentation for more information.

---

## Syntax

```
I = ismissing(ds)
I = ismissing(ds,Name,Value)
```

## Description

`I = ismissing(ds)` returns a logical array that indicates which elements in the dataset array, `ds`, contain a missing value. By default, `ismissing` recognizes NaN as a missing value in numeric variables, ' ' as a missing value in character variables, and <undefined> as a missing value in categorical arrays.

- `ds2 = ds(~any(I,2),:)` creates a new dataset array containing only the complete observations in `ds`.
- `ds2 = ds(:,~any(I,1))` creates a new dataset array containing only the variables from `ds` with no missing values.

`I = ismissing(ds,Name,Value)` returns missing value indices with additional options specified by one or more `Name,Value` pair arguments.

## Input Arguments

### **ds**

dataset array

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### **NumericTreatAsMissing**

Vector of numeric values to treat as missing value indicators in floating point `ds` variables. `ismissing` always treats a NaN value as a missing value.

### **Default:**

### **StringTreatAsMissing**

Character vector or cell array of character vectors to treat as missing value indicators in character `ds` variables. `ismissing` always treats ' ' as a missing value.

## Output Arguments

### **I**

Logical array indicating which elements in `ds` contain a missing value. `I` is the same size as `ds`, with value 1 for elements that contain a missing value.

## See Also

`dataset` | `isempty` | `isnan` | `isundefined` | `replaceWithMissing`

## Topics

“Clean Messy and Missing Data” on page 2-123

“Dataset Arrays” on page 2-142



# invalid

**Class:** grandstream

Test handle validity

## Syntax

```
tf = invalid(h)
```

## Description

`tf = invalid(h)` performs an element-wise check for validity on the handle elements of `h`. The result is a logical array of the same dimensions as `h`, where each element is the element-wise validity result.

A handle is invalid if it has been deleted or if it is an element of a handle array and has not yet been initialized.

## See Also

`delete` | `grandstream`

## NumIterations property

**Class:** `gmdistribution`

Number of iterations

### Description

The number of iterations of the algorithm.

---

**Note** This property applies only to `gmdistribution` objects constructed with `fitgmdist`.

---

# iwishrnd

Inverse Wishart random numbers

## Syntax

```
W = iwishrnd(Tau,df)
W = iwishrnd(Tau,df,DI)
[W,DI] = iwishrnd(Tau,df)
```

## Description

`W = iwishrnd(Tau,df)` generates a random matrix `W` from the inverse Wishart distribution with parameters `Tau` and `df`. The inverse of `W` has the Wishart distribution with covariance matrix `Sigma = inv(Tau)` and with `df` degrees of freedom. `Tau` is a symmetric and positive definite matrix.

`W = iwishrnd(Tau,df,DI)` expects `DI` to be the transpose of the inverse of the Cholesky factor of `Tau`, so that `DI'*DI = inv(Tau)`, where `inv` is the MATLAB inverse function. `DI` is lower-triangular and the same size as `Tau`. If you call `iwishrnd` multiple times using the same value of `Tau`, it is more efficient to supply `DI` instead of computing it each time.

`[W,DI] = iwishrnd(Tau,df)` returns `DI` so you can use it as an input in future calls to `iwishrnd`.

Note that different sources use different parametrizations for the inverse Wishart distribution. This function defines the parameter `tau` so that the mean of the output matrix is `Tau/(df-d-1)` where `d` is the dimension of `Tau`.

## See Also

`wishrnd`

## **Topics**

“Inverse Wishart Distribution” on page B-83

**Introduced before R2006a**

# jackknife

Jackknife sampling

## Syntax

```
jackstat = jackknife(jackfun,X)
jackstat = jackknife(jackfun,X,Y,...)
jackstat = jackknife(jackfun,...,'Options',option)
```

## Description

`jackstat = jackknife(jackfun,X)` draws jackknife data samples from the  $n$ -by- $p$  data array `X`, computes statistics on each sample using the function `jackfun`, and returns the results in the matrix `jackstat`. `jackknife` regards each row of `X` as one data sample, so there are  $n$  data samples. Each of the  $n$  rows of `jackstat` contains the results of applying `jackfun` to one jackknife sample. `jackfun` is a function handle specified with `@`. Row  $i$  of `jackstat` contains the results for the sample consisting of `X` with the  $i$ th row omitted:

```
s = x;
s(i,:) = [];
jackstat(i,:) = jackfun(s);
```

If `jackfun` returns a matrix or array, then this output is converted to a row vector for storage in `jackstat`. If `X` is a row vector, it is converted to a column vector.

`jackstat = jackknife(jackfun,X,Y,...)` accepts additional arguments to be supplied as inputs to `jackfun`. They may be scalars, column vectors, or matrices. `jackknife` creates each jackknife sample by sampling with replacement from the rows of the non-scalar data arguments (these must have the same number of rows). Scalar data are passed to `jackfun` unchanged. Non-scalar arguments must have the same number of rows, and each jackknife sample omits the same row from these arguments.

`jackstat = jackknife(jackfun,...,'Options',option)` provides an option to perform jackknife iterations in parallel, if the Parallel Computing Toolbox is available.

Set 'Options' as a structure you create with `statset`. `jackknife` uses the following field in the structure:

`'UseParallel'` If `true` and if a parpool of the Parallel Computing Toolbox is open, use multiple processors to compute jackknife iterations. If the Parallel Computing Toolbox is not installed, or a parpool is not open, computation occurs in serial mode. Default is `false`, or serial computation.

## Examples

Estimate the bias of the MLE variance estimator of random samples taken from the vector `y` using `jackknife`. The bias has a known formula in this problem, so you can compare the `jackknife` value to this formula.

```
sigma = 5;
y = normrnd(0,sigma,100,1);
m = jackknife(@var, y, 1);
n = length(y);
bias = -sigma^2 / n % known bias formula
jbias = (n - 1)*(mean(m)-var(y,1)) % jackknife bias estimate

bias =
    -0.2500

jbias =
    -0.3378
```

## See Also

`bootstrp` | `histogram` | `ksdensity` | `random` | `randsample`

## Topics

“Jackknife Resampling” on page 3-20

Introduced in R2006a

# jbtest

Jarque-Bera test

## Syntax

```
h = jbtest(x)
h = jbtest(x,alpha)
h = jbtest(x,alpha,mctol)
[h,p] = jbtest(____)
[h,p,jbstat,critval] = jbtest(____)
```

## Description

`h = jbtest(x)` returns a test decision for the null hypothesis that the data in vector `x` comes from a normal distribution with an unknown mean and variance, using the Jarque-Bera test on page 32-3118. The alternative hypothesis is that it does not come from such a distribution. The result `h` is 1 if the test rejects the null hypothesis at the 5% significance level, and 0 otherwise.

`h = jbtest(x,alpha)` returns a test decision for the null hypothesis at the significance level specified by `alpha`.

`h = jbtest(x,alpha,mctol)` returns a test decision based on a  $p$ -value computed using a Monte Carlo simulation with a maximum Monte Carlo standard error on page 32-3118 less than or equal to `mctol`.

`[h,p] = jbtest(____)` also returns the  $p$ -value `p` of the hypothesis test, using any of the input arguments from the previous syntaxes.

`[h,p,jbstat,critval] = jbtest(____)` also returns the test statistic `jbstat` and the critical value `critval` for the test.

## Examples

### Test for a Normal Distribution

Load the data set.

```
load carbig
```

Test the null hypothesis that car mileage, in miles per gallon (MPG), follows a normal distribution across different makes of cars.

```
h = jbtest(MPG)
```

```
h =
```

```
1
```

The returned value of  $h = 1$  indicates that `jbtest` rejects the null hypothesis at the default 5% significance level.

### Test the Hypothesis at a Different Significance Level

Load the data set.

```
load carbig
```

Test the null hypothesis that car mileage in miles per gallon (MPG) follows a normal distribution across different makes of cars at the 1% significance level.

```
[h,p] = jbtest(MPG,0.01)
```

```
h =
```

```
1
```

```
p =
```

```
0.0022
```



The returned value of  $h = 1$ , and the returned  $P$ -value less than  $\alpha = 0.01$  indicate that `jbtest` rejects the null hypothesis.

### Test for a Normal Distribution Using Monte Carlo Simulation

Load the data set.

```
load carbig
```

Test the null hypothesis that car mileage, in miles per gallon (MPG), follows a normal distribution across different makes of cars. Use a Monte Carlo simulation to obtain an exact  $P$ -value.

```
[h,p,jbstat,critval] = jbtest(MPG,[],0.0001)
```

```
h =
```

```
    1
```

```
p =
```

```
  0.0022
```

```
jbstat =
```

```
  18.2275
```

```
critval =
```

```
   5.8461
```

The returned value of `h = 1` indicates that `jbtest` rejects the null hypothesis at the default 5% significance level. Additionally, the test statistic, `jbstat`, is larger than the critical value, `critval`, which indicates rejection of the null hypothesis.

## Input Arguments

### **x** — Sample data

vector

Sample data for the hypothesis test, specified as a vector. `jbtest` treats NaN values in `x` as missing values and ignores them.

Data Types: `single` | `double`

### **alpha** — Significance level

0.05 (default) | scalar value in the range (0,1)

Significance level of the hypothesis test, specified as a scalar value in the range (0,1). If `alpha` is in the range [0.001,0.50], and if the sample size is less than or equal to 2000, `jbtest` looks up the critical value for the test in a table of precomputed values. To conduct the test at a significance level outside of these specifications, use `mctol`.

Example: 0.01

Data Types: `single` | `double`

### **mctol** — Maximum Monte Carlo standard error

nonnegative scalar value

Maximum Monte Carlo standard error on page 32-3118 for the  $p$ -value, `p`, specified as a nonnegative scalar value. If you specify a value for `mctol`, `jbtest` computes a Monte Carlo approximation for `p` directly, rather than interpolating into a table of precomputed values. `jbtest` chooses the number of Monte Carlo replications large enough to make the Monte Carlo standard error for `p` less than `mctol`.

If you specify a value for `mctol`, you must also specify a value for `alpha`. You can specify `alpha` as `[]` to use the default value of 0.05.

Example: 0.0001

Data Types: `single` | `double`

## Output Arguments

### **h** — Hypothesis test result

1 | 0

Hypothesis test result, returned as a logical value.

- If  $h = 1$ , this indicates the rejection of the null hypothesis at the alpha significance level.
- If  $h = 0$ , this indicates a failure to reject the null hypothesis at the alpha significance level.

### **p** — *p*-value

scalar value in the range (0,1)

*p*-value of the test, returned as a scalar value in the range (0,1). *p* is the probability of observing a test statistic as extreme as, or more extreme than, the observed value under the null hypothesis. Small values of *p* cast doubt on the validity of the null hypothesis.

`jbstest` warns when *p* is not found within the tabulated range of [0.001,0.50], and returns either the smallest or largest tabulated value. In this case, you can use `mctol` to compute a more accurate *p*-value.

### **jbstat** — Test statistic

nonnegative scalar value

Test statistic for the Jarque-Bera test, returned as a nonnegative scalar value.

### **critval** — Critical value

nonnegative scalar value

Critical value for the Jarque-Bera test at the alpha significance level, returned as a nonnegative scalar value. If alpha is in the range [0.001,0.50], and if the sample size is less than or equal to 2000, `jbstest` looks up the critical value for the test in a table of precomputed values. If you use `mctol`, `jbstest` determines the critical value of the test using a Monte Carlo simulation. The null hypothesis is rejected when `jbstat > critval`.

## Definitions

### Jarque-Bera Test

The Jarque-Bera test is a two-sided goodness-of-fit test suitable when a fully specified null distribution is unknown and its parameters must be estimated.

The test is specifically designed for alternatives in the Pearson system of distributions. The test statistic is

$$JB = \frac{n}{6} \left( s^2 + \frac{(k-3)^2}{4} \right),$$

where  $n$  is the sample size,  $s$  is the sample skewness, and  $k$  is the sample kurtosis. For large sample sizes, the test statistic has a chi-square distribution with two degrees of freedom.

### Monte Carlo Standard Error

The Monte Carlo standard error is the error due to simulating the  $p$ -value.

The Monte Carlo standard error is calculated as

$$SE = \sqrt{\frac{(\hat{p})(1-\hat{p})}{\text{mcreps}}},$$

where  $\hat{p}$  is the estimated  $p$ -value of the hypothesis test, and `mcreps` is the number of Monte Carlo replications performed. `jbtest` chooses the number of Monte Carlo

replications, `mcreps`, large enough to make the Monte Carlo standard error for  $\hat{p}$  less than the value specified for `mctol`.

## Algorithms

Jarque-Bera tests often use the chi-square distribution to estimate critical values for large samples, deferring to the Lilliefors test (see `lillietest`) for small samples.

`jbtest`, by contrast, uses a table of critical values computed using Monte Carlo simulation for sample sizes less than 2000 and significance levels from 0.001 to 0.50.

Critical values for a test are computed by interpolating into the table, using the analytic chi-square approximation only when extrapolating for larger sample sizes.

## References

- [1] Jarque, C. M., and A. K. Bera. “A Test for Normality of Observations and Regression Residuals.” *International Statistical Review*. Vol. 55, No. 2, 1987, pp. 163–172.
- [2] Deb, P., and M. Sefton. “The Distribution of a Lagrange Multiplier Test of Normality.” *Economics Letters*. Vol. 51, 1996, pp. 123–130. This paper proposed a Monte Carlo simulation for determining the distribution of the test statistic. The results of this function are based on an independent Monte Carlo simulation, not the results in this paper.

## See Also

`adtest` | `kstest` | `lillietest`

## Topics

“Generating Data Using the Pearson System” on page 7-29

Introduced before R2006a

## johnsrnd

Johnson system random numbers

### Syntax

```
r = johnsrnd(quantiles,m,n)
r = johnsrnd(quantiles)
[r,type] = johnsrnd(...)
[r,type,coefs] = johnsrnd(...)
```

### Description

`r = johnsrnd(quantiles,m,n)` returns an *m*-by-*n* matrix of random numbers drawn from the distribution in the Johnson system that satisfies the quantile specification given by `quantiles`. `quantiles` is a four-element vector of quantiles for the desired distribution that correspond to the standard normal quantiles  $[-1.5 -0.5 0.5 1.5]$ . In other words, you specify a distribution from which to draw random values by designating quantiles that correspond to the cumulative probabilities  $[0.067 0.309 0.691 0.933]$ . `quantiles` may also be a 2-by-4 matrix whose first row contains four standard normal quantiles, and whose second row contains the corresponding quantiles of the desired distribution. The standard normal quantiles must be spaced evenly.

---

**Note** Because `r` is a random sample, its sample quantiles typically differ somewhat from the specified distribution quantiles.

---

`r = johnsrnd(quantiles)` returns a scalar value.

`r = johnsrnd(quantiles,m,n,...)` or `r = johnsrnd(quantiles,[m,n,...])` returns an *m*-by-*n*-by-... array.

`[r,type] = johnsrnd(...)` returns the type of the specified distribution within the Johnson system. `type` is 'SN', 'SL', 'SB', or 'SU'. Set *m* and *n* to zero to identify the distribution type without generating any random values.

The four distribution types in the Johnson system correspond to the following transformations of a normal random variate:

- 'SN' — Identity transformation (normal distribution on page B-134)
- 'SL' — Exponential transformation (lognormal distribution on page B-100)
- 'SB' — Logistic transformation (bounded)
- 'SU' — Hyperbolic sine transformation (unbounded)

`[r,type,coefs] = johnsrnd(...)` returns coefficients `coefs` of the transformation that defines the distribution. `coefs` is `[gamma, eta, epsilon, lambda]`. If `z` is a standard normal random variable and `h` is one of the transformations defined above,  $r = \lambda * h((z - \text{gamma}) / \text{eta}) + \text{epsilon}$  is a random variate from the distribution type corresponding to `h`.

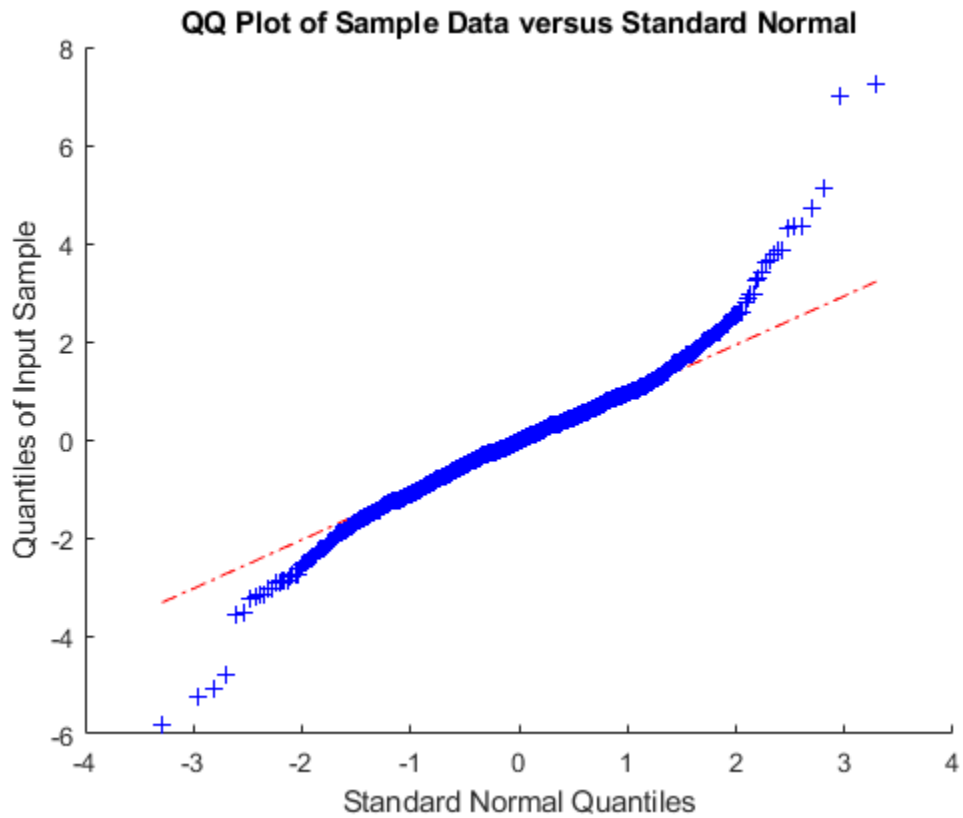
## Examples

### Generate Random Samples Using the Johnson System

This example shows several different approaches to using the Johnson system of flexible distribution families to generate random numbers and fit a distribution to sample data.

Generate random values with longer tails than a standard normal.

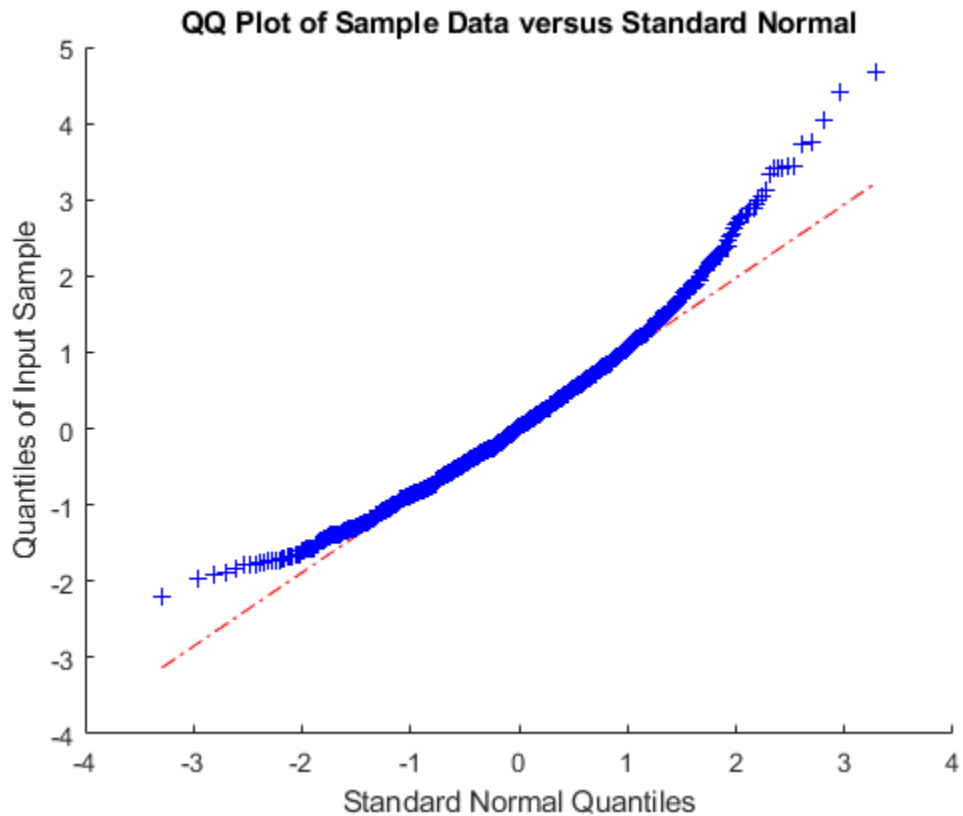
```
rng default; % For reproducibility
r = johnsrnd([-1.7 -.5 .5 1.7],1000,1);
figure;
qqplot(r);
```



Generate random values skewed to the right.

```
r = johnsrnd([-1.3 -.5 .5 1.7],1000,1);  
figure;  
qqplot(r);
```





Generate random values that match some sample data well in the right-hand tail.

```
load carbig;
qnorm = [.5 1 1.5 2];
q = quantile(Acceleration, normcdf(qnorm));
r = johnsrnd([qnorm;q], 1000, 1);
[q; quantile(r, normcdf(qnorm))]
```

ans =

```
16.7000    18.2086    19.5376    21.7263
16.6986    18.2220    19.9078    22.0918
```

Determine the distribution type and the coefficients.

```
[r,type,coefs] = johnsrnd([qnorm;q],0)
```

```
r =
```

```
    []
```

```
type =
```

```
    'SU'
```

```
coefs =
```

```
    1.0920    0.5829    18.4382    1.4494
```

## See Also

`pearsrnd` | `random`

## Topics

“Johnson System” on page B-85

**Introduced in R2006a**

# join

**Class:** dataset

Merge observations

---

**Note** The `dataset` data type might be removed in a future release. To work with heterogeneous data, use the MATLAB® `table` data type instead. See MATLAB `table` documentation for more information.

---

## Syntax

```
C = join(A,B)
C = join(A,B,keys)
C = join(A,B,param1, val1,param2, val2, ...)
[C,IB] = join(...)
C = join(A,B,'Type',TYPE,...)
C = join(A,B,'Type',TYPE,'MergeKeys',true,...)
[C,IA,IB] = join(A,B,'Type',TYPE,...)
```

## Description

`C = join(A,B)` creates a dataset array `C` by merging observations from the two dataset arrays `A` and `B`. `join` performs the merge by first finding *key variables*, that is, pairs of dataset variables, one in `A` and one in `B`, that share the same name. Each observation in `B` must contain a unique combination of values in the key variables, and must contain all combinations of values that are present in the keys from `A`. `join` then uses these key variables to define a many-to-one correspondence between observations in `A` and those in `B`. `join` uses this correspondence to replicate the observations in `B` and combine them with the observations in `A` to create `C`.

`C = join(A,B,keys)` performs the merge using the variables specified by `keys` as the key variables in both `A` and `B`. `keys` is a positive integer, a vector of positive integers, a variable name, a cell array of variable names, or a logical vector.

C contains one observation for each observation in A. Variables in C include all of the variables from A, as well as one variable corresponding to each variable in B (except for the keys from B). If A and B contain variables with identical names, `join` adds the suffix `'_left'` and `'_right'` to the corresponding variables in C.

`C = join(A, B, param1, val1, param2, val2, ...)` specifies optional parameter name/value pairs to control how the dataset variables in A and B are used in the merge.

Parameters are:

- `'Keys'` — Specifies the variables to use as keys in both A and B.
- `'LeftKeys'` — Specifies the variables to use as keys in A.
- `'RightKeys'` — Specifies the variables to use as keys in B.

You may provide either the `'Keys'` parameter, or both the `'LeftKeys'` and `'RightKeys'` parameters. The value for these parameters is a positive integer, a vector of positive integers, a variable name, a cell array containing variable names, or a logical vector. `'LeftKeys'` or `'RightKeys'` must both specify the same number of key variables, and `join` pairs the left and right keys in the order specified.

- `'LeftVars'` — Specifies which variables from A to include in C. By default, `join` includes all variables from A.
- `'RightVars'` — Specifies which variables from B to include in C. By default, `join` includes all variables from B except the key variables.

You can use `'LeftVars'` or `'RightVars'` to include or exclude key variables as well as data variables. The value for these parameters is a positive integer, a vector of positive integers, a variable name, a cell array containing one or more variable names, or a logical vector.

`[C, IB] = join(...)` returns an index vector IB, where `join` constructs C by horizontally concatenating `A(:, LeftVars)` and `B(IB, RightVars)`. `join` can also perform more complicated inner and outer join operations that allow a many-to-many correspondence between A and B, and allow unmatched observations in either A or B.

`C = join(A, B, 'Type', TYPE, ...)` performs the join operation specified by TYPE. TYPE is one of `'inner'`, `'leftouter'`, `'rightouter'`, `'fullouter'`, or `'outer'` (which is a synonym for `'fullouter'`). For an inner join, C only contains observations corresponding to a combination of key values that occurred in both A and B. For a left (or right) outer join, C also contains observations corresponding to keys in A (or B) that did

not match any in B (or A). Variables in C taken from A (or B) contain null values in those observations. A full outer join is equivalent to a left and right outer join. C contains variables corresponding to the key variables from both A and B, and `join` sorts the observations in C by the key values.

For inner and outer joins, C contains variables corresponding to the key variables from both A and B by default, as well as all the remaining variables. `join` sorts the observations in the result C by the key values.

`C = join(A,B, 'Type', TYPE, 'MergeKeys', true, ...)` includes a single variable in C for each key variable pair from A and B, rather than including two separate variables. For outer joins, `join` creates the single variable by merging the key values from A and B, taking values from A where a corresponding observation exists in A, and from B otherwise. Setting the `'MergeKeys'` parameter to `true` overrides inclusion or exclusion of any key variables specified via the `'LeftVars'` or `'RightVars'` parameter. Setting the `'MergeKeys'` parameter to `false` is equivalent to not passing in the `'MergeKeys'` parameter.

`[C, IA, IB] = join(A,B, 'Type', TYPE, ...)` returns index vectors IA and IB indicating the correspondence between observations in C and those in A and B. For an inner join, `join` constructs C by horizontally concatenating A (IA, LeftVars) and B (IB, RightVars). For an outer join, IA or IB may also contain zeros, indicating the observations in C that do not correspond to observations in A or B, respectively.

## Examples

Create a dataset array from Fisher's iris data:

```
load fisheriris
NumObs = size(meas,1);
NameObs = strcat({'Obs'}, num2str((1:NumObs)', '%-d'));
iris = dataset({nominal(species), 'species'}, ...
              {meas, 'SL', 'SW', 'PL', 'PW'}, ...
              'ObsNames', NameObs);
```

Create a separate dataset array with the diploid chromosome counts for each species of iris:

```
snames = nominal({'setosa'; 'versicolor'; 'virginica'});
CC = dataset({snames, 'species'}, {[38;108;70], 'cc'})
```

```
CC =
  species      cc
  setosa       38
  versicolor   108
  virginica    70
```

**Broadcast the data in CC to the rows of iris using the key variable species in each dataset:**

```
iris2 = join(iris,CC);
iris2([1 2 51 52 101 102],:)
ans =
  species      SL      SW      PL      PW      cc
  Obs1        setosa    5.1    3.5    1.4    0.2    38
  Obs2        setosa    4.9     3    1.4    0.2    38
  Obs51       versicolor  7      3.2    4.7    1.4   108
  Obs52       versicolor  6.4    3.2    4.5    1.5   108
  Obs101      virginica   6.3    3.3     6    2.5    70
  Obs102      virginica   5.8    2.7    5.1    1.9    70
```

**Create two datasets and join them using the 'MergeKeys' flag:**

```
% Create two data sets that both contain the key variable
% 'Key1'. The two arrays contain observations with common
% values of Key1, but each array also contains observations
% with values of Key1 not present in the other.
a = dataset({'a' 'b' 'c' 'e' 'h'},[1 2 3 11 17]',...
  'VarNames',{'Key1' 'Var1'})
b = dataset({'a' 'b' 'd' 'e'},[4 5 6 7]',...
  'VarNames',{'Key1' 'Var2'})

% Combine a and b with an outer join, which matches up
% observations with common key values, but also retains
% observations whose key values don't have a match.
% Keep the key values as separate variables in the result.
couter = join(a,b,'key','Key1','Type','outer')

% Join a and b, merging the key values as a single variable
% in the result.
coutermerge = join(a,b,'key','Key1','Type','outer',...
  'MergeKeys',true)

% Join a and b, retaining only observations whose key
% values match.
cinner = join(a,b,'key','Key1','Type','inner',...)
```

```
'MergeKeys',true)

a =

  Key1    Var1
  'a'      1
  'b'      2
  'c'      3
  'e'     11
  'h'     17

b =

  Key1    Var2
  'a'      4
  'b'      5
  'd'      6
  'e'      7

couter =

  Key1_left  Var1  Key1_right  Var2
  'a'        1    'a'          4
  'b'        2    'b'          5
  'c'        3    ''           NaN
  ''         NaN  'd'          6
  'e'       11    'e'          7
  'h'       17    ''           NaN

coutermerge =

  Key1    Var1  Var2
  'a'      1    4
  'b'      2    5
  'c'      3    NaN
  'd'     NaN    6
  'e'     11    7
  'h'     17    NaN

cinner =
```

Key1	Var1	Var2
'a'	1	4
'b'	2	5
'e'	11	7

## See Also

sortrows



# KDTreeSearcher

Grow *Kd*-tree

## Syntax

```
Mdl = KDTreeSearcher(X)
Mdl = KDTreeSearcher(X,Name,Value)
```

## Description

`Mdl = KDTreeSearcher(X)` grows a default *Kd*-tree (`Mdl`) using the  $n$ -by- $K$  numeric matrix of training data (`X`). `Mdl` is a `KDTreeSearcher` model object that stores the results of the grown *Kd*-tree. You can use `Mdl` to search the training data (`X`) for the nearest neighbors to the query data.

`Mdl = KDTreeSearcher(X,Name,Value)` grows a *Kd*-tree (`Mdl`) with additional options specified by one or more `Name,Value` pair arguments. For example, you can specify a distance metric or the maximum number of observations in each leaf node (i.e., the bucket size).

## Examples

### Grow a Default *Kd*-Tree

Load Fisher's iris data set.

```
load fisheriris
X = meas;
[n,k] = size(X)
```

```
n =
```

```
150
```

```
k =  
    4
```

`X` has 150 observations and 4 predictors.

Grow a 4-dimensional  $K$  d-tree using the entire data set as training data.

```
Mdl = KDTreeSearcher(X)
```

```
Mdl =
```

```
  KDTreeSearcher with properties:
```

```
    BucketSize: 50  
    Distance: 'euclidean'  
    DistParameter: []  
           X: [150x4 double]
```

`Mdl` is a `KDTreeSearcher` model object, and its properties appear in the Command Window. It contains information about the grown 4-dimensional  $K$  d-tree, such as the distance metric. You can alter property values using dot notation

To find the nearest neighbors in `X` to a batch of query data, pass `Mdl` and the query data to `knnsearch` or `rangesearch`.

### Specify the Minkowski Distance for Nearest Neighbor Search

Load Fisher's iris data. Focus on the petal dimensions.

```
load fisheriris  
X = meas(:, [3 4]); % Predictors
```

Grow a two-dimensional  $K$  d-tree using `createns` and the training data. Specify the Minkowski distance metric.

```
Mdl = createns(X, 'NSMethod', 'kdtree', 'Distance', 'Minkowski')
```

```
Mdl =

KDTreesearcher with properties:

    BucketSize: 50
      Distance: 'minkowski'
DistParameter: 2
           X: [150x2 double]
```

Mdl is a KDTreesearcher model object. Access properties of Mdl using dot notation. For example, use Mdl.DistParameter to access the Minkowski distance exponent.

```
Mdl.DistParameter
```

```
ans =

     2
```

You can pass query data and Mdl to:

- knnsearch to find indices and distances of nearest neighbors.
- rangesearch to find indices of all nearest neighbors within a distance that you specify.

### Search for Nearest Neighbors of Query Data Using the Minkowski Distance

Load Fisher's iris data set.

```
load fisheriris
```

Remove five irises randomly from the predictor data to use as a query set.

```
rng(1); % For reproducibility
n = size(meas,1); % Sample size
qIdx = randsample(n,5); % Indices of query data
tIdx = ~ismember(1:n,qIdx); % Indices of training data
Q = meas(qIdx,:);
X = meas(tIdx,:);
```

Grow a four-dimensional  $K$  d-tree using the training data. Specify to use the Minkowski distance for finding nearest neighbors later.

```
Mdl = createns(X, 'NSMethod', 'kdtree', 'Distance', 'minkowski')
```

```
Mdl =
```

```
    KDTreeSearcher with properties:
```

```
        BucketSize: 50
           Distance: 'minkowski'
    DistParameter: 2
                X: [145x4 double]
```

Mdl is a KDTreeSearcher model object. By default, the Minkowski distance exponent is 2.

Find the indices of the training data (X) that are the two nearest neighbors of each point in the query data (Q).

```
IdxNN = knnsearch(Mdl,Q, 'K', 2)
```

```
IdxNN =
```

```
    17     4
     6     2
     1    12
    89    66
   124   100
```

Each row of NN corresponds to a query data observation, and the column order corresponds to the order of the nearest neighbors. For example, using the Minkowski distance, the second nearest neighbor of  $Q(3, :)$  is  $X(12, :)$ .

## Input Arguments

### **x** — Training data

numeric matrix

Training data that grows the *Kd*-tree, specified as a numeric matrix.  $X$  has  $n$  rows, each corresponding to an observation (i.e., an instance or example), and  $K$  columns, each corresponding to a predictor or feature.

Data Types: `single` | `double`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'Distance', 'minkowski', 'P', 3, 'BucketSize', 10` specifies to use the Minkowski distance when searching for nearest neighbors, to use 3 for the Minkowski distance metric exponent, and to use 10 for the bucket size.

### Distance — Distance metric

`'euclidean'` (default) | `'chebychev'` | `'cityblock'` | `'minkowski'`

Distance metric used to find nearest neighbors of query points, specified as the comma-separated pair consisting of `'Distance'` and one of these values.

Value	Description
<code>'chebychev'</code>	Chebychev distance (maximum coordinate difference)
<code>'cityblock'</code>	City block distance
<code>'euclidean'</code>	Euclidean distance
<code>'minkowski'</code>	Minkowski distance

For more details, see “Distance Metrics” on page 18-16.

The software does not use the distance metric for training the *Kd*-tree, so you can alter it after training using dot notation.

Example: `'Distance', 'minkowski'`

Data Types: `char`

### P — Exponent for Minkowski distance metric

2 (default) | positive scalar

Exponent for the Minkowski distance metric, specified as the comma-separated pair consisting of 'P' and a positive scalar. If you specify P and do not specify 'Distance', 'minkowski', then the software throws an error.

Example: 'P', 3

Data Types: double | single

**BucketSize** — Maximum number of data points in each leaf node

50 (default) | positive integer

Maximum number of data points in each leaf node of the *Kd*-tree, specified as the comma-separated pair consisting of 'BucketSize' and a positive integer.

Example: 'BucketSize', 10

Data Types: double | single

## Output Arguments

**Mdl** — Grown *Kd*-tree

KDTreeSearcher model object

Grown *Kd*-tree, returned as a KDTreeSearcher model object. To search the training data for the nearest neighbors of the query data, pass the query data and Mdl to `knnsearch` or `rangesearch`.

## See Also

`createns` | `knnsearch` | `rangesearch`

## Topics

KDTreeSearcher

“k-Nearest Neighbor Search and Radius Search” on page 18-18

“Distance Metrics” on page 18-16

Introduced in R2010a

# KDTreeSearcher

Nearest neighbor search using *Kd*-tree

## Description

`KDTreeSearcher` model objects store results of a nearest neighbor search using the *Kd*-tree algorithm. Results that you can store include the training data, the distance metric and its parameters, and the maximal number of data points in each leaf node (i.e., the bucket size). The *Kd*-tree algorithm partitions an  $n$ -by- $K$  data set by recursively splitting  $n$  points in  $K$ -dimensional space into a binary tree. To find the nearest neighbors of a query observation, `KDTreeSearcher` restricts the training data space to the training observations in the leaf node that the query observation belongs to.

Once you create or train a `KDTreeSearcher` model object, you can search the stored tree to find all neighboring points to the query data by performing a nearest neighbor search using `knnsearch` or radius search using `rangesearch`. The *Kd*-tree algorithm is particularly useful when:

- $K$  is relatively small (i.e.,  $K < 10$ ).
- The training and query sets are not sparse.
- The training and query sets have many observations.

## Creation

Create a `KDTreeSearcher` model object using `KDTreeSearcher` or `createns`.

## Properties

### **Distance** — Distance metric

'chebychev' | 'cityblock' | 'euclidean' | 'minkowski'

Distance metric used to find nearest neighbors of query points, specified as one of these values.

Value	Description
'chebychev'	Chebychev distance (maximum coordinate difference)
'cityblock'	City block distance
'euclidean'	Euclidean distance
'minkowski'	Minkowski distance

For more details, see “Distance Metrics” on page 18-16.

The software does not use the distance metric for training the *Kd*-tree, so you can alter it after training using dot notation.

Data Types: `char`

#### **DistParameter** — Distance metric parameter values

`[]` | positive scalar

Distance metric parameter values, specified as empty (`[]`) or as a positive scalar.

If `Distance` is 'minkowski', then:

- `DistParameter` is the exponent in the Minkowski distance formula.
- You can alter `DistParameter` of a trained `KDTreeSearcher` model.

Otherwise, `DistParameter` is `[]`, indicating that the specified distance metric formula has no parameters.

Data Types: `single` | `double`

#### **x** — Training data

numeric matrix

This property is read-only.

Training data that grows the *Kd*-tree, specified as a numeric matrix. `X` has  $n$  rows, each corresponding to an observation (i.e., an instance or example), and  $K$  columns, each corresponding to a predictor or feature.

Data Types: `single` | `double`



## Object Functions

`knnsearch`     k-nearest neighbor search using Kd-tree or exhaustive search  
`rangesearch`   Find all neighbors within specified distance using exhaustive search or Kd-tree

## Examples

### Grow a Default *K*d-Tree

Load Fisher's iris data set.

```
load fisheriris  
X = meas;  
[n,k] = size(X)
```

```
n =  
  
    150
```

```
k =  
  
     4
```

X has 150 observations and 4 predictors.

Grow a 4-dimensional *K* d-tree using the entire data set as training data.

```
Mdl = KdTreeSearcher(X)
```

```
Mdl =  
  
KdTreeSearcher with properties:  
  
    BucketSize: 50  
    Distance: 'euclidean'  
    DistParameter: []  
           X: [150x4 double]
```

Mdl is a `KDTreeSearcher` model object, and its properties appear in the Command Window. It contains information about the grown 4-dimensional  $K$  d-tree, such as the distance metric. You can alter property values using dot notation

To find the nearest neighbors in `X` to a batch of query data, pass Mdl and the query data to `knnsearch` or `rangesearch`.

### Alter Properties of `KDTreeSearcher` Model

Load Fisher's iris data set.

```
load fisheriris
X = meas;
```

Grow a default four-dimensional  $K$  d-tree using the entire data set as training data.

```
Mdl = KDTreeSearcher(X)
```

```
Mdl =
```

```
  KDTreeSearcher with properties:
```

```
    BucketSize: 50
      Distance: 'euclidean'
  DistParameter: []
             X: [150x4 double]
```

Specify that the neighbor searcher use the Minkowski metric to compute the distances between the training and query data.

```
Mdl.Distance = 'minkowski'
```

```
Mdl =
```

```
  KDTreeSearcher with properties:
```

```
    BucketSize: 50
      Distance: 'minkowski'
  DistParameter: 2
```

```
X: [150x4 double]
```

Pass `Mdl` and the query data to either `knnsearch` or `rangesearch` to find the nearest neighbors to the points in the query data using the Minkowski distance.

### Search for Nearest Neighbors of Query Data Using the Minkowski Distance

Load Fisher's iris data set.

```
load fisheriris
```

Remove five irises randomly from the predictor data to use as a query set.

```
rng(1); % For reproducibility
n = size(meas,1); % Sample size
qIdx = randsample(n,5); % Indices of query data
tIdx = ~ismember(1:n,qIdx); % Indices of training data
Q = meas(qIdx,:);
X = meas(tIdx,:);
```

Grow a four-dimensional  $K$  d-tree using the training data. Specify to use the Minkowski distance for finding nearest neighbors later.

```
Mdl = createns(X, 'NSMethod', 'kdtree', 'Distance', 'minkowski')
```

```
Mdl =
```

```
KDTreearcher with properties:
```

```
    BucketSize: 50
    Distance: 'minkowski'
    DistParameter: 2
    X: [145x4 double]
```

`Mdl` is a `KDTreearcher` model object. By default, the Minkowski distance exponent is 2.

Find the indices of the training data (`X`) that are the two nearest neighbors of each point in the query data (`Q`).

```
IdxNN = knnsearch(Mdl,Q,'K',2)
```

```
IdxNN =
```

```
    17     4  
     6     2  
     1    12  
    89    66  

```

Each row of NN corresponds to a query data observation, and the column order corresponds to the order of the nearest neighbors. For example, using the Minkowski distance, the second nearest neighbor of  $Q(3, :)$  is  $X(12, :)$ .

## See Also

`ExhaustiveSearcher`

## Topics

“k-Nearest Neighbor Search and Radius Search” on page 18-18

“Distance Metrics” on page 18-16

**Introduced in R2010a**

# Kernel property

**Class:** ProbDistKernel

Read-only name of kernel smoothing function for ProbDistKernel object

## Description

Kernel is a read-only property of the ProbDistKernel class. Kernel is a character vector specifying the name of the kernel smoothing function used to create a ProbDistKernel object.

## Values

'normal'  
'box'  
'triangle'  
'epanechnikov'

Use this information to view and compare the kernel smoothing function used to create distributions.

## See Also

ksdensity

## kfoldEdge

**Class:** ClassificationPartitionedECOC

Classification edge for observations not used for training

### Syntax

```
edge = kfoldEdge(CVMdl)
edge = kfoldEdge(CVMdl, Name, Value)
```

### Description

`edge = kfoldEdge(CVMdl)` returns the classification edge on page 32-3151 obtained by the cross-validated ECOC model (`ClassificationPartitionedECOC`) `CVMdl`. For every fold, `kfoldEdge` computes the classification edge for in-fold observations using an ECOC model trained on out-of-fold observations. `CVMdl.X` contains both sets of observations.

`edge = kfoldEdge(CVMdl, Name, Value)` returns the classification edge with additional options specified by one or more `Name, Value` pair arguments.

For example, specify the number of folds, decoding scheme, or verbosity level.

### Input Arguments

**`CVMdl` — Cross-validated ECOC model**

`ClassificationPartitionedECOC` model

Cross-validated ECOC model, specified as a `ClassificationPartitionedECOC` model. You can create a `ClassificationPartitionedECOC` model by:

- Passing a trained ECOC model (`ClassificationECOC`) to `crossval`

- Training an ECOC model using `fitcecoc` and setting any one of these cross-validation name-value pair arguments: 'CrossVal', 'KFold', 'Holdout', 'Leaveout', or 'CVPartition'

## Name-Value Pair Arguments

Specify optional comma-separated pairs of Name, Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as Name1, Value1, ..., NameN, ValueN.

### BinaryLoss — Binary learner loss function

'hamming' | 'linear' | 'logit' | 'exponential' | 'binodeviance' | 'hinge' | 'quadratic' | function handle

Binary learner loss function, specified as the comma-separated pair consisting of 'BinaryLoss' and a built-in, loss-function name or function handle.

- This table contains names and descriptions of the built-in functions, where  $y_j$  is a class label for a particular binary learner (in the set  $\{-1, 1, 0\}$ ),  $s_j$  is the score for observation  $j$ , and  $g(y_j, s_j)$  is the binary loss formula.

Value	Description	Score Domain	$g(y_j, s_j)$
'binodeviance'	Binomial deviance	$(-\infty, \infty)$	$\log[1 + \exp(-2y_j s_j)] / [2\log(2)]$
'exponential'	Exponential	$(-\infty, \infty)$	$\exp(-y_j s_j) / 2$
'hamming'	Hamming	$[0, 1]$ or $(-\infty, \infty)$	$[1 - \text{sign}(y_j s_j)] / 2$
'hinge'	Hinge	$(-\infty, \infty)$	$\max(0, 1 - y_j s_j) / 2$
'linear'	Linear	$(-\infty, \infty)$	$(1 - y_j s_j) / 2$
'logit'	Logistic	$(-\infty, \infty)$	$\log[1 + \exp(-y_j s_j)] / [2\log(2)]$
'quadratic'	Quadratic	$[0, 1]$	$[1 - y_j(2s_j - 1)]^2 / 2$

The software normalizes the binary losses such that the loss is 0.5 when  $y_j = 0$ . Also, the software calculates the mean binary loss for each class.

- For a custom binary loss function, e.g., `customFunction`, specify its function handle 'BinaryLoss', @customFunction.

`customFunction` should have this form:

```
bLoss = customFunction(M,s)
```

where:

- $M$  is the  $K$ -by- $L$  coding matrix stored in `Mdl.CodingMatrix`.
- $s$  is the 1-by- $L$  row vector of classification scores.
- `bLoss` is the classification loss. This scalar aggregates the binary losses for every learner in a particular class. For example, you can use the mean binary loss to aggregate the loss over the learners for each class.
- $K$  is the number of classes.
- $L$  is the number of binary learners.

For an example of passing a custom binary loss function, see “Predict Test-Sample Labels of ECOC Models Using Custom Binary Loss Function” on page 32-4933.

By default, if all binary learners are:

- SVMs or linear classification models of SVM learners, then `BinaryLoss` is 'hinge'
- Ensembles trained by `AdaboostM1` or `GentleBoost`, then `BinaryLoss` is 'exponential'
- Ensembles trained by `LogitBoost`, then `BinaryLoss` is 'binodeviance'
- Linear classification models of logistic regression learners, or you specify to predict class posterior probabilities (i.e., set 'FitPosterior', 1 in `fitcecoc`), then `BinaryLoss` is 'quadratic'

Otherwise, the default `BinaryLoss` is 'hamming'. To check the default value, use dot notation to display the `BinaryLoss` property of the trained model at the command line.

Example: `'BinaryLoss','binodeviance'`

Data Types: `char` | `function_handle`

### **Decoding — Decoding scheme**

`'lossweighted'` (default) | `'lossbased'`

Decoding scheme that aggregates the binary losses on page 32-4941, specified as the comma-separated pair consisting of 'Decoding' and 'lossweighted' or 'lossbased'.

Example: `'Decoding','lossbased'`



Data Types: char

### **Folds** — Fold indices for prediction

1:Mdl.KFold (default) | numeric vector of positive integers

Fold indices for prediction, specified as the comma-separated pair consisting of 'Folds' and a numeric vector of positive integers. The elements of Folds must range from 1 through Mdl.KFold.

The software only uses the folds specified in Folds for prediction.

Example: 'Folds', [1 4 10]

Data Types: single | double

### **Mode** — Edge aggregation level

'average' (default) | 'individual'

Edge meaning, specified as the comma-separated pair consisting of 'Mode' and 'average' or 'individual'.

This table describes the values.

Value	Description
'average'	edge is the scalar average over all folds.
'individual'	edge is a vector of length $k$ containing one edge per fold. $k$ is the number of folds.

Example: 'Mode', 'individual'

Data Types: char

### **Options** — Estimation options

[] (default) | structure array returned by statset

Estimation options, specified as the comma-separated pair consisting of 'Options' and a structure array returned by statset.

To invoke parallel computing:

- You need a Parallel Computing Toolbox license.
- Specify 'Options', statset('UseParallel',1).

**Verbose — Verbosity level**

0 (default) | 1

Verbosity level, specified as the comma-separated pair consisting of 'Verbose' and 0 or 1. Verbose controls the amount of diagnostic messages that the software displays in the Command Window.

If Verbose is 0, then the software does not display diagnostic messages. Otherwise, the software displays diagnostic messages.

Example: 'Verbose',1

Data Types: single | double

## Output Arguments

**edge — Classification edge**

numeric scalar | numeric row vector

Classification edge on page 32-3151, returned as a numeric scalar or numeric row vector.

If Mode is 'average', then edge is the average classification edge among all binary learners. Otherwise, edge is a 1-by-L numeric row vector containing the classification edge for each, respective binary learner, where L is the number of binary learners (`size(CVMdl.CodingMatrix,2)`).

Data Types: single | double

## Examples

**Estimate  $k$ -Fold Cross-Validation Edge of ECOC Models**

Load Fisher's iris data set.

```
load fisheriris
X = meas;
Y = categorical(species);
classOrder = unique(Y);
rng(1); % For reproducibility
```

Train an ECOC model using SVM binary classifiers and specify to cross validate. It is good practice to define the class order. Specify to standardize the predictors using an SVM template.

```
t = templateSVM('Standardize',1);
CVMdl = fitcecoc(X,Y,'CrossVal','on','Learners',t,'ClassNames',classOrder);
```

CVMdl is a ClassificationPartitionedModel model. By default, the software implements 10-fold cross validation. You can alter the number of folds using the 'KFold' name-value pair argument.

Estimate the average of the out-of-fold edges.

```
edge = kfoldEdge(CVMdl)
```

```
edge =
    0.4825
```

Alternatively, you can obtain the per-fold edges by specifying the name-value pair 'Mode','individual' in kfoldEdge.

### Display Individual Edges for Each Cross-Validation Fold

The classification edge is a relative measure of classifier quality. You can determine ill-performing folds by displaying the edges for each fold.

Load Fisher's iris data set.

```
load fisheriris
X = meas;
Y = categorical(species);
classOrder = unique(Y);
rng(1); % For reproducibility
```

Train an ECOC model using SVM binary classifiers and specify to use 8-fold cross validation. It is good practice to standardize the predictors and define the class order. Specify to standardize the predictors using an SVM template.

```
t = templateSVM('Standardize',1);
CVMdl = fitcecoc(X,Y,'KFold',8,'Learners',t,'ClassNames',classOrder);
```

Estimate the classification edge for each fold.

```
edges = kfoldEdge(CVMdl, 'Mode', 'individual')
```

```
edges =  
  
    0.4790  
    0.4872  
    0.4260  
    0.5301  
    0.5064  
    0.4575  
    0.4860  
    0.4687
```

The edges have similar magnitudes across folds. Ill-performing folds have low edges relative to the other folds.

You can return the classification edge for the entire model by specifying the well-performing folds using the 'Folds' name-value pair argument.

### Select ECOC Model Features by Comparing Cross-Validation Edges

The classifier edge measures the average of the classifier margins. One way to perform feature selection is to compare cross-validation edges from multiple models. Based solely on this criterion, the classifier with the highest edge is the best classifier.

Load Fisher's iris data set.

```
load fisheriris  
X = meas;  
Y = categorical(species);  
classOrder = unique(Y); % Class order  
rng(1); % For reproducibility
```

Define these two data sets:

- fullX contains all predictors.
- partX contains the petal dimensions.

```
fullX = X;
partX = X(:,3:4);
```

Train an ECOC model using SVM binary classifiers for each predictor set, and specify to cross validate. It is good practice to define the class order. Specify to standardize the predictors using an SVM template.

```
t = templateSVM('Standardize',1);
CVMdl = fitcecoc(fullX,Y,'CrossVal','on','Learners',t,...
    'ClassNames',classOrder);
PCVMdl = fitcecoc(partX,Y,'CrossVal','on','Learners',t,...
    'ClassNames',classOrder);
```

CVMdl and PCVMdl are ClassificationPartitionedECOC models. By default, the software implements 10-fold cross validation.

Estimate the test-sample edge for each classifier.

```
fullEdge = kfoldEdge(CVMdl)
partEdge = kfoldEdge(PCVMdl)
```

```
fullEdge =
    0.4825
```

```
partEdge =
    0.4951
```

PCVMdl achieves an edge that is similar to the more complex model CVMdl.

- “Quick Start Parallel Computing for Statistics and Machine Learning Toolbox” on page 30-2

## Definitions

### Classification Edge

The classification edge is the weighted mean of the classification margins.

One way to choose among multiple classifiers, e.g., to perform feature selection, is to choose the classifier that yields the highest edge.

## Classification Margin

The classification margins are, for each observation, the difference between the negative loss for the true class and maximal negative loss among the false classes. If the margins are on the same scale, then they serve as a classification confidence measure, i.e., among multiple classifiers, those that yield larger margins are better.

## Binary Loss

A binary loss is a function of the class and classification score that determines how well a binary learner classifies an observation into the class.

Let:

- $m_{kj}$  be element  $(k,j)$  of the coding design matrix  $M$  (i.e., the code corresponding to class  $k$  of binary learner  $j$ )
- $s_j$  be the score of binary learner  $j$  for an observation
- $g$  be the binary loss function
- $\hat{k}$  be the predicted class for the observation

In loss-based decoding [15], the class producing the minimum sum of the binary losses over binary learners determines the predicted class of an observation, that is,

$$\hat{k} = \operatorname{argmin}_k \sum_{j=1}^L |m_{kj}| g(m_{kj}, s_j).$$

In loss-weighted decoding [15], the class producing the minimum average of the binary losses over binary learners determines the predicted class of an observation, that is,

$$\hat{k} = \operatorname{argmin}_k \frac{\sum_{j=1}^L |m_{kj}| g(m_{kj}, s_j)}{\sum_{j=1}^L |m_{kj}|}.$$

Allwein et al. [2] suggest that loss-weighted decoding improves classification accuracy by keeping loss values for all classes in the same dynamic range.

This table summarizes the supported loss functions, where  $y_j$  is a class label for a particular binary learner (in the set  $\{-1, 1, 0\}$ ),  $s_j$  is the score for observation  $j$ , and  $g(y_j, s_j)$ .

Value	Description	Score Domain	$g(y_j, s_j)$
'binodeviance'	Binomial deviance	$(-\infty, \infty)$	$\log[1 + \exp(-2y_j s_j)] / [2\log(2)]$
'exponential'	Exponential	$(-\infty, \infty)$	$\exp(-y_j s_j) / 2$
'hamming'	Hamming	$[0, 1]$ or $(-\infty, \infty)$	$[1 - \text{sign}(y_j s_j)] / 2$
'hinge'	Hinge	$(-\infty, \infty)$	$\max(0, 1 - y_j s_j) / 2$
'linear'	Linear	$(-\infty, \infty)$	$(1 - y_j s_j) / 2$
'logit'	Logistic	$(-\infty, \infty)$	$\log[1 + \exp(-y_j s_j)] / [2\log(2)]$
'quadratic'	Quadratic	$[0, 1]$	$[1 - y_j(2s_j - 1)]^2 / 2$

The software normalizes the binary losses such that the loss is 0.5 when  $y_j = 0$ , and aggregates using the average of the binary learners [2].

Do not confuse the binary loss with the overall classification loss (specified by the `LossFun` name-value pair argument of `predict` and `loss`), e.g., classification error, which measures how well an ECOC classifier performs as a whole.

## See Also

`ClassificationECOC` | `ClassificationPartitionedModel` | `edge` | `fitcecoc` | `kfoldMargin` | `kfoldPredict` | `statset`

## Topics

“Quick Start Parallel Computing for Statistics and Machine Learning Toolbox” on page 30-2

“Reproducibility in Parallel Statistical Computations” on page 30-14

“Concepts of Parallel Computing in Statistics and Machine Learning Toolbox” on page 30-8

## kfoldEdge

**Class:** ClassificationPartitionedEnsemble

Classification edge for observations not used for training

### Syntax

```
E = kfoldEdge(obj)
E = kfoldEdge(obj, Name, Value)
```

### Description

`E = kfoldEdge(obj)` returns classification edge (average classification margin) obtained by cross-validated classification ensemble `obj`. For every fold, this method computes classification edge for in-fold observations using an ensemble trained on out-of-fold observations.

`E = kfoldEdge(obj, Name, Value)` calculates edge with additional options specified by one or more `Name, Value` pair arguments. You can specify several name-value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### Input Arguments

#### **ens**

Object of class `ClassificationPartitionedEnsemble`. Create `ens` with `fitensemble` along with one of the cross-validation options: `'crossval'`, `'kfold'`, `'holdout'`, `'leaveout'`, or `'cvpartition'`. Alternatively, create `ens` from a classification ensemble with `crossval`.

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single



quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### **folds**

Indices of folds ranging from 1 to `ens.KFold`. Use only these folds for predictions.

**Default:** `1:ens.KFold`

### **mode**

Character vector representing the meaning of the output `edge`:

- 'average' — `edge` is a scalar value, the average over all folds.
- 'individual' — `edge` is a vector of length `ens.KFold` with one element per fold.
- 'cumulative' — `edge` is a vector of length `min(ens.NTrainedPerFold)` in which element `J` is obtained by averaging values across all folds for weak learners `1:J` in each fold.

**Default:** 'average'

## Output Arguments

### **E**

The average classification margin. `E` is a scalar or vector, depending on the setting of the `mode` name-value pair.

## Examples

### **Compute K-Fold Edge of Held-Out Observations**

Compute the k-fold edge for an ensemble trained on the Fisher iris data.

Load the sample data set.

```
load fisheriris
```

Train an ensemble of 100 boosted classification trees.

```
t = templateTree('MaxNumSplits',1); % Weak learner template tree object
ens = fitcensemble(meas,species,'Learners',t);
```

Create a cross-validated ensemble from `ens` and find the classification edge.

```
rng(10,'twister') % For reproducibility
cvens = crossval(ens);
E = kfoldEdge(cvens)

E = 3.2066
```

## Definitions

### Edge

The edge is the weighted mean value of the classification margin. The weights are the class probabilities in `obj.Prior`.

### Margin

The classification margin is the difference between the classification score for the true class and maximal classification score for the false classes. Margin is a column vector with the same number of rows as in the matrix `obj.X`.

### Score (ensemble)

For ensembles, a classification score represents the confidence of a classification into a class. The higher the score, the higher the confidence.

Different ensemble algorithms have different definitions for their scores. Furthermore, the range of scores depends on ensemble type. For example:

- AdaBoostM1 scores range from  $-\infty$  to  $\infty$ .
- Bag scores range from 0 to 1.

## See Also

`crossval` | `kfoldLoss` | `kfoldMargin` | `kfoldPredict` | `kfoldfun`

## kfoldEdge

**Class:** ClassificationPartitionedLinear

Classification edge for observations not used for training

## Syntax

```
e = kfoldEdge(CVMdl)
e = kfoldEdge(CVMdl, Name, Value)
```

## Description

`e = kfoldEdge(CVMdl)` returns the cross-validated classification edges on page 32-3166 obtained by the cross-validated, binary, linear classification model `CVMdl`. That is, for every fold, `kfoldEdge` estimates the classification edge for observations that it holds out when it trains using all other observations.

`e` contains a classification edge for each regularization strength in the linear classification models that comprise `CVMdl`.

`e = kfoldEdge(CVMdl, Name, Value)` uses additional options specified by one or more `Name, Value` pair arguments. For example, indicate which folds to use for the edge calculation.

## Input Arguments

**`CVMdl` — Cross-validated, binary, linear classification model**

ClassificationPartitionedLinear model object

Cross-validated, binary, linear classification model, specified as a ClassificationPartitionedLinear model object. You can create a ClassificationPartitionedLinear model using `fitclinear` and specifying any one of the cross-validation, name-value pair arguments, for example, `CrossVal`.

To obtain estimates, `ClassificationPartitionedLinear.kfoldEdge` applies the same data used to cross-validate the linear classification model ( $X$  and  $Y$ ).

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### **Folds** — Fold indices to use for classification-score prediction

`1:CVMDL.KFold` (default) | numeric vector of positive integers

Fold indices to use for classification-score prediction, specified as the comma-separated pair consisting of 'Folds' and a numeric vector of positive integers. The elements of `Folds` must range from 1 through `CVMDL.KFold`.

Example: 'Folds', [1 4 10]

Data Types: single | double

### **Mode** — Edge aggregation level

'average' (default) | 'individual'

Edge aggregation level, specified as the comma-separated pair consisting of 'Mode' and 'average' or 'individual'.

Value	Description
'average'	Returns classification edges averaged over all folds
'individual'	Returns classification edges for each fold

Example: 'Mode', 'individual'

Data Types: char

## Output Arguments

### **e** — Cross-validated classification edges

numeric scalar | numeric vector | numeric matrix

Cross-validated classification edges on page 32-3166, returned as a numeric scalar, vector, or matrix.

Let  $L$  be the number of regularization strengths in the cross-validated models (that is,  $L$  is `numel(CVMdl.Trained{1}.Lambda)`) and  $F$  be the number of folds (stored in `CVMdl.KFold`).

- If `Mode` is `'average'`, then  $e$  is a 1-by- $L$  vector.  $e(j)$  is the average classification edge over all folds of the cross-validated model that uses regularization strength  $j$ .
- Otherwise,  $e$  is an  $F$ -by- $L$  matrix.  $e(i, j)$  is the classification edge for fold  $i$  of the cross-validated model that uses regularization strength  $j$ .

To estimate  $e$ , `kfoldEdge` uses the data that created `CVMdl` (see  $X$  and  $Y$ ).

## Examples

### Estimate $k$ -Fold Cross-Validation Edge

Load the NLP data set.

```
load nlpdata
```

$X$  is a sparse matrix of predictor data, and  $Y$  is a categorical vector of class labels. There are more than two classes in the data.

The models should identify whether the word counts in a web page are from the Statistics and Machine Learning Toolbox™ documentation. So, identify the labels that correspond to the Statistics and Machine Learning Toolbox™ documentation web pages.

```
Ystats = Y == 'stats';
```

Cross-validate a binary, linear classification model that can identify whether the word counts in a documentation web page are from the Statistics and Machine Learning Toolbox™ documentation.

```
rng(1); % For reproducibility  
CVMdl = fitlinear(X, Ystats, 'CrossVal', 'on');
```

CVMdl is a ClassificationPartitionedLinear model. By default, the software implements 10-fold cross validation. You can alter the number of folds using the 'Kfold' name-value pair argument.

Estimate the average of the out-of-fold edges.

```
e = kfoldEdge(CVMdl)
```

```
e =
```

```
8.1243
```

Alternatively, you can obtain the per-fold edges by specifying the name-value pair 'Mode', 'individual' in kfoldEdge.

### Feature Selection Using *k*-fold Edges

One way to perform feature selection is to compare *k*-fold edges from multiple models. Based solely on this criterion, the classifier with the highest edge is the best classifier.

Load the NLP data set. Preprocess the data as in “Estimate *k*-Fold Cross-Validation Edge” on page 32-3160.

```
load nlpdata
Ystats = Y == 'stats';
X = X';
```

Create these two data sets:

- fullX contains all predictors.
- partX contains 1/2 of the predictors chosen at random.

```
rng(1); % For reproducibility
p = size(X,1); % Number of predictors
halfPredIdx = randsample(p,ceil(0.5*p));
fullX = X;
partX = X(halfPredIdx,:);
```

Cross-validate two binary, linear classification models: one that uses the all of the predictors and one that uses half of the predictors. Optimize the objective function using SpaRSA, and indicate that observations correspond to columns.

```
CVMD1 = fitlinear(fullX,Ystats,'CrossVal','on','Solver','sparsa',...  
    'ObservationsIn','columns');  
PCVMD1 = fitlinear(partX,Ystats,'CrossVal','on','Solver','sparsa',...  
    'ObservationsIn','columns');
```

CVMD1 and PCVMD1 are ClassificationPartitionedLinear models.

Estimate the  $k$ -fold edge for each classifier.

```
fullEdge = kfoldEdge(CVMD1)  
partEdge = kfoldEdge(PCVMD1)
```

```
fullEdge =
```

```
    16.5629
```

```
partEdge =
```

```
    13.9030
```

Based on the  $k$ -fold edges, the classifier that uses all of the predictors is the better model.

### Find Good Lasso Penalty Using $k$ -fold Edge

To determine a good lasso-penalty strength for a linear classification model that uses a logistic regression learner, compare  $k$ -fold edges.

Load the NLP data set. Preprocess the data as in “Estimate  $k$ -Fold Cross-Validation Edge” on page 32-3160.

```
load nlpdata  
Ystats = Y == 'stats';  
X = X';
```

Create a set of 11 logarithmically-spaced regularization strengths from  $10^{-8}$  through  $10^1$ .



```
Lambda = logspace(-8,1,11);
```

Cross-validate a binary, linear classification model using 5-fold cross-validation and that uses each of the regularization strengths. Optimize the objective function using SpaRSA. Lower the tolerance on the gradient of the objective function to  $1e-8$ .

```
rng(10); % For reproducibility
CVMdl = fitclinear(X,Ystats,'ObservationsIn','columns','KFold',5,...
    'Learner','logistic','Solver','sparsa','Regularization','lasso',...
    'Lambda',Lambda,'GradientTolerance',1e-8)
```

```
CVMdl =
```

```
classreg.learning.partition.ClassificationPartitionedLinear
  CrossValidatedModel: 'Linear'
    ResponseName: 'Y'
  NumObservations: 31572
        KFold: 5
    Partition: [1x1 cvpartition]
    ClassNames: [0 1]
  ScoreTransform: 'none'
```

CVMdl is a ClassificationPartitionedLinear model. Because fitclinear implements 5-fold cross-validation, CVMdl contains 5 ClassificationLinear models that the software trains on each fold.

Estimate the edges for each fold and regularization strength.

```
eFolds = kfoldEdge(CVMdl,'Mode','individual')
```

```
eFolds =
```

```
Columns 1 through 7
```

0.9958	0.9958	0.9958	0.9958	0.9958	0.9923	0.9770
0.9990	0.9990	0.9990	0.9990	0.9990	0.9937	0.9780
0.9992	0.9992	0.9992	0.9992	0.9992	0.9943	0.9781
0.9974	0.9974	0.9974	0.9974	0.9974	0.9931	0.9772
0.9977	0.9977	0.9977	0.9977	0.9977	0.9942	0.9781

```
Columns 8 through 11
```

```
0.9167    0.8444    0.8127    0.8127
0.9181    0.8262    0.8128    0.8128
0.9141    0.8253    0.8128    0.8128
0.9203    0.8460    0.8130    0.8130
0.9199    0.8415    0.8127    0.8127
```

`eFolds` is a 5-by-11 matrix of edges. Rows correspond to folds and columns correspond to regularization strengths in `Lambda`. You can use `eFolds` to identify ill-performing folds, that is, unusually low edges.

Estimate the average edge over all folds for each regularization strength.

```
e = kfoldEdge(CVMdl)
```

```
e =
```

```
Columns 1 through 7
```

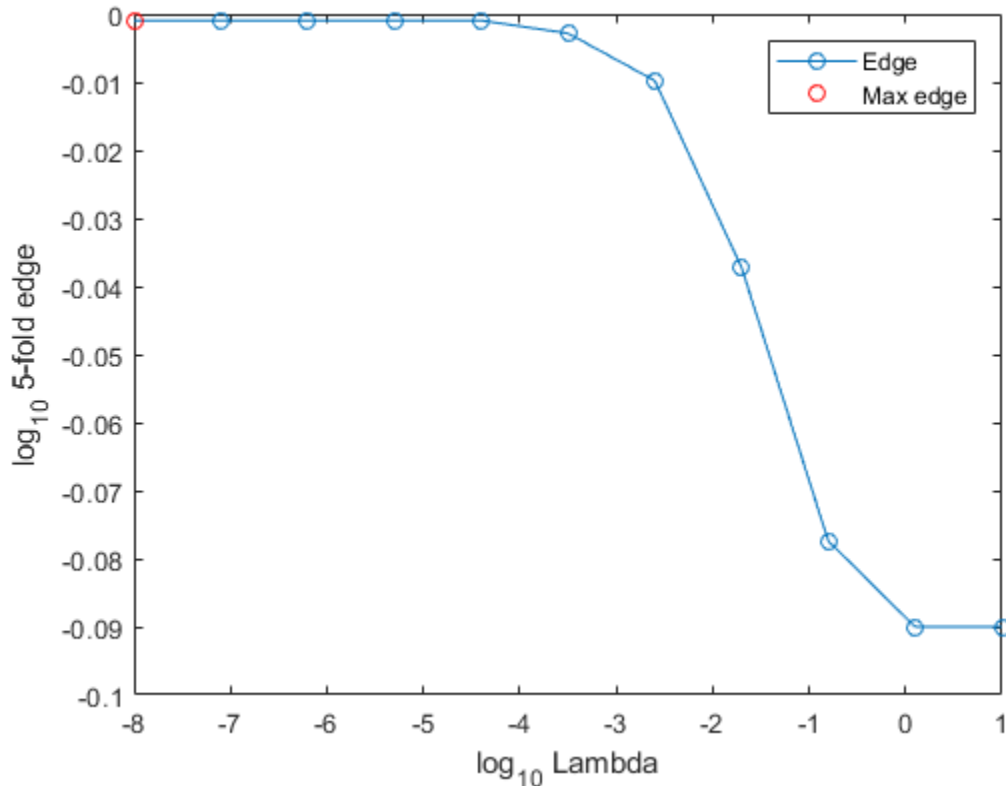
```
0.9978    0.9978    0.9978    0.9978    0.9978    0.9935    0.9777
```

```
Columns 8 through 11
```

```
0.9178    0.8367    0.8128    0.8128
```

Determine how well the models generalize by plotting the averages of the 5-fold edge for each regularization strength. Identify the regularization strength that maximizes the 5-fold edge over the grid.

```
figure;
plot(log10(Lambda), log10(e), '-o')
[~, maxEIdx] = max(e);
maxLambda = Lambda(maxEIdx);
hold on
plot(log10(maxLambda), log10(e(maxEIdx)), 'ro');
ylabel('log_{10} 5-fold edge')
xlabel('log_{10} Lambda')
legend('Edge', 'Max edge')
hold off
```



Several values of Lambda yield similarly high edges. Higher values of lambda lead to predictor variable sparsity, which is a good quality of a classifier.

Choose the regularization strength that occurs just before the edge starts decreasing.

```
LambdaFinal = Lambda(5);
```

Train a linear classification model using the entire data set and specify the regularization strength LambdaFinal.

```
MdlFinal = fitclinear(X,Ystats,'ObservationsIn','columns',...
    'Learner','logistic','Solver','sparsa','Regularization','lasso',...
    'Lambda',LambdaFinal);
```

To estimate labels for new observations, pass `MdlFinal` and the new data to `predict`.

## Definitions

### Classification Edge

The classification edge is the weighted mean of the classification margins.

One way to choose among multiple classifiers, e.g., to perform feature selection, is to choose the classifier that yields the highest edge.

### Classification Margin

The classification margins for binary classification are, for each observation, the difference between the classification score for the true class and the classification score for the false class.

The software defines the classification margin for binary classification as

$$m = 2yf(x).$$

$x$  is an observation. If the true label of  $x$  is the positive class, then  $y$  is 1, and  $-1$  otherwise.  $f(x)$  is the positive-class classification score for the observation  $x$ . The literature commonly defines the margin as  $m = yf(x)$ .

If the margins are on the same scale, then they serve as a classification confidence measure, i.e., among multiple classifiers, those that yield larger margins are better.

### Classification Score

For linear classification models, the raw classification score for classifying the observation  $x$ , a row vector, into the positive class is defined by

$$f_j(x) = x\beta_j + b_j.$$

For the model with regularization strength  $j$ ,  $\beta_j$  is the estimated column vector of coefficients (the model property `Beta(:, j)`) and  $b_j$  is the estimated, scalar bias (the model property `Bias(j)`).

The raw classification score for classifying  $x$  into the negative class is  $-f(x)$ . The software classifies observations into the class that yields the positive score.

If the linear classification model consists of logistic regression learners, then the software applies the 'logit' score transformation to the raw classification scores (see `ScoreTransform`).

## See Also

`ClassificationLinear` | `ClassificationPartitionedLinear` | `edge` | `kfoldMargin` | `kfoldPredict`

**Introduced in R2016a**

## kfoldEdge

**Class:** ClassificationPartitionedLinearECOC

Classification edge for observations not used for training

## Syntax

```
e = kfoldEdge(CVMdl)
e = kfoldEdge(CVMdl, Name, Value)
```

## Description

`e = kfoldEdge(CVMdl)` returns the cross-validated classification edges on page 32-3179 obtained by the cross-validated, error-correcting output codes (ECOC) model composed of linear classification models `CVMdl`. That is, for every fold, `kfoldEdge` estimates the classification edge for observations that it holds out when it trains using all other observations.

`e` contains a classification edge for each regularization strength in the linear classification models that comprise `CVMdl`.

`e = kfoldEdge(CVMdl, Name, Value)` uses additional options specified by one or more `Name, Value` pair arguments. For example, specify a decoding scheme, which folds to use for the edge calculation, or verbosity level.

## Input Arguments

**CVMdl** — Cross-validated, ECOC model composed of linear classification models

ClassificationPartitionedLinearECOC model object

Cross-validated, ECOC model composed of linear classification models, specified as a ClassificationPartitionedLinearECOC model object. You can create a ClassificationPartitionedLinearECOC model using `fitcecoc` and by:

- 1 Specifying any one of the cross-validation, name-value pair arguments, for example, `CrossVal`
- 2 Setting the name-value pair argument `Learners` to `'linear'` or a linear classification model template returned by `templateLinear`

To obtain estimates, `ClassificationPartitionedLinearECOC.kfoldEdge` applies the same data used to cross-validate the ECOC model ( $X$  and  $Y$ ).

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### BinaryLoss — Binary learner loss function

`'hamming'` | `'linear'` | `'logit'` | `'exponential'` | `'binodeviance'` | `'hinge'`  
| `'quadratic'` | function handle

Binary learner loss function, specified as the comma-separated pair consisting of `'BinaryLoss'` and a built-in, loss-function name or function handle.

- This table contains names and descriptions of the built-in functions, where  $y_j$  is a class label for a particular binary learner (in the set  $\{-1, 1, 0\}$ ),  $s_j$  is the score for observation  $j$ , and  $g(y_j, s_j)$  is the binary loss formula.

Value	Description	Score Domain	$g(y_j, s_j)$
<code>'binodeviance'</code>	Binomial deviance	$(-\infty, \infty)$	$\log[1 + \exp(-2y_j s_j)] / [2\log(2)]$
<code>'exponential'</code>	Exponential	$(-\infty, \infty)$	$\exp(-y_j s_j) / 2$
<code>'hamming'</code>	Hamming	$[0, 1]$ or $(-\infty, \infty)$	$[1 - \text{sign}(y_j s_j)] / 2$
<code>'hinge'</code>	Hinge	$(-\infty, \infty)$	$\max(0, 1 - y_j s_j) / 2$
<code>'linear'</code>	Linear	$(-\infty, \infty)$	$(1 - y_j s_j) / 2$
<code>'logit'</code>	Logistic	$(-\infty, \infty)$	$\log[1 + \exp(-y_j s_j)] / [2\log(2)]$
<code>'quadratic'</code>	Quadratic	$[0, 1]$	$[1 - y_j(2s_j - 1)]^2 / 2$

The software normalizes the binary losses such that the loss is 0.5 when  $y_j = 0$ . Also, the software calculates the mean binary loss for each class.

- For a custom binary loss function, e.g., `customFunction`, specify its function handle `'BinaryLoss', @customFunction`.

`customFunction` should have this form

```
bLoss = customFunction(M,s)
```

where:

- `M` is the  $K$ -by- $L$  coding matrix stored in `Mdl.CodingMatrix`.
- `s` is the 1-by- $L$  row vector of classification scores.
- `bLoss` is the classification loss. This scalar aggregates the binary losses for every learner in a particular class. For example, you can use the mean binary loss to aggregate the loss over the learners for each class.
- $K$  is the number of classes.
- $L$  is the number of binary learners.

For an example of passing a custom binary loss function, see “Predict Test-Sample Labels of ECOC Models Using Custom Binary Loss Function” on page 32-4933.

By default, if all binary learners are linear classification models using:

- SVM, then `BinaryLoss` is `'hinge'`
- Logistic regression, then `BinaryLoss` is `'quadratic'`

Example: `'BinaryLoss', 'binodeviance'`

Data Types: `char` | `function_handle`

#### **Decoding — Decoding scheme**

`'lossweighted'` (default) | `'lossbased'`

Decoding scheme that aggregates the binary losses on page 32-4941, specified as the comma-separated pair consisting of `'Decoding'` and `'lossweighted'` or `'lossbased'`.

Example: `'Decoding', 'lossbased'`

Data Types: `char`

#### **Folds — Fold indices to use for classification-score prediction**

`1:CVMdl.KFold` (default) | numeric vector of positive integers



Fold indices to use for classification-score prediction, specified as the comma-separated pair consisting of 'Folds' and a numeric vector of positive integers. The elements of Folds must range from 1 through `CVMDL.KFold`.

Example: 'Folds', [1 4 10]

Data Types: single | double

#### Mode — Edge aggregation level

'average' (default) | 'individual'

Edge aggregation level, specified as the comma-separated pair consisting of 'Mode' and 'average' or 'individual'.

Value	Description
'average'	Returns classification edges averaged over all folds
'individual'	Returns classification edges for each fold

Example: 'Mode', 'individual'

Data Types: char

#### Options — Estimation options

[] (default) | structure array returned by `statset`

Estimation options, specified as the comma-separated pair consisting of 'Options' and a structure array returned by `statset`.

To invoke parallel computing:

- You need a Parallel Computing Toolbox license.
- Specify 'Options', `statset('UseParallel',1)`.

#### Verbose — Verbosity level

0 (default) | 1

Verbosity level, specified as the comma-separated pair consisting of 'Verbose' and 0 or 1. Verbose controls the amount of diagnostic messages that the software displays in the Command Window.

If `Verbose` is 0, then the software does not display diagnostic messages. Otherwise, the software displays diagnostic messages.

Example: `'Verbose', 1`

Data Types: `single` | `double`

## Output Arguments

### **e** — Cross-validated classification edges

numeric scalar | numeric vector | numeric matrix

Cross-validated classification edges on page 32-3179, returned as a numeric scalar, vector, or matrix.

Let  $L$  be the number of regularization strengths in the cross-validated models (that is,  $L$  is `numel(CVMdl.Trained{1}.BinaryLearners{1}.Lambda)`) and  $F$  be the number of folds (stored in `CVMdl.KFold`).

- If `Mode` is `'average'`, then  $e$  is a 1-by- $L$  vector.  $e(j)$  is the average classification edge over all folds of the cross-validated model that uses regularization strength  $j$ .
- Otherwise,  $e$  is a  $F$ -by- $L$  matrix.  $e(i, j)$  is the classification edge for fold  $i$  of the cross-validated model that uses regularization strength  $j$ .

## Examples

### Estimate $k$ -Fold Cross-Validation Edge

Load the NLP data set.

```
load nlpdata
```

$X$  is a sparse matrix of predictor data, and  $Y$  is a categorical vector of class labels.

For simplicity, use the label `'others'` for all observations in  $Y$  that are not `'simulink'`, `'dsp'`, or `'comm'`.

```
Y(~(ismember(Y, {'simulink', 'dsp', 'comm'}))) = 'others';
```

Cross-validate a multiclass, linear classification model.

```
rng(1); % For reproducibility
CVMdl = fitcecoc(X,Y, 'Learner', 'linear', 'CrossVal', 'on');
```

CVMdl is a ClassificationPartitionedLinearECOC model. By default, the software implements 10-fold cross validation. You can alter the number of folds using the 'KFold' name-value pair argument.

Estimate the average of the out-of-fold edges.

```
e = kfoldEdge(CVMdl)
```

```
e =
    0.7232
```

Alternatively, you can obtain the per-fold edges by specifying the name-value pair 'Mode', 'individual' in kfoldEdge.

## Feature Selection Using $k$ -fold Edges

One way to perform feature selection is to compare  $k$ -fold edges from multiple models. Based solely on this criterion, the classifier with the highest edge is the best classifier.

Load the NLP data set. Preprocess the data as in “Estimate  $k$ -Fold Cross-Validation Edge” on page 32-3172, and orient the predictor data so that observations correspond to columns.

```
load nlpdata
Y(~(ismember(Y, {'simulink', 'dsp', 'comm'}))) = 'others';
X = X';
```

Create these two data sets:

- fullX contains all predictors.
- partX contains a 1/2 of the predictors chosen at random.

```
rng(1); % For reproducibility
p = size(X,1); % Number of predictors
```

```
halfPredIdx = randsample(p, ceil(0.5*p));
fullX = X;
partX = X(halfPredIdx, :);
```

Create a linear classification model template that specifies to optimize the objective function using SpARSA.

```
t = templateLinear('Solver', 'sparsa');
```

Cross-validate two ECOC models composed of binary, linear classification models: one that uses the all of the predictors and one that uses half of the predictors. Indicate that observations correspond to columns.

```
CVMD1 = fitcecoc(fullX, Y, 'Learners', t, 'CrossVal', 'on', ...
    'ObservationsIn', 'columns');
PCVMD1 = fitcecoc(partX, Y, 'Learners', t, 'CrossVal', 'on', ...
    'ObservationsIn', 'columns');
```

CVMD1 and PCVMD1 are ClassificationPartitionedLinearECOC models.

Estimate the  $k$ -fold edge for each classifier.

```
fullEdge = kfoldEdge(CVMD1)
partEdge = kfoldEdge(PCVMD1)
```

```
fullEdge =
    0.3090
```

```
partEdge =
    0.2617
```

Based on the  $k$ -fold edges, the classifier that uses all of the predictors is the better model.

### Find Good Lasso Penalty Using $k$ -fold Edge

To determine a good lasso-penalty strength for a linear classification model that uses a logistic regression learner, compare  $k$ -fold edges.

Load the NLP data set. Preprocess the data as in “Feature Selection Using k-fold Edges” on page 32-3173.

```
load nlpdata
Y(~(ismember(Y,{'simulink','dsp','comm'}))) = 'others';
X = X';
```

Create a set of 8 logarithmically-spaced regularization strengths from  $10^{-8}$  through  $10^1$ .

```
Lambda = logspace(-8,1,8);
```

Create a linear classification model template that specifies to use logistic regression with a lasso penalty, use each of the regularization strengths, optimize the objective function using SpaRSA, and reduce the tolerance on the gradient of the objective function to  $1e-8$ .

```
t = templateLinear('Learner','logistic','Solver','sparsa',...
    'Regularization','lasso','Lambda',Lambda,'GradientTolerance',1e-8);
```

Cross-validate an ECOC model composed of binary, linear classification models using 5-fold cross-validation and that

```
rng(10); % For reproducibility
CVMdl = fitcecoc(X,Y,'Learners',t,'ObservationsIn','columns','KFold',5)
```

```
CVMdl =
```

```
classreg.learning.partition.ClassificationPartitionedLinearECOC
  CrossValidatedModel: 'Linear'
    ResponseName: 'Y'
  NumObservations: 31572
    KFold: 5
    Partition: [1x1 cvpartition]
  ClassNames: [comm dsp simulink others]
  ScoreTransform: 'none'
```

CVMdl is a ClassificationPartitionedLinearECOC model.

Estimate the edges for each fold and regularization strength.

```
eFolds = kfoldEdge(CVMdl,'Mode','individual')
```

```
eFolds =
```

```
Columns 1 through 7
```

```
0.5558    0.5554    0.5558    0.5534    0.4956    0.2934    0.1039
0.5248    0.5263    0.5268    0.5274    0.4800    0.2949    0.1045
0.5263    0.5274    0.5280    0.5274    0.4740    0.2891    0.1042
0.5429    0.5445    0.5446    0.5416    0.4878    0.2911    0.1015
0.5505    0.5559    0.5584    0.5578    0.4959    0.2952    0.1021
```

```
Column 8
```

```
0.0854
0.0867
0.0865
0.0853
0.0849
```

`eFolds` is a 5-by-8 matrix of edges. Rows correspond to folds and columns correspond to regularization strengths in `Lambda`. You can use `eFolds` to identify ill-performing folds, that is, unusually low edges.

Estimate the average edge over all folds for each regularization strength.

```
e = kfoldEdge(CVMdl)
```

```
e =
```

```
Columns 1 through 7
```

```
0.5401    0.5419    0.5427    0.5415    0.4867    0.2927    0.1032
```

```
Column 8
```

```
0.0858
```

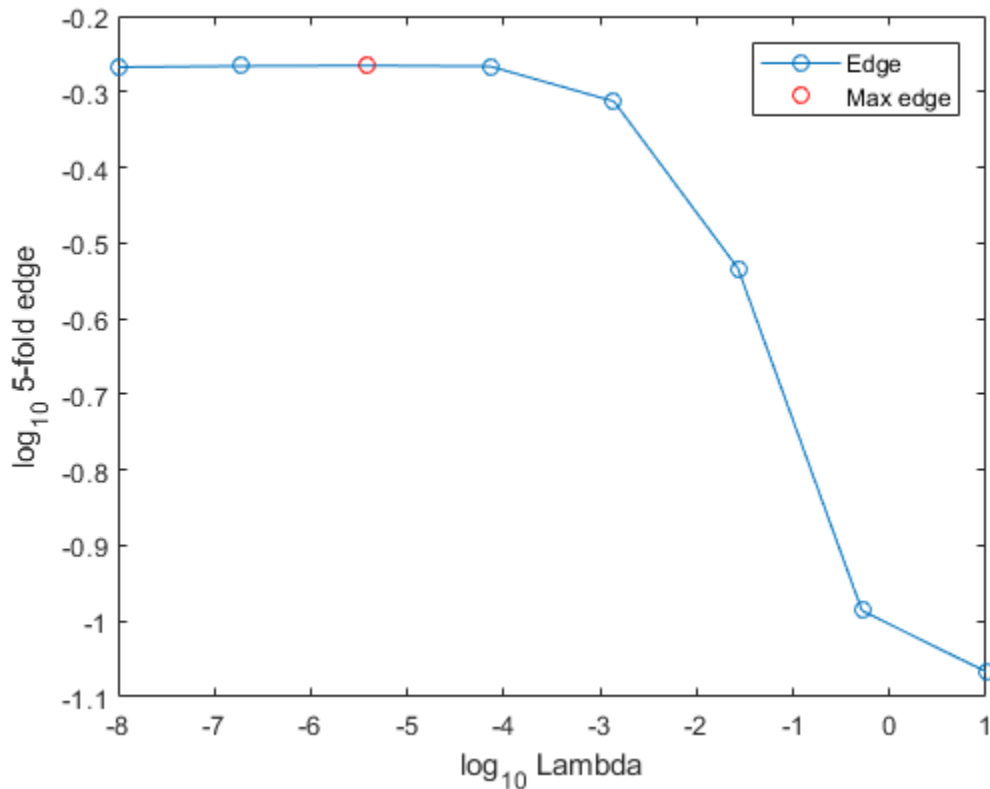
Determine how well the models generalize by plotting the averages of the 5-fold edge for each regularization strength. Identify the regularization strength that maximizes the 5-fold edge over the grid.

```
figure;
plot(log10(Lambda), log10(e), '-o')
```

```

[~, maxEIdx] = max(e);
maxLambda = Lambda(maxEIdx);
hold on
plot(log10(maxLambda), log10(e(maxEIdx)), 'ro');
ylabel('log_{10} 5-fold edge')
xlabel('log_{10} Lambda')
legend('Edge', 'Max edge')
hold off

```



Several values of Lambda yield similarly high edges. Greater regularization strength values lead to predictor variable sparsity, which is a good quality of a classifier.

Choose the regularization strength that occurs just before the edge starts decreasing.

```
LambdaFinal = Lambda(4);
```

Train an ECOC model composed of linear classification model using the entire data set and specify the regularization strength `LambdaFinal`.

```
t = templateLinear('Learner','logistic','Solver','sparsa',...  
    'Regularization','lasso','Lambda',LambdaFinal,'GradientTolerance',1e-8);  
MdlFinal = fitcecoc(X,Y,'Learners',t,'ObservationsIn','columns');
```

To estimate labels for new observations, pass `MdlFinal` and the new data to `predict`.

- “Quick Start Parallel Computing for Statistics and Machine Learning Toolbox” on page 30-2

## Definitions

### Binary Loss

A binary loss is a function of the class and classification score that determines how well a binary learner classifies an observation into the class.

Let:

- $m_{kj}$  be element  $(k,j)$  of the coding design matrix  $M$  (i.e., the code corresponding to class  $k$  of binary learner  $j$ )
- $s_j$  be the score of binary learner  $j$  for an observation
- $g$  be the binary loss function
- $\hat{k}$  be the predicted class for the observation

In loss-based decoding [15], the class producing the minimum sum of the binary losses over binary learners determines the predicted class of an observation, that is,

$$\hat{k} = \operatorname{argmin}_k \sum_{j=1}^L |m_{kj}| g(m_{kj}, s_j).$$

In loss-weighted decoding [15], the class producing the minimum average of the binary losses over binary learners determines the predicted class of an observation, that is,



$$\hat{k} = \operatorname{argmin}_k \frac{\sum_{j=1}^L |m_{kj}| g(m_{kj}, s_j)}{\sum_{j=1}^L |m_{kj}|}.$$

Allwein et al. [2] suggest that loss-weighted decoding improves classification accuracy by keeping loss values for all classes in the same dynamic range.

This table summarizes the supported loss functions, where  $y_j$  is a class label for a particular binary learner (in the set  $\{-1, 1, 0\}$ ),  $s_j$  is the score for observation  $j$ , and  $g(y_j, s_j)$ .

Value	Description	Score Domain	$g(y_j, s_j)$
'binodeviance'	Binomial deviance	$(-\infty, \infty)$	$\log[1 + \exp(-2y_j s_j)] / [2\log(2)]$
'exponential'	Exponential	$(-\infty, \infty)$	$\exp(-y_j s_j) / 2$
'hamming'	Hamming	$[0, 1]$ or $(-\infty, \infty)$	$[1 - \operatorname{sign}(y_j s_j)] / 2$
'hinge'	Hinge	$(-\infty, \infty)$	$\max(0, 1 - y_j s_j) / 2$
'linear'	Linear	$(-\infty, \infty)$	$(1 - y_j s_j) / 2$
'logit'	Logistic	$(-\infty, \infty)$	$\log[1 + \exp(-y_j s_j)] / [2\log(2)]$
'quadratic'	Quadratic	$[0, 1]$	$[1 - y_j(2s_j - 1)]^2 / 2$

The software normalizes the binary losses such that the loss is 0.5 when  $y_j = 0$ , and aggregates using the average of the binary learners [2].

Do not confuse the binary loss with the overall classification loss (specified by the `LossFun` name-value pair argument of `predict` and `loss`), e.g., classification error, which measures how well an ECOC classifier performs as a whole.

## Classification Edge

The classification edge is the weighted mean of the classification margins.

One way to choose among multiple classifiers, e.g., to perform feature selection, is to choose the classifier that yields the highest edge.

## Classification Margin

The classification margins are, for each observation, the difference between the negative loss for the true class and maximal negative loss among the false classes. If the margins are on the same scale, then they serve as a classification confidence measure, i.e., among multiple classifiers, those that yield larger margins are better.

## References

- [1] Allwein, E., R. Schapire, and Y. Singer. “Reducing multiclass to binary: A unifying approach for margin classifiers.” *Journal of Machine Learning Research*. Vol. 1, 2000, pp. 113–141.
- [2] Escalera, S., O. Pujol, and P. Radeva. “On the decoding process in ternary error-correcting output codes.” *IEEE Transactions on Pattern Analysis and Machine Intelligence*. Vol. 32, Issue 7, 2010, pp. 120–134.
- [3] Escalera, S., O. Pujol, and P. Radeva. “Separability of ternary codes for sparse designs of error-correcting output codes.” *Pattern Recogn.* Vol. 30, Issue 3, 2009, pp. 285–297.

## See Also

`ClassificationECOC` | `ClassificationLinear` |  
`ClassificationPartitionedLinearECOC` | `edge` | `fitcecoc` | `kfoldMargin` |  
`kfoldPredict` | `statset`

## Topics

“Quick Start Parallel Computing for Statistics and Machine Learning Toolbox” on page 30-2

“Reproducibility in Parallel Statistical Computations” on page 30-14

“Concepts of Parallel Computing in Statistics and Machine Learning Toolbox” on page 30-8

**Introduced in R2016a**

# kfoldEdge

**Class:** ClassificationPartitionedModel

Classification edge for observations not used for training

## Syntax

```
E = kfoldEdge(obj)
E = kfoldEdge(obj, Name, Value)
```

## Description

`E = kfoldEdge(obj)` returns classification edge (average classification margin) obtained by cross-validated classification model `obj`. For every fold, this method computes classification edge for in-fold observations using an ensemble trained on out-of-fold observations.

`E = kfoldEdge(obj, Name, Value)` calculates edge with additional options specified by one or more `Name, Value` pair arguments. You can specify several name-value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

## Input Arguments

**obj**

Object of class `ClassificationPartitionedModel`.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

**fold**

Indices of folds ranging from 1 to `obj.KFold`. Use only these folds for predictions.

**Default:** `1:obj.KFold`

**mode**

Character vector representing the meaning of the output edge:

- 'average' — edge is a scalar value, the average over all folds.
- 'individual' — edge is a vector of length `obj.KFold` with one element per fold.

**Default:** 'average'

## Output Arguments

**E**

The average classification margin. `E` is a scalar or vector, depending on the setting of the `mode` name-value pair.

## Examples

**Estimate the  $k$ -fold Edge of a Classifier**

Compute the  $k$ -fold edge for a model trained on Fisher's iris data.

Load Fisher's iris data set.

```
load fisheriris
```

Train a classification tree classifier.

```
tree = fitctree(meas,species);
```

Cross validate the classifier using 10-fold cross validation.

```
cvtree = crossval(tree);
```

Compute the  $k$ -fold edge.

```
edge = kfoldEdge(cvtree)
```

```
edge =
```

```
0.8578
```

## Definitions

### Edge

The edge is the weighted mean value of the classification margin. The weights are class prior probabilities. If you supply additional weights, those weights are normalized to sum to the prior probabilities in the respective classes, and are then used to compute the weighted average.

### Margin

The classification margin is the difference between the classification score for the true class and maximal classification score for the false classes.

The classification margin is a column vector with the same number of rows as in the matrix  $X$ . A high value of margin indicates a more reliable prediction than a low value.

### Score

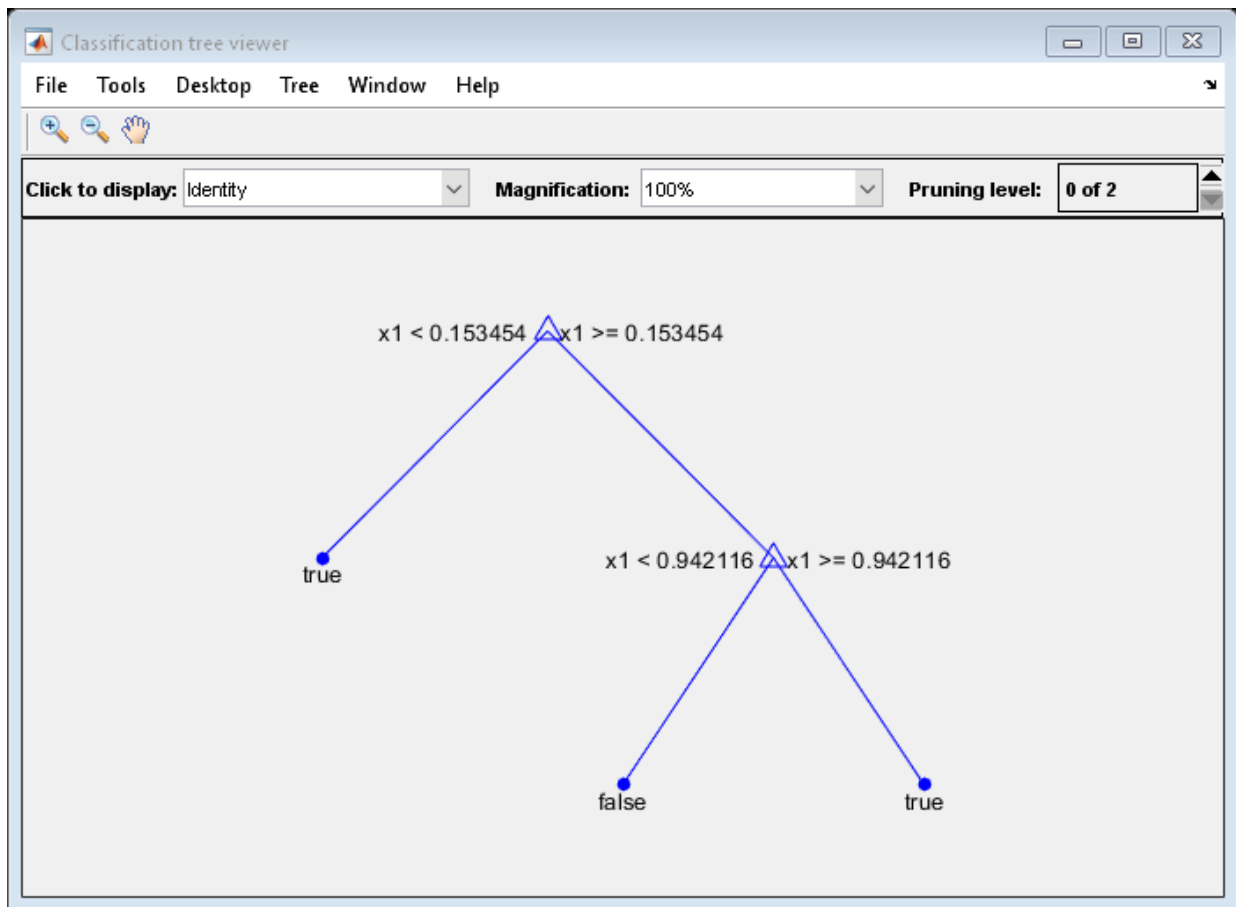
For discriminant analysis, the score of a classification is the posterior probability of the classification. For the definition of posterior probability in discriminant analysis, see “Posterior Probability” on page 20-8.

For trees, the score of a classification of a leaf node is the posterior probability of the classification at that node. The posterior probability of the classification at a node is the number of training sequences that lead to that node with the classification, divided by the number of training sequences that lead to that node.

For example, consider classifying a predictor  $X$  as `true` when  $X < 0.15$  or  $X > 0.95$ , and  $X$  is `false` otherwise.

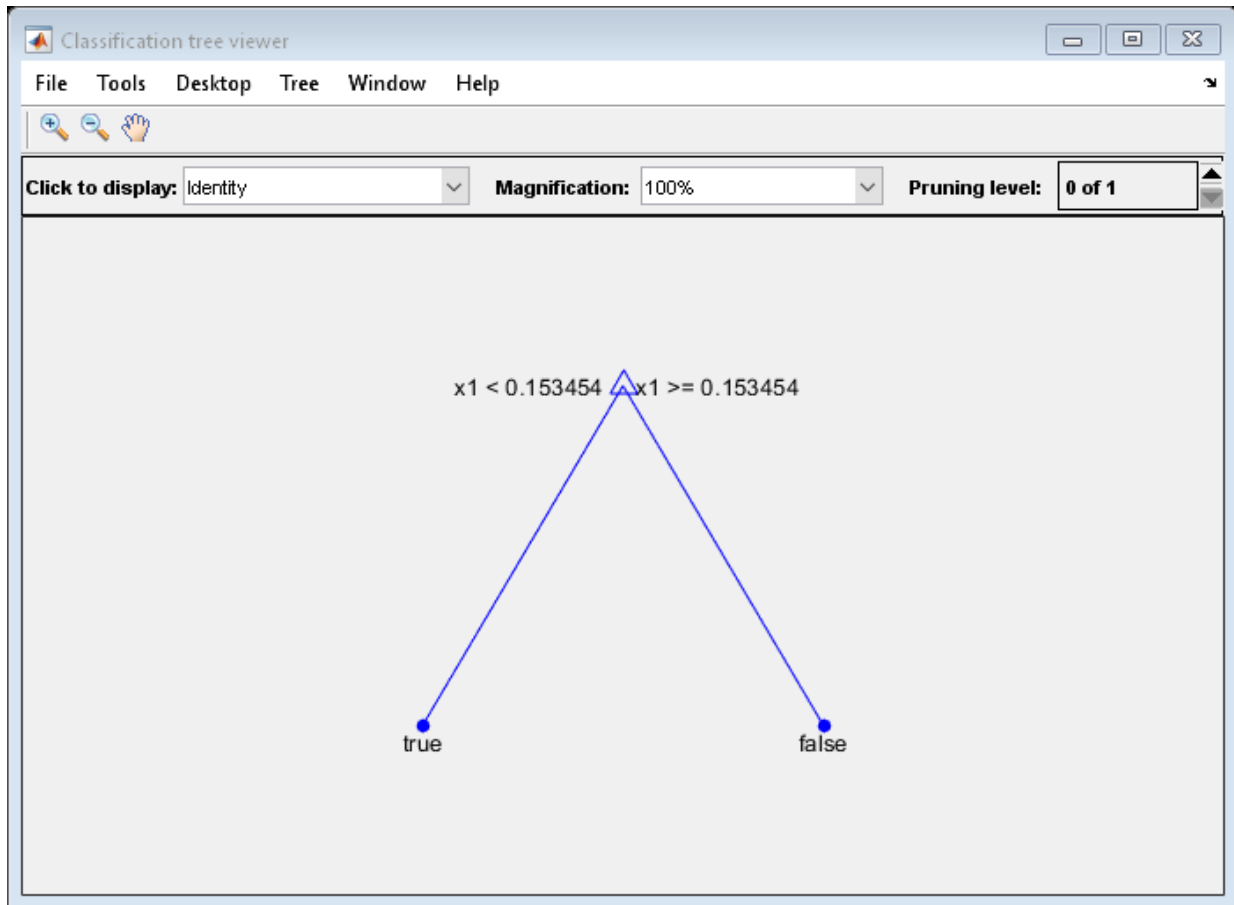
Generate 100 random points and classify them:

```
rng(0, 'twister') % for reproducibility
X = rand(100,1);
Y = (abs(X - .55) > .4);
tree = fitctree(X,Y);
view(tree, 'Mode', 'Graph')
```



Prune the tree:

```
tree1 = prune(tree, 'Level', 1);
view(tree1, 'Mode', 'Graph')
```



The pruned tree correctly classifies observations that are less than 0.15 as `true`. It also correctly classifies observations from .15 to .94 as `false`. However, it incorrectly classifies observations that are greater than .94 as `false`. Therefore, the score for observations that are greater than .15 should be about  $.05/.85=.06$  for `true`, and about  $.8/.85=.94$  for `false`.

Compute the prediction scores for the first 10 rows of  $X$ :

```
[~,score] = predict(tree1,X(1:10));  
[score X(1:10,:)]
```

```
ans =
```

```
    0.9059    0.0941    0.8147  
    0.9059    0.0941    0.9058  
         0    1.0000    0.1270  
    0.9059    0.0941    0.9134  
    0.9059    0.0941    0.6324  
         0    1.0000    0.0975  
    0.9059    0.0941    0.2785  
    0.9059    0.0941    0.5469  
    0.9059    0.0941    0.9575  
    0.9059    0.0941    0.9649
```

Indeed, every value of `X` (the right-most column) that is less than 0.15 has associated scores (the left and center columns) of 0 and 1, while the other values of `X` have associated scores of 0.91 and 0.09. The difference (score 0.09 instead of the expected .06) is due to a statistical fluctuation: there are 8 observations in `X` in the range (.95, 1) instead of the expected 5 observations.

## See Also

`ClassificationPartitionedEnsemble` | `ClassificationPartitionedModel` | `crossval` | `kfoldLoss` | `kfoldMargin` | `kfoldPredict` | `kfoldfun`



# kfoldfun

**Class:** ClassificationPartitionedECOC

Cross validate function

## Syntax

```
vals = kfoldfun(CVMdl, fun)
```

## Description

`vals = kfoldfun(CVMdl, fun)` cross validates the function `fun` by applying `fun` to the data stored in the cross-validated model `CVMdl`. You must pass `fun` as a function handle.

## Input Arguments

**CVMdl** — Cross-validated model

ClassificationPartitionedECOC model |  
ClassificationPartitionedEnsemble model |  
ClassificationPartitionedModel model

Cross-validated model, specified as a ClassificationPartitionedECOC model, ClassificationPartitionedEnsemble model, or a ClassificationPartitionedModel model.

**fun** — Cross-validated function

function handle

Cross-validated function, specified as a function handle. `fun` has the syntax

```
testvals = fun(CMP, Xtrain, Ytrain, Wtrain, Xtest, Ytest, Wtest)
```

- `CMP` is a compact model stored in one element of the `CVMdl.Trained` property.

- `Xtrain` is the training matrix of predictor values.
- `Ytrain` is the training array of response values.
- `Wtrain` are the training weights for observations.
- `Xtest` and `Ytest` are the test data, with associated weights `Wtest`.
- The returned value `testvals` needs the same size across all folds.

Data Types: `function_handle`

## Output Arguments

### **vals** — Cross-validation results

numeric matrix

Cross-validation results, returned as an numeric matrix. `vals` is the arrays of `testvals` output, concatenated vertically over all folds. For example, if `testvals` from every fold is a numeric vector of length `N`, `kfoldfun` returns a `KFold-by-N` numeric matrix with one row per fold.

Data Types: `double`

## Examples

### Estimate Classification Error Using a Custom Loss Function

Train an ECOC multiclass classifier, and then cross validate it using a custom  $k$ -fold loss function.

Load Fisher's iris data set.

```
load fisheriris
X = meas;
Y = categorical(species);
classOrder = unique(Y); % Class order
rng(1); % For reproducibility
```

Train an ECOC model using SVM binary classifiers for each predictor set, and specify to cross validate. It is good practice to standardize the predictors and define the class order. Specify to standardize the predictors using an SVM template.

```
t = templateSVM('Standardize',1);
CVMdl = fitcecoc(X,Y,'CrossVal','on','Learners',t,...
    'ClassNames',classOrder);
```

CVMdl is a ClassificationPartitionedECOC model. By default, the software implements 10-fold cross validation.

Compute the classification error (proportion of misclassified observations) for the out-of-fold observations.

```
L = kfoldLoss(CVMdl)
```

```
L =
```

```
0.0400
```

Examine the result when the cost of misclassifying a flower as 'versicolor' is 10, and any other error is 1. Write a function called `noversicolor.m` that attributes a cost of 1 for misclassification, but 10 for misclassifying a flower as versicolor, and save it on your MATLAB path.

```
function averageCost = noversicolor(CMP,Xtrain,Ytrain,Wtrain,Xtest,Ytest,Wtest)
%noversicolor Example custom cross-validation function
%   Attributes a cost of 10 for misclassifying versicolor irises, and 1 for
%   the other irises. This example function requires the |fisheriris| data
%   set.
Ypredict = predict(CMP,Xtest);
misclassified = not(strcmp(Ypredict,Ytest)); % Different result
classifiedAsVersicolor = strcmp(Ypredict,'versicolor'); % Index of bad decisions
cost = sum(misclassified) + ...
    9*sum(misclassified & classifiedAsVersicolor); % Total differences
averageCost = cost/numel(Ytest); % Average error
end
```

Compute the mean misclassification error with the `noversicolor` cost.

```
foldLoss = kfoldfun(CVMdl,@noversicolor);
mean(foldLoss)
```

```
ans =  
    0.0667
```

## See Also

[ClassificationECOC](#) | [ClassificationPartitionedECOC](#) |  
[ClassificationPartitionedModel](#) | [crossval](#) | [fitcecoc](#) | [kfoldEdge](#) |  
[kfoldLoss](#) | [kfoldMargin](#) | [kfoldPredict](#)

# kfoldfun

**Class:** ClassificationPartitionedModel

Cross validate function

## Syntax

```
vals = kfoldfun(CVMdl, fun)
```

## Description

`vals = kfoldfun(CVMdl, fun)` cross validates the function `fun` by applying `fun` to the data stored in the cross-validated model `CVMdl`. You must pass `fun` as a function handle.

## Input Arguments

**CVMdl** — Cross-validated model

ClassificationPartitionedECOC model |  
ClassificationPartitionedEnsemble model |  
ClassificationPartitionedModel model

Cross-validated model, specified as a ClassificationPartitionedECOC model, ClassificationPartitionedEnsemble model, or a ClassificationPartitionedModel model.

**fun** — Cross-validated function

function handle

Cross-validated function, specified as a function handle. `fun` has the syntax

```
testvals = fun(CMP, Xtrain, Ytrain, Wtrain, Xtest, Ytest, Wtest)
```

- `CMP` is a compact model stored in one element of the `CVMdl.Trained` property.

- `Xtrain` is the training matrix of predictor values.
- `Ytrain` is the training array of response values.
- `Wtrain` are the training weights for observations.
- `Xtest` and `Ytest` are the test data, with associated weights `Wtest`.
- The returned value `testvals` needs the same size across all folds.

Data Types: `function_handle`

## Output Arguments

### **vals** — Cross-validation results

numeric matrix

Cross-validation results, returned as an numeric matrix. `vals` is the arrays of `testvals` output, concatenated vertically over all folds. For example, if `testvals` from every fold is a numeric vector of length `N`, `kfoldfun` returns a `KFold-by-N` numeric matrix with one row per fold.

Data Types: `double`

## Examples

### Estimate Classification Error Using a Custom Loss Function

Train a classification tree classifier, and then cross validate it using a custom  $k$ -fold loss function.

Load Fisher's iris data set.

```
load fisheriris
```

Train a classification tree classifier.

```
Mdl = fitctree(meas,species);
```

`Mdl` is a `ClassificationTree` model.

Cross validate `Mdl` using the default 10-fold cross validation. Compute the classification error (proportion of misclassified observations) for the out-of-fold observations.

```
rng(1); % For reproducibility
CVMdl = crossval(Mdl);
L = kfoldLoss(CVMdl)
```

```
L =
    0.0467
```

Examine the result when the cost of misclassifying a flower as 'versicolor' is 10, and any other error is 1. Write a function called `noversicolor.m` that attributes a cost of 1 for misclassification, but 10 for misclassifying a flower as versicolor, and save it on your MATLAB path.

```
function averageCost = noversicolor(CMP,Xtrain,Ytrain,Wtrain,Xtest,Ytest,Wtest)
%noversicolor Example custom cross-validation function
%   Attributes a cost of 10 for misclassifying versicolor irises, and 1 for
%   the other irises. This example function requires the |fisheriris| data
%   set.
Ypredict = predict(CMP,Xtest);
misclassified = not(strcmp(Ypredict,Ytest)); % Different result
classifiedAsVersicolor = strcmp(Ypredict,'versicolor'); % Index of bad decisions
cost = sum(misclassified) + ...
    9*sum(misclassified & classifiedAsVersicolor); % Total differences
averageCost = cost/numel(Ytest); % Average error
end
```

Compute the mean misclassification error with the `noversicolor` cost.

```
mean(kfoldfun(CVMdl,@noversicolor))
```

```
ans =
    0.2267
```

## See Also

ClassificationPartitionedECOC | ClassificationPartitionedModel |  
crossval | crossval | kfoldEdge | kfoldLoss | kfoldMargin | kfoldPredict

## kfoldfun

**Class:** RegressionPartitionedModel

Cross validate function

### Syntax

```
vals = kfoldfun(obj, fun)
```

### Description

`vals = kfoldfun(obj, fun)` cross validates the function `fun` by applying `fun` to the data stored in the cross-validated model `obj`. You must pass `fun` as a function handle.

### Input Arguments

#### **obj**

Object of class `RegressionPartitionedModel` or `RegressionPartitionedEnsemble`. Create `obj` with `fitrtree` or `fitrensemble` along with one of the cross-validation options: `'CrossVal'`, `'KFold'`, `'Holdout'`, `'Leaveout'`, or `'CVPartition'`. Alternatively, create `obj` from a regression tree or regression ensemble with `crossval`.

#### **fun**

A function handle for a cross-validation function. `fun` has the syntax

```
testvals = fun(CMP, Xtrain, Ytrain, Wtrain, Xtest, Ytest, Wtest)
```

- `CMP` is a compact model stored in one element of the `obj.Trained` property.
- `Xtrain` is the training matrix of predictor values.
- `Ytrain` is the training array of response values.



- `Wtrain` are the training weights for observations.
- `Xtest` and `Ytest` are the test data, with associated weights `Wtest`.
- The returned value `testvals` must have the same size across all folds.

## Output Arguments

### `vals`

The arrays of `testvals` output, concatenated vertically over all folds. For example, if `testvals` from every fold is a numeric vector of length `N`, `kfoldfun` returns a `KFold-by-N` numeric matrix with one row per fold.

## Examples

Cross validate a regression tree, and obtain the mean squared error (see `kfoldLoss`):

```
load imports-85
t = fitrtree(X(:,[4 5]),X(:,16),...
    'predictorNames',{'length' 'width'},...
    'responseName','price');
cv = crossval(t);
L = kfoldLoss(cv)
```

```
L =
    1.5489e+007
```

Examine the result of simple averaging of responses instead of using predictions:

```
f = @(cmp,Xtrain,Ytrain,Wtrain,Xtest,Ytest,Wtest)...
    mean((Ytest-mean(Ytrain)).^2)
mean(kfoldfun(cv,f))
```

```
ans =
    6.3497e+007
```

## See Also

`RegressionPartitionedEnsemble` | `RegressionPartitionedModel` | `crossval` | `fitrtree` | `kfoldLoss` | `kfoldPredict`

# kfoldLoss

**Class:** ClassificationPartitionedECOC

Classification loss for observations not used for training

## Syntax

```
loss = kfoldLoss(CVMdl)
loss = kfoldLoss(CVMdl, Name, Value)
```

## Description

`loss = kfoldLoss(CVMdl)` returns the classification loss obtained by the cross-validated ECOC model (`ClassificationPartitionedECOC`) `CVMdl`. For every fold, this function computes the classification loss for in-fold observations using a model trained on out-of-fold observations. `CVMdl.X` contains both sets of observations.

`loss = kfoldLoss(CVMdl, Name, Value)` returns the classification loss with additional options specified by one or more `Name, Value` pair arguments.

For example, specify the number of folds, decoding scheme, or verbosity level.

## Input Arguments

**`CVMdl` — Cross-validated ECOC model**

`ClassificationPartitionedECOC` model

Cross-validated ECOC model, specified as a `ClassificationPartitionedECOC` model. You can create a `ClassificationPartitionedECOC` model by:

- Passing a trained ECOC model (`ClassificationECOC`) to `crossval`
- Training an ECOC model using `fitcecoc` and setting any one of these cross-validation name-value pair arguments: `'CrossVal'`, `'KFold'`, `'Holdout'`, `'Leaveout'`, or `'CVPartition'`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### BinaryLoss — Binary learner loss function

'hamming' | 'linear' | 'logit' | 'exponential' | 'binodeviance' | 'hinge'  
| 'quadratic' | function handle

Binary learner loss function, specified as the comma-separated pair consisting of 'BinaryLoss' and a built-in, loss-function name or function handle.

- This table contains names and descriptions of the built-in functions, where  $y_j$  is a class label for a particular binary learner (in the set  $\{-1, 1, 0\}$ ),  $s_j$  is the score for observation  $j$ , and  $g(y_j, s_j)$  is the binary loss formula.

Value	Description	Score Domain	$g(y_j, s_j)$
'binodeviance'	Binomial deviance	$(-\infty, \infty)$	$\log[1 + \exp(-2y_j s_j)] / [2\log(2)]$
'exponential'	Exponential	$(-\infty, \infty)$	$\exp(-y_j s_j) / 2$
'hamming'	Hamming	$[0, 1]$ or $(-\infty, \infty)$	$[1 - \text{sign}(y_j s_j)] / 2$
'hinge'	Hinge	$(-\infty, \infty)$	$\max(0, 1 - y_j s_j) / 2$
'linear'	Linear	$(-\infty, \infty)$	$(1 - y_j s_j) / 2$
'logit'	Logistic	$(-\infty, \infty)$	$\log[1 + \exp(-y_j s_j)] / [2\log(2)]$
'quadratic'	Quadratic	$[0, 1]$	$[1 - y_j(2s_j - 1)]^2 / 2$

The software normalizes the binary losses such that the loss is 0.5 when  $y_j = 0$ . Also, the software calculates the mean binary loss for each class.

- For a custom binary loss function, e.g., `customFunction`, specify its function handle 'BinaryLoss', @`customFunction`.

`customFunction` should have this form:

```
bLoss = customFunction(M, s)
```

where:

- $M$  is the  $K$ -by- $L$  coding matrix stored in `Mdl.CodingMatrix`.
- $s$  is the 1-by- $L$  row vector of classification scores.
- `bLoss` is the classification loss. This scalar aggregates the binary losses for every learner in a particular class. For example, you can use the mean binary loss to aggregate the loss over the learners for each class.
- $K$  is the number of classes.
- $L$  is the number of binary learners.

For an example of passing a custom binary loss function, see “Predict Test-Sample Labels of ECOC Models Using Custom Binary Loss Function” on page 32-4933.

By default, if all binary learners are:

- SVMs or linear classification models of SVM learners, then `BinaryLoss` is 'hinge'
- Ensembles trained by `AdaboostM1` or `GentleBoost`, then `BinaryLoss` is 'exponential'
- Ensembles trained by `LogitBoost`, then `BinaryLoss` is 'binodeviance'
- Linear classification models of logistic regression learners, or you specify to predict class posterior probabilities (i.e., set 'FitPosterior', 1 in `fitcecoc`), then `BinaryLoss` is 'quadratic'

Otherwise, the default `BinaryLoss` is 'hamming'. To check the default value, use dot notation to display the `BinaryLoss` property of the trained model at the command line.

Example: `'BinaryLoss', 'binodeviance'`

Data Types: `char` | `function_handle`

### Decoding — Decoding scheme

`'lossweighted'` (default) | `'lossbased'`

Decoding scheme that aggregates the binary losses on page 32-4941, specified as the comma-separated pair consisting of 'Decoding' and 'lossweighted' or 'lossbased'.

Example: `'Decoding', 'lossbased'`

Data Types: `char`

### Folds — Fold indices for prediction

`1:Mdl.KFold` (default) | numeric vector of positive integers

Fold indices for prediction, specified as the comma-separated pair consisting of 'Folds' and a numeric vector of positive integers. The elements of `Folds` must range from 1 through `Mdl.KFold`.

The software only uses the folds specified in `Folds` for prediction.

Example: 'Folds', [1 4 10]

Data Types: `single` | `double`

### **LossFun — Loss function**

'`classiferror`' (default) | function handle

Loss function, specified as the comma-separated pair consisting of 'LossFun' and a function handle or '`classiferror`'.

You can:

- Specify the built-in function '`classiferror`', then the loss function is the classification error on page 32-3206
- Specify your own function using function handle notation.

For what follows,  $n$  is the number of observations in the training data (`CVMDL.NumObservations`) and  $K$  is the number of classes (`numel(CVMDL.ClassNames)`). Your function needs the signature `lossvalue = lossfun(C,S,W,Cost)`, where:

- The output argument `lossvalue` is a scalar.
- You choose the function name (`lossfun`).
- $C$  is an  $n$ -by- $K$  logical matrix with rows indicating which class the corresponding observation belongs. The column order corresponds to the class order in `CVMDL.ClassNames`.

Construct  $C$  by setting  $C(p, q) = 1$  if observation  $p$  is in class  $q$ , for each row. Set every element of row  $p$  to 0.

- $S$  is an  $n$ -by- $K$  numeric matrix of negated loss values for classes. Each row corresponds to an observation. The column order corresponds to the class order in `CVMDL.ClassNames`.  $S$  resembles the output argument `NegLoss` of `kfoldPredict`.

- $\bar{w}$  is an  $n$ -by-1 numeric vector of observation weights. If you pass  $\bar{w}$ , the software normalizes its elements to sum to 1.
- $\text{Cost}$  is a  $K$ -by- $K$  numeric matrix of misclassification costs. For example,  $\text{Cost} = \text{ones}(K) - \text{eye}(K)$  specifies a cost of 0 for correct classification, and 1 for misclassification.

Specify your function using `'LossFun', @lossfun`.

Data Types: `function_handle` | `char`

### Mode — Edge aggregation level

`'average'` (default) | `'individual'`

Edge meaning, specified as the comma-separated pair consisting of `'Mode'` and `'average'` or `'individual'`.

This table describes the values.

Value	Description
<code>'average'</code>	edge is the scalar average over all folds.
<code>'individual'</code>	edge is a vector of length $k$ containing one edge per fold. $k$ is the number of folds.

Example: `'Mode', 'individual'`

Data Types: `char`

### Options — Estimation options

`[]` (default) | structure array returned by `statset`

Estimation options, specified as the comma-separated pair consisting of `'Options'` and a structure array returned by `statset`.

To invoke parallel computing:

- You need a Parallel Computing Toolbox license.
- Specify `'Options', statset('UseParallel', 1)`.

### Verbose — Verbosity level

0 (default) | 1

Verbosity level, specified as the comma-separated pair consisting of 'Verbose' and 0 or 1. Verbose controls the amount of diagnostic messages that the software displays in the Command Window.

If `Verbose` is 0, then the software does not display diagnostic messages. Otherwise, the software displays diagnostic messages.

Example: 'Verbose',1

Data Types: `single` | `double`

## Output Arguments

### **loss** — Classification loss

numeric scalar | numeric row vector

Classification loss, returned as a numeric scalar or numeric row vector.

If `Mode` is 'average', then `loss` is the average classification loss among all binary learners. Otherwise, `loss` is a 1-by-L numeric row vector containing the classification loss for each, respective binary learner, where L is the number of binary learners (`size(CVMdl.CodingMatrix,2)`).

Data Types: `single` | `double`

## Examples

### Determine $k$ -Fold Cross-Validation Loss of ECOC Models

Load Fisher's iris data set.

```
load fisheriris
X = meas;
Y = categorical(species);
classOrder = unique(Y); % Class order
rng(1); % For reproducibility
```

Train an ECOC model using SVM binary classifiers, and specify to cross validate. It is good practice to standardize the predictors and define the class order. Specify to standardize the predictors using an SVM template.



```
t = templateSVM('Standardize',1);
CVMdl = fitcecoc(X,Y,'CrossVal','on','Learners',t,'ClassNames',classOrder);
```

CVMdl is a ClassificationPartitionedECOC model. By default, the software implements 10-fold cross validation. You can alter the number of folds using the 'Kfold' name-value pair argument.

Estimate the average out-of-fold classification error.

```
L = kfoldLoss(CVMdl)
```

```
L =
    0.0400
```

The average classification error for the folds is 4%.

Alternatively, you can obtain the per-fold losses by specifying the name-value pair 'Mode','individual' in kfoldLoss.

### Display Individual Losses for Each Cross-Validation Fold

The classification loss is a measure of classifier quality. You can determine ill-performing folds by displaying the losses for each fold.

Load Fisher's iris data set.

```
load fisheriris
X = meas;
Y = categorical(species);
classOrder = unique(Y);
rng(1); % For reproducibility
```

Train an ECOC model using SVM binary classifiers and specify to use 8-fold cross validation. It is good practice to standardize the predictors and define the class order. Specify to standardize the predictors using an SVM template.

```
t = templateSVM('Standardize',1);
CVMdl = fitcecoc(X,Y,'Kfold',8,'Learners',t,'ClassNames',classOrder);
```

Estimate the average classification loss across folds, and the losses for each fold.

```
loss = kfoldLoss(CVMdl)
losses = kfoldLoss(CVMdl, 'Mode', 'individual')

loss =

    0.0333

losses =

    0.0556
    0.0526
    0.1579
         0
         0
         0
         0
         0
```

The third fold misclassifies a much higher portion of observations than any other fold.

Return the classification loss for the entire model by specifying the well-performing folds using the 'Folds' name-value pair argument.

```
loss = kfoldLoss(CVMdl, 'Folds', [1:2 4:8])

loss =

    0.0153
```

The total classification loss decreased by approximately half its original size.

Consider adjusting parameters of the binary classifiers or the coding design to see if performance for all folds improves.

### Determine ECOC Model Quality Using a Custom Cross-Validation Loss

Suppose that it is interesting to know how well a model classifies a particular class. This example shows how to pass such a custom loss function to `kfoldLoss`.

Load Fisher's iris data set.

```
load fisheriris
X = meas;
Y = categorical(species);
n = numel(Y);           % Sample size
classOrder = unique(Y) % Class order
K = numel(classOrder); % Number of classes
rng(1) % For reproducibility
```

```
classOrder =

    3x1 categorical array

    setosa
    versicolor
    virginica
```

Train an ECOC model using SVM binary classifiers, and specify to cross validate. It is good practice to standardize the predictors and define the class order. Specify to standardize the predictors using an SVM template.

```
t = templateSVM('Standardize',1);
CVMdl = fitcecoc(X,Y,'CrossVal','on','Learners',t,'ClassNames',classOrder);
```

CVMdl is a ClassificationPartitionedECOC model. By default, the software implements 10-fold cross validation. You can alter the number of folds using the 'KFold' name-value pair argument.

Compute the negated losses for the out-of-fold observations.

```
[~,negLoss] = kfoldPredict(CVMdl);
```

Create a function that takes the minimal loss for each observation, and then averages the minimal losses across all observations.

```
lossfun = @(C,S,~,~)mean(min(-negLoss,[],2));
```

Compute the cross-validated custom loss.

```
kfoldLoss(CVMdl,'LossFun',lossfun)
```

```
ans =
```

0.0101

The average, minimal, binary loss for the out-of-fold observations is 0.0101.

- “Quick Start Parallel Computing for Statistics and Machine Learning Toolbox” on page 30-2

## Definitions

### Classification Error

The classification error is a binary classification error measure that has the form

$$L = \frac{\sum_{j=1}^n w_j e_j}{\sum_{j=1}^n w_j},$$

where:

- $w_j$  is the weight for observation  $j$ . The software renormalizes the weights to sum to 1.
- $e_j = 1$  if the predicted class of observation  $j$  differs from its true class, and 0 otherwise.

In other words, it is the proportion of observations that the classifier misclassifies.

### Binary Loss

A binary loss is a function of the class and classification score that determines how well a binary learner classifies an observation into the class.

Let:

- $m_{kj}$  be element  $(k,j)$  of the coding design matrix  $M$  (i.e., the code corresponding to class  $k$  of binary learner  $j$ )
- $s_j$  be the score of binary learner  $j$  for an observation
- $g$  be the binary loss function

- $\hat{k}$  be the predicted class for the observation

In loss-based decoding [15], the class producing the minimum sum of the binary losses over binary learners determines the predicted class of an observation, that is,

$$\hat{k} = \operatorname{argmin}_k \sum_{j=1}^L |m_{kj}| g(m_{kj}, s_j).$$

In loss-weighted decoding [15], the class producing the minimum average of the binary losses over binary learners determines the predicted class of an observation, that is,

$$\hat{k} = \operatorname{argmin}_k \frac{\sum_{j=1}^L |m_{kj}| g(m_{kj}, s_j)}{\sum_{j=1}^L |m_{kj}|}.$$

Allwein et al. [2] suggest that loss-weighted decoding improves classification accuracy by keeping loss values for all classes in the same dynamic range.

This table summarizes the supported loss functions, where  $y_j$  is a class label for a particular binary learner (in the set  $\{-1, 1, 0\}$ ),  $s_j$  is the score for observation  $j$ , and  $g(y_j, s_j)$ .

Value	Description	Score Domain	$g(y_j, s_j)$
'binodeviance'	Binomial deviance	$(-\infty, \infty)$	$\log[1 + \exp(-2y_j s_j)] / [2\log(2)]$
'exponential'	Exponential	$(-\infty, \infty)$	$\exp(-y_j s_j) / 2$
'hamming'	Hamming	$[0, 1]$ or $(-\infty, \infty)$	$[1 - \operatorname{sign}(y_j s_j)] / 2$
'hinge'	Hinge	$(-\infty, \infty)$	$\max(0, 1 - y_j s_j) / 2$
'linear'	Linear	$(-\infty, \infty)$	$(1 - y_j s_j) / 2$
'logit'	Logistic	$(-\infty, \infty)$	$\log[1 + \exp(-y_j s_j)] / [2\log(2)]$
'quadratic'	Quadratic	$[0, 1]$	$[1 - y_j(2s_j - 1)]^2 / 2$

The software normalizes the binary losses such that the loss is 0.5 when  $y_j = 0$ , and aggregates using the average of the binary learners [2].

Do not confuse the binary loss with the overall classification loss (specified by the `LossFun` name-value pair argument of `predict` and `loss`), e.g., classification error, which measures how well an ECOC classifier performs as a whole.

## References

- [1] Allwein, E., R. Schapire, and Y. Singer. “Reducing multiclass to binary: A unifying approach for margin classifiers.” *Journal of Machine Learning Research*. Vol. 1, 2000, pp. 113–141.
- [2] Escalera, S., O. Pujol, and P. Radeva. “On the decoding process in ternary error-correcting output codes.” *IEEE Transactions on Pattern Analysis and Machine Intelligence*. Vol. 32, Issue 7, 2010, pp. 120–134.
- [3] Escalera, S., O. Pujol, and P. Radeva. “Separability of ternary codes for sparse designs of error-correcting output codes.” *Pattern Recogn.* Vol. 30, Issue 3, 2009, pp. 285–297.

## See Also

`ClassificationECOC` | `ClassificationPartitionedModel` | `fitcecoc` |  
`kfoldPredict` | `loss` | `statset`

## Topics

- “Quick Start Parallel Computing for Statistics and Machine Learning Toolbox” on page 30-2
- “Reproducibility in Parallel Statistical Computations” on page 30-14
- “Concepts of Parallel Computing in Statistics and Machine Learning Toolbox” on page 30-8

# kfoldLoss

**Class:** ClassificationPartitionedEnsemble

Classification loss for observations not used for training

## Syntax

```
L = kfoldLoss(ens)
L = kfoldLoss(ens,Name,Value)
```

## Description

`L = kfoldLoss(ens)` returns loss obtained by cross-validated classification model `ens`. For every fold, this method computes classification loss for in-fold observations using a model trained on out-of-fold observations.

`L = kfoldLoss(ens,Name,Value)` calculates loss with additional options specified by one or more `Name,Value` pair arguments. You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

## Input Arguments

### **ens**

Object of class `ClassificationPartitionedEnsemble`. Create `ens` with `fitcensemble` along with one of the cross-validation options: `'crossval'`, `'kfold'`, `'holdout'`, `'leaveout'`, or `'cvpartition'`. Alternatively, create `ens` from a classification ensemble with `crossval`.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single

quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

### **folds**

Indices of folds ranging from 1 to `ens.KFold`. Use only these folds for predictions.

**Default:** `1:ens.KFold`

### **lossfun**

Loss function, specified as the comma-separated pair consisting of 'LossFun' and a built-in, loss-function name or function handle.

- The following lists available loss functions. Specify one using its corresponding character vector.

Value	Description
'binodeviance'	Binomial deviance
'classiferror'	Classification error
'exponential'	Exponential
'hinge'	Hinge
'logit'	Logistic
'mincost'	Minimal expected misclassification cost (for classification scores that are posterior probabilities)
'quadratic'	Quadratic

'mincost' is appropriate for classification scores that are posterior probabilities.

- Bagged and subspace ensembles return posterior probabilities by default (`ens.Method` is 'Bag' or 'Subspace').
- If the ensemble method is 'AdaBoostM1', 'AdaBoostM2', GentleBoost, or 'LogitBoost', then, to use posterior probabilities as classification scores, you must specify the double-logit score transform by entering
 

```
ens.ScoreTransform = 'doublelogit';
```
- For all other ensemble methods, the software does not support posterior probabilities as classification scores.



- Specify your own function using function handle notation.

Suppose that  $n$  be the number of observations in  $X$  and  $K$  be the number of distinct classes (`numel(ens.ClassNames)`), `ens` is the input model). Your function must have this signature

```
lossvalue = lossfun(C,S,W,Cost)
```

where:

- The output argument `lossvalue` is a scalar.
- You choose the function name (`lossfun`).
- $C$  is an  $n$ -by- $K$  logical matrix with rows indicating which class the corresponding observation belongs. The column order corresponds to the class order in `ens.ClassNames`.

Construct  $C$  by setting  $C(p, q) = 1$  if observation  $p$  is in class  $q$ , for each row. Set all other elements of row  $p$  to 0.

- $S$  is an  $n$ -by- $K$  numeric matrix of classification scores. The column order corresponds to the class order in `ens.ClassNames`.  $S$  is a matrix of classification scores, similar to the output of `predict`.
- $W$  is an  $n$ -by-1 numeric vector of observation weights. If you pass  $W$ , the software normalizes them to sum to 1.
- $Cost$  is a  $K$ -by- $K$  numeric matrix of misclassification costs. For example, `Cost = ones(K) - eye(K)` specifies a cost of 0 for correct classification, and 1 for misclassification.

Specify your function using `'LossFun', @lossfun`.

For more details on loss functions, see “Classification Loss” on page 32-3213.

**Default:** `'classiferror'`

#### mode

A character vector for determining the output of `kfoldLoss`:

- `'average'` —  $L$  is a scalar, the loss averaged over all folds.
- `'individual'` —  $L$  is a vector of length `ens.KFold`, where each entry is the loss for a fold.

- 'cumulative' —  $L$  is a vector in which element  $J$  is obtained by using learners  $1:J$  from the input list of learners.

**Default:** 'average'

## Output Arguments

$L$

Loss, by default the fraction of misclassified data.  $L$  can be a vector, and can mean different things, depending on the name-value pair settings.

## Examples

### Estimate Cross-Validated Classification Error

Load the ionosphere data set.

```
load ionosphere
```

Boost 100 trees using AdaBoostM1.

```
ens = fitensemble(X,Y,'AdaBoostM1',100,'Tree');
```

Cross-validate the ensemble using 10-fold cross-validation.

```
cvens = crossval(ens);
```

Estimate the cross-validated classification error.

```
L = kfoldLoss(cvens)
```

```
L =
```

0.0655

## Definitions

### Classification Loss

Classification loss functions measure the predictive inaccuracy of classification models. When comparing the same type of loss among many models, lower loss indicates a better predictive model.

Suppose that:

- $L$  is the weighted average classification loss.
- $n$  is the sample size.
- For binary classification:
  - $y_j$  is the observed class label. The software codes it as  $-1$  or  $1$  indicating the negative or positive class, respectively.
  - $f(X_j)$  is the raw classification score for observation (row)  $j$  of the predictor data  $X$ .
  - $m_j = y_j f(X_j)$  is the classification score for classifying observation  $j$  into the class corresponding to  $y_j$ . Positive values of  $m_j$  indicate correct classification and do not contribute much to the average loss. Negative values of  $m_j$  indicate incorrect classification and contribute to the average loss.
- For algorithms that support multiclass classification (that is,  $K \geq 3$ ):
  - $y_j^*$  is a vector of  $K - 1$  zeros, and a 1 in the position corresponding to the true, observed class  $y_j$ . For example, if the true class of the second observation is the third class and  $K = 4$ , then  $y_2^* = [0 \ 0 \ 1 \ 0]'$ . The order of the classes corresponds to the order in the `ClassNames` property of the input model.
  - $f(X_j)$  is the length  $K$  vector of class scores for observation  $j$  of the predictor data  $X$ . The order of the scores corresponds to the order of the classes in the `ClassNames` property of the input model.
  - $m_j = y_j^{*'} f(X_j)$ . Therefore,  $m_j$  is the scalar classification score that the model predicts for the true, observed class.

- The weight for observation  $j$  is  $w_j$ . The software normalizes the observation weights so that they sum to the corresponding prior class probability. The software also normalizes the prior probabilities so they sum to 1. Therefore,

$$\sum_{j=1}^n w_j = 1.$$

The supported loss functions are:

- Binomial deviance, specified using 'LossFun', 'binodeviance'. Its equation is

$$L = \sum_{j=1}^n w_j \log \{1 + \exp[-2m_j]\}.$$

- Exponential loss, specified using 'LossFun', 'exponential'. Its equation is

$$L = \sum_{j=1}^n w_j \exp(-m_j).$$

- Classification error, specified using 'LossFun', 'classiferror'. It is the weighted fraction of misclassified observations, with equation

$$L = \sum_{j=1}^n w_j I\{\hat{y}_j \neq y_j\}.$$

$\hat{y}_j$  is the class label corresponding to the class with the maximal posterior probability.  $I\{x\}$  is the indicator function.

- Hinge loss, specified using 'LossFun', 'hinge'. Its equation is

$$L = \sum_{j=1}^n w_j \max\{0, 1 - m_j\}.$$

- Logit loss, specified using 'LossFun', 'logit'. Its equation is

$$L = \sum_{j=1}^n w_j \log(1 + \exp(-m_j)).$$

- Minimal cost, specified using 'LossFun', 'mincost'. The software computes the weighted minimal cost using this procedure for observations  $j = 1, \dots, n$ :

- 1 Estimate the 1-by- $K$  vector of expected classification costs for observation  $j$

$$\gamma_j = f(X_j)'C.$$

$f(X_j)$  is the column vector of class posterior probabilities for binary and multiclass classification.  $C$  is the cost matrix the input model stores in the property `Cost`.

- 2 For observation  $j$ , predict the class label corresponding to the minimum, expected classification cost:

$$\hat{y}_j = \min_{j=1,\dots,K}(\gamma_j).$$

- 3 Using  $C$ , identify the cost incurred ( $c_j$ ) for making the prediction.

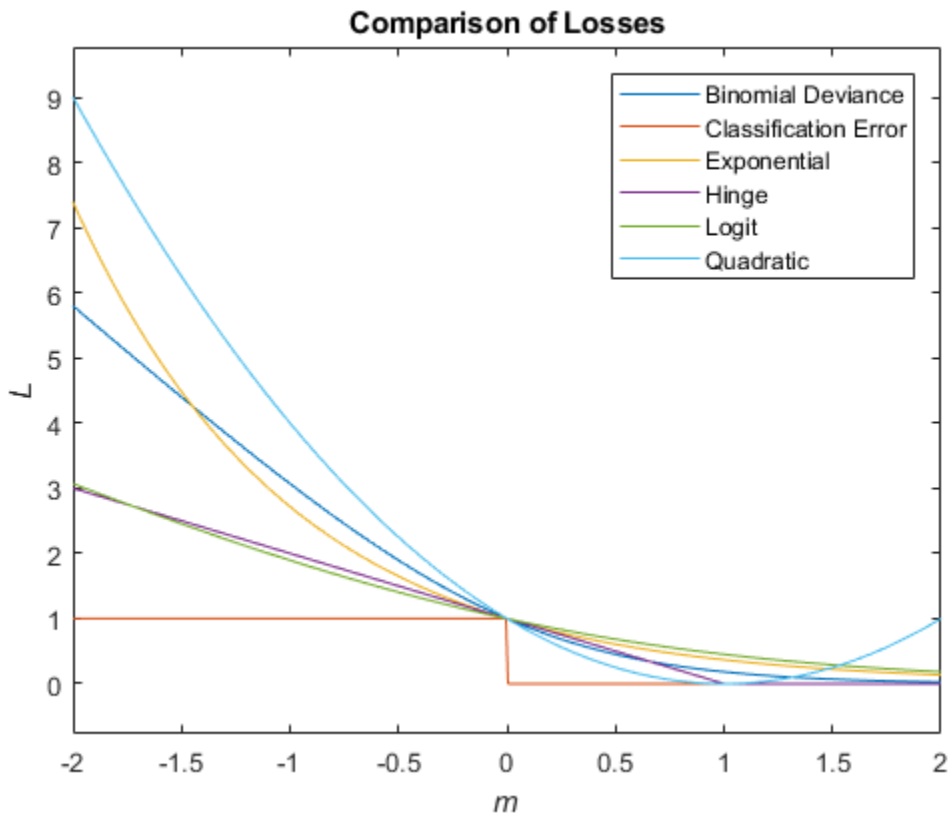
The weighted, average, minimum cost loss is

$$L = \sum_{j=1}^n w_j c_j.$$

- Quadratic loss, specified using 'LossFun', 'quadratic'. Its equation is

$$L = \sum_{j=1}^n w_j (1 - m_j)^2.$$

This figure compares some of the loss functions for one observation over  $m$  (some functions are normalized to pass through  $[0,1]$ ).



## See Also

[crossval](#) | [kfoldEdge](#) | [kfoldMargin](#) | [kfoldPredict](#) | [kfoldfun](#)

# kfoldLoss

**Class:** ClassificationPartitionedLinear

Classification loss for observations not used in training

## Syntax

```
L = kfoldLoss(CVMdl)
L = kfoldLoss(CVMdl, Name, Value)
```

## Description

`L = kfoldLoss(CVMdl)` returns the cross-validated classification losses on page 32-3227 obtained by the cross-validated, binary, linear classification model `CVMdl`. That is, for every fold, `kfoldLoss` estimates the classification loss for observations that it holds out when it trains using all other observations.

`L` contains a classification loss for each regularization strength in the linear classification models that compose `CVMdl`.

`L = kfoldLoss(CVMdl, Name, Value)` uses additional options specified by one or more `Name, Value` pair arguments. For example, indicate which folds to use for the loss calculation or specify the classification-loss function.

## Input Arguments

**`CVMdl` — Cross-validated, binary, linear classification model**

ClassificationPartitionedLinear model object

Cross-validated, binary, linear classification model, specified as a ClassificationPartitionedLinear model object. You can create a ClassificationPartitionedLinear model using `fitclinear` and specifying any one of the cross-validation, name-value pair arguments, for example, `CrossVal`.

To obtain estimates, `ClassificationPartitionedLinear.kfoldLoss` applies the same data used to cross-validate the linear classification model ( $X$  and  $Y$ ).

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### **Folds** — Fold indices to use for classification-score prediction

1: `CVMDL.KFold` (default) | numeric vector of positive integers

Fold indices to use for classification-score prediction, specified as the comma-separated pair consisting of 'Folds' and a numeric vector of positive integers. The elements of `Folds` must range from 1 through `CVMDL.KFold`.

Example: 'Folds', [1 4 10]

Data Types: `single` | `double`

### **LossFun** — Loss function

'`classiferror`' (default) | '`binodeviance`' | '`exponential`' | '`hinge`' | '`logit`' | '`mincost`' | '`quadratic`' | function handle

Loss function, specified as the comma-separated pair consisting of 'LossFun' and a built-in, loss-function name or function handle.

- The following lists available loss functions. Specify one using its corresponding character vector.

Value	Description
'binodeviance'	Binomial deviance
'classiferror'	Classification error
'exponential'	Exponential
'hinge'	Hinge
'logit'	Logistic



Value	Description
'mincost'	Minimal expected misclassification cost (for classification scores that are posterior probabilities)
'quadratic'	Quadratic

'mincost' is appropriate for classification scores that are posterior probabilities. For linear classification models, logistic regression learners return posterior probabilities as classification scores by default, but SVM learners do not (see `predict`).

- Specify your own function using function handle notation.

Let  $n$  be the number of observations in  $X$  and  $K$  be the number of distinct classes (`numel(Mdl.ClassNames)`,  $Mdl$  is the input model). Your function must have this signature

```
lossvalue = lossfun(C,S,W,Cost)
```

where:

- The output argument `lossvalue` is a scalar.
- You choose the function name (`lossfun`).
- $C$  is an  $n$ -by- $K$  logical matrix with rows indicating which class the corresponding observation belongs. The column order corresponds to the class order in `Mdl.ClassNames`.

Construct  $C$  by setting  $C(p, q) = 1$  if observation  $p$  is in class  $q$ , for each row. Set all other elements of row  $p$  to 0.

- $S$  is an  $n$ -by- $K$  numeric matrix of classification scores. The column order corresponds to the class order in `Mdl.ClassNames`.  $S$  is a matrix of classification scores, similar to the output of `predict`.
- $W$  is an  $n$ -by-1 numeric vector of observation weights. If you pass  $W$ , the software normalizes them to sum to 1.
- $Cost$  is a  $K$ -by- $K$  numeric matrix of misclassification costs. For example, `Cost = ones(K) - eye(K)` specifies a cost of 0 for correct classification, and 1 for misclassification.

Specify your function using `'LossFun', @lossfun`.

Data Types: `char` | `function_handle`

**Mode — Loss aggregation level**

'average' (default) | 'individual'

Loss aggregation level, specified as the comma-separated pair consisting of 'Mode' and 'average' or 'individual'.

Value	Description
'average'	Returns losses averaged over all folds
'individual'	Returns losses for each fold

Example: 'Mode', 'individual'

Data Types: char

## Output Arguments

**L — Cross-validated classification losses**

numeric scalar | numeric vector | numeric matrix

Cross-validated classification losses on page 32-3227, returned as a numeric scalar, vector, or matrix. The interpretation of **L** depends on `LossFun`.

Let  $R$  be the number of regularizations strengths is the cross-validated models (stored in `numel(CVMdl.Trained{1}.Lambda)`) and  $F$  be the number of folds (stored in `CVMdl.KFold`).

- If `Mode` is 'average', then **L** is a 1-by- $R$  vector.  $L(j)$  is the average classification loss over all folds of the cross-validated model that uses regularization strength  $j$ .
- Otherwise, **L** is an  $F$ -by- $R$  matrix.  $L(i, j)$  is the classification loss for fold  $i$  of the cross-validated model that uses regularization strength  $j$ .

To estimate **L**, `kfoldLoss` uses the data that created `CVMdl` (see  $X$  and  $Y$ ).

## Examples

## Estimate $k$ -Fold Cross-Validation Classification Error

Load the NLP data set.

```
load nlpdata
```

$X$  is a sparse matrix of predictor data, and  $Y$  is a categorical vector of class labels. There are more than two classes in the data.

The models should identify whether the word counts in a web page are from the Statistics and Machine Learning Toolbox™ documentation. So, identify the labels that correspond to the Statistics and Machine Learning Toolbox™ documentation web pages.

```
Ystats = Y == 'stats';
```

Cross-validate a binary, linear classification model that can identify whether the word counts in a documentation web page are from the Statistics and Machine Learning Toolbox™ documentation.

```
rng(1); % For reproducibility  
CVMdl = fitclinear(X, Ystats, 'CrossVal', 'on');
```

CVMdl is a ClassificationPartitionedLinear model. By default, the software implements 10-fold cross validation. You can alter the number of folds using the 'KFold' name-value pair argument.

Estimate the average of the out-of-fold, classification error rates.

```
ce = kfoldLoss(CVMdl)
```

```
ce =
```

```
7.6017e-04
```

Alternatively, you can obtain the per-fold classification error rates by specifying the name-value pair 'Mode', 'individual' in kfoldLoss.

### Specify Custom Classification Loss

Load the NLP data set. Preprocess the data as in “Estimate k-Fold Cross-Validation Classification Error” on page 32-3220, and transpose the predictor data.

```
load nlpdata
Ystats = Y == 'stats';
X = X';
```

Cross-validate a binary, linear classification model using 5-fold cross-validation. Optimize the objective function using SpaRSA. Specify that the predictor observations correspond to columns.

```
rng(1); % For reproducibility
CVMdl = fitclinear(X,Ystats,'Solver','sparsa','KFold',5,...
    'ObservationsIn','columns');
CMdl = CVMdl.Trained{1};
```

CVMdl is a ClassificationPartitionedLinear model. It contains the property Trained, which is a 5-by-1 cell array holding a ClassificationLinear models that the software trained using the training set of each fold.

Create an anonymous function that measures linear loss, that is,

$$L = \frac{\sum_j -w_j y_j f_j}{\sum_j w_j}.$$

$w_j$  is the weight for observation  $j$ ,  $y_j$  is response  $j$  (-1 for the negative class, and 1 otherwise), and  $f_j$  is the raw classification score of observation  $j$ . Custom loss functions must be written in a particular form. For rules on writing a custom loss function, see the LossFun name-value pair argument. Because the function does not use classification cost, use ~ to have kfoldLoss ignore its position.

```
linearloss = @(C,S,W,~) sum(-W.*sum(S.*C,2))/sum(W);
```

Estimate the average cross-validated classification loss using the linear loss function. Also, obtain the loss for each fold.

```
ce = kfoldLoss(CVMdl,'LossFun',linearloss)
ceFold = kfoldLoss(CVMdl,'LossFun',linearloss,'Mode','individual')

ce =
```

```

-8.0982

ceFold =

-8.3165
-8.7633
-7.4342
-8.0423
-7.9347

```

### Find Good Lasso Penalty Using $k$ -fold Classification Loss

To determine a good lasso-penalty strength for a linear classification model that uses a logistic regression learner, compare test-sample classification error rates.

Load the NLP data set. Preprocess the data as in “Specify Custom Classification Loss” on page 32-3221.

```

load nlpdata
Ystats = Y == 'stats';
X = X';

```

Create a set of 11 logarithmically-spaced regularization strengths from  $10^{-6}$  through  $10^{0.5}$ .

```
Lambda = logspace(-6,-0.5,11);
```

Cross-validate binary, linear classification models using 5-fold cross-validation, and that use each of the regularization strengths. Optimize the objective function using SpARSA. Lower the tolerance on the gradient of the objective function to  $1e-8$ .

```

rng(10); % For reproducibility
CVMdl = fitclinear(X,Ystats,'ObservationsIn','columns',...
    'KFold',5,'Learner','logistic','Solver','sparsa',...
    'Regularization','lasso','Lambda',Lambda,'GradientTolerance',1e-8)

CVMdl =

classreg.learning.partition.ClassificationPartitionedLinear

```

```
CrossValidatedModel: 'Linear'  
  ResponseName: 'Y'  
  NumObservations: 31572  
    KFold: 5  
    Partition: [1x1 cvpartition]  
    ClassNames: [0 1]  
  ScoreTransform: 'none'
```

**Extract a trained linear classification model.**

```
Mdl1 = CVMdl.Trained{1}
```

```
Mdl1 =
```

```
ClassificationLinear  
  ResponseName: 'Y'  
  ClassNames: [0 1]  
  ScoreTransform: 'logit'  
    Beta: [34023x11 double]  
    Bias: [1x11 double]  
    Lambda: [1x11 double]  
  Learner: 'logistic'
```

Mdl1 is a `ClassificationLinear` model object. Because `Lambda` is a sequence of regularization strengths, you can think of `Mdl` as 11 models, one for each regularization strength in `Lambda`.

**Estimate the cross-validated classification error.**

```
ce = kfoldLoss(CVMdl);
```

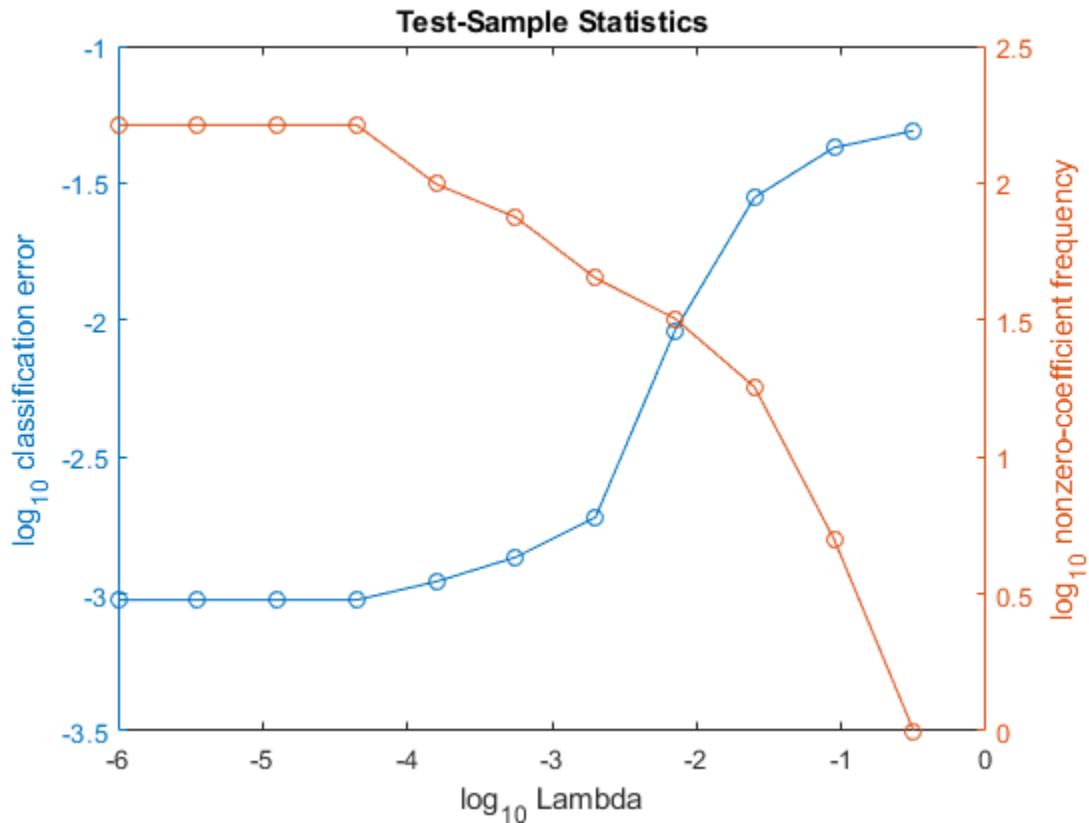
Because there are 11 regularization strengths, `ce` is a 1-by-11 vector of classification error rates.

Higher values of `Lambda` lead to predictor variable sparsity, which is a good quality of a classifier. For each regularization strength, train a linear classification model using the entire data set and the same options as when you cross-validated the models. Determine the number of nonzero coefficients per model.

```
Mdl = fitclinear(X,Ystats,'ObservationsIn','columns',...  
    'Learner','logistic','Solver','sparsa','Regularization','lasso',...  
    'Lambda',Lambda,'GradientTolerance',1e-8);  
numNZCoeff = sum(Mdl.Beta~=0);
```

In the same figure, plot the cross-validated, classification error rates and frequency of nonzero coefficients for each regularization strength. Plot all variables on the log scale.

```
figure;  
[h,hL1,hL2] = plotyy(log10(Lambda),log10(ce),...  
    log10(Lambda),log10(numNZCoeff));  
hL1.Marker = 'o';  
hL2.Marker = 'o';  
ylabel(h(1),'log_{10} classification error')  
ylabel(h(2),'log_{10} nonzero-coefficient frequency')  
xlabel('log_{10} Lambda')  
title('Test-Sample Statistics')  
hold off
```



Choose the index of the regularization strength that balances predictor variable sparsity and low classification error. In this case, a value between  $10^{-4}$  to  $10^{-1}$  should suffice.

```
idxFinal = 7;
```

Select the model from `Mdl` with the chosen regularization strength.

```
MdlFinal = selectModels(Mdl,idxFinal);
```



`MdlFinal` is a `ClassificationLinear` model containing one regularization strength. To estimate labels for new observations, pass `MdlFinal` and the new data to `predict`.

## Definitions

### Classification Loss

Classification loss functions measure the predictive inaccuracy of classification models. When comparing the same type of loss among many models, lower loss indicates a better predictive model.

Suppose that:

- $L$  is the weighted average classification loss.
- $n$  is the sample size.
- For binary classification:
  - $y_j$  is the observed class label. The software codes it as  $-1$  or  $1$  indicating the negative or positive class, respectively.
  - $f(X_j)$  is the raw classification score for observation (row)  $j$  of the predictor data  $X$ .
  - $m_j = y_j f(X_j)$  is the classification score for classifying observation  $j$  into the class corresponding to  $y_j$ . Positive values of  $m_j$  indicate correct classification and do not contribute much to the average loss. Negative values of  $m_j$  indicate incorrect classification and contribute to the average loss.
- For algorithms that support multiclass classification (that is,  $K \geq 3$ ):
  - $y_j^*$  is a vector of  $K - 1$  zeros, and a 1 in the position corresponding to the true, observed class  $y_j$ . For example, if the true class of the second observation is the third class and  $K = 4$ , then  $y_2^* = [0 \ 0 \ 1 \ 0]'$ . The order of the classes corresponds to the order in the `ClassNames` property of the input model.
  - $f(X_j)$  is the length  $K$  vector of class scores for observation  $j$  of the predictor data  $X$ . The order of the scores corresponds to the order of the classes in the `ClassNames` property of the input model.
  - $m_j = y_j^{*'} f(X_j)$ . Therefore,  $m_j$  is the scalar classification score that the model predicts for the true, observed class.

- The weight for observation  $j$  is  $w_j$ . The software normalizes the observation weights so that they sum to the corresponding prior class probability. The software also normalizes the prior probabilities so they sum to 1. Therefore,

$$\sum_{j=1}^n w_j = 1.$$

The supported loss functions are:

- Binomial deviance, specified using 'LossFun', 'binodeviance'. Its equation is

$$L = \sum_{j=1}^n w_j \log \{1 + \exp[-2m_j]\}.$$

- Exponential loss, specified using 'LossFun', 'exponential'. Its equation is

$$L = \sum_{j=1}^n w_j \exp(-m_j).$$

- Classification error, specified using 'LossFun', 'classiferror'. It is the weighted fraction of misclassified observations, with equation

$$L = \sum_{j=1}^n w_j I\{\hat{y}_j \neq y_j\}.$$

$\hat{y}_j$  is the class label corresponding to the class with the maximal posterior probability.  $I\{x\}$  is the indicator function.

- Hinge loss, specified using 'LossFun', 'hinge'. Its equation is

$$L = \sum_{j=1}^n w_j \max\{0, 1 - m_j\}.$$

- Logit loss, specified using 'LossFun', 'logit'. Its equation is

$$L = \sum_{j=1}^n w_j \log(1 + \exp(-m_j)).$$

- Minimal cost, specified using 'LossFun', 'mincost'. The software computes the weighted minimal cost using this procedure for observations  $j = 1, \dots, n$ :

- 1 Estimate the 1-by- $K$  vector of expected classification costs for observation  $j$

$$\gamma_j = f(X_j)'C.$$

$f(X_j)$  is the column vector of class posterior probabilities for binary and multiclass classification.  $C$  is the cost matrix the input model stores in the property `Cost`.

- 2 For observation  $j$ , predict the class label corresponding to the minimum, expected classification cost:

$$\hat{y}_j = \min_{j=1,\dots,K}(\gamma_j).$$

- 3 Using  $C$ , identify the cost incurred ( $c_j$ ) for making the prediction.

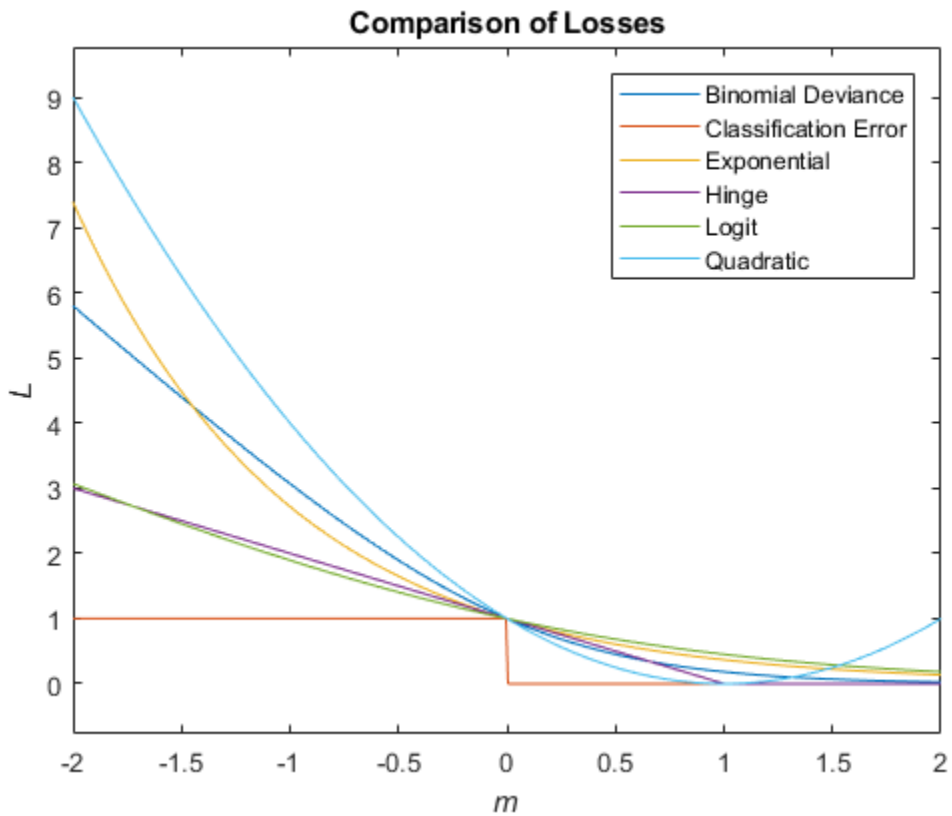
The weighted, average, minimum cost loss is

$$L = \sum_{j=1}^n w_j c_j.$$

- Quadratic loss, specified using 'LossFun', 'quadratic'. Its equation is

$$L = \sum_{j=1}^n w_j (1 - m_j)^2.$$

This figure compares some of the loss functions for one observation over  $m$  (some functions are normalized to pass through  $[0,1]$ ).



## See Also

`ClassificationLinear` | `ClassificationPartitionedLinear` | `kfoldPredict` | `loss`

Introduced in R2016a

# kfoldLoss

**Class:** ClassificationPartitionedLinearECOC

Classification loss for observations not used in training

## Syntax

```
L = kfoldLoss(CVMdl)
L = kfoldLoss(CVMdl, Name, Value)
```

## Description

`L = kfoldLoss(CVMdl)` returns the cross-validated classification error on page 32-3244 rates estimated by the cross-validated, error-correcting output codes (ECOC) model composed of linear classification models `CVMdl`. That is, for every fold, `kfoldLoss` estimates the classification error rate for observations that it holds out when it trains using all other observations. `kfoldLoss` applies the same data used create `CVMdl` (see `fitcecoc`).

`L` contains a classification loss for each regularization strength in the linear classification models that compose `CVMdl`.

`L = kfoldLoss(CVMdl, Name, Value)` uses additional options specified by one or more `Name, Value` pair arguments. For example, specify a decoding scheme, which folds to use for the loss calculation, or verbosity level.

## Input Arguments

**`CVMdl` — Cross-validated, ECOC model composed of linear classification models**

ClassificationPartitionedLinearECOC model object

Cross-validated, ECOC model composed of linear classification models, specified as a ClassificationPartitionedLinearECOC model object. You can create a ClassificationPartitionedLinearECOC model using `fitcecoc` and by:

- 1 Specifying any one of the cross-validation, name-value pair arguments, for example, `CrossVal`
- 2 Setting the name-value pair argument `Learners` to `'linear'` or a linear classification model template returned by `templateLinear`

To obtain estimates, `ClassificationPartitionedLinearECOC.kfoldLoss` applies the same data used to cross-validate the ECOC model ( $X$  and  $Y$ ).

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### BinaryLoss — Binary learner loss function

`'hamming'` | `'linear'` | `'logit'` | `'exponential'` | `'binodeviance'` | `'hinge'`  
| `'quadratic'` | function handle

Binary learner loss function, specified as the comma-separated pair consisting of `'BinaryLoss'` and a built-in, loss-function name or function handle.

- This table contains names and descriptions of the built-in functions, where  $y_j$  is a class label for a particular binary learner (in the set  $\{-1, 1, 0\}$ ),  $s_j$  is the score for observation  $j$ , and  $g(y_j, s_j)$  is the binary loss formula.

Value	Description	Score Domain	$g(y_j, s_j)$
<code>'binodeviance'</code>	Binomial deviance	$(-\infty, \infty)$	$\log[1 + \exp(-2y_j s_j)] / [2\log(2)]$
<code>'exponential'</code>	Exponential	$(-\infty, \infty)$	$\exp(-y_j s_j) / 2$
<code>'hamming'</code>	Hamming	$[0, 1]$ or $(-\infty, \infty)$	$[1 - \text{sign}(y_j s_j)] / 2$
<code>'hinge'</code>	Hinge	$(-\infty, \infty)$	$\max(0, 1 - y_j s_j) / 2$
<code>'linear'</code>	Linear	$(-\infty, \infty)$	$(1 - y_j s_j) / 2$
<code>'logit'</code>	Logistic	$(-\infty, \infty)$	$\log[1 + \exp(-y_j s_j)] / [2\log(2)]$
<code>'quadratic'</code>	Quadratic	$[0, 1]$	$[1 - y_j(2s_j - 1)]^2 / 2$

The software normalizes the binary losses such that the loss is 0.5 when  $y_j = 0$ . Also, the software calculates the mean binary loss for each class.

- For a custom binary loss function, e.g., `customFunction`, specify its function handle `'BinaryLoss', @customFunction`.

`customFunction` should have this form

```
bLoss = customFunction(M,s)
```

where:

- `M` is the  $K$ -by- $L$  coding matrix stored in `Mdl.CodingMatrix`.
- `s` is the 1-by- $L$  row vector of classification scores.
- `bLoss` is the classification loss. This scalar aggregates the binary losses for every learner in a particular class. For example, you can use the mean binary loss to aggregate the loss over the learners for each class.
- $K$  is the number of classes.
- $L$  is the number of binary learners.

For an example of passing a custom binary loss function, see “Predict Test-Sample Labels of ECOC Models Using Custom Binary Loss Function” on page 32-4933.

By default, if all binary learners are linear classification models using:

- SVM, then `BinaryLoss` is `'hinge'`
- Logistic regression, then `BinaryLoss` is `'quadratic'`

Example: `'BinaryLoss', 'binodeviance'`

Data Types: `char` | `function_handle`

### Decoding — Decoding scheme

`'lossweighted'` (default) | `'lossbased'`

Decoding scheme that aggregates the binary losses on page 32-4941, specified as the comma-separated pair consisting of `'Decoding'` and `'lossweighted'` or `'lossbased'`.

Example: `'Decoding', 'lossbased'`

Data Types: `char`

### Folds — Fold indices to use for classification-score prediction

`1:CVMdl.KFold` (default) | numeric vector of positive integers

Fold indices to use for classification-score prediction, specified as the comma-separated pair consisting of 'Folds' and a numeric vector of positive integers. The elements of Folds must range from 1 through `CVMDL.KFold`.

Example: 'Folds', [1 4 10]

Data Types: `single` | `double`

### **LossFun — Loss function**

'`classiferror`' (default) | function handle

Loss function, specified as the comma-separated pair consisting of 'LossFun' and a function handle or '`classiferror`'.

You can:

- Specify the built-in function '`classiferror`', then the loss function is the classification error on page 32-3244.
- Specify your own function using function handle notation.

For what follows,  $n$  is the number of observations in the training data (`CVMDL.NumObservations`) and  $K$  is the number of classes (`numel(CVMDL.ClassNames)`). Your function needs the signature `lossvalue = lossfun(C, S, W, Cost)`, where:

- The output argument `lossvalue` is a scalar.
- You choose the function name (`lossfun`).
- $C$  is an  $n$ -by- $K$  logical matrix with rows indicating which class the corresponding observation belongs. The column order corresponds to the class order in `CVMDL.ClassNames`.

Construct  $C$  by setting  $C(p, q) = 1$  if observation  $p$  is in class  $q$ , for each row. Set every element of row  $p$  to 0.

- $S$  is an  $n$ -by- $K$  numeric matrix of negated loss values for classes. Each row corresponds to an observation. The column order corresponds to the class order in `CVMDL.ClassNames`.  $S$  resembles the output argument `NegLoss` of `kfoldPredict`.
- $W$  is an  $n$ -by-1 numeric vector of observation weights. If you pass  $W$ , the software normalizes its elements to sum to 1.



- Cost is a K-by-K numeric matrix of misclassification costs. For example,  $\text{Cost} = \text{ones}(K) - \text{eye}(K)$  specifies a cost of 0 for correct classification, and 1 for misclassification.

Specify your function using 'LossFun', @lossfun.

Data Types: function\_handle | char

### Mode — Loss aggregation level

'average' (default) | 'individual'

Loss aggregation level, specified as the comma-separated pair consisting of 'Mode' and 'average' or 'individual'.

Value	Description
'average'	Returns losses averaged over all folds
'individual'	Returns losses for each fold

Example: 'Mode', 'individual'

Data Types: char

### Options — Estimation options

[] (default) | structure array returned by statset

Estimation options, specified as the comma-separated pair consisting of 'Options' and a structure array returned by statset.

To invoke parallel computing:

- You need a Parallel Computing Toolbox license.
- Specify 'Options', statset('UseParallel', 1).

### Verbose — Verbosity level

0 (default) | 1

Verbosity level, specified as the comma-separated pair consisting of 'Verbose' and 0 or 1. Verbose controls the amount of diagnostic messages that the software displays in the Command Window.

If `Verbose` is 0, then the software does not display diagnostic messages. Otherwise, the software displays diagnostic messages.

Example: `'Verbose', 1`

Data Types: `single` | `double`

## Output Arguments

### **L** — Cross-validated classification losses

numeric scalar | numeric vector | numeric matrix

Cross-validated classification losses on page 32-3244, returned as a numeric scalar, vector, or matrix. The interpretation of `L` depends on `LossFun`.

Let  $R$  be the number of regularizations strengths is the cross-validated models (`CVMDL.Trained{1}.BinaryLearners{1}.Lambda`) and  $F$  be the number of folds (stored in `CVMDL.KFold`).

- If `Mode` is `'average'`, then `L` is a 1-by- $R$  vector. `L(j)` is the average classification loss over all folds of the cross-validated model that uses regularization strength  $j$ .
- Otherwise, `L` is a  $F$ -by- $R$  matrix. `L(i, j)` is the classification loss for fold  $i$  of the cross-validated model that uses regularization strength  $j$ .

## Examples

### Estimate $k$ -Fold Cross-Validation Classification Error

Load the NLP data set.

```
load nlpdata
```

`X` is a sparse matrix of predictor data, and `Y` is a categorical vector of class labels.

Cross-validate an ECOC model of linear classification models.

```
rng(1); % For reproducibility
CVMDL = fitcecoc(X, Y, 'Learner', 'linear', 'CrossVal', 'on');
```

CVMDL is a ClassificationPartitionedLinearECOC model. By default, the software implements 10-fold cross validation.

Estimate the average of the out-of-fold classification error rates.

```
ce = kfoldLoss(CVMDL)
```

```
ce =  
    0.0958
```

Alternatively, you can obtain the per-fold classification error rates by specifying the name-value pair 'Mode', 'individual' in kfoldLoss.

### Specify Custom Classification Loss

Load the NLP data set. Transpose the predictor data.

```
load nlpdata  
X = X';
```

For simplicity, use the label 'others' for all observations in Y that are not 'simulink', 'dsp', or 'comm'.

```
Y(~(ismember(Y,{'simulink','dsp','comm'}))) = 'others';
```

Create a linear classification model template that specifies optimizing the objective function using SpARSA.

```
t = templateLinear('Solver','sparsa');
```

Cross-validate an ECOC model of linear classification models using 5-fold cross-validation. Optimize the objective function using SpARSA. Specify that the predictor observations correspond to columns.

```
rng(1); % For reproducibility  
CVMDL = fitcecoc(X,Y,'Learners',t,'KFold',5,'ObservationsIn','columns');  
CMDL1 = CVMDL.Trained{1}
```

```
CMDL1 =
```

```
classreg.learning.classif.CompactClassificationECOC
    ResponseName: 'Y'
    ClassNames: [comm    dsp    simulink    others]
    ScoreTransform: 'none'
    BinaryLearners: {6x1 cell}
    CodingMatrix: [4x6 double]
```

CVMD1 is a `ClassificationPartitionedLinearECOC` model. It contains the property `Trained`, which is a 5-by-1 cell array holding a `CompactClassificationECOC` models that the software trained using the training set of each fold.

Create a function that takes the minimal loss for each observation, and then averages the minimal losses across all observations. Because the function does not use the class-identifier matrix ( $C$ ), observation weights ( $\bar{w}$ ), and classification cost ( $Cost$ ), use `~` to have `kfoldLoss` ignore its their positions.

```
lossfun = @(~,S,~,~)mean(min(-S,[],2));
```

Estimate the average cross-validated classification loss using the minimal loss per observation function. Also, obtain the loss for each fold.

```
ce = kfoldLoss(CVMD1,'LossFun',lossfun)
ceFold = kfoldLoss(CVMD1,'LossFun',lossfun,'Mode','individual')
```

```
ce =
    0.0243
```

```
ceFold =
    0.0244
    0.0255
    0.0248
    0.0240
    0.0226
```

## Find Good Lasso Penalty Using Cross-Validation

To determine a good lasso-penalty strength for an ECOC model composed of linear classification models that use logistic regression learners, implement 5-fold cross-validation.

Load the NLP data set.

```
load nlpdata
```

X is a sparse matrix of predictor data, and Y is a categorical vector of class labels.

For simplicity, use the label 'others' for all observations in Y that are not 'simulink', 'dsp', or 'comm'.

```
Y(~(ismember(Y,{'simulink','dsp','comm'}))) = 'others';
```

Create a set of 11 logarithmically-spaced regularization strengths from  $10^{-7}$  through  $10^{-2}$ .

```
Lambda = logspace(-7,-2,11);
```

Create a linear classification model template that specifies to use logistic regression learners, use lasso penalties with strengths in Lambda, train using SpARSA, and lower the tolerance on the gradient of the objective function to  $1e-8$ .

```
t = templateLinear('Learner','logistic','Solver','sparsa',...
    'Regularization','lasso','Lambda',Lambda,'GradientTolerance',1e-8);
```

Cross-validate the models. To increase execution speed, transpose the predictor data and specify that the observations are in columns.

```
X = X';
rng(10); % For reproducibility
CVMdl = fitcecoc(X,Y,'Learners',t,'ObservationsIn','columns','KFold',5);
```

CVMdl is a ClassificationPartitionedLinearECOC model.

Dissect CVMdl, and each model within it.

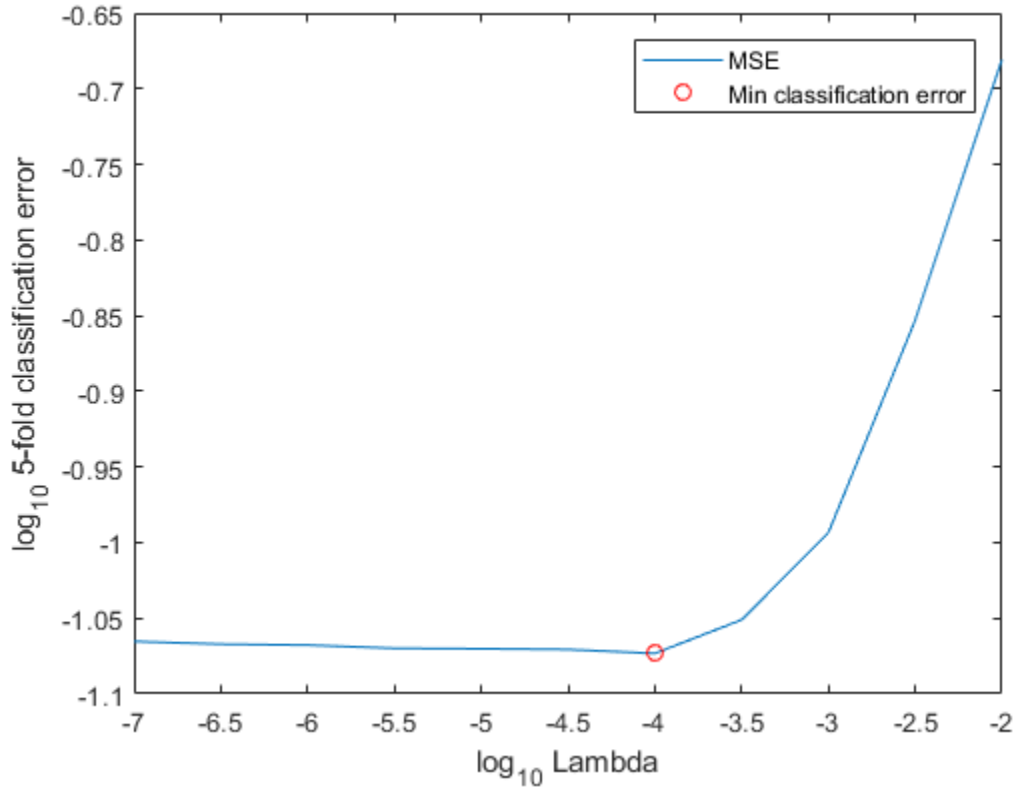
```
numECOCModels = numel(CVMdl.Trained)
ECOCmdl1 = CVMdl.Trained{1}
numCLModels = numel(ECOCmdl1.BinaryLearners)
CLMdl1 = ECOCmdl1.BinaryLearners{1}
```

```
numECOCModels =  
  
    5  
  
ECOCMdl1 =  
  
    classreg.learning.classif.CompactClassificationECOC  
        ResponseName: 'Y'  
        ClassNames: [comm    dsp    simulink    others]  
        ScoreTransform: 'none'  
        BinaryLearners: {6x1 cell}  
        CodingMatrix: [4x6 double]  
  
numCLModels =  
  
    6  
  
CLMdl1 =  
  
    ClassificationLinear  
        ResponseName: 'Y'  
        ClassNames: [-1 1]  
        ScoreTransform: 'logit'  
            Beta: [34023x11 double]  
            Bias: [1x11 double]  
            Lambda: [1x11 double]  
            Learner: 'logistic'
```

Because `fitcecoc` implements 5-fold cross-validation, `CVMD1` contains a 5-by-1 cell array of `CompactClassificationECOC` models that the software trains on each fold. The `BinaryLearners` property of each `CompactClassificationECOC` model contains the `ClassificationLinear` models. The number of `ClassificationLinear` models within each compact ECOC model depends on the number of distinct labels and coding design. Because `Lambda` is a sequence of regularization strengths, you can think of `CLMdl1` as 11 models, one for each regularization strength in `Lambda`.

Determine how well the models generalize by plotting the averages of the 5-fold classification error for each regularization strength. Identify the regularization strength that minimizes the generalization error over the grid.

```
ce = kfoldLoss(CVMdl);
figure;
plot(log10(Lambda), log10(ce))
[~, minCEIdx] = min(ce);
minLambda = Lambda(minCEIdx);
hold on
plot(log10(minLambda), log10(ce(minCEIdx)), 'ro');
ylabel('log_{10} 5-fold classification error')
xlabel('log_{10} Lambda')
legend('MSE', 'Min classification error')
hold off
```



Train an ECOC model composed of linear classification model using the entire data set, and specify the minimal regularization strength.

```
t = templateLinear('Learner','logistic','Solver','sparsa',...
    'Regularization','lasso','Lambda',minLambda,'GradientTolerance',1e-8);
MdlFinal = fitcecoc(X,Y,'Learners',t,'ObservationsIn','columns');
```

To estimate labels for new observations, pass `MdlFinal` and the new data to `predict`.

- “Quick Start Parallel Computing for Statistics and Machine Learning Toolbox” on page 30-2



## Definitions

### Binary Loss

A binary loss is a function of the class and classification score that determines how well a binary learner classifies an observation into the class.

Let:

- $m_{kj}$  be element  $(k,j)$  of the coding design matrix  $M$  (i.e., the code corresponding to class  $k$  of binary learner  $j$ )
- $s_j$  be the score of binary learner  $j$  for an observation
- $g$  be the binary loss function
- $\hat{k}$  be the predicted class for the observation

In loss-based decoding [15], the class producing the minimum sum of the binary losses over binary learners determines the predicted class of an observation, that is,

$$\hat{k} = \operatorname{argmin}_k \sum_{j=1}^L |m_{kj}| g(m_{kj}, s_j).$$

In loss-weighted decoding [15], the class producing the minimum average of the binary losses over binary learners determines the predicted class of an observation, that is,

$$\hat{k} = \operatorname{argmin}_k \frac{\sum_{j=1}^L |m_{kj}| g(m_{kj}, s_j)}{\sum_{j=1}^L |m_{kj}|}.$$

Allwein et al. [2] suggest that loss-weighted decoding improves classification accuracy by keeping loss values for all classes in the same dynamic range.

This table summarizes the supported loss functions, where  $y_j$  is a class label for a particular binary learner (in the set  $\{-1,1,0\}$ ),  $s_j$  is the score for observation  $j$ , and  $g(y_j, s_j)$ .

Value	Description	Score Domain	$g(y_j, s_j)$
'binodeviance'	Binomial deviance	$(-\infty, \infty)$	$\log[1 + \exp(-2y_j s_j)] / [2\log(2)]$

Value	Description	Score Domain	$g(y, s_j)$
'exponential'	Exponential	$(-\infty, \infty)$	$\exp(-y_j s_j)/2$
'hamming'	Hamming	$[0, 1]$ or $(-\infty, \infty)$	$[1 - \text{sign}(y_j s_j)]/2$
'hinge'	Hinge	$(-\infty, \infty)$	$\max(0, 1 - y_j s_j)/2$
'linear'	Linear	$(-\infty, \infty)$	$(1 - y_j s_j)/2$
'logit'	Logistic	$(-\infty, \infty)$	$\log[1 + \exp(-y_j s_j)]/[2\log(2)]$
'quadratic'	Quadratic	$[0, 1]$	$[1 - y_j(2s_j - 1)]^2/2$

The software normalizes the binary losses such that the loss is 0.5 when  $y_j = 0$ , and aggregates using the average of the binary learners [2].

Do not confuse the binary loss with the overall classification loss (specified by the `LossFun` name-value pair argument of `predict` and `loss`), e.g., classification error, which measures how well an ECOC classifier performs as a whole.

## Classification Error

The classification error is a binary classification error measure that has the form

$$L = \frac{\sum_{j=1}^n w_j e_j}{\sum_{j=1}^n w_j},$$

where:

- $w_j$  is the weight for observation  $j$ . The software renormalizes the weights to sum to 1.
- $e_j = 1$  if the predicted class of observation  $j$  differs from its true class, and 0 otherwise.

In other words, it is the proportion of observations that the classifier misclassifies.

## References

- [1] Allwein, E., R. Schapire, and Y. Singer. "Reducing multiclass to binary: A unifying approach for margin classifiers." *Journal of Machine Learning Research*. Vol. 1, 2000, pp. 113–141.

- [2] Escalera, S., O. Pujol, and P. Radeva. “On the decoding process in ternary error-correcting output codes.” *IEEE Transactions on Pattern Analysis and Machine Intelligence*. Vol. 32, Issue 7, 2010, pp. 120–134.
- [3] Escalera, S., O. Pujol, and P. Radeva. “Separability of ternary codes for sparse designs of error-correcting output codes.” *Pattern Recogn.* Vol. 30, Issue 3, 2009, pp. 285–297.

## See Also

ClassificationECOC | ClassificationLinear |  
ClassificationPartitionedLinearECOC | fitcecoc | kfoldPredict | loss |  
statset

## Topics

“Quick Start Parallel Computing for Statistics and Machine Learning Toolbox” on page 30-2

“Reproducibility in Parallel Statistical Computations” on page 30-14

“Concepts of Parallel Computing in Statistics and Machine Learning Toolbox” on page 30-8

**Introduced in R2016a**

## kfoldLoss

**Class:** ClassificationPartitionedModel

Classification loss for observations not used for training

### Syntax

```
L = kfoldLoss(obj)
L = kfoldLoss(obj, Name, Value)
```

### Description

`L = kfoldLoss(obj)` returns loss obtained by cross-validated classification model `obj`. For every fold, this method computes classification loss for in-fold observations using a model trained on out-of-fold observations.

`L = kfoldLoss(obj, Name, Value)` calculates loss with additional options specified by one or more `Name, Value` pair arguments. You can specify several name-value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### Input Arguments

**obj**

Object of class `ClassificationPartitionedModel`.

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

**folds**

Indices of folds ranging from 1 to `obj.KFold`. Use only these folds for predictions.

**Default:** `1:obj.KFold`

 **lossfun**

Loss function, specified as the comma-separated pair consisting of 'LossFun' and a built-in, loss-function name or function handle.

- The following lists available loss functions. Specify one using its corresponding character vector.

Value	Description
'binodeviance'	Binomial deviance
'classiferror'	Classification error
'exponential'	Exponential
'hinge'	Hinge
'logit'	Logistic
'mincost'	Minimal expected misclassification cost (for classification scores that are posterior probabilities)
'quadratic'	Quadratic

'mincost' is appropriate for classification scores that are posterior probabilities. All models use posterior probabilities as classification scores by default except SVM models. You can specify to use posterior probabilities as classification scores for SVM models by setting 'FitPosterior', true when you cross-validate the model using `fitcsvm`.

- Specify your own function using function handle notation.

Suppose that  $n$  be the number of observations in  $X$  and  $K$  be the number of distinct classes (`numel(obj.ClassNames)`, `obj` is the input model). Your function must have this signature

```
lossvalue = lossfun(C,S,W,Cost)
```

where:

- The output argument `lossvalue` is a scalar.
- You choose the function name (`lossfun`).
- `C` is an  $n$ -by- $K$  logical matrix with rows indicating which class the corresponding observation belongs. The column order corresponds to the class order in `obj.ClassNames`.

Construct `C` by setting  $C(p, q) = 1$  if observation  $p$  is in class  $q$ , for each row. Set all other elements of row  $p$  to 0.

- `S` is an  $n$ -by- $K$  numeric matrix of classification scores. The column order corresponds to the class order in `obj.ClassNames`. `S` is a matrix of classification scores, similar to the output of `predict`.
- `W` is an  $n$ -by-1 numeric vector of observation weights. If you pass `W`, the software normalizes them to sum to 1.
- `Cost` is a  $K$ -by- $K$  numeric matrix of misclassification costs. For example, `Cost = ones(K) - eye(K)` specifies a cost of 0 for correct classification, and 1 for misclassification.

Specify your function using `'LossFun', @lossfun`.

For more details on loss functions, see “Classification Loss” on page 32-3250.

**Default:** `'classiferror'`

**mode**

A character vector for determining the output of `kfoldLoss`:

- `'average'` — `L` is a scalar, the loss averaged over all folds.
- `'individual'` — `L` is a vector of length `obj.KFold`, where each entry is the loss for a fold.

**Default:** `'average'`

## Output Arguments

**L**

Loss, by default the fraction of misclassified data. **L** can be a vector, and can mean different things, depending on the name-value pair settings.

## Examples

### Estimate Cross-Validated Classification Error

Load the `ionosphere` data set.

```
load ionosphere
```

Grow a classification tree.

```
tree = fitctree(X,Y);
```

Cross-validate the classification tree using 10-fold cross-validation.

```
cvtree = crossval(tree);
```

Estimate the cross-validated classification error.

```
L = kfoldLoss(cvtree)
```

```
L =
```

0.1111

## Definitions

### Classification Loss

Classification loss functions measure the predictive inaccuracy of classification models. When comparing the same type of loss among many models, lower loss indicates a better predictive model.

Suppose that:

- $L$  is the weighted average classification loss.
- $n$  is the sample size.
- For binary classification:
  - $y_j$  is the observed class label. The software codes it as  $-1$  or  $1$  indicating the negative or positive class, respectively.
  - $f(X_j)$  is the raw classification score for observation (row)  $j$  of the predictor data  $X$ .
  - $m_j = y_j f(X_j)$  is the classification score for classifying observation  $j$  into the class corresponding to  $y_j$ . Positive values of  $m_j$  indicate correct classification and do not contribute much to the average loss. Negative values of  $m_j$  indicate incorrect classification and contribute to the average loss.
- For algorithms that support multiclass classification (that is,  $K \geq 3$ ):
  - $y_j^*$  is a vector of  $K - 1$  zeros, and a  $1$  in the position corresponding to the true, observed class  $y_j$ . For example, if the true class of the second observation is the third class and  $K = 4$ , then  $y_2^* = [0 \ 0 \ 1 \ 0]'$ . The order of the classes corresponds to the order in the `ClassNames` property of the input model.
  - $f(X_j)$  is the length  $K$  vector of class scores for observation  $j$  of the predictor data  $X$ . The order of the scores corresponds to the order of the classes in the `ClassNames` property of the input model.
  - $m_j = y_j^{*'} f(X_j)$ . Therefore,  $m_j$  is the scalar classification score that the model predicts for the true, observed class.



- The weight for observation  $j$  is  $w_j$ . The software normalizes the observation weights so that they sum to the corresponding prior class probability. The software also normalizes the prior probabilities so they sum to 1. Therefore,

$$\sum_{j=1}^n w_j = 1.$$

The supported loss functions are:

- Binomial deviance, specified using 'LossFun', 'binodeviance'. Its equation is

$$L = \sum_{j=1}^n w_j \log \{1 + \exp[-2m_j]\}.$$

- Exponential loss, specified using 'LossFun', 'exponential'. Its equation is

$$L = \sum_{j=1}^n w_j \exp(-m_j).$$

- Classification error, specified using 'LossFun', 'classiferror'. It is the weighted fraction of misclassified observations, with equation

$$L = \sum_{j=1}^n w_j I\{\hat{y}_j \neq y_j\}.$$

$\hat{y}_j$  is the class label corresponding to the class with the maximal posterior probability.  $I\{x\}$  is the indicator function.

- Hinge loss, specified using 'LossFun', 'hinge'. Its equation is

$$L = \sum_{j=1}^n w_j \max\{0, 1 - m_j\}.$$

- Logit loss, specified using 'LossFun', 'logit'. Its equation is

$$L = \sum_{j=1}^n w_j \log(1 + \exp(-m_j)).$$

- Minimal cost, specified using 'LossFun', 'mincost'. The software computes the weighted minimal cost using this procedure for observations  $j = 1, \dots, n$ :

- 1 Estimate the 1-by- $K$  vector of expected classification costs for observation  $j$

$$\gamma_j = f(X_j)'C.$$

$f(X_j)$  is the column vector of class posterior probabilities for binary and multiclass classification.  $C$  is the cost matrix the input model stores in the property `Cost`.

- 2 For observation  $j$ , predict the class label corresponding to the minimum, expected classification cost:

$$\hat{y}_j = \min_{j=1,\dots,K}(\gamma_j).$$

- 3 Using  $C$ , identify the cost incurred ( $c_j$ ) for making the prediction.

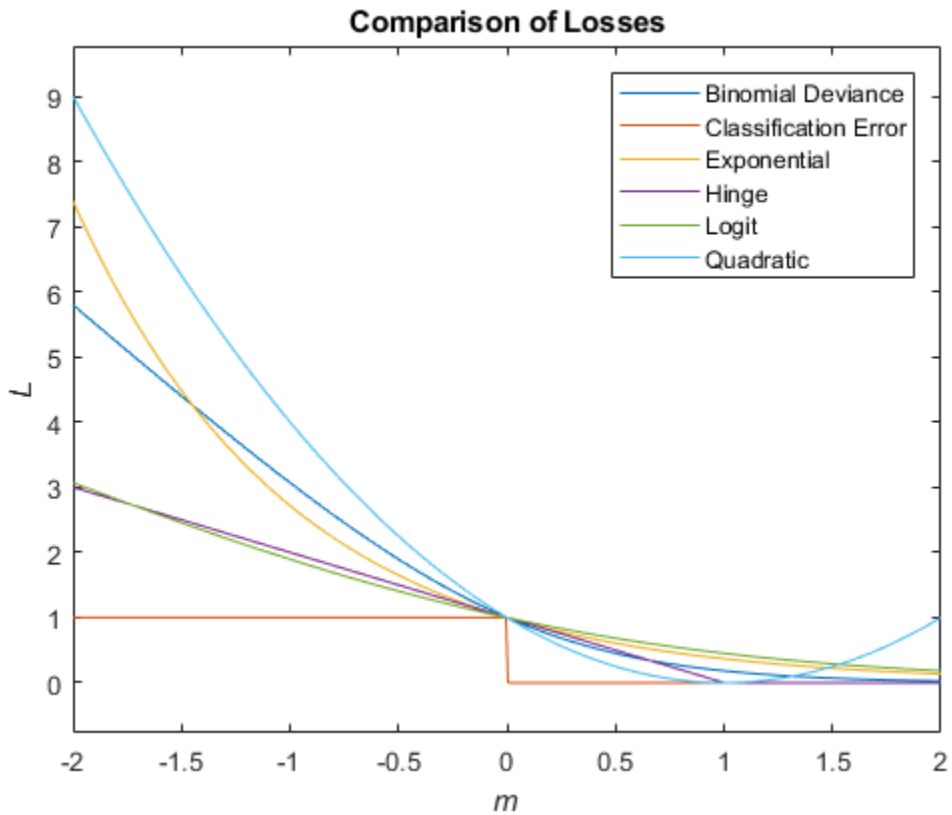
The weighted, average, minimum cost loss is

$$L = \sum_{j=1}^n w_j c_j.$$

- Quadratic loss, specified using `'LossFun', 'quadratic'`. Its equation is

$$L = \sum_{j=1}^n w_j (1 - m_j)^2.$$

This figure compares some of the loss functions for one observation over  $m$  (some functions are normalized to pass through  $[0,1]$ ).



## See Also

[ClassificationPartitionedModel](#) | [crossval](#) | [kfoldEdge](#) | [kfoldMargin](#) | [kfoldPredict](#) | [kfoldfun](#)

## Topics

“Examine Quality of KNN Classifier” on page 18-37

## kfoldLoss

**Class:** RegressionPartitionedEnsemble

Cross-validation loss of partitioned regression ensemble

### Syntax

```
L = kfoldLoss(cvens)
L = kfoldLoss(cvens, Name, Value)
```

### Description

`L = kfoldLoss(cvens)` returns the cross-validation loss of `cvens`.

`L = kfoldLoss(cvens, Name, Value)` returns cross-validation loss with additional options specified by one or more `Name, Value` pair arguments. You can specify several name-value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### Input Arguments

#### **cvens**

Object of class `RegressionPartitionedEnsemble`. Create `obj` with `fitrensemble` along with one of the cross-validation options: `'crossval'`, `'kfold'`, `'holdout'`, `'leaveout'`, or `'cvpartition'`. Alternatively, create `obj` from a regression ensemble with `crossval`.

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

**fold**

Indices of folds ranging from 1 to `cvens.KFold`. Use only these folds for predictions.

**Default:** `1:cvens.KFold`

**lossfun**

Function handle for loss function, or 'mse', meaning mean squared error. If you pass a function handle `fun`, `loss` calls it as

```
fun(Y,Yfit,W)
```

where `Y`, `Yfit`, and `W` are numeric vectors of the same length.

- `Y` is the observed response.
- `Yfit` is the predicted response.
- `W` is the observation weights.

The returned value `fun(Y,Yfit,W)` should be a scalar.

**Default:** 'mse'

**mode**

Character vector representing the meaning of the output `L`:

- 'ensemble' — `L` is a scalar value, the loss for the entire ensemble.
- 'individual' — `L` is a vector with one element per trained learner.
- 'cumulative' — `L` is a vector in which element `J` is obtained by using learners `1:J` from the input list of learners.

**Default:** 'ensemble'

## Output Arguments

**L**

The loss (mean squared error) between the observations in a fold when compared against predictions made with an ensemble trained on the out-of-fold data. **L** can be a vector, and can mean different things, depending on the name-value pair settings.

## Examples

### Find Cross-Validation Loss for Regression Ensemble

Find the cross-validation loss for a regression ensemble of the `carsmall` data.

Load the `carsmall` data set and select displacement, horsepower, and vehicle weight as predictors.

```
load carsmall
X = [Displacement Horsepower Weight];
```

Train an ensemble of regression trees.

```
rens = fitensemble(X,MPG);
```

Create a cross-validated ensemble from `rens` and find the k-fold cross-validation loss.

```
rng(10,'twister') % For reproducibility
cvrens = crossval(rens);
L = kfoldLoss(cvrens)
```

```
L = 28.7077
```

## See Also

[RegressionPartitionedEnsemble](#) | [kfoldPredict](#) | [loss](#)

# kfoldLoss

**Class:** RegressionPartitionedLinear

Regression loss for observations not used in training

## Syntax

```
L = kfoldLoss(CVMdl)
L = kfoldLoss(CVMdl, Name, Value)
```

## Description

Description

`L = kfoldLoss(CVMdl)` returns the cross-validated mean squared error (MSE) obtained by the cross-validated, linear regression model `CVMdl`. That is, for every fold, `kfoldLoss` estimates the regression loss for observations that it holds out when it trains using all other observations.

`L` contains a regression loss for each regularization strength in the linear regression models that compose `CVMdl`.

`L = kfoldLoss(CVMdl, Name, Value)` uses additional options specified by one or more `Name, Value` pair arguments. For example, indicate which folds to use for the loss calculation or specify the regression-loss function.

## Input Arguments

**`CVMdl` — Cross-validated, linear regression model**

RegressionPartitionedLinear model object

Cross-validated, linear regression model, specified as a RegressionPartitionedLinear model object. You can create a

RegressionPartitionedLinear model using `fitrlinear` and specifying any of the one of the cross-validation, name-value pair arguments, for example, `CrossVal`.

To obtain estimates, `RegressionPartitionedLinear.kfoldLoss` applies the same data used to cross-validate the linear regression model ( $X$  and  $Y$ ).

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### **Folds** — Fold indices to use for response prediction

`1:CVMD1.KFold` (default) | numeric vector of positive integers

Fold indices to use for response prediction, specified as the comma-separated pair consisting of 'Folds' and a numeric vector of positive integers. The elements of `Folds` must range from 1 through `CVMD1.KFold`.

Example: 'Folds', [1 4 10]

Data Types: `single` | `double`

### **LossFun** — Loss function

'mse' (default) | 'epsiloninsensitive' | function handle

Loss function, specified as the comma-separated pair consisting of 'LossFun' and a built-in, loss-function name or function handle.

- The following lists available loss functions. Specify one using its corresponding character vector. Also, in the table,  $f(x) = x\beta + b$ .
  - $\beta$  is a vector of  $p$  coefficients.
  - $x$  is an observation from  $p$  predictor variables.
  - $b$  is the scalar bias.



Value	Description
'epsiloninsensitive'	Epsilon-insensitive loss: $\ell[y, f(x)] = \max[0,  y - f(x)  - \epsilon]$
'mse'	MSE: $\ell[y, f(x)] = [y - f(x)]^2$

'epsiloninsensitive' is appropriate for SVM learners only.

- Specify your own function using function handle notation.

Let  $n$  be the number of observations in  $x$ . Your function must have this signature

```
lossvalue = lossfun(Y, Yhat, W)
```

where:

- The output argument `lossvalue` is a scalar.
- You choose the function name (`lossfun`).
- $Y$  is an  $n$ -dimensional vector of observed responses.  
RegressionPartitionedLinear.kfoldLoss passes the input argument  $Y$  in for  $Y$ .
- $Yhat$  is an  $n$ -dimensional vector of predicted responses, which is similar to the output of `predict`.
- $W$  is an  $n$ -by-1 numeric vector of observation weights.

Specify your function using 'LossFun', @`lossfun`.

Data Types: char | function\_handle

#### Mode — Loss aggregation level

'average' (default) | 'individual'

Loss aggregation level, specified as the comma-separated pair consisting of 'Mode' and 'average' or 'individual'.

Value	Description
'average'	Returns losses averaged over all folds
'individual'	Returns losses for each fold

Example: 'Mode', 'individual'

Data Types: char

## Output Arguments

### **L** — Cross-validated regression losses

numeric scalar | numeric vector | numeric matrix

Cross-validated regression losses, returned as a numeric scalar, vector, or matrix. The interpretation of **L** depends on `LossFun`.

Let  $R$  be the number of regularizations strengths is the cross-validated models (stored in `numel(CVMdl.Trained{1}.Lambda)`) and  $F$  be the number of folds (stored in `CVMdl.KFold`).

- If `Mode` is 'average', then **L** is a 1-by- $R$  vector.  $L(j)$  is the average regression loss over all folds of the cross-validated model that uses regularization strength  $j$ .
- Otherwise, **L** is an  $F$ -by- $R$  matrix.  $L(i, j)$  is the regression loss for fold  $i$  of the cross-validated model that uses regularization strength  $j$ .

To estimate **L**, `kfoldLoss` uses the data that created `CVMdl` (see  $X$  and  $Y$ ).

## Examples

### Estimate $k$ -Fold Mean Squared Error

Simulate 10000 observations from this model

$$y = x_{100} + 2x_{200} + e.$$

- $X = \{x_1, \dots, x_{1000}\}$  is a 10000-by-1000 sparse matrix with 10% nonzero standard normal elements.
- $e$  is random normal error with mean 0 and standard deviation 0.3.

```
rng(1) % For reproducibility
n = 1e4;
d = 1e3;
```

```

nz = 0.1;
X = sprandn(n,d,nz);
Y = X(:,100) + 2*X(:,200) + 0.3*randn(n,1);

```

Cross-validate a linear regression model using SVM learners.

```

rng(1); % For reproducibility
CVMdl = fitrlinear(X,Y,'CrossVal','on');

```

CVMdl is a RegressionPartitionedLinear model. By default, the software implements 10-fold cross validation. You can alter the number of folds using the 'KFold' name-value pair argument.

Estimate the average of the test-sample MSEs.

```
mse = kfoldLoss(CVMdl)
```

```

mse =

    0.1735

```

Alternatively, you can obtain the per-fold MSEs by specifying the name-value pair 'Mode','individual' in kfoldLoss.

### Specify Custom Regression Loss

Simulate data as in “Estimate k-Fold Mean Squared Error” on page 32-3260.

```

rng(1) % For reproducibility
n = 1e4;
d = 1e3;
nz = 0.1;
X = sprandn(n,d,nz);
Y = X(:,100) + 2*X(:,200) + 0.3*randn(n,1);
X = X'; % Put observations in columns for faster training

```

Cross-validate a linear regression model using 10-fold cross-validation. Optimize the objective function using SpARSA.

```

CVMdl = fitrlinear(X,Y,'CrossVal','on','ObservationsIn','columns',...
    'Solver','sparsa');

```

CVMdl is a `RegressionPartitionedLinear` model. It contains the property `Trained`, which is a 10-by-1 cell array holding `RegressionLinear` models that the software trained using the training set.

Create an anonymous function that measures Huber loss ( $\delta = 1$ ), that is,

$$L = \frac{1}{\sum w_j} \sum_{j=1}^n w_j \ell_j, \text{ where}$$

$$\ell_j = \begin{cases} 0.5\hat{e}^2 & \text{for } |\hat{e}| \leq 1 \\ |\hat{e}| - 0.5 & \text{otherwise} \end{cases} .$$

$\hat{e}_j$  is the residual for observation  $j$ . Custom loss functions must be written in a particular form. For rules on writing a custom loss function, see the `LossFun` name-value pair argument.

```
huberloss = @(Y,Yhat,W) sum(W.*((0.5*(abs(Y-Yhat)<=1).*(Y-Yhat).^2) + ...
    ((abs(Y-Yhat)>1).*abs(Y-Yhat)-0.5)))/sum(W);
```

Estimate the average Huber loss over the folds. Also, obtain the Huber loss for each fold.

```
mseAve = kfoldLoss(CVMdl, 'LossFun', huberloss)
mseFold = kfoldLoss(CVMdl, 'LossFun', huberloss, 'Mode', 'individual')
```

```
mseAve =
    -0.4447
```

```
mseFold =
    -0.4454
    -0.4479
    -0.4454
    -0.4468
    -0.4435
    -0.4429
    -0.4461
    -0.4422
    -0.4440
    -0.4427
```

## Find Good Lasso Penalty Using Cross-Validation

To determine a good lasso-penalty strength for a linear regression model that uses least squares, implement 5-fold cross-validation.

Simulate 10000 observations from this model

$$y = x_{100} + 2x_{200} + e.$$

- $X = \{x_1, \dots, x_{1000}\}$  is a 10000-by-1000 sparse matrix with 10% nonzero standard normal elements.
- $e$  is random normal error with mean 0 and standard deviation 0.3.

```
rng(1) % For reproducibility
n = 1e4;
d = 1e3;
nz = 0.1;
X = sprandn(n,d,nz);
Y = X(:,100) + 2*X(:,200) + 0.3*randn(n,1);
```

Create a set of 15 logarithmically-spaced regularization strengths from  $10^{-5}$  through  $10^{-1}$ .

```
Lambda = logspace(-5,-1,15);
```

Cross-validate the models. To increase execution speed, transpose the predictor data and specify that the observations are in columns. Optimize the objective function using SpARSA.

```
X = X';
CVMdl = fitrlinear(X,Y,'ObservationsIn','columns','KFold',5,'Lambda',Lambda,...
    'Learner','leastsquares','Solver','sparsa','Regularization','lasso');

numCLModels = numel(CVMdl.Trained)
```

```
numCLModels =
```

```
5
```

CVMdl is a `RegressionPartitionedLinear` model. Because `fitrlinear` implements 5-fold cross-validation, CVMdl contains 5 `RegressionLinear` models that the software trains on each fold.

Display the first trained linear regression model.

```
Mdl1 = CVMdl.Trained{1}
```

```
Mdl1 =
```

```
RegressionLinear
  ResponseName: 'Y'
  ResponseTransform: 'none'
  Beta: [1000x15 double]
  Bias: [1x15 double]
  Lambda: [1x15 double]
  Learner: 'leastsquares'
```

Mdl1 is a `RegressionLinear` model object. `fitrlinear` constructed Mdl1 by training on the first four folds. Because `Lambda` is a sequence of regularization strengths, you can think of Mdl1 as 11 models, one for each regularization strength in `Lambda`.

Estimate the cross-validated MSE.

```
mse = kfoldLoss(CVMdl);
```

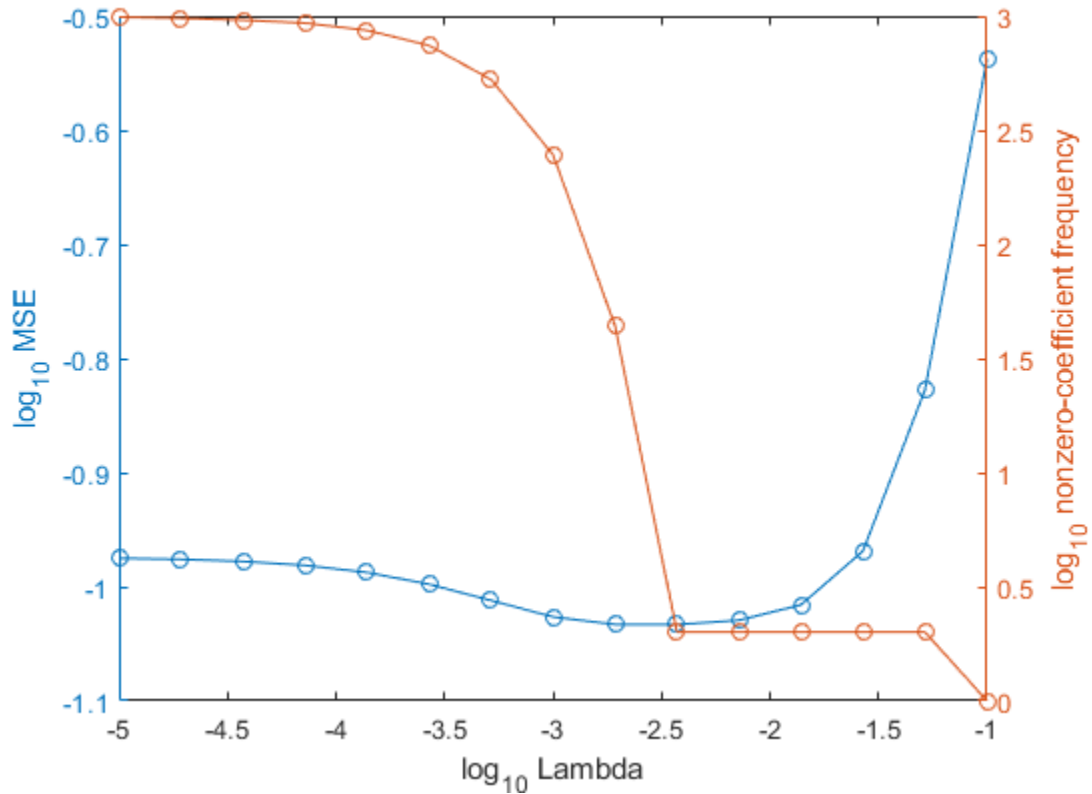
Higher values of `Lambda` lead to predictor variable sparsity, which is a good quality of a regression model. For each regularization strength, train a linear regression model using the entire data set and the same options as when you cross-validated the models. Determine the number of nonzero coefficients per model.

```
Mdl = fitrlinear(X,Y,'ObservationsIn','columns','Lambda',Lambda,...
  'Learner','leastsquares','Solver','sparsa','Regularization','lasso');
numNZCoeff = sum(Mdl.Beta~=0);
```

In the same figure, plot the cross-validated MSE and frequency of nonzero coefficients for each regularization strength. Plot all variables on the log scale.

```
figure;
[h,hL1,hL2] = plotyy(log10(Lambda),log10(mse),...
  log10(Lambda),log10(numNZCoeff));
hL1.Marker = 'o';
hL2.Marker = 'o';
ylabel(h(1),'log_{10} MSE')
ylabel(h(2),'log_{10} nonzero-coefficient frequency')
```

```
xlabel('log10 Lambda')
hold off
```



Choose the index of the regularization strength that balances predictor variable sparsity and low MSE (for example, `Lambda(10)`).

```
idxFinal = 10;
```

Extract the model with corresponding to the minimal MSE.

```
MdlFinal = selectModels(Mdl,idxFinal)
idxNZCoeff = find(MdlFinal.Beta~=0)
EstCoeff = Mdl.Beta(idxNZCoeff)
```

```
MdlFinal =  
  
  RegressionLinear  
    ResponseName: 'Y'  
    ResponseTransform: 'none'  
      Beta: [1000x1 double]  
      Bias: -0.0050  
      Lambda: 0.0037  
    Learner: 'leastsquares'
```

```
idxNZCoeff =  
  
  100  
  200
```

```
EstCoeff =  
  
  1.0051  
  1.9965
```

`MdlFinal` is a `RegressionLinear` model with one regularization strength. The nonzero coefficients `EstCoeff` are close to the coefficients that simulated the data.

## See Also

`RegressionLinear` | `RegressionPartitionedLinear` | `kfoldPredict` | `loss`

**Introduced in R2016a**



# kfoldLoss

**Class:** RegressionPartitionedModel

Cross-validation loss of partitioned regression model

## Syntax

```
L = kfoldLoss(cvmodel)
L = kfoldLoss(cvmodel, Name, Value)
```

## Description

`L = kfoldLoss(cvmodel)` returns the cross-validation loss of `cvmodel`.

`L = kfoldLoss(cvmodel, Name, Value)` returns cross-validation loss with additional options specified by one or more `Name, Value` pair arguments. You can specify several name-value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

## Input Arguments

### **cvmodel**

Object of class `RegressionPartitionedModel`. Create `obj` with `fitrtree` along with one of the cross-validation options: `'CrossVal'`, `'KFold'`, `'Holdout'`, `'Leaveout'`, or `'CVPartition'`. Alternatively, create `obj` from a regression tree with `crossval`.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

**fold**

Indices of folds ranging from 1 to `obj.KFold`. Use only these folds for predictions.

**Default:** `1:obj.KFold`

**lossfun**

Function handle for loss function or `'mse'`, meaning mean squared error. If you pass a function handle `fun`, `kfoldLoss` calls it as

```
fun(Y, Yfit, W)
```

where `Y`, `Yfit`, and `W` are numeric vectors of the same length.

- `Y` is the observed response.
- `Yfit` is the predicted response.
- `W` is the observation weights.

The returned value `fun(Y, Yfit, W)` should be a scalar.

**Default:** `'mse'`

**mode**

One of the following:

- `'average'` — `L` is the average loss over all folds.
- `'individual'` — `L` is a vector of the individual losses of in-fold observations trained on out-of-fold data.

**Default:** `'average'`

## Output Arguments

**L**

The loss (mean squared error) between the observations in a fold when compared against predictions made with a tree trained on the out-of-fold data. If `mode` is `'individual'`, `L` is a vector of the losses. If `mode` is `'average'`, `L` is the average loss.

## Examples

Construct a partitioned regression model, and examine the cross-validation losses for the folds:

```
load carsmall
XX = [Cylinders Displacement Horsepower Weight];
YY = MPG;
cvmodel = fitrtree(XX,YY,'crossval','on');
L = kfoldLoss(cvmodel,'mode','individual')
```

```
L =
    44.9635
    11.8525
    18.2046
     9.2965
    29.4329
    54.8659
    24.6446
     8.2085
    19.7593
    16.7394
```

## Alternatives

You can avoid constructing a cross-validated tree model by calling `cvLoss` instead of `kfoldLoss`. The cross-validated tree can save time if you are going to examine it more than once.

## See Also

`fitrtree` | `kfoldPredict` | `loss`

## kfoldMargin

**Class:** ClassificationPartitionedECOC

Classification margins for observations not used for training

### Syntax

```
margin = kfoldMargin(CVMdl)
margin = kfoldMargin(CVMdl, Name, Value)
```

### Description

`margin = kfoldMargin(CVMdl)` returns classification margins obtained by the cross-validated ECOC model (`ClassificationPartitionedECOC`) `CVMdl`. For every fold, this method computes classification margins for in-fold observations using a model trained on out-of-fold observations. `CVMdl.X` contains both sets of observations.

`margin = kfoldMargin(CVMdl, Name, Value)` returns classification margins with additional options specified by one or more `Name, Value` pair arguments.

For example, specify the binary learner loss function, decoding scheme, or verbosity level.

### Input Arguments

**CVMdl** — Cross-validated ECOC model

`ClassificationPartitionedECOC` model

Cross-validated ECOC model, specified as a `ClassificationPartitionedECOC` model. You can create a `ClassificationPartitionedECOC` model by:

- Passing a trained ECOC model (`ClassificationECOC`) to `crossval`
- Training an ECOC model using `fitcecoc` and setting any one of these cross-validation name-value pair arguments: `'CrossVal'`, `'KFold'`, `'Holdout'`, `'Leaveout'`, or `'CVPartition'`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### BinaryLoss — Binary learner loss function

'hamming' | 'linear' | 'logit' | 'exponential' | 'binodeviance' | 'hinge'  
| 'quadratic' | function handle

Binary learner loss function, specified as the comma-separated pair consisting of 'BinaryLoss' and a built-in, loss-function name or function handle.

- This table contains names and descriptions of the built-in functions, where  $y_j$  is a class label for a particular binary learner (in the set  $\{-1, 1, 0\}$ ),  $s_j$  is the score for observation  $j$ , and  $g(y_j, s_j)$  is the binary loss formula.

Value	Description	Score Domain	$g(y_j, s_j)$
'binodeviance'	Binomial deviance	$(-\infty, \infty)$	$\log[1 + \exp(-2y_j s_j)] / [2\log(2)]$
'exponential'	Exponential	$(-\infty, \infty)$	$\exp(-y_j s_j) / 2$
'hamming'	Hamming	$[0, 1]$ or $(-\infty, \infty)$	$[1 - \text{sign}(y_j s_j)] / 2$
'hinge'	Hinge	$(-\infty, \infty)$	$\max(0, 1 - y_j s_j) / 2$
'linear'	Linear	$(-\infty, \infty)$	$(1 - y_j s_j) / 2$
'logit'	Logistic	$(-\infty, \infty)$	$\log[1 + \exp(-y_j s_j)] / [2\log(2)]$
'quadratic'	Quadratic	$[0, 1]$	$[1 - y_j(2s_j - 1)]^2 / 2$

The software normalizes the binary losses such that the loss is 0.5 when  $y_j = 0$ . Also, the software calculates the mean binary loss for each class.

- For a custom binary loss function, e.g., `customFunction`, specify its function handle 'BinaryLoss', @`customFunction`.

`customFunction` should have this form:

```
bLoss = customFunction(M, s)
```

where:

- $M$  is the  $K$ -by- $L$  coding matrix stored in `Mdl.CodingMatrix`.
- $s$  is the 1-by- $L$  row vector of classification scores.
- `bLoss` is the classification loss. This scalar aggregates the binary losses for every learner in a particular class. For example, you can use the mean binary loss to aggregate the loss over the learners for each class.
- $K$  is the number of classes.
- $L$  is the number of binary learners.

For an example of passing a custom binary loss function, see “Predict Test-Sample Labels of ECOC Models Using Custom Binary Loss Function” on page 32-4933.

By default, if all binary learners are:

- SVMs or linear classification models of SVM learners, then `BinaryLoss` is 'hinge'
- Ensembles trained by `AdaboostM1` or `GentleBoost`, then `BinaryLoss` is 'exponential'
- Ensembles trained by `LogitBoost`, then `BinaryLoss` is 'binodeviance'
- Linear classification models of logistic regression learners, or you specify to predict class posterior probabilities (i.e., set 'FitPosterior', 1 in `fitcecoc`), then `BinaryLoss` is 'quadratic'

Otherwise, the default `BinaryLoss` is 'hamming'. To check the default value, use dot notation to display the `BinaryLoss` property of the trained model at the command line.

Example: `'BinaryLoss', 'binodeviance'`

Data Types: `char` | `function_handle`

#### **Decoding — Decoding scheme**

`'lossweighted'` (default) | `'lossbased'`

Decoding scheme that aggregates the binary losses on page 32-4941, specified as the comma-separated pair consisting of 'Decoding' and 'lossweighted' or 'lossbased'.

Example: `'Decoding', 'lossbased'`

Data Types: `char`

#### **Options — Estimation options**

`[]` (default) | structure array returned by `statset`

Estimation options, specified as the comma-separated pair consisting of 'Options' and a structure array returned by `statset`.

To invoke parallel computing:

- You need a Parallel Computing Toolbox license.
- Specify 'Options', `statset('UseParallel',1)`.

### **Verbose** — Verbosity level

0 (default) | 1

Verbosity level, specified as the comma-separated pair consisting of 'Verbose' and 0 or 1. Verbose controls the amount of diagnostic messages that the software displays in the Command Window.

If `Verbose` is 0, then the software does not display diagnostic messages. Otherwise, the software displays diagnostic messages.

Example: 'Verbose',1

Data Types: `single` | `double`

## Output Arguments

### **margin** — Classification margins

numeric vector

Classification margins on page 32-3278, returned as a numeric vector. `margin` is an  $n$ -by-1 vector, where each row is the margin of the corresponding observation, and  $n$  is the number of observations (i.e., `size(CVMdl.X,1)`).

Data Types: `single` | `double`

## Examples

### Estimate $k$ -Fold Cross-Validated Margins of ECOC Models

Load Fisher's iris data set.

```
load fisheriris
X = meas;
Y = categorical(species);
classOrder = unique(Y);
rng(1); % For reproducibility
```

Train an ECOC model using SVM binary classifiers and specify to cross validate. It is good practice to standardize the predictors and define the class order. Specify to standardize the predictors using an SVM template.

```
t = templateSVM('Standardize',1);
CVMdl = fitcecoc(X,Y,'CrossVal','on','Learners',t,'ClassNames',classOrder);
```

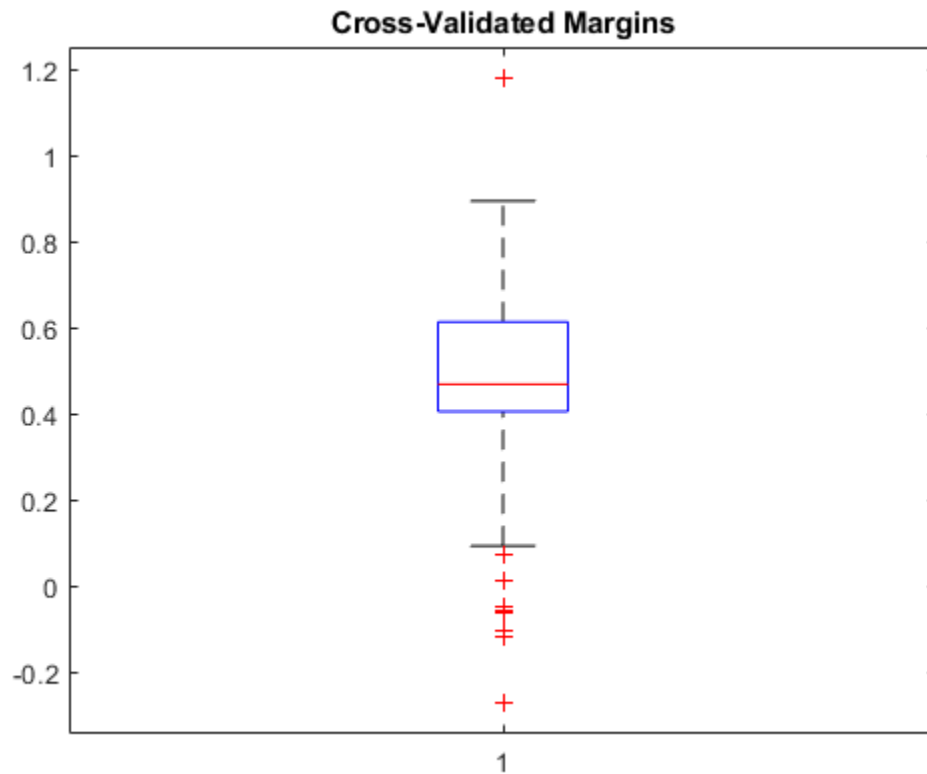
CVMdl is a ClassificationPartitionedModel model. By default, the software implements 10-fold cross validation. You can alter the number of folds using the 'KFold' name-value pair argument.

Estimate the out-of-fold margins. Display the distribution of the mnargins using a boxplot.

```
margin = kfoldMargin(CVMdl);

figure;
boxplot(margin);
title('Cross-Validated Margins')
```





An observation margin is the positive-class, negated loss minus the maximum negative-class, negated loss. Classifiers that yield relatively large margins are desirable.

### Select ECOC Model Features by Comparing Cross-Validation Margins

The classifier margin measures the average of the classifier margins. One way to perform feature selection is to compare cross-validation margins from multiple models. Based solely on this criterion, the classifier with the greater margins is the best classifier.

Load Fisher's iris data set.

```
load fisheriris
X = meas;
Y = categorical(species);
classOrder = unique(Y); % Class order
rng(1); % For reproducibility
```

Define these two data sets:

- fullX contains all predictors.
- partX contains the petal dimensions.

```
fullX = X;
partX = X(:,3:4);
```

Train an ECOC model using SVM binary classifiers for each predictor set, and specify to cross validate. It is good practice to standardize the predictors and define the class order. Specify to standardize the predictors using an SVM template.

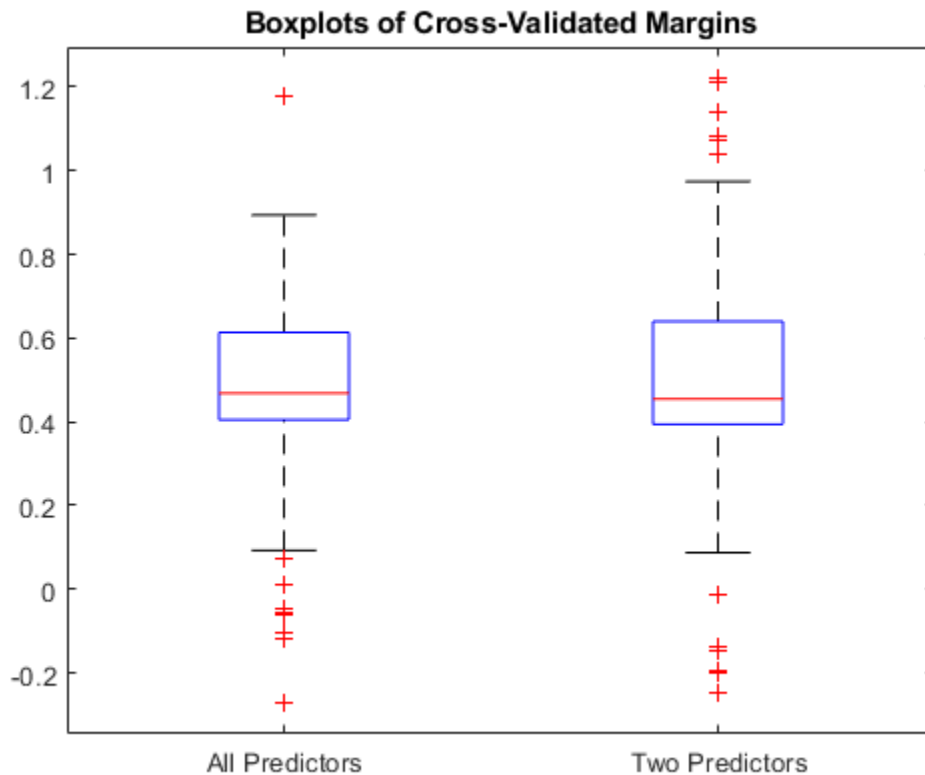
```
t = templateSVM('Standardize',1);
CVMdl = fitcecoc(fullX,Y,'CrossVal','on','Learners',t,...
    'ClassNames',classOrder);
PCVMdl = fitcecoc(partX,Y,'CrossVal','on','Learners',t,...
    'ClassNames',classOrder);
```

CVMdl and PCVMdl are ClassificationPartitionedECOC models. By default, the software implements 10-fold cross validation.

Estimate the test sample margin for each classifier. Specify to use loss-based decoding for aggregating the binary learner results. For each model, display the distribution of the margins using a boxplot.

```
fullMargins = kfoldMargin(CVMdl,'Decoding','lossbased');
partMargins = kfoldMargin(PCVMdl,'Decoding','lossbased');

figure;
boxplot([fullMargins partMargins],'Labels',{'All Predictors','Two Predictors'});
title('Boxplots of Cross-Validated Margins')
```



The margin distributions are approximately the same, but PCVMd1 is a less complex model, which might make it more desirable.

- “Quick Start Parallel Computing for Statistics and Machine Learning Toolbox” on page 30-2

## Definitions

### Classification Margin

The classification margins are, for each observation, the difference between the negative loss for the true class and maximal negative loss among the false classes. If the margins are on the same scale, then they serve as a classification confidence measure, i.e., among multiple classifiers, those that yield larger margins are better.

### Binary Loss

A binary loss is a function of the class and classification score that determines how well a binary learner classifies an observation into the class.

Let:

- $m_{kj}$  be element  $(k,j)$  of the coding design matrix  $M$  (i.e., the code corresponding to class  $k$  of binary learner  $j$ )
- $s_j$  be the score of binary learner  $j$  for an observation
- $g$  be the binary loss function
- $\hat{k}$  be the predicted class for the observation

In loss-based decoding [15], the class producing the minimum sum of the binary losses over binary learners determines the predicted class of an observation, that is,

$$\hat{k} = \operatorname{argmin}_k \sum_{j=1}^L |m_{kj}| g(m_{kj}, s_j).$$

In loss-weighted decoding [15], the class producing the minimum average of the binary losses over binary learners determines the predicted class of an observation, that is,

$$\hat{k} = \operatorname{argmin}_k \frac{\sum_{j=1}^L |m_{kj}| g(m_{kj}, s_j)}{\sum_{j=1}^L |m_{kj}|}.$$

Allwein et al. [2] suggest that loss-weighted decoding improves classification accuracy by keeping loss values for all classes in the same dynamic range.

This table summarizes the supported loss functions, where  $y_j$  is a class label for a particular binary learner (in the set  $\{-1,1,0\}$ ),  $s_j$  is the score for observation  $j$ , and  $g(y_j, s_j)$ .

Value	Description	Score Domain	$g(y_j, s_j)$
'binodeviance'	Binomial deviance	$(-\infty, \infty)$	$\log[1 + \exp(-2y_j s_j)] / [2\log(2)]$
'exponential'	Exponential	$(-\infty, \infty)$	$\exp(-y_j s_j) / 2$
'hamming'	Hamming	$[0, 1]$ or $(-\infty, \infty)$	$[1 - \text{sign}(y_j s_j)] / 2$
'hinge'	Hinge	$(-\infty, \infty)$	$\max(0, 1 - y_j s_j) / 2$
'linear'	Linear	$(-\infty, \infty)$	$(1 - y_j s_j) / 2$
'logit'	Logistic	$(-\infty, \infty)$	$\log[1 + \exp(-y_j s_j)] / [2\log(2)]$
'quadratic'	Quadratic	$[0, 1]$	$[1 - y_j(2s_j - 1)]^2 / 2$

The software normalizes the binary losses such that the loss is 0.5 when  $y_j = 0$ , and aggregates using the average of the binary learners [2].

Do not confuse the binary loss with the overall classification loss (specified by the `LossFun` name-value pair argument of `predict` and `loss`), e.g., classification error, which measures how well an ECOC classifier performs as a whole.

## References

- [1] Allwein, E., R. Schapire, and Y. Singer. "Reducing multiclass to binary: A unifying approach for margin classifiers." *Journal of Machine Learning Research*. Vol. 1, 2000, pp. 113–141.
- [2] Escalera, S., O. Pujol, and P. Radeva. "On the decoding process in ternary error-correcting output codes." *IEEE Transactions on Pattern Analysis and Machine Intelligence*. Vol. 32, Issue 7, 2010, pp. 120–134.
- [3] Escalera, S., O. Pujol, and P. Radeva. "Separability of ternary codes for sparse designs of error-correcting output codes." *Pattern Recogn.* Vol. 30, Issue 3, 2009, pp. 285–297.

## See Also

`ClassificationECOC` | `ClassificationPartitionedModel` | `fitcecoc` |  
`kfoldEdge` | `kfoldPredict` | `margin` | `statset`

## Topics

“Quick Start Parallel Computing for Statistics and Machine Learning Toolbox” on page 30-2

“Reproducibility in Parallel Statistical Computations” on page 30-14

“Concepts of Parallel Computing in Statistics and Machine Learning Toolbox” on page 30-8

# kfoldMargin

**Class:** ClassificationPartitionedLinear

Classification margins for observations not used in training

## Syntax

```
m = kfoldMargin(CVMdl)
```

## Description

`m = kfoldMargin(CVMdl)` returns the cross-validated classification margins on page 32-3289 obtained by the cross-validated, binary, linear classification model `CVMdl`. That is, for every fold, `kfoldMargin` estimates the classification margins for observations that it holds out when it trains using all other observations.

`m` contains classification margins for each regularization strength in the linear classification models that comprise `CVMdl`.

## Input Arguments

**CVMdl** — Cross-validated, binary, linear classification model

ClassificationPartitionedLinear model object

Cross-validated, binary, linear classification model, specified as a ClassificationPartitionedLinear model object. You can create a ClassificationPartitionedLinear model using `fitlinear` and specifying any one of the cross-validation, name-value pair arguments, for example, `CrossVal`.

To obtain estimates, ClassificationPartitionedLinear.kfoldMargin applies the same data used to cross-validate the linear classification model ( $X$  and  $Y$ ).

## Output Arguments

### **m** — Cross-validated classification margins

numeric vector | numeric matrix

Cross-validated classification margins on page 32-3289, returned as a numeric vector or matrix.

$m$  is  $n$ -by- $L$ , where  $n$  is the number of observations in the data that created `CVMD1` (see  $X$  and  $Y$ ) and  $L$  is the number of regularization strengths in `CVMD1` (that is, `numel(CVMD1.Trained{1}.Lambda)`).

$m(i, j)$  is the cross-validated classification margin of observation  $i$  using the linear classification model that has regularization strength `CVMD1.Trained{1}.Lambda(j)`.

Data Types: `single` | `double`

## Examples

### Estimate $k$ -Fold Cross-Validation Margins

Load the NLP data set.

```
load nlpdata
```

$X$  is a sparse matrix of predictor data, and  $Y$  is a categorical vector of class labels. There are more than two classes in the data.

The models should identify whether the word counts in a web page are from the Statistics and Machine Learning Toolbox™ documentation. So, identify the labels that correspond to the Statistics and Machine Learning Toolbox™ documentation web pages.

```
Ystats = Y == 'stats';
```

Cross-validate a binary, linear classification model that can identify whether the word counts in a documentation web page are from the Statistics and Machine Learning Toolbox™ documentation.

```
rng(1); % For reproducibility  
CVMD1 = fitclinear(X, Ystats, 'CrossVal', 'on');
```



CVMDL is a `ClassificationPartitionedLinear` model. By default, the software implements 10-fold cross validation. You can alter the number of folds using the 'KFold' name-value pair argument.

Estimate the cross-validated margins.

```
m = kfoldMargin(CVMDL);  
size(m)
```

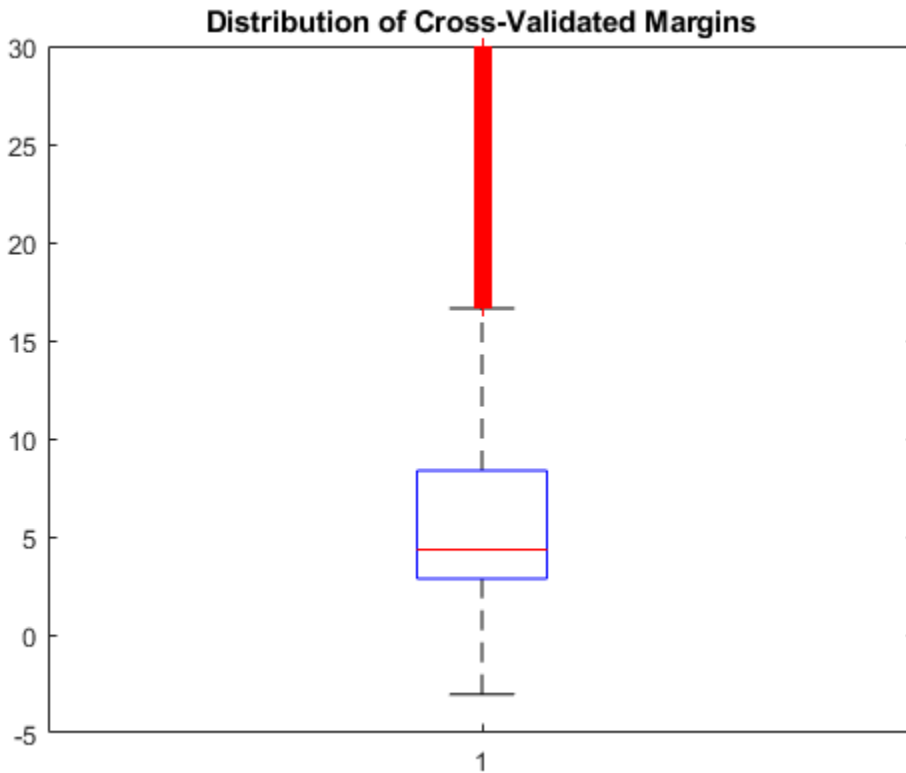
```
ans =
```

```
31572      1
```

`m` is a 31572-by-1 vector. `m(j)` is the average of the out-of-fold margins for observation `j`.

Plot the  $k$ -fold margins using box plots.

```
figure;  
boxplot(m);  
h = gca;  
h.YLim = [-5 30];  
title('Distribution of Cross-Validated Margins')
```



### Feature Selection Using $k$ -fold Margins

One way to perform feature selection is to compare  $k$ -fold margins from multiple models. Based solely on this criterion, the classifier with the larger margins is the better classifier.

Load the NLP data set. Preprocess the data as in “Estimate  $k$ -Fold Cross-Validation Margins” on page 32-3282.

```
load nlpdata
Ystats = Y == 'stats';
X = X';
```

Create these two data sets:

- `fullX` contains all predictors.
- `partX` contains 1/2 of the predictors chosen at random.

```
rng(1); % For reproducibility
p = size(X,1); % Number of predictors
halfPredIdx = randsample(p,ceil(0.5*p));
fullX = X;
partX = X(halfPredIdx,:);
```

Cross-validate two binary, linear classification models: one that uses the all of the predictors and one that uses half of the predictors. Optimize the objective function using `SparSA`, and indicate that observations correspond to columns.

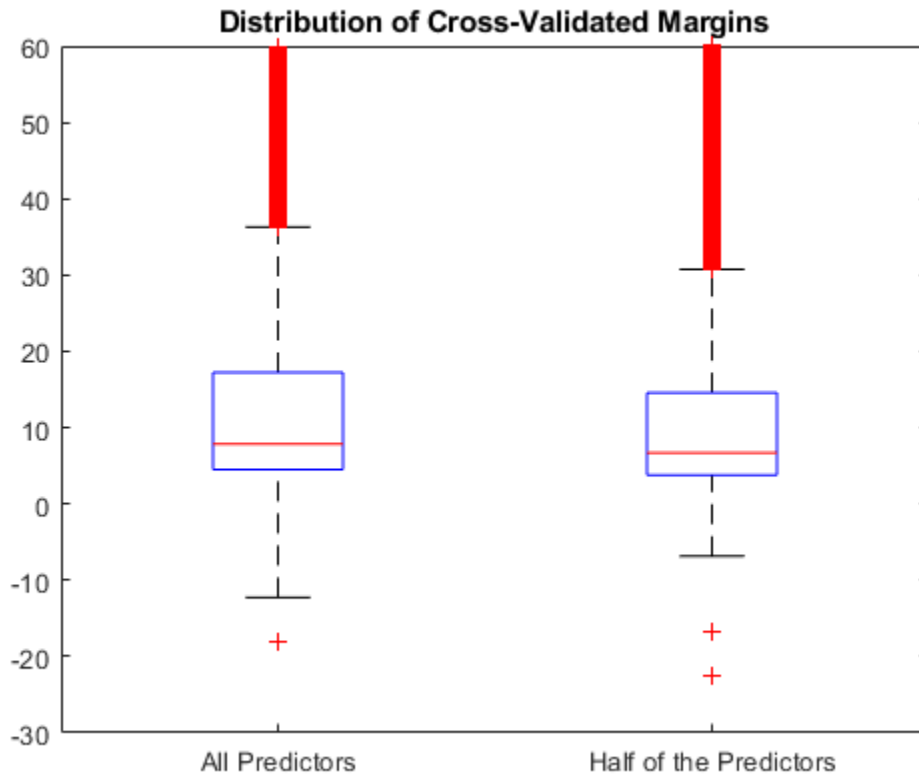
```
CVMD1 = fitclinear(fullX,Ystats,'CrossVal','on','Solver','sparsa',...
    'ObservationsIn','columns');
PCVMD1 = fitclinear(partX,Ystats,'CrossVal','on','Solver','sparsa',...
    'ObservationsIn','columns');
```

`CVMD1` and `PCVMD1` are `ClassificationPartitionedLinear` models.

Estimate the  $k$ -fold margins for each classifier. Plot the distribution of the  $k$ -fold margins sets using box plots.

```
fullMargins = kfoldMargin(CVMD1);
partMargins = kfoldMargin(PCVMD1);

figure;
boxplot([fullMargins partMargins],'Labels',...
    {'All Predictors','Half of the Predictors'});
h = gca;
h.YLim = [-30 60];
title('Distribution of Cross-Validated Margins')
```



The distributions of the margins of the two classifiers are similar.

### Find Good Lasso Penalty Using $k$ -fold Margins

To determine a good lasso-penalty strength for a linear classification model that uses a logistic regression learner, compare distributions of  $k$ -fold margins.

Load the NLP data set. Preprocess the data as in “Estimate  $k$ -Fold Cross-Validation Margins” on page 32-3282.

```
load nlpdata
Ystats = Y == 'stats';
X = X';
```

Create a set of 11 logarithmically-spaced regularization strengths from  $10^{-8}$  through  $10^1$ .

```
Lambda = logspace(-8,1,11);
```

Cross-validate a binary, linear classification model using 5-fold cross-validation and that uses each of the regularization strengths. Optimize the objective function using SpaRSA. Lower the tolerance on the gradient of the objective function to  $1e-8$ .

```
rng(10); % For reproducibility
CVMdl = fitlinear(X,Ystats,'ObservationsIn','columns','KFold',5,...
    'Learner','logistic','Solver','sparsa','Regularization','lasso',...
    'Lambda',Lambda,'GradientTolerance',1e-8)
```

```
CVMdl =
```

```
classreg.learning.partition.ClassificationPartitionedLinear
  CrossValidatedModel: 'Linear'
    ResponseName: 'Y'
  NumObservations: 31572
    KFold: 5
    Partition: [1x1 cvpartition]
    ClassNames: [0 1]
  ScoreTransform: 'none'
```

CVMdl is a ClassificationPartitionedLinear model. Because fitlinear implements 5-fold cross-validation, CVMdl contains 5 ClassificationLinear models that the software trains on each fold.

Estimate the  $k$ -fold margins for each regularization strength.

```
m = kfoldMargin(CVMdl);
size(m)
```

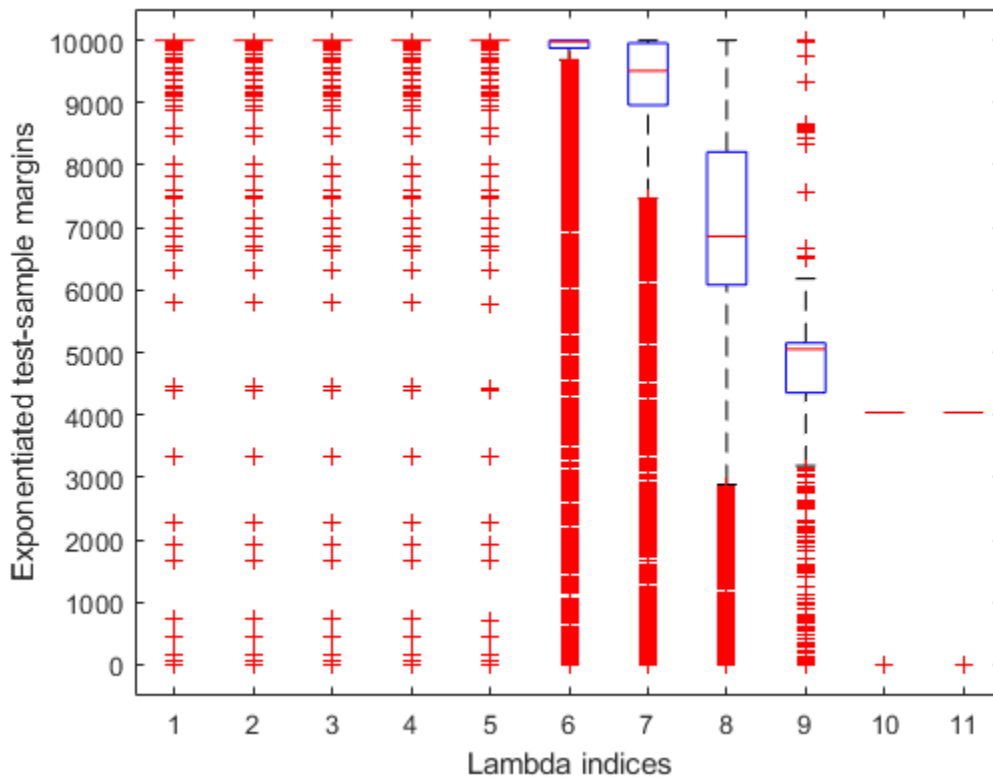
```
ans =
```

```
31572    11
```

$m$  is a 31572-by-11 matrix of cross-validated margins for each observation. The columns correspond to the regularization strengths.

Plot the  $k$ -fold margins for each regularization strength. Because logistic regression scores are in  $[0,1]$ , margins are in  $[-1,1]$ . Rescale the margins to help identify the regularization strength that maximizes the margins over the grid.

```
figure;
boxplot(10000.^m)
ylabel('Exponentiated test-sample margins')
xlabel('Lambda indices')
```



Several values of `Lambda` yield  $k$ -fold margin distributions that are compacted near **10000**<sup>1</sup>. Higher values of `lambda` lead to predictor variable sparsity, which is a good quality of a classifier.

Choose the regularization strength that occurs just before the centers of the  $k$ -fold margin distributions start decreasing.

```
LambdaFinal = Lambda(5);
```

Train a linear classification model using the entire data set and specify the desired regularization strength.

```
MdlFinal = fitclinear(X,Ystats,'ObservationsIn','columns',...
    'Learner','logistic','Solver','sparsa','Regularization','lasso',...
    'Lambda',LambdaFinal);
```

To estimate labels for new observations, pass `MdlFinal` and the new data to `predict`.

## Definitions

### Classification Margin

The classification margins for binary classification are, for each observation, the difference between the classification score for the true class and the classification score for the false class.

The software defines the classification margin for binary classification as

$$m = 2yf(x).$$

$x$  is an observation. If the true label of  $x$  is the positive class, then  $y$  is 1, and  $-1$  otherwise.  $f(x)$  is the positive-class classification score for the observation  $x$ . The literature commonly defines the margin as  $m = yf(x)$ .

If the margins are on the same scale, then they serve as a classification confidence measure, i.e., among multiple classifiers, those that yield larger margins are better.

### Classification Score

For linear classification models, the raw classification score for classifying the observation  $x$ , a row vector, into the positive class is defined by

$$f_j(x) = x\beta_j + b_j.$$

For the model with regularization strength  $j$ ,  $\beta_j$  is the estimated column vector of coefficients (the model property `Beta(:, j)`) and  $b_j$  is the estimated, scalar bias (the model property `Bias(j)`).

The raw classification score for classifying  $x$  into the negative class is  $-f(x)$ . The software classifies observations into the class that yields the positive score.

If the linear classification model consists of logistic regression learners, then the software applies the 'logit' score transformation to the raw classification scores (see `ScoreTransform`).

## See Also

`ClassificationLinear` | `ClassificationPartitionedLinear` | `kfoldEdge` | `kfoldPredict` | `margin`

**Introduced in R2016a**



# kfoldMargin

**Class:** ClassificationPartitionedLinearECOC

Classification margins for observations not used in training

## Syntax

```
m = kfoldMargin(CVMdl)
m = kfoldMargin(CVMdl, Name, Value)
```

## Description

`m = kfoldMargin(CVMdl)` returns the cross-validated classification margins on page 32-3303 obtained by `CVMdl`, which is a cross-validated, error-correcting output codes (ECOC) model composed of linear classification models. That is, for every fold, `kfoldMargin` estimates the classification margins for observations that it holds out when it trains using all other observations.

`m` contains classification margins for each regularization strength in the linear classification models that comprise `CVMdl`.

`m = kfoldMargin(CVMdl, Name, Value)` uses additional options specified by one or more `Name, Value` pair arguments. For example, specify a decoding scheme or verbosity level.

## Input Arguments

**CVMdl** — Cross-validated, ECOC model composed of linear classification models

ClassificationPartitionedLinearECOC model object

Cross-validated, ECOC model composed of linear classification models, specified as a ClassificationPartitionedLinearECOC model object. You can create a ClassificationPartitionedLinearECOC model using `fitcecoc` and by:

- 1 Specifying any one of the cross-validation, name-value pair arguments, for example, `CrossVal`
- 2 Setting the name-value pair argument `Learners` to `'linear'` or a linear classification model template returned by `templateLinear`

To obtain estimates, `ClassificationPartitionedLinearECOC.kfoldMargin` applies the same data used to cross-validate the ECOC model ( $X$  and  $Y$ ).

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### BinaryLoss — Binary learner loss function

`'hamming'` | `'linear'` | `'logit'` | `'exponential'` | `'binodeviance'` | `'hinge'`  
| `'quadratic'` | function handle

Binary learner loss function, specified as the comma-separated pair consisting of `'BinaryLoss'` and a built-in, loss-function name or function handle.

- This table contains names and descriptions of the built-in functions, where  $y_j$  is a class label for a particular binary learner (in the set  $\{-1, 1, 0\}$ ),  $s_j$  is the score for observation  $j$ , and  $g(y_j, s_j)$  is the binary loss formula.

Value	Description	Score Domain	$g(y_j, s_j)$
<code>'binodeviance'</code>	Binomial deviance	$(-\infty, \infty)$	$\log[1 + \exp(-2y_j s_j)] / [2\log(2)]$
<code>'exponential'</code>	Exponential	$(-\infty, \infty)$	$\exp(-y_j s_j) / 2$
<code>'hamming'</code>	Hamming	$[0, 1]$ or $(-\infty, \infty)$	$[1 - \text{sign}(y_j s_j)] / 2$
<code>'hinge'</code>	Hinge	$(-\infty, \infty)$	$\max(0, 1 - y_j s_j) / 2$
<code>'linear'</code>	Linear	$(-\infty, \infty)$	$(1 - y_j s_j) / 2$
<code>'logit'</code>	Logistic	$(-\infty, \infty)$	$\log[1 + \exp(-y_j s_j)] / [2\log(2)]$
<code>'quadratic'</code>	Quadratic	$[0, 1]$	$[1 - y_j(2s_j - 1)]^2 / 2$

The software normalizes the binary losses such that the loss is 0.5 when  $y_j = 0$ . Also, the software calculates the mean binary loss for each class.

- For a custom binary loss function, e.g., `customFunction`, specify its function handle `'BinaryLoss', @customFunction`.

`customFunction` should have this form

```
bLoss = customFunction(M,s)
```

where:

- `M` is the  $K$ -by- $L$  coding matrix stored in `Mdl.CodingMatrix`.
- `s` is the 1-by- $L$  row vector of classification scores.
- `bLoss` is the classification loss. This scalar aggregates the binary losses for every learner in a particular class. For example, you can use the mean binary loss to aggregate the loss over the learners for each class.
- $K$  is the number of classes.
- $L$  is the number of binary learners.

For an example of passing a custom binary loss function, see “Predict Test-Sample Labels of ECOC Models Using Custom Binary Loss Function” on page 32-4933.

By default, if all binary learners are linear classification models using:

- SVM, then `BinaryLoss` is `'hinge'`
- Logistic regression, then `BinaryLoss` is `'quadratic'`

Example: `'BinaryLoss', 'binodeviance'`

Data Types: `char` | `function_handle`

### Decoding — Decoding scheme

`'lossweighted'` (default) | `'lossbased'`

Decoding scheme that aggregates the binary losses on page 32-4941, specified as the comma-separated pair consisting of `'Decoding'` and `'lossweighted'` or `'lossbased'`.

Example: `'Decoding', 'lossbased'`

Data Types: `char`

### Options — Estimation options

`[]` (default) | structure array returned by `statset`

Estimation options, specified as the comma-separated pair consisting of 'Options' and a structure array returned by `statset`.

To invoke parallel computing:

- You need a Parallel Computing Toolbox license.
- Specify 'Options', `statset('UseParallel',1)`.

### **verbose** — Verbosity level

0 (default) | 1

Verbosity level, specified as the comma-separated pair consisting of 'Verbose' and 0 or 1. `Verbose` controls the amount of diagnostic messages that the software displays in the Command Window.

If `Verbose` is 0, then the software does not display diagnostic messages. Otherwise, the software displays diagnostic messages.

Example: `'Verbose',1`

Data Types: `single` | `double`

## Output Arguments

### **m** — Cross-validated classification margins

numeric vector | numeric matrix

Cross-validated classification margins on page 32-3303, returned as a numeric vector or matrix.

$m$  is  $n$ -by- $L$ , where  $n$  is the number of observations in  $X$  and  $L$  is the number of regularization strengths in `Mdl` (that is, `numel(Mdl.Lambda)`).

$m(i,j)$  is the cross-validated classification margin of observation  $i$  using the ECOC model, composed of linear classification models, that has regularization strength `Mdl.Lambda(j)`.

## Examples

## Estimate $k$ -Fold Cross-Validation Margins

Load the NLP data set.

```
load nlpdata
```

$X$  is a sparse matrix of predictor data, and  $Y$  is a categorical vector of class labels.

For simplicity, use the label 'others' for all observations in  $Y$  that are not 'simulink', 'dsp', or 'comm'.

```
Y(~(ismember(Y,{'simulink','dsp','comm'}))) = 'others';
```

Cross-validate a multiclass, linear classification model.

```
rng(1); % For reproducibility
CVMdl = fitcecoc(X,Y,'Learner','linear','CrossVal','on');
```

CVMdl is a ClassificationPartitionedLinearECOC model. By default, the software implements 10-fold cross validation. You can alter the number of folds using the 'KFold' name-value pair argument.

Estimate the  $k$ -fold margins.

```
m = kfoldMargin(CVMdl);
size(m)
```

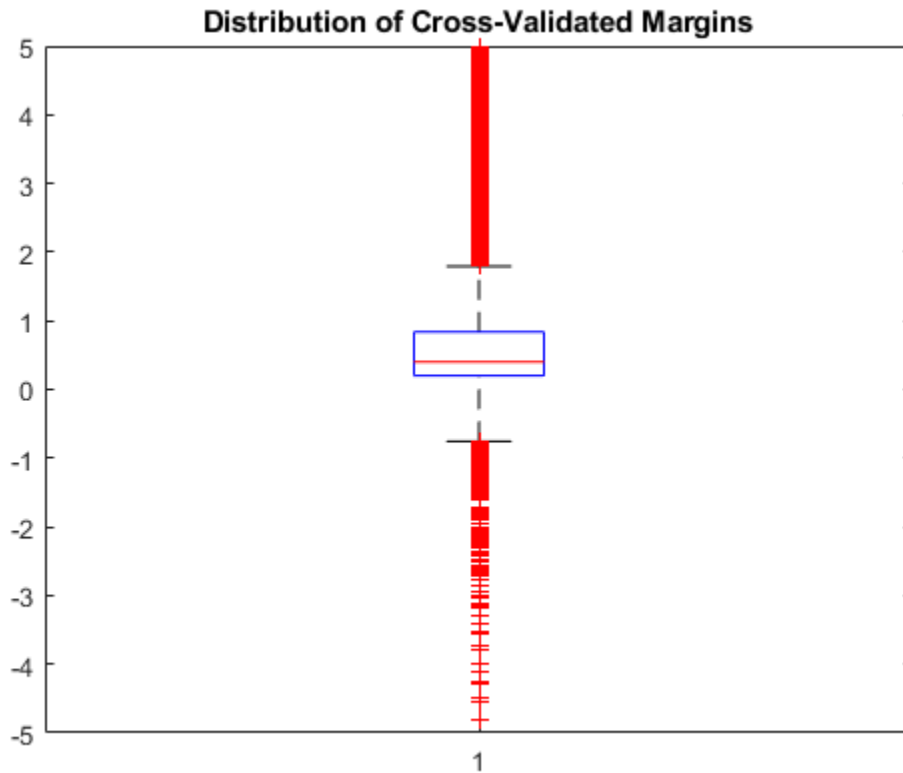
```
ans =
```

```
31572    1
```

$m$  is a 31572-by-1 vector.  $m(j)$  is the average of the out-of-fold margins for observation  $j$ .

Plot the  $k$ -fold margins using box plots.

```
figure;
boxplot(m);
h = gca;
h.YLim = [-5 5];
title('Distribution of Cross-Validated Margins')
```



### Feature Selection Using $k$ -fold Margins

One way to perform feature selection is to compare  $k$ -fold margins from multiple models. Based solely on this criterion, the classifier with the larger margins is the better classifier.

Load the NLP data set. Preprocess the data as in “Estimate  $k$ -Fold Cross-Validation Margins” on page 32-3294, and orient the predictor data so that observations correspond to columns.

```
load nlpdata
Y(~(ismember(Y,{'simulink','dsp','comm'}))) = 'others';
X = X';
```

Create these two data sets:

- fullX contains all predictors.
- partX contains 1/2 of the predictors chosen at random.

```
rng(1); % For reproducibility
p = size(X,1); % Number of predictors
halfPredIdx = randsample(p,ceil(0.5*p));
fullX = X;
partX = X(halfPredIdx,:);
```

Create a linear classification model template that specifies optimizing the objective function using SpARSA.

```
t = templateLinear('Solver','sparsa');
```

Cross-validate two ECOC models composed of binary, linear classification models: one that uses the all of the predictors and one that uses half of the predictors. Indicate that observations correspond to columns.

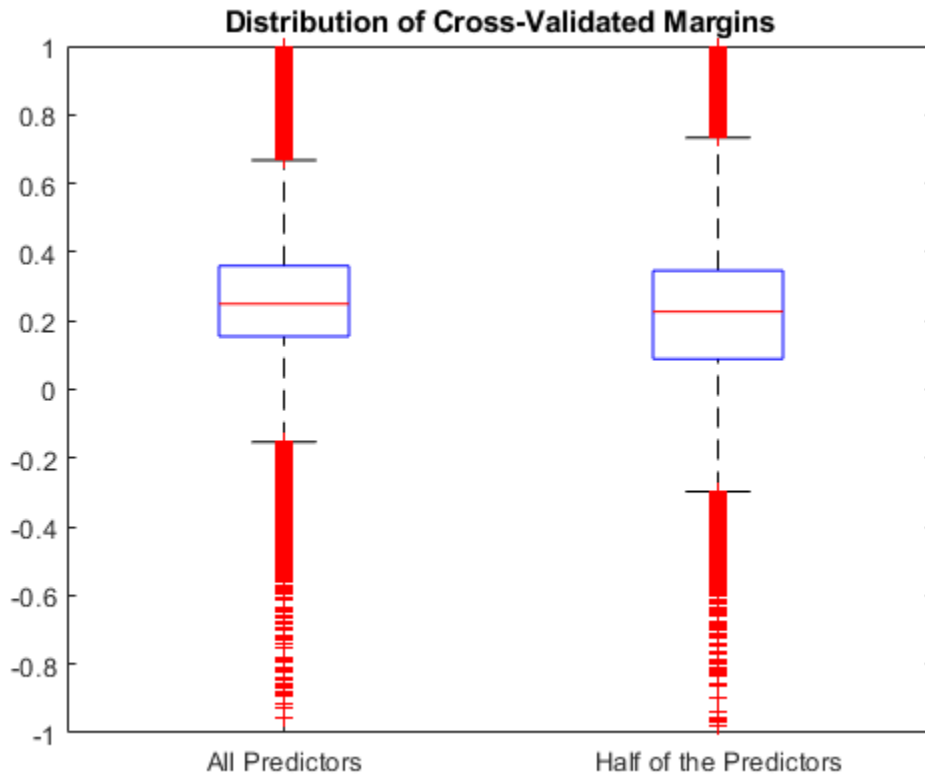
```
CVMD1 = fitcecoc(fullX,Y,'Learners',t,'CrossVal','on',...
    'ObservationsIn','columns');
PCVMD1 = fitcecoc(partX,Y,'Learners',t,'CrossVal','on',...
    'ObservationsIn','columns');
```

CVMD1 and PCVMD1 are ClassificationPartitionedLinearECOC models.

Estimate the  $k$ -fold margins for each classifier. Plot the distribution of the  $k$ -fold margins sets using box plots.

```
fullMargins = kfoldMargin(CVMD1);
partMargins = kfoldMargin(PCVMD1);

figure;
boxplot([fullMargins partMargins],'Labels',...
    {'All Predictors','Half of the Predictors'});
h = gca;
h.YLim = [-1 1];
title('Distribution of Cross-Validated Margins')
```



The distributions of the  $k$ -fold margins of the two classifiers are similar.

### Find Good Lasso Penalty Using $k$ -fold Margins

To determine a good lasso-penalty strength for a linear classification model that uses a logistic regression learner, compare distributions of  $k$ -fold margins.

Load the NLP data set. Preprocess the data as in “Feature Selection Using  $k$ -fold Margins” on page 32-3296.



```
load nlpdata
Y(~(ismember(Y,{'simulink','dsp','comm'}))) = 'others';
X = X';
```

Create a set of 11 logarithmically-spaced regularization strengths from  $10^{-8}$  through  $10^1$ .

```
Lambda = logspace(-8,1,11);
```

Create a linear classification model template that specifies using logistic regression with a lasso penalty, using each of the regularization strengths, optimizing the objective function using SpARSA, and reducing the tolerance on the gradient of the objective function to  $1e-8$ .

```
t = templateLinear('Learner','logistic','Solver','sparsa',...
    'Regularization','lasso','Lambda',Lambda,'GradientTolerance',1e-8);
```

Cross-validate an ECOC model composed of binary, linear classification models using 5-fold cross-validation and that

```
rng(10); % For reproducibility
CVMdl = fitcecoc(X,Y,'Learners',t,'ObservationsIn','columns','KFold',5)
```

```
CVMdl =
```

```
classreg.learning.partition.ClassificationPartitionedLinearECOC
  CrossValidatedModel: 'Linear'
    ResponseName: 'Y'
  NumObservations: 31572
        KFold: 5
    Partition: [1x1 cvpartition]
  ClassNames: [comm dsp simulink others]
  ScoreTransform: 'none'
```

CVMdl is a ClassificationPartitionedLinearECOC model.

Estimate the  $k$ -fold margins for each regularization strength. The scores for logistic regression are in  $[0,1]$ . Apply the quadratic binary loss.

```
m = kfoldMargin(CVMdl,'BinaryLoss','quadratic');
size(m)
```

```
ans =
```

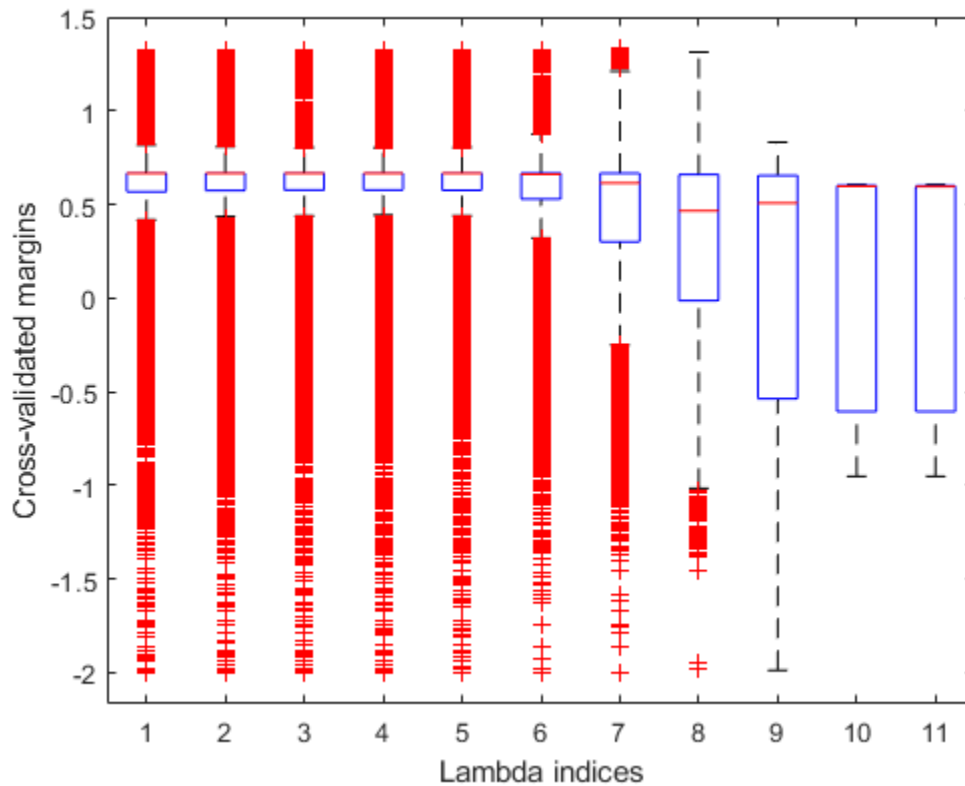
31572

11

$m$  is a 31572-by-11 matrix of cross-validated margins for each observation. The columns correspond to the regularization strengths.

Plot the  $k$ -fold margins for each regularization strength.

```
figure;  
boxplot(m)  
ylabel('Cross-validated margins')  
xlabel('Lambda indices')
```



Several values of `Lambda` yield similarly high margin distribution centers with low spreads. Higher values of `Lambda` lead to predictor variable sparsity, which is a good quality of a classifier.

Choose the regularization strength that occurs just before the margin distribution center starts decreasing and spread starts increasing.

```
LambdaFinal = Lambda(5);
```

Train an ECOC model composed of linear classification model using the entire data set and specify the regularization strength `LambdaFinal`.

```
t = templateLinear('Learner','logistic','Solver','sparsa',...
    'Regularization','lasso','Lambda',Lambda(5),'GradientTolerance',1e-8);
MdlFinal = fitcecoc(X,Y,'Learners',t,'ObservationsIn','columns');
```

To estimate labels for new observations, pass `MdlFinal` and the new data to `predict`.

- “Quick Start Parallel Computing for Statistics and Machine Learning Toolbox” on page 30-2

## Definitions

### Binary Loss

A binary loss is a function of the class and classification score that determines how well a binary learner classifies an observation into the class.

Let:

- $m_{kj}$  be element  $(k,j)$  of the coding design matrix  $M$  (i.e., the code corresponding to class  $k$  of binary learner  $j$ )
- $s_j$  be the score of binary learner  $j$  for an observation
- $g$  be the binary loss function
- $\hat{k}$  be the predicted class for the observation

In loss-based decoding [15], the class producing the minimum sum of the binary losses over binary learners determines the predicted class of an observation, that is,

$$\hat{k} = \operatorname{argmin}_k \sum_{j=1}^L |m_{kj}| g(m_{kj}, s_j).$$

In loss-weighted decoding [15], the class producing the minimum average of the binary losses over binary learners determines the predicted class of an observation, that is,

$$\hat{k} = \operatorname{argmin}_k \frac{\sum_{j=1}^L |m_{kj}| g(m_{kj}, s_j)}{\sum_{j=1}^L |m_{kj}|}.$$

Allwein et al. [2] suggest that loss-weighted decoding improves classification accuracy by keeping loss values for all classes in the same dynamic range.

This table summarizes the supported loss functions, where  $y_j$  is a class label for a particular binary learner (in the set  $\{-1, 1, 0\}$ ),  $s_j$  is the score for observation  $j$ , and  $g(y_j, s_j)$ .

Value	Description	Score Domain	$g(y_j, s_j)$
'binodeviance'	Binomial deviance	$(-\infty, \infty)$	$\log[1 + \exp(-2y_j s_j)] / [2\log(2)]$
'exponential'	Exponential	$(-\infty, \infty)$	$\exp(-y_j s_j) / 2$
'hamming'	Hamming	$[0, 1]$ or $(-\infty, \infty)$	$[1 - \operatorname{sign}(y_j s_j)] / 2$
'hinge'	Hinge	$(-\infty, \infty)$	$\max(0, 1 - y_j s_j) / 2$
'linear'	Linear	$(-\infty, \infty)$	$(1 - y_j s_j) / 2$
'logit'	Logistic	$(-\infty, \infty)$	$\log[1 + \exp(-y_j s_j)] / [2\log(2)]$
'quadratic'	Quadratic	$[0, 1]$	$[1 - y_j(2s_j - 1)]^2 / 2$

The software normalizes the binary losses such that the loss is 0.5 when  $y_j = 0$ , and aggregates using the average of the binary learners [2].

Do not confuse the binary loss with the overall classification loss (specified by the `LossFun` name-value pair argument of `predict` and `loss`), e.g., classification error, which measures how well an ECOC classifier performs as a whole.

## Classification Margin

The classification margins are, for each observation, the difference between the negative loss for the true class and maximal negative loss among the false classes. If the margins are on the same scale, then they serve as a classification confidence measure, i.e., among multiple classifiers, those that yield larger margins are better.

## References

- [1] Allwein, E., R. Schapire, and Y. Singer. “Reducing multiclass to binary: A unifying approach for margin classifiers.” *Journal of Machine Learning Research*. Vol. 1, 2000, pp. 113–141.
- [2] Escalera, S., O. Pujol, and P. Radeva. “On the decoding process in ternary error-correcting output codes.” *IEEE Transactions on Pattern Analysis and Machine Intelligence*. Vol. 32, Issue 7, 2010, pp. 120–134.
- [3] Escalera, S., O. Pujol, and P. Radeva. “Separability of ternary codes for sparse designs of error-correcting output codes.” *Pattern Recogn.* Vol. 30, Issue 3, 2009, pp. 285–297.

## See Also

`ClassificationLinear` | `ClassificationPartitionedLinearECOC` | `kfoldEdge`  
| `kfoldPredict` | `margin`

## Topics

“Quick Start Parallel Computing for Statistics and Machine Learning Toolbox” on page 30-2

“Reproducibility in Parallel Statistical Computations” on page 30-14

“Concepts of Parallel Computing in Statistics and Machine Learning Toolbox” on page 30-8

**Introduced in R2016a**

## kfoldMargin

**Class:** ClassificationPartitionedModel

Classification margins for observations not used for training

### Syntax

```
M = kfoldMargin(obj)
```

### Description

`M = kfoldMargin(obj)` returns classification margins obtained by cross-validated classification model `obj`. For every fold, this method computes classification margins for in-fold observations using a model trained on out-of-fold observations.

### Input Arguments

**obj**

A partitioned classification model of type `ClassificationPartitionedModel` or `ClassificationPartitionedEnsemble`.

### Output Arguments

**M**

The classification margin.

### Examples

## Estimate the $k$ -fold Margins of a Classifier

Find the  $k$ -fold margins for an ensemble that classifies the ionosphere data.

Load the ionosphere data set.

```
load ionosphere
```

Train a classification ensemble of decision trees.

```
Mdl = fitensemble(X,Y,'AdaBoostM1',100,'Tree');
```

Cross validate the classifier using 10-fold cross validation.

```
cvens = crossval(Mdl);
```

Compute the `_k_fold` margins. Display summary statistics for the margins.

```
m = kfoldMargin(cvens);
marginStats = table(min(m),mean(m),max(m),...
    'VariableNames',{'Min','Mean','Max'})
```

```
marginStats =
```

```
1x3 table
```

Min	Mean	Max
-11.312	7.3236	23.517

## Definitions

### Margin

The classification margin is the difference between the classification score for the true class and maximal classification score for the false classes.

The classification margin is a column vector with the same number of rows as in the matrix  $X$ . A high value of margin indicates a more reliable prediction than a low value.

## Score

For discriminant analysis, the score of a classification is the posterior probability of the classification. For the definition of posterior probability in discriminant analysis, see “Posterior Probability” on page 20-8.

For ensembles, a classification score represents the confidence of a classification into a class. The higher the score, the higher the confidence.

Different ensemble algorithms have different definitions for their scores. Furthermore, the range of scores depends on ensemble type. For example:

- AdaBoostM1 scores range from  $-\infty$  to  $\infty$ .
- Bag scores range from 0 to 1.

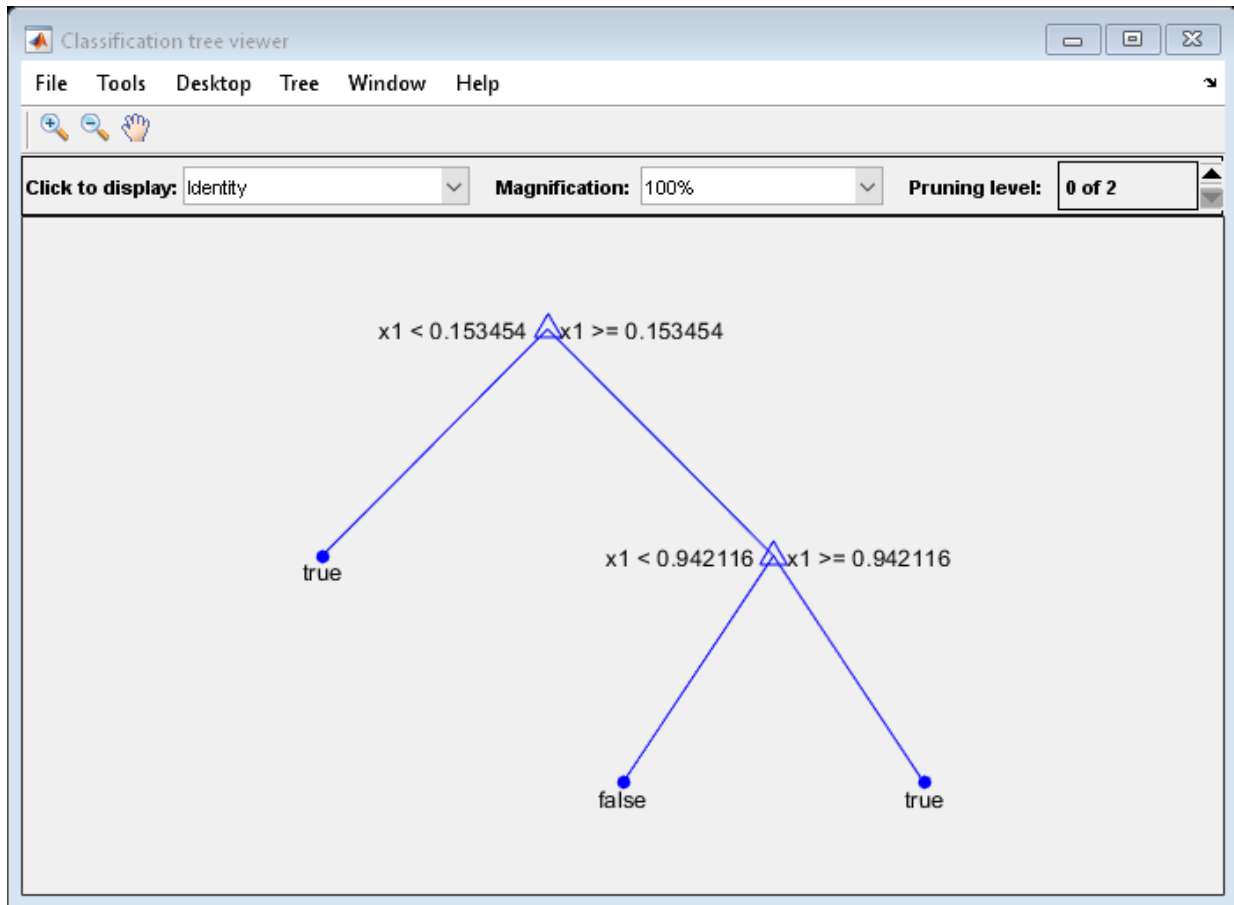
For trees, the score of a classification of a leaf node is the posterior probability of the classification at that node. The posterior probability of the classification at a node is the number of training sequences that lead to that node with the classification, divided by the number of training sequences that lead to that node.

For example, consider classifying a predictor `X` as `true` when `X < 0.15` or `X > 0.95`, and `X` is `false` otherwise.

Generate 100 random points and classify them:

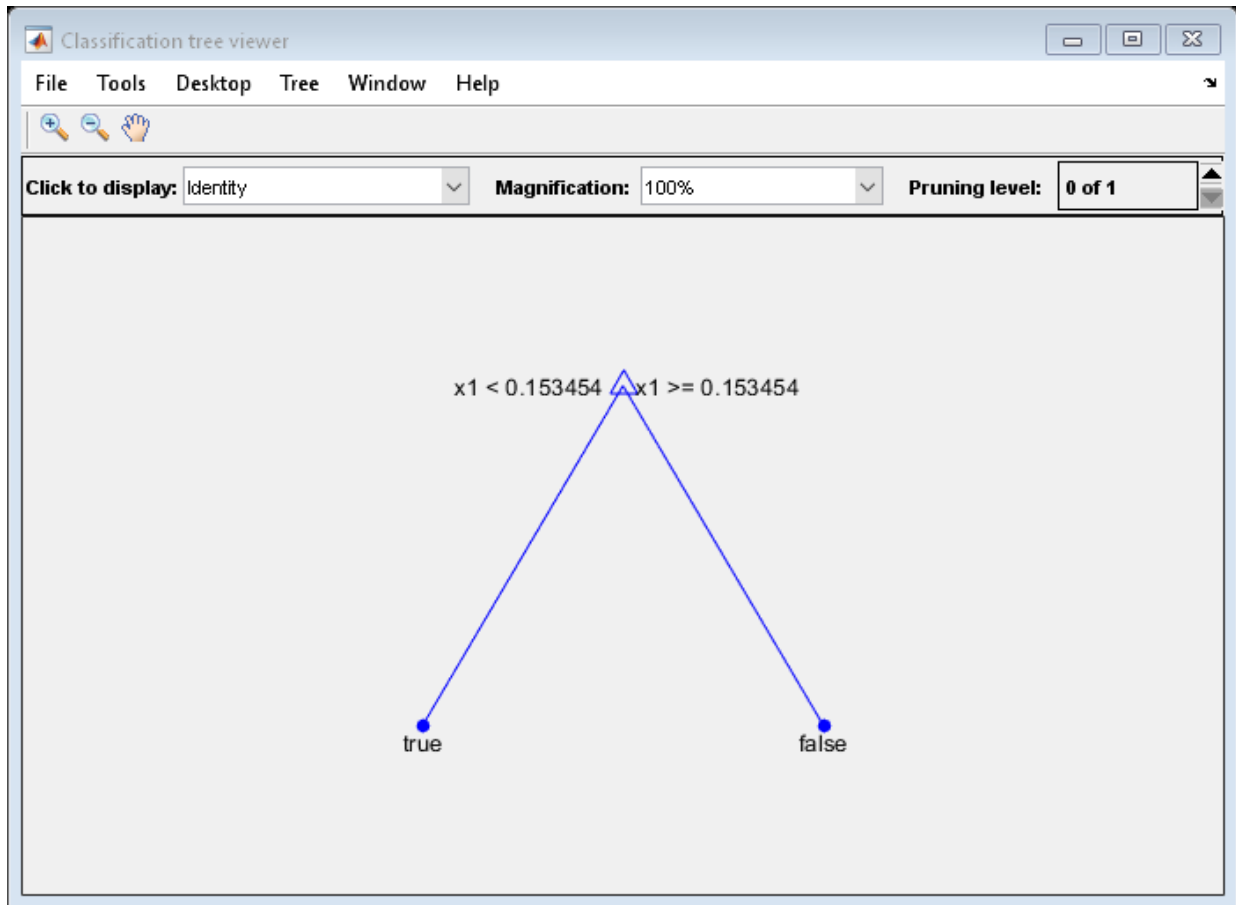
```
rng(0, 'twister') % for reproducibility
X = rand(100,1);
Y = (abs(X - .55) > .4);
tree = fitctree(X,Y);
view(tree, 'Mode', 'Graph')
```





Prune the tree:

```
tree1 = prune(tree, 'Level', 1);  
view(tree1, 'Mode', 'Graph')
```



The pruned tree correctly classifies observations that are less than 0.15 as `true`. It also correctly classifies observations from .15 to .94 as `false`. However, it incorrectly classifies observations that are greater than .94 as `false`. Therefore, the score for observations that are greater than .15 should be about  $.05/.85=.06$  for `true`, and about  $.8/.85=.94$  for `false`.

Compute the prediction scores for the first 10 rows of `X`:

```
[~,score] = predict(tree1,X(1:10));
[score X(1:10,:)]
```

```
ans =  
  
    0.9059    0.0941    0.8147  
    0.9059    0.0941    0.9058  
         0     1.0000    0.1270  
    0.9059    0.0941    0.9134  
    0.9059    0.0941    0.6324  
         0     1.0000    0.0975  
    0.9059    0.0941    0.2785  
    0.9059    0.0941    0.5469  
    0.9059    0.0941    0.9575  
    0.9059    0.0941    0.9649
```

Indeed, every value of  $x$  (the right-most column) that is less than 0.15 has associated scores (the left and center columns) of 0 and 1, while the other values of  $x$  have associated scores of 0.91 and 0.09. The difference (score 0.09 instead of the expected 0.06) is due to a statistical fluctuation: there are 8 observations in  $x$  in the range  $(.95, 1)$  instead of the expected 5 observations.

## See Also

[ClassificationPartitionedModel](#) | [crossval](#) | [kfoldEdge](#) | [kfoldLoss](#) | [kfoldPredict](#) | [kfoldfun](#)

## kfoldPredict

**Class:** ClassificationPartitionedECOC

Predict responses for observations not used for training

### Syntax

```
label = kfoldPredict(CVMdl)
label = kfoldPredict(CVMdl, Name, Value)
[label, NegLoss, PBScore] = kfoldPredict(____)
[label, NegLoss, PBScore, Posterior] = kfoldPredict(____)
```

### Description

`label = kfoldPredict(CVMdl)` returns class labels predicted by the cross-validated ECOC model (`ClassificationPartitionedECOC`) `CVMdl`. That is, for every fold, `kfoldPredict` predicts class labels for observations that it holds out when it trains. `CVMdl.X` contains both sets of observations.

The software predicts the classification of an observation by assigning the observation to the class yielding the largest negated average binary loss (or, equivalently, the smallest average binary loss).

`label = kfoldPredict(CVMdl, Name, Value)` returns predicted class labels with additional options specified by one or more `Name, Value` pair arguments.

For example, specify the posterior probability estimation method, decoding scheme, or verbosity level.

`[label, NegLoss, PBScore] = kfoldPredict(____)` additionally returns negated values of the average binary loss per class (`NegLoss`) for in-fold observations, and positive-class scores (`PBScore`) for in-fold observations classified by each binary learner.

If the coding matrix varies across folds (that is, if the coding scheme is `sparserandom` or `denserandom`), then `PBScore` is empty (`[]`).

[label, NegLoss, PBScore, Posterior] = kfoldPredict(\_\_\_\_) additionally returns posterior class probability estimates for in-fold observations (Posterior).

To obtain posterior class probabilities, you must set 'FitPosterior', 1 when training the ECOC model using fitcecoc. Otherwise, kfoldPredict throws an error.

## Input Arguments

### **cvmdl** — Cross-validated ECOC model

ClassificationPartitionedECOC model

Cross-validated ECOC model, specified as a ClassificationPartitionedECOC model. You can create a ClassificationPartitionedECOC model by:

- Passing a trained ECOC model (ClassificationECOC) to crossval
- Training an ECOC model using fitcecoc and setting any one of these cross-validation name-value pair arguments: 'CrossVal', 'KFold', 'Holdout', 'Leaveout', or 'CVPartition'

## Name-Value Pair Arguments

Specify optional comma-separated pairs of Name, Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as Name1, Value1, ..., NameN, ValueN.

### **BinaryLoss** — Binary learner loss function

'hamming' | 'linear' | 'logit' | 'exponential' | 'binodeviance' | 'hinge'  
| 'quadratic' | function handle

Binary learner loss function, specified as the comma-separated pair consisting of 'BinaryLoss' and a built-in, loss-function name or function handle.

- This table contains names and descriptions of the built-in functions, where  $y_j$  is a class label for a particular binary learner (in the set  $\{-1, 1, 0\}$ ),  $s_j$  is the score for observation  $j$ , and  $g(y_j, s_j)$  is the binary loss formula.

Value	Description	Score Domain	$g(y, s_j)$
'binodeviance'	Binomial deviance	$(-\infty, \infty)$	$\log[1 + \exp(-2y_j s_j)] / [2\log(2)]$
'exponential'	Exponential	$(-\infty, \infty)$	$\exp(-y_j s_j) / 2$
'hamming'	Hamming	$[0, 1]$ or $(-\infty, \infty)$	$[1 - \text{sign}(y_j s_j)] / 2$
'hinge'	Hinge	$(-\infty, \infty)$	$\max(0, 1 - y_j s_j) / 2$
'linear'	Linear	$(-\infty, \infty)$	$(1 - y_j s_j) / 2$
'logit'	Logistic	$(-\infty, \infty)$	$\log[1 + \exp(-y_j s_j)] / [2\log(2)]$
'quadratic'	Quadratic	$[0, 1]$	$[1 - y_j (2s_j - 1)]^2 / 2$

The software normalizes the binary losses such that the loss is 0.5 when  $y_j = 0$ . Also, the software calculates the mean binary loss for each class.

- For a custom binary loss function, e.g., `customFunction`, specify its function handle `'BinaryLoss', @customFunction`.

`customFunction` should have this form:

```
bLoss = customFunction(M, s)
```

where:

- $M$  is the  $K$ -by- $L$  coding matrix stored in `Mdl.CodingMatrix`.
- $s$  is the 1-by- $L$  row vector of classification scores.
- `bLoss` is the classification loss. This scalar aggregates the binary losses for every learner in a particular class. For example, you can use the mean binary loss to aggregate the loss over the learners for each class.
- $K$  is the number of classes.
- $L$  is the number of binary learners.

For an example of passing a custom binary loss function, see “Predict Test-Sample Labels of ECOC Models Using Custom Binary Loss Function” on page 32-4933.

By default, if all binary learners are:

- SVMs or linear classification models of SVM learners, then `BinaryLoss` is `'hinge'`
- Ensembles trained by `AdaboostM1` or `GentleBoost`, then `BinaryLoss` is `'exponential'`

- Ensembles trained by `LogitBoost`, then `BinaryLoss` is `'binodeviance'`
- Linear classification models of logistic regression learners, or you specify to predict class posterior probabilities (i.e., set `'FitPosterior', 1` in `fitcecoc`), then `BinaryLoss` is `'quadratic'`

Otherwise, the default `BinaryLoss` is `'hamming'`. To check the default value, use dot notation to display the `BinaryLoss` property of the trained model at the command line.

Example: `'BinaryLoss', 'binodeviance'`

Data Types: `char` | `function_handle`

### **Decoding — Decoding scheme**

`'lossweighted'` (default) | `'lossbased'`

Decoding scheme that aggregates the binary losses on page 32-4941, specified as the comma-separated pair consisting of `'Decoding'` and `'lossweighted'` or `'lossbased'`.

Example: `'Decoding', 'lossbased'`

Data Types: `char`

### **NumKLInitializations — Number of random initial values**

0 (default) | nonnegative integer

Number of random initial values for fitting posterior probabilities by Kullback-Leibler divergence minimization, specified as the comma-separated pair consisting of `'NumKLInitializations'` and a nonnegative integer.

If you do not request the fourth output argument (`Posterior`) and set `'PosteriorMethod', 'kl'` (the default), then the software ignores the value of `NumKLInitializations`.

For more details, see “Posterior Estimation Using Kullback-Leibler Divergence” on page 32-4943.

Example: `'NumKLInitializations', 5`

Data Types: `single` | `double`

### **Options — Estimation options**

`[]` (default) | structure array returned by `statset`

Estimation options, specified as the comma-separated pair consisting of 'Options' and a structure array returned by `statset`.

To invoke parallel computing:

- You need a Parallel Computing Toolbox license.
- Specify 'Options', `statset('UseParallel',1)`.

### **PosteriorMethod — Posterior probability estimation method**

'kl' (default) | 'qp'

Posterior probability estimation method, specified as the comma-separated pair consisting of 'PosteriorMethod' and 'kl' or 'qp'.

- If `PosteriorMethod` is 'kl', then the software estimates multiclass posterior probabilities by minimizing the Kullback-Leibler divergence between the predicted and expected posterior probabilities returned by binary learners. For details, see “Posterior Estimation Using Kullback-Leibler Divergence” on page 32-4943.
- If `PosteriorMethod` is 'qp', then the software estimates multiclass posterior probabilities by solving a least-squares problem using quadratic programming. You need an Optimization Toolbox license to use this option. For details, see “Posterior Estimation Using Quadratic Programming” on page 32-4945.
- If you do not request the fourth output argument (`Posterior`), then the software ignores the value of `PosteriorMethod`.

Example: 'PosteriorMethod', 'qp'

Data Types: char

### **Verbose — Verbosity level**

0 (default) | 1

Verbosity level, specified as the comma-separated pair consisting of 'Verbose' and 0 or 1. `Verbose` controls the amount of diagnostic messages that the software displays in the Command Window.

If `Verbose` is 0, then the software does not display diagnostic messages. Otherwise, the software displays diagnostic messages.

Example: 'Verbose', 1

Data Types: single | double



## Output Arguments

### **label** — Predicted class labels

categorical array | character array | logical vector | vector of numeric values | cell array of character vectors

Predicted class labels, returned as a categorical or character array, logical or numeric vector, or cell array of character vectors.

label:

- Is the same data type as `CVMdl.Y`
- Has length equal to the number of rows of `CVMdl.X`

The software predicts the classification of an observation by assigning the observation to the class yielding the largest negated average binary loss (or, equivalently, the smallest average binary loss).

### **NegLoss** — Negated average binary losses

numeric matrix

Negated average binary losses, returned as a numeric matrix. `NegLoss` is an  $n$ -by- $K$  matrix, where  $n$  is the number of observations (`size(CVMdl.X,1)`) and  $K$  is the number of unique classes (`size(CVMdl.ClassNames,1)`).

### **PBScore** — Positive-class scores

numeric matrix

Positive-class scores for each binary learner, returned as a numeric matrix. `PBScore` is an  $n$ -by- $L$  matrix, where  $n$  is the number of observations (`size(CVMdl.X,1)`) and  $L$  is the number of binary learners (`size(CVMdl.CodingMatrix,2)`).

If the coding matrix varies across folds (that is, if the coding scheme is `sparsrandom` or `denserandom`), then `PBScore` is empty (`[]`).

### **Posterior** — Posterior class probabilities

numeric matrix

Posterior class probabilities, returned as a numeric matrix. `Posterior` is an  $n$ -by- $K$  matrix, where  $n$  is the number of observations (`size(CVMdl.X,1)`) and  $K$  is the number of unique classes (`size(CVMdl.ClassNames,1)`).

You must set `'FitPosterior', 1` when training the ECOC model using `fitcecoc` to request `Posterior`. Otherwise, the software throws an error.

## Examples

### Predict $k$ -Fold Cross-Validation Labels of ECOC Models

Load Fisher's iris data set.

```
load fisheriris
X = meas;
Y = categorical(species);
classOrder = unique(Y);
rng(1); % For reproducibility
```

Train an ECOC model using SVM binary classifiers and specify to cross validate. It is good practice to standardize the predictors and define the class order. Specify to standardize the predictors using an SVM template.

```
t = templateSVM('Standardize', 1);
CVMdl = fitcecoc(X, Y, 'CrossVal', 'on', 'Learners', t, 'ClassNames', classOrder);
```

`CVMdl` is a `ClassificationPartitionedModel` model. By default, the software implements 10-fold cross validation. You can alter the number of folds using the `'KFold'` name-value pair argument.

Predict the out-of-fold labels. Print a random subset of true and predicted labels.

```
labels = kfoldPredict(CVMdl);
idx = randsample(numel(labels), 10);
table(Y(idx), labels(idx), ...
      'VariableNames', {'TrueLabels', 'PredictedLabels'})
```

```
ans =
```

```
10x2 table
```

```
TrueLabels    PredictedLabels
```

```
_____
```

```

setosa      setosa
versicolor versicolor
setosa      setosa
virginica   virginica
versicolor versicolor
setosa      setosa
virginica   virginica
virginica   virginica
setosa      setosa
setosa      setosa

```

CVMdl correctly labeled the out-of-fold observations with indices `idx`.

## Predict Cross-Validation Labels of ECOC Models Using Custom Binary Loss Function

Load Fisher's iris data set.

```

load fisheriris
X = meas;
Y = categorical(species);
classOrder = unique(Y); % Class order
K = numel(classOrder); % Number of classes
rng(1); % For reproducibility

```

Train an ECOC model using SVM binary classifiers and specify to cross validate. It is good practice to standardize the predictors and define the class order. Specify to standardize the predictors using an SVM template.

```

t = templateSVM('Standardize',1);
CVMdl = fitcecoc(X,Y,'CrossVal','on','Learners',t,'ClassNames',classOrder);

```

CVMdl is a ClassificationPartitionedModel model. By default, the software implements 10-fold cross validation. You can alter the number of folds using the 'KFold' name-value pair argument.

SVM scores are signed distances from the observation to the decision boundary. Therefore, the domain is  $(-\infty, \infty)$ . Create a custom binary loss function that:

- Maps the coding design matrix ( $M$ ) and positive-class classification scores ( $s$ ) for each learner to the binary loss for each observation

- Uses linear loss
- Aggregates the binary learner loss using the median.

You can create a separate function for the binary loss function, and then save it on the MATLAB® path. Or, you can specify an anonymous binary loss function.

```
customBL = @(M,s)nanmedian(1 - bsxfun(@times,M,s),2)/2;
```

Predict cross-validation labels and estimate the median binary loss per class. Print the median negative binary losses per class for a random set of 10 out-of-fold observations.

```
[label,NegLoss] = kfoldPredict(CVMdl,'BinaryLoss',customBL);
```

```
idx = randsample(numel(label),10);
classOrder
table(Y(idx),label(idx),NegLoss(idx,:), 'VariableNames',...
      {'TrueLabel','PredictedLabel','NegLoss'})
```

```
classOrder =
```

```
3x1 categorical array
```

```
setosa
versicolor
virginica
```

```
ans =
```

```
10x3 table
```

TrueLabel	PredictedLabel	NegLoss		
setosa	versicolor	0.37141	2.1292	-4.0006
versicolor	versicolor	-1.2169	0.36685	-0.64995
setosa	versicolor	0.23914	2.0795	-3.8187
virginica	virginica	-1.915	-0.19952	0.61451
versicolor	versicolor	-1.3744	0.45537	-0.58093
setosa	versicolor	0.20073	2.2777	-3.9784
virginica	versicolor	-1.4926	0.09073	-0.098159
virginica	virginica	-1.7667	-0.13462	0.40134
setosa	versicolor	0.20006	1.9112	-3.6112

```
setosa      versicolor    0.16108      1.9684      -3.6294
```

The order of the columns corresponds to the elements of `classOrder`. The software predicts the label based on the maximum negated loss. The results seem to indicate that the median of the linear losses might not perform as well as other losses.

### Estimate Cross-Validation Posterior Probabilities of ECOC Models

Load Fisher's iris data set. Train the classifier using the petal dimensions as predictors.

```
load fisheriris
X = meas(:,3:4);
Y = categorical(species);
classOrder = unique(Y);
rng(1); % For reproducibility
```

Create an SVM template, and specify the Gaussian kernel. It is good practice to standardize the predictors.

```
t = templateSVM('Standardize',1,'KernelFunction','gaussian');
```

`t` is an SVM template. Most of its properties are empty. When the software trains the ECOC classifier, it sets the applicable properties to their default values.

Train the ECOC classifier using the SVM template, and specify to cross validate. Transform classification scores to class posterior probabilities (returned by `kfoldPredict`) using the `'FitPosterior'` name-value pair argument. It is good practice to specify the class order.

```
CVMD1 = fitcecoc(X,Y,'Learners',t,'CrossVal','on','FitPosterior',true,...
    'ClassNames',classOrder);
```

`CVMD1` is a `ClassificationPartitionedECOC` model. By default, the software uses 10-fold cross validation.

Predict the out-of-fold class posterior probabilities. Specify to use 10 random initial values for the Kullback-Leibler algorithm.

```
[label,~,~,Posterior] = kfoldPredict(CVMD1,'NumKLInitializations',10);
```

The software assigns an observation to the class that yields the smallest average binary loss. Since all binary learners are computing posterior probabilities, the binary loss function is quadratic.

Display a random set of results.

```
idx = randsample(size(X,1),10);
CVMdl.ClassNames
table(Y(idx),label(idx),Posterior(idx,:),...
      'VariableNames',{'TrueLabel','PredLabel','Posterior'})
```

```
ans =
```

```
3x1 categorical array
```

```
setosa
versicolor
virginica
```

```
ans =
```

```
10x3 table
```

TrueLabel	PredLabel	Posterior		
versicolor	versicolor	0.0086394	0.98243	0.0089291
versicolor	virginica	2.2197e-14	0.12447	0.87553
setosa	setosa	0.999	0.00022837	0.00076884
versicolor	versicolor	2.2194e-14	0.98916	0.010839
virginica	virginica	0.012318	0.012925	0.97476
virginica	virginica	0.0015571	0.0015638	0.99688
virginica	virginica	0.0042895	0.0043556	0.99135
setosa	setosa	0.999	0.00028329	0.00071382
virginica	virginica	0.0094721	0.0098231	0.9807
setosa	setosa	0.999	0.00013562	0.00086192

The columns of Posterior correspond to the class order of `Mdl.ClassNames`.

## Estimate Cross-Validation Posterior Probabilities Using Parallel Computing

Train an error-correcting output codes, multiclass model and estimate posterior probabilities using parallel computing.

Load the arrhythmia data set.

```
load arrhythmia
Y = categorical(Y);
tabulate(Y)
n = numel(Y);
K = numel(unique(Y));
```

Value	Count	Percent
1	245	54.20%
2	44	9.73%
3	15	3.32%
4	15	3.32%
5	13	2.88%
6	25	5.53%
7	3	0.66%
8	2	0.44%
9	9	1.99%
10	50	11.06%
14	4	0.88%
15	5	1.11%
16	22	4.87%

Several classes are not represented in the data, and many of the other classes have low relative frequencies.

Specify an ensemble learning template that uses the GentleBoost method, and 50 weak, classification tree learners.

```
t = templateEnsemble('GentleBoost',50,'Tree');
```

`t` is a template object. Most of the options are empty (`[]`). The software uses default values for all empty options during training.

Since there are many classes, specify a sparse random coding design.

```
rng(1); % For reproducibility
Coding = designecoc(K,'sparserandom');
```

Train an ECOC model using parallel computing, and specify to cross validate and fit posterior probabilities (returned by `kfoldPredict`).

```
pool = parpool; % Invokes workers
options = statset('UseParallel',1);
CVMdl = fitcecoc(X,Y,'Learner',t,'Options',options,'Coding',Coding,...
    'FitPosterior',1,'CrossVal','on');
```

Starting parallel pool (parpool) using the 'local' profile ... connected to 4 workers.  
Warning: One or more folds do not contain points from all the groups.

`CVMdl` is a `ClassificationPartitionedECOC` model. By default, the software implements 10-fold cross validation. You can alter the number of folds using the `'KFold'` name-value pair argument.

The pool invokes four workers. The number of workers might vary among systems. Also, there is a good chance that one or more folds do not contain observations from all classes since some classes have low relative frequency.

Estimate posterior probabilities, and display the posterior probability of being classified as not having arrhythmia (class 1) given the data for a random set of out-of-fold observations.

```
[~,~,~,posterior] = kfoldPredict(CVMdl,'Options',options);
idx = randsample(n,10);
table(idx,Y(idx),posterior(idx,1),...
    'VariableNames',{'OOFSampleIndex','TrueLabel','PosteriorNoArrhythmia'})
```

ans =

OOFSampleIndex	TrueLabel	PosteriorNoArrhythmia
171	1	0.33654
221	1	0.85135
72	16	0.9174
3	10	0.025649
202	1	0.8438
243	1	0.94338
18	1	0.81789
49	6	0.090153



234	1	0.61626
315	1	0.97187

- “Quick Start Parallel Computing for Statistics and Machine Learning Toolbox” on page 30-2

## Definitions

### Binary Loss

A binary loss is a function of the class and classification score that determines how well a binary learner classifies an observation into the class.

Let:

- $m_{kj}$  be element  $(k,j)$  of the coding design matrix  $M$  (i.e., the code corresponding to class  $k$  of binary learner  $j$ )
- $s_j$  be the score of binary learner  $j$  for an observation
- $g$  be the binary loss function
- $\hat{k}$  be the predicted class for the observation

In loss-based decoding [15], the class producing the minimum sum of the binary losses over binary learners determines the predicted class of an observation, that is,

$$\hat{k} = \operatorname{argmin}_k \sum_{j=1}^L |m_{kj}| g(m_{kj}, s_j).$$

In loss-weighted decoding [15], the class producing the minimum average of the binary losses over binary learners determines the predicted class of an observation, that is,

$$\hat{k} = \operatorname{argmin}_k \frac{\sum_{j=1}^L |m_{kj}| g(m_{kj}, s_j)}{\sum_{j=1}^L |m_{kj}|}.$$

Allwein et al. [2] suggest that loss-weighted decoding improves classification accuracy by keeping loss values for all classes in the same dynamic range.

This table summarizes the supported loss functions, where  $y_j$  is a class label for a particular binary learner (in the set  $\{-1, 1, 0\}$ ),  $s_j$  is the score for observation  $j$ , and  $g(y_j, s_j)$ .

Value	Description	Score Domain	$g(y_j, s_j)$
'binodeviance'	Binomial deviance	$(-\infty, \infty)$	$\log[1 + \exp(-2y_j s_j)] / [2\log(2)]$
'exponential'	Exponential	$(-\infty, \infty)$	$\exp(-y_j s_j) / 2$
'hamming'	Hamming	$[0, 1]$ or $(-\infty, \infty)$	$[1 - \text{sign}(y_j s_j)] / 2$
'hinge'	Hinge	$(-\infty, \infty)$	$\max(0, 1 - y_j s_j) / 2$
'linear'	Linear	$(-\infty, \infty)$	$(1 - y_j s_j) / 2$
'logit'	Logistic	$(-\infty, \infty)$	$\log[1 + \exp(-y_j s_j)] / [2\log(2)]$
'quadratic'	Quadratic	$[0, 1]$	$[1 - y_j(2s_j - 1)]^2 / 2$

The software normalizes the binary losses such that the loss is 0.5 when  $y_j = 0$ , and aggregates using the average of the binary learners [2].

Do not confuse the binary loss with the overall classification loss (specified by the `LossFun` name-value pair argument of `predict` and `loss`), e.g., classification error, which measures how well an ECOC classifier performs as a whole.

## Algorithms

- The software can estimate class posterior probabilities using quadratic programming or by minimizing the Kullback-Leibler divergence. For the following descriptions of the posterior estimation algorithms, let:
  - $m_{kj}$  be the element  $(k, j)$  of the coding design matrix  $M$ .
  - $I$  be the indicator function.
  - $\hat{p}_k$  be the class posterior probability estimate for class  $k$  of an observation,  $k = 1, \dots, K$ .
  - $r_j$  be the positive-class posterior probability for binary learner  $j$ . That is,  $r_j$  is the probability that binary learner  $j$  classifies an observation into the positive class, given the training data.

## Posterior Estimation Using Kullback-Leibler Divergence

By default, the software minimizes the Kullback-Leibler divergence to estimate class posterior probabilities. The Kullback-Leibler divergence between the expected and observed positive-class posterior probabilities is

$$\Delta(r, r) = \sum_{j=1}^L w_j \left[ r_j \log \frac{r_j}{r_j} + (1 - r_j) \log \frac{1 - r_j}{1 - r_j} \right],$$

where  $w_j = \sum_{S_j} w_i^*$  is the weight for binary learner  $j$  with  $S_j$  the set of observation indices that binary learner  $j$  is trained on and  $w_i^*$  is the weight of observation  $i$ . The software minimizes the divergence iteratively. The first step is to choose initial values

$\hat{p}_k^{(0)}$ ;  $k = 1, \dots, K$  for the class posterior probabilities.

- If you do not specify `NumKLIterations`, then the software uses both sets of deterministic initial values described next, and uses the one that minimizes  $\Delta$ .

- $\hat{p}_k^{(0)} = 1 / K$ ;  $k = 1, \dots, K$ .

- $\hat{p}_k^{(0)}$ ;  $k = 1, \dots, K$  is the solution of the system
 
$$M_{01} \hat{p}^{(0)} = r,$$

where  $M_{01}$  is  $M$  with all  $m_{kj} = -1$  replaced with 0, and  $r$  is a vector of positive-class posterior probabilities returned by the  $L$  binary learners [14]. The software uses `lsqnonneg` to solve the system.

- If you specify `'NumKLIterations', c`, where  $c$  is a natural number, then the

software does the following to choose  $\hat{p}_k^{(0)}$ ;  $k = 1, \dots, K$ , and uses the one that minimizes  $\Delta$ .

- The software chooses both sets of deterministic initial values as described previously.
- The software randomly generates  $c$  vectors of length  $K$  using `rand`, and then normalizes each vector to sum to 1.

At iteration  $t$ , the software:

- 1 Computes

$$\hat{r}_j^{(t)} = \frac{\sum_{k=1}^K \hat{p}_k^{(t)} I(m_{kj} = +1)}{\sum_{k=1}^K \hat{p}_k^{(t)} I(m_{kj} = +1 \cup m_{kj} = -1)}.$$

- 2 Estimates the next class posterior probability using

$$\hat{p}_k^{(t+1)} = \hat{p}_k^{(t)} \frac{\sum_{j=1}^L w_j [r_j I(m_{kj} = +1) + (1 - r_j) I(m_{kj} = -1)]}{\sum_{j=1}^L w_j [\hat{r}_j^{(t)} I(m_{kj} = +1) + (1 - \hat{r}_j^{(t)}) I(m_{kj} = -1)]}.$$

- 3 Normalizes  $\hat{p}_k^{(t+1)}$ ;  $k = 1, \dots, K$  so that they sum to 1.
- 4 Checks for convergence.

For more details, see [23] and [49].

## Posterior Estimation Using Quadratic Programming

Posterior probability estimation using quadratic programming requires an Optimization Toolbox license. To estimate posterior probabilities for an observation using this method, the software:

- 1 Estimates the positive-class posterior probabilities,  $r_j$ , for binary learners  $j = 1, \dots, L$ .

- 2 Using the relationship between  $r_j$  and  $\hat{p}_k$  [45], minimizes

$$\sum_{j=1}^L \left[ -r_j \sum_{k=1}^K \hat{p}_k I(m_{kj} = -1) + (1 - r_j) \sum_{k=1}^K \hat{p}_k I(m_{kj} = +1) \right]^2$$

with respect to  $\hat{p}_k$  and the restrictions

$$0 \leq \hat{p}_k \leq 1$$

$$\sum_k \hat{p}_k = 1.$$

The software performs minimization using `quadprog`.

## References

- [1] Allwein, E., R. Schapire, and Y. Singer. “Reducing multiclass to binary: A unifying approach for margin classifiers.” *Journal of Machine Learning Research*. Vol. 1, 2000, pp. 113–141.
- [2] Dietterich, T., and G. Bakiri. “Solving Multiclass Learning Problems Via Error-Correcting Output Codes.” *Journal of Artificial Intelligence Research*. Vol. 2, 1995, pp. 263–286.
- [3] Escalera, S., O. Pujol, and P. Radeva. “On the decoding process in ternary error-correcting output codes.” *IEEE Transactions on Pattern Analysis and Machine Intelligence*. Vol. 32, Issue 7, 2010, pp. 120–134.
- [4] Escalera, S., O. Pujol, and P. Radeva. “Separability of ternary codes for sparse designs of error-correcting output codes.” *Pattern Recogn.* Vol. 30, Issue 3, 2009, pp. 285–297.
- [5] Hastie, T., and R. Tibshirani. “Classification by Pairwise Coupling.” *Annals of Statistics*. Vol. 26, Issue 2, 1998, pp. 451–471.
- [6] Wu, T. F., C. J. Lin, and R. Weng. “Probability Estimates for Multi-Class Classification by Pairwise Coupling.” *Journal of Machine Learning Research*. Vol. 5, 2004, pp. 975–1005.
- [7] Zadrozny, B. “Reducing Multiclass to Binary by Coupling Probability Estimates.” *NIPS 2001: Proceedings of Advances in Neural Information Processing Systems 14*, 2001, pp. 1041–1048.

## See Also

`ClassificationECOC` | `ClassificationPartitionedModel` | `edge` | `fitcecoc` | `predict` | `quadprog` | `statset`

## Topics

“Quick Start Parallel Computing for Statistics and Machine Learning Toolbox” on page 30-2

“Reproducibility in Parallel Statistical Computations” on page 30-14

“Concepts of Parallel Computing in Statistics and Machine Learning Toolbox” on page 30-8

# kfoldPredict

**Class:** ClassificationPartitionedLinear

Predict labels for observations not used for training

## Syntax

```
Label = kfoldPredict(CVMdl)
[Label,Score] = kfoldPredict(CVMdl)
```

## Description

`Label = kfoldPredict(CVMdl)` returns cross-validated class labels predicted by the cross-validated, binary, linear classification model `CVMdl`. That is, for every fold, `kfoldPredict` predicts class labels for observations that it holds out when it trains using all other observations.

`Label` contains predicted class labels for each regularization strength in the linear classification models that compose `CVMdl`.

`[Label,Score] = kfoldPredict(CVMdl)` also returns cross-validated classification scores on page 32-3339 for both classes. `Score` contains classification scores for each regularization strength in `CVMdl`.

## Input Arguments

**`CVMdl` — Cross-validated, binary, linear classification model**

`ClassificationPartitionedLinear` model object

Cross-validated, binary, linear classification model, specified as a `ClassificationPartitionedLinear` model object. You can create a `ClassificationPartitionedLinear` model using `fitclinear` and specifying any one of the cross-validation, name-value pair arguments, for example, `CrossVal`.

To obtain estimates, `ClassificationPartitionedLinear.kfoldPredict` applies the same data used to cross-validate the linear classification model (`X` and `Y`).

## Output Arguments

### **Label** — Cross-validated, predicted class labels

categorical array | character array | logical matrix | numeric matrix | cell array of character vectors

Cross-validated, predicted class labels, returned as a categorical or character array, logical or numeric matrix, or cell array of character vectors.

In most cases, `Label` is an  $n$ -by- $L$  array of the same data type as the observed class labels (see `Y`) used to create `CVMDL`.  $n$  is the number of observations in the predictor data (see `X`) and  $L$  is the number of regularization strengths in `CVMDL.Trained{1}.Lambda`. That is, `Label(i, j)` is the predicted class label for observation  $i$  using the linear classification model that has regularization strength `CVMDL.Trained{1}.Lambda(j)`.

If `Y` is a character array and  $L > 1$ , then `Label` is a cell array of class labels.

### **Score** — Cross-validated classification scores

numeric array

Cross-validated classification scores on page 32-3339, returned as an  $n$ -by-2-by- $L$  numeric array.  $n$  is the number of observations in the predictor data that created `CVMDL` (see `X`) and  $L$  is the number of regularization strengths in `CVMDL.Trained{1}.Lambda`. `Score(i, k, j)` is the score for classifying observation  $i$  into class  $k$  using the linear classification model that has regularization strength `CVMDL.Trained{1}.Lambda(j)`. `CVMDL.ClassNames` stores the order of the classes.

If `CVMDL.Trained{1}.Learner` is 'logistic', then classification scores are posterior probabilities.

## Examples



## Predict *k*-fold Cross-Validation Labels

Load the NLP data set.

```
load nlpdata
```

$X$  is a sparse matrix of predictor data, and  $Y$  is a categorical vector of class labels. There are more than two classes in the data.

The models should identify whether the word counts in a web page are from the Statistics and Machine Learning Toolbox™ documentation. So, identify the labels that correspond to the Statistics and Machine Learning Toolbox™ documentation web pages.

```
Ystats = Y == 'stats';
```

Cross-validate a binary, linear classification model using the entire data set, which can identify whether the word counts in a documentation web page are from the Statistics and Machine Learning Toolbox™ documentation.

```
rng(1); % For reproducibility
CVMdl = fitclinear(X,Ystats,'CrossVal','on');
Mdl1 = CVMdl.Trained{1}
```

```
Mdl1 =

    ClassificationLinear
      ResponseName: 'Y'
      ClassNames: [0 1]
    ScoreTransform: 'none'
           Beta: [34023x1 double]
           Bias: -1.0008
           Lambda: 3.5193e-05
           Learner: 'svm'
```

CVMdl is a ClassificationPartitionedLinear model. By default, the software implements 10-fold cross validation. You can alter the number of folds using the 'KFold' name-value pair argument.

Predict labels for the observations that fitclinear did not use in training the folds.

```
label = kfoldPredict(CVMdl);
```

Because there is one regularization strength in `Mdl1`, `label` is a column vector of predictions containing as many rows as observations in `X`.

Construct a confusion matrix.

```
ConfusionTrain = confusionmat(Ystats,label)
```

```
ConfusionTrain =  
  
      30009      9  
      15      1539
```

The model misclassifies 15 'stats' documentation pages as being outside of the Statistics and Machine Learning Toolbox documentation, and misclassifies nine pages as 'stats' pages.

### Estimate *k*-fold Cross-Validation Posterior Class Probabilities

Linear classification models return posterior probabilities for logistic regression learners only.

Load the NLP data set and preprocess it as in “Predict *k*-fold Cross-Validation Labels” on page 32-3330. Transpose the predictor data matrix.

```
load nlpdata  
Ystats = Y == 'stats';  
X = X';
```

Cross-validate binary, linear classification models using 5-fold cross-validation. Optimize the objective function using SpARSA. Lower the tolerance on the gradient of the objective function to  $1e-8$ .

```
rng(10); % For reproducibility  
CVMdl = fitlinear(X,Ystats,'ObservationsIn','columns',...  
    'KFold',5,'Learner','logistic','Solver','sparsa',...  
    'Regularization','lasso','GradientTolerance',1e-8);
```

Predict the posterior class probabilities for observations not used to train each fold.

```
[~,posterior] = kfoldPredict(CVMdl);  
CVMdl.ClassNames
```

```
ans =  
  
    2x1 logical array  
  
    0  
    1
```

Because there is one regularization strength in `CVMD1`, `posterior` is a matrix with 2 columns and rows equal to the number of observations. Column  $i$  contains posterior probabilities of `Mdl.ClassNames(i)` given a particular observation.

Obtain false and true positive rates, and estimate the AUC. Specify that the second class is the positive class.

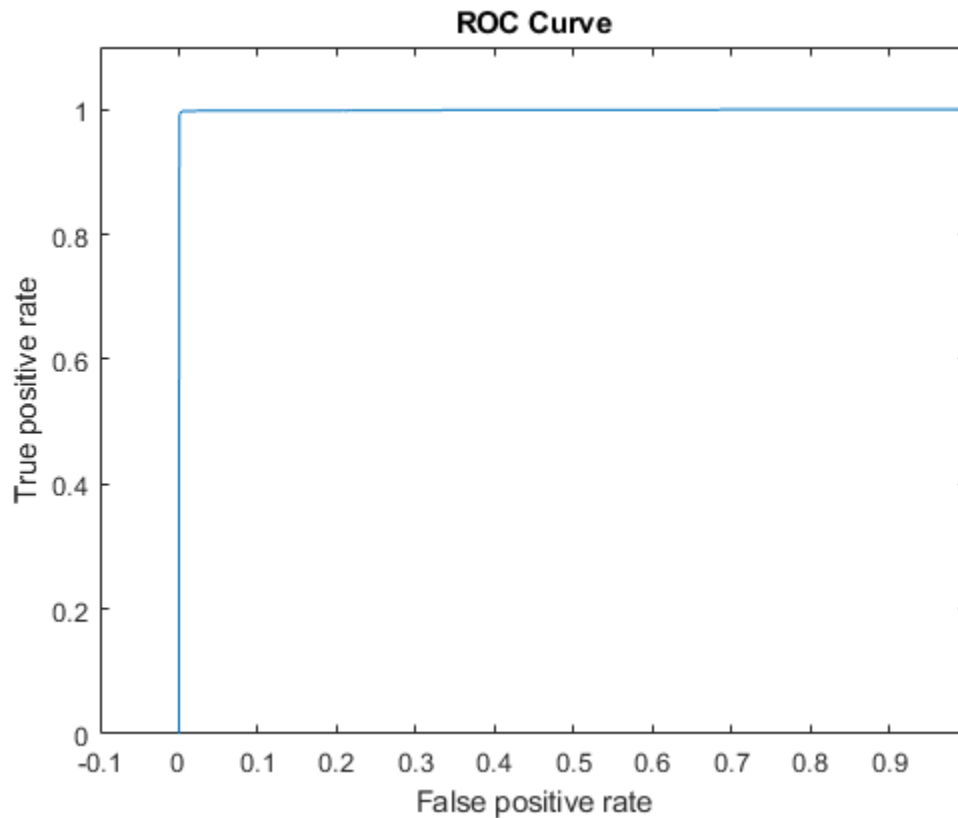
```
[fpr,tpr,~,auc] = perfcurve(Ystats,posterior(:,2),CVMD1.ClassNames(2));  
auc
```

```
auc =  
  
    0.9990
```

The AUC is 0.9990, which indicates a model that predicts well.

Plot an ROC curve.

```
figure;  
plot(fpr,tpr)  
h = gca;  
h.XLim(1) = -0.1;  
h.YLim(2) = 1.1;  
xlabel('False positive rate')  
ylabel('True positive rate')  
title('ROC Curve')
```



The ROC curve indicates that the model classifies almost perfectly.

### Find Good Lasso Penalty Using Cross-Validated AUC

To determine a good lasso-penalty strength for a linear classification model that uses a logistic regression learner, compare cross-validated AUC values.

Load the NLP data set. Preprocess the data as in “Estimate k-fold Cross-Validation Posterior Class Probabilities” on page 32-3332.

```
load nlpdata
Ystats = Y == 'stats';
X = X';
```

There are 9471 observations in the test sample.

Create a set of 11 logarithmically-spaced regularization strengths from  $10^{-6}$  through  $10^{-0.5}$ .

```
Lambda = logspace(-6,-0.5,11);
```

Cross-validate a binary, linear classification models that use each of the regularization strengths and 5-fold cross-validation. Optimize the objective function using SpARSA. Lower the tolerance on the gradient of the objective function to  $1e-8$ .

```
rng(10); % For reproducibility
CVMdl = fitlinear(X,Ystats,'ObservationsIn','columns',...
    'KFold',5,'Learner','logistic','Solver','sparsa',...
    'Regularization','lasso','Lambda',Lambda,'GradientTolerance',1e-8)
Mdl1 = CVMdl.Trained{1}
```

```
CVMdl =
```

```
classreg.learning.partition.ClassificationPartitionedLinear
  CrossValidatedModel: 'Linear'
  ResponseName: 'Y'
  NumObservations: 31572
  KFold: 5
  Partition: [1x1 cvpartition]
  ClassNames: [0 1]
  ScoreTransform: 'none'
```

```
Mdl1 =
```

```
ClassificationLinear
  ResponseName: 'Y'
  ClassNames: [0 1]
  ScoreTransform: 'logit'
  Beta: [34023x11 double]
  Bias: [1x11 double]
  Lambda: [1x11 double]
```

```
Learner: 'logistic'
```

Mdl1 is a `ClassificationLinear` model object. Because `Lambda` is a sequence of regularization strengths, you can think of `Mdl1` as 11 models, one for each regularization strength in `Lambda`.

Predict the cross-validated labels and posterior class probabilities.

```
[label,posterior] = kfoldPredict(CVMdl);  
CVMdl.ClassNames;  
[n,K,L] = size(posterior)  
posterior(3,1,5)
```

```
n =
```

```
31572
```

```
K =
```

```
2
```

```
L =
```

```
11
```

```
ans =
```

```
1.0000
```

`label` is a 31572-by-11 matrix of predicted labels. Each column corresponds to the predicted labels of the model trained using the corresponding regularization strength. `posterior` is a 31572-by-2-by-11 matrix of posterior class probabilities. Columns correspond to classes and pages correspond to regularization strengths. For example, `posterior(3,1,5)` indicates that the posterior probability that the first class (label 0) is assigned to observation 3 by the model that uses `Lambda(5)` as a regularization strength is 1.0000.

For each model, compute the AUC. Designate the second class as the positive class.

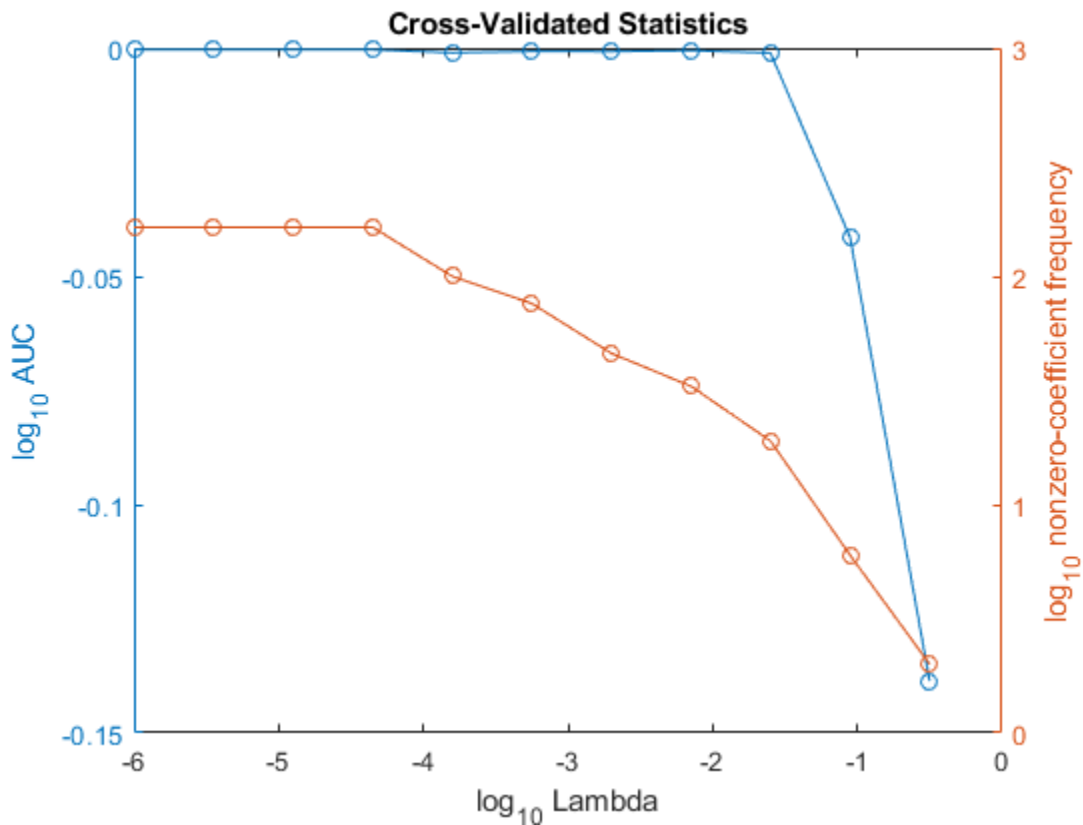
```
auc = 1:numel(Lambda); % Preallocation
for j = 1:numel(Lambda)
    [~,~,~,auc(j)] = perfcurve(Ystats,posterior(:,2,j),CVmdl.ClassNames(2));
end
```

Higher values of Lambda lead to predictor variable sparsity, which is a good quality of a classifier. For each regularization strength, train a linear classification model using the entire data set and the same options as when you trained the model. Determine the number of nonzero coefficients per model.

```
Mdl = fitclinear(X,Ystats,'ObservationsIn','columns',...
    'Learner','logistic','Solver','sparsa','Regularization','lasso',...
    'Lambda',Lambda,'GradientTolerance',1e-8);
numNZCoeff = sum(Mdl.Beta~=0);
```

In the same figure, plot the test-sample error rates and frequency of nonzero coefficients for each regularization strength. Plot all variables on the log scale.

```
figure;
[h,hL1,hL2] = plotyy(log10(Lambda),log10(auc),...
    log10(Lambda),log10(numNZCoeff + 1));
hL1.Marker = 'o';
hL2.Marker = 'o';
ylabel(h(1),'log_{10} AUC')
ylabel(h(2),'log_{10} nonzero-coefficient frequency')
xlabel('log_{10} Lambda')
title('Cross-Validated Statistics')
hold off
```



Choose the index of the regularization strength that balances predictor variable sparsity and high AUC. In this case, a value between  $10^{-3}$  to  $10^{-1}$  should suffice.

```
idxFinal = 9;
```

Select the model from `Mdl` with the chosen regularization strength.

```
MdlFinal = selectModels(Mdl,idxFinal);
```



`MdlFinal` is a `ClassificationLinear` model containing one regularization strength. To estimate labels for new observations, pass `MdlFinal` and the new data to `predict`.

## Definitions

### Classification Score

For linear classification models, the raw classification score for classifying the observation  $x$ , a row vector, into the positive class is defined by

$$f_j(x) = x\beta_j + b_j.$$

For the model with regularization strength  $j$ ,  $\beta_j$  is the estimated column vector of coefficients (the model property `Beta(:, j)`) and  $b_j$  is the estimated, scalar bias (the model property `Bias(j)`).

The raw classification score for classifying  $x$  into the negative class is  $-f(x)$ . The software classifies observations into the class that yields the positive score.

If the linear classification model consists of logistic regression learners, then the software applies the 'logit' score transformation to the raw classification scores (see `ScoreTransform`).

### See Also

`ClassificationLinear` | `ClassificationPartitionedLinear` | `confusionmat` | `perfcurve` | `predict` | `testcholdout`

Introduced in R2016a

## kfoldPredict

**Class:** ClassificationPartitionedLinearECOC

Predict labels for observations not used for training

### Syntax

```
Label = kfoldPredict(CVMdl)
Label = kfoldPredict(CVMdl, Name, Value)
[Label, NegLoss, PBScore] = kfoldPredict(____)
[Label, NegLoss, PBScore, Posterior] = kfoldPredict(____)
```

### Description

`Label = kfoldPredict(CVMdl)` returns class labels predicted by the cross-validated ECOC model composed of linear classification models `CVMdl`. That is, for every fold, `kfoldPredict` predicts class labels for observations that it holds out when it trains using all other observations. `kfoldPredict` applies the same data used create `CVMdl` (see `fitcecoc`).

Also, `Label` contains class labels for each regularization strength in the linear classification models that compose `CVMdl`.

`Label = kfoldPredict(CVMdl, Name, Value)` returns predicted class labels with additional options specified by one or more `Name, Value` pair arguments. For example, specify the posterior probability estimation method, decoding scheme, or verbosity level.

`[Label, NegLoss, PBScore] = kfoldPredict(____)` additionally returns, for held-out observations and each regularization strength:

- Negated values of the average binary loss per class (`NegLoss`).
- Positive-class scores (`PBScore`) for each binary learner.

`[Label, NegLoss, PBScore, Posterior] = kfoldPredict(____)` additionally returns posterior class probability estimates for held-out observations and for each

regularization strength. To return posterior probabilities, the linear classification model learners must be logistic regression models.

## Input Arguments

### **cvm1** — Cross-validated, ECOC model composed of linear classification models

`ClassificationPartitionedLinearECOC` model object

Cross-validated, ECOC model composed of linear classification models, specified as a `ClassificationPartitionedLinearECOC` model object. You can create a `ClassificationPartitionedLinearECOC` model using `fitcecoc` and by:

- 1 Specifying any one of the cross-validation, name-value pair arguments, for example, `CrossVal`
- 2 Setting the name-value pair argument `Learners` to `'linear'` or a linear classification model template returned by `templateLinear`

To obtain estimates, `ClassificationPartitionedLinearECOC.kfoldPredict` applies the same data used to cross-validate the ECOC model ( $X$  and  $Y$ ).

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### **BinaryLoss** — Binary learner loss function

`'hamming'` | `'linear'` | `'logit'` | `'exponential'` | `'binodeviance'` | `'hinge'`  
| `'quadratic'` | function handle

Binary learner loss function, specified as the comma-separated pair consisting of `'BinaryLoss'` and a built-in, loss-function name or function handle.

- This table contains names and descriptions of the built-in functions, where  $y_j$  is a class label for a particular binary learner (in the set  $\{-1, 1, 0\}$ ),  $s_j$  is the score for observation  $j$ , and  $g(y_j, s_j)$  is the binary loss formula.

Value	Description	Score Domain	$g(y_j, s_j)$
'binodeviance'	Binomial deviance	$(-\infty, \infty)$	$\log[1 + \exp(-2y_j s_j)] / [2\log(2)]$
'exponential'	Exponential	$(-\infty, \infty)$	$\exp(-y_j s_j) / 2$
'hamming'	Hamming	$[0, 1]$ or $(-\infty, \infty)$	$[1 - \text{sign}(y_j s_j)] / 2$
'hinge'	Hinge	$(-\infty, \infty)$	$\max(0, 1 - y_j s_j) / 2$
'linear'	Linear	$(-\infty, \infty)$	$(1 - y_j s_j) / 2$
'logit'	Logistic	$(-\infty, \infty)$	$\log[1 + \exp(-y_j s_j)] / [2\log(2)]$
'quadratic'	Quadratic	$[0, 1]$	$[1 - y_j(2s_j - 1)]^2 / 2$

The software normalizes the binary losses such that the loss is 0.5 when  $y_j = 0$ . Also, the software calculates the mean binary loss for each class.

- For a custom binary loss function, e.g., `customFunction`, specify its function handle `'BinaryLoss', @customFunction`.

`customFunction` should have this form

```
bLoss = customFunction(M, s)
```

where:

- $M$  is the  $K$ -by- $L$  coding matrix stored in `Mdl.CodingMatrix`.
- $s$  is the 1-by- $L$  row vector of classification scores.
- `bLoss` is the classification loss. This scalar aggregates the binary losses for every learner in a particular class. For example, you can use the mean binary loss to aggregate the loss over the learners for each class.
- $K$  is the number of classes.
- $L$  is the number of binary learners.

For an example of passing a custom binary loss function, see “Predict Test-Sample Labels of ECOC Models Using Custom Binary Loss Function” on page 32-4933.

By default, if all binary learners are linear classification models using:

- SVM, then `BinaryLoss` is `'hinge'`
- Logistic regression, then `BinaryLoss` is `'quadratic'`

Example: `'BinaryLoss', 'binodeviance'`

Data Types: `char` | `function_handle`

### **Decoding — Decoding scheme**

`'lossweighted'` (default) | `'lossbased'`

Decoding scheme that aggregates the binary losses on page 32-4941, specified as the comma-separated pair consisting of `'Decoding'` and `'lossweighted'` or `'lossbased'`.

Example: `'Decoding', 'lossbased'`

Data Types: `char`

### **NumKLInitializations — Number of random initial values**

0 (default) | nonnegative integer

Number of random initial values for fitting posterior probabilities by Kullback-Leibler divergence minimization, specified as the comma-separated pair consisting of `'NumKLInitializations'` and a nonnegative integer.

To use this option, you must:

- Return the fourth output argument (`Posterior`).
- The linear classification models that compose the ECOC models must use logistic regression learners (that is, `CVMdl.Trained{1}.BinaryLearners{1}.Learner` must be `'logistic'`).
- `PosteriorMethod` must be `'kl'`.

For more details, see “Posterior Estimation Using Kullback-Leibler Divergence” on page 32-4943.

Example: `'NumKLInitializations', 5`

Data Types: `single` | `double`

### **Options — Estimation options**

`[]` (default) | structure array returned by `statset`

Estimation options, specified as the comma-separated pair consisting of `'Options'` and a structure array returned by `statset`.

To invoke parallel computing:

- You need a Parallel Computing Toolbox license.
- Specify `'Options', statset('UseParallel',1)`.

### **PosteriorMethod — Posterior probability estimation method**

`'kl'` (default) | `'qp'`

Posterior probability estimation method, specified as the comma-separated pair consisting of `'PosteriorMethod'` and `'kl'` or `'qp'`.

- To use this option, you must return the fourth output argument (`Posterior`) and the linear classification models that compose the ECOC models must use logistic regression learners (that is, `CVMdl.Trained{1}.BinaryLearners{1}.Learner` must be `'logistic'`).
- If `PosteriorMethod` is `'kl'`, then the software estimates multiclass posterior probabilities by minimizing the Kullback-Leibler divergence between the predicted and expected posterior probabilities returned by binary learners. For details, see “Posterior Estimation Using Kullback-Leibler Divergence” on page 32-4943.
- If `PosteriorMethod` is `'qp'`, then the software estimates multiclass posterior probabilities by solving a least-squares problem using quadratic programming. You need an Optimization Toolbox license to use this option. For details, see “Posterior Estimation Using Quadratic Programming” on page 32-4945.

Example: `'PosteriorMethod','qp'`

Data Types: `char`

### **Verbose — Verbosity level**

`0` (default) | `1`

Verbosity level, specified as the comma-separated pair consisting of `'Verbose'` and `0` or `1`. `Verbose` controls the amount of diagnostic messages that the software displays in the Command Window.

If `Verbose` is `0`, then the software does not display diagnostic messages. Otherwise, the software displays diagnostic messages.

Example: `'Verbose',1`

Data Types: `single` | `double`

## Output Arguments

### **Label** — Cross-validated, predicted class labels

categorical array | character array | logical matrix | numeric matrix | cell array of character vectors

Cross-validated, predicted class labels, returned as a categorical or character array, logical or numeric matrix, or cell array of character vectors.

In most cases, `Label` is an  $n$ -by- $L$  array of the same data type as the observed class labels ( $Y$ ) used to create `CVMDL`.  $n$  is the number of observations in the predictor data ( $X$ ) and  $L$  is the number of regularization strengths in the linear classification models that compose the cross-validated ECOC model. That is, `Label(i, j)` is the predicted class label for observation  $i$  using the ECOC model of linear classification models that has regularization strength `CVMDL.Trained{1}.BinaryLearners{1}.Lambda(j)`.

If  $Y$  is a character array and  $L > 1$ , then `Label` is a cell array of class labels.

The software assigns the predicted label corresponding to the class with the largest, negated, average binary loss (`NegLoss`), or, equivalently, the smallest average binary loss.

### **NegLoss** — Cross-validated, negated, average binary losses

numeric array

Cross-validated, negated, average binary losses, returned as an  $n$ -by- $K$ -by- $L$  numeric matrix or array.  $K$  is the number of distinct classes in the training data and columns correspond to the classes in `CVMDL.ClassNames`. For  $n$  and  $L$ , see `Label`.

`NegLoss(i, k, j)` is the negated, average binary loss for classifying observation  $i$  into class  $k$  using the linear classification model that has regularization strength `CVMDL.Trained{1}.BinaryLoss{1}.Lambda(j)`.

### **PBScore** — Cross-validated, positive-class scores

numeric array

Cross-validated, positive-class scores, returned as an  $n$ -by- $B$ -by- $L$  numeric array.  $B$  is the number of binary learners in the cross-validated ECOC model and columns correspond to the binary learners in `CVMDL.Trained{1}.BinaryLearners`. For  $n$  and  $L$ , see `Label`.

`PBScore(i, b, j)` is the positive-class score of binary learner  $b$  for classifying observation  $i$  into its positive class, using the linear classification model that has regularization strength `CVMDL.Trained{1}.BinaryLearners{1}.Lambda(j)`.

If the coding matrix varies across folds (that is, if the coding scheme is `sparserandom` or `denserandom`), then `PBScore` is empty (`[]`).

### **Posterior — Cross-validated posterior class probabilities**

numeric array

Cross-validated posterior class probabilities, returned as an  $n$ -by- $K$ -by- $L$  numeric array. For dimension definitions, see `NegLoss`. `Posterior(i, k, j)` is the posterior probability for classifying observation  $i$  into class  $k$  using the linear classification model that has regularization strength `CVMDL.Trained{1}.BinaryLearners{1}.Lambda(j)`.

To return posterior probabilities, `CVMDL.Trained{1}.BinaryLearner{1}.Learner` must be `'logistic'`.

## Examples

### **Predict $k$ -fold Cross-Validation Labels**

Load the NLP data set.

```
load nlpdata
```

$X$  is a sparse matrix of predictor data, and  $Y$  is a categorical vector of class labels.

Cross-validate an ECOC model of linear classification models.

```
rng(1); % For reproducibility
CVMDL = fitcecoc(X, Y, 'Learner', 'linear', 'CrossVal', 'on');
```

`CVMDL` is a `ClassificationPartitionedLinearECOC` model. By default, the software implements 10-fold cross validation.

Predict labels for the observations that `fitcecoc` did not use in training the folds.

```
label = kfoldPredict(CVMDL);
```

Because there is one regularization strength in `CVMDL`, `label` is a column vector of predictions containing as many rows as observations in  $X$ .

Construct a confusion matrix.



```
ConfusionTrain = confusionmat(Y,label)
CVMdl.ClassNames
```

```
ConfusionTrain =
```

```
Columns 1 through 6
```

2488	107	0	2	10	26
97	3892	2	18	13	21
1	7	1143	19	6	2
1	9	16	939	7	1
8	13	6	15	988	1
22	28	0	0	0	2051
15	22	1	6	5	21
39	140	55	44	27	21
0	0	0	2	0	2
19	30	80	5	24	8
3	0	1	3	0	4
19	199	0	7	2	12
8	14	8	2	12	1

```
Columns 7 through 12
```

2	218	2	30	3	16
2	305	1	34	1	115
0	104	0	45	5	2
0	81	1	11	2	3
0	45	0	22	0	4
1	61	1	11	0	3
2244	96	3	37	4	9
21	4537	0	55	3	49
0	2	1544	2	1	1
1	47	1	2434	0	8
4	6	3	3	2381	1
3	103	1	21	0	1442
0	32	0	19	0	19

```
Column 13
```

```
0
1
4
0
1
```

```
0
4
7
0
2
0
2
2463
```

```
ans =
```

```
13x1 categorical array
```

```
comm
dsp
ecoder
fixedpoint
hdlcoder
phased
physmod
simulink
stats
supportpkg
symbolic
vision
xpc
```

Rows correspond to the true class and columns to the predicted class. The dimensions correspond to the order in `CVMdl.ClassNames`.

### Specify Custom Binary Loss

Load the NLP data set. Transpose the predictor data.

```
load nlpdata
X = X';
```

For simplicity, use the label 'others' for all observations in `Y` that are not 'simulink', 'dsp', or 'comm'.

```
Y(~(ismember(Y,{'simulink','dsp','comm'}))) = 'others';
```

Create a linear classification model template that specifies optimizing the objective function using SpARSA.

```
t = templateLinear('Solver','sparsa');
```

Cross-validate an ECOC model of linear classification models using 5-fold cross-validation. Specify that the predictor observations correspond to columns.

```
rng(1); % For reproducibility
CVMdl = fitcecoc(X,Y,'Learners',t,'KFold',5,'ObservationsIn','columns');
CMdl1 = CVMdl.Trained{1}
```

```
CMdl1 =
```

```
classreg.learning.classif.CompactClassificationECOC
  ResponseName: 'Y'
  ClassNames: [comm dsp simulink others]
  ScoreTransform: 'none'
  BinaryLearners: {6x1 cell}
  CodingMatrix: [4x6 double]
```

CVMdl is a `ClassificationPartitionedLinearECOC` model. It contains the property `Trained`, which is a 5-by-1 cell array holding a `CompactClassificationECOC` models that the software trained using the training set of each fold.

By default, the linear classification models that compose the ECOC models use SVMs. SVM scores are signed distances from the observation to the decision boundary.

Therefore, the domain is  $(-\infty, \infty)$ . Create a custom binary loss function that:

- Maps the coding design matrix ( $M$ ) and positive-class classification scores ( $s$ ) for each learner to the binary loss for each observation
- Uses linear loss
- Aggregates the binary learner loss using the median.

You can create a separate function for the binary loss function, and then save it on the MATLAB® path. Or, you can specify an anonymous binary loss function.

```
customBL = @(M,s)nanmedian(1 - bsxfun(@times,M,s),2)/2;
```

Predict cross-validation labels and estimate the median binary loss per class. Print the median negative binary losses per class for a random set of 10 out-of-fold observations.

```
[label,NegLoss] = kfoldPredict(CVMdl,'BinaryLoss',customBL);

idx = randsample(numel(label),10);
table(Y(idx),label(idx),NegLoss(idx,1),NegLoss(idx,2),NegLoss(idx,3),...
      NegLoss(idx,4),'VariableNames',{ 'True'; 'Predicted' };...
      categories(CVMdl.ClassNames))

ans =

10x6 table

      True      Predicted      comm      dsp      simulink      others
-----
others      others      -1.2319    -1.0488    0.048758    1.6175
simulink    simulink    -16.407    -12.218    21.531     11.218
dsp         dsp         -0.7387    -0.11534   -0.88466    -0.2613
others      others      -0.1251    -0.8749    -0.99766    0.14517
dsp         dsp         2.5867     6.4187     -3.5867     -4.4165
others      others      -0.025358  -1.2287    -0.97464    0.19747
others      others      -2.6725    -0.56708   -0.51092    2.7453
others      others      -1.1605    -0.88321   -0.11679    0.43504
others      others      -1.9511    -1.3175    0.24735     0.95111
simulink    others      -7.848     -5.8203    4.8203     6.8457
```

The software predicts the label based on the maximum negated loss.

### Estimate Posterior Class Probabilities

ECOC models composed of linear classification models return posterior probabilities for logistic regression learners only. This example requires the Parallel Computing Toolbox™ and the Optimization Toolbox™

Load the NLP data set and preprocess the data as in “Specify Custom Binary Loss” on page 32-3348.

```
load nlpdata
X = X';
Y(~(ismember(Y,{'simulink','dsp','comm'}))) = 'others';
```

Create a set of 5 logarithmically-spaced regularization strengths from  $10^{-5}$  through  $10^{-0.5}$ .

```
Lambda = logspace(-6,-0.5,5);
```

Create a linear classification model template that specifies optimizing the objective function using SpARSA and to use logistic regression learners.

```
t = templateLinear('Solver','sparsa','Learner','logistic','Lambda',Lambda);
```

Cross-validate an ECOC model of linear classification models using 5-fold cross-validation. Specify that the predictor observations correspond to columns, and to use parallel computing.

```
rng(1); % For reproducibility
Options = statset('UseParallel',true);
CVMdl = fitcecoc(X,Y,'Learners',t,'KFold',5,'ObservationsIn','columns',...
    'Options',Options);
```

```
Starting parallel pool (parpool) using the 'local' profile ...
connected to 6 workers.
```

Predict the cross-validated posterior class probabilities. Specify to use parallel computing and to estimate posterior probabilities using quadratic programming.

```
[label,~,~,Posterior] = kfoldPredict(CVMdl,'Options',Options,...
    'PosteriorMethod','qp');
size(label)
label(3,4)
size(Posterior)
Posterior(3,:,4)
```

```
ans =
```

```
    31572         5
```

```
ans =
```

```
    categorical
```

```

        others

ans =

        31572         4         5

ans =

    0.0293    0.0373    0.1738    0.7596

```

Because there are five regularization strengths:

- `label` is a 31572-by-5 categorical array. `label(3,4)` is the predicted, cross-validated label for observation 3 using the model trained with regularization strength `Lambda(4)`.
- `Posterior` is a 31572-by-4-by-5 matrix. `Posterior(3,:,4)` is the vector of all estimated, posterior class probabilities for observation 3 using the model trained with regularization strength `Lambda(4)`. The order of the second dimension corresponds to `CVMDL.ClassNames`. Display a random set of 10 posterior class probabilities.

Display a random sample of cross-validated labels and posterior probabilities for the model trained using `Lambda(4)`.

```

idx = randsample(size(label,1),10);
table(Y(idx),label(idx,4),Posterior(idx,1,4),Posterior(idx,2,4),...
      Posterior(idx,3,4),Posterior(idx,4,4),...
      'VariableNames',{'True'};{'Predicted'};categories(CVMDL.ClassNames)])

ans =

10×6 table

      True      Predicted      comm      dsp      simulink      others
      -----      -----      -----      -----      -----      -----
others      others      0.030309      0.022454      0.10401      0.84323
simulink    simulink    3.5104e-05      4.3154e-05      0.99877      0.0011543
dsp         others      0.15837       0.25784       0.18567      0.39811

```

others	others	0.093212	0.063752	0.12927	0.71376
dsp	dsp	0.0057401	0.89678	0.014939	0.082538
others	others	0.085715	0.054451	0.083765	0.77607
others	others	0.0061121	0.0057884	0.02409	0.96401
others	others	0.066741	0.074103	0.168	0.69115
others	others	0.05236	0.025631	0.13245	0.78956
simulink	simulink	0.00039812	0.00045575	0.73724	0.2619

- “Quick Start Parallel Computing for Statistics and Machine Learning Toolbox” on page 30-2

## Definitions

### Binary Loss

A binary loss is a function of the class and classification score that determines how well a binary learner classifies an observation into the class.

Let:

- $m_{kj}$  be element  $(k,j)$  of the coding design matrix  $M$  (i.e., the code corresponding to class  $k$  of binary learner  $j$ )
- $s_j$  be the score of binary learner  $j$  for an observation
- $g$  be the binary loss function
- $\hat{k}$  be the predicted class for the observation

In loss-based decoding [15], the class producing the minimum sum of the binary losses over binary learners determines the predicted class of an observation, that is,

$$\hat{k} = \underset{k}{\operatorname{argmin}} \sum_{j=1}^L |m_{kj}| g(m_{kj}, s_j).$$

In loss-weighted decoding [15], the class producing the minimum average of the binary losses over binary learners determines the predicted class of an observation, that is,

$$\hat{k} = \operatorname{argmin}_k \frac{\sum_{j=1}^L |m_{kj}| g(m_{kj}, s_j)}{\sum_{j=1}^L |m_{kj}|}.$$

Allwein et al. [2] suggest that loss-weighted decoding improves classification accuracy by keeping loss values for all classes in the same dynamic range.

This table summarizes the supported loss functions, where  $y_j$  is a class label for a particular binary learner (in the set  $\{-1, 1, 0\}$ ),  $s_j$  is the score for observation  $j$ , and  $g(y_j, s_j)$ .

Value	Description	Score Domain	$g(y_j, s_j)$
'binodeviance'	Binomial deviance	$(-\infty, \infty)$	$\log[1 + \exp(-2y_j s_j)] / [2\log(2)]$
'exponential'	Exponential	$(-\infty, \infty)$	$\exp(-y_j s_j) / 2$
'hamming'	Hamming	$[0, 1]$ or $(-\infty, \infty)$	$[1 - \operatorname{sign}(y_j s_j)] / 2$
'hinge'	Hinge	$(-\infty, \infty)$	$\max(0, 1 - y_j s_j) / 2$
'linear'	Linear	$(-\infty, \infty)$	$(1 - y_j s_j) / 2$
'logit'	Logistic	$(-\infty, \infty)$	$\log[1 + \exp(-y_j s_j)] / [2\log(2)]$
'quadratic'	Quadratic	$[0, 1]$	$[1 - y_j(2s_j - 1)]^2 / 2$

The software normalizes the binary losses such that the loss is 0.5 when  $y_j = 0$ , and aggregates using the average of the binary learners [2].

Do not confuse the binary loss with the overall classification loss (specified by the LossFun name-value pair argument of `predict` and `loss`), e.g., classification error, which measures how well an ECOC classifier performs as a whole.

## Algorithms

- The software can estimate class posterior probabilities using quadratic programming or by minimizing the Kullback-Leibler divergence. For the following descriptions of the posterior estimation algorithms, let:
  - $m_{kj}$  be the element  $(k, j)$  of the coding design matrix  $M$ .



- $I$  be the indicator function.
- $\hat{p}_k$  be the class posterior probability estimate for class  $k$  of an observation,  $k = 1, \dots, K$ .
- $r_j$  be the positive-class posterior probability for binary learner  $j$ . That is,  $r_j$  is the probability that binary learner  $j$  classifies an observation into the positive class, given the training data.

## Posterior Estimation Using Kullback-Leibler Divergence

By default, the software minimizes the Kullback-Leibler divergence to estimate class posterior probabilities. The Kullback-Leibler divergence between the expected and observed positive-class posterior probabilities is

$$\Delta(r, r) = \sum_{j=1}^L w_j \left[ r_j \log \frac{r_j}{r_j} + (1 - r_j) \log \frac{1 - r_j}{1 - r_j} \right],$$

where  $w_j = \sum_{S_j} w_i^*$  is the weight for binary learner  $j$  with  $S_j$  the set of observation indices that binary learner  $j$  is trained on and  $w_i^*$  is the weight of observation  $i$ . The software minimizes the divergence iteratively. The first step is to choose initial values

$\hat{p}_k^{(0)}$ ;  $k = 1, \dots, K$  for the class posterior probabilities.

- If you do not specify `NumKLIterations`, then the software uses both sets of deterministic initial values described next, and uses the one that minimizes  $\Delta$ .

- $\hat{p}_k^{(0)} = 1 / K$ ;  $k = 1, \dots, K$ .

- $\hat{p}_k^{(0)}$ ;  $k = 1, \dots, K$  is the solution of the system

$$M_{01} \hat{p}^{(0)} = r,$$

where  $M_{01}$  is  $M$  with all  $m_{kj} = -1$  replaced with 0, and  $r$  is a vector of positive-class posterior probabilities returned by the  $L$  binary learners [14]. The software uses `lsqnonneg` to solve the system.

- If you specify 'NumKLIterations',  $c$ , where  $c$  is a natural number, then the software does the following to choose  $\hat{p}_k^{(0)}$ ;  $k = 1, \dots, K$ , and uses the one that minimizes  $\Delta$ .
  - The software chooses both sets of deterministic initial values as described previously.
  - The software randomly generates  $c$  vectors of length  $K$  using `rand`, and then normalizes each vector to sum to 1.

At iteration  $t$ , the software:

- 1 Computes

$$\hat{r}_j^{(t)} = \frac{\sum_{k=1}^K \hat{p}_k^{(t)} I(m_{kj} = +1)}{\sum_{k=1}^K \hat{p}_k^{(t)} I(m_{kj} = +1 \cup m_{kj} = -1)}.$$

- 2 Estimates the next class posterior probability using

$$\hat{p}_k^{(t+1)} = \hat{p}_k^{(t)} \frac{\sum_{j=1}^L w_j [r_j I(m_{kj} = +1) + (1 - r_j) I(m_{kj} = -1)]}{\sum_{j=1}^L w_j [\hat{r}_j^{(t)} I(m_{kj} = +1) + (1 - \hat{r}_j^{(t)}) I(m_{kj} = -1)]}.$$

- 3 Normalizes  $\hat{p}_k^{(t+1)}$ ;  $k = 1, \dots, K$  so that they sum to 1.
- 4 Checks for convergence.

For more details, see [23] and [49].

## Posterior Estimation Using Quadratic Programming

Posterior probability estimation using quadratic programming requires an Optimization Toolbox license. To estimate posterior probabilities for an observation using this method, the software:

- 1 Estimates the positive-class posterior probabilities,  $r_j$ , for binary learners  $j = 1, \dots, L$ .

- 2 Using the relationship between  $r_j$  and  $\hat{p}_k$  [45], minimizes

$$\sum_{j=1}^L \left[ -r_j \sum_{k=1}^K \hat{p}_k I(m_{kj} = -1) + (1 - r_j) \sum_{k=1}^K \hat{p}_k I(m_{kj} = +1) \right]^2$$

with respect to  $\hat{p}_k$  and the restrictions

$$0 \leq \hat{p}_k \leq 1$$

$$\sum_k \hat{p}_k = 1.$$

The software performs minimization using quadprog.

## References

- [1] Allwein, E., R. Schapire, and Y. Singer. “Reducing multiclass to binary: A unifying approach for margin classifiers.” *Journal of Machine Learning Research*. Vol. 1, 2000, pp. 113–141.
- [2] Dietterich, T., and G. Bakiri. “Solving Multiclass Learning Problems Via Error-Correcting Output Codes.” *Journal of Artificial Intelligence Research*. Vol. 2, 1995, pp. 263–286.
- [3] Escalera, S., O. Pujol, and P. Radeva. “On the decoding process in ternary error-correcting output codes.” *IEEE Transactions on Pattern Analysis and Machine Intelligence*. Vol. 32, Issue 7, 2010, pp. 120–134.
- [4] Escalera, S., O. Pujol, and P. Radeva. “Separability of ternary codes for sparse designs of error-correcting output codes.” *Pattern Recogn.* Vol. 30, Issue 3, 2009, pp. 285–297.
- [5] Hastie, T., and R. Tibshirani. “Classification by Pairwise Coupling.” *Annals of Statistics*. Vol. 26, Issue 2, 1998, pp. 451–471.
- [6] Wu, T. F., C. J. Lin, and R. Weng. “Probability Estimates for Multi-Class Classification by Pairwise Coupling.” *Journal of Machine Learning Research*. Vol. 5, 2004, pp. 975–1005.

- [7] Zadrozny, B. “Reducing Multiclass to Binary by Coupling Probability Estimates.” *NIPS 2001: Proceedings of Advances in Neural Information Processing Systems 14*, 2001, pp. 1041–1048.

## See Also

`ClassificationECOC` | `ClassificationLinear` |  
`ClassificationPartitionedLinearECOC` | `confusionmat` | `fitcecoc` |  
`perfcurve` | `predict` | `statset` | `testcholdout`

## Topics

“Quick Start Parallel Computing for Statistics and Machine Learning Toolbox” on page 30-2

“Reproducibility in Parallel Statistical Computations” on page 30-14

“Concepts of Parallel Computing in Statistics and Machine Learning Toolbox” on page 30-8

**Introduced in R2016a**

# kfoldPredict

**Class:** ClassificationPartitionedModel

Predict response for observations not used for training

## Syntax

```
label = kfoldPredict(obj)
[label, score] = kfoldPredict(obj)
[label, score, cost] = kfoldPredict(obj)
```

## Description

`label = kfoldPredict(obj)` returns class labels predicted by `obj`, a cross-validated classification. For every fold, `kfoldPredict` predicts class labels for in-fold observations using a model trained on out-of-fold observations.

`[label, score] = kfoldPredict(obj)` returns the predicted classification scores for in-fold observations using a model trained on out-of-fold observations.

`[label, score, cost] = kfoldPredict(obj)` returns misclassification costs.

## Input Arguments

**obj**

Object of class `ClassificationPartitionedModel` or `ClassificationPartitionedEnsemble`.

## Output Arguments

### **label**

Vector of class labels of the same type as the response data used in training `obj`. Each entry of `label` corresponds to a predicted class label for the corresponding row of `X`.

### **score**

Numeric matrix of size N-by-K, where N is the number of observations (rows) in `obj.X`, and K is the number of classes (in `obj.ClassNames`). `score(i,j)` represents the confidence that row `i` of `obj.X` is of class `j`. For details, see “Definitions” on page 32-3361.

### **cost**

Numeric matrix of misclassification costs of size N-by-K. `cost(i,j)` is the average misclassification cost of predicting that row `i` of `obj.X` is of class `j`.

## Examples

### **Estimate Cross-Validation Predictions from an Ensemble**

Find the cross-validation predictions for a model based on Fisher's iris data.

Load Fisher's iris data set.

```
load fisheriris
```

Train an ensemble of classification trees.

```
rng(1); % For reproducibility
Mdl = fitensemble(meas,species,'AdaBoostM2',100,'Tree');
```

Cross validate the trained ensemble using 10-fold cross validation.

```
CVMdl = crossval(Mdl);
```

Estimate cross-validation predicted labels and scores.

```
[elabel,escore] = kfoldPredict(CVMdl);
```

Display the maximum and minimum scores of each class.

```
max(esome)
min(esome)

ans =

    9.3862    8.9871   10.1866

ans =

    0.0017    3.8359    0.8981
```

## Definitions

### Cost (discriminant analysis)

The average misclassification cost is the mean misclassification cost for predictions made by the cross-validated classifiers trained on out-of-fold observations. The matrix of expected costs per observation is defined in “Cost” on page 20-9.

### Score

For discriminant analysis, the score of a classification is the posterior probability of the classification. For the definition of posterior probability in discriminant analysis, see “Posterior Probability” on page 20-8.

For ensembles, a classification score represents the confidence of a classification into a class. The higher the score, the higher the confidence.

Different ensemble algorithms have different definitions for their scores. Furthermore, the range of scores depends on ensemble type. For example:

- AdaBoostM1 scores range from  $-\infty$  to  $\infty$ .
- Bag scores range from 0 to 1.

For trees, the score of a classification of a leaf node is the posterior probability of the classification at that node. The posterior probability of the classification at a node is the

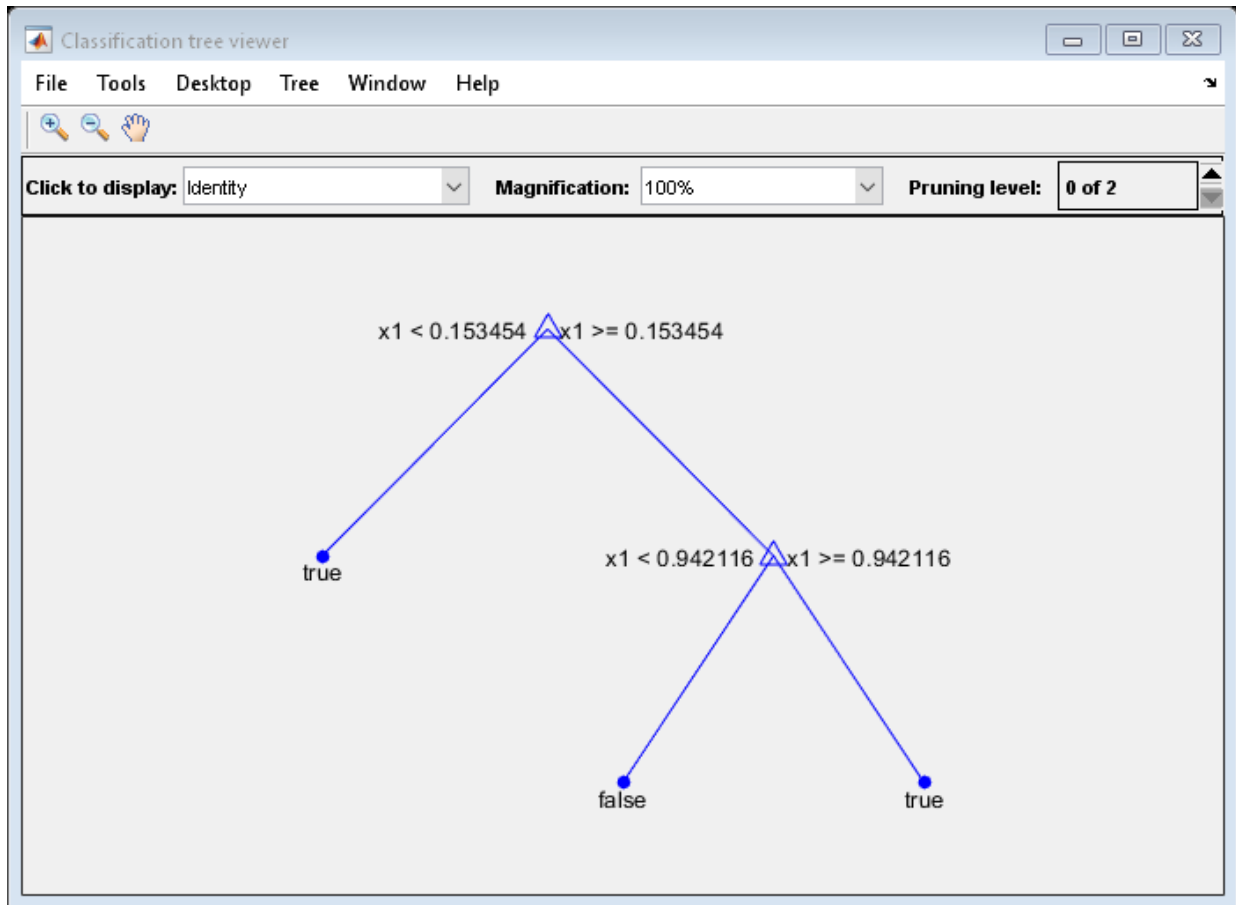
number of training sequences that lead to that node with the classification, divided by the number of training sequences that lead to that node.

For example, consider classifying a predictor `X` as `true` when `X < 0.15` or `X > 0.95`, and `X` is `false` otherwise.

Generate 100 random points and classify them:

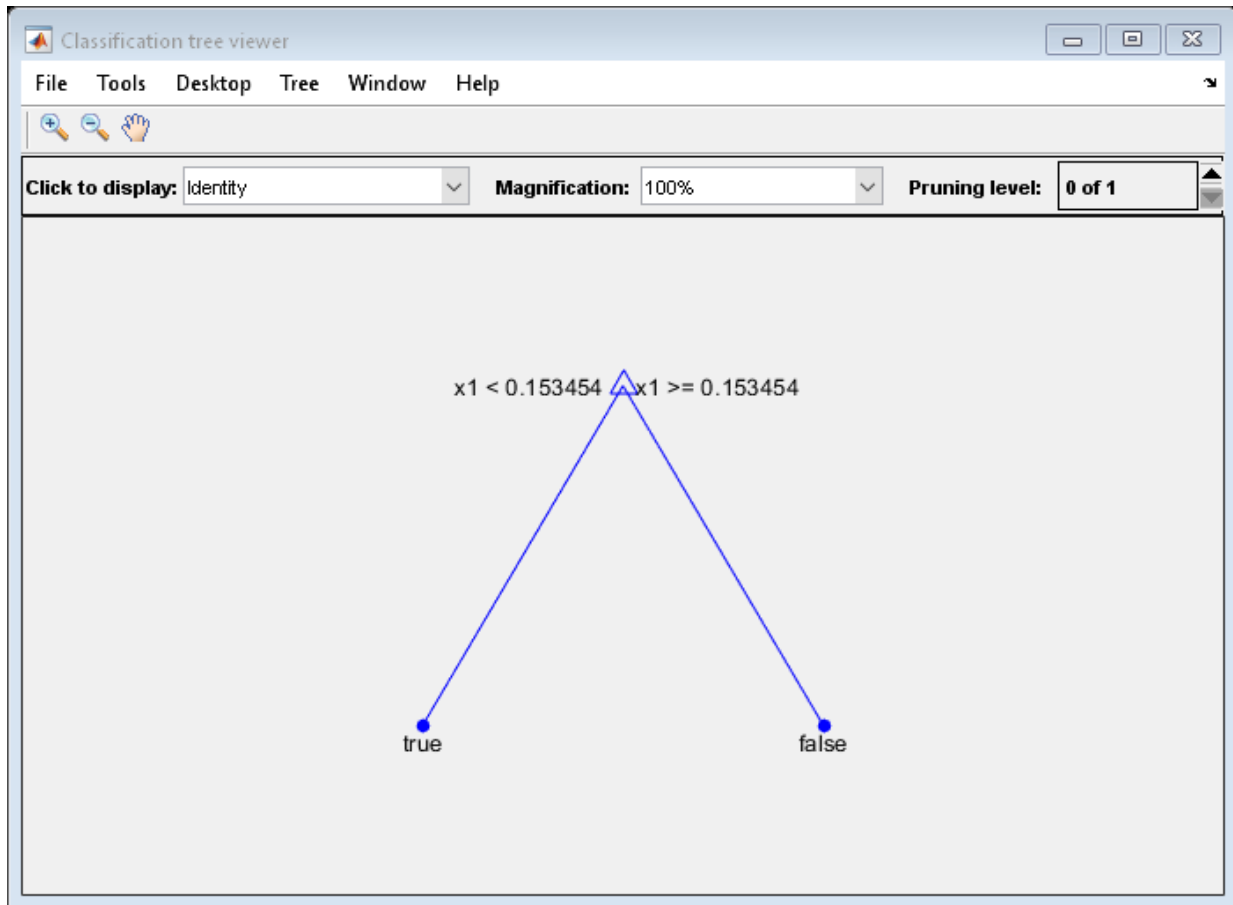
```
rng(0, 'twister') % for reproducibility
X = rand(100,1);
Y = (abs(X - .55) > .4);
tree = fitctree(X,Y);
view(tree, 'Mode', 'Graph')
```





Prune the tree:

```
tree1 = prune(tree, 'Level', 1);  
view(tree1, 'Mode', 'Graph')
```



The pruned tree correctly classifies observations that are less than 0.15 as `true`. It also correctly classifies observations from .15 to .94 as `false`. However, it incorrectly classifies observations that are greater than .94 as `false`. Therefore, the score for observations that are greater than .15 should be about  $.05/.85=.06$  for `true`, and about  $.8/.85=.94$  for `false`.

Compute the prediction scores for the first 10 rows of `X`:

```
[~,score] = predict(tree1,X(1:10));
[score X(1:10,:)]
```

```
ans =  
  
    0.9059    0.0941    0.8147  
    0.9059    0.0941    0.9058  
         0     1.0000    0.1270  
    0.9059    0.0941    0.9134  
    0.9059    0.0941    0.6324  
         0     1.0000    0.0975  
    0.9059    0.0941    0.2785  
    0.9059    0.0941    0.5469  
    0.9059    0.0941    0.9575  
    0.9059    0.0941    0.9649
```

Indeed, every value of  $x$  (the right-most column) that is less than 0.15 has associated scores (the left and center columns) of 0 and 1, while the other values of  $x$  have associated scores of 0.91 and 0.09. The difference (score 0.09 instead of the expected 0.06) is due to a statistical fluctuation: there are 8 observations in  $x$  in the range  $(.95, 1)$  instead of the expected 5 observations.

## See Also

[ClassificationPartitionedModel](#) | [crossval](#) | [kfoldEdge](#) | [kfoldLoss](#) | [kfoldMargin](#) | [kfoldfun](#)

## kfoldPredict

**Class:** RegressionPartitionedLinear

Predict responses for observations not used for training

### Syntax

```
YHat = kfoldPredict(CVMdl)
```

### Description

`YHat = kfoldPredict(CVMdl)` returns cross-validated predicted responses by the cross-validated linear regression model `CVMdl`. That is, for every fold, `kfoldPredict` predicts responses for observations that it holds out when it trains using all other observations.

`YHat` contains predicted responses for each regularization strength in the linear regression models that compose `CVMdl`.

### Input Arguments

**`CVMdl` — Cross-validated, linear regression model**

RegressionPartitionedLinear model object

Cross-validated, linear regression model, specified as a `RegressionPartitionedLinear` model object. You can create a `RegressionPartitionedLinear` model using `fitrlinear` and specifying any of the one of the cross-validation, name-value pair arguments, for example, `CrossVal`.

To obtain estimates, `RegressionPartitionedLinear.kfoldPredict` applies the same data used to cross-validate the linear regression model ( $X$  and  $Y$ ).

## Output Arguments

### **YHat** — Cross-validated predicted responses

numeric array

Cross-validated predicted responses, returned as an  $n$ -by- $L$  numeric array.  $n$  is the number of observations in the predictor data that created `CVMD1` (see `X`) and  $L$  is the number of regularization strengths in `CVMD1.Trained{1}.Lambda`. `YHat(i,j)` is the predicted response for observation  $i$  using the linear regression model that has regularization strength `CVMD1.Trained{1}.Lambda(j)`.

The predicted response using the model with regularization strength  $j$  is  $\hat{y}_j = x\beta_j + b_j$ .

- $x$  is an observation from the predictor data matrix  $X$ , and is row vector.
- $\beta_j$  is the estimated column vector of coefficients. The software stores this vector in `Mdl.Beta(:,j)`.
- $b_j$  is the estimated, scalar bias, which the software stores in `Mdl.Bias(j)`.

## Examples

### Predict Cross-Validated Responses

Simulate 10000 observations from this model

$$y = x_{100} + 2x_{200} + e.$$

- $X = x_1, \dots, x_{1000}$  is a 10000-by-1000 sparse matrix with 10% nonzero standard normal elements.
- $e$  is random normal error with mean 0 and standard deviation 0.3.

```
rng(1) % For reproducibility
n = 1e4;
d = 1e3;
nz = 0.1;
X = sprandn(n,d,nz);
Y = X(:,100) + 2*X(:,200) + 0.3*randn(n,1);
```

Cross-validate a linear regression model.

```
CVMD1 = fitrlinear(X,Y,'CrossVal','on')
Mdl1 = CVMD1.Trained{1}
```

```
CVMD1 =
```

```
classreg.learning.partition.RegressionPartitionedLinear
  CrossValidatedModel: 'Linear'
  ResponseName: 'Y'
  NumObservations: 10000
  KFold: 10
  Partition: [1x1 cvpartition]
  ResponseTransform: 'none'
```

```
Mdl1 =
```

```
RegressionLinear
  ResponseName: 'Y'
  ResponseTransform: 'none'
  Beta: [1000x1 double]
  Bias: 0.0107
  Lambda: 1.1111e-04
  Learner: 'svm'
```

By default, `fitrlinear` implements 10-fold cross-validation. `CVMD1` is a `RegressionPartitionedLinear` model. It contains the property `Trained`, which is a 10-by-1 cell array holding 10 `RegressionLinear` models that the software trained using the training set.

Predict responses for observations that `fitrlinear` did not use in training the folds.

```
yHat = kfoldPredict(CVMD1);
```

Because there is one regularization strength in `Mdl1`, `yHat` is a numeric vector.

## Predict for Models Containing Several Regularization Strengths

Simulate 10000 observations as in “Predict Cross-Validated Responses” on page 32-3367.

```
rng(1) % For reproducibility
n = 1e4;
d = 1e3;
nz = 0.1;
X = sprandn(n,d,nz);
Y = X(:,100) + 2*X(:,200) + 0.3*randn(n,1);
```

Create a set of 15 logarithmically-spaced regularization strengths from  $10^{-5}$  through  $10^{-1}$ .

```
Lambda = logspace(-5,-1,15);
```

Cross-validate the models. To increase execution speed, transpose the predictor data and specify that the observations are in columns. Specify using least squares with a lasso penalty and optimizing the objective function using SpaRSA.

```
X = X';
CVMdl = fitrlinear(X,Y,'ObservationsIn','columns','KFold',5,'Lambda',Lambda,...
    'Learner','leastsquares','Solver','sparsa','Regularization','lasso');
```

CVMdl is a `RegressionPartitionedLinear` model. Its `Trained` property contains a 5-by-1 cell array of trained `RegressionLinear` models, each one holds out a different fold during training. Because `fitrlinear` trained using 15 regularization strengths, you can think of each `RegressionLinear` model as 15 models.

Predict cross-validated responses.

```
YHat = kfoldPredict(CVMdl);
size(YHat)
YHat(2,:)
```

```
ans =
```

```
    10000         15
```

```
ans =
```

```
Columns 1 through 7
```

```
-1.7338 -1.7332 -1.7319 -1.7299 -1.7266 -1.7239 -1.7135
```

```
Columns 8 through 14
```

```
-1.7210 -1.7324 -1.7063 -1.6397 -1.5112 -1.2631 -0.7841
```

```
Column 15
```

```
-0.0096
```

`YHat` is a 10000-by-15 matrix. `YHat(2, :)` is the cross-validated response for observation 2 using the model regularized with all 15 regularization values.

## See Also

`RegressionLinear` | `RegressionPartitionedLinear` | `fitrlinear` | `predict`

**Introduced in R2016a**



# kfoldPredict

**Class:** RegressionPartitionedModel

Predict response for observations not used for training.

## Syntax

```
yfit = kfoldPredict(obj)
```

## Description

`yfit = kfoldPredict(obj)` returns the predicted values for the responses of the training data based on `obj`, an object trained on out-of-fold observations.

## Input Arguments

**obj**

Object of class `RegressionPartitionedModel`. Create `obj` with `fitrtree` or `fitrensemble` along with one of the cross-validation options: `'crossval'`, `'kfold'`, `'holdout'`, `'leaveout'`, or `'cvpartition'`. Alternatively, create `obj` from a regression tree or regression ensemble with `crossval`.

## Output Arguments

**yfit**

A vector of predicted values for the response data based on a model trained on out-of-fold observations.

## Examples

Construct a partitioned regression model, and examine the cross-validation loss. The cross-validation loss is the mean squared error between `yfit` and the true response data:

```
load carsmall
XX = [Cylinders Displacement Horsepower Weight];
YY = MPG;
tree = fitrtree(XX,YY);
cvmodel = crossval(tree);
L = kfoldLoss(cvmodel)
```

```
L =
    26.5271
```

```
yfit = kfoldPredict(cvmodel);
mean( (yfit - tree.Y).^2 )
```

```
ans =
    26.5271
```

## See Also

`fitrtree` | `kfoldLoss`

# kmeans

*k*-means clustering

## Syntax

```
idx = kmeans(X,k)
idx = kmeans(X,k,Name,Value)
[idx,C] = kmeans(____)
[idx,C,sumd] = kmeans(____)
[idx,C,sumd,D] = kmeans(____)
```

## Description

`idx = kmeans(X,k)` performs *k*-means clustering on page 32-3389 to partition the observations of the *n*-by-*p* data matrix *X* into *k* clusters, and returns an *n*-by-1 vector (*idx*) containing cluster indices of each observation. Rows of *X* correspond to points and columns correspond to variables.

By default, `kmeans` uses the squared Euclidean distance measure and the *k*-means++ algorithm on page 32-3390 for cluster center initialization.

`idx = kmeans(X,k,Name,Value)` returns the cluster indices with additional options specified by one or more *Name, Value* pair arguments.

For example, specify the cosine distance, the number of times to repeat the clustering using new initial values, or to use parallel computing.

`[idx,C] = kmeans(____)` returns the *k* cluster centroid locations in the *k*-by-*p* matrix *C*.

`[idx,C,sumd] = kmeans(____)` returns the within-cluster sums of point-to-centroid distances in the *k*-by-1 vector *sumd*.

`[idx,C,sumd,D] = kmeans(____)` returns distances from each point to every centroid in the *n*-by-*k* matrix *D*.

## Examples

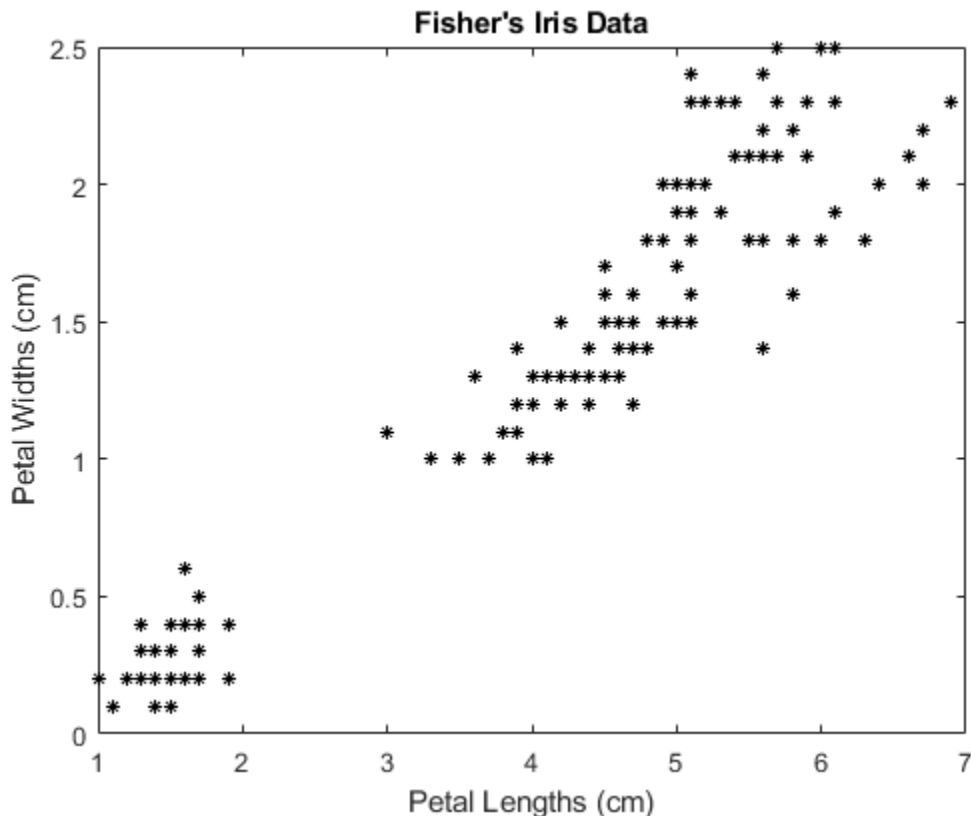
### Train a *k*-Means Clustering Algorithm

Cluster data using *k*-means clustering, then plot the cluster regions.

Load Fisher's iris data set. Use the petal lengths and widths as predictors.

```
load fisheriris
X = meas(:,3:4);

figure;
plot(X(:,1),X(:,2),'k*','MarkerSize',5);
title 'Fisher''s Iris Data';
xlabel 'Petal Lengths (cm)';
ylabel 'Petal Widths (cm)';
```



The larger cluster seems to be split into a lower variance region and a higher variance region. This might indicate that the larger cluster is two, overlapping clusters.

Cluster the data. Specify  $k = 3$  clusters.

```
rng(1); % For reproducibility
[idx,C] = kmeans(X,3);
```

`kmeans` uses the  $k$ -means++ algorithm for centroid initialization and squared Euclidean distance by default. It is good practice to search for lower, local minima by setting the 'Replicates' name-value pair argument.

`idx` is a vector of predicted cluster indices corresponding to the observations in `X`. `C` is a 3-by-2 matrix containing the final centroid locations.

Use `kmeans` to compute the distance from each centroid to points on a grid. To do this, pass the centroids (`C`) and points on a grid to `kmeans`, and implement one iteration of the algorithm.

```
x1 = min(X(:,1)):0.01:max(X(:,1));
x2 = min(X(:,2)):0.01:max(X(:,2));
[x1G,x2G] = meshgrid(x1,x2);
XGrid = [x1G(:),x2G(:)]; % Defines a fine grid on the plot

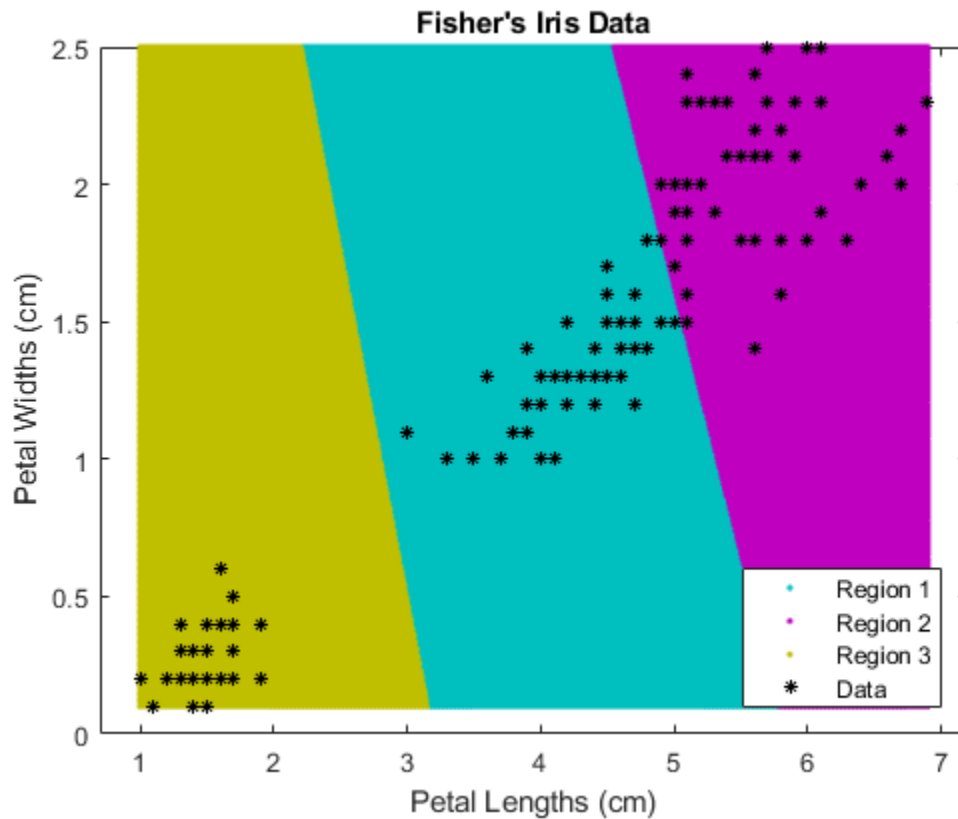
idx2Region = kmeans(XGrid,3,'MaxIter',1,'Start',C);
    % Assigns each node in the grid to the closest centroid
```

```
Warning: Failed to converge in 1 iterations.
```

`kmeans` displays a warning stating that the algorithm did not converge, which you should expect since the software only implemented one iteration.

Plot the cluster regions.

```
figure;
gscatter(XGrid(:,1),XGrid(:,2),idx2Region,...
    [0,0.75,0.75;0.75,0,0.75;0.75,0.75,0],'.');
hold on;
plot(X(:,1),X(:,2),'k*','MarkerSize',5);
title 'Fisher''s Iris Data';
xlabel 'Petal Lengths (cm)';
ylabel 'Petal Widths (cm)';
legend('Region 1','Region 2','Region 3','Data','Location','SouthEast');
hold off;
```

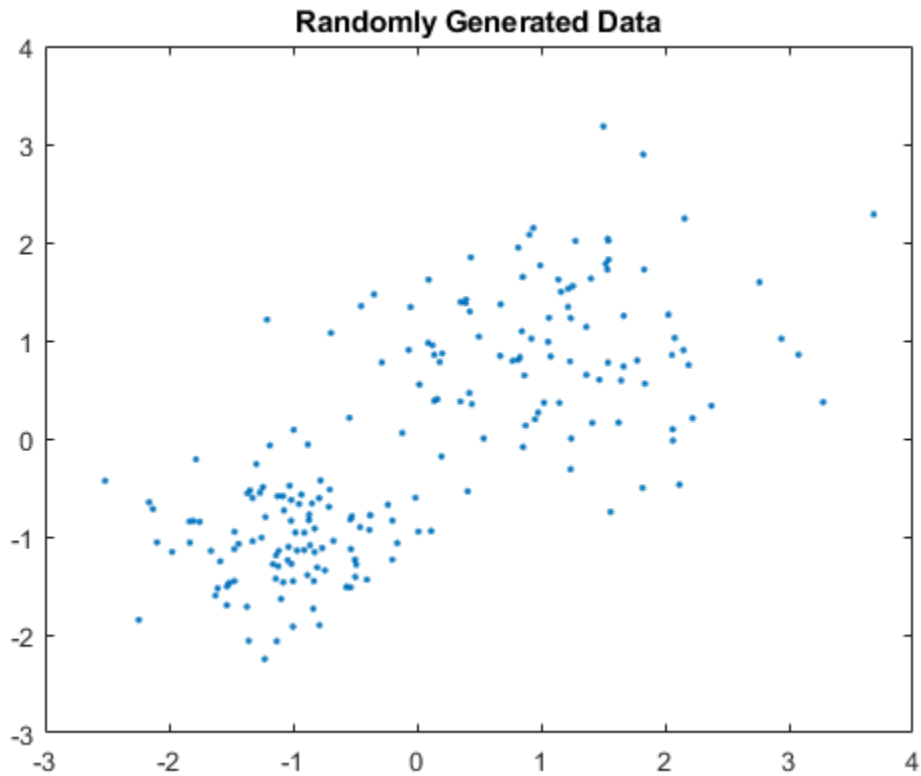


### Partition Data into Two Clusters

Randomly generate the sample data.

```
rng default; % For reproducibility
X = [randn(100,2)*0.75+ones(100,2);
     randn(100,2)*0.5-ones(100,2)];
```

```
figure;
plot(X(:,1),X(:,2),'.');
title 'Randomly Generated Data';
```



There appears to be two clusters in the data.

Partition the data into two clusters, and choose the best arrangement out of five initializations. Display the final output.

```
opts = statset('Display','final');  
[idx,C] = kmeans(X,2,'Distance','cityblock',...  
    'Replicates',5,'Options',opts);
```

```
Replicate 1, 3 iterations, total sum of distances = 201.533.  
Replicate 2, 5 iterations, total sum of distances = 201.533.  
Replicate 3, 3 iterations, total sum of distances = 201.533.  
Replicate 4, 3 iterations, total sum of distances = 201.533.
```



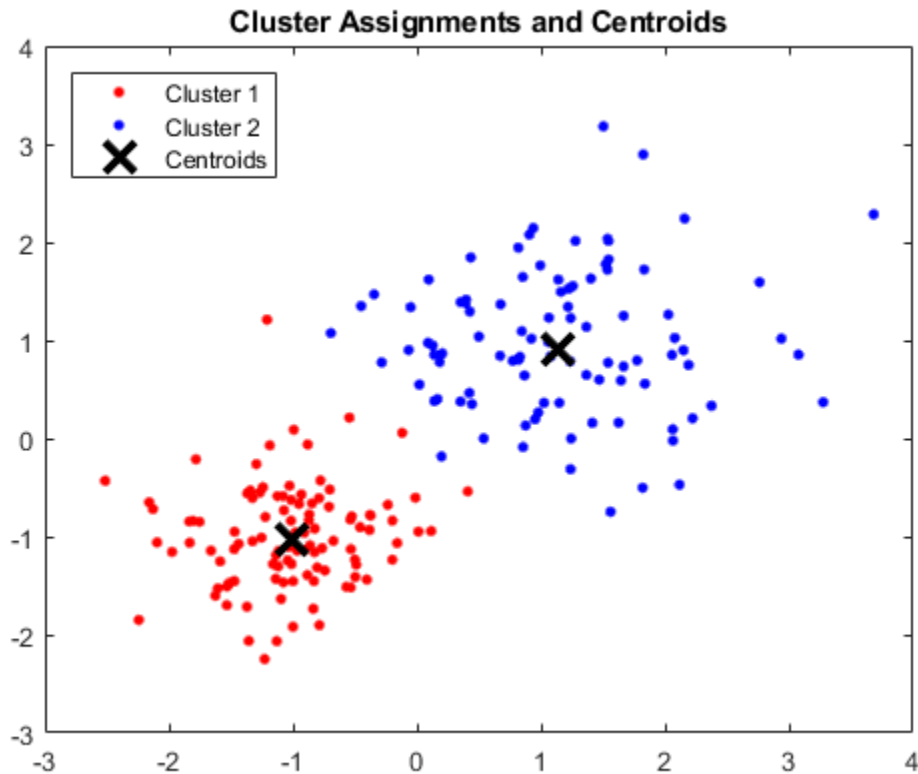
---

Replicate 5, 2 iterations, total sum of distances = 201.533.  
Best total sum of distances = 201.533

By default, the software initializes the replicates separately using *k*-means++.

Plot the clusters and the cluster centroids.

```
figure;
plot(X(idx==1,1),X(idx==1,2), 'r.', 'MarkerSize',12)
hold on
plot(X(idx==2,1),X(idx==2,2), 'b.', 'MarkerSize',12)
plot(C(:,1),C(:,2), 'kx', ...
      'MarkerSize',15, 'LineWidth',3)
legend('Cluster 1', 'Cluster 2', 'Centroids', ...
      'Location', 'NW')
title 'Cluster Assignments and Centroids'
hold off
```



You can determine how well separated the clusters are by passing `idx` to `silhouette`.

### Cluster Data Using Parallel Computing

Clustering large data sets might take time, particularly if you use online updates (set by default). If you have a Parallel Computing Toolbox license and you invoke a pool of workers, then `kmeans` runs each clustering task (or replicate) in parallel. Therefore, if `Replicates > 1`, then the parallel computing decreases time to convergence.

Randomly generate a large data set from a Gaussian mixture model.

```

Mu = bsxfun(@times,ones(20,30),(1:20)); % Gaussian mixture mean
rn30 = randn(30,30);
Sigma = rn30'*rn30; % Symmetric and positive-definite covariance
Mdl = gmdistribution(Mu,Sigma);

rng(1); % For reproducibility
X = random(Mdl,10000);

```

Mdl is a 30-dimensional gmdistribution model with 20 components. X is a 10000-by-30 matrix of data generated from Mdl.

Invoke a parallel pool of workers. Specify options for parallel computing.

```

pool = parpool; % Invokes workers
stream = RandStream('mlfg6331_64'); % Random number stream
options = statset('UseParallel',1,'UseSubstreams',1,...
    'Streams',stream);

```

Starting parallel pool (parpool) using the 'local' profile ... connected to 4 workers.

The input argument 'mlfg6331\_64' of RandStream specifies to use the multiplicative lagged Fibonacci generator algorithm. options is a structure array containing fields that specify options for controlling estimation.

The Command Window indicates that four workers are available. The number of workers might vary on your system.

Cluster the data using  $k$ -means clustering. Specify that there are  $k = 20$  clusters in the data and increase the number of iterations. Typically, the objective function contains local minima. Specify 10 replicates to help find a lower, local minimum.

```

tic; % Start stopwatch timer
[idx,C,sumd,D] = kmeans(X,20,'Options',options,'MaxIter',10000,...
    'Display','final','Replicates',10);
toc % Terminate stopwatch timer

```

```

Replicate 7, 44 iterations, total sum of distances = 7.55218e+06.
Replicate 4, 95 iterations, total sum of distances = 7.53848e+06.
Replicate 2, 104 iterations, total sum of distances = 7.54232e+06.
Replicate 6, 80 iterations, total sum of distances = 7.54237e+06.
Replicate 8, 111 iterations, total sum of distances = 7.54445e+06.
Replicate 1, 52 iterations, total sum of distances = 7.55817e+06.
Replicate 5, 70 iterations, total sum of distances = 7.55278e+06.
Replicate 3, 94 iterations, total sum of distances = 7.54858e+06.

```

```
Replicate 10, 56 iterations, total sum of distances = 7.54547e+06.  
Replicate 9, 83 iterations, total sum of distances = 7.53701e+06.  
Best total sum of distances = 7.53701e+06  
Elapsed time is 3.239232 seconds.
```

The Command Window displays the number of iterations and the terminal objective function value for each replicate. The output arguments contain the results of replicate 9 because it has the lowest total sum of distances.

- “Create Clusters and Determine Separation” on page 16-21
- “Determine the Correct Number of Clusters” on page 16-23
- “Avoid Local Minima” on page 16-26

## Input Arguments

### **x** — Data

numeric matrix

Data, specified as a numeric matrix. The rows of **x** correspond to observations, and the columns correspond to variables.

If **x** is a numeric vector, then `kmeans` treats it as an  $n$ -by-1 data matrix, regardless of its orientation.

Data Types: `single` | `double`

### **k** — Number of clusters

positive integer

Number of clusters in the data, specified as a positive integer.

Data Types: `single` | `double`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'Distance', 'cosine', 'Replicates', 10, 'Options', statset('UseParallel', 1)` specifies the cosine distance, 10 replicate clusters at different starting values, and to use parallel computing.

#### Display — Level of output to display

`'off' (default) | 'final' | 'iter'`

Level of output to display in the Command Window, specified as the comma-separated pair consisting of `'Display'` and one of the following options:

- `'final'` — Displays results of the final iteration
- `'iter'` — Displays results of each iteration
- `'off'` — Displays nothing

Example: `'Display', 'final'`

Data Types: char

#### Distance — Distance measure

`'sqeuclidean' (default) | 'cityblock' | 'cosine' | 'correlation' | 'hamming'`

Distance measure, in  $p$ -dimensional space, used for minimization, specified as the comma-separated pair consisting of `'Distance'` and `'sqeuclidean'`, `'cityblock'`, `'cosine'`, `'correlation'`, or `'hamming'`.

`kmeans` computes centroid clusters differently for the different, supported distance measures. This table summarizes the available distance measures. In the formulae,  $x$  is an observation (that is, a row of  $X$ ) and  $c$  is a centroid (a row vector).

Distance Measure	Description	Formula
<code>'sqeuclidean'</code>	Squared Euclidean distance (default). Each centroid is the mean of the points in that cluster.	$d(x, c) = (x - c)(x - c)'$
<code>'cityblock'</code>	Sum of absolute differences, i.e., the $L1$ distance. Each centroid is the component-wise median of the points in that cluster.	$d(x, c) = \sum_{j=1}^p  x_j - c_j $

Distance Measure	Description	Formula
'cosine'	One minus the cosine of the included angle between points (treated as vectors). Each centroid is the mean of the points in that cluster, after normalizing those points to unit Euclidean length.	$d(x, c) = 1 - \frac{xc'}{\sqrt{(xx')(cc')}}$
'correlation'	One minus the sample correlation between points (treated as sequences of values). Each centroid is the component-wise mean of the points in that cluster, after centering and normalizing those points to zero mean and unit standard deviation.	$d(x, c) = 1 - \frac{(x - \bar{x})(c - \bar{c})'}{\sqrt{(x - \bar{x})(x - \bar{x})'}\sqrt{(c - \bar{c})(c - \bar{c})'}}$ <p>where</p> <ul style="list-style-type: none"> <li>• <math display="block">\bar{x} = \frac{1}{p} \left( \sum_{j=1}^p x_j \right) \bar{1}_p</math></li> <li>• <math display="block">\bar{c} = \frac{1}{p} \left( \sum_{j=1}^p c_j \right) \bar{1}_p</math></li> <li>• <math>\bar{1}_p</math> is a row vector of <math>p</math> ones.</li> </ul>
'hamming'	This measure is only suitable for binary data.  It is the proportion of bits that differ. Each centroid is the component-wise median of points in that cluster.	$d(x, y) = \frac{1}{p} \sum_{j=1}^p I\{x_j \neq y_j\},$ <p>where <math>I</math> is the indicator function.</p>

Example: 'Distance', 'cityblock'

Data Types: char

**EmptyAction** — Action to take if cluster loses all member observations

'singleton' (default) | 'error' | 'drop'

Action to take if a cluster loses all its member observations, specified as the comma-separated pair consisting of 'EmptyAction' and one of the following options.

Value	Description
'error'	Treat an empty cluster as an error.
'drop'	Remove any clusters that become empty. <code>kmeans</code> sets the corresponding return values in C and D to NaN.
'singleton'	Create a new cluster consisting of the one point furthest from its centroid (default).

Example: 'EmptyAction', 'error'

Data Types: char

#### **MaxIter** — Maximum number of iterations

100 (default) | positive integer

Maximum number of iterations, specified as the comma-separated pair consisting of 'MaxIter' and a positive integer.

Example: 'MaxIter', 1000

Data Types: double | single

#### **OnlinePhase** — Online update flag

'off' (default) | 'on'

Online update flag, specified as the comma-separated pair consisting of 'OnlinePhase' and 'off' or 'on'.

If `OnlinePhase` is on, then `kmeans` performs an online update phase in addition to a batch update phase. The online phase can be time consuming for large data sets, but guarantees a solution that is a local minimum of the distance criterion. In other words, the software finds a partition of the data in which moving any single point to a different cluster increases the total sum of distances.

Example: 'OnlinePhase', 'on'

Data Types: char

#### **Options** — Options for controlling iterative algorithm for minimizing fitting criteria

[] (default) | structure array returned by `statset`

Options for controlling the iterative algorithm for minimizing the fitting criteria, specified as the comma-separated pair consisting of 'Options' and a structure array returned by `statset`. These options require Parallel Computing Toolbox.

This table summarizes the available options.

Option	Description
'Streams'	<p>A <code>RandStream</code> object or cell array of such objects. If you do not specify <code>Streams</code>, <code>kmeans</code> uses the default stream or streams. If you specify <code>Streams</code>, use a single object except when:</p> <ul style="list-style-type: none"> <li>• You have an open parallel pool</li> <li>• <code>UseParallel</code> is <code>true</code>.</li> <li>• <code>UseSubstreams</code> is <code>false</code>.</li> </ul> <p>In that case, use a cell array the same size as the parallel pool. If a parallel pool is not open, then <code>Streams</code> must supply a single random number stream.</p>
'UseParallel'	<ul style="list-style-type: none"> <li>• If <code>true</code>, <code>Replicates</code> &gt; 1, and if a parallel pool of workers from the Parallel Computing Toolbox is open, then the software implements <i>k</i>-means on each replicate in parallel.</li> <li>• If the Parallel Computing Toolbox is not installed, or a parallel pool of workers is not open, computation occurs in serial mode. Default is <code>default</code>, meaning serial computation.</li> </ul>
'UseSubstreams'	<p>Set to <code>true</code> to compute in parallel in a reproducible fashion. Default is <code>false</code>. To compute reproducibly, set <code>Streams</code> to a type allowing substreams: <code>'mlfg6331_64'</code> or <code>'mrg32k3a'</code>.</p>

To ensure more predictable results, use `parpool` and explicitly create a parallel pool before invoking `kmeans` and setting 'Options', `statset('UseParallel',1)`.



Example: `'Options', statset('UseParallel',1)`

Data Types: `struct`

### **Replicates** — Number of times to repeat clustering using new initial cluster centroid positions

1 (default) | positive integer

Number of times to repeat clustering using new initial cluster centroid positions, specified as the comma-separated pair consisting of `'Replicates'` and an integer. `kmeans` returns the solution with the lowest `sumd`.

You can set `'Replicates'` implicitly by supplying a 3-D array as the value for the `'Start'` name-value pair argument.

Example: `'Replicates',5`

Data Types: `double` | `single`

### **start** — Method for choosing initial cluster centroid positions

`'plus'` (default) | `'cluster'` | `'sample'` | `'uniform'` | numeric matrix | numeric array

Method for choosing initial cluster centroid positions (or seeds), specified as the comma-separated pair consisting of `'Start'` and `'cluster'`, `'plus'`, `'sample'`, `'uniform'`, a numeric matrix, or a numeric array, . This table summarizes the available options for choosing seeds.

Value	Description
<code>'cluster'</code>	Perform a preliminary clustering phase on a random 10% subsample of <code>X</code> . This preliminary phase is itself initialized using <code>'sample'</code> .
<code>'plus'</code> (default)	Select <code>k</code> seeds by implementing the <code>k</code> -means++ algorithm on page 32-3390 for cluster center initialization.
<code>'sample'</code>	Select <code>k</code> observations from <code>X</code> at random.
<code>'uniform'</code>	Select <code>k</code> points uniformly at random from the range of <code>X</code> . Not valid with the Hamming distance.

Value	Description
numeric matrix	$k$ -by- $p$ matrix of centroid starting locations. The rows of <code>Start</code> correspond to seeds. The software infers $k$ from the first dimension of <code>Start</code> , so you can pass in <code>[]</code> for $k$ .
numeric array	$k$ -by- $p$ - $r$ array of centroid starting locations. The rows of each page correspond to seeds. The third dimension invokes replication of the clustering routine. Page $j$ contains the set of seeds for replicate $j$ . The software infers the number of replicates (specified by the 'Replicates' name-value pair argument) from the size of the third dimension.

Example: 'Start', 'sample'

Data Types: char | double | single

---

**Note** The software treats NaNs as missing data, and removes any row of `X` containing at least one NaN. Removing rows of `X` reduces the sample size.

---

## Output Arguments

### **idx** — Cluster indices

numeric column vector

Cluster indices, returned as a numeric column vector. `idx` has as many rows as `X`, and each row indicates the cluster assignment of the corresponding observation.

### **c** — Cluster centroid locations

numeric matrix

Cluster centroid locations, returned as a numeric matrix. `C` is a  $k$ -by- $p$  matrix, where row  $j$  is the centroid of cluster  $j$ .

### **sumd** — Within-cluster sums of point-to-centroid distances

numeric column vector

Within-cluster sums of point-to-centroid distances, returned as a numeric column vector. `sumd` is a  $k$ -by-1 vector, where element  $j$  is the sum of point-to-centroid distances within cluster  $j$ .

#### **D — Distances from each point to every centroid**

numeric matrix

Distances from each point to every centroid, returned as a numeric matrix. `D` is an  $n$ -by- $k$  matrix, where element  $(j,m)$  is the distance from observation  $j$  to centroid  $m$ .

## Definitions

### ***k*-Means Clustering**

*k*-means clustering, or Lloyd's algorithm [2], is an iterative, data-partitioning algorithm that assigns  $n$  observations to exactly one of  $k$  clusters defined by centroids, where  $k$  is chosen before the algorithm starts.

The algorithm proceeds as follows:

- 1 Choose  $k$  initial cluster centers (centroid). For example, choose  $k$  observations at random (by using 'Start', 'sample') or use the *k*-means ++ algorithm on page 32-3390 for cluster center initialization (the default).
- 2 Compute point-to-cluster-centroid distances of all observations to each centroid.
- 3 There are two ways to proceed (specified by `OnlinePhase`):
  - Batch update — Assign each observation to the cluster with the closest centroid.
  - Online update — Individually assign observations to a different centroid if the reassignment decreases the sum of the within-cluster, sum-of-squares point-to-cluster-centroid distances.

For more details, see “Algorithms” on page 32-3391.

- 4 Compute the average of the observations in each cluster to obtain  $k$  new centroid locations.
- 5 Repeat steps 2 through 4 until cluster assignments do not change, or the maximum number of iterations is reached.

## ***k*-means++ Algorithm**

The *k*-means++ algorithm uses an heuristic to find centroid seeds for *k*-means clustering. According to Arthur and Vassilvitskii [1], *k*-means++ improves the running time of Lloyd's algorithm, and the quality of the final solution.

The *k*-means++ algorithm chooses seeds as follows, assuming the number of clusters is *k*.

- 1 Select an observation uniformly at random from the data set, *X*. The chosen observation is the first centroid, and is denoted  $c_1$ .
- 2 Compute distances from each observation to  $c_1$ . Denote the distance between  $c_j$  and

the observation *m* as  $d(x_m, c_j)$ .

- 3 Select the next centroid,  $c_2$  at random from *X* with probability

$$\frac{d^2(x_m, c_1)}{\sum_{j=1}^n d^2(x_j, c_1)}.$$

- 4 To choose center *j*:

- a Compute the distances from each observation to each centroid, and assign each observation to its closest centroid.

- b For  $m = 1, \dots, n$  and  $p = 1, \dots, j - 1$ , select centroid *j* at random from *X* with probability

$$\frac{d^2(x_m, c_p)}{\sum_{\{h; x_h \in C_p\}} d^2(x_h, c_p)},$$

where  $C_p$  is the set of all observations closest to centroid  $c_p$  and  $x_m$  belongs to  $C_p$ .

That is, select each subsequent center with a probability proportional to the distance from itself to the closest center that you already chose.

- 5 Repeat step 4 until *k* centroids are chosen.

Arthur and Vassilvitskii [1] demonstrate, using a simulation study for several cluster orientations, that *k*-means++ achieves faster convergence to a lower sum of within-cluster, sum-of-squares point-to-cluster-centroid distances than Lloyd's algorithm.

## Algorithms

- `kmeans` uses a two-phase iterative algorithm to minimize the sum of point-to-centroid distances, summed over all  $k$  clusters.
  - 1 This first phase uses batch updates, where each iteration consists of reassigning points to their nearest cluster centroid, all at once, followed by recalculation of cluster centroids. This phase occasionally does not converge to solution that is a local minimum. That is, a partition of the data where moving any single point to a different cluster increases the total sum of distances. This is more likely for small data sets. The batch phase is fast, but potentially only approximates a solution as a starting point for the second phase.
  - 2 This second phase uses online updates, where points are individually reassigned if doing so reduces the sum of distances, and cluster centroids are recomputed after each reassignment. Each iteration during this phase consists of one pass through all the points. This phase converges to a local minimum, although there might be other local minima with lower total sum of distances. In general, finding the global minimum is solved by an exhaustive choice of starting points, but using several replicates with random starting points typically results in a solution that is a global minimum.
- If `Replicates = r > 1` and `Start` is `plus` (the default), then the software selects  $r$  possibly different sets of seeds according to the  $k$ -means++ algorithm on page 32-3390.
- If you enable the `UseParallel` option in `Options` and `Replicates > 1`, then each worker selects seeds and clusters in parallel.

## References

- [1] Arthur, David, and Sergi Vassilvitskii. “K-means++: The Advantages of Careful Seeding.” *SODA '07: Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms*. 2007, pp. 1027–1035.
- [2] Lloyd, Stuart P. “Least Squares Quantization in PCM.” *IEEE Transactions on Information Theory*. Vol. 28, 1982, pp. 129–137.
- [3] Seber, G. A. F. *Multivariate Observations*. Hoboken, NJ: John Wiley & Sons, Inc., 1984.

- [4] Spath, H. *Cluster Dissection and Analysis: Theory, FORTRAN Programs, Examples*. Translated by J. Goldschmidt. New York: Halsted Press, 1985.

## Extended Capabilities

### Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function supports tall arrays for out-of-memory data with some limitations.

Only random sample initialization is supported. Supported syntaxes:

- `idx = kmeans(X, k)` performs classic k-means clustering.
- `[idx, C] = kmeans(X, k)` also returns the k cluster centroid locations.
- `[idx, C, sumd] = kmeans(X, k)` additionally returns the k within-cluster sums of point-to-centroid distances.
- `[___] = kmeans(___, Name, Value)` specifies additional name-value pair options using any of the other syntaxes. Valid options are:
  - 'Start' — Method used to choose the initial cluster centroid positions. Value can be:
    - 'plus' (default) — Select k observations from X using a variant of the kmeans ++ algorithm adapted for tall data.
    - 'sample' — Select k observations from X at random.
    - Numeric matrix — A k-by-p matrix to explicitly specify starting locations.
  - 'Options' — An options structure created using the `statset` function. For tall arrays, `kmeans` uses the fields listed here and ignores all other fields in the options structure:
    - 'Display' — Level of display. Choices are 'iter' (default), 'off', and 'final'.
    - 'MaxIter' — Maximum number of iterations. Default is 100.

- `'TolFun'` — Convergency tolerance for the within-cluster sums of point-to-centroid distances. Default is  $1e-4$ . This option field only works with tall arrays.

For more information, see “Tall Arrays” (MATLAB).

## C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- If the `Start` method uses random selections, the initial centroid cluster positions might not match MATLAB.
- If the number of rows in `X` is fixed, code generation does not remove rows of `X` that contain a NaN.
- The cluster centroid locations in `C` can have a different order than in MATLAB. In this case, the cluster indices in `idx` have corresponding differences.
- If you provide `Display`, its value must be `'off'`.
- If you provide `Streams`, it must be empty and `UseSubstreams` must be `false`.
- When you set the `UseParallel` option to `true`:
  - Some computations can execute in parallel even when `Replicates` is 1. For large data sets, when `Replicates` is 1, consider setting the `UseParallel` option to `true`.
  - `kmeans` uses `parfor` to create loops that run in parallel on supported shared-memory multicore platforms. Loops that run in parallel can be faster than loops that run on a single thread. If your compiler does not support the Open Multiprocessing (OpenMP) application interface or you disable OpenMP library, MATLAB Coder treats the `parfor`-loops as `for`-loops. To find supported compilers, see [http://www.mathworks.com/support/compilers/current\\_release/](http://www.mathworks.com/support/compilers/current_release/).

## See Also

`clusterdata` | `gmdistribution` | `linkage` | `parpool` | `silhouette` | `statset`

## **Topics**

“Create Clusters and Determine Separation” on page 16-21

“Determine the Correct Number of Clusters” on page 16-23

“Avoid Local Minima” on page 16-26

“Introduction to k-Means Clustering” on page 16-21

**Introduced before R2006a**



# kmedoids

*k*-medoids clustering

## Syntax

```
idx = kmedoids(X,k)
idx = kmedoids(X,k,Name,Value)
[idx,C] = kmedoids(____)
[idx,C,sumd] = kmedoids(____)
[idx,C,sumd,D] = kmedoids(____)
[idx,C,sumd,D,midx] = kmedoids(____)
[idx,C,sumd,D,midx,info] = kmedoids(____)
```

## Description

`idx = kmedoids(X,k)` performs “k-medoids Clustering” on page 32-3411 to partition the observations of the  $n$ -by- $p$  matrix  $X$  into  $k$  clusters, and returns an  $n$ -by-1 vector `idx` containing cluster indices of each observation. Rows of  $X$  correspond to points and columns correspond to variables. By default, `kmedoids` uses squared Euclidean distance measure and the  $k$ -means++ algorithm on page 32-3390 for choosing initial cluster medoid positions.

`idx = kmedoids(X,k,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

`[idx,C] = kmedoids(____)` returns the  $k$  cluster medoid locations in the  $k$ -by- $p$  matrix  $C$ .

`[idx,C,sumd] = kmedoids(____)` returns the within-cluster sums of point-to-medoid distances in the  $k$ -by-1 vector `sumd`.

`[idx,C,sumd,D] = kmedoids(____)` returns distances from each point to every medoid in the  $n$ -by- $k$  matrix  $D$ .

`[idx,C,sumd,D,midx] = kmedoids(____)` returns the indices `midx` such that  $C = X(\text{midx},:)$ . `midx` is a  $k$ -by-1 vector.

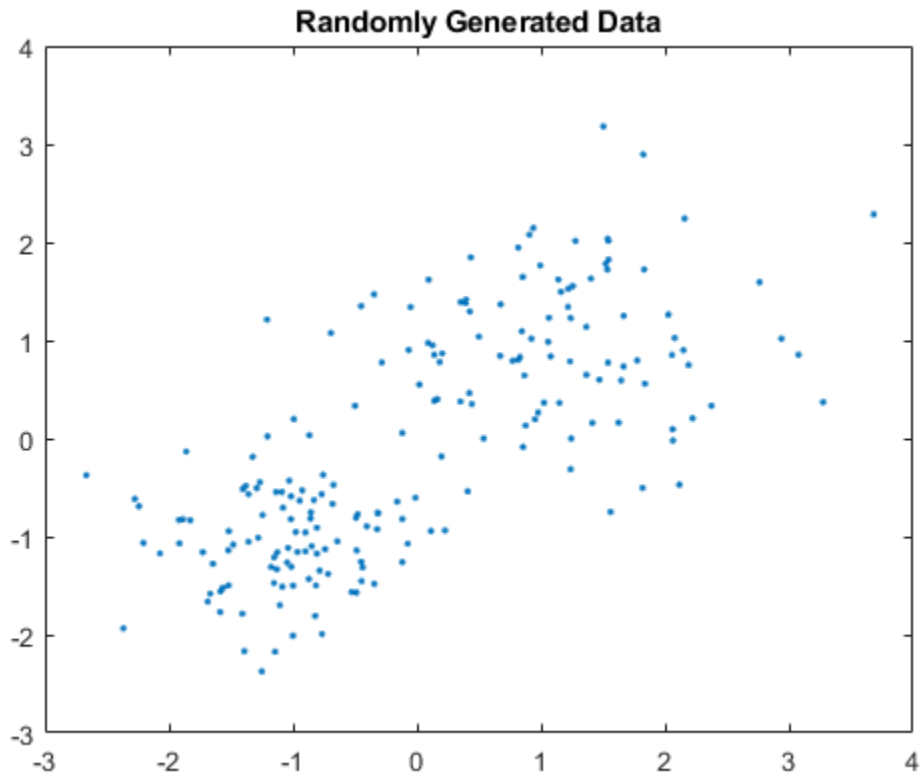
`[idx,C,sumd,D,midx,info] = kmedoids(____)` returns a structure `info` with information about the options used by the algorithm when executed.

## Examples

### Group Data into Two Clusters

Randomly generate data.

```
rng('default'); % For reproducibility
X = [randn(100,2)*0.75+ones(100,2);
     randn(100,2)*0.55-ones(100,2)];
figure;
plot(X(:,1),X(:,2),'.');
title('Randomly Generated Data');
```



Group data into two clusters using `kmedoids`. Use the `cityblock` distance measure.

```
opts = statset('Display','iter');
[idx,C,sumd,d,midx,info] = kmedoids(X,2,'Distance','cityblock','Options',opts);
```

```
    rep      iter      sum
     1         1    209.856
     1         2    209.856
```

```
Best total sum of distances = 209.856
```

`info` is a struct that contains information about how the algorithm was executed. For example, `bestReplicate` field indicates the replicate that was used to produce the final solution. In this example, the replicate number 1 was used since the default number of replicates is 1 for the default algorithm, which is `pam` in this case.

```
info

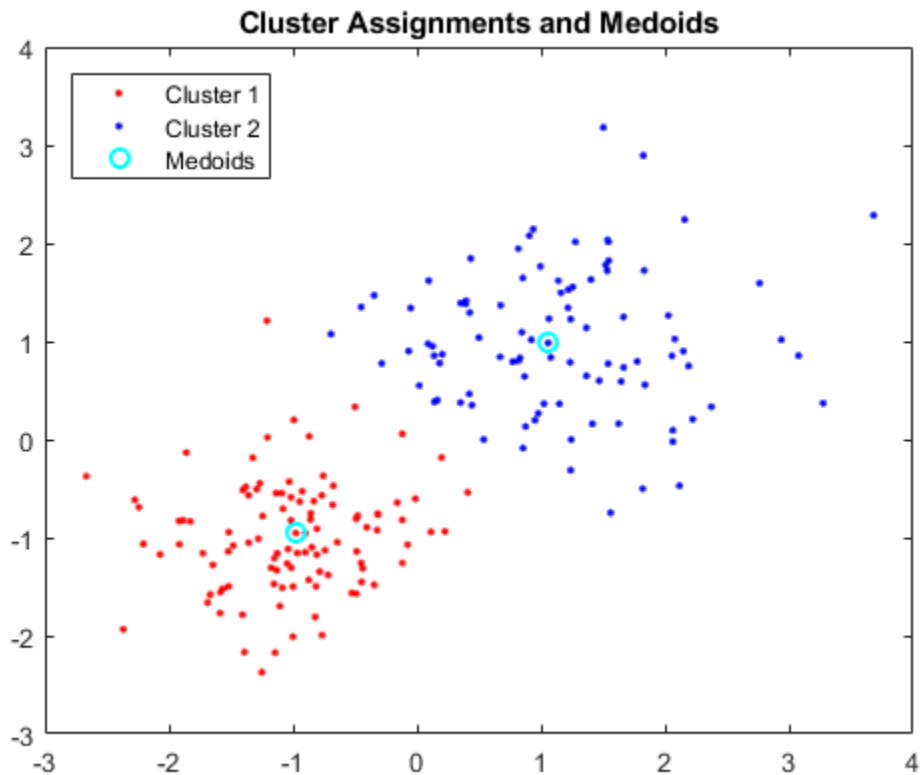
info =

  struct with fields:

    algorithm: 'pam'
    start: 'plus'
    distance: 'cityblock'
    iterations: 2
    bestReplicate: 1
```

Plot the clusters and the cluster medoids.

```
figure;
plot(X(idx==1,1),X(idx==1,2),'r.','MarkerSize',7)
hold on
plot(X(idx==2,1),X(idx==2,2),'b.','MarkerSize',7)
plot(C(:,1),C(:,2),'co',...
     'MarkerSize',7,'LineWidth',1.5)
legend('Cluster 1','Cluster 2','Medoids',...
      'Location','NW');
title('Cluster Assignments and Medoids');
hold off
```



### Cluster Categorical Data Using k-Medoids

This example uses "Mushroom" data set [3][4][5] [6][7] from the UCI machine learning archive [7], described in <http://archive.ics.uci.edu/ml/datasets/Mushroom>. The data set includes 22 predictors for 8,124 observations of various mushrooms. The predictors are categorical data types. For example, cap shape is categorized with features of 'b' for bell-shaped cap and 'c' for conical. Mushroom color is also categorized with features of 'n' for brown, and 'p' for pink. The data set also includes a classification for each mushroom of either edible or poisonous.

Since the features of the mushroom data set are categorical, it is not possible to define the mean of several data points, and therefore the widely-used  $k$ -means clustering algorithm cannot be meaningfully applied to this data set.  $k$ -medoids is a related algorithm that partitions data into  $k$  distinct clusters, by finding medoids that minimize the sum of dissimilarities between points in the data and their nearest medoid.

The medoid of a set is a member of that set whose average dissimilarity with the other members of the set is the smallest. Similarity can be defined for many types of data that do not allow a mean to be calculated, allowing  $k$ -medoids to be used for a broader range of problems than  $k$ -means.

Using  $k$ -medoids, this example clusters the mushrooms into two groups, based on the predictors provided. It then explores the relationship between those clusters and the classifications of the mushrooms as either edible or poisonous.

This example assumes that you have downloaded the "Mushroom" data set [3][4][5] [6][7] from the UCI database (<http://archive.ics.uci.edu/ml/machine-learning-databases/mushroom/>) and saved it in your current directory as a text file named `agaricus-lepiota.txt`. There is no column headers in the data, so `readtable` uses the default variable names.

```
clear all
data = readtable('agaricus-lepiota.txt', 'ReadVariableNames', false);
```

Display the first 5 mushrooms with their first few features.

```
data(1:5, 1:10)
```

```
ans =
```

Var1	Var2	Var3	Var4	Var5	Var6	Var7	Var8	Var9	Var10
'p'	'x'	's'	'n'	't'	'p'	'f'	'c'	'n'	'k'
'e'	'x'	's'	'y'	't'	'a'	'f'	'c'	'b'	'k'
'e'	'b'	's'	'w'	't'	'l'	'f'	'c'	'b'	'n'
'p'	'x'	'y'	'w'	't'	'p'	'f'	'c'	'n'	'n'
'e'	'x'	's'	'g'	'f'	'n'	'f'	'w'	'b'	'k'

Extract the first column, labeled data for edible and poisonous groups. Then delete the column.

```
labels = data(:,1);
labels = categorical(labels{:,:});
data(:,1) = [];
```

Store the names of predictors (features), which are described in <http://archive.ics.uci.edu/ml/machine-learning-databases/mushroom/agaricus-lepiota.names>.

```
VarNames = {'cap_shape' 'cap_surface' 'cap_color' 'bruises' 'odor' ...
            'gill_attachment' 'gill_spacing' 'gill_size' 'gill_color' ...
            'stalk_shape' 'stalk_root' 'stalk_surface_above_ring' ...
            'stalk_surface_below_ring' 'stalk_color_above_ring' ...
            'stalk_color_below_ring' 'veil_type' 'veil_color' 'ring_number' ....
            'ring_type' 'spore_print_color' 'population' 'habitat'};
```

Set the variable names.

```
data.Properties.VariableNames = VarNames;
```

There are a total of 2480 missing values denoted as '?'.  
 sum(char(data{:,:}) == '?')

```
ans =
```

```
2480
```

Based on the inspection of the data set and its description, the missing values belong only to the 11th variable (`stalk_root`). Remove the column from the table.

```
data(:,11) = [];
```

`kmedoids` only accepts numeric data. You need to cast the categories you have into numeric type. The distance function you will use to define the dissimilarity of the data will be based on the double representation of the categorical data.

```
cats = categorical(data{:,:});
data = double(cats);
```

`kmedoids` can use any distance metric supported by `pdist2` to cluster. For this example you will cluster the data using the Hamming distance because this is an appropriate distance metric for categorical data as illustrated below. The Hamming distance between two vectors is the percentage of the vector components that differ. For instance, consider these two vectors.

```
v1 = [1 0 2 1];
```

```
v2 = [1 1 2 1];
```

They are equal in the 1st, 3rd and 4th coordinate. Since 1 of the 4 coordinates differ, the Hamming distance between these two vectors is .25.

You can use the function `pdist2` to measure the Hamming distance between the first and second row of data, the numerical representation of the categorical mushroom data. The value `.2857` means that 6 of the 21 features of the mushroom differ.

```
pdist2(data(1,:),data(2:,:), 'hamming')
```

```
ans =
```

```
0.2857
```

In this example, you're clustering the mushroom data into two clusters based on features to see if the clustering corresponds to edibility. The `kmedoids` function is guaranteed to converge to a local minima of the clustering criterion; however, this may not be a global minimum for the problem. It is a good idea to cluster the problem a few times using the 'replicates' parameter. When 'replicates' is set to a value,  $n$ , greater than 1, the `k-medoids` algorithm is run  $n$  times, and the best result is returned.

To run `kmedoids` to cluster data into 2 clusters, based on the Hamming distance and to return the best result of 3 replicates, you run the following.

```
rng('default'); % For reproducibility  
[IDX, C, SUMD, D, MIDX, INFO] = kmedoids(data,2, 'distance', 'hamming', 'replicates',3);
```

Let's assume that mushrooms in the predicted group 1 are poisonous and group 2 are all edible. To determine the performance of clustering results, calculate how many mushrooms in group 1 are indeed poisonous and group 2 are edible based on the known labels. In other words, calculate the number of false positives, false negatives, as well as true positives and true negatives.

Construct a confusion matrix (or matching matrix), where the diagonal elements represent the number of true positives and true negatives, respectively. The off-diagonal elements represent false negatives and false positives, respectively. For convenience, use the `confusionmat` function, which calculates a confusion matrix given known labels and predicted labels. Get the predicted label information from the `IDX` variable. `IDX` contains values of 1 and 2 for each data point, representing poisonous and edible groups, respectively.



```

predLabels = labels; % Initialize a vector for predicted labels.
predLabels(IDX==1) = categorical({'p'}); % Assign group 1 to be poisonous.
predLabels(IDX==2) = categorical({'e'}); % Assign group 2 to be edible.
confMatrix = confusionmat(labels,predLabels)

confMatrix =

    4176     32
    816    3100

```

Out of 4208 edible mushrooms, 4176 were correctly predicted to be in group 2 (edible group), and 32 were incorrectly predicted to be in group 1 (poisonous group). Similarly, out of 3916 poisonous mushrooms, 3100 were correctly predicted to be in group 1 (poisonous group), and 816 were incorrectly predicted to be in group 2 (edible group).

Given this confusion matrix, calculate the accuracy, which is the proportion of true results (both true positives and true negatives) against the overall data, and precision, which is the proportion of the true positives against all the positive results (true positives and false positives).

```

accuracy = (confMatrix(1,1)+confMatrix(2,2))/(sum(sum(confMatrix)))

accuracy =

    0.8956

precision = confMatrix(1,1) / (confMatrix(1,1)+confMatrix(2,1))

precision =

    0.8365

```

The results indicated that applying the k-medoids algorithm to the categorical features of mushrooms resulted in clusters that were associated with edibility.

## Input Arguments

### **x** — Data

numeric matrix

Data, specified as a numeric matrix. The rows of X correspond to observations, and the columns correspond to variables.

**$\kappa$  — Number of medoids**

positive integer

Number of medoids in the data, specified as a positive integer.

**Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'Distance', 'euclidean', 'Replicates', 3, 'Options', statset('UseParallel', 1)` specifies Euclidean distance, three replicate medoids at different starting values, and to use parallel computing.

**Algorithm — Algorithm to find medoids**`'pam' | 'small' | 'clara' | 'large'`

Algorithm to find medoids, specified as the comma-separated pair consisting of `'Algorithm'` and `'pam'`, `'small'`, `'clara'`, or `'large'`. The default algorithm depends on the number of rows of `X`.

- If the number of rows of `X` is less than 3000, `'pam'` is the default algorithm.
- If the number of rows is between 3000 and 10000, `'small'` is the default algorithm.
- For all other cases, `'large'` is the default algorithm.

You can override the default choice by explicitly stating the algorithm. This table summarizes the available algorithms.

Algorithm	Description
'pam'	<p>Partitioning Around Medoids (PAM) is the classical algorithm for solving the <math>k</math>-medoids problem described in [1]. After applying the initialization function to select initial medoid positions, the program performs the swap-step of the PAM algorithm, that is, it searches over all possible swaps between medoids and non-medoids to see if the sum of medoid to cluster member distances goes down. You can specify which initialization function to use via the 'Start' name-value pair argument.</p> <p>The algorithm proceeds as follows.</p> <ol style="list-style-type: none"><li data-bbox="497 644 1337 743">1 Build-step: Each of <math>k</math> clusters is associated with a potential medoid. This assignment is performed using a technique specified by the 'Start' name-value pair argument.</li><li data-bbox="497 753 1337 916">2 Swap-step: Within each cluster, each point is tested as a potential medoid by checking if the sum of within-cluster distances gets smaller using that point as the medoid. If so, the point is defined as a new medoid. Every point is then assigned to the cluster with the closest medoid.</li></ol> <p>The algorithm iterates the build- and swap-steps until the medoids do not change, or other termination criteria are met.</p> <p>The algorithm can produce better solutions than the other algorithms in some situations, but it can be prohibitively long running.</p>

Algorithm	Description
'small'	<p>Use an algorithm similar to the k-means algorithm to find <math>k</math> medoids. This option employs a variant of the Lloyd's iterations based on [2].</p> <p>The algorithm proceeds as follows.</p> <ol style="list-style-type: none"> <li>1 For each point in each cluster, calculate the sum of distances from the point to every other point in the cluster. Choose the point that minimizes the sum as the new medoid.</li> <li>2 Update the cluster membership for each data point to reflect the new medoid.</li> </ol> <p>The algorithm repeats these steps until no further updates occur or other termination criteria are met. The algorithm has an optional PAM-like online update phase (specified by the 'OnlinePhase' name-value pair argument) that improves cluster quality. It tends to return higher quality solutions than the <code>clara</code> or <code>large</code> algorithms, but it may not be the best choice for very large data.</p>
'clara'	<p>Clustering LARge Applications (CLARA) [1] repeatedly performs the PAM algorithm on random subsets of the data. It aims to overcome scaling challenges posed by the PAM algorithm through sampling.</p> <p>The algorithm proceeds as follows.</p> <ol style="list-style-type: none"> <li>1 Select a subset of the data and apply the PAM algorithm to the subset.</li> <li>2 Assign points of the full data set to clusters by picking the closest medoid.</li> </ol> <p>The algorithm repeats these steps until the medoids do not change, or other termination criteria are met.</p> <p>For the best performance, it is recommended that you perform multiple replicates. By default, the program performs five replicates. Each replicate samples <math>s</math> rows from <math>X</math> (specified by 'NumSamples' name-value pair argument) to perform clustering on. By default, <math>40+2*k</math> samples are selected.</p>

Algorithm	Description
'large'	This is similar to the <code>small</code> scale algorithm and repeatedly performs searches using a k-means like update. However, the algorithm examines only a random sample of cluster members during each iteration. The user-adjustable parameter, <code>'PercentNeighbors'</code> , controls the number of neighbors to examine. If there is no improvement after the neighbors are examined, the algorithm terminates the local search. The algorithm performs a total of <code>r</code> replicates (specified by <code>'Replicates'</code> name-value pair argument) and returns the best clustering result. The algorithm has an optional PAM-like online phase (specified by the <code>'OnlinePhase'</code> name-value pair argument) that improves cluster quality.

Example: `'Algorithm', 'pam'`

#### **OnlinePhase — Flag to perform PAM-like online update phase**

`'on'` (default) | `'off'`

A flag to perform PAM-like online update phase, specified as a comma-separated pair consisting of `'OnlinePhase'` and `'on'` or `'off'`.

If it is on, then `kmedoids` performs a PAM-like update to the medoids after the Lloyd iterations in the `small` and `large` algorithms. During this online update phase, the algorithm chooses a small subset of data points in each cluster that are the furthest from and nearest to medoid. For each chosen point, it reassigns the clustering of the entire data set and check if this creates a smaller sum of distances than the best known.

In other words, the swap considerations are limited to the points near the medoids and far from the medoids. The near points are considered in order to refine the clustering. The far points are considered in order to escape local minima. Turning on this feature tends to improve the quality of solutions generated by both algorithms. Total run time tends to increase as well, but the increase typically is less than one iteration of PAM.

Example: `OnlinePhase, 'off'`

#### **Distance — Distance measure**

`'sqeuclidean'` (default) | `'euclidean'` | `'seuclidean'` | `'cityblock'` | `'minkowski'` | `'chebychev'` | `'mahalanobis'` | `'cosine'` | `'correlation'` | `'spearman'` | `'hamming'` | `'jaccard'` | custom distance function

Distance measure, in p-dimensional space, specified as the comma-separate pair consisting of 'Distance' and 'euclidean', 'seuclidean', 'cityblock', 'minkowski', 'chebychev', 'mahalanobis', 'cosine', 'correlation', 'spearman', 'hamming', 'jaccard', or a custom distance function.

`kmedoids` minimizes the sum of medoid to cluster member distances. See `pdist` for the definition of each distance measure. `kmedoids` supports all distance measures supported by `pdist`.

Example: 'Distance', 'hamming'

### Options — Options to control iterative algorithm to minimize fitting criteria

[] (default) | structure array returned by `statset`

Options to control the iterative algorithm to minimize fitting criteria, specified as the comma-separated pair consisting of 'Options' and a structure array returned by `statset`. This table summarizes these options.

Option	Description
Display	Level of display output. Choices are 'off' (default) and 'iter'.
MaxIter	Maximum number of iterations allowed. The default is 100.
UseParallel	If <code>true</code> and a <code>parpool</code> is open, compute in parallel. If Parallel Computing Toolbox is not available, or a <code>parpool</code> is not open, computation occurs in serial mode. The default is <code>false</code> , meaning serial computation.
UseSubstreams	Set to <code>true</code> to compute in parallel in a reproducible fashion. The default is <code>false</code> . To compute reproducibly, you must also set <code>Streams</code> to a type allowing substreams: 'mlfg6331_64' or 'mrg32k3a'.
Streams	A <code>RandStream</code> object or cell array of such objects. For details about these options and parallel computing in Statistics and Machine Learning Toolbox, see “Speed Up Statistical Computations” or enter <code>help parallelstats</code> at the command line.

Example: 'Options', `statset('Display','off')`

**Replicates** — Number of times to repeat clustering using new initial cluster medoid positions

positive integer

Number of times to repeat clustering using new initial cluster medoid positions, specified as a positive integer. The default value depends on the choice of algorithm. For `pam` and `small`, the default is 1. For `clara`, the default is 5. For `large`, the default is 3.

Example: `'Replicates', 4`

**NumSamples** — Number of samples to take from data when executing clara algorithm

40+2\*k (default) | positive integer

Number of samples to take from the data when executing the `clara` algorithm, specified as a positive integer. The default number of samples is calculated as 40+2\*k.

Example: `'NumSamples', 160`

**PercentNeighbors** — Percent of data set to examine using large algorithm

0.001 (default) | scalar value between 0 and 1

Percent of the data set to examine using the `large` algorithm, specified as a positive number.

The program examines `percentneighbors*size(X,1)` number of neighbors for the medoids. If there is no improvement in the within-cluster sum of distances, then the algorithm terminates.

The value of this parameter between 0 and 1, where a value closer to 1 tends to give higher quality solutions, but the algorithm takes longer to run, and a value closer to 0 tends to give lower quality solutions, but finishes faster.

Example: `'PercentNeighbors', 0.01`

**start** — Method for choosing initial cluster medoid positions

'plus' (default) | 'sample' | 'cluster' | matrix

Method for choosing initial cluster medoid positions, specified as the comma-separated pair consisting of 'start' and 'plus', 'sample', 'cluster', or a matrix. This table summarizes the available methods.

Method	Description
'plus' (default)	Select $k$ observations from $X$ according to the $k$ -means++ algorithm on page 32-3390 for cluster center initialization.
'sample'	Select $k$ observations from $X$ at random.
'cluster'	Perform preliminary clustering phase on a random subsample (10%) of $X$ . This preliminary phase is itself initialized using <code>sample</code> , that is, the observations are selected at random.
matrix	A custom $k$ -by- $p$ matrix of starting locations. In this case, you can pass in <code>[]</code> for the $k$ input argument, and <code>kmedoids</code> infers $k$ from the first dimension of the matrix. You can also supply a 3-D array, implying a value for 'Replicates' from the array's third dimension.

Example: 'Start', 'sample'

## Output Arguments

### **idx** — Medoid indices

numeric column vector

Medoid indices, returned as a numeric column vector. `idx` has as many rows as  $X$ , and each row indicates the medoid assignment of the corresponding observation.

### **c** — Cluster medoid locations

numeric matrix

Cluster medoid locations, returned as a numeric matrix.  $C$  is a  $k$ -by- $p$  matrix, where row  $j$  is the medoid of cluster  $j$

### **sumd** — Within-cluster sums of point-to-medoid distances

numeric column vector



Within-cluster sums of point-to-medoid distances, returned as a numeric column vector. `sumd` is a  $k$ -by-1 vector, where element  $j$  is the sum of point-to-medoid distances within cluster  $j$ .

#### **D — Distances from each point to every medoid**

numeric matrix

Distances from each point to every medoid, returned as a numeric matrix. `D` is an  $n$ -by- $k$  matrix, where element  $(j,m)$  is the distance from observation  $j$  to medoid  $m$ .

#### **midx — Index to X**

column vector

Index to `X`, returned as a column vector of indices. `midx` is a  $k$ -by-1 vector and the indices satisfy `C = X(midx, :)`.

#### **info — Algorithm information**

struct

Algorithm information, returned as a struct. `info` contains options used by the function when executed such as  $k$ -medoid clustering algorithm (`algorithm`), method used to choose initial cluster medoid positions (`start`), distance measure (`distance`), number of iterations taken in the best replicate (`iterations`) and the replicate number of the returned results (`bestReplicate`).

## Definitions

### **$k$ -medoids Clustering**

$k$ -medoids clustering is a partitioning method commonly used in domains that require robustness to outlier data, arbitrary distance metrics, or ones for which the mean or median does not have a clear definition.

It is similar to  $k$ -means, and the goal of both methods is to divide a set of measurements or observations into  $k$  subsets or clusters so that the subsets minimize the sum of distances between a measurement and a center of the measurement's cluster. In the  $k$ -means algorithm, the center of the subset is the mean of measurements in the subset, often called a centroid. In the  $k$ -medoids algorithm, the center of the subset is a member of the subset, called a medoid.

The  $k$ -medoids algorithm returns medoids which are the actual data points in the data set. This allows you to use the algorithm in situations where the mean of the data does not exist within the data set. This is the main difference between  $k$ -medoids and  $k$ -means where the centroids returned by  $k$ -means may not be within the data set. Hence  $k$ -medoids is useful for clustering categorical data where a mean is impossible to define or interpret.

The function `kmedoids` provides several iterative algorithms that minimize the sum of distances from each object to its cluster medoid, over all clusters. One of the algorithms is called partitioning around medoids (PAM) [1] which proceeds in two steps.

- 1 Build-step: Each of  $k$  clusters is associated with a potential medoid. This assignment is performed using a technique specified by the 'Start' name-value pair argument.
- 2 Swap-step: Within each cluster, each point is tested as a potential medoid by checking if the sum of within-cluster distances gets smaller using that point as the medoid. If so, the point is defined as a new medoid. Every point is then assigned to the cluster with the closest medoid.

The algorithm iterates the build- and swap-steps until the medoids do not change, or other termination criteria are met.

You can control the details of the minimization using several optional input parameters to `kmedoids`, including ones for the initial values of the cluster medoids, and for the maximum number of iterations. By default, `kmedoids` uses the  $k$ -means++ algorithm on page 32-3390 for cluster medoid initialization and the squared Euclidean metric to determine distances.

## References

- [1] Kaufman, L., and Rousseeuw, P. J. (2009). Finding Groups in Data: An Introduction to Cluster Analysis. Hoboken, New Jersey: John Wiley & Sons, Inc.
- [2] Park, H-S, and Jun, C-H. (2009). A simple and fast algorithm for K-medoids clustering. *Expert Systems with Applications*. 36, 3336-3341.
- [3] Schlimmer, J.S. (1987). Concept Acquisition Through Representational Adjustment (Technical Report 87-19). Doctoral dissertation, Department of Information and Computer Science, University of California, Irvine.

- [4] Iba, W., Wogulis, J., and Langley, P. (1988). Trading off Simplicity and Coverage in Incremental Concept Learning. In Proceedings of the 5th International Conference on Machine Learning, 73-79. Ann Arbor, Michigan: Morgan Kaufmann.
- [5] Duch W, A.R., and Grabczewski, K. (1996) Extraction of logical rules from training data using backpropagation networks. Proc. of the 1st Online Workshop on Soft Computing, 19-30, pp. 25-30.
- [6] Duch, W., Adamczak, R., Grabczewski, K., Ishikawa, M., and Ueda, H. (1997). Extraction of crisp logical rules using constrained backpropagation networks - comparison of two new approaches. Proc. of the European Symposium on Artificial Neural Networks (ESANN'97), Bruges, Belgium 16-18.
- [7] Bache, K. and Lichman, M. (2013). UCI Machine Learning Repository [<http://archive.ics.uci.edu/ml>]. Irvine, CA: University of California, School of Information and Computer Science.

## See Also

`clusterdata` | `evalclusters` | `kmeans` | `linkage` | `linkage` | `pdist` | `silhouette`

**Introduced in R2014b**

## knnsearch

*k*-nearest neighbor search using *Kd*-tree or exhaustive search

### Syntax

```
Idx = knnsearch(Mdl, Y)
Idx = knnsearch(Mdl, Y, Name, Value)
[Idx, D] = knnsearch( ___ )
```

### Description

`Idx = knnsearch(Mdl, Y)` searches for the nearest neighbor (i.e., the closest point, row, or observation) in `Mdl.X` to each point (i.e., row or observation) in the query data `Y` using an exhaustive search or a *Kd*-tree. `knnsearch` returns `Idx`, which is a column vector of the indices in `Mdl.X` representing the nearest neighbors.

`Idx = knnsearch(Mdl, Y, Name, Value)` returns the indices of the closest points in `Mdl.X` to `Y` with additional options specified by one or more `Name, Value` pair arguments. For example, specify the number of nearest neighbors to search for, distance metric different from the one stored in `Mdl.Distance`. You can also specify which action to take if the closest distances are tied.

`[Idx, D] = knnsearch( ___ )` additionally returns the matrix `D` using any of the input arguments in the previous syntaxes. `D` contains the distances between each observation in `Y` that correspond to the closest observations in `Mdl.X`. The function arranges the columns of `D` in ascending order by closeness, with respect to the distance metric.

### Examples

#### Search for Nearest Neighbors Using a *Kd*-tree and Exhaustive Search

`knnsearch` accepts `ExhaustiveSearcher` or `KDTreeSearcher` model objects to search the training data for the nearest neighbors to the query data. An `ExhaustiveSearcher`

model invokes the exhaustive searcher algorithm, and a `KDTreeSearcher` model defines a  $K$  d-tree, which `knnsearch` uses to search for nearest neighbors.

Load Fisher's iris data set. Randomly reserve five observations from the data for query data.

```
load fisheriris
rng(1); % For reproducibility
n = size(meas,1);
idx = randsample(n,5);
X = meas(~ismember(1:n,idx),:); % Training data
Y = meas(idx,:); % Query data
```

The variable `meas` contains 4 predictors.

Grow a default four-dimensional  $K$  d-tree.

```
MdlKDT = KDTreeSearcher(X)
```

```
MdlKDT =
```

```
  KDTreeSearcher with properties:
```

```
    BucketSize: 50
    Distance: 'euclidean'
    DistParameter: []
             X: [145x4 double]
```

`MdlKDT` is a `KDTreeSearcher` model object. You can alter its writable properties using dot notation.

Prepare an exhaustive nearest neighbors searcher.

```
MdlES = ExhaustiveSearcher(X)
```

```
MdlES =
```

```
  ExhaustiveSearcher with properties:
```

```
    Distance: 'euclidean'
    DistParameter: []
```

```
X: [145x4 double]
```

MdlKDT is an `ExhaustiveSearcher` model object. It contains the options, such as the distance metric, to use to find nearest neighbors.

Alternatively, you can grow a  $K$  d-tree or prepare an exhaustive nearest neighbors searcher using `createns`.

Search the training data for the nearest neighbors indices that correspond to each query observation. Conduct both types of searches using the default settings. By default, the number of neighbors to search for per query observation is 1.

```
IdxKDT = knnsearch(MdlKDT,Y);  
IdxES = knnsearch(MdlES,Y);  
[IdxKDT IdxES]
```

```
ans =
```

```
    17    17  
     6     6  
     1     1  
    89    89  
   124   124
```

In this case, the results of the search are the same.

### Search for Nearest Neighbors of Query Data Using the Minkowski Distance

Load Fisher's iris data set.

```
load fisheriris
```

Remove five irises randomly from the predictor data to use as a query set.

```
rng(1); % For reproducibility  
n = size(meas,1); % Sample size  
qIdx = randsample(n,5); % Indices of query data  
X = meas(~ismember(1:n,qIdx),:);  
Y = meas(qIdx,:);
```

Grow a four-dimensional  $K$  d-tree using the training data. Specify to use the Minkowski distance for finding nearest neighbors later.

```
Mdl = KDTreeSearcher(X, 'Distance', 'minkowski')
```

```
Mdl =
```

```
    KDTreeSearcher with properties:
```

```
        BucketSize: 50
        Distance: 'minkowski'
        DistParameter: 2
        X: [145x4 double]
```

`Mdl` is a `KDTreeSearcher` model object. By default, the Minkowski distance exponent is 2.

Find the indices of the training data ( $X$ ) that are the two nearest neighbors of each point in the query data ( $Y$ ).

```
Idx = knnsearch(Mdl, Y, 'K', 2)
```

```
Idx =
```

```
    17     4
     6     2
     1    12
    89    66
   124   100
```

Each row of `Idx` corresponds to a query data observation, and the column order corresponds to the order of the nearest neighbors, with respect to ascending distance. For example, using the Minkowski distance, the second nearest neighbor of  $Y(3, :)$  is  $X(12, :)$ .

### Include Ties in Nearest Neighbors Search

Load Fisher's iris data set.

```
load fisheriris
```

Remove five irises randomly from the predictor data to use as a query set.

```
rng(4); % For reproducibility
n = size(meas,1); % Sample size
qIdx = randsample(n,5); % Indices of query data
X = meas(~ismember(1:n,qIdx),:);
Y = meas(qIdx,:);
```

Grow a four-dimensional  $K$  d-tree using the training data. Specify to use the Minkowski distance for finding nearest neighbors later.

```
Mdl = KDTreeSearcher(X);
```

`Mdl` is a `KDTreeSearcher` model object. By default, the distance metric for finding nearest neighbors is the Euclidean metric.

Find the indices of the training data ( $X$ ) that are the seven nearest neighbors of each point in the query data ( $Y$ ).

```
[Idx,D] = knnsearch(Mdl,Y,'K',7,'IncludeTies',true);
```

`Idx` and `D` are five-element cell arrays of vectors, with each vector having at least seven elements.

Display the lengths of the vectors in `Idx`.

```
cellfun('length',Idx)
```

```
ans =
```

```
8
7
7
7
7
```

Because cell 1 contains a vector with length greater than  $k = 7$ , query observation 1 ( $Y(1,:)$ ) is equally close to at least two observations in  $X$ .

Display the indices of the nearest neighbors to  $Y(1,:)$  and their distances.



```
nn5 = Idx{1}
nn5d = D{1}

nn5 =
    91    98    67    69    71    93    88    95

nn5d =
    Columns 1 through 7
    0.1414    0.2646    0.2828    0.3000    0.3464    0.3742    0.3873
    Column 8
    0.3873
```

Training observations 88 and 95 are 0.3873 cm away from query observation 1.

## Input Arguments

### **mdl** — Nearest neighbor searcher

ExhaustiveSearcher model object | KDTreeSearcher model object

Nearest neighbor searcher, specified as an ExhaustiveSearcher or KDTreeSearcher model object, respectively. To create Mdl, with the appropriate mode creator. You can also use createns.

If Mdl is an ExhaustiveSearcher model, then knnsearch searches for nearest neighbors using an exhaustive search. Otherwise, knnsearch uses the grown *Kd*-tree to search for nearest neighbors.

### **y** — Query data

numeric matrix

Query data, specified as a numeric matrix.

$Y$  is an  $m$ -by- $K$  matrix. Rows of  $Y$  correspond to observations (i.e., examples), and columns correspond to predictors (i.e., variables or features).  $Y$  must have the same number of columns as the training data stored in `Mdl.X`.

Data Types: `single` | `double`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'K', 2, 'Distance', 'minkowski'` specifies to find the two nearest neighbors of `Mdl.X` to each point in  $Y$  and to use the Minkowski distance metric.

### For Both Nearest Neighbor Searchers

#### Distance — Distance metric

`Mdl.Distance` (default) | `'cityblock'` | `'euclidean'` | `'mahalanobis'` | `'minkowski'` | `'seuclidean'` | function handle | ...

Distance metric used to find neighbors of the training data to the query observations, specified as the comma-separated pair consisting of `'Distance'` and a character vector or function handle.

For both types of nearest neighbor searchers, `Mdl` supports these distance metrics.

Value	Description
<code>'chebychev'</code>	Chebychev distance (maximum coordinate difference)
<code>'cityblock'</code>	City block distance
<code>'euclidean'</code>	Euclidean distance
<code>'minkowski'</code>	Minkowski distance

If `Mdl` is an `ExhaustiveSearcher` model object, then `knnsearch` supports these distance metrics.

Value	Description
'correlation'	One minus the sample linear correlation between observations (treated as sequences of values)
'cosine'	One minus the cosine of the included angle between observations (row vectors)
'hamming'	Hamming distance, which is the percentage of coordinates that differ.
'jaccard'	One minus the Jaccard coefficient, which is the percentage of nonzero coordinates that differ
'mahalanobis'	Mahalanobis distance
'seuclidean'	Standardized Euclidean distance
'spearman'	One minus the sample Spearman's rank correlation between observations (treated as sequences of values)

If `Mdl` is an `ExhaustiveSearcher` model object, then you can also specify a function handle for a custom distance metric using `@` (for example, `@distfun`). The custom distance function must:

- Have the form function `D2 = distfun(ZI, ZJ)`.
- Take as arguments:
  - A 1-by- $K$  vector `ZI` containing a single row from  $X$  or from the query points  $Y$
  - An  $m$ -by- $K$  matrix `ZJ` containing multiple rows of  $X$  or  $Y$
- Return an  $m$ -by-1 vector of distances `D2`. `D2(j)` is the distance between the observations `ZI` and `ZJ(j, :)`.

For more details, see “Distance Metrics” on page 18-16.

Example: `'Distance', 'minkowski'`

Data Types: `char` | `function_handle`

**IncludeTies** — Flag to include nearest neighbors that have the same distance from query observations

`false` (0) (default) | `true` (1)

Flag to include nearest neighbors that have the same distance from query observations, specified as the comma-separated pair consisting of 'IncludeTies' and false (0) or true (1).

If `IncludeTies` is true, then:

- `knnsearch` includes all nearest neighbors whose distances are equal to the  $K$ th smallest distance in the output arguments.
- `Idx` and `D` are  $m$ -by-1 cell arrays such that each cell contains a vector of at least  $K$  indices and distances, respectively. Each vector in `D` contains arranged distances in ascending order. Each row in `Idx` contains the indices of the nearest neighbors corresponding to these smallest distances in `D`.

If `IncludeTies` is false, then `knnsearch` chooses the observation with the smallest index among the observations that have the same distance from a query point.

Example: 'IncludeTies', true

**$\kappa$  — Number of nearest neighbors to search for in the training data per query observation**  
1 (default) | positive integer

Number of nearest neighbors to search for in the training data per query observation, specified as the comma-separated pair consisting of 'K' and a positive integer.

Example: 'K', 2

Data Types: single | double

**$p$  — Exponent for Minkowski distance metric**  
2 (default) | positive scalar

Exponent for the Minkowski distance metric, specified as the comma-separated pair consisting of 'P' and a positive scalar. If you specify `P` and do not specify 'Distance', 'minkowski', then the software throws an error.

Example: 'P', 3

Data Types: double | single

**For Exhaustive Nearest Neighbor Searchers**

**`cov` — Covariance matrix for Mahalanobis distance metric**  
`nancov(X)` (default) | positive definite matrix

Covariance matrix for the Mahalanobis distance metric, specified as the comma-separated pair consisting of 'Cov' and a positive definite matrix. Cov is a  $K$ -by- $K$  matrix, where  $K$  is the number of columns of X. If you specify Cov and do not specify 'Distance', 'mahalanobis', then knnsearch throws an error.

Example: 'Cov', eye(3)

Data Types: double | single

### scale — Scale parameter value for standard Euclidean distance metric

nanstd(X) (default) | nonnegative numeric vector

Scale parameter value for the standard Euclidean distance metric, specified as the comma-separated pair consisting of 'Scale' and a nonnegative numeric vector. Scale has length  $K$ , where  $K$  is the number of columns of X.

The software scales each difference between the training and query data using the corresponding element of Scale. If you specify Scale and do not specify 'Distance', 'seuclidean', then knnsearch throws an error.

Example: 'Scale', quantile(X,0.75) - quantile(X,0.25)

Data Types: double | single

---

**Note** If you specify 'Distance', 'Cov', 'P', or 'Scale', then Mdl.Distance and Mdl.DistanceParameter do not change value.

---

## Output Arguments

### Idx — Training data indices of nearest neighbors

numeric matrix | cell array of numeric vectors

Training data indices of nearest neighbors, returned as a numeric matrix or cell array of numeric vectors.

- If you do not specify IncludeTies (false by default), then Idx is an  $m$ -by- $K$  numeric matrix, where  $m$  is the number of rows in Y and  $K$  is the number of searched nearest neighbors. Idx(j,k) indicates that Mdl.X(Idx(j,k), :) is the observation with the  $k$ th smallest distance to the query observation Y(j, :).

- If you specify `'IncludeTies', true`, then `Idx` is an  $m$ -by-1 cell array such that cell `j (Idx{j})` contains a vector of at least  $K$  indices of the closest observations in `Mdl.X` to the query observation `Y(j, :)`. The function arranges the elements of the vectors in ascending order by distance.

#### **D — Distances of nearest neighbors to the query data**

numeric matrix | cell array of numeric vectors

Distances of the nearest neighbors to the query data, returned as a numeric matrix or cell array of numeric vectors.

- If you do not specify `IncludeTies` (`false` by default), then `D` is an  $m$ -by- $K$  numeric matrix, where  $m$  is the number of rows in `Y` and  $K$  is the number of searched nearest neighbors. `D(j, k)` is the distance `Mdl.X(Idx(j, k), :)` is from the query observation `Y(j, :)` with respect to the distance metric, and it represents the  $k$ th smallest distance.
- If you specify `'IncludeTies', true`, then `D` is an  $m$ -by-1 cell array such that cell `j (D{j})` contains a vector of at least  $K$  distances of the closest observations in `Mdl.X` to the query observation `Y(j, :)`. The function arranges the elements of the vectors in ascending order by distance.

## Algorithms

For positive integer  $K$ , `knnsearch` finds the  $K$  points in `Mdl.X` that are nearest each `Y` point. In contrast, for positive scalar  $r$ , `rangesearch` finds all the points in `Mdl.X` that are within a distance  $r$  of each `Y` point.

## Alternatives

- `knnsearch` is an object function that requires an `ExhaustiveSearcher` or a `KDTreeSearcher` model object and query data. Under equivalent conditions, `knnsearch` returns the same results as `knnsearch` when you specify the name-value pair argument `'NSMethod', 'exhaustive'` or `'NSMethod', 'kdtree'`, respectively.
- For  $k$ -nearest neighbors classification, see `fitcknn` and `ClassificationKNN`.

## References

- [1] Friedman, J. H., Bentley, J., and Finkel, R. A. (1977). “An Algorithm for Finding Best Matches in Logarithmic Expected Time.” *ACM Transactions on Mathematical Software* Vol. 3, Issue 3, Sept. 1977, pp. 209–226.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This table contains notes about arguments.

Argument	Notes and Limitations
Mdl	<ul style="list-style-type: none"> <li>• Only the exhaustive search algorithm supports code generation. Therefore, Mdl must be an ExhaustiveSearcher model object.</li> <li>• There are two ways to use Mdl in code generation. For an example, see “Code Generation Workflow for Exhaustive Searcher” on page 31-47. <ul style="list-style-type: none"> <li>• Save a trained model by using <code>saveCompactModel</code>, and then load the model by using <code>loadCompactModel</code> within a function that you declare.</li> <li>• Include <code>coder.Constant(Mdl)</code> in the <code>-args</code> value of <code>codegen</code>.</li> </ul> </li> <li>• Must be a compile-time constant; its value cannot change in the generated code.</li> </ul>
'Distance'	<ul style="list-style-type: none"> <li>• Cannot be a custom distance function.</li> <li>• Must be a compile-time constant; its value cannot change in the generated code.</li> </ul>

Argument	Notes and Limitations
'IncludeTies'	Must be a compile-time constant; its value cannot change in the generated code.
Name-value pair arguments	Names in name-value pair arguments must be compile-time constants. For example, to allow a user-defined exponent for the Minkowski distance in the generated code, include <code>{coder.Constant('Distance'), coder.Constant('Minkowski'), coder.Constant('P'), 0}</code> in the <code>-args</code> value of <code>codegen</code> .
Idx	When you specify 'IncludeTies' as true, the sorted order of tied distances in the generated code can be different from the order in MATLAB due to numerical precision.

- For code generation, `knnsearch` uses `parfor` (by default) to create loops that run in parallel on supported shared-memory multicore platforms. If your compiler does not support the Open Multiprocessing (OpenMP) application interface or you disable OpenMP library, MATLAB Coder treats the `parfor`-loops as `for`-loops. To find supported compilers, see [http://www.mathworks.com/support/compilers/current\\_release/](http://www.mathworks.com/support/compilers/current_release/). To disable OpenMP library, specify the `EnableOpenMP` property of the `codegen` configuration object as `false`. For details, see `coder.CodeConfig`.

## See Also

`ClassificationKNN` | `ExhaustiveSearcher` | `KDTreeSearcher` | `createns` | `fitcknn` | `knnsearch` | `rangesearch`

## Topics

`ExhaustiveSearcher`

`KDTreeSearcher`

“k-Nearest Neighbor Search and Radius Search” on page 18-18

“Distance Metrics” on page 18-16

“Code Generation Workflow for Exhaustive Searcher” on page 31-47

**Introduced in R2010a**



# knnsearch

Find  $k$ -nearest neighbors using data

## Syntax

```
IDX = knnsearch(X,Y)
[IDX,D] = knnsearch(X,Y)
[IDX,D] = knnsearch(X,Y,'Name',Value)
```

## Description

`IDX = knnsearch(X,Y)` finds the nearest neighbor in `X` for each point in `Y`. `IDX` is a column vector with  $m_y$  rows. Each row in `IDX` contains the index of nearest neighbor in `X` for the corresponding row in `Y`.

`[IDX,D] = knnsearch(X,Y)` returns an  $m_y$ -by-1 vector `D` containing the distances between each observation in `Y` and the corresponding closest observation in `X`. That is, `D(i)` is the distance between `X(Idx(i),:)` and `Y(i,:)`.

`[IDX,D] = knnsearch(X,Y,'Name',Value)` accepts one or more optional comma-separated name-value pair arguments. Specify `Name` inside single quotes.

`knnsearch` does not save a search object. To create a search object, use `createns`.

## Input Arguments

### **X**

An  $m_x$ -by- $n$  numeric matrix. Rows of `X` correspond to observations and columns correspond to variables.

### **Y**

An  $m_y$ -by- $n$  numeric matrix of query points. Rows of `Y` correspond to observations and columns correspond to variables.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### **K**

Positive integer specifying the number of nearest neighbors in `X` for each point in `Y`. Default is 1. `IDX` and `D` are *my-by-K* matrices. `D` sorts the distances in each row in ascending order. Each row in `IDX` contains the indices of the `K` closest neighbors in `X` corresponding to the `K` smallest distances in `D`.

### **IncludeTies**

A logical value indicating whether `knnsearch` includes all the neighbors whose distance values are equal to the `K`th smallest distance. If `IncludeTies` is `true`, `knnsearch` includes all these neighbors. In this case, `IDX` and `D` are *my-by-1* cell arrays. Each row in `IDX` and `D` contains a vector with at least `K` numeric numbers. `D` sorts the distances in each vector in ascending order. Each row in `IDX` contains the indices of the closest neighbors corresponding to these smallest distances in `D`.

**Default:** `false`

### **NSMethod**

Nearest neighbor search method. Value is either:

- `'kdtree'` — Creates and uses a `Kd`-tree to find nearest neighbors. This is the default value when the number of columns of `X` is less than 10, `X` is not sparse, and the distance measure is one of the following measures. `'kdtree'` is only valid when the distance measure is one of the following:
  - `'euclidean'`
  - `'cityblock'`
  - `'minkowski'`
  - `'chebychev'`
- `'exhaustive'` — Uses the exhaustive search algorithm by computing the distance values from all the points in `X` to each point in `Y` to find nearest neighbors.

## Distance

A character vector or a function handle specifying the distance metric. The value can be one of the following:

- 'euclidean' — Euclidean distance (default).
- 'seuclidean' — Standardized Euclidean distance. Each coordinate difference between rows in  $X$  and the query matrix is scaled by dividing by the corresponding element of the standard deviation computed from  $X$ ,  $S = \text{nanstd}(X)$ . To specify another value for  $S$ , use the `Scale` argument.
- 'cityblock' — City block distance.
- 'chebychev' — Chebychev distance (maximum coordinate difference).
- 'minkowski' — Minkowski distance. The default exponent is 2. To specify a different exponent, use the 'P' argument.
- 'mahalanobis' — Mahalanobis distance, computed using a positive definite covariance matrix  $C$ . The default value of  $C$  is  $\text{nancov}(X)$ . To change the value of  $C$ , use the `Cov` parameter.
- 'cosine' — 1 minus the cosine of the included angle between observations (treated as vectors).
- 'correlation' — One minus the sample linear correlation between observations (treated as sequences of values).
- 'spearman' — One minus the sample Spearman's rank correlation between observations (treated as sequences of values).
- 'hamming' — Hamming distance, which is the percentage of coordinates that differ.
- 'jaccard' — One minus the Jaccard coefficient, which is the percentage of nonzero coordinates that differ.
- custom distance function — A distance function specified using @ (for example, @distfun). A custom distance function must
  - Have the form `function D2 = distfun(ZI, ZJ)`
  - Take as arguments:
    - A 1-by- $n$  vector  $ZI$  containing a single row from  $X$  or from the query points  $Y$
    - An  $m2$ -by- $n$  matrix  $ZJ$  containing multiple rows of  $X$  or  $Y$
  - Return an  $m2$ -by-1 vector of distances  $D2$ , whose  $j$ th element is the distance between the observations  $ZI$  and  $ZJ(j, :)$

For more information on these distance metrics, see “Distance Metrics” on page 18-16.

### **P**

A positive scalar,  $p$ , indicating the exponent of the Minkowski distance. This parameter is only valid if the `Distance` is 'minkowski'. Default is 2.

### **Cov**

A positive definite matrix indicating the covariance matrix when computing the Mahalanobis distance. This parameter is only valid when `Distance` is 'mahalanobis'. Default is `nancov(X)`.

### **Scale**

A vector  $S$  containing nonnegative values, with length equal to the number of columns in  $X$ . Each coordinate of  $X$  and each query point is scaled by the corresponding element of  $S$  when computing the standardized Euclidean distance. This argument is only valid when `Distance` is 'seuclidean'. Default is `nanstd(X)`.

### **BucketSize**

The maximum number of data points in the leaf node of the  $kd$ -tree. This argument is only meaningful when using the  $kd$ -tree search method. Default is 50.

## Examples

### **Classify Using k-Nearest Neighbors**

Find the 10 nearest neighbors in  $x$  to each point in  $y$  using first the 'minkowski' distance metric with a  $p$  value of 5, and then using the 'chebychev' distance metric.

Load Fisher's iris data set

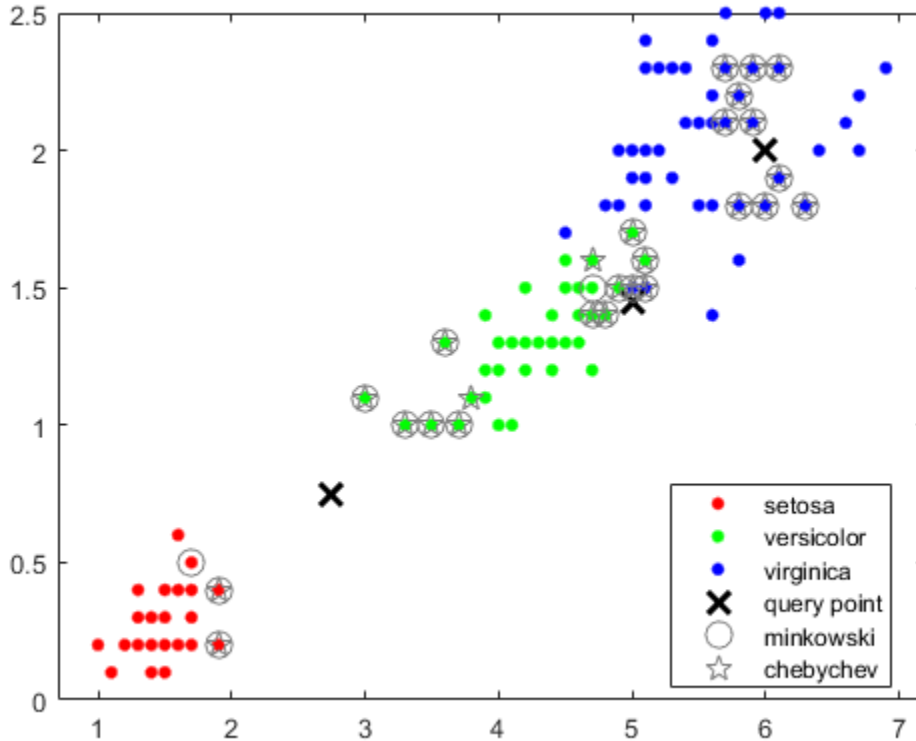
```
load fisheriris
x = meas(:,3:4);
y = [5 1.45;6 2;2.75 .75];
```

Perform a `knnsearch` between  $x$  and the query points in  $y$ , using first Minkowski then Chebychev distance metrics.

```
[n,d]=knnsearch(x,y,'k',10,'distance','minkowski','p',5);  
[ncb,dcb] = knnsearch(x,y,'k',10,...  
    'distance','chebychev');
```

Visualize the results of the two different nearest neighbors searches. Plot the training data. Plot an X for the query points. Use circles to denote the Minkowski nearest neighbors. Use pentagrams to denote the Chebychev nearest neighbors.

```
gscatter(x(:,1),x(:,2),species)  
line(y(:,1),y(:,2),'marker','x','color','k',...  
    'markersize',10,'linewidth',2,'linestyle','none')  
line(x(n,1),x(n,2),'color',[.5 .5 .5],'marker','o',...  
    'linestyle','none','markersize',10)  
line(x(ncb,1),x(ncb,2),'color',[.5 .5 .5],'marker','p',...  
    'linestyle','none','markersize',10)  
legend('setosa','versicolor','virginica','query point',...  
    'minkowski','chebychev','Location','best')
```



## Tips

- For a fixed positive integer  $K$ , `knnsearch` finds the  $K$  points in  $X$  that are nearest each point in  $Y$ . In contrast, for a fixed positive real value  $r$ , `rangeseach` finds all the points in  $X$  that are within a distance  $r$  of each point in  $Y$ .

## Algorithms

For information on a specific search algorithm, see “Distance Metrics” on page 18-16.

## Alternatives

`knnsearch` is the object function of `ExhaustiveSearcher` and `KDTreeSearcher` models for a  $k$ -nearest neighbor search. If you set the `NSMethod` name-value pair argument to the appropriate value ('`exhaustive`' for an exhaustive search or '`kdtree`' for a *Kd-tree*), then the search results are equivalent to conducting a distance search using `knnsearch` and without using model objects.

## References

- [1] Friedman, J. H., Bentely, J., and Finkel, R. A. “An Algorithm for Finding Best Matches in Logarithmic Expected Time”, *ACM Transactions on Mathematical Software*. Vol. 3, Issue 3, 1977, pp. 209–226.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Only the exhaustive search algorithm supports code generation. Therefore, the default value of '`NSMethod`' is '`exhaustive`' for code generation, and you cannot change the value.
- The value of the '`Distance`' name-value pair argument must be a compile-time constant and cannot be a custom distance function.
- The value of the '`IncludeTies`' name-value pair argument must be a compile-time constant.
- Names in name-value pair arguments must be compile-time constants. For example, to allow a user-defined exponent for the Minkowski distance in the generated code, include 

```
{coder.Constant('Distance'),coder.Constant('Minkowski'),coder.Constant('P'),0}
```

 in the `-args` value of `codegen`.

- When you specify 'IncludeTies' as true, the sorted order of tied distances in the generated code can be different from the order in MATLAB due to numerical precision.
- For code generation, knnsearch uses parfor (by default) to create loops that run in parallel on supported shared-memory multicore platforms. If your compiler does not support the Open Multiprocessing (OpenMP) application interface or you disable OpenMP library, MATLAB Coder treats the parfor-loops as for-loops. To find supported compilers, see [http://www.mathworks.com/support/compilers/current\\_release/](http://www.mathworks.com/support/compilers/current_release/). To disable OpenMP library, specify the EnableOpenMP property of the codegen configuration object as false. For details, see coder.CodeConfig.

## See Also

ExhaustiveSearcher | KDTreeSearcher | createns | knnsearch

## Topics

ExhaustiveSearcher

KDTreeSearcher

“k-Nearest Neighbor Search and Radius Search” on page 18-18

“Distance Metrics” on page 18-16

**Introduced in R2010a**



# kruskalwallis

Kruskal-Wallis test

## Syntax

```
p = kruskalwallis(x)
p = kruskalwallis(x,group)
p = kruskalwallis(x,group,displayopt)
[p,tbl,stats] = kruskalwallis(____)
```

## Description

`p = kruskalwallis(x)` returns the  $p$ -value for the null hypothesis that the data in each column of the matrix `x` comes from the same distribution, using a Kruskal-Wallis test on page 32-3444. The alternative hypothesis is that not all samples come from the same distribution. `kruskalwallis` also returns an ANOVA table and a box plot.

`p = kruskalwallis(x,group)` returns the  $p$ -value for a test of the null hypothesis that the data in each categorical group, as specified by the grouping variable `group` comes from the same distribution. The alternative hypothesis is that not all groups come from the same distribution.

`p = kruskalwallis(x,group,displayopt)` returns the  $p$ -value of the test and lets you display or suppress the ANOVA table and box plot.

`[p,tbl,stats] = kruskalwallis(____)` also returns the ANOVA table as the cell array `tbl` and the structure `stats` containing information about the test statistics.

## Examples

### Test Data Samples for the Same Distribution

Create two different normal probability distribution objects. The first distribution has  $\mu = 0$  and  $\sigma = 1$ , and the second distribution has  $\mu = 2$  and  $\sigma = 1$ .

```
pd1 = makedist('Normal');  
pd2 = makedist('Normal','mu',2,'sigma',1);
```

Create a matrix of sample data by generating random numbers from these two distributions.

```
rng('default'); % for reproducibility  
x = [random(pd1,20,2), random(pd2,20,1)];
```

The first two columns of  $x$  contain data generated from the first distribution, while the third column contains data generated from the second distribution.

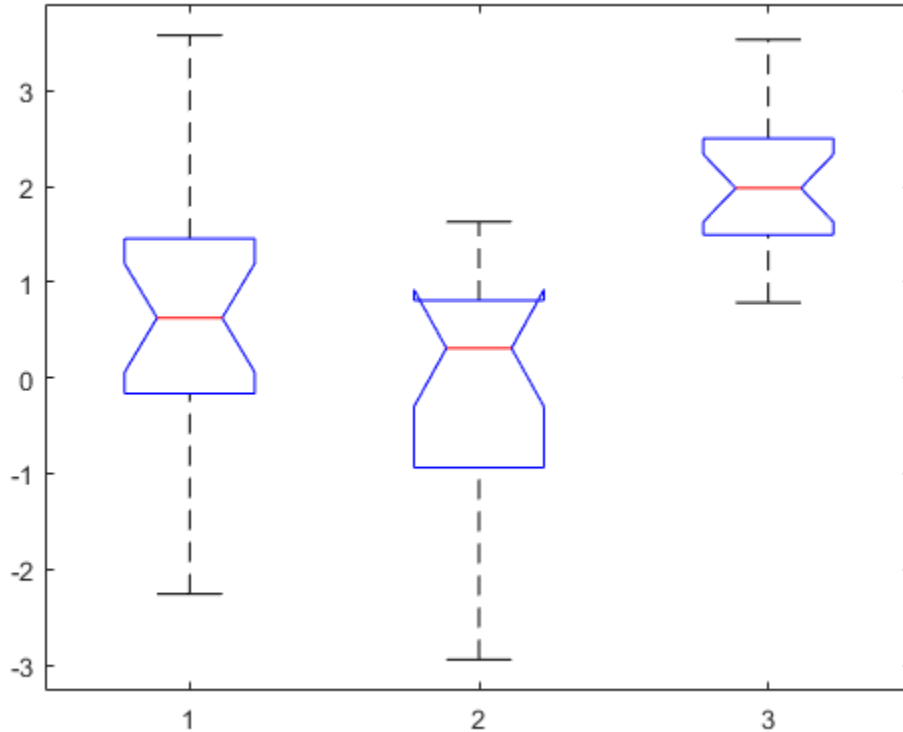
Test the null hypothesis that the sample data from each column in  $x$  comes from the same distribution.

```
p = kruskalwallis(x)
```

```
p =  
3.6896e-06
```

**Kruskal-Wallis ANOVA Table**

Source	SS	df	MS	Chi-sq	Prob>Chi-sq
Columns	7631.1	2	3815.55	25.02	3.68957e-06
Error	10363.9	57	181.82		
Total	17995	59			



The returned value of `p` indicates that `kruskalwallis` rejects the null hypothesis that all three data samples come from the same distribution at a 1% significance level. The ANOVA table provides additional test results, and the box plot visually presents the summary statistics for each column in `x`.

### Conduct Followup Tests for Unequal Medians

Create two different normal probability distribution objects. The first distribution has `mu = 0` and `sigma = 1`. The second distribution has `mu = 2` and `sigma = 1`.

```
pd1 = makedist('Normal');
pd2 = makedist('Normal', 'mu', 2, 'sigma', 1);
```

Create a matrix of sample data by generating random numbers from these two distributions.

```
rng('default'); % for reproducibility
x = [random(pd1,20,2), random(pd2,20,1)];
```

The first two columns of `x` contain data generated from the first distribution, while the third column contains data generated from the second distribution.

Test the null hypothesis that the sample data from each column in `x` comes from the same distribution. Suppress the output displays, and generate the structure `stats` to use in further testing.

```
[p,tbl,stats] = kruskalwallis(x,[],'off')
```

```
p =
```

```
3.6896e-06
```

```
tbl =
```

```
4x6 cell array
```

```
Columns 1 through 5
```

{'Source' }	{'SS' }	{'df' }	{'MS' }	{'Chi-sq' }
{'Columns' }	{[7.6311e+03]}	{[ 2]}	{[3.8155e+03]}	{[ 25.0200]}
{'Error' }	{[1.0364e+04]}	{[57]}	{[ 181.8228]}	{0x0 double }
{'Total' }	{[ 17995]}	{[59]}	{0x0 double }	{0x0 double }

```
Column 6
```

{'Prob>Chi-sq' }
{[ 3.6896e-06]}
{0x0 double }
{0x0 double }

```
stats =
```

```
struct with fields:
```

```
gnames: [3x1 char]
n: [20 20 20]
source: 'kruskalwallis'
meanranks: [26.7500 18.9500 45.8000]
sumt: 0
```

The returned value of `p` indicates that the test rejects the null hypothesis at the 1% significance level. You can use the structure `stats` to perform additional followup testing. The cell array `tbl` contains the same data as the graphical ANOVA table, including column and row labels.

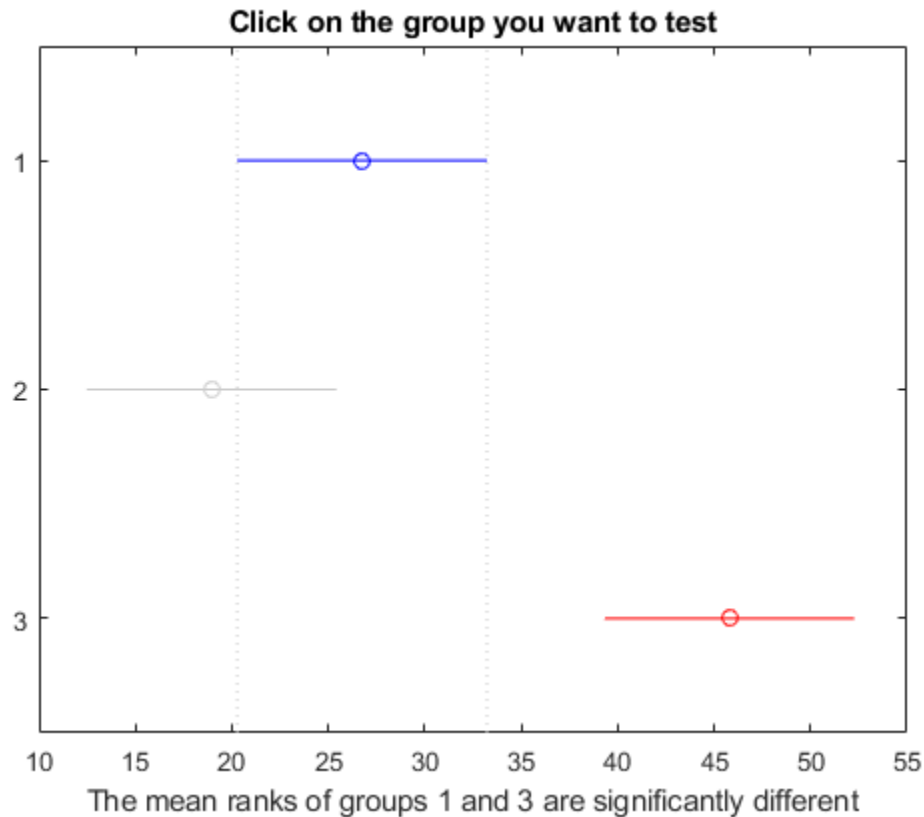
Conduct a followup test to identify which data sample comes from a different distribution.

```
c = multcompare(stats)
```

Note: Intervals can be used for testing but are not simultaneous confidence intervals.

```
c =
```

1.0000	2.0000	-5.1435	7.8000	20.7435	0.3345
1.0000	3.0000	-31.9935	-19.0500	-6.1065	0.0016
2.0000	3.0000	-39.7935	-26.8500	-13.9065	0.0000



The results indicate that there is a significant difference between groups 1 and 3, so the test rejects the null hypothesis that the data in these two groups comes from the same distribution. The same is true for groups 2 and 3. However, there is not a significant difference between groups 1 and 2, so the test does not reject the null hypothesis that these two groups come from the same distribution. Therefore, these results suggest that the data in groups 1 and 2 come from the same distribution, and the data in group 3 comes from a different distribution.

### Test for the Same Distribution Across Groups

Create a vector, `strength`, containing measurements of the strength of metal beams. Create a second vector, `alloy`, indicating the type of metal alloy from which the corresponding beam is made.

```
strength = [82 86 79 83 84 85 86 87 74 82 ...  
           78 75 76 77 79 79 77 78 82 79];  
  
alloy = {'st', 'st', 'st', 'st', 'st', 'st', 'st', 'st', ...  
        'al1', 'al1', 'al1', 'al1', 'al1', 'al1', ...  
        'al2', 'al2', 'al2', 'al2', 'al2', 'al2'};
```

Test the null hypothesis that the beam strength measurements have the same distribution across all three alloys.

```
p = kruskalwallis(strength, alloy, 'off')
```

```
p =  
  
    0.0018
```

The returned value of `p` indicates that the test rejects the null hypothesis at the 1% significance level.

## Input Arguments

### **x** — Sample data

vector | matrix

Sample data for the hypothesis test, specified as a vector or an  $m$ -by- $n$  matrix. If `x` is an  $m$ -by- $n$  matrix, each of the  $n$  columns represents an independent sample containing  $m$  mutually independent observations.

Data Types: `single` | `double`

### **group** — Grouping variable

categorical variable | vector | character array | cell array



Grouping variable, specified as a categorical variable, vector, character array, or cell array.

- If `x` is a vector, then each element in `group` identifies the group to which the corresponding element in `x` belongs, and `group` must be a vector of the same length as `x`. If a row of `group` contains an empty cell or empty character vector, that row and the corresponding observation in `x` are disregarded. NaN values in either `x` or `group` are similarly ignored.
- If `x` is a matrix, then each column in `x` represents a different group, and you can use `group` to specify labels for these columns. The number of elements in `group` and the number of columns in `x` must be equal.

The labels contained in `group` also annotate the box plot.

Example:

```
{'red','blue','green','blue','red','blue','green','green','red'}
```

Data Types: `single` | `double` | `char` | `cell`

#### **displayopt** — Display option

`'on'` (default) | `'off'`

Display option, specified as `'on'` or `'off'`. If `displayopt` is `'on'`, `kruskalwallis` displays the following figures:

- An ANOVA table containing the sums of squares, degrees of freedom, and other quantities calculated based on the ranks of the data in `x`.
- A box plot of the data in each column of the data matrix `x`. The box plots are based on the actual data values, rather than on the ranks.

If `displayopt` is `'off'`, `kruskalwallis` does not display these figures.

If you specify a value for `displayopt`, you must also specify a value for `group`. If you do not have a grouping variable, specify `group` as `[]`.

Example: `'off'`

## Output Arguments

### **p** — *p*-value

scalar value in the range [0,1]

*p*-value of the test, returned as a scalar value in the range [0,1]. *p* is the probability of observing a test statistic as extreme as, or more extreme than, the observed value under the null hypothesis. Small values of *p* cast doubt on the validity of the null hypothesis.

### **tbl** — ANOVA table

cell array

ANOVA table of test results, returned as a cell array. `tbl` includes the sums of squares, degrees of freedom, and other quantities calculated based on the ranks of the data in `x`, as well as column and row labels.

### **stats** — Test data

structure

Test data, returned as a structure. You can perform followup multiple comparison tests on pairs of sample medians by using `multcompare`, with `stats` as the input value.

## Definitions

### Kruskal-Wallis Test

The Kruskal-Wallis test is a nonparametric version of classical one-way ANOVA, and an extension of the Wilcoxon rank sum test to more than two groups. It compares the medians of the groups of data in `x` to determine if the samples come from the same population (or, equivalently, from different populations with the same distribution).

The Kruskal-Wallis test uses ranks of the data, rather than numeric values, to compute the test statistics. It finds ranks by ordering the data from smallest to largest across all groups, and taking the numeric index of this ordering. The rank for a tied observation is equal to the average rank of all observations tied with it. The *F*-statistic used in classical one-way ANOVA is replaced by a chi-square statistic, and the *p*-value measures the significance of the chi-square statistic.

The Kruskal-Wallis test assumes that all samples come from populations having the same continuous distribution, apart from possibly different locations due to group effects,

and that all observations are mutually independent. By contrast, classical one-way ANOVA replaces the first assumption with the stronger assumption that the populations have normal distributions.

## See Also

`anova1` | `boxplot` | `friedman` | `multcompare` | `ranksum`

## Topics

“Grouping Variables” on page 2-59

**Introduced before R2006a**

## ksdensity

Kernel smoothing function estimate for univariate and bivariate data

### Syntax

```
[f,xi] = ksdensity(x)
[f,xi] = ksdensity(x,pts)
[f,xi] = ksdensity(x,pts,Name,Value)
[f,xi,bw] = ksdensity(____)

ksdensity(____)
ksdensity(ax,____)
```

### Description

`[f,xi] = ksdensity(x)` returns a probability density estimate, `f`, for the sample data in the vector or two-column matrix `x`. The estimate is based on a normal kernel function, and is evaluated at equally-spaced points, `xi`, that cover the range of the data in `x`. `ksdensity` estimates the density at 100 points for univariate data, or 900 points for bivariate data.

`ksdensity` works best with continuously distributed samples.

`[f,xi] = ksdensity(x,pts)` returns a probability density estimate, `f`, for the data sample in the vector or two-column matrix `x`, evaluated at the specified values in vector `pts`. Here, `xi` and `pts` contain identical values.

`[f,xi] = ksdensity(x,pts,Name,Value)` returns a probability density estimate, `f`, for the sample in the vector or two-column matrix `x`, with additional options specified by one or more `Name,Value` pair arguments.

For example, you can define the function type `ksdensity` evaluates, such as probability density, cumulative probability, survivor function, and so on. Or you can specify the bandwidth of the smoothing window.

`[f,xi,bw] = ksdensity(____)` also returns the bandwidth of the kernel smoothing window, `bw`. The default bandwidth is the optimal for normal densities.

`ksdensity(____)` plots the kernel smoothing function estimate.

`ksdensity(ax,____)` plots the results using axes with the handle, `ax`, instead of the current axes returned by `gca`.

## Examples

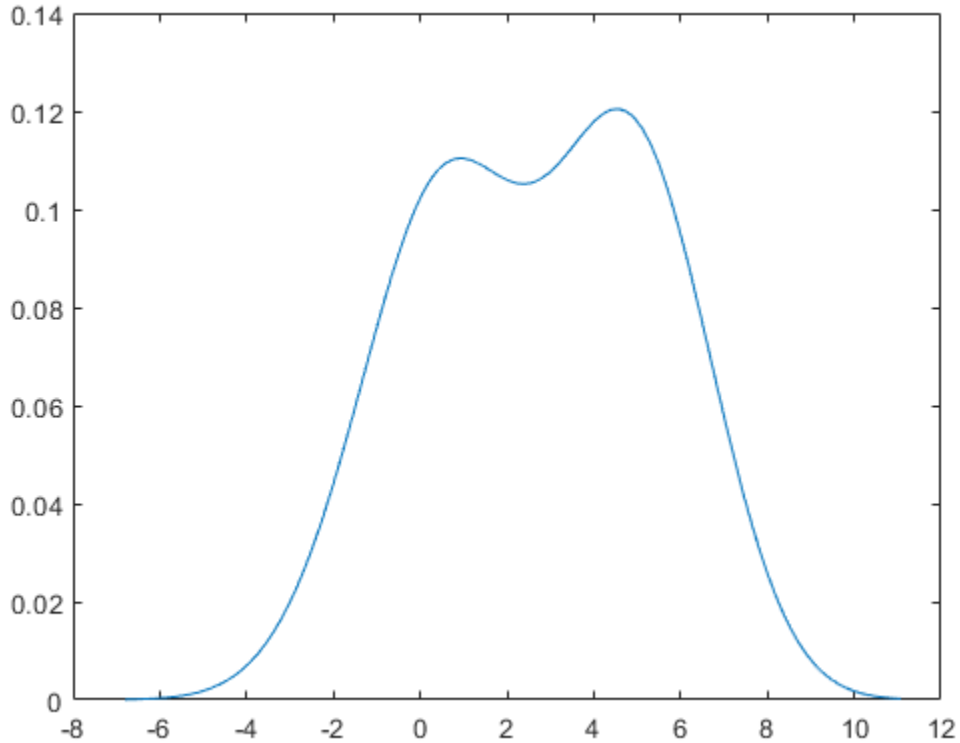
### Estimate Density

Generate a sample data set from a mixture of two normal distributions.

```
rng('default') % For reproducibility
x = [randn(30,1); 5+randn(30,1)];
```

Plot the estimated density.

```
[f,xi] = ksdensity(x);
figure
plot(xi,f);
```



The density estimate shows the bimodality of the sample.

### Estimate Density with Boundary Correction

Generate a nonnegative sample data set from the half-normal distribution.

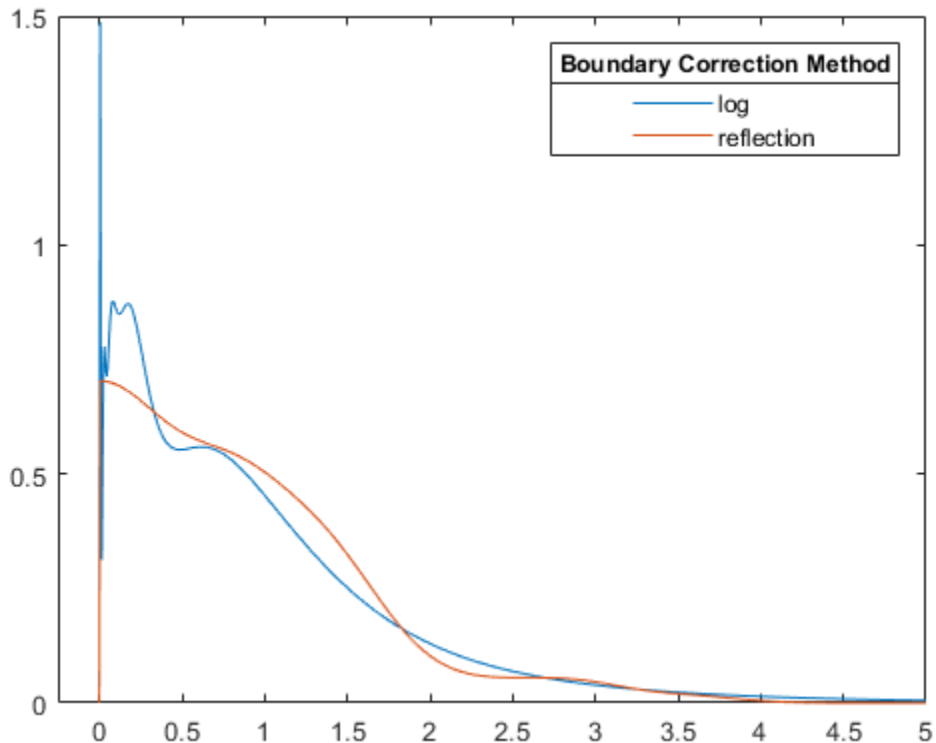
```
rng('default') % For reproducibility
pd = makedist('HalfNormal','mu',0,'sigma',1);
x = random(pd,100,1);
```

Estimate pdfs with two different boundary correction methods, log transformation and reflection, by using the 'BoundaryCorrection' name-value pair argument.

```
pts = linspace(0,5,1000); % points to evaluate the estimator
[f1,xi1] = ksdensity(x,pts,'Support','positive');
[f2,xi2] = ksdensity(x,pts,'Support','positive','BoundaryCorrection','reflection');
```

Plot the two estimated pdfs.

```
plot(xi1,f1,xi2,f2)
lgd = legend('log','reflection');
title(lgd, 'Boundary Correction Method')
xl = xlim;
xlim([xl(1)-0.25 xl(2)])
```



ksdensity uses a boundary correction method when you specify either positive or bounded support. The default boundary correction method is log transformation. When

`ksdensity` transforms the support back, it introduces the  $1/x$  term in the kernel density estimator. Therefore, the estimate has a peak near  $x = 0$ . On the other hand, the reflection method does not cause undesirable peaks near the boundary.

### Estimate Cumulative Distribution Function at Specified Values

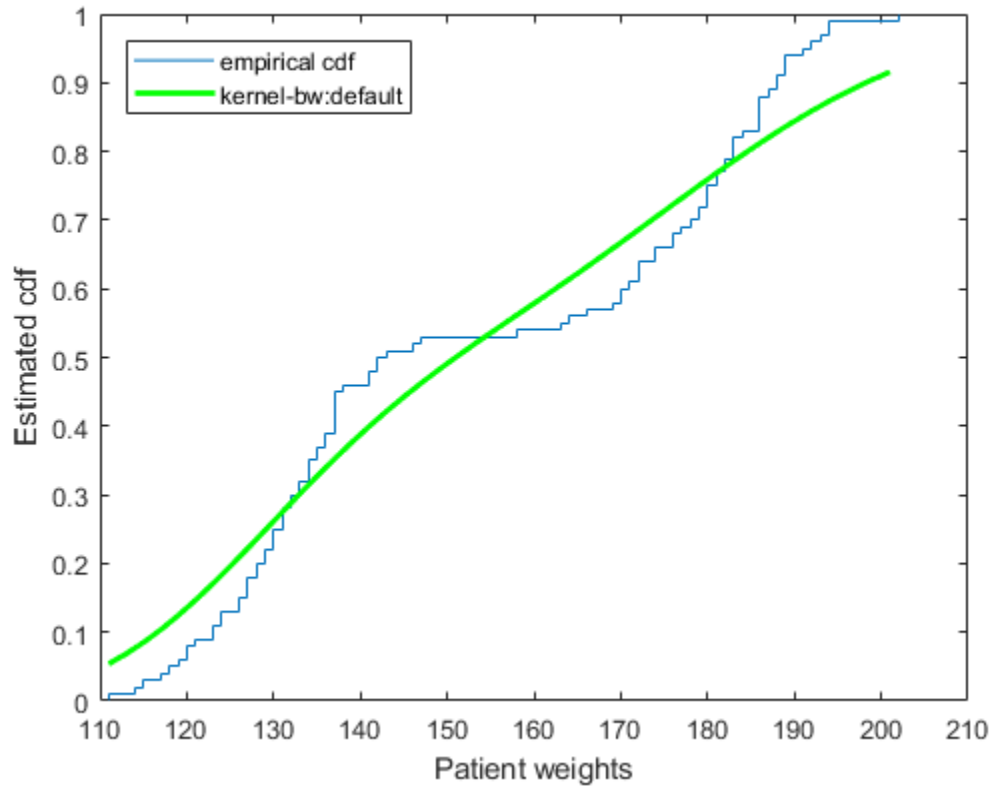
Load the sample data.

```
load hospital
```

Compute and plot the estimated cdf evaluated at a specified set of values.

```
pts = (min(hospital.Weight):2:max(hospital.Weight));  
figure()  
ecdf(hospital.Weight)  
hold on  
[f,xi,bw] = ksdensity(hospital.Weight,pts,'Support','positive',...  
    'Function','cdf');  
plot(xi,f,'-g','LineWidth',2)  
legend('empirical cdf','kernel-bw:default','Location','northwest')  
xlabel('Patient weights')  
ylabel('Estimated cdf')
```





`ksdensity` seems to smooth the cumulative distribution function estimate too much. An estimate with a smaller bandwidth might produce a closer estimate to the empirical cumulative distribution function.

Return the bandwidth of the smoothing window.

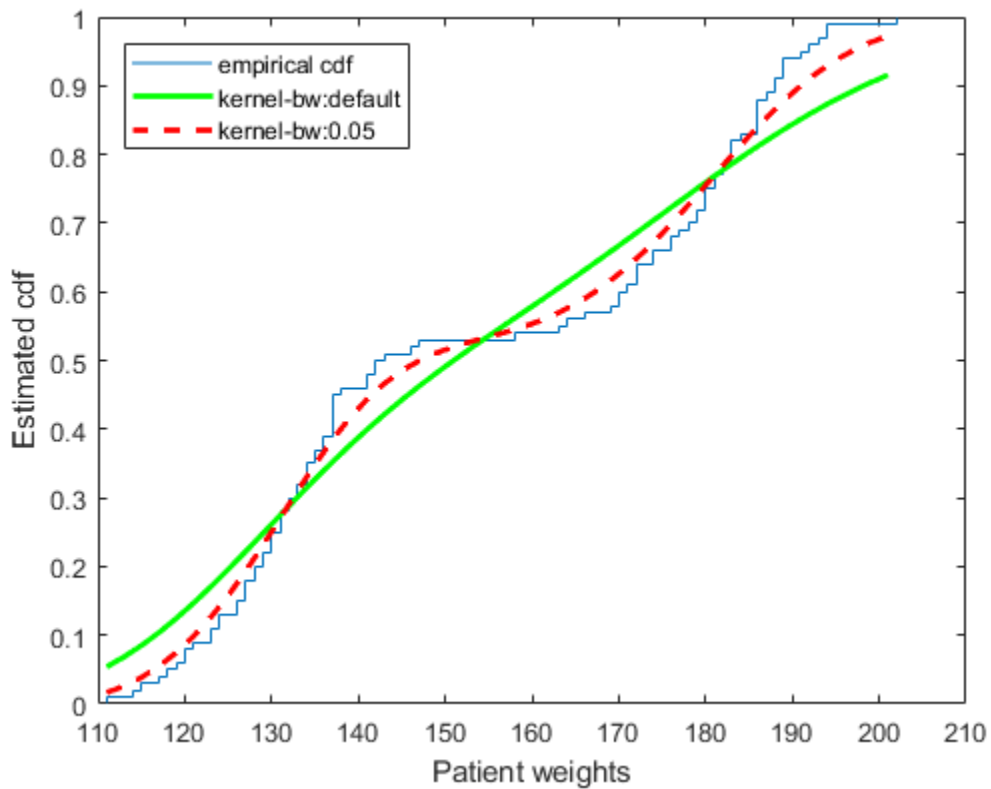
```
bw
```

```
bw =
```

```
0.1070
```

Plot the cumulative distribution function estimate using a smaller bandwidth.

```
[f,xi] = ksdensity(hospital.Weight,pts,'Support','positive',...  
    'Function','cdf','Bandwidth',0.05);  
plot(xi,f,'--r','LineWidth',2)  
legend('empirical cdf','kernel-bw:default','kernel-bw:0.05',...  
    'Location','northwest')  
hold off
```



The `ksdensity` estimate with a smaller bandwidth matches the empirical cumulative distribution function better.

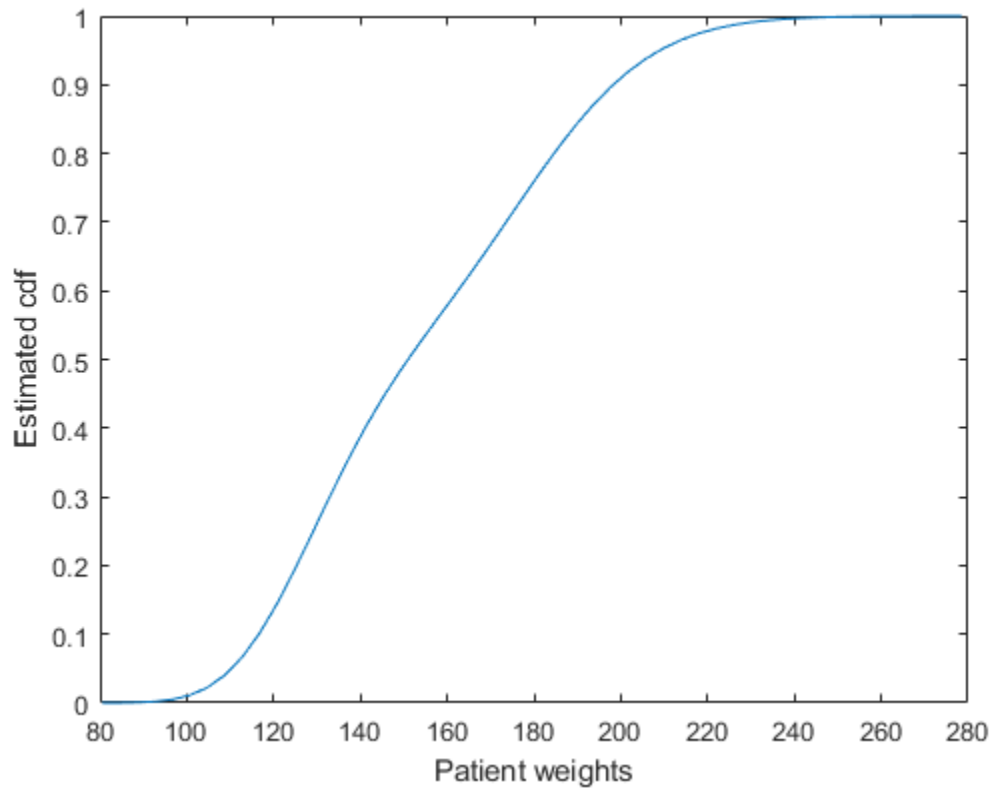
### Plot Estimated Cumulative Density Function for Given Number of Points

Load the sample data.

```
load hospital
```

Plot the estimated cdf evaluated at 50 equally spaced points.

```
figure()  
ksdensity(hospital.Weight, 'Support', 'positive', 'Function', 'cdf', ...  
          'NumPoints', 50)  
xlabel('Patient weights')  
ylabel('Estimated cdf')
```



**Estimate Survivor and Cumulative Hazard for Censored Failure Data**

Generate sample data from an exponential distribution with mean 3.

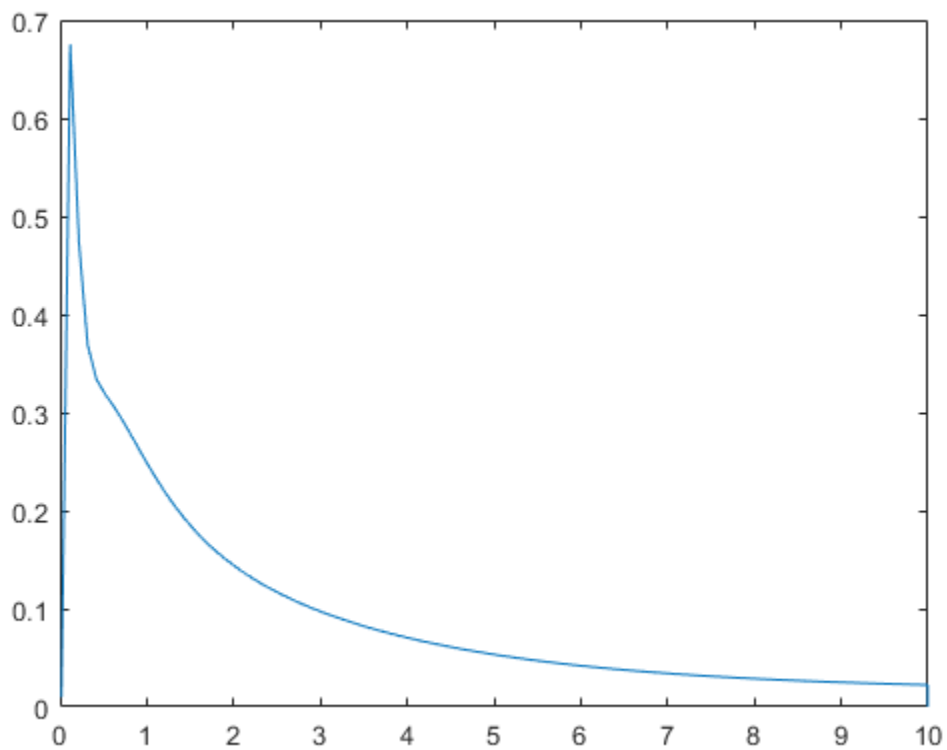
```
rng('default') % For reproducibility
x = random('exp', 3, 100, 1);
```

Create a logical vector that indicates censoring. Here, observations with lifetimes longer than 10 are censored.

```
T = 10;
cens = (x>T);
```

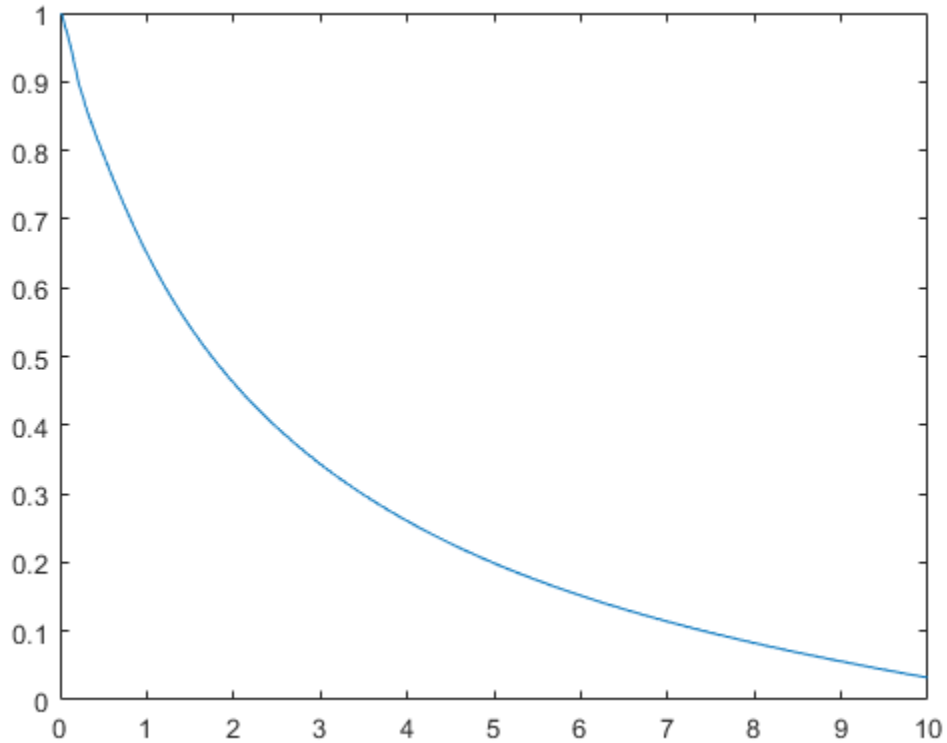
Compute and plot the estimated density function.

```
figure
ksdensity(x, 'Support', 'positive', 'Censoring', cens);
```



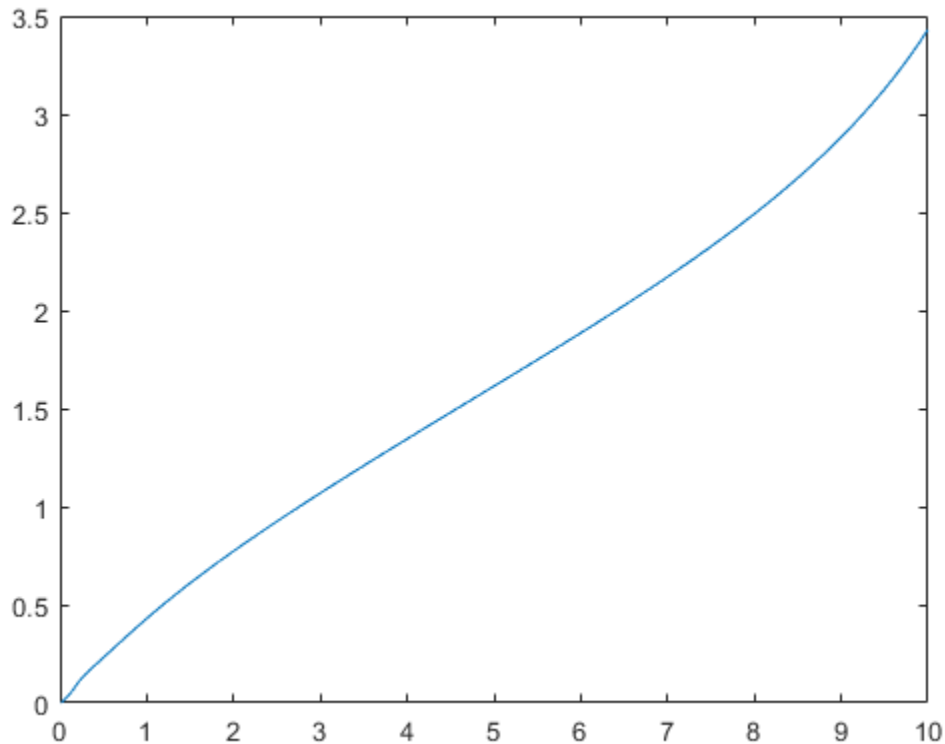
Compute and plot the survivor function.

```
figure
ksdensity(x, 'Support', 'positive', 'Censoring', cens, ...
'Function', 'survivor');
```



Compute and plot the cumulative hazard function.

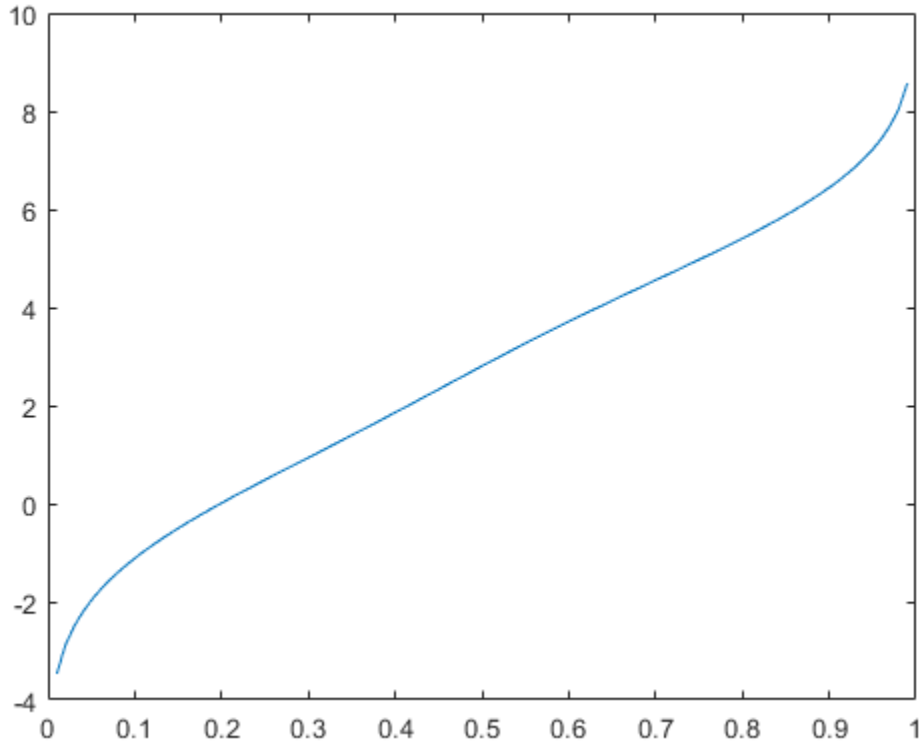
```
figure
ksdensity(x, 'Support', 'positive', 'Censoring', cens, ...
'Function', 'cumhazard');
```



### Estimate Inverse Cumulative Distribution Function for Specified Probability Values

Generate a mixture of two normal distributions, and plot the estimated inverse cumulative distribution function at a specified set of probability values.

```
rng('default') % For reproducibility
x = [randn(30,1); 5+randn(30,1)];
pi = linspace(.01, .99, 99);
figure
ksdensity(x,pi,'Function','icdf');
```



### Return Bandwidth of Smoothing Window

Generate a mixture of two normal distributions.

```
rng('default') % For reproducibility
x = [randn(30,1); 5+randn(30,1)];
```

Return the bandwidth of the smoothing window for the probability density estimate.

```
[f,xi,bw] = ksdensity(x);
bw
```

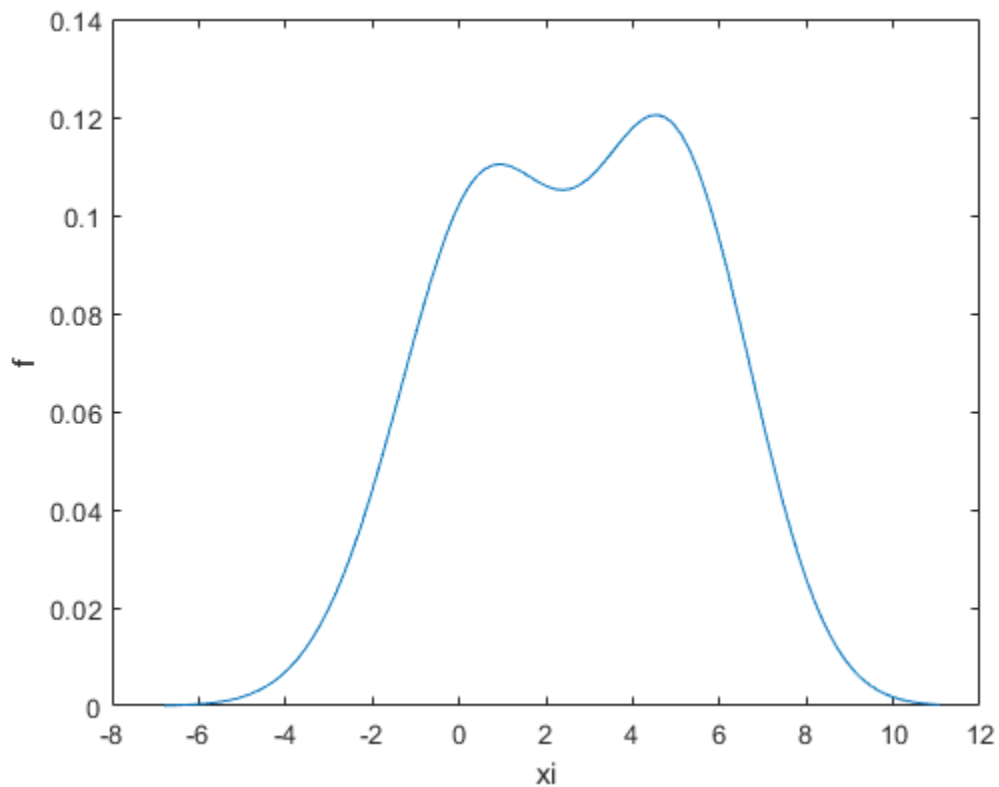


```
bw =  
  
1.5141
```

The default bandwidth is optimal for normal densities.

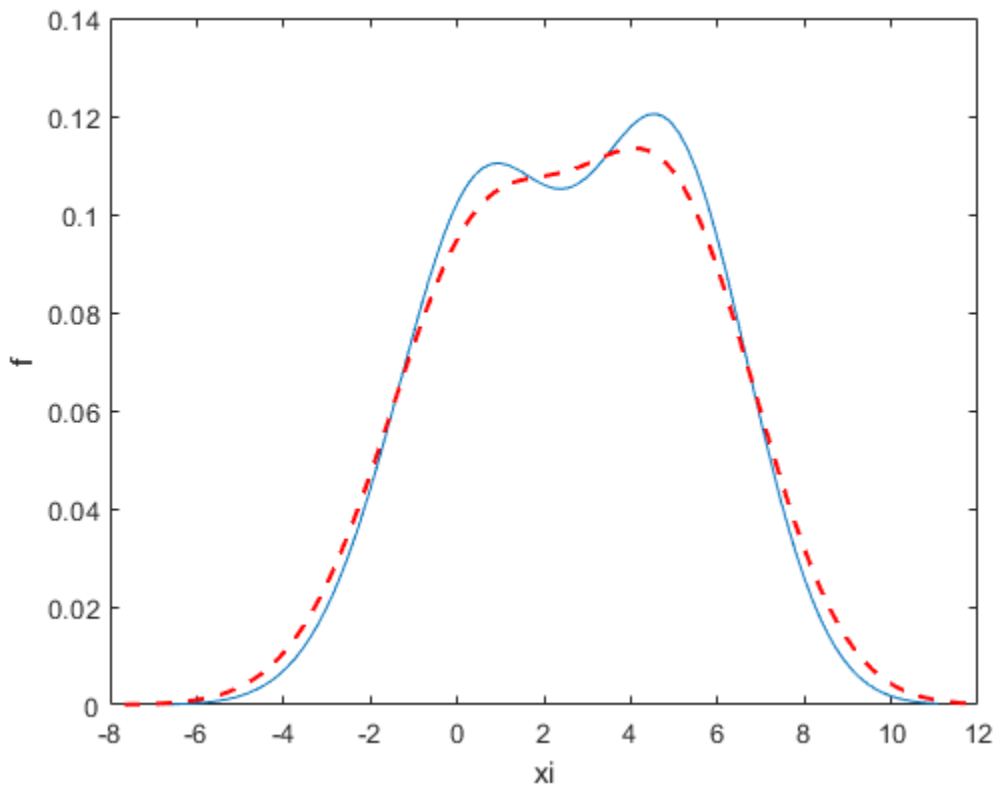
Plot the estimated density.

```
figure  
plot(xi,f);  
xlabel('xi')  
ylabel('f')  
hold on
```



Plot the density using an increased bandwidth value.

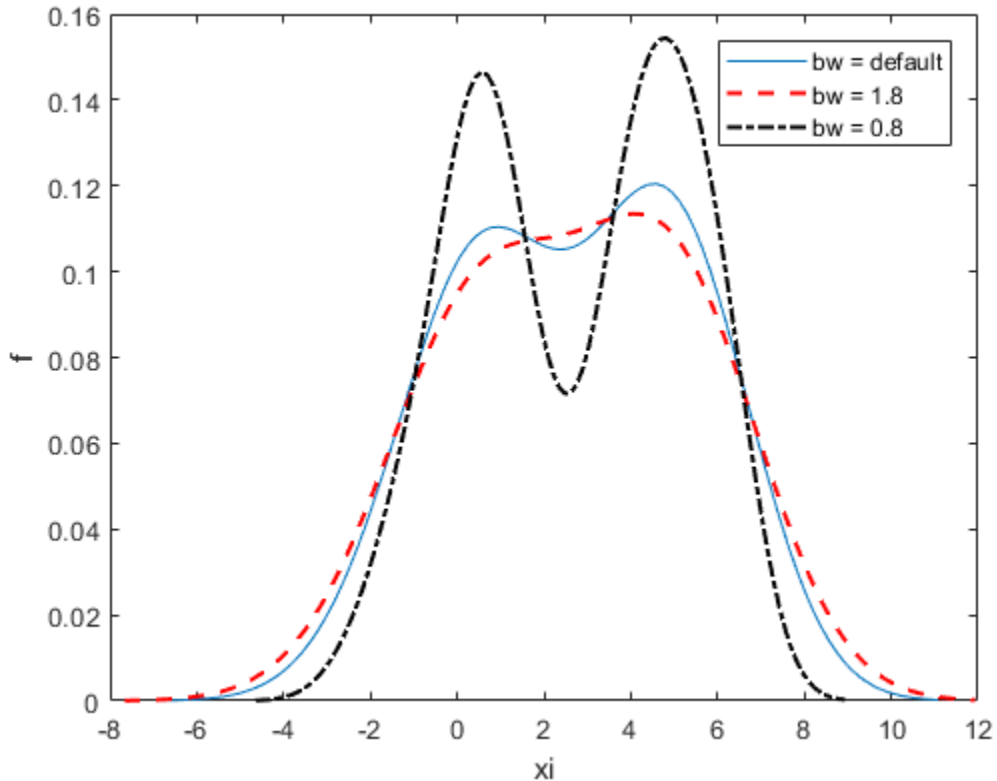
```
[f,xi] = ksdensity(x,'Bandwidth',1.8);  
plot(xi,f,'--r','LineWidth',1.5)
```



A higher bandwidth further smooths the density estimate, which might mask some characteristics of the distribution.

Now, plot the density using a decreased bandwidth value.

```
[f,xi] = ksdensity(x,'Bandwidth',0.8);  
plot(xi,f,'-.k','LineWidth',1.5)  
legend('bw = default','bw = 1.8','bw = 0.8')  
hold off
```



A smaller bandwidth smooths the density estimate less, which exaggerates some characteristics of the sample.

### Plot Kernel Density Estimate of Bivariate Data

Create a two-column vector of points at which to evaluate the density.

```
gridx1 = -0.25:.05:1.25;  
gridx2 = 0:.1:15;  
[x1,x2] = meshgrid(gridx1, gridx2);  
x1 = x1(:);
```

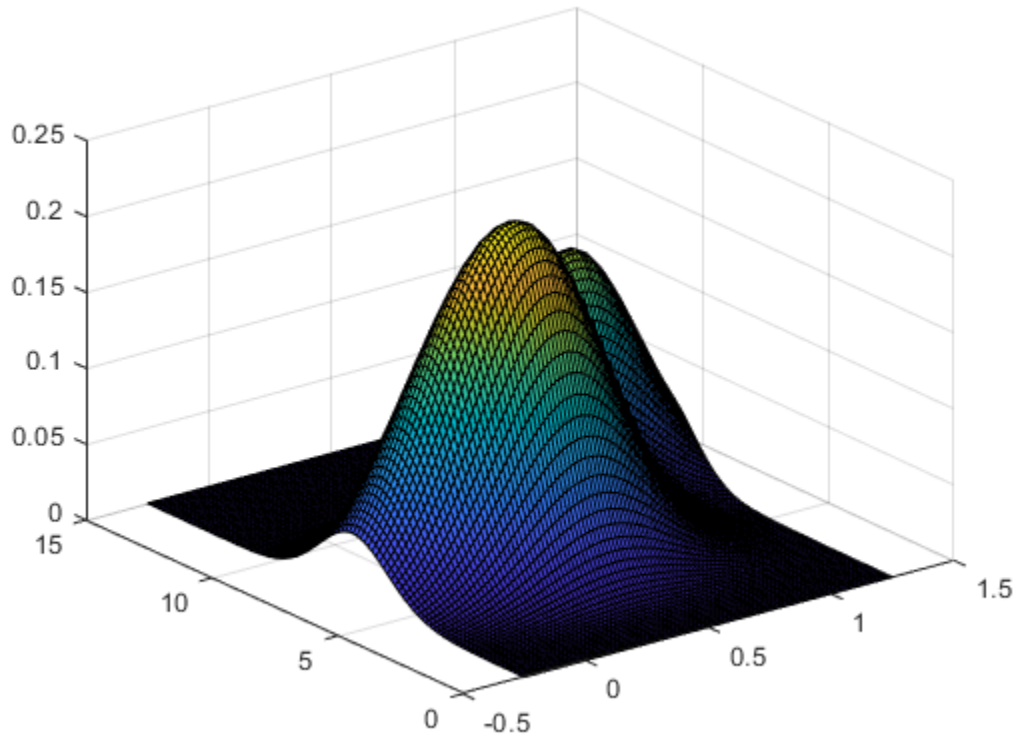
```
x2 = x2(:);  
xi = [x1 x2];
```

Generate a 30-by-2 matrix containing random numbers from a mixture of bivariate normal distributions.

```
rng('default') % For reproducibility  
x = [0+.5*rand(20,1) 5+2.5*rand(20,1);  
     .75+.25*rand(10,1) 8.75+1.25*rand(10,1)];
```

Plot the estimated density of the sample data.

```
figure  
ksdensity(x,xi);
```



- “Fit Kernel Distribution Using `ksdensity`” on page 5-51
- “Fit Distributions to Grouped Data Using `ksdensity`” on page 5-54

## Input Arguments

### **x** — Sample data

column vector | two-column matrix

Sample data for which `ksdensity` returns `f` values, specified as a column vector or two-column matrix. Use a column vector for univariate data, and a two-column matrix for bivariate data.

Example: `[f,xi] = ksdensity(x)`

Data Types: `single` | `double`

**pts — Points to evaluate  $f$**

vector | two-column matrix

Points to evaluate  $f$  at, specified as a vector or two-column matrix. For univariate data, `pts` can be a row or column vector. The returned output `f` has the same dimensions as `pts`.

Example: `pts = (0:1:25); ksdensity(x,pts);`

Data Types: `single` | `double`

**ax — Axes handle**

handle

Axes handle for the figure `ksdensity` plots to, specified as a handle.

For example, if `h` is a handle for a figure, then `ksdensity` can plot to that figure as follows.

Example: `ksdensity(h,x)`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'Censoring', cens, 'Kernel', 'triangle', 'NumPoints', 20, 'Function', 'cdf'` specifies that `ksdensity` estimates the cdf by evaluating at 20 equally spaced points that covers the range of data, using the triangle kernel smoothing function and accounting for the censored data information in vector `cens`.

**Bandwidth — Bandwidth of the kernel smoothing window**

optimal value for normal densities (default) | scalar value | two-element vector

The bandwidth of the kernel-smoothing window, which is a function of the number of points in `x`, specified as the comma-separated pair consisting of `'Bandwidth'` and a

scalar value. If the sample data is bivariate, Bandwidth can also be a two-element vector. The default is optimal for estimating normal densities, but you might want to choose a larger or smaller value to smooth more or less.

If you specify 'BoundaryCorrection' as 'log'(default) and 'Support' as either 'positive' or a vector [L U], ksdensity converts bounded data to be unbounded by using log transformation. The value of 'Bandwidth' is on the scale of the transformed values.

Example: 'Bandwidth',0.8

Data Types: single | double

### BoundaryCorrection — Boundary correction method

'log'(default) | 'reflection'

Boundary correction method, specified as the comma-separated pair consisting of 'BoundaryCorrection' and 'log' or 'reflection'.

Value	Description
'log'	<p>ksdensity converts bounded data <math>x</math> to be unbounded by one of the following transformations. Then, it transforms back to the original bounded scale after density estimation.</p> <ul style="list-style-type: none"> <li>For univariate data, if you specify 'Support', 'positive', then ksdensity applies <math>\log(x)</math>.</li> <li>For univariate data, if you specify 'Support', [L U], where L and U are numeric scalars and <math>L &lt; U</math>, then ksdensity applies <math>\log((x-L)/(U-x))</math>.</li> <li>For bivariate data, ksdensity transforms each column of <math>x</math> in the same way with the univariate data.</li> </ul> <p>The value of 'Bandwidth' and the bw output are on the scale of the transformed values.</p>
'reflection'	<p>ksdensity augments bounded data by adding reflected data near the boundaries, then it returns estimates corresponding to the original support. For details, see “Reflection Method” on page 32-3470.</p>

ksdensity applies boundary correction only when you specify 'Support' as a value other than 'unbounded'.

Example: 'BoundaryCorrection', 'reflection'

Data Types: char

### **Censoring** — Logical vector

vector of 0s (default) | vector of 0s and 1s

Logical vector indicating which entries are censored, specified as the comma-separated pair consisting of 'Censoring' and a vector of binary values. A value of 0 indicates there is no censoring, 1 indicates that observation is censored. Default is there is no censoring. This name-value pair is only valid for univariate data.

Example: 'Censoring', censdata

Data Types: logical

### **Function** — Function to estimate

'pdf' (default) | 'cdf' | 'icdf' | 'survivor' | 'cumhazard'

Function to estimate, specified as the comma-separated pair consisting of 'Function' and one of the following.

Value	Description
'pdf'	Probability density function.
'cdf'	Cumulative distribution function.
'icdf'	Inverse cumulative distribution function. <code>ksdensity</code> computes the estimated inverse cdf of the values in <code>x</code> , and evaluates it at the probability values specified in <code>pi</code> .  This value is valid only for univariate data.
'survivor'	Survivor function.
'cumhazard'	Cumulative hazard function.  This value is valid only for univariate data.

Example: 'Function', 'icdf'

Data Types: char

### **kernel** — Type of kernel smoother

'normal' (default) | 'box' | 'triangle' | 'epanechnikov' | function handle | character vector



Type of kernel smoother, specified as the comma-separated pair consisting of 'Kernel' and one of the following.

- 'normal' (default)
- 'box'
- 'triangle'
- 'epanechnikov'
- You can also specify a custom kernel function, as a function handle or as a character vector, e.g., @normpdf or 'normpdf'. This calls the function with one argument that is an array of distances between data values and locations where the density is evaluated. The function must return an array of the same size containing corresponding values of the kernel function.

When 'Function' is 'pdf', this kernel function returns density values. Otherwise, it returns cumulative probability values.

Specifying a custom kernel when 'Function' is 'icdf' returns an error.

For bivariate data, `ksdensity` applies the same kernel to each dimension.

Example: 'Kernel', 'box'

Data Types: char | function\_handle

#### **NumPoints** — Number of equally spaced points

100 (default) | scalar value

Number of equally spaced points in  $x_i$ , specified as the comma-separated pair consisting of 'NumPoints' and a scalar value. This name-value pair is only valid for univariate data.

For example, for a kernel smooth estimate of a specified function at 80 equally spaced points within the range of sample data, input:

Example: 'NumPoints', 80

Data Types: single | double

#### **Support** — Support for the density

'unbounded' (default) | 'positive' | two-element vector, [L U] | two-by-two matrix, [L1 L2 ; U1 U2]

Support for the density, specified as the comma-separated pair consisting of 'support' and one of the following.

Value	Description
'unbounded'	Default. Allow the density to extend over the whole real line.
'positive'	Restrict the density to positive values.
Two-element vector, [L U]	Give the finite lower and upper bounds for the support of the density. This option is only valid for univariate sample data.
Two-by-two matrix, [L1 L2 ; U1 U2]	Give the finite lower and upper bounds for the support of the density. The first row contains the lower limits and the second row contains the upper limits. This option is only valid for bivariate sample data.

For bivariate data, 'Support' can be a combination of positive, unbounded, or bounded variables specified as [0 -Inf ; Inf Inf] or [0 L ; Inf U].

Example: 'Support', 'positive'

Example: 'Support', [0 10]

Data Types: single | double | char

### PlotFcn — Function used to create kernel density plot

'surf' (default) | 'contour' | 'plot3' | 'surfc'

Function used to create kernel density plot, specified as the comma-separated pair consisting of 'PlotFcn' and one of the following.

Value	Description
'surf'	3-D shaded surface plot, created using surf
'contour'	Contour plot, created using contour
'plot3'	3-D line plot, created using plot3
'surfc'	Contour plot under a 3-D shaded surface plot, created using surfc

This name-value pair is only valid for bivariate sample data.

Example: 'PlotFcn', 'contour'

### weights — Weights for sample data

vector

Weights for sample data, specified as the comma-separated pair consisting of 'Weights' and a vector of length `size(x,1)`, where `x` is the sample data.

Example: 'Weights', `xw`

Data Types: `single` | `double`

## Output Arguments

### **`f`** — Estimated function values

vector

Estimated function values, returned as a vector of the same dimension as `xi` or `pts`.

### **`xi`** — Evaluation points

100 equally spaced points | 900 equally spaced points | vector

Evaluation points at which `ksdensity` calculates `f`, returned as a vector. For univariate data, the default is 100 equally-spaced points that cover the range of data in `x`. For bivariate data, the default is 900 equally-spaced points created using `meshgrid` from 30 equally-spaced points in each dimension.

### **`bw`** — Bandwidth of smoothing window

scalar value

Bandwidth of smoothing window, returned as a scalar value.

If you specify 'BoundaryCorrection' as 'log'(default) and 'Support' as either 'positive' or a vector [`L` `U`], `ksdensity` converts bounded data to be unbounded by using log transformation. The value of `bw` is on the scale of the transformed values.

## Definitions

### Kernel Distribution

A kernel distribution is a nonparametric representation of the probability density function (pdf) of a random variable. You can use a kernel distribution when a parametric distribution cannot properly describe the data, or when you want to avoid making

assumptions about the distribution of the data. A kernel distribution is defined by a smoothing function and a bandwidth value, which control the smoothness of the resulting density curve.

The kernel density estimator is the estimated pdf of a random variable. For any real values of  $x$ , the kernel density estimator's formula is given by

$$\hat{f}_h(x) = \frac{1}{nh} \sum_{i=1}^n K\left(\frac{x - x_i}{h}\right),$$

where  $x_1, x_2, \dots, x_n$  are random samples from an unknown distribution,  $n$  is the sample size,  $K(\bullet)$  is the kernel smoothing function, and  $h$  is the bandwidth.

The kernel estimator for the cumulative distribution function (cdf), for any real values of  $x$ , is given by

$$\hat{F}_h(x) = \int_{-\infty}^x \hat{f}_h(t) dt = \frac{1}{n} \sum_{i=1}^n G\left(\frac{x - x_i}{h}\right),$$

where  $G(x) = \int_{-\infty}^x K(t) dt$ .

For more details, see “Kernel Distribution” on page B-86.

## Reflection Method

The reflection method is a boundary correction method that accurately finds kernel density estimators when a random variable has bounded support. If you specify 'BoundaryCorrection', 'reflection', `ksdensity` uses the reflection method. This method augments bounded data by adding reflected data near the boundaries, and estimates the pdf. Then, `ksdensity` returns the estimated pdf corresponding to the original support with proper normalization, so that the estimated pdf's integral over the original support is equal to one.

If you additionally specify 'Support', [L U], then `ksdensity` finds the kernel estimator as follows.

- If 'Function' is 'pdf', then the kernel density estimator is

$$\hat{f}_h(x) = \frac{1}{nh} \sum_{i=1}^n \left[ K\left(\frac{x-x_i^-}{h}\right) + K\left(\frac{x-x_i}{h}\right) + K\left(\frac{x-x_i^+}{h}\right) \right] \text{ for } L \leq x \leq U,$$

where  $x_i^- = 2L - x_i$ ,  $x_i^+ = 2U - x_i$ , and  $x_i$  is the  $i$ th sample data.

- If 'Function' is 'cdf', then the kernel estimator for cdf is

$$\hat{F}_h(x) = \frac{1}{n} \sum_{i=1}^n \left[ G\left(\frac{x-x_i^-}{h}\right) + G\left(\frac{x-x_i}{h}\right) + G\left(\frac{x-x_i^+}{h}\right) \right] - \frac{1}{n} \sum_{i=1}^n \left[ G\left(\frac{L-x_i^-}{h}\right) + G\left(\frac{L-x_i}{h}\right) + G\left(\frac{L-x_i^+}{h}\right) \right]$$

- To obtain a kernel estimator for an inverse cdf, a survivor function, or a cumulative hazard function (when 'Function' is 'icdf', 'survivor', or 'cumhazrd'),

ksdensity uses both  $\hat{f}_h(x)$  and  $\hat{F}_h(x)$ .

If you additionally specify 'Support' as 'positive' or [0 inf], then ksdensity finds the kernel estimator by replacing [L U] with [0 inf] in the above equations.

## References

- [1] Bowman, A. W., and A. Azzalini. *Applied Smoothing Techniques for Data Analysis*. New York: Oxford University Press Inc., 1997.
- [2] Hill, P. D. "Kernel estimation of a distribution function." *Communications in Statistics - Theory and Methods*. Vol 14, Issue. 3, 1985, pp. 605-620.
- [3] Jones, M. C. "Simple boundary correction for kernel density estimation." *Statistics and Computing*. Vol. 3, Issue 3, 1993, pp. 135-146.
- [4] Silverman, B. W. *Density Estimation for Statistics and Data Analysis*. Chapman & Hall/CRC, 1986.

## Extended Capabilities

### Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function supports tall arrays for out-of-memory data with some limitations.

- Some options that require extra passes or sorting of the input data are not supported:
  - 'BoundaryCorrection'
  - 'Censoring'
  - 'Support' (support is always unbounded).
- Uses standard deviation (instead of median absolute deviation) to compute the bandwidth.

For more information, see “Tall Arrays” (MATLAB).

### See Also

histogram | mvksdensity

### Topics

“Fit Kernel Distribution Using ksdensity” on page 5-51

“Fit Distributions to Grouped Data Using ksdensity” on page 5-54

“Working with Probability Distributions” on page 5-3

“Nonparametric and Empirical Probability Distributions” on page 5-37

“Supported Distributions” on page 5-17

**Introduced before R2006a**

## kstest

One-sample Kolmogorov-Smirnov test

### Syntax

```
h = kstest(x)
h = kstest(x,Name,Value)
[h,p] = kstest(____)
[h,p,ksstat,cv] = kstest(____)
```

### Description

`h = kstest(x)` returns a test decision for the null hypothesis that the data in vector `x` comes from a standard normal distribution, against the alternative that it does not come from such a distribution, using the one-sample Kolmogorov-Smirnov test on page 32-3482. The result `h` is 1 if the test rejects the null hypothesis at the 5% significance level, or 0 otherwise.

`h = kstest(x,Name,Value)` returns a test decision for the one-sample Kolmogorov-Smirnov test with additional options specified by one or more name-value pair arguments. For example, you can test for a distribution other than standard normal, change the significance level, or conduct a one-sided test.

`[h,p] = kstest(____)` also returns the  $p$ -value `p` of the hypothesis test, using any of the input arguments from the previous syntaxes.

`[h,p,ksstat,cv] = kstest(____)` also returns the value of the test statistic `ksstat` and the approximate critical value `cv` of the test.

### Examples

### Test for a Standard Normal Distribution

Load the sample data. Create a vector containing the first column of the students' exam grades data.

```
load examgrades;  
test1 = grades(:,1);
```

Test the null hypothesis that the data comes from a normal distribution with a mean of 75 and a standard deviation of 10. Use these parameters to center and scale each element of the data vector since, by default, `kstest` tests for a standard normal distribution.

```
x = (test1-75)/10;  
h = kstest(x)
```

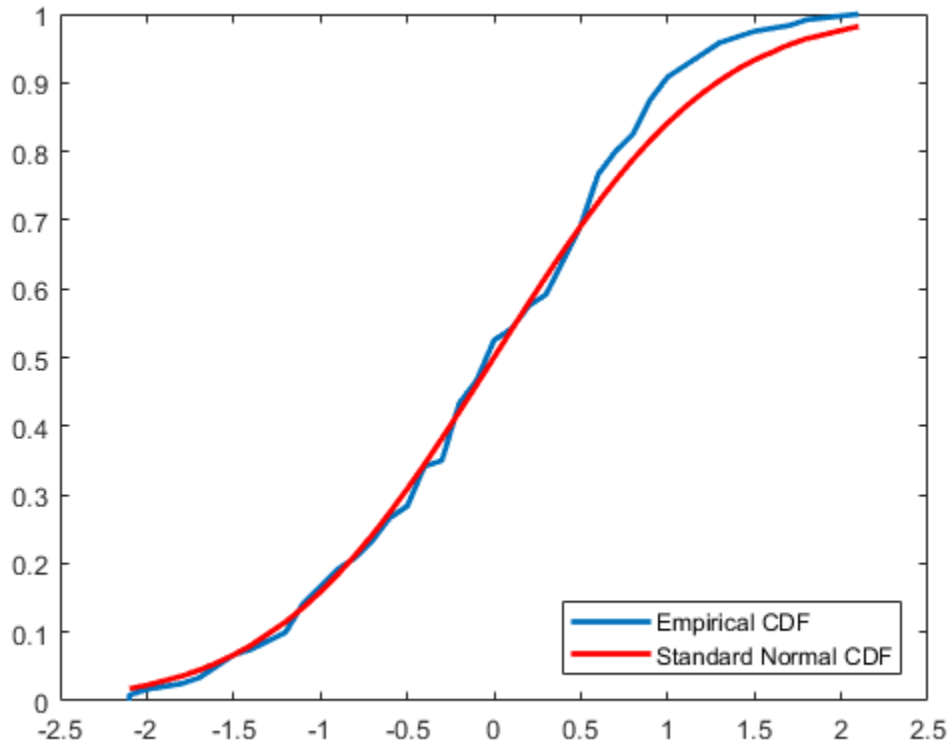
```
h =  
  
    logical  
  
    0
```

The returned value of `h = 0` indicates that `kstest` fails to reject the null hypothesis at the default 5% significance level.

Plot the empirical cumulative distribution function (cdf) and the standard normal cdf for a visual comparison.

```
[f,x_values] = ecdf(x);  
F = plot(x_values,f);  
set(F,'LineWidth',2);  
hold on;  
G = plot(x_values,normcdf(x_values,0,1),'r-');  
set(G,'LineWidth',2);  
legend([F G],...  
       'Empirical CDF','Standard Normal CDF',...  
       'Location','SE');
```





The plot shows the similarity between the empirical cdf of the centered and scaled data vector and the cdf of the standard normal distribution.

### Specify the Hypothesized Distribution Using a Two-Column Matrix

Load the sample data. Create a vector containing the first column of the students' exam grades data.

```
load examgrades;  
x = grades(:,1);
```

Specify the hypothesized distribution as a two-column matrix. Column 1 contains the data vector  $x$ . Column 2 contains cdf values evaluated at each value in  $x$  for a hypothesized Student's  $t$  distribution with a location parameter of 75, a scale parameter of 10, and one degree of freedom.

```
test_cdf = [x, cdf('tlocationscale', x, 75, 10, 1)];
```

Test if the data are from the hypothesized distribution.

```
h = kstest(x, 'CDF', test_cdf)
```

```
h =  
  
    logical  
  
     1
```

The returned value of  $h = 1$  indicates that `kstest` rejects the null hypothesis at the default 5% significance level.

### Specify the Hypothesized Distribution Using a Probability Distribution Object

Load the sample data. Create a vector containing the first column of the students' exam grades data.

```
load examgrades;  
x = grades(:,1);
```

Create a probability distribution object to test if the data comes from a Student's  $t$  distribution with a location parameter of 75, a scale parameter of 10, and one degree of freedom.

```
test_cdf = makedist('tlocationscale', 'mu', 75, 'sigma', 10, 'nu', 1);
```

Test the null hypothesis that the data comes from the hypothesized distribution.

```
h = kstest(x, 'CDF', test_cdf)
```

```
h =
```

```
logical
1
```

The returned value of  $h = 1$  indicates that `kstest` rejects the null hypothesis at the default 5% significance level.

### Test the Hypothesis at Different Significance Levels

Load the sample data. Create a vector containing the first column of the students' exam grades.

```
load examgrades;
x = grades(:,1);
```

Create a probability distribution object to test if the data comes from a Student's  $t$  distribution with a location parameter of 75, a scale parameter of 10, and one degree of freedom.

```
test_cdf = makedist('tlocation_scale','mu',75,'sigma',10,'nu',1);
```

Test the null hypothesis that data comes from the hypothesized distribution at the 1% significance level.

```
[h,p] = kstest(x,'CDF',test_cdf,'Alpha',0.01)
```

```
h =
logical
1

p =
0.0021
```

The returned value of  $h = 1$  indicates that `kstest` rejects the null hypothesis at the 1% significance level.

### Conduct a One-Sided Hypothesis Test

Load the sample data. Create a vector containing the third column of the stock return data matrix.

```
load stockreturns;  
x = stocks(:,3);
```

Test the null hypothesis that the data comes from a standard normal distribution, against the alternative hypothesis that the population cdf of the data is larger than the standard normal cdf.

```
[h,p,k,c] = kstest(x, 'Tail', 'larger')
```

```
h =
```

```
logical
```

```
1
```

```
p =
```

```
5.0854e-05
```

```
k =
```

```
0.2197
```

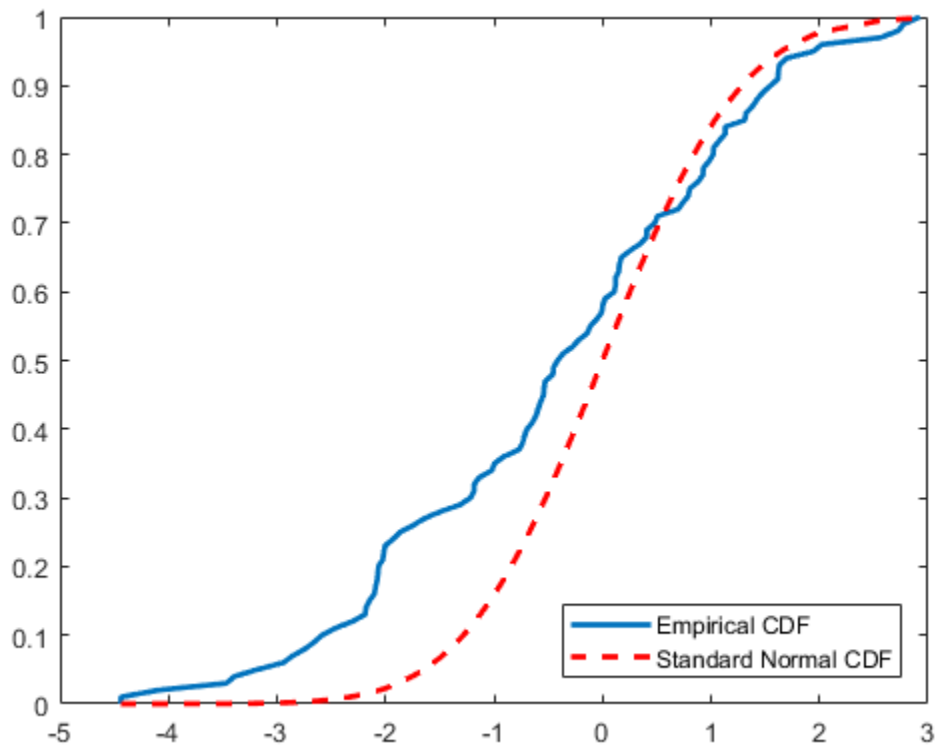
```
c =
```

```
0.1207
```

The returned value of  $h = 1$  indicates that `kstest` rejects the null hypothesis in favor of the alternative hypothesis at the default 5% significance level.

Plot the empirical cdf and the standard normal cdf for a visual comparison.

```
[f,x_values] = ecdf(x);  
J = plot(x_values,f);  
hold on;  
K = plot(x_values,normcdf(x_values),'r--');  
set(J,'LineWidth',2);  
set(K,'LineWidth',2);  
legend([J K], 'Empirical CDF', 'Standard Normal CDF', 'Location', 'SE');
```



The plot shows the difference between the empirical cdf of the data vector `x` and the cdf of the standard normal distribution.

## Input Arguments

### **x** — Sample data

vector

Sample data, specified as a vector.

Data Types: `single` | `double`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'Tail', 'larger', 'Alpha', 0.01` specifies a test using the alternative hypothesis that the cdf of the population from which the sample data is drawn is greater than the cdf of the hypothesized distribution, conducted at the 1% significance level.

### **Alpha** — Significance level

0.05 (default) | scalar value in the range (0,1)

Significance level of the hypothesis test, specified as the comma-separated pair consisting of `'Alpha'` and a scalar value in the range (0,1).

Example: `'Alpha', 0.01`

Data Types: `single` | `double`

### **CDF** — cdf of hypothesized continuous distribution

matrix | probability distribution object

cdf of hypothesized continuous distribution, specified the comma-separated pair consisting of `'CDF'` and either a two-column matrix or a continuous probability distribution object. When `CDF` is a matrix, column 1 contains a set of possible  $x$  values, and column 2 contains the corresponding hypothesized cumulative distribution function

values  $G(x)$ . The calculation is most efficient if CDF is specified such that column 1 contains the values in the data vector  $x$ . If there are values in  $x$  not found in column 1 of CDF, `kstest` approximates  $G(x)$  by interpolation. All values in  $x$  must lie in the interval between the smallest and largest values in the first column of CDF. By default, `kstest` tests for a standard normal distribution.

The one-sample Kolmogorov-Smirnov test on page 32-3482 is only valid for continuous cumulative distribution functions, and requires CDF to be predetermined. The result is not accurate if CDF is estimated from the data. To test  $x$  against the normal, lognormal, extreme value, Weibull, or exponential distribution without specifying distribution parameters, use `lillietest` instead.

Data Types: `single` | `double`

#### **Tail** — Type of alternative hypothesis

`'unequal'` (default) | `'larger'` | `'smaller'`

Type of alternative hypothesis to evaluate, specified as the comma-separated pair consisting of `'Tail'` and one of the following.

<code>'unequal'</code>	Test the alternative hypothesis that the cdf of the population from which $x$ is drawn is not equal to the cdf of the hypothesized distribution.
<code>'larger'</code>	Test the alternative hypothesis that the cdf of the population from which $x$ is drawn is greater than the cdf of the hypothesized distribution.
<code>'smaller'</code>	Test the alternative hypothesis that the cdf of the population from which $x$ is drawn is less than the cdf of the hypothesized distribution.

If the values in the data vector  $x$  tend to be larger than expected from the hypothesized distribution, the empirical distribution function of  $x$  tends to be smaller, and vice versa.

Example: `'Tail', 'larger'`

## Output Arguments

### **h** — Hypothesis test result

1 | 0

Hypothesis test result, returned as a logical value.

- If  $h = 1$ , this indicates the rejection of the null hypothesis at the Alpha significance level.
- If  $h = 0$ , this indicates a failure to reject the null hypothesis at the Alpha significance level.

**p — p-value**

scalar value in the range [0,1]

$p$ -value of the test, returned as a scalar value in the range [0,1].  $p$  is the probability of observing a test statistic as extreme as, or more extreme than, the observed value under the null hypothesis. Small values of  $p$  cast doubt on the validity of the null hypothesis.

**ksstat — Test statistic**

nonnegative scalar value

Test statistic of the hypothesis test, returned as a nonnegative scalar value.

**cv — Critical value**

nonnegative scalar value

Critical value, returned as a nonnegative scalar value.

## Definitions

### One-Sample Kolmogorov-Smirnov Test

The one-sample Kolmogorov-Smirnov test is a nonparametric test of the null hypothesis that the population cdf of the data is equal to the hypothesized cdf.

The two-sided test for “unequal” cdf functions tests the null hypothesis against the alternative that the population cdf of the data is not equal to the hypothesized cdf. The test statistic is the maximum absolute difference between the empirical cdf calculated from  $x$  and the hypothesized cdf:

$$D^* = \max_x \left( \left| \hat{F}(x) - G(x) \right| \right),$$

where  $\hat{F}(x)$  is the empirical cdf and  $G(x)$  is the cdf of the hypothesized distribution.



The one-sided test for a “larger” cdf function tests the null hypothesis against the alternative that the population cdf of the data is greater than the hypothesized cdf. The test statistic is the maximum amount by which the empirical cdf calculated from  $x$  exceeds the hypothesized cdf:

$$D^* = \max_x \left( \hat{F}(x) - G(x) \right).$$

The one-sided test for a “smaller” cdf function tests the null hypothesis against the alternative that the population cdf of the data is less than the hypothesized cdf. The test statistic is the maximum amount by which the hypothesized cdf exceeds the empirical cdf calculated from  $x$ :

$$D^* = \max_x \left( G(x) - \hat{F}(x) \right).$$

`kstest` computes the critical value `cv` using an approximate formula or by interpolation in a table. The formula and table cover the range  $0.01 \leq \alpha \leq 0.2$  for two-sided tests and  $0.005 \leq \alpha \leq 0.1$  for one-sided tests. `cv` is returned as `NaN` if `alpha` is outside this range.

## Algorithms

`kstest` decides to reject the null hypothesis by comparing the  $p$ -value `p` with the significance level `Alpha`, not by comparing the test statistic `ksstat` with the critical value `cv`. Since `cv` is approximate, comparing `ksstat` with `cv` occasionally leads to a different conclusion than comparing `p` with `Alpha`.

## References

- [1] Massey, F. J. “The Kolmogorov-Smirnov Test for Goodness of Fit.” *Journal of the American Statistical Association*. Vol. 46, No. 253, 1951, pp. 68–78.
- [2] Miller, L. H. “Table of Percentage Points of Kolmogorov Statistics.” *Journal of the American Statistical Association*. Vol. 51, No. 273, 1956, pp. 111–121.
- [3] Marsaglia, G., W. Tsang, and J. Wang. “Evaluating Kolmogorov’s Distribution.” *Journal of Statistical Software*. Vol. 8, Issue 18, 2003.

## See Also

`adtest` | `kstest2` | `lillietest`

**Introduced before R2006a**

## kstest2

Two-sample Kolmogorov-Smirnov test

### Syntax

```
h = kstest2(x1,x2)
h = kstest2(x1,x2,Name,Value)
[h,p] = kstest2(____)
[h,p,ks2stat] = kstest2(____)
```

### Description

`h = kstest2(x1,x2)` returns a test decision for the null hypothesis that the data in vectors `x1` and `x2` are from the same continuous distribution, using the two-sample Kolmogorov-Smirnov test on page 32-3490. The alternative hypothesis is that `x1` and `x2` are from different continuous distributions. The result `h` is 1 if the test rejects the null hypothesis at the 5% significance level, and 0 otherwise.

`h = kstest2(x1,x2,Name,Value)` returns a test decision for a two-sample Kolmogorov-Smirnov test with additional options specified by one or more name-value pair arguments. For example, you can change the significance level or conduct a one-sided test.

`[h,p] = kstest2(____)` also returns the asymptotic  $p$ -value `p`, using any of the input arguments from the previous syntaxes.

`[h,p,ks2stat] = kstest2(____)` also returns the test statistic `ks2stat`.

### Examples

#### Test Two Samples for the Same Distribution

Generate sample data from two different Weibull distributions.

```
rng(1);      % For reproducibility
x1 = wblrnd(1,1,1,50);
x2 = wblrnd(1.2,2,1,50);
```

Test the null hypothesis that data in vectors `x1` and `x2` comes from populations with the same distribution.

```
h = kstest2(x1,x2)
```

```
h =
    logical
     1
```

The returned value of `h = 1` indicates that `kstest` rejects the null hypothesis at the default 5% significance level.

### Test the Hypothesis at Different Significance Levels

Generate sample data from two different Weibull distributions.

```
rng(1);      % For reproducibility
x1 = wblrnd(1,1,1,50);
x2 = wblrnd(1.2,2,1,50);
```

Test the null hypothesis that data vectors `x1` and `x2` are from populations with the same distribution at the 1% significance level.

```
[h,p] = kstest2(x1,x2,'Alpha',0.01)
```

```
h =
    logical
     0
```

```
p =
```

```
0.0317
```

The returned value of  $h = 0$  indicates that `kstest` does not reject the null hypothesis at the 1% significance level.

### One-Sided Hypothesis Test

Generate sample data from two different Weibull distributions.

```
rng(1);      % For reproducibility
x1 = wblrnd(1,1,1,50);
x2 = wblrnd(1.2,2,1,50);
```

Test the null hypothesis that data in vectors `x1` and `x2` comes from populations with the same distribution, against the alternative hypothesis that the cdf of the distribution of `x1` is larger than the cdf of the distribution of `x2`.

```
[h,p,k] = kstest2(x1,x2,'Tail','larger')
```

```
h =
```

```
logical
```

```
1
```

```
p =
```

```
0.0158
```

```
k =
```

```
0.2800
```

The returned value of  $h = 1$  indicates that `kstest` rejects the null hypothesis, in favor of the alternative hypothesis that the cdf of the distribution of `x1` is larger than the cdf of

the distribution of  $x_2$ , at the default 5% significance level. The returned value of  $k$  is the test statistic for the two-sample Kolmogorov-Smirnov test.

## Input Arguments

### **$x_1$ — Sample data**

vector

Sample data from the first sample, specified as a vector. Data vectors  $x_1$  and  $x_2$  do not need to be the same size.

Data Types: `single` | `double`

### **$x_2$ — Sample data**

vector

Sample data from the second sample, specified as a vector. Data vectors  $x_1$  and  $x_2$  do not need to be the same size.

Data Types: `single` | `double`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example:

### **`Alpha` — Significance level**

0.05 (default) | scalar value in the range (0,1)

Significance level of the hypothesis test, specified as the comma-separated pair consisting of 'Alpha' and a scalar value in the range (0,1).

Example: 'Alpha', 0.01

Data Types: `single` | `double`

### **`Tail` — Type of alternative hypothesis**

'unequal' (default) | 'larger' | 'smaller'

Type of alternative hypothesis to evaluate, specified as the comma-separated pair consisting of 'Tail' and one of the following.

- 'unequal' Test the alternative hypothesis that the empirical cdf of  $x_1$  is unequal to the empirical cdf of  $x_2$ .
- 'larger' Test the alternative hypothesis that the empirical cdf of  $x_1$  is larger than the empirical cdf of  $x_2$ .
- 'smaller' Test the alternative hypothesis that the empirical cdf of  $x_1$  is smaller than the empirical cdf of  $x_2$ .

If the data values in  $x_1$  tend to be larger than those in  $x_2$ , the empirical distribution function of  $x_1$  tends to be smaller than that of  $x_2$ , and vice versa.

Example: 'Tail', 'larger'

## Output Arguments

### **h** — Hypothesis test result

1 | 0

Hypothesis test result, returned as a logical value.

- If  $h = 1$ , this indicates the rejection of the null hypothesis at the Alpha significance level.
- If  $h = 0$ , this indicates a failure to reject the null hypothesis at the Alpha significance level.

### **p** — Asymptotic $p$ -value

scalar value in the range (0,1)

Asymptotic  $p$ -value of the test, returned as a scalar value in the range (0,1).  $p$  is the probability of observing a test statistic as extreme as, or more extreme than, the observed value under the null hypothesis. The asymptotic  $p$ -value becomes very accurate for large sample sizes, and is believed to be reasonably accurate for sample sizes  $n_1$  and  $n_2$ , such that  $(n_1 * n_2) / (n_1 + n_2) \geq 4$ .

### **ks2stat** — Test statistic

nonnegative scalar value

Test statistic, returned as a nonnegative scalar value.

## Definitions

### Two-Sample Kolmogorov-Smirnov Test

The two-sample Kolmogorov-Smirnov test is a nonparametric hypothesis test that evaluates the difference between the cdfs of the distributions of the two sample data vectors over the range of  $x$  in each data set.

The two-sided test uses the maximum absolute difference between the cdfs of the distributions of the two data vectors. The test statistic is

$$D^* = \max_x \left( \left| F_1(x) - F_2(x) \right| \right),$$

where  $\hat{F}_1(x)$  is the proportion of  $x_1$  values less than or equal to  $x$  and  $\hat{F}_2(x)$  is the proportion of  $x_2$  values less than or equal to  $x$ .

The one-sided test uses the actual value of the difference between the cdfs of the distributions of the two data vectors rather than the absolute value. The test statistic is

$$D^* = \max_x \left( F_1(x) - F_2(x) \right).$$

## Algorithms

In `kstest2`, the decision to reject the null hypothesis is based on comparing the  $p$ -value `p` with the significance level `Alpha`, not by comparing the test statistic `ks2stat` with a critical value.

## References

- [1] Massey, F. J. “The Kolmogorov-Smirnov Test for Goodness of Fit.” *Journal of the American Statistical Association*. Vol. 46, No. 253, 1951, pp. 68–78.
- [2] Miller, L. H. “Table of Percentage Points of Kolmogorov Statistics.” *Journal of the American Statistical Association*. Vol. 51, No. 273, 1956, pp. 111–121.
- [3] Marsaglia, G., W. Tsang, and J. Wang. “Evaluating Kolmogorov’s Distribution.” *Journal of Statistical Software*. Vol. 8, Issue 18, 2003.



## See Also

`adtest` | `kstest` | `lillietest`

**Introduced before R2006a**

# kurtosis

Kurtosis

## Syntax

```
k = kurtosis(X)
k = kurtosis(X, flag)
k = kurtosis(X, flag, dim)
```

## Description

`k = kurtosis(X)` returns the sample kurtosis of `X`. For vectors, `kurtosis(x)` is the kurtosis of the elements in the vector `x`. For matrices `kurtosis(X)` returns the sample kurtosis for each column of `X`. For N-dimensional arrays, `kurtosis` operates along the first nonsingleton dimension of `X`.

`k = kurtosis(X, flag)` specifies whether to correct for bias (`flag` is 0) or not (`flag` is 1, the default). When `X` represents a sample from a population, the kurtosis of `X` is biased, that is, it will tend to differ from the population kurtosis by a systematic amount that depends on the size of the sample. You can set `flag` to 0 to correct for this systematic bias.

`k = kurtosis(X, flag, dim)` takes the kurtosis along dimension `dim` of `X`.

`kurtosis` treats NaNs as missing values and removes them.

## Examples

```
X = randn([5 4])
X =
    1.1650    1.6961   -1.4462   -0.3600
    0.6268    0.0591   -0.7012   -0.1356
    0.0751    1.7971    1.2460   -1.3493
    0.3516    0.2641   -0.6390   -1.2704
```

```

-0.6965  0.8717  0.5774  0.9846

k = kurtosis(X)
k =
  2.1658  1.2967  1.6378  1.9589

```

## Algorithms

Kurtosis is a measure of how outlier-prone a distribution is. The kurtosis of the normal distribution is 3. Distributions that are more outlier-prone than the normal distribution have kurtosis greater than 3; distributions that are less outlier-prone have kurtosis less than 3.

The kurtosis of a distribution is defined as

$$k = \frac{E(x - \mu)^4}{\sigma^4}$$

where  $\mu$  is the mean of  $x$ ,  $\sigma$  is the standard deviation of  $x$ , and  $E(t)$  represents the expected value of the quantity  $t$ . `kurtosis` computes a sample version of this population value.

---

**Note** Some definitions of kurtosis subtract 3 from the computed value, so that the normal distribution has kurtosis of 0. The `kurtosis` function does not use this convention.

---

When you set `flag` to 1, the following equation applies:

$$k_1 = \frac{\frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})^4}{\left( \frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})^2 \right)^2}$$

When you set `flag` to 0, the following equation applies:

$$k_0 = \frac{n-1}{(n-2)(n-3)} ((n+1)k_1 - 3(n-1)) + 3$$

This bias-corrected formula requires that `X` contain at least four elements.

## Extended Capabilities

### Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays” (MATLAB).

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`mean` | `moment` | `skewness` | `std` | `var`

**Introduced before R2006a**

# lasso

Regularized least-squares regression using lasso or elastic net algorithms

## Syntax

```
B = lasso(X,Y)
B = lasso(X,Y,Name,Value)
[B,FitInfo] = lasso(____)
```

## Description

`B = lasso(X,Y)` returns fitted least-squares regression coefficients for a set of regularization coefficients `Lambda`.

`B = lasso(X,Y,Name,Value)` fits regularized regressions with additional options specified by one or more `Name,Value` pair arguments.

`[B,FitInfo] = lasso(____)`, for any previous input syntax, also returns a structure containing information about the fits.

## Input Arguments

### **x**

Numeric matrix. Each row represents one observation, and each column represents one predictor (variable).

### **y**

Numeric vector of length `n`, where `n` is the number of rows of `X`. `Y(i)` is the response to row `i` of `X`.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### **AbsTol**

Absolute error tolerance used to determine convergence of “ADMM Algorithm” on page 32-3506. The algorithm converges when successive estimates of the coefficient vector differ by an amount less than `AbsTol`.

---

**Note** This option only applies when using `lasso` on tall arrays. See “Extended Capabilities” on page 32-0 for more information.

---

**Default:** `1e-4`

### **Alpha**

Scalar value in the interval  $(0, 1]$  representing the weight of lasso ( $L^1$ ) versus ridge ( $L^2$ ) optimization. `Alpha = 1` represents lasso regression, `Alpha` close to 0 approaches ridge regression, and other values represent elastic net optimization. See “Definitions” on page 32-3504.

**Default:** `1`

### **B0**

Initial values for  $x$ -coefficients in “ADMM Algorithm” on page 32-3506.

---

**Note** This option only applies when using `lasso` on tall arrays. See “Extended Capabilities” on page 32-0 for more information.

---

**Default:** Vector of zeros

### **CV**

Method `lasso` uses to estimate mean squared error:

- `K`, a positive integer — `lasso` uses `K`-fold cross-validation.
- `cvp`, a `cvpartition` object — `lasso` uses the cross-validation method expressed in `cvp`. You cannot use a 'leaveout' partition with `lasso`.
- 'resubstitution' — `lasso` uses `X` and `Y` to fit the model and to estimate the mean squared error without cross-validation.

**Default:** 'resubstitution'

#### **DFmax**

Maximum number of nonzero coefficients in the model. `lasso` returns results only for `Lambda` values that satisfy this criterion.

**Default:** `Inf`

#### **Lambda**

Vector of nonnegative `Lambda` values. See “Definitions” on page 32-3504.

- If you do not supply `Lambda`, `lasso` calculates the largest value of `Lambda` that gives a nonnull model. In this case, `LambdaRatio` gives the ratio of the smallest to the largest value of the sequence, and `NumLambda` gives the length of the vector.
- If you supply `Lambda`, `lasso` ignores `LambdaRatio` and `NumLambda`.

**Default:** Geometric sequence of `NumLambda` values, the largest just sufficient to produce  $B = 0$

#### **LambdaRatio**

Positive scalar, the ratio of the smallest to the largest `Lambda` value when you do not set `Lambda`.

If you set `LambdaRatio = 0`, `lasso` generates a default sequence of `Lambda` values, and replaces the smallest one with 0.

**Default:** `1e-4`

#### **MaxIter**

Maximum number of iterations allowed, specified as a positive integer. If the algorithm executes `MaxIter` iterations before reaching the convergence tolerance, then the function

stops iterating and returns a warning message. The function can return more than one warning when `NumLambda` is greater than 1.

**Default:** `1e5` (standard), `1e4` (for tall arrays)

### **MCReps**

Positive integer, the number of Monte Carlo repetitions for cross-validation.

- If CV is 'resubstitution' or a `cvpartition` of type 'resubstitution', `MCReps` must be 1.
- If CV is a `cvpartition` of type 'holdout', `MCReps` must be greater than 1.

**Default:** 1

### **NumLambda**

Positive integer, the number of `Lambda` values `lasso` uses when you do not set `Lambda`. `lasso` can return fewer than `NumLambda` fits if the residual error of the fits drops below a threshold fraction of the variance of  $Y$ .

**Default:** 100

### **Options**

Structure that specifies whether to cross-validate in parallel, and specifies the random streams. Create the `Options` structure with `statset`. Option fields:

- `UseParallel` — Set to `true` to compute in parallel. Default is `false`.
- `UseSubstreams` — Set to `true` to compute in parallel in a reproducible fashion. To compute reproducibly, set `Streams` to a type allowing substreams: `'mlfg6331_64'` or `'mrg32k3a'`. Default is `false`.
- `Streams` — A `RandStream` object or cell array consisting of one such object. If you do not specify `Streams`, `lasso` uses the default stream.

### **PredictorNames**

Cell array of character vectors representing names of the predictor variables, in the order in which they appear in `X`. For an example, see “Remove Redundant Predictors by Using Cross-Validated Fits” on page 32-3501.



**Default:** {}

### **RelTol**

Convergence threshold for the coordinate descent algorithm [3]. The algorithm terminates when successive estimates of the coefficient vector differ in the  $L^2$  norm by a relative amount less than `RelTol`.

**Default:** 1e-4

### **Rho**

Augmented Lagrangian parameter  $\rho$  for “ADMM Algorithm” on page 32-3506.

---

**Note** This option only applies when using `lasso` on tall arrays. See “Extended Capabilities” on page 32-0 for more information.

---

**Default:** Automatic selection

### **Standardize**

Boolean value specifying whether `lasso` scales  $X$  before fitting the models. This affects whether the regularization is applied to the coefficients on the standardized scale or original scale. The results are always presented on the original data scale.

$X$  and  $Y$  are always centered.

**Default:** true

### **u0**

Initial value of scaled dual variable  $u$  in “ADMM Algorithm” on page 32-3506.

---

**Note** This option only applies when using `lasso` on tall arrays. See “Extended Capabilities” on page 32-0 for more information.

---

**Default:** Vector of zeros

**Weights**

Observation weights, a nonnegative vector of length  $n$ , where  $n$  is the number of rows of  $X$ . `lasso` scales `Weights` to sum to 1.

**Default:** `1/n * ones(n,1)`

**Output Arguments****B**

Fitted coefficients, a  $p$ -by- $L$  matrix, where  $p$  is the number of predictors (columns) in  $X$ , and  $L$  is the number of `Lambda` values.

**FitInfo**

Structure containing information about the model fits.

Field in FitInfo	Description
Intercept	Intercept term $\beta_0$ for each linear model, a 1-by- $L$ vector
Lambda	Lambda parameters in ascending order, a 1-by- $L$ vector
Alpha	Value of Alpha parameter, a scalar
DF	Number of nonzero coefficients in <code>B</code> for each value of <code>Lambda</code> , a 1-by- $L$ vector
MSE	Mean squared error (MSE), a 1-by- $L$ vector

If you set the `CV` name-value pair to cross-validate, the `FitInfo` structure contains additional fields.

Field in FitInfo	Description
SE	The standard error of MSE for each <code>Lambda</code> , as calculated during cross-validation, a 1-by- $L$ vector
LambdaMinMSE	The <code>Lambda</code> value with minimum MSE, a scalar
Lambda1SE	The largest <code>Lambda</code> such that MSE is within one standard error of the minimum MSE, a scalar
IndexMinMSE	The index of <code>Lambda</code> with value <code>LambdaMinMSE</code> , a scalar

Field in FitInfo	Description
Index1SE	The index of Lambda with value Lambda1SE, a scalar

## Examples

### Remove Redundant Predictors

Construct a data set with redundant predictors and identify those predictors by using `lasso`.

Create a matrix `X` of 100 five-dimensional normal variables. Create a response vector `Y` from just two components of `X` and add a small amount of noise.

```
rng default % For reproducibility
X = randn(100,5);
r = [0;2;0;-3;0]; % Only two nonzero coefficients
Y = X*r + randn(100,1)*.1; % Small added noise
```

Construct the default lasso fit.

```
B = lasso(X,Y);
```

Find the coefficient vector for the 25th value in `B`.

```
B(:,25)
```

```
ans =
```

```

      0
  1.6093
      0
 -2.5865
      0
```

`lasso` identifies and removes the redundant predictors.

### Remove Redundant Predictors by Using Cross-Validated Fits

Construct a data set with redundant predictors and identify those predictors by using cross-validated lasso.

Create a matrix  $X$  of 100 five-dimensional normal variables. Create a response vector  $Y$  from two components of  $X$  and add a small amount of noise.

```
rng default % For reproducibility
X = randn(100,5);
r = [0;2;0;-3;0]; % Only two nonzero coefficients
Y = X*r + randn(100,1)*.1; % Small added noise
```

Construct the lasso fit by using tenfold cross-validation with labeled predictor variables.

```
[B,FitInfo] = lasso(X,Y,'CV',10,'PredictorNames',{'x1','x2','x3','x4','x5'});
```

Display the variables in the model that corresponds to the minimum cross-validated mean squared error (MSE).

```
minMSEModel = FitInfo.PredictorNames(B(:,FitInfo.IndexMinMSE)~=0)
```

```
minMSEModel =
    1x2 cell array
    {'x2'}    {'x4'}
```

Display the variables in the sparsest model within one standard error of the minimum MSE.

```
sparseModel = FitInfo.PredictorNames(B(:,FitInfo.Index1SE)~=0)
```

```
sparseModel =
    1x2 cell array
    {'x2'}    {'x4'}
```

In this example, `lasso` identifies the same predictors for the two models and removes the redundant predictors. However, in general, `lasso` can choose a different set of predictors.

### Lasso Plot with Cross-Validated Fits

Visually examine the cross-validated error of various levels of regularization.

Load the sample data.

```
load acetylene
```

Prepare the design matrix for a lasso fit with interactions.

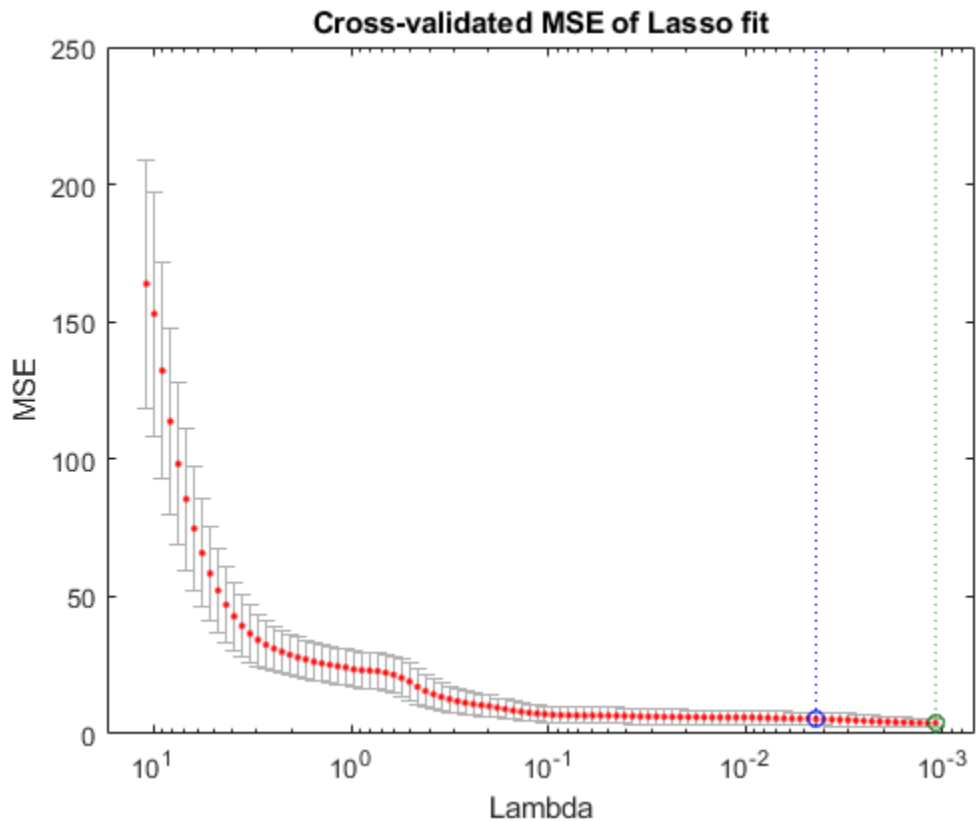
```
X = [x1 x2 x3];  
D = x2fx(X, 'interaction');  
D(:,1) = []; % No constant term
```

Construct the lasso fit using ten-fold cross-validation. Include the `FitInfo` output so you can plot the result.

```
rng default % For reproducibility  
[B,FitInfo] = lasso(D,y, 'CV',10);
```

Plot the cross-validated fits.

```
lassoPlot(B,FitInfo, 'PlotType', 'CV');
```



The green circle and dashed line locate the  $\text{Lambda}$  with minimum cross-validation error. The blue circle and dashed line locate the point with minimum cross-validation error plus one standard deviation.

## Definitions

### Lasso

For a given value of  $\lambda$ , a nonnegative parameter, `lasso` solves the problem

$$\min_{\beta_0, \beta} \left( \frac{1}{2N} \sum_{i=1}^N (y_i - \beta_0 - x_i^T \beta)^2 + \lambda \sum_{j=1}^p |\beta_j| \right).$$

- $N$  is the number of observations.
- $y_i$  is the response at observation  $i$ .
- $x_i$  is data, a vector of  $p$  values at observation  $i$ .
- $\lambda$  is a nonnegative regularization parameter corresponding to one value of  $\Lambda$ .
- The parameters  $\beta_0$  and  $\beta$  are a scalar and a vector of length  $p$ , respectively.

As  $\lambda$  increases, the number of nonzero components of  $\beta$  decreases.

The lasso problem involves the  $L^1$  norm of  $\beta$ , as contrasted with the elastic net algorithm.

## Elastic Net

For an  $\alpha$  strictly between 0 and 1, and a nonnegative  $\lambda$ , elastic net solves the problem

$$\min_{\beta_0, \beta} \left( \frac{1}{2N} \sum_{i=1}^N (y_i - \beta_0 - x_i^T \beta)^2 + \lambda P_\alpha(\beta) \right),$$

where

$$P_\alpha(\beta) = \frac{(1-\alpha)}{2} \|\beta\|_2^2 + \alpha \|\beta\|_1 = \sum_{j=1}^p \left( \frac{(1-\alpha)}{2} \beta_j^2 + \alpha |\beta_j| \right).$$

Elastic net is the same as lasso when  $\alpha = 1$ . As  $\alpha$  shrinks toward 0, elastic net approaches ridge regression. For other values of  $\alpha$ , the penalty term  $P_\alpha(\beta)$  interpolates between the  $L^1$  norm of  $\beta$  and the squared  $L^2$  norm of  $\beta$ .

## Algorithms

### ADMM Algorithm

When operating on tall arrays, `lasso` uses an algorithm based on the Alternating Direction Method of Multipliers (ADMM) [5]. The notation used here is the same as in the reference paper. This method solves problems of the form

$$\text{Minimize } l(x) + g(z)$$

$$\text{Subject to } Ax + Bz = c$$

Using this notation the lasso regression problem is

$$\text{Minimize } l(x) + g(z) = \frac{1}{2} \|Ax - b\|_2^2 + \lambda \|z\|_1$$

$$\text{Subject to } x - z = 0$$

Since the loss function  $l(x) = \frac{1}{2} \|Ax - b\|_2^2$  is quadratic, the iterative updates performed by the algorithm amount to solving a linear system of equations with a single coefficient matrix but several right-hand sides. The updates performed by the algorithm during each iteration are

$$x^{k+1} = (A^T A + \rho I)^{-1} (A^T b + \rho (z^k - u^k))$$

$$z^{k+1} = S_{\lambda/\rho} (x^{k+1} + u^k)$$

$$u^{k+1} = u^k + x^{k+1} - z^{k+1}.$$

$A$  is the dataset (a tall array),  $x$  contains the coefficients,  $\rho$  is the penalty parameter (augmented Lagrangian parameter),  $b$  is the response (a tall array), and  $S$  is the soft thresholding operator.

$$S_{\kappa}(a) = \begin{cases} a - \kappa, & a > \kappa \\ 0, & |a| \leq \kappa \\ a + \kappa, & a < -\kappa \end{cases}.$$



lasso solves the linear system using Cholesky factorization since the coefficient matrix

$A^T A + \rho I$  is symmetric and positive definite. Since  $\rho$  does not change between iterations, the Cholesky factorization is cached between iterations instead of solving from scratch.

Even though  $A$  and  $b$  are tall arrays, they appear only in the terms  $A^T A$  and  $A^T b$ . The results of these two matrix multiplications are small enough to fit in memory, so they are precomputed and the iterative updates between iterations are performed entirely within memory.

## References

- [1] Tibshirani, R. “Regression Shrinkage and Selection via the Lasso.” *Journal of the Royal Statistical Society. Series B*, Vol. 58, No. 1, 1996, pp. 267–288.
- [2] Zou, H. and T. Hastie. “Regularization and Variable Selection via the Elastic Net.” *Journal of the Royal Statistical Society. Series B*, Vol. 67, No. 2, 2005, pp. 301–320.
- [3] Friedman, J., R. Tibshirani, and T. Hastie. “Regularization Paths for Generalized Linear Models via Coordinate Descent.” *Journal of Statistical Software*. Vol. 33, No. 1, 2010. <http://www.jstatsoft.org/v33/i01>
- [4] Hastie, T., R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*. 2nd edition. New York: Springer, 2008.
- [5] Boyd, S. “Distributed Optimization and Statistical Learning via the Alternating Direction Method of Multipliers”. *Foundations and Trends in Machine Learning*. Vol 3, No. 1, 2010, pp. 1–122.

## Extended Capabilities

### Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function supports tall arrays for out-of-memory data with some limitations.

- With tall arrays, `lasso` uses an algorithm based on ADMM (alternating direction method of multipliers).
- No elastic-net support. The `'Alpha'` parameter is always 1.
- No cross-validation (`'CV'` parameter) support, which includes the related parameter `'MCReps'`.
- The second output `FitInfo` does not contain the additional fields: `'SE'`, `'LambdaMinMSE'`, `'Lambda1SE'`, `'IndexMinMSE'`, and `'Index1SE'`.
- The `'Options'` parameter is not supported, since it does not contain options that apply to the ADMM algorithm. You can tune the ADMM algorithm using name-value pairs.
- Supported name-value pairs are:
  - `'Lambda'`
  - `'LambdaRatio'`
  - `'NumLambda'`
  - `'Standardize'`
  - `'PredictorNames'`
  - `'RelTol'`
  - `'Weights'`
- Additional name-value pairs to control the ADMM algorithm are:
  - `'Rho'` — Augmented Lagrangian parameter,  $\rho$ . Default value is automatic selection.
  - `'AbsTol'` — Absolute tolerance used to determine convergence. Default value is  $1e-4$ .
  - `'MaxIter'` — Maximum number of iterations. Default value is  $1e4$ .
  - `'B0'` — Initial values for the coefficients  $x$ . Default value is a vector of zeros.
  - `'U0'` — Initial values of scaled dual variable  $u$ . Default value is a vector of zeros.

For more information, see “Tall Arrays” (MATLAB).

## See Also

`lassoPlot` | `ridge`

## Topics

“Lasso and Elastic Net” on page 11-141

**Introduced in R2011b**

## lassoglm

Lasso or elastic net regularization for generalized linear model regression

### Syntax

```
B = lassoglm(X, Y)
B = lassoglm(X, Y, distr)
B = lassoglm(X, Y, distr, Name, Value)
[B, FitInfo] = lassoglm( ___ )
```

### Description

`B = lassoglm(X, Y)` returns penalized maximum-likelihood fitted coefficients for a generalized linear model of the response `Y` to the data matrix `X`. The values in `Y` are assumed to have a Gaussian probability distribution.

`B = lassoglm(X, Y, distr)` fits the model using the probability distribution type for `Y` specified in `distr`.

`B = lassoglm(X, Y, distr, Name, Value)` fits regularized generalized linear regressions with additional options specified by one or more `Name, Value` pair arguments.

`[B, FitInfo] = lassoglm( ___ )`, for any previous input syntax, also returns a structure containing information about the fits.

### Input Arguments

#### **x**

Numeric matrix with `n` rows and `p` columns. Each row represents one observation, and each column represents one predictor (variable).

**Y**

When `distr` is not 'binomial', `Y` is a numeric vector or categorical array of length `n`, where `n` is the number of rows of `X`. `Y(i)` is the response to row `i` of `X`.

When `distr` is 'binomial', `Y` is either:

- A numeric vector of length `n`, where each entry represents success (1) or failure (0)
- A logical vector of length `n`, where each entry represents success or failure
- A categorical array of length `n`, where each entry represents success or failure
- A two-column numeric matrix, where the first column contains the number of successes for each observation and the second column contains the total number of trials

**distr**

Distributional family for the nonsystematic variation in the responses. Choices:

- 'normal'
- 'binomial'
- 'poisson'
- 'gamma'
- 'inverse gaussian'

By default, `lassoglm` uses the canonical link function on page 32-3519 corresponding to `distr`. Specify another link function using the 'link' name-value pair.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

**Alpha**

Scalar value from 0 to 1 (excluding 0) representing the weight of lasso ( $L^1$ ) versus ridge ( $L^2$ ) optimization. `Alpha = 1` represents lasso regression, and other values represent

elastic net optimization. Alpha close to 0 approaches ridge regression. See “Definitions” on page 32-3504.

**Default:** 1

### **cv**

Method `lassoglm` uses to estimate deviance:

- `K`, a positive integer — `lassoglm` uses `K`-fold cross-validation.
- `cvp`, a `cvpartition` object — `lassoglm` uses the cross-validation method expressed in `cvp`. You cannot use a 'leaveout' partition with `lassoglm`.
- 'resubstitution' — `lassoglm` uses `X` and `Y` to fit the model and to estimate the deviance, without cross-validation.

**Default:** 'resubstitution'

### **DFmax**

Maximum number of nonzero coefficients in the model. `lassoglm` returns results for `Lambda` values that satisfy this criterion.

**Default:** `Inf`

### **Lambda**

Vector of nonnegative `Lambda` values. See “Lasso” on page 32-3520.

- If you do not supply `Lambda`, `lassoglm` estimates the largest value of `Lambda` that gives a nonnull model. In this case, `LambdaRatio` gives the ratio of the smallest to the largest value of the sequence, and `NumLambda` gives the length of the vector.
- If you supply `Lambda`, `lassoglm` ignores `LambdaRatio` and `NumLambda`.

**Default:** Geometric sequence of `NumLambda` values, the largest just sufficient to produce  $B = 0$

### **LambdaRatio**

Positive scalar, the ratio of the smallest to the largest `Lambda` value when you do not explicitly set `Lambda`.

If you set `LambdaRatio = 0`, `lassoglm` generates a default sequence of `Lambda` values, and replaces the smallest one with 0.

**Default:** `1e-4`

### Link

Specify the mapping between the mean  $\mu$  of the response and the linear predictor  $Xb$ .

Value	Description
'comploglog'	$\log(-\log((1-\mu))) = Xb$
'identity', default for the distribution 'normal'	$\mu = Xb$
'log', default for the distribution 'poisson'	$\log(\mu) = Xb$
'logit', default for the distribution 'binomial'	$\log(\mu/(1-\mu)) = Xb$
'loglog'	$\log(-\log(\mu)) = Xb$
'probit'	$\Phi^{-1}(\mu) = Xb$ , where $\Phi$ is the normal (Gaussian) CDF function
'reciprocal', default for the distribution 'gamma'	$\mu^{-1} = Xb$
<code>p</code> (a number), default for the distribution 'inverse gaussian' (with $p = -2$ )	$\mu^p = Xb$
A cell array of the form {FL FD FI}, containing three function handles, created using @, that define the link (FL), the derivative of the link (FD), and the inverse link (FI), or equivalently, a structure of function handles with field <code>Link</code> containing FL, field <code>Derivative</code> containing FD, and field <code>Inverse</code> containing FI	User-specified link function (see “Custom Link Function” on page 12-16)

**MaxIter**

Maximum number of iterations allowed, specified as positive integer. If the algorithm executes `MaxIter` iterations before reaching the convergence tolerance `RelTol`, the function stops iterating and returns a warning message. The function can return more than one warning when `NumLambda` is greater than 1.

**Default:** `1e4`

**MCREps**

Positive integer, the number of Monte Carlo repetitions for cross-validation.

- If `CV` is `'resubstitution'` or a `cvpartition` of type `'resubstitution'`, `MCREps` must be 1.
- If `CV` is a `cvpartition` of type `'holdout'`, `MCREps` must be greater than 1.

**Default:** `1`

**NumLambda**

Positive integer, the number of `Lambda` values `lassoglm` uses when you do not set `Lambda`. `lassoglm` can return fewer than `NumLambda` fits if the deviance of the fits drops below a threshold fraction of the null deviance (deviance of the fit without any predictors `X`).

**Default:** `100`

**Offset**

Numeric vector with the same number of rows as `X`. `lassoglm` uses `Offset` as an additional predictor variable, but keeps its coefficient value fixed at 1.0.

**Options**

Structure that specifies whether to cross-validate in parallel, and specifies the random stream or streams. Create the `Options` structure with `statset`. Option fields:

- `UseParallel` — Set to `true` to compute in parallel. Default is `false`.
- `UseSubstreams` — Set to `true` to compute in parallel in a reproducible fashion. To compute reproducibly, set `Streams` to a type allowing substreams: `'mlfg6331_64'` or `'mrg32k3a'`. Default is `false`.



- `Streams` — `RandStream` object or cell array consisting of one such object. If you do not specify `Streams`, `lassoglm` uses the default stream.

**PredictorNames**

Cell array of character vectors representing names of the predictor variables, in the order in which they appear in `X`.

**Default:** `{}`

**RelTol**

Convergence threshold for the coordinate descent algorithm (see Friedman, Tibshirani, and Hastie [3]). The algorithm terminates when successive estimates of the coefficient vector differ in the  $L^2$  norm by a relative amount less than `RelTol`.

**Default:** `1e-4`

**Standardize**

Boolean value specifying whether `lassoglm` scales `X` before fitting the models. This affects whether the regularization is applied to the coefficients on the standardized scale or original scale. The results are always presented on the original scale.

**Default:** `true`

**Weights**

Observation weights, a nonnegative vector of length `n`, where `n` is the number of rows of `X`. At least two values must be positive.

**Default:** `1/n * ones(n,1)`

## Output Arguments

**B**

Fitted coefficients, a `p`-by-`L` matrix, where `p` is the number of predictors (columns) in `X`, and `L` is the number of `Lambda` values.

**FitInfo**

Structure containing information about the model fits.

Field in FitInfo	Description
Alpha	Value of Alpha parameter, a scalar.
Deviance	Deviance of the fitted model for each value of Lambda, a 1-by-L vector. If cross-validation was performed, the values for Deviance represent the estimated expected deviance of the model applied to new data, as calculated by cross-validation. Otherwise, Deviance is the deviance of the fitted model applied to the data used to perform the fit.
DF	Number of nonzero coefficients in B for each Lambda value, a 1-by-L vector.
Intercept	Intercept term $\beta_0$ for each linear model, a 1-by-L vector.
Lambda	Lambda parameters in ascending order, a 1-by-L vector.

If you set the CV name-value pair to cross-validate, the FitInfo structure contains additional fields.

Field in FitInfo	Description
IndexMinDeviance	Index of Lambda with value LambdaMinDeviance, a scalar.
Index1SE	Index of Lambda with value Lambda1SE, a scalar.
LambdaMinDeviance	Lambda value with minimum expected deviance, as calculated by cross-validation, a scalar.
Lambda1SE	Largest Lambda such that Deviance is within one standard error of the minimum, a scalar.
SE	Standard error of Deviance for each Lambda, as calculated during cross-validation, a 1-by-L vector.

**Examples**

## Lasso Regularization of a Generalized Linear Model

Construct data from a Poisson model, and identify the important predictors using `lassoglm`.

Create data with 20 predictors, and Poisson responses using just three of the predictors plus a constant.

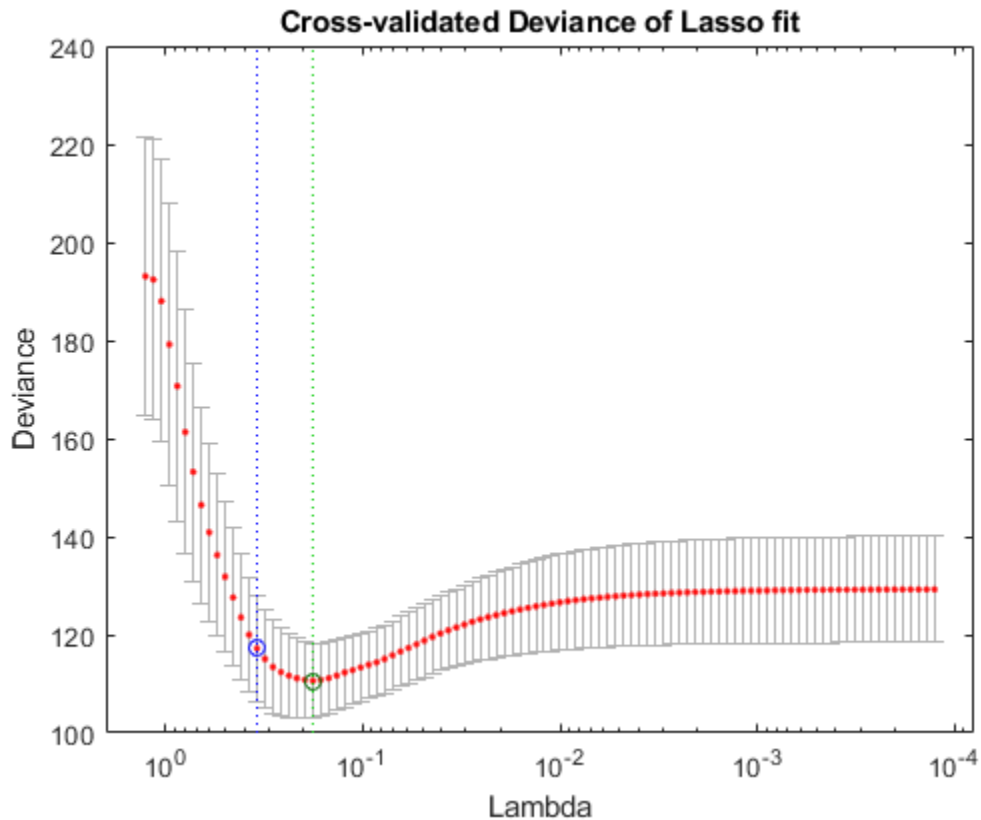
```
rng default % For reproducibility
X = randn(100,20);
mu = exp(X(:, [5 10 15]) * [.4; .2; .3] + 1);
y = poissrnd(mu);
```

Construct a cross-validated lasso regularization of a Poisson regression model of the data.

```
[B, FitInfo] = lassoglm(X,y, 'poisson', 'CV', 10);
```

Examine the cross-validation plot to see the effect of the Lambda regularization parameter.

```
lassoPlot(B, FitInfo, 'plottype', 'CV');
```



The green circle and dashed line locate the `Lambda` with minimum cross-validation error. The blue circle and dashed line locate the point with minimum cross-validation error plus one standard deviation.

Find the nonzero model coefficients corresponding to the two identified points.

```
minpts = find(B(:,FitInfo.IndexMinDeviance))
```

```
minpts =
```

```
3
5
6
```

```

10
11
15
16

```

```
minlpts = find(B(:,FitInfo.Index1SE))
```

```
minlpts =
```

```

5
10
15

```

The coefficients from the minimum-plus-one standard error point are exactly those coefficients used to create the data.

- “Regularize Poisson Regression” on page 12-49
- “Regularize Logistic Regression” on page 12-53
- “Regularize Wide Data in Parallel” on page 12-61

## Definitions

### Link Function

A link function  $f(\mu)$  maps a distribution with mean  $\mu$  to a linear model with data  $X$  and coefficient vector  $b$  using the formula

$$f(\mu) = Xb.$$

Find the formulas for the link functions in the `Link` name-value pair description. The following table lists the link functions that are typically used for each distribution.

Distributional Family	Default Link Function	Other Typical Link Functions
'normal'	'identity'	
'binomial'	'logit'	'comploglog', 'loglog', 'probit'

Distributional Family	Default Link Function	Other Typical Link Functions
'poisson'	'log'	
'gamma'	'reciprocal'	
'inverse gaussian'	-2	

## Lasso

For a nonnegative value of  $\lambda$ , `lasso` solves the problem

$$\min_{\beta_0, \beta} \left( \frac{1}{N} \text{Deviance}(\beta_0, \beta) + \lambda \sum_{j=1}^p |\beta_j| \right).$$

- The function `Deviance` in this equation is the deviance of the model fit to the responses using intercept  $\beta_0$  and predictor coefficients  $\beta$ . The formula for `Deviance` depends on the `dist` parameter you supply to `lasso glm`. Minimizing the  $\lambda$ -penalized deviance is equivalent to maximizing the  $\lambda$ -penalized log likelihood.
- $N$  is the number of observations.
- $\lambda$  is a nonnegative regularization parameter corresponding to one value of `Lambda`.
- Parameters  $\beta_0$  and  $\beta$  are a scalar and a vector of length  $p$ , respectively.

As  $\lambda$  increases, the number of nonzero components of  $\beta$  decreases.

The lasso problem involves the  $L^1$  norm of  $\beta$ , as contrasted with the elastic net algorithm.

## Elastic Net

For an  $\alpha$  strictly between 0 and 1, and a nonnegative  $\lambda$ , elastic net solves the problem

$$\min_{\beta_0, \beta} \left( \frac{1}{N} \text{Deviance}(\beta_0, \beta) + \lambda P_\alpha(\beta) \right),$$

where

$$P_\alpha(\beta) = \frac{(1-\alpha)}{2} \|\beta\|_2^2 + \alpha \|\beta\|_1 = \sum_{j=1}^p \left( \frac{(1-\alpha)}{2} \beta_j^2 + \alpha |\beta_j| \right)$$

Elastic net is the same as lasso when  $a = 1$ . For other values of  $a$ , the penalty term  $P_a(\beta)$  interpolates between the  $L^1$  norm of  $\beta$  and the squared  $L^2$  norm of  $\beta$ . As  $a$  shrinks toward 0, elastic net approaches ridge regression.

## References

- [1] Tibshirani, R. “Regression Shrinkage and Selection via the Lasso.” *Journal of the Royal Statistical Society. Series B*, Vol. 58, No. 1, 1996, pp. 267–288.
- [2] Zou, H. and T. Hastie. “Regularization and Variable Selection via the Elastic Net.” *Journal of the Royal Statistical Society. Series B*, Vol. 67, No. 2, 2005, pp. 301–320.
- [3] Friedman, J., R. Tibshirani, and T. Hastie. “Regularization Paths for Generalized Linear Models via Coordinate Descent.” *Journal of Statistical Software*. Vol. 33, No. 1, 2010. <http://www.jstatsoft.org/v33/i01>
- [4] Hastie, T., R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*. 2nd edition. New York: Springer, 2008.
- [5] Dobson, A. J. *An Introduction to Generalized Linear Models*. 2nd edition. New York: Chapman & Hall/CRC Press, 2002.
- [6] McCullagh, P., and J. A. Nelder. *Generalized Linear Models*. 2nd edition. New York: Chapman & Hall/CRC Press, 1989.
- [7] Collett, D. *Modelling Binary Data*, 2nd edition. New York: Chapman & Hall/CRC Press, 2003.

## See Also

`glmfit` | `lasso` | `lassoPlot` | `ridge`

## Topics

- “Regularize Poisson Regression” on page 12-49
- “Regularize Logistic Regression” on page 12-53
- “Regularize Wide Data in Parallel” on page 12-61
- “Lasso Regularization of Generalized Linear Models” on page 12-46

**Introduced in R2012a**



# lassoPlot

Trace plot of lasso fit

## Syntax

```
lassoPlot(B)  
lassoPlot(B,FitInfo)  
lassoPlot(B,FitInfo,Name,Value)  
[ax,figh] = lassoPlot(____)
```

## Description

`lassoPlot(B)` creates a trace plot of the values in `B` against the  $L^1$  norm of `B`.

`lassoPlot(B,FitInfo)` creates a plot with type depending on the data type of `FitInfo` and the value, if any, of the `PlotType` name-value pair.

`lassoPlot(B,FitInfo,Name,Value)` creates a plot with additional options specified by one or more `Name,Value` pair arguments.

`[ax,figh] = lassoPlot(____)`, for any previous input syntax, returns a handle `ax` to the plot axis, and a handle `figh` to the figure window.

## Input Arguments

### **B**

Coefficients of a sequence of regression fits, as returned from the `lasso` or `lassoglm` functions. `B` is a `p`-by-`NLambda` matrix, where `p` is the number of predictors, and each column of `B` is a set of coefficients `lasso` calculates using one `Lambda` penalty value.

### **FitInfo**

Information controlling the plot:

- `FitInfo` is a structure, especially as returned from `lasso` or `lassoglm` — `lassoPlot` creates a plot based on the `PlotType` name-value pair.
- `FitInfo` is a vector — `lassoPlot` forms the  $x$ -axis of the plot from the values in `FitInfo`. The length of `FitInfo` must equal the number of columns of `B`.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### Parent

Axis in which to draw the plot.

**Default:** New plot

### PlotType

Plot type when you specify a `FitInfo` vector or structure:

PlotType	Plot
'L1'	<code>lassoPlot</code> creates the $x$ -axis from the $L^1$ norm of the coefficients in <code>B</code> . The $x$ -axis at the top of the plot contains the degrees of freedom (df), meaning the number of nonzero coefficients of <code>B</code> .
'Lambda' When you choose this value, <code>FitInfo</code> must be a structure.	<code>lassoPlot</code> creates the $x$ -axis from the <code>Lambda</code> field of <code>FitInfo</code> . The $x$ -axis at the top of the plot contains the degrees of freedom (df), meaning the number of nonzero coefficients of <code>B</code> .

PlotType	Plot
'CV' When you choose this value, <code>FitInfo</code> must be a cross-validated structure.	<ul style="list-style-type: none"> <li>For each <code>Lambda</code>, <code>lassoPlot</code> plots an estimate of the mean squared prediction error on new data for the model fitted by <code>lasso</code> with that value of <code>Lambda</code>.</li> <li><code>lassoPlot</code> plots error bars for the estimates.</li> </ul>

If you include a cross-validated `FitInfo` structure, `lassoPlot` also indicates two specific `Lambda` values with green and blue dashed lines.

- A green, dashed line indicates the value of `Lambda` with a minimum cross-validated mean squared error (MSE).
- A blue, dashed line indicates the greatest `Lambda` that is within one standard error of the minimum MSE. This `Lambda` value makes the sparsest model with relatively low MSE.

**Default:** 'L1'

### **PredictorNames**

Cell array of character vectors to label each coefficient of `B`. If the length of `PredictorNames` is less than the number of rows of `B`, the remaining labels are padded with default values.

`lassoPlot` uses `PredictorNames` in `FitInfo` only if:

- You created `FitInfo` with a call to `lasso` that included a `PredictorNames` name-value pair.
- You call `lassoPlot` *without* a `PredictorNames` name-value pair.
- You include `FitInfo` in your `lassoPlot` call.

For an example, see “Lasso Plot with Coefficients Labeled” on page 32-3527.

**Default:** {'B1', 'B2', ...}

**XScale**

- 'linear' for linear x-axis
- 'log' for logarithmic scaled x-axis

**Default:** 'linear', except 'log' for the 'CV' plot type

## Output Arguments

**ax**

Handle to the axis of the plot (see “Axes Appearance” (MATLAB)).

**figh**

Handle to the figure window (see “Special Object Identifiers” (MATLAB)).

## Examples

### Lasso Plot with Default Plot Type

Load the sample data

```
load acetylene
```

Prepare the design matrix for lasso fit with interactions.

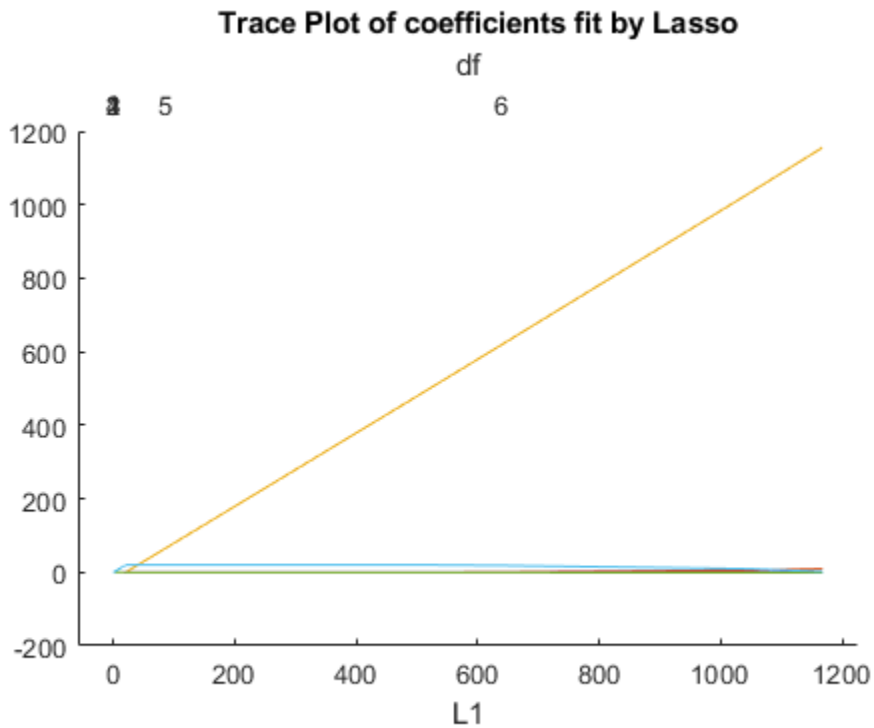
```
X = [x1 x2 x3];  
D = x2fx(X, 'interaction');  
D(:,1) = []; % No constant term
```

Fit a regularized model of the data using `lasso`.

```
B = lasso(D,y);
```

Plot the fits with the default plot type.

```
lassoPlot(B);
```



### Lasso Plot with Coefficients Labeled

Load the sample data.

```
load acetylene
```

Prepare the data for lasso fit with interactions.

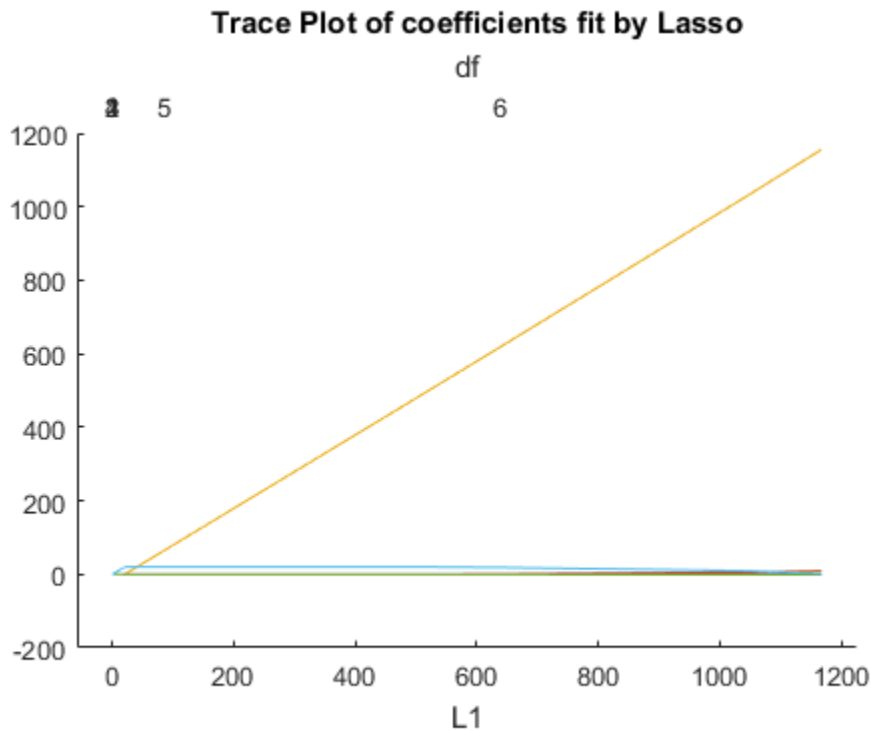
```
X = [x1 x2 x3];
D = x2fx(X, 'interaction');
D(:,1) = []; % No constant term
```


Fit a regularized model of the data with `lasso`.

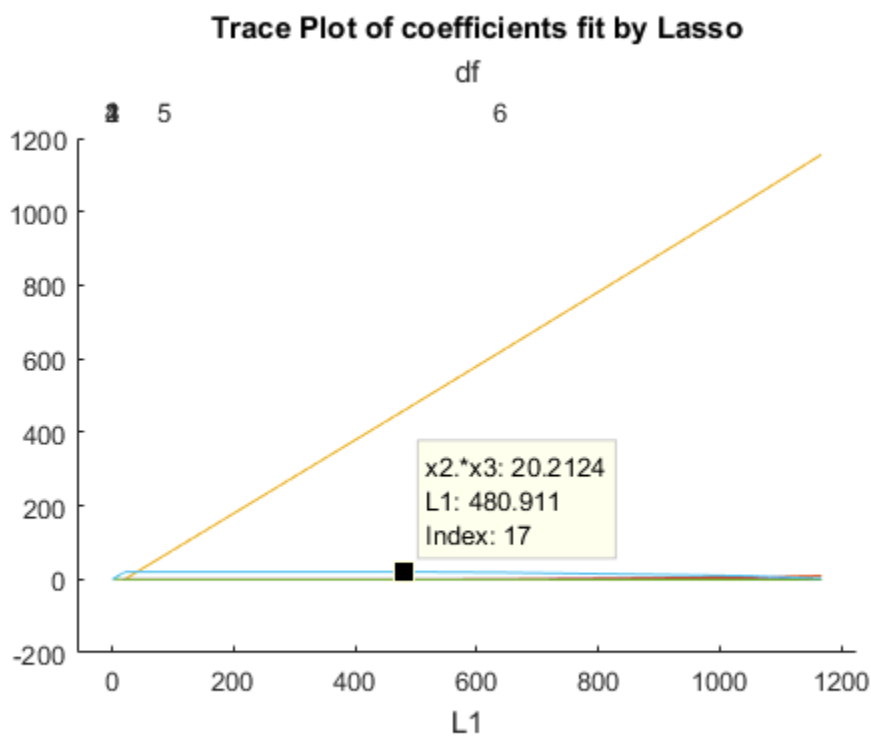
```
[B,FitInfo] = lasso(D,y);
```

Plot the lasso fits with labeled coefficients by using the `PredictorNames` name-value pair. The `x2fx` reference page shows the order of linear terms and interaction terms: `x1`, `x2`, `x3`, `x1.*x2`, `x1.*x3`, and `x2.*x3`.

```
lassoPlot(B,FitInfo,'PredictorNames',{'x1','x2','x3','x1.*x2','x1.*x3','x2.*x3'});
```



Each line represents a trace of the values in `B` of a single predictor variable. Select the data cursor icon  on the figure toolbar or select **Tools > Data Cursor**. Once you have enabled data cursor mode, selecting a point on the line opens a data tip.



A data tip displays these lines of information: the name of the selected coefficient with a fitted value, the  $L^1$  norm of a set of coefficients including the selected coefficient, and the index of the corresponding  $\text{Lambda}$ .

### Lasso Plot with Lambda Plot Type

Load the sample data.

```
load acetylene
```

Prepare the data for lasso fit with interactions.

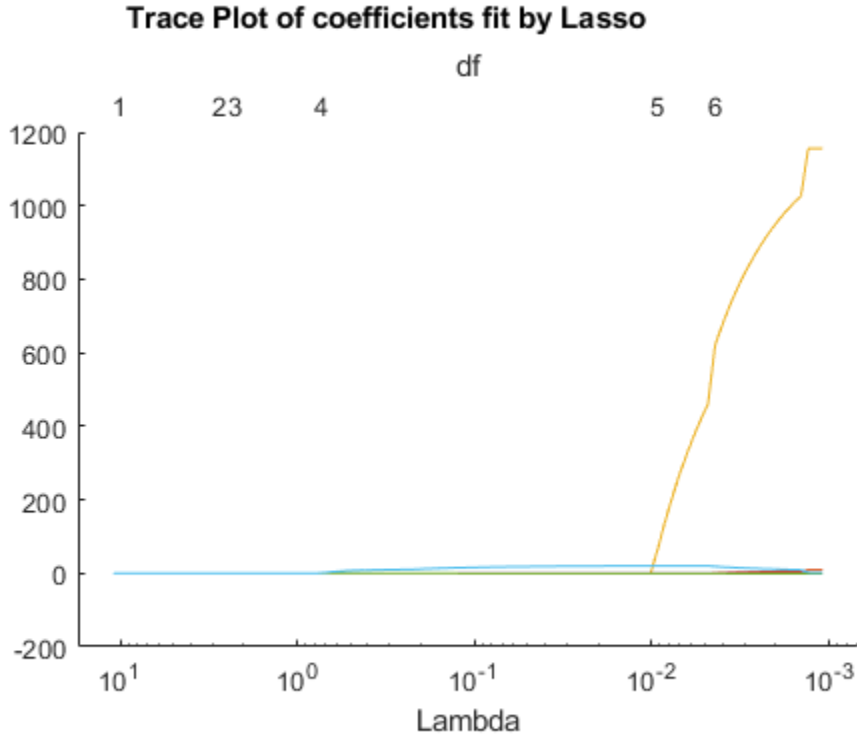
```
X = [x1 x2 x3];  
D = x2fx(X, 'interaction');  
D(:,1) = []; % No constant term
```

Fit a regularized model of the data with `lasso`.

```
[B,FitInfo] = lasso(D,y);
```

Plot the fits with the `Lambda` plot type and logarithmic scaling.

```
lassoPlot(B,FitInfo, 'PlotType', 'Lambda', 'XScale', 'log');
```





## Lasso Plot with Cross-Validated Fits

Visually examine the cross-validated error of various levels of regularization.

Load the sample data.

```
load acetylene
```

Prepare the design matrix for a lasso fit with interactions.

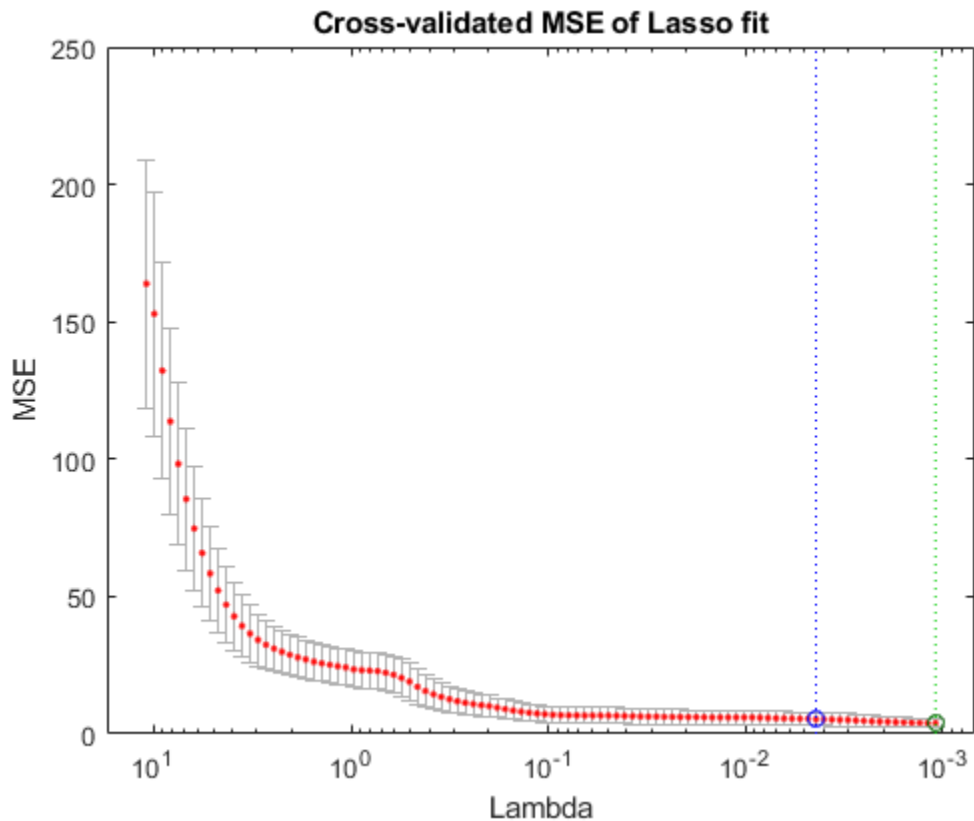
```
X = [x1 x2 x3];  
D = x2fx(X, 'interaction');  
D(:,1) = []; % No constant term
```

Construct the lasso fit using ten-fold cross-validation. Include the `FitInfo` output so you can plot the result.

```
rng default % For reproducibility  
[B,FitInfo] = lasso(D,y,'CV',10);
```

Plot the cross-validated fits.

```
lassoPlot(B,FitInfo,'PlotType','CV');
```



The green circle and dashed line locate the  $\text{Lambda}$  with minimum cross-validation error. The blue circle and dashed line locate the point with minimum cross-validation error plus one standard deviation.

## See Also

`lasso` | `lassoglm`

## Topics

“Lasso and Elastic Net” on page 11-141

**Introduced in R2011b**

## le

**Class:** grandstream

Less than or equal relation for handles

## Syntax

```
h1 <= h2
```

## Description

Handles are equal if they are handles for the same object. All comparisons use a number associated with each handle object. Nothing can be assumed about the result of a handle comparison except that the repeated comparison of two handles in the same MATLAB session will yield the same result. The order of handle values is purely arbitrary and has no connection to the state of the handle objects being compared.

`h1 <= h2` performs element-wise comparisons between handle arrays `h1` and `h2`. `h1` and `h2` must be of the same dimensions unless one is a scalar. The result is a logical array of the same dimensions, where each element is an element-wise `<=` result.

If one of `h1` or `h2` is scalar, scalar expansion is performed and the result will match the dimensions of the array that is not scalar.

`tf = le(h1, h2)` stores the result in a logical array of the same dimensions.

## See Also

`eq` | `ge` | `gt` | `lt` | `ne` | `grandstream`

## Leap property

**Class:** grandset

Interval between points

### Description

Number of points to leap over and omit for each point taken from the sequence. The Leap property of a point set contains a positive integer which specifies the number of points in the sequence to leap over and omit for every point taken. The default Leap value is 0, which corresponds to taking every point from the sequence.

Leaping is a technique used to improve the quality of a point set. However, you must choose the Leap values with care; many Leap values create sequences that fail to touch on large sub-hyper-rectangles of the unit hypercube, and so fail to be a uniform quasi-random point set.

### Choosing Leap Values for Halton Sets

A known rule for choosing Leap values for Halton sets is to set it to  $(P-1)$  where  $P$  is a prime number that has not been used to generate one of the dimensions, i.e. for a  $k$ -dimensional point set  $P$  would be the  $(k+1)$ th or greater prime.

### Examples

Experiment with different leap values:

```
% No leaping produces the standard Halton sequence.
P = haltonset(5);
P(1:5, :)
```

```
% Set a leap of 1. The point set now includes every other
% point from the sequence.
P.Leanp = 1;
P(1:5, :)
```

## See Also

`skip` | `haltonset` | `net` | `grandset` | `subsref`

# length

**Class:** dataset

Length of dataset array

---

**Note** The `dataset` data type might be removed in a future release. To work with heterogeneous data, use the MATLAB® `table` data type instead. See MATLAB `table` documentation for more information.

---

## Syntax

```
n = length(A)
```

## Description

`n = length(A)` returns the number of observations in the dataset A. `length` is equivalent to `size(A,1)`.

## See Also

`size`

## length

**Class:** grandset

Length of point set

## Syntax

`length(p)`

## Description

`length(p)` returns the number of points in the point set `p`. It is equivalent to `size(p, 1)`.

## See Also

`grandset` | `size`



# levelcounts

Element counts by level of a nominal or ordinal array

---

**Note** The `nominal` and `ordinal` array data types might be removed in a future release. To represent ordered and unordered discrete, nonnumeric data, use the MATLAB® `categorical` data type instead.

---

## Syntax

```
C = levelcounts(A)
C = levelcounts(A,dim)
```

## Description

`C = levelcounts(A)` returns counts of the number of elements in the nominal or ordinal array `A` equal to each of the possible levels in `A` into the vector `C`, which has as many elements as `A` has levels.

- If `A` is a matrix, then `C` is a matrix of column counts.
- If `A` is an  $N$ -dimensional array, `levelcounts` operates along the first nonsingleton dimension.

`C = levelcounts(A,dim)` operates along the dimension `dim`.

## Examples

### Count Observations in Each Level

Create a nominal array from data in a cell array.

```
colors = nominal({'r','b','g';'g','r','b';'b','r','g'},...
                 {'blue','green','red'})
```

```
colors =  
  
3x3 nominal array  
  
   red      blue     green  
   green    red      blue  
   blue     red      green
```

Count the number of observations of each level in each column.

```
levelcounts(colors)
```

```
ans =  
  
   1   1   1  
   1   0   2  
   1   2   0
```

Count the number of observations of each level in each row.

```
levelcounts(colors,2)
```

```
ans =  
  
   1   1   1  
   1   1   1  
   1   1   1
```

Alternatively, you can use `summary` to display the counts with their labels. The default is to count elements in each column.

```
summary(colors)  
  
   blue      1      1      1  
   green     1      0      2  
   red       1      2      0
```

You can also count elements in each row.

```
summary(colors,2)
```

blue	green	red
1	1	1
1	1	1
1	1	1

## Input Arguments

### **a** — Nominal or ordinal array

nominal array | ordinal array

Nominal or ordinal array, specified as a nominal or ordinal array object created using `nominal` or `ordinal`.

### **dim** — Dimension along which to count

positive integer value

Dimension along which to count the number of elements in each level, specified as a positive integer value. For example, if the dimension is 1, then `levelcounts` counts along each column, while if the dimension is 2, then `levelcounts` counts along each row.

Data Types: `double` | `single`

## See Also

`nominal` | `ordinal` | `summary`

## Topics

`nominal`  
`ordinal`

Introduced in R2007a

## leverage

Leverage

### Syntax

```
h = leverage(data)
h = leverage(data, model)
```

### Description

`h = leverage(data)` finds the leverage of each row (point) in the matrix `data` for a linear additive regression model.

`h = leverage(data, model)` finds the leverage on a regression, using a specified model type, where `model` can be one of the following:

- 'linear' - includes constant and linear terms
- 'interaction' - includes constant, linear, and cross product terms
- 'quadratic' - includes interactions and squared terms
- 'purequadratic' - includes constant, linear, and squared terms

Leverage is a measure of the influence of a given observation on a regression due to its location in the space of the inputs.

### Examples

One rule of thumb is to compare the leverage to  $2p/n$  where  $n$  is the number of observations and  $p$  is the number of parameters in the model. For the Hald data set this value is 0.7692.

```
load hald
h = max(leverage(ingredients, 'linear'))
h =
    0.7004
```

Since  $0.7004 < 0.7692$ , there are no high leverage points using this rule.

## Algorithms

```
[Q,R] = qr(x2fx(data, 'model'));  
leverage = (sum(Q' .* Q'))'
```

## References

- [1] Goodall, C. R. "Computation Using the QR Decomposition." *Handbook in Statistics*. Vol. 9, Amsterdam: Elsevier/North-Holland, 1993.

## See Also

### Topics

regstats

Introduced before R2006a

## lhsdesign

Latin hypercube sample

### Syntax

```
X = lhsdesign(n,p)
X = lhsdesign(...,'smooth','off')
X = lhsdesign(...,'criterion',criterion)
X = lhsdesign(...,'iterations',k)
```

### Description

`X = lhsdesign(n,p)` returns an  $n$ -by- $p$  matrix,  $X$ , containing a latin hypercube sample of  $n$  values on each of  $p$  variables. For each column of  $X$ , the  $n$  values are randomly distributed with one from each interval  $(0, 1/n)$ ,  $(1/n, 2/n)$ , ...,  $(1-1/n, 1)$ , and they are randomly permuted.

`X = lhsdesign(...,'smooth','off')` produces points at the midpoints of the above intervals:  $0.5/n, 1.5/n, \dots, 1-0.5/n$ . The default is 'on'.

`X = lhsdesign(...,'criterion',criterion)` iteratively generates latin hypercube samples to find the best one according to *criterion*, which can be 'none', 'maximin', or 'correlation'.

Criterion	Description
'none'	No iteration.
'maximin'	Maximize minimum distance between points. This is the default.
'correlation'	Reduce correlation.

`X = lhsdesign(...,'iterations',k)` iterates up to  $k$  times in an attempt to improve the design according to the specified criterion. The default is  $k = 5$ .

## See Also

haltonset | lhsnorm | sobolset | unifrnd

**Introduced before R2006a**

## lhsnorm

Latin hypercube sample from normal distribution

### Syntax

```
X = lhsnorm(mu, sigma, n)
X = lhsnorm(mu, sigma, n, flag)
[X, Z] = lhsnorm(...)
```

### Description

`X = lhsnorm(mu, sigma, n)` returns an  $n$ -by- $p$  matrix,  $X$ , containing a latin hypercube sample of size  $n$  from a  $p$ -dimensional multivariate normal distribution with mean vector,  $\mu$ , and covariance matrix,  $\sigma$ .

$X$  is similar to a random sample from the multivariate normal distribution, but the marginal distribution of each column is adjusted so that its sample marginal distribution is close to its theoretical normal distribution.

`X = lhsnorm(mu, sigma, n, flag)` controls the amount of smoothing in the sample. If *flag* is 'off', each column has points equally spaced on the probability scale. In other words, each column is a permutation of the values  $G(0.5/n)$ ,  $G(1.5/n)$ , ...,  $G(1-0.5/n)$ , where  $G$  is the inverse normal cumulative distribution for that column's marginal distribution. If *flag* is 'on' (the default), each column has points uniformly distributed on the probability scale. For example, in place of  $0.5/n$  you use a value having a uniform distribution on the interval  $(0/n, 1/n)$ .

`[X, Z] = lhsnorm(...)` also returns  $Z$ , the original multivariate normal sample before the marginals are adjusted to obtain  $X$ .



## References

- [1] Stein, M. “Large sample properties of simulations using latin hypercube sampling.” *Technometrics*. Vol. 29, No. 2, 1987, pp. 143–151. Correction, Vol. 32, p. 367.

## See Also

lhsdesign | mvnrnd

Introduced before R2006a

## lillietest

Lilliefors test

### Syntax

```
h = lillietest(x)
h = lillietest(x,Name,Value)
[h,p] = lillietest(____)
[h,p,kstat,critval] = lillietest(____)
```

### Description

`h = lillietest(x)` returns a test decision for the null hypothesis that the data in vector `x` comes from a distribution in the normal family, against the alternative that it does not come from such a distribution, using a Lilliefors test. The result `h` is 1 if the test rejects the null hypothesis at the 5% significance level, and 0 otherwise.

`h = lillietest(x,Name,Value)` returns a test decision with additional options specified by one or more name-value pair arguments. For example, you can test the data against a different distribution family, change the significance level, or calculate the  $p$ -value using a Monte Carlo approximation.

`[h,p] = lillietest(____)` also returns the  $p$ -value `p`, using any of the input arguments from the previous syntaxes.

`[h,p,kstat,critval] = lillietest(____)` also returns the test statistic `kstat` and the critical value `critval` for the test.

### Examples

## Test for a Normal Distribution

Load the sample data. Test the null hypothesis that car mileage, in miles per gallon (MPG), follows a normal distribution across different makes of cars.

```
load carbig
[h,p,k,c] = lillietest(MPG)

Warning: P is less than the smallest tabulated value, returning 0.001.

h =

     1

p =

 1.0000e-03

k =

 0.0789

c =

 0.0451
```

The test statistic  $k$  is greater than the critical value  $c$ , so `lillietest` returns a result of  $h = 1$  to indicate rejection of the null hypothesis at the default 5% significance level. The warning indicates that the returned  $P$ -value is the smallest value in the table of precomputed values. To find a more accurate  $P$ -value, use `MCTol` to run a Monte Carlo approximation.

## Test the Hypothesis at Different Significance Levels

Load the sample data. Create a vector containing the first column of the students' exam grades data.

```
load examgrades
x = grades(:,1);
```

Test the null hypothesis that the sample data comes from a normal distribution at the 1% significance level.

```
[h,p] = lillietest(x, 'Alpha', 0.01)
```

```
h =  
  
    0
```

```
p =  
  
    0.0348
```

The returned value of  $h = 0$  indicates that `lillietest` does not reject the null hypothesis at the 1% significance level.

### Test for an Exponential Distribution

Load the sample data. Test the null hypothesis that car mileage, in miles per gallon (MPG), follows an exponential distribution across different makes of cars.

```
load carbig  
h = lillietest(MPG, 'Distr', 'exp')
```

```
h =  
  
    1
```

The returned value of  $h = 1$  indicates that `lillietest` rejects the null hypothesis at the default 5% significance level.

### Determine the p-value Using Monte Carlo Approximation

Load the sample data. Test the null hypothesis that car mileage, in miles per gallon (MPG), follows a normal distribution across different makes of cars. Determine the *P*-value

using a Monte Carlo approximation with a maximum Monte Carlo standard error of  $1e-4$ .

```
load carbig
[h,p] = lillietest(MPG,'MCTol',1e-4)
```

```
h =
```

```
    1
```

```
p =
```

```
 8.3333e-06
```

The returned value of `h = 1` indicates that `lillietest` rejects the null hypothesis that the data comes from a normal distribution at the 5% significance level.

## Input Arguments

### **x** — Sample data

vector

Sample data, specified as a vector.

Data Types: `single` | `double`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'Distr','exp','Alpha',0.01` tests the null hypothesis that the population distribution belongs to the exponential distribution family at the 1% significance level.

### **Alpha** — Significance level

0.05 (default) | scalar value in the range (0,1)

Significance level of the hypothesis test, specified as the comma-separated pair consisting of 'Alpha' and a scalar value in the range (0,1).

- If `MCTol` is not used, Alpha must be in the range [0.001,0.50].
- If `MCTol` is used, Alpha must be in the range (0,1).

Example: 'Alpha',0.01

Data Types: single | double

### **Distr** — Distribution family

'norm' (default) | 'exp' | 'ev'

Distribution family for the hypothesis test, specified as the comma-separated pair consisting of 'Distr' and one of the following.

'norm'	Normal distribution
'exp'	Exponential distribution
'ev'	Extreme value distribution

Example: 'Distr','exp'

### **MCTol** — Maximum Monte Carlo standard error

scalar value in the range (0,1)

Maximum Monte Carlo standard error on page 32-3554 for  $p$ , the  $p$ -value of the test, specified as the comma-separated pair consisting of 'MCTol' and a scalar value in the range (0,1).

Example: 'MCTol',0.001

Data Types: single | double

## Output Arguments

### **h** — Hypothesis test result

1 | 0

Hypothesis test result, returned as a logical value.

- If  $h = 1$ , this indicates the rejection of the null hypothesis at the Alpha significance level.

- If  $h = 0$ , this indicates a failure to reject the null hypothesis at the Alpha significance level.

**p — p-value**

scalar value in the range (0,1)

$p$ -value of the test, returned as a scalar value in the range (0,1).  $p$  is the probability of observing a test statistic as extreme as, or more extreme than, the observed value under the null hypothesis. Small values of  $p$  cast doubt on the validity of the null hypothesis.

- If `MCTol` is not used,  $p$  is computed using inverse interpolation into the table of critical values, and is returned as a scalar value in the range [0.001,0.50]. `lillietest` warns when  $p$  is not found within the tabulated range and returns either the smallest or largest tabulated value.
- If `MCTol` is used, `lillietest` conducts a Monte Carlo simulation to compute a more accurate  $p$ -value, and  $p$  is returned as a scalar value in the range (0,1).

**kstat — Test statistic**

nonnegative scalar value

Test statistic, returned as a nonnegative scalar value.

**critval — Critical value**

nonnegative scalar value

Critical value for the hypothesis test, returned as a nonnegative scalar value.

## Definitions

### Lilliefors Test

The Lilliefors test is a two-sided goodness-of-fit test suitable when the parameters of the null distribution are unknown and must be estimated. This is in contrast to the one-sample Kolmogorov-Smirnov test, which requires the null distribution to be completely specified.

The Lilliefors test statistic is:

$$D^* = \max_x |F(x) - G(x)|,$$

where  $\hat{F}(x)$  is the empirical cdf of the sample data and  $G(x)$  is the cdf of the hypothesized distribution with estimated parameters equal to the sample parameters.

`lillietest` can be used to test whether the data vector `x` has a lognormal or Weibull distribution by applying a transformation to the data vector and running the appropriate Lilliefors test:

- To test `x` for a lognormal distribution, test if `log(x)` has a normal distribution.
- To test `x` for a Weibull distribution, test if `log(x)` has an extreme value distribution.

The Lilliefors test cannot be used when the null hypothesis is not a location-scale family of distributions.

## Monte Carlo Standard Error

The Monte Carlo standard error is the error due to simulating the  $p$ -value.

The Monte Carlo standard error is calculated as:

$$SE = \sqrt{\frac{(\hat{p})(1 - \hat{p})}{\text{mcreps}}},$$

where  $\hat{p}$  is the estimated  $p$ -value of the hypothesis test, and `mcreps` is the number of Monte Carlo replications performed.

The number of Monte Carlo replications, `mcreps`, is determined such that the Monte Carlo standard error for  $\hat{p}$  less than the value specified for `MCTol`.

## Algorithms

To compute the critical value for the hypothesis test, `lillietest` interpolates into a table of critical values pre-computed using Monte Carlo simulation for sample sizes less than 1000 and significance levels between 0.001 and 0.50. The table used by `lillietest` is larger and more accurate than the table originally introduced by Lilliefors. If a more accurate  $p$ -value is desired, or if the desired significance level is less than 0.001 or greater than 0.50, the `MCTol` input argument can be used to run a Monte Carlo simulation to calculate the  $p$ -value more exactly.



When the computed value of the test statistic is greater than the critical value, `lillietest` rejects the null hypothesis at significance level `Alpha`.

`lillietest` treats NaN values in `x` as missing values and ignores them.

## References

- [1] Conover, W. J. *Practical Nonparametric Statistics*. Hoboken, NJ: John Wiley & Sons, Inc., 1980.
- [2] Lilliefors, H. W. “On the Kolmogorov-Smirnov test for the exponential distribution with mean unknown.” *Journal of the American Statistical Association*. Vol. 64, 1969, pp. 387–389.
- [3] Lilliefors, H. W. “On the Kolmogorov-Smirnov test for normality with mean and variance unknown.” *Journal of the American Statistical Association*. Vol. 62, 1967, pp. 399–402.

## See Also

`adtest` | `cdfplot` | `jbtest` | `kstest` | `kstest2`

Introduced before R2006a

## LinearModel class

### Superclasses:

Linear regression model class

## Description

An object comprising training data, model description, diagnostic information, and fitted coefficients for a linear regression. Predict model responses with the `predict` or `feval` methods.

## Construction

`mdl = fitlm(tbl)` or `mdl = fitlm(X,y)` create a linear model of a table or dataset array `tbl`, or of the responses `y` to a data matrix `X`. For details, see `fitlm`.

`mdl = stepwiselm(tbl)` or `mdl = stepwiselm(X,y)` create a linear model of a table or dataset array `tbl`, or of the responses `y` to a data matrix `X`, with unimportant predictors excluded. For details, see `stepwiselm`.

## Input Arguments

### **tbl** — Input data

table | dataset array

Input data, specified as a table or dataset array. When `modelspec` is a formula, it specifies the variables to be used as the predictors and response. Otherwise, if you do not specify the predictor and response variables, the last variable is the response variable and the others are the predictor variables by default.

Predictor variables can be numeric, or any grouping variable type, such as logical or categorical (see “Grouping Variables” on page 2-59). The response must be numeric or logical.

To set a different column as the response variable, use the `ResponseVar` name-value pair argument. To use a subset of the columns as predictors, use the `PredictorVars` name-value pair argument.

### **x — Predictor variables**

matrix

Predictor variables, specified as an  $n$ -by- $p$  matrix, where  $n$  is the number of observations and  $p$  is the number of predictor variables. Each column of `X` represents one variable, and each row represents one observation.

By default, there is a constant term in the model, unless you explicitly remove it, so do not include a column of 1s in `X`.

Data Types: `single` | `double` | `logical`

### **y — Response variable**

vector

Response variable, specified as an  $n$ -by-1 vector, where  $n$  is the number of observations. Each entry in `y` is the response for the corresponding row of `X`.

Data Types: `single` | `double` | `logical`

## Properties

### **CoefficientCovariance — Covariance matrix of coefficient estimates**

numeric matrix

Covariance matrix of coefficient estimates, stored as a  $p$ -by- $p$  matrix of numeric values.  $p$  is the number of coefficients in the fitted model.

### **CoefficientNames — Coefficient names**

cell array of character vectors

Coefficient names, stored as a cell array of character vectors containing a label for each coefficient.

### **Coefficients — Coefficient values**

table

Coefficient values, stored as a table. `Coefficients` has one row for each coefficient and the following columns:

- `Estimate` — Estimated coefficient value
- `SE` — Standard error of the estimate
- `tStat` —  $t$  statistic for a test that the coefficient is zero
- `pValue` —  $p$ -value for the  $t$  statistic

To obtain any of these columns as a vector, index into the property using dot notation. For example, in `mdl` the estimated coefficient vector is

```
beta = mdl.Coefficients.Estimate
```

Use `coefTest` to perform other tests on the coefficients.

#### **DFE — Degrees of freedom for error**

positive integer value

Degrees of freedom for error (residuals), equal to the number of observations minus the number of estimated coefficients, stored as a positive integer value.

#### **Diagnostics — Diagnostic values**

table

Diagnostic values, stored as a table with the same number of rows as the input data (`tbl` or `X`). `Diagnostics` contains diagnostics helpful in finding outliers and influential observations. Many diagnostics describe the effect on the fit of deleting single observations. `Diagnostics` contains the following fields.

Field	Meaning	Utility
Leverage	Diagonal elements of HatMatrix	Leverage indicates to what extent the predicted value for an observation is determined by the observed value for that observation. A value close to 1 indicates that the prediction is largely determined by that observation, with little contribution from the other observations. A value close to 0 indicates the fit is largely determined by the other observations. For a model with $P$ coefficients and $N$ observations, the average value of Leverage is $P/N$ . An observation with Leverage larger than $2 * P/N$ can be regarded as having high leverage.
CooksDistance	Cook's measure of scaled change in fitted values	CooksDistance is a measure of scaled change in fitted values. An observation with CooksDistance larger than three times the mean Cook's distance can be an outlier.
Dffits	Delete-1 scaled differences in fitted values vs. observation number	Dffits is the scaled change in the fitted values for each observation that would result from excluding that observation from the fit. Values with an absolute value larger than $2 * \sqrt{P/N}$ may be considered influential.
S2_i	Delete-1 variance vs. observation number	S2_i is a set of residual variance estimates obtained by deleting each observation in turn. These can be compared with the value of the MSE property.
CovRatio	Delete-1 ratio of determinant of covariance vs. observation number	CovRatio is the ratio of the determinant of the coefficient covariance matrix with each observation deleted in turn to the determinant of the covariance matrix for the full model. Values larger than $1 + 3 * P/N$ or smaller than $1 - 3 * P/N$ indicate influential points.
Dfbetas	Delete-1 scaled differences in covariance estimates vs. observation number	Dfbetas is an $N$ -by- $P$ matrix of the scaled change in the coefficient estimates that would result from excluding each observation in turn. Values larger than $3 / \sqrt{N}$ in absolute value indicate that the observation has a large influence on the corresponding coefficient.

Field	Meaning	Utility
HatMatrix	Projection matrix to compute fitted from observed responses	HatMatrix is an N-by-N matrix such that $\text{Fitted} = \text{HatMatrix} * Y$ , where Y is the response vector and Fitted is the vector of fitted response values.

Rows not used in the fit because of missing values (in `ObservationInfo.Missing`) contain NaN values.

Rows not used in the fit because of excluded values (in `ObservationInfo.Excluded`) contain NaN values, with the following exception: Delete-1 diagnostics refer to the statistic with and without that observation (row) included in the fit. These diagnostics help identify important observations.

#### **Fitted** — Fitted response values based on input data

vector of numeric values

Fitted (predicted) response values based on input data, stored as an  $n$ -by-1 vector of numeric values.  $n$  is the number of observations in the input data. Use `predict` to compute predictions for other predictor values, or to compute confidence bounds on Fitted.

#### **Formula** — Model information

`LinearFormula` object | `NonLinearFormula` object

Model information, stored as a `LinearFormula` object or `NonLinearFormula` object. If you fit a linear or generalized linear regression model, then `Formula` is a `LinearFormula` object. If you fit a nonlinear regression model, then `Formula` is a `NonLinearFormula` object.

#### **LogLikelihood** — Log likelihood

numeric value

Log likelihood of the model distribution at the response values, stored as a numeric value. The mean is fitted from the model, and other parameters are estimated as part of the model fit.

#### **ModelCriterion** — Criterion for model comparison

structure

Criterion for model comparison, stored as a structure with the following fields:

- **AIC** — Akaike information criterion
- **AIC<sub>c</sub>** — Akaike information criterion corrected for sample size
- **BIC** — Bayesian information criterion
- **CAIC** — Consistent Akaike information criterion

To obtain any of these values as a scalar, index into the property using dot notation. For example, in a model `mdl`, the AIC value `aic` is:

```
aic = mdl.ModelCriterion.AIC
```

### **MSE — Mean squared error**

numeric value

Mean squared error (residuals), stored as a numeric value. Mean square error is calculated as  $MSE = SSE / DFE$ , where  $MSE$  is the mean square error,  $SSE$  is the sum of squared errors, and  $DFE$  is the degrees of freedom.

### **NumCoefficients — Number of model coefficients**

positive integer

Number of model coefficients, stored as a positive integer. `NumCoefficients` includes coefficients that are set to zero when the model terms are rank deficient.

### **NumEstimatedCoefficients — Number of estimated coefficients**

positive integer

Number of estimated coefficients in the model, stored as a positive integer. `NumEstimatedCoefficients` does not include coefficients that are set to zero when the model terms are rank deficient. `NumEstimatedCoefficients` is the degrees of freedom for regression.

### **NumObservations — Number of observations**

positive integer

Number of observations the fitting function used in fitting, stored as a positive integer. This is the number of observations supplied in the original table, dataset, or matrix, minus any excluded rows (set with the `Excluded` name-value pair) or rows with missing values.

### **NumPredictors — Number of predictor variables**

positive integer

Number of predictor variables used to fit the model, stored as a positive integer.

**NumVariables — Number of variables**

positive integer

Number of variables in the input data, stored as a positive integer. `NumVariables` is the number of variables in the original table or dataset, or the total number of columns in the predictor matrix and response vector when the fit is based on those arrays. It includes variables, if any, that are not used as predictors or as the response.

**ObservationInfo — Observation information**

table

Observation information, stored as a  $n$ -by-4 table, where  $n$  is equal to the number of rows of input data. The four columns of `ObservationInfo` contain the following:

Field	Description
Weights	Observation weights. Default is all 1.
Excluded	Logical value, 1 indicates an observation that you excluded from the fit with the <code>Exclude</code> name-value pair.
Missing	Logical value, 1 indicates a missing value in the input. Missing values are not used in the fit.
Subset	Logical value, 1 indicates the observation is not excluded or missing, so is used in the fit.

**ObservationNames — Observation names**

cell array of character vectors

Observation names, stored as a cell array of character vectors containing the names of the observations used in the fit.

- If the fit is based on a table or dataset containing observation names, `ObservationNames` uses those names.
- Otherwise, `ObservationNames` is an empty cell array

**PredictorNames — Names of predictors used to fit the model**

cell array of character vectors

Names of predictors used to fit the model, stored as a cell array of character vectors.



**Residuals — Residuals for fitted model**

table

Residuals for fitted model, stored as a table that contains one row for each observation and the following columns.

Field	Description
Raw	Observed minus fitted values.
Pearson	Raw residuals divided by RMSE.
Standardized	Raw residuals divided by their estimated standard deviation.
Studentized	Residual divided by an independent estimate of the residual standard deviation. The residual for observation $i$ is divided by an estimate of the error standard deviation based on all observations except for observation $i$ .

To obtain any of these columns as a vector, index into the property using dot notation. For example, in a model `mdl`, the ordinary raw residual vector `r` is:

```
r = mdl.Residuals.Raw
```

Rows not used in the fit because of missing values (in `ObservationInfo.Missing`) contain NaN values.

Rows not used in the fit because of excluded values (in `ObservationInfo.Excluded`) contain NaN values, with the following exceptions:

- `raw` contains the difference between the observed and predicted values.
- `standardized` is the residual, standardized in the usual way.
- `studentized` matches the standardized values because this residual is not used in the estimate of the residual standard deviation.

**ResponseName — Response variable name**

character vector

Response variable name, stored as a character vector.

**RMSE — Root mean squared error**

numeric value

Root mean squared error (residuals), stored as a numeric value. The root mean squared error (RMSE) is equal to  $RMSE = \sqrt{MSE}$ , where  $MSE$  is the mean squared error.

### **Robust** — Robust fit information

structure

Robust fit information, stored as a structure with the following fields:

Field	Description
WgtFun	Robust weighting function, such as 'bisquare' (see <code>robustfit</code> )
Tune	Value specified for tuning parameter (can be [])
Weights	Vector of weights used in final iteration of robust fit. This field is empty for compacted <code>CompactLinearModel</code> models.

This structure is empty unless `fitlm` constructed the model using robust regression.

### **Rsqared** — R-squared value for the model

structure

R-squared value for the model, stored as a structure.

For a linear or nonlinear model, `Rsqared` is a structure with two fields:

- `Ordinary` — Ordinary (unadjusted) R-squared
- `Adjusted` — R-squared adjusted for the number of coefficients

For a generalized linear model, `Rsqared` is a structure with five fields:

- `Ordinary` — Ordinary (unadjusted) R-squared
- `Adjusted` — R-squared adjusted for the number of coefficients
- `LLR` — Log-likelihood ratio
- `Deviance` — Deviance
- `AdjGeneralized` — Adjusted generalized R-squared

The R-squared value is the proportion of total sum of squares explained by the model. The ordinary R-squared value relates to the `SSR` and `SST` properties:

$$Rsqared = SSR/SST = 1 - SSE/SST.$$

To obtain any of these values as a scalar, index into the property using dot notation. For example, the adjusted R-squared value in `mdl` is

```
r2 = mdl.Rsquared.Adjusted
```

### **SSE — Sum of squared errors**

numeric value

Sum of squared errors (residuals), stored as a numeric value.

The Pythagorean theorem implies

$$SST = SSE + SSR.$$

### **SSR — Regression sum of squares**

numeric value

Regression sum of squares, stored as a numeric value. The regression sum of squares is equal to the sum of squared deviations of the fitted values from their mean.

The Pythagorean theorem implies

$$SST = SSE + SSR.$$

### **SST — Total sum of squares**

numeric value

Total sum of squares, stored as a numeric value. The total sum of squares is equal to the sum of squared deviations of  $y$  from  $\text{mean}(y)$ .

The Pythagorean theorem implies

$$SST = SSE + SSR.$$

### **Steps — Stepwise fitting information**

structure

Stepwise fitting information, stored as a structure with the following fields.

Field	Description
Start	Formula representing the starting model

Field	Description
Lower	Formula representing the lower bound model, these terms that must remain in the model
Upper	Formula representing the upper bound model, model cannot contain more terms than Upper
Criterion	Criterion used for the stepwise algorithm, such as 'sse'
PEnter	Value of the parameter, such as 0.05
PRemove	Value of the parameter, such as 0.10
History	Table representing the steps taken in the fit

The `History` table has one row for each step including the initial fit, and the following variables (columns).

Field	Description
Action	Action taken during this step, one of: <ul style="list-style-type: none"> <li>'Start' — First step</li> <li>'Add' — A term is added</li> <li>'Remove' — A term is removed</li> </ul>
TermName	<ul style="list-style-type: none"> <li>'Start' step: The starting model specification</li> <li>'Add' or 'Remove' steps: The term moved in that step</li> </ul>
Terms	Terms matrix (see <code>modelspec</code> of <code>fitlm</code> )
DF	Regression degrees of freedom after this step
delDF	Change in regression degrees of freedom from previous step (negative for steps that remove a term)
Deviance	Deviance (residual sum of squares) at that step
FStat	$F$ statistic that led to this step
PValue	$p$ -value of the $F$ statistic

The structure is empty unless you use `stepwiselm` or `stepwiseglm` to fit the model.

### **variableInfo** — Information about input variables

table

Information about input variables contained in `Variables`, stored as a table with one row for each model term and the following columns.

Field	Description
Class	Character vector giving variable class, such as 'double'
Range	Cell array giving variable range: <ul style="list-style-type: none"> <li>• Continuous variable — Two-element vector <math>[min, max]</math>, the minimum and maximum values</li> <li>• Categorical variable — Cell array of distinct variable values</li> </ul>
InModel	Logical vector, where <code>true</code> indicates the variable is in the model
IsCategorical	Logical vector, where <code>true</code> indicates a categorical variable

**VariableNames** — Names of variables used in fit

cell array of character vectors

Names of variables used in fit, stored as a cell array of character vectors.

- If the fit is based on a table or dataset, this property provides the names of the variables in that table or dataset.
- If the fit is based on a predictor matrix and response vector, `VariableNames` is the values in the `VarNames` name-value pair of the fitting method.
- Otherwise the variables have the default fitting names.

**Variables** — Data used to fit the model

table

Data used to fit the model, stored as a table. `Variables` contains both observation and response values. If the fit is based on a table or dataset array, `Variables` contains all of the data from that table or dataset array. Otherwise, `Variables` is a table created from the input data matrix  $X$  and response vector  $y$ .

## Methods

<code>addTerms</code>	Add terms to linear regression model
<code>compact</code>	Compact linear regression model
<code>dwtest</code>	Durbin-Watson test of linear model
<code>fit</code>	Create linear regression model
<code>plot</code>	Scatter plot or added variable plot of linear model
<code>plotAdded</code>	Added variable plot or leverage plot for linear model
<code>plotAdjustedResponse</code>	Adjusted response plot for linear regression model
<code>plotDiagnostics</code>	Plot diagnostics of linear regression model
<code>plotResiduals</code>	Plot residuals of linear regression model
<code>removeTerms</code>	Remove terms from linear model
<code>step</code>	Improve linear regression model by adding or removing terms
<code>stepwise</code>	Create linear regression model by stepwise regression

## Inherited Methods

<code>anova</code>	Analysis of variance for linear model
<code>coefCI</code>	Confidence intervals of coefficient estimates of linear model
<code>coefTest</code>	Linear hypothesis test on linear regression model coefficients
<code>disp</code>	Display linear regression model
<code>feval</code>	Evaluate linear regression model prediction
<code>plotEffects</code>	Plot main effects of each predictor in linear regression model
<code>plotInteraction</code>	Plot interaction effects of two predictors in linear regression model
<code>plotSlice</code>	Plot of slices through fitted linear regression surface
<code>predict</code>	Predict response of linear regression model
<code>random</code>	Simulate responses for linear regression model

## Copy Semantics

Value. To learn how value classes affect copy operations, see [Copying Objects \(MATLAB\)](#).

## Examples

### Linear Regression Model of Matrix Data

Fit a linear model of the Hald data.

Load the data.

```
load hald
X = ingredients; % Predictor variables
y = heat; % Response
```

Fit a default linear model to the data.

```
mdl = fitlm(X,y)
```

```
mdl =
```

```
Linear regression model:
  y ~ 1 + x1 + x2 + x3 + x4
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	62.405	70.071	0.8906	0.39913
x1	1.5511	0.74477	2.0827	0.070822
x2	0.51017	0.72379	0.70486	0.5009
x3	0.10191	0.75471	0.13503	0.89592
x4	-0.14406	0.70905	-0.20317	0.84407

```
Number of observations: 13, Error degrees of freedom: 8
Root Mean Squared Error: 2.45
R-squared: 0.982, Adjusted R-Squared 0.974
F-statistic vs. constant model: 111, p-value = 4.76e-07
```

### Linear Regression with Categorical Predictor

Fit a model of a table that contains a categorical predictor.

Load the `carsmall` data.

```
load carsmall
```

Construct a table containing continuous predictor variable `Weight`, nominal predictor variable `Year`, and response variable `MPG`.

```
tbl = table(MPG,Weight);  
tbl.Year = nominal(Model_Year);
```

Create a fitted model of `MPG` as a function of `Year`, `Weight`, and `Weight^2`. (You don't have to include `Weight` explicitly in your formula because it is a lower-order term of `Weight^2`) and is included automatically.

```
mdl = fitlm(tbl,'MPG ~ Year + Weight^2')
```

```
mdl =
```

Linear regression model:

```
MPG ~ 1 + Weight + Year + Weight^2
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	54.206	4.7117	11.505	2.6648e-19
Weight	-0.016404	0.0031249	-5.2493	1.0283e-06
Year_76	2.0887	0.71491	2.9215	0.0044137
Year_82	8.1864	0.81531	10.041	2.6364e-16
Weight^2	1.5573e-06	4.9454e-07	3.149	0.0022303

Number of observations: 94, Error degrees of freedom: 89

Root Mean Squared Error: 2.78

R-squared: 0.885, Adjusted R-Squared 0.88

F-statistic vs. constant model: 172, p-value = 5.52e-41

`fitlm` creates two dummy (indicator) variables for the nominal variate, `Year`. The dummy variable `Year_76` takes the value 1 if model year is 1976 and takes the value 0 if it is not. The dummy variable `Year_82` takes the value 1 if model year is 1982 and takes the value 0 if it is not. And the year 1970 is the reference year. The corresponding model is



$$\hat{MPG} = 54.206 - 0.0164(\textit{Weight}) + 2.0887(\textit{Year}_{.76}) + 8.1864(\textit{Year}_{.82}) + 1.557e - 06(\textit{Weight}^2)$$

### Fit a Robust Linear Regression Model

Fit a linear regression model using a robust fitting method.

Load the sample data.

```
load hald
```

The `hald` data measures the effect of cement composition on its hardening heat. The matrix `ingredients` contains the percent composition of four chemicals present in the cement. The array `heat` contains the heat of hardening after 180 days for each cement sample.

Fit a robust linear model to the data.

```
mdl = fitlm(ingredients,heat,'linear','RobustOpts','on')
```

```
mdl =
```

```
Linear regression model (robust fit):
  y ~ 1 + x1 + x2 + x3 + x4
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	60.09	75.818	0.79256	0.4509
x1	1.5753	0.80585	1.9548	0.086346
x2	0.5322	0.78315	0.67957	0.51596
x3	0.13346	0.8166	0.16343	0.87424
x4	-0.12052	0.7672	-0.15709	0.87906

```
Number of observations: 13, Error degrees of freedom: 8
Root Mean Squared Error: 2.65
```

R-squared: 0.979, Adjusted R-Squared 0.969  
F-statistic vs. constant model: 94.6, p-value = 9.03e-07

- “Linear Regression Workflow” on page 11-43
- “Robust Regression versus Standard Least-Squares Fit” on page 11-133

## Definitions

### Hat Matrix

The hat matrix  $H$  is defined in terms of the data matrix  $X$ :

$$H = X(X^T X)^{-1} X^T.$$

The diagonal elements  $h_{ii}$  satisfy

$$0 \leq h_{ii} \leq 1$$

$$\sum_{i=1}^n h_{ii} = p,$$

where  $n$  is the number of observations (rows of  $X$ ), and  $p$  is the number of coefficients in the regression model.

### Leverage

The leverage of observation  $i$  is the value of the  $i$ th diagonal term,  $h_{ii}$ , of the hat matrix  $H$ . Because the sum of the leverage values is  $p$  (the number of coefficients in the regression model), an observation  $i$  can be considered to be an outlier if its leverage substantially exceeds  $p/n$ , where  $n$  is the number of observations.

### Cook's Distance

Cook's distance is the scaled change in fitted values. Each element in `CooksDistance` is the normalized change in the vector of coefficients due to the deletion of an observation. The Cook's distance,  $D_i$ , of observation  $i$  is

$$D_i = \frac{\sum_{j=1}^n (\hat{y}_j - \hat{y}_{j(i)})^2}{p \text{MSE}},$$

where

- $\hat{y}_j$  is the  $j$ th fitted response value.
- $\hat{y}_{j(i)}$  is the  $j$ th fitted response value, where the fit does not include observation  $i$ .
- $MSE$  is the mean squared error.
- $p$  is the number of coefficients in the regression model.

Cook's distance is algebraically equivalent to the following expression:

$$D_i = \frac{r_i^2}{p MSE} \left( \frac{h_{ii}}{(1-h_{ii})^2} \right),$$

where  $r_i$  is the  $i$ th residual, and  $h_{ii}$  is the  $i$ th leverage value.

`CooksDistance` is an  $n$ -by-1 column vector in the `Diagnostics` table of the `LinearModel` object.

## Algorithms

The main fitting algorithm is QR decomposition. For robust fitting, the algorithm is `robustfit`.

## Alternatives

To remove redundant predictors in linear regression using lasso or elastic net, use the `lasso` function.

To regularize a regression with correlated terms using ridge regression, use the `ridge` or `lasso` functions.

To regularize a regression with correlated terms using partial least squares, use the `plsregress` function.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Only the `predict` and `random` functions support code generation.
- When you fit a model by using `fitlm` or `stepwiselm`, you cannot supply training data in a table that contains at least one categorical predictor, and you cannot use the `'CategoricalVars'` name-value pair argument. Code generation does not support categorical predictors. To dummy-code variables that you want treated as categorical, preprocess the categorical data by using `dummyvar` before fitting the model.

### See Also

`fitlm` | `plotPartialDependence` | `stepwiselm`

### Topics

“Linear Regression Workflow” on page 11-43

“Robust Regression versus Standard Least-Squares Fit” on page 11-133

“Linear Regression” on page 11-11

**Introduced in R2012a**

# LinearMixedModel class

Linear mixed-effects model class

## Description

A `LinearMixedModel` object represents a model of a response variable with fixed and random effects. It comprises data, a model description, fitted coefficients, covariance parameters, design matrices, residuals, residual plots, and other diagnostic information for a linear mixed-effects model. You can predict model responses with the `predict` function and generate random data at new design points using the `random` function.

## Construction

You can fit a linear mixed-effects model using `fitlme(tbl, formula)` if your data is in a table or dataset array. Alternatively, if your model is not easily described using a formula, you can create matrices to define the fixed and random effects, and fit the model using `fitlmematrix(X, y, Z, G)`.

## Input Arguments

### **tbl** — Input data

table | dataset array

Input data, which includes the response variable, predictor variables, and grouping variables, specified as a table or dataset array. The predictor variables can be continuous or grouping variables (see “Grouping Variables” on page 2-59). You must specify the model for the variables using `formula`.

Data Types: table

### **formula** — Formula for model specification

character vector of the form 'y ~ fixed + (random1|grouping1) + ... + (randomR|groupingR)'

Formula for model specification, specified as a character vector of the form `'y ~ fixed + (random1|grouping1) + ... + (randomR|groupingR)'`. For a full description, see “Formula” on page 32-3593.

Example: `'y ~ treatment +(1|block)'`

### **x — Fixed-effects design matrix**

*n*-by-*p* matrix

Fixed-effects design matrix, specified as an *n*-by-*p* matrix, where *n* is the number of observations, and *p* is the number of fixed-effects predictor variables. Each row of *X* corresponds to one observation, and each column of *X* corresponds to one variable.

Data Types: `single` | `double`

### **y — Response values**

*n*-by-1 vector

Response values, specified as an *n*-by-1 vector, where *n* is the number of observations.

Data Types: `single` | `double`

### **z — Random-effects design**

*n*-by-*q* matrix | cell array of *R* *n*-by-*q*(*r*) matrices, *r* = 1, 2, ..., *R*

Random-effects design, specified as either of the following.

- If there is one random-effects term in the model, then *Z* must be an *n*-by-*q* matrix, where *n* is the number of observations and *q* is the number of variables in the random-effects term.
- If there are *R* random-effects terms, then *Z* must be a cell array of length *R*. Each cell of *Z* contains an *n*-by-*q*(*r*) design matrix *Z*{*r*}, *r* = 1, 2, ..., *R*, corresponding to each random-effects term. Here, *q*(*r*) is the number of random effects term in the *r*th random effects design matrix, *Z*{*r*}.

Data Types: `single` | `double` | `cell`

### **g — Grouping variable or variables**

*n*-by-1 vector | cell array of *R* *n*-by-1 vectors

Grouping variable or variables on page 2-59, specified as either of the following.

- If there is one random-effects term, then  $G$  must be an  $n$ -by-1 vector corresponding to a single grouping variable with  $M$  levels or groups.

$G$  can be a categorical vector, numeric vector, character array, or cell array of character vectors.

- If there are multiple random-effects terms, then  $G$  must be a cell array of length  $R$ . Each cell of  $G$  contains a grouping variable  $G\{r\}$ ,  $r = 1, 2, \dots, R$ , with  $M(r)$  levels.

$G\{r\}$  can be a categorical vector, numeric vector, character array, or cell array of character vectors.

Data Types: `single` | `double` | `char` | `cell`

## Properties

### **Coefficients** — Fixed-effects coefficient estimates

dataset array

Fixed-effects coefficient estimates and related statistics, stored as a dataset array containing the following fields.

Name	Name of the term.
Estimate	Estimated value of the coefficient.
SE	Standard error of the coefficient.
tStat	$t$ -statistics for testing the null hypothesis that the coefficient is equal to zero.
DF	Degrees of freedom for the $t$ -test. Method to compute DF is specified by the 'DFMethod' name-value pair argument. <b>Coefficients</b> always uses the 'Residual' method for 'DFMethod'.
pValue	$p$ -value for the $t$ -test.
Lower	Lower limit of the confidence interval for coefficient. <b>Coefficients</b> always uses the 95% confidence level, i.e. 'alpha' is 0.05.

Upper

Upper limit of confidence interval for coefficient. `Coefficients` always uses the 95% confidence level, i.e. 'alpha' is 0.05.

You can change 'DFMethod' and 'alpha' while computing confidence intervals for or testing hypotheses involving fixed- and random-effects, using the `coefCI` and `coefTest` methods.

### **CoefficientCovariance** — Covariance of the estimated fixed-effects coefficients

*p*-by-*p* matrix

Covariance of the estimated fixed-effects coefficients of the linear mixed-effects model, stored as a *p*-by-*p* matrix, where *p* is the number of fixed-effects coefficients.

You can display the covariance parameters associated with the random effects using the `covarianceParameters` method.

Data Types: `double`

### **CoefficientNames** — Names of the fixed-effects coefficients

1-by-*p* cell array of character vectors

Names of the fixed-effects coefficients of a linear mixed-effects model, stored as a 1-by-*p* cell array of character vectors.

Data Types: `cell`

### **DFE** — Residual degrees of freedom

positive integer value

Residual degrees of freedom, stored as a positive integer value.  $DFE = n - p$ , where *n* is the number of observations, and *p* is the number of fixed-effects coefficients.

This corresponds to the 'Residual' method of calculating degrees of freedom in the `fixedEffects` and `randomEffects` methods.

Data Types: `double`

### **FitMethod** — Method used to fit the linear mixed-effects model

ML | REML

Method used to fit the linear mixed-effects model, stored as either of the following.



- ML, if the fitting method is maximum likelihood
- REML, if the fitting method is restricted maximum likelihood

Data Types: `char`

**Formula** — Specification of the fixed- and random-effects terms, and grouping variables  
object

Specification of the fixed-effects terms, random-effects terms, and grouping variables that define the linear mixed-effects model, stored as an object.

For more information on how to specify the model to fit using a formula, see “Formula” on page 32-3593.

**LogLikelihood** — Maximized log or restricted log likelihood  
scalar value

Maximized log likelihood or maximized restricted log likelihood of the fitted linear mixed-effects model depending on the fitting method you choose, stored as a scalar value.

Data Types: `double`

**ModelCriterion** — Model criterion  
dataset array

Model criterion to compare fitted linear mixed-effects models, stored as a dataset array with the following columns.

AIC	Akaike Information Criterion
BIC	Bayesian Information Criterion
Loglikelihood	Log likelihood value of the model
Deviance	-2 times the log likelihood of the model

If  $n$  is the number of observations used in fitting the model, and  $p$  is the number of fixed-effects coefficients, then for calculating AIC and BIC,

- The total number of parameters is  $nc + p + 1$ , where  $nc$  is the total number of parameters in the random-effects covariance excluding the residual variance
- The effective number of observations is
  - $n$ , when the fitting method is maximum likelihood (ML)

- $n - p$ , when the fitting method is restricted maximum likelihood (REML)

**MSE — ML or REML estimate**

positive scalar value

ML or REML estimate, based on the fitting method used for estimating  $\sigma^2$ , stored as a positive scalar value.  $\sigma^2$  is the residual variance or variance of the observation error term of the linear mixed-effects model.

Data Types: double

**NumCoefficients — Number of fixed-effects coefficients**

positive integer value

Number of fixed-effects coefficients in the fitted linear mixed-effects model, stored as a positive integer value.

Data Types: double

**NumEstimatedCoefficients — Number of estimated fixed-effects coefficients**

positive integer value

Number of estimated fixed-effects coefficients in the fitted linear mixed-effects model, stored as a positive integer value.

Data Types: double

**NumObservations — Number of observations**

positive integer value

Number of observations used in the fit, stored as a positive integer value. This is the number of rows in the table or dataset array, or the design matrices minus the excluded rows or rows with NaN values.

Data Types: double

**NumPredictors — Number of predictors**

positive integer value

Number of variables used as predictors in the linear mixed-effects model, stored as a positive integer value.

Data Types: double

**NumVariables** — Total number of variables

positive integer value

Total number of variables including the response and predictors, stored as a positive integer value.

- If the sample data is in a table or dataset array `tbl`, `NumVariables` is the total number of variables in `tbl` including the response variable.
- If the fit is based on matrix input, `NumVariables` is the total number of columns in the predictor matrix or matrices, and response vector.

`NumVariables` includes variables, if there are any, that are not used as predictors or as the response.

Data Types: `double`**ObservationInfo** — Information about the observations

table

Information about the observations used in the fit, stored as a table.

`ObservationInfo` has one row for each observation and the following four columns.

Weights	The value of the weighted variable for that observation. Default value is 1.
Excluded	<code>true</code> , if the observation was excluded from the fit using the 'Exclude' name-value pair argument, <code>false</code> , otherwise. 1 stands for <code>true</code> and 0 stands for <code>false</code> .
Missing	<code>true</code> , if the observation was excluded from the fit because any response or predictor value is missing, <code>false</code> , otherwise.  Missing values include <code>NaN</code> for numeric variables, empty cells for cell arrays, blank rows for character arrays, and the <code>&lt;undefined&gt;</code> value for categorical arrays.
Subset	<code>true</code> , if the observation was used in the fit, <code>false</code> , if it was not used because it is missing or excluded.

Data Types: `table`

**ObservationNames — Names of observations**

cell array of character vectors

Names of observations used in the fit, stored as a cell array of character vectors.

- If the data is in a table or dataset array, `tbl`, containing observation names, `ObservationNames` has those names.
- If the data is provided in matrices, or a table or dataset array without observation names, then `ObservationNames` is an empty cell array.

Data Types: `cell`

**PredictorNames — Names of predictors**

cell array of character vectors

Names of the variables that you use as predictors in the fit, stored as a cell array of character vectors that has the same length as `NumPredictors`.

Data Types: `cell`

**ResponseName — Names of response variable**

character vector

Name of the variable used as the response variable in the fit, stored as a character vector.

Data Types: `char`

**Rsqquared — Proportion of variability in the response explained by the fitted model**

structure

Proportion of variability in the response explained by the fitted model, stored as a structure. It is the multiple correlation coefficient or R-squared. `Rsqquared` has two fields.

`Ordinary`

R-squared value, stored as a scalar value in a structure. `Rsqquared.Ordinary = 1 - SSE./SST`

Adjusted

R-squared value adjusted for the number of fixed-effects coefficients, stored as a scalar value in a structure.

$$\text{Rsquared.Adjusted} = 1 - (\text{SSE.}/\text{SST}) * (\text{DFT.}/\text{DFE}),$$

where  $\text{DFE} = n - p$ ,  $\text{DFT} = n - 1$ , and  $n$  is the total number of observations,  $p$  is the number of fixed-effects coefficients.

Data Types: `struct`

### **SSE — Error sum of squares**

positive scalar value

Error sum of squares, that is, sum of the squared conditional residuals, stored as a positive scalar value.

$\text{SSE} = \text{sum}((y - F).^2)$ , where  $y$  is the response vector, and  $F$  is the fitted conditional response of the linear mixed-effects model. The conditional model has contributions from both fixed and random effects.

Data Types: `double`

### **SSR — Regression sum of squares**

positive scalar value

Regression sum of squares, that is, the sum of squares explained by the linear mixed-effects regression, stored as a positive scalar value. It is the sum of squared deviations of the conditional fitted values from their mean.

$\text{SSR} = \text{sum}((F - \text{mean}(F)).^2)$ , where  $F$  is the fitted conditional response of the linear mixed-effects model. The conditional model has contributions from both fixed and random effects.

Data Types: `double`

### **SST — Total sum of squares**

positive scalar value

Total sum of squares, that is, the sum of the squared deviations of the observed response values from their mean, stored as a positive scalar value.

$SST = \text{sum}((y - \text{mean}(y)).^2) = SSR + SSE$ , where  $y$  is the response vector.

Data Types: `double`

### **Variables — Variables**

`table`

Variables, stored as a table.

- If the fit is based on a table or dataset array `tbl`, then `Variables` is identical to `tbl`.
- If the fit is based on matrix input, then `Variables` is a table containing all the variables in the predictor matrix or matrices, and response variable.

Data Types: `table`

### **VariableInfo — Information about the variables**

`table`

Information about the variables used in the fit, stored as a table.

`VariableInfo` has one row for each variable and contains the following four columns.

<code>Class</code>	Class of the variable ('double', 'cell', 'nominal', and so on).
<code>Range</code>	Value range of the variable. <ul style="list-style-type: none"><li>• For a numerical variable, it is a two-element vector of the form <code>[min,max]</code>.</li><li>• For a cell or categorical variable, it is a cell or categorical array containing all unique values of the variable.</li></ul>
<code>InModel</code>	<code>true</code> , if the variable is a predictor in the fitted model.  <code>false</code> , if the variable is not in the fitted model.

IsCategorical

true, if the variable has a type that is treated as a categorical predictor, such as cell, logical, or categorical, or if it is specified as categorical by the 'Categorical' name-value pair argument of the fit method.

false, if it is a continuous predictor.

Data Types: table

**VariableNames — Names of the variables**

cell array of character vectors

Names of the variables used in the fit, stored as a cell array of character vectors.

- If sample data is in a table or dataset array `tbl`, `VariableNames` contains the names of the variables in `tbl`.
- If sample data is in matrix format, then `VariableInfo` includes variable names you supply while fitting the model. If you do not supply the variable names, then `VariableInfo` contains the default names.

Data Types: cell

## Methods

<code>anova</code>	Analysis of variance for linear mixed-effects model
<code>coefCI</code>	Confidence intervals for coefficients of linear mixed-effects model
<code>coefTest</code>	Hypothesis test on fixed and random effects of linear mixed-effects model
<code>compare</code>	Compare linear mixed-effects models
<code>covarianceParameters</code>	Extract covariance parameters of linear mixed-effects model
<code>designMatrix</code>	Fixed- and random-effects design matrices
<code>disp</code>	Display linear mixed-effects model
<code>fit</code>	Fit linear mixed-effects model using tables
<code>fitmatrix</code>	Fit linear mixed-effects model using design matrices
<code>fitted</code>	Fitted responses from a linear mixed-effects model
<code>fixedEffects</code>	Estimates of fixed effects and related statistics
<code>plotResiduals</code>	Plot residuals of linear mixed-effects model
<code>predict</code>	Predict response of linear mixed-effects model
<code>random</code>	Generate random responses from fitted linear mixed-effects model
<code>randomEffects</code>	Estimates of random effects and related statistics
<code>residuals</code>	Residuals of fitted linear mixed-effects model
<code>response</code>	Response vector of the linear mixed-effects model

## Copy Semantics

Value. To learn how value classes affect copy operations, see [Copying Objects \(MATLAB\)](#).

## Examples

### Random Intercept Model with Categorical Predictor

Load the sample data.



```
load flu
```

The `flu` dataset array has a `Date` variable, and 10 variables containing estimated influenza rates (in 9 different regions, estimated from Google® searches, plus a nationwide estimate from the Center for Disease Control and Prevention, CDC).

To fit a linear-mixed effects model, your data must be in a properly formatted dataset array. To fit a linear mixed-effects model with the influenza rates as the responses and region as the predictor variable, combine the nine columns corresponding to the regions into an array. The new dataset array, `flu2`, must have the response variable, `FluRate`, the nominal variable, `Region`, that shows which region each estimate is from, and the grouping variable `Date`.

```
flu2 = stack(flu,2:10,'NewDataVarName','FluRate',...
    'IndVarName','Region');
flu2.Date = nominal(flu2.Date);
```

Fit a linear mixed-effects model with fixed effects for region and a random intercept that varies by `Date`.

Because region is a nominal variable, `fitlme` takes the first region, NE, as the reference and creates eight dummy variables representing the other eight regions. For example,  $I[\text{MidAtl}]$  is the dummy variable representing the region `MidAtl`. For details, see “Dummy Indicator Variables” on page 2-62.

The corresponding model is

$$y_{im} = \beta_0 + \beta_1 I[\text{MidAtl}]_i + \beta_2 I[\text{ENCentral}]_i + \beta_3 I[\text{WNCentral}]_i + \beta_4 I[\text{SATl}]_i \\ + \beta_5 I[\text{ESCentral}]_i + \beta_6 I[\text{WSCentral}]_i + \beta_7 I[\text{Mtn}]_i + \beta_8 I[\text{Pac}]_i + b_{0m} + \varepsilon_{im}, \quad m = 1, 2, \dots, 52,$$

where  $y_{im}$  is the observation  $i$  for level  $m$  of grouping variable `Date`,  $\beta_j$ ,  $j = 0, 1, \dots, 8$ , are the fixed-effects coefficients,  $b_{0m}$  is the random effect for level  $m$  of the grouping variable `Date`, and  $\varepsilon_{im}$  is the observation error for observation  $i$ . The random effect has the prior distribution,  $b_{0m} \sim N(0, \sigma_b^2)$  and the error term has the distribution,  $\varepsilon_{im} \sim N(0, \sigma^2)$ .

```
lme = fitlme(flu2,'FluRate ~ 1 + Region + (1|Date)')
```

```
lme =
```

```
Linear mixed-effects model fit by ML
```

Model information:

Number of observations	468
Fixed effects coefficients	9
Random effects coefficients	52
Covariance parameters	2

Formula:

FluRate ~ 1 + Region + (1 | Date)

Model fit statistics:

AIC	BIC	LogLikelihood	Deviance
318.71	364.35	-148.36	296.71

Fixed effects coefficients (95% CIs):

Name	Estimate	SE	tStat	DF
'(Intercept)'	1.2233	0.096678	12.654	459
'Region_MidAtl'	0.010192	0.052221	0.19518	459
'Region_ENCentral'	0.051923	0.052221	0.9943	459
'Region_WNCentral'	0.23687	0.052221	4.5359	459
'Region_SAtl'	0.075481	0.052221	1.4454	459
'Region_ESCentral'	0.33917	0.052221	6.495	459
'Region_WSCentral'	0.069	0.052221	1.3213	459
'Region_Mtn'	0.046673	0.052221	0.89377	459
'Region_Pac'	-0.16013	0.052221	-3.0665	459

pValue	Lower	Upper
1.085e-31	1.0334	1.4133
0.84534	-0.092429	0.11281
0.3206	-0.050698	0.15454
7.3324e-06	0.13424	0.33949
0.14902	-0.02714	0.1781
2.1623e-10	0.23655	0.44179
0.18705	-0.033621	0.17162
0.37191	-0.055948	0.14929
0.0022936	-0.26276	-0.057514

Random effects covariance parameters (95% CIs):

Group: Date (52 Levels)

Name1	Name2	Type	Estimate	Lower
'(Intercept)'	'(Intercept)'	'std'	0.6443	0.5297

```
Upper
0.78368
```

```
Group: Error
```

Name	Estimate	Lower	Upper
'Res Std'	0.26627	0.24878	0.285

The  $P$ -values  $7.3324e-06$  and  $2.1623e-10$  respectively show that the fixed effects of the flu rates in regions `WNCentral` and `ESCentral` are significantly different relative to the flu rates in region `NE`.

The confidence limits for the standard deviation of the random-effects term,  $\sigma_b$ , do not include 0 ( $0.5297$ ,  $0.78368$ ), which indicates that the random-effects term is significant. You can also test the significance of the random-effects terms using the `compare` method.

The estimated value of an observation is the sum of the fixed effects and the random-effect value at the grouping variable level corresponding to that observation. For example, the estimated best linear unbiased predictor (BLUP) of the flu rate for region `WNCentral` in week `10/9/2005` is

$$\begin{aligned}\hat{y}_{\text{WNCentral},10/9/2005} &= \hat{\beta}_0 + \hat{\beta}_3 I[\text{WNCentral}] + \hat{b}_{10/9/2005} \\ &= 1.2233 + 0.23687 - 0.1718 \\ &= 1.28837.\end{aligned}$$

This is the fitted conditional response, since it includes contribution to the estimate from both the fixed and random effects. You can compute this value as follows.

```
beta = fixedEffects(lme);
[~,~,STATS] = randomEffects(lme); % Compute the random-effects statistics (STATS)
STATS.Level = nominal(STATS.Level);
y_hat = beta(1) + beta(4) + STATS.Estimate(STATS.Level=='10/9/2005')

y_hat =

    1.2884
```

You can simply display the fitted value using the `fitted` method.

```
F = fitted(lme);
F(flu2.Date == '10/9/2005' & flu2.Region == 'WNCentral')
```

```
ans =  
  
1.2884
```

Compute the fitted marginal response for region `WNCentral` in week `10/9/2005`.

```
F = fitted(lme, 'Conditional', false);  
F(flu2.Date == '10/9/2005' & flu2.Region == 'WNCentral')
```

```
ans =  
  
1.4602
```

### Linear Mixed-Effects Model with a Random Slope

Load the sample data.

```
load carbig
```

Fit a linear mixed-effects model for miles per gallon (MPG), with fixed effects for acceleration, horsepower and the cylinders, and uncorrelated random-effect for intercept and acceleration grouped by the model year. This model corresponds to  $MPG_{im} = \beta_0 + \beta_1 Acc_i + \beta_2 HP + b_{0m} + b_{1m} Acc_{im} + \varepsilon_{im}$ ,  $m = 1, 2, 3$ ,

with the random-effects terms having the following prior distributions:

$$b_m = \begin{pmatrix} b_{0m} \\ b_{1m} \end{pmatrix} \sim N \left( 0, \begin{pmatrix} \sigma_0^2 & \sigma_{0,1} \\ \sigma_{0,1} & \sigma_1^2 \end{pmatrix} \right),$$

where  $m$  represents the model year.

First, prepare the design matrices for fitting the linear mixed-effects model.

```
X = [ones(406,1) Acceleration Horsepower];  
Z = [ones(406,1) Acceleration];  
Model_Year = nominal(Model_Year);  
G = Model_Year;
```

Now, fit the model using `fitlmematrix` with the defined design matrices and grouping variables. Use the `'fminunc'` optimization algorithm.

```
lme = fitlmematrix(X,MPG,Z,G,'FixedEffectPredictors',....
{'Intercept','Acceleration','Horsepower'},'RandomEffectPredictors',...
{'Intercept','Acceleration'}},'RandomEffectGroups',{'Model_Year'},...
'FitMethod','REML')
```

```
lme =
```

```
Linear mixed-effects model fit by REML
```

```
Model information:
```

Number of observations	392
Fixed effects coefficients	3
Random effects coefficients	26
Covariance parameters	4

```
Formula:
```

```
Linear Mixed Formula with 4 predictors.
```

```
Model fit statistics:
```

AIC	BIC	LogLikelihood	Deviance
2202.9	2230.7	-1094.5	2188.9

```
Fixed effects coefficients (95% CIs):
```

Name	Estimate	SE	tStat	DF	pValue
'Intercept'	50.064	2.3176	21.602	389	1.4185e-68
'Acceleration'	-0.57897	0.13843	-4.1825	389	3.5654e-05
'Horsepower'	-0.16958	0.0073242	-23.153	389	3.5289e-75

Lower	Upper
45.507	54.62
-0.85112	-0.30681
-0.18398	-0.15518

```
Random effects covariance parameters (95% CIs):
```

```
Group: Model_Year (13 Levels)
```

Name1	Name2	Type	Estimate
'Intercept'	'Intercept'	'std'	3.72
'Acceleration'	'Intercept'	'corr'	-0.8769

```
'Acceleration'      'Acceleration'      'std'      0.3593
```

```
Lower      Upper
  1.5215    9.0954
-0.98274   -0.33846
  0.19418    0.66483
```

```
Group: Error
```

```
Name      Estimate      Lower      Upper
'Res Std'  3.6913      3.4331    3.9688
```

The fixed effects coefficients display includes the estimate, standard errors (SE), and the 95% confidence interval limits (Lower and Upper). The *P*-values for (pValue) indicate that all three fixed-effects coefficients are significant.

The confidence intervals for the standard deviations and the correlation between the random effects for intercept and acceleration do not include zeros, hence they seem significant. Use the `compare` method to test for the random effects.

Display the covariance matrix of the estimated fixed-effects coefficients.

```
lme.CoefficientCovariance
```

```
ans =
```

```
  5.3711   -0.2809   -0.0126
 -0.2809    0.0192    0.0005
 -0.0126    0.0005    0.0001
```

The diagonal elements show the variances of the fixed-effects coefficient estimates. For example, the variance of the estimate of the intercept is 5.3711. Note that the standard errors of the estimates are the square roots of the variances. For example, the standard error of the intercept is 2.3176, which is `sqrt(5.3711)`.

The off-diagonal elements show the correlation between the fixed-effects coefficient estimates. For example, the correlation between the intercept and acceleration is  $-0.2809$  and the correlation between acceleration and horsepower is 0.0005.

Display the coefficient of determination for the model.

```

lme.Rsquared

ans =

  struct with fields:

    Ordinary: 0.7826
    Adjusted: 0.7815

```

The adjusted value is the R-squared value adjusted for the number of predictors in the model.

## Definitions

### Formula

In general, a formula for model specification is a character vector of the form 'y ~ terms'. For the linear mixed-effects models, this formula is in the form 'y ~ fixed + (random1|grouping1) + ... + (randomR|groupingR)', where *fixed* and *random* contain the fixed-effects and the random-effects terms.

Suppose a table `tbl` contains the following:

- A response variable,  $y$
- Predictor variables,  $X_j$ , which can be continuous or grouping variables
- Grouping variables,  $g_1, g_2, \dots, g_R$ ,

where the grouping variables in  $X_j$  and  $g_r$  can be categorical, logical, character arrays, or cell arrays of character vectors.

Then, in a formula of the form, 'y ~ fixed + (random<sub>1</sub>|g<sub>1</sub>) + ... + (random<sub>R</sub>|g<sub>R</sub>)', the term *fixed* corresponds to a specification of the fixed-effects design matrix  $X$ , *random<sub>1</sub>* is a specification of the random-effects design matrix  $Z_1$  corresponding to grouping variable  $g_1$ , and similarly *random<sub>R</sub>* is a specification of the random-effects design matrix  $Z_R$  corresponding to grouping variable  $g_R$ . You can express the *fixed* and *random* terms using Wilkinson notation.

Wilkinson notation describes the factors present in models. The notation relates to factors present in models, not to the multipliers (coefficients) of those factors.

Wilkinson Notation	Factors in Standard Notation
1	Constant (intercept) term
$X^k$ , where $k$ is a positive integer	$X, X^2, \dots, X^k$
$X1 + X2$	$X1, X2$
$X1 * X2$	$X1, X2, X1.*X2$ (elementwise multiplication of $X1$ and $X2$ )
$X1 : X2$	$X1.*X2$ only
$- X2$	Do not include $X2$
$X1 * X2 + X3$	$X1, X2, X3, X1 * X2$
$X1 + X2 + X3 + X1 : X2$	$X1, X2, X3, X1 * X2$
$X1 * X2 * X3 - X1 : X2 : X3$	$X1, X2, X3, X1 * X2, X1 * X3, X2 * X3$
$X1 * (X2 + X3)$	$X1, X2, X3, X1 * X2, X1 * X3$

Statistics and Machine Learning Toolbox notation always includes a constant term unless you explicitly remove the term using  $-1$ . Here are some examples for linear mixed-effects model specification.

#### Examples:

Formula	Description
' $y \sim X1 + X2$ '	Fixed effects for the intercept, $X1$ and $X2$ . This is equivalent to ' $y \sim 1 + X1 + X2$ '.
' $y \sim -1 + X1 + X2$ '	No intercept and fixed effects for $X1$ and $X2$ . The implicit intercept term is suppressed by including $-1$ .
' $y \sim 1 + (1   g1)$ '	Fixed effects for the intercept plus random effect for the intercept for each level of the grouping variable $g1$ .
' $y \sim X1 + (1   g1)$ '	Random intercept model with a fixed slope.



Formula	Description
'y ~ X1 + (X1   g1) '	Random intercept and slope, with possible correlation between them. This is equivalent to 'y ~ 1 + X1 + (1 + X1   g1) '.
'y ~ X1 + (1   g1) + (-1 + X1   g1) '	Independent random effects terms for intercept and slope.
'y ~ 1 + (1   g1) + (1   g2) + (1   g1:g2) '	Random intercept model with independent main effects for g1 and g2, plus an independent interaction effect.

## See Also

[fitlme](#) | [fitlmematrix](#) | [plotPartialDependence](#)

## linhyptest

Linear hypothesis test

### Syntax

```
p = linhyptest(beta, COVB, c, H, dfe)
[p, t, r] = linhyptest(...)
```

### Description

`p = linhyptest(beta, COVB, c, H, dfe)` returns the  $p$  value  $p$  of a hypothesis test on a vector of parameters. `beta` is a vector of  $k$  parameter estimates. `COVB` is the  $k$ -by- $k$  estimated covariance matrix of the parameter estimates. `c` and `H` specify the null hypothesis in the form  $H*b = c$ , where  $b$  is the vector of unknown parameters estimated by `beta`. `dfe` is the degrees of freedom for the `COVB` estimate, or `Inf` if `COVB` is known rather than estimated.

`beta` is required. The remaining arguments have default values:

- `COVB = eye(k)`
- `c = zeros(k,1)`
- `H = eye(K)`
- `dfe = Inf`

If `H` is omitted, `c` must have  $k$  elements and it specifies the null hypothesis values for the entire parameter vector.

---

**Note** The following functions return outputs suitable for use as the `COVB` input argument to `linhyptest`: `nlinfit`, `coxphfit`, `glmfit`, `mnrfit`, `regstats`, `robustfit`. `nlinfit` returns `COVB` directly; the other functions return `COVB` in `stats.covb`.

---

`[p, t, r] = linhyptest(...)` also returns the test statistic  $t$  and the rank  $r$  of the hypothesis matrix  $H$ . If `dfe` is `Inf` or is not given,  $t*r$  is a chi-square statistic with  $r$

degrees of freedom . If `dfe` is specified as a finite value, `t` is an  $F$  statistic with `r` and `dfe` degrees of freedom.

`linhyptest` performs a test based on an asymptotic normal distribution for the parameter estimates. It can be used after any estimation procedure for which the parameter covariances are available, such as `regstats` or `glmfit`. For linear regression, the  $p$ -values are exact. For other procedures, the  $p$ -values are approximate, and may be less accurate than other procedures such as those based on a likelihood ratio.

## Examples

Fit a multiple linear model to the data in `hald.mat`:

```
load hald
stats = regstats(heat,ingredients,'linear');
beta = stats.beta
beta =
    62.4054
     1.5511
     0.5102
     0.1019
    -0.1441
```

Perform an  $F$ -test that the last two coefficients are both 0:

```
SIGMA = stats.covb;
dfe = stats.fstat.dfe;
H = [0 0 0 1 0;0 0 0 0 1];
c = [0;0];
[p,F] = linhyptest(beta,SIGMA,c,H,dfe)
p =
    0.4668
F =
    0.8391
```

## See Also

`coxphfit` | `glmfit` | `mnrfit` | `nlinfit` | `regstats` | `robustfit`

Introduced in R2007a

## linkage

Agglomerative hierarchical cluster tree

### Syntax

```
Z = linkage(X)
Z = linkage(X,method)
Z = linkage(X,method,metric)
Z = linkage(X,method,pdist_inputs)
Z = linkage(X,method,metric,'savememory',value)
Z = linkage(Y)
Z = linkage(Y,method)
```

### Description

`Z = linkage(X)` returns a matrix `Z` that encodes a tree of hierarchical clusters of the rows of the real matrix `X`.

`Z = linkage(X,method)` creates the tree using the specified *method*, where *method* describes how to measure the distance between clusters.

`Z = linkage(X,method,metric)` performs clustering using the distance measure *metric* to compute distances between the rows of `X`.

`Z = linkage(X,method,pdist_inputs)` passes parameters to the `pdist` function, which is the function that computes the distance between rows of `X`.

`Z = linkage(X,method,metric,'savememory',value)` uses a memory-saving algorithm when *value* is `'true'`, and uses the standard algorithm when *value* is `'false'`.

`Z = linkage(Y)` uses a vector representation `Y` of a distance matrix. `Y` can be a distance matrix as computed by `pdist`, or a more general dissimilarity matrix conforming to the output format of `pdist`.

`Z = linkage(Y,method)` creates the tree using the specified *method*, where *method* describes how to measure the distance between clusters.

## Input Arguments

### **x**

Matrix with two or more rows. The rows represent observations, the columns represent categories or dimensions.

### **method**

Algorithm for computing distance between clusters.

Method	Description
'average'	Unweighted average distance (UPGMA)
'centroid'	Centroid distance (UPGMC), appropriate for Euclidean distances only
'complete'	Furthest distance
'median'	Weighted center of mass distance (WPGMC), appropriate for Euclidean distances only
'single'	Shortest distance
'ward'	Inner squared distance (minimum variance algorithm), appropriate for Euclidean distances only
'weighted'	Weighted average distance (WPGMA)

**Default:** 'single'

### **metric**

Any distance metric that the `pdist` function accepts.

Use `pdist_inputs` instead of `metric` to specify an additional input argument, `DistParameter` of `pdist`, for 'seuclidean', 'minkowski', or 'mahalanobis'.

Value	Description
'euclidean'	Euclidean distance (default).

Value	Description
'squaredeuclidean'	Squared Euclidean distance. (This option is provided for efficiency only. It does not satisfy the triangle inequality.)
'seuclidean'	Standardized Euclidean distance. Each coordinate difference between observations is scaled by dividing by the corresponding element of the standard deviation, $S = \text{nanstd}(X)$ . Use <code>DistParameter</code> to specify another value for $S$ .
'mahalanobis'	Mahalanobis distance using the sample covariance of $X$ , $C = \text{nancov}(X)$ . Use <code>DistParameter</code> to specify another value for $C$ , where the matrix $C$ is symmetric and positive definite.
'cityblock'	City block distance.
'minkowski'	Minkowski distance. The default exponent is 2. Use <code>DistParameter</code> to specify a different exponent $P$ , where $P$ is a positive scalar value of the exponent.
'chebychev'	Chebychev distance (maximum coordinate difference).
'cosine'	One minus the cosine of the included angle between points (treated as vectors).
'correlation'	One minus the sample correlation between points (treated as sequences of values).
'hamming'	Hamming distance, which is the percentage of coordinates that differ.
'jaccard'	One minus the Jaccard coefficient, which is the percentage of nonzero coordinates that differ.
'spearman'	One minus the sample Spearman's rank correlation between observations (treated as sequences of values).

Value	Description
@ <i>distfun</i>	<p>Custom distance function handle. A distance function has the form</p> <pre>function D2 = DISTFUN(ZI,ZJ) % calculation of distance ...</pre> <p>where</p> <ul style="list-style-type: none"> <li>• ZI is a 1-by-n vector containing a single observation.</li> <li>• ZJ is an m2-by-n matrix containing multiple observations. <i>distfun</i> must accept a matrix XJ with an arbitrary number of observations.</li> <li>• D2 is an m2-by-1 vector of distances, and D2(k) is the distance between observations ZI and ZJ(k, :).</li> </ul> <p>If your data is not sparse, you can generally compute distance more quickly by using a built-in distance instead of a function handle.</p>

For more information on these distance metrics, see “Distance Metrics” on page 32-4665.

**Default:** 'euclidean'

### **pdist\_inputs**

Distance metric and distance metric option, specified as a cell array of the comma-separated pair consisting of the two input arguments *Distance* and *DistParameter* of the function *pdist*. For example, to use 'minkowski' with an exponent of 5, set *pdist\_inputs* to {'minkowski', 5}. This argument is valid only when you use 'seuclidean', 'minkowski', or 'mahalanobis'.

### **savememory**

Either 'on' or 'off'. When applicable, the 'on' setting causes *linkage* to construct clusters without computing the distance matrix. *savememory* is applicable when:

- *linkage* is 'centroid', 'median', or 'ward'
- *distance* is 'euclidean' (default)

When `savememory` is 'on', linkage run time is proportional to the number of dimensions (number of columns of `X`). When `savememory` is 'off', linkage memory requirement is proportional to  $N^2$ , where  $N$  is the number of observations. So choosing the best (least-time) setting for `savememory` depends on the problem dimensions, number of observations, and available memory. The default `savememory` setting is a rough approximation of an optimal setting.

**Default:** 'on' when `X` has 20 columns or fewer, or the computer does not have enough memory to store the distance matrix; otherwise 'off'

### **Y**

A vector of distances with the same format as the output of the `pdist` function:

- A row vector of length  $m(m-1)/2$ , corresponding to pairs of observations in a matrix `X` with  $m$  rows
- Distances arranged in the order (2,1), (3,1), ..., (m,1), (3,2), ..., (m,2), ..., (m,m-1)

`Y` can be a more general dissimilarity matrix conforming to the output format of `pdist`.

## Output Arguments

### **Z**

`Z` is a  $(m-1)$ -by-3 matrix, where  $m$  is the number of observations in the original data. Columns 1 and 2 of `Z` contain cluster indices linked in pairs to form a binary tree. The leaf nodes are numbered from 1 to  $m$ . Leaf nodes are the singleton clusters from which all higher clusters are built. Each newly-formed cluster, corresponding to row `Z(I, :)`, is assigned the index  $m+I$ . `Z(I, 1:2)` contains the indices of the two component clusters that form cluster  $m+I$ . There are  $m-1$  higher clusters which correspond to the interior nodes of the clustering tree. `Z(I, 3)` contains the linkage distances between the two clusters merged in row `Z(I, :)`.

For example, suppose there are 30 initial nodes and at step 12 cluster 5 and cluster 7 are combined. Suppose their distance at that time is 1.5. Then `Z(12, :)` will be `[5, 7, 1.5]`. The newly formed cluster will have index  $12 + 30 = 42$ . If cluster 42 appears in a later row, it means the cluster created at step 12 is being combined into some larger cluster.



## Examples

### Compare Cluster Assignments to Clusters

Load the sample data.

```
load fisheriris
```

Compute four clusters of the Fisher iris data using Ward linkage and ignoring species information.

```
Z = linkage(meas, 'ward', 'euclidean');
c = cluster(Z, 'maxclust', 4);
```

See how the cluster assignments correspond to the three species.

```
crosstab(c, species)
```

```
ans =
```

0	25	1
0	24	14
0	1	35
50	0	0

Display the first five rows of Z.

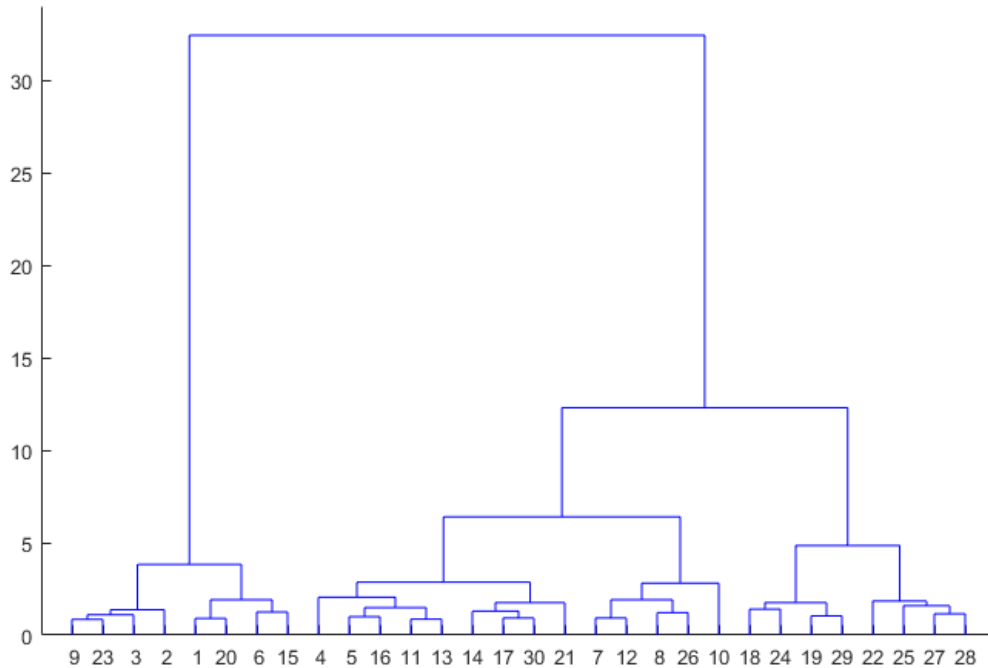
```
firstfive = Z(1:5, :)
```

```
firstfive =
```

102.0000	143.0000	0
8.0000	40.0000	0.1000
1.0000	18.0000	0.1000
10.0000	35.0000	0.1000
129.0000	133.0000	0.1000

Create a dendrogram plot of Z .

```
dendrogram(Z)
```



### Cluster Data and Plot the Result

Randomly generate the sample data with 20000 observations.

```
rng default; % For reproducibility
X = rand(20000,3);
```

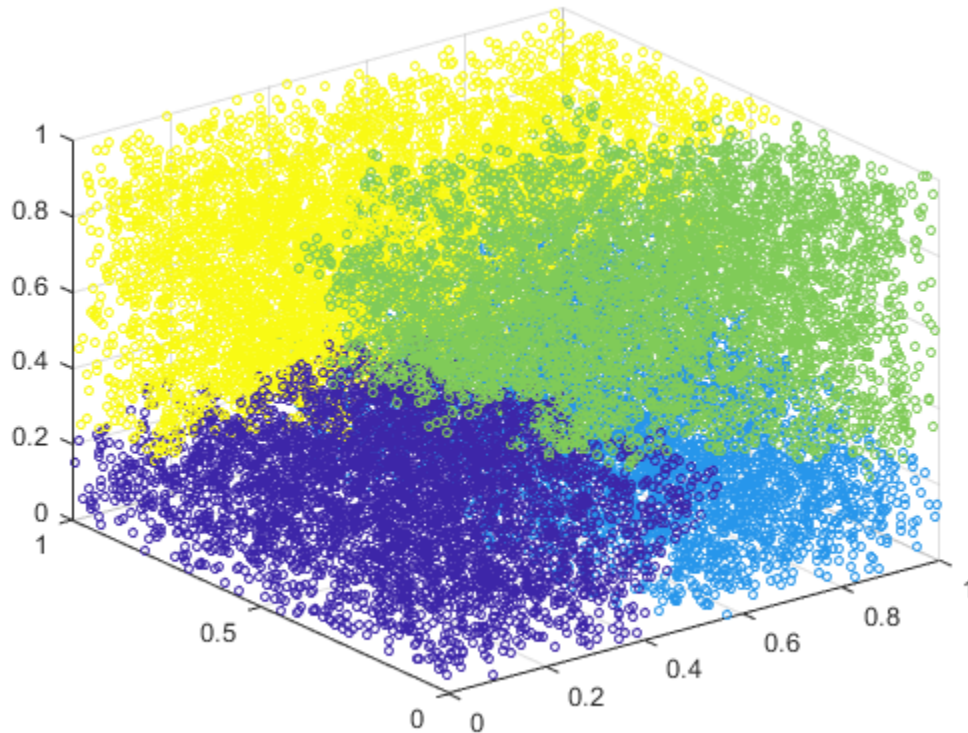
Create a hierarchical cluster tree using Ward's linkage.

```
Z = linkage(X,'ward','euclidean','savememory','on');
```

If you set `savememory` to `'off'`, you can get an out-of-memory error if your machine doesn't have enough memory to hold the distance matrix.

Cluster data into four groups and plot the result.

```
c = cluster(Z, 'maxclust', 4);  
scatter3(X(:,1), X(:,2), X(:,3), 10, c)
```



## Definitions

### Linkages

The following notation is used to describe the linkages used by the various methods:

- Cluster  $r$  is formed from clusters  $p$  and  $q$ .
- $n_r$  is the number of objects in cluster  $r$ .

- $x_{ri}$  is the  $i$ th object in cluster  $r$ .
- *Single linkage*, also called *nearest neighbor*, uses the smallest distance between objects in the two clusters:  

$$d(r, s) = \min(\text{dist}(x_{ri}, x_{sj}), i \in (1, \dots, n_r), j \in (1, \dots, n_s))$$
- *Complete linkage*, also called *furthest neighbor*, uses the largest distance between objects in the two clusters:  

$$d(r, s) = \max(\text{dist}(x_{ri}, x_{sj}), i \in (1, \dots, n_r), j \in (1, \dots, n_s))$$
- *Average linkage* uses the average distance between all pairs of objects in any two clusters:

$$d(r, s) = \frac{1}{n_r n_s} \sum_{i=1}^{n_r} \sum_{j=1}^{n_s} \text{dist}(x_{ri}, x_{sj})$$

- *Centroid linkage* uses the Euclidean distance between the centroids of the two clusters:

$$d(r, s) = \|\bar{x}_r - \bar{x}_s\|_2$$

where

$$\bar{x}_r = \frac{1}{n_r} \sum_{i=1}^{n_r} x_{ri}$$

- *Median linkage* uses the Euclidean distance between weighted centroids of the two clusters,

$$d(r, s) = \|\tilde{x}_r - \tilde{x}_s\|_2$$

where  $\tilde{x}_r$  and  $\tilde{x}_s$  are weighted centroids for the clusters  $r$  and  $s$ . If cluster  $r$  was created by combining clusters  $p$  and  $q$ ,  $\tilde{x}_r$  is defined recursively as

$$\tilde{x}_r = \frac{1}{2}(\tilde{x}_p + \tilde{x}_q)$$

- *Ward's linkage* uses the incremental sum of squares; that is, the increase in the total within-cluster sum of squares as a result of joining two clusters. The within-cluster sum of squares is defined as the sum of the squares of the distances between all objects in the cluster and the centroid of the cluster. The sum of squares measure is

equivalent to the following distance measure  $d(r,s)$ , which is the formula linkage uses:

$$d(r,s) = \sqrt{\frac{2n_r n_s}{n_r + n_s}} \|\bar{x}_r - \bar{x}_s\|_2,$$

where

- $\|\cdot\|_2$  is Euclidean distance
- $\bar{x}_r$  and  $\bar{x}_s$  are the centroids of clusters  $r$  and  $s$
- $n_r$  and  $n_s$  are the number of elements in clusters  $r$  and  $s$

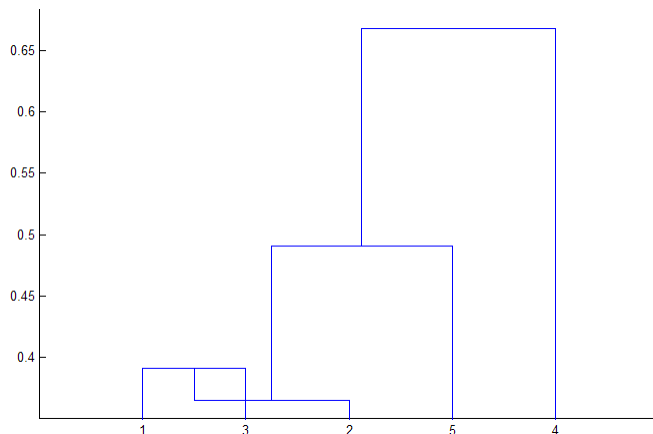
In some references the Ward linkage does not use the factor of 2 multiplying  $n_r n_s$ . The linkage function uses this factor so the distance between two singleton clusters is the same as the Euclidean distance.

- Weighted average linkage uses a recursive definition for the distance between two clusters. If cluster  $r$  was created by combining clusters  $p$  and  $q$ , the distance between  $r$  and another cluster  $s$  is defined as the average of the distance between  $p$  and  $s$  and the distance between  $q$  and  $s$ :

$$d(r,s) = \frac{(d(p,s) + d(q,s))}{2}$$

## Tips

- Computing `linkage(Y)` can be slow when  $Y$  is a vector representation of the distance matrix. For the 'centroid', 'median', and 'ward' methods, linkage checks whether  $Y$  is a Euclidean distance. Avoid this time-consuming check by passing in  $X$  instead of  $Y$ .
- The centroid and median methods can produce a cluster tree that is not monotonic. This occurs when the distance from the union of two clusters,  $r$  and  $s$ , to a third cluster is less than the distance between  $r$  and  $s$ . In this case, in a dendrogram drawn with the default orientation, the path from a leaf to the root node takes some downward steps. To avoid this, use another method. The following image shows a nonmonotonic cluster tree.



In this case, cluster 1 and cluster 3 are joined into a new cluster, while the distance between this new cluster and cluster 2 is less than the distance between cluster 1 and cluster 3. This leads to a nonmonotonic tree.

- You can provide the output `Z` to other functions including `dendrogram` to display the tree, `cluster` to assign points to clusters, `inconsistent` to compute inconsistent measures, and `cophenet` to compute the cophenetic correlation coefficient.

## See Also

`cluster` | `clusterdata` | `cophenet` | `dendrogram` | `inconsistent` | `kmeans` | `pdist` | `silhouette` | `squareform`

**Introduced before R2006a**

# loadCompactModel

Reconstruct model object from saved model for code generation

To generate C code, you must have a MATLAB Coder license.

## Syntax

```
Mdl = loadCompactModel(filename)
```

## Description

`Mdl = loadCompactModel(filename)` reconstructs a compact classification model, compact regression model, or nearest neighbor searcher (`Mdl`) from the saved model stored in the MATLAB formatted binary file (MAT-file) `filename`. You must create `filename` using `saveCompactModel`.

## Examples

### Load Compact SVM Model from Structure Array

Load the ionosphere data set.

```
load ionosphere
```

Train an SVM classification model using the entire data set. Specify to standardize the data.

```
Mdl = fitcsvm(X,Y,'Standardize',true)
```

```
Mdl =
```

```
ClassificationSVM
    ResponseName: 'Y'
```

```
CategoricalPredictors: []
    ClassNames: {'b' 'g'}
    ScoreTransform: 'none'
    NumObservations: 351
        Alpha: [90x1 double]
        Bias: -0.1343
    KernelParameters: [1x1 struct]
        Mu: [1x34 double]
        Sigma: [1x34 double]
    BoxConstraints: [351x1 double]
    ConvergenceInfo: [1x1 struct]
    IsSupportVector: [351x1 logical]
    Solver: 'SMO'
```

Mdl is a ClassificationSVM model.

Save the SVM classification model to the file 'SVMIonosphere.mat'.

```
saveCompactModel(Mdl, 'SVMIonosphere');
```

'SVMIonosphere.mat' appears in your present working directory. saveCompactModel reduces the memory footprint of the model by removing properties that are not needed for prediction, for example, the training data. Then, saveCompactModel saves a structure array that characterizes Mdl in 'SVMIonosphere.mat'.

Load the structure array in 'SVMIonosphere.mat' to the Workspace.

```
CompactMdl = loadCompactModel('SVMIonosphere')
```

```
CompactMdl =
classreg.learning.classif.CompactClassificationSVM
    ResponseName: 'Y'
    CategoricalPredictors: []
        ClassNames: {'b' 'g'}
        ScoreTransform: 'none'
        Alpha: [90x1 double]
        Bias: -0.1343
    KernelParameters: [1x1 struct]
        Mu: [1x34 double]
        Sigma: [1x34 double]
```



```
SupportVectors: [90x34 double]
SupportVectorLabels: [90x1 double]
```

CompactMdl is a CompactClassificationSVM model equipped to make predictions.

### Generate Code From Function That Classifies New Data

Declare a function that classifies new observations using a trained classification model. This example requires a MATLAB® Coder™ license.

Load Fisher's iris data set.

```
load fisheriris
```

Train an ECOC model composed of SVM binary learners. Using an SVM template, standardize the data, hold out 30% of the data from training, and specify the Gaussian kernel.

```
t = templateSVM('Standardize',true,'KernelFunction','gaussian');
CVMdl = fitcecoc(meas,species,'Learners',t,'Holdout',0.30);
```

CVMdl is a ClassificationPartitionedECOC model, which stores, among other things:

- A compact ECOC model, which `fitcecoc` trained using the training set.
- The data partition object, which specifies how `fitcecoc` randomly split the data into training and holdout sets.

Extract the compact ECOC classification model and the holdout set indices using dot notation. Identify the holdout observations.

```
CompactMdl = CVMdl.Trained{1};
cvp = CVMdl.Partition;
idxTest = test(cvp);
measHO = meas(idxTest,:);
```

CompactMdl is a CompactClassificationECOC model equipped to classify new observations.

Save the compact ECOC classification model to the file `ECOCIris.mat`.

```
saveCompactModel(CompactMdl, 'ECOCIris');
```

Declare a function in your current working folder called `classifyIris.m` that:

- Accepts iris-flower measurements commensurate with `meas`, and returns predicted labels
- Loads the compact ECOC classification model
- Passes the loaded model and iris-flower measurements to `predict`

```
function label = classifyIris(X) %#codegen
%CLASSIFYIRISES Classify iris species using ECOC Model
% CLASSIFYIRISES classifies the iris-flower measurements in X
% using the compact ECOC model in the file ECOCIris.mat, and then
% returns class labels in label.
CompactMdl = loadCompactModel('ECOCIris');
label = predict(CompactMdl, X);

end
```

The `%codegen` compilation directive indicates that the MATLAB® code is intended for code generation.

Generate a MEX function from `classifyIris.m`. Because C uses static typing, `codegen` must determine the properties of all variables in MATLAB® files at compile time. To ensure that the MEX function can use the same input, specify the holdout observations as arguments to the function using the `'-args'` option.

```
codegen classifyIris.m -args {measHO}
```

The MEX file `classifyIris_mex.mexw64` generates to your present working directory. The file extension depends on your platform.

Compare labels predicted using `predict` at the command line, `classifyIris.m`, and `classifyIris_mex`.

```
label1 = predict(CompactMdl, measHO);
label2 = classifyIris(measHO);
label3 = classifyIris_mex(measHO);

comp12 = cellfun(@strcmp, label1, label2);
comp23 = cellfun(@strcmp, label2, label3);
```

```
agree12 = sum(comp12) == numel(label1)
agree23 = sum(comp23) == numel(label1)
```

```
agree12 =
```

```
    logical
```

```
    1
```

```
agree23 =
```

```
    logical
```

```
    1
```

The labels predicted all three ways are the same.

- “C Code Generation and Classification Learner” on page 31-8
- “Predict Class Labels Using MATLAB Function Block” on page 31-18
- “C Code Generation for Image Classifier”

## Input Arguments

**filename** — MAT-file name containing structure array representing a model object  
character vector

MAT-file name containing the structure array representing a model object, specified as a character vector. You must create `filename` using `saveCompactModel`.

`loadCompactModel` reconstructs the model stored in `filename` at compile time. For supported models, see the `Mdl` input argument of `saveCompactModel`.

If `filename` has no extension (that is, no period followed by text), then `loadCompactModel` appends `.mat`.

If `filename` does not include a full path, then `loadCompactModel` loads from the present working directory.

Example: 'mdl'

## Output Arguments

**mdl** — Compact classification model, compact regression model, or nearest neighbor searcher

model object

Compact classification model, compact regression model, or nearest neighbor searcher, returned as one of the model objects in this table.

	Model Objects	Notes
Compact classification model object	ClassificationKNN	A ClassificationKNN model object is a full object that does not have a corresponding compact object. <code>saveCompactModel</code> saves a compact version that does not include cross-validation parameters and hyperparameters. <code>loadCompactModel</code> returns the compact version saved using <code>saveCompactModel</code> .
	ClassificationLinear	This model object is a compact object that does not have a corresponding full object.
	CompactClassificationDiscriminant	
	CompactClassificationECOC	
	CompactClassificationEnsemble	

	Model Objects	Notes
	CompactClassificationSVM	If you use saveCompactModel to save an SVM model that is equipped to predict posterior probabilities, and use loadCompactModel to load the model, then loadCompactModel cannot restore the ScoreTransform property into the MATLAB Workspace. However, loadCompactModel can load the model, including the ScoreTransform property, at compile time for code generation, within a function that you declare.
	CompactClassificationTree	
Compact regression model object	CompactGeneralizedLinearModel	
	CompactLinearModel	If you train a linear model by using fitlm and specifying 'RobustOpts' as a structure with an anonymous function handle for the RobustWgtFun field, use saveCompactModel to save the model, and use loadCompactModel to load the model, then loadCompactModel cannot restore the Robust property into the MATLAB Workspace. However, loadCompactModel can load the model at compile time for code generation within a function that you declare.
	CompactRegressionEnsemble	
	CompactRegressionGP	
	CompactRegressionSVM	
	CompactRegressionTree	

	Model Objects	Notes
	<code>RegressionLinear</code>	This model object is a compact object that does not have a corresponding full object.
Nearest neighbor searcher object	<code>ExhaustiveSearcher</code>	This model object is a full object that does not have a corresponding compact object. <code>saveCompactModel</code> saves the model as it is, and <code>loadCompactModel</code> returns the saved model.

`mdl` must be a compile-time constant. You cannot change it after you load it using `loadCompactModel`.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`saveCompactModel`

### Topics

“C Code Generation and Classification Learner” on page 31-8

“Predict Class Labels Using MATLAB Function Block” on page 31-18

“C Code Generation for Image Classifier”

“Code Generation Workflows” on page 31-2

“Code Generation Support, Usage Notes, and Limitations” on page 31-24

**Introduced in R2016b**

# logncdf

Lognormal cumulative distribution function

## Syntax

```
p = logncdf(x,mu,sigma)
[p,plo,pup] = logncdf(x,mu,sigma,pcov,alpha)
[p,plo,pup] = logncdf(____,'upper')
```

## Description

`p = logncdf(x,mu,sigma)` returns values at `x` of the lognormal cdf with distribution parameters `mu` and `sigma`. `mu` and `sigma` are the mean and standard deviation, respectively, of the associated normal distribution. `x`, `mu`, and `sigma` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input for `x`, `mu`, or `sigma` is expanded to a constant array with the same dimensions as the other inputs.

`[p,plo,pup] = logncdf(x,mu,sigma,pcov,alpha)` returns confidence bounds for `p` when the input parameters `mu` and `sigma` are estimates. `pcov` is the covariance matrix of the estimated parameters. `alpha` specifies `100(1 - alpha)%` confidence bounds. The default value of `alpha` is `0.05`. `plo` and `pup` are arrays of the same size as `p` containing the lower and upper confidence bounds.

`[p,plo,pup] = logncdf(____,'upper')` returns the complement of the lognormal cdf at each value in `x`, using an algorithm that more accurately computes the extreme upper tail probabilities. You can use `'upper'` with any of the previous syntaxes.

`logncdf` computes confidence bounds for `p` using a normal approximation to the distribution of the estimate

$$\frac{x - \hat{\mu}}{\hat{\sigma}}$$

and then transforming those bounds to the scale of the output `p`. The computed bounds give approximately the desired confidence level when you estimate `mu`, `sigma`, and `pcov`

from large samples, but in smaller samples other methods of computing the confidence bounds might be more accurate.

The lognormal cdf is

$$p = F(x | \mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} \int_0^x e^{-\frac{(\ln(t)-\mu)^2}{2\sigma^2}} dt$$

## Examples

### Compute the Lognormal Distribution cdf

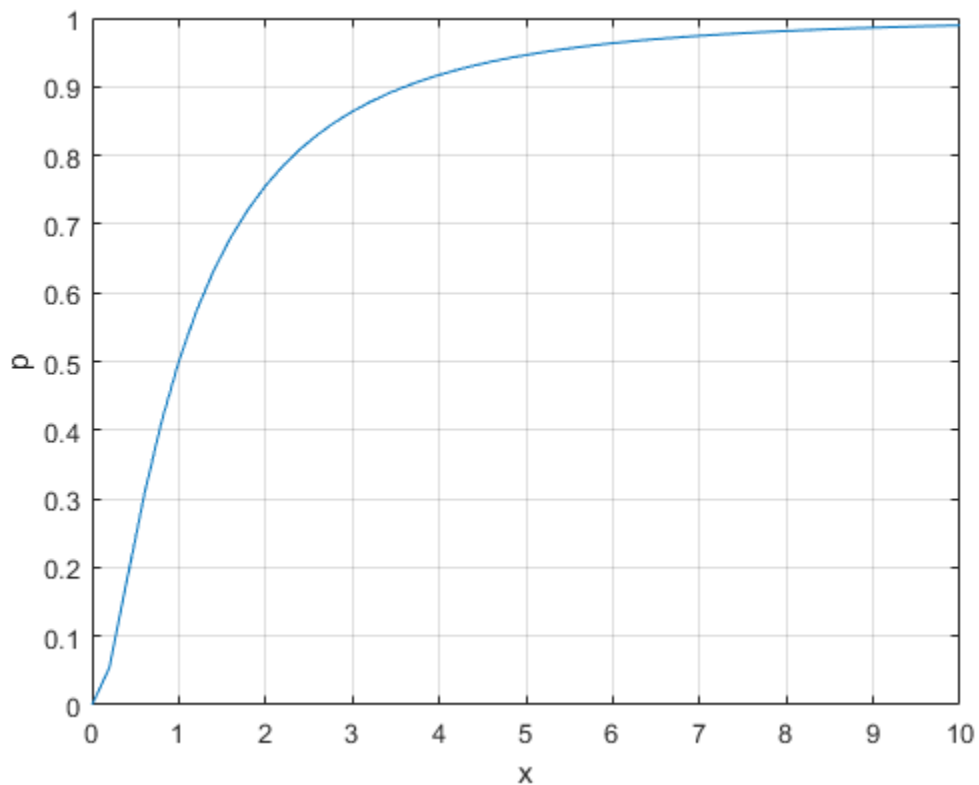
Compute the cdf of a lognormal distribution with `mu = 0` and `sigma = 1`.

```
x = (0:0.2:10);  
y = logncdf(x,0,1);
```

Plot the cdf.

```
plot(x,y);  
grid;  
xlabel('x');  
ylabel('p');
```





## References

- [1] Evans, M., N. Hastings, and B. Peacock. *Statistical Distributions*. 2nd ed., Hoboken, NJ: John Wiley & Sons, Inc., 1993, pp. 102–105.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`cdf` | `lognfit` | `logninv` | `lognlike` | `lognpdf` | `lognrnd` | `lognstat`

### Topics

“Lognormal Distribution” on page B-100

Introduced before R2006a

# lognfit

Lognormal parameter estimates

## Syntax

```
parmhat = lognfit(data)
[parmhat,parmci] = lognfit(data)
[parmhat,parmci] = lognfit(data,alpha)
[...] = lognfit(data,alpha,censoring)
[...] = lognfit(data,alpha,censoring,freq)
[...] = lognfit(data,alpha,censoring,freq,options)
```

## Description

`parmhat = lognfit(data)` returns a vector of maximum likelihood estimates `parmhat(1) = mu` and `parmhat(2) = sigma` of parameters for a lognormal distribution fitting `data`. `mu` and `sigma` are the mean and standard deviation, respectively, of the associated normal distribution.

`[parmhat,parmci] = lognfit(data)` returns 95% confidence intervals for the parameter estimates `mu` and `sigma` in the 2-by-2 matrix `parmci`. The first column of the matrix contains the lower and upper confidence bounds for parameter `mu`, and the second column contains the confidence bounds for parameter `sigma`.

`[parmhat,parmci] = lognfit(data,alpha)` returns  $100(1 - \alpha)$  % confidence intervals for the parameter estimates, where `alpha` is a value in the range (0 1) specifying the width of the confidence intervals. By default, `alpha` is 0.05, which corresponds to 95% confidence intervals.

`[...] = lognfit(data,alpha,censoring)` accepts a Boolean vector `censoring`, of the same size as `data`, which is 1 for observations that are right-censored and 0 for observations that are observed exactly.

`[...] = lognfit(data,alpha,censoring,freq)` accepts a frequency vector, `freq`, of the same size as `data`. Typically, `freq` contains integer frequencies for the

corresponding elements in `data`, but can contain any nonnegative values. Pass in `[]` for `alpha`, `censoring`, or `freq` to use their default values.

`[...] = lognfit(data, alpha, censoring, freq, options)` accepts a structure, `options`, that specifies control parameters for the iterative algorithm the function uses to compute maximum likelihood estimates when there is censoring. The lognormal fit function accepts an `options` structure which can be created using the function `statset`. Enter `statset('lognfit')` to see the names and default values of the parameters that `lognfit` accepts in the `options` structure. See the reference page for `statset` for more information about these options.

---

**Note** With no censoring, `lognfit` computes `sigma` using the square root of the unbiased estimator of the variance. With censoring, `sigma` is the maximum likelihood estimate.

---

## Examples

This example generates 100 independent samples of lognormally distributed data with  $\mu = 0$  and  $\sigma = 3$ . `parmhat` estimates  $\mu$  and  $\sigma$  and `parmci` gives 99% confidence intervals around `parmhat`. Notice that `parmci` contains the true values of  $\mu$  and  $\sigma$ .

```
data = lognrnd(0,3,100,1);
[parmhat,parmci] = lognfit(data,0.01)
parmhat =
    -0.2480    2.8902
parmci =
    -1.0071    2.4393
     0.5111    3.5262
```

## See Also

`logncdf` | `logninv` | `lognlike` | `lognpdf` | `lognrnd` | `lognstat` | `mle`

## Topics

“Lognormal Distribution” on page B-100

Introduced before R2006a

# logninv

Lognormal inverse cumulative distribution function

## Syntax

```
X = logninv(P,mu,sigma)
[X,XLO,XUP] = logninv(P,mu,sigma,pcov,alpha)
```

## Description

`X = logninv(P,mu,sigma)` returns values at `P` of the inverse lognormal cdf with distribution parameters `mu` and `sigma`. `mu` and `sigma` are the mean and standard deviation, respectively, of the associated normal distribution. `mu` and `sigma` can be vectors, matrices, or multidimensional arrays that all have the same size, which is also the size of `X`. A scalar input for `P`, `mu`, or `sigma` is expanded to a constant array with the same dimensions as the other inputs.

`[X,XLO,XUP] = logninv(P,mu,sigma,pcov,alpha)` returns confidence bounds for `X` when the input parameters `mu` and `sigma` are estimates. `pcov` is the covariance matrix of the estimated parameters. `alpha` specifies 100(1 - `alpha`)% confidence bounds. The default value of `alpha` is 0.05. `XLO` and `XUP` are arrays of the same size as `X` containing the lower and upper confidence bounds.

`logninv` computes confidence bounds for `P` using a normal approximation to the distribution of the estimate

$$\hat{\mu} + \sigma q$$

where  $q$  is the  $P$ th quantile from a normal distribution with mean 0 and standard deviation 1. The computed bounds give approximately the desired confidence level when you estimate `mu`, `sigma`, and `pcov` from large samples, but in smaller samples other methods of computing the confidence bounds might be more accurate.

The lognormal inverse function is defined in terms of the lognormal cdf as

$$x = F^{-1}(p | \mu, \sigma) = \{x : F(x | \mu, \sigma) = p\}$$

where

$$p = F(x | \mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} \int_0^x e^{-\frac{(\ln(t)-\mu)^2}{2\sigma^2}} dt$$

## Examples

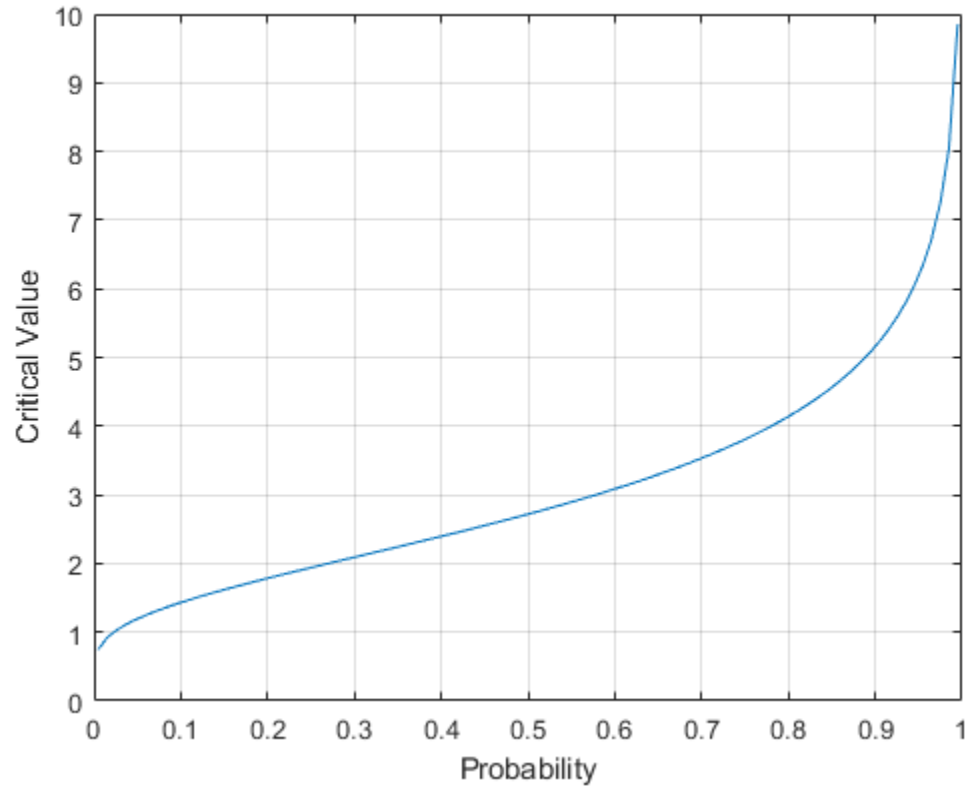
### Compute the Lognormal Distribution Inverse cdf

Compute the inverse cdf of a lognormal distribution with  $\mu = 0$  and  $\sigma = 0.5$ .

```
p = (0.005:0.01:0.995);  
crit = logninv(p,1,0.5);
```

Plot the inverse cdf.

```
figure;  
plot(p,crit)  
xlabel('Probability');  
ylabel('Critical Value');  
grid
```



## References

- [1] Evans, M., N. Hastings, and B. Peacock. *Statistical Distributions*. Hoboken, NJ: Wiley-Interscience, 2000. pp. 102–105.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`icdf` | `logncdf` | `lognfit` | `lognlike` | `lognpdf` | `lognrnd` | `lognstat`

### Topics

“Lognormal Distribution” on page B-100

Introduced before R2006a



# lognlike

Lognormal negative log-likelihood

## Syntax

```
nlogL = lognlike(params, data)
[nlogL, avar] = lognlike(params, data)
[...] = lognlike(params, data, censoring)
[...] = lognlike(params, data, censoring, freq)
```

## Description

`nlogL = lognlike(params, data)` returns the negative log-likelihood of data for the lognormal distribution with parameters `params`. `params(1)` is the mean of the associated normal distribution, `mu`, and `params(2)` is the standard deviation of the associated normal distribution, `sigma`. The values of `mu` and `sigma` are scalars, and the output `nlogL` is a scalar.

`[nlogL, avar] = lognlike(params, data)` returns the inverse of Fisher's information matrix. If the input parameter value in `params` is the maximum likelihood estimate, `avar` is its asymptotic variance. `avar` is based on the observed Fisher's information, not the expected information.

`[...] = lognlike(params, data, censoring)` accepts a Boolean vector, `censoring`, of the same size as `data`, which is 1 for observations that are right-censored and 0 for observations that are observed exactly.

`[...] = lognlike(params, data, censoring, freq)` accepts a frequency vector, `freq`, of the same size as `data`. The vector `freq` typically contains integer frequencies for the corresponding elements in `data`, but can contain any nonnegative values. Pass in `[]` for `censoring` to use its default value.

## See Also

`logncdf` | `lognfit` | `logninv` | `lognpdf` | `lognrnd` | `lognstat`

## **Topics**

“Lognormal Distribution” on page B-100

**Introduced before R2006a**

# lognpdf

Lognormal probability density function

## Syntax

`Y = lognpdf(X, mu, sigma)`

## Description

`Y = lognpdf(X, mu, sigma)` returns values at `X` of the lognormal pdf with distribution parameters `mu` and `sigma`. `mu` and `sigma` are the mean and standard deviation, respectively, of the associated normal distribution. `X`, `mu`, and `sigma` can be vectors, matrices, or multidimensional arrays that all have the same size, which is also the size of `Y`. A scalar input for `X`, `mu`, or `sigma` is expanded to a constant array with the same dimensions as the other inputs.

The lognormal pdf is

$$y = f(x | \mu, \sigma) = \frac{1}{x\sigma\sqrt{2\pi}} e^{-\frac{(\ln x - \mu)^2}{2\sigma^2}}$$

The normal and lognormal distributions are closely related. If  $X$  is distributed lognormally with parameters  $\mu$  and  $\sigma$ , then  $\log(X)$  is distributed normally with mean  $\mu$  and standard deviation  $\sigma$ .

The mean  $m$  and variance  $v$  of a lognormal random variable are functions of  $\mu$  and  $\sigma$  that can be calculated with the `lognstat` function. They are:

$$m = \exp\left(\mu + \frac{\sigma^2}{2}\right)$$

$$v = \exp\left(2\mu + \sigma^2\right)\left(\exp\left(\sigma^2\right) - 1\right)$$

So, a lognormal distribution with mean  $m$  and variance  $v$  has parameters

$$\mu = \log\left(m^2 / \sqrt{v + m^2}\right)$$

$$\sigma = \sqrt{\log(v / m^2 + 1)}$$

If you do not know the population mean and variance,  $m$  and  $v$ , for the lognormal distribution, you can estimate  $\mu$  and  $\sigma$  in the following way:

```
mu = mean(log(X))  
sigma = std(log(X))
```

The lognormal distribution is applicable when the quantity of interest must be positive, since  $\log(X)$  exists only when  $X$  is positive.

## Examples

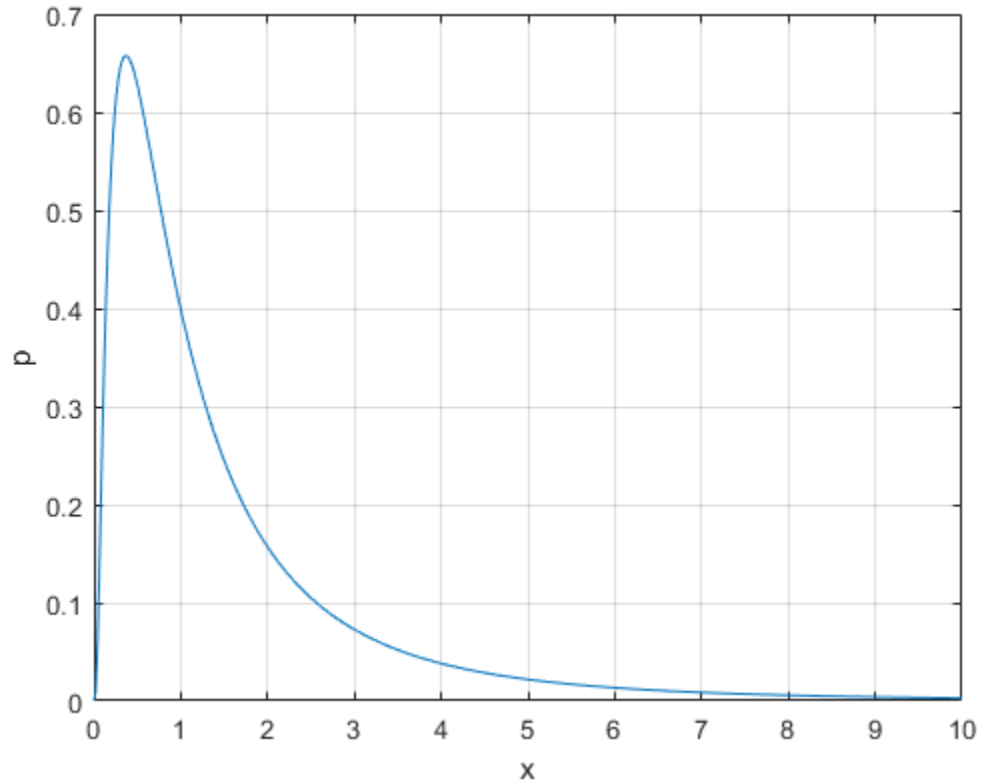
### Compute the Lognormal Distribution pdf

Compute the pdf of a lognormal distribution with `mu = 0` and `sigma = 1`.

```
x = (0:0.02:10);  
y = lognpdf(x,0,1);
```

Plot the pdf.

```
plot(x,y); grid;  
xlabel('x'); ylabel('p')
```



## References

- [1] Mood, A. M., F. A. Graybill, and D. C. Boes. *Introduction to the Theory of Statistics*. 3rd ed., New York: McGraw-Hill, 1974. pp. 540–541.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`logncdf` | `lognfit` | `logninv` | `lognlike` | `lognrnd` | `lognstat` | `pdf`

### Topics

“Lognormal Distribution” on page B-100

Introduced before R2006a

# lognrnd

Lognormal random numbers

## Syntax

```
R = lognrnd(mu, sigma)
R = lognrnd(mu, sigma, m, n, ...)
R = lognrnd(mu, sigma, [m, n, ...])
```

## Description

`R = lognrnd(mu, sigma)` returns an array of random numbers generated from the lognormal distribution with parameters `mu` and `sigma`. `mu` and `sigma` are the mean and standard deviation, respectively, of the associated normal distribution. `mu` and `sigma` can be vectors, matrices, or multidimensional arrays that have the same size, which is also the size of `R`. A scalar input for `mu` or `sigma` is expanded to a constant array with the same dimensions as the other input.

`R = lognrnd(mu, sigma, m, n, ...)` or `R = lognrnd(mu, sigma, [m, n, ...])` generates an `m-by-n-by-...` array. The `mu`, `sigma` parameters can each be scalars or arrays of the same size as `R`.

The normal and lognormal distributions are closely related. If  $X$  is distributed lognormally with parameters  $\mu$  and  $\sigma$ , then  $\log(X)$  is distributed normally with mean  $\mu$  and standard deviation  $\sigma$ .

The mean  $m$  and variance  $v$  of a lognormal random variable are functions of  $\mu$  and  $\sigma$  that can be calculated with the `lognstat` function. They are:

$$m = \exp\left(\mu + \frac{\sigma^2}{2}\right)$$

$$v = \exp\left(2\mu + \sigma^2\right)\left(\exp\left(\sigma^2\right) - 1\right)$$

A lognormal distribution with mean  $m$  and variance  $v$  has parameters

$$\mu = \log\left(m^2 / \sqrt{v + m^2}\right)$$
$$\sigma = \sqrt{\log(v / m^2 + 1)}$$

## Examples

Generate one million lognormally distributed random numbers with mean 1 and variance 2:

```
m = 1;
v = 2;
mu = log((m^2)/sqrt(v+m^2));
sigma = sqrt(log(v/(m^2)+1));

[M,V]= lognstat(mu,sigma)
M =
    1
V =
    2.0000

X = lognrnd(mu,sigma,1,1e6);

MX = mean(X)
MX =
    0.9974
VX = var(X)
VX =
    1.9776
```

## References

- [1] Evans, M., N. Hastings, and B. Peacock. *Statistical Distributions*. Hoboken, NJ: Wiley-Interscience, 2000. pp. 102–105.



## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

The generated code can return a different sequence of numbers than MATLAB if either of the following is true:

- The output is nonscalar.
- An input parameter is invalid for the distribution.

### See Also

`logncdf` | `lognfit` | `logninv` | `lognlike` | `lognpdf` | `lognstat` | `normrnd` | `random`

### Topics

“Lognormal Distribution” on page B-100

Introduced before R2006a

## lognstat

Lognormal mean and variance

### Syntax

```
[M,V] = lognstat(mu,sigma)
```

### Description

`[M,V] = lognstat(mu,sigma)` returns the mean of and variance of the lognormal distribution with parameters `mu` and `sigma`. `mu` and `sigma` are the mean and standard deviation, respectively, of the associated normal distribution. `mu` and `sigma` can be vectors, matrices, or multidimensional arrays that all have the same size, which is also the size of `M` and `V`. A scalar input for `mu` or `sigma` is expanded to a constant array with the same dimensions as the other input.

The normal and lognormal distributions are closely related. If  $X$  is distributed lognormally with parameters  $\mu$  and  $\sigma$ , then  $\log(X)$  is distributed normally with mean  $\mu$  and standard deviation  $\sigma$ .

The mean  $m$  and variance  $v$  of a lognormal random variable are functions of  $\mu$  and  $\sigma$  that can be calculated with the `lognstat` function. They are:

$$m = \exp(\mu + \sigma^2 / 2)$$

$$v = \exp(2\mu + \sigma^2)(\exp(\sigma^2) - 1)$$

A lognormal distribution with mean  $m$  and variance  $v$  has parameters

$$\mu = \log\left(m^2 / \sqrt{v + m^2}\right)$$

$$\sigma = \sqrt{\log(v / m^2 + 1)}$$

## Examples

Generate one million lognormally distributed random numbers with mean 1 and variance 2:

```
m = 1;
v = 2;
mu = log((m^2)/sqrt(v+m^2));
sigma = sqrt(log(v/(m^2)+1));

[M,V]= lognstat(mu,sigma)
M =
    1
V =
    2.0000

X = lognrnd(mu,sigma,1,1e6);

MX = mean(X)
MX =
    0.9974
VX = var(X)
VX =
    1.9776
```

## References

- [1] Mood, A. M., F. A. Graybill, and D. C. Boes. *Introduction to the Theory of Statistics*. 3rd ed., New York: McGraw-Hill, 1974. pp. 540–541.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

## See Also

`logncdf` | `lognfit` | `logninv` | `lognlike` | `lognpdf` | `lognrnd`

## Topics

“Lognormal Distribution” on page B-100

**Introduced before R2006a**

# logP

**Class:** CompactClassificationDiscriminant

Log unconditional probability density for discriminant analysis classifier

## Syntax

```
lp = logP(obj, Xnew)
```

## Description

`lp = logP(obj, Xnew)` returns the log of the unconditional probability density of each row of `Xnew`, computed using the discriminant analysis model `obj`.

## Input Arguments

**obj**

Discriminant analysis classifier, produced using `fitcdiscr`.

**Xnew**

Matrix where each row represents an observation, and each column represents a predictor. The number of columns in `Xnew` must equal the number of predictors in `obj`.

## Output Arguments

**lp**

Column vector with the same number of rows as `Xnew`. Each entry is the logarithm of the unconditional probability density of the corresponding row of `Xnew`.

## Examples

### Compute the Log Unconditional Probability Density of an Observation

Construct a discriminant analysis classifier for Fisher's iris data, and examine its prediction for an average measurement.

Load Fisher's iris data and construct a default discriminant analysis classifier.

```
load fisheriris
Mdl = fitcdiscr(meas, species);
```

Find the log probability of the discriminant model applied to an average iris.

```
logPAverage = logP(Mdl, mean(meas))
```

```
logPAverage =  
    -1.7254
```

## Definitions

### Unconditional Probability Density

The unconditional probability density of a point  $x$  of a discriminant analysis model is

$$P(x) = \sum_{k=1}^K P(x, k),$$

where  $P(x, k)$  is the conditional density of the model at  $x$  for class  $k$ , when the total number of classes is  $K$ .

The conditional density  $P(x, k)$  is

$$P(x, k) = P(k)P(x | k),$$

where  $P(k)$  is the prior probability of class  $k$ , and  $P(x|k)$  is the conditional density of  $x$  given class  $k$ . The conditional density function of the multivariate normal with mean  $\mu_k$  and covariance  $\Sigma_k$  at a point  $x$  is

$$P(x|k) = \frac{1}{(2\pi|\Sigma_k|)^{1/2}} \exp\left(-\frac{1}{2}(x-\mu_k)^T \Sigma_k^{-1}(x-\mu_k)\right),$$

where  $|\Sigma_k|$  is the determinant of  $\Sigma_k$ , and  $\Sigma_k^{-1}$  is the inverse matrix.

## See Also

`CompactClassificationDiscriminant` | `fitcdiscr` | `mahal`

## Topics

“Discriminant Analysis Classification” on page 20-2

## logP

**Class:** CompactClassificationNaiveBayes

Log unconditional probability density for naive Bayes classifier

## Syntax

```
lp = logP(Mdl, tbl)
lp = logP(Mdl, X)
```

## Description

`lp = logP(Mdl, tbl)` returns the log unconditional probability density on page 32-3646 of the observations (rows) in `tbl` using the naive Bayes model `Mdl`.

You can use `lp` to identify outliers in the training data.

`lp = logP(Mdl, X)` returns the log unconditional probability density on page 32-3646 of the observations (rows) in `X` using the naive Bayes model `Mdl`.

## Input Arguments

### **mdl** — Naive Bayes classifier

`ClassificationNaiveBayes` model | `CompactClassificationNaiveBayes` model

Naive Bayes classifier, specified as a `ClassificationNaiveBayes` model or `CompactClassificationNaiveBayes` model returned by `fitcnb` or `compact`, respectively.

### **tbl** — Sample data

table

Sample data, specified as a table. Each row of `tbl` corresponds to one observation, and each column corresponds to one predictor variable. Optionally, `tbl` can contain



additional columns for the response variable and observation weights. `tbl` must contain all the predictors used to train `Mdl`. Multi-column variables and cell arrays other than cell arrays of character vectors are not allowed.

If you trained `Mdl` using sample data contained in a `table`, then the input data for this method must also be in a `table`.

Data Types: `table`

### **x** — Predictor data

numeric matrix

Predictor data, specified as a numeric matrix.

Each row of `X` corresponds to one observation (also known as an instance or example), and each column corresponds to one variable (also known as a feature). The variables making up the columns of `X` must be the same as the variables that trained `Mdl`.

Data Types: `double` | `single`

## Output Arguments

### **lp** — Log of unconditional probability density

numeric column vector

Log of the unconditional probability density of the predictors, returned as a numeric column vector. `lp` has as many elements as rows in `X`, and each element is the log probability density of the corresponding row in `X`.

If any rows in `X` contain at least one `NaN`, then the corresponding element of `lp` is `NaN`.

## Examples

### Compute Unconditional Probability Densities of Observations

Load Fisher's iris data set.

```
load fisheriris
X = meas;    % Predictors
Y = species; % Response
```

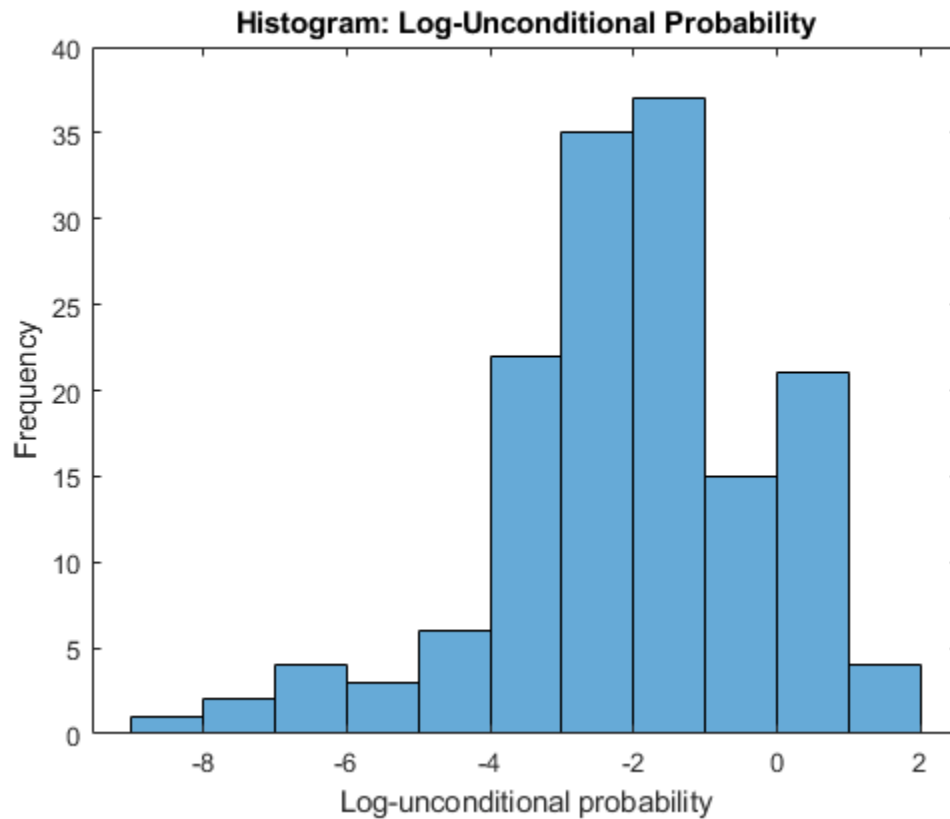
Train a naive Bayes classifier. It is good practice to specify the class order. Assume that each predictor is conditionally normally distributed given its label.

```
Mdl = fitcnb(X,Y,'ClassNames',{'setosa','versicolor','virginica'});
```

Mdl is a trained `ClassificationNaiveBayes` classifier.

Compute the unconditional probability densities of the in-sample observations.

```
lp = logP(Mdl,X);
histogram(lp)
xlabel 'Log-unconditional probability'
ylabel 'Frequency'
title 'Histogram: Log-Unconditional Probability'
```



Identify indices of observations having log-unconditional probability less than -7.

```
idx = find(lp < -7)
```

```
idx =
```

```
    61  
   118
```

## Definitions

### Unconditional Probability Density

The unconditional probability density of the predictors is its distribution marginalized over the classes.

In other words, the unconditional probability density is

$$P(X_1, \dots, X_P) = \sum_{k=1}^K P(X_1, \dots, X_P, Y = k) = \sum_{k=1}^K P(X_1, \dots, X_P | y = k) \pi(Y = k),$$

where  $\pi(Y = k)$  is the class prior probability. The conditional distribution of the data given the class ( $P(X_1, \dots, X_P | y = k)$ ) and the class prior probability distributions are training options (i.e., are specified when training the classifier).

### Prior Probability

The prior probability is the believed relative frequency that observations from a class occur in the population for each class.

## See Also

`ClassificationNaiveBayes` | `CompactClassificationNaiveBayes` | `fitcnb` | `predict`

## Topics

“Naive Bayes Classification” on page 21-2

# loss

**Class:** ClassificationKNN

Loss of  $k$ -nearest neighbor classifier

## Syntax

```
L = loss (mdl, tbl, ResponseVarName)
```

```
L = loss (mdl, tbl, Y)
```

```
L = loss (mdl, X, Y)
```

```
L = loss ( ____, Name, Value)
```

## Description

`L = loss (mdl, tbl, ResponseVarName)` returns a scalar representing how well `mdl` classifies the data in `tbl`, when `tbl.ResponseVarName` contains the true classifications.

When computing the loss, `loss` normalizes the class probabilities in `tbl.ResponseVarNames` to the class probabilities used for training, stored in the `Prior` property of `mdl`.

`L = loss (mdl, tbl, Y)` returns a scalar representing how well `mdl` classifies the data in `tbl`, when `Y` contains the true classifications.

When computing the loss, `loss` normalizes the class probabilities in `Y` to the class probabilities used for training, stored in the `Prior` property of `mdl`.

`L = loss (mdl, X, Y)` returns a scalar representing how well `mdl` classifies the data in `X`, when `Y` contains the true classifications.

When computing the loss, `loss` normalizes the class probabilities in `Y` to the class probabilities used for training, stored in the `Prior` property of `mdl`.

`L = loss ( ____, Name, Value)` returns the loss with additional options specified by one or more `Name, Value` pair arguments, using any of the previous syntaxes.

## Input Arguments

### **mdl** — Classifier model

classifier model object

*k*-nearest neighbor classifier model, returned as a classifier model object.

Note that using the 'CrossVal', 'Kfold', 'Holdout', 'Leaveout', or 'CVPartition' options results in a model of class `ClassificationPartitionedModel`. You cannot use a partitioned tree for prediction, so this kind of tree does not have a `predict` method.

Otherwise, `mdl` is of class `ClassificationKNN`, and you can use the `predict` method to make predictions.

### **tbl** — Sample data

table

Sample data used to train the model, specified as a table. Each row of `tbl` corresponds to one observation, and each column corresponds to one predictor variable. Optionally, `tbl` can contain one additional column for the response variable. Multi-column variables and cell arrays other than cell arrays of character vectors are not allowed.

If `tbl` contains the response variable used to train `mdl`, then you do not need to specify `ResponseVarName` or `Y`.

If you trained `mdl` using sample data contained in a table, then the input data for this method must also be in a table.

Data Types: `table`

### **ResponseVarName** — Response variable name

name of a variable in `tbl`

Response variable name, specified as the name of a variable in `tbl`. If `tbl` contains the response variable used to train `mdl`, then you do not need to specify `ResponseVarName`.

If you specify `ResponseVarName`, then you must do so as a character vector. For example, if the response variable is stored as `tbl.response`, then specify it as `'response'`. Otherwise, the software treats all columns of `tbl`, including `tbl.response`, as predictors.

The response variable must be a categorical or character array, logical or numeric vector, or cell array of character vectors. If the response variable is a character array, then each element must correspond to one row of the array.

### **x** — Matrix of predictor values

matrix

Matrix of predictor values. Each column of **X** represents one variable, and each row represents one observation.

### **Y** — Categorical variables

categorical array | cell array of character vectors | character array | logical vector | numeric vector

A categorical array, cell array of character vectors, character array, logical vector, or a numeric vector with the same number of rows as **X**. Each row of **Y** represents the classification of the corresponding row of **X**.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of **Name**, **Value** arguments. **Name** is the argument name and **Value** is the corresponding value. **Name** must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as **Name1**, **Value1**, ..., **NameN**, **ValueN**.

### **LossFun** — Loss function

'classiferror' (default) | 'binodeviance' | 'exponential' | 'hinge' | 'logit' | 'mincost' | 'quadratic' | function handle

Loss function, specified as the comma-separated pair consisting of 'LossFun' and a built-in, loss-function name or function handle.

- The following lists available loss functions. Specify one using its corresponding character vector.

Value	Description
'binodeviance'	Binomial deviance
'classiferror'	Classification error
'exponential'	Exponential

Value	Description
'hinge'	Hinge
'logit'	Logistic
'mincost'	Minimal expected misclassification cost (for classification scores that are posterior probabilities)
'quadratic'	Quadratic

'mincost' is appropriate for classification scores that are posterior probabilities  $k$ -nearest neighbor models return posterior probabilities as classification scores by default (see `predict`).

- Specify your own function using function handle notation.

Suppose that  $n$  be the number of observations in  $X$  and  $K$  be the number of distinct classes (`numel(mdl.ClassNames)`). Your function must have this signature

```
lossvalue = lossfun(C,S,W,Cost)
```

where:

- The output argument `lossvalue` is a scalar.
- You choose the function name (`lossfun`).
- $C$  is an  $n$ -by- $K$  logical matrix with rows indicating which class the corresponding observation belongs. The column order corresponds to the class order in `mdl.ClassNames`.

Construct  $C$  by setting  $C(p, q) = 1$  if observation  $p$  is in class  $q$ , for each row. Set all other elements of row  $p$  to 0.

- $S$  is an  $n$ -by- $K$  numeric matrix of classification scores. The column order corresponds to the class order in `mdl.ClassNames`.  $S$  is a matrix of classification scores, similar to the output of `predict`.
- $W$  is an  $n$ -by-1 numeric vector of observation weights. If you pass  $W$ , the software normalizes them to sum to 1.
- $Cost$  is a  $K$ -by- $K$  numeric matrix of misclassification costs. For example, `Cost = ones(K) - eye(K)` specifies a cost of 0 for correct classification, and 1 for misclassification.

Specify your function using `'LossFun', @lossfun`.



For more details on loss functions, see “Classification Loss” on page 32-3652.

Data Types: `char` | `function_handle`

### **Weights** — Observation weights

`ones(size(X,1),1)` (default) | numeric vector | name of a variable in `tbl`

Observation weights, specified as the comma-separated pair consisting of `'Weights'` and a numeric vector or the name of a variable in `tbl`.

If you specify `Weights` as a vector, then the size of `Weights` must be equal to the number of rows of `X` or `tbl`.

If you specify `Weights` as the name of a variable in `tbl`, you must do so as a character vector, such as `'W'`. For example, if the weights are stored as `tbl.w`, then specify it as `'w'`. Otherwise, the software treats all columns of `tbl`, including `tbl.w`, as predictors.

`loss` normalizes the weights so that observation weights in each class sum to the prior probability of that class. When you supply `Weights`, `loss` computes weighted classification loss.

Data Types: `single` | `double`

## Output Arguments

### **L** — Classification loss

scalar value

Classification loss on page 32-3652, returned as a scalar value. The meaning of the error depends on the values in `weights` and `lossfun`.

## Examples

### Loss Calculation

Construct a  $k$ -nearest neighbor classifier for the Fisher iris data, where  $k = 5$ .

Load the data.

```
load fisheriris
```

Construct a classifier for 5-nearest neighbors.

```
mdl = fitcknn(meas,species,'NumNeighbors',5);
```

Examine the loss of the classifier for a mean observation classified 'versicolor'.

```
X = mean(meas);  
Y = {'versicolor'};  
L = loss(mdl,X,Y)
```

```
L =  
  
    0
```

The classifier has no doubt that 'versicolor' is the correct classification (all five nearest neighbors classify as 'versicolor').

- “Examine Quality of KNN Classifier” on page 18-37
- “Predict Classification Using KNN Classifier” on page 18-38
- “Modify KNN Classifier” on page 18-39

## Definitions

### Classification Loss

Classification loss functions measure the predictive inaccuracy of classification models. When comparing the same type of loss among many models, lower loss indicates a better predictive model.

Suppose that:

- $L$  is the weighted average classification loss.
- $n$  is the sample size.
- For binary classification:

- $y_j$  is the observed class label. The software codes it as  $-1$  or  $1$  indicating the negative or positive class, respectively.
- $f(X_j)$  is the raw classification score for observation (row)  $j$  of the predictor data  $X$ .
- $m_j = y_j f(X_j)$  is the classification score for classifying observation  $j$  into the class corresponding to  $y_j$ . Positive values of  $m_j$  indicate correct classification and do not contribute much to the average loss. Negative values of  $m_j$  indicate incorrect classification and contribute to the average loss.
- For algorithms that support multiclass classification (that is,  $K \geq 3$ ):
  - $y_j^*$  is a vector of  $K - 1$  zeros, and a  $1$  in the position corresponding to the true, observed class  $y_j$ . For example, if the true class of the second observation is the third class and  $K = 4$ , then  $y_2^* = [0 \ 0 \ 1 \ 0]'$ . The order of the classes corresponds to the order in the `ClassNames` property of the input model.
  - $f(X_j)$  is the length  $K$  vector of class scores for observation  $j$  of the predictor data  $X$ . The order of the scores corresponds to the order of the classes in the `ClassNames` property of the input model.
  - $m_j = y_j^* f(X_j)$ . Therefore,  $m_j$  is the scalar classification score that the model predicts for the true, observed class.
- The weight for observation  $j$  is  $w_j$ . The software normalizes the observation weights so that they sum to the corresponding prior class probability. The software also normalizes the prior probabilities so they sum to 1. Therefore,

$$\sum_{j=1}^n w_j = 1.$$

The supported loss functions are:

- Binomial deviance, specified using `'LossFun', 'binodeviance'`. Its equation is

$$L = \sum_{j=1}^n w_j \log \{1 + \exp[-2m_j]\}.$$

- Exponential loss, specified using `'LossFun', 'exponential'`. Its equation is

$$L = \sum_{j=1}^n w_j \exp(-m_j).$$

- Classification error, specified using `'LossFun', 'classiferror'`. It is the weighted fraction of misclassified observations, with equation

$$L = \sum_{j=1}^n w_j I\{\hat{y}_j \neq y_j\}.$$

$\hat{y}_j$  is the class label corresponding to the class with the maximal posterior probability.  $I\{x\}$  is the indicator function.

- Hinge loss, specified using 'LossFun', 'hinge'. Its equation is

$$L = \sum_{j=1}^n w_j \max\{0, 1 - m_j\}.$$

- Logit loss, specified using 'LossFun', 'logit'. Its equation is

$$L = \sum_{j=1}^n w_j \log(1 + \exp(-m_j)).$$

- Minimal cost, specified using 'LossFun', 'mincost'. The software computes the weighted minimal cost using this procedure for observations  $j = 1, \dots, n$ :

- 1 Estimate the 1-by- $K$  vector of expected classification costs for observation  $j$

$$\gamma_j = f(X_j)'C.$$

$f(X_j)$  is the column vector of class posterior probabilities for binary and multiclass classification.  $C$  is the cost matrix the input model stores in the property `Cost`.

- 2 For observation  $j$ , predict the class label corresponding to the minimum, expected classification cost:

$$\hat{y}_j = \min_{j=1, \dots, K} (\gamma_j).$$

- 3 Using  $C$ , identify the cost incurred ( $c_j$ ) for making the prediction.

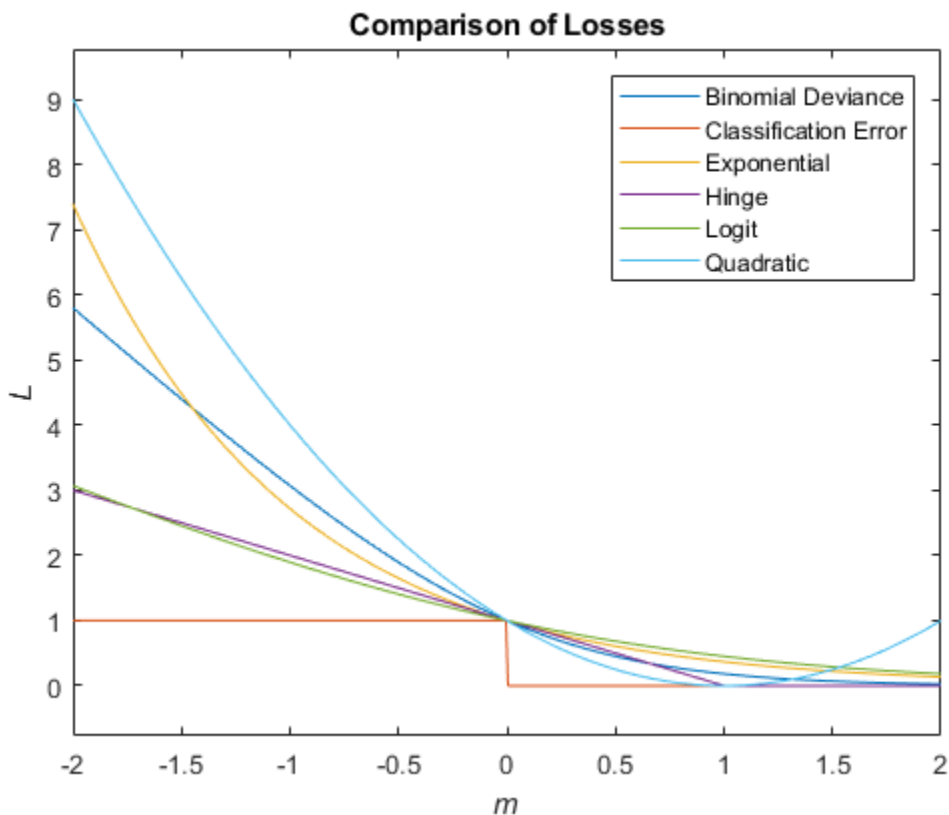
The weighted, average, minimum cost loss is

$$L = \sum_{j=1}^n w_j c_j.$$

- Quadratic loss, specified using 'LossFun', 'quadratic'. Its equation is

$$L = \sum_{j=1}^n w_j (1 - m_j)^2.$$

This figure compares some of the loss functions for one observation over  $m$  (some functions are normalized to pass through  $[0,1]$ ).



## True Misclassification Cost

There are two costs associated with KNN classification: the true misclassification cost per class, and the expected misclassification cost per observation.

You can set the true misclassification cost per class in the `Cost` name-value pair when you run `fitcknn`. `Cost(i, j)` is the cost of classifying an observation into class  $j$  if its true class is  $i$ . By default, `Cost(i, j) = 1` if  $i \neq j$ , and `Cost(i, j) = 0` if  $i = j$ . In other words, the cost is 0 for correct classification, and 1 for incorrect classification.

## Expected Cost

There are two costs associated with KNN classification: the true misclassification cost per class, and the expected misclassification cost per observation. The third output of `predict` is the expected misclassification cost per observation.

Suppose you have `Nobs` observations that you want to classify with a trained classifier `mdl`. Suppose you have `K` classes. You place the observations into a matrix `Xnew` with one observation per row. The command

```
[label,score,cost] = predict(mdl,Xnew)
```

returns, among other outputs, a `cost` matrix of size `Nobs`-by-`K`. Each row of the `cost` matrix contains the expected (average) cost of classifying the observation into each of the `K` classes. `cost(n,k)` is

$$\sum_{i=1}^K \hat{P}(i | X_{new}(n)) C(k | i),$$

where

- $K$  is the number of classes.
- $\hat{P}(i | X_{new}(n))$  is the posterior probability on page 32-4897 of class  $i$  for observation  $X_{new}(n)$ .
- $C(k | i)$  is the true misclassification cost on page 32-4898 of classifying an observation as  $k$  when its true class is  $i$ .

## Extended Capabilities

### Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays” (MATLAB).

## See Also

ClassificationKNN | edge | fitcknn | margin

## Topics

“Examine Quality of KNN Classifier” on page 18-37

“Predict Classification Using KNN Classifier” on page 18-38

“Modify KNN Classifier” on page 18-39

“Classification Using Nearest Neighbors” on page 18-16

## loss

**Class:** ClassificationLinear

Classification loss for linear classification models

## Syntax

```
L = loss(Mdl,X,Y)
```

```
L = loss( ____,Name,Value)
```

## Description

`L = loss(Mdl,X,Y)` returns the classification losses on page 32-3668 for the binary, linear classification model `Mdl` using predictor data in `X` and corresponding class labels in `Y`. `L` contains classification error rates for each regularization strength in `Mdl`.

`L = loss( ____,Name,Value)` uses any of the previous syntaxes and additional options specified by one or more `Name,Value` pair arguments. For example, you can specify that columns in the predictor data correspond to observations or specify the classification loss function.

## Input Arguments

**md1** — Binary, linear classification model

ClassificationLinear model object

Binary, linear classification model, specified as a ClassificationLinear model object. You can create a ClassificationLinear model object using `fitclinear`.

**x** — Predictor data

full matrix | sparse matrix



Predictor data, specified as an  $n$ -by- $p$  full or sparse matrix. This orientation of  $X$  indicates that rows correspond to individual observations, and columns correspond to individual predictor variables.

---

**Note** If you orient your predictor matrix so that observations correspond to columns and specify 'ObservationsIn', 'columns', then you might experience a significant reduction in computation time.

---

The length of  $Y$  and the number of observations in  $X$  must be equal.

Data Types: `single` | `double`

### **$Y$ — Class labels**

categorical array | character array | logical vector | vector of numeric values | cell array of character vectors

Class labels, specified as a categorical or character array, logical or numeric vector, or cell array of character vectors.

- The data type of  $Y$  must be the same `Mdl.ClassNames`.
- The distinct classes in  $Y$  must be a subset of `Mdl.ClassNames`.
- If  $Y$  is a character array, then each element must correspond to one row of the array.
- The length of  $Y$  and the number of observations in  $X$  must be equal.

Data Types: `categorical` | `cell` | `char` | `double` | `logical` | `single`

## **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### **LossFun — Loss function**

'classiferror' (default) | 'binodeviance' | 'exponential' | 'hinge' | 'logit' | 'mincost' | 'quadratic' | function handle

Loss function, specified as the comma-separated pair consisting of 'LossFun' and a built-in, loss-function name or function handle.

- The following lists available loss functions. Specify one using its corresponding character vector.

Value	Description
'binodeviance'	Binomial deviance
'classiferror'	Classification error
'exponential'	Exponential
'hinge'	Hinge
'logit'	Logistic
'mincost'	Minimal expected misclassification cost (for classification scores that are posterior probabilities)
'quadratic'	Quadratic

'mincost' is appropriate for classification scores that are posterior probabilities. For linear classification models, logistic regression learners return posterior probabilities as classification scores by default, but SVM learners do not (see `predict`).

- Specify your own function using function handle notation.

Let  $n$  be the number of observations in  $X$  and  $K$  be the number of distinct classes (`numel(Mdl.ClassNames)`,  $Mdl$  is the input model). Your function must have this signature

```
lossvalue = lossfun(C,S,W,Cost)
```

where:

- The output argument `lossvalue` is a scalar.
- You choose the function name (`lossfun`).
- $C$  is an  $n$ -by- $K$  logical matrix with rows indicating which class the corresponding observation belongs. The column order corresponds to the class order in `Mdl.ClassNames`.

Construct  $C$  by setting  $C(p, q) = 1$  if observation  $p$  is in class  $q$ , for each row. Set all other elements of row  $p$  to 0.

- $S$  is an  $n$ -by- $K$  numeric matrix of classification scores. The column order corresponds to the class order in `Mdl.ClassNames`.  $S$  is a matrix of classification scores, similar to the output of `predict`.
- $W$  is an  $n$ -by-1 numeric vector of observation weights. If you pass  $W$ , the software normalizes them to sum to 1.
- $Cost$  is a  $K$ -by- $K$  numeric matrix of misclassification costs. For example,  $Cost = \text{ones}(K) - \text{eye}(K)$  specifies a cost of 0 for correct classification, and 1 for misclassification.

Specify your function using `'LossFun', @lossfun`.

Data Types: `char` | `function_handle`

#### **ObservationsIn** — Predictor data observation dimension

`'rows'` (default) | `'columns'`

Predictor data observation dimension, specified as the comma-separated pair consisting of `'ObservationsIn'` and `'columns'` or `'rows'`.

---

**Note** If you orient your predictor matrix so that observations correspond to columns and specify `'ObservationsIn', 'columns'`, then you might experience a significant reduction in optimization-execution time.

---

Data Types: `char`

#### **Weights** — Observation weights

numeric vector of positive values

Observation weights, specified as the comma-separated pair consisting of `'Weights'` and a numeric vector of positive values. If you supply weights, `loss` computes the weighted classification loss.

Let  $n$  be the number of observations in  $X$ .

- `numel(Weights)` must be  $n$ .
- By default, `Weights` is `ones(n, 1)`.

`ClassificationLinear.loss` normalizes `Weights` to sum up to the value of the prior probability in the respective class.

Data Types: `double` | `single`

## Output Arguments

### **L** — Classification losses

numeric scalar | numeric row vector

Classification losses, returned as a numeric scalar or row vector. The interpretation of `L` depends on `Weights` and `LossFun`.

`L` is the same size as `Mdl.Lambda`. `L(j)` is the classification loss of the linear classification model trained using the regularization strength `Mdl.Lambda(j)`.

## Examples

### Estimate Test-Sample Classification Loss

Load the NLP data set.

```
load nlpdata
```

`X` is a sparse matrix of predictor data, and `Y` is a categorical vector of class labels. There are more than two classes in the data.

The models should identify whether the word counts in a web page are from the Statistics and Machine Learning Toolbox™ documentation. So, identify the labels that correspond to the Statistics and Machine Learning Toolbox™ documentation web pages.

```
Ystats = Y == 'stats';
```

Train a binary, linear classification model that can identify whether the word counts in a documentation web page are from the Statistics and Machine Learning Toolbox™ documentation. Specify to hold out 30% of the observations. Optimize the objective function using `SpaRSA`.

```
rng(1); % For reproducibility
CVMdl = fitclinear(X,Ystats,'Solver','sparsa','Holdout',0.30);
Cmdl = CVMdl.Trained{1};
```

CVMDL is a `ClassificationPartitionedLinear` model. It contains the property `Trained`, which is a 1-by-1 cell array holding a `ClassificationLinear` model that the software trained using the training set.

Extract the training and test data from the partition definition.

```
trainIdx = training(CVMDL.Partition);
testIdx = test(CVMDL.Partition);
```

Estimate the training- and test-sample classification error.

```
ceTrain = loss(CMdl,X(trainIdx,:),Ystats(trainIdx))
ceTest = loss(CMdl,X(testIdx,:),Ystats(testIdx))
```

```
ceTrain =
    1.3572e-04

ceTest =
    5.2804e-04
```

Because there is one regularization strength in `CMdl`, `ceTrain` and `ceTest` are numeric scalars.

### Specify Custom Classification Loss

Load the NLP data set. Preprocess the data as in “Estimate Test-Sample Classification Loss” on page 32-3662, and transpose the predictor data.

```
load nlpdata
Ystats = Y == 'stats';
X = X';
```

Train a binary, linear classification model. Specify to hold out 30% of the observations. Optimize the objective function using `SparSA`. Specify that the predictor observations correspond to columns.

```
rng(1); % For reproducibility
CVMDL = fitlinear(X,Ystats,'Solver','sparsa','Holdout',0.30,...
```

```
    'ObservationsIn', 'columns');  
CMdl = CVMdl.Trained{1};
```

CVMdl is a `ClassificationPartitionedLinear` model. It contains the property `Trained`, which is a 1-by-1 cell array holding a `ClassificationLinear` model that the software trained using the training set.

Extract the training and test data from the partition definition.

```
trainIdx = training(CVMdl.Partition);  
testIdx = test(CVMdl.Partition);
```

Create an anonymous function that measures linear loss, that is,

$$L = \frac{\sum_j -w_j y_j f_j}{\sum_j w_j}.$$

$w_j$  is the weight for observation  $j$ ,  $y_j$  is response  $j$  (-1 for the negative class, and 1 otherwise), and  $f_j$  is the raw classification score of observation  $j$ . Custom loss functions must be written in a particular form. For rules on writing a custom loss function, see the `LossFun` name-value pair argument.

```
linearloss = @(C,S,W,Cost) sum(-W.*sum(S.*C,2))/sum(W);
```

Estimate the training- and test-sample classification loss using the linear loss function.

```
ceTrain = loss(CMdl,X(:,trainIdx),Ystats(trainIdx), 'LossFun', linearloss, ...  
    'ObservationsIn', 'columns')  
ceTest = loss(CMdl,X(:,testIdx),Ystats(testIdx), 'LossFun', linearloss, ...  
    'ObservationsIn', 'columns')
```

```
ceTrain =  
  
    -7.8330  
  
ceTest =  
  
    -7.7383
```

## Find Good Lasso Penalty Using Classification Loss

To determine a good lasso-penalty strength for a linear classification model that uses a logistic regression learner, compare test-sample classification error rates.

Load the NLP data set. Preprocess the data as in “Specify Custom Classification Loss” on page 32-3663.

```
load nlpdata
Ystats = Y == 'stats';
X = X';

rng(10); % For reproducibility
Partition = cvpartition(Ystats, 'Holdout', 0.30);
testIdx = test(Partition);
XTest = X(:, testIdx);
YTest = Ystats(testIdx);
```

Create a set of 11 logarithmically-spaced regularization strengths from  $10^{-6}$  through  $10^{-0.5}$ .

```
Lambda = logspace(-6, -0.5, 11);
```

Train binary, linear classification models that use each of the regularization strengths. Optimize the objective function using SpARSA. Lower the tolerance on the gradient of the objective function to  $1e-8$ .

```
CVMdl = fitlinear(X, Ystats, 'ObservationsIn', 'columns', ...
    'CVPartition', Partition, 'Learner', 'logistic', 'Solver', 'sparsa', ...
    'Regularization', 'lasso', 'Lambda', Lambda, 'GradientTolerance', 1e-8)
```

```
CVMdl =
```

```
classreg.learning.partition.ClassificationPartitionedLinear
  CrossValidatedModel: 'Linear'
  ResponseName: 'Y'
  NumObservations: 31572
  KFold: 1
  Partition: [1x1 cvpartition]
  ClassNames: [0 1]
  ScoreTransform: 'none'
```

Extract the trained linear classification model.

```
Mdl = CVMdl.Trained{1}

Mdl =

  ClassificationLinear
    ResponseName: 'Y'
    ClassNames: [0 1]
    ScoreTransform: 'logit'
      Beta: [34023x11 double]
      Bias: [1x11 double]
      Lambda: [1x11 double]
    Learner: 'logistic'
```

Mdl is a `ClassificationLinear` model object. Because `Lambda` is a sequence of regularization strengths, you can think of Mdl as 11 models, one for each regularization strength in `Lambda`.

Estimate the test-sample classification error.

```
ce = loss(Mdl,X(:,testIdx),Ystats(testIdx),'ObservationsIn','columns');
```

Because there are 11 regularization strengths, `ce` is a 1-by-11 vector of classification error rates.

Higher values of `Lambda` lead to predictor variable sparsity, which is a good quality of a classifier. For each regularization strength, train a linear classification model using the entire data set and the same options as when you cross-validated the models. Determine the number of nonzero coefficients per model.

```
Mdl = fitclinear(X,Ystats,'ObservationsIn','columns',...
    'Learner','logistic','Solver','sparsa','Regularization','lasso',...
    'Lambda',Lambda,'GradientTolerance',1e-8);
numNZCoeff = sum(Mdl.Beta~=0);
```

In the same figure, plot the test-sample error rates and frequency of nonzero coefficients for each regularization strength. Plot all variables on the log scale.

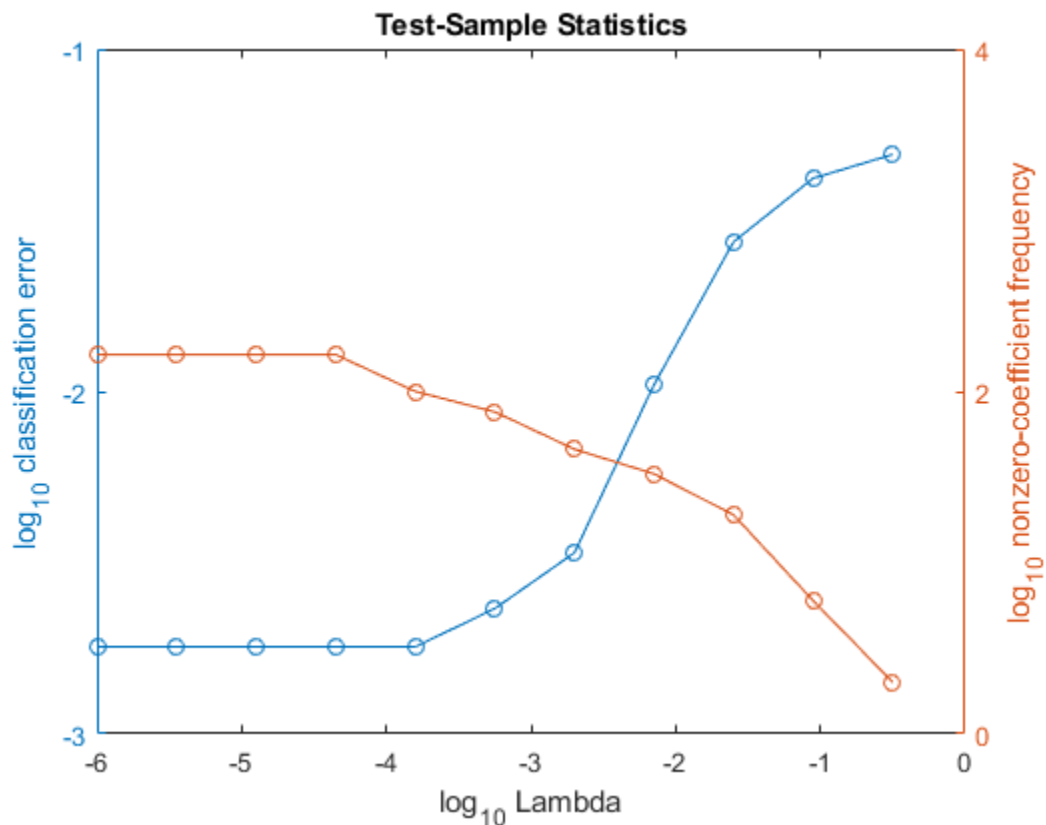
```
figure;
[h,hL1,hL2] = plotyy(log10(Lambda),log10(ce),...
```



```

log10(Lambda),log10(numNZCoeff + 1));
hL1.Marker = 'o';
hL2.Marker = 'o';
ylabel(h(1),'log_{10} classification error')
ylabel(h(2),'log_{10} nonzero-coefficient frequency')
xlabel('log_{10} Lambda')
title('Test-Sample Statistics')
hold off

```



Choose the index of the regularization strength that balances predictor variable sparsity and low classification error. In this case, a value between  $10^{-4}$  to  $10^{-1}$  should suffice.

```
idxFinal = 7;
```

Select the model from `Mdl` with the chosen regularization strength.

```
MdlFinal = selectModels(Mdl, idxFinal);
```

`MdlFinal` is a `ClassificationLinear` model containing one regularization strength. To estimate labels for new observations, pass `MdlFinal` and the new data to `predict`.

## Definitions

### Classification Loss

Classification loss functions measure the predictive inaccuracy of classification models. When comparing the same type of loss among many models, lower loss indicates a better predictive model.

Suppose that:

- $L$  is the weighted average classification loss.
- $n$  is the sample size.
- For binary classification:
  - $y_j$  is the observed class label. The software codes it as  $-1$  or  $1$  indicating the negative or positive class, respectively.
  - $f(X_j)$  is the raw classification score for observation (row)  $j$  of the predictor data  $X$ .
  - $m_j = y_j f(X_j)$  is the classification score for classifying observation  $j$  into the class corresponding to  $y_j$ . Positive values of  $m_j$  indicate correct classification and do not contribute much to the average loss. Negative values of  $m_j$  indicate incorrect classification and contribute to the average loss.
- For algorithms that support multiclass classification (that is,  $K \geq 3$ ):
  - $y_j^*$  is a vector of  $K - 1$  zeros, and a  $1$  in the position corresponding to the true, observed class  $y_j$ . For example, if the true class of the second observation is the third class and  $K = 4$ , then  $y_2^* = [0 \ 0 \ 1 \ 0]'$ . The order of the classes corresponds to the order in the `ClassNames` property of the input model.
  - $f(X_j)$  is the length  $K$  vector of class scores for observation  $j$  of the predictor data  $X$ . The order of the scores corresponds to the order of the classes in the `ClassNames` property of the input model.

- $m_j = y_j^* f(X_j)$ . Therefore,  $m_j$  is the scalar classification score that the model predicts for the true, observed class.
- The weight for observation  $j$  is  $w_j$ . The software normalizes the observation weights so that they sum to the corresponding prior class probability. The software also normalizes the prior probabilities so they sum to 1. Therefore,

$$\sum_{j=1}^n w_j = 1.$$

The supported loss functions are:

- Binomial deviance, specified using 'LossFun', 'binodeviance'. Its equation is

$$L = \sum_{j=1}^n w_j \log \{1 + \exp[-2m_j]\}.$$

- Exponential loss, specified using 'LossFun', 'exponential'. Its equation is

$$L = \sum_{j=1}^n w_j \exp(-m_j).$$

- Classification error, specified using 'LossFun', 'classiferror'. It is the weighted fraction of misclassified observations, with equation

$$L = \sum_{j=1}^n w_j I\{\hat{y}_j \neq y_j\}.$$

$\hat{y}_j$  is the class label corresponding to the class with the maximal posterior probability.  $I\{x\}$  is the indicator function.

- Hinge loss, specified using 'LossFun', 'hinge'. Its equation is

$$L = \sum_{j=1}^n w_j \max\{0, 1 - m_j\}.$$

- Logit loss, specified using 'LossFun', 'logit'. Its equation is

$$L = \sum_{j=1}^n w_j \log(1 + \exp(-m_j)).$$

- Minimal cost, specified using 'LossFun', 'mincost'. The software computes the weighted minimal cost using this procedure for observations  $j = 1, \dots, n$ :

- 1 Estimate the 1-by- $K$  vector of expected classification costs for observation  $j$

$$\gamma_j = f(X_j)'C.$$

$f(X_j)$  is the column vector of class posterior probabilities for binary and multiclass classification.  $C$  is the cost matrix the input model stores in the property `Cost`.

- 2 For observation  $j$ , predict the class label corresponding to the minimum, expected classification cost:

$$\hat{y}_j = \min_{j=1,\dots,K}(\gamma_j).$$

- 3 Using  $C$ , identify the cost incurred ( $c_j$ ) for making the prediction.

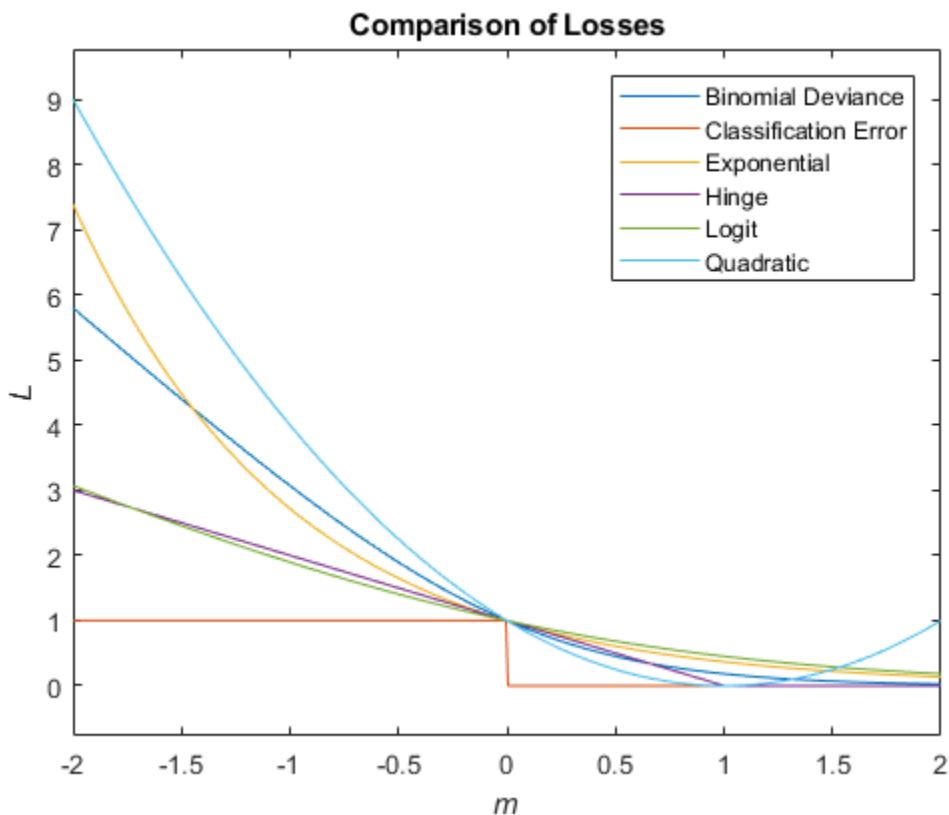
The weighted, average, minimum cost loss is

$$L = \sum_{j=1}^n w_j c_j.$$

- Quadratic loss, specified using `'LossFun', 'quadratic'`. Its equation is

$$L = \sum_{j=1}^n w_j (1 - m_j)^2.$$

This figure compares some of the loss functions for one observation over  $m$  (some functions are normalized to pass through  $[0,1]$ ).



## Algorithms

By default, observation weights are prior class probabilities. If you supply weights using `weights`, then the software normalizes them to sum to the prior probabilities in the respective classes. The software uses the renormalized weights to estimate the weighted classification loss.

## Extended Capabilities

### Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays” (MATLAB).

### See Also

`ClassificationLinear` | `fitclinear` | `predict`

**Introduced in R2016a**

## loss

**Class:** CompactClassificationDiscriminant

Classification error

## Syntax

```
L = loss(obj, X, Y)
```

```
L = loss(obj, X, Y, Name, Value)
```

## Description

`L = loss(obj, X, Y)` returns the classification loss on page 32-3677, which is a scalar representing how well `obj` classifies the data in `X`, when `Y` contains the true classifications.

When computing the loss, `loss` normalizes the class probabilities in `Y` to the class probabilities used for training, stored in the `Prior` property of `obj`.

`L = loss(obj, X, Y, Name, Value)` returns the loss with additional options specified by one or more `Name, Value` pair arguments.

## Input Arguments

**obj**

Discriminant analysis classifier of class `ClassificationDiscriminant` or `CompactClassificationDiscriminant`, typically constructed with `fitcdiscr`.

**X**

Matrix where each row represents an observation, and each column represents a predictor. The number of columns in `X` must equal the number of predictors in `obj`.

**Y**

Class labels, with the same data type as exists in `obj`. The number of elements of `Y` must equal the number of rows of `X`.

**Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

**LossFun**

Built-in, loss-function name (character vector in the table) or function handle.

- The following lists available loss functions. Specify one using its corresponding character vector.

Value	Description
'binodeviance'	Binomial deviance
'classiferror'	Classification error
'exponential'	Exponential
'hinge'	Hinge
'logit'	Logistic
'mincost'	Minimal expected misclassification cost (for classification scores that are posterior probabilities)
'quadratic'	Quadratic

'mincost' is appropriate for classification scores that are posterior probabilities.

Discriminant analysis models return posterior probabilities as classification scores by default (see `predict`).

- Specify your own function using function handle notation.

Suppose that  $n$  be the number of observations in `X` and  $K$  be the number of distinct classes (`numel(Mdl.ClassNames)`). Your function must have this signature

```
lossvalue = lossfun(C,S,W,Cost)
```



where:

- The output argument `lossvalue` is a scalar.
- You choose the function name (`lossfun`).
- `C` is an  $n$ -by- $K$  logical matrix with rows indicating which class the corresponding observation belongs. The column order corresponds to the class order in `Mdl.ClassNames`.

Construct `C` by setting  $C(p, q) = 1$  if observation  $p$  is in class  $q$ , for each row. Set all other elements of row  $p$  to 0.

- `S` is an  $n$ -by- $K$  numeric matrix of classification scores. The column order corresponds to the class order in `Mdl.ClassNames`. `S` is a matrix of classification scores, similar to the output of `predict`.
- `W` is an  $n$ -by-1 numeric vector of observation weights. If you pass `W`, the software normalizes them to sum to 1.
- `Cost` is a  $K$ -by- $K$  numeric matrix of misclassification costs. For example, `Cost = ones(K) - eye(K)` specifies a cost of 0 for correct classification, and 1 for misclassification.

Specify your function using `'LossFun', @lossfun`.

For more details on loss functions, see “Classification Loss” on page 32-3677.

**Default:** `'mincost'`

### **Weights**

Numeric vector of length  $N$ , where  $N$  is the number of rows of `X`. `weights` are nonnegative. `loss` normalizes the weights so that observation weights in each class sum to the prior probability of that class. When you supply `weights`, `loss` computes weighted classification loss.

**Default:** `ones(N, 1)`

## Output Arguments

**L**

Classification loss on page 32-3677, a scalar. The interpretation of `L` depends on the values in `weights` and `lossfun`.

## Examples

### Estimate Classification Error

Load Fisher's iris data set.

```
load fisheriris
```

Train a discriminant analysis model using all observations in the data.

```
Mdl = fitcdiscr(meas, species);
```

Estimate the classification error of the model using the training observations.

```
L = loss(Mdl, meas, species)
```

```
L =
```

```
0.0200
```

Alternatively, if `Mdl` is not compact, then you can estimate the training-sample classification error by passing `Mdl` to `resubLoss`.

## Definitions

### Classification Loss

Classification loss functions measure the predictive inaccuracy of classification models. When comparing the same type of loss among many models, lower loss indicates a better predictive model.

Suppose that:

- $L$  is the weighted average classification loss.
- $n$  is the sample size.
- For binary classification:
  - $y_j$  is the observed class label. The software codes it as  $-1$  or  $1$  indicating the negative or positive class, respectively.
  - $f(X_j)$  is the raw classification score for observation (row)  $j$  of the predictor data  $X$ .
  - $m_j = y_j f(X_j)$  is the classification score for classifying observation  $j$  into the class corresponding to  $y_j$ . Positive values of  $m_j$  indicate correct classification and do not contribute much to the average loss. Negative values of  $m_j$  indicate incorrect classification and contribute to the average loss.
- For algorithms that support multiclass classification (that is,  $K \geq 3$ ):
  - $y_j^*$  is a vector of  $K - 1$  zeros, and a  $1$  in the position corresponding to the true, observed class  $y_j$ . For example, if the true class of the second observation is the third class and  $K = 4$ , then  $y_2^* = [0 \ 0 \ 1 \ 0]'$ . The order of the classes corresponds to the order in the `ClassNames` property of the input model.
  - $f(X_j)$  is the length  $K$  vector of class scores for observation  $j$  of the predictor data  $X$ . The order of the scores corresponds to the order of the classes in the `ClassNames` property of the input model.
  - $m_j = y_j^{*'} f(X_j)$ . Therefore,  $m_j$  is the scalar classification score that the model predicts for the true, observed class.

- The weight for observation  $j$  is  $w_j$ . The software normalizes the observation weights so that they sum to the corresponding prior class probability. The software also normalizes the prior probabilities so they sum to 1. Therefore,

$$\sum_{j=1}^n w_j = 1.$$

The supported loss functions are:

- Binomial deviance, specified using 'LossFun', 'binodeviance'. Its equation is

$$L = \sum_{j=1}^n w_j \log \{1 + \exp[-2m_j]\}.$$

- Exponential loss, specified using 'LossFun', 'exponential'. Its equation is

$$L = \sum_{j=1}^n w_j \exp(-m_j).$$

- Classification error, specified using 'LossFun', 'classiferror'. It is the weighted fraction of misclassified observations, with equation

$$L = \sum_{j=1}^n w_j I\{\hat{y}_j \neq y_j\}.$$

$\hat{y}_j$  is the class label corresponding to the class with the maximal posterior probability.  $I\{x\}$  is the indicator function.

- Hinge loss, specified using 'LossFun', 'hinge'. Its equation is

$$L = \sum_{j=1}^n w_j \max\{0, 1 - m_j\}.$$

- Logit loss, specified using 'LossFun', 'logit'. Its equation is

$$L = \sum_{j=1}^n w_j \log(1 + \exp(-m_j)).$$

- Minimal cost, specified using 'LossFun', 'mincost'. The software computes the weighted minimal cost using this procedure for observations  $j = 1, \dots, n$ :

- 1 Estimate the 1-by- $K$  vector of expected classification costs for observation  $j$

$$\gamma_j = f(X_j)'C.$$

$f(X_j)$  is the column vector of class posterior probabilities for binary and multiclass classification.  $C$  is the cost matrix the input model stores in the property `Cost`.

- 2 For observation  $j$ , predict the class label corresponding to the minimum, expected classification cost:

$$\hat{y}_j = \min_{j=1,\dots,K}(\gamma_j).$$

- 3 Using  $C$ , identify the cost incurred ( $c_j$ ) for making the prediction.

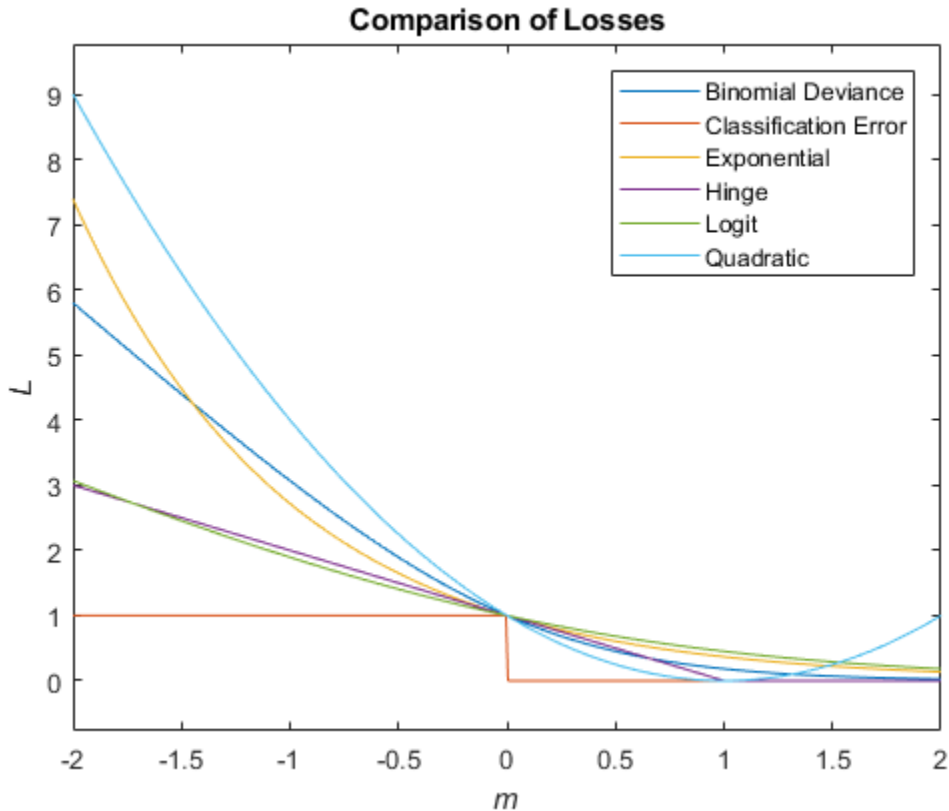
The weighted, average, minimum cost loss is

$$L = \sum_{j=1}^n w_j c_j.$$

- Quadratic loss, specified using 'LossFun', 'quadratic'. Its equation is

$$L = \sum_{j=1}^n w_j (1 - m_j)^2.$$

This figure compares some of the loss functions for one observation over  $m$  (some functions are normalized to pass through  $[0,1]$ ).



## Posterior Probability

The posterior probability that a point  $z$  belongs to class  $j$  is the product of the prior probability and the multivariate normal density. The density function of the multivariate normal with mean  $\mu_j$  and covariance  $\Sigma_j$  at a point  $z$  is

$$P(x | k) = \frac{1}{(2\pi|\Sigma_k|)^{1/2}} \exp\left(-\frac{1}{2}(x - \mu_k)^T \Sigma_k^{-1} (x - \mu_k)\right),$$

where  $|\Sigma_k|$  is the determinant of  $\Sigma_k$ , and  $\Sigma_k^{-1}$  is the inverse matrix.

Let  $P(k)$  represent the prior probability of class  $k$ . Then the posterior probability that an observation  $x$  is of class  $k$  is

$$\hat{P}(k | x) = \frac{P(x | k)P(k)}{P(x)},$$

where  $P(x)$  is a normalization constant, the sum over  $k$  of  $P(x | k)P(k)$ .

## Prior Probability

The prior probability is one of three choices:

- 'uniform' — The prior probability of class  $k$  is one over the total number of classes.
- 'empirical' — The prior probability of class  $k$  is the number of training samples of class  $k$  divided by the total number of training samples.
- Custom — The prior probability of class  $k$  is the  $k$ th element of the `prior` vector. See `fitcdiscr`.

After creating a classification model (`Mdl`) you can set the prior using dot notation:

```
Mdl.Prior = v;
```

where `v` is a vector of positive elements representing the frequency with which each element occurs. You do not need to retrain the classifier when you set a new prior.

## Cost

The matrix of expected costs per observation is defined in “Cost” on page 20-9.

## Extended Capabilities

### Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays” (MATLAB).

## See Also

`ClassificationDiscriminant` | `edge` | `fitcdiscr` | `margin` | `predict`

## Topics

“Discriminant Analysis Classification” on page 20-2



# loss

**Class:** CompactClassificationECOC

Classification loss for multiclass, error-correcting output codes model

## Syntax

```
L = loss(Mdl,tbl,ResponseVarName)
```

```
L = loss(Mdl,tbl,Y)
```

```
L = loss(Mdl,X,Y)
```

```
L = loss( ____,Name,Value)
```

## Description

`L = loss(Mdl,tbl,ResponseVarName)` returns the classification loss ( $L$ ), a scalar representing how well the trained, multiclass, error-correcting output code (ECOC) model `Mdl` classifies the predictor data (`tbl`) as compared to the true class labels (`ResponseVarName`). Each row of `tbl` and `ResponseVarName` is an observation.

`L = loss(Mdl,tbl,Y)` returns the classification loss ( $L$ ), a scalar representing how well the trained error-correcting output code (ECOC) multiclass classifier `Mdl` classifies the predictor data (`tbl`) as compared to the true class labels (`Y`). Each row of `tbl` and `Y` is an observation.

`L = loss(Mdl,X,Y)` returns the classification loss ( $L$ ), a scalar representing how well the trained error-correcting output code (ECOC) multiclass classifier `Mdl` classifies the predictor data (`X`) as compared to the true class labels (`Y`). Each row of `X` and `Y` is an observation.

`L = loss( ____,Name,Value)` returns the classification loss with additional options specified by one or more `Name,Value` pair arguments, using any of the previous syntaxes. For example, you can specify a decoding scheme, classification loss function, or verbosity level.

## Input Arguments

### **mdl** — Full or compact, multiclass ECOC model

`ClassificationECOC` model object | `CompactClassificationECOC` model object

Full or compact, multiclass ECOC model, specified as a `ClassificationECOC` or `CompactClassificationECOC` model object.

To create a full or compact ECOC model, see `ClassificationECOC` or `CompactClassificationECOC`.

### **tbl** — Sample data

table

Sample data, specified as a table. Each row of `tbl` corresponds to one observation, and each column corresponds to one predictor variable. Optionally, `tbl` can contain additional columns for the response variable and observation weights. `tbl` must contain all the predictors used to train `Mdl`. Multi-column variables and cell arrays other than cell arrays of character vectors are not allowed.

If you trained `Mdl` using sample data contained in a `table`, then the input data for this method must also be in a `table`.

---

**Note** If `Mdl.BinaryLearners` contains linear classification models (that is, `ClassificationLinear` model objects), then you cannot specify sample data in a `table`. Instead, pass a full or sparse matrix (`X`) and class labels (`Y`).

---

Data Types: `table`

### **ResponseVarName** — Response variable name

name of a variable in `tbl`

Response variable name, specified as the name of a variable in `tbl`.

You must specify `ResponseVarName` as a character vector. For example, if the response variable `y` is stored as `tbl.y`, then specify it as `'y'`. Otherwise, the software treats all columns of `tbl`, including `y`, as predictors when training the model.

The response variable must be a categorical or character array, logical or numeric vector, or cell array of character vectors. If the response variable is a character array, then each element must correspond to one row of the array.

Data Types: `char`

### **x — Predictor data**

numeric matrix

Predictor data, specified as a numeric matrix.

Each row of  $X$  corresponds to one observation, and each column corresponds to one variable. The variables composing the columns of  $X$  should be the same as the variables that trained the `Mdl` classifier.

The length of  $Y$  and the number of rows of  $X$  must be equal.

Data Types: `double` | `single`

### **y — Class labels**

categorical array | character array | logical vector | vector of numeric values | cell array of character vectors

Class labels, specified as a categorical or character array, logical or numeric vector, or cell array of character vectors.  $Y$  must be the same as the data type of `Mdl.ClassNames`.

The length of  $Y$  and the number of rows of  $X$  must be equal.

## **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### **BinaryLoss — Binary learner loss function**

'hamming' | 'linear' | 'logit' | 'exponential' | 'binodeviance' | 'hinge' | 'quadratic' | function handle

Binary learner loss function, specified as the comma-separated pair consisting of 'BinaryLoss' and a built-in, loss-function name or function handle.

- This table contains names and descriptions of the built-in functions, where  $y_j$  is a class label for a particular binary learner (in the set  $\{-1, 1, 0\}$ ),  $s_j$  is the score for observation  $j$ , and  $g(y_j, s_j)$  is the binary loss formula.

Value	Description	Score Domain	$g(y_j, s_j)$
'binodeviance'	Binomial deviance	$(-\infty, \infty)$	$\log[1 + \exp(-2y_j s_j)] / [2\log(2)]$
'exponential'	Exponential	$(-\infty, \infty)$	$\exp(-y_j s_j) / 2$
'hamming'	Hamming	$[0, 1]$ or $(-\infty, \infty)$	$[1 - \text{sign}(y_j s_j)] / 2$
'hinge'	Hinge	$(-\infty, \infty)$	$\max(0, 1 - y_j s_j) / 2$
'linear'	Linear	$(-\infty, \infty)$	$(1 - y_j s_j) / 2$
'logit'	Logistic	$(-\infty, \infty)$	$\log[1 + \exp(-y_j s_j)] / [2\log(2)]$
'quadratic'	Quadratic	$[0, 1]$	$[1 - y_j(2s_j - 1)]^2 / 2$

The software normalizes the binary losses such that the loss is 0.5 when  $y_j = 0$ . Also, the software calculates the mean binary loss for each class.

- For a custom binary loss function, e.g., `customFunction`, specify its function handle `'BinaryLoss', @customFunction`.

`customFunction` should have this form:

```
bLoss = customFunction(M, s)
```

where:

- `M` is the  $K$ -by- $L$  coding matrix stored in `Mdl.CodingMatrix`.
- `s` is the 1-by- $L$  row vector of classification scores.
- `bLoss` is the classification loss. This scalar aggregates the binary losses for every learner in a particular class. For example, you can use the mean binary loss to aggregate the loss over the learners for each class.
- $K$  is the number of classes.
- $L$  is the number of binary learners.

For an example of passing a custom binary loss function, see “Predict Test-Sample Labels of ECOC Models Using Custom Binary Loss Function” on page 32-4933.

By default, if all binary learners are:

- SVMs or linear classification models of SVM learners, then `BinaryLoss` is 'hinge'
- Ensembles trained by `AdaboostM1` or `GentleBoost`, then `BinaryLoss` is 'exponential'
- Ensembles trained by `LogitBoost`, then `BinaryLoss` is 'binodeviance'
- Linear classification models of logistic regression learners, or you specify to predict class posterior probabilities (i.e., set 'FitPosterior', 1 in `fitcecoc`), then `BinaryLoss` is 'quadratic'

Otherwise, the default `BinaryLoss` is 'hamming'. To check the default value, use dot notation to display the `BinaryLoss` property of the trained model at the command line.

Example: `'BinaryLoss', 'binodeviance'`

Data Types: char | function\_handle

### Decoding — Decoding scheme

'lossweighted' (default) | 'lossbased'

Decoding scheme that aggregates the binary losses on page 32-4941, specified as the comma-separated pair consisting of 'Decoding' and 'lossweighted' or 'lossbased'.

Example: `'Decoding', 'lossbased'`

Data Types: char

### LossFun — Loss function

'classiferror' (default) | function handle

Loss function, specified as the comma-separated pair consisting of 'LossFun' and a function handle or 'classiferror'.

You can:

- Specify the built-in function 'classiferror' for classification error, i.e., the proportion of misclassified observations.
- Specify your own function using function handle notation.

Suppose that  $n = \text{size}(X, 1)$  is the sample size and  $k$  is the number of classes. Your function must have the signature `lossvalue = lossfun(C, S, W, Cost)`, where:

- The output argument `lossvalue` is a scalar.
- You choose the function name (`lossfun`).
- `C` is an  $n$ -by- $k$  logical matrix with rows indicating which class the corresponding observation belongs. The column order corresponds to the class order in `Mdl.ClassNames`.

Construct `C` by setting  $C(p, q) = 1$  if observation  $p$  is in class  $q$ , for each row. Set all other elements of row  $p$  to 0.

- `S` is an  $n$ -by- $k$  numeric matrix of negated loss values for classes. Each row corresponds to an observation. The column order corresponds to the class order in `Mdl.ClassNames`. `S` resembles the output argument `NegLoss` of `predict`.
- `W` is an  $n$ -by-1 numeric vector of observation weights. If you pass `W`, the software normalizes its elements to sum to 1.
- `Cost` is a  $k$ -by- $k$  numeric matrix of misclassification costs. For example, `Cost = ones(K) - eye(K)` specifies a cost of 0 for correct classification, and 1 for misclassification.

Specify your function using `'LossFun', @lossfun`.

Data Types: `function_handle` | `char`

### ObservationsIn — Predictor data observation dimension

`'rows'` (default) | `'columns'`

Predictor data observation dimension, specified as the comma-separated pair consisting of `'ObservationsIn'` and `'columns'` or `'rows'`. `Mdl.BinaryLearners` must contain linear classification models.

---

**Note** If you orient your predictor matrix so that observations correspond to columns and specify `'ObservationsIn', 'columns'`, you can experience a significant reduction in execution time.

---

Data Types: `char`

### Options — Estimation options

`[]` (default) | structure array returned by `statset`

Estimation options, specified as the comma-separated pair consisting of 'Options' and a structure array returned by `statset`.

To invoke parallel computing:

- You need a Parallel Computing Toolbox license.
- Specify 'Options', `statset('UseParallel',1)`.

### **Verbose — Verbosity level**

0 (default) | 1

Verbosity level, specified as the comma-separated pair consisting of 'Verbose' and 0 or 1. Verbose controls the amount of diagnostic messages that the software displays in the Command Window.

If `Verbose` is 0, then the software does not display diagnostic messages. Otherwise, the software displays diagnostic messages.

Example: 'Verbose',1

Data Types: `single` | `double`

### **Weights — Observation weights**

`ones(size(X,1))` (default) | numeric vector | name of a variable in `tbl`

Observation weights, specified as the comma-separated pair consisting of 'Weights' and a numeric vector or the name of a variable in `tbl`. If you supply weights, then `loss` computes the weighted loss.

`Weights` requires the same length as the number of observations in `X` or `tbl`.

If you specify `Weights` as the name of a variable in `tbl`, you must do so as a character vector. For example, if the weights are stored as `tbl.w`, then specify it as 'w'. Otherwise, the software treats all columns of `tbl`, including `tbl.w`, as predictors.

If you do not specify your own loss function (using `LossFun`), then the software normalizes `Weights` to sum up to the value of the prior probability in the respective class.

If `Mdl.BinaryLearners` contains linear classification models, then you must specify a vector.

## Output Arguments

### **L** — Classification loss

numeric scalar | numeric row vector

Classification loss, returned as a numeric scalar or row vector. **L** is a generalization or resubstitution quality measure. Its interpretation depends on the loss function and weighting scheme, but, in general, better classifiers yield smaller loss values.

If `Mdl.BinaryLearners` contains linear classification models, then **L** is a 1-by- $\ell$  vector, where  $\ell$  is the number of regularization strengths in the linear classification models (i.e., `numel(Mdl.BinaryLearners{1}.Lambda)`). `L(j)` is the loss for the model trained using regularization strength `Mdl.BinaryLearners{1}.Lambda(j)`.

Otherwise, **L** is a scalar.

## Examples

### Determine the Test Sample Loss of ECOC Models

Load Fisher's iris data set.

```
load fisheriris
X = meas;
Y = categorical(species);
classOrder = unique(Y); % Class order
rng(1); % For reproducibility
```

Train an ECOC model using SVM binary classifiers, and specify a 15% holdout sample. It is good practice to standardize the predictors and define the class order. Specify to standardize the predictors using an SVM template.

```
t = templateSVM('Standardize',1);
CVMdl = fitcecoc(X,Y,'Holdout',0.15,'Learners',t,'ClassNames',classOrder);
CMdl = CVMdl.Trained{1}; % Extract trained, compact classifier
testInds = test(CVMdl.Partition); % Extract the test indices
XTest = X(testInds,:);
YTest = Y(testInds,:);
```



CVMDL is a ClassificationPartitionedECOC model. It contains the property `Trained`, which is a 1-by-1 cell array holding a CompactClassificationECOC model that the software trained using the training set.

Estimate the test-sample loss.

```
L = loss(CMDL, XTest, YTest)
```

```
L =
```

```
0
```

The ECOC model correctly classifies all out-of-sample irises.

### Determine ECOC Model Quality Using a Custom Loss

Suppose that it is interesting to know how well a model classifies a particular class. This example shows how to pass such a custom loss function to `loss`.

Load Fisher's iris data set.

```
load fisheriris
X = meas;
Y = categorical(species);
n = numel(Y); % Sample size
classOrder = unique(Y) % Class order
K = numel(classOrder); % Number of classes
rng(1) % For reproducibility
```

```
classOrder =
```

```
3x1 categorical array
```

```
setosa
versicolor
virginica
```

Train an ECOC model using SVM binary classifiers and specifying a 15% holdout sample. It is good practice to define the class order. Specify to standardize the predictors using an SVM template.

```
t = templateSVM('Standardize',1);
CVMdl = fitcecoc(X,Y,'Holdout',0.15,'Learners',t,'ClassNames',classOrder);
CMdl = CVMdl.Trained{1};           % Extract trained, compact classifier
testInds = test(CVMdl.Partition); % Extract the test indices
XTest = X(testInds,:);
YTest = Y(testInds,:);
```

CVMdl is a ClassificationPartitionedECOC model. It contains the property Trained, which is a 1-by-1 cell array holding a CompactClassificationECOC model that the software trained using the training set.

Compute the negated losses for the test-sample observations.

```
[~,negLoss] = predict(CMdl,XTest);
```

Create a function that takes the minimal loss for each observation, and then averages the minimal losses across all observations.

```
lossfun = @(~,S,~,~)mean(min(-S,[],2));
```

Compute the test-sample custom loss.

```
loss(CMdl,XTest,YTest,'LossFun',lossfun)
```

```
ans =
```

```
    0.0033
```

The average, minimal, binary loss in the test sample is 0.0033.

- “Quick Start Parallel Computing for Statistics and Machine Learning Toolbox” on page 30-2

## Definitions

### Classification Error

The classification error is a binary classification error measure that has the form

$$L = \frac{\sum_{j=1}^n w_j e_j}{\sum_{j=1}^n w_j},$$

where:

- $w_j$  is the weight for observation  $j$ . The software renormalizes the weights to sum to 1.
- $e_j = 1$  if the predicted class of observation  $j$  differs from its true class, and 0 otherwise.

In other words, it is the proportion of observations that the classifier misclassifies.

### Binary Loss

A binary loss is a function of the class and classification score that determines how well a binary learner classifies an observation into the class.

Let:

- $m_{kj}$  be element  $(k,j)$  of the coding design matrix  $M$  (i.e., the code corresponding to class  $k$  of binary learner  $j$ )
- $s_j$  be the score of binary learner  $j$  for an observation
- $g$  be the binary loss function
- $\hat{k}$  be the predicted class for the observation

In loss-based decoding [15], the class producing the minimum sum of the binary losses over binary learners determines the predicted class of an observation, that is,

$$\hat{k} = \operatorname{argmin}_k \sum_{j=1}^L |m_{kj}| g(m_{kj}, s_j).$$

In loss-weighted decoding [15], the class producing the minimum average of the binary losses over binary learners determines the predicted class of an observation, that is,

$$\hat{k} = \underset{k}{\operatorname{argmin}} \frac{\sum_{j=1}^L |m_{kj}| g(m_{kj}, s_j)}{\sum_{j=1}^L |m_{kj}|}.$$

Allwein et al. [2] suggest that loss-weighted decoding improves classification accuracy by keeping loss values for all classes in the same dynamic range.

This table summarizes the supported loss functions, where  $y_j$  is a class label for a particular binary learner (in the set  $\{-1, 1, 0\}$ ),  $s_j$  is the score for observation  $j$ , and  $g(y_j, s_j)$ .

Value	Description	Score Domain	$g(y_j, s_j)$
'binodeviance'	Binomial deviance	$(-\infty, \infty)$	$\log[1 + \exp(-2y_j s_j)] / [2\log(2)]$
'exponential'	Exponential	$(-\infty, \infty)$	$\exp(-y_j s_j) / 2$
'hamming'	Hamming	$[0, 1]$ or $(-\infty, \infty)$	$[1 - \operatorname{sign}(y_j s_j)] / 2$
'hinge'	Hinge	$(-\infty, \infty)$	$\max(0, 1 - y_j s_j) / 2$
'linear'	Linear	$(-\infty, \infty)$	$(1 - y_j s_j) / 2$
'logit'	Logistic	$(-\infty, \infty)$	$\log[1 + \exp(-y_j s_j)] / [2\log(2)]$
'quadratic'	Quadratic	$[0, 1]$	$[1 - y_j(2s_j - 1)]^2 / 2$

The software normalizes the binary losses such that the loss is 0.5 when  $y_j = 0$ , and aggregates using the average of the binary learners [2].

Do not confuse the binary loss with the overall classification loss (specified by the `LossFun` name-value pair argument of `predict` and `loss`), e.g., classification error, which measures how well an ECOC classifier performs as a whole.

## Algorithms

If you trained `Mdl` specifying to standardize the predictor data, then the software standardizes the columns of  $X$  using the corresponding means and standard deviations that the software stored in `Mdl.BinaryLearner{j}.Mu` and `Mdl.BinaryLearner{j}.Sigma` for learner  $j$ .

## References

- [1] Allwein, E., R. Schapire, and Y. Singer. “Reducing multiclass to binary: A unifying approach for margin classifiers.” *Journal of Machine Learning Research*. Vol. 1, 2000, pp. 113–141.
- [2] Escalera, S., O. Pujol, and P. Radeva. “On the decoding process in ternary error-correcting output codes.” *IEEE Transactions on Pattern Analysis and Machine Intelligence*. Vol. 32, Issue 7, 2010, pp. 120–134.
- [3] Escalera, S., O. Pujol, and P. Radeva. “Separability of ternary codes for sparse designs of error-correcting output codes.” *Pattern Recogn.* Vol. 30, Issue 3, 2009, pp. 285–297.

## Extended Capabilities

### Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays” (MATLAB).

### See Also

`ClassificationECOC` | `ClassificationLinear` | `CompactClassificationECOC` | `fitcecoc` | `predict` | `resubLoss`

### Topics

“Quick Start Parallel Computing for Statistics and Machine Learning Toolbox” on page 30-2  
“Reproducibility in Parallel Statistical Computations” on page 30-14  
“Concepts of Parallel Computing in Statistics and Machine Learning Toolbox” on page 30-8

## loss

**Class:** CompactClassificationEnsemble

Classification error

## Syntax

```
L = loss(ens, tbl, ResponseVarName)
L = loss(ens, tbl, Y)
L = loss(ens, X, Y)
L = loss( ____, Name, Value)
```

## Description

`L = loss(ens, tbl, ResponseVarName)` returns the classification error for ensemble `ens` computed using table of predictors `tbl` and true class labels `tbl.ResponseVarName`.

`L = loss(ens, tbl, Y)` returns the classification error for ensemble `ens` computed using table of predictors `tbl` and true class labels `Y`.

`L = loss(ens, X, Y)` returns the classification error for ensemble `ens` computed using matrix of predictors `X` and true class labels `Y`.

`L = loss( ____, Name, Value)` computes classification error with additional options specified by one or more `Name, Value` pair arguments, using any of the previous syntaxes.

When computing the loss, `loss` normalizes the class probabilities in `ResponseVarName` or `Y` to the class probabilities used for training, stored in the `Prior` property of `ens`.

## Input Arguments

### **ens**

Classification ensemble created with `fitcensemble`, or a compact classification ensemble created with `compact`.

### **tbl**

Sample data, specified as a table. Each row of `tbl` corresponds to one observation, and each column corresponds to one predictor variable. `tbl` must contain all of the predictors used to train the model. Multi-column variables and cell arrays other than cell arrays of character vectors are not allowed.

If you trained `ens` using sample data contained in a `table`, then the input data for this method must also be in a `table`.

### **ResponseVarName**

Response variable name, specified as the name of a variable in `tbl`. The response variable must be a numeric vector.

You must specify `ResponseVarName` as a character vector. For example, if the response variable `Y` is stored as `tbl.Y`, then specify it as `'Y'`. Otherwise, the software treats all columns of `tbl`, including `Y`, as predictors when training the model.

### **X**

Matrix of data to classify. Each row of `X` represents one observation, and each column represents one predictor. `X` must have the same number of columns as the data used to train `ens`. `X` should have the same number of rows as the number of elements in `Y`.

If you trained `ens` using sample data contained in a matrix, then the input data for this method must also be in a matrix.

### **Y**

Classification of `tbl` or `X`. `Y` should be of the same type as the classification used to train `ens`, and its number of elements should equal the number of rows of `tbl` or `X`.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### learners

Indices of weak learners in the ensemble ranging from 1 to `ens.NumTrained`. `loss` uses only these learners for calculating loss.

**Default:** `1:NumTrained`

### Lossfun

Loss function, specified as the comma-separated pair consisting of 'LossFun' and a built-in, loss-function name or function handle.

- The following lists available loss functions. Specify one using its corresponding character vector.

Value	Description
'binodeviance'	Binomial deviance
'classiferror'	Classification error
'exponential'	Exponential
'hinge'	Hinge
'logit'	Logistic
'mincost'	Minimal expected misclassification cost (for classification scores that are posterior probabilities)
'quadratic'	Quadratic

'mincost' is appropriate for classification scores that are posterior probabilities.

- Bagged and subspace ensembles return posterior probabilities by default (`ens.Method` is 'Bag' or 'Subspace').
- If the ensemble method is 'AdaBoostM1', 'AdaBoostM2', 'GentleBoost', or 'LogitBoost', then, to use posterior probabilities as classification scores, you must specify the double-logit score transform by entering



```
ens.ScoreTransform = 'doublelogit';
```

- For all other ensemble methods, the software does not support posterior probabilities as classification scores.
- Specify your own function using function handle notation.

Suppose that  $n$  be the number of observations in  $X$  and  $K$  be the number of distinct classes (`numel(ens.ClassNames)`, `ens` is the input model). Your function must have this signature

```
lossvalue = lossfun(C,S,W,Cost)
```

where:

- The output argument `lossvalue` is a scalar.
- You choose the function name (`lossfun`).
- $C$  is an  $n$ -by- $K$  logical matrix with rows indicating which class the corresponding observation belongs. The column order corresponds to the class order in `ens.ClassNames`.

Construct  $C$  by setting  $C(p, q) = 1$  if observation  $p$  is in class  $q$ , for each row. Set all other elements of row  $p$  to 0.

- $S$  is an  $n$ -by- $K$  numeric matrix of classification scores. The column order corresponds to the class order in `ens.ClassNames`.  $S$  is a matrix of classification scores, similar to the output of `predict`.
- $W$  is an  $n$ -by-1 numeric vector of observation weights. If you pass  $W$ , the software normalizes them to sum to 1.
- $Cost$  is a  $K$ -by- $K$  numeric matrix of misclassification costs. For example, `Cost = ones(K) - eye(K)` specifies a cost of 0 for correct classification, and 1 for misclassification.

Specify your function using `'LossFun', @lossfun`.

For more details on loss functions, see “Classification Loss” on page 32-3701.

**Default:** `'classiferror'`

**mode**

Meaning of the output  $L$ :

- 'ensemble' —  $L$  is a scalar value, the loss for the entire ensemble.
- 'individual' —  $L$  is a vector with one element per trained learner.
- 'cumulative' —  $L$  is a vector in which element  $J$  is obtained by using learners  $1:J$  from the input list of learners.

**Default:** 'ensemble'

#### **UseObsForLearner**

A logical matrix of size  $N$ -by- $T$ , where:

- $N$  is the number of rows of  $X$ .
- $T$  is the number of weak learners in `ens`.

When `UseObsForLearner(i, j)` is true, learner  $j$  is used in predicting the class of row  $i$  of  $X$ .

**Default:** `true(N, T)`

#### **weights**

Vector of observation weights, with nonnegative entries. The length of `weights` must equal the number of rows in  $X$ . When you specify `weights`, `loss` normalizes the weights so that observation weights in each class sum to the prior probability of that class.

**Default:** `ones(size(X, 1), 1)`

## Output Arguments

### **$L$**

Classification loss on page 32-3701, by default the fraction of misclassified data.  $L$  can be a vector, and can mean different things, depending on the name-value pair settings.

## Examples

## Estimate Classification Error

Load Fisher's iris data set.

```
load fisheriris
```

Boost 100 classification trees using AdaBoostM2.

```
ens = fitensemble(meas,species,'AdaBoostM2',100,'Tree');
```

Estimate the classification error of the model using the training observations.

```
L = loss(ens,meas,species)
```

```
L =
```

```
    0.0333
```

Alternatively, if `ens` is not compact, then you can estimate the training-sample classification error by passing `ens` to `resubLoss`.

## Definitions

### Classification Loss

Classification loss functions measure the predictive inaccuracy of classification models. When comparing the same type of loss among many models, lower loss indicates a better predictive model.

Suppose that:

- $L$  is the weighted average classification loss.
- $n$  is the sample size.
- For binary classification:
  - $y_j$  is the observed class label. The software codes it as  $-1$  or  $1$  indicating the negative or positive class, respectively.
  - $f(X_j)$  is the raw classification score for observation (row)  $j$  of the predictor data  $X$ .

- $m_j = y_j f(X_j)$  is the classification score for classifying observation  $j$  into the class corresponding to  $y_j$ . Positive values of  $m_j$  indicate correct classification and do not contribute much to the average loss. Negative values of  $m_j$  indicate incorrect classification and contribute to the average loss.
- For algorithms that support multiclass classification (that is,  $K \geq 3$ ):
  - $y_j^*$  is a vector of  $K - 1$  zeros, and a 1 in the position corresponding to the true, observed class  $y_j$ . For example, if the true class of the second observation is the third class and  $K = 4$ , then  $y_2^* = [0 \ 0 \ 1 \ 0]'$ . The order of the classes corresponds to the order in the `ClassNames` property of the input model.
  - $f(X_j)$  is the length  $K$  vector of class scores for observation  $j$  of the predictor data  $X$ . The order of the scores corresponds to the order of the classes in the `ClassNames` property of the input model.
  - $m_j = y_j^* f(X_j)$ . Therefore,  $m_j$  is the scalar classification score that the model predicts for the true, observed class.
- The weight for observation  $j$  is  $w_j$ . The software normalizes the observation weights so that they sum to the corresponding prior class probability. The software also normalizes the prior probabilities so they sum to 1. Therefore,

$$\sum_{j=1}^n w_j = 1.$$

The supported loss functions are:

- Binomial deviance, specified using `'LossFun', 'binodeviance'`. Its equation is

$$L = \sum_{j=1}^n w_j \log \{1 + \exp[-2m_j]\}.$$

- Exponential loss, specified using `'LossFun', 'exponential'`. Its equation is

$$L = \sum_{j=1}^n w_j \exp(-m_j).$$

- Classification error, specified using `'LossFun', 'classiferror'`. It is the weighted fraction of misclassified observations, with equation

$$L = \sum_{j=1}^n w_j I\{\hat{y}_j \neq y_j\}.$$

$\hat{y}_j$  is the class label corresponding to the class with the maximal posterior probability.  $I\{x\}$  is the indicator function.

- Hinge loss, specified using 'LossFun', 'hinge'. Its equation is

$$L = \sum_{j=1}^n w_j \max\{0, 1 - m_j\}.$$

- Logit loss, specified using 'LossFun', 'logit'. Its equation is

$$L = \sum_{j=1}^n w_j \log(1 + \exp(-m_j)).$$

- Minimal cost, specified using 'LossFun', 'mincost'. The software computes the weighted minimal cost using this procedure for observations  $j = 1, \dots, n$ :

- 1 Estimate the 1-by- $K$  vector of expected classification costs for observation  $j$

$$\gamma_j = f(X_j)'C.$$

$f(X_j)$  is the column vector of class posterior probabilities for binary and multiclass classification.  $C$  is the cost matrix the input model stores in the property `Cost`.

- 2 For observation  $j$ , predict the class label corresponding to the minimum, expected classification cost:

$$\hat{y}_j = \min_{j=1, \dots, K} (\gamma_j).$$

- 3 Using  $C$ , identify the cost incurred ( $c_j$ ) for making the prediction.

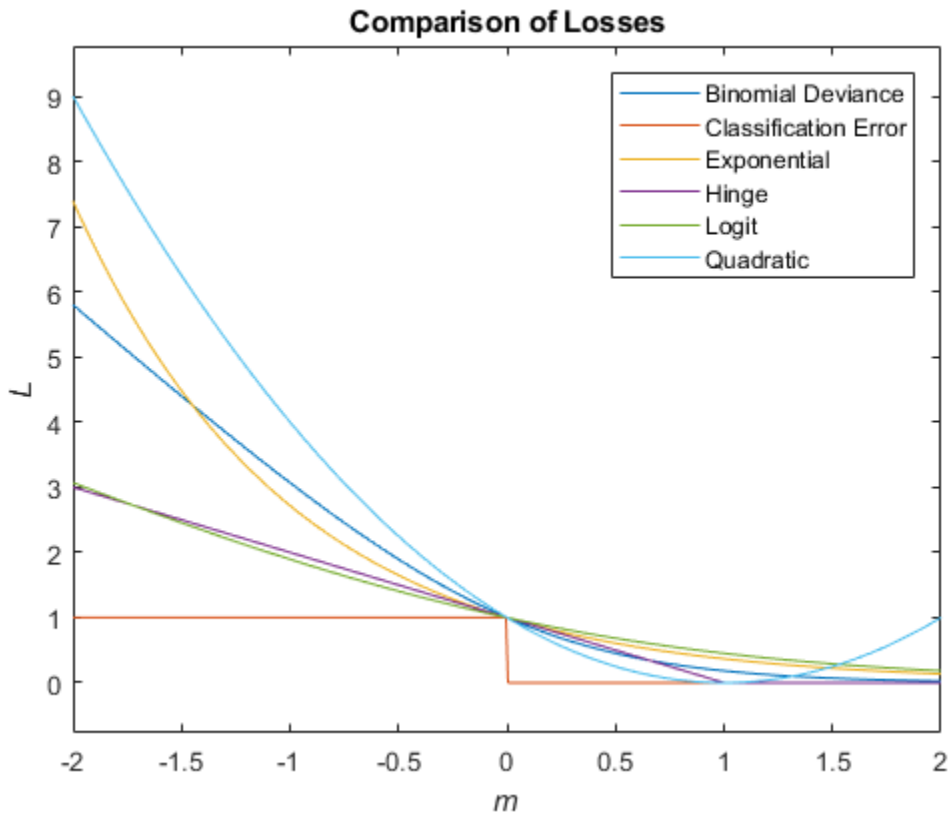
The weighted, average, minimum cost loss is

$$L = \sum_{j=1}^n w_j c_j.$$

- Quadratic loss, specified using 'LossFun', 'quadratic'. Its equation is

$$L = \sum_{j=1}^n w_j (1 - m_j)^2.$$

This figure compares some of the loss functions for one observation over  $m$  (some functions are normalized to pass through  $[0, 1]$ ).



## Extended Capabilities

### Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays” (MATLAB).

## See Also

edge | loss | margin | predict

## loss

**Class:** CompactClassificationNaiveBayes

Classification error for naive Bayes classifier

## Syntax

```
L = loss(Mdl, tbl, ResponseVarName)
```

```
L = loss(Mdl, tbl, Y)
```

```
L = loss(Mdl, X, Y)
```

```
L = loss( ____, Name, Value)
```

## Description

`L = loss(Mdl, tbl, ResponseVarName)` returns the minimum classification loss (see “Classification Loss” on page 32-3713), a scalar representing how well the trained naive Bayes classifier `Mdl` classifies the predictor data in table `tbl`) as compared to the true class labels in `tbl.ResponseVarName`.

`loss` normalizes the class probabilities in `tbl.ResponseVarName` to the prior class probabilities `fitcnb` used for training, stored in the `Prior` property of `Mdl`.

`L = loss(Mdl, tbl, Y)` returns the minimum classification loss (`L`), a scalar representing how well the trained naive Bayes classifier `Mdl` classifies the predictor data in table `tbl`) as compared to the true class labels in `Y`.

`loss` normalizes the class probabilities in `Y` to the prior class probabilities `fitcnb` used for training, stored in the `Prior` property of `Mdl`.

`L = loss(Mdl, X, Y)` returns the minimum classification loss (`L`), a scalar representing how well the trained naive Bayes classifier `Mdl` classifies the predictor data (`X`) as compared to the true class labels (`Y`).



`loss` normalizes the class probabilities in `Y` to the prior class probabilities `fitcnb` used for training, stored in the `Prior` property of `Mdl`.

`L = loss( ___, Name, Value)` returns the classification loss with additional options specified by one or more `Name, Value` pair arguments, using any of the previous syntaxes.

## Input Arguments

### **mdl** — Naive Bayes classifier

`ClassificationNaiveBayes model` | `CompactClassificationNaiveBayes model`

Naive Bayes classifier, specified as a `ClassificationNaiveBayes model` or `CompactClassificationNaiveBayes model` returned by `fitcnb` or `compact`, respectively.

### **tbl** — Sample data

`table`

Sample data, specified as a table. Each row of `tbl` corresponds to one observation, and each column corresponds to one predictor variable. Optionally, `tbl` can contain additional columns for the response variable and observation weights. `tbl` must contain all the predictors used to train `Mdl`. Multi-column variables and cell arrays other than cell arrays of character vectors are not allowed.

If you trained `Mdl` using sample data contained in a `table`, then the input data for this method must also be in a table.

Data Types: `table`

### **ResponseVarName** — Response variable name

name of a variable in `tbl`

Response variable name, specified as the name of a variable in `tbl`.

You must specify `ResponseVarName` as a character vector. For example, if the response variable `y` is stored as `tbl.y`, then specify it as `'y'`. Otherwise, the software treats all columns of `tbl`, including `y`, as predictors when training the model.

The response variable must be a categorical or character array, logical or numeric vector, or cell array of character vectors. If the response variable is a character array, then each element must correspond to one row of the array.

**x — Predictor data**

numeric matrix

Predictor data, specified as a numeric matrix.

Each row of `X` corresponds to one observation (also known as an instance or example), and each column corresponds to one variable (also known as a feature). The variables making up the columns of `X` should be the same as the variables that trained `Mdl`.

The length of `Y` and the number of rows of `X` must be equal.

Data Types: `double` | `single`

**y — Class labels**

categorical array | character array | logical vector | vector of numeric values | cell array of character vectors

Class labels, specified as a categorical or character array, logical or numeric vector, or cell array of character vectors. `Y` must be the same as the data type of `Mdl.ClassNames`.

The length of `Y` and the number of rows of `tbl` or `X` must be equal.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( `' '` ). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

**LossFun — Loss function**

`'classiferror'` (default) | `'binodeviance'` | `'exponential'` | `'hinge'` | `'logit'` | `'mincost'` | `'quadratic'` | function handle

Loss function, specified as the comma-separated pair consisting of `'LossFun'` and a built-in, loss-function name or function handle.

- The following lists available loss functions. Specify one using its corresponding character vector.

Value	Description
'binodeviance'	Binomial deviance
'classiferror'	Classification error
'exponential'	Exponential
'hinge'	Hinge
'logit'	Logistic
'mincost'	Minimal expected misclassification cost (for classification scores that are posterior probabilities)
'quadratic'	Quadratic

'mincost' is appropriate for classification scores that are posterior probabilities. Naive Bayes models return posterior probabilities as classification scores by default (see `predict`).

- Specify your own function using function handle notation.

Suppose that  $n$  be the number of observations in  $X$  and  $K$  be the number of distinct classes (`numel(Mdl.ClassNames)`,  $Mdl$  is the input model). Your function must have this signature

```
lossvalue = lossfun(C,S,W,Cost)
```

where:

- The output argument `lossvalue` is a scalar.
- You choose the function name (`lossfun`).
- $C$  is an  $n$ -by- $K$  logical matrix with rows indicating which class the corresponding observation belongs. The column order corresponds to the class order in `Mdl.ClassNames`.

Construct  $C$  by setting  $C(p, q) = 1$  if observation  $p$  is in class  $q$ , for each row. Set all other elements of row  $p$  to 0.

- $S$  is an  $n$ -by- $K$  numeric matrix of classification scores. The column order corresponds to the class order in `Mdl.ClassNames`.  $S$  is a matrix of classification scores, similar to the output of `predict`.
- $W$  is an  $n$ -by-1 numeric vector of observation weights. If you pass  $W$ , the software normalizes them to sum to 1.

- `Cost` is a  $K$ -by- $K$  numeric matrix of misclassification costs. For example, `Cost = ones(K) - eye(K)` specifies a cost of 0 for correct classification, and 1 for misclassification.

Specify your function using `'LossFun', @lossfun`.

For more details on loss functions, see “Classification Loss” on page 32-3713.

Data Types: `char` | `function_handle`

### **weights** — Observation weights

`ones(size(X,1),1)` (default) | numeric vector | name of a variable in `tbl`

Observation weights, specified as the comma-separated pair consisting of `'Weights'` and a numeric vector or the name of a variable in `tbl`. The software weighs the observations in each row of `X` or `tbl` with the corresponding weight in `Weights`.

If you specify `Weights` as a vector, then the size of `Weights` must be equal to the number of rows of `X` or `tbl`.

If you specify `Weights` as the name of a variable in `tbl`, you must do so as a character vector. For example, if the weights are stored as `tbl.w`, then specify it as `'w'`. Otherwise, the software treats all columns of `tbl`, including `tbl.w`, as predictors.

If you do not specify your own loss function, then the software normalizes `Weights` to add up to 1.

Data Types: `double`

## Output Arguments

### **L** — Classification loss

scalar

Classification loss, returned as a scalar. `L` is a generalization or resubstitution quality measure. Its interpretation depends on the loss function and weighting scheme, but, in general, better classifiers yield smaller loss values.

## Examples

### Determine Test Sample Minimum Cost Loss of Naive Bayes Classifiers

Load Fisher's iris data set.

```
load fisheriris
X = meas;      % Predictors
Y = species;   % Response
rng(1);        % For reproducibility
```

Train a naive Bayes classifier. Specify a 15% holdout sample for testing. It is good practice to specify the class order. Assume that each predictor is conditionally normally distributed given its label.

```
CVMDL = fitcnb(X,Y,'ClassNames',{'setosa','versicolor','virginica'},...
    'Holdout',0.15);
CMdl = CVMDL.Trained{1}; % Extract the trained, compact classifier
testInds = test(CVMDL.Partition); % Extract the test indices
XTest = X(testInds,:);
YTest = Y(testInds);
```

CVMDL is a `ClassificationPartitionedModel` classifier. It contains the property `Trained`, which is a 1-by-1 cell array holding a `CompactClassificationNaiveBayes` classifier that the software trained using the training set.

Determine how well the algorithm generalizes by estimating the test sample minimum cost loss.

```
L = loss(CMdl,XTest,YTest)
```

```
L =
    0.0476
```

The test sample average classification cost is approximately 0.05.

You might improve the classification error by specifying better predictor distributions when you train the classifier.

## Determine the Test Sample Classification Error of Naive Bayes Classifiers

Load Fisher's iris data set.

```
load fisheriris
X = meas;      % Predictors
Y = species;  % Response
rng(1); % For reproducibility
```

Train a naive Bayes classifier. Specify a 15% holdout sample for testing. It is good practice to specify the class order. Assume that each predictor is conditionally normally distributed given its label.

```
CVMDL = fitcnb(X,Y,'ClassNames',{'setosa','versicolor','virginica'},...
    'Holdout',0.15);
CMDL = CVMDL.Trained{1}; % Extract the trained, compact classifier
testInds = test(CVMDL.Partition); % Extract the test indices
XTest = X(testInds,:);
YTest = Y(testInds);
```

CVMDL is a `ClassificationPartitionedModel` classifier. It contains the property `Trained`, which is a 1-by-1 cell array holding a `CompactClassificationNaiveBayes` classifier that the software trained using the training set.

Determine how well the algorithm generalizes by estimating the test sample classification error.

```
L = loss(CMDL,XTest,YTest,'LossFun','classiferror')
```

```
L =
    0.0476
```

The classifier misclassified approximately 5% of the test sample observations.

## Definitions

### Classification Loss

Classification loss functions measure the predictive inaccuracy of classification models. When comparing the same type of loss among many models, lower loss indicates a better predictive model.

Suppose that:

- $L$  is the weighted average classification loss.
- $n$  is the sample size.
- For binary classification:
  - $y_j$  is the observed class label. The software codes it as  $-1$  or  $1$  indicating the negative or positive class, respectively.
  - $f(X_j)$  is the raw classification score for observation (row)  $j$  of the predictor data  $X$ .
  - $m_j = y_j f(X_j)$  is the classification score for classifying observation  $j$  into the class corresponding to  $y_j$ . Positive values of  $m_j$  indicate correct classification and do not contribute much to the average loss. Negative values of  $m_j$  indicate incorrect classification and contribute to the average loss.
- For algorithms that support multiclass classification (that is,  $K \geq 3$ ):
  - $y_j^*$  is a vector of  $K - 1$  zeros, and a 1 in the position corresponding to the true, observed class  $y_j$ . For example, if the true class of the second observation is the third class and  $K = 4$ , then  $y_2^* = [0 \ 0 \ 1 \ 0]'$ . The order of the classes corresponds to the order in the `ClassNames` property of the input model.
  - $f(X_j)$  is the length  $K$  vector of class scores for observation  $j$  of the predictor data  $X$ . The order of the scores corresponds to the order of the classes in the `ClassNames` property of the input model.
  - $m_j = y_j^{*'} f(X_j)$ . Therefore,  $m_j$  is the scalar classification score that the model predicts for the true, observed class.
- The weight for observation  $j$  is  $w_j$ . The software normalizes the observation weights so that they sum to the corresponding prior class probability. The software also normalizes the prior probabilities so they sum to 1. Therefore,

$$\sum_{j=1}^n w_j = 1.$$

The supported loss functions are:

- Binomial deviance, specified using 'LossFun', 'binodeviance'. Its equation is

$$L = \sum_{j=1}^n w_j \log \{1 + \exp[-2m_j]\}.$$

- Exponential loss, specified using 'LossFun', 'exponential'. Its equation is

$$L = \sum_{j=1}^n w_j \exp(-m_j).$$

- Classification error, specified using 'LossFun', 'classiferror'. It is the weighted fraction of misclassified observations, with equation

$$L = \sum_{j=1}^n w_j I\{\hat{y}_j \neq y_j\}.$$

$\hat{y}_j$  is the class label corresponding to the class with the maximal posterior probability.  $I\{x\}$  is the indicator function.

- Hinge loss, specified using 'LossFun', 'hinge'. Its equation is

$$L = \sum_{j=1}^n w_j \max\{0, 1 - m_j\}.$$

- Logit loss, specified using 'LossFun', 'logit'. Its equation is

$$L = \sum_{j=1}^n w_j \log(1 + \exp(-m_j)).$$

- Minimal cost, specified using 'LossFun', 'mincost'. The software computes the weighted minimal cost using this procedure for observations  $j = 1, \dots, n$ :

- 1 Estimate the 1-by- $K$  vector of expected classification costs for observation  $j$

$$\gamma_j = f(X_j)'C.$$



$f(X_j)$  is the column vector of class posterior probabilities for binary and multiclass classification.  $C$  is the cost matrix the input model stores in the property `Cost`.

- 2 For observation  $j$ , predict the class label corresponding to the minimum, expected classification cost:

$$\hat{y}_j = \min_{j=1, \dots, K} (\gamma_j).$$

- 3 Using  $C$ , identify the cost incurred ( $c_j$ ) for making the prediction.

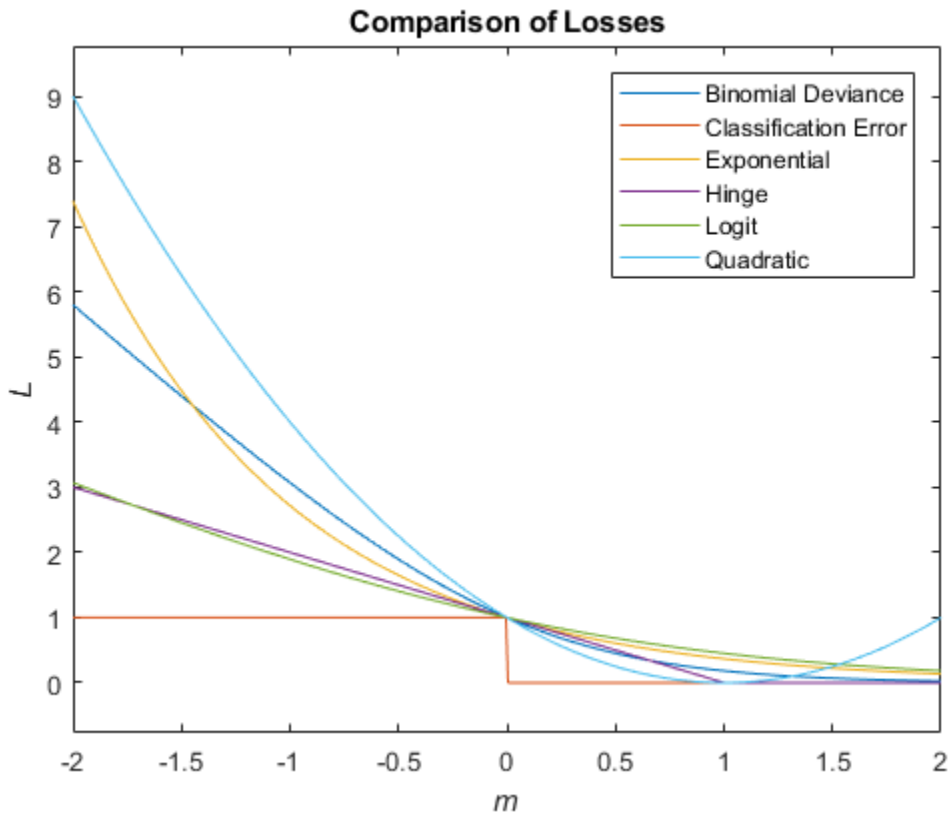
The weighted, average, minimum cost loss is

$$L = \sum_{j=1}^n w_j c_j.$$

- Quadratic loss, specified using `'LossFun', 'quadratic'`. Its equation is

$$L = \sum_{j=1}^n w_j (1 - m_j)^2.$$

This figure compares some of the loss functions for one observation over  $m$  (some functions are normalized to pass through  $[0,1]$ ).



## Misclassification Cost

A misclassification cost is the relative severity of a classifier labeling an observation into the wrong class.

There are two types of misclassification costs: true and expected. Let  $K$  be the number of classes.

- True misclassification cost — A  $K$ -by- $K$  matrix, where element  $(i, j)$  indicates the misclassification cost of predicting an observation into class  $j$  if its true class is  $i$ . The software stores the misclassification cost in the property `Mdl.Cost`, and used in computations. By default, `Mdl.Cost(i, j) = 1` if  $i \neq j$ , and `Mdl.Cost(i, j) = 0` if  $i = j$ .

=  $j$ . In other words, the cost is 0 for correct classification, and 1 for any incorrect classification.

- Expected misclassification cost — A  $K$ -dimensional vector, where element  $k$  is the weighted average misclassification cost of classifying an observation into class  $k$ , weighted by the class posterior probabilities. In other words,

$$c_k = \sum_{j=1}^K \hat{P}(Y = j | x_1, \dots, x_P) \text{Cost}_{jk}.$$

the software classifies observations to the class corresponding with the lowest expected misclassification cost.

## Posterior Probability

The posterior probability is the probability that an observation belongs in a particular class, given the data.

For naive Bayes, the posterior probability that a classification is  $k$  for a given observation  $(x_1, \dots, x_P)$  is

$$\hat{P}(Y = k | x_1, \dots, x_P) = \frac{P(X_1, \dots, X_P | y = k) \pi(Y = k)}{P(X_1, \dots, X_P)},$$

where:

- $P(X_1, \dots, X_P | y = k)$  is the conditional joint density of the predictors given they are in class  $k$ . `Mdl.DistributionNames` stores the distribution names of the predictors.
- $\pi(Y = k)$  is the class prior probability distribution. `Mdl.Prior` stores the prior distribution.
- $P(X_1, \dots, X_P)$  is the joint density of the predictors. The classes are discrete, so

$$P(X_1, \dots, X_P) = \sum_{k=1}^K P(X_1, \dots, X_P | y = k) \pi(Y = k).$$

## Prior Probability

The prior probability is the believed relative frequency that observations from a class occur in the population for each class.

## References

- [1] Hastie, T., R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*, second edition. Springer, New York, 2008.

## Extended Capabilities

### Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays” (MATLAB).

### See Also

`ClassificationNaiveBayes` | `CompactClassificationNaiveBayes` | `fitcnb` | `predict` | `resubLoss`

### Topics

“Naive Bayes Classification” on page 21-2

## loss

**Class:** CompactClassificationSVM

Classification error for support vector machine classifiers

## Syntax

```
L = loss(SVMModel, TBL, ResponseVarName)
```

```
L = loss(SVMModel, TBL, Y)
```

```
L = loss(SVMModel, X, Y)
```

```
L = loss( ____, Name, Value)
```

## Description

`L = loss(SVMModel, TBL, ResponseVarName)` returns the classification error (see “Classification Loss” on page 32-3725), a scalar representing how well the trained support vector machine (SVM) classifier `SVMModel` classifies the predictor data in table `TBL` as compared to the true class labels in `TBL.ResponseVarName`.

`loss` normalizes the class probabilities in `TBL.ResponseVarName` to the prior class probabilities `fitcsvm` used for training, stored in the `Prior` property of `SVMModel`.

`L = loss(SVMModel, TBL, Y)` returns the classification error for the predictor data in table `TBL` and the true class labels in `Y`.

`loss` normalizes the class probabilities in `Y` to the prior class probabilities `fitcsvm` used for training, stored in the `Prior` property of `SVMModel`.

`L = loss(SVMModel, X, Y)` returns the classification error based on the predictor data in matrix `X` as compared to the true class labels in `Y`.

`L = loss( ____, Name, Value)` returns the classification error with additional options specified by one or more `Name, Value` pair arguments, using any of the previous syntaxes. For example, you can specify the loss function or classification weights.

## Input Arguments

### **SVModel** — SVM classification model

ClassificationSVM model object | CompactClassificationSVM model object

SVM classification model, specified as a ClassificationSVM model object or CompactClassificationSVM model object returned by `fitsvm` or `compact`, respectively.

### **TBL** — Sample data

table

Sample data, specified as a table. Each row of TBL corresponds to one observation, and each column corresponds to one predictor variable. Optionally, TBL can contain additional columns for the response variable and observation weights. TBL must contain all of the predictors used to train SVModel. Multi-column variables and cell arrays other than cell arrays of character vectors are not allowed.

If TBL contains the response variable used to train SVModel, then you do not need to specify ResponseVarName or Y.

If you trained SVModel using sample data contained in a table, then the input data for this method must also be in a table.

Data Types: table

### **ResponseVarName** — Response variable name

name of a variable in TBL

Response variable name, specified as the name of a variable in TBL.

You must specify ResponseVarName as a character vector. For example, if the response variable Y is stored as TBL.Y, then specify it as 'Y'. Otherwise, the software treats all columns of TBL, including Y, as predictors when training the model.

The response variable must be a categorical or character array, logical or numeric vector, or cell array of character vectors. If the response variable is a character array, then each element must correspond to one row of the array.

### **x** — Predictor data

numeric matrix

Predictor data, specified as a numeric matrix.

Each row of `X` corresponds to one observation (also known as an instance or example), and each column corresponds to one variable (also known as a feature). The variables making up the columns of `X` must be the same as the variables that trained the `SVMMModel` classifier.

The length of `Y` and the number of rows of `X` must be equal.

If you set `'Standardize', true` in `fitcsvm` to train `SVMMModel`, then the software standardizes the columns of `X` using the corresponding means in `SVMMModel.Mu` and standard deviations in `SVMMModel.Sigma`.

Data Types: `double` | `single`

#### **y** — Class labels

categorical array | character array | logical vector | vector of numeric values | cell array of character vectors

Class labels, specified as a categorical or character array, logical or numeric vector, or cell array of character vectors. `Y` must be the same as the data type of `SVMMModel.ClassNames`.

The length of `Y` must equal the number of rows of `TBL` or `X` must be equal.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

#### **LossFun** — Loss function

`'classiferror'` (default) | `'binodeviance'` | `'exponential'` | `'hinge'` | `'logit'` | `'mincost'` | `'quadratic'` | function handle

Loss function, specified as the comma-separated pair consisting of `'LossFun'` and a built-in, loss-function name or function handle.

- The following lists available loss functions. Specify one using its corresponding character vector.

Value	Description
'binodeviance'	Binomial deviance
'classiferror'	Classification error
'exponential'	Exponential
'hinge'	Hinge
'logit'	Logistic
'mincost'	Minimal expected misclassification cost (for classification scores that are posterior probabilities)
'quadratic'	Quadratic

'mincost' is appropriate for classification scores that are posterior probabilities. You can specify to use posterior probabilities as classification scores for SVM models by setting 'FitPosterior', true when you cross-validate the model using `fitcsvm`.

- Specify your own function using function handle notation.

Suppose that  $n$  be the number of observations in  $X$  and  $K$  be the number of distinct classes (`numel(SVMModel.ClassNames)`, `SVMModel` is the input model). Your function must have this signature

```
lossvalue = lossfun(C,S,W,Cost)
```

where:

- The output argument `lossvalue` is a scalar.
- You choose the function name (`lossfun`).
- $C$  is an  $n$ -by- $K$  logical matrix with rows indicating which class the corresponding observation belongs. The column order corresponds to the class order in `SVMModel.ClassNames`.

Construct  $C$  by setting  $C(p, q) = 1$  if observation  $p$  is in class  $q$ , for each row. Set all other elements of row  $p$  to 0.

- $S$  is an  $n$ -by- $K$  numeric matrix of classification scores. The column order corresponds to the class order in `SVMModel.ClassNames`.  $S$  is a matrix of classification scores, similar to the output of `predict`.



- $W$  is an  $n$ -by-1 numeric vector of observation weights. If you pass  $W$ , the software normalizes them to sum to 1.
- $Cost$  is a  $K$ -by- $K$  numeric matrix of misclassification costs. For example,  $Cost = ones(K) - eye(K)$  specifies a cost of 0 for correct classification, and 1 for misclassification.

Specify your function using `'LossFun', @lossfun`.

For more details on loss functions, see “Classification Loss” on page 32-3725.

Data Types: `char` | `function_handle`

### **Weights — Observation weights**

`ones(size(X,1),1)` (default) | numeric vector | name of a variable in TBL

Observation weights, specified as the comma-separated pair consisting of `'Weights'` and a numeric vector or the name of a variable in TBL. The software weighs the observations in each row of  $X$  or TBL with the corresponding weight in  $Weights$ .

If you specify  $Weights$  as a vector, then the size of  $Weights$  must be equal to the number of rows of  $X$  or TBL.

If you specify  $Weights$  as the name of a variable in TBL, you must do so as a character vector. For example, if the weights are stored as `TBL.W`, then specify it as `'W'`. Otherwise, the software treats all columns of TBL, including `TBL.W`, as predictors.

If you do not specify your own loss function, then the software normalizes  $Weights$  to sum up to the value of the prior probability in the respective class.

Data Types: `single` | `double`

## **Output Arguments**

### **L — Classification loss**

scalar

Classification loss, returned as a scalar.  $L$  is a generalization or resubstitution quality measure. Its interpretation depends on the loss function and weighting scheme, but, in general, better classifiers yield smaller loss values.

## Examples

### Determine the Test Sample Classification Error of SVM Classifiers

Load the ionosphere data set.

```
load ionosphere
rng(1); % For reproducibility
```

Train an SVM classifier. Specify a 15% holdout sample for testing. It is good practice to specify the class order and standardize the data.

```
CVSVMModel = fitcsvm(X,Y,'Holdout',0.15,'ClassNames',{'b','g'},...
    'Standardize',true);
CompactSVMModel = CVSVMModel.Trained{1}; % Extract the trained, compact classifier
testInds = test(CVSVMModel.Partition); % Extract the test indices
XTest = X(testInds,:);
YTest = Y(testInds,:);
```

CVSVMModel is a `ClassificationPartitionedModel` classifier. It contains the property `Trained`, which is a 1-by-1 cell array holding a `CompactClassificationSVM` classifier that the software trained using the training set.

Determine how well the algorithm generalizes by estimating the test sample classification error.

```
L = loss(CompactSVMModel,XTest,YTest)
```

```
L =
    0.0787
```

The SVM classifier misclassifies approximately 8% of the test sample radar returns.

### Determine the Test Sample Hinge Loss of SVM Classifiers

Load the ionosphere data set.

```
load ionosphere
rng(1); % For reproducibility
```

Train an SVM classifier. Specify a 15% holdout sample for testing. It is good practice to specify the class order and standardize the data.

```
CVSVMModel = fitcsvm(X,Y,'Holdout',0.15,'ClassNames',{'b','g'},...
    'Standardize',true);
CompactSVMModel = CVSVMModel.Trained{1}; % Extract the trained, compact classifier
testInds = test(CVSVMModel.Partition); % Extract the test indices
XTest = X(testInds,:);
YTest = Y(testInds,:);
```

CVSVMModel is a ClassificationPartitionedModel classifier. It contains the property Trained, which is a 1-by-1 cell array holding a CompactClassificationSVM classifier that the software trained using the training set.

Determine how well the algorithm generalizes by estimating the test sample hinge loss.

```
L = loss(CompactSVMModel,XTest,YTest,'LossFun','Hinge')
```

```
L =
```

```
    0.2998
```

The hinge loss is approximately 0.3. Classifiers with hinge losses close to 0 are desirable.

## Definitions

### Classification Loss

Classification loss functions measure the predictive inaccuracy of classification models. When comparing the same type of loss among many models, lower loss indicates a better predictive model.

Suppose that:

- $L$  is the weighted average classification loss.
- $n$  is the sample size.

- For binary classification:
  - $y_j$  is the observed class label. The software codes it as  $-1$  or  $1$  indicating the negative or positive class, respectively.
  - $f(X_j)$  is the raw classification score for observation (row)  $j$  of the predictor data  $X$ .
  - $m_j = y_j f(X_j)$  is the classification score for classifying observation  $j$  into the class corresponding to  $y_j$ . Positive values of  $m_j$  indicate correct classification and do not contribute much to the average loss. Negative values of  $m_j$  indicate incorrect classification and contribute to the average loss.
- For algorithms that support multiclass classification (that is,  $K \geq 3$ ):
  - $y_j^*$  is a vector of  $K - 1$  zeros, and a  $1$  in the position corresponding to the true, observed class  $y_j$ . For example, if the true class of the second observation is the third class and  $K = 4$ , then  $y_2^* = [0 \ 0 \ 1 \ 0]'$ . The order of the classes corresponds to the order in the `ClassNames` property of the input model.
  - $f(X_j)$  is the length  $K$  vector of class scores for observation  $j$  of the predictor data  $X$ . The order of the scores corresponds to the order of the classes in the `ClassNames` property of the input model.
  - $m_j = y_j^{*'} f(X_j)$ . Therefore,  $m_j$  is the scalar classification score that the model predicts for the true, observed class.
- The weight for observation  $j$  is  $w_j$ . The software normalizes the observation weights so that they sum to the corresponding prior class probability. The software also normalizes the prior probabilities so they sum to 1. Therefore,

$$\sum_{j=1}^n w_j = 1.$$

The supported loss functions are:

- Binomial deviance, specified using `'LossFun', 'binodeviance'`. Its equation is

$$L = \sum_{j=1}^n w_j \log \{1 + \exp[-2m_j]\}.$$

- Exponential loss, specified using `'LossFun', 'exponential'`. Its equation is

$$L = \sum_{j=1}^n w_j \exp(-m_j).$$

- Classification error, specified using 'LossFun', 'classiferror'. It is the weighted fraction of misclassified observations, with equation

$$L = \sum_{j=1}^n w_j I\{\hat{y}_j \neq y_j\}.$$

$\hat{y}_j$  is the class label corresponding to the class with the maximal posterior probability.  $I\{x\}$  is the indicator function.

- Hinge loss, specified using 'LossFun', 'hinge'. Its equation is

$$L = \sum_{j=1}^n w_j \max\{0, 1 - m_j\}.$$

- Logit loss, specified using 'LossFun', 'logit'. Its equation is

$$L = \sum_{j=1}^n w_j \log(1 + \exp(-m_j)).$$

- Minimal cost, specified using 'LossFun', 'mincost'. The software computes the weighted minimal cost using this procedure for observations  $j = 1, \dots, n$ :

- 1 Estimate the 1-by- $K$  vector of expected classification costs for observation  $j$

$$\gamma_j = f(X_j)'C.$$

$f(X_j)$  is the column vector of class posterior probabilities for binary and multiclass classification.  $C$  is the cost matrix the input model stores in the property `Cost`.

- 2 For observation  $j$ , predict the class label corresponding to the minimum, expected classification cost:

$$\hat{y}_j = \min_{j=1, \dots, K} (\gamma_j).$$

- 3 Using  $C$ , identify the cost incurred ( $c_j$ ) for making the prediction.

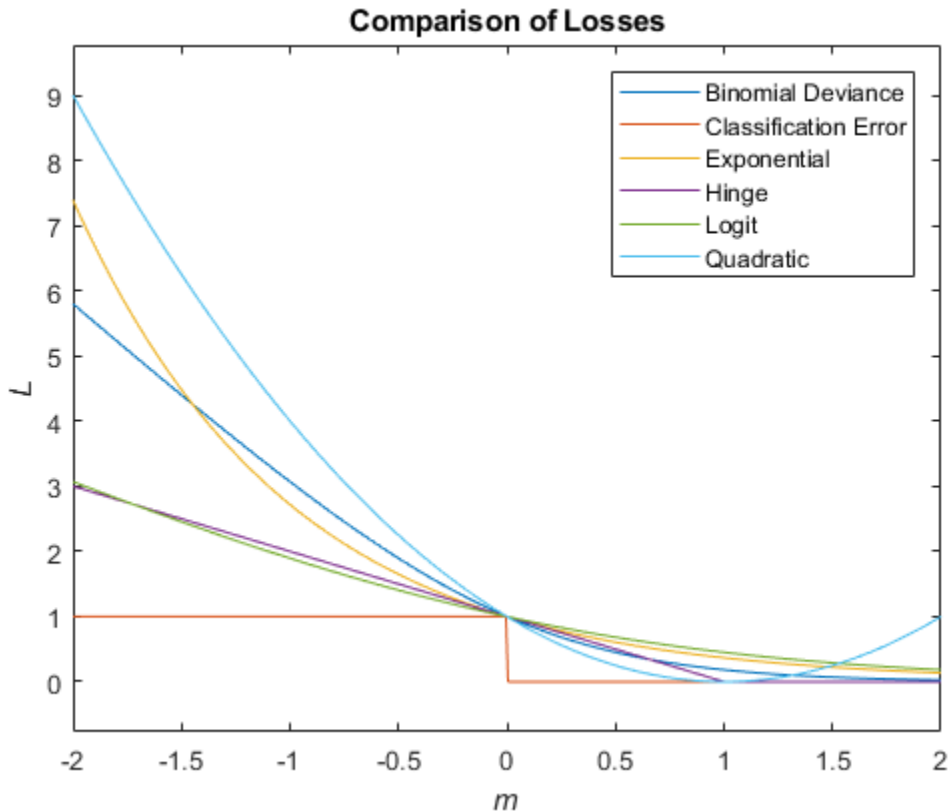
The weighted, average, minimum cost loss is

$$L = \sum_{j=1}^n w_j c_j.$$

- Quadratic loss, specified using 'LossFun', 'quadratic'. Its equation is

$$L = \sum_{j=1}^n w_j (1 - m_j)^2.$$

This figure compares some of the loss functions for one observation over  $m$  (some functions are normalized to pass through  $[0,1]$ ).



## Score

The SVM classification score for classifying observation  $x$  is the signed distance from  $x$  to the decision boundary ranging from  $-\infty$  to  $+\infty$ . A positive score for a class indicates that  $x$  is predicted to be in that class. A negative score indicates otherwise.

The score for predicting  $x$  into the positive class, also the numerical, predicted response for  $x$ ,  $f(x)$ , is the trained SVM classification function

$$f(x) = \sum_{j=1}^n \alpha_j y_j G(x_j, x) + b,$$

where  $(\alpha_1, \dots, \alpha_n, b)$  are the estimated SVM parameters,  $G(x_j, x)$  is the dot product in the predictor space between  $x$  and the support vectors, and the sum includes the training set observations. The score for predicting  $x$  into the negative class is  $-f(x)$ .

If  $G(x_j, x) = x_j'x$  (the linear kernel), then the score function reduces to

$$f(x) = (x / s)' \beta + b.$$

$s$  is the kernel scale and  $\beta$  is the vector of fitted linear coefficients.

## References

- [1] Hastie, T., R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*, second edition. Springer, New York, 2008.

## Extended Capabilities

### Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays” (MATLAB).

### See Also

`ClassificationSVM` | `CompactClassificationSVM` | `edge` | `fitcsvm` | `predict`

## loss

**Class:** CompactClassificationTree

Classification error

## Syntax

```
L = loss(tree, TBL, ResponseVarName)
```

```
L = loss(tree, TBL, Y)
```

```
L = loss(tree, X, Y)
```

```
L = loss(____, Name, Value)
```

```
[L, se, NLeaf, bestlevel] = loss(____)
```

## Description

`L = loss(tree, TBL, ResponseVarName)` returns a scalar representing how well `tree` classifies the data in `TBL`, when `TBL.ResponseVarName` contains the true classifications.

When computing the loss, `loss` normalizes the class probabilities in `Y` to the class probabilities used for training, stored in the `Prior` property of `tree`.

`L = loss(tree, TBL, Y)` returns a scalar representing how well `tree` classifies the data in `TBL`, when `Y` contains the true classifications.

`L = loss(tree, X, Y)` returns a scalar representing how well `tree` classifies the data in `X`, when `Y` contains the true classifications.

`L = loss(____, Name, Value)` returns the loss with additional options specified by one or more `Name, Value` pair arguments, using any of the previous syntaxes. For example, you can specify the loss function or observation weights.

`[L, se, NLeaf, bestlevel] = loss(____)` also returns the vector of standard errors of the classification errors (`se`), the vector of numbers of leaf nodes in the trees of the



pruning sequence (`NLeaf`), and the best pruning level as defined in the `TreeSize` name-value pair (`bestlevel`).

---

**Note** `loss` returns `se` and further outputs only when the `LossFun` name-value pair is the default `'classiferror'`.

---

## Input Arguments

### **tree** — Trained classification tree

`ClassificationTree` model object | `CompactClassificationTree` model object

Trained classification tree, specified as a `ClassificationTree` or `CompactClassificationTree` model object. That is, `tree` is a trained classification model returned by `fitctree` or `compact`.

### **TBL** — Sample data

table

Sample data, specified as a table. Each row of `TBL` corresponds to one observation, and each column corresponds to one predictor variable. Optionally, `TBL` can contain additional columns for the response variable and observation weights. `TBL` must contain all the predictors used to train `tree`. Multi-column variables and cell arrays other than cell arrays of character vectors are not allowed.

If `TBL` contains the response variable used to train `tree`, then you do not need to specify `ResponseVarName` or `Y`.

If you train `tree` using sample data contained in a table, then the input data for this method must also be in a table.

Data Types: table

### **x** — Data to classify

numeric matrix

Data to classify, specified as a numeric matrix. Each row of `X` represents one observation, and each column represents one predictor. `X` must have the same number of columns as the data used to train `tree`. `X` must have the same number of rows as the number of elements in `Y`.

Data Types: `single` | `double`

**ResponseVarName** — Response variable name

name of a variable in TBL

Response variable name, specified as the name of a variable in TBL. If TBL contains the response variable used to train `tree`, then you do not need to specify `ResponseVarName`.

If you specify `ResponseVarName`, then you must do so as a character vector. For example, if the response variable is stored as `TBL.Response`, then specify it as `'Response'`. Otherwise, the software treats all columns of TBL, including `TBL.ResponseVarName`, as predictors.

The response variable must be a categorical or character array, logical or numeric vector, or cell array of character vectors. If the response variable is a character array, then each element must correspond to one row of the array.

**Y** — Class labels

categorical array | character array | logical vector | vector of numeric values | cell array of character vectors

Class labels, specified as a categorical or character array, a logical or numeric vector, or a cell array of character vectors. `Y` must be of the same type as the classification used to train `tree`, and its number of elements must equal the number of rows of `X`.

Data Types: `single` | `double` | `categorical` | `char` | `logical` | `cell`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

**LossFun** — Loss function

`'mincost'` (default) | `'binodeviance'` | `'classiferror'` | `'exponential'` | `'hinge'` | `'logit'` | `'quadratic'` | function handle

Loss function, specified as the comma-separated pair consisting of `'LossFun'` and a built-in, loss-function name or function handle.

- The following lists available loss functions. Specify one using its corresponding character vector.

Value	Description
'binodeviance'	Binomial deviance
'classiferror'	Classification error
'exponential'	Exponential
'hinge'	Hinge
'logit'	Logistic
'mincost'	Minimal expected misclassification cost (for classification scores that are posterior probabilities)
'quadratic'	Quadratic

'mincost' is appropriate for classification scores that are posterior probabilities. Classification trees return posterior probabilities as classification scores by default (see `predict`).

- Specify your own function using function handle notation.

Suppose that  $n$  be the number of observations in  $X$  and  $K$  be the number of distinct classes (`numel(tree.ClassNames)`). Your function must have this signature

```
lossvalue = lossfun(C,S,W,Cost)
```

where:

- The output argument `lossvalue` is a scalar.
- You choose the function name (`lossfun`).
- $C$  is an  $n$ -by- $K$  logical matrix with rows indicating which class the corresponding observation belongs. The column order corresponds to the class order in `tree.ClassNames`.

Construct  $C$  by setting  $C(p, q) = 1$  if observation  $p$  is in class  $q$ , for each row. Set all other elements of row  $p$  to 0.

- $S$  is an  $n$ -by- $K$  numeric matrix of classification scores. The column order corresponds to the class order in `tree.ClassNames`.  $S$  is a matrix of classification scores, similar to the output of `predict`.

- `W` is an  $n$ -by-1 numeric vector of observation weights. If you pass `W`, the software normalizes them to sum to 1.
- `Cost` is a  $K$ -by- $K$  numeric matrix of misclassification costs. For example, `Cost = ones(K) - eye(K)` specifies a cost of 0 for correct classification, and 1 for misclassification.

Specify your function using `'LossFun', @lossfun`.

For more details on loss functions, see “Classification Loss” on page 32-3740.

### **Weights — Observation weights**

`ones(size(X,1))` (default) | name of a variable in TBL | numeric vector of positive values

Observation weights, specified as the comma-separated pair consisting of `'Weights'` and a numeric vector of positive values or the name of a variable in TBL.

If you specify `Weights` as a numeric vector, then the size of `Weights` must be equal to the number of rows in `X` or TBL.

If you specify `Weights` as the name of a variable in TBL, you must do so as a character vectors. For example, if the weights are stored as `TBL.W`, then specify it as `'W'`. Otherwise, the software treats all columns of TBL, including `TBL.W`, as predictors.

`loss` normalizes the weights so that observation weights in each class sum to the prior probability of that class. When you supply `Weights`, `loss` computes weighted classification loss.

Data Types: `single` | `double`

Name, Value arguments associated with pruning subtrees:

### **Subtrees — Pruning level**

0 (default) | vector of nonnegative integers | `'all'`

Pruning level, specified as the comma-separated pair consisting of `'Subtrees'` and a vector of nonnegative integers in ascending order or `'all'`.

If you specify a vector, then all elements must be at least 0 and at most `max(tree.PruneList)`. 0 indicates the full, unpruned tree and `max(tree.PruneList)` indicates the completely pruned tree (i.e., just the root node).

If you specify 'all', then `CompactClassificationTree.loss` operates on all subtrees (i.e., the entire pruning sequence). This specification is equivalent to using `0:max(tree.PruneList)`.

`CompactClassificationTree.loss` prunes tree to each level indicated in `Subtrees`, and then estimates the corresponding output arguments. The size of `Subtrees` determines the size of some output arguments.

To invoke `Subtrees`, the properties `PruneList` and `PruneAlpha` of `tree` must be nonempty. In other words, grow tree by setting 'Prune', 'on', or by pruning tree using `prune`.

Example: 'Subtrees', 'all'

### **TreeSize — Tree size**

'se' (default) | 'min'

Tree size, specified as the comma-separated pair consisting of 'TreeSize' and one of the following character vectors:

- 'se' — `loss` returns the highest pruning level with loss within one standard deviation of the minimum ( $L+se$ , where  $L$  and  $se$  relate to the smallest value in `Subtrees`).
- 'min' — `loss` returns the element of `Subtrees` with smallest loss, usually the smallest element of `Subtrees`.

## Output Arguments

### **L — Classification loss**

vector of scalar values

Classification loss on page 32-3740, returned as a vector the length of `Subtrees`. The meaning of the error depends on the values in `Weights` and `LossFun`.

### **se — Standard error of loss**

vector of scalar values

Standard error of loss, returned as a vector the length of `Subtrees`.

**NLeaf** — Number of leaf nodes

vector of integer values

Number of leaves (terminal nodes) in the pruned subtrees, returned as a vector the length of `Subtrees`.

**bestlevel** — Best pruning level

scalar value

Best pruning level as defined in the `TreeSize` name-value pair, returned as a scalar whose value depends on `TreeSize`:

- `TreeSize = 'se'` — `loss` returns the highest pruning level with loss within one standard deviation of the minimum ( $L+se$ , where  $L$  and  $se$  relate to the smallest value in `Subtrees`).
- `TreeSize = 'min'` — `loss` returns the element of `Subtrees` with smallest loss, usually the smallest element of `Subtrees`.

By default, `bestlevel` is the pruning level that gives loss within one standard deviation of minimal loss.

## Examples

**Compute the In-sample Classification Error**

Compute the resubstituted classification error for the `ionosphere` data set.

```
load ionosphere
tree = fitctree(X,Y);
L = loss(tree,X,Y)
```

```
L =
```

```
0.0114
```

## Examine the Classification Error for Each Subtree

Unpruned decision trees tend to overfit. One way to balance model complexity and out-of-sample performance is to prune a tree (or restrict its growth) so that in-sample and out-of-sample performance are satisfactory.

Load Fisher's iris data set. Partition the data into training (50%) and validation (50%) sets.

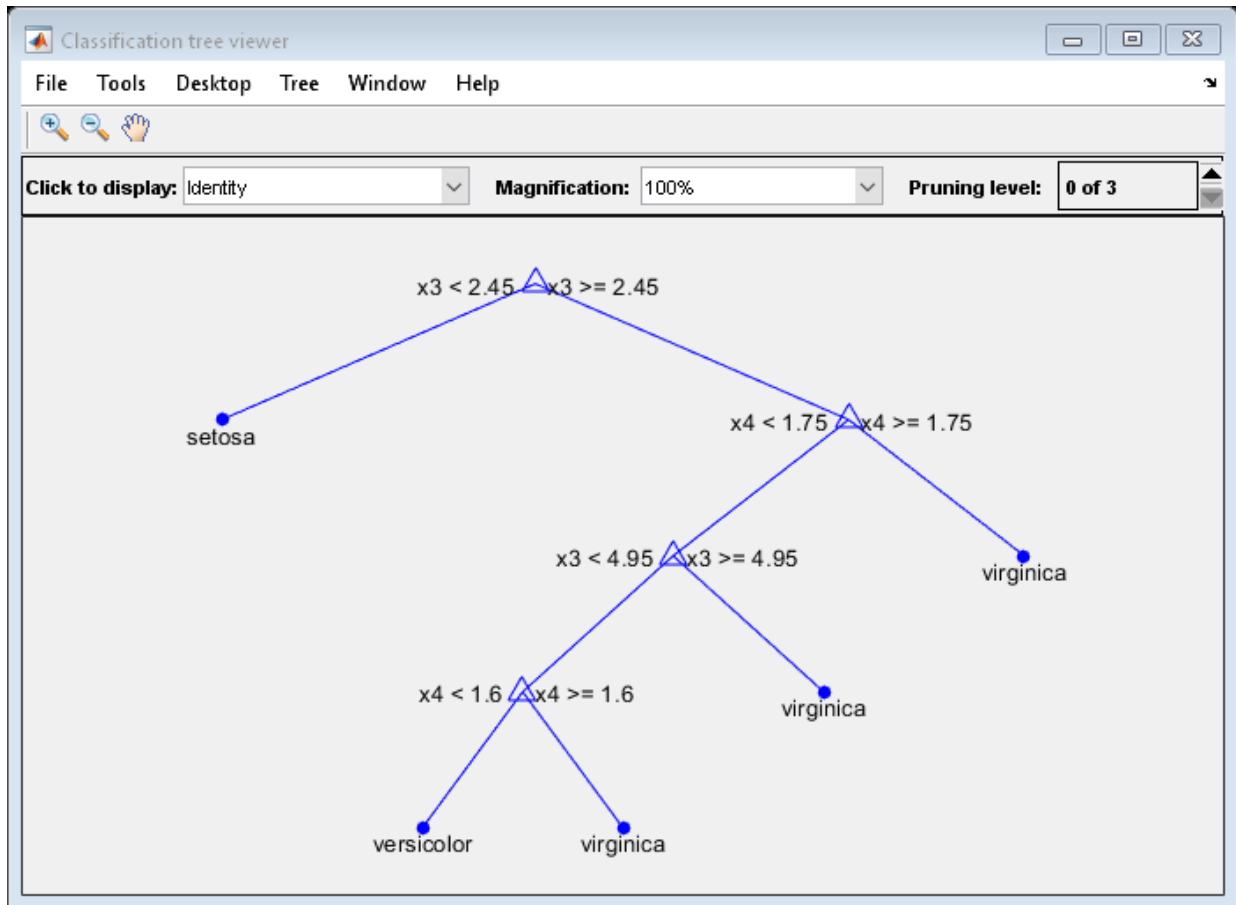
```
load fisheriris
n = size(meas,1);
rng(1) % For reproducibility
idxTrn = false(n,1);
idxTrn(randsample(n,round(0.5*n))) = true; % Training set logical indices
idxVal = idxTrn == false; % Validation set logical indices
```

Grow a classification tree using the training set.

```
Mdl = fitctree(meas(idxTrn,:), species(idxTrn));
```

View the classification tree.

```
view(Mdl, 'Mode', 'graph');
```



The classification tree has four pruning levels. Level 0 is the full, unpruned tree (as displayed). Level 3 is just the root node (i.e., no splits).

Examine the training sample classification error for each subtree (or pruning level) excluding the highest level.

```
m = max(Mdl.PruneList) - 1;
trnLoss = resubLoss(Mdl, 'SubTrees', 0:m)
```

```
trnLoss =
```



```
0.0267
0.0533
0.3067
```

- The full, unpruned tree misclassifies about 2.7% of the training observations.
- The tree pruned to level 1 misclassifies about 5.3% of the training observations.
- The tree pruned to level 2 (i.e., a stump) misclassifies about 30.6% of the training observations.

Examine the validation sample classification error at each level excluding the highest level.

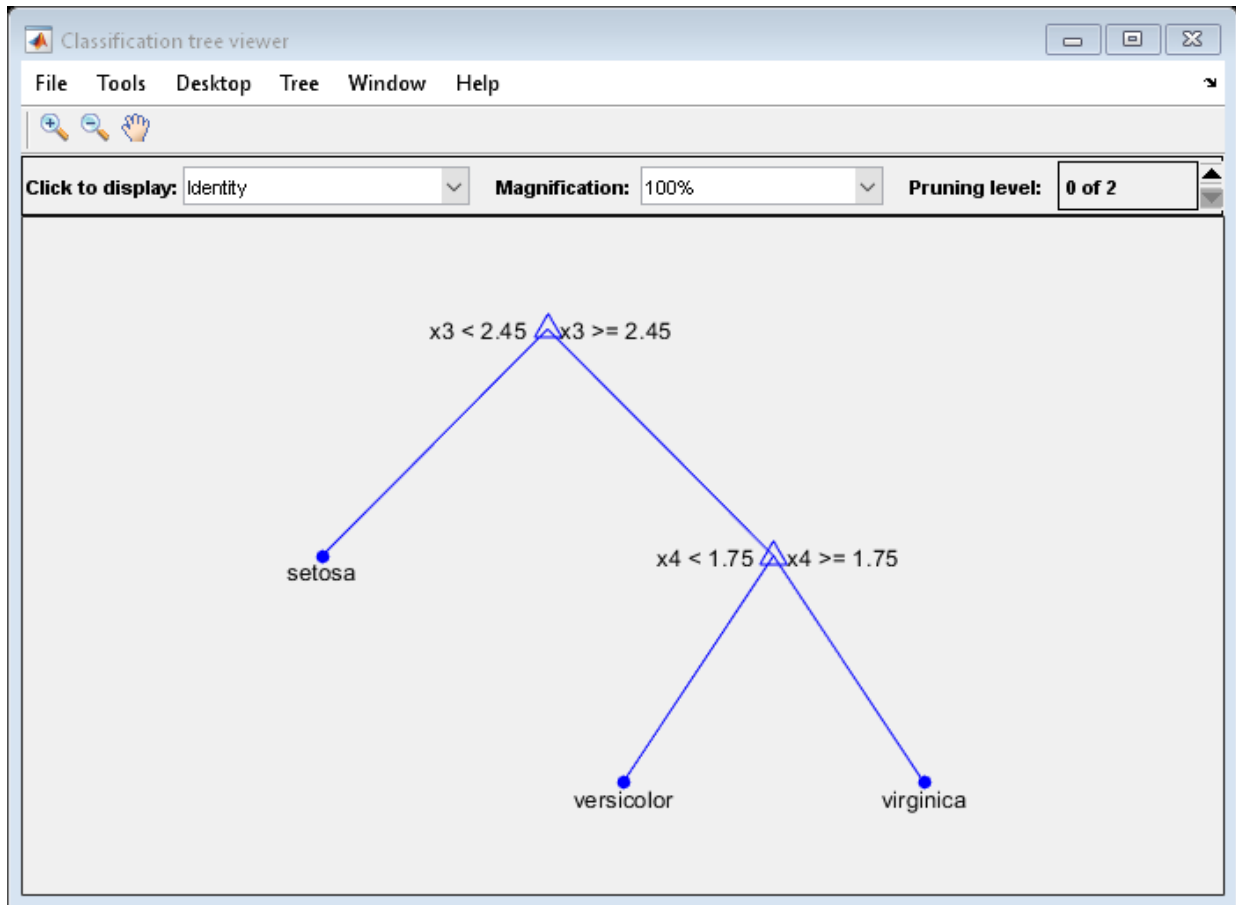
```
valLoss = loss(Mdl, meas(idxVal, :), species(idxVal), 'SubTrees', 0:m)
```

```
valLoss =
    0.0369
    0.0237
    0.3067
```

- The full, unpruned tree misclassifies about 3.7% of the validation observations.
- The tree pruned to level 1 misclassifies about 2.4% of the validation observations.
- The tree pruned to level 2 (i.e., a stump) misclassifies about 30.7% of the validation observations.

To balance model complexity and out-of-sample performance, consider pruning `Mdl` to level 1.

```
pruneMdl = prune(Mdl, 'Level', 1);
view(pruneMdl, 'Mode', 'graph')
```



## Definitions

### Classification Loss

Classification loss functions measure the predictive inaccuracy of classification models. When comparing the same type of loss among many models, lower loss indicates a better predictive model.

Suppose that:

- $L$  is the weighted average classification loss.
- $n$  is the sample size.
- For binary classification:
  - $y_j$  is the observed class label. The software codes it as  $-1$  or  $1$  indicating the negative or positive class, respectively.
  - $f(X_j)$  is the raw classification score for observation (row)  $j$  of the predictor data  $X$ .
  - $m_j = y_j f(X_j)$  is the classification score for classifying observation  $j$  into the class corresponding to  $y_j$ . Positive values of  $m_j$  indicate correct classification and do not contribute much to the average loss. Negative values of  $m_j$  indicate incorrect classification and contribute to the average loss.
- For algorithms that support multiclass classification (that is,  $K \geq 3$ ):
  - $y_j^*$  is a vector of  $K - 1$  zeros, and a 1 in the position corresponding to the true, observed class  $y_j$ . For example, if the true class of the second observation is the third class and  $K = 4$ , then  $y_2^* = [0 \ 0 \ 1 \ 0]'$ . The order of the classes corresponds to the order in the `ClassNames` property of the input model.
  - $f(X_j)$  is the length  $K$  vector of class scores for observation  $j$  of the predictor data  $X$ . The order of the scores corresponds to the order of the classes in the `ClassNames` property of the input model.
  - $m_j = y_j^{*'} f(X_j)$ . Therefore,  $m_j$  is the scalar classification score that the model predicts for the true, observed class.
- The weight for observation  $j$  is  $w_j$ . The software normalizes the observation weights so that they sum to the corresponding prior class probability. The software also normalizes the prior probabilities so they sum to 1. Therefore,

$$\sum_{j=1}^n w_j = 1.$$

The supported loss functions are:

- Binomial deviance, specified using `'LossFun', 'binodeviance'`. Its equation is

$$L = \sum_{j=1}^n w_j \log \{1 + \exp[-2m_j]\}.$$

- Exponential loss, specified using `'LossFun', 'exponential'`. Its equation is

$$L = \sum_{j=1}^n w_j \exp(-m_j).$$

- Classification error, specified using 'LossFun', 'classiferror'. It is the weighted fraction of misclassified observations, with equation

$$L = \sum_{j=1}^n w_j I\{\hat{y}_j \neq y_j\}.$$

$\hat{y}_j$  is the class label corresponding to the class with the maximal posterior probability.  $I\{x\}$  is the indicator function.

- Hinge loss, specified using 'LossFun', 'hinge'. Its equation is

$$L = \sum_{j=1}^n w_j \max\{0, 1 - m_j\}.$$

- Logit loss, specified using 'LossFun', 'logit'. Its equation is

$$L = \sum_{j=1}^n w_j \log(1 + \exp(-m_j)).$$

- Minimal cost, specified using 'LossFun', 'mincost'. The software computes the weighted minimal cost using this procedure for observations  $j = 1, \dots, n$ :

- 1 Estimate the 1-by- $K$  vector of expected classification costs for observation  $j$

$$\gamma_j = f(X_j)'C.$$

$f(X_j)$  is the column vector of class posterior probabilities for binary and multiclass classification.  $C$  is the cost matrix the input model stores in the property `Cost`.

- 2 For observation  $j$ , predict the class label corresponding to the minimum, expected classification cost:

$$\hat{y}_j = \min_{j=1, \dots, K}(\gamma_j).$$

- 3 Using  $C$ , identify the cost incurred ( $c_j$ ) for making the prediction.

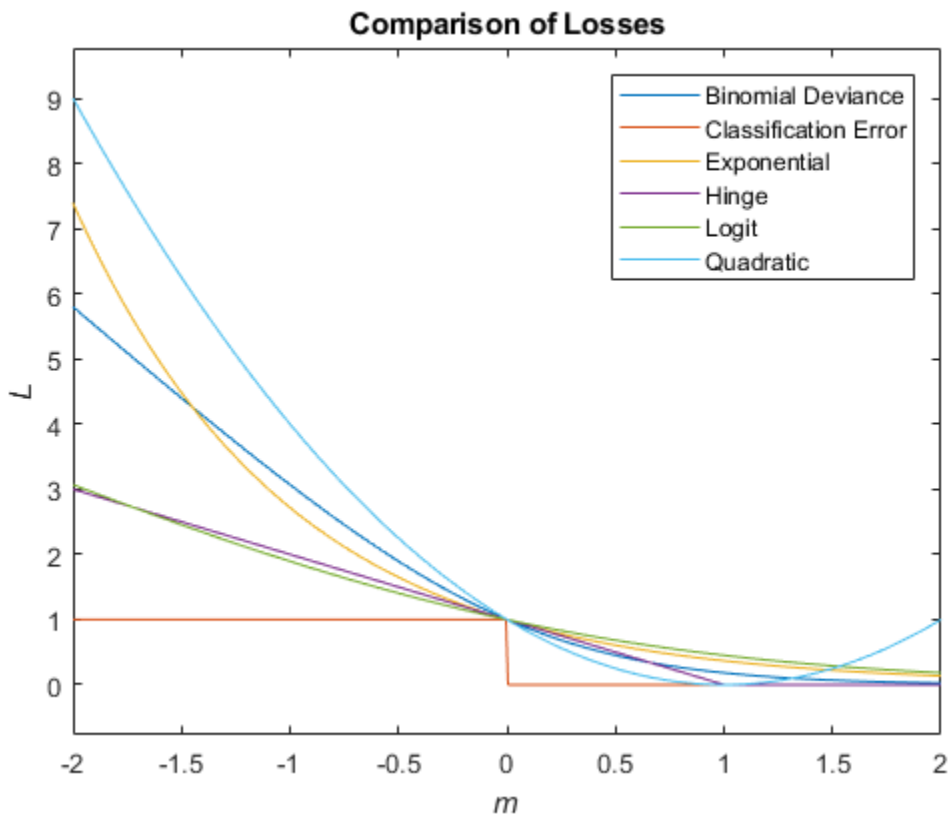
The weighted, average, minimum cost loss is

$$L = \sum_{j=1}^n w_j c_j.$$

- Quadratic loss, specified using 'LossFun', 'quadratic'. Its equation is

$$L = \sum_{j=1}^n w_j (1 - m_j)^2.$$

This figure compares some of the loss functions for one observation over  $m$  (some functions are normalized to pass through  $[0,1]$ ).



## True Misclassification Cost

There are two costs associated with classification: the true misclassification cost per class, and the expected misclassification cost per observation.

You can set the true misclassification cost per class in the `Cost` name-value pair when you create the classifier using the `fitctree` method.  $\text{Cost}(i, j)$  is the cost of classifying an observation into class  $j$  if its true class is  $i$ . By default,  $\text{Cost}(i, j) = 1$  if  $i \neq j$ , and  $\text{Cost}(i, j) = 0$  if  $i = j$ . In other words, the cost is 0 for correct classification, and 1 for incorrect classification.

## Expected Misclassification Cost

There are two costs associated with classification: the true misclassification cost per class, and the expected misclassification cost per observation.

Suppose you have `Nobs` observations that you want to classify with a trained classifier. Suppose you have  $K$  classes. You place the observations into a matrix `Xnew` with one observation per row.

The expected cost matrix `CE` has size `Nobs`-by- $K$ . Each row of `CE` contains the expected (average) cost of classifying the observation into each of the  $K$  classes.  $\text{CE}(n, k)$  is

$$\sum_{i=1}^K \hat{P}(i | X_{\text{new}}(n)) C(k | i),$$

where

- $K$  is the number of classes.
- $\hat{P}(i | X_{\text{new}}(n))$  is the posterior probability of class  $i$  for observation  $X_{\text{new}}(n)$ .
- $C(k | i)$  is the true misclassification cost of classifying an observation as  $k$  when its true class is  $i$ .

## Score (tree)

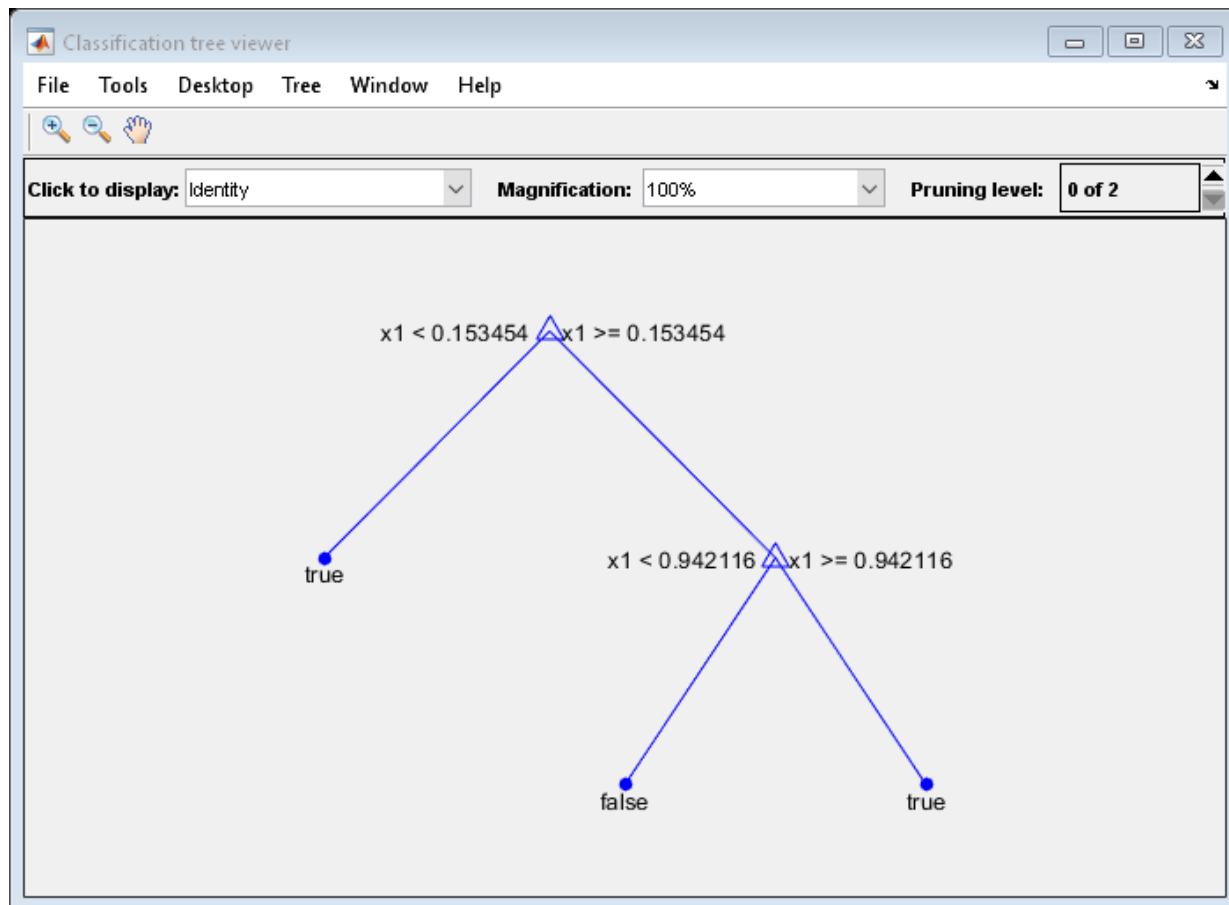
For trees, the score of a classification of a leaf node is the posterior probability of the classification at that node. The posterior probability of the classification at a node is the

number of training sequences that lead to that node with the classification, divided by the number of training sequences that lead to that node.

For example, consider classifying a predictor  $X$  as `true` when  $X < 0.15$  or  $X > 0.95$ , and  $X$  is `false` otherwise.

Generate 100 random points and classify them:

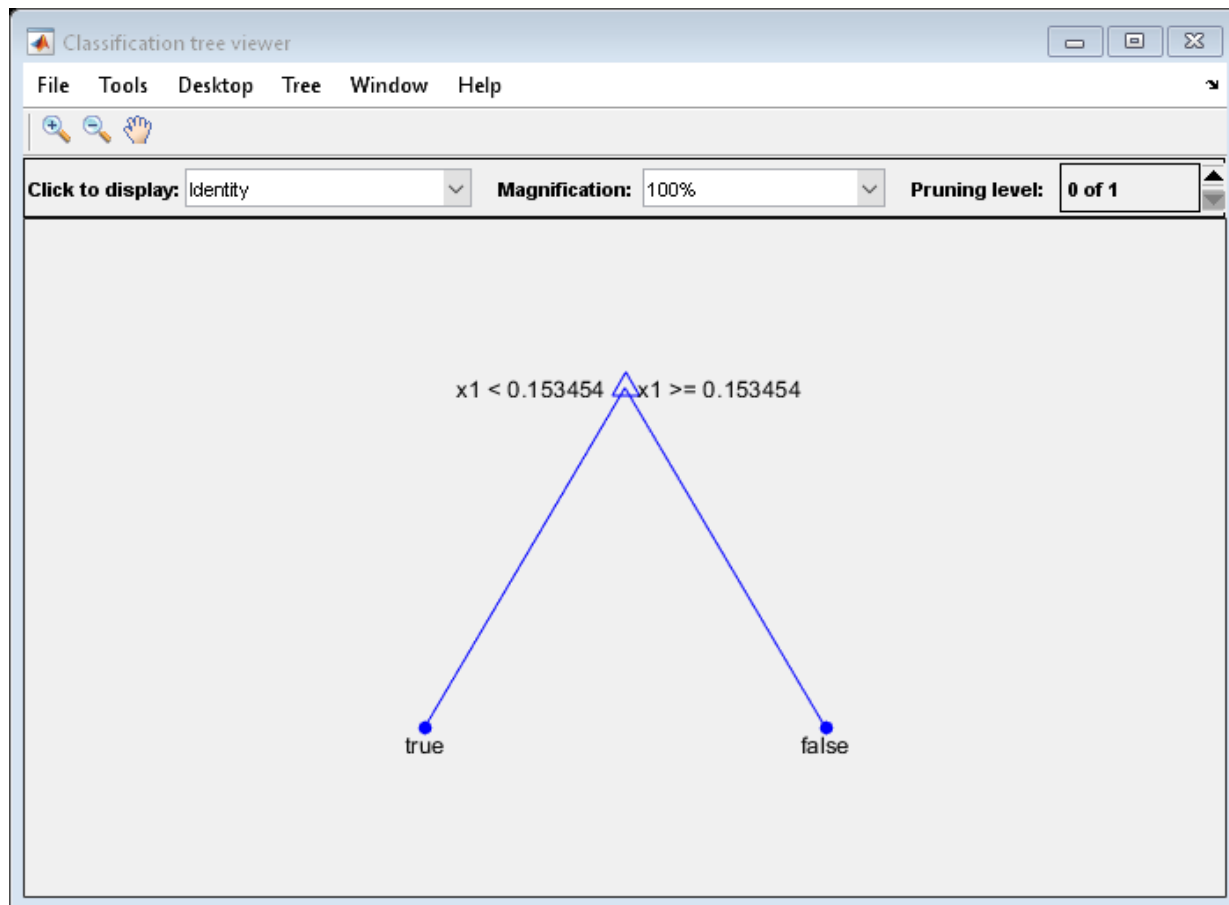
```
rng(0, 'twister') % for reproducibility
X = rand(100,1);
Y = (abs(X - .55) > .4);
tree = fitctree(X,Y);
view(tree, 'Mode', 'Graph')
```



Prune the tree:

```
tree1 = prune(tree, 'Level', 1);  
view(tree1, 'Mode', 'Graph')
```





The pruned tree correctly classifies observations that are less than 0.15 as `true`. It also correctly classifies observations from .15 to .94 as `false`. However, it incorrectly classifies observations that are greater than .94 as `false`. Therefore, the score for observations that are greater than .15 should be about  $.05/.85=.06$  for `true`, and about  $.8/.85=.94$  for `false`.

Compute the prediction scores for the first 10 rows of `X`:

```
[~,score] = predict(tree1,X(1:10));
[score X(1:10,:)]
```

```
ans =  
  
    0.9059    0.0941    0.8147  
    0.9059    0.0941    0.9058  
         0     1.0000    0.1270  
    0.9059    0.0941    0.9134  
    0.9059    0.0941    0.6324  
         0     1.0000    0.0975  
    0.9059    0.0941    0.2785  
    0.9059    0.0941    0.5469  
    0.9059    0.0941    0.9575  
    0.9059    0.0941    0.9649
```

Indeed, every value of  $X$  (the right-most column) that is less than 0.15 has associated scores (the left and center columns) of 0 and 1, while the other values of  $X$  have associated scores of 0.91 and 0.09. The difference (score 0.09 instead of the expected .06) is due to a statistical fluctuation: there are 8 observations in  $X$  in the range  $(.95, 1)$  instead of the expected 5 observations.

## Extended Capabilities

### Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function supports tall arrays for out-of-memory data with the limitation:

- Only one output is supported.

For more information, see “Tall Arrays” (MATLAB).

### See Also

`edge` | `fitctree` | `margin` | `predict`

# loss

**Class:** CompactRegressionEnsemble

Regression error

## Syntax

```
L = loss(ens, tbl, ResponseVarName)
L = loss(ens, tbl, Y)
L = loss(ens, X, Y)
L = loss( ____, Name, Value)
```

## Description

`L = loss(ens, tbl, ResponseVarName)` returns the mean squared error between the predictions of `ens` to the data in `tbl`, compared to the true responses `tbl.ResponseVarName`.

`L = loss(ens, tbl, Y)` returns the mean squared error between the predictions of `ens` to the data in `tbl`, compared to the true responses `Y`.

`L = loss(ens, X, Y)` returns the mean squared error between the predictions of `ens` to the data in `X`, compared to the true responses `Y`.

`L = loss( ____, Name, Value)` computes the error in prediction with additional options specified by one or more `Name, Value` pair arguments, using any of the previous syntaxes.

## Input Arguments

### **ens**

A regression ensemble created with `fitrensemble`, or the `compact` method.

**tbl**

Sample data, specified as a table. Each row of `tbl` corresponds to one observation, and each column corresponds to one predictor variable. `tbl` must contain all of the predictors used to train the model. Multi-column variables and cell arrays other than cell arrays of character vectors are not allowed.

If you trained `ens` using sample data contained in a table, then the input data for this method must also be in a table.

**ResponseVarName**

Response variable name, specified as the name of a variable in `tbl`. The response variable must be a numeric vector.

You must specify `ResponseVarName` as a character vector. For example, if the response variable `Y` is stored as `tbl.Y`, then specify it as `'Y'`. Otherwise, the software treats all columns of `tbl`, including `Y`, as predictors when training the model.

**X**

A matrix of predictor values. Each column of `X` represents one variable, and each row represents one observation.

NaN values in `X` are taken to be missing values. Observations with all missing values for `X` are not used in the calculation of loss.

If you trained `ens` using sample data contained in a matrix, then the input data for this method must also be in a matrix.

**Y**

A numeric column vector with the same number of rows as `tbl` or `X`. Each entry in `Y` is the response to the data in the corresponding row of `tbl` or `X`.

NaN values in `Y` are taken to be missing values. Observations with missing values for `Y` are not used in the calculation of loss.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single

quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### **learners**

Indices of weak learners in the ensemble ranging from 1 to `ens.NumTrained`. `oobEdge` uses only these learners for calculating loss.

**Default:** `1:NumTrained`

### **lossfun**

Function handle for loss function, or 'mse', meaning mean squared error. If you pass a function handle `fun`, `loss` calls it as

```
fun(Y, Yfit, W)
```

where `Y`, `Yfit`, and `W` are numeric vectors of the same length.

- `Y` is the observed response.
- `Yfit` is the predicted response.
- `W` is the observation weights.

The returned value `fun(Y, Yfit, W)` should be a scalar.

**Default:** 'mse'

### **mode**

Meaning of the output `L`:

- 'ensemble' — `L` is a scalar value, the loss for the entire ensemble.
- 'individual' — `L` is a vector with one element per trained learner.
- 'cumulative' — `L` is a vector in which element `J` is obtained by using learners `1:J` from the input list of learners.

**Default:** 'ensemble'

**UseObsForLearner**

A logical matrix of size `N-by-NumTrained`, where `N` is the number of observations in `ens.X`, and `NumTrained` is the number of weak learners. When `UseObsForLearner(I,J)` is true, `predict` uses learner `J` in predicting observation `I`.

**Default:** `true(N,NumTrained)`

**weights**

Numeric vector of observation weights with the same number of elements as `Y`. The formula for `loss` with `weights` is in “Weighted Mean Squared Error” on page 32-3753.

**Default:** `ones(size(Y))`

## Output Arguments

**L**

Weighted mean squared error of predictions. The formula for `loss` is in “Weighted Mean Squared Error” on page 32-3753.

## Examples

**Find Mean-Squared Error of Ensemble Predictions**

Find the loss of an ensemble predictor using the `carsmall` data set.

Load the `carsmall` data set and select engine displacement, horsepower, and vehicle weight as predictors.

```
load carsmall
X = [Displacement Horsepower Weight];
```

Train an ensemble of regression trees and find the regression error for predicting MPG.

```
ens = fitensemble(X,MPG);
L = loss(ens,X,MPG)
```

```
L = 0.3463
```

## Definitions

### Weighted Mean Squared Error

Let  $n$  be the number of rows of data,  $x_j$  be the  $j$ th row of data,  $y_j$  be the true response to  $x_j$ , and let  $f(x_j)$  be the response prediction of `ens` to  $x_j$ . Let  $w$  be the vector of weights (all one by default).

First the weights are divided by their sum so they add to one:  $w \rightarrow w/\sum w$ . The mean squared error  $L$  is

$$L = \sum_{j=1}^n w_j (f(x_j) - y_j)^2.$$

## Extended Capabilities

### Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays” (MATLAB).

### See Also

`fitrensemble` | `predict`

## loss

**Class:** CompactRegressionGP

Regression error for Gaussian process regression model

## Syntax

```
L = loss(gprMdl, Xnew, Ynew)
L = loss(gprMdl, Xnew, Ynew, Name, Value)
```

## Description

`L = loss(gprMdl, Xnew, Ynew)` returns the mean squared error for the Gaussian process regression (GPR) model `gpr`, using the predictors in `Xnew` and observed response in `Ynew`.

`L = loss(gprMdl, Xnew, Ynew, Name, Value)` returns the mean squared error for the GPR model, `gpr`, with additional options specified by one or more `Name, Value` pair arguments. For example, you can specify a custom loss function or the observation weights.

## Input Arguments

**gprMdl** — Gaussian process regression model

RegressionGP object | CompactRegressionGP object

Gaussian process regression model, specified as a RegressionGP (full) or CompactRegressionGP (compact) object.

**Xnew** — New observed data

table |  $m$ -by- $d$  matrix

New data, specified as a table or an  $n$ -by- $d$  matrix, where  $m$  is the number of observations, and  $d$  is the number of predictor variables in the training data.



If you trained `gprMdl` on a table, then `Xnew` must be a table that contains all the predictor variables used to train `gprMdl`.

If `Xnew` is a table, then it can also contain `Ynew`. And if it does, then you do not have to specify `Ynew`.

If you trained `gprMdl` on a matrix, then `Xnew` must be a numeric matrix with  $d$  columns, and can only contain values for the predictor variables.

Data Types: `single` | `double` | `table`

### **`Ynew` — New response values**

$n$ -by-1 vector

New observed response values, that correspond to the predictor values in `Xnew`, specified as an  $n$ -by-1 vector.  $n$  is the number of rows in `Xnew`. Each entry in `Ynew` is the observed response based on the predictor data in the corresponding row of `Xnew`.

If `Xnew` is a table containing new response values, you do not have to specify `Ynew`.

Data Types: `single` | `double`

## **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### **`lossfun` — Loss function**

'mse' (default) | function handle

Loss function, specified as 'mse' (mean squared error) or a function handle.

If you pass a function handle, say `fun`, `loss` calls it as shown below: `fun(Y, Ypred, W)`, where `Y`, `Ypred` and `W` are numeric vectors of length  $n$ , and  $n$  is the number of rows in `Xnew`. `Y` is the observed response, `Ypred` is the predicted response, and `W` is the observation weights.

Example: `'lossfun', Fct` calls the loss function `Fct`.

Data Types: `char` | `function_handle`

**weights** — Observation weightsvector of 1s (default) |  $n$ -by-1 vector

Observation weights, specified as  $n$ -by-1 vector, where  $n$  is the number of rows in  $X_{new}$ . By default, the weight of each observation is 1.

Example: 'weights',  $W$  uses the observation weights in vector  $W$ .

Data Types: double | single

## Output Arguments

**L** — Regression error

scalar value

Regression error for the trained Gaussian process regression model, `gprMdl`, returned as a scalar value.

## Examples

### Compute Regression Loss for Test Data

Load the sample data.

```
load(fullfile(matlabroot, 'examples', 'stats', 'gprdata.mat'))
```

The data has 8 predictor variables and contains 500 observations in training data and 100 observations in test data. This is simulated data.

Fit a GPR model using the squared exponential kernel function with separate length scales for each predictor. Standardize the predictor values in the training data. Use the exact method for fitting and prediction.

```
gprMdl = fitrgp(Xtrain, ytrain, 'FitMethod', 'exact', ...  
    'PredictMethod', 'exact', 'KernelFunction', 'ardsquaredexponential', ...  
    'Standardize', 1);
```

Compute the regression error for the test data.

```
L = loss(gprMdl, Xtest, ytest)
```

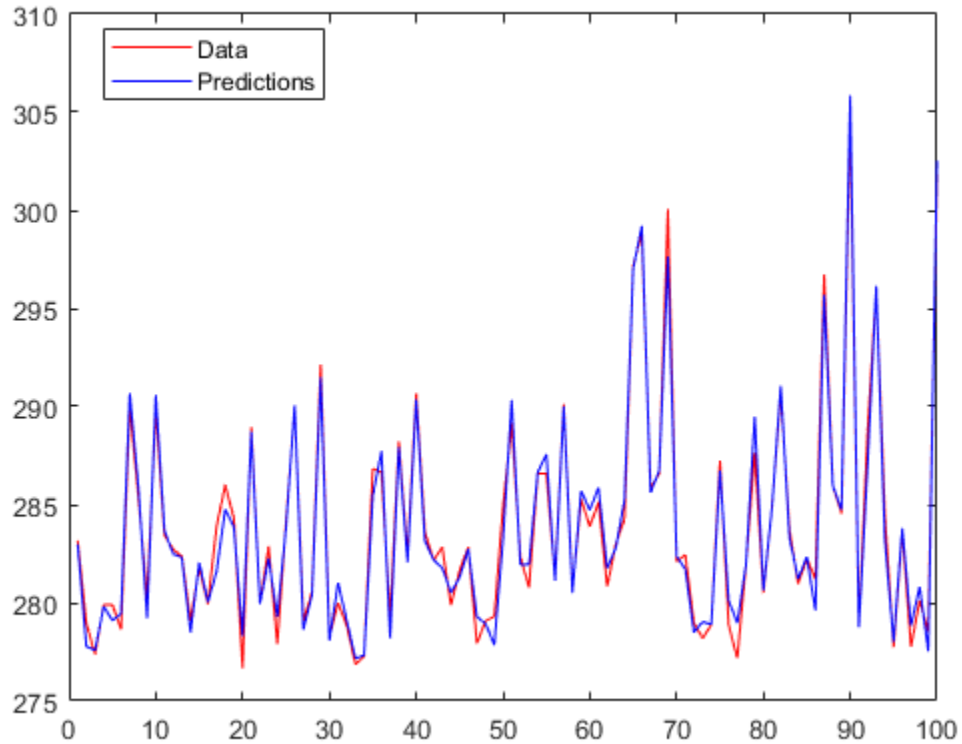
```
L =  
    0.6928
```

**Predict the responses for test data.**

```
ypredtest = predict(gprMdl,Xtest);
```

**Plot the test response along with the predictions.**

```
figure;  
plot(ytest, 'r');  
hold on;  
plot(ypredtest, 'b');  
legend('Data', 'Predictions', 'Location', 'Best');
```



Manually compute the regression loss.

```
L = (ytest - ypredtest)'*(ytest - ypredtest)/length(ytest)
```

```
L =
```

```
0.6928
```

### Specify Custom Loss Function

Load the sample data and store in a `table`.

```
load fisheriris
tbl = table(meas(:,1),meas(:,2),meas(:,3),meas(:,4),species,...
'VariableNames',{'meas1','meas2','meas3','meas4','species'});
```

Fit a GPR model using the first measurement as the response and the other variables as the predictors.

```
mdl = fitrgp(tbl,'meas1');
```

Predict the responses using the trained model.

```
ypred = predict(mdl,tbl);
```

Compute the mean absolute error.

```
n = height(tbl);
y = tbl.meas1;
fun = @(y,ypred,w) sum(abs(y-ypred))/n;
L = loss(mdl,tbl,'lossfun',fun)
```

```
L =
```

```
0.2345
```

## Alternatives

You can use `resubLoss` to compute the regression error for the trained GPR model at the observations in the training data.

## Extended Capabilities

### Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays” (MATLAB).

## See Also

`CompactRegressionGP` | `RegressionGP` | `compact` | `fitrgp` | `predict` | `resubLoss`

**Introduced in R2015b**

# loss

**Class:** CompactRegressionSVM

Regression error for support vector machine regression model

## Syntax

```
L = loss (mdl, tbl, ResponseVarName)
L = loss (mdl, tbl, Y)
L = loss (mdl, X, Y)
L = loss ( ____, Name, Value)
```

## Description

`L = loss (mdl, tbl, ResponseVarName)` returns the loss for the predictions of the support vector machine (SVM) regression model, `mdl`, based on the predictor data in the table `tbl` and the true response values in `tbl.ResponseVarName`.

`L = loss (mdl, tbl, Y)` returns the loss for the predictions of the support vector machine (SVM) regression model, `mdl`, based on the predictor data in the table `X` and the true response values in the vector `Y`.

`L = loss (mdl, X, Y)` returns the loss for the predictions of the support vector machine (SVM) regression model, `mdl`, based on the predictor data in `X` and the true responses in `Y`.

`L = loss ( ____, Name, Value)` returns the loss with additional options specified by one or more `Name, Value` pair arguments, using any of the previous syntaxes. For example, you can specify the loss function or observation weights.

## Input Arguments

**mdl** — SVM regression model

RegressionSVM model | CompactRegressionSVM model

SVM regression model, specified as a `RegressionSVM` model or `CompactRegressionSVM` model returned by `fitrsvm` or `compact`, respectively.

**tbl** — Sample data

table

Sample data, specified as a table. Each row of `tbl` corresponds to one observation, and each column corresponds to one predictor variable. Optionally, `tbl` can contain additional columns for the response variable and observation weights. `tbl` must contain all of the predictors used to train `mdl`. Multi-column variables and cell arrays other than cell arrays of character vectors are not allowed.

If you trained `mdl` using sample data contained in a `table`, then the input data for this method must also be in a `table`.

Data Types: `table`

**ResponseVarName** — Response variable name

name of a variable in `tbl`

Response variable name, specified as the name of a variable in `tbl`. The response variable must be a numeric vector.

You must specify `ResponseVarName` as a character vector. For example, if the response variable `Y` is stored as `tbl.Y`, then specify it as `'Y'`. Otherwise, the software treats all columns of `tbl`, including `Y`, as predictors when training the model.

**x** — Predictor data

numeric matrix

Predictor data, specified as a numeric matrix or table. Each row of `x` corresponds to one observation (also known as an instance or example), and each column corresponds to one variable (also known as a feature).

If you trained `mdl` using a matrix of predictor values, then `x` must be a numeric matrix with  $p$  columns.  $p$  is the number of predictors used to train `mdl`.

The length of `Y` and the number of rows of `x` must be equal.

Data Types: `single` | `double`



**Y — Observed response values**

vector of numeric values

Observed response values, specified as a vector of length  $n$  containing numeric values. Each entry in  $Y$  is the observed response based on the predictor data in the corresponding row of  $X$ .

Data Types: `single` | `double`**Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

**LossFun — Loss function**`'mse'` (default) | `'epsiloninsensitive'` | function handle

Loss function, specified as the comma-separated pair consisting of `'LossFun'` and `'mse'`, `'epsiloninsensitive'`, or a function handle.

- The following lists available loss functions.

Value	Loss Function
<code>'mse'</code>	“Weighted Mean Squared Error” on page 32-3765
<code>'epsiloninsensitive'</code>	“Epsilon-Insensitive Loss Function” on page 32-3766

- Specify your own function using function handle notation.

Your function must have the signature `lossvalue = lossfun(Y, Yfit, W)`, where:

- The output argument `lossvalue` is a scalar value.
- You choose the function name (*lossfun*).
- $Y$  is an  $n$ -by-1 numeric vector of observed response values.
- $Yfit$  is an  $n$ -by-1 numeric vector of predicted response values, calculated using the corresponding predictor values in  $X$  (similar to the output of `predict`).

- $W$  is an  $n$ -by-1 numeric vector of observation weights. If you pass  $W$ , the software normalizes them to sum to 1.

Specify your function using `'LossFun', @lossfun`.

Example: `'LossFun', 'epsiloninsensitive'`

Data Types: `char` | `function_handle`

### **Weights — Observation weights**

`ones(size(X,1))` (default) | numeric vector

Observation weights, specified as the comma-separated pair consisting of `'Weights'` and a numeric vector. `Weights` must be the same length as the number of rows in  $X$ . The software weighs the observations in each row of  $X$  using the corresponding weight value in `Weights`.

Weights are normalized to sum to 1.

Data Types: `single` | `double`

## Output Arguments

### **L — Regression loss**

scalar value

Regression loss, returned as a scalar value.

## Examples

### **Calculate Test Sample Loss for SVM Regression Model**

This example shows how to train an SVM regression model, then calculate the resubstitution mean square error and epsilon-insensitive error.

Load the `carsmall` sample data.

```
load carsmall
rng default % for reproducibility
```

Specify Horsepower and Weight as the predictor variables (X), and MPG as the response variable (Y).

```
X = [Horsepower,Weight];  
Y = MPG;
```

Train a linear SVM regression model. Standardize the data.

```
mdl = fitrsvm(X,Y,'Standardize',true);
```

mdl is a RegressionSVM model.

Determine how well the trained model generalizes to new predictor values by estimating the test sample mean square error and epsilon-insensitive error.

```
lossMSE = loss(mdl,X,Y)  
lossEI = loss(mdl,X,Y,'LossFun','epsiloninsensitive')
```

```
lossMSE =  
  
    17.0256
```

```
lossEI =  
  
     2.2506
```

## Definitions

### Weighted Mean Squared Error

The weighted mean squared error is calculated as follows:

$$\text{mse} = \frac{\sum_{j=1}^n w_j (f(x_j) - y_j)^2}{\sum_{j=1}^n w_j},$$

where:

- $n$  is the number of rows of data
- $x_j$  is the  $j$ th row of data
- $y_j$  is the true response to  $x_j$
- $f(x_j)$  is the response prediction of the SVM regression model `mdl` to  $x_j$
- $w$  is the vector of weights.

The weights in  $w$  are all equal to one by default. You can specify different values for weights using the 'Weights' name-value pair argument. If you specify weights, each value is divided by the sum of all weights, such that the normalized weights add to one.

## Epsilon-Insensitive Loss Function

The epsilon-insensitive loss function ignores errors that are within the distance epsilon ( $\epsilon$ ) of the function value. It is formally described as:

$$Loss_{\epsilon} = \begin{cases} 0, & \text{if } |y - f(x)| \leq \epsilon \\ |y - f(x)| - \epsilon, & \text{otherwise.} \end{cases}$$

The mean epsilon-insensitive loss is calculated as follows:

$$Loss = \frac{\sum_{j=1}^n w_j \max(0, |y_j - f(x_j)| - \epsilon)}{\sum_{j=1}^n w_j},$$

where:

- $n$  is the number of rows of data
- $x_j$  is the  $j$ th row of data
- $y_j$  is the true response to  $x_j$
- $f(x_j)$  is the response prediction of the SVM regression model `mdl` to  $x_j$
- $w$  is the vector of weights.

The weights in  $w$  are all equal to one by default. You can specify different values for weights using the 'Weights' name-value pair argument. If you specify weights, each value is divided by the sum of all weights, such that the normalized weights add to one.

## Tips

- If `mdl` is a cross-validated `RegressionPartitionedSVM` model, use `kfoldloss` instead of `loss` to calculate the regression error.

## Extended Capabilities

### Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays” (MATLAB).

### See Also

`CompactRegressionSVM` | `RegressionSVM` | `fitrsvm` | `kfoldloss`

**Introduced in R2015b**

## loss

**Class:** CompactRegressionTree

Regression error

## Syntax

```
L = loss(tree, tbl, ResponseVarName)
L = loss(tree, x, y)
L = loss(____, Name, Value)
[L, se, NLeaf, bestlevel] = loss(____)
```

## Description

`L = loss(tree, tbl, ResponseVarName)` returns the mean squared error between the predictions of `tree` to the data in `tbl`, compared to the true responses `tbl.ResponseVarName`.

`L = loss(tree, x, y)` returns the mean squared error between the predictions of `tree` to the data in `x`, compared to the true responses `y`.

`L = loss(____, Name, Value)` computes the error in prediction with additional options specified by one or more `Name, Value` pair arguments, using any of the previous syntaxes.

`[L, se, NLeaf, bestlevel] = loss(____)` also returns the standard error of the loss (`se`), the number of leaves (terminal nodes) in the tree (`NLeaf`), and the optimal pruning level for `tree` (`bestlevel`).

## Input Arguments

**tree** — Trained regression tree

RegressionTree object | CompactRegressionTree object

Trained regression tree, specified as a `RegressionTree` object constructed by `fitrtree` or a `CompactRegressionTree` object constructed by `compact`.

**x — Predictor values**

matrix of floating-point values

Predictor values, specified as matrix of floating-point values. Each column of `x` represents one variable, and each row represents one observation.

Data Types: `single` | `double`

**ResponseVarName — Response variable name**

name of a variable in `tbl`

Response variable name, specified as the name of a variable in `tbl`.

You must specify `ResponseVarName` as a character vector. For example, if the response variable `y` is stored as `tbl.y`, then specify it as `'response'`. Otherwise, the software treats all columns of `tbl`, including `y`, as predictors when training the model.

The response variable must be a categorical or character array, logical or numeric vector, or cell array of character vectors. If `y` is a character array, then each element must correspond to one row of the array.

It is good practice to specify the order of the classes using the `ClassNames` name-value pair argument.

**y — Response data**

numeric column vector

Response data, specified as a numeric column vector with the same number of rows as `x`. Each entry in `y` is the response to the data in the corresponding row of `x`.

Data Types: `single` | `double`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

**LossFun — Loss function**`'mse'` (default) | function handle

Loss function, specified as the comma-separated pair consisting of `'LossFun'` and a function handle for loss, or `'mse'` representing mean-squared error. If you pass a function handle `fun`, `loss` calls `fun` as:

```
fun(Y,Yfit,W)
```

- `Y` is the vector of true responses.
- `Yfit` is the vector of predicted responses.
- `W` is the observation weights. If you pass `W`, the elements are normalized to sum to 1.

All the vectors have the same number of rows as `Y`.

Example: `'LossFun', 'mse'`

Data Types: `function_handle`

**Subtrees — Pruning level**`0` (default) | vector of nonnegative integers | `'all'`

Pruning level, specified as the comma-separated pair consisting of `'Subtrees'` and a vector of nonnegative integers in ascending order or `'all'`.

If you specify a vector, then all elements must be at least 0 and at most `max(tree.PruneList)`. 0 indicates the full, unpruned tree and `max(tree.PruneList)` indicates the completely pruned tree (i.e., just the root node).

If you specify `'all'`, then `CompactRegressionTree.loss` operates on all subtrees (i.e., the entire pruning sequence). This specification is equivalent to using `0:max(tree.PruneList)`.

`CompactRegressionTree.loss` prunes `tree` to each level indicated in `Subtrees`, and then estimates the corresponding output arguments. The size of `Subtrees` determines the size of some output arguments.

To invoke `Subtrees`, the properties `PruneList` and `PruneAlpha` of `tree` must be nonempty. In other words, grow `tree` by setting `'Prune', 'on'`, or by pruning `tree` using `prune`.

Example: `'Subtrees', 'all'`



**TreeSize — Tree size**`'se' (default) | 'min'`

Tree size, specified as the comma-separated pair consisting of `'TreeSize'` and one of the following:

- `'se'` — `loss` returns `bestlevel` that corresponds to the smallest tree whose mean squared error (MSE) is within one standard error of the minimum MSE.
- `'min'` — `loss` returns `bestlevel` that corresponds to the minimal MSE tree.

Example: `'TreeSize', 'min'`

**Weights — Observation weights**`ones(size(X,1),1) (default) | vector of scalar values`

Observation weights, specified as the comma-separated pair consisting of `'Weights'` and a vector of scalar values. The software weights the observations in each row of `x` or `tbl` with the corresponding value in `Weights`. The size of `Weights` must equal the number of rows in `x` or `tbl`.

If you specify the input data as a table `tbl`, then `Weights` can be the name of a variable in `tbl` that contains a numeric vector. In this case, you must specify `Weights` as a variable name. For example, if weights vector `W` is stored as `tbl.W`, then specify it as `'W'`. Otherwise, the software treats all columns of `tbl`, including `W`, as predictors when training the model.

Data Types: `single` | `double`

## Output Arguments

**L — Classification error**

vector of scalar values

Classification error, returned as a vector the length of `Subtrees`. The error for each tree is the mean squared error, weighted with `Weights`. If you include `LossFun`, `L` reflects the loss calculated with `LossFun`.

**se — Standard error of loss**

vector of scalar values

Standard error of loss, returned as a vector the length of `Subtrees`.

**nLeaf** — Number of leaf nodes

vector of integer values

Number of leaves (terminal nodes) in the pruned subtrees, returned as a vector the length of `Subtrees`.

**bestlevel** — Best pruning level

scalar value

Best pruning level as defined in the `TreeSize` name-value pair, returned as a scalar whose value depends on `TreeSize`:

- `TreeSize = 'se'` — `loss` returns the highest pruning level with loss within one standard deviation of the minimum ( $L+se$ , where  $L$  and  $se$  relate to the smallest value in `Subtrees`).
- `TreeSize = 'min'` — `loss` returns the element of `Subtrees` with smallest loss, usually the smallest element of `Subtrees`.

## Examples

**Compute the In-Sample MSE**

Load the `carsmall` data set. Consider `Displacement`, `Horsepower`, and `Weight` as predictors of the response `MPG`.

```
load carsmall
X = [Displacement Horsepower Weight];
```

Grow a regression tree using all observations.

```
tree = fitrtree(X,MPG);
```

Estimate the in-sample MSE.

```
L = loss(tree,X,MPG)
```

```
L =
```

---

4.8952

### Find the Pruning Level Yielding the Optimal In-sample Loss

Load the `carsmall` data set. Consider `Displacement`, `Horsepower`, and `Weight` as predictors of the response `MPG`.

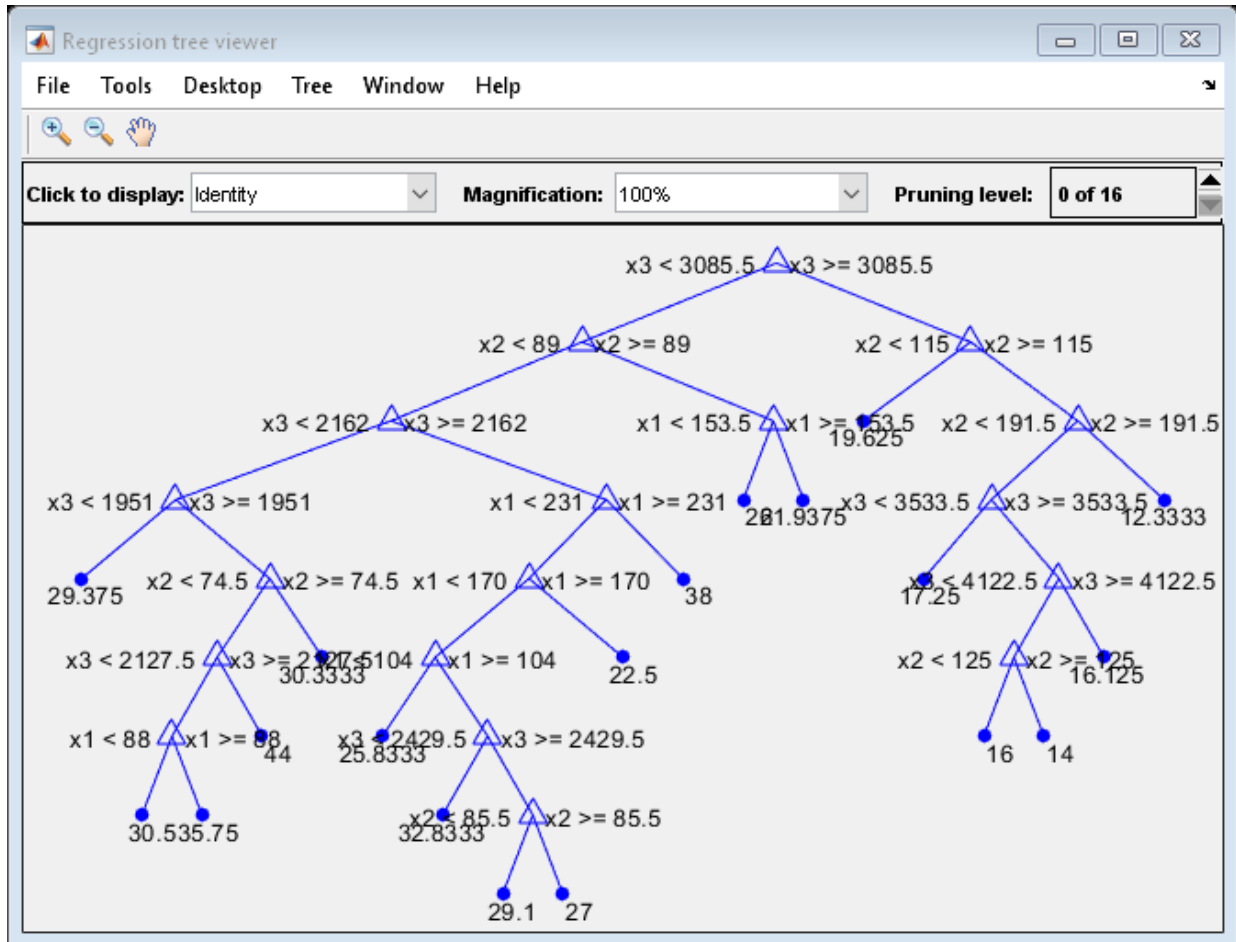
```
load carsmall
X = [Displacement Horsepower Weight];
```

Grow a regression tree using all observations.

```
Mdl = fitrtree(X,MPG);
```

View the regression tree.

```
view(Mdl, 'Mode', 'graph');
```



Find the best pruning level that yields the optimal in-sample loss.

```
[L, se, NLeaf, bestLevel] = loss(Mdl, X, MPG, 'Subtrees', 'all');
bestLevel
```

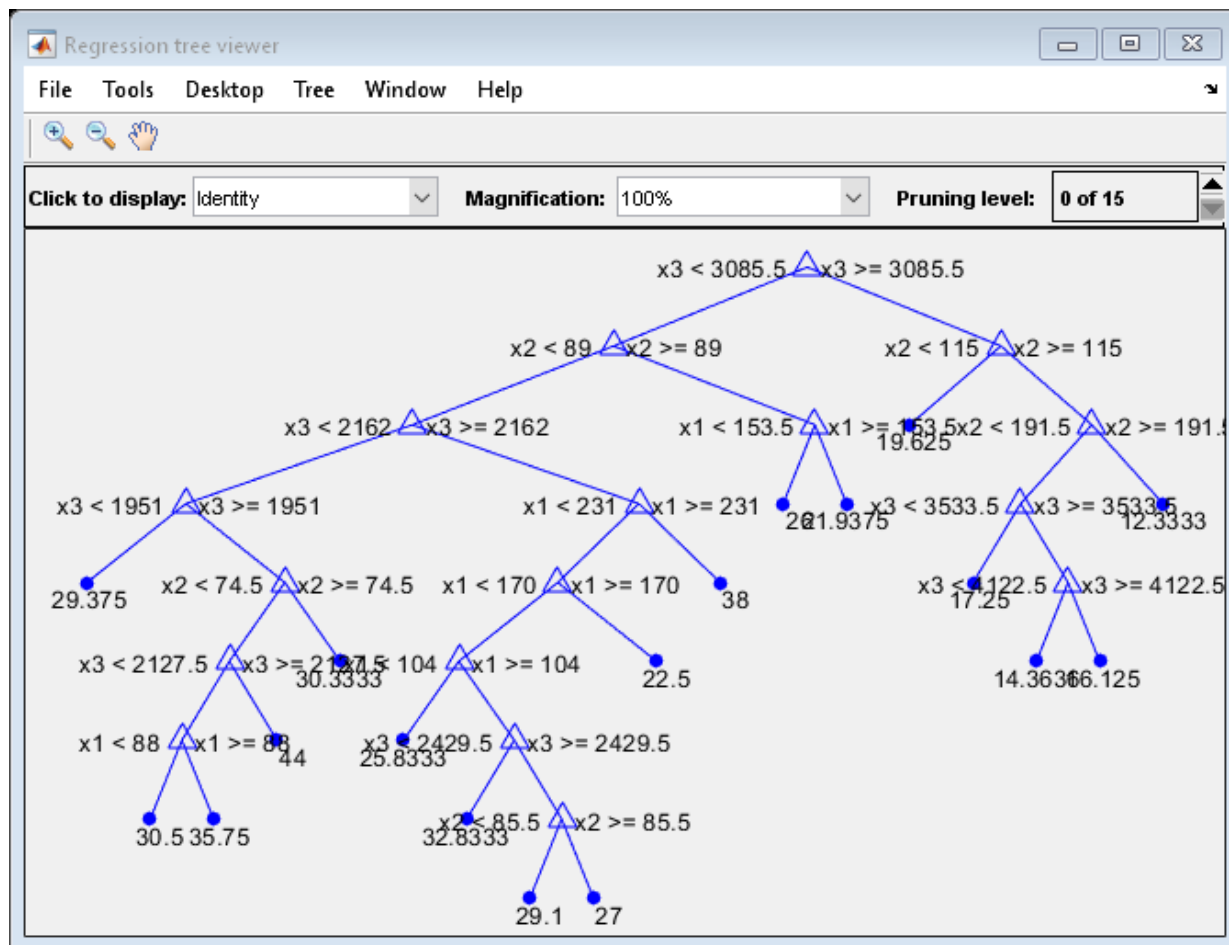
```
bestLevel =
```

```
1
```

The best pruning level is level 1.

Prune the tree to level 1.

```
pruneMdl = prune(Mdl, 'Level', bestLevel);
view(pruneMdl, 'Mode', 'graph');
```



### Examine the MSE for Each Subtree

Unpruned decision trees tend to overfit. One way to balance model complexity and out-of-sample performance is to prune a tree (or restrict its growth) so that in-sample and out-of-sample performance are satisfactory.

Load the `carsmall` data set. Consider Displacement, Horsepower, and Weight as predictors of the response MPG.

```
load carsmall
X = [Displacement Horsepower Weight];
Y = MPG;
```

Partition the data into training (50%) and validation (50%) sets.

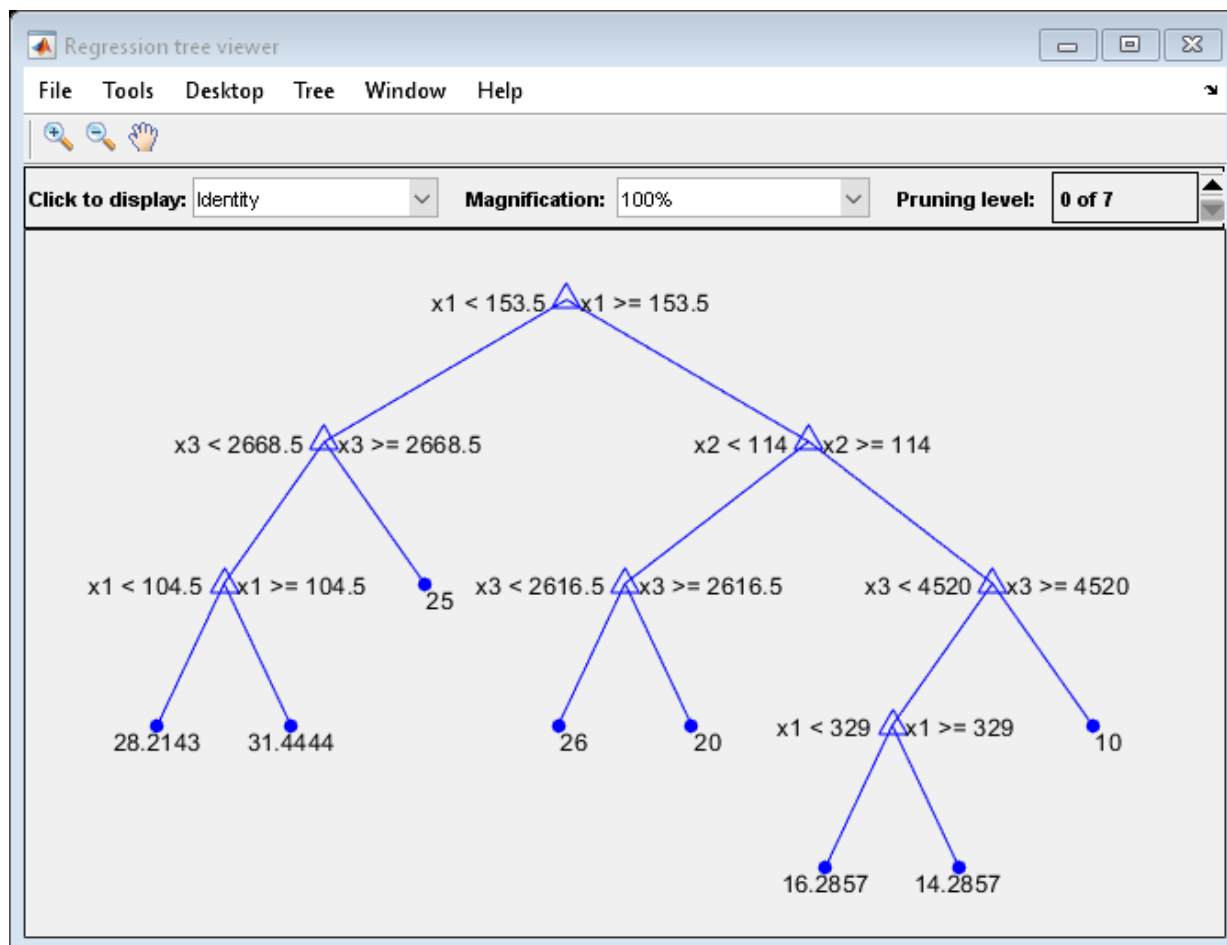
```
n = size(X,1);
rng(1) % For reproducibility
idxTrn = false(n,1);
idxTrn(randsample(n,round(0.5*n))) = true; % Training set logical indices
idxVal = idxTrn == false; % Validation set logical indices
```

Grow a regression tree using the training set.

```
Mdl = fitrtree(X(idxTrn,:),Y(idxTrn));
```

View the regression tree.

```
view(Mdl, 'Mode', 'graph');
```



The regression tree has seven pruning levels. Level 0 is the full, unpruned tree (as displayed). Level 7 is just the root node (i.e., no splits).

Examine the training sample MSE for each subtree (or pruning level) excluding the highest level.

```
m = max(Mdl.PruneList) - 1;
trnLoss = resubLoss(Mdl, 'SubTrees', 0:m)
```

```
trnLoss =
```

```
5.9789
6.2768
6.8316
7.5209
8.3951
10.7452
14.8445
```

- The MSE for the full, unpruned tree is about 6 units.
- The MSE for the tree pruned to level 1 is about 6.3 units.
- The MSE for the tree pruned to level 6 (i.e., a stump) is about 14.8 units.

Examine the validation sample MSE at each level excluding the highest level.

```
valLoss = loss(Mdl,X(idxVal,:),Y(idxVal),'SubTrees',0:m)
```

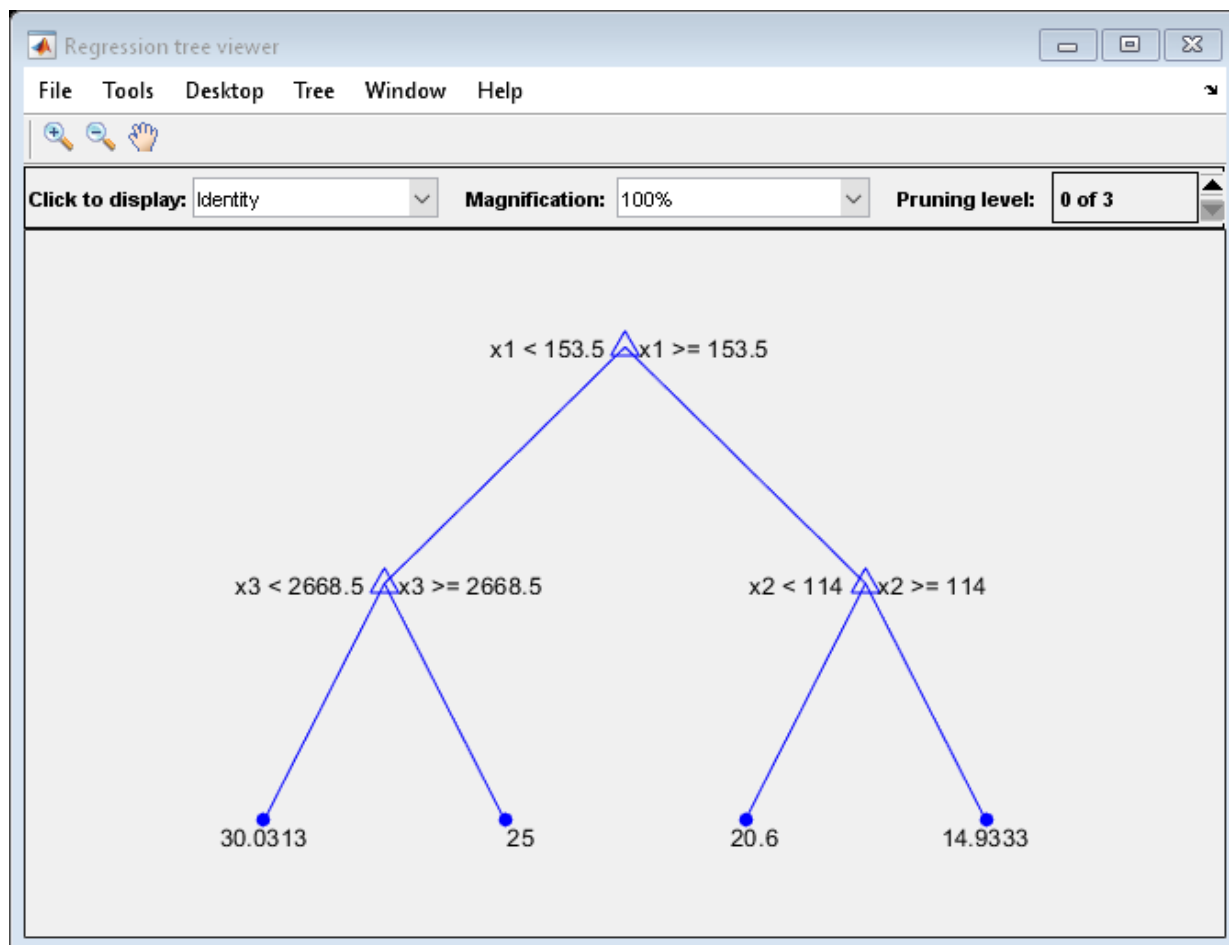
```
valLoss =
32.1205
31.5035
32.0541
30.8183
26.3535
30.0137
38.4695
```

- The MSE for the full, unpruned tree (level 0) is about 32.1 units.
- The MSE for the tree pruned to level 4 is about 26.4 units.
- The MSE for the tree pruned to level 5 is about 30.0 units.
- The MSE for the tree pruned to level 6 (i.e., a stump) is about 38.5 units.

To balance model complexity and out-of-sample performance, consider pruning `Mdl` to level 4.

```
pruneMdl = prune(Mdl,'Level',4);
view(pruneMdl,'Mode','graph')
```





## Definitions

### Mean Squared Error

The mean squared error  $m$  of the predictions  $f(X_n)$  with weight vector  $w$  is

$$m = \frac{\sum w_n (f(X_n) - Y_n)^2}{\sum w_n}.$$

## Extended Capabilities

### Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function supports tall arrays for out-of-memory data with the limitation:

- Only one output is supported.

For more information, see “Tall Arrays” (MATLAB).

### See Also

`fitrtree` | `predict`

# loss

**Class:** FeatureSelectionNCAClassification

Evaluate accuracy of learned feature weights on test data

## Syntax

```
err = loss mdl, X, Y
err = loss mdl, X, Y, Name, Value
```

## Description

`err = loss mdl, X, Y` computes the misclassification error of the model `mdl`, for the predictors in `X` and the class labels in `Y`.

`err = loss mdl, X, Y, Name, Value` computes the classification error with additional options specified by one or more `Name, Value` pair arguments.

## Input Arguments

**mdl** — Neighborhood component analysis model for classification

FeatureSelectionNCAClassification object

Neighborhood component analysis model for classification, returned as a FeatureSelectionNCAClassification object.

**x** — Predictor variable values

*n*-by-*p* matrix

Predictor variable values, specified as an *n*-by-*p* matrix, where *n* is the number of observations and *p* is the number of predictor variables.

Data Types: single | double

**$\mathbf{Y}$  — Class labels**

categorical vector | logical vector | numeric vector | cell array of character vectors of length  $n$  | character matrix with  $n$  rows

Class labels, specified as a categorical vector, logical vector, numeric vector, cell array of character vectors of length  $n$ , or character matrix with  $n$  rows, where  $n$  is the number of observations. Element  $i$  or row  $i$  of  $\mathbf{Y}$  is the class label corresponding to row  $i$  of  $\mathbf{X}$  (observation  $i$ ).

Data Types: `single` | `double` | `logical` | `char` | `cell` | `categorical`

**Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

**LossFunction — Loss function type**

'`classiferror`' (default) | '`quadratic`'

Loss function type, specified as a comma-separated pair consisting of '`Loss Function`' and one of the following.

- '`classiferror`' — Misclassification error, defined as

$$\frac{1}{n} \sum_{i=1}^n I(k_i \neq t_i),$$

where  $k_i$  is the predicted class and  $t_i$  is the true class for observation  $i$ .  $I(k_i \neq t_i)$  is the indicator for when the  $k_i$  is not the same as  $t_i$ .

- '`quadratic`' — Quadratic loss function, defined as

$$\frac{1}{n} \sum_{i=1}^n \sum_{k=1}^c (p_{ik} - I(i,k))^2,$$

where  $c$  is the number of classes,  $p_{ik}$  is the estimate probability that  $i$ th observation belongs to class  $k$ , and  $I(i,k)$  is the indicator that  $i$ th observation belongs to class  $k$ .

Example: 'LossFunction', 'quadratic'

Data Types: char

## Output Arguments

**err** — Smaller-the-better accuracy measure for learned feature weights

scalar value

Smaller-the-better accuracy measure for learned feature weights, returned as a scalar value. You can specify the measure of accuracy using the `LossFunction` name-value pair argument.

## Examples

### Tune NCA Model for Classification

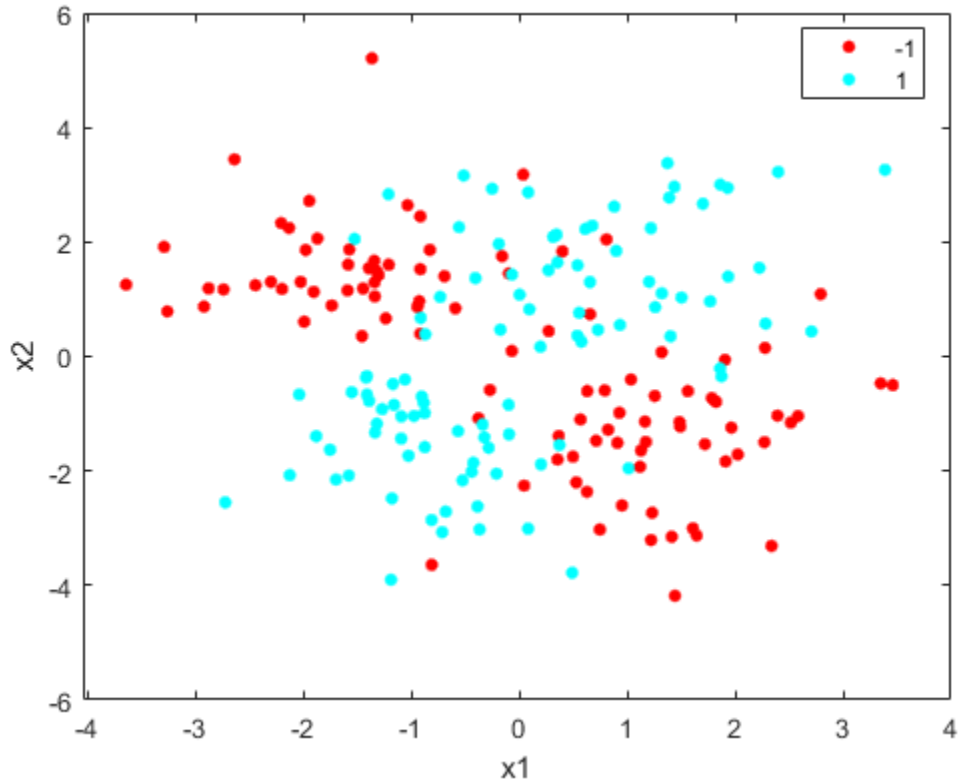
Load the sample data.

```
load(fullfile(matlabroot, 'examples', 'stats', 'twodimclassdata.mat'));
```

This data set is simulated using the scheme described in [1]. This is a two-class classification problem in two dimensions. Data from the first class are drawn from two bivariate normal distributions  $N(\mu_1, \Sigma)$  or  $N(\mu_2, \Sigma)$  with equal probability, where  $\mu_1 = [-0.75, -1.5]$ ,  $\mu_2 = [0.75, 1.5]$ , and  $\Sigma = I_2$ . Similarly, data from the second class are drawn from two bivariate normal distributions  $N(\mu_3, \Sigma)$  or  $N(\mu_4, \Sigma)$  with equal probability, where  $\mu_3 = [1.5, -1.5]$ ,  $\mu_4 = [-1.5, 1.5]$ , and  $\Sigma = I_2$ . The normal distribution parameters used to create this data set result in tighter clusters in data than the data used in [1].

Create a scatter plot of the data grouped by the class.

```
figure
gscatter(X(:,1), X(:,2), y)
xlabel('x1')
ylabel('x2')
```



Add 100 irrelevant features to  $X$ . First generate data from a Normal distribution with a mean of 0 and a variance of 20.

```
n = size(X,1);
rng('default')
XwithBadFeatures = [X, randn(n,100)*sqrt(20)];
```

Normalize the data so that all points are between 0 and 1.

```
XwithBadFeatures = bsxfun(@rdivide, ...
    bsxfun(@minus, XwithBadFeatures, min(XwithBadFeatures, [], 1)), ...
    range(XwithBadFeatures, 1));
X = XwithBadFeatures;
```

Fit a neighborhood component analysis (NCA) model to the data using the default Lambda (regularization parameter,  $\lambda$ ) value. Use the LBFGS solver and display the convergence information.

```
ncaMdl = fscnca(X,y,'FitMethod','exact','Verbose',1, ...
               'Solver','lbfgs');
```

```
o Solver = LBFGS, HessianHistorySize = 15, LineSearchMethod = weakwolfe
```

```
=====
| ITER | FUN VALUE | NORM GRAD | NORM STEP | CURV | GAMMA | ALPH |
|=====|=====|=====|=====|=====|=====|=====|
| 0 | 9.519258e-03 | 1.494e-02 | 0.000e+00 | | 4.015e+01 | 0.000 |
| 1 | -3.093574e-01 | 7.186e-03 | 4.018e+00 | OK | 8.956e+01 | 1.000 |
| 2 | -4.809455e-01 | 4.444e-03 | 7.123e+00 | OK | 9.943e+01 | 1.000 |
| 3 | -4.938877e-01 | 3.544e-03 | 1.464e+00 | OK | 9.366e+01 | 1.000 |
| 4 | -4.964759e-01 | 2.901e-03 | 6.084e-01 | OK | 1.554e+02 | 1.000 |
| 5 | -4.972077e-01 | 1.323e-03 | 6.129e-01 | OK | 1.195e+02 | 5.000 |
| 6 | -4.974743e-01 | 1.569e-04 | 2.155e-01 | OK | 1.003e+02 | 1.000 |
| 7 | -4.974868e-01 | 3.844e-05 | 4.161e-02 | OK | 9.835e+01 | 1.000 |
| 8 | -4.974874e-01 | 1.417e-05 | 1.073e-02 | OK | 1.043e+02 | 1.000 |
| 9 | -4.974874e-01 | 4.893e-06 | 1.781e-03 | OK | 1.530e+02 | 1.000 |
| 10 | -4.974874e-01 | 9.404e-08 | 8.947e-04 | OK | 1.670e+02 | 1.000 |
|=====|=====|=====|=====|=====|=====|=====|
```

```
Infinity norm of the final gradient = 9.404e-08
```

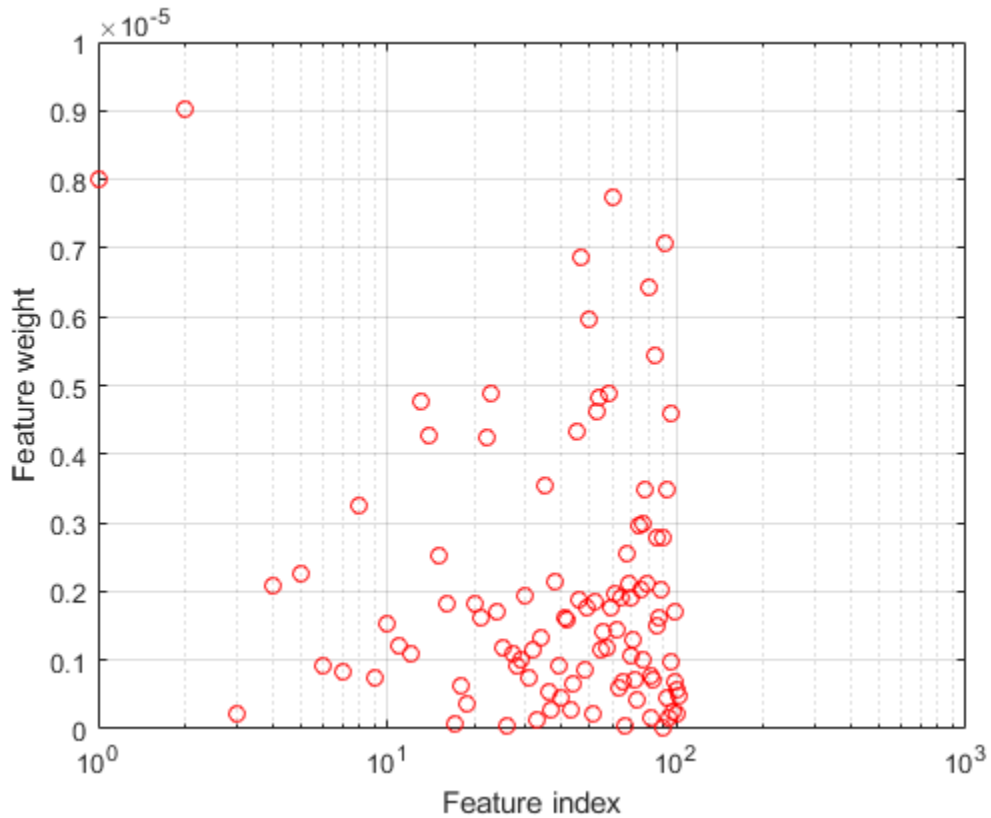
```
Two norm of the final step = 8.947e-04, TolX = 1.000e-06
```

```
Relative infinity norm of the final gradient = 9.404e-08, TolFun = 1.000e-06
```

```
EXIT: Local minimum found.
```

Plot the feature weights. The weights of the irrelevant features should be very close to zero.

```
figure
semilogx(ncaMdl.FeatureWeights,'ro')
xlabel('Feature index')
ylabel('Feature weight')
grid on
```



Predict the classes using the NCA model and compute the confusion matrix.

```
ypred = predict(ncaMdl,X);
confusionmat(y,ypred)
```

```
ans =
```

```
40    60
 6    94
```

Confusion matrix shows that 40 of the data that are in class 1 are predicted as belonging to class 1. 60 of the data from class 1 are predicted to be in class 2. Similarly, 94 of the



data from class 2 are predicted to be from class 2 and 6 of them are predicted to be from class 1. The prediction accuracy for class 1 is not good.

All weights are very close to zero, which indicates that the value of  $\lambda$  used in training the model is too large. When  $\lambda \rightarrow \infty$ , all features weights approach to zero. Hence, it is important to tune the regularization parameter in most cases to detect the relevant features.

Use five-fold cross-validation to tune  $\lambda$  for feature selection by using `fscnca`. Tuning  $\lambda$  means finding the  $\lambda$  value that will produce the minimum classification loss. To tune  $\lambda$  using cross-validation:

1. Partition the data into five folds. For each fold, `cvpartition` assigns four-fifths of the data as a training set and one-fifth of the data as a test set. Again for each fold, `cvpartition` creates a stratified partition, where each partition has roughly the same proportion of classes.

```
cvp = cvpartition(y, 'kfold', 5);
numtestsets = cvp.NumTestSets;
lambdavalues = linspace(0, 2, 20)/length(y);
lossvalues = zeros(length(lambdavalues), numtestsets);
```

2. Train the neighborhood component analysis (nca) model for each  $\lambda$  value using the training set in each fold.

3. Compute the classification loss for the corresponding test set in the fold using the nca model. Record the loss value.

4. Repeat this process for all folds and all  $\lambda$  values.

```
for i = 1:length(lambdavalues)
    for k = 1:numtestsets

        % Extract the training set from the partition object
        Xtrain = X(cvp.training(k), :);
        ytrain = y(cvp.training(k), :);

        % Extract the test set from the partition object
        Xtest = X(cvp.test(k), :);
        ytest = y(cvp.test(k), :);

        % Train an NCA model for classification using the training set
        ncaMdl = fscnca(Xtrain, ytrain, 'FitMethod', 'exact', ...
```

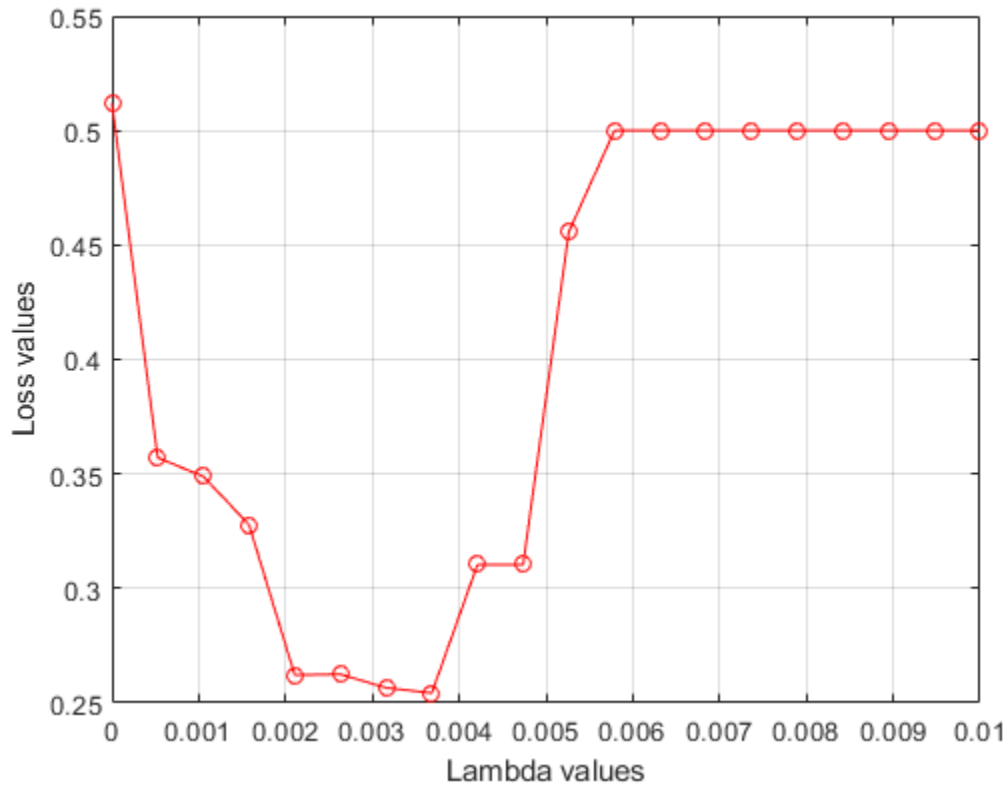
```
        'Solver','lbfgs','Lambda',lambdavalues(i));

    % Compute the classification loss for the test set using the NCA
    % model
    lossvalues(i,k) = loss(ncaMdl,Xtest,ytest, ...
        'LossFunction','quadratic');

end
end
```

Plot the average loss values of the folds versus the  $\lambda$  values. If the  $\lambda$  value that corresponds to the minimum loss falls on the boundary of the tested  $\lambda$  values, the range of  $\lambda$  values should be reconsidered.

```
figure
plot(lambdavalues,mean(lossvalues,2),'ro-')
xlabel('Lambda values')
ylabel('Loss values')
grid on
```



Find the  $\lambda$  value that corresponds to the minimum average loss.

```
[~,idx] = min(mean(lossvalues,2)); % Find the index
bestlambda = lambdavalues(idx) % Find the best lambda value
```

```
bestlambda =
```

```
0.0037
```

Fit the NCA model to all of the data using the best  $\lambda$  value. Use the LBFGS solver and display the convergence information.

```
ncaMdl = fscnca(X,y,'FitMethod','exact','Verbose',1, ...
               'Solver','lbfgs','Lambda',bestlambda);
```

```
o Solver = LBFSGS, HessianHistorySize = 15, LineSearchMethod = weakwolfe
```

```
=====
| ITER | FUN VALUE | NORM GRAD | NORM STEP | CURV | GAMMA | ALPH
|=====|=====|=====|=====|=====|=====|=====|
| 0 | -1.246913e-01 | 1.231e-02 | 0.000e+00 | | 4.873e+01 | 0.000
| 1 | -3.411330e-01 | 5.717e-03 | 3.618e+00 | OK | 1.068e+02 | 1.000
| 2 | -5.226111e-01 | 3.763e-02 | 8.252e+00 | OK | 7.825e+01 | 1.000
| 3 | -5.817731e-01 | 8.496e-03 | 2.340e+00 | OK | 5.591e+01 | 5.000
| 4 | -6.132632e-01 | 6.863e-03 | 2.526e+00 | OK | 8.228e+01 | 1.000
| 5 | -6.135264e-01 | 9.373e-03 | 7.341e-01 | OK | 3.244e+01 | 1.000
| 6 | -6.147894e-01 | 1.182e-03 | 2.933e-01 | OK | 2.447e+01 | 1.000
| 7 | -6.148714e-01 | 6.392e-04 | 6.688e-02 | OK | 3.195e+01 | 1.000
| 8 | -6.149524e-01 | 6.521e-04 | 9.934e-02 | OK | 1.236e+02 | 1.000
| 9 | -6.149972e-01 | 1.154e-04 | 1.191e-01 | OK | 1.171e+02 | 1.000
| 10 | -6.149990e-01 | 2.922e-05 | 1.983e-02 | OK | 7.365e+01 | 1.000
| 11 | -6.149993e-01 | 1.556e-05 | 8.354e-03 | OK | 1.288e+02 | 1.000
| 12 | -6.149994e-01 | 1.147e-05 | 7.256e-03 | OK | 2.332e+02 | 1.000
| 13 | -6.149995e-01 | 1.040e-05 | 6.781e-03 | OK | 2.287e+02 | 1.000
| 14 | -6.149996e-01 | 9.015e-06 | 6.265e-03 | OK | 9.974e+01 | 1.000
| 15 | -6.149996e-01 | 7.763e-06 | 5.206e-03 | OK | 2.919e+02 | 1.000
| 16 | -6.149997e-01 | 8.374e-06 | 1.679e-02 | OK | 6.878e+02 | 1.000
| 17 | -6.149997e-01 | 9.387e-06 | 9.542e-03 | OK | 1.284e+02 | 5.000
| 18 | -6.149997e-01 | 3.250e-06 | 5.114e-03 | OK | 1.225e+02 | 1.000
| 19 | -6.149997e-01 | 1.574e-06 | 1.275e-03 | OK | 1.808e+02 | 1.000
|=====|=====|=====|=====|=====|=====|=====|
```

```
=====
| ITER | FUN VALUE | NORM GRAD | NORM STEP | CURV | GAMMA | ALPH
|=====|=====|=====|=====|=====|=====|=====|
| 20 | -6.149997e-01 | 5.764e-07 | 6.765e-04 | OK | 2.905e+02 | 1.000
|=====|=====|=====|=====|=====|=====|=====|
```

```
Infinity norm of the final gradient = 5.764e-07
```

```
Two norm of the final step = 6.765e-04, TolX = 1.000e-06
```

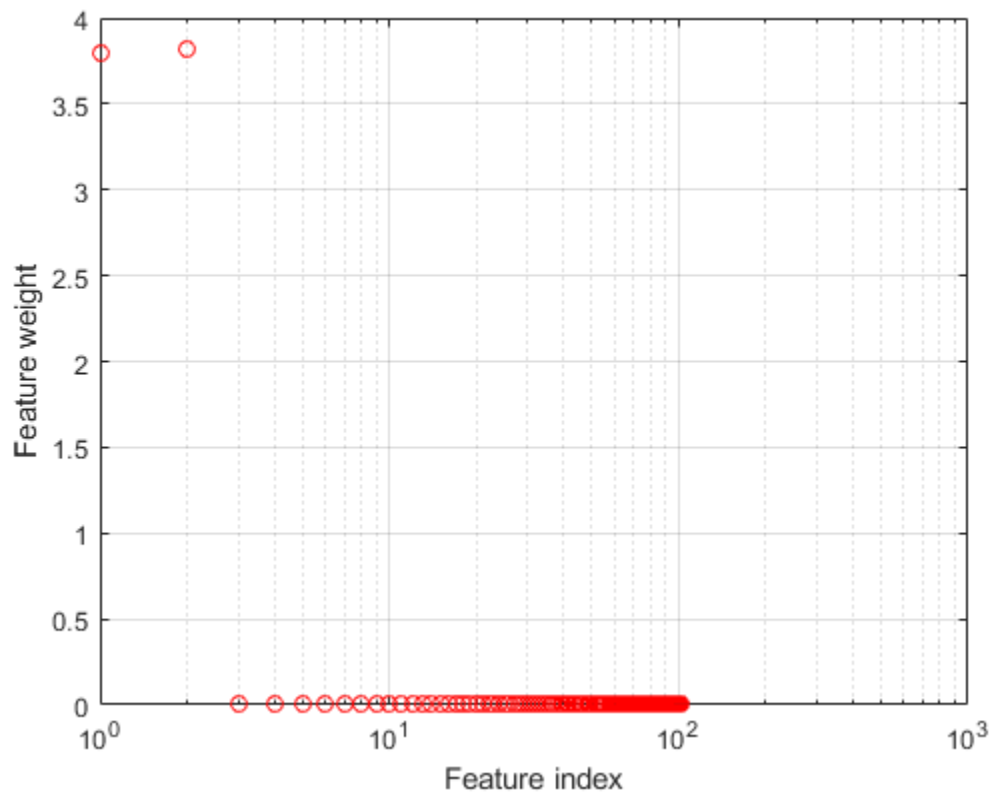
```
Relative infinity norm of the final gradient = 5.764e-07, TolFun = 1.000e-06
```

```
EXIT: Local minimum found.
```

Plot the feature weights.

```
figure
semilogx(ncaMdl.FeatureWeights,'ro')
xlabel('Feature index')
```

```
ylabel('Feature weight')
grid on
```



`fscnca` correctly figures out that the first two features are relevant and that the rest are not. The first two features are not individually informative, but when taken together result in an accurate classification model.

Predict the classes using the new model and compute the accuracy.

```
ypred = predict(ncaMdl,X);
confusionmat(y,ypred)
```

```
ans =
```

88	12
8	92

Confusion matrix shows that prediction accuracy for class 1 has improved. 88 of the data from class 1 are predicted to be from 1, and 12 of them are predicted to be from class 2. 92 of the data from class 2 are predicted to be from class 2 and 8 of them are predicted to be from class 1.

### References

1. Yang, W., K. Wang, W. Zuo. "Neighborhood Component Feature Selection for High-Dimensional Data." *Journal of Computers*. Vol. 7, Number 1, January, 2012.

### See Also

`FeatureSelectionNCAClassification` | `fscnca` | `predict` | `refit`

**Introduced in R2016b**

# loss

**Class:** FeatureSelectionNCARegression

Evaluate accuracy of learned feature weights on test data

## Syntax

```
err = loss mdl, X, Y
err = loss mdl, X, Y, Name, Value
```

## Description

`err = loss mdl, X, Y` returns the mean squared error as the measure of accuracy in `err`, for the model `mdl`, predictor values in `X`, and response values in `Y`.

`err = loss mdl, X, Y, Name, Value` returns the measure of accuracy, `err`, with the additional option specified by the `Name, Value` pair argument.

## Input Arguments

**mdl** — Neighborhood component analysis model for regression

FeatureSelectionNCARegression object

Neighborhood component analysis model for regression, specified as a FeatureSelectionNCARegression object.

**x** — Predictor variable values

*n*-by-*p* matrix

Predictor variable values, specified as an *n*-by-*p* matrix, where *n* is the number of observations and *p* is the number of predictor variables.

Data Types: single | double

**$y$  — Response values**numeric real vector of length  $n$ 

Response values, specified as a numeric real vector of length  $n$ , where  $n$  is the number of observations.

Data Types: `single` | `double`**Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

**`LossFunction` — Loss function type**

'mse' (default) | 'mad'

Loss function type, specified as a comma-separated pair consisting of '`LossFunction`' and one of the following.

Loss Function Type	Description
'mse'	Mean squared error
'mad'	Mean absolute deviation

Example: '`LossFunction`', 'mse'Data Types: `char`**Output Arguments****`err` — Smaller-the-better accuracy measure for learned feature weights**

scalar value

Smaller-the-better accuracy measure for learned feature weights, returned as a scalar value. You can specify the measure of accuracy using the `LossFunction` name-value pair argument.



## Examples

### Tune NCA Model for Regression Using `loss` and `predict`

#### Load the sample data.

Download the housing data [1], from the UCI Machine Learning Repository [2]. The dataset has 506 observations. The first 13 columns contain the predictor values and the last column contains the response values. The goal is to predict the median value of owner-occupied homes in suburban Boston as a function of 13 predictors.

Load the data and define the response vector and the predictor matrix.

```
load('housing.data');
X = housing(:,1:13);
y = housing(:,end);
```

Divide the data into training and test sets using the 4th predictor as the grouping variable for a stratified partitioning. This ensures that each partition includes similar amount of observations from each group.

```
rng(1) % For reproducibility
cvp = cvpartition(X(:,4), 'Holdout', 56);
Xtrain = X(cvp.training, :);
ytrain = y(cvp.training, :);
Xtest = X(cvp.test, :);
ytest = y(cvp.test, :);
```

`cvpartition` randomly assigns 56 observations into a test set and the rest of the data into a training set.

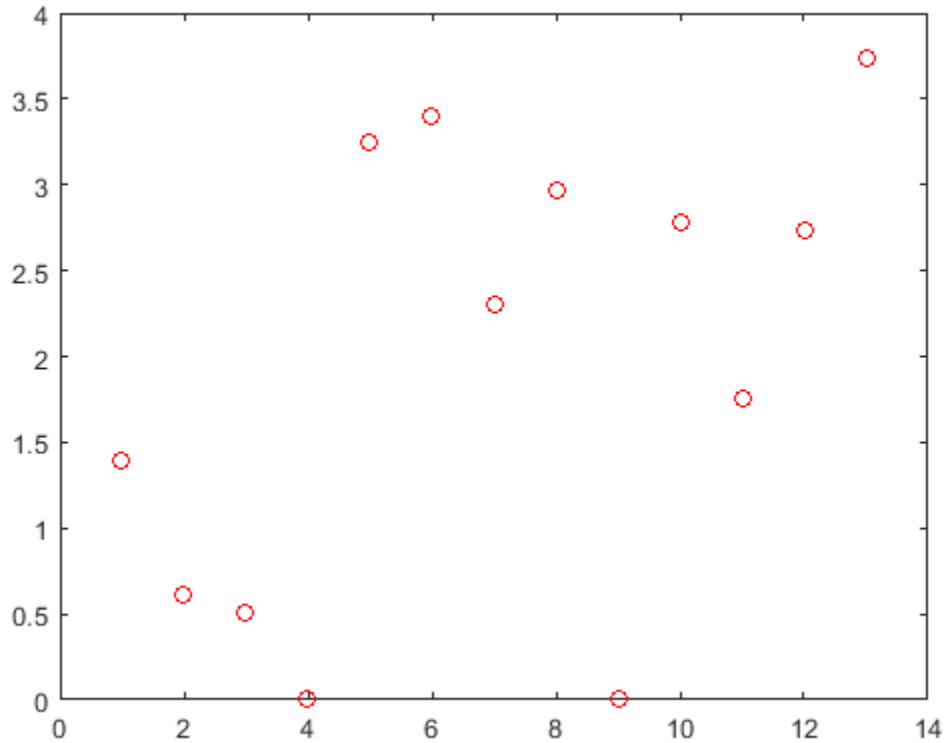
#### Perform Feature Selection Using Default Settings

Perform feature selection using NCA model for regression. Standardize the predictor values.

```
nca = fsrnca(Xtrain, ytrain, 'Standardize', 1);
```

Plot the feature weights.

```
figure()
plot(nca.FeatureWeights, 'ro')
```



The weights of irrelevant features are expected to approach zero. `fsrnca` identifies two features as irrelevant.

Compute the regression loss.

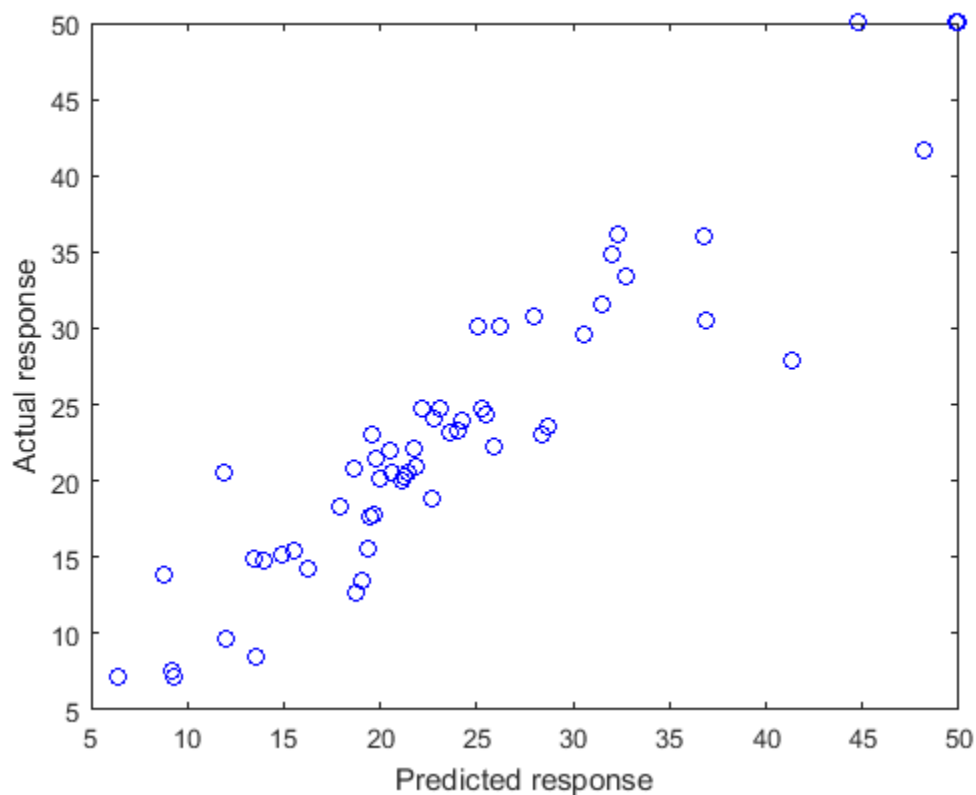
```
L = loss(nca,Xtest,ytest,'LossFunction','mad')
```

```
L =
```

```
2.5394
```

Compute the predicted response values for the test set and plot them versus the actual response.

```
y_pred = predict(nca, X_test);  
figure()  
plot(y_pred, y_test, 'bo')  
xlabel('Predicted response')  
ylabel('Actual response')
```



A perfect fit versus the actual values forms a 45 degree straight line. In this plot, the predicted and actual response values seem to be scattered around this line. Tuning  $\lambda$  (regularization parameter) value usually helps improve the performance.

### Tune the regularization parameter using 10-fold cross-validation

Tuning  $\lambda$  means finding the  $\lambda$  value that will produce the minimum regression loss. Here are the steps for tuning  $\lambda$  using 10-fold cross-validation:

1. First partition the data into 10 folds. For each fold, `cvpartition` assigns 1/10th of the data as a training set, and 9/10th of the data as a test set.

```
n = length(ytrain);  
cvp = cvpartition(Xtrain(:,4), 'kfold', 10);  
numvalidsets = cvp.NumTestSets;
```

Assign the  $\lambda$  values for the search. Create an array to store the loss values.

```
lambdaval = linspace(0, 2, 30) * std(ytrain) / n;  
lossvals = zeros(length(lambdaval), numvalidsets);
```

2. Train the neighborhood component analysis (nca) model for each  $\lambda$  value using the training set in each fold.

3. Fit a Gaussian process regression (gpr) model using the selected features. Next, compute the regression loss for the corresponding test set in the fold using the gpr model. Record the loss value.

4. Repeat this for each  $\lambda$  value and each fold.

```
for i = 1:length(lambdaval)  
    for k = 1:numvalidsets  
        X = Xtrain(cvp.training(k), :);  
        y = ytrain(cvp.training(k), :);  
        Xvalid = Xtrain(cvp.test(k), :);  
        yvalid = ytrain(cvp.test(k), :);  
  
        nca = fsrnca(X, y, 'FitMethod', 'exact', ...  
                   'Lambda', lambdaval(i), ...  
                   'Standardize', 1, 'LossFunction', 'mad');  
  
        % Select features using the feature weights and a relative  
        % threshold.  
        tol = 1e-3;  
        selidx = nca.FeatureWeights > tol * max(1, max(nca.FeatureWeights));  
  
        % Fit a non-ARD GPR model using selected features.  
        gpr = fitrgp(X(:, selidx), y, 'Standardize', 1, ...
```

---

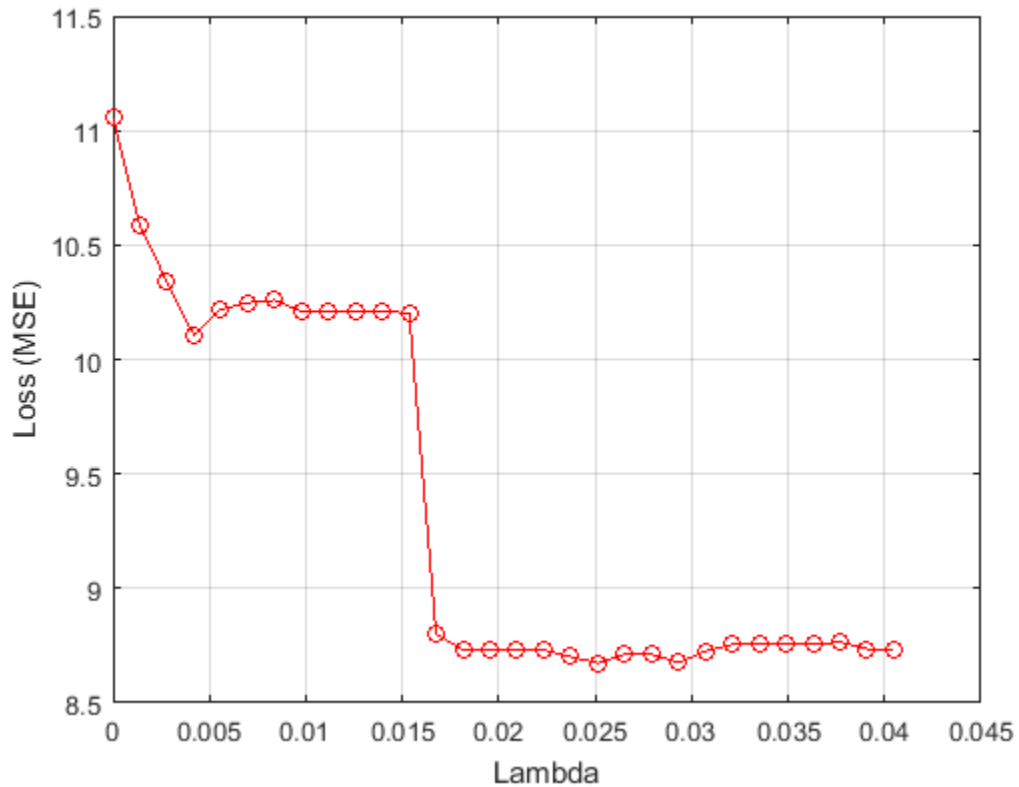
```
        'KernelFunction','squareexponential','Verbose',0);

        lossvals(i,k) = loss(gpr,Xvalid(:,selidx),yvalid);

    end
end
```

Compute the average loss obtained from the folds for each  $\lambda$  value. Plot the mean loss versus the  $\lambda$  values.

```
meanloss = mean(lossvals,2);
figure;
plot(lambdaval,meanloss,'ro-');
xlabel('Lambda');
ylabel('Loss (MSE)');
grid on;
```



Find the  $\lambda$  value that produces the minimum loss value.

```
[~,idx] = min(meanloss);  
bestlambda = lambdaval(idx)
```

```
bestlambda =
```

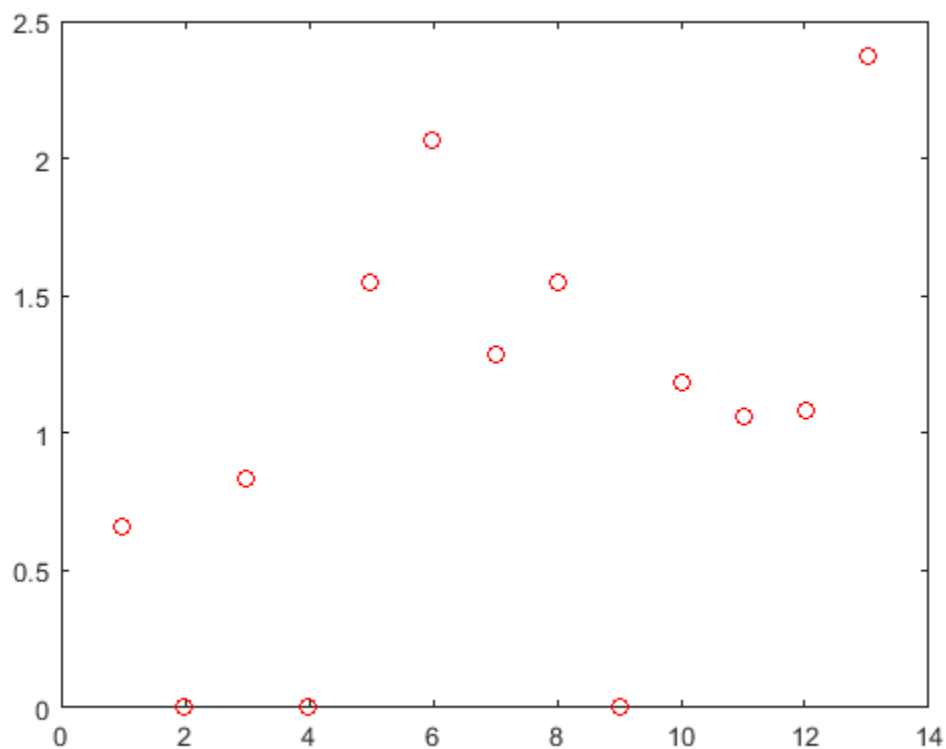
```
0.0251
```

Perform feature selection for regression using the best  $\lambda$  value. Standardize the predictor values.

```
nca2 = fsrnca(Xtrain,ytrain,'Standardize',1,'Lambda',bestlambda,...  
             'LossFunction','mad');
```

Plot the feature weights.

```
figure()  
plot(nca.FeatureWeights,'ro')
```



Compute the loss using the new nca model on the test data, which is not used to select the features.

```
L2 = loss(nca2,Xtest,ytest,'LossFunction','mad')
```

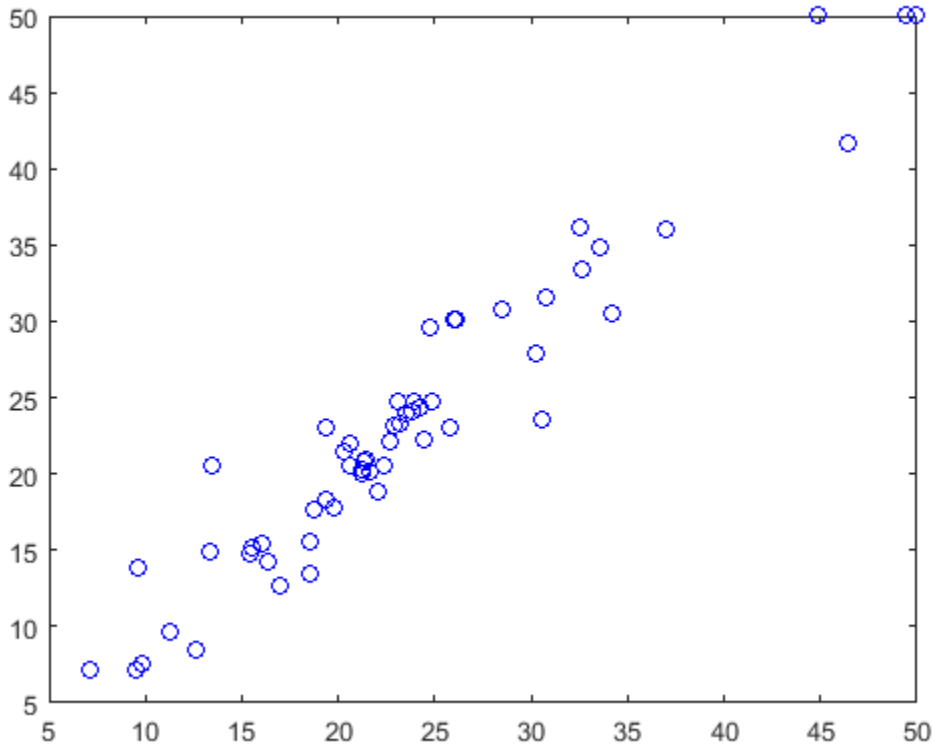
```
L2 =
```

```
2.0560
```

Tuning the regularization parameter helps identify the relevant features and reduces the loss.

Plot the predicted versus the actual response values in the test set.

```
ypred = predict(nca2,Xtest);  
figure;  
plot(ypred,ytest,'bo');
```



The predicted response values seem to be closer to the actual values as well.



## References

[1] Harrison, D. and D.L., Rubinfeld. "Hedonic prices and the demand for clean air." J. Environ. Economics & Management. Vol.5, 1978, pp. 81-102.

[2] Lichman, M. UCI Machine Learning Repository, Irvine, CA: University of California, School of Information and Computer Science, 2013. <http://archive.ics.uci.edu/ml>.

## See Also

`FeatureSelectionNCARegression` | `fsrnca` | `predict` | `refit`

**Introduced in R2016b**

## loss

**Class:** RegressionLinear

Regression loss for linear regression models

## Syntax

```
L = loss(Mdl, X, Y)
```

```
L = loss( ____, Name, Value)
```

## Description

`L = loss(Mdl, X, Y)` returns the mean squared error (MSE) for the linear regression model `Mdl` using predictor data in `X` and corresponding responses in `Y`. `L` contains an MSE for each regularization strength in `Mdl`.

`L = loss( ____, Name, Value)` uses any of the previous syntaxes and additional options specified by one or more `Name, Value` pair arguments. For example, specify that columns in the predictor data correspond to observations or specify the regression loss function.

## Input Arguments

**mdl** — Linear regression model

RegressionLinear model object

Linear regression model, specified as a RegressionLinear model object. You can create a RegressionLinear model object using `fitrlinear`.

**x** — Predictor data

full matrix | sparse matrix

Predictor data, specified as an  $n$ -by- $p$  full or sparse matrix. This orientation of `X` indicates that rows correspond to individual observations, and columns correspond to individual predictor variables.

---

**Note** If you orient your predictor matrix so that observations correspond to columns and specify 'ObservationsIn', 'columns', then you might experience a significant reduction in computation time.

---

The length of  $Y$  and the number of observations in  $X$  must be equal.

Data Types: `single` | `double`

### **$y$ — Response data**

numeric vector

Response data, specified as an  $n$ -dimensional numeric vector. The length of  $Y$  and the number of observations in  $X$  must be equal.

Data Types: `single` | `double`

## **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1`, `Value1`, ..., `NameN`, `ValueN`.

### **LossFun — Loss function**

'mse' (default) | 'epsiloninsensitive' | function handle

Loss function, specified as the comma-separated pair consisting of 'LossFun' and a built-in, loss-function name or function handle.

- The following lists available loss functions. Specify one using its corresponding character vector. Also, in the table,  $f(x) = x\beta + b$ .
  - $\beta$  is a vector of  $p$  coefficients.
  - $x$  is an observation from  $p$  predictor variables.
  - $b$  is the scalar bias.

Value	Description
'epsiloninsensitive'	Epsilon-insensitive loss: $\ell[y, f(x)] = \max[0,  y - f(x)  - \epsilon]$
'mse'	MSE: $\ell[y, f(x)] = [y - f(x)]^2$

'epsiloninsensitive' is appropriate for SVM learners only.

- Specify your own function using function handle notation.

Let  $n$  be the number of observations in  $X$ . Your function must have this signature

```
lossvalue = lossfun(Y, Yhat, W)
```

where:

- The output argument `lossvalue` is a scalar.
- You choose the function name (`lossfun`).
- $Y$  is an  $n$ -dimensional vector of observed responses. `RegressionLinear.loss` passes the input argument  $Y$  in for  $Y$ .
- $Yhat$  is an  $n$ -dimensional vector of predicted responses, which is similar to the output of `predict`.
- $W$  is an  $n$ -by-1 numeric vector of observation weights.

Specify your function using `'LossFun', @lossfun`.

Data Types: `char` | `function_handle`

### ObservationsIn — Predictor data observation dimension

'rows' (default) | 'columns'

Predictor data observation dimension, specified as the comma-separated pair consisting of 'ObservationsIn' and 'columns' or 'rows'.

---

**Note** If you orient your predictor matrix so that observations correspond to columns and specify 'ObservationsIn', 'columns', then you might experience a significant reduction in optimization-execution time.

---

Data Types: `char`

### **Weights** — Observation weights

numeric vector of positive values

Observation weights, specified as the comma-separated pair consisting of `'Weights'` and a numeric vector of positive values. If you supply `weights`, `loss` computes the weighted classification loss.

Let  $n$  be the number of observations in  $X$ .

- `numel(Weights)` must be  $n$ .
- By default, `Weights` is `ones(n,1)`.

Data Types: `double` | `single`

## Output Arguments

### **L** — Regression losses

numeric scalar | numeric row vector

Regression losses, returned as a numeric scalar or row vector. The interpretation of `L` depends on `Weights` and `LossFun`.

`L` is the same size as `Mdl.Lambda`. `L(j)` is the regression loss of the linear regression model trained using the regularization strength `Mdl.Lambda(j)`.

---

**Note** If `Mdl.FittedLoss` is `'mse'`, then the loss term in the objective function is half of the MSE. `loss` returns the MSE by default. Therefore, if you use `loss` to check the resubstitution (training) error, then there is a discrepancy between the MSE and optimization results that `fitrlinear` returns.

---

## Examples

### Estimate Test-Sample Mean Squared Error

Simulate 10000 observations from this model

$$y = x_{100} + 2x_{200} + e.$$

- $X = x_1, \dots, x_{1000}$  is a 10000-by-1000 sparse matrix with 10% nonzero standard normal elements.
- $e$  is random normal error with mean 0 and standard deviation 0.3.

```
rng(1) % For reproducibility
n = 1e4;
d = 1e3;
nz = 0.1;
X = sprandn(n,d,nz);
Y = X(:,100) + 2*X(:,200) + 0.3*randn(n,1);
```

Train a linear regression model. Reserve 30% of the observations as a holdout sample.

```
CVMDL = fitrlinear(X,Y,'Holdout',0.3);
Mdl = CVMDL.Trained{1}
```

```
Mdl =
    RegressionLinear
        ResponseName: 'Y'
    ResponseTransform: 'none'
            Beta: [1000x1 double]
            Bias: -0.0066
            Lambda: 1.4286e-04
        Learner: 'svm'
```

CVMDL is a `RegressionPartitionedLinear` model. It contains the property `Trained`, which is a 1-by-1 cell array holding a `RegressionLinear` model that the software trained using the training set.

Extract the training and test data from the partition definition.

```
trainIdx = training(CVMDL.Partition);
testIdx = test(CVMDL.Partition);
```

Estimate the training- and test-sample MSE.

```
mseTrain = loss(Mdl,X(trainIdx,:),Y(trainIdx))
mseTest = loss(Mdl,X(testIdx,:),Y(testIdx))
```

```
mseTrain =
    0.1496
```

```
mseTest =
    0.1799
```

Because there is one regularization strength in `Mdl`, `mseTrain` and `mseTest` are numeric scalars.

### Specify Custom Regression Loss

Simulate 10000 observations from this model

$$y = x_{100} + 2x_{200} + e.$$

- $X = x_1, \dots, x_{1000}$  is a 10000-by-1000 sparse matrix with 10% nonzero standard normal elements.
- $e$  is random normal error with mean 0 and standard deviation 0.3.

```
rng(1) % For reproducibility
n = 1e4;
d = 1e3;
nz = 0.1;
X = sprandn(n,d,nz);
Y = X(:,100) + 2*X(:,200) + 0.3*randn(n,1);
X = X'; % Put observations in columns for faster training
```

Train a linear regression model. Reserve 30% of the observations as a holdout sample.

```
CVMdl = fitrlinear(X,Y,'Holdout',0.3,'ObservationsIn','columns');
Mdl = CVMdl.Trained{1}
```

```
Mdl =
    RegressionLinear
        ResponseName: 'Y'
        ResponseTransform: 'none'
```

```

Beta: [1000x1 double]
Bias: -0.0066
Lambda: 1.4286e-04
Learner: 'svm'

```

CVMdl is a `RegressionPartitionedLinear` model. It contains the property `Trained`, which is a 1-by-1 cell array holding a `RegressionLinear` model that the software trained using the training set.

Extract the training and test data from the partition definition.

```

trainIdx = training(CVMdl.Partition);
testIdx = test(CVMdl.Partition);

```

Create an anonymous function that measures Huber loss ( $\delta = 1$ ), that is,

$$L = \frac{1}{\sum w_j} \sum_{j=1}^n w_j \ell_j, \text{ where}$$

$$\ell_j = \begin{cases} 0.5\hat{e}^2 & \text{for } |\hat{e}| \leq 1 \\ |\hat{e}| - 0.5 & \text{otherwise} \end{cases} .$$

$\hat{e}_j$  is the residual for observation  $j$ . Custom loss functions must be written in a particular form. For rules on writing a custom loss function, see the `LossFun` name-value pair argument.

```

huberloss = @(Y,Yhat,W) sum(W.*((0.5*(abs(Y-Yhat)<=1).*(Y-Yhat).^2) + ...
    (abs(Y-Yhat)>1).*abs(Y-Yhat)-0.5))/sum(W);

```

Estimate the training- and test-sample regression loss using the Huber loss function.

```

eTrain = loss(Mdl,X(:,trainIdx),Y(trainIdx),'LossFun',huberloss,...
    'ObservationsIn','columns')
eTest = loss(Mdl,X(:,testIdx),Y(testIdx),'LossFun',huberloss,...
    'ObservationsIn','columns')

```

```

eTrain =

    -0.4186

```

```

eTest =

```



```
-0.4012
```

## Find Good Lasso Penalty Using Regression Loss

Simulate 10000 observations from this model

$$y = x_{100} + 2x_{200} + e.$$

- $X = \{x_1, \dots, x_{1000}\}$  is a 10000-by-1000 sparse matrix with 10% nonzero standard normal elements.
- $e$  is random normal error with mean 0 and standard deviation 0.3.

```
rng(1) % For reproducibility
n = 1e4;
d = 1e3;
nz = 0.1;
X = sprandn(n,d,nz);
Y = X(:,100) + 2*X(:,200) + 0.3*randn(n,1);
```

Create a set of 15 logarithmically-spaced regularization strengths from  $10^{-4}$  through  $10^{-1}$ .

```
Lambda = logspace(-4,-1,15);
```

Hold out 30% of the data for testing. Identify the test-sample indices.

```
cvp = cvpartition(numel(Y), 'Holdout', 0.30);
idxTest = test(cvp);
```

Train a linear regression model using lasso penalties with the strengths in `Lambda`. Specify the regularization strengths, optimizing the objective function using `SpaRSA`, and the data partition. To increase execution speed, transpose the predictor data and specify that the observations are in columns.

```
X = X';
CVMdl = fitrlinear(X,Y,'ObservationsIn','columns','Lambda',Lambda,...
    'Solver','sparsa','Regularization','lasso','CVPartition',cvp);
Mdl1 = CVMdl.Trained{1};
numel(Mdl1.Lambda)
```

```
ans =  
  
    15
```

Mdl1 is a `RegressionLinear` model. Because `Lambda` is a 15-dimensional vector of regularization strengths, you can think of Mdl1 as 15 trained models, one for each regularization strength.

Estimate the test-sample mean squared error for each regularized model.

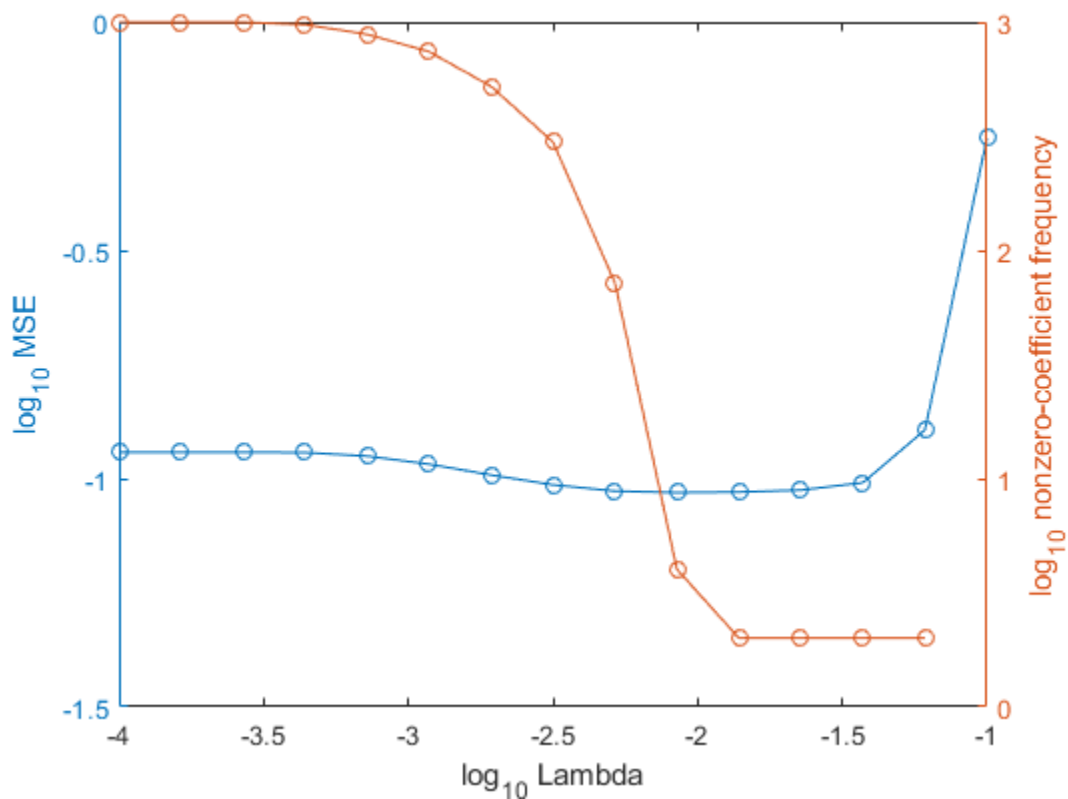
```
mse = loss(Mdl1,X(:,idxTest),Y(idxTest),'ObservationsIn','columns');
```

Higher values of `Lambda` lead to predictor variable sparsity, which is a good quality of a regression model. Retrain the model using the entire data set and all options used previously, except the data-partition specification. Determine the number of nonzero coefficients per model.

```
Mdl = fitrlinear(X,Y,'ObservationsIn','columns','Lambda',Lambda,...  
    'Solver','sparsa','Regularization','lasso');  
numNZCoeff = sum(Mdl.Beta~=0);
```

In the same figure, plot the MSE and frequency of nonzero coefficients for each regularization strength. Plot all variables on the log scale.

```
figure;  
[h,hL1,hL2] = plotyy(log10(Lambda),log10(mse),...  
    log10(Lambda),log10(numNZCoeff));  
hL1.Marker = 'o';  
hL2.Marker = 'o';  
ylabel(h(1),'log_{10} MSE')  
ylabel(h(2),'log_{10} nonzero-coefficient frequency')  
xlabel('log_{10} Lambda')  
hold off
```



Select the index or indices of Lambda that balance minimal classification error and predictor-variable sparsity (for example, Lambda (11)).

```
idx = 11;  
MdlFinal = selectModels(Mdl,idx);
```

`MdlFinal` is a trained `RegressionLinear` model object that uses `Lambda(11)` as a regularization strength.

## Extended Capabilities

### Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays” (MATLAB).

### See Also

`RegressionLinear` | `fitrlinear` | `predict`

**Introduced in R2016a**

# lowerparams

**Class:** paretotails

Lower Pareto tails parameters

## Syntax

```
params = lowerparams(obj)
```

## Description

`params = lowerparams(obj)` returns the 2-element vector `params` of shape and scale parameters, respectively, of the lower tail of the Pareto tails object `obj`. `lowerparams` does not return a location parameter.

## Examples

Fit Pareto tails to a  $t$  distribution at cumulative probabilities 0.1 and 0.9:

```
t = trnd(3,100,1);  
obj = paretotails(t,0.1,0.9);
```

```
lowerparams(obj)  
ans =  
    -0.1901    1.1898  
upperparams(obj)  
ans =  
    0.3646    0.5103
```

## See Also

`paretotails` | `upperparams`

## lt

**Class:** grandstream

Less than relation for handles

## Syntax

`h1 < h2`

## Description

`h1 < h2` performs element-wise comparisons between handle arrays `h1` and `h2`. `h1` and `h2` must be of the same dimensions unless one is a scalar. The result is a logical array of the same dimensions, where each element is an element-wise `<` result.

If one of `h1` or `h2` is scalar, scalar expansion is performed and the result will match the dimensions of the array that is not scalar.

`tf = lt(h1, h2)` stores the result in a logical array of the same dimensions.

## See Also

`eq` | `ge` | `gt` | `le` | `ne` | `grandstream`

# lsline

Add least-squares line to scatter plot

## Syntax

```
lsline  
lsline(ax)  
h = lsline(____)
```

## Description

`lsline` superimposes a least-squares line on each scatter plot in the current axes. Scatter plots are produced by the MATLAB `scatter` and `plot` functions. Data points connected with solid, dashed, or dash-dot lines ('-', '--', or '-.') are not considered to be scatter plots by `lsline`, and are ignored.

`lsline(ax)` superimposes a least-squares line on the scatter plot in axis `ax`.

`h = lsline(____)` returns a column vector of handles `h` to the least-squares lines, using any of the previous syntaxes.

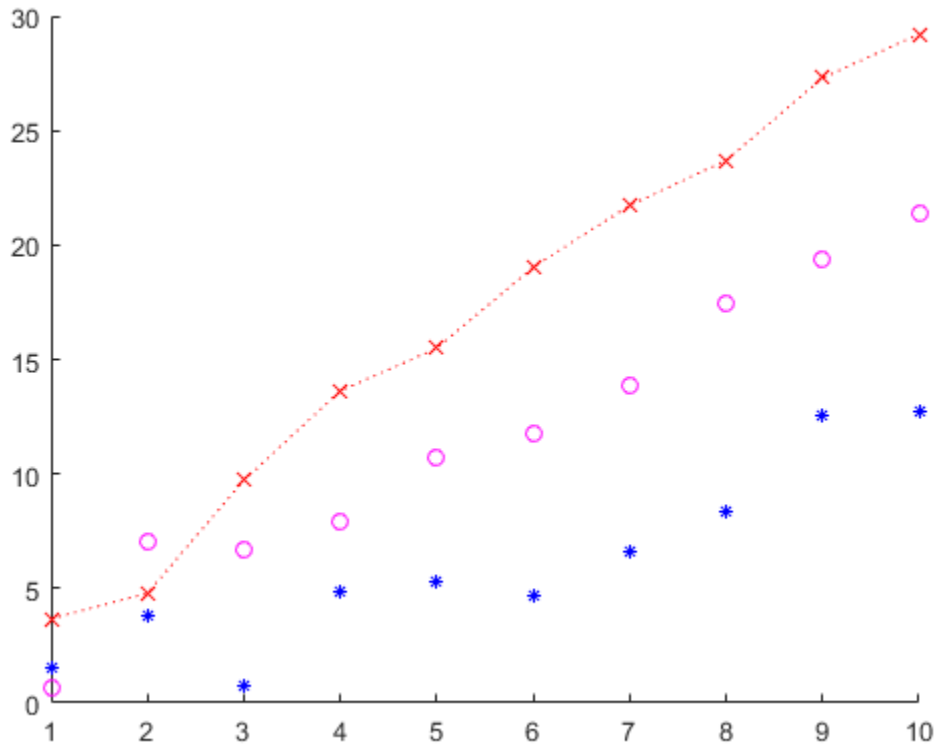
## Examples

### Plot a Least-Squares Line

Generate three sets of sample data and plot on the same figure.

```
x = 1:10;  
rng default; % For reproducibility  
figure;  
  
y1 = x + randn(1,10);  
scatter(x,y1,25,'b','*')  
hold on
```

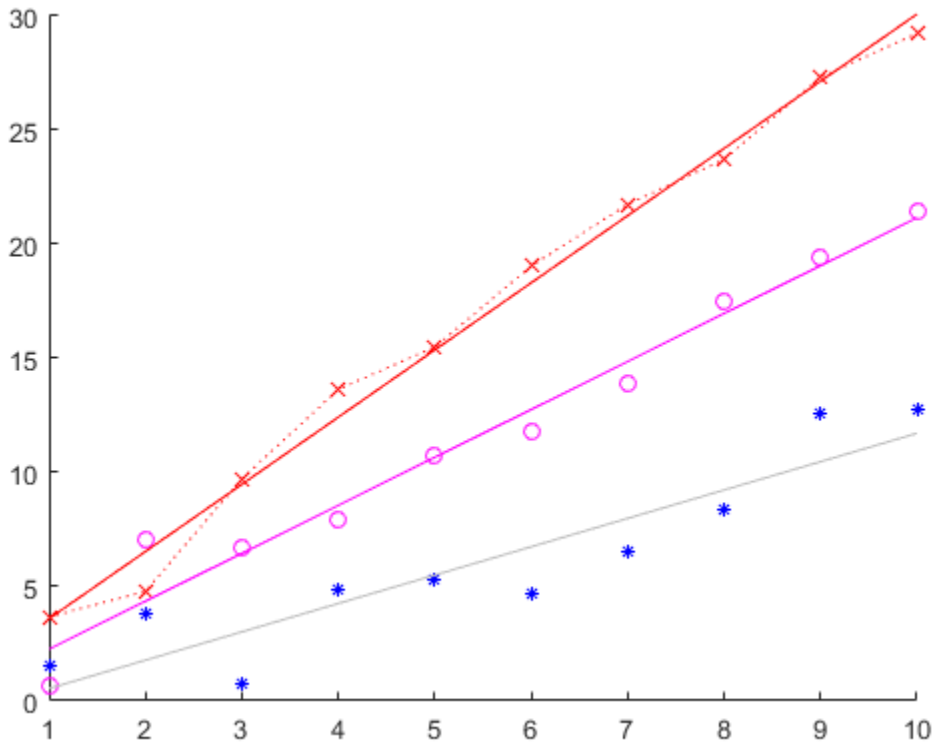
```
y2 = 2*x + randn(1,10);  
plot(x,y2, 'mo')  
  
y3 = 3*x + randn(1,10);  
plot(x,y3, 'rx:')
```



Add a least-squares line for each set of sample data.

```
lsline
```





### Specify Axes for Least-Squares and Reference Lines

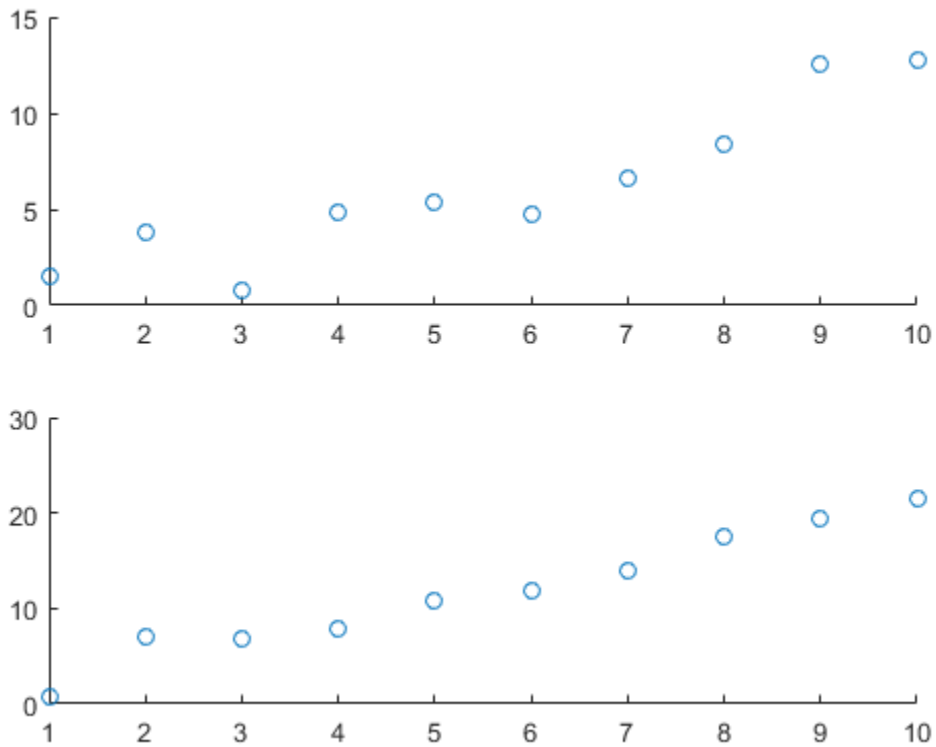
Define the x-variable and two different y-variables to use for the plots.

```
rng default % For reproducibility
x = 1:10;
y1 = x + randn(1,10);
y2 = 2*x + randn(1,10);
```

Define `ax1` as the top half of the figure, and `ax2` as the bottom half of the figure. Create the first scatter plot on the top axis using `y1`, and the second scatter plot on the bottom axis using `y2`.

```
figure
ax1 = subplot(2,1,1);
ax2 = subplot(2,1,2);

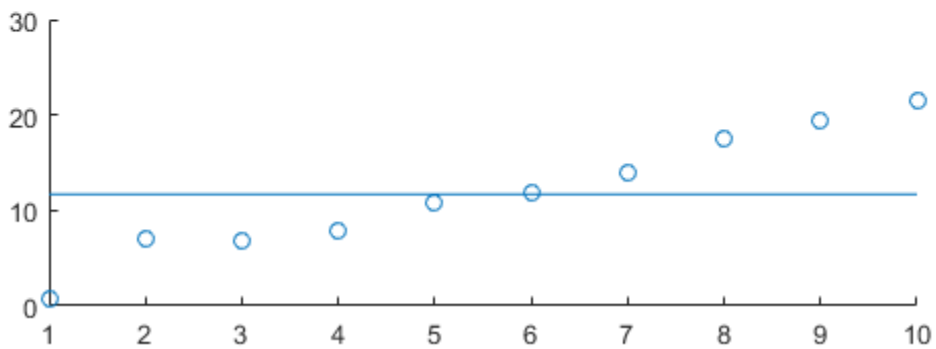
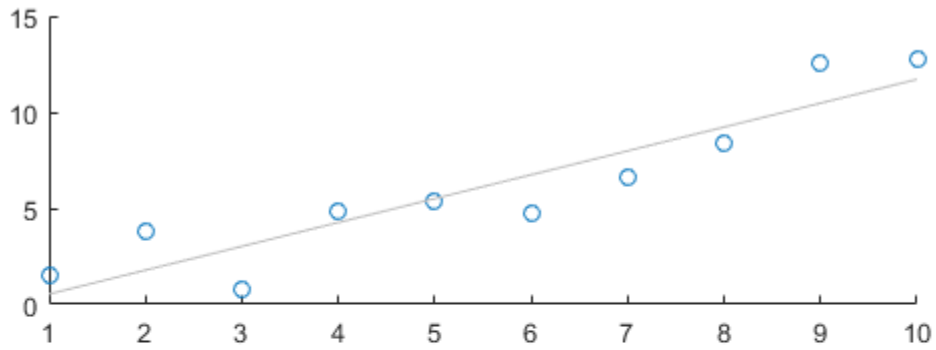
scatter(ax1,x,y1)
scatter(ax2,x,y2)
```



Superimpose a least-squares line on the top plot, and a reference line at the mean of the  $y_2$  values in the bottom plot.

```
lsline(ax1)

mu = mean(y2);
refline(ax2,[0 mu])
```



## See Also

`gline` | `plot` | `refcurve` | `refline` | `scatter`

Introduced before R2006a

## mad

Mean or median absolute deviation

### Syntax

```
y = mad(X)
y = mad(X, 1)
y = mad(X, 0)
y = mad(X, flag, dim)
```

### Description

`y = mad(X)` returns the mean absolute deviation of the values in `X`. For vector input, `y` is `mean(abs(X-mean(X)))`. For a matrix input, `y` is a row vector containing the mean absolute deviation of each column of `X`. For  $N$ -dimensional arrays, `mad` operates along the first nonsingleton dimension of `X`.

`y = mad(X, 1)` returns the median absolute deviation of the values in `X`. For vector input, `y` is `median(abs(X-median(X)))`. For a matrix input, `y` is a row vector containing the median absolute deviation of each column of `X`. For  $N$ -dimensional arrays, `mad` operates along the first nonsingleton dimension of `X`.

`y = mad(X, 0)` is the same as `mad(X)`, and returns the mean absolute deviation of the values in `X`.

`y = mad(X, flag, dim)` computes absolute deviations along the dimension `dim` of `X`. `flag` is 0 or 1 to indicate mean or median absolute deviation, respectively.

`mad` treats NaNs as missing values and removes them.

For normally distributed data, multiply `mad` by one of the following factors to obtain an estimate of the normal scale parameter  $\sigma$ :

- `sigma = 1.253*mad(X, 0)` — For mean absolute deviation

- `sigma = 1.4826*mad(X,1)` — For median absolute deviation

## Examples

The following compares the robustness of different scale estimates for normally distributed data in the presence of outliers:

```
x = normrnd(0,1,1,50);  
xo = [x 10]; % Add outlier
```

```
r1 = std(xo)/std(x)  
r1 =  
    1.7385
```

```
r2 = mad(xo,0)/mad(x,0)  
r2 =  
    1.2306
```

```
r3 = mad(xo,1)/mad(x,1)  
r3 =  
    1.0602
```

## References

- [1] Mosteller, F., and J. Tukey. *Data Analysis and Regression*. Upper Saddle River, NJ: Addison-Wesley, 1977.
- [2] Sachs, L. *Applied Statistics: A Handbook of Techniques*. New York: Springer-Verlag, 1984, p. 253.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Empty dim input is not supported.

## See Also

`iqr` | `range` | `std`

**Introduced before R2006a**

# mahal

Mahalanobis distance

## Syntax

```
d = mahal(Y,X)
```

## Description

`d = mahal(Y,X)` computes the Mahalanobis distance (in squared units) of each observation in `Y` from the reference sample in matrix `X`. If `Y` is  $n$ -by- $m$ , where  $n$  is the number of observations and  $m$  is the dimension of the data, `d` is  $n$ -by-1. `X` and `Y` must have the same number of columns, but can have different numbers of rows. `X` must have more rows than columns.

For observation `I`, the Mahalanobis distance is defined by  $d(I) = (Y(I, :) - \mu) * \text{inv}(\text{SIGMA}) * (Y(I, :) - \mu)'$ , where `mu` and `SIGMA` are the sample mean and covariance of the data in `X`. `mahal` performs an equivalent, but more efficient, computation.

## Examples

### Compare Mahalanobis and Squared Euclidean Distances

Generate correlated bivariate data.

```
X = mvnrnd([0;0],[1 .9;.9 1],100);
```

Input observations.

```
Y = [1 1;1 -1;-1 1;-1 -1];
```

Compute the Mahalanobis distance of observations in `Y` from the reference sample in `X`.

```
d1 = mahal(Y,X)
```

```
d1 =
```

```
    0.6288  
   19.3520  
   21.1384  
    0.9404
```

Compute their squared Euclidean distances from the mean of X .

```
d2 = sum((Y-repmat(mean(X),4,1)).^2, 2)
```

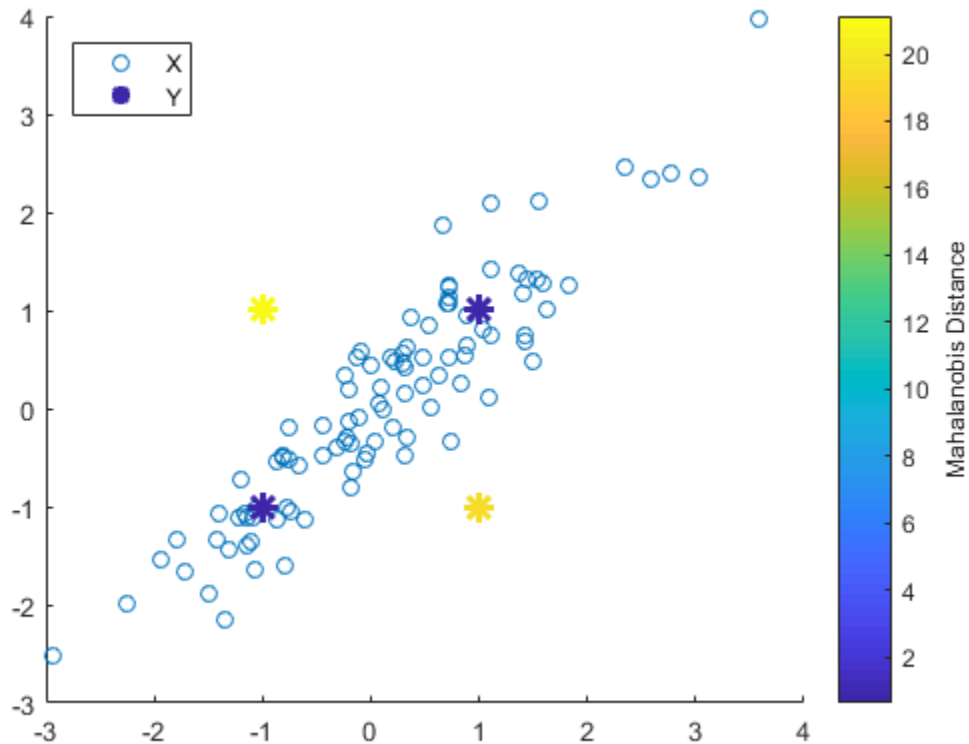
```
d2 =
```

```
    1.6170  
    1.9334  
    2.1094  
    2.4258
```

Plot the observations with Y values colored according to the Mahalanobis distance.

```
scatter(X(:,1),X(:,2))  
hold on  
scatter(Y(:,1),Y(:,2),100,d1,'*','LineWidth',2)  
hb = colorbar;  
ylabel(hb,'Mahalanobis Distance')  
legend('X','Y','Location','NW')
```





The observations in Y with equal coordinate values are much closer to X in Mahalanobis distance than observations with opposite coordinate values, even though all observations are approximately equidistant from the mean of X in Euclidean distance. The Mahalanobis distance, by considering the covariance of the data and the scales of the different variables, is useful for detecting outliers in such cases.

## See Also

`mahal` | `pdist`

Introduced before R2006a

## mahal

**Class:** CompactClassificationDiscriminant

Mahalanobis distance to class means

## Syntax

`M = mahal(obj, X)`

`M = mahal(obj, X, Name, Value)`

## Description

`M = mahal(obj, X)` returns the squared Mahalanobis distances from observations in `X` to the class means in `obj`.

`M = mahal(obj, X, Name, Value)` computes the squared Mahalanobis distance with additional options specified by one or more `Name, Value` pair arguments.

## Input Arguments

**obj**

Discriminant analysis classifier of class `ClassificationDiscriminant` or `CompactClassificationDiscriminant`, typically constructed with `fitcdiscr`.

**X**

Numeric matrix of size `n-by-p`, where `p` is the number of predictors in `obj`, and `n` is any positive integer. `mahal` computes the Mahalanobis distances from the rows of `X` to each of the `K` means of the classes in `obj`.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single

quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### **ClassLabels**

Class labels consisting of  $n$  elements of `obj.Y`, where  $n$  is the number of rows of `X`.

## Output Arguments

### **M**

Size and meaning of output `M` depends on whether the `ClassLabels` name-value pair is present:

- No `ClassLabels` — `M` is a numeric matrix of size  $n$ -by- $K$ , where  $K$  is the number of classes in `obj`, and  $n$  is the number of rows in `X`. `M(i, j)` is the squared Mahalanobis distance from the  $i$ th row of `X` to the mean of class  $j$ .
- `ClassLabels` exists — `M` is a column vector with  $n$  elements. `M(i)` is the squared Mahalanobis distance from the  $i$ th row of `X` to the mean for the class of the  $i$ th element of `ClassLabels`.

## Examples

Find the Mahalanobis distances from the mean of the Fisher iris data to the class means, using distinct covariance matrices for each class:

```
load fisheriris
obj = fitcdiscr(meas, species, ...
    'DiscrimType', 'quadratic');
mahadist = mahal(obj, mean(meas))
```

```
mahadist =
    220.0667     5.0254    30.5804
```

## Definitions

### Mahalanobis Distance

The Mahalanobis distance  $d(x,y)$  between  $n$ -dimensional points  $x$  and  $y$ , with respect to a given  $n$ -by- $n$  covariance matrix  $S$ , is

$$d(x, y) = \sqrt{(x - y)^T S^{-1} (x - y)}.$$

### See Also

`CompactClassificationDiscriminant` | `fitcdiscr` | `gmdistribution` | `mahal`

### Topics

“Discriminant Analysis Classification” on page 20-2

# mahal

**Class:** gmdistribution

Mahalanobis distance to component means

## Syntax

```
D = mahal(obj,X)
```

## Description

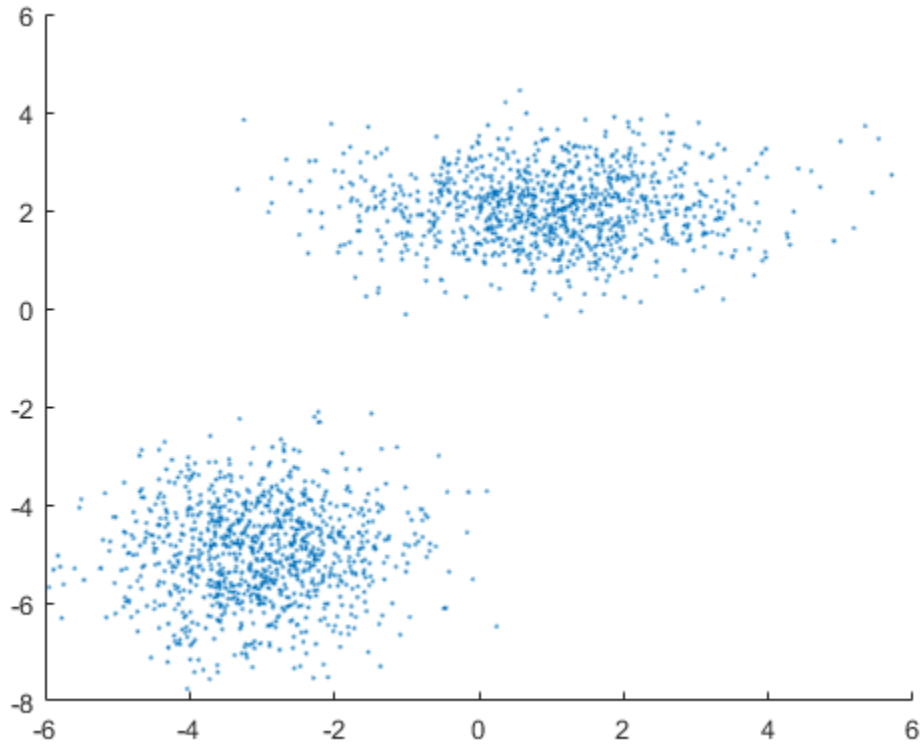
`D = mahal(obj,X)` computes the Mahalanobis distance (in squared units) of each observation in `X` to the mean of each of the  $k$  components of the Gaussian mixture distribution defined by `obj`. `obj` is an object created by `gmdistribution` or `fitgmdist`. `X` is an  $n$ -by- $d$  matrix, where  $n$  is the number of observations and  $d$  is the dimension of the data. `D` is  $n$ -by- $k$ , with `D(I,J)` the distance of observation `I` from the mean of component `J`.

## Examples

### Measure Mahalanobis Distances in Gaussian Mixture Data

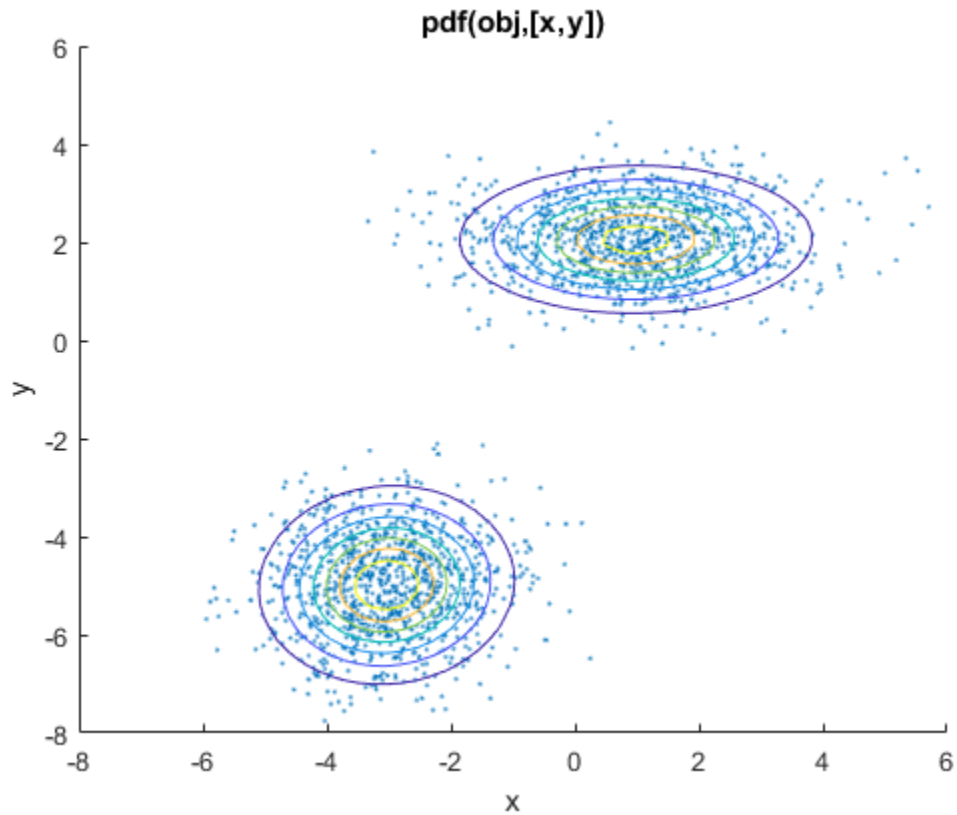
Generate data from a mixture of two bivariate Gaussian distributions using the `mvnrnd` function.

```
MU1 = [1 2];  
SIGMA1 = [2 0; 0 .5];  
MU2 = [-3 -5];  
SIGMA2 = [1 0; 0 1];  
rng(1); % For reproducibility  
X = [mvnrnd(MU1,SIGMA1,1000);mvnrnd(MU2,SIGMA2,1000)];  
  
scatter(X(:,1),X(:,2),10,'.')  
hold on
```



Fit a two-component Gaussian mixture model.

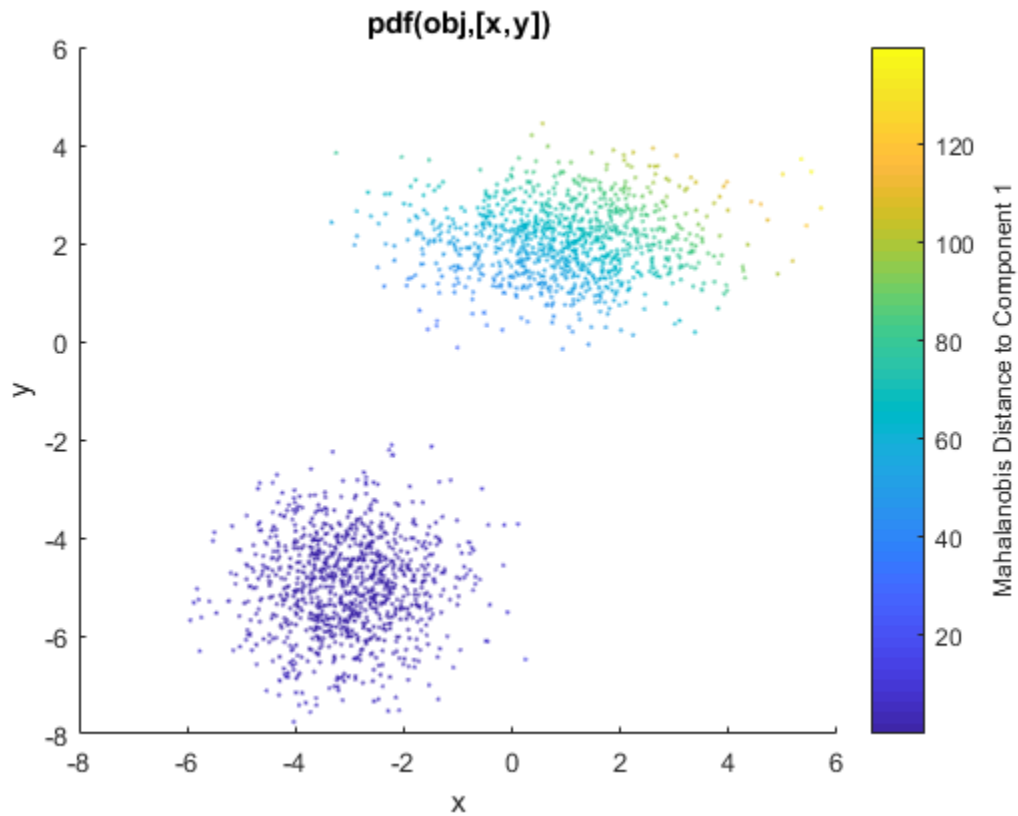
```
obj = fitgmdist(X,2);  
h = ezcontour(@ (x,y) pdf(obj,[x y]),[-8 6],[-8 6]);
```



Compute the Mahalanobis distance of each point in  $X$  to the mean of each component of  $obj$ .

```
D = mahal(obj,X);
```

```
delete(h)  
scatter(X(:,1),X(:,2),10,D(:,1),'.')  
hb = colorbar;  
ylabel(hb,'Mahalanobis Distance to Component 1')
```



## See Also

[cluster](#) | [gmdistribution](#) | [mahal](#) | [posterior](#)



# maineffectsplot

Main effects plot for grouped data

## Syntax

```
maineffectsplot(Y,GROUP)
maineffectsplot(Y,GROUP,param1, val1,param2, val2,...)
[figh,AXESH] = maineffectsplot(...)
```

## Description

`maineffectsplot(Y,GROUP)` displays main effects plots for the group means of matrix `Y` with groups defined by entries in the cell array `GROUP`. `Y` is a numeric matrix or vector. If `Y` is a matrix, the rows represent different observations and the columns represent replications of each observation. Each cell of `GROUP` must contain a grouping variable that can be a categorical variable, numeric vector, character matrix, or single-column cell array of character vectors. `GROUP` can also be a matrix whose columns represent different grouping variables. Each grouping variable must have the same number of rows as `Y`. The number of grouping variables must be greater than 1.

The display has one subplot per grouping variable, with each subplot showing the group means of `Y` as a function of one grouping variable.

`maineffectsplot(Y,GROUP,param1, val1,param2, val2,...)` specifies one or more of the following name/value pairs:

- `'varnames'` — Grouping variable names in a character matrix or a cell array of character vectors, one per grouping variable. Default names are `'X1'`, `'X2'`, ... .
- `'statistic'` — Values that indicate whether the group mean or the group standard deviation should be plotted. Use `'mean'` or `'std'`. The default is `'mean'`. If the value is `'std'`, `Y` is required to have multiple columns.
- `'parent'` — A handle to the figure window for the plots. The default is the current figure window.

`[figh,AXESH] = maineffectsplo(...)` returns the handle `figh` to the figure window and an array of handles `AXESH` to the subplot axes.

## Examples

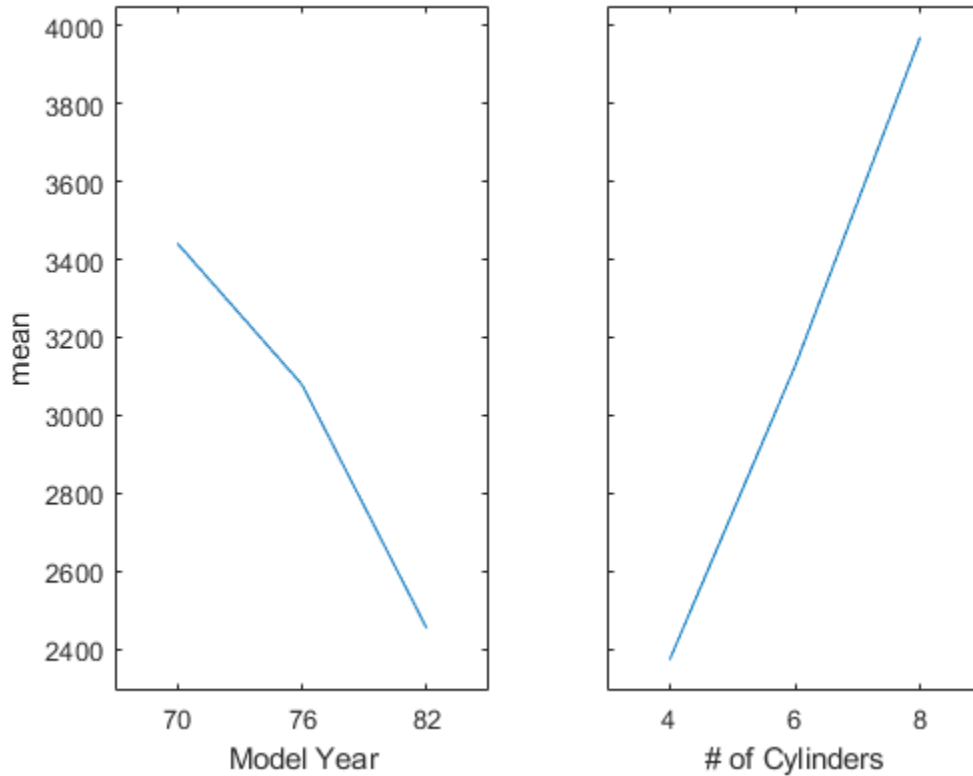
### Main Effects Plot

Load the sample data.

```
load carsmall;
```

Display main effects plots for car weight with two grouping variables, model year and number of cylinders.

```
maineffectsplo(Weight,{Model_Year,Cylinders}, ...  
               'varnames',{'Model Year','# of Cylinders'})
```



## See Also

`interactionplot` | `multivarichart`

## Topics

“Grouping Variables” on page 2-59

Introduced in R2006b

## ClassificationDiscriminant.make

**Class:** ClassificationDiscriminant

Construct discriminant analysis classifier from parameters (to be removed)

---

**Note** `ClassificationDiscriminant.make` will be removed in a future release. Use `makecdiscr` instead.

---

### Syntax

```
cobj = ClassificationDiscriminant.make (Mu, Sigma)
cobj = ClassificationDiscriminant.make (Mu, Sigma, Name, Value)
```

### Description

`cobj = ClassificationDiscriminant.make (Mu, Sigma)` constructs a compact discriminant analysis classifier from the class means `Mu` and covariance matrix `Sigma`.

`cobj = ClassificationDiscriminant.make (Mu, Sigma, Name, Value)` constructs a compact classifier with additional options specified by one or more `Name, Value` pair arguments.

### Input Arguments

#### **Mu** — Class means

matrix of scalar values

Class means, specified as a  $K$ -by- $p$  matrix of scalar values class means of size.  $K$  is the number of classes, and  $p$  is the number of predictors. Each row of `Mu` represents the mean of the multivariate normal distribution of the corresponding class. The class indices are in the `ClassNames` attribute.

Data Types: `single` | `double`

**Sigma — Within-class covariance**

matrix of scalar values

Within-class covariance, specified as a matrix of scalar values.

- For a linear discriminant, `Sigma` is a symmetric, positive semidefinite matrix of size  $p$ -by- $p$ , where  $p$  is the number of predictors.
- For a quadratic discriminant, `Sigma` is an array of size  $p$ -by- $p$ -by- $K$ , where  $K$  is the number of classes. For each  $i$ , `Sigma(:, :, i)` is a symmetric, positive semidefinite matrix.

Data Types: `single` | `double`

**Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

**ClassNames — Class names**

numeric vector | categorical vector | logical vector | character array | cell array of character vectors

Class names as ordered in `Mu`, specified as the comma-separated pair consisting of 'ClassNames' and an array containing grouping variables. Use any data type for a grouping variable, including numeric vector, categorical vector, logical vector, character array, or cell array of character vectors.

`ClassNames` names the classes, as ordered in `Mu`.

Default is  $1:K$ , where  $K$  is the number of classes (the number of rows of `Mu`).

Data Types: `single` | `double` | `logical` | `char` | `cell`

**Cost — Cost of misclassification**

square matrix | structure

Cost of misclassification, specified as the comma-separated pair consisting of 'Cost' and a square matrix, where `Cost(i, j)` is the cost of classifying a point into class  $j$  if its true

class is  $i$ . Alternatively, `Cost` can be a structure  $S$  having two fields:  $S.ClassNames$  containing the group names as a variable of the same type as  $y$ , and  $S.ClassificationCosts$  containing the cost matrix.

The default is  $Cost(i, j) = 1$  if  $i \neq j$ , and  $Cost(i, j) = 0$  if  $i = j$ .

Data Types: `single` | `double` | `struct`

#### **PredictorNames** — Predictor variable names

`{'X1', 'X2', ...}` (default) | cell array of character vectors

Predictor variable names, specified as the comma-separated pair consisting of `'PredictorNames'` and a cell array of character vectors containing the names for the predictor variables, in the order in which they appear in  $X$ .

Data Types: `cell`

#### **Prior** — Prior probabilities

`'uniform'` (default) | vector of scalar values | structure

Prior probabilities for each class, specified as the comma-separated pair consisting of `'Prior'` and one of the following.

- `'uniform'`, a character vector meaning all class prior probabilities are equal.
- A vector containing one scalar value for each class.
- A structure  $S$  with two fields:
  - $S.ClassNames$  containing the class names as a variable of the same type as `ClassNames`.
  - $S.ClassProbs$  containing a vector of corresponding probabilities.

Data Types: `single` | `double` | `struct`

#### **ResponseName** — Response variable name

`'Y'` (default) | character vector

Response variable name, specified as the comma-separated pair consisting of `'ResponseName'` and a character vector containing the name of the response variable  $y$ .

Example: `'ResponseName', 'Response'`

Data Types: `char`

## Output Arguments

### **cobj** — Discriminant analysis classifier

discriminant analysis classifier object

Discriminant analysis classifier, returned as a discriminant analysis classifier object of class `CompactClassificationDiscriminant`. You can use the `predict` method to predict classification labels for new data.

## Examples

### Construct a Compact Linear Discriminant Analysis Classifier

Construct a compact linear discriminant analysis classifier from the means and covariances of the Fisher iris data.

```
load fisheriris
mu(1,:) = mean(meas(1:50,:));
mu(2,:) = mean(meas(51:100,:));
mu(3,:) = mean(meas(101:150,:));

mm1 = repmat(mu(1,:),50,1);
mm2 = repmat(mu(2,:),50,1);
mm3 = repmat(mu(3,:),50,1);
cc = meas;
cc(1:50,:) = cc(1:50,:) - mm1;
cc(51:100,:) = cc(51:100,:) - mm2;
cc(101:150,:) = cc(101:150,:) - mm3;
sigstar = cc' * cc / 147; % unbiased estimator of sigma
cpct = ClassificationDiscriminant.make(mu,sigstar,...
    'ClassNames',{'setosa','versicolor','virginica'})

cpct =

classreg.learning.classif.CompactClassificationDiscriminant
    PredictorNames: {'x1' 'x2' 'x3' 'x4'}
    ResponseName: 'Y'
    CategoricalPredictors: []
    ClassNames: {'setosa' 'versicolor' 'virginica'}
    ScoreTransform: 'none'
```

```
DiscrimType: 'linear'  
    Mu: [3x4 double]  
    Coeffs: [3x3 struct]
```

## See Also

[CompactClassificationDiscriminant](#) | [compact](#) | [fitcdiscr](#) | [makecdiscr](#)

## Topics

“Discriminant Analysis Classification” on page 20-2



# makecdiscr

Construct discriminant analysis classifier from parameters

## Syntax

```
cobj = makecdiscr(Mu, Sigma)
cobj = makecdiscr(Mu, Sigma, Name, Value)
```

## Description

`cobj = makecdiscr(Mu, Sigma)` constructs a compact discriminant analysis classifier from the class means `Mu` and covariance matrix `Sigma`.

`cobj = makecdiscr(Mu, Sigma, Name, Value)` constructs a compact classifier with additional options specified by one or more name-value pair arguments. For example, you can specify the cost of misclassification or the prior probabilities for each class.

## Examples

### Construct a Compact Linear Discriminant Analysis Classifier

Construct a compact linear discriminant analysis classifier from the means and covariances of the Fisher iris data.

```
load fisheriris
mu(1,:) = mean(meas(1:50,:));
mu(2,:) = mean(meas(51:100,:));
mu(3,:) = mean(meas(101:150,:));

mm1 = repmat(mu(1,:), 50, 1);
mm2 = repmat(mu(2,:), 50, 1);
mm3 = repmat(mu(3,:), 50, 1);
cc = meas;
cc(1:50,:) = cc(1:50,:) - mm1;
```

```
cc(51:100,:) = cc(51:100,:) - mm2;
cc(101:150,:) = cc(101:150,:) - mm3;
sigstar = cc' * cc / 147; % unbiased estimator of sigma
cpct = makecdiscr(mu,sigstar,...
    'ClassNames',{'setosa','versicolor','virginica'})

cpct =

classreg.learning.classif.CompactClassificationDiscriminant
    PredictorNames: {'x1' 'x2' 'x3' 'x4'}
    ResponseName: 'Y'
    CategoricalPredictors: []
    ClassNames: {'setosa' 'versicolor' 'virginica'}
    ScoreTransform: 'none'
    DiscrimType: 'linear'
    Mu: [3x4 double]
    Coeffs: [3x3 struct]
```

## Input Arguments

### **Mu** — Class means

matrix of scalar values

Class means, specified as a  $K$ -by- $p$  matrix of scalar values class means of size.  $K$  is the number of classes, and  $p$  is the number of predictors. Each row of  $Mu$  represents the mean of the multivariate normal distribution of the corresponding class. The class indices are in the `ClassNames` attribute.

Data Types: `single` | `double`

### **Sigma** — Within-class covariance

matrix of scalar values

Within-class covariance, specified as a matrix of scalar values.

- For a linear discriminant,  $Sigma$  is a symmetric, positive semidefinite matrix of size  $p$ -by- $p$ , where  $p$  is the number of predictors.

- For a quadratic discriminant, `Sigma` is an array of size `p-by-p-by-K`, where `K` is the number of classes. For each `i`, `Sigma(:, :, i)` is a symmetric, positive semidefinite matrix.

Data Types: `single` | `double`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'ClassNames', {'setosa' 'versicolor' 'virginica'}` specifies a discriminant analysis classifier that uses `'setosa'`, `'versicolor'`, and `'virginica'` as the grouping variables.

### **ClassNames** — Class names

numeric vector | categorical vector | logical vector | character array | cell array of character vectors

Class names as ordered in `Mu`, specified as the comma-separated pair consisting of `'ClassNames'` and an array containing grouping variables. Use any data type for a grouping variable, including numeric vector, categorical vector, logical vector, character array, or cell array of character vectors.

The default is `1:K`, where `K` is the number of classes (the number of rows of `Mu`).

Example: `'ClassNames', {'setosa' 'versicolor' 'virginica'}`

Data Types: `single` | `double` | `logical` | `char` | `cell`

### **Cost** — Cost of misclassification

square matrix | structure

Cost of misclassification, specified as the comma-separated pair consisting of `'Cost'` and a square matrix, where `Cost(i, j)` is the cost of classifying a point into class `j` if its true class is `i`. Alternatively, `Cost` can be a structure `S` having two fields: `S.ClassNames` containing the group names as a variable of the same type as `y`, and `S.ClassificationCosts` containing the cost matrix.

The default is `Cost(i, j) = 1` if `i ~ j`, and `Cost(i, j) = 0` if `i = j`.

Data Types: `single` | `double` | `struct`

**PredictorNames** — Predictor variable names

`{'X1','X2',...}` (default) | cell array of character vectors

Predictor variable names, specified as the comma-separated pair consisting of `'PredictorNames'` and a cell array of character vectors containing the names for the predictor variables, in the order in which they appear in `X`.

Data Types: `cell`

**Prior** — Prior probabilities

`'uniform'` (default) | vector of scalar values | structure

Prior probabilities for each class, specified as the comma-separated pair consisting of `'Prior'` and one of the following:

- `'uniform'`, meaning all class prior probabilities are equal
- A vector containing one scalar value for each class
- A structure `S` with two fields:
  - `S.ClassNames` containing the class names as a variable of the same type as `ClassNames`
  - `S.ClassProbs` containing a vector of corresponding probabilities

Data Types: `single` | `double` | `struct`

**ResponseName** — Response variable name

`'Y'` (default) | character vector

Response variable name, specified as the comma-separated pair consisting of `'ResponseName'` and a character vector containing the name of the response variable `y`.

Example: `'ResponseName','Response'`

Data Types: `char`

## Output Arguments

**cobj** — Discriminant analysis classifier

discriminant analysis classifier object

Discriminant analysis classifier, returned as a discriminant analysis classifier object of class `CompactClassificationDiscriminant`. You can use the `predict` method to predict classification labels for new data.

## Tips

- You can change the discriminant type using dot notation after constructing `cobj`:

```
cobj.DiscrimType = 'discrimType'
```

where *discrimType* is one of 'linear', 'quadratic', 'diagLinear', 'diagQuadratic', 'pseudoLinear', or 'pseudoQuadratic'. You can change between linear types or between quadratic types, but cannot change between a linear and a quadratic type.

- `cobj` is a linear classifier when `Sigma` is a matrix. `cobj` is a quadratic classifier when `Sigma` is a three-dimensional array.

## See Also

`CompactClassificationDiscriminant` | `compact` | `fitcdiscr` | `predict`

## Topics

“Discriminant Analysis Classification” on page 20-2

**Introduced in R2014a**

## makedist

Create probability distribution object

### Syntax

```
pd = makedist(distname)
pd = makedist(distname, Name, Value)

list = makedist

makedist -reset
```

### Description

`pd = makedist(distname)` creates a probability distribution object for the distribution `distname`, using the default parameter values.

`pd = makedist(distname, Name, Value)` creates a probability distribution object with one or more distribution parameter values specified by name-value pair arguments.

`list = makedist` returns a cell array `list` containing a list of the probability distributions that `makedist` can create.

`makedist -reset` resets the list of distributions by searching the path for files contained in a package named `prob` and implementing classes derived from `ProbabilityDistribution`.

### Examples

#### Create a Normal Distribution Object

Create a normal distribution object using the default parameter values.

```
pd = makedist('Normal')
```

```
pd =  
  
NormalDistribution  
  
Normal distribution  
  mu = 0  
  sigma = 1
```

Compute the interquartile range of the distribution.

```
r = iqr(pd)
```

```
r =  
  
1.3490
```

### Create a Gamma Distribution Object

Create a gamma distribution object using the default parameter values.

```
pd = makedist('Gamma')
```

```
pd =  
  
GammaDistribution  
  
Gamma distribution  
  a = 1  
  b = 1
```

Compute the mean of the gamma distribution.

```
mean = mean(pd)
```

```
mean =
```

1

### Specify Parameters for a Normal Distribution Object

Create a normal distribution object with parameter values  $\mu = 75$  and  $\sigma = 10$ .

```
pd = makedist('Normal', 'mu', 75, 'sigma', 10)
```

```
pd =
```

```
NormalDistribution
```

```
Normal distribution
```

```
mu = 75
```

```
sigma = 10
```

### Specify Parameters for a Gamma Distribution Object

Create a gamma distribution object with the parameter value  $a = 3$  and the default value  $b = 1$ .

```
pd = makedist('Gamma', 'a', 3)
```

```
pd =
```

```
GammaDistribution
```

```
Gamma distribution
```

```
a = 3
```

```
b = 1
```

## Input Arguments

**distname** — Distribution name

character vector



Distribution name, specified as one of the following character vectors. The distribution specified by `distname` determines the type of the returned probability distribution object.

Distribution Name	Description	Distribution Object
'Beta'	Beta distribution	BetaDistribution
'Binomial'	Binomial distribution	BinomialDistribution
'BirnbaumSaunders'	Birnbaum-Saunders distribution	BirnbaumSaundersDistribution
'Burr'	Burr distribution	BurrDistribution
'Exponential'	Exponential distribution	ExponentialDistribution
'ExtremeValue'	Extreme Value distribution	ExtremeValueDistribution
'Gamma'	Gamma distribution	GammaDistribution
'GeneralizedExtremeValue'	Generalized Extreme Value distribution	GeneralizedExtremeValueDistribution
'GeneralizedPareto'	Generalized Pareto distribution	GeneralizedParetoDistribution
'HalfNormal'	Half-normal distribution	HalfNormalDistribution
'InverseGaussian'	Inverse Gaussian distribution	InverseGaussianDistribution
'Logistic'	Logistic distribution	LogisticDistribution
'Loglogistic'	Loglogistic distribution	LoglogisticDistribution
'Lognormal'	Lognormal distribution	LognormalDistribution
'Multinomial'	Multinomial distribution	MultinomialDistribution
'Nakagami'	Nakagami distribution	NakagamiDistribution
'NegativeBinomial'	Negative Binomial distribution	NegativeBinomialDistribution
'Normal'	Normal distribution	NormalDistribution

Distribution Name	Description	Distribution Object
'PiecewiseLinear'	Piecewise Linear distribution	PiecewiseLinearDistribution
'Poisson'	Poisson distribution	PoissonDistribution
'Rayleigh'	Rayleigh distribution	RayleighDistribution
'Rician'	Rician distribution	RicianDistribution
'Stable'	Stable distribution	StableDistribution
'tLocationScale'	$t$ Location-Scale distribution	tLocationScaleDistribution
'Triangular'	Triangular distribution	TriangularDistribution
'Uniform'	Uniform distribution	UniformDistribution
'Weibull'	Weibull distribution	WeibullDistribution

## Name-Value Pair Arguments

Specify optional comma-separated pairs of *Name*, *Value* arguments. *Name* is the argument name and *Value* is the corresponding value. *Name* must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as *Name1*, *Value1*, . . . , *NameN*, *ValueN*.

Example: `makedist('Normal', 'mu', 10)` specifies a normal distribution with parameter *mu* equal to 10, and parameter *sigma* equal to the default value of 1.

### Beta Distribution

#### **a** — First shape parameter

1 (default) | nonnegative scalar value

Example: 'a', 3

Data Types: single | double

#### **b** — Second shape parameter

1 (default) | nonnegative scalar value

Example: 'b', 5

Data Types: single | double

## Binomial Distribution

### **n** — Number of trials

1 (default) | positive integer value

Example: 'n', 25

Data Types: single | double

### **p** — Probability of success

0.5 (default) | scalar value in the range [0,1]

Example: 'p', 0.25

Data Types: single | double

## Birnbaum-Saunders Distribution

### **beta** — Scale parameter

1 (default) | positive scalar value

Example: 'beta', 2

Data Types: single | double

### **gamma** — Shape parameter

1 (default) | nonnegative scalar value

Example: 'gamma', 0

Data Types: single | double

## Burr Distribution

### **alpha** — Scale parameter

1 (default) | positive scalar value

Example: 'alpha', 2

Data Types: single | double

### **c** — First shape parameter

1 (default) | positive scalar value

Example: 'c', 2

Data Types: `single` | `double`

**k — Second shape parameter**

1 (default) | positive scalar value

Example: `'k', 5`

Data Types: `single` | `double`

**Exponential Distribution**

**mu — Mean parameter**

1 (default) | positive scalar value

Example: `'mu', 5`

Data Types: `single` | `double`

**Extreme Value Distribution**

**mu — Location parameter**

0 (default) | scalar value

Example: `'mu', -2`

Data Types: `single` | `double`

**sigma — Scale parameter**

1 (default) | nonnegative scalar value

Example: `'sigma', 2`

Data Types: `single` | `double`

**Gamma Distribution**

**a — Shape parameter**

1 (default) | positive scalar value

Example: `'a', 2`

Data Types: `single` | `double`

**b — Scale parameter**

1 (default) | nonnegative scalar value

Example: 'b',0

Data Types: single | double

### Generalized Extreme Value Distribution

#### **k** — Shape parameter

0 (default) | scalar value

Example: 'k',0

Data Types: single | double

#### **sigma** — Scale parameter

1 (default) | nonnegative scalar value

Example: 'sigma',2

Data Types: single | double

#### **mu** — Location parameter

0 (default) | scalar value

Example: 'mu',1

Data Types: single | double

### Generalized Pareto Distribution

#### **k** — Shape parameter

1 (default) | scalar value

Example: 'k',0

Data Types: single | double

#### **sigma** — Scale parameter

1 (default) | nonnegative scalar value

Example: 'sigma',2

Data Types: single | double

#### **theta** — Location parameter

1 (default) | scalar value

Example: 'theta', 2

Data Types: single | double

### Half-Normal Distribution

#### **mu — Location parameter**

0 (default) | positive scalar value

Example: 'mu', 1

Data Types: single | double

#### **sigma — Shape parameter**

1 (default) | positive scalar value

Example: 'sigma', 2

Data Types: single | double

### Inverse Gaussian Distribution

#### **mu — Scale parameter**

1 (default) | positive scalar value

Example: 'mu', 2

Data Types: single | double

#### **lambda — Shape parameter**

1 (default) | positive scalar value

Example: 'lambda', 4

Data Types: single | double

### Logistic Distribution

#### **mu — Mean**

0 (default) | scalar value

Example: 'mu', 2

Data Types: single | double

**sigma — Scale parameter**

1 (default) | nonnegative scalar value

Example: 'sigma', 4

Data Types: single | double

**Loglogistic Distribution****mu — Log mean**

0 (default) | scalar value

Example: 'mu', 2

Data Types: single | double

**sigma — Log scale parameter**

1 (default) | nonnegative scalar value

Example: 'sigma', 4

Data Types: single | double

**Lognormal Distribution****mu — Log mean**

0 (default) | scalar value

Example: 'mu', 2

Data Types: single | double

**sigma — Log standard deviation**

1 (default) | nonnegative scalar value

Example: 'sigma', 2

Data Types: single | double

**Multinomial Distribution****probabilities — Outcome probabilities**

[0.500 0.500] (default) | vector of scalar values in the range [0,1]

Outcome probabilities, specified as a vector of scalar values in the range [0,1]. The probabilities sum to 1 and correspond to outcomes [1, 2, ..., k], where  $k$  is the number of elements in the probabilities vector.

Example: 'probabilities', [0.1 0.2 0.5 0.2] gives the probabilities that the outcome is 1, 2, 3, or 4, respectively.

Data Types: single | double

### **Nakagami Distribution**

#### **mu — Shape parameter**

1 (default) | positive scalar value

Example: 'mu', 5

Data Types: single | double

#### **omega — Scale parameter**

1 (default) | positive scalar value

Example: 'omega', 5

Data Types: single | double

### **Negative Binomial Distribution**

#### **R — Number of successes**

1 (default) | positive scalar value

Example: 'R', 5

Data Types: single | double

#### **p — Probability of success**

0.5 (default) | scalar value in the range (0,1]

Example: 'p', 0.1

Data Types: single | double

### **Normal Distribution**

#### **mu — Mean**

0 (default) | scalar value



Example: 'mu', 2

Data Types: single | double

#### **sigma — Standard deviation**

1 (default) | nonnegative scalar value

Example: 'sigma', 2

Data Types: single | double

### **Piecewise Linear Distribution**

#### **x — Data values**

1 (default) | monotonically increasing vector of scalar values

Example: 'x', [1 2 3]

Data Types: single | double

#### **Fx — cdf values**

1 (default) | monotonically increasing vector of scalar values that start at 0 and end at 1

Example: 'Fx', [0.2 0.5 1]

Data Types: single | double

### **Poisson Distribution**

#### **lambda — Mean**

1 (default) | nonnegative scalar value

Example: 'lambda', 5

Data Types: single | double

### **Rayleigh Distribution**

#### **b — Defining parameter**

1 (default) | positive scalar value

Example: 'b', 3

Data Types: single | double

### **Rician Distribution**

**s — Noncentrality parameter**

1 (default) | nonnegative scalar value

Example: 's', 0

Data Types: single | double

**sigma — Scale parameter**

1 (default) | positive scalar value

Example: 'sigma', 2

Data Types: single | double

**Stable Distribution**

**alpha — First shape parameter**

2 (default) | scalar value in the range (0,2]

Example: 'alpha', 1

Data Types: single | double

**beta — Second shape parameter**

0 (default) | scalar value in the range [-1,1]

Example: 'beta', 0.5

Data Types: single | double

**gam — Scale parameter**

1 (default) | scalar value in the range (0,∞)

Example: 'gam', 2

Data Types: single | double

**delta — Location parameter**

0 (default) | scalar value

Example: 'delta', 5

Data Types: single | double

**t Location-Scale Distribution**

**mu — Location parameter**

0 (default) | scalar value

Example: 'mu', -2

Data Types: single | double

**sigma — Scale parameter**

1 (default) | positive scalar value

Example: 'sigma', 2

Data Types: single | double

**nu — Degrees of freedom**

5 (default) | positive scalar value

Example: 'nu', 20

Data Types: single | double

**Triangular Distribution****a — Lower limit**

0 (default) | scalar value

Example: 'a', -2

Data Types: single | double

**b — Peak location**

0.5 (default) | scalar value greater than or equal to a

Example: 'b', 1

Data Types: single | double

**c — Upper limit**

1 (default) | scalar value greater than or equal to b

Example: 'c', 5

Data Types: single | double

**Uniform Distribution**

**lower** — Lower parameter

0 (default) | scalar value

Example: 'lower', -4

Data Types: single | double

**upper** — Upper parameter

1 (default) | scalar value greater than lower

Example: 'upper', 2

Data Types: single | double

**Weibull Distribution****a** — Scale parameter

1 (default) | positive scalar value

Example: 'a', 2

Data Types: single | double

**b** — Shape parameter

1 (default) | positive scalar value

Example: 'b', 5

Data Types: single | double

## Output Arguments

**pd** — Probability distribution

probability distribution object

Probability distribution, returned as a probability distribution object of the type specified by `distname`.

**list** — List of probability distributions

cell array of character vectors

List of probability distributions that `makedist` can create, returned as a cell array of character vectors.

## Alternative Functionality

### App

The Distribution Fitter app opens a graphical user interface for you to import data from the workspace and interactively fit a probability distribution to that data. You can then save the distribution to the workspace as a probability distribution object. Open the Distribution Fitter app using `distributionFitter`, or click Distribution Fitter on the Apps tab.

### See Also

`distributionFitter` | `fitdist`

**Introduced in R2013a**

## manova

**Class:** RepeatedMeasuresModel

Multivariate analysis of variance

### Syntax

```
manovatbl = manova(rm)
manovatbl = manova(rm, Name, Value)
[manovatbl, A, C, D] = manova(____)
```

### Description

`manovatbl = manova(rm)` returns the results of multivariate analysis of variance (manova) for the repeated measures model `rm`.

`manovatbl = manova(rm, Name, Value)` also returns manova results with additional options, specified by one or more `Name, Value` pair arguments.

`[manovatbl, A, C, D] = manova(____)` also returns arrays `A`, `C`, and `D` for the hypotheses tests of the form  $A*B*C = D$ , where `D` is zero.

### Input Arguments

**rm — Repeated measures model**

RepeatedMeasuresModel object

Repeated measures model, returned as a RepeatedMeasuresModel object.

For properties and methods of this object, see RepeatedMeasuresModel.

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single

quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### **WithinModel** — Model specifying within-subjects hypothesis test

'separatemeans' (default) | model specification using formula

Model specifying the within-subjects hypothesis test, specified as one of the following:

- 'separatemeans' — Compute a separate mean for each group, and test for equality among the means.
- Model specification — This is a model specification in the within-subject factors. Test each term in the model. In this case, `tbl` contains a separate manova for each term in the formula, with the multivariate response equal to the vector of coefficients of that term.
- An  $r$ -by- $nc$  matrix,  $C$ , specifying  $nc$  contrasts among the  $r$  repeated measures. If  $Y$  represents the matrix of repeated measures you use in the repeated measures model `rm`, then the output `tbl` contains a separate manova for each column of  $Y*C$ .

Example: `'WithinModel', 'separatemeans'`

### **By** — Single between-subjects factor

character vector

Single between-subjects factor, specified as the comma-separated pair consisting of 'By' and a character vector. `manova` performs a separate test of the within-subjects model for each value of this factor.

For example, if you have a between-subjects factor, `Drug`, then you can specify that factor to perform `manova` as follows.

Example: `'By', 'Drug'`

## Output Arguments

### **manova**`tbl` — Results of multivariate analysis of variance

table

Results of multivariate analysis of variance for the repeated measures model `rm`, returned as a `table`.

`manova` uses these methods to measure the contributions of the model terms to the overall covariance:

- Wilks' Lambda
- Pillai's trace
- Hotelling-Lawley trace
- Roy's maximum root statistic

For details, see “Multivariate Analysis of Variance for Repeated Measures” on page 9-85.

`manova` returns the results for these tests for each group. `manovatbl` contains the following columns.

Column Name	Definition
Within	Within-subject terms
Between	Between-subject terms
Statistic	Name of the statistic computed
Value	Value of the corresponding statistic
F	$F$ -statistic value
RSquare	Measure for variance explained
df1	Numerator degrees of freedom
df2	Denominator degrees of freedom
pValue	$p$ -value for the corresponding $F$ -statistic value

Data Types: `table`

#### **A** — Specification based on between-subjects model

`matrix` | `cell array`

Specification based on the between-subjects model, returned as a matrix or a cell array. It permits the hypothesis on the elements within given columns of `B` (within time hypothesis). If `manovatbl` contains multiple hypothesis tests, `A` might be a cell array.

Data Types: `single` | `double` | `cell`

#### **c** — Specification based on within-subjects model

`matrix` | `cell array`



Specification based on the within-subjects model, returned as a matrix or a cell array. It permits the hypotheses on the elements within given rows of **B** (between time hypotheses). If `manovatbl` contains multiple hypothesis tests, **C** might be a cell array.

Data Types: `single` | `double` | `cell`

### **D** — Hypothesis value

0

Hypothesis value, returned as 0.

## Examples

### Perform Multivariate Analysis of Variance

Load the sample data.

```
load fisheriris
```

The column vector `species` consists of iris flowers of three different species: `setosa`, `versicolor`, `virginica`. The double matrix `meas` consists of four types of measurements on the flowers: the length and width of sepals and petals in centimeters, respectively.

Store the data in a table array.

```
t = table(species,meas(:,1),meas(:,2),meas(:,3),meas(:,4),...
'VariableNames',{'species','meas1','meas2','meas3','meas4'});
Meas = table([1 2 3 4'],'VariableNames',{'Measurements'});
```

Fit a repeated measures model where the measurements are the responses and the species is the predictor variable.

```
rm = fitrm(t,'meas1-meas4~species','WithinDesign',Meas);
```

Perform multivariate analysis of variance.

```
manova(rm)
```

```
ans =
```

8x9 table

Within	Between	Statistic	Value	F	RSquare	df1	df2
Constant	(Intercept)	Pillai	0.99013	4847.5	0.99013	3	14
Constant	(Intercept)	Wilks	0.0098724	4847.5	0.99013	3	14
Constant	(Intercept)	Hotelling	100.29	4847.5	0.99013	3	14
Constant	(Intercept)	Roy	100.29	4847.5	0.99013	3	14
Constant	species	Pillai	0.96909	45.749	0.48455	6	29
Constant	species	Wilks	0.041153	189.92	0.79714	6	29
Constant	species	Hotelling	23.051	555.17	0.92016	6	28
Constant	species	Roy	23.04	1121.3	0.9584	3	14

Perform multivariate anova separately for each species.

```
manova(rm, 'By', 'species')
```

```
ans =
```

12x9 table

Within	Between	Statistic	Value	F	RSquare	df1	df2
Constant	species=setosa	Pillai	0.9823	2682.7	0.9823	3	14
Constant	species=setosa	Wilks	0.017698	2682.7	0.9823	3	14
Constant	species=setosa	Hotelling	55.504	2682.7	0.9823	3	14
Constant	species=setosa	Roy	55.504	2682.7	0.9823	3	14
Constant	species=versicolor	Pillai	0.97	1562.8	0.97	3	14
Constant	species=versicolor	Wilks	0.029999	1562.8	0.97	3	14
Constant	species=versicolor	Hotelling	32.334	1562.8	0.97	3	14
Constant	species=versicolor	Roy	32.334	1562.8	0.97	3	14
Constant	species=virginica	Pillai	0.97261	1716.1	0.97261	3	14
Constant	species=virginica	Wilks	0.027394	1716.1	0.97261	3	14
Constant	species=virginica	Hotelling	35.505	1716.1	0.97261	3	14
Constant	species=virginica	Roy	35.505	1716.1	0.97261	3	14

## Return Arrays of the Hypothesis Test

Load the sample data.

```
load fisheriris
```

The column vector `species` consists of iris flowers of three different species: `setosa`, `versicolor`, `virginica`. The double matrix `meas` consists of four types of measurements on the flowers: the length and width of sepals and petals in centimeters, respectively.

Store the data in a table array.

```
t = table(species,meas(:,1),meas(:,2),meas(:,3),meas(:,4),...
'VariableNames',{'species','meas1','meas2','meas3','meas4'});
Meas = dataset([1 2 3 4]','VarNames',{'Measurements'});
```

Fit a repeated measures model where the measurements are the responses and the species is the predictor variable.

```
rm = fitrm(t,'meas1-meas4~species','WithinDesign',Meas);
```

Perform multivariate analysis of variance. Also return the arrays for constructing the hypothesis test.

```
[manovatbl,A,C,D] = manova(rm)
```

```
manovatbl =
```

```
8x9 table
```

Within	Between	Statistic	Value	F	RSquare	df1	df2
Constant	(Intercept)	Pillai	0.99013	4847.5	0.99013	3	14
Constant	(Intercept)	Wilks	0.0098724	4847.5	0.99013	3	14
Constant	(Intercept)	Hotelling	100.29	4847.5	0.99013	3	14
Constant	(Intercept)	Roy	100.29	4847.5	0.99013	3	14
Constant	species	Pillai	0.96909	45.749	0.48455	6	29
Constant	species	Wilks	0.041153	189.92	0.79714	6	29
Constant	species	Hotelling	23.051	555.17	0.92016	6	28
Constant	species	Roy	23.04	1121.3	0.9584	3	14

```
A =
```

```
2x1 cell array
```

```
{1x3 double}  
{2x3 double}
```

```
C =
```

```
     1     0     0  
    -1     1     0  
     0    -1     1  
     0     0    -1
```

```
D =
```

```
     0
```

**Index into matrix A.**

```
A{1}
```

```
ans =
```

```
     1     0     0
```

```
A{2}
```

```
ans =
```

```
     0     1     0  
     0     0     1
```

## Tips

- The multivariate response for each observation (subject) is the vector of repeated measures.

- To test a more general hypothesis  $A*B*C = D$ , use `coefstest`.

## See Also

`anova` | `coefstest` | `fitrm` | `ranova`

## Topics

“Model Specification for Repeated Measures Models” on page 9-79

“Multivariate Analysis of Variance for Repeated Measures” on page 9-85

## manova1

One-way multivariate analysis of variance

### Syntax

```
d = manova1(X,group)
d = manova1(X,group,alpha)
[d,p] = manova1(...)
[d,p,stats] = manova1(...)
```

### Description

`d = manova1(X,group)` performs a one-way Multivariate Analysis of Variance (MANOVA) for comparing the multivariate means of the columns of `X`, grouped by `group`. `X` is an  $m$ -by- $n$  matrix of data values, and each row is a vector of measurements on  $n$  variables for a single observation. `group` is a grouping variable defined as a categorical variable, vector, character array, or cell array of character vectors. Two observations are in the same group if they have the same value in the `group` array. The observations in each group represent a sample from a population.

The function returns `d`, an estimate of the dimension of the space containing the group means. `manova1` tests the null hypothesis that the means of each group are the same  $n$ -dimensional multivariate vector, and that any difference observed in the sample `X` is due to random chance. If `d = 0`, there is no evidence to reject that hypothesis. If `d = 1`, then you can reject the null hypothesis at the 5% level, but you cannot reject the hypothesis that the multivariate means lie on the same line. Similarly, if `d = 2` the multivariate means may lie on the same plane in  $n$ -dimensional space, but not on the same line.

`d = manova1(X,group,alpha)` gives control of the significance level, `alpha`. The return value `d` will be the smallest dimension having `p > alpha`, where `p` is a  $p$ -value for testing whether the means lie in a space of that dimension.

`[d,p] = manova1(...)` also returns a `p`, a vector of  $p$ -values for testing whether the means lie in a space of dimension 0, 1, and so on. The largest possible dimension is either

the dimension of the space, or one less than the number of groups. There is one element of  $p$  for each dimension up to, but not including, the largest.

If the  $i$ th  $p$ -value is near zero, this casts doubt on the hypothesis that the group means lie on a space of  $i-1$  dimensions. The choice of a critical  $p$ -value to determine whether the result is judged statistically significant is left to the researcher and is specified by the value of the input argument `alpha`. It is common to declare a result significant if the  $p$ -value is less than 0.05 or 0.01.

`[d,p,stats] = manova1(...)` also returns `stats`, a structure containing additional MANOVA results. The structure contains the following fields.

Field	Contents
<code>W</code>	Within-groups sum of squares and cross-products matrix
<code>B</code>	Between-groups sum of squares and cross-products matrix
<code>T</code>	Total sum of squares and cross-products matrix
<code>dfW</code>	Degrees of freedom for <code>W</code>
<code>dfB</code>	Degrees of freedom for <code>B</code>
<code>dfT</code>	Degrees of freedom for <code>T</code>
<code>lambda</code>	Vector of values of Wilk's lambda test statistic for testing whether the means have dimension 0, 1, etc.
<code>chisq</code>	Transformation of <code>lambda</code> to an approximate chi-square distribution
<code>chisqdf</code>	Degrees of freedom for <code>chisq</code>
<code>eigenval</code>	Eigenvalues of $W^{-1}B$
<code>eigenvec</code>	Eigenvectors of $W^{-1}B$ ; these are the coefficients for the canonical variables <code>C</code> , and they are scaled so the within-group variance of the canonical variables is 1
<code>canon</code>	Canonical variables <code>C</code> , equal to <code>XC*eigenvec</code> , where <code>XC</code> is <code>X</code> with columns centered by subtracting their means
<code>mdist</code>	A vector of Mahalanobis distances from each point to the mean of its group
<code>gmdist</code>	A matrix of Mahalanobis distances between each pair of group means

The canonical variables `C` are linear combinations of the original variables, chosen to maximize the separation between groups. Specifically, `C(:,1)` is the linear combination

of the  $X$  columns that has the maximum separation between groups. This means that among all possible linear combinations, it is the one with the most significant  $F$  statistic in a one-way analysis of variance.  $C(:, 2)$  has the maximum separation subject to it being orthogonal to  $C(:, 1)$ , and so on.

You may find it useful to use the outputs from `manova1` along with other functions to supplement your analysis. For example, you may want to start with a grouped scatter plot matrix of the original variables using `gplotmatrix`. You can use `gscatter` to visualize the group separation using the first two canonical variables. You can use `manovacluster` to graph a dendrogram showing the clusters among the group means.

## Assumptions

The MANOVA test makes the following assumptions about the data in  $X$ :

- The populations for each group are normally distributed.
- The variance-covariance matrix is the same for each population.
- All observations are mutually independent.

## Examples

you can use `manova1` to determine whether there are differences in the averages of four car characteristics, among groups defined by the country where the cars were made.

```
load carbig
[d,p] = manova1([MPG Acceleration Weight Displacement],...
               Origin)
d =
     3
p =
     0
    0.0000
    0.0075
    0.1934
```

There are four dimensions in the input matrix, so the group means must lie in a four-dimensional space. `manova1` shows that you cannot reject the hypothesis that the means lie in a 3-D subspace.



## References

- [1] Krzanowski, W. J. *Principles of Multivariate Analysis: A User's Perspective*. New York: Oxford University Press, 1988.

## See Also

`anova1` | `canoncorr` | `gplotmatrix` | `gscatter` | `manovacluster`

## Topics

“Grouping Variables” on page 2-59

**Introduced before R2006a**

## manovacluster

Dendrogram of group mean clusters following MANOVA

### Syntax

```
manovacluster(stats)
manovacluster(stats,method)
H = manovacluster(stats,method)
```

### Description

`manovacluster(stats)` generates a dendrogram plot of the group means after a multivariate analysis of variance (MANOVA). `stats` is the output `stats` structure from `manova1`. The clusters are computed by applying the single linkage method to the matrix of Mahalanobis distances between group means.

See `dendrogram` for more information on the graphical output from this function. The dendrogram is most useful when the number of groups is large.

`manovacluster(stats,method)` uses the specified method in place of single linkage. `method` can be any of the following character vectors that identify ways to create the cluster hierarchy. (See `linkage` for additional information.)

Method	Description
'single'	Shortest distance (default)
'complete'	Largest distance
'average'	Average distance
'centroid'	Centroid distance
'ward'	Incremental sum of squares

`H = manovacluster(stats,method)` returns a vector of handles to the lines in the figure.

## Examples

### Dendrogram of Group Means After MANOVA

Load the sample data.

```
load carbig
```

Define the variable matrix.

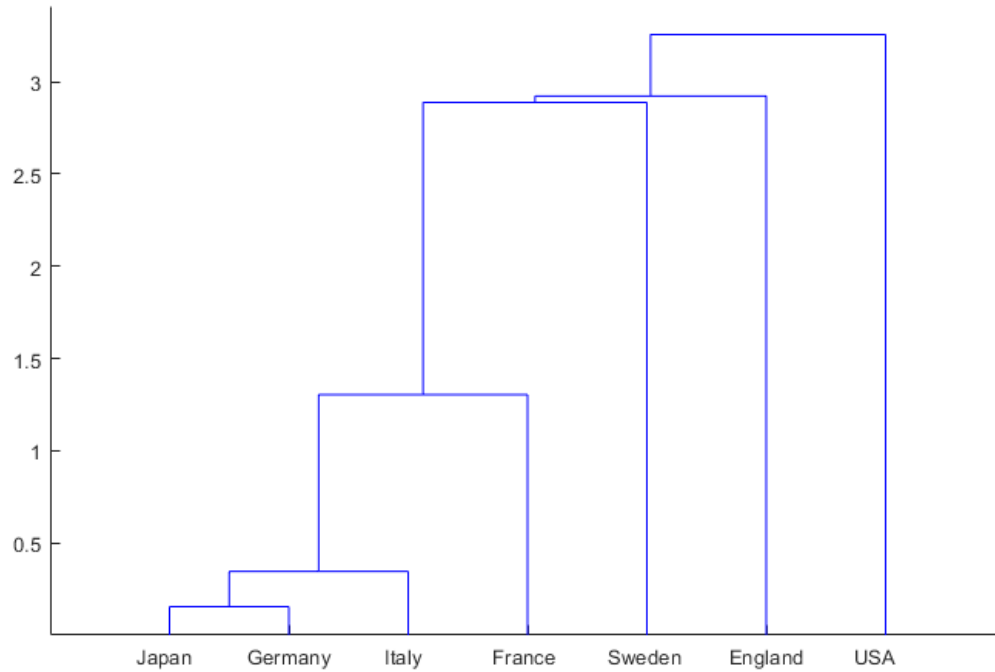
```
X = [MPG Acceleration Weight Displacement];
```

Perform one-way MANOVA to compare the means of MPG, Acceleration, Weight, and Displacement grouped by Origin.

```
[d,p,stats] = manoval(X,Origin);
```

Create a dendrogram plot of the group means.

```
manovacluster(stats)
```



## See Also

`cluster` | `dendrogram` | `linkage` | `manova1`

Introduced before R2006a

# margin

**Class:** ClassificationKNN

Margin of  $k$ -nearest neighbor classifier

## Syntax

```
m = margin mdl, tbl, ResponseVarName)
m = margin mdl, tbl, Y)
m = margin mdl, X, Y)
```

## Description

`m = margin(mdl, tbl, ResponseVarName)` returns the classification margins for the matrix of predictors  $X$  and class labels  $Y$ . For the definition, see “Margin” on page 32-3882.

`m = margin(mdl, tbl, Y)` returns the classification margins for the matrix of predictors  $X$  and class labels  $Y$ .

`m = margin(mdl, X, Y)` returns the classification margins for the matrix of predictors  $X$  and class labels  $Y$ .

## Input Arguments

**mdl** — Classifier model

classifier model object

$k$ -nearest neighbor classifier model, returned as a classifier model object.

Note that using the 'CrossVal', 'Kfold', 'Holdout', 'Leaveout', or 'CVPartition' options results in a model of class `ClassificationPartitionedModel`. You cannot use a partitioned tree for prediction, so this kind of tree does not have a `predict` method.

Otherwise, `mdl` is of class `ClassificationKNN`, and you can use the `predict` method to make predictions.

**tbl — Sample data**

table

Sample data used to train the model, specified as a table. Each row of `tbl` corresponds to one observation, and each column corresponds to one predictor variable. Optionally, `tbl` can contain one additional column for the response variable. Multi-column variables and cell arrays other than cell arrays of character vectors are not allowed.

If `tbl` contains the response variable used to train `mdl`, then you do not need to specify `ResponseVarName` or `Y`.

If you trained `mdl` using sample data contained in a `table`, then the input data for this method must also be in a table.

Data Types: `table`

**ResponseVarName — Response variable name**

name of a variable in `tbl`

Response variable name, specified as the name of a variable in `tbl`. If `tbl` contains the response variable used to train `mdl`, then you do not need to specify `ResponseVarName`.

If you specify `ResponseVarName`, then you must do so as a character vector. For example, if the response variable is stored as `tbl.response`, then specify it as `'response'`. Otherwise, the software treats all columns of `tbl`, including `tbl.response`, as predictors.

The response variable must be a categorical or character array, logical or numeric vector, or cell array of character vectors. If the response variable is a character array, then each element must correspond to one row of the array.

**x — Matrix of predictor values**

matrix

Matrix of predictor values. Each column of `X` represents one variable, and each row represents one observation.

**Y — Categorical variables**

categorical array | cell array of character vectors | character array | logical vector | numeric vector

A categorical array, cell array of character vectors, character array, logical vector, or a numeric vector with the same number of rows as X. Each row of Y represents the classification of the corresponding row of X.

## Output Arguments

**m — Classification margins**

numeric vector

Numeric column vector of length `size(X,1)`. Each entry in m represents the margin for the corresponding rows of X and (true class) Y, computed using mdl.

## Examples

**Margin Calculation**

Construct a *k*-nearest neighbor classifier for the Fisher iris data, where *k* = 5.

Load the data.

```
load fisheriris
```

Construct a classifier for 5-nearest neighbors.

```
mdl = fitcknn(meas,species,'NumNeighbors',5);
```

Examine the margin of the classifier for a mean observation classified 'versicolor'.

```
X = mean(meas);  
Y = {'versicolor'};  
m = margin(mdl,X,Y)
```

```
m =
```

1

The classifier has no doubt that 'versicolor' is the correct classification (all five nearest neighbors classify as 'versicolor').

## Definitions

### Margin

The classification margin is the difference between the classification score for the true class and maximal classification score for the false classes.

### Score

The score of a classification is the posterior probability of the classification. The posterior probability is the number of neighbors that have that classification, divided by the number of neighbors. For a more detailed definition that includes weights and prior probabilities, see “Posterior Probability” on page 32-4897.

## Extended Capabilities

### Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays” (MATLAB).

### See Also

`ClassificationKNN` | `edge` | `fitcknn` | `loss`



## Topics

“Classification Using Nearest Neighbors” on page 18-16

## margin

**Class:** ClassificationLinear

Classification margins for linear classification models

## Syntax

```
m = margin(Mdl, X, Y)
```

```
m = margin( ____, Name, Value)
```

## Description

`m = margin(Mdl, X, Y)` returns the classification margins on page 32-3894 for the binary, linear classification model `Mdl` using predictor data in `X` and corresponding class labels in `Y`. `m` contains classification margins for each regularization strength in `Mdl`.

`m = margin( ____, Name, Value)` uses any of the previous syntaxes and additional options specified by one or more `Name, Value` pair arguments. For example, you can specify that columns in the predictor data correspond to observations.

## Input Arguments

**mdl** — Binary, linear classification model

ClassificationLinear model object

Binary, linear classification model, specified as a `ClassificationLinear` model object. You can create a `ClassificationLinear` model object using `fitclinear`.

**x** — Predictor data

full matrix | sparse matrix

Predictor data, specified as an  $n$ -by- $p$  full or sparse matrix. This orientation of `X` indicates that rows correspond to individual observations, and columns correspond to individual predictor variables.

---

**Note** If you orient your predictor matrix so that observations correspond to columns and specify 'ObservationsIn', 'columns', then you might experience a significant reduction in computation time.

---

The length of  $Y$  and the number of observations in  $X$  must be equal.

Data Types: `single` | `double`

### $y$ — Class labels

categorical array | character array | logical vector | vector of numeric values | cell array of character vectors

Class labels, specified as a categorical or character array, logical or numeric vector, or cell array of character vectors.

- The data type of  $Y$  must be the same `Mdl.ClassNames`.
- The distinct classes in  $Y$  must be a subset of `Mdl.ClassNames`.
- If  $Y$  is a character array, then each element must correspond to one row of the array.
- The length of  $Y$  and the number of observations in  $X$  must be equal.

Data Types: `categorical` | `cell` | `char` | `double` | `logical` | `single`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### `ObservationsIn` — Predictor data observation dimension

'rows' (default) | 'columns'

Predictor data observation dimension, specified as the comma-separated pair consisting of 'ObservationsIn' and 'columns' or 'rows'.

---

**Note** If you orient your predictor matrix so that observations correspond to columns and specify 'ObservationsIn', 'columns', then you might experience a significant reduction in optimization-execution time.

---

Data Types: `char`

## Output Arguments

### **m** — Classification margins

numeric column vector | numeric matrix

Classification margins on page 32-3894, returned as a numeric column vector or matrix.

$m$  is  $n$ -by- $L$ , where  $n$  is the number of observations in  $X$  and  $L$  is the number of regularization strengths in  $Mdl$  (that is, `numel(Mdl.Lambda)`).

$m(i, j)$  is the classification margin of observation  $i$  using the trained linear classification model that has regularization strength  $Mdl.Lambda(j)$ .

## Examples

### Estimate Test-Sample Margins

Load the NLP data set.

```
load nlpdata
```

$X$  is a sparse matrix of predictor data, and  $Y$  is a categorical vector of class labels. There are more than two classes in the data.

The models should identify whether the word counts in a web page are from the Statistics and Machine Learning Toolbox™ documentation. So, identify the labels that correspond to the Statistics and Machine Learning Toolbox™ documentation web pages.

```
Ystats = Y == 'stats';
```

Train a binary, linear classification model that can identify whether the word counts in a documentation web page are from the Statistics and Machine Learning Toolbox™ documentation. Specify to hold out 30% of the observations. Optimize the objective function using SpARSA.

```
rng(1); % For reproducibility
CVMdl = fitclinear(X,Ystats,'Solver','sparsa','Holdout',0.30);
CMdl = CVMdl.Trained{1};
```

CVMdl is a ClassificationPartitionedLinear model. It contains the property Trained, which is a 1-by-1 cell array holding a ClassificationLinear model that the software trained using the training set.

Extract the training and test data from the partition definition.

```
trainIdx = training(CVMdl.Partition);
testIdx = test(CVMdl.Partition);
```

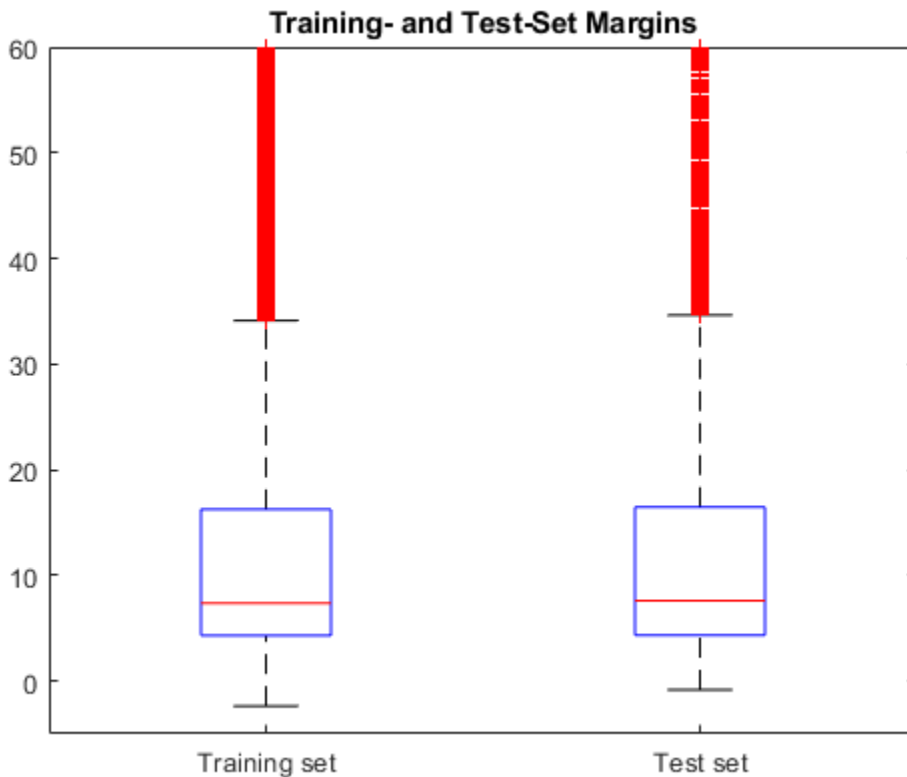
Estimate the training- and test-sample margins.

```
mTrain = margin(CMdl,X(trainIdx,:),Ystats(trainIdx));
mTest = margin(CMdl,X(testIdx,:),Ystats(testIdx));
```

Because there is one regularization strength in CMdl, mTrain and mTest are column vectors with lengths equal to the number of training and test observations, respectively.

Plot both sets of margins using box plots.

```
figure;
boxplot([mTrain; mTest],[zeros(size(mTrain,1),1); ones(size(mTest,1),1)], ...
'Labels',{'Training set','Test set'});
h = gca;
h.YLim = [-5 60];
title 'Training- and Test-Set Margins'
```



The distributions of the margins between the training and test sets appear similar.

### Feature Selection Using Test-Sample Margins

One way to perform feature selection is to compare test-sample margins from multiple models. Based solely on this criterion, the classifier with the larger margins is the better classifier.

Load the NLP data set. Preprocess the data as in “Estimate Test-Sample Margins” on page 32-3886.

```
load nlpdata
Ystats = Y == 'stats';
X = X';
rng(1); % For reproducibility
```

Create a data partition which holds out 30% of the observations for testing.

```
Partition = cvpartition(Ystats,'Holdout',0.30);
testIdx = test(Partition); % Test-set indices
XTest = X(:,testIdx);
YTest = Ystats(testIdx);
```

Partition is a `cvpartition` object that defines the data set partition.

Randomly choose 10% of the predictor variables.

```
p = size(X,1); % Number of predictors
idxPart = randsample(p,ceil(0.1*p));
```

Train two binary, linear classification models: one that uses the all of the predictors and one that uses the random 10%. Optimize the objective function using `SparSA`, and indicate that observations correspond to columns.

```
CVMDL = fitlinear(X,Ystats,'CVPartition',Partition,'Solver','sparsa',...
    'ObservationsIn','columns');
PCVMDL = fitlinear(X(idxPart,:),Ystats,'CVPartition',Partition,'Solver','sparsa',...
    'ObservationsIn','columns');
```

CVMDL and PCVMDL are `ClassificationPartitionedLinear` models.

Extract the trained `ClassificationLinear` models from the cross-validated models.

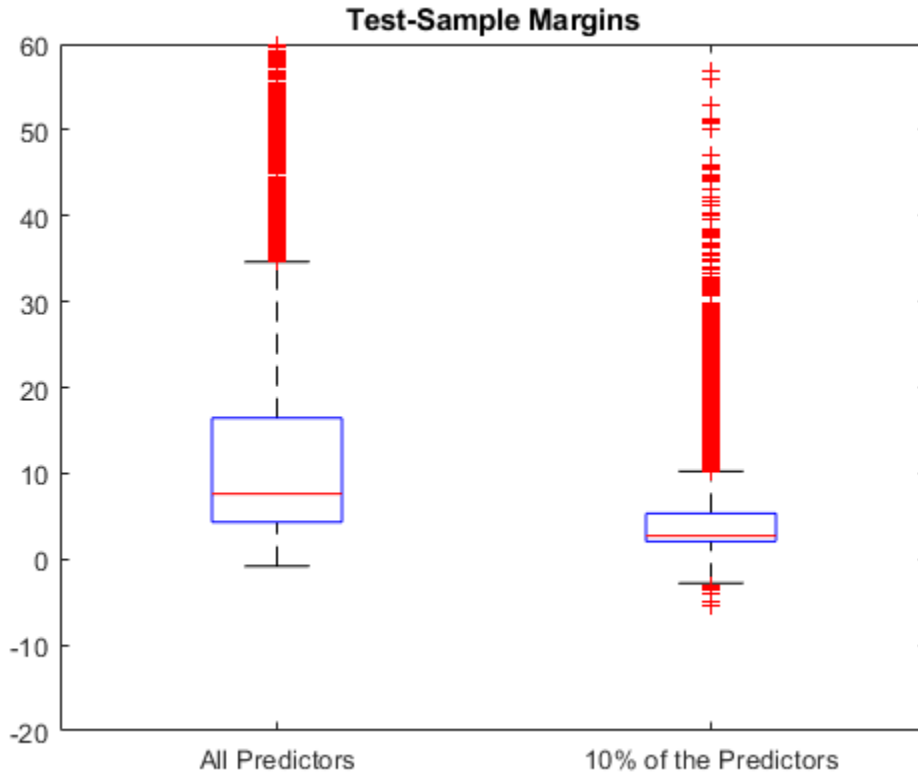
```
CMdl = CVMDL.Trained{1};
PCMdl = PCVMDL.Trained{1};
```

Estimate the test sample margins for each classifier. Plot the distribution of the margins sets using box plots.

```
fullMargins = margin(CMdl,XTest,YTest,'ObservationsIn','columns');
partMargins = margin(PCMdl,XTest(idxPart,:),YTest,...
    'ObservationsIn','columns');

figure;
boxplot([fullMargins partMargins],'Labels',...
    {'All Predictors','10% of the Predictors'});
```

```
h = gca;  
h.YLim = [-20 60];  
title('Test-Sample Margins')
```



The margin distribution of `CMdl` is situated higher than the margin distribution of `PCMdl`.

### Find Good Lasso Penalty Using Margins

To determine a good lasso-penalty strength for a linear classification model that uses a logistic regression learner, compare distributions of test-sample margins.



Load the NLP data set. Preprocess the data as in “Estimate Test-Sample Margins” on page 32-3886.

```
load nlpdata
Ystats = Y == 'stats';
X = X';

Partition = cvpartition(Ystats, 'Holdout', 0.30);
testIdx = test(Partition);
XTest = X(:, testIdx);
YTest = Ystats(testIdx);
```

Create a set of 11 logarithmically-spaced regularization strengths from  $10^{-8}$  through  $10^1$ .

```
Lambda = logspace(-8, 1, 11);
```

Train binary, linear classification models that use each of the regularization strengths. Optimize the objective function using SpARSA. Lower the tolerance on the gradient of the objective function to  $1e-8$ .

```
rng(10); % For reproducibility
CVMdl = fitlinear(X, Ystats, 'ObservationsIn', 'columns', ...
    'CVPartition', Partition, 'Learner', 'logistic', 'Solver', 'sparsa', ...
    'Regularization', 'lasso', 'Lambda', Lambda, 'GradientTolerance', 1e-8)
```

```
CVMdl =
```

```
classreg.learning.partition.ClassificationPartitionedLinear
  CrossValidatedModel: 'Linear'
  ResponseName: 'Y'
  NumObservations: 31572
  KFold: 1
  Partition: [1x1 cvpartition]
  ClassNames: [0 1]
  ScoreTransform: 'none'
```

Extract the trained linear classification model.

```
Mdl = CVMdl.Trained{1}
```

```
Mdl =
```

```
ClassificationLinear
  ResponseName: 'Y'
  ClassNames: [0 1]
  ScoreTransform: 'logit'
    Beta: [34023x11 double]
    Bias: [1x11 double]
    Lambda: [1x11 double]
  Learner: 'logistic'
```

Mdl is a `ClassificationLinear` model object. Because `Lambda` is a sequence of regularization strengths, you can think of `Mdl` as 11 models, one for each regularization strength in `Lambda`.

Estimate the test-sample margins.

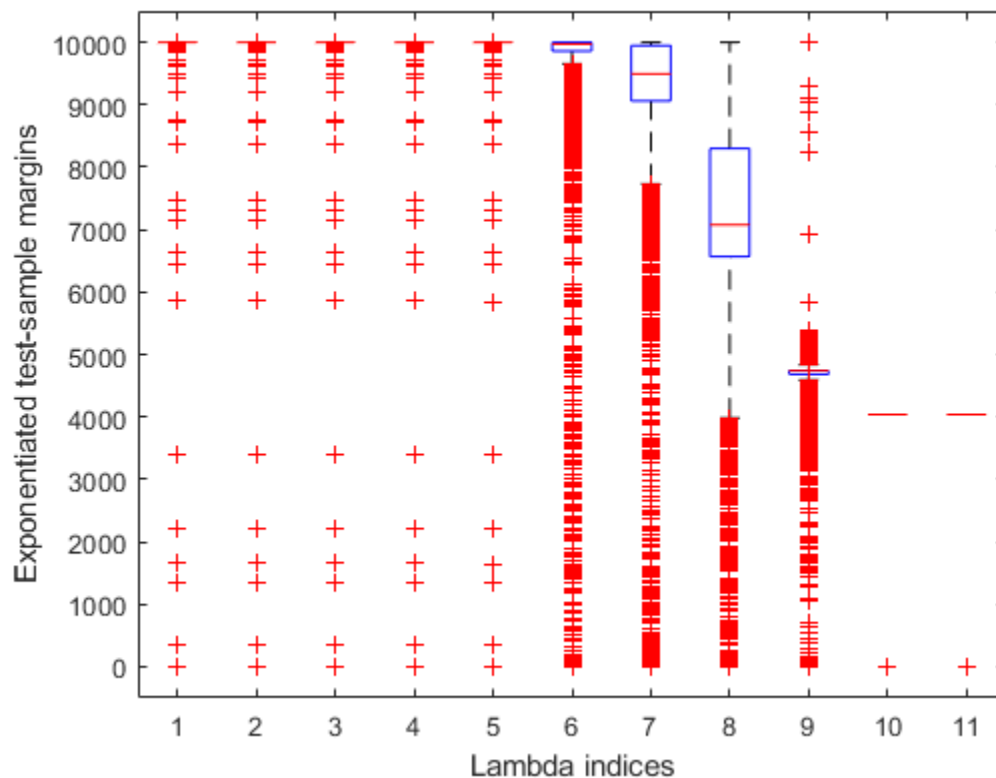
```
m = margin(Mdl,X(:,testIdx),Ystats(testIdx),'ObservationsIn','columns');
size(m)
```

```
ans =
      9471      11
```

Because there are 11 regularization strengths, `m` has 11 columns.

Plot the test-sample margins for each regularization strength. Because logistic regression scores are in  $[0,1]$ , margins are in  $[-1,1]$ . Rescale the margins to help identify the regularization strength that maximizes the margins over the grid.

```
figure;
boxplot(10000.^m)
ylabel('Exponentiated test-sample margins')
xlabel('Lambda indices')
```



Several values of Lambda yield margin distributions that are compacted near **10000<sup>1</sup>**. Higher values of lambda lead to predictor variable sparsity, which is a good quality of a classifier.

Choose the regularization strength that occurs just before the centers of the margin distributions start decreasing.

```
LambdaFinal = Lambda(5);
```

Train a linear classification model using the entire data set and specify the desired regularization strength.

```
MdlFinal = fitclinear(X, Ystats, 'ObservationsIn', 'columns', ...  
    'Learner', 'logistic', 'Solver', 'sparsa', 'Regularization', 'lasso', ...  
    'Lambda', LambdaFinal);
```

To estimate labels for new observations, pass `MdlFinal` and the new data to `predict`.

## Definitions

### Classification Margin

The classification margins for binary classification are, for each observation, the difference between the classification score for the true class and the classification score for the false class.

The software defines the classification margin for binary classification as

$$m = 2yf(x).$$

$x$  is an observation. If the true label of  $x$  is the positive class, then  $y$  is 1, and  $-1$  otherwise.  $f(x)$  is the positive-class classification score for the observation  $x$ . The literature commonly defines the margin as  $m = yf(x)$ .

If the margins are on the same scale, then they serve as a classification confidence measure, i.e., among multiple classifiers, those that yield larger margins are better.

### Classification Score

For linear classification models, the raw classification score for classifying the observation  $x$ , a row vector, into the positive class is defined by

$$f_j(x) = x\beta_j + b_j.$$

For the model with regularization strength  $j$ ,  $\beta_j$  is the estimated column vector of coefficients (the model property `Beta(:, j)`) and  $b_j$  is the estimated, scalar bias (the model property `Bias(j)`).

The raw classification score for classifying  $x$  into the negative class is  $-f(x)$ . The software classifies observations into the class that yields the positive score.

If the linear classification model consists of logistic regression learners, then the software applies the 'logit' score transformation to the raw classification scores (see `ScoreTransform`).

## Extended Capabilities

### Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays” (MATLAB).

### See Also

`ClassificationLinear` | `edge` | `fitclinear` | `predict`

**Introduced in R2016a**

## margin

**Class:** CompactClassificationDiscriminant

Classification margins

## Syntax

```
m = margin(obj, X, Y)
```

## Description

`m = margin(obj, X, Y)` returns the classification margins for the matrix of predictors `X` and class labels `Y`. For the definition, see “Definitions” on page 32-3898.

## Input Arguments

**obj**

Discriminant analysis classifier of class `ClassificationDiscriminant` or `CompactClassificationDiscriminant`, typically constructed with `fitcdiscr`.

**X**

Matrix where each row represents an observation, and each column represents a predictor. The number of columns in `X` must equal the number of predictors in `obj`.

**Y**

Class labels, with the same data type as exists in `obj`. The number of elements of `Y` must equal the number of rows of `X`.

## Output Arguments

**m**

Numeric column vector of length `size(X,1)`. Each entry in `m` represents the margin for the corresponding rows of `X` and (true class) `Y`, computed using `obj`.

## Examples

Compute the classification margin for the Fisher iris data, trained on its first two columns of data, and view the last 10 entries:

```
load fisheriris
X = meas(:,1:2);
obj = fitcdiscr(X,species);
M = margin(obj,X,species);
M(end-10:end)
```

```
ans =
    0.6551
    0.4838
    0.6551
   -0.5127
    0.5659
    0.4611
    0.4949
    0.1024
    0.2787
   -0.1439
   -0.4444
```

The classifier trained on all the data is better:

```
obj = fitcdiscr(meas,species);
M = margin(obj,meas,species);
M(end-10:end)
```

```
ans =
    0.9983
    1.0000
    0.9991
    0.9978
```

```
1.0000
1.0000
0.9999
0.9882
0.9937
1.0000
0.9649
```

## Definitions

### Margin

The classification margin is the difference between the classification score for the true class and maximal classification score for the false classes.

The classification margin is a column vector with the same number of rows as in the matrix  $X$ . A high value of margin indicates a more reliable prediction than a low value.

### Score (discriminant analysis)

For discriminant analysis, the score of a classification is the posterior probability of the classification. For the definition of posterior probability in discriminant analysis, see “Posterior Probability” on page 20-8.

## Extended Capabilities

### Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays” (MATLAB).

### See Also

`ClassificationDiscriminant` | `edge` | `fitcdiscr` | `loss` | `predict`



## Topics

“Discriminant Analysis Classification” on page 20-2

## margin

**Class:** CompactClassificationECOC

Classification margins for multiclass, error-correcting output codes model

### Syntax

```
m = margin(Mdl, tbl, ResponseVarName)
m = margin(Mdl, tbl, Y)
m = margin(Mdl, X, Y)
m = margin(Mdl, ____, Name, Value)
```

### Description

`m = margin(Mdl, tbl, ResponseVarName)` returns the classification margins on page 32-3911 (m) for the trained, multiclass, error-correcting output code (ECOC) model `Mdl` using the predictor data in table `tbl` and the class labels in `tbl.ResponseVarName`. Each row of `tbl` and `ResponseVarName` is an observation.

`m = margin(Mdl, tbl, Y)` returns the classification margins for the trained classifier `Mdl` using the predictor data in table `tbl` and the class labels in vector `Y`. Each row of `tbl` and `Y` is an observation.

`m = margin(Mdl, X, Y)` returns the classification margins for the trained classifier `Mdl` using the predictor data `X` and class labels `Y`. Each row of `X` and `Y` is an observation.

`m = margin(Mdl, ____, Name, Value)` returns the classification margins with additional options specified by one or more `Name, Value` pair arguments, using any of the previous syntaxes. For example, you can specify a decoding scheme, binary learner loss function, or verbosity level.

## Input Arguments

### **mdl** — Full or compact, multiclass ECOC model

`ClassificationECOC` model object | `CompactClassificationECOC` model object

Full or compact, multiclass ECOC model, specified as a `ClassificationECOC` or `CompactClassificationECOC` model object.

To create a full or compact ECOC model, see `ClassificationECOC` or `CompactClassificationECOC`.

### **tbl** — Sample data

table

Sample data, specified as a table. Each row of `tbl` corresponds to one observation, and each column corresponds to one predictor variable. Optionally, `tbl` can contain additional columns for the response variable and observation weights. `tbl` must contain all the predictors used to train `Mdl`. Multi-column variables and cell arrays other than cell arrays of character vectors are not allowed.

If you trained `Mdl` using sample data contained in a table, then the input data for this method must also be in a table.

---

**Note** If `Mdl.BinaryLearners` contains linear classification models (that is, `ClassificationLinear` model objects), then you cannot specify sample data in a table. Instead, pass a full or sparse matrix (`X`) and class labels (`Y`).

---

Data Types: table

### **ResponseVarName** — Response variable name

name of a variable in `tbl`

Response variable name, specified as the name of a variable in `tbl`.

You must specify `ResponseVarName` as a character vector. For example, if the response variable `y` is stored as `tbl.y`, then specify it as `'y'`. Otherwise, the software treats all columns of `tbl`, including `y`, as predictors when training the model.

The response variable must be a categorical or character array, logical or numeric vector, or cell array of character vectors. If the response variable is a character array, then each element must correspond to one row of the array.

Data Types: `char`

### **x — Predictor data**

numeric matrix

Predictor data, specified as a numeric matrix.

Each row of  $X$  corresponds to one observation, and each column corresponds to one variable. The variables composing the columns of  $X$  should be the same as the variables that trained the `Mdl` classifier.

The length of  $Y$  and the number of rows of  $X$  must be equal.

Data Types: `double` | `single`

### **y — Class labels**

categorical array | character array | logical vector | vector of numeric values | cell array of character vectors

Class labels, specified as a categorical or character array, logical or numeric vector, or cell array of character vectors.  $Y$  must be the same as the data type of `Mdl.ClassNames`.

The length of  $Y$  and the number of rows of  $X$  must be equal.

## **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### **BinaryLoss — Binary learner loss function**

'hamming' | 'linear' | 'logit' | 'exponential' | 'binodeviance' | 'hinge' | 'quadratic' | function handle

Binary learner loss function, specified as the comma-separated pair consisting of 'BinaryLoss' and a built-in, loss-function name or function handle.

- This table contains names and descriptions of the built-in functions, where  $y_j$  is a class label for a particular binary learner (in the set  $\{-1, 1, 0\}$ ),  $s_j$  is the score for observation  $j$ , and  $g(y_j, s_j)$  is the binary loss formula.

Value	Description	Score Domain	$g(y_j, s_j)$
'binodeviance'	Binomial deviance	$(-\infty, \infty)$	$\log[1 + \exp(-2y_j s_j)] / [2\log(2)]$
'exponential'	Exponential	$(-\infty, \infty)$	$\exp(-y_j s_j) / 2$
'hamming'	Hamming	$[0, 1]$ or $(-\infty, \infty)$	$[1 - \text{sign}(y_j s_j)] / 2$
'hinge'	Hinge	$(-\infty, \infty)$	$\max(0, 1 - y_j s_j) / 2$
'linear'	Linear	$(-\infty, \infty)$	$(1 - y_j s_j) / 2$
'logit'	Logistic	$(-\infty, \infty)$	$\log[1 + \exp(-y_j s_j)] / [2\log(2)]$
'quadratic'	Quadratic	$[0, 1]$	$[1 - y_j(2s_j - 1)]^2 / 2$

The software normalizes the binary losses such that the loss is 0.5 when  $y_j = 0$ . Also, the software calculates the mean binary loss for each class.

- For a custom binary loss function, e.g., `customFunction`, specify its function handle `'BinaryLoss', @customFunction`.

`customFunction` should have this form:

```
bLoss = customFunction(M, s)
```

where:

- `M` is the  $K$ -by- $L$  coding matrix stored in `Mdl.CodingMatrix`.
- `s` is the 1-by- $L$  row vector of classification scores.
- `bLoss` is the classification loss. This scalar aggregates the binary losses for every learner in a particular class. For example, you can use the mean binary loss to aggregate the loss over the learners for each class.
- $K$  is the number of classes.
- $L$  is the number of binary learners.

For an example of passing a custom binary loss function, see “Predict Test-Sample Labels of ECOC Models Using Custom Binary Loss Function” on page 32-4933.

By default, if all binary learners are:

- SVMs or linear classification models of SVM learners, then `BinaryLoss` is 'hinge'
- Ensembles trained by `AdaboostM1` or `GentleBoost`, then `BinaryLoss` is 'exponential'
- Ensembles trained by `LogitBoost`, then `BinaryLoss` is 'binodeviance'
- Linear classification models of logistic regression learners, or you specify to predict class posterior probabilities (i.e., set 'FitPosterior', 1 in `fitcecoc`), then `BinaryLoss` is 'quadratic'

Otherwise, the default `BinaryLoss` is 'hamming'. To check the default value, use dot notation to display the `BinaryLoss` property of the trained model at the command line.

Example: `'BinaryLoss', 'binodeviance'`

Data Types: `char` | `function_handle`

#### **Decoding — Decoding scheme**

`'lossweighted'` (default) | `'lossbased'`

Decoding scheme that aggregates the binary losses on page 32-4941, specified as the comma-separated pair consisting of 'Decoding' and 'lossweighted' or 'lossbased'.

Example: `'Decoding', 'lossbased'`

Data Types: `char`

#### **ObservationsIn — Predictor data observation dimension**

`'rows'` (default) | `'columns'`

Predictor data observation dimension, specified as the comma-separated pair consisting of 'ObservationsIn' and 'columns' or 'rows'. `Mdl.BinaryLearners` must contain linear classification models.

---

**Note** If you orient your predictor matrix so that observations correspond to columns and specify 'ObservationsIn', 'columns', you can experience a significant reduction in execution time.

---

Data Types: `char`

**Options — Estimation options**

[] (default) | structure array returned by `statset`

Estimation options, specified as the comma-separated pair consisting of 'Options' and a structure array returned by `statset`.

To invoke parallel computing:

- You need a Parallel Computing Toolbox license.
- Specify 'Options', `statset('UseParallel',1)`.

**verbose — Verbosity level**

0 (default) | 1

Verbosity level, specified as the comma-separated pair consisting of 'Verbose' and 0 or 1. Verbose controls the amount of diagnostic messages that the software displays in the Command Window.

If `Verbose` is 0, then the software does not display diagnostic messages. Otherwise, the software displays diagnostic messages.

Example: 'Verbose',1

Data Types: `single` | `double`

## Output Arguments

**m — Classification margins**

numeric column vector | numeric matrix

Classification margins on page 32-3911, returned as a numeric column vector.

`m` has the same length as `Y`. The software estimates each entry of `m` using the trained ECOC model `Mdl`, the corresponding row of `X`, and the true class label `Y`.

If `Mdl.BinaryLearners` contains linear classification models, then `m` is an  $n$ -by- $L$  vector, where  $n$  is the number of observations in `X` and  $L$  is the number of regularization strengths in the linear classification models (i.e., `numel(Mdl.BinaryLearners{1}.Lambda)`). `m(i,j)` is the margin of observation  $i$  for the model trained using regularization strength `Mdl.BinaryLearners{1}.Lambda(j)`.

Otherwise,  $m$  is a column vector of length  $n$ .

## Examples

### Estimate Test-Sample Classification Margins of ECOC Models

Load Fisher's iris data set.

```
load fisheriris
X = meas;
Y = categorical(species);
classOrder = unique(Y); % Class order
rng(1); % For reproducibility
```

Train an ECOC model using SVM binary classifiers, and specify a 30% holdout sample. It is good practice to standardize the predictors and define the class order. Specify to standardize the predictors using an SVM template.

```
t = templateSVM('Standardize',1);
CVMdl = fitcecoc(X,Y,'Holdout',0.30,'Learners',t,'ClassNames',classOrder);
CMdl = CVMdl.Trained{1}; % Extract trained, compact classifier
testInds = test(CVMdl.Partition); % Extract the test indices
XTest = X(testInds,:);
YTest = Y(testInds,:);
```

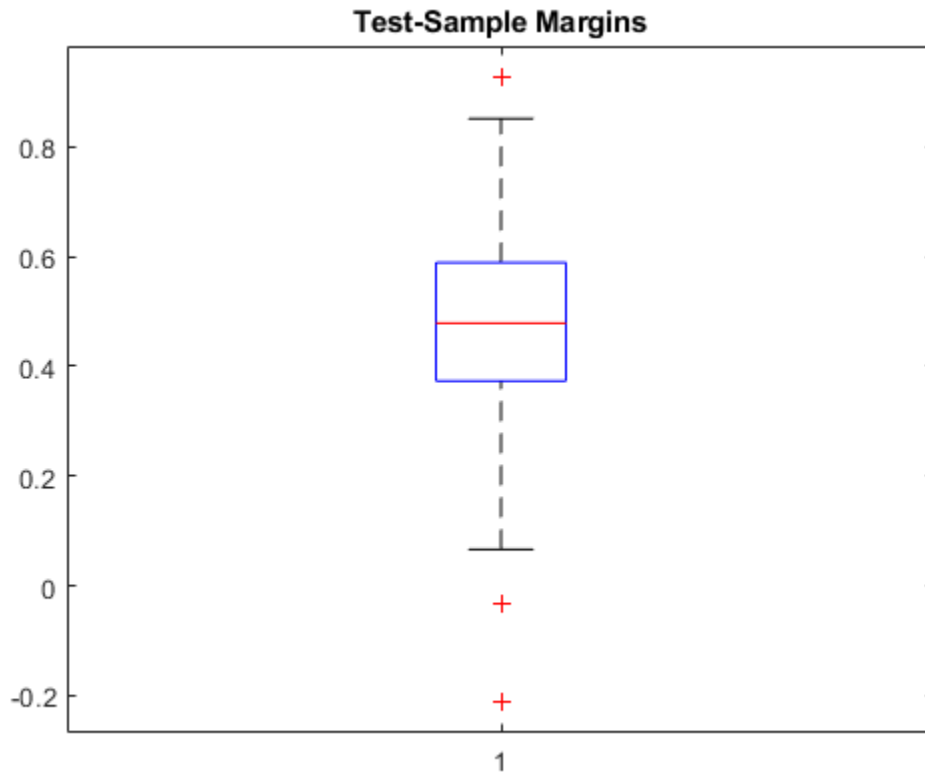
CVMdl is a ClassificationPartitionedECOC model. It contains the property Trained, which is a 1-by-1 cell array holding a CompactClassificationECOC model that the software trained using the training set.

Estimate the test-sample classification margins. Display the distribution of the margins using a boxplot.

```
m = margin(CMdl,XTest,YTest);

figure;
boxplot(m);
title 'Test-Sample Margins'
```





An observation margin is the positive-class, negated loss minus the maximum negative-class, negated loss. Classifiers that yield relatively large margins are desirable.

### Select ECOC Model Features by Examining Test-Sample Margins

The classifier margins measure, for each observation, the difference between the positive-class, negated loss score and the maximal negative-class, negated loss. One way to perform feature selection is to compare test-sample margins from multiple models. Based solely on this criterion, the model with the highest margins is the best model.

Load Fisher's iris data set.

```
load fisheriris
X = meas;
Y = categorical(species);
classOrder = unique(Y); % Class order
rng(1); % For reproducibility
```

Partition the data set into training and test sets. Specify a 30% holdout sample for testing.

```
Partition = cvpartition(Y, 'Holdout', 0.30);
testInds = test(Partition); % Indices for the test set
XTest = X(testInds, :);
YTest = Y(testInds, :);
```

Define these two data sets:

- `fullX` contains all four predictors.
- `partX` contains the sepal measurements.

```
fullX = X;
partX = X(:, 1:2);
```

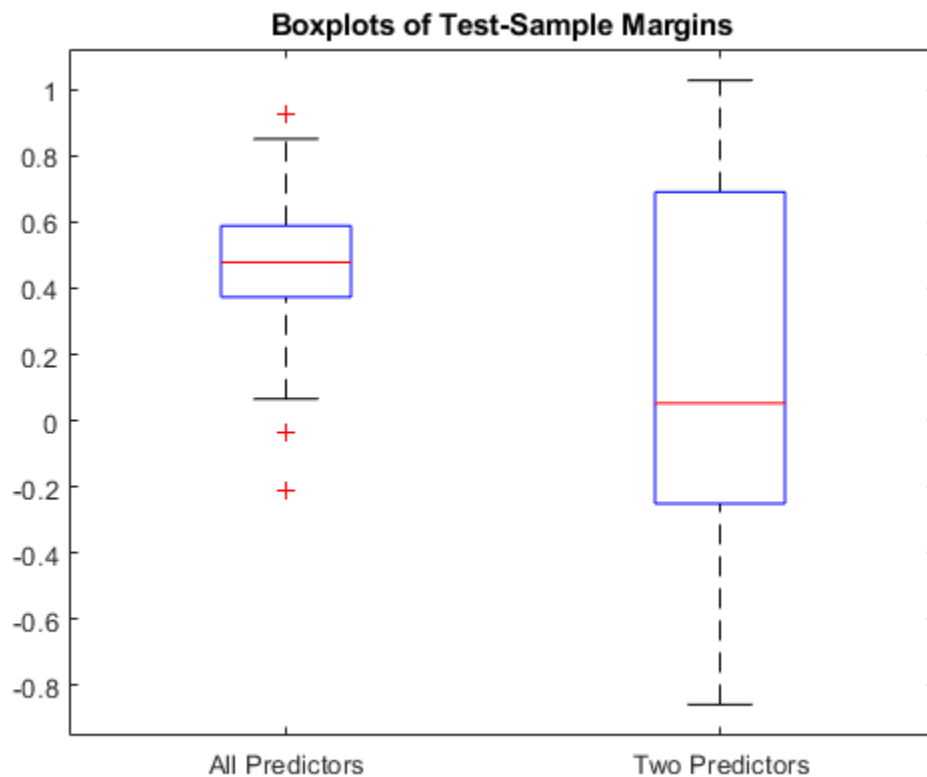
Train an ECOC model using SVM binary classifiers for each predictor set, and specify the partition definition. It is good practice to define the class order. Specify to standardize the predictors using an SVM template.

```
t = templateSVM('Standardize', 1);
CVMdl = fitcecoc(fullX, Y, 'CVPartition', Partition, 'Learners', t, ...
    'ClassNames', classOrder);
PCVMdl = fitcecoc(partX, Y, 'CVPartition', Partition, 'Learners', t, ...
    'ClassNames', classOrder);
CMdl = CVMdl.Trained{1};
PCMdl = PCVMdl.Trained{1};
```

Estimate the test-sample margins for each classifier. For each model, display the distribution of the margins using a boxplot.

```
fullMargins = margin(CMdl, XTest, YTest);
partMargins = margin(PCMdl, XTest(:, 3:4), YTest);

figure;
boxplot([fullMargins partMargins], 'Labels', {'All Predictors', 'Two Predictors'});
title('Boxplots of Test-Sample Margins')
```



The margin distribution of  $CMd1$  is situated higher, and with less variability than the margin distribution of  $PCMd1$ .

- “Quick Start Parallel Computing for Statistics and Machine Learning Toolbox” on page 30-2

## Definitions

### Binary Loss

A binary loss is a function of the class and classification score that determines how well a binary learner classifies an observation into the class.

Let:

- $m_{kj}$  be element  $(k,j)$  of the coding design matrix  $M$  (i.e., the code corresponding to class  $k$  of binary learner  $j$ )
- $s_j$  be the score of binary learner  $j$  for an observation
- $g$  be the binary loss function
- $\hat{k}$  be the predicted class for the observation

In loss-based decoding [15], the class producing the minimum sum of the binary losses over binary learners determines the predicted class of an observation, that is,

$$\hat{k} = \operatorname{argmin}_k \sum_{j=1}^L |m_{kj}| g(m_{kj}, s_j).$$

In loss-weighted decoding [15], the class producing the minimum average of the binary losses over binary learners determines the predicted class of an observation, that is,

$$\hat{k} = \operatorname{argmin}_k \frac{\sum_{j=1}^L |m_{kj}| g(m_{kj}, s_j)}{\sum_{j=1}^L |m_{kj}|}.$$

Allwein et al. [2] suggest that loss-weighted decoding improves classification accuracy by keeping loss values for all classes in the same dynamic range.

This table summarizes the supported loss functions, where  $y_j$  is a class label for a particular binary learner (in the set  $\{-1,1,0\}$ ),  $s_j$  is the score for observation  $j$ , and  $g(y_j, s_j)$ .

Value	Description	Score Domain	$g(y_j, s_j)$
'binodeviance'	Binomial deviance	$(-\infty, \infty)$	$\log[1 + \exp(-2y_j s_j)] / [2\log(2)]$

Value	Description	Score Domain	$g(y, s_j)$
'exponential'	Exponential	$(-\infty, \infty)$	$\exp(-y_j s_j)/2$
'hamming'	Hamming	$[0, 1]$ or $(-\infty, \infty)$	$[1 - \text{sign}(y_j s_j)]/2$
'hinge'	Hinge	$(-\infty, \infty)$	$\max(0, 1 - y_j s_j)/2$
'linear'	Linear	$(-\infty, \infty)$	$(1 - y_j s_j)/2$
'logit'	Logistic	$(-\infty, \infty)$	$\log[1 + \exp(-y_j s_j)]/[2\log(2)]$
'quadratic'	Quadratic	$[0, 1]$	$[1 - y_j(2s_j - 1)]^2/2$

The software normalizes the binary losses such that the loss is 0.5 when  $y_j = 0$ , and aggregates using the average of the binary learners [2].

Do not confuse the binary loss with the overall classification loss (specified by the `LossFun` name-value pair argument of `predict` and `loss`), e.g., classification error, which measures how well an ECOC classifier performs as a whole.

## Classification Margin

The classification margins are, for each observation, the difference between the negative loss for the true class and maximal negative loss among the false classes. If the margins are on the same scale, then they serve as a classification confidence measure, i.e., among multiple classifiers, those that yield larger margins are better.

## Tip

To compare margins or edges of several classifiers, use template objects to specify a common score transform function among the classifiers when you train them using `fitcecoc`.

## Algorithms

If you trained `Mdl` specifying to standardize the predictor data, then the software standardizes the columns of `X` using the corresponding means and standard deviations that the software stored in `Mdl.BinaryLearner{j}.Mu` and `Mdl.BinaryLearner{j}.Sigma` for learner `j`.

## References

- [1] Allwein, E., R. Schapire, and Y. Singer. “Reducing multiclass to binary: A unifying approach for margin classifiers.” *Journal of Machine Learning Research*. Vol. 1, 2000, pp. 113–141.
- [2] Escalera, S., O. Pujol, and P. Radeva. “On the decoding process in ternary error-correcting output codes.” *IEEE Transactions on Pattern Analysis and Machine Intelligence*. Vol. 32, Issue 7, 2010, pp. 120–134.
- [3] Escalera, S., O. Pujol, and P. Radeva. “Separability of ternary codes for sparse designs of error-correcting output codes.” *Pattern Recogn.* Vol. 30, Issue 3, 2009, pp. 285–297.

## Extended Capabilities

### Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays” (MATLAB).

### See Also

`ClassificationECOC` | `ClassificationLinear` | `CompactClassificationECOC` | `edge` | `fitcecoc` | `predict` | `resubMargin`

### Topics

“Quick Start Parallel Computing for Statistics and Machine Learning Toolbox” on page 30-2  
“Reproducibility in Parallel Statistical Computations” on page 30-14  
“Concepts of Parallel Computing in Statistics and Machine Learning Toolbox” on page 30-8

# margin

**Class:** CompactClassificationEnsemble

Classification margins

## Syntax

```
M = margin(ens, tbl, ResponseVarName)
M = margin(ens, tbl, Y)
M = margin(ens, X, Y)
M = margin( ___ Name, Value)
```

## Description

`M = margin(ens, tbl, ResponseVarName)` returns the classification margin for the predictions of `ens` on data `tbl`, when the true classifications are `tbl.ResponseVarName`.

`M = margin(ens, tbl, Y)` returns the classification margin for the predictions of `ens` on data `tbl`, when the true classifications are `Y`.

`M = margin(ens, X, Y)` returns the classification margin for the predictions of `ens` on data `X`, when the true classifications are `Y`.

`M = margin( ___ Name, Value)` calculates margin with additional options specified by one or more `Name, Value` pair arguments, using any of the previous syntaxes.

## Input Arguments

### **ens**

Classification ensemble created with `fitcensemble`, or a compact classification ensemble created with `compact`.

**tbl**

Sample data, specified as a table. Each row of `tbl` corresponds to one observation, and each column corresponds to one predictor variable. `tbl` must contain all of the predictors used to train the model. Multi-column variables and cell arrays other than cell arrays of character vectors are not allowed.

If you trained `ens` using sample data contained in a table, then the input data for this method must also be in a table.

**ResponseVarName**

Response variable name, specified as the name of a variable in `tbl`. The response variable must be a numeric vector.

You must specify `ResponseVarName` as a character vector. For example, if the response variable `Y` is stored as `tbl.Y`, then specify it as `'Y'`. Otherwise, the software treats all columns of `tbl`, including `Y`, as predictors when training the model.

**X**

Matrix of data to classify. Each row of `X` represents one observation, and each column represents one predictor. `X` must have the same number of columns as the data used to train `ens`. `X` should have the same number of rows as the number of elements in `Y`.

If you trained `ens` using sample data contained in a matrix, then the input data for this method must also be in a matrix.

**Y**

Classification of `X`. `Y` should be of the same type as the classification used to train `ens`, and its number of elements should equal the number of rows of `tbl` or `X`.

**Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.



**learners**

Indices of weak learners in the ensemble ranging from 1 to `ens.NumTrained`. `oobEdge` uses only these learners for calculating loss.

**Default:** `1:NumTrained`

**UseObsForLearner**

A logical matrix of size N-by-T, where:

- N is the number of rows of X.
- T is the number of weak learners in `ens`.

When `UseObsForLearner(i, j)` is true, learner j is used in predicting the class of row i of X.

**Default:** `true(N, T)`

## Output Arguments

**M**

A numeric column vector with the same number of rows as `tbl` or X. Each row of M gives the classification margin for that row of `tbl` or X.

## Examples

**Find Classification Margin**

Find the margin for classifying an average flower from the `fisheriris` data as 'versicolor'.

Load the Fisher iris data set.

```
load fisheriris
```

Train an ensemble of 100 boosted classification trees using `AdaBoostM2`.

```
t = templateTree('MaxNumSplits',1); % Weak learner template tree object
ens = fitcensemble(meas,species,'Method','AdaBoostM2','Learners',t);
```

Classify an average flower and find the classification margin.

```
flower = mean(meas);
predict(ens,flower)
```

```
ans = 1x1 cell array
    {'versicolor'}
```

```
margin(ens,flower,'versicolor')
```

```
ans = 3.2140
```

## Definitions

### Margin

The classification margin is the difference between the classification score for the true class and maximal classification score for the false classes. Margin is a column vector with the same number of rows as in the matrix  $X$ .

### Score (ensemble)

For ensembles, a classification score represents the confidence of a classification into a class. The higher the score, the higher the confidence.

Different ensemble algorithms have different definitions for their scores. Furthermore, the range of scores depends on ensemble type. For example:

- AdaBoostM1 scores range from  $-\infty$  to  $\infty$ .
- Bag scores range from 0 to 1.

## Extended Capabilities

### Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays” (MATLAB).

### See Also

`edge` | `loss` | `predict`

## margin

**Class:** CompactClassificationNaiveBayes

Classification margins for naive Bayes classifiers

### Syntax

```
m = margin(Mdl, tbl, ResponseVarName)
m = margin(Mdl, tbl, Y)
m = margin(Mdl, X, Y)
```

### Description

`m = margin(Mdl, tbl, ResponseVarName)` returns the classification margins on page 32-3924 (m) for the trained naive Bayes classifier `Mdl` using the predictor data in table `tbl` and the class labels in `tbl.ResponseVarName`.

`m = margin(Mdl, tbl, Y)` returns the classification margins (m) for the trained naive Bayes classifier `Mdl` using the predictor data in table `tbl` and the class labels in vector `Y`.

`m = margin(Mdl, X, Y)` returns the classification margins (m) for the trained naive Bayes classifier `Mdl` using the predictor data `X` and class labels `Y`.

### Input Arguments

**mdl** — Naive Bayes classifier

ClassificationNaiveBayes model | CompactClassificationNaiveBayes model

Naive Bayes classifier, specified as a `ClassificationNaiveBayes` model or `CompactClassificationNaiveBayes` model returned by `fitcnb` or `compact`, respectively.

**tbl** — Sample data

table

Sample data, specified as a table. Each row of `tbl` corresponds to one observation, and each column corresponds to one predictor variable. Optionally, `tbl` can contain additional columns for the response variable and observation weights. `tbl` must contain all the predictors used to train `Mdl`. Multi-column variables and cell arrays other than cell arrays of character vectors are not allowed.

If you trained `Mdl` using sample data contained in a table, then the input data for this method must also be in a table.

Data Types: `table`

### **ResponseVarName** — Response variable name

name of a variable in `tbl`

Response variable name, specified as the name of a variable in `tbl`.

You must specify `ResponseVarName` as a character vector. For example, if the response variable `y` is stored as `tbl.y`, then specify it as `'y'`. Otherwise, the software treats all columns of `tbl`, including `y`, as predictors when training the model.

The response variable must be a categorical or character array, logical or numeric vector, or cell array of character vectors. If the response variable is a character array, then each element must correspond to one row of the array.

### **x** — Predictor data

numeric matrix

Predictor data, specified as a numeric matrix.

Each row of `X` corresponds to one observation (also known as an instance or example), and each column corresponds to one variable (also known as a feature). The variables making up the columns of `X` should be the same as the variables that trained `Mdl`.

The length of `Y` and the number of rows of `X` must be equal.

Data Types: `double` | `single`

### **y** — Class labels

categorical array | character array | logical vector | vector of numeric values | cell array of character vectors

Class labels, specified as a categorical or character array, logical or numeric vector, or cell array of character vectors. `Y` must be the same as the data type of `Mdl.ClassNames`.

The length of `Y` and the number of rows of `tbl` or `X` must be equal.

## Output Arguments

### `m` — Classification margins

numeric vector

Classification margins on page 32-3924, returned as a numeric vector.

`m` has the same length equal to `size(X,1)`. Each entry of `m` is the classification margin of the corresponding observation (row) of `X` and element of `Y`.

## Examples

### Estimate Test-Sample Classification Margins of Naive Bayes Classifiers

Load Fisher's iris data set.

```
load fisheriris
X = meas;      % Predictors
Y = species;  % Response
rng(1);
```

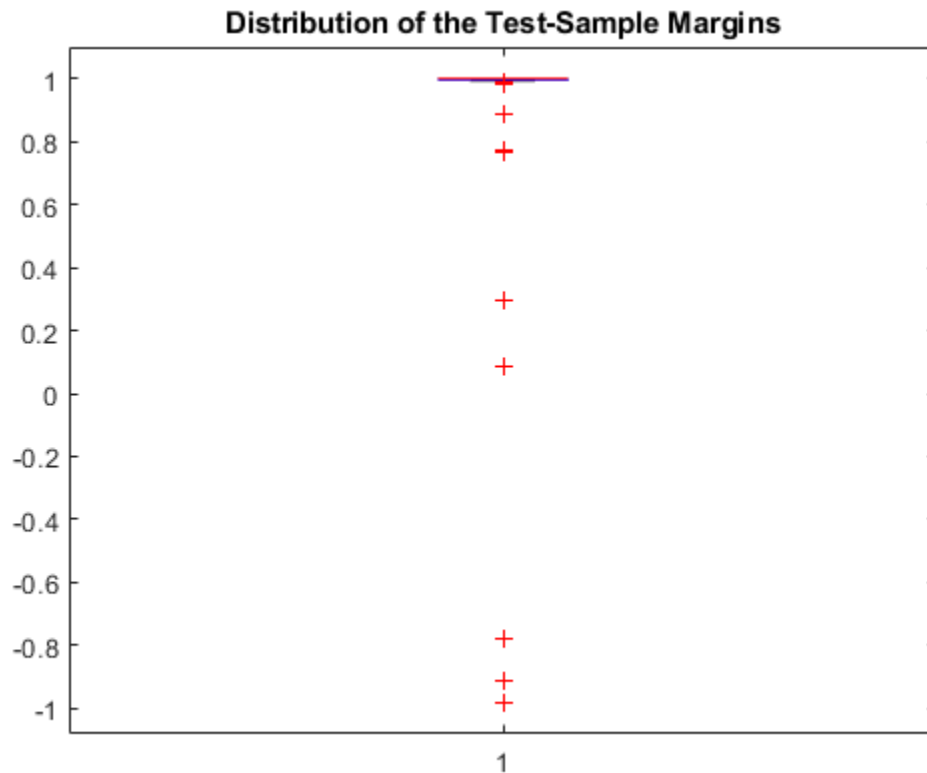
Train a naive Bayes classifier. Specify a 30% holdout sample for testing. It is good practice to specify the class order. Assume that each predictor is conditionally normally distributed given its label.

```
CVMDL = fitcnb(X,Y,'Holdout',0.30,...
    'ClassNames',{'setosa','versicolor','virginica'});
CMDL = CVMDL.Trained{1}; ...
    % Extract the trained, compact classifier
testInds = test(CVMDL.Partition); % Extract the test indices
XTest = X(testInds,:);
YTest = Y(testInds);
```

`CVMDL` is a `ClassificationPartitionedModel` classifier. It contains the property `Trained`, which is a 1-by-1 cell array holding a `CompactClassificationNaiveBayes` classifier that the software trained using the training set.

Estimate the test sample classification margins. Display the distribution of the margins using a boxplot.

```
m = margin(CMdl,XTest,YTest);  
  
figure;  
boxplot(m);  
title 'Distribution of the Test-Sample Margins';
```



An observation margin is the observed true class score minus the maximum false class score among all scores in the respective class. Classifiers that yield relatively large margins are desirable.

### Select Naive Bayes Classifier Features by Examining Test Sample Margins

The classifier margins measure, for each observation, the difference between the true class observed score and the maximal false class score for a particular class. One way to perform feature selection is to compare test sample margins from multiple models. Based solely on this criterion, the model with the highest margins is the best model.

Load Fisher's iris data set.

```
load fisheriris
X = meas;      % Predictors
Y = species;  % Response
rng(1);
```

Partition the data set into training and test sets. Specify a 30% holdout sample for testing.

```
Partition = cvpartition(Y, 'Holdout', 0.30);
testInds = test(Partition); % Indices for the test set
XTest = X(testInds, :);
YTest = Y(testInds);
```

Partition defines the data set partition.

Define these two data sets:

- `fullX` contains all predictors.
- `partX` contains the last 2 predictors.

```
fullX = X;
partX = X(:, 3:4);
```

Train naive Bayes classifiers for each predictor set. Specify the partition definition.

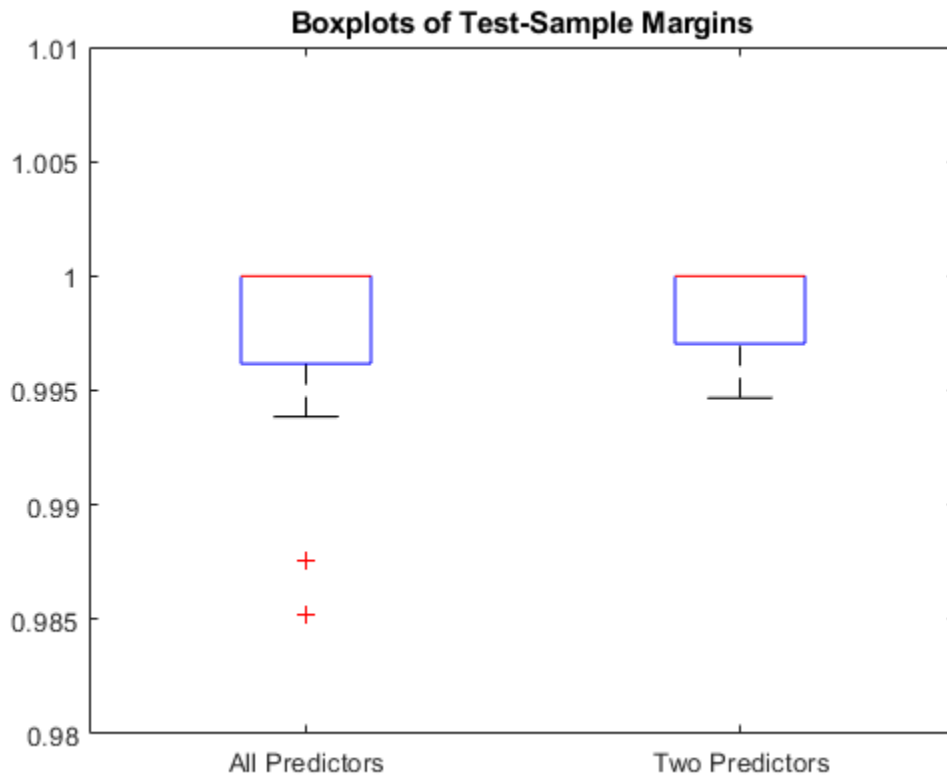
```
FCVMD1 = fitcnb(fullX, Y, 'CVPartition', Partition);
PCVMD1 = fitcnb(partX, Y, 'CVPartition', Partition);
FCMD1 = FCVMD1.Trained{1};
PCMD1 = PCVMD1.Trained{1};
```

`FullCVMD1` and `PartCVMD1` are `ClassificationPartitionedModel` classifiers. They contain the property `Trained`, which is a 1-by-1 cell array holding a `CompactClassificationNaiveBayes` classifier that the software trained using the training set.



Estimate the test sample margins for each classifier. Display the distributions of the margins for each model using boxplots.

```
fullM = margin(FCmdl,XTest,YTest);  
partM = margin(PCmdl,XTest(:,3:4),YTest);  
  
figure;  
boxplot([fullM partM], 'Labels', {'All Predictors', 'Two Predictors'})  
h = gca;  
h.YLim = [0.98 1.01]; % Modify axis to see boxes.  
title 'Boxplots of Test-Sample Margins';
```



The margins have a similar distribution, but `PCmdl` is less complex.

## Definitions

### Classification Edge

The classification edge is the weighted mean of the classification margins.

If you supply weights, then the software normalizes them to sum to the prior probability of their respective class. The software uses the normalized weights to compute the weighted mean.

One way to choose among multiple classifiers, e.g., to perform feature selection, is to choose the classifier that yields the highest edge.

### Classification Margin

The classification margins are, for each observation, the difference between the score for the true class and maximal score for the false classes. Provided that they are on the same scale, margins serve as a classification confidence measure, i.e., among multiple classifiers, those that yield larger margins are better.

### Posterior Probability

The posterior probability is the probability that an observation belongs in a particular class, given the data.

For naive Bayes, the posterior probability that a classification is  $k$  for a given observation  $(x_1, \dots, x_P)$  is

$$\hat{P}(Y = k | x_1, \dots, x_P) = \frac{P(X_1, \dots, X_P | y = k)\pi(Y = k)}{P(X_1, \dots, X_P)},$$

where:

- $P(X_1, \dots, X_P | y = k)$  is the conditional joint density of the predictors given they are in class  $k$ . `Mdl.DistributionNames` stores the distribution names of the predictors.

- $\pi(Y = k)$  is the class prior probability distribution. `Mdl.Prior` stores the prior distribution.
- $P(X_1, \dots, X_P)$  is the joint density of the predictors. The classes are discrete, so

$$P(X_1, \dots, X_P) = \sum_{k=1}^K P(X_1, \dots, X_P | y = k) \pi(Y = k).$$

## Prior Probability

The prior probability is the believed relative frequency that observations from a class occur in the population for each class.

## Score

The naive Bayes score is the class posterior probability given the observation.

# Extended Capabilities

## Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays” (MATLAB).

## See Also

`ClassificationNaiveBayes` | `CompactClassificationNaiveBayes` | `edge` | `fitcnb` | `loss` | `predict`

## Topics

“Naive Bayes Classification” on page 21-2

## margin

**Class:** CompactClassificationSVM

Classification margins for support vector machine classifiers

## Syntax

```
m = margin(SVMModel, TBL, ResponseVarName)
```

```
m = margin(SVMModel, TBL, Y)
```

```
m = margin(SVMModel, X, Y)
```

## Description

`m = margin(SVMModel, TBL, ResponseVarName)` returns the classification margins on page 32-3931 (`m`) for the trained support vector machine (SVM) classifier `SVMModel` using the sample data in table `TBL` and class labels in `TBL.ResponseVarName`.

`m = margin(SVMModel, TBL, Y)` returns the classification margins (`m`) for the trained support vector machine (SVM) classifier `SVMModel` using the sample data in table `TBL` and class labels in `Y`.

`m = margin(SVMModel, X, Y)` returns the classification margins for `SVMModel` using the predictor data in matrix `X` and class labels `Y`.

## Input Arguments

**SVMModel** — SVM classification model

ClassificationSVM model object | CompactClassificationSVM model object

SVM classification model, specified as a `ClassificationSVM` model object or `CompactClassificationSVM` model object returned by `fitcsvm` or `compact`, respectively.

**TBL — Sample data**

table

Sample data, specified as a table. Each row of TBL corresponds to one observation, and each column corresponds to one predictor variable. Optionally, TBL can contain additional columns for the response variable and observation weights. TBL must contain all of the predictors used to train `SVMMModel`. Multi-column variables and cell arrays other than cell arrays of character vectors are not allowed.

If TBL contains the response variable used to train `SVMMModel`, then you do not need to specify `ResponseVarName` or `Y`.

If you trained `SVMMModel` using sample data contained in a table, then the input data for this method must also be in a table.

Data Types: table

**x — Predictor data**

numeric matrix

Predictor data, specified as a numeric matrix.

Each row of X corresponds to one observation (also known as an instance or example), and each column corresponds to one variable (also known as a feature). The variables making up the columns of X must be the same as the variables that trained the `SVMMModel` classifier.

The length of Y and the number of rows of X must be equal.

If you set `'Standardize', true` in `fitcsvm` to train `SVMMModel`, then the software standardizes the columns of X using the corresponding means in `SVMMModel.Mu` and standard deviations in `SVMMModel.Sigma`.

Data Types: double | single

**ResponseVarName — Response variable name**

name of a variable in TBL

Response variable name, specified as the name of a variable in TBL. If TBL contains the response variable used to train `SVMMModel`, then you do not need to specify `ResponseVarName`.

If you specify `ResponseVarName`, then you must do so as a character vector. For example, if the response variable is stored as `TBL.Response`, then specify it as `'Response'`. Otherwise, the software treats all columns of `TBL`, including `TBL.Response`, as predictors.

The response variable must be a categorical or character array, logical or numeric vector, or cell array of character vectors. If the response variable is a character array, then each element must correspond to one row of the array.

#### **Y — Class labels**

categorical array | character array | logical vector | vector of numeric values | cell array of character vectors

Class labels, specified as a categorical or character array, logical or numeric vector, or cell array of character vectors. `Y` must be the same as the data type of `SVMMODEL.ClassNames`.

The length of `Y` must equal the number of rows of `TBL` or `X` must be equal.

## Output Arguments

#### **m — Classification margins**

numeric vector

Classification margins, returned as a numeric vector.

`m` has the same length as `Y`. The software estimates each entry of `m` using the trained SVM classifier `SVMMODEL`, the corresponding row of `X`, and the true class label `Y`.

## Examples

### **Estimate Test Sample Classification Margins of SVM Classifiers**

Load the ionosphere data set.

```
load ionosphere
rng(1); % For reproducibility
```

Train an SVM classifier. Specify a 15% holdout sample for testing. It is good practice to specify the class order and standardize the data.

```
CVSVMModel = fitcsvm(X,Y,'Holdout',0.15,'ClassNames',{'b','g'},...  
    'Standardize',true);  
CompactSVMModel = CVSVMModel.Trained{1}; ...  
    % Extract the trained, compact classifier  
testInds = test(CVSVMModel.Partition); % Extract the test indices  
XTest = X(testInds,:);  
YTest = Y(testInds,:);
```

CVSVMModel is a ClassificationPartitionedModel classifier. It contains the property Trained, which is a 1-by-1 cell array holding a CompactClassificationSVM classifier that the software trained using the training set.

Estimate the test sample classification margins.

```
m = margin(CompactSVMModel,XTest,YTest);  
m(10:20)
```

```
ans =  
  
    3.5461  
    5.5939  
    4.9948  
    4.5611  
   -4.7963  
    5.5127  
   -2.8776  
    1.8673  
    9.4986  
    9.5018  
   20.9954
```

An observation margin is the observed true class score minus the maximum false class score among all scores in the respective class. Classifiers that yield relatively large margins are desirable.

### Select SVM Classifier Features by Examining Test Sample Margins

The classifier margins measure, for each observation, the difference between the true class observed score and the maximal false class score for a particular class. One way to perform feature selection is to compare test sample margins from multiple models. Based solely on this criterion, the model with the highest margins is the best model.

Load the ionosphere data set.

```
load ionosphere
rng(1); % For reproducibility
```

Partition the data set into training and test sets. Specify a 15% holdout sample for testing.

```
Partition = cvpartition(Y, 'Holdout', 0.15);
testInds = test(Partition); % Indices for the test set
XTest = X(testInds, :);
YTest = Y(testInds, :);
```

Partition defines the data set partition.

Define these two data sets:

- `fullX` contains all predictors (except the removed column of 0s).
- `partX` contains the last 20 predictors.

```
fullX = X;
partX = X(:, end-20:end);
```

Train SVM classifiers for each predictor set. Specify the partition definition.

```
FullCVSVMModel = fitcsvm(fullX, Y, 'CVPartition', Partition);
PartCVSVMModel = fitcsvm(partX, Y, 'CVPartition', Partition);
FCSVMModel = FullCVSVMModel.Trained{1};
PCSVMModel = PartCVSVMModel.Trained{1};
```

`FullCVSVMModel` and `PartCVSVMModel` are `ClassificationPartitionedModel` classifiers. They contain the property `Trained`, which is a 1-by-1 cell array holding a `CompactClassificationSVM` classifier that the software trained using the training set.

Estimate the test sample margins for each classifier.

```
fullM = margin(FCSVMModel, XTest, YTest);
partM = margin(PCSVMModel, XTest(:, end-20:end), YTest);
```



```
n = size(XTest,1);  
p = sum(fullM < partM)/n
```

```
p =  
  
    0.2500
```

Approximately 25% of the margins from the full model are less than those from the model with fewer predictors. This suggests that the model trained using all of the predictors is better.

## Definitions

### Classification Edge

The edge is the weighted mean of the classification margins.

The weights are the prior class probabilities. If you supply weights, then the software normalizes them to sum to the prior probabilities in the respective classes. The software uses the renormalized weights to compute the weighted mean.

One way to choose among multiple classifiers, e.g., to perform feature selection, is to choose the classifier that yields the highest edge.

### Classification Margin

The classification margins for binary classification are, for each observation, the difference between the classification score for the true class and the classification score for the false class.

The software defines the classification margin for binary classification as

$$m = 2yf(x).$$

$x$  is an observation. If the true label of  $x$  is the positive class, then  $y$  is 1, and  $-1$  otherwise.  $f(x)$  is the positive-class classification score for the observation  $x$ . The literature commonly defines the margin as  $m = yf(x)$ .

If the margins are on the same scale, then they serve as a classification confidence measure, i.e., among multiple classifiers, those that yield larger margins are better.

## Classification Score

The SVM classification score for classifying observation  $x$  is the signed distance from  $x$  to the decision boundary ranging from  $-\infty$  to  $+\infty$ . A positive score for a class indicates that  $x$  is predicted to be in that class. A negative score indicates otherwise.

The score for predicting  $x$  into the positive class, also the numerical, predicted response for  $x$ ,  $f(x)$ , is the trained SVM classification function

$$f(x) = \sum_{j=1}^n \alpha_j y_j G(x_j, x) + b,$$

where  $(\alpha_1, \dots, \alpha_n, b)$  are the estimated SVM parameters,  $G(x_j, x)$  is the dot product in the predictor space between  $x$  and the support vectors, and the sum includes the training set observations. The score for predicting  $x$  into the negative class is  $-f(x)$ .

If  $G(x_j, x) = x_j'x$  (the linear kernel), then the score function reduces to

$$f(x) = (x/s)' \beta + b.$$

$s$  is the kernel scale and  $\beta$  is the vector of fitted linear coefficients.

## Algorithms

For binary classification, the software defines the margin for observation  $j$ ,  $m_j$ , as

$$m_j = 2y_j f(x_j),$$

where  $y_j \in \{-1, 1\}$ , and  $f(x_j)$  is the predicted score of observation  $j$  for the positive class. However, the literature commonly uses  $m_j = y_j f(x_j)$  to define the margin.

## References

- [1] Christianini, N., and J. C. Shawe-Taylor. *An Introduction to Support Vector Machines and Other Kernel-Based Learning Methods*. Cambridge, UK: Cambridge University Press, 2000.

## Extended Capabilities

### Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays” (MATLAB).

### See Also

`ClassificationSVM` | `CompactClassificationSVM` | `edge` | `fitcsvm` | `loss` | `predict`

## margin

**Class:** CompactClassificationTree

Classification margins

### Syntax

```
m = margin(tree, TBL, ResponseVarName)
m = margin(tree, TBL, Y)
m = margin(tree, X, Y)
```

### Description

`m = margin(tree, TBL, ResponseVarName)` returns the classification margins for the table of predictors `TBL` and class labels `TBL.ResponseVarName`. For the definition, see “Margin” on page 32-3937.

`m = margin(tree, TBL, Y)` returns the classification margins for the table of predictors `TBL` and class labels `Y`.

`m = margin(tree, X, Y)` returns the classification margins for the matrix of predictors `X` and class labels `Y`.

### Input Arguments

**tree** — Trained classification tree

ClassificationTree model object | CompactClassificationTree model object

Trained classification tree, specified as a ClassificationTree or CompactClassificationTree model object. That is, `tree` is a trained classification model returned by `fitctree` or `compact`.

**TBL** — Sample data

table

Sample data, specified as a table. Each row of `TBL` corresponds to one observation, and each column corresponds to one predictor variable. Optionally, `TBL` can contain additional columns for the response variable and observation weights. `TBL` must contain all the predictors used to train `tree`. Multi-column variables and cell arrays other than cell arrays of character vectors are not allowed.

If `TBL` contains the response variable used to train `tree`, then you do not need to specify `ResponseVarName` or `Y`.

If you train `tree` using sample data contained in a table, then the input data for this method must also be in a table.

Data Types: `table`

### **x — Data to classify**

numeric matrix

Data to classify, specified as a numeric matrix. Each row of `X` represents one observation, and each column represents one predictor. `X` must have the same number of columns as the data used to train `tree`. `X` must have the same number of rows as the number of elements in `Y`.

Data Types: `single` | `double`

### **ResponseVarName — Response variable name**

name of a variable in `TBL`

Response variable name, specified as the name of a variable in `TBL`. If `TBL` contains the response variable used to train `tree`, then you do not need to specify `ResponseVarName`.

If you specify `ResponseVarName`, then you must do so as a character vector. For example, if the response variable is stored as `TBL.Response`, then specify it as `'Response'`. Otherwise, the software treats all columns of `TBL`, including `TBL.ResponseVarName`, as predictors.

The response variable must be a categorical or character array, logical or numeric vector, or cell array of character vectors. If the response variable is a character array, then each element must correspond to one row of the array.

### **y — Class labels**

categorical array | character array | logical vector | vector of numeric values | cell array of character vectors

Class labels, specified as a categorical or character array, a logical or numeric vector, or a cell array of character vectors.  $Y$  must be of the same type as the classification used to train `tree`, and its number of elements must equal the number of rows of  $X$ .

Data Types: `single` | `double` | `categorical` | `char` | `logical` | `cell`

## Output Arguments

### **m** — Margin

numeric column vector

Margin, returned as a numeric column vector of length `size(X,1)`. Each entry in `m` represents the margin for the corresponding rows of  $X$  and (true class)  $Y$ , computed using `tree`.

## Examples

Compute the classification margin for the Fisher iris data, trained on its first two columns of data, and view the last 10 entries.

```
load fisheriris
X = meas(:,1:2);
tree = fitctree(X,species);
M = margin(tree,X,species);
M(end-10:end)
```

```
ans =
    0.1111
    0.1111
    0.1111
   -0.2857
    0.6364
    0.6364
    0.1111
    0.7500
    1.0000
    0.6364
    0.2000
```

The classification tree trained on all the data is better.

```
tree = fitctree(meas,species);
M = margin(tree,meas,species);
M(end-10:end)

ans =
    0.9565
    0.9565
    0.9565
    0.9565
    0.9565
    0.9565
    0.9565
    0.9565
    0.9565
    0.9565
```

## Definitions

### Margin

The classification margin is the difference between the classification score for the true class and maximal classification score for the false classes. Margin is a column vector with the same number of rows as in the matrix  $X$ .

### Score (tree)

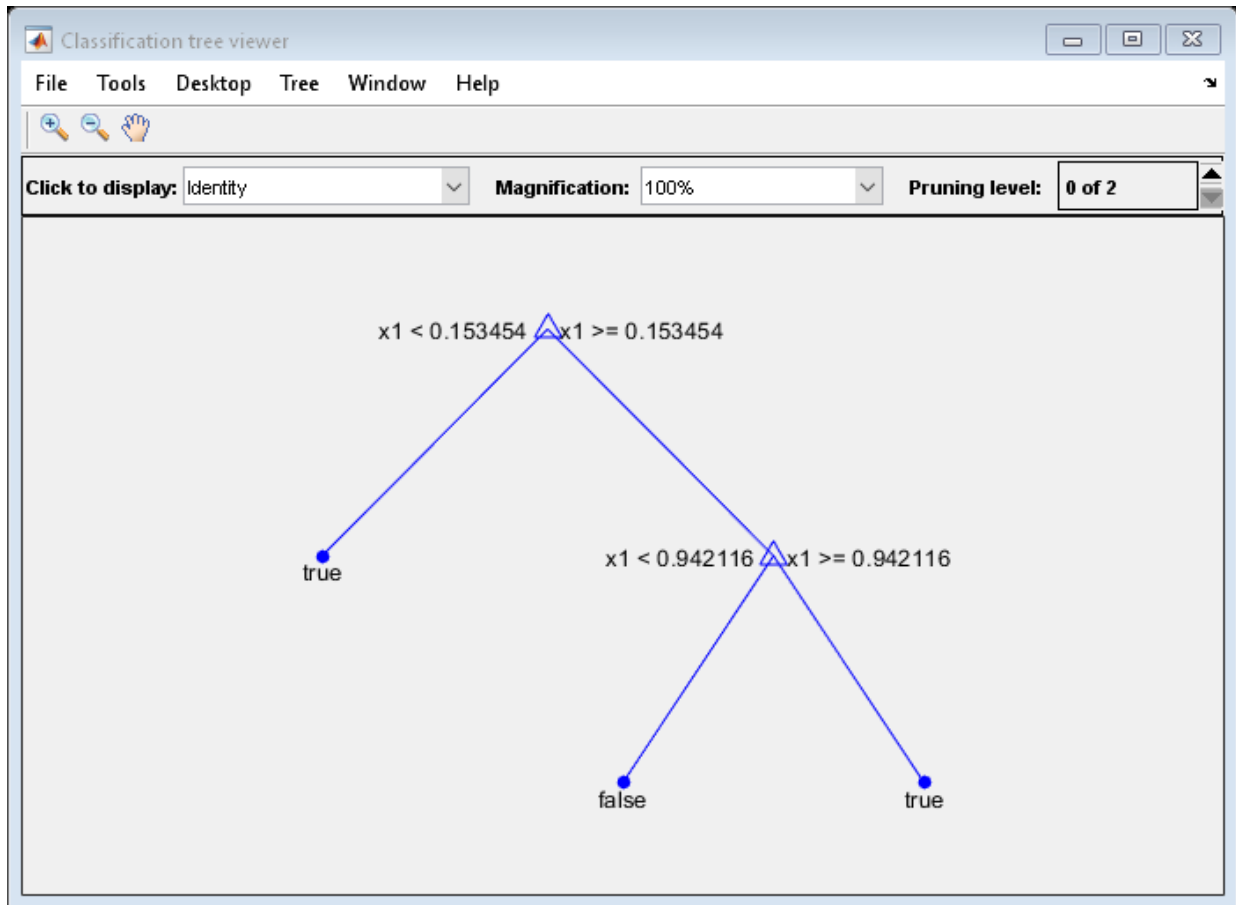
For trees, the score of a classification of a leaf node is the posterior probability of the classification at that node. The posterior probability of the classification at a node is the number of training sequences that lead to that node with the classification, divided by the number of training sequences that lead to that node.

For example, consider classifying a predictor  $X$  as `true` when  $X < 0.15$  or  $X > 0.95$ , and  $X$  is `false` otherwise.

Generate 100 random points and classify them:

```
rng(0, 'twister') % for reproducibility
X = rand(100,1);
Y = (abs(X - .55) > .4);
```

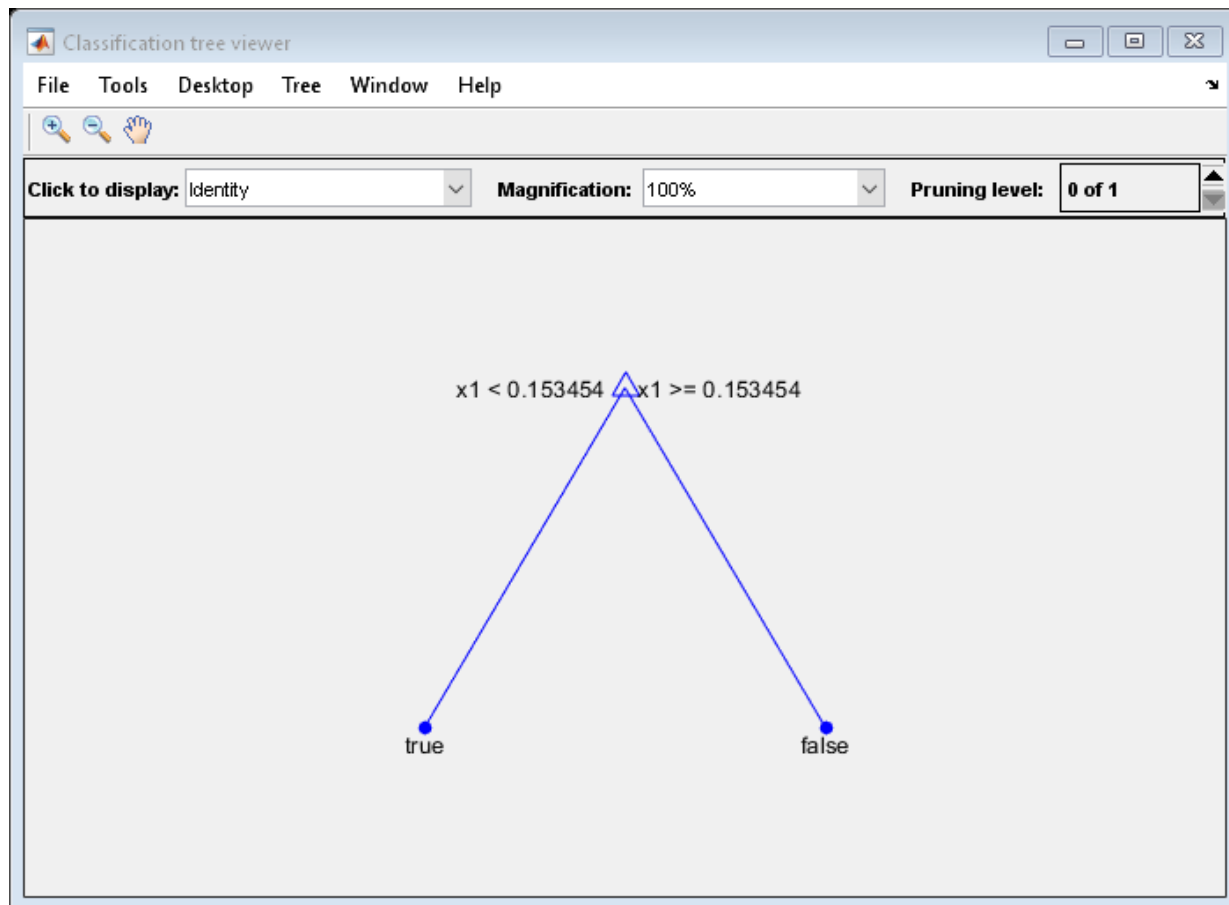
```
tree = fitctree(X,Y);  
view(tree,'Mode','Graph')
```



Prune the tree:

```
tree1 = prune(tree,'Level',1);  
view(tree1,'Mode','Graph')
```





The pruned tree correctly classifies observations that are less than 0.15 as `true`. It also correctly classifies observations from .15 to .94 as `false`. However, it incorrectly classifies observations that are greater than .94 as `false`. Therefore, the score for observations that are greater than .15 should be about  $.05/.85=.06$  for `true`, and about  $.8/.85=.94$  for `false`.

Compute the prediction scores for the first 10 rows of `X`:

```
[~,score] = predict(tree1,X(1:10));
[score X(1:10,:)]
```

```
ans =  
  
    0.9059    0.0941    0.8147  
    0.9059    0.0941    0.9058  
         0    1.0000    0.1270  
    0.9059    0.0941    0.9134  
    0.9059    0.0941    0.6324  
         0    1.0000    0.0975  
    0.9059    0.0941    0.2785  
    0.9059    0.0941    0.5469  
    0.9059    0.0941    0.9575  
    0.9059    0.0941    0.9649
```

Indeed, every value of  $x$  (the right-most column) that is less than 0.15 has associated scores (the left and center columns) of 0 and 1, while the other values of  $x$  have associated scores of 0.91 and 0.09. The difference (score 0.09 instead of the expected .06) is due to a statistical fluctuation: there are 8 observations in  $x$  in the range  $(.95, 1)$  instead of the expected 5 observations.

## Extended Capabilities

### Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays” (MATLAB).

### See Also

`edge` | `fitctree` | `loss` | `predict`

# margin

**Class:** CompactTreeBagger

Classification margin

## Syntax

```
mar = margin(B, TBLnew, Ynew)
mar = margin(B, Xnew, Ynew)
mar = margin(B, TBLnew, Ynew, 'param1', val1, 'param2', val2, ...)
mar = margin(B, Xnew, Ynew, 'param1', val1, 'param2', val2, ...)
```

## Description

`mar = margin(B, TBLnew, Ynew)` computes the classification margins for the predictors contained in the table `TBLnew` given true response `Ynew`. You can omit `Ynew` if `TBLnew` contains the response variable. If you trained `B` using sample data contained in a table, then the input data for this method must also be in a table.

`mar = margin(B, Xnew, Ynew)` computes the classification margins for the predictors contained in the matrix `Xnew` given true response `Ynew`.

`Ynew` can be a numeric vector, character matrix, cell array of character vectors, categorical vector or logical vector. `mar` is a numeric array of size `Nobs-by-NTrees`, where `Nobs` is the number of rows of `TBLnew` and `Ynew`, and `NTrees` is the number of trees in the ensemble `B`. For observation `I` and tree `J`, `mar(I, J)` is the difference between the score for the true class and the largest score for other classes. This method is available for classification ensembles only.

`mar = margin(B, TBLnew, Ynew, 'param1', val1, 'param2', val2, ...)` or `mar = margin(B, Xnew, Ynew, 'param1', val1, 'param2', val2, ...)` specifies optional parameter name-value pairs:

'Mode '	How the method computes errors. If set to 'cumulative' (default), margin computes cumulative errors and mar is an Nobs-by-NTrees matrix, where the first column gives error from trees (1), second column gives error from trees (1:2) etc., up to trees (1:NTrees). If set to 'individual', mar is a Nobs-by-NTrees matrix, where each element is an error from each tree in the ensemble. If set to 'ensemble', mar a single column of length Nobs showing the cumulative margins for the entire ensemble.
'Trees '	Vector of indices indicating what trees to include in this calculation. By default, this argument is set to 'all' and the method uses all trees. If 'Trees' is a numeric vector, the method returns a vector of length NTrees for 'cumulative' and 'individual' modes, where NTrees is the number of elements in the input vector, and a scalar for 'ensemble' mode. For example, in the 'cumulative' mode, the first element gives error from trees (1), the second element gives error from trees (1:2) etc.
'TreeWeights '	Vector of tree weights. This vector must have the same length as the 'Trees' vector. The method uses these weights to combine output from the specified trees by taking a weighted average instead of the simple non-weighted majority vote. You cannot use this argument in the 'individual' mode.
'UseInstanceForTree '	Logical matrix of size Nobs-by-NTrees indicating which trees should be used to make predictions for each observation. By default the method uses all trees for all observations.

## See Also

TreeBagger.margin

# margin

**Class:** TreeBagger

Classification margin

## Syntax

```
mar = margin(B, TBLnew, Ynew)
mar = margin(B, Xnew, Ynew)
mar = margin(B, TBLnew, Ynew, 'param1', val1, 'param2', val2, ...)
mar = margin(B, Xnew, Ynew, 'param1', val1, 'param2', val2, ...)
```

## Description

`mar = margin(B, TBLnew, Ynew)` computes the classification margins for the predictors contained in the table `TBLnew` given true response `Ynew`. You can omit `Ynew` if `TBLnew` contains the response variable. If you trained `B` using sample data contained in a table, then the input data for this method must also be in a table.

`mar = margin(B, Xnew, Ynew)` computes the classification margins for the predictors contained in the matrix `Xnew` given true response `Ynew`. If you trained `B` using sample data contained in a matrix, then the input data for this method must also be in a matrix.

`Ynew` can be a numeric vector, character matrix, cell array of character vectors, categorical vector or logical vector. `mar` is a numeric array of size `Nobs-by-NTrees`, where `Nobs` is the number of rows of `TBLnew` and `Ynew`, and `NTrees` is the number of trees in the ensemble `B`. For observation `I` and tree `J`, `mar(I, J)` is the difference between the score for the true class and the largest score for other classes. This method is available for classification ensembles only.

`mar = margin(B, TBLnew, Ynew, 'param1', val1, 'param2', val2, ...)` or `mar = margin(B, Xnew, Ynew, 'param1', val1, 'param2', val2, ...)` specifies optional parameter name-value pairs:

'Mode '	Character vector indicating how the method computes errors. If set to 'cumulative' (default), <code>margin</code> computes cumulative errors and <code>mar</code> is an <code>Nobs-by-NTrees</code> matrix, where the first column gives error from <code>trees(1)</code> , second column gives error from <code>trees(1:2)</code> etc., up to <code>trees(1:NTrees)</code> . If set to 'individual', <code>mar</code> is a <code>Nobs-by-NTrees</code> matrix, where each element is an error from each tree in the ensemble. If set to 'ensemble', <code>mar</code> a single column of length <code>Nobs</code> showing the cumulative margins for the entire ensemble.
'Trees '	Vector of indices indicating what trees to include in this calculation. By default, this argument is set to 'all' and the method uses all trees. If 'Trees' is a numeric vector, the method returns a vector of length <code>NTrees</code> for 'cumulative' and 'individual' modes, where <code>NTrees</code> is the number of elements in the input vector, and a scalar for 'ensemble' mode. For example, in the 'cumulative' mode, the first element gives error from <code>trees(1)</code> , the second element gives error from <code>trees(1:2)</code> etc.
'TreeWeights '	Vector of tree weights. This vector must have the same length as the 'Trees' vector. The method uses these weights to combine output from the specified trees by taking a weighted average instead of the simple non-weighted majority vote. You cannot use this argument in the 'individual' mode.
'UseInstanceForTree '	Logical matrix of size <code>Nobs-by-NTrees</code> indicating which trees should be used to make predictions for each observation. By default the method uses all trees for all observations.

## See Also

`CompactTreeBagger.margin`

# margmean

**Class:** RepeatedMeasuresModel

Estimate marginal means

## Syntax

```
tbl = margmean(rm, vars)
tbl = margmean(rm, vars, 'alpha', alpha)
```

## Description

`tbl = margmean(rm, vars)` returns the estimated marginal means for the variables `vars`, in the table `tbl`.

`tbl = margmean(rm, vars, 'alpha', alpha)` returns the  $100 \times (1 - \alpha)\%$  confidence intervals for the marginal means.

## Input Arguments

**rm** — Repeated measures model

RepeatedMeasuresModel object

Repeated measures model, returned as a RepeatedMeasuresModel object.

For properties and methods of this object, see RepeatedMeasuresModel.

**vars** — Variables for which to compute the marginal means

character vector | cell array of character vectors

Variables for which to compute the marginal means, specified as a character vector representing the name of a between or within-subjects factor in `rm`, or a cell array of character vectors representing the names of multiple variables. Each between-subjects factor must be categorical.

For example, if you want to compute the marginal means for the variables Drug and Gender, then you can specify as follows.

Example: `{ 'Drug', 'Gender' }`

Data Types: `char` | `cell`

### **alpha** — Confidence level

0.05 (default) | scalar value in the range of 0 to 1

Confidence level of the confidence intervals for population marginal means, specified as a scalar value in the range of 0 to 1. The confidence level is  $100*(1-\alpha)\%$ .

For example, you can specify a 99% confidence level as follows.

Example: `'alpha', 0.01`

Data Types: `double` | `single`

## Output Arguments

### **tbl** — Estimated marginal means

table

Estimated marginal means, returned as a table. `tbl` contains one row for each combination of the groups of the variables you specify in `vars`, one column for each variable, and the following columns.

Column name	Description
Mean	Estimated marginal means
StdErr	Standard errors of the estimates
Lower	Lower limit of a 95% confidence interval for the true population mean
Upper	Upper limit of a 95% confidence interval for the true population mean

## Examples

### Compute Marginal Means Grouped by Two Factors

Load the sample data.



```
load repeatedmeas
```

The table `between` includes the between-subject variables `age`, `IQ`, `group`, `gender`, and eight repeated measures `y1` to `y8` as responses. The table `within` includes the within-subject variables `w1` and `w2`. This is simulated data.

Fit a repeated measures model, where the repeated measures `y1` to `y8` are the responses, and `age`, `IQ`, `group`, `gender`, and the `group-gender` interaction are the predictor variables. Also specify the within-subject design matrix.

```
rm = fitrm(between, 'y1-y8 ~ Group*Gender + Age + IQ', 'WithinDesign', within);
```

Compute the marginal means grouped by the factors `Group` and `Gender`.

```
M = margmean(rm, {'Group' 'Gender'})
```

```
M =
```

```
6x6 table
```

Group	Gender	Mean	StdErr	Lower	Upper
A	Female	15.946	5.6153	4.3009	27.592
A	Male	8.0726	5.7236	-3.7973	19.943
B	Female	11.758	5.7091	-0.08189	23.598
B	Male	2.2858	5.6748	-9.483	14.055
C	Female	-8.6183	5.871	-20.794	3.5574
C	Male	-13.551	5.7283	-25.431	-1.6712

Display the description for table `M`.

```
M.Properties.Description
```

```
ans =
```

```
'Estimated marginal means
Means computed with Age=13.7, IQ=98.2667'
```

### Compute Estimated Marginal Means and Confidence Intervals

Load the sample data.

```
load fisheriris
```

The column vector, `species`, consists of iris flowers of three different species, `setosa`, `versicolor`, `virginica`. The double matrix `meas` consists of four types of measurements on the flowers, the length and width of sepals and petals in centimeters, respectively.

Store the data in a table array.

```
t = table(species,meas(:,1),meas(:,2),meas(:,3),meas(:,4),...  
'VariableNames',{'species','meas1','meas2','meas3','meas4'});  
Meas = dataset([1 2 3 4]','VarNames',{'Measurements'});
```

Fit a repeated measures model, where the measurements are the responses and the species is the predictor variable.

```
rm = fitrm(t,'meas1-meas4~species','WithinDesign',Meas);
```

Compute the marginal means grouped by the factor species.

```
margmean(rm,'species')
```

```
ans =
```

```
3x5 table
```

species	Mean	StdErr	Lower	Upper
'setosa'	2.5355	0.042807	2.4509	2.6201
'versicolor'	3.573	0.042807	3.4884	3.6576
'virginica'	4.285	0.042807	4.2004	4.3696

`StdError` field shows the standard errors of the estimated marginal means. The `Lower` and `Upper` fields show the lower and upper bounds for the 95% confidence intervals of the group marginal means, respectively. None of the confidence intervals overlap, which indicates that marginal means differ with species. You can also plot the estimated marginal means using the `plotprofile` method.

Compute the 99% confidence intervals for the marginal means.

```
margmean(rm, 'species', 'alpha', 0.01)
```

```
ans =
```

```
3x5 table
```

species	Mean	StdErr	Lower	Upper
'setosa'	2.5355	0.042807	2.4238	2.6472
'versicolor'	3.573	0.042807	3.4613	3.6847
'virginica'	4.285	0.042807	4.1733	4.3967

## See Also

[multcompare](#) | [fitrm](#) | [plotprofile](#)

## mauchly

**Class:** RepeatedMeasuresModel

Mauchly's test for sphericity

### Syntax

```
tbl = mauchly(rm)
tbl = mauchly(rm, C)
```

### Description

`tbl = mauchly(rm)` returns the result of the Mauchly's test for sphericity for the repeated measures model `rm`.

It tests the null hypothesis that the sphericity assumption is true for the response variables in `rm`.

For more information, see “Mauchly's Test of Sphericity” on page 9-83.

`tbl = mauchly(rm, C)` returns the result of the Mauchly's test based on the contrast matrix `C`.

### Input Arguments

**rm** — Repeated measures model

RepeatedMeasuresModel object

Repeated measures model, returned as a RepeatedMeasuresModel object.

For properties and methods of this object, see RepeatedMeasuresModel.

**c** — Contrasts

matrix

Contrasts, specified as a matrix. The default value of `C` is the  $Q$  factor in a QR decomposition of the matrix  $M$ , where  $M$  is defined so that  $Y*M$  is the difference between all successive pairs of columns of the repeated measures matrix  $Y$ .

Data Types: `single` | `double`

## Output Arguments

### `tbl` — Results of Mauchly's test of sphericity

table

Results of Mauchly's test for sphericity for the repeated measures model `rm`, returned as a table.

`tbl` contains the following columns.

Column Name	Definition
<code>W</code>	Value of Mauchly's $W$ statistic
<code>ChiStat</code>	Chi-square statistic value
<code>DF</code>	Degrees of freedom of the Chi-square statistic
<code>pValue</code>	$p$ -value corresponding to the Chi-square statistic

Data Types: `table`

## Examples

### Perform Mauchly's Test

Load the sample data.

```
load fisheriris
```

The column vector `species` consists of iris flowers of three different species: `setosa`, `versicolor`, and `virginica`. The double matrix `meas` consists of four types of measurements on the flowers: the length and width of sepals and petals in centimeters, respectively.

Store the data in a table array.

```
t = table(species,meas(:,1),meas(:,2),meas(:,3),meas(:,4),...  
'VariableNames',{'species','meas1','meas2','meas3','meas4'});  
Meas = dataset([1 2 3 4]','VarNames',{'Measurements'});
```

Fit a repeated measures model, where the measurements are the responses and the species is the predictor variable.

```
rm = fitrm(t,'meas1-meas4~species','WithinDesign',Meas);
```

Perform Mauchly’s test to assess the sphericity assumption.

```
mauchly(rm)
```

```
ans =
```

```
1x4 table
```

W	ChiStat	DF	pValue
0.55814	84.976	5	7.6149e-17

The small *P*-value (in the `pValue` field) indicates that the sphericity, hence the compound symmetry assumption, does not hold. You should use epsilon corrections to compute the *P*-values for a repeated measures anova. You can compute the epsilon corrections using the `epsilon` method and perform the repeated measures anova with the corrected *P*-values using the `ranova` method.

## See Also

`epsilon` | `fitrm` | `ranova`

## Topics

“Mauchly’s Test of Sphericity” on page 9-83

“Compound Symmetry Assumption and Epsilon Corrections” on page 9-81

# mat2dataset

Convert matrix to dataset array

---

**Note** The `dataset` data type might be removed in a future release. To work with heterogeneous data, use the MATLAB® `table` data type instead. See MATLAB `table` documentation for more information.

---

## Syntax

```
ds = mat2dataset(X)
ds = mat2dataset(X,Name,Value)
```

## Description

`ds = mat2dataset(X)` converts a matrix to a dataset array.

`ds = mat2dataset(X,Name,Value)` performs the conversion using additional options specified by one or more `Name,Value` pair arguments.

## Examples

### Convert Matrix to Dataset Array

Convert a matrix to a dataset array using the default options.

Load sample data.

```
load('fisheriris')
X = meas;
size(X)
```

```
ans =
```

```
150      4
```

Convert the matrix to a dataset array.

```
ds = mat2dataset(X);  
size(ds)
```

```
ans =
```

```
150      4
```

```
ds(1:5, :)
```

```
ans =
```

X1	X2	X3	X4
5.1	3.5	1.4	0.2
4.9	3	1.4	0.2
4.7	3.2	1.3	0.2
4.6	3.1	1.5	0.2
5	3.6	1.4	0.2

When you do not specify variable names, `mat2dataset` uses the matrix name and column numbers to create default variable names.

### Convert Matrix to Dataset Array with Variable Names

Load sample data.

```
load('fisheriris')  
X = meas;  
size(X)
```

```
ans =
```

```
150      4
```



Convert the matrix to a dataset array, providing a variable name for each of the four column of X.

```
ds = mat2dataset(X, 'VarNames', {'SLength', ...  
'SWidth', 'PLength', 'PWidth'});  
size(ds)
```

```
ans =  
  
    150     4
```

```
ds(1:5, :)
```

```
ans =  
  
    SLength    SWidth    PLength    PWidth  
    5.1        3.5        1.4        0.2  
    4.9         3         1.4        0.2  
    4.7        3.2        1.3        0.2  
    4.6        3.1        1.5        0.2  
     5         3.6        1.4        0.2
```

### Create a Dataset Array with Multicolumn Variables

Convert a matrix to a dataset array containing multicolumn variables.

Load sample data.

```
load('fisheriris')  
X = meas;  
size(X)
```

```
ans =  
  
    150     4
```

Convert the matrix to a dataset array, combining the sepal measurements (the first two columns) into one variable named `SepalMeas`, and the petal measurements (third and fourth columns) into one variable names `PetalMeas`.

```
ds = mat2dataset(X, 'NumCols', [2,2], ...  
'VarNames', {'SepalMeas', 'PetalMeas'});  
ds(1:5, :)
```

```
ans =
```

SepalMeas		PetalMeas	
5.1	3.5	1.4	0.2
4.9	3	1.4	0.2
4.7	3.2	1.3	0.2
4.6	3.1	1.5	0.2
5	3.6	1.4	0.2

The output dataset array has 150 observations and 2 variables.

```
size(ds)
```

```
ans =
```

```
150    2
```

- “Create a Dataset Array from Workspace Variables” on page 2-70
- “Create a Dataset Array from a File” on page 2-77

## Input Arguments

### **x** — Input matrix

matrix

Input matrix to convert to a dataset array, specified as an  $M$ -by- $N$  numeric matrix. Each column of  $x$  becomes a variable in the output  $M$ -by- $N$  dataset array.

Data Types: `single` | `double`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'NumCols', [1, 1, 2, 1]` specifies that the 3rd and 4th columns of the input matrix should be combined into a single variable.

### **VarNames** — Variable names for output dataset array

cell array of character vectors

Variable names for the output dataset array, specified as the comma-separated pair consisting of `'VarNames'` and a cell array of character vectors. You must provide a variable name for each variable in `ds`. The names must be valid MATLAB identifiers, and must be unique.

Example: `'VarNames', {'myVar1', 'myVar2', 'myVar3'}`

### **ObsNames** — Observation names for output dataset array

cell array of character vectors

Observation names for the output dataset array, specified as the comma-separated pair consisting of `'ObsNames'` and a cell array of character vectors. The names do not need to be valid MATLAB identifiers, but they must be unique.

### **NumCols** — Number of columns for each variable

vector of nonnegative integers

Number of columns for each variable in `ds`, specified as the comma-separated pair consisting of `'NumCols'` and a vector of nonnegative integers. When the number of columns for a variable is greater than one, `mat2dataset` combines multiple columns in `X` into a single variable in `ds`. The vector you assign to `NumCols` must sum to `size(X, 2)`.

For example, to convert a matrix with eight columns into a dataset array with five variables, specify a vector with five elements that sum to eight, such as `'NumCols', [1, 1, 3, 1, 2]`.

## Output Arguments

### **ds** — Output dataset array

dataset array

Output dataset array, returned by default with a variable for each column of  $X$ , and an observation for each row of  $X$ . If you specify `NumCols`, then the number of variables in `ds` is equal to the length of the specified vector of column numbers.

## See Also

`cell2dataset` | `dataset` | `struct2dataset`

## Topics

“Create a Dataset Array from Workspace Variables” on page 2-70

“Create a Dataset Array from a File” on page 2-77

“Dataset Arrays” on page 2-142

**Introduced in R2012b**

# mdscale

Nonclassical multidimensional scaling

## Syntax

```
Y = mdscale(D,p)
[Y, stress] = mdscale(D,p)
[Y, stress, disparities] = mdscale(D,p)
[...] = mdscale(D,p, 'Name', value)
```

## Description

`Y = mdscale(D,p)` performs nonmetric multidimensional scaling on the  $n$ -by- $n$  dissimilarity matrix  $D$ , and returns  $Y$ , a configuration of  $n$  points (rows) in  $p$  dimensions (columns). The Euclidean distances between points in  $Y$  approximate a monotonic transformation of the corresponding dissimilarities in  $D$ . By default, `mdscale` uses Kruskal's normalized stress1 criterion.

You can specify  $D$  as either a full  $n$ -by- $n$  matrix, or in upper triangle form such as is output by `pdist`. A full dissimilarity matrix must be real and symmetric, and have zeros along the diagonal and non-negative elements everywhere else. A dissimilarity matrix in upper triangle form must have real, non-negative entries. `mdscale` treats NaNs in  $D$  as missing values, and ignores those elements. `Inf` is not accepted.

You can also specify  $D$  as a full similarity matrix, with ones along the diagonal and all other elements less than one. `mdscale` transforms a similarity matrix to a dissimilarity matrix in such a way that distances between the points returned in  $Y$  approximate  $\sqrt{1-D}$ . To use a different transformation, transform the similarities prior to calling `mdscale`.

`[Y, stress] = mdscale(D,p)` returns the minimized stress, i.e., the stress evaluated at  $Y$ .

`[Y, stress, disparities] = mdscale(D,p)` returns the disparities, that is, the monotonic transformation of the dissimilarities  $D$ .

[...] = `mdscale(D, p, 'Name', value)` specifies one or more optional parameter name/value pairs that control further details of `mdscale`. Specify *Name* in single quotes.

Available parameters are

- **Criterion**— The goodness-of-fit criterion to minimize. This also determines the type of scaling, either non-metric or metric, that `mdscale` performs. Choices for non-metric scaling are:
  - `'stress'` — Stress normalized by the sum of squares of the inter-point distances, also known as `stress1`. This is the default.
  - `'sstress'` — Squared stress, normalized with the sum of 4th powers of the inter-point distances.

Choices for metric scaling are:

- `'metricstress'` — Stress, normalized with the sum of squares of the dissimilarities.
- `'metricsstress'` — Squared stress, normalized with the sum of 4th powers of the dissimilarities.
- `'sammon'` — Sammon's nonlinear mapping criterion. Off-diagonal dissimilarities must be strictly positive with this criterion.
- `'strain'` — A criterion equivalent to that used in classical multidimensional scaling.
- **Weights** — A matrix or vector the same size as `D`, containing nonnegative dissimilarity weights. You can use these to weight the contribution of the corresponding elements of `D` in computing and minimizing stress. Elements of `D` corresponding to zero weights are effectively ignored.

---

**Note** When you specify weights as a full matrix, its diagonal elements are ignored and have no effect, since the corresponding diagonal elements of `D` do not enter into the stress calculation.

---

- **Start** — Method used to choose the initial configuration of points for `Y`. The choices are
  - `'cmdscale'` — Use the classical multidimensional scaling solution. This is the default. `'cmdscale'` is not valid when there are zero weights.
  - `'random'` — Choose locations randomly from an appropriately scaled `p`-dimensional normal distribution with uncorrelated coordinates.

- An  $n$ -by- $p$  matrix of initial locations, where  $n$  is the size of the matrix  $D$  and  $p$  is the number of columns of the output matrix  $Y$ . In this case, you can pass in `[]` for  $p$  and `mdscale` infers  $p$  from the second dimension of the matrix. You can also supply a 3-D array, implying a value for 'Replicates' from the array's third dimension.
- `Replicates` — Number of times to repeat the scaling, each with a new initial configuration. The default is 1.
- `Options` — Options for the iterative algorithm used to minimize the fitting criterion. Pass in an options structure created by `statset`. For example,

```
opts = statset(param1, val1, param2, val2, ...);
[...] = mdscale(..., 'Options', opts)
```

The choices of `statset` parameters are

- 'Display' — Level of display output. The choices are 'off' (the default), 'iter', and 'final'.
- 'MaxIter' — Maximum number of iterations allowed. The default is 200.
- 'TolFun' — Termination tolerance for the stress criterion and its gradient. The default is  $1e-4$ .
- 'TolX' — Termination tolerance for the configuration location step size. The default is  $1e-4$ .

## Examples

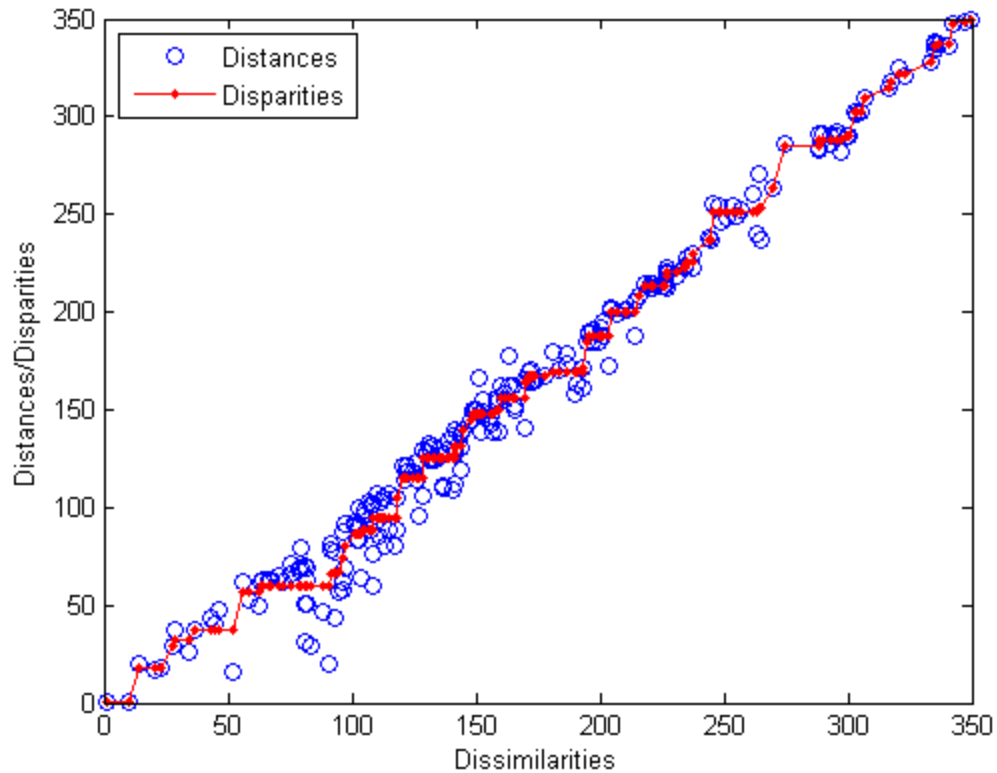
```
load cereal.mat
X = [Calories Protein Fat Sodium Fiber ...
     Carbo Sugars Shelf Potass Vitamins];

% Take a subset from a single manufacturer.
X = X(strcmp('K', cellstr(Mfg)), :);

% Create a dissimilarity matrix.
dissimilarities = pdist(X);

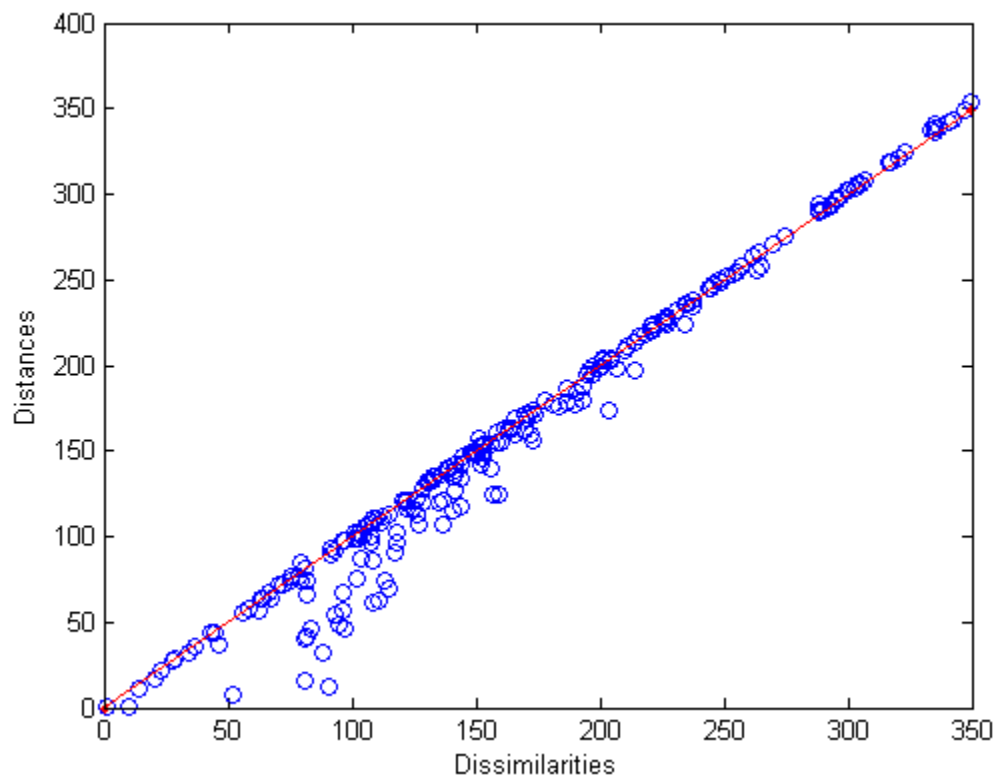
% Use non-metric scaling to recreate the data in 2D,
% and make a Shepard plot of the results.
[Y, stress, disparities] = mdscale(dissimilarities, 2);
distances = pdist(Y);
```

```
[dum,ord] = sortrows([disparities(:) dissimilarities(:)]);
plot(dissimilarities,distances,'bo', ...
dissimilarities(ord),disparities(ord),'r.-');
xlabel('Dissimilarities'); ylabel('Distances/Disparities')
legend({'Distances' 'Disparities'},'Location','NW');
```



```
% Do metric scaling on the same dissimilarities.
figure
[Y, stress] = ...
mdscale(dissimilarities,2,'criterion','metricsstress');
distances = pdist(Y);
plot(dissimilarities,distances,'bo', ...
[0 max(dissimilarities)], [0 max(dissimilarities)], 'r.-');
xlabel('Dissimilarities'); ylabel('Distances')
```





## See Also

`cmdscale` | `pdist` | `statset`

Introduced before R2006a

## mdsprox

**Class:** CompactTreeBagger

Multidimensional scaling of proximity matrix

### Syntax

```
[SC,EIGEN] = mdsprox(B,X)
[SC,EIGEN] = mdsprox(B,X,'param1',val1,'param2',val2,...)
```

### Description

[SC,EIGEN] = mdsprox(B,X) applies classical multidimensional scaling to the proximity matrix computed for the data in the matrix X, and returns scaled coordinates SC and eigenvalues EIGEN of the scaling transformation. The method applies multidimensional scaling to the matrix of distances defined as 1-prox, where prox is the proximity matrix returned by the proximity method.

You can supply the proximity matrix directly by using the 'Data' parameter.

[SC,EIGEN] = mdsprox(B,X,'param1',val1,'param2',val2,...) specifies optional parameter name/value pairs:

'Data'	Flag indicating how the method treats the X input argument. If set to 'predictors' (default), mdsprox assumes X to be a matrix of predictors and used for computation of the proximity matrix. If set to 'proximity', the method treats X as a proximity matrix returned by the proximity method.
--------	---

'Colors'	If you supply this argument, <code>mdsprox</code> makes overlaid scatter plots of two scaled coordinates using specified colors for different classes. You must supply the colors as a character vector with one character for each color. If there are more classes in the data than characters in the supplied character vector, <code>mdsprox</code> only plots the first <code>C</code> classes, where <code>C</code> is the length of the character vector. For regression or if you do not provide the vector of true class labels, the method uses the first color for all observations in <code>X</code> .
'Labels'	Vector of true class labels for a classification ensemble. True class labels can be either a numeric vector, character matrix, or cell array of character vectors. If supplied, this vector must have as many elements as there are observations (rows) in <code>X</code> . This argument has no effect unless you also supply the 'Colors' argument.
'MDSCoordinates'	Indices of the two scaled coordinates to plot. By default, <code>mdsprox</code> makes a scatter plot of the first and second scaled coordinates which correspond to the two largest eigenvalues. You can specify any other two or three indices not exceeding the dimensionality of the scaled data. This argument has no effect unless you also supply the 'Colors' argument.

## See Also

`TreeBagger.mdsprox` | `cmdscale` | `proximity`

## mdsprox

**Class:** TreeBagger

Multidimensional scaling of proximity matrix

### Syntax

```
[S,E] = mdsprox(B)
[S,E] = mdsprox(B, 'param1', val1, 'param2', val2, ...)
```

### Description

`[S,E] = mdsprox(B)` returns scaled coordinates, `S`, and eigenvalues, `E`, for the proximity matrix in the ensemble `B`. An earlier call to `fillprox(B)` must create the proximity matrix.

`[S,E] = mdsprox(B, 'param1', val1, 'param2', val2, ...)` specifies optional parameter name/value pairs:

- |          |  |
|----------|--|
| 'Keep'   | Array of indices of observations in the training data to use for multidimensional scaling. By default, this argument is set to 'all'. If you provide numeric or logical indices, the method uses only the subset of the training data specified by these indices to compute the scaled coordinates and eigenvalues.  |
| 'Colors' | If you supply this argument, <code>mdsprox</code> makes overlaid scatter plots of two scaled coordinates using specified colors for different classes. You must supply the colors as a character vector with one character for each color. If there are more classes in the data than characters in the supplied character vector, <code>mdsprox</code> only plots the first <code>C</code> classes, where <code>C</code> is the length of the character vector. For regression or if you do not provide the vector of true class labels, the method uses the first color for all observations in <code>X</code> . |

'MDSCoordinate Indices of the two scaled coordinates to plot. By default, `mdsprox`  
s' makes a scatter plot of the first and second scaled coordinates which correspond to the two largest eigenvalues. You can specify any other two or three indices not exceeding the dimensionality of the scaled data. This argument has no effect unless you also supply the 'Colors' argument.

## See Also

`CompactTreeBagger.mdsprox` | `cmdscale` | `fillprox`

## mean

Mean of probability distribution

## Syntax

```
m = mean(pd)
```

## Description

`m = mean(pd)` returns the mean `m` of the probability distribution `pd`.

## Examples

### Mean of a Fitted Distribution

Load the sample data. Create a vector containing the first column of students' exam grade data.

```
load examgrades
x = grades(:,1);
```

Create a normal distribution object by fitting it to the data.

```
pd = fitdist(x, 'Normal')
```

```
pd =
```

```
NormalDistribution
```

```
Normal distribution
```

```
mu = 75.0083 [73.4321, 76.5846]
sigma = 8.7202 [7.7391, 9.98843]
```

Compute the mean of the fitted distribution.

```
m = mean(pd)
```

```
m =
```

```
75.0083
```

The mean of the normal distribution is equal to the parameter  $\mu$ .

### Mean of a Skewed Distribution

Create a Weibull probability distribution object.

```
pd = makedist('Weibull', 'a', 5, 'b', 2)
```

```
pd =
```

```
WeibullDistribution
```

```
Weibull distribution
```

```
A = 5
```

```
B = 2
```

Compute the mean of the distribution.

```
mean = mean(pd)
```

```
mean =
```

```
4.4311
```

## Input Arguments

**pd** — Probability distribution  
probability distribution object

Probability distribution, specified as a probability distribution object. Create a probability distribution object with specified parameter values using `makedist`. Alternatively, create a probability distribution object by fitting it to data using `fitdist` or the Distribution Fitter app.

## Output Arguments

### **m** — Mean

scalar value

Mean of the probability distribution, returned as a scalar value.

## See Also

`distributionFitter` | `fitdist` | `makedist`

## Topics

“Working with Probability Distributions” on page 5-3

Introduced before R2006a



## mean

**Class:** ProbDistUnivParam

Return mean of ProbDistUnivParam object

## Syntax

$M = \text{mean}(PD)$

## Description

$M = \text{mean}(PD)$  returns  $M$ , the mean of the ProbDistUnivParam object  $PD$ .

## Input Arguments

$PD$  An object of the class ProbDistUnivParam.

## Output Arguments

$M$  The mean of the ProbDistUnivParam object  $PD$ .

## See Also

mean

## mean

**Class:** prob.ParametricTruncatableDistribution

**Package:** prob

Mean of probability distribution object

## Syntax

```
m = mean(pd)
```

## Description

`m = mean(pd)` returns the mean `m` of the probability distribution `pd`.

## Input Arguments

**pd — Probability distribution**

probability distribution object

Probability distribution, specified as a probability distribution object. Create a probability distribution object with specified parameter values using `makedist`.

## Output Arguments

**m — Mean**

scalar value

Mean of the probability distribution, returned as a scalar value.

## Examples

## Mean of a Uniform Distribution

Create a uniform distribution object

```
pd = makedist('Uniform', 'lower', -3, 'upper', 5)
```

```
pd =
```

```
UniformDistribution
```

```
Uniform distribution
```

```
Lower = -3
```

```
Upper = 5
```

Compute the mean of the distribution.

```
m = mean(pd)
```

```
m =
```

```
1
```

## See Also

`makedist`

## mean

**Class:** prob.ToolboxFittableParametricDistribution

**Package:** prob

Mean of probability distribution object

## Syntax

```
m = mean(pd)
```

## Description

`m = mean(pd)` returns the mean `m` of the probability distribution `pd`.

## Input Arguments

**pd** — Probability distribution

probability distribution object

Probability distribution, specified as a probability distribution object. Create a probability distribution object with specified parameter values using `makedist`. Alternatively, create a probability distribution object by fitting it to data using `fitdist` or the Distribution Fitter app.

## Output Arguments

**m** — Mean

scalar value

Mean of the probability distribution, returned as a scalar value.

## Examples

### Mean of a Fitted Distribution

Load the sample data. Create a vector containing the first column of students' exam grade data.

```
load examgrades;  
x = grades(:,1);
```

Create a normal distribution object by fitting it to the data.

```
pd = fitdist(x, 'Normal')  
  
pd =  
  
NormalDistribution  
  
Normal distribution  
    mu = 75.0083    [73.4321, 76.5846]  
    sigma = 8.7202    [7.7391, 9.98843]
```

Compute the mean of the fitted distribution.

```
m = mean(pd)
```

```
m =  
  
75.0083
```

The mean of the normal distribution is equal to the parameter mu.

### Mean of a Skewed Distribution

Create a Weibull probability distribution object.

```
pd = makedist('Weibull', 'a', 5, 'b', 2)
```

```
pd =  
  
WeibullDistribution  
  
Weibull distribution  
  A = 5  
  B = 2
```

Compute the mean of the distribution.

```
mean = mean(pd)
```

```
mean =  
  
  4.4311
```

## See Also

[distributionFitter](#) | [fitdist](#) | [makedist](#)

# meanMargin

**Class:** CompactTreeBagger

Mean classification margin

## Syntax

```
mar = meanMargin(B, TBLnew, Ynew)
mar = meanMargin(B, Xnew, Ynew)
mar = meanMargin(B, TBLnew, Ynew, 'param1', val1, 'param2', val2, ...)
mar = meanMargin(B, Xnew, Ynew, 'param1', val1, 'param2', val2, ...)
```

## Description

`mar = meanMargin(B, TBLnew, Ynew)` computes average classification margins for the predictors contained in the table `TBLnew` given the true response `Ynew`. You can omit `Ynew` if `TBLnew` contains the response variable. If you trained `B` using sample data contained in a table, then the input data for this method must also be in a table.

`mar = meanMargin(B, Xnew, Ynew)` computes average classification margins for the predictors contained in the matrix `Xnew` given true response `Ynew`. If you trained `B` using sample data contained in a matrix, then the input data for this method must also be in a matrix.

`Ynew` can be either a numeric vector, character matrix, cell array of character vectors, categorical vector or logical vector. `meanMargin` averages the margins over all observations (rows) in `TBLnew` or `Xnew` for each tree. `mar` is a matrix of size 1-by-`NTrees`, where `NTrees` is the number of trees in the ensemble `B`. This method is available for classification ensembles only.

`mar = meanMargin(B, TBLnew, Ynew, 'param1', val1, 'param2', val2, ...)` or `mar = meanMargin(B, Xnew, Ynew, 'param1', val1, 'param2', val2, ...)` specifies optional parameter name-value pairs:

'Mode '	How meanMargin computes errors. If set to 'cumulative' (default), is a vector of length NTrees where the first element gives mean margin from trees (1), second column gives mean margins from trees (1:2) etc., up to trees (1:NTrees). If set to 'individual', mar is a vector of length NTrees, where each element is a mean margin from each tree in the ensemble. If set to 'ensemble', mar is a scalar showing the cumulative mean margin for the entire ensemble.
'Trees '	Vector of indices indicating what trees to include in this calculation. By default, this argument is set to 'all' and the method uses all trees. If 'Trees' is a numeric vector, the method returns a vector of length NTrees for 'cumulative' and 'individual' modes, where NTrees is the number of elements in the input vector, and a scalar for 'ensemble' mode. For example, in the 'cumulative' mode, the first element gives mean margin from trees (1), the second element gives mean margin from trees (1:2) etc.
'TreeWeights '	Vector of tree weights. This vector must have the same length as the 'Trees' vector. meanMargin uses these weights to combine output from the specified trees by taking a weighted average instead of the simple nonweighted majority vote. You cannot use this argument in the 'individual' mode.

## See Also

TreeBagger.meanMargin



# meanMargin

**Class:** TreeBagger

Mean classification margin

## Syntax

```
mar = meanMargin(B, TBLnew, Ynew)
mar = meanMargin(B, Xnew, Ynew)
mar = meanMargin(B, TBLnew, Ynew, 'param1', val1, 'param2', val2, ...)
mar = meanMargin(B, Xnew, Ynew, 'param1', val1, 'param2', val2, ...)
```

## Description

`mar = meanMargin(B, TBLnew, Ynew)` computes average classification margins for the predictors contained in the table `TBLnew` given the true response `Ynew`. You can omit `Ynew` if `TBLnew` contains the response variable. If you trained `B` using sample data contained in a table, then the input data for this method must also be in a table.

`mar = meanMargin(B, Xnew, Ynew)` computes average classification margins for the predictors contained in the matrix `Xnew` given true response `Ynew`. If you trained `B` using sample data contained in a matrix, then the input data for this method must also be in a matrix.

`Ynew` can be either a numeric vector, character matrix, cell array of character vectors, categorical vector or logical vector. `meanMargin` averages the margins over all observations (rows) in `TBLnew` or `Xnew` for each tree. `mar` is a matrix of size 1-by-`NTrees`, where `NTrees` is the number of trees in the ensemble `B`. This method is available for classification ensembles only.

`mar = meanMargin(B, TBLnew, Ynew, 'param1', val1, 'param2', val2, ...)` or `mar = meanMargin(B, Xnew, Ynew, 'param1', val1, 'param2', val2, ...)` specifies optional parameter name-value pairs:

'Mode '	Character vector indicating how <code>meanMargin</code> computes errors. If set to 'cumulative' (default), is a vector of length <code>NTrees</code> where the first element gives mean margin from <code>trees(1)</code> , second column gives mean margins from <code>trees(1:2)</code> etc, up to <code>trees(1:NTrees)</code> . If set to 'individual', <code>mar</code> is a vector of length <code>NTrees</code> , where each element is a mean margin from each tree in the ensemble. If set to 'ensemble', <code>mar</code> is a scalar showing the cumulative mean margin for the entire ensemble.
'Trees '	Vector of indices indicating what trees to include in this calculation. By default, this argument is set to 'all' and the method uses all trees. If 'Trees' is a numeric vector, the method returns a vector of length <code>NTrees</code> for 'cumulative' and 'individual' modes, where <code>NTrees</code> is the number of elements in the input vector, and a scalar for 'ensemble' mode. For example, in the 'cumulative' mode, the first element gives mean margin from <code>trees(1)</code> , the second element gives mean margin from <code>trees(1:2)</code> etc.
'TreeWeights '	Vector of tree weights. This vector must have the same length as the 'Trees' vector. <code>meanMargin</code> uses these weights to combine output from the specified trees by taking a weighted average instead of the simple nonweighted majority vote. You cannot use this argument in the 'individual' mode.
'Weights '	Vector of observation weights to use for margin averaging. By default the weight of every observation is set to 1. The length of this vector must be equal to the number of rows in <code>X</code> .
'UseInstanceForTree '	Logical matrix of size <code>Nobs-by-NumTrees</code> indicating which trees to use to make predictions for each observation. By default the method uses all trees for all observations.

## See Also

`CompactTreeBagger.meanMargin`

# meansurrvarassoc

**Class:** `classregtree`

Mean predictive measure of association for surrogate splits in decision tree

---

**Note** `classregtree` will be removed in a future release. See `fitctree`, `fitrtree`, `ClassificationTree`, or `RegressionTree` instead.

---

## Syntax

`MA = meansurrvarassoc(T)`

`MA = meansurrvarassoc(T,N)`

## Description

`MA = meansurrvarassoc(T)` returns a  $p$ -by- $p$  matrix, `MA`, with predictive measures of association for  $p$  predictors. Element  $MA(i,j)$  is the predictive measure of association averaged over surrogate splits on predictor  $j$  for which predictor  $i$  is the optimal split predictor. This average is computed by summing positive values of the predictive measure of association over optimal splits on predictor  $i$  and surrogate splits on predictor  $j$  and dividing by the total number of optimal splits on predictor  $i$ , including splits for which the predictive measure of association between predictors  $i$  and  $j$  is negative.

`MA = meansurrvarassoc(T,N)` takes an array `N` of node numbers and returns the predictive measure of association averaged over the specified nodes.

## See Also

`classregtree` | `surrctcategories` | `surrctflip` | `surrctpoint` |  
`surrcttype` | `surrctvar` | `surrvarassoc`

## surrogateAssociation

**Class:** CompactClassificationTree

Mean predictive measure of association for surrogate splits in decision tree

### Syntax

```
ma = surrogateAssociation(tree)
ma = surrogateAssociation(tree,N)
```

### Description

`ma = surrogateAssociation(tree)` returns a matrix of predictive measures of association for the predictors in `tree`.

`ma = surrogateAssociation(tree,N)` returns a matrix of predictive measures of association averaged over the nodes in vector `N`.

### Input Arguments

**tree**

A classification tree constructed with `fitctree`, or a compact regression tree constructed with `compact`.

**N**

Vector of node numbers in `tree`.

## Output Arguments

### ma

- `ma = surrogateAssociation(tree)` returns a P-by-P matrix, where P is the number of predictors in `tree`. `ma(i, j)` is the predictive measure of association on page 32-3984 between the optimal split on variable `i` and a surrogate split on variable `j`. For more details, see “Algorithms” on page 32-3985.
- `ma = surrogateAssociation(tree, N)` returns a P-by-P representing the predictive measure of association between variables averaged over nodes in the vector `N`. `N` contains node numbers from 1 to `max(tree.NumNodes)`.

## Examples

### Estimate Predictive Measures of Association for Surrogate Splits

Load Fisher's iris data set.

```
load fisheriris
```

Grow a classification tree using `species` as the response. Specify to use surrogate splits for missing values.

```
tree = fitctree(meas, species, 'surrogate', 'on');
```

Find the mean predictive measure of association between the predictor variables.

```
ma = surrogateAssociation(tree)
```

```
ma =
```

```

1.0000         0         0         0
         0 1.0000         0         0
0.4633 0.2500 1.0000 0.5000
0.2065 0.1413 0.4022 1.0000
```

Find the mean predictive measure of association averaged over the odd-numbered nodes in `tree`.

```

N = 1:2:tree.NumNodes;
ma = surrogateAssociation(tree,N)

ma =

    1.0000         0         0         0
         0    1.0000         0         0
    0.7600    0.5000    1.0000    1.0000
    0.4130    0.2826    0.8043    1.0000

```

## Definitions

### Predictive Measure of Association

The predictive measure of association is a value that indicates the similarity between decision rules that split observations. Among all possible decision splits that are compared to the optimal split (found by growing the tree), the best surrogate decision split on page 32-2174 yields the maximum predictive measure of association. The second-best surrogate split has the second-largest predictive measure of association.

Suppose  $x_j$  and  $x_k$  are predictor variables  $j$  and  $k$ , respectively, and  $j \neq k$ . At node  $t$ , the predictive measure of association between the optimal split  $x_j < u$  and a surrogate split  $x_k < v$  is

$$\lambda_{jk} = \frac{\min(P_L, P_R) - (1 - P_{L_j L_k} - P_{R_j R_k})}{\min(P_L, P_R)}.$$

- $P_L$  is the proportion of observations in node  $t$ , such that  $x_j < u$ . The subscript  $L$  stands for the left child of node  $t$ .
- $P_R$  is the proportion of observations in node  $t$ , such that  $x_j \geq u$ . The subscript  $R$  stands for the right child of node  $t$ .
- $P_{L_j L_k}$  is the proportion of observations at node  $t$ , such that  $x_j < u$  and  $x_k < v$ .
- $P_{R_j R_k}$  is the proportion of observations at node  $t$ , such that  $x_j \geq u$  and  $x_k \geq v$ .
- Observations with missing values for  $x_j$  or  $x_k$  do not contribute to the proportion calculations.

$\lambda_{jk}$  is a value in  $(-\infty, 1]$ . If  $\lambda_{jk} > 0$ , then  $x_k < v$  is a worthwhile surrogate split for  $x_j < u$ .

## Surrogate Decision Splits

A surrogate decision split is an alternative to the optimal decision split at a given node in a decision tree. The optimal split is found by growing the tree; the surrogate split uses a similar or correlated predictor variable and split criterion.

When the value of the optimal split predictor for an observation is missing, the observation is sent to the left or right child node using the best surrogate predictor. When the value of the best surrogate split predictor for the observation is also missing, the observation is sent to the left or right child node using the second-best surrogate predictor, and so on. Candidate splits are sorted in descending order by their predictive measure of association on page 32-2664.

## Algorithms

Element  $ma(i, j)$  is the predictive measure of association averaged over surrogate splits on predictor  $j$  for which predictor  $i$  is the optimal split predictor. This average is computed by summing positive values of the predictive measure of association over optimal splits on predictor  $i$  and surrogate splits on predictor  $j$  and dividing by the total number of optimal splits on predictor  $i$ , including splits for which the predictive measure of association between predictors  $i$  and  $j$  is negative.

## See Also

`ClassificationTree` | `fitctree`

## surrogateAssociation

**Class:** CompactRegressionTree

Mean predictive measure of association for surrogate splits in decision tree

### Syntax

```
ma = surrogateAssociation(tree)
ma = surrogateAssociation(tree,N)
```

### Description

`ma = surrogateAssociation(tree)` returns a matrix of predictive measures of association for the predictors in `tree`.

`ma = surrogateAssociation(tree,N)` returns a matrix of predictive measures of association averaged over the nodes in vector `N`.

### Input Arguments

**tree**

A regression tree constructed with `fitrtree`, or a compact regression tree constructed with `compact`.

**N**

Vector of node numbers in `tree`.



## Output Arguments

**ma**

- `ma = surrogateAssociation(tree)` returns a P-by-P matrix, where P is the number of predictors in `tree`. `ma(i,j)` is the predictive measure of association on page 32-3988 between the optimal split on variable `i` and a surrogate split on variable `j`. For more details, see “Algorithms” on page 32-3989.
- `ma = surrogateAssociation(tree,N)` returns a P-by-P representing the predictive measure of association between variables averaged over nodes in the vector `N`. `N` contains node numbers from 1 to `max(tree.NumNodes)`.

## Examples

### Estimate Predictive Measures of Association for Surrogate Splits

Load the `carsmall` data set. Specify `Displacement`, `Horsepower`, and `Weight` as predictor variables.

```
load carsmall
X = [Displacement Horsepower Weight];
```

Grow a regression tree using `MPG` as the response. Specify to use surrogate splits for missing values.

```
tree = fitrtree(X,MPG,'surrogate','on');
```

Find the mean predictive measure of association between the predictor variables.

```
ma = surrogateAssociation(tree)
```

```
ma =
```

```
    1.0000    0.2167    0.5083
    0.4521    1.0000    0.3769
    0.2540    0.2659    1.0000
```

Find the mean predictive measure of association averaged over the odd-numbered nodes in tree.

```
N = 1:2:tree.NumNodes;
ma = surrogateAssociation(tree,N)
```

```
ma =
    1.0000    0.1250    0.6875
    0.5632    1.0000    0.5861
    0.3333    0.3148    1.0000
```

## Definitions

### Predictive Measure of Association

The predictive measure of association is a value that indicates the similarity between decision rules that split observations. Among all possible decision splits that are compared to the optimal split (found by growing the tree), the best surrogate decision split on page 32-2174 yields the maximum predictive measure of association. The second-best surrogate split has the second-largest predictive measure of association.

Suppose  $x_j$  and  $x_k$  are predictor variables  $j$  and  $k$ , respectively, and  $j \neq k$ . At node  $t$ , the predictive measure of association between the optimal split  $x_j < u$  and a surrogate split  $x_k < v$  is

$$\lambda_{jk} = \frac{\min(P_L, P_R) - (1 - P_{L_j L_k} - P_{R_j R_k})}{\min(P_L, P_R)}.$$

- $P_L$  is the proportion of observations in node  $t$ , such that  $x_j < u$ . The subscript  $L$  stands for the left child of node  $t$ .
- $P_R$  is the proportion of observations in node  $t$ , such that  $x_j \geq u$ . The subscript  $R$  stands for the right child of node  $t$ .
- $P_{L_j L_k}$  is the proportion of observations at node  $t$ , such that  $x_j < u$  and  $x_k < v$ .
- $P_{R_j R_k}$  is the proportion of observations at node  $t$ , such that  $x_j \geq u$  and  $x_k \geq v$ .

- Observations with missing values for  $x_j$  or  $x_k$  do not contribute to the proportion calculations.

$\lambda_{jk}$  is a value in  $(-\infty, 1]$ . If  $\lambda_{jk} > 0$ , then  $x_k < v$  is a worthwhile surrogate split for  $x_j < u$ .

## Surrogate Decision Splits

A surrogate decision split is an alternative to the optimal decision split at a given node in a decision tree. The optimal split is found by growing the tree; the surrogate split uses a similar or correlated predictor variable and split criterion.

When the value of the optimal split predictor for an observation is missing, the observation is sent to the left or right child node using the best surrogate predictor. When the value of the best surrogate split predictor for the observation is also missing, the observation is sent to the left or right child node using the second-best surrogate predictor, and so on. Candidate splits are sorted in descending order by their predictive measure of association on page 32-2664.

## Algorithms

Element  $ma(i, j)$  is the predictive measure of association averaged over surrogate splits on predictor  $j$  for which predictor  $i$  is the optimal split predictor. This average is computed by summing positive values of the predictive measure of association over optimal splits on predictor  $i$  and surrogate splits on predictor  $j$  and dividing by the total number of optimal splits on predictor  $i$ , including splits for which the predictive measure of association between predictors  $i$  and  $j$  is negative.

## See Also

`RegressionTree` | `fitrtree` | `prune`

## median

Median of probability distribution

### Syntax

```
m = median(pd)
```

### Description

`m = median(pd)` returns the median `m` for the probability distribution `pd`

### Examples

#### Median of a Fitted Distribution

Load the sample data. Create a vector containing the first column of students' exam grade data.

```
load examgrades
x = grades(:,1);
```

Create a normal distribution object by fitting it to the data.

```
pd = fitdist(x, 'Normal')
```

```
pd =
```

```
NormalDistribution
```

```
Normal distribution
```

```
mu = 75.0083 [73.4321, 76.5846]
sigma = 8.7202 [7.7391, 9.98843]
```

Compute the median of the fitted distribution.

```
m = median(pd)
```

```
m =
```

```
75.0083
```

For a symmetrical distribution such as the normal distribution, the median is equal to the mean,  $\mu$ .

### Median of a Skewed Distribution

Create a Weibull probability distribution object.

```
pd = makedist('Weibull', 'a', 5, 'b', 2)
```

```
pd =
```

```
WeibullDistribution
```

```
Weibull distribution
```

```
A = 5
```

```
B = 2
```

Compute the median of the distribution.

```
m = median(pd)
```

```
m =
```

```
4.1628
```

For a skewed distribution such as the Weibull distribution, the median and the mean may not be equal.

Calculate the mean of the Weibull distribution and compare it to the median.

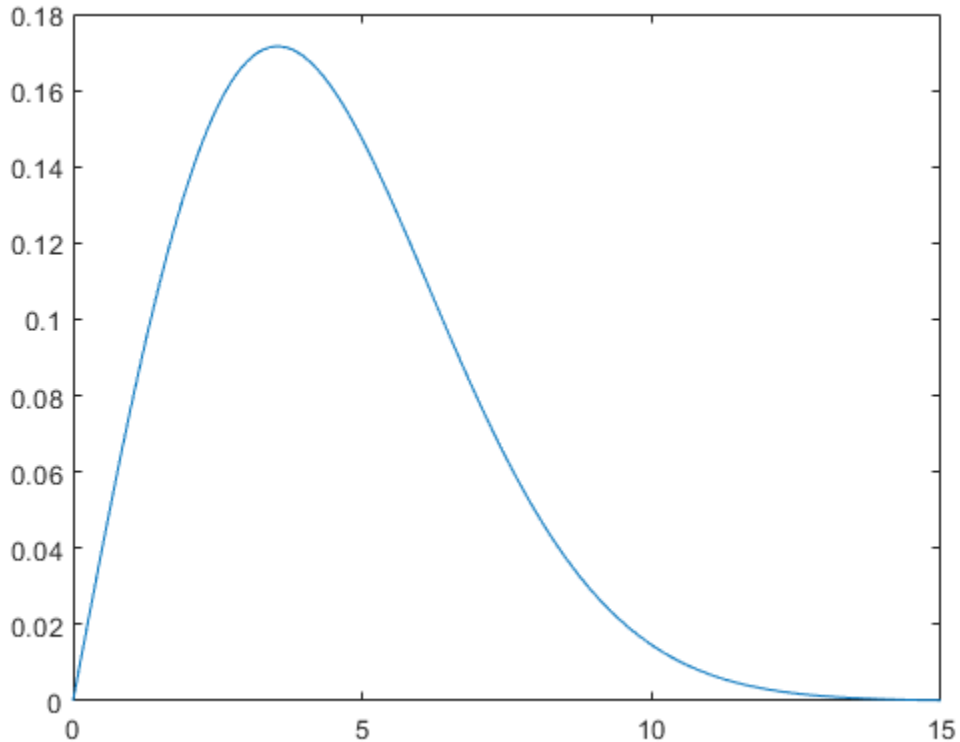
```
mean = mean(pd)
```

```
mean =  
    4.4311
```

The mean of the distribution is greater than the median.

Plot the pdf to visualize the distribution.

```
x = [0:.1:15];  
pdf = pdf(pd,x);  
plot(x,pdf)
```



## Input Arguments

**pd** — Probability distribution  
probability distribution object

Probability distribution, specified as a probability distribution object. Create a probability distribution object with specified parameter values using `makedist`. Alternatively, for fittable distributions, create a probability distribution object by fitting it to data using `fitdist` or the Distribution Fitting app.

## Output Arguments

### **m** — Median

scalar value

Median of the probability distribution, returned as a scalar value. The value of `m` is the 50th percentile of the probability distribution.

## See Also

`distributionFitter` | `fitdist` | `makedist` | `mean` | `median`

## Topics

“Working with Probability Distributions” on page 5-3

**Introduced before R2006a**



# median

**Class:** ProbDistUnivKernel

Return median of ProbDistUnivKernel object

## Syntax

$M = \text{median}(PD)$

## Description

$M = \text{median}(PD)$  returns  $M$ , the median of the ProbDistUnivKernel object  $PD$ .

## Input Arguments

$PD$  An object of the class ProbDistUnivKernel.

## Output Arguments

$M$  The median of the ProbDistUnivKernel object  $PD$ .

## See Also

median

## median

**Class:** ProbDistUnivParam

Return median of ProbDistUnivParam object

## Syntax

$M = \text{median}(PD)$

## Description

$M = \text{median}(PD)$  returns  $M$ , the median of the ProbDistUnivParam object  $PD$ .

## Input Arguments

$PD$  An object of the class ProbDistUnivParam.

## Output Arguments

$M$  The median of the ProbDistUnivParam object  $PD$ .

## See Also

median

---

# median

**Class:** prob.TruncatableDistribution

**Package:** prob

Median of probability distribution object

## Syntax

```
m = median(pd)
```

## Description

`m = median(pd)` returns the median `m` for the probability distribution `pd`.

## Input Arguments

**pd** — Probability distribution

probability distribution object

Probability distribution, specified as a probability distribution object. Create a probability distribution object with specified parameter values using `makedist`. Alternatively, for fittable distributions, create a probability distribution object by fitting it to data using `fitdist` or the Distribution Fitting app.

## Output Arguments

**m** — Median

scalar value

Median of the probability distribution, returned as a scalar value. The value of `m` is the 50th percentile of the probability distribution.

## Examples

### Median of a Fitted Distribution

Load the sample data. Create a vector containing the first column of students' exam grade data.

```
load examgrades;  
x = grades(:,1);
```

Create a normal distribution object by fitting it to the data.

```
pd = fitdist(x, 'Normal')  
  
pd =  
  
NormalDistribution  
  
Normal distribution  
mu = 75.0083 [73.4321, 76.5846]  
sigma = 8.7202 [7.7391, 9.98843]
```

Compute the median of the fitted distribution.

```
m = median(pd)  
  
m =  
  
75.0083
```

For a symmetrical distribution such as the normal distribution, the median will be equal to the mean,  $\mu$ .

### Median of a Skewed Distribution

Create a Weibull probability distribution object.

```
pd = makedist('Weibull', 'a', 5, 'b', 2)
```

```
pd =  
  
  WeibullDistribution  
  
  Weibull distribution  
    A = 5  
    B = 2
```

Compute the median of the distribution.

```
m = median(pd)
```

```
m =  
  
  4.1628
```

For a skewed distribution such as the Weibull distribution, the median and the mean may not be equal.

Calculate the mean of the Weibull distribution and compare it to the median.

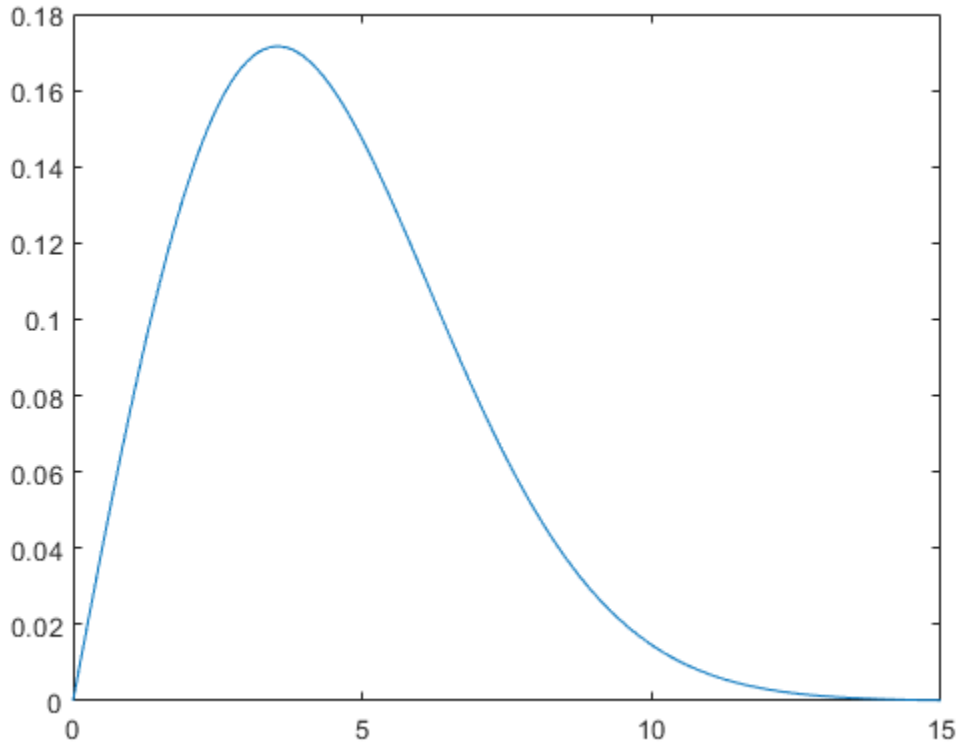
```
mean = mean(pd)
```

```
mean =  
  
  4.4311
```

The mean of the distribution is greater than the median.

Plot the pdf to visualize the distribution.

```
x = 0:.1:15;  
pdfx = pdf(pd,x);  
plot(x,pdfx)
```



## See Also

`distributionFitter` | `fitdist` | `makedist` | `mean` | `median`

## MergeLeaves property

**Class:** TreeBagger

Flag to merge leaves that do not improve risk

### Description

The `MergeLeaves` property is true if decision trees have their leaves with the same parent merged for splits that do not decrease the total risk, and false otherwise. The default value is false.

### See Also

`ClassificationTree` | `RegressionTree` | `TreeBagger` | `fitctree` | `fitrtree`

## mergelevels

Merge levels of nominal or ordinal arrays

---

**Note** The nominal and ordinal array data types might be removed in a future release. To represent ordered and unordered discrete, nonnumeric data, use the MATLAB® categorical data type instead.

---

### Syntax

```
B = mergelevels(A,oldlevels)
B = mergelevels(A,oldlevels,newlevel)
```

### Description

`B = mergelevels(A,oldlevels)` merges two or more levels of A.

- If A is a nominal array, `mergelevels` uses the first label in `oldlevels` as the new level.
- If A is an ordinal array, the levels specified by `oldlevels` must be consecutive, and `mergelevels` uses the label corresponding to the lowest level in `oldlevels` as the label for the new level.

`B = mergelevels(A,oldlevels,newlevel)` merges two or more levels into the new level with label `newlevel`.

### Examples

#### Create New Category From Merged Levels

Create a nominal array from data in a cell array.

```
colors = nominal({'r','b','g';'g','r','b';'b','r','g'},...
                {'blue','green','red'})
```



```

colors =
    3x3 nominal array
        red      blue      green
        green    red       blue
        blue     red       green

```

Merge the elements of the 'red' and 'blue' levels into a new level labeled 'purple'.

```

colors = mergelevels(colors, {'red', 'blue'}, 'purple')

```

```

colors =
    3x3 nominal array
        purple    purple    green
        green     purple    purple
        purple    purple    green

```

Display the levels of colors.

```

getlevels(colors)

```

```

ans =
    1x2 nominal array
        purple    green

```

- “Merge Category Levels” on page 2-22

## Input Arguments

**A** — Nominal or ordinal array

nominal array | ordinal array

Nominal or ordinal array, specified as a `nominal` or `ordinal` array object created using `nominal` or `ordinal`.

**oldlevels** — Levels to merge

cell array of character vectors | 2-D character array

Levels to merge, specified as a cell array of character vectors or 2-D character array. For ordinal arrays, the levels in `oldlevels` must be consecutive.

Data Types: `char` | `cell`

**newlevel** — Level to create

character vector

Level to create from the merged levels, specified as a character vector that gives the label for the new level.

Data Types: `char`

## Output Arguments

**B** — Nominal or ordinal array

nominal array | ordinal array

Nominal or ordinal array, returned as a `nominal` or `ordinal` array object.

## See Also

`addlevels` | `droplevels` | `nominal` | `ordinal` | `reorderlevels`

## Topics

“Merge Category Levels” on page 2-22

`nominal`

`ordinal`

Introduced in R2007a

## Method property

**Class:** TreeBagger

Method used by trees (classification or regression)

## Description

The Method property is 'classification' for classification ensembles and 'regression' for regression ensembles.

## mhsample

Metropolis-Hastings sample

### Syntax

```
smp1 = mhsample(start, nsamples, 'pdf', pdf, 'proppdf', proppdf,
'proprnd', proprnd)
smp1 = mhsample(..., 'symmetric', sym)
smp1 = mhsample(..., 'burnin', K)
smp1 = mhsample(..., 'thin', m)
smp1 = mhsample(..., 'nchain', n)
[smp1, accept] = mhsample(...)
```

### Description

`smp1 = mhsample(start, nsamples, 'pdf', pdf, 'proppdf', proppdf, 'proprnd', proprnd)` draws `nsamples` random samples from a target stationary distribution `pdf` using the Metropolis-Hastings algorithm.

`start` is a row vector containing the start value of the Markov Chain, `nsamples` is an integer specifying the number of samples to be generated, and `pdf`, `proppdf`, and `proprnd` are function handles created using `@`. `proppdf` defines the proposal distribution density, and `proprnd` defines the random number generator for the proposal distribution. `pdf` and `proprnd` take one argument as an input with the same type and size as `start`. `proppdf` takes two arguments as inputs with the same type and size as `start`.

`smp1` is a column vector or matrix containing the samples. If the log density function is preferred, `'pdf'` and `'proppdf'` can be replaced with `'logpdf'` and `'logproppdf'`. The density functions used in Metropolis-Hastings algorithm are not necessarily normalized.

The proposal distribution  $q(x,y)$  gives the probability density for choosing  $x$  as the next point when  $y$  is the current point. It is sometimes written as  $q(x|y)$ .

If the `proppdf` or `logproppdf` satisfies  $q(x,y) = q(y,x)$ , that is, the proposal distribution is symmetric, `mhsample` implements Random Walk Metropolis-Hastings sampling. If the `proppdf` or `logproppdf` satisfies  $q(x,y) = q(x)$ , that is, the proposal distribution is independent of current values, `mhsample` implements Independent Metropolis-Hastings sampling.

`smpl = mhsample(..., 'symmetric', sym)` draws `nsamples` random samples from a target stationary distribution `pdf` using the Metropolis-Hastings algorithm. `sym` is a logical value that indicates whether the proposal distribution is symmetric. The default value is `false`, which corresponds to the asymmetric proposal distribution. If `sym` is `true`, for example, the proposal distribution is symmetric, `proppdf` and `logproppdf` are optional.

`smpl = mhsample(..., 'burnin', K)` generates a Markov chain with values between the starting point and the  $k^{\text{th}}$  point omitted in the generated sequence. Values beyond the  $k^{\text{th}}$  point are kept. `k` is a nonnegative integer with default value of 0.

`smpl = mhsample(..., 'thin', m)` generates a Markov chain with  $m-1$  out of  $m$  values omitted in the generated sequence. `m` is a positive integer with default value of 1.

`smpl = mhsample(..., 'nchain', n)` generates `n` Markov chains using the Metropolis-Hastings algorithm. `n` is a positive integer with a default value of 1. `smpl` is a matrix containing the samples. The last dimension contains the indices for individual chains.

`[smpl, accept] = mhsample(...)` also returns `accept`, the acceptance rate of the proposed distribution. `accept` is a scalar if a single chain is generated and is a vector if multiple chains are generated.

## Examples

### Estimate Moments Using Independent Metropolis-Hastings Sampling

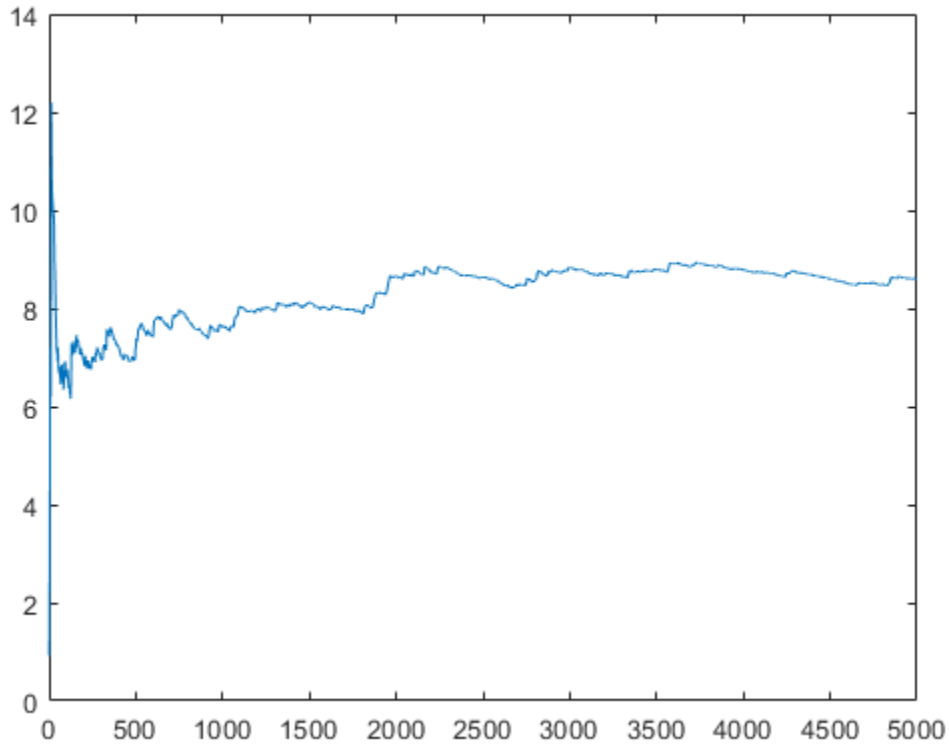
Use Independent Metropolis-Hastings sampling to estimate the second order moment of a Gamma distribution.

```
rng default; % For reproducibility
alpha = 2.43;
```

```
beta = 1;
pdf = @(x)gampdf(x,alpha,beta); % Target distribution
proppdf = @(x,y)gampdf(x,floor(alpha),floor(alpha)/alpha);
proprnd = @(x)sum(...
    exprnd(floor(alpha)/alpha,floor(alpha),1));
nsamples = 5000;
smp1 = mhsample(1,nsamples,'pdf',pdf,'proprnd',proprnd,...
    'proppdf',proppdf);
```

Plot the results.

```
xxhat = cumsum(smp1.^2)./(1:nsamples)';
figure;
plot(1:nsamples,xxhat)
```



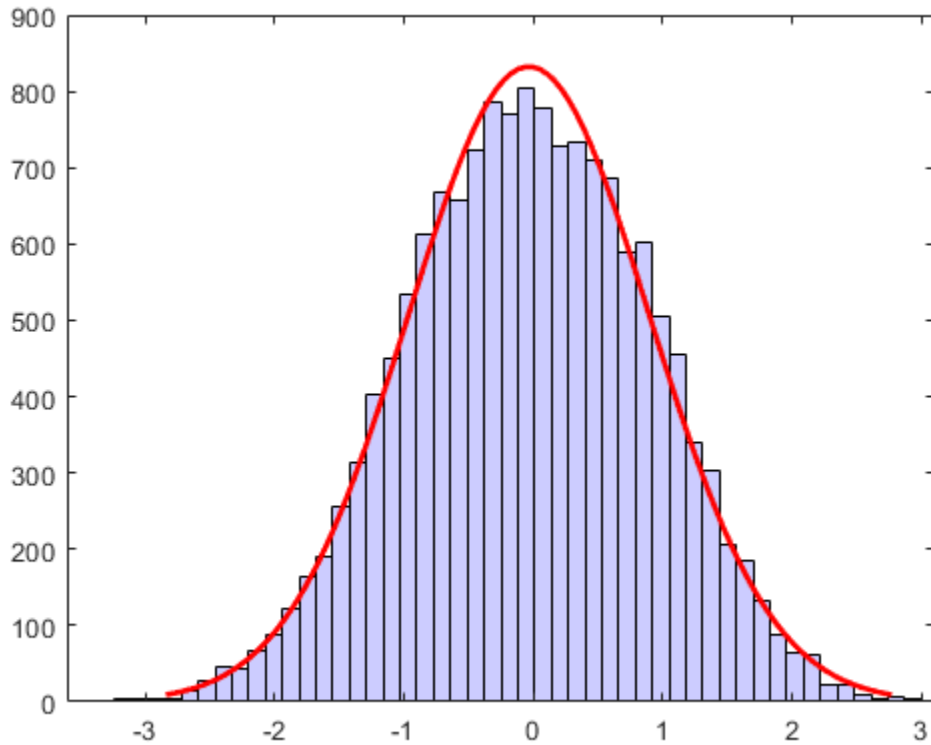
### Random Walk Metropolis-Hastings Sampling

Use Random Walk Metropolis-Hastings sampling to generate sample data from a standard normal distribution.

```
rng default % For reproducibility
delta = .5;
pdf = @(x) normpdf(x);
proppdf = @(x,y) unifpdf(y-x,-delta,delta);
proprnd = @(x) x + rand*2*delta - delta;
nsamples = 15000;
x = mhsample(1,nsamples,'pdf',pdf,'proprnd',proprnd,'symmetric',1);
```

Plot the sample data.

```
figure;  
h = histfit(x,50);  
h(1).FaceColor = [.8 .8 1];
```



- “Using the Metropolis-Hastings Algorithm” on page 7-14

## See Also

rand | slicesample



## **Topics**

“Using the Metropolis-Hastings Algorithm” on page 7-14

**Introduced in R2006a**

## MinLeafSize property

**Class:** TreeBagger

Minimum number of observations per tree leaf

### Description

The `MinLeafSize` property specifies the minimum number of observations per tree leaf. The default values are 1 for classification and 5 for regression. For `fitctree` or `fitrtree` training, the `'MinParentSize'` value is set to  $2 * \text{MinLeafSize}$ .

### See Also

`ClassificationTree` | `RegressionTree` | `TreeBagger` | `fitctree` | `fitrtree`

# mle

Maximum likelihood estimates

## Syntax

```
phat = mle(data)
phat = mle(data, 'distribution', dist)

phat = mle(data, 'pdf', pdf, 'start', start)
phat = mle(data, 'pdf', pdf, 'start', start, 'cdf', cdf)

phat = mle(data, 'logpdf', logpdf, 'start', start)
phat = mle(data, 'logpdf', logpdf, 'start', start, 'logsf', logsf)

phat = mle(data, 'nloglf', nloglf, 'start', start)

phat = mle(___, Name, Value)
[phat, pci] = mle(___)
```

## Description

`phat = mle(data)` returns maximum likelihood estimates (MLEs) for the parameters of a normal distribution, using the sample data in the vector `data`.

`phat = mle(data, 'distribution', dist)` returns parameter estimates for a distribution specified by `dist`.

`phat = mle(data, 'pdf', pdf, 'start', start)` returns parameter estimates for a custom distribution specified by the probability density function `pdf`. You must also specify the initial parameter values, `start`.

`phat = mle(data, 'pdf', pdf, 'start', start, 'cdf', cdf)` returns parameter estimates for a custom distribution specified by the probability density function `pdf` and custom cumulative distribution function `cdf`.

`phat = mle(data, 'logpdf', logpdf, 'start', start)` returns parameter estimates for a custom distribution specified by the log probability density function `logpdf`. You must also specify the initial parameter values, `start`.

`phat = mle(data, 'logpdf', logpdf, 'start', start, 'logsf', logsf)` returns parameter estimates for a custom distribution specified by the log probability density function `logpdf` and custom log survival function on page 32-4031 `logsf`.

`phat = mle(data, 'nloglf', nloglf, 'start', start)` returns parameter estimates for the custom distribution specified by the negative loglikelihood function `nloglf`. You must also specify the initial parameter values, `start`.

`phat = mle(____, Name, Value)` also returns the parameter estimates with additional options specified by one or more name-value pair arguments. You can use any of the input arguments in the previous syntaxes.

`[phat, pci] = mle(____)` also returns the 95% confidence intervals for the parameters.

## Examples

### Estimate Parameters of Burr Distribution

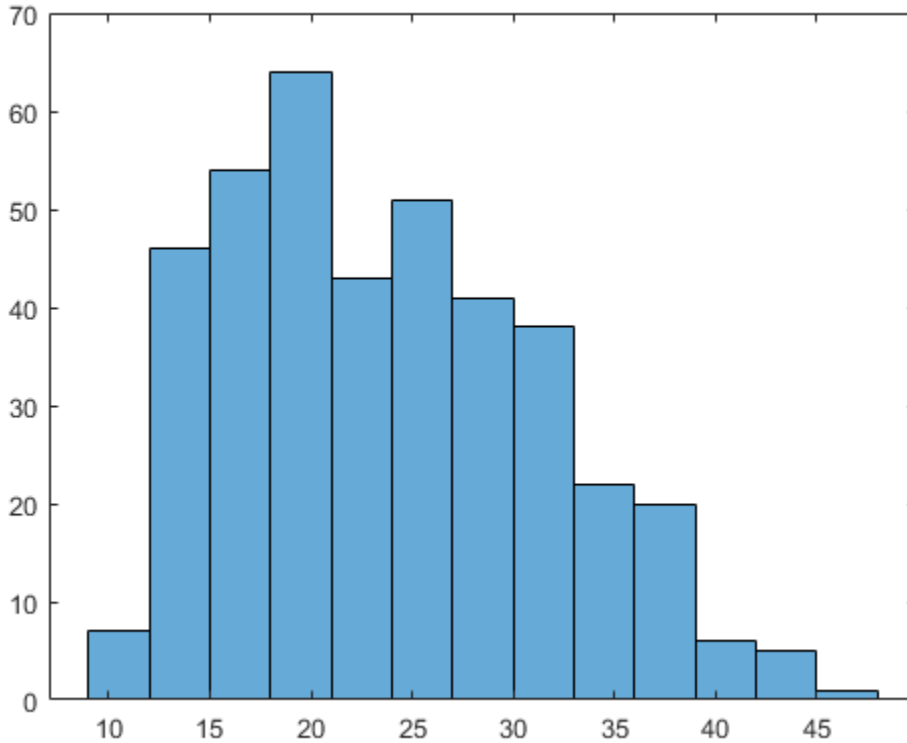
Load the sample data.

```
load carbig
```

The variable `MPG` has the miles per gallon for different models of cars.

Draw a histogram of `MPG` data.

```
histogram(MPG)
```



The distribution is somewhat right skewed. A symmetric distribution, such as normal distribution, might not be a good fit.

Estimate the parameters of the Burr Type XII distribution for the MPG data.

```
phat = mle(MPG, 'distribution', 'burr')
```

```
phat =
```

```
34.6447    3.7898    3.5722
```

The maximum likelihood estimates for the scale parameter  $\alpha$  is 34.6447. The estimates for the two shape parameters  $c$  and  $k$  of the Burr Type XII distribution are 3.7898 and 3.5722, respectively.

### Estimate Parameters of a Noncentral Chi-Square Distribution

Generate sample data of size 1000 from a noncentral chi-square distribution with degrees of freedom 8 and noncentrality parameter 3.

```
rng default % for reproducibility
x = ncx2rnd(8,3,1000,1);
```

Estimate the parameters of the noncentral chi-square distribution from the sample data. To do this, custom define the noncentral chi-square pdf using the `pdf` input argument.

```
[phat,pci] = mle(x,'pdf',@(x,v,d)ncx2pdf(x,v,d),'start',[1,1])
```

```
phat =
```

```
8.1052    2.6693
```

```
pci =
```

```
7.1121    1.6025
9.0983    3.7362
```

The estimate for the degrees of freedom is 8.1052 and the noncentrality parameter is 2.6693. The 95% confidence interval for the degrees of freedom is (7.1121,9.0983) and the noncentrality parameter is (1.6025,3.7362). The confidence intervals include the true parameter values of 8 and 3, respectively.

### Fit Custom Distribution to Censored Data

Load the sample data.

```
load(fullfile(matlabroot,'examples','stats','readmissiontimes.mat'));
```

The data includes `ReadmissionTime`, which has readmission times for 100 patients. The column vector `Censored` has the censorship information for each patient, where 1 indicates a censored observation, and 0 indicates the exact readmission time is observed. This is simulated data.

Define a custom probability density and cumulative distribution function.

```
custpdf = @(data,lambda) lambda*exp(-lambda*data);
custcdf = @(data,lambda) 1-exp(-lambda*data);
```

Estimate the parameter, `lambda`, of the custom distribution for the censored sample data.

```
phat = mle(ReadmissionTime, 'pdf', custpdf, 'cdf', custcdf, 'start', 0.05, 'Censoring', Censored);

phat =

    0.1096
```

## Fit Custom Log pdf and Survival Function

Load the sample data.

```
load(fullfile(matlabroot, 'examples', 'stats', 'readmissiontimes.mat'));
```

The data includes `ReadmissionTime`, which has readmission times for 100 patients. The column vector `Censored` has the censorship information for each patient, where 1 indicates a censored observation, and 0 indicates the exact readmission time is observed. This is simulated data.

Define a custom log probability density and survival function.

```
custlogpdf = @(data,lambda,k) log(k)-k*log(lambda)+(k-1)*log(data)-(data/lambda).^k;
custlogsf = @(data,lambda,k) -(data/lambda).^k;
```

Estimate the parameters, `lambda` and `k`, of the custom distribution for the censored sample data.

```
phat = mle(ReadmissionTime, 'logpdf', custlogpdf, 'logsf', custlogsf, ...
    'start', [1, 0.75], 'Censoring', Censored)
```

```
phat =  
    9.2090    1.4223
```

The scale and shape parameters of the custom-defined distribution are 9.2090 and 1.4223, respectively.

### Fit Custom Log Negative Likelihood Function

Load the sample data.

```
load(fullfile(matlabroot, 'examples', 'stats', 'readmissiontimes.mat'));
```

The data includes `ReadmissionTime`, which has readmission times for 100 patients. This is simulated data.

Define a negative log likelihood function.

```
custnloglf = @(lambda,data,cens,freq) - length(data)*log(lambda) + nansum(lambda*data);
```

Estimate the parameters of the defined distribution.

```
phat = mle(ReadmissionTime, 'nloglf', custnloglf, 'start', 0.05)
```

```
phat =  
    0.1462
```

### Estimate Probability of Success

Generate 100 random observations from a binomial distribution with the number of trials,  $n = 20$ , and the probability of success,  $P = 0.75$ .

```
data = binornd(20, 0.75, 100, 1);
```

Estimate the probability of success and 95% confidence limits using the simulated sample data.



```
[phat,pci] = mle(data, 'distribution', 'binomial', 'alpha', .05, 'ntrials', 20)
```

```
phat =
```

```
0.7615
```

```
pci =
```

```
0.7422
```

```
0.7800
```

The estimate of probability of success is 0.7615 and the lower and upper limits of the 95% confidence interval are 0.7422 and 0.78. This interval covers the true value used to simulate the data.

### Fit Distribution with Known Parameter

Generate sample data of size 1000 from a noncentral chi-square distribution with degrees of freedom 10 and noncentrality parameter 5.

```
rng default % for reproducibility  
x = ncx2rnd(10,5,1000,1);
```

Suppose the noncentrality parameter is fixed at the value 5. Estimate the degrees of freedom of the noncentral chi-square distribution from the sample data. To do this, custom define the noncentral chi-square pdf using the pdf input argument.

```
[phat,pci] = mle(x, 'pdf', @(x,v,d)ncx2pdf(x,v,5), 'start', 1)
```

```
phat =
```

```
9.9307
```

```
pci =
```

```
9.5626
```

```
10.2989
```

The estimate for the noncentrality parameter is 9.9307, with a 95% confidence interval of 9.5626 and 10.2989. The confidence interval includes the true parameter value of 10.

### Fit Rician Distribution with Known Scale Parameter

Generate sample data of size 1000 from a Rician distribution with noncentrality parameter of 8 and scale parameter of 5. First create the Rician distribution.

```
r = makedist('Rician','s',8,'sigma',5);
```

Now, generate sample data from the distribution you created above.

```
rng default % For reproducibility  
x = random(r,1000,1);
```

Suppose the scale parameter is known, and estimate the noncentrality parameter from sample data. To do this using `mle`, you must custom define the Rician probability density function.

```
[phat,pci] = mle(x,'pdf',@(x,s,sigma) pdf('rician',x,s,5),'start',10)
```

```
phat =
```

```
7.8953
```

```
pci =
```

```
7.5405
```

```
8.2501
```

The estimate for the noncentrality parameter is 7.8953, with a 95% confidence interval of 7.5404 and 8.2501. The confidence interval includes the true parameter value of 8.

### Fit a Distribution with Additional Parameter

Add a scale parameter to the chi-square distribution for adapting to the scale of data and fit it. First, generate sample data of size 1000 from a chi-square distribution with degrees of freedom 5, and scale it by the factor of 100.

```
rng default % For reproducibility
x = 100*chi2rnd(5,1000,1);
```

Estimate the degrees of freedom and the scaling factor. To do this, custom define the chi-square probability density function using the `pdf` input argument. The density function requires a  $1/s$  factor for data scaled by  $s$ .

```
[phat,pci] = mle(x,'pdf',@(x,v,s)chi2pdf(x/s,v)/s,'start',[1,200])
```

```
phat =
```

```
    5.1079    99.1681
```

```
pci =
```

```
    4.6862    90.1215
    5.5297   108.2146
```

The estimate for the degrees of freedom is 5.1079 and the scale is 99.1681. The 95% confidence interval for the degrees of freedom is (4.6862,5.5279) and the scale parameter is (90.1215,108.2146). The confidence intervals include the true parameter values of 5 and 100, respectively.

## Input Arguments

### **data** — Sample data

vector

Sample data `mle` uses to estimate the distribution parameters, specified as a vector.

Data Types: `single` | `double`

### **dist** — Distribution type

'normal' (default) | character vector

Distribution type to estimate parameters for, specified as one of the following.

dist	Description	Parameter 1	Parameter 2	Parameter 3	Parameter 4
'bernoulli'	“Bernoulli Distribution” on page B-2	p: probability of success for each trial	—	—	—
'beta' or 'Beta'	“Beta Distribution” on page B-4	a	b	—	—
'bino' or 'Binomial'	“Binomial Distribution” on page B-9	n: number of trials	p: probability of success for each trial	—	—
'birnbaumsaunders'	“Birnbaum-Saunders Distribution” on page B-13	$\beta$ : scale	$\gamma$ : shape	—	—
'burr' or 'Burr'	“Burr Type XII Distribution” on page B-15	$\alpha$ : scale	c: first shape	k: second shape	—
'Discrete Uniform' or 'unid'	“Uniform Distribution (Discrete)” on page B-190	N: maximum observable value	—	—	—
'exp' or 'Exponential'	“Exponential Distribution” on page B-34	$\mu$ : mean	—	—	—
'ev' or 'Extreme Value'	“Extreme Value Distribution” on page B-38	$\mu$ : location	$\sigma$ : scale	—	—
'gam' or 'Gamma'	“Gamma Distribution” on page B-47	a: shape	b: scale	—	—
'gev' or 'Generalized Extreme Value'	“Generalized Extreme Value Distribution” on page B-53	k: shape	$\sigma$ : scale	$\mu$ : location	—
'gp' or 'Generalized Pareto'	“Generalized Pareto Distribution” on page B-58	k: tail index (shape)	$\sigma$ : scale	$\theta$ : threshold	—

dist	Description	Parameter 1	Parameter 2	Parameter 3	Parameter 4
'geo' or 'Geometric'	“Geometric Distribution” on page B-64	$p$ : probability	—	—	—
'hn' or 'Half Normal'	“Half-Normal Distribution” on page B-73	$\mu$ : location	$\sigma$ : scale	—	—
'inversegaussian'	“Inverse Gaussian Distribution” on page B-82	$\mu$ : scale	$\lambda$ : shape	—	—
'logistic'	“Logistic Distribution” on page B-96	$\mu$ : location	$\sigma$ : scale	—	—
'loglogistic'	“Loglogistic Distribution” on page B-98	$\mu$ : log location	$\sigma$ : log scale	—	—
'logn' or 'Lognormal'	“Lognormal Distribution” on page B-100	$\mu$ : log location	$\sigma$ : log scale	—	—
'nakagami'	“Nakagami Distribution” on page B-118	$\mu$ : shape	$\omega$ : scale	—	—
'nbin' or 'Negative Binomial'	“Negative Binomial Distribution” on page B-120	$r$ : number of successes	$p$ : probability of success in a single trial	—	—
'norm' or 'Normal'	“Normal Distribution” on page B-134	$\mu$ : location (mean)	$\sigma$ : scale (standard deviation)	—	—
'poiss' or 'Poisson'	“Poisson Distribution” on page B-143	$\lambda$ : mean	—	—	—
'rayl' or 'Rayleigh'	“Rayleigh Distribution” on page B-146	$b$ : scale	—	—	—
'rician'	“Rician Distribution” on page B-149	$s$ : noncentrality	$\sigma$ : scale	—	—

dist	Description	Parameter 1	Parameter 2	Parameter 3	Parameter 4
'stable'	“Stable Distribution” on page B-151	$\alpha$ : first shape parameter	$\beta$ : second shape parameter	$\gamma$ : scale parameter	$\delta$ : location parameter
'tlocationscale'	“t Location-Scale Distribution” on page B-175	$\mu$ : location	$\sigma$ : scale	$\nu$ : degrees of freedom	—
'unif' or 'Uniform'	“Uniform Distribution (Continuous)” on page B-184	a: lower endpoint (minimum)	b: upper endpoint (maximum)	—	—
'wbl' or 'Weibull'	“Weibull Distribution” on page B-193	a: scale	b: shape	—	—

Example: 'rician'

#### pdf — Custom probability density function function handle

Custom probability distribution function, specified as a function handle created using @.

This custom function accepts the vector `data` and one or more individual distribution parameters as input parameters, and returns a vector of probability density values.

For example, if the name of the custom probability density function is `newpdf`, then you can specify the function handle in `mle` as follows.

Example: `@newpdf`

Data Types: `function_handle`

#### cdf — Custom cumulative distribution function function handle

Custom cumulative distribution function, specified as a function handle created using @.

This custom function accepts the vector `data` and one or more individual distribution parameters as input parameters, and returns a vector of cumulative probability values.

You must define `cdf` with `pdf` if data is censored and you use the `'censoring'` name-value pair argument. If `'censoring'` is not present, you do not have to specify `cdf` while using `pdf`.

For example, if the name of the custom cumulative distribution function is `newcdf`, then you can specify the function handle in `mle` as follows.

Example: `@newcdf`

Data Types: `function_handle`

### **logpdf — Custom log probability density function**

function handle

Custom log probability density function, specified as a function handle created using `@`.

This custom function accepts the vector `data` and one or more individual distribution parameters as input parameters, and returns a vector of log probability values.

For example, if the name of the custom log probability density function is `customlogpdf`, then you can specify the function handle in `mle` as follows.

Example: `@customlogpdf`

Data Types: `function_handle`

### **logsf — Custom log survival function**

function handle

Custom log survival function on page 32-4031, specified as a function handle created using `@`.

This custom function accepts the vector `data` and one or more individual distribution parameters as input parameters, and returns a vector of log survival probability values.

You must define `logsf` with `logpdf` if data is censored and you use the `'censoring'` name-value pair argument. If `'censoring'` is not present, you do not have to specify `logsf` while using `logpdf`.

For example, if the name of the custom log survival function is `logsurvival`, then you can specify the function handle in `mle` as follows.

Example: `@logsurvival`

Data Types: `function_handle`

### **nloglf** — Custom negative loglikelihood function

function handle

Custom negative loglikelihood function, specified as a function handle created using `@`.

This custom function accepts the following input arguments.

<code>params</code>	Vector of distribution parameter values. <code>mle</code> detects the number of parameters from the number of elements in <code>start</code> .
<code>data</code>	Vector of data.
<code>cens</code>	Boolean vector of censored values.
<code>freq</code>	Vector of integer data frequencies.

`nloglf` must accept all four arguments even if you do not use the `'censoring'` or `'frequency'` name-value pair arguments. You can write `'nloglf'` to ignore `cens` and `freq` arguments in that case.

`nloglf` returns a scalar negative loglikelihood value and optionally, a negative loglikelihood gradient vector (see the `'GradObj'` field in `'options'`).

If the name of the custom negative log likelihood function is `negloglik`, then you can specify the function handle in `mle` as follows.

Example: `@negloglik`

Data Types: `function_handle`

### **start** — Initial parameter values

scalar | vector

Initial parameter values for the custom functions, specified as a scalar value or a vector of scalar values.

Use `start` when you fit custom distributions, that is, when you use `pdf` and `cdf`, `logpdf` and `logsf`, or `nloglf` input arguments.

Example: `0.05`

Example: `[100,2]`

Data Types: `single` | `double`



## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'Censoring', Cens, 'Alpha', 0.01, 'Options', Opt` specifies that `mle` estimates the parameters for the distribution of censored data specified by array `Cens`, computes the 99% confidence limits for the parameter estimates, and uses the algorithm control parameters specified by the structure `Opt`.

### **Censoring** — Indicator for censoring

array of 0s (default) | array of 0s and 1s

Indicator for censoring, specified as the comma-separated pair consisting of `'Censoring'` and a Boolean array of the same size as `data`. Use 1 for observations that are right censored and 0 for observations that are fully observed. The default is all observations are fully observed.

For example, if the censored data information is in the binary array called `Censored`, then you can specify the censored data as follows.

Example: `'censoring', Censored`

`mle` supports censoring for the following distributions:

Birnbaum-Saunders	Logistic
Burr	Lognormal
Exponential	Nakagami
Extreme Value	Normal
Gamma	Rician
Inverse Gaussian	t Location-Scale
Kernel	Weibull
Log-Logistic	

Data Types: `logical`

### **Frequency** — Frequency of observations

array of 1s (default) | vector of nonnegative integer counts

Frequency of observations, specified as the comma-separated pair consisting of 'frequency' and an array containing nonnegative integer counts, which is the same size as data. The default is one observation per element of data.

For example, if the observation frequencies are stored in an array named `Freq`, you can specify the frequencies as follows.

Example: `'frequency', Freq`

Data Types: `single | double`

### **Alpha — Confidence level**

0.05 (default) | scalar value in the range (0,1)

Confidence level for the confidence interval of parameter estimates, `pci`, specified as the comma-separated pair consisting of 'alpha' and a scalar value in the range (0,1). The confidence level of `pci` is  $100(1-\text{alpha})\%$ . The default is 0.05 for 95% confidence.

For example, for 99% confidence limits, you can specify the confidence level as follows.

Example: `'alpha', 0.01`

Data Types: `single | double`

### **NTrials — Number of trials**

scalar value | vector

Number of trials for the corresponding element of data, specified as the comma-separated pair consisting of 'ntrials' and a scalar or a vector of the same size as data.

Applies only to binomial distribution.

Example: `'ntrials', total`

Data Types: `single | double`

### **mu — Location parameter**

0 (default) | scalar value

Location parameter for the half-normal distribution, specified as the comma-separated pair consisting of 'mu' and a scalar value.

Applies only to half-normal distribution.

Example: `'mu', 1`

Data Types: `single` | `double`

### Options — Fitting algorithm control parameters

structure

Fitting algorithm control parameters, specified as the comma-separated pair consisting of 'options' and a structure returned by `statset`.

Not applicable to all distributions.

Use the 'options' name-value pair argument to control details of the maximum likelihood optimization when fitting a custom distribution. For parameter names and default values, type `statset('mlecustom')`. You can set the options under a new name and use that in the name-value pair argument. `mle` interprets the following `statset` parameters for custom distribution fitting.

Parameter	Value
'GradObj'	<p>Default is 'off'.</p> <p>'on' or 'off', indicating whether or not <code>fmincon</code> can expect the custom function provided with the <code>nloglf</code> input argument to return the gradient vector of the negative log-likelihood as a second output.</p> <p><code>mle</code> ignores 'GradObj' when using <code>fminsearch</code>.</p>
'DerivStep'	<p>Default is <math>\text{eps}^{(1/3)}</math>.</p> <p>The relative difference, specified as a scalar or a vector the same size as <code>start</code>, used in finite difference derivative approximations when using <code>fmincon</code>, and 'GradObj' is 'off'.</p> <p><code>mle</code> ignores 'DerivStep' when using <code>fminsearch</code>.</p>
'FunValCheck'	<p>Default is 'on'.</p> <p>'on' or 'off', indicating whether or not <code>mle</code> should check the values returned by the custom distribution functions for validity.</p> <p>A poor choice of starting point can sometimes cause these functions to return NaNs, infinite values, or out-of-range values if they are written without suitable error checking.</p>

Parameter	Value
'TolBnd'	<p>Default is <math>1e-6</math>.</p> <p>An offset for lower and upper bounds when using <code>fmincon</code>.</p> <p><code>mle</code> treats lower and upper bounds as strict inequalities, that is, open bounds. With <code>fmincon</code>, this is approximated by creating closed bounds inset from the specified lower and upper bounds by <code>TolBnd</code>.</p>

Example: `'options', statset('mlecustom')`

Data Types: `struct`

### **LowerBound** — Lower bounds for distribution parameters

$-\infty$  (default) | vector

Lower bounds for distribution parameters, specified as the comma-separated pair consisting of `'lowerbound'` and a vector the same size as `start`.

This name-value pair argument is valid only when you use the `pdf` and `cdf`, `logpdf` and `logcdf`, or `nloglf` input arguments.

Example: `'lowerbound', 0`

Data Types: `single` | `double`

### **UpperBound** — Upper bounds for distribution parameters

$\infty$  (default) | vector

Upper bounds for distribution parameters, specified as the comma-separated pair consisting of `'upperbound'` and a vector the same size as `start`.

This name-value pair argument is valid only when you use the `pdf` and `cdf`, `logpdf` and `logsf`, or `nloglf` input arguments.

Example: `'upperbound', 1`

Data Types: `single` | `double`

### **OptimFun** — Optimization function

`'fminsearch'` (default) | `'fmincon'`

Optimization function `mle` uses in maximizing the likelihood, specified as the comma-separated pair consisting of `'optimfun'` and either `'fminsearch'` or `'fmincon'`.

Default is `'fminsearch'`.

You can only specify `'fmincon'` if Optimization Toolbox is available.

The `'optimfun'` name-value pair argument is valid only when you fit custom distributions, that is, when you use the `pdf` and `cdf`, `logpdf` and `logsf`, or `nloglf` input arguments.

Example: `'optimfun','fmincon'`

## Output Arguments

### **phat** — Parameter estimates

scalar value | row vector

Parameter estimates, returned as a scalar value or a row vector.

### **pci** — Confidence intervals for parameter estimates

2-by- $k$  matrix

Confidence intervals for parameter estimates, returned as a column vector or a matrix depending on the number of parameters, hence the size of `phat`.

`pci` is a 2-by- $k$  matrix, where  $k$  is the number of parameters `mle` estimates. The first and second rows of the `pci` show the lower and upper confidence limits, respectively.

## Definitions

### Survival Function

The survival function is the probability of survival as a function of time. It is also called the survivor function. It gives the probability that the survival time of an individual exceeds a certain value. Since the cumulative distribution function,  $F(t)$ , is the probability that the survival time is less than or equal to a given point in time, the

survival function for a continuous distribution,  $S(t)$ , is the complement of the cumulative distribution function:  $S(t) = 1 - F(t)$ .

## Tips

When you supply distribution functions, `mle` computes the parameter estimates using an iterative maximization algorithm. With some models and data, a poor choice of starting point can cause `mle` to converge to a local optimum that is not the global maximizer, or to fail to converge entirely. Even in cases for which the log-likelihood is well-behaved near the global maximum, the choice of starting point is often crucial to convergence of the algorithm. In particular, if the initial parameter values are far from the MLEs, underflow in the distribution functions can lead to infinite log-likelihoods.

## See Also

`fitdist` | `mlecov` | `statset`

## Topics

“What Is Survival Analysis?” on page 14-2

“Estimate Parameters of a Three-Parameter Weibull Distribution” on page B-195

**Introduced before R2006a**

# mlecov

Asymptotic covariance of maximum likelihood estimators

## Syntax

```
acov = mlecov(params,data,'pdf',pdf)
acov = mlecov(params,data,'pdf',pdf,'cdf',cdf)

acov = mlecov(params,data,'logpdf',logpdf)
acov = mlecov(params,data,'logpdf',logpdf,'logsf',logsf)

acov = mlecov(params,data,'nloglf',nloglf)

acov = mlecov( ____,Name,Value)
```

## Description

`acov = mlecov(params,data,'pdf',pdf)` returns an approximation to the asymptotic covariance matrix of the maximum likelihood estimators of the parameters for a distribution specified by the custom probability density function `pdf`.

`mlecov` computes a finite difference approximation to the Hessian of the log-likelihood at the maximum likelihood estimates `params`, given the observed data, and returns the negative inverse of that Hessian.

`acov = mlecov(params,data,'pdf',pdf,'cdf',cdf)` returns an approximation to the asymptotic covariance matrix of the maximum likelihood estimators of the parameters for a distribution specified by the custom probability density function `pdf` and cumulative distribution function `cdf`.

`acov = mlecov(params,data,'logpdf',logpdf)` returns an approximation to the asymptotic covariance matrix of the maximum likelihood estimators of the parameters for a distribution specified by the custom log probability density function `logpdf`.

`acov = mlecov(params,data,'logpdf',logpdf,'logsf',logsf)` returns an approximation to the asymptotic covariance matrix of the maximum likelihood

estimators of the parameters for a distribution specified by the custom log probability density function `logpdf` and custom log survival function on page 32-4042 `logsf`.

`acov = mlecov(params,data,'nloglf',nloglf)` returns an approximation to the asymptotic covariance matrix of the maximum likelihood estimators of the parameters for a distribution specified by the custom negative loglikelihood function `nloglf`.

`acov = mlecov(___,Name,Value)` also returns an approximation to the asymptotic covariance matrix of the maximum likelihood estimators of the parameters with additional options specified by one or more name-value pair arguments. You can use any of the input arguments in the previous syntaxes.

## Examples

### Custom Probability Density Function

Load the sample data.

```
load carbig
```

The vector `Weight` shows the weights of 406 cars.

In the MATLAB Editor, create a function that returns the probability density function (pdf) of a lognormal distribution. Save the file in your current working folder as `lognormpdf.m`.

```
function newpdf = lognormpdf(data,mu,sigma)
newpdf = exp((-log(data)-mu).^2)/(2*sigma^2) ./ (data*sigma*sqrt(2*pi));
```

Estimate the parameters, `mu` and `sigma`, of the custom-defined distribution.

```
phat = mle(Weight,'pdf',@lognormpdf,'start',[4.5 0.3])
```

```
phat =
```

```
    7.9600    0.2804
```

Compute the approximate covariance matrix of the parameter estimates.

```
acov = mlecov(phat,Weight,'pdf',@lognormpdf)
```



```
acov =

    1.0e-03 *

    0.1937    -0.0000
   -0.0000    0.0968
```

Estimate the standard errors of estimates.

```
se = sqrt(diag(acov))

se =

    0.0139
    0.0098
```

The standard error of the estimates of mu and sigma are 0.0139 and 0.0098, respectively.

### Custom Log Probability Density Function

In the MATLAB Editor, create a function that returns the log probability density function of a beta distribution. Save the file in your current working folder as `betalogpdf.m`.

```
function logpdf = betalogpdf(x,a,b)
logpdf = (a-1)*log(x)+(b-1)*log(1-x)-betaln(a,b);
```

Generate sample data from a beta distribution with parameters 1.23 and 3.45 and estimate the parameters using the simulated data.

```
rng('default')
x = betarnd(1.23,3.45,25,1);
phat = mle(x,'dist','beta')

phat =

    1.1213    2.7182
```

Compute the approximate covariance matrix of the parameter estimates.

```
acov = mlecov(phat,x,'logpdf',@betalogpdf)

acov =
```

```
0.0810    0.1646
0.1646    0.6074
```

### Custom Log pdf and Survival Function

Load the sample data.

```
load(fullfile(matlabroot, 'examples', 'stats', 'readmissiontimes.mat'));
```

The sample data includes `ReadmissionTime`, which has readmission times for 100 patients. The column vector `Censored` has the censorship information for each patient, where 1 indicates a censored observation, and 0 indicates the exact readmission time is observed. This is simulated data.

Define a custom log probability density and survival function.

```
custlogpdf = @(data, lambda, k) log(k) - k*log(lambda) ...
             + (k-1)*log(data) - (data/lambda).^k;
custlogsf = @(data, lambda, k) -(data/lambda).^k;
```

Estimate the parameters, `lambda` and `k`, of the custom distribution for the censored sample data.

```
phat = mle(ReadmissionTime, 'logpdf', custlogpdf, ...
           'logsf', custlogsf, 'start', [1, 0.75], 'Censoring', Censored)
```

```
phat =
```

```
9.2090    1.4223
```

The scale and shape parameters of the custom-defined distribution are 9.2090 and 1.4223, respectively.

Compute the approximate covariance matrix of the parameter estimates.

```
acov = mlecov(phat, ReadmissionTime, ...
              'logpdf', custlogpdf, 'logsf', custlogsf, 'Censoring', Censored)
```

```
acov =
```

```
0.5653    0.0102
0.0102    0.0163
```

### Custom Log Negative Likelihood Function

Load the sample data.

```
load(fullfile(matlabroot, 'examples', 'stats', 'readmissiontimes.mat'));
```

The sample data includes `ReadmissionTime`, which has readmission times for 100 patients. This is simulated data.

Define a negative log likelihood function.

```
custnloglf = @(lambda,data,cens,freq) -length(data)*log(lambda)...
+ nansum(lambda*data);
```

Estimate the parameters of the defined distribution.

```
phat = mle(ReadmissionTime, 'nloglf', custnloglf, 'start', 0.05)
```

```
phat =
```

```
0.1462
```

Compute the variance of the parameter estimate.

```
acov = mlecov(phat, ReadmissionTime, 'nloglf', custnloglf)
```

```
acov =
```

```
2.1374e-04
```

Compute the standard error.

```
sqrt(acov)
```

```
ans =
```

0.0146

## Input Arguments

### **params** — Parameter estimates

scalar value | vector

Parameter estimates, specified as a scalar value or vector of scalar values. These parameter estimates must be maximum likelihood estimates. For example, you can specify parameter estimates returned by `mle`.

Data Types: `single` | `double`

### **data** — Sample data

vector

Sample data `mle` uses to estimate the distribution parameters, specified as a vector.

Data Types: `single` | `double`

### **pdf** — Custom probability density function

function handle

Custom probability distribution function, specified as a function handle created using `@`.

This custom function accepts the vector `data` and one or more individual distribution parameters as input parameters, and returns a vector of probability density values.

For example, if the name of the custom probability density function is `newpdf`, then you can specify the function handle in `mlecov` as follows.

Example: `@newpdf`

Data Types: `function_handle`

### **cdf** — Custom cumulative distribution function

function handle

Custom cumulative distribution function, specified as a function handle created using `@`.

This custom function accepts the vector `data` and one or more individual distribution parameters as input parameters, and returns a vector of cumulative probability values.

You must define `cdf` with `pdf` if data is censored and you use the `'censoring'` name-value pair argument. If `'censoring'` is not present, you do not have to specify `cdf` while using `pdf`.

For example, if the name of the custom cumulative distribution function is `newcdf`, then you can specify the function handle in `mlecov` as follows.

Example: `@newcdf`

Data Types: `function_handle`

### **logpdf** — Custom log probability density function

`function handle`

Custom log probability density function, specified as a function handle created using `@`.

This custom function accepts the vector `data` and one or more individual distribution parameters as input parameters, and returns a vector of log probability values.

For example, if the name of the custom log probability density function is `customlogpdf`, then you can specify the function handle in `mlecov` as follows.

Example: `@customlogpdf`

Data Types: `function_handle`

### **logsf** — Custom log survival function

`function handle`

Custom log survival function on page 32-4042, specified as a function handle created using `@`.

This custom function accepts the vector `data` and one or more individual distribution parameters as input parameters, and returns a vector of log survival probability values.

You must define `logsf` with `logpdf` if data is censored and you use the `'censoring'` name-value pair argument. If `'censoring'` is not present, you do not have to specify `logsf` while using `logpdf`.

For example, if the name of the custom log survival function is `logsurvival`, then you can specify the function handle in `mlecov` as follows.

Example: `@logsurvival`

Data Types: `function_handle`

### **nloglf** — Custom negative loglikelihood function

function handle

Custom negative loglikelihood function, specified as a function handle created using `@`.

This custom function accepts the following input arguments.

<code>params</code>	Vector of distribution parameter values
<code>data</code>	Vector of data
<code>cens</code>	Boolean vector of censored values
<code>freq</code>	Vector of integer data frequencies

`nloglf` must accept all four arguments even if you do not use the `'censoring'` or `'frequency'` name-value pair arguments. You can write `'nloglf'` to ignore `cens` and `freq` arguments in that case.

`nloglf` returns a scalar negative loglikelihood value and optionally, a negative loglikelihood gradient vector (see the `'GradObj'` field in `'options'`).

If the name of the custom negative log likelihood function is `negloglik`, then you can specify the function handle in `mlecov` as follows.

Example: `@negloglik`

Data Types: `function_handle`

## **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'censoring', cens, 'options', opt` specifies that `mlecov` reads the censored data information from the vector `cens` and performs according to the new options structure `opt`.

#### **censoring** — Indicator for censoring

array of 0s (default) | array of 0s and 1s

Indicator for censoring, specified as the comma-separated pair consisting of `'censoring'` and a Boolean array of the same size as `data`. Use 1 for observations that are right censored and 0 for observations that are fully observed. The default is all observations are fully observed.

For censored data, you must use `cdf` with `pdf`, or `logsf` with `logpdf`, or `nloglf` must be defined to account for censoring.

For example, if the censored data information is in the binary array called `Censored`, then you can specify the censored data as follows.

Example: `'censoring', Censored`

Data Types: `logical`

#### **frequency** — Frequency of observations

array of 1s (default) | vector of nonnegative integer counts

Frequency of observations, specified as the comma-separated pair consisting of `'frequency'` and an array containing nonnegative integer counts, which is the same size as `data`. The default is one observation per element of `data`.

For example, if the observation frequencies are stored in an array named `Freq`, you can specify the frequencies as follows.

Example: `'frequency', Freq`

Data Types: `single` | `double`

#### **options** — Numerical options

structure

Numerical options for the finite difference Hessian calculation, specified as the comma-separated pair consisting of `'options'` and a structure returned by `statset`.

You can set the options under a new name and use it in the name-value pair argument. The applicable `statset` parameters are as follows.

Parameter	Value
'GradObj'	Default is 'off'.  'on' or 'off', indicating whether or not the function provided with the <code>nloglf</code> input argument can return the gradient vector of the negative log-likelihood as a second output.
'DerivStep'	Default is $\text{eps}^{(1/4)}$ .  Relative step size used in finite difference for Hessian calculations. It can be a scalar, or the same size as <code>params</code> . A smaller value than the default might be appropriate if 'GradObj' is 'on'.

Example: `'options', statset('mlecov')`

Data Types: `struct`

## Output Arguments

**acov** — Approximation to asymptotic covariance matrix

*p*-by-*p* matrix

Approximation to asymptotic covariance matrix, returned as a *p*-by-*p* matrix, where *p* is the number of parameters in `params`.

## Definitions

### Survival Function

The survival function is the probability of survival as a function of time. It is also called the survivor function. It gives the probability that the survival time of an individual exceeds a certain value. Since the cumulative distribution function,  $F(t)$ , is the probability that the survival time is less than or equal to a given point in time, the survival function for a continuous distribution,  $S(t)$ , is the complement of the cumulative distribution function:  $S(t) = 1 - F(t)$ .



## See Also

mle

## Topics

“What Is Survival Analysis?” on page 14-2

Introduced before R2006a

## mnpdf

Multinomial probability density function

### Syntax

```
Y = mnpdf(X,PROB)
```

### Description

`Y = mnpdf(X,PROB)` returns the pdf for the multinomial distribution with probabilities `PROB`, evaluated at each row of `X`. `X` and `PROB` are  $m$ -by- $k$  matrices or 1-by- $k$  vectors, where  $k$  is the number of multinomial bins or categories. Each row of `PROB` must sum to one, and the sample sizes for each observation (rows of `X`) are given by the row sums `sum(X,2)`. `Y` is an  $m$ -by-1 vector, and `mnpdf` computes each row of `Y` using the corresponding rows of the inputs, or replicates them if needed.

### Examples

#### Compute the Multinomial Distribution pdf

Compute the pdf of a multinomial distribution with a sample size of  $n = 10$ . The probabilities are  $p = 1/2$  for outcome 1,  $p = 1/3$  for outcome 2, and  $p = 1/6$  for outcome 3.

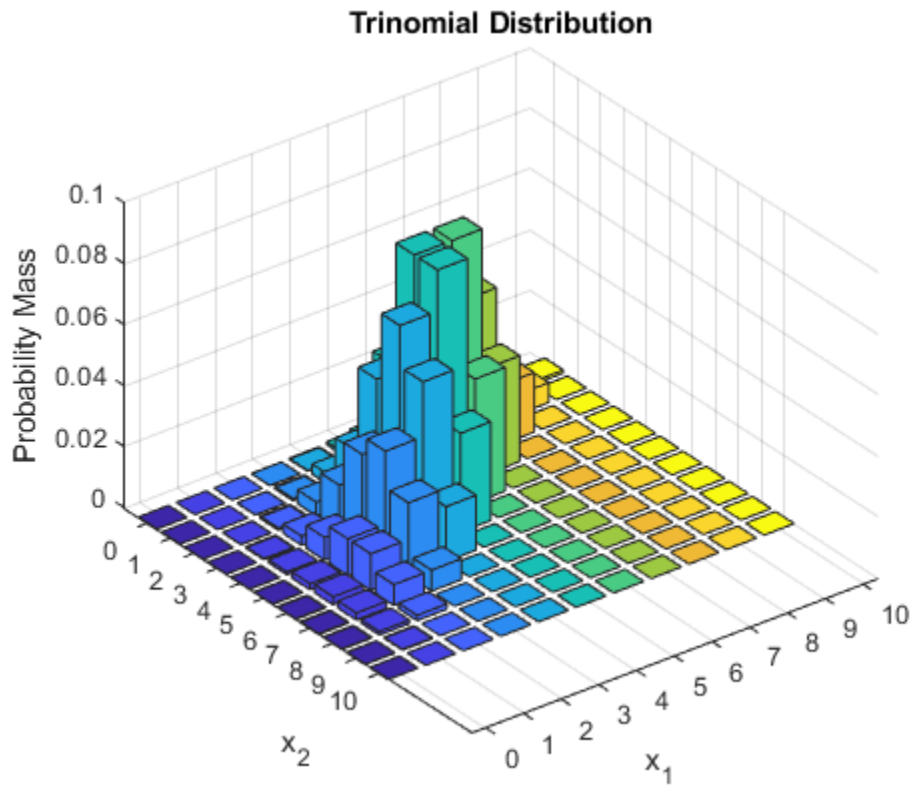
```
p = [1/2 1/3 1/6];  
n = 10;  
x1 = 0:n;  
x2 = 0:n;  
[X1,X2] = meshgrid(x1,x2);  
X3 = n-(X1+X2);
```

Compute the pdf of the distribution.

```
Y = mnpdf([X1(:),X2(:),X3(:)], repmat(p,(n+1)^2,1));
```

Plot the pdf on a 3-dimensional figure.

```
Y = reshape(Y,n+1,n+1);  
bar3(Y)  
h = gca;  
h.XTickLabel = [0:n];  
h.YTickLabel = [0:n];  
xlabel('x_1')  
ylabel('x_2')  
zlabel('Probability Mass')  
title('Trinomial Distribution')
```



Note that the visualization does not show  $x_3$ , which is determined by the constraint  $x_1 + x_2 + x_3 = n$ .

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`mnrnd`

### Topics

“Multinomial Distribution” on page B-103

Introduced in R2006b

# mnrfit

Multinomial logistic regression

## Syntax

```
B = mnrfi(X,Y)
B = mnrfi(X,Y,Name,Value)
[B,dev,stats] = mnrfi(____)
```

## Description

`B = mnrfi(X,Y)` returns a matrix, `B`, of coefficient estimates for a multinomial logistic regression of the nominal responses in `Y` on the predictors in `X`.

`B = mnrfi(X,Y,Name,Value)` returns a matrix, `B`, of coefficient estimates for a multinomial model fit with additional options specified by one or more `Name, Value` pair arguments.

For example, you can fit a nominal, an ordinal, or a hierarchical model, or change the link function.

`[B,dev,stats] = mnrfi(____)` also returns the deviance of the fit, `dev`, and the structure `stats` for any of the previous input arguments. `stats` contains model statistics such as degrees of freedom, standard errors for coefficient estimates, and residuals.

## Examples

### Multinomial Regression for Nominal Responses

Fit a multinomial regression for nominal outcomes and interpret the results.

Load the sample data.

```
load fisheriris
```

The column vector, `species`, consists of iris flowers of three different species, `setosa`, `versicolor`, `virginica`. The double matrix `meas` consists of four types of measurements on the flowers, the length and width of sepals and petals in centimeters, respectively.

Define the nominal response variable using a categorical array.

```
sp = categorical(species);
```

Fit a multinomial regression model to predict the species using the measurements.

```
[B,dev,stats] = mnrfits(meas,sp);  
B
```

```
B =
```

```
1.0e+03 *  
  
1.9345    0.0426  
0.6460    0.0025  
-0.5449   0.0067  
-0.4948  -0.0094  
-2.6472  -0.0183
```

This is a nominal model for the response category relative risks, with separate slopes on all four predictors, that is, each category of `meas`. The first row of `B` contains the intercept terms for the relative risk of the first two response categories, `setosa` and `versicolor` versus the reference category, `virginica`. The last four rows contain the slopes for the models for the first two categories. `mnrfits` accepts the third category as the reference category.

The relative risk of an iris flower being species 2 (`versicolor`) versus species 3 (`virginica`) is the ratio of the two probabilities (the probability of being species 2 and the probability of being species 3). The model for the relative risk is

$$\ln \left( \frac{\pi_{\text{versicolor}}}{\pi_{\text{virginica}}} \right) = 42.6 + 2.5X_1 + 6.7X_2 - 9.4X_3 - 18.3X_4.$$

The coefficients express both the effects of the predictor variables on the relative risk and the log odds of being in one category versus the reference category. For example, the estimated coefficient 2.5 indicates that the relative risk of being species 2 (`versicolor`)

versus species 3 (virginica) increases  $\exp(2.5)$  times for each unit increase in  $X_1$ , the first measurement, given all else is equal. The relative log odds of being versicolor versus virginica increases 2.5 times with a one-unit increase in  $X_1$ , given all else is equal.

If the coefficients are converging toward infinity or negative infinity, the estimated coefficients can vary slightly depending on your operating system.

Check the statistical significance of the model coefficients.

```
stats.p
```

```
ans =
```

```

0      0.0000
0      0.0281
0      0.0000
0      0.0000
0      0.0000
```

The small  $P$ -values indicate that all measures are significant on the relative risk of being a setosa versus a virginica (species 1 compared to species 3) and being a versicolor versus a virginica (species 2 compared to species 3).

Request the standard errors of coefficient estimates.

```
stats.se
```

```
ans =
```

```

12.4038    5.2719
 3.5783    1.1228
 3.1760    1.4789
 3.5403    1.2934
 7.1203    2.0967
```

Calculate the 95% confidence limits for the coefficients.

```
LL = stats.beta - 1.96.*stats.se;
UL = stats.beta + 1.96.*stats.se;
```

Display the confidence intervals for the coefficients of the model for the relative risk of being a setosa versus a virginica (the first column of coefficients in  $\mathbb{B}$ ).

```
[LL(:,1) UL(:,1)]
```

```
ans =
```

```
1.0e+03 *  
 1.9101    1.9588  
 0.6390    0.6530  
-0.5511   -0.5387  
-0.5017   -0.4879  
-2.6611   -2.6332
```

Find the confidence intervals for the coefficients of the model for the relative risk of being a versicolor versus a virginica (the second column of coefficients in  $\mathbb{B}$ ).

```
[LL(:,2) UL(:,2)]
```

```
ans =
```

```
32.3049    52.9707  
 0.2645     4.6660  
 3.7823     9.5795  
-11.9644   -6.8944  
-22.3957  -14.1766
```

### Multinomial Regression for Ordinal Responses

Fit a multinomial regression model for categorical responses with natural ordering among categories.

Load the sample data and define the predictor variables.

```
load carbig  
X = [Acceleration Displacement Horsepower Weight];
```



The predictor variables are the acceleration, engine displacement, horsepower, and weight of the cars. The response variable is miles per gallon (mpg).

Create an ordinal response variable categorizing MPG into four levels from 9 to 48 mpg by labeling the response values in the range 9-19 as 1, 20-29 as 2, 30-39 as 3, and 40-48 as 4.

```
miles = ordinal(MPG, {'1', '2', '3', '4'}, [], [9, 19, 29, 39, 48]);
```

Fit an ordinal response model for the response variable miles.

```
[B, dev, stats] = mnrfits(X, miles, 'model', 'ordinal');
B
```

```
B =
-16.6895
-11.7208
-8.0606
 0.1048
 0.0103
 0.0645
 0.0017
```

The first three elements of `B` are the intercept terms for the models, and the last four elements of `B` are the coefficients of the covariates, assumed common across all categories. This model corresponds to *parallel regression*, which is also called the *proportional odds* model, where there is a different intercept but common slopes among categories. You can specify this using the `'interactions', 'off'` name-value pair argument, which is the default for ordinal models.

```
[B(1:3)'; repmat(B(4:end), 1, 3)]
```

```
ans =
-16.6895 -11.7208 -8.0606
 0.1048  0.1048  0.1048
 0.0103  0.0103  0.0103
 0.0645  0.0645  0.0645
 0.0017  0.0017  0.0017
```

The link function in the model is `logit('link', 'logit')`, which is the default for an ordinal model. The coefficients express the relative risk or log odds of the mpg of a car being less than or equal to one value versus greater than that value.

The proportional odds model in this example is

$$\ln \left( \frac{P(\text{mpg} \leq 19)}{P(\text{mpg} > 19)} \right) = -16.6895 + 0.1048X_A + 0.0103X_D + 0.0645X_H + 0.0017X_W$$

$$\ln \left( \frac{P(\text{mpg} \leq 29)}{P(\text{mpg} > 29)} \right) = -11.7208 + 0.1048X_A + 0.0103X_D + 0.0645X_H + 0.0017X_W$$

$$\ln \left( \frac{P(\text{mpg} \leq 39)}{P(\text{mpg} > 39)} \right) = -8.0606 + 0.1048X_A + 0.0103X_D + 0.0645X_H + 0.0017X_W$$

For example, the coefficient estimate of 0.1048 indicates that a unit change in acceleration would impact the odds of the mpg of a car being less than or equal to 19 versus more than 19, or being less than or equal to 29 versus greater than 29, or being less than or equal to 39 versus greater than 39, by a factor of  $\exp(0.1048)$  given all else is equal.

Assess the significance of the coefficients.

```
stats.p
```

```
ans =
```

```
0.0000
0.0000
0.0000
0.1899
0.0350
0.0000
0.0118
```

The *P*-values of 0.035, 0.0000, and 0.0118 for engine displacement, horsepower, and weight of a car, respectively, indicate that these factors are significant on the odds of mpg of a car being less than or equal to a certain value versus being greater than that value.

## Hierarchical Multinomial Regression Model

Fit a hierarchical multinomial regression model.

Load the sample data.

```
load(fullfile(matlabroot, 'examples', 'stats', 'smoking.mat'));
```

The data set `smoking` contains five variables: sex, age, weight, and systolic and diastolic blood pressure. Sex is a binary variable where 1 indicates female patients, and 0 indicates male patients.

Define the response variable.

```
Y = categorical(smoking.Smoker);
```

The data in `Smoker` has four categories:

- 0: Nonsmoker, 0 cigarettes a day
- 1: Smoker, 1–5 cigarettes a day
- 2: Smoker, 6–10 cigarettes a day
- 3: Smoker, 11 or more cigarettes a day

Define the predictor variables.

```
X = [smoking.Sex smoking.Age smoking.Weight...
     smoking.SystolicBP smoking.DiastolicBP];
```

Fit a hierarchical multinomial model.

```
[B, dev, stats] = mnrfit(X, Y, 'model', 'hierarchical');
B
```

B =

```
43.8148    5.9571   44.0712
 1.8709   -0.0230    0.0662
 0.0188    0.0625    0.1335
 0.0046   -0.0072   -0.0130
-0.2170    0.0416   -0.0324
-0.2273   -0.1449   -0.4824
```

The first column of  $B$  includes the intercept and the coefficient estimates for the model of the relative risk of being a nonsmoker versus a smoker. The second column includes the parameter estimates for modeling the log odds of smoking 1–5 cigarettes a day versus more than five cigarettes a day given that a person is a smoker. Finally, the third column includes the parameter estimates for modeling the log odds of a person smoking 6–10 cigarettes a day versus more than 10 cigarettes a day given he/she smokes more than 5 cigarettes a day.

The coefficients differ across categories. You can specify this using the 'interactions', 'on' name-value pair argument, which is the default for hierarchical models. So, the model in this example is

$$\ln\left(\frac{P(y=0)}{P(y>0)}\right) = 43.8148 + 1.8709X_S + 0.0188X_A + 0.0046X_W - 0.2170X_{SBP} - 0.2273X_{DBP}$$

$$\ln\left(\frac{P(1 \leq y \leq 5)}{P(y > 5)}\right) = 5.9571 - 0.0230X_S + 0.0625X_A - 0.0072X_W + 0.0416X_{SBP} - 0.1449X_{DBP}$$

$$\ln\left(\frac{P(6 \leq y \leq 10)}{P(y > 10)}\right) = 44.0712 + 0.0662X_S + 0.1335X_A - 0.0130X_W - 0.0324X_{SBP} - 0.4824X_{DBP}$$

For example, the coefficient estimate of 1.8709 indicates that the likelihood of being a smoker versus a nonsmoker increases by  $\exp(1.8709) = 6.49$  times as the gender changes from female to male given everything else held constant.

Assess the statistical significance of the terms.

```
stats.p
```

```
ans =
```

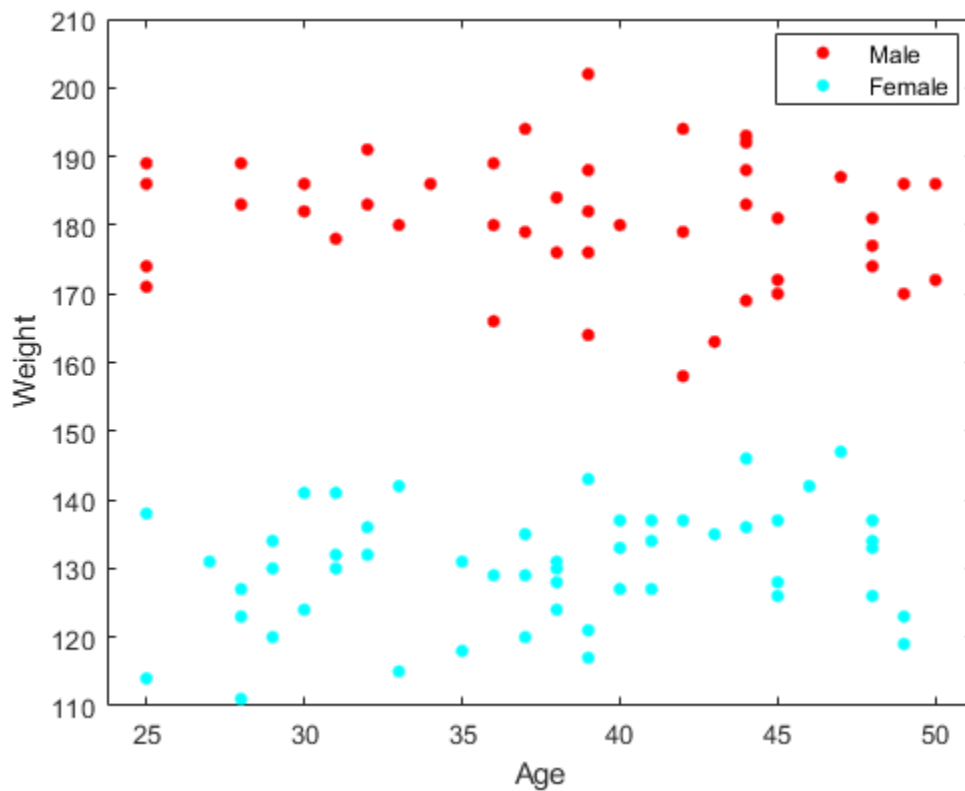
0.0000	0.5363	0.2149
0.3549	0.9912	0.9835
0.6850	0.2676	0.2313
0.9032	0.8523	0.8514
0.0009	0.5187	0.8165
0.0004	0.0483	0.0545

Sex, age, or weight don't appear significant on any level. The  $P$ -values of 0.0009 and 0.0004 indicate that both types of blood pressure are significant on the relative risk of a person being a smoker versus a nonsmoker. The  $P$ -value of 0.0483 shows that only diastolic blood pressure is significant on the odds of a person smoking 0–5 cigarettes a

day versus more than 5 cigarettes a day. Similarly, the  $P$ -value of 0.0545 indicates that diastolic blood pressure is significant on the odds of a person smoking 6–10 cigarettes a day versus more than 10 cigarettes a day.

Check if any nonsignificant factors are correlated to each other. Draw a scatterplot of age versus weight grouped by sex.

```
figure()  
gscatter(smoking.Age, smoking.Weight, smoking.Sex)  
legend('Male', 'Female')  
xlabel('Age')  
ylabel('Weight')
```



The range of weight of an individual seems to differ according to gender. Age does not seem to have any obvious correlation with sex or weight. Age is insignificant and weight seems to be correlated with sex, so you can eliminate both and reconstruct the model.

Eliminate age and weight from the model and fit a hierarchical model with sex, systolic blood pressure, and diastolic blood pressure as the predictor variables.

```
X = double([smoking.Sex smoking.SystolicBP...
smoking.DiastolicBP]);
[B,dev,stats] = mnrfits(X,Y,'model','hierarchical');
B
```

B =

44.8456	5.3230	25.0248
1.6045	0.2330	0.4982
-0.2161	0.0497	0.0179
-0.2222	-0.1358	-0.3092

Here, a coefficient estimate of 1.6045 indicates that the likelihood of being a nonsmoker versus a smoker increases by  $\exp(1.6045) = 4.97$  times as sex changes from male to female. A unit increase in the systolic blood pressure indicates an  $\exp(-.2161) = 0.8056$  decrease in the likelihood of being a nonsmoker versus a smoker. Similarly, a unit increase in the diastolic blood pressure indicates an  $\exp(-.2222) = 0.8007$  decrease in the relative rate of being a nonsmoker versus being a smoker.

Assess the statistical significance of the terms.

```
stats.p
```

ans =

0.0000	0.4715	0.2325
0.0210	0.7488	0.6362
0.0010	0.4107	0.8899
0.0003	0.0483	0.0718

The *P*-values of 0.0210, 0.0010, and 0.0003 indicate that the terms sex and both types of blood pressure are significant on the relative risk of a person being a nonsmoker versus a smoker, given the other terms in the model. Based on the *P*-value of 0.0483, diastolic

blood pressure appears significant on the relative risk of a person smoking 1–5 cigarettes versus more than 5 cigarettes a day, given that this person is a smoker. Because none of the  $P$ -values on the third column are less than 0.05, you can say that none of the variables are statistically significant on the relative risk of a person smoking from 6–10 cigarettes versus more than 10 cigarettes, given that this person smokes more than 5 cigarettes a day.

## Input Arguments

### **x** — Observations on predictor variables

*n*-by-*p* matrix

Observations on predictor variables, specified as an *n*-by-*p* matrix. **x** contains *n* observations for *p* predictors.

---

**Note** `mnrfit` automatically includes a constant term (intercept) in all models. Do not include a column of 1s in **x**.

---

Data Types: `single` | `double`

### **y** — Response values

*n*-by-*k* matrix | *n*-by-1 column vector

Response values, specified as a column vector or a matrix. **y** can be one of the following:

- An *n*-by-*k* matrix, where  $Y(i,j)$  is the number of outcomes of the multinomial category *j* for the predictor combinations given by  $X(i,:)$ . In this case, the number of observations are made at each predictor combination.
- An *n*-by-1 column vector of scalar integers from 1 to *k* indicating the value of the response for each observation. In this case, all sample sizes are 1.
- An *n*-by-1 categorical array indicating the nominal or ordinal value of the response for each observation. In this case, all sample sizes are 1.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single

quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'Model', 'ordinal', 'Link', 'probit'` specifies an ordinal model with a probit link function.

### **Model** — Type of model to fit

`'nominal' (default) | 'ordinal' | 'hierarchical'`

Type of model to fit, specified as the comma-separated pair consisting of `'Model'` and one of the following.

<code>'nominal'</code>	Default. There is no ordering among the response categories.
<code>'ordinal'</code>	There is a natural ordering among the response categories.
<code>'hierarchical'</code>	The choice of response category is sequential/nested.

Example: `'Model', 'ordinal'`

### **Interactions** — Indicator for interaction between multinomial categories and coefficients

`'on' | 'off'`

Indicator for an interaction between the multinomial categories and coefficients, specified as the comma-separated pair consisting of `'Interactions'` and one of the following.

<code>'on'</code>	Default for nominal and hierarchical models. Fit a model with different coefficients across categories.
<code>'off'</code>	Default for ordinal models. Fit a model with a common set of coefficients for the predictor variables, across all multinomial categories. This is often described as <i>parallel regression</i> or the <i>proportional odds model</i> .

In all cases, the model has different intercepts across categories. The choice of `'Interactions'` determines the dimensions of the output array B.

Example: `'Interactions', 'off'`

Data Types: logical

### **Link** — Link function

`'logit' (default) | 'probit' | 'comloglog' | 'loglog'`

Link function to use for ordinal and hierarchical models, specified as the comma-separated pair consisting of `'Link'` and one of the following.



'logit'	Default. $f(\gamma) = \ln(\gamma/(1-\gamma))$
'probit'	$f(\gamma) = \Phi^{-1}(\gamma)$ — error term is normally distributed with variance 1
'comloglog'	Complementary log-log $f(\gamma) = \ln(-\ln(1-\gamma))$
'loglog'	$f(\gamma) = \ln(-\ln(\gamma))$

The link function defines the relationship between response probabilities and the linear combination of predictors,  $X\beta$ . The link functions might be functions of cumulative or conditional probabilities based on whether the model is for an ordinal or a sequential/nested response. For example, for an ordinal model,  $\gamma$  represents the cumulative probability of being in categories 1 to  $j$  and the model with a logit link function as follows:

$$\ln\left(\frac{\gamma}{1-\gamma}\right) = \ln\left(\frac{\pi_1 + \pi_2 + \dots + \pi_j}{\pi_{j+1} + \dots + \pi_k}\right) = \beta_{0j} + \beta_1 X_1 + \beta_2 X_2 + \dots + \beta_p X_p,$$

where  $k$  represents the last category.

You cannot specify the 'Link' parameter for nominal models; these always use a multinomial logit link,

$$\ln\left(\frac{\pi_j}{\pi_r}\right) = \beta_{j0} + \beta_{j1} X_{j1} + \beta_{j2} X_{j2} + \dots + \beta_{jp} X_{jp}, \quad j = 1, \dots, k-1,$$

where  $\pi$  stands for a categorical probability, and  $r$  corresponds to the reference category. mnrfit uses the last category as the reference category for nominal models.

Example: 'Link', 'loglog'

#### **EstDisp — Indicator for estimating dispersion parameter**

'off' (default) | 'on'

Indicator for estimating a dispersion parameter, specified as the comma-separated pair consisting of 'EstDisp' and one of the following.

'off'	Default. Use the theoretical dispersion value of 1.
'on'	Estimate a dispersion parameter for the multinomial distribution in computing standard errors.

Example: 'EstDisp', 'on'

## Output Arguments

### **B** — Coefficient estimates

vector | matrix

Coefficient estimates for a multinomial logistic regression of the responses in  $Y$ , returned as a vector or a matrix.

- If 'Interaction' is 'off', then  $B$  is a  $k - 1 + p$  vector. The first  $k - 1$  rows of  $B$  correspond to the intercept terms, one for each  $k - 1$  multinomial categories, and the remaining  $p$  rows correspond to the predictor coefficients, which are common for all of the first  $k - 1$  categories.
- If 'Interaction' is 'on', then  $B$  is a  $(p + 1)$ -by- $(k - 1)$  matrix. Each column of  $B$  corresponds to the estimated intercept term and predictor coefficients, one for each of the first  $k - 1$  multinomial categories.

The estimates for the  $k$ th category are taken to be zero as `mnrfit` takes the last category as the reference category.

### **dev** — Deviance of the fit

scalar value

Deviance of the fit, returned as a scalar value. It is twice the difference between the maximum achievable log likelihood and that attained under the fitted model. This corresponds to the sum of deviance residuals,

$$dev = 2 * \sum_i^n \sum_j^k y_{ij} * \log \left( \frac{y_{ij}}{\pi_{ij} * m_i} \right) = \sum_i^n rd_i,$$

where  $rd_i$  are the deviance residuals. For deviance residuals see `stats`.

### **stats** — Model statistics

structure

Model statistics, returned as a structure that contains the following fields.

beta	The coefficient estimates. These are the same as $B$ .
------	--

dfe	Degrees of freedom for error <ul style="list-style-type: none"> <li>• If 'Interactions' is 'off', then degrees of freedom is <math>n*(k - 1) - (k - 1 + p)</math>.</li> <li>• If 'Interactions' is 'on', then degrees of freedom is <math>(n - p + 1)*(k - 1)</math>.</li> </ul>
sfit	Estimated dispersion parameter.
s	Theoretical or estimated dispersion parameter. <ul style="list-style-type: none"> <li>• If 'Estdisp' is 'off', then s is the theoretical dispersion parameter, 1.</li> <li>• If 'Estdisp' is 'on', then s is equal to the estimated dispersion parameter, sfit.</li> </ul>
estdisp	Indicator for a theoretical or estimated dispersion parameter.
se	Standard errors of coefficient estimates, B.
coeffcorr	Estimated correlation matrix for B.
covb	Estimated covariance matrix for B.
t	<i>t</i> statistics for B.
p	<i>p</i> -values for B.
resid	Raw residuals. Observed minus fitted values, $r_{ij} = y_{ij} - \hat{\pi}_{ij} * m_i, \quad \begin{cases} i = 1, \dots, n \\ j = 1, \dots, k \end{cases}$ where $\pi_{ij}$ is the categorical, cumulative or conditional probability, and $m_i$ is the corresponding sample size.
residp	Pearson residuals, which are the raw residuals scaled by the estimated standard deviation: $rp_{ij} = \frac{r_{ij}}{\hat{\sigma}_{ij}} = \frac{y_{ij} - \hat{\pi}_{ij} * m_i}{\sqrt{\hat{\pi}_{ij} * (1 - \hat{\pi}_{ij}) * m_i}}, \quad \begin{cases} i = 1, \dots, n \\ j = 1, \dots, k \end{cases}$ where $\pi_{ij}$ is the categorical, cumulative, or conditional probability, and $m_i$ is the corresponding sample size.

residd	<p>Deviance residuals:</p> $rd_i = 2 * \sum_j^k y_{ij} * \log \left( \frac{y_{ij}}{\pi_{ij} * m_i} \right) \quad i = 1, \dots, n.$ <p>where <math>\pi_{ij}</math> is the categorical, cumulative, or conditional probability, and <math>m_i</math> is the corresponding sample size.</p>
--------	--

## Algorithms

`mnrfit` treats NaNs in either X or Y as missing values, and ignores them.

## References

- [1] McCullagh, P., and J. A. Nelder. *Generalized Linear Models*. New York: Chapman & Hall, 1990.
- [2] Long, J. S. *Regression Models for Categorical and Limited Dependent Variables*. Sage Publications, 1997.
- [3] Dobson, A. J., and A. G. Barnett. *An Introduction to Generalized Linear Models*. Chapman and Hall/CRC. Taylor & Francis Group, 2008.

## See Also

`fitglm` | `glmfit` | `glmval` | `mnrval`

## Topics

- “Multinomial Distribution” on page B-103
- “Multinomial Models for Nominal Responses” on page 12-2
- “Multinomial Models for Ordinal Responses” on page 12-5
- “Hierarchical Multinomial Models” on page 12-9

Introduced in R2006b

# mnrnd

Multinomial random numbers

## Syntax

```
r = mnrnd(n,p)
R = mnrnd(n,p,m)
R = mnrnd(N,P)
```

## Description

`r = mnrnd(n,p)` returns random values `r` from the multinomial distribution with parameters `n` and `p`. `n` is a positive integer specifying the number of trials (sample size) for each multinomial outcome. `p` is a 1-by- $k$  vector of multinomial probabilities, where  $k$  is the number of multinomial bins or categories. `p` must sum to one. (If `p` does not sum to one, `r` consists entirely of NaN values.) `r` is a 1-by- $k$  vector, containing counts for each of the  $k$  multinomial bins.

`R = mnrnd(n,p,m)` returns `m` random vectors from the multinomial distribution with parameters `n` and `p`. `R` is a  $m$ -by- $k$  matrix, where  $k$  is the number of multinomial bins or categories. Each row of `R` corresponds to one multinomial outcome.

`R = mnrnd(N,P)` generates outcomes from different multinomial distributions. `P` is a  $m$ -by- $k$  matrix, where  $k$  is the number of multinomial bins or categories and each of the  $m$  rows contains a different set of multinomial probabilities. Each row of `P` must sum to one. (If any row of `P` does not sum to one, the corresponding row of `R` consists entirely of NaN values.) `N` is a  $m$ -by-1 vector of positive integers or a single positive integer (replicated by `mnrnd` to a  $m$ -by-1 vector). `R` is a  $m$ -by- $k$  matrix. Each row of `R` is generated using the corresponding rows of `N` and `P`.

## Examples

Generate 2 random vectors with the same probabilities:

```
n = 1e3;
p = [0.2, 0.3, 0.5];
R = mnrnd(n, p, 2)
R =
    215    282    503
    194    303    503
```

Generate 2 random vectors with different probabilities:

```
n = 1e3;
P = [0.2, 0.3, 0.5; ...
     0.3, 0.4, 0.3];
R = mnrnd(n, P)
R =
    186    290    524
    290    389    321
```

## See Also

`mnpdf`

## Topics

“Multinomial Distribution” on page B-103

**Introduced in R2006b**

# mnrval

Multinomial logistic regression values

## Syntax

```

pihat = mnrvl(B,X)
[pihat,dlow,dhi] = mnrvl(B,X,stats)
[pihat,dlow,dhi] = mnrvl(B,X,stats,Name,Value)

yhat = mnrvl(B,X,ssize)
[yhat,dlow,dhi] = mnrvl(B,X,ssize,stats)
[yhat,dlow,dhi] = mnrvl(B,X,ssize,stats,Name,Value)

```

## Description

`pihat = mnrvl(B,X)` returns the predicted probabilities for the multinomial logistic regression model with predictors,  $X$ , and the coefficient estimates,  $B$ .

`pihat` is an  $n$ -by- $k$  matrix of predicted probabilities for each multinomial category.  $B$  is the vector or matrix that contains the coefficient estimates returned by `mnrfit`. And  $X$  is an  $n$ -by- $p$  matrix which contains  $n$  observations for  $p$  predictors.

---

**Note** `mnrval` automatically includes a constant term in all models. Do not enter a column of 1s in  $X$ .

---

`[pihat,dlow,dhi] = mnrvl(B,X,stats)` also returns 95% error bounds on the predicted probabilities, `pihat`, using the statistics in the structure, `stats`, returned by `mnrfit`.

The lower and upper confidence bounds for `pihat` are `pihat` minus `dlow` and `pihat` plus `dhi`, respectively. Confidence bounds are nonsimultaneous and only apply to the fitted curve, not to new observations.

`[pihat, dlow, dhi] = mnrval(B, X, stats, Name, Value)` returns the predicted probabilities and 95% error bounds on the predicted probabilities `pihat`, with additional options specified by one or more `Name, Value` pair arguments.

For example, you can specify the model type, link function, and the type of probabilities to return.

`yhat = mnrval(B, X, ssize)` returns the predicted category counts for sample sizes, `ssize`.

`[yhat, dlow, dhi] = mnrval(B, X, ssize, stats)` also computes 95% error bounds on the predicted counts `yhat`, using the statistics in the structure, `stats`, returned by `mnrfit`.

The lower and upper confidence bounds for `yhat` are `yhat` minus `dlo` and `yhat` plus `dhi`, respectively. Confidence bounds are nonsimultaneous and they apply to the fitted curve, not to new observations.

`[yhat, dlow, dhi] = mnrval(B, X, ssize, stats, Name, Value)` returns the predicted category counts and 95% error bounds on the predicted counts `yhat`, with additional options specified by one or more `Name, Value` pair arguments.

For example, you can specify the model type, link function, and the type of predicted counts to return.

## Examples

### Estimate Category Probabilities for Nominal Responses

Fit a multinomial regression for nominal outcomes and estimate the category probabilities.

Load the sample data.

```
load fisheriris
```

The column vector, `species`, consists of iris flowers of three different species, `setosa`, `versicolor`, `virginica`. The double matrix `meas` consists of four types of measurements on the flowers, the length and width of sepals and petals in centimeters, respectively.



Define the nominal response variable.

```
sp = nominal(species);
sp = double(sp);
```

Now in `sp`, 1, 2, and 3 indicate the species *setosa*, *versicolor*, and *virginica*, respectively.

Fit a nominal model to estimate the species using the flower measurements as the predictor variables.

```
[B, dev, stats] = mnrfits(meas, sp);
```

Estimate the probability of being a certain kind of species for an iris flower having the measurements (6.3, 2.8, 4.9, 1.7).

```
x = [6.3, 2.8, 4.9, 1.7];
pihat = mnrvl(B, x);
pihat
```

```
pihat =
      0      0.3977      0.6023
```

The probability of an iris flower having the measurements (6.3, 2.8, 4.9, 1.7) being a *setosa* is 0, a *versicolor* is 0.3977, and a *virginica* is 0.6023.

### Estimate Upper and Lower Error Bounds for Probability Estimates of Ordinal Responses

Fit a multinomial regression model for categorical responses with natural ordering among categories. Then estimate the upper and lower confidence bounds for the category probability estimates.

Load the sample data and define the predictor variables.

```
load('carbig.mat')
X = [Acceleration Displacement Horsepower Weight];
```

The predictor variables are the acceleration, engine displacement, horsepower, and the weight of the cars. The response variable is miles per gallon (MPG).

Create an ordinal response variable categorizing MPG into four levels from 9 to 48 mpg.

```
miles = ordinal(MPG, {'1', '2', '3', '4'}, [], [9, 19, 29, 39, 48]);
miles = double(miles);
```

Now in miles, 1 indicates the cars with miles per gallon from 9 to 19, and 2 indicates the cars with miles per gallon from 20 to 29. Similarly, 3 and 4 indicate the cars with miles per gallon from 30 to 39 and 40 to 48, respectively.

Fit a multinomial regression model for the response variable miles. For an ordinal model, the default 'link' is logit and the default 'interactions' is 'off'.

```
[B, dev, stats] = mnrfits(X, miles, 'model', 'ordinal');
```

Compute the probability estimates and 95% error bounds for probability confidence intervals for miles per gallon of a car with  $x = (12, 113, 110, 2670)$ .

```
x = [12, 113, 110, 2670];
[pihat, dlow, hi] = mnrvl(B, x, stats, 'model', 'ordinal');
pihat
```

```
pihat =
    0.0615    0.8426    0.0932    0.0027
```

Calculate the confidence bounds for the category probability estimates.

```
LL = pihat - dlow;
UL = pihat + hi;
[LL; UL]

ans =
    0.0073    0.7829    0.0283   -0.0003
    0.1157    0.9022    0.1580    0.0057
```

### Estimate Category Counts and Error Bounds for Nominal Responses

Fit a multinomial regression for nominal outcomes and estimate the category counts.

Load the sample data.

```
load fisheriris
```

The column vector, `species`, consists of iris flowers of three different species, `setosa`, `versicolor`, and `virginica`. The double matrix `meas` consists of four types of measurements on the flowers, the length and width of sepals and petals in centimeters, respectively.

Define the nominal response variable.

```
sp = nominal(species);
sp = double(sp);
```

Now in `sp`, 1, 2, and 3 indicate the species `setosa`, `versicolor`, and `virginica`, respectively.

Fit a nominal model to estimate the species based on the flower measurements.

```
[B,dev,stats] = mnrfits(meas,sp);
```

Estimate the number in each species category for a sample of 100 iris flowers all with the measurements (6.3, 2.8, 4.9, 1.7).

```
x = [6.3, 2.8, 4.9, 1.7];
yhat = mnrv(B,x,18)
```

```
yhat =
      0      7.1578     10.8422
```

Estimate the error bounds for the counts.

```
[yhat,dlow,hi] = mnrv(B,x,18,stats,'model','nominal');
```

Calculate the confidence bounds for the category probability estimates.

```
LL = yhat - dlow;
UL = yhat + hi;
[LL;UL]

ans =
      0      3.3019      6.9863
```

```
0 11.0137 14.6981
```

### Plot the Count Estimates

Create sample data with one predictor variable and a categorical response variable with three categories.

```
x = [-3 -2 -1 0 1 2 3]';
Y = [1 11 13; 2 9 14; 6 14 5; 5 10 10; ...
     5 14 6; 7 13 5; 8 11 6];
[Y x]
```

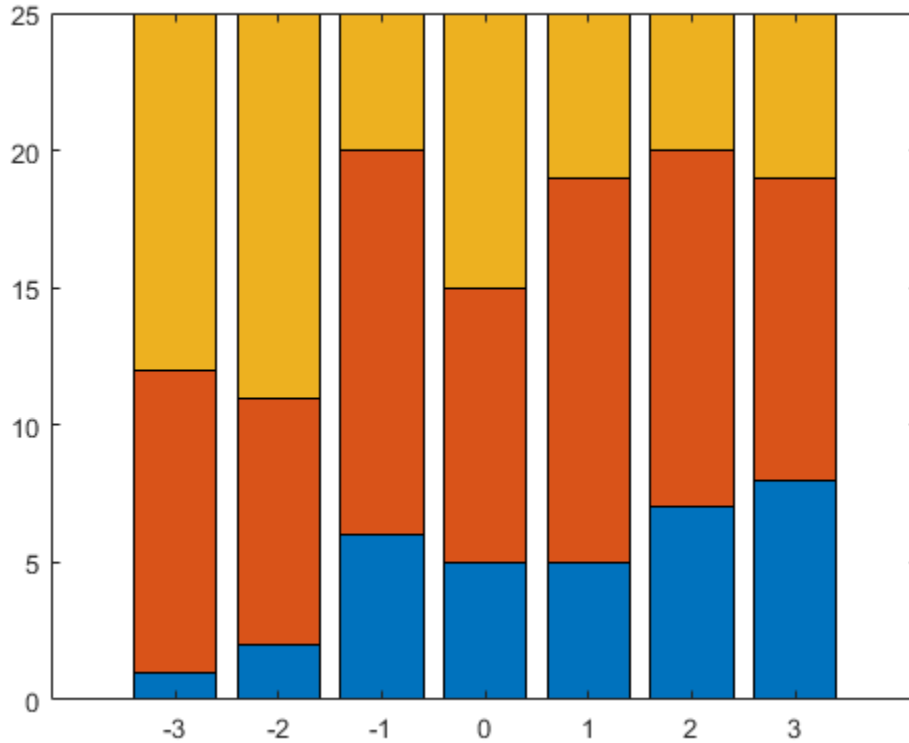
```
ans =
```

```
1 11 13 -3
2 9 14 -2
6 14 5 -1
5 10 10 0
5 14 6 1
7 13 5 2
8 11 6 3
```

There are observations on seven different values of the predictor variable  $x$ . The response variable  $Y$  has three categories and the data shows how many of the 25 individuals are in each category of  $Y$  for each observation of  $x$ . For example, when  $x$  is -3, 1 of 25 individuals is observed in category 1, 11 observed in category 2, and 13 observed in category 3. Similarly, when  $x$  is 1, 5 of the individuals are observed in category 1, 14 are observed in category 2, and 6 are observed in category 3.

Plot the number in each category versus the  $x$  values, on a stacked bar graph.

```
bar(x, Y, 'stacked');
ylim([0 25]);
```



Fit a nominal model for the individual response category probabilities, with separate slopes on the single predictor variable,  $x$ , for each category.

```
betaHatNom = mnrfits(x,Y,'model','nominal',...  
                    'interactions','on')
```

```
betaHatNom =
```

```
-0.6028    0.3832  
 0.4068    0.1948
```

The first row of `betaHatOrd` contains the intercept terms for the first two response categories. The second row contains the slopes. `mnrfit` accepts the third category as the reference category and hence assumes the coefficients for the third category are zero.

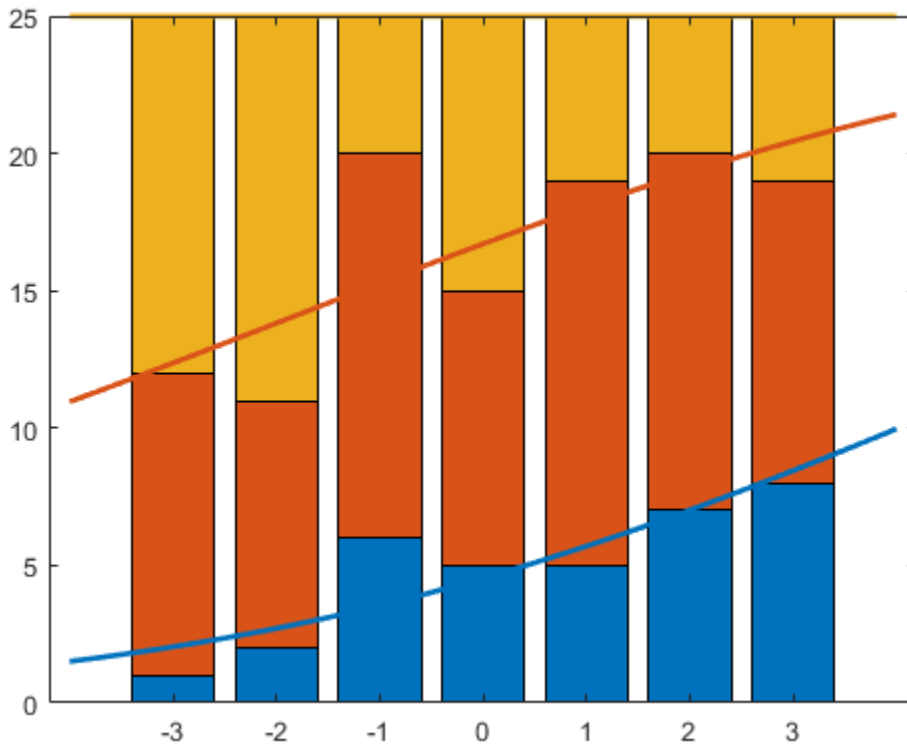
Compute the predicted probabilities for the three response categories.

```
xx = linspace(-4,4)';
piHatNom = mnrfval(betaHatNom,xx,'model','nominal',...
    'interactions','on');
```

The probability of being in the third category is simply  $1 - P(Y = 1) - P(Y = 2)$ .

Plot the estimated cumulative number in each category on the bar graph.

```
line(xx,cumsum(25*piHatNom,2),'LineWidth',2);
```



The cumulative probability for the third category is always 1.

Now, fit a "parallel" ordinal model for the cumulative response category probabilities, with a common slope on the single predictor variable,  $x$ , across all categories:

```
betaHatOrd = mnrfit(x,Y,'model','ordinal',...  
    'interactions','off')
```

```
betaHatOrd =
```

```
-1.5001  
0.7266  
0.2642
```

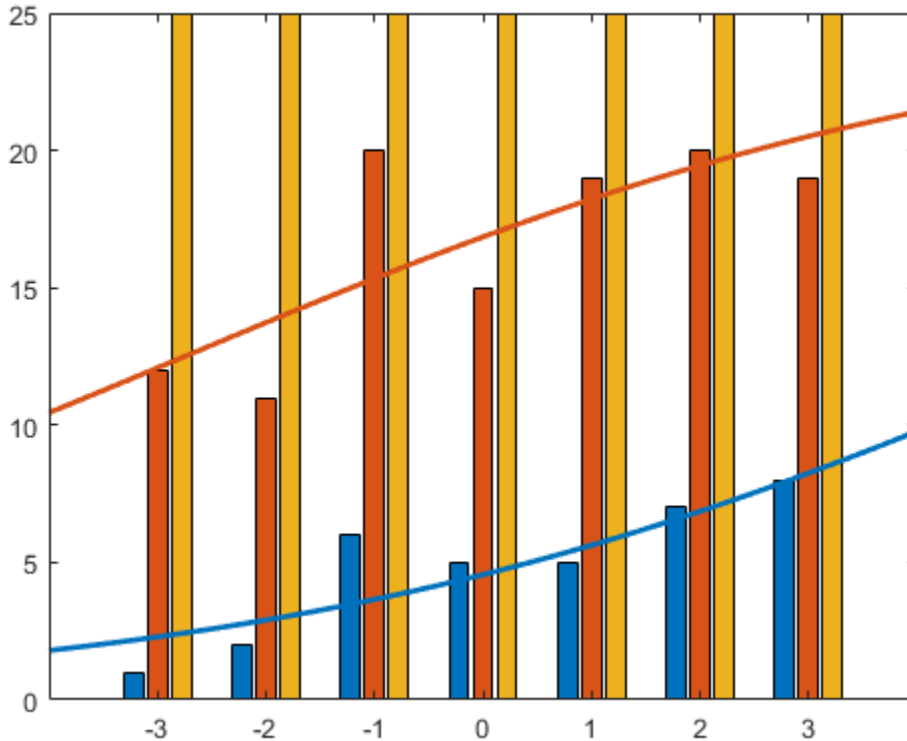
The first two elements of `betaHatOrd` are the intercept terms for the first two response categories. The last element of `betaHatOrd` is the common slope.

Compute the predicted cumulative probabilities for the first two response categories. The cumulative probability for the third category is always 1.

```
piHatOrd = mnrvl(betaHatOrd,xx,'type','cumulative',...  
    'model','ordinal','interactions','off');
```

Plot the estimated cumulative number on the bar graph of the observed cumulative number.

```
figure()  
bar(x,cumsum(Y,2),'grouped');  
ylim([0 25]);  
line(xx,25*piHatOrd,'LineWidth',2);
```



## Input Arguments

### **B** — Coefficient estimates

vector or matrix returned by `mnrfit`

Coefficient estimates for the multinomial logistic regression model, specified as a vector or matrix returned by `mnrfit`. It is a vector or matrix depending on the model and interactions.

Example: `B = mnrfit(X,y); pihat = mnrval(B,X)`

Data Types: `single` | `double`



**x — Sample data**

matrix

Sample data on predictors, specified as an  $n$ -by- $p$ .  $X$  contains  $n$  observations for  $p$  predictors.

---

**Note** `mnrval` automatically includes a constant term in all models. Do not enter a column of 1s in  $X$ .

---

Example: `pihat = mnrval(B,X)`

Data Types: `single | double`

**stats — Model statistics**structure returned by `mnrfit`

Model statistics, specified as a structure returned by `mnrfit`. You must use the `stats` input argument in `mnrval` to compute the lower and upper error bounds on the category probabilities and counts.

Example: `[B,dev,stats] = mnrfit(X,y); [pihat,dlo,dhi] = mnrval(B,X,stats)`

**ssize — Sample sizes**

column vector of positive integers

Sample sizes to return the number of items in response categories for each combination of the predictor variables, specified as an  $n$ -by-1 column vector of positive integers.

For example, for a response variable having three categories, if an observation of the number of individuals in each category is  $y_1$ ,  $y_2$ , and  $y_3$ , respectively, then the sample size,  $m$ , for that observation is  $m = y_1 + y_2 + y_3$ .

If the sample sizes for  $n$  observations are in vector `sample`, then you can enter the sample sizes as follows.

Example: `yhat = mnrval(B,X,sample)`

Data Types: `single | double`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'model', 'ordinal', 'link', 'probit', 'type', 'cumulative'` specifies that `mnrval` returns the estimates for cumulative probabilities for an ordinal model with a probit link function.

### **model** — Type of multinomial model

`'nominal'` (default) | `'ordinal'` | `'hierarchical'`

Type of multinomial model fit by `mnrfit`, specified as the comma-separated pair consisting of `'model'` and one of the following.

<code>'nominal'</code>	Default. Specify when there is no ordering among the response categories.
<code>'ordinal'</code>	Specify when there is a natural ordering among the response categories.
<code>'hierarchical'</code>	Specify when the choice of response category is sequential.

Example: `'model', 'ordinal'`

### **interactions** — Indicator for interaction between multinomial categories and coefficients

`'on'` | `'off'`

Indicator for an interaction between the multinomial categories and coefficients in the model fit by `mnrfit`, specified as the comma-separated pair consisting of `'interactions'` and one of the following.

<code>'on'</code>	Default for nominal and hierarchical models. Specify to fit a model with different intercepts and coefficients across categories.
<code>'off'</code>	Default for ordinal models. Specify to fit a model with different intercepts, but a common set of coefficients for the predictor variables, across all multinomial categories. This is often described as <i>parallel regression</i> or <i>proportional odds model</i> .

Example: `'interactions', 'off'`

Data Types: `logical`

**link — Link function**

'logit' (default) | 'probit' | 'comploglog' | 'loglog'

Link function `mnrfit` uses for ordinal and hierarchical models, specified as the comma-separated pair consisting of 'link' and one of the following.

'logit'	Default. $f(\gamma) = \ln(\gamma/(1 - \gamma))$
'probit'	$f(\gamma) = \Phi^{-1}(\gamma)$ — error term is normally distributed with variance 1
'comploglog'	Complementary log-log $f(\gamma) = \ln(-\ln(1 - \gamma))$
'loglog'	$f(\gamma) = \ln(-\ln(\gamma))$

The link function defines the relationship between response probabilities and the linear combination of predictors,  $X\beta$ .

$\gamma$  might be cumulative or conditional probabilities based on whether the model is for an ordinal or a sequential/nested response.

You cannot specify the 'link' parameter for nominal models; these always use a multinomial logit link,

$$\ln\left(\frac{\pi_j}{\pi_r}\right) = \beta_{j0} + \beta_{j1}X_{j1} + \beta_{j2}X_{j2} + \cdots + \beta_{jp}X_{jp}, \quad j = 1, \dots, k-1,$$

where  $\pi$  stands for a categorical probability, and  $r$  corresponds to the reference category,  $k$  is the total number of response categories,  $p$  is the number of predictor variables. `mnrfit` uses the last category as the reference category for nominal models.

Example: 'link','loglog'

**type — Type of probabilities or counts to estimate**

'category' (default) | 'cumulative' | 'conditional'

Type of probabilities or counts to estimate, specified as the comma-separated pair including 'type' and one of the following.

'category'	Default. Specify to return predictions and error bounds for the probabilities (or counts) of the $k$ multinomial categories.
------------	--

'cumulative'	Specify to return predictions and confidence bounds for the cumulative probabilities (or counts) of the first $k - 1$ multinomial categories, as an $n$ -by- $(k - 1)$ matrix. The predicted cumulative probability for the $k$ th category is always 1.
'conditional'	Specify to return predictions and error bounds in terms of the first $k - 1$ conditional category probabilities (counts), i.e., the probability (count) for category $j$ , given an outcome in category $j$ or higher. When 'type' is 'conditional', and you supply the sample size argument <code>ssize</code> , the predicted counts at each row of <code>X</code> are conditioned on the corresponding element of <code>ssize</code> , across all categories.

Example: `'type','cumulative'`

#### **confidence** — Confidence level

0.95 (default) | scalar value in the range (0,1)

Confidence level for the error bounds, specified as the comma-separated pair consisting of 'confidence' and a scalar value in the range (0,1).

For example, for 99% error bounds, you can specify the confidence as follows:

Example: `'confidence',0.99`

Data Types: `single` | `double`

## Output Arguments

#### **pihat** — Probability estimates

$n$ -by- $(k - 1)$  matrix

Probability estimates for each multinomial category, returned as an  $n$ -by- $(k - 1)$  matrix, where  $n$  is the number of observations, and  $k$  is the number of response categories.

#### **yhat** — Count estimates

$n$ -by- $k - 1$  matrix

Count estimates for the number in each response category, returned as an  $n$ -by- $k - 1$  matrix, where  $n$  is the number of observations, and  $k$  is the number of response categories.

**dlow — Lower error bound**

column vector

Lower error bound to compute the lower confidence bound for `pihat` or `yhat`, returned as a column vector.

The lower confidence bound for `pihat` is `pihat` minus `dlow`. Similarly, the lower confidence bound for `yhat` is `yhat` minus `dlow`. Confidence bounds are nonsimultaneous and only apply to the fitted curve, not to new observations.

**dhi — Upper error bound**

column vector

Upper error bound to compute the upper confidence bound for `pihat` or `yhat`, returned as a column vector.

The upper confidence bound for `pihat` is `pihat` plus `dhi`. Similarly, the upper confidence bound for `yhat` is `yhat` plus `dhi`. Confidence bounds are nonsimultaneous and only apply to the fitted curve, not to new observations.

## References

[1] McCullagh, P., and J. A. Nelder. *Generalized Linear Models*. New York: Chapman & Hall, 1990.

## See Also

`fitglm` | `glmfit` | `glmval` | `mnrfit`

## Topics

“Multinomial Models for Nominal Responses” on page 12-2

“Multinomial Models for Ordinal Responses” on page 12-5

“Hierarchical Multinomial Models” on page 12-9

Introduced in R2006b

## moment

Central moments

### Syntax

```
m = moment(X,order)
moment(X,order,dim)
```

### Description

`m = moment(X,order)` returns the central sample moment of `X` specified by the positive integer `order`. For vectors, `moment(x,order)` returns the central moment of the specified order for the elements of `x`. For matrices, `moment(X,order)` returns central moment of the specified order for each column. For N-dimensional arrays, `moment` operates along the first nonsingleton dimension of `X`.

`moment(X,order,dim)` takes the moment along dimension `dim` of `X`.

### Examples

```
X = randn([6 5])
X =
    1.1650    0.0591    1.2460   -1.2704   -0.0562
    0.6268    1.7971   -0.6390    0.9846    0.5135
    0.0751    0.2641    0.5774   -0.0449    0.3967
    0.3516    0.8717   -0.3600   -0.7989    0.7562
   -0.6965   -1.4462   -0.1356   -0.7652    0.4005
    1.6961   -0.7012   -1.3493    0.8617   -1.3414

m = moment(X,3)
m =
   -0.0282    0.0571    0.1253    0.1460   -0.4486
```

## Tips

Note that the central first moment is zero, and the second central moment is the variance computed using a divisor of  $n$  rather than  $n - 1$ , where  $n$  is the length of the vector  $x$  or the number of rows in the matrix  $X$ .

The central moment of order  $k$  of a distribution is defined as

$$m_k = E(x - \mu)^k$$

where  $E(x)$  is the expected value of  $x$ .

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- If `order` is nonintegral and `X` is real, use `moment(complex(X), order)`.

### See Also

`kurtosis` | `mean` | `skewness` | `std` | `var`

Introduced before R2006a

## mu property

**Class:** gmdistribution

Input matrix of means mu

## Description

Input matrix of means mu.



# multcompare

Multiple comparison test

## Syntax

```
c = multcompare(stats)
c = multcompare(stats,Name,Value)
[c,m] = multcompare(____)
[c,m,h] = multcompare(____)
[c,m,h,gnames] = multcompare(____)
```

## Description

`c = multcompare(stats)` returns a matrix `c` of the pairwise comparison results from a multiple comparison test using the information contained in the `stats` structure. `multcompare` also displays an interactive graph of the estimates and comparison intervals. Each group mean is represented by a symbol, and the interval is represented by a line extending out from the symbol. Two group means are significantly different if their intervals are disjoint; they are not significantly different if their intervals overlap. If you use your mouse to select any group, then the graph will highlight all other groups that are significantly different, if any.

`c = multcompare(stats,Name,Value)` returns a matrix of pairwise comparison results, `c`, using additional options specified by one or more `Name,Value` pair arguments. For example, you can specify the confidence interval, or the type of critical value to use in the multiple comparison.

`[c,m] = multcompare(____)` also returns a matrix, `m`, which contains estimated values of the means (or whatever statistics are being compared) for each group and the corresponding standard errors. You can use any of the previous syntaxes.

`[c,m,h] = multcompare(____)` also returns a handle, `h`, to the comparison graph.

`[c,m,h,gnames] = multcompare(____)` also returns a cell array, `gnames`, which contains the names of the groups.

## Examples

### Multiple Comparison of Group Means

Load the sample data.

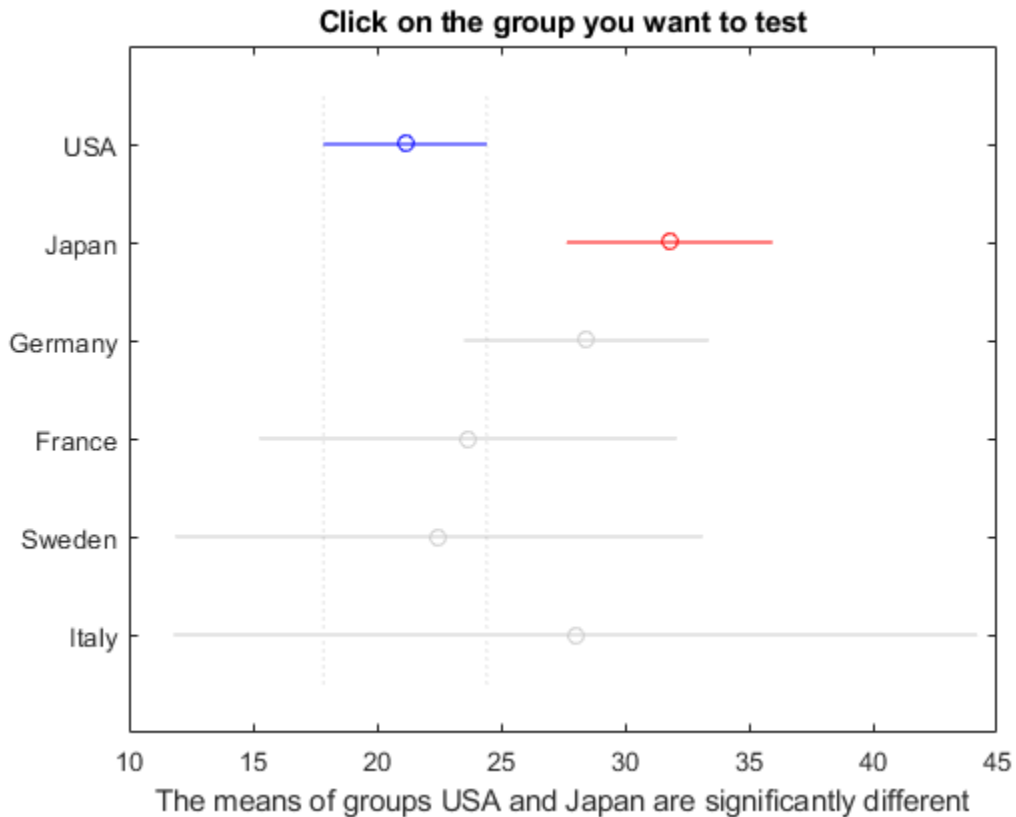
```
load carsmall
```

Perform a one-way analysis of variance (ANOVA) to see if there is any difference between the mileage of the cars by origin.

```
[p,t,stats] = anova1(MPG,Origin,'off');
```

Perform a multiple comparison of the group means.

```
[c,m,h,nms] = multcompare(stats);
```



multcompare displays the estimates with comparison intervals around them. You can click the graphs of each country to compare its mean to those of other countries.

Now display the mean estimates and the standard errors with the corresponding group names.

```
[nms num2cell(m)]
```

```
ans =
```

```
6x3 cell array
```

```
 {'USA' } { [21.1328] } { [0.8814] }
```

```
{'Japan' }    {[31.8000]}    {[1.8206]}
{'Germany'} {[28.4444]} {[2.3504]}
{'France' } {[23.6667]} {[4.0711]}
{'Sweden' } {[22.5000]} {[4.9860]}
{'Italy'  } {[ 28]}    {[7.0513]}
```

## Multiple Comparisons for Two-Way ANOVA

Load the sample data.

```
load popcorn
popcorn
```

```
popcorn =
```

```
5.5000  4.5000  3.5000
5.5000  4.5000  4.0000
6.0000  4.0000  3.0000
6.5000  5.0000  4.0000
7.0000  5.5000  5.0000
7.0000  5.0000  4.5000
```

The data is from a study of popcorn brands and popper types (Hogg 1987). The columns of the matrix `popcorn` are brands (Gourmet, National, and Generic). The rows are popper types oil and air. In the study, researchers popped a batch of each brand three times with each popper. The values are the yield in cups of popped popcorn.

Perform a two-way ANOVA. Also compute the statistics that you need to perform a multiple comparison test on the main effects.

```
[~,~,stats] = anova2(popcorn,3,'off')
```

```
stats =
```

```
struct with fields:
    source: 'anova2'
    sigmasq: 0.1389
    colmeans: [6.2500 4.7500 4]
```

```

      coln: 6
rowmeans: [4.5000 5.5000]
      rown: 9
      inter: 1
      pval: 0.7462
      df: 12

```

The stats structure includes

- The mean squared error (`sigmasq`)
- The estimates of the mean yield for each popcorn brand (`colmeans`)
- The number of observations for each popcorn brand (`coln`)
- The estimate of the mean yield for each popper type (`rowmeans`)
- The number of observations for each popper type (`rown`)
- The number of interactions (`inter`)
- The  $p$ -value that shows the significance level of the interaction term (`pval`)
- The error degrees of freedom (`df`).

Perform a multiple comparison test to see if the popcorn yield differs between pairs of popcorn brands (columns).

```
c = multcompare(stats)
```

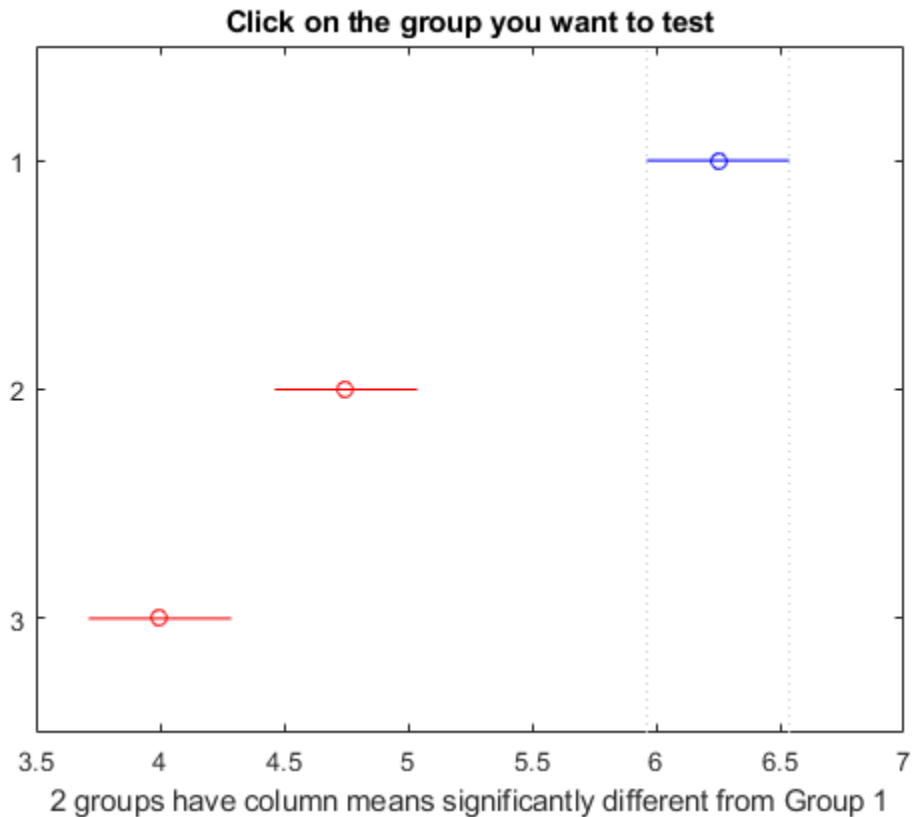
Note: Your model includes an interaction term. A test of main effects can be difficult to interpret when the model includes interactions.

```
c =
```

```

1.0000    2.0000    0.9260    1.5000    2.0740    0.0000
1.0000    3.0000    1.6760    2.2500    2.8240    0.0000
2.0000    3.0000    0.1760    0.7500    1.3240    0.0116

```



The first two columns of  $c$  show the groups that are compared. The fourth column shows the difference between the estimated group means. The third and fifth columns show the lower and upper limits for 95% confidence intervals for the true mean difference. The sixth column contains the  $p$ -value for a hypothesis test that the corresponding mean difference is equal to zero. All  $p$ -values (0, 0, and 0.0116) are very small, which indicates that the popcorn yield differs across all three brands.

The figure shows the multiple comparison of the means. By default, the group 1 mean is highlighted and the comparison interval is in blue. Because the comparison intervals for the other two groups do not intersect with the intervals for the group 1 mean, they are highlighted in red. This lack of intersection indicates that both means are different than group 1 mean. Select other group means to confirm that all group means are significantly different from each other.

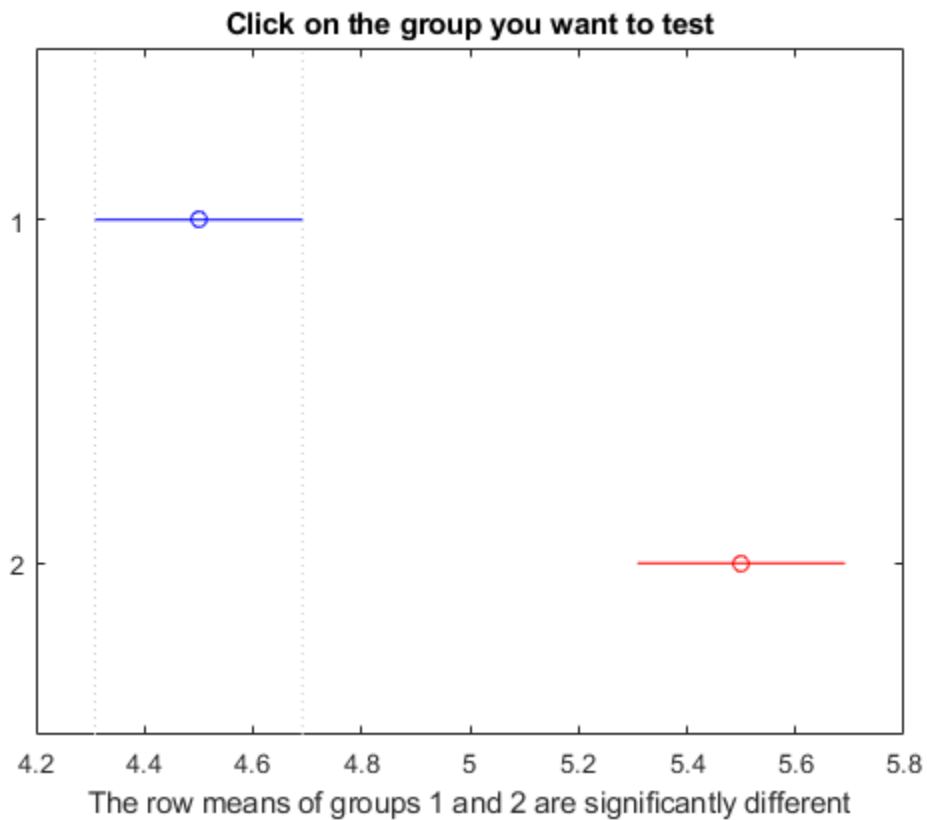
Perform a multiple comparison test to see the popcorn yield differs between the two popper types (rows).

```
c = multcompare(stats, 'Estimate', 'row')
```

Note: Your model includes an interaction term. A test of main effects can be difficult to interpret when the model includes interactions.

```
c =
```

```
1.0000    2.0000   -1.3828   -1.0000   -0.6172    0.0001
```



The small  $p$ -value of 0.0001 indicates that the popcorn yield differs between the two popper types (air and oil). The figure shows the same results. The disjoint comparison intervals indicate that the group means are significantly different from each other.

### Multiple Comparisons for Three-Way ANOVA

Load the sample data.

```
y = [52.7 57.5 45.9 44.5 53.0 57.0 45.9 44.0]';  
g1 = [1 2 1 2 1 2 1 2];  
g2 = {'hi'; 'hi'; 'lo'; 'lo'; 'hi'; 'hi'; 'lo'; 'lo'};  
g3 = {'may'; 'may'; 'may'; 'may'; 'june'; 'june'; 'june'; 'june'};
```

$y$  is the response vector and  $g1$ ,  $g2$ , and  $g3$  are the grouping variables (factors). Each factor has two levels, and every observation in  $y$  is identified by a combination of factor levels. For example, observation  $y(1)$  is associated with level 1 of factor  $g1$ , level 'hi' of factor  $g2$ , and level 'may' of factor  $g3$ . Similarly, observation  $y(6)$  is associated with level 2 of factor  $g1$ , level 'hi' of factor  $g2$ , and level 'june' of factor  $g3$ .

Test if the response is the same for all factor levels. Also compute the statistics required for multiple comparison tests.

```
[~,~,stats] = anovan(y,{g1 g2 g3}, 'model', 'interaction', ...  
    'varnames', {'g1', 'g2', 'g3'});
```



Figure 1: N-Way ANOVA

File Edit View Insert Tools Desktop Window Help

### Analysis of Variance

Source	Sum Sq.	d. f.	Mean Sq.	F	Prob>F
g1	3.781	1	3.781	336.11	0.0347
g2	199.001	1	199.001	17689	0.0048
g3	0.061	1	0.061	5.44	0.2578
g1*g2	18.301	1	18.301	1626.78	0.0158
g1*g3	0.211	1	0.211	18.78	0.1444
g2*g3	0.011	1	0.011	1	0.5
Error	0.011	1	0.011		
Total	221.379	7			

Constrained (Type III) sums of squares.

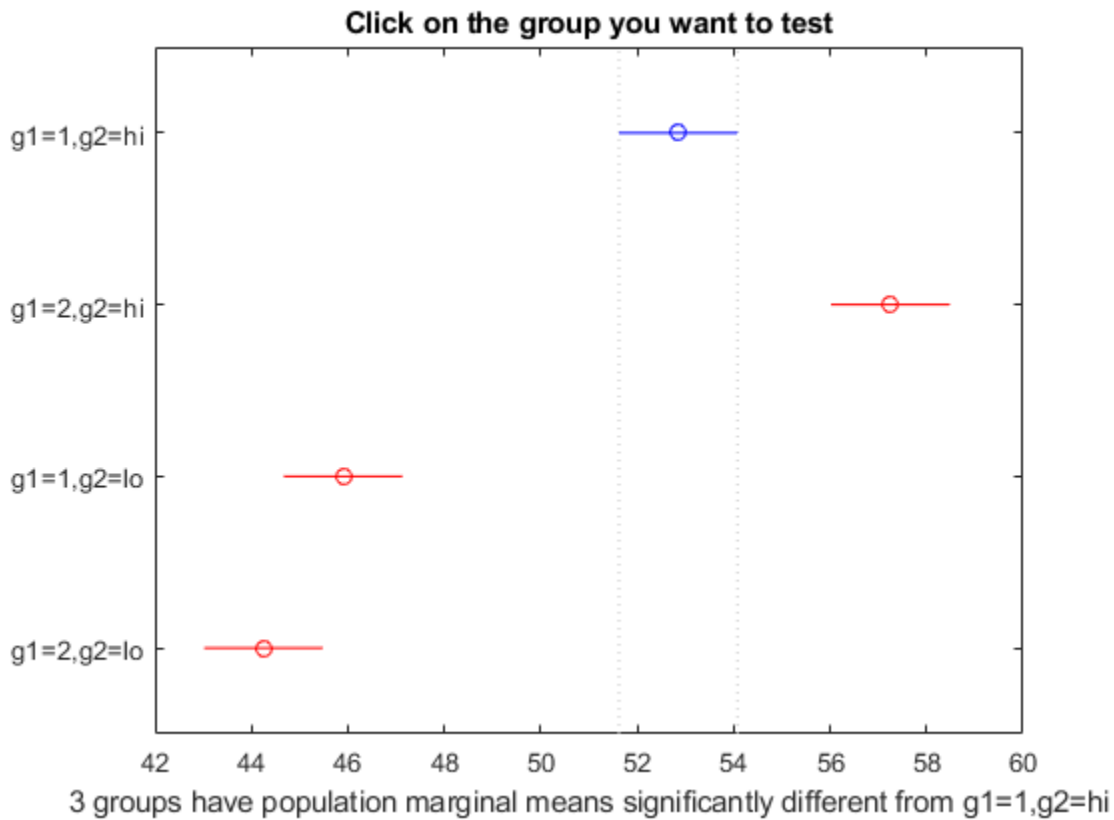
The  $p$ -value of 0.2578 indicates that the mean responses for levels 'may' and 'june' of factor  $g_3$  are not significantly different. The  $p$ -value of 0.0347 indicates that the mean responses for levels 1 and 2 of factor  $g_1$  are significantly different. Similarly, the  $p$ -value of 0.0048 indicates that the mean responses for levels 'hi' and 'lo' of factor  $g_2$  are significantly different.

Perform multiple comparison tests to find out which groups of the factors  $g_1$  and  $g_2$  are significantly different.

```
results = multcompare(stats, 'Dimension', [1 2])
```

```
results =
```

1.0000	2.0000	-6.8604	-4.4000	-1.9396	0.0280
1.0000	3.0000	4.4896	6.9500	9.4104	0.0177
1.0000	4.0000	6.1396	8.6000	11.0604	0.0143
2.0000	3.0000	8.8896	11.3500	13.8104	0.0108
2.0000	4.0000	10.5396	13.0000	15.4604	0.0095
3.0000	4.0000	-0.8104	1.6500	4.1104	0.0745



`multcompare` compares the combinations of groups (levels) of the two grouping variables, `g1` and `g2`. In the `results` matrix, the number 1 corresponds to the combination of level 1 of `g1` and level `hi` of `g2`, the number 2 corresponds to the combination of level 2 of `g1` and level `hi` of `g2`. Similarly, the number 3 corresponds to the combination of level 1 of `g1` and level `lo` of `g2`, and the number 4 corresponds to the combination of level 2 of `g1` and level `lo` of `g2`. The last column of the matrix contains the  $p$ -values.

For example, the first row of the matrix shows that the combination of level 1 of `g1` and level `hi` of `g2` has the same mean response values as the combination of level 2 of `g1` and level `hi` of `g2`. The  $p$ -value corresponding to this test is 0.0280, which indicates that the mean responses are significantly different. You can also see this result in the figure. The blue bar shows the comparison interval for the mean response for the combination of level 1 of `g1` and level `hi` of `g2`. The red bars are the comparison intervals for the mean response for other group combinations. None of the red bars overlap with the blue bar, which means the mean response for the combination of level 1 of `g1` and level `hi` of `g2` is significantly different from the mean response for other group combinations.

You can test the other groups by clicking on the corresponding comparison interval for the group. The bar you click on turns to blue. The bars for the groups that are significantly different are red. The bars for the groups that are not significantly different are gray. For example, if you click on the comparison interval for the combination of level 1 of `g1` and level `lo` of `g2`, the comparison interval for the combination of level 2 of `g1` and level `lo` of `g2` overlaps, and is therefore gray. Conversely, the other comparison intervals are red, indicating significant difference.

## Input Arguments

### **stats** — Test data

structure

Test data, specified as a structure. You can create a structure using one of the following functions:

- `anova1` — One-way analysis of variance.
- `anova2` — Two-way analysis of variance.
- `anovan` —  $N$ -way analysis of variance.

- `aocool` — Interactive analysis of covariance tool.
- `friedman` — Friedman's test.
- `kruskalwallis` — Kruskal-Wallis test.

`multcompare` does not support multiple comparisons using `anovan` output for a model that includes random or nested effects. The calculations for a random effects model produce a warning that all effects are treated as fixed. Nested models are not accepted.

Data Types: `struct`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

Example: `'Alpha', 0.01, 'CType', 'bonferroni', 'Display', 'off'` computes the Bonferroni critical values, conducts the hypothesis tests at the 1% significance level, and omits the interactive display.

### **Alpha** — Significance level

0.05 (default) | scalar value in the range (0,1)

Significance level of the multiple comparison test, specified as the comma-separated pair consisting of `'Alpha'` and a scalar value in the range (0,1). The value specified for `'Alpha'` determines the  $100 \times (1 - \alpha)$  confidence levels of the intervals returned in the matrix `c` and in the figure.

Example: `'Alpha', 0.01`

Data Types: `single` | `double`

### **CType** — Type of critical value

`'tukey-kramer'` (default) | `'hsd'` | `'lsd'` | `'bonferroni'` | `'dunn-sidak'` | `'scheffe'`

Type of critical value to use for the multiple comparison, specified as the comma-separated pair consisting of `'CType'` and one of the following.

Value	Description
'tukey-kramer' or 'hsd'	Tukey's honest significant difference criterion on page 9-33
'bonferroni'	Bonferroni method on page 9-33
'dunn-sidak'	Dunn and Sidák's approach on page 9-33
'lsd'	Fisher's least significant difference procedure on page 9-34
'scheffe'	Scheffé's S procedure on page 9-35

Example: 'CType', 'bonferroni'

### Display — Display toggle

'on' (default) | 'off'

Display toggle, specified as the comma-separated pair consisting of 'Display' and either 'on' or 'off'. If you specify 'on', then `multcompare` displays a graph of the estimates and their comparison intervals. If you specify 'off', then `multcompare` omits the graph.

Example: 'Display', 'off'

### Dimension — Dimension over which to calculate marginal means

1 (default) | positive integer value | vector of positive integer values

A vector specifying the dimension or dimensions over which to calculate the population marginal means, specified as a positive integer value, or a vector of such values. Use the 'Dimension' name-value pair only if you create the input structure `stats` using the function `anovan`.

For example, if you specify 'Dimension' as 1, then `multcompare` compares the means for each value of the first grouping variable, adjusted by removing effects of the other grouping variables as if the design were balanced. If you specify 'Dimension' as [1, 3], then `multcompare` computes the population marginal means for each combination of the first and third grouping variables, removing effects of the second grouping variable. If you fit a singular model, some cell means may not be estimable and any population marginal means that depend on those cell means will have the value NaN.

Population marginal means are described by Milliken and Johnson (1992) and by Searle, Speed, and Milliken (1980). The idea behind population marginal means is to remove any effect of an unbalanced design by fixing the values of the factors specified by

'Dimension', and averaging out the effects of other factors as if each factor combination occurred the same number of times. The definition of population marginal means does not depend on the number of observations at each factor combination. For designed experiments where the number of observations at each factor combination has no meaning, population marginal means can be easier to interpret than simple means ignoring other factors. For surveys and other studies where the number of observations at each combination does have meaning, population marginal means may be harder to interpret.

Example: 'Dimension', [1,3]

Data Types: single | double

### **Estimate** — Estimates to be compared

'column' (default) | 'row' | 'slope' | 'intercept' | 'pmm'

Estimates to be compared, specified as the comma-separated pair consisting of 'Estimate' and an allowable value. The allowable values for 'Estimate' depend on the function used to generate the input structure `stats`, according to the following table.

Source	Values
<code>anova1</code>	None. This name-value pair is ignored, and <code>multcompare</code> always compares the group means.
<code>anova2</code>	Either 'column' to compare column means, or 'row' to compare row means.
<code>anovan</code>	None. This name-value pair is ignored, and <code>multcompare</code> always compares the population marginal means as specified by the 'Dimension' name-value pair argument.
<code>aocool</code>	Either 'slope', 'intercept', or 'pmm' to compare slopes, intercepts, or population marginal means, respectively. If the analysis of covariance model did not include separate slopes, then 'slope' is not allowed. If it did not include separate intercepts, then no comparisons are possible.
<code>friedman</code>	None. This name-value pair is ignored, and <code>multcompare</code> always compares the average column ranks.
<code>kruskalwallis</code>	None. This name-value pair is ignored, and <code>multcompare</code> always compares the average group ranks.

Example: 'Estimate', 'row'

## Output Arguments

### **c** — Matrix of multiple comparison results

matrix of scalar values

Matrix of multiple comparison results, returned as an  $p$ -by-6 matrix of scalar values, where  $p$  is the number of pairs of groups. Each row of the matrix contains the result of one paired comparison test. Columns 1 and 2 contain the indices of the two samples being compared. Column 3 contains the lower confidence interval, column 4 contains the estimate, and column 5 contains the upper confidence interval. Column 6 contains the  $p$ -value for the hypothesis test that the corresponding mean difference is not equal to 0.

For example, suppose one row contains the following entries.

```
2.0000  5.0000  1.9442  8.2206  14.4971  0.0432
```

These numbers indicate that the mean of group 2 minus the mean of group 5 is estimated to be 8.2206, and a 95% confidence interval for the true difference of the means is [1.9442, 14.4971]. The  $p$ -value for the corresponding hypothesis test that the difference of the means of groups 2 and 5 is significantly different from zero is 0.0432.

In this example the confidence interval does not contain 0, so the difference is significant at the 5% significance level. If the confidence interval did contain 0, the difference would not be significant. The  $p$ -value of 0.0432 also indicates that the difference of the means of groups 2 and 5 is significantly different from 0.

### **m** — Matrix of estimates

matrix of scalar values

Matrix of the estimates, returned as a matrix of scalar values. The first column of  $m$  contains the estimated values of the means (or whatever statistics are being compared) for each group, and the second column contains their standard errors.

### **h** — Handle to the figure

handle

Handle to the figure containing the interactive graph, returned as a handle. The title of this graph contains instructions for interacting with the graph, and the  $x$ -axis label contains information about which means are significantly different from the selected mean. If you plan to use this graph for presentation, you may want to omit the title and the  $x$ -axis label. You can remove them using interactive features of the graph window, or you can use the following commands.

```
title('')  
xlabel('')
```

**gnames** — Group names

cell array of character vectors

Group names, returned as a cell array of character vectors. Each row of `gnames` contains the name of a group.

## Definitions

### Multiple Comparison Tests

Analysis of variance compares the means of several groups to test the hypothesis that they are all equal, against the general alternative that they are not all equal. Sometimes this alternative may be too general. You may need information about which pairs of means are significantly different, and which are not. A *multiple comparison test* can provide this information.

When you perform a simple *t*-test of one group mean against another, you specify a significance level that determines the cutoff value of the *t*-statistic. For example, you can specify the value `alpha = 0.05` to insure that when there is no real difference, you will incorrectly find a significant difference no more than 5% of the time. When there are many group means, there are also many pairs to compare. If you applied an ordinary *t*-test in this situation, the `alpha` value would apply to each comparison, so the chance of incorrectly finding a significant difference would increase with the number of comparisons. Multiple comparison procedures are designed to provide an upper bound on the probability that *any* comparison will be incorrectly found significant.

## References

- [1] Hochberg, Y., and A. C. Tamhane. *Multiple Comparison Procedures*. Hoboken, NJ: John Wiley & Sons, 1987.
- [2] Milliken, G. A., and D. E. Johnson. *Analysis of Messy Data, Volume I: Designed Experiments*. Boca Raton, FL: Chapman & Hall/CRC Press, 1992.



[3] Searle, S. R., F. M. Speed, and G. A. Milliken. “Population marginal means in the linear model: an alternative to least-squares means.” *American Statistician*. 1980, pp. 216–221.

## See Also

`anova1` | `anova2` | `anovan` | `aoctool` | `friedman` | `kruskalwallis`

**Introduced before R2006a**

## multcompare

**Class:** RepeatedMeasuresModel

Multiple comparison of estimated marginal means

### Syntax

```
tbl = multcompare(rm, var)
tbl = multcompare(rm, var, Name, Value)
```

### Description

`tbl = multcompare(rm, var)` returns multiple comparisons of the estimated marginal means based on the variable `var` in the repeated measures model `rm`.

`tbl = multcompare(rm, var, Name, Value)` returns multiple comparisons of the estimated marginal means with additional options specified by one or more `Name, Value` pair arguments.

For example, you can specify the comparison type or which variable to group by.

### Input Arguments

**rm** — Repeated measures model  
RepeatedMeasuresModel object

Repeated measures model, returned as a RepeatedMeasuresModel object.

For properties and methods of this object, see RepeatedMeasuresModel.

**var** — Variables for which to compute marginal means  
character vector

Variables for which to compute the marginal means, specified as a character vector representing the name of a between- or within-subjects factor in `rm`. If `var` is a between-subjects factor, it must be categorical.

Data Types: `char` | `cell`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### Alpha — Confidence level

0.05 (default) | scalar value in the range of 0 through 1

Confidence level of the confidence intervals for population marginal means, specified as the comma-separated pair consisting of 'alpha' and a scalar value in the range of 0 through 1. The confidence level is  $100 \cdot (1 - \text{alpha})\%$ .

Example: 'alpha', 0.01

Data Types: `double` | `single`

### By — Factor to perform comparisons by

character vector

Factor to do the comparisons by, specified as the comma-separated pair consisting of 'By' and a character vector. The comparison between levels of `var` occurs separately for each value of the factor you specify.

If you have more than one between-subjects factors, *A*, *B*, and *C*, and if you want to do the comparisons of *A* levels separately for each level of *C*, then specify *A* as the `var` argument and specify *C* using the 'By' argument as follows.

Example: 'By', *C*

Data Types: `char`

### ComparisonType — Type of critical value to use

'tukey-kramer' (default) | 'dunn-sidak' | 'bonferroni' | 'scheffe' | 'lsd'

Type of critical value to use, specified as the comma-separated pair consisting of 'ComparisonType' and one of the following.

Comparison Type	Definition
'tukey-kramer'	<p>Default. Also called Tukey's Honest Significant Difference procedure. It is based on the Studentized range distribution. According to the unproven Tukey-Kramer conjecture, it is also accurate for problems where the quantities being compared are correlated, as in analysis of covariance with unbalanced covariate values.</p>
'dunn-sidak'	<p>Use critical values from the <math>t</math> distribution, after an adjustment for multiple comparisons that was proposed by Dunn and proved accurate by Sidák. The critical value is</p> $ t  = \frac{ \bar{y}_i - \bar{y}_j }{\sqrt{MSE \left( \frac{1}{n_i} + \frac{1}{n_j} \right)}} > t_{1-\eta/2, v},$ <p>where</p> $\eta = 1 - (1 - \alpha)^{1/\binom{g}{2}}$ <p>and <math>ng</math> is the number of groups (marginal means). This procedure is similar to, but less conservative than, the Bonferroni procedure.</p>

Comparison Type	Definition
'bonferroni'	<p>Use critical values from the <math>t</math> distribution, after a Bonferroni adjustment to compensate for multiple comparisons. The critical value is</p> $t_{\alpha/2\binom{ng}{2},v},$ <p>where <math>ng</math> is the number of groups (marginal means), and <math>v</math> is the error degrees of freedom. This procedure is conservative, but usually less so than the Scheffé procedure.</p>
'scheffe'	<p>Use critical values from Scheffé's <math>S</math> procedure, derived from the <math>F</math> distribution. The critical value is</p> $\sqrt{(ng-1)F_{\alpha,ng-1,v}},$ <p>where <math>ng</math> is the number of groups (marginal means), and <math>v</math> is the error degrees of freedom. This procedure provides a simultaneous confidence level for comparisons of all linear combinations of the means, and it is conservative for comparisons of simple differences of pairs.</p>
'lsd'	<p>Least significant difference. This option uses plain <math>t</math>-tests. The critical value is</p> $t_{\alpha/2,v},$ <p>where <math>v</math> is the error degrees of freedom. It provides no protection against the multiple comparison problem.</p>

Example: 'ComparisonType', 'dunn-sidak'

## Output Arguments

### tbl — Results of multiple comparison

table

Results of multiple comparisons of estimated marginal means, returned as a table. `tbl` has the following columns.

Column Name	Description
Difference	Estimated difference between the corresponding two marginal means
StdErr	Standard error of the estimated difference between the corresponding two marginal means
pValue	$p$ -value for a test that the difference between the corresponding two marginal means is 0
Lower	Lower limit of simultaneous 95% confidence intervals for the true difference
Upper	Upper limit of simultaneous 95% confidence intervals for the true difference

## Examples

### Multiple Comparison of Estimated Marginal Means

Load the sample data.

```
load fisheriris
```

The column vector `species` consists of iris flowers of three different species: `setosa`, `versicolor`, and `virginica`. The double matrix `meas` consists of four types of measurements on the flowers: the length and width of sepals and petals in centimeters, respectively.

Store the data in a table array.

```
t = table(species,meas(:,1),meas(:,2),meas(:,3),meas(:,4),...
'VariableNames',{'species','meas1','meas2','meas3','meas4'});
Meas = dataset([1 2 3 4]','VarNames',{'Measurements'});
```

Fit a repeated measures model, where the measurements are the responses and the species is the predictor variable.

```
rm = fitrm(t, 'meas1-meas4~species', 'WithinDesign', Meas);
```

Perform a multiple comparison of the estimated marginal means of species.

```
tbl = multcompare(rm, 'species')
```

```
tbl =
```

```
6x7 table
```

species_1	species_2	Difference	StdErr	pValue	Lower
'setosa'	'versicolor'	-1.0375	0.060539	9.5606e-10	-1.1794
'setosa'	'virginica'	-1.7495	0.060539	9.5606e-10	-1.8914
'versicolor'	'setosa'	1.0375	0.060539	9.5606e-10	0.89562
'versicolor'	'virginica'	-0.712	0.060539	9.5606e-10	-0.85388
'virginica'	'setosa'	1.7495	0.060539	9.5606e-10	1.6076
'virginica'	'versicolor'	0.712	0.060539	9.5606e-10	0.57012

The small  $P$ -values (in the pValue field) indicate that the estimated marginal means for the three species significantly differ from each other.

### Perform Multiple Comparisons with Specified Options

Load the sample data.

```
load repeatedmeas
```

The table `between` includes the between-subject variables `age`, `IQ`, `group`, `gender`, and eight repeated measures `y1` through `y8` as responses. The table `within` includes the within-subject variables `w1` and `w2`. This is simulated data.

Fit a repeated measures model, where the repeated measures `y1` through `y8` are the responses, and `age`, `IQ`, `group`, `gender`, and the group-gender interaction are the predictor variables. Also specify the within-subject design matrix.

```
R = fitrm(between, 'y1-y8 ~ Group*Gender + Age + IQ', 'WithinDesign', within);
```

Perform a multiple comparison of the estimated marginal means based on the variable Group.

```
T = multcompare(R, 'Group')
```

T =

6x7 table

Group_1	Group_2	Difference	StdErr	pValue	Lower	Upper
A	B	4.9875	5.6271	0.65436	-9.1482	19.123
A	C	23.094	5.9261	0.0021493	8.2074	37.981
B	A	-4.9875	5.6271	0.65436	-19.123	9.1482
B	C	18.107	5.8223	0.013588	3.4805	32.732
C	A	-23.094	5.9261	0.0021493	-37.981	-8.2074
C	B	-18.107	5.8223	0.013588	-32.732	-3.4805

The small *P*-value of 0.0021493 indicates that there is significant difference between the marginal means of groups A and C. The *P*-value of 0.65436 indicates that the difference between the marginal means for groups A and B is not significantly different from 0.

`multcompare` uses the Tukey-Kramer test statistic by default. Change the comparison type to the Scheffe procedure.

```
T = multcompare(R, 'Group', 'ComparisonType', 'Scheffe')
```

T =

6x7 table

Group_1	Group_2	Difference	StdErr	pValue	Lower	Upper
A	B	4.9875	5.6271	0.67981	-9.7795	19.755
A	C	23.094	5.9261	0.0031072	7.5426	38.646
B	A	-4.9875	5.6271	0.67981	-19.755	9.7795
B	C	18.107	5.8223	0.018169	2.8273	33.386
C	A	-23.094	5.9261	0.0031072	-38.646	-7.5426
C	B	-18.107	5.8223	0.018169	-33.386	-2.8273



The Scheffe test produces larger  $P$ -values, but similar conclusions.

Perform multiple comparisons of estimated marginal means based on the variable Group for each gender separately.

```
T = multcompare(R, 'Group', 'By', 'Gender')
```

```
T =
```

```
12x8 table
```

Gender	Group_1	Group_2	Difference	StdErr	pValue	Lower
Female	A	B	4.1883	8.0177	0.86128	-15.953
Female	A	C	24.565	8.2083	0.017697	3.9449
Female	B	A	-4.1883	8.0177	0.86128	-24.329
Female	B	C	20.376	8.1101	0.049957	0.0033459
Female	C	A	-24.565	8.2083	0.017697	-45.184
Female	C	B	-20.376	8.1101	0.049957	-40.749
Male	A	B	5.7868	7.9498	0.74977	-14.183
Male	A	C	21.624	8.1829	0.038022	1.0676
Male	B	A	-5.7868	7.9498	0.74977	-25.757
Male	B	C	15.837	8.0511	0.14414	-4.3881
Male	C	A	-21.624	8.1829	0.038022	-42.179
Male	C	B	-15.837	8.0511	0.14414	-36.062

The results indicate that the difference between marginal means for groups A and B is not significant from 0 for either gender (corresponding  $P$ -values are 0.86128 for females and 0.74977 for males). The difference between marginal means for groups A and C is significant for both genders (corresponding  $P$ -values are 0.017697 for females and 0.038022 for males). While the difference between marginal means for groups B and C is significantly different from 0 for females ( $P$ -value is 0.049957), it is not significantly different from 0 for males ( $P$ -value is 0.14414).

## References

- [1] G. A. Milliken, and Johnson, D. E. *Analysis of Messy Data. Volume I: Designed Experiments*. New York, NY: Chapman & Hall, 1992.

## See Also

`fitrm` | `margmean` | `plotprofile`

# multivarichart

Multivari chart for grouped data

## Syntax

```
multivarichart(y, GROUP)
multivarichart(Y)
multivarichart(..., param1, val1, param2, val2, ...)
[charhandle, AXESH] = multivarichart(...)
```

## Description

`multivarichart(y, GROUP)` displays the multivari chart for the vector `y` grouped by entries in the cell array `GROUP`. Each cell of `GROUP` must contain a grouping variable that can be a categorical variable, numeric vector, character matrix, or single-column cell array of character vectors. `GROUP` can also be a matrix whose columns represent different grouping variables. Each grouping variable must have the same number of elements as `y`. The number of grouping variables must be 2, 3, or 4.

Each subplot of the plot matrix contains a multivari chart for the first and second grouping variables. The *x*-axis in each subplot indicates values of the first grouping variable. The legend at the bottom of the figure window indicates values of the second grouping variable. The subplot at position (*i,j*) is the multivari chart for the subset of `y` at the *i*th level of the third grouping variable and the *j*th level of the fourth grouping variable. If the third or fourth grouping variable is absent, it is considered to have only one level.

`multivarichart(Y)` displays the multivari chart for a matrix `Y`. The data in different columns represent changes in one factor. The data in different rows represent changes in another factor.

`multivarichart(..., param1, val1, param2, val2, ...)` specifies one or more of the following name/value pairs:

- `'varnames'` — Grouping variable names in a character matrix or a cell array of character vectors, one per grouping variable. Default names are `'X1'`, `'X2'`, ...

- 'plotorder' — 'sorted' or a vector containing a permutation of the integers from 1 to the number of grouping variables.

If 'plotorder' is 'sorted', the grouping variables are rearranged in descending order according to the number of levels in each variable.

If 'plotorder' is a vector, it indicates the order in which each grouping variable should be plotted. For example, [2, 3, 1, 4] indicates that the second grouping variable should be used as the *x*-axis of each subplot, the third grouping variable should be used as the legend, the first grouping variable should be used as the columns of the plot, and the fourth grouping variable should be used as the rows of the plot.

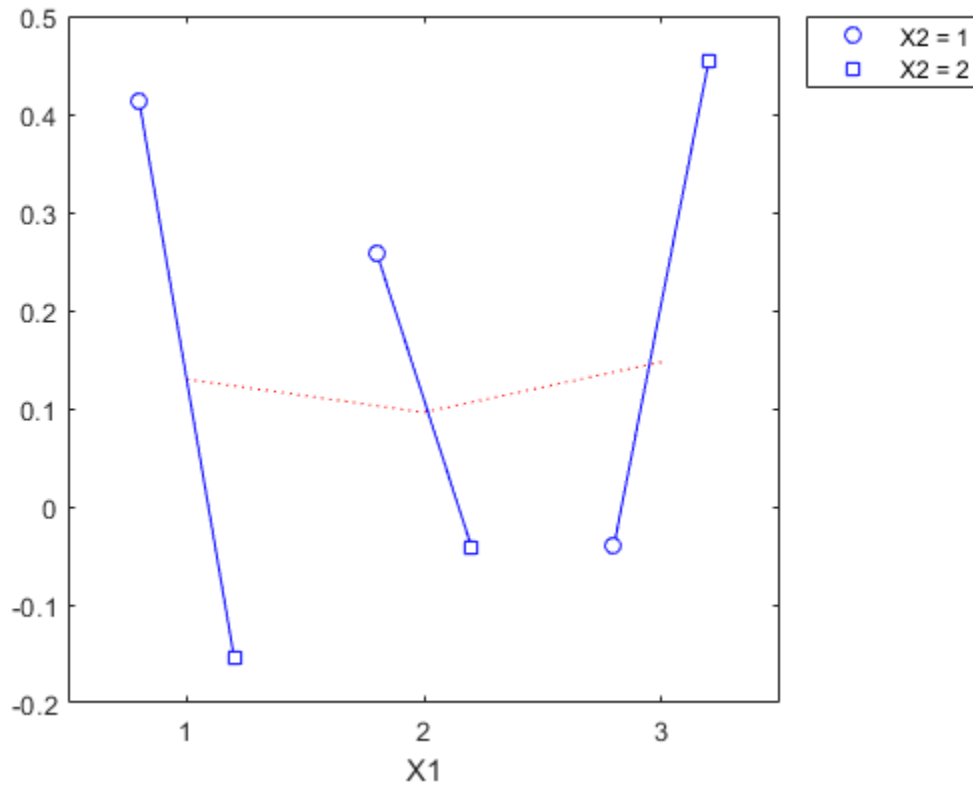
[charthandle,AXESH] = multivarichart(...) returns a handle charthandle to the figure window and a matrix AXESH of handles to the subplot axes.

## Examples

### Multivari Chart for Grouped Data

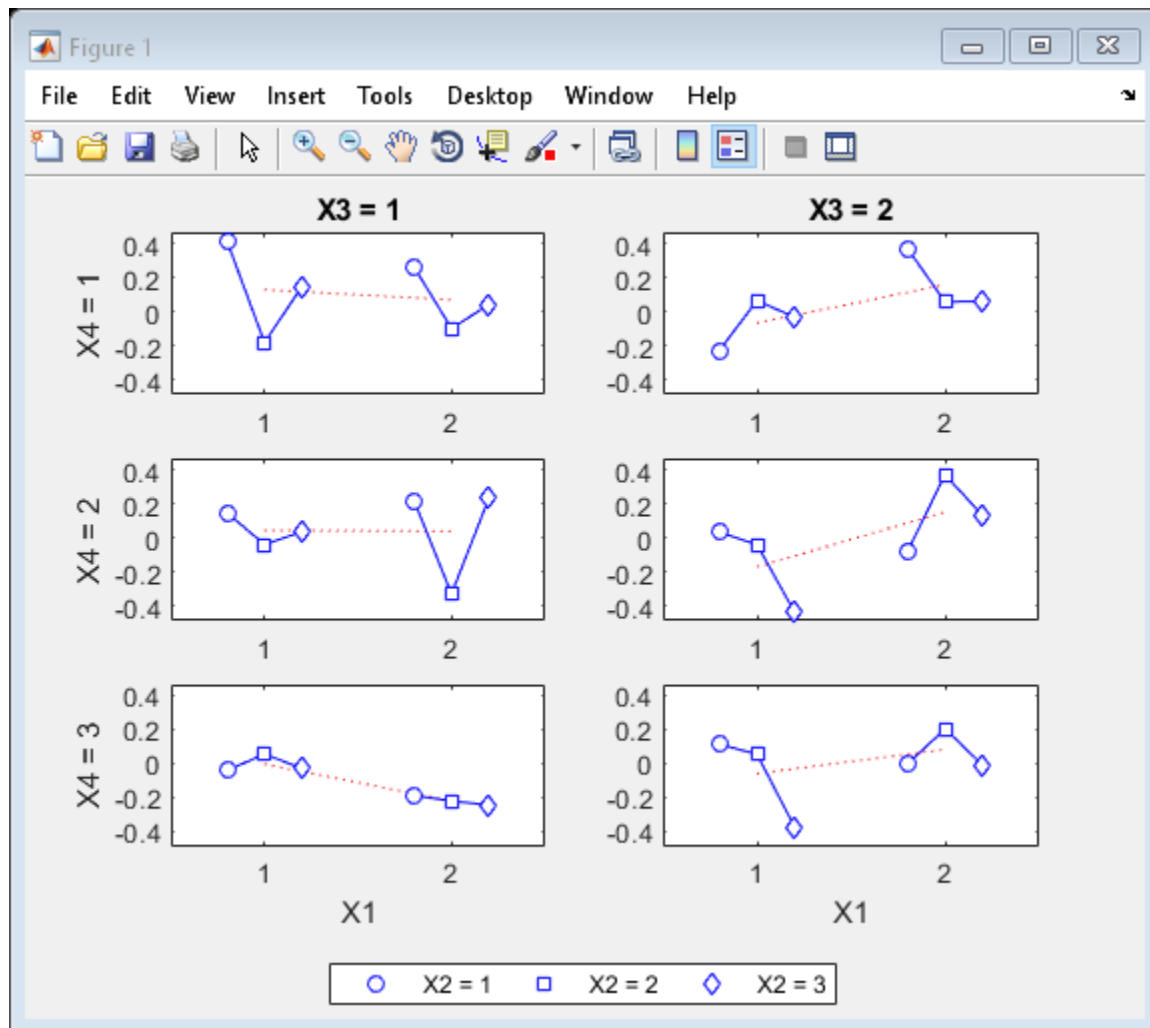
Display a multivari chart for data with two grouping variables.

```
rng default; % For reproducibility
y = randn(100,1); % Randomly generate response
group = [ceil(3*rand(100,1)) ceil(2*rand(100,1))];
multivarichart(y,group)
```



Display a multivari chart for data with four grouping variables.

```
y = randn(1000,1); % Randomly generate response
group = {ceil(2*rand(1000,1)),ceil(3*rand(1000,1)), ...
         ceil(2*rand(1000,1)),ceil(3*rand(1000,1))};
multivarichart(y,group)
```



## See Also

`interactionplot` | `maineffectsplot`

## Topics

“Grouping Variables” on page 2-59

**Introduced in R2006b**

## mvksdensity

Kernel smoothing function estimate for multivariate data

### Syntax

```
f = mvksdensity(x,pts,'Bandwidth',bw)
f = mvksdensity(x,pts,'Bandwidth',bw,Name,Value)
```

### Description

`f = mvksdensity(x,pts,'Bandwidth',bw)` computes a probability density estimate of the sample data in the  $n$ -by- $d$  matrix `x`, evaluated at the points in `pts` using the required name-value pair argument value `bw` for the bandwidth value. The estimation is based on a product Gaussian kernel function.

For univariate or bivariate data, use `ksdensity` instead.

`f = mvksdensity(x,pts,'Bandwidth',bw,Name,Value)` returns any of the previous output arguments, using additional options specified by one or more `Name,Value` pair arguments. For example, you can define the function type that `mvksdensity` evaluates, such as probability density, cumulative probability, or survivor function. You can also assign weights to the input values.

### Examples

#### Estimate Multivariate Kernel Density

Load the Hald cement data.

```
load hald
```

The data measures the heat of hardening for 13 different cement compositions. The predictor matrix `ingredients` contains the percent composition for each of four cement



ingredients. The response matrix `heat` contains the heat of hardening (in cal/g) after 180 days.

Estimate the kernel density for the first three observations in `ingredients`.

```
xi = ingredients(1:3,:);
f = mvksdensity(ingredients,xi,'Bandwidth',0.8);
```

### Estimate Multivariate Kernel Density Using Grids

Load the Hald cement data.

```
load hald
```

The data measures the heat of hardening for 13 different cement compositions. The predictor matrix `ingredients` contains the percent composition for each of four cement ingredients. The response matrix `heat` contains the heat of hardening (in cal/g) after 180 days.

Create a array of points at which to estimate the density. First, define the range and spacing for each variable, using a similar number of points in each dimension.

```
gridx1 = 0:2:22;
gridx2 = 20:5:80;
gridx3 = 0:2:24;
gridx4 = 5:5:65;
```

Next, use `ndgrid` to generate a full grid of points using the defined range and spacing.

```
[x1,x2,x3,x4] = ndgrid(gridx1,gridx2,gridx3,gridx4);
```

Finally, transform and concatenate to create an array that contains the points at which to estimate the density. This array has one column for each variable.

```
x1 = x1(:, :)' ;
x2 = x2(:, :)' ;
x3 = x3(:, :)' ;
x4 = x4(:, :)' ;
xi = [x1(:) x2(:) x3(:) x4(:)];
```

Estimate the density.

```
[f,bw] = mvksdensity(ingredients,xi,...
    'Bandwidth',[4.0579 10.7345 4.4185 11.5466],...
    'Kernel','normpdf');
```

View the size of `xi` and `f` to confirm that `mvksdensity` calculates the density at each point in `xi`.

```
size_xi = size(xi)
size_f = size(f)
```

```
size_xi =
    26364     4
```

```
size_f =
    26364     1
```

## Input Arguments

### **x** — Sample data

numeric matrix

Sample data for which `mvksdensity` returns the probability density estimate, specified as an  $n$ -by- $d$  matrix of numeric values.  $n$  is the number of data points (rows) in `x`, and  $d$  is the number of dimensions (columns).

Data Types: `single` | `double`

### **pts** — Points at which to evaluate `f`

matrix

Points at which to evaluate the probability density estimate `f`, specified as a matrix with the same number of columns as `x`. The returned estimate `f` has the same length as `pts`.

Data Types: `single` | `double`

### **bw** — Value for the bandwidth of the kernel smoothing window

scalar value |  $d$ -element vector

Value for the bandwidth of the kernel-smoothing window, specified as a scalar value or  $d$ -element vector.  $d$  is the number of dimensions (columns) in the sample data  $x$ . If `bw` is a scalar value, it applies to all dimensions.

If you specify `'BoundaryCorrection'` as `'log'` (default) and `'Support'` as either `'positive'` or a two-row matrix, `mvksdensity` converts bounded data to be unbounded by using log transformation. The value of `bw` is on the scale of the transformed values.

Silverman's rule of thumb for the bandwidth is

$$b_i = \sigma_i \left\{ \frac{4}{(d+2)n} \right\}^{1/(d+4)}, \quad i = 1, 2, \dots, d,$$

where  $d$  is the number of dimensions,  $n$  is the number of observations, and  $\sigma_i$  is the standard deviation of the  $i^{\text{th}}$  variate [4].

Example: `'Bandwidth', 0.8`

Data Types: `single` | `double`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'Kernel', 'triangle', 'Function', 'cdf'` specifies that `mvksdensity` estimates the cdf of the sample data using the triangle kernel function.

### **BoundaryCorrection** — Boundary correction method

`'log'` (default) | `'reflection'`

Boundary correction method, specified as the comma-separated pair consisting of `'BoundaryCorrection'` and either `'log'` or `'reflection'`.

Value	Description
'log'	<p><code>mvksdensity</code> converts bounded data to be unbounded by using one of the following transformations. Then, it transforms back to the original bounded scale after density estimation.</p> <ul style="list-style-type: none"> <li>• If you specify 'Support', 'positive', then <code>mvksdensity</code> applies <math>\log(x_j)</math> for each dimension, where <math>x_j</math> is the <math>j</math>th column of the input argument <math>x</math>.</li> <li>• If you specify 'Support' as a two-row matrix consisting of the lower and upper limits for each dimension, then <code>mvksdensity</code> applies <math>\log((x_j - L_j)/(U_j - x_j))</math> for each dimension, where <math>L_j</math> and <math>U_j</math> are the lower and upper limits of the <math>j</math>th dimension, respectively.</li> </ul> <p>The value of <code>bw</code> is on the scale of the transformed values.</p>
'reflection'	<p><code>mvksdensity</code> augments bounded data by adding reflected data near the boundaries, then it returns estimates corresponding to the original support. For details, see “Reflection Method” on page 32-4121.</p>

`mvksdensity` applies boundary correction only when you specify 'Support' as a value other than 'unbounded'.

Example: 'BoundaryCorrection', 'reflection'

Data Types: char

#### Function — Function to estimate

'pdf' (default) | 'cdf' | 'survivor'

Function to estimate, specified as the comma-separated pair consisting of 'Function' and one of the following.

Value	Description
'pdf'	Probability density function
'cdf'	Cumulative distribution function
'survivor'	Survivor function

Example: 'Function', 'cdf'

Data Types: char

**Kernel** — Type of kernel smoother

'normal' (default) | 'box' | 'triangle' | 'epanechnikov' | function handle | function name

Type of kernel smoother, specified as the comma-separated pair consisting of 'Kernel' and one of the following.

Value	Description
'normal'	Normal (Gaussian) kernel
'box'	Box kernel
'triangle'	Triangular kernel
'epanechnikov'	Epanechnikov kernel

You can also specify a custom kernel function as a function handle (for example, @normpdf) or a function name in single quotes (for example, 'normpdf'). This calls the specified function using one argument that is an array of distances between data values and locations where the density is evaluated, normalized by the bandwidth in that dimension. The function must return an array of the same size containing the corresponding values of the kernel function.

mvksdensity applies the same kernel to each dimension.

Example: 'Kernel', 'box'

Data Types: char | function\_handle

**Support** — Support for the density

'unbounded' (default) | 'positive' | 2-by- $d$  matrix

Support for the density, specified as the comma-separated pair consisting of 'support' and one of the following.

Value	Description
'unbounded'	Allow the density to extend over the whole real line
'positive'	Restrict the density to positive values

Value	Description
2-by- $d$ matrix	Specify the finite lower and upper bounds for the support of the density. The first row contains the lower limits and the second row contains the upper limits. Each column contains the limits for one dimension of $x$ .

'Support' can also be a combination of positive, unbounded, and bounded variables specified as `[0 -Inf L ; Inf Inf U]`.

Example: 'Support', 'positive'

Data Types: `single` | `double` | `char`

#### **Weights** — Weights for sample data vector

Weights for sample data, specified as the comma-separated pair consisting of 'Weights' and a vector of length `size(x,1)`, where  $x$  is the sample data.

Example: 'Weights', `xw`

Data Types: `single` | `double`

## Output Arguments

#### **$\hat{f}$** — Estimated function values vector

Estimated function values, returned as a vector of the same length as  $x$ .

## Definitions

### Multivariate Kernel Distribution

A multivariate kernel distribution is a nonparametric representation of the probability density function (pdf) of a random vector. You can use a kernel distribution when a parametric distribution cannot properly describe the data, or when you want to avoid

making assumptions about the distribution of the data. A multivariate kernel distribution is defined by a smoothing function and a bandwidth matrix, which control the smoothness of the resulting density curve.

The multivariate kernel density estimator is the estimated pdf of a random vector. Let  $x = (x_1, x_2, \dots, x_d)'$  be a  $d$ -dimensional random vector with a density function  $f$  and let  $y_i = (y_{i1}, y_{i2}, \dots, y_{id})'$  be a random sample drawn from  $f$  for  $i = 1, 2, \dots, n$ , where  $n$  is the number of random samples. For any real vectors of  $x$ , the multivariate kernel density estimator is given by

$$\hat{f}_H(x) = \frac{1}{n} \sum_{i=1}^n K_H(x - y_i),$$

where  $K_H(x) = |H|^{-1/2} K(H^{-1/2}x)$ ,  $K(\bullet)$  is the kernel smoothing function, and  $H$  is the  $d$ -by- $d$  bandwidth matrix.

`mvksdensity` uses a diagonal bandwidth matrix and a product kernel. That is,  $H^{1/2}$  is a square diagonal matrix with the elements of vector  $(h_1, h_2, \dots, h_d)$  on the main diagonal.

$K(x)$  takes the product form  $K(x) = k(x_1)k(x_2) \cdots k(x_d)$ , where  $k(\bullet)$  is a one-dimensional kernel smoothing function. Then, the multivariate kernel density estimator becomes

$$\hat{f}_H(x) = \frac{1}{n} \sum_{i=1}^n K_H(x - y_i) = \frac{1}{nh_1 h_2 \cdots h_d} \sum_{i=1}^n K\left(\frac{x_1 - y_{i1}}{h_1}, \frac{x_2 - y_{i2}}{h_2}, \dots, \frac{x_d - y_{id}}{h_d}\right) = \frac{1}{nh_1 h_2 \cdots h_d} \sum_{i=1}^n \prod_{j=1}^d k\left(\frac{x_j - y_{ij}}{h_j}\right)$$

The kernel estimator for the cumulative distribution function (cdf), for any real vectors of  $x$ , is given by

$$\hat{F}_H(x) = \int_{-\infty}^{x_1} \int_{-\infty}^{x_2} \cdots \int_{-\infty}^{x_d} \hat{f}_H(t) dt_d \cdots dt_2 dt_1 = \frac{1}{n} \sum_{i=1}^n \prod_{j=1}^d G\left(\frac{x_j - y_{ij}}{h_j}\right),$$

where  $G(x_j) = \int_{-\infty}^{x_j} k(t_j) dt_j$ .

## Reflection Method

The reflection method is a boundary correction method that accurately finds kernel density estimators when a random variable has bounded support. If you specify 'BoundaryCorrection', 'reflection', `mvksdensity` uses the reflection method.

If you additionally specify 'Support' as a two-row matrix consisting of the lower and upper limits for each dimension, then `mvksdensity` finds the kernel estimator as follows.

- If 'Function' is 'pdf', then the kernel density estimator is

$$\hat{f}_H(x) = \frac{1}{nh_1 h_2 \cdots h_d} \sum_{i=1}^n \prod_{j=1}^d \left[ k\left(\frac{x_j - y_{ij}^-}{h_j}\right) + k\left(\frac{x_j - y_{ij}}{h_j}\right) + k\left(\frac{x_j - y_{ij}^+}{h_j}\right) \right] \text{ for } L_j \leq x_j \leq U_j,$$

where  $y_{ij}^- = 2L_j - y_{ij}$ ,  $y_{ij}^+ = 2U_j - y_{ij}$ , and  $y_{ij}$  is the  $j$ th element of the  $i$ th sample data corresponding to  $x(i, j)$  of the input argument  $x$ .  $L_j$  and  $U_j$  are the lower and upper limits of the  $j$ th dimension, respectively.

- If 'Function' is 'cdf', then the kernel estimator for cdf is

$$\hat{F}_H(x) = \frac{1}{n} \sum_{i=1}^n \prod_{j=1}^d \left[ G\left(\frac{x_j - y_{ij}^-}{h_j}\right) + G\left(\frac{x_j - y_{ij}}{h_j}\right) + G\left(\frac{x_j - y_{ij}^+}{h_j}\right) - G\left(\frac{L_j - y_{ij}^-}{h_j}\right) - G\left(\frac{L_j - y_{ij}}{h_j}\right) - G\left(\frac{L_j - y_{ij}^+}{h_j}\right) \right]$$

- To obtain a kernel estimator for a survivor function (when 'Function' is

'survivor'), `mvksdensity` uses both  $\hat{f}_H(x)$  and  $\hat{F}_H(x)$ .

If you additionally specify 'Support' as 'positive' or a matrix including  $[0 \text{ } \text{inf}]$ , then `mvksdensity` finds the kernel density estimator by replacing  $[L_j \text{ } U_j]$  with  $[0 \text{ } \text{inf}]$  in the above equations.

## References

- [1] Bowman, A. W., and A. Azzalini. *Applied Smoothing Techniques for Data Analysis*. New York: Oxford University Press Inc., 1997.
- [2] Hill, P. D. "Kernel estimation of a distribution function." *Communications in Statistics – Theory and Methods*. Vol. 14, Issue 3, 1985, pp. 605-620.
- [3] Jones, M. C. "Simple boundary correction for kernel density estimation." *Statistics and Computing*. Vol. 3, Issue 3, 1993, pp. 135-146.



- [4] Silverman, B. W. *Density Estimation for Statistics and Data Analysis*. Chapman & Hall/CRC, 1986.
- [5] Scott, D. W. *Multivariate Density Estimation: Theory, Practice, and Visualization*. John Wiley & Sons, 2015.

## See Also

ksdensity

## Topics

“Working with Probability Distributions” on page 5-3

“Nonparametric and Empirical Probability Distributions” on page 5-37

“Supported Distributions” on page 5-17

**Introduced in R2016a**

## mvncdf

Multivariate normal cumulative distribution function

### Syntax

```
y = mvncdf(X)
y = mvncdf(X, mu, SIGMA)
y = mvncdf(xl, xu, mu, SIGMA)
[y, err] = mvncdf(...)
[...] = mvncdf(..., options)
```

### Description

`y = mvncdf(X)` returns the cumulative probability of the multivariate normal distribution with zero mean and identity covariance matrix, evaluated at each row of `X`. Rows of the  $n$ -by- $d$  matrix `X` correspond to observations or points, and columns correspond to variables or coordinates. `y` is an  $n$ -by-1 vector.

`y = mvncdf(X, mu, SIGMA)` returns the cumulative probability of the multivariate normal distribution with mean `mu` and covariance `SIGMA`, evaluated at each row of `X`. `mu` is a 1-by- $d$  vector, and `SIGMA` is a  $d$ -by- $d$  symmetric, positive definite matrix. `mu` can also be a scalar value, which `mvncdf` replicates to match the size of `X`. If the covariance matrix is diagonal, containing variances along the diagonal and zero covariances off the diagonal, `SIGMA` may also be specified as a 1-by- $d$  vector containing just the diagonal. Pass in the empty matrix `[]` for `mu` to use as its default value when you want to only specify `SIGMA`.

The multivariate normal cumulative probability at `X` is defined as the probability that a random vector `V`, distributed as multivariate normal, will fall within the semi-infinite rectangle with upper limits defined by `X`, for example,  $\Pr\{V(1) \leq X(1), V(2) \leq X(2), \dots, V(d) \leq X(d)\}$ .

`y = mvncdf(xl, xu, mu, SIGMA)` returns the multivariate normal cumulative probability evaluated over the rectangle with lower and upper limits defined by `xl` and `xu`, respectively.

`[y,err] = mvncdf(...)` returns an estimate of the error in  $y$ . For bivariate and trivariate distributions, `mvncdf` uses adaptive quadrature on a transformation of the  $t$  density, based on methods developed by Drezner and Wesolowsky and by Genz, as described in the references. The default absolute error tolerance for these cases is  $1e-8$ . For four or more dimensions, `mvncdf` uses a quasi-Monte Carlo integration algorithm based on methods developed by Genz and Bretz, as described in the references. The default absolute error tolerance for these cases is  $1e-4$ .

`[...] = mvncdf(...,options)` specifies control parameters for the numerical integration used to compute  $y$ . This argument can be created by a call to `statset`. Choices of `statset` parameters:

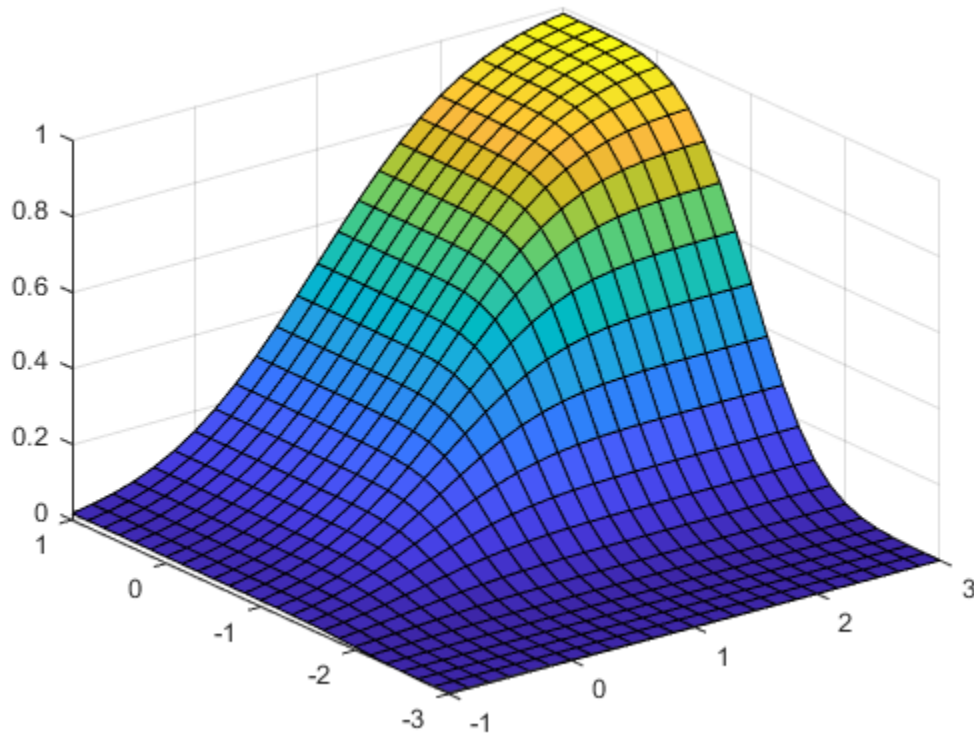
- `'TolFun'` — Maximum absolute error tolerance. Default is  $1e-8$  when  $d < 4$ , or  $1e-4$  when  $d \geq 4$ .
- `'MaxFunEvals'` — Maximum number of integrand evaluations allowed when  $d \geq 4$ . Default is  $1e7$ . `'MaxFunEvals'` is ignored when  $d < 4$ .
- `'Display'` — Level of display output. Choices are `'off'` (the default), `'iter'`, and `'final'`. `'Display'` is ignored when  $d < 4$ .

## Examples

### Compute the Multivariate Normal cdf

Compute and plot the cdf of a multivariate normal distribution with parameters  $\mu = [1 \ -1]$  and  $\text{SIGMA} = [.9 \ .4; \ .4 \ .3]$ .

```
mu = [1 -1];
SIGMA = [.9 .4; .4 .3];
figure;
[X1,X2] = meshgrid(linspace(-1,3,25)',linspace(-3,1,25)');
X = [X1(:) X2(:)];
p = mvncdf(X,mu,SIGMA);
surf(X1,X2,reshape(p,25,25));
```



## References

- [1] Drezner, Z. “Computation of the Trivariate Normal Integral.” *Mathematics of Computation*. Vol. 63, 1994, pp. 289–294.
- [2] Drezner, Z., and G. O. Wesolowsky. “On the Computation of the Bivariate Normal Integral.” *Journal of Statistical Computation and Simulation*. Vol. 35, 1989, pp. 101–107.
- [3] Genz, A. “Numerical Computation of Rectangular Bivariate and Trivariate Normal and t Probabilities.” *Statistics and Computing*. Vol. 14, No. 3, 2004, pp. 251–260.

- [4] Genz, A., and F. Bretz. “Numerical Computation of Multivariate t Probabilities with Application to Power Calculation of Multiple Contrasts.” *Journal of Statistical Computation and Simulation*. Vol. 63, 1999, pp. 361–378.
- [5] Genz, A., and F. Bretz. “Comparison of Methods for the Computation of Multivariate t Probabilities.” *Journal of Computational and Graphical Statistics*. Vol. 11, No. 4, 2002, pp. 950–971.

## See Also

mvnpdf | mvnrnd

## Topics

“Multivariate Normal Distribution” on page B-106

**Introduced in R2006a**

## mvnpdf

Multivariate normal probability density function

### Syntax

```
y = mvnpdf(X)
y = mvnpdf(X, MU)
y = mvnpdf(X, MU, SIGMA)
```

### Description

`y = mvnpdf(X)` returns the  $n$ -by-1 vector `y`, containing the probability density of the multivariate normal distribution with zero mean and identity covariance matrix, evaluated at each row of the  $n$ -by- $d$  matrix `X`. Rows of `X` correspond to observations and columns correspond to variables or coordinates.

`y = mvnpdf(X, MU)` returns the density of the multivariate normal distribution with mean `mu` and identity covariance matrix, evaluated at each row of `X`. `MU` is a 1-by- $d$  vector, or an  $n$ -by- $d$  matrix. If `MU` is a matrix, the density is evaluated for each row of `X` with the corresponding row of `MU`. `MU` can also be a scalar value, which `mvnpdf` replicates to match the size of `X`.

`y = mvnpdf(X, MU, SIGMA)` returns the density of the multivariate normal distribution with mean `MU` and covariance `SIGMA`, evaluated at each row of `X`. `SIGMA` is a  $d$ -by- $d$  matrix, or a  $d$ -by- $d$ -by- $n$  array, in which case the density is evaluated for each row of `X` with the corresponding page of `SIGMA`, i.e., `mvnpdf` computes `y(i)` using `X(i, :)` and `SIGMA(:, :, i)`. If the covariance matrix is diagonal, containing variances along the diagonal and zero covariances off the diagonal, `SIGMA` may also be specified as a 1-by- $d$  vector or a 1-by- $d$ -by- $n$  array, containing just the diagonal. Specify `[]` for `MU` to use its default value when you want to specify only `SIGMA`.

If `X` is a 1-by- $d$  vector, `mvnpdf` replicates it to match the leading dimension of `mu` or the trailing dimension of `SIGMA`.

## Examples

```
mu = [1 -1];  
SIGMA = [.9 .4; .4 .3];  
X = mvnrnd(mu, SIGMA, 10);  
p = mvnpdf(X, mu, SIGMA);
```

## See Also

`mvncdf` | `mvnrnd` | `normpdf`

## Topics

“Multivariate Normal Distribution” on page B-106

**Introduced before R2006a**

## mvregress

Multivariate linear regression

### Syntax

```
beta = mvregress(X, Y)
beta = mvregress(X, Y, Name, Value)
[beta, Sigma] = mvregress(____)
[beta, Sigma, E, CovB, logL] = mvregress(____)
```

### Description

`beta = mvregress(X, Y)` returns the estimated coefficients for a multivariate normal regression on page 32-4145 of the  $d$ -dimensional responses in  $Y$  on the design matrices in  $X$ .

`beta = mvregress(X, Y, Name, Value)` returns the estimated coefficients using additional options specified by one or more name-value pair arguments. For example, you can specify the estimation algorithm, initial estimate values, or maximum number of iterations for the regression.

`[beta, Sigma] = mvregress(____)` also returns the estimated  $d$ -by- $d$  variance-covariance matrix of  $Y$ , using any of the input arguments from the previous syntaxes.

`[beta, Sigma, E, CovB, logL] = mvregress(____)` also returns a matrix of residuals  $E$ , estimated variance-covariance matrix of the regression coefficients  $CovB$ , and the value of the log likelihood objective function after the last iteration  $logL$ .

### Examples



## Multivariate Regression Model for Panel Data with Different Intercepts

Fit a multivariate regression model to panel data, assuming different intercepts and common slopes.

Load the sample data.

```
load('flu')
```

The dataset array `flu` contains national CDC flu estimates, and nine separate regional estimates based on Google® query data.

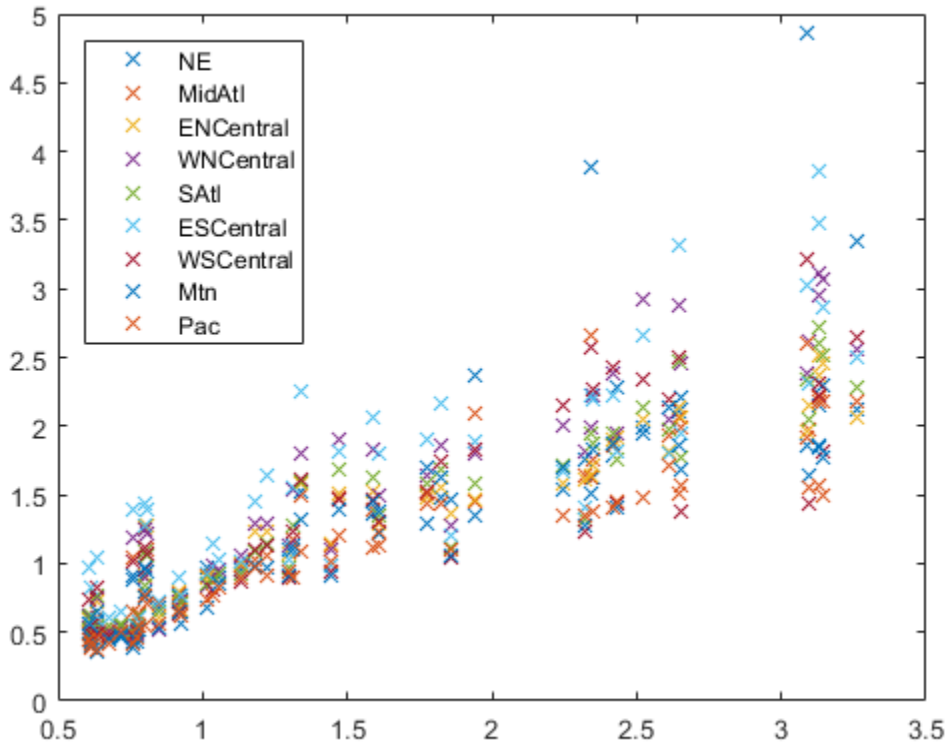
Extract the response and predictor data.

```
Y = double(flu(:,2:end-1));  
[n,d] = size(Y);  
x = flu.WtdILI;
```

The responses in  $Y$  are the nine regional flu estimates. Observations exist for every week over a one-year period, so  $n = 52$ . The dimension of the responses corresponds to the regions, so  $d = 9$ . The predictors in  $x$  are the weekly national flu estimates.

Plot the flu data, grouped by region.

```
figure;  
regions = flu.Properties.VarNames(2:end-1);  
plot(x,Y,'x')  
legend(regions,'Location','NorthWest')
```



Fit the multivariate regression model

$$y_{ij} = \alpha_j + \beta_j x_{ij} + \epsilon_{ij},$$

where  $i = 1, \dots, n$  and  $j = 1, \dots, d$ , with between-region concurrent correlation  $COV(\epsilon_{ij}, \epsilon_{ij}) = \sigma_{jj}$ .

There are  $K = 10$  regression coefficients to estimate: nine intercept terms and a common slope. The input argument `X` should be an  $n$ -element cell array of  $d$ -by- $K$  design matrices.

```
X = cell(n,1);
for i = 1:n
    X{i} = [eye(d) repmat(x(i),d,1)];
```

```
end
[beta, Sigma] = mvregress(X, Y);
```

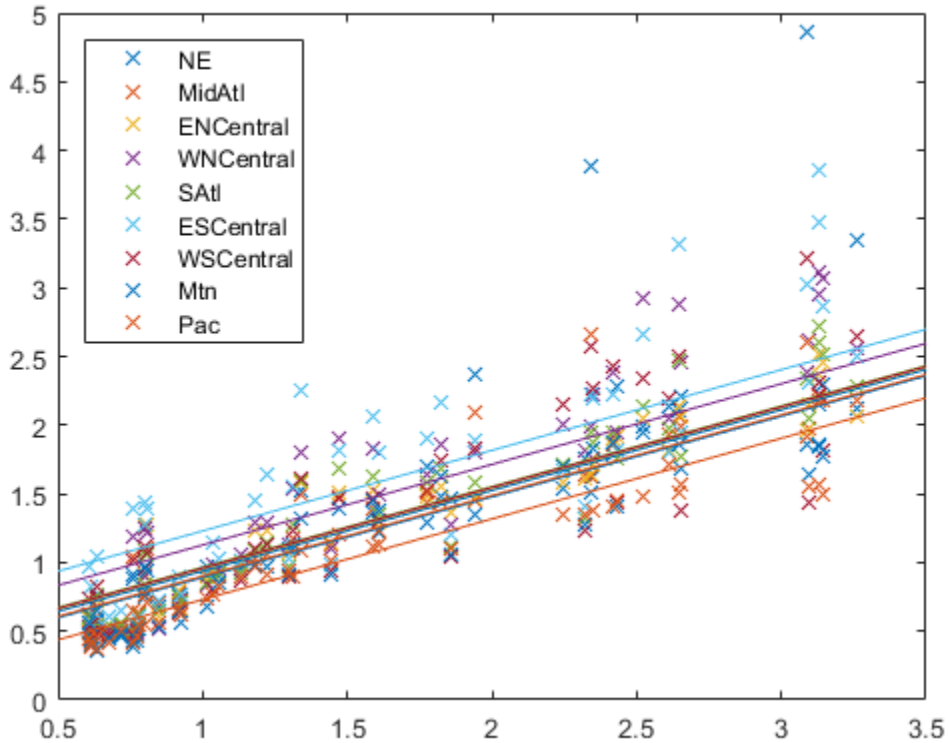
beta contains estimates of the  $K$ -dimensional coefficient vector  $(\alpha_1, \alpha_2, \dots, \alpha_9, \beta_1, \beta_2, \dots, \beta_9)'$ .

Sigma contains estimates of the  $d$ -by- $d$  variance-covariance matrix  $(\sigma_{ij})_{d \times d}$ ,  $i, j = 1, \dots, d$  for the between-region concurrent correlations.

Plot the fitted regression model.

```
B = [beta(1:d)'; repmat(beta(end), 1, d)];
xx = linspace(.5, 3.5)';
fits = [ones(size(xx)), xx]*B;

figure;
h = plot(x, Y, 'x', xx, fits, '-');
for i = 1:d
    set(h(d+i), 'color', get(h(i), 'color'));
end
legend(regions, 'Location', 'NorthWest');
```



The plot shows that each regression line has a different intercept but the same slope. Upon visual inspection, some regression lines appear to fit the data better than others.

### Multivariate Regression for Panel Data with Different Slopes

Fit a multivariate regression model to panel data using least squares, assuming different intercepts and slopes.

Load the sample data.

```
load('flu');
```

The dataset array `flu` contains national CDC flu estimates, and nine separate regional estimates based on Google® queries.

Extract the response and predictor data.

```
Y = double(flu(:,2:end-1));
[n,d] = size(Y);
x = flu.WtdILLI;
```

The responses in `Y` are the nine regional flu estimates. Observations exist for every week over a one-year period, so  $n = 52$ . The dimension of the responses corresponds to the regions, so  $d = 9$ . The predictors in `x` are the weekly national flu estimates.

Fit the multivariate regression model

$$y_{ij} = \alpha_j + \beta_j x_{ij} + \epsilon_{ij},$$

where  $i = 1, \dots, n$  and  $j = 1, \dots, d$ , with between-region concurrent correlation  $COV(\epsilon_{ij}, \epsilon_{ij}) = \sigma_{jj}$ .

There are  $K = 18$  regression coefficients to estimate: nine intercept terms, and nine slope terms. `X` is an  $n$ -element cell array of  $d$ -by- $K$  design matrices.

```
X = cell(n,1);
for i = 1:n
    X{i} = [eye(d) x(i)*eye(d)];
end
[beta,Sigma] = mvregress(X,Y,'algorithm','cwls');
```

`beta` contains estimates of the  $K$ -dimensional coefficient vector

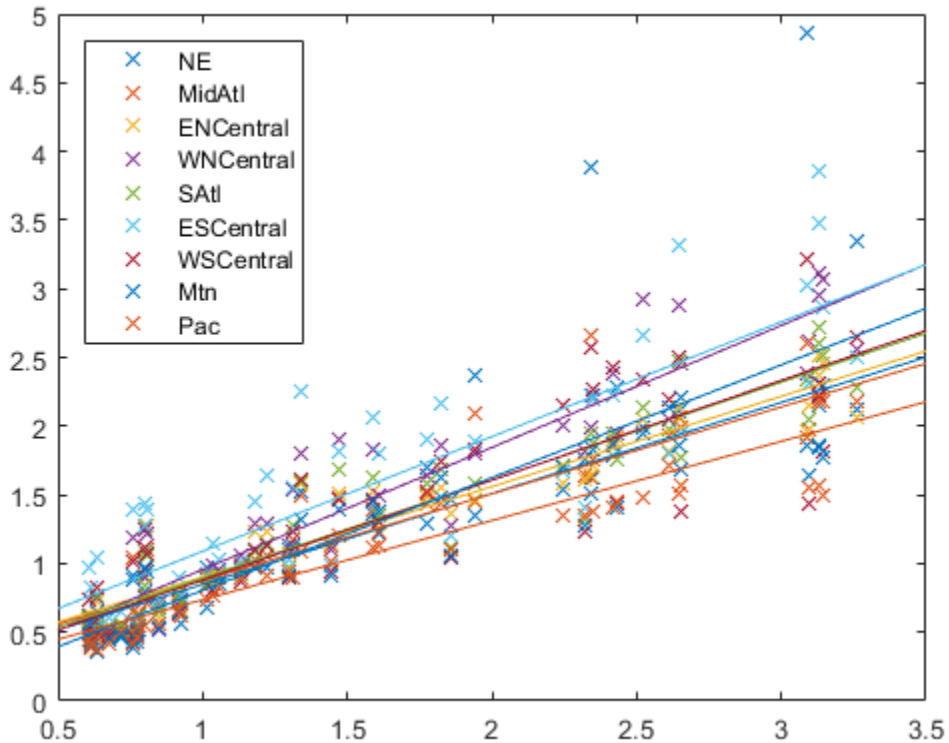
$$(\alpha_1, \alpha_2, \dots, \alpha_9, \beta_1, \beta_2, \dots, \beta_9)'$$

Plot the fitted regression model.

```
B = [beta(1:d)';beta(d+1:end)'];
xx = linspace(.5,3.5)';
fits = [ones(size(xx)),xx]*B;

figure;
h = plot(x,Y,'x',xx,fits,'-');
for i = 1:d
    set(h(d+i),'color',get(h(i),'color'));
end
```

```
regions = flu.Properties.VarNames(2:end-1);
legend(regions, 'Location', 'NorthWest');
```



The plot shows that each regression line has a different intercept and slope.

### Multivariate Regression With a Single Design Matrix

Fit a multivariate regression model using a single  $n$ -by- $P$  design matrix for all response dimensions.

Load the sample data.

```
load('flu');
```

The dataset array `flu` contains national CDC flu estimates, and nine separate regional estimates based on Google® queries.

Extract the response and predictor data.

```
Y = double(flu(:,2:end-1));
[n,d] = size(Y);
x = flu.WtdILI;
```

The responses in  $Y$  are the nine regional flu estimates. Observations exist for every week over a one-year period, so  $n = 52$ . The dimension of the responses corresponds to the regions, so  $d = 9$ . The predictors in  $x$  are the weekly national flu estimates.

Create an  $n$ -by- $P$  design matrix  $X$ . Add a column of ones to include a constant term in the regression.

```
X = [ones(size(x)), x];
```

Fit the multivariate regression model

$$y_{ij} = \alpha_j + \beta_j x_{ij} + \epsilon_{ij},$$

where  $i = 1, \dots, n$  and  $j = 1, \dots, d$ , with between-region concurrent correlation  $COV(\epsilon_{ij}, \epsilon_{ij}) = \sigma_{jj}$ .

There are 18 regression coefficients to estimate: nine intercept terms, and nine slope terms.

```
[beta, Sigma, E, CovB, logL] = mvregress(X, Y);
```

`beta` contains estimates of the  $P$ -by- $d$  coefficient matrix. `Sigma` contains estimates of the  $d$ -by- $d$  variance-covariance matrix for the between-region concurrent correlations. `E` is a matrix of the residuals. `CovB` is the estimated variance-covariance matrix of the regression coefficients. `logL` is the value of the log likelihood objective function after the last iteration.

Plot the fitted regression model.

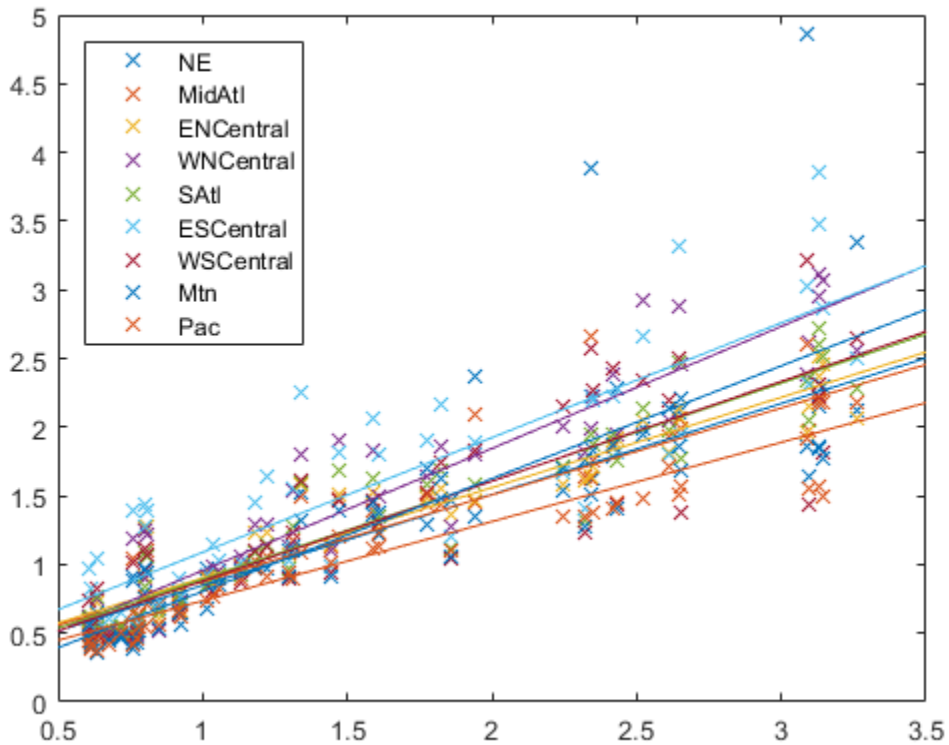
```
B = beta;
xx = linspace(.5, 3.5)';
fits = [ones(size(xx)), xx]*B;
```

```

figure;
h = plot(x,Y,'x', xx,fits,'-');
for i = 1:d
    set(h(d+i),'color',get(h(i),'color'));
end

regions = flu.Properties.VarNames(2:end-1);
legend(regions,'Location','NorthWest');

```



The plot shows that each regression line has a different intercept and slope.

- “Set Up Multivariate Regression Problems” on page 15-16



- “Multivariate General Linear Model” on page 15-29
- “Fixed Effects Panel Model with Concurrent Correlation” on page 15-35
- “Longitudinal Analysis” on page 15-43

## Input Arguments

### **x** — Design matrices

matrix | cell array of matrices

Design matrices for the multivariate regression, specified as a matrix or cell array of matrices.  $n$  is the number of observations in the data,  $K$  is the number of regression coefficients to estimate,  $p$  is the number of predictor variables, and  $d$  is the number of dimensions in the response variable matrix  $Y$ .

- If  $d = 1$ , then specify  $X$  as a single  $n$ -by- $K$  design matrix.
- If  $d > 1$  and all  $d$  dimensions have the same design matrix, then you can specify  $X$  as a single  $n$ -by- $p$  design matrix (not in a cell array).
- If  $d > 1$  and all  $n$  observations have the same design matrix, then you can specify  $X$  as a cell array containing a single  $d$ -by- $K$  design matrix.
- If  $d > 1$  and all  $n$  observations do not have the same design matrix, then specify  $X$  as a cell array of length  $n$  containing  $d$ -by- $K$  design matrices.

To include a constant term in the regression model, each design matrix should contain a column of ones.

`mvregress` treats NaN values in  $X$  as missing values, and ignores rows in  $X$  with missing values.

Data Types: `single` | `double` | `cell`

### **y** — Response variables

matrix

Response variables, specified as an  $n$ -by- $d$  matrix.  $n$  is the number of observations in the data, and  $d$  is the number of dimensions in the response. When  $d = 1$ , `mvregress` treats the values in  $Y$  like  $n$  independent response values.

`mvregress` treats NaN values in  $Y$  as missing values, and handles them according to the estimation algorithm specified using the name-value pair argument `algorithm`.

Data Types: `single` | `double`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'algorithm', 'cwls', 'covar0', C` specifies covariance-weighted least squares estimation using the covariance matrix `C`.

### **algorithm** — Estimation algorithm

`'mvn'` | `'ecm'` | `'cwls'`

Estimation algorithm, specified as the comma-separated pair consisting of `'algorithm'` and one of the following.

<code>'mvn'</code>	Ordinary multivariate normal maximum likelihood estimation.
<code>'ecm'</code>	Maximum likelihood estimation via the ECM algorithm.
<code>'cwls'</code>	Covariance-weighted least squares estimation.

The default algorithm depends on the presence of missing data.

- For complete data, the default is `'mvn'`.
- If there are any missing responses (indicated by `NaN`), the default is `'ecm'`, provided the sample size is sufficient to estimate all parameters. Otherwise, the default algorithm is `'cwls'`.

---

**Note** If `algorithm` has the value `'mvn'`, then `mvregress` removes observations with missing response values before estimation.

---

Example: `'algorithm', 'ecm'`

### **beta0** — Initial estimates for regression coefficients

vector

Initial estimates for the regression coefficients, specified as the comma-separated pair consisting of 'beta0' and a vector with  $K$  elements. The default value is a vector of 0s.

The beta0 argument is not used if the estimation algorithm is 'mvn'.

**covar0** — Initial estimate for variance-covariance matrix  
matrix

Initial estimate for the variance-covariance matrix, Sigma, specified as the comma-separated pair consisting of 'covar0' and a symmetric, positive definite,  $d$ -by- $d$  matrix. The default value is the identity matrix.

If the estimation algorithm is 'cwls', then mvregress uses covar0 as the weighting matrix at each iteration, without changing it.

**covtype** — Type of variance-covariance matrix  
'full' (default) | 'diagonal'

Type of variance-covariance matrix to estimate for  $Y$ , specified as the comma-separated pair consisting of 'covtype' and one of the following.

'full'	Estimate all $d(d + 1)/2$ variance-covariance elements.
'diagonal'	Estimate only the $d$ diagonal elements of the variance-covariance matrix.

Example: 'covtype', 'diagonal'

**maxiter** — Maximum number of iterations  
100 (default) | positive integer

Maximum number of iterations for the estimation algorithm, specified as the comma-separated pair consisting of 'maxiter' and a positive integer.

Iterations continue until estimates are within the convergence tolerances tolbeta and tolobj, or the maximum number of iterations specified by maxiter is reached. If both tolbeta and tolobj are 0, then mvregress performs maxiter iterations with no convergence tests.

Example: 'maxiter', 50

**outputfcn** — Function to evaluate each iteration  
function handle

Function to evaluate at each iteration, specified as the comma-separated pair consisting of 'outputfcn' and a function handle. The function must return a logical `true` or `false`. At each iteration, `mvregress` evaluates the function. If the result is `true`, iterations stop. Otherwise, iterations continue. For example, you could specify a function that plots or displays current iteration results, and returns `true` if you close the figure.

The function must accept three input arguments, in this order:

- Vector of current coefficient estimates
- Structure containing these three fields:

<code>Covar</code>	Current value of the variance-covariance matrix
<code>iteration</code>	Current iteration number
<code>fval</code>	Current value of the loglikelihood objective function
- Text that takes these three values:

<code>'init'</code>	When the function is called during initialization
<code>'iter'</code>	When the function is called after an iteration
<code>'done'</code>	When the function is called after completion

### **tolbeta** — Convergence tolerance for regression coefficients

`sqrt(eps)` (default) | positive scalar value

Convergence tolerance for regression coefficients, specified as the comma-separated pair consisting of 'tolbeta' and a positive scalar value.

Let  $\mathbf{b}^t$  denote the estimate of the coefficient vector at iteration  $t$ , and  $\tau_\beta$  be the tolerance specified by `tolbeta`. The convergence criterion for regression coefficient estimation is

$$\|\mathbf{b}^t - \mathbf{b}^{t-1}\| < \tau_\beta \sqrt{K} (1 + \|\mathbf{b}^t\|),$$

where  $K$  is the length of  $\mathbf{b}^t$  and  $\|\mathbf{v}\|$  is the norm of a vector  $\mathbf{v}$ .

Iterations continue until estimates are within the convergence tolerances `tolbeta` and `tolobj`, or the maximum number of iterations specified by `maxiter` is reached. If both `tolbeta` and `tolobj` are 0, then `mvregress` performs `maxiter` iterations with no convergence tests.

Example: `'tolbeta', 1e-5`

**tolobj** — Convergence tolerance for loglikelihood objective function

`eps^(3/4)` (default) | positive scalar value

Convergence tolerance for the loglikelihood objective function, specified as the comma-separated pair consisting of `'tolobj'` and a positive scalar value.

Let  $L^t$  denote the value of the loglikelihood objective function at iteration  $t$ , and  $\tau_\ell$  be the tolerance specified by `tolobj`. The convergence criterion for the objective function is

$$|L^t - L^{t-1}| < \tau_\ell (1 + |L^t|).$$

Iterations continue until estimates are within the convergence tolerances `tolbeta` and `tolobj`, or the maximum number of iterations specified by `maxiter` is reached. If both `tolbeta` and `tolobj` are 0, then `mvregress` performs `maxiter` iterations with no convergence tests.

Example: `'tolobj', 1e-5`

**varformat** — Format for parameter estimate variance-covariance matrix

`'beta'` (default) | `'full'`

Format for the parameter estimate variance-covariance matrix, `CovB`, specified as the comma-separated pair consisting of `'varformat'` and one of the following.

<code>'beta'</code>	Return the variance-covariance matrix for only the regression coefficient estimates, <code>beta</code> .
<code>'full'</code>	Return the variance-covariance matrix for both the regression coefficient estimates, <code>beta</code> , and the variance-covariance matrix estimate, <code>Sigma</code> .

Example: `'varformat', 'full'`

**vartype** — Type of variance-covariance matrix for parameter estimates

`'hessian'` (default) | `'fisher'`

Type of variance-covariance matrix for parameter estimates, specified as the comma-separated pair consisting of `'vartype'` and either `'hessian'` or `'fisher'`.

- If the value is `'hessian'`, then `mvregress` uses the Hessian, or observed information, matrix to compute `CovB`.

- If the value is 'fisher', then `mvregress` uses the complete-data Fisher, or expected information, matrix to compute `CovB`.

The 'hessian' method takes into account the increase uncertainties due to missing data, while the 'fisher' method does not.

Example: 'vartype', 'fisher'

## Output Arguments

### **beta** — Estimated regression coefficients

column vector | matrix

Estimated regression coefficients, returned as a column vector or matrix.

- If you specify `X` as a single  $n$ -by- $K$  design matrix, then `mvregress` returns `beta` as a column vector of length  $K$ . For example, if `X` is a 20-by-5 design matrix, then `beta` is a 5-by-1 column vector.
- If you specify `X` as a cell array containing one or more  $d$ -by- $K$  design matrices, then `mvregress` returns `beta` as a column vector of length  $K$ . For example, if `X` is a cell array containing 2-by-10 design matrices, then `beta` is a 10-by-1 column vector.
- If you specify `X` as a single  $n$ -by- $p$  design matrix (not in a cell array), and `Y` has dimension  $d > 1$ , then `mvregress` returns `beta` as a  $p$ -by- $d$  matrix. For example, if `X` is a 20-by-5 design matrix, and `Y` has two dimensions such that  $d = 2$ , then `beta` is a 5-by-2 matrix, and the fitted `Y` values are `X * beta`.

### **sigma** — Estimated variance-covariance matrix

square matrix

Estimated variance-covariance matrix for the responses in `Y`, returned as a  $d$ -by- $d$  square matrix.

---

**Note** The estimated variance-covariance matrix, `Sigma`, is not the sample covariance matrix of the residual matrix, `E`.

---

### **E** — Residuals

matrix

Residuals for the fitted regression model, returned as an  $n$ -by- $d$  matrix.

If `algorithm` has the value 'ecm' or 'cwlsl', then `mvregress` computes the residual values corresponding to missing values in  $Y$  as the difference between the conditionally imputed values on page 32-4146 and the fitted values.

---

**Note** If `algorithm` has the value 'mvn', then `mvregress` removes observations with missing response values before estimation.

---

### **CovB** — Parameter estimate variance-covariance matrix

square matrix

Parameter estimate variance-covariance matrix, returned as a square matrix.

- If `varformat` has the value 'beta' (default), then `CovB` is the estimated variance-covariance matrix of the coefficient estimates in `beta`.
- If `varformat` has the value 'full', then `CovB` is the estimated variance-covariance matrix of the combined estimates in `beta` and `Sigma`.

### **logL** — Loglikelihood objective function value

scalar value

Loglikelihood objective function value after the last iteration, returned as a scalar value.

## Definitions

### Multivariate Normal Regression

Multivariate normal regression is the regression of a  $d$ -dimensional response on a design matrix of predictor variables, with normally distributed errors. The errors can be heteroscedastic and correlated.

The model is

$$\mathbf{y}_i = \mathbf{X}_i\boldsymbol{\beta} + \mathbf{e}_i, \quad i = 1, \dots, n,$$

where

- $\mathbf{y}_i$  is a  $d$ -dimensional vector of responses.
- $\mathbf{X}_i$  is a design matrix of predictor variables.
- $\beta$  is vector or matrix of regression coefficients.
- $\mathbf{e}_i$  is a  $d$ -dimensional vector of error terms, with multivariate normal distribution  $\mathbf{e}_i \sim MVN_d(\mathbf{0}, \Sigma)$ .

## Conditionally Imputed Values

The expectation/conditional maximization ('ecm') and covariance-weighted least squares ('cwlsl') estimation algorithms include imputation of missing response values.

Let  $\tilde{\mathbf{y}}$  denote missing observations. The conditionally imputed values are the expected value of the missing observation given the observed data,  $E(\tilde{\mathbf{y}} | \mathbf{y})$ .

The joint distribution of the missing and observed responses is a multivariate normal distribution,

$$\begin{pmatrix} \tilde{\mathbf{y}} \\ \mathbf{y} \end{pmatrix} \sim MVN \left\{ \begin{pmatrix} \tilde{\mathbf{X}}\beta \\ \mathbf{X}\beta \end{pmatrix}, \begin{pmatrix} \Sigma_{\tilde{\mathbf{y}}} & \Sigma_{\tilde{\mathbf{y}}\mathbf{y}} \\ \Sigma_{\mathbf{y}\tilde{\mathbf{y}}} & \Sigma_{\mathbf{y}} \end{pmatrix} \right\}.$$

Using properties of the multivariate normal distribution, the imputed conditional expectation is given by

$$E(\tilde{\mathbf{y}} | \mathbf{y}) = \tilde{\mathbf{X}}\beta + \Sigma_{\tilde{\mathbf{y}}\mathbf{y}}\Sigma_{\mathbf{y}}^{-1}(\mathbf{y} - \mathbf{X}\beta).$$

---

**Note** `mvregress` only imputes missing response values. Observations with missing values in the design matrix are removed.

---

## References

- [1] Little, Roderick J. A., and Donald B. Rubin. *Statistical Analysis with Missing Data*. 2nd ed., Hoboken, NJ: John Wiley & Sons, Inc., 2002.



- [2] Meng, Xiao-Li, and Donald B. Rubin. “Maximum Likelihood Estimation via the ECM Algorithm.” *Biometrika*. Vol. 80, No. 2, 1993, pp. 267–278.
- [3] Sexton, Joe, and A. R. Swensen. “ECM Algorithms that Converge at the Rate of EM.” *Biometrika*. Vol. 87, No. 3, 2000, pp. 651–662.
- [4] Dempster, A. P., N. M. Laird, and D. B. Rubin. “Maximum Likelihood from Incomplete Data via the EM Algorithm.” *Journal of the Royal Statistical Society. Series B*, Vol. 39, No. 1, 1977, pp. 1–37.

## See Also

manova1 | mvregresslike

## Topics

- “Set Up Multivariate Regression Problems” on page 15-16
- “Multivariate General Linear Model” on page 15-29
- “Fixed Effects Panel Model with Concurrent Correlation” on page 15-35
- “Longitudinal Analysis” on page 15-43
- “Multivariate Linear Regression” on page 15-4
- “Estimation of Multivariate Regression Models” on page 15-7

**Introduced in R2006b**

## mvregresslike

Negative log-likelihood for multivariate regression

### Syntax

```
nlogL = mvregresslike(X, Y, b, SIGMA, alg)
[nlogL, COVB] = mvregresslike(...)
[nlogL, COVB] = mvregresslike(..., type, format)
```

### Description

`nlogL = mvregresslike(X, Y, b, SIGMA, alg)` computes the negative log-likelihood `nlogL` for a multivariate regression of the  $d$ -dimensional multivariate observations in the  $n$ -by- $d$  matrix `Y` on the predictor variables in the matrix or cell array `X`, evaluated for the  $p$ -by-1 column vector `b` of coefficient estimates and the  $d$ -by- $d$  matrix `SIGMA` specifying the covariance of a row of `Y`. If  $d = 1$ , `X` can be an  $n$ -by- $p$  design matrix of predictor variables. For any value of  $d$ , `X` can also be a cell array of length  $n$ , with each cell containing a  $d$ -by- $p$  design matrix for one multivariate observation. If all observations have the same  $d$ -by- $p$  design matrix, `X` can be a single cell.

NaN values in `X` or `Y` are taken as missing. Observations with missing values in `X` are ignored. Treatment of missing values in `Y` depends on the algorithm specified by `alg`.

`alg` should match the algorithm used by `mvregress` to obtain the coefficient estimates `b`, and must be one of the following:

- 'ecm' — ECM algorithm
- 'cwlsl' — Least squares conditionally weighted by `SIGMA`
- 'mvn' — Multivariate normal estimates computed after omitting rows with any missing values in `Y`

`[nlogL, COVB] = mvregresslike(...)` also returns an estimated covariance matrix `COVB` of the parameter estimates `b`.

[nlogL, COVB] = mvregresslike(..., *type*, *format*) specifies the type and format of COVB.

*type* is either:

- 'hessian' — To use the Hessian or observed information. This method takes into account the increased uncertainties due to missing data. This is the default.
- 'fisher' — To use the Fisher or expected information. This method uses the complete data expected information, and does not include uncertainty due to missing data.

*format* is either:

- 'beta' — To compute COVB for *b* only. This is the default.
- 'full' — To compute COVB for both *b* and SIGMA.

## See Also

manoval | mvregress

## Topics

“Multivariate Normal Distribution” on page B-106

**Introduced in R2007a**

## mvnrnd

Multivariate normal random numbers

### Syntax

```
R = mvnrnd(MU, SIGMA)
r = mvnrnd(MU, SIGMA, cases)
```

### Description

`R = mvnrnd(MU, SIGMA)` returns an  $n$ -by- $d$  matrix `R` of random vectors chosen from the multivariate normal distribution with mean `MU`, and covariance `SIGMA`. `MU` is a vector or  $n$ -by- $d$  matrix, and `mvnrnd` generates each row of `R` using the corresponding row of `mu`. `SIGMA` is a  $d$ -by- $d$  symmetric positive semi-definite matrix, or a  $d$ -by- $d$ -by- $n$  array. If `SIGMA` is an array, `mvnrnd` generates each row of `R` using the corresponding page of `SIGMA`, i.e., `mvnrnd` computes `R(i, :)` using `MU(i, :)` and `SIGMA(:, :, i)`. If the covariance matrix is diagonal, containing variances along the diagonal and zero covariances off the diagonal, `SIGMA` may also be specified as a 1-by- $d$  vector or a 1-by- $d$ -by- $n$  array, containing just the diagonal. If `MU` is a 1-by- $d$  vector, `mvnrnd` replicates it to match the trailing dimension of `SIGMA`.

`r = mvnrnd(MU, SIGMA, cases)` returns a cases-by- $d$  matrix `R` of random vectors chosen from the multivariate normal distribution with a common 1-by- $d$  mean vector `MU`, and a common  $d$ -by- $d$  covariance matrix `SIGMA`.

### Examples

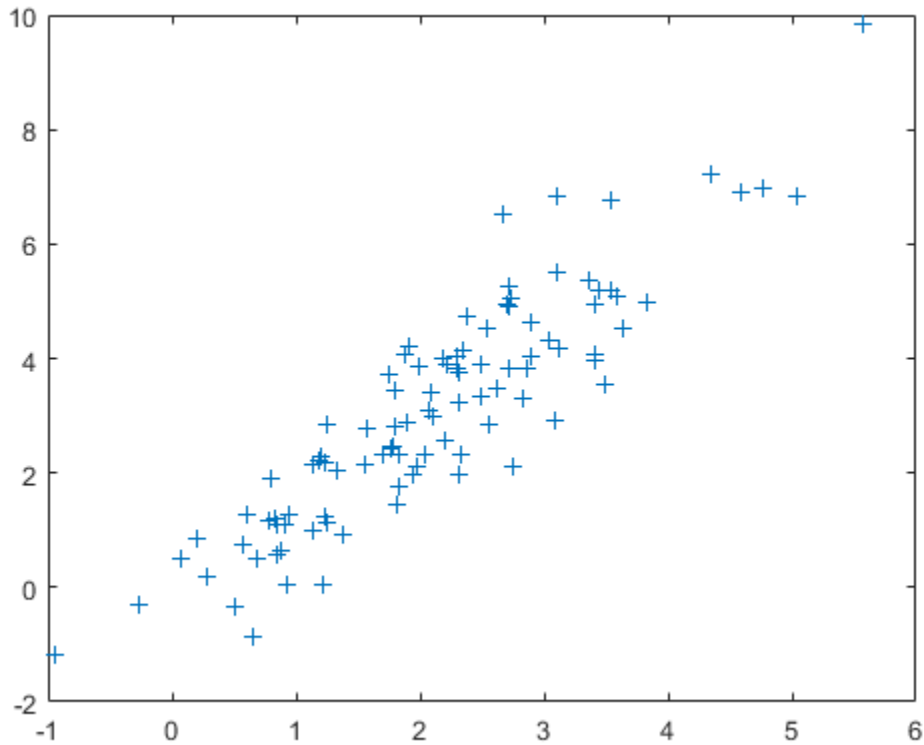
#### Generate Multivariate Normal Random Numbers

Generate random numbers from a multivariate normal distribution with parameters `mu = [2, 3]` and `sigma = [1, 1.5; 1.5, 3]`.

```
mu = [2,3];  
sigma = [1,1.5;1.5,3];  
rng default % For reproducibility  
r = mvnrnd(mu,sigma,100);
```

Plot the random numbers.

```
figure  
plot(r(:,1),r(:,2),'+')
```



## See Also

mvncdf | mvnpdf | normrnd

## **Topics**

“Multivariate Normal Distribution” on page B-106

**Introduced before R2006a**

## mvtcdf

Multivariate  $t$  cumulative distribution function

### Syntax

```
y = mvtcdf(X,C,DF)
y = mvtcdf(xl,xu,C,DF)
[y,err] = mvtcdf(...)
[...] = mvntcdf(...,options)
```

### Description

`y = mvtcdf(X,C,DF)` returns the cumulative probability of the multivariate  $t$  distribution with correlation parameters  $C$  and degrees of freedom  $DF$ , evaluated at each row of  $X$ . Rows of the  $n$ -by- $d$  matrix  $X$  correspond to observations or points, and columns correspond to variables or coordinates.  $y$  is an  $n$ -by-1 vector.

$C$  is a symmetric, positive definite,  $d$ -by- $d$  matrix, typically a correlation matrix. If its diagonal elements are not 1, `mvtcdf` scales  $C$  to correlation form. `mvtcdf` does not rescale  $X$ .  $DF$  is a scalar, or a vector with  $n$  elements.

The multivariate  $t$  cumulative probability at  $X$  is defined as the probability that a random vector  $T$ , distributed as multivariate  $t$ , will fall within the semi-infinite rectangle with upper limits defined by  $X$ , i.e.,  $\Pr\{T(1) \leq X(1), T(2) \leq X(2), \dots, T(d) \leq X(d)\}$ .

`y = mvtcdf(xl,xu,C,DF)` returns the multivariate  $t$  cumulative probability evaluated over the rectangle with lower and upper limits defined by  $x_l$  and  $x_u$ , respectively.

`[y,err] = mvtcdf(...)` returns an estimate of the error in  $y$ . For bivariate and trivariate distributions, `mvtcdf` uses adaptive quadrature on a transformation of the  $t$  density, based on methods developed by Genz, as described in the references. The default absolute error tolerance for these cases is  $1e-8$ . For four or more dimensions, `mvtcdf` uses a quasi-Monte Carlo integration algorithm based on methods developed by Genz and Bretz, as described in the references. The default absolute error tolerance for these cases is  $1e-4$ .

`[...] = mvntdf(...,options)` specifies control parameters for the numerical integration used to compute  $y$ . This argument can be created by a call to `statset`. Choices of `statset` parameters are:

- `'TolFun'` — Maximum absolute error tolerance. Default is  $1e-8$  when  $d < 4$ , or  $1e-4$  when  $d \geq 4$ .
- `'MaxFunEvals'` — Maximum number of integrand evaluations allowed when  $d \geq 4$ . Default is  $1e7$ . `'MaxFunEvals'` is ignored when  $d < 4$ .
- `'Display'` — Level of display output. Choices are `'off'` (the default), `'iter'`, and `'final'`. `'Display'` is ignored when  $d < 4$ .

## Examples

### Compute the Multivariate $t$ Distribution cdf

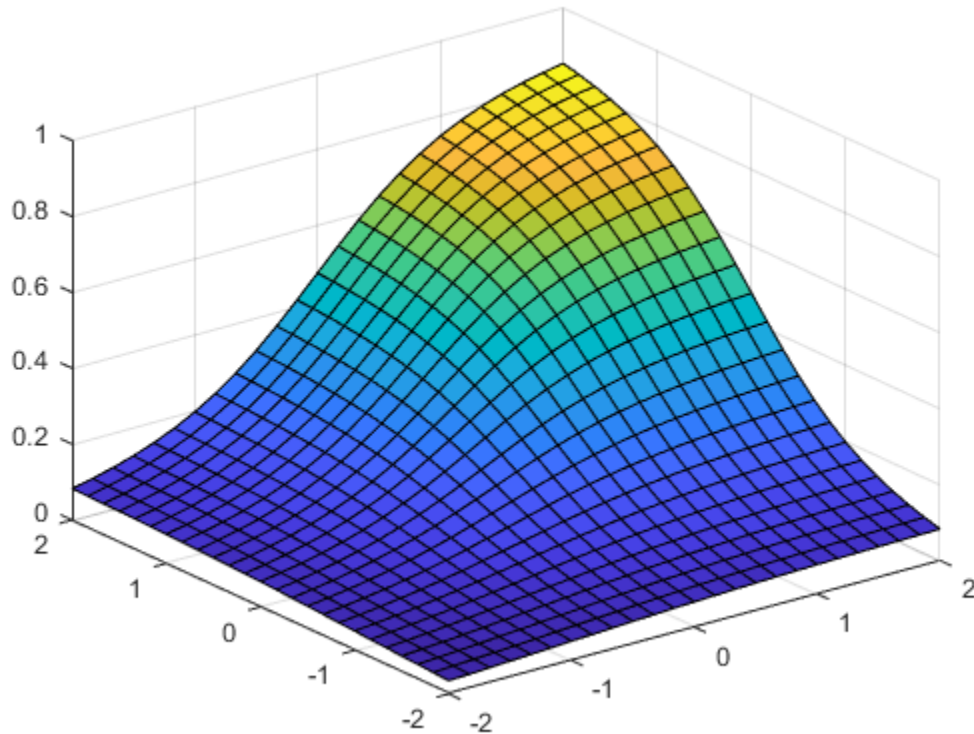
Compute the cdf of a multivariate  $t$  distribution with correlation parameters  $C = \begin{bmatrix} 1 & .4 \\ .4 & 1 \end{bmatrix}$  and 2 degrees of freedom.

```
C = [1 .4; .4 1];
df = 2;
[X1,X2] = meshgrid(linspace(-2,2,25)',linspace(-2,2,25)');
X = [X1(:) X2(:)];
p = mvtcdf(X,C,df);
```

Plot the cdf.

```
figure;
surf(X1,X2,reshape(p,25,25));
```





## References

- [1] Genz, A. “Numerical Computation of Rectangular Bivariate and Trivariate Normal and t Probabilities.” *Statistics and Computing*. Vol. 14, No. 3, 2004, pp. 251–260.
- [2] Genz, A., and F. Bretz. “Numerical Computation of Multivariate t Probabilities with Application to Power Calculation of Multiple Contrasts.” *Journal of Statistical Computation and Simulation*. Vol. 63, 1999, pp. 361–378.
- [3] Genz, A., and F. Bretz. “Comparison of Methods for the Computation of Multivariate t Probabilities.” *Journal of Computational and Graphical Statistics*. Vol. 11, No. 4, 2002, pp. 950–971.

## See Also

`mvtpdf` | `mvtrnd`

## Topics

“Multivariate t Distribution” on page B-112

**Introduced in R2006a**

# mvtpdf

Multivariate  $t$  probability density function

## Syntax

```
y = mvtpdf(X,C,df)
```

## Description

`y = mvtpdf(X,C,df)` returns the probability density of the multivariate  $t$  distribution with correlation parameters `C` and degrees of freedom `df`, evaluated at each row of `X`. Rows of the  $n$ -by- $d$  matrix `X` correspond to observations or points, and columns correspond to variables or coordinates. `C` is a symmetric, positive definite,  $d$ -by- $d$  matrix, typically a correlation matrix. If its diagonal elements are not 1, `mvtpdf` scales `C` to correlation form. `mvtcdf` does not rescale `X`. `df` is a scalar, or a vector with  $n$  elements. `y` is an  $n$ -by-1 vector.

## Examples

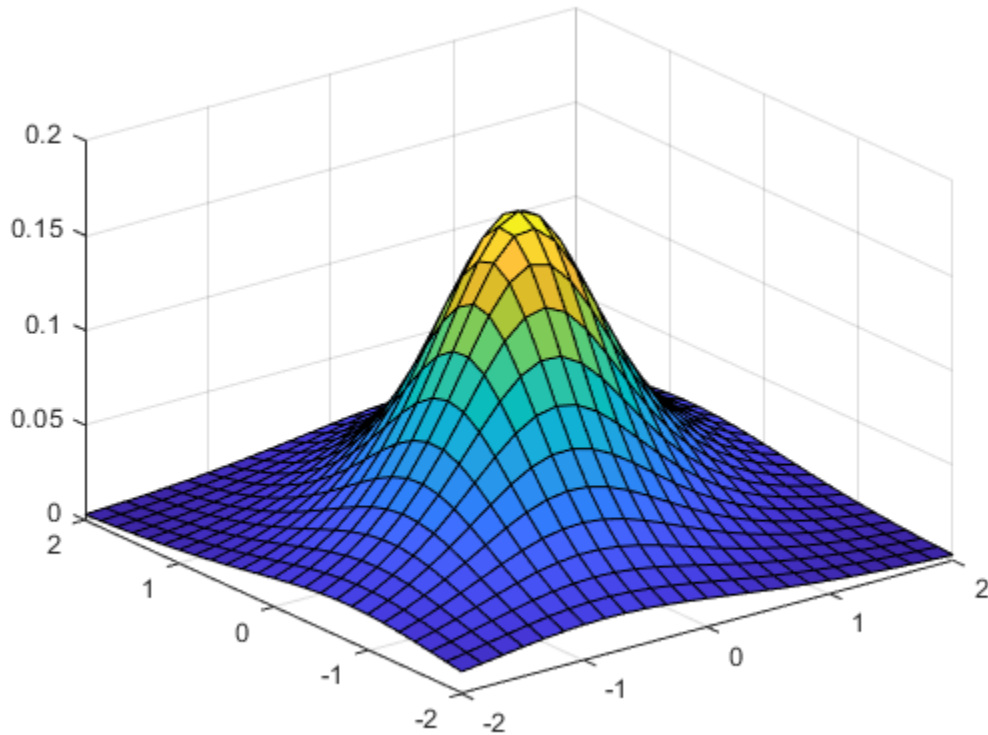
### Compute the Multivariate $t$ Distribution pdf

Compute the pdf of a multivariate  $t$  distribution with correlation parameters `C = [1 .4; .4 1]` and 2 degrees of freedom.

```
[X1,X2] = meshgrid(linspace(-2,2,25)',linspace(-2,2,25)');  
X = [X1(:) X2(:)];  
C = [1 .4; .4 1];  
df = 2;  
p = mvtpdf(X,C,df);
```

Plot the pdf.

```
figure;  
surf(X1,X2,reshape(p,25,25))
```



## See Also

`mvtcdf` | `mvtrnd`

## Topics

“Multivariate t Distribution” on page B-112

Introduced in R2006b

# mvtrnd

Multivariate  $t$  random numbers

## Syntax

```
R = mvtrnd(C,df,cases)
R = mvtrnd(C,df)
```

## Description

`R = mvtrnd(C,df,cases)` returns a matrix of random numbers chosen from the multivariate  $t$  distribution, where  $C$  is a correlation matrix. `df` is the degrees of freedom and is either a scalar or is a vector with `cases` elements. If  $p$  is the number of columns in  $C$ , then the output  $R$  has `cases` rows and  $p$  columns.

Let  $\tau$  represent a row of  $R$ . Then the distribution of  $\tau$  is that of a vector having a multivariate normal distribution with mean 0, variance 1, and covariance matrix  $C$ , divided by an independent chi-square random value having `df` degrees of freedom. The rows of  $R$  are independent.

$C$  must be a square, symmetric and positive definite matrix. If its diagonal elements are not all 1 (that is, if  $C$  is a covariance matrix rather than a correlation matrix), `mvtrnd` rescales  $C$  to transform it to a correlation matrix before generating the random numbers.

`R = mvtrnd(C,df)` returns a single random number from the multivariate  $t$  distribution.

## Examples

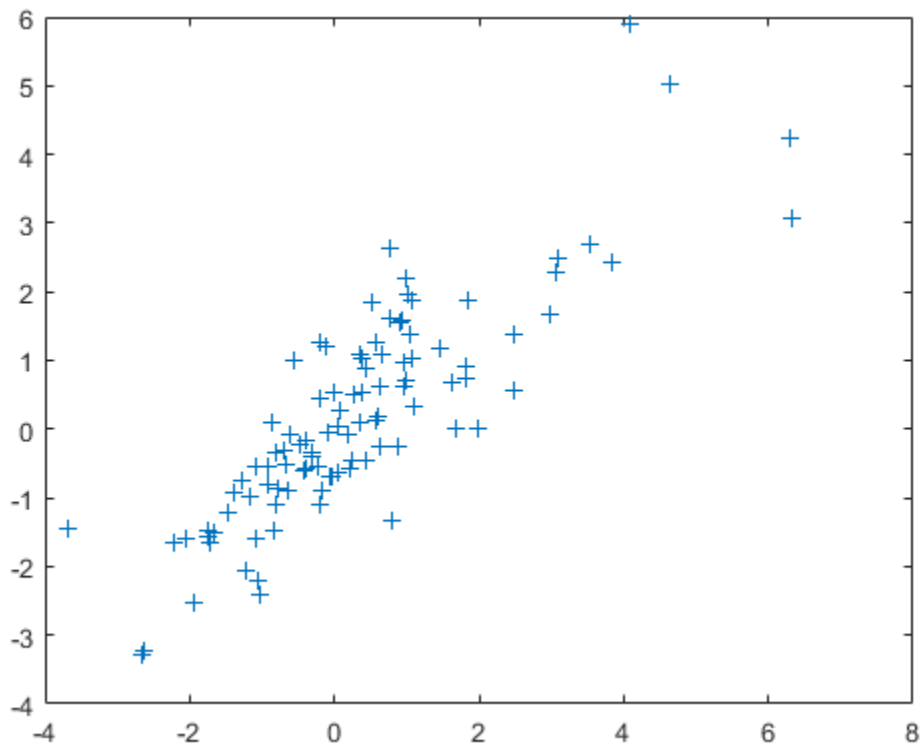
### Generate Multivariate $t$ Distribution Random Numbers

Generate random numbers from a multivariate  $t$  distribution with correlation parameters `SIGMA = [1 0.8; 0.8 1]` and 3 degrees of freedom.

```
rng default; % For reproducibility
SIGMA = [1 0.8;0.8 1];
R = mvtrnd(SIGMA,3,100);
```

Plot the random numbers.

```
figure;
plot(R(:,1),R(:,2), '+')
```



## See Also

[mvtcdf](#) | [mvtpdf](#)

## Topics

“Multivariate t Distribution” on page B-112

Introduced before R2006a

## NumObservations property

**Class:** cvpartition

Number of observations (including observations with missing group values)

### Description

Number of observations (including observations with missing group values).



# NaiveBayes class

Naive Bayes classifier

---

**Note** `NaiveBayes` will be removed in a future release. Use `fitcnb` to create a `ClassificationNaiveBayes` object instead.

---

## Description

A `NaiveBayes` object defines a Naive Bayes classifier. A Naive Bayes classifier assigns a new observation to the most probable class, assuming the features are conditionally independent given the class value.

## Construction

`NaiveBayes`                      Create `NaiveBayes` object

## Methods

<code>disp</code>	Display <code>NaiveBayes</code> classifier object
<code>display</code>	Display <code>NaiveBayes</code> classifier object
<code>fit</code>	Create Naive Bayes classifier object by fitting training data
<code>posterior</code>	Compute posterior probability of each class for test data
<code>predict</code>	Predict class label for test data
<code>subsasgn</code>	Subscripted reference for <code>NaiveBayes</code> object
<code>subsref</code>	Subscripted reference for <code>NaiveBayes</code> object

## Properties

CIIsNonEmpty	Flag for non-empty classes
ClassLevels	Class levels
Prior	Class priors
Dist	Distribution names
NClasses	Number of classes
NDims	Number of dimensions
Params	Parameter estimates

## Copy Semantics

Value. To learn how this affects your use of the class, see [Comparing Handle and Value Classes \(MATLAB\)](#) in the MATLAB Object-Oriented Programming documentation.

## Examples

Predict the class label using the Naive Bayes classifier:

```
load fisheriris
```

Use the default Gaussian distribution and a confusion matrix:

```
O1 = fitNaiveBayes(meas,species);  
C1 = O1.predict(meas);  
cMat1 = confusionmat(species,C1)
```

This returns:

```
cMat1 =  
  
    50     0     0  
     0    47     3  
     0     3    47
```

Use the Gaussian distribution for features 1 and 3 and use the kernel density estimation for features 2 and 4:

```
O2 = fitNaiveBayes(meas, species, 'dist', ...  
{ 'normal', 'kernel', 'normal', 'kernel' });  
C2 = O2.predict(meas);  
cMat2 = confusionmat(species, C2)
```

This returns:

cMat2 =

50	0	0
0	47	3
0	3	47

## References

- [1] Mitchell, T. (1997) Machine Learning, McGraw Hill.
- [2] Vangelis M., Ion A., and Geogios P. Spam Filtering with Naive Bayes - Which Naive Bayes? (2006) Third Conference on Email and Anti-Spam.
- [3] George H. John and Pat Langley. Estimating continuous distributions in bayesian classifiers (1995) the Eleventh Conference on Uncertainty in Artificial Intelligence.

## See Also

### Topics

- “Naive Bayes Classification” on page 21-2
- “Grouping Variables” on page 2-59

## NaiveBayes

**Class:** NaiveBayes

Create NaiveBayes object

---

**Note** NaiveBayes will be removed in a future release. Use `fitcnb` to create a `ClassificationNaiveBayes` object instead.

---

## Description

You cannot create a NaiveBayes classifier by calling the constructor. Use `fitNaiveBayes` to create a NaiveBayes classifier by fitting the object to training data.

## See Also

`fitNaiveBayes`

**Introduced in R2009a**

## nancov

Covariance ignoring NaN values

### Syntax

```
Y = nancov(X)
Y = nancov(X1,X2)
Y = nancov(...,1)
Y = nancov(...,'pairwise')
```

### Description

$Y = \text{nancov}(X)$  is the covariance `cov` of  $X$ , computed after removing observations with NaN values.

For vectors  $x$ ,  $\text{nancov}(x)$  is the sample variance of the remaining elements, once NaN values are removed. For matrices  $X$ ,  $\text{nancov}(X)$  is the sample covariance of the remaining observations, once observations (rows) containing any NaN values are removed.

$Y = \text{nancov}(X1, X2)$ , where  $X1$  and  $X2$  are matrices with the same number of elements, is equivalent to  $\text{nancov}(X)$ , where  $X = [X1(:) \ X2(:)]$ .

`nancov` removes the mean from each variable (column for matrix  $X$ ) before calculating  $Y$ . If  $n$  is the number of remaining observations after removing observations with NaN values, `nancov` normalizes  $Y$  by either  $n - 1$  or  $n$ , depending on whether  $n > 1$  or  $n = 1$ , respectively. To specify normalization by  $n$ , use  $Y = \text{nancov}(\dots, 1)$ .

$Y = \text{nancov}(\dots, 'pairwise')$  computes  $Y(i, j)$  using rows with no NaN values in columns  $i$  or  $j$ . The result  $Y$  may not be a positive definite matrix.

### Examples

Generate random data for two variables (columns) with random missing values:

```
X = rand(10,2);
p = randperm(numel(X));
X(p(1:5)) = NaN
X =
    0.8147    0.1576
      NaN      NaN
    0.1270    0.9572
    0.9134      NaN
    0.6324      NaN
    0.0975    0.1419
    0.2785    0.4218
    0.5469    0.9157
    0.9575    0.7922
    0.9649      NaN
```

Establish a correlation between a third variable and the other two variables:

```
X(:,3) = sum(X,2)
X =
    0.8147    0.1576    0.9723
      NaN      NaN      NaN
    0.1270    0.9572    1.0842
    0.9134      NaN      NaN
    0.6324      NaN      NaN
    0.0975    0.1419    0.2394
    0.2785    0.4218    0.7003
    0.5469    0.9157    1.4626
    0.9575    0.7922    1.7497
    0.9649      NaN      NaN
```

Compute the covariance matrix for the three variables after removing observations (rows) with NaN values:

```
Y = nancov(X)
Y =
    0.1311    0.0096    0.1407
    0.0096    0.1388    0.1483
    0.1407    0.1483    0.2890
```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- If the input is variable-size and is [] at run time, the generated code returns [] not NaN.

### See Also

NaN | cov | nanvar | var

Introduced before R2006a

## nanmax

Maximum ignoring NaN values

### Syntax

```
y = nanmax(X)
Y = nanmax(X1,X2)
y = nanmax(X, [], dim)
[y, indices] = nanmax(...)
```

### Description

`y = nanmax(X)` is the maximum `max` of `X`, computed after removing NaN values.

For vectors `x`, `nanmax(x)` is the maximum of the remaining elements, once NaN values are removed. For matrices `X`, `nanmax(X)` is a row vector of column maxima, once NaN values are removed. For multidimensional arrays `X`, `nanmax` operates along the first nonsingleton dimension.

`Y = nanmax(X1,X2)` returns an array `Y` the same size as `X1` and `X2` with `Y(i,j) = nanmax(X1(i,j), X2(i,j))`. Scalar inputs are expanded to an array of the same size as the other input.

`y = nanmax(X, [], dim)` operates along the dimension `dim` of `X`.

`[y, indices] = nanmax(...)` also returns the row indices of the maximum values for each column in the vector `indices`.

### Examples

Find column maxima and their indices for data with missing values:

```
X = magic(3);
X([1 6:9]) = repmat(NaN,1,5)
```



```
X =  
    NaN     1    NaN  
     3     5    NaN  
     4    NaN    NaN  
[y,indices] = nanmax(X)  
y =  
     4     5    NaN  
indices =  
     3     2     1
```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

[NaN](#) | [max](#) | [nanmin](#)

Introduced before R2006a

## nanmean

Mean ignoring NaN values

### Syntax

```
y = nanmean(X)
y = nanmean(X, dim)
```

### Description

`y = nanmean(X)` is the mean of `X`, computed after removing NaN values.

For vectors `x`, `nanmean(x)` is the mean of the remaining elements, once NaN values are removed. For matrices `X`, `nanmean(X)` is a row vector of column means, once NaN values are removed. For multidimensional arrays `X`, `nanmean` operates along the first nonsingleton dimension.

`y = nanmean(X, dim)` takes the mean along dimension `dim` of `X`.

---

**Note** If `X` contains a vector of all NaN values along some dimension, the vector is empty once the NaN values are removed, so the sum of the remaining elements is 0. Since the mean involves division by 0, its value is NaN. The output NaN is not a mean of NaN values.

---

### Examples

Find column means for data with missing values:

```
X = magic(3);
X([1 6:9]) = repmat(NaN,1,5)
X =
    NaN     1    NaN
     3     5    NaN
     4    NaN    NaN
```

```
y = nanmean(X)
Y =
    3.5000    3.0000    NaN
```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

NaN | mean | nanmedian

**Introduced before R2006a**

## nanmedian

Median ignoring NaN values

### Syntax

```
y = nanmedian(X)
y = nanmedian(X,dim)
```

### Description

`y = nanmedian(X)` is the median of `X`, computed after removing NaN values.

For vectors `x`, `nanmedian(x)` is the median of the remaining elements, once NaN values are removed. For matrices `X`, `nanmedian(X)` is a row vector of column medians, once NaN values are removed. For multidimensional arrays `X`, `nanmedian` operates along the first nonsingleton dimension.

`y = nanmedian(X,dim)` takes the mean along dimension `dim` of `X`.

### Examples

Find column medians for data with missing values:

```
X = magic(3);
X([1 6:9]) = repmat(NaN,1,5)
X =
    NaN     1     NaN
     3     5     NaN
     4    NaN    NaN
y = nanmedian(X)
y =
    3.5000    3.0000    NaN
```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`NaN` | `median` | `nanmean`

**Introduced before R2006a**

## nanmin

Minimum ignoring NaN values

### Syntax

```
y = nanmin(X)
Y = nanmin(X1,X2)
y = nanmin(X, [], dim)
[y, indices] = nanmin(...)
```

### Description

`y = nanmin(X)` is the minimum `min` of `X`, computed after removing NaN values.

For vectors `x`, `nanmin(x)` is the minimum of the remaining elements, once NaN values are removed. For matrices `X`, `nanmin(X)` is a row vector of column minima, once NaN values are removed. For multidimensional arrays `X`, `nanmin` operates along the first nonsingleton dimension.

`Y = nanmin(X1,X2)` returns an array `Y` the same size as `X1` and `X2` with `Y(i,j) = nanmin(X1(i,j), X2(i,j))`. Scalar inputs are expanded to an array of the same size as the other input.

`y = nanmin(X, [], dim)` operates along the dimension `dim` of `X`.

`[y, indices] = nanmin(...)` also returns the row indices of the minimum values for each column in the vector `indices`.

### Examples

Find column minima and their indices for data with missing values:

```
X = magic(3);
X([1 6:9]) = repmat(NaN,1,5)
```

```
X =  
    NaN     1     NaN  
     3     5     NaN  
     4    NaN     NaN  
[y,indices] = nanmin(X)  
y =  
     3     1     NaN  
indices =  
     2     1     1
```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

[NaN](#) | [min](#) | [nanmax](#)

Introduced before R2006a

## nanstd

Standard deviation ignoring NaN values

### Syntax

```
y = nanstd(X)
y = nanstd(X,1)
y = nanstd(X,flag,dim)
```

### Description

`y = nanstd(X)` is the standard deviation `std` of `X`, computed after removing NaN values.

For vectors `x`, `nanstd(x)` is the sample standard deviation of the remaining elements, once NaN values are removed. For matrices `X`, `nanstd(X)` is a row vector of column sample standard deviations, once NaN values are removed. For multidimensional arrays `X`, `nanstd` operates along the first nonsingleton dimension.

If  $n$  is the number of remaining observations after removing observations with NaN values, `nanstd` normalizes `y` by  $n-1$ . To specify normalization by  $n$ , use `y = nanstd(X, 1)`.

`y = nanstd(X,flag,dim)` takes the standard deviation along the dimension `dim` of `X`. The `flag` is 0 or 1 to specify normalization by  $n-1$  or  $n$ , respectively, where  $n$  is the number of remaining observations after removing observations with NaN values.

### Examples

Find column standard deviations for data with missing values:

```
X = magic(3);
X([1 6:9]) = repmat(NaN,1,5)
X =
```



```
NaN     1     NaN
     3     5     NaN
     4     NaN   NaN
y = nanstd(X)
y =
    0.7071    2.8284    NaN
```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

NaN | nanmean | nanvar | std

**Introduced before R2006a**

## nansum

Sum ignoring NaN values

### Syntax

```
y = nansum(X)
y = nansum(X,dim)
```

### Description

`y = nansum(X)` is the sum of `X`, computed after removing NaN values.

For vectors `x`, `nansum(x)` is the sum of the remaining elements, once NaN values are removed. For matrices `X`, `nansum(X)` is a row vector of column sums, once NaN values are removed. For multidimensional arrays `X`, `nansum` operates along the first nonsingleton dimension.

`y = nansum(X, dim)` takes the sum along dimension `dim` of `X`.

---

**Note** If `X` contains a vector of all NaN values along some dimension, the vector is empty once the NaN values are removed, so the sum of the remaining elements is 0. The output 0 is not a sum of NaN values.

---

### Examples

Find column sums for data with missing values:

```
X = magic(3);
X([1 6:9]) = repmat(NaN,1,5)
X =
    NaN     1    NaN
     3     5    NaN
     4    NaN    NaN
y = nansum(X)
```

$y =$   
7 6 0

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

NaN | sum

Introduced before R2006a

## nanvar

Variance, ignoring NaN values

### Syntax

```
y = nanvar(X)
y = nanvar(X,1)
y = nanvar(X,w)
y = nanvar(X,w,dim)
```

### Description

`y = nanvar(X)` is the variance `var` of `X`, computed after removing NaN values.

For vectors `x`, `nanvar(x)` is the sample variance of the remaining elements, once NaN values are removed. For matrices `X`, `nanvar(X)` is a row vector of column sample variances, once NaN values are removed. For multidimensional arrays `X`, `nanvar` operates along the first nonsingleton dimension.

`nanvar` removes the mean from each variable (column for matrix `X`) before calculating `y`. If `n` is the number of remaining observations after removing observations with NaN values, `nanvar` normalizes `y` by either `n - 1` or `n`, depending on whether `n > 1` or `n = 1`, respectively. To specify normalization by `n`, use `y = nanvar(X,1)`.

`y = nanvar(X,w)` computes the variance using the weight vector `w`. The length of `w` must equal the length of the dimension over which `nanvar` operates, and its elements must be nonnegative. Elements of `X` corresponding to NaN values of `w` are ignored.

`y = nanvar(X,w,dim)` takes the variance along the dimension `dim` of `X`. Set `w` to `[]` to use the default normalization by `n - 1`.

### Examples

Find column standard deviations for data with missing values:

```
X = magic(3);
X([1 6:9]) = repmat(NaN,1,5)
X =
    NaN     1    NaN
     3     5    NaN
     4    NaN    NaN
y = nanvar(X)
y =
    0.5000    8.0000    NaN
```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

[NaN](#) | [nanmean](#) | [nanstd](#) | [var](#)

Introduced before R2006a

## nbincdf

Negative binomial cumulative distribution function

### Syntax

```
y = nbincdf(x, R, p)
y = nbincdf(x, R, p, 'upper')
```

### Description

`y = nbincdf(x, R, p)` computes the negative binomial cdf at each of the values in `x` using the corresponding number of successes, `R` and probability of success in a single trial, `p`. `x`, `R`, and `p` can be vectors, matrices, or multidimensional arrays that all have the same size, which is also the size of `y`. A scalar input for `x`, `R`, or `p` is expanded to a constant array with the same dimensions as the other inputs.

`y = nbincdf(x, R, p, 'upper')` returns the complement of the negative binomial cdf at each value in `x`, using an algorithm that more accurately computes the extreme upper tail probabilities.

The negative binomial cdf is

$$y = F(x | r, p) = \sum_{i=0}^x \binom{r+i-1}{i} p^r q^i I_{(0,1,\dots)}(i)$$

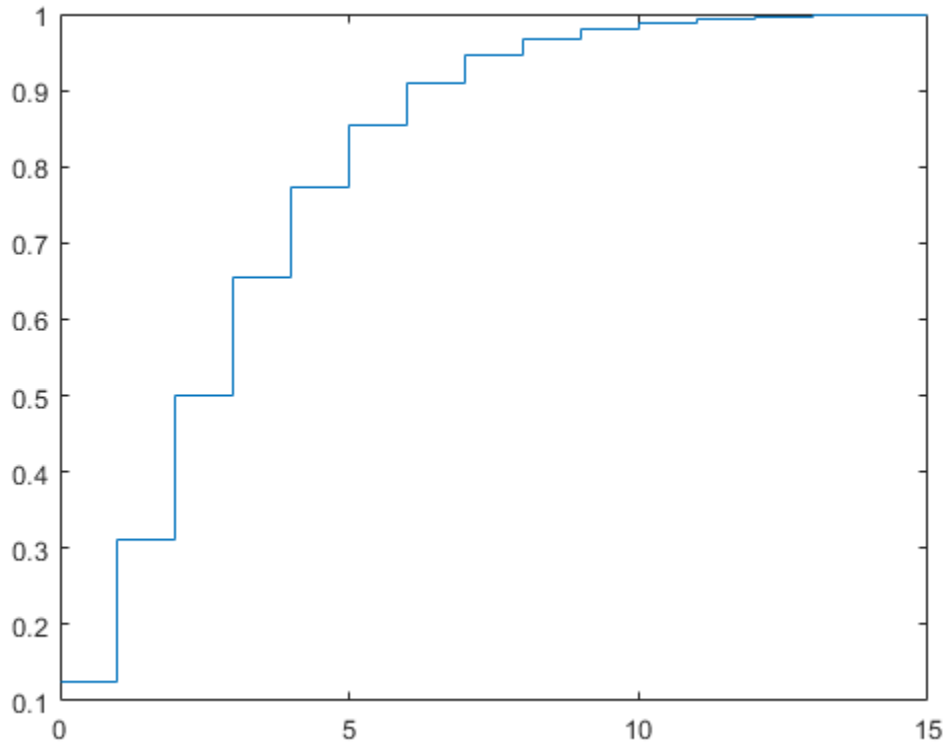
The simplest motivation for the negative binomial is the case of successive random trials, each having a constant probability `p` of success. The number of *extra* trials you must perform in order to observe a given number `R` of successes has a negative binomial distribution. However, consistent with a more general interpretation of the negative binomial, `nbincdf` allows `R` to be any positive value, including nonintegers. When `R` is noninteger, the binomial coefficient in the definition of the cdf is replaced by the equivalent expression

$$\frac{\Gamma(r+i)}{\Gamma(r)\Gamma(i+1)}$$

## Examples

### Compute Negative Binomial Distribution CDF

```
x = (0:15);  
p = nbincdf(x,3,0.5);  
stairs(x,p)
```



## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`cdf` | `nbinfit` | `nbininv` | `nbinpdf` | `nbinrnd` | `nbinstat`



## Topics

“Negative Binomial Distribution” on page B-120

Introduced before R2006a

## nbinfit

Negative binomial parameter estimates

### Syntax

```
parmhat = nbinfit(data)
[parmhat,parmci] = nbinfit(data,alpha)
[...] = nbinfit(data,alpha,options)
```

### Description

`parmhat = nbinfit(data)` returns the maximum likelihood estimates (MLEs) of the parameters of the negative binomial distribution given the data in the vector `data`.

`[parmhat,parmci] = nbinfit(data,alpha)` returns MLEs and 100(1-alpha) percent confidence intervals. By default, `alpha = 0.05`, which corresponds to 95% confidence intervals.

`[...] = nbinfit(data,alpha,options)` accepts a structure, `options`, that specifies control parameters for the iterative algorithm the function uses to compute maximum likelihood estimates. The negative binomial fit function accepts an `options` structure which you can create using the function `statset`. Enter `statset('nbinfit')` to see the names and default values of the parameters that `nbinfit` accepts in the `options` structure. See the reference page for `statset` for more information about these options.

---

**Note** The variance of a negative binomial distribution is greater than its mean. If the sample variance of the data in `data` is less than its sample mean, `nbinfit` cannot compute MLEs. You should use the `poissfit` function instead.

---

### See Also

`mle` | `nbincdf` | `nbininv` | `nbinpdf` | `nbinrnd` | `nbinstat` | `statset`

## **Topics**

“Negative Binomial Distribution” on page B-120

**Introduced before R2006a**

## nbininv

Negative binomial inverse cumulative distribution function

### Syntax

```
X = nbininv(Y,R,P)
```

### Description

`X = nbininv(Y,R,P)` returns the inverse of the negative binomial cdf with corresponding number of successes, `R` and probability of success in a single trial, `P`. Since the binomial distribution is discrete, `nbininv` returns the least integer `X` such that the negative binomial cdf evaluated at `X` equals or exceeds `Y`. `Y`, `R`, and `P` can be vectors, matrices, or multidimensional arrays that all have the same size, which is also the size of `X`. A scalar input for `Y`, `R`, or `P` is expanded to a constant array with the same dimensions as the other inputs.

The simplest motivation for the negative binomial is the case of successive random trials, each having a constant probability `P` of success. The number of *extra* trials you must perform in order to observe a given number `R` of successes has a negative binomial distribution. However, consistent with a more general interpretation of the negative binomial, `nbininv` allows `R` to be any positive value, including nonintegers.

### Examples

How many times would you need to flip a fair coin to have a 99% probability of having observed 10 heads?

```
flips = nbininv(0.99,10,0.5) + 10  
flips =  
    33
```

Note that you have to flip at least 10 times to get 10 heads. That is why the second term on the right side of the equals sign is a 10.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`icdf` | `nbincdf` | `nbinfit` | `nbinpdf` | `nbinrnd` | `nbinstat`

### Topics

“Negative Binomial Distribution” on page B-120

Introduced before R2006a

## nbinpdf

Negative binomial probability density function

### Syntax

```
Y = nbinpdf(X,R,P)
```

### Description

`Y = nbinpdf(X,R,P)` returns the negative binomial pdf at each of the values in `X` using the corresponding number of successes, `R` and probability of success in a single trial, `P`. `X`, `R`, and `P` can be vectors, matrices, or multidimensional arrays that all have the same size, which is also the size of `Y`. A scalar input for `X`, `R`, or `P` is expanded to a constant array with the same dimensions as the other inputs. Note that the density function is zero unless the values in `X` are integers.

The negative binomial pdf is

$$y = f(x | r, p) = \binom{r+x-1}{x} p^r q^x I_{(0,1,\dots)}(x)$$

The simplest motivation for the negative binomial is the case of successive random trials, each having a constant probability `P` of success. The number of *extra* trials you must perform in order to observe a given number `R` of successes has a negative binomial distribution. However, consistent with a more general interpretation of the negative binomial, `nbinpdf` allows `R` to be any positive value, including nonintegers. When `R` is noninteger, the binomial coefficient in the definition of the pdf is replaced by the equivalent expression

$$\frac{\Gamma(r+x)}{\Gamma(r)\Gamma(x+1)}$$

### Examples

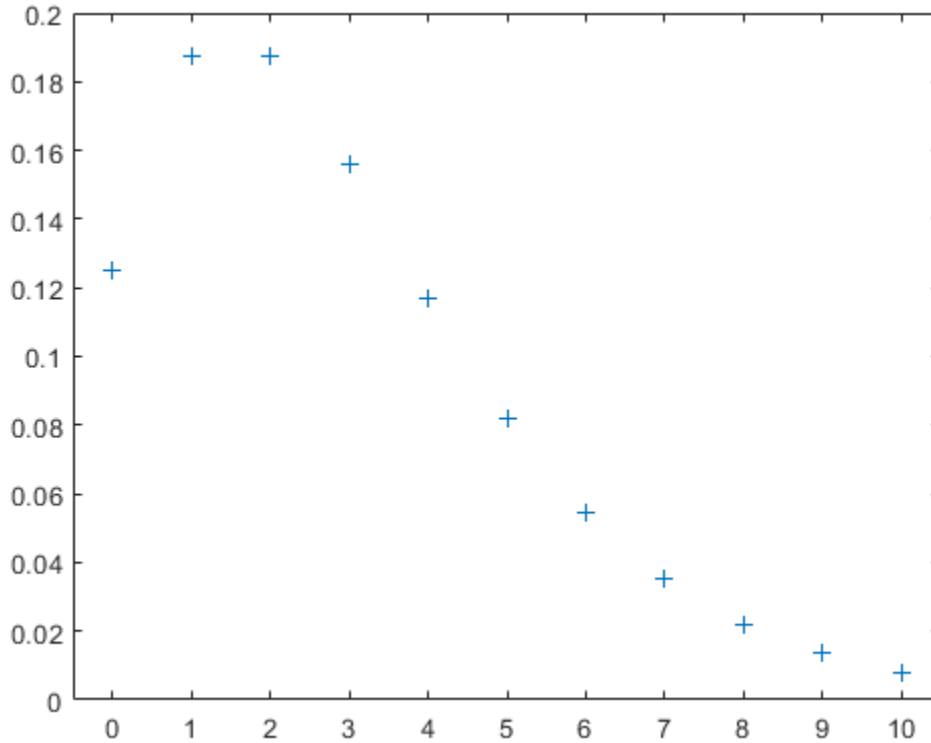
**Compute the Negative Binomial Distribution pdf**

Compute the pdf of a negative binomial distribution with parameters  $R = 3$  and  $p = 0.5$ .

```
x = (0:10);  
y = nbinpdf(x,3,0.5);
```

Plot the pdf.

```
figure;  
plot(x,y,'+')  
xlim([-0.5,10.5])
```



## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

[nbincdf](#) | [nbinfit](#) | [nbininv](#) | [nbinrnd](#) | [nbinstat](#) | [pdf](#)



## **Topics**

“Negative Binomial Distribution” on page B-120

**Introduced before R2006a**

## nbinrnd

Negative binomial random numbers

### Syntax

```
RND = nbinrnd(R,P)
RND = nbinrnd(R,P,m,n,...)
RND = nbinrnd(R,P,[m,n,...])
```

### Description

`RND = nbinrnd(R,P)` is a matrix of random numbers chosen from a negative binomial distribution with corresponding number of successes, `R` and probability of success in a single trial, `P`. `R` and `P` can be vectors, matrices, or multidimensional arrays that have the same size, which is also the size of `RND`. A scalar input for `R` or `P` is expanded to a constant array with the same dimensions as the other input.

`RND = nbinrnd(R,P,m,n,...)` or `RND = nbinrnd(R,P,[m,n,...])` generates an `m`-by-`n`-by-... array. The `R`, `P` parameters can each be scalars or arrays of the same size as `R`.

The simplest motivation for the negative binomial is the case of successive random trials, each having a constant probability `P` of success. The number of *extra* trials you must perform in order to observe a given number `R` of successes has a negative binomial distribution. However, consistent with a more general interpretation of the negative binomial, `nbinrnd` allows `R` to be any positive value, including nonintegers.

### Examples

Suppose you want to simulate a process that has a defect probability of 0.01. How many units might Quality Assurance inspect before finding three defective items?

```
r = nbinrnd(3,0.01,1,6)+3
r =
    496    142    420    396    851    178
```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

The generated code can return a different sequence of numbers than MATLAB if either of the following is true:

- The output is nonscalar.
- An input parameter is invalid for the distribution.

### See Also

`nbincdf` | `nbinfit` | `nbininv` | `nbinpdf` | `nbinstat` | `random`

### Topics

“Negative Binomial Distribution” on page B-120

**Introduced before R2006a**

## nbinstat

Negative binomial mean and variance

### Syntax

```
[M,V] = nbinstat(R,P)
```

### Description

`[M,V] = nbinstat(R,P)` returns the mean of and variance for the negative binomial distribution with corresponding number of successes,  $R$  and probability of success in a single trial,  $P$ .  $R$  and  $P$  can be vectors, matrices, or multidimensional arrays that all have the same size, which is also the size of  $M$  and  $V$ . A scalar input for  $R$  or  $P$  is expanded to a constant array with the same dimensions as the other input.

The mean of the negative binomial distribution with parameters  $r$  and  $p$  is  $rq / p$ , where  $q = 1 - p$ . The variance is  $rq / p^2$ .

The simplest motivation for the negative binomial is the case of successive random trials, each having a constant probability  $P$  of success. The number of *extra* trials you must perform in order to observe a given number  $R$  of successes has a negative binomial distribution. However, consistent with a more general interpretation of the negative binomial, `nbinstat` allows  $R$  to be any positive value, including nonintegers.

### Examples

```
p = 0.1:0.2:0.9;  
r = 1:5;  
[R,P] = meshgrid(r,p);  
[M,V] = nbinstat(R,P)  
M =  
 9.0000 18.0000 27.0000 36.0000 45.0000  
 2.3333 4.6667 7.0000 9.3333 11.6667  
 1.0000 2.0000 3.0000 4.0000 5.0000  
 0.4286 0.8571 1.2857 1.7143 2.1429
```

```
0.1111 0.2222 0.3333 0.4444 0.5556
```

```
V =
```

```
90.0000 180.0000 270.0000 360.0000 450.0000  
7.7778 15.5556 23.3333 31.1111 38.8889  
2.0000 4.0000 6.0000 8.0000 10.0000  
0.6122 1.2245 1.8367 2.4490 3.0612  
0.1235 0.2469 0.3704 0.4938 0.6173
```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

[nbincdf](#) | [nbinfit](#) | [nbininv](#) | [nbinpdf](#) | [nbinrnd](#)

### Topics

“Negative Binomial Distribution” on page B-120

**Introduced before R2006a**

## FeatureSelectionNCAClassification class

Feature selection for classification using neighborhood component analysis (NCA)

### Description

`FeatureSelectionNCAClassification` object contains the data, fitting information, feature weights, and other parameters of a neighborhood component analysis (NCA) model. `fscnca` learns the feature weights using a diagonal adaptation of NCA and returns an instance of a `FeatureSelectionNCAClassification` object. The function achieves feature selection by regularizing the feature weights.

### Construction

`mdl = fscnca(X, Y)` performs feature selection for classification using the predictors in `X` and class labels in `Y`.

`mdl = fscnca(X, Y, Name, Value)` performs feature selection for classification with additional options specified by one or more name-value pair arguments. For a full list and description of the name-value pair arguments, see `fscnca`.

### Input Arguments

#### **x** — Predictor variable values

*n*-by-*p* matrix

Predictor variable values, specified as an *n*-by-*p* matrix, where *n* is the number of observations and *p* is the number of predictor variables.

Data Types: `single` | `double`

#### **y** — Class labels

categorical vector | logical vector | numeric vector | cell array of character vectors of length *n* | character matrix with *n* rows

Class labels, specified as a categorical vector, logical vector, numeric vector, cell array of character vectors of length  $n$ , or character matrix with  $n$  rows, where  $n$  is the number of observations. Element  $i$  or row  $i$  of  $Y$  is the class label corresponding to row  $i$  of  $X$  (observation  $i$ ).

Data Types: `single` | `double` | `logical` | `char` | `cell` | `categorical`

## Properties

### **NumObservations** — Number of observations in the training data

scalar

Number of observations in the training data ( $X$  and  $Y$ ) after removing NaN or Inf values, stored as a scalar.

Data Types: `double`

### **ModelParameters** — Model parameters

structure

Model parameters used for training the model, stored as a structure.

You can access the fields of `ModelParameters` using dot notation.

For example, for a `FeatureSelectionNCAClassification` object named `mdl`, you can access the `LossFunction` value using `mdl.ModelParameters.LossFunction`.

Data Types: `struct`

### **Lambda** — Regularization parameter

scalar

Regularization parameter used for training this model, stored as a scalar. For  $n$  observations, the best `Lambda` value that minimizes the generalization error of the NCA model is expected to be a multiple of  $1/n$ .

Data Types: `double`

### **FitMethod** — Name of fitting method

character vector

Name of the fitting method used to fit this model, stored as one of the following:

- 'exact' — Perform fitting using all of the data.
- 'none' — No fitting. Use this option to evaluate the generalization error of the NCA model using the initial feature weights supplied in the call to `fscnca`.
- 'average' — Divide the data into partitions (subsets), fit each partition using the exact method, and return the average of the feature weights. You can specify the number of partitions using the `NumPartitions` name-value pair argument.

Data Types: `char`

**`solver`** — Name of the solver used to fit this model

character vector

Name of the solver used to fit this model, stored as one of the following:

- 'lbfgs' — Limited memory Broyden-Fletcher-Goldfarb-Shanno (LBFGS) algorithm
- 'sgd' — Stochastic gradient descent (SGD) algorithm
- 'minibatch-lbfgs' — stochastic gradient descent with LBFGS algorithm applied to mini-batches

Data Types: `char`

**`GradientTolerance`** — Relative convergence tolerance on gradient norm

positive scalar

Relative convergence tolerance on the gradient norm for the 'lbfgs' and 'minibatch-lbfgs' solvers, stored as a positive scalar value.

Data Types: `double`

**`IterationLimit`** — Maximum number of iterations for optimization

positive integer

Maximum number of iterations for optimization, stored as a positive integer value.

Data Types: `double`

**`PassLimit`** — Maximum number of passes

positive integer

Maximum number of passes for 'sgd' and 'minibatch-lbfgs' solvers. Every pass processes all of the observations in the data.



Data Types: double

**InitialLearningRate — Initial learning rate**

positive real scalar

Initial learning rate for the 'sgd' and 'minibatch-lbfgs' solvers, stored as a positive real scalar. The learning rate decays over iterations starting at the value specified for `InitialLearningRate`.

Use the `NumTuningIterations` and `TuningSubsetSize` name-value pair arguments to control the automatic tuning of initial learning rate in the call to `fscnca`.

Data Types: double

**verbose — Verbosity level indicator**

nonnegative integer

Verbosity level indicator, stored as a nonnegative integer. Possible values are:

- 0 — No convergence summary
- 1 — Convergence summary, including norm of gradient and objective function value
- >1 — More convergence information, depending on the fitting algorithm. When you use the 'minibatch-lbfgs' solver and verbosity level > 1, the convergence information includes the iteration log from intermediate minibatch LBFGS fits.

Data Types: double

**InitialFeatureWeights — Initial feature weights**

$p$ -by-1 vector of positive real scalars

Initial feature weights, stored as a  $p$ -by-1 vector of positive real scalars, where  $p$  is the number of predictors in  $X$ .

Data Types: double

**FeatureWeights — Feature weights**

$p$ -by-1 vector of real scalars

Feature weights, stored as a  $p$ -by-1 vector of real scalars, where  $p$  is the number of predictors in  $X$ .

If `FitMethod` is 'average', then `FeatureWeights` is a  $p$ -by- $m$  matrix.  $m$  is the number of partitions specified via the 'NumPartitions' name-value pair argument in the call to `fscnca`.

The absolute value of `FeatureWeights(k)` is a measure of the importance of predictor  $k$ . A `FeatureWeights(k)` value that is close to 0 indicates that predictor  $k$  does not influence the response in  $Y$ .

Data Types: `double`

### **FitInfo** — Fit information

structure

Fit information, stored as a structure with the following fields.

Field Name	Meaning
Iteration	Iteration index
Objective	Regularized objective function for minimization
UnregularizedObjective	Unregularized objective function for minimization
Gradient	Gradient of regularized objective function for minimization

- For classification, `UnregularizedObjective` represents the negative of the leave-one-out accuracy of the NCA classifier on the training data.
- For regression, `UnregularizedObjective` represents the leave-one-out loss between the true response and the predicted response when using the NCA regression model.
- For the 'lbfgs' solver, `Gradient` is the final gradient. For the 'sgd' and 'minibatch-lbfgs' solvers, `Gradient` is the final mini-batch gradient.
- If `FitMethod` is 'average', then `FitInfo` is an  $m$ -by-1 structure array, where  $m$  is the number of partitions specified via the 'NumPartitions' name-value pair argument.

You can access the fields of `FitInfo` using dot notation. For example, for a `FeatureSelectionNCAClassification` object named `mdl`, you can access the `Objective` field using `mdl.FitInfo.Objective`.

Data Types: `struct`

**Mu — Predictor means**

$p$ -by-1 vector | []

Predictor means, stored as a  $p$ -by-1 vector for standardized training data. In this case, the `predict` method centers predictor matrix  $X$  by subtracting the respective element of  $\mu$  from every column.

If data is not standardized during training, then  $\mu$  is empty.

Data Types: `double`

**Sigma — Predictor standard deviations**

$p$ -by-1 vector | []

Predictor standard deviations, stored as a  $p$ -by-1 vector for standardized training data. In this case, the `predict` method scales predictor matrix  $X$  by dividing every column by the respective element of  $\sigma$  after centering the data using  $\mu$ .

If data is not standardized during training, then  $\sigma$  is empty.

Data Types: `double`

**x — Predictor values**

$n$ -by- $p$  matrix

Predictor values used to train this model, stored as an  $n$ -by- $p$  matrix.  $n$  is the number of observations and  $p$  is the number of predictor variables in the training data.

Data Types: `double`

**y — Response values**

numeric vector of size  $n$

Response values used to train this model, stored as a numeric vector of size  $n$ , where  $n$  is the number of observations.

Data Types: `double`

**w — Observation weights**

numeric vector of size  $n$

Observation weights used to train this model, stored as a numeric vector of size  $n$ . The sum of observation weights is  $n$ .

Data Types: `double`

## Methods

`loss` Evaluate accuracy of learned feature weights on test data  
`predict` Predict responses using neighborhood component analysis (NCA) classifier  
`refit` Refit neighborhood component analysis (NCA) model for classification

## Examples

### Explore `FeatureSelectionNCAClassification` Object

Load the sample data.

```
load ionosphere
```

The data set has 34 continuous predictors. The response variable is the radar returns, labeled as b (bad) or g (good).

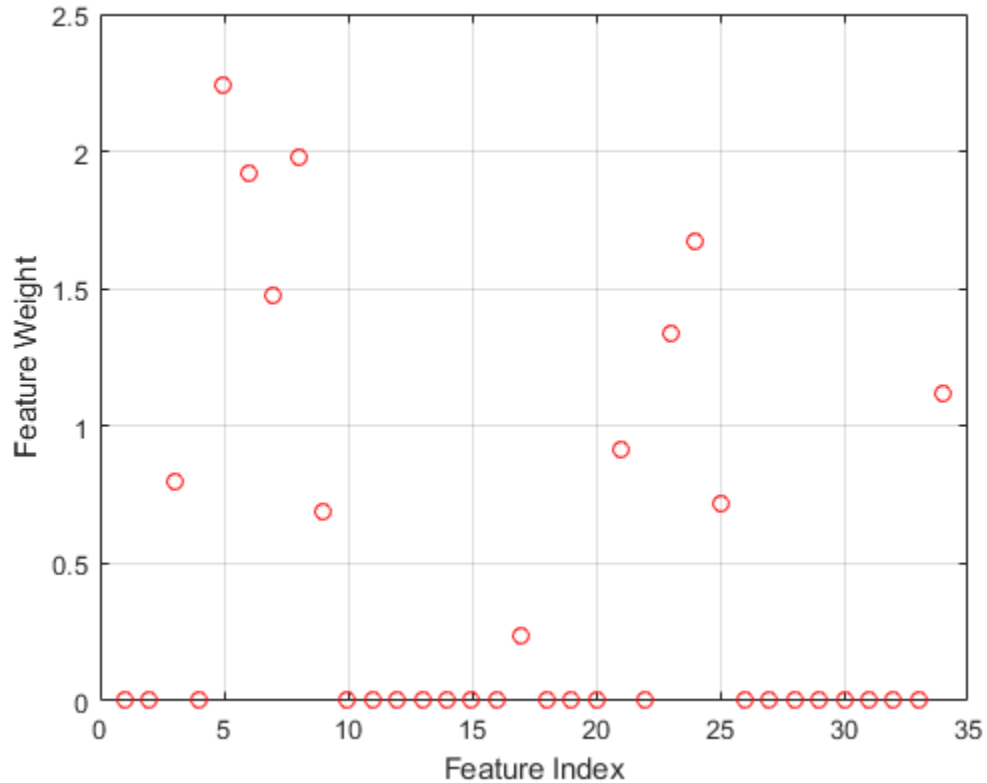
Fit a neighborhood component analysis (NCA) model for classification to detect the relevant features.

```
mdl = fscnca(X,Y);
```

The returned NCA model, `mdl`, is a `FeatureSelectionNCAClassification` object. This object stores information about the training data, model, and optimization. You can access the object properties, such as the feature weights, using dot notation.

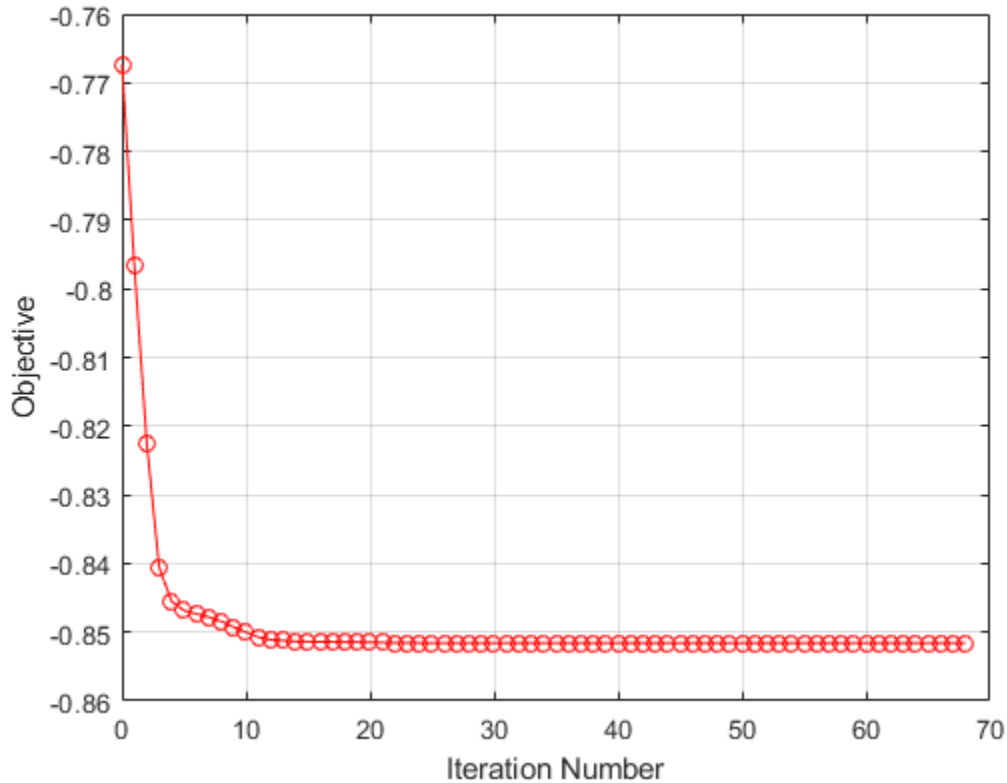
Plot the feature weights.

```
figure()  
plot(mdl.FeatureWeights,'ro')  
xlabel('Feature Index')  
ylabel('Feature Weight')  
grid on
```



The weights of the irrelevant features are zero. The 'Verbose', 1 option in the call to `fscnca` displays the optimization information on the command line. You can also visualize the optimization process by plotting the objective function versus the iteration number.

```
figure
plot mdl.FitInfo.Iteration, mdl.FitInfo.Objective, 'ro-'
grid on
xlabel('Iteration Number')
ylabel('Objective')
```



The `ModelParameters` property is a struct that contains more information about the model. You can access the fields of this property using dot notation. For example, see if the data was standardized or not.

```
mdl.ModelParameters.Standardize
```

```
ans =
```

```
logical
```

```
0
```

0 means that the data was not standardized before fitting the NCA model. You can standardize the predictors when they are on very different scales using the 'Standardize', 1 name-value pair argument in the call to `fscnca` .

## Copy Semantics

Value. To learn how value classes affect copy operations, see [Copying Objects \(MATLAB\)](#).

## See Also

`fscnca` | `loss` | `predict` | `refit`

## Topics

[Class Attributes \(MATLAB\)](#)

[Property Attributes \(MATLAB\)](#)

**Introduced in R2016b**

## FeatureSelectionNCARegression class

Feature selection for regression using neighborhood component analysis (NCA)

### Description

`FeatureSelectionNCARegression` contains the data, fitting information, feature weights, and other model parameters of a neighborhood component analysis (NCA) model. `fsrnca` learns the feature weights using a diagonal adaptation of NCA and returns an instance of `FeatureSelectionNCARegression` object. The function achieves feature selection by regularizing the feature weights.

### Construction

`mdl = fsrnca(X, Y)` performs feature selection for regression using the predictors in `X` and responses in `Y`.

`mdl = fsrnca(X, Y, Name, Value)` performs feature selection for regression with additional options specified by one or more name-value pair arguments. For a full list and description of the name-value pair arguments, see `fsrnca`.

### Input Arguments

#### **x** — Predictor variable values

*n*-by-*p* matrix

Predictor variable values, specified as an *n*-by-*p* matrix, where *n* is the number of observations and *p* is the number of predictor variables.

Data Types: `single` | `double`

#### **y** — Response values

numeric real vector of length *n*

Response values, specified as a numeric real vector of length *n*, where *n* is the number of observations and is equal to the number of rows in `X`.



Data Types: `single` | `double`

## Properties

### **NumObservations** — Number of observations in the training data

scalar

Number of observations in the training data (X and Y) after removing NaN or Inf values, stored as a scalar.

Data Types: `double`

### **ModelParameters** — Model parameters

structure

Model parameters used for training the model, stored as a structure.

You can access the fields of `ModelParameters` using dot notation.

For example, for a `FeatureSelectionNCARegression` object named `mdl`, you can access the `LossFunction` value using `mdl.ModelParameters.LossFunction`.

Data Types: `struct`

### **Lambda** — Regularization parameter

scalar

Regularization parameter used for training this model, stored as a scalar. For  $n$  observations, the best `Lambda` value that minimizes the generalization error of the NCA model is expected to be a multiple of  $1/n$ .

Data Types: `double`

### **FitMethod** — Name of the fitting method used to fit this model

character vector

Name of the fitting method used to fit this model, stored as a character vector. Possible values are:

- `'exact'` — Perform fitting using all of the data.

- `'none'` — No fitting. Use this option to evaluate the generalization error of the NCA model using the initial feature weights supplied in the call to `fsrnca`.
- `'average'` — The software divides the data into partitions (subsets), fits each partition using the `exact` method, and returns the average of the feature weights. You can specify the number of partitions using the `NumPartitions` name-value pair argument.

Data Types: `char`

**`solver`** — Name of the solver used to fit this model

character vector

Name of the solver used to fit this model, stored as one of the following:

- `'lbfgs'` — Limited memory Broyden-Fletcher-Goldfarb-Shanno (LBFGS) algorithm
- `'sgd'` — Stochastic gradient descent (SGD) algorithm
- `'minibatch-lbfgs'` — stochastic gradient descent with LBFGS algorithm applied to mini-batches

Data Types: `char`

**`GradientTolerance`** — Relative convergence tolerance on gradient norm

positive scalar

Relative convergence tolerance on the gradient norm for the `'lbfgs'` and `'minibatch-lbfgs'` solvers, stored as a positive scalar value.

Data Types: `double`

**`IterationLimit`** — Maximum number of iterations for optimization

positive integer

Maximum number of iterations for optimization, stored as a positive integer value.

Data Types: `double`

**`PassLimit`** — Maximum number of passes

positive integer

Maximum number of passes for `'sgd'` and `'minibatch-lbfgs'` solvers. Every pass processes all of the observations in the data.

Data Types: double

**InitialLearningRate** — Initial learning rate

positive real scalar

Initial learning rate for 'sgd' and 'minibatch-lbfgs' solvers. The learning rate decays over iterations starting at the value specified for `InitialLearningRate`.

Use the `NumTuningIterations` and `TuningSubsetSize` to control the automatic tuning of initial learning rate in the call to `fsrnca`.

Data Types: double

**verbose** — Verbosity level indicator

nonnegative integer

Verbosity level indicator, stored as a nonnegative integer. Possible values are:

- 0 — No convergence summary
- 1 — Convergence summary, including norm of gradient and objective function value
- >1 — More convergence information, depending on the fitting algorithm. When you use the 'minibatch-lbfgs' solver and verbosity level > 1, the convergence information includes the iteration log from intermediate minibatch LBFGS fits.

Data Types: double

**InitialFeatureWeights** — Initial feature weights

$p$ -by-1 vector of positive real scalars

Initial feature weights, stored as a  $p$ -by-1 vector of positive real scalars, where  $p$  is the number of predictors in  $X$ .

Data Types: double

**FeatureWeights** — Feature weights

$p$ -by-1 vector of real scalar values

Feature weights, stored as a  $p$ -by-1 vector of real scalar values, where  $p$  is the number of predictors in  $X$ .

For 'FitMethod' equal to 'average', `FeatureWeights` is a  $p$ -by- $m$  matrix, where  $m$  is the number of partitions specified via the 'NumPartitions' name-value pair argument in the call to `fsrnca`.

The absolute value of `FeatureWeights(k)` is a measure of the importance of predictor  $k$ . If `FeatureWeights(k)` is close to 0, then this indicates that predictor  $k$  does not influence the response in  $Y$ .

Data Types: `double`

### **FitInfo** — Fit information

structure

Fit information, stored as a structure with the following fields.

Field Name	Meaning
Iteration	Iteration index
Objective	Regularized objective function for minimization
UnregularizedObjective	Unregularized objective function for minimization
Gradient	Gradient of regularized objective function for minimization

- For classification, `UnregularizedObjective` represents the negative of the leave-one-out accuracy of the NCA classifier on the training data.
- For regression, `UnregularizedObjective` represents the leave-one-out loss between the true response and the predicted response when using the NCA regression model.
- For the 'lbfgs' solver, `Gradient` is the final gradient. For the 'sgd' and 'minibatch-lbfgs' solvers, `Gradient` is the final mini-batch gradient.
- If `FitMethod` is 'average', then `FitInfo` is an  $m$ -by-1 structure array, where  $m$  is the number of partitions specified via the 'NumPartitions' name-value pair argument.

You can access the fields of `FitInfo` using dot notation. For example, for a `FeatureSelectionNCARegression` object named `mdl`, you can access the `Objective` field using `mdl.FitInfo.Objective`.

Data Types: `struct`

**Mu — Predictor means**

$p$ -by-1 vector | []

Predictor means, stored as a  $p$ -by-1 vector for standardized training data. In this case, the `predict` method centers predictor matrix  $X$  by subtracting the respective element of  $\mu$  from every column.

If data is not standardized during training, then  $\mu$  is empty.

Data Types: `double`

**Sigma — Predictor standard deviations**

$p$ -by-1 vector | []

Predictor standard deviations, stored as a  $p$ -by-1 vector for standardized training data. In this case, the `predict` method scales predictor matrix  $X$  by dividing every column by the respective element of  $\sigma$  after centering the data using  $\mu$ .

If data is not standardized during training, then  $\sigma$  is empty.

Data Types: `double`

**x — Predictor values**

$n$ -by- $p$  matrix

Predictor values used to train this model, stored as an  $n$ -by- $p$  matrix.  $n$  is the number of observations and  $p$  is the number of predictor variables in the training data.

Data Types: `double`

**y — Response values**

numeric vector of size  $n$

Response values used to train this model, stored as a numeric vector of size  $n$ , where  $n$  is the number of observations.

Data Types: `double`

**w — Observation weights**

numeric vector of size  $n$

Observation weights used to train this model, stored as a numeric vector of size  $n$ . The sum of observation weights is  $n$ .

Data Types: `double`

## Methods

- `loss` Evaluate accuracy of learned feature weights on test data
- `predict` Predict responses using neighborhood component analysis (NCA) regression model
- `refit` Refit neighborhood component analysis (NCA) model for regression

## Examples

### Explore `FeatureSelectionNCARegression` Object

Load the sample data.

```
load imports-85
```

The first 15 columns contain the continuous predictor variables, whereas the 16th column contains the response variable, which is the price of a car. Define the variables for the neighborhood component analysis model.

```
Predictors = X(:,1:15);  
Y = X(:,16);
```

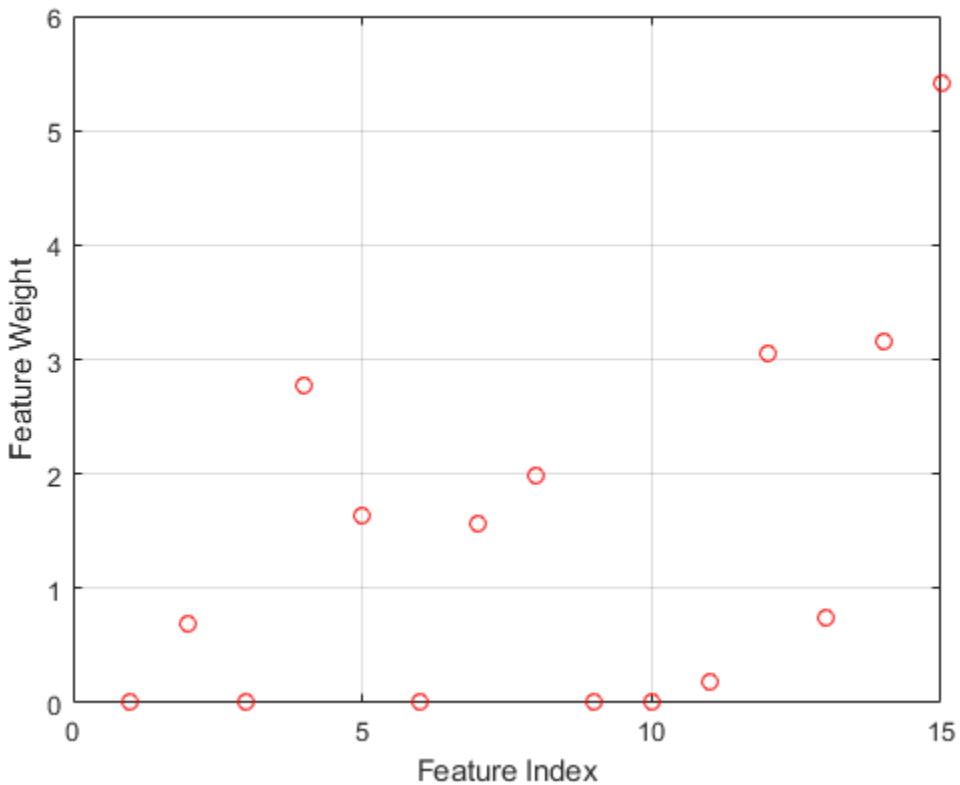
Fit a neighborhood component analysis (NCA) model for regression to detect the relevant features.

```
mdl = fsrnca(Predictors,Y);
```

The returned NCA model, `mdl`, is a `FeatureSelectionNCARegression` object. This object stores information about the training data, model, and optimization. You can access the object properties, such as the feature weights, using dot notation.

Plot the feature weights.

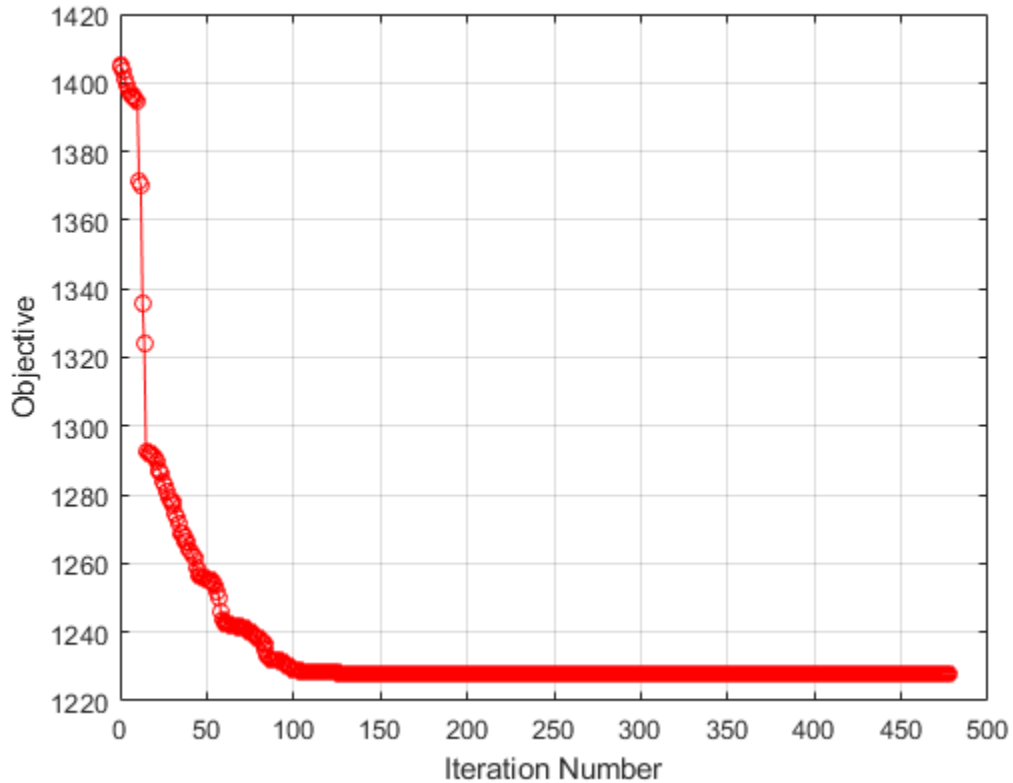
```
figure()
plot mdl.FeatureWeights, 'ro'
xlabel('Feature Index')
ylabel('Feature Weight')
grid on
```



The weights of the irrelevant features are zero. The 'Verbose', 1 option in the call to `fsrnca` displays the optimization information on the command line. You can also visualize the optimization process by plotting the objective function versus the iteration number.

```
figure()
plot mdl.FitInfo.Iteration, mdl.FitInfo.Objective, 'ro-'
grid on
```

```
xlabel('Iteration Number')  
ylabel('Objective')
```



The `ModelParameters` property is a struct that contains more information about the model. You can access the fields of this property using dot notation. For example, see if the data was standardized or not.

```
mdl.ModelParameters.Standardize
```

```
ans =
```

```
logical
```



0

0 means that the data was not standardized before fitting the NCA model. You can standardize the predictors when they are on very different scales using the 'Standardize', 1 name-value pair argument in the call to `fsrnca`.

## Copy Semantics

Value. To learn how value classes affect copy operations, see [Copying Objects \(MATLAB\)](#).

## See Also

`fsrnca` | `loss` | `predict` | `refit`

## Topics

[Class Attributes \(MATLAB\)](#)

[Property Attributes \(MATLAB\)](#)

**Introduced in R2016b**

## ncfcdf

Noncentral  $F$  cumulative distribution function

### Syntax

```
p = ncfcdf(x, nu1, nu2, delta)
p = ncfcdf(x, nu1, nu2, delta, 'upper')
```

### Description

`p = ncfcdf(x, nu1, nu2, delta)` computes the noncentral  $F$  cdf at each value in `x` using the corresponding numerator degrees of freedom in `nu1`, denominator degrees of freedom in `nu2`, and positive noncentrality parameters in `delta`. `nu1`, `nu2`, and `delta` can be vectors, matrices, or multidimensional arrays that have the same size, which is also the size of `p`. A scalar input for `x`, `nu1`, `nu2`, or `delta` is expanded to a constant array with the same dimensions as the other inputs.

`p = ncfcdf(x, nu1, nu2, delta, 'upper')` returns the complement of the noncentral  $F$  cdf at each value in `x`, using an algorithm that more accurately computes the extreme upper tail probabilities.

The noncentral  $F$  cdf is

$$F(x | v_1, v_2, \delta) = \sum_{j=0}^{\infty} \left( \frac{\left(\frac{1}{2}\delta\right)^j}{j!} e^{-\frac{\delta}{2}} \right) I\left(\frac{v_1 \cdot x}{v_2 + v_1 \cdot x} \middle| \frac{v_1}{2} + j, \frac{v_2}{2}\right)$$

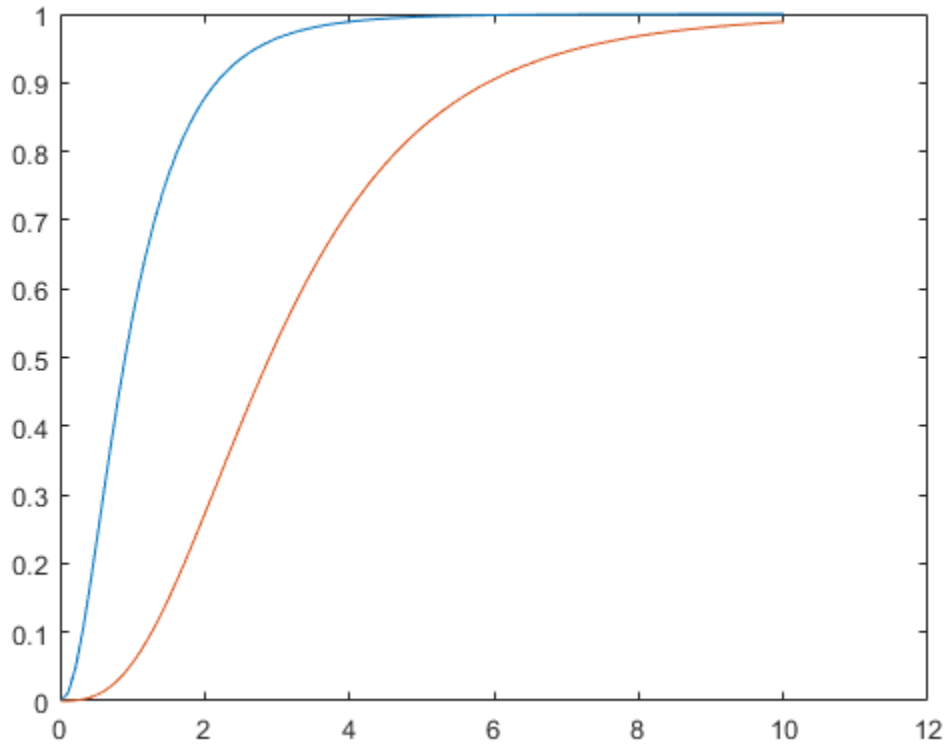
where  $I(x | a, b)$  is the incomplete beta function with parameters  $a$  and  $b$ .

### Examples

**Compute Noncentral F Distribution cdf**

Compare the noncentral  $F$  cdf with  $\delta = 10$  to the  $F$  cdf with the same number of numerator and denominator degrees of freedom (5 and 20 respectively).

```
x = (0.01:0.1:10.01)';  
p1 = ncfcdf(x,5,20,10);  
p = fcdf(x,5,20);  
plot(x,p,'-',x,p1,'-')
```



## References

- [1] Johnson, N., and S. Kotz. *Distributions in Statistics: Continuous Univariate Distributions-2*. Hoboken, NJ: John Wiley & Sons, Inc., 1970, pp. 189–200.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`cdf` | `ncfinv` | `ncfpdf` | `ncfrnd` | `ncfstat`

### Topics

“Noncentral F Distribution” on page B-128

Introduced before R2006a

## ncfinv

Noncentral  $F$  inverse cumulative distribution function

## Syntax

```
X = ncfinv(P, NU1, NU2, DELTA)
```

## Description

`X = ncfinv(P, NU1, NU2, DELTA)` returns the inverse of the noncentral  $F$  cdf with numerator degrees of freedom `NU1`, denominator degrees of freedom `NU2`, and positive noncentrality parameter `DELTA` for the corresponding probabilities in `P`. `P`, `NU1`, `NU2`, and `DELTA` can be vectors, matrices, or multidimensional arrays that all have the same size, which is also the size of `X`. A scalar input for `P`, `NU1`, `NU2`, or `DELTA` is expanded to a constant array with the same dimensions as the other inputs.

## Examples

One hypothesis test for comparing two sample variances is to take their ratio and compare it to an  $F$  distribution. If the numerator and denominator degrees of freedom are 5 and 20 respectively, then you reject the hypothesis that the first variance is equal to the second variance if their ratio is less than that computed below.

```
critical = finv(0.95, 5, 20)
critical =
    2.7109
```

Suppose the truth is that the first variance is twice as big as the second variance. How likely is it that you would detect this difference?

```
prob = 1 - ncfcdf(critical, 5, 20, 2)
prob =
    0.1297
```

If the true ratio of variances is 2, what is the typical (median) value you would expect for the  $F$  statistic?

```
ncfinv(0.5,5,20,2)
ans =
    1.2786
```

## References

- [1] Evans, M., N. Hastings, and B. Peacock. *Statistical Distributions*. Hoboken, NJ: Wiley-Interscience, 2000.
- [2] Johnson, N., and S. Kotz. *Distributions in Statistics: Continuous Univariate Distributions-2*. Hoboken, NJ: John Wiley & Sons, Inc., 1970, pp. 189–200.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`icdf` | `ncfcdf` | `ncfpdf` | `ncfrnd` | `ncfstat`

### Topics

“Noncentral F Distribution” on page B-128

Introduced before R2006a

## ncfpdf

Noncentral  $F$  probability density function

### Syntax

```
Y = ncfpdf(X, NU1, NU2, DELTA)
```

### Description

`Y = ncfpdf(X, NU1, NU2, DELTA)` computes the noncentral  $F$  pdf at each of the values in  $X$  using the corresponding numerator degrees of freedom in  $NU1$ , denominator degrees of freedom in  $NU2$ , and positive noncentrality parameters in  $DELTA$ .  $X$ ,  $NU1$ ,  $NU2$ , and  $DELTA$  can be vectors, matrices, or multidimensional arrays that all have the same size, which is also the size of  $Y$ . A scalar input for  $NU1$ ,  $NU2$ , or  $DELTA$  is expanded to a constant array with the same dimensions as the other inputs.

The  $F$  distribution is a special case of the noncentral  $F$  where  $\delta = 0$ . As  $\delta$  increases, the distribution flattens like the plot in the example.

### Examples

#### Compute Noncentral $F$ Distribution pdf

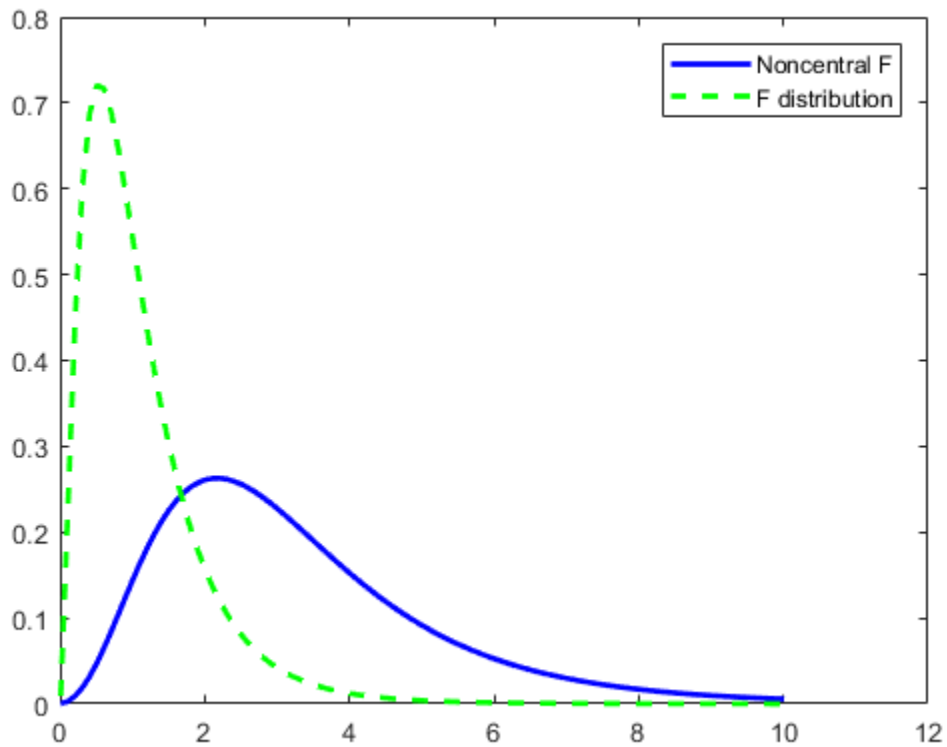
Compute the pdf of a noncentral  $F$  distribution with degrees of freedom  $NU1 = 5$  and  $NU2 = 20$ , and noncentrality parameter  $DELTA = 10$ . For comparison, also compute the pdf of an  $F$  distribution with the same degrees of freedom.

```
x = (0.01:0.1:10.01)';  
p1 = ncfpdf(x, 5, 20, 10);  
p = fpdf(x, 5, 20);
```

Plot the pdf of the noncentral  $F$  distribution and the pdf of the  $F$  distribution on the same figure.



```
figure;  
plot(x,p1,'b-','LineWidth',2)  
hold on  
plot(x,p,'g--','LineWidth',2)  
legend('Noncentral F','F distribution')
```



## References

- [1] Johnson, N., and S. Kotz. *Distributions in Statistics: Continuous Univariate Distributions-2*. Hoboken, NJ: John Wiley & Sons, Inc., 1970, pp. 189–200.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`ncfcdf` | `ncfinv` | `ncfrnd` | `ncfstat` | `pdf`

### Topics

“Noncentral F Distribution” on page B-128

Introduced before R2006a

# ncfrnd

Noncentral  $F$  random numbers

## Syntax

```
R = ncfrnd(NU1, NU2, DELTA)
R = ncfrnd(NU1, NU2, DELTA, m, n, ...)
R = ncfrnd(NU1, NU2, DELTA, [m, n, ...])
```

## Description

`R = ncfrnd(NU1, NU2, DELTA)` returns a matrix of random numbers chosen from the noncentral  $F$  distribution with corresponding numerator degrees of freedom in `NU1`, denominator degrees of freedom in `NU2`, and positive noncentrality parameters in `DELTA`. `NU1`, `NU2`, and `DELTA` can be vectors, matrices, or multidimensional arrays that have the same size, which is also the size of `R`. A scalar input for `NU1`, `NU2`, or `DELTA` is expanded to a constant matrix with the same dimensions as the other inputs.

`R = ncfrnd(NU1, NU2, DELTA, m, n, ...)` or `R = ncfrnd(NU1, NU2, DELTA, [m, n, ...])` generates an `m`-by-`n`-by-... array. The `NU1`, `NU2`, `DELTA` parameters can each be scalars or arrays of the same size as `R`.

## Examples

Compute six random numbers from a noncentral  $F$  distribution with 10 numerator degrees of freedom, 100 denominator degrees of freedom and a noncentrality parameter,  $\delta$ , of 4.0. Compare this to the  $F$  distribution with the same degrees of freedom.

```
r = ncfrnd(10, 100, 4, 1, 6)
r =
    2.5995    0.8824    0.8220    1.4485    1.4415    1.4864

r1 = frnd(10, 100, 1, 6)
r1 =
    0.9826    0.5911    1.0967    0.9681    2.0096    0.6598
```

## References

- [1] Johnson, N., and S. Kotz. *Distributions in Statistics: Continuous Univariate Distributions-2*. Hoboken, NJ: John Wiley & Sons, Inc., 1970, pp. 189–200.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

The generated code can return a different sequence of numbers than MATLAB if either of the following is true:

- The output is nonscalar.
- An input parameter is invalid for the distribution.

### See Also

`ncfcdf` | `ncfinv` | `ncfpdf` | `ncfstat` | `random`

### Topics

“Noncentral F Distribution” on page B-128

Introduced before R2006a

# ncfstat

Noncentral  $F$  mean and variance

## Syntax

`[M,V] = ncfstat (NU1,NU2,DELTA)`

## Description

`[M,V] = ncfstat (NU1,NU2,DELTA)` returns the mean of and variance for the noncentral  $F$  pdf with corresponding numerator degrees of freedom in NU1, denominator degrees of freedom in NU2, and positive noncentrality parameters in DELTA. NU1, NU2, and DELTA can be vectors, matrices, or multidimensional arrays that all have the same size, which is also the size of M and V. A scalar input for NU1, NU2, or DELTA is expanded to a constant array with the same dimensions as the other input.

The mean of the noncentral  $F$  distribution with parameters  $\nu_1$ ,  $\nu_2$ , and  $\delta$  is

$$\frac{\nu_2(\delta + \nu_1)}{\nu_1(\nu_2 - 2)}$$

where  $\nu_2 > 2$ .

The variance is

$$2 \left( \frac{\nu_2}{\nu_1} \right)^2 \left[ \frac{(\delta + \nu_1)^2 + (2\delta + \nu_1)(\nu_2 - 2)}{(\nu_2 - 2)^2(\nu_2 - 4)} \right]$$

where  $\nu_2 > 4$ .

## Examples

```
[m,v]= ncfstat(10,100,4)
m =
```

1.4286  
v =  
0.4252

## References

- [1] Evans, M., N. Hastings, and B. Peacock. *Statistical Distributions*. 2nd ed., Hoboken, NJ: John Wiley & Sons, Inc., 1993, pp. 73–74.
- [2] Johnson, N., and S. Kotz. *Distributions in Statistics: Continuous Univariate Distributions-2*. Hoboken, NJ: John Wiley & Sons, Inc., 1970, pp. 189–200.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`ncfcdf` | `ncfinv` | `ncfpdf` | `ncfrnd`

### Topics

“Noncentral F Distribution” on page B-128

Introduced before R2006a

## NClasses property

**Class:** NaiveBayes

Number of classes

### Description

The `NClasses` property specifies the number of classes in the grouping variable used to create the Naive Bayes classifier.

## NumComponents property

**Class:** gmdistribution

Number  $k$  of mixture components

### Description

The number  $k$  of mixture components.



# nctcdf

Noncentral  $t$  cumulative distribution function

## Syntax

```
p = nctcdf(x,nu,delta)
p = nctcdf(x,nu,delta,'upper')
```

## Description

`p = nctcdf(x,nu,delta)` computes the noncentral  $t$  cdf at each value in `x` using the corresponding degrees of freedom in `nu` and noncentrality parameters in `delta`. `x`, `nu`, and `delta` can be vectors, matrices, or multidimensional arrays that have the same size, which is also the size of `p`. A scalar input for `x`, `nu`, or `delta` is expanded to a constant array with the same dimensions as the other inputs.

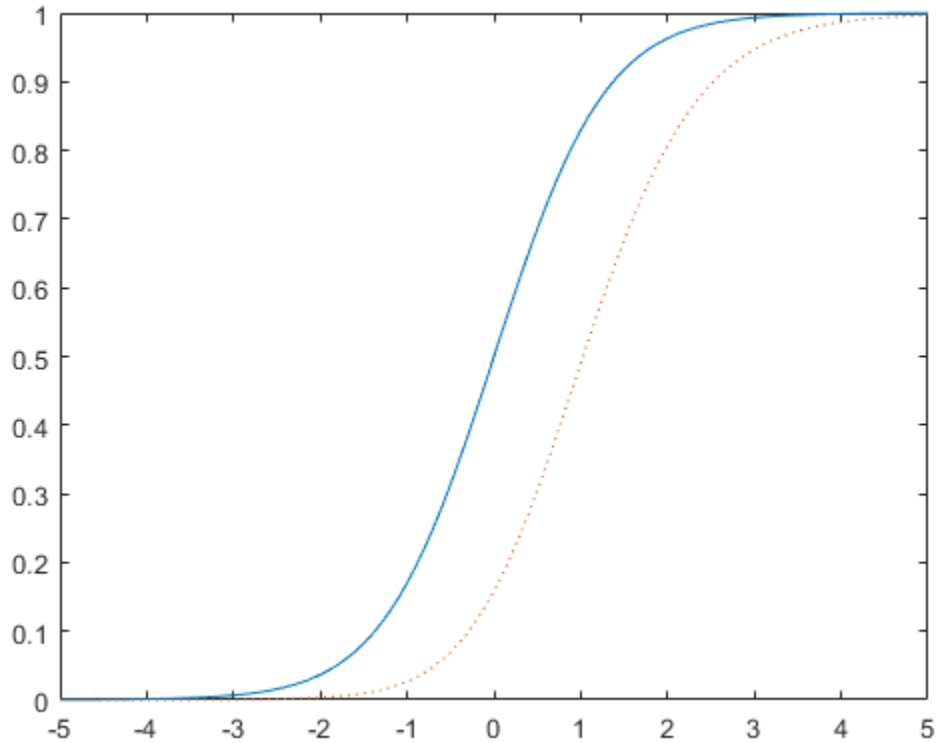
`p = nctcdf(x,nu,delta,'upper')` returns the complement of the noncentral  $t$  cdf at each value in `x`, using an algorithm that more accurately computes the extreme upper tail probabilities.

## Examples

### Compute Noncentral $t$ Distribution cdf

Compare the noncentral  $t$  cdf with `DELTA = 1` to the  $t$  cdf with the same number of degrees of freedom (10).

```
x = (-5:0.1:5)';
p1 = nctcdf(x,10,1);
p = tcdf(x,10);
plot(x,p,'-',x,p1,':')
```



## References

- [1] Evans, M., N. Hastings, and B. Peacock. *Statistical Distributions*. 2nd ed., Hoboken, NJ: John Wiley & Sons, Inc., 1993, pp. 147–148.
- [2] Johnson, N., and S. Kotz. *Distributions in Statistics: Continuous Univariate Distributions-2*. Hoboken, NJ: John Wiley & Sons, Inc., 1970, pp. 201–219.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`cdf` | `nctinv` | `nctpdf` | `nctrnd` | `nctstat`

### Topics

“Noncentral t Distribution” on page B-131

Introduced before R2006a

## nctinv

Noncentral  $t$  inverse cumulative distribution function

## Syntax

```
X = nctinv(P,NU,DELTA)
```

## Description

`X = nctinv(P,NU,DELTA)` returns the inverse of the noncentral  $t$  cdf with `NU` degrees of freedom and noncentrality parameter `DELTA` for the corresponding probabilities in `P`. `P`, `NU`, and `DELTA` can be vectors, matrices, or multidimensional arrays that all have the same size, which is also the size of `X`. A scalar input for `P`, `NU`, or `DELTA` is expanded to a constant array with the same dimensions as the other inputs.

## Examples

```
x = nctinv([0.1 0.2],10,1)
x =
    -0.2914    0.1618
```

## References

- [1] Evans, M., N. Hastings, and B. Peacock. *Statistical Distributions*. 2nd ed., Hoboken, NJ: John Wiley & Sons, Inc., 1993, pp. 147–148.
- [2] Johnson, N., and S. Kotz. *Distributions in Statistics: Continuous Univariate Distributions-2*. Hoboken, NJ: John Wiley & Sons, Inc., 1970, pp. 201–219.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`icdf` | `nctcdf` | `nctpdf` | `nctrnd` | `nctstat`

### Topics

“Noncentral t Distribution” on page B-131

Introduced before R2006a

## nctpdf

Noncentral  $t$  probability density function

## Syntax

```
Y = nctpdf(X,V,DELTA)
```

## Description

`Y = nctpdf(X,V,DELTA)` computes the noncentral  $t$  pdf at each of the values in `X` using the corresponding degrees of freedom in `V` and noncentrality parameters in `DELTA`. Vector or matrix inputs for `X`, `V`, and `DELTA` must have the same size, which is also the size of `Y`. A scalar input for `X`, `V`, or `DELTA` is expanded to a constant matrix with the same dimensions as the other inputs.

## Examples

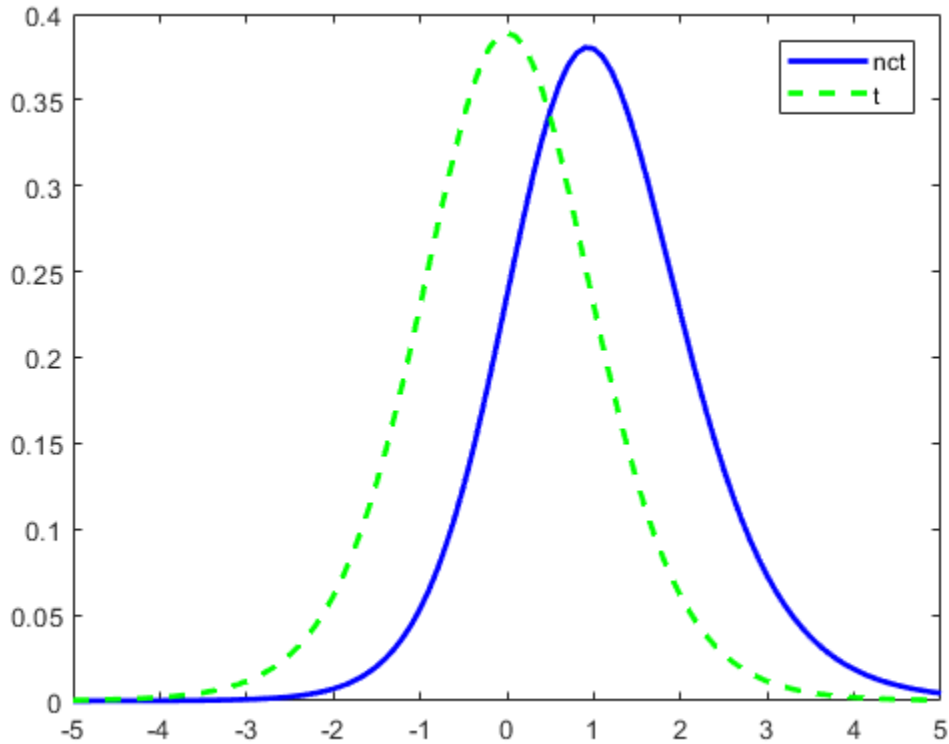
### Compute Noncentral $t$ Distribution pdf

Compute the pdf of a noncentral  $t$  distribution with degrees of freedom  $V = 10$  and noncentrality parameter  $DELTA = 1$ . For comparison, also compute the pdf of a  $t$  distribution with the same degrees of freedom.

```
x = (-5:0.1:5)';  
nct = nctpdf(x,10,1);  
t = tpdf(x,10);
```

Plot the pdf of the noncentral  $t$  distribution and the pdf of the  $t$  distribution on the same figure.

```
plot(x,nct,'b-','LineWidth',2)  
hold on  
plot(x,t,'g--','LineWidth',2)  
legend('nct','t')
```



## References

- [1] Evans, M., N. Hastings, and B. Peacock. *Statistical Distributions*. 2nd ed., Hoboken, NJ: John Wiley & Sons, Inc., 1993, pp. 147–148.
- [2] Johnson, N., and S. Kotz. *Distributions in Statistics: Continuous Univariate Distributions-2*. Hoboken, NJ: John Wiley & Sons, Inc., 1970, pp. 201–219.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`nctcdf` | `nctinv` | `nctrnd` | `nctstat` | `pdf`

### Topics

“Noncentral t Distribution” on page B-131

Introduced before R2006a



# nctrnd

Noncentral  $t$  random numbers

## Syntax

```
R = nctrnd(V, DELTA)
R = nctrnd(V, DELTA, m, n, ...)
R = nctrnd(V, DELTA, [m, n, ...])
```

## Description

`R = nctrnd(V, DELTA)` returns a matrix of random numbers chosen from the noncentral T distribution using the corresponding degrees of freedom in `V` and noncentrality parameters in `DELTA`. `V` and `DELTA` can be vectors, matrices, or multidimensional arrays. A scalar input for `V` or `DELTA` is expanded to a constant array with the same dimensions as the other input.

`R = nctrnd(V, DELTA, m, n, ...)` or `R = nctrnd(V, DELTA, [m, n, ...])` generates an `m-by-n-by-...` array. The `V`, `DELTA` parameters can each be scalars or arrays of the same size as `R`.

## Examples

```
nctrnd(10, 1, 5, 1)
ans =
    1.6576
    1.0617
    1.4491
    0.2930
    3.6297
```

## References

- [1] Evans, M., N. Hastings, and B. Peacock. *Statistical Distributions*. 2nd ed., Hoboken, NJ: John Wiley & Sons, Inc., 1993, pp. 147–148.
- [2] Johnson, N., and S. Kotz. *Distributions in Statistics: Continuous Univariate Distributions-2*. Hoboken, NJ: John Wiley & Sons, Inc., 1970, pp. 201–219.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

The generated code can return a different sequence of numbers than MATLAB if either of the following is true:

- The output is nonscalar.
- An input parameter is invalid for the distribution.

### See Also

`nctcdf` | `nctinv` | `nctpdf` | `nctstat` | `random`

### Topics

“Noncentral t Distribution” on page B-131

**Introduced before R2006a**

# nctstat

Noncentral  $t$  mean and variance

## Syntax

`[M,V] = nctstat(NU,DELTA)`

## Description

`[M,V] = nctstat(NU,DELTA)` returns the mean of and variance for the noncentral  $t$  pdf with `NU` degrees of freedom and noncentrality parameter `DELTA`. `NU` and `DELTA` can be vectors, matrices, or multidimensional arrays that all have the same size, which is also the size of `M` and `V`. A scalar input for `NU` or `DELTA` is expanded to a constant array with the same dimensions as the other input.

The mean of the noncentral  $t$  distribution with parameters  $\nu$  and  $\delta$  is

$$\frac{\delta(\nu/2)^{1/2}\Gamma((\nu-1)/2)}{\Gamma(\nu/2)}$$

where  $\nu > 1$ .

The variance is

$$\frac{\nu}{(\nu-2)}(1+\delta^2) - \frac{\nu}{2}\delta^2 \left[ \frac{\Gamma((\nu-1)/2)}{\Gamma(\nu/2)} \right]^2$$

where  $\nu > 2$ .

## Examples

`[m,v] = nctstat(10,1)`

`m =`

1.0837

v =  
1.3255

## References

- [1] Evans, M., N. Hastings, and B. Peacock. *Statistical Distributions*. 2nd ed., Hoboken, NJ: John Wiley & Sons, Inc., 1993, pp. 147–148.
- [2] Johnson, N., and S. Kotz. *Distributions in Statistics: Continuous Univariate Distributions-2*. Hoboken, NJ: John Wiley & Sons, Inc., 1970, pp. 201–219.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`nctcdf` | `nctinv` | `nctpdf` | `nctrnd`

### Topics

“Noncentral t Distribution” on page B-131

Introduced before R2006a

## ncx2cdf

Noncentral chi-square cumulative distribution function

### Syntax

```
p = ncx2cdf(x,v,delta)
p = ncx2cdf(x,v,delta,'upper')
```

### Description

`p = ncx2cdf(x,v,delta)` computes the noncentral chi-square cdf at each value in `x` using the corresponding degrees of freedom in `v` and positive noncentrality parameters in `delta`. `x`, `v`, and `delta` can be vectors, matrices, or multidimensional arrays that all have the same size, which is also the size of `p`. A scalar input for `x`, `v`, or `delta` is expanded to a constant array with the same dimensions as the other inputs.

`p = ncx2cdf(x,v,delta,'upper')` returns the complement of the noncentral chi-square cdf at each value in `x`, using an algorithm that more accurately computes the extreme upper tail probabilities.

Some texts refer to this distribution as the generalized Rayleigh, Rayleigh-Rice, or Rice distribution.

The noncentral chi-square cdf is

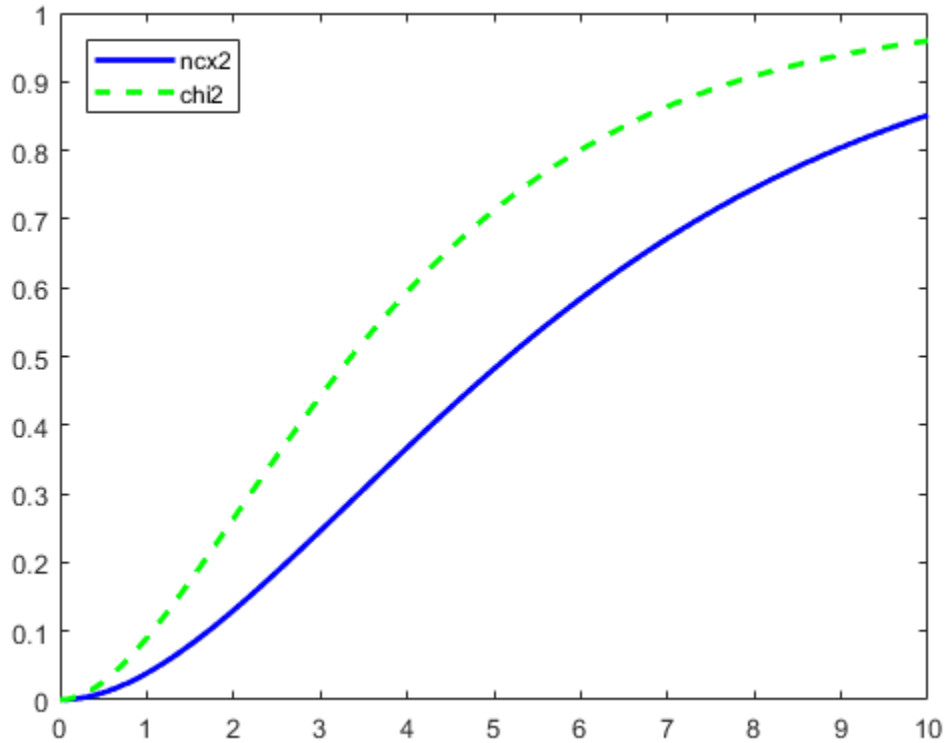
$$F(x|v,\delta) = \sum_{j=0}^{\infty} \left( \frac{\left(\frac{1}{2}\delta\right)^j}{j!} e^{-\frac{\delta}{2}} \right) \Pr\left[\chi_{v+2j}^2 \leq x\right]$$

### Examples

### Compute Noncentral Chi-Square cdf

Compare the noncentral chi-square cdf with  $\text{DELTA} = 2$  to the chi-square cdf with the same number of degrees of freedom (4):

```
x = (0:0.1:10)';  
ncx2 = ncx2cdf(x,4,2);  
chi2 = chi2cdf(x,4);  
  
plot(x,ncx2,'b-','LineWidth',2)  
hold on  
plot(x,chi2,'g--','LineWidth',2)  
legend('ncx2','chi2','Location','NW')
```



## References

- [1] Johnson, N., and S. Kotz. *Distributions in Statistics: Continuous Univariate Distributions-2*. Hoboken, NJ: John Wiley & Sons, Inc., 1970, pp. 130–148.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`cdf` | `ncx2inv` | `ncx2pdf` | `ncx2rnd` | `ncx2stat`

### Topics

“Noncentral Chi-Square Distribution” on page B-125

Introduced before R2006a



## ncx2inv

Noncentral chi-square inverse cumulative distribution function

### Syntax

```
X = ncx2inv(P,V,DELTA)
```

### Description

`X = ncx2inv(P,V,DELTA)` returns the inverse of the noncentral chi-square cdf using the corresponding degrees of freedom in `V` and positive noncentrality parameters in `DELTA`, at the corresponding probabilities in `P`. `P`, `V`, and `DELTA` can be vectors, matrices, or multidimensional arrays that all have the same size, which is also the size of `X`. A scalar input for `P`, `V`, or `DELTA` is expanded to a constant array with the same dimensions as the other inputs.

### Examples

```
ncx2inv([0.01 0.05 0.1],4,2)
ans =
    0.4858    1.1498    1.7066
```

### Algorithms

`ncx2inv` uses Newton's method to converge to the solution.

### References

- [1] Evans, M., N. Hastings, and B. Peacock. *Statistical Distributions*. 2nd ed., Hoboken, NJ: John Wiley & Sons, Inc., 1993, pp. 50–52.

[2] Johnson, N., and S. Kotz. *Distributions in Statistics: Continuous Univariate Distributions-2*. Hoboken, NJ: John Wiley & Sons, Inc., 1970, pp. 130–148.

## See Also

`icdf` | `ncx2cdf` | `ncx2pdf` | `ncx2rnd` | `ncx2stat`

## Topics

“Noncentral Chi-Square Distribution” on page B-125

**Introduced before R2006a**

# ncx2pdf

Noncentral chi-square probability density function

## Syntax

```
Y = ncx2pdf(X,V,DELTA)
```

## Description

`Y = ncx2pdf(X,V,DELTA)` computes the noncentral chi-square pdf at each of the values in `X` using the corresponding degrees of freedom in `V` and positive noncentrality parameters in `DELTA`. Vector or matrix inputs for `X`, `V`, and `DELTA` must have the same size, which is also the size of `Y`. A scalar input for `X`, `V`, or `DELTA` is expanded to a constant array with the same dimensions as the other inputs.

Some texts refer to this distribution as the generalized Rayleigh, Rayleigh-Rice, or Rice distribution.

## Examples

### Compute Noncentral Chi-Square Distribution pdf

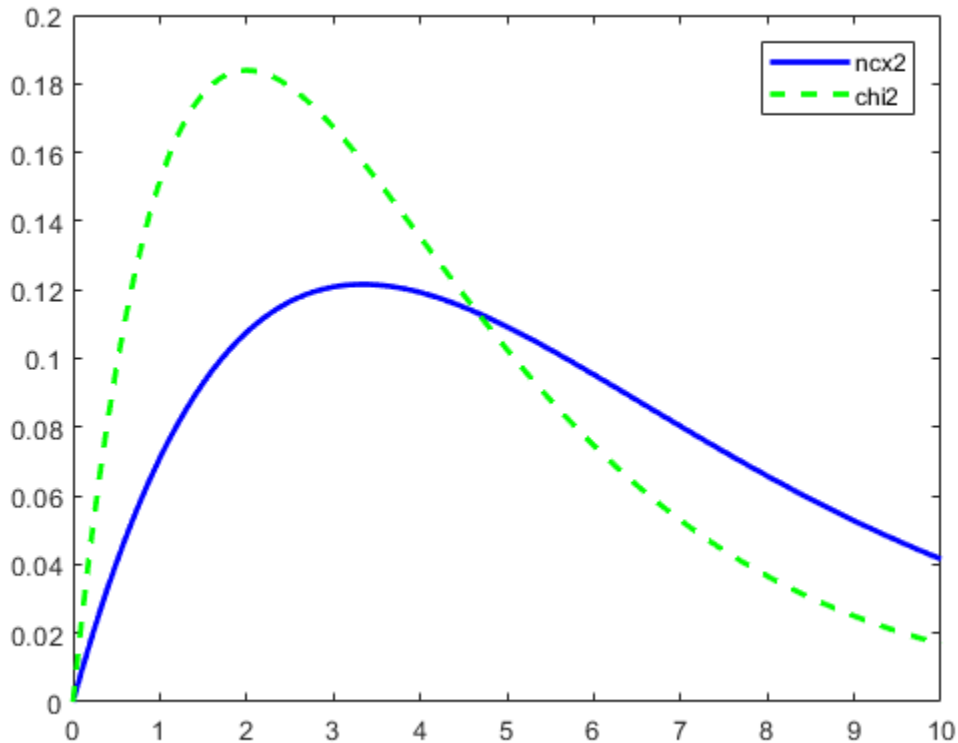
Compute the pdf of a noncentral chi-square distribution with degrees of freedom `V = 4` and noncentrality parameter `DELTA = 2`. For comparison, also compute the pdf of a chi-square distribution with the same degrees of freedom.

```
x = (0:0.1:10)';  
ncx2 = ncx2pdf(x,4,2);  
chi2 = chi2pdf(x,4);
```

Plot the pdf of the noncentral chi-square distribution on the same figure as the pdf of the chi-square distribution.

```
figure;  
plot(x,ncx2,'b-','LineWidth',2)
```

```
hold on
plot(x,chi2,'g--','LineWidth',2)
legend('ncx2','chi2')
```



## References

- [1] Johnson, N., and S. Kotz. *Distributions in Statistics: Continuous Univariate Distributions-2*. Hoboken, NJ: John Wiley & Sons, Inc., 1970, pp. 130–148.

## See Also

[ncx2cdf](#) | [ncx2inv](#) | [ncx2rnd](#) | [ncx2stat](#) | [pdf](#)

## Topics

“Noncentral Chi-Square Distribution” on page B-125

Introduced before R2006a

## ncx2rnd

Noncentral chi-square random numbers

### Syntax

```
R = ncx2rnd(V, DELTA)
R = ncx2rnd(V, DELTA, m, n, ...)
R = ncx2rnd(V, DELTA, [m, n, ...])
```

### Description

`R = ncx2rnd(V, DELTA)` returns a matrix of random numbers chosen from the noncentral chi-square distribution using the corresponding degrees of freedom in `V` and positive noncentrality parameters in `DELTA`. `V` and `DELTA` can be vectors, matrices, or multidimensional arrays that have the same size, which is also the size of `R`. A scalar input for `V` or `DELTA` is expanded to a constant array with the same dimensions as the other input.

`R = ncx2rnd(V, DELTA, m, n, ...)` or `R = ncx2rnd(V, DELTA, [m, n, ...])` generates an `m`-by-`n`-by-... array. The `V`, `DELTA` parameters can each be scalars or arrays of the same size as `R`.

### Examples

```
ncx2rnd(4, 2, 6, 3)
ans =
    6.8552    5.9650   11.2961
    5.2631    4.2640    5.9495
    9.1939    6.7162    3.8315
   10.3100    4.4828    7.1653
    2.1142    1.9826    4.6400
    3.8852    5.3999    0.9282
```

## References

- [1] Evans, M., N. Hastings, and B. Peacock. *Statistical Distributions*. 2nd ed., Hoboken, NJ: John Wiley & Sons, Inc., 1993, pp. 50–52.
- [2] Johnson, N., and S. Kotz. *Distributions in Statistics: Continuous Univariate Distributions-2*. Hoboken, NJ: John Wiley & Sons, Inc., 1970, pp. 130–148.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

The generated code can return a different sequence of numbers than MATLAB if either of the following is true:

- The output is nonscalar.
- An input parameter is invalid for the distribution.

### See Also

`ncx2cdf` | `ncx2inv` | `ncx2pdf` | `ncx2stat` | `random`

### Topics

“Noncentral Chi-Square Distribution” on page B-125

Introduced before R2006a

## ncx2stat

Noncentral chi-square mean and variance

### Syntax

```
[M,V] = ncx2stat(NU,DELTA)
```

### Description

`[M,V] = ncx2stat(NU,DELTA)` returns the mean of and variance for the noncentral chi-square pdf with `NU` degrees of freedom and noncentrality parameter `DELTA`. `NU` and `DELTA` can be vectors, matrices, or multidimensional arrays that all have the same size, which is also the size of `M` and `V`. A scalar input for `NU` or `DELTA` is expanded to a constant array with the same dimensions as the other input.

The mean of the noncentral chi-square distribution with parameters  $v$  and  $\delta$  is  $v+\delta$ , and the variance is  $2(v+2\delta)$ .

### Examples

```
[m,v] = ncx2stat(4,2)
m =
     6
v =
    16
```

### References

- [1] Evans, M., N. Hastings, and B. Peacock. *Statistical Distributions*. 2nd ed., Hoboken, NJ: John Wiley & Sons, Inc., 1993, pp. 50–52.
- [2] Johnson, N., and S. Kotz. *Distributions in Statistics: Continuous Univariate Distributions-2*. Hoboken, NJ: John Wiley & Sons, Inc., 1970, pp. 130–148.



## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

[ncx2cdf](#) | [ncx2inv](#) | [ncx2pdf](#) | [ncx2rnd](#)

### Topics

“Noncentral Chi-Square Distribution” on page B-125

Introduced before R2006a

## NumVariables property

**Class:** `gmdistribution`

Dimension  $d$  of multivariate Gaussian distributions

### Description

The dimension  $d$  of the multivariate Gaussian distributions.

# ndims

**Class:** dataset

Number of dimensions of dataset array

---

**Note** The `dataset` data type might be removed in a future release. To work with heterogeneous data, use the MATLAB® `table` data type instead. See MATLAB `table` documentation for more information.

---

## Syntax

```
n = ndims(A)
```

## Description

`n = ndims(A)` returns the number of dimensions in the dataset A. The number of dimensions in an array is always 2.

## See Also

`size`

## ndims

**Class:** grandset

Number of dimensions in matrix

## Syntax

```
n = ndims(p)
```

## Description

`n = ndims(p)` returns the number of dimensions in the matrix that is created by the syntax `p(:, :)`. Since this is always a 2-D matrix, `n` is always equal to 2.

## See Also

`grandset` | `size`

## NDims property

**Class:** NaiveBayes

Number of dimensions

### Description

The `NDims` property specifies the number of dimensions, which is equal to the number of features in the training data used to create the Naive Bayes classifier.

## ne

**Class:** grandstream

Not equal relation for handles

## Syntax

```
h1 ~= h2
```

## Description

Handles are equal if they are handles for the same object and are unequal otherwise.

`h1 ~= h2` performs element-wise comparisons between handle arrays `h1` and `h2`. `h1` and `h2` must be of the same dimensions unless one is a scalar. The result is a logical array of the same dimensions, where each element is an element-wise `~=` result.

If one of `h1` or `h2` is scalar, scalar expansion is performed and the result will match the dimensions of the array that is not scalar.

`tf = ne(h1, h2)` stores the result in a logical array of the same dimensions.

## See Also

`eq` | `ge` | `gt` | `le` | `lt` | `grandstream`

# negloglik

Negative log likelihood of probability distribution

## Syntax

```
nll = negloglik(pd)
```

## Description

`nll = negloglik(pd)` returns the value of the negative loglikelihood function for the data used to fit the probability distribution `pd`.

## Examples

### Negative Log Likelihood for a Fitted Distribution

Load the sample data.

```
load carsmall
```

Create a Weibull distribution object by fitting it to the mile per gallon (MPG) data.

```
pd = fitdist(MPG, 'Weibull')
```

```
pd =
```

```
WeibullDistribution
```

```
Weibull distribution
```

```
A = 26.5079    [24.8333, 28.2954]
```

```
B = 3.27193   [2.79441, 3.83104]
```

Compute the negative log likelihood for the fitted Weibull distribution.

```
wnll = negloglik(pd)
```

```
wnll =  
  327.4942
```

## Input Arguments

**pd** — **Probability distribution**  
probability distribution object

Probability distribution, specified as a probability distribution object. Create a probability distribution object with specified parameter values using `makedist`. Alternatively, create a probability distribution object by fitting it to data using `fitdist` or the Distribution Fitter app.

## Output Arguments

**nll** — **Negative log likelihood**  
scalar value

Negative log likelihood value for the data used to fit the distribution, returned as a scalar value.

## See Also

`distributionFitter` | `fitdist` | `makedist`

## Topics

“Working with Probability Distributions” on page 5-3

**Introduced before R2006a**



# negloglik

**Class:** prob.ToolboxFittableParametricDistribution

**Package:** prob

Negative log likelihood of probability distribution object

## Syntax

```
nll = negloglik(pd)
```

## Description

`nll = negloglik(pd)` returns the value of the negative log likelihood function for the data used to fit the probability distribution `pd`.

## Input Arguments

**pd** — Probability distribution

probability distribution object

Probability distribution, specified as a probability distribution object. Create a probability distribution object with specified parameter values using `makedist`. Alternatively, create a probability distribution object by fitting it to data using `fitdist` or the Distribution Fitter app.

## Output Arguments

**nll** — Negative log likelihood

scalar value

Negative log likelihood value for the data used to fit the distribution, returned as a scalar value.

## Examples

### Negative Log Likelihood for a Fitted Distribution

Load the sample data.

```
load carsmall
```

Create a Weibull distribution object by fitting it to the mile per gallon (MPG) data.

```
pd = fitdist(MPG, 'Weibull')
```

```
pd =
```

```
WeibullDistribution
```

```
Weibull distribution
```

```
A = 26.5079 [24.8333, 28.2954]
```

```
B = 3.27193 [2.79441, 3.83104]
```

Compute the negative log likelihood for the fitted Weibull distribution.

```
wll = negloglik(pd)
```

```
wll =
```

```
327.4942
```

## See Also

`distributionFitter` | `fitdist` | `makedist`

# net

**Class:** grandset

Generate quasi-random point set

## Syntax

```
X = net(p, n)
```

## Description

`X = net(p, n)` returns the first  $n$  points  $X$  from the point set  $p$  of the `grandset` on page 32-5245 class.  $X$  is  $n$ -by- $d$ , where  $d$  is the dimension of the point set.

Objects  $p$  of the `@grandset` class encapsulate properties of a specified quasi-random sequence. Values of the point set are not generated and stored in memory until  $p$  is accessed using `net` or parenthesis indexing.

## Examples

Use `haltonset` to generate a 3-D Halton point set, skip the first 1000 values, and then retain every 101st point:

```
p = haltonset(3, 'Skip', 1e3, 'Leap', 1e2)
p =
    Halton point set in 3 dimensions (8.918019e+013 points)
    Properties:
        Skip : 1000
        Leap : 100
        ScrambleMethod : none
```

Use `scramble` to apply reverse-radix scrambling:

```
p = scramble(p, 'RR2')
p =
```

```
Halton point set in 3 dimensions (8.918019e+013 points)
Properties:
    Skip : 1000
    Leap : 100
    ScrambleMethod : RR2
```

Use `net` to generate the first four points:

```
X0 = net(p,4)
X0 =
    0.0928    0.6950    0.0029
    0.6958    0.2958    0.8269
    0.3013    0.6497    0.4141
    0.9087    0.7883    0.2166
```

Use parenthesis indexing to generate every third point, up to the 11th point:

```
X = p(1:3:11,:)
X =
    0.0928    0.6950    0.0029
    0.9087    0.7883    0.2166
    0.3843    0.9840    0.9878
    0.6831    0.7357    0.7923
```

## See Also

`haltonset` | `grandstream` | `sobolset`

## nLinearCoeffs

**Class:** CompactClassificationDiscriminant

Number of nonzero linear coefficients

### Syntax

```
ncoeffs = nLinearCoeffs(obj)
ncoeffs = nLinearCoeffs(obj,delta)
```

### Description

`ncoeffs = nLinearCoeffs(obj)` returns the number of nonzero linear coefficients in the linear discriminant model `obj`.

`ncoeffs = nLinearCoeffs(obj,delta)` returns the number of nonzero linear coefficients for threshold parameter `delta`.

### Input Arguments

**obj**

Discriminant analysis classifier, produced using `fitcdiscr`.

**delta**

Scalar or vector value of the `Delta` parameter. See “Gamma and Delta” on page 32-4273.

### Output Arguments

**ncoeffs**

Nonnegative integer, the number of nonzero coefficients in the discriminant analysis model `obj`.

If you call `nLinearCoeffs` with a `delta` argument, `ncoeffs` is the number of nonzero linear coefficients for threshold parameter `delta`. If `delta` is a vector, `ncoeffs` is a vector with the same number of elements.

If `obj` is a quadratic discriminant model, `ncoeffs` is the number of predictors in `obj`.

## Examples

### Find the Number of Nonzero Coefficients in a Discriminant Analysis Classifier

Find the number of nonzero coefficients in a discriminant analysis classifier for various `Delta` values.

Create a discriminant analysis classifier from the `fishseriris` data.

```
load fisheriris
obj = fitcdiscr(meas,species);
```

Find the number of nonzero coefficients in `obj`.

```
ncoeffs = nLinearCoeffs(obj)
```

```
ncoeffs =
```

```
4
```

Find the number of nonzero coefficients for `delta = 1, 2, 4, and 8`.

```
delta = [1 2 4 8];
ncoeffs = nLinearCoeffs(obj,delta)
```

```
ncoeffs =
```

```
4
```

```
4
```

```
3
```

```
0
```

The `DeltaPredictor` property gives the values of `delta` where the number of nonzero coefficients changes.

```
ncoeffs2 = nLinearCoeffs(obj,obj.DeltaPredictor)
```

```
ncoeffs2 =
```

```
4
3
1
2
```

## Definitions

### Gamma and Delta

Regularization is the process of finding a small set of predictors that yield an effective predictive model. For linear discriminant analysis, there are two parameters,  $\gamma$  and  $\delta$ , that control regularization as follows. `cvshrink` helps you select appropriate values of the parameters.

Let  $\Sigma$  represent the covariance matrix of the data  $X$ , and let  $\hat{X}$  be the centered data (the data  $X$  minus the mean by class). Define

$$D = \text{diag}(\hat{X}^T * \hat{X}).$$

The regularized covariance matrix  $\tilde{\Sigma}$  is

$$\tilde{\Sigma} = (1 - \gamma)\Sigma + \gamma D.$$

Whenever  $\gamma \geq \text{MinGamma}$ ,  $\tilde{\Sigma}$  is nonsingular.

Let  $\mu_k$  be the mean vector for those elements of  $X$  in class  $k$ , and let  $\mu_0$  be the global mean vector (the mean of the rows of  $X$ ). Let  $C$  be the correlation matrix of the data  $X$ , and let

$\tilde{C}$  be the regularized correlation matrix:

$$\tilde{C} = (1 - \gamma)C + \gamma I,$$

where  $I$  is the identity matrix.

The linear term in the regularized discriminant analysis classifier for a data point  $x$  is

$$(x - \mu_0)^T \tilde{\Sigma}^{-1} (\mu_k - \mu_0) = \left[ (x - \mu_0)^T D^{-1/2} \right] \left[ \tilde{C}^{-1} D^{-1/2} (\mu_k - \mu_0) \right].$$

The parameter  $\delta$  enters into this equation as a threshold on the final term in square

brackets. Each component of the vector  $\left[ \tilde{C}^{-1} D^{-1/2} (\mu_k - \mu_0) \right]$  is set to zero if it is smaller in magnitude than the threshold  $\delta$ . Therefore, for class  $k$ , if component  $j$  is thresholded to zero, component  $j$  of  $x$  does not enter into the evaluation of the posterior probability.

The `DeltaPredictor` property is a vector related to this threshold. When

$\delta \geq \text{DeltaPredictor}(i)$ , all classes  $k$  have

$$\left| \tilde{C}^{-1} D^{-1/2} (\mu_k - \mu_0) \right| \leq \delta.$$

Therefore, when  $\delta \geq \text{DeltaPredictor}(i)$ , the regularized classifier does not use predictor  $i$ .

## See Also

`CompactClassificationDiscriminant` | `cvshrink` | `fitcdiscr`

## Topics

“Discriminant Analysis Classification” on page 20-2



# nlinfit

Nonlinear regression

## Syntax

```
beta = nlinfit(X,Y,modelfun,beta0)
beta = nlinfit(X,Y,modelfun,beta0,options)
beta = nlinfit(____,Name,Value)
[beta,R,J,CovB,MSE,ErrorModelInfo] = nlinfit(____)
```

## Description

`beta = nlinfit(X,Y,modelfun,beta0)` returns a vector of estimated coefficients for the nonlinear regression of the responses in *Y* on the predictors in *X* using the model specified by `modelfun`. The coefficients are estimated using iterative least squares estimation, with initial values specified by `beta0`.

`beta = nlinfit(X,Y,modelfun,beta0,options)` fits the nonlinear regression using the algorithm control parameters in the structure `options`. You can return any of the output arguments in the previous syntaxes.

`beta = nlinfit(____,Name,Value)` uses additional options specified by one or more name-value pair arguments. For example, you can specify observation weights or a nonconstant error model. You can use any of the input arguments in the previous syntaxes.

`[beta,R,J,CovB,MSE,ErrorModelInfo] = nlinfit(____)` additionally returns the residuals, *R*, the Jacobian of `modelfun`, *J*, the estimated variance-covariance matrix for the estimated coefficients, *CovB*, an estimate of the variance of the error term, *MSE*, and a structure containing details about the error model, `ErrorModelInfo`.

## Examples

### Nonlinear Regression Model Using Default Options

Load sample data.

```
S = load('reaction');
X = S.reactants;
y = S.rate;
beta0 = S.beta;
```

Fit the Hougen-Watson model to the rate data using the initial values in beta0.

```
beta = nlinfit(X,y,@hougen,beta0)
```

```
beta =
    1.2526
    0.0628
    0.0400
    0.1124
    1.1914
```

### Nonlinear Regression Using Robust Options

Generate sample data from the nonlinear regression model  $y = b_1 + b_2 \cdot \exp\{-b_3x\} + \epsilon$ , where  $b_1$ ,  $b_2$ , and  $b_3$  are coefficients, and the error term is normally distributed with mean 0 and standard deviation 0.1.

```
modelfun = @(b,x) (b(1)+b(2)*exp(-b(3)*x));

rng('default') % for reproducibility
b = [1;3;2];
x = exprnd(2,100,1);
y = modelfun(b,x) + normrnd(0,0.1,100,1);
```

Set robust fitting options.

```
opts = statset('nlinfit');
opts.RobustWgtFun = 'bisquare';
```

Fit the nonlinear model using the robust fitting options.

```
beta0 = [2;2;2];
beta = nlinfit(x,y,modelfun,beta0,opts)
```

```
beta =

    1.0041
    3.0997
    2.1483
```

### Nonlinear Regression Using Observation Weights

Load sample data.

```
S = load('reaction');
X = S.reactants;
y = S.rate;
beta0 = S.beta;
```

Specify a vector of known observation weights.

```
W = [8 2 1 6 12 9 12 10 10 12 2 10 8]';
```

Fit the Hougen-Watson model to the rate data using the specified observation weights.

```
[beta,R,J,CovB] = nlinfit(X,y,@hougen,beta0,'Weights',W);
beta
```

```
beta =

    2.2068
    0.1077
    0.0766
    0.1818
    0.6516
```

Display the coefficient standard errors.

```
sqrt(diag(CovB))
```

```
ans =
```

```
2.5721
0.1251
0.0950
0.2043
0.7735
```

### Nonlinear Regression Using Weights Function Handle

Load sample data.

```
S = load('reaction');
X = S.reactants;
y = S.rate;
beta0 = S.beta;
```

Specify a function handle for observation weights. The function accepts the model fitted values as input, and returns a vector of weights.

```
a = 1; b = 1;
weights = @(yhat) 1./((a + b*abs(yhat)).^2);
```

Fit the Hougen-Watson model to the rate data using the specified observation weights function.

```
[beta,R,J,CovB] = nlinfit(X,y,@hougen,beta0,'Weights',weights);
beta
```

```
beta =

    0.8308
    0.0409
    0.0251
    0.0801
    1.8261
```

Display the coefficient standard errors.

```
sqrt(diag(CovB))
```

```
ans =  
  
    0.5822  
    0.0297  
    0.0197  
    0.0578  
    1.2810
```

## Nonlinear Regression Using Nonconstant Error Model

Load sample data.

```
S = load('reaction');  
X = S.reactants;  
y = S.rate;  
beta0 = S.beta;
```

Fit the Hougen-Watson model to the rate data using the combined error model.

```
[beta,R,J,CovB,MSE,ErrorModelInfo] = nlinfit(X,y,@hougen,beta0,'ErrorModel','combined')  
beta
```

```
beta =  
  
    1.2526  
    0.0628  
    0.0400  
    0.1124  
    1.1914
```

Display the error model information.

```
ErrorModelInfo
```

```
ErrorModelInfo =  
  
    struct with fields:
```

```
ErrorModel: 'combined'  
ErrorParameters: [0.1517 5.6783e-08]  
ErrorVariance: [function_handle]  
MSE: 1.6245  
ScheffeSimPred: 6  
WeightFunction: 0  
FixedWeights: 0  
RobustWeightFunction: 0
```

## Input Arguments

### **x** — Predictor variables

matrix

Predictor variables for the nonlinear regression function, specified as a matrix. Typically,  $X$  is a design matrix of predictor (independent variable) values, with one row for each value in  $Y$ , and one column for each coefficient. However,  $X$  can be any array that `modelfun` can accept.

Data Types: `single` | `double`

### **y** — Response values

vector

Response values (dependent variable) for fitting the nonlinear regression function, specified as a vector with the same number of rows as  $X$ .

Data Types: `single` | `double`

### **modelfun** — Nonlinear regression model function

function handle

Nonlinear regression model function, specified as a function handle. `modelfun` must accept two input arguments, a coefficient vector and an array  $X$ —in that order—and return a vector of fitted response values.

For example, to specify the `hougen` nonlinear regression function, use the function handle `@hougen`.

Data Types: `function_handle`

**beta0 — Initial coefficient values**

vector

Initial coefficient values for the least squares estimation algorithm, specified as a vector.

---

**Note** Poor starting values can lead to a solution with large residual error.

---

Data Types: `single` | `double`

**options — Estimation algorithm options**structure created using `statset`

Estimation algorithm options, specified as a structure you create using `statset`. The following `statset` parameters are applicable to `nlinfit`.

**DerivStep — Relative difference for finite difference gradient** $\text{eps}^{(1/3)}$  (default) | positive scalar value | vector

Relative difference for the finite difference gradient calculation, specified as a positive scalar value, or a vector the same size as `beta`. Use a vector to specify a different relative difference for each coefficient.

**Display — Level of output display**

'off' (default) | 'iter' | 'final'

Level of output display during estimation, specified as one of 'off', 'iter', or 'final'. If you specify 'iter', output is displayed at each iteration. If you specify 'final', output is displayed after the final iteration.

**FunValCheck — Indicator for whether to check for invalid values**

'on' (default) | 'off'

Indicator for whether to check for invalid values such as NaN or Inf from the objective function, specified as 'on' or 'off'.

**MaxIter — Maximum number of iterations**

100 (default) | positive integer

Maximum number of iterations for the estimation algorithm, specified as a positive integer. Iterations continue until estimates are within the convergence tolerance, or the maximum number of iterations specified by `MaxIter` is reached.

### **RobustWgtFun** — Weight function

character vector | function handle | []

Weight function for robust fitting, specified as a valid character vector or function handle.

---

**Note** `RobustWgtFun` must have value [] when you use observation weights, `w`.

---

The following table describes the possible character vectors. Let  $r$  denote normalized residuals and  $w$  denote robust weights. The indicator function  $I[x]$  is equal to 1 if the expression  $x$  is true, and 0 otherwise.

Weight Function	Equation	Default Tuning Constant
' ' (default)	No robust fitting	—
'andrews'	$w = I[ r  < \pi] \times \sin(r) / r$	1.339
'bisquare'	$w = I[ r  < 1] \times (1 - r^2)^2$	4.685
'cauchy'	$w = \frac{1}{1 + r^2}$	2.385
'fair'	$w = \frac{1}{1 +  r }$	1.400
'huber'	$w = \frac{1}{\max(1,  r )}$	1.345
'logistic'	$w = \frac{\tanh(r)}{r}$	1.205
'talwar'	$w = I[ r  < 1]$	2.795



Weight Function	Equation	Default Tuning Constant
'welsch'	$w = \exp\{-r^2\}$	2.985

You can alternatively specify a function handle that accepts a vector of normalized residuals as input, and returns a vector of robust weights as output. If you use a function handle, you must provide a `Tune` constant.

### **Tune** — Tuning constant

positive scalar value

Tuning constant for robust fitting, specified as a positive scalar value. The tuning constant is used to normalize residuals before applying a robust weight function. The default tuning constant depends on the function specified by `RobustWgtFun`.

If you use a function handle to specify `RobustWgtFun`, then you must specify a value for `Tune`.

### **TolFun** — Termination tolerance on residual sum of squares

1e-8 (default) | positive scalar value

Termination tolerance for the residual sum of squares, specified as a positive scalar value. Iterations continue until estimates are within the convergence tolerance, or the maximum number of iterations specified by `MaxIter` is reached.

### **TolX** — Termination tolerance on estimated coefficients

1e-8 (default) | positive scalar value

Termination tolerance on the estimated coefficients, `beta`, specified as a positive scalar value. Iterations continue until estimates are within the convergence tolerance, or the maximum number of iterations specified by `MaxIter` is reached.

### **Robust** — Indicator for robust fitting

'off' (default) | 'on'

Indicator for robust fitting, specified as 'off' or 'on'.

---

**Note** `Robust` will be removed in a future software release. Use `RobustWgtFun` for robust fitting.

---

**WgtFun** — Weight function for robust fitting

character vector | function handle

Weight function for robust fitting, specified as a character vector indicating a weight function, or a function handle. `WgtFun` is valid only when `Robust` has value 'on'.

---

**Note** `WgtFun` will be removed in a future software release. Use `RobustWgtFun` instead.

---

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'ErrorModel', 'proportional', 'ErrorParameters', 0.5` specifies a proportional error model, with initial value 0.5 for the error parameter estimation

**ErrorModel** — Form of error term

'constant' (default) | 'proportional' | 'combined'

Form of the error term, specified as the comma-separated pair consisting of 'ErrorModel' and 'constant', 'proportional', or 'combined' indicating the error model. Each model defines the error using a standard mean-zero and unit-variance variable  $e$  in combination with independent components: the function value  $f$ , and one or two parameters  $a$  and  $b$ .

'constant' (default)

$$y = f + ae$$

'proportional'

$$y = f + bfe$$

'combined'

$$y = f + (a + b|f|)e$$

The only allowed error model when using `Weights` is 'constant'.

---

**Note** `options.RobustWgtFun` must have value [] when using an error model other than 'constant'.

---

**ErrorParameters** — Initial estimates for error model parameters

1 or [1,1] (default) | scalar value | two-element vector

Initial estimates for the error model parameters in the chosen `ErrorModel`, specified as the comma-separated pair consisting of `'ErrorParameters'` and a scalar value or two-element vector.

Error Model	Parameters	Default Values
'constant'	$a$	1
'proportional'	$b$	1
'combined'	$a, b$	[1,1]

For example, if `'ErrorModel'` has the value `'combined'`, you can specify the starting value 1 for  $a$  and the starting value 2 for  $b$  as follows.

Example: `'ErrorParameters', [1,2]`

You can only use the `'constant'` error model when using `Weights`.

---

**Note** `options.RobustWgtFun` must have value `[]` when using an error model other than `'constant'`.

---

Data Types: `double` | `single`

**weights** — Observation weights

vector | function handle

Observation weights, specified as the comma-separated pair consisting of `'Weights'` and a vector of real positive weights or a function handle. You can use observation weights to down-weight the observations that you want to have less influence on the fitted model.

- If  $W$  is a vector, then it must be the same size as  $Y$ .
- If  $W$  is a function handle, then it must accept a vector of predicted response values as input, and return a vector of real positive weights as output.

---

**Note** `options.RobustWgtFun` must have value `[]` when you use observation weights.

---

Data Types: `double` | `single` | `function_handle`

## Output Arguments

### **beta** — Estimated regression coefficients

vector

Estimated regression coefficients, returned as a vector. The number of elements in `beta` equals the number of elements in `beta0`.

Let  $f(X_i, \mathbf{b})$  denote the nonlinear function specified by `modelfun`, where  $\mathbf{x}_i$  are the predictors for observation  $i$ ,  $i = 1, \dots, N$ , and  $\mathbf{b}$  are the regression coefficients. The vector of coefficients returned in `beta` minimizes the weighted least squares equation,

$$\sum_{i=1}^N w_i [y_i - f(\mathbf{x}_i, \mathbf{b})]^2.$$

For unweighted nonlinear regression, all of the weight terms are equal to 1.

### **R** — Residuals

vector

Residuals for the fitted model, returned as a vector.

- If you specify observation weights using the name-value pair argument `Weights`, then `R` contains weighted residuals on page 32-4288.
- If you specify an error model other than `'constant'` using the name-value pair argument `ErrorModel`, then you can no longer interpret `R` as model fit residuals.

### **J** — Jacobian

matrix

Jacobian of the nonlinear regression model, `modelfun`, returned as an  $N$ -by- $p$  matrix, where  $N$  is the number of observations and  $p$  is the number of estimated coefficients.

- If you specify observation weights using the name-value pair argument `Weights`, then `J` is the weighted model function Jacobian on page 32-4288.
- If you specify an error model other than `'constant'` using the name-value pair argument `ErrorModel`, then you can no longer interpret `J` as the model function Jacobian.

**CovB — Estimated variance-covariance matrix**

matrix

Estimated variance-covariance matrix for the fitted coefficients, `beta`, returned as a  $p$ -by- $p$  matrix, where  $p$  is the number of estimated coefficients. If the model Jacobian,  $J$ , has full column rank, then  $\text{CovB} = \text{inv}(J' * J) * \text{MSE}$ , where `MSE` is the mean squared error.

**MSE — Mean squared error**

scalar value

Mean squared error (MSE) of the fitted model, returned as a scalar value. MSE is an estimate of the variance of the error term. If the model Jacobian,  $J$ , has full column rank, then  $\text{MSE} = (R' * R) / (N - p)$ , where  $N$  is the number of observations, and  $p$  is the number of estimated coefficients.

**ErrorModelInfo — Information about error model fit**

structure

Information about the error model fit, returned as a structure with the following fields:

<code>ErrorModel</code>	Chosen error model
<code>ErrorParameters</code>	Estimated error parameters
<code>ErrorVariance</code>	Function handle that accepts an $N$ -by- $p$ matrix, $X$ , and returns an $N$ -by-1 vector of error variances using the estimated error model
<code>MSE</code>	Mean squared error
<code>ScheffeSimPred</code>	Scheffé parameter for simultaneous prediction intervals when using the estimated error model
<code>WeightFunction</code>	Logical with value <code>true</code> if you used a custom weight function previously in <code>nlinfit</code>
<code>FixedWeights</code>	Logical with value <code>true</code> if you used fixed weights previously in <code>nlinfit</code>
<code>RobustWeightFunction</code>	Logical with value <code>true</code> if you used robust fitting previously in <code>nlinfit</code>

## Definitions

### Weighted Residuals

A weighted residual is a residual multiplied by the square root of the corresponding observation weight.

Given estimated regression coefficients,  $\mathbf{b}$ , the residual for observation  $i$  is

$$r_i = y_i - f(\mathbf{x}_i, \mathbf{b}),$$

where  $y_i$  is the observed response and  $f(\mathbf{x}_i, \mathbf{b})$  is the fitted response at predictors  $\mathbf{x}_i$ .

When you fit a weighted nonlinear regression with weights  $w_i$ ,  $i = 1, \dots, N$ , `nlinfit` returns the weighted residuals,

$$r_i^* = \sqrt{w_i} (y_i - f(\mathbf{x}_i, \mathbf{b})).$$

### Weighted Model Function Jacobian

The weighted model function Jacobian is the nonlinear model Jacobian multiplied by the square root of the observation weight matrix.

Given estimated regression coefficients,  $\mathbf{b}$ , the estimated model Jacobian,  $\mathbf{J}$ , for the nonlinear function  $f(\mathbf{x}_i, \mathbf{b})$  has elements

$$\mathbf{J}_{ij} = \frac{\partial f(\mathbf{x}_i, \mathbf{b})}{\partial b_j},$$

where  $b_j$  is the  $j$ th element of  $\mathbf{b}$ .

When you fit a weighted nonlinear regression with diagonal weights matrix  $\mathbf{W}$ , `nlinfit` returns the weighted Jacobian matrix,

$$\mathbf{J}^* = \mathbf{W}^{1/2} \mathbf{J}.$$

## Tips

- To produce error estimates on predictions, use the optional output arguments `R`, `J`, `CovB`, or `MSE` as inputs to `nlpredci`.
- To produce error estimates on the estimated coefficients, `beta`, use the optional output arguments `R`, `J`, `CovB`, or `MSE` as inputs to `nlparci`.
- If you use the robust fitting option, `RobustWgtFun`, you must use `CovB`—and might need `MSE`—as inputs to `nlpredci` or `nlparci` to ensure that the confidence intervals take the robust fit properly into account.

## Algorithms

- `nlinfit` treats NaN values in `Y` or `modelfun(beta0, X)` as missing data, and ignores the corresponding observations.
- For nonrobust estimation, `nlinfit` uses the Levenberg-Marquardt nonlinear least squares algorithm [1].
- For robust estimation, `nlinfit` uses an iterative reweighted least squares algorithm ([2], [3]). At each iteration, the robust weights are recalculated based on each observation's residual from the previous iteration. These weights downweight outliers, so that their influence on the fit is decreased. Iterations continue until the weights converge.
- When you specify a function handle for observation weights, the weights depend on the fitted model. In this case, `nlinfit` uses an iterative generalized least squares algorithm to fit the nonlinear regression model.

## References

- [1] Seber, G. A. F., and C. J. Wild. *Nonlinear Regression*. Hoboken, NJ: Wiley-Interscience, 2003.
- [2] DuMouchel, W. H., and F. L. O'Brien. "Integrating a Robust Option into a Multiple Regression Computing Environment." *Computer Science and Statistics: Proceedings of the 21st Symposium on the Interface*. Alexandria, VA: American Statistical Association, 1989.

- [3] Holland, P. W., and R. E. Welsch. “Robust Regression Using Iteratively Reweighted Least-Squares.” *Communications in Statistics: Theory and Methods*, A6, 1977, pp. 813–827.

## See Also

`fitnlm` | `nlintool` | `nlparci` | `nlpredci`

## Topics

“Nonlinear Regression” on page 13-2

**Introduced before R2006a**



# nlintool

Interactive nonlinear regression

## Syntax

```
nlintool(X,y,fun,beta0)
nlintool(X,y,fun,beta0,alpha)
nlintool(X,y,fun,beta0,alpha,'xname','yname')
```

## Description

`nlintool(X,y,fun,beta0)` is a graphical user interface to the `nlinfit` function, and uses the same input arguments. The interface displays plots of the fitted response against each predictor, with the other predictors held fixed. The fixed values are in the text boxes below each predictor axis. Change the fixed values by typing in a new value or by dragging the vertical lines in the plots to new positions. When you change the value of a predictor, all plots update to display the model at the new point in predictor space. Dashed red curves show 95% simultaneous confidence bands for the function.

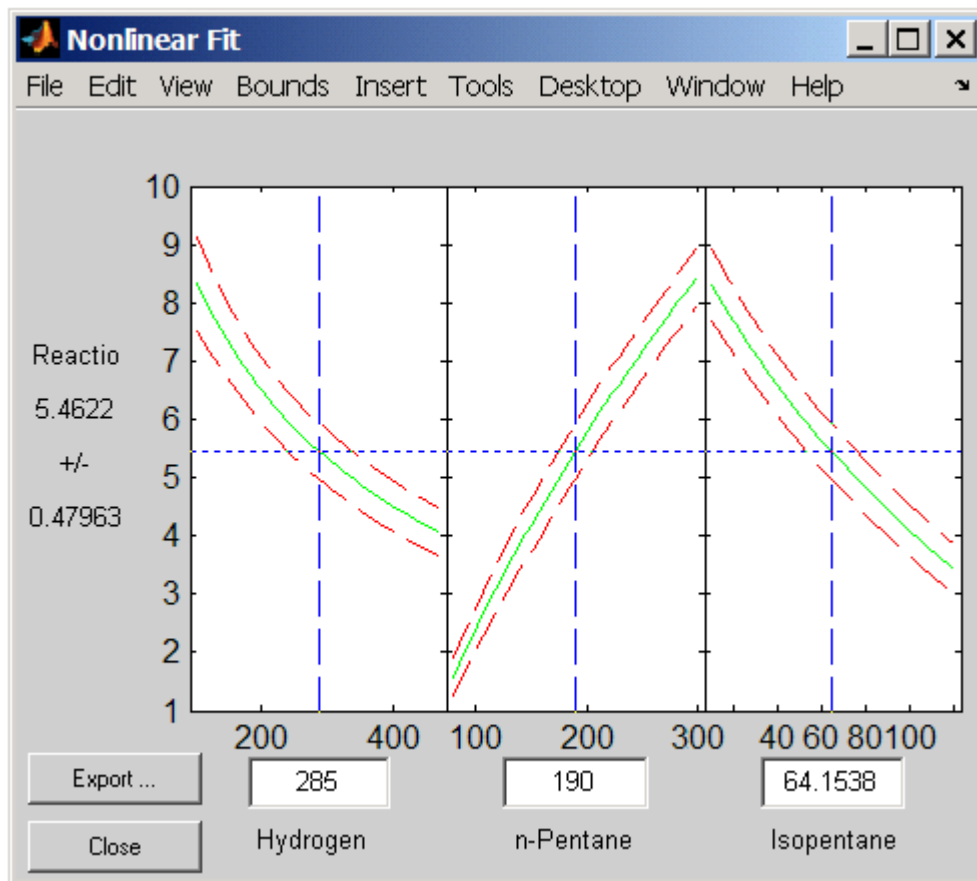
`nlintool(X,y,fun,beta0,alpha)` shows 100(1-alpha)% confidence bands. These are simultaneous confidence bounds for the function value. Using the **Bounds** menu you can switch between simultaneous and non-simultaneous bounds, and between bounds on the function and bounds for predicting a new observation.

`nlintool(X,y,fun,beta0,alpha,'xname','yname')` labels the plots using the character matrix 'xname' for the predictors and the character vector 'yname' for the response.

## Examples

The data in `reaction.mat` are partial pressures of three chemical reactants and the corresponding reaction rates. The function `hougen` implements the nonlinear Hougen-Watson model for reaction rates. The following fits the model to the data:

```
load reaction
nlintool(reactants,rate,@hougen,beta,0.01,xn,yn)
```



## See Also

nlinfit | polytool | rstool

Introduced before R2006a

# nlmefit

Nonlinear mixed-effects estimation

## Syntax

```
beta = nlmefit(X,y,group,V,fun,beta0)
[beta,PSI] = nlmefit(X,y,group,V,fun,beta0)
[beta,PSI,stats] = nlmefit(X,y,group,V,fun,beta0)
[beta,PSI,stats,B] = nlmefit(X,y,group,V,fun,beta0)
[beta,PSI,stats,B] = nlmefit(X,y,group,V,fun,beta0,'Name',value)
```

## Description

`beta = nlmefit(X,y,group,V,fun,beta0)` fits a nonlinear mixed-effects regression model and returns estimates of the fixed effects in `beta`. By default, `nlmefit` fits a model in which each parameter is the sum of a fixed and a random effect, and the random effects are uncorrelated (their covariance matrix is diagonal).

`X` is an  $n$ -by- $h$  matrix of  $n$  observations on  $h$  predictors.

`y` is an  $n$ -by-1 vector of responses.

`group` is a grouping variable indicating  $m$  groups in the observations. `group` is a categorical variable, a numeric vector, a character matrix with rows for group names, or a cell array of character vectors. For more information on grouping variables, see “Grouping Variables” on page 2-59.

`V` is an  $m$ -by- $g$  matrix or cell array of  $g$  group-specific predictors. These are predictors that take the same value for all observations in a group. The rows of `V` are assigned to groups using `grp2idx`, according to the order specified by `grp2idx(group)`. Use a cell array for `V` if group predictors vary in size across groups. Use `[]` for `V` if there are no group-specific predictors.

`fun` is a handle to a function that accepts predictor values and model parameters and returns fitted values. `fun` has the form

```
yfit = modelfun(PHI, XFUN, VFUN)
```

The arguments are:

- PHI — A 1-by- $p$  vector of model parameters.
- XFUN — A  $k$ -by- $h$  array of predictors, where:
  - $k = 1$  if XFUN is a single row of X.
  - $k = n_i$  if XFUN contains the rows of X for a single group of size  $n_i$ .
  - $k = n$  if XFUN contains all rows of X.
- VFUN — Group-specific predictors given by one of:
  - A 1-by- $g$  vector corresponding to a single group and a single row of V.
  - An  $n$ -by- $g$  array, where the  $j$ th row is  $V(I,:)$  if the  $j$ th observation is in group I.

If V is empty, `nlmefit` calls `modelfun` with only two inputs.

- yfit — A  $k$ -by-1 vector of fitted values

When either PHI or VFUN contains a single row, it corresponds to all rows in the other two input arguments.

---

**Note** If `modelfun` can compute `yfit` for more than one vector of model parameters per call, use the 'Vectorization' parameter (described later) for improved performance.

---

`beta0` is a  $q$ -by-1 vector with initial estimates for  $q$  fixed effects. By default,  $q$  is the number of model parameters  $p$ .

`nlmefit` fits the model by maximizing an approximation to the marginal likelihood with random effects integrated out, assuming that:

- Random effects are multivariate normally distributed and independent between groups.
- Observation errors are independent, identically normally distributed, and independent of the random effects.

`[beta, PSI] = nlmefit(X, y, group, V, fun, beta0)` also returns `PSI`, an  $r$ -by- $r$  estimated covariance matrix for the random effects. By default,  $r$  is equal to the number of model parameters  $p$ .

`[beta, PSI, stats] = nlmefit(X, y, group, V, fun, beta0)` also returns `stats`, a structure with fields:

- `dfe` — The error degrees of freedom for the model
- `logl` — The maximized loglikelihood for the fitted model
- `rmse` — The square root of the estimated error variance (computed on the log scale for the exponential error model)
- `errorparam` — The estimated parameters of the error variance model
- `aic` — The Akaike information criterion, calculated as  $aic = -2 * logl + 2 * numParam$ , where `numParam` is the number of fitting parameters, including the degree of freedom for covariance matrix of the random effects, the number of fixed effects and the number of parameters of the error model, and `logl` is a field in the `stats` structure
- `bic` — The Bayesian information criterion, calculated as  $bic = -2 * logl + log(M) * numParam$ 
  - `M` is the number of groups.
  - `numParam` and `logl` are defined as in `aic`.

Note that some literature suggests that the computation of `bic` should be  $bic = -2 * logl + log(N) * numParam$ , where `N` is the number of observations.

- `covb` — The estimated covariance matrix of the parameter estimates
- `sebeta` — The standard errors for `beta`
- `ires` — The population residuals ( $y - y_{population}$ ), where `y_population` is the individual predicted values
- `pres` — The population residuals ( $y - y_{population}$ ), where `y_population` is the population predicted values
- `iwres` — The individual weighted residuals
- `pwres` — The population weighted residuals
- `cwres` — The conditional weighted residuals

`[beta, PSI, stats, B] = nlmefit(X, y, group, V, fun, beta0)` also returns `B`, an  $r$ -by- $m$  matrix of estimated random effects for the  $m$  groups. By default,  $r$  is equal to the number of model parameters  $p$ .

`[beta, PSI, stats, B] = nlmeFit(X, y, group, V, fun, beta0, 'Name', value)` specifies one or more optional parameter name/value pairs. Specify *Name* inside single quotes.

Use the following parameters to fit a model different from the default. (The default model is obtained by setting both `FEConstDesign` and `REConstDesign` to `eye(p)`, or by setting both `FEParamsSelect` and `REParamsSelect` to `1:p`.) Use at most one parameter with an 'FE' prefix and one parameter with an 'RE' prefix. The `nlmeFit` function requires you to specify at least one fixed effect and one random effect.

Parameter	Value
<code>FEParamsSelect</code>	A vector specifying which elements of the parameter vector <code>PHI</code> include a fixed effect, given as a numeric vector of indices from 1 to $p$ or as a 1-by- $p$ logical vector. If $q$ is the specified number of elements, then the model includes $q$ fixed effects.
<code>FEConstDesign</code>	A $p$ -by- $q$ design matrix <code>ADESIGN</code> , where <code>ADESIGN*beta</code> are the fixed components of the $p$ elements of <code>PHI</code> .
<code>FEGroupDesign</code>	A $p$ -by- $q$ -by- $m$ array specifying a different $p$ -by- $q$ fixed-effects design matrix for each of the $m$ groups.
<code>FEObsDesign</code>	A $p$ -by- $q$ -by- $n$ array specifying a different $p$ -by- $q$ fixed-effects design matrix for each of the $n$ observations.
<code>REParamsSelect</code>	A vector specifying which elements of the parameter vector <code>PHI</code> include a random effect, given as a numeric vector of indices from 1 to $p$ or as a 1-by- $p$ logical vector. The model includes $r$ random effects, where $r$ is the specified number of elements.
<code>REConstDesign</code>	A $p$ -by- $r$ design matrix <code>BDESIGN</code> , where <code>BDESIGN*B</code> are the random components of the $p$ elements of <code>PHI</code> .
<code>REGroupDesign</code>	A $p$ -by- $r$ -by- $m$ array specifying a different $p$ -by- $r$ random-effects design matrix for each of $m$ groups.
<code>REObsDesign</code>	A $p$ -by- $r$ -by- $n$ array specifying a different $p$ -by- $r$ random-effects design matrix for each of $n$ observations.

Use the following parameters to control the iterative algorithm for maximizing the likelihood:

Parameter	Value
RefineBeta0	Determines whether nlmefit makes an initial refinement of beta0 by first fitting modelfun without random effects and replacing beta0 with beta. Choices are 'on' and 'off'. The default value is 'on'.
ErrorModel	<p>A character vector specifying the form of the error term. Default is 'constant'. Each model defines the error using a standard normal (Gaussian) variable <math>e</math>, the function value <math>f</math>, and one or two parameters <math>a</math> and <math>b</math>. Choices are:</p> <ul style="list-style-type: none"> <li>• 'constant': <math>y = f + a * e</math></li> <li>• 'proportional': <math>y = f + b * f * e</math></li> <li>• 'combined': <math>y = f + (a + b * f) * e</math></li> <li>• 'exponential': <math>y = f * \exp(a * e)</math>, or equivalently <math>\log(y) = \log(f) + a * e</math></li> </ul> <p>If this parameter is given, the output stats.errorparam field has the value</p> <ul style="list-style-type: none"> <li>• <math>a</math> for 'constant' and 'exponential'</li> <li>• <math>b</math> for 'proportional'</li> <li>• <math>[a \ b]</math> for 'combined'</li> </ul>

Parameter	Value
ApproximationType	<p>The method used to approximate the likelihood of the model. Choices are:</p> <ul style="list-style-type: none"><li data-bbox="664 392 1348 482">• 'LME' — Use the likelihood for the linear mixed-effects model at the current conditional estimates of <code>beta</code> and <code>B</code>. This is the default.</li><li data-bbox="664 505 1348 595">• 'RELME' — Use the restricted likelihood for the linear mixed-effects model at the current conditional estimates of <code>beta</code> and <code>B</code>.</li><li data-bbox="664 618 1348 673">• 'FO' — First-order Laplacian approximation without random effects.</li><li data-bbox="664 696 1348 751">• 'FOCE' — First-order Laplacian approximation at the conditional estimates of <code>B</code>.</li></ul>



Parameter	Value
Vectorization	<p>Indicates acceptable sizes for the PHI, XFUN, and VFUN input arguments to <code>modelfun</code>. Choices are:</p> <ul style="list-style-type: none"> <li>• 'SinglePhi' — <code>modelfun</code> can only accept a single set of model parameters at a time, so PHI must be a single row vector in each call. <code>nlmefit</code> calls <code>modelfun</code> in a loop, if necessary, with a single PHI vector and with XFUN containing rows for a single observation or group at a time. VFUN may be a single row that applies to all rows of XFUN, or a matrix with rows corresponding to rows in XFUN. This is the default.</li> <li>• 'SingleGroup' — <code>modelfun</code> can only accept inputs corresponding to a single group in the data, so XFUN must contain rows of X from a single group in each call. Depending on the model, PHI is a single row that applies to the entire group or a matrix with one row for each observation. VFUN is a single row.</li> <li>• 'Full' — <code>modelfun</code> can accept inputs for multiple parameter vectors and multiple groups in the data. Either PHI or VFUN may be a single row that applies to all rows of XFUN or a matrix with rows corresponding to rows in XFUN. This option can improve performance by reducing the number of calls to <code>modelfun</code>, but may require <code>modelfun</code> to perform singleton expansion on PHI or V.</li> </ul>
CovParameterization	<p>Specifies the parameterization used internally for the scaled covariance matrix. Choices are 'chol' for the Cholesky factorization or 'logm' the matrix logarithm. The default is 'logm'.</p>

Parameter	Value
CovPattern	<p>Specifies an <math>r</math>-by-<math>r</math> logical or numeric matrix <math>P</math> that defines the pattern of the random-effects covariance matrix <math>\Psi</math>. <code>nlmefit</code> estimates the variances along the diagonal of <math>\Psi</math> and the covariances specified by nonzeros in the off-diagonal elements of <math>P</math>. Covariances corresponding to zero off-diagonal elements in <math>P</math> are constrained to be zero. If <math>P</math> does not specify a row-column permutation of a block diagonal matrix, <code>nlmefit</code> adds nonzero elements to <math>P</math> as needed. The default value of <math>P</math> is <code>eye(r)</code>, corresponding to uncorrelated random effects.</p> <p>Alternatively, <math>P</math> may be a 1-by-<math>r</math> vector containing values in <math>1:r</math>, with equal values specifying groups of random effects. In this case, <code>nlmefit</code> estimates covariances only within groups, and constrains covariances across groups to be zero.</p>
ParamTransform	<p>A vector of <math>p</math>-values specifying a transformation function <math>f()</math> for each of the <math>P</math> parameters: <math>XB = ADESIGN*BETA + BDESIGN*PHI = f(XB)</math>. Each element of the vector must be one of the following integer codes specifying the transformation for the corresponding value of <math>PHI</math>:</p> <ul style="list-style-type: none"> <li>• 0: <math>PHI = XB</math> (default for all parameters)</li> <li>• 1: <math>\log(PHI) = XB</math></li> <li>• 2: <math>\text{probit}(PHI) = XB</math></li> <li>• 3: <math>\text{logit}(PHI) = XB</math></li> </ul>

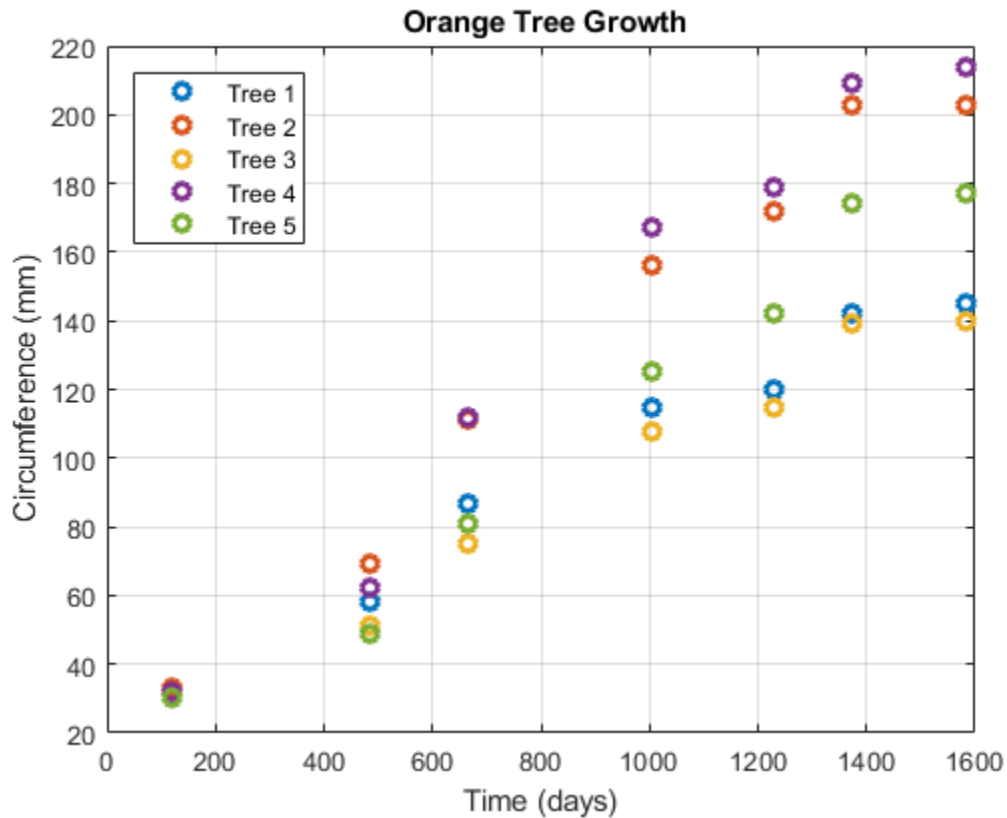
Parameter	Value
Options	<p>A structure of the form returned by <code>statset.nlmefit</code> uses the following <code>statset</code> parameters:</p> <ul style="list-style-type: none"> <li>• 'DerivStep' — Relative difference used in finite difference gradient calculation. May be a scalar, or a vector whose length is the number of model parameters <math>p</math>. The default is <math>\text{eps}^{(1/3)}</math>.</li> <li>• 'Display' — Level of iterative display during estimation. Choices are: <ul style="list-style-type: none"> <li>• 'off' (default) — Displays no information</li> <li>• 'final' — Displays information after the final iteration</li> <li>• 'iter' — Displays information at each iteration</li> </ul> </li> <li>• 'FunValCheck' — Check for invalid values, such as NaN or Inf, from <code>modelfun</code>. Choices are 'on' and 'off'. The default is 'on'.</li> <li>• 'MaxIter' — Maximum number of iterations allowed. The default is 200.</li> <li>• 'OutputFcn' — Function handle specified using @, a cell array with function handles or an empty array (default). The solver calls all output functions after each iteration.</li> <li>• 'TolFun' — Termination tolerance on the loglikelihood function. The default is <math>1e-4</math>.</li> <li>• 'TolX' — Termination tolerance on the estimated fixed and random effects. The default is <math>1e-4</math>.</li> </ul>
OptimFun	<p>Specifies the optimization function used in maximizing the likelihood. Choices are 'fminsearch' to use <code>fminsearch</code> or 'fminunc' to use <code>fminunc</code>. The default is 'fminsearch'. You can specify 'fminunc' only if Optimization Toolbox software is installed.</p>

## Examples

### Nonlinear Mixed-Effects Model

Enter and display data on the growth of five orange trees.

```
CIRC = [30 58 87 115 120 142 145;  
        33 69 111 156 172 203 203;  
        30 51 75 108 115 139 140;  
        32 62 112 167 179 209 214;  
        30 49 81 125 142 174 177];  
time = [118 484 664 1004 1231 1372 1582];  
  
h = plot(time,CIRC','o','LineWidth',2);  
xlabel('Time (days)')  
ylabel('Circumference (mm)')  
title('\bf Orange Tree Growth')  
legend([repmat('Tree ',5,1),num2str((1:5)')],...  
       'Location','NW')  
grid on  
hold on
```



Use an anonymous function to specify a logistic growth model.

```
model = @(PHI,t) (PHI(:,1))./(1+exp(-(t-PHI(:,2))./PHI(:,3))));
```

Fit the model using `nlmefit` with default settings (that is, assuming each parameter is the sum of a fixed and a random effect, with no correlation among the random effects):

```
TIME = repmat(time,5,1);
NUMS = repmat((1:5)',size(time));

beta0 = [100 100 100];
[beta1,PSI1,stats1] = nlmefit(TIME(:),CIRC(:),NUMS(:),...
    [],model,beta0)
```

```
beta1 =  
  
    191.3189  
    723.7608  
    346.2517  
  
PSI1 =  
  
    962.1534         0         0  
         0     0.0000         0  
         0         0     297.9880  
  
stats1 =  
  
    struct with fields:  
  
        dfe: 28  
        logl: -131.5457  
        mse: 59.7882  
        rmse: 7.9016  
        errorparam: 7.7323  
        aic: 277.0913  
        bic: 274.3574  
        covb: [3x3 double]  
        sebeta: [15.2249 33.1579 26.8235]  
        ires: [35x1 double]  
        pres: [35x1 double]  
        iwres: [35x1 double]  
        pwres: [35x1 double]  
        cwres: [35x1 double]
```

The negligible variance of the second random effect,  $\text{PSI1}(2,2)$ , suggests that it can be removed to simplify the model.

```
[beta2, PSI2, stats2, b2] = nlmeFit(TIME(:), CIRC(:), ...  
    NUMS(:), [], model, beta0, 'REParamsSelect', [1 3])  
  
beta2 =  
  
    191.3193
```

```

723.7627
346.2529

PSI2 =

  962.4853      0
      0 297.9475

stats2 =

struct with fields:

  dfe: 29
  logl: -131.5456
  mse: 59.7850
  rmse: 7.7642
  errorparam: 7.7321
  aic: 275.0913
  bic: 272.7479
  covb: [3x3 double]
  sebeta: [15.2270 33.1574 26.8228]
  ires: [35x1 double]
  pres: [35x1 double]
  iwres: [35x1 double]
  pwres: [35x1 double]
  cwres: [35x1 double]

b2 =

-28.5263  31.6066 -36.5079  39.0750  -5.6473
  9.9967  -0.7622   6.0038  -9.4567  -5.7816

```

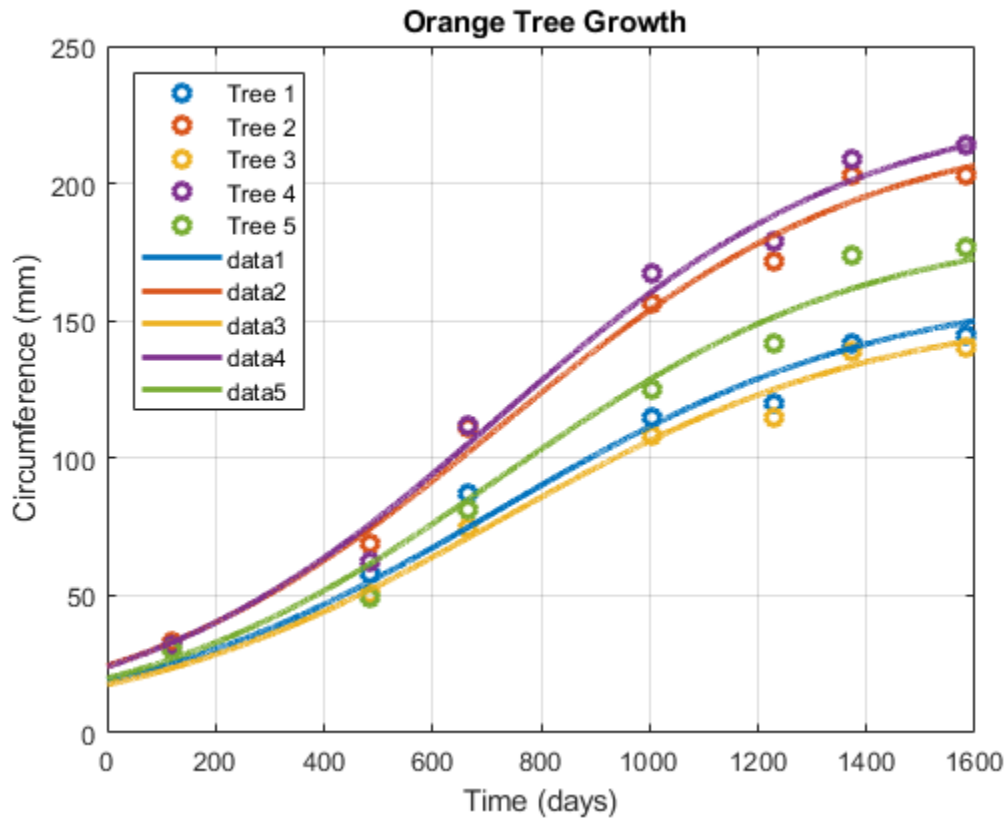
The loglikelihood `logl` is unaffected, and both the Akaike and Bayesian information criteria (`aic` and `bic`) are reduced, supporting the decision to drop the second random effect from the model.

Use the estimated fixed effects in `beta2` and the estimated random effects for each tree in `b2` to plot the model through the data.

```
PHI = repmat(beta2,1,5) + ...           % Fixed effects
      [b2(1,:);zeros(1,5);b2(2,:)];    % Random effects

tplot = 0:0.1:1600;
for I = 1:5
    fitted_model=@(t) (PHI(1,I))./(1+exp(-(t-PHI(2,I))./ ...
        PHI(3,I)));
    plot(tplot,fitted_model(tplot),'Color',h(I).Color, ...
        'LineWidth',2)
end
```





## References

- [1] Lindstrom, M. J., and D. M. Bates. "Nonlinear mixed-effects models for repeated measures data." *Biometrics*. Vol. 46, 1990, pp. 673–687.
- [2] Davidian, M., and D. M. Giltinan. *Nonlinear Models for Repeated Measurements Data*. New York: Chapman & Hall, 1995.
- [3] Pinheiro, J. C., and D. M. Bates. "Approximations to the log-likelihood function in the nonlinear mixed-effects model." *Journal of Computational and Graphical Statistics*. Vol. 4, 1995, pp. 12–35.

[4] Demidenko, E. *Mixed Models: Theory and Applications*. Hoboken, NJ: John Wiley & Sons, Inc., 2004.

## See Also

`nlinfit` | `nlmefitsa` | `nlpredci`

## Topics

“Mixed-Effects Models” on page 13-22

“Grouping Variables” on page 2-59

**Introduced in R2008b**

## nlmefitsa

Fit nonlinear mixed-effects model with stochastic EM algorithm

### Syntax

```
[BETA, PSI, STATS, B] = nlmefitsa(X, Y, GROUP, V, MODELFUN, BETA0)
[BETA, PSI, STATS, B] =
nlmefitsa(X, Y, GROUP, V, MODELFUN, BETA0, 'Name', Value)
```

### Description

[BETA, PSI, STATS, B] = nlmefitsa(X, Y, GROUP, V, MODELFUN, BETA0) fits a nonlinear mixed-effects regression model and returns estimates of the fixed effects in BETA. By default, nlmefitsa fits a model where each model parameter is the sum of a corresponding fixed and random effect, and the covariance matrix of the random effects is diagonal, i.e., uncorrelated random effects.

The BETA, PSI, and other values this function returns are the result of a random (Monte Carlo) simulation designed to converge to the maximum likelihood estimates of the parameters. Because the results are random, it is advisable to examine the plot of simulation to results to be sure that the simulation has converged. It may also be helpful to run the function multiple times, using multiple starting values, or use the 'Replicates' parameter to perform multiple simulations.

```
[BETA, PSI, STATS, B] =
nlmefitsa(X, Y, GROUP, V, MODELFUN, BETA0, 'Name', Value) accepts one or more
comma-separated parameter name/value pairs. Specify Name inside single quotes.
```

### Input Arguments

#### Definitions:

In the following list of arguments, the following variable definitions apply:

- $n$  — number of observations
- $h$  — number of predictor variables
- $m$  — number of groups
- $g$  — number of group-specific predictor variables
- $p$  — number of parameters
- $f$  — number of fixed effects

**X**

An  $n$ -by- $h$  matrix of  $n$  observations on  $h$  predictor variables.

**Y**

An  $n$ -by-1 vector of responses.

**GROUP**

A grouping variable indicating to which of  $m$  groups each observation belongs. GROUP can be a categorical variable, a numeric vector, a character matrix with rows for group names, or a cell array of character vectors.

**V**

An  $m$ -by- $g$  matrix of  $g$  group-specific predictor variables for each of the  $m$  groups in the data. These are predictor values that take on the same value for all observations in a group. Rows of  $V$  are ordered according to `GRP2IDX(GROUP)`. Use an  $m$ -by- $g$  cell array for  $V$  if any of the group-specific predictor values vary in size across groups. Specify `[]` for  $V$  if there are no group predictors.

**MODELFUN**

A handle to a function that accepts predictor values and model parameters, and returns fitted values. MODELFUN has the form `YFIT = MODELFUN(PHI, XFUN, VFUN)` with input arguments

- PHI — A 1-by- $p$  vector of model parameters.
- XFUN — An  $l$ -by- $h$  array of predictor variables where
  - $l$  is 1 if XFUN is a single row of X

- $l$  is  $n_i$  if XFUN contains the rows of X for a single group of size  $n_i$
- $l$  is  $n$  if XFUN contains all rows of X.
- VFUN — Either
  - A 1-by- $g$  vector of group-specific predictors for a single group, corresponding to a single row of V
  - An  $n$ -by- $g$  matrix, where the  $k$ -th row of VFUN is  $V(i,:)$  if the  $k$ -th observation is in group  $i$ .

If V is empty, nlmefitsa calls MODELFUN with only two inputs.

MODELFUN returns an  $l$ -by-1 vector of fitted values YFIT. When either PHI or VFUN contains a single row, that one row corresponds to all rows in the other two input arguments. For improved performance, use the 'Vectorization' parameter name/value pair (described below) if MODELFUN can compute YFIT for more than one vector of model parameters in one call.

### **BETA0**

An  $f$ -by-1 vector with initial estimates for the  $f$  fixed effects. By default,  $f$  is equal to the number of model parameters  $p$ . BETA0 can also be an  $f$ -by-REPS matrix, and the estimation is repeated REPS times using each column of BETA0 as a set of starting values.

## **Name-Value Pair Arguments**

By default, nlmefitsa fits a model where each model parameter is the sum of a corresponding fixed and random effect. Use the following parameter name/value pairs to fit a model with a different number of or dependence on fixed or random effects. Use at most one parameter name with an 'FE' prefix and one parameter name with an 'RE' prefix. Note that some choices change the way nlmefitsa calls MODELFUN, as described further below.

### **FEParamsSelect**

A vector specifying which elements of the model parameter vector PHI include a fixed effect, as a numeric vector with elements in  $1:p$ , or as a 1-by- $p$  logical vector. The model will include  $f$  fixed effects, where  $f$  is the specified number of elements.

**FEConstDesign**

A  $p$ -by- $f$  design matrix `ADESIGN`, where `ADESIGN*BETA` are the fixed components of the  $p$  elements of `PHI`.

**FEGroupDesign**

A  $p$ -by- $f$ -by- $m$  array specifying a different  $p$ -by- $f$  fixed effects design matrix for each of the  $m$  groups.

**REParamsSelect**

A vector specifying which elements of the model parameter vector `PHI` include a random effect, as a numeric vector with elements in  $1:p$ , or as a 1-by- $p$  logical vector. The model will include  $r$  random effects, where  $r$  is the specified number of elements.

**REConstDesign**

A  $p$ -by- $r$  design matrix `BDESIGN`, where `BDESIGN*B` are the random components of the  $p$  elements of `PHI`. This matrix must consist of 0s and 1s, with at most one 1 per row.

The default model is equivalent to setting both `FEConstDesign` and `REConstDesign` to `eye(p)`, or to setting both `FEParamsSelect` and `REParamsSelect` to  $1:p$ .

Additional optional parameter name/value pairs control the iterative algorithm used to maximize the likelihood:

**CovPattern**

Specifies an  $r$ -by- $r$  logical or numeric matrix `PAT` that defines the pattern of the random effects covariance matrix `PSI`. `nlmefitsa` computes estimates for the variances along the diagonal of `PSI` as well as covariances that correspond to non-zeroes in the off-diagonal of `PAT`. `nlmefitsa` constrains the remaining covariances, i.e., those corresponding to off-diagonal zeroes in `PAT`, to be zero. `PAT` must be a row-column permutation of a block diagonal matrix, and `nlmefitsa` adds non-zero elements to `PAT` as needed to produce such a pattern. The default value of `PAT` is `eye(r)`, corresponding to uncorrelated random effects.

Alternatively, specify `PAT` as a 1-by- $r$  vector containing values in  $1:r$ . In this case, elements of `PAT` with equal values define groups of random effects, `nlmefitsa` estimates covariances only within groups, and constrains covariances across groups to be zero.

**Cov0**

Initial value for the covariance matrix `PSI`. Must be an  $r$ -by- $r$  positive definite matrix. If empty, the default value depends on the values of `BETA0`.

**ComputeStdErrors**

`true` to compute standard errors for the coefficient estimates and store them in the output `STATS` structure, or `false` (default) to omit this computation.

**ErrorModel**

A character vector specifying the form of the error term. Default is `'constant'`. Each model defines the error using a standard normal (Gaussian) variable  $e$ , the function value  $f$ , and one or two parameters  $a$  and  $b$ . Choices are

- `'constant'` —  $y = f + a*e$
- `'proportional'` —  $y = f + b*f*e$
- `'combined'` —  $y = f + (a+b*f)*e$
- `'exponential'` —  $y = f*\exp(a*e)$ , or equivalently  $\log(y) = \log(f) + a*e$

If this parameter is given, the output `STATS.errorparam` field has the value

- $a$  for `'constant'` and `'exponential'`
- $b$  for `'proportional'`
- $[a\ b]$  for `'combined'`

**ErrorParameters**

A scalar or two-element vector specifying starting values for parameters of the error model. This specifies the  $a$ ,  $b$ , or  $[a\ b]$  values depending on the `ErrorModel` parameter.

**LogLikMethod**

Specifies the method for approximating the loglikelihood. Choices are:

- `'is'` — Importance sampling
- `'gq'` — Gaussian quadrature
- `'lin'` — Linearization

- 'none' — Omit the loglikelihood approximation (default)

**NBurnIn**

Number of initial burn-in iterations during which the parameter estimates are not recomputed. Default is 5.

**NChains**

Number  $c$  of "chains" simulated. Default is 1. Setting  $c > 1$  causes  $c$  simulated coefficient vectors to be computed for each group during each iteration. Default depends on the data, and is chosen to provide about 100 groups across all chains.

**NIterations**

Number of iterations. This can be a scalar or a three-element vector. Controls how many iterations are performed for each of three phases of the algorithm:

- 1 simulated annealing
- 2 full step size
- 3 reduced step size

Default is [150 150 100]. A scalar is distributed across the three phases in the same proportions as the default.

**NMCMCIterations**

Number of Markov Chain Monte Carlo (MCMC) iterations. This can be a scalar or a three-element vector. Controls how many of three different types of MCMC updates are performed during each phase of the main iteration:

- 1 full multivariate update
- 2 single coordinate update
- 3 multiple coordinate update

Default is [2 2 2]. A scalar value is treated as a three-element vector with all elements equal to the scalar.



**OptimFun**

Either 'fminsearch' or 'fminunc', specifying the optimization function to be used during the estimation process. Default is 'fminsearch'. Use of 'fminunc' requires Optimization Toolbox.

**Options**

A structure created by a call to `statset.nlmefitsa` uses the following `statset` parameters:

- 'DerivStep' — Relative difference used in finite difference gradient calculation. May be a scalar, or a vector whose length is the number of model parameters  $p$ . The default is  $\text{eps}^{(1/3)}$ .
- Display — Level of display during estimation.
  - 'off' (default) — Displays no information
  - 'final' — Displays information after the final iteration of the estimation algorithm
  - 'iter' — Displays information at each iteration
- FunValCheck
  - 'on' (sdefault) — Check for invalid values (such as NaN or Inf) from MODELFUN
  - 'off' — Skip this check
- OutputFcn — Function handle specified using @, a cell array with function handles or an empty array. `nlmefitsa` calls all output functions after each iteration. See `nlmefitoutputfcn.m` (the default output function for `nlmefitsa`) for an example of an output function.

**ParamTransform**

A vector of  $p$ -values specifying a transformation function  $f()$  for each of the  $p$  parameters:

```
XB = ADESIGN*BETA + BDESIGN*B
PHI = f(XB)
```

Each element of the vector must be one of the following integer codes specifying the transformation for the corresponding value of PHI:

- 0:  $\text{PHI} = \text{XB}$  (default for all parameters)
- 1:  $\log(\text{PHI}) = \text{XB}$
- 2:  $\text{probit}(\text{PHI}) = \text{XB}$
- 3:  $\text{logit}(\text{PHI}) = \text{XB}$

**Replicates**

Number REPS of estimations to perform starting from the starting values in the vector BETA0. If BETA0 is a matrix, REPS must match the number of columns in BETA0. Default is the number of columns in BETA0.

**Vectorization**

Determines the possible sizes of the PHI, XFUN, and VFUN input arguments to MODELFUN. Possible values are:

- 'SinglePhi' — MODELFUN is a function (such as an ODE solver) that can only compute YFIT for a single set of model parameters at a time, i.e., PHI must be a single row vector in each call. nlmefitsa calls MODELFUN in a loop if necessary using a single PHI vector and with XFUN containing rows for a single observation or group at a time. VFUN may be a single row that applies to all rows of XFUN, or a matrix with rows corresponding to rows in XFUN.
- 'SingleGroup' — MODELFUN can only accept inputs corresponding to a single group in the data, i.e., XFUN must contain rows of X from a single group in each call. Depending on the model, PHI is a single row that applies to the entire group, or a matrix with one row for each observation. VFUN is a single row.
- 'Full' — MODELFUN can accept inputs for multiple parameter vectors and multiple groups in the data. Either PHI or VFUN may be a single row that applies to all rows of XFUN, or a matrix with rows corresponding to rows in XFUN. Using this option can improve performance by reducing the number of calls to MODELFUN, but may require MODELFUN to perform singleton expansion on PHI or V.

The default for 'Vectorization' is 'SinglePhi'. In all cases, if V is empty, nlmefitsa calls MODELFUN with only two inputs.

## Output Arguments

### BETA

Estimates of the fixed effects

### PSI

An  $r$ -by- $r$  estimated covariance matrix for the random effects. By default,  $r$  is equal to the number of model parameters  $p$ .

### STATS

A structure with the following fields:

- `logl` — The maximized loglikelihood for the fitted model; empty if the `LogLikMethod` parameter has its default value of 'none'
- `rmse` — The square root of the estimated error variance (computed on the log scale for the exponential error model)
- `errorparam` — The estimated parameters of the error variance model
- `aic` — The Akaike information criterion (empty if `logl` is empty), calculated as  $aic = -2 * logl + 2 * numParam$ , where
  - `logl` is the maximized loglikelihood.
  - `numParam` is the number of fitting parameters, including the degree of freedom for covariance matrix of the random effects, the number of fixed effects and the number of parameters of the error model.
- `bic` — The Bayesian information criterion (empty if `logl` is empty), calculated as  $bic = -2 * logl + log(M) * numParam$ 
  - `M` is the number of groups.
  - `logl` and `numParam` are defined as in `aic`.

Note that some literature suggests that the computation of `bic` should be  $bic = -2 * logl + log(N) * numParam$ , where `N` is the number of observations. To adjust the value of the output you can redefine `bic` as follows:  $bic = bic - numel(unique(group)) + numel(Y)$

- `sebeta` — The standard errors for BETA (empty if the `ComputeStdErrors` parameter has its default value of false)

- `covb` — The estimated covariance of the parameter estimates (empty if `ComputeStdErrors` is false)
- `dfe` — The error degrees of freedom
- `pres` — The population residuals ( $y - y_{\text{population}}$ ), where `y_population` is the population predicted values
- `ires` — The population residuals ( $y - y_{\text{population}}$ ), where `y_population` is the individual predicted values
- `pwres` — The population weighted residuals
- `cwres` — The conditional weighted residuals
- `iwres` — The individual weighted residuals

## Examples

### Nonlinear Mixed-Effects Model with Stochastic EM Algorithm

Load the sample data.

```
load indomethacin
```

Fit a model to data on concentrations of the drug indomethacin in the bloodstream of six subjects over eight hours.

```
model = @(phi,t)(phi(:,1).*exp(-phi(:,2).*t)+phi(:,3).*exp(-phi(:,4).*t));  
phi0 = [1 1 1 1];  
xform = [0 1 0 1]; % log transform for 2nd and 4th parameters  
[beta,PSI,stats,br] = nlmefitsa(time,concentration,...  
    subject,[],model,phi0,'ParamTransform',xform)
```

```
beta =
```

```
    0.8563  
   -0.7950  
    2.7744  
    1.0772
```

```
PSI =
```

```
0.0529      0      0      0
      0 0.0220      0      0
      0      0 0.4762      0
      0      0      0 0.0120
```

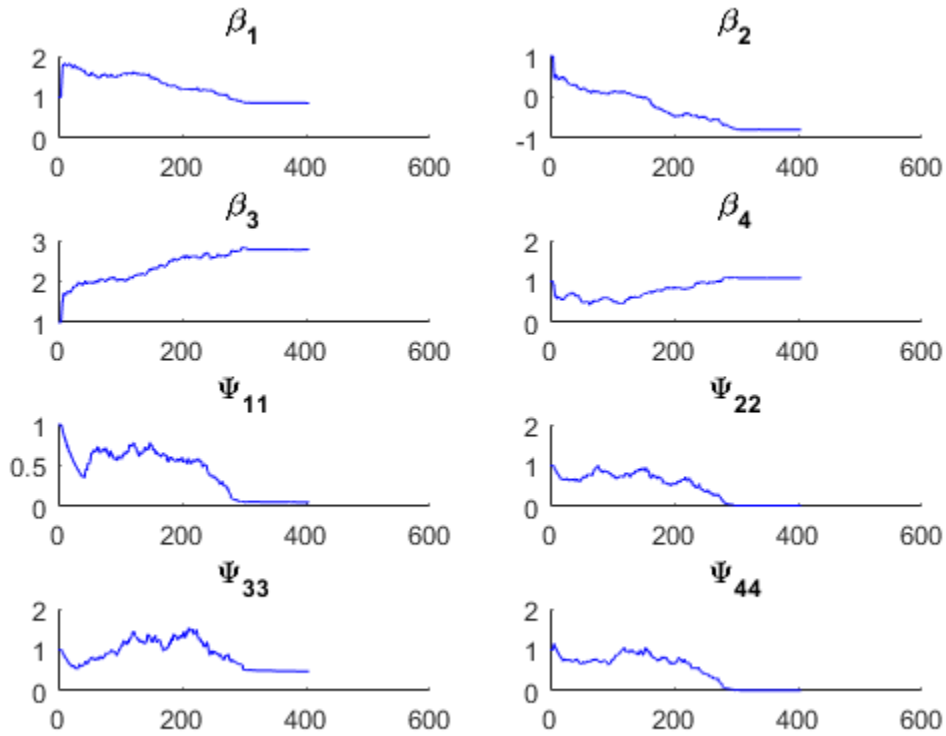
```
stats =
```

```
struct with fields:
```

```
logl: []
aic: []
bic: []
sebeta: []
dfe: 57
covb: []
errorparam: 0.0809
rmse: 0.0775
ires: [66x1 double]
pres: [66x1 double]
iwres: [66x1 double]
pwres: [66x1 double]
cwres: [66x1 double]
```

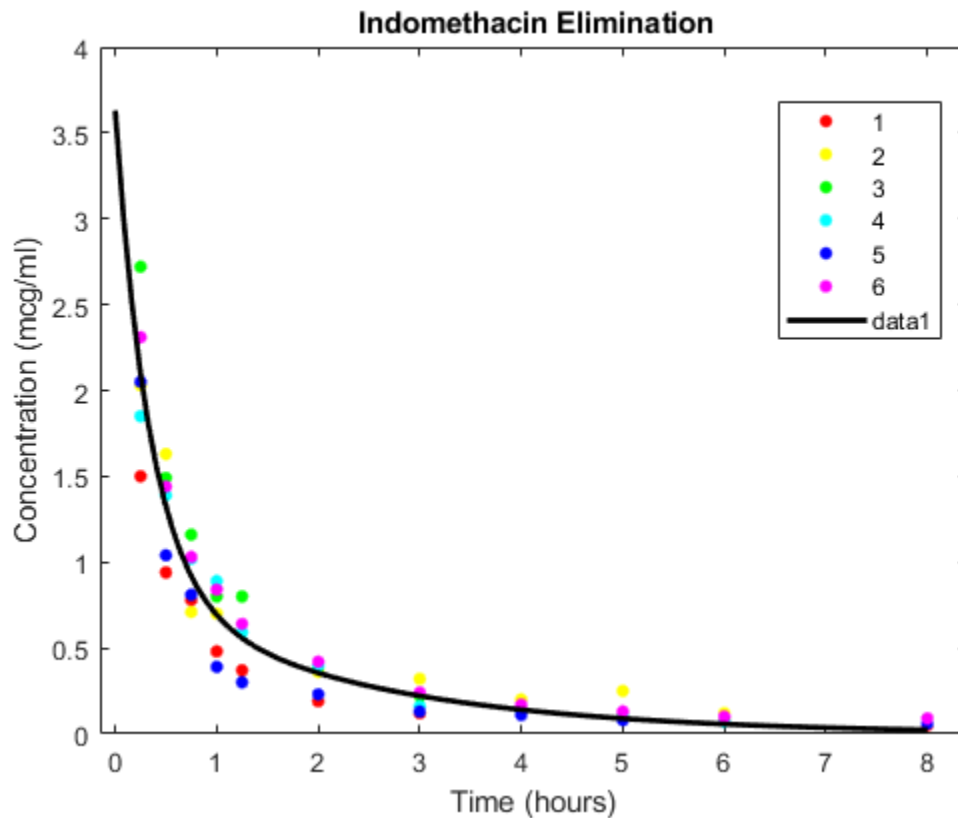
```
br =
```

```
-0.2255  0.0063  0.1600  0.1773 -0.3269  0.1157
 0.0350 -0.1384  0.0058  0.0431  0.0093 -0.0453
-0.7557 -0.0550  0.8736 -0.7875  0.5304  0.1727
-0.0010 -0.0198  0.0137 -0.0757  0.0478 -0.0076
```



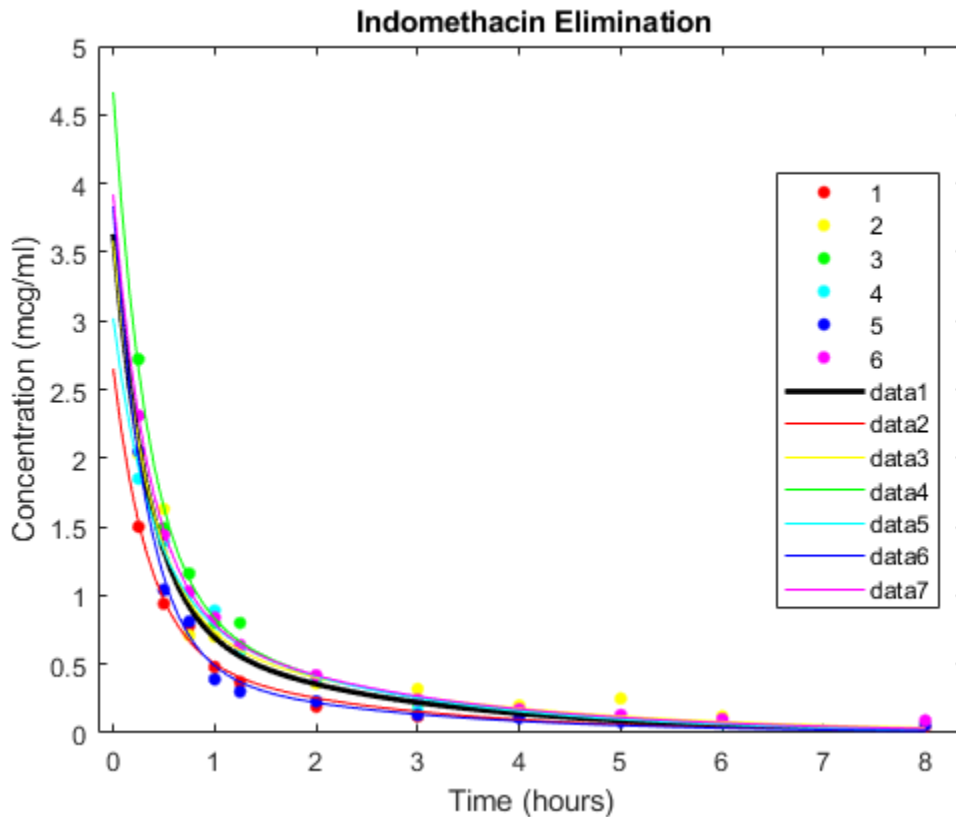
Plot the data along with an overall population fit

```
clf
phi = [beta(1), exp(beta(2)), beta(3), exp(beta(4))];
h = gscatter(time, concentration, subject);
xlabel('Time (hours)')
ylabel('Concentration (mcg/ml)')
title('{\bf Indomethacin Elimination}')
xx = linspace(0, 8);
line(xx, model(phi, xx), 'linewidth', 2, 'color', 'k')
```



Plot individual curves based on random-effect estimates.

```
for j=1:6
    phir = [beta(1)+br(1,j), exp(beta(2)+br(2,j)), ...
            beta(3)+br(3,j), exp(beta(4)+br(4,j))];
    line(xx,model(phir,xx), 'color',get(h(j), 'color'))
end
```



## Algorithms

In order to estimate the parameters of a nonlinear mixed effects model, we would like to choose the parameter values that maximize a likelihood function. These values are called the maximum likelihood estimates. The likelihood function can be written in the form

$$p(y | \beta, \sigma^2, \Sigma) = \int p(y | \beta, b, \sigma^2) p(b | \Sigma) db$$

where

- $y$  is the response data



- $\beta$  is the vector of population coefficients
- $\sigma^2$  is the residual variance
- $\Sigma$  is the covariance matrix for the random effects
- $b$  is the set of unobserved random effects

Each  $p()$  function on the right-hand-side is a normal (Gaussian) likelihood function that may depend on covariates.

Since the integral does not have a closed form, it is difficult to find parameters that maximize it. Delyon, Lavielle, and Moulines [1] proposed to find the maximum likelihood estimates using an Expectation-Maximization (EM) algorithm in which the E step is replaced by a stochastic procedure. They called their algorithm SAEM, for Stochastic Approximation EM. They demonstrated that this algorithm has desirable theoretical properties, including convergence under practical conditions and convergence to a local maximum of the likelihood function. Their proposal involves three steps:

- 1 Simulation: Generate simulated values of the random effects  $b$  from the posterior density  $p(b | \Sigma)$  given the current parameter estimates.
- 2 Stochastic approximation: Update the expected value of the loglikelihood function by taking its value from the previous step, and moving part way toward the average value of the loglikelihood calculated from the simulated random effects.
- 3 Maximization step: Choose new parameter estimates to maximize the loglikelihood function given the simulated values of the random effects.

## References

- [1] Delyon, B., M. Lavielle, and E. Moulines, *Convergence of a stochastic approximation version of the EM algorithm*, Annals of Statistics, 27, 94-128, 1999.
- [2] Mentré, France, and Marc Lavielle, *Stochastic EM algorithms in population PKPD analyses*, 2008.

## See Also

`nlminfit` | `nlmefit` | `nlpredci`

## **Topics**

“Mixed-Effects Models” on page 13-22

“Grouping Variables” on page 2-59

**Introduced in R2010a**

## NegativeLogLikelihood property

**Class:** `gmdistribution`

Negative of log-likelihood

### Description

The negative of the log-likelihood of the data.

---

**Note** This property applies only to `gmdistribution` objects constructed with `fitgmdist`.

---

## NLogL property

**Class:** ProbDistParametric

Read-only value specifying negative log likelihood for input data to ProbDistParametric object

## Description

NLogL is a read-only property of the ProbDistParametric class. NLogL is a value specifying the negative log likelihood for input data used to fit a distribution represented by a ProbDistParametric object.

## Values

The value is a numeric scalar for a distribution fit to input data, that is, a distribution created using the `fitdist` function. This property is empty for distributions created without fitting to data, that is, by using the ProbDistUnivParam constructor. Use this information to view and compare the negative log likelihood for input data supplied to create distributions.

## NLogL property

**Class:** ProbDistUnivKernel

Read-only value specifying negative log likelihood for input data to ProbDistUnivKernel object

## Description

NLogL is a read-only property of the ProbDistUnivKernel class. NLogL is a value specifying the negative log likelihood for input data used to fit a distribution represented by a ProbDistUnivKernel object.

## Values

The value is a numeric scalar for a distribution fit to input data, that is, a distribution created using the `fitdist` function. Use this information to view and compare the negative log likelihood for input data used to create distributions.

## nlparci

Nonlinear regression parameter confidence intervals

### Syntax

```
ci = nlparci(beta, resid, 'covar', sigma)
ci = nlparci(beta, resid, 'jacobian', J)
ci = nlparci(..., 'alpha', alpha)
```

### Description

`ci = nlparci(beta, resid, 'covar', sigma)` returns the 95% confidence intervals `ci` for the nonlinear least squares parameter estimates `beta`. Before calling `nlparci`, use `nlinfit` to fit a nonlinear regression model and get the coefficient estimates `beta`, residuals `resid`, and estimated coefficient covariance matrix `sigma`.

`ci = nlparci(beta, resid, 'jacobian', J)` is an alternative syntax that also computes 95% confidence intervals. `J` is the Jacobian computed by `nlinfit`. If the 'robust' option is used with `nlinfit`, use the 'covar' input rather than the 'jacobian' input so that the required `sigma` parameter takes the robust fitting into account.

`ci = nlparci(..., 'alpha', alpha)` returns 100(1-alpha)% confidence intervals.

`nlparci` treats NaNs in `resid` or `J` as missing values, and ignores the corresponding observations.

The confidence interval calculation is valid for systems where the length of `resid` exceeds the length of `beta` and `J` has full column rank. When `J` is ill-conditioned, confidence intervals may be inaccurate.

## Examples

### Fit to exponential decay

Suppose you have data, and want to fit a model of the form

$$y_i = a_1 + a_2 \exp(-a_3 x_i) + \varepsilon_i.$$

Here the  $a_i$  are the parameters you want to estimate,  $x_i$  are the data points, the  $y_i$  are the responses, and the  $\varepsilon_i$  are noise terms.

- 1 Write a function handle that represents the model:

```
mdl = @(a,x) (a(1) + a(2)*exp(-a(3)*x));
```

- 2 Generate synthetic data with parameters  $a = [1; 3; 2]$ , with the  $x$  data points distributed exponentially with parameter 2, and normally distributed noise with standard deviation 0.1:

```
rng(9845,'twister') % for reproducibility
a = [1;3;2];
x = exprnd(2,100,1);
epsn = normrnd(0,0.1,100,1);
y = mdl(a,x) + epsn;
```

- 3 Fit the model to data starting from the arbitrary guess  $a_0 = [2; 2; 2]$ :

```
a0 = [2;2;2];
[ahat,r,J,cov,mse] = nlinfit(x,y,mdl,a0);
ahat
```

```
ahat =
    1.0153
    3.0229
    2.1070
```

- 4 Check whether  $[1; 3; 2]$  is in a 95% confidence interval using the Jacobian argument in `nlparci`:

```
ci = nlparci(ahat,r,'Jacobian',J)
```

```
ci =
    0.9869    1.0438
    2.9401    3.1058
    1.9963    2.2177
```

- 5 You can obtain the same result using the covariance argument:

```
ci = nlparci(ahat,r,'covar',cov)
```

```
ci =  
  0.9869    1.0438  
  2.9401    3.1058  
  1.9963    2.2177
```

## See Also

`nlinfit` | `nlpredci`

**Introduced before R2006a**



## nlpredci

Nonlinear regression prediction confidence intervals

### Syntax

```
[Ypred,delta] = nlpredci(modelfun,X,beta,R,'Covar',CovB)
[Ypred,delta] = nlpredci(modelfun,X,beta,R,'Covar',CovB,Name,Value)

[Ypred,delta] = nlpredci(modelfun,X,beta,R,'Jacobian',J)
[Ypred,delta] = nlpredci(modelfun,X,beta,R,'Jacobian',J,Name,Value)
```

### Description

`[Ypred,delta] = nlpredci(modelfun,X,beta,R,'Covar',CovB)` returns predictions, `Ypred`, and 95% confidence interval half-widths, `delta`, for the nonlinear regression model `modelfun` at input values `X`. Before calling `nlpredci`, use `nlinfit` to fit `modelfun` and get the estimated coefficients, `beta`, residuals, `R`, and variance-covariance matrix, `CovB`.

`[Ypred,delta] = nlpredci(modelfun,X,beta,R,'Covar',CovB,Name,Value)` uses additional options specified by one or more name-value pair arguments.

`[Ypred,delta] = nlpredci(modelfun,X,beta,R,'Jacobian',J)` returns predictions, `Ypred`, and 95% confidence interval half-widths, `delta`, for the nonlinear regression model `modelfun` at input values `X`. Before calling `nlpredci`, use `nlinfit` to fit `modelfun` and get the estimated coefficients, `beta`, residuals, `R`, and Jacobian, `J`.

If you use a robust option with `nlinfit`, then you should use the `Covar` syntax rather than the `Jacobian` syntax. The variance-covariance matrix, `CovB`, is required to properly take the robust fitting into account.

`[Ypred,delta] = nlpredci(modelfun,X,beta,R,'Jacobian',J,Name,Value)` uses additional options specified by one or more name-value pair arguments.

## Examples

### Confidence Interval for Nonlinear Regression Curve

Load sample data.

```
S = load('reaction');  
X = S.reactants;  
y = S.rate;  
beta0 = S.beta;
```

Fit the Hougen-Watson model to the rate data using the initial values in `beta0`.

```
[beta,R,J] = nlinfit(X,y,@hougen,beta0);
```

Obtain the predicted response and 95% confidence interval half-width for the value of the curve at average reactant levels.

```
[ypred,delta] = nlpredci(@hougen,mean(X),beta,R,'Jacobian',J)
```

```
ypred =  
  
    5.4622
```

```
delta =  
  
    0.1921
```

Compute the 95% confidence interval for the value of the curve.

```
[ypred-delta,ypred+delta]
```

```
ans =  
  
    5.2702    5.6543
```

## Prediction Interval for New Observation

Load sample data.

```
S = load('reaction');  
X = S.reactants;  
y = S.rate;  
beta0 = S.beta;
```

Fit the Hougen-Watson model to the rate data using the initial values in beta0.

```
[beta,R,J] = nlinfit(X,y,@hougen,beta0);
```

Obtain the predicted response and 95% prediction interval half-width for a new observation with reactant levels [100,100,100].

```
[ypred,delta] = nlpredci(@hougen,[100,100,100],beta,R,'Jacobian',J,...  
                        'PredOpt','observation')
```

```
ypred =  
  
    1.8346
```

```
delta =  
  
    0.5101
```

Compute the 95% prediction interval for the new observation.

```
[ypred-delta,ypred+delta]
```

```
ans =  
  
    1.3245    2.3447
```

### Simultaneous Confidence Intervals for Robust Fit Curve

Generate sample data from the nonlinear regression model  $y = b_1 + b_2 \cdot \exp\{b_3 x\} + \epsilon$ , where  $b_1$ ,  $b_2$ , and  $b_3$  are coefficients, and the error term is normally distributed with mean 0 and standard deviation 0.5.

```
modelfun = @(b,x) (b(1)+b(2)*exp(-b(3)*x));

rng('default') % for reproducibility
b = [1;3;2];
x = exprnd(2,100,1);
y = modelfun(b,x) + normrnd(0,0.5,100,1);
```

Fit the nonlinear model using robust fitting options.

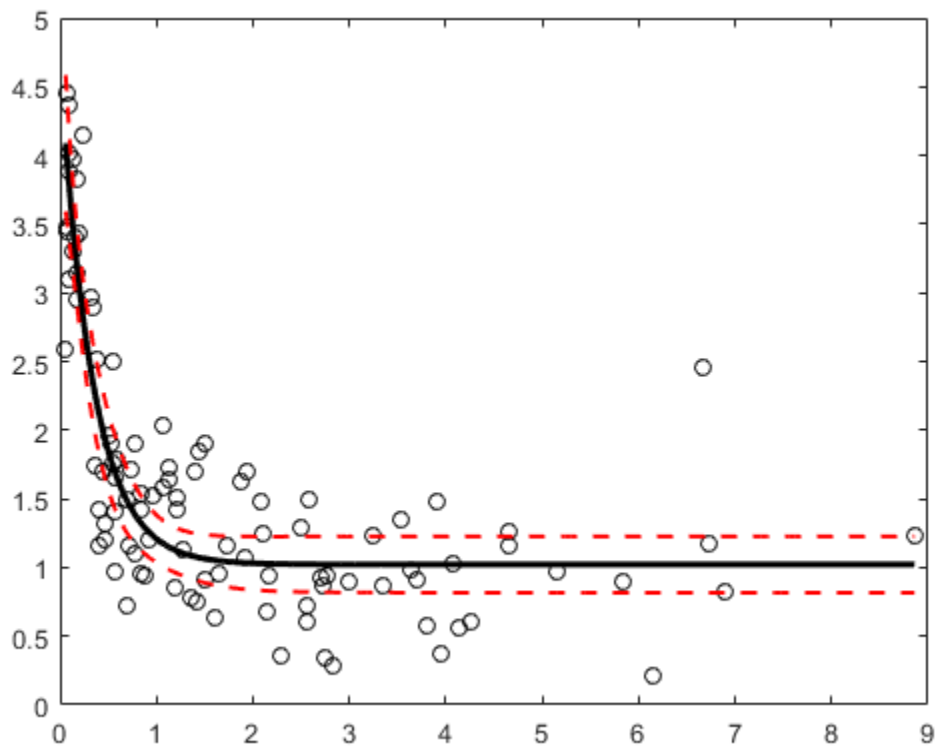
```
opts = statset('nlinfit');
opts.RobustWgtFun = 'bisquare';
beta0 = [2;2;2];
[beta,R,J,CovB,MSE] = nlinfit(x,y,modelfun,beta0,opts);
```

Plot the fitted regression model and simultaneous 95% confidence bounds.

```
xrange = min(x):.01:max(x);
[ypred,delta] = nlpredci(modelfun,xrange,beta,R,'Covar',CovB,...
    'MSE',MSE,'SimOpt','on');

lower = ypred - delta;
upper = ypred + delta;

figure()
plot(x,y,'ko') % observed data
hold on
plot(xrange,ypred,'k','LineWidth',2)
plot(xrange,[lower;upper],'r--','LineWidth',1.5)
```



### Confidence Interval Using Observation Weights

Load sample data.

```
S = load('reaction');  
X = S.reactants;  
y = S.rate;  
beta0 = S.beta;
```

Specify a function handle for observation weights, then fit the Hougen-Watson model to the rate data using the specified observation weights function.

```
a = 1; b = 1;
weights = @(yhat) 1./((a + b*abs(yhat)).^2);
[beta,R,J,CovB] = nlinfit(X,y,@hougen,beta0,'Weights',weights);
```

Compute the 95% prediction interval for a new observation with reactant levels [100,100,100] using the observation weight function.

```
[ypred,delta] = nlpredci(@hougen,[100,100,100],beta,R,'Jacobian',J,...
    'PredOpt','observation','Weights',weights);
[ypred-delta,ypred+delta]
```

```
ans =
```

```
    1.5264    2.1033
```

### Confidence Interval Using Nonconstant Error Model

Load sample data.

```
S = load('reaction');
X = S.reactants;
y = S.rate;
beta0 = S.beta;
```

Fit the Hougen-Watson model to the rate data using the combined error variance model.

```
[beta,R,J,CovB,MSE,S] = nlinfit(X,y,@hougen,beta0,'ErrorModel','combined');
```

Compute the 95% prediction interval for a new observation with reactant levels [100,100,100] using the fitted error variance model.

```
[ypred,delta] = nlpredci(@hougen,[100,100,100],beta,R,'Jacobian',J,...
    'PredOpt','observation','ErrorModelInfo',S);
[ypred-delta,ypred+delta]
```

```
ans =
```

1.3245 2.3447

## Input Arguments

### **modelfun** — Nonlinear regression model function

function handle

Nonlinear regression model function, specified as a function handle. `modelfun` must accept two input arguments, a coefficient vector and an array `X`—in that order—and return a vector of fitted response values.

For example, to specify the `hougen` nonlinear regression function, use the function handle `@hougen`.

Data Types: `function_handle`

### **x** — Input values for predictions

matrix

Input values for predictions, specified as a matrix. `nlpredci` makes a prediction for the covariates in each row of `x`. There should be a column in `x` for each coefficient in the model.

Data Types: `single` | `double`

### **beta** — Estimated regression coefficients

vector returned by `nlinfit`

Estimated regression coefficients, specified as the vector of fitted coefficients returned by a previous call to `nlinfit`.

Data Types: `single` | `double`

### **R** — Residuals

vector returned by `nlinfit`

Residuals for the fitted `modelfun`, specified as the vector of residuals returned by a previous call to `nlinfit`.

### **CovB** — Estimated variance-covariance matrix

matrix returned by `nlinfit`

Estimated variance-covariance matrix for the fitted coefficients, `beta`, specified as the variance-covariance matrix returned by a previous call to `nlinfit`.

### **J** — Estimated Jacobian

matrix returned by `nlinfit`

Estimated Jacobian of the nonlinear regression model, `modelfun`, specified as the Jacobian matrix returned by a previous call to `nlinfit`.

## **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'Alpha', 0.1, 'PredOpt', 'observation'` specifies 90% prediction intervals for new observations.

### **Alpha** — Significance level

0.05 (default) | scalar value in the range (0,1)

Significance level for the confidence interval, specified as the comma-separated pair consisting of `'Alpha'` and a scalar value in the range (0,1). If `Alpha` has value  $\alpha$ , then `nlpredci` returns intervals with  $100 \times (1 - \alpha)\%$  confidence level.

The default confidence level is 95% ( $\alpha = 0.05$ ).

Example: `'Alpha', 0.1`

Data Types: `single` | `double`

### **ErrorModelInfo** — Information about error model fit

structure returned by `nlinfit`

Information about the error model fit, specified as the comma-separated pair consisting of `'ErrorModelInfo'` and a structure returned by a previous call to `nlinfit`.

`ErrorModelInfo` only has an effect on the returned prediction interval when `PredOpt` has the value `'observation'`. If you do not use `ErrorModelInfo`, then `nlpredci` assumes the error variance model is `'constant'`.



The error model structure returned by `nlinfit` has the following fields:

<code>ErrorModel</code>	Chosen error model
<code>ErrorParameters</code>	Estimated error parameters
<code>ErrorVariance</code>	Function handle that accepts an $N$ -by- $p$ matrix, $x$ , and returns an $N$ -by-1 vector of error variances using the estimated error model
<code>MSE</code>	Mean squared error
<code>ScheffeSimPred</code>	Scheffé parameter for simultaneous prediction intervals when using the estimated error model
<code>WeightFunction</code>	Logical with value <code>true</code> if you used a custom weight function previously in <code>nlinfit</code>
<code>FixedWeights</code>	Logical with value <code>true</code> if you used fixed weights previously in <code>nlinfit</code>
<code>RobustWeightFunction</code>	Logical with value <code>true</code> if you used robust fitting previously in <code>nlinfit</code>

### **MSE — Mean squared error**

MSE returned by `nlinfit`

Mean squared error (MSE) for the fitted nonlinear regression model, specified as the comma-separated pair consisting of 'MSE' and the MSE value returned by a previous call to `nlinfit`.

If you use a robust option with `nlinfit`, then you must specify the MSE when predicting new observations to properly take the robust fitting into account. If you do not specify the MSE, then `nlpredci` computes the MSE from the residuals,  $R$ , and does not take the robust fitting into account.

For example, if `mse` is the MSE value returned by `nlinfit`, then you can specify `'MSE', mse`.

Data Types: `single` | `double`

### **PredOpt — Prediction interval to compute**

'curve' (default) | 'observation'

Prediction interval to compute, specified as the comma-separated pair consisting of 'PredOpt' and either 'curve' or 'observation'.

- If you specify the value `'curve'`, then `nlpredci` returns confidence intervals for the estimated curve (function value) at the observations `X`.
- If you specify the value `'observation'`, then `nlpredci` returns prediction intervals for new observations at `X`.

If you specify `'observation'` after using a robust option with `nlmfit`, then you must also specify a value for `MSE` to provide the robust estimate of the mean squared error.

Example: `'PredOpt', 'observation'`

Data Types: `char`

#### **SimOpt** — Indicator for specifying simultaneous bounds

`'off'` (default) | `'on'`

Indicator for specifying simultaneous bounds, specified as the comma-separated pair consisting of `'SimOpt'` and either `'off'` or `'on'`. Use the value `'off'` to compute nonsimultaneous bounds, and `'on'` for simultaneous bounds.

#### **Weights** — Observation weights

vector | function handle

Observation weights, specified as the comma-separated pair consisting of `'Weights'` and a vector of positive scalar values or a function handle. The default is no weights.

- If you specify a vector of weights, then it must have the same number of elements as the number of observations (rows) in `X`.
- If you specify a function handle for the weights, then it must accept a vector of predicted response values as input, and return a vector of real positive weights as output.

Given weights, `W`, `nlpredci` estimates the error variance at observation `i` by  $mse * (1/W(i))$ , where `mse` is the mean squared error value specified using `MSE`.

Example: `'Weights', @WFun`

Data Types: `double` | `single` | `function_handle`

## Output Arguments

### **Ypred** — Predicted responses

vector

Predicted responses, returned as a vector with the same number of rows as  $X$ .

### **delta** — Confidence interval half-widths

vector

Confidence interval half-widths, returned as a vector with the same number of rows as  $X$ . By default, `delta` contains the half-widths for nonsimultaneous 95% confidence intervals for `modelfun` at the observations in  $X$ . You can compute the lower and upper bounds of the confidence intervals as  $Y_{\text{pred}} - \text{delta}$  and  $Y_{\text{pred}} + \text{delta}$ , respectively.

If 'PredOpt' has value 'observation', then `delta` contains the half-widths for prediction intervals of new observations at the values in  $X$ .

## Definitions

### Confidence Intervals for Estimable Predictions

When the estimated model Jacobian is not of full rank, then it might not be possible to construct sensible confidence intervals at all prediction points. In this case, `nlpredci` still tries to construct confidence intervals for any estimable prediction points.

For example, suppose you fit the linear function  $f(\mathbf{x}_i, \beta) = \beta_1 x_{i1} + \beta_2 x_{i2} + \beta_3 x_{i3}$  at the points in the design matrix

$$\mathbf{X} = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \\ 1 & 0 & 1 \\ 1 & 0 & 1 \end{pmatrix}.$$

The estimated Jacobian at the values in  $\mathbf{X}$  is the design matrix itself,  $\mathbf{J} = \mathbf{X}$ . Thus, the Jacobian is not of full rank:

```
rng('default') % For reproducibility
y = randn(6,1);

linfun = @(b,x) x*b;
beta0 = [1;1;1];
X = [repmat([1 1 0],3,1); repmat([1 0 1],3,1)];

[beta,R,J] = nlinfit(X,y,linfun,beta0);
```

```
Warning: The Jacobian at the solution is ill-conditioned, and
some model parameters may not be estimated well (they are not
identifiable). Use caution in making predictions.
> In nlinfit at 283
```

In this example, `nlpredci` can only compute prediction intervals at points that satisfy the linear relationship

$$x_{i1} = x_{i2} + x_{i3}.$$

If you try to compute confidence intervals for predictions at nonidentifiable points, `nlpredci` returns NaN for the corresponding interval half-widths:

```
xpred = [1 1 1;0 1 -1;2 1 1];
[ypred,delta] = nlpredci(linfun,xpred,beta,R,'Jacobian',J)

ypred =

    -0.0035
     0.0798
    -0.0047

delta =

     NaN
    3.8102
    3.8102
```

Here, the first element of `delta` is NaN because the first row in `xpred` does not satisfy the required linear dependence, and is therefore not an estimable contrast.

## Tips

- To compute confidence intervals for complex parameters or data, you need to split the problem into its real and imaginary parts. When calling `nlinfit`:
  - 1 Define your parameter vector `beta` as the concatenation of the real and imaginary parts of the original parameter vector.
  - 2 Concatenate the real and imaginary parts of the response vector `Y` as a single vector.
  - 3 Modify your model function `modelfun` to accept `X` and the purely real parameter vector, and return a concatenation of the real and imaginary parts of the fitted values.

With the problem formulated this way, `nlinfit` computes real estimates, and confidence intervals are feasible.

## Algorithms

- `nlpredci` treats NaN values in the residuals,  $R$ , or the Jacobian,  $J$ , as missing values, and ignores the corresponding observations.
- If the Jacobian,  $J$ , does not have full column rank, then some of the model parameters might be nonidentifiable. In this case, `nlpredci` tries to construct confidence intervals for estimable predictions on page 32-4341, and returns NaN for those that are not.

## References

- [1] Lane, T. P. and W. H. DuMouchel. “Simultaneous Confidence Intervals in Multiple Regression.” *The American Statistician*. Vol. 48, No. 4, 1994, pp. 315–321.
- [2] Seber, G. A. F., and C. J. Wild. *Nonlinear Regression*. Hoboken, NJ: Wiley-Interscience, 2003.

## See Also

NonLinearModel | `nlinfit` | `nlparci`

**Introduced before R2006a**

# nnmf

Nonnegative matrix factorization

## Syntax

```
[W, H] = nnmf(A, k)
[W, H] = nnmf(A, k, param1, val1, param2, val2, ...)
[W, H, D] = nnmf(...)
```

## Description

`[W, H] = nnmf(A, k)` factors the nonnegative  $n$ -by- $m$  matrix  $A$  into nonnegative factors  $W$  ( $n$ -by- $k$ ) and  $H$  ( $k$ -by- $m$ ). The factorization is not exact;  $W^*H$  is a lower-rank approximation to  $A$ . The factors  $W$  and  $H$  are chosen to minimize the root-mean-squared residual  $D$  between  $A$  and  $W^*H$ :

$$D = \text{norm}(A - W^*H, 'fro') / \sqrt{N * M}$$

The factorization uses an iterative method starting with random initial values for  $W$  and  $H$ . Because the root-mean-squared residual  $D$  may have local minima, repeated factorizations may yield different  $W$  and  $H$ . Sometimes the algorithm converges to a solution of lower rank than  $k$ , which may indicate that the result is not optimal.

$W$  and  $H$  are normalized so that the rows of  $H$  have unit length. The columns of  $W$  are ordered by decreasing length.

`[W, H] = nnmf(A, k, param1, val1, param2, val2, ...)` specifies optional parameter name/value pairs from the following table.

Parameter	Value
'algorithm'	<p>Either 'als' (the default) to use an alternating least-squares algorithm, or 'mult' to use a multiplicative update algorithm.</p> <p>In general, the 'als' algorithm converges faster and more consistently. The 'mult' algorithm is more sensitive to initial values, which makes it a good choice when using 'replicates' to find W and H from multiple random starting values.</p>
'w0'	An $n$ -by- $k$ matrix to be used as the initial value for W.
'h0'	A $k$ -by- $m$ matrix to be used as the initial value for H.



Parameter	Value
'options'	<p>An options structure as created by the <code>statset</code> function. <code>nnmf</code> uses the following fields of the options structure:</p> <ul style="list-style-type: none"> <li>• <code>Display</code> — Level of display. Choices: <ul style="list-style-type: none"> <li>• <code>'off'</code> (default) — No display</li> <li>• <code>'final'</code> — Display final result</li> <li>• <code>'iter'</code> — Iterative display of intermediate results</li> </ul> </li> <li>• <code>MaxIter</code> — Maximum number of iterations. Default is 100. Unlike in optimization settings, reaching <code>MaxIter</code> iterations is treated as convergence.</li> <li>• <code>TolFun</code> — Termination tolerance on change in size of the residual. Default is <math>1e-4</math>.</li> <li>• <code>TolX</code> — Termination tolerance on relative change in the elements of <code>W</code> and <code>H</code>. Default is <math>1e-4</math>.</li> <li>• <code>UseParallel</code> — Set to <code>true</code> to compute in parallel. Default is <code>false</code>.</li> <li>• <code>UseSubstreams</code> — Set to <code>true</code> to compute in parallel in a reproducible fashion. Default is <code>false</code>. To compute reproducibly, set <code>Streams</code> to a type allowing substreams: <code>'mlfg6331_64'</code> or <code>'mrg32k3a'</code>.</li> <li>• <code>Streams</code> — A <code>RandStream</code> object or cell array of such objects. If you do not specify <code>Streams</code>, <code>nnmf</code> uses the default stream or streams. If you choose to specify <code>Streams</code>, use a single object except in the case <ul style="list-style-type: none"> <li>• You have an open <code>Parallel</code> pool</li> <li>• <code>UseParallel</code> is <code>true</code></li> <li>• <code>UseSubstreams</code> is <code>false</code></li> </ul> <p>In that case, use a cell array the same size as the <code>Parallel</code> pool.</p> </li> </ul>

Parameter	Value
'replicates'	The number of times to repeat the factorization, using new random starting values for W and H, except at the first replication if 'w0' and 'h0' are given. This is most beneficial with the 'mult' algorithm. The default is 1.

`[W,H,D] = nnmf(...)` also returns D, the root mean square residual.

## Examples

### Nonnegative Rank-Two Approximation and Biplot

Load the sample data.

```
load fisheriris
```

Compute a nonnegative rank-two approximation of the measurements of the four variables in Fisher's iris data.

```
rng(1) % For reproducibility
[W,H] = nnmf(meas,2);
H
```

H =

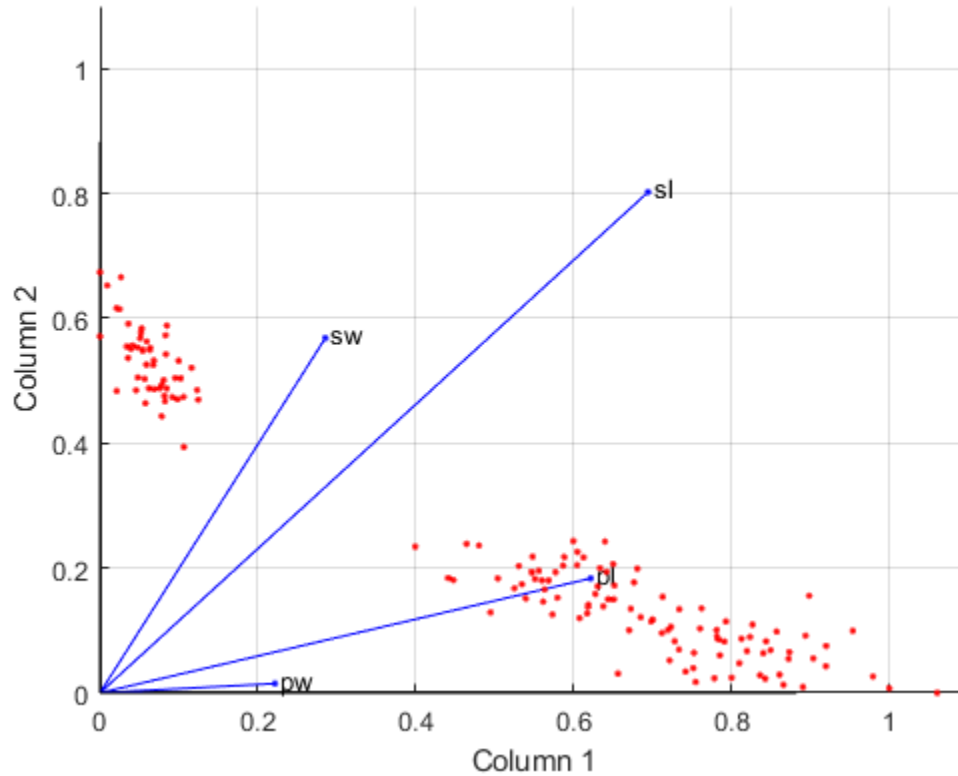
```
    0.6945    0.2856    0.6220    0.2218
    0.8020    0.5683    0.1834    0.0149
```

The first and third variables in `meas` (sepal length and petal length, with coefficients 0.6945 and 0.6220, respectively) provide relatively strong weights to the first column of W. The first and second variables in `meas` (sepal length and sepal width, with coefficients 0.8020 and 0.5683) provide relatively strong weights to the second column of W.

Create a biplot of the data and the variables in `meas` in the column space of W.

```
biplot(H,'scores',W,'varlabels',{'sl','sw','pl','pw'});
axis([0 1.1 0 1.1])
```

```
xlabel('Column 1')  
ylabel('Column 2')
```



### Change Algorithm

Starting from a random array  $X$  with rank 20, try a few iterations at several replicates using the multiplicative algorithm:

```
X = rand(100,20)*rand(20,50);  
opt = statset('MaxIter',5,'Display','final');  
[W0,H0] = nnmf(X,5,'replicates',10,...  
               'options',opt,...  
               'algorithm','mult');
```

rep	iteration	rms resid	delta x
1	5	0.560887	0.0245182
2	5	0.66418	0.0364471
3	5	0.609125	0.0358355
4	5	0.608894	0.0415491
5	5	0.619291	0.0455135
6	5	0.621549	0.0299965
7	5	0.640549	0.0438758
8	5	0.673015	0.0366856
9	5	0.606835	0.0318931
10	5	0.633526	0.0319591

Final root mean square residual = 0.560887

Continue with more iterations from the best of these results using alternating least squares:

```
opt = statset('Maxiter',1000,'Display','final');
[W,H] = nnmf(X,5,'w0',W0,'h0',H0,...
            'options',opt,...
            'algorithm','als');
```

rep	iteration	rms resid	delta x
1	24	0.257336	0.00271859

Final root mean square residual = 0.257336

## References

- [1] Berry, M. W., et al. “Algorithms and Applications for Approximate Nonnegative Matrix Factorization.” *Computational Statistics and Data Analysis*. Vol. 52, No. 1, 2007, pp. 155–173.

## See Also

factoran | pca | statset

Introduced in R2008a

## nodeclass

**Class:** classregtree

Class values of nodes of classification tree

---

**Note** `classregtree` will be removed in a future release. See `fitctree`, `fitrtree`, `ClassificationTree`, or `RegressionTree` instead.

---

## Syntax

```
NAME=nodeclass(T)
NAME=nodeclass(T,J)
[NAME,ID]=nodeclass(...)
```

## Description

`NAME=nodeclass(T)` returns an  $n$ -element cell array with the names of the most probable classes in each node of the tree `T`, where  $n$  is the number of nodes in the tree. Every element of this array is a character vector equal to one of the class names returned by `classname(T)`. For regression trees, `nodeclass` returns an empty cell array.

`NAME=nodeclass(T,J)` takes an array `J` of node numbers and returns the class names for the specified nodes.

`[NAME,ID]=nodeclass(...)` also returns a numeric array with the class index for each node. The class index is determined by the order of classes `classname` returns.

## See Also

`classname` | `classregtree` | `numnodes`

## nodeerr

**Class:** classregtree

Return vector of node errors

---

**Note** `classregtree` will be removed in a future release. See `fitctree`, `fitrtree`, `ClassificationTree`, or `RegressionTree` instead.

---

## Syntax

```
e = nodeerr(t)
e = nodeerr(t,nodes)
```

## Description

`e = nodeerr(t)` returns an  $n$ -element vector  $e$  of the errors of the nodes in the tree  $t$ , where  $n$  is the number of nodes. For a regression tree, the error  $e(i)$  for node  $i$  is the variance of the observations assigned to node  $i$ . For a classification tree,  $e(i)$  is the misclassification probability for node  $i$ .

`e = nodeerr(t,nodes)` takes a vector `nodes` of node numbers and returns the errors for the specified nodes.

The error  $e$  is the so-called *resubstitution error* computed by applying the tree to the same data used to create the tree. This error is likely to under estimate the error you would find if you applied the tree to new data. The `test` function provides options to compute the error (or cost) using cross-validation or a test sample.

## Examples

Create a classification tree for Fisher's iris data:

```
load fisheriris;
```

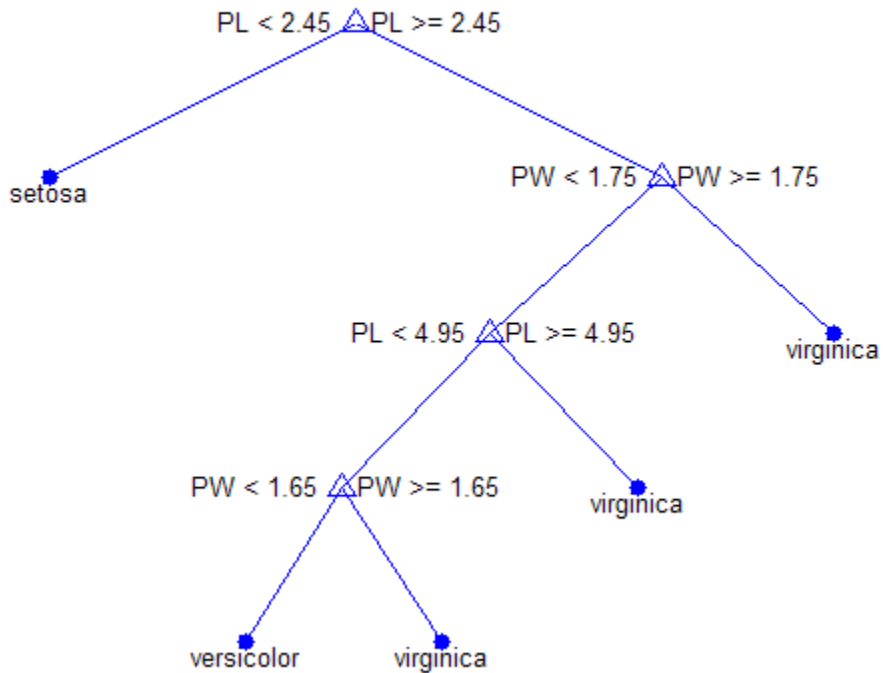
```

t = classregtree(meas,species,...
                'names',{'SL' 'SW' 'PL' 'PW'})
t =
Decision tree for classification
1 if PL<2.45 then node 2 elseif PL>=2.45 then node 3 else setosa
2 class = setosa
3 if PW<1.75 then node 4 elseif PW>=1.75 then node 5 else versicolor
4 if PL<4.95 then node 6 elseif PL>=4.95 then node 7 else versicolor
5 class = virginica
6 if PW<1.65 then node 8 elseif PW>=1.65 then node 9 else versicolor
7 class = virginica
8 class = versicolor
9 class = virginica

view(t)

```

Click to display:  Magnification:  Pruning level:



```
e = nodeerr(t)
e =
  0.6667
     0
  0.5000
  0.0926
  0.0217
  0.0208
  0.3333
     0
     0
```

## References

- [1] Breiman, L., J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Boca Raton, FL: CRC Press, 1984.

## See Also

`classregtree` | `numnodes` | `test`



# nodemean

**Class:** `classregtree`

Mean values of nodes of regression tree

---

**Note** `classregtree` will be removed in a future release. See `fitctree`, `fitrtree`, `ClassificationTree`, or `RegressionTree` instead.

---

## Syntax

`NM = nodemean(T)`  
`NM = nodemean(T, J)`

## Description

`NM = nodemean(T)` returns an  $n$ -element numeric array with mean values in each node of the tree `T`, where  $n$  is the number of nodes in the tree. Every element of this array is computed by averaging true  $Y$  values over all observations in the node. For classification trees, `nodemean` returns an empty numeric array.

`NM = nodemean(T, J)` takes an array `J` of node numbers and returns the mean values for the specified nodes.

## See Also

`classregtree` | `numnodes`

## nodeprob

**Class:** `classregtree`

Node probabilities

---

**Note** `classregtree` will be removed in a future release. See `fitctree`, `fitrtree`, `ClassificationTree`, or `RegressionTree` instead.

---

## Syntax

```
p = nodeprob(t)
p = nodeprob(t, nodes)
```

## Description

`p = nodeprob(t)` returns an  $n$ -element vector `p` of the probabilities of the nodes in the tree `t`, where  $n$  is the number of nodes. The probability of a node is computed as the proportion of observations from the original data that satisfy the conditions for the node. For a classification tree, this proportion is adjusted for any prior probabilities assigned to each class.

`p = nodeprob(t, nodes)` takes a vector `nodes` of node numbers and returns the probabilities for the specified nodes.

## Examples

Create a classification tree for Fisher's iris data:

```
load fisheriris;

t = classregtree(meas, species, ...
                'names', {'SL' 'SW' 'PL' 'PW'})
t =
Decision tree for classification
1 if PL<2.45 then node 2 elseif PL>=2.45 then node 3 else setosa
```

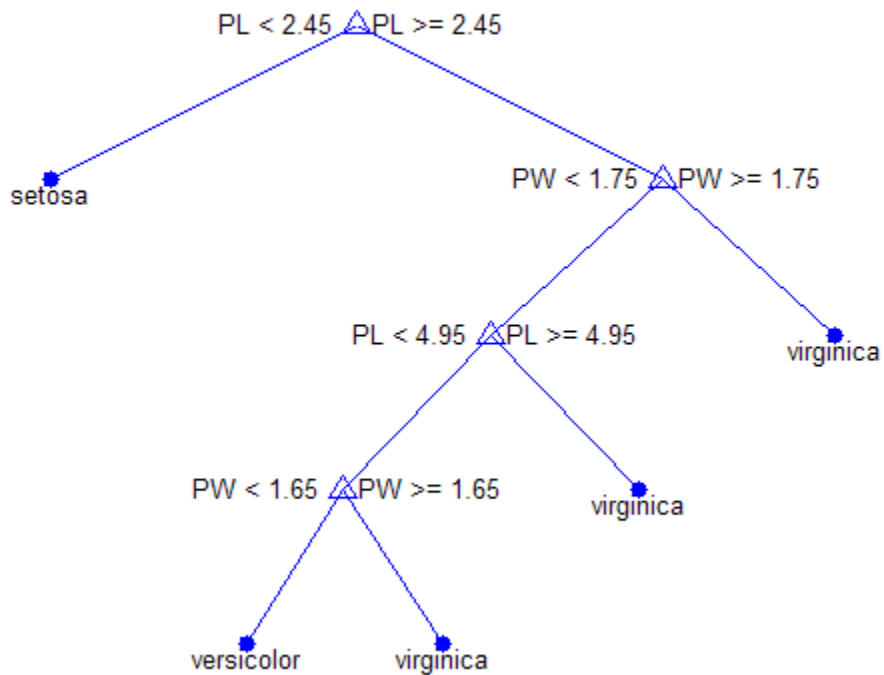
```

2 class = setosa
3 if PW<1.75 then node 4 elseif PW>=1.75 then node 5 else versicolor
4 if PL<4.95 then node 6 elseif PL>=4.95 then node 7 else versicolor
5 class = virginica
6 if PW<1.65 then node 8 elseif PW>=1.65 then node 9 else versicolor
7 class = virginica
8 class = versicolor
9 class = virginica

```

```
view(t)
```

Click to display:  Magnification:  Pruning level:



```

p = nodeprob(t)
p =
  1.0000
  0.3333
  0.6667

```

0.3600  
0.3067  
0.3200  
0.0400  
0.3133  
0.0067

## References

- [1] Breiman, L., J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Boca Raton, FL: CRC Press, 1984.

## See Also

`classregtree` | `nodesize` | `numnodes`

# nodesize

**Class:** classregtree

Return node size

---

**Note** classregtree will be removed in a future release. See fitctree, fitrtree, ClassificationTree, or RegressionTree instead.

---

## Syntax

```
sizes = nodesize(t)
sizes = nodesize(t,nodes)
```

## Description

sizes = nodesize(t) returns an  $n$ -element vector sizes of the sizes of the nodes in the tree  $t$ , where  $n$  is the number of nodes. The size of a node is defined as the number of observations from the data used to create the tree that satisfy the conditions for the node.

sizes = nodesize(t,nodes) takes a vector nodes of node numbers and returns the sizes for the specified nodes.

## Examples

Create a classification tree for Fisher's iris data:

```
load fisheriris;

t = classregtree(meas,species,...
                'names',{'SL' 'SW' 'PL' 'PW'})
t =
Decision tree for classification
1  if PL<2.45 then node 2 elseif PL>=2.45 then node 3 else setosa
2  class = setosa
3  if PW<1.75 then node 4 elseif PW>=1.75 then node 5 else versicolor
```

```

4 if PL<4.95 then node 6 elseif PL>=4.95 then node 7 else versicolor
5 class = virginica
6 if PW<1.65 then node 8 elseif PW>=1.65 then node 9 else versicolor
7 class = virginica
8 class = versicolor
9 class = virginica

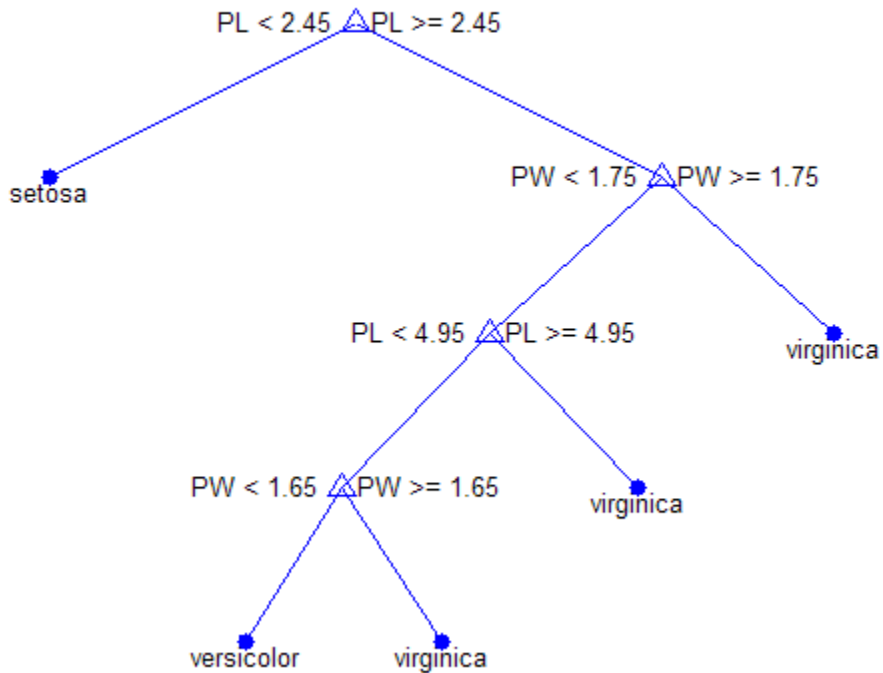
```

```
view(t)
```

Click to display:

Magnification:

Pruning level:



```
sizes = nodesize(t)
```

```
sizes =
```

```
150
```

```
50
```

```
100
```

```
54
```

```
46
```

48  
6  
47  
1

## References

- [1] Breiman, L., J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Boca Raton, FL: CRC Press, 1984.

## See Also

`classregtree` | `numnodes`

## nominal

Create nominal array

After creating a `nominal` array, you can use related functions to add, drop, or merge categories, and more.

For more information, see `nominal`.

---

**Note** The `nominal` and `ordinal` array data types might be removed in a future release. To represent ordered and unordered discrete, nonnumeric data, use the MATLAB® `categorical` data type instead.

---

## Syntax

```
B = nominal(X)
B = nominal(X, labels)
B = nominal(X, labels, levels)

B = nominal(X, labels, [], edges)
```

## Description

`B = nominal(X)` creates a nominal array `B` from the array `X`. `nominal` creates the levels of `B` from the sorted unique values in `X`, and creates default labels for them.

`B = nominal(X, labels)` labels the levels in `B` according to `labels`.

`B = nominal(X, labels, levels)` creates a nominal array with possible levels defined by `levels`.

`B = nominal(X, labels, [], edges)` creates a nominal array by binning a numeric array `X` with bin edges given by the numeric vector `edges`.



## Examples

### Create and Label Nominal Arrays

Create nominal arrays from a cell array of character vectors and from integer data, and provide explicit labels.

Create a nominal array from a cell array of character vectors with values 'r', 'g', and 'b'. Label these levels 'red', 'green', and 'blue', respectively. Note that the labels are specified according to the sorted (alphabetical) order of the elements in X.

```
X = {'r' 'b' 'g'; 'g' 'r' 'b'; 'b' 'r' 'g'}
B = nominal(X, {'blue', 'green', 'red'})
```

X =

3x3 cell array

```
{'r'}    {'b'}    {'g'}
{'g'}    {'r'}    {'b'}
{'b'}    {'r'}    {'g'}
```

B =

3x3 nominal array

```
red      blue     green
green    red      blue
blue     red      green
```

Create a nominal array from integer data with values 1 to 4, merging odd and even values into two nominal levels with labels 'odd' and 'even'. Achieve the merging by duplicating the labels.

```
X = randi([1 4],5,2)
B = nominal(X, {'odd', 'even', 'odd', 'even'})
```

X =

```
4     1
4     2
1     3
4     4
3     4
```

B =

```
5x2 nominal array

even    odd
even    even
odd     odd
even    even
odd     even
```

- “Create Nominal and Ordinal Arrays” on page 2-4
- “Plot Data Grouped by Category” on page 2-29

## Input Arguments

### **x** — Input array

numeric | logical | character | categorical | cell array of character vectors

Input array to convert to `nominal`, specified as a numeric, logical, character, or categorical array, or a cell array of character vectors. The levels of the resulting `nominal` array correspond to the sorted unique values in `X`.

### **labels** — Labels for the discrete levels

character array | cell array of character vectors

Labels for the discrete levels, specified as a character array or cell array of character vectors. By default, `nominal` assigns the labels to the levels in `B` in order according to the sorted unique values in `X`.

You can include duplicate labels in `labels` in order to merge multiple values in `X` into a single level in `B`.

Data Types: `char` | `cell`

**levels — Possible nominal levels**

vector

Possible nominal levels for the output `nominal` array, specified as a vector whose values can be compared to those in `X` using the equality operator. `nominal` assigns labels to each level from the corresponding elements of `labels`. If `X` contains any values not present in `levels`, the levels of the corresponding elements of `B` are undefined.

**edges — Bin edges**

numeric vector

Bin edges to create a nominal array by binning a numeric array, specified as a numeric vector. The uppermost bin includes values equal to the right-most edge. `nominal` assigns labels to each level in the resulting nominal array from the corresponding elements of `labels`. When you specify `edges`, it must have one more element than `labels`.

## Output Arguments

**B — Nominal array**

nominal array object

Nominal array, returned as a `nominal` array object.

By default, an element of `B` is undefined if the corresponding element of `X` is `NaN` (when `X` is numeric), an empty character vector (when `X` is a character), or undefined (when `X` is categorical). `nominal` treats such elements as “undefined” or “missing” and does not include entries for them among the possible levels. To create an explicit level for such elements instead of treating them as undefined, you must use the `levels` input argument, and include `NaN`, the empty character vector, or an undefined element.

## See Also

`ordinal`

## Topics

“Create Nominal and Ordinal Arrays” on page 2-4

“Plot Data Grouped by Category” on page 2-29

`nominal`

**Introduced in R2007a**

# nominal

Arrays for nominal data

## Description

Nominal data are discrete, nonnumeric values that do not have a natural ordering. `nominal` array objects provide efficient storage and convenient manipulation of such data, while also maintaining meaningful labels for the values.

You can manipulate `nominal` arrays much like ordinary numeric arrays, including subscripting, concatenating, and reshaping. It can be useful to use `nominal` arrays as grouping variables when the elements indicate the group an observation belongs to.

---

**Note** The `nominal` and `ordinal` array data types might be removed in a future release. To represent ordered and unordered discrete, nonnumeric data, use the MATLAB `categorical` data type instead.

---

## Creation

Create `nominal` arrays using the `nominal` function.

## Properties

### `labels` — Level labels

cell array of character vectors

This property is read-only.

Level labels, specified as a cell array of character vectors. Access labels using `getlabels`.

Data Types: `cell`

**undeflabel** — Label for undefined levels`'<undefined>'` (default)

This property is read-only.

Label for undefined levels, specified as `'<undefined>'`. You can find undefined elements in categorical arrays using `isundefined`.

## Object Functions

<code>addlevels</code>	Add levels to nominal or ordinal arrays
<code>droplevels</code>	Drop levels from a nominal or ordinal array
<code>getlabels</code>	Access categorical array labels
<code>getlevels</code>	Access categorical array levels
<code>islevel</code>	Determine if levels are in nominal or ordinal array
<code>levelcounts</code>	Element counts by level of a nominal or ordinal array
<code>mergelevels</code>	Merge levels of nominal or ordinal arrays
<code>reorderlevels</code>	Reorder levels of nominal or ordinal arrays
<code>setlabels</code>	Assign labels to levels of nominal or ordinal arrays

You can also use many other MATLAB array functions with categorical arrays. The following is a partial list. For a complete list, see “Other MATLAB Functions Supporting Nominal and Ordinal Arrays” on page 2-3.

<code>double</code>	Double-precision arrays
<code>histogram</code>	Histogram plot
<code>isequal</code>	Determine array equality
<code>isundefined</code>	Find undefined elements in categorical array
<code>pie</code>	Pie chart
<code>summary</code>	Print summary of table, timetable, or categorical array
<code>times</code>	Element-wise multiplication

## Examples

### Create and Manipulate Nominal Arrays

Create a nominal array from data in a cell array.

```
colors = nominal({'r','b','g';'g','r','b';'b','r','g'},...  
                {'blue','green','red'})
```

```
colors =  
  
3x3 nominal array  
  
    red    blue    green  
    green  red     blue  
    blue   red     green
```

Identify the elements in `colors` that are members of the level 'red'. A value of 1 in the resulting array indicates that the corresponding element of `colors` is a member of 'red'.

```
colors == 'red'
```

```
ans =  
  
3x3 logical array  
  
    1    0    0  
    0    1    0  
    0    1    0
```

Identify the elements of `colors` that are members of either 'red' or 'blue'.

```
ismember(colors,{'red' 'blue'})
```

```
ans =  
  
3x3 logical array  
  
    1    1    0  
    0    1    1  
    1    1    0
```

Merge the elements of the 'red' and 'blue' levels into a new level labeled 'purple'.

```
colors = mergelevels(colors,{'red','blue'},'purple')
```

```
colors =
```

```
3x3 nominal array
```

```
purple    purple    green
green     purple    purple
purple    purple    green
```

Display the levels of `colors`.

```
getlevels(colors)
```

```
ans =
```

```
1x2 nominal array
```

```
purple    green
```

Summarize the number of elements in each level. By default, `summary` returns counts for each column of the input array.

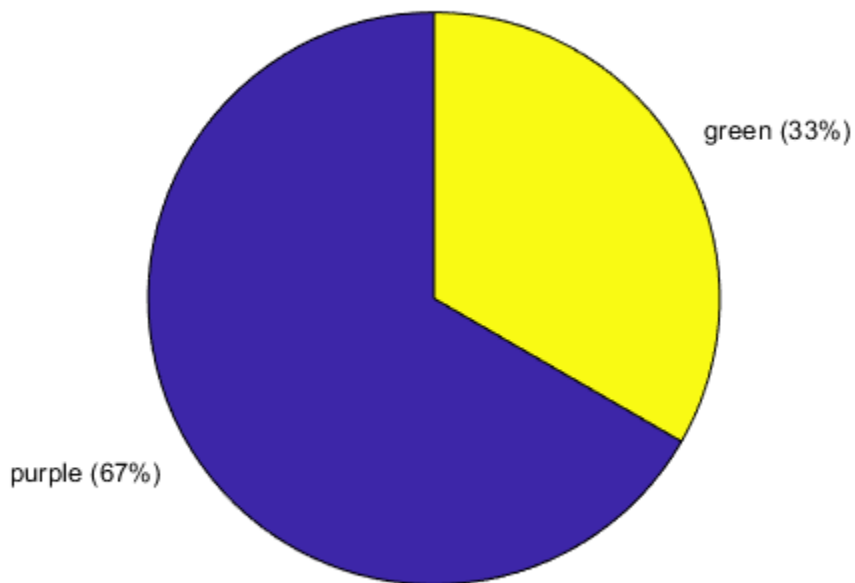
```
summary(colors)
```

```
purple    2    3    1
green     1    0    2
```

Create a pie chart for the data in `colors`.

```
pie(colors)
```





- “Create Nominal and Ordinal Arrays” on page 2-4
- “Add and Drop Category Levels” on page 2-25
- “Plot Data Grouped by Category” on page 2-29
- “Summary Statistics Grouped by Category” on page 2-44

## See Also

ordinal

## Topics

“Create Nominal and Ordinal Arrays” on page 2-4

- “Add and Drop Category Levels” on page 2-25
- “Plot Data Grouped by Category” on page 2-29
- “Summary Statistics Grouped by Category” on page 2-44
- “Advantages of Using Categorical Arrays” on page 2-50
- “Index and Search Using Categorical Arrays” on page 2-53
- “Grouping Variables” on page 2-59

# notify

**Class:** grandstream

Notify listeners of event

## Syntax

```
notify(h, 'eventname')  
notify(h, 'eventname', data)
```

## Description

`notify(h, 'eventname')` notifies listeners added to the event named `eventname` on handle object array `h` that the event is taking place. `h` is the array of handles to objects triggering the event, and `eventname` must be a character vector.

`notify(h, 'eventname', data)` provides a way of encapsulating information about an event which can then be accessed by each registered listener. `data` must belong to the `event.eventdata` class.

## See Also

`addlistener` | `event.EventData` | `events` | `grandstream`

## NonLinearModel class

Nonlinear regression model class

### Description

An object comprising training data, model description, diagnostic information, and fitted coefficients for a nonlinear regression. Predict model responses with the `predict` or `feval` methods.

### Construction

`nlm = fitnlm(tbl,modelfun,beta0)` or `nlm = fitnlm(X,y,modelfun,beta0)`  
create a nonlinear model of a table or dataset array `tbl`, or of the responses `y` to a data matrix `X`. For details, see `fitnlm`.

### Input Arguments

#### **tbl** — Input data

table | dataset array

Input data, specified as a table or dataset array. When `modelspec` is a formula, it specifies the variables to be used as the predictors and response. Otherwise, if you do not specify the predictor and response variables, the last variable is the response variable and the others are the predictor variables by default.

Predictor variables can be numeric, or any grouping variable type, such as logical or categorical (see “Grouping Variables” on page 2-59). The response must be numeric or logical.

To set a different column as the response variable, use the `ResponseVar` name-value pair argument. To use a subset of the columns as predictors, use the `PredictorVars` name-value pair argument.

**x — Predictor variables**

matrix

Predictor variables, specified as an  $n$ -by- $p$  matrix, where  $n$  is the number of observations and  $p$  is the number of predictor variables. Each column of  $X$  represents one variable, and each row represents one observation.

By default, there is a constant term in the model, unless you explicitly remove it, so do not include a column of 1s in  $X$ .

Data Types: `single` | `double` | `logical`

**y — Response variable**

vector

Response variable, specified as an  $n$ -by-1 vector, where  $n$  is the number of observations. Each entry in  $y$  is the response for the corresponding row of  $X$ .

Data Types: `single` | `double` | `logical`

**modelfun — Functional form of the model**

function handle | character vector of the form ' $y \sim f(b_1, b_2, \dots, b_j, x_1, x_2, \dots, x_k)$ '

Functional form of the model, specified as either of the following.

- Function handle `@modelfun` or `@(b, x) modelfun`, where
  - $b$  is a coefficient vector with the same number of elements as `beta0`.
  - $x$  is a matrix with the same number of columns as  $X$  or the number of predictor variable columns of `tbl`.

`modelfun(b, x)` returns a column vector that contains the same number of rows as  $x$ . Each row of the vector is the result of evaluating `modelfun` on the corresponding row of  $x$ . In other words, `modelfun` is a vectorized function, one that operates on all data rows and returns all evaluations in one function call. `modelfun` should return real numbers to obtain meaningful coefficients.

- Character vector of the form ' $y \sim f(b_1, b_2, \dots, b_j, x_1, x_2, \dots, x_k)$ ', where  $f$  represents a scalar function of the scalar coefficient variables  $b_1, \dots, b_j$  and the scalar data variables  $x_1, \dots, x_k$ .

**beta0 — Coefficients**

numeric vector

Coefficients for the nonlinear model, specified as a numeric vector. `NonLinearModel` starts its search for optimal coefficients from `beta0`.

Data Types: `single` | `double`

## Properties

### **CoefficientCovariance** — Covariance matrix of coefficient estimates

numeric matrix

Covariance matrix of coefficient estimates, stored as a  $p$ -by- $p$  matrix of numeric values.  $p$  is the number of coefficients in the fitted model.

### **CoefficientNames** — Coefficient names

cell array of character vectors

Coefficient names, stored as a cell array of character vectors containing a label for each coefficient.

### **Coefficients** — Coefficient values

table

Coefficient values, stored as a table. `Coefficients` has one row for each coefficient and the following columns:

- `Estimate` — Estimated coefficient value
- `SE` — Standard error of the estimate
- `tStat` —  $t$  statistic for a test that the coefficient is zero
- `pValue` —  $p$ -value for the  $t$  statistic

To obtain any of these columns as a vector, index into the property using dot notation. For example, in `mdl` the estimated coefficient vector is

```
beta = mdl.Coefficients.Estimate
```

Use `coefTest` to perform other tests on the coefficients.

### **Diagnostics** — Diagnostic information

table

Diagnostic information for the model, stored as a table. Diagnostics can help identify outliers and influential observations. `Diagnostics` contains the following fields.

Field	Meaning	Utility
Leverage	Diagonal elements of <code>HatMatrix</code>	Leverage indicates to what extent the predicted value for an observation is determined by the observed value for that observation. A value close to 1 indicates that the prediction is largely determined by that observation, with little contribution from the other observations. A value close to 0 indicates the fit is largely determined by the other observations. For a model with $P$ coefficients and $N$ observations, the average value of Leverage is $P/N$ . An observation with Leverage larger than $2 * P/N$ can be regarded as having high leverage.
CooksDistance	Cook's measure of scaled change in fitted values	CooksDistance is a measure of scaled change in fitted values. An observation with CooksDistance larger than three times the mean Cook's distance can be an outlier.
HatMatrix	Projection matrix to compute fitted from observed responses	HatMatrix is an $N$ -by- $N$ matrix such that $Fitted = HatMatrix * Y$ , where $Y$ is the response vector and Fitted is the vector of fitted response values.

#### **DFE** — Degrees of freedom for error

positive integer value

Degrees of freedom for error (residuals), equal to the number of observations minus the number of estimated coefficients, stored as a positive integer value.

#### **Fitted** — Fitted response values based on input data

numeric vector

Fitted (predicted) values based on the input data, stored as a numeric vector. `fitnlm` attempts to make Fitted as close as possible to the response data.

#### **Formula** — Model information

`LinearFormula` object | `NonLinearFormula` object

Model information, stored as a `LinearFormula` object or `NonLinearFormula` object. If you fit a linear or generalized linear regression model, then `Formula` is a `LinearFormula` object. If you fit a nonlinear regression model, then `Formula` is a `NonLinearFormula` object.

**Iterative — Information about fitting process**

structure

Information about the fitting process, stored as a structure with the following fields:

- `InitialCoefs` — Initial coefficient values (the `beta0` vector)
- `IterOpts` — Options included in the `Options` name-value pair argument for `fitnlm`.

**LogLikelihood — Log likelihood**

numeric value

Log likelihood of the model distribution at the response values, stored as a numeric value. The mean is fitted from the model, and other parameters are estimated as part of the model fit.

**ModelCriterion — Criterion for model comparison**

structure

Criterion for model comparison, stored as a structure with the following fields:

- `AIC` — Akaike information criterion
- `AICc` — Akaike information criterion corrected for sample size
- `BIC` — Bayesian information criterion
- `CAIC` — Consistent Akaike information criterion

To obtain any of these values as a scalar, index into the property using dot notation. For example, in a model `mdl`, the `AIC` value `aic` is:

```
aic = mdl.ModelCriterion.AIC
```

**MSE — Mean squared error**

numeric value

Mean squared error, stored as a numeric value. The mean squared error is an estimate of the variance of the error term in the model.



**NumCoefficients — Number of model coefficients**

positive integer

Number of coefficients in the fitted model, stored as a positive integer.

`NumCoefficients` is the same as `NumEstimatedCoefficients` for `NonLinearModel` objects. `NumEstimatedCoefficients` is equal to the degrees of freedom for regression.

**NumEstimatedCoefficients — Number of estimated coefficients**

positive integer

Number of estimated coefficients in the fitted model, stored as a positive integer.

`NumEstimatedCoefficients` is the same as `NumCoefficients` for `NonLinearModel` objects. `NumEstimatedCoefficients` is equal to the degrees of freedom for regression.

**NumPredictors — Number of predictor variables**

positive integer

Number of predictor variables used to fit the model, stored as a positive integer.

**NumVariables — Number of variables**

positive integer

Number of variables in the input data, stored as a positive integer. `NumVariables` is the number of variables in the original table or dataset, or the total number of columns in the predictor matrix and response vector when the fit is based on those arrays. It includes variables, if any, that are not used as predictors or as the response.

**ObservationInfo — Observation information**

table

Observation information, stored as a  $n$ -by-4 table, where  $n$  is equal to the number of rows of input data. The four columns of `ObservationInfo` contain the following:

Field	Description
Weights	Observation weights. Default is all 1.
Excluded	Logical value, 1 indicates an observation that you excluded from the fit with the <code>Exclude</code> name-value pair.
Missing	Logical value, 1 indicates a missing value in the input. Missing values are not used in the fit.

Field	Description
Subset	Logical value, 1 indicates the observation is not excluded or missing, so is used in the fit.

**ObservationNames — Observation names**

cell array of character vectors

Observation names, stored as a cell array of character vectors containing the names of the observations used in the fit.

- If the fit is based on a table or dataset containing observation names, `ObservationNames` uses those names.
- Otherwise, `ObservationNames` is an empty cell array

**PredictorNames — Names of predictors used to fit the model**

cell array of character vectors

Names of predictors used to fit the model, stored as a cell array of character vectors.

**Residuals — Residuals for fitted model**

table

Residuals for fitted model, stored as a table that contains one row for each observation and the following columns.

Field	Description
Raw	Observed minus fitted values.
Pearson	Raw residuals divided by RMSE.
Standardized	Raw residuals divided by their estimated standard deviation.
Studentized	Residual divided by an independent estimate of the residual standard deviation. The residual for observation $i$ is divided by an estimate of the error standard deviation based on all observations except for observation $i$ .

To obtain any of these columns as a vector, index into the property using dot notation. For example, in a model `mdl`, the ordinary raw residual vector `r` is:

```
r = mdl.Residuals.Raw
```

Rows not used in the fit because of missing values (in `ObservationInfo.Missing`) contain NaN values.

Rows not used in the fit because of excluded values (in `ObservationInfo.Excluded`) contain NaN values, with the following exceptions:

- `raw` contains the difference between the observed and predicted values.
- `standardized` is the residual, standardized in the usual way.
- `studentized` matches the standardized values because this residual is not used in the estimate of the residual standard deviation.

### **ResponseName** — Response variable name

character vector

Response variable name, stored as a character vector.

### **RMSE** — Root mean squared error

numeric value

Root mean squared error, stored as a numeric value. The root mean squared error is an estimate of the standard deviation of the error term in the model.

### **Robust** — Robust fit information

structure

Robust fit information, stored as a structure with the following fields:

Field	Description
<code>WgtFun</code>	Robust weighting function, such as 'bisquare' (see <code>robustfit</code> )
<code>Tune</code>	Value specified for tuning parameter (can be [])
<code>Weights</code>	Vector of weights used in final iteration of robust fit

This structure is empty unless `fitnlm` constructed the model using robust regression.

### **Rsquared** — R-squared value for the model

structure

R-squared value for the model, stored as a structure.

For a linear or nonlinear model, `Rsquared` is a structure with two fields:

- Ordinary — Ordinary (unadjusted) R-squared
- Adjusted — R-squared adjusted for the number of coefficients

For a generalized linear model, `Rsquared` is a structure with five fields:

- Ordinary — Ordinary (unadjusted) R-squared
- Adjusted — R-squared adjusted for the number of coefficients
- LLR — Log-likelihood ratio
- Deviance — Deviance
- `AdjGeneralized` — Adjusted generalized R-squared

The R-squared value is the proportion of total sum of squares explained by the model. The ordinary R-squared value relates to the `SSR` and `SST` properties:

$$\text{Rsquared} = \text{SSR}/\text{SST} = 1 - \text{SSE}/\text{SST}.$$

To obtain any of these values as a scalar, index into the property using dot notation. For example, the adjusted R-squared value in `mdl` is

```
r2 = mdl.Rsquared.Adjusted
```

### **SSE — Sum of squared errors**

numeric value

Sum of squared errors (residuals), stored as a numeric value.

The Pythagorean theorem implies

$$\text{SST} = \text{SSE} + \text{SSR}.$$

### **SSR — Regression sum of squares**

numeric value

Regression sum of squares, stored as a numeric value. The regression sum of squares is equal to the sum of squared deviations of the fitted values from their mean.

The Pythagorean theorem implies

$$\text{SST} = \text{SSE} + \text{SSR}.$$

**sst** — Total sum of squares

numeric value

Total sum of squares, stored as a numeric value. The total sum of squares is equal to the sum of squared deviations of  $y$  from  $\text{mean}(y)$ .

The Pythagorean theorem implies

$$\text{SST} = \text{SSE} + \text{SSR}.$$

**VariableInfo** — Information about input variables

table

Information about input variables contained in `Variables`, stored as a table with one row for each model term and the following columns.

Field	Description
Class	Character vector giving variable class, such as 'double'
Range	Cell array giving variable range: <ul style="list-style-type: none"> <li>• Continuous variable — Two-element vector <math>[min, max]</math>, the minimum and maximum values</li> <li>• Categorical variable — Cell array of distinct variable values</li> </ul>
InModel	Logical vector, where <code>true</code> indicates the variable is in the model
IsCategorical	Logical vector, where <code>true</code> indicates a categorical variable

**VariableNames** — Names of variables used in fit

cell array of character vectors

Names of variables used in fit, stored as a cell array of character vectors.

- If the fit is based on a table or dataset, this property provides the names of the variables in that table or dataset.
- If the fit is based on a predictor matrix and response vector, `VariableNames` is the values in the `VarNames` name-value pair of the fitting method.
- Otherwise the variables have the default fitting names.

**Variables** — Data used to fit the model

table

Data used to fit the model, stored as a table. `Variables` contains both observation and response values. If the fit is based on a table or dataset array, `Variables` contains all of the data from that table or dataset array. Otherwise, `Variables` is a table created from the input data matrix  $X$  and response vector  $y$ .

## Methods

<code>coefCI</code>	Confidence intervals of coefficient estimates of nonlinear regression model
<code>coefTest</code>	Linear hypothesis test on nonlinear regression model coefficients
<code>disp</code>	Display nonlinear regression model
<code>feval</code>	Evaluate nonlinear regression model prediction
<code>fit</code>	Fit nonlinear regression model
<code>plotDiagnostics</code>	Plot diagnostics of nonlinear regression model
<code>plotResiduals</code>	Plot residuals of nonlinear regression model
<code>plotSlice</code>	Plot of slices through fitted nonlinear regression surface
<code>predict</code>	Predict response of nonlinear regression model
<code>random</code>	Simulate responses for nonlinear regression model

## Copy Semantics

Value. To learn how value classes affect copy operations, see [Copying Objects \(MATLAB\)](#).

## Examples

### Fit a Nonlinear Regression Model

Fit a nonlinear regression model for auto mileage based on the `carbig` data. Predict the mileage of an average car.

Load the sample data. Create a matrix  $X$  containing the measurements for the horsepower (Horsepower) and weight (Weight) of each car. Create a vector  $y$  containing the response values in miles per gallon (MPG).

```
load carbig
X = [Horsepower,Weight];
y = MPG;
```

Fit a nonlinear regression model.

```
modelfun = @(b,x)b(1) + b(2)*x(:,1).^b(3) + ...
    b(4)*x(:,2).^b(5);
beta0 = [-50 500 -1 500 -1];
mdl = fitnlm(X,y,modelfun,beta0)
```

```
mdl =
```

```
Nonlinear regression model:
    y ~ b1 + b2*x1^b3 + b4*x2^b5
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
b1	-49.383	119.97	-0.41164	0.68083
b2	376.43	567.05	0.66384	0.50719
b3	-0.78193	0.47168	-1.6578	0.098177
b4	422.37	776.02	0.54428	0.58656
b5	-0.24127	0.48325	-0.49926	0.61788

```
Number of observations: 392, Error degrees of freedom: 387
Root Mean Squared Error: 3.96
R-Squared: 0.745, Adjusted R-Squared 0.743
F-statistic vs. constant model: 283, p-value = 1.79e-113
```

Find the predicted mileage of an average auto. Since the sample data contains some missing (NaN) observations, compute the mean using nanmean.

```
Xnew = nanmean(X)
MPGnew = predict(mdl,Xnew)
```

```
Xnew =  
  
1.0e+03 *  
0.1051    2.9794
```

```
MPGnew =  
  
21.8073
```

- “Nonlinear Regression Workflow” on page 13-15

## Definitions

### Hat Matrix

The hat matrix  $H$  is defined in terms of the data matrix  $X$  and the Jacobian matrix  $J$ :

$$J_{i,j} = \left. \frac{\partial f}{\partial \beta_j} \right|_{x_i, \beta}$$

Here  $f$  is the nonlinear model function, and  $\beta$  is the vector of model coefficients.

The Hat Matrix  $H$  is

$$H = J(J^T J)^{-1} J^T.$$

The diagonal elements  $H_{ii}$  satisfy

$$0 \leq h_{ii} \leq 1$$

$$\sum_{i=1}^n h_{ii} = p,$$

where  $n$  is the number of observations (rows of  $X$ ), and  $p$  is the number of coefficients in the regression model.



## Leverage

The leverage of observation  $i$  is the value of the  $i$ th diagonal term,  $h_{ii}$ , of the hat matrix  $H$ . Because the sum of the leverage values is  $p$  (the number of coefficients in the regression model), an observation  $i$  can be considered to be an outlier if its leverage substantially exceeds  $p/n$ , where  $n$  is the number of observations.

## Cook's Distance

The Cook's distance  $D_i$  of observation  $i$  is

$$D_i = \frac{\sum_{j=1}^n (\hat{y}_j - \hat{y}_{j(i)})^2}{p \text{MSE}},$$

where

- $\hat{y}_j$  is the  $j$ th fitted response value.
- $\hat{y}_{j(i)}$  is the  $j$ th fitted response value, where the fit does not include observation  $i$ .
- $\text{MSE}$  is the mean squared error.
- $p$  is the number of coefficients in the regression model.

Cook's distance is algebraically equivalent to the following expression:

$$D_i = \frac{r_i^2}{p \text{MSE}} \left( \frac{h_{ii}}{(1-h_{ii})^2} \right),$$

where  $e_i$  is the  $i$ th residual.

## See Also

GeneralizedLinearModel | LinearModel | fitnlm | nlinfit |  
plotPartialDependence | predict

## Topics

“Nonlinear Regression Workflow” on page 13-15

“Nonlinear Regression” on page 13-2

# normcdf

Normal cumulative distribution function

## Syntax

```
p = normcdf(x)
p = normcdf(x,mu,sigma)
[p,plo,pup] = normcdf(x,mu,sigma,pcov,alpha)
[p,plo,pup] = normcdf( ____, 'upper')
```

## Description

`p = normcdf(x)` returns the standard normal cdf at each value in `x`. The standard normal distribution has parameters `mu = 0` and `sigma = 1`. `x` can be a vector, matrix, or multidimensional array.

`p = normcdf(x,mu,sigma)` returns the normal cdf at each value in `x` using the specified values for the mean `mu` and standard deviation `sigma`. `x`, `mu`, and `sigma` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array with the same dimensions as the other inputs. The parameters in `sigma` must be positive.

`[p,plo,pup] = normcdf(x,mu,sigma,pcov,alpha)` returns confidence bounds for `p` when the input parameters `mu` and `sigma` are estimates. `pcov` is the covariance matrix of the estimated parameters. `alpha` specifies  $100(1 - \alpha)\%$  confidence bounds. The default value of `alpha` is 0.05. `plo` and `pup` are arrays of the same size as `p` containing the lower and upper confidence bounds.

`[p,plo,pup] = normcdf( ____, 'upper')` returns the complement of the normal cdf at each value in `x`, using an algorithm that more accurately computes the extreme upper tail probabilities. You can use 'upper' with any of the previous syntaxes.

The function `normcdf` computes confidence bounds for `p` using a normal approximation to the distribution of the estimate

$$\frac{X - \hat{\mu}}{\hat{\sigma}}$$

and then transforming those bounds to the scale of the output `p`. The computed bounds give approximately the desired confidence level when you estimate `mu`, `sigma`, and `pcov` from large samples, but in smaller samples other methods of computing the confidence bounds might be more accurate.

The normal cdf is

$$p = F(x | \mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} \int_{-\infty}^x e^{-\frac{(t-\mu)^2}{2\sigma^2}} dt$$

The result,  $p$ , is the probability that a single observation from a normal distribution with parameters  $\mu$  and  $\sigma$  will fall in the interval  $(-\infty, x]$ .

The *standard normal* distribution has  $\mu = 0$  and  $\sigma = 1$ .

## Examples

### Compute Normal Distribution cdf

What is the probability that an observation from a standard normal distribution will fall on the interval `[-1 1]`?

```
p = normcdf([-1 1]);  
p(2) - p(1)
```

```
ans =
```

```
0.6827
```

More generally, about 68% of the observations from a normal distribution fall within one standard deviation,  $\sigma$ , of the mean,  $\mu$ .

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`cdf` | `normfit` | `norminv` | `normlike` | `normpdf` | `normrnd` | `normstat`

### Topics

“Normal Distribution” on page B-134

Introduced before R2006a

## normfit

Normal parameter estimates

### Syntax

```
[muhat, sigmahat] = normfit(data)
[muhat, sigmahat, muc1, sigmac1] = normfit(data)
[muhat, sigmahat, muc1, sigmac1] = normfit(data, alpha)
[...] = normfit(data, alpha, censoring)
[...] = normfit(data, alpha, censoring, freq)
[...] = normfit(data, alpha, censoring, freq, options)
```

### Description

`[muhat, sigmahat] = normfit(data)` returns an estimate of the mean  $\mu$  in `muhat`, and an estimate of the standard deviation  $\sigma$  in `sigmahat`, of the normal distribution given the data in `data`.

`[muhat, sigmahat, muc1, sigmac1] = normfit(data)` returns 95% confidence intervals for the parameter estimates on the mean and standard deviation in the arrays `muc1` and `sigmac1`, respectively. The first row of `muc1` contains the lower bounds of the confidence intervals for  $\mu$  the second row contains the upper bounds. The first row of `sigmac1` contains the lower bounds of the confidence intervals for  $\sigma$ , and the second row contains the upper bounds.

`[muhat, sigmahat, muc1, sigmac1] = normfit(data, alpha)` returns  $100(1 - \text{alpha})\%$  confidence intervals for the parameter estimates, where `alpha` is a value in the range `[0 1]` specifying the width of the confidence intervals. By default, `alpha` is `0.05`, which corresponds to 95% confidence intervals.

`[...] = normfit(data, alpha, censoring)` accepts a Boolean vector, `censoring`, of the same size as `data`, which is 1 for observations that are right-censored and 0 for observations that are observed exactly. `data` must be a vector in order to pass in the argument `censoring`.

[...] = normfit(data,alpha,censoring,freq) accepts a frequency vector, freq, of the same size as data. Typically, freq contains integer frequencies for the corresponding elements in data, but can contain any nonnegative values. Pass in [] for alpha, censoring, or freq to use their default values.

[...] = normfit(data,alpha,censoring,freq,options) accepts a structure, options, that specifies control parameters for the iterative algorithm the function uses to compute maximum likelihood estimates when there is censoring. The normal fit function accepts an options structure which you can create using the function statset. Enter statset('normfit') to see the names and default values of the parameters that normfit accepts in the options structure. See the reference page for statset for more information about these options.

---

**Note** With no censoring, normfit computes muhat using the sample mean and sigmahat using the square root of the unbiased estimator of the variance. With censoring, both muhat and sigmahat are the maximum likelihood estimates.

---

## Examples

In this example the data is a two-column random normal matrix. Both columns have  $\mu = 10$  and  $\sigma = 2$ . Note that the confidence intervals below contain the "true values."

```
data = normrnd(10,2,100,2);
[mu,sigma,muci,sigmaci] = normfit(data)
mu =
    10.1455    10.0527
sigma =
    1.9072    2.1256
muci =
    9.7652    9.6288
    10.5258    10.4766
sigmaci =
    1.6745    1.8663
    2.2155    2.4693
```

## See Also

mle | normcdf | norminv | normlike | normpdf | normrnd | normstat

## **Topics**

“Normal Distribution” on page B-134

**Introduced before R2006a**



# norminv

Normal inverse cumulative distribution function

## Syntax

```
X = norminv(P)
X = norminv(P,mu,sigma)
[X,XLO,XUP] = norminv(P,mu,sigma,pcov,alpha)
```

## Description

`X = norminv(P)` computes the inverse of the standard normal cdf. The standard normal distribution has parameters  $\mu = 0$  and  $\sigma = 1$ . `P` can be a vector, matrix, or multidimensional array, and the values in `P` must lie in the interval  $[0\ 1]$ .

`X = norminv(P,mu,sigma)` computes the inverse of the normal cdf using the corresponding mean  $\mu$  and standard deviation  $\sigma$  at the corresponding probabilities in `P`. `P`,  $\mu$ , and  $\sigma$  can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array with the same dimensions as the other inputs. The parameters in  $\sigma$  must be positive, and the values in `P` must lie in the interval  $[0\ 1]$ .

`[X,XLO,XUP] = norminv(P,mu,sigma,pcov,alpha)` produces confidence bounds for `X` when the input parameters  $\mu$  and  $\sigma$  are estimates. `pcov` is the covariance matrix of the estimated parameters. `alpha` specifies  $100(1 - \alpha)\%$  confidence bounds. The default value of `alpha` is `0.05`. `XLO` and `XUP` are arrays of the same size as `X` containing the lower and upper confidence bounds.

The function `norminv` computes confidence bounds for `P` using a normal approximation to the distribution of the estimate

$$\hat{\mu} + \hat{\sigma}q$$

where  $q$  is the  $P$ th quantile from a normal distribution with mean 0 and standard deviation 1. The computed bounds give approximately the desired confidence level when

you estimate `mu`, `sigma`, and `pcov` from large samples, but in smaller samples other methods of computing the confidence bounds may be more accurate.

The normal inverse function is defined in terms of the normal cdf as

$$x = F^{-1}(p | \mu, \sigma) = \{x : F(x | \mu, \sigma) = p\}$$

where

$$p = F(x | \mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} \int_{-\infty}^x e^{-\frac{(t-\mu)^2}{2\sigma^2}} dt$$

The result, `x`, is the solution of the integral equation above where you supply the desired probability, `p`.

## Examples

Find an interval that contains 95% of the values from a standard normal distribution.

```
x = norminv([0.025 0.975])
x =
   -1.9600    1.9600
```

Note that the interval `x` is not the only such interval, but it is the shortest.

```
x1 = norminv([0.01 0.96])
x1 =
   -2.3263    1.7507
```

The interval `x1` also contains 95% of the probability, but it is longer than `x`.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

## See Also

`icdf` | `normcdf` | `normfit` | `normlike` | `normpdf` | `normrnd` | `normstat`

## Topics

“Normal Distribution” on page B-134

**Introduced before R2006a**

## normlike

Normal negative log-likelihood

### Syntax

```
nlogL = normlike(params, data)
[nlogL, AVAR] = normlike(params, data)
[...] = normlike(param, data, censoring)
[...] = normlike(param, data, censoring, freq)
```

### Description

`nlogL = normlike(params, data)` returns the negative of the normal log-likelihood function. `params(1)` is the mean, `mu`, and `params(2)` is the standard deviation, `sigma`.

`[nlogL, AVAR] = normlike(params, data)` also returns the inverse of Fisher's information matrix, `AVAR`. If the input parameter values in `params` are the maximum likelihood estimates, the diagonal elements of `AVAR` are their asymptotic variances. `AVAR` is based on the observed Fisher's information, not the expected information.

`[...] = normlike(param, data, censoring)` accepts a Boolean vector, `censoring`, of the same size as `data`, which is 1 for observations that are right-censored and 0 for observations that are observed exactly.

`[...] = normlike(param, data, censoring, freq)` accepts a frequency vector, `freq`, of the same size as `data`. The vector `freq` typically contains integer frequencies for the corresponding elements in `data`, but can contain any nonnegative values. Pass in `[]` for `censoring` to use its default value.

`normlike` is a utility function for maximum likelihood estimation.

### See Also

`normcdf` | `normfit` | `norminv` | `normpdf` | `normrnd` | `normstat`

## Topics

“Normal Distribution” on page B-134

Introduced before R2006a

## normpdf

Normal probability density function

### Syntax

```
Y = normpdf(X, mu, sigma)
```

```
Y = normpdf(X)
```

```
Y = normpdf(X, mu)
```

### Description

`Y = normpdf(X, mu, sigma)` computes the pdf at each of the values in `X` using the normal distribution with mean `mu` and standard deviation `sigma`. `X`, `mu`, and `sigma` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array with the same dimensions as the other inputs. The parameters in `sigma` must be positive.

The normal pdf is

$$y = f(x | \mu, \sigma) = \frac{1}{\sigma \sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

The *likelihood function* is the pdf viewed as a function of the parameters. Maximum likelihood estimators (MLEs) are the values of the parameters that maximize the likelihood function for a fixed value of `x`.

The *standard normal* distribution has  $\mu = 0$  and  $\sigma = 1$ .

If `x` is standard normal, then `xσ + μ` is also normal with mean  $\mu$  and standard deviation  $\sigma$ . Conversely, if `y` is normal with mean  $\mu$  and standard deviation  $\sigma$ , then `x = (y - μ) / σ` is standard normal.

`Y = normpdf(X)` uses the standard normal distribution (`mu = 0, sigma = 1`).

`Y = normpdf(X, mu)` uses the normal distribution with unit standard deviation (`sigma = 1`).

## Examples

```
mu = [0:0.1:2];  
[y i] = max(normpdf(1.5,mu,1));  
MLE = mu(i)  
MLE =  
    1.5000
```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

[mvnpdf](#) | [normcdf](#) | [normfit](#) | [norminv](#) | [normlike](#) | [normrnd](#) | [normstat](#) | [pdf](#)

### Topics

“Normal Distribution” on page B-134

Introduced before R2006a

## normplot

Normal probability plot

### Syntax

```
normplot(x)  
h = normplot(x)
```

### Description

`normplot(x)` displays a normal probability plot of the data contained in `x`. Use a normal probability plot to assess visually whether the sample data in `x` comes from a population with a normal distribution. If the sample data has a normal distribution, then the data appears along the reference line. Distributions other than normal can introduce curvature in the plot.

`h = normplot(x)` also returns a column vector of handles to the `Line` objects created by `normplot`.

### Examples

#### Generate a Normal Probability Plot

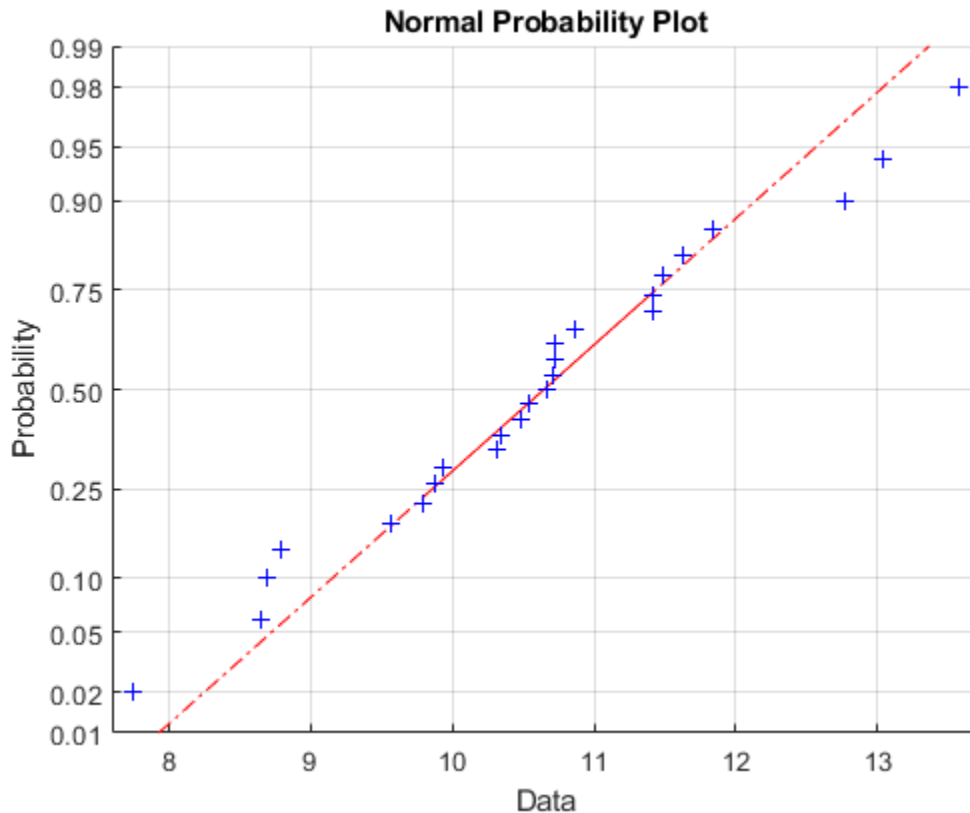
Generate random sample data from a normal distribution with  $\mu = 10$  and  $\sigma = 1$ .

```
rng default; % For reproducibility  
x = normrnd(10,1,25,1);
```

Create a normal probability plot of the sample data.

```
figure;  
normplot(x)
```





The plot indicates that the data follows a normal distribution.

### Adjust Normal Probability Plot Line Properties

Create a 50-by-2 matrix containing 50 random numbers from each of two different distributions: A standard normal distribution in column 1, and a set of Pearson random numbers with mu equal to 0, sigma equal to 1, skewness equal to 0.5, and kurtosis equal to 3 (a "right-skewed" distribution) in column 2.

```
rng default % For reproducibility
x = [normrnd(0,1,[50,1]) pearsrnd(0,1,0.5,3,[50,1])];
```

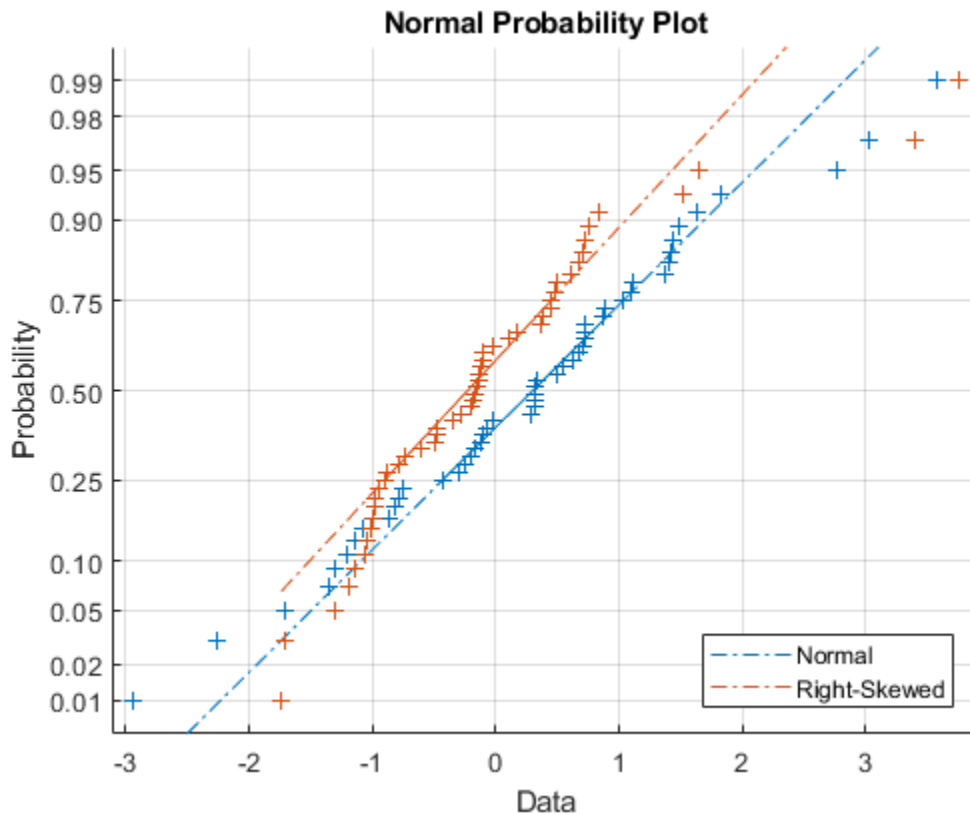
Create a normal probability plot for both samples on the same figure. Return the plot line graphic handles.

```
figure
h = normplot(x)
legend({'Normal', 'Right-Skewed'}, 'Location', 'southeast')
```

```
h =
```

```
6x1 Line array:
```

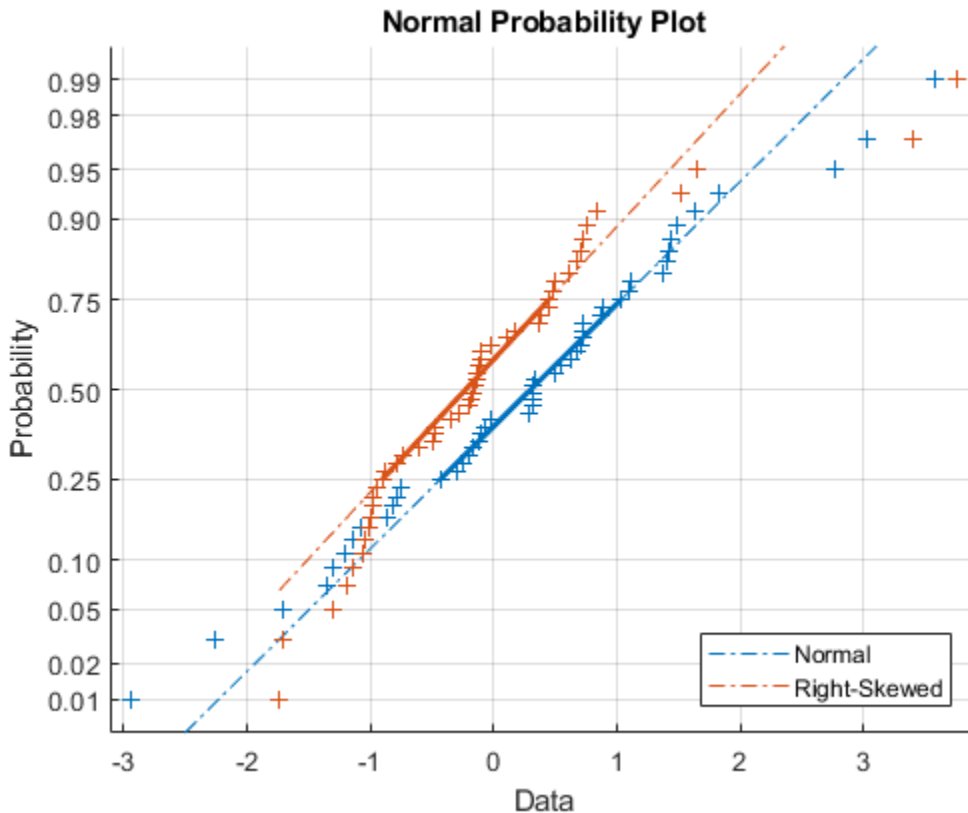
```
Line
Line
Line
Line
Line
Line
```



The handles h(1) and h(2) correspond to the data points for the normal and skewed distributions, respectively. The handles h(3) and h(4) correspond to the second and third quartile line fit to the sample data. The handles h(5) and h(6) correspond to the extrapolated line that extends to the minimum and maximum of each set of sample data.

To illustrate, increase the line width of the second and third quartile line for the normally distributed data sample (represented by h(3)) to 2.

```
h(3).LineWidth = 2;
h(4).LineWidth = 2;
```



### Assess Normality Using a Normal Probability Plot

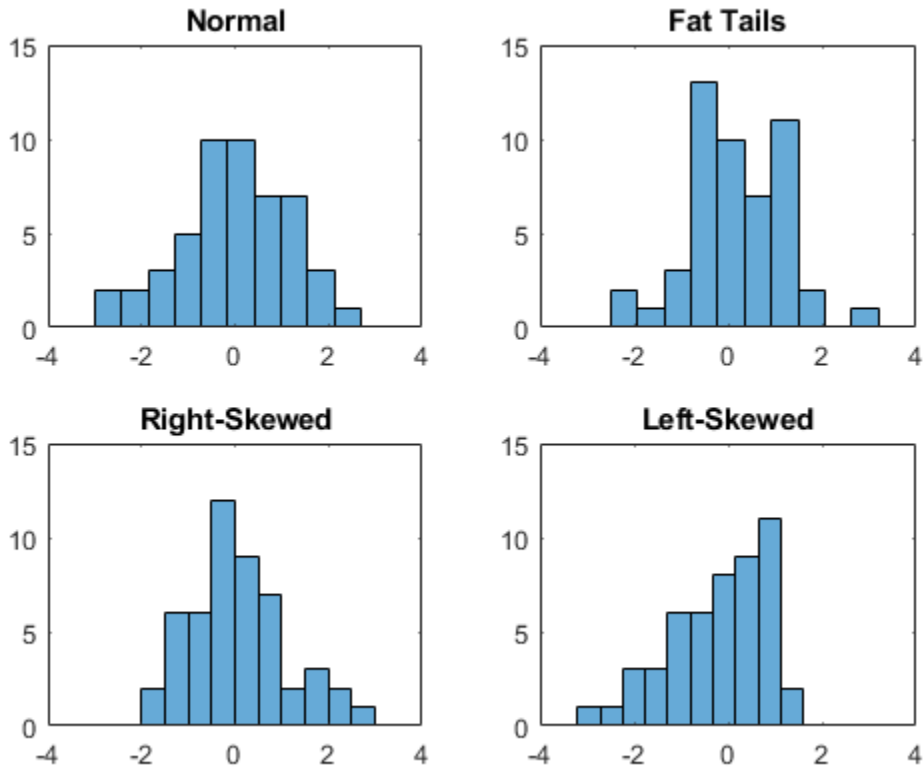
Generate 50 random numbers from each of four different distributions: A standard normal distribution; a Student's-t distribution with five degrees of freedom (a "fat-tailed" distribution); a set of Pearson random numbers with  $\mu$  equal to 0,  $\sigma$  equal to 1, skewness equal to 0.5, and kurtosis equal to 3 (a "right-skewed" distribution); and a set of Pearson random numbers with  $\mu$  equal to 0,  $\sigma$  equal to 1, skewness equal to -0.5, and kurtosis equal to 3 (a "left-skewed" distribution).

```
rng(11) % For reproducibility
x1 = normrnd(0,1,[50,1]);
x2 = trnd(5,[50,1]);
```

```
x3 = pearsrnd(0,1,0.5,3,[50,1]);  
x4 = pearsrnd(0,1,-0.5,3,[50,1]);
```

Plot four histograms on the same figure for a visual comparison of the pdf of each distribution.

```
figure  
subplot(2,2,1)  
histogram(x1,10)  
title('Normal')  
axis([-4,4,0,15])  
  
subplot(2,2,2)  
histogram(x2,10)  
title('Fat Tails')  
axis([-4,4,0,15])  
  
subplot(2,2,3)  
histogram(x3,10)  
title('Right-Skewed')  
axis([-4,4,0,15])  
  
subplot(2,2,4)  
histogram(x4,10)  
title('Left-Skewed')  
axis([-4,4,0,15])
```



The histograms show how each sample differs from the normal distribution.

Create a normal probability plot for each sample.

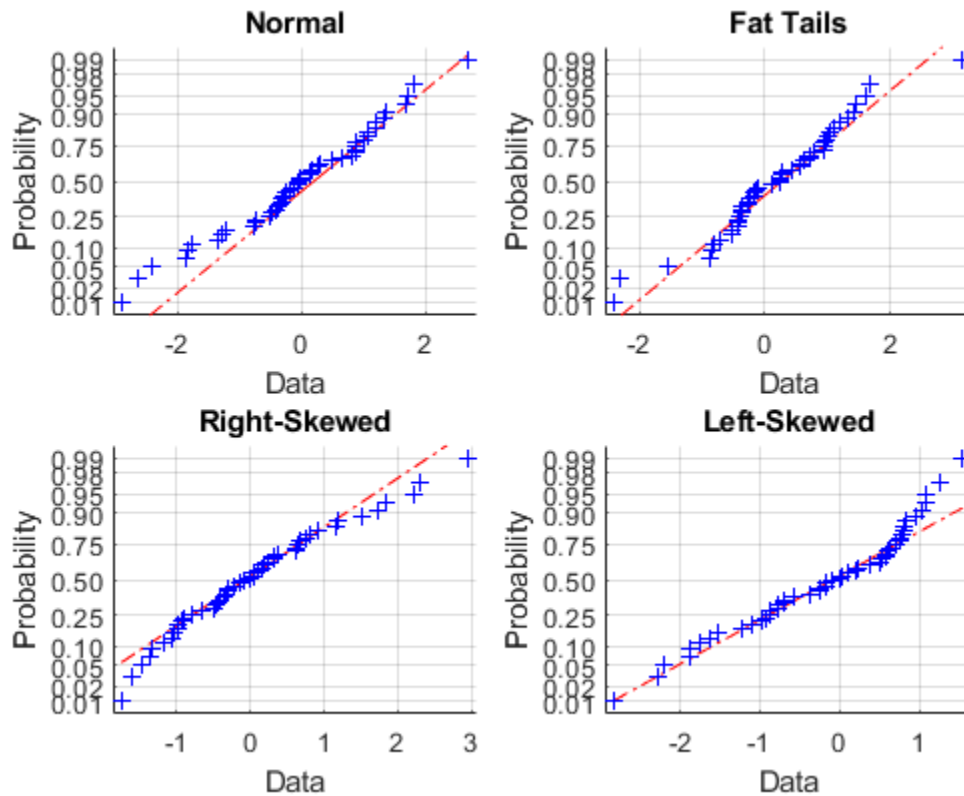
```
figure
subplot(2,2,1)
normplot(x1)
title('Normal')

subplot(2,2,2)
normplot(x2)
title('Fat Tails')

subplot(2,2,3)
```

```
normplot(x3)
title('Right-Skewed')
```

```
subplot(2,2,4)
normplot(x4)
title('Left-Skewed')
```



## Input Arguments

**x** — Sample data

numeric vector | numeric matrix

Sample data, specified as a numeric vector or numeric matrix. `normplot` displays each value in `x` using the symbol '+'. If `x` is a matrix, then `normplot` displays a separate line for each column of `x`.

Data Types: `single` | `double`

## Output Arguments

### **h** — Graphic handles for line objects

vector of `Line` graphic handles

Graphic handles for line objects, returned as a vector of `Line` graphic handles. Graphic handles are unique identifiers that you can use to query and modify the properties of a specific line on the plot. For each column of `x`, `normplot` returns three handles:

- The line representing the data points. `normplot` represents each data point in `x` using the plot symbol '+'.
- The line joining the first and third quartiles of each column of `x`, represented as a solid line.
- The extrapolation of the quartile line, extended to the minimum and maximum values of `x`, represented as a dashed line.

To view and set properties of line objects, use dot notation. For information on using dot notation, see “Access Property Values” (MATLAB). For information on the `Line` properties that you can set, see `Primitive Line`.

## Algorithms

`normplot` matches the quantiles of sample data to the quantiles of a normal distribution. The sample data is sorted and plotted on the x-axis. The y-axis represents the quantiles of the normal distribution, converted into probability values. Therefore, the y-axis scaling is not linear.

Where the x-axis value is the  $i$ th sorted value from a sample of size  $N$ , the y-axis value is the midpoint between evaluation points of the empirical cumulative distribution function



of the data. In the case of uncensored data, the midpoint is equal to  $\frac{(i-0.5)}{N}$ . When the data includes censored observations, use `probplot` instead.

`normplot` superimposes a reference line to assess the linearity of the plot. The line goes through the first and third quartiles of the data.

## See Also

`cdfplot` | `ecdf` | `probplot` | `wblplot`

## Topics

“Distribution Plots” on page 4-10

“Normal Distribution” on page B-134

**Introduced before R2006a**

## normrnd

Normal random numbers

### Syntax

```
R = normrnd(mu, sigma)
R = normrnd(mu, sigma, m, n, ...)
R = normrnd(mu, sigma, [m, n, ...])
```

### Description

`R = normrnd(mu, sigma)` generates random numbers from the normal distribution with mean parameter `mu` and standard deviation parameter `sigma`. `mu` and `sigma` can be vectors, matrices, or multidimensional arrays that have the same size, which is also the size of `R`. A scalar input for `mu` or `sigma` is expanded to a constant array with the same dimensions as the other input.

`R = normrnd(mu, sigma, m, n, ...)` or `R = normrnd(mu, sigma, [m, n, ...])` generates an `m`-by-`n`-by-... array. The `mu`, `sigma` parameters can each be scalars or arrays of the same size as `R`.

### Examples

```
n1 = normrnd(1:6, 1./(1:6))
n1 =
    2.1650    2.3134    3.0250    4.0879    4.8607    6.2827

n2 = normrnd(0, 1, [1 5])
n2 =
    0.0591    1.7971    0.2641    0.8717   -1.4462

n3 = normrnd([1 2 3; 4 5 6], 0.1, 2, 3)
n3 =
    0.9299    1.9361    2.9640
    4.1246    5.0577    5.9864
```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

The generated code can return a different sequence of numbers than MATLAB if either of the following is true:

- The output is nonscalar.
- An input parameter is invalid for the distribution.

### See Also

`lognrnd` | `mvnrnd` | `normcdf` | `normfit` | `norminv` | `normlike` | `normpdf` | `normstat` | `random`

### Topics

“Normal Distribution” on page B-134

**Introduced before R2006a**

## normspec

Normal density plot between specifications

### Syntax

```
normspec(specs)
normspec(specs,mu,sigma)
normspec(specs,mu,sigma,region)
p = normspec(...)
[p,h] = normspec(...)
```

### Description

`normspec(specs)` plots the standard normal density, shading the portion inside the specification limits given by the two-element vector `specs`. Set `specs(1)` to `-Inf` if there is no lower limit; set `specs(2)` to `Inf` if there is no upper limit.

`normspec(specs,mu,sigma)` shades the portion inside the specification limits of a normal density with parameters `mu` and `sigma`. The defaults are `mu = 0` and `sigma = 1`.

`normspec(specs,mu,sigma,region)` shades the `region` either `'inside'` or `'outside'` the specification limits. The default is `'inside'`.

`p = normspec(...)` also returns the probability, `p`, of the shaded area.

`[p,h] = normspec(...)` also returns a handle `h` to the line objects.

### Examples

#### Create a Normal Density Plot

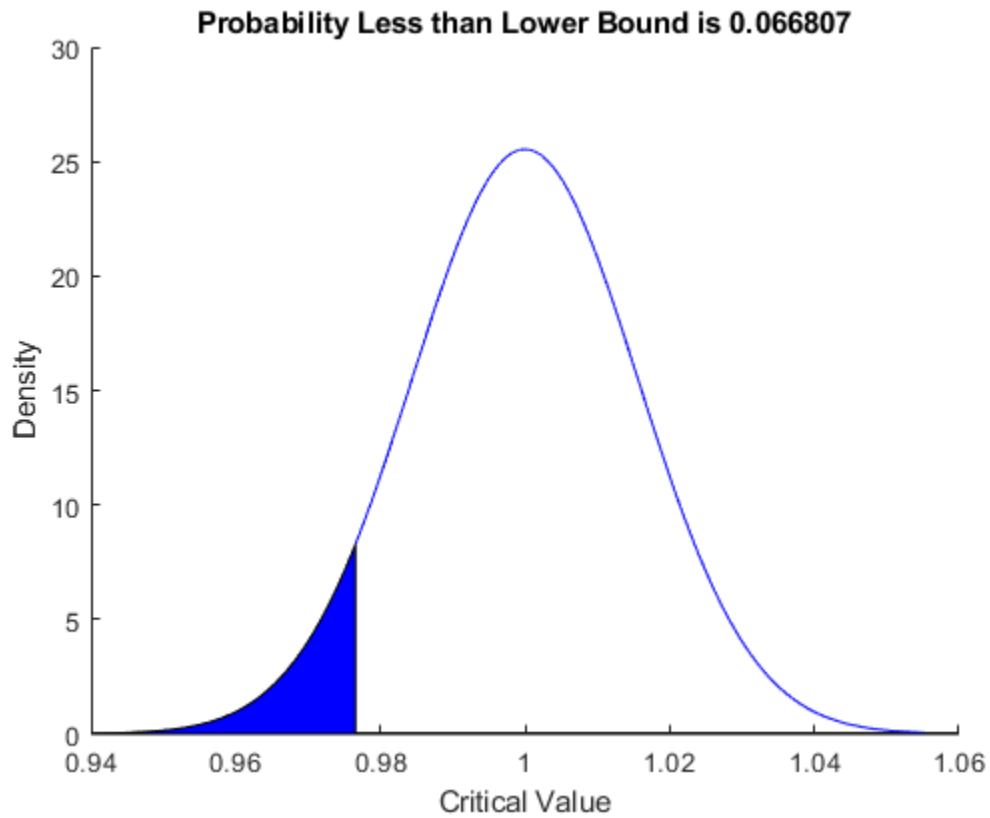
This example shows how to create a normal density plot.

A production process fills cans of paint. The average amount of paint in any can is 1 gallon, but variability in the process produces a standard deviation of 2 ounces ( $2/128$  gallons). What is the probability that the cans will be filled under specification by 3 or more ounces?

```
p = normspec([1-3/128, Inf], 1, 2/128, 'outside')
```

```
p =
```

```
0.0668
```



## See Also

`caplot` | `histfit`

## Topics

“Normal Distribution” on page B-134

Introduced before R2006a

# normstat

Normal mean and variance

## Syntax

```
[M,V] = normstat(mu,sigma)
```

## Description

`[M,V] = normstat(mu,sigma)` returns the mean and variance for the normal distribution using the corresponding mean `mu` and standard deviation `sigma`. `mu` and `sigma` can be vectors, matrices, or multidimensional arrays that all have the same size, which is also the size of `M` and `V`. A scalar input for `mu` or `sigma` is expanded to a constant array with the same dimensions as the other input.

The mean of the normal distribution with parameters  $\mu$  and  $\sigma$  is  $\mu$ , and the variance is  $\sigma^2$ .

## Examples

```
n = 1:5;
[m,v] = normstat(n'*n,n'*n)
m =
     1     2     3     4     5
     2     4     6     8    10
     3     6     9    12    15
     4     8    12    16    20
     5    10    15    20    25

v =
     1     4     9    16    25
     4    16    36    64   100
     9    36    81   144   225
    16    64   144   256   400
    25   100   225   400   625
```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`normcdf` | `normfit` | `norminv` | `normlike` | `normpdf` | `normrnd`

### Topics

“Normal Distribution” on page B-134

Introduced before R2006a



# nsegments

**Class:** `piecewisedistribution`

Number of segments

## Syntax

```
n = nsegments(obj)
```

## Description

`n = nsegments(obj)` returns the number of segments `n` in the piecewise distribution object `obj`.

## Examples

Fit Pareto tails to a  $t$  distribution at cumulative probabilities 0.1 and 0.9:

```
t = trnd(3,100,1);  
obj = paretotails(t,0.1,0.9);
```

```
n = nsegments(obj)  
n =  
    3
```

## See Also

`boundary` | `paretotails` | `segment`

## NTrees property

**Class:** TreeBagger

Number of decision trees in ensemble

## Description

The NTrees property is a scalar equal to the number of decision trees in the ensemble.

## See Also

Trees

## NumParams property

**Class:** ProbDistParametric

Read-only value specifying number of parameters of ProbDistParametric object

### Description

NumParams is a read-only property of the ProbDistParametric class. NumParams is a value specifying the number of parameters of a distribution represented by a ProbDistParametric object.

### Values

This value is an integer that counts both the specified parameters and parameters that are fit to the data. Use this information to view and compare the number of parameters supplied to create distributions.

## numel

**Class:** dataset

Number of elements in dataset array

---

**Note** The `dataset` data type might be removed in a future release. To work with heterogeneous data, use the MATLAB® `table` data type instead. See MATLAB `table` documentation for more information.

---

## Syntax

```
n = numel(A)
n = numel(A, varargin)
```

## Description

`n = numel(A)` returns 1. To find the number of elements, `n`, in the dataset array `A`, use `prod(size(A))` or `numel(A, ':', ':')`.

`n = numel(A, varargin)` returns the number of subscripted elements, `n`, in `A(index1, index2, ..., indexn)`, where `varargin` is a cell array whose elements are `index1, index2, ... indexn`.

## See Also

`length` | `size`

## numnodes

**Class:** classregtree

Number of nodes

---

**Note** classregtree will be removed in a future release. See fitctree, fitrtree, ClassificationTree, or RegressionTree instead.

---

## Syntax

```
n = numnodes(t)
```

## Description

n = numnodes(t) returns the number of nodes n in the tree t.

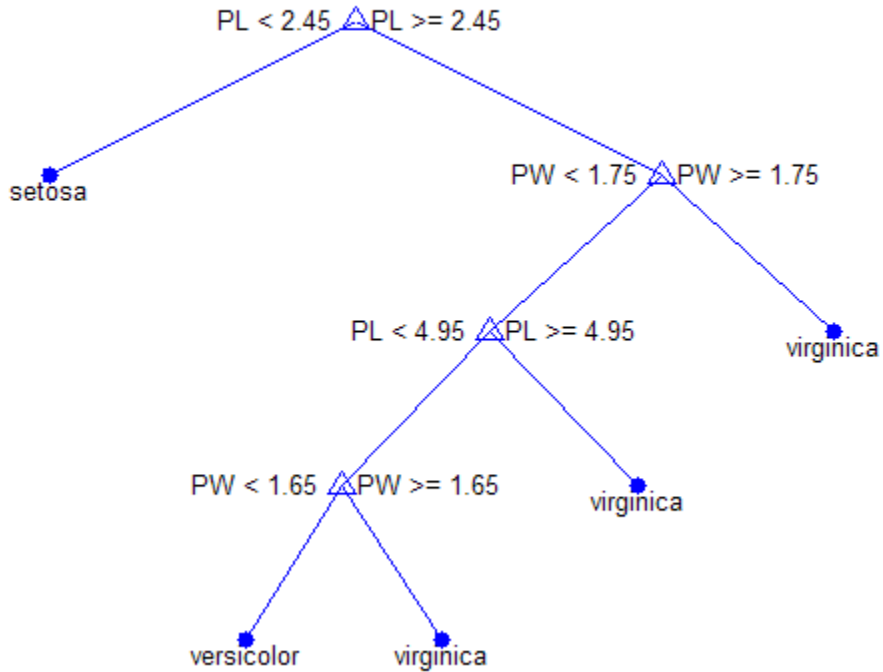
## Examples

Create a classification tree for Fisher's iris data:

```
load fisheriris;

t = classregtree(meas,species,...
                'names',{'SL' 'SW' 'PL' 'PW'})
t=
Decision tree for classification
1  if PL<2.45 then node 2 elseif PL>=2.45 then node 3 else setosa
2  class = setosa
3  if PW<1.75 then node 4 elseif PW>=1.75 then node 5 else versicolor
4  if PL<4.95 then node 6 elseif PL>=4.95 then node 7 else versicolor
5  class = virginica
6  if PW<1.65 then node 8 elseif PW>=1.65 then node 9 else versicolor
7  class = virginica
8  class = versicolor
9  class = virginica
view(t)
```

Click to display:  Magnification:  Pruning level:



`n = numnodes(t)`

`n =`  
9

## References

- [1] Breiman, L., J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Boca Raton, FL: CRC Press, 1984.

## See Also

`classregtree`

## NumTestSets property

**Class:** cvpartition

Number of test sets

### Description

Value is the number of folds in partitions of type 'kfold' and 'leaveout'.

Value is 1 in partitions of type 'holdout' and 'resubstitution'.



# NVarToSample property

**Class:** TreeBagger

Number of variables for random feature selection

## Description

The `NVarToSample` property specifies the number of predictor or feature variables to select at random for each decision split. By default, it is set to the square root of the total number of variables for classification and one third of the total number of variables for regression. Setting this argument to any valid value except 'all' invokes Breiman's "random forest" algorithm.

## See Also

`ClassificationTree` | `RegressionTree` | `TreeBagger` | `fitctree` | `fitrtree`

## ObsNames property

**Class:** dataset

Cell array of nonempty, distinct character vectors giving names of observations in data set

---

**Note** The `dataset` data type might be removed in a future release. To work with heterogeneous data, use the MATLAB® `table` data type instead. See MATLAB `table` documentation for more information.

---

## Description

A cell array of nonempty, distinct character vectors giving the names of the observations in the data set. This property may be empty, but if not empty, the number of character vectors must equal the number of observations.

# optimalleaforder

Optimal leaf ordering for hierarchical clustering

## Syntax

```
leafOrder = optimalleaforder(tree,D)  
leafOrder = optimalleaforder(tree,D,Name,Value)
```

## Description

`leafOrder = optimalleaforder(tree,D)` returns an optimal leaf ordering for the hierarchical binary cluster tree, `tree`, using the distances, `D`. An optimal leaf ordering of a binary tree maximizes the sum of the similarities between adjacent leaves by flipping tree branches without dividing the clusters.

`leafOrder = optimalleaforder(tree,D,Name,Value)` returns the optimal leaf ordering using one or more name-value pair arguments.

## Examples

### Plot Dendrogram With Optimal Leaf Order

Create a hierarchical binary cluster tree using `linkage`. Then, compare the dendrogram plot with the default ordering to a dendrogram with an optimal leaf ordering.

Generate sample data.

```
rng('default') % For reproducibility  
X = rand(10,2);
```

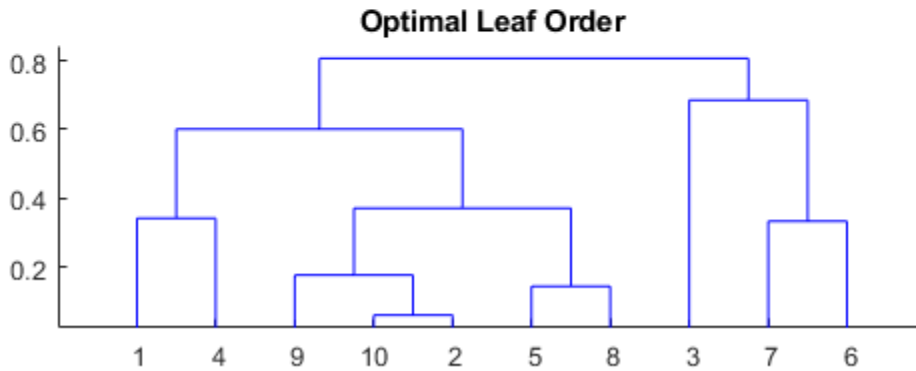
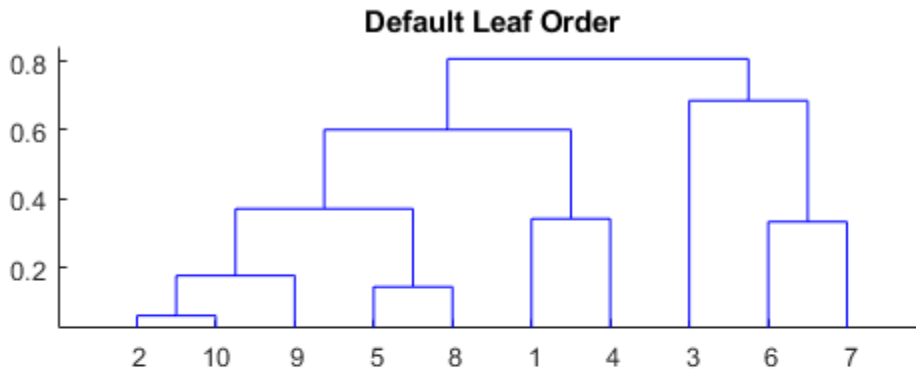
Create a distance vector and a hierarchical binary clustering tree. Use the distances and clustering tree to determine an optimal leaf order.

```
D = pdist(X);
tree = linkage(D, 'average');
leafOrder = optimalleaforder(tree,D);
```

Plot the dendrogram with the default ordering and the dendrogram with the optimal leaf ordering.

```
figure()
subplot(2,1,1)
dendrogram(tree)
title('Default Leaf Order')

subplot(2,1,2)
dendrogram(tree, 'reorder', leafOrder)
title('Optimal Leaf Order')
```



The order of the leaves in the bottom figure corresponds to the elements in `leafOrder`.

```
leafOrder
```

```
leafOrder =
```

```
1 4 9 10 2 5 8 3 7 6
```

### Optimal Leaf Order Using Inverse Distance Similarity

Generate sample data.

```
rng('default') % For reproducibility
X = rand(10,2);
```

Create a distance vector and a hierarchical binary clustering tree.

```
D = pdist(X);
tree = linkage(D, 'average');
```

Use the inverse distance similarity transformation to determine an optimal leaf order.

```
leafOrder = optimalleaforder(tree,D, 'Transformation', 'inverse')
```

```
leafOrder =
```

```
     1     4     9    10     2     5     8     3     7     6
```

## Input Arguments

### **tree** — Hierarchical binary cluster tree

matrix returned by `linkage`

Hierarchical binary cluster tree, specified as an  $(M - 1)$ -by-3 matrix that you generate using `linkage`, where  $M$  is the number of leaves.

### **D** — Distances

matrix | vector

Distances for determining similarities between leaves, specified as a matrix or vector of distances. For example, you can generate distances using `pdist`.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: 'Criteria', 'group', 'Transformation', 'inverse' specifies that the sum of similarities be maximized between every leaf and all other leaves in adjacent clusters, using an inverse similarity transformation.

### Criteria — Optimization criterion

'adjacent' (default) | 'group'

Optimization criterion for determining an optimal leaf ordering, specified as the comma-separated pair consisting of 'criteria' and one of these values:

'adjacent'	Maximize the sum of similarities between adjacent leaves.
'group'	Maximize the sum of similarities between every leaf and all other leaves in the adjacent clusters at the same level of the dendrogram.

Example: 'Criteria', 'group'

Data Types: char

### Transformation — Method for transforming distances to similarities

'linear' (default) | 'inverse' | function handle

Method for transforming distances to similarities, specified as the comma-separated pair consisting of 'Transformation' and one of 'linear', 'inverse', or a function handle.

Let  $d_{i,j}$  and  $Sim_{i,j}$  denote the distance and similarity between leaves  $i$  and  $j$ , respectively. The included similarity transformations are:

'linear'	$Sim_{i,j} = \max_{i,j} (d_{i,j}) - d_{i,j}$
'inverse'	$Sim_{i,j} = 1/d_{i,j}$

To use a custom transformation function, specify a handle to a function that accepts a matrix of distances,  $D$ , and returns a matrix of similarities,  $S$ . The function should be monotonic decreasing in the range of distance values.  $S$  must have the same size as  $D$ , with  $S(i, j)$  being the similarity computed based on  $D(i, j)$ .

Example: 'Transformation', @myTransform

Data Types: char | function\_handle

## Output Arguments

### `leafOrder` — Optimal leaf order

vector

Optimal leaf order, returned as a length- $M$  vector, where  $M$  is the number of leaves. `leafOrder` is a permutation of the vector  $1:M$ , giving an optimal leaf ordering based on the specified distances and similarity transformation.

## References

- [1] Bar-Joseph, Z., Gifford, D.K., and Jaakkola, T.S. (2001). Fast optimal leaf ordering for hierarchical clustering. *Bioinformatics* *17*, Suppl 1:S22–9. PMID: 11472989.

## See Also

`dendrogram` | `linkage` | `pdist`

Introduced in R2012b



# oobEdge

**Class:** ClassificationBaggedEnsemble

Out-of-bag classification edge

## Syntax

```
edge = oobEdge(ens)
edge = oobEdge(ens, Name, Value)
```

## Description

`edge = oobEdge(ens)` returns out-of-bag classification edge for `ens`.

`edge = oobEdge(ens, Name, Value)` computes classification edge with additional options specified by one or more `Name, Value` pair arguments. You can specify several name-value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

## Input Arguments

### **ens**

A classification bagged ensemble, constructed with `fitcensemble`.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

**learners**

Indices of weak learners in the ensemble ranging from 1 to `ens.NumTrained`. `oobEdge` uses only these learners for calculating loss.

**Default:** `1:NumTrained`

**mode**

Character vector representing the meaning of the output `L`:

- `'ensemble'` — `L` is a scalar value, the loss for the entire ensemble.
- `'individual'` — `L` is a vector with one element per trained learner.
- `'cumulative'` — `L` is a vector in which element `J` is obtained by using learners `1:J` from the input list of learners.

**Default:** `'ensemble'`

## Output Arguments

**edge**

Classification edge, a weighted average of the classification margin.

## Examples

**Estimate Out-of-Bag Edge**

Load Fisher's iris data set.

```
load fisheriris
```

Train an ensemble of 100 bagged classification trees using the entire data set.

```
rng(1) % For reproducibility
Mdl = fitcensemble(meas, species, 'Method', 'Bag');
```

Estimate the out-of-bag edge.

```
edge = oobEdge(Mdl)
```

```
edge =
```

```
    0.8638
```

## Definitions

### Edge

The edge is the weighted mean value of the classification margin. The weights are the class probabilities in `ens.Prior`.

### Margin

The classification margin is the difference between the classification score for the true class and maximal classification score for the false classes. Margin is a column vector with the same number of rows as in the matrix `ens.X`.

### Out of Bag

Bagging, which stands for “bootstrap aggregation”, is a type of ensemble learning. To bag a weak learner such as a decision tree on a dataset, `fitcensemble` generates many bootstrap replicas of the dataset and grows decision trees on these replicas. `fitcensemble` obtains each bootstrap replica by randomly selecting `N` observations out of `N` with replacement, where `N` is the dataset size. To find the predicted response of a trained ensemble, `predict` take an average over predictions from individual trees.

Drawing `N` out of `N` observations with replacement omits on average 37% ( $1/e$ ) of observations for each decision tree. These are "out-of-bag" observations. For each observation, `oobLoss` estimates the out-of-bag prediction by averaging over predictions from all trees in the ensemble for which this observation is out of bag. It then compares the computed prediction against the true response for this observation. It calculates the out-of-bag error by comparing the out-of-bag predicted responses against the true responses for all observations used for training. This out-of-bag average is an unbiased estimator of the true ensemble error.

## See Also

`oobLoss` | `oobMargin` | `oobPredict`

# oobError

**Class:** TreeBagger

Out-of-bag error

## Syntax

```
err = oobError(B)
err = oobError(B, 'param1', val1, 'param2', val2, ...)
```

## Description

`err = oobError(B)` computes the misclassification probability (for classification trees) or mean squared error (for regression trees) for out-of-bag observations in the training data, using the trained bagger `B`. `err` is a vector of length `NTrees`, where `NTrees` is the number of trees in the ensemble.

`err = oobError(B, 'param1', val1, 'param2', val2, ...)` specifies optional parameter name/value pairs:

`'Mode'` Character vector indicating how `oobError` computes errors. If set to `'cumulative'` (default), the method computes cumulative errors and `err` is a vector of length `NTrees`, where the first element gives error from trees (1), second element gives error from trees (1:2) etc., up to trees (1:NTrees). If set to `'individual'`, `err` is a vector of length `NTrees`, where each element is an error from each tree in the ensemble. If set to `'ensemble'`, `err` is a scalar showing the cumulative error for the entire ensemble.

'Trees'	Vector of indices indicating what trees to include in this calculation. By default, this argument is set to 'all' and the method uses all trees. If 'Trees' is a numeric vector, the method returns a vector of length <code>NTrees</code> for 'cumulative' and 'individual' modes, where <code>NTrees</code> is the number of elements in the input vector, and a scalar for 'ensemble' mode. For example, in the 'cumulative' mode, the first element gives error from <code>trees(1)</code> , the second element gives error from <code>trees(1:2)</code> etc.
'TreeWeights'	Vector of tree weights. This vector must have the same length as the 'Trees' vector. <code>oobError</code> uses these weights to combine output from the specified trees by taking a weighted average instead of the simple nonweighted majority vote. You cannot use this argument in the 'individual' mode.

## Algorithms

`oobError` estimates the weighted ensemble error for out-of-bag observations. That is, `oobError` applies `error` to the training data stored in the input `TreeBagger` model `B`, and selects the out-of-bag observations for each tree to compose the ensemble error.

- `B.X` and `B.Y` are the training data predictors and responses, respectively.
- `B.OOBIndices` specifies which observations are out-of-bag for each tree in the ensemble.
- `B.W` specifies the observation weights.
- Optionally:
  - Using the 'Mode' name-value pair argument, you can specify to return the individual, weighted ensemble error for each tree, or the entire, weighted ensemble error. By default, `oobError` returns the cumulative, weighted ensemble error.
  - Using the 'Trees' name-value pair argument, you can choose which trees to use in the ensemble error calculations.
  - Using the 'TreeWeights' name-value pair argument, you can attribute each tree with a weight.

`oobError` applies the algorithms described below. For more details, see `error` and `predict`.

For regression problems, `oobError` returns the weighted MSE.

- 1 `oobError` predicts responses for all out-of-bag observations.
- 2 The MSE estimate depends on the value of 'Mode'.
  - If you specify 'Mode', 'Individual', then `oobError` sets any in bag observations within a selected tree to the weighted sample average of the observed, training data responses. Then, `oobError` computes the weighted MSE for each selected tree.
  - If you specify 'Mode', 'Cumulative', then `oobError` returns a vector of cumulative, weighted MSEs, where  $MSE_t$  is the cumulative, weighted MSE for selected tree  $t$ . To compute  $MSE_t$ , for each observation that is out of bag for at least one tree through tree  $t$ , `oobError` computes the cumulative, weighted mean of the predicted responses through tree  $t$ . `oobError` sets observations that are in bag for all selected trees through tree  $t$  to the weighted sample average of the observed, training data responses. Then, `oobError` computes  $MSE_t$ .
  - If you specify 'Mode', 'Ensemble', then, for each observation that is out of bag for at least one tree, `oobError` computes the weighted mean over all selected trees. `oobError` sets observations that are in bag for all selected trees to the weighted sample average of the observed, training data responses. Then, `oobError` computes the weighted MSE, which is the same as the final, cumulative, weighted MSE.

In classification problems, `oobError` returns the weighted misclassification rate.

- 1 `oobError` predicts classes for all out-of-bag observations.
- 2 The weighted misclassification rate estimate depends on the value of 'Mode'.
  - If you specify 'Mode', 'Individual', then `oobError` sets any in bag observations within a selected tree to the predicted, weighted, most popular class over all training responses. If there are multiple most popular classes, `error` considers the one listed first in the `ClassNames` property of the `TreeBagger` model the most popular. Then, `oobError` computes the weighted misclassification rate for each selected tree.
  - If you specify 'Mode', 'Cumulative', then `oobError` returns a vector of cumulative, weighted misclassification rates, where  $e_t^*$  is the cumulative,

weighted misclassification rate for selected tree  $t$ . To compute  $e_t^*$ , for each observation that is out of bag for at least one tree through tree  $t$ , `oobError` finds the predicted, cumulative, weighted most popular class through tree  $t$ . `oobError` sets observations that are in bag for all selected trees through tree  $t$  to the weighted, most popular class over all training responses. If there are multiple most popular classes, `error` considers the one listed first in the `ClassNames` property of the `TreeBagger` model the most popular. Then, `oobError` computes  $e_t^*$ .

- If you specify `'Mode', 'Ensemble'`, then, for each observation that is out of bag for at least one tree, `oobError` computes the weighted, most popular class over all selected trees. `oobError` sets observations that are in bag for all selected trees through tree  $t$  to the predicted, weighted, most popular class over all training responses. If there are multiple most popular classes, `error` considers the one listed first in the `ClassNames` property of the `TreeBagger` model the most popular. Then, `oobError` computes the weighted misclassification rate, which is the same as the final, cumulative, weighted misclassification rate.

## See Also

`TreeBagger` | `error` | `oobPredict` | `oobQuantileError` | `predict`

## Topics

“Bootstrap Aggregation (Bagging) of Regression Trees” on page 18-141

“Bootstrap Aggregation (Bagging) of Classification Trees” on page 18-154



## OOBIndices property

**Class:** TreeBagger

Indicator matrix for out-of-bag observations

### Description

The `OOBIndices` property is a logical array of size `Nobs-by-NTrees` where `Nobs` is the number of observations in the training data and `NTrees` is the number of trees in the ensemble. The  $(I, J)$  element is true if observation  $I$  is out-of-bag for tree  $J$  and false otherwise. In other words, a true value means observation  $I$  was not selected for the training data used to grow tree  $J$ .

### See Also

`ClassificationTree` | `RegressionTree` | `TreeBagger` | `fitctree` | `fitrtree`

## OOBInstanceWeight property

**Class:** TreeBagger

Count of out-of-bag trees for each observation

### Description

The `OOBInstanceWeight` property is a numeric array of size `Nobs-by-1` containing the number of trees used for computing out-of-bag response for each observation. `Nobs` is the number of observations in the training data used to create the ensemble.

# oobLoss

**Class:** ClassificationBaggedEnsemble

Out-of-bag classification error

## Syntax

```
L = oobloss(ens)
L = oobloss(ens, Name, Value)
```

## Description

`L = oobloss(ens)` returns the classification error for `ens` computed for out-of-bag data.

`L = oobloss(ens, Name, Value)` computes error with additional options specified by one or more `Name, Value` pair arguments. You can specify several name-value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

## Input Arguments

### **ens**

A classification bagged ensemble, constructed with `fitcensemble`.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

**learners**

Indices of weak learners in the ensemble ranging from 1 to NumTrained. `oobLoss` uses only these learners for calculating loss.

**Default:** `1:NumTrained`

**lossfun**

Loss function, specified as the comma-separated pair consisting of 'LossFun' and a built-in, loss-function name or function handle.

- The following lists available loss functions. Specify one using its corresponding character vector.

Value	Description
'binodeviance'	Binomial deviance
'classiferror'	Classification error
'exponential'	Exponential
'hinge'	Hinge
'logit'	Logistic
'mincost'	Minimal expected misclassification cost (for classification scores that are posterior probabilities)
'quadratic'	Quadratic

'mincost' is appropriate for classification scores that are posterior probabilities. Bagged ensembles return posterior probabilities as classification scores by default.

- Specify your own function using function handle notation.

Suppose that  $n$  be the number of observations in  $X$  and  $K$  be the number of distinct classes (`numel(ens.ClassNames)`, `ens` is the input model). Your function must have this signature

```
lossvalue = lossfun(C,S,W,Cost)
```

where:

- The output argument `lossvalue` is a scalar.
- You choose the function name (`lossfun`).

- $C$  is an  $n$ -by- $K$  logical matrix with rows indicating which class the corresponding observation belongs. The column order corresponds to the class order in `ens.ClassNames`.

Construct  $C$  by setting  $C(p, q) = 1$  if observation  $p$  is in class  $q$ , for each row. Set all other elements of row  $p$  to 0.

- $S$  is an  $n$ -by- $K$  numeric matrix of classification scores. The column order corresponds to the class order in `ens.ClassNames`.  $S$  is a matrix of classification scores, similar to the output of `predict`.
- $W$  is an  $n$ -by-1 numeric vector of observation weights. If you pass  $W$ , the software normalizes them to sum to 1.
- $Cost$  is a  $K$ -by- $K$  numeric matrix of misclassification costs. For example,  $Cost = ones(K) - eye(K)$  specifies a cost of 0 for correct classification, and 1 for misclassification.

Specify your function using `'LossFun', @lossfun`.

For more details on loss functions, see “Classification Loss” on page 32-4449.

**Default:** `'classiferror'`

#### **mode**

Character vector representing the meaning of the output  $L$ :

- `'ensemble'` —  $L$  is a scalar value, the loss for the entire ensemble.
- `'individual'` —  $L$  is a vector with one element per trained learner.
- `'cumulative'` —  $L$  is a vector in which element  $J$  is obtained by using learners  $1:J$  from the input list of learners.

**Default:** `'ensemble'`

## Output Arguments

### **L**

Classification loss on page 32-4449 of the out-of-bag observations, a scalar.  $L$  can be a vector, or can represent a different quantity, depending on the name-value settings.

## Examples

### Estimate Out-Of-Bag Error

Load Fisher's iris data set.

```
load fisheriris
```

Grow a bag of 100 classification trees.

```
rng(1) % For reproducibility
ens = fitensemble(meas, species, 'Bag', 100, 'Tree', 'type', 'classification');
```

Estimate the out-of-bag classification error.

```
L = oobLoss(ens)
```

```
L =
```

```
    0.0533
```

## Definitions

### Out of Bag

Bagging, which stands for “bootstrap aggregation”, is a type of ensemble learning. To bag a weak learner such as a decision tree on a dataset, `fitensemble` generates many bootstrap replicas of the dataset and grows decision trees on these replicas.

`fitensemble` obtains each bootstrap replica by randomly selecting  $N$  observations out of  $N$  with replacement, where  $N$  is the dataset size. To find the predicted response of a trained ensemble, `predict` take an average over predictions from individual trees.

Drawing  $N$  out of  $N$  observations with replacement omits on average 37% ( $1/e$ ) of observations for each decision tree. These are "out-of-bag" observations. For each observation, `oobLoss` estimates the out-of-bag prediction by averaging over predictions from all trees in the ensemble for which this observation is out of bag. It then compares the computed prediction against the true response for this observation. It calculates the

out-of-bag error by comparing the out-of-bag predicted responses against the true responses for all observations used for training. This out-of-bag average is an unbiased estimator of the true ensemble error.

## Classification Loss

Classification loss functions measure the predictive inaccuracy of classification models. When comparing the same type of loss among many models, lower loss indicates a better predictive model.

Suppose that:

- $L$  is the weighted average classification loss.
- $n$  is the sample size.
- For binary classification:
  - $y_j$  is the observed class label. The software codes it as  $-1$  or  $1$  indicating the negative or positive class, respectively.
  - $f(X_j)$  is the raw classification score for observation (row)  $j$  of the predictor data  $X$ .
  - $m_j = y_j f(X_j)$  is the classification score for classifying observation  $j$  into the class corresponding to  $y_j$ . Positive values of  $m_j$  indicate correct classification and do not contribute much to the average loss. Negative values of  $m_j$  indicate incorrect classification and contribute to the average loss.
- For algorithms that support multiclass classification (that is,  $K \geq 3$ ):
  - $y_j^*$  is a vector of  $K - 1$  zeros, and a  $1$  in the position corresponding to the true, observed class  $y_j$ . For example, if the true class of the second observation is the third class and  $K = 4$ , then  $y_2^* = [0 \ 0 \ 1 \ 0]'$ . The order of the classes corresponds to the order in the `ClassNames` property of the input model.
  - $f(X_j)$  is the length  $K$  vector of class scores for observation  $j$  of the predictor data  $X$ . The order of the scores corresponds to the order of the classes in the `ClassNames` property of the input model.
  - $m_j = y_j^{*'} f(X_j)$ . Therefore,  $m_j$  is the scalar classification score that the model predicts for the true, observed class.
- The weight for observation  $j$  is  $w_j$ . The software normalizes the observation weights so that they sum to the corresponding prior class probability. The software also normalizes the prior probabilities so they sum to  $1$ . Therefore,

$$\sum_{j=1}^n w_j = 1.$$

The supported loss functions are:

- Binomial deviance, specified using 'LossFun', 'binodeviance'. Its equation is

$$L = \sum_{j=1}^n w_j \log \{1 + \exp[-2m_j]\}.$$

- Exponential loss, specified using 'LossFun', 'exponential'. Its equation is

$$L = \sum_{j=1}^n w_j \exp(-m_j).$$

- Classification error, specified using 'LossFun', 'classiferror'. It is the weighted fraction of misclassified observations, with equation

$$L = \sum_{j=1}^n w_j I\{\hat{y}_j \neq y_j\}.$$

$\hat{y}_j$  is the class label corresponding to the class with the maximal posterior probability.  $I\{x\}$  is the indicator function.

- Hinge loss, specified using 'LossFun', 'hinge'. Its equation is

$$L = \sum_{j=1}^n w_j \max\{0, 1 - m_j\}.$$

- Logit loss, specified using 'LossFun', 'logit'. Its equation is

$$L = \sum_{j=1}^n w_j \log(1 + \exp(-m_j)).$$

- Minimal cost, specified using 'LossFun', 'mincost'. The software computes the weighted minimal cost using this procedure for observations  $j = 1, \dots, n$ :

- 1 Estimate the 1-by- $K$  vector of expected classification costs for observation  $j$

$$\gamma_j = f(X_j)'C.$$



$f(X_j)$  is the column vector of class posterior probabilities for binary and multiclass classification.  $C$  is the cost matrix the input model stores in the property `Cost`.

- 2 For observation  $j$ , predict the class label corresponding to the minimum, expected classification cost:

$$\hat{y}_j = \min_{j=1, \dots, K} (\gamma_j).$$

- 3 Using  $C$ , identify the cost incurred ( $c_j$ ) for making the prediction.

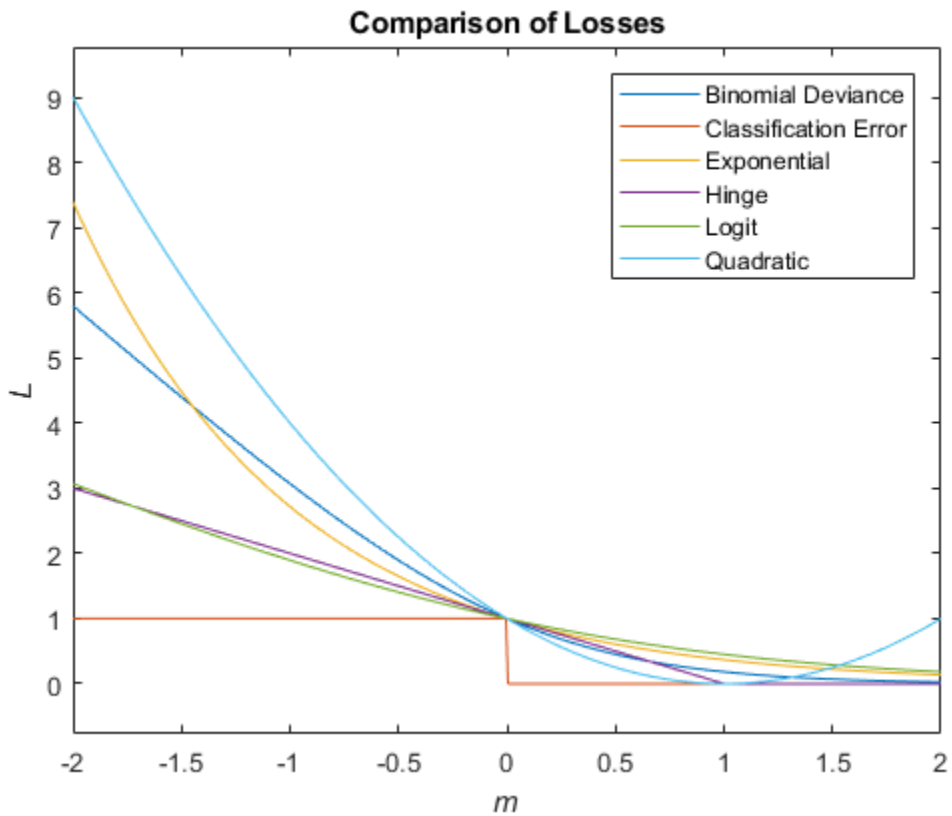
The weighted, average, minimum cost loss is

$$L = \sum_{j=1}^n w_j c_j.$$

- Quadratic loss, specified using `'LossFun', 'quadratic'`. Its equation is

$$L = \sum_{j=1}^n w_j (1 - m_j)^2.$$

This figure compares some of the loss functions for one observation over  $m$  (some functions are normalized to pass through  $[0,1]$ ).



## See Also

[loss](#) | [oobEdge](#) | [oobMargin](#) | [oobPredict](#)

# oobLoss

**Class:** RegressionBaggedEnsemble

Out-of-bag regression error

## Syntax

`L = oobLoss(ens)`

`L = oobLoss(ens, Name, Value)`

## Description

`L = oobLoss(ens)` returns the mean squared error for `ens` computed for out-of-bag data.

`L = oobLoss(ens, Name, Value)` computes error with additional options specified by one or more `Name, Value` pair arguments. You can specify several name-value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

## Input Arguments

### **ens**

A regression bagged ensemble, constructed with `fitrensemble`.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

**learners**

Indices of weak learners in the ensemble ranging from 1 to NumTrained. `oobLoss` uses only these learners for calculating loss.

**Default:** `1:NumTrained`

**lossfun**

Function handle for loss function, or 'mse', meaning mean squared error. If you pass a function handle `fun`, `oobLoss` calls it as

```
FUN(Y, Yfit, W)
```

where `Y`, `Yfit`, and `W` are numeric vectors of the same length. `Y` is the observed response, `Yfit` is the predicted response, and `W` is the observation weights.

**Default:** 'mse'

**mode**

Character vector representing the meaning of the output `L`:

- 'ensemble' — `L` is a scalar value, the loss for the entire ensemble.
- 'individual' — `L` is a vector with one element per trained learner.
- 'cumulative' — `L` is a vector in which element `J` is obtained by using learners `1:J` from the input list of learners.

**Default:** 'ensemble'

## Output Arguments

**L**

Mean squared error of the out-of-bag observations, a scalar. `L` can be a vector, or can represent a different quantity, depending on the name-value settings.

## Examples

### Find Out-of-Bag Regression Error

Compute the out-of-bag error for the `carsmall` data.

Load the `carsmall` data set and select engine displacement, horsepower, and vehicle weight as predictors.

```
load carsmall
X = [Displacement Horsepower Weight];
```

Train an ensemble of bagged regression trees.

```
ens = fitrensemble(X,MPG, 'Method','Bag');
```

Find the out-of-bag error.

```
rng(10, 'twister') % For reproducibility
L = oobLoss(ens)

L = 16.7197
```

## Definitions

### Out of Bag

Bagging, which stands for “bootstrap aggregation”, is a type of ensemble learning. To bag a weak learner such as a decision tree on a dataset, `fitrensemble` generates many bootstrap replicas of the dataset and grows decision trees on these replicas. `fitrensemble` obtains each bootstrap replica by randomly selecting  $N$  observations out of  $N$  with replacement, where  $N$  is the dataset size. To find the predicted response of a trained ensemble, `predict` takes an average over predictions from individual trees.

Drawing  $N$  out of  $N$  observations with replacement omits on average 37% ( $1/e$ ) of observations for each decision tree. These are “out-of-bag” observations. For each observation, `oobLoss` estimates the out-of-bag prediction by averaging over predictions from all trees in the ensemble for which this observation is out of bag. It then compares the computed prediction against the true response for this observation. It calculates the

out-of-bag error by comparing the out-of-bag predicted responses against the true responses for all observations used for training. This out-of-bag average is an unbiased estimator of the true ensemble error.

## See Also

`loss` | `oobPredict`

# oobMargin

**Class:** ClassificationBaggedEnsemble

Out-of-bag classification margins

## Syntax

```
margin = oobMargin(ens)
margin = oobMargin(ens, Name, Value)
```

## Description

`margin = oobMargin(ens)` returns out-of-bag classification margins.

`margin = oobMargin(ens, Name, Value)` calculates margins with additional options specified by one or more `Name, Value` pair arguments.

## Input Arguments

### **ens**

A classification bagged ensemble, constructed with `fitcensemble`.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### **learners**

Indices of weak learners in the ensemble ranging from 1 to `ens.NumTrained`. `oobEdge` uses only these learners for calculating loss.

**Default:** 1:NumTrained

## Output Arguments

**margin**

A numeric column vector of length `size(ens.X,1)`.

## Examples

### Find Out-of-Bag Classification Margins

Find the out-of-bag margins for a bagged ensemble from the Fisher iris data.

Load the sample data set.

```
load fisheriris
```

Train an ensemble of bagged classification trees.

```
ens = fitcensemble(meas,species,'Method','Bag');
```

Find the number of out-of-bag margins that are equal to 1.

```
rng(10,'twister') % For reproducibility
margin = oobMargin(ens);
sum(margin == 1)

ans = 105
```

## Definitions

### Out of Bag

Bagging, which stands for “bootstrap aggregation”, is a type of ensemble learning. To bag a weak learner such as a decision tree on a dataset, `fitensemble` generates many bootstrap replicas of the dataset and grows decision trees on these replicas.



`fitensemble` obtains each bootstrap replica by randomly selecting  $N$  observations out of  $N$  with replacement, where  $N$  is the dataset size. To find the predicted response of a trained ensemble, `predict` takes an average over predictions from individual trees.

Drawing  $N$  out of  $N$  observations with replacement omits on average 37% ( $1/e$ ) of observations for each decision tree. These are "out-of-bag" observations. For each observation, `oobLoss` estimates the out-of-bag prediction by averaging over predictions from all trees in the ensemble for which this observation is out of bag. It then compares the computed prediction against the true response for this observation. It calculates the out-of-bag error by comparing the out-of-bag predicted responses against the true responses for all observations used for training. This out-of-bag average is an unbiased estimator of the true ensemble error.

## Margin

The classification margin is the difference between the classification score for the true class and maximal classification score for the false classes. Margin is a column vector with the same number of rows as in the matrix `ens.X`.

## See Also

`margin` | `oobEdge` | `oobLoss` | `oobPredict`

## oobMargin

**Class:** TreeBagger

Out-of-bag margins

### Syntax

```
mar = oobMargin(B)
mar = oobMargin(B, 'param1', val1, 'param2', val2, ...)
```

### Description

`mar = oobMargin(B)` computes an `Nobs-by-NTrees` matrix of classification margins for out-of-bag observations in the training data, using the trained bagger `B`.

`mar = oobMargin(B, 'param1', val1, 'param2', val2, ...)` specifies optional parameter name/value pairs:

`'Mode'`

Character vector indicating how `oobMargin` computes errors. If set to `'cumulative'` (default), the method computes cumulative margins and `mar` is an `Nobs-by-NTrees` matrix, where the first column gives margins from `trees(1)`, second column gives margins from `trees(1:2)` etc., up to `trees(1:NTrees)`. If set to `'individual'`, `mar` is an `Nobs-by-NTrees` matrix, where each column gives margins from each tree in the ensemble. If set to `'ensemble'`, `mar` is a single column of length `Nobs` showing the cumulative margins for the entire ensemble.

'Trees'	Vector of indices indicating what trees to include in this calculation. By default, this argument is set to 'all' and the method uses all trees. If 'Trees' is a numeric vector, the method returns an Nobs-by-NTrees matrix for 'cumulative' and 'individual' modes, where NTrees is the number of elements in the input vector, and a single column for 'ensemble' mode. For example, in the 'cumulative' mode, the first column gives margins from trees(1), the second column gives margins from trees(1:2) etc.
'TreeWeights'	Vector of tree weights. This vector must have the same length as the 'Trees' vector. oobMargin uses these weights to combine output from the specified trees by taking a weighted average instead of the simple nonweighted majority vote. You cannot use this argument in the 'individual' mode.

## See Also

`CompactTreeBagger.margin`

## oobMeanMargin

**Class:** TreeBagger

Out-of-bag mean margins

### Syntax

```
mar = oobMeanMargin(B)
mar = oobMeanMargin(B, 'param1', val1, 'param2', val2, ...)
```

### Description

`mar = oobMeanMargin(B)` computes average classification margins for out-of-bag observations in the training data, using the trained bagger `B`. `oobMeanMargin` averages the margins over all out-of-bag observations. `mar` is a row-vector of length `NTrees`, where `NTrees` is the number of trees in the ensemble.

`mar = oobMeanMargin(B, 'param1', val1, 'param2', val2, ...)` specifies optional parameter name/value pairs:

<code>'Mode'</code>	Character vector indicating how <code>oobMargin</code> computes errors. If set to <code>'cumulative'</code> (default), is a vector of length <code>NTrees</code> where the first element gives mean margin from trees (1), second column gives mean margins from trees (1:2) etc., up to trees (1:NTrees). If set to <code>'individual'</code> , <code>mar</code> is a vector of length <code>NTrees</code> , where each element is a mean margin from each tree in the ensemble. If set to <code>'ensemble'</code> , <code>mar</code> is a scalar showing the cumulative mean margin for the entire ensemble.
---------------------	--

- 'Trees' Vector of indices indicating what trees to include in this calculation. By default, this argument is set to 'all' and the method uses all trees. If 'Trees' is a numeric vector, the method returns a vector of length `NTrees` for 'cumulative' and 'individual' modes, where `NTrees` is the number of elements in the input vector, and a scalar for 'ensemble' mode. For example, in the 'cumulative' mode, the first element gives mean margin from `trees(1)`, the second element gives mean margin from `trees(1:2)` etc.
- 'TreeWeights' Vector of tree weights. This vector must have the same length as the 'Trees' vector. `oobMeanMargin` uses these weights to combine output from the specified trees by taking a weighted average instead of the simple nonweighted majority vote. You cannot use this argument in the 'individual' mode.

## See Also

`CompactTreeBagger.meanMargin`

## oobPermutedPredictorImportance

**Class:** ClassificationBaggedEnsemble

Predictor importance estimates by permutation of out-of-bag predictor observations for random forest of classification trees

### Syntax

```
Imp = oobPermutedPredictorImportance (Mdl)
Imp = oobPermutedPredictorImportance (Mdl, Name, Value)
```

### Description

`Imp = oobPermutedPredictorImportance (Mdl)` returns a vector of out-of-bag, predictor importance estimates by permutation on page 32-4471 using the random forest of classification trees `Mdl`. `Mdl` must be a `ClassificationBaggedEnsemble` model object.

`Imp = oobPermutedPredictorImportance (Mdl, Name, Value)` uses additional options specified by one or more `Name, Value` pair arguments. For example, you can speed up computation using parallel computing or indicate which trees to use in the predictor importance estimation.

### Input Arguments

**Mdl** — Random forest of classification trees

`ClassificationBaggedEnsemble` model object

Random forest of classification trees, specified as a `ClassificationBaggedEnsemble` model object created by `fitcensemble`.

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single

quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### **Learners** — Indices of learners to use in predictor importance estimation

`1:Mdl.NumTrained` (default) | numeric vector of positive integers

Indices of learners to use in predictor importance estimation, specified as the comma-separated pair consisting of 'Learners' and a numeric vector of positive integers.

Values must be at most `Mdl.NumTrained`. When

`ClassificationBaggedEnsemble.oobPermutedPredictorImportance` estimates the predictor importance, it includes the learners in `Mdl.Trained(Learners)` only, where *learners* is the value of 'Learners'.

Example: `'Learners', [1:2:Mdl.NumTrained]`

### **Options** — Parallel computing options

`[]` (default) | structure array returned by `statset`

Parallel computing options, specified as the comma-separated pair consisting of 'Options' and a structure array returned by `statset`. 'Options' requires a Parallel Computing Toolbox license.

`ClassificationBaggedEnsemble.oobPermutedPredictorImportance` uses the 'UseParallel' field only. `statset('UseParallel', true)` invokes a pool of workers.

Example: `'Options', statset('UseParallel', true)`

## Output Arguments

### **Imp** — Out-of-bag, predictor importance estimates by permutation

numeric vector

Out-of-bag, predictor importance estimates by permutation on page 32-4471, returned as a 1-by-*p* numeric vector. *p* is the number of predictor variables in the training data (`size(Mdl.X, 2)`). `Imp(j)` is the predictor importance of the predictor

`Mdl.PredictorNames(j)`.

## Examples

### Estimate Importance of Predictors

Load the `census1994` data set. Consider a model that predicts a person's salary category given their age, working class, education level, marital status, race, sex, capital gain and loss, and number of working hours per week.

```
load census1994
X = adultdata(:, {'age', 'workClass', 'education_num', 'marital_status', 'race', ...
    'sex', 'capital_gain', 'capital_loss', 'hours_per_week', 'salary'});
rng('default'); % For reproducibility
```

Train a random forest of 50 classification trees using the entire data set.

```
Mdl = fitcensemble(X, 'salary', 'Method', 'bag', 'NumLearningCycles', 50);
```

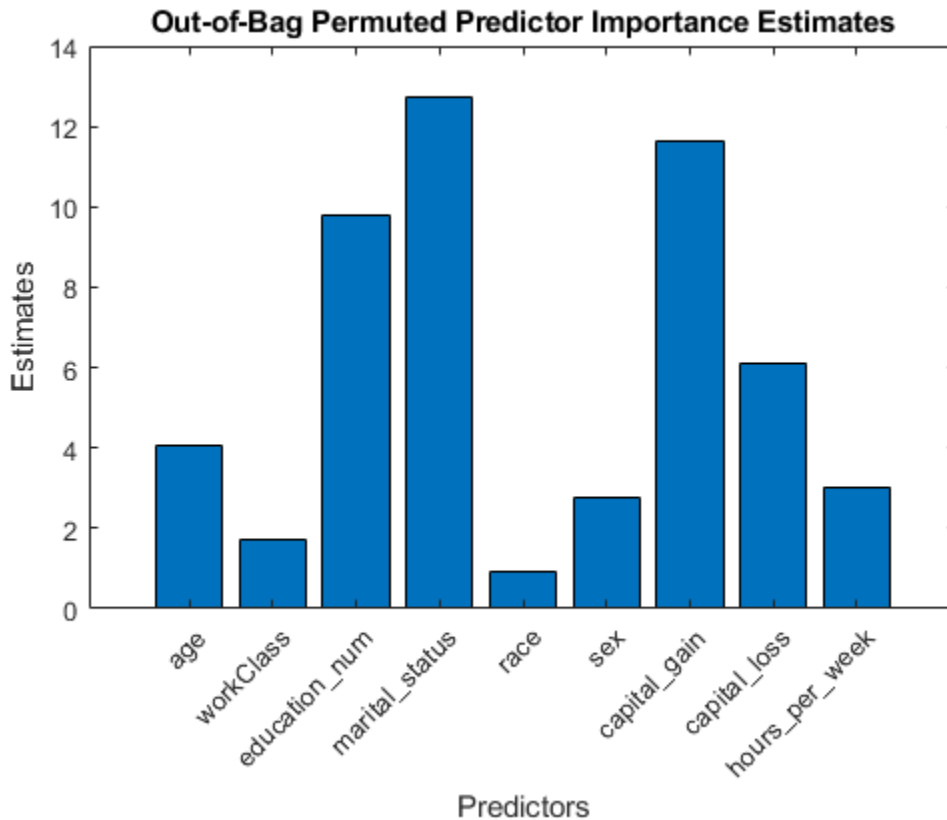
`Mdl` is a `ClassificationBaggedEnsemble` model.

Estimate predictor importance measures by permuting out-of-bag observations. Compare the estimates using a bar graph.

```
imp = oobPermutedPredictorImportance(Mdl);

figure;
bar(imp);
title('Out-of-Bag Permuted Predictor Importance Estimates');
ylabel('Estimates');
xlabel('Predictors');
h = gca;
h.XTickLabel = Mdl.PredictorNames;
h.XTickLabelRotation = 45;
h.TickLabelInterpreter = 'none';
```





`imp` is a 1-by-9 vector of predictor importance estimates. Larger values indicate predictors that have a greater influence on predictions. In this case, `marital_status` is the most important predictor, followed by `capital_gain`.

### Unbiased Estimates of Predictor Importance Using Parallel Computing

This example requires a Parallel Computing Toolbox™ license.

Load the `census1994` data set. Consider a model that predicts a person's salary category given their age, working class, education level, marital status, race, sex, capital gain and loss, and number of working hours per week.

```
load census1994
X = adultdata(:, {'age', 'workClass', 'education_num', 'marital_status', 'race', ...
    'sex', 'capital_gain', 'capital_loss', 'hours_per_week', 'salary'});
rng('default'); % For reproducibility
```

Display the number of categories represented in the categorical variables using `summary`.

```
summary(X)
```

Variables:

```
age: 32561×1 double
```

```
Values:
```

```
min      17
median   37
max      90
```

```
workClass: 32561×1 categorical
```

```
Values:
```

```
Federal-gov      960
Local-gov        2093
Never-worked      7
Private          22696
Self-emp-inc     1116
Self-emp-not-inc 2541
State-gov        1298
Without-pay      14
<undefined>     1836
```

```
education_num: 32561×1 double
```

```
Values:
```

```
min      1
median   10
max      16
```

```
marital_status: 32561×1 categorical
```

```
Values:
```

```
Divorced          4443
Married-AF-spouse 23
Married-civ-spouse 14976
```

Married-spouse-absent	418
Never-married	10683
Separated	1025
Widowed	993

race: 32561×1 categorical

Values:

Amer-Indian-Eskimo	311
Asian-Pac-Islander	1039
Black	3124
Other	271
White	27816

sex: 32561×1 categorical

Values:

Female	10771
Male	21790

capital\_gain: 32561×1 double

Values:

min	0
median	0
max	99999

capital\_loss: 32561×1 double

Values:

min	0
median	0
max	4356

hours\_per\_week: 32561×1 double

Values:

min	1
median	40
max	99

salary: 32561×1 categorical

Values:

```
<=50K    24720  
>50K     7841
```

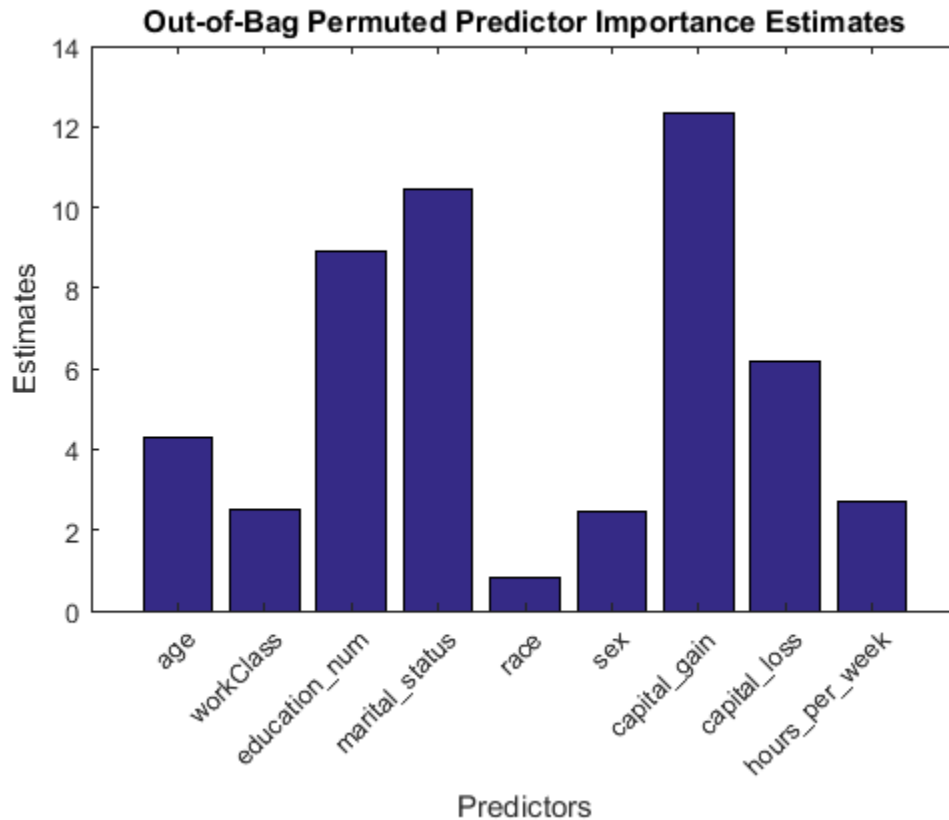
Because there are few categories represented in the categorical variables compared to levels in the continuous variables, the standard CART, predictor-splitting algorithm prefers splitting a continuous predictor over the categorical variables.

Train a random forest of 50 classification trees using the entire data set. To grow unbiased trees, specify usage of the curvature test for splitting predictors. Because there are missing values in the data, specify usage of surrogate splits.

```
t = templateTree('PredictorSelection','curvature','Surrogate','on');  
Mdl = fitcensemble(X,'salary','Method','bag','NumLearningCycles',50,...  
    'Learners',t);
```

Estimate predictor importance measures by permuting out-of-bag observations. Perform calculations in parallel. Compare the estimates using a bar graph.

```
options = statset('UseParallel',true);  
imp = oobPermutedPredictorImportance(Mdl,'Options',options);  
  
figure;  
bar(imp);  
title('Out-of-Bag Permuted Predictor Importance Estimates');  
ylabel('Estimates');  
xlabel('Predictors');  
h = gca;  
h.XTickLabel = Mdl.PredictorNames;  
h.XTickLabelRotation = 45;  
h.TickLabelInterpreter = 'none';
```



In this case, `capital_gain` is the most important predictor, followed by `marital_status`. Compare these results to the results in “Estimate Importance of Predictors” on page 32-4465.

## Definitions

### Out-of-Bag, Predictor Importance Estimates by Permutation

Out-of-bag, predictor importance estimates by permutation measure how influential the predictor variables in the model are at predicting the response. The influence of a predictor increases with the value of this measure.

If a predictor is influential in prediction, then permuting its values should affect the model error. If a predictor is not influential, then permuting its values should have little to no effect on the model error.

The following process describes the estimation of out-of-bag predictor importance values by permutation. Suppose that  $R$  is a random forest of  $T$  learners and  $p$  is the number of predictors in the training data.

- 1 For tree  $t$ ,  $t = 1, \dots, T$ :
  - a Identify the out-of-bag observations and the indices of the predictor variables that were split to grow tree  $t$ ,  $s_t \subseteq \{1, \dots, p\}$ .
  - b Estimate the out-of-bag error  $\varepsilon_t$ .
  - c For each predictor variable  $x_j$ ,  $j \in s_t$ :
    - i Randomly permute the observations of  $x_j$ .
    - ii Estimate the model error,  $\varepsilon_{ij}$ , using the out-of-bag observations containing the permuted values of  $x_j$ .
    - iii Take the difference  $d_{ij} = \varepsilon_{ij} - \varepsilon_t$ . Predictor variables not split when growing tree  $t$  are attributed a difference of 0.
- 2 For each predictor variable in the training data, compute the mean,  $\bar{d}_j$ , and standard deviation,  $\sigma_j$ , of the differences over the learners,  $j = 1, \dots, p$ .
- 3 The out-of-bag predictor importance by permutation for  $x_j$  is  $\bar{d}_j / \sigma_j$ .

## Tips

When growing a random forest using `fitcensemble`:

- Standard CART tends to select split predictors containing many distinct values, e.g., continuous variables, over those containing few distinct values, e.g., categorical variables [3]. If the predictor data set is heterogeneous, or if there are predictors that have relatively fewer distinct values than other variables, then consider specifying the curvature or interaction test.
- Trees grown using standard CART are not sensitive to predictor variable interactions. Also, such trees are less likely to identify important variables in the presence of many irrelevant predictors than the application of the interaction test. Therefore, to account

for predictor interactions and identify importance variables in the presence of many irrelevant variables, specify the interaction test [2].

For more details, see `templateTree`.

## References

- [1] Breiman, L., J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Boca Raton, FL: CRC Press, 1984.
- [2] Loh, W.Y. “Regression Trees with Unbiased Variable Selection and Interaction Detection.” *Statistica Sinica*, Vol. 12, 2002, pp. 361–386.
- [3] Loh, W.Y. and Y.S. Shih. “Split Selection Methods for Classification Trees.” *Statistica Sinica*, Vol. 7, 1997, pp. 815–840.

## See Also

`ClassificationBaggedEnsemble` | `fitcensemble` | `predictorImportance`

**Introduced in R2016b**

## `oobPermutedPredictorImportance`

**Class:** `RegressionBaggedEnsemble`

Predictor importance estimates by permutation of out-of-bag predictor observations for random forest of regression trees

### Syntax

```
Imp = oobPermutedPredictorImportance (Mdl)
Imp = oobPermutedPredictorImportance (Mdl, Name, Value)
```

### Description

`Imp = oobPermutedPredictorImportance (Mdl)` returns a vector of out-of-bag, predictor importance estimates by permutation on page 32-4480 using the random forest of regression trees `Mdl`. `Mdl` must be a `RegressionBaggedEnsemble` model object.

`Imp = oobPermutedPredictorImportance (Mdl, Name, Value)` uses additional options specified by one or more `Name, Value` pair arguments. For example, you can speed up computation using parallel computing or indicate which trees to use in the predictor importance estimation.

### Input Arguments

**`mdl` — Random forest of regression trees**

`RegressionBaggedEnsemble` model object

Random forest of regression trees, specified as a `RegressionBaggedEnsemble` model object created by `fitrensemble`.

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single



quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### **Learners** — Indices of learners to use in predictor importance estimation

`1:Mdl.NumTrained` (default) | numeric vector of positive integers

Indices of learners to use in predictor importance estimation, specified as the comma-separated pair consisting of 'Learners' and a numeric vector of positive integers.

Values must be at most `Mdl.NumTrained`. When

`RegressionBaggedEnsemble.oobPermutedPredictorImportance` estimates the predictor importance, it includes the learners in `Mdl.Trained(learners)` only, where `learners` is the value of 'Learners'.

Example: `'Learners', [1:2:Mdl.NumTrained]`

### **Options** — Parallel computing options

`[]` (default) | structure array returned by `statset`

Parallel computing options, specified as the comma-separated pair consisting of 'Options' and a structure array returned by `statset`. 'Options' requires a Parallel Computing Toolbox license.

`RegressionBaggedEnsemble.oobPermutedPredictorImportance` uses the 'UseParallel' field only. `statset('UseParallel', true)` invokes a pool of workers.

Example: `'Options', statset('UseParallel', true)`

## Output Arguments

### **Imp** — Out-of-bag, predictor importance estimates by permutation

numeric vector

Out-of-bag, predictor importance estimates by permutation on page 32-4480, returned as a 1-by- $p$  numeric vector.  $p$  is the number of predictor variables in the training data (`size(Mdl.X, 2)`). `Imp(j)` is the predictor importance of the predictor

`Mdl.PredictorNames(j)`.

## Examples

### Estimate Importance of Predictors

Load the `carsmall` data set. Consider a model that predicts the mean fuel economy of a car given its acceleration, number of cylinders, engine displacement, horsepower, manufacturer, model year, and weight. Consider `Cylinders`, `Mfg`, and `Model_Year` as categorical variables.

```
load carsmall
Cylinders = categorical(Cylinders);
Mfg = categorical(cellstr(Mfg));
Model_Year = categorical(Model_Year);
X = table(Acceleration,Cylinders,Displacement,Horsepower,Mfg,...
         Model_Year,Weight,MPG);
rng('default'); % For reproducibility
```

Train a random forest of 500 regression trees using the entire data set.

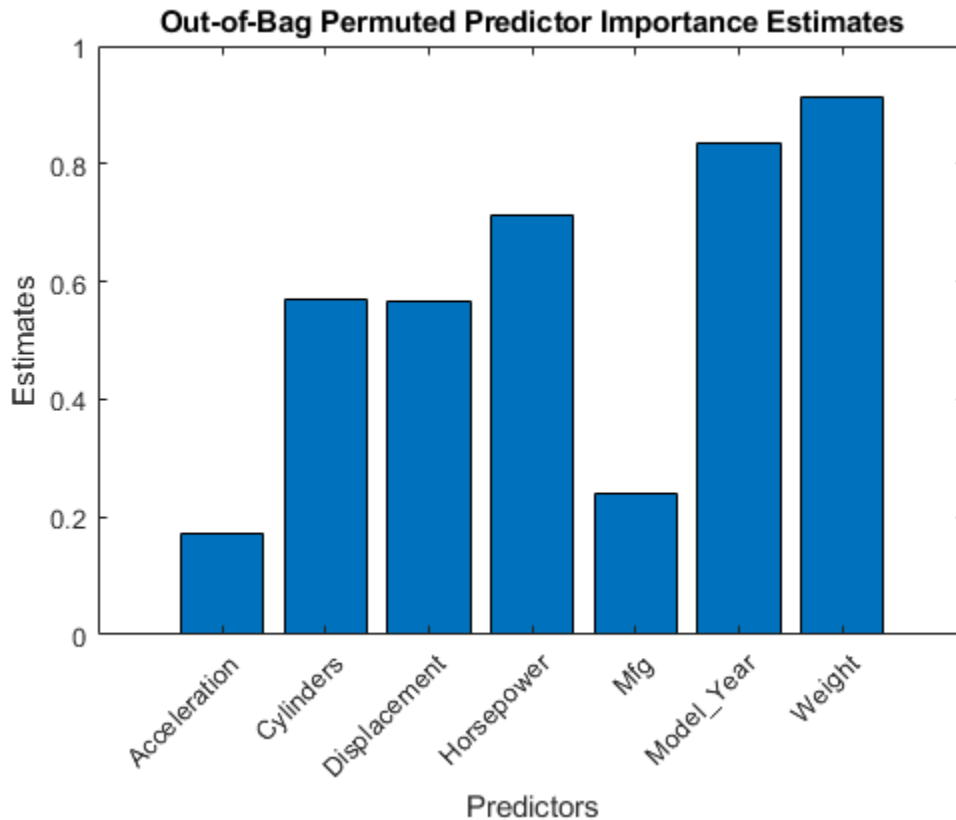
```
Mdl = fitrensemble(X,'MPG','Method','bag','NumLearningCycles',500);
```

`Mdl` is a `RegressionBaggedEnsemble` model.

Estimate predictor importance measures by permuting out-of-bag observations. Compare the estimates using a bar graph.

```
imp = oobPermutedPredictorImportance(Mdl);

figure;
bar(imp);
title('Out-of-Bag Permuted Predictor Importance Estimates');
ylabel('Estimates');
xlabel('Predictors');
h = gca;
h.XTickLabel = Mdl.PredictorNames;
h.XTickLabelRotation = 45;
h.TickLabelInterpreter = 'none';
```



`imp` is a 1-by-7 vector of predictor importance estimates. Larger values indicate predictors that have a greater influence on predictions. In this case, `Weight` is the most important predictor, followed by `Model_Year`.

### Unbiased Estimates of Predictor Importance Using Parallel Computing

This example requires a Parallel Computing Toolbox™ license.

Load the `carsmall` data set. Consider a model that predicts the mean fuel economy of a car given its acceleration, number of cylinders, engine displacement, horsepower,

manufacturer, model year, and weight. Consider `Cylinders`, `Mfg`, and `Model_Year` as categorical variables.

```
load carsmall
Cylinders = categorical(Cylinders);
Mfg = categorical(cellstr(Mfg));
Model_Year = categorical(Model_Year);
X = table(Acceleration,Cylinders,Displacement,Horsepower,Mfg,...
         Model_Year,Weight,MPG);
rng('default'); % For reproducibility
```

Display the number of categories represented in the categorical variables.

```
numCylinders = numel(categories(Cylinders))
numMfg = numel(categories(Mfg))
numModelYear = numel(categories(Model_Year))
```

```
numCylinders =
```

```
3
```

```
numMfg =
```

```
28
```

```
numModelYear =
```

```
3
```

Because there are 3 categories only in `Cylinders` and `Model_Year`, the standard CART, predictor-splitting algorithm prefers splitting a continuous predictor over these two variables.

Train a random forest of 500 regression trees using the entire data set. To grow unbiased trees, specify usage of the curvature test for splitting predictors. Because there are missing values in the data, specify usage of surrogate splits.

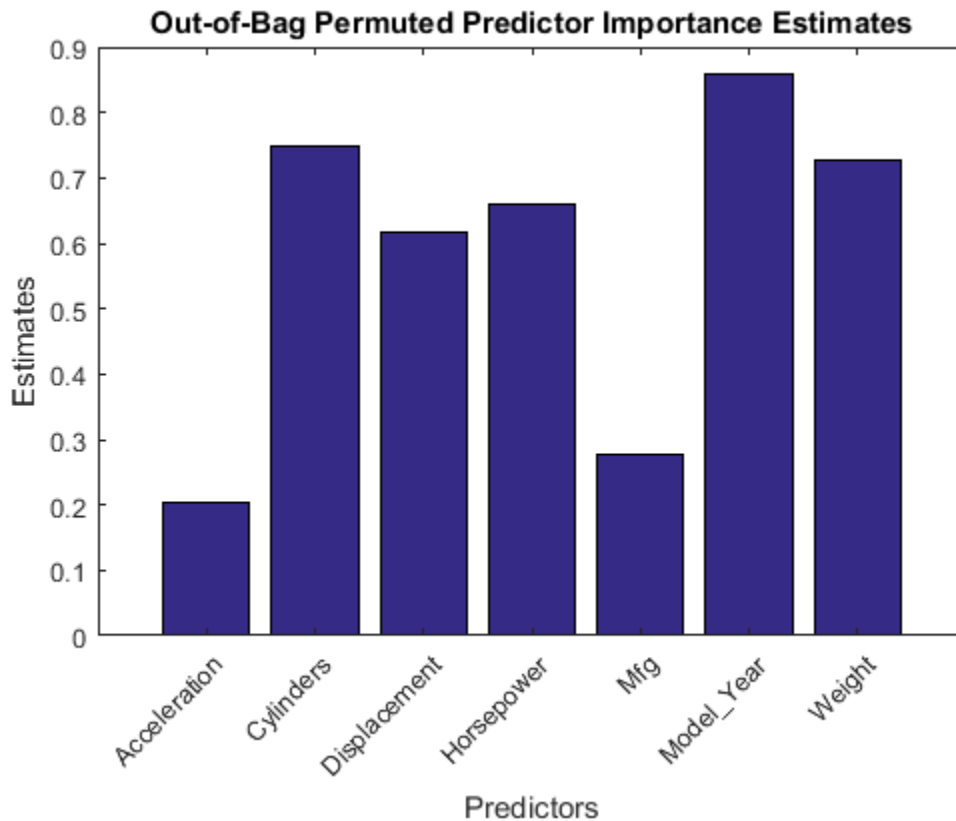
```
t = templateTree('PredictorSelection','curvature','Surrogate','on');
Mdl = fitrensemble(X,'MPG','Method','bag','NumLearningCycles',500,...
                 'Learners',t);
```

Estimate predictor importance measures by permuting out-of-bag observations. Perform calculations in parallel. Compare the estimates using a bar graph.

```
options = statset('UseParallel',true);
imp = oobPermutedPredictorImportance(Mdl,'Options',options);

figure;
bar(imp);
title('Out-of-Bag Permuted Predictor Importance Estimates');
ylabel('Estimates');
xlabel('Predictors');
h = gca;
h.XTickLabel = Mdl.PredictorNames;
h.XTickLabelRotation = 45;
h.TickLabelInterpreter = 'none';
```

```
Starting parallel pool (parpool) using the 'local' profile ... connected to 4 workers.
```



In this case, `Model_Year` is the most important predictor, followed by `Cylinders`. Compare these results to the results in “Estimate Importance of Predictors” on page 32-4475.

## Definitions

### Out-of-Bag, Predictor Importance Estimates by Permutation

Out-of-bag, predictor importance estimates by permutation measure how influential the predictor variables in the model are at predicting the response. The influence of a predictor increases with the value of this measure.

If a predictor is influential in prediction, then permuting its values should affect the model error. If a predictor is not influential, then permuting its values should have little to no effect on the model error.

The following process describes the estimation of out-of-bag predictor importance values by permutation. Suppose that  $R$  is a random forest of  $T$  learners and  $p$  is the number of predictors in the training data.

- 1 For tree  $t$ ,  $t = 1, \dots, T$ :
  - a Identify the out-of-bag observations and the indices of the predictor variables that were split to grow tree  $t$ ,  $s_t \subseteq \{1, \dots, p\}$ .
  - b Estimate the out-of-bag error  $\varepsilon_t$ .
  - c For each predictor variable  $x_j$ ,  $j \in s_t$ :
    - i Randomly permute the observations of  $x_j$ .
    - ii Estimate the model error,  $\varepsilon_{ij}$ , using the out-of-bag observations containing the permuted values of  $x_j$ .
    - iii Take the difference  $d_{ij} = \varepsilon_{ij} - \varepsilon_t$ . Predictor variables not split when growing tree  $t$  are attributed a difference of 0.
- 2 For each predictor variable in the training data, compute the mean,  $\bar{d}_j$ , and standard deviation,  $\sigma_j$ , of the differences over the learners,  $j = 1, \dots, p$ .
- 3 The out-of-bag predictor importance by permutation for  $x_j$  is  $\bar{d}_j / \sigma_j$ .

## Tips

When growing a random forest using `fitrensemble`:

- Standard CART tends to select split predictors containing many distinct values, e.g., continuous variables, over those containing few distinct values, e.g., categorical variables [3]. If the predictor data set is heterogeneous, or if there are predictors that have relatively fewer distinct values than other variables, then consider specifying the curvature or interaction test.
- Trees grown using standard CART are not sensitive to predictor variable interactions. Also, such trees are less likely to identify important variables in the presence of many irrelevant predictors than the application of the interaction test. Therefore, to account

for predictor interactions and identify importance variables in the presence of many irrelevant variables, specify the `interaction` test [2].

For more details, see `templateTree`.

## References

- [1] Breiman, L., J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Boca Raton, FL: CRC Press, 1984.
- [2] Loh, W.Y. “Regression Trees with Unbiased Variable Selection and Interaction Detection.” *Statistica Sinica*, Vol. 12, 2002, pp. 361–386.
- [3] Loh, W.Y. and Y.S. Shih. “Split Selection Methods for Classification Trees.” *Statistica Sinica*, Vol. 7, 1997, pp. 815–840.

## See Also

`RegressionBaggedEnsemble` | `fitrensemble` | `plotPartialDependence` | `predictorImportance`

**Introduced in R2016b**



# OOBPermutedVarCountRaiseMargin property

**Class:** TreeBagger

Variable importance for raising margin

## Description

The `OOBPermutedVarCountRaiseMargin` property is a numeric array of size 1-by-`Nvars` containing a measure of variable importance for each predictor. For any variable, the measure is the difference between the number of raised margins and the number of lowered margins if the values of that variable are permuted across the out-of-bag observations. This measure is computed for every tree, then averaged over the entire ensemble and divided by the standard deviation over the entire ensemble. This property is empty for regression trees.

## OOBPermutedVarDeltaError property

**Class:** TreeBagger

Variable importance for prediction error

### Description

The `OOBPermutedVarDeltaError` property is a numeric array of size 1-by-*Nvars* containing a measure of importance for each predictor variable (feature). For any variable, the measure is the increase in prediction error if the values of that variable are permuted across the out-of-bag observations. This measure is computed for every tree, then averaged over the entire ensemble and divided by the standard deviation over the entire ensemble.

# OOBPermutedVarDeltaMeanMargin property

**Class:** TreeBagger

Variable importance for classification margin

## Description

The `OOBPermutedVarDeltaMeanMargin` property is a numeric array of size 1-by-Nvars containing a measure of importance for each predictor variable (feature). For any variable, the measure is the decrease in the classification margin if the values of that variable are permuted across the out-of-bag observations. This measure is computed for every tree, then averaged over the entire ensemble and divided by the standard deviation over the entire ensemble. This property is empty for regression trees.

## oobPredict

**Class:** ClassificationBaggedEnsemble

Predict out-of-bag response of ensemble

### Syntax

```
[label, score] = oobPredict(ens)
[label, score] = oobPredict(ens, Name, Value)
```

### Description

`[label, score] = oobPredict(ens)` returns class labels and scores for `ens` for out-of-bag data.

`[label, score] = oobPredict(ens, Name, Value)` computes labels and scores with additional options specified by one or more `Name, Value` pair arguments.

### Input Arguments

**ens**

A classification bagged ensemble, constructed with `fitcensemble`.

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

**learners**

Indices of weak learners in the ensemble ranging from 1 to `ens.NumTrained`. `oobEdge` uses only these learners for calculating loss.

**Default:** `1:NumTrained`

## Output Arguments

**label**

Classification labels of the same data type as the training data `Y`. There are `N` elements or rows, where `N` is the number of training observations. The label is the class with the highest score. In case of a tie, the label is earliest in `ens.ClassNames`.

**score**

An `N`-by-`K` numeric matrix for `N` observations and `K` classes. A high score indicates that an observation is likely to come from this class. Scores are in the range 0 to 1.

## Examples

### Find Out-of-Bag Response of Classification Ensemble

Find the out-of-bag predictions and scores for the Fisher iris data. Find the scores with notable uncertainty in the resulting classifications.

Load the sample data set.

```
load fisheriris
```

Train an ensemble of bagged classification trees.

```
ens = fitcensemble(meas,species,'Method','Bag');
```

Find the out-of-bag predictions and scores.

```
rng(10,'twister') % For reproducibility  
[label,score] = oobPredict(ens);
```

Find the scores in the range  $(0.2, 0.8)$ . These scores have notable uncertainty in the resulting classifications.

```
unsure = ((score > .2) & (score < .8));  
sum(sum(unsure)) % Number of uncertain predictions  
  
ans = 18
```

## Definitions

### Out of Bag

Bagging, which stands for “bootstrap aggregation”, is a type of ensemble learning. To bag a weak learner such as a decision tree on a dataset, `fitensemble` generates many bootstrap replicas of the dataset and grows decision trees on these replicas.

`fitensemble` obtains each bootstrap replica by randomly selecting  $N$  observations out of  $N$  with replacement, where  $N$  is the dataset size. To find the predicted response of a trained ensemble, `predict` takes an average over predictions from individual trees.

Drawing  $N$  out of  $N$  observations with replacement omits on average 37% ( $1/e$ ) of observations for each decision tree. These are “out-of-bag” observations. For each observation, `oobLoss` estimates the out-of-bag prediction by averaging over predictions from all trees in the ensemble for which this observation is out of bag. It then compares the computed prediction against the true response for this observation. It calculates the out-of-bag error by comparing the out-of-bag predicted responses against the true responses for all observations used for training. This out-of-bag average is an unbiased estimator of the true ensemble error.

### Score (ensemble)

For ensembles, a classification score represents the confidence of a classification into a class. The higher the score, the higher the confidence.

Different ensemble algorithms have different definitions for their scores. Furthermore, the range of scores depends on ensemble type. For example:

- `AdaBoostM1` scores range from  $-\infty$  to  $\infty$ .

- Bag scores range from 0 to 1.

## Algorithms

`oobPredict` and `predict` similarly predict classes and responses.

- In regression problems:
  - For each observation that is out of bag for at least one tree, `oobPredict` composes the weighted mean by selecting responses of trees in which the observation is out of bag. For this computation, the `'TreeWeights'` name-value pair argument specifies the weights.
  - For each observation that is in bag for all trees, the predicted response is the weighted mean of all of the training responses. For this computation, the `W` property of the `TreeBagger` model (i.e., the observation weights) specify the weights.
- In classification problems:
  - For each observation that is out of bag for at least one tree, `oobPredict` composes the weighted mean of the class posterior probabilities by selecting the trees in which the observation is out of bag. Consequently, the predicted class is the class corresponding to the largest weighted mean. For this computation, the `'TreeWeights'` name-value pair argument specifies the weights.
  - For each observation that is in bag for all trees, the predicted class is the weighted, most popular class over all training responses. For this computation, the `W` property of the `TreeBagger` model (i.e., the observation weights) specify the weights. If there are multiple most popular classes, `oobPredict` considers the one listed first in the `ClassNames` property of the `TreeBagger` model the most popular.

## See Also

`oobEdge` | `oobLoss` | `oobMargin` | `oobPredict` | `predict`

## **oobPredict**

**Class:** RegressionBaggedEnsemble

Predict out-of-bag response of ensemble

### **Syntax**

```
Yfit = oobPredict(ens)
Yfit = oobPredict(ens, Name, Value)
```

### **Description**

`Yfit = oobPredict(ens)` returns the predicted responses for the out-of-bag data in `ens`.

`Yfit = oobPredict(ens, Name, Value)` predicts responses with additional options specified by one or more `Name, Value` pair arguments.

### **Input Arguments**

#### **ens**

A regression bagged ensemble, constructed with `fitrensemble`.

### **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.



**learners**

Indices of weak learners in the ensemble ranging from 1 to NumTrained. oobLoss uses only these learners for calculating loss.

**Default:** 1:NumTrained

## Output Arguments

**Yfit**

A vector of predicted responses for out-of-bag data. Yfit has size(ens.X,1) elements.

You can find the indices of out-of-bag observations for weak learner L with the command

```
~ens.UseObsForLearner(:,L)
```

## Examples

### Find Out-of-Bag Response of Regression Ensemble

Compute the out-of-bag predictions for the carsmall data set. Display the first three terms of the fit.

Load the carsmall data set and select displacement, horsepower, and vehicle weight as predictors.

```
load carsmall
X = [Displacement Horsepower Weight];
```

Train an ensemble of bagged regression trees.

```
ens = fitensemble(X,MPG,'Method','Bag');
```

Find the out-of-bag predictions, and display the first three terms of the fit.

```
rng(10,'twister') % For reproducibility
Yfit = oobPredict(ens);
Yfit(1:3) % First three terms

ans =
```

```
15.9678  
15.1642  
15.0815
```

## Definitions

### Out of Bag

Bagging, which stands for “bootstrap aggregation”, is a type of ensemble learning. To bag a weak learner such as a decision tree on a dataset, `fitensemble` generates many bootstrap replicas of the dataset and grows decision trees on these replicas.

`fitensemble` obtains each bootstrap replica by randomly selecting `N` observations out of `N` with replacement, where `N` is the dataset size. To find the predicted response of a trained ensemble, `predict` takes an average over predictions from individual trees.

Drawing `N` out of `N` observations with replacement omits on average 37% ( $1/e$ ) of observations for each decision tree. These are "out-of-bag" observations. For each observation, `oobLoss` estimates the out-of-bag prediction by averaging over predictions from all trees in the ensemble for which this observation is out of bag. It then compares the computed prediction against the true response for this observation. It calculates the out-of-bag error by comparing the out-of-bag predicted responses against the true responses for all observations used for training. This out-of-bag average is an unbiased estimator of the true ensemble error.

### See Also

`oobLoss` | `predict`

# oobPredict

**Class:** TreeBagger

Ensemble predictions for out-of-bag observations

## Syntax

```
Y = oobPredict(B)
Y = oobPredict(B, 'param1', val1, 'param2', val2, ...)
```

## Description

`Y = oobPredict(B)` computes predicted responses using the trained bagger `B` for out-of-bag observations in the training data. The output has one prediction for each observation in the training data. The returned `Y` is a cell array of character vectors for classification and a numeric array for regression.

`Y = oobPredict(B, 'param1', val1, 'param2', val2, ...)` specifies optional parameter name/value pairs:

'Trees'	Array of tree indices to use for computation of responses. Default is 'all'.
'TreeWeights'	Array of <code>NTrees</code> weights for weighting votes from the specified trees.

## Algorithms

`oobPredict` and `predict` similarly predict classes and responses.

- In regression problems:
  - For each observation that is out of bag for at least one tree, `oobPredict` composes the weighted mean by selecting responses of trees in which the observation is out of bag. For this computation, the `'TreeWeights'` name-value pair argument specifies the weights.

- For each observation that is in bag for all trees, the predicted response is the weighted mean of all of the training responses. For this computation, the `w` property of the `TreeBagger` model (i.e., the observation weights) specify the weights.
- In classification problems:
  - For each observation that is out of bag for at least one tree, `oobPredict` composes the weighted mean of the class posterior probabilities by selecting the trees in which the observation is out of bag. Consequently, the predicted class is the class corresponding to the largest weighted mean. For this computation, the `'TreeWeights'` name-value pair argument specifies the weights.
  - For each observation that is in bag for all trees, the predicted class is the weighted, most popular class over all training responses. For this computation, the `w` property of the `TreeBagger` model (i.e., the observation weights) specify the weights. If there are multiple most popular classes, `oobPredict` considers the one listed first in the `ClassNames` property of the `TreeBagger` model the most popular.

## See Also

`TreeBagger` | `compact` | `oobQuantilePredict` | `predict`

## Topics

“Bootstrap Aggregation (Bagging) of Regression Trees” on page 18-141

“Bootstrap Aggregation (Bagging) of Classification Trees” on page 18-154

# oobQuantileError

**Class:** TreeBagger

Out-of-bag quantile loss of bag of regression trees

## Syntax

```
err = quantileError(Mdl)
err = quantileError(Mdl, Name, Value)
```

## Description

`err = quantileError(Mdl)` returns half of the out-of-bag on page 32-4501 mean absolute deviation (MAD) from comparing the true responses in `Mdl.Y` to the predicted, out-of-bag medians at `Mdl.X`, the predictor data, and using the bag of regression trees `Mdl`. `Mdl` must be a `TreeBagger` model object.

`err = quantileError(Mdl, Name, Value)` uses additional options specified by one or more `Name, Value` pair arguments. For example, specify quantile probabilities, the error type, or which trees to include in the quantile-regression-error estimation.

## Input Arguments

**Mdl** — Bag of regression trees

`TreeBagger` model object (default)

Bag of regression trees, specified as a `TreeBagger` model object created by `TreeBagger`.

- The value of `Mdl.Method` must be `regression`.
- When you train `Mdl` using `TreeBagger`, you must specify the name-value pair `'OOBPrediction', 'on'`. Consequently, `TreeBagger` saves required out-of-bag observation index matrix in `Mdl.OOBIndices`.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### Mode — Ensemble error type

'ensemble' (default) | 'cumulative' | 'individual'

Ensemble error type, specified as the comma-separated pair consisting of 'Mode' and a value in this table. Suppose  $\tau$  is the value of `Quantile`.

Value	Description
'cumulative'	<code>err</code> is a <code>Mdl.NumTrees-by-numel(<math>\tau</math>)</code> numeric matrix of cumulative quantile regression errors. <code>err(j,k)</code> is the $\tau(k)$ quantile regression error using the learners in <code>Mdl.Trees(1:j)</code> only.
'ensemble'	<code>err</code> is a 1-by- <code>numel(<math>\tau</math>)</code> numeric vector of cumulative quantile regression errors for the entire ensemble. <code>err(k)</code> is the $\tau(k)$ ensemble quantile regression error.
'individual'	<code>err</code> is a <code>Mdl.NumTrees-by-numel(<math>\tau</math>)</code> numeric matrix of quantile regression errors from individual learners. <code>err(j,k)</code> is the $\tau(k)$ quantile regression error using the learner in <code>Mdl.Trees(j)</code> only.

For 'cumulative' and 'individual', if you choose to include fewer trees in quantile estimation using `Trees`, then this action affects the number of rows in `err` and corresponding row indices.

Example: 'Mode', 'cumulative'

### Quantile — Quantile probability

0.5 (default) | numeric vector containing values in [0,1]

Quantile probability, specified as the comma-separated pair consisting of 'Quantile' and a numeric vector containing values in the interval [0,1]. For each observation (row) in `Mdl.X`, `TreeBagger.oobQuantileError` estimates corresponding quantiles for all probabilities in `Quantile`.

Example: 'Quantile', [0 0.25 0.5 0.75 1]

Data Types: single | double

**Trees** — Indices of trees to use in response estimation

'all' (default) | numeric vector of positive integers

Indices of trees to use in response estimation, specified as the comma-separated pair consisting of 'Trees' and 'all' or a numeric vector of positive integers. Indices correspond to the cells of `Mdl.Trees`; each cell therein contains a tree in the ensemble. The maximum value of `Trees` must be less than or equal to the number of trees in the ensemble (`Mdl.NumTrees`).

For 'all', `TreeBagger.oobQuantileError` uses all trees in the ensemble (that is, the indices `1:Mdl.NumTrees`).

Values other than the default can affect the number of rows in `err`.

Example: 'Trees', [1 10 Mdl.NumTrees]

Data Types: char | single | double

**TreeWeights** — Weights to attribute to responses from individual trees

ones(`Mdl.NumTrees`,1) (default) | numeric vector of nonnegative values

Weights to attribute to responses from individual trees, specified as the comma-separated pair consisting of 'TreeWeights' and a numeric vector of `numel(trees)` nonnegative values. `trees` is the value of `Trees`.

If you specify 'Mode', 'individual', then `TreeBagger.oobQuantileError` ignores `TreeWeights`.

Data Types: single | double

## Output Arguments

**err** — Half of out-of-bag quantile regression error

numeric scalar | numeric matrix

Half of the out-of-bag quantile regression error on page 32-4501, returned as a numeric scalar or  $T$ -by-`numel(tau)` matrix. `tau` is the value of `Quantile`.

$T$  depends on the values of `Mode`, `Trees`, and `Quantile`. Suppose that you specify 'Quantile', `tau` and 'Trees', `trees`.

- For 'Mode', 'cumulative', `err` is a `numel(trees)`-by-`numel(tau)` numeric matrix. `err(j,k)` is the  $\tau(k)$  cumulative, out-of-bag quantile regression error using the learners in `Mdl.Trees(trees(1:j))`.
- For 'Mode', 'ensemble', `err` is a 1-by-`numel(tau)` numeric vector. `err(k)` is the  $\tau(k)$  cumulative, out-of-bag quantile regression error using the learners in `Mdl.Trees(trees)`.
- For 'Mode', 'individual', `err` is a `numel(trees)`-by-`numel(tau)` numeric matrix. `err(j,k)` is the  $\tau(k)$  out-of-bag quantile regression error using the learner in `Mdl.Trees(trees(j))`.

## Examples

### Estimate Out-of-Bag Quantile Regression Error

Load the `carsmall` data set. Consider a model that predicts the fuel economy of a car given its engine displacement, weight, and number of cylinders. Consider `Cylinders` a categorical variable.

```
load carsmall
Cylinders = categorical(Cylinders);
X = table(Displacement,Weight,Cylinders,MPG);
```

Train an ensemble of bagged regression trees using the entire data set. Specify 100 weak learners and save the out-of-bag indices.

```
rng(1); % For reproducibility
Mdl = TreeBagger(100,X,'MPG','Method','regression','OOBPrediction','on');
```

`Mdl` is a `TreeBagger` ensemble.

Perform quantile regression, and out-of-bag estimate the MAD of the entire ensemble using the predicted conditional medians.

```
oobErr = oobQuantileError(Mdl)
```

```
oobErr =
    1.5321
```



`oobErr` is an unbiased estimate of the quantile regression error for the entire ensemble.

### Find Appropriate Ensemble Size Using Out-of-Bag Quantile Regression Error

Load the `carsmall` data set. Consider a model that predicts the fuel economy of a car given its engine displacement, weight, and number of cylinders.

```
load carsmall
X = table(Displacement,Weight,Cylinders,MPG);
```

Train an ensemble of bagged regression trees using the entire data set. Specify 250 weak learners and save the out-of-bag indices.

```
rng('default'); % For reproducibility
Mdl = TreeBagger(250,X,'MPG','Method','regression',...
    'OOBPrediction','on');
```

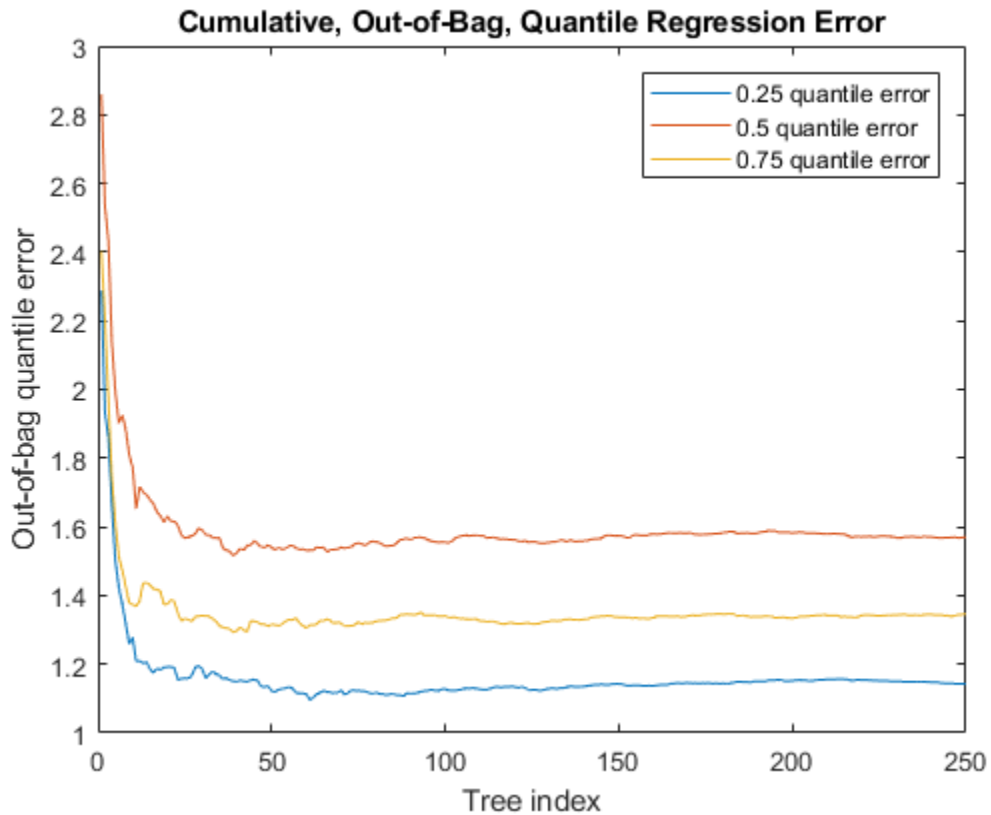
Estimate the cumulative; out-of-bag; 0.25, 0.5, and 0.75 quantile regression errors.

```
err = oobQuantileError(Mdl,'Quantile',[0.25 0.5 0.75],'Mode','cumulative');
```

`err` is an 250-by-3 matrix of cumulative, out-of-bag, quantile regression errors. Columns correspond to quantile probabilities and rows correspond to trees in the ensemble. The errors are cumulative, so they incorporate aggregated predictions from previous trees.

Plot the cumulative, out-of-bag, quantile errors on the same plot.

```
figure;
plot(err);
legend('0.25 quantile error','0.5 quantile error','0.75 quantile error');
ylabel('Out-of-bag quantile error');
xlabel('Tree index');
title('Cumulative, Out-of-Bag, Quantile Regression Error')
```



All quantile error curves appear to level off after training about 50 trees. So, training 50 trees appears to be sufficient to achieve minimal quantile error for the three quantile probabilities.

- “Tune Random Forest Using Quantile Error and Bayesian Optimization” on page 18-181

## Definitions

### Out-of-Bag

In a bagged ensemble, observations are out-of-bag when they are left out of the training sample for a particular learner. Observations are in-bag when they are used to train a particular learner.

When bagging learners, a practitioner takes a bootstrap sample (that is, a random sample with replacement) of size  $n$  for each learner, and then trains the learners using their respective bootstrap samples. Drawing  $n$  out of  $n$  observations with replacement omits on average about 37% of observations for each learner.

The out-of-bag ensemble error, the ensemble error estimated using out-of-bag observations only, is an unbiased estimator of the true ensemble error.

### Quantile Regression Error

The quantile regression error of a model given observed predictor data and responses is the weighted mean absolute deviation (MAD). If the model under-predicts the response, then deviation weights are  $\tau$ , the quantile probability. If the model over-predicts, then deviation weights are  $1 - \tau$ .

That is, the  $\tau$  quantile regression error is

$$L_\tau = \tau \frac{\sum_{\{j: y_j \geq \hat{y}_{\tau,j}\}} w_j (y_j - \hat{y}_{\tau,j})}{\sum_{j=1}^n w_j} + (1 - \tau) \frac{\sum_{\{j: y_j < \hat{y}_{\tau,j}\}} w_j (\hat{y}_{\tau,j} - y_j)}{\sum_{j=1}^n w_j}.$$

$y_j$  is true response  $j$ ,  $\hat{y}_{\tau,j}$  is the  $\tau$  quantile that the model predicts, and  $w_j$  is observation weight  $j$ .

### Tip

The out-of-bag ensemble error estimator is unbiased for the true ensemble error. So, to tune parameters of a random forest, estimate the out-of-bag ensemble error instead of implementing cross-validation.

## References

- [1] Breiman, L. *Random Forests*. *Machine Learning* 45, pp. 5–32, 2001.
- [2] Meinshausen, N. “Quantile Regression Forests.” *Journal of Machine Learning Research*, Vol. 7, 2006, pp. 983–999.

## See Also

`TreeBagger` | `error` | `oobQuantilePredict` | `quantileError`

## Topics

“Tune Random Forest Using Quantile Error and Bayesian Optimization” on page 18-181

**Introduced in R2016b**

# oobQuantilePredict

**Class:** TreeBagger

Quantile predictions for out-of-bag observations from bag of regression trees

## Syntax

```
YFit = oobQuantilePredict(Mdl)
YFit = oobQuantilePredict(Mdl, Name, Value)
[YFit, YW] = oobQuantilePredict(____)
```

## Description

`YFit = oobQuantilePredict(Mdl)` returns a vector of medians of the predicted responses at all out-of-bag on page 32-4512 observations in `Mdl.X`, the predictor data, and using `Mdl`, which is a bag of regression trees. `Mdl` must be a `TreeBagger` model object and `Mdl.OOBIndices` must be nonempty.

`YFit = oobQuantilePredict(Mdl, Name, Value)` uses additional options specified by one or more `Name, Value` pair arguments. For example, specify quantile probabilities or trees to include for quantile estimation.

`[YFit, YW] = oobQuantilePredict(____)` also returns a sparse matrix of response weights on page 32-4513 using any of the previous syntaxes.

## Input Arguments

### **Mdl** — Bag of regression trees

`TreeBagger` model object (default)

Bag of regression trees, specified as a `TreeBagger` model object created by `TreeBagger`.

- The value of `Mdl.Method` must be `regression`.

- When you train `Mdl` using `TreeBagger`, you must specify the name-value pair `'OOBPrediction', 'on'`. Consequently, `TreeBagger` saves required out-of-bag observation index matrix in `Mdl.OOBIndices`.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### **Quantile** — Quantile probability

0.5 (default) | numeric vector containing values in [0,1]

Quantile probability, specified as the comma-separated pair consisting of `'Quantile'` and a numeric vector containing values in the interval [0,1]. For each observation (row) in `Mdl.X`, `TreeBagger.oobQuantilePredict` estimates corresponding quantiles for all probabilities in `Quantile`.

Example: `'Quantile', [0 0.25 0.5 0.75 1]`

Data Types: `single` | `double`

### **Trees** — Indices of trees to use in response estimation

'all' (default) | numeric vector of positive integers

Indices of trees to use in response estimation, specified as the comma-separated pair consisting of `'Trees'` and `'all'` or a numeric vector of positive integers. Indices correspond to the cells of `Mdl.Trees`; each cell therein contains a tree in the ensemble. The maximum value of `Trees` must be less than or equal to the number of trees in the ensemble (`Mdl.NumTrees`).

For `'all'`, `TreeBagger.oobQuantilePredict` uses the indices `1:Mdl.NumTrees`.

Example: `'Trees', [1 10 Mdl.NumTrees]`

Data Types: `char` | `single` | `double`

### **TreeWeights** — Weights to attribute to responses from individual trees

numeric vector of nonnegative values

Weights to attribute to responses from individual trees, specified as the comma-separated pair consisting of 'TreeWeights' and a numeric vector of `numel(trees)` nonnegative values. `trees` is the value of the `Trees` name-value pair argument.

The default is `ones(size(trees))`.

Data Types: `single` | `double`

## Output Arguments

### **YFit** — Estimated quantiles

numeric matrix

Estimated quantiles for out-of-bag observations, returned as an  $n$ -by-`numel(tau)` numeric matrix.  $n$  is the number of observations in the training data (`numel(Mdl.Y)`) and `tau` is the value of the `Quantile` name-value pair argument. That is, `YFit(j,k)` is the estimated  $100 \cdot \tau(k)$  percentile of the response distribution given `X(j,:)` and using `Mdl`.

### **YW** — Response weights

sparse matrix

Response weights on page 32-4513, returned as an  $n$ -by- $n$  sparse matrix.  $n$  is the number of responses in the training data (`numel(Mdl.Y)`). `YW(:,j)` specifies the response weights for the observation in `Mdl.X(j,:)`.

`TreeBagger.oobQuantilePredict` predicts quantiles using linear interpolation of the empirical cumulative distribution function (cdf). For a particular observation, you can use its response weights to estimate quantiles using alternative methods, such as approximating the cdf using kernel smoothing on page B-86.

## Examples

### Predict Out-of-Bag Medians Using Quantile Regression

Load the `carsmall` data set. Consider a model that predicts the fuel economy (in MPG) of a car given its engine displacement.

```
load carsmall
```

Train an ensemble of bagged regression trees using the entire data set. Specify 100 weak learners and save out-of-bag indices.

```
rng(1); % For reproducibility
Mdl = TreeBagger(100, Displacement, MPG, 'Method', 'regression', ...
    'OOBPrediction', 'on');
```

Mdl is a `TreeBagger` ensemble.

Perform quantile regression to predict the out-of-bag median fuel economy for all training observations.

```
oobMedianMPG = oobQuantilePredict(Mdl);
```

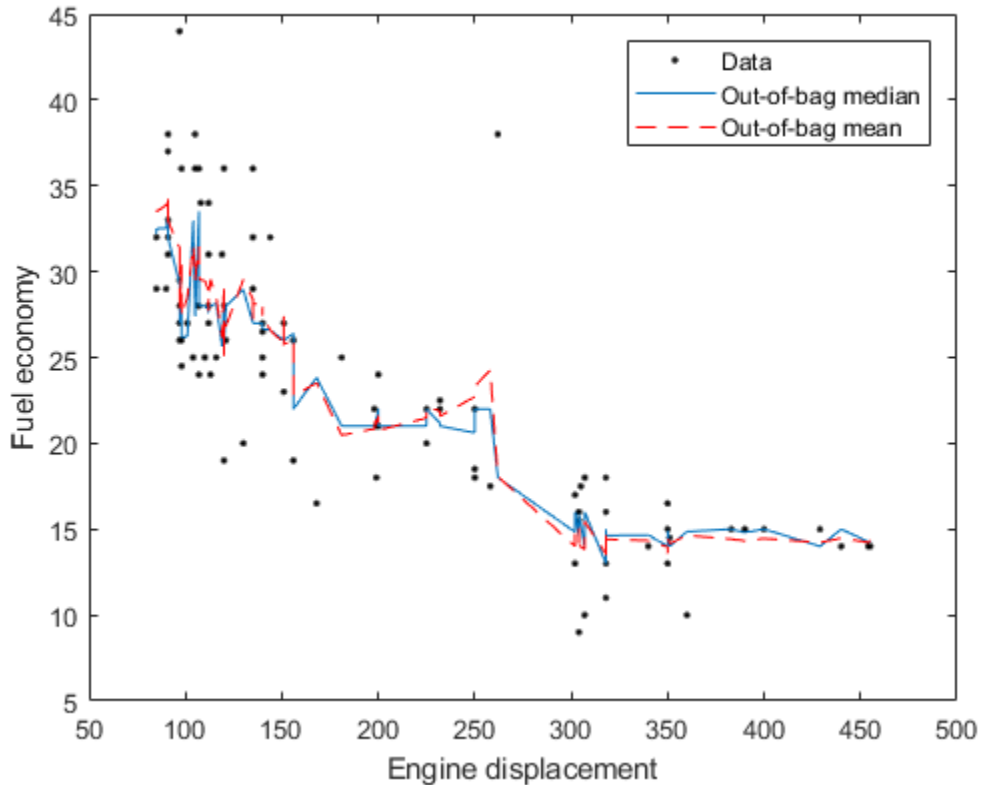
`oobMedianMPG` is an `n`-by-1 numeric vector of medians corresponding to the conditional distribution of the response given the sorted observations in `Mdl.X`. `n` is the number of observations, `size(Mdl.X, 1)`.

Sort the observations in ascending order. Plot the observations and the estimated medians on the same figure. Compare the out-of-bag median and mean responses.

```
[sX, idx] = sort(Mdl.X);
oobMeanMPG = oobPredict(Mdl);

figure;
plot(Displacement, MPG, 'k. ');
hold on
plot(sX, oobMedianMPG(idx));
plot(sX, oobMeanMPG(idx), 'r--');
ylabel('Fuel economy');
xlabel('Engine displacement');
legend('Data', 'Out-of-bag median', 'Out-of-bag mean');
hold off;
```





### Estimate Out-of-Bag Prediction Intervals Using Percentiles

Load the `carsmall` data set. Consider a model that predicts the fuel economy of a car (in MPG) given its engine displacement.

```
load carsmall
```

Train an ensemble of bagged regression trees using the entire data set. Specify 100 weak learners and save out-of-bag indices.

```
rng(1); % For reproducibility
Mdl = TreeBagger(100, Displacement, MPG, 'Method', 'regression', ...
    'OOBPrediction', 'on');
```

Perform quantile regression to predict the out-of-bag 2.5% and 97.5% percentiles.

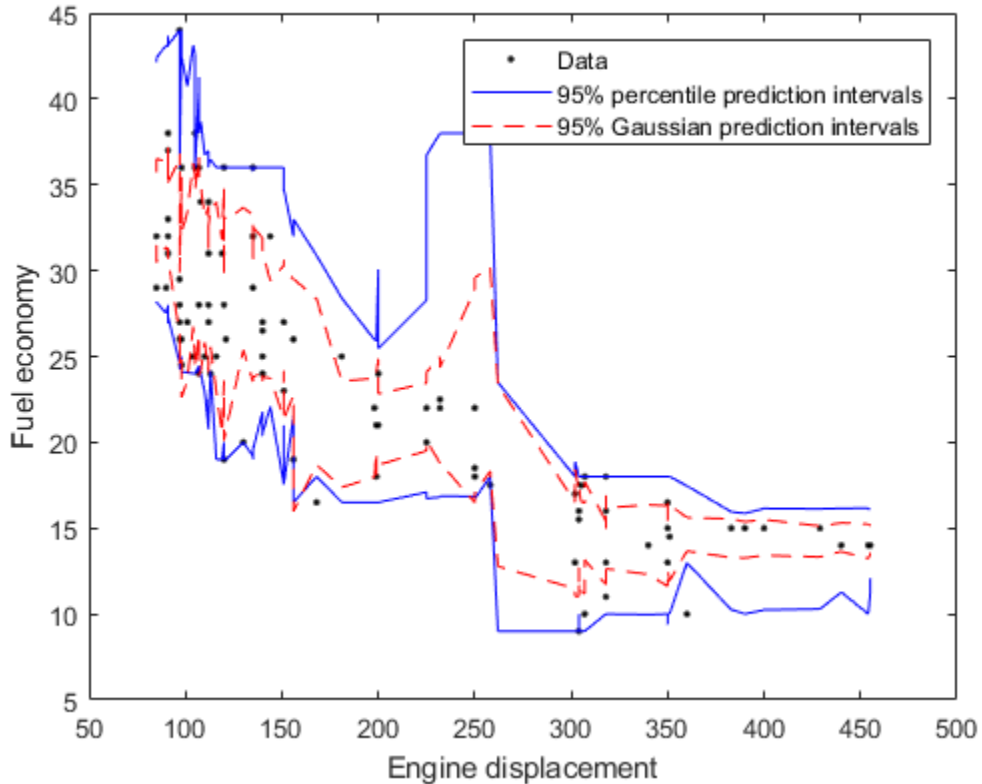
```
oobQuantPredInts = oobQuantilePredict(Mdl, 'Quantile', [0.025, 0.975]);
```

`oobQuantPredInts` is an  $n$ -by-2 numeric matrix of prediction intervals corresponding to the out-of-bag observations in `Mdl.X`.  $n$  is number of observations, `size(Mdl.X, 1)`. The first column contains the 2.5% percentiles and the second column contains the 97.5% percentiles.

Plot the observations and the estimated medians on the same figure. Compare the percentile prediction intervals and the 95% prediction intervals, assuming the conditional distribution of MPG is Gaussian.

```
[oobMeanMPG, oobSTEMeanMPG] = oobPredict(Mdl);
STDNPredInts = oobMeanMPG + [-1 1]*norminv(0.975).*oobSTEMeanMPG;
[sX, idx] = sort(Mdl.X);

figure;
h1 = plot(Displacement, MPG, 'k. ');
hold on
h2 = plot(sX, oobQuantPredInts(idx, :), 'b');
h3 = plot(sX, STDNPredInts(idx, :), 'r--');
ylabel('Fuel economy');
xlabel('Engine displacement');
legend([h1, h2(1), h3(1)], {'Data', '95% percentile prediction intervals', ...
    '95% Gaussian prediction intervals'});
hold off;
```



### Estimate Out-of-Bag Conditional Cumulative Distribution Using Quantile Regression

Load the `carsmall` data set. Consider a model that predicts the fuel economy of a car (in MPG) given its engine displacement.

```
load carsmall
```

Train an ensemble of bagged regression trees using the entire data set. Specify 100 weak learners and save the out-of-bag indices.

```
rng(1); % For reproducibility
Mdl = TreeBagger(100, Displacement, MPG, 'Method', 'regression', ...
    'OOBPrediction', 'on');
```

Estimate the out-of-bag response weights.

```
[~,YW] = oobQuantilePredict(Mdl);
```

`YW` is an  $n$ -by- $n$  sparse matrix containing the response weights.  $n$  is the number of training observations, `numel(Y)`. The response weights for the observation in `Mdl.X(j,:)` are in `YW(:,j)`. Response weights are independent of any specified quantile probabilities.

Estimate the out-of-bag, conditional cumulative distribution function (ccdf) of the responses by:

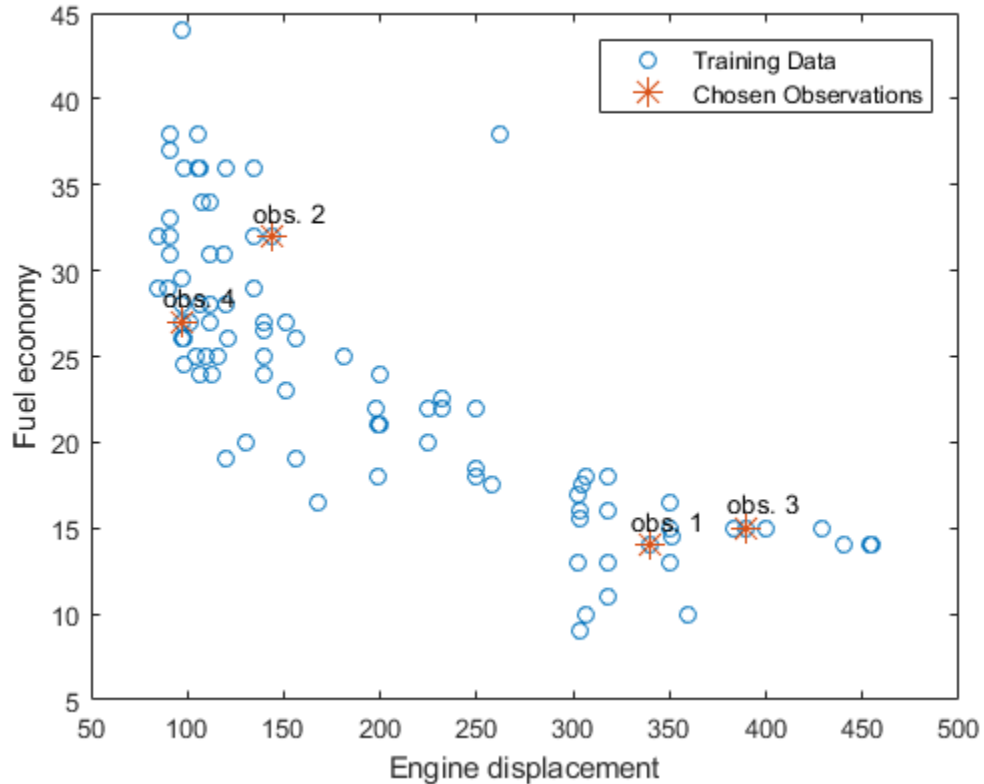
- 1 Sorting the responses in ascending order, and then sorting the response weights using the indices induced by sorting the responses.
- 2 Computing the cumulative sums over each column of the sorted response weights.

```
[sortY,sortIdx] = sort(Mdl.Y);  
cpdf = full(YW(sortIdx,:));  
ccdf = cumsum(cpdf);
```

`ccdf(:,j)` is the empirical out-of-bag ccdf of the response, given observation  $j$ .

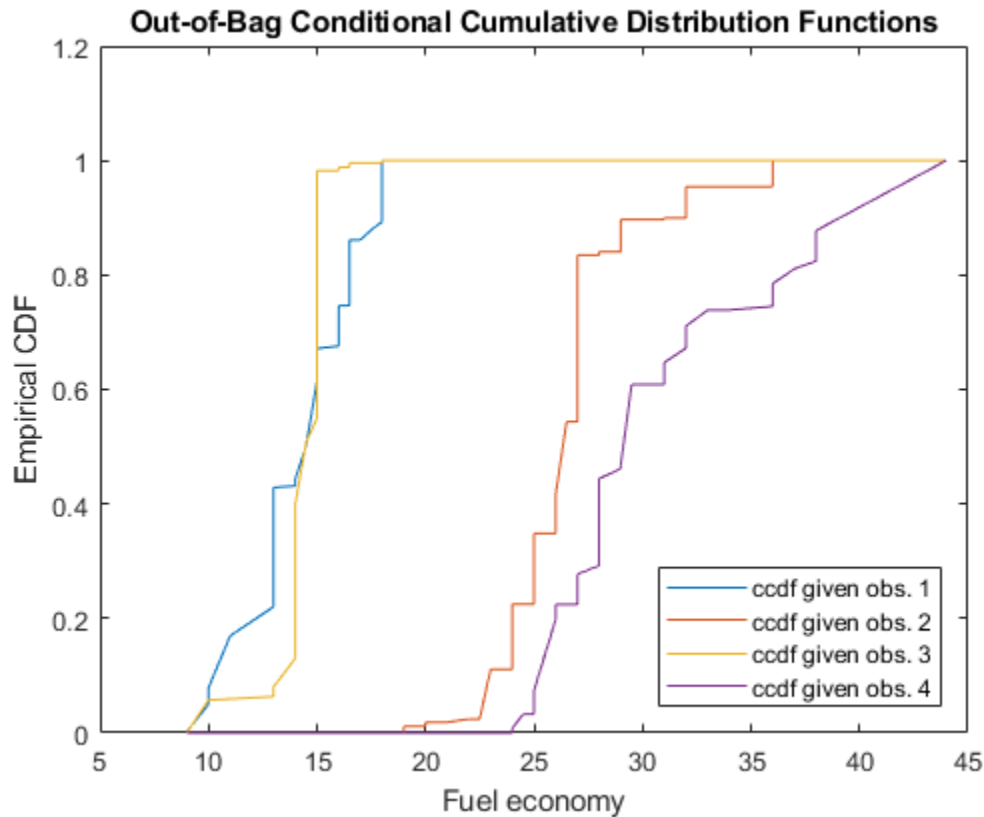
Choose a random sample of four training observations. Plot the training sample and identify the chosen observations.

```
[randX,idx] = datasample(Mdl.X,4);  
figure;  
plot(Mdl.X,Mdl.Y,'o');  
hold on  
plot(randX,Mdl.Y(idx),'*', 'MarkerSize',10);  
text(randX-10,Mdl.Y(idx)+1.5,{'obs. 1' 'obs. 2' 'obs. 3' 'obs. 4'});  
legend('Training Data','Chosen Observations');  
xlabel('Engine displacement')  
ylabel('Fuel economy')  
hold off
```



Plot the out-of-bag ccdf for the four chosen responses in the same figure.

```
figure;
plot(sortY,ccdf(:,idx));
legend('ccdf given obs. 1','ccdf given obs. 2',...
       'ccdf given obs. 3','ccdf given obs. 4',...
       'Location','SouthEast')
title('Out-of-Bag Conditional Cumulative Distribution Functions')
xlabel('Fuel economy')
ylabel('Empirical CDF')
```



- “Conditional Quantile Estimation Using Kernel Smoothing” on page 18-177

## Definitions

### Out-of-Bag

In a bagged ensemble, observations are out-of-bag when they are left out of the training sample for a particular learner. Observations are in-bag when they are used to train a particular learner.

When bagging learners, a practitioner takes a bootstrap sample (that is, a random sample with replacement) of size  $n$  for each learner, and then trains the learners using

their respective bootstrap samples. Drawing  $n$  out of  $n$  observations with replacement omits on average about 37% of observations for each learner.

The out-of-bag ensemble error, the ensemble error estimated using out-of-bag observations only, is an unbiased estimator of the true ensemble error.

## Quantile Random Forest

Quantile random forest [2] is a quantile-regression method that uses a random forest [1] of regression trees to model the conditional distribution of a response variable, given the value of predictor variables. You can use a fitted model to estimate quantiles in the conditional distribution of the response.

Besides quantile estimation, you can use quantile regression to estimate prediction intervals or detect outliers. For example:

- To estimate 95% quantile prediction intervals, estimate the 0.025 and 0.975 quantiles.
- To detect outliers, estimate the 0.01 and 0.99 quantiles. All observations smaller than the 0.01 quantile and larger than the 0.99 quantile are outliers. All observations that are outside the interval  $[L, U]$  can be considered outliers:

$$L = Q_1 - 1.5 * IQR$$

and

$$U = Q_3 + 1.5 * IQR,$$

where:

- $Q_1$  is the 0.25 quantile.
- $Q_3$  is the 0.75 quantile.
- $IQR = Q_3 - Q_1$  (the interquartile range).

## Response Weights

Response weights are scalars that represent the conditional distribution of the response given a value in the predictor space. The observations in the bootstrap samples and the leaves that the training and test observations share induce response weights.

Given the observation  $x$ , the response weight for observation  $j$  in the training sample using tree  $t$  in the ensemble is

$$w_{tj}(x) = \frac{I\{X_j \in S_t(x)\}}{\sum_{k=1}^{n_{\text{train}}} I\{X_k \in S_t(x)\}},$$

where:

- $I\{h\}$  is the indicator function.
- $S_t(x)$  is the leaf of tree  $t$  containing  $x$ .
- $n_{\text{train}}$  is the number of training observations.

In other words, the response weights of a particular tree form the conditional relative frequency distribution of the response.

The response weights for the entire ensemble are averaged over the trees:

$$w_j^*(x) = \frac{1}{T} \sum_{t=1}^T w_{tj}(x).$$

## Algorithms

`TreeBagger.oobQuantilePredict` estimates out-of-bag quantiles by applying `quantilePredict` to all observations in the training data (`Mdl.X`). For each observation, the method uses only the trees for which the observation is out-of-bag.

For observations that are in-bag for all trees in the ensemble, `TreeBagger.oobQuantilePredict` assigns the sample quantile of the response data. In other words, `TreeBagger.oobQuantilePredict` does not use quantile regression for out-of-bag observations. Instead, it assigns `quantile(Mdl.Y, tau)`, where `tau` is the value of the `Quantile` name-value pair argument.

## References

- [1] Meinshausen, N. “Quantile Regression Forests.” *Journal of Machine Learning Research*, Vol. 7, 2006, pp. 983–999.
- [2] Breiman, L. “Random Forests.” *Machine Learning*. Vol. 45, 2001, pp. 5–32.



## See Also

`TreeBagger` | `oobQuantileError` | `predict` | `quantilePredict`

## Topics

“Conditional Quantile Estimation Using Kernel Smoothing” on page 18-177

**Introduced in R2016b**

## optimizableVariable

Variable description for `bayesopt` or other optimizers

### Description

Create variables for optimizers.

### Creation

```
variable = optimizableVariable(Name, Range)
variable = optimizableVariable(Name, Range, Name, Value)
```

### Description

`variable = optimizableVariable(Name, Range)` creates a variable with the specified name and range of values.

`variable = optimizableVariable(Name, Range, Name, Value)` sets properties on page 32-4516 using `Name, Value` pairs. You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

### Properties

Set the properties of an `optimizableVariable` object using name-value pairs during construction.

#### **Name** — Variable name

character vector

Variable name, specified as a character vector. The name must be unique, meaning different from those of other variables in the optimization.

---

#### Note

- There are two names associated with an `optimizableVariable`:
  - The MATLAB workspace variable name
  - The name of the variable in the optimization

For example,

```
xvar = optimizableVariable('spacevar', [1,100]);
```

`xvar` is the MATLAB workspace variable, and `'spacevar'` is the variable in the optimization.

Use these names as follows:

- Use `xvar` as an element in the vector of variables you pass to `bayesopt`. For example,

```
results = bayesopt(fun, [xvar, tvar])
```

- Use `'spacevar'` as the name of the variable in the optimization. For example, in an objective function,

```
function objective = mysvmfun(x, cdata, grp)
SVMModel = fitsvm(cdata, grp, 'KernelFunction', 'rbf', ...
    'BoxConstraint', x.spacevar, ...
    'KernelScale', x.tvar);
objective = kfoldLoss(crossval(SVMModel));
```

---

Example: 'X1'

Data Types: char

### Range — Variable range

2-element increasing real vector | cell array of names of categorical variables

Variable range, specified as a 2-element finite increasing real vector, or as a cell array of names of categorical variables:

- For real or integer variables, Range gives the lower bound and upper bound of that variable.
- For categorical variables, Range gives the possible values.

Example: [-10,1]

Example: {'red','blue','black'}

Data Types: double | cell

**Type — Variable type**

'real' (default) | 'integer' | 'categorical'

Variable type, specified as 'real' (real variable), 'integer' (integer variable), or 'categorical' (categorical variable).

---

**Note** The MATLAB data type of both 'real' and 'integer' variables is the standard double-precision floating point number. The data type of 'categorical' variables is categorical. So, for example, to read a value of a categorical variable named 'colorv' in a table of variables named `x`, use the command `char(x.colorv)`. For an example, see the objective function in “Custom Output Functions” on page 10-23.

---

Example: 'categorical'

Data Types: char

**Transform — Transform applied to variable**

'none' (default) | 'log'

Transform applied to variable, specified as 'none' (no transform) or 'log' (logarithmic transform).

For 'log', the variable must be 'real' or 'integer' and positive. The variable is searched and modeled on a log scale.

Example: 'log'

Data Types: char

**Optimize — Indication to use variable in optimization**

true (default) | false

Indication to use variable in optimization, specified as true (use the variable) or false (do not use the variable).

Example: false

Data Types: logical

---

**Note** You can use dot notation to change the following properties after construction.

- Range of real or integer variables. For example,

```
xvar = optimizableVariable('x', [-10, 10]);  
% Modify the range:  
xvar.Range = [1, 5];
```

- Type between 'integer' and 'real'. For example,

```
xvar.Type = 'integer';
```

- Transform of real or integer variables between 'log' and 'none'. For example,

```
xvar.Transform = 'log';
```

You can use this flexibility, for example, to tweak an optimization that you want to continue. Update the range or transform using dot notation and then call `resume`.

---

## Object Functions

`bayesopt` Select optimal machine learning hyperparameters using Bayesian optimization

## Examples

### Variables for Optimization Examples

Real variable from 0 to 1:

```
var1 = optimizableVariable('xvar', [0 1])
```

```
var1 =
```

```
optimizableVariable with properties:
```

```
    Name: 'xvar'  
    Range: [0 1]
```

```
    Type: 'real'  
  Transform: 'none'  
  Optimize: 1
```

Integer variable from 1 to 1000 on a log scale:

```
var2 = optimizableVariable('ivar',[1 1000],'Type','integer','Transform','log')
```

```
var2 =
```

```
  optimizableVariable with properties:
```

```
    Name: 'ivar'  
   Range: [1 1000]  
    Type: 'integer'  
  Transform: 'log'  
  Optimize: 1
```

Categorical variable of rainbow colors:

```
var3 = optimizableVariable('rvar',{'r' 'o' 'y' 'g' 'b' 'i' 'v'},'Type','categorical')
```

```
var3 =
```

```
  optimizableVariable with properties:
```

```
    Name: 'rvar'  
   Range: {'r' 'o' 'y' 'g' 'b' 'i' 'v'}  
    Type: 'categorical'  
  Transform: 'none'  
  Optimize: 1
```

- “Optimize a Cross-Validated SVM Classifier Using Bayesian Optimization” on page 10-53
- “Bayesian Optimization Using bayesopt” on page 10-32

## See Also

BayesianOptimization | bayesopt

## Topics

“Optimize a Cross-Validated SVM Classifier Using Bayesian Optimization” on page 10-53

“Bayesian Optimization Using bayesopt” on page 10-32

**Introduced in R2016b**

## ordinal

Create ordinal array

After creating an `ordinal` array, you can use related functions to add, drop, or merge categories, and more.

For more information, see `ordinal`.

---

**Note** The `nominal` and `ordinal` array data types might be removed in a future release. To represent ordered and unordered discrete, nonnumeric data, use the MATLAB® `categorical` data type instead.

---

## Syntax

```
B = ordinal(X)
B = ordinal(X, labels)
B = ordinal(X, labels, levels)

B = ordinal(X, labels, [], edges)
```

## Description

`B = ordinal(X)` creates an ordinal array object `B` from the array `X`. `ordinal` creates the levels of `B` from the sorted unique values in `X`, and creates default labels for them.

`B = ordinal(X, labels)` labels the levels in `B` according to `labels`.

`B = ordinal(X, labels, levels)` creates an ordinal array with possible levels defined by `levels`.

`B = ordinal(X, labels, [], edges)` creates an ordinal array by binning a numeric array `X` with bin edges given by the numeric vector `edges`.



## Examples

### Create and Label Ordinal Arrays

Create an ordinal array from integer data, providing explicit labels.

```
quality = ordinal([1 2 3 3 2 1 2 1 3], ...
                  {'low' 'medium' 'high'})
```

```
quality =
```

```
1x9 ordinal array
```

```
Columns 1 through 7
```

```
    low    medium    high    high    medium    low    medium
```

```
Columns 8 through 9
```

```
    low    high
```

Show that the first element is less than the second element (low is less than medium).

```
quality(1) < quality(2)
```

```
ans =
```

```
logical
```

```
1
```

Create an ordinal array by binning values between 0 and 1 into thirds with labels 'small', 'medium', and 'large'.

```
X = rand(5,2)
```

```
A = ordinal(X, {'small' 'medium' 'large'}, [], [0 1/3 2/3 1])
```

```
X =
```

```
0.8147    0.0975
0.9058    0.2785
0.1270    0.5469
0.9134    0.9575
0.6324    0.9649
```

A =

```
5x2 ordinal array
```

```
large      small
large      small
small      medium
large      large
medium     large
```

- “Create Nominal and Ordinal Arrays” on page 2-4

## Input Arguments

### **x** — Input array

numeric | logical | character | categorical | cell array of character vectors

Input array to convert to `ordinal`, specified as a numeric, logical, character, or categorical array, or a cell array of character vectors. The levels of the resulting `ordinal` array correspond to the sorted unique values in `X`.

### **labels** — Labels for the discrete levels

character array | cell array of character vectors

Labels for the discrete levels, specified as a character array or cell array of character vectors. By default, `ordinal` assigns the labels to the levels in `B` in order according to the sorted unique values in `X`.

You can include duplicate labels in `labels` in order to merge multiple values in `X` into a single level in `B`.

Data Types: `char` | `cell`

**levels — Possible ordinal levels**

vector

Possible ordinal levels for the output `ordinal` array, specified as a vector whose values can be compared to those in `X` using the equality operator. `ordinal` assigns labels to each level from the corresponding elements of `labels`. If `X` contains any values not present in `levels`, the levels of the corresponding elements of `B` are undefined.

**edges — Bin edges**

numeric vector

Bin edges to create a ordinal array by binning a numeric array, specified as a numeric vector. The uppermost bin includes values equal to the right-most edge. `ordinal` assigns labels to each level in the resulting nominal array from the corresponding elements of `labels`. When you specify `edges`, it must have one more element than `labels`.

## Output Arguments

**B — Ordinal array**

ordinal array object

Nominal array, returned as an ordinal array object.

By default, an element of `B` is undefined if the corresponding element of `X` is `NaN` (when `X` is numeric), an empty character vector (when `X` is a character), or undefined (when `X` is categorical). `nominal` treats such elements as “undefined” or “missing” and does not include entries for them among the possible levels. To create an explicit level for such elements instead of treating them as undefined, you must use the `levels` input argument, and include `NaN`, the empty character vector, or an undefined element.

## See Also

`nominal`

## Topics

“Create Nominal and Ordinal Arrays” on page 2-4

`ordinal`

**Introduced in R2007a**

# ordinal

Arrays for ordinal data

## Description

Ordinal data are discrete, nonnumeric values that have a natural ordering. `ordinal` array objects provide efficient storage and convenient manipulation of such data, while also maintaining meaningful labels for the values.

You can manipulate `ordinal` arrays much like ordinary numeric arrays, including subscripting, concatenating, and reshaping. It can be useful to use `ordinal` arrays as grouping variables when the elements indicate the group an observation belongs to.

---

**Note** The `nominal` and `ordinal` array data types might be removed in a future release. To represent ordered and unordered discrete, nonnumeric data, use the MATLAB `categorical` data type instead.

---

## Creation

Create `ordinal` arrays using the `ordinal` function.

## Properties

### `labels` — Level labels

cell array of character vectors

This property is read-only.

Level labels, specified as a cell array of character vectors. Access labels using `getlabels`.

Data Types: `cell`

**undeflabel** — Label for undefined levels

'&lt;undefined&gt;' (default)

This property is read-only.

Label for undefined levels, specified as '<undefined>'. You can find undefined elements in categorical arrays using `isundefined`.

## Object Functions

<code>addlevels</code>	Add levels to nominal or ordinal arrays
<code>droplevels</code>	Drop levels from a nominal or ordinal array
<code>getlabels</code>	Access categorical array labels
<code>getlevels</code>	Access categorical array levels
<code>islevel</code>	Determine if levels are in nominal or ordinal array
<code>levelcounts</code>	Element counts by level of a nominal or ordinal array
<code>mergelevels</code>	Merge levels of nominal or ordinal arrays
<code>reorderlevels</code>	Reorder levels of nominal or ordinal arrays
<code>setlabels</code>	Assign labels to levels of nominal or ordinal arrays

You can also use many other MATLAB array functions with categorical arrays. The following is a partial list. For a complete list, see “Other MATLAB Functions Supporting Nominal and Ordinal Arrays” on page 2-3.

<code>double</code>	Double-precision arrays
<code>histogram</code>	Histogram plot
<code>isequal</code>	Determine array equality
<code>isundefined</code>	Find undefined elements in categorical array
<code>pie</code>	Pie chart
<code>summary</code>	Print summary of table, timetable, or categorical array
<code>times</code>	Element-wise multiplication

## Examples

### Create and Manipulate Ordinal Arrays

Create an ordinal array from integer data.

```
quality = ordinal([1 2 3; 3 2 1; 2 1 3], {'low' 'medium' 'high'})
```

```
quality =  
  
3x3 ordinal array  
  
    low      medium    high  
    high     medium    low  
    medium   low       high
```

Identify the elements in `quality` that are members of a level that is greater than or equal to 'medium'. A value of 1 in the resulting array indicates that the corresponding element of `quality` is in this category.

```
quality >= 'medium'
```

```
ans =  
  
3x3 logical array  
  
    0     1     1  
    1     1     0  
    1     0     1
```

Identify the elements of `quality` that are members of either 'low' or 'high'.

```
ismember(quality,{'low' 'high'})
```

```
ans =  
  
3x3 logical array  
  
    1     0     1  
    1     0     1  
    0     1     1
```

Merge the elements of the 'medium' and 'high' levels into a new level labeled 'ok'.

```
quality = mergelevels(quality,{'medium','high'},'ok')
```

```
quality =
```

```
3x3 ordinal array  
  
  low      ok      ok  
  ok       ok      low  
  ok       low     ok
```

Display the levels of quality.

```
getlevels(quality)
```

```
ans =
```

```
1x2 ordinal array  
  
  low      ok
```

Summarize the number of elements in each level. By default, `summary` returns counts for each column of the input array.

```
summary(quality)  
  
  low      1      1      1  
  ok       2      2      2
```

- “Create Nominal and Ordinal Arrays” on page 2-4
- “Reorder Category Levels” on page 2-12
- “Categorize Numeric Data” on page 2-18
- “Merge Category Levels” on page 2-22
- “Sort Ordinal Arrays” on page 2-46

## See Also

`nominal`

## Topics

“Create Nominal and Ordinal Arrays” on page 2-4  
“Reorder Category Levels” on page 2-12



- “Categorize Numeric Data” on page 2-18
- “Merge Category Levels” on page 2-22
- “Sort Ordinal Arrays” on page 2-46
- “Advantages of Using Categorical Arrays” on page 2-50
- “Index and Search Using Categorical Arrays” on page 2-53
- “Grouping Variables” on page 2-59

## outlierMeasure

**Class:** CompactTreeBagger

Outlier measure for data

### Syntax

```
out = outlierMeasure(B,X)
out = outlierMeasure(B,X,'param1',val1,'param2',val2,...)
```

### Description

`out = outlierMeasure(B,X)` computes outlier measures for predictors `X` using trees in the ensemble `B`. The method computes the outlier measure for a given observation by taking an inverse of the average squared proximity between this observation and other observations. `outlierMeasure` then normalizes these outlier measures by subtracting the median of their distribution, taking the absolute value of this difference, and dividing by the median absolute deviation. A high value of the outlier measure indicates that this observation is an outlier.

You can supply the proximity matrix directly by using the `'Data'` parameter.

`out = outlierMeasure(B,X,'param1',val1,'param2',val2,...)` specifies optional parameter name/value pairs:

<code>'Data'</code>	Flag indicating how to treat the <code>X</code> input argument. If set to <code>'predictors'</code> (default), the method assumes <code>X</code> is a matrix of predictors and uses it for computation of the proximity matrix. If set to <code>'proximity'</code> , the method treats <code>X</code> as a proximity matrix returned by the <code>proximity</code> method. If you do not supply the proximity matrix, <code>outlierMeasure</code> computes it internally. If you use the <code>proximity</code> method to compute a proximity matrix, supplying it as input to <code>outlierMeasure</code> reduces computing time.
---------------------	--

'Labels'

Vector of true class labels. True class labels can be either a numeric vector, character matrix, or cell array of character vectors. When you supply this parameter, the method performs the outlier calculation for any observations using only other observations from the same class. This parameter must specify one label for each observation (row) in X.

## See Also

`proximity`

## OutlierMeasure property

**Class:** TreeBagger

Measure for determining outliers

### Description

The `OutlierMeasure` property is a numeric array of size `Nobs-by-1`, where `Nobs` is the number of observations in the training data, containing outlier measures for each observation.

### See Also

`CompactTreeBagger.outlierMeasure`

# parallelcoords

Parallel coordinates plot

## Syntax

```
parallelcoords(x)  
parallelcoords(x, Name, Value)
```

```
parallelcoords(ax, ___)
```

```
h = parallelcoords(___)
```

## Description

`parallelcoords(x)` creates a parallel coordinates plot of the multivariate data in the matrix `x`. Use a parallel coordinates plot to visualize high dimensional data, where each observation is represented by the sequence of its coordinate values plotted against their coordinate indices.

`parallelcoords(x, Name, Value)` creates a parallel coordinates plot with additional options specified by one or more `Name, Value` pair arguments. For example, you can standardize the data in `x` or label the coordinate tick marks along the horizontal axis of the plot.

`parallelcoords(ax, ___)` creates a parallel coordinates plot using the axes specified by the axes graphic object `ax`, using any of the previous syntaxes.

`h = parallelcoords(___)` returns a column vector of handles to the `Line` objects created by `parallelcoords`, with one handle for each row of `x`.

## Examples

### Parallel Coordinates Plot for Grouped Data

Load the Fisher iris sample data.

```
load fisheriris
```

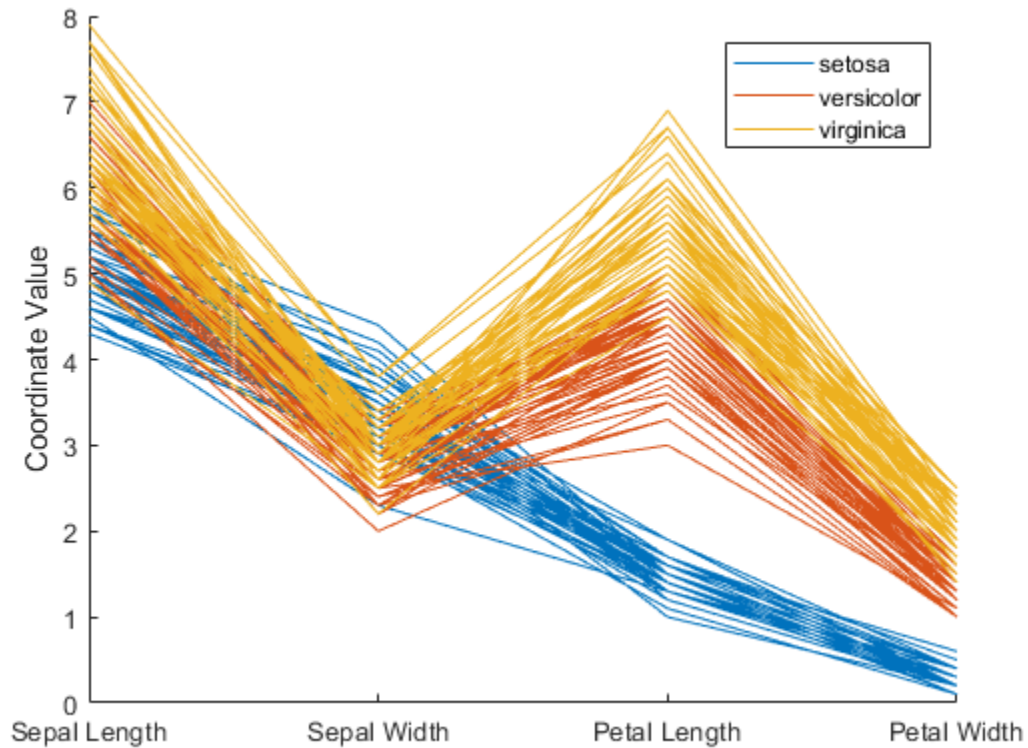
The data contains four measurements (sepal length, sepal width, petal length, and petal width) from three species of iris flowers. The matrix `meas` contains all four measurements for each of 150 flowers. The cell array `species` contains the species name for each of the 150 flowers.

Create a cell array that contains the name of each measurement variable in the sample data.

```
labels = {'Sepal Length', 'Sepal Width', 'Petal Length', 'Petal Width'};
```

Create a parallel coordinate plot using the measurement data in `meas`. Use a different color for each group as identified in `species`, and label the horizontal axis using the variable names.

```
parallelcoords(meas, 'Group', species, 'Labels', labels)
```



The resulting plot contains one line for each observation (flower). The color of each line indicates the flower species.

### Parallel Coordinates Plot with Quantile Values

Load the Fisher iris sample data.

```
load fisheriris
```

The data contains four measurements (sepal length, sepal width, petal length, and petal width) from three species of iris flowers. The matrix `meas` contains all four

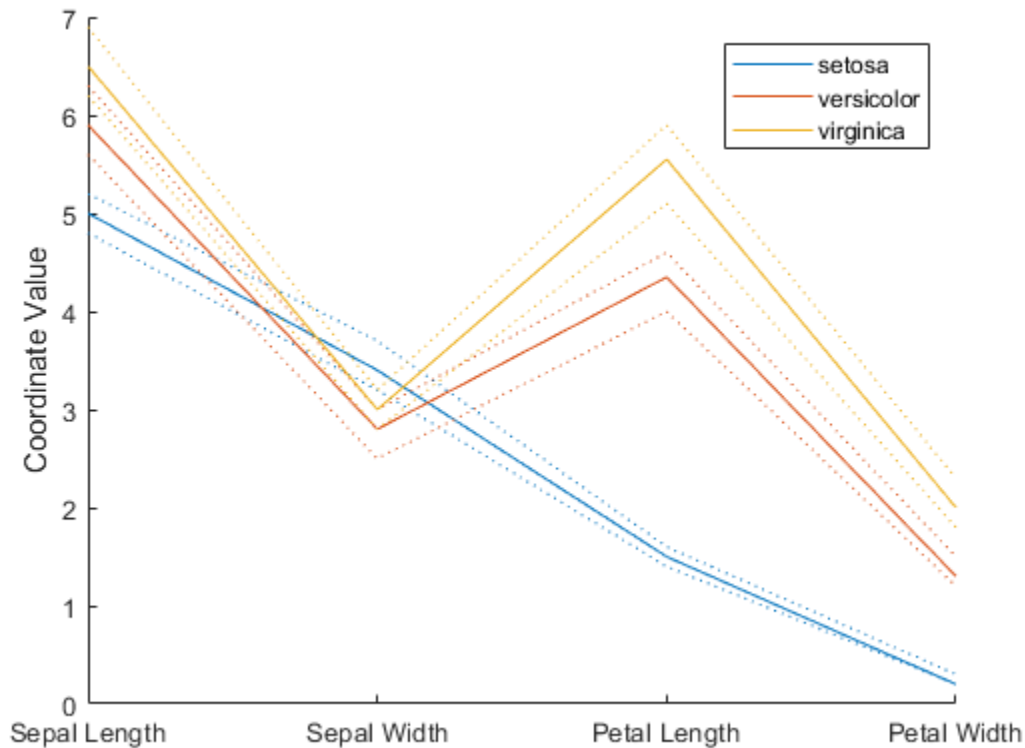
measurements for each of 150 flowers. The cell array `species` contains the species name for each of the 150 flowers.

Create a cell array that contains the name of each measurement variable in the sample data.

```
labels = {'Sepal Length', 'Sepal Width', 'Petal Length', 'Petal Width'};
```

Create a parallel coordinates plot using the measurement data in `meas`. Plot only the median, 25 percent, and 75 percent quartile values for each group identified in `species`. Label the horizontal axis using the variable names.

```
parallelcoords(meas, 'group', species, 'labels', labels, ...  
              'quantile', .25)
```





The plot shows the median values for each group as a solid line and the quartile values as dotted lines of the same color. For example, the solid blue line shows the median value measured for each variable on *setosa* irises. The dotted blue line below the solid blue line shows the 25th percentile of measurements for each variable on *setosa* irises. The dotted blue line above the solid blue line shows the 75th percentile of measurements for each variable on *setosa* irises.

### Adjust Line Properties in Parallel Coordinates Plot

Load the Fisher iris sample data.

```
load fisheriris
```

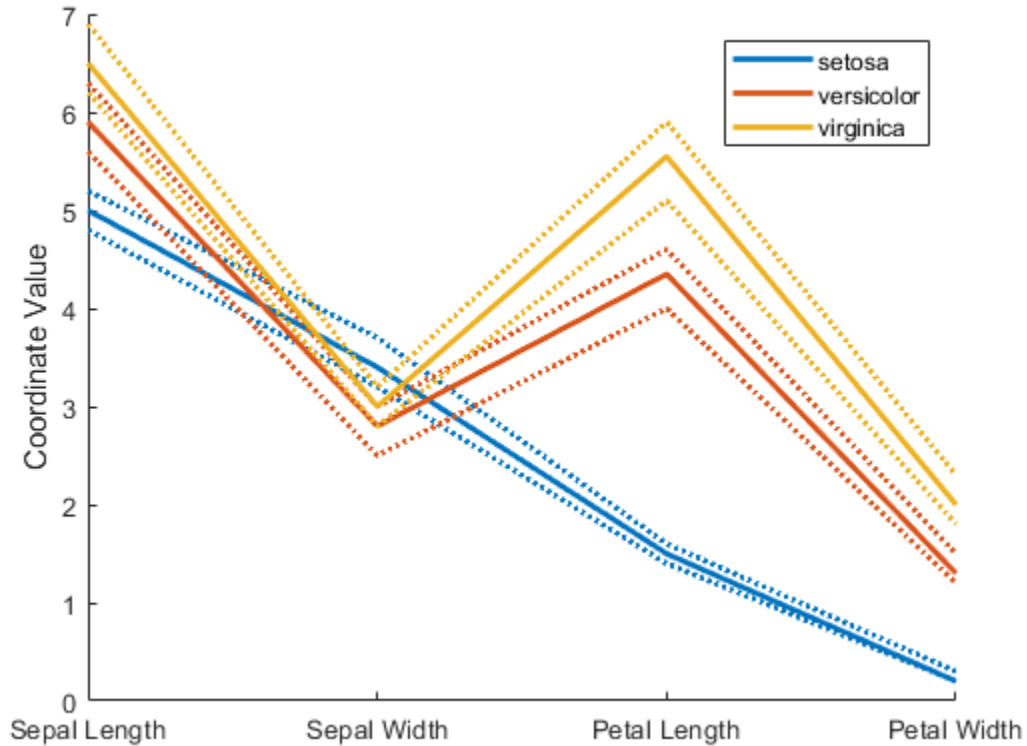
The data contains four measurements (sepal length, sepal width, petal length, and petal width) from three species of iris flowers. The matrix `meas` contains all four measurements for each of 150 flowers. The cell array `species` contains the species name for each of the 150 flowers.

Create a cell array that contains the name of each measurement variable in the sample data.

```
labels = {'Sepal Length', 'Sepal Width', 'Petal Length', 'Petal Width'};
```

Create a parallel coordinates plot using the measurement data in `meas`. Plot only the median, 25 percent, and 75 percent quartile values for each group identified in `species`. Label the horizontal axis using the variable names. Set the line width to 2.

```
parallelcoords(meas, 'group', species, 'labels', labels, ...  
              'quantile', .25, 'LineWidth', 2)
```



Specifying 'LineWidth' in this way sets the width of every line in the plot to 2.

Recreate the parallel coordinates plot, but this time, use handles to increase the width of only the line representing the median value for each measurement made on irises in the `setosa` group.

```
h = parallelcoords(meas, 'group', species, 'labels', labels, ...
                  'quantile', .25)
```

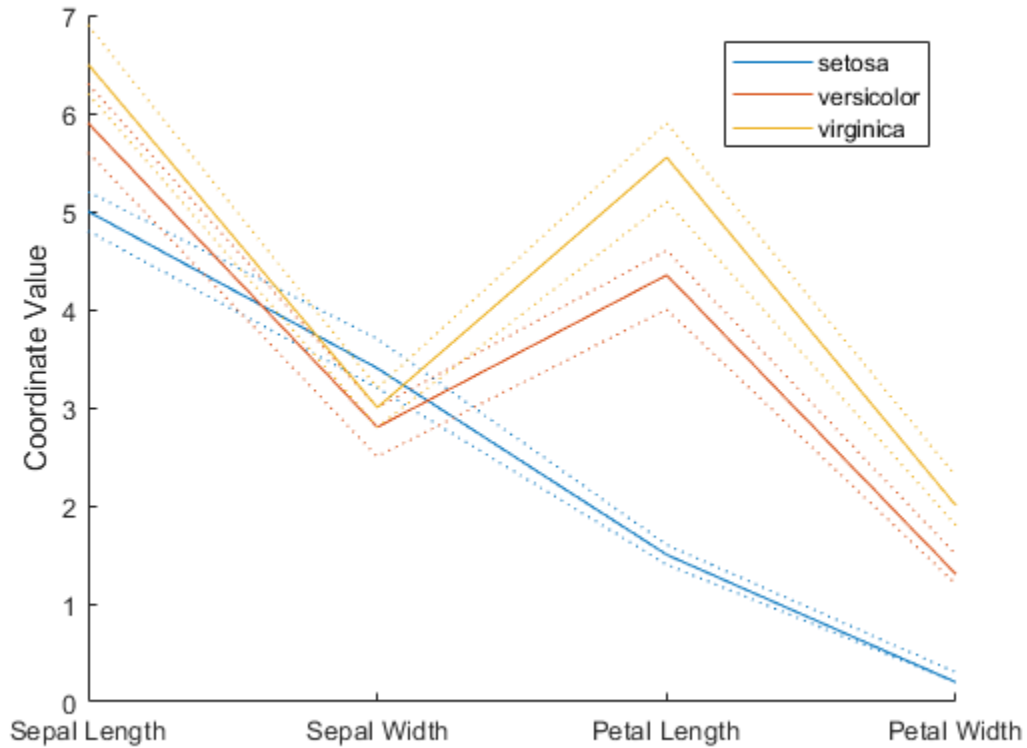
```
h =
```

```
9x1 Line array:
```

```

Line (median)
Line (lower quantile)
Line (upper quantile)
Line (median)
Line (lower quantile)
Line (upper quantile)
Line (median)
Line (lower quantile)
Line (upper quantile)

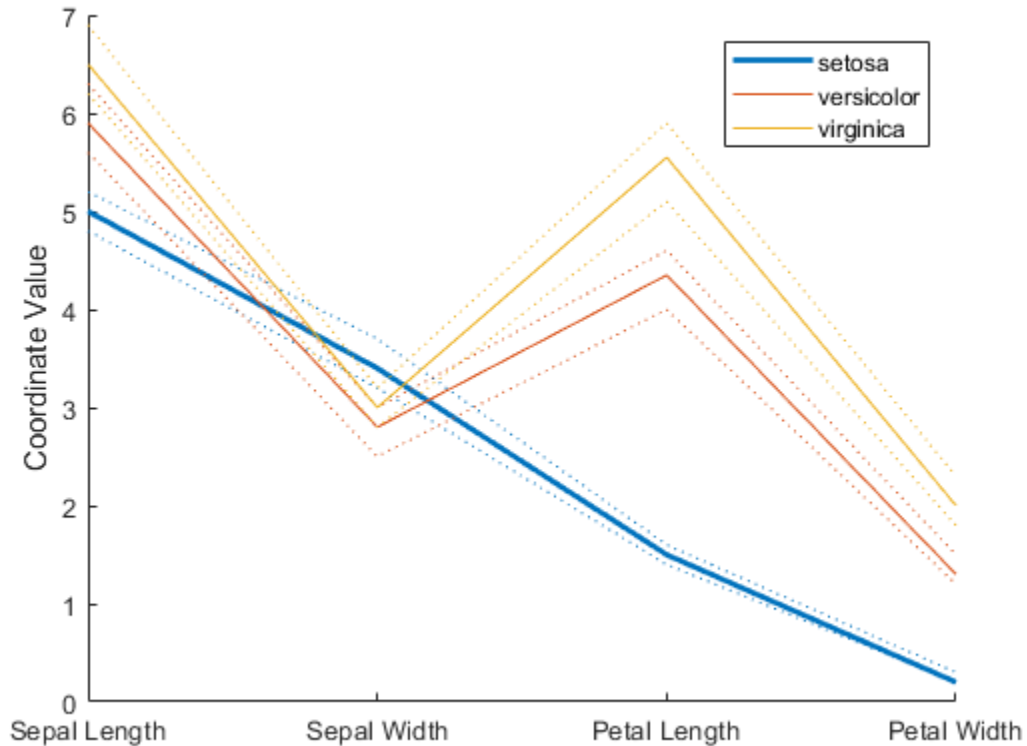
```



The returned column vector `h` contains handles that correspond to each line object created by `parallelcoords`. For example, `h(1)` corresponds to the median line for the first grouping variable (`setosa`).

Use dot notation to increase the width of the line showing the median value for each measurement made on irises in the `setosa` group.

```
h(1).LineWidth = 2;
```



## Input Arguments

**x** — Multivariate input data  
numeric matrix

Multivariate input data, specified as an  $n$ -by- $p$  matrix of numeric values.  $n$  is the number of rows of  $x$ , and each row corresponds to an observation in  $x$ .  $p$  is the number of columns in  $x$ , and each column corresponds to a variable in  $x$ .

`parallelcoords` treats NaN values in  $x$  as missing values and does not plot those coordinate values.

Data Types: `single` | `double`

### **ax — Axes for plot**

axes graphic object

Axes for plot, specified as an axes graphic object. If you do not specify `ax`, then `parallelcoords` creates the plot using the current axis. For more information on creating an axes graphic object, see `axes` and `Axes`.

## **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'Group', species, 'Quantile', .25` plots the median, 25 percent, and 75 percent quartile values for the input data, using a different color for each group identified in the variable `species`.

### **Group — Grouping variable for input data**

numeric array | categorical variable | character matrix | cell array

Grouping variable for input data, specified as the comma-separated pair consisting of `'Group'` and a numeric array containing a group index for each observation. Alternatively, the array can be a categorical variable, character matrix, or cell array containing a group name for each observation.

Data Types: `single` | `double` | `categorical` | `char` | `cell`

### **Labels — Horizontal axis labels**

character array | cell array

Horizontal axis labels, specified as the comma-separated pair consisting of `'Labels'` and a character array or cell array containing the label names.

Example: `'Labels', {'Sepal Width', 'Sepal Length'}`

Data Types: `char` | `cell`

### **Quantile** — Quantiles of input data to plot

numeric value in the range (0,1)

Quantiles of input data to plot, specified as the comma-separated pair consisting of 'Quantile' and a numeric value in the range (0,1). If you specify a value *alpha* for 'Quantile', then `parallelcoords` plots only the median, *alpha*, and  $1 - \alpha$  quantiles for each of the variables (columns) in `x`.

The quantile plot option provides a useful summary of the data when `x` contains many observations.

Example: `'Quantile', .25`

Data Types: `single` | `double`

### **Standardize** — Method to standardize input data

`'on'` | `'PCA'` | `'PCAStd'`

Method to standardize input data, specified as the comma-separated pair consisting of 'Standardize' and one of the following.

<code>'on'</code>	Scale each column of <code>x</code> to have a mean equal to 0 and a standard deviation equal to 1 before plotting.
<code>'PCA'</code>	Create plot from the principal component scores of <code>x</code> , in order of decreasing eigenvalues. <code>parallelcoords</code> removes rows of <code>x</code> containing missing values (NaN) for PCA standardization.
<code>'PCAStd'</code>	Create plot using the standardized principal component scores.

Example: `'Standardize', 'on'`

- You can modify certain aspects of the plot lines by specifying a property name and value for any of the properties listed in Primitive Line. However, this approach applies the modification to all the lines in the plot. To modify only certain plot lines, use the syntax that returns graphics handles and use dot notation to adjust each line property individually. For an illustration, see “Adjust Line Properties in Parallel Coordinates Plot” on page 32-4539.

## Output Arguments

### **h** — Graphic handles for line objects

vector of `Line` graphic handles

Graphic handles for line objects, returned as a vector of `Line` graphic handles. Graphic handles are unique identifiers that you can use to query and modify the properties of a specific line on the plot. To view and set properties of line objects, use dot notation. For information on using dot notation, see “Access Property Values” (MATLAB). For information on the `Line` properties that you can set, see `Primitive Line`.

If you use the `'Quantile'` name-value pair argument, then `h` contains one handle for each of the three lines objects created. If you use both the `'Quantile'` and the `'Group'` name-value pair arguments, then `h` contains three handles for each group.

## See Also

`andrewsplot` | `glyphplot`

## Topics

“Grouping Variables” on page 2-59

“Access Property Values” (MATLAB)

**Introduced before R2006a**

## paramci

Confidence intervals for probability distribution parameters

### Syntax

```
ci = paramci(pd)
ci = paramci(pd, Name, Value)
```

### Description

`ci = paramci(pd)` returns the array `ci` containing the lower and upper boundaries of the 95% confidence interval for each parameter in probability distribution `pd`.

`ci = paramci(pd, Name, Value)` returns confidence intervals with additional options specified by one or more name-value pair arguments. For example, you can specify a different percentage for the confidence interval, or compute confidence intervals only for selected parameters.

### Examples

#### Parameter Confidence Intervals

Load the sample data. Create a vector containing the first column of students' exam grade data.

```
load examgrades
x = grades(:,1);
```

Fit a normal distribution object to the data.

```
pd = fitdist(x, 'Normal')
```

```
pd =
```



```

NormalDistribution

Normal distribution
  mu = 75.0083 [73.4321, 76.5846]
  sigma = 8.7202 [7.7391, 9.98843]

```

The intervals next to the parameter estimates are the 95% confidence intervals for the distribution parameters.

You can also obtain these intervals by using the function `paramci`.

```
ci = paramci(pd)
```

```

ci =

    73.4321    7.7391
    76.5846    9.9884

```

Column 1 of `ci` contains the lower and upper 95% confidence interval boundaries for the  $\mu$  parameter, and column 2 contains the boundaries for the  $\sigma$  parameter.

### Change Parameter Confidence Intervals

Load the sample data. Create a vector containing the first column of students' exam grade data.

```
load examgrades
x = grades(:,1);
```

Fit a normal distribution object to the data.

```
pd = fitdist(x, 'Normal')
```

```

pd =

NormalDistribution

Normal distribution

```

```
mu = 75.0083    [73.4321, 76.5846]
sigma = 8.7202  [7.7391, 9.98843]
```

Compute the 99% confidence interval for the distribution parameters.

```
ci = paramci(pd, 'Alpha', .01)
```

```
ci =
    72.9245    7.4627
    77.0922   10.4403
```

Column 1 of `ci` contains the lower and upper 99% confidence interval boundaries for the `mu` parameter, and column 2 contains the boundaries for the `sigma` parameter.

## Input Arguments

### **pd** — Probability distribution

probability distribution object

Probability distribution, specified as a probability distribution object. Create a probability distribution object with specified parameter values using `makedist`. Alternatively, create a probability distribution object by fitting it to data using `fitdist` or the Distribution Fitter app.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'Alpha', 0.01` specifies a 99% confidence interval.

### **Alpha** — Alpha level

0.05 (default) | scalar value in the range (0,1)

Alpha level for the confidence interval, specified as the comma-separated pair consisting of 'Alpha' and a scalar value in the range (0,1). The default value 0.05 corresponds to a 95% confidence interval.

Example: 'Alpha',0.01

Data Types: single | double

#### **Parameter — Parameter list**

vector | cell array of character vectors

Parameter list for which to compute confidence intervals, specified as the comma-separated pair consisting of 'Parameter' and a vector or a cell array of character vectors containing the parameter names. By default, paramci computes confidence intervals for all distribution parameters.

Example: 'Parameter','mu'

Data Types: char

#### **Type — Computation method**

'exact' | 'Wald' | 'lr'

Computation method for the confidence intervals, specified as the comma-separated pair consisting of 'Type' and 'exact', 'Wald', or 'lr'.

'exact' computes the confidence intervals using an exact method, and is available for the following distributions.

Distribution	Computation Method
Binomial	Compute using the Clopper-Pearson method based on exact probability calculations. This method does not provide exact coverage probabilities.
Exponential	Compute using a method based on a chi-square distribution. This method provides exact coverage for complete and Type 2 censored samples.

Distribution	Computation Method
Normal	Computation method based on $t$ and chi-square distributions for uncensored samples provides exact coverage for uncensored samples. For censored samples, <code>paramci</code> uses the Wald method if <code>Type</code> is <code>exact</code> .
Lognormal	Computation method based on $t$ and chi-square distributions for uncensored samples provides exact coverage. For censored samples, <code>paramci</code> uses the Wald method if <code>Type</code> is <code>exact</code> .
Poisson	Computation method based on a chi-square distribution provides exact coverage. For large degrees of freedom, the chi-square is approximated by a normal distribution for numerical efficiency.
Rayleigh	Computation method based on a chi-square distribution provides exact coverage probabilities.

'exact' is the default when it is available. Alternatively, you can specify 'Wald' to compute the confidence intervals using the Wald method, or 'lr' to compute the confidence intervals using the likelihood ratio method.

Example: `'Type', 'Wald'`

#### **LogFlag** — Boolean flag for log scale

vector

Boolean flag for the log scale, specified as the comma-separated pair consisting of 'LogFlag' and a vector containing Boolean values corresponding to each distribution parameter. The flag specifies which Wald intervals to compute on a log scale. The default values depend on the distribution.

Example: `'LogFlag', [0, 1]`

Data Types: `logical`

## Output Arguments

**ci** — Confidence interval

array

Confidence interval, returned as a  $p$ -by-2 array containing the lower and upper bounds of the  $(1 - \text{Alpha})\%$  confidence interval for each distribution parameter.  $p$  is the number of distribution parameters.

## See Also

`distributionFitter` | `fitdist` | `makedist`

## Topics

“Working with Probability Distributions” on page 5-3

Introduced before R2006a

## paramci

**Class:** ProbDistUnivParam

Return parameter confidence intervals of ProbDistUnivParam object

### Syntax

```
CI = paramci(PD)
CI = paramci(PD, Alpha)
```

### Description

`CI = paramci(PD)` returns `CI`, a 2-by-`N` array containing 95% confidence intervals for the parameters of the `ProbDistUnivParam` object `PD`. `N` is the number of parameters in the distribution. When you create `PD` by specifying parameters (such as using the `ProbDistUnivParam` constructor or using the `fitdist` function and specifying a 'binomial' or 'generalized pareto' distribution) rather than by fitting to data, the confidence intervals have a width of 0 because the parameters are viewed as estimates of an unknown parameter.

`CI = paramci(PD, Alpha)` returns  $100 \cdot (1 - \text{Alpha})\%$  confidence intervals. Default `Alpha` is 0.05, which specifies 95% confidence intervals.

---

**Note** If you create `PD` with a distribution that does not support confidence intervals, then `CI` contains NaN values.

---

### Input Arguments

<code>PD</code>	An object of the class <code>ProbDistUnivParam</code> .
<code>Alpha</code>	A value between 0 and 1 that specifies a confidence interval. Default is 0.05, which specifies 95% confidence intervals.

## Output Arguments

CI                    A 2-by-N array containing  $100 \cdot (1 - \text{Alpha})\%$  confidence intervals for the parameters of the `ProbDistUnivParam` object PD. N is the number of parameters in the distribution.

## See Also

`fitdist`

## paramci

**Class:** prob.ToolboxFittableParametricDistribution

**Package:** prob

Confidence intervals for probability distribution parameters

## Syntax

```
ci = paramci(pd)
ci = paramci(pd,Name,Value)
```

## Description

`ci = paramci(pd)` returns the array `ci` containing the lower and upper boundaries of the 95% confidence interval for each parameter in probability distribution `pd`.

`ci = paramci(pd,Name,Value)` returns confidence intervals with additional options specified by one or more name-value pair arguments. For example, you can specify a different percentage for the confidence interval, or compute confidence intervals only for selected parameters.

## Input Arguments

**pd** — Probability distribution

probability distribution object

Probability distribution, specified as a probability distribution object. Create a probability distribution object with specified parameter values using `makedist`. Alternatively, create a probability distribution object by fitting it to data using `fitdist` or the Distribution Fitter app.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single



quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'Alpha', 0.01` specifies a 99% confidence interval.

### **Alpha — Alpha level**

`0.05` (default) | scalar value in the range (0,1)

Alpha level for the confidence interval, specified as the comma-separated pair consisting of `'Alpha'` and a scalar value in the range (0,1). The default value `0.05` corresponds to a 95% confidence interval.

Example: `'Alpha', 0.01`

Data Types: `single` | `double`

### **Parameter — Parameter list**

`vector` | cell array of character vectors

Parameter list for which to compute confidence intervals, specified as the comma-separated pair consisting of `'Parameter'` and a vector or a cell array of character vectors containing the parameter names. By default, `paramci` computes confidence intervals for all distribution parameters.

Example: `'Parameter', 'mu'`

Data Types: `char`

### **Type — Computation method**

`'exact'` | `'Wald'` | `'lr'`

Computation method for the confidence intervals, specified as the comma-separated pair consisting of `'Type'` and `'exact'`, `'Wald'`, or `'lr'`.

`'exact'` computes the confidence intervals using an exact method, and is available for the following distributions.

Binomial	Compute using the Clopper-Pearson method based on exact probability calculations. This method does not provide exact coverage probabilities.
Exponential	Compute using a method based on a chi-square distribution. This method provides exact coverage for complete and Type 2 censored samples.

Normal	Computation method based on $t$ and chi-square distributions for uncensored samples provides exact coverage for uncensored samples. For censored samples, <code>paramci</code> uses the Wald method if <code>Type</code> is <code>exact</code> .
Lognormal	Computation method based on $t$ and chi-square distributions for uncensored samples provides exact coverage. For censored samples, <code>paramci</code> uses the Wald method if <code>Type</code> is <code>exact</code> .
Poisson	Computation method based on a chi-square distribution provides exact coverage. For large degrees of freedom, the chi-square is approximated by a normal distribution for numerical efficiency.
Rayleigh	Computation method based on a chi-square distribution provides exact coverage probabilities.

'exact' is the default when it is available. Alternatively, you can specify 'Wald' to compute the confidence intervals using the Wald method, or 'lr' to compute the confidence intervals using the likelihood ratio method.

Example: 'Type', 'Wald'

### **LogFlag** — Boolean flag for log scale

vector

Boolean flag for the log scale, specified as the comma-separated pair consisting of 'LogFlag' and a vector containing Boolean values corresponding to each distribution parameter. The flag specifies which Wald intervals to compute on a log scale. The default values depend on the distribution.

Example: 'LogFlag', [0,1]

Data Types: `logical`

## Output Arguments

### **ci** — Confidence interval

array

Confidence interval, returned as a  $p$ -by-2 array containing the lower and upper bounds of the  $(1 - \text{Alpha})\%$  confidence interval for each distribution parameter.  $p$  is the number of distribution parameters.

## Examples

### Parameter Confidence Intervals

Load the sample data. Create a vector containing the first column of students' exam grade data.

```
load examgrades
x = grades(:,1);
```

Fit a normal distribution object to the data.

```
pd = fitdist(x, 'Normal')
```

```
pd =
```

```
NormalDistribution
```

```
Normal distribution
```

```
mu = 75.0083 [73.4321, 76.5846]
```

```
sigma = 8.7202 [7.7391, 9.98843]
```

The intervals next to the parameter estimates are the 95% confidence intervals for the distribution parameters.

You can also obtain these intervals by using the function `paramci`.

```
ci = paramci(pd)
```

```
ci =
```

```
73.4321 7.7391
```

```
76.5846 9.9884
```

Column 1 of `ci` contains the lower and upper 95% confidence interval boundaries for the `mu` parameter, and column 2 contains the boundaries for the `sigma` parameter.

### Change Parameter Confidence Intervals

Load the sample data. Create a vector containing the first column of students' exam grade data.

```
load examgrades
x = grades(:,1);
```

Fit a normal distribution object to the data.

```
pd = fitdist(x, 'Normal')

pd =

NormalDistribution

Normal distribution
    mu = 75.0083    [73.4321, 76.5846]
    sigma = 8.7202    [7.7391, 9.98843]
```

Compute the 99% confidence interval for the distribution parameters.

```
ci = paramci(pd, 'Alpha', .01)

ci =

    72.9245    7.4627
    77.0922   10.4403
```

Column 1 of `ci` contains the lower and upper 99% confidence interval boundaries for the `mu` parameter, and column 2 contains the boundaries for the `sigma` parameter.

### See Also

`distributionFitter` | `fitdist` | `makedist`

## ParamCov property

**Class:** ProbDistParametric

Read-only covariance matrix of parameter estimates of ProbDistParametric object

### Description

ParamCov is a read-only property of the ProbDistParametric class. ParamCov is a covariance matrix containing the parameter estimates of a distribution represented by a ProbDistParametric object. ParamCov has a size of NumParams-by-NumParams.

### Values

This covariance matrix includes estimates for both the specified parameters and parameters that are fit to the data. For specified parameters, the covariance is 0, indicating the parameter is known exactly. Use this information to view and compare the descriptions of parameters supplied to create distributions.

## ParamDescription property

**Class:** ProbDistParametric

Read-only cell array specifying descriptions of parameters of ProbDistParametric object

### Description

`ParamDescription` is a read-only property of the `ProbDistParametric` class. `ParamDescription` is a cell array of character vectors specifying the descriptions or meanings of the parameters of a distribution represented by a `ProbDistParametric` object. `ParamDescription` has a length of `NumParams`.

### Values

This cell array includes a brief description of the meaning of both the specified parameters and parameters that are fit to the data. The description is the same as the parameter name when no further description information is available. Use this information to view and compare the descriptions of parameters used to create distributions.

## ParamsFixed property

**Class:** ProbDistParametric

Read-only logical array specifying fixed parameters of ProbDistParametric object

### Description

ParamsFixed is a read-only property of the ProbDistParametric class.

ParamsFixed is a logical array specifying the fixed parameters of a distribution represented by a ProbDistParametric object. ParamsFixed has a length of NumParams.

### Values

This array specifies a 1 (*true*) for fixed parameters, and a 0 (*false*) for parameters that are estimated from the input data. Use this information to view and compare the fixed parameters used to create distributions.

## ParamNames property

**Class:** ProbDistParametric

Read-only cell array specifying names of parameters of ProbDistParametric object

### Description

`ParamNames` is a read-only property of the `ProbDistParametric` class. `ParamNames` is a cell array of character vectors specifying the names of the parameters of a distribution represented by a `ProbDistParametric` object. `ParamNames` has a length of `NumParams`.

### Values

This cell array includes the names of both the specified parameters and parameters that are fit to the data. Use this information to view and compare the names of parameters used to create distributions.



# Params property

**Class:** NaiveBayes

Parameter estimates

## Description

The `Params` property is an `NClasses-by-NDims` cell array containing the parameter estimates, excluding the class priors. `Params(i,j)` contains the parameter estimates for the `j`th feature in the `i`th class. `Params(i,j)` is an empty cell if the `i`th class is empty.

The entry in `Params(i,j)` depends on the distribution type used for the `j`th feature, as follows:

'normal'	A vector of length two. The first element is the mean, and the second element is standard deviation.
'kernel'	A <code>ProbDistUnivKernel</code> object
'mvmn'	A vector containing the probability for each possible value of the <code>j</code> th feature in the <code>i</code> th class. The order of the probabilities is decided by the sorted order of all the unique values of the <code>j</code> th feature.
'mn'	A scalar representing the probability the <code>j</code> th token appearing in the <code>i</code> th class, $\text{Prob}(\text{token } j \mid \text{class } i)$ . It is estimated as $(1 + \text{the number of occurrence of token } J \text{ in class } I) / (\text{NDims} + \text{the total number of token occurrence in class } I)$ .

## Params property

**Class:** ProbDistParametric

Read-only array specifying values of parameters of ProbDistParametric object

## Description

Params is a read-only property of the ProbDistParametric class. Params is an array of values specifying the values of the parameters of a distribution represented by a ProbDistParametric object. Params has a length of NumParams.

## Values

This array includes the values of both the specified parameters and parameters that are fit to the data. Use this information to view and compare the values of parameters used to create distributions.

# prob.ParametricTruncatableDistribution class

**Package:** prob

**Superclasses:**

Parametric truncatable probability distribution object

## Description

Create a probability distribution object with specified parameter values using `makedist`.

## Methods

`mean` Mean of probability distribution object  
`std` Standard deviation of probability distribution object  
`var` Variance of probability distribution object

## Inherited Methods

`cdf` Cumulative distribution function of probability distribution object  
`icdf` Inverse cumulative distribution function of probability distribution object  
`iqr` Interquartile range of probability distribution object  
`median` Median of probability distribution object  
`pdf` Probability density function of probability distribution object  
`random` Generate random numbers from probability distribution object  
`truncate` Truncate probability distribution object

## See Also

`makedist`

## **Topics**

Class Attributes (MATLAB)

Property Attributes (MATLAB)

# parent

**Class:** classregtree

Parent node

---

**Note** classregtree will be removed in a future release. See fitctree, fitrtree, ClassificationTree, or RegressionTree instead.

---

## Syntax

```
p = parent(t)
p = parent(t,nodes)
```

## Description

`p = parent(t)` returns an  $n$ -element vector `p` containing the number of the parent node for each node in the tree `t`, where  $n$  is the number of nodes. The parent of the root node is 0.

`p = parent(t,nodes)` takes a vector `nodes` of node numbers and returns the parent nodes for the specified nodes.

## Examples

Create a classification tree for Fisher's iris data:

```
load fisheriris;

t = classregtree(meas,species,...
                'names',{'SL' 'SW' 'PL' 'PW'})
t =
Decision tree for classification
1  if PL<2.45 then node 2 else node 3
2  class = setosa
```

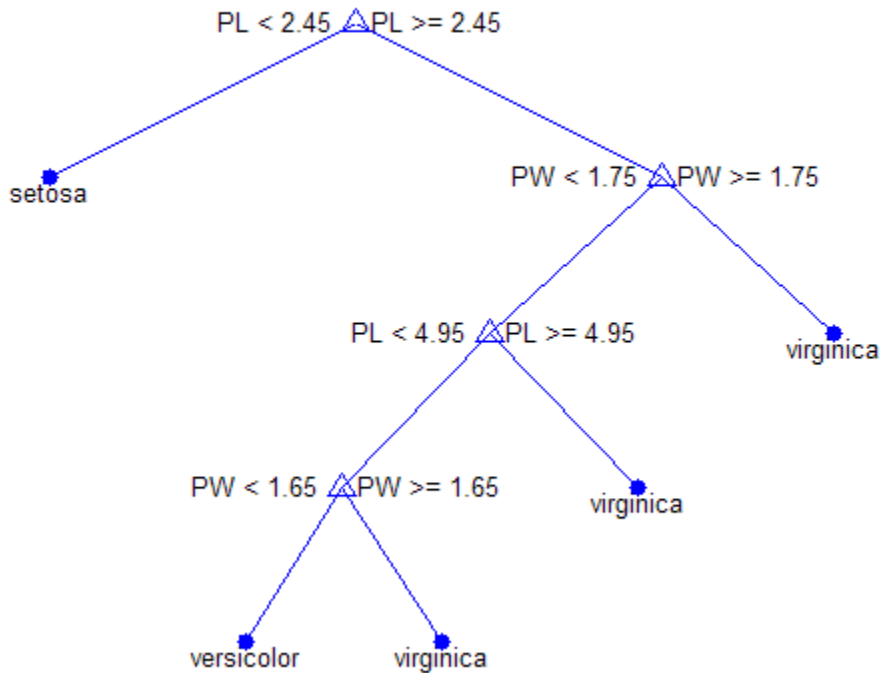
```

3  if PW<1.75 then node 4 else node 5
4  if PL<4.95 then node 6 else node 7
5  class = virginica
6  if PW<1.65 then node 8 else node 9
7  class = virginica
8  class = versicolor
9  class = virginica

```

```
view(t)
```

Click to display:  Magnification:  Pruning level:



```

p = parent(t)
p =
  0
  1

```

1  
3  
3  
4  
4  
6  
6

## References

[1] Breiman, L., J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Boca Raton, FL: CRC Press, 1984.

## See Also

`children` | `classregtree` | `numnodes`

## paretotails class

### Superclasses:

Empirical distributions with Pareto tails

## Construction

paretotails                      Construct Pareto tails object

## Methods

lowerparams                      Lower Pareto tails parameters

upperparams                      Upper Pareto tails parameters

## Inherited Methods

Methods in the following table are inherited from `piecwisedistribution` on page 32-4720.

boundary	Piecewise distribution boundaries
cdf	Cumulative distribution function for piecewise distribution
disp	Display <code>piecwisedistribution</code> object
display	Display <code>piecwisedistribution</code> object
icdf	Inverse cumulative distribution function for piecewise distribution
nsegments	Number of segments
pdf	Probability density function for piecewise distribution
random	Random numbers from piecewise distribution
segment	Segments containing values



## Properties

### **NumSegments**

Number of segments in the piecewise distribution fit to data, stored as a scalar value.

### **UpperParameters**

The parameters of the two-parameter “Generalized Pareto Distribution” on page B-58 (GPD) fit to the observations in the upper tail, stored as a vector. First value is the shape parameter and the second value is the scale parameter of the GPD.

### **LowerParameters**

The parameters of the two-parameter GPD fit to the observations in the lower tail, stored as a vector. First value is the shape parameter and the second value is the scale parameter of the GPD.

## Copy Semantics

Value. To learn how this affects your use of the class, see [Comparing Handle and Value Classes \(MATLAB\)](#) in the MATLAB Object-Oriented Programming documentation.

## See Also

### **Topics**

“Generalized Pareto Distribution” on page B-58

## paretotails

**Class:** `paretotails`

Construct Pareto tails object

### Syntax

```
obj = paretotails(x, pl, pu)
obj = paretotails(x, pl, pu, cdfun)
```

### Description

`obj = paretotails(x, pl, pu)` creates an object `obj` defining a distribution consisting of the empirical distribution of `x` in the center and Pareto distributions in the tails. `x` is a real-valued vector of data values whose extreme observations are fit to generalized Pareto distributions (GPDs). `pl` and `pu` identify the lower- and upper-tail cumulative probabilities such that  $100 \cdot pl$  and  $100 \cdot (1 - pu)$  percent of the observations in `x` are, respectively, fit to a GPD by maximum likelihood. If `pl` is 0, or if there are not at least two distinct observations in the lower tail, then no lower Pareto tail is fit. If `pu` is 1, or if there are not at least two distinct observations in the upper tail, then no upper Pareto tail is fit.

`obj = paretotails(x, pl, pu, cdfun)` uses `cdfun` to estimate the cdf of `x` between the lower and upper tail probabilities. `cdfun` may be any of the following:

- `'ecdf'` — Uses an interpolated empirical cdf, with data values as the midpoints in the vertical steps in the empirical cdf, and computed by linear interpolation between data values. This is the default.
- `'kernel'` — Uses a kernel-smoothing estimate of the cdf.
- `@fun` — Uses a handle to a function of the form `[p, xi] = fun(x)` that accepts the input data vector `x` and returns a vector `p` of cdf values and a vector `xi` of evaluation points. Values in `xi` must be sorted and distinct but need not equal the values in `x`.

`cdfun` is used to compute the quantiles corresponding to `pl` and `pu` by inverse interpolation, and to define the fitted distribution between these quantiles.

The output object `obj` is a Pareto tails object with methods to evaluate the cdf, inverse cdf, and other functions of the fitted distribution. These methods are well-suited to copula and other Monte Carlo simulations. The `pdf` method in the tails is the GPD density, but in the center it is computed as the slope of the interpolated cdf.

The `paretotails` class is a subclass of the `piecewisedistribution` class, and many of its methods are derived from that class.

## Examples

### Fit Pareto Tails to a Probability Distribution

Generate sample data containing 100 random numbers from a  $t$  distribution with 3 degrees of freedom.

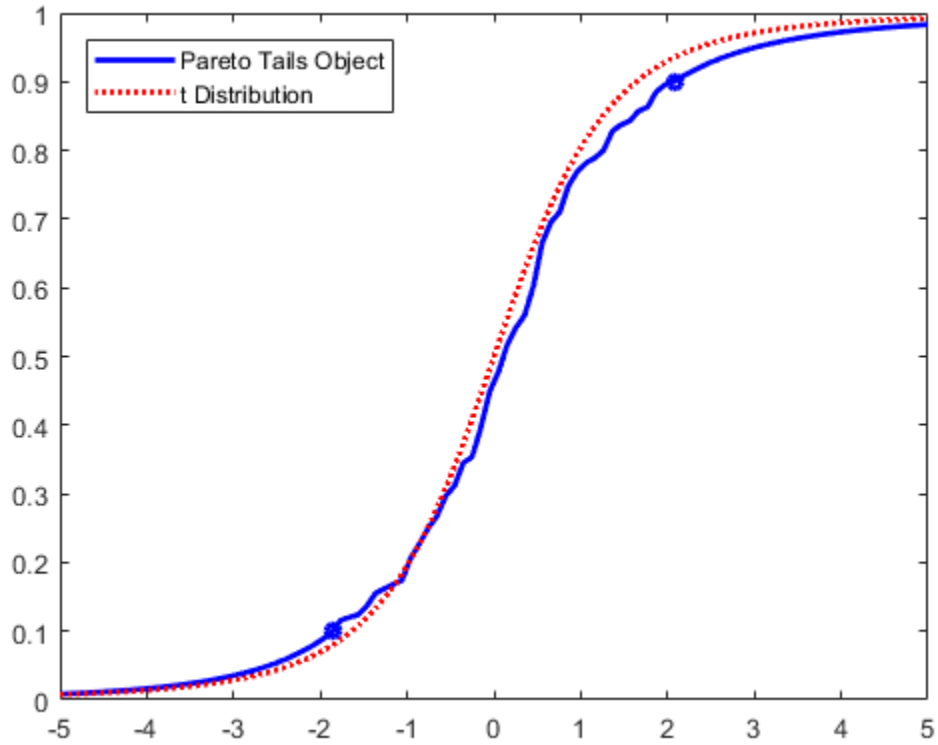
```
rng default; % For reproducibility
t = trnd(3,100,1);
```

Fit pareto tails to the distribution at cumulative probabilities 0.1 and 0.9.

```
obj = paretotails(t,0.1,0.9);
[p,q] = boundary(obj);
```

Plot the cdf of the Pareto tails and the cdf of the fitted  $t$  distribution on the same figure.

```
x = linspace(-5,5);
plot(x,cdf(obj,x),'b-','LineWidth',2)
hold on;
plot(x,tcdf(x,3),'r:','LineWidth',2)
plot(q,p,'bo','LineWidth',2,'MarkerSize',5)
legend('Pareto Tails Object','t Distribution',...
       'Location','NW')
hold off;
```



## See Also

`cdf` | `ecdf` | `gpfit` | `icdf` | `ksdensity`

Introduced in R2007a

# partialcorr

Linear or rank partial correlation coefficients

## Syntax

```
rho = partialcorr(x)
rho = partialcorr(x, z)
rho = partialcorr(x, y, z)
rho = partialcorr( ____, Name, Value)
[rho, pval] = partialcorr( ____)
```

## Description

`rho = partialcorr(x)` returns the sample linear partial correlation coefficients between pairs of variables in `x`, controlling for the remaining variables in `x`.

`rho = partialcorr(x, z)` returns the sample linear partial correlation coefficients between pairs of variables in `x`, controlling for the variables in `z`.

`rho = partialcorr(x, y, z)` returns the sample linear partial correlation coefficients between pairs of variables in `x` and `y`, controlling for the variables in `z`.

`rho = partialcorr( ____, Name, Value)` returns the sample linear partial correlation coefficients with additional options specified by one or more name-value pair arguments, using input arguments from any of the previous syntaxes. For example, you can specify whether to use Pearson or Spearman partial correlations, or specify how to treat missing values.

`[rho, pval] = partialcorr( ____)` also returns a matrix `pval` of  $p$ -values for testing the hypothesis of no partial correlation against the one- or two-sided alternative that there is a nonzero partial correlation.

## Examples

### Compute Partial Correlation Coefficients

Compute partial correlation coefficients between pairs of variables in the input matrix.

Load the sample data. Convert the genders in `hospital.Sex` to numeric group identifiers.

```
load hospital;
hospital.SexID = grp2idx(hospital.Sex);
```

Create an input matrix containing the sample data.

```
x = [hospital.SexID hospital.Age hospital.Smoker hospital.Weight];
```

Each row in `x` contains a patient's gender, age, smoking status, and weight.

Compute partial correlation coefficients between pairs of variables in `x`, while controlling for the effects of the remaining variables in `x`.

```
rho = partialcorr(x)
```

```
rho =

    1.0000    -0.0105    0.0273    0.9421
   -0.0105    1.0000    0.0419    0.0369
    0.0273    0.0419    1.0000    0.0451
    0.9421    0.0369    0.0451    1.0000
```

The matrix `rho` indicates, for example, a correlation of 0.9421 between gender and weight after controlling for all other variables in `x`. You can return the  $P$ -values as a second output, and examine them to confirm whether these correlations are statistically significant.

For a clearer display, create a table with appropriate variable and row labels.

```
rho = array2table(rho, ...
    'VariableNames', {'SexID', 'Age', 'Smoker', 'Weight'}, ...
    'RowNames', {'SexID', 'Age', 'Smoker', 'Weight'});

disp('Partial Correlation Coefficients')
disp(rho)
```

Partial Correlation Coefficients				
	SexID	Age	Smoker	Weight
	-----	-----	-----	-----
SexID	1	-0.01052	0.027324	0.9421
Age	-0.01052	1	0.041945	0.036873
Smoker	0.027324	0.041945	1	0.045106
Weight	0.9421	0.036873	0.045106	1

### Test for Partial Correlations with Controlled Variables

Test for partial correlation between pairs of variables in the input matrix, while controlling for the effects of a second set of variables.

Load the sample data. Convert the genders in `hospital.Sex` to numeric group identifiers.

```
load hospital;
hospital.SexID = grp2idx(hospital.Sex);
```

Create two matrices containing the sample data.

```
x = [hospital.Age hospital.BloodPressure];
z = [hospital.SexID hospital.Smoker hospital.Weight];
```

The `x` matrix contains the variables to test for partial correlation. The `z` matrix contains the variables to control for. The measurements for `BloodPressure` are contained in two columns: The first column contains the upper (systolic) number, and the second column contains the lower (diastolic) number. `partialcorr` treats each column as a separate variable.

Test for partial correlation between pairs of variables in `x`, while controlling for the effects of the variables in `z`. Compute the correlation coefficients.

```
[rho,pval] = partialcorr(x,z)
```

```
rho =
    1.0000    0.1300    0.0462
    0.1300    1.0000    0.0012
```

```
0.0462    0.0012    1.0000

pval =

    0    0.2044    0.6532
0.2044    0    0.9903
0.6532    0.9903    0
```

The large values in `pval` indicate that there is no significant correlation between age and either blood pressure measurement after controlling for gender, smoking status, and weight.

For a clearer display, create tables with appropriate variable and row labels.

```
rho = array2table(rho, ...
    'VariableNames', {'Age', 'BPTop', 'BPBottom'}, ...
    'RowNames', {'Age', 'BPTop', 'BPBottom'});

pval = array2table(pval, ...
    'VariableNames', {'Age', 'BPTop', 'BPBottom'}, ...
    'RowNames', {'Age', 'BPTop', 'BPBottom'});

disp('Partial Correlation Coefficients')
disp(rho)
disp('p-values')
disp(pval)
```

```
Partial Correlation Coefficients
```

	Age	BPTop	BPBottom
Age	1	0.13	0.046202
BPTop	0.13	1	0.0012475
BPBottom	0.046202	0.0012475	1

```
p-values
```

	Age	BPTop	BPBottom
Age	0	0.20438	0.65316
BPTop	0.20438	0	0.99032



```
BPBottom    0.65316    0.99032    0
```

### Test for Paired Partial Correlation Coefficients

Test for partial correlation between pairs of variables in the  $x$  and  $y$  input matrices, while controlling for the effects of a third set of variables.

Load the sample data. Convert the genders in `hospital.Sex` to numeric group identifiers.

```
load hospital;
hospital.SexID = grp2idx(hospital.Sex);
```

Create three matrices containing the sample data.

```
x = [hospital.BloodPressure];
y = [hospital.Weight hospital.Age];
z = [hospital.SexID hospital.Smoker];
```

`partialcorr` can test for partial correlation between the pairs of variables in  $x$  (the systolic and diastolic blood pressure measurements) and  $y$  (weight and age), while controlling for the variables in  $z$  (gender and smoking status). The measurements for `BloodPressure` are contained in two columns: The first column contains the upper (systolic) number, and the second column contains the lower (diastolic) number. `partialcorr` treats each column as a separate variable.

Test for partial correlation between pairs of variables in  $x$  and  $y$ , while controlling for the effects of the variables in  $z$ . Compute the correlation coefficients.

```
[rho,pval] = partialcorr(x,y,z)
```

```
rho =
```

```
-0.0257    0.1289
 0.0292    0.0472
```

```
pval =
```

```
0.8018    0.2058
```

```
0.7756    0.6442
```

The results in `pval` indicate that, after controlling for gender and smoking status, there is no significant correlation between either of a patient's blood pressure measurements and that patient's weight or age.

For a clearer display, create tables with appropriate variable and row labels.

```
rho = array2table(rho, ...
    'VariableNames',{'BPTop','BPBottom'},...
    'RowNames',{'Weight','Age'});

pval = array2table(pval, ...
    'VariableNames',{'BPTop','BPBottom'},...
    'RowNames',{'Weight','Age'});

disp('Partial Correlation Coefficients')
disp(rho)
disp('p-values')
disp(pval)
```

```
Partial Correlation Coefficients
           BPTop      BPBottom
-----
Weight    -0.02568    0.12893
Age        0.029168    0.047226

p-values
           BPTop      BPBottom
-----
Weight    0.80182    0.2058
Age       0.77556    0.64424
```

### One-Tailed Partial Correlation Test

Test the hypothesis that pairs of variables have no correlation, against the alternative hypothesis that the correlation is greater than 0.

Load the sample data. Convert the genders in `hospital.Sex` to numeric group identifiers.

```
load hospital;
hospital.SexID = grp2idx(hospital.Sex);
```

Create three matrices containing the sample data.

```
x = [hospital.BloodPressure];
y = [hospital.Weight hospital.Age];
z = [hospital.SexID hospital.Smoker];
```

`partialcorr` can test for partial correlation between the pairs of variables in `x` (the systolic and diastolic blood pressure measurements) and `y` (weight and age), while controlling for the variables in `z` (gender and smoking status). The measurements for `BloodPressure` are contained in two columns: The first column contains the upper (systolic) number, and the second column contains the lower (diastolic) number. `partialcorr` treats each column as a separate variable.

Compute the correlation coefficients using a right-tailed test.

```
[rho,pval] = partialcorr(x,y,z,'Tail','right')
```

```
rho =
```

```
   -0.0257    0.1289
    0.0292    0.0472
```

```
pval =
```

```
   0.5991    0.1029
   0.3878    0.3221
```

The results in `pval` indicate that `partialcorr` does not reject the null hypothesis of nonzero correlations between the variables in `x` and `y`, after controlling for the variables in `z`, when the alternative hypothesis is that the correlations are greater than 0.

For a clearer display, create tables with appropriate variable and row labels.

```
rho = array2table(rho, ...
    'VariableNames',{'BPTop','BPBottom'},...
```

```
    'RowNames', {'Weight', 'Age'});  
  
pval = array2table(pval, ...  
    'VariableNames', {'BPTop', 'BPBottom'}, ...  
    'RowNames', {'Weight', 'Age'});  
  
disp('Partial Correlation Coefficients')  
disp(rho)  
disp('p-values')  
disp(pval)
```

```
Partial Correlation Coefficients  
                BPTop      BPBottom  
                -----      -----  
Weight          -0.02568      0.12893  
Age             0.029168      0.047226  
  
p-values  
                BPTop      BPBottom  
                -----      -----  
Weight          0.59909      0.1029  
Age             0.38778      0.32212
```

## Input Arguments

### **x** — Data matrix

matrix

Data matrix, specified as an  $n$ -by- $p_x$  matrix. The rows of  $x$  correspond to observations, and the columns correspond to variables.

Data Types: `single` | `double`

### **y** — Data matrix

matrix

Data matrix, specified as an  $n$ -by- $p_y$  matrix. The rows of  $y$  correspond to observations, and the columns correspond to variables.

Data Types: `single` | `double`

**z — Data matrix**

matrix

Data matrix, specified as an  $n$ -by- $p_z$  matrix. The rows of  $z$  correspond to observations, and columns correspond to variables.

Data Types: `single` | `double`

**Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'Type', 'Spearman', 'Rows', 'complete'` computes Spearman partial correlations using only the data in rows that contain no missing values.

**Type — Type of partial correlations**

`'Pearson'` (default) | `'Spearman'`

Type of partial correlations to compute, specified as the comma-separated pair consisting of `'Type'` and one of the following.

<code>'Pearson'</code>	Compute Pearson (linear) partial correlations.
<code>'Spearman'</code>	Compute Spearman (rank) partial correlations.

Example: `'Type', 'Spearman'`

**Rows — Rows to use in computation**

`'all'` (default) | `'complete'` | `'pairwise'`

Rows to use in computation, specified as the comma-separated pair consisting of `'Rows'` and one of the following.

<code>'all'</code>	Use all rows regardless of missing (NaN) values.
<code>'complete'</code>	Use only rows with no missing values.

`'pairwise'` Compute  $\rho(i, j)$  using rows with no missing values in column  $i$  or  $j$ .

Using `'pairwise'` can produce a  $\rho$  that is not positive definite. `'complete'` always produces a positive definite  $\rho$ , but when data is missing, the estimates are generally based on fewer observations.

Example: `'Rows', 'complete'`

### **Tail — Alternative hypothesis**

`'both' (default) | 'right' | 'left'`

Alternative hypothesis to test against, specified as the comma-separated pair consisting of `'Tail'` and one of the following.

<code>'both'</code>	Test the alternative hypothesis that the correlation is not 0.
<code>'right'</code>	Test the alternative hypothesis that the correlation is greater than 0.
<code>'left'</code>	Test the alternative hypothesis that the correlation is less than 0.

Example: `'Tail', 'right'`

## Output Arguments

### **$\rho$ — Sample linear partial correlation coefficients**

matrix

Sample linear partial correlation coefficients, returned as a matrix.

- If you input only an  $x$  matrix,  $\rho$  is a symmetric  $p_x$ -by- $p_x$  matrix. The  $(i,j)$ th entry is the sample linear partial correlation between the  $i$ -th and  $j$ -th columns in  $x$ .
- If you input  $x$  and  $z$  matrices,  $\rho$  is a symmetric  $p_x$ -by- $p_x$  matrix. The  $(i,j)$ th entry is the sample linear partial correlation between the  $i$ th and  $j$ th columns in  $x$ , controlled for the variables in  $z$ .
- If you input  $x$ ,  $y$ , and  $z$  matrices,  $\rho$  is a  $p_x$ -by- $p_y$  matrix, where the  $(i,j)$ th entry is the sample linear partial correlation between the  $i$ th column in  $x$  and the  $j$ th column in  $y$ , controlled for the variables in  $z$ .

If the covariance matrix of  $[x, z]$  is

$$S = \begin{pmatrix} S_{xx} & S_{xz} \\ S_{xz}^T & S_{zz} \end{pmatrix},$$

then the partial correlation matrix of  $x$ , controlling for  $z$ , can be defined formally as a normalized version of the covariance matrix  $S_{xy} = S_{xx} - (S_{xz}S_{zz}^{-1}S_{xz}^T)$

**pval** — ***p*-values**  
matrix

*p*-values, returned as a matrix. Each element of `pval` is the *p*-value for the corresponding element of `rho`.

If `pval(i, j)` is small, then the corresponding partial correlation `rho(i, j)` is statistically significantly different from 0.

`partialcorr` computes *p*-values for linear and rank partial correlations using a Student's *t* distribution for a transformation of the correlation. This is exact for linear partial correlation when  $x$  and  $z$  are normal, but is a large-sample approximation otherwise.

## See Also

`corr` | `corrcoef` | `partialcorri` | `tiedrank`

Introduced before R2006a

## partialcorri

Partial correlation coefficients adjusted for internal variables

### Syntax

```
rho = partialcorri(y, x)
rho = partialcorri(y, x, z)
rho = partialcorri( ____, Name, Value)
[rho, pval] = partialcorri( ____, Name, Value)
```

### Description

`rho = partialcorri(y, x)` returns the sample linear partial correlation coefficients between pairs of variables in `y` and `x`, adjusting for the remaining variables in `x`.

`rho = partialcorri(y, x, z)` returns the sample linear partial correlation coefficients between pairs of variables in `y` and `x`, adjusting for the remaining variables in `x`, after first controlling both `x` and `y` for the variables in `z`.

`rho = partialcorri( ____, Name, Value)` returns the sample linear partial correlation coefficients with additional options specified by one or more name-value pair arguments, using input arguments from any of the previous syntaxes. For example, you can specify whether to use Pearson or Spearman partial correlations, or specify how to treat missing values.

`[rho, pval] = partialcorri( ____, Name, Value)` also returns a matrix `pval` of  $p$ -values for testing the hypothesis of no partial correlation against the one- or two-sided alternative that there is a nonzero partial correlation.

### Examples



## Compute Partial Correlation Coefficients

Compute partial correlation coefficients for each pair of variables in the  $x$  and  $y$  input matrices, while controlling for the effects of the remaining variables in  $x$ .

Load the sample data.

```
load carsmall;
```

The data contains measurements from cars manufactured in 1970, 1976, and 1982. It includes MPG and Acceleration as performance measures, and Displacement, Horsepower, and Weight as design variables. Acceleration is the time required to accelerate from 0 to 60 miles per hour, so a high value for Acceleration corresponds to a vehicle with low acceleration.

Define the input matrices. The  $y$  matrix includes the performance measures, and the  $x$  matrix includes the design variables.

```
y = [MPG,Acceleration];
x = [Displacement,Horsepower,Weight];
```

Compute the correlation coefficients. Include only rows with no missing values in the computation.

```
rho = partialcorri(y,x,'Rows','complete')
```

```
rho =
    -0.0537    -0.1520    -0.4856
    -0.3994    -0.4008     0.4912
```

The results suggest, for example, a 0.4912 correlation between weight and acceleration after controlling for the effects of displacement and horsepower. You can return the  $P$ -values as a second output, and examine them to confirm whether these correlations are statistically significant.

For a clearer display, create a table with appropriate variable and row labels.

```
rho = array2table(rho, ...
    'VariableNames',{'Displacement','Horsepower','Weight'}, ...
    'RowNames',{'MPG','Acceleration'});
```

```
disp('Partial Correlation Coefficients')
disp(rho)
```

Partial Correlation Coefficients	Displacement	Horsepower	Weight
MPG	-0.053684	-0.15199	-0.48563
Acceleration	-0.39941	-0.40075	0.49123

### Test Partial Correlations While Controlling for Additional Variables

Test for partial correlation between pairs of variables in the  $x$  and  $y$  input matrices, while controlling for the effects of the remaining variables in  $x$  plus additional variables in matrix  $z$ .

Load the sample data.

```
load carsmall;
```

The data contains measurements from cars manufactured in 1970, 1976, and 1982. It includes MPG and Acceleration as performance measures, and Displacement, Horsepower, and Weight as design variables. Acceleration is the time required to accelerate from 0 to 60 miles per hour, so a high value for Acceleration corresponds to a vehicle with low acceleration.

Create a new variable Headwind, and randomly generate data to represent the notion of an average headwind along the performance measurement route.

```
rng('default'); % For reproducibility
Headwind = (10:-0.2:-9.8)' + 5*randn(100,1);
```

Since headwind can affect the performance measures, control for its effects when testing for partial correlation between the remaining variables.

Define the input matrices. The  $y$  matrix includes the performance measures, and the  $x$  matrix includes the design variables. The  $z$  matrix contains additional variables to control for when computing the partial correlations, such as headwind.

```
y = [MPG,Acceleration];
x = [Displacement,Horsepower,Weight];
z = Headwind;
```

Compute the partial correlation coefficients. Include only rows with no missing values in the computation.

```
[rho,pval] = partialcorri(y,x,z,'Rows','complete')
```

```
rho =
    0.0572    -0.1055   -0.5736
   -0.3845   -0.3966    0.4674
```

```
pval =
    0.5923    0.3221    0.0000
    0.0002    0.0001    0.0000
```

The small returned **P**-value of 0.001 in `pval` indicates, for example, a significant negative correlation between horsepower and acceleration, after controlling for displacement, weight, and headwind.

For a clearer display, create tables with appropriate variable and row labels.

```
rho = array2table(rho, ...
    'VariableNames',{'Displacement','Horsepower','Weight'}, ...
    'RowNames',{'MPG','Acceleration'});

pval = array2table(pval, ...
    'VariableNames',{'Displacement','Horsepower','Weight'}, ...
    'RowNames',{'MPG','Acceleration'});

disp('Partial Correlation Coefficients, Accounting for Headwind')
disp(rho)
disp('p-values, Accounting for Headwind')
disp(pval)
```

```
Partial Correlation Coefficients, Accounting for Headwind
                Displacement    Horsepower    Weight
                _____    _____    _____
```

MPG	0.057197	-0.10555	-0.57358
Acceleration	-0.38452	-0.39658	0.4674
p-values, Accounting for Headwind			
	Displacement	Horsepower	Weight
MPG	0.59233	0.32212	3.4401e-09
Acceleration	0.00018272	0.00010902	3.4091e-06

## Input Arguments

### **x** — Data matrix

matrix

Data matrix, specified as an  $n$ -by- $p_x$  matrix. The rows of **x** correspond to observations, and the columns correspond to variables.

Data Types: `single` | `double`

### **y** — Data matrix

matrix

Data matrix, specified as an  $n$ -by- $p_y$  matrix. The rows of **y** correspond to observations, and the columns correspond to variables.

Data Types: `single` | `double`

### **z** — Data matrix

matrix

Data matrix, specified as an  $n$ -by- $p_z$  matrix. The rows of **z** correspond to observations, and the columns correspond to variables.

Data Types: `single` | `double`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside `single`

quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

Example: `'Type', 'Spearman', 'Rows', 'complete'` computes Spearman partial correlations using only the data in rows that contain no missing values.

### **Type** — Type of partial correlations

`'Pearson'` (default) | `'Spearman'`

Type of partial correlations to compute, specified as the comma-separated pair consisting of `'Type'` and either `'Pearson'` or `'Spearman'`. `Pearson` computes the Pearson (linear) partial correlations. `Spearman` computes the Spearman (rank) partial correlations.

Example: `'Type', 'Spearman'`

### **Rows** — Rows to use in computation

`'all'` (default) | `'complete'` | `'pairwise'`

Rows to use in computation, specified as the comma-separated pair consisting of `'Rows'` and one of the following.

`'all'`

Use all rows regardless of missing (NaN) values.

`'complete'`

Use only rows with no missing values.

`'pairwise'`

Use all available values in each column of  $y$  when computing the partial correlation coefficients and  $p$ -values corresponding to that column. For each column of  $y$ , rows will be dropped corresponding to missing values in  $x$  (and/or  $z$ , if supplied). However, remaining rows with valid values in that column of  $y$  are used, even if there are missing values in other columns of  $y$ .

Example: `'Rows', 'complete'`

### **Tail** — Alternative hypothesis

`'both'` (default) | `'right'` | `'left'`

Alternative hypothesis to test against, specified as the comma-separated pair consisting of `'Tail'` and one of the following.

<code>'both'</code>	Test the alternative hypothesis that the correlation is not zero.
<code>'right'</code>	Test the alternative hypothesis that the correlation is greater than 0.
<code>'left'</code>	Test the alternative hypothesis that the correlation is less than 0.

Example: `'Tail', 'right'`

## Output Arguments

### **rho** — Sample linear partial correlation coefficients

matrix

Sample linear partial correlation coefficients, returned as a  $p_y$ -by- $p_x$  matrix.

- If you input  $x$  and  $y$  matrices, the  $(i,j)$ th entry is the sample linear partial correlation between the  $i$ th column in  $y$  and the  $j$ th column in  $x$ , controlled for all the columns of  $x$  except column  $j$ .
- If you input  $x$ ,  $y$ , and  $z$  matrices, the  $(i,j)$ th entry is the sample linear partial correlation between the  $i$ th column in  $y$  and the  $j$ th column in  $x$ , adjusted for all the columns of  $x$  except column  $j$ , after first controlling both  $x$  and  $y$  for the variables in  $z$ .

### **pval** — $p$ -values

matrix

$p$ -values, returned as a matrix. Each element of `pval` is the  $p$ -value for the corresponding element of `rho`. If `pval(i,j)` is small, then the corresponding partial correlation `rho(i,j)` is statistically significantly different from zero.

`partialcorri` computes  $p$ -values for linear and rank partial correlations using a Student's  $t$  distribution for a transformation of the correlation. This is exact for linear partial correlation when  $x$  and  $z$  are normal, but is a large-sample approximation otherwise.

## See Also

`corr` | `partialcorr`

Introduced in R2013b

## pca

Principal component analysis of raw data

### Syntax

```
coeff = pca(X)
coeff = pca(X, Name, Value)
[coeff, score, latent] = pca( ___ )
[coeff, score, latent, tsquared] = pca( ___ )
[coeff, score, latent, tsquared, explained, mu] = pca( ___ )
```

### Description

`coeff = pca(X)` returns the principal component coefficients, also known as loadings, for the  $n$ -by- $p$  data matrix `X`. Rows of `X` correspond to observations and columns correspond to variables. The coefficient matrix is  $p$ -by- $p$ . Each column of `coeff` contains coefficients for one principal component, and the columns are in descending order of component variance. By default, `pca` centers the data and uses the singular value decomposition (SVD) algorithm.

`coeff = pca(X, Name, Value)` returns any of the output arguments in the previous syntaxes using additional options for computation and handling of special data types, specified by one or more `Name, Value` pair arguments.

For example, you can specify the number of principal components `pca` returns or an algorithm other than SVD to use.

`[coeff, score, latent] = pca( ___ )` also returns the principal component scores in `score` and the principal component variances in `latent`. You can use any of the input arguments in the previous syntaxes.

Principal component scores are the representations of `X` in the principal component space. Rows of `score` correspond to observations, and columns correspond to components.

The principal component variances are the eigenvalues of the covariance matrix of `X`.



`[coeff, score, latent, tsquared] = pca( ___ )` also returns the Hotelling's T-squared statistic for each observation in  $X$ .

`[coeff, score, latent, tsquared, explained, mu] = pca( ___ )` also returns `explained`, the percentage of the total variance explained by each principal component and `mu`, the estimated mean of each variable in  $X$ .

## Examples

### Principal Components of a Data Set

Load the sample data set.

```
load hald
```

The ingredients data has 13 observations for 4 variables.

Find the principal components for the ingredients data.

```
coeff = pca(ingredients)
```

```
coeff =
    -0.0678    -0.6460     0.5673     0.5062
    -0.6785    -0.0200    -0.5440     0.4933
     0.0290     0.7553     0.4036     0.5156
     0.7309    -0.1085    -0.4684     0.4844
```

The rows of `coeff` contain the coefficients for the four ingredient variables, and its columns correspond to four principal components.

### PCA in the Presence of Missing Data

Find the principal component coefficients when there are missing values in a data set.

Load the sample data set.

```
load imports-85
```

Data matrix  $X$  has 13 continuous variables in columns 3 to 15: wheel-base, length, width, height, curb-weight, engine-size, bore, stroke, compression-ratio, horsepower, peak-rpm, city-mpg, and highway-mpg. The variables bore and stroke are missing four values in rows 56 to 59, and the variables horsepower and peak-rpm are missing two values in rows 131 and 132.

Perform principal component analysis.

```
coeff = pca(X(:,3:15));
```

By default, `pca` performs the action specified by the 'Rows', 'complete' name-value pair argument. This option removes the observations with NaN values before calculation. Rows of NaNs are reinserted into `score` and `tsquared` at the corresponding locations, namely rows 56 to 59, 131, and 132.

Use 'pairwise' to perform the principal component analysis.

```
coeff = pca(X(:,3:15), 'Rows', 'pairwise');
```

In this case, `pca` computes the  $(i,j)$  element of the covariance matrix using the rows with no NaN values in the columns  $i$  or  $j$  of  $X$ . Note that the resulting covariance matrix might not be positive definite. This option applies when the algorithm `pca` uses is eigenvalue decomposition. When you don't specify the algorithm, as in this example, `pca` sets it to 'eig'. If you require 'svd' as the algorithm, with the 'pairwise' option, then `pca` returns a warning message, sets the algorithm to 'eig' and continues.

If you use the 'Rows', 'all' name-value pair argument, `pca` terminates because this option assumes there are no missing values in the data set.

```
coeff = pca(X(:,3:15), 'Rows', 'all');
```

```
Error using pca (line 180)
```

```
Raw data contains NaN missing value while 'Rows' option is set to 'all'. Consider using
```

## Weighted PCA

Use the inverse variable variances as weights while performing the principal components analysis.

Load the sample data set.

```
load hald
```

Perform the principal component analysis using the inverse of variances of the ingredients as variable weights.

```
[wcoeff,~,latent,~,explained] = pca(ingredients,...
'VariableWeights','variance')
```

```
wcoeff =
```

```

-2.7998    2.9940   -3.9736    1.4180
-8.7743   -6.4411    4.8927    9.9863
 2.5240   -3.8749   -4.0845    1.7196
 9.1714    7.5529    3.2710   11.3273
```

```
latent =
```

```

 2.2357
 1.5761
 0.1866
 0.0016
```

```
explained =
```

```

55.8926
39.4017
 4.6652
 0.0406
```

Note that the coefficient matrix, `wcoeff`, is not orthonormal.

Calculate the orthonormal coefficient matrix.

```
coefforth = inv(diag(std(ingredients))) * wcoeff
```

```
coefforth =
```

```

-0.4760    0.5090   -0.6755    0.2411
-0.5639   -0.4139    0.3144    0.6418
 0.3941   -0.6050   -0.6377    0.2685
```

```
0.5479    0.4512    0.1954    0.6767
```

Check orthonormality of the new coefficient matrix, `coefforth`.

```
coefforth*coefforth'
```

```
ans =
```

```
1.0000    0.0000   -0.0000   -0.0000
0.0000    1.0000   -0.0000    0
-0.0000   -0.0000    1.0000    0.0000
-0.0000         0    0.0000    1.0000
```

### PCA Using ALS for Missing Data

Find the principal components using the alternating least squares (ALS) algorithm when there are missing values in the data.

Load the sample data.

```
load hald
```

The ingredients data has 13 observations for 4 variables.

Perform principal component analysis using the ALS algorithm and display the component coefficients.

```
[coeff,score,latent,tsquared,explained] = pca(ingredients);
coeff
```

```
coeff =
```

```
-0.0678   -0.6460    0.5673    0.5062
-0.6785   -0.0200   -0.5440    0.4933
0.0290    0.7553    0.4036    0.5156
0.7309   -0.1085   -0.4684    0.4844
```

Introduce missing values randomly.

```

y = ingredients;
rng('default'); % for reproducibility
ix = random('unif',0,1,size(y))<0.30;
y(ix) = NaN

```

```

y =

```

```

     7     26     6   NaN
     1     29    15    52
    NaN   NaN     8    20
    11    31   NaN    47
     7     52     6    33
    NaN    55   NaN   NaN
    NaN    71   NaN     6
     1    31   NaN    44
     2   NaN   NaN    22
    21    47     4    26
    NaN    40    23    34
    11    66     9   NaN
    10    68     8    12

```

Approximately 30% of the data has missing values now, indicated by NaN.

Perform principal component analysis using the ALS algorithm and display the component coefficients.

```

[coeff1,score1,latent,tsquared,explained,mu1] = pca(y,...
'algorithm','als');
coeff1

```

```

coeff1 =

```

```

-0.0362    0.8215   -0.5252    0.2190
-0.6831   -0.0998    0.1828    0.6999
 0.0169    0.5575    0.8215   -0.1185
 0.7292   -0.0657    0.1261    0.6694

```

Display the estimated mean.

```

mu1

```

```
mul =  
  
    8.9956    47.9088    9.0451    28.5515
```

Reconstruct the observed data.

```
t = score1*coeff1' + repmat(mul,13,1)
```

```
t =  
  
    7.0000    26.0000    6.0000    51.5250  
    1.0000    29.0000    15.0000    52.0000  
   10.7819    53.0230     8.0000    20.0000  
   11.0000    31.0000    13.5500    47.0000  
    7.0000    52.0000     6.0000    33.0000  
   10.4818    55.0000     7.8328    17.9362  
    3.0982    71.0000    11.9491     6.0000  
    1.0000    31.0000   -0.5161    44.0000  
    2.0000    53.7914     5.7710    22.0000  
   21.0000    47.0000     4.0000    26.0000  
   21.5809    40.0000    23.0000    34.0000  
   11.0000    66.0000     9.0000     5.7078  
   10.0000    68.0000     8.0000    12.0000
```

The ALS algorithm estimates the missing values in the data.

Another way to compare the results is to find the angle between the two spaces spanned by the coefficient vectors. Find the angle between the coefficients found for complete data and data with missing values using ALS.

```
subspace(coeff,coeff1)
```

```
ans =  
  
    4.1842e-16
```

This is a small value. It indicates that the results if you use `pca` with `'Rows','complete'` name-value pair argument when there is no missing data and if

you use `pca` with `'algorithm', 'als'` name-value pair argument when there is missing data are close to each other.

Perform the principal component analysis using `'Rows', 'complete'` name-value pair argument and display the component coefficients.

```
[coeff2,score2,latent,tsquared,explained,mu2] = pca(y,...
'Rows','complete');
coeff2
```

```
coeff2 =

    -0.2054    0.8587    0.0492
    -0.6694   -0.3720    0.5510
     0.1474   -0.3513   -0.5187
     0.6986   -0.0298    0.6518
```

In this case, `pca` removes the rows with missing values, and `y` has only four rows with no missing values. `pca` returns only three principal components. You cannot use the `'Rows', 'pairwise'` option because the covariance matrix is not positive semidefinite and `pca` returns an error message.

Find the angle between the coefficients found for complete data and data with missing values using listwise deletion (when `'Rows', 'complete'`).

```
subspace(coeff(:,1:3),coeff2)
```

```
ans =

    0.3576
```

The angle between the two spaces is substantially larger. This indicates that these two results are different.

Display the estimated mean.

```
mu2
```

```
mu2 =
```

```
7.8889 46.9091 9.8750 29.6000
```

In this case, the mean is just the sample mean of  $y$ .

Reconstruct the observed data.

```
score2*coeff2'
```

```
ans =
```

```
      NaN      NaN      NaN      NaN
-7.5162 -18.3545  4.0968 22.0056
      NaN      NaN      NaN      NaN
      NaN      NaN      NaN      NaN
-0.5644  5.3213 -3.3432  3.6040
      NaN      NaN      NaN      NaN
      NaN      NaN      NaN      NaN
      NaN      NaN      NaN      NaN
      NaN      NaN      NaN      NaN
12.8315 -0.1076 -6.3333 -3.7758
      NaN      NaN      NaN      NaN
      NaN      NaN      NaN      NaN
 1.4680 20.6342 -2.9292 -18.0043
```

This shows that deleting rows containing NaN values does not work as well as the ALS algorithm. Using ALS is better when the data has too many missing values.

### Principal Component Coefficients, Scores, and Variances

Find the coefficients, scores, and variances of the principal components.

Load the sample data set.

```
load hald
```

The ingredients data has 13 observations for 4 variables.

Find the principal component coefficients, scores, and variances of the components for the ingredients data.



```
[coeff,score,latent] = pca(ingredients)
```

```
coeff =
```

```

-0.0678    -0.6460     0.5673     0.5062
-0.6785    -0.0200    -0.5440     0.4933
 0.0290     0.7553     0.4036     0.5156
 0.7309    -0.1085    -0.4684     0.4844

```

```
score =
```

```

 36.8218   -6.8709   -4.5909    0.3967
 29.6073    4.6109   -2.2476   -0.3958
-12.9818   -4.2049    0.9022   -1.1261
 23.7147   -6.6341    1.8547   -0.3786
 -0.5532   -4.4617   -6.0874    0.1424
-10.8125   -3.6466    0.9130   -0.1350
-32.5882    8.9798   -1.6063    0.0818
 22.6064   10.7259    3.2365    0.3243
 -9.2626    8.9854   -0.0169   -0.5437
 -3.2840  -14.1573    7.0465    0.3405
  9.2200   12.3861    3.4283    0.4352
-25.5849   -2.7817   -0.3867    0.4468
-26.9032   -2.9310   -2.4455    0.4116

```

```
latent =
```

```

517.7969
 67.4964
 12.4054
  0.2372

```

Each column of `score` corresponds to one principal component. The vector, `latent`, stores the variances of the four principal components.

Reconstruct the centered ingredients data.

```
Xcentered = score*coeff'
```

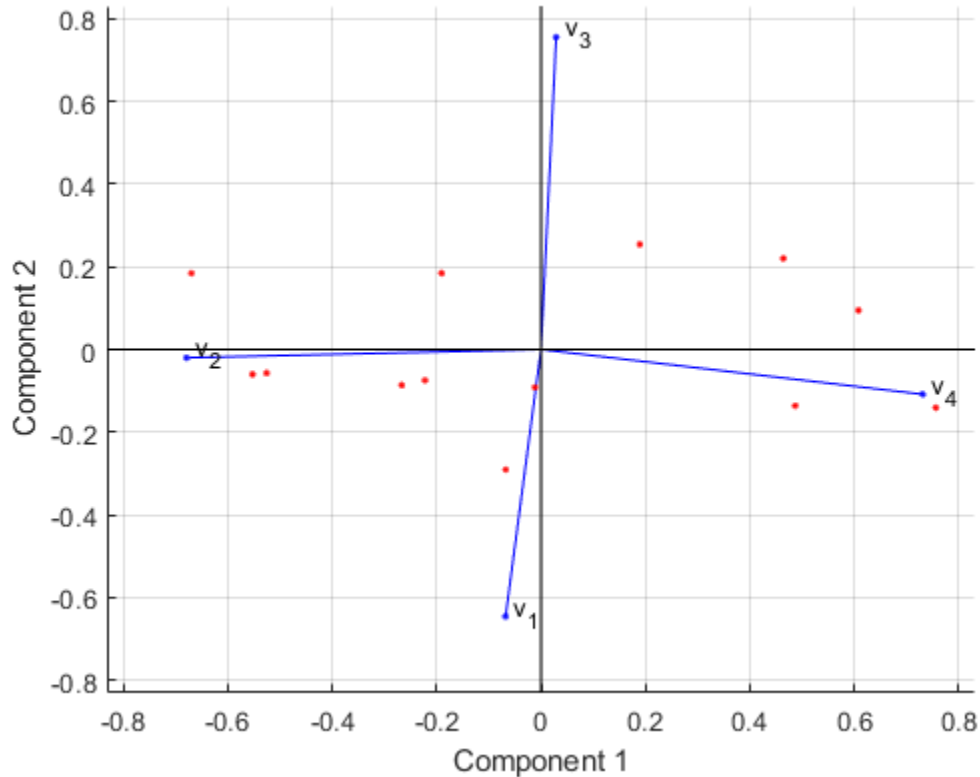
```
Xcentered =
```

```
-0.4615 -22.1538 -5.7692 30.0000
-6.4615 -19.1538 3.2308 22.0000
3.5385 7.8462 -3.7692 -10.0000
3.5385 -17.1538 -3.7692 17.0000
-0.4615 3.8462 -5.7692 3.0000
3.5385 6.8462 -2.7692 -8.0000
-4.4615 22.8462 5.2308 -24.0000
-6.4615 -17.1538 10.2308 14.0000
-5.4615 5.8462 6.2308 -8.0000
13.5385 -1.1538 -7.7692 -4.0000
-6.4615 -8.1538 11.2308 4.0000
3.5385 17.8462 -2.7692 -18.0000
2.5385 19.8462 -3.7692 -18.0000
```

The new data in `Xcentered` is the original ingredients data centered by subtracting the column means from corresponding columns.

Visualize both the orthonormal principal component coefficients for each variable and the principal component scores for each observation in a single plot.

```
biplot(coeff(:,1:2), 'scores', score(:,1:2), 'varlabels', {'v_1', 'v_2', 'v_3', 'v_4'});
```



All four variables are represented in this biplot by a vector, and the direction and length of the vector indicate how each variable contributes to the two principal components in the plot. For example, the first principal component, which is on the horizontal axis, has positive coefficients for the third and fourth variables. Therefore, vectors  $v_3$  and  $v_4$  are directed into the right half of the plot. The largest coefficient in the first principal component is the fourth, corresponding to the variable  $v_4$ .

The second principal component, which is on the vertical axis, has negative coefficients for the variables  $v_1$ ,  $v_2$ , and  $v_4$ , and a positive coefficient for the variable  $v_3$ .

This 2-D biplot also includes a point for each of the 13 observations, with coordinates indicating the score of each observation for the two principal components in the plot. For example, points near the left edge of the plot have the lowest scores for the first principal

component. The points are scaled with respect to the maximum score value and maximum coefficient length, so only their relative locations can be determined from the plot.

### T-Squared Statistic

Find the Hotelling's T-squared statistic values.

Load the sample data set.

```
load hald
```

The ingredients data has 13 observations for 4 variables.

Perform the principal component analysis and request the T-squared values.

```
[coeff,score,latent,tsquared] = pca(ingredients);  
tsquared
```

```
tsquared =
```

```
5.6803  
3.0758  
6.0002  
2.6198  
3.3681  
0.5668  
3.4818  
3.9794  
2.6086  
7.4818  
4.1830  
2.2327  
2.7216
```

Request only the first two principal components and compute the T-squared values in the reduced space of requested principal components.

```
[coeff,score,latent,tsquared] = pca(ingredients,'NumComponents',2);  
tsquared
```

```
tsquared =  
5.6803  
3.0758  
6.0002  
2.6198  
3.3681  
0.5668  
3.4818  
3.9794  
2.6086  
7.4818  
4.1830  
2.2327  
2.7216
```

Note that even when you specify a reduced component space, `pca` computes the T-squared values in the full space, using all four components.

The T-squared value in the reduced space corresponds to the Mahalanobis distance in the reduced space.

```
tsqreduced = mahal(score, score)
```

```
tsqreduced =  
3.3179  
2.0079  
0.5874  
1.7382  
0.2955  
0.4228  
3.2457  
2.6914  
1.3619  
2.9903  
2.4371  
1.3788  
1.5251
```

Calculate the T-squared values in the discarded space by taking the difference of the T-squared values in the full space and Mahalanobis distance in the reduced space.

```
tsqdiscarded = tsquared - tsqreduced
```

```
tsqdiscarded =
```

```
2.3624  
1.0679  
5.4128  
0.8816  
3.0726  
0.1440  
0.2362  
1.2880  
1.2467  
4.4915  
1.7459  
0.8539  
1.1965
```

### Percent Variability Explained by Principal Components

Find the percent variability explained by the principal components. Show the data representation in the principal components space.

Load the sample data set.

```
load imports-85
```

Data matrix  $X$  has 13 continuous variables in columns 3 to 15: wheel-base, length, width, height, curb-weight, engine-size, bore, stroke, compression-ratio, horsepower, peak-rpm, city-mpg, and highway-mpg.

Find the percent variability explained by principal components of these variables.

```
[coeff,score,latent,tsquared,explained] = pca(X(:,3:15));
```

```
explained
```

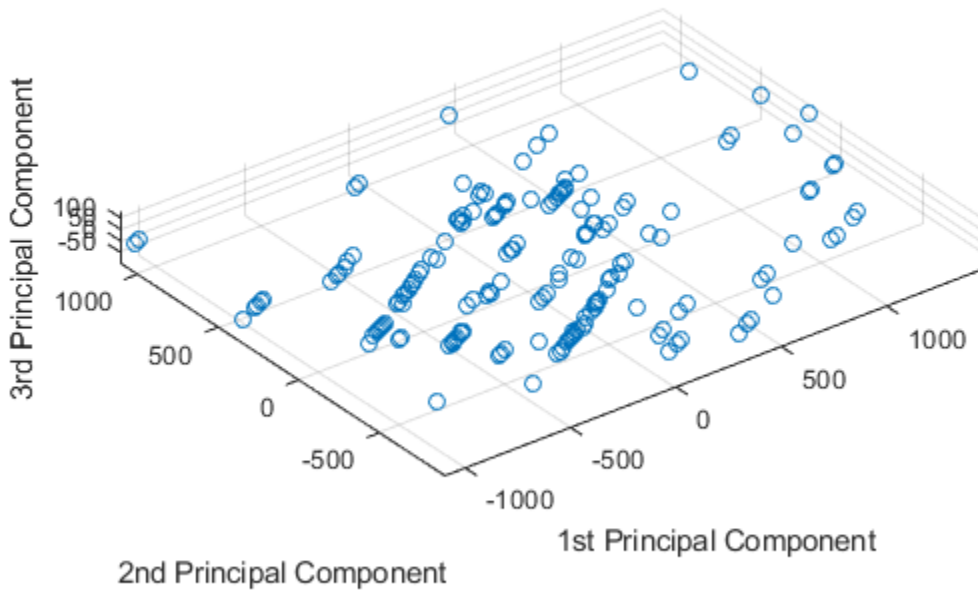
```
explained =
```

```
64.3429
35.4484
 0.1550
 0.0379
 0.0078
 0.0048
 0.0013
 0.0011
 0.0005
 0.0002
 0.0002
 0.0000
 0.0000
```

The first three components explain 99.95% of all variability.

Visualize the data representation in the space of the first three principal components.

```
scatter3(score(:,1),score(:,2),score(:,3))
axis equal
xlabel('1st Principal Component')
ylabel('2nd Principal Component')
zlabel('3rd Principal Component')
```



The data shows the largest variability along the first principal component axis. This is the largest possible variance among all possible choices of the first axis. The variability along the second principal component axis is the largest among all possible remaining choices of the second axis. The third principal component axis has the third largest variability, which is significantly smaller than the variability along the second principal component axis. The fourth through thirteenth principal component axes are not worth inspecting, because they explain only 0.05% of all variability in the data.

To skip any of the outputs, you can use `~` instead in the corresponding element. For example, if you don't want to get the T-squared values, specify



```
[coeff,score,latent,~,explained] = pca(X(:,3:15));
```

- “Analyze Quality of Life in U.S. Cities Using PCA” on page 15-85

## Input Arguments

### **x** — Input data

matrix

Input data for which to compute the principal components, specified as an  $n$ -by- $p$  matrix. Rows of  $X$  correspond to observations and columns to variables.

Data Types: `single` | `double`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example:

`'Algorithm','eig','Centered',false,'Rows','all','NumComponents',3` specifies that `pca` uses eigenvalue decomposition algorithm, not center the data, use all of the observations, and return only the first three principal components.

### **Algorithm** — Principal component algorithm

`'svd'` (default) | `'eig'` | `'als'`

Principal component algorithm that `pca` uses to perform the principal component analysis, specified as the comma-separated pair consisting of `'Algorithm'` and one of the following.

<code>'svd'</code>	Default. Singular value decomposition (SVD) of $X$ .
<code>'eig'</code>	Eigenvalue decomposition (EIG) of the covariance matrix. The EIG algorithm is faster than SVD when the number of observations, $n$ , exceeds the number of variables, $p$ , but is less accurate because the condition number of the covariance is the square of the condition number of $X$ .

'als'	<p>Alternating least squares (ALS) algorithm. This algorithm finds the best rank-<math>k</math> approximation by factoring <math>X</math> into a <math>n</math>-by-<math>k</math> left factor matrix, <math>L</math>, and a <math>p</math>-by-<math>k</math> right factor matrix, <math>R</math>, where <math>k</math> is the number of principal components. The factorization uses an iterative method starting with random initial values.</p> <p>ALS is designed to better handle missing values. It is preferable to pairwise deletion ('Rows', 'pairwise') and deals with missing values without listwise deletion ('Rows', 'complete'). It can work well for data sets with a small percentage of missing data at random, but might not perform well on sparse data sets.</p>
-------	--

Example: 'Algorithm', 'eig'

Data Types: char

#### Centered — Indicator for centering columns

true (default) | false

Indicator for centering the columns, specified as the comma-separated pair consisting of 'Centered' and one of these logical expressions.

true	Default. <code>pca</code> centers $X$ by subtracting column means before computing singular value decomposition or eigenvalue decomposition. If $X$ contains NaN missing values, <code>nanmean</code> is used to find the mean with any available data. You can reconstruct the centered data using <code>score*coeff</code> .
false	In this case <code>pca</code> does not center the data. You can reconstruct the original data using <code>score*coeff</code> .

Example: 'Centered', false

Data Types: logical

#### Economy — Indicator for economy size output

true (default) | false

Indicator for the economy size output when the degrees of freedom on page 32-4617,  $d$ , is smaller than the number of variables,  $p$ , specified as the comma-separated pair consisting of 'Economy' and one of these logical expressions.

<code>true</code>	<p>Default. <code>pca</code> returns only the first <math>d</math> elements of <code>latent</code> and the corresponding columns of <code>coeff</code> and <code>score</code>.</p> <p>This option can be significantly faster when the number of variables <math>p</math> is much larger than <math>d</math>.</p>
<code>false</code>	<code>pca</code> returns all elements of <code>latent</code> . The columns of <code>coeff</code> and <code>score</code> corresponding to zero elements in <code>latent</code> are zeros.

Note that when  $d < p$ , `score(:, d+1:p)` and `latent(d+1:p)` are necessarily zero, and the columns of `coeff(:, d+1:p)` define directions that are orthogonal to  $X$ .

Example: `'Economy', false`

Data Types: `logical`

#### **NumComponents** — Number of components requested

number of variables (default) | scalar integer

Number of components requested, specified as the comma-separated pair consisting of `'NumComponents'` and a scalar integer  $k$  satisfying  $0 < k \leq p$ , where  $p$  is the number of original variables in  $X$ . When specified, `pca` returns the first  $k$  columns of `coeff` and `score`.

Example: `'NumComponents', 3`

Data Types: `single` | `double`

#### **Rows** — Action to take for NaN values

`'complete'` (default) | `'pairwise'` | `'all'`

Action to take for NaN values in the data matrix  $X$ , specified as the comma-separated pair consisting of `'Rows'` and one of the following.

<code>'complete'</code>	<p>Default. Observations with NaN values are removed before calculation. Rows of NaNs are reinserted into <code>score</code> and <code>tsquared</code> at the corresponding locations.</p>
-------------------------	--

'pairwise'	<p>This option only applies when the algorithm is 'eig'. If you don't specify the algorithm along with 'pairwise', then <code>pca</code> sets it to 'eig'. If you specify 'svd' as the algorithm, along with the option 'Rows', 'pairwise', then <code>pca</code> returns a warning message, sets the algorithm to 'eig' and continues.</p> <p>When you specify the 'Rows', 'pairwise' option, <code>pca</code> computes the <math>(i,j)</math> element of the covariance matrix using the rows with no NaN values in the columns <math>i</math> or <math>j</math> of <math>X</math>.</p> <p>Note that the resulting covariance matrix might not be positive definite. In that case, <code>pca</code> terminates with an error message.</p>
'all'	$X$ is expected to have no missing values. <code>pca</code> uses all of the data and terminates if any NaN value is found.

Example: 'Rows', 'pairwise'

Data Types: char

### **Weights — Observation weights**

ones (default) | row vector

Observation weights, specified as the comma-separated pair consisting of 'Weights' and a vector of length  $n$  containing all positive elements.

Data Types: single | double

### **VariableWeights — Variable weights**

row vector | 'variance'

Variable weights on page 32-4618, specified as the comma-separated pair consisting of 'VariableWeights' and one of the following.

row vector	Vector of length $p$ containing all positive elements.
'variance'	<p>The variable weights are the inverse of sample variance. If you also assign weights to observations using 'Weights', then the variable weights become the inverse of weighted sample variance.</p> <p>If 'Centered' is set to <code>true</code> at the same time, the data matrix <math>X</math> is centered and standardized. In this case, <code>pca</code> returns the principal components based on the correlation matrix.</p>

Example: 'VariableWeights', 'variance'

Data Types: single | double | char

#### **Coeff0** — Initial value for coefficients

matrix of random values (default) |  $p$ -by- $k$  matrix

Initial value for the coefficient matrix `coeff`, specified as the comma-separated pair consisting of 'Coeff0' and a  $p$ -by- $k$  matrix, where  $p$  is the number of variables, and  $k$  is the number of principal components requested.

---

**Note** You can use this name-value pair only when 'algorithm' is 'als'.

---

Data Types: single | double

#### **Score0** — Initial value for scores

matrix of random values (default) |  $k$ -by- $m$  matrix

Initial value for scores matrix `score`, specified as a comma-separated pair consisting of 'Score0' and an  $n$ -by- $k$  matrix, where  $n$  is the number of observations and  $k$  is the number of principal components requested.

---

**Note** You can use this name-value pair only when 'algorithm' is 'als'.

---

Data Types: single | double

#### **Options** — Options for iterations

structure

Options for the iterations, specified as a comma-separated pair consisting of 'Options' and a structure created by the `statset` function. `pca` uses the following fields in the options structure.

'Display'	Level of display output. Choices are 'off', 'final', and 'iter'.
'MaxIter'	Maximum number steps allowed. The default is 1000. Unlike in optimization settings, reaching the <code>MaxIter</code> value is regarded as convergence.

'TolFun'	Positive number giving the termination tolerance for the cost function. The default is 1e-6.
'TolX'	Positive number giving the convergence threshold for the relative change in the elements of the left and right factor matrices, L and R, in the ALS algorithm. The default is 1e-6.

---

**Note** You can use this name-value pair only when 'algorithm' is 'als'.

---

You can change the values of these fields and specify the new structure in `pca` using the 'Options' name-value pair argument.

```
Example: opt = statset('pca'); opt.MaxIter = 2000; coeff =  
pca(X, 'Options', opt);
```

Data Types: struct

## Output Arguments

### **coeff** — Principal component coefficients

matrix

Principal component coefficients, returned as a  $p$ -by- $p$  matrix. Each column of `coeff` contains coefficients for one principal component. The columns are in the order of descending component variance, `latent`.

### **score** — Principal component scores

matrix

Principal component scores, returned as a matrix. Rows of `score` correspond to observations, and columns to components.

### **latent** — Principal component variances

column vector

Principal component variances, that is the eigenvalues of the covariance matrix of `X`, returned as a column vector.

### **tsquared** — Hotelling's T-squared statistic

column vector

“Hotelling’s T-Squared Statistic” on page 32-4617, which is the sum of squares of the standardized scores for each observation, returned as a column vector.

**explained — Percentage of total variance explained**

column vector

Percentage of the total variance explained by each principal component, returned as a column vector.

**mu — Estimated means**

row vector

Estimated means of the variables in  $X$ , returned as a row vector when `Centered` is set to `true`. When `Centered` is `false`, the software does not compute the means and returns a vector of zeros.

## Definitions

### Hotelling’s T-Squared Statistic

Hotelling’s T-squared statistic is a statistical measure of the multivariate distance of each observation from the center of the data set.

Even when you request fewer components than the number of variables, `pca` uses all principal components to compute the T-squared statistic (computes it in the full space). If you want the T-squared statistic in the reduced or the discarded space, do one of the following:

- For the T-squared statistic in the reduced space, use `mahal(score, score)`.
- For the T-squared statistic in the discarded space, first compute the T-squared statistic using `[coeff, score, latent, tsquared] = pca(X, 'NumComponents', k, ...)`, compute the T-squared statistic in the reduced space using `tsqreduced = mahal(score, score)`, and then take the difference: `tsquared - tsqreduced`.

### Degrees of Freedom

The degrees of freedom,  $d$ , is equal to  $n - 1$ , if data is centered and  $n$  otherwise, where:

- $n$  is the number of rows without any NaNs if you use 'Rows', 'complete'.
- $n$  is the number of rows without any NaNs in the column pair that has the maximum number of rows without NaNs if you use 'Rows', 'pairwise'.

## Variable Weights

Note that when variable weights are used, the coefficient matrix is not orthonormal. Suppose the variable weights vector you used is called `varwei`, and the principal component coefficients vector `pca` returned is `wcoeff`. You can then calculate the orthonormal coefficients using the transformation `diag(sqrt(varwei))*wcoeff`.

## References

- [1] Jolliffe, I. T. *Principal Component Analysis*. 2nd ed., Springer, 2002.
- [2] Krzanowski, W. J. *Principles of Multivariate Analysis*. Oxford University Press, 1988.
- [3] Seber, G. A. F. *Multivariate Observations*. Wiley, 1984.
- [4] Jackson, J. E. A. *User's Guide to Principal Components*. Wiley, 1988.
- [5] Roweis, S. “EM Algorithms for PCA and SPCA.” *In Proceedings of the 1997 Conference on Advances in Neural Information Processing Systems*. Vol.10 (NIPS 1997), Cambridge, MA, USA: MIT Press, 1998, pp. 626–632.
- [6] Ilin, A., and T. Raiko. “Practical Approaches to Principal Component Analysis in the Presence of Missing Values.” *J. Mach. Learn. Res.*. Vol. 11, August 2010, pp. 1957–2000.

## Extended Capabilities

### Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function supports tall arrays for out-of-memory data with some limitations.



- `pca` works directly with tall arrays by computing the covariance matrix and using the in-memory `pcacov` function to compute the principle components.
- Supported syntaxes are:
  - `coeff = pca(X)`
  - `[coeff,score,latent] = pca(X)`
  - `[coeff,score,latent,explained] = pca(X)`
  - `[coeff,score,latent,tsquared] = pca(X)`
  - `[coeff,score,latent,tsquared,explained] = pca(X)`
- Name-value pair arguments are not supported.

For more information, see “Tall Arrays” (MATLAB).

## C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- When 'Algorithm' is 'als', the 'Display' value for 'Options' is ignored.
- If supplied, 'Weights' and 'VariableWeights' must be real.
- The generated code always returns the fifth output `explained` as a column vector.
- The generated code always returns the sixth output `mu` as a row vector.
- If `mu` is empty, `pca` returns `mu` as a 1-by-0 array. `pca` does not convert `mu` to a 0-by-0 empty array.
- The generated code does not treat an input matrix `X` that has all NaN values as a special case. The output dimensions are commensurate with corresponding finite inputs.

## See Also

`barttest` | `biplot` | `canoncorr` | `factoran` | `pcacov` | `pcares` | `ppca` | `rotatefactors`

## Topics

“Analyze Quality of Life in U.S. Cities Using PCA” on page 15-85

“Principal Component Analysis (PCA)” on page 15-83

**Introduced in R2012b**

## pcacov

Principal component analysis on covariance matrix

### Syntax

```
COEFF = pcacov(V)
[COEFF,latent] = pcacov(V)
[COEFF,latent,explained] = pcacov(V)
```

### Description

`COEFF = pcacov(V)` performs principal components analysis on the p-by-p covariance matrix `V` and returns the principal component coefficients, also known as loadings. `COEFF` is a p-by-p matrix, with each column containing coefficients for one principal component. The columns are in order of decreasing component variance.

`pcacov` does not standardize `V` to have unit variances. To perform principal components analysis on standardized variables, use the correlation matrix  $R = V ./ (SD * SD')$ , where  $SD = \text{sqrt}(\text{diag}(V))$ , in place of `V`. To perform principal components analysis directly on the data matrix, use `pca`.

`[COEFF,latent] = pcacov(V)` returns `latent`, a vector containing the principal component variances, that is, the eigenvalues of `V`.

`[COEFF,latent,explained] = pcacov(V)` returns `explained`, a vector containing the percentage of the total variance explained by each principal component.

### Examples

```
load hald
covx = cov(ingredients);
[COEFF,latent,explained] = pcacov(covx)
COEFF =
    0.0678 -0.6460    0.5673 -0.5062
```

```
0.6785 -0.0200 -0.5440 -0.4933  
-0.0290 0.7553 0.4036 -0.5156  
-0.7309 -0.1085 -0.4684 -0.4844
```

```
latent =  
517.7969  
67.4964  
12.4054  
0.2372
```

```
explained =  
86.5974  
11.2882  
2.0747  
0.0397
```

## References

- [1] Jackson, J. E. *A User's Guide to Principal Components*. Hoboken, NJ: John Wiley and Sons, 1991.
- [2] Jolliffe, I. T. *Principal Component Analysis*. 2nd ed., New York: Springer-Verlag, 2002.
- [3] Krzanowski, W. J. *Principles of Multivariate Analysis: A User's Perspective*. New York: Oxford University Press, 1988.
- [4] Seber, G. A. F., *Multivariate Observations*, Wiley, 1984.

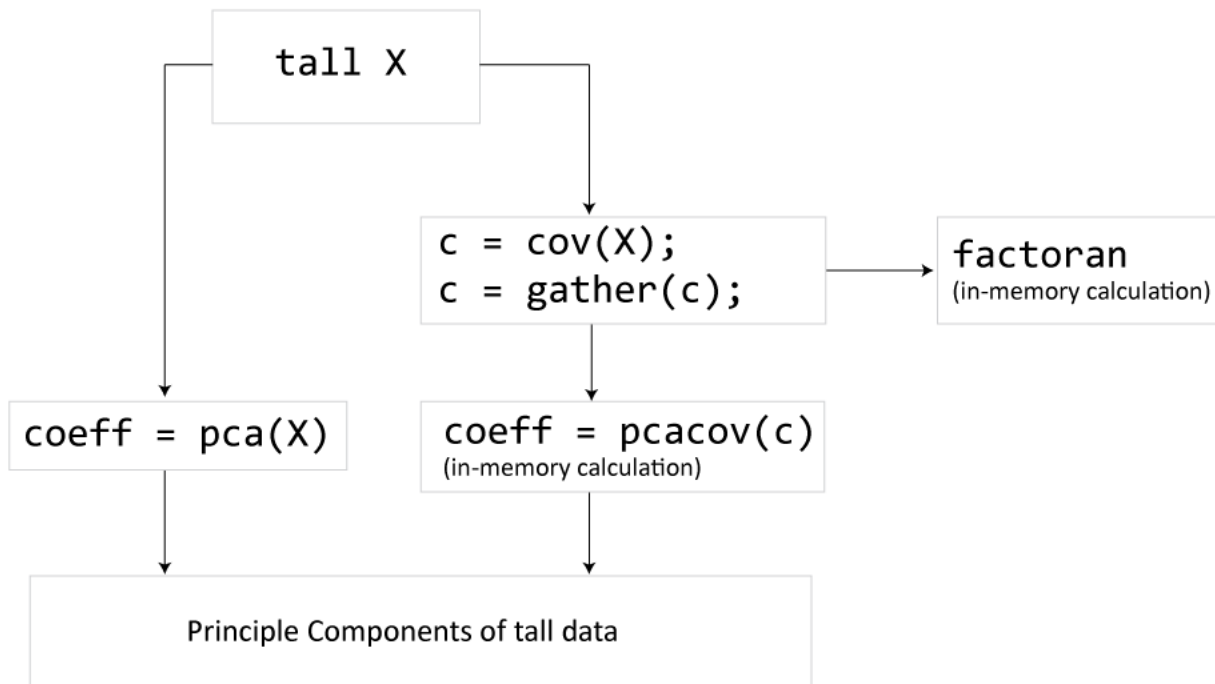
## Extended Capabilities

### Tall Arrays

Calculate with arrays that have more rows than fit in memory.

`pcacov` and `factoran` do *not* work directly on tall arrays. Instead, use `C = gather(cov(X))` to compute the covariance matrix of a tall array. Then, you can use

`pcacov` or `factoran` on the in-memory covariance matrix. Alternatively, you can use `pca` directly on a tall array.



For more information, see “Tall Arrays” (MATLAB).

## See Also

`barttest` | `biplot` | `factoran` | `pca` | `pcares` | `rotatefactors`

Introduced before R2006a

## pcares

Residuals from principal component analysis

### Syntax

```
residuals = pcares(X, ndim)
[residuals, reconstructed] = pcares(X, ndim)
```

### Description

`residuals = pcares(X, ndim)` returns the residuals obtained by retaining `ndim` principal components of the `n`-by-`p` matrix `X`. Rows of `X` correspond to observations, columns to variables. `ndim` is a scalar and must be less than or equal to `p`. `residuals` is a matrix of the same size as `X`. Use the data matrix, *not* the covariance matrix, with this function.

`pcares` does not normalize the columns of `X`. To perform the principal components analysis based on standardized variables, that is, based on correlations, use `pcares(zscore(X), ndim)`. You can perform principal components analysis directly on a covariance or correlation matrix, but without constructing residuals, by using `pcacov`.

`[residuals, reconstructed] = pcares(X, ndim)` returns the reconstructed observations; that is, the approximation to `X` obtained by retaining its first `ndim` principal components.

### Examples

This example shows the drop in the residuals from the first row of the Hald data as the number of component dimensions increases from one to three.

```
load hald
r1 = pcares(ingredients, 1);
r2 = pcares(ingredients, 2);
```

```
r3 = pcares(ingredients, 3);  
  
r11 = r1(1, :)  
r11 =  
    2.0350  2.8304 -6.8378  3.0879  
  
r21 = r2(1, :)  
r21 =  
   -2.4037  2.6930 -1.6482  2.3425  
  
r31 = r3(1, :)  
r31 =  
    0.2008  0.1957  0.2045  0.1921
```

## References

- [1] Jackson, J. E., *A User's Guide to Principal Components*, John Wiley and Sons, 1991.
- [2] Jolliffe, I. T., *Principal Component Analysis*, 2nd Edition, Springer, 2002.
- [3] Krzanowski, W. J. *Principles of Multivariate Analysis: A User's Perspective*. New York: Oxford University Press, 1988.
- [4] Seber, G. A. F. *Multivariate Observations*. Hoboken, NJ: John Wiley & Sons, Inc., 1984.

## See Also

`factoran` | `pca` | `pcacov`

Introduced before R2006a

## ppca

Probabilistic principal component analysis

### Syntax

```
[coeff, score, pcvar] = ppca(Y, K)
[coeff, score, pcvar] = ppca(Y, K, Name, Value)
[coeff, score, pcvar, mu] = ppca(____)
[coeff, score, pcvar, mu, v, S] = ppca(____)
```

### Description

`[coeff, score, pcvar] = ppca(Y, K)` returns the principal component coefficients for the  $n$ -by- $p$  data matrix  $Y$  based on a probabilistic principal component analysis on page 32-4635 (PPCA). It also returns the principal component scores, which are the representations of  $Y$  in the principal component space, and the principal component variances, which are the eigenvalues of the covariance matrix of  $Y$ , in `pcvar`.

Each column of `coeff` contains coefficients for one principal component, and the columns are in descending order of component variance. Rows of `score` correspond to observations, and columns correspond to components. Rows of  $Y$  correspond to observations and columns correspond to variables.

Probabilistic principal component analysis might be preferable to other algorithms that handle missing data, such as the alternating least squares algorithm when any data vector has one or more missing values. It assumes that the values are missing at random through the data set. An expectation-maximization algorithm is used for both complete and missing data.

`[coeff, score, pcvar] = ppca(Y, K, Name, Value)` returns the principal component coefficients, scores, and variances using additional options for computation and handling of special data types, specified by one or more `Name, Value` pair arguments.

For example, you can introduce initial values for the residual variance, `v`, or change the termination criteria.



`[coeff, score, pcv, mu] = ppca ( ___ )` also returns the estimated mean of each variable in  $Y$ . You can use any of the input arguments in the previous syntaxes.

`[coeff, score, pcv, mu, v, S] = ppca ( ___ )` also returns the isotropic residual variance in  $v$  and the final results at convergence in structure  $S$ .

## Examples

### Perform Probabilistic Principal Component Analysis

Load the sample data.

```
load fisheriris
```

The double matrix `meas` consists of four types of measurements on the flowers, which, respectively, are the length and width of sepals and petals.

Introduce missing values randomly.

```
y = meas;
rng('default'); % for reproducibility
ix = random('unif',0,1,size(y))<0.20;
y(ix) = NaN;
```

Now, approximately 20% of the data is missing, indicated by `NaN`.

Perform probabilistic principal component analysis and request the component coefficients and variances.

```
[coeff, score, pcv, mu] = ppca(y,3);
coeff
```

```
coeff =
```

```
    0.3562    0.6709   -0.5518
   -0.0765    0.7120    0.6332
    0.8592   -0.1597    0.0596
    0.3592   -0.1318    0.5395
```

```
pcv
```

```
pcvar =  
  
    4.0914  
    0.2125  
    0.0617
```

Perform principal component analysis using the alternating least squares algorithm and request the component coefficients and variances.

```
[coeff2,score2,pcvar2,mu2] = pca(y,'algorithm','als',...  
'NumComponents',3);  
coeff2
```

```
coeff2 =  
  
    0.3376    0.4952    0.7406  
   -0.0731    0.8609   -0.4476  
    0.8657   -0.1168   -0.1233  
    0.3623   -0.0086   -0.4857
```

```
pcvar2
```

```
pcvar2 =  
  
    4.0733  
    0.2652  
    0.1222
```

The coefficients and the variances of the first two principal components are similar.

Another way to compare the results is to find the angle between the two spaces spanned by the coefficient vectors.

```
subspace(coeff,coeff2)
```

```
ans =  
  
    0.0884
```

The angle between the two spaces is pretty small. This indicates that these two results are close to each other.

### Change the Termination Criteria for Probabilistic Principal Component Analysis

Load the sample data set.

```
load imports-85
```

Data matrix  $X$  has 13 continuous variables in columns 3 to 15: wheel-base, length, width, height, curb-weight, engine-size, bore, stroke, compression-ratio, horsepower, peak-rpm, city-mpg, and highway-mpg. The variables bore and stroke are missing four values in rows 56 to 59, and the variables horsepower and peak-rpm are missing two values in rows 131 and 132.

Perform probabilistic principal component analysis and display the first three principal components.

```
[coeff,score,pcvar] = ppca(X(:,3:15),3);
```

```
Warning: Maximum number of iterations 1000 reached.
```

Change the termination tolerance for the cost function to 0.01.

```
opt = statset('ppca');  
opt.TolFun = 0.01;
```

Perform probabilistic principal component analysis.

```
[coeff,score,pcvar] = ppca(X(:,3:15),3,'Options',opt);
```

```
Warning: Maximum number of iterations 1000 reached.
```

ppca now terminates before the maximum number of iterations is reached because it meets the tolerance for the cost function.

### Reconstruct Observations

Load the sample data.

```
load hald
y = ingredients;
```

The ingredients data has 13 observations for 4 variables.

Introduce missing values to the data.

```
y(2:16:end) = NaN;
```

Every 16th value is NaN. This corresponds to 7.69% of the data.

Find the first three principal components of data using PPCA and display the reconstructed observations.

```
[coeff,score,pcvar,mu,v,S] = ppca(y,3);
S.Recon
```

```
Warning: Maximum number of iterations 1000 reached.
```

```
ans =

    6.8536    25.8700    5.8389    59.8730
    1.0433    28.9710   14.9654    51.9738
   11.5770    56.5067    8.6352    20.5076
   11.0835    31.0722    8.0920    47.0748
    7.0679    52.2556    6.0748    33.0598
   11.0486    55.0430    9.0534    22.0423
    2.8493    70.8691   16.8339    5.8656
    1.0333    31.0281   19.6907    44.0306
    2.0400    54.0354   18.0440    22.0349
   20.7822    46.8091    3.7603    25.8081
    0.9539    39.9591   22.9493    31.1542
   10.8250    65.8475    8.8072    11.8445
    9.9172    67.9285    7.9088    11.9257
```

You can also reconstruct the observations using the principal components and the estimated mean.

```
t = score*coeff' + repmat(mu,13,1);
```

## Results at Convergence

Load the data.

```
load hald
```

Here, `ingredients` is a real-valued matrix of predictor variables.

Perform the probabilistic principal components analysis and display coefficients.

```
[coeff,score,pcvariance,mu,v,S] = ppca(ingredients,3);  
coeff
```

```
Warning: Maximum number of iterations 1000 reached.
```

```
coeff =
```

```
-0.0693   -0.6459    0.5673  
-0.6786   -0.0184   -0.5440  
 0.0308    0.7552    0.4036  
 0.7306   -0.1102   -0.4684
```

Display the algorithm results at convergence of the PPCA.

```
S
```

```
S =
```

```
struct with fields:
```

```
    W: [4x3 double]  
  Xexp: [13x3 double]  
 Recon: [13x4 double]  
    v: 0.2372  
 NumIter: 1000  
 RMSResid: 0.2340  
  nloglk: 149.3388
```

Display the matrix `W`.

```
S.W
```

```
ans =  
  
    0.5624    2.0279    5.4075  
    4.8320   -10.3894    5.9202  
   -3.7521   -3.0555   -4.1552  
   -1.5144   11.7122   -7.2564
```

Orthogonalizing  $W$  recovers the coefficients.

```
orth(S.W)  
  
ans =  
  
   -0.0693    0.6459    0.5673  
   -0.6786    0.0184   -0.5440  
    0.0308   -0.7552    0.4036  
    0.7306    0.1102   -0.4684
```

## Input Arguments

### **$Y$** — Input data

*n*-by-*p* matrix

Input data for which to compute the principal components, specified as an *n*-by-*p* matrix. Rows of  $Y$  correspond to observations and columns correspond to variables.

Data Types: `single` | `double`

### **$k$** — Number of principal components

positive integer value less than rank

Number of principal components to return, specified as an integer value less than the rank of data. The maximum possible rank is  $\min(n,p)$ , where *n* is the number of observations and *p* is the number of variables. However, if the data is correlated, the rank might be smaller than  $\min(n,p)$ .

`ppca` orders the components based on their variance.

If  $K$  is  $\min(n,p)$ , ppca sets  $K$  equal to  $\min(n,p) - 1$ , and 'w0' is truncated to  $\min(p,n) - 1$  columns if you specify a  $p$ -by- $p$  w0 matrix.

For example, you can request only the first three components, based on the component variance as follows.

Example: `coeff = ppca(Y,3)`

Data Types: `single` | `double`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: 'w0', `init`, 'Options', `opt` specifies that the initial values for 'w0' are in matrix `init` and ppca uses the options defined by `opt`.

### w0 — Initial value of $W$

matrix of random values (default) |  $p$ -by- $k$  matrix

Initial value of  $W$  in the probabilistic principal component analysis on page 32-4635 algorithm, specified as a comma-separated pair consisting of 'w0' and a  $p$ -by- $k$  matrix.

Data Types: `single` | `double`

### v0 — Initial value of residual variance

random number (default) | positive scalar value

Initial value of residual variance, specified as the comma-separated pair consisting of 'v0' and a positive scalar value.

Data Types: `single` | `double`

### Options — Options for iterations

structure

Options for the iterations, specified as a comma-separated pair 'Options' and a structure created by the `statset` function. ppca uses the following fields in the options structure.

'Display'	Level of display output. Choices are 'off', 'final', and 'iter'.
'MaxIter'	Maximum number of steps allowed. The default is 1000. Unlike in optimization settings, reaching the <code>MaxIter</code> value is regarded as convergence.
'TolFun'	Positive integer stating the termination tolerance for the cost function. The default is 1e-6.
'TolX'	Positive integer stating the convergence threshold for the relative change in the elements of $W$ . The default is 1e-6.

You can change the values of these fields and specify the new structure in `ppca` using the 'Options' name-value pair argument.

```
Example: opt = statset('ppca'); opt.MaxIter = 2000; coeff = ppca(Y,
3, 'Options', opt);
```

Data Types: `struct`

## Output Arguments

### **coeff** — Principal component coefficients

*p*-by-*k* matrix

Principal component coefficients, returned as a *p*-by-*k* matrix. Each column of `coeff` contains coefficients for one principal component. The columns are in the order of descending component variance, `pcvar`.

### **score** — Principal component scores

*n*-by-*k* matrix

Principal component scores, returned as an *n*-by-*k* matrix. Rows of `score` correspond to observations, and columns correspond to components.

### **pcvar** — Principal component variances

column vector

Principal component variances, which are the eigenvalues of the covariance matrix of  $Y$ , returned as a column vector.



**$\mu$  — Estimated mean**

row vector

Estimated mean of each variable in  $Y$ , returned as a row vector.

 **$\nu$  — Isotropic residual variance**

scalar value

Isotropic residual variance, returned as a scalar value.

 **$s$  — Final results at convergence**

structure

Final results at convergence, returned as a structure containing the following fields.

$W$	$W$ at convergence.
$X_{exp}$	Conditional expectation of the estimated latent variable $x$ .
$Recon$	Reconstructed observations using $k$ principal components. This is a low dimension approximation of the input data $Y$ , and is equal to $\mu + score * coeff'$ .
$\nu$	Residual variance.
$RMSResid$	Root mean square of residuals.
$NumIter$	Number of iteration counts.
$nloglk$	Negative loglikelihood function value.

## Definitions

### Probabilistic Principal Component Analysis

Probabilistic principal component analysis (PPCA) is a method to estimate the principal axes when any data vector has one or more missing values.

PPCA is based on an isotropic error model. It seeks to relate a  $p$ -dimensional observation vector  $y$  to a corresponding  $k$ -dimensional vector of latent (or unobserved) variable  $x$ , which is normal with mean zero and covariance  $I(k)$ . The relationship is

$$y^T = W * x^T + \mu + \varepsilon,$$

where  $y$  is the row vector of observed variable,  $x$  is the row vector of latent variables, and  $\varepsilon$  is the isotropic error term.  $\varepsilon$  is Gaussian with mean zero and covariance of  $v * I(k)$ , where  $v$  is the residual variance. Here,  $k$  needs to be smaller than the rank for the residual variance to be greater than 0 ( $v > 0$ ). Standard principal component analysis, where the residual variance is zero, is the limiting case of PPCA. The observed variables,  $y$ , are conditionally independent given the values of the latent variables,  $x$ . So, the latent variables explain the correlations between the observation variables and the error explains the variability unique to a particular  $y_i$ . The  $p$ -by- $k$  matrix  $W$  relates the latent and observation variables, and the vector  $\mu$  permits the model to have a nonzero mean. PPCA assumes that the values are missing at random through the data set. This means that whether a data value is missing or not does not depend on the latent variable given the observed data values.

Under this model,

$$y \sim N(\mu, W * W^T + v * I(k)).$$

There is no closed-form analytical solution for  $W$  and  $v$ , so their estimates are determined by iterative maximization of the corresponding loglikelihood using an expectation-maximization (EM) algorithm. This EM algorithm handles missing values by treating them as additional latent variables. At convergence, the columns of  $W$  spans the subspace, but they are not orthonormal. `ppca` obtains the orthonormal coefficients, `coeff`, for the components by orthogonalization of  $W$ .

## References

- [1] Tipping, M. E., and C. M. Bishop. Probabilistic Principal Component Analysis. *Journal of the Royal Statistical Society. Series B (Statistical Methodology)*, Vol. 61, No.3, 1999, pp. 611–622.
- [2] Roweis, S. “EM Algorithms for PCA and SPCA.” *In Proceedings of the 1997 Conference on Advances in Neural Information Processing Systems*. Vol.10 (NIPS 1997), Cambridge, MA, USA: MIT Press, 1998, pp. 626–632.
- [3] Ilin, A., and T. Raiko. “Practical Approaches to Principal Component Analysis in the Presence of Missing Values.” *J. Mach. Learn. Res.*. Vol. 11, August, 2010, pp. 1957–2000.

## See Also

`barttest` | `biplot` | `canoncorr` | `factoran` | `pca` | `pcacov` | `pcares` |  
`rotatefactors`

**Introduced in R2013a**

## ComponentProportion property

**Class:** gmdistribution

Input vector of mixing proportions

### Description

Optional input vector of mixing proportions  $p$ , or its default value.

# pdf

Probability density functions

## Syntax

```
y = pdf('name', x, A)
y = pdf('name', x, A, B)
y = pdf('name', x, A, B, C)
y = pdf('name', x, A, B, C, D)

y = pdf(pd, x)
```

## Description

`y = pdf('name', x, A)` returns the probability density function (pdf) for the one-parameter distribution family specified by 'name', evaluated at the values in `x`. `A` contains the parameter value for the distribution.

`y = pdf('name', x, A, B)` returns the pdf for the two-parameter distribution family specified by 'name', evaluated at the values in `x`. `A` and `B` contain the parameter values for the distribution.

`y = pdf('name', x, A, B, C)` returns the pdf for the three-parameter distribution family specified by 'name', evaluated at the values in `x`. `A`, `B`, and `C` contain the parameter values for the distribution.

`y = pdf('name', x, A, B, C, D)` returns the pdf for the four-parameter distribution family specified by 'name', evaluated at the values in `x`. `A`, `B`, `C`, and `D` contain the parameter values for the distribution.

`y = pdf(pd, x)` returns the probability density function of the probability distribution object, `pd`, evaluated at the values in `x`.

## Examples

### Compute the Normal Distribution pdf

Create a standard normal distribution object with the mean  $\mu$  equal to 0 and the standard deviation  $\sigma$  equal to 1.

```
mu = 0;  
sigma = 1;  
pd = makedist('Normal',mu,sigma);
```

Define the input vector  $x$  to contain the values at which to calculate the pdf.

```
x = [-2 -1 0 1 2];
```

Compute the pdf values for the standard normal distribution at the values in  $x$ .

```
y = pdf(pd,x)
```

```
y =  
    0.0540    0.2420    0.3989    0.2420    0.0540
```

Each value in  $y$  corresponds to a value in the input vector  $x$ . For example, at the value  $x$  equal to 1, the corresponding pdf value  $y$  is equal to 0.2420.

Alternatively, you can compute the same pdf values without creating a probability distribution object. Use the `pdf` function, and specify a standard normal distribution using the same parameter values for  $\mu$  and  $\sigma$ .

```
y2 = pdf('Normal',x,mu,sigma)
```

```
y2 =  
    0.0540    0.2420    0.3989    0.2420    0.0540
```

The pdf values are the same as those computed using the probability distribution object.

## Compute the Poisson Distribution pdf

Create a Poisson distribution object with the rate parameter,  $\lambda$ , equal to 2.

```
lambda = 2;  
pd = makedist('Poisson', lambda);
```

Define the input vector  $x$  to contain the values at which to calculate the pdf.

```
x = [0 1 2 3 4];
```

Compute the pdf values for the Poisson distribution at the values in  $x$ .

```
y = pdf(pd, x)
```

```
y =
```

```
    0.1353    0.2707    0.2707    0.1804    0.0902
```

Each value in  $y$  corresponds to a value in the input vector  $x$ . For example, at the value  $x$  equal to 3, the corresponding pdf value in  $y$  is equal to 0.1804.

Alternatively, you can compute the same pdf values without creating a probability distribution object. Use the `pdf` function, and specify a Poisson distribution using the same value for the rate parameter,  $\lambda$ .

```
y2 = pdf('Poisson', x, lambda)
```

```
y2 =
```

```
    0.1353    0.2707    0.2707    0.1804    0.0902
```

The pdf values are the same as those computed using the probability distribution object.

## Input Arguments

**'name'** — Probability distribution name

valid probability distribution name

Probability distribution name, specified as one of the following.

name	Distribution	Input Parameter A	Input Parameter B	Input Parameter C	Input Parameter D
'Beta '	“Beta Distribution” on page B-4	$\alpha$ : first shape parameter	$b$ : second shape parameter	—	—
'Binomial '	“Binomial Distribution” on page B-9	$n$ : number of trials	$p$ : probability of success for each trial	—	—
'BirnbauSaunders '	“Birnbau-Saunders Distribution” on page B-13	$\beta$ : scale parameter	$\gamma$ : shape parameter	—	—
'Burr '	“Burr Type XII Distribution” on page B-15	$\alpha$ : scale parameter	$c$ : first shape parameter	$k$ : second shape parameter	—
'Chisquare '	“Chi-Square Distribution” on page B-28	$\nu$ : degrees of freedom	—	—	—
'Exponential '	“Exponential Distribution” on page B-34	$\mu$ : mean	—	—	—
'Extreme Value '	“Extreme Value Distribution” on page B-38	$\mu$ : location parameter	$\sigma$ : scale parameter	—	—
'F '	“F Distribution” on page B-44	$\nu_1$ : numerator degrees of freedom	$\nu_2$ : denominator degrees of freedom	—	—
'Gamma '	“Gamma Distribution” on page B-47	$\alpha$ : shape parameter	$b$ : scale parameter	—	—
'Generalized Extreme Value '	“Generalized Extreme Value Distribution” on page B-53	$k$ : shape parameter	$\sigma$ : scale parameter	$\mu$ : location parameter	—



name	Distribution	Input Parameter A	Input Parameter B	Input Parameter C	Input Parameter D
'Generalized Pareto'	“Generalized Pareto Distribution” on page B-58	$k$ : tail index (shape) parameter	$\sigma$ : scale parameter	$\mu$ : threshold (location) parameter	—
'Geometric'	“Geometric Distribution” on page B-64	$p$ : probability parameter	—	—	—
'HalfNormal'	“Half-Normal Distribution” on page B-73	$\mu$ : location parameter	$\sigma$ : scale parameter	—	—
'Hypergeometric'	“Hypergeometric Distribution” on page B-79	$m$ : size of the population	$k$ : number of items with the desired characteristic in the population	$n$ : number of samples drawn	—
'InverseGaussian'	“Inverse Gaussian Distribution” on page B-82	$\mu$ : scale parameter	$\lambda$ : shape parameter	—	—
'Logistic'	“Logistic Distribution” on page B-96	$\mu$ : mean	$\sigma$ : scale parameter	—	—
'LogLogistic'	“Loglogistic Distribution” on page B-98	$\mu$ : log mean	$\sigma$ : log scale parameter	—	—
'Lognormal'	“Lognormal Distribution” on page B-100	$\mu$ : log mean	$\sigma$ : log standard deviation	—	—
'Nakagami'	“Nakagami Distribution” on page B-118	$\mu$ : shape parameter	$\omega$ : scale parameter	—	—
'Negative Binomial'	“Negative Binomial Distribution” on page B-120	$r$ : number of successes	$p$ : probability of success in a single trial	—	—

name	Distribution	Input Parameter A	Input Parameter B	Input Parameter C	Input Parameter D
'Noncentral F'	“Noncentral F Distribution” on page B-128	$\nu_1$ : numerator degrees of freedom	$\nu_2$ : denominator degrees of freedom	$\delta$ : noncentrality parameter	—
'Noncentral t'	“Noncentral t Distribution” on page B-131	$\nu$ : degrees of freedom	$\delta$ : noncentrality parameter	—	—
'Noncentral Chi-square'	“Noncentral Chi-Square Distribution” on page B-125	$\nu$ : degrees of freedom	$\delta$ : noncentrality parameter	—	—
'Normal'	“Normal Distribution” on page B-134	$\mu$ : mean	$\sigma$ : standard deviation	—	—
'Poisson'	“Poisson Distribution” on page B-143	$\lambda$ : mean	—	—	—
'Rayleigh'	“Rayleigh Distribution” on page B-146	$b$ : scale parameter	—	—	—
'Rician'	“Rician Distribution” on page B-149	$s$ : noncentrality parameter	$\sigma$ : scale parameter	—	—
'Stable'	“Stable Distribution” on page B-151	$\alpha$ : first shape parameter	$\beta$ : second shape parameter	$\gamma$ : scale parameter	$\delta$ : location parameter
'T'	“Student's t Distribution” on page B-166	$\nu$ : degrees of freedom	—	—	—
'tLocationScale'	“t Location-Scale Distribution” on page B-175	$\mu$ : location parameter	$\sigma$ : scale parameter	$\nu$ : shape parameter	—
'Uniform'	“Uniform Distribution (Continuous)” on page B-184	$a$ : lower endpoint (minimum)	$b$ : upper endpoint (maximum)	—	—

name	Distribution	Input Parameter A	Input Parameter B	Input Parameter C	Input Parameter D
'Discrete Uniform'	“Uniform Distribution (Discrete)” on page B-190	$n$ : maximum observable value	—	—	—
'Weibull'	“Weibull Distribution” on page B-193	$a$ : scale parameter	$b$ : shape parameter	—	—

### **x** — Values at which to evaluate pdf

scalar value | array of scalar values

Values at which to evaluate the pdf, specified as a scalar value, or an array of scalar values.

- If  $x$  is a scalar value, and if you specify distribution parameters A, B, C, or D as arrays, then `cdf` expands  $x$  into a constant array of the same size as the parameters.
- If  $x$  is an array, and if you specify distribution parameters A, B, C, or D as arrays, then  $x$ , A, B, C, and D must all be the same size.

Example: `[0.1,0.25,0.5,0.75,0.9]`

Data Types: `single` | `double`

### **A** — First probability distribution parameter

scalar value | array of scalar values

First probability distribution parameter, specified as a scalar value, or an array of scalar values.

If  $x$  and A are arrays, they must be the same size. If  $x$  is a scalar, then `cdf` expands it into a constant matrix the same size as A. If A is a scalar, then `cdf` expands it into a constant matrix the same size as  $x$ .

Data Types: `single` | `double`

### **B** — Second probability distribution parameter

scalar value | array of scalar values

Second probability distribution parameter, specified as a scalar value, or an array of scalar values.

If  $x$ ,  $A$ , and  $B$  are arrays, they must be the same size. If  $x$  is a scalar, then `cdf` expands it into a constant matrix the same size as  $A$  and  $B$ . If  $A$  or  $B$  are scalars, then `cdf` expands them into constant matrices the same size as  $x$ .

Data Types: `single` | `double`

### **C** — Third probability distribution parameter

scalar value | array of scalar values

Third probability distribution parameter, specified as a scalar value, or an array of scalar values.

If  $x$ ,  $A$ ,  $B$ , and  $C$  are arrays, they must be the same size. If  $x$  is a scalar, then `cdf` expands it into a constant matrix the same size as  $A$ ,  $B$ , and  $C$ . If any of  $A$ ,  $B$  or  $C$  are scalars, then `cdf` expands them into constant matrices the same size as  $x$ .

Data Types: `single` | `double`

### **D** — Fourth probability distribution parameter

scalar value | array of scalar values

Fourth probability distribution parameter, specified as a scalar value, or an array of scalar values.

If  $x$ ,  $A$ ,  $B$ ,  $C$ , and  $D$  are arrays, they must be the same size. If  $x$  is a scalar, then `cdf` expands it into a constant array the same size as  $A$ ,  $B$ ,  $C$ , and  $D$ . If any of  $A$ ,  $B$ ,  $C$ , or  $D$  are scalars, then `cdf` expands them into constant matrices the same size as  $x$ .

Data Types: `single` | `double`

### **pd** — Probability distribution

probability distribution object

Probability distribution, specified as a probability distribution object created using one of the following.

<code>makedist</code>	Create a probability distribution object using specified parameter values.
-----------------------	--

<code>fitdist</code>	Fit a probability distribution object to sample data.
<code>dfittool</code>	Fit a probability distribution object to sample data using the interactive Distribution Fitting app.
<code>paretotails</code>	Create a Pareto tails object.
<code>gmdistribution</code>	Create a Gaussian mixture distribution object.

## Output Arguments

### **y** — Probability density function

array

Probability density function of the specified probability distribution, returned as an array.

- If you specify distribution parameters A, B, C, or D, then **y** is the same size as **x**, A, B, C, and D after any necessary scalar expansion.
- If you specify a probability distribution object, **pd**, then **y** has the same dimensions as **x**.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`cdf` | `icdf` | `mle` | `random`

Introduced before R2006a

## pdf

**Class:** `gmdistribution`

Probability density function for Gaussian mixture distribution

## Syntax

```
y = pdf(obj,X)
```

## Description

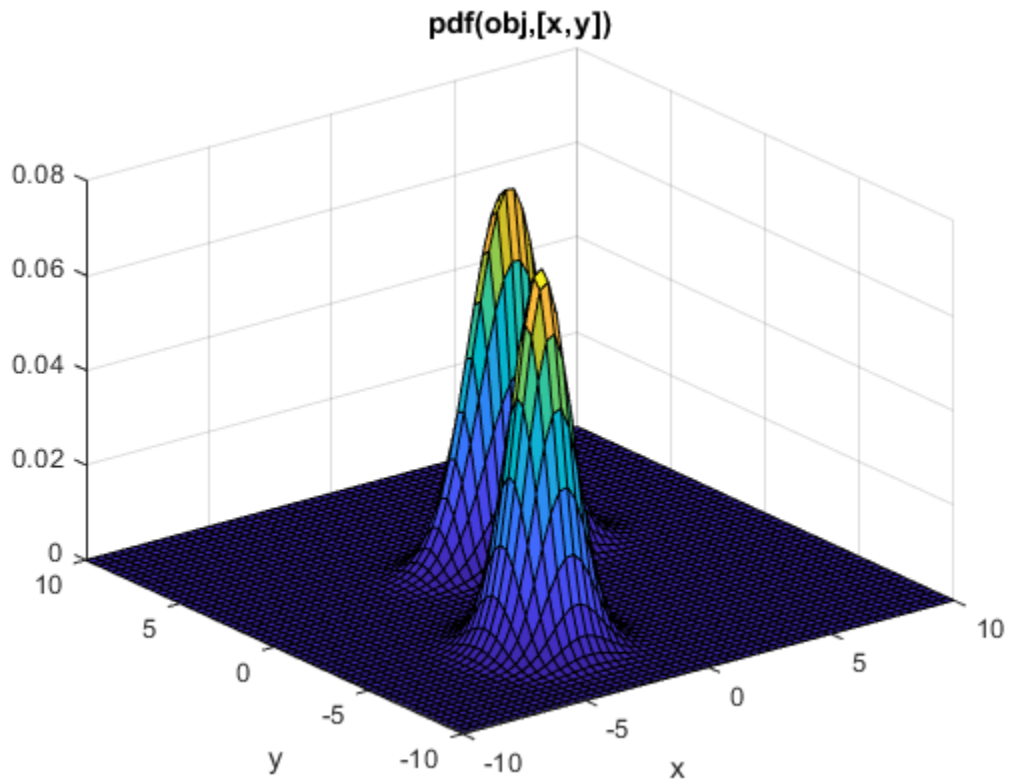
`y = pdf(obj,X)` returns a vector `y` of length  $n$  containing the values of the probability density function (pdf) for the `gmdistribution` object `obj`, evaluated at the  $n$ -by- $d$  data matrix `X`, where  $n$  is the number of observations and  $d$  is the dimension of the data. `obj` is an object created by `gmdistribution` or `fitgmdist`. `y(I)` is the pdf of observation `I`.

## Examples

### Construct a Gaussian Mixture Distribution

Create a `gmdistribution` distribution defining a two-component mixture of bivariate Gaussian distributions.

```
mu = [1 2;-3 -5];  
sigma = cat(3,[2 0;0 .5],[1 0;0 1]);  
p = ones(1,2)/2;  
obj = gmdistribution(mu,sigma,p);  
  
ezsurf(@(x,y)pdf(obj,[x y]),[-10 10],[-10 10])
```



## See Also

[cdf](#) | [fitgmdist](#) | [gmdistribution](#) | [mvnpdf](#)

## pdf

**Class:** `piecewisedistribution`

Probability density function for piecewise distribution

## Syntax

```
P = pdf(obj, X)
```

## Description

`P = pdf(obj, X)` returns an array `P` of values of the probability density function for the piecewise distribution object `obj`, evaluated at the values in the array `X`.

---

**Note** For a Pareto tails object, the pdf is computed using the generalized Pareto distribution in the tails. In the center, the pdf is computed using the slopes of the cdf, which are interpolated between a set of discrete values. Therefore the pdf in the center is piecewise constant. It is noisy for a `cdf` specified in `paretotails` via the `'ecdf'` option, and somewhat smoother for the `'kernel'` option, but generally not a good estimate of the underlying density of the original data.

---

## Examples

Fit Pareto tails to a  $t$  distribution at cumulative probabilities 0.1 and 0.9:

```
t = trnd(3,100,1);
obj = paretotails(t,0.1,0.9);
[p,q] = boundary(obj)
p =
    0.1000
    0.9000
q =
   -1.7766
    1.8432
```



```
pdf(obj,q)
ans =
    0.2367
    0.1960
```

## See Also

[cdf](#) | [paretotails](#)

## pdf

**Class:** ProbDist

Return probability density function (PDF) for ProbDist object

## Syntax

$Y = \text{pdf}(PD, X)$

## Description

$Y = \text{pdf}(PD, X)$  returns  $Y$ , an array containing the probability density function (PDF) for the ProbDist object  $PD$ , evaluated at values in  $X$ .

## Input Arguments

$PD$	An object of the class ProbDistUnivParam or ProbDistUnivKernel.
$X$	A numeric array of values where you want to evaluate the PDF.

## Output Arguments

$Y$	An array containing the probability density function (PDF) for the ProbDist object $PD$ .
-----	---

## See Also

pdf

# pdf

**Class:** prob.TruncatableDistribution

**Package:** prob

Probability density function of probability distribution object

## Syntax

$y = \text{pdf}(\text{pd}, x)$

## Description

$y = \text{pdf}(\text{pd}, x)$  returns the probability density function (pdf) of the continuous probability distribution `pd` at the values in `x`. For discrete distributions, `pdf` returns the probability mass function.

## Input Arguments

**pd** — Probability distribution

probability distribution object

Probability distribution, specified as a probability distribution object. Create a probability distribution object with specified parameter values using `makedist`. Alternatively, for fittable distributions, create a probability distribution object by fitting it to data using `fitdist` or the Distribution Fitting app.

**x** — Values at which to calculate pdf

array

Values at which to calculate pdf, specified as an array.

Data Types: `single` | `double`

## Output Arguments

**y** — Probability density function

array

Probability density function of `pd`, evaluated at the values in data vector `x`, returned as a array. `y` has the same dimensions as input `x`.

## Examples

### Plot the pdf of a Standard Normal Distribution

Create a standard normal distribution object.

```
pd = makedist('Normal')
```

```
pd =
```

```
NormalDistribution
```

```
Normal distribution
```

```
mu = 0
```

```
sigma = 1
```

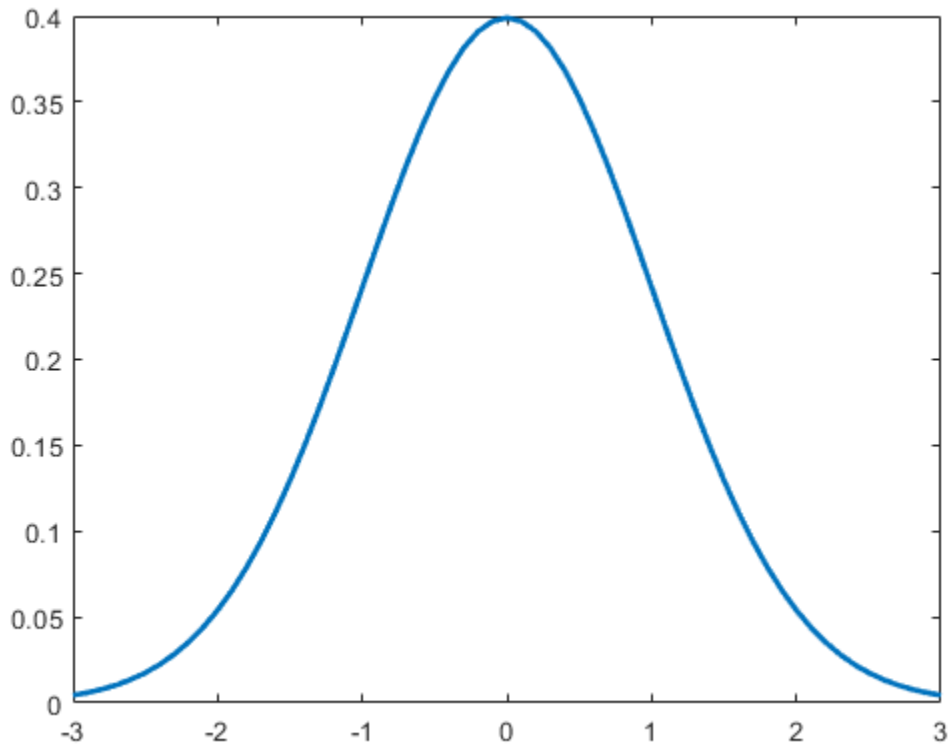
Specify the `x` values and compute the pdf.

```
x = -3:.1:3;
```

```
pdf_normal = pdf(pd,x);
```

Plot the pdf.

```
plot(x,pdf_normal,'LineWidth',2)
```



### Plot the pdf of a Weibull Distribution

Create a Weibull probability distribution object.

```
pd = makedist('Weibull', 'a', 5, 'b', 2)
```

```
pd =
```

```
WeibullDistribution
```

```
Weibull distribution
```

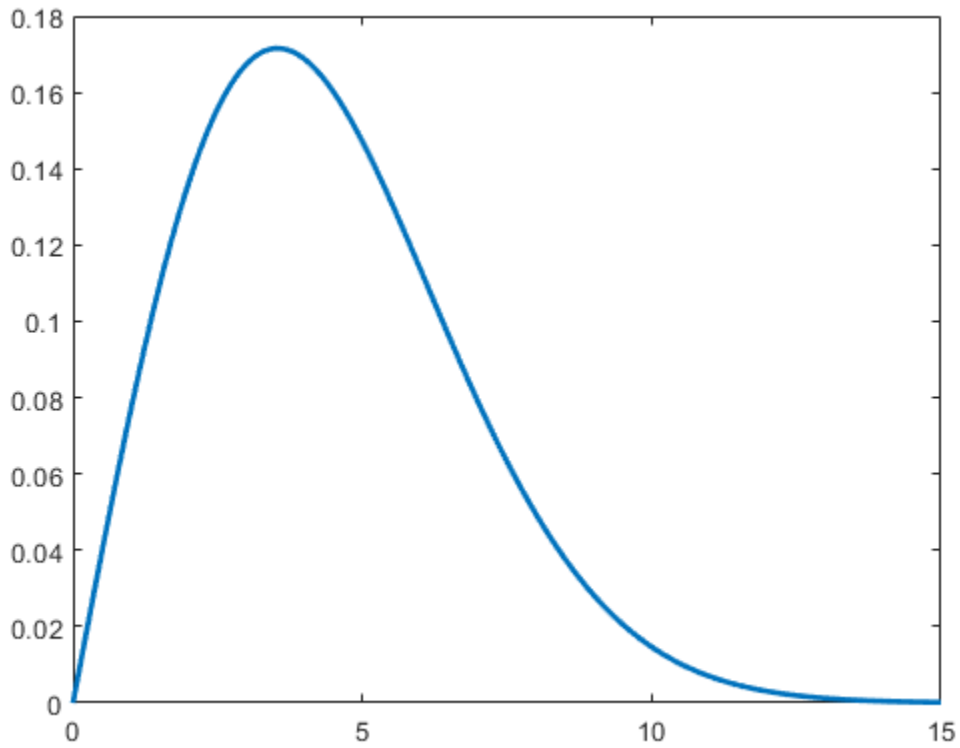
```
A = 5  
B = 2
```

Specify the  $x$  values and compute the pdf.

```
x = 0:.1:15;  
y = pdf(pd,x);
```

Plot the pdf.

```
plot(x,y,'LineWidth',2)
```



## See Also

[cdf](#) | [distributionFitter](#) | [fitdist](#) | [icdf](#) | [makedist](#) | [pdf](#)

## pdist

Pairwise distance between pairs of observations

### Syntax

```
D = pdist(X)
D = pdist(X,Distance)
D = pdist(X,Distance,DistParameter)
```

### Description

`D = pdist(X)` returns the Euclidean distance between pairs of observations in `X`.

`D = pdist(X,Distance)` returns the distance by using the method specified by `Distance`.

`D = pdist(X,Distance,DistParameter)` returns the distance by using the method specified by `Distance` and `DistParameter`. You can specify `DistParameter` only when `Distance` is `'seuclidean'`, `'minkowski'`, or `'mahalanobis'`.

### Examples

#### Compute Euclidean Distance

Compute the Euclidean distance between pairs of observations, and format the distance matrix using `squareform`.

Create a matrix with three observations and two variables.

```
rng('default') % For reproducibility
X = rand(3,2);
```

Compute the Euclidean distance.



```
D = pdist(X)
```

```
D =
```

```
    0.2954    1.0670    0.9448
```

The pairwise distances are arranged in the order (2,1), (3,1), (3,2). You can easily locate the distance between observations  $i$  and  $j$  by using `squareform`.

```
Z = squareform(D)
```

```
Z =
```

```

      0    0.2954    1.0670
    0.2954    0    0.9448
    1.0670    0.9448    0

```

`squareform` returns a symmetric matrix where  $Z(i,j)$  corresponds to the pairwise distance between observations  $i$  and  $j$ . For example, you can find the distance between observations 2 and 3.

```
Z(2,3)
```

```
ans =
```

```
    0.9448
```

## Compute Minkowski Distance

Create a matrix with three observations and two variables.

```
rng('default') % For reproducibility
X = rand(3,2);
```

Compute the Minkowski distance with the default exponent 2.

```
D1 = pdist(X, 'minkowski')
```

```
D1 =  
  
    0.2954    1.0670    0.9448
```

Compute the Minkowski distance with an exponent of 1, which is equal to the city block distance.

```
D2 = pdist(X, 'minkowski', 1)  
D3 = pdist(X, 'cityblock')
```

```
D2 =  
  
    0.3721    1.5036    1.3136
```

```
D3 =  
  
    0.3721    1.5036    1.3136
```

### Compute Pairwise Distance with Missing Elements Using a Custom Distance Function

Define a custom distance function that ignores coordinates with NaN values, and compute pairwise distance by using the custom distance function.

Create a matrix with three observations and two variables.

```
rng('default') % For reproducibility  
X = rand(3,2);
```

Assume that the first element of the first observation is missing.

```
X(1,1) = NaN;
```

Compute the Euclidean distance.

```
D1 = pdist(X)
```

```
D1 =
```

```
NaN      NaN      0.9448
```

If observation  $i$  or  $j$  contains NaN values, the function `pdist` returns NaN for the pairwise distance between  $i$  and  $j$ . Therefore,  $D1(1)$  and  $D1(2)$ , the pairwise distances  $(2,1)$  and  $(3,1)$ , are NaN values.

Define a custom distance function `naneucdist` that ignores coordinates with NaN values and returns the Euclidean distance.

```
function D2 = naneucdist(XI,XJ)
%NANEUCDIST Euclidean distance ignoring coordinates with NaNs
n = size(XI,2);
sqdx = (XI-XJ).^2;
nstar = sum(~isnan(sqdx),2); % Number of pairs that do not contain NaNs
nstar(nstar == 0) = NaN; % To return NaN if all pairs include NaNs
D2squared = nansum(sqdx,2).*n./nstar; % Correction for missing coordinates
D2 = sqrt(D2squared);
```

Compute the distance with `naneucdist` by passing the function handle as an input argument of `pdist`.

```
D2 = pdist(X,@naneucdist)
```

```
D2 =
```

```
0.3974      1.1538      0.9448
```

## Input Arguments

### **x** — Input data

numeric matrix

Input data, specified as a numeric matrix of size  $m$ -by- $n$ . Rows correspond to individual observations, and columns correspond to individual variables.

Data Types: `single` | `double`

**Distance — Distance metric**

character vector | function handle

Distance metric, specified as a character vector or function handle, as described in the following table.

Value	Description
'euclidean'	Euclidean distance (default).
'squaredeuclidean'	Squared Euclidean distance. (This option is provided for efficiency only. It does not satisfy the triangle inequality.)
'seuclidean'	Standardized Euclidean distance. Each coordinate difference between observations is scaled by dividing by the corresponding element of the standard deviation, $S = \text{nanstd}(X)$ . Use <code>DistParameter</code> to specify another value for $S$ .
'mahalanobis'	Mahalanobis distance using the sample covariance of $X$ , $C = \text{nancov}(X)$ . Use <code>DistParameter</code> to specify another value for $C$ , where the matrix $C$ is symmetric and positive definite.
'cityblock'	City block distance.
'minkowski'	Minkowski distance. The default exponent is 2. Use <code>DistParameter</code> to specify a different exponent $P$ , where $P$ is a positive scalar value of the exponent.
'chebychev'	Chebychev distance (maximum coordinate difference).
'cosine'	One minus the cosine of the included angle between points (treated as vectors).
'correlation'	One minus the sample correlation between points (treated as sequences of values).
'hamming'	Hamming distance, which is the percentage of coordinates that differ.
'jaccard'	One minus the Jaccard coefficient, which is the percentage of nonzero coordinates that differ.
'spearman'	One minus the sample Spearman's rank correlation between observations (treated as sequences of values).

Value	Description
@ <i>distfun</i>	<p>Custom distance function handle. A distance function has the form</p> <pre>function D2 = DISTFUN(ZI,ZJ) % calculation of distance ...</pre> <p>where</p> <ul style="list-style-type: none"> <li>• ZI is a 1-by-n vector containing a single observation.</li> <li>• ZJ is an m2-by-n matrix containing multiple observations. <i>distfun</i> must accept a matrix XJ with an arbitrary number of observations.</li> <li>• D2 is an m2-by-1 vector of distances, and D2(k) is the distance between observations ZI and ZJ(k, :).</li> </ul> <p>If your data is not sparse, you can generally compute distance more quickly by using a built-in distance instead of a function handle.</p>

For definitions, see “Distance Metrics” on page 32-4665.

When you use 'seuclidean', 'minkowski', or 'mahalanobis', you can specify an additional input argument *DistParameter* to control these metrics. You can also use these metrics in the same way as the other metrics with a default value of *DistParameter*.

Example: 'minkowski'

Data Types: char | function\_handle

#### **DistParameter** — Distance metric parameter values

positive scalar | numeric vector | numeric matrix

Distance metric parameter values, specified as a positive scalar, numeric vector, or numeric matrix. This argument is valid only when you specify *Distance* as 'seuclidean', 'minkowski', or 'mahalanobis'.

- If *Distance* is 'seuclidean', *DistParameter* is a vector of scaling factors for each dimension, specified as a positive vector. The default value is `nanstd(X)`.

- If Distance is 'minkowski', DistParameter is the exponent of Minkowski distance, specified as a positive scalar. The default value is 2.
- If Distance is 'mahalanobis', DistParameter is a covariance matrix, specified as a numeric matrix. The default value is `nancov(X)`. DistParameter must be symmetric and positive definite.

Example: `'minkowski', 3`

Data Types: `single` | `double`

## Output Arguments

### **D** — Pairwise distances

numeric row vector

Pairwise distances, returned as a numeric row vector of length  $m(m-1)/2$ , corresponding to pairs of observations, where  $m$  is the number of observations in  $X$ .

The distances are arranged in the order  $(2,1), (3,1), \dots, (m,1), (3,2), \dots, (m,2), \dots, (m,m-1)$ , i.e., the lower-left triangle of the full  $m$ -by- $m$  distance matrix in column order. The pairwise distance between observations  $i$  and  $j$  is in  $D((i-1)*(m-i/2)+j-i)$  for  $i \leq j$ .

You can convert  $D$  into a symmetric matrix by using the `squareform` function.  $Z = \text{squareform}(D)$  returns an  $m$ -by- $m$  matrix where  $Z(i, j)$  corresponds to the pairwise distance between observations  $i$  and  $j$ .

If observation  $i$  or  $j$  contains NaNs, then the corresponding value in  $D$  is NaN for the built-in distance functions.

$D$  is commonly used as a dissimilarity matrix in clustering or multidimensional scaling. For details, see “Hierarchical Clustering” on page 16-3 and the function reference pages for `cmdscale`, `cophenet`, `linkage`, `mdscale`, and `optimalleaforder`. These functions take  $D$  as an input argument.

## Definitions

### Distance Metrics

A distance metric is a function that defines a distance between two observations. `pdist` supports various distance metrics: Euclidean distance, standardized Euclidean distance, Mahalanobis distance, city block distance, Minkowski distance, Chebychev distance, cosine distance, correlation distance, Hamming distance, Jaccard distance, and Spearman distance.

Given an  $m$ -by- $n$  data matrix  $X$ , which is treated as  $m$  (1-by- $n$ ) row vectors  $x_1, x_2, \dots, x_m$ , the various distances between the vector  $x_s$  and  $x_t$  are defined as follows:

- Euclidean distance

$$d_{st}^2 = (x_s - x_t)(x_s - x_t)'$$

The Euclidean distance is a special case of the Minkowski distance, where  $p = 2$ .

- Standardized Euclidean distance

$$d_{st}^2 = (x_s - x_t)V^{-1}(x_s - x_t)'$$

where  $V$  is the  $n$ -by- $n$  diagonal matrix whose  $j$ th diagonal element is  $(S(j))^2$ , where  $S$  is a vector of scaling factors for each dimension.

- Mahalanobis distance

$$d_{st}^2 = (x_s - x_t)C^{-1}(x_s - x_t)'$$

where  $C$  is the covariance matrix.

- City block distance

$$d_{st} = \sum_{j=1}^n |x_{sj} - x_{tj}|$$

The city block distance is a special case of the Minkowski distance, where  $p = 1$ .

- Minkowski distance

$$d_{st} = \sqrt[p]{\sum_{j=1}^n |x_{sj} - x_{tj}|^p}$$

For the special case of  $p = 1$ , the Minkowski distance gives the city block distance. For the special case of  $p = 2$ , the Minkowski distance gives the Euclidean distance. For the special case of  $p = \infty$ , the Minkowski distance gives the Chebychev distance.

- Chebychev distance

$$d_{st} = \max_j \{|x_{sj} - x_{tj}|\}.$$

The Chebychev distance is a special case of the Minkowski distance, where  $p = \infty$ .

- Cosine distance

$$d_{st} = 1 - \frac{x_s x_t'}{\sqrt{(x_s x_s')(x_t x_t')}}.$$

- Correlation distance

$$d_{st} = 1 - \frac{(x_s - \bar{x}_s)(x_t - \bar{x}_t)'}{\sqrt{(x_s - \bar{x}_s)(x_s - \bar{x}_s)'} \sqrt{(x_t - \bar{x}_t)(x_t - \bar{x}_t)'}}$$

where

$$\bar{x}_s = \frac{1}{n} \sum_j x_{sj} \quad \text{and} \quad \bar{x}_t = \frac{1}{n} \sum_j x_{tj}.$$

- Hamming distance

$$d_{st} = (\#(x_{sj} \neq x_{tj}) / n).$$

- Jaccard distance

$$d_{st} = \frac{\#[(x_{sj} \neq x_{tj}) \cap ((x_{sj} \neq 0) \cup (x_{tj} \neq 0))]}{\#[(x_{sj} \neq 0) \cup (x_{tj} \neq 0)]}.$$

- Spearman distance

$$d_{st} = 1 - \frac{(r_s - \bar{r}_s)(r_t - \bar{r}_t)'}{\sqrt{(r_s - \bar{r}_s)(r_s - \bar{r}_s)'} \sqrt{(r_t - \bar{r}_t)(r_t - \bar{r}_t)'}}$$

where



- $r_{sj}$  is the rank of  $x_{sj}$  taken over  $x_{1j}, x_{2j}, \dots, x_{mj}$ , as computed by `tiedrank`.
- $r_s$  and  $r_t$  are the coordinate-wise rank vectors of  $x_s$  and  $x_t$ , i.e.,  $r_s = (r_{s1}, r_{s2}, \dots, r_{sn})$ .
- 

$$\bar{r}_s = \frac{1}{n} \sum_j r_{sj} = \frac{(n+1)}{2}.$$

$$\bar{r}_t = \frac{1}{n} \sum_j r_{tj} = \frac{(n+1)}{2}.$$

## See Also

`cluster` | `clusterdata` | `cmdscale` | `cophenet` | `dendrogram` | `inconsistent` | `linkage` | `pdist2` | `silhouette` | `squareform`

## Topics

“Introduction to Cluster Analysis” on page 16-2

“Hierarchical Clustering” on page 16-3

Introduced before R2006a

## pdist2

Pairwise distance between two sets of observations

### Syntax

```
D = pdist2(X,Y,Distance)
```

```
D = pdist2(X,Y,Distance,DistParameter)
```

```
D = pdist2( ____,Name,Value)
```

```
[D,I] = pdist2( ____,Name,Value)
```

### Description

`D = pdist2(X,Y,Distance)` returns the distance between each pair of observations in `X` and `Y` using the metric specified by `Distance`.

`D = pdist2(X,Y,Distance,DistParameter)` returns the distance using the metric specified by `Distance` and `DistParameter`. You can specify `DistParameter` only when `Distance` is `'seuclidean'`, `'minkowski'`, or `'mahalanobis'`.

`D = pdist2( ____,Name,Value)` specifies an additional option using one of the name-value pair arguments `'Smallest'` or `'Largest'` in addition to any of the arguments in the previous syntaxes.

For example,

- `D = pdist2(X,Y,Distance,'Smallest',K)` computes the distance using the metric specified by `Distance` and returns the `K` smallest pairwise distances to observations in `X` for each observation in `Y` in ascending order.
- `D = pdist2(X,Y,Distance,DistParameter,'Largest',K)` computes the distance using the metric specified by `Distance` and `DistParameter` and returns the `K` largest pairwise distances in descending order.

`[D,I] = pdist2( ____, Name, Value)` also returns the matrix `I` using any of the arguments in the previous syntaxes. The matrix `I` contains the indices of the observations in `X` corresponding to the distances in `D`.

## Examples

### Compute Euclidean Distance

Create two matrices with three observations and two variables.

```
rng('default') % For reproducibility
X = rand(3,2);
Y = rand(3,2);
```

Compute the Euclidean distance. The default value of the input argument `Distance` is `'euclidean'`. When computing the Euclidean distance without using a name-value pair argument, you do not need to specify `Distance`.

```
D = pdist2(X,Y)
```

```
D =
    0.5387    0.8018    0.1538
    0.7100    0.5951    0.3422
    0.8805    0.4242    1.2050
```

`D(i,j)` corresponds to the pairwise distance between observation `i` in `X` and observation `j` in `Y`.

### Compute Minkowski Distance

Create two matrices with three observations and two variables.

```
rng('default') % For reproducibility
X = rand(3,2);
Y = rand(3,2);
```

Compute the Minkowski distance with the default exponent 2.

```
D1 = pdist2(X,Y,'minkowski')
```

```
D1 =
```

```
    0.5387    0.8018    0.1538  
    0.7100    0.5951    0.3422  
    0.8805    0.4242    1.2050
```

Compute the Minkowski distance with an exponent of 1, which is equal to the city block distance.

```
D2 = pdist2(X,Y,'minkowski',1)
```

```
D3 = pdist2(X,Y,'cityblock')
```

```
D2 =
```

```
    0.5877    1.0236    0.2000  
    0.9598    0.8337    0.3899  
    1.0189    0.4800    1.7036
```

```
D3 =
```

```
    0.5877    1.0236    0.2000  
    0.9598    0.8337    0.3899  
    1.0189    0.4800    1.7036
```

### Find the Two Smallest Pairwise Distances

Create two matrices with three observations and two variables.

```
rng('default') % For reproducibility  
X = rand(3,2);  
Y = rand(3,2);
```

Find the two smallest pairwise Euclidean distances to observations in X for each observation in Y.

```
[D,I] = pdist2(X,Y,'euclidean','Smallest',2)
```

```
D =
    0.5387    0.4242    0.1538
    0.7100    0.5951    0.3422
```

```
I =
     1     3     1
     2     2     2
```

For each observation in *Y*, `pdist2` finds the two smallest distances by computing and comparing the distance values to all the observations in *X*. The function then sorts the distances in each column of *D* in ascending order. *I* contains the indices of the observations in *X* corresponding to the distances in *D*.

### Compute Pairwise Distance with Missing Elements Using a Custom Distance Function

Define a custom distance function that ignores coordinates with NaN values, and compute pairwise distance by using the custom distance function.

Create two matrices with three observations and three variables.

```
rng('default') % For reproducibility
X = rand(3,3)
Y = [X(:,1:2) rand(3,1)]
```

```
X =
    0.8147    0.9134    0.2785
    0.9058    0.6324    0.5469
    0.1270    0.0975    0.9575
```

```
Y =
    0.8147    0.9134    0.9649
    0.9058    0.6324    0.1576
    0.1270    0.0975    0.9706
```

The first two columns of  $X$  and  $Y$  are identical. Assume that  $X(1,1)$  is missing.

```
X(1,1) = NaN
```

```
X =
```

```
      NaN    0.9134    0.2785
    0.9058    0.6324    0.5469
    0.1270    0.0975    0.9575
```

Compute the Hamming distance.

```
D1 = pdist2(X,Y, 'hamming')
```

```
D1 =
```

```
      NaN      NaN      NaN
    1.0000    0.3333    1.0000
    1.0000    1.0000    0.3333
```

If observation  $i$  in  $X$  or observation  $j$  in  $Y$  contains NaN values, the function `pdist2` returns NaN for the pairwise distance between  $i$  and  $j$ . Therefore,  $D1(1,1)$ ,  $D1(1,2)$ , and  $D1(1,3)$  are NaN values.

Define a custom distance function `nanhamdist` that ignores coordinates with NaN values and computes the Hamming distance. When working with a large number of observations, you can compute the distance more quickly by looping over coordinates of the data.

```
function D2 = nanhamdist(XI,XJ)
%NANHAMDIST Hamming distance ignoring coordinates with NaNs
[m,p] = size(XJ);
nesum = zeros(m,1);
pstar = zeros(m,1);
for q = 1:p
    notnan = ~(isnan(XI(q)) | isnan(XJ(:,q)));
    nesum = nesum + ((XI(q) ~= XJ(:,q)) & notnan);
    pstar = pstar + notnan;
end
```

```
D2 = nesum./pstar;
```

Compute the distance with `nanhamdist` by passing the function handle as an input argument of `pdist2`.

```
D2 = pdist2(X,Y,@nanhamdist)
```

```
D2 =
```

```
    0.5000    1.0000    1.0000
    1.0000    0.3333    1.0000
    1.0000    1.0000    0.3333
```

## Input Arguments

### **x, y** — Input data

numeric matrix

Input data, specified as a numeric matrix.  $X$  is an  $m \times n$  matrix and  $Y$  is an  $m \times n$  matrix. Rows correspond to individual observations, and columns correspond to individual variables.

Data Types: `single` | `double`

### **Distance** — Distance metric

character vector | function handle

Distance metric, specified as a character vector or function handle, as described in the following table.

Value	Description
'euclidean'	Euclidean distance (default).
'squaredeuclidean'	Squared Euclidean distance. (This option is provided for efficiency only. It does not satisfy the triangle inequality.)

Value	Description
'seuclidean'	Standardized Euclidean distance. Each coordinate difference between observations is scaled by dividing by the corresponding element of the standard deviation, $S = \text{nanstd}(X)$ . Use <code>DistParameter</code> to specify another value for $S$ .
'mahalanobis'	Mahalanobis distance using the sample covariance of $X$ , $C = \text{nancov}(X)$ . Use <code>DistParameter</code> to specify another value for $C$ , where the matrix $C$ is symmetric and positive definite.
'cityblock'	City block distance.
'minkowski'	Minkowski distance. The default exponent is 2. Use <code>DistParameter</code> to specify a different exponent $P$ , where $P$ is a positive scalar value of the exponent.
'chebychev'	Chebychev distance (maximum coordinate difference).
'cosine'	One minus the cosine of the included angle between points (treated as vectors).
'correlation'	One minus the sample correlation between points (treated as sequences of values).
'hamming'	Hamming distance, which is the percentage of coordinates that differ.
'jaccard'	One minus the Jaccard coefficient, which is the percentage of nonzero coordinates that differ.
'spearman'	One minus the sample Spearman's rank correlation between observations (treated as sequences of values).



Value	Description
@ <i>distfun</i>	<p>Custom distance function handle. A distance function has the form</p> <pre>function D2 = DISTFUN(ZI,ZJ) % calculation of distance ...</pre> <p>where</p> <ul style="list-style-type: none"> <li>• ZI is a 1-by-n vector containing a single observation.</li> <li>• ZJ is an m2-by-n matrix containing multiple observations. <i>distfun</i> must accept a matrix XJ with an arbitrary number of observations.</li> <li>• D2 is an m2-by-1 vector of distances, and D2(k) is the distance between observations ZI and ZJ(k, :).</li> </ul> <p>If your data is not sparse, you can generally compute distance more quickly by using a built-in distance instead of a function handle.</p>

For definitions, see “Distance Metrics” on page 32-4677.

When you use 'seuclidean', 'minkowski', or 'mahalanobis', you can specify an additional input argument *DistParameter* to control these metrics. You can also use these metrics in the same way as the other metrics with a default value of *DistParameter*.

Example: 'minkowski'

Data Types: char | function\_handle

#### **DistParameter** — Distance metric parameter values

positive scalar | numeric vector | numeric matrix

Distance metric parameter values, specified as a positive scalar, numeric vector, or numeric matrix. This argument is valid only when you specify *Distance* as 'seuclidean', 'minkowski', or 'mahalanobis'.

- If *Distance* is 'seuclidean', *DistParameter* is a vector of scaling factors for each dimension, specified as a positive vector. The default value is `nanstd(X)`.

- If Distance is 'minkowski', DistParameter is the exponent of Minkowski distance, specified as a positive scalar. The default value is 2.
- If Distance is 'mahalanobis', DistParameter is a covariance matrix, specified as a numeric matrix. The default value is `nancov(X)`. DistParameter must be symmetric and positive definite.

Example: `'minkowski',3`

Data Types: `single | double`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of Name, Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as Name1, Value1, ..., NameN, ValueN.

Example: Either `'Smallest',K` or `'Largest',K`. You cannot use both `'Smallest'` and `'Largest'`.

### **Smallest** — Number of smallest distances to find

positive integer

Number of smallest distances to find, specified as the comma-separated pair consisting of 'Smallest' and a positive integer. If you specify 'Smallest', then `pdist2` sorts the distances in each column of `D` in ascending order.

Example: `'Smallest',3`

Data Types: `single | double`

### **Largest** — Number of largest distances to find

positive integer

Number of largest distances to find, specified as the comma-separated pair consisting of 'Largest' and a positive integer. If you specify 'Largest', then `pdist2` sorts the distances in each column of `D` in descending order.

Example: `'Largest',3`

Data Types: `single | double`

## Output Arguments

### **D** — Pairwise distances

numeric matrix

Pairwise distances, returned as a numeric matrix.

If you do not specify either 'Smallest' or 'Largest', then **D** is an  $m_x$ -by- $m_y$  matrix, where  $m_x$  and  $m_y$  are the number of observations in **X** and **Y**, respectively.  $D(i, j)$  is the distance between observation  $i$  in **X** and observation  $j$  in **Y**. If observation  $i$  in **X** or observation  $j$  in **Y** contains NaN, then  $D(i, j)$  is NaN for the built-in distance functions.

If you specify either 'Smallest' or 'Largest' as **K**, then **D** is a  $K$ -by- $m_y$  matrix. **D** contains either the  $K$  smallest or  $K$  largest pairwise distances to observations in **X** for each observation in **Y**. For each observation in **Y**, `pdist2` finds the  $K$  smallest or largest distances by computing and comparing the distance values to all the observations in **X**. If **K** is greater than  $m_x$ , `pdist2` returns an  $m_x$ -by- $m_y$  matrix.

### **I** — Sort index

positive integer matrix

Sort index, returned as a positive integer matrix. **I** is the same size as **D**. **I** contains the indices of the observations in **X** corresponding to the distances in **D**.

## Definitions

### Distance Metrics

A distance metric is a function that defines a distance between two observations. `pdist2` supports various distance metrics: Euclidean distance, standardized Euclidean distance, Mahalanobis distance, city block distance, Minkowski distance, Chebychev distance, cosine distance, correlation distance, Hamming distance, Jaccard distance, and Spearman distance.

Given an  $m_x$ -by- $n$  data matrix **X**, which is treated as  $m_x$  (1-by- $n$ ) row vectors  $x_1, x_2, \dots, x_{m_x}$ , and an  $m_y$ -by- $n$  data matrix **Y**, which is treated as  $m_y$  (1-by- $n$ ) row vectors  $y_1, y_2, \dots, y_{m_y}$ , the various distances between the vector  $x_s$  and  $y_t$  are defined as follows:

- Euclidean distance

$$d_{st}^2 = (x_s - y_t)(x_s - y_t)'$$

The Euclidean distance is a special case of the Minkowski distance, where  $p = 2$ .

- Standardized Euclidean distance

$$d_{st}^2 = (x_s - y_t)V^{-1}(x_s - y_t)'$$

where  $V$  is the  $n$ -by- $n$  diagonal matrix whose  $j$ th diagonal element is  $(S(j))^2$ , where  $S$  is a vector of scaling factors for each dimension.

- Mahalanobis distance

$$d_{st}^2 = (x_s - y_t)C^{-1}(x_s - y_t)'$$

where  $C$  is the covariance matrix.

- City block distance

$$d_{st} = \sum_{j=1}^n |x_{sj} - y_{tj}|$$

The city block distance is a special case of the Minkowski distance, where  $p = 1$ .

- Minkowski distance

$$d_{st} = \sqrt[p]{\sum_{j=1}^n |x_{sj} - y_{tj}|^p}$$

For the special case of  $p = 1$ , the Minkowski distance gives the city block distance. For the special case of  $p = 2$ , the Minkowski distance gives the Euclidean distance. For the special case of  $p = \infty$ , the Minkowski distance gives the Chebychev distance.

- Chebychev distance

$$d_{st} = \max_j \{|x_{sj} - y_{tj}|\}$$

The Chebychev distance is a special case of the Minkowski distance, where  $p = \infty$ .

- Cosine distance

$$d_{st} = \left( 1 - \frac{x_s y_t'}{\sqrt{(x_s x_s')(y_t y_t')}} \right)$$

- Correlation distance

$$d_{st} = 1 - \frac{(x_s - \bar{x}_s)(y_t - \bar{y}_t)'}{\sqrt{(x_s - \bar{x}_s)(x_s - \bar{x}_s)'(y_t - \bar{y}_t)(y_t - \bar{y}_t)'}}$$

where

$$\bar{x}_s = \frac{1}{n} \sum_j x_{sj}$$

and

$$\bar{y}_t = \frac{1}{n} \sum_j y_{tj}.$$

- Hamming distance

$$d_{st} = (\#(x_{sj} \neq y_{tj}) / n).$$

- Jaccard distance

$$d_{st} = \frac{\#[(x_{sj} \neq y_{tj}) \cap ((x_{sj} \neq 0) \cup (y_{tj} \neq 0))]}{\#[(x_{sj} \neq 0) \cup (y_{tj} \neq 0)]}.$$

- Spearman distance

$$d_{st} = 1 - \frac{(r_s - \bar{r}_s)(r_t - \bar{r}_t)'}{\sqrt{(r_s - \bar{r}_s)(r_s - \bar{r}_s)'(r_t - \bar{r}_t)(r_t - \bar{r}_t)'}}$$

where

- $r_{sj}$  is the rank of  $x_{sj}$  taken over  $x_{1j}, x_{2j}, \dots, x_{mj,j}$ , as computed by `tiedrank`.
- $r_{tj}$  is the rank of  $y_{tj}$  taken over  $y_{1j}, y_{2j}, \dots, y_{mj,j}$ , as computed by `tiedrank`.
- $r_s$  and  $r_t$  are the coordinate-wise rank vectors of  $x_s$  and  $y_t$ , i.e.,  $r_s = (r_{s1}, r_{s2}, \dots, r_{sn})$  and  $r_t = (r_{t1}, r_{t2}, \dots, r_{tn})$ .
- 

$$\bar{r}_s = \frac{1}{n} \sum_j r_{sj} = \frac{(n+1)}{2}.$$

$$\bar{r}_t = \frac{1}{n} \sum_j r_{tj} = \frac{(n+1)}{2}.$$

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- The distance input argument value (`Distance`) must be a compile-time constant. For example, to use the Minkowski distance, include `coder.Constant('Minkowski')` in the `-args` value of `codegen`.
- The distance input argument value (`Distance`) cannot be a custom distance function.
- Names in name-value pair arguments must be compile-time constants. For example, to use the 'Smallest' name-value pair argument in the generated code, include `{coder.Constant('Smallest'), 0}` in the `-args` value of `codegen`.
- The sorted order of tied distances in the generated code can be different from the order in MATLAB due to numerical precision.
- For code generation, `pdist2` uses `parfor` (by default) to create loops that run in parallel on supported shared-memory multicore platforms. If your compiler does not support the Open Multiprocessing (OpenMP) application interface or you disable OpenMP library, MATLAB Coder treats the `parfor`-loops as `for`-loops. To find supported compilers, see [http://www.mathworks.com/support/compilers/current\\_release/](http://www.mathworks.com/support/compilers/current_release/). To disable OpenMP library, specify the `EnableOpenMP` property of the `codegen` configuration object as `false`. For details, see `coder.CodeConfig`.

### See Also

`ExhaustiveSearcher` | `KDTreeSearcher` | `createns` | `knnsearch` | `pdist`

Introduced in R2010a

## pearsrnd

Pearson system random numbers

### Syntax

```
r = pearsrnd(mu, sigma, skew, kurt, m, n)
r = pearsrnd(mu, sigma, skew, kurt)
r = pearsrnd(mu, sigma, skew, kurt, m, n, ...)
r = pearsrnd(mu, sigma, skew, kurt, [m, n, ...])
[r, type] = pearsrnd(...)
[r, type, coefs] = pearsrnd(...)
```

### Description

`r = pearsrnd(mu, sigma, skew, kurt, m, n)` returns an  $m$ -by- $n$  matrix of random numbers drawn from the distribution in the Pearson system with mean `mu`, standard deviation `sigma`, skewness `skew`, and kurtosis `kurt`. The parameters `mu`, `sigma`, `skew`, and `kurt` must be scalars.

---

**Note** Because `r` is a random sample, its sample moments, especially the skewness and kurtosis, typically differ somewhat from the specified distribution moments.

`pearsrnd` uses the definition of kurtosis for which a normal distribution has a kurtosis of 3. Some definitions of kurtosis subtract 3, so that a normal distribution has a kurtosis of 0. The `pearsrnd` function does not use this convention.

---

Some combinations of moments are not valid; in particular, the kurtosis must be greater than the square of the skewness plus 1. The kurtosis of the normal distribution is defined to be 3.

`r = pearsrnd(mu, sigma, skew, kurt)` returns a scalar value.

`r = pearsrnd(mu, sigma, skew, kurt, m, n, ...)` or `r = pearsrnd(mu, sigma, skew, kurt, [m, n, ...])` returns an  $m$ -by- $n$ -by-... array.

`[r,type] = pearsrnd(...)` returns the type of the specified distribution within the Pearson system. `type` is a scalar integer from 0 to 7. Set `m` and `n` to 0 to identify the distribution type without generating any random values.

The seven distribution types in the Pearson system correspond to the following distributions:

- 0 — Normal on page B-134 distribution
- 1 — Four-parameter beta on page B-4 distribution
- 2 — Symmetric four-parameter beta on page B-4 distribution
- 3 — Three-parameter gamma on page B-47 distribution
- 4 — Not related to any standard distribution. The density is proportional to:

$$(1 + ((x - a)/b)^2)^{-c} \exp(-d \arctan((x - a)/b)).$$

- 5 — Inverse gamma on page B-47 location-scale distribution
- 6 —  $F$  on page B-44 location-scale distribution
- 7 — Student's  $t$  location-scale on page B-175 distribution

`[r,type,coefs] = pearsrnd(...)` returns the coefficients `coefs` of the quadratic polynomial that defines the distribution via the differential equation

$$\frac{d}{dx} \log(p(x)) = \frac{-(a+x)}{c(0) + c(1)x + c(2)x^2}.$$

## Examples

Generate random values from the standard normal distribution:

```
r = pearsrnd(0,1,0,3,100,1); % Equivalent to randn(100,1)
```

Determine the distribution type:

```
[r,type] = pearsrnd(0,1,1,4,0,0);  
r =  
    []  
type =  
    1
```



## References

- [1] Johnson, N.L., S. Kotz, and N. Balakrishnan (1994) Continuous Univariate Distributions, Volume 1, Wiley-Interscience, Pg 15, Eqn 12.33.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

The generated code matches MATLAB only when generated output `r` is scalar.

### See Also

`johnsrnd` | `random`

Introduced in R2006a

## perfcurve

Receiver operating characteristic (ROC) curve or other performance curve for classifier output

### Syntax

```
[X,Y] = perfcurve(labels,scores,posclass)
[X,Y,T] = perfcurve(labels,scores,posclass)
[X,Y,T,AUC] = perfcurve(labels,scores,posclass)
[X,Y,T,AUC,OPTROCPT] = perfcurve(labels,scores,posclass)
[X,Y,T,AUC,OPTROCPT,SUBY] = perfcurve(labels,scores,posclass)
[X,Y,T,AUC,OPTROCPT,SUBY,SUBYNAMES] = perfcurve(labels,scores,
posclass)
[ ___ ] = perfcurve(labels,scores,posclass,Name,Value)
```

### Description

`[X,Y] = perfcurve(labels,scores,posclass)` returns the X and Y coordinates of an ROC curve for a vector of classifier predictions, `scores`, given true class labels, `labels`, and the positive class label, `posclass`. You can visualize the performance curve using `plot(X,Y)`.

`[X,Y,T] = perfcurve(labels,scores,posclass)` returns an array of thresholds on classifier scores for the computed values of X and Y.

`[X,Y,T,AUC] = perfcurve(labels,scores,posclass)` returns the area under the curve for the computed values of X and Y.

`[X,Y,T,AUC,OPTROCPT] = perfcurve(labels,scores,posclass)` returns the optimal operating point of the ROC curve.

`[X,Y,T,AUC,OPTROCPT,SUBY] = perfcurve(labels,scores,posclass)` returns the Y values for negative subclasses.

`[X,Y,T,AUC,OPTROCPT,SUBY,SUBYNAMES] = perfcurve(labels,scores, posclass)` returns the negative class names.

[ \_\_\_ ] = perfcurve(labels, scores, posclass, Name, Value) returns the coordinates of a ROC curve and any other output argument from the previous syntaxes, with additional options specified by one or more Name, Value pair arguments.

For example, you can provide a list of negative classes, change the X or Y criterion, compute pointwise confidence bounds on page 32-4715 using cross validation or bootstrap, specify the misclassification cost, or compute the confidence bounds in parallel.

## Examples

### Plot ROC Curve for Classification by Logistic Regression

Load the sample data.

```
load fisheriris
```

Use only the first two features as predictor variables. Define a binary classification problem by using only the measurements that correspond to the species versicolor and virginica.

```
pred = meas(51:end, 1:2);
```

Define the binary response variable.

```
resp = (1:100) '>50'; % Versicolor = 0, virginica = 1
```

Fit a logistic regression model.

```
mdl = fitglm(pred, resp, 'Distribution', 'binomial', 'Link', 'logit');
```

Compute the ROC curve. Use the probability estimates from the logistic regression model as scores.

```
scores = mdl.Fitted.Probability;
[X, Y, T, AUC] = perfcurve(species(51:end, :), scores, 'virginica');
```

perfcurve stores the threshold values in the array T.

Display the area under the curve.

```
AUC
```

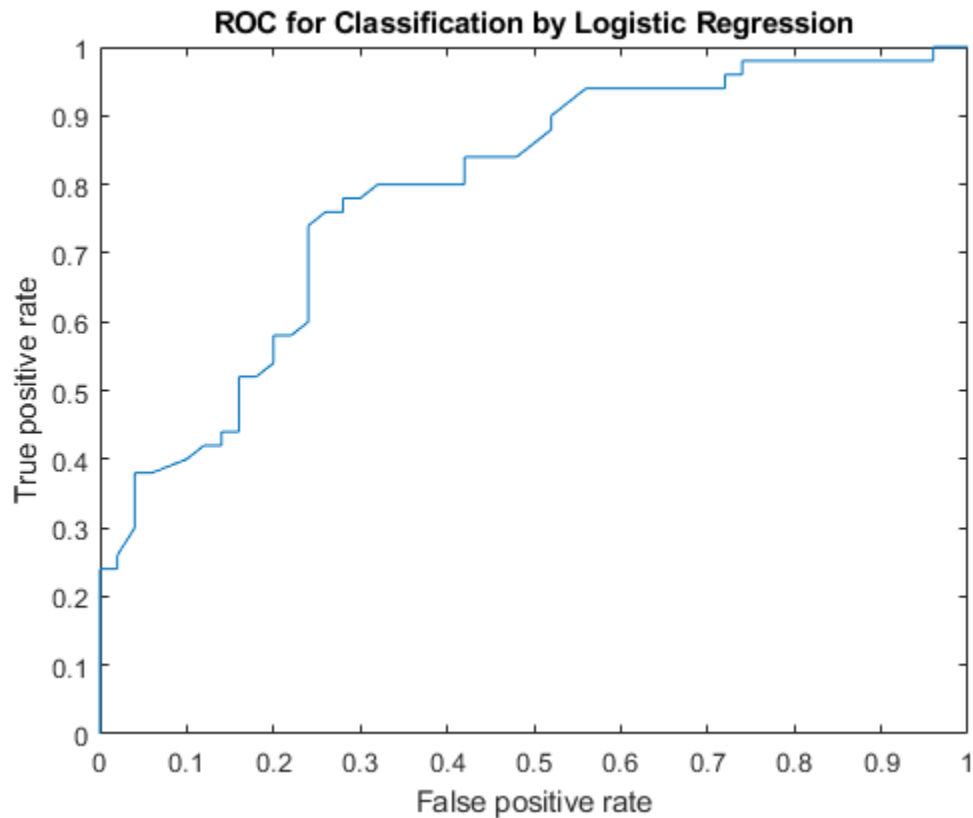
AUC =

0.7918

The area under the curve is 0.7918. The maximum AUC is 1, which corresponds to a perfect classifier. Larger AUC values indicate better classifier performance.

Plot the ROC curve.

```
plot(X, Y)
xlabel('False positive rate')
ylabel('True positive rate')
title('ROC for Classification by Logistic Regression')
```



### Compare Classification Methods Using ROC Curve

Load the sample data.

```
load ionosphere
```

$X$  is a 351x34 real-valued matrix of predictors.  $Y$  is a character array of class labels: 'b' for bad radar returns and 'g' for good radar returns.

Reformat the response to fit a logistic regression. Use the predictor variables 3 through 34.

```
resp = strcmp(Y, 'b'); % resp = 1, if Y = 'b', or 0 if Y = 'g'  
pred = X(:,3:34);
```

Fit a logistic regression model to estimate the posterior probabilities for a radar return to be a bad one.

```
mdl = fitglm(pred, resp, 'Distribution', 'binomial', 'Link', 'logit');  
score_log = mdl.Fitted.Probability; % Probability estimates
```

Compute the standard ROC curve using the probabilities for scores.

```
[Xlog, Ylog, Tlog, AUClog] = perfcurve(resp, score_log, 'true');
```

Train an SVM classifier on the same sample data. Standardize the data.

```
mdlSVM = fitcsvm(pred, resp, 'Standardize', true);
```

Compute the posterior probabilities (scores).

```
mdlSVM = fitPosterior(mdlSVM);  
[~, score_svm] = resubPredict(mdlSVM);
```

The second column of `score_svm` contains the posterior probabilities of bad radar returns.

Compute the standard ROC curve using the scores from the SVM model.

```
[Xsvm, Ysvm, Tsvm, AUCsvm] = perfcurve(resp, score_svm(:, mdlSVM.ClassNames), 'true');
```

Fit a naive Bayes classifier on the same sample data.

```
mdlNB = fitcnb(pred, resp);
```

Compute the posterior probabilities (scores).

```
[~, score_nb] = resubPredict(mdlNB);
```

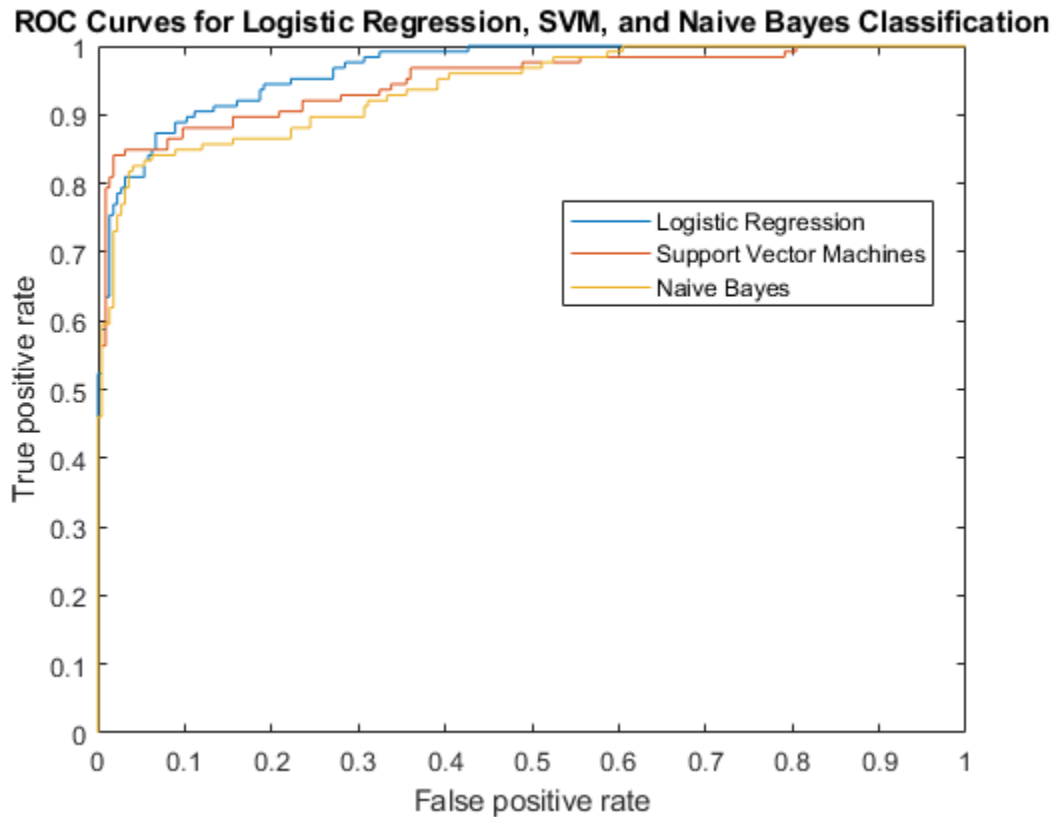
Compute the standard ROC curve using the scores from the naive Bayes classification.

```
[Xnb, Ynb, Tnb, AUCnb] = perfcurve(resp, score_nb(:, mdlNB.ClassNames), 'true');
```

Plot the ROC curves on the same graph.

```
plot(Xlog, Ylog)  
hold on  
plot(Xsvm, Ysvm)
```

```
plot(Xnb, Ynb)
legend('Logistic Regression', 'Support Vector Machines', 'Naive Bayes', 'Location', 'Best')
xlabel('False positive rate'); ylabel('True positive rate');
title('ROC Curves for Logistic Regression, SVM, and Naive Bayes Classification')
hold off
```



Although SVM produces better ROC values for higher thresholds, logistic regression is usually better at distinguishing the bad radar returns from the good ones. The ROC curve for naive Bayes is generally lower than the other two ROC curves, which indicates worse in-sample performance than the other two classifier methods.

Compare the area under the curve for all three classifiers.

```
AUClog
AUCsvm
AUCnb

AUClog =

    0.9659

AUCsvm =

    0.9488

AUCnb =

    0.9393
```

Logistic regression has the highest AUC measure for classification and naive Bayes has the lowest. This result suggests that logistic regression has better in-sample average performance for this sample data.

### Determine the Parameter Value for Custom Kernel Function

This example shows how to determine the better parameter value for a custom kernel function in a classifier using the ROC curves.

Generate a random set of points within the unit circle.

```
rng(1); % For reproducibility
n = 100; % Number of points per quadrant

r1 = sqrt(rand(2*n,1)); % Random radii
t1 = [pi/2*rand(n,1); (pi/2*rand(n,1)+pi)]; % Random angles for Q1 and Q3
X1 = [r1.*cos(t1) r1.*sin(t1)]; % Polar-to-Cartesian conversion

r2 = sqrt(rand(2*n,1));
t2 = [pi/2*rand(n,1)+pi/2; (pi/2*rand(n,1)-pi/2)]; % Random angles for Q2 and Q4
X2 = [r2.*cos(t2) r2.*sin(t2)];
```



Define the predictor variables. Label points in the first and third quadrants as belonging to the positive class, and those in the second and fourth quadrants in the negative class.

```
pred = [X1; X2];
resp = ones(4*n,1);
resp(2*n + 1:end) = -1; % Labels
```

Create the function `mysigmoid.m`, which accepts two matrices in the feature space as inputs, and transforms them into a Gram matrix using the sigmoid kernel.

```
function G = mysigmoid(U,V)
% Sigmoid kernel function with slope gamma and intercept c
gamma = 1;
c = -1;
G = tanh(gamma*U*V' + c);
end
```

Train an SVM classifier using the sigmoid kernel function. It is good practice to standardize the data.

```
SVMMoDel1 = fitcsvm(pred,resp,'KernelFunction','mysigmoid',...
    'Standardize',true);
SVMMoDel1 = fitPosterior(SVMMoDel1);
[~,scores1] = resubPredict(SVMMoDel1);
```

Set `gamma = 0.5`; within `mysigmoid.m` and save as `mysigmoid2.m`. And, train an SVM classifier using the adjusted sigmoid kernel.

```
function G = mysigmoid2(U,V)
% Sigmoid kernel function with slope gamma and intercept c
gamma = 0.5;
c = -1;
G = tanh(gamma*U*V' + c);
end
```

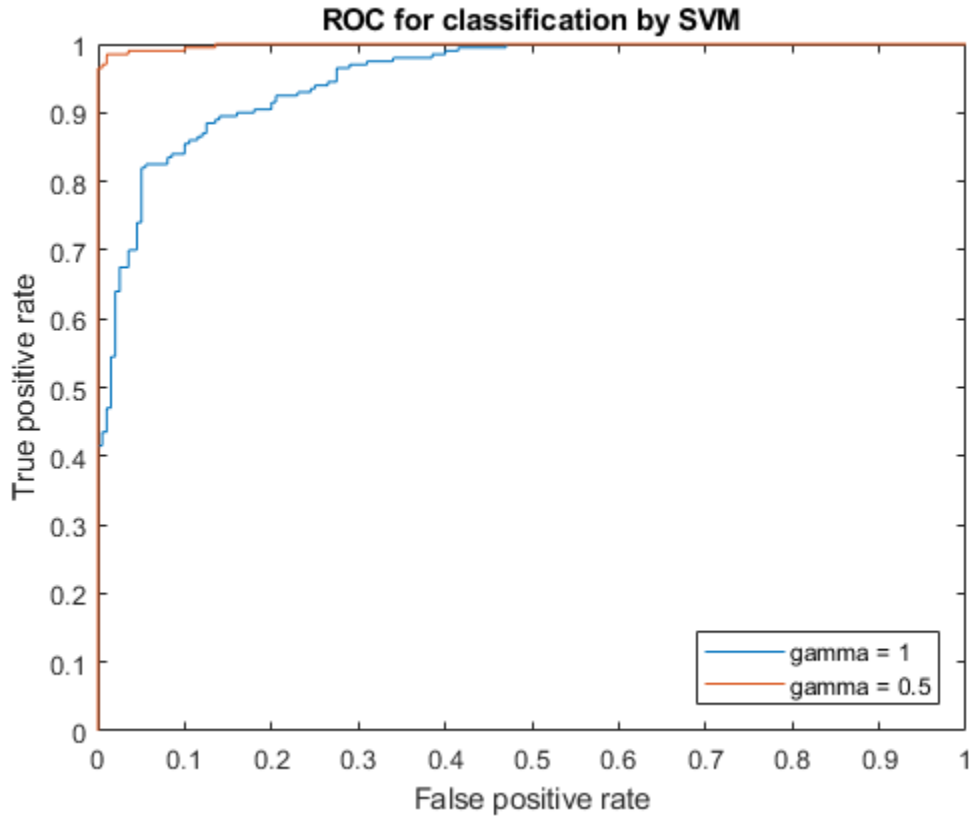
```
SVMMoDel2 = fitcsvm(pred,resp,'KernelFunction','mysigmoid2',...
    'Standardize',true);
SVMMoDel2 = fitPosterior(SVMMoDel2);
[~,scores2] = resubPredict(SVMMoDel2);
```

Compute the ROC curves and the area under the curve (AUC) for both models.

```
[x1,y1,~,auc1] = perfcurve(resp,scores1(:,2),1);  
[x2,y2,~,auc2] = perfcurve(resp,scores2(:,2),1);
```

Plot the ROC curves.

```
plot(x1,y1)  
hold on  
plot(x2,y2)  
hold off  
legend('gamma = 1','gamma = 0.5','Location','SE');  
xlabel('False positive rate'); ylabel('True positive rate');  
title('ROC for classification by SVM');
```



The kernel function with the gamma parameter set to 0.5 gives better in-sample results.

Compare the AUC measures.

```
auc1
auc2

auc1 =
    0.9518

auc2 =
    0.9985
```

The area under the curve for gamma set to 0.5 is higher than that for gamma set to 1. This also confirms that gamma parameter value of 0.5 produces better results. For visual comparison of the classification performance with these two gamma parameter values, see “Train SVM Classifier Using Custom Kernel” on page 18-199.

## Plot ROC Curve for Classification Tree

Load the sample data.

```
load fisheriris
```

The column vector, `species`, consists of iris flowers of three different species: `setosa`, `versicolor`, `virginica`. The double matrix `meas` consists of four types of measurements on the flowers: sepal length, sepal width, petal length, and petal width. All measures are in centimeters.

Train a classification tree using the sepal length and width as the predictor variables. It is a good practice to specify the class names.

```
Model = fitctree(meas(:,1:2),species, ...
    'ClassNames',{'setosa','versicolor','virginica'});
```

Predict the class labels and scores for the species based on the tree `Model`.

```
[~,score] = resubPredict(Model);
```

The scores are the posterior probabilities that an observation (a row in the data matrix) belongs to a class. The columns of `score` correspond to the classes specified by `'ClassNames'`. So, the first column corresponds to `setosa`, the second corresponds to `versicolor`, and the third column corresponds to `virginica`.

Compute the ROC curve for the predictions that an observation belongs to `versicolor`, given the true class labels `species`. Also compute the optimal operating point and `y` values for negative subclasses. Return the names of the negative classes.

Because this is a multiclass problem, you cannot merely supply `score(:,2)` as input to `perfcurve`. Doing so would not give `perfcurve` enough information about the scores for the two negative classes (`setosa` and `virginica`). This problem is unlike a binary classification problem, where knowing the scores of one class is enough to determine the scores of the other class. Therefore, you must supply `perfcurve` with a function that factors in the scores of the two negative classes. One such function is `score(:,2) - max(score(:,1), score(:,3))`.

```
diffscore = score(:,2) - max(score(:,1), score(:,3));  
[X,Y,T,~,OPTROCPT,suby,subnames] = perfcurve(species,diffscore,'versicolor');
```

`X`, by default, is the false positive rate (fallout or 1-specificity) and `Y`, by default, is the true positive rate (recall or sensitivity). The positive class label is `versicolor`. Because a negative class is not defined, `perfcurve` assumes that the observations that do not belong to the positive class are in one class. The function accepts it as the negative class.

```
OPTROCPT  
suby  
subnames
```

```
OPTROCPT =
```

```
0.1000    0.8000
```

```
suby =
```

```
0         0  
0.1800    0.1800  
0.4800    0.4800  
0.5800    0.5800  
0.6200    0.6200
```

```
0.8000    0.8000
0.8800    0.8800
0.9200    0.9200
0.9600    0.9600
0.9800    0.9800
1.0000    1.0000
1.0000    1.0000
```

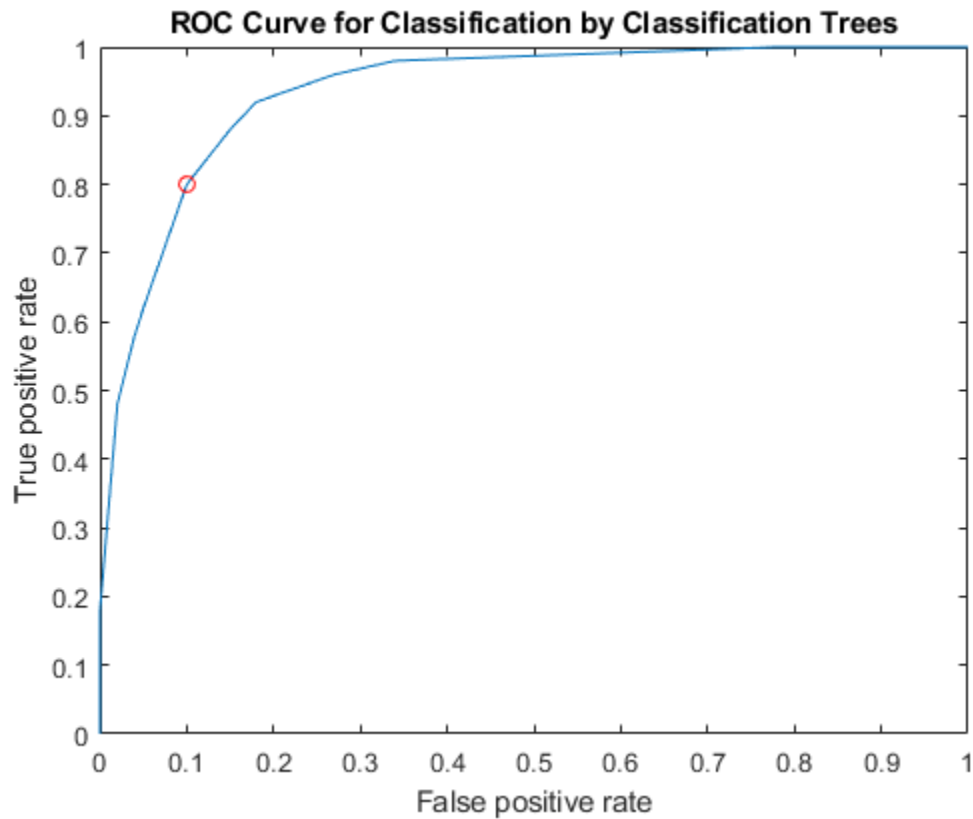
```
subnames =
```

```
1x2 cell array
```

```
    {'setosa'}    {'virginica'}
```

Plot the ROC curve and the optimal operating point on the ROC curve.

```
plot(X,Y)
hold on
plot(OPTROCPT(1),OPTROCPT(2),'ro')
xlabel('False positive rate')
ylabel('True positive rate')
title('ROC Curve for Classification by Classification Trees')
hold off
```



Find the threshold that corresponds to the optimal operating point.

```
T ( (X==OPTROCPT (1)) & (Y==OPTROCPT (2)) )
```

```
ans =
```

```
0.2857
```

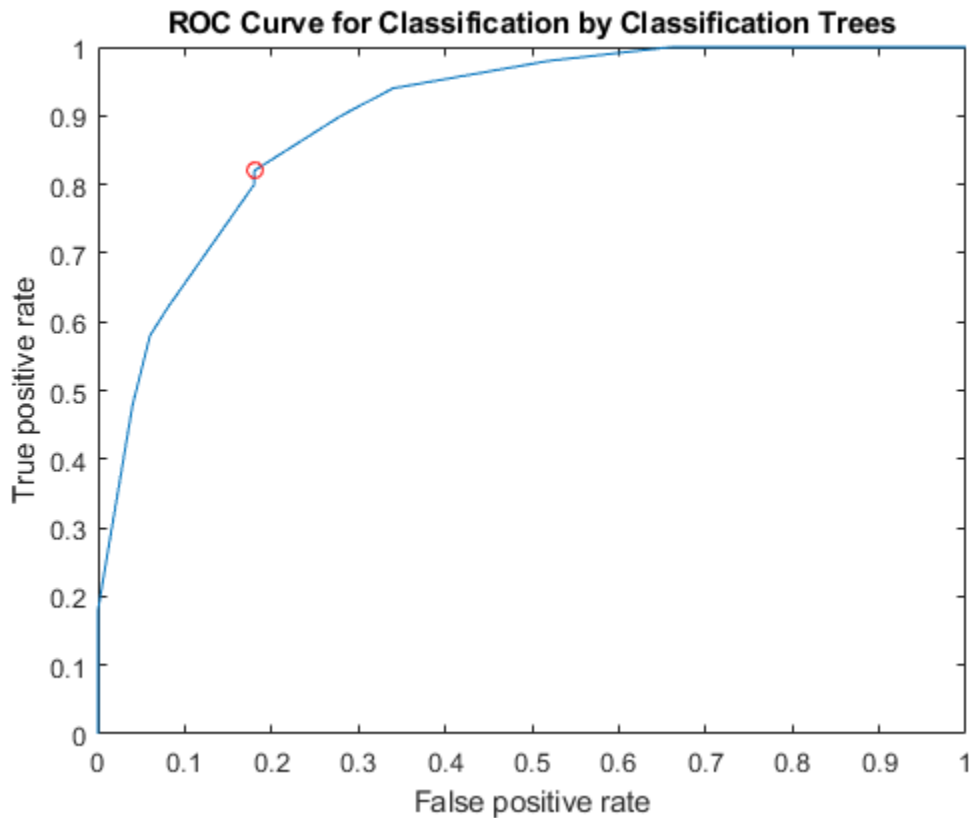
Specify *virginica* as the negative class and compute and plot the ROC curve for *versicolor*.

Again, you must supply `perfcurve` with a function that factors in the scores of the negative class. An example of a function to use is `score(:,2) - score(:,3)`.

```
diffscore = score(:,2) - score(:,3);
[X,Y,~,~,OPTROCPT] = perfcurve(species,diffscore,'versicolor', ...
    'negClass','virginica');
OPTROCPT
figure, plot(X,Y)
hold on
plot(OPTROCPT(1),OPTROCPT(2),'ro')
xlabel('False positive rate')
ylabel('True positive rate')
title('ROC Curve for Classification by Classification Trees')
hold off

OPTROCPT =

    0.1800    0.8200
```



### Compute Pointwise Confidence Intervals for ROC Curve

Load the sample data.

```
load fisheriris
```

The column vector `species` consists of iris flowers of three different species: *setosa*, *versicolor*, *virginica*. The double matrix `meas` consists of four types of measurements on the flowers: sepal length, sepal width, petal length, and petal width. All measures are in centimeters.



Use only the first two features as predictor variables. Define a binary problem by using only the measurements that correspond to the versicolor and virginica species.

```
pred = meas(51:end,1:2);
```

Define the binary response variable.

```
resp = (1:100) '>50'; % Versicolor = 0, virginica = 1
```

Fit a logistic regression model.

```
mdl = fitglm(pred,resp,'Distribution','binomial','Link','logit');
```

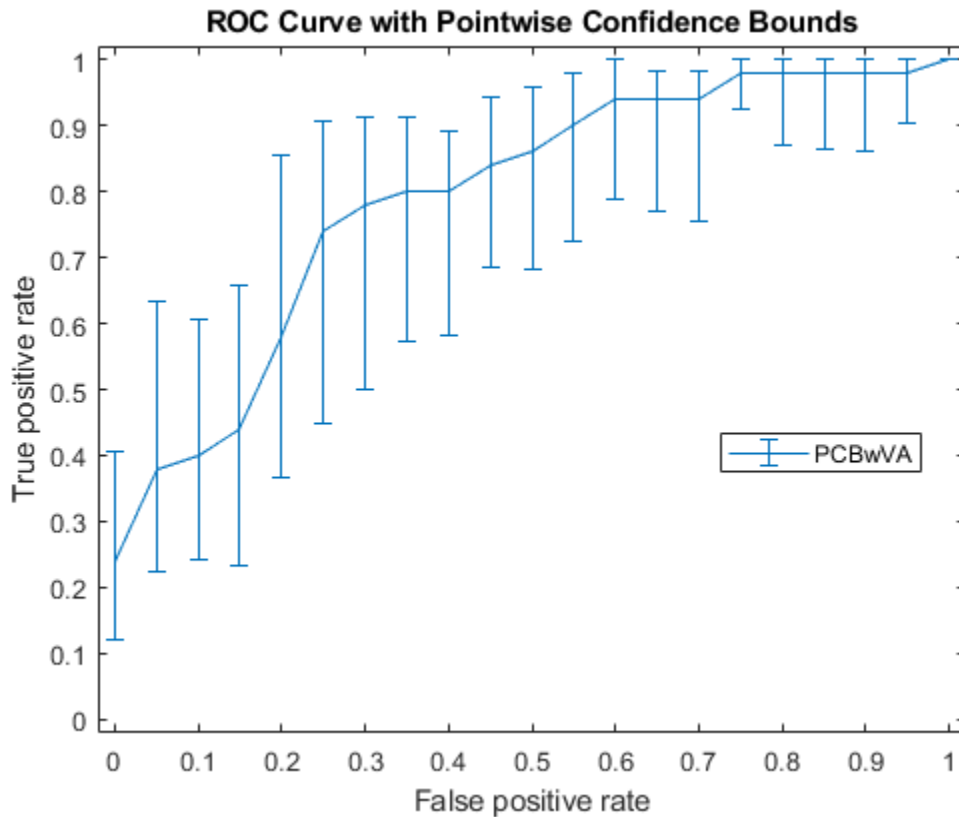
Compute the pointwise confidence intervals on the true positive rate (TPR) by vertical averaging (VA) and sampling using bootstrap.

```
[X,Y,T] = perfcurve(species(51:end,:),mdl.Fitted.Probability,...
    'virginica','NBoot',1000,'XVals',[0:0.05:1]);
```

'NBoot',1000 sets the number of bootstrap replicas to 1000. 'XVals','All' prompts perfcurve to return X, Y, and T values for all scores, and average the Y values (true positive rate) at all X values (false positive rate) using vertical averaging. If you do not specify XVals, then perfcurve computes the confidence bounds using threshold averaging by default.

Plot the pointwise confidence intervals.

```
errorbar(X,Y(:,1),Y(:,1)-Y(:,2),Y(:,3)-Y(:,1));
xlim([-0.02,1.02]); ylim([-0.02,1.02]);
xlabel('False positive rate')
ylabel('True positive rate')
title('ROC Curve with Pointwise Confidence Bounds')
legend('PCBwVA','Location','Best')
```



It might not always be possible to control the false positive rate (FPR, the X value in this example). So you might want to compute the pointwise confidence intervals on true positive rates (TPR) by threshold averaging.

```
[X1,Y1,T1] = perfcurve(species(51:end,:),mdl.Fitted.Probability,...
    'virginica','NBoot',1000);
```

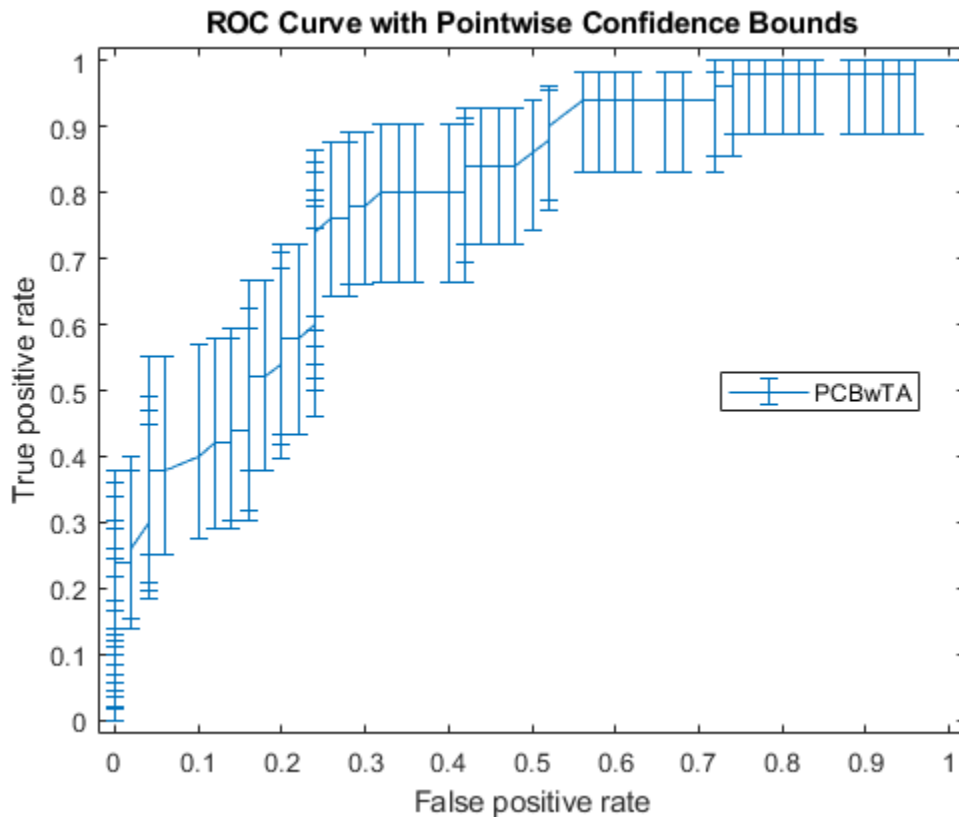
If you set 'TVals' to 'All', or if you do not specify 'TVals' or 'Xvals', then `perfcurve` returns X, Y, and T values for all scores and computes pointwise confidence bounds for X and Y using threshold averaging.

Plot the confidence bounds.

```

figure()
errorbar(X1(:,1),Y1(:,1),Y1(:,1)-Y1(:,2),Y1(:,3)-Y1(:,1));
xlim([-0.02,1.02]); ylim([-0.02,1.02]);
xlabel('False positive rate')
ylabel('True positive rate')
title('ROC Curve with Pointwise Confidence Bounds')
legend('PCBwTA','Location','Best')

```



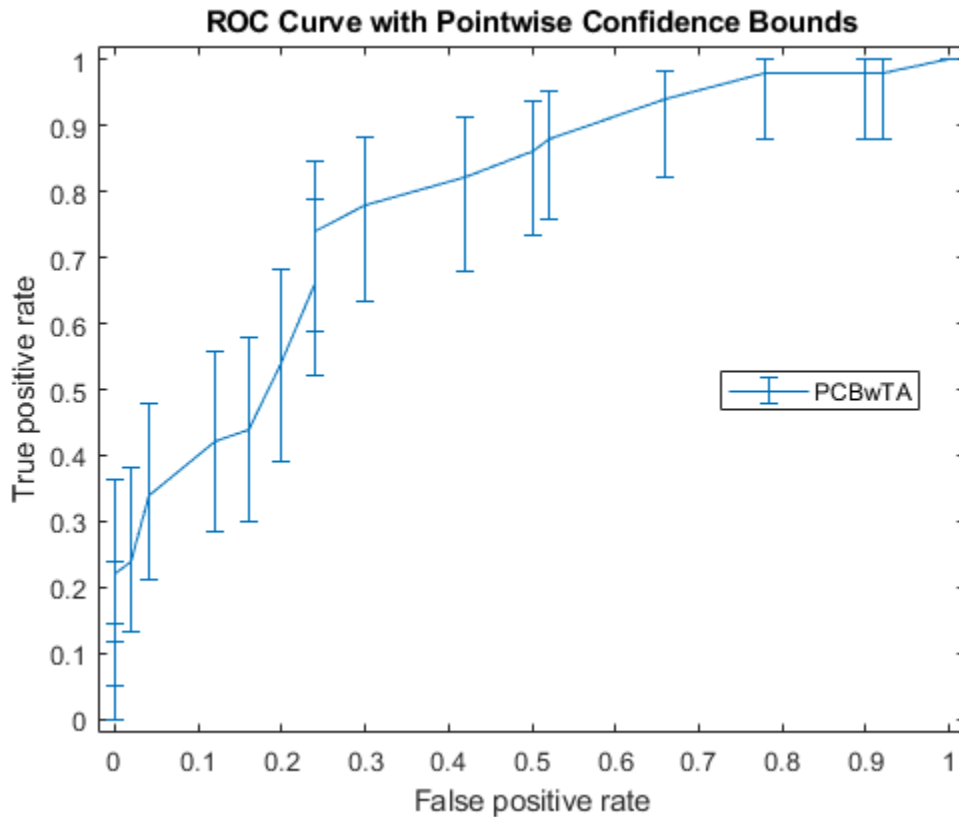
Specify the threshold values to fix and compute the ROC curve. Then plot the curve.

```

[X1,Y1,T1] = perfcurve(species(51:end,:),mdl.Fitted.Probability,...
    'virginica','NBoot',1000,'TVals',0:0.05:1);
figure()
errorbar(X1(:,1),Y1(:,1),Y1(:,1)-Y1(:,2),Y1(:,3)-Y1(:,1));

```

```
xlim([-0.02,1.02]); ylim([-0.02,1.02]);
xlabel('False positive rate')
ylabel('True positive rate')
title('ROC Curve with Pointwise Confidence Bounds')
legend('PCBwTA','Location','Best')
```



## Input Arguments

### `labels` — True class labels

numeric vector | logical vector | character matrix | cell array of character vectors | categorical array

True class labels, specified as a numeric vector, a logical vector, a character matrix, a cell array of character vectors, or a categorical array. For more information, see “Grouping Variables” on page 2-59.

Example: {'hi', 'mid', 'hi', 'low', ..., 'mid'}

Example: ['H', 'M', 'H', 'L', ..., 'M']

Data Types: single | double | logical | char | cell

### **scores** — Scores returned by a classifier

vector of floating points

Scores returned by a classifier for some sample data, specified as a vector of floating points. `scores` must have the same number of elements as `labels`.

Data Types: single | double

### **posclass** — Positive class label

numeric value | logical value | character vector | cell array of character vectors | categorical value

Positive class label, specified as a numeric value, a logical value, a character array, or a cell array of character vectors. The positive class must be a member of the input labels. The value of `posclass` that you can specify depends on the value of `labels`.

<b>labels value</b>	<b>posclass value</b>
Numeric vector	Numeric scalar
Logical vector	Logical scalar
Character matrix	Character vector
Cell array of character vectors	Character vector or cell containing character vector
Categorical vector	Categorical scalar

For example, in a cancer diagnosis problem, if a malignant tumor is the positive class, then specify `posclass` as 'malignant'.

Data Types: single | double | logical | char | cell

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'NegClass', 'versicolor', 'XCrit', 'fn', 'NBoot', 1000, 'BootType', 'per'` specifies the species `versicolor` as the negative class, the criterion for the X-coordinate as false negative, the number of bootstrap samples as 1000. It also specifies that the pointwise confidence bounds are computed using the percentile method.

### **NegClass** — List of negative classes

'all' (default) | numeric array | categorical array

List of negative classes, specified as the comma-separated pair consisting of `'NegClass'`, and a numeric array or a categorical array. By default, `perfcurve` sets `NegClass` to `'all'` and considers all nonpositive classes found in the input array of labels to be negative.

If `NegClass` is a subset of the classes found in the input array of labels, then `perfcurve` discards the instances with labels that do not belong to either positive or negative classes.

Example: `'NegClass', {'versicolor', 'setosa'}`

Data Types: `single` | `double`

### **xCrit** — Criterion to compute for x

'fpr' (default) | 'fnr' | 'tnr' | 'ppv' | 'ecost' | ...

Criterion to compute for `x`, specified as the comma-separated pair consisting of `'xCrit'` and one of the following.

Criterion	Description
<code>tp</code>	Number of true positive instances
<code>fn</code>	Number of false negative instances.
<code>fp</code>	Number of false positive instances.
<code>tn</code>	Number of true negative instances.

Criterion	Description
tp+fp	Sum of true positive and false positive instances.
rpp	Rate of positive predictions. $rpp = (tp+fp) / (tp+fn+fp+tn)$
rnp	Rate of negative predictions. $rnp = (tn+fn) / (tp+fn+fp+tn)$
accu	Accuracy. $accu = (tp+tn) / (tp+fn+fp+tn)$
tpr, or sens, or reca	True positive rate, or sensitivity, or recall. $tpr = sens = reca = tp / (tp+fn)$
fnr, or miss	False negative rate, or miss. $fnr = miss = fn / (tp+fn)$
fpr, or fall	False positive rate, or fallout, or 1 – specificity. $fpr = fall = fp / (tn+fp)$
tnr, or spec	True negative rate, or specificity. $tnr = spec = tn / (tn+fp)$
ppv, or prec	Positive predictive value, or precision. $ppv = prec = tp / (tp+fp)$
npv	Negative predictive value. $npv = tn / (tn+fn)$
ecost	Expected cost. $ecost = (tp * Cost(P P) + fn * Cost(N P) + fp * Cost(P N) + tn * Cost(N N)) / (tp+fn+fp+tn)$
Custom criterion	A custom-defined function with the input arguments (C, scale, cost), where C is a 2-by-2 confusion matrix, scale is a 2-by-1 array of class scales, and cost is a 2-by-2 misclassification cost matrix.

---

**Caution** Some of these criteria return NaN values at one of the two special thresholds, 'reject all' and 'accept all'.

---

Example: 'xCrit', 'ecost'

**YCrit** — Criterion to compute for Y

tpr (default) | same criteria options for X

Criterion to compute for Y, specified as the comma-separated pair consisting of 'YCrit' and one of the same criteria options as for X. This criterion does not have to be a monotone function of the positive class score.

Example: 'YCrit', 'ecost'

**XVals** — Values for the X criterion

'all' (default) | numeric array

Values for the X criterion, specified as the comma-separated pair consisting of 'XVals' and a numeric array.

- If you specify XVals, then `perfcurve` computes X and Y and the pointwise confidence bounds on page 32-4715 for Y (when applicable) only for the specified XVals.
- If you do not specify XVals, then `perfcurve`, computes X and Y and the values for all scores by default.

---

**Note** You cannot set XVals and TVals at the same time.

---

Example: 'XVals', [0:0.05:1]

Data Types: single | double

**TVals** — Thresholds for the positive class score

'all' (default) | numeric array

Thresholds for the positive class score, specified as the comma-separated pair consisting of 'TVals' and either 'all' or a numeric array.

- If TVals is set to 'all' or not specified, and XVals is not specified, then `perfcurve` returns X, Y, and T values for all scores and computes pointwise confidence bounds on page 32-4715 for X and Y using threshold averaging.
- If TVals is set to a numeric array, then `perfcurve` returns X, Y, and T values for the specified thresholds and computes pointwise confidence bounds for X and Y at these thresholds using threshold averaging.



---

**Note** You cannot set `XVals` and `TVals` at the same time.

---

Example: `'TVals', [0:0.05:1]`

Data Types: `single` | `double`

**UseNearest** — Indicator to use the nearest values in the data

`'on'` (default) | `'off'`

Indicator to use the nearest values in the data instead of the specified numeric `XVals` or `TVals`, specified as the comma-separated pair consisting of `'UseNearest'` and either `'on'` or `'off'`.

- If you specify numeric `XVals` and set `UseNearest` to `'on'`, then `perfcurve` returns the nearest unique X values found in the data, and it returns the corresponding values of Y and T.
- If you specify numeric `XVals` and set `UseNearest` to `'off'`, then `perfcurve` returns the sorted `XVals`.
- If you compute confidence bounds by cross validation or bootstrap, then this parameter is always `'off'`.

Example: `'UseNearest', 'off'`

**ProcessNaN** — `perfcurve` method for processing NaN scores

`'ignore'` (default) | `'addtofalse'`

`perfcurve` method for processing NaN scores, specified as the comma-separated pair consisting of `'ProcessNaN'` and `'ignore'` or `'addtofalse'`.

- If `ProcessNaN` is `'ignore'`, then `perfcurve` removes observations with NaN scores from the data.
- If `ProcessNaN` is `'addtofalse'`, then `perfcurve` adds instances with NaN scores to false classification counts in the respective class. That is, `perfcurve` always counts instances from the positive class as false negative (FN), and it always counts instances from the negative class as false positive (FP).

Example: `'ProcessNaN', 'addtofalse'`

**Prior** — Prior probabilities for positive and negative classes

`'empirical'` (default) | `'uniform'` | array with two elements

Prior probabilities for positive and negative classes, specified as the comma-separated pair consisting of 'Prior' and 'empirical', 'uniform', or an array with two elements.

If `Prior` is 'empirical', then `perfcurve` derives prior probabilities from class frequencies.

If `Prior` is 'uniform', then `perfcurve` sets all prior probabilities to be equal.

Example: 'Prior', [0.3,0.7]

Data Types: single | double | char

### **Cost — Misclassification costs**

[0 0.5;0.5 0] (default) | 2-by-2 matrix

Misclassification costs, specified as the comma-separated pair consisting of 'Cost' and a 2-by-2 matrix, containing [Cost (P|P), Cost (N|P); Cost (P|N), Cost (N|N)].

Cost (N|P) is the cost of misclassifying a positive class as a negative class. Cost (P|N) is the cost of misclassifying a negative class as a positive class. Usually, Cost (P|P) = 0 and Cost (N|N) = 0, but `perfcurve` allows you to specify nonzero costs for correct classification as well.

Example: 'Cost', [0 0.7;0.3 0]

Data Types: single | double

### **Alpha — Confidence level**

0.05 (default) | scalar value in the range 0 through 1

Confidence level for the confidence bounds, specified as the comma-separated pair consisting of 'Alpha' and a scalar value in the range 0 through 1. `perfcurve` computes  $100*(1 - \alpha)$  percent pointwise confidence bounds on page 32-4715 for X, Y, T, and AUC for a confidence level of  $\alpha$ .

Example: 'Alpha', 0.01 specifies 99% confidence bounds

Data Types: single | double

### **Weights — Observation weights**

(default) | vector of nonnegative scalar values | cell array of vectors of nonnegative scalar values

Observation weights, specified as the comma-separated pair consisting of 'Weights' and a vector of nonnegative scalar values. This vector must have as many elements as scores or labels do.

If scores and labels are in cell arrays and you need to supply Weights, the weights must be in a cell array as well. In this case, every element in Weights must be a numeric vector with as many elements as the corresponding element in scores. For example, `numel(weights{1}) == numel(scores{1})`.

When perfcurve computes the X, Y and T or confidence bounds using cross-validation, it uses these observation weights instead of observation counts.

When perfcurve computes confidence bounds using bootstrap, it samples  $N$  out of  $N$  observations with replacement, using these weights as multinomial sampling probabilities.

Data Types: `single` | `double` | `cell`

#### **NBoot** — Number of bootstrap replicas

0 (default) | positive integer

Number of bootstrap replicas for computation of confidence bounds, specified as the comma-separated pair consisting of 'NBoot' and a positive integer. The default value 0 means the confidence bounds are not computed.

If labels and scores are cell arrays, this parameter must be 0 because perfcurve can use either cross-validation or bootstrap to compute confidence bounds.

Example: `'NBoot', 500`

Data Types: `single` | `double`

#### **BootType** — Confidence interval type for bootci

'bca' (default) | 'norm' | 'per' | 'cper' | 'stud'

Confidence interval type for bootci to use to compute confidence bounds, specified as the comma-separated pair consisting of 'BootType' and one of the following:

- 'bca' — Bias corrected and accelerated percentile method
- 'norm' or 'normal' — Normal approximated interval with bootstrapped bias and standard error

- 'per' or 'percentile' — Percentile method
- 'cper' or 'corrected percentile' — Bias corrected percentile method
- 'stud' or 'student' — Studentized confidence interval

Example: 'BootType', 'cper'

#### **BootArg** — Optional input arguments for `bootci`

[] (default) |

Optional input arguments for `bootci` to compute confidence bounds, specified as the comma-separated pair consisting of 'BootArg' and one of the inputs or name-value pair arguments that `bootci` accepts.

Example: 'BootArg', {'stderr', stderr} specifies the standard error of the bootstrap statistics

#### **Options** — Options for controlling the computation of confidence intervals

[] (default) | structure array returned by `statset`

Options for controlling the computation of confidence intervals, specified as the comma-separated pair consisting of 'Options' and a structure array returned by `statset`. These options require Parallel Computing Toolbox. `perfcurve` uses this argument for computing pointwise confidence bounds only. To compute these bounds, you must pass cell arrays for `labels` and `scores` or set `NBoot` to a positive integer.

This table summarizes the available options.

Option	Description
'UseParallel'	<ul style="list-style-type: none"> <li>• <code>false</code> — Serial computation (default).</li> <li>• <code>true</code> — Parallel computation. You need Parallel Computing Toolbox for this option to work.</li> </ul>

Option	Description
'UseSubstreams'	<ul style="list-style-type: none"> <li>• false — Do not use a separate substream for each iteration (default).</li> <li>• true — Use a separate substream for each iteration to compute in parallel in a reproducible fashion. To compute reproducibly, set <code>Streams</code> to a type allowing substreams: <code>'mlfg6331_64'</code> or <code>'mrg32k3a'</code>.</li> </ul>
'Streams'	<p>A <code>RandStream</code> object, or a cell array of such objects. If you specify <code>Streams</code>, use a single object, except when:</p> <ul style="list-style-type: none"> <li>• You have an open parallel pool.</li> <li>• <code>UseParallel</code> is true.</li> <li>• <code>UseSubstreams</code> is false.</li> </ul> <p>In that case, use a cell array of the same size as the parallel pool. If a parallel pool is not open, then <code>Streams</code> must supply a single random number stream.</p>

If `'UseParallel'` is true and `'UseSubstreams'` is false, then the length of `'Streams'` must equal the number of workers used by `perfcurve`. If a parallel pool is already open, then the length of `'Streams'` is the size of the parallel pool. If a parallel pool is not already open, then MATLAB might open a pool for you, depending on your installation and preferences. To ensure more predictable results, use `parpool` and explicitly create a parallel pool before invoking `perfcurve` and setting `'Options', statset('UseParallel', true)`.

Example: `'Options', statset('UseParallel', true)`

Data Types: `struct`

## Output Arguments

**x** — *x*-coordinates for the performance curve  
vector, `fpr` (default) | *m*-by-3 matrix

$x$ -coordinates for the performance curve, returned as a vector or an  $m$ -by-3 matrix. By default,  $X$  values are the false positive rate, FPR (fallout or  $1 - \text{specificity}$ ). To change  $X$ , use the `XCrit` name-value pair argument.

- If `perfcurve` does not compute the pointwise confidence bounds on page 32-4715, or if it computes them using vertical averaging, then  $X$  is a vector.
- If `perfcurve` computes the confidence bounds using threshold averaging, then  $X$  is an  $m$ -by-3 matrix, where  $m$  is the number of fixed threshold values. The first column of  $X$  contains the mean value. The second and third columns contain the lower bound and the upper bound, respectively, of the pointwise confidence bounds.

#### **$Y$ — $y$ -coordinates for the performance curve**

vector, `tpr` (default) |  $m$ -by-3 matrix

$y$ -coordinates for the performance curve, returned as a vector or an  $m$ -by-3 matrix. By default,  $Y$  values are the true positive rate, TPR (recall or sensitivity). To change  $Y$ , use `YCrit` name-value pair argument.

- If `perfcurve` does not compute the pointwise confidence bounds on page 32-4715, then  $Y$  is a vector.
- If `perfcurve` computes the confidence bounds, then  $Y$  is an  $m$ -by-3 matrix, where  $m$  is the number of fixed  $X$  values or thresholds ( $T$  values). The first column of  $Y$  contains the mean value. The second and third columns contain the lower bound and the upper bound, respectively, of the pointwise confidence bounds.

#### **$T$ — Thresholds on classifier scores**

vector |  $m$ -by-3 matrix

Thresholds on classifier scores for the computed values of  $X$  and  $Y$ , returned as a vector or  $m$ -by-3 matrix.

- If `perfcurve` does not compute the pointwise confidence bounds on page 32-4715, or computes them using threshold averaging, then  $T$  is a vector.
- If `perfcurve` computes the confidence bounds using vertical averaging,  $T$  is an  $m$ -by-3 matrix, where  $m$  is the number of fixed  $X$  values. The first column of  $T$  contains the mean value. The second and third columns contain the lower bound, and the upper bound, respectively, of the pointwise confidence bounds.

For each threshold,  $TP$  is the count of true positive observations with scores greater than or equal to this threshold, and  $FP$  is the count of false positive observations with scores

greater than or equal to this threshold. `perfcurve` defines negative counts, `TN` and `FN`, in a similar way. The function then sorts the thresholds in the descending order that corresponds to the ascending order of positive counts.

For the  $m$  distinct thresholds found in the array of scores, `perfcurve` returns the `X`, `Y` and `T` arrays with  $m + 1$  rows. `perfcurve` sets elements `T(2:m+1)` to the distinct thresholds, and `T(1)` replicates `T(2)`. By convention, `T(1)` represents the highest 'reject all' threshold, and `perfcurve` computes the corresponding values of `X` and `Y` for `TP = 0` and `FP = 0`. The `T(end)` value is the lowest 'accept all' threshold for which `TN = 0` and `FN = 0`.

### AUC — Area under the curve

scalar value | 3-by-1 vector

Area under the curve (AUC) for the computed values of `X` and `Y`, returned as a scalar value or a 3-by-1 vector.

- If `perfcurve` does not compute the pointwise confidence bounds on page 32-4715, AUC is a scalar value.
- If `perfcurve` computes the confidence bounds using vertical averaging, AUC is a 3-by-1 vector. The first column of AUC contains the mean value. The second and third columns contain the lower bound and the upper bound, respectively, of the confidence bound.

For a perfect classifier,  $AUC = 1$ . For a classifier that randomly assigns observations to classes,  $AUC = 0.5$ .

If you set `XVals` to 'all' (default), then `perfcurve` computes AUC using the returned `X` and `Y` values.

If `XVals` is a numeric array, then `perfcurve` computes AUC using `X` and `Y` values from all distinct scores in the interval, which are specified by the smallest and largest elements of `XVals`. More precisely, `perfcurve` finds `X` values for all distinct thresholds as if `XVals` were set to 'all', and then uses a subset of these (with corresponding `Y` values) between `min(XVals)` and `max(XVals)` to compute AUC.

`perfcurve` uses trapezoidal approximation to estimate the area. If the first or last value of `X` or `Y` are NaNs, then `perfcurve` removes them to allow calculation of AUC. This takes care of criteria that produce NaNs for the special 'reject all' or 'accept all'

thresholds, for example, positive predictive value (PPV) or negative predictive value (NPV).

**OPTROCPT — Optimal operating point of the ROC curve**

1-by-2 array

Optimal operating point of the ROC curve, returned as a 1-by-2 array with false positive rate (FPR) and true positive rate (TPR) values for the optimal ROC operating point.

`perfcurve` computes `OPTROCPT` for the standard ROC curve only, and sets to NaNs otherwise. To obtain the optimal operating point for the ROC curve, `perfcurve` first finds the slope,  $S$ , using

$$S = \frac{\text{Cost}(P | N) - \text{Cost}(N | N) * N}{\text{Cost}(N | P) - \text{Cost}(P | P) * P}$$

- $\text{Cost}(N | P)$  is the cost of misclassifying a positive class as a negative class.  $\text{Cost}(P | N)$  is the cost of misclassifying a negative class as a positive class.
- $P = TP + FN$  and  $N = TN + FP$ . They are the total instance counts in the positive and negative class, respectively.

`perfcurve` then finds the optimal operating point by moving the straight line with slope  $S$  from the upper left corner of the ROC plot ( $FPR = 0$ ,  $TPR = 1$ ) down and to the right, until it intersects the ROC curve.

**SUBY — Values for negative subclasses**

array

Values for negative subclasses, returned as an array.

- If you specify only one negative class, then `SUBY` is identical to `Y`.
- If you specify  $k$  negative classes, then `SUBY` is a matrix of size  $m$ -by- $k$ , where  $m$  is the number of returned values for `X` and `Y`, and  $k$  is the number of negative classes. `perfcurve` computes `Y` values by summing counts over all negative classes.

`SUBY` gives values of the `Y` criterion for each negative class separately. For each negative class, `perfcurve` places a new column in `SUBY` and fills it with `Y` values for true negative (TN) and false positive (FP) counted just for this class.

**SUBYNAMES — Negative class names**

cell array



Negative class names, returned as a cell array.

- If you provide an input array of negative class names, `NegClass`, then `perfcurve` copies names into `SUBYNAMES`.
- If you do not provide `NegClass`, then `perfcurve` extracts `SUBYNAMES` from the input labels. The order of `SUBYNAMES` is the same as the order of columns in `SUBY`. That is, `SUBY(:,1)` is for negative class `SUBYNAMES{1}`, `SUBY(:,2)` is for negative class `SUBYNAMES{2}`, and so on.

## Algorithms

### Pointwise Confidence Bounds

If you supply cell arrays for `labels` and `scores`, or if you set `NBoot` to a positive integer, then `perfcurve` returns pointwise confidence bounds for `X`, `Y`, `T`, and `AUC`. You cannot supply cell arrays for `labels` and `scores` and set `NBoot` to a positive integer at the same time.

`perfcurve` resamples data to compute confidence bounds using either cross validation or bootstrap.

- **Cross-validation** — If you supply cell arrays for `labels` and `scores`, then `perfcurve` uses cross-validation and treats elements in the cell arrays as cross-validation folds. `labels` can be a cell array of numeric vectors, logical vectors, character matrices, cell arrays of character vectors, or categorical vectors. All elements in `labels` must have the same type. `scores` can be a cell array of numeric vectors. The cell arrays for `labels` and `scores` must have the same number of elements. The number of labels in cell `j` of `labels` must be equal to the number of scores in cell `j` of `scores` for any `j` in the range from 1 to the number of elements in `scores`.
- **Bootstrap** — If you set `NBoot` to a positive integer `n`, `perfcurve` generates `n` bootstrap replicas to compute pointwise confidence bounds. If you use `XCrit` or `YCrit` to set the criterion for `X` or `Y` to an anonymous function, `perfcurve` can compute confidence bounds only using bootstrap.

`perfcurve` estimates the confidence bounds using one of two methods:

- Vertical averaging (VA) — `perfcurve` estimates confidence bounds on  $Y$  and  $T$  at fixed values of  $X$ . That is, `perfcurve` takes samples of the ROC curves for fixed  $X$  values, averages the corresponding  $Y$  and  $T$  values, and computes the standard errors. You can use the `XVals` name-value pair argument to fix the  $X$  values for computing confidence bounds. If you do not specify `XVals`, then `perfcurve` computes the confidence bounds at all  $X$  values.
- Threshold averaging (TA) — `perfcurve` takes samples of the ROC curves at fixed thresholds  $T$  for the positive class score, averages the corresponding  $X$  and  $Y$  values, and estimates the confidence bounds. You can use the `TVals` name-value pair argument to use this method for computing confidence bounds. If you set `TVals` to 'all' or do not specify `TVals` or `XVals`, then `perfcurve` returns  $X$ ,  $Y$ , and  $T$  values for all scores and computes pointwise confidence bounds for  $Y$  and  $X$  using threshold averaging.

When you compute the confidence bounds,  $Y$  is an  $m$ -by-3 array, where  $m$  is the number of fixed  $X$  values or thresholds ( $T$  values). The first column of  $Y$  contains the mean value. The second and third columns contain the lower bound and the upper bound, respectively, of the pointwise confidence bounds.  $AUC$  is a row vector with three elements, following the same convention. If `perfcurve` computes the confidence bounds using VA, then  $T$  is an  $m$ -by-3 matrix, and  $X$  is a column vector. If `perfcurve` uses TA, then  $X$  is an  $m$ -by-3 matrix and  $T$  is a column-vector.

`perfcurve` returns pointwise confidence bounds. It does not return a simultaneous confidence band for the entire curve.

## References

- [1] T. Fawcett. “ROC Graphs: Notes and Practical Considerations for Researchers”, 2004.
- [2] Zweig, M., and G. Campbell. “Receiver-Operating Characteristic (ROC) Plots: A Fundamental Evaluation Tool in Clinical Medicine.” *Clin. Chem.* 1993, 39/4, pp. 561–577 .
- [3] Davis, J., and M. Goadrich. “The Relationship Between Precision-Recall and ROC Curves.” *Proceedings of ICML '06*, 2006, pp. 233–240.
- [4] Moskowitz, C., and M. Pepe. “Quantifying and comparing the predictive accuracy of continuous prognostic factors for binary outcomes.” *Biostatistics*, 2004, 5, pp. 113–127.

- [5] Huang, Y., M. Pepe, and Z. Feng. “Evaluating the Predictiveness of a Continuous Marker.” *U. Washington Biostatistics Paper Series*, 2006, 250–261.
- [6] Briggs, W., and R. Zaretzki. “The Skill Plot: A Graphical Technique for Evaluating Continuous Diagnostic Tests.” *Biometrics*, 2008, 63, pp. 250 – 261.
- [7] R. Bettinger. “Cost-Sensitive Classifier Selection Using the ROC Convex Hull Method.” *SAS Institute*.

## See Also

`bootci` | `classify` | `fitcnb` | `fitctree` | `fitrtree` | `glmfit` | `mnrfit`

## Topics

“Performance Curves” on page 17-3

Introduced in R2009a

## perms

Enumeration of permutations

### Syntax

```
P = perms(v)
```

### Description

`P = perms(v)`, where `v` is a row vector of length `n`, creates a matrix whose rows consist of all possible permutations of the `n` elements of `v`. The matrix `P` contains  $n!$  rows and `n` columns.

`perms` is only practical when `n` is less than about 11 (for `n = 11`, the output takes over 3 gigabytes).

### Examples

```
perms([2 4 6])
```

```
ans =
```

```
6 4 2
6 2 4
4 6 2
4 2 6
2 4 6
2 6 4
```

### See Also

```
combnk
```

**Introduced before R2006a**

## piecewisedistribution class

Piecewise-defined distributions

### Construction

`piecewisedistribution` is an abstract class. To construct a `piecewisedistribution` object, use the subclass constructor, `paretotails`.

### Methods

<code>boundary</code>	Piecewise distribution boundaries
<code>cdf</code>	Cumulative distribution function for piecewise distribution
<code>disp</code>	Display <code>piecewisedistribution</code> object
<code>display</code>	Display <code>piecewisedistribution</code> object
<code>icdf</code>	Inverse cumulative distribution function for piecewise distribution
<code>nsegments</code>	Number of segments
<code>pdf</code>	Probability density function for piecewise distribution
<code>random</code>	Random numbers from piecewise distribution
<code>segment</code>	Segments containing values

### Properties

Objects of the `piecewisedistribution` class have no properties accessible by dot indexing, `get` methods, or `set` methods. To obtain information about a `piecewisedistribution` object, use the appropriate method.

## Copy Semantics

Value. To learn how this affects your use of the class, see [Comparing Handle and Value Classes \(MATLAB\)](#) in the [MATLAB Object-Oriented Programming](#) documentation.

## piecewisedistribution

**Class:** `piecewisedistribution`

Create piecewise distribution object

### Description

`piecewisedistribution` is an abstract class, and you cannot create instances of it directly. You can create `paretotails` objects that are derived from this class.

### See Also

`paretotails`

Introduced in R2009a



# plot

**Class:** clustering.evaluation.ClusterCriterion

**Package:** clustering.evaluation

Plot clustering evaluation object criterion values

## Syntax

```
plot(eva)
h = plot(eva)
```

## Description

`plot(eva)` displays a plot of the criterion values versus the number of clusters, based on the values stored in the clustering evaluation object `eva`.

`h = plot(eva)` returns a handle to the plot line.

## Input Arguments

**eva** — Clustering evaluation data

clustering evaluation object

Clustering evaluation data, specified as a clustering evaluation object. Create a clustering evaluation object using `evalclusters`.

## Output Arguments

**h** — Handle to plot line

scalar value

Handle to the plot line, returned as a scalar value.

## Examples

### Plot the Clustering Evaluation Criterion Values

Plot the criterion values versus the number of clusters for each clustering solution stored in a clustering evaluation object.

Load the sample data.

```
load fisheriris
```

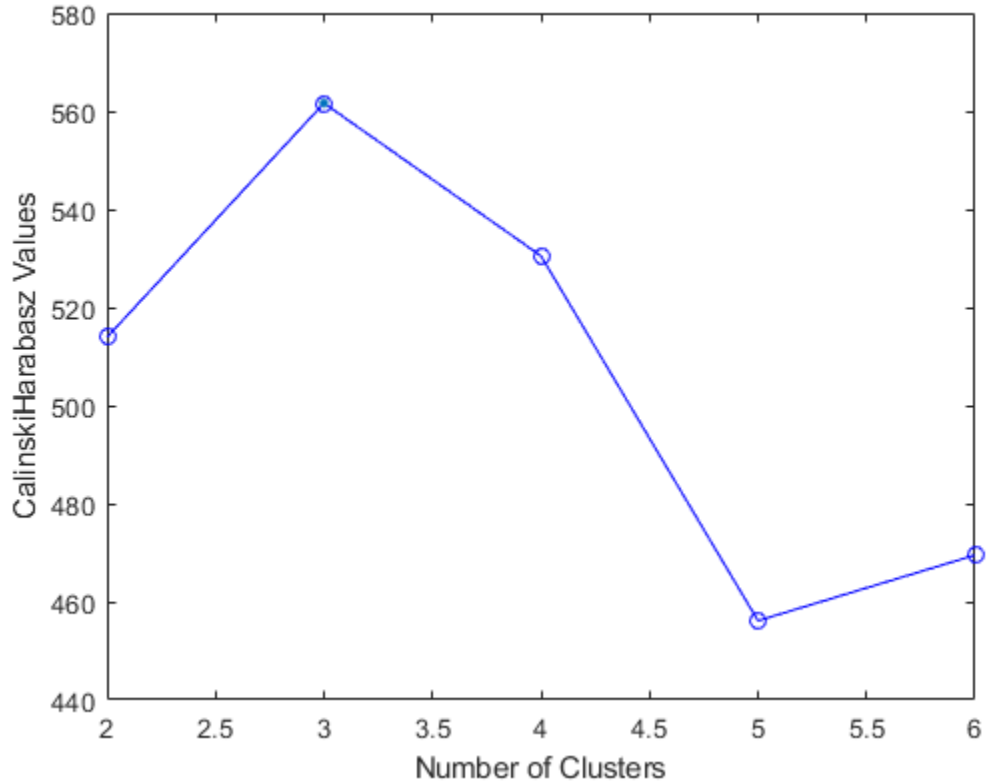
The data contains length and width measurements from the sepals and petals of three species of iris flowers.

Create a clustering evaluation object. Cluster the data using `kmeans`, and evaluate the optimal number of clusters using the Calinski-Harabasz criterion.

```
rng('default'); % For reproducibility
eva = evalclusters(meas, 'kmeans', 'CalinskiHarabasz', 'KList', [1:6]);
```

Plot the Calinski-Harabasz criterion values for each number of clusters tested.

```
figure;
plot(eva);
```



The plot shows that the highest Calinski-Harabasz value occurs at three clusters, suggesting that the optimal number of clusters is three.

## See Also

`evalclusters`

## plot

**Class:** LinearModel

Scatter plot or added variable plot of linear model

## Syntax

```
plot mdl
h = plot mdl
```

## Description

`plot(mdl)` creates a plot of the full, fitted linear model, `mdl`. The plot type depends on the number of predictor variables.

- If there is just one predictor variable, `plot` creates a scatter plot of the data along with a fitted curve and confidence bounds.
- If there are multiple predictor variables, `plot` creates an added variable plot.
- If there are no predictors, `plot` creates a histogram of the residuals.

`h = plot(mdl)` returns handles to the lines in the plot.

## Input Arguments

**mdl** — Full, fitted linear regression model

LinearModel object

Full, fitted linear regression model, specified as a `LinearModel` object constructed using `fitlm` or `stepwiselm`.

---

## Output Arguments

### **h** — Graphics handles

vector of graphics handle objects

Graphics handles, returned as a vector of graphics handles corresponding to the lines or patches in the plot.

## Examples

### Create an Added Variable Plot

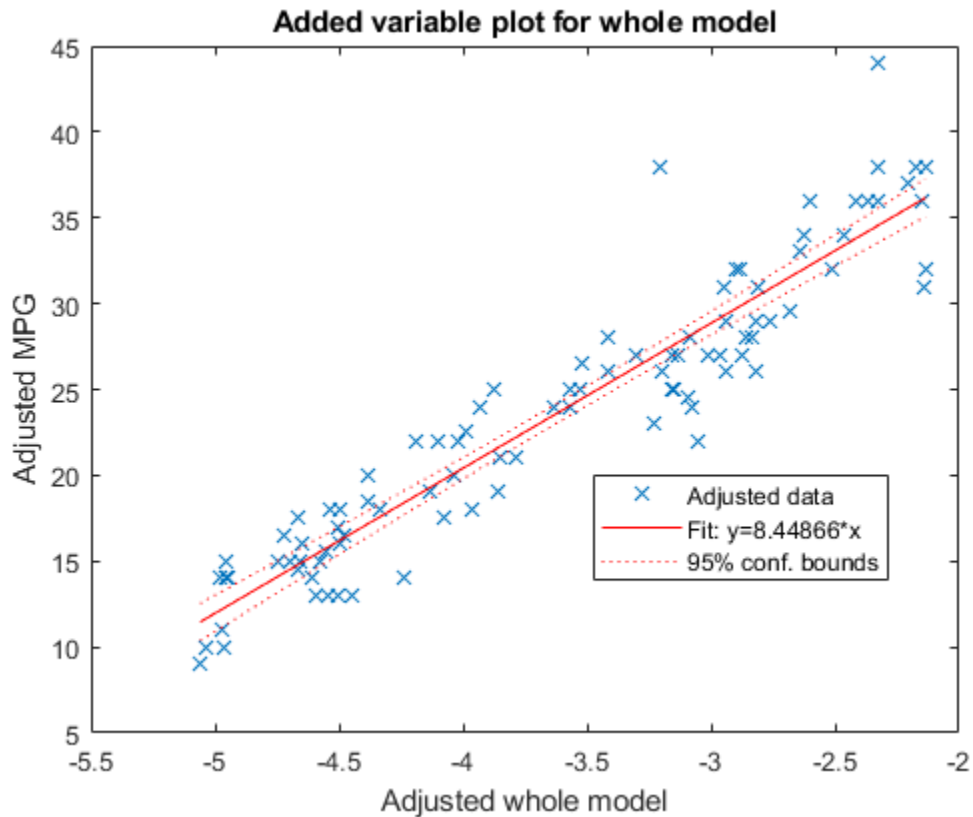
Create a model of car mileage as a function of weight and model year. Then create a plot to see the significance of the model.

Create a linear model of mileage from the `carsmall` data.

```
load carsmall
tbl = table(MPG,Weight);
tbl.Year = ordinal(Model_Year);
mdl = fitlm(tbl,'MPG ~ Year + Weight^2');
```

Create an added variable plot.

```
plot(mdl)
```



The plot illustrates that the model is significant because a horizontal line does not fit between the confidence bounds.

## Definitions

### Added Variable Plot and Adjust Response

An added variable plot illustrates the incremental effect on the response of specified terms by removing the effects of all other terms. The slope of the fitted line is the coefficient of the linear combination of the specified terms projected onto the best-fitting

direction. The adjusted response includes the constant (intercept) terms, and averages out all other terms.

## Alternatives

Use `plotAdded` to select particular predictors for an added variable plot.

## See Also

`LinearModel` | `plotAdded`

## Topics

“Linear Regression” on page 11-11

# plot

**Class:** RepeatedMeasuresModel

Plot data with optional grouping

## Syntax

```
plot(rm)
plot(rm, Name, Value)
H = plot( ___ )
```

## Description

`plot(rm)` plots the measurements in the repeated measures model `rm` for each subject as a function of time. If there is a single numeric within-subjects factor, `plot` uses the values of that factor as the time values. Otherwise, `plot` uses the discrete values 1 through `r` as the time values, where `r` is the number of repeated measurements.

`plot(rm, Name, Value)` also plots the measurements in the repeated measures model `rm`, with additional options specified by one or more `Name, Value` pair arguments.

For example, you can specify the factors to group by or change the line colors.

`H = plot( ___ )` returns handles, `H`, to the plotted lines.

## Input Arguments

**rm** — Repeated measures model

RepeatedMeasuresModel object

Repeated measures model, returned as a RepeatedMeasuresModel object.

For properties and methods of this object, see RepeatedMeasuresModel.



## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

### **Group** — Name of between-subject factor or factors

character vector | cell array of character vectors

Name of between-subject factor or factors, specified as the comma-separated pair consisting of 'Group' and a character vector or cell array of character vectors. This name-value pair argument groups the lines according to the factor values.

For example, if you have two between-subject factors, drug and sex, and you want to group the lines in the plot according to them, you can specify these factors as follows.

Example: 'Group', {'Drug', 'Sex'}

Data Types: char | cell

### **Marker** — Marker to use for each group

cell array of character vectors

Marker to use for each group, specified as the comma-separated pair consisting of 'Marker' and a cell array of character vectors.

For example, if you have two between-subject factors, drug and sex, with each having two groups, you can specify `o` as the marker for the groups of drug and `x` as the marker for the groups of sex as follows.

Example: 'Marker', {'o', 'o', 'x', 'x'}

Data Types: cell

### **Color** — Color for each group

character vector | cell array of character vectors | rows of a three-column RGB matrix

Color for each group, specified as the comma-separated pair consisting of 'Color' and a character vector, cell array of character vectors, or rows of a three-column RGB matrix.

For example, if you have two between-subject factors, drug and sex, with each having two groups, you can specify red as the color for the groups of drug and blue as the color for the groups of sex as follows.

Example: `'Color', 'rrbb'`

Data Types: `single | double | cell`

### **LineStyle** — Line style for each group

cell array of character vectors

Line style for each group, specified as the comma-separated pair consisting of `'LineStyle'` and a cell array of character vectors.

For example, if you have two between-subject factors, drug and sex, with each having two groups, you can specify `-` as the line style of one group and `:` as the line style for the other group as follows.

Example: `'LineStyle', {'-' ':' '-' ':'}`

Data Types: `cell`

## Output Arguments

### **h** — Handle to plotted lines

handle

Handle to plotted lines, returned as a handle.

## Examples

### **Plot Data by Group**

Load the sample data.

```
load fisheriris
```

The column vector `species` consists of iris flowers of three different species: `setosa`, `versicolor`, and `virginica`. The double matrix `meas` consists of four types of measurements on the flowers: the length and width of sepals and petals in centimeters, respectively.

Store the data in a table array.

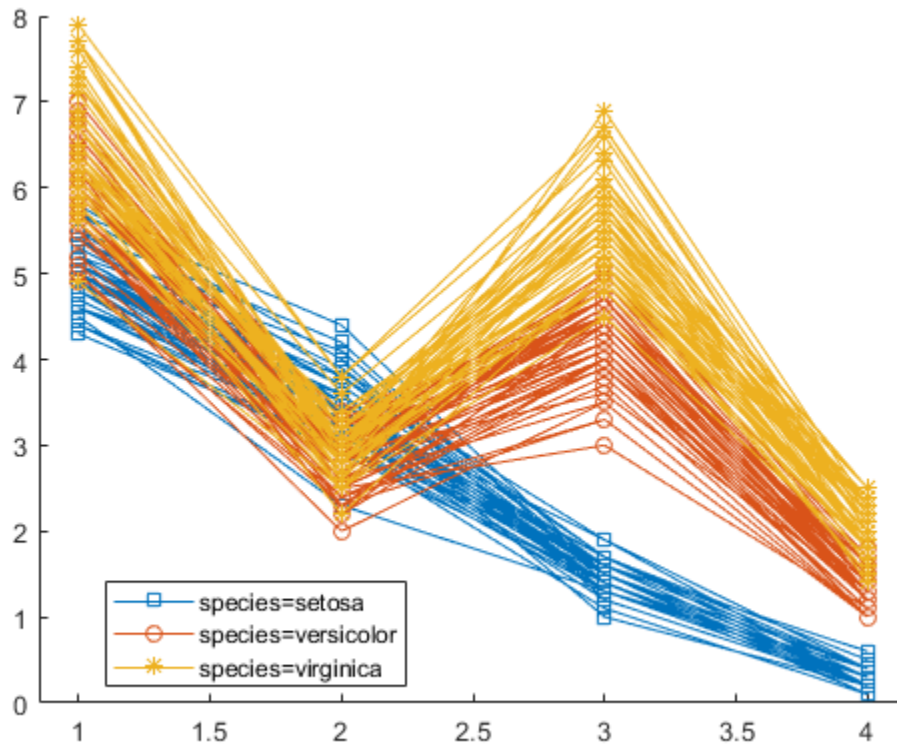
```
t = table(species,meas(:,1),meas(:,2),meas(:,3),meas(:,4),...  
'VariableNames',{'species','meas1','meas2','meas3','meas4'});  
Meas = dataset([1 2 3 4]','VarNames',{'Measurements'});
```

Fit a repeated measures model, where the measurements are the responses and the species is the predictor variable.

```
rm = fitrm(t,'meas1-meas4~species','WithinDesign',Meas);
```

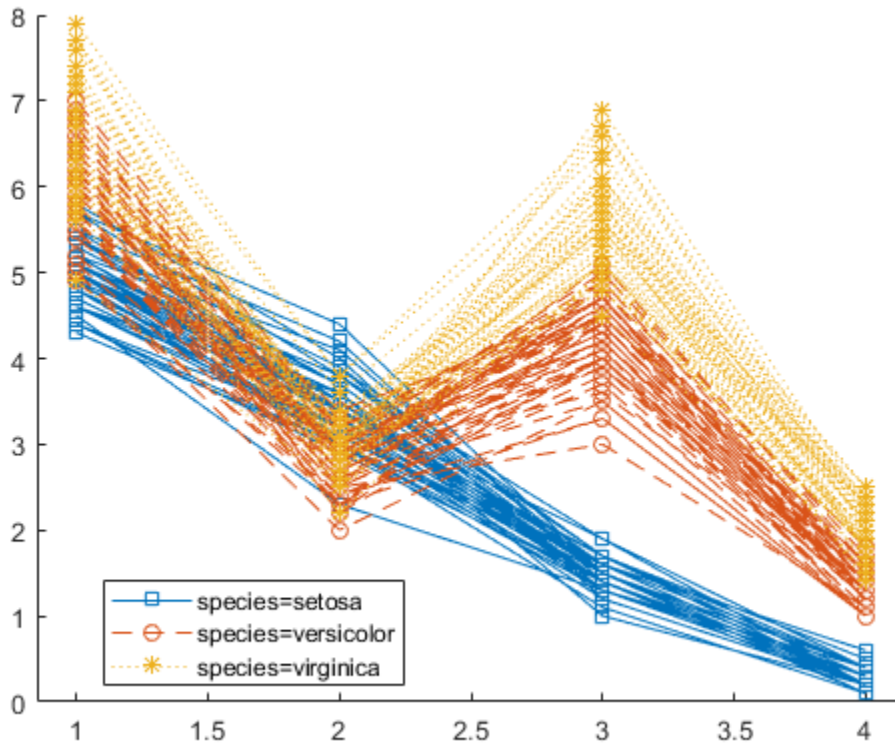
Plot data grouped by the factor species.

```
plot(rm,'group','species')
```



Change the line style for each group.

```
plot(rm, 'group', 'species', 'LineStyle', {'-', '--', ':'})
```



### Plot Data Grouped by Two Factors

Load the sample data.

```
load repeatedmeas
```

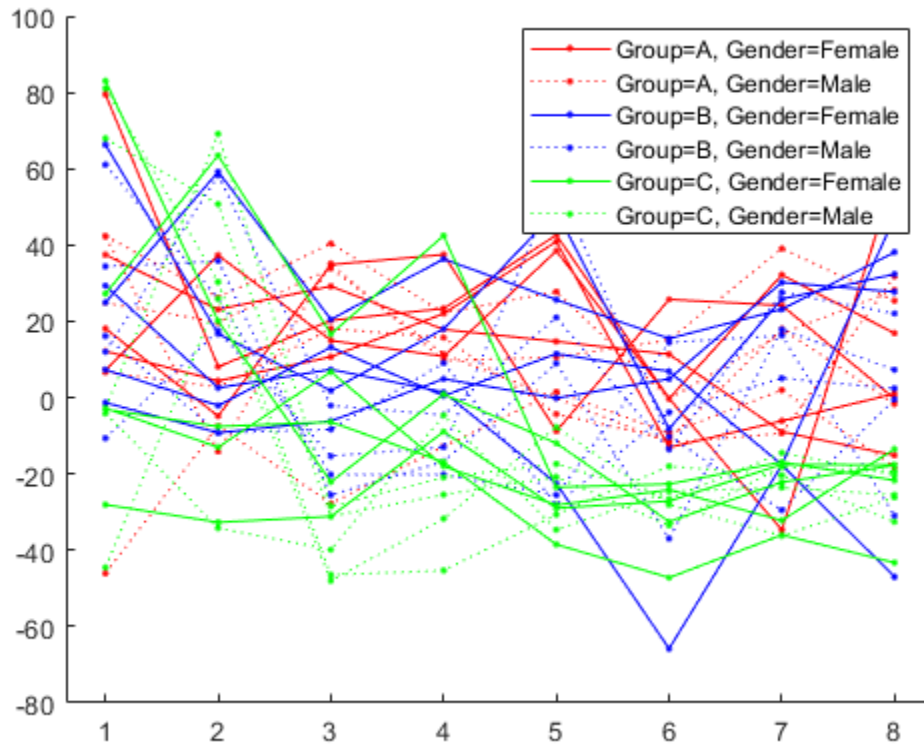
The table `between` includes the between-subject variables `age`, `IQ`, `group`, `gender`, and eight repeated measures `y1` through `y8` as responses. The table `within` includes the within-subject variables `w1` and `w2`. This is simulated data.

Fit a repeated measures model, where the repeated measures y1 through y8 are the responses, and age, IQ, group, gender, and the group-gender interaction are the predictor variables. Also specify the within-subject design matrix.

```
rm = fitrm(between, 'y1-y8 ~ Group*Gender + Age + IQ', 'WithinDesign', within);
```

Plot data with Group coded by color and Gender coded by line type.

```
plot(rm, 'group', {'Group' 'Gender'}, 'Color', 'rrbbgg', ...  
      'LineStyle', {'-' ':' '-' ':' '-' ':'}, 'Marker', '.')
```



## See Also

`fitrm` | `multcompare` | `plotprofile`

# plotAdded

**Class:** LinearModel

Added variable plot or leverage plot for linear model

## Syntax

```
plotAdded mdl
plotAdded mdl, coef
h = plotAdded mdl, ____
h = plotAdded mdl, coef, Name, Value
```

## Description

`plotAdded(mdl)` produces a generalized added variable plot for all terms in the full, fitted regression model `mdl`, except the constant term.

`plotAdded(mdl, coef)` produces an added variable plot for the `coef` terms in `mdl`, after adjusting for all other terms.

`h = plotAdded(mdl, ____)` returns handles to the lines in the plot, using any of the previous syntaxes.

`h = plotAdded(mdl, coef, Name, Value)` plots with additional options specified by one or more `Name, Value` pair arguments.

## Input Arguments

**mdl** — Full, fitted linear regression model

LinearModel object

Full, fitted linear regression model, specified as a `LinearModel` object constructed using `fitlm` or `stepwiselm`.

**coef** — Coefficients in regression model

character vector | vector of coefficient numbers

Coefficients in regression model `mdl`, specified as one of the following:

- Character vector giving a single coefficient name
- Vector of coefficient numbers in the `mdl.CoefficientNames` property.

**Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

---

**Note** The plot property name-value pairs apply to the first returned handle `h(1)`.

---

**Color** — Color of line or marker`ColorSpec` specificationColor of the line or marker, specified as the comma-separated pair consisting of 'Color' and a `ColorSpec` specification. For details, see `linespec`.**LineStyle** — Line style`Chart Line Properties` specificationLine style, specified as the comma-separated pair consisting of 'LineStyle' and a `Chart Line` specification. For details, see `linespec`.**LineWidth** — Width of line or edges

0.5 (default) | positive numeric value

Width of the line or edges of filled area, in points, specified as the comma-separated pair consisting of 'LineWidth' and a positive numeric value. One point is equal to 1/72 inch.

**MarkerEdgeColor** — Color of marker or edge`ColorSpec` specification



Color of the marker or edge color for filled markers, specified as the comma-separated pair consisting of 'MarkerEdgeColor' and a ColorSpec specification. For details, see linespec.

**MarkerFaceColor — Color of marker face**

ColorSpec specification

Color of the marker face for filled markers, specified as the comma-separated pair consisting of 'MarkerFaceColor' and a ColorSpec specification. For details, see linespec.

**MarkerSize — Size of marker**

positive numeric value

Size of the marker in points, specified as the comma-separated pair consisting of 'MarkerSize' and a positive numeric value. One point is 1/72 inch.

## Output Arguments

**h — Graphics handles**

vector of graphics handle objects

Graphics handles, returned as a vector of graphics handles corresponding to the lines or patches in the plot.

## Examples

**Create an Added Variable Plot**

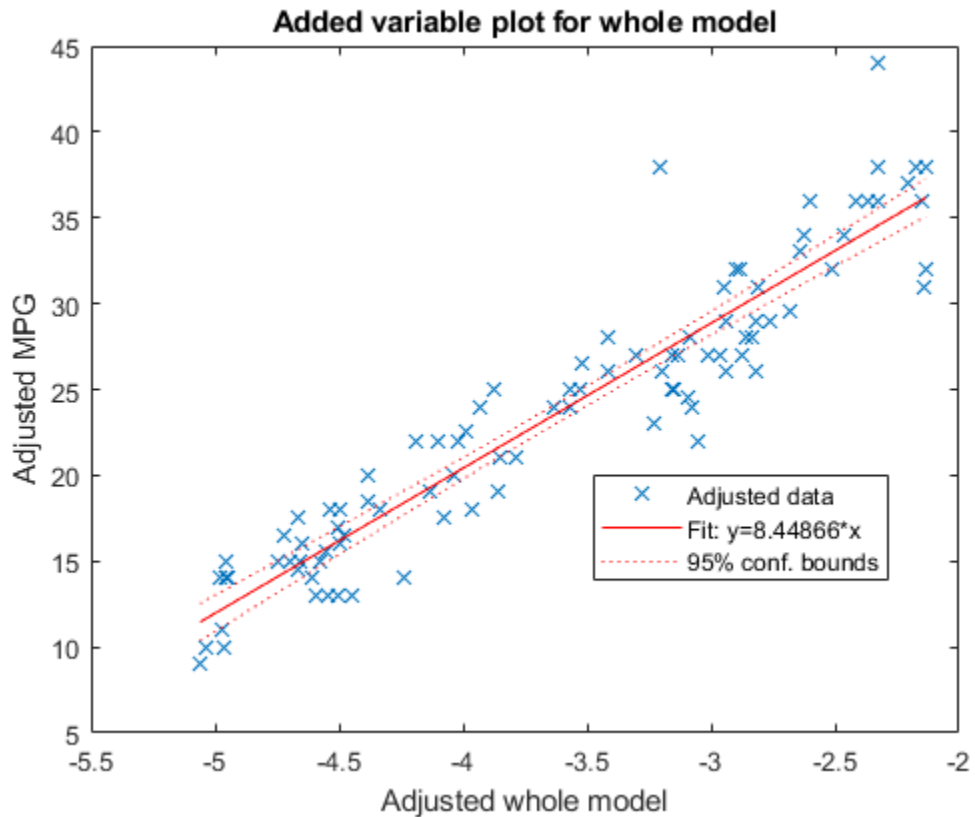
Create a model of car mileage as a function of weight and model year. Then create a plot to see the significance of the model.

Create a linear model of mileage from the carsmall data.

```
load carsmall
tbl = table(MPG,Weight);
tbl.Year = ordinal(Model_Year);
mdl = fitlm(tbl,'MPG ~ Year + Weight^2');
```

Create an added variable plot.

```
plot mdl)
```



The plot illustrates that the model is significant because a horizontal line does not fit between the confidence bounds.

### Create an Added Variable Plot for Particular Variables

Create a model of car mileage as a function of weight and model year. Then create a plot to see the effect of the weight terms (`Weight` and `Weight^2`).

Create a linear model of mileage from the `carsmall` data.

```
load carsmall
tbl = table(MPG,Weight);
tbl.Year = ordinal(Model_Year);
mdl = fitlm(tbl,'MPG ~ Year + Weight^2');
```

Find the terms in the model corresponding to the Weight and Weight<sup>2</sup>.

```
mdl.CoefficientNames
```

```
ans =
```

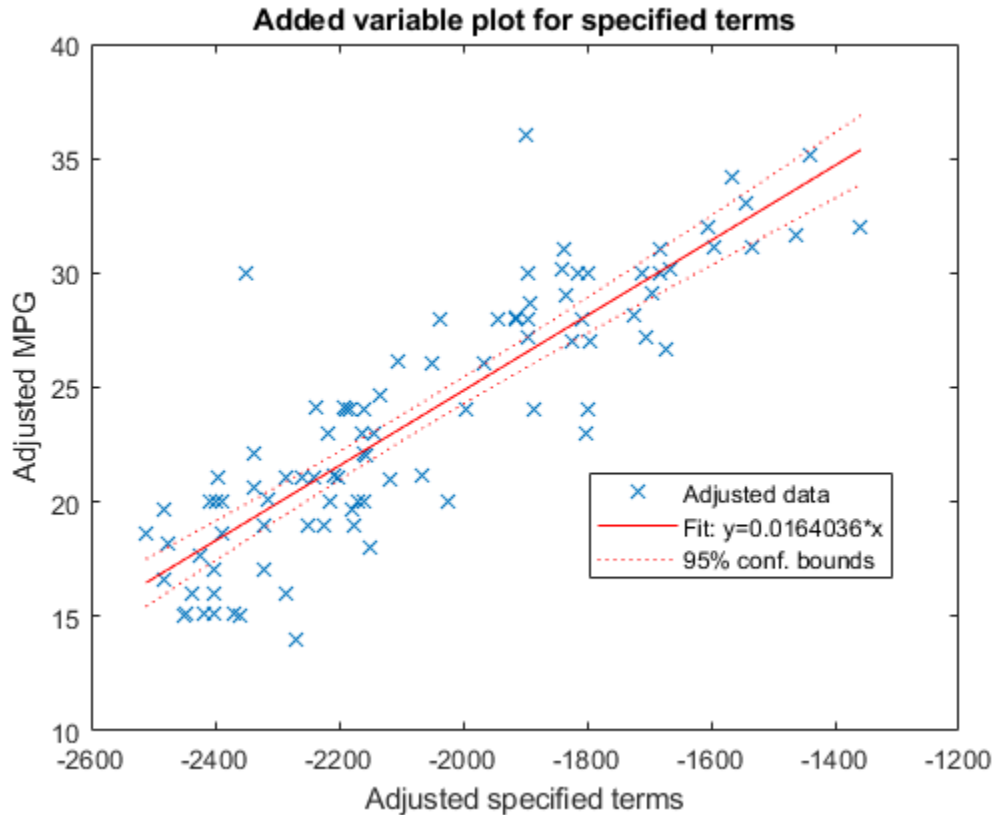
```
1x5 cell array
```

```
{'(Intercept)'} {'Weight'} {'Year_76'} {'Year_82'} {'Weight^2'}
```

The weight terms are 2 and 5.

Create an added variable plot with the weight terms.

```
coef = [2 5];
plotAdded(mdl,coef)
```



The plot illustrates that the weight terms are significant because a horizontal line does not fit between the confidence bounds.

- “Plots to Understand Terms Effects” on page 11-34

## Definitions

### Added Variable Plot and Adjusted Response

An added variable plot illustrates the incremental effect on the response of specified terms by removing the effects of all other terms. The slope of the fitted line is the

coefficient of the linear combination of the specified terms projected onto the best-fitting direction. The adjusted response includes the constant (intercept) terms, and averages out all other terms.

## Tips

- For many plots, the Data Cursor tool in the figure window displays the  $x$  and  $y$  values for any data point, along with the observation name or number.

## See Also

`LinearModel` | `plot`

## Topics

“Plots to Understand Terms Effects” on page 11-34

“Linear Regression” on page 11-11

## plotAdjustedResponse

**Class:** LinearModel

Adjusted response plot for linear regression model

### Syntax

```
plotAdjustedResponse mdl, var
h = plotAdjustedResponse mdl, var
h = plotAdjustedResponse mdl, var, Name, Value
```

### Description

`plotAdjustedResponse(mdl, var)` gives an adjusted response plot for the variable `var` in the `mdl` regression model.

`h = plotAdjustedResponse(mdl, var)` returns handles to the lines in the plot.

`h = plotAdjustedResponse(mdl, var, Name, Value)` plots with additional options specified by one or more `Name, Value` pair arguments.

### Input Arguments

**mdl** — Full, fitted linear regression model

LinearModel object

Full, fitted linear regression model, specified as a `LinearModel` object constructed using `fitlm` or `stepwiselm`.

**var** — Variable for adjusted response plot

variable name | integer value

Variable for adjusted response plot, specified as a variable name or an integer value representing the index of a variable in `mdl.CoefficientNames`.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

---

**Note** The plot property name-value pairs apply to the first returned handle `h(1)`.

---

### **Color** — Color of line or marker

`ColorSpec` specification

Color of the line or marker, specified as the comma-separated pair consisting of 'Color' and a `ColorSpec` specification. For details, see `linespec`.

### **LineStyle** — Line style

`Chart Line Properties` specification

Line style, specified as the comma-separated pair consisting of 'LineStyle' and a `Chart Line` specification. For details, see `linespec`.

### **LineWidth** — Width of line or edges

0.5 (default) | positive numeric value

Width of the line or edges of filled area, in points, specified as the comma-separated pair consisting of 'LineWidth' and a positive numeric value. One point is equal to 1/72 inch.

### **MarkerEdgeColor** — Color of marker or edge

`ColorSpec` specification

Color of the marker or edge color for filled markers, specified as the comma-separated pair consisting of 'MarkerEdgeColor' and a `ColorSpec` specification. For details, see `linespec`.

### **MarkerFaceColor** — Color of marker face

`ColorSpec` specification

Color of the marker face for filled markers, specified as the comma-separated pair consisting of 'MarkerFaceColor' and a ColorSpec specification. For details, see `linespec`.

**MarkerSize — Size of marker**

positive numeric value

Size of the marker in points, specified as the comma-separated pair consisting of 'MarkerSize' and a positive numeric value. One point is 1/72 inch.

## Output Arguments

**h — Graphics handles**

vector of graphics handle objects

Graphics handles, returned as a vector of graphics handles corresponding to the lines or patches in the plot.

## Examples

**Plot Adjusted Responses**

Plot the adjusted responses of a fitted linear model.

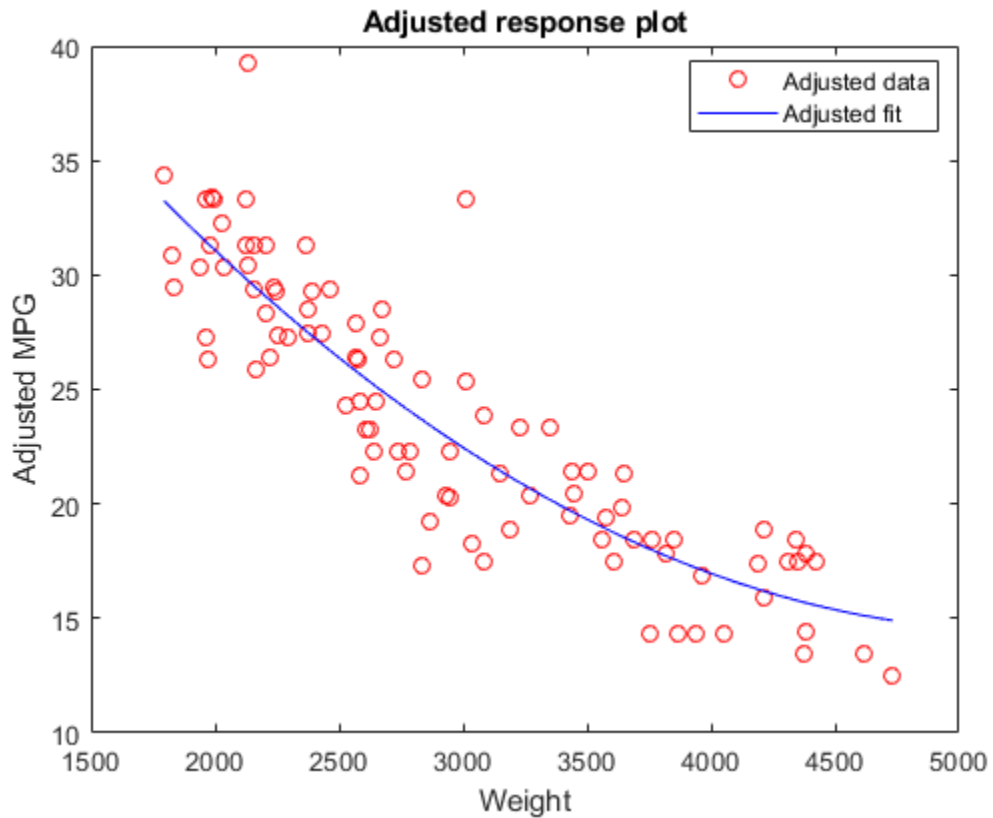
Load the `carsmall` data and fit a linear model of the mileage as a function of model year, weight, and weight squared.

```
load carsmall
tbl = table(MPG,Weight);
tbl.Year = categorical(Model_Year);
mdl = fitlm(tbl,'MPG ~ Year + Weight^2');
```

Plot the effect of 'Weight' averaged over Year values.

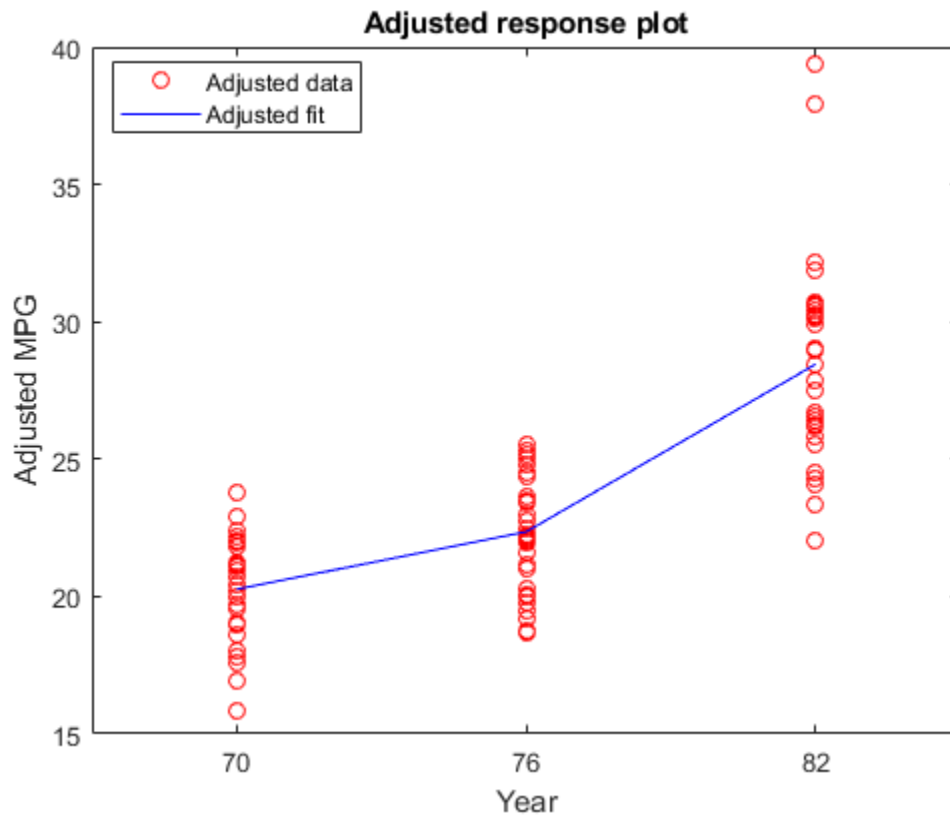
```
plotAdjustedResponse(mdl,'Weight')
```





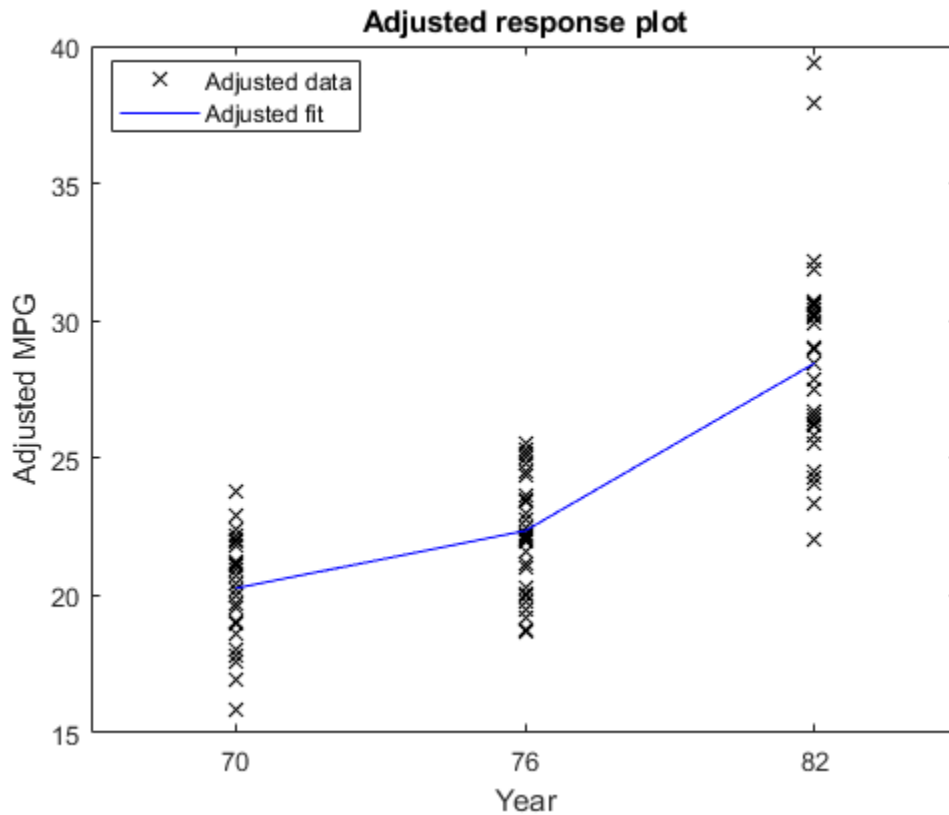
Plot the effect of Year averaged over 'Weight' values. Include the h output.

```
h = plotAdjustedResponse mdl, 'Year';
```



Change the adjusted data to black x instead of red o.

```
h(1).Marker = 'x';  
h(1).Color = 'k';
```



## Definitions

### Adjusted Response Plot

The adjusted response plot shows the fitted response as a function of `var`, with the other predictors averaged out by averaging the fitted values over the data used in the fit. Adjusted data points are computed by adding the residual to the adjusted fitted value for each observation.

## Tips

- For many plots, the Data Cursor tool in the figure window displays the  $x$  and  $y$  values for any data point, along with the observation name or number.

## See Also

`LinearModel` | `plotAdded` | `plotEffects` | `plotInteraction`

## Topics

“Linear Regression” on page 11-11

# plot

Plot Bayesian optimization results

## Syntax

```
plot(results, 'all')  
plot(results, plotFcn1, plotFcn2, ...)
```

## Description

`plot(results, 'all')` calls all predefined plot functions on `results`.

`plot(results, plotFcn1, plotFcn2, ...)` calls the listed plot functions on `results`.

## Examples

### Plot After Optimization

This example shows how to plot the error model and the best objective trace after the optimization has finished. The objective function for this example throws an error for points with norm larger than 2.

```
function f = makeanerror(x)  
f = x.x1 - x.x2 - sqrt(4-x.x1^2-x.x2^2);
```

```
fun = @makeanerror;
```

Create the variables for optimization.

```
var1 = optimizableVariable('x1', [-5, 5]);  
var2 = optimizableVariable('x2', [-5, 5]);  
vars = [var1, var2];
```

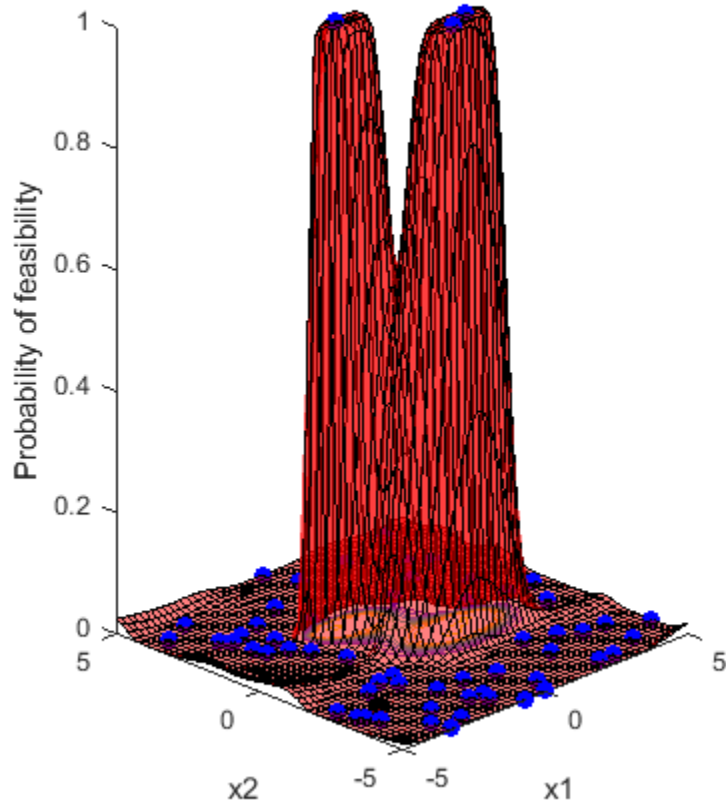
Run the optimization without any plots. For reproducibility, set the random seed and use the 'expected-improvement-plus' acquisition function. Optimize for 60 iterations so the error model becomes well-trained.

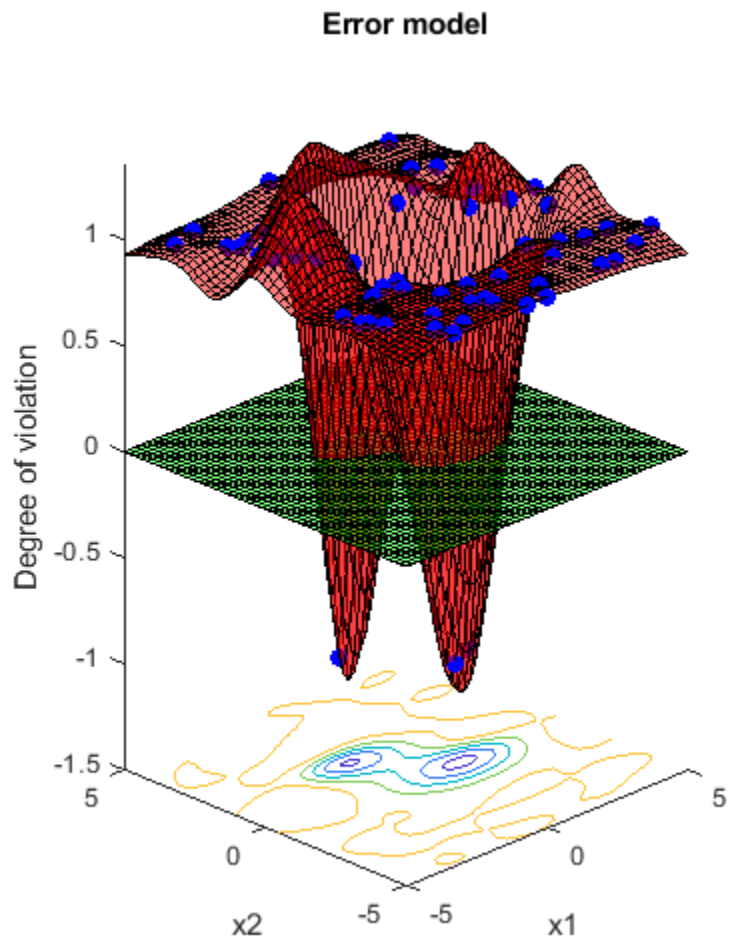
```
rng default
results = bayesopt(fun,vars,'MaxObjectiveEvaluations',60,...
    'AcquisitionFunctionName','expected-improvement-plus',...
    'PlotFcn',[],'Verbose',0);
```

Plot the error model and the best objective trace.

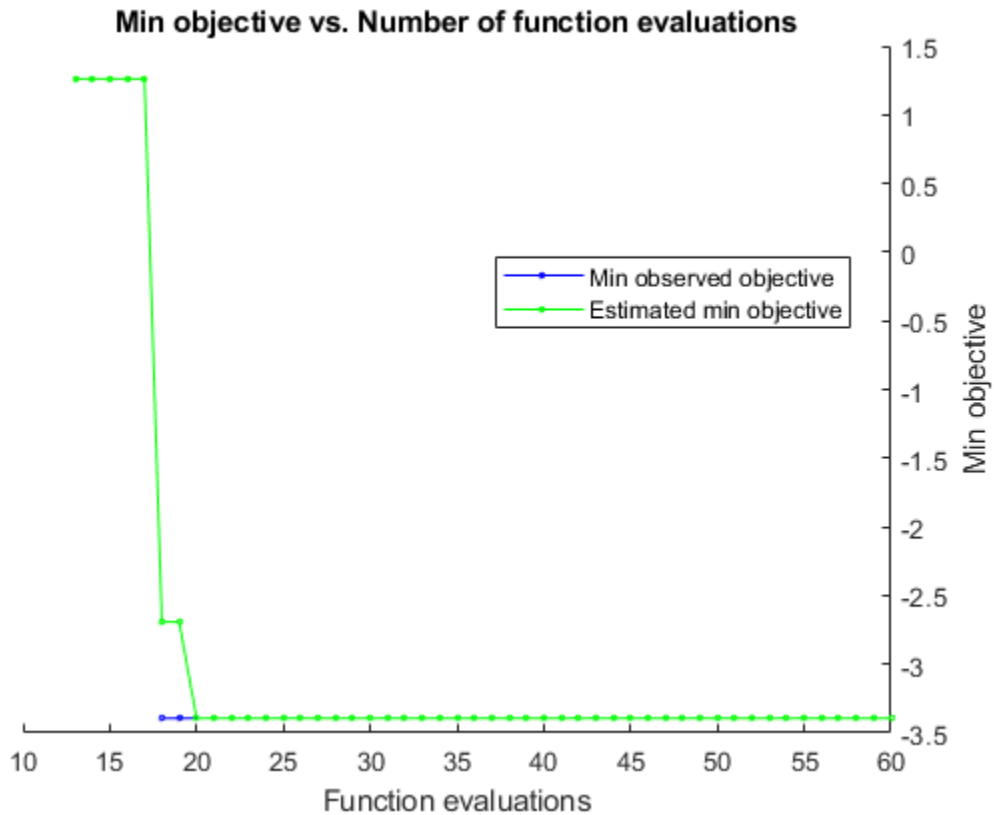
```
plot(results,@plotConstraintModels,@plotMinObjective)
```

### Probability of feasibility









## Input Arguments

### **results** — Bayesian optimization results

BayesianOptimization object

Bayesian optimization results, specified as a BayesianOptimization object.

### **plotFcn** — Plot function

function handle

Plot function, specified as a function handle.

There are several built-in plot functions:

<b>Model Plots — Apply When <math>D \leq 2</math></b>	<b>Description</b>
@plotAcquisitionFunction	Plot the acquisition function surface.
@plotConstraintModels	Plot each constraint model surface. Also plots a $P(\text{feasible})$ surface. Negative values indicate feasible points.  Also plot the error model, if it exists, which ranges from -1 to 1. Negative values mean that the model probably does not error, positive values mean that it probably does error. The model is:  Plotted error = $-2 * \text{Probability}(\text{error}) + 1$ .
@plotObjectiveEvaluationTimeModel	Plot the objective function evaluation time model surface.
@plotObjectiveModel	Plot the fun model surface, the estimated location of the minimum, and the location of the next proposed point to evaluate. For one-dimensional problems, plot envelopes one credible interval above and below the mean function, and envelopes one noise standard deviation above and below the mean.
<b>Trace Plots — Apply to All D</b>	<b>Description</b>
@plotObjective	Plot each observed function value versus the number of function evaluations.
@plotObjectiveEvaluationTime	Plot each observed function evaluation run time versus the number of function evaluations.
@plotMinObjective	Plot the minimum observed and estimated function values versus the number of function evaluations.
@plotElapsedTime	Plot three curves: the total elapsed time of the optimization, the total function evaluation time, and the total modeling and point selection time, all versus the number of function evaluations.

You can include a handle to your own plot functions. For details, see “Bayesian Optimization Plot Functions” on page 10-14.

Example: @plotObjective

Data Types: `function_handle`

## Alternative Functionality

You can specify plot functions in the `bayesopt` `PlotFcn` name-value pair. This allows you to monitor the progress of the optimization.

## See Also

`BayesianOptimization` | `bayesopt`

**Introduced in R2016b**

## plotDiagnostics

**Class:** GeneralizedLinearModel

Plot diagnostics of generalized linear regression model

### Syntax

```
plotDiagnostics mdl
plotDiagnostics mdl,plottype
h = plotDiagnostics(...)
h = plotDiagnostics mdl,plottype,Name,Value
```

### Description

`plotDiagnostics(mdl)` plots diagnostics from the `mdl` linear model using leverage as the plot type.

`plotDiagnostics(mdl,plottype)` plots diagnostics from the `mdl` generalized linear model in a plot of type `plottype`.

`h = plotDiagnostics(...)` returns handles to the lines in the plot.

`h = plotDiagnostics(mdl,plottype,Name,Value)` plots with additional options specified by one or more `Name,Value` pair arguments.

### Input Arguments

**mdl**

Generalized linear model, as constructed by `fitglm` or `stepwiseglm`.

**plottype**

Character vector specifying the type of plot:

'contour'	Residual vs. leverage with overlaid Cook's contours
'cookd'	Cook's distance
'leverage'	Leverage (diagonal of Hat matrix)

**Default:** 'leverage'

## Name-Value Pair Arguments

Specify optional comma-separated pairs of *Name*, *Value* arguments. *Name* is the argument name and *Value* is the corresponding value. *Name* must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as *Name1*, *Value1*, . . . , *NameN*, *ValueN*.

---

**Note** The plot property name-value pairs apply to the first returned handle `h(1)`.

---

### Color

Color of the line or marker, a `ColorSpec` specification. For details, see `linespec`.

### LineStyle

Type of line, a `Chart Line` specification. For details, see `linespec`.

### LineWidth

Width of the line or edges of filled area, in points, a positive scalar. One point is 1/72 inch.

**Default:** 0.5

### MarkerEdgeColor

Color of the marker or edge color for filled markers, a `ColorSpec` specification. For details, see `linespec`.

### MarkerFaceColor

Color of the marker face for filled markers, a `ColorSpec` specification. For details, see `linespec`.

**MarkerSize**

Size of the marker in points, a strictly positive scalar. One point is 1/72 inch.

## Output Arguments

**h**

Vector of handles to lines or patches in the plot.

## Examples

### Diagnostic Plots for Generalized Linear Models

Create leverage and Cook's distance plots of a fitted generalized linear model.

Generate artificial data for the model, Poisson random numbers with two underlying predictors  $X(1)$  and  $X(2)$ .

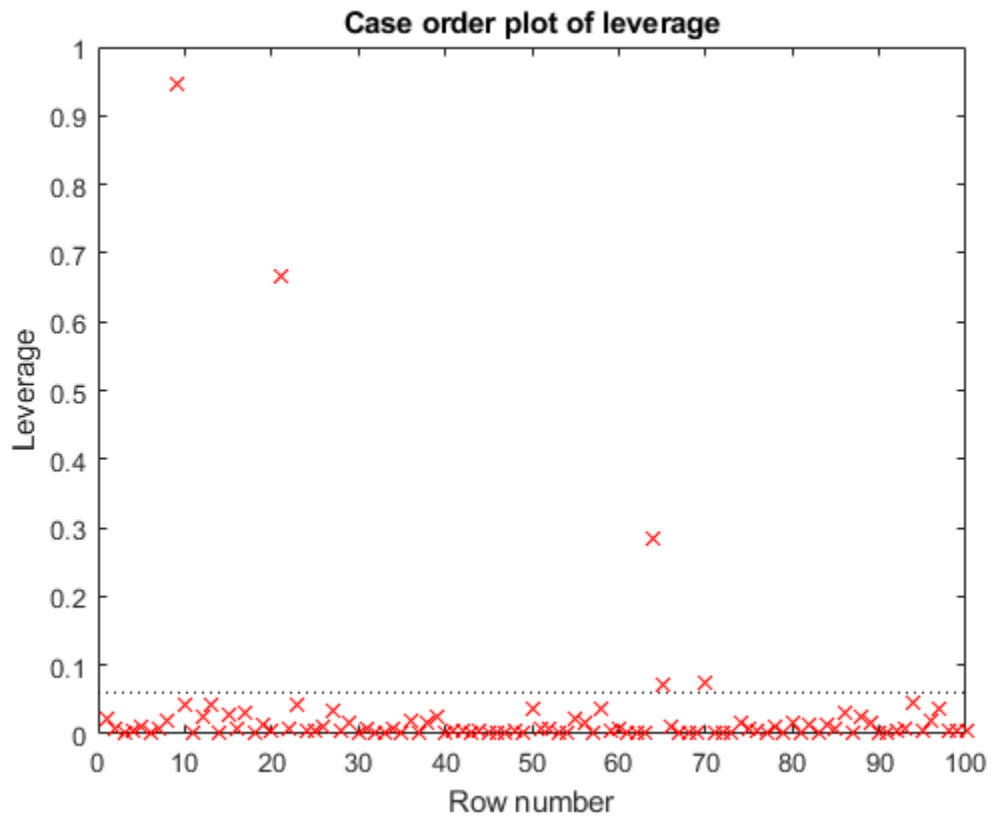
```
rng default % for reproducibility
rndvars = randn(100,2);
X = [2+rndvars(:,1), rndvars(:,2)];
mu = exp(1 + X*[1;2]);
y = poissrnd(mu);
```

Create a generalized linear regression model of Poisson data.

```
mdl = fitglm(X,y,'y ~ x1 + x2','distr','poisson');
```

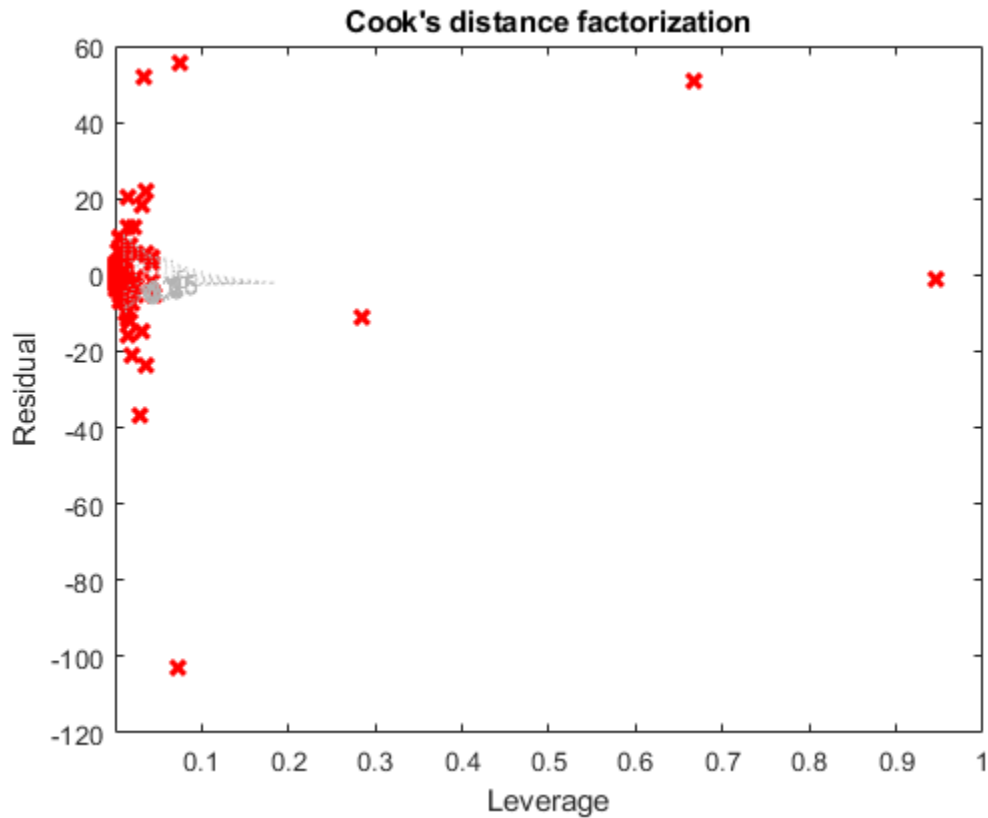
Create a leverage plot.

```
plotDiagnostics(mdl)
```



Create a contour plot with Cook's distance.

```
plotDiagnostics mdl, 'contour')
```



- “Diagnostic Plots” on page 12-25

## Definitions

### Hat Matrix

The hat matrix  $H$  is defined in terms of the data matrix  $X$  and a diagonal weight matrix  $W$ :

$$H = X(X^T W X)^{-1} X^T W^T.$$

$W$  has diagonal elements  $w_i$ :



$$w_i = \frac{g'(\mu_i)}{\sqrt{V(\mu_i)}},$$

where

- $g$  is the link function mapping  $y_i$  to  $x_i b$ .
- $g'$  is the derivative of the link function  $g$ .
- $V$  is the variance function.
- $\mu_i$  is the  $i$ th mean.

The diagonal elements  $H_{ii}$  satisfy

$$0 \leq h_{ii} \leq 1$$

$$\sum_{i=1}^n h_{ii} = p,$$

where  $n$  is the number of observations (rows of  $X$ ), and  $p$  is the number of coefficients in the regression model.

## Leverage

The leverage of observation  $i$  is the value of the  $i$ th diagonal term,  $h_{ii}$ , of the hat matrix  $H$ . Because the sum of the leverage values is  $p$  (the number of coefficients in the regression model), an observation  $i$  can be considered to be an outlier if its leverage substantially exceeds  $p/n$ , where  $n$  is the number of observations.

## Cook's Distance

The Cook's distance  $D_i$  of observation  $i$  is

$$D_i = w_i \frac{e_i^2}{p \hat{\phi} (1 - h_{ii})^2},$$

where

- $\hat{\phi}$  is the dispersion parameter (estimated or theoretical).

- $e_i$  is the linear predictor residual,  $g(y_i) - x_i \hat{\beta}$ , where
  - $g$  is the link function.
  - $y_i$  is the observed response.
  - $x_i$  is the observation.
- $\hat{\beta}$  is the estimated coefficient vector.
- $p$  is the number of coefficients in the regression model.
- $h_{ii}$  is the  $i$ th diagonal element of the Hat Matrix on page 32-2843 *H*.

## Tips

- For many plots, the Data Cursor tool in the figure window displays the  $x$  and  $y$  values for any data point, along with the observation name or number.

## References

- [1] Neter, J., M. H. Kutner, C. J. Nachtsheim, and W. Wasserman. *Applied Linear Statistical Models*, Fourth Edition. Irwin, Chicago, 1996.

## See Also

GeneralizedLinearModel

## Topics

- “Diagnostic Plots” on page 12-25  
“Generalized Linear Models” on page 12-12

# plotDiagnostics

**Class:** LinearModel

Plot diagnostics of linear regression model

## Syntax

```
plotDiagnostics mdl
plotDiagnostics mdl,plottype
h = plotDiagnostics( ___ )
h = plotDiagnostics mdl,plottype,Name,Value)
```

## Description

`plotDiagnostics(mdl)` plots diagnostics from the `mdl` linear model using the leverage values.

`plotDiagnostics(mdl,plottype)` plots diagnostics in a plot of type `plottype`.

`h = plotDiagnostics( ___ )` returns handles to the lines in the plot, using any of the previous syntaxes.

`h = plotDiagnostics(mdl,plottype,Name,Value)` plots with additional options specified by one or more `Name,Value` pair arguments.

## Input Arguments

**mdl** — Full, fitted linear regression model

LinearModel object

Full, fitted linear regression model, specified as a `LinearModel` object constructed using `fitlm` or `stepwiselm`.

**plottype** — Type of plot

'leverage' (default) | 'contour' | 'cookd' | 'covratio' | 'dfbetas' | 'dffits' | 's2\_i'

Type of plot, specified as one of the following:

'contour'	Residual vs. leverage with overlaid Cook's contours
'cookd'	Cook's distance
'covratio'	Delete-1 ratio of determinant of covariance
'dfbetas'	Scaled delete-1 coefficient estimates
'dffits'	Scaled delete-1 fitted values
'leverage'	Leverage
's2_i'	Delete-1 variance estimate

Delete-1 means compute a new model without the current observation. If the delete-1 calculation differs significantly from the model using all observations, then the observation is influential.

**Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

---

**Note** The plot property name-value pairs apply to the first returned handle `h(1)`.

---

**Color** — Color of line or marker

`ColorSpec` specification

Color of the line or marker, specified as the comma-separated pair consisting of 'Color' and a `ColorSpec` specification. For details, see `linespec`.

**LineStyle** — Line style

`Chart Line Properties` specification

Line style, specified as the comma-separated pair consisting of 'LineStyle' and a Chart Line specification. For details, see `linespec`.

**LineWidth** — Width of line or edges

0.5 (default) | positive numeric value

Width of the line or edges of filled area, in points, specified as the comma-separated pair consisting of 'LineWidth' and a positive numeric value. One point is equal to 1/72 inch.

**MarkerEdgeColor** — Color of marker or edge

ColorSpec specification

Color of the marker or edge color for filled markers, specified as the comma-separated pair consisting of 'MarkerEdgeColor' and a ColorSpec specification. For details, see `linespec`.

**MarkerFaceColor** — Color of marker face

ColorSpec specification

Color of the marker face for filled markers, specified as the comma-separated pair consisting of 'MarkerFaceColor' and a ColorSpec specification. For details, see `linespec`.

**MarkerSize** — Size of marker

positive numeric value

Size of the marker in points, specified as the comma-separated pair consisting of 'MarkerSize' and a positive numeric value. One point is 1/72 inch.

## Output Arguments

**h** — Graphics handles

vector of graphics handle objects

Graphics handles, returned as a vector of graphics handles corresponding to the lines or patches in the plot.

## Examples

### Leverage Plot of Linear Model

Plot the leverage values of observations in a fitted model.

Load the `carsmall` data and fit a linear model of the mileage as a function of model year, weight, and weight squared.

```
load carsmall
tbl = table(MPG,Weight);
tbl.Year = categorical(Model_Year);
mdl = fitlm(tbl,'MPG ~ Year + Weight^2');
```

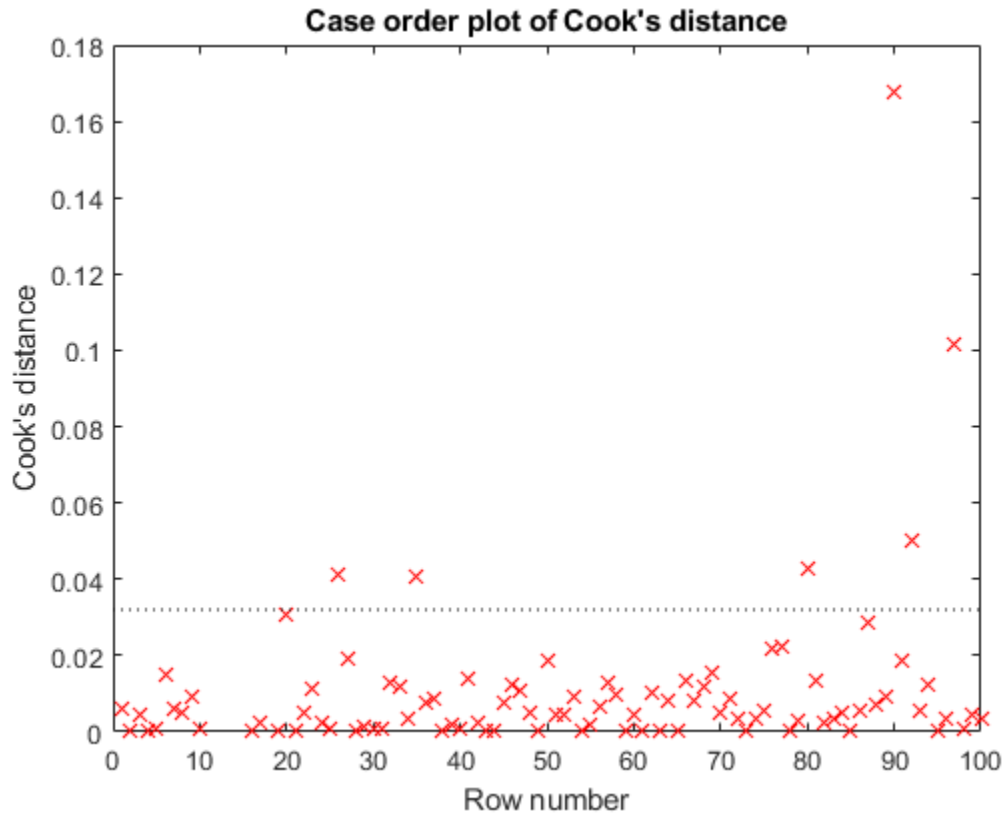
Plot the leverage values.

```
plotDiagnostics(mdl)
```



Plot the Cook's distance.

```
plotDiagnostics mdl, 'cookd')
```



The two diagnostic plots give different results.

- “Diagnostic Plots” on page 11-22

## Definitions

### Hat Matrix

The hat matrix  $H$  is defined in terms of the data matrix  $X$ :

$$H = X(X^T X)^{-1} X^T.$$

The diagonal elements  $h_{ii}$  satisfy

$$0 \leq h_{ii} \leq 1$$

$$\sum_{i=1}^n h_{ii} = p,$$

where  $n$  is the number of observations (rows of  $X$ ), and  $p$  is the number of coefficients in the regression model.

### Leverage

The leverage of observation  $i$  is the value of the  $i$ th diagonal term,  $h_{ii}$ , of the hat matrix  $H$ . Because the sum of the leverage values is  $p$  (the number of coefficients in the regression model), an observation  $i$  can be considered to be an outlier if its leverage substantially exceeds  $p/n$ , where  $n$  is the number of observations.

### Cook's Distance

Cook's distance is the scaled change in fitted values. Each element in `CooksDistance` is the normalized change in the vector of coefficients due to the deletion of an observation. The Cook's distance,  $D_i$ , of observation  $i$  is

$$D_i = \frac{\sum_{j=1}^n (\hat{y}_j - \hat{y}_{j(i)})^2}{p \text{MSE}},$$

where

- $\hat{y}_j$  is the  $j$ th fitted response value.



- $\hat{y}_{j(i)}$  is the  $j$ th fitted response value, where the fit does not include observation  $i$ .
- $MSE$  is the mean squared error.
- $p$  is the number of coefficients in the regression model.

Cook's distance is algebraically equivalent to the following expression:

$$D_i = \frac{r_i^2}{p MSE} \left( \frac{h_{ii}}{(1-h_{ii})^2} \right),$$

where  $r_i$  is the  $i$ th residual, and  $h_{ii}$  is the  $i$ th leverage value.

`CooksDistance` is an  $n$ -by-1 column vector in the `Diagnostics` table of the `LinearModel` object.

## Tips

- For many plots, the Data Cursor tool in the figure window displays the  $x$  and  $y$  values for any data point, along with the observation name or number.

## Alternatives

The `mdl.Diagnostics` property contains the information that `plotDiagnostics` uses to create plots.

## References

- [1] Neter, J., M. H. Kutner, C. J. Nachtsheim, and W. Wasserman. *Applied Linear Statistical Models*, Fourth Edition. Irwin, Chicago, 1996.

## See Also

`LinearModel`

## **Topics**

“Diagnostic Plots” on page 11-22

“Linear Regression” on page 11-11

# plotDiagnostics

**Class:** NonLinearModel

Plot diagnostics of nonlinear regression model

## Syntax

```
plotDiagnostics mdl
plotDiagnostics mdl,plottype
h = plotDiagnostics(...)
h = plotDiagnostics mdl,plottype,Name,Value)
```

## Description

`plotDiagnostics(mdl)` plots diagnostics from the `mdl` linear model using leverage as the plot type.

`plotDiagnostics(mdl,plottype)` plots diagnostics in a plot of type `plottype`.

`h = plotDiagnostics(...)` returns handles to the lines in the plot.

`h = plotDiagnostics(mdl,plottype,Name,Value)` plots with additional options specified by one or more `Name,Value` pair arguments.

## Input Arguments

### **mdl**

Nonlinear regression model, constructed by `fitnlm`.

### **plottype**

Character vector specifying the type of plot:

'contour'	Residual vs. leverage with overlaid Cook's contours
-----------	---

'cookd'	Cook's distance
'leverage'	Leverage (diagonal of Hat matrix)

**Default:** 'leverage'

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

---

**Note** The plot property name-value pairs apply to the first returned handle `h(1)`.

---

### **Color**

Color of the line or marker, a `ColorSpec` specification. For details, see `linespec`.

### **LineStyle**

Type of line, a `Chart Line` specification. For details, see `linespec`.

### **LineWidth**

Width of the line or edges of filled area, in points, a positive scalar. One point is 1/72 inch.

**Default:** 0.5

### **MarkerEdgeColor**

Color of the marker or edge color for filled markers, a `ColorSpec` specification. For details, see `linespec`.

### **MarkerFaceColor**

Color of the marker face for filled markers, a `ColorSpec` specification. For details, see `linespec`.

**MarkerSize**

Size of the marker in points, a strictly positive scalar. One point is 1/72 inch.

## Output Arguments

**h**

Vector of handles to lines or patches in the plot.

## Examples

**Nonlinear Model Leverage Plot**

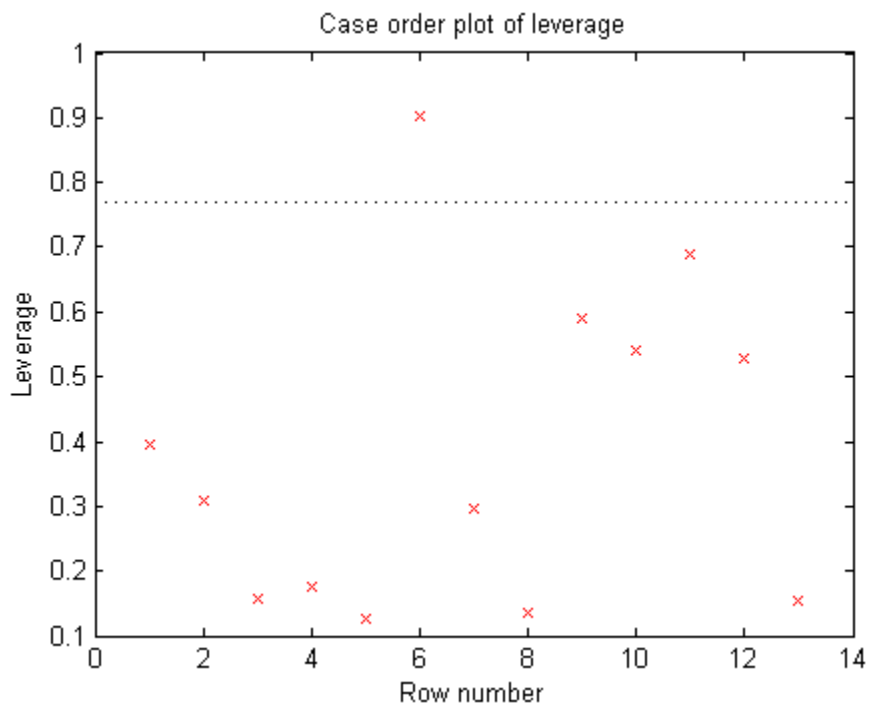
Create a leverage plot of a fitted nonlinear model, and find the points with high leverage.

Load the reaction data and fit a model of the reaction rate as a function of reactants.

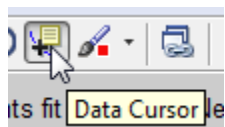
```
load reaction
mdl = fitnlm(reactants,rate,@hougen,[1 .05 .02 .1 2]);
```

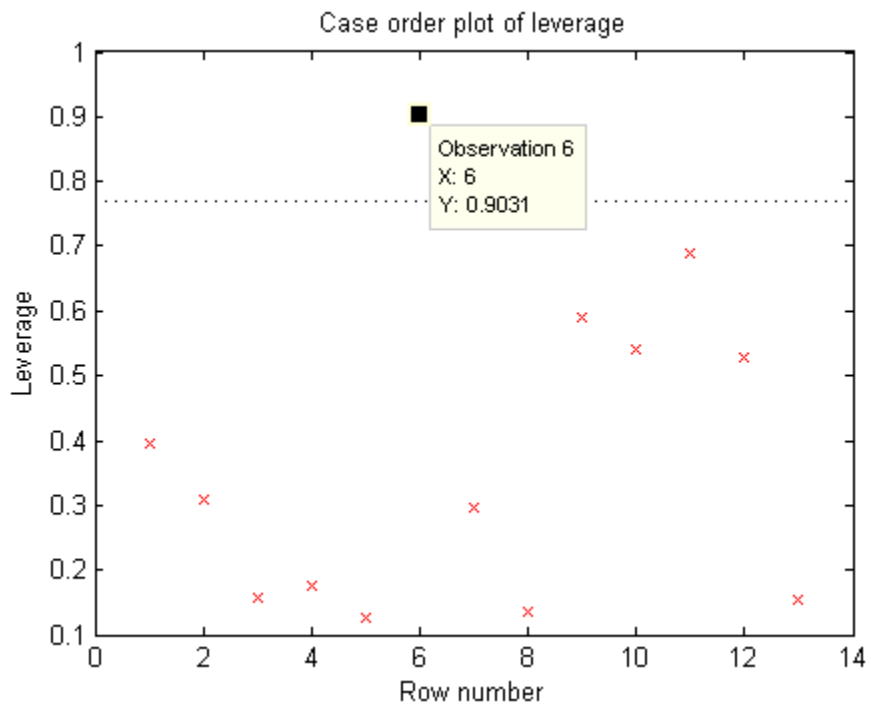
Create a leverage plot of the fitted model.

```
plotDiagnostics(mdl)
```



To examine the observation with high leverage, activate the Data Cursor and click the observation.





Alternatively, find the high-leverage observation at the command line.

```
find mdl.Diagnostics.Leverage > 0.8)
```

```
ans =
```

```
6
```

- “Examine Quality and Adjust the Fitted Nonlinear Model” on page 13-7
- “Nonlinear Regression Workflow” on page 13-15

## Definitions

### Hat Matrix

The hat matrix  $H$  is defined in terms of the data matrix  $X$  and the Jacobian matrix  $J$ :

$$J_{i,j} = \left. \frac{\partial f}{\partial \beta_j} \right|_{x_i, \beta}$$

Here  $f$  is the nonlinear model function, and  $\beta$  is the vector of model coefficients.

The Hat Matrix  $H$  is

$$H = J(J^T J)^{-1} J^T.$$

The diagonal elements  $H_{ii}$  satisfy

$$0 \leq h_{ii} \leq 1$$

$$\sum_{i=1}^n h_{ii} = p,$$

where  $n$  is the number of observations (rows of  $X$ ), and  $p$  is the number of coefficients in the regression model.

## Leverage

The leverage of observation  $i$  is the value of the  $i$ th diagonal term,  $h_{ii}$ , of the hat matrix  $H$ . Because the sum of the leverage values is  $p$  (the number of coefficients in the regression model), an observation  $i$  can be considered to be an outlier if its leverage substantially exceeds  $p/n$ , where  $n$  is the number of observations.

## Cook's Distance

The Cook's distance  $D_i$  of observation  $i$  is

$$D_i = \frac{\sum_{j=1}^n (\hat{y}_j - \hat{y}_{j(i)})^2}{p \text{MSE}},$$

where

- $\hat{y}_j$  is the  $j$ th fitted response value.
- $\hat{y}_{j(i)}$  is the  $j$ th fitted response value, where the fit does not include observation  $i$ .



- $MSE$  is the mean squared error.
- $p$  is the number of coefficients in the regression model.

Cook's distance is algebraically equivalent to the following expression:

$$D_i = \frac{r_i^2}{p MSE} \left( \frac{h_{ii}}{(1-h_{ii})^2} \right),$$

where  $e_i$  is the  $i$ th residual.

## Tips

- For many plots, the Data Cursor tool in the figure window displays the  $x$  and  $y$  values for any data point, along with the observation name or number.

## References

- [1] Neter, J., M. H. Kutner, C. J. Nachtsheim, and W. Wasserman. *Applied Linear Statistical Models*, Fourth Edition. Irwin, Chicago, 1996.

## See Also

`NonLinearModel` | `plotResiduals`

## Topics

- “Examine Quality and Adjust the Fitted Nonlinear Model” on page 13-7
- “Nonlinear Regression Workflow” on page 13-15
- “Nonlinear Regression” on page 13-2

## plotEffects

**Class:** CompactLinearModel

Plot main effects of each predictor in linear regression model

### Syntax

```
plotEffects mdl  
h = plotEffects mdl
```

### Description

`plotEffects(mdl)` produces an effects plot for the predictors in the `mdl` regression model. The plot shows the estimated effect on the response from changing each predictor value, averaging out the effects of the other predictors. `plotEffects` chooses values to produce a relatively large effect on the response.

`h = plotEffects(mdl)` returns handles to the lines in the plot.

### Input Arguments

**mdl** — Linear model object

LinearModel object | CompactLinearModel object

Linear model object, specified as a full `LinearModel` object constructed using `fitlm` or `stepwiselm`, or a compacted `CompactLinearModel` object constructed using `compact`.

### Output Arguments

**h** — Graphics handles

vector of graphics handles

Graphics handles to lines or patches in plot, returned as a vector of graphics handles.  $h(1)$  is a handle to the circles that represent the main effect estimates.  $h(j+1)$  is a handle to the line that defines the confidence interval for the effect of predictor  $j$ .

## Examples

### Effects Plot for Fitted Linear Regression Model

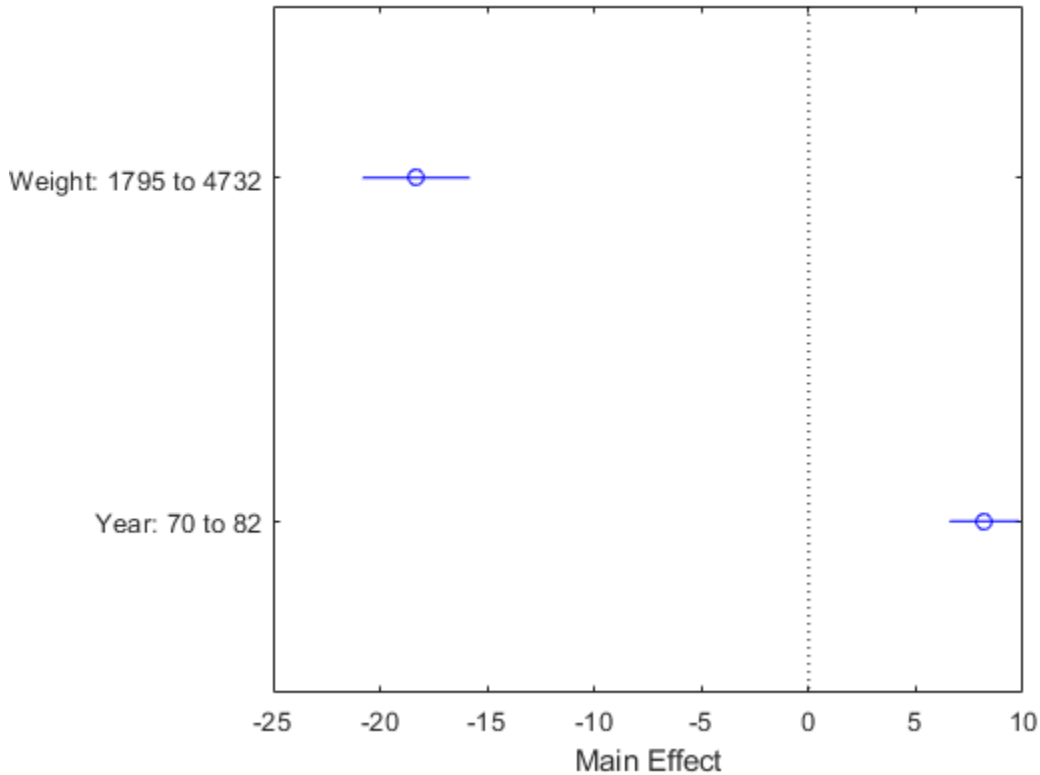
Plot the effects of two predictors in a fitted linear model.

Load the `carsmall` data and fit a linear model of the mileage as a function of model year, weight, and weight squared.

```
load carsmall
tbl = table(MPG,Weight);
tbl.Year = ordinal(Model_Year);
mdl = fitlm(tbl,'MPG ~ Year + Weight^2');
```

Create an effects plot.

```
plotEffects(mdl)
```



The width of each horizontal line in the figure shows a confidence interval for the effect on the response of the listed change in each predictor. The estimated effect of changing Year from 70 to 82 is an increase of about 8, and is between 6 and 10 with 95% confidence.

- “Plots to Understand Predictor Effects” on page 11-29

## Tips

- For many plots, the Data Cursor tool in the figure window displays the  $x$  and  $y$  values for any data point, along with the observation name or number.

## Alternatives

Use `plotInteraction` for an effects plot of the interactions of two specified variables.

## See Also

`CompactLinearModel` | `LinearModel` | `plotAdjustedResponse` | `plotInteraction`

## Topics

“Plots to Understand Predictor Effects” on page 11-29

“Linear Regression” on page 11-11

## plotInteraction

**Class:** CompactLinearModel

Plot interaction effects of two predictors in linear regression model

### Syntax

```
plotInteraction mdl, var1, var2
plotInteraction mdl, var1, var2, ptype
h = plotInteraction( ___ )
```

### Description

`plotInteraction(mdl, var1, var2)` creates a plot of the interaction effects of the predictors `var1` and `var2` in `mdl`. The plot shows the estimated effect on the response from changing each predictor value, averaging out the effects of the other predictors. The plot also shows the estimated effect with the other predictor fixed at certain values. `plotInteraction` chooses values to produce a relatively large effect on the response. The plot lets you examine whether the effect of one predictor depends on the value of the other predictor.

`plotInteraction(mdl, var1, var2, ptype)` returns a plot of the type specified in `ptype`.

`h = plotInteraction( ___ )` returns handles to the lines in the plot, using any of the previous syntaxes.

### Input Arguments

**mdl** — Linear model object

LinearModel object | CompactLinearModel object

Linear model object, specified as a full `LinearModel` object constructed using `fitlm` or `stepwiselm`, or a compacted `CompactLinearModel` object constructed using `compact`.

**var1 — First variable for plot**

variable name

First variable for plot, specified as a variable name. `plotInteraction` chooses values of `var1` to create relatively large changes in the response. If you set `pType` equal to `'predictions'`, then the plot shows curves as a function of `var2` with various fixed values of `var1`.

**var2 — Second variable for plot**

variable name

Second variable for plot, specified as a variable name. `plotInteraction` chooses values of `var2` to create relatively large changes in the response. If you set `pType` equal to `'predictions'`, then the plot shows curves as a function of `var2` with various fixed values of `var1`.

**pType — Plot type**`'effects' (default) | 'predictions'`

Plot type, specified as one of the following.

- `'effects'` — The plot shows each effect as a circle, with a horizontal bar showing the confidence interval for the estimated effect. `plotInteraction` computes the effect values from the adjusted response curve, as shown by the `plotAdjustedResponse` function.
- `'predictions'` — The plot shows the adjusted response curve as a function of `var2`, with `var1` fixed at certain values.

## Output Arguments

**h — Graphics handles**

vector of graphics handle objects

Graphics handles, returned as a vector of graphics handles corresponding to the lines or patches in the plot.

## Examples

### Interaction Plot for Linear Regression Model

Create a model of car mileage as a function of weight and model year. Then create a plot to see if the predictors have interactions.

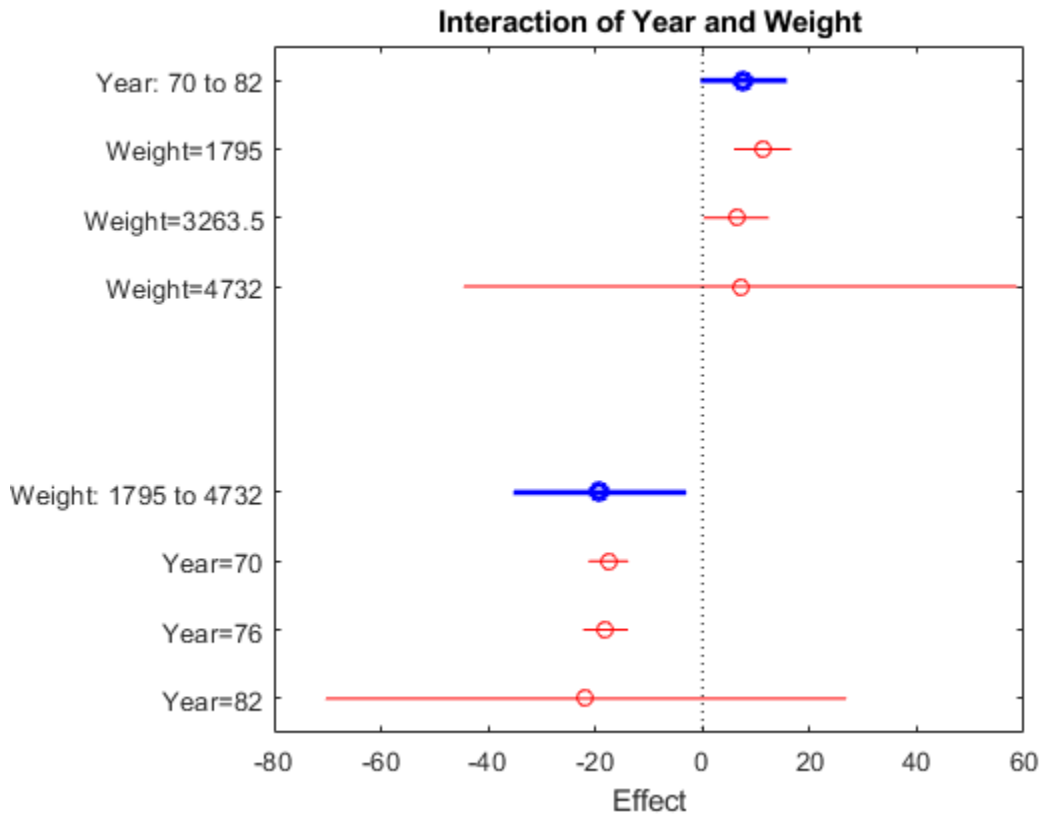
Create a linear model of mileage from the `carsmall` data.

```
load carsmall
tbl = table(MPG,Weight);
tbl.Year = ordinal(Model_Year);
var1 = 'Year';
var2 = 'Weight';
mdl = fitlm(tbl, 'MPG ~ Year * Weight^2');
```

Create an interaction plot.

```
plotInteraction(mdl,var1,var2)
```





The plot might show an interaction, because the groups of points are not perfectly vertical. But the error bars seem large enough that a vertical line could pass within all of the confidence intervals for each group, possibly indicating no interaction.

### Prediction Curve Interaction Plot for Linear Model

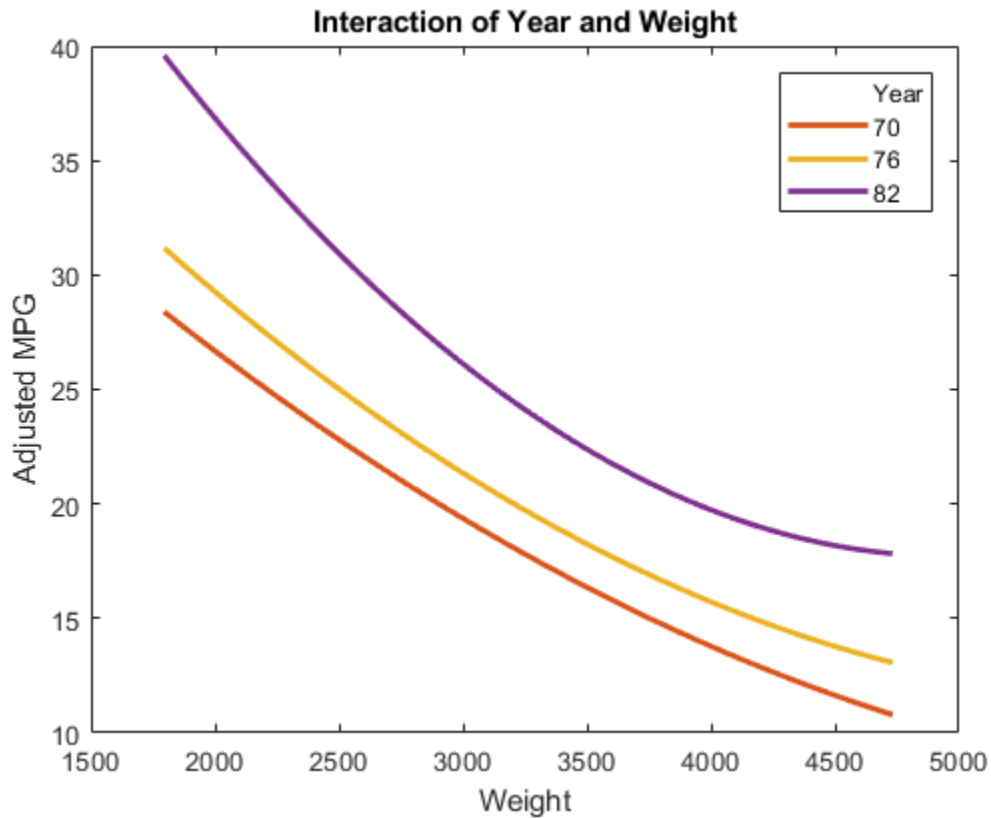
Create a model of car mileage as a function of weight and model year. Then create an interaction curve plot to see if the predictors have interactions.

Create a linear model of mileage from the `carsmall` data.

```
load carsmall
tbl = table(MPG,Weight);
tbl.Year = ordinal(Model_Year);
var1 = 'Year';
var2 = 'Weight';
mdl = fitlm(tbl,'MPG ~ Year * Weight^2');
```

Create an interaction plot with type 'predictions'.

```
plotInteraction(mdl,var1,var2,'predictions')
```



The curves are not parallel. This indicates interactions between the predictors. The effect is subtle enough not to definitively indicate a interaction.

- “Plots to Understand Predictor Effects” on page 11-29

## Tips

- For many plots, the Data Cursor tool in the figure window displays the  $x$  and  $y$  values for any data point, along with the observation name or number.

## Alternatives

Use `plotEffects` for an effects plot showing separate effects for all predictors.

## See Also

`CompactLinearModel` | `LinearModel` | `plotAdjustedResponse` | `plotEffects`

## Topics

“Plots to Understand Predictor Effects” on page 11-29

“Linear Regression” on page 11-11

## plotprofile

**Class:** RepeatedMeasuresModel

Plot expected marginal means with optional grouping

## Syntax

```
plotprofile(rm, X)
plotprofile(rm, Name, Value)
H = plotprofile(____)
```

## Description

`plotprofile(rm, X)` plots the expected marginal means computed from the repeated measures model `rm` as a function of the variable `X`.

`plotprofile(rm, Name, Value)` plots the expected marginal means computed from the repeated measures model `rm` with additional options specified by one or more `Name, Value` pair arguments.

For example, you can specify the factors to group by or change the line colors.

`H = plotprofile(____)` returns handles, `H`, to the plotted lines.

## Input Arguments

**rm** — Repeated measures model

RepeatedMeasuresModel object

Repeated measures model, returned as a RepeatedMeasuresModel object.

For properties and methods of this object, see RepeatedMeasuresModel.

**x** — Name of between-subjects or within-subjects factor

character vector

Name of a between-subjects or within-subjects factor, specified as a character vector.

For example, if you want to plot the marginal means as a function of the groups of a between-subjects variable `drug`, you can specify it as follows.

Example: `'Drug'`

Data Types: `char`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

### **Group** — Name of between-subject factor or factors

character vector | cell array of character vectors

Name of between-subject factor or factors, specified as the comma-separated pair consisting of `'Group'` and a character vector or cell array of character vectors. This name-value pair argument groups the lines according to the factor values.

For example, if you have two between-subject factors, `drug` and `sex`, and you want to group the lines in the plot according to them, you can specify these factors as follows.

Example: `'Group', {'Drug', 'Sex'}`

Data Types: `char` | `cell`

### **Marker** — Marker to use for each group

cell array of character vectors

Marker to use for each group, specified as the comma-separated pair consisting of `'Marker'` and a cell array of character vectors.

For example, if you have two between-subject factors, `drug` and `sex`, with each having two groups, you can specify `o` as the marker for the groups of `drug` and `x` as the marker for the groups of `sex` as follows.

Example: `'Marker', {'o', 'o', 'x', 'x'}`

Data Types: `cell`

**Color** — Color for each group

character vector | cell array of character vectors | rows of a three-column RGB matrix

Color for each group, specified as the comma-separated pair consisting of 'Color' and a character vector, cell array of character vectors, or rows of a three-column RGB matrix.

For example, if you have two between-subject factors, drug and sex, with each having two groups, you can specify red as the color for the groups of drug and blue as the color for the groups of sex as follows.

Example: 'Color', 'rbbb'

Data Types: single | double | cell

**LineStyle** — Line style for each group

cell array of character vectors

Line style for each group, specified as the comma-separated pair consisting of 'LineStyle' and a cell array of character vectors.

For example, if you have two between-subject factors, drug and sex, with each having two groups, you can specify - as the line style of one group and : as the line style for the other group as follows.

Example: 'LineStyle', {'-' ':' '-' ':'}

Data Types: cell

## Output Arguments

**h** — Handle to plotted lines

handle

Handle to plotted lines, returned as a handle.

## Examples

**Plot Expected Marginal Means**

Load the sample data.

```
load fisheriris
```

The column vector `species` consists of iris flowers of three different species: `setosa`, `versicolor`, and `virginica`. The double matrix `meas` consists of four types of measurements on the flowers: the length and width of sepals and petals in centimeters, respectively.

Store the data in a table array.

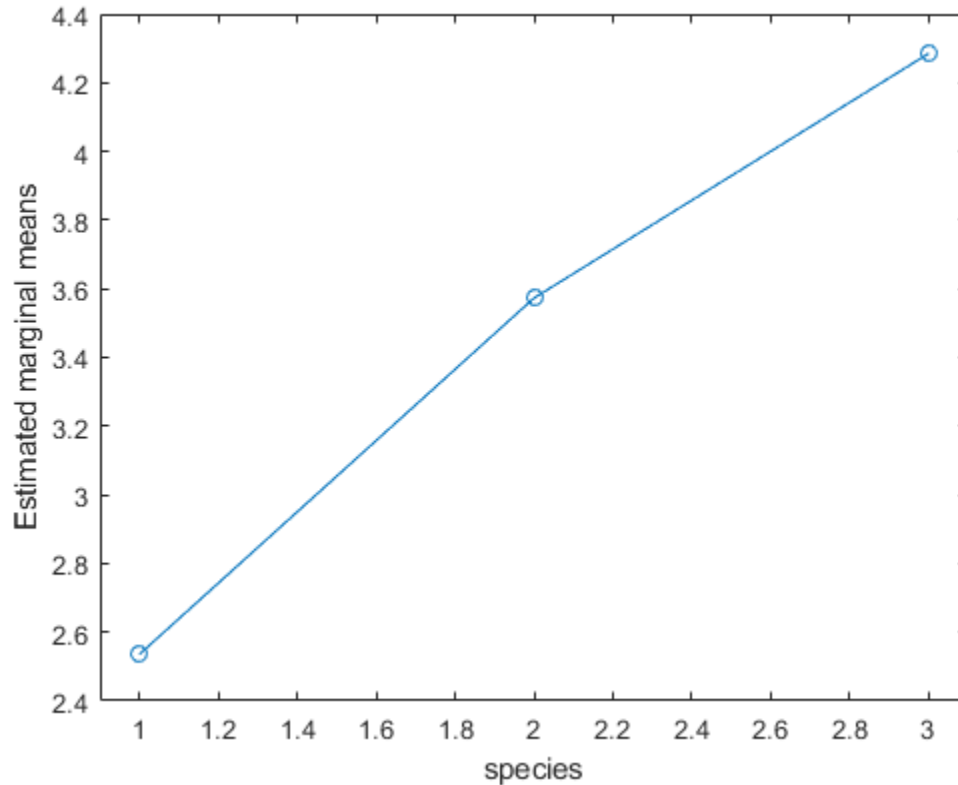
```
t = table(species,meas(:,1),meas(:,2),meas(:,3),meas(:,4),...  
'VariableNames',{'species','meas1','meas2','meas3','meas4'});  
Meas = dataset([1 2 3 4]','VarNames',{'Measurements'});
```

Fit a repeated measures model, where the measurements are the responses and the `species` is the predictor variable.

```
rm = fitrm(t,'meas1-meas4~species','WithinDesign',Meas);
```

Perform data grouped by the factor `species`.

```
plotprofile(rm,'species')
```



The estimated marginal means seem to differ with group. You can compute the standard error and the 95% confidence intervals for the marginal means using the `margmean` method.

### Plot Marginal Means for Two Groups

Load the sample data.

```
load repeatedmeas
```



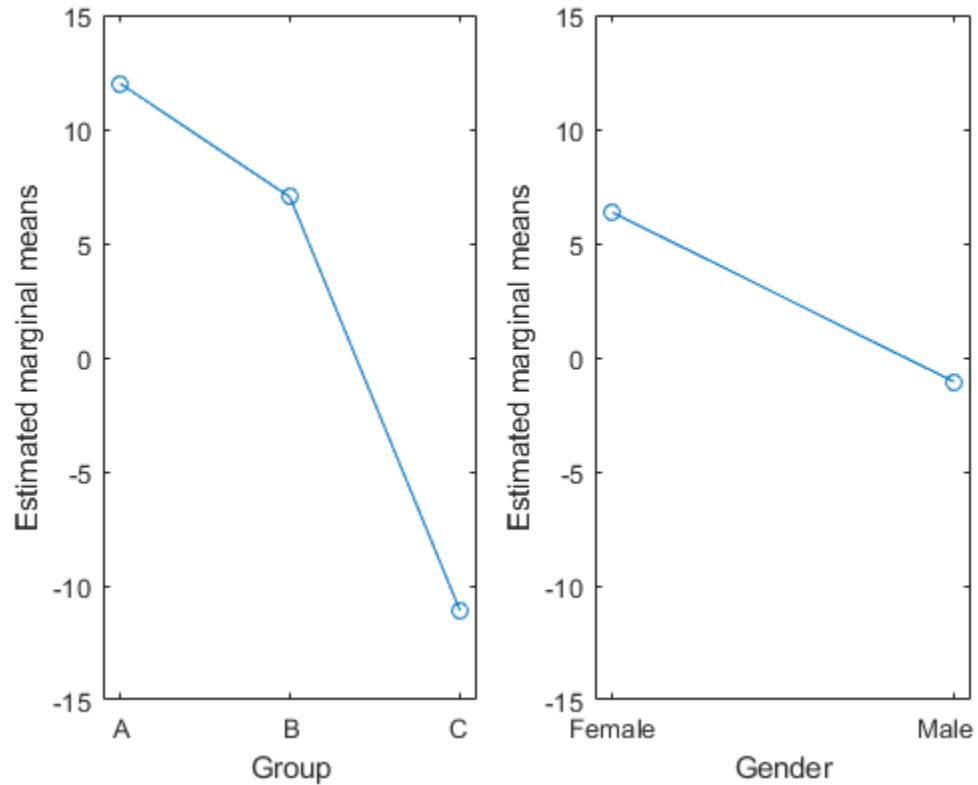
The table `between` includes the between-subject variables `age`, `IQ`, `group`, `gender`, and eight repeated measures `y1` through `y8` as responses. The table `within` includes the within-subject variables `w1` and `w2`. This is simulated data.

Fit a repeated measures model, where the repeated measures `y1` through `y8` are the responses, and `age`, `IQ`, `group`, `gender`, and the group-gender interaction are the predictor variables. Also specify the within-subject design matrix.

```
rm = fitrm(between, 'y1-y8 ~ Group*Gender + Age + IQ', 'WithinDesign', within);
```

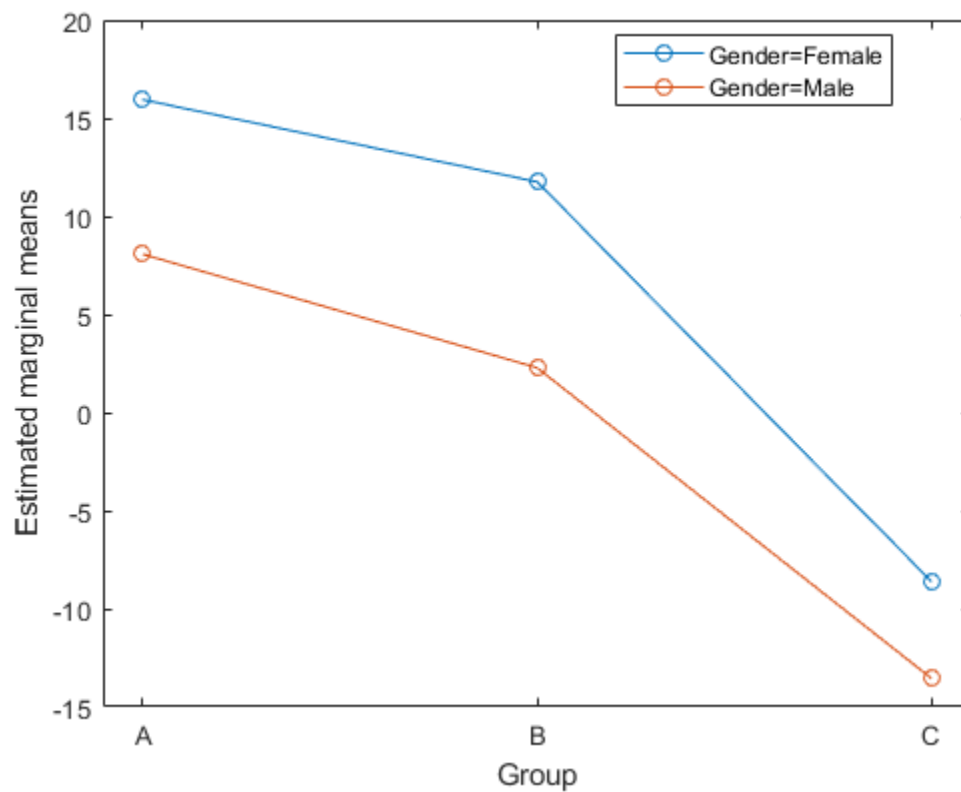
Plot the estimated marginal means based on the factors `Group` and `Gender`.

```
ax1 = subplot(1,2,1);  
plotprofile(rm, 'Group')  
ax2 = subplot(1,2,2);  
plotprofile(rm, 'Gender')  
linkaxes([ax1 ax2], 'y')
```



Plot the estimated marginal means based on the factor Group and grouped by Gender.

```
figure()  
plotprofile(m, 'Group', 'Group', 'Gender')
```



## See Also

`fitrm` | `margmean` | `plot`

## plotResiduals

**Class:** GeneralizedLinearModel

Plot residuals of generalized linear regression model

### Syntax

```
plotResiduals mdl
plotResiduals mdl, plottype
h = plotResiduals (...)
h = plotResiduals mdl, plottype, Name, Value
```

### Description

`plotResiduals(mdl)` gives a histogram plot of the residuals of the `mdl` nonlinear model.

`plotResiduals(mdl, plottype)` plots residuals in a plot of type `plottype`.

`h = plotResiduals(...)` returns handles to the lines in the plot.

`h = plotResiduals(mdl, plottype, Name, Value)` plots with additional options specified by one or more `Name, Value` pair arguments.

### Input Arguments

**mdl**

Generalized linear model, as constructed by `fitglm` or `stepwiseglm`.

**plottype**

Character vector specifying the type of plot:

'caseorder'	Residuals vs. case (row) order
-------------	--------------------------------

'fitted'	Residuals vs. fitted values
'histogram'	Histogram
'lagged'	Residuals vs. lagged residual ( $r(t)$ vs. $r(t-1)$ )
'probability'	Normal probability plot
'symmetry'	Symmetry plot

**Default:** 'histogram'

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

---

**Note** The plot property name-value pairs apply to the first returned handle `h(1)`.

---

### Color

Color of the line or marker, a `ColorSpec` specification. For details, see `linespec`.

### LineStyle

Type of line, a `Chart Line` specification. For details, see `linespec`.

### LineWidth

Width of the line or edges of filled area, in points, a positive scalar. One point is 1/72 inch.

**Default:** 0.5

### MarkerEdgeColor

Color of the marker or edge color for filled markers, a `ColorSpec` specification. For details, see `linespec`.

**MarkerFaceColor**

Color of the marker face for filled markers, a `ColorSpec` specification. For details, see `linespec`.

**MarkerSize**

Size of the marker in points, a strictly positive scalar. One point is 1/72 inch.

**ResidualType**

Type of residual used in the plot.

'Raw'	Observed minus fitted values
'LinearPredictor'	Residuals on the linear predictor scale, equal to the adjusted response value minus the fitted linear combination of the predictors
'Pearson'	Raw residuals divided by RMSE
'Anscombe'	Residuals defined on transformed data with the transformation chosen to remove skewness
'Deviance'	Residuals based on the contribution of each observation to the deviance

**Default:** 'Raw'

## Output Arguments

**h**

Vector of handles to lines or patches in the plot.

## Examples

### Residual Plots for Generalized Linear Models

Create residual plots of a fitted generalized linear model.

Generate artificial data for the model, Poisson random numbers with two underlying predictors  $X(1)$  and  $X(2)$ .

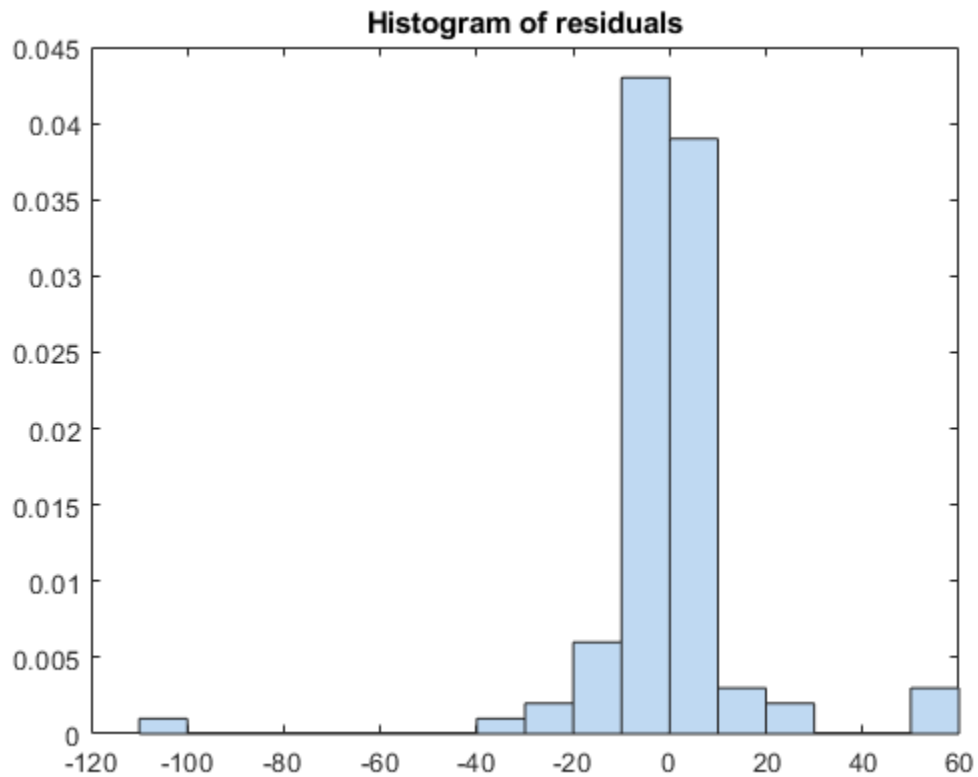
```
rng('default') % for reproducibility
rndvars = randn(100,2);
X = [2+rndvars(:,1),rndvars(:,2)];
mu = exp(1 + X*[1;2]);
y = poissrnd(mu);
```

Create a generalized linear regression model of Poisson data.

```
mdl = fitglm(X,y,'y ~ x1 + x2','distr','poisson');
```

Create a default residuals plot.

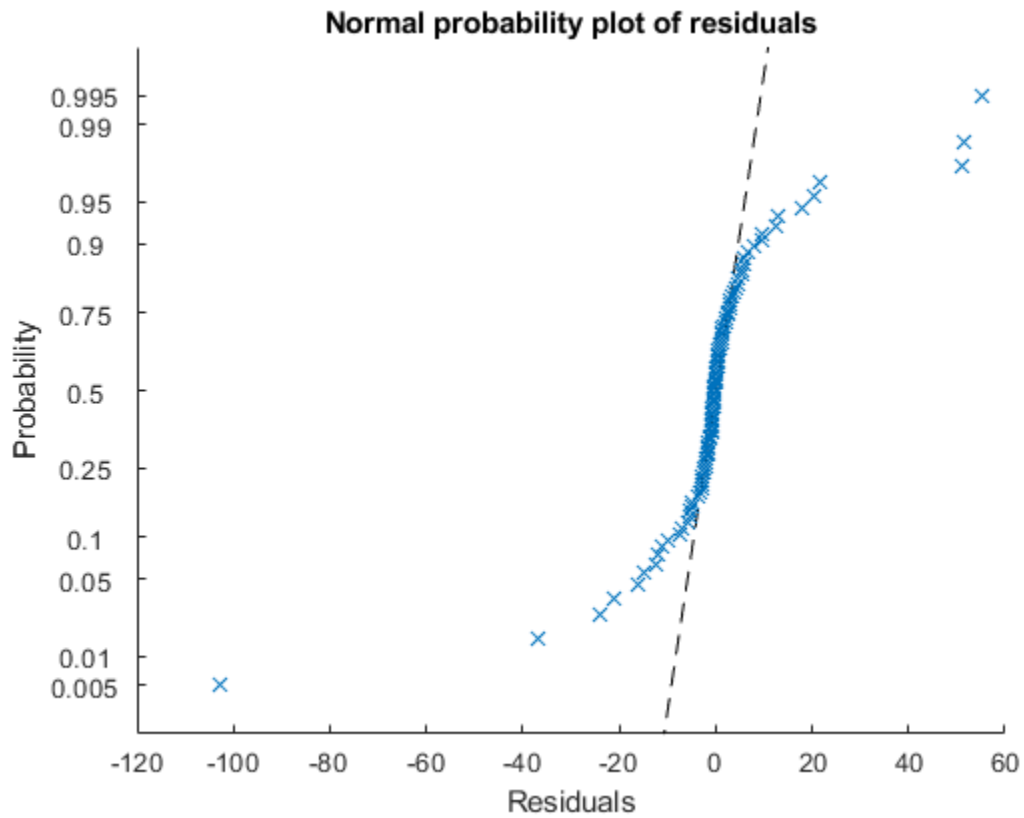
```
plotResiduals(mdl)
```



Create a probability plot. The residuals do not match a normal distribution in the tails because they are more spread out.

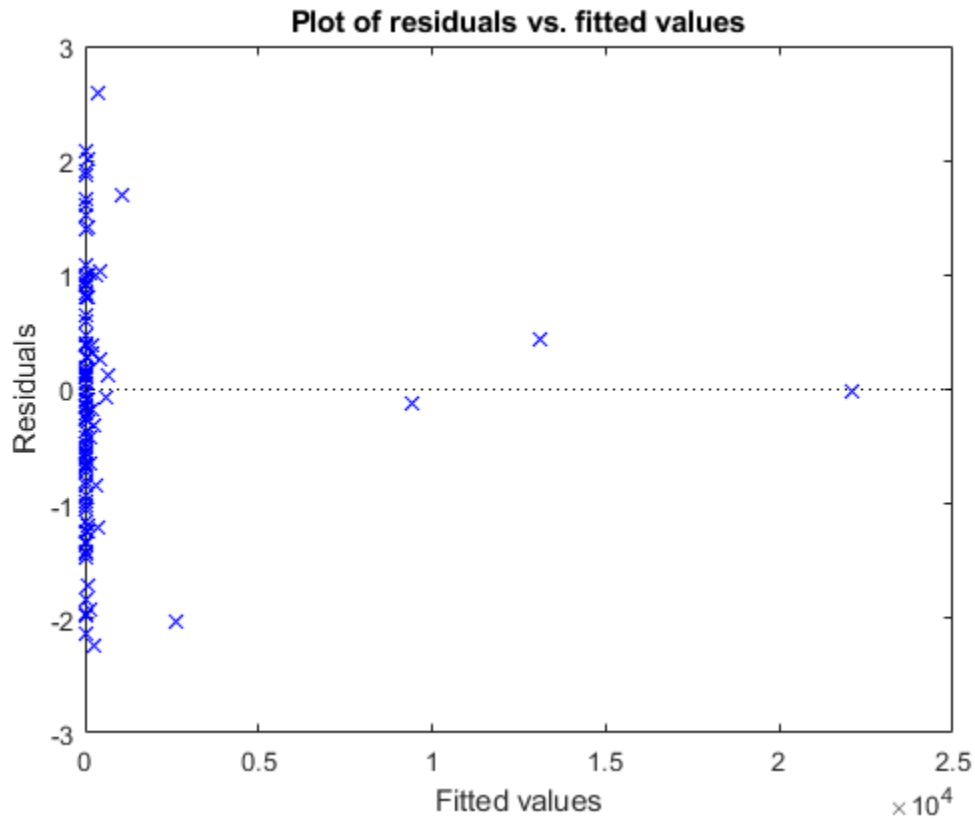
```
plotResiduals(mdl, 'probability')
```





Create a plot of the fitted residuals of Anscombe type.

```
plotResiduals(mdl, 'fitted', 'ResidualType', 'Anscombe')
```



- “Residuals — Model Quality for Training Data” on page 12-28

## Definitions

### Deviance

Deviance is twice the log likelihood of the model. Because this overall log likelihood is a sum of log likelihoods for each observation, the residual plot of deviance type shows the log likelihood per observation.

## Tips

- For many plots, the Data Cursor tool in the figure window displays the  $x$  and  $y$  values for any data point, along with the observation name or number.

## See Also

`GeneralizedLinearModel`

## Topics

“Residuals — Model Quality for Training Data” on page 12-28

“Generalized Linear Models” on page 12-12

## plotResiduals

**Class:** GeneralizedLinearMixedModel

Plot residuals of generalized linear mixed-effects model

### Syntax

```
plotResiduals(glme,plottype)
plotResiduals(glme,plottype,Name,Value)

h = plotResiduals( ___ )
```

### Description

`plotResiduals(glme,plottype)` plots the raw conditional residuals of the generalized linear mixed-effects model `glme` in a plot of the type specified by `plottype`.

`plotResiduals(glme,plottype,Name,Value)` plots the conditional residuals of `glme` using additional options specified by one or more `Name,Value` pair arguments. For example, you can specify to plot the Pearson residuals.

`h = plotResiduals( ___ )` returns a handle, `h`, to the lines or patches in the plot of residuals.

### Input Arguments

**glme** — Generalized linear mixed-effects model

GeneralizedLinearMixedModel object

Generalized linear mixed-effects model, specified as a GeneralizedLinearMixedModel object. For properties and methods of this object, see GeneralizedLinearMixedModel.

**plottype** — Type of residual plot

'histogram' (default) | 'caseorder' | 'fitted' | 'lagged' | 'probability' | 'symmetry'

Type of residual plot, specified as one of the following.

Value	Description
'histogram'	Histogram of residuals
'caseorder'	Residuals versus case order. Case order is the same as the row order used in the input data <code>tbl</code> when fitting the model using <code>fitglm</code> .
'fitted'	Residuals versus fitted values
'lagged'	Residuals versus lagged residual ( $r(t)$ versus $r(t - 1)$ )
'probability'	Normal probability plot
'symmetry'	Symmetry plot

Example: `plotResiduals(glm, 'lagged')`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### ResidualType — Residual type

'raw' (default) | 'Pearson'

Residual type, specified by the comma-separated pair consisting of `ResidualType` and one of the following.

Residual Type	Formula
'raw'	$r_{ci} = y_i - g^{-1}(x_i^T \hat{\beta} + z_i^T \hat{b} + \delta_i)$
'Pearson'	$r_{ci}^{pearson} = \frac{r_{ci}}{\sqrt{\frac{\hat{\sigma}^2}{w_i} v_i(\mu_i(\hat{\beta}, \hat{b}))}}$

In each of these equations:

- $y_i$  is the  $i$ th element of the  $n$ -by-1 response vector,  $y$ , where  $i = 1, \dots, n$ .
- $g^{-1}$  is the inverse link function for the model.
- $x_i^T$  is the  $i$ th row of the fixed-effects design matrix  $X$ .
- $z_i^T$  is the  $i$ th row of the random-effects design matrix  $Z$ .
- $\delta_i$  is the  $i$ th offset value.
- $\sigma^2$  is the dispersion parameter.
- $w_i$  is the  $i$ th observation weight.
- $v_i$  is the variance term for the  $i$ th observation.
- $\mu_i$  is the mean of the response for the  $i$ th observation.
- $\hat{\beta}$  and  $\hat{b}$  are estimated values of  $\beta$  and  $b$ .

Raw residuals from a generalized linear mixed-effects model have nonconstant variance. Pearson residuals are expected to have an approximately constant variance, and are generally used for analysis.

Example: `'ResidualType', 'Pearson'`

## Output Arguments

### **h** — Handle to residual plot

graphics object

Handle to the residual plot, returned as a graphics object. You can use dot notation to change certain property values of the object, including face color for a histogram, and marker style and color for a scatterplot. For more information, see “Access Property Values” (MATLAB).

## Examples

### Create Plots of Residuals

Load the sample data.

```
load mfr
```

This simulated data is from a manufacturing company that operates 50 factories across the world, with each factory running a batch process to create a finished product. The company wants to decrease the number of defects in each batch, so it developed a new manufacturing process. To test the effectiveness of the new process, the company selected 20 of its factories at random to participate in an experiment: Ten factories implemented the new process, while the other ten continued to run the old process. In each of the 20 factories, the company ran five batches (for a total of 100 batches) and recorded the following data:

- Flag to indicate whether the batch used the new process (`newprocess`)
- Processing time for each batch, in hours (`time`)
- Temperature of the batch, in degrees Celsius (`temp`)
- Categorical variable indicating the supplier (A, B, or C) of the chemical used in the batch (`supplier`)
- Number of defects in the batch (`defects`)

The data also includes `time_dev` and `temp_dev`, which represent the absolute deviation of time and temperature, respectively, from the process standard of 3 hours at 20 degrees Celsius.

Fit a generalized linear mixed-effects model using `newprocess`, `time_dev`, `temp_dev`, and `supplier` as fixed-effects predictors. Include a random-effects term for intercept grouped by `factory`, to account for quality differences that might exist due to factory-specific variations. The response variable `defects` has a Poisson distribution, and the appropriate link function for this model is `log`. Use the Laplace fit method to estimate the coefficients. Specify the dummy variable encoding as `'effects'`, so the dummy variable coefficients sum to 0.

The number of defects can be modeled using a Poisson distribution:

$$\text{defects}_{ij} \sim \text{Poisson}(\mu_{ij})$$

This corresponds to the generalized linear mixed-effects model

$$\log(\mu_{ij}) = \beta_0 + \beta_1 \text{newprocess}_{ij} + \beta_2 \text{time.dev}_{ij} + \beta_3 \text{temp.dev}_{ij} + \beta_4 \text{supplier.C}_{ij} + \beta_5 \text{supplier.B}_{ij} + b_i,$$

where

- `defectsij` is the number of defects observed in the batch produced by factory  $i$  during batch  $j$ .

- $\mu_{ij}$  is the mean number of defects corresponding to factory  $i$  (where  $i = 1, 2, \dots, 20$ ) during batch  $j$  (where  $j = 1, 2, \dots, 5$ ).
- `newprocessij`, `time_devij`, and `temp_devij` are the measurements for each variable that correspond to factory  $i$  during batch  $j$ . For example, `newprocessij` indicates whether the batch produced by factory  $i$  during batch  $j$  used the new process.
- `supplier_Cij` and `supplier_Bij` are dummy variables that use effects (sum-to-zero) coding to indicate whether company C or B, respectively, supplied the process chemicals for the batch produced by factory  $i$  during batch  $j$ .
- $b_i \sim N(0, \sigma_b^2)$  is a random-effects intercept for each factory  $i$  that accounts for factory-specific variation in quality.

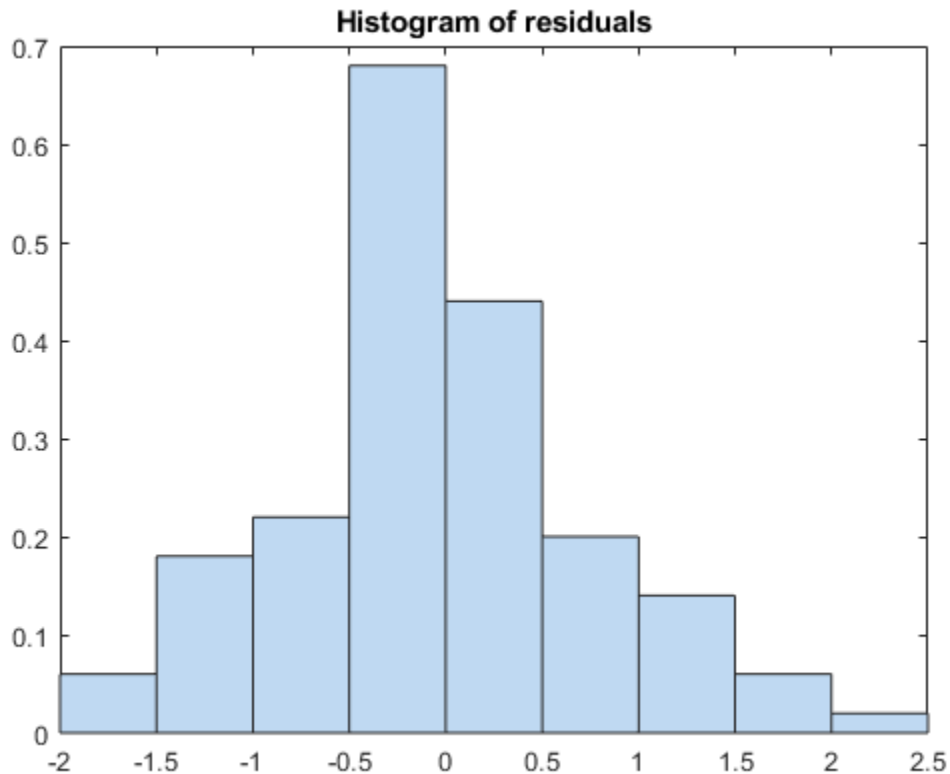
```
glme = fitglme(mfr, 'defects ~ 1 + newprocess + time_dev + temp_dev + supplier + (1|fact
```

Create diagnostic plots using Pearson residuals to test the model assumptions.

Plot a histogram to visually confirm that the mean of the Pearson residuals is equal to 0. If the model is correct, we expect the Pearson residuals to be centered at 0.

```
plotResiduals(glme, 'histogram', 'ResidualType', 'Pearson')
```

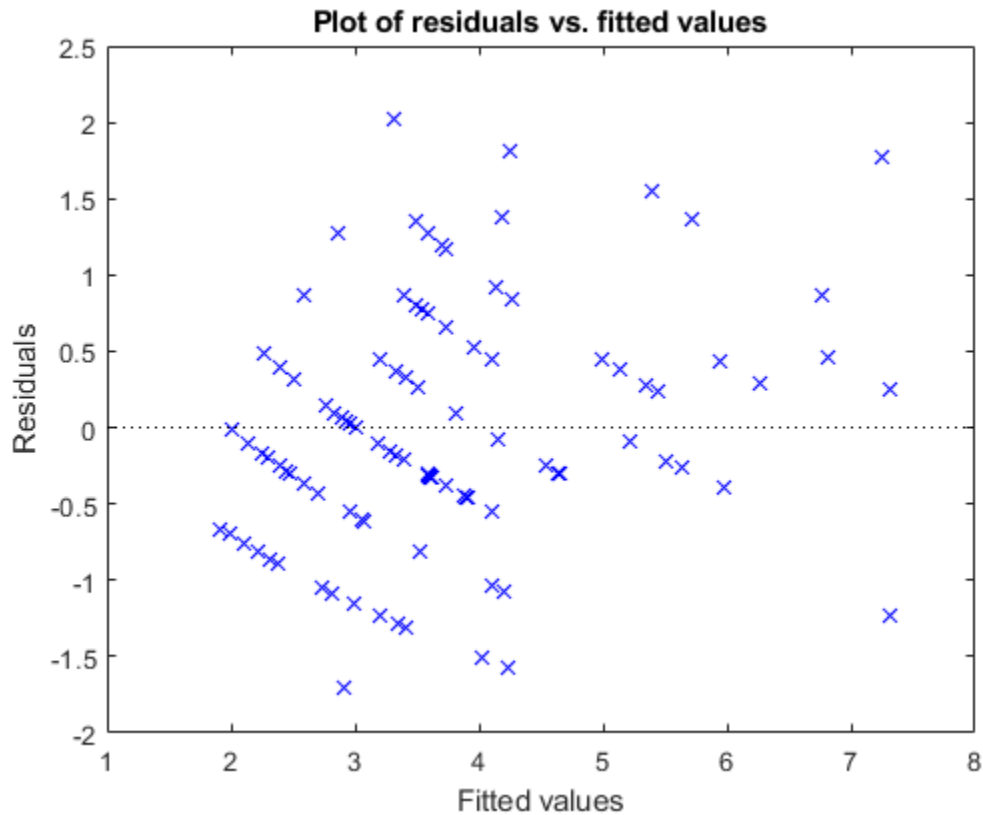




The histogram shows that the Pearson residuals are centered at 0.

Plot the Pearson residuals versus the fitted values, to check for signs of nonconstant variance among the residuals (heteroscedasticity). We expect the conditional Pearson residuals to have a constant variance. Therefore, a plot of conditional Pearson residuals versus conditional fitted values should not reveal any systematic dependence on the conditional fitted values.

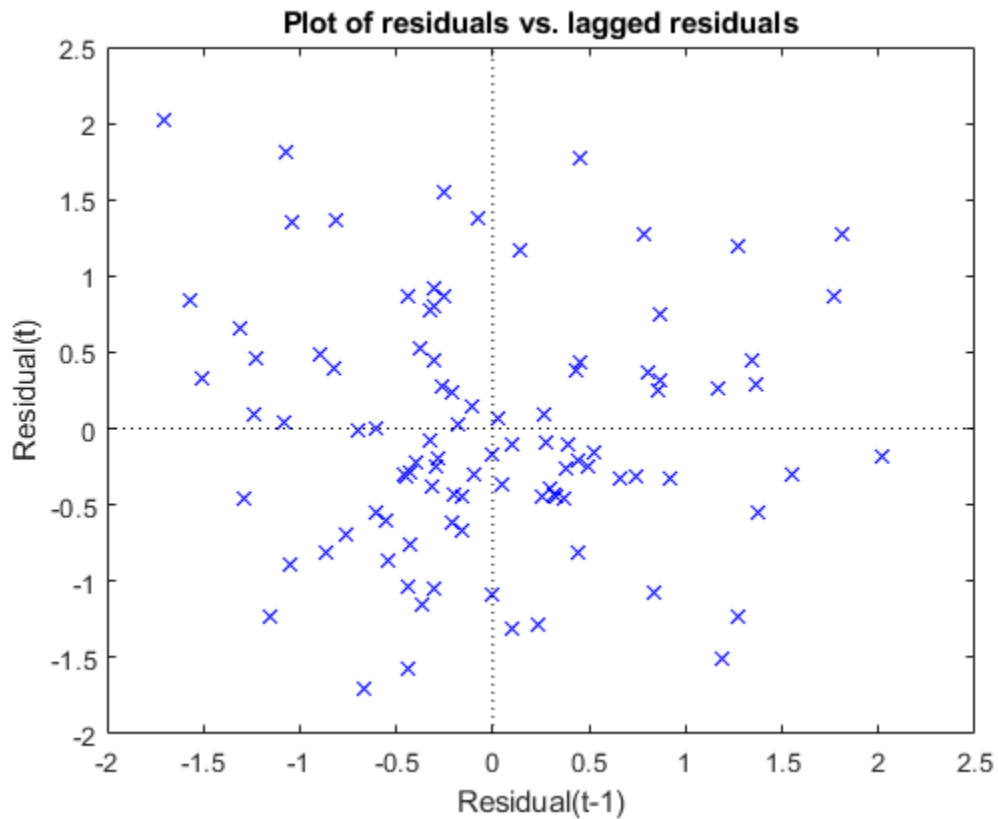
```
plotResiduals(glme, 'fitted', 'ResidualType', 'Pearson')
```



The plot does not show a systematic dependence on the fitted values, so there are no signs of nonconstant variance among the residuals.

Plot the Pearson residuals versus lagged residuals, to check for correlation among the residuals. The conditional independence assumption in GLME implies that the conditional Pearson residuals are approximately uncorrelated.

```
plotResiduals(glme, 'lagged', 'ResidualType', 'Pearson')
```



There is no pattern to the plot, so there are no signs of correlation among the residuals.

## See Also

`GeneralizedLinearMixedModel` | `fitglm` | `fitted` | `plot` | `residuals`

## plotResiduals

**Class:** LinearModel

Plot residuals of linear regression model

### Syntax

```
plotResiduals mdl
plotResiduals mdl, plottype
h = plotResiduals( ___ )
h = plotResiduals( mdl, plottype, Name, Value)
```

### Description

`plotResiduals( mdl )` gives a histogram plot of the residuals of the `mdl` linear model.

`plotResiduals( mdl, plottype )` plots residuals in a plot of type `plottype`.

`h = plotResiduals( ___ )` returns handles to the lines in the plot, using any of the previous syntaxes.

`h = plotResiduals( mdl, plottype, Name, Value )` plots with additional options specified by one or more `Name, Value` pair arguments.

### Input Arguments

**mdl** — Full, fitted linear regression model

LinearModel object

Full, fitted linear regression model, specified as a LinearModel object constructed using `fitlm` or `stepwiselm`.

**plottype** — Plot type

'histogram' (default) | 'caseorder' | 'fitted' | 'lagged' | 'probability' | 'symmetry'

Plot type, specified as one of the following:

'caseorder'	Residuals vs. case (row) order
'fitted'	Residuals vs. fitted values
'histogram'	Histogram
'lagged'	Residuals vs. lagged residual ( $r(t)$ vs. $r(t-1)$ )
'probability'	Normal probability plot
'symmetry'	Symmetry plot

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

---

**Note** The plot property name-value pairs apply to the first returned handle `h(1)`.

---

### **Color** — Color of line or marker

`ColorSpec` specification

Color of the line or marker, specified as the comma-separated pair consisting of 'Color' and a `ColorSpec` specification. For details, see `linespec`.

### **LineStyle** — Line style

`Chart Line Properties` specification

Line style, specified as the comma-separated pair consisting of 'LineStyle' and a `Chart Line` specification. For details, see `linespec`.

### **LineWidth** — Width of line or edges

0.5 (default) | positive numeric value

Width of the line or edges of filled area, in points, specified as the comma-separated pair consisting of 'LineWidth' and a positive numeric value. One point is equal to 1/72 inch.

### **MarkerEdgeColor** — Color of marker or edge

`ColorSpec` specification

Color of the marker or edge color for filled markers, specified as the comma-separated pair consisting of 'MarkerEdgeColor' and a ColorSpec specification. For details, see linespec.

**MarkerFaceColor** — Color of marker face

ColorSpec specification

Color of the marker face for filled markers, specified as the comma-separated pair consisting of 'MarkerFaceColor' and a ColorSpec specification. For details, see linespec.

**MarkerSize** — Size of marker

positive numeric value

Size of the marker in points, specified as the comma-separated pair consisting of 'MarkerSize' and a positive numeric value. One point is 1/72 inch.

**ResidualType** — Residual type

'raw' (default) | 'pearson' | 'standardized' | 'studentized'

Type of residual used in the plot, specified as the comma-separated pair consisting of 'ResidualType' and one of the following:

'raw'	Observed minus fitted values
'pearson'	Raw residuals divided by RMSE
'standardized'	Raw residuals divided by their estimated standard deviation
'studentized'	Raw residuals divided by an independent (delete-1) estimate of their standard deviation

## Output Arguments

**h** — Graphics handles

vector of graphics handle objects

Graphics handles, returned as a vector of graphics handles corresponding to the lines or patches in the plot.

## Examples

### Histogram of Linear Model Residuals

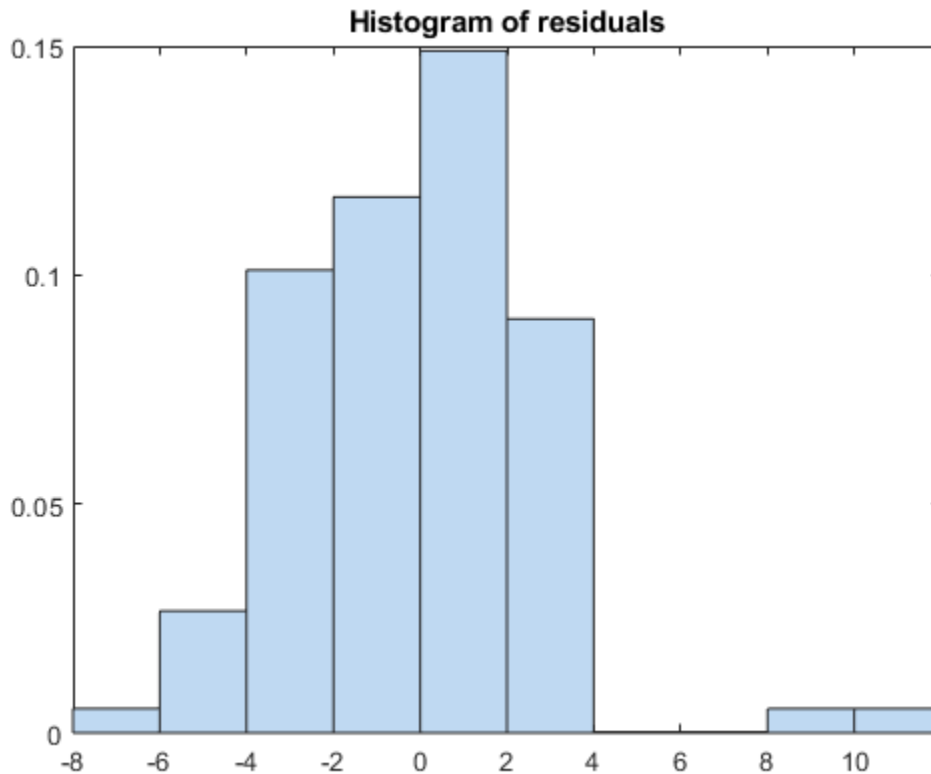
Plot a histogram of the residuals of a fitted linear model.

Load the `carsmall` data and fit a linear model of the mileage as a function of model year, weight, and weight squared.

```
load carsmall
tbl = table(MPG,Weight);
tbl.Year = ordinal(Model_Year);
mdl = fitlm(tbl,'MPG ~ Year + Weight^2');
```

Plot the raw residuals.

```
plotResiduals(mdl)
```



### Normal Probability Plot of Linear Model Residuals

Create a normal probability plot of the residuals of a fitted linear model.

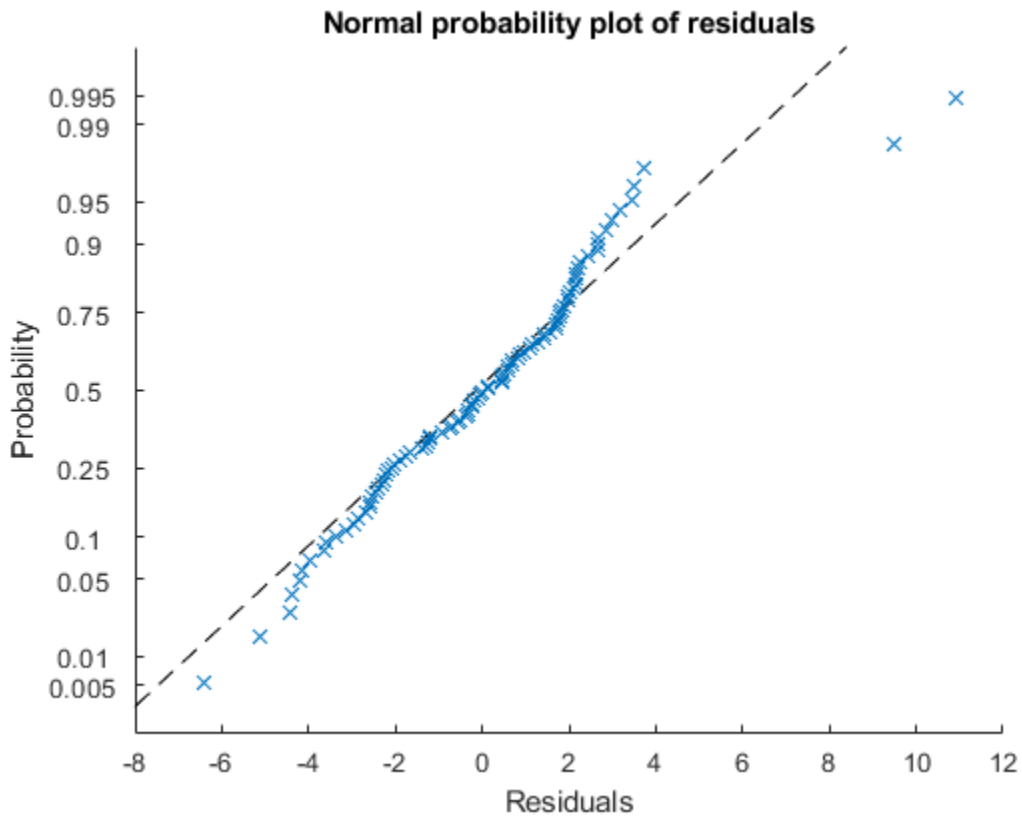
Load the `carsmall` data and fit a linear model of the mileage as a function of model year, weight, and weight squared.

```
load carsmall
X = [Weight,Model_Year];
mdl = fitlm(X,MPG,...
    'y ~ x2 + x1^2','Categorical',2);
```

Create a normal probability plot of the residuals of the fitted model.



```
plotResiduals(mdl, 'probability')
```



- “Residuals — Model Quality for Training Data” on page 11-24
- “Linear Regression Workflow” on page 11-43
- “Compare large and small stepwise models” on page 11-127

## Tips

- For many plots, the Data Cursor tool in the figure window displays the  $x$  and  $y$  values for any data point, along with the observation name or number.

## Alternatives

The `mdl.Residuals` table contains the information in residual plots.

## See Also

`LinearModel` | `plotDiagnostics`

## Topics

“Residuals — Model Quality for Training Data” on page 11-24

“Linear Regression Workflow” on page 11-43

“Compare large and small stepwise models” on page 11-127

“Robust Regression versus Standard Least-Squares Fit” on page 11-133

# plotResiduals

**Class:** LinearMixedModel

Plot residuals of linear mixed-effects model

## Syntax

```
plotResiduals(lme, plottype)
plotResiduals(lme, plottype, Name, Value)
```

```
h = plotResiduals( ___ )
```

## Description

`plotResiduals(lme, plottype)` plots the raw conditional residuals of the linear mixed-effects model `lme` in a plot of the type specified by `plottype`.

`plotResiduals(lme, plottype, Name, Value)` also plots the residuals of the linear mixed-effects model `lme` with additional options specified by one or more name-value pair arguments. For example, you can specify the residual type to plot.

`plotResiduals` also accepts some other name-value pair arguments that specify the properties of the primary line in the plot. For those name-value pairs, see `plot`.

`h = plotResiduals( ___ )` returns a handle, `h`, to the lines or patches in the plot of residuals.

## Input Arguments

**lme** — Linear mixed-effects model

LinearMixedModel object

Linear mixed-effects model, specified as a LinearMixedModel object constructed using `fitlme` or `fitlmematrix`.

**plottype** — Type of residual plot

'histogram' (default) | 'caseorder' | 'fitted' | 'lagged' | 'probability' | 'symmetry'

Type of residual plot, specified as one of the following.

'histogram'	Default. Histogram of residuals
'caseorder'	Residuals versus case (row) order
'fitted'	Residuals versus fitted values
'lagged'	Residuals versus lagged residual ( $r(t)$ versus $r(t - 1)$ )
'probability'	Normal probability plot
'symmetry'	Symmetry plot

Example: `plotResiduals(lme, 'lagged')`

**Name-Value Pair Arguments**

Specify optional comma-separated pairs of Name, Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as Name1, Value1, ..., NameN, ValueN.

**ResidualType** — Residual type

'Raw' (default) | 'Pearson' | 'Standardized'

Residual type, specified by the comma-separated pair consisting of ResidualType and one of the following.

Residual Type	Conditional	Marginal
'Raw'	$r_i^C = [y - X\hat{\beta} - Z\hat{b}]_i$	$r_i^M = [y - X\hat{\beta}]_{\cdot i}$
'Pearson'	$pr_i^C = \frac{r_i^C}{\sqrt{[\widehat{\text{Var}}_{y,b}(y - X\hat{\beta} - Zb)]_{ii}}}$	$pr_i^M = \frac{r_i^M}{\sqrt{[\widehat{\text{Var}}_y(y - X\hat{\beta})]_{ii}}}$

Residual Type	Conditional	Marginal
'Standardized'	$st_i^C = \frac{r_i^C}{\sqrt{[\widehat{Var}_y(r^C)]_{ii}}}$	$st_i^M = \frac{r_i^M}{\sqrt{[\widehat{Var}_y(r^M)]_{ii}}}$

For more information on the conditional and marginal residuals and residual variances, see [Definitions](#) at the end of this page.

Example: `'ResidualType', 'Standardized'`

## Output Arguments

**h** — Handle to residual plot

handle

Handle to the residual plot, returned as a handle.

## Examples

### Examine Residuals

Load the sample data.

```
load(fullfile(matlabroot, 'examples', 'stats', 'weight.mat'));
```

`weight` contains data from a longitudinal study, where 20 subjects are randomly assigned to 4 exercise programs, and their weight loss is recorded over six 2-week time periods. This is simulated data.

Store the data in a table. Define `Subject` and `Program` as categorical variables.

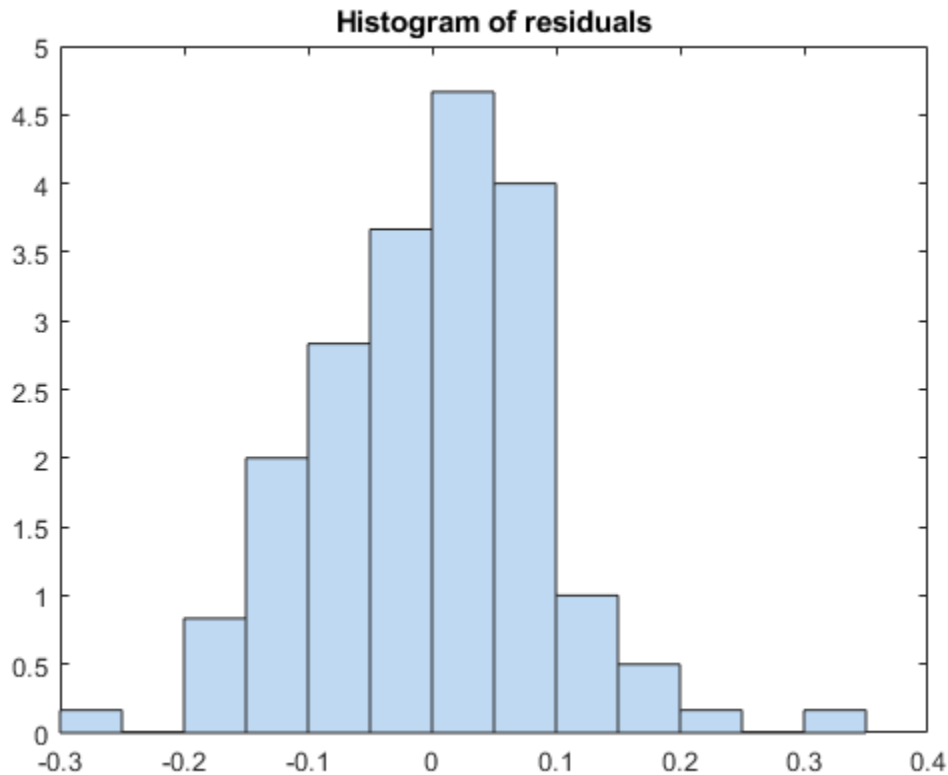
```
tbl = table(InitialWeight, Program, Subject, Week, y);
tbl.Subject = nominal(tbl.Subject);
tbl.Program = nominal(tbl.Program);
```

Fit a linear mixed-effects model where the initial weight, type of program, week, and the interaction between the week and type of program are the fixed effects. The intercept and week vary by subject.

```
lme = fitlme(tbl, 'y ~ InitialWeight + Program*Week + (Week|Subject)');
```

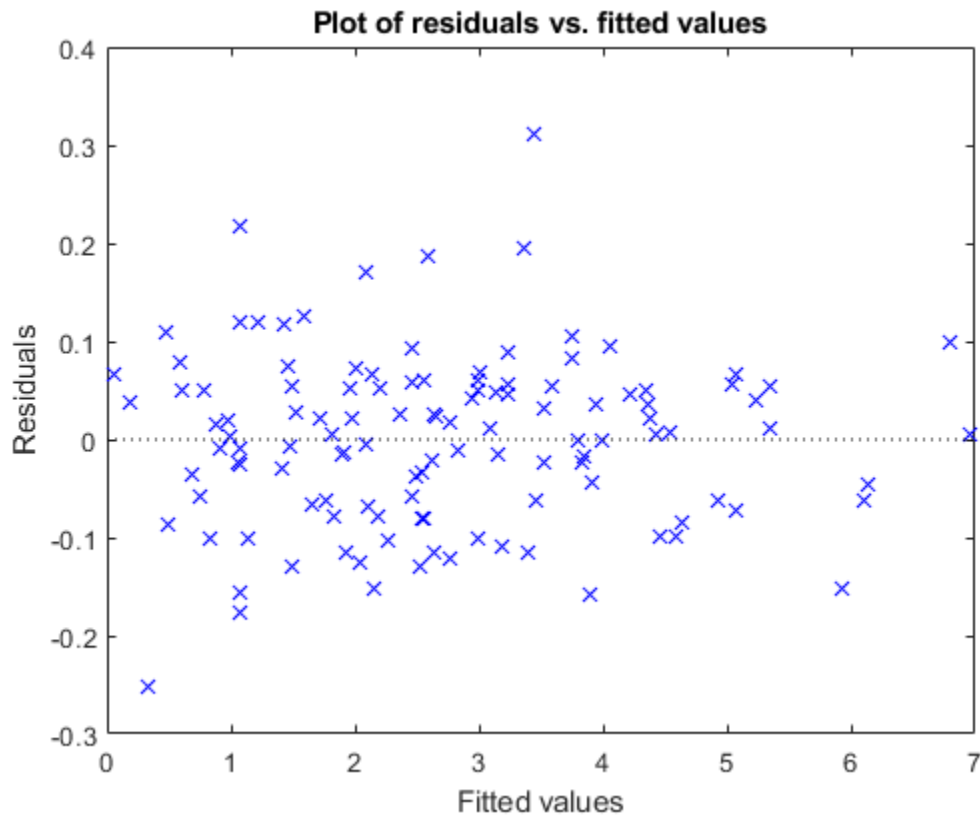
Plot the histogram of the raw residuals.

```
plotResiduals(lme)
```



Plot the residuals versus the fitted values.

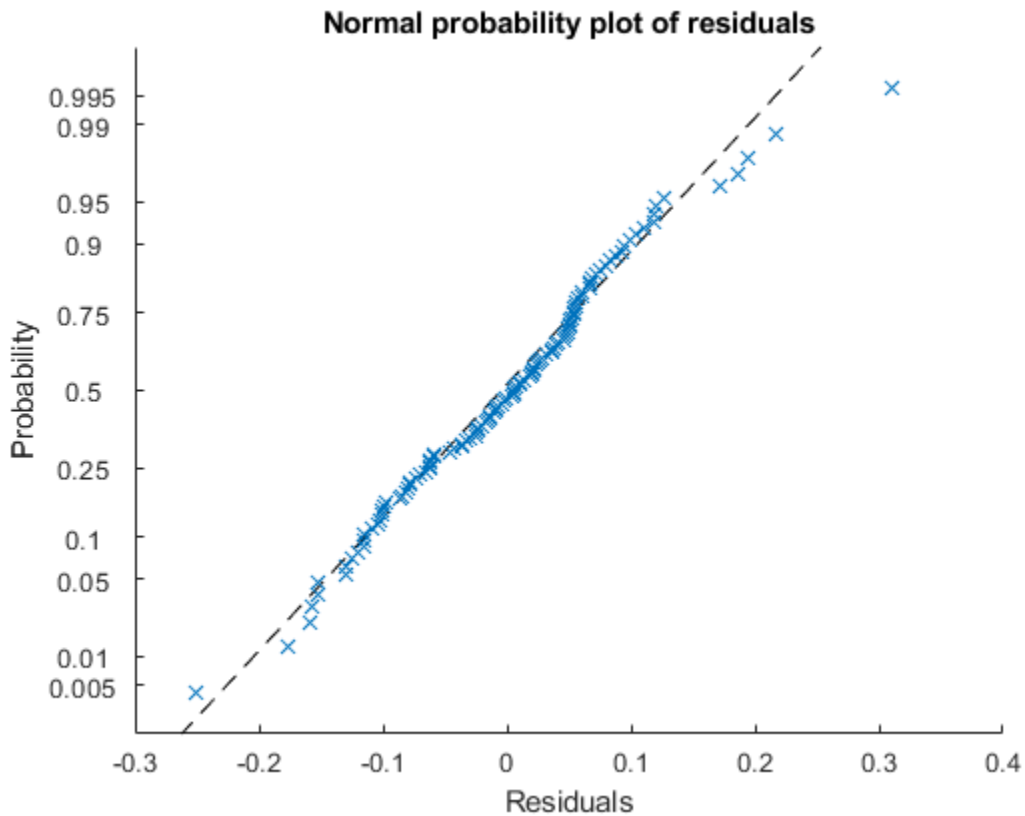
```
figure();  
plotResiduals(lme, 'fitted')
```



There is no obvious pattern, so there are no immediate signs of heteroscedasticity.

Create the normal probability plot of residuals.

```
figure();  
plotResiduals(lme, 'probability')
```



Data appears to be normal.

Find the observation number for the data that appears to be an outlier to the right of the plot.

```
find(residuals(lme)>0.25)
```

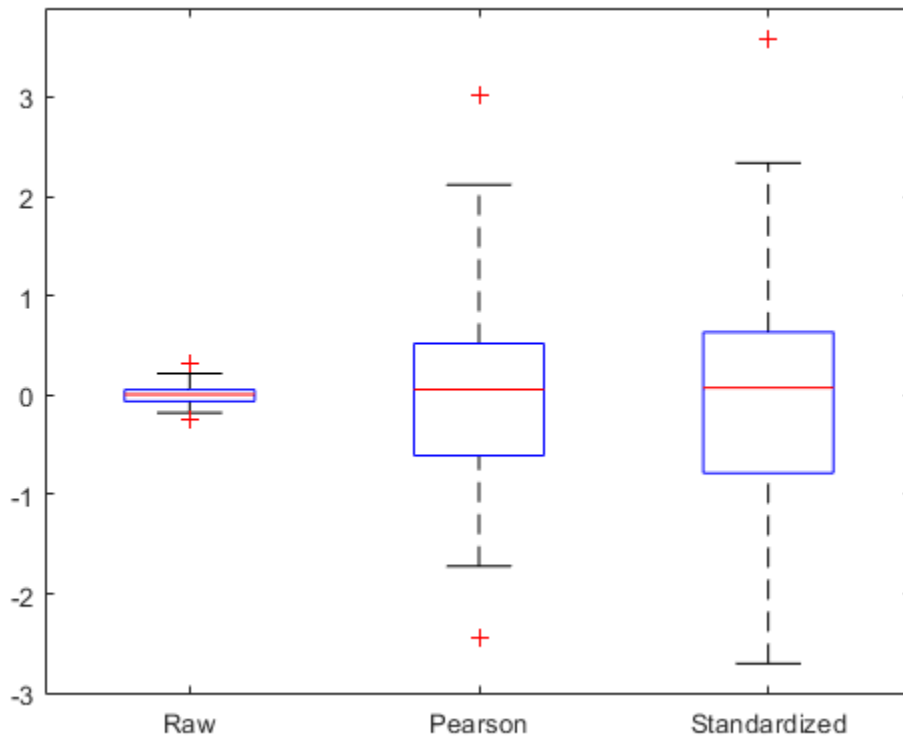
```
ans =
```

```
101
```

Create a box plot of the raw, Pearson, and standardized residuals.



```
r = residuals(lme);  
pr = residuals(lme, 'ResidualType', 'Pearson');  
st = residuals(lme, 'ResidualType', 'Standardized');  
X = [r pr st];  
boxplot(X, 'labels', {'Raw', 'Pearson', 'Standardized'});
```



All three box plots point out the outlier on the right tail of the distribution. The box plots of raw and Pearson residuals also point out a second possible outlier on the left tail. Find the corresponding observation number.

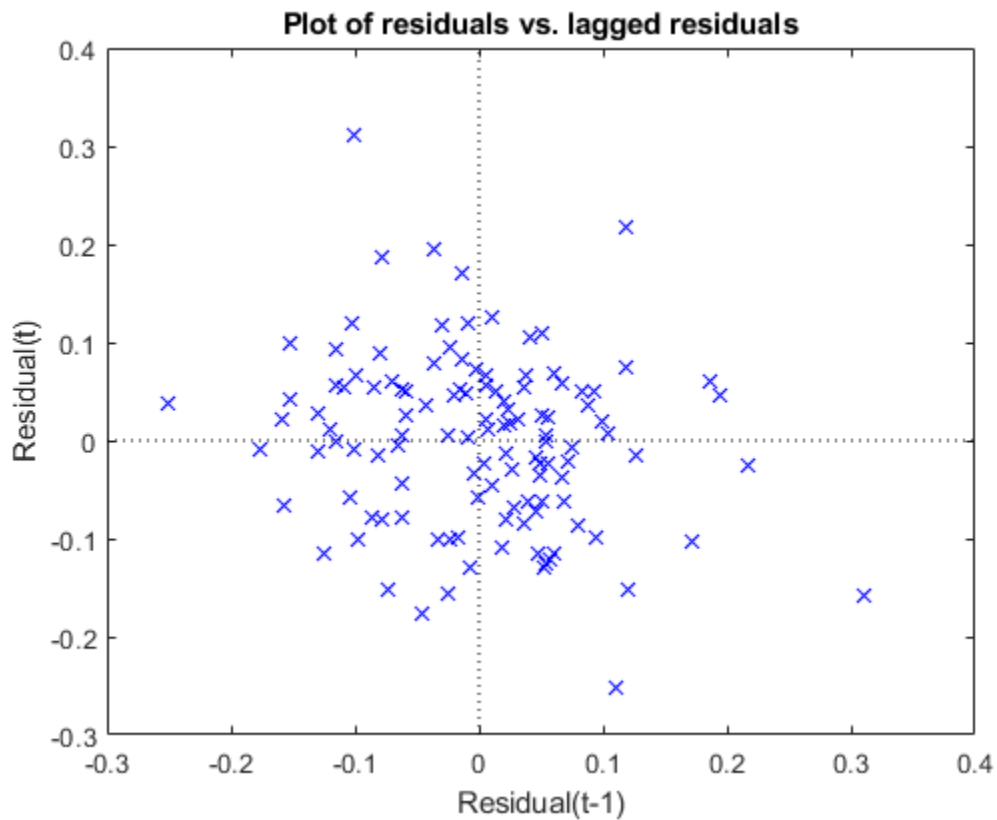
```
find(pr<-2)
```

```
ans =
```

10

Plot the raw residuals versus lagged residuals.

```
plotResiduals(lme, 'lagged')
```



There is no obvious pattern in the graph. The residuals do not appear to be correlated.

## See Also

`LinearMixedModel` | `fitted` | `residuals`

# plotResiduals

**Class:** NonLinearModel

Plot residuals of nonlinear regression model

## Syntax

```
plotResiduals mdl
plotResiduals mdl, plottype
h = plotResiduals (...)
h = plotResiduals mdl, plottype, Name, Value
```

## Description

`plotResiduals(mdl)` gives a histogram plot of the residuals of the `mdl` nonlinear model.

`plotResiduals(mdl, plottype)` plots residuals in a plot of type `plottype`.

`h = plotResiduals(...)` returns handles to the lines in the plot.

`h = plotResiduals(mdl, plottype, Name, Value)` plots with additional options specified by one or more `Name, Value` pair arguments.

## Input Arguments

### **mdl**

Nonlinear regression model, constructed by `fitnlm`.

### **plottype**

Character vector specifying the type of plot:

'caseorder'	Residuals vs. case (row) order
-------------	--------------------------------

'fitted'	Residuals vs. fitted values
'histogram'	Histogram
'lagged'	Residuals vs. lagged residual ( $r(t)$ vs. $r(t-1)$ )
'probability'	Normal probability plot
'symmetry'	Symmetry plot

**Default:** 'histogram'

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

---

**Note** The plot property name-value pairs apply to the first returned handle `h(1)`.

---

### Color

Color of the line or marker, a `ColorSpec` specification. For details, see `linespec`.

### LineStyle

Type of line, a `Chart Line` specification. For details, see `linespec`.

### LineWidth

Width of the line or edges of filled area, in points, a positive scalar. One point is 1/72 inch.

**Default:** 0.5

### MarkerEdgeColor

Color of the marker or edge color for filled markers, a `ColorSpec` specification. For details, see `linespec`.

**MarkerFaceColor**

Color of the marker face for filled markers, a `ColorSpec` specification. For details, see `linespec`.

**MarkerSize**

Size of the marker in points, a strictly positive scalar. One point is 1/72 inch.

**ResidualType**

Type of residual used in the plot:

'Raw'	Observed minus fitted values
'Pearson'	Raw residuals divided by RMSE
'Standardized'	Raw residuals divided by their estimated standard deviation
'Studentized'	Raw residuals divided by an independent (delete-1) estimate of their standard deviation

**Default:** 'Raw'

## Output Arguments

**h**

Vector of handles to lines or patches in the plot.

## Examples

**Residual Plot**

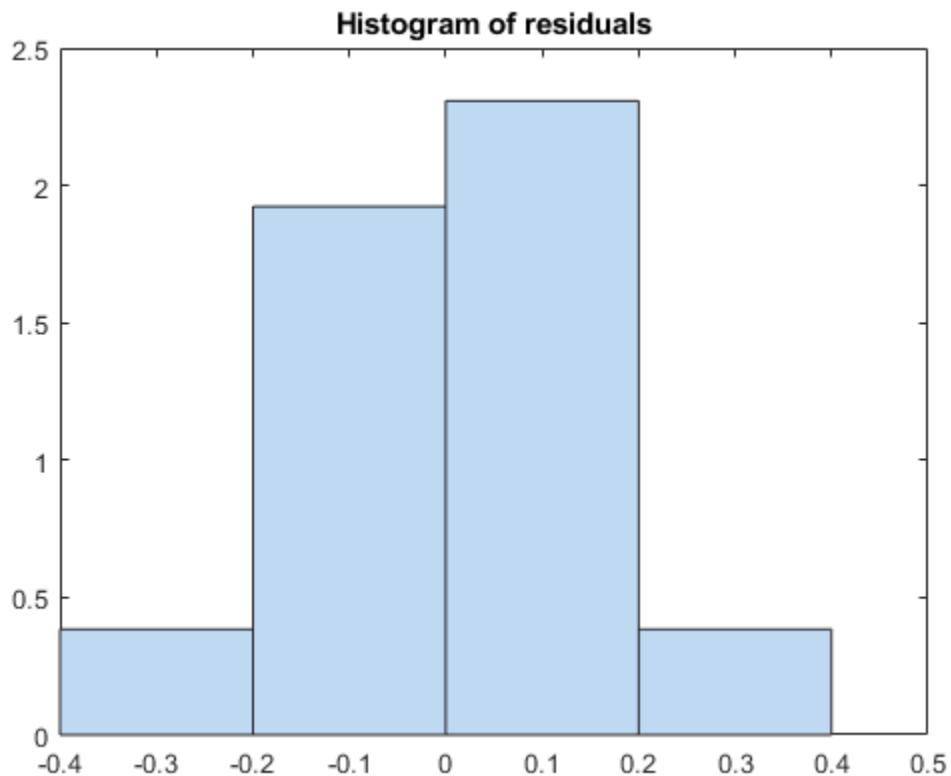
Plot the residuals of a fitted nonlinear model.

Load the reaction data and fit a model of the reaction rate as a function of reactants.

```
load reaction
mdl = fitnlm(reactants,rate,@hougen,[1 .05 .02 .1 2]);
```

Plot the residuals of the fitted model.

```
plotResiduals mdl)
```



### Residual Probability Plot

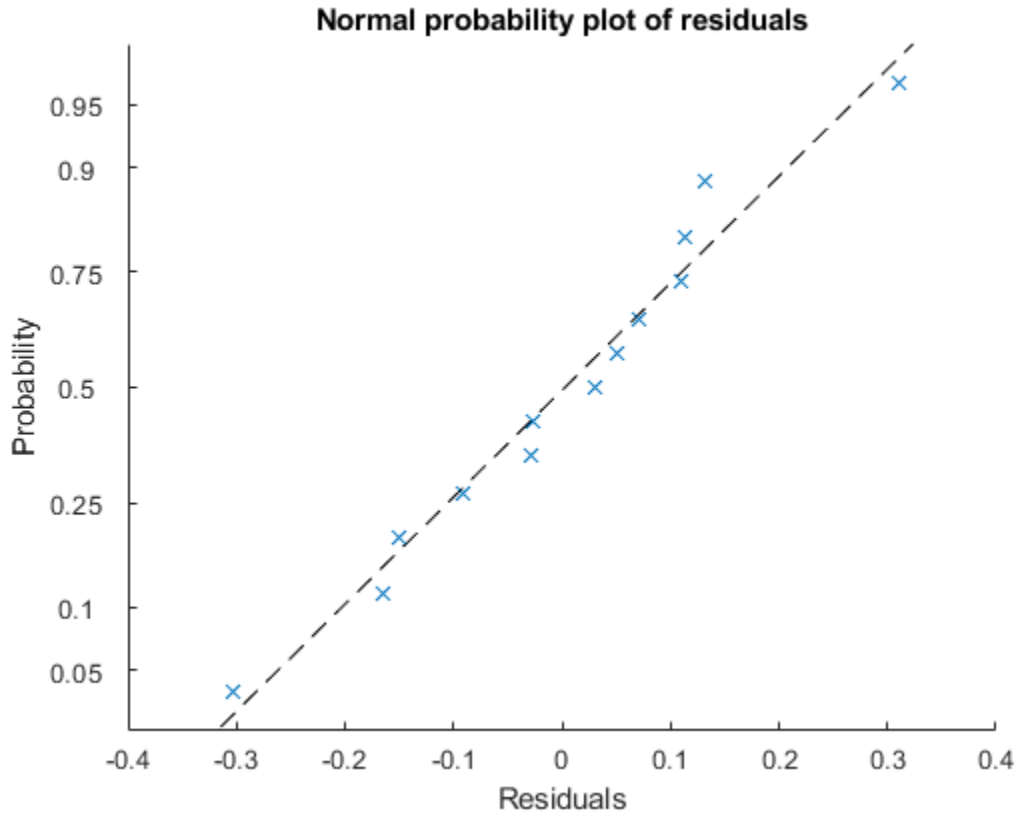
Create a normal probability plot of the residuals of a fitted nonlinear model.

Load the reaction data and fit a model of the reaction rate as a function of reactants.

```
load reaction
mdl = fitnlm(reactants, rate, @hougen, [1 .05 .02 .1 2]);
```

Create a normal probability plot of the residuals of the fitted model.

```
plotResiduals mdl, 'probability')
```



- “Examine Quality and Adjust the Fitted Nonlinear Model” on page 13-7
- “Nonlinear Regression Workflow” on page 13-15

## Tips

- For many plots, the Data Cursor tool in the figure window displays the  $x$  and  $y$  values for any data point, along with the observation name or number.

## See Also

`NonLinearModel` | `plotDiagnostics`

## Topics

“Examine Quality and Adjust the Fitted Nonlinear Model” on page 13-7

“Nonlinear Regression Workflow” on page 13-15

“Nonlinear Regression” on page 13-2



# plotSlice

**Class:** CompactGeneralizedLinearModel

Plot of slices through fitted generalized linear regression surface

## Syntax

```
plotSlice mdl  
h = plotSlice(mdl)
```

## Description

`plotSlice(mdl)` creates a new figure containing a series of plots, each representing a slice through the regression surface predicted by `mdl`. For each plot, the surface slice is shown as a function of a single predictor variable, with the other predictor variables held constant.

`h = plotSlice(mdl)` returns handles to the lines in the plot.

## Input Arguments

**mdl**

Generalized linear model, specified as a full `GeneralizedLinearModel` object constructed using `fitglm` or `stepwiseglm`, or a compacted `CompactGeneralizedLinearModel` object constructed using `compact`.

## Output Arguments

**h**

Vector of handles to lines or patches in the plot.

## Examples

### Slice Plot of Generalized Linear Regression Model

Create a slice plot of a Poisson generalized linear model.

Generate artificial data for the model using Poisson random numbers with two underlying predictors  $X(1)$  and  $X(2)$ .

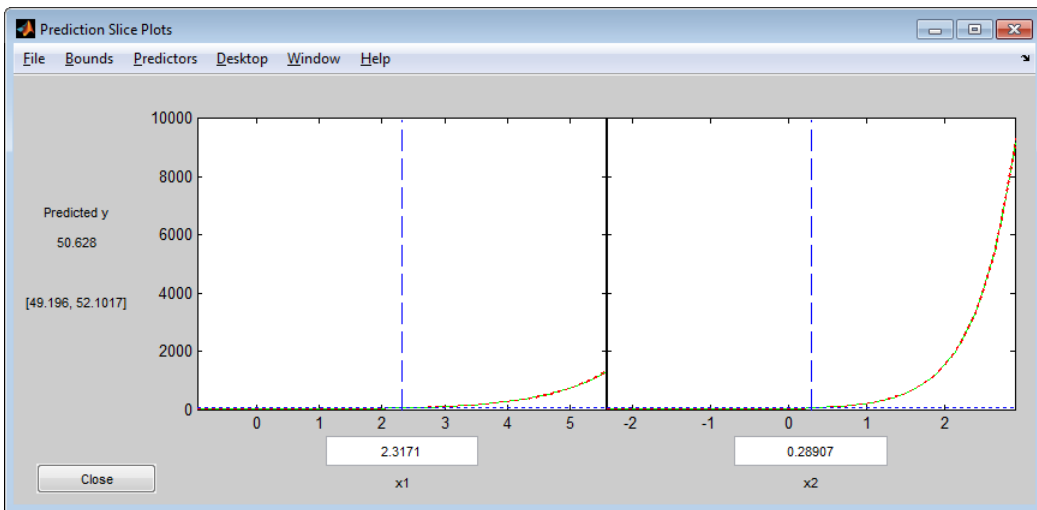
```
rng('default') % for reproducibility
rndvars = randn(100,2);
X = [2+rndvars(:,1), rndvars(:,2)];
mu = exp(1 + X*[1;2]);
y = poissrnd(mu);
```

Create a generalized linear regression model of Poisson data.

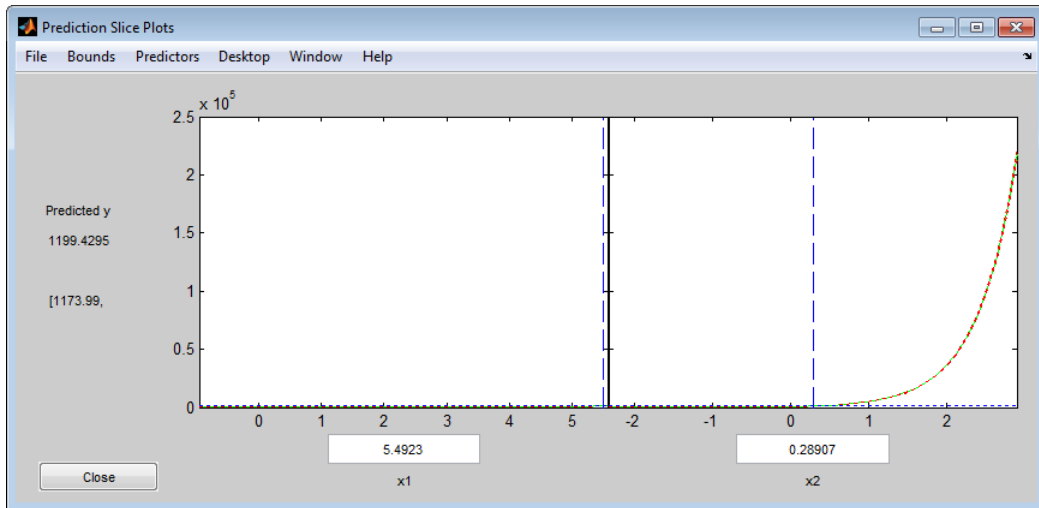
```
mdl = fitglm(X,y,'y ~ x1 + x2','distr','poisson');
```

Create the slice plot.

```
plotSlice(mdl)
```



Drag the  $x_1$  prediction line to the right and view the changes in the prediction and the response curve for the  $x_2$  predictor.



- “Diagnostic Plots” on page 12-25
- “Plots to Understand Predictor Effects and How to Modify a Model” on page 12-31

## Tips

- If there are more than eight predictors, `plotSlice` selects the first five for plotting. Use the **Predictors** menu to control which predictors are plotted.
- The **Bounds** menu lets you choose between simultaneous or non-simultaneous bounds, and between bounds on the function or bounds on a new observation.

## See Also

`CompactGeneralizedLinearModel` | `GeneralizedLinearModel` | `predict`

## Topics

- “Diagnostic Plots” on page 12-25
- “Plots to Understand Predictor Effects and How to Modify a Model” on page 12-31
- “Generalized Linear Models” on page 12-12

## plotSlice

**Class:** CompactLinearModel

Plot of slices through fitted linear regression surface

### Syntax

```
plotSlice mdl  
h = plotSlice mdl
```

### Description

`plotSlice mdl` creates a new figure containing a series of plots, each representing a slice through the regression surface predicted by `mdl`. For each plot, the surface slice is shown as a function of a single predictor variable, with the other predictor variables held constant.

`h = plotSlice mdl` returns handles to the lines in the plot.

### Input Arguments

**mdl** — Linear model object

LinearModel object | CompactLinearModel object

Linear model object, specified as a full `LinearModel` object constructed using `fitlm` or `stepwiselm`, or a compacted `CompactLinearModel` object constructed using `compact`.

### Output Arguments

**h** — Graphics handles

vector of graphics handle objects

Graphics handles, returned as a vector of graphics handles corresponding to the lines or patches in the plot.

## Examples

### Slice Plot for Linear Regression Model

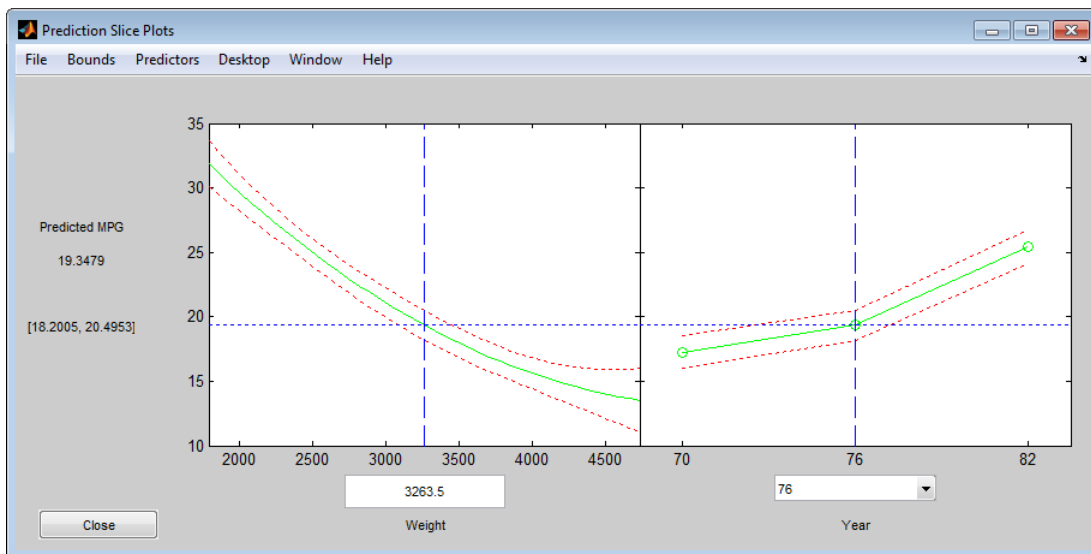
Plot the slices through a fitted linear model.

Load the `carsmall` data and fit a linear model of the mileage as a function of model year, weight, and weight squared.

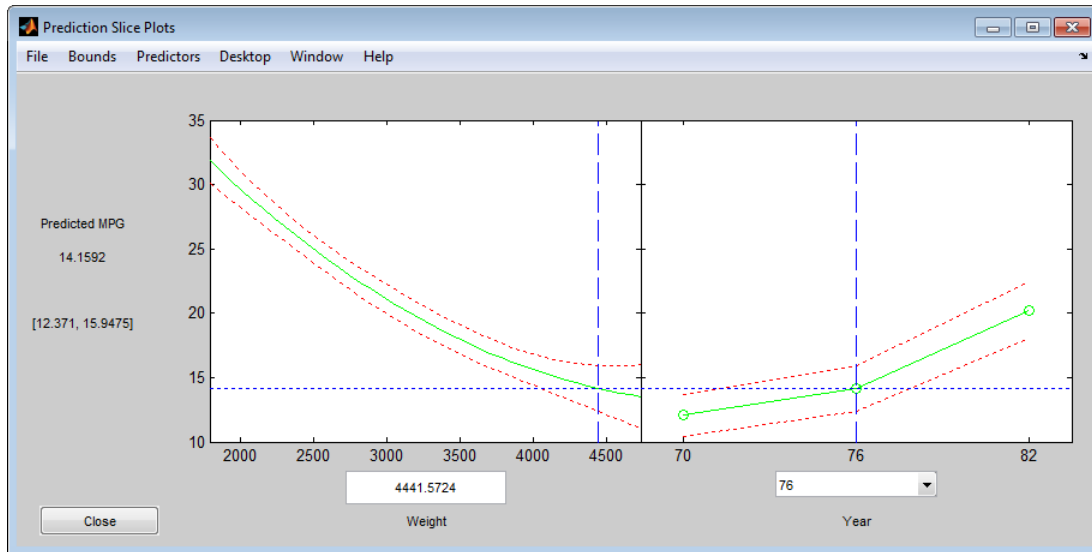
```
load carsmall
tbl = table(MPG,Weight);
tbl.Year = ordinal(Model_Year);
mdl = fitlm(tbl,'MPG ~ Year + Weight^2');
```

Create a slice plot.

```
plotSlice(mdl)
```



Drag the `Weight` prediction line to the right and observe the change in the predicted MPG and the response curve for `Year`.



- “Plots to Understand Predictor Effects” on page 11-29

## Tips

- If there are more than eight predictors, `plotSlice` selects the first five for plotting. Use the **Predictors** menu to control which predictors are plotted.
- The **Bounds** menu lets you choose between simultaneous or non-simultaneous bounds, and between bounds on the function or bounds on a new observation.

## See Also

`CompactLinearModel` | `LinearModel` | `predict`

## Topics

“Plots to Understand Predictor Effects” on page 11-29  
 “Linear Regression” on page 11-11

# plotSlice

**Class:** NonLinearModel

Plot of slices through fitted nonlinear regression surface

## Syntax

```
plotSlice mdl  
h = plotSlice(mdl)
```

## Description

`plotSlice(mdl)` creates a new figure containing a series of plots, each representing a slice through the regression surface predicted by `mdl`. For each plot, the surface slice is shown as a function of a single predictor variable, with the other predictor variables held constant.

`h = plotSlice(mdl)` returns handles to the lines in the plot.

## Input Arguments

**mdl**

Nonlinear regression model, constructed by `fitnlm`.

## Output Arguments

**h**

Vector of handles to lines or patches in the plot.

## Examples

### Slice Plot

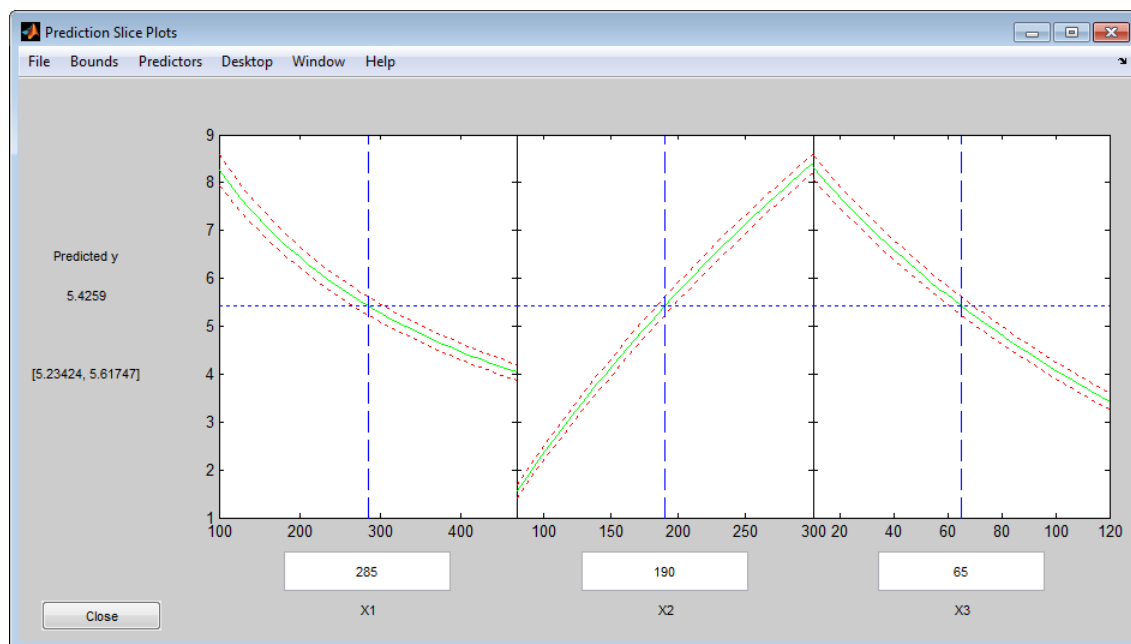
Plot slices of a fitted nonlinear model.

Load the reaction data and fit a model of the reaction rate as a function of reactants.

```
load reaction
mdl = fitnlm(reactants,...
    rate,@hougen,[1 .05 .02 .1 2]);
```

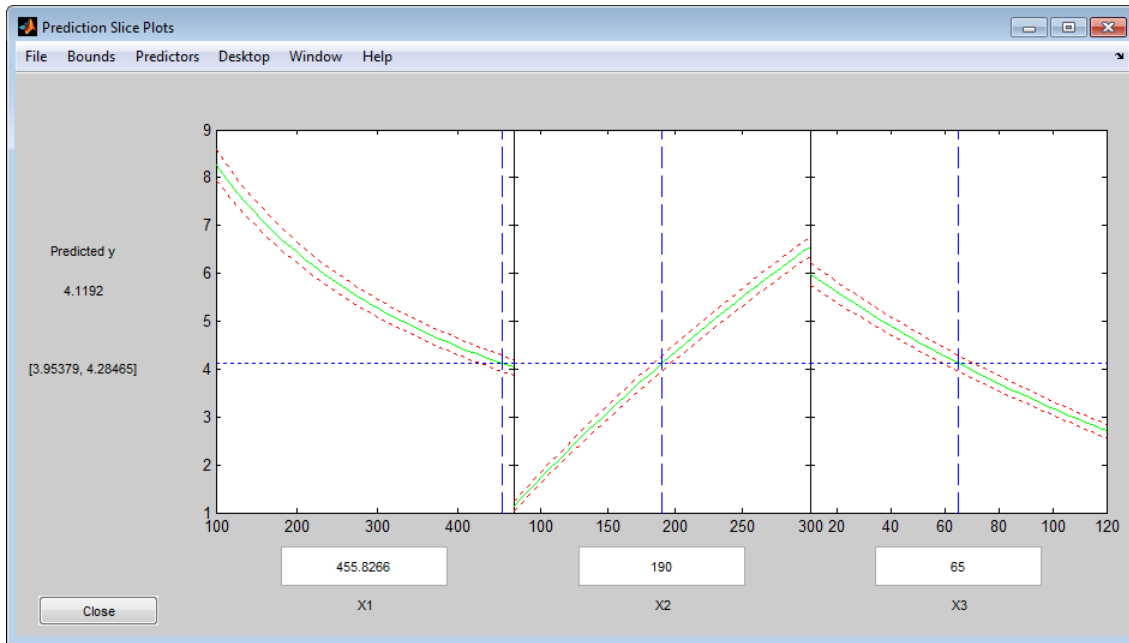
Create a slice plot

```
plotSlice(mdl)
```



Drag the  $x_1$  prediction line to the right, and observe the change in the predicted response  $y$  and in the predicted response curves to  $x_2$  and  $x_3$ .





- “Examine Quality and Adjust the Fitted Nonlinear Model” on page 13-7
- “Predict or Simulate Responses Using a Nonlinear Model” on page 13-10
- “Nonlinear Regression Workflow” on page 13-15

## Tips

- If there are more than eight predictors, `plotSlice` selects the first five for plotting. Use the **Predictors** menu to control which predictors are plotted.
- The **Bounds** menu lets you choose between simultaneous or non-simultaneous bounds, and between bounds on the function or bounds on a new observation.

## See Also

`NonLinearModel` | `predict`

## Topics

- “Examine Quality and Adjust the Fitted Nonlinear Model” on page 13-7
- “Predict or Simulate Responses Using a Nonlinear Model” on page 13-10
- “Nonlinear Regression Workflow” on page 13-15
- “Nonlinear Regression” on page 13-2

## plsregress

Partial least-squares regression

### Syntax

```
[XL,YL] = plsregress(X,Y,ncomp)
[XL,YL,XS] = plsregress(X,Y,ncomp)
[XL,YL,XS,YS] = plsregress(X,Y,ncomp)
[XL,YL,XS,YS,BETA] = plsregress(X,Y,ncomp,...)
[XL,YL,XS,YS,BETA,PCTVAR] = plsregress(X,Y,ncomp)
[XL,YL,XS,YS,BETA,PCTVAR,MSE] = plsregress(X,Y,ncomp)
[XL,YL,XS,YS,BETA,PCTVAR,MSE] =
plsregress(...,param1, val1,param2, val2,...)
[XL,YL,XS,YS,BETA,PCTVAR,MSE,stats] = plsregress(X,Y,ncomp,...)
```

### Description

`[XL,YL] = plsregress(X,Y,ncomp)` computes a partial least-squares (PLS) regression of  $Y$  on  $X$ , using  $ncomp$  PLS components, and returns the predictor and response loadings in  $XL$  and  $YL$ , respectively.  $X$  is an  $n$ -by- $p$  matrix of predictor variables, with rows corresponding to observations and columns to variables.  $Y$  is an  $n$ -by- $m$  response matrix.  $XL$  is a  $p$ -by- $ncomp$  matrix of predictor loadings, where each row contains coefficients that define a linear combination of PLS components that approximate the original predictor variables.  $YL$  is an  $m$ -by- $ncomp$  matrix of response loadings, where each row contains coefficients that define a linear combination of PLS components that approximate the original response variables.

`[XL,YL,XS] = plsregress(X,Y,ncomp)` returns the predictor scores  $XS$ , that is, the PLS components that are linear combinations of the variables in  $X$ .  $XS$  is an  $n$ -by- $ncomp$  orthonormal matrix with rows corresponding to observations and columns to components.

`[XL,YL,XS,YS] = plsregress(X,Y,ncomp)` returns the response scores  $YS$ , that is, the linear combinations of the responses with which the PLS components  $XS$  have

maximum covariance.  $YS$  is an  $n$ -by- $ncomp$  matrix with rows corresponding to observations and columns to components.  $YS$  is neither orthogonal nor normalized.

`plsregress` uses the SIMPLS algorithm, first centering  $X$  and  $Y$  by subtracting off column means to get centered variables  $X0$  and  $Y0$ . However, it does not rescale the columns. To perform PLS with standardized variables, use `zscore` to normalize  $X$  and  $Y$ .

If `ncomp` is omitted, its default value is `min(size(X,1)-1, size(X,2))`.

The relationships between the scores, loadings, and centered variables  $X0$  and  $Y0$  are:

$$XL = (XS \setminus X0)' = X0' * XS,$$

$$YL = (XS \setminus Y0)' = Y0' * XS,$$

$XL$  and  $YL$  are the coefficients from regressing  $X0$  and  $Y0$  on  $XS$ , and  $XS * XL'$  and  $XS * YL'$  are the PLS approximations to  $X0$  and  $Y0$ .

`plsregress` initially computes  $YS$  as:

$$YS = Y0 * YL = Y0 * Y0' * XS,$$

By convention, however, `plsregress` then orthogonalizes each column of  $YS$  with respect to preceding columns of  $XS$ , so that  $XS' * YS$  is lower triangular.

`[XL, YL, XS, YS, BETA] = plsregress(X, Y, ncomp, ...)` returns the PLS regression coefficients  $BETA$ .  $BETA$  is a  $(p+1)$ -by- $m$  matrix, containing intercept terms in the first row:

$$Y = [\text{ones}(n, 1), X] * BETA + Yresiduals,$$

$Y0 = X0 * BETA(2:end, :) + Yresiduals$ . Here  $Yresiduals$  is the vector of response residuals.

`[XL, YL, XS, YS, BETA, PCTVAR] = plsregress(X, Y, ncomp)` returns a 2-by- $ncomp$  matrix  $PCTVAR$  containing the percentage of variance explained by the model. The first row of  $PCTVAR$  contains the percentage of variance explained in  $X$  by each PLS component, and the second row contains the percentage of variance explained in  $Y$ .

`[XL, YL, XS, YS, BETA, PCTVAR, MSE] = plsregress(X, Y, ncomp)` returns a 2-by- $(ncomp+1)$  matrix  $MSE$  containing estimated mean-squared errors for PLS models with

0:ncomp components. The first row of MSE contains mean-squared errors for the predictor variables in X, and the second row contains mean-squared errors for the response variable(s) in Y.

[XL, YL, XS, YS, BETA, PCTVAR, MSE] =  
 plsregress(..., param1, val1, param2, val2, ...) specifies optional parameter name/value pairs from the following table to control the calculation of MSE.

Parameter	Value
'cv'	<p>The method used to compute MSE.</p> <ul style="list-style-type: none"> <li>• When the value is a positive integer <math>k</math>, plsregress uses <math>k</math>-fold cross-validation.</li> <li>• When the value is an object of the cvpartition on page 32-1292 class, other forms of cross-validation can be specified.</li> <li>• When the value is 'resubstitution', plsregress uses X and Y both to fit the model and to estimate the mean-squared errors, without cross-validation.</li> </ul> <p>The default is 'resubstitution'.</p>
'mcreps'	<p>A positive integer indicating the number of Monte-Carlo repetitions for cross-validation. The default value is 1. The value must be 1 if the value of 'cv' is 'resubstitution'.</p>
options	<p>A structure that specifies whether to run in parallel, and specifies the random stream or streams. Create the options structure with statset. Option fields:</p> <ul style="list-style-type: none"> <li>• UseParallel — Set to true to compute in parallel. Default is false.</li> <li>• UseSubstreams — Set to true to compute in parallel in a reproducible fashion. Default is false. To compute reproducibly, set Streams to a type allowing substreams: 'mlfg6331_64' or 'mrg32k3a'.</li> <li>• Streams — A RandStream object or cell array consisting of one such object. If you do not specify Streams, plsregress uses the default stream.</li> </ul>

[XL, YL, XS, YS, BETA, PCTVAR, MSE, stats] = plsregress(X, Y, ncomp, ...)  
 returns a structure stats with the following fields:

- W — A  $p$ -by- $n_{\text{comp}}$  matrix of PLS weights so that  $XS = X0 * W$ .
- T2 — The  $T^2$  statistic for each point in XS.

- `Xresiduals` — The predictor residuals, that is,  $X_0 - X_S * X_L$ '.
- `Yresiduals` — The response residuals, that is,  $Y_0 - X_S * Y_L$ '.

## Examples

### Perform Partial Least-Squares Regression

Load data on near infrared (NIR) spectral intensities of 60 samples of gasoline at 401 wavelengths, and their octane ratings.

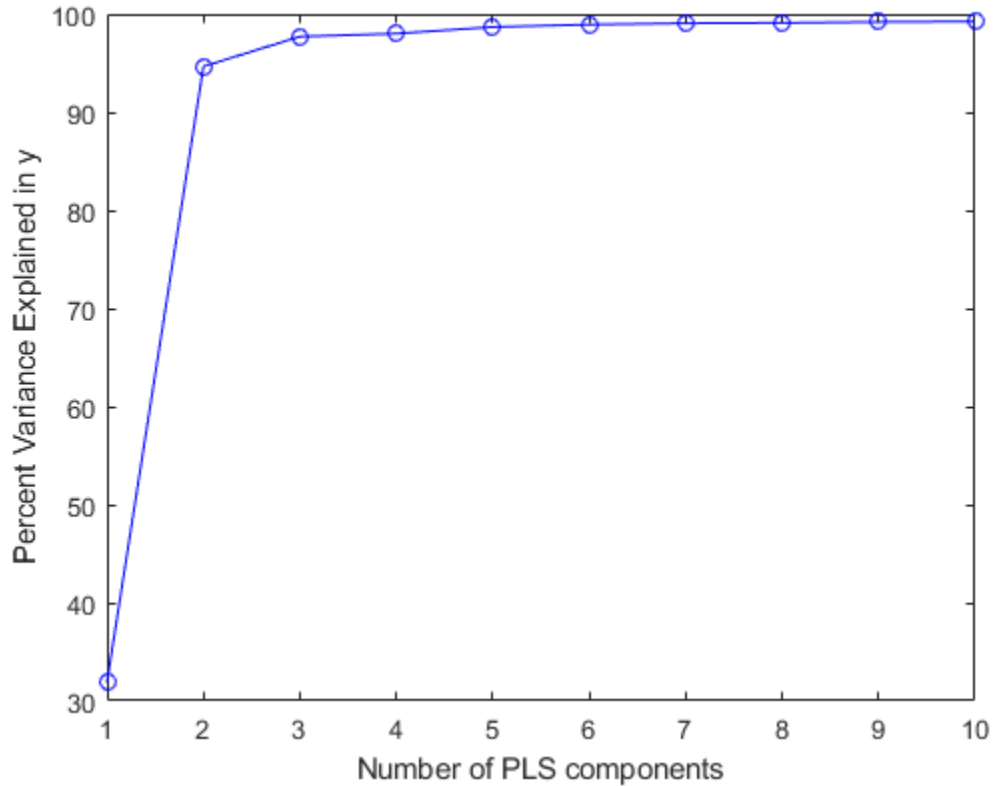
```
load spectra
X = NIR;
y = octane;
```

Perform PLS regression with ten components.

```
[XL,y1,XS,YS,beta,PCTVAR] = plsregress(X,y,10);
```

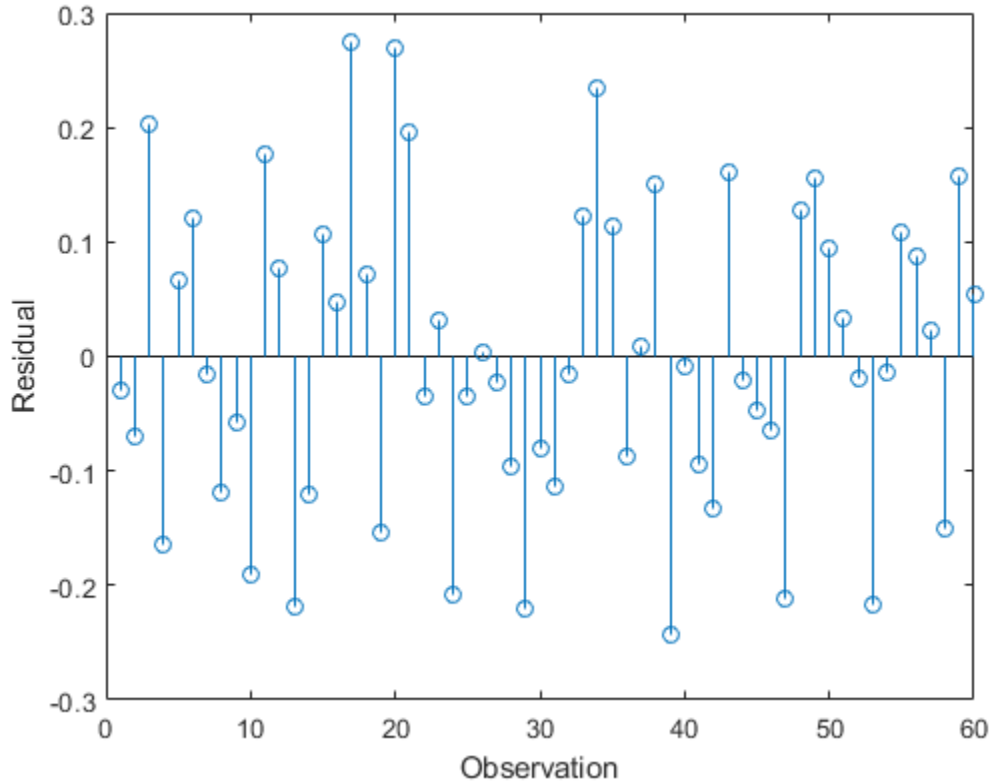
Plot the percent of variance explained in the response variable as a function of the number of components.

```
plot(1:10,cumsum(100*PCTVAR(2,:)),'-bo');
xlabel('Number of PLS components');
ylabel('Percent Variance Explained in y');
```



Compute the fitted response and display the residuals.

```
yfit = [ones(size(X,1),1) X]*beta;  
residuals = y - yfit;  
stem(residuals)  
xlabel('Observation');  
ylabel('Residual');
```



## References

- [1] de Jong, S. “SIMPLS: An Alternative Approach to Partial Least Squares Regression.” *Chemometrics and Intelligent Laboratory Systems*. Vol. 18, 1993, pp. 251–263.
- [2] Rosipal, R., and N. Kramer. “Overview and Recent Advances in Partial Least Squares.” *Subspace, Latent Structure and Feature Selection: Statistical and Optimization Perspectives Workshop (SLSFS 2005), Revised Selected Papers (Lecture Notes in Computer Science 3940)*. Berlin, Germany: Springer-Verlag, 2006, pp. 34–51.



## See Also

`regress` | `sequentialfs`

**Introduced in R2008a**

## PointOrder property

**Class:** sobolset

Point generation method

### Description

The `PointOrder` property contains a character vector that specifies the order in which the Sobol sequence points are produced. The property value must be one of `'standard'` or `'graycode'`. When set to `'standard'` the points produced match the original Sobol sequence implementation. When set to `'graycode'`, the sequence is generated using an implementation that uses the Gray code of the index instead of the index itself.

## PointSet property

**Class:** grandstream

Point set from which stream is drawn

## Description

The `PointSet` property contains a copy of the point set from which the stream is providing points. The point set is specified during construction of a quasi-random stream and cannot subsequently be altered.

## Examples

```
Q = grandstream('sobol', 5, 'Skip', 8);  
% Create a new stream based on the same sequence as that in Q  
Q2 = grandstream(Q.PointSet);  
u1 = grand(Q, 10)  
u2 = grand(Q2, 10) % contains exactly the same values as u1
```

## poisscdf

Poisson cumulative distribution function

### Syntax

```
p = poisscdf(x, lambda)
p = poisscdf(x, lambda, 'upper')
```

### Description

`p = poisscdf(x, lambda)` returns the Poisson cdf at each value in `x` using the corresponding mean parameters in `lambda`. `x` and `lambda` can be vectors, matrices, or multidimensional arrays that have the same size. A scalar input is expanded to a constant array with the same dimensions as the other input. The parameters in `lambda` must be positive.

`p = poisscdf(x, lambda, 'upper')` returns the complement of the Poisson cdf at each value in `x`, using an algorithm that more accurately computes the extreme upper tail probabilities.

The Poisson cdf is

$$p = F(x | \lambda) = e^{-\lambda} \sum_{i=0}^{\text{floor}(x)} \frac{\lambda^i}{i!}$$

### Examples

#### Compute Poisson Distribution cdf

For example, consider a Quality Assurance department that performs random tests of individual hard disks. Their policy is to shut down the manufacturing process if an inspector finds more than four bad sectors on a disk. What is the probability of shutting down the process if the mean number of bad sectors ( $\lambda$ ) is two?

```
probability = 1-poisscdf(4,2)
```

```
probability =  
    0.0527
```

About 5% of the time, a normally functioning manufacturing process produces more than four flaws on a hard disk.

Suppose the average number of flaws ( $\lambda$ ) increases to four. What is the probability of finding fewer than five flaws on a hard drive?

```
probability = poisscdf(4,4)
```

```
probability =  
    0.6288
```

This means that this faulty manufacturing process continues to operate after this first inspection almost 63% of the time.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

[cdf](#) | [poissfit](#) | [poissinv](#) | [poisspdf](#) | [poissrnd](#) | [poisstat](#)

### Topics

“Poisson Distribution” on page B-143

**Introduced before R2006a**

# poissfit

Poisson parameter estimates

## Syntax

```
lambdahat = poissfit(data)
[lambdahat,lambdaci] = poissfit(data)
[lambdahat,lambdaci] = poissfit(data,alpha)
```

## Description

`lambdahat = poissfit(data)` returns the maximum likelihood estimate (MLE) of the parameter of the Poisson distribution,  $\lambda$ , given the data `data`.

`[lambdahat,lambdaci] = poissfit(data)` also gives 95% confidence intervals in `lambdaci`.

`[lambdahat,lambdaci] = poissfit(data,alpha)` gives  $100(1 - \text{alpha})\%$  confidence intervals. For example `alpha = 0.001` yields 99.9% confidence intervals.

The sample mean is the MLE of  $\lambda$ .

$$\hat{\lambda} = \frac{1}{n} \sum_{i=1}^n x_i$$

## Examples

```
r = poissrnd(5,10,2);
[l,lci] = poissfit(r)
l =
    7.4000    6.3000
lci =
    5.8000    4.8000
    9.1000    7.9000
```

## See Also

`mle` | `poisscdf` | `poissinv` | `poisspdf` | `poissrnd` | `poisstat`

## Topics

“Poisson Distribution” on page B-143

**Introduced before R2006a**



## poissinv

Poisson inverse cumulative distribution function

### Syntax

```
X = poissinv(P,lambda)
```

### Description

`X = poissinv(P,lambda)` returns the smallest value  $X$  such that the Poisson cdf evaluated at  $X$  equals or exceeds  $P$ , using mean parameters in `lambda`.  $P$  and `lambda` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array with the same dimensions as the other input.

### Examples

If the average number of defects ( $\lambda$ ) is two, what is the 95th percentile of the number of defects?

```
poissinv(0.95,2)
ans =
     5
```

What is the median number of defects?

```
median_defects = poissinv(0.50,2)
median_defects =
     2
```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`icdf` | `poisscdf` | `poissfit` | `poisspdf` | `poissrnd` | `poisstat`

### Topics

“Poisson Distribution” on page B-143

Introduced before R2006a

# poisspdf

Poisson probability density function

## Syntax

```
Y = poisspdf(X, lambda)
```

## Description

`Y = poisspdf(X, lambda)` computes the Poisson pdf at each of the values in `X` using mean parameters in `lambda`. `X` and `lambda` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array with the same dimensions as the other input. The parameters in `lambda` must all be positive.

The Poisson pdf is

$$f(x | \lambda) = \frac{\lambda^x}{x!} e^{-\lambda}; x = 0, 1, 2, \dots, \infty.$$

The density function is zero unless  $x$  is an integer.

## Examples

A computer hard disk manufacturer has observed that flaws occur randomly in the manufacturing process at the average rate of two flaws in a 4 GB hard disk and has found this rate to be acceptable. What is the probability that a disk will be manufactured with no defects?

In this problem,  $\lambda = 2$  and  $x = 0$ .

```
p = poisspdf(0, 2)
p =
    0.1353
```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`pdf` | `poisscdf` | `poissfit` | `poissinv` | `poissrnd` | `poisstat`

### Topics

“Poisson Distribution” on page B-143

Introduced before R2006a

# poissrnd

Poisson random numbers

## Syntax

```
R = poissrnd(lambda)
R = poissrnd(lambda,m,n,...)
R = poissrnd(lambda,[m,n,...])
```

## Description

`R = poissrnd(lambda)` generates random numbers from the Poisson distribution with mean parameter `lambda`. `lambda` can be a vector, a matrix, or a multidimensional array. The size of `R` is the size of `lambda`.

`R = poissrnd(lambda,m,n,...)` or `R = poissrnd(lambda,[m,n,...])` generates an `m-by-n-by-...` array. The `lambda` parameter can be a scalar or an array of the same size as `R`.

## Examples

Generate a random sample of 10 pseudo-observations from a Poisson distribution with  $\lambda = 2$ .

```
lambda = 2;

random_sample1 = poissrnd(lambda,1,10)
random_sample1 =
    1    0    1    2    1    3    4    2    0    0

random_sample2 = poissrnd(lambda,[1 10])
random_sample2 =
    1    1    1    5    0    3    2    2    3    4

random_sample3 = poissrnd(lambda(ones(1,10)))
```

```
random_sample3 =  
    3     2     1     1     0     0     4     0     2     0
```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

The generated code can return a different sequence of numbers than MATLAB if either of the following is true:

- The output is nonscalar.
- An input parameter is invalid for the distribution.

### See Also

`poisscdf` | `poissfit` | `poissinv` | `poisspdf` | `poisstat` | `random`

### Topics

“Poisson Distribution” on page B-143

Introduced before R2006a

# poisstat

Poisson mean and variance

## Syntax

```
M = poisstat(lambda)
[M,V] = poisstat(lambda)
```

## Description

`M = poisstat(lambda)` returns the mean of the Poisson distribution using mean parameters in `lambda`. The size of `M` is the size of `lambda`.

`[M,V] = poisstat(lambda)` also returns the variance `V` of the Poisson distribution.

For the Poisson distribution with parameter  $\lambda$ , both the mean and variance are equal to  $\lambda$ .

## Examples

Find the mean and variance for the Poisson distribution with  $\lambda = 2$ .

```
[m,v] = poisstat([1 2; 3 4])
m =
     1     2
     3     4
v =
     1     2
     3     4
```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`poisscdf` | `poissfit` | `poissinv` | `poisspdf` | `poissrnd`

### Topics

“Poisson Distribution” on page B-143

Introduced before R2006a



# polyconf

Polynomial confidence intervals

## Syntax

```
Y = polyconf(p,X)
[Y,DELTA] = polyconf(p,X,S)
[Y,DELTA] = polyconf(p,X,S,param1,val1,param2,val2,...)
```

## Description

`Y = polyconf(p,X)` evaluates the polynomial `p` at the values in `X`. `p` is a vector of coefficients in descending powers.

`[Y,DELTA] = polyconf(p,X,S)` takes outputs `p` and `S` from `polyfit` and generates 95% prediction intervals  $Y \pm \text{DELTA}$  for new observations at the values in `X`.

`[Y,DELTA] = polyconf(p,X,S,param1,val1,param2,val2,...)` specifies optional parameter name/value pairs chosen from the following list.

Parameter	Value
'alpha'	A value between 0 and 1 specifying a confidence level of $100 * (1 - \text{alpha})\%$ . The default is 0.05.
'mu'	A two-element vector containing centering and scaling parameters. With this option, <code>polyconf</code> uses $(X - \text{mu}(1)) / \text{mu}(2)$ in place of <code>X</code> .
'predopt'	Either 'observation' (the default) to compute prediction intervals for new observations at the values in <code>X</code> , or 'curve' to compute confidence intervals for the fit evaluated at the values in <code>X</code> . See below.
'simopt'	Either 'off' (the default) for nonsimultaneous bounds, or 'on' for simultaneous bounds. See below.

The 'predopt' and 'simopt' parameters can be understood in terms of the following functions:

- $p(x)$  — the unknown mean function estimated by the fit
- $l(x)$  — the lower confidence bound
- $u(x)$  — the upper confidence bound

Suppose you make a new observation  $y_{n+1}$  at  $x_{n+1}$ , so that

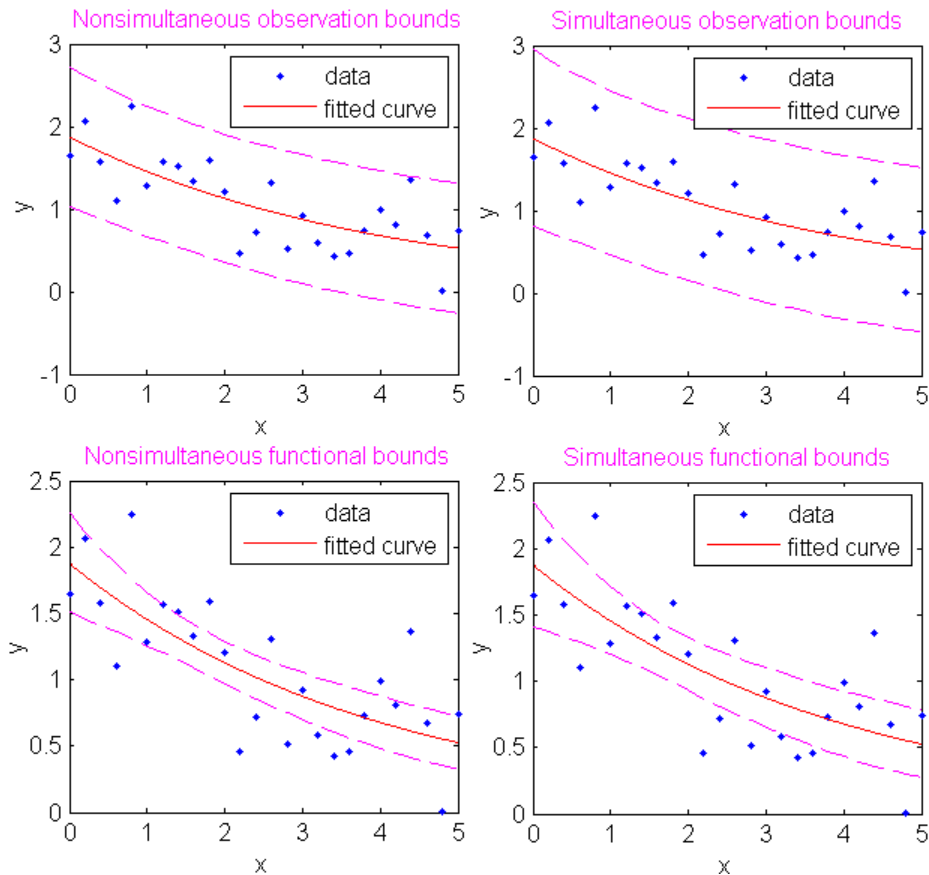
$$y_{n+1}(x_{n+1}) = p(x_{n+1}) + \varepsilon_{n+1}$$

By default, the interval  $[l_{n+1}(x_{n+1}), u_{n+1}(x_{n+1})]$  is a 95% confidence bound on  $y_{n+1}(x_{n+1})$ .

The following combinations of the 'predopt' and 'simopt' parameters allow you to specify other bounds.

'simopt'	'predopt'	Bounded Quantity
'off'	'observation'	$y_{n+1}(x_{n+1})$ (default)
'off'	'curve'	$p(x_{n+1})$
'on'	'observation'	$y_{n+1}(x)$ , for all $x$
'on'	'curve'	$p(x)$ , for all $x$

In general, 'observation' intervals are wider than 'curve' intervals, because of the additional uncertainty of predicting a new response value (the curve plus random errors). Likewise, simultaneous intervals are wider than nonsimultaneous intervals, because of the additional uncertainty of bounding values for all predictors  $x$ .



## Examples

This example uses code from the documentation example function `polydemo`, and calls the documentation example function `polystr` to convert the coefficient vector `p` into text for the polynomial expression displayed in the figure title. It combines the functions `polyfit`, `polyval`, `roots`, and `polyconf` to produce a formatted display of data with a polynomial fit.

**Note** Statistics and Machine Learning Toolbox documentation example files are located in the `\help\toolbox\stats\examples` subdirectory of your MATLAB root folder (`matlabroot`). This subdirectory is not on the MATLAB path at installation. To use the files in this subdirectory, either add the subdirectory to the MATLAB path (`addpath`) or make the subdirectory your current working folder (`cd`).

---

Display simulated data with a quadratic trend, a fitted quadratic polynomial, and 95% prediction intervals for new observations:

```
xdata = -5:5;
ydata = xdata.^2 - 5*xdata - 3 + 5*randn(size(xdata));

degree = 2;           % Degree of the fit
alpha = 0.05;        % Significance level

% Compute the fit and return the structure used by
% POLYCONF.
[p,S] = polyfit(xdata,ydata,degree);

% Compute the real roots and determine the extent of the
% data.
r = roots(p)';           % Roots as a row vector.
real_r = r(imag(r) == 0); % Real roots.

% Assure that the data are row vectors.
xdata = reshape(xdata,1,length(xdata));
ydata = reshape(ydata,1,length(ydata));

% Extent of the data.
mx = min([real_r,xdata]);
Mx = max([real_r,xdata]);
my = min([ydata,0]);
My = max([ydata,0]);

% Scale factors for plotting.
sx = 0.05*(Mx-mx);
sy = 0.05*(My-my);

% Plot the data, the fit, and the roots.
hdata = plot(xdata,ydata,'md','MarkerSize',5,...
             'LineWidth',2);
hold on
```

```

xfit = mx-sx:0.01:Mx+sx;
yfit = polyval(p,xfit);
hfit = plot(xfit,yfit,'b-','LineWidth',2);
hroots = plot(real_r,zeros(size(real_r)),...
              'bo','MarkerSize',5,...
              'LineWidth',2,...
              'MarkerFaceColor','b');

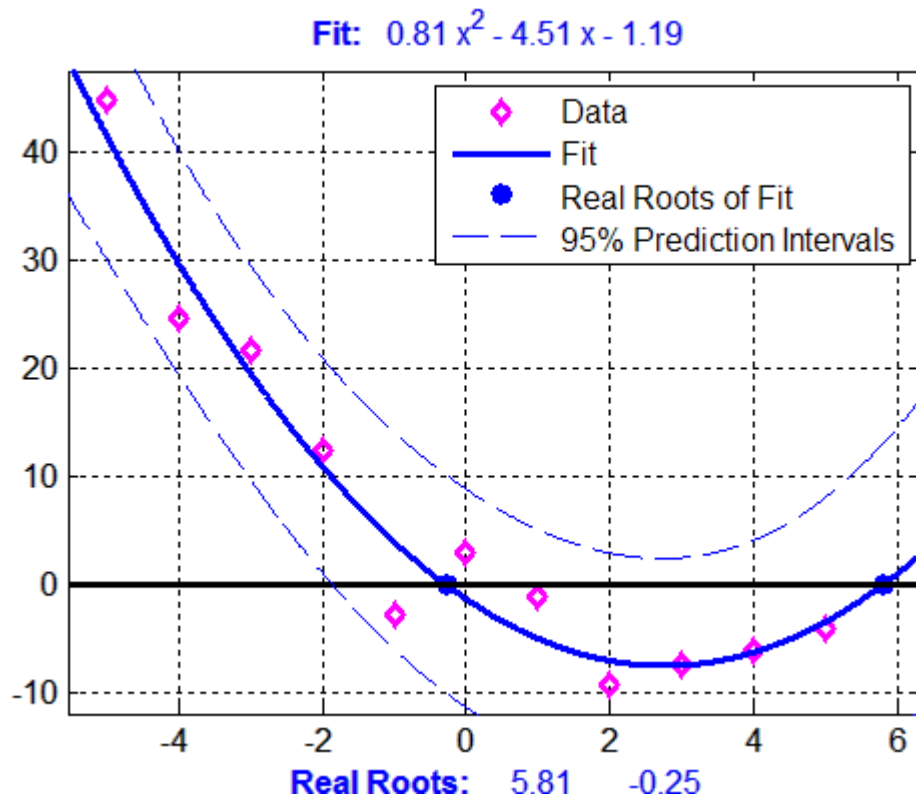
grid on
plot(xfit,zeros(size(xfit)),'k-','LineWidth',2)
axis([mx-sx Mx+sx my-sy My+sy])

% Add prediction intervals to the plot.
[Y,DELTA] = polyconf(p,xfit,S,'alpha',alpha);
hconf = plot(xfit,Y+DELTA,'b--');
plot(xfit,Y-DELTA,'b--')

% Display the polynomial fit and the real roots.
approx_p = round(100*p)/100; % Round for display.
htitle = title(['{\bf Fit:  }',...
               texlabel(polystr(approx_p))]);
set(htitle,'Color','b')
approx_real_r = round(100*real_r)/100; % Round for display.
hxlabel = xlabel(['{\bf Real Roots:  }',...
                 num2str(approx_real_r)]);
set(hxlabel,'Color','b')

% Add a legend.
legend([hdata,hfit,hroots,hconf],...
       'Data','Fit','Real Roots of Fit',...
       '95% Prediction Intervals')

```



## See Also

`polyfit` | `polytool` | `polyval`

Introduced before R2006a

# polytool

Interactive polynomial fitting

## Syntax

```
polytool(x,y)
polytool(x,y,n)
polytool(x,y,n,alpha)
polytool(x,y,n,alpha,xname,yname)
h = polytool(...)
```

## Description

`polytool(x,y)` fits a line to the vectors `x` and `y` and displays an interactive plot of the result in a graphical interface. You can use the interface to explore the effects of changing the parameters of the fit and to export fit results to the workspace.

`polytool(x,y,n)` initially fits a polynomial of degree `n`. The default is 1, which produces a linear fit.

`polytool(x,y,n,alpha)` initially plots  $100(1 - \text{alpha})\%$  confidence intervals on the predicted values. The default is 0.05 which results in 95% confidence intervals.

`polytool(x,y,n,alpha,xname,yname)` labels the `x` and `y` values on the graphical interface using `xname` and `yname`. Specify `n` and `alpha` as `[]` to use their default values.

`h = polytool(...)` outputs a vector of handles, `h`, to the line objects in the plot. The handles are returned in the degree: data, fit, lower bounds, upper bounds.

## Examples

### Interactive polynomial fitting

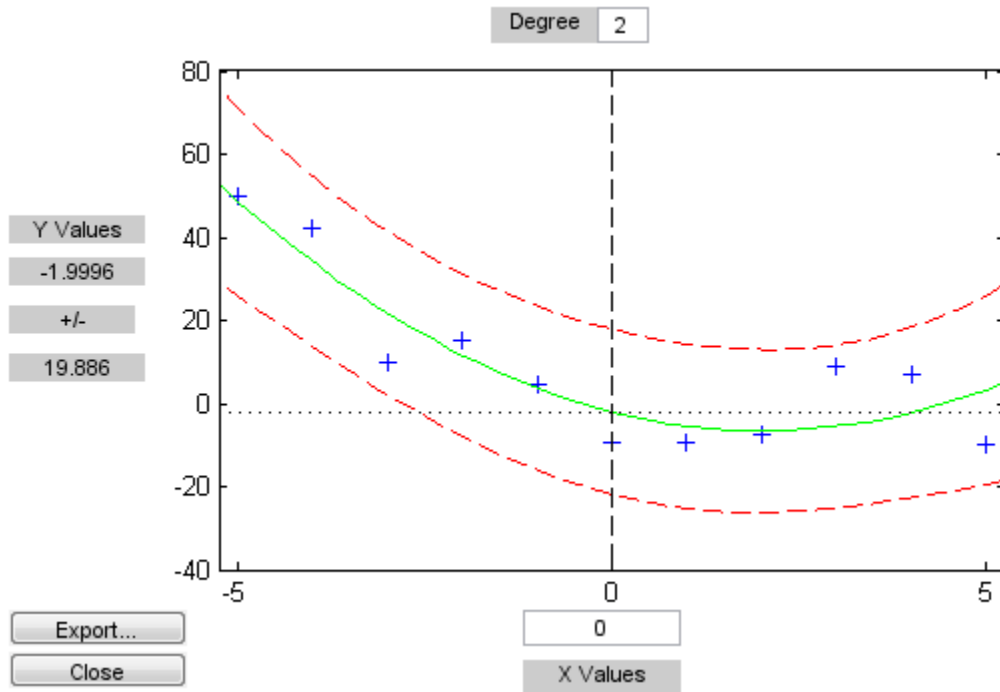
This example shows how to start an interactive fitting session with `polytool`.

Generate data from a quadratic curve with added noise.

```
rng('default') % for reproducibility
x = -5:5;
y = x.^2 - 5*x - 3 + 5*randn(size(x));
```

Fit a quadratic (degree-2) model with 0.90 confidence intervals.

```
n = 2;
alpha = 0.1;
polytool(x, y, n, alpha)
```



## See Also

`invpred` | `polyconf` | `polyfit` | `polyval`

Introduced before R2006a



# posterior

**Class:** `gmdistribution`

Posterior probabilities of components

## Syntax

```
P = posterior(obj,X)
[P,nlogl] = posterior(obj,X)
```

## Description

`P = posterior(obj,X)` returns the posterior probabilities of each of the  $k$  components in the Gaussian mixture distribution defined by `obj` for each observation in the data matrix  $X$ .  $X$  is  $n$ -by- $d$ , where  $n$  is the number of observations and  $d$  is the dimension of the data. `obj` is an object created by `gmdistribution` or `fitgmdist`.  $P$  is  $n$ -by- $k$ , with  $P(I, J)$  the probability of component  $J$  given observation  $I$ .

`posterior` treats NaN values as missing data. Rows of  $X$  with NaN values are excluded from the computation.

`[P,nlogl] = posterior(obj,X)` also returns `nlogl`, the negative log-likelihood of the data.

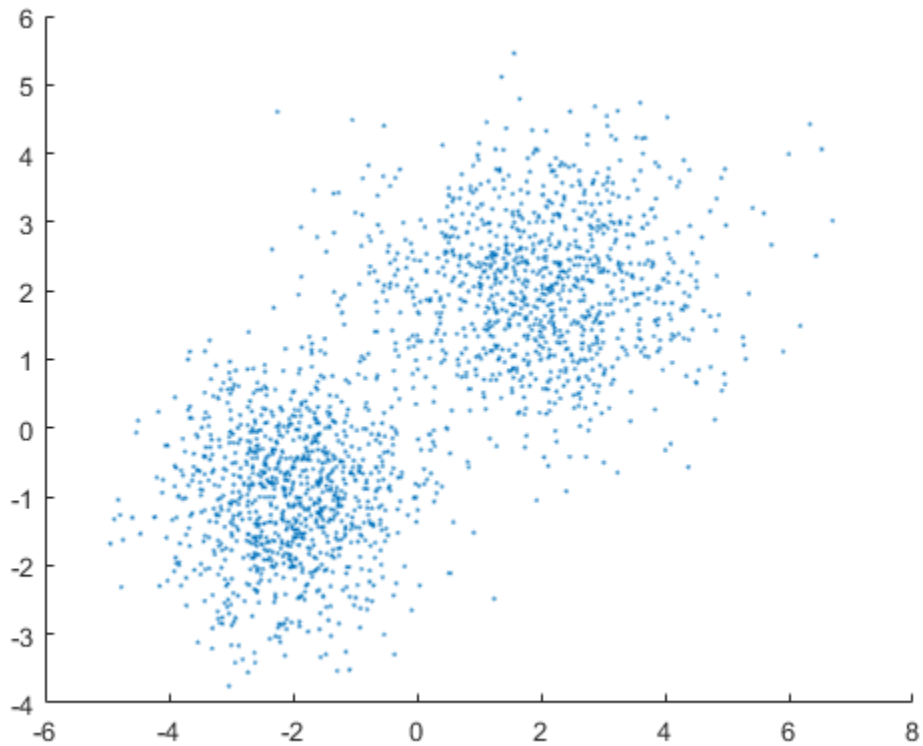
## Examples

### Compute Posterior Probabilities for Gaussian Mixture Variates

Generate data from a mixture of two bivariate Gaussian distributions using the `mvnrnd` function.

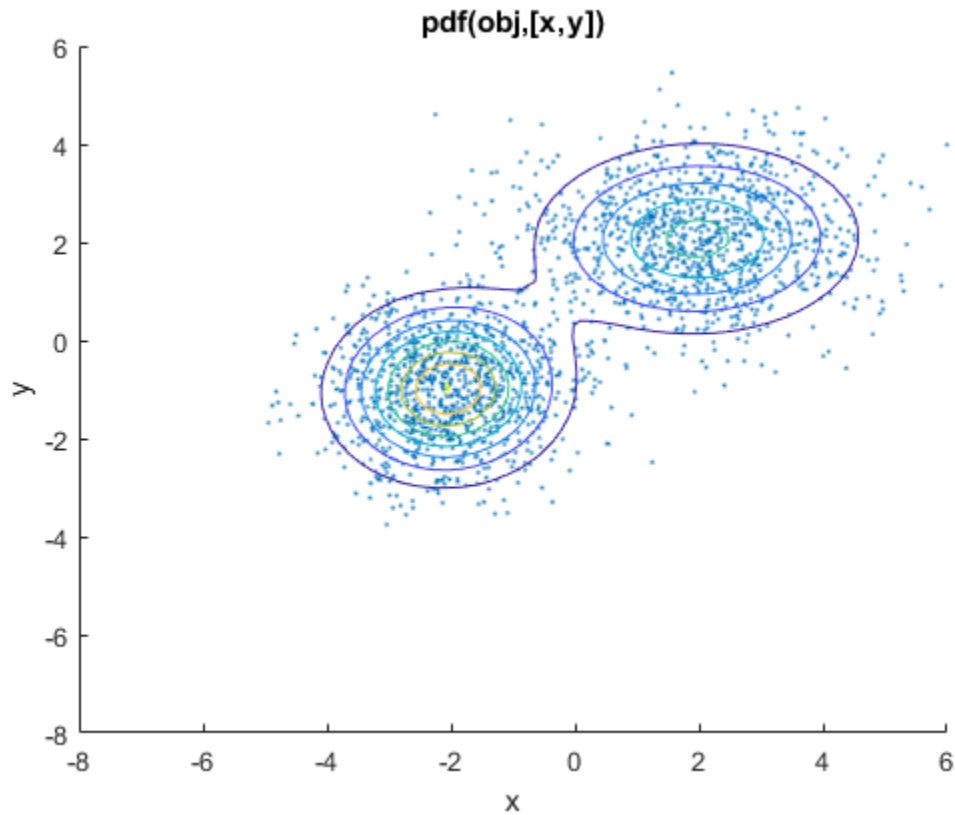
```
MU1 = [2 2];
SIGMA1 = [2 0; 0 1];
```

```
MU2 = [-2 -1];  
SIGMA2 = [1 0; 0 1];  
rng(1); % For reproducibility  
X = [mvnrnd(MU1,SIGMA1,1000);mvnrnd(MU2,SIGMA2,1000)];  
  
scatter(X(:,1),X(:,2),10,'.')  
hold on
```



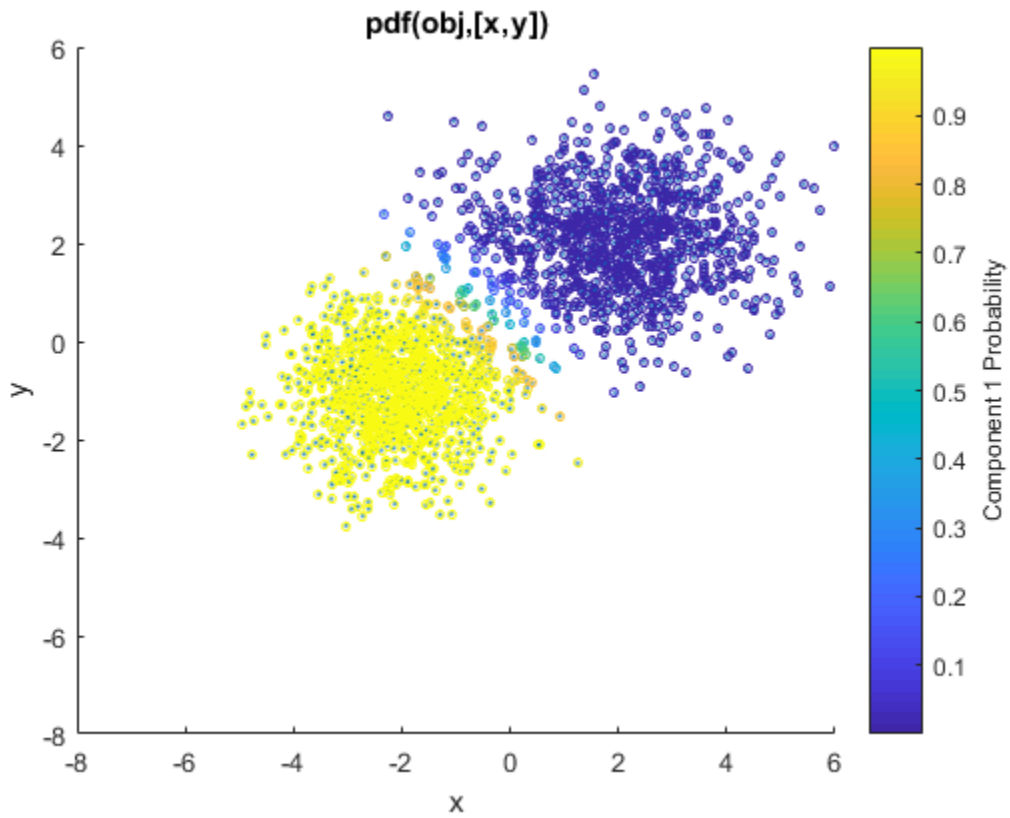
Fit a two-component Gaussian mixture model.

```
obj = fitgmdist(X,2);  
h = ezcontour(@(x,y)pdf(obj,[x y]),[-8 6],[-8 6]);
```



Compute posterior probabilities of the components.

```
P = posterior(obj,X);  
  
delete(h)  
scatter(X(:,1),X(:,2),10,P(:,1))  
hb = colorbar;  
ylabel(hb,'Component 1 Probability')
```



## See Also

`cluster` | `fitgmdist` | `gmdistribution` | `mahal`

# posterior

**Class:** NaiveBayes

Compute posterior probability of each class for test data

## Syntax

```
post = posterior(nb, test)
[post, cpre] = posterior(nb, test)
[post, cpre, logp] = posterior(nb, test)
[...] = posterior(..., 'HandleMissing', val)
```

## Description

`post = posterior(nb, test)` returns the posterior probability of the observations in `test` according to the NaiveBayes object `nb`. `test` is a `N-by-nb.ndims` matrix, where `N` is the number of observations in the test data. Rows of `test` correspond to points, columns of `test` correspond to features. `post` is a `N-by-nb.nclasses` matrix containing the posterior probability of each observation for each class. `post(i, j)` is the posterior probability of point `i` belonging to class `j`. Classes are ordered the same as `nb.clevels`, i.e., column `j` of `post` corresponds to the `j`th class in `nb.clevels`. The posterior probabilities corresponding to any empty classes are `NaN`.

`[post, cpre] = posterior(nb, test)` returns `cpre`, an `N-by-1` vector, containing the class to which each row of `test` has been assigned. `cpre` has the same type as `nb.CLevels`.

`[post, cpre, logp] = posterior(nb, test)` returns `logp`, an `N-by-1` vector containing estimates of the log of the probability density function (PDF). `logp(i)` is the log of the PDF of point `i`. The PDF value of point `i` is the sum of  $\text{Prob}(\text{point } i \mid \text{class } J) * \text{Pr}\{\text{class } J\}$  taken over all classes.

`[...] = posterior(..., 'HandleMissing', val)` specifies how `posterior` treats `NaN` (missing values). `val` can be one of the following:

'off' (default)	Observations with NaN in any of the columns are not classified into any class. The corresponding rows in <code>post</code> and <code>logp</code> are NaN. The corresponding rows in <code>cpre</code> are NaN (if <code>obj.clevels</code> is numeric or logical), empty character vectors (if <code>obj.clevels</code> is character array or cell array of character vectors) or <code>&lt;undefined&gt;</code> (if <code>obj.clevels</code> is categorical).
'on'	For observations having NaN in some (but not all) columns, <code>post</code> and <code>cpre</code> are computed using the columns with non-NaN values. Corresponding <code>logp</code> values are NaN.

## See Also

`NaiveBayes` | `fitNaiveBayes` | `predict`

# postFitStatistics

**Class:** RegressionGP

Compute post-fit statistics for the exact Gaussian process regression model

## Syntax

```
loores = postFitStatistics(gprMdl)
[loores,neff] = postFitStatistics(gprMdl)
```

## Description

`loores = postFitStatistics(gprMdl)` returns the leave-one-out residuals, `loores`, for the trained Gaussian process regression (GPR) model.

`[loores,neff] = postFitStatistics(gprMdl)` also returns the number of effective parameters, `neff`.

## Input Arguments

**gprMdl** — Gaussian process regression model

RegressionGP object

Gaussian process regression model, specified as a RegressionGP object.

## Output Arguments

**loores** — Leave-one-out residuals

*n*-by-1 matrix

Leave-one-out residuals, returned as an *n*-by-1 matrix, where *n* is the number of observations in the training data.

**neff** — Number of effective parameters*n*-by-1 matrix

Number of effective parameters, returned as an *n*-by-1 matrix, where *n* is the number of observations in the training data.

## Examples

### Compute Post-Fit Statistics

Generate sample data.

```
rng(0, 'twister'); % For reproducibility
n = 1500;
x = linspace(-10, 10, n)';
y = sin(3*x).*cos(3*x) + sin(2*x).*cos(2*x) + sin(x) + cos(x) + 0.2*randn(n,1);
```

Fit a GPR model using the exact method for fitting and prediction.

```
gprMdl = fitrgp(x, y, 'Basis', 'linear', 'FitMethod', 'exact', ...
    'PredictMethod', 'exact', 'KernelFunction', 'matern52');
```

Compute the leave-one-out residuals and the effective number of parameters in the trained model.

```
[loores, neff] = postFitStatistics(gprMdl);
```

Predict the responses using the trained model.

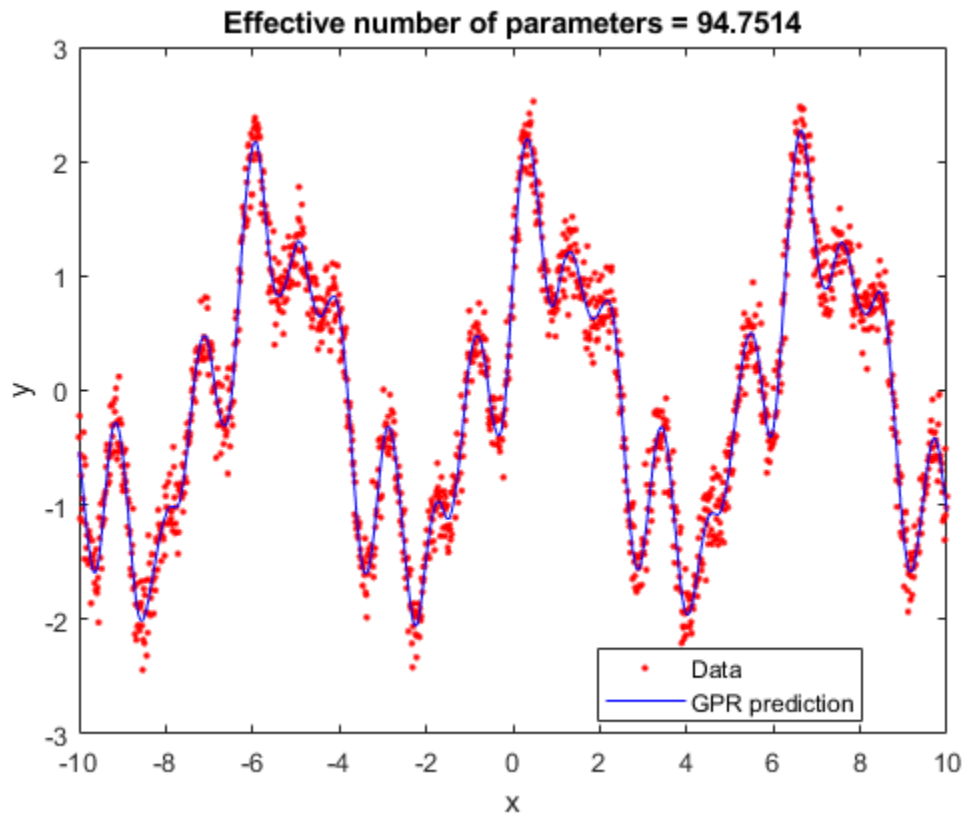
```
ypred = resubPredict(gprMdl);
```

Plot the true and predicted responses, and display effective number of parameters in the fit.

```
figure()
plot(x, y, 'r. ');
hold on;
plot(x, ypred, 'b');
xlabel('x');
ylabel('y');
legend('Data', 'GPR prediction', 'Location', 'Best');
```

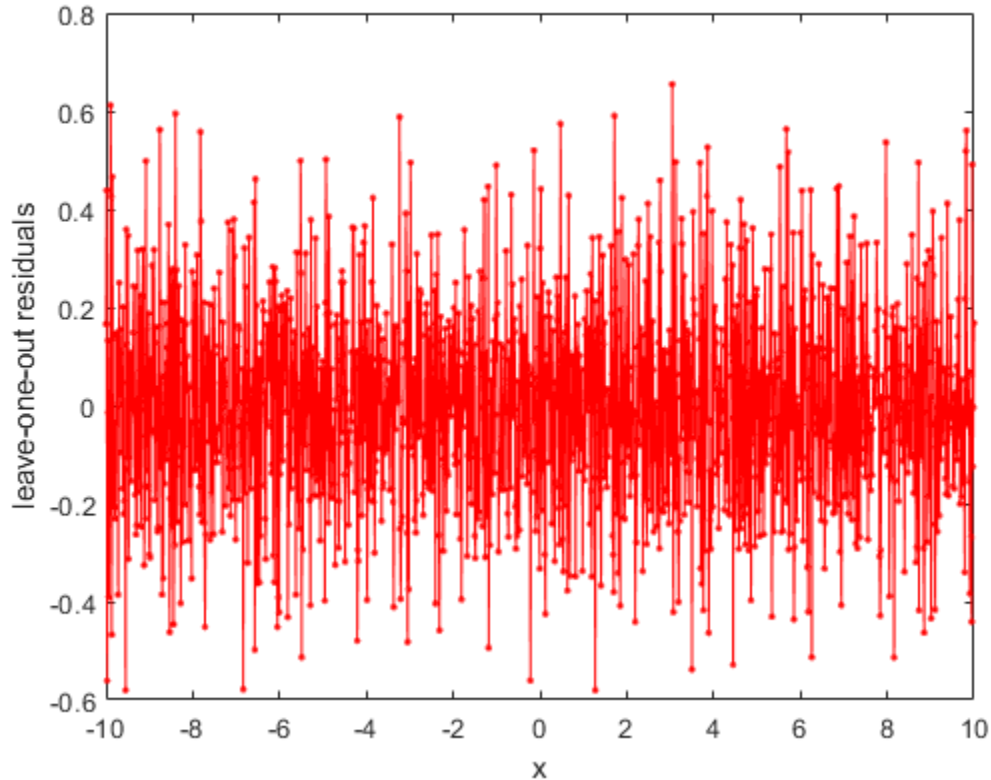


```
title(['Effective number of parameters = ', num2str(neff)]);  
hold off
```



Plot leave-one-out residuals.

```
figure()  
plot(x, loores, 'r.-');  
xlabel('x');  
ylabel('leave-one-out residuals');
```



## Tips

- You can only compute the post-fit statistics when `PredictMethod` is `'exact'`.
- If `FitMethod` is `'exact'`, then `postFitStatistics` accounts for the fact that the fixed basis function coefficients are estimated from the data.
- If `FitMethod` is different than `'exact'`, then `postFitStatistics` treats the fixed basis function coefficients as known.
- For all `PredictMethod` and `FitMethod` options, `postFitStatistics` treats the estimated kernel parameters and noise standard deviation as known.

## See Also

RegressionGP | fitrgp

**Introduced in R2015b**

## prctile

Percentiles of a data set

### Syntax

```
Y = prctile(X,p)
Y = prctile(X,p,dim)
```

### Description

`Y = prctile(X,p)` returns percentiles of the values in a data vector or matrix `X` for the percentages `p` in the interval `[0,100]`.

- If `X` is a vector, then `Y` is a scalar or a vector with the same length as the number of percentiles required (`length(p)`). `Y(i)` contains the `p(i)` percentile.
- If `X` is a matrix, then `Y` is a row vector or a matrix, where the number of rows of `Y` is equal to the number of percentiles required (`length(p)`). The `i`th row of `Y` contains the `p(i)` percentiles of each column of `X`.
- For multidimensional arrays on page 32-4890, `prctile` operates along the first nonsingleton dimension on page 32-4890 of `X`.

`Y = prctile(X,p,dim)` returns percentiles along dimension `dim`.

### Examples

#### Percentiles of a Data Vector

Generate a data set of size 10.

```
rng('default'); % for reproducibility
x = normrnd(5,2,1,10)
```

`x =`

Columns 1 through 7

```
6.0753    8.6678    0.4823    6.7243    5.6375    2.3846    4.1328
```

Columns 8 through 10

```
5.6852    12.1568    10.5389
```

Calculate the 42nd percentile.

```
Y = prctile(x,42)
```

```
Y =
```

```
5.6709
```

### Percentiles of a Data Matrix

Calculate the percentiles along the columns and rows of a data matrix for specified percentages.

Generate a 5-by-5 data matrix.

```
X = (1:5)'*(2:6)
```

```
X =
```

```
2     3     4     5     6
4     6     8    10    12
6     9    12    15    18
8    12    16    20    24
10   15    20    25    30
```

Calculate the 25th, 50th, and 75th percentiles along the columns of X.

```
Y = prctile(X,[25 50 75],1)
```

```
Y =  
  
    3.5000    5.2500    7.0000    8.7500   10.5000  
    6.0000    9.0000   12.0000   15.0000   18.0000  
    8.5000   12.7500   17.0000   21.2500   25.5000
```

The rows of  $Y$  correspond to the percentiles of columns of  $X$ . For example, the 25th, 50th, and 75th percentiles of the third column of  $X$  with elements (4, 8, 12, 16, 20) are 7, 12, and 17, respectively. `Y = prctile(X, [25 50 75])` returns the same percentile matrix.

Calculate the 25th, 50th, and 75th percentiles along the rows of  $X$ .

```
Y = prctile(X, [25 50 75], 2)
```

```
Y =  
  
    2.7500    4.0000    5.2500  
    5.5000    8.0000   10.5000  
    8.2500   12.0000   15.7500  
   11.0000   16.0000   21.0000  
   13.7500   20.0000   26.2500
```

The rows of  $Y$  correspond to the percentiles of rows of  $X$ . For example, the 25th, 50th, and 75th percentiles of the first row of  $X$  with elements (2, 3, 4, 5, 6) are 2.75, 4, and 5.25, respectively.

## Input Arguments

### **x** — Input data

vector | array

Input data, specified as a vector or array.

Data Types: double | single

### **p** — Percentages

scalar | vector

Percentages for which to compute percentiles, returned as a scalar or vector of scalars from 0 to 100.

Example: 25

Example: [25, 50, 75]

Data Types: `double` | `single`

### **dim** — Dimension

1 (default) | positive integer

Dimension along which the percentiles of  $X$  are required, specified as a positive integer. For example, for a matrix  $X$ , when  $\text{dim} = 1$ , `prctile` returns the quantile(s) of the columns of  $X$  and when  $\text{dim} = 2$ , `prctile` returns the quantile(s) of the rows of  $X$ . For a multidimensional array  $X$ , the length of the  $\text{dim}$ th dimension of  $Y$  is equal to the length of  $p$ .

Data Types: `double`

## Output Arguments

### **Y** — Percentiles

scalar | array

Percentiles of a data vector or array, specified as a scalar or array for one or more percentage values.

- If  $X$  is a vector, then  $Y$  is a scalar or a vector with the same length as the number of percentiles required (`length(p)`).  $Y(i)$  contains the  $p(i)$ th percentile.
- If  $X$  is a matrix, then  $Y$  is a vector or a matrix with the length of the  $\text{dim}$ th dimension equal to the number percentiles required (`length(p)`). When  $\text{dim} = 1$ , for example, the  $i$ th row of  $Y$  contains the  $p(i)$ th percentiles of columns of  $X$ .
- If  $X$  is an array of dimension  $d$ , then  $Y$  is an array with the length of the  $\text{dim}$ th dimension equal to the number of percentiles required (`length(p)`).

## Definitions

### Multidimensional Array

A *multidimensional array* is an array with more than two dimensions. For example, if  $X$  is a 1-by-3-by-4 array, then  $X$  is a 3-D array.

### Nonsingleton Dimension

A *first nonsingleton dimension* is the first dimension of an array whose size is not equal to 1. For example, if  $X$  is a 1-by-2-by-3-by-4 array, then the second dimension is the first nonsingleton dimension of  $X$ .

### Linear Interpolation

Linear interpolation uses linear polynomials to find  $y_i = f(x_i)$ , the values of the underlying function  $Y = f(X)$  at the points in the vector or array  $x$ . Given the data points  $(x_1, y_1)$  and  $(x_2, y_2)$ , where  $y_1 = f(x_1)$  and  $y_2 = f(x_2)$ , linear interpolation finds  $y = f(x)$  for a given  $x$  between  $x_1$  and  $x_2$  as follows:

$$y = f(x) = y_1 + \frac{(x - x_1)}{(x_2 - x_1)}(y_2 - y_1).$$

Similarly, if the 100(1.5/ $n$ )th percentile is  $y_{1.5/n}$  and the 100(2.5/ $n$ )th percentile is  $y_{2.5/n}$ , then linear interpolation finds the 100(2.3/ $n$ )th percentile,  $y_{2.3/n}$  as:

$$y_{\frac{2.3}{n}} = y_{\frac{1.5}{n}} + \frac{\left(\frac{2.3}{n} - \frac{1.5}{n}\right)}{\left(\frac{2.5}{n} - \frac{1.5}{n}\right)} \left( y_{\frac{2.5}{n}} - y_{\frac{1.5}{n}} \right).$$

## Algorithms

For an  $n$ -element vector  $X$ , `prctile` returns percentiles as follows:

- 1 The sorted values in  $X$  are taken as the 100(0.5/ $n$ )th, 100(1.5/ $n$ )th, ..., 100( $[n - 0.5]/n$ )th percentiles. For example:



- For a data vector of five elements such as {6, 3, 2, 10, 1}, the sorted elements {1, 2, 3, 6, 10} respectively correspond to the 10th, 30th, 50th, 70th, and 90th percentiles.
  - For a data vector of six elements such as {6, 3, 2, 10, 8, 1}, the sorted elements {1, 2, 3, 6, 8, 10} respectively correspond to the (50/6)th, (150/6)th, (250/6)th, (350/6)th, (450/6)th, and (550/6)th percentiles.
- 2 `prctile` uses linear interpolation on page 32-4890 to compute percentiles for percentages between  $100(0.5/n)$  and  $100([n - 0.5]/n)$ .
  - 3 `prctile` assigns the minimum or maximum values in `X` to the percentiles corresponding to the percentages outside that range.

`prctile` treats NaNs as missing values and removes them.

## References

- [1] Langford, E. “Quartiles in Elementary Statistics”, *Journal of Statistics Education*. Vol. 14, No. 3, 2006.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- See “Automatic dimension restriction” (MATLAB Coder).
- If the output `Y` is a vector, the orientation of `Y` differs from MATLAB when all of the following are true:
  - You do not supply the `dim` input.
  - `X` is a variable-size array.
  - `X` is not a variable-length vector.
  - `X` is a vector at run time.

- The orientation of the vector  $x$  does not match the orientation of the vector  $p$ .

In this case, the output  $Y$  matches the orientation of  $x$ , not the orientation of  $p$ .

## See Also

`iqr` | `median` | `quantile`

## Topics

“Quantiles and Percentiles” on page 3-7

**Introduced before R2006a**

# predict

**Class:** ClassificationKNN

Predict labels using *k*-nearest neighbor classification model

## Syntax

```
label = predict(Mdl,X)
[label,score,cost] = predict(Mdl,X)
```

## Description

`label = predict(Mdl,X)` returns a vector of predicted class label on page 32-4897 for the predictor data in the table or matrix *X*, based on the trained *k*-nearest neighbor classification model *Mdl*.

`[label,score,cost] = predict(Mdl,X)` also returns:

- A matrix of classification scores (`score`) indicating the likelihood that a label comes from a particular class. For *k*-nearest neighbor, scores are posterior probabilities on page 32-4897.
- A matrix of expected classification cost on page 32-4898 (`cost`). For each observation in *X*, the predicted class label corresponds to the minimum expected classification costs among all classes.

## Input Arguments

***mdl*** — *k*-nearest neighbor classification model

ClassificationKNN model object

*k*-nearest neighbor classification model, specified as a ClassificationKNN model object returned by `fitcknn`.

**x — Predictor data to be classified**

numeric matrix | table

Predictor data to be classified, specified as a numeric matrix or table.

Each row of `X` corresponds to one observation, and each column corresponds to one variable.

- For a numeric matrix:
  - The variables making up the columns of `X` must have the same order as the predictor variables that trained `Mdl`.
  - If you trained `Mdl` using a table (for example, `Tbl`), then `X` can be a numeric matrix if `Tbl` contains all numeric predictor variables. To treat all numeric predictors in `Tbl` as categorical during training (*k*-nearest neighbors requires homogeneous predictors), set `CategoricalPredictors`, 'all' when you train using `fitcknn`. If `Tbl` contains heterogeneous predictors (for example, numeric and categorical data types) and `X` is a numeric matrix, then `predict` throws an error.
- For a table:
  - `predict` does not support multi-column variables and cell arrays other than cell arrays of character vectors.
  - If you trained `Mdl` using a table (for example, `Tbl`), then all predictor variables in `X` must have the same variable names and data types as those that trained `Mdl` (stored in `Mdl.PredictorNames`). However, the column order of `X` does not need to correspond to the column order of `Tbl`. `Tbl` and `X` can contain additional variables (response variables, observation weights, etc.), but `predict` ignores them.
  - If you trained `Mdl` using a numeric matrix, then the predictor names in `Mdl.PredictorNames` and corresponding predictor variable names in `X` must be the same. To specify predictor names during training, see the `PredictorNames` name-value pair argument of `fitcknn`. All predictor variables in `X` must be numeric vectors. `X` can contain additional variables (response variables, observation weights, etc.), but `predict` ignores them.

If you set 'Standardize', true in `fitcknn` to train `Mdl`, then the software standardizes the columns of `X` using the corresponding means in `Mdl.Mu` and standard deviations in `Mdl.Sigma`.

Data Types: `table` | `double` | `single`

## Output Arguments

### **label** — Predicted class labels

categorical array | character array | logical vector | vector of numeric values | cell array of character vectors

Predicted class labels for the observations (rows) in `X`, returned as a vector with length equal to the number of rows of `X`. The label is the class with minimal expected cost. See “Predicted Class Label” on page 32-4897.

### **score** — Predicted class scores or posterior probabilities

numeric column vector | numeric matrix

Predicted class scores or posterior probabilities, returned as a numeric matrix of size `N`-by-`K`. `N` is the number of observations (rows) in `X`, and `K` is the number of classes (in `Mdl.ClassNames`). `score(i,j)` is the posterior probability that observation `i` in `X` is of class `j` in `Mdl.ClassNames`. See “Posterior Probability” on page 32-4897.

### **cost** — Expected classification costs

numeric matrix

Expected costs, returned as a matrix of size `N`-by-`K`. `N` is the number of observations (rows) in `X`, and `K` is the number of classes (in `Mdl.ClassNames`). `cost(i,j)` is the cost of classifying row `i` of `X` as class `j` in `Mdl.ClassNames`. See “Expected Cost” on page 32-4898.

## Examples

### ***k*-Nearest Neighbor Classification Predictions**

Construct a  $k$ -nearest neighbor classifier for Fisher's iris data, where  $k = 5$ . Evaluate some model predictions on new data.

Load the data.

```
load fisheriris
X = meas;
Y = species;
```

Construct a classifier for 5-nearest neighbors. It is good practice to standardize non-categorical predictor data.

```
mdl = fitcknn(X,Y,'NumNeighbors',5,'Standardize',1);
```

Predict the classifications for flowers with minimum, mean, and maximum characteristics.

```
Xnew = [min(X);mean(X);max(X)];
[label,score,cost] = predict(mdl,Xnew)
```

```
label =
```

```
3x1 cell array
```

```
 {'versicolor'}
 {'versicolor'}
 {'virginica' }
```

```
score =
```

```
0.4000    0.6000         0
         0    1.0000         0
         0         0    1.0000
```

```
cost =
```

```
0.6000    0.4000    1.0000
1.0000         0    1.0000
1.0000    1.0000         0
```

The second and third rows of the score and cost matrices have binary values, meaning all five nearest neighbors of the mean and maximum flower measurements have identical classifications.

- “Predict Classification Using KNN Classifier” on page 18-38

## Definitions

### Predicted Class Label

`predict` classifies so as to minimize the expected classification cost:

$$\hat{y} = \arg \min_{y=1, \dots, K} \sum_{k=1}^K \hat{P}(k | x) C(y | k),$$

where

- $\hat{y}$  is the predicted classification.
- $K$  is the number of classes.
- $\hat{P}(k | x)$  is the posterior probability of class  $k$  for observation  $x$ .
- $C(y | k)$  is the cost of classifying an observation as  $y$  when its true class is  $k$ .

### Posterior Probability

For a vector (single query point)  $X_{\text{new}}$  and model `mdl`, let:

- $K$  be the number of nearest neighbors used in prediction, `mdl.NumNeighbors`
- `nbdd(mdl, Xnew)` be the  $K$  nearest neighbors to  $X_{\text{new}}$  in `mdl.X`
- `Y(nbd)` be the classifications of the points in `nbdd(mdl, Xnew)`, namely `mdl.Y(nbd)`
- `W(nbd)` be the weights of the points in `nbdd(mdl, Xnew)`
- `prior` be the priors of the classes in `mdl.Y`

If there is a vector of prior probabilities, then the observation weights `W` are normalized by class to sum to the priors. This might involve a calculation for the point  $X_{\text{new}}$ , because weights can depend on the distance from  $X_{\text{new}}$  to the points in `mdl.X`.

The posterior probability  $p(j | X_{\text{new}})$  is

$$p(j | X_{\text{new}}) = \frac{\sum_{i \in \text{nbdd}} W(i) 1_{Y(X(i)=j)}}{\sum_{i \in \text{nbdd}} W(i)}.$$

Here,  $1_{Y(X(i)=j)}$  means 1 when `mdl.Y(i) = j`, and 0 otherwise.

## True Misclassification Cost

There are two costs associated with KNN classification: the true misclassification cost per class, and the expected misclassification cost per observation.

You can set the true misclassification cost per class in the `Cost` name-value pair when you run `fitcknn`. `Cost(i, j)` is the cost of classifying an observation into class `j` if its true class is `i`. By default, `Cost(i, j)=1` if `i~j`, and `Cost(i, j)=0` if `i=j`. In other words, the cost is 0 for correct classification, and 1 for incorrect classification.

## Expected Cost

There are two costs associated with KNN classification: the true misclassification cost per class, and the expected misclassification cost per observation. The third output of `predict` is the expected misclassification cost per observation.

Suppose you have `Nobs` observations that you want to classify with a trained classifier `mdl`. Suppose you have `K` classes. You place the observations into a matrix `Xnew` with one observation per row. The command

```
[label, score, cost] = predict(mdl, Xnew)
```

returns, among other outputs, a `cost` matrix of size `Nobs-by-K`. Each row of the `cost` matrix contains the expected (average) cost of classifying the observation into each of the `K` classes. `cost(n, k)` is

$$\sum_{i=1}^K \hat{P}(i | X_{new(n)}) C(k | i),$$

where

- $K$  is the number of classes.
- $\hat{P}(i | X_{new(n)})$  is the posterior probability on page 32-4897 of class  $i$  for observation  $X_{new(n)}$ .



- $C(k | i)$  is the true misclassification cost on page 32-4898 of classifying an observation as  $k$  when its true class is  $i$ .

## Extended Capabilities

### Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays” (MATLAB).

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This table contains notes about arguments.

Argument	Notes and Limitations
Mdl	<ul style="list-style-type: none"> <li>• Save a trained model by using <code>saveCompactModel</code>, and then load the model by using <code>loadCompactModel</code> within a function that you declare.</li> <li>• A <code>ClassificationKNN</code> model object is a full object that does not have a corresponding compact object. <code>saveCompactModel</code> saves a compact version that does not include cross-validation parameters and hyperparameters.</li> <li>• Must be a compile-time constant; its value cannot change in the generated code.</li> <li>• For more details, see “Code Generation” on page 32-0 of the <code>ClassificationKNN</code> class.</li> </ul>

Argument	Notes and Limitations
X	<ul style="list-style-type: none"> <li>• Must be a single- or double-precision matrix and can be variable-sized. However, the number of columns in X must be <code>numel(Mdl.PredictorNames)</code>.</li> <li>• Rows and columns must correspond to observations and predictors, respectively.</li> </ul>

- For code generation, `ClassificationKNN.predict` uses `parfor` (by default) to create loops that run in parallel on supported shared-memory multicore platforms. If your compiler does not support the Open Multiprocessing (OpenMP) application interface or you disable OpenMP library, MATLAB Coder treats the `parfor`-loops as `for`-loops. To find supported compilers, see [http://www.mathworks.com/support/compilers/current\\_release/](http://www.mathworks.com/support/compilers/current_release/). To disable OpenMP library, specify the `EnableOpenMP` property of the `codegen` configuration object as `false`. For details, see `coder.CodeConfig`.

## See Also

`ClassificationKNN` | `fitcknn`

## Topics

“Predict Classification Using KNN Classifier” on page 18-38

“Classification Using Nearest Neighbors” on page 18-16

“Code Generation Workflows” on page 31-2

# predict

**Class:** ClassificationLinear

Predict labels for linear classification models

## Syntax

```
Label = predict(Mdl,X)
Label = predict(Mdl,X,Name,Value)
[Label,Score] = predict(____)
```

## Description

`Label = predict(Mdl,X)` returns predicted class labels for each observation in the predictor data `X` based on the trained, binary, linear classification model `Mdl`. `Label` contains class labels for each regularization strength in `Mdl`.

`Label = predict(Mdl,X,Name,Value)` returns predicted class labels with additional options specified by one or more `Name,Value` pair arguments. For example, you can specify that columns in the predictor data correspond to observations.

`[Label,Score] = predict(____)` also returns classification scores on page 32-4913 for both classes using any of the previous syntaxes. `Score` contains classification scores for each regularization strength in `Mdl`.

## Input Arguments

**Mdl** — Binary, linear classification model

ClassificationLinear model object

Binary, linear classification model, specified as a `ClassificationLinear` model object. You can create a `ClassificationLinear` model object using `fitclinear`.

**x — Predictor data**

full matrix | sparse matrix

Predictor data, specified as an  $n$ -by- $p$  full or sparse matrix. This orientation of  $x$  indicates that rows correspond to individual observations, and columns correspond to individual predictor variables.

---

**Note** If you orient your predictor matrix so that observations correspond to columns and specify 'ObservationsIn', 'columns', then you might experience a significant reduction in computation time.

---

Data Types: single | double

**Name-Value Pair Arguments**

Specify optional comma-separated pairs of Name, Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as Name1, Value1, ..., NameN, ValueN.

**ObservationsIn — Predictor data observation dimension**

'rows' (default) | 'columns'

Predictor data observation dimension, specified as the comma-separated pair consisting of 'ObservationsIn' and 'columns' or 'rows'.

---

**Note** If you orient your predictor matrix so that observations correspond to columns and specify 'ObservationsIn', 'columns', then you might experience a significant reduction in optimization-execution time.

---

Data Types: char

## Output Arguments

### **Label** — Predicted class labels

categorical array | character array | logical matrix | numeric matrix | cell array of character vectors

Predicted class labels, returned as a categorical or character array, logical or numeric matrix, or cell array of character vectors.

In most cases, `Label` is an  $n$ -by- $L$  array of the same data type as the observed class labels (`Y`) used to train `Mdl`.  $n$  is the number of observations in `X` and  $L$  is the number of regularization strengths in `Mdl.Lambda`. That is, `Label(i, j)` is the predicted class label for observation  $i$  using the linear classification model that has regularization strength `Mdl.Lambda(j)`.

If `Y` is a character array and  $L > 1$ , then `Label` is a cell array of class labels.

### **Score** — Classification scores

numeric array

Classification scores on page 32-4913, returned as a  $n$ -by-2-by- $L$  numeric array.  $n$  is the number of observations in `X` and  $L$  is the number of regularization strengths in `Mdl.Lambda`. `Score(i, k, j)` is the score for classifying observation  $i$  into class  $k$  using the linear classification model that has regularization strength `Mdl.Lambda(j)`. `Mdl.ClassNames` stores the order of the classes.

If `Mdl.Learner` is 'logistic', then classification scores are posterior probabilities.

## Examples

### **Predict Training-Sample Labels**

Load the NLP data set.

```
load nlpdata
```

`X` is a sparse matrix of predictor data, and `Y` is a categorical vector of class labels. There are more than two classes in the data.

The models should identify whether the word counts in a web page are from the Statistics and Machine Learning Toolbox™ documentation. So, identify the labels that correspond to the Statistics and Machine Learning Toolbox™ documentation web pages.

```
Ystats = Y == 'stats';
```

Train a binary, linear classification model using the entire data set, which can identify whether the word counts in a documentation web page are from the Statistics and Machine Learning Toolbox™ documentation.

```
rng(1); % For reproducibility  
Mdl = fitclinear(X,Ystats);
```

Mdl is a `ClassificationLinear` model.

Predict the training-sample, or resubstitution, labels.

```
label = predict(Mdl,X);
```

Because there is one regularization strength in Mdl, label is column vectors with lengths equal to the number of observations.

Construct a confusion matrix.

```
ConfusionTrain = confusionmat(Ystats,label)
```

```
ConfusionTrain =  
  
    30018     0  
         1    1553
```

The model misclassifies only one 'stats' documentation page as being outside of the Statistics and Machine Learning Toolbox documentation.

### **Predict Test-Sample Labels**

Load the NLP data set and preprocess it as in “Predict Training-Sample Labels” on page 32-4903. Transpose the predictor data matrix.

```
load nlpdata
Ystats = Y == 'stats';
X = X';
```

Train a binary, linear classification model that can identify whether the word counts in a documentation web page are from the Statistics and Machine Learning Toolbox™ documentation. Specify to hold out 30% of the observations. Optimize the objective function using SpARSA.

```
rng(1); % For reproducibility
CVMdl = fitclinear(X, Ystats, 'Solver', 'sparsa', 'Holdout', 0.30, ...
    'ObservationsIn', 'columns');
Mdl = CVMdl.Trained{1};
```

CVMdl is a ClassificationPartitionedLinear model. It contains the property Trained, which is a 1-by-1 cell array holding a ClassificationLinear model that the software trained using the training set.

Extract the training and test data from the partition definition.

```
trainIdx = training(CVMdl.Partition);
testIdx = test(CVMdl.Partition);
```

Predict the training- and test-sample labels.

```
labelTrain = predict(Mdl, X(:, trainIdx), 'ObservationsIn', 'columns');
labelTest = predict(Mdl, X(:, testIdx), 'ObservationsIn', 'columns');
```

Because there is one regularization strength in Mdl, labelTrain and labelTest are column vectors with lengths equal to the number of training and test observations, respectively.

Construct a confusion matrix for the training data.

```
ConfusionTrain = confusionmat(Ystats(trainIdx), labelTrain)
```

```
ConfusionTrain =

    21013     0
         3    1085
```

The model misclassifies only three documentation pages as being outside of Statistics and Machine Learning Toolbox documentation.

Construct a confusion matrix for the test data.

```
ConfusionTest = confusionmat(Ystats(testIdx),labelTest)
```

```
ConfusionTest =  
  
      9003      2  
      3      463
```

The model misclassifies three documentation pages as being outside the Statistics and Machine Learning Toolbox, and two pages as being inside.

### Estimate Posterior Class Probabilities

Estimate test-sample, posterior class probabilities, and determine the quality of the model by plotting a ROC curve. Linear classification models return posterior probabilities for logistic regression learners only.

Load the NLP data set and preprocess it as in “Predict Test-Sample Labels” on page 32-4904.

```
load nlpdata  
Ystats = Y == 'stats';  
X = X';
```

Randomly partition the data into training and test sets by specifying a 30% holdout sample. Identify the test-set indices.

```
cvp = cvpartition(Ystats, 'Holdout', 0.30);  
idxTest = test(cvp);
```

Train a binary linear classification model. Fit logistic regression learners using SpARSA. To hold out the test set, specify the partitioned model.

```
CVMdl = fitlinear(X, Ystats, 'ObservationsIn', 'columns', 'CVPPartition', cvp, ...  
    'Learner', 'logistic', 'Solver', 'sparsa');  
Mdl = CVMdl.Trained{1};
```

Mdl is a ClassificationLinear model trained using the training set specified in the partition cvp only.



Predict the test-sample posterior class probabilities.

```
[~,posterior] = predict(Mdl,X(:,idxTest),'ObservationsIn','columns');
```

Because there is one regularization strength in `Mdl`, `posterior` is a matrix with 2 columns and rows equal to the number of test-set observations. Column  $i$  contains posterior probabilities of `Mdl.ClassNames(i)` given a particular observation.

Obtain false and true positive rates, and estimate the AUC. Specify that the second class is the positive class.

```
[fpr,tpr,~,auc] = perfcurve(Ystats(idxTest),posterior(:,2),Mdl.ClassNames(2));  
auc
```

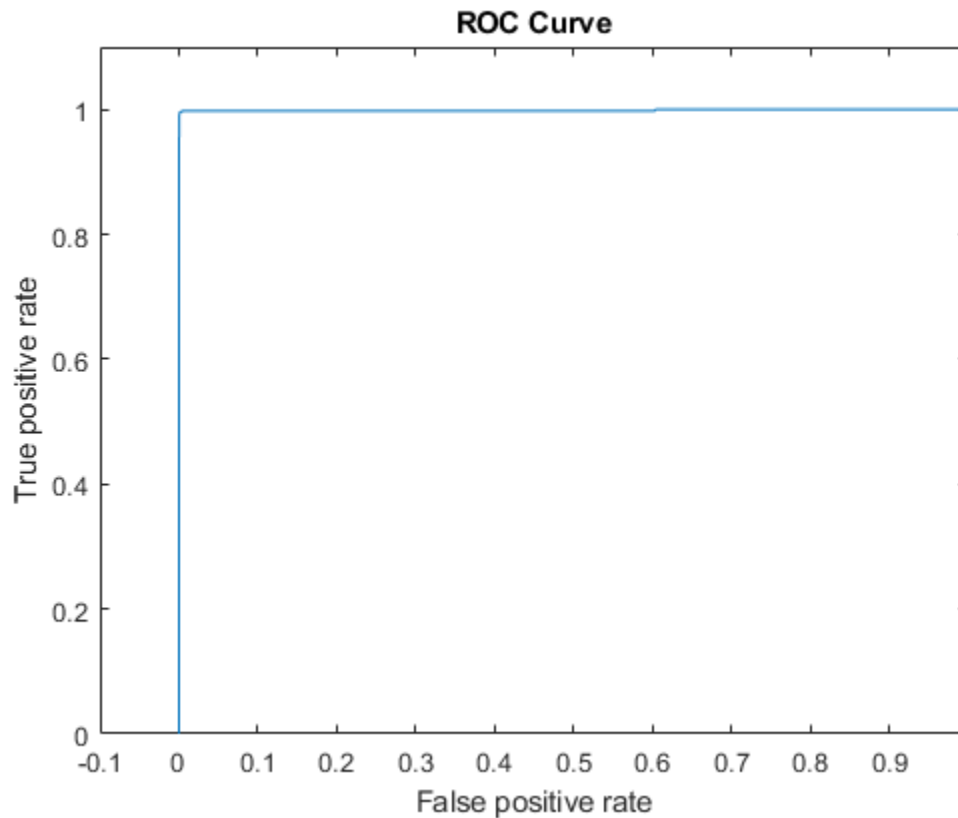
```
auc =
```

```
    0.9985
```

The AUC is 1, which indicates a model that predicts well.

Plot an ROC curve.

```
figure;  
plot(fpr,tpr)  
h = gca;  
h.XLim(1) = -0.1;  
h.YLim(2) = 1.1;  
xlabel('False positive rate')  
ylabel('True positive rate')  
title('ROC Curve')
```



The ROC curve and AUC indicate that the model classifies the test-sample observations almost perfectly.

### Find Good Lasso Penalty Using AUC

To determine a good lasso-penalty strength for a linear classification model that uses a logistic regression learner, compare test-sample values of the AUC.

Load the NLP data set. Preprocess the data as in “Predict Test-Sample Labels” on page 32-4904.

```
load nlpdata
Ystats = Y == 'stats';
X = X';
```

Create a data partition that specifies to holdout 10% of the observations. Extract test-sample indices.

```
rng(10); % For reproducibility
Partition = cvpartition(Ystats, 'Holdout', 0.10);
testIdx = test(Partition);
XTest = X(:, testIdx);
n = sum(testIdx)
YTest = Ystats(testIdx);
```

```
n =

    3157
```

There are 3157 observations in the test sample.

Create a set of 11 logarithmically-spaced regularization strengths from  $10^{-6}$  through  $10^{-0.5}$ .

```
Lambda = logspace(-6, -0.5, 11);
```

Train binary, linear classification models that use each of the regularization strengths. Optimize the objective function using SpARSA. Lower the tolerance on the gradient of the objective function to  $1e-8$ .

```
CVMD1 = fitclinear(X, Ystats, 'ObservationsIn', 'columns', ...
    'CVPartition', Partition, 'Learner', 'logistic', 'Solver', 'sparsa', ...
    'Regularization', 'lasso', 'Lambda', Lambda, 'GradientTolerance', 1e-8)
```

```
CVMD1 =

classreg.learning.partition.ClassificationPartitionedLinear
  CrossValidatedModel: 'Linear'
    ResponseName: 'Y'
  NumObservations: 31572
        KFold: 1
    Partition: [1x1 cvpartition]
```

```
ClassNames: [0 1]
ScoreTransform: 'none'
```

Extract the trained linear classification model.

```
Mdl1 = CVMdl.Trained{1}
```

```
Mdl1 =
```

```
ClassificationLinear
  ResponseName: 'Y'
  ClassNames: [0 1]
  ScoreTransform: 'logit'
  Beta: [34023x11 double]
  Bias: [1x11 double]
  Lambda: [1x11 double]
  Learner: 'logistic'
```

Mdl is a `ClassificationLinear` model object. Because `Lambda` is a sequence of regularization strengths, you can think of Mdl as 11 models, one for each regularization strength in `Lambda`.

Estimate the test-sample predicted labels and posterior class probabilities.

```
[label,posterior] = predict(Mdl1,XTest,'ObservationsIn','columns');
Mdl1.ClassNames;
posterior(3,1,5)
```

```
ans =
```

```
1.0000
```

`label` is a 3157-by-11 matrix of predicted labels. Each column corresponds to the predicted labels of the model trained using the corresponding regularization strength. `posterior` is a 3157-by-2-by-11 matrix of posterior class probabilities. Columns correspond to classes and pages correspond to regularization strengths. For example, `posterior(3,1,5)` indicates that the posterior probability that the first class (label 0)

is assigned to observation 3 by the model that uses  $\text{Lambda}(5)$  as a regularization strength is 1.0000.

For each model, compute the AUC. Designate the second class as the positive class.

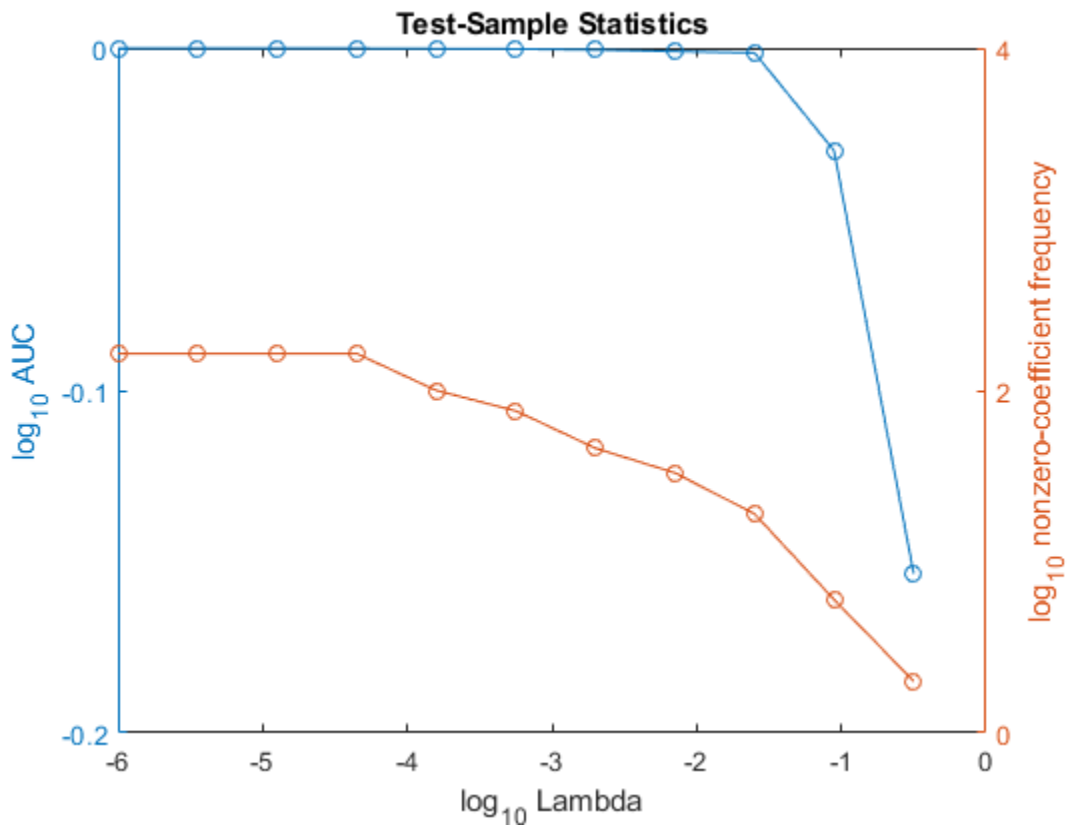
```
auc = 1:numel(Lambda); % Preallocation
for j = 1:numel(Lambda)
    [~,~,~,auc(j)] = perfcurve(YTest,posterior(:,2,j),Mdl1.ClassNames(2));
end
```

Higher values of  $\text{Lambda}$  lead to predictor variable sparsity, which is a good quality of a classifier. For each regularization strength, train a linear classification model using the entire data set and the same options as when you trained the model. Determine the number of nonzero coefficients per model.

```
Mdl = fitclinear(X,Ystats,'ObservationsIn','columns',...
    'Learner','logistic','Solver','sparsa','Regularization','lasso',...
    'Lambda',Lambda,'GradientTolerance',1e-8);
numNZCoeff = sum(Mdl.Beta~=0);
```

In the same figure, plot the test-sample error rates and frequency of nonzero coefficients for each regularization strength. Plot all variables on the log scale.

```
figure;
[h,hL1,hL2] = plotyy(log10(Lambda),log10(auc),...
    log10(Lambda),log10(numNZCoeff + 1));
hL1.Marker = 'o';
hL2.Marker = 'o';
ylabel(h(1),'log_{10} AUC')
ylabel(h(2),'log_{10} nonzero-coefficient frequency')
xlabel('log_{10} Lambda')
title('Test-Sample Statistics')
hold off
```



Choose the index of the regularization strength that balances predictor variable sparsity and high AUC. In this case, a value between  $10^{-2}$  to  $10^{-1}$  should suffice.

```
idxFinal = 9;
```

Select the model from `Mdl` with the chosen regularization strength.

```
MdlFinal = selectModels(Mdl,idxFinal);
```

`MdlFinal` is a `ClassificationLinear` model containing one regularization strength. To estimate labels for new observations, pass `MdlFinal` and the new data to `predict`.

## Definitions

### Classification Score

For linear classification models, the raw classification score for classifying the observation  $x$ , a row vector, into the positive class is defined by

$$f_j(x) = x\beta_j + b_j.$$

For the model with regularization strength  $j$ ,  $\beta_j$  is the estimated column vector of coefficients (the model property `Beta(:,j)`) and  $b_j$  is the estimated, scalar bias (the model property `Bias(j)`).

The raw classification score for classifying  $x$  into the negative class is  $-f(x)$ . The software classifies observations into the class that yields the positive score.

If the linear classification model consists of logistic regression learners, then the software applies the 'logit' score transformation to the raw classification scores (see `ScoreTransform`).

## Extended Capabilities

### Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays” (MATLAB).

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

Argument	Notes and Limitations
Mdl	<ul style="list-style-type: none"> <li>• Save a trained model by using <code>saveCompactModel</code>, and then load the model by using <code>loadCompactModel</code> within a function that you declare.</li> <li>• Must be a compile-time constant; its value cannot change in the generated code.</li> <li>• For more details, see “Code Generation” on page 32-0 of the <code>ClassificationLinear</code> class.</li> </ul>
X	<ul style="list-style-type: none"> <li>• Must be a single- or double-precision matrix and can be variable-sized.</li> <li>• If you specify <code>'ObservationsIn', 'rows'</code> (default), then the number of columns in X must be <code>numel(Mdl.PredictorNames)</code>. Rows and columns must correspond to observations and predictors, respectively.</li> <li>• If you specify <code>'ObservationsIn', 'columns'</code>, then the number of rows in X must be <code>numel(Mdl.PredictorNames)</code>. Rows and columns must correspond to predictors and observations, respectively.</li> </ul>
Name-value pair arguments	Names in name-value pair arguments must be compile-time constants. For example, to use the <code>'ObservationsIn'</code> name-value pair argument in the generated code, include <code>{coder.Constant('ObservationsIn'), coder.typeof('a', [1,7], [0,1])}</code> in the <code>-args</code> value of <code>codegen</code> .

## See Also

`ClassificationLinear` | `confusionmat` | `fitclinear` | `loss` | `perfcurve` | `testcholdout`

## Topics

“Code Generation Workflows” on page 31-2

Introduced in R2016a



# predict

**Class:** CompactClassificationDiscriminant

Predict labels using discriminant analysis classification model

## Syntax

```
label = predict(Mdl,X)
[label,score,cost] = predict(Mdl,X)
```

## Description

`label = predict(Mdl,X)` returns a vector of predicted class labels for the predictor data in the table or matrix `X`, based on the trained discriminant analysis classification model `Mdl`.

`[label,score,cost] = predict(Mdl,X)` also returns:

- A matrix of classification scores (`score`) indicating the likelihood that a label comes from a particular class. For discriminant analysis, scores are posterior probabilities on page 32-4920.
- A matrix of expected classification cost on page 32-4921 (`cost`). For each observation in `X`, the predicted class label corresponds to the minimum expected classification cost among all classes.

## Input Arguments

**mdl** — Discriminant analysis classification model

ClassificationDiscriminant model object |

CompactClassificationDiscriminant model object

Discriminant analysis classification model, specified as a ClassificationDiscriminant or CompactClassificationDiscriminant model object returned by `fitcdiscr`.

**x — Predictor data to be classified**

numeric matrix | table

Predictor data to be classified, specified as a numeric matrix or table.

Each row of `X` corresponds to one observation, and each column corresponds to one variable. All predictor variables in `X` must be numeric vectors.

- For a numeric matrix, the variables that compose the columns of `X` must have the same order as the predictor variables that trained `Mdl`.
- For a table:
  - `predict` does not support multi-column variables and cell arrays other than cell arrays of character vectors.
  - If you trained `Mdl` using a table (for example, `Tbl`), then all predictor variables in `X` must have the same variable names and data types as those that trained `Mdl` (stored in `Mdl.PredictorNames`). However, the column order of `X` does not need to correspond to the column order of `Tbl`. `Tbl` and `X` can contain additional variables (response variables, observation weights, etc.), but `predict` ignores them.
  - If you trained `Mdl` using a numeric matrix, then the predictor names in `Mdl.PredictorNames` and corresponding predictor variable names in `X` must be the same. To specify predictor names during training, see the `PredictorNames` name-value pair argument of `fitcdiscr`. `X` can contain additional variables (response variables, observation weights, etc.), but `predict` ignores them.

Data Types: `table` | `double` | `single`

## Output Arguments

**label — Predicted class labels**

categorical array | character array | logical vector | vector of numeric values | cell array of character vectors

Predicted class labels on page 32-4921, returned as a categorical or character array, logical or numeric vector, or cell array of character vectors.

label:

- Is the same data type as the observed class labels (Y) that trained `SVModel`
- Has length equal to the number of rows of X

### **score** — Predicted class posterior probabilities

numeric matrix

Predicted class posterior probabilities on page 32-4920, returned as a numeric matrix of size N-by-K. N is the number of observations (rows) in X, and K is the number of classes (in `Mdl.ClassNames`). `score(i,j)` is the posterior probability that observation i in X is of class j in `Mdl.ClassNames`.

### **cost** — Expected classification costs

numeric matrix

Expected classification costs on page 32-4921, returned as a matrix of size N-by-K. N is the number of observations (rows) in X, and K is the number of classes (in `Mdl.ClassNames`). `cost(i,j)` is the cost of classifying row i of X as class j in `Mdl.ClassNames`.

## Examples

### **Predict Class Labels Using Discriminant Analysis Model**

Load Fisher's iris data set. Determine the sample size.

```
load fisheriris
N = size(meas,1);
```

Partition the data into training and test sets. Hold out 10% of the data for testing.

```
rng(1); % For reproducibility
cvp = cvpartition(N, 'Holdout', 0.1);
idxTrn = training(cvp); % Training set indices
idxTest = test(cvp); % Test set indices
```

Store the training data in a table.

```
tblTrn = array2table(meas(idxTrn,:));
tblTrn.Y = species(idxTrn);
```

Train a discriminant analysis model using the training set and default options.

```
Mdl = fitcdiscr(tblTrn, 'Y');
```

Predict labels for the test set. You trained `Mdl` using a table of data, but you can predict labels using a matrix.

```
labels = predict(Mdl, meas(idxTest, :));
```

Construct a confusion matrix for the test set.

```
Mdl.ClassNames  
confusionmat(species(idxTest), labels)
```

```
ans =
```

```
3x1 cell array  
  
{'setosa' }  
{'versicolor'}  
{'virginica' }
```

```
ans =
```

```
8    0    0  
0    2    1  
0    0    4
```

`Mdl` misclassifies one versicolor iris as virginica in the test set.

### Plot Class Posterior Probability Regions

Load Fisher's iris data set. Consider training using the petal lengths and widths only.

```
load fisheriris  
X = meas(:, 3:4);
```

Train a quadratic discriminant analysis model using the entire data set.

```
Mdl = fitcdiscr(X, species, 'DiscrimType', 'quadratic');
```

Define a grid of values in the observed predictor space. Predict the posterior probabilities for each instance in the grid.

```
xMax = max(X);
xMin = min(X);
d = 0.01;
[x1Grid,x2Grid] = meshgrid(xMin(1):d:xMax(1),xMin(2):d:xMax(2));

[~,score] = predict(Mdl,[x1Grid(:),x2Grid(:)]);
Mdl.ClassNames
```

```
ans =
```

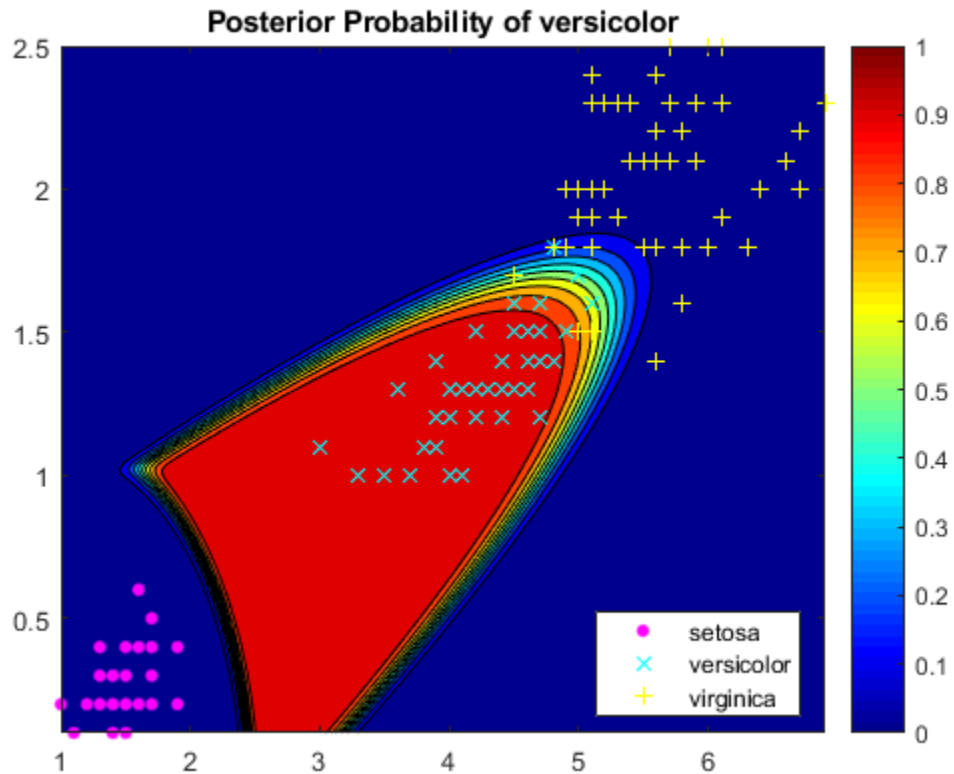
```
3x1 cell array

    {'setosa'    }
    {'versicolor'}
    {'virginica' }
```

score is a matrix of class posterior probabilities. The columns correspond to the classes in Mdl.ClassNames. For example, score(j,1) is the posterior probability that observation j is a setosa iris.

Plot the posterior probability of versicolor classification for each observation in the grid and plot the training data.

```
figure;
contourf(x1Grid,x2Grid,reshape(score(:,2),size(x1Grid,1),size(x1Grid,2)));
h = colorbar;
caxis([0 1]);
colormap jet;
hold on
gscatter(X(:,1),X(:,2),species,'mcy','.x+');
axis tight
title('Posterior Probability of versicolor');
hold off
```



The posterior probability region exposes a portion of the decision boundary.

## Definitions

### Posterior Probability

The posterior probability that a point  $z$  belongs to class  $j$  is the product of the prior probability and the multivariate normal density. The density function of the multivariate normal with mean  $\mu_j$  and covariance  $\Sigma_j$  at a point  $z$  is

$$P(x | k) = \frac{1}{(2\pi|\Sigma_k|)^{1/2}} \exp\left(-\frac{1}{2}(x - \mu_k)^T \Sigma_k^{-1} (x - \mu_k)\right),$$

where  $|\Sigma_k|$  is the determinant of  $\Sigma_k$ , and  $\Sigma_k^{-1}$  is the inverse matrix.

Let  $P(k)$  represent the prior probability of class  $k$ . Then the posterior probability that an observation  $x$  is of class  $k$  is

$$\hat{P}(k | x) = \frac{P(x | k)P(k)}{P(x)},$$

where  $P(x)$  is a normalization constant, the sum over  $k$  of  $P(x | k)P(k)$ .

## Prior Probability

The prior probability is one of three choices:

- 'uniform' — The prior probability of class  $k$  is one over the total number of classes.
- 'empirical' — The prior probability of class  $k$  is the number of training samples of class  $k$  divided by the total number of training samples.
- Custom — The prior probability of class  $k$  is the  $k$ th element of the prior vector. See `fitcdiscr`.

After creating a classification model (`Mdl`) you can set the prior using dot notation:

```
Mdl.Prior = v;
```

where `v` is a vector of positive elements representing the frequency with which each element occurs. You do not need to retrain the classifier when you set a new prior.

## Cost

The matrix of expected costs per observation is defined in “Cost” on page 20-9.

## Predicted Class Label

`predict` classifies so as to minimize the expected classification cost:

$$\hat{y} = \arg \min_{y=1,\dots,K} \sum_{k=1}^K \hat{P}(k | x) C(y | k),$$

where

- $\hat{y}$  is the predicted classification.
- $K$  is the number of classes.
- $\hat{P}(k | x)$  is the posterior probability on page 20-8 of class  $k$  for observation  $x$ .
- $C(y | k)$  is the cost on page 20-9 of classifying an observation as  $y$  when its true class is  $k$ .

## Extended Capabilities

### Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays” (MATLAB).

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:



Argument	Notes and Limitations
Mdl	<ul style="list-style-type: none"> <li>• Save a trained model by using <code>saveCompactModel</code>, and then load the model by using <code>loadCompactModel</code> within a function that you declare.</li> <li>• Must be a compile-time constant; its value cannot change in the generated code.</li> <li>• For more details, see “Code Generation” on page 32-0 of the <code>CompactClassificationDiscriminant</code> class.</li> </ul>
X	<ul style="list-style-type: none"> <li>• Must be a single- or double-precision matrix and can be variable-sized. However, the number of columns in X must be <code>numel(Mdl.PredictorNames)</code>.</li> <li>• Rows and columns must correspond to observations and predictors, respectively.</li> </ul>

## See Also

`ClassificationDiscriminant` | `CompactClassificationDiscriminant` | `edge` | `fitcdiscr` | `loss` | `margin`

## Topics

“Discriminant Analysis Classification” on page 20-2  
“Code Generation Workflows” on page 31-2

**Introduced in R2011b**

## predict

**Class:** CompactClassificationECOC

Predict labels using multiclass, error-correcting output codes model

### Syntax

```
label = predict(Mdl, X)
label = predict(Mdl, X, Name, Value)
[label, NegLoss, PBScore] = predict(____)
[label, NegLoss, PBScore, Posterior] = predict(____)
```

### Description

`label = predict(Mdl, X)` returns a vector of predicted class labels for the predictor data in the table or matrix `X`, based on the full or compact, trained, multiclass, error-correcting output code (ECOC) model `Mdl`.

`label = predict(Mdl, X, Name, Value)` uses additional options specified by one or more `Name, Value` pair arguments. For example, specify the posterior probability estimation method, decoding scheme, or verbosity level.

`[label, NegLoss, PBScore] = predict(____)` uses any of the input arguments in the previous syntaxes and additionally returns:

- An array of negated average binary loss on page 32-4941 per class (`NegLoss`). For each observation in `X`, `predict` assigns the label of the class yielding the largest negated average binary loss (or, equivalently, the smallest average binary loss).
- An array of positive-class scores (`PBScore`) for the observations classified by each binary learner.

`[label, NegLoss, PBScore, Posterior] = predict(____)` additionally returns posterior class probability estimates for observations (`Posterior`).

To obtain posterior class probabilities, you must set `'FitPosterior', 1` when training the ECOC model using `fitcecoc`. Otherwise, `predict` throws an error.

## Input Arguments

### **mdl** — Full or compact, multiclass ECOC model

`ClassificationECOC` model object | `CompactClassificationECOC` model object

Full or compact, multiclass ECOC model, specified as a `ClassificationECOC` or `CompactClassificationECOC` model object.

To create a full or compact ECOC model, see `ClassificationECOC` or `CompactClassificationECOC`.

### **x** — Predictor data to be classified

numeric matrix | table

Predictor data to be classified, specified as a numeric matrix or table.

Each row of `X` corresponds to one observation, and each column corresponds to one variable.

- For a numeric matrix:
  - The variables making up the columns of `X` must have the same order as the predictor variables that trained `Mdl`.
  - If you trained `Mdl` using a table (for example, `Tbl`), then `X` can be a numeric matrix if `Tbl` contains all numeric predictor variables. To treat numeric predictors in `Tbl` as categorical during training, identify categorical predictors using the `CategoricalPredictors` name-value pair argument of `fitcecoc`. If `Tbl` contains heterogeneous predictor variables (for example, numeric and categorical data types) and `X` is a numeric matrix, then `predict` throws an error.
- For a table:
  - `predict` does not support multicolumn variables and cell arrays other than cell arrays of character vectors.
  - If you trained `Mdl` using a table (for example, `Tbl`), then all predictor variables in `X` must have the same variable names and data types as those that trained `Mdl`.

(stored in `Mdl.PredictorNames`). However, the column order of `X` does not need to correspond to the column order of `Tbl`. `Tbl` and `X` can contain additional variables (response variables, observation weights, etc.), but `predict` ignores them.

- If you trained `Mdl` using a numeric matrix, then the predictor names in `Mdl.PredictorNames` and corresponding predictor variable names in `X` must be the same. To specify predictor names during training, see the `PredictorNames` name-value pair argument of `fitcecoc`. All predictor variables in `X` must be numeric vectors. `X` can contain additional variables (response variables, observation weights, etc.), but `predict` ignores them.

---

**Note** If `Mdl.BinaryLearners` contains linear classification models (that is, `ClassificationLinear` model objects), then you cannot specify sample data in a table. Instead, pass a full or sparse matrix of predictor data.

---

Data Types: `table` | `double` | `single`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### **BinaryLoss** — Binary learner loss function

'hamming' | 'linear' | 'logit' | 'exponential' | 'binodeviance' | 'hinge'  
| 'quadratic' | function handle

Binary learner loss function, specified as the comma-separated pair consisting of 'BinaryLoss' and a built-in, loss-function name or function handle.

- This table contains names and descriptions of the built-in functions, where  $y_j$  is a class label for a particular binary learner (in the set  $\{-1, 1, 0\}$ ),  $s_j$  is the score for observation  $j$ , and  $g(y_j, s_j)$  is the binary loss formula.

Value	Description	Score Domain	$g(y, s_j)$
'binodeviance'	Binomial deviance	$(-\infty, \infty)$	$\log[1 + \exp(-2y_j s_j)] / [2\log(2)]$
'exponential'	Exponential	$(-\infty, \infty)$	$\exp(-y_j s_j) / 2$
'hamming'	Hamming	$[0, 1]$ or $(-\infty, \infty)$	$[1 - \text{sign}(y_j s_j)] / 2$
'hinge'	Hinge	$(-\infty, \infty)$	$\max(0, 1 - y_j s_j) / 2$
'linear'	Linear	$(-\infty, \infty)$	$(1 - y_j s_j) / 2$
'logit'	Logistic	$(-\infty, \infty)$	$\log[1 + \exp(-y_j s_j)] / [2\log(2)]$
'quadratic'	Quadratic	$[0, 1]$	$[1 - y_j (2s_j - 1)]^2 / 2$

The software normalizes the binary losses such that the loss is 0.5 when  $y_j = 0$ . Also, the software calculates the mean binary loss for each class.

- For a custom binary loss function, e.g., `customFunction`, specify its function handle `'BinaryLoss', @customFunction`.

`customFunction` should have this form:

```
bLoss = customFunction(M, s)
```

where:

- $M$  is the  $K$ -by- $L$  coding matrix stored in `Mdl.CodingMatrix`.
- $s$  is the 1-by- $L$  row vector of classification scores.
- `bLoss` is the classification loss. This scalar aggregates the binary losses for every learner in a particular class. For example, you can use the mean binary loss to aggregate the loss over the learners for each class.
- $K$  is the number of classes.
- $L$  is the number of binary learners.

For an example of passing a custom binary loss function, see “Predict Test-Sample Labels of ECOC Models Using Custom Binary Loss Function” on page 32-4933.

By default, if all binary learners are:

- SVMs or linear classification models of SVM learners, then `BinaryLoss` is `'hinge'`
- Ensembles trained by `AdaboostM1` or `GentleBoost`, then `BinaryLoss` is `'exponential'`

- Ensembles trained by `LogitBoost`, then `BinaryLoss` is `'binodeviance'`
- Linear classification models of logistic regression learners, or you specify to predict class posterior probabilities (i.e., set `'FitPosterior', 1` in `fitcecoc`), then `BinaryLoss` is `'quadratic'`

Otherwise, the default `BinaryLoss` is `'hamming'`. To check the default value, use dot notation to display the `BinaryLoss` property of the trained model at the command line.

Example: `'BinaryLoss', 'binodeviance'`

Data Types: `char` | `function_handle`

#### **Decoding — Decoding scheme**

`'lossweighted'` (default) | `'lossbased'`

Decoding scheme that aggregates the binary losses on page 32-4941, specified as the comma-separated pair consisting of `'Decoding'` and `'lossweighted'` or `'lossbased'`.

Example: `'Decoding', 'lossbased'`

Data Types: `char`

#### **NumKLInitializations — Number of random initial values**

0 (default) | nonnegative integer

Number of random initial values for fitting posterior probabilities by Kullback-Leibler divergence minimization, specified as the comma-separated pair consisting of `'NumKLInitializations'` and a nonnegative integer.

If you do not request the fourth output argument (`Posterior`) and set `'PosteriorMethod', 'kl'` (the default), then the software ignores the value of `NumKLInitializations`.

For more details, see “Posterior Estimation Using Kullback-Leibler Divergence” on page 32-4943.

Example: `'NumKLInitializations', 5`

Data Types: `single` | `double`

#### **ObservationsIn — Predictor data observation dimension**

`'rows'` (default) | `'columns'`

Predictor data observation dimension, specified as the comma-separated pair consisting of 'ObservationsIn' and 'columns' or 'rows'. `Mdl.BinaryLearners` must contain linear classification models.

---

**Note** If you orient your predictor matrix so that observations correspond to columns and specify 'ObservationsIn', 'columns', you can experience a significant reduction in execution time.

---

Data Types: char

### Options — Estimation options

[] (default) | structure array returned by `statset`

Estimation options, specified as the comma-separated pair consisting of 'Options' and a structure array returned by `statset`.

To invoke parallel computing:

- You need a Parallel Computing Toolbox license.
- Specify 'Options', `statset('UseParallel',1)`.

### PosteriorMethod — Posterior probability estimation method

'kl' (default) | 'qp'

Posterior probability estimation method, specified as the comma-separated pair consisting of 'PosteriorMethod' and 'kl' or 'qp'.

- If `PosteriorMethod` is 'kl', then the software estimates multiclass posterior probabilities by minimizing the Kullback-Leibler divergence between the predicted and expected posterior probabilities returned by binary learners. For details, see “Posterior Estimation Using Kullback-Leibler Divergence” on page 32-4943.
- If `PosteriorMethod` is 'qp', then the software estimates multiclass posterior probabilities by solving a least-squares problem using quadratic programming. You need an Optimization Toolbox license to use this option. For details, see “Posterior Estimation Using Quadratic Programming” on page 32-4945.
- If you do not request the fourth output argument (`Posterior`), then the software ignores the value of `PosteriorMethod`.

Example: 'PosteriorMethod', 'qp'

Data Types: `char`

### **verbose** — Verbosity level

0 (default) | 1

Verbosity level, specified as the comma-separated pair consisting of `'Verbose'` and 0 or 1. `Verbose` controls the amount of diagnostic messages that the software displays in the Command Window.

If `Verbose` is 0, then the software does not display diagnostic messages. Otherwise, the software displays diagnostic messages.

Example: `'Verbose', 1`

Data Types: `single` | `double`

## Output Arguments

### **label** — Predicted class labels

categorical array | character array | logical array | numeric array | cell array of character vectors

Predicted class labels, returned as a categorical, character, logical, or numeric array, or cell array of character vectors. The software predicts the classification of an observation by assigning the observation to the class yielding the largest negated average binary loss (or, equivalently, the smallest average binary loss).

`label` is of the same data type as the class labels used to train `Mdl`. The number of rows of `label` and the number of observations in `X` are equal.

If `Mdl.BinaryLearners` contains linear classification models, then `label` is an  $m$ -by- $L$  matrix, where  $m$  is the number of observations in `X`, and  $L$  is the number of regularization strengths in the linear classification models (i.e., `numel(Mdl.BinaryLearners{1}.Lambda)`). `label(i, j)` is the predicted label of observation  $i$  for the model trained using regularization strength `Mdl.BinaryLearners{1}.Lambda(j)`.

Otherwise, `label` is a column vector of length  $m$ .

### **NegLoss** — Negated average binary losses

numeric matrix | numeric array



Negated, average binary losses, returned as a numeric matrix or array.

- If `Mdl.BinaryLearners` contains linear classification models, then `NegLoss` is a  $m$ -by- $K$ -by- $L$  array.
  - $m$  is the number of observations in  $X$ .
  - $K$  is the number of distinct classes in the training data (that is, `numel(Mdl.ClassNames)`).
  - $L$  is the number of regularization strengths in the linear classification models (that is, `numel(Mdl.BinaryLearners{1}.Lambda)`).

`NegLoss(i, k, j)` is the negated, average binary loss for observation  $i$ , corresponding to class `Mdl.ClassNames(k)`, for the model trained using regularization strength `Mdl.BinaryLearners{1}.Lambda(j)`.

- Otherwise, `NegLoss` is an  $m$ -by- $K$  matrix.

#### **PBScore — Positive-class scores**

numeric matrix | numeric array

Positive-class scores for each binary learner, returned as a numeric matrix or array.

- If `Mdl.BinaryLearners` contains linear classification models, then `PBScore` is an  $m$ -by- $B$ -by- $L$  array.
  - $m$  is the number of observations in  $X$ .
  - $B$  is the number of binary learners (that is, `numel(Mdl.BinaryLearners)`).
  - $L$  is the number of regularization strengths in the linear classification models (that is, `numel(Mdl.BinaryLearners{1}.Lambda)`).

`PBScore(i, b, j)` is the positive-class score for observation  $i$ , using binary learner  $b$ , for the model trained using regularization strength `Mdl.BinaryLearners{1}.Lambda(j)`.

- Otherwise, `PBScore` is an  $m$ -by- $B$  matrix.

#### **Posterior — Posterior class probabilities**

numeric matrix | numeric array

Posterior class probabilities, returned as a numeric matrix or array.

- If `Mdl.BinaryLearners` contains linear classification models, then `Posterior` is an  $m$ -by- $K$ -by- $L$  array. For dimension definitions, see `NegLoss`. `Posterior(i, k, j)` is the posterior probability that observation  $i$  comes from class `Mdl.ClassNames(k)`, for the model trained using regularization strength `Mdl.BinaryLearners{1}.Lambda(j)`.
- Otherwise, `Posterior` is an  $m$ -by- $K$  matrix.

## Examples

### Predict Test-Sample Labels of Training Data Using ECOC Models

Load Fisher's iris data set.

```
load fisheriris
X = meas;
Y = categorical(species);
classOrder = unique(Y);
rng(1); % For reproducibility
```

Train an ECOC model using SVM binary classifiers and specify a 30% holdout sample. It is good practice to define the class order. Specify to standardize the predictors using an SVM template.

```
t = templateSVM('Standardize',1);
CVMdl = fitcecoc(X,Y,'Holdout',0.30,'Learners',t,'ClassNames',classOrder);
CMdl = CVMdl.Trained{1}; % Extract trained, compact classifier
testInds = test(CVMdl.Partition); % Extract the test indices
XTest = X(testInds,:);
YTest = Y(testInds,:);
```

`CVMdl` is a `ClassificationPartitionedECOC` model. It contains the property `Trained`, which is a 1-by-1 cell array holding a `CompactClassificationECOC` model that the software trained using the training set.

Predict the test-sample labels. Print a random subset of true and predicted labels.

```
labels = predict(CMdl,XTest);
idx = randsample(sum(testInds),10);
table(YTest(idx),labels(idx),...
      'VariableNames',{'TrueLabels','PredictedLabels'})
```

```
ans =

10x2 table

   TrueLabels   PredictedLabels
   _____   _____

   setosa       setosa
   versicolor   virginica
   setosa       setosa
   virginica    virginica
   versicolor   versicolor
   setosa       setosa
   virginica    virginica
   virginica    virginica
   setosa       setosa
   setosa       setosa
```

Mdl correctly labeled all except one of the test-sample observations with indices idx.

## Predict Test-Sample Labels of ECOC Models Using Custom Binary Loss Function

Load Fisher's iris data set.

```
load fisheriris
X = meas;
Y = categorical(species);
classOrder = unique(Y); % Class order
K = numel(classOrder); %Number of classes
rng(1); % For reproducibility
```

Train an ECOC model using SVM binary classifiers and specify a 30% holdout sample. It is good practice to define the class order. Specify to standardize the predictors using an SVM template.

```
t = templateSVM('Standardize',1);
CVMdl = fitcecoc(X,Y,'Holdout',0.30,'Learners',t,'ClassNames',classOrder);
CMDL = CVMdl.Trained{1}; % Extract trained, compact classifier
testInds = test(CVMdl.Partition); % Extract the test indices
XTest = X(testInds,:);
YTest = Y(testInds,:);
```

CVMdl is a `ClassificationPartitionedECOC` model. It contains the property `Trained`, which is a 1-by-1 cell array holding a `CompactClassificationECOC` model that the software trained using the training set.

SVM scores are signed distances from the observation to the decision boundary. Therefore,  $(-\infty, \infty)$  is the domain. Create a custom binary loss function that:

- Maps the coding design matrix ( $M$ ) and positive-class classification scores ( $s$ ) for each learner to the binary loss for each observation
- Uses linear loss
- Aggregates the binary learner loss using the median.

You can create a separate function for the binary loss function, and then save it on the MATLAB® path. Or, you can specify an anonymous binary loss function.

```
customBL = @(M,s)nanmedian(1 - bsxfun(@times,M,s),2)/2;
```

Predict test-sample labels and estimate the median binary loss per class. Print the median negative binary losses per class for a random set of 10 test-sample observations.

```
[label,NegLoss] = predict(CMdl,XTest,'BinaryLoss',customBL);

idx = randsample(sum(testInds),10);
classOrder
table(YTest(idx),label(idx),NegLoss(idx,:), 'VariableNames',...
      {'TrueLabel','PredictedLabel','NegLoss'})

classOrder =

    3x1 categorical array

    setosa
    versicolor
    virginica

ans =

    10x3 table

    TrueLabel    PredictedLabel    NegLoss
```

setosa	versicolor	0.18578	1.9877	-3.6735
versicolor	virginica	-1.3315	-0.12341	-0.045053
setosa	versicolor	0.13889	1.9262	-3.5651
virginica	virginica	-1.513	-0.38289	0.39592
versicolor	versicolor	-0.8722	0.74788	-1.3757
setosa	versicolor	0.4841	1.9969	-3.981
virginica	virginica	-1.936	-0.67548	1.1114
virginica	virginica	-1.5786	-0.83374	0.91235
setosa	versicolor	0.51025	2.1205	-4.1308
setosa	versicolor	0.36126	2.0594	-3.9206

The order of the columns corresponds to the elements of `classOrder`. The software predicts the label based on the maximum negated loss. The results seem to indicate that the median of the linear losses might not perform as well as other losses.

### Estimate Posterior Probabilities Using ECOC Classifiers

Load Fisher's iris data set. Train the classifier using the petal dimensions as predictors.

```
load fisheriris
X = meas(:,3:4);
Y = species;
rng(1); % For reproducibility
```

Create an SVM template, and specify the Gaussian kernel. It is good practice to standardize the predictors.

```
t = templateSVM('Standardize',1,'KernelFunction','gaussian');
```

`t` is an SVM template. Most of its properties are empty. When the software trains the ECOC classifier, it sets the applicable properties to their default values.

Train the ECOC classifier using the SVM template. Transform classification scores to class posterior probabilities (which are returned by `predict` or `resubPredict`) using the `'FitPosterior'` name-value pair argument. Display diagnostic messages during the training using the `'Verbose'` name-value pair argument. It is good practice to specify the class order.

```
Mdl = fitcecoc(X,Y,'Learners',t,'FitPosterior',1,...
    'ClassNames',{'setosa','versicolor','virginica'},...
    'Verbose',2);

Training binary learner 1 (SVM) out of 3 with 50 negative and 50 positive observations.
Negative class indices: 2
Positive class indices: 1

Fitting posterior probabilities for learner 1 (SVM).
Training binary learner 2 (SVM) out of 3 with 50 negative and 50 positive observations.
Negative class indices: 3
Positive class indices: 1

Fitting posterior probabilities for learner 2 (SVM).
Training binary learner 3 (SVM) out of 3 with 50 negative and 50 positive observations.
Negative class indices: 3
Positive class indices: 2

Fitting posterior probabilities for learner 3 (SVM).
```

Mdl is a ClassificationECOC model. The same SVM template applies to each binary learner, but you can adjust options for each binary learner by passing in a cell vector of templates.

**Predict the in-sample labels and class posterior probabilities. Display diagnostic messages during the computation of labels and class posterior probabilities using the 'Verbose' name-value pair argument.**

```
[label,~,~,Posterior] = resubPredict(Mdl,'Verbose',1);
Mdl.BinaryLoss

Predictions from all learners have been computed.
Loss for all observations has been computed.
Computing posterior probabilities...

ans =

    'quadratic'
```

The software assigns an observation to the class that yields the smallest average binary loss. Since all binary learners are computing posterior probabilities, the binary loss function is quadratic.

Display a random set of results.

```

idx = randsample(size(X,1),10,1);
Mdl.ClassNames
table(Y(idx),label(idx),Posterior(idx,:),...
      'VariableNames',{'TrueLabel','PredLabel','Posterior'})

```

```
ans =
```

```

3x1 cell array

{'setosa'   }
{'versicolor'}
{'virginica' }

```

```
ans =
```

```

10x3 table

      TrueLabel      PredLabel      Posterior
      _____      _____      _____
'virginica'  'virginica'  0.0039321  0.0039869  0.99208
'virginica'  'virginica'  0.017067   0.018263   0.96467
'virginica'  'virginica'  0.014948   0.015856   0.9692
'versicolor' 'versicolor' 2.2197e-14  0.87317    0.12683
'setosa'     'setosa'     0.999      0.00025091 0.00074639
'versicolor' 'virginica'  2.2195e-14  0.059429   0.94057
'versicolor' 'versicolor' 2.2194e-14  0.97001    0.029986
'setosa'     'setosa'     0.999      0.0002499  0.00074741
'versicolor' 'versicolor' 0.0085646  0.98259    0.008849
'setosa'     'setosa'     0.999      0.00025013 0.00074718

```

The columns of Posterior correspond to the class order of Mdl.ClassNames.

Define a grid of values in the observed predictor space. Predict the posterior probabilities for each instance in the grid.

```

xMax = max(X);
xMin = min(X);

x1Pts = linspace(xMin(1),xMax(1));
x2Pts = linspace(xMin(2),xMax(2));
[x1Grid,x2Grid] = meshgrid(x1Pts,x2Pts);

```

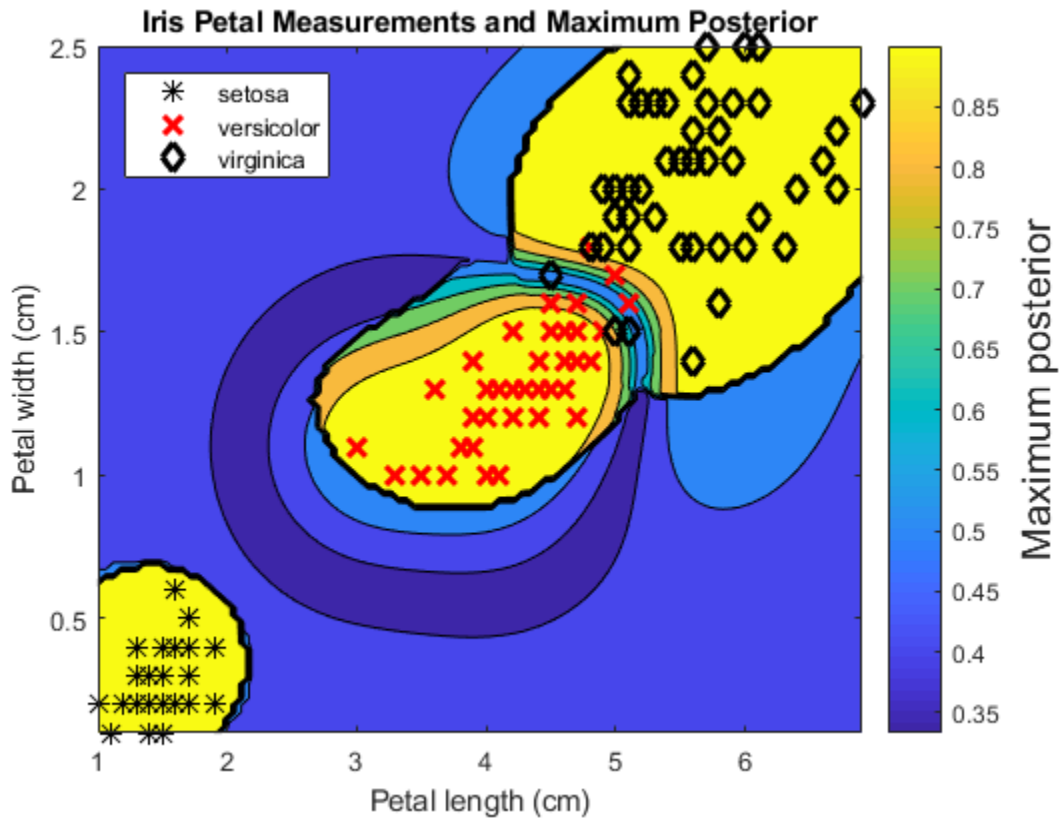
```
[~,~,~,PosteriorRegion] = predict(Mdl,[x1Grid(:),x2Grid(:)]);
```

For each coordinate on the grid, plot the maximum class posterior probability among all classes.

```
figure;
contourf(x1Grid,x2Grid,...
         reshape(max(PosteriorRegion,[],2),size(x1Grid,1),size(x1Grid,2)));
h = colorbar;
h.YLabel.String = 'Maximum posterior';
h.YLabel.FontSize = 15;
hold on
gh = gscatter(X(:,1),X(:,2),Y,'krk','*xd',8);
gh(2).LineWidth = 2;
gh(3).LineWidth = 2;

title 'Iris Petal Measurements and Maximum Posterior';
xlabel 'Petal length (cm)';
ylabel 'Petal width (cm)';
axis tight
legend(gh,'Location','NorthWest')
hold off
```





### Estimate Test-Sample Posterior Probabilities Using Parallel Computing

Train an error-correcting output codes, multiclass model and estimate posterior probabilities using parallel computing.

Load the `arrhythmia` data set. Determine the class distribution.

```
load arrhythmia
Y = categorical(Y);
tabulate(Y)
n = numel(Y);
K = numel(unique(Y));
```

Value	Count	Percent
1	245	54.20%
2	44	9.73%
3	15	3.32%
4	15	3.32%
5	13	2.88%
6	25	5.53%
7	3	0.66%
8	2	0.44%
9	9	1.99%
10	50	11.06%
14	4	0.88%
15	5	1.11%
16	22	4.87%

Several classes are not represented in the data, and many of the other classes have low relative frequencies.

Specify an ensemble learning template that uses the GentleBoost method and 50 weak, classification tree learners.

```
t = templateEnsemble('GentleBoost',50,'Tree');
```

`t` is a template object. Most of the options are empty (`[]`). The software uses default values for all empty options during training.

Since there are many classes, specify a sparse random coding design.

```
rng(1); % For reproducibility
Coding = designecoc(K,'sparserandom');
```

Train an ECOC model using parallel computing. Hold out 15% of the data and fit posterior probabilities.

```
pool = parpool; % Invokes workers
options = statset('UseParallel',1);
CVMdl = fitcecoc(X,Y,'Learner',t,'Options',options,'Coding',Coding,...
    'FitPosterior',1,'Holdout',0.15);

CMdl = CVMdl.Trained{1}; % Extract trained, compact classifier
testInds = test(CVMdl.Partition); % Extract the test indices
XTest = X(testInds,:);
YTest = Y(testInds,:);
```

```
Starting parallel pool parpool using the 'local' profile ... connected to 4 workers.
```

`CVMDL` is a `ClassificationPartitionedECOC` model. It contains the property `Trained`, which is a 1-by-1 cell array holding a `CompactClassificationECOC` model that the software trained using the training set.

The pool invokes four workers. The number of workers might vary among systems.

Estimate posterior probabilities, and display the posterior probability of being classified as not having arrhythmia (class 1) given the data for a random set of test-sample observations.

```
[~,~,~,posterior] = predict(CMDL,XTest,'Options',options);
idx = randsample(sum(testInds),10);
table(idx,YTest(idx),posterior(idx,1),...
      'VariableNames',{'TestSampleIndex','TrueLabel','PosteriorNoArrhythmia'})
```

ans =

TestSampleIndex	TrueLabel	PosteriorNoArrhythmia
11	6	0.60631
41	4	0.23674
51	2	0.13802
33	10	0.43831
12	1	0.94332
8	1	0.97278
37	1	0.62807
24	10	0.96876
56	16	0.29375
30	1	0.64512

- “Quick Start Parallel Computing for Statistics and Machine Learning Toolbox” on page 30-2

## Definitions

### Binary Loss

A binary loss is a function of the class and classification score that determines how well a binary learner classifies an observation into the class.

Let:

- $m_{kj}$  be element  $(k,j)$  of the coding design matrix  $M$  (i.e., the code corresponding to class  $k$  of binary learner  $j$ )
- $s_j$  be the score of binary learner  $j$  for an observation
- $g$  be the binary loss function
- $\hat{k}$  be the predicted class for the observation

In loss-based decoding [15], the class producing the minimum sum of the binary losses over binary learners determines the predicted class of an observation, that is,

$$\hat{k} = \operatorname{argmin}_k \sum_{j=1}^L |m_{kj}| g(m_{kj}, s_j).$$

In loss-weighted decoding [15], the class producing the minimum average of the binary losses over binary learners determines the predicted class of an observation, that is,

$$\hat{k} = \operatorname{argmin}_k \frac{\sum_{j=1}^L |m_{kj}| g(m_{kj}, s_j)}{\sum_{j=1}^L |m_{kj}|}.$$

Allwein et al. [2] suggest that loss-weighted decoding improves classification accuracy by keeping loss values for all classes in the same dynamic range.

This table summarizes the supported loss functions, where  $y_j$  is a class label for a particular binary learner (in the set  $\{-1,1,0\}$ ),  $s_j$  is the score for observation  $j$ , and  $g(y_j, s_j)$ .

Value	Description	Score Domain	$g(y_j, s_j)$
'binodeviance'	Binomial deviance	$(-\infty, \infty)$	$\log[1 + \exp(-2y_j s_j)] / [2\log(2)]$
'exponential'	Exponential	$(-\infty, \infty)$	$\exp(-y_j s_j) / 2$
'hamming'	Hamming	$[0,1]$ or $(-\infty, \infty)$	$[1 - \operatorname{sign}(y_j s_j)] / 2$
'hinge'	Hinge	$(-\infty, \infty)$	$\max(0, 1 - y_j s_j) / 2$
'linear'	Linear	$(-\infty, \infty)$	$(1 - y_j s_j) / 2$
'logit'	Logistic	$(-\infty, \infty)$	$\log[1 + \exp(-y_j s_j)] / [2\log(2)]$

Value	Description	Score Domain	$g(y_j, s_j)$
'quadratic'	Quadratic	[0,1]	$[1 - y_j(2s_j - 1)]^2/2$

The software normalizes the binary losses such that the loss is 0.5 when  $y_j = 0$ , and aggregates using the average of the binary learners [2].

Do not confuse the binary loss with the overall classification loss (specified by the `LossFun` name-value pair argument of `predict` and `loss`), e.g., classification error, which measures how well an ECOC classifier performs as a whole.

## Algorithms

- If you train `Mdl` specifying to standardize the predictor data, then the software standardizes the columns of `X` using the corresponding means and standard deviations that the software stored in `Mdl.BinaryLearner{j}.Mu` and `Mdl.BinaryLearner{j}.Sigma` for learner `j`.
- The software can estimate class posterior probabilities using quadratic programming or by minimizing the Kullback-Leibler divergence. For the following descriptions of the posterior estimation algorithms, let:
  - $m_{kj}$  be the element  $(k,j)$  of the coding design matrix  $M$ .
  - $I$  be the indicator function.
  - $\hat{p}_k$  be the class posterior probability estimate for class  $k$  of an observation,  $k = 1, \dots, K$ .
  - $r_j$  be the positive-class posterior probability for binary learner  $j$ . That is,  $r_j$  is the probability that binary learner  $j$  classifies an observation into the positive class, given the training data.

## Posterior Estimation Using Kullback-Leibler Divergence

By default, the software minimizes the Kullback-Leibler divergence to estimate class posterior probabilities. The Kullback-Leibler divergence between the expected and observed positive-class posterior probabilities is

$$\Delta(r, r) = \sum_{j=1}^L w_j \left[ r_j \log \frac{r_j}{r_j} + (1 - r_j) \log \frac{1 - r_j}{1 - r_j} \right],$$

where  $w_j = \sum_{S_j} w_i^*$  is the weight for binary learner  $j$  with  $S_j$  the set of observation indices that binary learner  $j$  is trained on and  $w_i^*$  is the weight of observation  $i$ . The software minimizes the divergence iteratively. The first step is to choose initial values

$\hat{p}_k^{(0)}$ ;  $k = 1, \dots, K$  for the class posterior probabilities.

- If you do not specify `NumKLIterations`, then the software uses both sets of deterministic initial values described next, and uses the one that minimizes  $\Delta$ .

- $\hat{p}_k^{(0)} = 1 / K$ ;  $k = 1, \dots, K$ .

- $\hat{p}_k^{(0)}$ ;  $k = 1, \dots, K$  is the solution of the system  
 $M_{01} \hat{p}^{(0)} = r$ ,

where  $M_{01}$  is  $M$  with all  $m_{kj} = -1$  replaced with 0, and  $r$  is a vector of positive-class posterior probabilities returned by the  $L$  binary learners [14]. The software uses `lsqnonneg` to solve the system.

- If you specify '`NumKLIterations`',  $c$ , where  $c$  is a natural number, then the

software does the following to choose  $\hat{p}_k^{(0)}$ ;  $k = 1, \dots, K$ , and uses the one that minimizes  $\Delta$ .

- The software chooses both sets of deterministic initial values as described previously.
- The software randomly generates  $c$  vectors of length  $K$  using `rand`, and then normalizes each vector to sum to 1.

At iteration  $t$ , the software:

- 1 Computes

$$\hat{r}_j^{(t)} = \frac{\sum_{k=1}^K \hat{p}_k^{(t)} I(m_{kj} = +1)}{\sum_{k=1}^K \hat{p}_k^{(t)} I(m_{kj} = +1 \cup m_{kj} = -1)}.$$

- 2 Estimates the next class posterior probability using

$$\hat{p}_k^{(t+1)} = \hat{p}_k^{(t)} \frac{\sum_{j=1}^L w_j [r_j I(m_{kj} = +1) + (1 - r_j) I(m_{kj} = -1)]}{\sum_{j=1}^L w_j [\hat{r}_j^{(t)} I(m_{kj} = +1) + (1 - \hat{r}_j^{(t)}) I(m_{kj} = -1)]}.$$

- 3 Normalizes  $\hat{p}_k^{(t+1)}$ ;  $k = 1, \dots, K$  so that they sum to 1.
- 4 Checks for convergence.

For more details, see [23] and [49].

## Posterior Estimation Using Quadratic Programming

Posterior probability estimation using quadratic programming requires an Optimization Toolbox license. To estimate posterior probabilities for an observation using this method, the software:

- 1 Estimates the positive-class posterior probabilities,  $r_j$ , for binary learners  $j = 1, \dots, L$ .
- 2 Using the relationship between  $r_j$  and  $\hat{p}_k$  [45], minimizes

$$\sum_{j=1}^L \left[ -r_j \sum_{k=1}^K \hat{p}_k I(m_{kj} = -1) + (1 - r_j) \sum_{k=1}^K \hat{p}_k I(m_{kj} = +1) \right]^2$$

with respect to  $\hat{p}_k$  and the restrictions

$$0 \leq \hat{p}_k \leq 1$$

$$\sum_k \hat{p}_k = 1.$$

The software performs minimization using quadprog.

## References

- [1] Allwein, E., R. Schapire, and Y. Singer. "Reducing multiclass to binary: A unifying approach for margin classifiers." *Journal of Machine Learning Research*. Vol. 1, 2000, pp. 113–141.

- [2] Dietterich, T., and G. Bakiri. “Solving Multiclass Learning Problems Via Error-Correcting Output Codes.” *Journal of Artificial Intelligence Research*. Vol. 2, 1995, pp. 263–286.
- [3] Escalera, S., O. Pujol, and P. Radeva. “On the decoding process in ternary error-correcting output codes.” *IEEE Transactions on Pattern Analysis and Machine Intelligence*. Vol. 32, Issue 7, 2010, pp. 120–134.
- [4] Escalera, S., O. Pujol, and P. Radeva. “Separability of ternary codes for sparse designs of error-correcting output codes.” *Pattern Recognition*. Vol. 30, Issue 3, 2009, pp. 285–297.
- [5] Hastie, T., and R. Tibshirani. “Classification by Pairwise Coupling.” *Annals of Statistics*. Vol. 26, Issue 2, 1998, pp. 451–471.
- [6] Wu, T. F., C. J. Lin, and R. Weng. “Probability Estimates for Multi-Class Classification by Pairwise Coupling.” *Journal of Machine Learning Research*. Vol. 5, 2004, pp. 975–1005.
- [7] Zadrozny, B. “Reducing Multiclass to Binary by Coupling Probability Estimates.” *NIPS 2001: Proceedings of Advances in Neural Information Processing Systems 14*, 2001, pp. 1041–1048.

## Extended Capabilities

### Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays” (MATLAB).

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:



Argument	Notes and Limitations
Mdl	<ul style="list-style-type: none"> <li>• Save a trained model by using <code>saveCompactModel</code>, and then load the model by using <code>loadCompactModel</code> within a function that you declare.</li> <li>• Must be a compile-time constant; its value cannot change in the generated code.</li> <li>• For more details, see “Code Generation” on page 32-0 of the <code>CompactClassificationECOC</code> class.</li> </ul>
X	<ul style="list-style-type: none"> <li>• Must be a single- or double-precision matrix and can be variable-sized.</li> <li>• If you specify <code>'ObservationsIn', 'rows'</code> (default), then the number of columns in X must be <code>numel(Mdl.PredictorNames)</code>. Rows and columns must correspond to observations and predictors, respectively.</li> <li>• If you specify <code>'ObservationsIn', 'columns'</code>, then the number of rows in X must be <code>numel(Mdl.PredictorNames)</code>. Rows and columns must correspond to predictors and observations, respectively.</li> </ul>
Posterior	You cannot return this output argument.
Name-value pair arguments	Names in name-value pair arguments must be compile-time constants. For example, to use the <code>'ObservationsIn'</code> name-value pair argument in the generated code, include <code>{coder.Constant('ObservationsIn'), coder.typeof('a', [1, 7], [0, 1])}</code> in the <code>-args</code> value of <code>codegen</code> .
BinaryLoss	Must be a supported binary loss function name, for example, <code>'linear'</code>
NumKLInitializations	You cannot specify this name-value pair argument.
Options	You cannot specify this name-value pair argument.
PosteriorMethod	You cannot specify this name-value pair argument.
Verbose	If you plan to generate a MEX file, then you can specify <code>Verbose</code> . Otherwise, <code>codegen</code> does not support <code>Verbose</code> for other file types.

## See Also

`ClassificationECOC` | `ClassificationLinear` | `CompactClassificationECOC` | `fitcecoc` | `quadprog` | `resubPredict` | `statset`

## Topics

“Quick Start Parallel Computing for Statistics and Machine Learning Toolbox” on page 30-2

“Reproducibility in Parallel Statistical Computations” on page 30-14

“Concepts of Parallel Computing in Statistics and Machine Learning Toolbox” on page 30-8

“Code Generation Workflows” on page 31-2

**Introduced in R2014b**

# predict

**Class:** CompactClassificationEnsemble

Predict labels using ensemble of classification models

## Syntax

```
labels = predict(Mdl,X)
labels = predict(Mdl,X,Name,Value)
[labels,score] = predict(____)
```

## Description

`labels = predict(Mdl,X)` returns a vector of predicted class labels for the predictor data in the table or matrix `X`, based on the full or compact, trained classification ensemble `Mdl`.

`labels = predict(Mdl,X,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

`[labels,score] = predict(____)` also returns a matrix of classification scores on page 32-4953 (`score`), indicating the likelihood that a label comes from a particular class, using any of the input arguments in the previous syntaxes. For each observation in `X`, the predicted class label corresponds to the maximum score among all classes.

## Input Arguments

### **Mdl**

A classification ensemble created by `fitcensemble` or a compact classification ensemble created by `compact`.

### **X**

Predictor data to be classified, specified as a numeric matrix or table.

Each row of `X` corresponds to one observation, and each column corresponds to one variable.

- For a numeric matrix:
  - The variables making up the columns of `X` must have the same order as the predictor variables that trained `Mdl`.
  - If you trained `Mdl` using a table (for example, `Tbl`), then `X` can be a numeric matrix if `Tbl` contains all numeric predictor variables. To treat numeric predictors in `Tbl` as categorical during training, identify categorical predictors using the `CategoricalPredictors` name-value pair argument of `fitcensemble`. If `Tbl` contains heterogeneous predictor variables (for example, numeric and categorical data types) and `X` is a numeric matrix, then `predict` throws an error.
- For a table:
  - `predict` does not support multi-column variables and cell arrays other than cell arrays of character vectors.
  - If you trained `Mdl` using a table (for example, `Tbl`), then all predictor variables in `X` must have the same variable names and be of the same data types as those that trained `Mdl` (stored in `Mdl.PredictorNames`). However, the column order of `X` does not need to correspond to the column order of `Tbl`. `Tbl` and `X` can contain additional variables (response variables, observation weights, etc.), but `predict` ignores them.
  - If you trained `Mdl` using a numeric matrix, then the predictor names in `Mdl.PredictorNames` and corresponding predictor variable names in `X` must be the same. To specify predictor names during training, see the `PredictorNames` name-value pair argument of `fitcensemble`. All predictor variables in `X` must be numeric vectors. `X` can contain additional variables (response variables, observation weights, etc.), but `predict` ignores them.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### Learners

Indices of weak learners `predict` uses for computation of responses, a numeric vector.

**Default:** `1:T`, where `T` is the number of weak learners in `Mdl`

### UseObsForLearner

A logical matrix of size `N`-by-`T`, where:

- `N` is the number of rows of `X`.
- `T` is the number of weak learners in `Mdl`.

When `UseObsForLearner(i, j)` is `true`, learner `j` is used in predicting the class of row `i` of `X`.

**Default:** `true(N, T)`

## Output Arguments

### labels

Vector of classification labels. `labels` has the same data type as the labels used in training `Mdl`.

### score

A matrix with one row per observation and one column per class. For each observation and each class, the score generated by each tree is the probability of this observation originating from this class computed as the fraction of observations of this class in a tree leaf. `predict` averages these scores over all trees in the ensemble.

## Examples

### Predict Class Labels Using Classification Ensemble

Load Fisher's iris data set. Determine the sample size.

```
load fisheriris
N = size(meas,1);
```

Partition the data into training and test sets. Hold out 10% of the data for testing.

```
rng(1); % For reproducibility
cvp = cvpartition(N, 'Holdout', 0.1);
idxTrn = training(cvp); % Training set indices
idxTest = test(cvp); % Test set indices
```

Store the training data in a table.

```
tblTrn = array2table(meas(idxTrn,:));
tblTrn.Y = species(idxTrn);
```

Boost 100 classification trees using AdaBoostM2 and the training set.

```
Mdl = fitensemble(tblTrn, 'Y', 'AdaBoostM2', 100, 'tree');
```

Predict labels for the test set. You trained model using a table of data, but you can predict labels using a matrix.

```
labels = predict(Mdl, meas(idxTest,:));
```

Construct a confusion matrix for the test set.

```
Mdl.ClassNames
confusionmat(species(idxTest), labels)
```

```
ans =
```

```
3x1 cell array

    {'setosa'    }
    {'versicolor'}
    {'virginica' }
```

```
ans =
```

```
      8      0      0
      0      2      1
      0      0      4
```

Mdl misclassifies one versicolor iris as virginica in the test set.

## Definitions

### Score (ensemble)

For ensembles, a classification score represents the confidence of a classification into a class. The higher the score, the higher the confidence.

Different ensemble algorithms have different definitions for their scores. Furthermore, the range of scores depends on ensemble type. For example:

- AdaBoostM1 scores range from  $-\infty$  to  $\infty$ .
- Bag scores range from 0 to 1.

## Extended Capabilities

### Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays” (MATLAB).

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

Argument	Notes and Limitations
Mdl	<ul style="list-style-type: none"> <li>• Save a trained model by using <code>saveCompactModel</code>, and then load the model by using <code>loadCompactModel</code> within a function that you declare.</li> <li>• Must be a compile-time constant; its value cannot change in the generated code.</li> <li>• For more details, see “Code Generation” on page 32-0 of the <code>CompactClassificationEnsemble</code> class.</li> </ul>
X	<ul style="list-style-type: none"> <li>• Must be a single- or double-precision matrix and can be variable-sized. However, the number of columns in X must be <code>numel(Mdl.PredictorNames)</code>.</li> <li>• Rows and columns must correspond to observations and predictors, respectively.</li> </ul>
Name-value pair arguments	Names in name-value pair arguments must be compile-time constants. For example, to allow user-defined indices up to 5 weak learners in the generated code, include <code>{coder.Constant('Learners'), coder.typeof(0, [1, 5], [0, 1])}</code> in the <code>-args</code> value of <code>codegen</code> .

## See Also

`ClassificationBaggedEnsemble` | `ClassificationEnsemble` | `CompactClassificationEnsemble` | `edge` | `fitcensemble` | `loss` | `margin`

## Topics

“Code Generation Workflows” on page 31-2

Introduced in R2011a



# predict

**Class:** CompactClassificationNaiveBayes

Predict labels using naive Bayes classification model

## Syntax

```
label = predict(Mdl,X)
[label,Posterior,Cost] = predict(Mdl,X)
```

## Description

`label = predict(Mdl,X)` returns a vector of predicted class labels for the predictor data in the table or matrix `X`, based on the trained, full or compact naive Bayes classifier `Mdl`.

`[label,Posterior,Cost] = predict(Mdl,X)` also returns:

- A matrix of posterior probabilities on page 32-4964 (`Posterior`) indicating the likelihood that a label comes from a particular class.
- A matrix of misclassification costs on page 32-4963 (`Cost`). For each observation in `X`, the predicted class label corresponds to the minimum expected classification costs among all classes.

## Input Arguments

**Mdl** — Naive Bayes classifier

ClassificationNaiveBayes model | CompactClassificationNaiveBayes model

Naive Bayes classifier, specified as a `ClassificationNaiveBayes` model or `CompactClassificationNaiveBayes` model returned by `fitcnb` or `compact`, respectively.

**x — Predictor data to be classified**

numeric matrix | table

Predictor data to be classified, specified as a numeric matrix or table.

Each row of `X` corresponds to one observation, and each column corresponds to one variable.

- For a numeric matrix:
  - The variables making up the columns of `X` must have the same order as the predictor variables that trained `Mdl`.
  - If you trained `Mdl` using a table (for example, `Tbl`), then `X` can be a numeric matrix if `Tbl` contains all numeric predictor variables. To treat numeric predictors in `Tbl` as categorical during training, identify categorical predictors using the `CategoricalPredictors` name-value pair argument of `fitcnb`. If `Tbl` contains heterogeneous predictor variables (for example, numeric and categorical data types) and `X` is a numeric matrix, then `predict` throws an error.
- For a table:
  - `predict` does not support multi-column variables and cell arrays other than cell arrays of character vectors.
  - If you trained `Mdl` using a table (for example, `Tbl`), then all predictor variables in `X` must have the same variable names and data types as those that trained `Mdl` (stored in `Mdl.PredictorNames`). However, the column order of `X` does not need to correspond to the column order of `Tbl`. `Tbl` and `X` can contain additional variables (response variables, observation weights, etc.), but `predict` ignores them.
  - If you trained `Mdl` using a numeric matrix, then the predictor names in `Mdl.PredictorNames` and corresponding predictor variable names in `X` must be the same. To specify predictor names during training, see the `PredictorNames` name-value pair argument of `fitcnb`. All predictor variables in `X` must be numeric vectors. `X` can contain additional variables (response variables, observation weights, etc.), but `predict` ignores them.

Data Types: table | double | single

---

**Notes:**

- If `Mdl.DistributionNames` is 'mn', then the software returns NaNs corresponding to rows of `X` containing at least one NaN.
- If `Mdl.DistributionNames` is not 'mn', then the software ignores NaN values when estimating misclassification costs and posterior probabilities. Specifically, the software computes the conditional density of the predictors given the class by leaving out the factors corresponding to missing predictor values.
- For predictor distribution specified as 'mvmn', if `X` contains levels that are not represented in the training data (i.e., not in `Mdl.CategoricalLevels` for that predictor), then the conditional density of the predictors given the class is 0. For those observations, the software returns the corresponding value of `Posterior` as a NaN. The software determines the class label for such observations using the class prior probability, stored in `Mdl.Prior`.

---

## Output Arguments

### **label** — Predicted class labels

categorical vector | character array | logical vector | numeric vector | cell array of character vectors

Predicted class labels, returned as a categorical vector, character array, logical or numeric vector, or cell array of character vectors.

label:

- Is the same data type as the observed class labels (`Mdl.Y`) that trained `Mdl`
- Has length equal to the number of rows of `Mdl.X`
- Is the class yielding the lowest expected misclassification cost (`Cost`)

### **Posterior** — Class posterior probabilities

numeric matrix

Class posterior probabilities on page 32-4964, returned as a numeric matrix. `Posterior` has rows equal to the number of rows of `X` and columns equal to the number of distinct classes in the training data (`size(Mdl.ClassNames,1)`).

`Posterior(j,k)` is the predicted posterior probability of class `k` (i.e., in class `Mdl.ClassNames(k)`) given the observation in row `j` of `X`.

Data Types: double

### **Cost** — Expected misclassification costs

numeric matrix

Expected misclassification costs on page 32-4963, returned as a numeric matrix. `Cost` has rows equal to the number of rows of `X` and columns equal to the number of distinct classes in the training data (`size(Mdl.ClassNames,1)`).

`Cost(j,k)` is the expected misclassification cost of the observation in row `j` of `X` being predicted into class `k` (i.e., in class `Mdl.ClassNames(k)`).

## Examples

### Label Test Sample Observations of Naive Bayes Classifiers

Load Fisher's iris data set.

```
load fisheriris
X = meas;      % Predictors
Y = species;  % Response
rng(1);
```

Train a naive Bayes classifier and specify to holdout 30% of the data for a test sample. It is good practice to specify the class order. Assume that each predictor is conditionally, normally distributed given its label.

```
CVMDL = fitcnb(X,Y,'Holdout',0.30,...
    'ClassNames',{'setosa','versicolor','virginica'});
CMDL = CVMDL.Trained{1};           % Extract trained, compact classifier
testIdx = test(CVMDL.Partition); % Extract the test indices
XTest = X(testIdx,:);
YTest = Y(testIdx);
```

`CVMDL` is a `ClassificationPartitionedModel` classifier. It contains the property `Trained`, which is a 1-by-1 cell array holding a `CompactClassificationNaiveBayes` classifier that the software trained using the training set.

Label the test sample observations. Display the results for a random set of 10 observations in the test sample.

```

idx = randsample(sum(testIdx),10);
label = predict(CMdl,XTest);
table(YTest(idx),label(idx),'VariableNames',...
      {'TrueLabel','PredictedLabel'})

```

```
ans =
```

```
10x2 table
```

TrueLabel	PredictedLabel
'setosa'	'setosa'
'versicolor'	'versicolor'
'setosa'	'setosa'
'virginica'	'virginica'
'versicolor'	'versicolor'
'setosa'	'setosa'
'virginica'	'virginica'
'virginica'	'virginica'
'setosa'	'setosa'
'setosa'	'setosa'

## Estimate Posterior Probabilities and Misclassification Costs

A goal of classification is to estimate posterior probabilities of new observations using a trained algorithm. Many applications train algorithms on large data sets, which can use resources that are better used elsewhere. This example shows how to efficiently estimate posterior probabilities of new observations using a Naive Bayes classifier.

Load Fisher's iris data set.

```

load fisheriris
X = meas; % Predictors
Y = species; % Response
rng(1);

```

Partition the data set into two sets: one in the training set, and the other is new unobserved data. Reserve 10 observations for the new data set.

```
n = size(X,1);
newInds = randsample(n,10);
inds = ~ismember(1:n,newInds);
XNew = X(newInds,:);
YNew = Y(newInds);
```

Train a naive Bayes classifier. It is good practice to specify the class order. Assume that each predictor is conditionally, normally distributed given its label. Conserve memory by reducing the size of the trained SVM classifier.

```
Mdl = fitcnb(X(inds,:),Y(inds),...
    'ClassNames',{'setosa','versicolor','virginica'});
CMdl = compact(Mdl);
whos('Mdl','CMdl')
```

Name	Size	Bytes	Class
CMdl	1x1	4994	classreg.learning.classif.CompactClassificationNaive
Mdl	1x1	12319	ClassificationNaiveBayes

The `CompactClassificationNaiveBayes` classifier (CMdl) uses less space than the `ClassificationNaiveBayes` classifier (Mdl) because the latter stores the data.

Predict the labels, posterior probabilities, and expected class misclassification costs. Since true labels are available, compare them with the predicted labels.

```
CMdl.ClassNames
[labels,PostProbs,MisClassCost] = predict(CMdl,XNew);
table(YNew,labels,PostProbs,'VariableNames',...
    {'TrueLabels','PredictedLabels',...
    'PosteriorProbabilities'})
MisClassCost
```

```
ans =
```

```
3x1 cell array

    {'setosa'      }
    {'versicolor' }
    {'virginica'   }
```

```
ans =
```

10x3 table

TrueLabels	PredictedLabels	PosteriorProbabilities		
'setosa'	'setosa'	1	4.1259e-16	1.1846e-23
'versicolor'	'versicolor'	1.0373e-60	0.99999	5.8053e-06
'virginica'	'virginica'	4.8708e-211	0.00085645	0.99914
'setosa'	'setosa'	1	1.4053e-19	2.2672e-26
'versicolor'	'versicolor'	2.9308e-75	0.99987	0.00012869
'setosa'	'setosa'	1	2.629e-18	4.4297e-25
'versicolor'	'versicolor'	1.4238e-67	0.99999	9.733e-06
'versicolor'	'versicolor'	2.0667e-110	0.94237	0.057625
'setosa'	'setosa'	1	4.3779e-19	3.5139e-26
'setosa'	'setosa'	1	1.1792e-17	2.2912e-24

MisClassCost =

0.0000	1.0000	1.0000
1.0000	0.0000	1.0000
1.0000	0.9991	0.0009
0.0000	1.0000	1.0000
1.0000	0.0001	0.9999
0.0000	1.0000	1.0000
1.0000	0.0000	1.0000
1.0000	0.0576	0.9424
0.0000	1.0000	1.0000
0.0000	1.0000	1.0000

PostProbs and MisClassCost are 15-by- 3 numeric matrices, where each row corresponds to a new observation and each column corresponds to a class. The order of the columns corresponds to the order of `CMdl.ClassNames`.

### Plot Posterior Probability Regions for Naive Bayes Classifiers

Load Fisher's iris data set. Train the classifier using the petal lengths and widths.

```
load fisheriris
X = meas(:,3:4);
Y = species;
```

Train a naive Bayes classifier. It is good practice to specify the class order. Assume that each predictor is conditionally, normally distributed given its label.

```
Mdl = fitcnb(X,Y,...  
    'ClassNames',{'setosa','versicolor','virginica'});
```

Mdl is a ClassificationNaiveBayes model. You can access its properties using dot notation.

Define a grid of values in the observed predictor space. Predict the posterior probabilities for each instance in the grid.

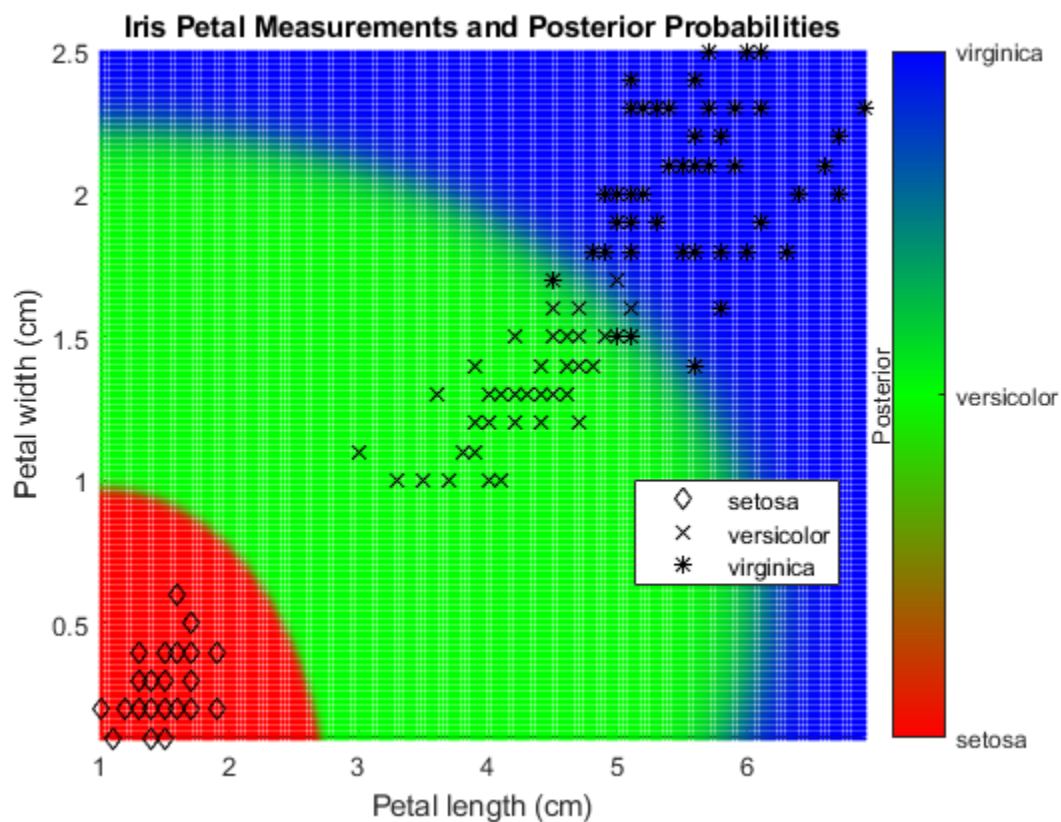
```
xMax = max(X);  
xMin = min(X);  
h = 0.01;  
[x1Grid,x2Grid] = meshgrid(xMin(1):h:xMax(1),xMin(2):h:xMax(2));  
  
[~,PosteriorRegion] = predict(Mdl,[x1Grid(:),x2Grid(:)]);
```

Plot the posterior probability regions and the training data.

```
figure;  
% Plot posterior regions  
scatter(x1Grid(:),x2Grid(:),1,PosteriorRegion);  
% Adjust color bar options  
h = colorbar;  
h.Ticks = [0 0.5 1];  
h.TickLabels = {'setosa','versicolor','virginica'};  
h.YLabel.String = 'Posterior';  
h.YLabel.Position = [-0.5 0.5 0];  
% Adjust color map options  
d = 1e-2;  
cmap = zeros(201,3);  
cmap(1:101,1) = 1:-d:0;  
cmap(1:201,2) = [0:d:1 1-d:-d:0];  
cmap(101:201,3) = 0:d:1;  
colormap(cmap);  
% Plot data  
hold on  
gh = gscatter(X(:,1),X(:,2),Y,'k','dx*');  
title 'Iris Petal Measurements and Posterior Probabilities';  
xlabel 'Petal length (cm)';  
ylabel 'Petal width (cm)';  
axis tight
```



```
legend(gh, 'Location', 'Best')
hold off
```



## Definitions

### Misclassification Cost

A misclassification cost is the relative severity of a classifier labeling an observation into the wrong class.

There are two types of misclassification costs: true and expected. Let  $K$  be the number of classes.

- True misclassification cost — A  $K$ -by- $K$  matrix, where element  $(i, j)$  indicates the misclassification cost of predicting an observation into class  $j$  if its true class is  $i$ . The software stores the misclassification cost in the property `Mdl.Cost`, and used in computations. By default, `Mdl.Cost(i, j) = 1` if  $i \neq j$ , and `Mdl.Cost(i, j) = 0` if  $i = j$ . In other words, the cost is 0 for correct classification, and 1 for any incorrect classification.
- Expected misclassification cost — A  $K$ -dimensional vector, where element  $k$  is the weighted average misclassification cost of classifying an observation into class  $k$ , weighted by the class posterior probabilities. In other words,

$$c_k = \sum_{j=1}^K \hat{P}(Y = j | x_1, \dots, x_P) \text{Cost}_{jk}.$$

the software classifies observations to the class corresponding with the lowest expected misclassification cost.

## Posterior Probability

The posterior probability is the probability that an observation belongs in a particular class, given the data.

For naive Bayes, the posterior probability that a classification is  $k$  for a given observation  $(x_1, \dots, x_P)$  is

$$\hat{P}(Y = k | x_1, \dots, x_P) = \frac{P(X_1, \dots, X_P | y = k) \pi(Y = k)}{P(X_1, \dots, X_P)},$$

where:

- $P(X_1, \dots, X_P | y = k)$  is the conditional joint density of the predictors given they are in class  $k$ . `Mdl.DistributionNames` stores the distribution names of the predictors.
- $\pi(Y = k)$  is the class prior probability distribution. `Mdl.Prior` stores the prior distribution.
- $P(X_1, \dots, X_P)$  is the joint density of the predictors. The classes are discrete, so

$$P(X_1, \dots, X_P) = \sum_{k=1}^K P(X_1, \dots, X_P | y = k) \pi(Y = k).$$

## Prior Probability

The prior probability is the believed relative frequency that observations from a class occur in the population for each class.

## References

[1] Hastie, T., R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*, Second Edition. NY: Springer, 2008.

## Extended Capabilities

### Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays” (MATLAB).

### See Also

`ClassificationNaiveBayes` | `CompactClassificationNaiveBayes` | `fitcnb` | `loss` | `resubPredict`

### Topics

“Naive Bayes Classification” on page 21-2

“Grouping Variables” on page 2-59

## predict

**Class:** CompactClassificationSVM

Predict labels using support vector machine classification model

## Syntax

```
label = predict(SVMModel, X)
[label, score] = predict(SVMModel, X)
```

## Description

`label = predict(SVMModel, X)` returns a vector of predicted class labels for the predictor data in the table or matrix `X`, based on the full or compact, trained SVM classification model `SVMModel`.

`[label, score] = predict(SVMModel, X)` also returns a matrix of scores (`score`), indicating the likelihood that a label comes from a particular class. For SVM, likelihood measures are either classification scores on page 32-4972 or class posterior probabilities on page 32-4973. For each observation in `X`, the predicted class label corresponds to the maximum score among all classes.

## Input Arguments

**SVMModel** — SVM classification model

ClassificationSVM model object | CompactClassificationSVM model object

SVM classification model, specified as a `ClassificationSVM` model object or `CompactClassificationSVM` model object returned by `fitcsvm` or `compact`, respectively.

**x** — Predictor data to be classified

numeric matrix | table

Predictor data to be classified, specified as a numeric matrix or table.

Each row of `X` corresponds to one observation, and each column corresponds to one variable.

- For a numeric matrix:
  - The variables making up the columns of `X` must have the same order as the predictor variables that trained `SVMMModel`.
  - If you trained `SVMMModel` using a table (for example, `Tbl`) and `Tbl` contains all numeric predictor variables, then `X` can be a numeric matrix. To treat numeric predictors in `Tbl` as categorical during training, identify categorical predictors using the `CategoricalPredictors` name-value pair argument of `fitcsvm`. If `Tbl` contains heterogeneous predictor variables (for example, numeric and categorical data types) and `X` is a numeric matrix, then `predict` throws an error.
- For a table:
  - `predict` does not support multicolumn variables and cell arrays other than cell arrays of character vectors.
  - If you trained `SVMMModel` using a table (for example, `Tbl`), then all predictor variables in `X` must have the same variable names and data types as those that trained `SVMMModel` (stored in `SVMMModel.PredictorNames`). However, the column order of `X` does not need to correspond to the column order of `Tbl`. `Tbl` and `X` can contain additional variables (response variables, observation weights, etc.), but `predict` ignores them.
  - If you trained `SVMMModel` using a numeric matrix, then the predictor names in `SVMMModel.PredictorNames` and corresponding predictor variable names in `X` must be the same. To specify predictor names during training, see the `PredictorNames` name-value pair argument of `fitcsvm`. All predictor variables in `X` must be numeric vectors. `X` can contain additional variables (response variables, observation weights, etc.), but `predict` ignores them.

If you set `'Standardize', true` in `fitcsvm` to train `SVMMModel`, then the software standardizes the columns of `X` using the corresponding means in `SVMMModel.Mu` and standard deviations in `SVMMModel.Sigma`.

Data Types: `table` | `double` | `single`

## Output Arguments

### **label** — Predicted class labels

categorical array | character array | logical vector | vector of numeric values | cell array of character vectors

Predicted class labels, returned as a categorical or character array, logical or numeric vector, or cell array of character vectors.

label:

- Is the same data type as the observed class labels (Y) that trained `SVMModel`
- Has length equal to the number of rows of X

For one-class learning, the elements of `label` are the one class represented in the observed class labels.

### **score** — Predicted class scores or posterior probabilities

numeric column vector | numeric matrix

Predicted class scores on page 32-4972 or posterior probabilities on page 32-4973, returned as a numeric column vector or numeric matrix.

- For one-class learning, `score` is a column vector with the same number of rows as the training observations (X). The elements are the positive class scores for the corresponding observations. You cannot obtain posterior probabilities for one-class learning.
- For two-class learning, `score` is a two-column matrix with the same number of rows as X.
  - If you fit the optimal score-to-posterior probability transformation function using `fitPosterior` or `fitSVMPosterior`, then `score` contains class posterior probabilities. That is, if the value of `SVMModel.ScoreTransform` is not `none`, then the elements of the first and second columns of `score` are the negative class (`SVMModel.ClassNames{1}`) and positive class (`SVMModel.ClassNames{2}`) posterior probabilities for the corresponding observations, respectively.
  - Otherwise, the elements of the first column are the negative class scores and the elements of the second column are the positive class scores for the corresponding observations.

If `SVMModel.KernelParameters.Function` is 'linear', then the software estimates the classification score for the observation  $x$  using

$$f(x) = (x/s)' \beta + b.$$

`SVMModel` stores  $\beta$ ,  $b$ ,  $s$  in the properties `Beta`, `Bias`, and `KernelParameters.Scale`, respectively.

## Examples

### Label Test Sample Observations of SVM Classifiers

Load the ionosphere data set.

```
load ionosphere
rng(1); % For reproducibility
```

Train an SVM classifier. Specify a 15% holdout sample for testing. It is good practice to specify the class order and standardize the data.

```
CVSVMModel = fitcsvm(X,Y,'Holdout',0.15,'ClassNames',{'b','g'},...
    'Standardize',true);
CompactSVMModel = CVSVMModel.Trained{1}; % Extract trained, compact classifier
testInds = test(CVSVMModel.Partition); % Extract the test indices
XTest = X(testInds,:);
YTest = Y(testInds,:);
```

`CVSVMModel` is a `ClassificationPartitionedModel` classifier. It contains the property `Trained`, which is a 1-by-1 cell array holding a `CompactClassificationSVM` classifier that the software trained using the training set.

Label the test sample observations. Display the results for the first 10 observations in the test sample.

```
[label,score] = predict(CompactSVMModel,XTest);
table(YTest(1:10),label(1:10),score(1:10,2),'VariableNames',...
    {'TrueLabel','PredictedLabel','Score'})
```

```
ans =
```

```
10x3 table
```

TrueLabel	PredictedLabel	Score
'b'	'b'	-1.7177
'g'	'g'	2.0003
'b'	'b'	-9.6841
'g'	'g'	2.5618
'b'	'b'	-1.548
'g'	'g'	2.0984
'b'	'b'	-2.7018
'b'	'b'	-0.66291
'g'	'g'	1.6046
'g'	'g'	1.7731

### Predict Labels and Posterior Probabilities of SVM Classifiers

A goal of classification is to predict labels of new observations using a trained algorithm. Many applications train algorithms on large data sets, which can use resources that are better used elsewhere. This example shows how to efficiently label new observations using an SVM classifier.

Load the ionosphere data set. Suppose that the last 10 observations become available after training the SVM classifier.

```
load ionosphere

n = size(X,1);           % Training sample size
isInds = 1:(n-10);     % In-sample indices
oosInds = (n-9):n;     % Out-of-sample indices
```

Train an SVM classifier. It is good practice to standardize the predictors and specify the order of the classes. Conserve memory by reducing the size of the trained SVM classifier.

```
SVMModel = fitcsvm(X(isInds,:),Y(isInds),'Standardize',true,...
    'ClassNames',{'b','g'});
CompactSVMModel = compact(SVMModel);
whos('SVMModel','CompactSVMModel')
```

Name	Size	Bytes	Class
CompactSVMModel	1x1	29936	classreg.learning.classif.CompactClassif



```
SVMModel          1x1          137268 ClassificationSVM
```

The positive class is 'g'. The CompactClassificationSVM classifier (CompactSVMModel) uses less space than the ClassificationSVM classifier (SVMModel) because the latter stores the data.

Estimate the optimal score-to-posterior-probability-transformation function.

```
CompactSVMModel = fitPosterior(CompactSVMModel, ...
    X(isInds,:), Y(isInds))
```

```
CompactSVMModel =
```

```
classreg.learning.classif.CompactClassificationSVM
    ResponseName: 'Y'
    CategoricalPredictors: []
        ClassNames: {'b' 'g'}
    ScoreTransform: '@(S) sigmoid(S,-1.968452e+00,3.121267e-01)'
        Alpha: [88x1 double]
        Bias: -0.2143
    KernelParameters: [1x1 struct]
        Mu: [1x34 double]
        Sigma: [1x34 double]
    SupportVectors: [88x34 double]
    SupportVectorLabels: [88x1 double]
```

The optimal score transformation function (CompactSVMModel.ScoreTransform) is the sigmoid function because the classes are inseparable.

Predict the out-of-sample labels and positive class posterior probabilities. Since true labels are available, compare them with the predicted labels.

```
[labels, PostProbs] = predict(CompactSVMModel, X(oosInds,:));
table(Y(oosInds), labels, PostProbs(:,2), 'VariableNames', ...
    {'TrueLabels', 'PredictedLabels', 'PosClassPosterior'})
```

```
ans =
```

```
10x3 table
```

TrueLabels	PredictedLabels	PosClassPosterior
'g'	'g'	0.98419
'g'	'g'	0.95545
'g'	'g'	0.67792
'g'	'g'	0.94448
'g'	'g'	0.98744
'g'	'g'	0.92482
'g'	'g'	0.97111
'g'	'g'	0.96986
'g'	'g'	0.97803
'g'	'g'	0.94361

`PostProbs` is a 10-by-2 matrix, where the first column is the negative class posterior probabilities, and the second column is the positive class posterior probabilities corresponding to the new observations.

- “Plot Posterior Probability Regions for SVM Classification Models” on page 18-218
- “Train SVM Classifier Using Custom Kernel” on page 18-199
- “Analyze Images Using Linear Support Vector Machines” on page 18-221

## Definitions

### Classification Score

The SVM classification score for classifying observation  $x$  is the signed distance from  $x$  to the decision boundary ranging from  $-\infty$  to  $+\infty$ . A positive score for a class indicates that  $x$  is predicted to be in that class. A negative score indicates otherwise.

The score for predicting  $x$  into the positive class, also the numerical, predicted response for  $x$ ,  $f(x)$ , is the trained SVM classification function

$$f(x) = \sum_{j=1}^n \alpha_j y_j G(x_j, x) + b,$$

where  $(\alpha_1, \dots, \alpha_n, b)$  are the estimated SVM parameters,  $G(x_j, x)$  is the dot product in the predictor space between  $x$  and the support vectors, and the sum includes the training set observations. The score for predicting  $x$  into the negative class is  $-f(x)$ .

If  $G(x_j, x) = x_j'x$  (the linear kernel), then the score function reduces to

$$f(x) = (x/s)' \beta + b.$$

$s$  is the kernel scale and  $\beta$  is the vector of fitted linear coefficients.

## Posterior Probability

The probability that an observation belongs in a particular class, given the data.

For SVM, the posterior probability is a function of the score,  $P(s)$ , that observation  $j$  is in class  $k = \{-1, 1\}$ .

- For separable classes, the posterior probability is the step function

$$P(s_j) = \begin{cases} 0; & s < \max_{y_k=-1} s_k \\ \pi; & \max_{y_k=-1} s_k \leq s_j \leq \min_{y_k=+1} s_k \\ 1; & s_j > \min_{y_k=+1} s_k \end{cases}$$

where:

- $s_j$  is the score of observation  $j$ .
- $+1$  and  $-1$  denote the positive and negative classes, respectively.
- $\pi$  is the prior probability that an observation is in the positive class.
- For inseparable classes, the posterior probability is the sigmoid function

$$P(s_j) = \frac{1}{1 + \exp(As_j + B)},$$

where the parameters  $A$  and  $B$  are the slope and intercept parameters.

## Prior Probability

The prior probability is the believed relative frequency that observations from a class occur in the population for each class.

## Algorithms

- By default and irrespective of the model kernel function, MATLAB uses the dual representation of the score function to classify observations based on trained SVM models, specifically

$$\hat{f}(x) = \sum_{j=1}^n \hat{\alpha}_j y_j G(x, x_j) + \hat{b}.$$

This prediction method requires, among other things, the trained support vectors and  $\alpha$  coefficients (see the `SupportVectors` and `Alpha` properties of the SVM model).

If you are using a linear SVM model for classification and there are many support vectors, then this prediction method can be slow. To efficiently classify observations based on a linear SVM model, remove the support vectors from the model object using `discardSupportVectors`. The resulting model uses the simple linear score function for prediction instead, specifically

$$\hat{f}(x) = x' \hat{\beta} + \hat{b}.$$

For more details, see “Support Vector Machines for Binary Classification” on page 32-2129.

- By default, the software computes optimal posterior probabilities using Platt’s method [1]:
  - 1 Performing 10-fold cross validation
  - 2 Fitting the sigmoid function parameters to the scores returned from the cross validation
  - 3 Estimating the posterior probabilities by entering the cross-validation scores into the fitted sigmoid function
- The software incorporates prior probabilities in the SVM objective function during training.
- For SVM, `predict` classifies observations into the class yielding the largest score (i.e., the largest posterior probability). The software accounts for misclassification costs by applying the average-cost correction before training the classifier. That is, given the class prior vector  $P$ , misclassification cost matrix  $C$ , and observation weight vector  $w$ , the software defines a new vector of observation weights ( $W$ ) such that

$$W_j = w_j P_j \sum_{k=1}^K C_{jk}.$$

## References

- [1] Platt, J. “Probabilistic outputs for support vector machines and comparisons to regularized likelihood methods.” *Advances in Large Margin Classifiers*. MIT Press, 1999, pages 61–74.

## Extended Capabilities

### Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays” (MATLAB).

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

Argument	Notes and Limitations
SVMModel	<ul style="list-style-type: none"> <li>• Save a trained model by using <code>saveCompactModel</code>, and then load the model by using <code>loadCompactModel</code> within a function that you declare.</li> <li>• If you use <code>saveCompactModel</code> to save an SVM model that is equipped to predict posterior probabilities, and use <code>loadCompactModel</code> to load the model, then <code>loadCompactModel</code> cannot restore the <code>ScoreTransform</code> property into the MATLAB Workspace. However, <code>loadCompactModel</code> can load the model, including the <code>ScoreTransform</code> property, at compile time for code generation, within a function that you declare.</li> <li>• Must be a compile-time constant; its value cannot change in the generated code.</li> <li>• For more details, see “Code Generation” on page 32-0 of the <code>CompactClassificationSVM</code> class.</li> </ul>
X	<ul style="list-style-type: none"> <li>• Must be a single- or double-precision matrix and can be variable-sized. However, the number of columns in X must be <code>numel(SVMdl.PredictorNames)</code>.</li> <li>• Rows and columns must correspond to observations and predictors, respectively.</li> </ul>

## See Also

`ClassificationSVM` | `CompactClassificationSVM` | `fitSVMPosterior` | `fitsvm` | `loss`

## Topics

“Plot Posterior Probability Regions for SVM Classification Models” on page 18-218

“Train SVM Classifier Using Custom Kernel” on page 18-199

“Analyze Images Using Linear Support Vector Machines” on page 18-221

“Support Vector Machines for Binary Classification” on page 18-186

“Code Generation Workflows” on page 31-2

Introduced in R2014a

# predict

**Class:** CompactClassificationTree

Predict labels using classification tree

## Syntax

```
label = predict(Mdl,X)
label = predict(Mdl,X,Name,Value)
[label,score,node,cnum] = predict(____)
```

## Description

`label = predict(Mdl,X)` returns a vector of predicted class labels for the predictor data in the table or matrix `X`, based on the trained, full or compact classification tree `Mdl`.

`label = predict(Mdl,X,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments. For example, you can specify to prune `Mdl` to a particular level before predicting labels.

`[label,score,node,cnum] = predict(____)` uses any of the input argument in the previous syntaxes and additionally returns:

- A matrix of classification scores (`score`) indicating the likelihood that a label comes from a particular class. For classification trees, scores are posterior probabilities. For each observation in `X`, the predicted class label on page 32-4984 corresponds to the minimum expected misclassification cost on page 32-4988 among all classes.
- A vector of predicted node numbers for the classification (`node`).
- A vector of predicted class number for the classification (`cnum`).

## Input Arguments

**mdl** — Trained classification tree

ClassificationTree model object | CompactClassificationTree model object

Trained classification tree, specified as a `ClassificationTree` or `CompactClassificationTree` model object. That is, `Mdl` is a trained classification model returned by `fitctree` or `compact`.

### **x** — Predictor data to be classified

numeric matrix | table

Predictor data to be classified, specified as a numeric matrix or table.

Each row of `X` corresponds to one observation, and each column corresponds to one variable.

- For a numeric matrix:
  - The variables making up the columns of `X` must have the same order as the predictor variables that trained `Mdl`.
  - If you trained `Mdl` using a table (for example, `Tbl`), then `X` can be a numeric matrix if `Tbl` contains all numeric predictor variables. To treat numeric predictors in `Tbl` as categorical during training, identify categorical predictors using the `CategoricalPredictors` name-value pair argument of `fitctree`. If `Tbl` contains heterogeneous predictor variables (for example, numeric and categorical data types) and `X` is a numeric matrix, then `predict` throws an error.
- For a table:
  - `predict` does not support multi-column variables and cell arrays other than cell arrays of character vectors.
  - If you trained `Mdl` using a table (for example, `Tbl`), then all predictor variables in `X` must have the same variable names and data types as those that trained `Mdl` (stored in `Mdl.PredictorNames`). However, the column order of `X` does not need to correspond to the column order of `Tbl`. `Tbl` and `X` can contain additional variables (response variables, observation weights, etc.), but `predict` ignores them.
  - If you trained `Mdl` using a numeric matrix, then the predictor names in `Mdl.PredictorNames` and corresponding predictor variable names in `X` must be the same. To specify predictor names during training, see the `PredictorNames` name-value pair argument of `fitctree`. All predictor variables in `X` must be numeric vectors. `X` can contain additional variables (response variables, observation weights, etc.), but `predict` ignores them.



Data Types: `table` | `double` | `single`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### Subtrees — Pruning level

0 (default) | vector of nonnegative integers | 'all'

Pruning level, specified as the comma-separated pair consisting of 'Subtrees' and a vector of nonnegative integers in ascending order or 'all'.

If you specify a vector, then all elements must be at least 0 and at most `max(Mdl.PruneList)`. 0 indicates the full, unpruned tree and `max(Mdl.PruneList)` indicates the completely pruned tree (i.e., just the root node).

If you specify 'all', then `CompactClassificationTree.predict` operates on all subtrees (i.e., the entire pruning sequence). This specification is equivalent to using `0:max(Mdl.PruneList)`.

`CompactClassificationTree.predict` prunes `Mdl` to each level indicated in `Subtrees`, and then estimates the corresponding output arguments. The size of `Subtrees` determines the size of some output arguments.

To invoke `Subtrees`, the properties `PruneList` and `PruneAlpha` of `Mdl` must be nonempty. In other words, grow `Mdl` by setting 'Prune', 'on', or by pruning `Mdl` using `prune`.

Example: 'Subtrees', 'all'

## Output Arguments

### label — Predicted class labels

vector | array

Predicted class labels on page 32-4984, returned as a vector or array. Each entry of `label` corresponds to the class with minimal expected cost for the corresponding row of `X`.

Suppose `Subtrees` is a numeric vector containing `T` elements (for 'all', see `Subtrees`), and `X` has `N` rows.

- If the response data type is `char` and:
  - `T = 1`, then `label` is a character matrix containing `N` rows. Each row contains the predicted label produced by subtree `Subtrees`.
  - `T > 1`, then `label` is an `N`-by-`T` cell array.
- Otherwise, `labels` is an `N`-by-`T` array having the same data type as the response.

In the latter two cases, column `j` of `labels` contains the vector of predicted labels produced by subtree `Subtrees(j)`.

### **score — Posterior probabilities**

numeric matrix

Posterior probabilities, returned as a numeric matrix of size `N`-by-`K`, where `N` is the number of observations (rows) in `X`, and `K` is the number of classes (in `Mdl.ClassNames`). `score(i,j)` is the posterior probability that row `i` of `X` is of class `j`.

If `Subtrees` has `T` elements, and `X` has `N` rows, then `score` is an `N`-by-`K`-by-`T` array, and `node` and `cnum` are `N`-by-`T` matrices.

### **node — Node numbers**

numeric vector

Node numbers for the predicted classes, returned as a numeric vector. Each entry corresponds to the predicted node in `Mdl` for the corresponding row of `X`.

### **cnum — Class numbers**

numeric vector

Class numbers corresponding to the predicted `labels`, returned as a numeric vector. Each entry of `cnum` corresponds to a predicted class number for the corresponding row of `X`.

## Examples

### Predict Labels Using a Classification Tree

Examine predictions for a few rows in a data set left out of training.

Load Fisher's iris data set.

```
load fisheriris
```

Partition the data into training (50%) and validation (50%) sets.

```
n = size(meas,1);
rng(1) % For reproducibility
idxTrn = false(n,1);
idxTrn(randsample(n,round(0.5*n))) = true; % Training set logical indices
idxVal = idxTrn == false; % Validation set logical indices
```

Grow a classification tree using the training set.

```
Mdl = fitctree(meas(idxTrn,:),species(idxTrn));
```

Predict labels for the validation data. Count the number of misclassified observations.

```
label = predict(Mdl,meas(idxVal,:));
label(randsample(numel(label),5)) % Display several predicted labels
numMisclass = sum(~strcmp(label,species(idxVal)))
```

```
ans =
```

```
5x1 cell array

    {'setosa' }
    {'setosa' }
    {'setosa' }
    {'virginica' }
    {'versicolor' }
```

```
numMisclass =
```

```
3
```

The software misclassifies three out-of-sample observations.

### Estimate Class Posterior Probabilities Using a Classification Tree

Load Fisher's iris data set.

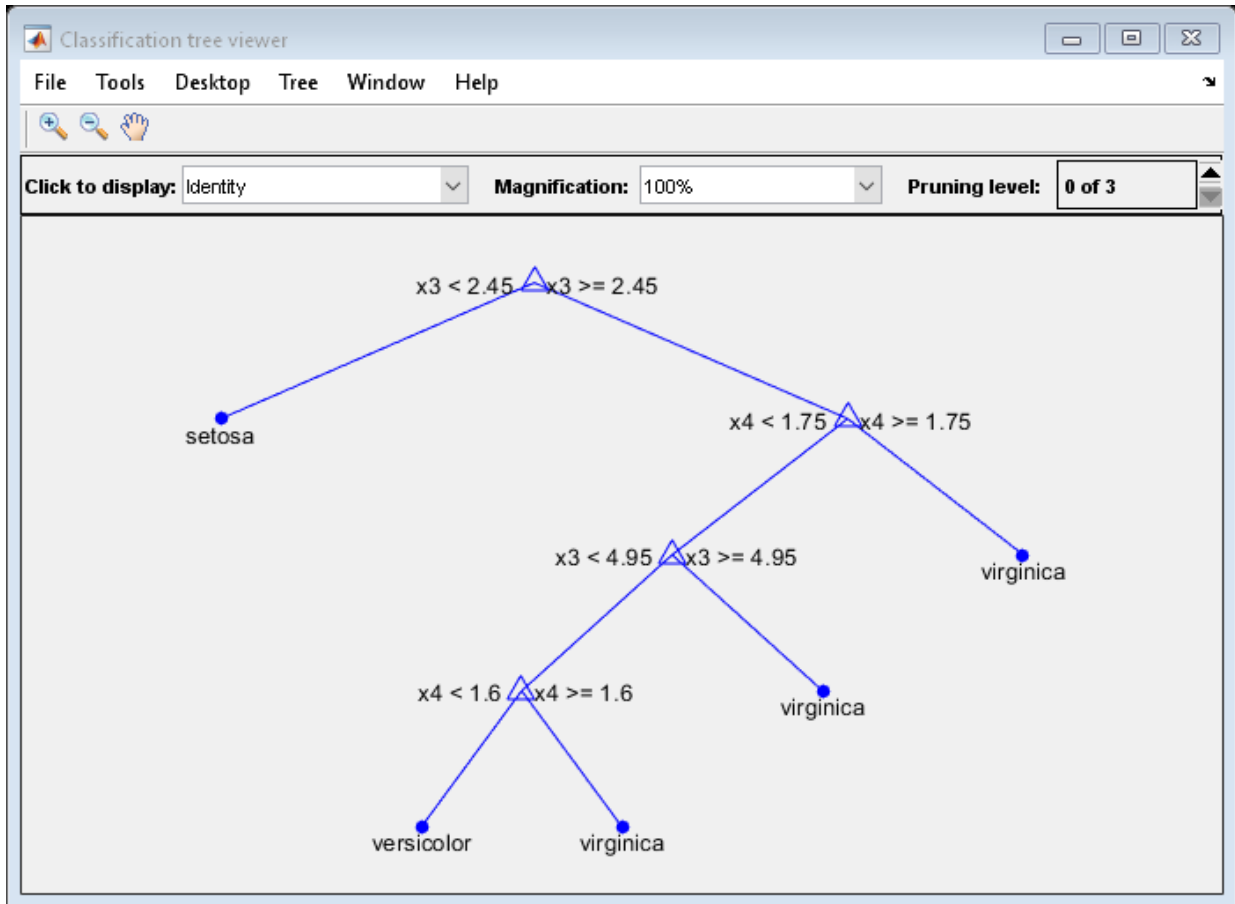
```
load fisheriris
```

Partition the data into training (50%) and validation (50%) sets.

```
n = size(meas,1);  
rng(1) % For reproducibility  
idxTrn = false(n,1);  
idxTrn(randsample(n,round(0.5*n))) = true; % Training set logical indices  
idxVal = idxTrn == false; % Validation set logical indices
```

Grow a classification tree using the training set, and then view it.

```
Mdl = fitctree(meas(idxTrn,:), species(idxTrn));  
view(Mdl, 'Mode', 'graph')
```



The resulting tree has four levels.

Estimate posterior probabilities for the test set using subtrees pruned to levels 1 and 3.

```
[~,Posterior] = predict(Mdl,meas(idxVal,:), 'SubTrees', [1 3]);
Mdl.ClassNames
Posterior(randsample(size(Posterior,1),5),:,:) ...
    % Display several posterior probabilities
```

ans =

```
3x1 cell array

    {'setosa'    }
    {'versicolor'}
    {'virginica' }

ans(:,:,1) =

    1.0000         0         0
    1.0000         0         0
    1.0000         0         0
         0         0     1.0000
         0     0.8571     0.1429

ans(:,:,2) =

    0.3733     0.3200     0.3067
    0.3733     0.3200     0.3067
    0.3733     0.3200     0.3067
    0.3733     0.3200     0.3067
    0.3733     0.3200     0.3067
```

The elements of `Posterior` are class posterior probabilities:

- Rows correspond to observations in the validation set.
- Columns correspond to the classes as listed in `Mdl.ClassNames`.
- Pages correspond to the subtrees.

The subtree pruned to level 1 is more sure of its predictions than the subtree pruned to level 3 (i.e., the root node).

- “Specify Variable-Sized Arguments for Code Generation” on page 31-32

## Definitions

### Predicted Class Label

`predict` classifies so as to minimize the expected classification cost:

$$\hat{y} = \arg \min_{y=1,\dots,K} \sum_{k=1}^K \hat{P}(k|x) C(y|k),$$

where

- $\hat{y}$  is the predicted classification.
- $K$  is the number of classes.
- $\hat{P}(k|x)$  is the posterior probability of class  $k$  for observation  $x$ .
- $C(y|k)$  is the cost of classifying an observation as  $y$  when its true class is  $k$ .

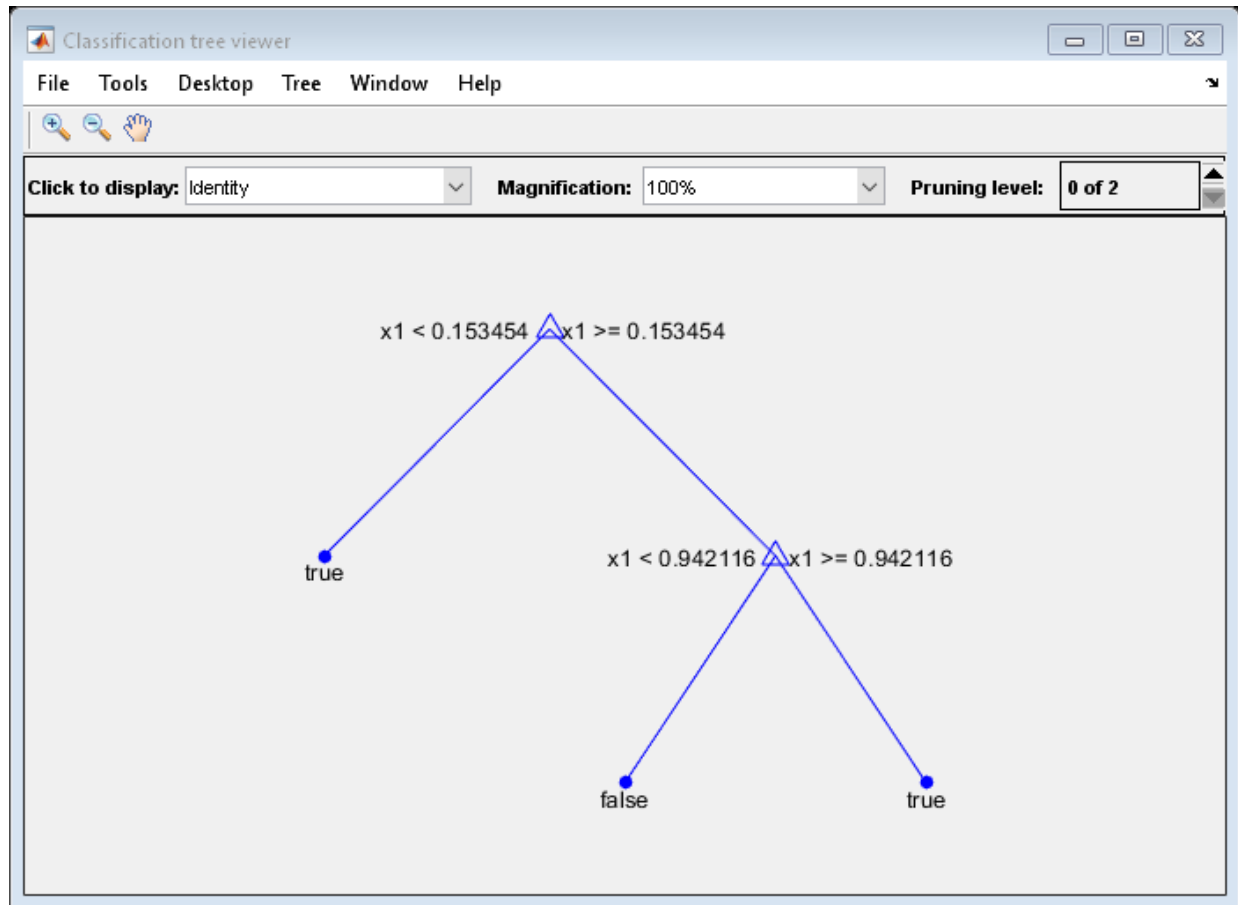
## Score (tree)

For trees, the score of a classification of a leaf node is the posterior probability of the classification at that node. The posterior probability of the classification at a node is the number of training sequences that lead to that node with the classification, divided by the number of training sequences that lead to that node.

For example, consider classifying a predictor  $X$  as `true` when  $X < 0.15$  or  $X > 0.95$ , and  $X$  is false otherwise.

Generate 100 random points and classify them:

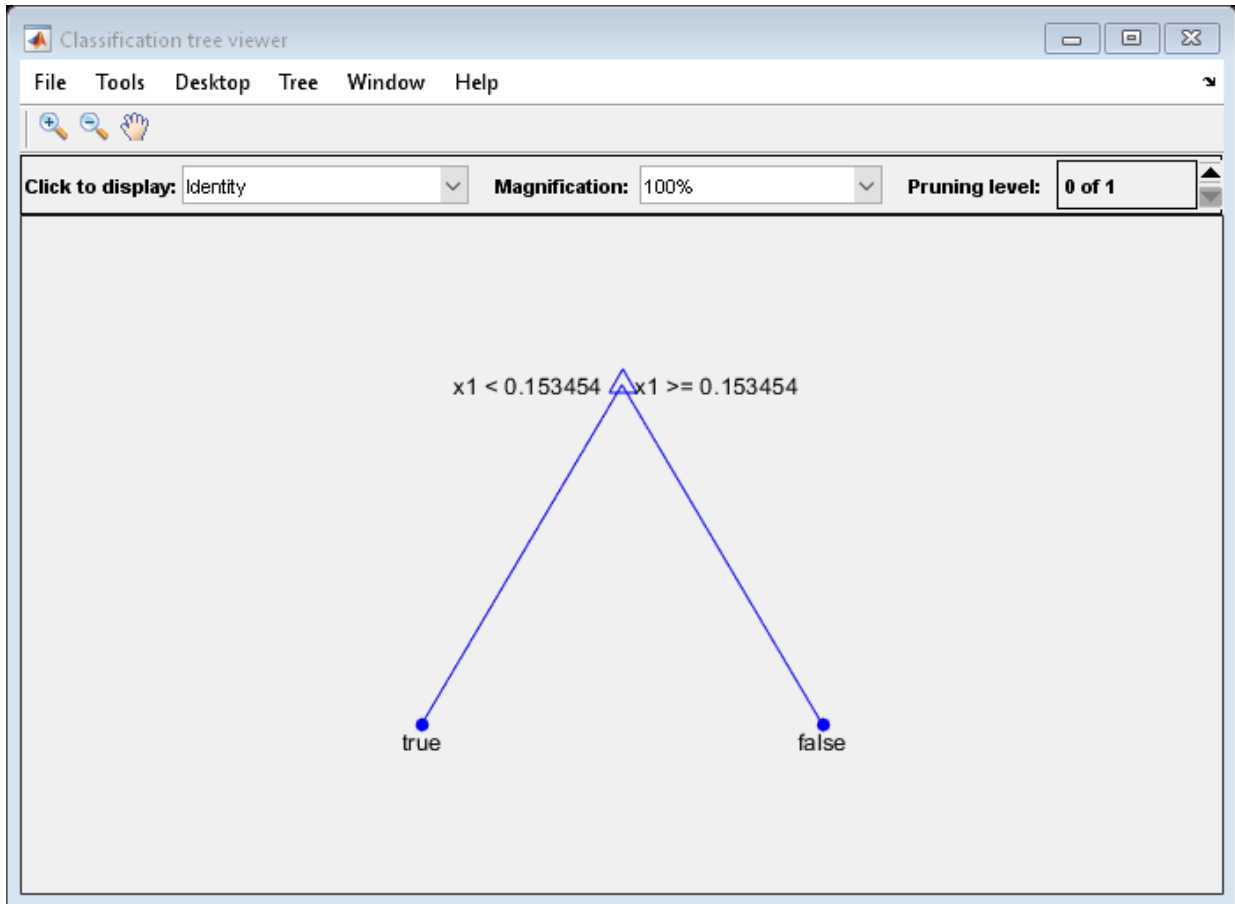
```
rng(0, 'twister') % for reproducibility
X = rand(100,1);
Y = (abs(X - .55) > .4);
tree = fitctree(X,Y);
view(tree, 'Mode', 'Graph')
```



Prune the tree:

```
tree1 = prune(tree, 'Level', 1);  
view(tree1, 'Mode', 'Graph')
```





The pruned tree correctly classifies observations that are less than 0.15 as `true`. It also correctly classifies observations from .15 to .94 as `false`. However, it incorrectly classifies observations that are greater than .94 as `false`. Therefore, the score for observations that are greater than .15 should be about  $.05/.85=.06$  for `true`, and about  $.8/.85=.94$  for `false`.

Compute the prediction scores for the first 10 rows of `X`:

```
[~,score] = predict(tree1,X(1:10));
[score X(1:10,:)]
```

```
ans =  
  
    0.9059    0.0941    0.8147  
    0.9059    0.0941    0.9058  
         0     1.0000    0.1270  
    0.9059    0.0941    0.9134  
    0.9059    0.0941    0.6324  
         0     1.0000    0.0975  
    0.9059    0.0941    0.2785  
    0.9059    0.0941    0.5469  
    0.9059    0.0941    0.9575  
    0.9059    0.0941    0.9649
```

Indeed, every value of  $X$  (the right-most column) that is less than 0.15 has associated scores (the left and center columns) of 0 and 1, while the other values of  $X$  have associated scores of 0.91 and 0.09. The difference (score 0.09 instead of the expected 0.06) is due to a statistical fluctuation: there are 8 observations in  $X$  in the range  $(.95, 1)$  instead of the expected 5 observations.

## True Misclassification Cost

There are two costs associated with classification: the true misclassification cost per class, and the expected misclassification cost per observation.

You can set the true misclassification cost per class in the `Cost` name-value pair when you create the classifier using the `fitctree` method. `Cost(i, j)` is the cost of classifying an observation into class  $j$  if its true class is  $i$ . By default, `Cost(i, j) = 1` if  $i \neq j$ , and `Cost(i, j) = 0` if  $i = j$ . In other words, the cost is 0 for correct classification, and 1 for incorrect classification.

## Expected Cost

There are two costs associated with classification: the true misclassification cost per class, and the expected misclassification cost per observation.

Suppose you have `Nobs` observations that you want to classify with a trained classifier. Suppose you have  $K$  classes. You place the observations into a matrix `Xnew` with one observation per row.

The expected cost matrix CE has size Nobs-by-K. Each row of CE contains the expected (average) cost of classifying the observation into each of the K classes. CE (n, k) is

$$\sum_{i=1}^K \hat{P}(i | X_{new}(n)) C(k | i),$$

where

- $K$  is the number of classes.
- $\hat{P}(i | X_{new}(n))$  is the posterior probability of class  $i$  for observation  $X_{new}(n)$ .
- $C(k | i)$  is the true misclassification cost of classifying an observation as  $k$  when its true class is  $i$ .

## Predictive Measure of Association

The predictive measure of association is a value that indicates the similarity between decision rules that split observations. Among all possible decision splits that are compared to the optimal split (found by growing the tree), the best surrogate decision split on page 32-2174 yields the maximum predictive measure of association. The second-best surrogate split has the second-largest predictive measure of association.

Suppose  $x_j$  and  $x_k$  are predictor variables  $j$  and  $k$ , respectively, and  $j \neq k$ . At node  $t$ , the predictive measure of association between the optimal split  $x_j < u$  and a surrogate split  $x_k < v$  is

$$\lambda_{jk} = \frac{\min(P_L, P_R) - (1 - P_{L_j L_k} - P_{R_j R_k})}{\min(P_L, P_R)}.$$

- $P_L$  is the proportion of observations in node  $t$ , such that  $x_j < u$ . The subscript  $L$  stands for the left child of node  $t$ .
- $P_R$  is the proportion of observations in node  $t$ , such that  $x_j \geq u$ . The subscript  $R$  stands for the right child of node  $t$ .
- $P_{L_j L_k}$  is the proportion of observations at node  $t$ , such that  $x_j < u$  and  $x_k < v$ .
- $P_{R_j R_k}$  is the proportion of observations at node  $t$ , such that  $x_j \geq u$  and  $x_k \geq v$ .

- Observations with missing values for  $x_j$  or  $x_k$  do not contribute to the proportion calculations.

$\lambda_{jk}$  is a value in  $(-\infty, 1]$ . If  $\lambda_{jk} > 0$ , then  $x_k < v$  is a worthwhile surrogate split for  $x_j < u$ .

## Algorithms

`predict` generates predictions by following the branches of `Mdl` until it reaches a leaf node or a missing value. If `predict` reaches a leaf node, it returns the classification of that node.

If `predict` reaches a node with a missing value for a predictor, its behavior depends on the setting of the `Surrogate` name-value pair when `fitctree` constructs `Mdl`.

- **Surrogate = 'off'** (default) — `predict` returns the label with the largest number of training samples that reach the node.
- **Surrogate = 'on'** — `predict` uses the best surrogate split at the node. If all surrogate split variables with positive predictive measure of association are missing, `predict` returns the label with the largest number of training samples that reach the node. For a definition, see “Predictive Measure of Association” on page 32-4989.

## Extended Capabilities

### Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays” (MATLAB).

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

Argument	Notes and Limitations
Mdl	<ul style="list-style-type: none"> <li>Save a trained model by using <code>saveCompactModel</code>, and then load the model by using <code>loadCompactModel</code> within a function that you declare.</li> <li>Must be a compile-time constant; its value cannot change in the generated code.</li> <li>For more details, see “Code Generation” on page 32-0 of the <code>CompactClassificationTree</code> class.</li> </ul>
X	<ul style="list-style-type: none"> <li>Must be a single- or double-precision matrix and can be variable-sized. However, the number of columns in X must be <code>numel(Mdl.PredictorNames)</code>.</li> <li>Rows and columns must correspond to observations and predictors, respectively.</li> </ul>
label	If the response data type is <code>char</code> and <code>codegen</code> cannot ascertain that the value of <code>Subtrees</code> is a scalar, then <code>label</code> is a cell array of character vectors.
Name-value pair arguments	Names in name-value pair arguments must be compile-time constants. For example, to allow user-defined pruning levels in the generated code, include <code>{coder.Constant('Subtrees'),coder.typeof(0,[1,n],[0,1])}</code> in the <code>-args</code> value of <code>codegen</code> , where <code>n</code> is <code>max(Mdl.PruneList)</code> .

## See Also

`ClassificationTree` | `CompactClassificationTree` | `compact` | `edge` | `fitctree` | `loss` | `margin` | `prune`

## Topics

“Specify Variable-Sized Arguments for Code Generation” on page 31-32

“Code Generation Workflows” on page 31-2

Introduced in R2011a

## predict

**Class:** CompactRegressionEnsemble

Predict responses using ensemble of regression models

## Syntax

```
Yfit = predict(Mdl, X)
Yfit = predict(Mdl, X, Name, Value)
```

## Description

`Yfit = predict(Mdl, X)` returns predicted responses to the predictor data in the table or matrix `X`, based on the regression ensemble model `Mdl`.

`Yfit = predict(Mdl, X, Name, Value)` uses additional options specified by one or more `Name, Value` pair arguments.

## Input Arguments

### **Mdl**

Regression ensemble created by `fitrensemble`, or by the `compact` method.

### **X**

Predictor data used to generate responses, specified as a numeric matrix or table.

Each row of `X` corresponds to one observation, and each column corresponds to one variable.

- For a numeric matrix:
  - The variables making up the columns of `X` must have the same order as the predictor variables that trained `Mdl`.

- If you trained `Mdl` using a table (for example, `Tbl`), then `X` can be a numeric matrix if `Tbl` contains all numeric predictor variables. To treat numeric predictors in `Tbl` as categorical during training, identify categorical predictors using the `CategoricalPredictors` name-value pair argument of `fitrensemble`. If `Tbl` contains heterogeneous predictor variables (for example, numeric and categorical data types) and `X` is a numeric matrix, then `predict` throws an error.
- For a table:
  - `predict` does not support multi-column variables and cell arrays other than cell arrays of character vectors.
  - If you trained `Mdl` using a table (for example, `Tbl`), then all predictor variables in `X` must have the same variable names and data types as those that trained `Mdl` (stored in `Mdl.PredictorNames`). However, the column order of `X` does not need to correspond to the column order of `Tbl`. `Tbl` and `X` can contain additional variables (response variables, observation weights, etc.), but `predict` ignores them.
  - If you trained `Mdl` using a numeric matrix, then the predictor names in `Mdl.PredictorNames` and corresponding predictor variable names in `X` must be the same. To specify predictor names during training, see the `PredictorNames` name-value pair argument of `fitrensemble`. All predictor variables in `X` must be numeric vectors. `X` can contain additional variables (response variables, observation weights, etc.), but `predict` ignores them.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### Learners

Indices of weak learners in the ensemble ranging from 1 to `NumTrained`, where `NumTrained` is the number of weak learners.

**Default:** `1:NumTrained`

**UseObsForLearner**

A logical matrix of size  $N$ -by- $\text{NumTrained}$ , where  $N$  is the number of observations in  $X$ , and  $\text{NumTrained}$  is the number of weak learners. When `UseObsForLearner(I, J)` is true, `predict` uses learner  $J$  in predicting observation  $I$ .

**Default:** `true(N, NumTrained)`

## Output Arguments

**Yfit**

A numeric column vector with the same number of rows as `TBLdata` or `Xdata`. Each row of `Yfit` gives the predicted response to the corresponding row of `TBLdata` or `Xdata`, based on the `ens` regression model.

## Examples

**Predict Responses Based on Regression Ensemble**

Find the predicted mileage for a car based on regression ensemble trained on the `carsmall` data.

Load the `carsmall` data set and select the number of cylinders, engine displacement, horsepower, and vehicle weight as predictors.

```
load carsmall
X = [Cylinders Displacement Horsepower Weight];
```

Train an ensemble of regression trees and predict MPG for a four-cylinder car, with 200 cubic inch engine displacement, 150 horsepower, weighing 3000 lbs.

```
rens = fitrensemble(X,MPG);
Mileage = predict(rens,[4 200 150 3000])
```

```
Mileage = 25.6467
```



## Extended Capabilities

### Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays” (MATLAB).

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

Argument	Notes and Limitations
Mdl	<ul style="list-style-type: none"> <li>Save a trained model by using <code>saveCompactModel</code>, and then load the model by using <code>loadCompactModel</code> within a function that you declare.</li> <li>Must be a compile-time constant; its value cannot change in the generated code.</li> <li>For more details, see “Code Generation” on page 32-0 of the <code>CompactRegressionEnsemble</code> class.</li> </ul>
X	<ul style="list-style-type: none"> <li>Must be a single- or double-precision matrix and can be variable-sized. However, the number of columns in X must be <code>numel(Mdl.PredictorNames)</code>.</li> <li>Rows and columns must correspond to observations and predictors, respectively.</li> </ul>
Name-value pair arguments	Names in name-value pair arguments must be compile-time constants. For example, to allow user-defined indices up to 5 weak learners in the generated code, include <code>{coder.Constant('Learners'), coder.typeof(0, [1, 5], [0, 1])}</code> in the <code>-args</code> value of <code>codegen</code> .

## See Also

`CompactRegressionEnsemble` | `RegressionBaggedEnsemble` |  
`RegressionEnsemble` | `fitrensemble` | `loss`

## Topics

“Code Generation Workflows” on page 31-2

# predict

**Class:** CompactRegressionGP

Predict response of Gaussian process regression model

## Syntax

```
ypred = predict(gprMdl, Xnew)  
[ypred, ysd] = predict(gprMdl, Xnew)  
[ypred, ysd, yint] = predict(gprMdl, Xnew)  
[ypred, ysd, yint] = predict(gprMdl, Xnew, Name, Value)
```

## Description

`ypred = predict(gprMdl, Xnew)` returns the predicted responses `ypred` for the full or compact Gaussian process regression (GPR) model, `gprMdl`, and the predictor values in `Xnew`.

`[ypred, ysd] = predict(gprMdl, Xnew)` also returns the estimated standard deviations for the new responses at the predictor values in `Xnew` from a trained GPR model.

`[ypred, ysd, yint] = predict(gprMdl, Xnew)` also returns the 95% prediction intervals, `yint`, for the true responses corresponding to each row of `Xnew`.

`[ypred, ysd, yint] = predict(gprMdl, Xnew, Name, Value)` also returns the prediction intervals with additional options specified by one or more `Name, Value` pair arguments. For example, you can specify the confidence level of the prediction interval.

## Input Arguments

**gprMdl** — Gaussian process regression model

RegressionGP object | CompactRegressionGP object

Gaussian process regression model, specified as a `RegressionGP` (full) or `CompactRegressionGP` (compact) object.

**Xnew — New values for the predictors**

table |  $m$ -by- $d$  matrix

New values for the predictors that `fitrgp` uses in training the GPR model, specified as a table or an  $m$ -by- $d$  matrix.  $m$  is the number of observations and  $d$  is the number of predictor variables in the training data.

If you trained `gprMdl` on a table, then `Xnew` must be a table that contains all the predictor variables used to train `gprMdl`.

If you trained `gprMdl` on a matrix, then `Xnew` must be a numeric matrix with  $d$  columns.

Data Types: single | double | table

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

**Alpha — Confidence level**

0.05 (default) | scalar value in the range from 0 to 1

Confidence level for the prediction intervals, specified as the comma-separated pair consisting of 'Alpha' and a scalar value in the range from 0 to 1.

Example: 'Alpha', 0.01 specifies 99% prediction intervals.

Data Types: single | double

## Output Arguments

**ypred — Predicted response values**

$n$ -by-1 vector

Predicted response values, returned as an  $n$ -by-1 vector.

**y<sub>sd</sub>** — Estimated standard deviation of the new response values*n*-by-1 vector

Estimated standard deviation of the new response values, returned as an *n*-by-1 vector, where  $y_{sd}(i)$ ,  $i = 1, 2, \dots, n$ , contains the estimated standard deviation of the new response corresponding to the predictor values at the  $i^{\text{th}}$  row of  $X_{\text{new}}$  from the trained GPR model.

**y<sub>int</sub>** — Prediction intervals for the true response values*n*-by-2 matrix

Prediction intervals for the true response values corresponding to each row of  $X_{\text{new}}$ , returned as an *n*-by-2 matrix. The first column of  $y_{\text{int}}$  contains the lower limits and the second column contains the upper limits of the prediction intervals.

## Examples

### Compute Predicted Responses

Generate the sample data.

```
n = 10000;
rng(1) % For reproducibility
x = linspace(0.5, 2.5, n)';
y = sin(10*pi.*x) ./ (2.*x) + (x-1).^4 + 1.5*rand(n,1);
```

Fit a GPR model using the Matern 3/2 kernel function with separate length scale for each predictor and an active set size of 100. Use the subset of regressors approximation method for parameter estimation and fully independent conditional method for prediction.

```
gprMdl = fitrgp(x, y, 'KernelFunction', 'ardmatern32', ...
'ActiveSetSize', 100, 'FitMethod', 'sr', 'PredictMethod', 'fic');
```

Compute the predictions.

```
[ypred,~,yci] = predict(gprMdl,x);
```

Plot the data along with the predictions and prediction intervals.

```
plot(x,y,'r.');
```

```
hold on
```

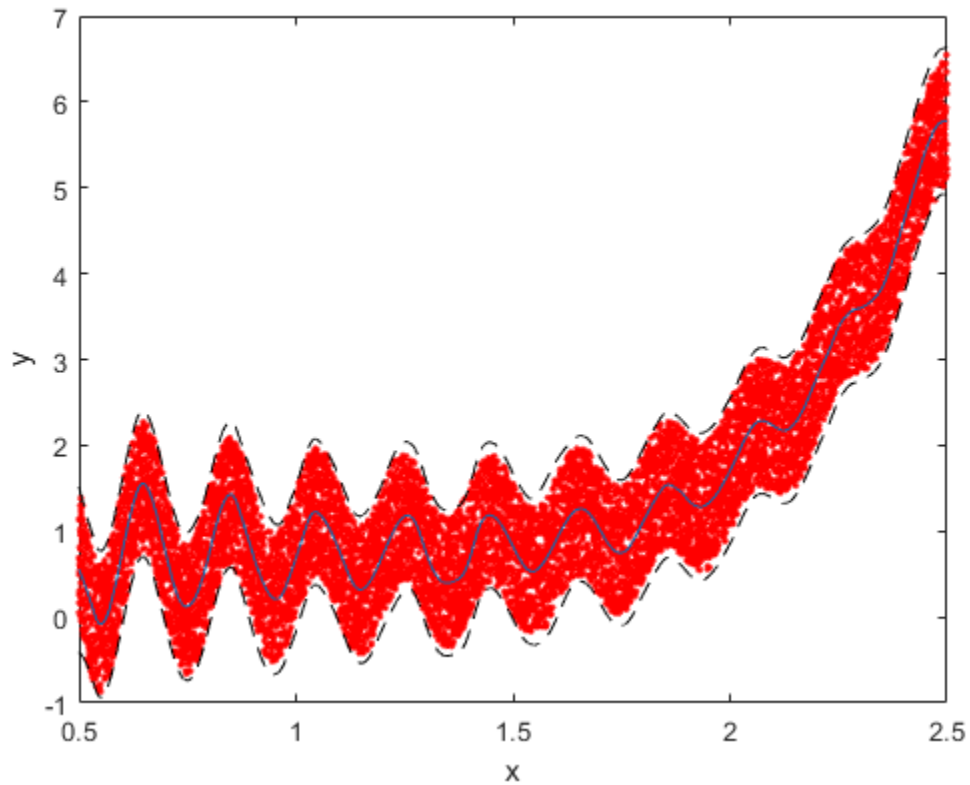
```
plot(x,ypred);
```

```
plot(x,yci(:,1),'k--');
```

```
plot(x,yci(:,2),'k--');
```

```
xlabel('x');
```

```
ylabel('y');
```



### Compute Predictions When Data in Table

Load the sample data and store in a table.

```
load fisheriris
tbl = table(meas(:,1),meas(:,2),meas(:,3),meas(:,4),species,...
'VariableNames',{'meas1','meas2','meas3','meas4','species'});
```

Fit a GPR model using the first measurement as the response and the other variables as the predictors.

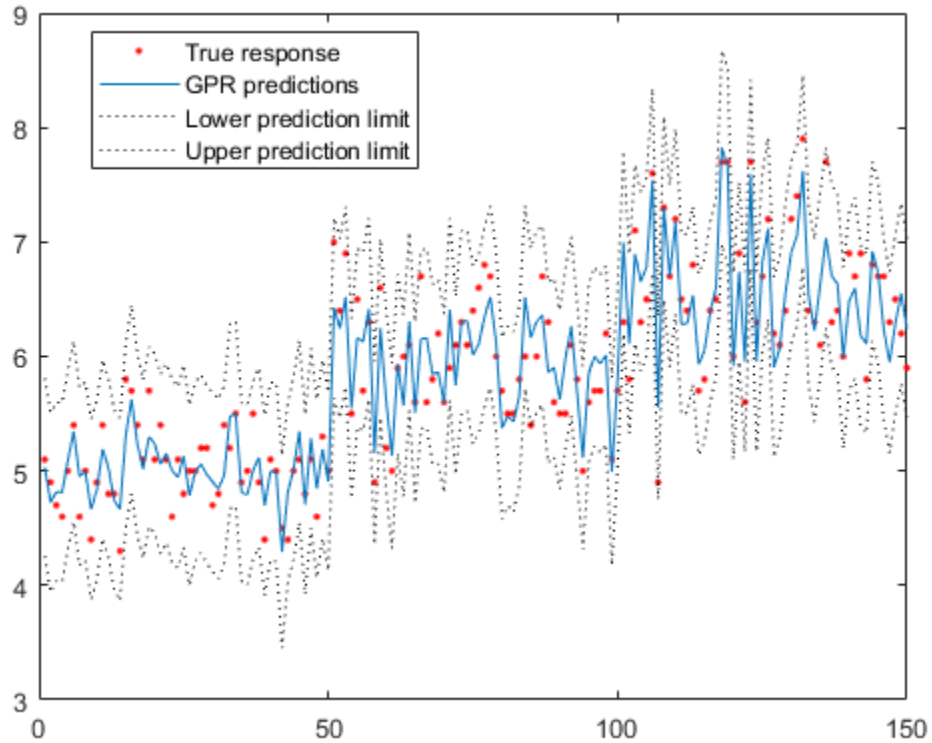
```
mdl = fitrgp(tbl,'meas1');
```

Compute the predictions and the 99% confidence intervals.

```
[ypred,~,yci] = predict(mdl,tbl,'Alpha',0.01);
```

Plot the true response and the predictions along with the prediction intervals.

```
figure();
plot(mdl.Y,'r. ');
hold on;
plot(ypred);
plot(yci(:,1),'k: ');
plot(yci(:,2),'k: ');
legend('True response','GPR predictions',...
'Lower prediction limit','Upper prediction limit',...
'Location','Best');
```



### Plot Predicted Response for Test Data

Load the sample data.

```
load(fullfile(matlabroot, 'examples', 'stats', 'gprdata.mat'));
```

The data contains training and test data. There are 500 observations in training data and 100 observations in test data. The data has 8 predictor variables. This is simulated data.



Fit a GPR model using the squared exponential kernel function with a separate length scale for each predictor. Standardize predictors in the training data. Use the exact fitting and prediction methods.

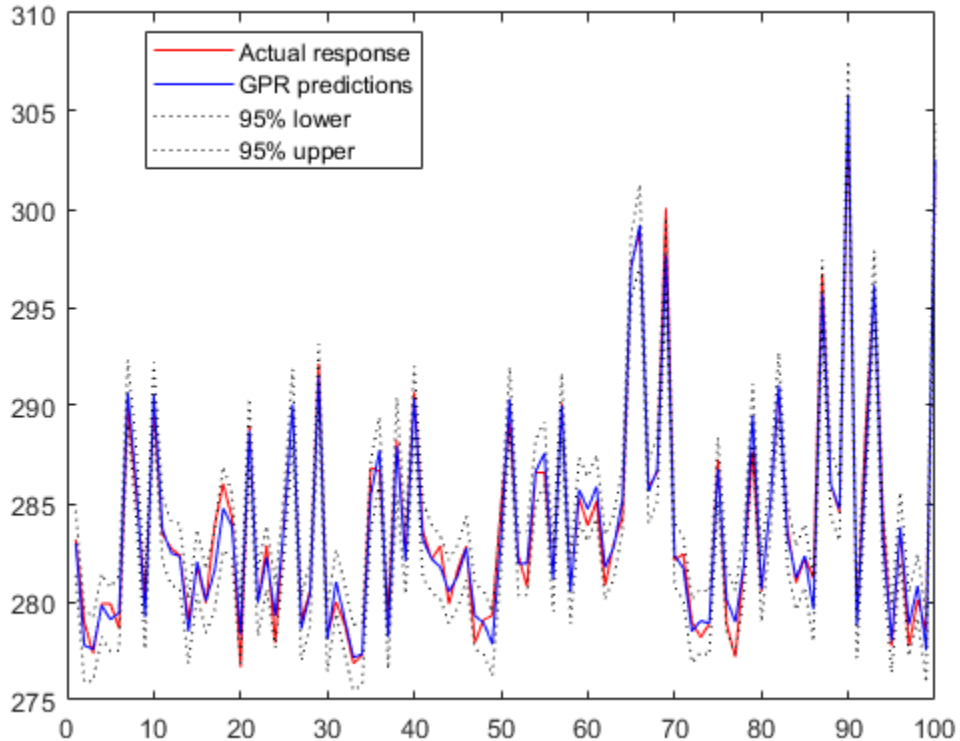
```
gprMdl = fitrgp(Xtrain,ytrain,'Basis','constant',...  
'FitMethod','exact','PredictMethod','exact',...  
'KernelFunction','ardsquaredexponential','Standardize',1);
```

Predict the responses for test data.

```
[ytestpred,~,ytestci] = predict(gprMdl,Xtest);
```

Plot the test response along with the predictions.

```
figure;  
plot(ytest,'r');  
hold on;  
plot(ytestpred,'b');  
plot(ytestci(:,1),'k:');  
plot(ytestci(:,2),'k:');  
legend('Actual response','GPR predictions',...  
'95% lower','95% upper','Location','Best');  
hold off
```



## Tips

- You can choose the prediction method while training the GPR model using the `PredictMethod` name-value pair argument in `fitrgp`. The default prediction method is `'exact'` for  $n \leq 10000$ , where  $n$  is the number of observations in the training data, and `'bcd'` (block coordinate descent), otherwise.
- Computation of standard deviations, `ysd`, and prediction intervals, `yint`, is not supported when `PredictMethod` is `'bcd'`.
- If `gprMdl` is a `CompactRegressionGP` object, you cannot compute standard deviations, `ysd`, or prediction intervals, `yint`, for `PredictMethod` equal to `'sr'` or

'fic'. To compute `y` and `yint` for `PredictMethod` equal to 'sr' or 'fic', use the full regression (`RegressionGP`) object.

## Alternatives

You can use `resubPredict` to compute the predicted responses for the trained GPR model at the observations in the training data.

## Extended Capabilities

### Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays” (MATLAB).

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

Argument	Notes and Limitations
<code>Mdl</code>	<ul style="list-style-type: none"> <li>Save a trained model by using <code>saveCompactModel</code>, and then load the model by using <code>loadCompactModel</code> within a function that you declare.</li> <li>Must be a compile-time constant; its value cannot change in the generated code.</li> <li>For more details, see “Code Generation” on page 32-0 of the <code>CompactRegressionGP</code> class.</li> </ul>

Argument	Notes and Limitations
X	<ul style="list-style-type: none"><li>• Must be a single- or double-precision matrix and can be variable-sized. However, the number of columns in Xnew must be <code>numel(gprMdl.PredictorNames)</code>.</li><li>• Rows and columns must correspond to observations and predictors, respectively.</li></ul>
Name-value pair arguments	Names in name-value pair arguments must be compile-time constants. For example, to allow a user-defined significance level in the generated code, include <code>{coder.Constant('Alpha'),0}</code> in the <code>-args</code> value of <code>codegen</code> .

## See Also

`CompactRegressionGP` | `RegressionGP` | `compact` | `fitrgp` | `loss` | `resubPredict`

## Topics

“Code Generation Workflows” on page 31-2

Introduced in R2015b

# predict

**Class:** CompactRegressionSVM

Predict responses using support vector machine regression model

## Syntax

```
yfit = predict(Mdl,X)
```

## Description

`yfit = predict(Mdl,X)` returns a vector of predicted responses for the predictor data in the table or matrix `X`, based on the full or compact, trained support vector machine (SVM) regression model `Mdl`.

## Input Arguments

### **Mdl** — SVM regression model

RegressionSVM object | CompactRegressionSVM object

SVM regression model, specified as a `RegressionSVM` model or a `CompactRegressionSVM` model, returned by `fitrsvm` or `compact`, respectively.

### **x** — Predictor data used to generate responses

numeric matrix | table

Predictor data used to generate responses, specified as a numeric matrix or table.

Each row of `X` corresponds to one observation, and each column corresponds to one variable.

- For a numeric matrix:
  - The variables making up the columns of `X` must have the same order as the predictor variables that trained `Mdl`.

- If you trained `Mdl` using a table (for example, `Tbl`), then `X` can be a numeric matrix if `Tbl` contains all numeric predictor variables. To treat numeric predictors in `Tbl` as categorical during training, identify categorical predictors using the `CategoricalPredictors` name-value pair argument of `fitrsvm`. If `Tbl` contains heterogeneous predictor variables (for example, numeric and categorical data types) and `X` is a numeric matrix, then `predict` throws an error.
- For a table:
  - `predict` does not support multi-column variables and cell arrays other than cell arrays of character vectors.
  - If you trained `Mdl` using a table (for example, `Tbl`), then all predictor variables in `X` must have the same variable names and data types as those that trained `Mdl` (stored in `Mdl.PredictorNames`). However, the column order of `X` does not need to correspond to the column order of `Tbl`. `Tbl` and `X` can contain additional variables (response variables, observation weights, etc.), but `predict` ignores them.
  - If you trained `Mdl` using a numeric matrix, then the predictor names in `Mdl.PredictorNames` and corresponding predictor variable names in `X` must be the same. To specify predictor names during training, see the `PredictorNames` name-value pair argument of `fitrsvm`. All predictor variables in `X` must be numeric vectors. `X` can contain additional variables (response variables, observation weights, etc.), but `predict` ignores them.

If you set `'Standardize', true` in `fitrsvm` to train `Mdl`, then the software standardizes the columns of `X` using the corresponding means in `Mdl.Mu` and standard deviations in `Mdl.Sigma`.

Data Types: `table` | `double` | `single`

## Output Arguments

**`yfit`** — Predicted responses  
vector

Predicted responses, returned as a vector of length  $n$ , where  $n$  is the number of observations in the training data.

For details about how to predict responses, see “Equation 25-1” on page 25-4 and “Equation 25-2” on page 25-6 in “Understanding Support Vector Machine Regression” on page 25-2.

## Examples

### Predict Test Sample Response for SVM Regression Model

Load the `carsmall` data set. Consider a model that predicts a car's fuel efficiency given its horsepower and weight. Determine the sample size.

```
load carsmall
tbl = table(Horsepower,Weight,MPG);
N = size(tbl,1);
```

Partition the data into training and test sets. Hold out 10% of the data for testing.

```
rng(10); % For reproducibility
cvp = cvpartition(N,'Holdout',0.1);
idxTrn = training(cvp); % Training set indices
idxTest = test(cvp); % Test set indices
```

Train a linear SVM regression model. Standardize the data.

```
Mdl = fitrsvm(tbl(idxTrn,:), 'MPG', 'Standardize', true);
```

`Mdl` is a `RegressionSVM` model.

Predict responses for the test set.

```
YFit = predict(Mdl,tbl(idxTest,:));
```

Create a table containing the observed response values and the predicted response values side by side.

```
table(tbl.MPG(idxTest),YFit,'VariableNames',...
      {'ObservedValue','PredictedValue'})
```

```
ans =
```

```
10x2 table
```

ObservedValue	PredictedValue
14	9.4833
27	28.938
10	7.765
28	27.155
22	21.054
29	31.484
24.5	30.306
18.5	19.12
32	28.225
28	26.632

## Tips

- If `mdl` is a cross-validated `RegressionPartitionedSVM` model, use `kfoldPredict` instead of `predict` to predict new response values.

## Extended Capabilities

### Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays” (MATLAB).

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:



Argument	Notes and Limitations
Mdl	<ul style="list-style-type: none"> <li>• Save a trained model by using <code>saveCompactModel</code>, and then load the model by using <code>loadCompactModel</code> within a function that you declare.</li> <li>• Must be a compile-time constant; its value cannot change in the generated code.</li> <li>• For more details, see “Code Generation” on page 32-0 of the <code>CompactRegressionSVM</code> class.</li> </ul>
X	<ul style="list-style-type: none"> <li>• Must be a single- or double-precision matrix and can be variable-sized. However, the number of columns in X must be <code>numel(Mdl.PredictorNames)</code>.</li> <li>• Rows and columns must correspond to observations and predictors, respectively.</li> </ul>

## See Also

`CompactRegressionSVM` | `RegressionSVM` | `fitrsvm` | `kfoldPredict`

## Topics

“Code Generation Workflows” on page 31-2

Introduced in R2015b

## predict

**Class:** CompactRegressionTree

Predict responses using regression tree

## Syntax

```
Yfit = predict(Mdl,X)
Yfit = predict(Mdl,X,Name,Value)
[Yfit,node] = predict(____)
```

## Description

`Yfit = predict(Mdl,X)` returns a vector of predicted responses for the predictor data in the table or matrix `X`, based on the full or compact regression tree `Mdl`.

`Yfit = predict(Mdl,X,Name,Value)` predicts response values with additional options specified by one or more `Name,Value` pair arguments. For example, you can specify to prune `Mdl` to a particular level before predicting responses.

`[Yfit,node] = predict(____)` also returns a vector of predicted node numbers for the responses, using any of the input arguments in the previous syntaxes.

## Input Arguments

**mdl** — Trained regression tree

RegressionTree model object | CompactRegressionTree model object

Trained classification tree, specified as a `RegressionTree` or `CompactRegressionTree` model object. That is, `Mdl` is a trained classification model returned by `fitrtree` or `compact`.

**x** — Predictor data to be classified

numeric matrix | table

Predictor data to be classified, specified as a numeric matrix or table.

Each row of `X` corresponds to one observation, and each column corresponds to one variable.

- For a numeric matrix:
  - The variables making up the columns of `X` must have the same order as the predictor variables that trained `Mdl`.
  - If you trained `Mdl` using a table (for example, `Tbl`), then `X` can be a numeric matrix if `Tbl` contains all numeric predictor variables. To treat numeric predictors in `Tbl` as categorical during training, identify categorical predictors using the `CategoricalPredictors` name-value pair argument of `fitrtree`. If `Tbl` contains heterogeneous predictor variables (for example, numeric and categorical data types) and `X` is a numeric matrix, then `predict` throws an error.
- For a table:
  - `predict` does not support multi-column variables and cell arrays other than cell arrays of character vectors.
  - If you trained `Mdl` using a table (for example, `Tbl`), then all predictor variables in `X` must have the same variable names and data types as those that trained `Mdl` (stored in `Mdl.PredictorNames`). However, the column order of `X` does not need to correspond to the column order of `Tbl`. `Tbl` and `X` can contain additional variables (response variables, observation weights, etc.), but `predict` ignores them.
  - If you trained `Mdl` using a numeric matrix, then the predictor names in `Mdl.PredictorNames` and corresponding predictor variable names in `X` must be the same. To specify predictor names during training, see the `PredictorNames` name-value pair argument of `fitrtree`. All predictor variables in `X` must be numeric vectors. `X` can contain additional variables (response variables, observation weights, etc.), but `predict` ignores them.

Data Types: `table` | `double` | `single`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single

quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

**Subtrees — Pruning level**

0 (default) | vector of nonnegative integers | 'all'

Pruning level, specified as the comma-separated pair consisting of 'Subtrees' and a vector of nonnegative integers in ascending order or 'all'.

If you specify a vector, then all elements must be at least 0 and at most `max(Mdl.PruneList)`. 0 indicates the full, unpruned tree and `max(Mdl.PruneList)` indicates the completely pruned tree (i.e., just the root node).

If you specify 'all', then `CompactRegressionTree.predict` operates on all subtrees (i.e., the entire pruning sequence). This specification is equivalent to using `0:max(Mdl.PruneList)`.

`CompactRegressionTree.predict` prunes `Mdl` to each level indicated in `Subtrees`, and then estimates the corresponding output arguments. The size of `Subtrees` determines the size of some output arguments.

To invoke `Subtrees`, the properties `PruneList` and `PruneAlpha` of `Mdl` must be nonempty. In other words, grow `Mdl` by setting 'Prune', 'on', or by pruning `Mdl` using `prune`.

Example: 'Subtrees', 'all'

## Output Arguments

**`yfit` — Predicted response values**

numeric column vector

Predicted response values, returned as a numeric column vector with the same number of rows as `X`. Each row of `Yfit` gives the predicted response to the corresponding row of `X`, based on the `Mdl`.

**`node` — Node numbers**

numeric vector

Node numbers for the predictions, specified as a numeric vector. Each entry corresponds to the predicted leaf node in `Mdl` for the corresponding row of `X`.

## Examples

### Predict a Response Using a Regression Tree

Load the `carsmall` data set. Consider `Displacement`, `Horsepower`, and `Weight` as predictors of the response `MPG`.

```
load carsmall
X = [Displacement Horsepower Weight];
```

Grow a regression tree using the entire data set.

```
Mdl = fitrtree(X,MPG);
```

Predict the `MPG` for a car with 200 cubic inch engine displacement, 150 horsepower, and that weighs 3000 lbs.

```
X0 = [200 150 3000];
MPG0 = predict(Mdl,X0)
```

```
MPG0 =
```

```
21.9375
```

The regression tree predicts the car's efficiency to be 21.94 mpg.

- “Predict Out-of-Sample Responses of Subtrees” on page 19-13
- “Decision Trees” on page 19-2
- “Prediction Using Classification and Regression Trees” on page 19-11
- “Specify Variable-Sized Arguments for Code Generation” on page 31-32

## Extended Capabilities

### Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays” (MATLAB).

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

Argument	Notes and Limitations
Mdl	<ul style="list-style-type: none"> <li>Save a trained model by using <code>saveCompactModel</code>, and then load the model by using <code>loadCompactModel</code> within a function that you declare.</li> <li>Must be a compile-time constant; its value cannot change in the generated code.</li> <li>For more details, see “Code Generation” on page 32-0 of the <code>CompactRegressionTree</code> class.</li> </ul>
X	<ul style="list-style-type: none"> <li>Must be a single- or double-precision matrix and can be variable-sized. However, the number of columns in X must be <code>numel(Mdl.PredictorNames)</code>.</li> <li>Rows and columns must correspond to observations and predictors, respectively.</li> </ul>
Name-value pair arguments	<p>Names in name-value pair arguments must be compile-time constants. For example, to allow user-defined pruning levels in the generated code, include <code>{coder.Constant('Subtrees'), coder.typeof(0, [1, n], [0, 1])}</code> in the <code>-args</code> value of <code>codegen</code>, where <code>n</code> is <code>max(Mdl.PruneList)</code>.</p>

## See Also

`CompactRegressionTree` | `RegressionTree` | `compact` | `fitrtree` | `loss`

## Topics

“Predict Out-of-Sample Responses of Subtrees” on page 19-13

“Decision Trees” on page 19-2

“Prediction Using Classification and Regression Trees” on page 19-11

“Specify Variable-Sized Arguments for Code Generation” on page 31-32

“Code Generation Workflows” on page 31-2

**Introduced in R2011a**

# predict

**Class:** RegressionLinear

Predict response of linear regression model

## Syntax

```
YHat = predict(Mdl,X)
```

```
YHat = predict(Mdl,X,Name,Value)
```

## Description

`YHat = predict(Mdl,X)` returns predicted responses for each observation in the predictor data `X` based on the trained linear regression model `Mdl`. `YHat` contains responses for each regularization strength in `Mdl`.

`YHat = predict(Mdl,X,Name,Value)` returns predicted responses with additional options specified by one or more `Name,Value` pair arguments. For example, specify that columns in the predictor data correspond to observations.

## Input Arguments

### **mdl** — Linear regression model

RegressionLinear model object

Linear regression model, specified as a RegressionLinear model object. You can create a RegressionLinear model object using `fitrlinear`.

### **x** — Predictor data

full matrix | sparse matrix

Predictor data, specified as an  $n$ -by- $p$  full or sparse matrix. This orientation of `X` indicates that rows correspond to individual observations, and columns correspond to individual predictor variables.



---

**Note** If you orient your predictor matrix so that observations correspond to columns and specify 'ObservationsIn', 'columns', then you might experience a significant reduction in computation time.

---

The length of  $Y$  and the number of observations in  $X$  must be equal.

Data Types: `single` | `double`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### ObservationsIn — Predictor data observation dimension

'rows' (default) | 'columns'

Predictor data observation dimension, specified as the comma-separated pair consisting of 'ObservationsIn' and 'columns' or 'rows'.

---

**Note** If you orient your predictor matrix so that observations correspond to columns and specify 'ObservationsIn', 'columns', then you might experience a significant reduction in optimization-execution time.

---

Data Types: `char`

## Output Arguments

### $\hat{Y}$ — Predicted responses

numeric matrix

Predicted responses, returned as a  $n$ -by- $L$  numeric matrix.  $n$  is the number of observations in  $X$  and  $L$  is the number of regularization strengths in `Mdl.Lambda`.  $\hat{Y}(i, j)$  is the response for observation  $i$  using the linear regression model that has regularization strength `Mdl.Lambda(j)`.

The predicted response using the model with regularization strength  $j$  is  $\hat{y}_j = x\beta_j + b_j$ .

- $x$  is an observation from the predictor data matrix  $X$ , and is row vector.
- $\beta_j$  is the estimated column vector of coefficients. The software stores this vector in `Mdl.Beta(:,j)`.
- $b_j$  is the estimated, scalar bias, which the software stores in `Mdl.Bias(j)`.

## Examples

### Predict Test-Sample Responses

Simulate 10000 observations from this model

$$y = x_{100} + 2x_{200} + e.$$

- $X = x_1, \dots, x_{1000}$  is a 10000-by-1000 sparse matrix with 10% nonzero standard normal elements.
- $e$  is random normal error with mean 0 and standard deviation 0.3.

```
rng(1) % For reproducibility
n = 1e4;
d = 1e3;
nz = 0.1;
X = sprandn(n,d,nz);
Y = X(:,100) + 2*X(:,200) + 0.3*randn(n,1);
```

Train a linear regression model. Reserve 30% of the observations as a holdout sample.

```
CVMDL = fitrlinear(X,Y,'Holdout',0.3);
Mdl = CVMDL.Trained{1}
```

```
Mdl =

    RegressionLinear
        ResponseName: 'Y'
    ResponseTransform: 'none'
                Beta: [1000x1 double]
```

```

Bias: -0.0066
Lambda: 1.4286e-04
Learner: 'svm'

```

CVMdl is a `RegressionPartitionedLinear` model. It contains the property `Trained`, which is a 1-by-1 cell array holding a `RegressionLinear` model that the software trained using the training set.

Extract the training and test data from the partition definition.

```

trainIdx = training(CVMdl.Partition);
testIdx = test(CVMdl.Partition);

```

Predict the training- and test-sample responses.

```

yHatTrain = predict(Mdl,X(trainIdx,:));
yHatTest = predict(Mdl,X(testIdx,:));

```

Because there is one regularization strength in `Mdl`, `yHatTrain` and `yHatTest` are numeric vectors.

### Predict from Best-Performing Model

Predict responses from the best-performing, linear regression model that uses a lasso-penalty and least squares.

Simulate 10000 observations as in “Predict Test-Sample Responses” on page 32-5020.

```

rng(1) % For reproducibility
n = 1e4;
d = 1e3;
nz = 0.1;
X = sprandn(n,d,nz);
Y = X(:,100) + 2*X(:,200) + 0.3*randn(n,1);

```

Create a set of 15 logarithmically-spaced regularization strengths from  $10^{-5}$  through  $10^{-1}$ .

```

Lambda = logspace(-5,-1,15);

```

Cross-validate the models. To increase execution speed, transpose the predictor data and specify that the observations are in columns. Optimizing the objective function using SpaRSA.

```
X = X';
CVMdl = fitrlinear(X,Y,'ObservationsIn','columns','KFold',5,'Lambda',Lambda,...
    'Learner','leastsquares','Solver','sparsa','Regularization','lasso');
```

```
numCLModels = numel(CVMdl.Trained)
```

```
numCLModels =
```

```
5
```

CVMdl is a `RegressionPartitionedLinear` model. Because `fitrlinear` implements 5-fold cross-validation, CVMdl contains 5 `RegressionLinear` models that the software trains on each fold.

Display the first trained linear regression model.

```
Mdl1 = CVMdl.Trained{1}
```

```
Mdl1 =
```

```
RegressionLinear
  ResponseName: 'Y'
  ResponseTransform: 'none'
      Beta: [1000x15 double]
      Bias: [1x15 double]
      Lambda: [1x15 double]
  Learner: 'leastsquares'
```

Mdl1 is a `RegressionLinear` model object. `fitrlinear` constructed Mdl1 by training on the first four folds. Because `Lambda` is a sequence of regularization strengths, you can think of Mdl1 as 11 models, one for each regularization strength in `Lambda`.

Estimate the cross-validated MSE.

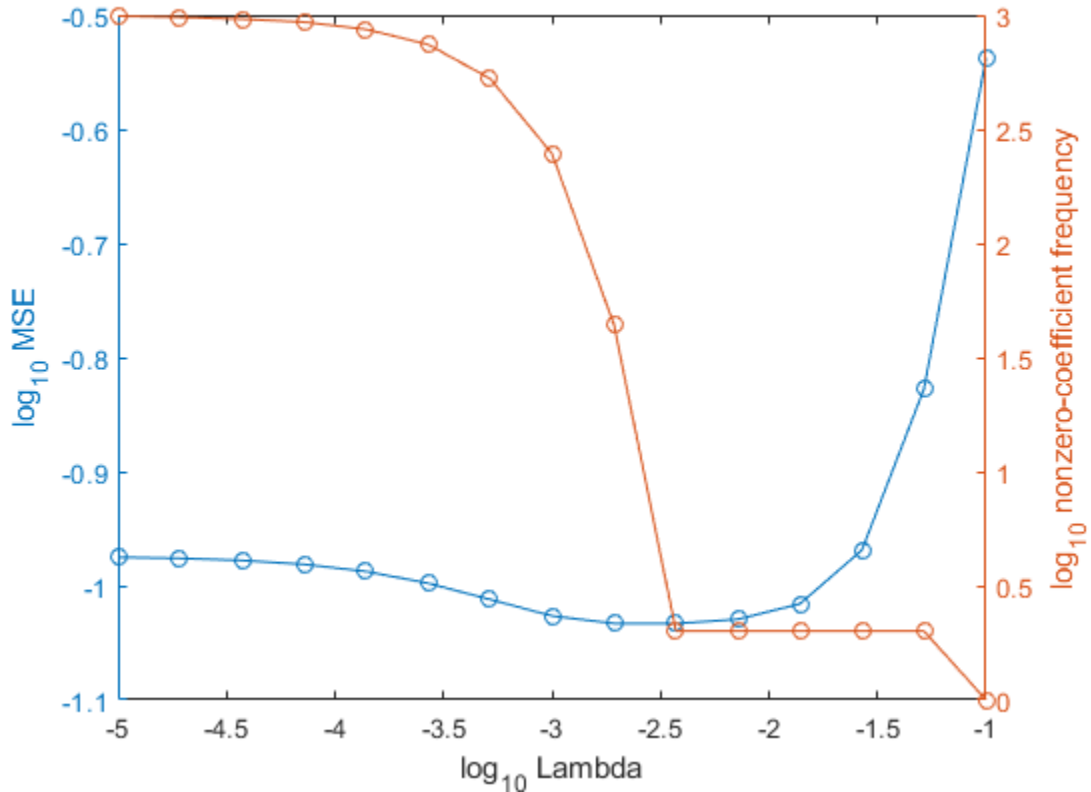
```
mse = kfoldLoss(CVMdl);
```

Higher values of `Lambda` lead to predictor variable sparsity, which is a good quality of a regression model. For each regularization strength, train a linear regression model using the entire data set and the same options as when you cross-validated the models. Determine the number of nonzero coefficients per model.

```
Mdl = fitrlinear(X,Y, 'ObservationsIn', 'columns', 'Lambda', Lambda, ...  
    'Learner', 'leastsquares', 'Solver', 'sparsa', 'Regularization', 'lasso');  
numNZCoeff = sum(Mdl.Beta~=0);
```

In the same figure, plot the cross-validated MSE and frequency of nonzero coefficients for each regularization strength. Plot all variables on the log scale.

```
figure;  
[h,hL1,hL2] = plotyy(log10(Lambda), log10(mse), ...  
    log10(Lambda), log10(numNZCoeff));  
hL1.Marker = 'o';  
hL2.Marker = 'o';  
ylabel(h(1), 'log_{10} MSE')  
ylabel(h(2), 'log_{10} nonzero-coefficient frequency')  
xlabel('log_{10} Lambda')  
hold off
```



Choose the index of the regularization strength that balances predictor variable sparsity and low MSE (for example, `Lambda(10)`).

```
idxFinal = 10;
```

Extract the model with corresponding to the minimal MSE.

```
MdlFinal = selectModels(Mdl,idxFinal)
idxNZCoeff = find(MdlFinal.Beta~=0)
EstCoeff = Mdl.Beta(idxNZCoeff)
```

```
MdlFinal =
```

```
RegressionLinear
  ResponseName: 'Y'
  ResponseTransform: 'none'
    Beta: [1000x1 double]
    Bias: -0.0050
    Lambda: 0.0037
  Learner: 'leastsquares'

idxNZCoeff =

    100
    200

EstCoeff =

    1.0051
    1.9965
```

`MdlFinal` is a `RegressionLinear` model with one regularization strength. The nonzero coefficients `EstCoeff` are close to the coefficients that simulated the data.

Simulate 10 new observations, and predict corresponding responses using the best-performing model.

```
XNew = sprandn(d,10,nz);
YHat = predict(MdlFinal,XNew,'ObservationsIn','columns');
```

## Extended Capabilities

### Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays” (MATLAB).

## C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

Argument	Notes and Limitations
Mdl	<ul style="list-style-type: none"> <li>• Save a trained model by using <code>saveCompactModel</code>, and then load the model by using <code>loadCompactModel</code> within a function that you declare.</li> <li>• Must be a compile-time constant; its value cannot change in the generated code.</li> <li>• For more details, see “Code Generation” on page 32-0 of the <code>RegressionLinear</code> class.</li> </ul>
X	<ul style="list-style-type: none"> <li>• Must be a single- or double-precision matrix and can be variable-sized.</li> <li>• If you specify <code>'ObservationsIn', 'rows'</code> (default), then the number of columns in X must be <code>numel(Mdl.PredictorNames)</code>. Rows and columns must correspond to observations and predictors, respectively.</li> <li>• If you specify <code>'ObservationsIn', 'columns'</code>, then the number of rows in X must be <code>numel(Mdl.PredictorNames)</code>. Rows and columns must correspond to predictors and observations, respectively.</li> </ul>
Name-value pair arguments	<p>Names in name-value pair arguments must be compile-time constants. For example, to use the <code>'ObservationsIn'</code> name-value pair argument in the generated code, include <code>{coder.Constant('ObservationsIn'), coder.typeof('a', [1, 7], [0, 1])}</code> in the <code>-args</code> value of <code>codegen</code>.</p>

## See Also

`RegressionLinear` | `fitrlinear`

## Topics

“Code Generation Workflows” on page 31-2



Introduced in R2016a

## predict

**Class:** CompactTreeBagger

Predict responses using ensemble of bagged decision trees

## Syntax

```
Yfit = predict(B,X)
[Yfit,stdevs] = predict(B,X)
[Yfit,scores] = predict(B,X)
[Yfit,scores,stdevs] = predict(B,X)
Yfit = predict(B,X,'param1',val1,'param2',val2,...)
```

## Description

`Yfit = predict(B,X)` returns a vector of predicted responses for the predictor data in the table or matrix `X`, based on the compact ensemble of bagged decision trees `B`. `Yfit` is a cell array of character vectors for classification and a numeric array for regression. By default, `predict` takes a democratic (nonweighted) average vote from all trees in the ensemble.

`B` is a trained `CompactTreeBagger` model object, that is, a model returned by `compact`.

`X` is a table or matrix of predictor data used to generate responses. Rows represent observations and columns represent variables.

- If `X` is a numeric matrix:
  - The variables making up the columns of `X` must have the same order as the predictor variables that trained `B`.
  - If you trained `B` using a table (for example, `Tbl`), then `X` can be a numeric matrix if `Tbl` contains all numeric predictor variables. To treat numeric predictors in `Tbl` as categorical during training, identify categorical predictors using the `CategoricalPredictors` name-value pair argument of `TreeBagger`. If `Tbl`

contains heterogeneous predictor variables (for example, numeric and categorical data types) and `X` is a numeric matrix, then `predict` throws an error.

- If `X` is a table:
  - `predict` does not support multi-column variables and cell arrays other than cell arrays of character vectors.
  - If you trained `B` using a table (for example, `Tbl`), then all predictor variables in `X` must have the same variable names and be of the same data types as those that trained `B` (stored in `B.PredictorNames`). However, the column order of `X` does not need to correspond to the column order of `Tbl`. `Tbl` and `X` can contain additional variables (response variables, observation weights, etc.), but `predict` ignores them.
  - If you trained `B` using a numeric matrix, then the predictor names in `B.PredictorNames` and corresponding predictor variable names in `X` must be the same. To specify predictor names during training, see the `PredictorNames` name-value pair argument of `TreeBagger`. All predictor variables in `X` must be numeric vectors. `X` can contain additional variables (response variables, observation weights, etc.), but `predict` ignores them.

For regression, `[Yfit, stdevs] = predict(B, X)` also returns standard deviations of the computed responses over the ensemble of the grown trees.

For classification, `[Yfit, scores] = predict(B, X)` also returns scores for all classes. `scores` is a matrix with one row per observation and one column per class. For each observation and each class, the score generated by each tree is the probability of this observation originating from this class computed as the fraction of observations of this class in a tree leaf. `predict` averages these scores over all trees in the ensemble.

`[Yfit, scores, stdevs] = predict(B, X)` also returns standard deviations of the computed scores for classification. `stdevs` is a matrix with one row per observation and one column per class, with standard deviations taken over the ensemble of the grown trees.

`Yfit = predict(B, X, 'param1', val1, 'param2', val2, ...)` specifies optional parameter name/value pairs:

'Trees'	Array of tree indices to use for computation of responses. Default is 'all'.
---------	--

'TreeWeights'	Array of NTrees weights for weighting votes from the specified trees.
'UseInstanceForTree'	Logical matrix of size Nobs-by-NTrees indicating which trees to use to make predictions for each observation. By default all trees are used for all observations.

## Algorithms

- For regression problems, the predicted response for an observation is the weighted average of the predictions using selected trees only. That is,

$$\hat{y}_{\text{bag}} = \frac{1}{\sum_{t=1}^T \alpha_t I(t \in S)} \sum_{t=1}^T \alpha_t \hat{y}_t I(t \in S).$$

- $\hat{y}_t$  is the prediction from tree  $t$  in the ensemble.
  - $S$  is the set of indices of selected trees that comprise the prediction (see 'Trees' and 'UseInstanceForTree').  $I(t \in S)$  is 1 if  $t$  is in the set  $S$ , and 0 otherwise.
  - $\alpha_t$  is the weight of tree  $t$  (see 'TreeWeights').
- For classification problems, the predicted class for an observation is the class that yields the largest weighted average of the class posterior probabilities (i.e., classification scores) computed using selected trees only. That is,

1

For each class  $c \in C$  and each tree  $t = 1, \dots, T$ , `predict` computes  $\hat{P}_t(c|x)$ , which is the estimated posterior probability of class  $c$  given observation  $x$  using tree  $t$ .  $C$  is the set of all distinct classes in the training data. For more details on classification tree posterior probabilities, see `fitctree` and `predict`.

- 2 `predict` computes the weighted average of the class posterior probabilities over the selected trees.

$$\hat{P}_{\text{bag}}(c|x) = \frac{1}{\sum_{t=1}^T \alpha_t I(t \in S)} \sum_{t=1}^T \alpha_t \hat{P}_t(c|x) I(t \in S).$$

- 3** The predicted class is the class that yields the largest weighted average.

$$\hat{y}_{\text{bag}} = \arg \max_{c \in C} \{ \hat{P}_{\text{bag}}(c | x) \}.$$

## See Also

`CompactTreeBagger` | `error`

## Topics

“Bootstrap Aggregation (Bagging) of Regression Trees” on page 18-141

“Bootstrap Aggregation (Bagging) of Classification Trees” on page 18-154

## predict

**Class:** CompactGeneralizedLinearModel

Predict response of generalized linear regression model

### Syntax

```
ypred = predict mdl, Xnew  
[ypred, yci] = predict mdl, Xnew  
[ypred, yci] = predict mdl, Xnew, Name, Value
```

### Description

`ypred = predict mdl, Xnew` returns the predicted response of the `mdl` generalized linear regression model to the points in `Xnew`.

`[ypred, yci] = predict mdl, Xnew` returns confidence intervals for the true mean responses.

`[ypred, yci] = predict mdl, Xnew, Name, Value` predicts responses with additional options specified by one or more `Name, Value` pair arguments.

### Input Arguments

#### **mdl**

Generalized linear model, specified as a full `GeneralizedLinearModel` object constructed using `fitglm` or `stepwiseglm`, or a compacted `CompactGeneralizedLinearModel` object constructed using `compact`.

#### **Xnew**

Points at which `mdl` predicts responses.

- If `Xnew` is a table or dataset array, it must contain the predictor names in `mdl`.
- If `Xnew` is a numeric matrix, it must have the same number of variables (columns) as was used to create `mdl`. Furthermore, all variables used in creating `mdl` must be numeric.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### Alpha

Positive scalar from 0 to 1. Confidence level of  $y_{ci}$  is  $100(1 - \text{alpha})\%$ .

**Default:** 0.05, meaning a 95% confidence interval.

### BinomialSize

Value of the binomial  $n$  parameter for each row in the training data. `BinomialSize` can be a vector the same length as `Xnew`, or a scalar that applies to each row. The default value 1 produces `ypred` values that are predicted proportions. Use `BinomialSize` only if `mdl` is fit to a binomial distribution.

**Default:** 1

### Offset

Value of the offset for each row in `Xnew`. `Offset` can be a vector the same length as `Xnew`, or a scalar that applies to each row. The offset is used as an additional predictor with a coefficient value fixed at 1. In other words, if `b` is the fitted coefficient vector, and `link` is the link function,

$$\text{link}(\text{ypred}) = \text{Offset} + \text{Xnew} * \text{b}.$$

**Default:** `zeros(size(Xnew,1))`

### Simultaneous

Logical value specifying whether the confidence bounds are for all predictor values simultaneously (`true`), or hold for each individual predictor value (`false`). `Simultaneous`

bounds are wider than separate bounds, because it is more stringent to require that the entire curve be within the bounds than to require that the curve at a single predictor value be within the bounds.

For details, see `polyconf`.

**Default:** `false`

## Output Arguments

**`ypred`**

Vector of predicted mean values at `Xnew`.

**`yci`**

Confidence intervals, a two-column matrix with each row providing one interval. The meaning of the confidence interval depends on the settings of the name-value pairs.

## Examples

### Generalized Linear Model Predictions

Create a generalized linear model, and predict its response to new data.

Generate artificial data for the model using Poisson random numbers with two underlying predictors `X(1)` and `X(2)`.

```
rng('default') % For reproducibility
rndvars = randn(100,2);
X = [2+rndvars(:,1), rndvars(:,2)];
mu = exp(1 + X*[1;2]);
y = poissrnd(mu);
```

Create a generalized linear regression model of Poisson data.

```
mdl = fitglm(X,y,'y ~ x1 + x2','distr','poisson');
```

Create points for prediction.



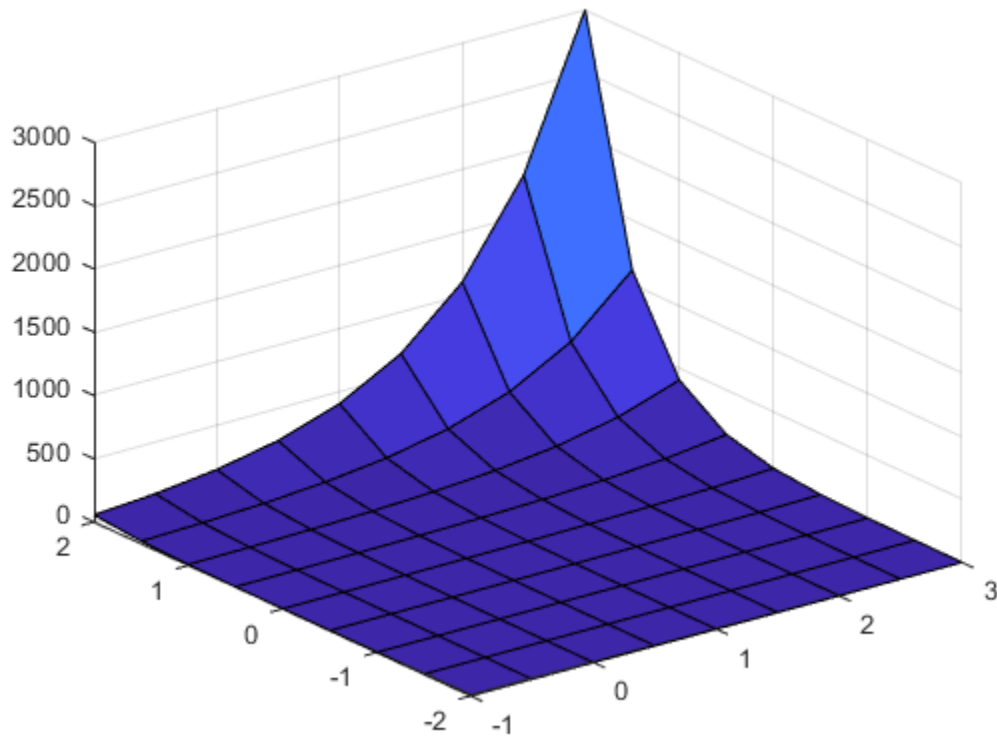
```
[Xtest1 Xtest2] = meshgrid(-1:.5:3,-2:.5:2);  
Xnew = [Xtest1(:),Xtest2(:)];
```

Predict responses at the new points.

```
ypred = predict(mdl,Xnew);
```

Plot the predictions.

```
surf(Xtest1,Xtest2,reshape(ypred,9,9))
```



Create confidence intervals on the predictions.

```
[ypred yci] = predict mdl, Xnew;
```

### Generate Code from Function That Predicts Responses Given New Data

Train a generalized linear model, and then generate code from a function that classifies new observations based on the model. This example is based on the “Generalized Linear Model Predictions” on page 32-5034 example.

Generate artificial data.

```
rng('default') % For reproducibility
rndvars = randn(100,2);
X = [2+rndvars(:,1), rndvars(:,2)];
mu = exp(1 + X*[1;2]);
y = poissrnd(mu);
```

Create a generalized linear regression model of Poisson data.

```
mdl = fitglm(X,y,'y ~ x1 + x2','Distribution','poisson');
```

Create points for prediction.

```
[Xtest1 Xtest2] = meshgrid(-1:.5:3,-2:.5:2);
Xnew = [Xtest1(:),Xtest2(:)];
```

Save the fitted generalized linear model to the file GLMMdl.mat.

```
saveCompactModel(mdl,'GLMMdl');
```

Declare a function in your current working folder called mypredictGLM.m that:

- Accepts measurements with columns corresponding to those in Xnew and valid name-value pair arguments
- Loads the fitted generalized linear model in GLMMdl.mat
- Returns predictions and confidence interval bounds

```
function [yhat,ci] = mypredictGLM(x,varargin) %#codegen
%MPREDICTGLM Predict response using GLM model
%   MYPREDICTGLM predicts responses for the n observations in the n-by-1
%   vector x using using the GLM model stored in the MAT-file GLMMdl.mat,
%   and then returns the predictions in the n-by-1 vector yhat.
```

```

% MYPREDICTGLM also returns confidence interval bounds for the
% predictions in the n-by-2 vector ci.
CompactMdl = loadCompactModel('GLMMdl');
narginchk(1,Inf);
[yhat,ci] = predict(CompactMdl,x,varargin{:});
end

```

Generate a MEX function from `mypredictGLM.m`. Specify returning 90% simultaneous confidence intervals on the predictions. Because C uses static typing, `codegen` must determine the properties of all variables in MATLAB® files at compile time. To designate arguments as compile-time constants, use `coder.Constant`.

```
codegen -config:mex mypredictGLM -args {Xnew,coder.Constant('Alpha'),0.1,coder.Constant('Simultaneous'),true};
```

The MEX file `mypredictGLM_mex.mexw64` generates to your present working directory. The file extension depends on your platform.

Compare predictions using `predict` and `mypredictGLM_mex`. Specify name-value pair arguments in the same order as in the `-args` argument in the call to `codegen`.

```

[yhat1,ci1] = predict(mdl,Xnew,'Alpha',0.1,'Simultaneous',true);
[yhat2,ci2] = mypredictGLM_mex(Xnew,'Alpha',0.1,'Simultaneous',true);

comp1 = (yhat1 - yhat2)'*(yhat1 - yhat2);
comp2 = diag((ci1 - ci2)'*(ci1 - ci2));
agree1 = comp1 < eps
agree2 = comp2 < eps

```

```
agree1 =
```

```
logical
```

```
1
```

```
agree2 =
```

```
2x1 logical array
```

```
1
```

```
1
```

The generated MEX file produces the same results as `predict`.

- “Predict or Simulate Responses to New Data” on page 12-35
- “Generalized Linear Model Workflow” on page 12-41

## Tips

- For predictions with added noise, use `random`.
- For a syntax that can be easier to use with models created from dataset arrays, try `feval`.

## Alternatives

`feval` gives the same predictions, but uses separate input arrays for each predictor, instead of one input array containing all predictors.

`random` predicts with added noise.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

Argument	Notes and Limitations
<code>mdl</code>	<ul style="list-style-type: none"><li>• Save a trained model by using <code>saveCompactModel</code>, and then load the model by using <code>loadCompactModel</code> within a function that you declare.</li><li>• Must be a compile-time constant; its value cannot change in the generated code.</li><li>• For more details, see “Code Generation” on page 32-0 of the <code>CompactGeneralizedLinearModel</code> class.</li></ul>

Argument	Notes and Limitations
Xnew	<ul style="list-style-type: none"> <li>• Must be a single- or double-precision matrix and can be variable-sized. However, the number of columns in Xnew must be <code>numel mdl.PredictorNames</code>.</li> <li>• Rows and columns must correspond to observations and predictors, respectively.</li> </ul>
Name-value pair arguments	Names in name-value pair arguments must be compile-time constants. For example, to allow a user-defined significance level in the generated code, include <code>{coder.Constant('Alpha'),0}</code> in the <code>-args</code> value of <code>codegen</code> .

## See Also

`CompactGeneralizedLinearModel` | `GeneralizedLinearModel` | `fitglm` | `random`

## Topics

“Predict or Simulate Responses to New Data” on page 12-35

“Generalized Linear Model Workflow” on page 12-41

“Generalized Linear Models” on page 12-12

“Code Generation Workflows” on page 31-2

Introduced in R2012a

## predict

**Class:** GeneralizedLinearMixedModel

Predict response of generalized linear mixed-effects model

## Syntax

```
ypred = predict(glme)
ypred = predict(glme, tblnew)
ypred = predict( ____, Name, Value)
[ypred, ypredCI] = predict( ____)
[ypred, ypredCI, DF] = predict( ____)
```

## Description

`ypred = predict(glme)` returns the predicted conditional means of the response, `ypred`, using the original predictor values used to fit the generalized linear mixed-effects model `glme`.

`ypred = predict(glme, tblnew)` returns the predicted conditional means using the new predictor values specified in `tblnew`.

If a grouping variable in `tblnew` has levels that are not in the original data, then the random effects for that grouping variable do not contribute to the 'Conditional' prediction at observations where the grouping variable has new levels.

`ypred = predict( ____, Name, Value)` returns the predicted conditional means of the response using additional options specified by one or more `Name, Value` pair arguments. For example, you can specify the confidence level, simultaneous confidence bounds, or contributions from only fixed effects. You can use any of the input arguments in the previous syntaxes.

`[ypred, ypredCI] = predict( ____)` also returns 95% point-wise confidence intervals, `ypredCI`, for each predicted value.

[ypred, ypredCI, DF] = predict( \_\_\_ ) also returns the degrees of freedom, DF, used to compute the confidence intervals.

## Input Arguments

### **glme** — Generalized linear mixed-effects model

GeneralizedLinearMixedModel object

Generalized linear mixed-effects model, specified as a GeneralizedLinearMixedModel object. For properties and methods of this object, see GeneralizedLinearMixedModel.

### **tblnew** — New input data

table | dataset array

New input data, which includes the response variable, predictor variables, and grouping variables on page 2-59, specified as a table or dataset array. The predictor variables can be continuous or grouping variables. tblnew must have the same variables as the original table or dataset array used in fitglme to fit the generalized linear mixed-effects model glme.

Data Types: single | double | logical | char

## Name-Value Pair Arguments

Specify optional comma-separated pairs of Name, Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as Name1, Value1, ..., NameN, ValueN.

### **Alpha** — Confidence level

0.05 (default) | scalar value in the range [0,1]

Confidence level, specified as the comma-separated pair consisting of 'Alpha' and a scalar value in the range [0,1]. For a value  $\alpha$ , the confidence level is  $100 \times (1 - \alpha)\%$ .

For example, for 99% confidence intervals, you can specify the confidence level as follows.

Example: 'Alpha', 0.01

Data Types: single | double

**Conditional** — Indicator for conditional predictions`true` (default) | `false`

Indicator for conditional predictions, specified as the comma-separated pair consisting of 'Conditional' and one of the following.

Value	Description
<code>true</code>	Contributions from both fixed effects and random effects (conditional)
<code>false</code>	Contribution from only fixed effects (marginal)

Example: `'Conditional', false`

**DFMethod** — Method for computing approximate degrees of freedom`'residual'` (default) | `'none'`

Method for computing approximate degrees of freedom, specified as the comma-separated pair consisting of 'DFMethod' and one of the following.

Value	Description
<code>'residual'</code>	The degrees of freedom are assumed to be constant and equal to $n - p$ , where $n$ is the number of observations and $p$ is the number of fixed effects.
<code>'none'</code>	All degrees of freedom are set to infinity.

Example: `'DFMethod', 'none'`

**Offset** — Model offset`zeros(m, 1)` (default) |  $m$ -by-1 vector of scalar values

Model offset, specified as a vector of scalar values of length  $m$ , where  $m$  is the number of rows in `tblnew`. The offset is used as an additional predictor and has a coefficient value fixed at 1.

**Simultaneous** — Type of confidence bounds`false` (default) | `true`

Type of confidence bounds, specified as the comma-separated pair consisting of 'Simultaneous' and either `false` or `true`.



- If 'Simultaneous' is false, then `predict` computes nonsimultaneous confidence bounds.
- If 'Simultaneous' is true, `predict` returns simultaneous confidence bounds.

Example: `'Simultaneous', true`

## Output Arguments

### **`ypred` — Predicted responses**

vector

Predicted responses, returned as a vector. If the 'Conditional' name-value pair argument is specified as `true`, `ypred` contains predictions for the conditional means of the responses given the random effects. Conditional predictions include contributions from both fixed and random effects. Marginal predictions include only contributions from fixed effects.

To compute marginal predictions, `predict` computes conditional predictions, but substitutes a vector of zeros in place of the empirical Bayes predictors (EBPs) of the random effects.

### **`ypredCI` — Point-wise confidence intervals**

two-column matrix

Point-wise confidence intervals for the predicted values, returned as a two-column matrix. The first column of `ypredCI` contains the lower bound, and the second column contains the upper bound. By default, `ypredCI` contains the 95% nonsimultaneous confidence intervals for the predictions. You can change the confidence level using the `Alpha` name-value pair argument, and make them simultaneous using the `Simultaneous` name-value pair argument.

When fitting a GLME model using `fitglme` and one of the maximum likelihood fit methods ('Laplace' or 'ApproximateLaplace'), `predict` computes the confidence intervals using the conditional mean squared error of prediction (CMSEP) approach conditional on the estimated covariance parameters and the observed response. Alternatively, you can interpret the confidence intervals as approximate Bayesian credible intervals conditional on the estimated covariance parameters and the observed response.

When fitting a GLME model using `fitglme` and one of the pseudo likelihood fit methods ('MPL' or 'REMP'), `predict` bases the computations on the fitted linear mixed-effects model from the final pseudo likelihood iteration.

**DF — Degrees of freedom**

vector | scalar value

Degrees of freedom used in computing the confidence intervals, returned as a vector or a scalar value.

- If 'Simultaneous' is false, then DF is a vector.
- If 'Simultaneous' is true, then DF is a scalar value.

## Examples

### Predict Responses at Original Design Values

Load the sample data.

```
load mfr
```

This simulated data is from a manufacturing company that operates 50 factories across the world, with each factory running a batch process to create a finished product. The company wants to decrease the number of defects in each batch, so it developed a new manufacturing process. To test the effectiveness of the new process, the company selected 20 of its factories at random to participate in an experiment: Ten factories implemented the new process, while the other ten continued to run the old process. In each of the 20 factories, the company ran five batches (for a total of 100 batches) and recorded the following data:

- Flag to indicate whether the batch used the new process (`newprocess`)
- Processing time for each batch, in hours (`time`)
- Temperature of the batch, in degrees Celsius (`temp`)
- Categorical variable indicating the supplier (A, B, or C) of the chemical used in the batch (`supplier`)
- Number of defects in the batch (`defects`)

The data also includes `time_dev` and `temp_dev`, which represent the absolute deviation of time and temperature, respectively, from the process standard of 3 hours at 20 degrees Celsius.

Fit a generalized linear mixed-effects model using `newprocess`, `time_dev`, `temp_dev`, and `supplier` as fixed-effects predictors. Include a random-effects term for intercept grouped by `factory`, to account for quality differences that might exist due to factory-specific variations. The response variable `defects` has a Poisson distribution, and the appropriate link function for this model is log. Use the Laplace fit method to estimate the coefficients. Specify the dummy variable encoding as 'effects', so the dummy variable coefficients sum to 0.

The number of defects can be modeled using a Poisson distribution:

$$\text{defects}_{ij} \sim \text{Poisson}(\mu_{ij})$$

This corresponds to the generalized linear mixed-effects model

$$\log(\mu_{ij}) = \beta_0 + \beta_1 \text{newprocess}_{ij} + \beta_2 \text{time.dev}_{ij} + \beta_3 \text{temp.dev}_{ij} + \beta_4 \text{supplier.C}_{ij} + \beta_5 \text{supplier.B}_{ij} + b_i,$$

where

- `defectsij` is the number of defects observed in the batch produced by factory *i* during batch *j*.
- $\mu_{ij}$  is the mean number of defects corresponding to factory *i* (where  $i = 1, 2, \dots, 20$ ) during batch *j* (where  $j = 1, 2, \dots, 5$ ).
- `newprocessij`, `time.devij`, and `temp.devij` are the measurements for each variable that correspond to factory *i* during batch *j*. For example, `newprocessij` indicates whether the batch produced by factory *i* during batch *j* used the new process.
- `supplier.Cij` and `supplier.Bij` are dummy variables that use effects (sum-to-zero) coding to indicate whether company C or B, respectively, supplied the process chemicals for the batch produced by factory *i* during batch *j*.
- $b_i \sim N(0, \sigma_b^2)$  is a random-effects intercept for each factory *i* that accounts for factory-specific variation in quality.

```
glme = fitglme(mfr, 'defects ~ 1 + newprocess + time_dev + temp_dev + supplier + (1|fact
```

Predict the response values at the original design values. Display the first ten predictions along with the observed response values.

```
ypred = predict(glme);  
[ypred(1:10),mfr.defects(1:10)]
```

```
ans =
```

```
    4.9883    6.0000  
    5.9423    7.0000  
    5.1318    6.0000  
    5.6295    5.0000  
    5.3499    6.0000  
    5.2134    5.0000  
    4.6430    4.0000  
    4.5342    4.0000  
    5.3903    9.0000  
    4.6529    4.0000
```

Column 1 contains the predicted response values at the original design values. Column 2 contains the observed response values.

### Predict Responses at Values in New Table

Load the sample data.

```
load mfr
```

This simulated data is from a manufacturing company that operates 50 factories across the world, with each factory running a batch process to create a finished product. The company wants to decrease the number of defects in each batch, so it developed a new manufacturing process. To test the effectiveness of the new process, the company selected 20 of its factories at random to participate in an experiment: Ten factories implemented the new process, while the other ten continued to run the old process. In each of the 20 factories, the company ran five batches (for a total of 100 batches) and recorded the following data:

- Flag to indicate whether the batch used the new process (`newprocess`)
- Processing time for each batch, in hours (`time`)
- Temperature of the batch, in degrees Celsius (`temp`)
- Categorical variable indicating the supplier (A, B, or C) of the chemical used in the batch (`supplier`)

- Number of defects in the batch (`defects`)

The data also includes `time_dev` and `temp_dev`, which represent the absolute deviation of time and temperature, respectively, from the process standard of 3 hours at 20 degrees Celsius.

Fit a generalized linear mixed-effects model using `newprocess`, `time_dev`, `temp_dev`, and `supplier` as fixed-effects predictors. Include a random-effects term for intercept grouped by `factory`, to account for quality differences that might exist due to factory-specific variations. The response variable `defects` has a Poisson distribution, and the appropriate link function for this model is `log`. Use the Laplace fit method to estimate the coefficients. Specify the dummy variable encoding as `'effects'`, so the dummy variable coefficients sum to 0.

The number of defects can be modeled using a Poisson distribution:

$$\text{defects}_{ij} \sim \text{Poisson}(\mu_{ij})$$

This corresponds to the generalized linear mixed-effects model

$$\log(\mu_{ij}) = \beta_0 + \beta_1 \text{newprocess}_{ij} + \beta_2 \text{time\_dev}_{ij} + \beta_3 \text{temp\_dev}_{ij} + \beta_4 \text{supplier\_C}_{ij} + \beta_5 \text{supplier\_B}_{ij} + b_i,$$

where

- `defectsij` is the number of defects observed in the batch produced by factory *i* during batch *j*.
- $\mu_{ij}$  is the mean number of defects corresponding to factory *i* (where  $i = 1, 2, \dots, 20$ ) during batch *j* (where  $j = 1, 2, \dots, 5$ ).
- `newprocessij`, `time_devij`, and `temp_devij` are the measurements for each variable that correspond to factory *i* during batch *j*. For example, `newprocessij` indicates whether the batch produced by factory *i* during batch *j* used the new process.
- `supplier_Cij` and `supplier_Bij` are dummy variables that use effects (sum-to-zero) coding to indicate whether company C or B, respectively, supplied the process chemicals for the batch produced by factory *i* during batch *j*.
- $b_i \sim N(0, \sigma_b^2)$  is a random-effects intercept for each factory *i* that accounts for factory-specific variation in quality.

```
glme = fitglme(mfr, 'defects ~ 1 + newprocess + time_dev + temp_dev + supplier + (1|fact
```

Predict the response values at the original design values.

```
ypred = predict(glme);
```

Create a new table by copying the first 10 rows of `mfr` into `tblnew`.

```
tblnew = mfr(1:10, :);
```

The first 10 rows of `mfr` include data collected from trials 1 through 5 for factories 1 and 2. Both factories used the old process for all of their trials during the experiment, so `newprocess = 0` for all 10 observations.

Change the value of `newprocess` to 1 for the observations in `tblnew`.

```
tblnew.newprocess = ones(height(tblnew), 1);
```

Compute predicted response values and nonsimultaneous 99% confidence intervals using `tblnew`. Display the first 10 rows of the predicted values based on `tblnew`, the predicted values based on `mfr`, and the observed response values.

```
[ypred_new, ypredCI] = predict(glme, tblnew, 'Alpha', 0.01);  
[ypred_new, ypred(1:10), mfr.defects(1:10)]
```

```
ans =
```

3.4536	4.9883	6.0000
4.1142	5.9423	7.0000
3.5530	5.1318	6.0000
3.8976	5.6295	5.0000
3.7040	5.3499	6.0000
3.6095	5.2134	5.0000
3.2146	4.6430	4.0000
3.1393	4.5342	4.0000
3.7320	5.3903	9.0000
3.2214	4.6529	4.0000

Column 1 contains predicted response values based on the data in `tblnew`, where `newprocess = 1`. Column 2 contains predicted response values based on the original data in `mfr`, where `newprocess = 0`. Column 3 contains the observed response values in `mfr`. Based on these results, if all other predictors retain their original values, the predicted number of defects appears to be smaller when using the new process.

Display the 99% confidence intervals for rows 1 through 10 corresponding to the new predicted response values.

```
ypredCI(1:10,1:2)
```

```
ans =
```

```
1.6983    7.0235
1.9191    8.8201
1.8735    6.7380
2.0149    7.5395
1.9034    7.2079
1.8918    6.8871
1.6776    6.1597
1.5404    6.3976
1.9574    7.1154
1.6892    6.1436
```

## References

- [1] Booth, J.G., and J.P. Hobert. “Standard Errors of Prediction in Generalized Linear Mixed Models.” *Journal of the American Statistical Association*, Vol. 93, 1998, pp. 262–272.

## See Also

`GeneralizedLinearMixedModel` | `fitglm` | `fitted` | `random`

## predict

**Class:** CompactLinearModel

Predict response of linear regression model

### Syntax

```
ypred = predict mdl, Xnew
[ypred, yci] = predict mdl, Xnew
[ypred, yci] = predict mdl, Xnew, Name, Value
```

### Description

`ypred = predict mdl, Xnew` returns the predicted response of the `mdl` linear regression model to the points in `Xnew`.

`[ypred, yci] = predict mdl, Xnew` returns confidence intervals for the true mean responses.

`[ypred, yci] = predict mdl, Xnew, Name, Value` predicts responses with additional options specified by one or more `Name, Value` pair arguments.

### Input Arguments

**mdl** — Linear model object

LinearModel object | CompactLinearModel object

Linear model object, specified as a full `LinearModel` object constructed using `fitlm` or `stepwiselm`, or a compacted `CompactLinearModel` object constructed using `compact`.

**xnew** — New predictor input values

table | dataset array | numeric matrix

New predictor input values, specified as a table, dataset array, or numeric matrix.



- If `Xnew` is a table or dataset array, it must contain the predictor names in `mdl`.
- If `Xnew` is a numeric matrix, it must have the same number of variables (columns) as was used to create `mdl`. Furthermore, all variables used in creating `mdl` must be numeric.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### **Alpha** — Alpha value for confidence interval

0.05 (default) | numeric value in the range [0,1]

Alpha value for confidence interval, specified as the comma-separated pair consisting of 'Alpha' and a numeric value in the range [0,1]. The confidence level of `yci` is equal to  $100(1-\alpha)\%$ . For example, specifying `Alpha` as 0.05 corresponds to a 95% confidence interval.

### **Prediction** — Prediction type

'curve' (default) | 'observation'

Prediction type, specified as the comma-separated pair consisting of 'Prediction' and one of the following.

- 'curve' — `predict` predicts confidence bounds for the fitted mean values.
- 'observation' — `predict` predicts confidence bounds for the new observations. This results in wider bounds because the error in a new observation is equal to the error in the estimated mean value, plus the variability in the observation from the true mean.

For details, see `polyconf`.

### **Simultaneous** — Flag to compute simultaneous confidence bounds

false (default) | true

Flag to compute simultaneous confidence bounds, specified as the comma-separated pair consisting of `Simultaneous` and either false or true. If you specify `true`, then the function

computes confidence bounds are for all predictor values simultaneously. If you specify `false`, then the function calculates confidence bounds using a `hold` for each individual predictor value. Simultaneous bounds are wider than separate bounds, because it is more stringent to require that the entire curve be within the bounds than to require that the curve at a single predictor value be within the bounds.

For details, see `polyconf`.

## Output Arguments

### **`ypred` — Predicted mean response values**

numeric vector

Predicted mean response values evaluated at `Xnew`, returned as a numeric vector. `ypred` is the same size as `Xnew`.

### **`yci` — Confidence intervals for predicted responses**

two-column numeric matrix

Confidence intervals for predicted responses, returned as a two-column matrix with each row providing one interval. The meaning of the confidence interval depends on the settings of the name-value pairs.

## Examples

### **Predict Response Values Using Linear Model**

Create a model of car mileage as a function of weight, and predict the response.

Create a quadratic model of car mileage as a function of weight from the `carsmall` data.

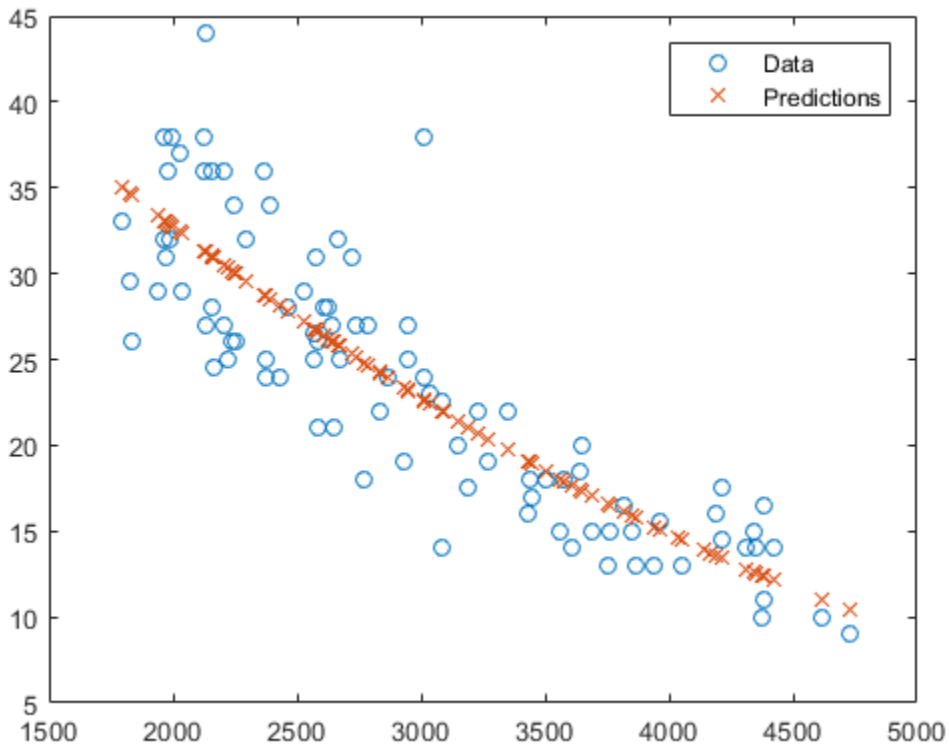
```
load carsmall
X = Weight;
y = MPG;
mdl = fitlm(X,y,'quadratic');
```

Create predicted responses to the data.

```
Xnew = X;  
ypred = predict(mdl, Xnew);
```

Plot the original responses and the predicted responses to see how they differ.

```
plot(X, y, 'o', Xnew, ypred, 'x')  
legend('Data', 'Predictions')
```



### Generate Code From Function That Predicts Responses Given New Data

This example is based on “Predict Response Values Using Linear Model” on page 32-5052.

Load the `carsmall` data set, and then fit the quadratic regression model.

```
load carsmall
X = Weight;
y = MPG;
mdl = fitlm(X,y,'quadratic');
```

Create points for prediction from the sorted in-sample data.

```
Xnew = sort(X);
```

Save the fitted quadratic model to the file `QLMMdl.mat`.

```
saveCompactModel(mdl,'QLMMdl');
```

Declare a function in your current working folder called `mypredictQLM.m` that:

- Accepts measurements commensurate with `Xnew` and perhaps valid name-value pair arguments
- Loads the fitted quadratic model in `QLMMdl.mat`
- Returns predictions and confidence interval bounds

```
function [yhat,ci] = mypredictQLM(x,varargin) %#codegen
%MYPREDICTQLM Predict response using linear model
% MYPREDICTQLM predicts responses for the n observations in the n-by-1
% vector x using using the linear model stored in the MAT-file
% QLMMdl.mat, and then returns the predictions in the n-by-1 vector yhat.
% MYPREDICTQLM also returns confidence interval bounds for the
% predictions in the n-by-2 vector ci.
CompactMdl = loadCompactModel('QLMMdl');
narginchk(1,Inf);
[yhat,ci] = predict(CompactMdl,x,varargin{:});
end
```

Generate a MEX function from `mypredictQLM.m`. Specify returning 90% simultaneous confidence intervals on the predictions. Because C uses static typing, `codegen` must determine the properties of all variables in MATLAB® files at compile time. To designate arguments as compile-time constants, use `coder.Constant`.

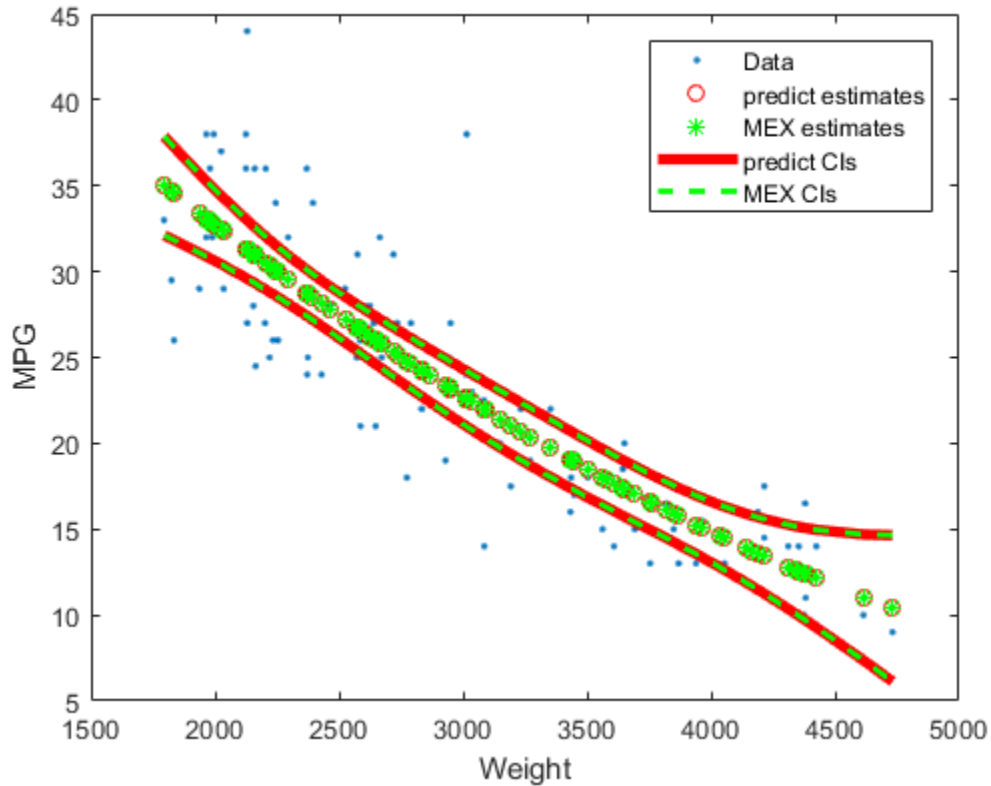
```
codegen -config:mex mypredictQLM -args {Xnew,coder.Constant('Alpha'),0.1,coder.Constant
```

The MEX file `mypredictQLM_mex.mexw64` generates to your present working directory. The file extension depends on your platform.

Compare predictions and confidence intervals using `predict` and `mypredictQLM_mex` by plotting both with the data. Specify name-value pair arguments in the same order as in the `-args` argument in the call to `codegen`.

```
[yhat1,ci1] = predict mdl,Xnew,'Alpha',0.1,'Simultaneous',true);
[yhat2,ci2] = mypredictQLM_mex(Xnew,'Alpha',0.1,'Simultaneous',true);

figure;
h1 = plot(X,y, '.');
hold on;
h2 = plot(Xnew,yhat1, 'ro',Xnew,yhat2, 'g*');
h3 = plot(Xnew,ci1, 'r-', 'LineWidth', 4);
h4 = plot(Xnew,ci2, 'g--', 'LineWidth', 2);
legend([h1; h2; h3(1); h4(1)], {'Data', 'predict estimates', 'MEX estimates', ...
    'predict CIs', 'MEX CIs'});
xlabel('Weight');
ylabel('MPG');
```



The generated MEX file and `predict` return essentially the same results.

- “Predict or Simulate Responses to New Data” on page 11-38

## Tips

- For predictions with added noise, use `random`.

## Alternatives

`feval` gives the same predictions, but uses multiple input arrays with one component in each input argument. `feval` can be simpler to use with a model created from a table or dataset array, although `feval` does not give confidence intervals on its predictions.

`random` predicts with added noise.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

Argument	Notes and Limitations
<code>mdl</code>	<ul style="list-style-type: none"> <li>• Save a trained model by using <code>saveCompactModel</code>, and then load the model by using <code>loadCompactModel</code> within a function that you declare.</li> <li>• If you train a linear model by using <code>fitlm</code> and specifying <code>'RobustOpts'</code> as a structure with an anonymous function handle for the <code>RobustWgtFun</code> field, use <code>saveCompactModel</code> to save the model, and use <code>loadCompactModel</code> to load the model, then <code>loadCompactModel</code> cannot restore the <code>Robust</code> property into the MATLAB Workspace. However, <code>loadCompactModel</code> can load the model at compile time for code generation within a function that you declare.</li> <li>• Must be a compile-time constant; its value cannot change in the generated code.</li> <li>• For more details, see “Code Generation” on page 32-0 of the <code>CompactLinearModel</code> class.</li> </ul>

Argument	Notes and Limitations
Xnew	<ul style="list-style-type: none"><li>• Must be a single- or double-precision matrix and can be variable-sized. However, the number of columns in Xnew must be <code>numel mdl.PredictorNames</code>.</li><li>• Rows and columns must correspond to observations and predictors, respectively.</li></ul>
Name-value pair arguments	Names in name-value pair arguments must be compile-time constants. For example, to allow a user-defined significance level in the generated code, include <code>{coder.Constant('Alpha'),0}</code> in the <code>-args</code> value of <code>codegen</code> .

## See Also

`CompactLinearModel` | `LinearModel` | `feval` | `fitlm` | `random` | `stepwiselm`

## Topics

“Predict or Simulate Responses to New Data” on page 11-38

“Linear Regression” on page 11-11

“Code Generation Workflows” on page 31-2

**Introduced in R2012a**



# predict

**Class:** LinearMixedModel

Predict response of linear mixed-effects model

## Syntax

```
ypred = predict(lme)
ypred = predict(lme, tblnew)
ypred = predict(lme, Xnew, Znew)
ypred = predict(lme, Xnew, Znew, Gnew)
ypred = predict(____, Name, Value)
[ypred, ypredCI] = predict(____)
[ypred, ypredCI, DF] = predict(____)
```

## Description

`ypred = predict(lme)` returns a vector of conditional predicted responses on page 32-5074 `ypred` at the original predictors used to fit the linear mixed-effects model `lme`.

`ypred = predict(lme, tblnew)` returns a vector of conditional predicted responses `ypred` from the fitted linear mixed-effects model `lme` at the values in the new table or dataset array `tblnew`. Use a table or dataset array for `predict` if you use a table or dataset array for fitting the model `lme`.

If a particular grouping variable in `tblnew` has levels that are not in the original data, then the random effects for that grouping variable do not contribute to the 'Conditional' prediction at observations where the grouping variable has new levels.

`ypred = predict(lme, Xnew, Znew)` returns a vector of conditional predicted responses `ypred` from the fitted linear mixed-effects model `lme` at the values in the new fixed- and random-effects design matrices, `Xnew` and `Znew`, respectively. `Znew` can also be a cell array of matrices. In this case, the grouping variable `G` is `ones(n, 1)`, where `n` is the number of observations used in the fit.

Use the matrix format for `predict` if using design matrices for fitting the model `lme`.

`ypred = predict(lme, Xnew, Znew, Gnew)` returns a vector of conditional predicted responses `ypred` from the fitted linear mixed-effects model `lme` at the values in the new fixed- and random-effects design matrices, `Xnew` and `Znew`, respectively, and the grouping variable `Gnew`.

`Znew` and `Gnew` can also be cell arrays of matrices and grouping variables, respectively.

`ypred = predict(____, Name, Value)` returns a vector of predicted responses `ypred` from the fitted linear mixed-effects model `lme` with additional options specified by one or more `Name, Value` pair arguments.

For example, you can specify the confidence level, simultaneous confidence bounds, or contributions from only fixed effects.

`[ypred, ypredCI] = predict(____)` also returns confidence intervals `ypredCI` for the predictions `ypred` for any of the input arguments in the previous syntaxes.

`[ypred, ypredCI, DF] = predict(____)` also returns the degrees of freedom `DF` used in computing the confidence intervals for any of the input arguments in the previous syntaxes.

## Input Arguments

### **lme** — Linear mixed-effects model

`LinearMixedModel` object

Linear mixed-effects model, specified as a `LinearMixedModel` object constructed using `fitlme` or `fitlmematrix`.

### **tblnew** — New input data

table | dataset array

New input data, which includes the response variable, predictor variables, and grouping variables on page 2-59, specified as a table or dataset array. The predictor variables can be continuous or grouping variables. `tblnew` must have the same variables as in the original table or dataset array used to fit the linear mixed-effects model `lme`.

Data Types: single | double | logical | char

**Xnew — New fixed-effects design matrix***n*-by-*p* matrix

New fixed-effects design matrix, specified as an *n*-by-*p* matrix, where *n* is the number of observations and *p* is the number of fixed predictor variables. Each row of **X** corresponds to one observation and each column of **X** corresponds to one variable.

Data Types: `single` | `double`**Znew — New random-effects design***n*-by-*q* matrix | cell array of length *R*

New random-effects design, specified as an *n*-by-*q* matrix or a cell array of *R* design matrices  $Z\{r\}$ , where  $r = 1, 2, \dots, R$ . If **Znew** is a cell array, then each  $Z\{r\}$  is an *n*-by- $q(r)$  matrix, where *n* is the number of observations, and  $q(r)$  is the number of random predictor variables.

Data Types: `single` | `double` | `logical` | `char` | `cell`**Gnew — New grouping variable or variables**vector | cell array of grouping variables of length *R*

New grouping variable or variables on page 2-59, specified as a vector or a cell array, of length *R*, of grouping variables with the same levels or groups as the original grouping variables used to fit the linear mixed-effects model `lme`.

Data Types: `single` | `double` | `logical` | `char` | `cell`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

**Alpha — Confidence level**

0.05 (default) | scalar value in the range 0 to 1

Confidence level, specified as the comma-separated pair consisting of 'Alpha' and a scalar value in the range 0 to 1. For a value  $\alpha$ , the confidence level is  $100 \cdot (1 - \alpha)\%$ .

For example, for 99% confidence intervals, you can specify the confidence level as follows.

Example: `'Alpha',0.01`

Data Types: `single | double`

**Conditional** — Indicator for conditional predictions

`true` (default) | `false`

Indicator for conditional prediction on page 32-5074s, specified as the comma-separated pair consisting of `'Conditional'` and one of the following.

<code>true</code>	Contributions from both fixed effects and random effects (conditional)
<code>false</code>	Contribution from only fixed effects (marginal)

Example: `'Conditional,false'`

**DFMethod** — Method for computing approximate degrees of freedom

`'residual'` (default) | `'satterthwaite'` | `'none'`

Method for computing approximate degrees of freedom to use in the confidence interval computation, specified as the comma-separated pair consisting of `'DFMethod'` and one of the following.

<code>'residual'</code>	Default. The degrees of freedom are assumed to be constant and equal to $n - p$ , where $n$ is the number of observations and $p$ is the number of fixed effects.
<code>'satterthwaite'</code>	Satterthwaite approximation.
<code>'none'</code>	All degrees of freedom are set to infinity.

For example, you can specify the Satterthwaite approximation as follows.

Example: `'DFMethod','satterthwaite'`

**Simultaneous** — Type of confidence bounds

`false` (default) | `true`

Type of confidence bounds, specified as the comma-separated pair consisting of `'Simultaneous'` and one of the following.

<code>false</code>	Default. Nonsimultaneous bounds.
--------------------	----------------------------------

true

Simultaneous bounds.

Example: `'Simultaneous', true`

### **Prediction** — Type of prediction

`'curve'` (default) | `'observation'`

Type of prediction, specified as the comma-separated pair consisting of `'Prediction'` and one of the following.

`'curve'`

Default. Confidence bounds for the predictions based on the fitted function.

`'observation'`

Variability due to observation error for the new observations is also included in the confidence bound calculations and this results in wider bounds.

Example: `'Prediction', 'observation'`

## Output Arguments

### **ypred** — Predicted responses

vector

Predicted responses, returned as a vector. `ypred` can contain the conditional or marginal responses, depending on the value choice of the `'Conditional'` name-value pair argument. Conditional predictions include contributions from both fixed and random effects.

### **ypredCI** — Point-wise confidence intervals

two-column matrix

Point-wise confidence intervals for the predicted values, returned as a two-column matrix. The first column of `yCI` contains the lower bounds, and the second column contains the upper bound. By default, `yCI` contains the 95% confidence intervals for the predictions. You can change the confidence level using the `Alpha` name-value pair argument, make them simultaneous using the `Simultaneous` name-value pair argument, and also make them for a new observation rather than for the curve using the `Prediction` name-value pair argument.

**DF — Degrees of freedom**

vector | scalar value

Degrees of freedom used in computing the confidence intervals, returned as a vector or a scalar value.

- If the 'Simultaneous' name-value pair argument is false, then DF is a vector.
- If the 'Simultaneous' name-value pair argument is true, then DF is a scalar value.

## Examples

### Predict Responses at the Original Design Values

Load the sample data.

```
load(fullfile(matlabroot, 'examples', 'stats', 'fertilizer.mat'));
```

The dataset array includes data from a split-plot experiment, where soil is divided into three blocks based on the soil type: sandy, silty, and loamy. Each block is divided into five plots, where five different types of tomato plants (cherry, heirloom, grape, vine, and plum) are randomly assigned to these plots. The tomato plants in the plots are then divided into subplots, where each subplot is treated by one of four fertilizers. This is simulated data.

Store the data in a dataset array called `ds`, for practical purposes, and define `Tomato`, `Soil`, and `Fertilizer` as categorical variables.

```
ds = fertilizer;  
ds.Tomato = nominal(ds.Tomato);  
ds.Soil = nominal(ds.Soil);  
ds.Fertilizer = nominal(ds.Fertilizer);
```

Fit a linear mixed-effects model, where `Fertilizer` and `Tomato` are the fixed-effects variables, and the mean yield varies by the block (soil type), and the plots within blocks (tomato types within soil types) independently.

```
lme = fitlme(ds, 'Yield ~ Fertilizer * Tomato + (1|Soil) + (1|Soil:Tomato)');
```

Predict the response values at the original design values. Display the first five predictions with the observed response values.

```
yhat = predict(lme);
[yhat(1:5) ds.Yield(1:5)]
```

```
ans =
```

```
115.4788 104.0000
135.1455 136.0000
152.8121 158.0000
160.4788 174.0000
58.0839 57.0000
```

## Plot Predictions vs. Observed Responses

Load the sample data.

```
load carsmall
```

Fit a linear mixed-effects model, with a fixed effect for `Weight`, and a random intercept grouped by `Model_Year`. First, store the data in a table.

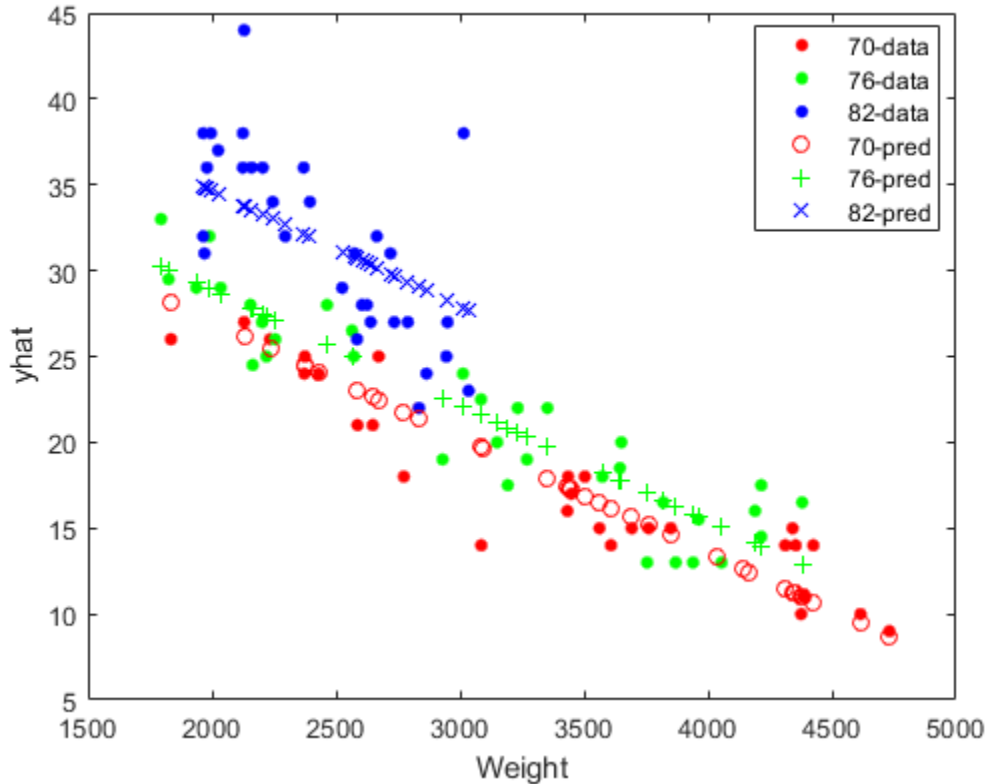
```
tbl = table(MPG,Weight,Model_Year);
lme = fitlme(tbl,'MPG ~ Weight + (1|Model_Year)');
```

Create predicted responses to the data.

```
yhat = predict(lme,tbl);
```

Plot the original responses and the predicted responses to see how they differ. Group them by model year.

```
figure()
gscatter(Weight,MPG,Model_Year)
hold on
gscatter(Weight,yhat,Model_Year,[],'o+x')
legend('70-data','76-data','82-data','70-pred','76-pred','82-pred')
hold off
```



### Predict Responses at Values in a New Dataset Array

Load the sample data.

```
load(fullfile(matlabroot, 'examples', 'stats', 'fertilizer.mat'));
```

The dataset array includes data from a split-plot experiment, where soil is divided into three blocks based on the soil type: sandy, silty, and loamy. Each block is divided into five plots, where five different types of tomato plants (cherry, heirloom, grape, vine, and plum) are randomly assigned to these plots. The tomato plants in the plots are then divided into subplots, where each subplot is treated by one of four fertilizers. This is simulated data.



Store the data in a dataset array called `ds`, for practical purposes, and define `Tomato`, `Soil`, and `Fertilizer` as categorical variables.

```
ds = fertilizer;
ds.Tomato = nominal(ds.Tomato);
ds.Soil = nominal(ds.Soil);
ds.Fertilizer = nominal(ds.Fertilizer);
```

Fit a linear mixed-effects model, where `Fertilizer` and `Tomato` are the fixed-effects variables, and the mean yield varies by the block (soil type), and the plots within blocks (tomato types within soil types) independently.

```
lme = fitlme(ds, 'Yield ~ Fertilizer * Tomato + (1|Soil) + (1|Soil:Tomato)');
```

Create a new dataset array with design values. The new dataset array must have the same variables as the original dataset array you use for fitting the model `lme`.

```
dsnew = dataset();
dsnew.Soil = nominal({'Sandy'; 'Silty'});
dsnew.Tomato = nominal({'Cherry'; 'Vine'});
dsnew.Fertilizer = nominal([2;2]);
```

Predict the conditional and marginal responses at the original design points.

```
yhatC = predict(lme, dsnew);
yhatM = predict(lme, dsnew, 'Conditional', false);
[yhatC yhatM]
```

```
ans =
```

```
    92.7505    111.6667
    87.5891     82.6667
```

## Predict Responses at the Values in New Design Matrices

Load the sample data.

```
load carbig
```

Fit a linear mixed-effects model for miles per gallon (MPG), with fixed effects for acceleration, horsepower, and cylinders, and potentially correlated random effects for intercept and acceleration grouped by model year.

First, prepare the design matrices for fitting the linear mixed-effects model.

```
X = [ones(406,1) Acceleration Horsepower];
Z = [ones(406,1) Acceleration];
Model_Year = nominal(Model_Year);
G = Model_Year;
```

Now, fit the model using `fitlmematrix` with the defined design matrices and grouping variables.

```
lme = fitlmematrix(X,MPG,Z,G,'FixedEffectPredictors',...
{'Intercept','Acceleration','Horsepower'},'RandomEffectPredictors',...
{'Intercept','Acceleration'}},'RandomEffectGroups',{'Model_Year'});
```

Create the design matrices that contain the data at which to predict the response values. `Xnew` must have three columns as in `X`. The first column must be a column of 1s. And the values in the last two columns must correspond to `Acceleration` and `Horsepower`, respectively. The first column of `Znew` must be a column of 1s, and the second column must contain the same `Acceleration` values as in `Xnew`. The original grouping variable in `G` is the model year. So, `Gnew` must contain values for the model year. Note that `Gnew` must contain nominal values.

```
Xnew = [1,13.5,185; 1,17,205; 1,21.2,193];
Znew = [1,13.5; 1,17; 1,21.2]; % alternatively Znew = Xnew(:,1:2);
Gnew = nominal([73 77 82]);
```

Predict the responses for the data in the new design matrices.

```
yhat = predict(lme,Xnew,Znew,Gnew)
```

```
yhat =
    8.7063
    5.4423
   12.5384
```

Now, repeat the same for a linear mixed-effects model with uncorrelated random-effects terms for intercept and acceleration. First, change the original random effects design and the random effects grouping variables. Then, refit the model.

```
Z = {ones(406,1),Acceleration};
G = {Model_Year,Model_Year};

lme = fitlmematrix(X,MPG,Z,G,'FixedEffectPredictors',....
{'Intercept','Acceleration','Horsepower'},'RandomEffectPredictors',...
{{'Intercept'}},{'Acceleration'}},'RandomEffectGroups',{'Model_Year','Model_Year'});
```

Now, recreate the new random effects design, `Znew`, and the grouping variable design, `Gnew`, using which to predict the response values.

```
Znew = {[1;1;1],[13.5;17;21.2]};
MY = nominal([73 77 82]);
Gnew = {MY,MY};
```

Predict the responses using the new design matrices.

```
yhat = predict(lme,Xnew,Znew,Gnew)
```

```
yhat =
    8.6365
    5.9199
   12.1247
```

## Compute Confidence Intervals for Predictions

Load the sample data.

```
load carbig
```

Fit a linear mixed-effects model for miles per gallon (MPG), with fixed effects for acceleration, horsepower, and cylinders, and potentially correlated random effects for intercept and acceleration grouped by model year. First, store the variables in a table.

```
tbl = table(MPG,Acceleration,Horsepower,Model_Year);
```

Now, fit the model using `fitlme` with the defined design matrices and grouping variables.

```
lme = fitlme(tbl, 'MPG ~ Acceleration + Horsepower + (Acceleration|Model_Year)');
```

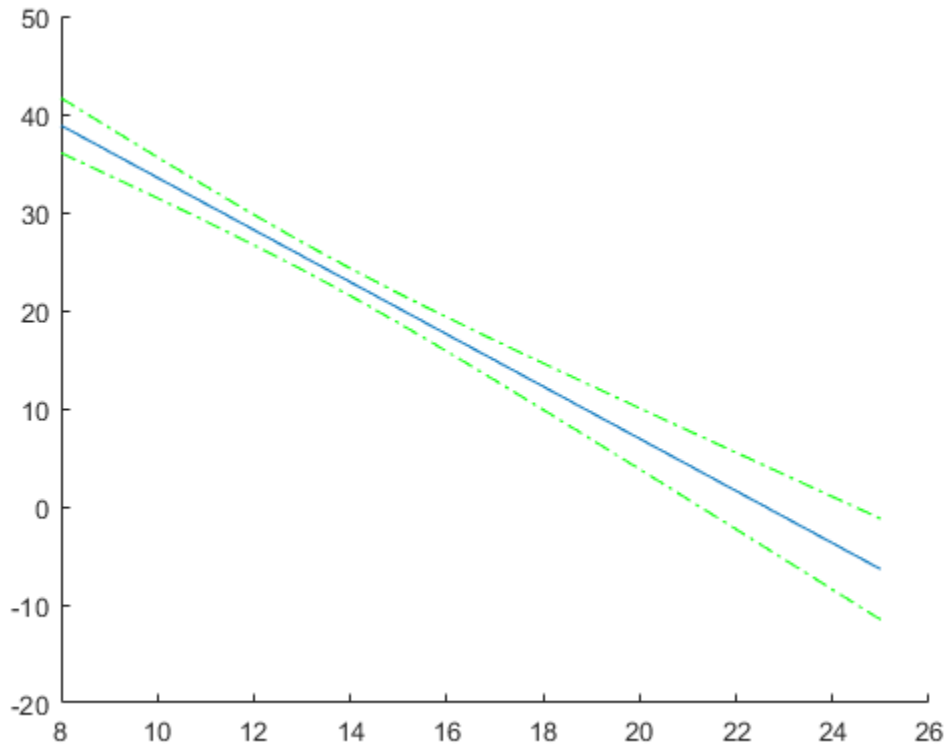
Create the new data and store it in a new table.

```
tblnew = table();  
tblnew.Acceleration = linspace(8,25)';  
tblnew.Horsepower = linspace(nanmin(Horsepower), nanmax(Horsepower))';  
tblnew.Model_Year = repmat(70,100,1);
```

`linspace` creates 100 equally distanced values between the lower and the upper input limits. `Model_Year` is fixed at 70. You can repeat this for any model year.

Compute and plot the predicted values and 95% confidence limits (nonsimultaneous).

```
[ypred, yCI, DF] = predict(lme, tblnew);  
figure();  
h1 = line(tblnew.Acceleration, ypred);  
hold on;  
h2 = plot(tblnew.Acceleration, yCI, 'g-.');
```



Display the degrees of freedom.

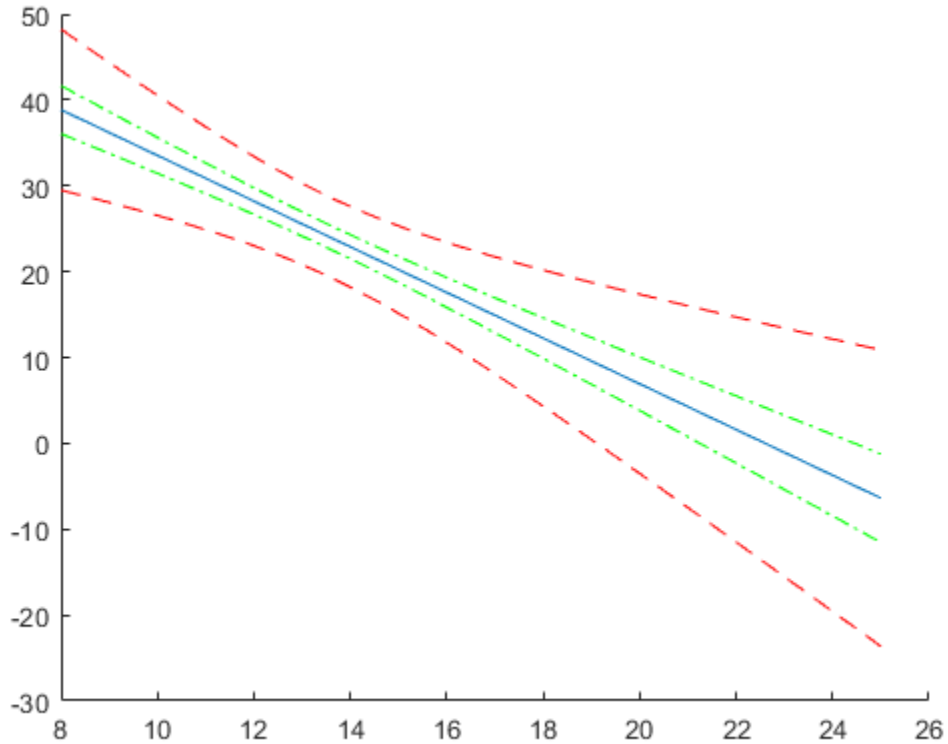
```
DF(1)
```

```
ans =
```

```
389
```

Compute and plot the simultaneous confidence bounds.

```
[ypred,yCI,DF] = predict(lme,tblnew,'Simultaneous',true);  
h3 = plot(tblnew.Acceleration,yCI,'r--');
```



Display the degrees of freedom.

```
DF
```

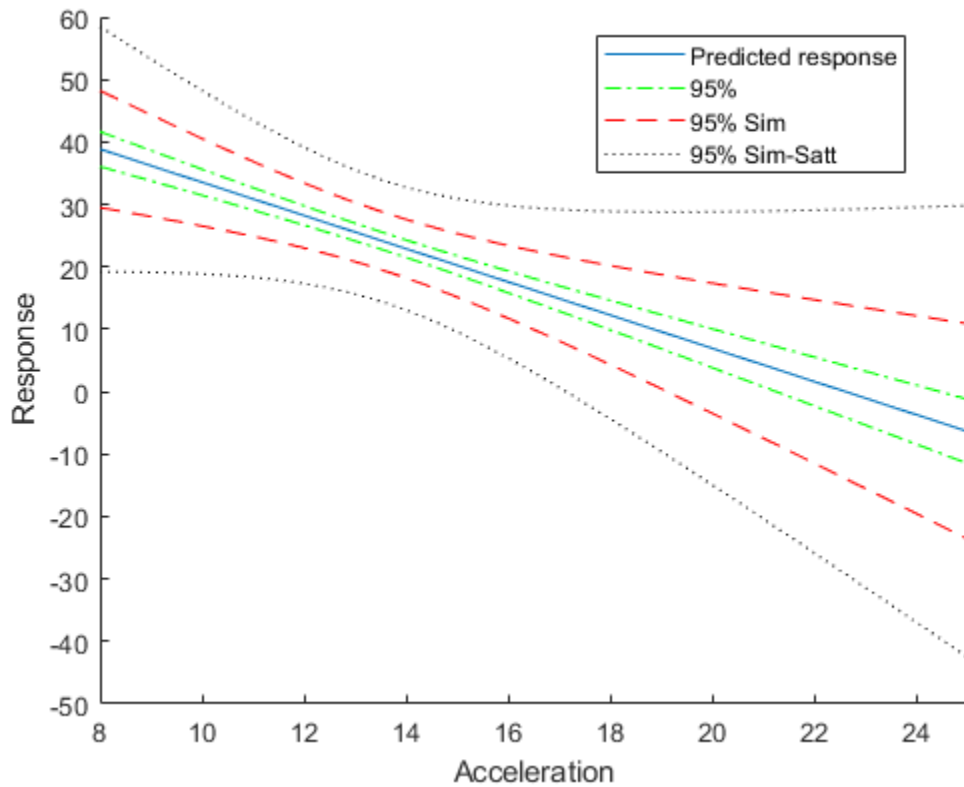
```
DF =
```

```
389
```

Compute the simultaneous confidence bounds using the Satterthwaite method to compute the degrees of freedom.

```
[ypred,yCI,DF] = predict(lme,tblnew,'Simultaneous',true,'DFMethod','satterthwaite');
h4 = plot(tblnew.Acceleration,yCI,'k:');
```

```
hold off
xlabel('Acceleration')
ylabel('Response')
ylim([-50, 60])
xlim([8, 25])
legend([h1,h2(1),h3(1),h4(1)], 'Predicted response', '95%', '95% Sim', ...
'95% Sim-Satt', 'Location', 'Best')
```



Display the degrees of freedom.

DF

DF =

3.6002

## Definitions

### Conditional and Marginal Predictions

A conditional prediction includes contributions from both fixed and random effects, whereas a marginal model includes contribution from only fixed effects.

Suppose the linear mixed-effects model `lme` has an  $n$ -by- $p$  fixed-effects design matrix  $X$  and an  $n$ -by- $q$  random-effects design matrix  $Z$ . Also, suppose the estimated  $p$ -by-1 fixed-effects vector is  $\hat{\beta}$ , and the  $q$ -by-1 estimated best linear unbiased predictor (BLUP) vector of random effects is  $\hat{b}$ . The predicted conditional response is

$$\hat{y}_{Cond} = X\hat{\beta} + Z\hat{b},$$

which corresponds to the `'Conditional'`, `'true'` name-value pair argument.

The predicted marginal response is

$$\hat{y}_{Mar} = X\hat{\beta},$$

which corresponds to the `'Conditional'`, `'false'` name-value pair argument.

When making predictions, if a particular grouping variable has new levels (1s that were not in the original data), then the random effects for the grouping variable do not contribute to the `'Conditional'` prediction at observations where the grouping variable has new levels.

## See Also

`LinearMixedModel` | `fitted` | `random`



# predict

**Class:** NaiveBayes

Predict class label for test data

## Syntax

```
cpre = predict(nb, test)
cpre = predict(..., 'HandleMissing', val)
```

## Description

`cpre = predict(nb, test)` classifies each row of data in `test` into one of the classes according to the NaiveBayes classifier `nb`, and returns the predicted class level `cpre`. `test` is an `N`-by-`nb.ndims` matrix, where `N` is the number of observations in the test data. Rows of `test` correspond to points, columns of `test` correspond to features. `cpre` is an `N`-by-1 vector of the same type as `nb.CLevels`, and it indicates the class to which each row of `test` has been assigned.

`cpre = predict(..., 'HandleMissing', val)` specifies how `predict` treats NaN (missing values). `val` can be one of the following:

- 'off' (default)      Observations with NaN in any of the columns are not classified into any class. The corresponding rows in `cpre` are NaN (if `obj.clevels` is numeric or logical), empty character vectors (if `obj.clevels` is a character array or cell array of character vectors) or `<undefined>` (if `obj.clevels` is categorical).
- 'on'                      For observations having NaN in some (but not all) columns, `predict` computes `cpre` using the columns with non-NaN values. Corresponding `cpre` values are NaN.

## See Also

NaiveBayes | fitNaiveBayes | posterior

## predict

**Class:** FeatureSelectionNCAClassification

Predict responses using neighborhood component analysis (NCA) classifier

## Syntax

```
[labels,postprobs,classnames] = predict mdl,X
```

## Description

`[labels,postprobs,classnames] = predict(mdl,X)` computes the predicted labels, `labels`, corresponding to the rows of `X`, using the model `mdl`.

## Input Arguments

**mdl** — Neighborhood component analysis model for classification

FeatureSelectionNCAClassification object

Neighborhood component analysis model for classification, specified as a FeatureSelectionNCAClassification object.

**x** — Predictor variable values

*n*-by-*p* matrix

Predictor variable values, specified as an *n*-by-*p* matrix, where *n* is the number of observations and *p* is the number of predictor variables.

Data Types: single | double

## Output Arguments

### **labels** — Predicted class labels

categorical vector | logical vector | numeric vector | cell array of character vectors | character array

Predicted class labels corresponding to the rows of  $X$ , returned as a categorical, logical, or numeric vector, a cell array of character vectors of length  $n$ , or a character array with  $n$  rows.  $n$  is the number of observations. The type of `labels` is the same as  $Y$  used in training.

### **postprobs** — Posterior probabilities

$n$ -by- $c$  matrix

Posterior probabilities, returned as an  $n$ -by- $c$  matrix, where  $n$  is the number of observations and  $c$  is the number of classes. A posterior probability, `postprobs(i, :)`, represents the membership of an observation in  $X(i, :)$  in classes 1 through  $c$ .

### **classnames** — Class names

cell array of character vectors

Class names corresponding to posterior probabilities, returned as a cell array of character vectors. Each character vector is the class name corresponding to a column of `postprobs`.

## Examples

### Tune NCA Model for Classification

Load the sample data.

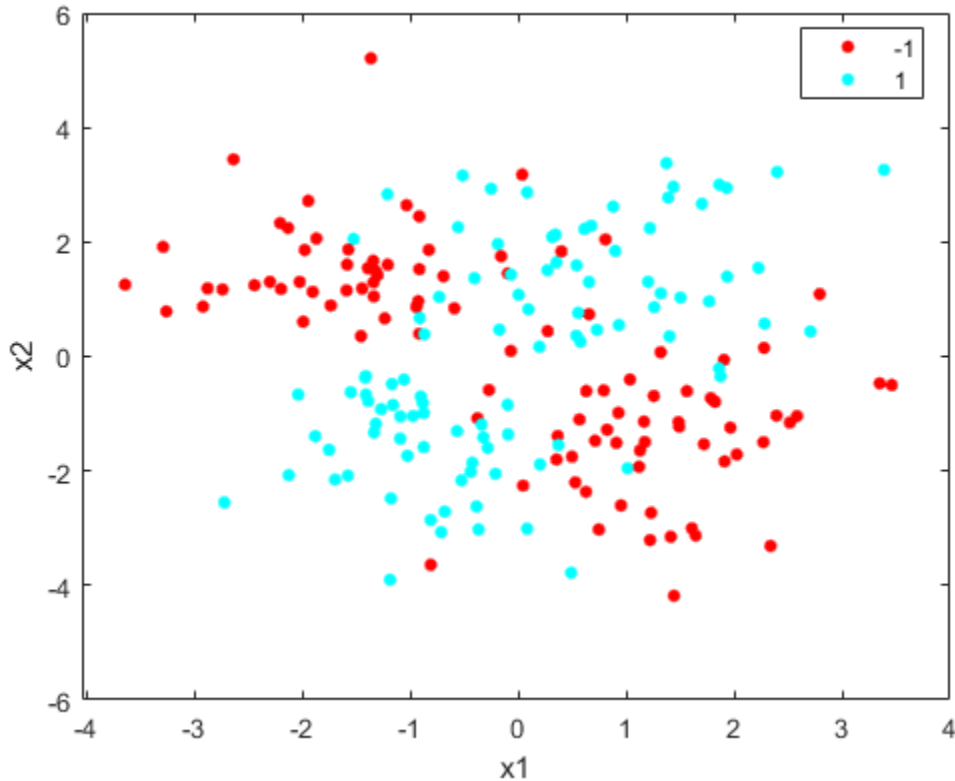
```
load(fullfile(matlabroot, 'examples', 'stats', 'twodimclassdata.mat'));
```

This data set is simulated using the scheme described in [1]. This is a two-class classification problem in two dimensions. Data from the first class are drawn from two bivariate normal distributions  $N(\mu_1, \Sigma)$  or  $N(\mu_2, \Sigma)$  with equal probability, where  $\mu_1 = [-0.75, -1.5]$ ,  $\mu_2 = [0.75, 1.5]$ , and  $\Sigma = I_2$ . Similarly, data from the second class are drawn from two bivariate normal distributions  $N(\mu_3, \Sigma)$  or  $N(\mu_4, \Sigma)$  with equal

probability, where  $\mu_3 = [1.5, -1.5]$ ,  $\mu_4 = [-1.5, 1.5]$ , and  $\Sigma = I_2$ . The normal distribution parameters used to create this data set result in tighter clusters in data than the data used in [1].

Create a scatter plot of the data grouped by the class.

```
figure
gscatter(X(:,1),X(:,2),y)
xlabel('x1')
ylabel('x2')
```



Add 100 irrelevant features to  $X$ . First generate data from a Normal distribution with a mean of 0 and a variance of 20.

```
n = size(X,1);
rng('default')
XwithBadFeatures = [X,randn(n,100)*sqrt(20)];
```

Normalize the data so that all points are between 0 and 1.

```
XwithBadFeatures = bsxfun(@rdivide,...
    bsxfun(@minus,XwithBadFeatures,min(XwithBadFeatures,[],1)), ...
    range(XwithBadFeatures,1));
X = XwithBadFeatures;
```

Fit a neighborhood component analysis (NCA) model to the data using the default Lambda (regularization parameter,  $\lambda$ ) value. Use the LBFGS solver and display the convergence information.

```
ncaMdl = fscnca(X,y,'FitMethod','exact','Verbose',1, ...
    'Solver','lbfgs');
```

```
o Solver = LBFGS, HessianHistorySize = 15, LineSearchMethod = weakwolfe
```

```
=====
| ITER | FUN VALUE | NORM GRAD | NORM STEP | CURV | GAMMA | ALPH |
|-----|-----|-----|-----|-----|-----|-----|
| 0 | 9.519258e-03 | 1.494e-02 | 0.000e+00 | | 4.015e+01 | 0.000 |
| 1 | -3.093574e-01 | 7.186e-03 | 4.018e+00 | OK | 8.956e+01 | 1.000 |
| 2 | -4.809455e-01 | 4.444e-03 | 7.123e+00 | OK | 9.943e+01 | 1.000 |
| 3 | -4.938877e-01 | 3.544e-03 | 1.464e+00 | OK | 9.366e+01 | 1.000 |
| 4 | -4.964759e-01 | 2.901e-03 | 6.084e-01 | OK | 1.554e+02 | 1.000 |
| 5 | -4.972077e-01 | 1.323e-03 | 6.129e-01 | OK | 1.195e+02 | 5.000 |
| 6 | -4.974743e-01 | 1.569e-04 | 2.155e-01 | OK | 1.003e+02 | 1.000 |
| 7 | -4.974868e-01 | 3.844e-05 | 4.161e-02 | OK | 9.835e+01 | 1.000 |
| 8 | -4.974874e-01 | 1.417e-05 | 1.073e-02 | OK | 1.043e+02 | 1.000 |
| 9 | -4.974874e-01 | 4.893e-06 | 1.781e-03 | OK | 1.530e+02 | 1.000 |
| 10 | -4.974874e-01 | 9.404e-08 | 8.947e-04 | OK | 1.670e+02 | 1.000
```

```
Infinity norm of the final gradient = 9.404e-08
```

```
Two norm of the final step = 8.947e-04, TolX = 1.000e-06
```

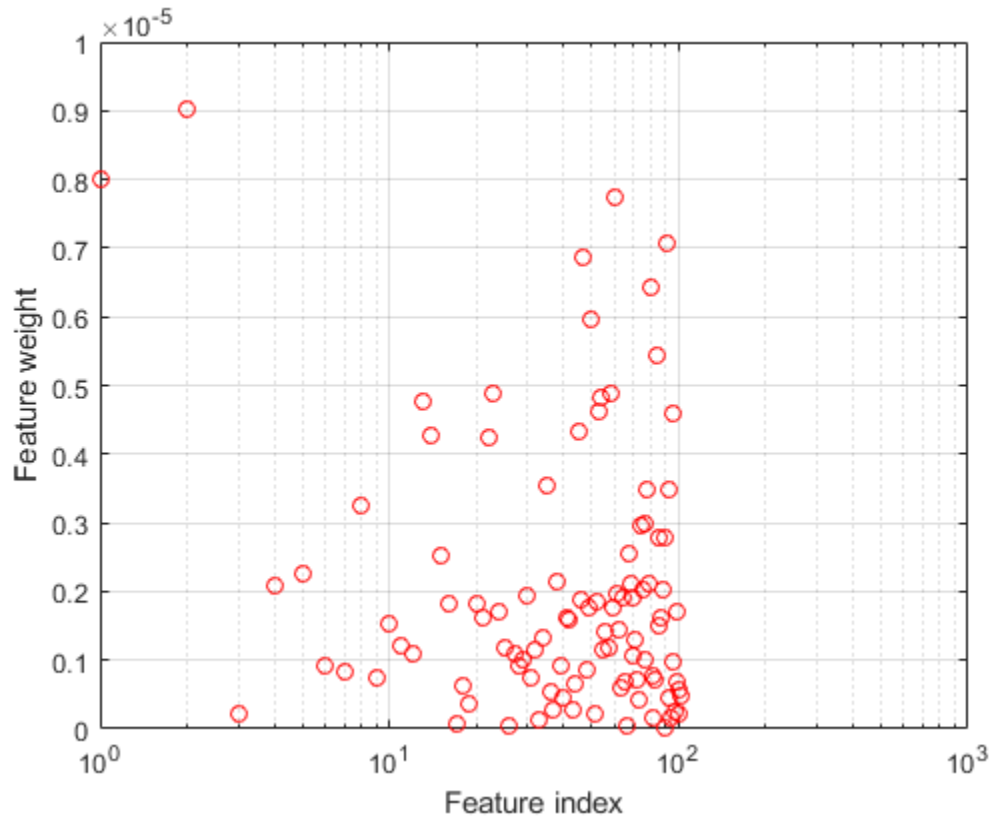
```
Relative infinity norm of the final gradient = 9.404e-08, TolFun = 1.000e-06
```

```
EXIT: Local minimum found.
```

Plot the feature weights. The weights of the irrelevant features should be very close to zero.

```
figure
semilogx(ncaMdl.FeatureWeights,'ro')
```

```
xlabel('Feature index')
ylabel('Feature weight')
grid on
```



Predict the classes using the NCA model and compute the confusion matrix.

```
ypred = predict(ncaMdl,X);
confusionmat(y,ypred)
```

```
ans =
```

```
40    60
 6    94
```

Confusion matrix shows that 40 of the data that are in class 1 are predicted as belonging to class 1. 60 of the data from class 1 are predicted to be in class 2. Similarly, 94 of the data from class 2 are predicted to be from class 2 and 6 of them are predicted to be from class 1. The prediction accuracy for class 1 is not good.

All weights are very close to zero, which indicates that the value of  $\lambda$  used in training the model is too large. When  $\lambda \rightarrow \infty$ , all features weights approach to zero. Hence, it is important to tune the regularization parameter in most cases to detect the relevant features.

Use five-fold cross-validation to tune  $\lambda$  for feature selection by using `fscnca`. Tuning  $\lambda$  means finding the  $\lambda$  value that will produce the minimum classification loss. To tune  $\lambda$  using cross-validation:

1. Partition the data into five folds. For each fold, `cvpartition` assigns four-fifths of the data as a training set and one-fifth of the data as a test set. Again for each fold, `cvpartition` creates a stratified partition, where each partition has roughly the same proportion of classes.

```
cvp = cvpartition(y, 'kfold', 5);
numtestsets = cvp.NumTestSets;
lambdavalues = linspace(0, 2, 20) / length(y);
lossvalues = zeros(length(lambdavalues), numtestsets);
```

2. Train the neighborhood component analysis (nca) model for each  $\lambda$  value using the training set in each fold.

3. Compute the classification loss for the corresponding test set in the fold using the nca model. Record the loss value.

4. Repeat this process for all folds and all  $\lambda$  values.

```
for i = 1:length(lambdavalues)
    for k = 1:numtestsets

        % Extract the training set from the partition object
        Xtrain = X(cvp.training(k), :);
        ytrain = y(cvp.training(k), :);

        % Extract the test set from the partition object
        Xtest = X(cvp.test(k), :);
        ytest = y(cvp.test(k), :);
```

```
% Train an NCA model for classification using the training set
ncaMdl = fscnca(Xtrain,ytrain,'FitMethod','exact', ...
    'Solver','lbfgs','Lambda',lambdavalues(i));

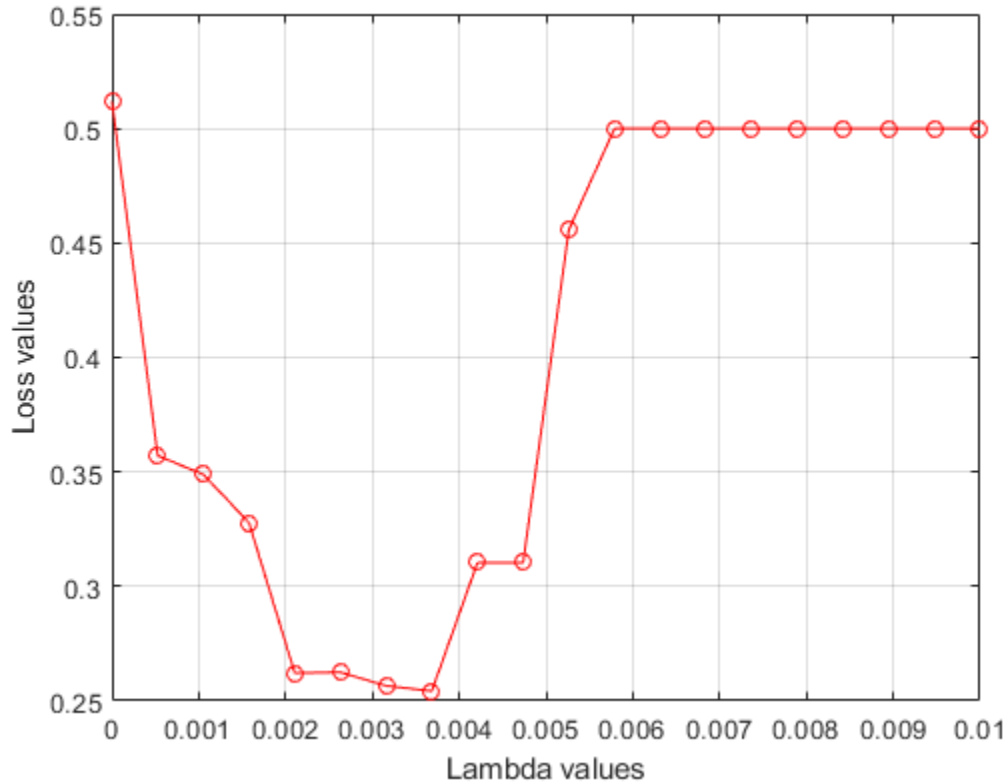
% Compute the classification loss for the test set using the NCA
% model
lossvalues(i,k) = loss(ncaMdl,Xtest,ytest, ...
    'LossFunction','quadratic');

    end
end
```

Plot the average loss values of the folds versus the  $\lambda$  values. If the  $\lambda$  value that corresponds to the minimum loss falls on the boundary of the tested  $\lambda$  values, the range of  $\lambda$  values should be reconsidered.

```
figure
plot(lambdavalues,mean(lossvalues,2),'ro-')
xlabel('Lambda values')
ylabel('Loss values')
grid on
```





Find the  $\lambda$  value that corresponds to the minimum average loss.

```
[~,idx] = min(mean(lossvalues,2)); % Find the index
bestlambda = lambdavalues(idx) % Find the best lambda value
```

```
bestlambda =
```

```
0.0037
```

Fit the NCA model to all of the data using the best  $\lambda$  value. Use the LBFGS solver and display the convergence information.

```
ncaMdl = fscnca(X,y,'FitMethod','exact','Verbose',1, ...
               'Solver','lbfgs','Lambda',bestlambda);
```

```
o Solver = LBFSGS, HessianHistorySize = 15, LineSearchMethod = weakwolfe
```

```
=====
| ITER | FUN VALUE | NORM GRAD | NORM STEP | CURV | GAMMA | ALPH
|=====|
| 0 | -1.246913e-01 | 1.231e-02 | 0.000e+00 | | 4.873e+01 | 0.000
| 1 | -3.411330e-01 | 5.717e-03 | 3.618e+00 | OK | 1.068e+02 | 1.000
| 2 | -5.226111e-01 | 3.763e-02 | 8.252e+00 | OK | 7.825e+01 | 1.000
| 3 | -5.817731e-01 | 8.496e-03 | 2.340e+00 | OK | 5.591e+01 | 5.000
| 4 | -6.132632e-01 | 6.863e-03 | 2.526e+00 | OK | 8.228e+01 | 1.000
| 5 | -6.135264e-01 | 9.373e-03 | 7.341e-01 | OK | 3.244e+01 | 1.000
| 6 | -6.147894e-01 | 1.182e-03 | 2.933e-01 | OK | 2.447e+01 | 1.000
| 7 | -6.148714e-01 | 6.392e-04 | 6.688e-02 | OK | 3.195e+01 | 1.000
| 8 | -6.149524e-01 | 6.521e-04 | 9.934e-02 | OK | 1.236e+02 | 1.000
| 9 | -6.149972e-01 | 1.154e-04 | 1.191e-01 | OK | 1.171e+02 | 1.000
| 10 | -6.149990e-01 | 2.922e-05 | 1.983e-02 | OK | 7.365e+01 | 1.000
| 11 | -6.149993e-01 | 1.556e-05 | 8.354e-03 | OK | 1.288e+02 | 1.000
| 12 | -6.149994e-01 | 1.147e-05 | 7.256e-03 | OK | 2.332e+02 | 1.000
| 13 | -6.149995e-01 | 1.040e-05 | 6.781e-03 | OK | 2.287e+02 | 1.000
| 14 | -6.149996e-01 | 9.015e-06 | 6.265e-03 | OK | 9.974e+01 | 1.000
| 15 | -6.149996e-01 | 7.763e-06 | 5.206e-03 | OK | 2.919e+02 | 1.000
| 16 | -6.149997e-01 | 8.374e-06 | 1.679e-02 | OK | 6.878e+02 | 1.000
| 17 | -6.149997e-01 | 9.387e-06 | 9.542e-03 | OK | 1.284e+02 | 5.000
| 18 | -6.149997e-01 | 3.250e-06 | 5.114e-03 | OK | 1.225e+02 | 1.000
| 19 | -6.149997e-01 | 1.574e-06 | 1.275e-03 | OK | 1.808e+02 | 1.000
|=====|
```

```
=====
| ITER | FUN VALUE | NORM GRAD | NORM STEP | CURV | GAMMA | ALPH
|=====|
| 20 | -6.149997e-01 | 5.764e-07 | 6.765e-04 | OK | 2.905e+02 | 1.000
|=====|
```

```
Infinity norm of the final gradient = 5.764e-07
```

```
Two norm of the final step = 6.765e-04, TolX = 1.000e-06
```

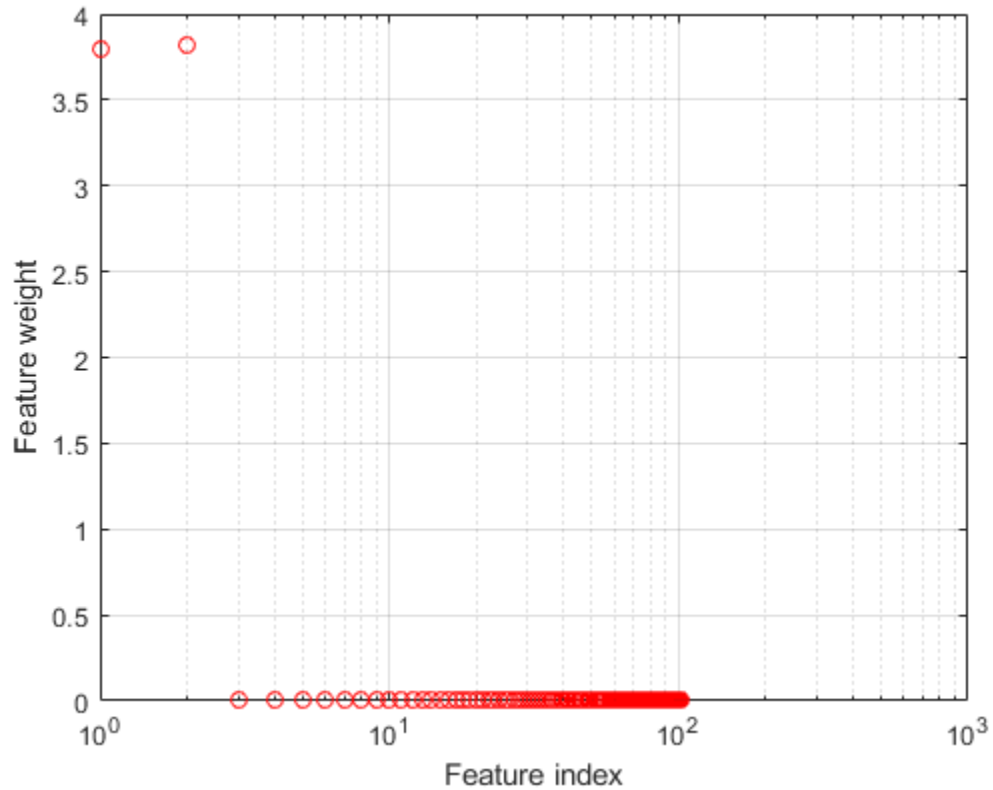
```
Relative infinity norm of the final gradient = 5.764e-07, TolFun = 1.000e-06
```

```
EXIT: Local minimum found.
```

Plot the feature weights.

```
figure
semilogx(ncaMdl.FeatureWeights,'ro')
xlabel('Feature index')
```

```
ylabel('Feature weight')
grid on
```



`fscnca` correctly figures out that the first two features are relevant and that the rest are not. The first two features are not individually informative, but when taken together result in an accurate classification model.

Predict the classes using the new model and compute the accuracy.

```
ypred = predict(ncaMdl,X);
confusionmat(y,ypred)
```

```
ans =
```

88	12
8	92

Confusion matrix shows that prediction accuracy for class 1 has improved. 88 of the data from class 1 are predicted to be from 1, and 12 of them are predicted to be from class 2. 92 of the data from class 2 are predicted to be from class 2 and 8 of them are predicted to be from class 1.

### References

1. Yang, W., K. Wang, W. Zuo. "Neighborhood Component Feature Selection for High-Dimensional Data." *Journal of Computers*. Vol. 7, Number 1, January, 2012.

### See Also

`FeatureSelectionNCAClassification` | `fscnca` | `loss` | `refit`

**Introduced in R2016b**

# predict

**Class:** FeatureSelectionNCARegression

Predict responses using neighborhood component analysis (NCA) regression model

## Syntax

```
ypred = predict mdl, X
```

## Description

`ypred = predict(mdl, X)` computes the predicted response values, `ypred`, corresponding to rows of `X`, using the model `mdl`.

## Input Arguments

**mdl** — Neighborhood component analysis model for regression

FeatureSelectionNCARegression object

Neighborhood component analysis model for regression, specified as a FeatureSelectionNCARegression object.

**x** — Predictor variable values

*n*-by-*p* matrix

Predictor variable values, specified as an *n*-by-*p* matrix, where *n* is the number of observations and *p* is the number of predictor variables.

Data Types: single | double

## Output Arguments

**ypred** — Predicted response values

*n*-by-1 vector

Predicted response values, specified as an  $n$ -by-1 vector, where  $n$  is the number of observations.

## Examples

### Tune NCA Model for Regression Using `loss` and `predict`

#### Load the sample data.

Download the housing data [1], from the UCI Machine Learning Repository [2]. The dataset has 506 observations. The first 13 columns contain the predictor values and the last column contains the response values. The goal is to predict the median value of owner-occupied homes in suburban Boston as a function of 13 predictors.

Load the data and define the response vector and the predictor matrix.

```
load('housing.data');  
X = housing(:,1:13);  
y = housing(:,end);
```

Divide the data into training and test sets using the 4th predictor as the grouping variable for a stratified partitioning. This ensures that each partition includes similar amount of observations from each group.

```
rng(1) % For reproducibility  
cvp = cvpartition(X(:,4), 'Holdout', 56);  
Xtrain = X(cvp.training, :);  
ytrain = y(cvp.training, :);  
Xtest = X(cvp.test, :);  
ytest = y(cvp.test, :);
```

`cvpartition` randomly assigns 56 observations into a test set and the rest of the data into a training set.

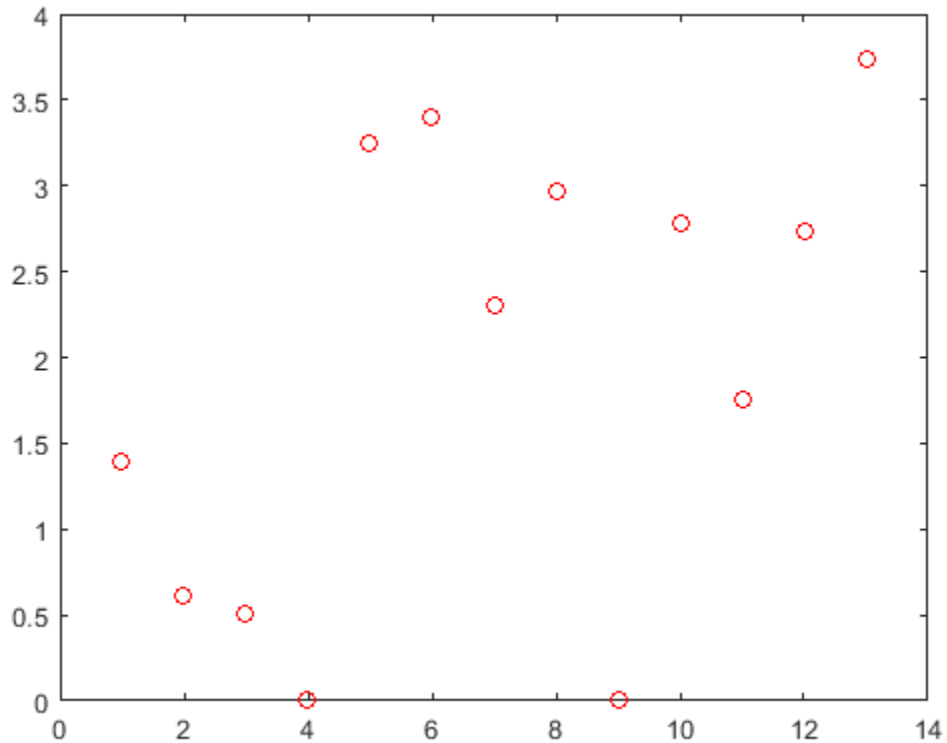
#### Perform Feature Selection Using Default Settings

Perform feature selection using NCA model for regression. Standardize the predictor values.

```
nca = fsrnca(Xtrain, ytrain, 'Standardize', 1);
```

Plot the feature weights.

```
figure()  
plot(nca.FeatureWeights, 'ro')
```



The weights of irrelevant features are expected to approach zero. `fsrcna` identifies two features as irrelevant.

Compute the regression loss.

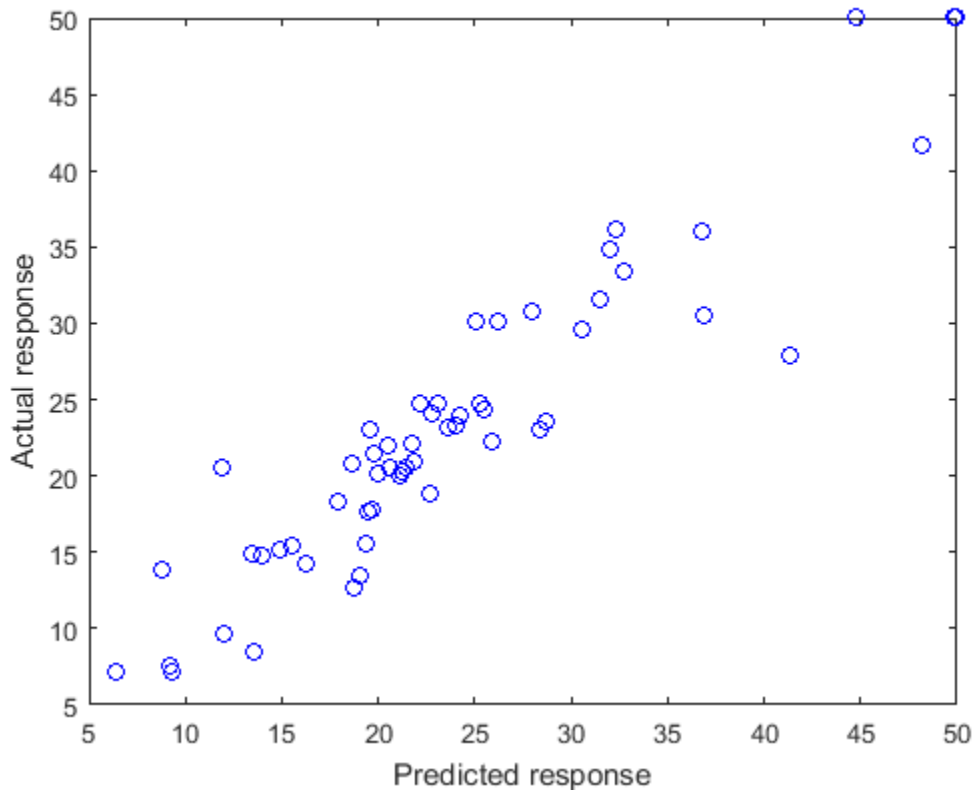
```
L = loss(nca, Xtest, ytest, 'LossFunction', 'mad')
```

```
L =
```

```
2.5394
```

Compute the predicted response values for the test set and plot them versus the actual response.

```
y_pred = predict(nca, X_test);  
figure()  
plot(y_pred, y_test, 'bo')  
xlabel('Predicted response')  
ylabel('Actual response')
```



A perfect fit versus the actual values forms a 45 degree straight line. In this plot, the predicted and actual response values seem to be scattered around this line. Tuning  $\lambda$  (regularization parameter) value usually helps improve the performance.



## Tune the regularization parameter using 10-fold cross-validation

Tuning  $\lambda$  means finding the  $\lambda$  value that will produce the minimum regression loss. Here are the steps for tuning  $\lambda$  using 10-fold cross-validation:

1. First partition the data into 10 folds. For each fold, `cvpartition` assigns 1/10th of the data as a training set, and 9/10th of the data as a test set.

```
n = length(ytrain);
cvp = cvpartition(Xtrain(:,4), 'kfold', 10);
numvalidsets = cvp.NumTestSets;
```

Assign the  $\lambda$  values for the search. Create an array to store the loss values.

```
lambdaval = linspace(0, 2, 30) * std(ytrain) / n;
lossvals = zeros(length(lambdaval), numvalidsets);
```

2. Train the neighborhood component analysis (nca) model for each  $\lambda$  value using the training set in each fold.

3. Fit a Gaussian process regression (gpr) model using the selected features. Next, compute the regression loss for the corresponding test set in the fold using the gpr model. Record the loss value.

4. Repeat this for each  $\lambda$  value and each fold.

```
for i = 1:length(lambdaval)
    for k = 1:numvalidsets
        X = Xtrain(cvp.training(k), :);
        y = ytrain(cvp.training(k), :);
        Xvalid = Xtrain(cvp.test(k), :);
        yvalid = ytrain(cvp.test(k), :);

        nca = fsrnca(X, y, 'FitMethod', 'exact', ...
                    'Lambda', lambdaval(i), ...
                    'Standardize', 1, 'LossFunction', 'mad');

        % Select features using the feature weights and a relative
        % threshold.
        tol = 1e-3;
        selidx = nca.FeatureWeights > tol * max(1, max(nca.FeatureWeights));

        % Fit a non-ARD GPR model using selected features.
        gpr = fitrgp(X(:, selidx), y, 'Standardize', 1, ...
```

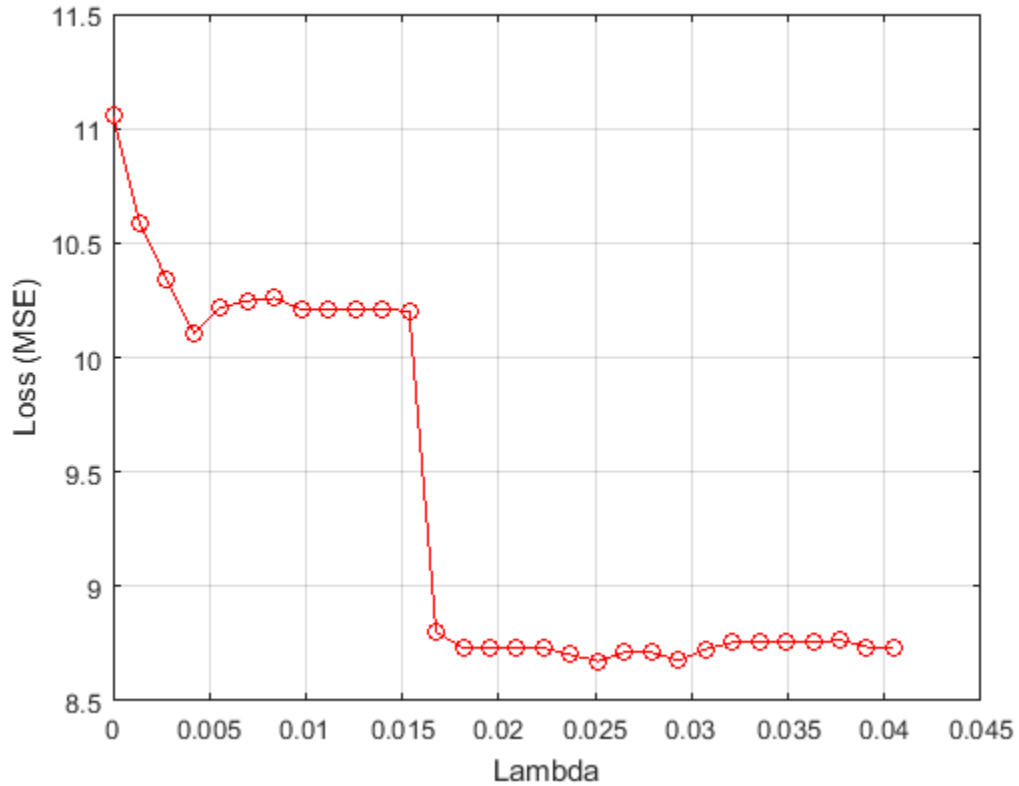
```
        'KernelFunction','squareexponential','Verbose',0);

        lossvals(i,k) = loss(gpr,Xvalid(:,selidx),yvalid);

    end
end
```

Compute the average loss obtained from the folds for each  $\lambda$  value. Plot the mean loss versus the  $\lambda$  values.

```
meanloss = mean(lossvals,2);
figure;
plot(lambdaval,meanloss,'ro-');
xlabel('Lambda');
ylabel('Loss (MSE)');
grid on;
```



Find the  $\lambda$  value that produces the minimum loss value.

```
[~,idx] = min(meanloss);  
bestlambda = lambdavals(idx)
```

```
bestlambda =
```

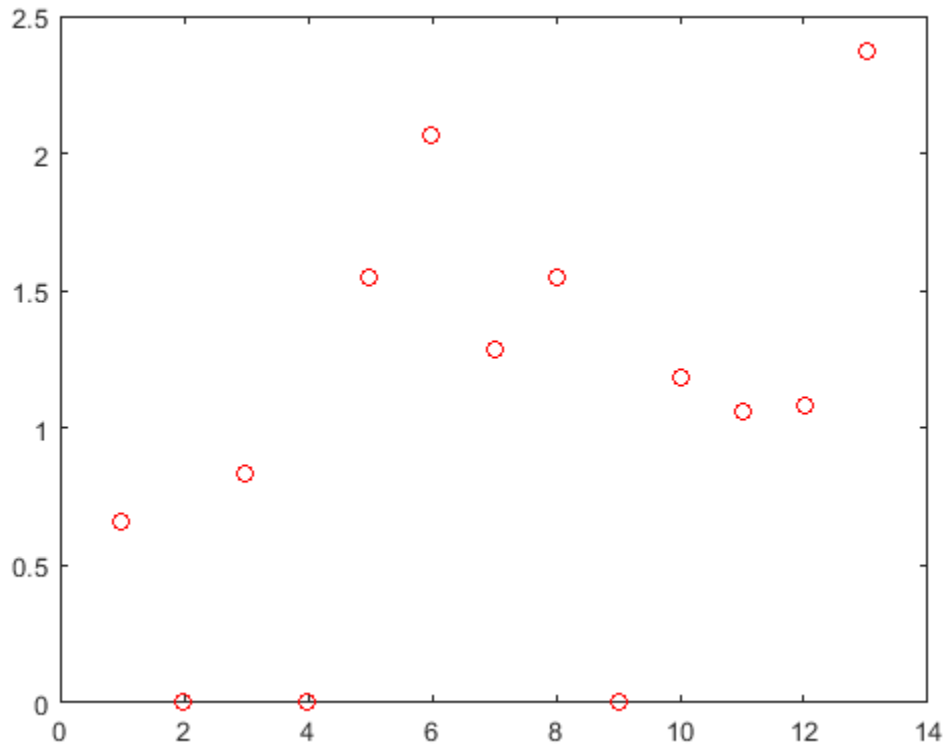
```
0.0251
```

Perform feature selection for regression using the best  $\lambda$  value. Standardize the predictor values.

```
nca2 = fsrnca(Xtrain,ytrain,'Standardize',1,'Lambda',bestlambda,...  
             'LossFunction','mad');
```

Plot the feature weights.

```
figure()  
plot(nca.FeatureWeights,'ro')
```



Compute the loss using the new nca model on the test data, which is not used to select the features.

```
L2 = loss(nca2,Xtest,ytest,'LossFunction','mad')
```

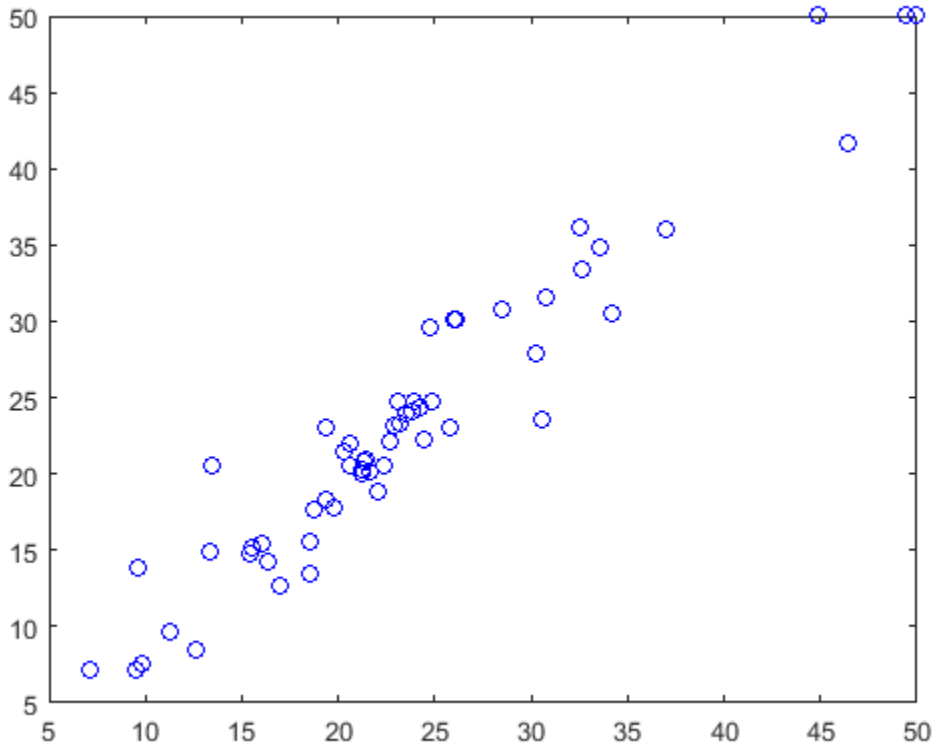
```
L2 =
```

2.0560

Tuning the regularization parameter helps identify the relevant features and reduces the loss.

Plot the predicted versus the actual response values in the test set.

```
ypred = predict(nca2,Xtest);  
figure;  
plot(ypred,ytest,'bo');
```



The predicted response values seem to be closer to the actual values as well.

## References

[1] Harrison, D. and D.L., Rubinfeld. "Hedonic prices and the demand for clean air." J. Environ. Economics & Management. Vol.5, 1978, pp. 81-102.

[2] Lichman, M. UCI Machine Learning Repository, Irvine, CA: University of California, School of Information and Computer Science, 2013. <http://archive.ics.uci.edu/ml>.

## See Also

`FeatureSelectionNCARegression` | `fsrnca` | `loss` | `refit`

**Introduced in R2016b**

# predict

**Class:** NonLinearModel

Predict response of nonlinear regression model

## Syntax

```
ypred = predict (mdl, Xnew)  
[ypred, yci] = predict (mdl, Xnew)  
[ypred, yci] = predict (mdl, Xnew, Name, Value)
```

## Description

`ypred = predict (mdl, Xnew)` returns the predicted response of the mdl nonlinear regression model to the points in X<sub>new</sub>.

`[ypred, yci] = predict (mdl, Xnew)` returns confidence intervals for the true mean responses.

`[ypred, yci] = predict (mdl, Xnew, Name, Value)` predicts responses with additional options specified by one or more Name, Value pair arguments.

## Input Arguments

### **mdl**

Nonlinear regression model, constructed by `fitnlm`.

### **X<sub>new</sub>**

Points at which `mdl` predicts responses.

- If X<sub>new</sub> is a table or dataset array, it must contain the predictor names in `mdl`.

- If `Xnew` is a numeric matrix, it must have the same number of variables (columns) as was used to create `mdl`. Furthermore, all variables used in creating `mdl` must be numeric.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### Alpha

Positive scalar from 0 to 1. Confidence level of `yci` is  $100(1 - \text{alpha})\%$ .

**Default:** `0.05`, meaning a 95% confidence interval.

### Prediction

Type of prediction:

- `'curve'` — `predict` predicts confidence bounds for the fitted mean values.
- `'observation'` — `predict` predicts confidence bounds for the new observations. This results in wider bounds because the error in a new observation is equal to the error in the estimated mean value, plus the variability in the observation from the true mean.

For details, see `polyconf`.

**Default:** `'curve'`

### Simultaneous

Logical value specifying whether the confidence bounds are for all predictor values simultaneously (`true`), or hold for each individual predictor value (`false`). Simultaneous bounds are wider than separate bounds, because it is more stringent to require that the entire curve be within the bounds than to require that the curve at a single predictor value be within the bounds.

For details, see `polyconf`.



**Default:** `false`

### **Weights**

Vector of real, positive value weights or a function handle.

- If you specify a vector, then it must have the same number of elements as the number of observations (or rows) in `Xnew`.
- If you specify a function handle, then the function must accept a vector of predicted response values as input, and return a vector of real positive weights as output.

Given weights, `W`, `predict` estimates the error variance at observation `i` by  $MSE * (1/W(i))$ , where `MSE` is the mean squared error.

**Default:** No weights

## Output Arguments

### **`ypred`**

Predicted mean values at `Xnew`. `ypred` is the same size as each component of `Xnew`.

### **`yci`**

Confidence intervals, a two-column matrix with each row providing one interval. The meaning of the confidence interval depends on the settings of the name-value pairs.

## Examples

### **Predict Responses**

Create a nonlinear model of car mileage as a function of weight, and predict the response.

Create an exponential model of car mileage as a function of weight from the `carsmall` data. Scale the weight by a factor of 1000 so all the variables are roughly equal in size.

```
load carsmall
X = Weight;
```

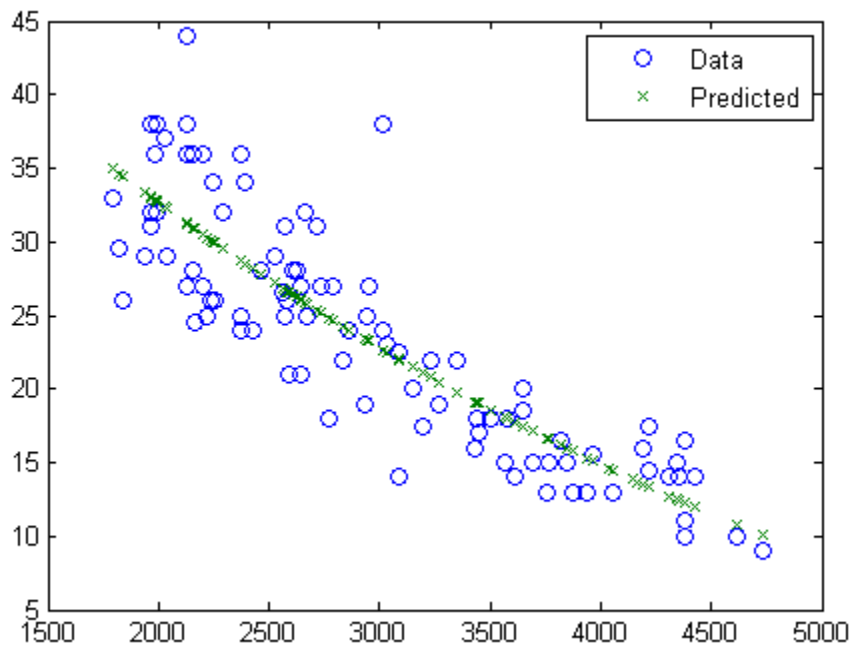
```
y = MPG;  
modelfun = 'y ~ b1 + b2*exp(-b3*x/1000)';  
beta0 = [1 1 1];  
mdl = fitnlm(X,y,modelfun,beta0);
```

Create predicted responses to the data.

```
Xnew = X;  
ypred = predict(mdl,Xnew);
```

Plot the original responses and the predicted responses to see how they differ.

```
plot(X,y,'o',X,ypred,'x')  
legend('Data','Predicted')
```



## Confidence Intervals for Predictions

Create a nonlinear model of car mileage as a function of weight, and examine confidence intervals of some responses.

Create an exponential model of car mileage as a function of weight from the `carsmall` data. Scale the weight by a factor of 1000 so all the variables are roughly equal in size.

```
load carsmall
X = Weight;
y = MPG;
modelfun = 'y ~ b1 + b2*exp(-b3*x/1000)';
beta0 = [1 1 1];
mdl = fitnlm(X,y,modelfun,beta0);
```

Create predicted responses to the smallest, mean, and largest data points.

```
Xnew = [min(X);mean(X);max(X)];
[ypred,yci] = predict(mdl,Xnew)
```

```
ypred =
    34.9469
    22.6868
    10.0617
```

```
yci =
    32.5212    37.3726
    21.4061    23.9674
     7.0148    13.1086
```

## Simultaneous Confidence Intervals for Robust Fit Curve

Generate sample data from the nonlinear regression model

$$y = b_1 + b_2 \exp\{-b_3 x\} + \varepsilon,$$

where  $b_1$ ,  $b_2$ , and  $b_3$  are coefficients, and the error term is normally distributed with mean 0 and standard deviation 0.5.

```
modelfun = @(b,x) (b(1)+b(2)*exp(-b(3)*x));

rng('default') % for reproducibility
b = [1;3;2];
```

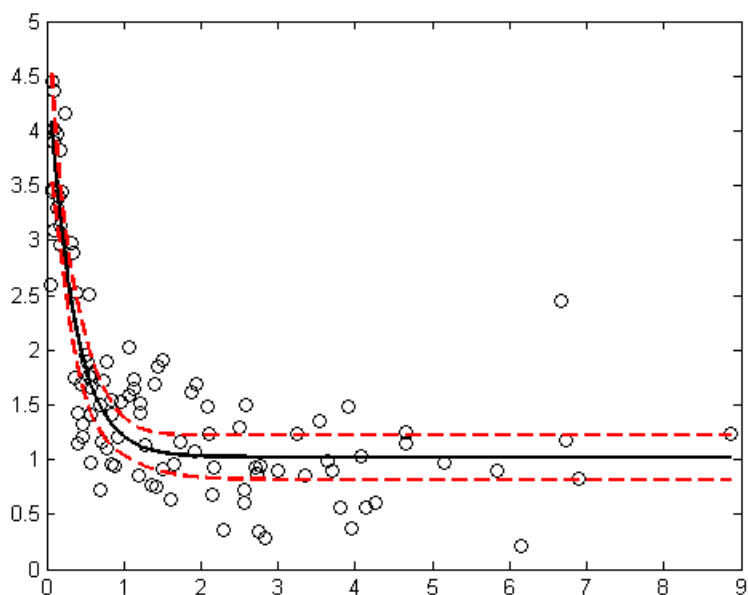
```
x = exprnd(2,100,1);  
y = modelfun(b,x) + normrnd(0,0.5,100,1);
```

Fit the nonlinear model using robust fitting options.

```
opts = statset('nlinfit');  
opts.RobustWgtFun = 'bisquare';  
b0 = [2;2;2];  
mdl = fitnlm(x,y,modelfun,b0,'Options',opts);
```

Plot the fitted regression model and simultaneous 95% confidence bounds.

```
xrange = [min(x):.01:max(x)]';  
[ypred,yci] = predict(mdl,xrange,'Simultaneous',true);  
  
figure()  
plot(x,y,'ko') % observed data  
hold on  
plot(xrange,ypred,'k','LineWidth',2)  
plot(xrange,yci,'r--','LineWidth',1.5)
```



### Confidence Interval Using Observation Weights

Load sample data.

```
S = load('reaction');
X = S.reactants;
y = S.rate;
beta0 = S.beta;
```

Specify a function handle for observation weights, then fit the Hougen-Watson model to the rate data using the specified observation weights function.

```
a = 1; b = 1;
weights = @(yhat) 1./((a + b*abs(yhat)).^2);
mdl = fitnlm(X,y,@hougen,beta0,'Weights',weights);
```

Compute the 95% prediction interval for a new observation with reactant levels [100,100,100] using the observation weight function.

```
[ypred, yci] = predict mdl, [100, 100, 100], 'Prediction', 'observation', ...  
                                'Weights', weights)
```

```
ypred =  
  
    1.8149
```

```
yci =  
  
    1.5264    2.1033
```

- “Predict or Simulate Responses Using a Nonlinear Model” on page 13-10
- “Nonlinear Regression Workflow” on page 13-15

## Tips

- For predictions with added noise, use `random`.
- For a syntax that can be easier to use with models created from tables or dataset arrays, try `feval`.

## References

- [1] Lane, T. P. and W. H. DuMouchel. “Simultaneous Confidence Intervals in Multiple Regression.” *The American Statistician*. Vol. 48, No. 4, 1994, pp. 315–321.
- [2] Seber, G. A. F., and C. J. Wild. *Nonlinear Regression*. Hoboken, NJ: Wiley-Interscience, 2003.

## See Also

`NonLinearModel` | `random`

## Topics

- “Predict or Simulate Responses Using a Nonlinear Model” on page 13-10
- “Nonlinear Regression Workflow” on page 13-15
- “Nonlinear Regression” on page 13-2

# predict

**Class:** RepeatedMeasuresModel

Compute predicted values given predictor values

## Syntax

```
ypred = predict(rm, tnew)  
ypred = predict(rm, tnew, Name, Value)  
[ypred, yci] = predict(____)
```

## Description

`ypred = predict(rm, tnew)` returns the predicted values from the repeated measures model `rm` using the predictor values from the table `t`.

`ypred = predict(rm, tnew, Name, Value)` returns the predicted values from the repeated measures model `rm` with additional options specified by one or more `Name, Value` pair arguments.

For example, you can specify the within-subjects design matrix.

`[ypred, yci] = predict(____)` also returns the 95% confidence interval for the predicted values.

## Input Arguments

**rm** — Repeated measures model

RepeatedMeasuresModel object

Repeated measures model, returned as a RepeatedMeasuresModel object.

For properties and methods of this object, see RepeatedMeasuresModel.

**tnew — New data**table used to create `rm` (default) | table

New data including the values of the response variables and the between-subject factors used as predictors in the repeated measures model, `rm`, specified as a table. `tnew` must contain all of the between-subject factors used to create `rm`.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

**Alpha — Confidence level**

0.05 (default) | scalar value in the range of 0 through 1

Confidence level of the confidence intervals for the predicted values, specified as the comma-separated pair consisting of 'alpha' and a scalar value in the range of 0 to 1. The confidence level is  $100 \cdot (1 - \text{alpha})\%$ .

Example: 'alpha', 0.01

Data Types: double | single

**WithinModel — Model for within-subject factors**

'separatemeans' | 'orthogonalcontrasts' | character vector

Model for the within-subject factors, specified as the comma-separated pair consisting of 'WithinModel' and one of the following:

- 'separatemeans' — Compute a separate mean for each group.
- 'orthogonalcontrasts' — Valid when the within-subject design consists of a single numeric factor  $T$ . This specifies a model consisting of orthogonal polynomials up to order  $T^{(r-1)}$ , where  $r$  is the number of repeated measures.
- A character vector that defines a model specification in the within-subject factors.

Example: 'WithinModel', 'orthogonalcontrasts'

**WithinDesign — Design for within-subject factors**

vector | matrix | table



Design for within-subject factors, specified as the comma-separated pair consisting of 'WithinDesign' and a vector, matrix, or a table. It provides the values of the within-subject factors in the same form as the `RM.WithinDesign` property.

Example: `'WithinDesign', 'Time'`

Data Types: `single` | `double` | `table`

## Output Arguments

### **`ypred` — Predicted values**

*n*-by-*r* matrix

Predicted values from the repeated measures model `rm`, returned as an *n*-by-*r* matrix, where *n* is the number of rows in `tnew` and *r* is the number of repeated measures in `rm`.

### **`yxi` — Confidence intervals for predicted values**

*n*-by-*r*-by-2 matrix

Confidence intervals for predicted values from the repeated measures model `rm`, returned as an *n*-by-*r*-by-2 matrix.

These are nonsimultaneous intervals for predicting the mean response at the specified predictor values. For predicted value `ypred(i, j)`, the lower limit of the interval is `yxi(i, j, 1)` and the upper limit is `yxi(i, j, 2)`.

## Examples

### **Predict Response Values**

Load the sample data.

```
load fisheriris
```

The column vector, `species` consists of iris flowers of three different species: `setosa`, `versicolor`, and `virginica`. The double matrix `meas` consists of four types of measurements on the flowers: the length and width of sepals and petals in centimeters, respectively.

Store the data in a table array.

```
t = table(species,meas(:,1),meas(:,2),meas(:,3),meas(:,4),...  
'VariableNames',{ 'species', 'meas1', 'meas2', 'meas3', 'meas4' });  
Meas = dataset([1 2 3 4]', 'VarNames', {'Measurements'});
```

Fit a repeated measures model, where the measurements are the responses and the species is the predictor variable.

```
rm = fitrm(t, 'meas1-meas4~species', 'WithinDesign', Meas);
```

Predict responses for the three species.

```
Y = predict(rm,t([1 51 101],:))
```

Y =

5.0060	3.4280	1.4620	0.2460
5.9360	2.7700	4.2600	1.3260
6.5880	2.9740	5.5520	2.0260

## Predict Response Values and Plot Predictions

Navigate to the folder containing sample data.

```
cd(matlabroot)  
cd('help/toolbox/stats/examples')
```

Load the sample data.

```
load('longitudinalData')
```

The matrix `Y` contains response data for 16 individuals. The response is the blood level of a drug measured at five time points (time = 0, 2, 4, 6, and 8). Each row of `Y` corresponds to an individual, and each column corresponds to a time point. The first eight subjects are female, and the second eight subjects are male. This is simulated data.

Define a variable that stores gender information.

```
Gender = ['F' 'F' 'F' 'F' 'F' 'F' 'F' 'F' 'M' 'M' 'M' 'M' 'M' 'M' 'M' 'M'];
```

Store the data in a proper table array format to perform repeated measures analysis.

```
t = table(Gender,Y(:,1),Y(:,2),Y(:,3),Y(:,4),Y(:,5),...  
'VariableNames',{ 'Gender', 't0', 't2', 't4', 't6', 't8' });
```

Define the within-subjects variable.

```
Time = [0 2 4 6 8]';
```

Fit a repeated measures model, where the blood levels are the responses and gender is the predictor variable.

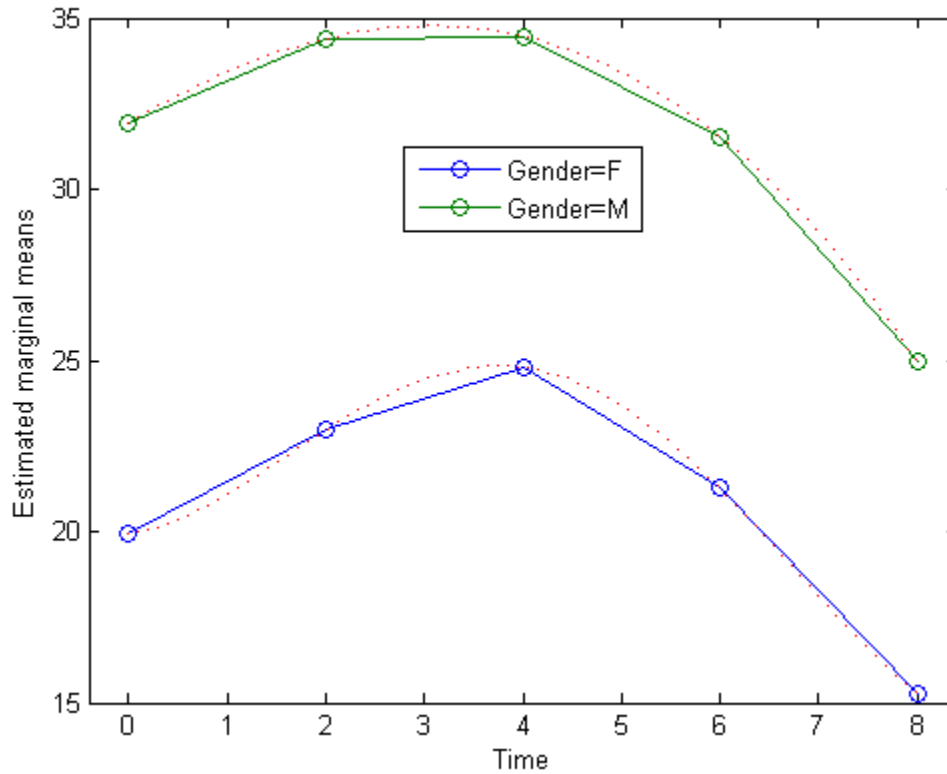
```
rm = fitrm(t, 't0-t8 ~ Gender', 'WithinDesign', Time);
```

Predict the responses at intermediate times.

```
time = linspace(0,8)';  
Y = predict(rm, t([1 5 8 12],:), ...  
    'WithinModel', 'orthogonalcontrasts', 'WithinDesign', time);
```

Plot the predictions along with the estimated marginal means.

```
plotprofile(rm, 'Time', 'Group', {'Gender'})  
hold on;  
plot(time, Y, 'Color', 'r', 'LineStyle', ':');  
hold off
```



### Compute and Plot Confidence Intervals

Navigate to the folder containing sample data.

```
cd(matlabroot)
cd('help/toolbox/stats/examples')
```

Load the sample data.

```
load('longitudinalData')
```

The matrix  $Y$  contains response data for 16 individuals. The response is the blood level of a drug measured at five time points (time = 0, 2, 4, 6, and 8). Each row of  $Y$  corresponds

to an individual, and each column corresponds to a time point. The first eight subjects are female, and the second eight subjects are male. This is simulated data.

Define a variable that stores gender information.

```
Gender = ['F' 'F' 'F' 'F' 'F' 'F' 'F' 'F' 'M' 'M' 'M' 'M' 'M' 'M' 'M' 'M'];
```

Store the data in a proper table array format to perform repeated measures analysis.

```
t = table(Gender, Y(:,1), Y(:,2), Y(:,3), Y(:,4), Y(:,5), ...
'VariableNames', {'Gender', 't0', 't2', 't4', 't6', 't8'});
```

Define the within-subjects variable.

```
Time = [0 2 4 6 8]';
```

Fit a repeated measures model, where the blood levels are the responses and gender is the predictor variable.

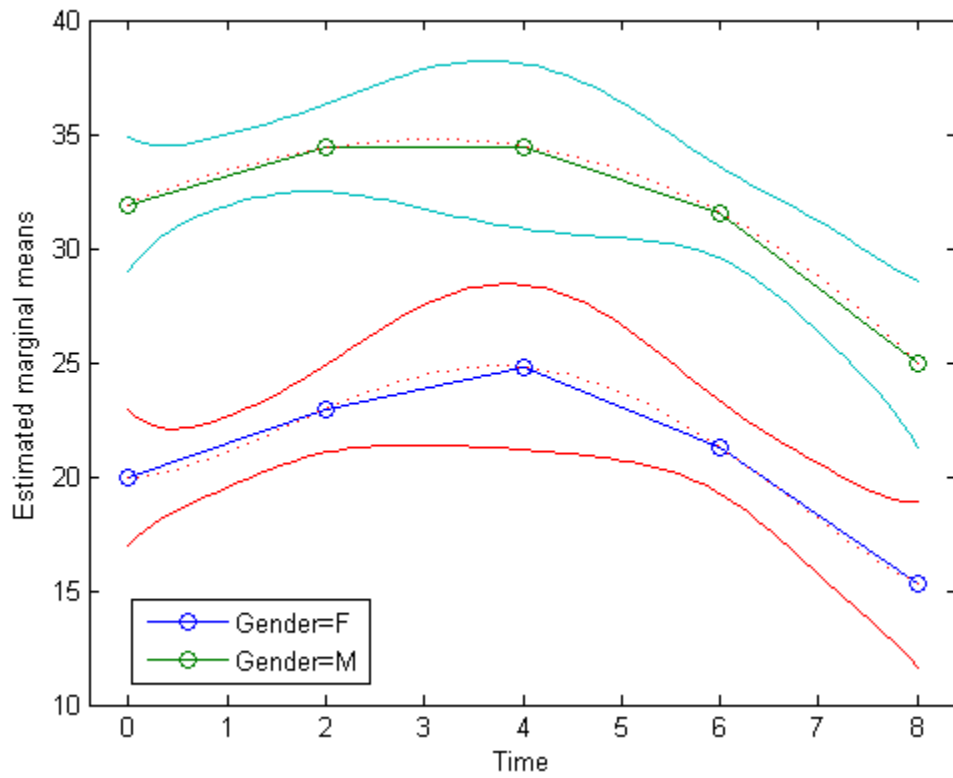
```
rm = fitrm(t, 't0-t8 ~ Gender', 'WithinDesign', Time);
```

Predict the responses at intermediate times.

```
time = linspace(0,8)';
[ypred, ypredci] = predict(rm, t([1 5 8 12],:), ...
'WithinModel', 'orthogonalcontrasts', 'WithinDesign', time);
```

Plot the predictions and the confidence intervals for predictions along with the estimated marginal means.

```
plotprofile(rm, 'Time', 'Group', {'Gender'})
hold on;
plot(time, ypred, 'Color', 'r', 'LineStyle', ':');
plot(time, ypredci(:, :, 1))
plot(time, ypredci(:, :, 2))
hold off
```



## See Also

`fitrm` | `random`

---

# predict

**Class:** TreeBagger

Predict responses using ensemble of bagged decision trees

## Syntax

```
Yfit = predict(B,X)
[Yfit,stdevs] = predict(B,X)
[Yfit,scores] = predict(B,X)
[Yfit,scores,stdevs] = predict(B,X)
Yfit = predict(B,X,'param1',val1,'param2',val2,...)
```

## Description

`Yfit = predict(B,X)` returns a vector of predicted responses for the predictor data in the table or matrix `X`, based on the ensemble of bagged decision trees `B`. `Yfit` is a cell array of character vectors for classification and a numeric array for regression. By default, `predict` takes a democratic (nonweighted) average vote from all trees in the ensemble.

`B` is a trained `TreeBagger` model object, that is, a model returned by `TreeBagger`.

`X` is a table or matrix of predictor data used to generate responses. Rows represent observations and columns represent variables.

- If `X` is a numeric matrix:
  - The variables making up the columns of `X` must have the same order as the predictor variables that trained `B`.
  - If you trained `B` using a table (for example, `Tbl`), then `X` can be a numeric matrix if `Tbl` contains all numeric predictor variables. To treat numeric predictors in `Tbl` as categorical during training, identify categorical predictors using the `CategoricalPredictors` name-value pair argument of `TreeBagger`. If `Tbl`

contains heterogeneous predictor variables (for example, numeric and categorical data types) and `X` is a numeric matrix, then `predict` throws an error.

- If `X` is a table:
  - `predict` does not support multi-column variables and cell arrays other than cell arrays of character vectors.
  - If you trained `B` using a table (for example, `Tbl`), then all predictor variables in `X` must have the same variable names and be of the same data types as those that trained `B` (stored in `B.PredictorNames`). However, the column order of `X` does not need to correspond to the column order of `Tbl`. `Tbl` and `X` can contain additional variables (response variables, observation weights, etc.), but `predict` ignores them.
  - If you trained `B` using a numeric matrix, then the predictor names in `B.PredictorNames` and corresponding predictor variable names in `X` must be the same. To specify predictor names during training, see the `PredictorNames` name-value pair argument of `TreeBagger`. All predictor variables in `X` must be numeric vectors. `X` can contain additional variables (response variables, observation weights, etc.), but `predict` ignores them.

For regression, `[Yfit, stdevs] = predict(B, X)` also returns standard deviations of the computed responses over the ensemble of the grown trees.

For classification, `[Yfit, scores] = predict(B, X)` also returns scores for all classes. `scores` is a matrix with one row per observation and one column per class. For each observation and each class, the score generated by each tree is the probability of this observation originating from this class computed as the fraction of observations of this class in a tree leaf. `predict` averages these scores over all trees in the ensemble.

`[Yfit, scores, stdevs] = predict(B, X)` also returns standard deviations of the computed scores for classification. `stdevs` is a matrix with one row per observation and one column per class, with standard deviations taken over the ensemble of the grown trees.

`Yfit = predict(B, X, 'param1', val1, 'param2', val2, ...)` specifies optional parameter name/value pairs:

'Trees'	Array of tree indices to use for computation of responses. Default is 'all'.
---------	--



'TreeWeights'	Array of NTrees weights for weighting votes from the specified trees.
'UseInstanceForTree'	Logical matrix of size Nobs-by-NTrees indicating which trees to use to make predictions for each observation. By default all trees are used for all observations.

## Algorithms

- For regression problems, the predicted response for an observation is the weighted average of the predictions using selected trees only. That is,

$$\hat{y}_{\text{bag}} = \frac{1}{\sum_{t=1}^T \alpha_t I(t \in S)} \sum_{t=1}^T \alpha_t \hat{y}_t I(t \in S).$$

- $\hat{y}_t$  is the prediction from tree  $t$  in the ensemble.
- $S$  is the set of indices of selected trees that comprise the prediction (see 'Trees' and 'UseInstanceForTree').  $I(t \in S)$  is 1 if  $t$  is in the set  $S$ , and 0 otherwise.
- $\alpha_t$  is the weight of tree  $t$  (see 'TreeWeights').
- For classification problems, the predicted class for an observation is the class that yields the largest weighted average of the class posterior probabilities (i.e., classification scores) computed using selected trees only. That is,

1

For each class  $c \in C$  and each tree  $t = 1, \dots, T$ , `predict` computes  $\hat{P}_t(c|x)$ , which is the estimated posterior probability of class  $c$  given observation  $x$  using tree  $t$ .  $C$  is the set of all distinct classes in the training data. For more details on classification tree posterior probabilities, see `fitctree` and `predict`.

- 2 `predict` computes the weighted average of the class posterior probabilities over the selected trees.

$$\hat{P}_{\text{bag}}(c|x) = \frac{1}{\sum_{t=1}^T \alpha_t I(t \in S)} \sum_{t=1}^T \alpha_t \hat{P}_t(c|x) I(t \in S).$$

- 3 The predicted class is the class that yields the largest weighted average.

$$\hat{y}_{\text{bag}} = \arg \max_{c \in C} \{ \hat{P}_{\text{bag}}(c | x) \}.$$

## See Also

`TreeBagger` | `error` | `oobPredict` | `predict` | `quantilePredict`

## Topics

“Bootstrap Aggregation (Bagging) of Regression Trees” on page 18-141

“Bootstrap Aggregation (Bagging) of Classification Trees” on page 18-154

# predictConstraints

Predict coupled constraint violations at a set of points

## Syntax

```
ConstraintViolations = predictConstraints(results,XTable)
[ConstraintViolations,sigma] = predictConstraints(results,XTable)
```

## Description

`ConstraintViolations = predictConstraints(results,XTable)` returns the coupled constraint function violations at the points in `XTable`.

`[ConstraintViolations,sigma] = predictConstraints(results,XTable)` also returns the standard deviations of the coupled constraint functions.

## Examples

### Predict Coupled Constraints

This example shows how to predict the coupled constraints of an optimized SVM model. For details of this model, see “Optimize a Cross-Validated SVM Classifier Using Bayesian Optimization” on page 10-53.

```
rng default
grnpop = mvnrnd([1,0],eye(2),10);
redpop = mvnrnd([0,1],eye(2),10);
redpts = zeros(100,2);
grnpts = redpts;
for i = 1:100
    grnpts(i,:) = mvnrnd(grnpop(randi(10),:),eye(2)*0.02);
    redpts(i,:) = mvnrnd(redpop(randi(10),:),eye(2)*0.02);
end
cdata = [grnpts;redpts];
```

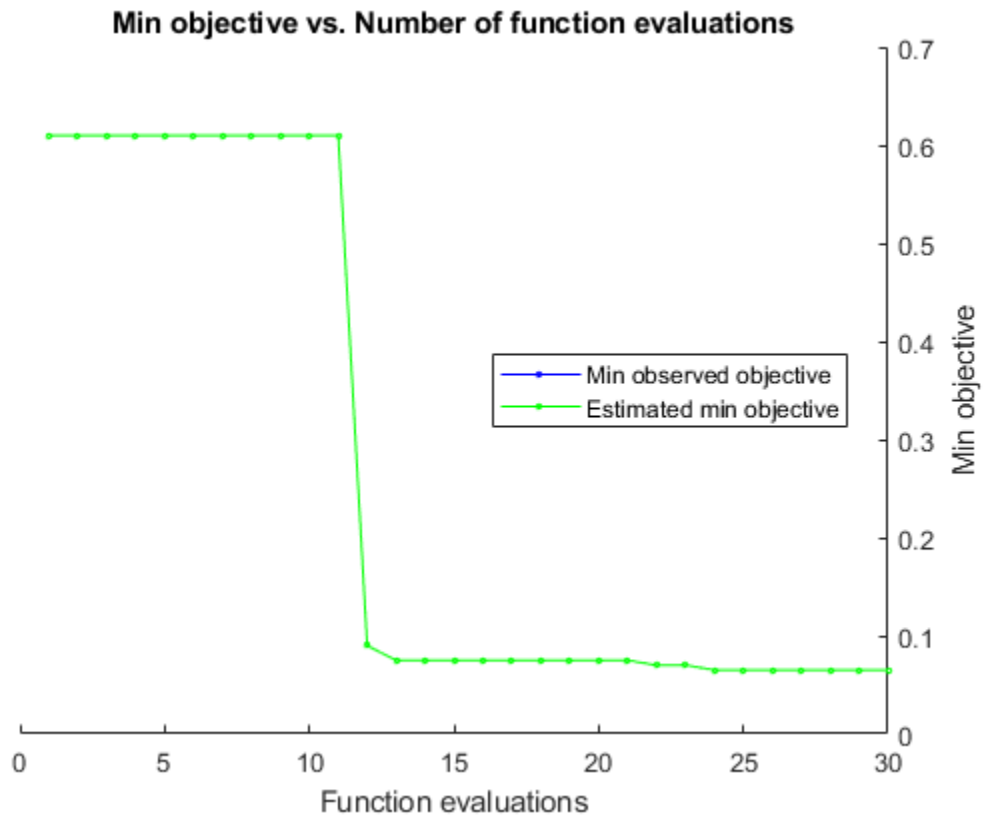
```
grp = ones(200,1);
grp(101:200) = -1;
c = cvpartition(200, 'KFold', 10);
sigma = optimizableVariable('sigma', [1e-5, 1e5], 'Transform', 'log');
box = optimizableVariable('box', [1e-5, 1e5], 'Transform', 'log');
```

The objective function is the cross-validation loss of the SVM model for the partition `c`. The coupled constraint is the number of support vectors in the model minus 100. The model has 200 data points, so the coupled constraint values range from -100 to 100. Positive values mean the constraint is not satisfied.

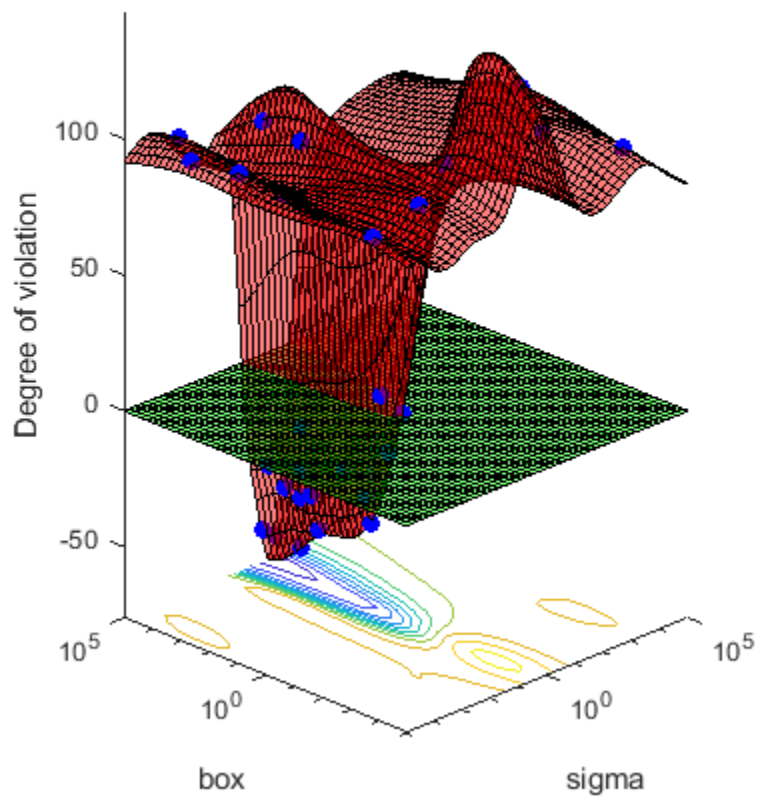
```
function [objective, constraint] = mysvmfun(x, cdata, grp, c)
SVMModel = fitcsvm(cdata, grp, 'KernelFunction', 'rbf', ...
    'BoxConstraint', x.box, ...
    'KernelScale', x.sigma);
cvModel = crossval(SVMModel, 'CVPartition', c);
objective = kfoldLoss(cvModel);
constraint = sum(SVMModel.IsSupportVector) - 100.5;
```

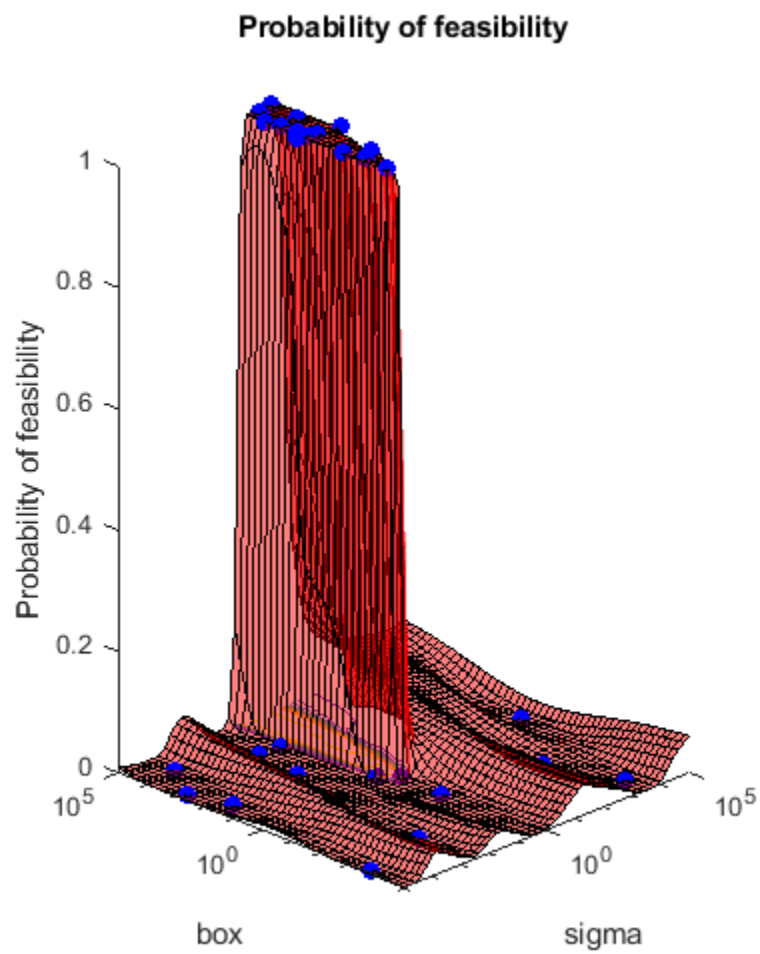
Call the optimizer using this function and its one coupled constraint.

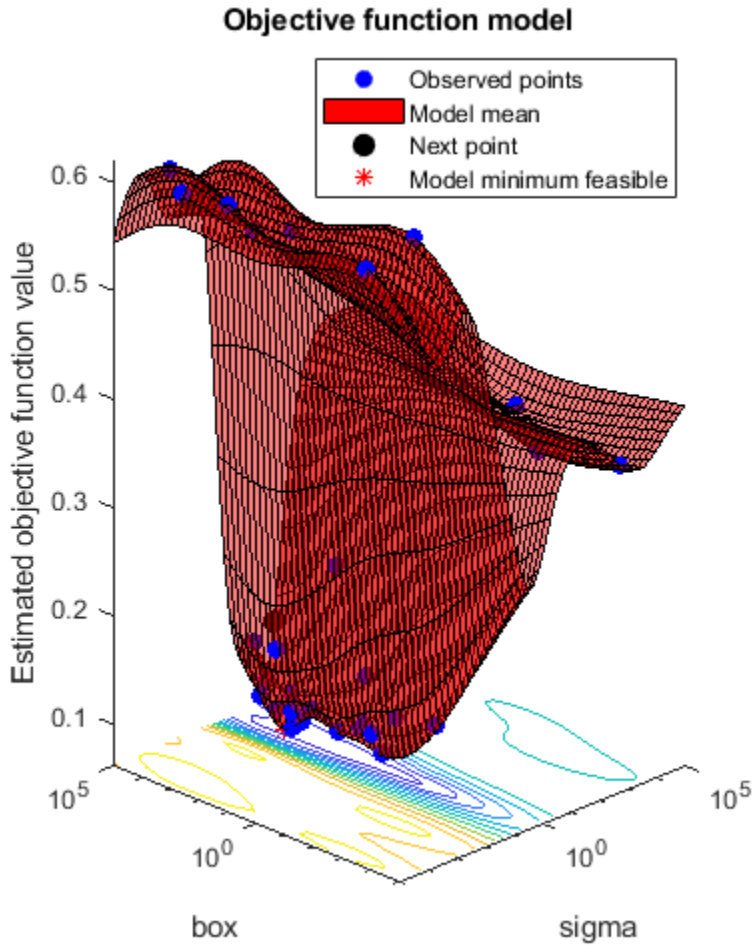
```
fun = @(x) mysvmfun(x, cdata, grp, c);
results = bayesopt(fun, [sigma, box], 'IsObjectiveDeterministic', true, ...
    'NumCoupledConstraints', 1, 'PlotFcn', ...
    {@plotMinObjective, @plotConstraintModels, @plotObjectiveModel}, ...
    'AcquisitionFunctionName', 'expected-improvement-plus', 'Verbose', 0);
```



Constraint 1 model







The constraint model plot shows that most parameters in the range are infeasible, and are feasible only for relatively high values of the `box` parameter and a small range of the `sigma` parameter. Predict the coupled constraint values for several values of the control variables `box` and `sigma`.

```
sigma = logspace(-2,2,11)';
box = logspace(0,5,11)';
```



```
XTable = table(sigma,box);
cons = predictConstraints(results,XTable);
[XTable,table(cons)]
```

```
ans =
```

```
11x3 table
```

sigma	box	cons
0.01	1	99.443
0.025119	3.1623	106.49
0.063096	10	94.468
0.15849	31.623	25.134
0.39811	100	-38.732
1	316.23	-55.156
2.5119	1000	-34.181
6.3096	3162.3	5.0153
15.849	10000	39.465
39.811	31623	60.9
100	1e+05	71.906

## Input Arguments

### **results** — Bayesian optimization results

BayesianOptimization object

Bayesian optimization results, specified as a BayesianOptimization object.

### **XTable** — Prediction points

table with D columns

Prediction points, specified as a table with D columns, where D is the number of variables in the problem. The function performs its predictions on these points.

Data Types: table

## Output Arguments

### **ConstraintViolations** — Constraint violations

N-by-K matrix

Constraint violations, returned as an N-by-K matrix, where there are N rows in `XTable` and K coupled constraints. The constraint violations are the posterior means of the Gaussian process model of the coupled constraints at the points in `XTable`.

### **sigma** — Constraint standard deviations

N-by-K matrix

Constraint standard deviations, returned as an N-by-K matrix, where there are N rows in `XTable` and K coupled constraints. The standard deviations represent those of the posterior distribution at the points in `XTable`.

## See Also

`BayesianOptimization`

**Introduced in R2016b**

# predictError

Predict error value at a set of points

## Syntax

```
error = predictError(results,XTable)
[error,sigma] = predictError(results,XTable)
```

## Description

`error = predictError(results,XTable)` returns the posterior mean of the error coupled constraint at the points in `XTable`.

`[error,sigma] = predictError(results,XTable)` also returns the posterior standard deviations.

## Examples

### Error Prediction

This example shows optimizing a function that throws an error when the evaluation point has norm larger than 2. The error model for the objective function learns this behavior.

Create variables named `x1` and `x2` that range from -5 to 5.

```
var1 = optimizableVariable('x1', [-5,5]);
var2 = optimizableVariable('x2', [-5,5]);
vars = [var1,var2];
```

The following objective function throws an error when the norm of  $x = [x1, x2]$  exceeds 2:

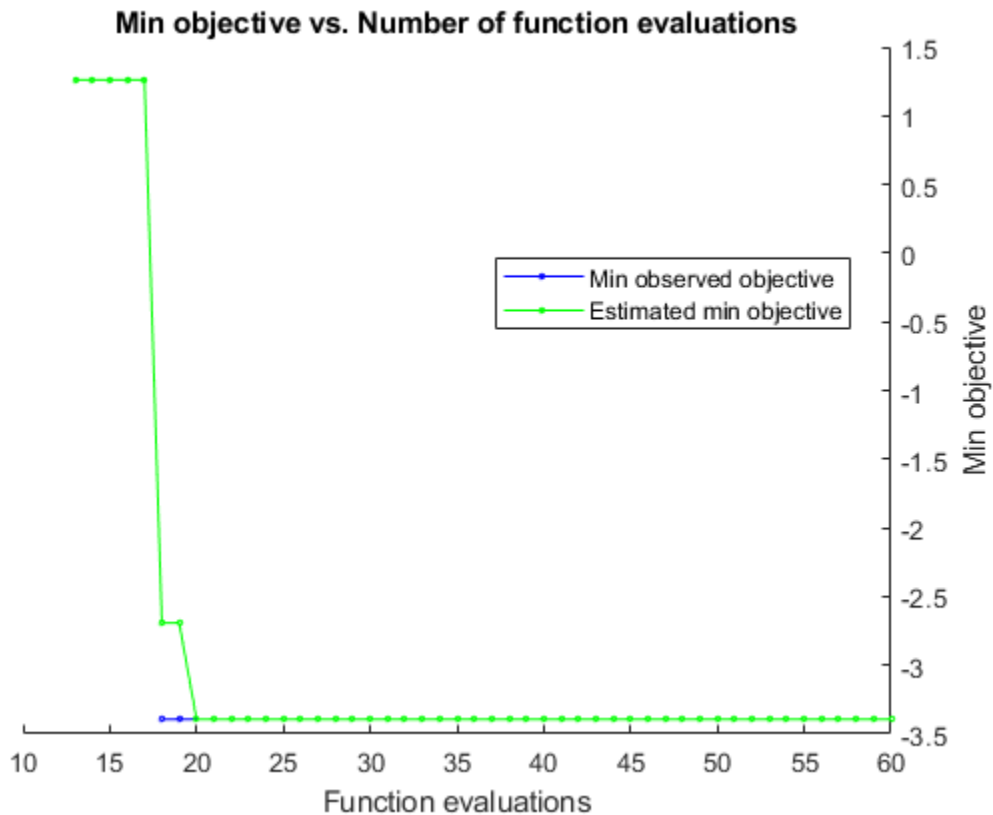
```
function f = makeanerror(x)
```

```
f = x.x1 - x.x2 - sqrt(4-x.x1^2-x.x2^2);
```

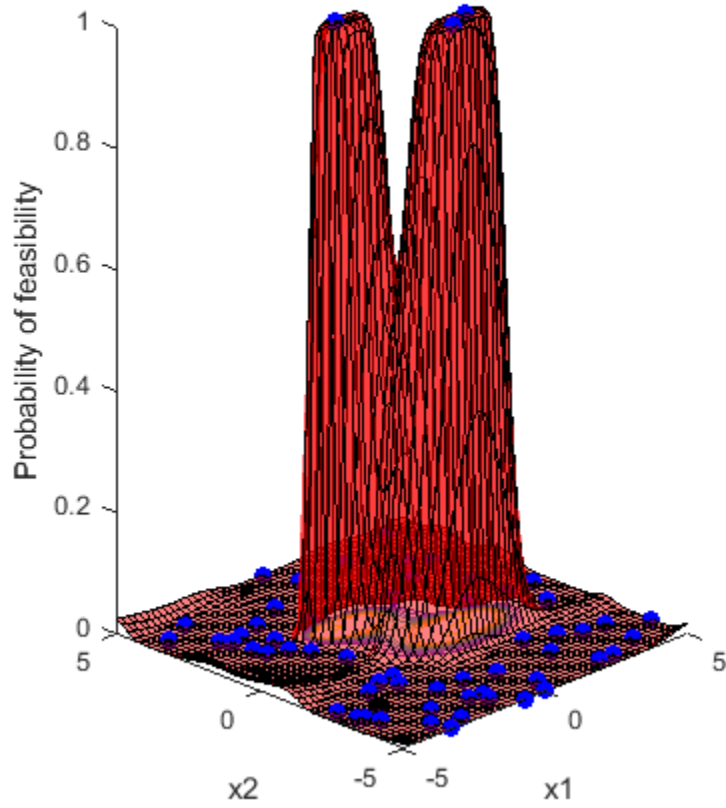
```
fun = @makeanerror;
```

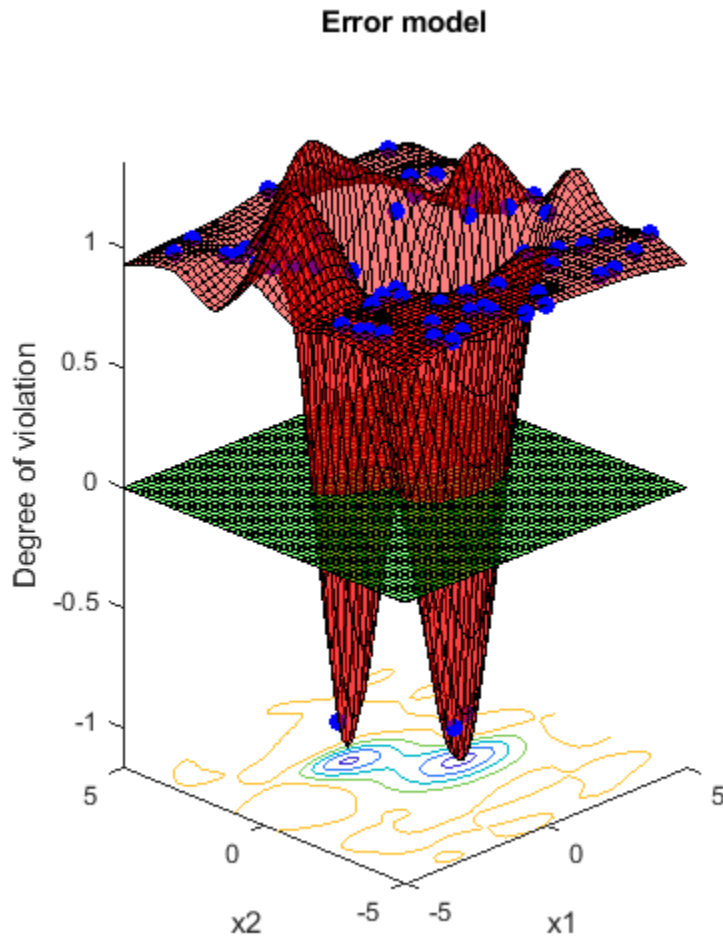
Plot the error model and minimum objective as the optimization proceeds. Optimize for 60 iterations so the error model becomes well-trained. For reproducibility, set the random seed and use the 'expected-improvement-plus' acquisition function.

```
rng default
results = bayesopt(fun,vars,'Verbose',0,'MaxObjectiveEvaluations',60,...
    'AcquisitionFunctionName','expected-improvement-plus',...
    'PlotFcn',{'@plotMinObjective','@plotConstraintModels'});
```



### Probability of feasibility





Predict the error at points on the line  $x_1 = x_2$ . If the error model were perfect, it would have value  $-1$  at every point where the norm of  $x$  is no more than  $2$ , and value  $1$  at all other points.

```
x1 = (-5:0.5:5)';
x2 = x1;
XTable = table(x1,x2);
```

```
error = predictError(results,XTable);
normx = sqrt(x1.^2 + x2.^2);
[XTable,table(normx,error)]
```

```
ans =
```

```
21x4 table
```

x1	x2	normx	error
-5	-5	7.0711	0.94663
-4.5	-4.5	6.364	0.97396
-4	-4	5.6569	0.99125
-3.5	-3.5	4.9497	1.0033
-3	-3	4.2426	1.0018
-2.5	-2.5	3.5355	0.99627
-2	-2	2.8284	1.0043
-1.5	-1.5	2.1213	0.89886
-1	-1	1.4142	0.4746
-0.5	-0.5	0.70711	0.0042389
0	0	0	-0.16004
0.5	0.5	0.70711	-0.012397
1	1	1.4142	0.30187
1.5	1.5	2.1213	0.88588
2	2	2.8284	1.0872
2.5	2.5	3.5355	0.997
3	3	4.2426	0.99861
3.5	3.5	4.9497	0.98894
4	4	5.6569	0.98941
4.5	4.5	6.364	0.98956
5	5	7.0711	0.95549

## Input Arguments

### **results** — Bayesian optimization results

BayesianOptimization object

Bayesian optimization results, specified as a BayesianOptimization object.

**XTable — Prediction points**table with  $D$  columns

Prediction points, specified as a table with  $D$  columns, where  $D$  is the number of variables in the problem. The function performs its predictions on these points.

Data Types: `table`

## Output Arguments

**error — Mean of error coupled constraint** $N$ -by-1 vector

Mean of error coupled constraint, returned as an  $N$ -by-1 vector, where  $N$  is the number of rows of `XTable`. The mean is the posterior mean of the error coupled constraint at the points in `XTable`.

`bayesopt` deems your objective function to return an error if it returns anything other than a finite real scalar. See “Objective Function Errors” on page 10-43.

**sigma — Standard deviation of error coupled constraint** $N$ -by-1 vector

Standard deviation of error coupled constraint, returned as an  $N$ -by-1 vector, where  $N$  is the number of rows of `XTable`.

## See Also

`BayesianOptimization`**Introduced in R2016b**



# predictObjective

Predict objective function at a set of points

## Syntax

```
objective = predictObjective(results,XTable)
[objective,sigma] = predictObjective(results,XTable)
```

## Description

`objective = predictObjective(results,XTable)` returns the estimated objective function value at the points in `XTable`.

`[objective,sigma] = predictObjective(results,XTable)` also returns estimated standard deviations.

## Examples

### Predict Cross-Validation Loss of an Optimized Classifier

This example shows how to estimate the cross-validation loss of an optimized classifier.

Optimize a KNN classifier for the `ionosphere` data, meaning find parameters that minimize the cross-validation loss. Minimize over nearest-neighborhood sizes from 1 to 30, and over the distance functions 'chebychev', 'euclidean', and 'minkowski'.

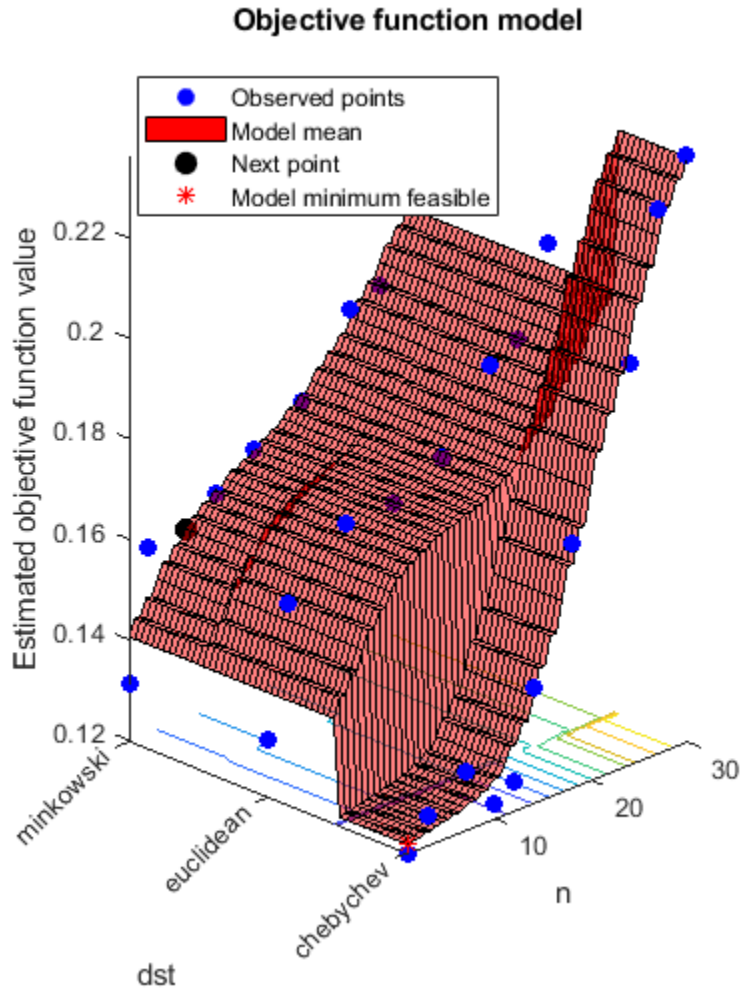
For reproducibility, set the random seed, and set the `AcquisitionFunctionName` option to 'expected-improvement-plus'.

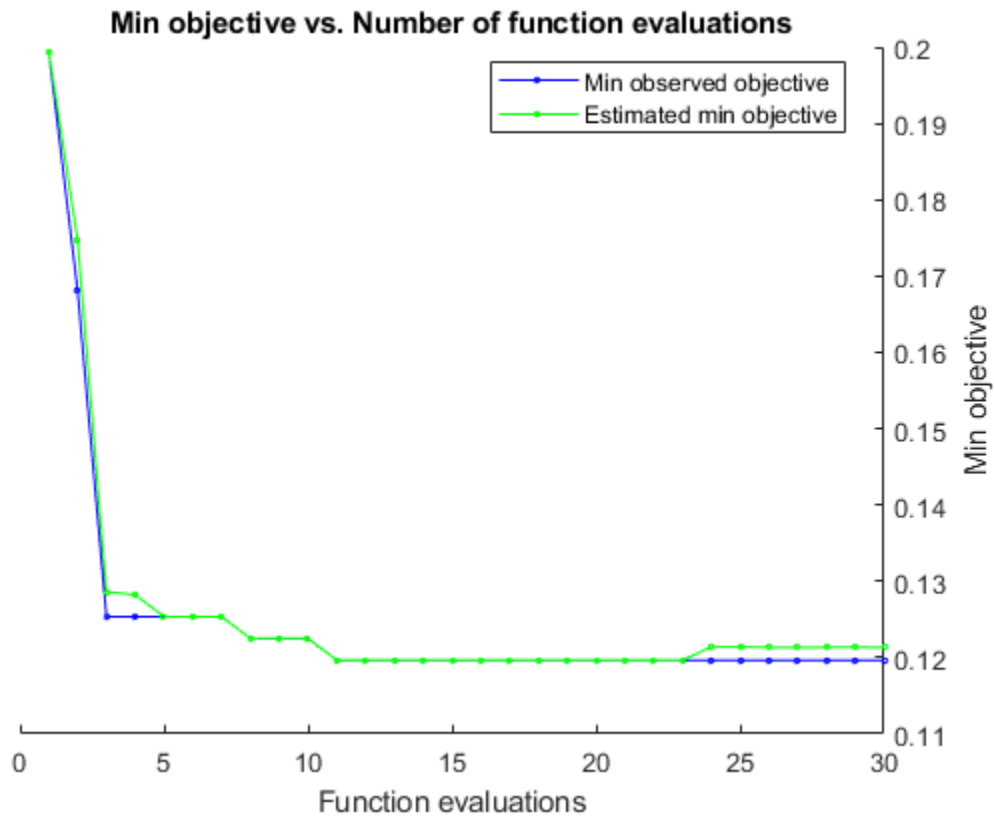
```
load ionosphere
rng default
num = optimizableVariable('n',[1,30],'Type','integer');
dst = optimizableVariable('dst',{'chebychev','euclidean','minkowski'},'Type','categorical');
c = cvpartition(351,'Kfold',5);
```

```

fun = @(x) kfoldLoss(fitcknn(X, Y, 'CVPartition', C, 'NumNeighbors', x.n, ...
    'Distance', char(x.dst), 'NSMethod', 'exhaustive'));
results = bayesopt(fun, [num, dst], 'Verbose', 0, ...
    'AcquisitionFunctionName', 'expected-improvement-plus');

```





Create a table of points to estimate.

```
b = categorical({'chebychev', 'euclidean', 'minkowski'});
n = [1;1;1;4;2;2];
dst = [b(1);b(2);b(3);b(1);b(1);b(3)];
XTable = table(n,dst);
```

Estimate the objective and standard deviation of the objective at these points.

```
[objective,sigma] = predictObjective(results,XTable);
[XTable,table(objective,sigma)]
```

ans =

6x4 table

n	dst	objective	sigma
—	—	—	—
1	chebychev	0.12132	0.0068029
1	euclidean	0.14052	0.0079128
1	minkowski	0.14057	0.0079117
4	chebychev	0.1227	0.0068805
2	chebychev	0.12176	0.0066739
2	minkowski	0.1437	0.0075448

## Input Arguments

### **results** — Bayesian optimization results

BayesianOptimization object

Bayesian optimization results, specified as a BayesianOptimization object.

### **XTable** — Prediction points

table with D columns

Prediction points, specified as a table with D columns, where D is the number of variables in the problem. The function performs its predictions on these points.

Data Types: table

## Output Arguments

### **objective** — Objective estimates

N-by-1 vector

Objective estimates, returned as an N-by-1 vector, where N is the number of rows of XTable. The estimates are the mean values of the posterior distribution of the Gaussian process model of the objective function.

### **sigma** — Standard deviations of objective function

N-by-1 vector

Standard deviations of objective function, returned as an N-by-1 vector, where N is the number of rows of `XTable`. The standard deviations are those of the posterior distribution of the Gaussian process model of the objective function.

## See Also

`BayesianOptimization` | `bayesopt`

**Introduced in R2016b**

## predictObjectiveEvaluationTime

Predict objective function run times at a set of points

### Syntax

```
time = predictObjectiveEvaluationTime(results,XTable)
```

### Description

`time = predictObjectiveEvaluationTime(results,XTable)` returns estimated objective evaluation times at the points in `XTable`.

### Examples

#### Predict Evaluation Time of Objective In an Optimized Model

This example shows how to estimate the objective function evaluation time in an optimized Bayesian model of SVM classification.

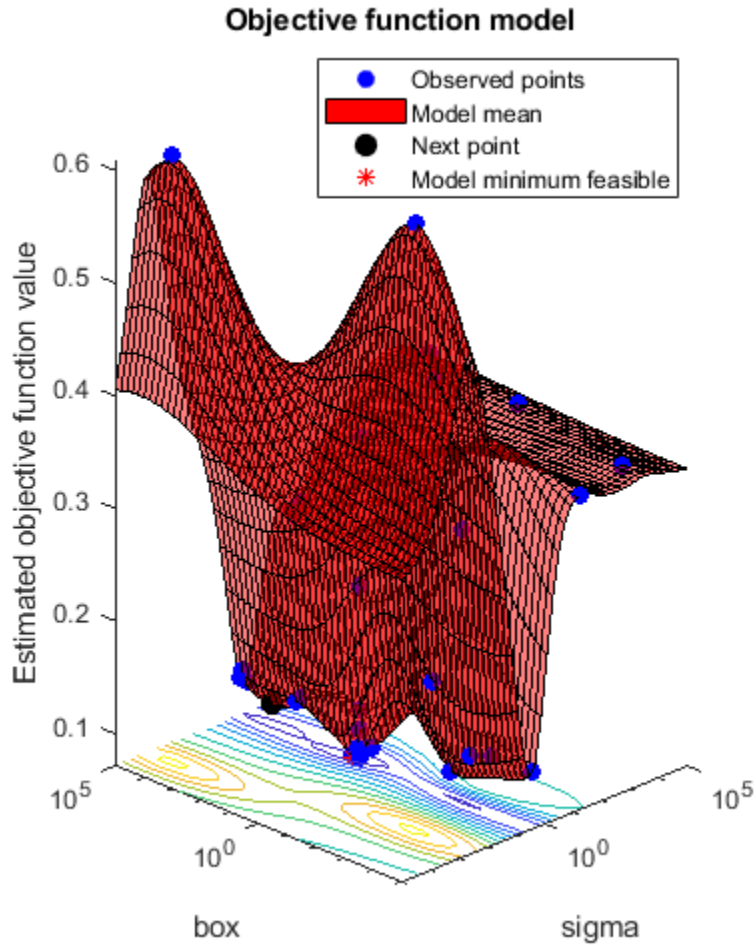
Create an optimized SVM model. For details of this model, see “Optimize a Cross-Validated SVM Classifier Using Bayesian Optimization” on page 10-53.

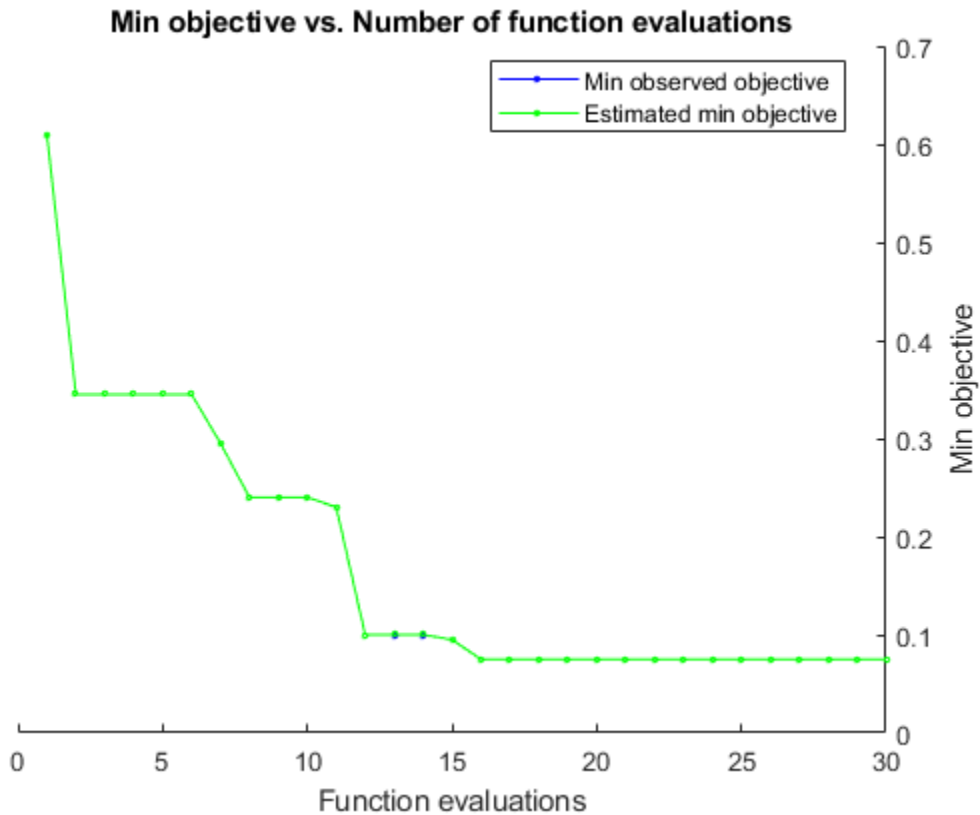
```
rng default
grpnpop = mvnrnd([1,0],eye(2),10);
redpop = mvnrnd([0,1],eye(2),10);
redpts = zeros(100,2);
grnpts = redpts;
for i = 1:100
    grnpts(i,:) = mvnrnd(grnpop(randi(10),:),eye(2)*0.02);
    redpts(i,:) = mvnrnd(redpop(randi(10),:),eye(2)*0.02);
end
cdata = [grnpts;redpts];
grp = ones(200,1);
grp(101:200) = -1;
```

```

c = cvpartition(200, 'KFold', 10);
sigma = optimizableVariable('sigma', [1e-5, 1e5], 'Transform', 'log');
box = optimizableVariable('box', [1e-5, 1e5], 'Transform', 'log');
minfn = @(z) kfoldLoss(fitcsvm(cdata, grp, 'CVPartition', c, ...
    'KernelFunction', 'rbf', 'BoxConstraint', z.box, ...
    'KernelScale', z.sigma));
results = bayesopt(minfn, [sigma, box], 'IsObjectiveDeterministic', true, ...
    'AcquisitionFunctionName', 'expected-improvement-plus', 'Verbose', 0);

```





Predict the evaluation time for various points.

```
sigma = logspace(-5,5,11)';
box = 1e5*ones(size(sigma));
XTable = table(sigma,box);
time = predictObjectiveEvaluationTime(results,XTable);
[XTable,table(time)]
```

ans =

11x3 table

sigma	box	time



1e-05	1e+05	0.5938
0.0001	1e+05	0.76254
0.001	1e+05	0.59292
0.01	1e+05	0.38223
0.1	1e+05	0.2931
1	1e+05	0.81908
10	1e+05	4.2514
100	1e+05	1.5006
1000	1e+05	0.29586
10000	1e+05	0.27402
1e+05	1e+05	0.23836

## Input Arguments

### **results** — Bayesian optimization results

BayesianOptimization object

Bayesian optimization results, specified as a BayesianOptimization object.

### **XTable** — Prediction points

table with D columns

Prediction points, specified as a table with D columns, where D is the number of variables in the problem. The function performs its predictions on these points.

Data Types: `table`

## Output Arguments

### **time** — Estimated objective evaluation times

N-by-1 vector

Estimated objective evaluation times, returned as an N-by-1 vector, where N is the number of rows of XTable. The estimated values are the means of the posterior distribution of the Gaussian process model of the evaluation times of the objective function.

## See Also

BayesianOptimization | bayesopt

**Introduced in R2016b**

# predictorImportance

**Class:** CompactClassificationEnsemble

Estimates of predictor importance

## Syntax

```
imp = predictorImportance(ens)
[imp,ma] = predictorImportance(ens)
```

## Description

`imp = predictorImportance(ens)` computes estimates of predictor importance for `ens` by summing these estimates over all weak learners in the ensemble. `imp` has one element for each input predictor in the data used to train this ensemble. A high value indicates that this predictor is important for `ens`.

`[imp,ma] = predictorImportance(ens)` returns a P-by-P matrix with predictive measures of association for P predictors, when the learners in `ens` contain surrogate splits. See “Definitions” on page 32-5144.

## Input Arguments

**ens**

A classification ensemble created by `fitcensemble`, or by the `compact` method.

## Output Arguments

### **imp**

A row vector with the same number of elements as the number of predictors (columns) in `ens.X`. The entries are the estimates of predictor importance, with 0 representing the smallest possible importance.

### **ma**

A  $P$ -by- $P$  matrix of predictive measures of association for  $P$  predictors. Element `ma(I,J)` is the predictive measure of association averaged over surrogate splits on predictor  $J$  for which predictor  $I$  is the optimal split predictor. `predictorImportance` averages this predictive measure of association over all trees in the ensemble.

## Examples

### Estimate Predictor Importance

Estimate the predictor importance for all variables in the Fisher iris data.

Load Fisher's iris data set.

```
load fisheriris
```

Grow an ensemble of 100 classification trees using `AdaBoostM2`.

```
ens = fitensemble(meas,species,'AdaBoostM2',100,'Tree');
```

Estimate the predictor importance for all predictor variables.

```
imp = predictorImportance(ens)
```

```
imp =
```

```
    0.0004    0.0016    0.1157    0.0434
```

The first two predictors are not very important in the ensemble.

### Predictor Importance and Surrogate Splits

Estimate the predictor importance for all variables in the Fisher iris data for an ensemble where the trees contain surrogate splits.

Load Fisher's iris data set.

```
load fisheriris
```

Grow an ensemble of 100 classification trees using AdaBoostM2. Specify to identify surrogate splits.

```
t = templateTree('Surrogate', 'on');  
ens = fitensemble(meas, species, 'AdaBoostM2', 100, t);
```

Estimate the predictor importance and predictive measures of association for all predictor variables.

```
[imp, ma] = predictorImportance(ens)
```

```
imp =  
  
    0.0674    0.0417    0.1582    0.1537
```

```
ma =  
  
    1.0000         0         0         0  
    0.0115    1.0000    0.0022    0.0054  
    0.2812    0.1752    1.0000    0.5220  
    0.0766    0.0458    0.2307    1.0000
```

The first two predictors show much more importance than the analysis in “Estimate Predictor Importance” on page 32-5142.

## Definitions

### Predictor Importance

`predictorImportance` computes estimates of predictor importance for `ens` by summing changes in the risk due to splits on every predictor and dividing the sum by the number of branch nodes. If `ens` is grown without surrogate splits, this sum is taken over best splits found at each branch node. If `ens` is grown with surrogate splits, this sum is taken over all splits at each branch node including surrogate splits. `imp` has one element for each input predictor in the data used to train `ens`. Predictor importance associated with this split is computed as the difference between the risk for the parent node and the total risk for the two children.

### Impurity and Node Error

`ClassificationTree` splits nodes based on either impurity or node error.

Impurity means one of several things, depending on your choice of the `SplitCriterion` name-value pair argument:

- Gini's Diversity Index (`gdi`) — The Gini index of a node is

$$1 - \sum_i p^2(i),$$

where the sum is over the classes  $i$  at the node, and  $p(i)$  is the observed fraction of classes with class  $i$  that reach the node. A node with just one class (a pure node) has Gini index 0; otherwise the Gini index is positive. So the Gini index is a measure of node impurity.

- Deviance ('deviance') — With  $p(i)$  defined the same as for the Gini index, the deviance of a node is

$$-\sum_i p(i) \log_2 p(i).$$

A pure node has deviance 0; otherwise, the deviance is positive.

- Twoing rule ('twoing') — Twoing is not a purity measure of a node, but is a different measure for deciding how to split a node. Let  $L(i)$  denote the fraction of members of class  $i$  in the left child node after a split, and  $R(i)$  denote the fraction of members of class  $i$  in the right child node after a split. Choose the split criterion to maximize

$$P(L)P(R) \left( \sum_i |L(i) - R(i)| \right)^2,$$

where  $P(L)$  and  $P(R)$  are the fractions of observations that split to the left and right respectively. If the expression is large, the split made each child node purer. Similarly, if the expression is small, the split made each child node similar to each other, and hence similar to the parent node, and so the split did not increase node purity.

- Node error — The node error is the fraction of misclassified classes at a node. If  $j$  is the class with the largest number of training samples at a node, the node error is

$$1 - p(j).$$

## Predictive Measure of Association

The predictive measure of association is a value that indicates the similarity between decision rules that split observations. Among all possible decision splits that are compared to the optimal split (found by growing the tree), the best surrogate decision split on page 32-2174 yields the maximum predictive measure of association. The second-best surrogate split has the second-largest predictive measure of association.

Suppose  $x_j$  and  $x_k$  are predictor variables  $j$  and  $k$ , respectively, and  $j \neq k$ . At node  $t$ , the predictive measure of association between the optimal split  $x_j < u$  and a surrogate split  $x_k < v$  is

$$\lambda_{jk} = \frac{\min(P_L, P_R) - (1 - P_{L_j L_k} - P_{R_j R_k})}{\min(P_L, P_R)}.$$

- $P_L$  is the proportion of observations in node  $t$ , such that  $x_j < u$ . The subscript  $L$  stands for the left child of node  $t$ .
- $P_R$  is the proportion of observations in node  $t$ , such that  $x_j \geq u$ . The subscript  $R$  stands for the right child of node  $t$ .

- $P_{L_j L_k}$  is the proportion of observations at node  $t$ , such that  $x_j < u$  and  $x_k < v$ .
- $P_{R_j R_k}$  is the proportion of observations at node  $t$ , such that  $x_j \geq u$  and  $x_k \geq v$ .
- Observations with missing values for  $x_j$  or  $x_k$  do not contribute to the proportion calculations.

$\lambda_{jk}$  is a value in  $(-\infty, 1]$ . If  $\lambda_{jk} > 0$ , then  $x_k < v$  is a worthwhile surrogate split for  $x_j < u$ .

## Algorithms

Element `ma(i, j)` is the predictive measure of association averaged over surrogate splits on predictor `j` for which predictor `i` is the optimal split predictor. This average is computed by summing positive values of the predictive measure of association over optimal splits on predictor `i` and surrogate splits on predictor `j` and dividing by the total number of optimal splits on predictor `i`, including splits for which the predictive measure of association between predictors `i` and `j` is negative.

## See Also

`predictorImportance` | `templateTree`



# predictorImportance

**Class:** CompactClassificationTree

Estimates of predictor importance

## Syntax

```
imp = predictorImportance(tree)
```

## Description

`imp = predictorImportance(tree)` computes estimates of predictor importance for `tree` by summing changes in the risk due to splits on every predictor and dividing the sum by the number of branch nodes.

## Input Arguments

**tree**

A classification tree created by `fitctree`, or by the `compact` method.

## Output Arguments

**imp**

A row vector with the same number of elements as the number of predictors (columns) in `tree.X`. The entries are the estimates of predictor importance, with 0 representing the smallest possible importance.

## Examples

### Estimate Predictor Importance Values

Load Fisher's iris data set.

```
load fisheriris
```

Grow a classification tree.

```
Mdl = fitctree(meas, species);
```

Compute predictor importance estimates for all predictor variables.

```
imp = predictorImportance(Mdl)

imp =

     0         0    0.0907    0.0682
```

The first two elements of `imp` are zero. Therefore, the first two predictors do not enter into `Mdl` calculations for classifying irises.

Estimates of predictor importance do not depend on the order of predictors if you use surrogate splits, but do depend on the order if you do not use surrogate splits.

Permute the order of the data columns in the previous example, grow another classification tree, and then compute predictor importance estimates.

```
measPerm = meas(:, [4 1 3 2]);
MdlPerm = fitctree(measPerm, species);
impPerm = predictorImportance(MdlPerm)
```

```
impPerm =

    0.1515         0    0.0074         0
```

The estimates of predictor importance are not a permutation of `imp`.

## Surrogate Splits and Predictor Importance

Load Fisher's iris data set.

```
load fisheriris
```

Grow a classification tree. Specify usage of surrogate splits.

```
Mdl = fitctree(meas, species, 'Surrogate', 'on');
```

Compute predictor importance estimates for all predictor variables.

```
imp = predictorImportance(Mdl)
```

```
imp =
    0.0791    0.0374    0.1530    0.1529
```

All predictors have some importance. The first two predictors are less important than the final two.

Permute the order of the data columns in the previous example, grow another classification tree specifying usage of surrogate splits, and then compute predictor importance estimates.

```
measPerm = meas(:, [4 1 3 2]);
MdlPerm = fitctree(measPerm, species, 'Surrogate', 'on');
impPerm = predictorImportance(MdlPerm)
```

```
impPerm =
    0.1529    0.0791    0.1530    0.0374
```

The estimates of predictor importance are a permutation of `imp`.

### Unbiased Predictor Importance Estimates

Load the `census1994` data set. Consider a model that predicts a person's salary category given their age, working class, education level, marital status, race, sex, capital gain and loss, and number of working hours per week.

```
load census1994
X = adultdata(:, {'age', 'workClass', 'education_num', 'marital_status', 'race', ...
                 'sex', 'capital_gain', 'capital_loss', 'hours_per_week', 'salary'});
```

Display the number of categories represented in the categorical variables using `summary`.

```
summary(X)
```

Variables:

```
age: 32561x1 double
```

```
Values:
```

Min	17
Median	37
Max	90

```
workClass: 32561x1 categorical
```

```
Values:
```

Federal-gov	960
Local-gov	2093
Never-worked	7
Private	22696
Self-emp-inc	1116
Self-emp-not-inc	2541
State-gov	1298
Without-pay	14
NumMissing	1836

```
education_num: 32561x1 double
```

```
Values:
```

Min	1
Median	10

Max 16

marital\_status: 32561x1 categorical

Values:

Divorced	4443
Married-AF-spouse	23
Married-civ-spouse	14976
Married-spouse-absent	418
Never-married	10683
Separated	1025
Widowed	993

race: 32561x1 categorical

Values:

Amer-Indian-Eskimo	311
Asian-Pac-Islander	1039
Black	3124
Other	271
White	27816

sex: 32561x1 categorical

Values:

Female	10771
Male	21790

capital\_gain: 32561x1 double

Values:

Min	0
Median	0
Max	99999

capital\_loss: 32561x1 double

Values:

Min	0
-----	---

```
        Median      0
        Max        4356

hours_per_week: 32561x1 double

Values:

        Min      1
        Median   40
        Max     99

salary: 32561x1 categorical

Values:

        <=50K    24720
        >50K     7841
```

Because there are few categories represented in the categorical variables compared to levels in the continuous variables, the standard CART, predictor-splitting algorithm prefers splitting a continuous predictor over the categorical variables.

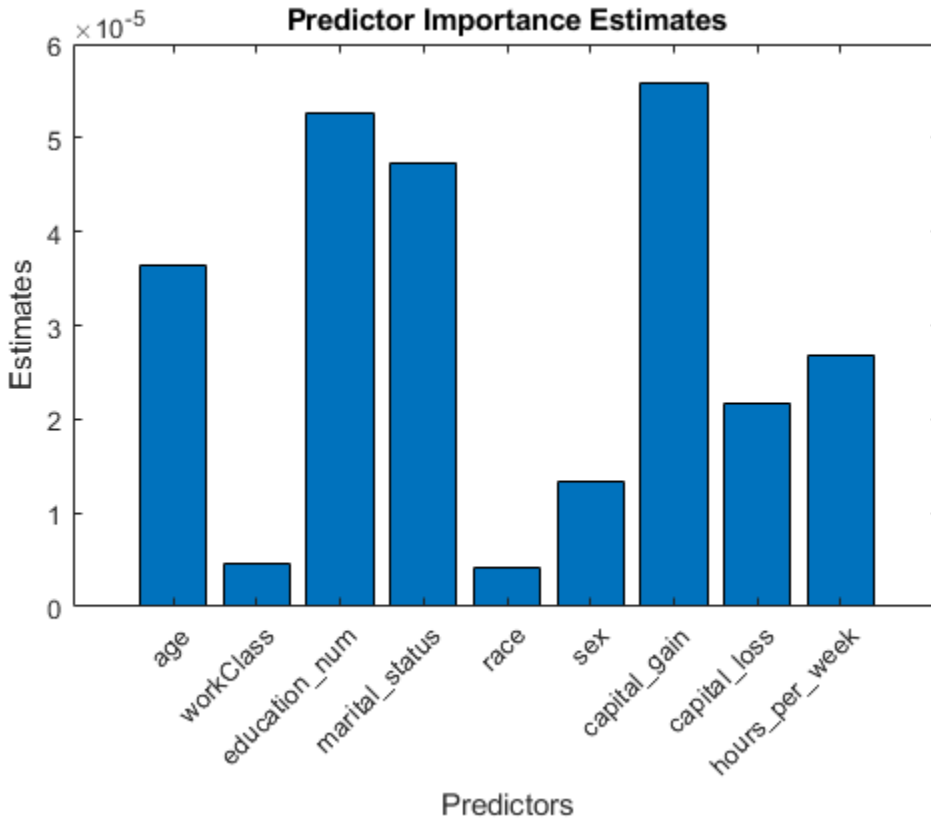
Train a classification tree using the entire data set. To grow unbiased trees, specify usage of the curvature test for splitting predictors. Because there are missing observations in the data, specify usage of surrogate splits.

```
Mdl = fitctree(X, 'salary', 'PredictorSelection', 'curvature', ...
    'Surrogate', 'on');
```

Estimate predictor importance values by summing changes in the risk due to splits on every predictor and dividing the sum by the number of branch nodes. Compare the estimates using a bar graph.

```
imp = predictorImportance(Mdl);

figure;
bar(imp);
title('Predictor Importance Estimates');
ylabel('Estimates');
xlabel('Predictors');
h = gca;
h.XTickLabel = Mdl.PredictorNames;
h.XTickLabelRotation = 45;
h.TickLabelInterpreter = 'none';
```



In this case, `capital_gain` is the most important predictor, followed by `education_num`.

## Definitions

### Predictor Importance

`predictorImportance` computes estimates of predictor importance for `tree` by summing changes in the risk due to splits on every predictor and dividing the sum by the number of branch nodes. If `tree` is grown without surrogate splits, this sum is taken over best splits found at each branch node. If `tree` is grown with surrogate splits, this

sum is taken over all splits at each branch node including surrogate splits. `imp` has one element for each input predictor in the data used to train `tree`. Predictor importance associated with this split is computed as the difference between the risk for the parent node and the total risk for the two children.

Estimates of predictor importance do not depend on the order of predictors if you use surrogate splits, but do depend on the order if you do not use surrogate splits.

If you use surrogate splits, `predictorImportance` computes estimates before the tree is reduced by pruning or merging leaves. If you do not use surrogate splits, `predictorImportance` computes estimates after the tree is reduced by pruning or merging leaves. Therefore, reducing the tree by pruning affects the predictor importance for a tree grown without surrogate splits, and does not affect the predictor importance for a tree grown with surrogate splits.

## Impurity and Node Error

`ClassificationTree` splits nodes based on either impurity or node error.

Impurity means one of several things, depending on your choice of the `SplitCriterion` name-value pair argument:

- Gini's Diversity Index (`gdi`) — The Gini index of a node is

$$1 - \sum_i p^2(i),$$

where the sum is over the classes  $i$  at the node, and  $p(i)$  is the observed fraction of classes with class  $i$  that reach the node. A node with just one class (a pure node) has Gini index 0; otherwise the Gini index is positive. So the Gini index is a measure of node impurity.

- Deviance (`'deviance'`) — With  $p(i)$  defined the same as for the Gini index, the deviance of a node is

$$-\sum_i p(i) \log_2 p(i).$$

A pure node has deviance 0; otherwise, the deviance is positive.

- Twoing rule (`'twoing'`) — Twoing is not a purity measure of a node, but is a different measure for deciding how to split a node. Let  $L(i)$  denote the fraction of



members of class  $i$  in the left child node after a split, and  $R(i)$  denote the fraction of members of class  $i$  in the right child node after a split. Choose the split criterion to maximize

$$P(L)P(R)\left(\sum_i |L(i) - R(i)|\right)^2,$$

where  $P(L)$  and  $P(R)$  are the fractions of observations that split to the left and right respectively. If the expression is large, the split made each child node purer. Similarly, if the expression is small, the split made each child node similar to each other, and hence similar to the parent node, and so the split did not increase node purity.

- Node error — The node error is the fraction of misclassified classes at a node. If  $j$  is the class with the largest number of training samples at a node, the node error is  $1 - p(j)$ .

## See Also

`fitcensemble` | `fitctree` | `oobPermutedPredictorImportance` | `predictorImportance`

## predictorImportance

**Class:** CompactRegressionEnsemble

Estimates of predictor importance

### Syntax

```
imp = predictorImportance(ens)
[imp,ma] = predictorImportance(ens)
```

### Description

`imp = predictorImportance(ens)` computes estimates of predictor importance for `ens` by summing these estimates over all weak learners in the ensemble. `imp` has one element for each input predictor in the data used to train this ensemble. A high value indicates that this predictor is important for `ens`.

`[imp,ma] = predictorImportance(ens)` returns a  $P$ -by- $P$  matrix with predictive measures of association for  $P$  predictors.

### Input Arguments

**ens**

A regression ensemble created by `fitensemble`, or by the `compact` method.

### Output Arguments

**imp**

A row vector with the same number of elements as the number of predictors (columns) in `ens.X`. The entries are the estimates of predictor importance, with 0 representing the smallest possible importance.

**ma**

A  $P$ -by- $P$  matrix of predictive measures of association for  $P$  predictors. Element  $ma(I, J)$  is the predictive measure of association averaged over surrogate splits on predictor  $J$  for which predictor  $I$  is the optimal split predictor. `predictorImportance` averages this predictive measure of association over all trees in the ensemble.

## Examples

### Estimate Predictor Importance

Estimate the predictor importance for all predictor variables in the data.

Load the `carsmall` data set.

```
load carsmall
```

Grow an ensemble of 100 regression trees for MPG using Acceleration, Cylinders, Displacement, Horsepower, Model\_Year, and Weight as predictors.

```
X = [Acceleration Cylinders Displacement Horsepower Model_Year Weight];
ens = fitensemble(X,MPG,'LSBoost',100,'Tree');
```

Estimate the predictor importance for all predictor variables.

```
imp = predictorImportance(ens)
```

```
imp =
```

```
    0.0150         0    0.0066    0.1111    0.0437    0.5181
```

Weight, the last predictor, has the most impact on mileage. The second predictor has importance 0, which means that the number of cylinders has no impact on predictions made with `ens`.

### Predictor Importance and Surrogate Splits

Estimate the predictor importance for all variables in the data and where the regression tree ensemble contains surrogate splits.

Load the `carsmall` data set.

```
load carsmall
```

Grow an ensemble of 100 regression trees for MPG using Acceleration, Cylinders, Displacement, Horsepower, Model\_Year, and Weight as predictors. Specify to identify surrogate splits.

```
X = [Acceleration Cylinders Displacement Horsepower Model_Year Weight];  
t = templateTree('Surrogate','on');  
ens = fitensemble(X,MPG,'LSBoost',100,t);
```

Estimate the predictor importance and predictive measures of association for all predictor variables.

```
[imp,ma] = predictorImportance(ens)
```

```
imp =
```

```
    0.2065    0.3799    0.4100    0.6190    0.3670    0.5476
```

```
ma =
```

```
    1.0000    0.0098    0.0102    0.0098    0.0033    0.0067  
         0    1.0000         0         0         0         0  
    0.0056    0.0084    1.0000    0.0078    0.0022    0.0084  
    0.2232    0.1574    0.2066    1.0000    0.0580    0.2059  
    0.0061    0.0070    0.0063    0.0064    1.0000    0.0056  
    0.0200    0.0368    0.0620    0.0521    0.0098    1.0000
```

Comparing `imp` to the results in “Estimate Predictor Importance” on page 32-5157, `Horsepower` has the greatest impact on mileage, with `Weight` having the second greatest impact.

## Definitions

### Predictor Importance

`predictorImportance` computes estimates of predictor importance for `tree` by summing changes in the mean squared error (MSE) due to splits on every predictor and dividing the sum by the number of branch nodes. If the tree is grown without surrogate splits, this sum is taken over best splits found at each branch node. If the tree is grown with surrogate splits, this sum is taken over all splits at each branch node including surrogate splits. `imp` has one element for each input predictor in the data used to train this tree. At each node, MSE is estimated as node error weighted by the node probability. Variable importance associated with this split is computed as the difference between MSE for the parent node and the total MSE for the two children.

### Predictive Measure of Association

The predictive measure of association is a value that indicates the similarity between decision rules that split observations. Among all possible decision splits that are compared to the optimal split (found by growing the tree), the best surrogate decision split on page 32-2174 yields the maximum predictive measure of association. The second-best surrogate split has the second-largest predictive measure of association.

Suppose  $x_j$  and  $x_k$  are predictor variables  $j$  and  $k$ , respectively, and  $j \neq k$ . At node  $t$ , the predictive measure of association between the optimal split  $x_j < u$  and a surrogate split  $x_k < v$  is

$$\lambda_{jk} = \frac{\min(P_L, P_R) - (1 - P_{L_j L_k} - P_{R_j R_k})}{\min(P_L, P_R)}.$$

- $P_L$  is the proportion of observations in node  $t$ , such that  $x_j < u$ . The subscript  $L$  stands for the left child of node  $t$ .
- $P_R$  is the proportion of observations in node  $t$ , such that  $x_j \geq u$ . The subscript  $R$  stands for the right child of node  $t$ .

- $P_{L_j L_k}$  is the proportion of observations at node  $t$ , such that  $x_j < u$  and  $x_k < v$ .
- $P_{R_j R_k}$  is the proportion of observations at node  $t$ , such that  $x_j \geq u$  and  $x_k \geq v$ .
- Observations with missing values for  $x_j$  or  $x_k$  do not contribute to the proportion calculations.

$\lambda_{jk}$  is a value in  $(-\infty, 1]$ . If  $\lambda_{jk} > 0$ , then  $x_k < v$  is a worthwhile surrogate split for  $x_j < u$ .

## Algorithms

Element `ma(i, j)` is the predictive measure of association averaged over surrogate splits on predictor `j` for which predictor `i` is the optimal split predictor. This average is computed by summing positive values of the predictive measure of association over optimal splits on predictor `i` and surrogate splits on predictor `j` and dividing by the total number of optimal splits on predictor `i`, including splits for which the predictive measure of association between predictors `i` and `j` is negative.

## See Also

`plotPartialDependence` | `predictorImportance` | `templateTree`

# predictorImportance

**Class:** CompactRegressionTree

Estimates of predictor importance

## Syntax

```
imp = predictorImportance(tree)
```

## Description

`imp = predictorImportance(tree)` computes estimates of predictor importance for `tree` by summing changes in the mean squared error due to splits on every predictor and dividing the sum by the number of branch nodes.

## Input Arguments

**tree**

A regression tree created by `fitrtree`, or by the `compact` method.

## Output Arguments

**imp**

A row vector with the same number of elements as the number of predictors (columns) in `tree.X`. The entries are the estimates of predictor importance, with 0 representing the smallest possible importance.

## Examples

### Estimate Predictor Importance

Estimate the predictor importance for all predictor variables in the data.

Load the `carsmall` data set.

```
load carsmall
```

Grow a regression tree for MPG using Acceleration, Cylinders, Displacement, Horsepower, Model\_Year, and Weight as predictors.

```
X = [Acceleration Cylinders Displacement Horsepower Model_Year Weight];  
tree = fitrtree(X,MPG);
```

Estimate the predictor importance for all predictor variables.

```
imp = predictorImportance(tree)
```

```
imp =
```

```
    0.0228    0.1068    0.1075    0.1829    0.3348    2.6645
```

Weight, the last predictor, has the most impact on mileage. The predictor with the minimal impact on making predictions is the first variable, which is Acceleration.

### Predictor Importance and Surrogate Splits

Estimate the predictor importance for all variables in the data and where the regression tree contains surrogate splits.

Load the `carsmall` data set.

```
load carsmall
```

Grow a regression tree for MPG using Acceleration, Cylinders, Displacement, Horsepower, Model\_Year, and Weight as predictors. Specify to identify surrogate splits.

```
X = [Acceleration Cylinders Displacement Horsepower Model_Year Weight];  
tree = fitrtree(X,MPG, 'Surrogate', 'on');
```



Estimate the predictor importance for all predictor variables.

```
imp = predictorImportance(tree)
```

```
imp =
```

```
1.0449    2.4560    2.5570    2.5788    2.0832    2.8938
```

Comparing `imp` to the results in “Estimate Predictor Importance” on page 32-5161, `Weight` still has the most impact on mileage, but `Cylinders` is the fourth most important predictor.

### Unbiased Predictor Importance Estimates

Load the `carsmall` data set. Consider a model that predicts the mean fuel economy of a car given its acceleration, number of cylinders, engine displacement, horsepower, manufacturer, model year, and weight. Consider `Cylinders`, `Mfg`, and `Model_Year` as categorical variables.

```
load carsmall
Cylinders = categorical(Cylinders);
Mfg = categorical(cellstr(Mfg));
Model_Year = categorical(Model_Year);
X = table(Acceleration,Cylinders,Displacement,Horsepower,Mfg,...
         Model_Year,Weight,MPG);
```

Display the number of categories represented in the categorical variables.

```
numCylinders = numel(categories(Cylinders))
numMfg = numel(categories(Mfg))
numModelYear = numel(categories(Model_Year))
```

```
numCylinders =
```

```
3
```

```
numMfg =
```

28

```
numModelYear =  
3
```

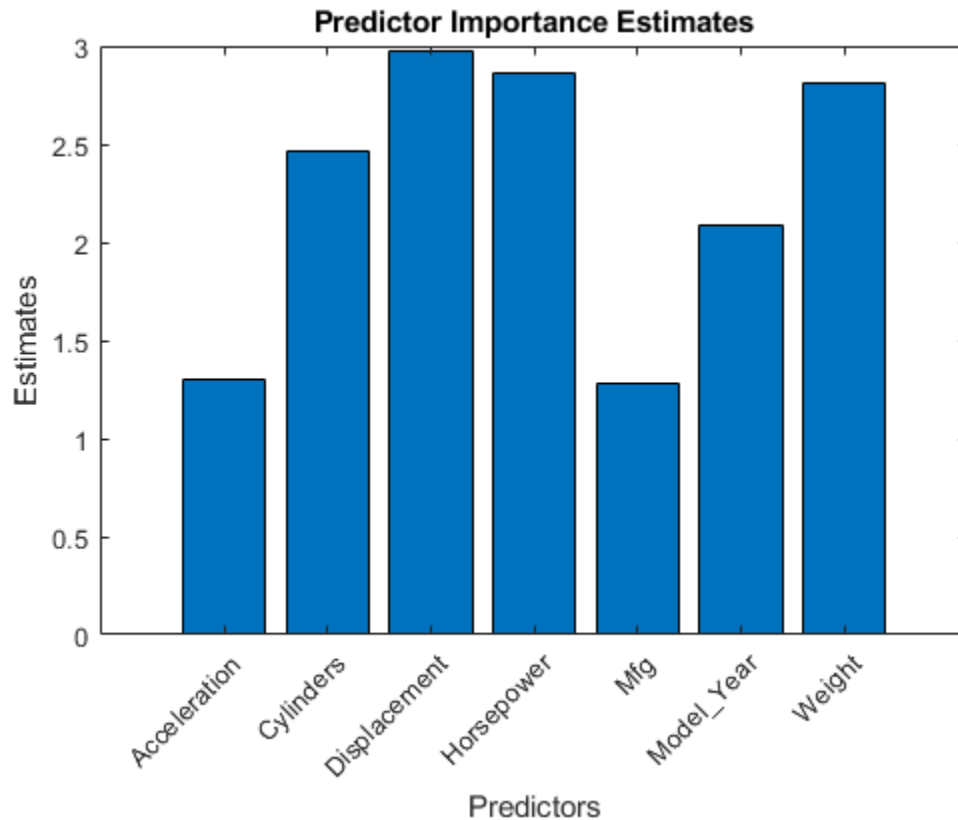
Because there are 3 categories only in `Cylinders` and `Model_Year`, the standard CART, predictor-splitting algorithm prefers splitting a continuous predictor over these two variables.

Train a regression tree using the entire data set. To grow unbiased trees, specify usage of the curvature test for splitting predictors. Because there are missing values in the data, specify usage of surrogate splits.

```
Mdl = fitrtree(X, 'MPG', 'PredictorSelection', 'curvature', 'Surrogate', 'on');
```

Estimate predictor importance values by summing changes in the risk due to splits on every predictor and dividing the sum by the number of branch nodes. Compare the estimates using a bar graph.

```
imp = predictorImportance(Mdl);  
  
figure;  
bar(imp);  
title('Predictor Importance Estimates');  
ylabel('Estimates');  
xlabel('Predictors');  
h = gca;  
h.XTickLabel = Mdl.PredictorNames;  
h.XTickLabelRotation = 45;  
h.TickLabelInterpreter = 'none';
```



In this case, `Displacement` is the most important predictor, followed by `Horsepower`.

## Definitions

### Predictor Importance

`predictorImportance` computes estimates of predictor importance for `tree` by summing changes in the mean squared error (MSE) due to splits on every predictor and dividing the sum by the number of branch nodes. If the tree is grown without surrogate splits, this sum is taken over best splits found at each branch node. If the tree is grown with surrogate splits, this sum is taken over all splits at each branch node including

surrogate splits. `imp` has one element for each input predictor in the data used to train this tree. At each node, MSE is estimated as node error weighted by the node probability. Variable importance associated with this split is computed as the difference between MSE for the parent node and the total MSE for the two children.

Estimates of predictor importance do not depend on the order of predictors if you use surrogate splits, but do depend on the order if you do not use surrogate splits.

If you use surrogate splits, `predictorImportance` computes estimates before the tree is reduced by pruning or merging leaves. If you do not use surrogate splits, `predictorImportance` computes estimates after the tree is reduced by pruning or merging leaves. Therefore, reducing the tree by pruning affects the predictor importance for a tree grown without surrogate splits, and does not affect the predictor importance for a tree grown with surrogate splits.

## See Also

`fitensemble` | `fitrtree` | `oobPermutedPredictorImportance` |  
`plotPartialDependence` | `predictorImportance`

# princomp

Principal component analysis (PCA) on data

---

**Note** `princomp` will be removed in a future release. Use `pca` instead.

---

## Syntax

```
[COEFF,SCORE] = princomp(X)
[COEFF,SCORE,latent] = princomp(X)
[COEFF,SCORE,latent,tsquare] = princomp(X)
[...] = princomp(X,'econ')
```

## Description

`COEFF = princomp(X)` performs principal components analysis (PCA) on the  $n$ -by- $p$  data matrix  $X$ , and returns the principal component coefficients, also known as loadings. Rows of  $X$  correspond to observations, columns to variables. `COEFF` is a  $p$ -by- $p$  matrix, each column containing coefficients for one principal component. The columns are in order of decreasing component variance.

`princomp` centers  $X$  by subtracting off column means, but does not rescale the columns of  $X$ . To perform principal components analysis with standardized variables, that is, based on correlations, use `princomp(zscore(X))`. To perform principal components analysis directly on a covariance or correlation matrix, use `pcacov`.

`[COEFF,SCORE] = princomp(X)` returns `SCORE`, the principal component scores; that is, the representation of  $X$  in the principal component space. Rows of `SCORE` correspond to observations, columns to components.

`[COEFF,SCORE,latent] = princomp(X)` returns `latent`, a vector containing the eigenvalues of the covariance matrix of  $X$ .

`[COEFF,SCORE,latent,tsquare] = princomp(X)` returns `tsquare`, which contains Hotelling's  $T^2$  statistic for each data point.

The scores are the data formed by transforming the original data into the space of the principal components. The values of the vector `latent` are the variance of the columns of `SCORE`. Hotelling's  $T^2$  is a measure of the multivariate distance of each observation from the center of the data set.

When  $n \leq p$ , `SCORE(:,n:p)` and `latent(n:p)` are necessarily zero, and the columns of `COEFF(:,n:p)` define directions that are orthogonal to  $X$ .

`[...] = princomp(X, 'econ')` returns only the elements of `latent` that are not necessarily zero, and the corresponding columns of `COEFF` and `SCORE`, that is, when  $n \leq p$ , only the first  $n-1$ . This can be significantly faster when  $p$  is much larger than  $n$ .

## Examples

Compute principal components for the `ingredients` data in the `Hald` data set, and the variance accounted for by each component.

```
load hald;
[pc,score,latent,tsquare] = princomp(ingredients);
pc,latent
```

```
pc =
    -0.0678    -0.6460     0.5673     0.5062
    -0.6785    -0.0200    -0.5440     0.4933
     0.0290     0.7553     0.4036     0.5156
     0.7309    -0.1085    -0.4684     0.4844
```

```
latent =
    517.7969
     67.4964
     12.4054
      0.2372
```

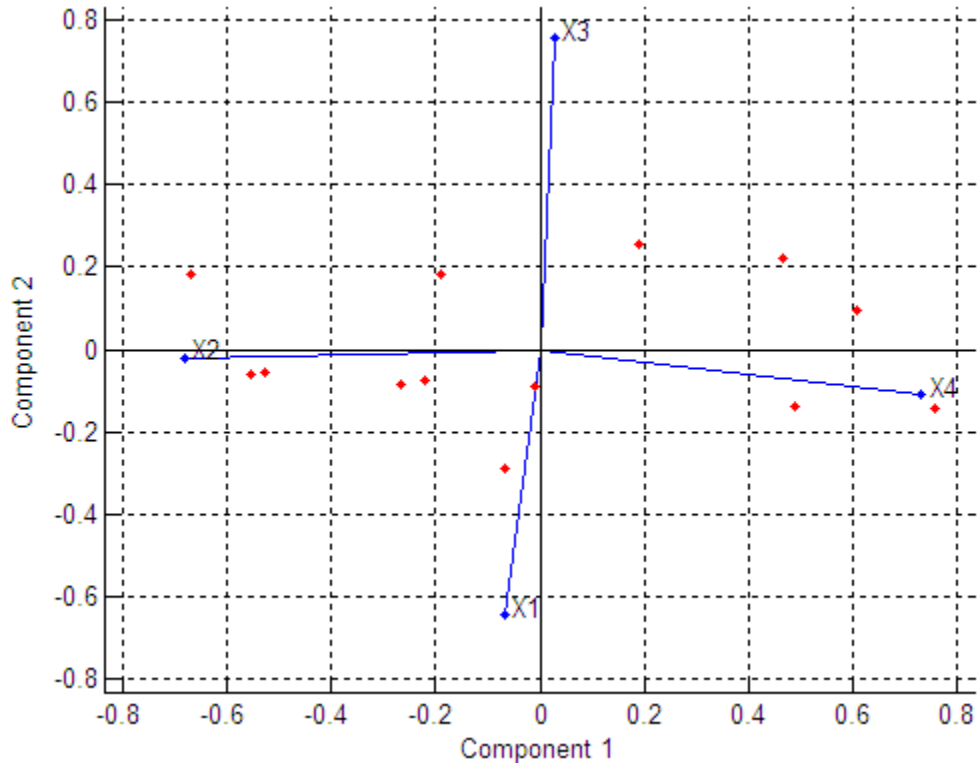
The following command and plot show that two components account for 98% of the variance:

```
cumsum(latent)./sum(latent)
ans =
```

```

0.86597
0.97886
0.9996
1
biplot(pc(:,1:2), 'Scores', score(:,1:2), 'VarLabels', ...
        {'X1' 'X2' 'X3' 'X4'})

```



For a more detailed example and explanation of this analysis method, see “Principal Component Analysis (PCA)” on page 15-83.

## References

- [1] Jackson, J. E., *A User's Guide to Principal Components*, John Wiley and Sons, 1991, p. 592.
- [2] Jolliffe, I. T., *Principal Component Analysis*, 2nd edition, Springer, 2002.

[3] Krzanowski, W. J. *Principles of Multivariate Analysis: A User's Perspective*. New York: Oxford University Press, 1988.

[4] Seber, G. A. F., *Multivariate Observations*, Wiley, 1984.

## See Also

`barttest` | `biplot` | `canoncorr` | `factoran` | `pca` | `pcacov` | `pcares` | `rotatefactors`

## Topics

“Principal Component Analysis (PCA)” on page 15-83

**Introduced before R2006a**



# Prior property

**Class:** TreeBagger

Prior class probabilities

## Description

The `Prior` property is a vector with prior probabilities for classes. This property is empty for ensembles of regression trees.

## See Also

`ClassificationTree` | `TreeBagger` | `fitctree`

## ProbDist class

Object representing probability distribution

---

**Note** `ProbDist` will be removed in a future release. To create and fit probability distribution objects, use `makedist` and `fitdist` instead.

---

### Description

`ProbDist` is an abstract class representing a probability distribution.

### Construction

`ProbDist` is an abstract class. You cannot create instances of this class directly. You can construct an object in a subclass, such as `ProbDistUnivParam` or `ProbDistUnivKernel`, by calling the subclass constructors (`ProbDistUnivParam` or `ProbDistUnivKernel`).

### Methods

<code>cdf</code>	Return cumulative distribution function (CDF) for <code>ProbDist</code> object
<code>pdf</code>	Return probability density function (PDF) for <code>ProbDist</code> object
<code>random</code>	Generate random number drawn from <code>ProbDist</code> object

## Properties

`DistName` Read-only probability distribution name of ProbDist object

`InputData` Read-only structure containing information about input data to ProbDist object

`Support` Read-only structure containing information about support of ProbDist object

## Copy Semantics

Value. To learn how this affects your use of the class, see [Copying Objects \(MATLAB\)](#) in the [MATLAB Programming Fundamentals](#) documentation.

## See Also

[ProbDistKernel](#) | [ProbDistParametric](#) | [ProbDistUnivKernel](#) |  
[ProbDistUnivKernel](#) | [ProbDistUnivParam](#) | [ProbDistUnivParam](#)

## ProbDistKernel class

### Superclasses:

Object representing nonparametric probability distribution defined by kernel smoothing

---

**Note** `ProbDistKernel` will be removed in a future release. To create and fit probability distribution objects, use `makedist` and `fitdist` instead.

---

## Description

`ProbDistKernel` is an abstract class defining the properties and methods of a nonparametric distribution defined by a kernel smoothing function.

## Construction

`ProbDistKernel` is an abstract class. You cannot create instances of this class directly. You can construct an object in a subclass, `ProbDistUnivKernel` by calling the subclass constructor, `ProbDistUnivKernel`.

## Methods

<code>cdf</code>	Return cumulative distribution function (CDF) for <code>ProbDist</code> object
<code>pdf</code>	Return probability density function (PDF) for <code>ProbDist</code> object
<code>random</code>	Generate random number drawn from <code>ProbDist</code> object

---

**Note** The above methods are inherited from the `ProbDist` class.

---

## Properties

BandWidth	Read-only value specifying bandwidth of kernel smoothing function for ProbDistKernel object
DistName	Read-only probability distribution name of ProbDist object
InputData	Read-only structure containing information about input data to ProbDist object
Kernel	Read-only name of kernel smoothing function for ProbDistKernel object
Support	Read-only structure containing information about support of ProbDist object

---

**Note** Some of the above properties are inherited from the `ProbDist` class.

---

## Copy Semantics

Value. To learn how this affects your use of the class, see [Copying Objects \(MATLAB\)](#) in the [MATLAB Programming Fundamentals](#) documentation.

## See Also

`ProbDist` | `ProbDistUnivKernel` | `ProbDistUnivKernel`

## ProbDistParametric class

### Superclasses:

Object representing parametric probability distribution

---

**Note** `ProbDistParametric` will be removed in a future release. To create and fit probability distribution objects, use `makedist` and `fitdist` instead.

---

## Description

`ProbDistParametric` is an abstract class defining the properties and methods of a parametric probability distribution.

## Construction

`ProbDistParametric` is an abstract class. You cannot create instances of this class directly. You can construct an object in its subclass, `ProbDistUnivParam`, by calling the subclass constructor, `ProbDistUnivParam`.

## Methods

`cdf`        Return cumulative distribution function (CDF) for `ProbDist` object  
`pdf`        Return probability density function (PDF) for `ProbDist` object  
`random`    Generate random number drawn from `ProbDist` object

---

**Note** The above methods are inherited from the `ProbDist` class.

---

## Properties

DistName	Read-only probability distribution name of ProbDist object
InputData	Read-only structure containing information about input data to ProbDist object
NLogL	Read-only value specifying negative log likelihood for input data to ProbDistParametric object
NumParams	Read-only value specifying number of parameters of ProbDistParametric object
ParamCov	Read-only covariance matrix of parameter estimates of ProbDistParametric object
ParamDescription	Read-only cell array specifying descriptions of parameters of ProbDistParametric object
ParamIsFixed	Read-only logical array specifying fixed parameters of ProbDistParametric object
ParamNames	Read-only cell array specifying names of parameters of ProbDistParametric object
Params	Read-only array specifying values of parameters of ProbDistParametric object
Support	Read-only structure containing information about support of ProbDist object

---

**Note** Some of the above properties are inherited from the ProbDist class.

---

## Copy Semantics

Value. To learn how this affects your use of the class, see Copying Objects (MATLAB) in the MATLAB Programming Fundamentals documentation.

## See Also

ProbDist | ProbDistUnivParam | ProbDistUnivParam

## ProbDistUnivKernel class

### Superclasses:

Object representing univariate kernel probability distribution

---

**Note** `ProbDistUnivKernel` will be removed in a future release. To create and fit probability distribution objects, use `makedist` and `fitdist` instead.

---

## Description

A `ProbDistUnivKernel` object represents a univariate nonparametric probability distribution defined by kernel smoothing. You create this object using the `ProbDistUnivKernel` function to fit the distribution to data.

## Construction

## Methods

<code>cdf</code>	Return cumulative distribution function (CDF) for <code>ProbDist</code> object
<code>icdf</code>	Return inverse cumulative distribution function (ICDF) for <code>ProbDistUnivKernel</code> object
<code>iqr</code>	Return interquartile range (IQR) for <code>ProbDistUnivKernel</code> object
<code>median</code>	Return median of <code>ProbDistUnivKernel</code> object
<code>pdf</code>	Return probability density function (PDF) for <code>ProbDist</code> object
<code>random</code>	Generate random number drawn from <code>ProbDist</code> object

---

**Note** Some of the above methods are inherited from the `ProbDistKernel` class.

---



## Properties

BandWidth	Read-only value specifying bandwidth of kernel smoothing function for ProbDistKernel object
DistName	Read-only probability distribution name of ProbDist object
InputData	Read-only structure containing information about input data to ProbDist object
Kernel	Read-only name of kernel smoothing function for ProbDistKernel object
NLogL	Read-only value specifying negative log likelihood for input data to ProbDistUnivKernel object
Support	Read-only structure containing information about support of ProbDist object

---

**Note** Some of the above properties are inherited from the ProbDistKernel class.

---

## Copy Semantics

Value. To learn how this affects your use of the class, see Copying Objects (MATLAB) in the MATLAB Programming Fundamentals documentation.

## References

- [1] Bowman, A. W., and A. Azzalini. *Applied Smoothing Techniques for Data Analysis*. New York: Oxford University Press, 1997.

## See Also

ProbDist | ProbDistKernel | ProbDistUnivKernel | fitdist | ksdensity

## ProbDistUnivKernel

**Class:** ProbDistUnivKernel

Construct ProbDistUnivKernel object

### Syntax

*PD* = ProbDistUnivKernel(*X*)

*PD* = ProbDistUnivKernel(*X*, *param1*, *val1*, *param2*, *val2*, ...)

### Description

*PD* = ProbDistUnivKernel(*X*) creates *PD*, a ProbDistUnivKernel object, which represents a nonparametric probability distribution, based on a normal kernel smoothing function.

*PD* = ProbDistUnivKernel(*X*, *param1*, *val1*, *param2*, *val2*, ...) specifies optional parameter name/value pairs, as described in the Parameter/Values table. Parameter and value names are case insensitive.

### Compatibility

ProbDistUnivKernel will be removed in a future release. To create and fit probability distribution objects, use `makedist` and `fitdist` instead.

### Input Arguments

*X*                                      A column vector of data.

---

**Note** Any NaN values in *X* are ignored by the fitting calculations.

---

Parameter	Values
'censoring'	<p>A Boolean vector the same size as <math>X</math>, containing 1s when the corresponding elements in <math>X</math> are right-censored observations and 0s when the corresponding elements are exact observations. Default is a vector of 0s.</p> <hr/> <p><b>Note</b> Any NaN values in this censoring vector are ignored by the fitting calculations.</p>
'frequency'	<p>A vector the same size as <math>X</math>, containing nonnegative integers specifying the frequencies for the corresponding elements in <math>X</math>. Default is a vector of 1s.</p> <hr/> <p><b>Note</b> Any NaN values in this frequency vector are ignored by the fitting calculations.</p>
'kernel'	<p>A character vector specifying the type of kernel smoother to use. Choices are:</p> <ul style="list-style-type: none"> <li>• 'normal' (default)</li> <li>• 'box'</li> <li>• 'triangle'</li> <li>• 'epanechnikov'</li> </ul>
'support'	<p>Any of the following to specify the support:</p> <ul style="list-style-type: none"> <li>• 'unbounded' — Default. If the density can extend over the whole real line.</li> <li>• 'positive' — To restrict it to positive values.</li> <li>• A two-element vector giving finite lower and upper limits for the support of the density.</li> </ul>
'width'	<p>A value specifying the bandwidth of the kernel smoothing window. The default is optimal for estimating normal densities, but you may want to choose a smaller value to reveal features such as multiple modes.</p>

## Output Arguments

*PD* An object in the `ProbDistUnivKernel` class, which is derived from the `ProbDist` class. It represents a nonparametric probability distribution.

## References

- [1] Bowman, A. W., and A. Azzalini. *Applied Smoothing Techniques for Data Analysis*. New York: Oxford University Press, 1997.

## See Also

`fitdist` | `ksdensity`

Introduced in R2009a

# ProbDistUnivParam class

## Superclasses:

Object representing univariate parametric probability distribution

---

**Note** `ProbDistUnivParam` will be removed in a future release. To create and fit probability distribution objects, use `makedist` and `fitdist` instead.

---

## Description

A `ProbDistUnivParam` object represents a univariate parametric probability distribution. You create this object by using the constructor (`ProbDistUnivParam`) and supplying parameter values.

## Construction

`ProbDistUnivParam`                      Construct `ProbDistUnivParam` object

## Methods

<code>cdf</code>	Return cumulative distribution function (CDF) for <code>ProbDist</code> object
<code>icdf</code>	Return inverse cumulative distribution function (ICDF) for <code>ProbDistUnivParam</code> object
<code>iqr</code>	Return interquartile range (IQR) for <code>ProbDistUnivParam</code> object
<code>mean</code>	Return mean of <code>ProbDistUnivParam</code> object
<code>median</code>	Return median of <code>ProbDistUnivParam</code> object
<code>paramci</code>	Return parameter confidence intervals of <code>ProbDistUnivParam</code> object
<code>pdf</code>	Return probability density function (PDF) for <code>ProbDist</code> object
<code>random</code>	Generate random number drawn from <code>ProbDist</code> object
<code>std</code>	Return standard deviation of <code>ProbDistUnivParam</code> object
<code>var</code>	Return variance of <code>ProbDistUnivParam</code> object

---

**Note** Some of the above methods are inherited from the `ProbDistParametric` class.

---

## Properties

DistName	Read-only probability distribution name of ProbDist object
InputData	Read-only structure containing information about input data to ProbDist object
NLogL	Read-only value specifying negative log likelihood for input data to ProbDistParametric object
NumParams	Read-only value specifying number of parameters of ProbDistParametric object
ParamCov	Read-only covariance matrix of parameter estimates of ProbDistParametric object
ParamDescription	Read-only cell array specifying descriptions of parameters of ProbDistParametric object
ParamIsFixed	Read-only logical array specifying fixed parameters of ProbDistParametric object
ParamNames	Read-only cell array specifying names of parameters of ProbDistParametric object
Params	Read-only array specifying values of parameters of ProbDistParametric object
Support	Read-only structure containing information about support of ProbDist object

---

**Note** The above properties are inherited from the ProbDistParametric class.

---

---

**Note** Parameter values are also properties. For example, if you create PD, a univariate parametric probability distribution object that represents a normal distribution, then PD.mu and PD.sigma are properties that give the values of the mu and sigma parameters.

---

## Copy Semantics

Value. To learn how this affects your use of the class, see Copying Objects (MATLAB) in the MATLAB Programming Fundamentals documentation.

## References

- [1] Johnson, N. L., S. Kotz, and N. Balakrishnan. *Continuous Univariate Distributions*. Vol. 1, Hoboken, NJ: Wiley-Interscience, 1993.
- [2] Johnson, N. L., S. Kotz, and N. Balakrishnan. *Continuous Univariate Distributions*. Vol. 2, Hoboken, NJ: Wiley-Interscience, 1994.

## See Also

[ProbDist](#) | [ProbDistParametric](#) | [ProbDistUnivParam](#)

## Topics

[Appendix B](#)



# ProbDistUnivParam

**Class:** ProbDistUnivParam

Construct ProbDistUnivParam object

## Syntax

```
PD = ProbDistUnivParam(DistName, Params)
```

## Description

*PD* = ProbDistUnivParam(*DistName*, *Params*) creates *PD*, a ProbDistUnivParam object, which represents a probability distribution. This distribution is defined by the parametric distribution specified by *DistName*, with parameters specified by the numeric vector *Params*.

## Compatibility

ProbDistUnivParam will be removed in a future release. To create and fit probability distribution objects, use `makedist` and `fitdist` instead.

## Input Arguments

*DistName*

A character vector specifying a distribution. Choices are:

- 'beta'
- 'binomial'
- 'birnbaumsaunders'
- 'exponential'
- 'extreme value' or 'ev'
- 'gamma'
- 'generalized extreme value' or 'gev'
- 'generalized pareto' or 'gp'
- 'inversegaussian'
- 'logistic'
- 'loglogistic'
- 'lognormal'
- 'nakagami'
- 'negative binomial' or 'nbin'
- 'normal'
- 'poisson'
- 'rayleigh'
- 'rician'
- 'tlocationscale'
- 'weibull' or 'wbl'

For more information on these parametric distributions, see Appendix B.

*Params*

Numeric vector of distribution parameters. The number and type of parameters depends on the distribution you specify with *DistName*. For information on parameters for each distribution type, see Appendix B.

## Output Arguments

*PD* An object in the `ProbDistUnivParam` class, which is derived from the `ProbDist` class. It represents a parametric probability distribution.

## Examples

- 1 Create an object representing a normal distribution with a mean of 100 and a standard deviation of 10.

```
pd = ProbDistUnivParam('normal',[100 10])

pd =

normal distribution

    mu = 100
    sigma = 10
```

- 2 Generate a 4-by-5 matrix of random values from this distribution.

```
random(pd,4,5)

ans =

    105.3767    103.1877    135.7840    107.2540     98.7586
    118.3389     86.9231    127.6944     99.3695    114.8970
     77.4115     95.6641     86.5011    107.1474    114.0903
    108.6217    103.4262    130.3492     97.9503    114.1719
```

## References

- [1] Johnson, N. L., S. Kotz, and N. Balakrishnan. *Continuous Univariate Distributions*. Vol. 1, Hoboken, NJ: Wiley-Interscience, 1993.
- [2] Johnson, N. L., S. Kotz, and N. Balakrishnan. *Continuous Univariate Distributions*. Vol. 2, Hoboken, NJ: Wiley-Interscience, 1994.

## See Also

`fitdist`

## Topics

Appendix B

Introduced in R2009a

## ProbabilityTolerance property

**Class:** gmdistribution

Tolerance for posterior probabilities

### Description

Value provided for the 'ProbabilityTolerance' name-value pair argument in the call to `fitgmdist`.

## probplot

Probability plots

### Syntax

```
probplot(y)
probplot(dist, y)

probplot(y, cens)
probplot(y, cens, freq)
probplot(dist, ___ )

probplot(ax, y)
probplot(ax, pd)
probplot(ax, fun, params)

probplot( ___, 'noref' )

h = probplot( ___ )
```

### Description

`probplot(y)` creates a normal probability plot comparing the distribution of the data in `y` to the normal distribution. The plot includes a reference line useful for judging whether the data follows a normal distribution.

`probplot(dist, y)` creates a probability plot for the distribution specified by `dist`, using the sample data in `y`.

`probplot(y, cens)` creates a probability plot using the censoring data in `cens`.

`probplot(y, cens, freq)` creates a probability plot using the censoring data in `cens` and the frequency data in `freq`.

`probplot(dist, ___ )` creates a probability plot for the distribution specified by `dist`, using any of the previous syntaxes.

`probplot(ax, y)` plots additional lines for the sample data in `y` to the probability plot specified by the axis handle `ax`.

`probplot(ax, pd)` plots a fitted line on the axes specified by `ax` to represent the probability distribution specified by `pd`.

`probplot(ax, fun, params)` plots a fitted line on the axes specified by `ax` to represent the function specified by `fun` with the parameters specified by `params`.

`probplot(____, 'noref')` omits the reference line from the plot.

`h = probplot(____)` returns graphics handles corresponding to the plotted lines.

## Examples

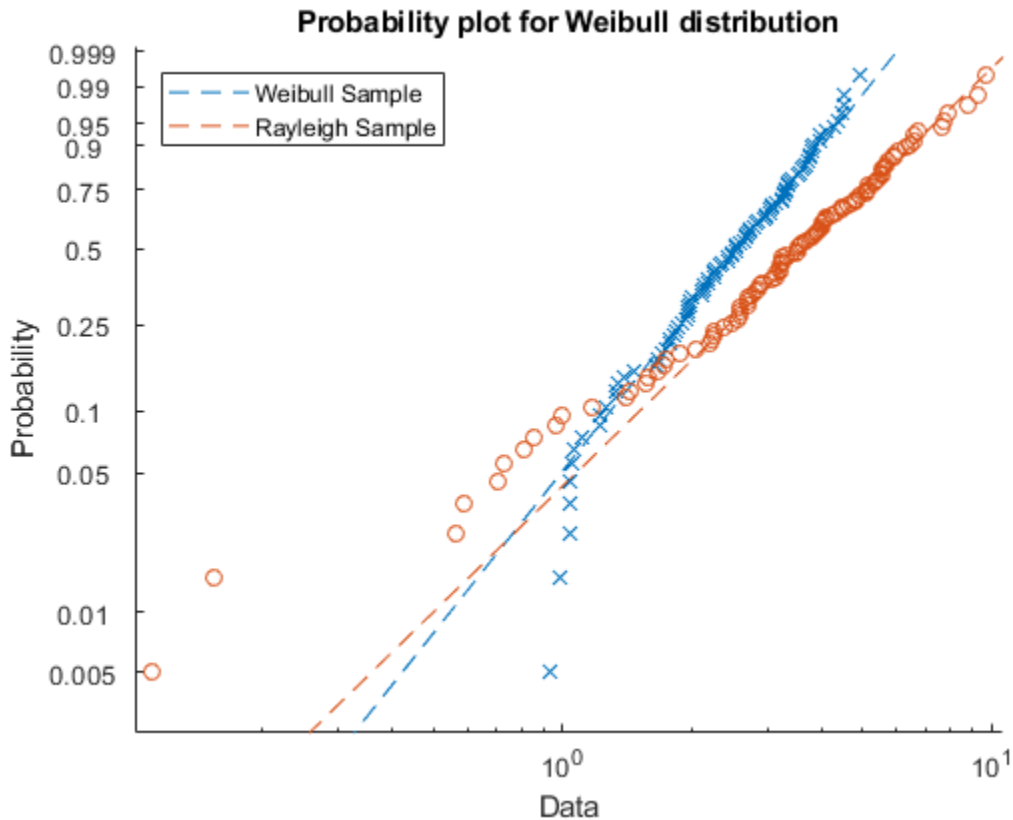
### Test Data for Weibull Distribution Using `probplot`

Generate sample data. The sample `x1` contains 100 random numbers from a Weibull distribution with scale parameter  $A = 3$  and shape parameter  $B = 3$ . The sample `x2` contains 100 random numbers from a Rayleigh distribution with scale parameter  $B = 3$ .

```
rng('default'); % For reproducibility
x1 = wblrnd(3,3,100,1);
x2 = raylrnd(3,100,1);
```

Create a probability plot to assess whether the data in `x1` and `x2` comes from a Weibull distribution.

```
figure;
probplot('weibull', [x1 x2])
legend('Weibull Sample', 'Rayleigh Sample', 'Location', 'NW')
```



The probability plot shows that the data in `x1` comes from a Weibull distribution, while the data in `x2` does not.

### Test Data for Normal Distribution Using `probpplot`

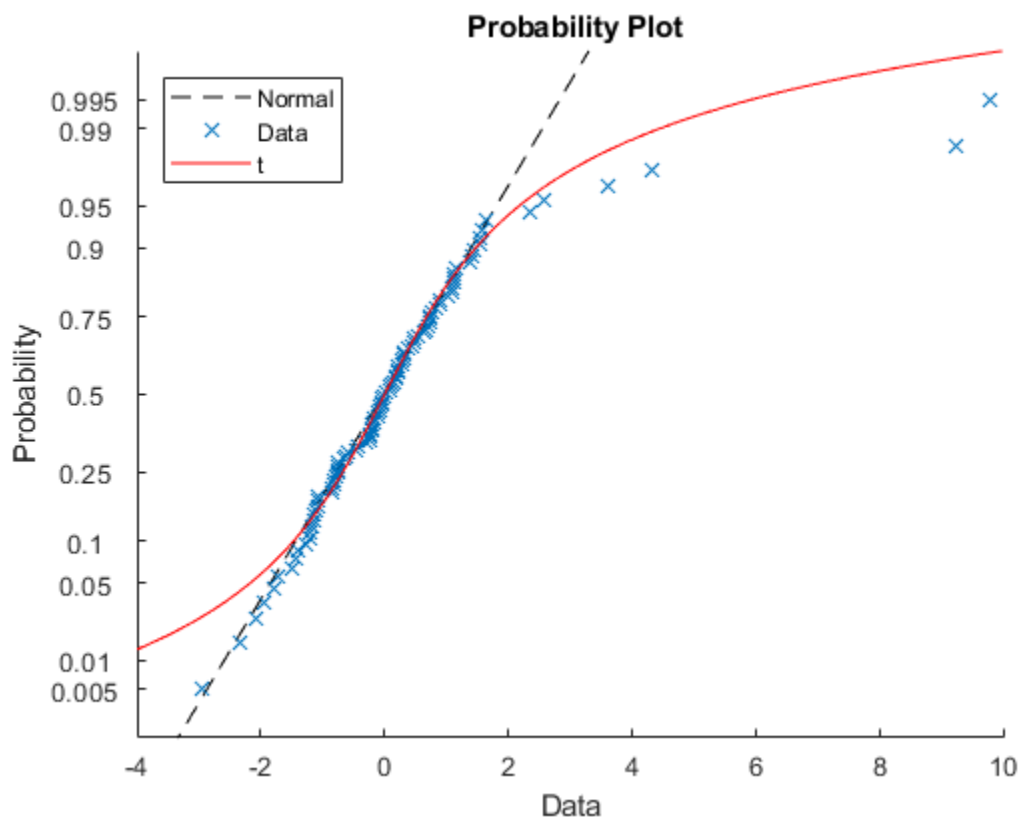
Generate sample data containing about 20% outliers in the tails. The left tail of the sample data contains 10 values randomly generated from an exponential distribution with parameter  $\mu = 1$ . The right tail contains 10 values randomly generated from an exponential distribution with parameter  $\mu = 5$ . The center of the sample data contains 80 values randomly generated from a standard normal distribution.



```
rng default % For reproducibility
left_tail = -exprnd(1,10,1);
right_tail = exprnd(5,10,1);
center = randn(80,1);
data = [left_tail;center;right_tail];
```

Create a probability plot to assess whether the sample data comes from a normal distribution. Plot a  $t$  location-scale curve on the same figure to compare with data.

```
figure;
probplot(data);
p = mle(data,'dist','tlo');
t = @(data,mu,sig,df)cdf('tlocationscale',data,mu,sig,df);
h = probplot(gca,t,p);
h.Color = 'r';
h.LineStyle = '-';
title('\bf Probability Plot')
legend('Normal','Data','t','Location','NW')
```



The plot shows that neither the normal line nor the  $t$  location-scale curve fits the tails very well because of the outliers.

### Identify Significant Effects with Half-Normal Probability Plot

Create a half-normal probability distribution plot to identify significant effects in an experiment to study factors that might influence flow rate in a chemical manufacturing process. The four factors are reactants A, B, C, and D. Each factor is present at two levels (high and low concentration). The experiment contains only one replication at each factor level.

Load the sample data.

```
load(fullfile(matlabroot, 'examples', 'stats', 'flowrate.mat'));
```

The first four columns of the table `flowrate` contain the design matrix for the factors and their interactions. The design matrix is coded to use 1 for the high factor level and -1 for the low factor level. The fifth column of `flowrate` contains the measured flow rate.

Fit a linear regression model using `rate` as the response variable. Use predictor variables A, B, C, D, and all of their interaction terms.

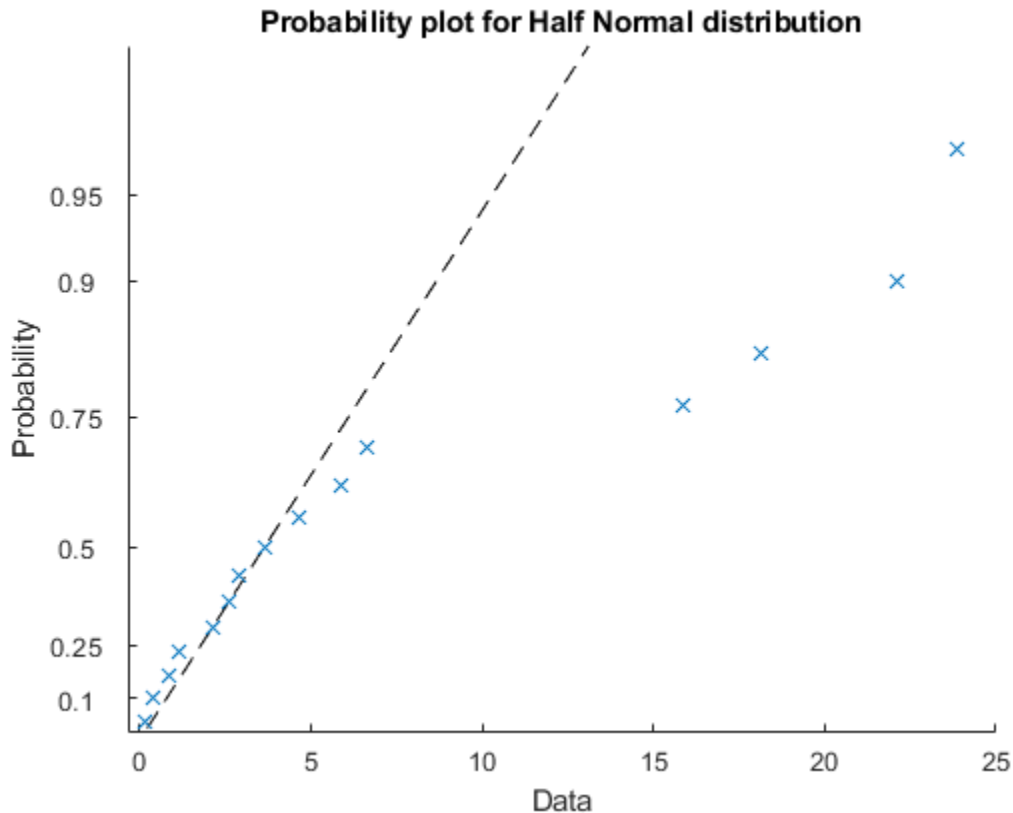
```
mdl = fitlm(flowrate, 'rate ~ A*B*C*D');
```

Calculate and store the absolute value of the factor effect estimates. To obtain the factor effect estimates, multiply the coefficient estimates obtained during the model fitting by two. This step is necessary because the regression coefficients measure the effect of a one-unit change in  $x$  on the mean of  $y$ . However, the effects estimates measure a two-unit change in  $x$  due to the design matrix coding of -1 and 1. Exclude the baseline measurement. Note that the factor order in `mdl` may be different from the order in the original design matrix.

```
effects = abs(mdl.Coefficients{2:end,1})*2);
```

Create a half-normal probability plot using the absolute value of the effects estimates, excluding the baseline.

```
figure  
h = probplot('halfnormal', effects);
```

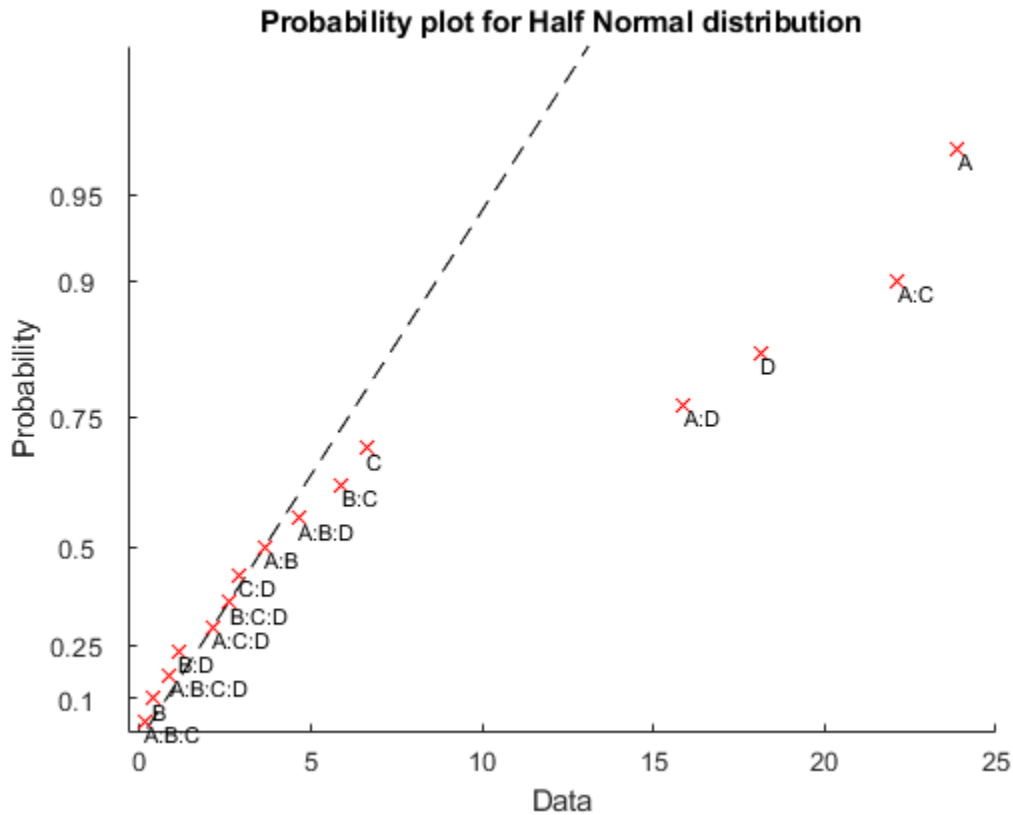


Label the points and format the plot. First, return the index values for the sorted effects estimates (from lowest to highest). Then use these index values to sort the probability values stored in the graphics handle (`h(1).YData`).

```
[b,i] = sort(effects);
prob(i) = h(1).YData;
```

Add text labels to the plot at each point. For each point, the x-value is the effects estimate and the y-value is the corresponding probability.

```
text(effects,prob,mdl.CoefficientNames(2:end),'FontSize',8,...
     'VerticalAlignment','top')
h(1).Color = 'r';
```



The points located far from the reference line represent the significant effects.

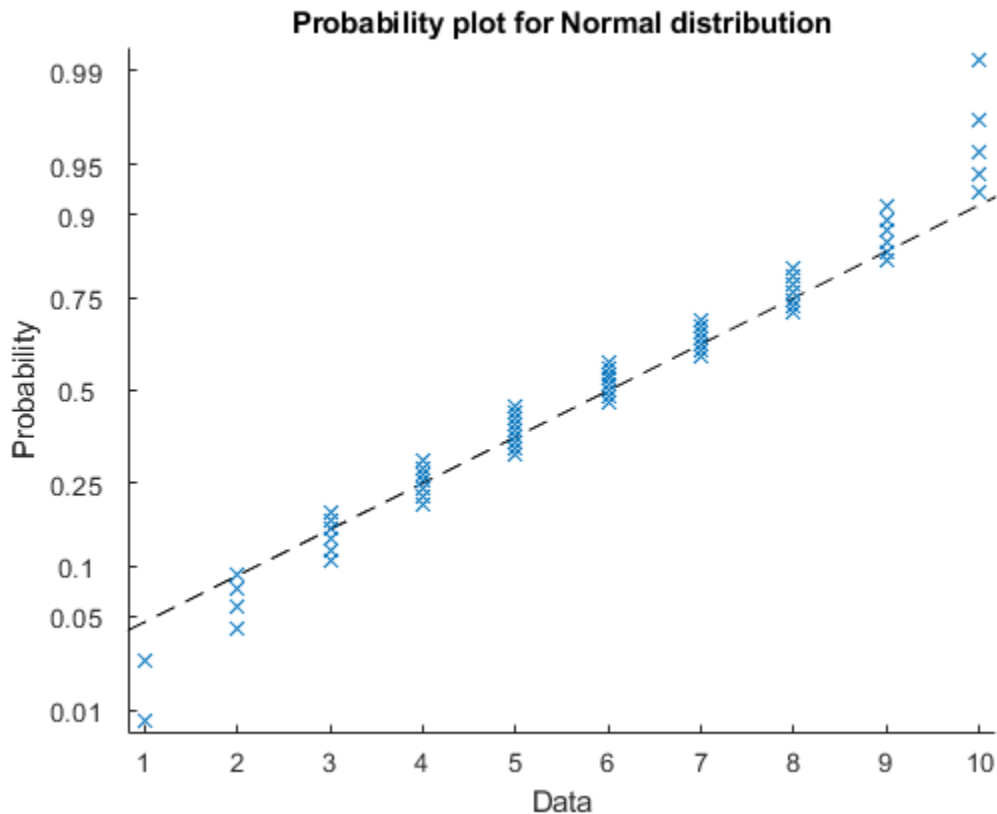
### Create a Normal Probability Plot Using Frequency Data

Generate simulated frequency data.

```
y = 1:10;
freq = [2 4 6 7 9 8 7 7 6 5];
```

Create a normal probability plot using the frequency data.

```
probplot(y, [], freq)
```



The normal probability plot shows that the data do not have a normal distribution.

## Input Arguments

### **y** — Sample data

numeric vector | numeric matrix

Sample data, specified as a numeric vector or numeric matrix. `probplot` displays each value in `y` using marker symbols including `'x'` and `'o'`. If `y` is a matrix, then `probplot` displays a separate line for each column of `y`.

Not all distributions are appropriate for all data sets. `probplot` errors if the data set is inappropriate for a specified distribution. See `dist` for appropriate data ranges for each distribution.

**dist** — Distribution for probability plot

'normal' (default) | probability distribution object | character vector

Distribution for probability plot, specified as a probability distribution object or one of the following distribution names:

Name	Plot Type	Data Range
'normal'	Normal probability plot	All values
'exponential'	Exponential probability plot	Nonnegative values
'extreme value'	Extreme value probability plot	All values
'half normal'	Half-normal probability plot	All values
'lognormal'	Lognormal probability plot	Positive values
'logistic'	Logistic probability plot	All values
'loglogistic'	Loglogistic probability plot	Positive values
'rayleigh'	Rayleigh probability plot	Positive values
'weibull'	Weibull probability plot	Positive values

You can create a probability distribution object with specified parameter values using `makedist`. Alternatively, fit a probability distribution object to sample data using `fitdist`. For more information on probability distribution objects, see “Working with Probability Distributions” on page 5-3.

The *y*-axis scale is based on the selected distribution. The *x*-axis has a log scale for the Weibull, loglogistic, and lognormal distributions, and a linear scale for the others.

Not all distributions are appropriate for all data sets. `probplot` errors if the data set is inappropriate for a specified distribution.

Example: `'weibull'`

**cens** — Censoring data

numeric vector

Censoring data, specified as a numeric vector. `cens` must be the same length as `y`, and contain a 1 value for observations that are right-censored and a 0 value for observations that are measured exactly.

Data Types: `single` | `double`

### **freq** — Frequency data

vector of integer values

Frequency data, specified as a vector of integer values. `freq` must be the same length as `y`. `freq` contains the integer frequencies for the corresponding elements in `y`.

To create a probability plot using frequency data but not censoring data, specify empty brackets (`[]`) for `cens`.

Data Types: `single` | `double`

### **ax** — Axis handle

axis handle object

Axis handle, specified as an axis handle object. `probplot` adds additional lines for the data in `y` to the plot corresponding to the axis specified by `ax`. Determine the axis handle for the current plot using `gca`.

### **pd** — Probability distribution for reference line

probability distribution object

Probability distribution for reference line, specified as a probability distribution object. `probplot` adds a fitted line to the axes specified by `ax` to represent the probability distribution specified by `pd`.

Create a probability distribution object with specified parameter values using `makedist`. Alternatively, fit a probability distribution object to sample data using `fitdist`. For more information on probability distribution objects, see “Working with Probability Distributions” on page 5-3.

### **fun** — Function for reference line

function handle

Function for reference line, specified as a function handle. `probplot` adds a fitted line to the axes specified by `ax` to represent the function specified by `fun`, evaluated at the parameters specified by `params`.



`fun` is a function handle to a cdf function, specified using the function handle operator `@`. The function must accept a vector of input values as its first argument, and return a vector containing the cdf evaluated at each input value. Specify the parameter values required to evaluate `fun` using the `params` argument. For more information on function handles, see “Create Function Handle” (MATLAB).

Example: `@wblpdf`

Data Types: `function_handle`

### **params** — Reference line function parameters

vector of numeric values | cell array

Reference line function parameters, specified as a vector of numeric values or a cell array. `probplot` adds a fitted line to the axes specified by `ax` to represent the function specified by `fun`, evaluated at the parameters specified by `params`.

`fun` is a function handle to a cdf function, specified using the function handle operator `@`. The function must accept a vector of values as its first argument, and return a vector of cdf values evaluated at each value. Specify the parameter values required to evaluate `fun` using the `params` argument. For more information on function handles, see “Create Function Handle” (MATLAB).

## Output Arguments

### **h** — Graphic handles for line objects

vector of `Line` graphic handles

Graphic handles for line objects, returned as a vector of `Line` graphic handles. Graphic handles are unique identifiers that you can use to query and modify the properties of a specific line on the plot. For each column of `y`, `probplot` returns two handles:

- The line representing the data points. `probplot` represents each data point in `y` using scatterplot symbols such as '+' and 'o'.
- The line showing the theoretical distribution for the probability plot, represented as a dashed line.

To view and set properties of line objects, use dot notation. For information on using dot notation, see “Access Property Values” (MATLAB). For information on the `Line` properties that you can set, see `Primitive Line`.

## Algorithms

`probplot` matches the quantiles of sample data to the quantiles of a given probability distribution. The sample data is sorted, scaled according to the choice of `dist`, and plotted on the x-axis. When `dist` is `'lognormal'`, `'loglogistic'`, or `'weibull'`, the scaling is logarithmic. Otherwise, the scaling is linear. The y-axis represents the quantiles of the distribution specified in `dist`, converted into probability values. The scaling depends on the given distribution and is not linear.

Where the x-axis value is the  $i$ th sorted value from a sample of size  $N$ , the y-axis value is the midpoint between evaluation points of the empirical cumulative distribution function

of the data. In the case of uncensored data, the midpoint is equal to  $\frac{(i-0.5)}{N}$ .

`probplot` superimposes a reference line to assess the linearity of the plot. If the data is uncensored, then the line goes through the first and third quartiles of the data, unless `dist` is `'half normal'`. If `dist` is `'half normal'`, then `probplot` uses the zeroth and second quartiles instead. If the data is censored, then the line shifts accordingly.

## See Also

`ecdf` | `normplot` | `wblplot`

## Topics

“Distribution Plots” on page 4-10

Introduced before R2006a

# procrustes

Procrustes analysis

## Syntax

```
d = procrustes(X,Y)
[d,Z] = procrustes(X,Y)
[d,Z,transform] = procrustes(X,Y)
[...] = procrustes(...,'scaling',flag)
[...] = procrustes(...,'reflection',flag)
```

## Description

`d = procrustes(X,Y)` determines a linear transformation (translation, reflection, orthogonal rotation, and scaling) of the points in matrix `Y` to best conform them to the points in matrix `X`. The goodness-of-fit criterion is the sum of squared errors.

`procrustes` returns the minimized value of this dissimilarity measure in `d`. `d` is standardized by a measure of the scale of `X`, given by:

```
sum(sum((X-repmat(mean(X,1),size(X,1),1)).^2,1))
```

That is, the sum of squared elements of a centered version of `X`. However, if `X` comprises repetitions of the same point, the sum of squared errors is not standardized.

`X` and `Y` must have the same number of points (rows), and `procrustes` matches `Y(i)` to `X(i)`. Points in `Y` can have smaller dimension (number of columns) than those in `X`. In this case, `procrustes` adds columns of zeros to `Y` as necessary.

`[d,Z] = procrustes(X,Y)` also returns the transformed `Y` values.

`[d,Z,transform] = procrustes(X,Y)` also returns the transformation that maps `Y` to `Z`. `transform` is a structure array with fields:

- `c` — Translation component
- `T` — Orthogonal rotation and reflection component

- `b` — Scale component

That is:

```
c = transform.c;  
T = transform.T;  
b = transform.b;
```

```
Z = b*Y*T + c;
```

`[...] = procrustes(..., 'scaling', flag)`, when *flag* is false, allows you to compute the transformation without a scale component (that is, with `b` equal to 1). The default *flag* is true.

`[...] = procrustes(..., 'reflection', flag)`, when *flag* is false, allows you to compute the transformation without a reflection component (that is, with `det(T)` equal to 1). The default *flag* is 'best', which computes the best-fitting transformation, whether or not it includes a reflection component. A *flag* of true forces the transformation to be computed with a reflection component (that is, with `det(T)` equal to -1)

## Examples

### Procrustes Analysis

Generate the sample data in two dimensions.

```
rng('default')  
n = 10;  
X = normrnd(0,1,[n 2]);
```

Rotate, scale, translate, and add some noise to sample points.

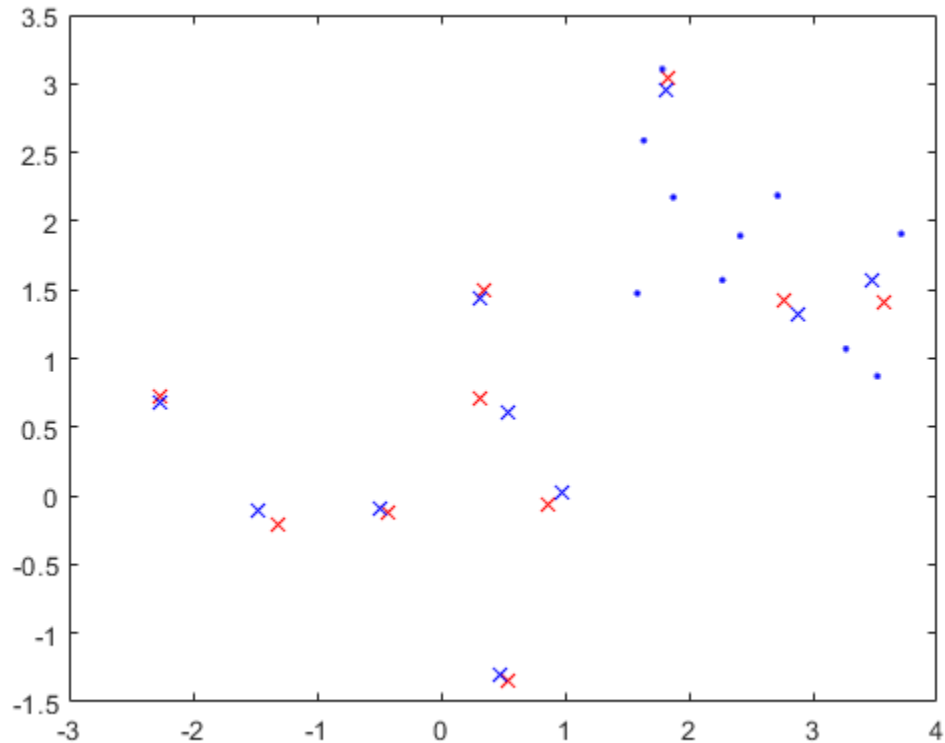
```
S = [0.5 -sqrt(3)/2; sqrt(3)/2 0.5];  
Y = normrnd(0.5*X*S+2,0.05,n,2);
```

Conform `Y` to `X` using procrustes analysis.

```
[d,Z,tr] = procrustes(X,Y);
```

Plot the original `X` and `Y` with the transformed `Y`.

```
plot(X(:,1),X(:,2),'rx',Y(:,1),Y(:,2),'b.',Z(:,1),Z(:,2),'bx');
```



## References

- [1] Kendall, David G. "A Survey of the Statistical Theory of Shape." *Statistical Science*. Vol. 4, No. 2, 1989, pp. 87–99.
- [2] Bookstein, Fred L. *Morphometric Tools for Landmark Data*. Cambridge, UK: Cambridge University Press, 1991.
- [3] Seber, G. A. F. *Multivariate Observations*. Hoboken, NJ: John Wiley & Sons, Inc., 1984.

## See Also

`cmdscale` | `factoran`

**Introduced before R2006a**

# proflik

Profile likelihood function for probability distribution

## Syntax

```
[ll,param] = proflik(pd,pnum)
[ll,param] = proflik(pd,pnum,'Display',display)
[ll,param] = proflik(pd,pnum,setparam)
[ll,param] = proflik(pd,pnum,setparam,'Display',display)
[ll,param,other] = proflik(____)
```

## Description

[ll,param] = proflik(pd,pnum) returns a vector ll of log likelihood values and a vector param of corresponding parameter values for the parameter in the position indicated by pnum.

[ll,param] = proflik(pd,pnum,'Display',display) returns the log likelihood values and corresponding parameter values, and plots the profile likelihood overlaid on an approximation of the log likelihood.

[ll,param] = proflik(pd,pnum,setparam) returns the log likelihood values and corresponding parameter values as specified by setparam.

[ll,param] = proflik(pd,pnum,setparam,'Display',display) returns the log likelihood values and corresponding parameter values as specified by setparam, and plots the profile likelihood overlaid on an approximation of the log likelihood.

[ll,param,other] = proflik(\_\_\_\_) also returns a matrix other containing the values of the other parameters that maximize the likelihood, using any of the input arguments from the previous syntaxes.

## Examples

### Profile Likelihood of a Distribution Parameter

Load the sample data. Create a probability distribution object by fitting a Weibull distribution to the miles per gallon (MPG) data.

```
load carsmall
pd = fitdist(MPG, 'Weibull')

pd =

    WeibullDistribution

    Weibull distribution
    A = 26.5079    [24.8333, 28.2954]
    B = 3.27193   [2.79441, 3.83104]
```

View the parameter names for the distribution.

```
pd.ParameterNames

ans =

    1x2 cell array

    {'A'}    {'B'}
```

For the Weibull distribution, A is in position 1, and B is in position 2.

Compute the profile likelihood for B, which is in position `pnum = 2`.

```
[ll,param] = proflik(pd,2);
```

Display the loglikelihood values for the estimated values of B.

```
[ll',param']

ans =

    -329.9688    2.7132
    -329.4312    2.7748
```



```
-328.9645    2.8365
-328.5661    2.8981
-328.2340    2.9597
-327.9658    3.0213
-327.7596    3.0830
-327.6135    3.1446
-327.5256    3.2062
-327.4943    3.2678
-327.5178    3.3295
-327.5946    3.3911
-327.7233    3.4527
-327.9023    3.5143
-328.1303    3.5760
-328.4060    3.6376
-328.7281    3.6992
-329.0956    3.7608
-329.5071    3.8224
-329.9617    3.8841
-330.4583    3.9457
```

These results show that the profile log likelihood is maximized between the estimated  $B$  values of 3.2678 and 3.3295, which correspond to loglikelihood values -327.4943 and -327.5178. From the earlier fit, the MLE of  $B$  is 3.27193, which is in this interval as expected.

### Profile Likelihood With Restricted Parameter Values

Load the sample data. Create a probability distribution object by fitting a generalized extreme value distribution to the miles per gallon (MPG) data.

```
load carsmall
pd = fitdist(MPG, 'GeneralizedExtremeValue')

pd =

GeneralizedExtremeValueDistribution

Generalized Extreme Value distribution
    k = -0.207765    [-0.381674, -0.0338564]
    sigma = 7.49674    [6.31755, 8.89603]
```

```
mu = 20.6233 [18.8859, 22.3606]
```

View the parameter names for the distribution.

```
pd.ParameterNames
```

```
ans =
```

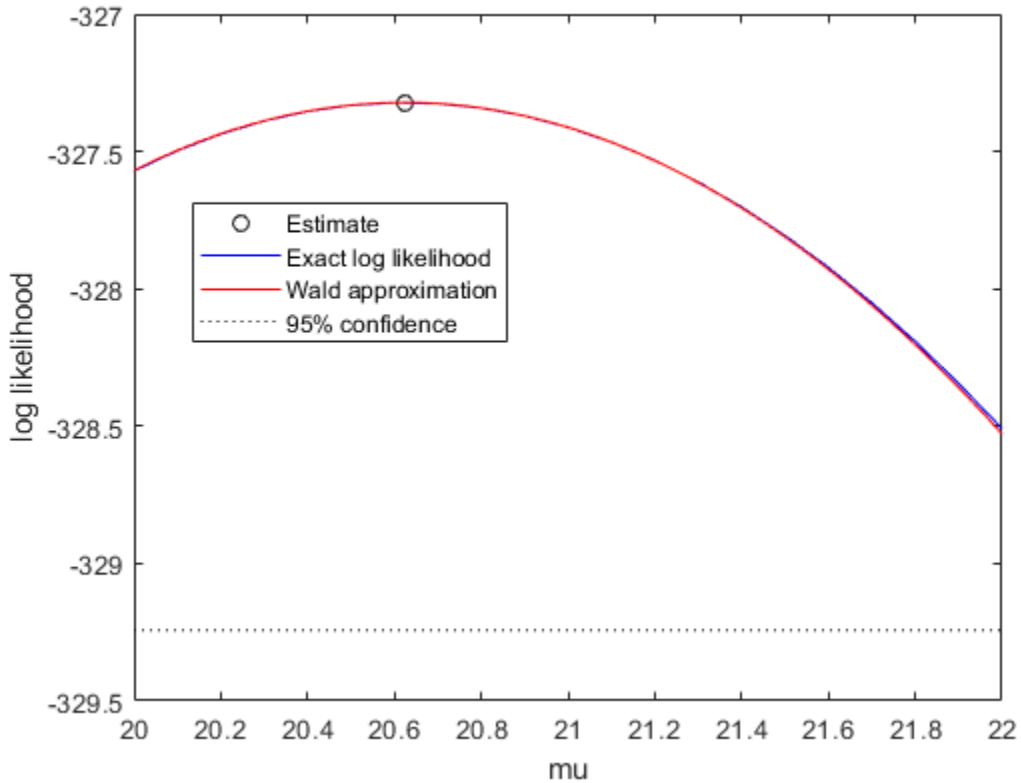
```
1x3 cell array
```

```
{'k'} {'sigma'} {'mu'}
```

For the generalized extreme value distribution, `k` is in position 1, `sigma` is in position 2, and `mu` is in position 3.

Compute the profile likelihood for `mu`, which is in position `pnum = 3`. Restrict the computation to parameter values from 20 to 22, and display the plot.

```
[ll,param,other] = proflik(pd,3,20:.1:22,'display','on');
```



The plot shows the estimated value for the parameter  $\mu$  that maximizes the loglikelihood.

Display the loglikelihood values for the estimated values of  $\mu$ , and the values of the other distribution parameters that maximize the corresponding loglikelihood.

```
[ll',param',other]
```

ans =

-327.5706	20.0000	-0.1803	7.4087
-327.4971	20.1000	-0.1846	7.4218
-327.4364	20.2000	-0.1890	7.4354

-327.3887	20.3000	-0.1934	7.4493
-327.3538	20.4000	-0.1978	7.4636
-327.3317	20.5000	-0.2023	7.4783
-327.3223	20.6000	-0.2067	7.4932
-327.3257	20.7000	-0.2112	7.5084
-327.3418	20.8000	-0.2156	7.5240
-327.3706	20.9000	-0.2201	7.5399
-327.4119	21.0000	-0.2245	7.5560
-327.4659	21.1000	-0.2289	7.5723
-327.5324	21.2000	-0.2333	7.5889
-327.6113	21.3000	-0.2378	7.6057
-327.7027	21.4000	-0.2422	7.6228
-327.8065	21.5000	-0.2465	7.6400
-327.9227	21.6000	-0.2509	7.6575
-328.0511	21.7000	-0.2553	7.6751
-328.1917	21.8000	-0.2596	7.6930
-328.3446	21.9000	-0.2639	7.7111
-328.5095	22.0000	-0.2682	7.7293

The first column contains the log likelihood value that corresponds to the estimate of  $\mu$  in the second column. The log likelihood is maximized between the parameter values 20.6000 and 20.7000, corresponding to log likelihood values -327.3223 and -327.3257. The third column contains the value of  $k$  that maximizes the corresponding log likelihood for  $\mu$ . The fourth column contains the value of  $\sigma$  that maximizes the corresponding log likelihood for  $\mu$ .

## Input Arguments

### **pd** — Probability distribution

probability distribution object

Probability distribution, specified as a probability distribution object. Create a probability distribution object with specified parameter values using `makedist`. Alternatively, create a probability distribution object by fitting it to data using `fitdist` or the Distribution Fitter app.

### **pnum** — Parameter number

positive integer value

Parameter number for which to compute the profile likelihood, specified as a positive integer value corresponding to the position of the desired parameter in the parameter

name vector. For example, a Weibull distribution has a parameter name vector `{ 'A', 'B' }`, so specify `pnum` as 2 to compute the profile likelihood for B.

Data Types: `single` | `double`

### **setparam** — Parameter value restriction

scalar value | vector of scalar values

Parameter value restriction, specified as a scalar value or a vector of such values. If you do not specify `setparam`, `proflik` chooses the values for output vector `param` based on the default confidence interval method for the probability distribution `pd`. If the parameter can take only restricted values, and if the confidence interval violates that restriction, you can use `setparam` to specify valid values.

Example: `[3, 3.5, 4]`

### **display** — Display toggle

'off' (default) | 'on'

Display toggle, specified as either 'on' or 'off'. Specify 'on' to display a plot of the profile log likelihood overlaid on an approximation of the log likelihood. Specify 'off' to omit the display. The approximation is based on a Taylor series expansion around the estimated parameter value, as a function of the parameter in position `pnum` or its logarithm. The intersection of the curves with the horizontal dotted line marks the endpoints of 95% confidence intervals.

## Output Arguments

### **ll** — Log likelihood values

vector

Log likelihood values, returned as a vector. The log likelihood is the value of the likelihood with the parameter in position `pnum` set to the values in `param`, maximized over the remaining parameters.

### **param** — Parameter values

vector

Parameter values corresponding to the loglikelihood values in `ll`, returned as a vector. If you specify parameter values using `setparam`, then `param` is equal to `setparam`.

**other** — Other parameter values`matrix`

Other parameter values that maximize the likelihood, returned as a matrix. Each row of `other` contains the values for all parameters except the parameter in position `pnum`.

**See Also**`distributionFitter` | `fitdist` | `makedist`**Topics**

“Working with Probability Distributions” on page 5-3

**Introduced before R2006a**

# proflik

**Class:** prob.ToolboxFittableParametricDistribution

**Package:** prob

Profile likelihood function for probability distribution object

## Syntax

```
[ll,param] = proflik(pd,pnum)
[ll,param] = proflik(pd,pnum,'Display',display)
[ll,param] = proflik(pd,pnum,setparam)
[ll,param] = proflik(pd,pnum,setparam,'Display',display)
[ll,param,other] = proflik(____)
```

## Description

[ll,param] = proflik(pd,pnum) returns a vector ll of loglikelihood values and a vector param of corresponding parameter values for the parameter in the position indicated by pnum.

[ll,param] = proflik(pd,pnum,'Display',display) returns the loglikelihood values and corresponding parameter values, and plots the profile likelihood overlaid on an approximation of the loglikelihood.

[ll,param] = proflik(pd,pnum,setparam) returns the loglikelihood values and corresponding parameter values as specified by setparam.

[ll,param] = proflik(pd,pnum,setparam,'Display',display) returns the loglikelihood values and corresponding parameter values as specified by setparam, and plots the profile likelihood overlaid on an approximation of the loglikelihood.

[ll,param,other] = proflik(\_\_\_\_) also returns a matrix other containing the values of the other parameters that maximize the likelihood, using any of the input arguments from the previous syntaxes.

## Input Arguments

### **pd** — Probability distribution

probability distribution object

Probability distribution, specified as a probability distribution object. Create a probability distribution object with specified parameter values using `makedist`. Alternatively, create a probability distribution object by fitting it to data using `fitdist` or the Distribution Fitter app.

### **pnum** — Parameter number

positive integer value

Parameter number for which to compute the profile likelihood, specified as a positive integer value corresponding to the position of the desired parameter in the parameter name vector. For example, a Weibull distribution has a parameter name vector `{ 'A', 'B' }`, so specify `pnum` as 2 to compute the profile likelihood for B.

Data Types: `single` | `double`

### **setparam** — Parameter value restriction

scalar value | vector of scalar values

Parameter value restriction, specified as a scalar value or a vector of such values. If you do not specify `setparam`, `proflik` chooses the values for output vector `param` based on the default confidence interval method for the probability distribution `pd`. If the parameter can take only restricted values, and if the confidence interval violates that restriction, you can use `setparam` to specify valid values.

Example: `[3, 3.5, 4]`

### **display** — Display toggle

'off' (default) | 'on'

Display toggle, specified as either 'on' or 'off'. Specify 'on' to display a plot of the profile loglikelihood overlaid on an approximation of the loglikelihood. Specify 'off' to omit the display. The approximation is based on a Taylor series expansion around the estimated parameter value, as a function of the parameter in position `pnum` or its logarithm. The intersection of the curves with the horizontal dotted line marks the endpoints of 95% confidence intervals.



## Output Arguments

### **ll** — Loglikelihood values

vector

Loglikelihood values, returned as a vector. The loglikelihood is the value of the likelihood with the parameter in position `pnum` set to the values in `param`, maximized over the remaining parameters.

### **param** — Parameter values

vector

Parameter values corresponding to the loglikelihood values in `ll`, returned as a vector. If you specify parameter values using `setparam`, then `param` is equal to `setparam`.

### **other** — Other parameter values

matrix

Other parameter values that maximize the likelihood, returned as a matrix. Each row of `other` contains the values for all parameters except the parameter in position `pnum`.

## Examples

### Profile Likelihood of a Distribution Parameter

Load the sample data. Create a probability distribution object by fitting a Weibull distribution to the miles per gallon (MPG) data.

```
load carsmall
pd = fitdist(MPG, 'Weibull')

pd =

WeibullDistribution

Weibull distribution
A = 26.5079    [24.8333, 28.2954]
B = 3.27193   [2.79441, 3.83104]
```

View the parameter names for the distribution.

```
pd.ParameterNames
```

```
ans =
```

```
1x2 cell array  
  
{'A'}    {'B'}
```

For the Weibull distribution, A is in position 1, and B is in position 2.

Compute the profile likelihood for B, which is in position `pnum = 2`.

```
[ll,param] = proflik(pd,2);
```

Display the loglikelihood values for the estimated values of B.

```
[ll',param']
```

```
ans =
```

```
-329.9688    2.7132  
-329.4312    2.7748  
-328.9645    2.8365  
-328.5661    2.8981  
-328.2340    2.9597  
-327.9658    3.0213  
-327.7596    3.0830  
-327.6135    3.1446  
-327.5256    3.2062  
-327.4943    3.2678  
-327.5178    3.3295  
-327.5946    3.3911  
-327.7233    3.4527  
-327.9023    3.5143  
-328.1303    3.5760  
-328.4060    3.6376  
-328.7281    3.6992  
-329.0956    3.7608  
-329.5071    3.8224  
-329.9617    3.8841
```

```
-330.4583    3.9457
```

These results show that the profile log likelihood is maximized between the estimated  $B$  values of 3.2678 and 3.3295, which correspond to loglikelihood values -327.4943 and -327.5178. From the earlier fit, the MLE of  $B$  is 3.27193, which is in this interval as expected.

### Profile Likelihood With Restricted Parameter Values

Load the sample data. Create a probability distribution object by fitting a generalized extreme value distribution to the miles per gallon (MPG) data.

```
load carsmall
pd = fitdist(MPG, 'GeneralizedExtremeValue')

pd =

GeneralizedExtremeValueDistribution

Generalized Extreme Value distribution
    k = -0.207765    [-0.381674, -0.0338564]
  sigma =  7.49674    [6.31755, 8.89603]
    mu =  20.6233    [18.8859, 22.3606]
```

View the parameter names for the distribution.

```
pd.ParameterNames

ans =

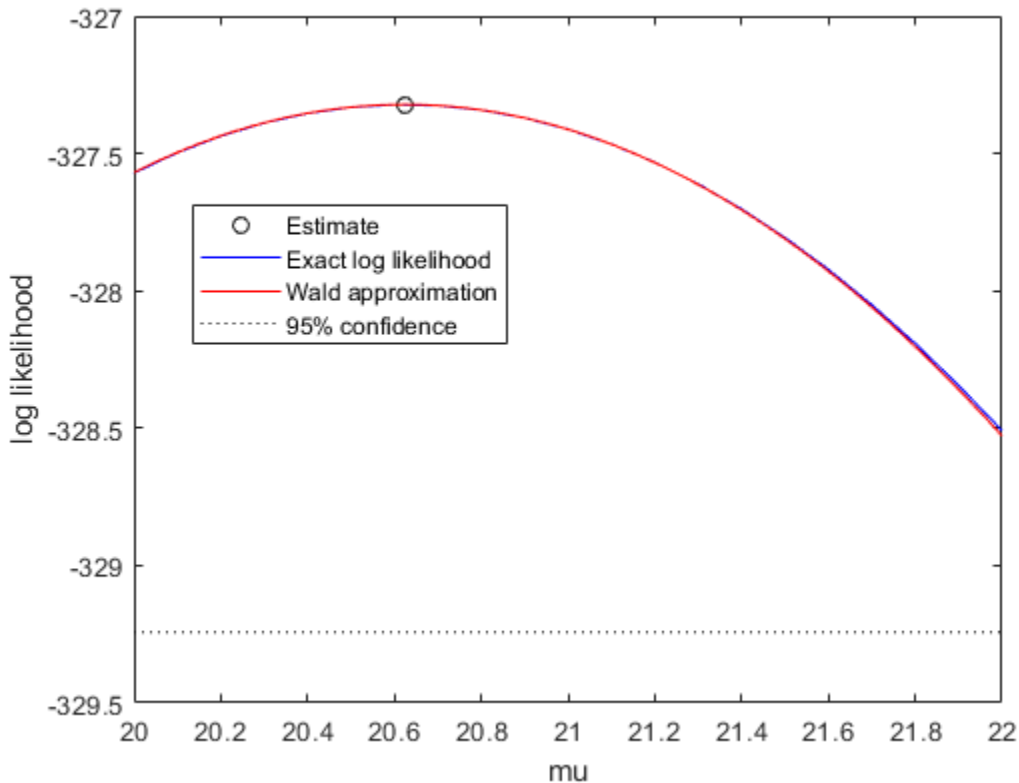
1x3 cell array

    {'k'}    {'sigma'}    {'mu'}
```

For the generalized extreme value distribution,  $k$  is in position 1,  $\sigma$  is in position 2, and  $\mu$  is in position 3.

Compute the profile likelihood for  $\mu$ , which is in position `pnum = 3`. Restrict the computation to parameter values from 20 to 22, and display the plot.

```
[ll,param,other] = proflik(pd,3,20:.1:22,'display','on');
```



The plot shows the estimated value for the parameter  $\mu$  that maximizes the loglikelihood.

Display the loglikelihood values for the estimated values of  $\mu$ , and the values of the other distribution parameters that maximize the corresponding loglikelihood.

```
[ll',param',other]
```

```
ans =
```

-327.5706	20.0000	-0.1803	7.4087
-327.4971	20.1000	-0.1846	7.4218
-327.4364	20.2000	-0.1890	7.4354
-327.3887	20.3000	-0.1934	7.4493
-327.3538	20.4000	-0.1978	7.4636
-327.3317	20.5000	-0.2023	7.4783
-327.3223	20.6000	-0.2067	7.4932
-327.3257	20.7000	-0.2112	7.5084
-327.3418	20.8000	-0.2156	7.5240
-327.3706	20.9000	-0.2201	7.5399
-327.4119	21.0000	-0.2245	7.5560
-327.4659	21.1000	-0.2289	7.5723
-327.5324	21.2000	-0.2333	7.5889
-327.6113	21.3000	-0.2378	7.6057
-327.7027	21.4000	-0.2422	7.6228
-327.8065	21.5000	-0.2465	7.6400
-327.9227	21.6000	-0.2509	7.6575
-328.0511	21.7000	-0.2553	7.6751
-328.1917	21.8000	-0.2596	7.6930
-328.3446	21.9000	-0.2639	7.7111
-328.5095	22.0000	-0.2682	7.7293

The first column contains the log likelihood value that corresponds to the estimate of  $\mu$  in the second column. The log likelihood is maximized between the parameter values 20.6000 and 20.7000, corresponding to log likelihood values -327.3223 and -327.3257. The third column contains the value of  $k$  that maximizes the corresponding log likelihood for  $\mu$ . The fourth column contains the value of  $\sigma$  that maximizes the corresponding log likelihood for  $\mu$ .

## See Also

`distributionFitter` | `fitdist` | `makedist`

## proximity

**Class:** CompactTreeBagger

Proximity matrix for data

## Syntax

```
prox = proximity(B, X)
```

## Description

`prox = proximity(B, X)` computes a numeric matrix of size `Nobs-by-Nobs` of proximities for data `X`, where `Nobs` is the number of observations (rows) in `X`. Proximity between any two observations in the input data is defined as a fraction of trees in the ensemble `B` for which these two observations land on the same leaf. This is a symmetric matrix with ones on the diagonal and off-diagonal elements ranging from 0 to 1.

# Proximity property

**Class:** `TreeBagger`

Proximity matrix for observations

## Description

The `Proximity` property is a numeric matrix of size `Nobs-by-Nobs`, where `Nobs` is the number of observations in the training data, containing measures of the proximity between observations. For any two observations, their proximity is defined as the fraction of trees for which these observations land on the same leaf. This is a symmetric matrix with 1s on the diagonal and off-diagonal elements ranging from 0 to 1.

## See Also

`ClassificationTree` | `RegressionTree` | `TreeBagger` | `fitctree` | `fitrtree` | `proximity`

## prune

**Class:** ClassificationTree

Produce sequence of subtrees by pruning

## Syntax

```
tree1 = prune(tree)
tree1 = prune(tree, Name, Value)
```

## Description

`tree1 = prune(tree)` creates a copy of the classification tree `tree` with its optimal pruning sequence filled in.

`tree1 = prune(tree, Name, Value)` creates a pruned tree with additional options specified by one `Name, Value` pair argument. You can specify several name-value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

## Input Arguments

### **tree**

A classification tree created with `fitctree`.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.



**Alpha**

A numeric scalar. `prune` prunes `tree` to the specified value of the pruning cost.

**Level**

A numeric scalar from 0 (no pruning) to the largest pruning level of this tree `max(tree.PruneList)`. `prune` returns the tree pruned to this level.

**Nodes**

A numeric vector with elements from 1 to `tree.NumNodes`. Any tree branch nodes listed in `nodes` become leaf nodes in `tree1`, unless their parent nodes are also pruned.

## Output Arguments

**tree1**

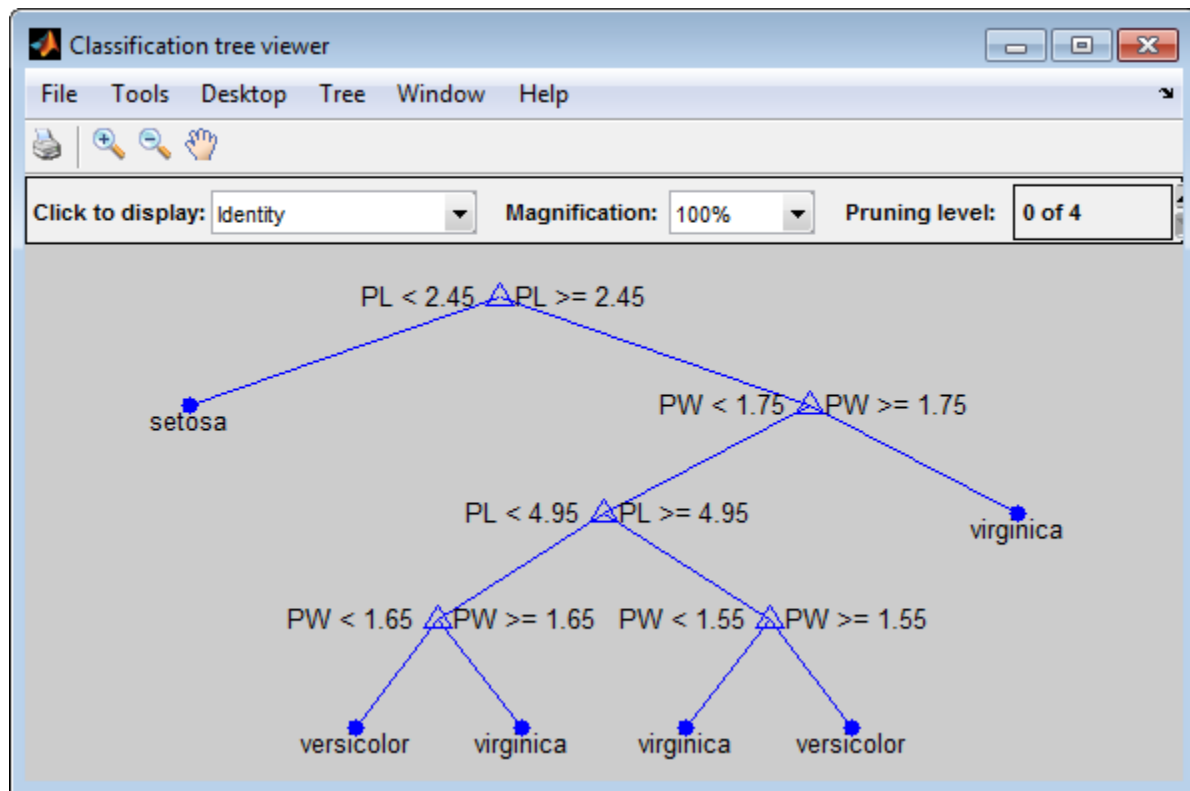
A classification tree.

## Examples

**Prune and Display a Classification Tree**

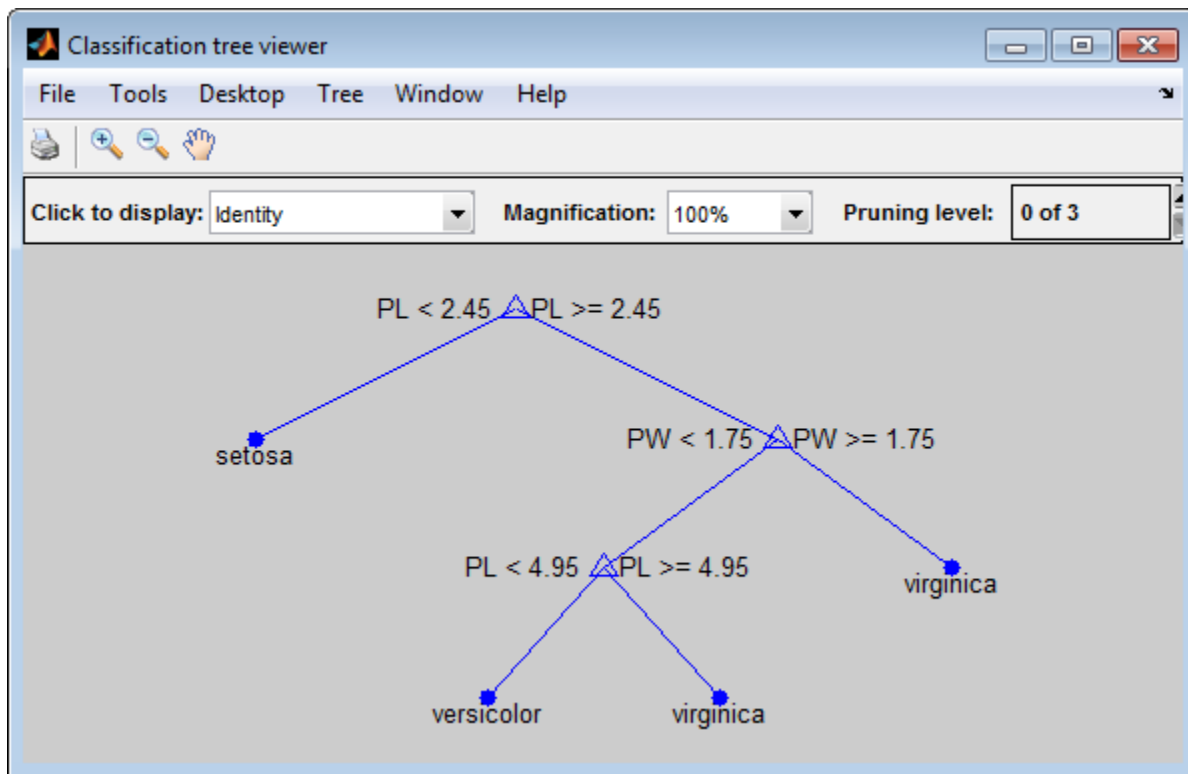
Construct and display a full classification tree for Fisher's iris data.

```
load fisheriris;
varnames = {'SL' 'SW' 'PL' 'PW'};
t1 = fitctree(meas,species,...
    'minparent',5,'predictorNames',varnames);
view(t1,'mode','graph');
```



Construct and display the next largest tree from the optimal pruning sequence.

```
t2 = prune(t1, 'level', 1);
view(t2, 'mode', 'graph');
```



## Tips

- `tree1 = prune(tree)` returns the decision tree `tree1` that is the full, unpruned tree, but with optimal pruning information added. This is useful only if you created `tree` by pruning another tree, or by using the `fitctree` function with pruning set 'off'. If you plan to prune a tree multiple times along the optimal pruning sequence, it is more efficient to create the optimal pruning sequence first.

## See Also

`fitctree`

## prune

**Class:** `classregtree`

Prune tree

---

**Note** `classregtree` will be removed in a future release. See `fitctree`, `fitrtree`, `ClassificationTree`, or `RegressionTree` instead.

---

## Syntax

```
t2 = prune(t1, 'level', level)
t2 = prune(t1, 'nodes', nodes)
t2 = prune(t1)
```

## Description

`t2 = prune(t1, 'level', level)` takes a decision tree `t1` and a pruning level `level`, and returns the decision tree `t2` pruned to that level. If `level` is 0, there is no pruning. Trees are pruned based on an optimal pruning scheme that first prunes branches giving less improvement in error cost.

`t2 = prune(t1, 'nodes', nodes)` prunes the nodes listed in the `nodes` vector from the tree. Any `t1` branch nodes listed in `nodes` become leaf nodes in `t2`, unless their parent nodes are also pruned. Use `view` to display the node numbers for any node you select.

`t2 = prune(t1)` returns the decision tree `t2` that is the full, unpruned `t1`, but with optimal pruning information added. This is useful only if `t1` is created by pruning another tree, or by using the `classregtree` function with the `'prune'` parameter set to `'off'`. If you plan to prune a tree multiple times along the optimal pruning sequence, it is more efficient to create the optimal pruning sequence first.

Pruning is the process of reducing a tree by turning some branch nodes into leaf nodes and removing the leaf nodes under the original branch.

## Examples

### Prune a Decision Tree

Display the full tree for Fisher's iris data:

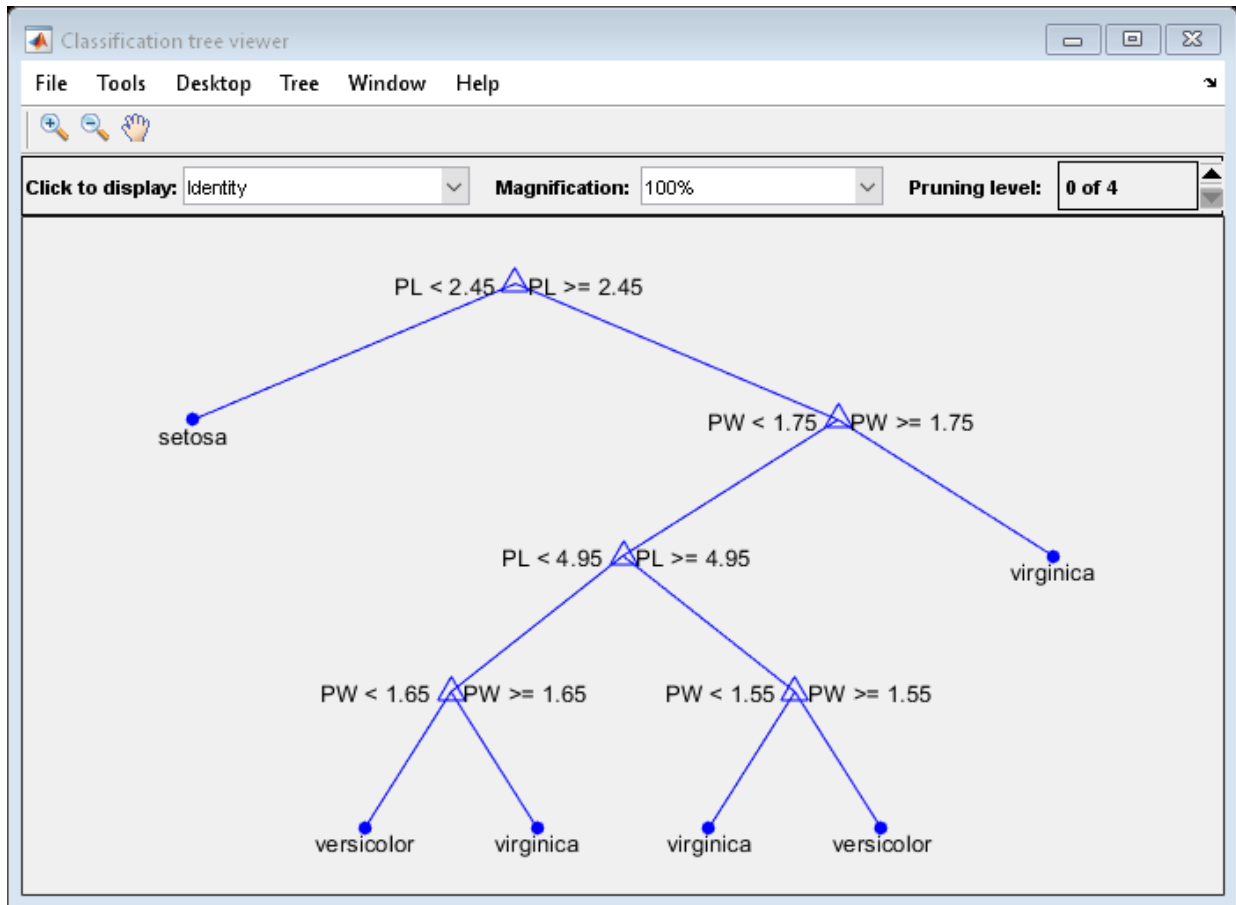
```
load fisheriris;
t1 = classregtree(meas, species, ...
                 'names', {'SL' 'SW' 'PL' 'PW'}, ...
                 'minparent', 5)
view(t1)
```

Warning: classregtree will be removed in a future release. Use fitctree or fitrtree instead.

```
t1 =
```

```
Decision tree for classification
```

```
1  if PL<2.45 then node 2 elseif PL>=2.45 then node 3 else setosa
2  class = setosa
3  if PW<1.75 then node 4 elseif PW>=1.75 then node 5 else versicolor
4  if PL<4.95 then node 6 elseif PL>=4.95 then node 7 else versicolor
5  class = virginica
6  if PW<1.65 then node 8 elseif PW>=1.65 then node 9 else versicolor
7  if PW<1.55 then node 10 elseif PW>=1.55 then node 11 else virginica
8  class = versicolor
9  class = virginica
10 class = virginica
11 class = versicolor
```



Display the next largest tree from the optimal pruning sequence:

```
t2 = prune(t1, 'level', 1)
view(t2)
```

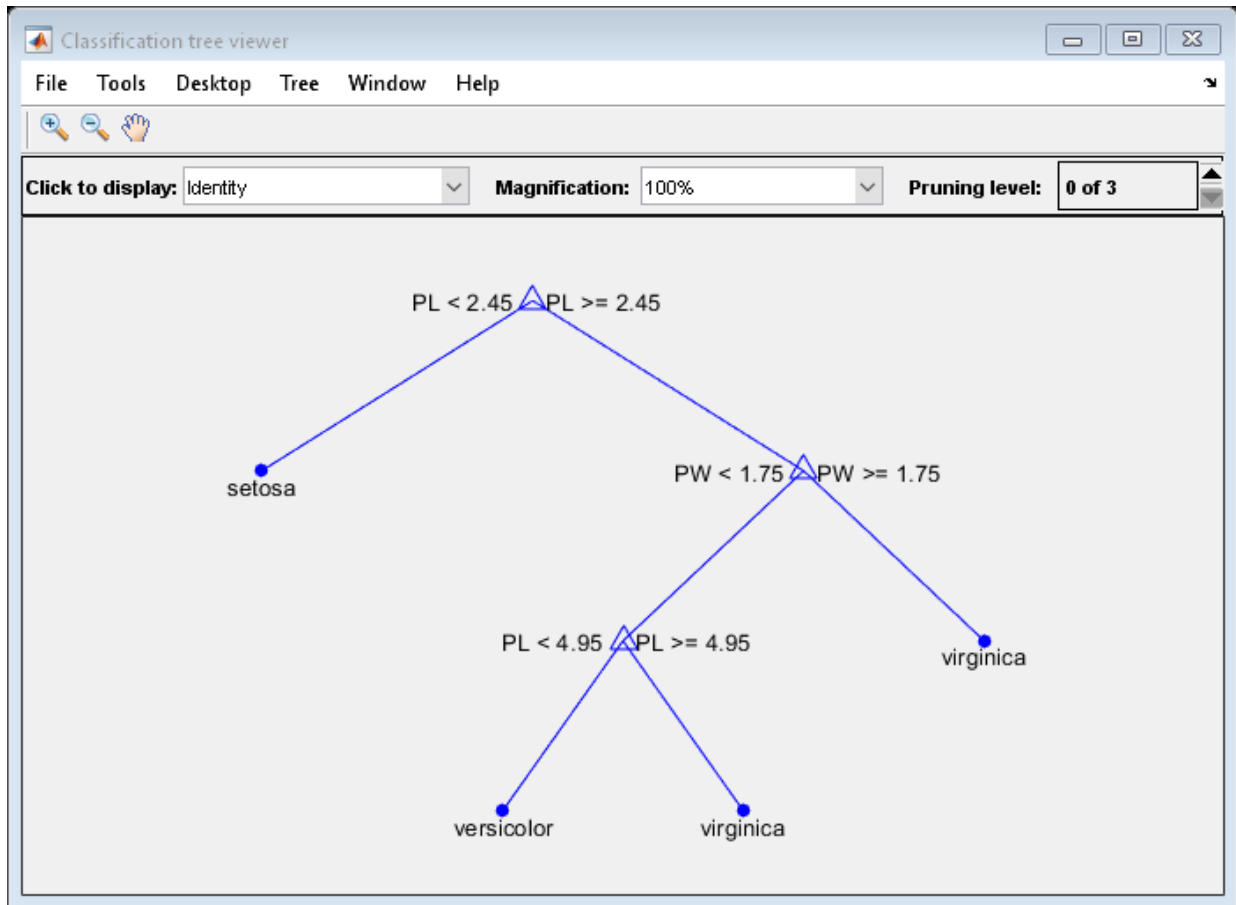
Warning: classregtree will be removed in a future release. Use the prune method of an object returned by fitctree or fitctree instead.

```
t2 =
```

```
Decision tree for classification
```

```
1 if PL<2.45 then node 2 elseif PL>=2.45 then node 3 else setosa
```

```
2 class = setosa
3 if PW<1.75 then node 4 elseif PW>=1.75 then node 5 else versicolor
4 if PL<4.95 then node 6 elseif PL>=4.95 then node 7 else versicolor
5 class = virginica
6 class = versicolor
7 class = virginica
```



## References

- [1] Breiman, L., J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Boca Raton, FL: CRC Press, 1984.

## See Also

[classregtree](#) | [test](#) | [view](#)



# prune

**Class:** RegressionTree

Produce sequence of subtrees by pruning

## Syntax

```
tree1 = prune(tree)
tree1 = prune(tree,Name,Value)
```

## Description

`tree1 = prune(tree)` creates a copy of the regression tree `tree` with its optimal pruning sequence filled in.

`tree1 = prune(tree,Name,Value)` creates a pruned tree with additional options specified by one `Name, Value` pair argument. You can specify several name-value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

## Input Arguments

**tree**

A regression tree created with `fitrtree`.

## Name-Value Pair Arguments

Optional comma-separated pair of `Name, Value` arguments, where `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify only one name-value pair argument.

**Alpha**

A numeric scalar from 0 (no pruning) to 1 (prune to one node). Prunes to minimize the sum of (Alpha times the number of leaf nodes) and a cost (mean squared error).

**Level**

A numeric scalar from 0 (no pruning) to the largest pruning level of this tree `max(tree.PruneList)`. `prune` returns the tree pruned to this level.

**Nodes**

A numeric vector with elements from 1 to `tree.NumNodes`. Any tree branch nodes listed in `Nodes` become leaf nodes in `tree1`, unless their parent nodes are also pruned.

## Output Arguments

**tree1**

A regression tree.

## Examples

**Prune Regression Tree**

Load the `carsmall` data set. Consider `Horsepower` and `Weight` as predictor variables.

```
load carsmall;
X = [Weight Horsepower];
varNames = {'Weight' 'Horsepower'};
```

Grow a regression tree using the entire data set. View the tree.

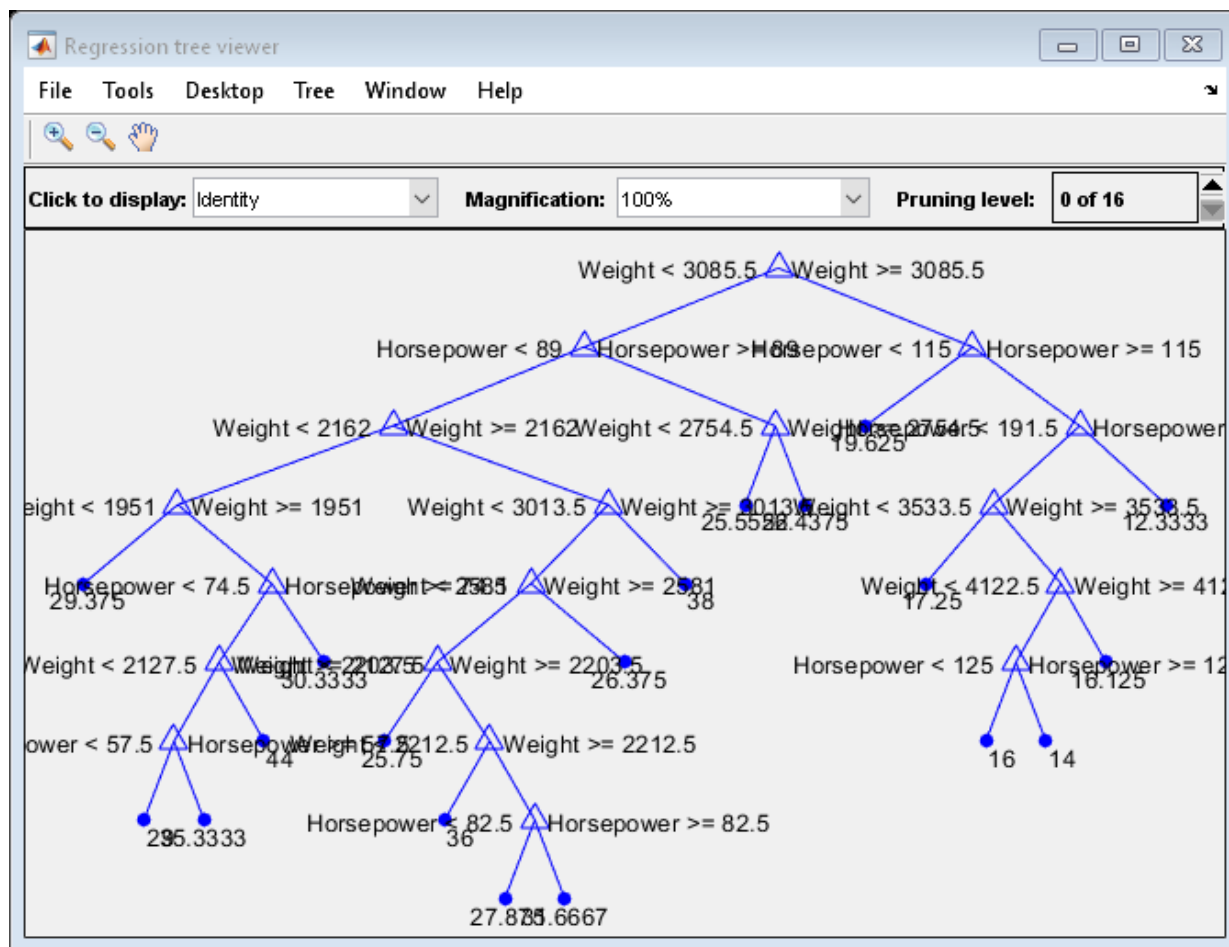
```
Mdl = fitrtree(X,MPG,'PredictorNames',varNames)
view(Mdl,'Mode','graph');
```

```
Mdl =
```

```

RegressionTree
  PredictorNames: {'Weight' 'Horsepower'}
  ResponseName: 'Y'
  CategoricalPredictors: []
  ResponseTransform: 'none'
  NumObservations: 94

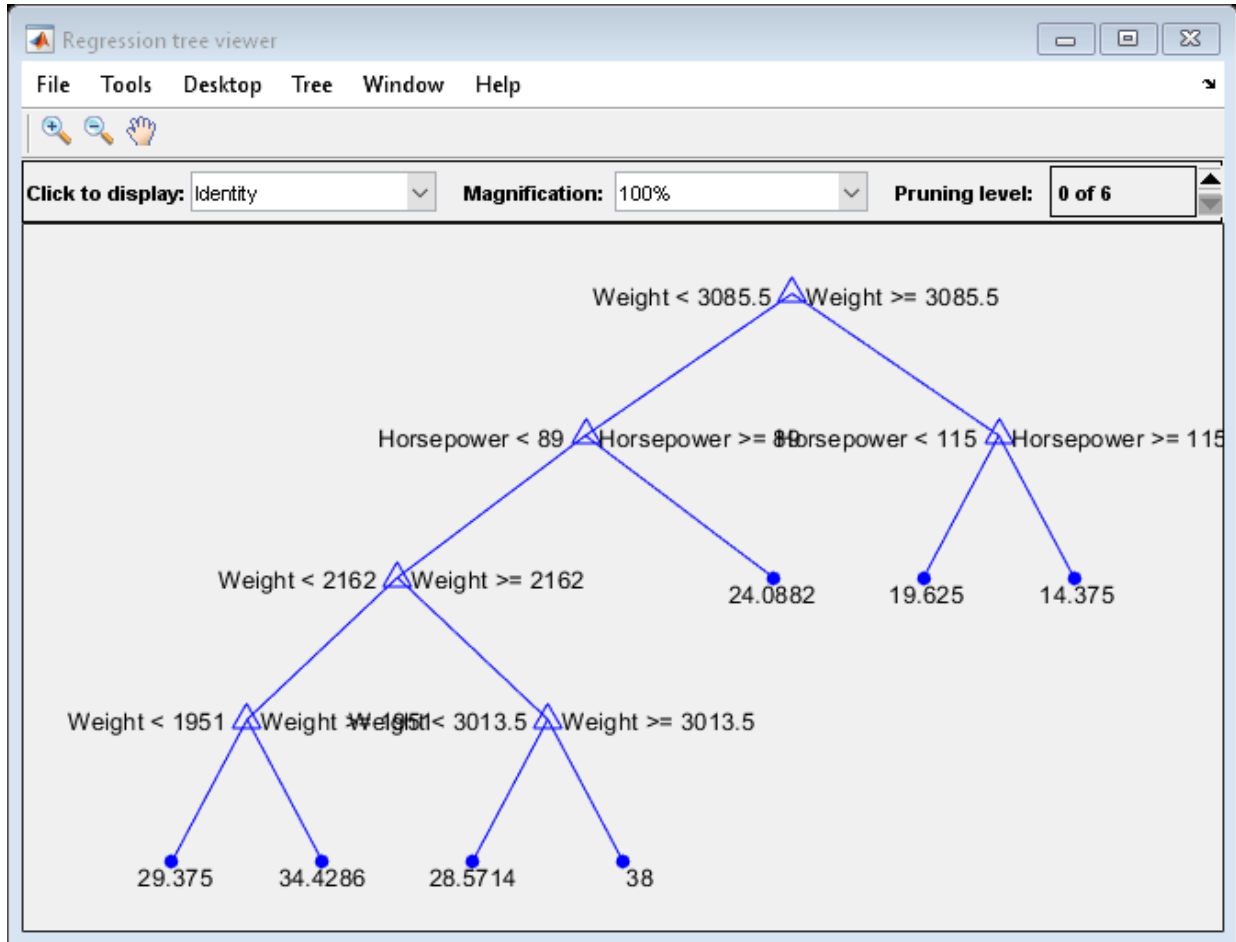
```



The regression tree has 16 pruning levels.

Prune the regression tree to pruning-level 10. View the pruned tree.

```
MdlPruned = prune(Mdl, 'Level', 10);
view(MdlPruned, 'Mode', 'graph');
```



The pruned tree has six pruning levels.

Alternatively, you can use the pruning-level field in the Regression tree viewer to prune the tree.

## Tips

- `tree1 = prune(tree)` returns the decision tree `tree1` that is the full, unpruned tree, but with optimal pruning information added. This is useful only if you created `tree` by pruning another tree, or by using `fitrtree` with pruning set 'off'. If you plan to prune a tree multiple times along the optimal pruning sequence, it is more efficient to create the optimal pruning sequence first.

## See Also

`fitrtree`

## Prune property

**Class:** `TreeBagger`

Flag to prune trees

## Description

The `Prune` property is true if decision trees are pruned and false if they are not. Pruning decision trees is not recommended for ensembles. The default value is false.

## See Also

`ClassificationTree` | `RegressionTree` | `TreeBagger` | `fitctree` | `fitrtree`

# prunelist

**Class:** `classregtree`

Pruning levels for decision tree nodes

---

**Note** `classregtree` will be removed in a future release. See `fitctree`, `fitrtree`, `ClassificationTree`, or `RegressionTree` instead.

---

## Syntax

```
P = prunelist(T)
P = prunelist(T,J)
```

## Description

`P = prunelist(T)` returns an  $n$ -element numeric vector with the pruning levels in each node of the tree `T`, where  $n$  is the number of nodes. When you call `prune(T, 'level', level)`, nodes with the pruning levels below `level` are pruned, and nodes with the pruning levels greater or equal to `level` are not pruned.

`P = prunelist(T,J)` takes an array `J` of node numbers and returns the pruning levels for the specified nodes.

## See Also

`classregtree` | `numnodes`

## qrand

**Class:** qrandstream

Generate quasi-random points from stream

## Syntax

```
x = qrand(q)
X = qrand(q, n)
```

## Description

`x = qrand(q)` returns the next value `x` in the quasi-random number stream `q` of the qrandstream on page 32-5248 class. `x` is a 1-by- $d$  vector, where  $d$  is the dimension of the stream. The command sets `q.State` to the index in the underlying point set of the next value to be returned.

`X = qrand(q, n)` returns the next `n` values `X` in an  $n$ -by- $d$  matrix.

Objects `q` of the qrandstream class encapsulate properties of a specified quasi-random number stream. Values of the stream are not generated and stored in memory until `q` is accessed using qrand.

## Examples

Use qrandstream to construct a 3-D Halton stream, based on a point set that skips the first 1000 values and then retains every 101st point:

```
q = qrandstream('halton', 3, 'Skip', 1e3, 'Leap', 1e2)
q =
    Halton quasi-random stream in 3 dimensions
    Point set properties:
        Skip : 1000
        Leap : 100
    ScrambleMethod : none
```



```
nextIdx = q.State
nextIdx =
    1
```

Use `grand` to generate two samples of size four:

```
X1 = grand(q,4)
X1 =
    0.0928    0.3475    0.0051
    0.6958    0.2035    0.2371
    0.3013    0.8496    0.4307
    0.9087    0.5629    0.6166
nextIdx = q.State
nextIdx =
    5
```

```
X2 = grand(q,4)
X2 =
    0.2446    0.0238    0.8102
    0.5298    0.7540    0.0438
    0.3843    0.5112    0.2758
    0.8335    0.2245    0.4694
nextIdx = q.State
nextIdx =
    9
```

Use `reset` to reset the stream, then generate another sample:

```
reset(q)
nextIdx = q.State
nextIdx =
    1

X = grand(q,4)
X =
    0.0928    0.3475    0.0051
    0.6958    0.2035    0.2371
    0.3013    0.8496    0.4307
    0.9087    0.5629    0.6166
```

## See Also

`grandstream` | `reset`

## grandset class

Quasi-random point sets

### Description

`grandset` is a base class that encapsulates a sequence of multi- dimensional quasi-random numbers. This base class is abstract and cannot be instantiated directly. Concrete subclasses include `sobolset` and `haltonset`.

### Construction

`grandset`          Abstract quasi-random point set class

### Methods

<code>disp</code>	Display grandset object
<code>end</code>	Last index in indexing expression for point set
<code>length</code>	Length of point set
<code>ndims</code>	Number of dimensions in matrix
<code>net</code>	Generate quasi-random point set
<code>scramble</code>	Scramble quasi-random point set
<code>size</code>	Number of dimensions in matrix
<code>suboref</code>	Subscripted reference for grandset

## Properties

Dimensions	Number of dimensions
Leap	Interval between points
ScrambleMethod	Settings that control scrambling
Skip	Number of initial points to omit from sequence
Type	Name of sequence on which point set $P$ is based

## Copy Semantics

Value. To learn how this affects your use of the class, see [Comparing Handle and Value Classes \(MATLAB\)](#) in the MATLAB Object-Oriented Programming documentation.

## See Also

`haltonset` | `sobolset`

## Topics

“Quasi-Random Point Sets” on page 7-19

# grandset

**Class:** grandset

Abstract quasi-random point set class

## Description

grandset is an abstract class, and you cannot create instances of it directly. You must use haltonset or sobolset to create a grandset object.

## See Also

haltonset | sobolset

**Introduced in R2009a**

## grandstream class

Quasi-random number streams

### Construction

`grandstream`      Construct quasi-random number stream

### Methods

<code>addlistener</code>	Add listener for event
<code>delete</code>	Delete handle object
<code>disp</code>	Display <code>grandstream</code> object
<code>eq</code>	Test handle equality
<code>findobj</code>	Find objects matching specified conditions
<code>findprop</code>	Find property of MATLAB handle object
<code>ge</code>	Greater than or equal relation for handles
<code>gt</code>	Greater than relation for handles
<code>isvalid</code>	Test handle validity
<code>le</code>	Less than or equal relation for handles
<code>lt</code>	Less than relation for handles
<code>ne</code>	Not equal relation for handles
<code>notify</code>	Notify listeners of event
<code>grand</code>	Generate quasi-random points from stream
<code>rand</code>	Generate quasi-random points from stream
<code>reset</code>	Reset state

## Properties

PointSet	Point set from which stream is drawn
State	Current state of the stream

## Copy Semantics

Handle. To learn how this affects your use of the class, see [Comparing Handle and Value Classes \(MATLAB\)](#) in the MATLAB Object-Oriented Programming documentation.

## grandstream

**Class:** grandstream

Construct quasi-random number stream

### Syntax

```
q = grandstream(type, d)  
q = grandstream(type, d, prop1, val1, prop2, val2, ...)   
q = grandstream(p)
```

### Description

`q = grandstream(type, d)` constructs a *d*-dimensional quasi-random number stream `q` of the grandstream on page 32-5248 class, of type specified by *type*. *type* is either 'halton' or 'sobol', and `q` is based on a point set from either the haltonset on page 32-2987 class or sobolset on page 32-6138 class, respectively, with default property settings.

`q = grandstream(type, d, prop1, val1, prop2, val2, ...)` specifies property name/value pairs for the point set on which the stream is based. Applicable properties depend on *type*.

`q = grandstream(p)` constructs a stream based on the specified point set *p*. *p* must be a point set from either the haltonset on page 32-2987 class or sobolset on page 32-6138 class.

### Examples

Construct a 3-D Halton stream, based on a point set that skips the first 1000 values and then retains every 101st point:

```
q = grandstream('halton', 3, 'Skip', 1e3, 'Leap', 1e2)  
q =
```



```
Halton quasi-random stream in 3 dimensions
Point set properties:
    Skip : 1000
    Leap : 100
    ScrambleMethod : none
```

```
nextIdx = q.State
nextIdx =
    1
```

Use `qrand` to generate two samples of size four:

```
X1 = qrand(q,4)
X1 =
    0.0928    0.3475    0.0051
    0.6958    0.2035    0.2371
    0.3013    0.8496    0.4307
    0.9087    0.5629    0.6166
nextIdx = q.State
nextIdx =
    5
```

```
X2 = qrand(q,4)
X2 =
    0.2446    0.0238    0.8102
    0.5298    0.7540    0.0438
    0.3843    0.5112    0.2758
    0.8335    0.2245    0.4694
nextIdx = q.State
nextIdx =
    9
```

Use `reset` to reset the stream, and then generate another sample:

```
reset(q)
nextIdx = q.State
nextIdx =
    1

X = qrand(q,4)
X =
    0.0928    0.3475    0.0051
    0.6958    0.2035    0.2371
    0.3013    0.8496    0.4307
    0.9087    0.5629    0.6166
```

## See Also

haltonset | grand | reset | sobolset

**Introduced in R2008a**

# qqplot

Quantile-quantile plot

## Syntax

```
qqplot(x)
```

```
qqplot(x, pd)
```

```
qqplot(x, y)
```

```
qqplot( ____, pvec)
```

```
h = qqplot( ____)
```

## Description

`qqplot(x)` displays a quantile-quantile plot of the quantiles of the sample data `x` versus the theoretical quantiles values from a normal distribution. If the distribution of `x` is normal, then the plot appears linear.

`qqplot(x, pd)` displays a quantile-quantile plot of the quantiles of the sample data `x` versus the theoretical quantiles of the distribution specified by the probability distribution object `pd`. If the distribution of `x` is the same as the distribution specified by `pd`, then the plot appears linear.

`qqplot(x, y)` displays a quantile-quantile plot of the quantiles of the sample data `x` versus the quantiles of the sample data `y`. If the samples come from the same distribution, then the plot appears linear.

`qqplot( ____, pvec)` displays a quantile-quantile plot with the quantiles specified in the vector `pvec`, using any of the previous syntaxes.

`h = qqplot( ____)` returns the handles (`h`) to the lines in the quantile-quantile plot.

## Examples

### Quantile-Quantile Plot for Normal Distribution

Use a quantile-quantile plot to determine whether gas prices in Massachusetts follow a normal distribution.

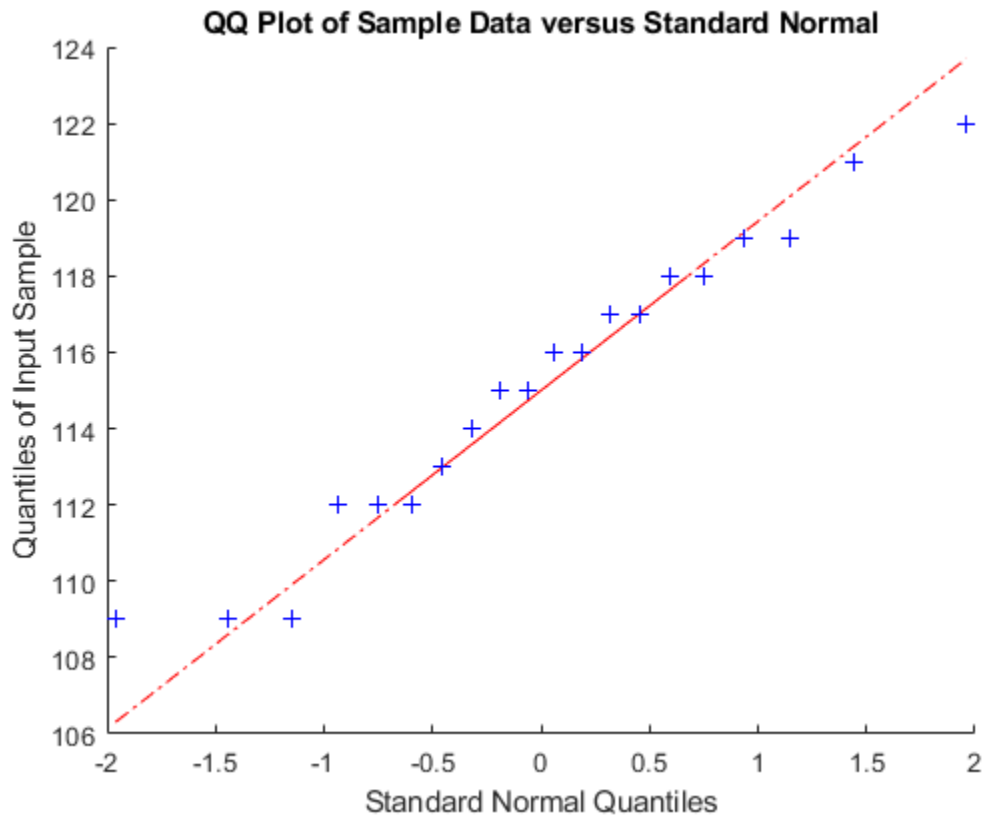
Load the sample data.

```
load gas
```

The sample data in `price1` and `price2` represent gasoline prices at 20 different gas stations in Massachusetts. The samples were collected during two different months.

Create a quantile-quantile plot to determine if the gas prices in `price1` follow a normal distribution.

```
figure  
qqplot(price1)
```



The plot produces an approximately straight line, suggesting that the gas prices follow a normal distribution.

### Quantile-Quantile Plot With Two Samples

Use a quantile-quantile plot to determine whether two sets of sample data come from the same distribution.

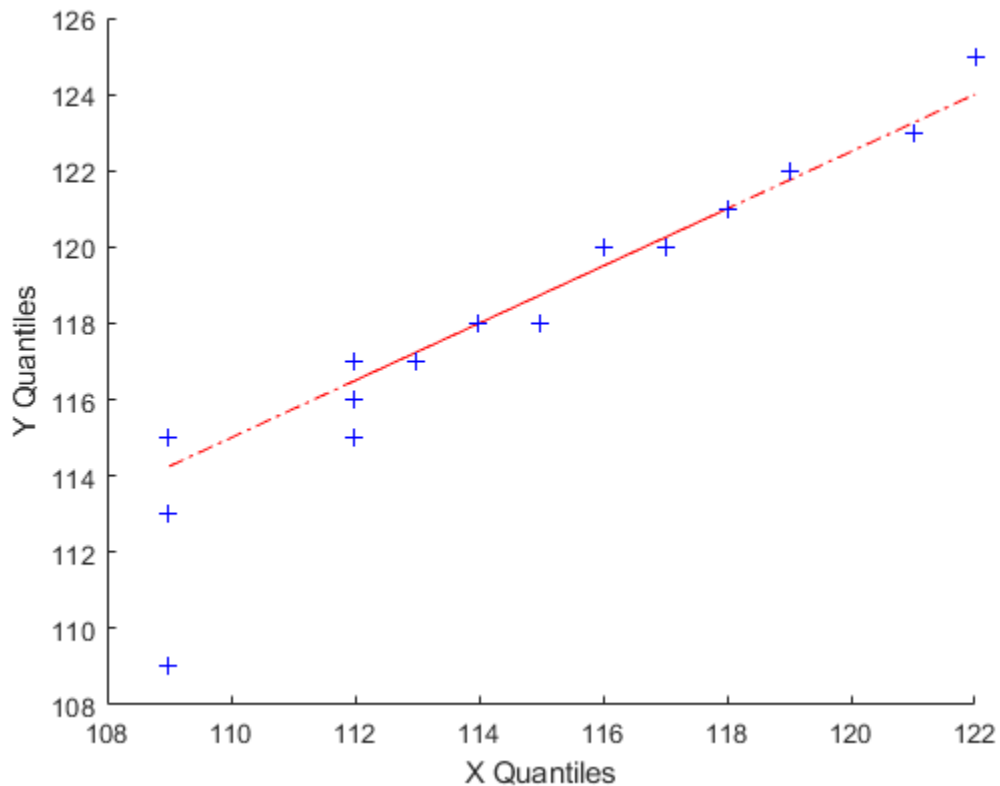
Load the sample data.

```
load gas
```

The sample data in `price1` and `price2` represent gasoline prices at 20 different gas stations in Massachusetts. The samples were collected during two different months.

Create a quantile-quantile plot using both sets of sample data, to assess whether prices at different times have the same distribution.

```
qqplot(price1,price2);
```



The plot produces an approximately straight line, suggesting that the two sets of sample data have the same distribution.

## Quantile-Quantile Plot for Weibull Distribution

Use a quantile-quantile plot to determine whether sample data comes from a Weibull distribution.

Load the sample data.

```
load(fullfile(matlabroot, 'examples', 'stats', 'lightbulb.mat'))
```

The first column of the data has the lifetime (in hours) of two types of lightbulbs. The second column has information about the type of light bulb. 1 indicates fluorescent bulbs whereas 0 indicates the incandescent bulb. The third column has censoring information. 1 indicates censored data, and 0 indicates the exact failure time. This is simulated data.

Remove the censored data.

```
lightbulb = [lightbulb(lightbulb(:,3)==0,1), ...  
            lightbulb(lightbulb(:,3)==0,2)];
```

Create a variable for each light bulb type. Include only uncensored data.

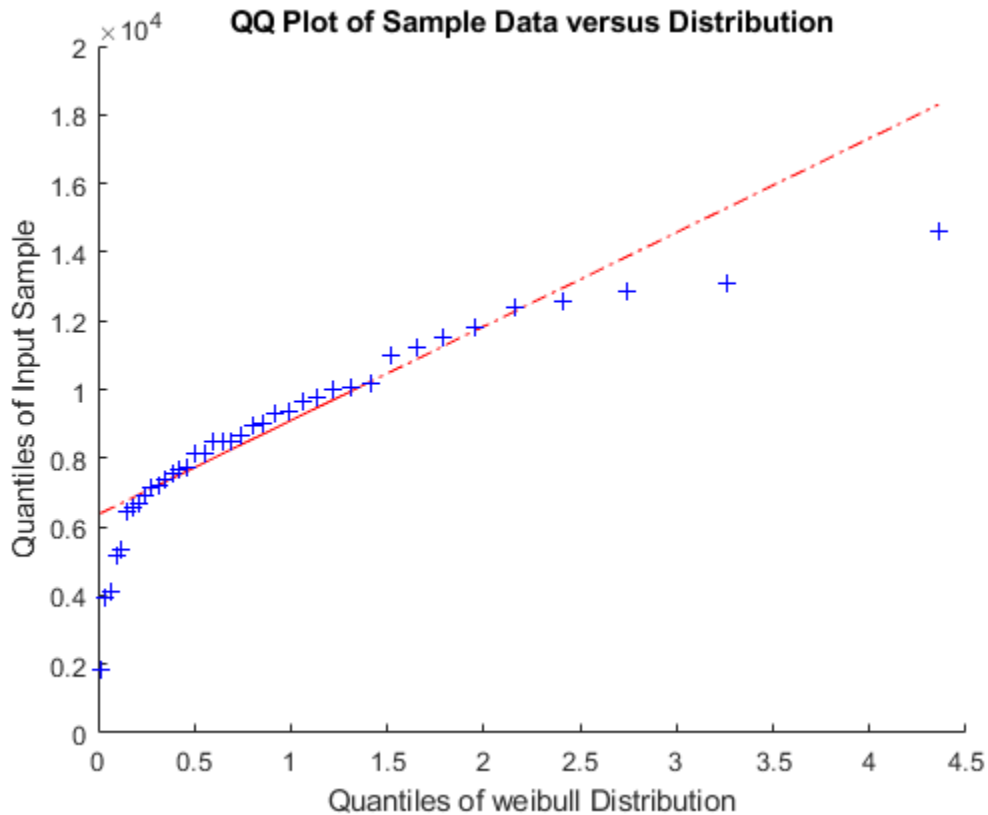
```
fluo = [lightbulb(lightbulb(:,2)==0,1)];  
insc = [lightbulb(lightbulb(:,2)==1,1)];
```

Create a Weibull probability distribution object using the default parameters of  $A = 1$  and  $B = 1$ .

```
pd = makedist('Weibull');
```

Create a q-q plot to determine whether the lifetime of fluorescent bulbs has a Weibull distribution.

```
figure  
qqplot(fluo,pd)
```



The plot is not a straight line, suggesting that the lifetime data for fluorescent bulbs does not follow a Weibull distribution.

## Input Arguments

### **x** — Sample data

numeric vector | numeric matrix

Sample data, specified as a numeric vector or numeric matrix. If **x** is a matrix, then `qqplot` displays a separate line for each column.



`qqplot` displays the sample data using the plot symbol '+' . A line joining the first and third quartiles of each distribution is superimposed on the plot. The line represents a robust linear fit of the order statistics for the data in `x`. This line is extrapolated out to the minimum and maximum values in `x` to help evaluate the linearity of the data.

Data Types: `single` | `double`

**`y` — Second set of sample data**

numeric vector | numeric matrix

Second set of sample data, specified as a numeric vector or numeric matrix. `x` and `y` do not need to be the same length. However, if `x` and `y` are matrices, they must contain the same number of columns. If `x` and `y` are matrices, then `qqplot` displays a separate line for each pair of columns.

`qqplot` selects the quantiles to plot based on the size of the smaller data set.

Data Types: `single` | `double`

**`pd` — Hypothesized probability distribution**

probability distribution object

Hypothesized probability distribution, specified as a probability distribution object. `qqplot` plots the quantiles of the input data `x` versus the theoretical quartiles of the distribution specified by `pd`.

Create a probability distribution object with specified parameter values using `makedist`, or fit a probability distribution object to data using `fitdist`.

**`pvec` — Quantiles for plot**

numeric value in the range [0,100] | vector of numeric values in the range [0,100]

Quantiles for plot, specified as a numeric value, or vector of numeric values, in the range [0,100].

For a single set of sample data (`x`), `qqplot` uses the quantiles in `x`. For two sets of sample data (`x` and `y`), `qqplot` uses the quantiles in the smaller of the two data sets.

Data Types: `single` | `double`

## Output Arguments

### **h** — Graphics handles for line objects

vector of `Line` graphic handles

Graphic handles for line objects, returned as a vector of `Line` graphic handles. Graphic handles are unique identifiers that you can use to query and modify the properties of a specific line on the plot. To view and set properties of line objects, use dot notation. For information on using dot notation, see “Access Property Values” (MATLAB). For information on the `Line` properties that you can set, see `Primitive Line`.

If the sample data is a vector, then `h` contains one handle for each of the three line objects created: One for the sample data, one for the fitted second and third quartile line, and one for the extrapolated first and fourth quartile line. If the sample data is a matrix, then `h` contains the three handles previously described for each of the three lines objects created. For example, if `x` is a two-column matrix, then `h` contains six line handles (three for each column of data).

## Definitions

### Quantile-Quantile Plot

A *quantile-quantile plot* (also called a *q-q plot*) visually assesses whether sample data comes from a specified distribution. Alternatively, a q-q plot assesses whether two sets of sample data come from the same distribution.

A q-q plot orders the sample data values from smallest to largest, then plots these values against the expected value for the specified distribution at each quantile in the sample data. The quantile values of the input sample appear along the *y*-axis, and the theoretical values of the specified distribution at the same quantiles appear along the *x*-axis. If the resulting plot is linear, then the sample data likely comes from the specified distribution.

The q-q plot selects quantiles based on the number of values in the sample data. If the sample data contains  $n$  values, then the plot uses  $n$  quantiles. Plot the  $i$ th ordered value

(also called the  $i$ th *order statistic*) against the  $\frac{i-0.5}{n}$  th quantile of the specified distribution.

A q-q plot can also assesses whether two sets of sample data have the same distribution, even if you do not know the underlying distribution. The quantile values for the first data set appear on the  $x$ -axis and the corresponding quantile values for the second data set appear on the  $y$ -axis. Since q-q plots rely on quantiles, the number of data points in the two samples does not need to be equal. If the sample sizes are unequal, the q-q plot chooses the quantiles based on the smaller data set. If the resulting plot is linear, then the two sets of sample data likely come from the same distribution.

## See Also

`normplot` | `probplot`

## Topics

“Distribution Plots” on page 4-10

**Introduced before R2006a**

## quantile

Quantiles of a data set

### Syntax

```
Y = quantile(X,p)
Y = quantile(X,p,dim)
```

```
Y = quantile(X,N)
Y = quantile(X,N,dim)
```

### Description

`Y = quantile(X,p)` returns quantiles of the values in data vector or matrix `X` for the cumulative probability or probabilities `p` in the interval  $[0,1]$ .

- If `X` is a vector, then `Y` is a scalar or a vector having the same length as `p`.
- If `X` is a matrix, then `Y` is a row vector or a matrix where the number of rows of `Y` is equal to the length of `p`.
- For multidimensional arrays on page 32-5270, `quantile` operates along the first nonsingleton dimension on page 32-5270 of `X`.

`Y = quantile(X,p,dim)` returns quantiles along dimension `dim`.

`Y = quantile(X,N)` returns quantiles for `N` evenly spaced cumulative probabilities  $(1/(N+1), 2/(N+1), \dots, N/(N+1))$  for integer  $N > 1$ .

- If `X` is a vector, then `Y` is a scalar or a vector with length `N`.
- If `X` is a matrix, then `Y` is a matrix where the number of rows of `Y` is equal to `N`.
- For multidimensional arrays on page 32-5270, `quantile` operates along the first nonsingleton dimension on page 32-5270 of `X`.

`Y = quantile(X,N,dim)` returns quantiles at the `N` evenly-spaced cumulative probabilities  $(1/(N+1), 2/(N+1), \dots, N/(N+1))$  for integer  $N > 1$  along dimension `dim`.

## Examples

### Quantiles for Given Probabilities

Calculate the quantiles of a data set for specified probabilities.

Generate a data set of size 10.

```
rng('default'); % for reproducibility
x = normrnd(0,1,1,10)
```

x =

Columns 1 through 7

```
    0.5377    1.8339   -2.2588    0.8622    0.3188   -1.3077   -0.4336
```

Columns 8 through 10

```
    0.3426    3.5784    2.7694
```

Calculate the 0.3 quantile.

```
y = quantile(x,0.30)
```

y =

```
-0.0574
```

Calculate the quantiles for the cumulative probabilities 0.025, 0.25, 0.5, 0.75, and 0.975.

```
y = quantile(x,[0.025 0.25 0.50 0.75 0.975])
```

y =

```
-2.2588   -0.4336    0.4401    1.8339    3.5784
```

### Quantiles of a Matrix for Given Probabilities

Calculate the quantiles along the columns and rows of a data matrix for specified probabilities.

Generate a 4-by-6 data matrix.

```
rng default % for reproducibility
X = normrnd(0,1,4,6)
```

X =

```
    0.5377    0.3188    3.5784    0.7254   -0.1241    0.6715
    1.8339   -1.3077    2.7694   -0.0631    1.4897   -1.2075
   -2.2588   -0.4336   -1.3499    0.7147    1.4090    0.7172
    0.8622    0.3426    3.0349   -0.2050    1.4172    1.6302
```

Calculate the 0.3 quantile for each column of X (dim = 1).

```
y = quantile(X,0.3,1)
```

y =

```
   -0.3013   -0.6958    1.5336   -0.1056    0.9491    0.1078
```

`quantile` returns a row vector `y` when calculating one quantile for each column of a matrix. For example, `-0.3013` is the 0.3 quantile of the first column of `X` with elements `(0.5377, 1.8339, -2.2588, 0.8622)`. `y = quantile(X,0.3)` returns the same answer because the default value of `dim` is 1.

Calculate the 0.3 quantile for each row of X (dim = 2).

```
y = quantile(X,0.3,2)
```

y =

```
    0.3844
   -0.8642
   -1.0750
```

```
0.4985
```

`quantile` returns a column vector `y` when calculating one quantile for each row of a matrix. For example 0.3844 is the 0.3 quantile of the first row of `|X|` with elements (0.5377, 0.3188, 3.5784, 0.7254, -0.1241, 0.6715).

### Quantiles for N Evenly Spaced Cumulative Probabilities

Calculate the quantiles of a data set for a given number of quantiles.

Generate a data set of size 10.

```
rng('default'); % for reproducibility
x = normrnd(0,1,1,10)
```

```
x =
```

```
Columns 1 through 7
```

```
0.5377    1.8339   -2.2588    0.8622    0.3188   -1.3077   -0.4336
```

```
Columns 8 through 10
```

```
0.3426    3.5784    2.7694
```

Calculate four evenly spaced quantiles.

```
y = quantile(x,4)
```

```
y =
```

```
-0.8706    0.3307    0.6999    2.3017
```

Using `y = quantile(x, [0.2, 0.4, 0.6, 0.8])` is another way to return the four evenly spaced quantiles.

### Quantiles of a Matrix for Given Number of Quantiles

Calculate the  $N$  evenly spaced quantiles along the columns and rows of a data matrix.

Generate a 6-by-10 data matrix.

```
rng('default'); % for reproducibility
X = unidrnd(10,6,7)
```

X =

```
     9     3    10     8     7     8     7
    10     6     5    10     8     1     4
     2    10     9     7     8     3    10
    10    10     2     1     4     1     1
     7     2     5     9     7     1     5
     1    10    10    10     2     9     4
```

Calculate three evenly spaced quantiles for each column of X (dim = 1).

```
y = quantile(X,3,1)
```

y =

```
    2.0000    3.0000    5.0000    7.0000    4.0000    1.0000    4.0000
    8.0000    8.0000    7.0000    8.5000    7.0000    2.0000    4.5000
   10.0000   10.0000   10.0000   10.0000    8.0000    8.0000    7.0000
```

Each column of matrix  $y$  corresponds to the three evenly spaced quantiles of each column of matrix  $X$ . For example, the first column of  $y$  with elements (2, 8, 10) has the quantiles for the first column of  $X$  with elements (9, 10, 2, 10, 7, 1).  $y = \text{quantile}(X, 3)$  returns the same answer because the default value of  $\text{dim}$  is 1.

Calculate three evenly spaced quantiles for each row of X (dim = 2).

```
y = quantile(X,3,2)
```

y =

```
    7.0000    8.0000    8.7500
```



```

4.2500    6.0000    9.5000
4.0000    8.0000    9.7500
1.0000    2.0000    8.5000
2.7500    5.0000    7.0000
2.5000    9.0000    10.0000

```

Each row of matrix  $y$  corresponds to the three evenly spaced quantiles of each row of matrix  $x$ . For example, the first row of  $y$  with elements (7, 8, 8.75) has the quantiles for the first column of  $x$  with elements (9, 3, 10, 8, 7, 8, 7).

### Median and Quartiles for Even Number of Data Elements

Find median and quartiles of a vector,  $x$ , with even number of elements.

Enter the data.

```
x = [2 5 6 10 11 13]
```

```
x =
```

```

     2     5     6    10    11    13

```

Calculate the median of  $x$ .

```
y = quantile(x,0.50)
```

```
y =
```

```

     8

```

Calculate the quartiles of  $x$ .

```
y = quantile(x,[0.25, 0.5, 0.75])
```

```
y =
```

```
5      8      11
```

Using `y = quantile(x, 3)` is another way to compute the quartiles of `x`.

These results might be different than the textbook definitions because `quantile` uses “Linear Interpolation” on page 32-5270 to find the median and quartiles.

### Median and Quartiles for Odd Number of Data Elements

Find median and quartiles of a vector, `x`, with odd number of elements.

Enter the data.

```
x = [2 4 6 8 10 12 14]
```

```
x =
```

```
2      4      6      8      10     12     14
```

Find the median of `x`.

```
y = quantile(x, 0.50)
```

```
y =
```

```
8
```

Find the quartiles of `x`.

```
y = quantile(x, [0.25, 0.5, 0.75])
```

```
y =
```

```
4.5000      8.0000      11.5000
```

Using `y = quantile(x, 3)` is another way to compute the quartiles of `x`.

These results might be different than the textbook definitions because `quantile` uses “Linear Interpolation” on page 32-5270 to find the median and quartiles.

## Input Arguments

### **x** — Input data

vector | array

Input data, specified as a vector or array.

Data Types: `double` | `single`

### **p** — Cumulative probabilities

scalar | vector

Cumulative probabilities, for which to compute the quantiles, specified as a scalar or vector of scalars from 0 to 1.

Example: 0.3

Example: [0.25, 0.5, 0.75]

Example: (0:0.25:1)

Data Types: `double` | `single`

### **n** — Number of quantiles

positive integer

Number of quantiles to compute, specified as a positive integer. `quantile` returns `N` quantiles that divide the data set into evenly distributed `N+1` segments.

Data Types: `double` | `single`

### **dim** — Dimension

1 (default) | positive integer

Dimension along which the quantiles of a matrix `X` are required, specified as a positive integer. For example, for a matrix `X`, when `dim = 1`, `quantile` returns the quantile(s) of the columns of `X` and when `dim = 2`, `quantile` returns the quantile(s) of the rows of `X`. For a multidimensional array `X`, the length of the `dim`th dimension of `Y` is same as length of `p`.

## Output Arguments

### **`Y`** — Quantiles

scalar | array

Quantiles of a data vector or matrix, returned as a scalar or array for one or multiple values of cumulative probabilities.

- If  $X$  is a vector, then  $Y$  is a scalar or a vector with the same length as the number of quantiles required (`N` or `length(p)`).  $Y(i)$  contains the  $p(i)$  quantile.
- If  $X$  is a matrix, then  $Y$  is a vector or a matrix with the length of `dimth` dimension equal to the number of quantiles required (`N` or `length(p)`). When `dim = 1`, for example, the  $i$ th row of  $Y$  contains the  $p(i)$  quantiles of columns of  $X$ .
- If  $X$  is an array of dimension  $d$ , then  $Y$  is an array with the length of `dimth` dimension equal to the number of quantiles required (`N` or `length(p)`).

## Definitions

### Multidimensional Array

A *multidimensional array* is an array with more than two dimensions. For example, if  $X$  is a 1-by-3-by-4 array, then  $X$  is a 3-D array.

### First Nonsingleton Dimension

A *first nonsingleton dimension* is the first dimension of an array whose size is not equal to 1. For example, if  $X$  is a 1-by-2-by-3-by-4 array, then the second dimension is the first nonsingleton dimension of  $X$ .

### Linear Interpolation

Linear interpolation uses linear polynomials to find  $y_i = f(x_i)$ , the values of the underlying function  $Y = f(X)$  at the points in the vector or array  $x$ . Given the data points  $(x_1, y_1)$  and  $(x_2, y_2)$ , where  $y_1 = f(x_1)$  and  $y_2 = f(x_2)$ , linear interpolation finds  $y = f(x)$  for a given  $x$  between  $x_1$  and  $x_2$  as follows:

$$y = f(x) = y_1 + \frac{(x - x_1)}{(x_2 - x_1)}(y_2 - y_1).$$

Similarly, if the  $1.5/n$  quantile is  $y_{1.5/n}$  and the  $2.5/n$  quantile is  $y_{2.5/n}$ , then linear interpolation finds the  $2.3/n$  quantile  $y_{2.3/n}$  as

$$y_{\frac{2.3}{n}} = y_{\frac{1.5}{n}} + \left( \frac{\frac{2.3}{n} - \frac{1.5}{n}}{\frac{2.5}{n} - \frac{1.5}{n}} \right) \left( y_{\frac{2.5}{n}} - y_{\frac{1.5}{n}} \right).$$

## Algorithms

For an  $n$ -element vector  $x$ , `quantile` computes quantiles as follows:

- 1 The sorted values in  $x$  are taken as the  $(0.5/n)$ ,  $(1.5/n)$ , ...,  $([n - 0.5]/n)$  quantiles. For example:
  - For a data vector of five elements such as {6, 3, 2, 10, 1}, the sorted elements {1, 2, 3, 6, 10} respectively correspond to the 0.1, 0.3, 0.5, 0.7, 0.9 quantiles.
  - For a data vector of six elements such as {6, 3, 2, 10, 8, 1}, the sorted elements {1, 2, 3, 6, 8, 10} respectively correspond to the  $(0.5/6)$ ,  $(1.5/6)$ ,  $(2.5/6)$ ,  $(3.5/6)$ ,  $(4.5/6)$ ,  $(5.5/6)$  quantiles.
- 2 `quantile` uses “Linear Interpolation” on page 32-5270 to compute quantiles for probabilities between  $(0.5/n)$  and  $([n - 0.5]/n)$ .
- 3 For the quantiles corresponding to the probabilities outside that range, `quantile` assigns the minimum or maximum values in  $x$ .

`quantile` treats NaNs as missing values and removes them.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

## See Also

`iqr` | `median` | `prctile`

## Topics

“Quantiles and Percentiles” on page 3-7

**Introduced before R2006a**

# rand

**Class:** grandstream

Generate quasi-random points from stream

## Syntax

```
rand
rand(q, n)
rand(q)
rand(q, m, n)
rand(q, [m, n])
rand(q, m, n, p, ...)
rand(q, [m, n, p, ...])
```

## Description

`rand` returns a matrix of quasi-random values and is intended to allow objects of the `grandstream` on page 32-5248 class to be used in code that contains calls to the `rand` method of the MATLAB pseudo-random `randstream` class. Due to the multidimensional nature of quasi-random numbers, only some syntaxes of `rand` are supported by the `grandstream` class.

`rand(q, n)` returns an  $n$ -by- $n$  matrix only when  $n$  is equal to the number of dimensions. Any other value of  $n$  produces an error.

`rand(q)` returns a scalar only when the stream is in one dimension. Having more than one dimension in  $q$  produces an error.

`rand(q, m, n)` or `rand(q, [m, n])` returns an  $m$ -by- $n$  matrix only when  $n$  is equal to the number of dimensions in the stream. Any other value of  $n$  produces an error.

`rand(q, m, n, p, ...)` or `rand(q, [m, n, p, ...])` produces an error unless  $p$  and all following dimensions sizes are equal to one.

## Examples

Generate the first 256 points from a 5-D Sobol sequence:

```
q = grandstream('sobol',5);  
X = rand(q,256,5);
```

## See Also

`grand` | `grandstream` | `rand`



# quantileError

**Class:** TreeBagger

Quantile loss using bag of regression trees

## Syntax

```
err = quantileError(Mdl,X)
err = quantileError(Mdl,X,ResponseVarName)
err = quantileError(Mdl,X,Y)
err = quantileError( ____,Name,Value)
```

## Description

`err = quantileError(Mdl,X)` returns half of the mean absolute deviation (MAD) from comparing the true responses in the table `X` to the predicted medians resulting from applying the bag of regression trees `Mdl` to the observations of the predictor data in `X`.

- `Mdl` must be a `TreeBagger` model object.
- The response variable name in `X` must have the same name as the response variable in the table containing the training data.

`err = quantileError(Mdl,X,ResponseVarName)` uses the true response and predictor variables contained in the table `X`. `ResponseVarName` is the name of the response variable and `Mdl.PredictorNames` contain the names of the predictor variables.

`err = quantileError(Mdl,X,Y)` uses the predictor data in the table or matrix `X` and the response data in the vector `Y`.

`err = quantileError( ____,Name,Value)` uses any of the previous syntaxes and additional options specified by one or more `Name,Value` pair arguments. For example, specify quantile probabilities, the error type, or which trees to include in the quantile-regression-error estimation.

## Input Arguments

### **mdl** — Bag of regression trees

`TreeBagger` model object (default)

Bag of regression trees, specified as a `TreeBagger` model object created by `TreeBagger`. The value of `Mdl.Method` must be `regression`.

### **x** — Sample data

numeric matrix | table

Sample data used to estimate quantiles, specified as a numeric matrix or table.

Each row of `X` corresponds to one observation, and each column corresponds to one variable. If you specify `Y`, then the number of rows in `X` must be equal to the length of `Y`.

- For a numeric matrix:
  - The variables making up the columns of `X` must have the same order as the predictor variables that trained `Mdl` (stored in `Mdl.PredictorNames`).
  - If you trained `Mdl` using a table (for example, `Tbl`), then `X` can be a numeric matrix if `Tbl` contains all numeric predictor variables. If `Tbl` contains heterogeneous predictor variables (for example, numeric and categorical data types), then `TreeBagger.quantileError` throws an error.
  - Specify `Y` for the true responses.
- For a table:
  - `TreeBagger.quantileError` does not support multi-column variables and cell arrays other than cell arrays of character vectors.
  - If you trained `Mdl` using a table (for example, `Tbl`), then all predictor variables in `X` must have the same variable names and data types as those variables that trained `Mdl` (stored in `Mdl.PredictorNames`). However, the column order of `X` does not need to correspond to the column order of `Tbl`. `Tbl` and `X` can contain additional variables (response variables, observation weights, etc.).
  - If you trained `Mdl` using a numeric matrix, then the predictor names in `Mdl.PredictorNames` and corresponding predictor variable names in `X` must be the same. To specify predictor names during training, see the `PredictorNames` name-value pair argument of `TreeBagger`. All predictor variables in `X` must be

numeric vectors.  $X$  can contain additional variables (response variables, observation weights, etc.).

- If  $X$  contains the response variable:
  - If the response variable has the same name as the response variable that trained `Mdl`, then you do not have to supply the response variable name or vector of true responses. `TreeBagger.quantileError` uses that variable for the true responses by default.
  - You can specify `ResponseVarName` or  $Y$  for the true responses.

Data Types: `table` | `double` | `single`

### **ResponseVarName** — Response variable name

character vector

Response variable name, specified as a character vector. `ResponseVarName` must be the name of the response variable in the table of sample data  $X$ .

If the table  $X$  contains the response variable, and it has the same name as the response variable used to train `Mdl`, then you do not have to specify `ResponseVarName`. `TreeBagger.quantileError` uses that variable for the true responses by default.

Data Types: `char`

### **Y** — True responses

numeric vector

True responses, specified as a character vector or numeric vector. The number of rows in  $X$  must be equal to the length of  $Y$ .

Data Types: `double` | `single`

## **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### **Mode** — Ensemble error type

'ensemble' (default) | 'cumulative' | 'individual'

Ensemble error type, specified as the comma-separated pair consisting of 'Mode' and a value in this table. Suppose  $\tau$  is the value of Quantile.

Value	Description
'cumulative'	<code>err</code> is a <code>Mdl.NumTrees-by-numel(<math>\tau</math>)</code> numeric matrix of cumulative quantile regression errors. <code>err(j,k)</code> is the $\tau(k)$ quantile regression error using the learners in <code>Mdl.Trees(1:j)</code> only.
'ensemble'	<code>err</code> is a <code>1-by-numel(<math>\tau</math>)</code> numeric vector of cumulative quantile regression errors for the entire ensemble. <code>err(k)</code> is the $\tau(k)$ ensemble quantile regression error.
'individual'	<code>err</code> is a <code>Mdl.NumTrees-by-numel(<math>\tau</math>)</code> numeric matrix of quantile regression errors from individual learners. <code>err(j,k)</code> is the $\tau(k)$ quantile regression error using the learner in <code>Mdl.Trees(j)</code> only.

For 'cumulative' and 'individual', if you include fewer trees in quantile estimation using `Trees` or `UseInstanceForTree`, then the number of rows in `err` decreases from `Mdl.NumTrees`.

Example: 'Mode', 'cumulative'

### Weights — Observation weights

`ones(size(X,1),1)` (default) | numeric vector of positive values

Observation weights, specified as the comma-separated pair consisting of 'Weights' and a numeric vector of positive values with length equal to `size(X,1)`.

`TreeBagger.quantileError` uses `Weights` to compute the weighted average of the deviations when estimating the quantile regression error.

By default, `TreeBagger.quantileError` attributes a weight of 1 to each observation, which yields an unweighted average of the deviations.

### Quantile — Quantile probability

0.5 (default) | numeric vector containing values in [0,1]

Quantile probability, specified as the comma-separated pair consisting of 'Quantile' and a numeric vector containing values in the interval [0,1]. For each element in `Quantile`, `TreeBagger.quantileError` returns corresponding quantile regression errors for all probabilities in `Quantile`.

Example: 'Quantile', [0 0.25 0.5 0.75 1]

Data Types: `single` | `double`

### **Trees** — Indices of trees to use in response estimation

'all' (default) | numeric vector of positive integers

Indices of trees to use in response estimation, specified as the comma-separated pair consisting of 'Trees' and 'all' or a numeric vector of positive integers. Indices correspond to the cells of `Mdl.Trees`; each cell therein contains a tree in the ensemble. The maximum value of `Trees` must be less than or equal to the number of trees in the ensemble (`Mdl.NumTrees`).

For 'all', `TreeBagger.quantileError` uses all trees in the ensemble (that is, the indices `1:Mdl.NumTrees`).

Values other than the default can affect the number of rows in `err`.

Example: 'Trees', [1 10 Mdl.NumTrees]

Data Types: `char` | `single` | `double`

### **TreeWeights** — Weights to attribute to responses from individual trees

`ones(Mdl.NumTrees,1)` (default) | numeric vector of nonnegative values

Weights to attribute to responses from individual trees, specified as the comma-separated pair consisting of 'TreeWeights' and a numeric vector of `numel(trees)` nonnegative values. `trees` is the value of `Trees`.

If you specify 'Mode', 'individual', then `TreeBagger.quantileError` ignores `TreeWeights`.

Data Types: `single` | `double`

### **UseInstanceForTree** — Indicators specifying which trees to use to make predictions for each observation

'all' (default) | logical matrix

Indicators specifying which trees to use to make predictions for each observation, specified as the comma-separated pair consisting of 'UseInstanceForTree' and an  $n$ -by-`Mdl.Trees` logical matrix.  $n$  is the number of observations (rows) in  $X$ . Rows of `UseInstanceForTree` correspond to observations and columns correspond to learners in `Mdl.Trees`. 'all' indicates to use all trees for all observations when estimating the quantiles.

If `UseInstanceForTree(j, k) = true`, then `TreeBagger.quantileError` uses the tree in `Mdl.Trees(k)` when it predicts the response for the observation `X(j, :)`.

You can estimate quantiles using the response data in `Mdl.Y` directly instead of using the predictions from the random forest by specifying a row composed entirely of `false` values. For example, to estimate the quantile for observation `j` using the response data, and to use the predictions from the random forest for all other observations, specify this matrix:

```
UseInstanceForTree = true(size(Mdl.X,2),Mdl.NumTrees);  
UseInstanceForTree(j,:) = false(1,Mdl.NumTrees);
```

Values other than the default can affect the number of rows in `err`. Also, the value of `Trees` affects the value of `UseInstanceForTree`. Suppose that `U` is the value of `UseInstanceForTree`. `TreeBagger.quantileError` ignores the columns of `U` corresponding to trees not being used in estimation from the specification of `Trees`. That is, `TreeBagger.quantileError` resets the value of `'UseInstanceForTree'` to `U(:, trees)`, where `trees` is the value of `'Trees'`.

Data Types: char | logical

## Output Arguments

### **err** — Half of quantile regression error

numeric scalar | numeric matrix

Half of the quantile regression error on page 32-5284, returned as a numeric scalar or  $T$ -by-`numel(tau)` matrix. `tau` is the value of `Quantile`.

$T$  depends on the values of `Mode`, `Trees`, `UseInstanceForTree`, and `Quantile`. Suppose that you specify `'Trees'`, `trees` and you use the default value of `'UseInstanceForTree'`.

- For `'Mode'`, `'cumulative'`, `err` is a `numel(trees)`-by-`numel(tau)` numeric matrix. `err(j, k)` is the `tau(k)` cumulative quantile regression error using the learners in `Mdl.Trees(trees(1:j))`.
- For `'Mode'`, `'ensemble'`, `err` is a 1-by-`numel(tau)` numeric vector. `err(k)` is the `tau(k)` cumulative quantile regression error using the learners in `Mdl.Trees(trees)`.

- For 'Mode', 'individual', `err` is a `numel(trees)`-by-`numel(tau)` numeric matrix. `err(j,k)` is the  $\tau(k)$  quantile regression error using the learner in `Mdl.Trees(trees(j))`.

## Examples

### Estimate In-Sample Quantile Regression Error

Load the `carsmall` data set. Consider a model that predicts the fuel economy of a car given its engine displacement, weight, and number of cylinders. Consider `Cylinders` a categorical variable.

```
load carsmall
Cylinders = categorical(Cylinders);
X = table(Displacement,Weight,Cylinders,MPG);
```

Train an ensemble of bagged regression trees using the entire data set. Specify 100 weak learners.

```
rng(1); % For reproducibility
Mdl = TreeBagger(100,X,'MPG','Method','regression');
```

`Mdl` is a `TreeBagger` ensemble.

Perform quantile regression, and estimate the MAD of the entire ensemble using the predicted conditional medians.

```
err = quantileError(Mdl,X)
```

```
err =
```

```
1.2586
```

Because `X` is a table containing the response and commensurate variable names, you do not have to specify the response variable name or data. However, you can specify the response using this syntax.

```
err = quantileError(Mdl,X,'MPG')
```

```
err =  
  
    1.2586
```

### Find Appropriate Ensemble Size Using Quantile Regression Error

Load the `carsmall` data set. Consider a model that predicts the fuel economy of a car given its engine displacement, weight, and number of cylinders.

```
load carsmall  
X = table(Displacement,Weight,Cylinders,MPG);
```

Randomly split the data into two sets: 75% training and 25% testing. Extract the subset indices.

```
rng(1); % For reproducibility  
cvp = cvpartition(size(X,1), 'Holdout', 0.25);  
idxTrn = training(cvp);  
idxTest = test(cvp);
```

Train an ensemble of bagged regression trees using the training set. Specify 250 weak learners.

```
Mdl = TreeBagger(250,X(idxTrn,:), 'MPG', 'Method', 'regression');
```

Estimate the cumulative 0.25, 0.5, and 0.75 quantile regression errors for the test set. Pass the predictor data in as a numeric matrix, and the response data in as a vector.

```
err = quantileError(Mdl,X{idxTest,1:3},MPG(idxTest), 'Quantile', [0.25 0.5 0.75], ...  
    'Mode', 'cumulative');
```

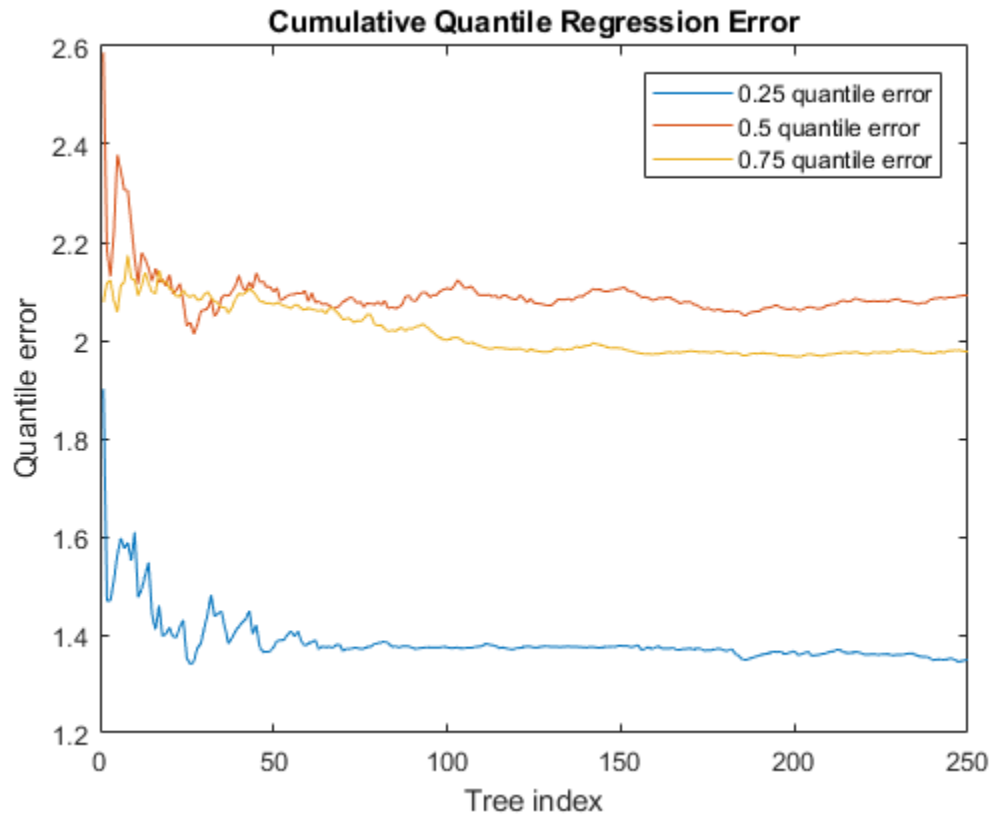
`err` is a 250-by-3 matrix of cumulative quantile regression errors. Columns correspond to quantile probabilities and rows correspond to trees in the ensemble. The errors are cumulative, so they incorporate aggregated predictions from previous trees. Although, `Mdl` was trained using a table, if all predictor variables in the table are numeric, then you can supply a matrix of predictor data instead.

Plot the cumulative quantile errors on the same plot.

```
figure;  
plot(err);
```



```
legend('0.25 quantile error', '0.5 quantile error', '0.75 quantile error');  
ylabel('Quantile error');  
xlabel('Tree index');  
title('Cumulative Quantile Regression Error')
```



Training using about 60 trees appears to be enough for the first two quartiles, but the third quartile requires about 150 trees.

## Definitions

### Quantile Regression Error

The quantile regression error of a model given observed predictor data and responses is the weighted mean absolute deviation (MAD). If the model under-predicts the response, then deviation weights are  $\tau$ , the quantile probability. If the model over-predicts, then deviation weights are  $1 - \tau$ .

That is, the  $\tau$  quantile regression error is

$$L_\tau = \tau \frac{\sum_{\{j: y_j \geq \hat{y}_{\tau,j}\}} w_j (y_j - \hat{y}_{\tau,j})}{\sum_{j=1}^n w_j} + (1 - \tau) \frac{\sum_{\{j: y_j < \hat{y}_{\tau,j}\}} w_j (\hat{y}_{\tau,j} - y_j)}{\sum_{j=1}^n w_j}.$$

$y_j$  is true response  $j$ ,  $\hat{y}_{\tau,j}$  is the  $\tau$  quantile that the model predicts, and  $w_j$  is observation weight  $j$ .

## Tips

- To tune the number of trees in the ensemble, set 'Mode', 'cumulative' and plot the quantile regression errors with respect to tree indices. The maximal number of required trees is the tree index where the quantile regression error appears to level off.
- To investigate the performance of a model when the training sample is small, use `oobQuantileError` instead.

## References

[1] Breiman, L. *Random Forests*. Machine Learning 45, pp. 5–32, 2001.

[2] Meinshausen, N. “Quantile Regression Forests.” *Journal of Machine Learning Research*, Vol. 7, 2006, pp. 983–999.

## See Also

`TreeBagger` | `error` | `oobQuantileError` | `quantilePredict`

**Introduced in R2016b**

## quantilePredict

**Class:** TreeBagger

Predict response quantile using bag of regression trees

### Syntax

```
YFit = quantilePredict(Mdl,X)
YFit = quantilePredict(Mdl,X,Name,Value)
[YFit,YW] = quantilePredict(____)
```

### Description

`YFit = quantilePredict(Mdl,X)` returns a vector of medians of the predicted responses at `X`, a table or matrix of predictor data, and using the bag of regression trees `Mdl`. `Mdl` must be a `TreeBagger` model object.

`YFit = quantilePredict(Mdl,X,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments. For example, specify quantile probabilities or which trees to include for quantile estimation.

`[YFit,YW] = quantilePredict(____)` also returns a sparse matrix of response weights on page 32-5297.

### Input Arguments

**mdl** — Bag of regression trees

`TreeBagger` model object (default)

Bag of regression trees, specified as a `TreeBagger` model object created by `TreeBagger`. The value of `Mdl.Method` must be `regression`.

**x** — Predictor data

numeric matrix | table

Predictor data used to estimate quantiles, specified as a numeric matrix or table.

Each row of `X` corresponds to one observation, and each column corresponds to one variable.

- For a numeric matrix:
  - The variables making up the columns of `X` must have the same order as the predictor variables that trained `Mdl`.
  - If you trained `Mdl` using a table (for example, `Tbl`), then `X` can be a numeric matrix if `Tbl` contains all numeric predictor variables. If `Tbl` contains heterogeneous predictor variables (for example, numeric and categorical data types) and `X` is a numeric matrix, then `TreeBagger.quantilePredict` throws an error.
- For a table:
  - `quantilePredict` does not support multi-column variables and cell arrays other than cell arrays of character vectors.
  - If you trained `Mdl` using a table (for example, `Tbl`), then all predictor variables in `X` must have the same variable names and data types as those variables that trained `Mdl` (stored in `Mdl.PredictorNames`). However, the column order of `X` does not need to correspond to the column order of `Tbl`. `Tbl` and `X` can contain additional variables (response variables, observation weights, etc.), but `TreeBagger.quantilePredict` ignores them.
  - If you trained `Mdl` using a numeric matrix, then the predictor names in `Mdl.PredictorNames` and corresponding predictor variable names in `X` must be the same. To specify predictor names during training, see the `PredictorNames` name-value pair argument of `TreeBagger`. All predictor variables in `X` must be numeric vectors. `X` can contain additional variables (response variables, observation weights, etc.), but `quantilePredict` ignores them.

Data Types: `table` | `double` | `single`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

**Quantile — Quantile probability**

0.5 (default) | numeric vector containing values in [0,1]

Quantile probability, specified as the comma-separated pair consisting of 'Quantile' and a numeric vector containing values in the interval [0,1]. For each observation (row) in `X`, `TreeBagger.quantilePredict` returns corresponding quantiles for all probabilities in `Quantile`.

Example: 'Quantile', [0 0.25 0.5 0.75 1]

Data Types: single | double

**Trees — Indices of trees to use in response estimation**

'all' (default) | numeric vector of positive integers

Indices of trees to use in response estimation, specified as the comma-separated pair consisting of 'Trees' and 'all' or a numeric vector of positive integers. Indices correspond to the cells of `Mdl.Trees`; each cell therein contains a tree in the ensemble. The maximum value of `Trees` must be less than or equal to the number of trees in the ensemble (`Mdl.NumTrees`).

For 'all', `TreeBagger.quantilePredict` uses the indices `1:Mdl.NumTrees`.

Example: 'Trees', [1 10 Mdl.NumTrees]

Data Types: char | single | double

**TreeWeights — Weights to attribute to responses from individual trees**

numeric vector of nonnegative values

Weights to attribute to responses from individual trees, specified as the comma-separated pair consisting of 'TreeWeights' and a numeric vector of `numel(trees)` nonnegative values. `trees` is the value of the `Trees` name-value pair argument.

The default is `ones(size(trees))`.

Data Types: single | double

**UseInstanceForTree — Indicators specifying which trees to use to make predictions for each observation**

'all' (default) | logical matrix

Indicators specifying which trees to use to make predictions for each observation, specified as the comma-separated pair consisting of 'UseInstanceForTree' and an  $n$ -

by-Mdl.Trees logical matrix.  $n$  is the number of observations (rows) in  $X$ . Rows of UseInstanceForTree correspond to observations and columns correspond to learners in Mdl.Trees. 'all' indicates to use all trees for all observations when estimating the quantiles.

If UseInstanceForTree( $j, k$ ) = true, then TreeBagger.quantilePredict uses the tree in Mdl.Trees(trees( $k$ )) when it predicts the response for the observation  $X(j, :)$ .

You can estimate the quantile using the response data in Mdl.Y directly instead of using the predictions from the random forest by specifying a row composed entirely of false values. For example, to estimate the quantile for observation  $j$  using the response data, and to use the predictions from the random forest for all other observations, specify this matrix:

```
UseInstanceForTree = true(size(Mdl.X,2),Mdl.NumTrees);
UseInstanceForTree(j,:) = false(1,Mdl.NumTrees);
```

Data Types: char | logical

## Output Arguments

### **YFit** — Estimated quantiles

numeric matrix

Estimated quantiles, returned as an  $n$ -by- $\text{numel}(\tau)$  numeric matrix.  $n$  is the number of observations in  $X$  ( $\text{size}(X, 1)$ ) and  $\tau$  is the value of Quantile. That is, YFit( $j, k$ ) is the estimated  $100 * \tau(k) \%$  percentile of the response distribution given  $X(j, :)$  and using Mdl.

### **yw** — Response weights

sparse matrix

Response weights on page 32-5297, returned as an  $n_{\text{train}}$ -by- $n$  sparse matrix.  $n_{\text{train}}$  is the number of responses in the training data ( $\text{numel}(\text{Mdl.Y})$ ) and  $n$  is the number of observations in  $X$  ( $\text{size}(X, 1)$ ).

TreeBagger.quantilePredict predicts quantiles using linear interpolation of the empirical cumulative distribution function (C.D.F.). For a particular observation, you can

use its response weights to estimate quantiles using alternative methods, such as approximating the C.D.F. using kernel smoothing.

---

**Note** `TreeBagger.quantilePredict` derives response weights by passing an observation through the trees in the ensemble. If you specify `UseInstanceForTree` and you compose row  $j$  entirely of false values, then  $YW(:, j) = Mdl.W$  instead, that is, the observation weights.

---

## Examples

### Predict Training Sample Medians

Load the `carsmall` data set. Consider a model that predicts the fuel economy of a car given its engine displacement.

```
load carsmall
```

Train an ensemble of bagged regression trees using the entire data set. Specify 100 weak learners.

```
rng(1); % For reproducibility
Mdl = TreeBagger(100, Displacement, MPG, 'Method', 'regression');
```

`Mdl` is a `TreeBagger` ensemble.

Perform quantile regression to predict the median MPG for all sorted training observations.

```
medianMPG = quantilePredict(Mdl, sort(Displacement));
```

`medianMPG` is an  $n$ -by-1 numeric vector of medians corresponding to the conditional distribution of the response given the sorted observations in `Displacement`.  $n$  is the number of observations in `Displacement`.

Plot the observations and the estimated medians on the same figure. Compare the median and mean responses.

```
meanMPG = predict(Mdl, sort(Displacement));
```



```
figure;  
plot(Displacement,MPG,'k.');
```

---

```
hold on  
plot(sort(Displacement),medianMPG);  
plot(sort(Displacement),meanMPG,'r--');
```

---

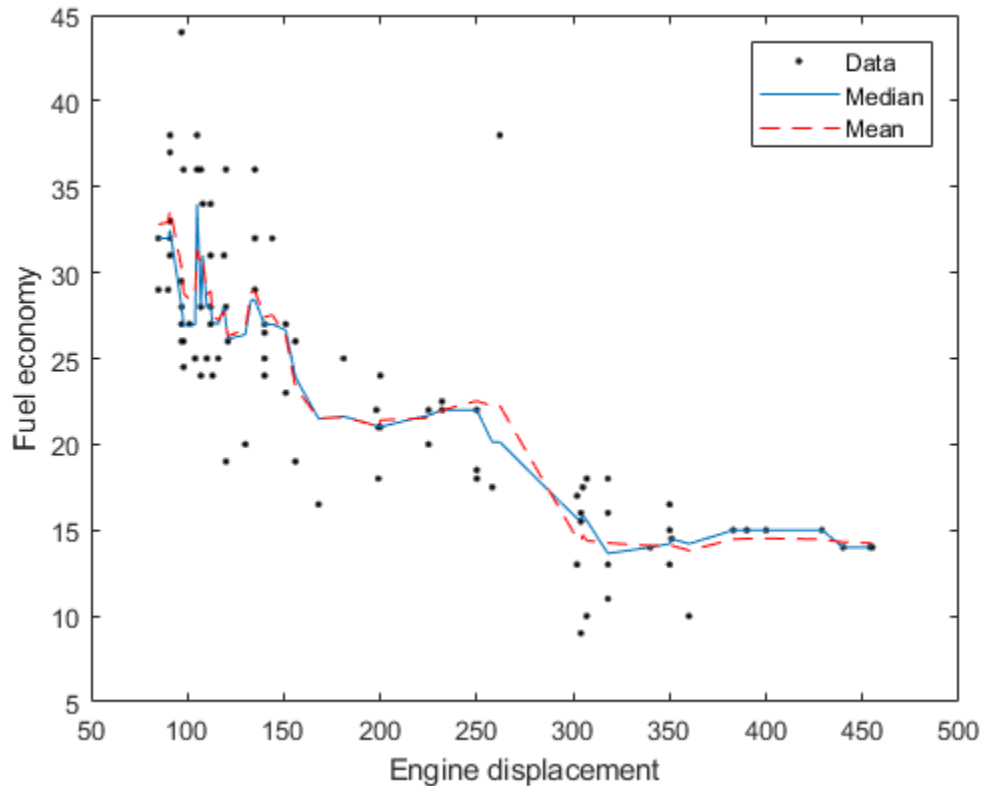
```
ylabel('Fuel economy');  
xlabel('Engine displacement');
```

---

```
legend('Data','Median','Mean');
```

---

```
hold off;
```



### Estimate Prediction Intervals Using Percentiles

Load the `carsmall` data set. Consider a model that predicts the fuel economy of a car given its engine displacement.

```
load carsmall
```

Train an ensemble of bagged regression trees using the entire data set. Specify 100 weak learners.

```
rng(1); % For reproducibility
Mdl = TreeBagger(100, Displacement, MPG, 'Method', 'regression');
```

Perform quantile regression to predict the 2.5% and 97.5% percentiles for ten equally-spaced engine displacements between the minimum and maximum in-sample displacement.

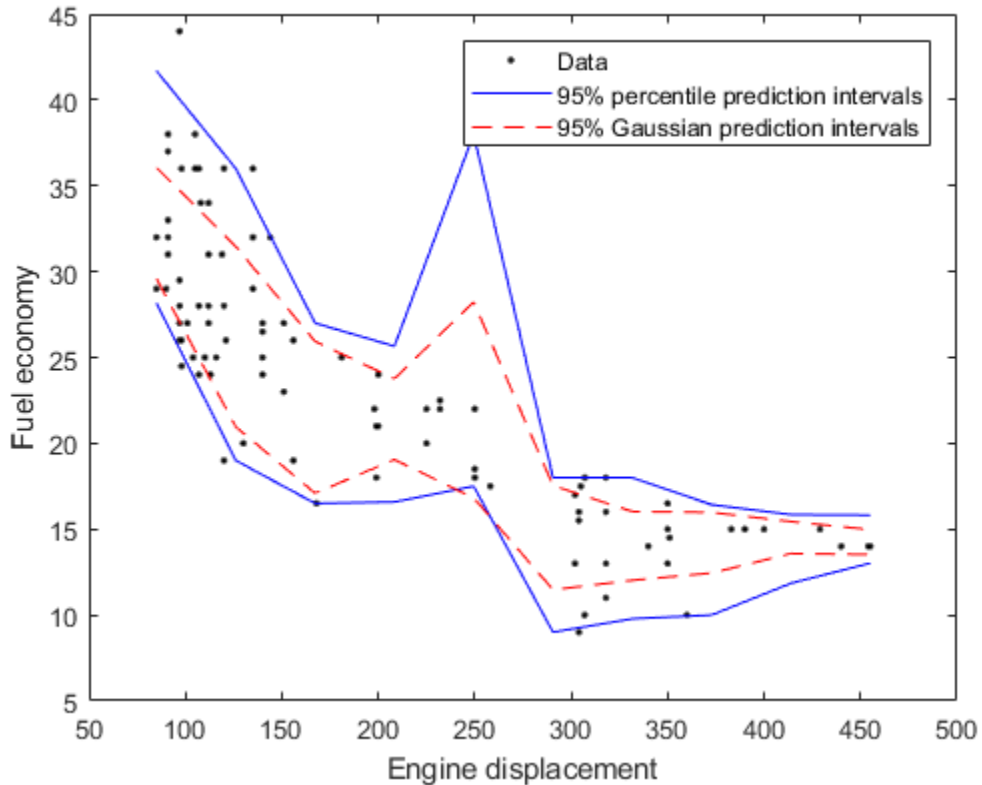
```
predX = linspace(min(Displacement), max(Displacement), 10)';
quantPredInts = quantilePredict(Mdl, predX, 'Quantile', [0.025, 0.975]);
```

`quantPredInts` is a 10-by-2 numeric matrix of prediction intervals corresponding to the observations in `predX`. The first column contains the 2.5% percentiles and the second column contains the 97.5% percentiles.

Plot the observations and the estimated medians on the same figure. Compare the percentile prediction intervals and the 95% prediction intervals assuming the conditional distribution of MPG is Gaussian.

```
[meanMPG, steMeanMPG] = predict(Mdl, predX);
stndPredInts = meanMPG + [-1 1]*norminv(0.975).*steMeanMPG;

figure;
h1 = plot(Displacement, MPG, 'k. ');
hold on
h2 = plot(predX, quantPredInts, 'b');
h3 = plot(predX, stndPredInts, 'r--');
ylabel('Fuel economy');
xlabel('Engine displacement');
legend([h1, h2(1), h3(1)], {'Data', '95% percentile prediction intervals', ...
    '95% Gaussian prediction intervals'});
hold off;
```



### Estimate Conditional Cumulative Distribution Using Quantile Regression

Load the `carsmall` data set. Consider a model that predicts the fuel economy of a car given its engine displacement.

```
load carsmall
```

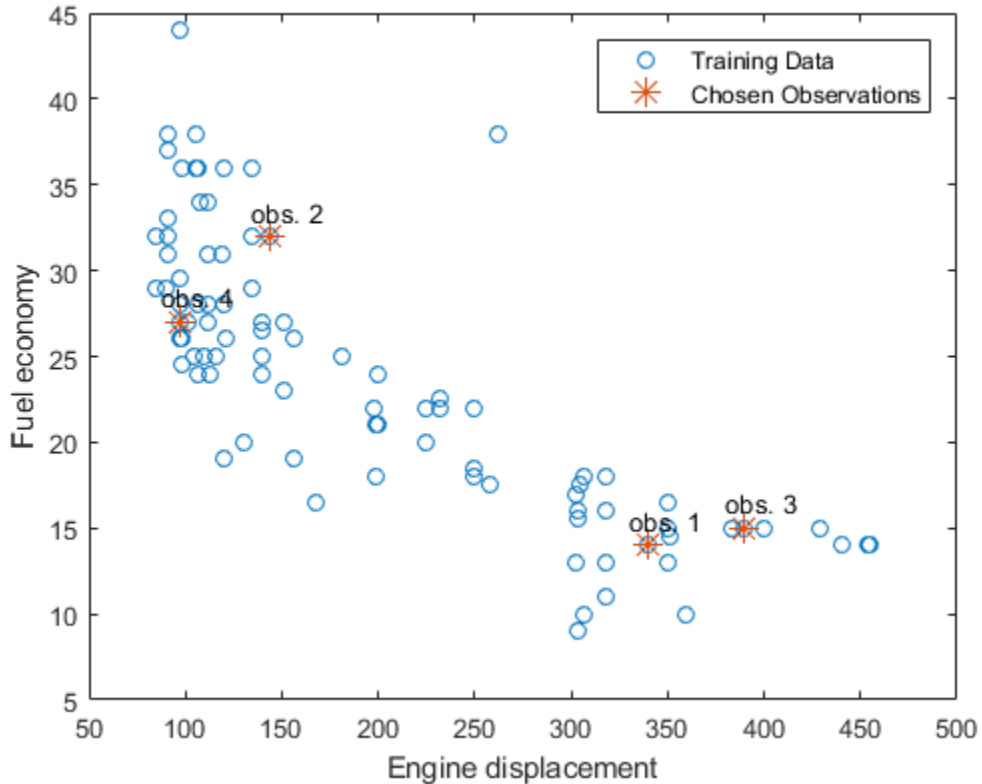
Train an ensemble of bagged regression trees using the entire data set. Specify 100 weak learners.

```
rng(1); % For reproducibility  
Mdl = TreeBagger(100, Displacement, MPG, 'Method', 'regression');
```

Estimate the response weights for a random sample of four training observations. Plot the training sample and identify the chosen observations.

```
[predX,idx] = datasample(Mdl.X,4);
[~,YW] = quantilePredict(Mdl,predX);
n = numel(Mdl.Y);

figure;
plot(Mdl.X,Mdl.Y,'o');
hold on
plot(predX,Mdl.Y(idx),'*', 'MarkerSize',10);
text(predX-10,Mdl.Y(idx)+1.5,{'obs. 1' 'obs. 2' 'obs. 3' 'obs. 4'});
legend('Training Data','Chosen Observations');
xlabel('Engine displacement')
ylabel('Fuel economy')
hold off
```



$YW$  is an  $n$ -by-4 sparse matrix containing the response weights. Columns correspond to test observations and rows correspond to responses in the training sample. Response weights are independent of the specified quantile probability.

Estimate the conditional cumulative distribution function (C.C.D.F.) of the responses by:

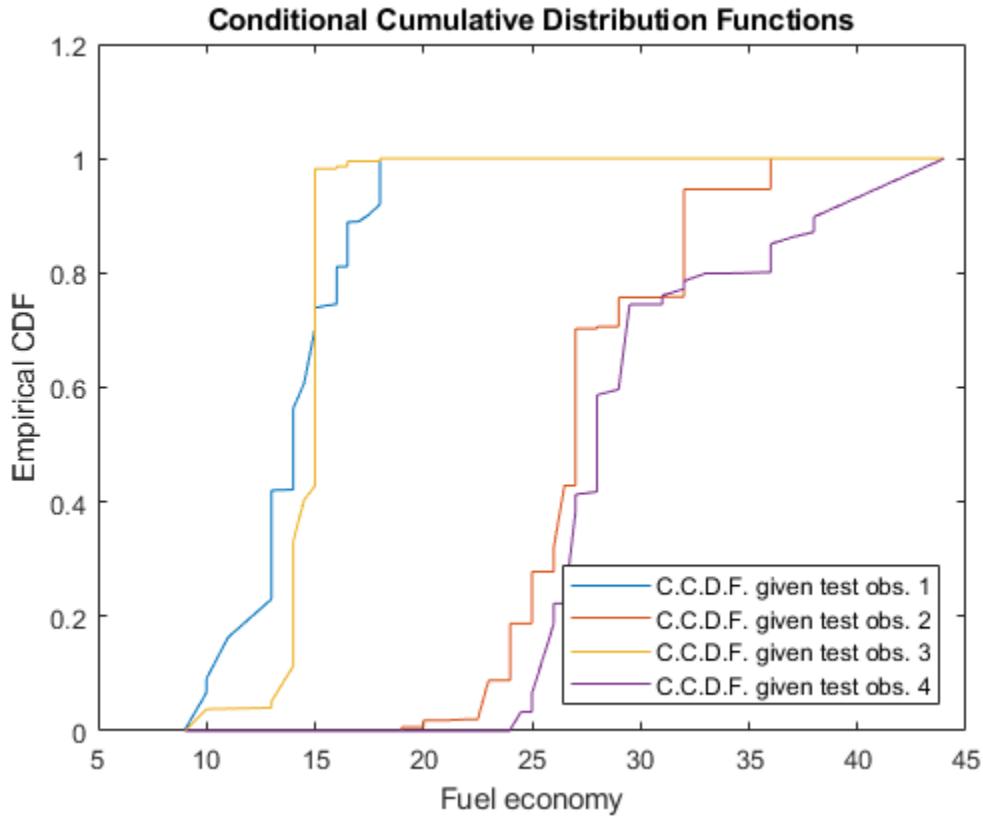
- 1 Sorting the responses in ascending order, and then sorting the response weights using the indices induced by sorting the responses.
- 2 Computing the cumulative sums over each column of the sorted response weights.

```
[sortY,sortIdx] = sort(Mdl.Y);
cpdf = full(YW(sortIdx,:));
ccdf = cumsum(cpdf);
```

`ccdf(:, j)` is the empirical C.C.D.F. of the response given test observation  $j$ .

Plot the four empirical C.C.D.F. in the same figure.

```
figure;
plot(sortY,ccdf);
legend('C.C.D.F. given test obs. 1','C.C.D.F. given test obs. 2',...
       'C.C.D.F. given test obs. 3','C.C.D.F. given test obs. 4',...
       'Location','SouthEast')
title('Conditional Cumulative Distribution Functions')
xlabel('Fuel economy')
ylabel('Empirical CDF')
```



- “Detect Outliers Using Quantile Regression” on page 18-170

## Definitions

### Response Weights

Response weights are scalars that represent the conditional distribution of the response given a value in the predictor space. The observations in the bootstrap samples and the leaves that the training and test observations share induce response weights.

Given the observation  $x$ , the response weight for observation  $j$  in the training sample using tree  $t$  in the ensemble is

$$w_{tj}(x) = \frac{I\{X_j \in S_t(x)\}}{\sum_{k=1}^{n_{\text{train}}} I\{X_k \in S_t(x)\}},$$

where:

- $I\{h\}$  is the indicator function.
- $S_t(x)$  is the leaf of tree  $t$  containing  $x$ .
- $n_{\text{train}}$  is the number of training observations.

In other words, the response weights of a particular tree form the conditional relative frequency distribution of the response.

The response weights for the entire ensemble are averaged over the trees:

$$w_j^*(x) = \frac{1}{T} \sum_{t=1}^T w_{tj}(x).$$

### Quantile Random Forest

Quantile random forest [2] is a quantile-regression method that uses a random forest [1] of regression trees to model the conditional distribution of a response variable, given the value of predictor variables. You can use a fitted model to estimate quantiles in the conditional distribution of the response.

Besides quantile estimation, you can use quantile regression to estimate prediction intervals or detect outliers. For example:

- To estimate 95% quantile prediction intervals, estimate the 0.025 and 0.975 quantiles.
- To detect outliers, estimate the 0.01 and 0.99 quantiles. All observations smaller than the 0.01 quantile and larger than the 0.99 quantile are outliers. All observations that are outside the interval  $[L, U]$  can be considered outliers:

$$L = Q_1 - 1.5 * IQR$$

and

$$U = Q_3 + 1.5 * IQR,$$

where:

- $Q_1$  is the 0.25 quantile.
- $Q_3$  is the 0.75 quantile.
- $IQR = Q_3 - Q_1$  (the interquartile range).

## Tip

`TreeBagger.quantilePredict` estimates the conditional distribution of the response using the training data every time you call it. To predict many quantiles efficiently, or quantiles for many observations efficiently, you should pass `X` as a matrix or table of observations and specify all quantiles in a vector using the `Quantile` name-value pair argument. That is, avoid calling `TreeBagger.quantilePredict` within a loop.

## Algorithms

- `TreeBagger` grows a random forest of regression trees using the training data. Then, to implement quantile random forest on page 32-5297, `TreeBagger.quantilePredict` predicts quantiles using the empirical conditional distribution of the response given an observation from the predictor variables. To obtain the empirical conditional distribution of the response:
  - 1 `TreeBagger.quantilePredict` passes all the training observations in `Mdl.X` through all the trees in the ensemble, and stores the leaf nodes of which the training observations are members.
  - 2 `TreeBagger.quantilePredict` similarly passes each observation in `X` through all the trees in the ensemble.



- 3 For each observation in  $X$ , `TreeBagger.quantilePredict`:
  - a Estimates the conditional distribution of the response by computing response weights on page 32-5297 for each tree.
  - b For observation  $k$  in  $X$ , aggregates the conditional distributions for the entire ensemble:

$$\hat{F}(y | X = x_k) = \sum_{j=1}^n \sum_{t=1}^T \frac{1}{T} w_{tj}(x_k) I\{Y_j \leq y\}.$$

$n$  is the number of training observations (`size(Y, 1)`) and  $T$  is the number of trees in the ensemble (`Mdl.NumTrees`).

- 4 For observation  $k$  in  $X$ , the  $\tau$  quantile or, equivalently, the  $100\tau\%$  percentile, is

$$Q_{\tau}(x_k) = \inf\{y : F(y | X = x_k) \geq \tau\}.$$

- This process describes how `TreeBagger.quantilePredict` uses all specified weights.

- 1 For all training observations  $j = 1, \dots, n$  and all chosen trees  $t = 1, \dots, T$ ,

`TreeBagger.quantilePredict` attributes the product  $v_{tj} = b_{tj}w_{j,\text{obs}}$  to training observation  $j$  (stored in `Mdl.X(j, :)` and `Mdl.Y(j)`).  $b_{tj}$  is the number of times observation  $j$  is in the bootstrap sample for tree  $t$ .  $w_{j,\text{obs}}$  is the observation weight in `Mdl.W(j)`.

- 2 For each chosen tree, `TreeBagger.quantilePredict` identifies the leaves in which each training observation falls. Let  $S_t(x_j)$  be the set of all observations contained in the leaf of tree  $t$  of which observation  $j$  is a member.
- 3 For each chosen tree, `TreeBagger.quantilePredict` normalizes all weights within a particular leaf to sum to 1, that is,

$$v_{tj}^* = \frac{v_{tj}}{\sum_{i \in S_t(x_j)} v_{ti}}.$$

- 4 For each training observation and tree, `TreeBagger.quantilePredict` incorporates tree weights ( $w_{t,\text{tree}}$ ) specified by `TreeWeights`, that is,  $w_{t,\text{tree}}^* v_{tj}^* = w_{t,\text{tree}} v_{tj}^*$ . Trees not chosen for prediction have 0 weight.
- 5 For all test observations  $k = 1, \dots, K$  in  $X$  and all chosen trees  $t = 1, \dots, T$ , `TreeBagger.quantilePredict` predicts the unique leaves in which the

observations fall, and then identifies all training observations within the predicted leaves. `TreeBagger.quantilePredict` attributes the weight  $u_{ij}$  such that

$$u_{ij} = \begin{cases} w_{ij,\text{tree}}^* & \text{if } x_k \in S_t(x_j) \\ 0 & \text{otherwise} \end{cases}.$$

- 6 `TreeBagger.quantilePredict` sums the weights over all chosen trees, that is,

$$u_j = \sum_{t=1}^T u_{tj}.$$

- 7 `TreeBagger.quantilePredict` creates response weights by normalizing the weights so that they sum to 1, that is,

$$w_j^* = \frac{u_j}{\sum_{j=1}^n u_j}.$$

## References

- [1] Breiman, L. *Random Forests*. *Machine Learning* 45, pp. 5–32, 2001.
- [2] Meinshausen, N. “Quantile Regression Forests.” *Journal of Machine Learning Research*, Vol. 7, 2006, pp. 983–999.

## See Also

`TreeBagger` | `oobQuantilePredict` | `predict` | `quantileError`

## Topics

“Detect Outliers Using Quantile Regression” on page 18-170

Introduced in R2016b

## randg

Gamma random numbers with unit scale

### Syntax

```
Y = randg
Y = randg(A)
Y = randg(A,m)
Y = randg(A,m,n,p,...)
Y = randg(A,[m,n,p,...])
```

### Description

`Y = randg` returns a scalar random value chosen from a gamma distribution with unit scale and shape.

`Y = randg(A)` returns a matrix of random values chosen from gamma distributions with unit scale. `Y` is the same size as `A`, and `randg` generates each element of `Y` using a shape parameter equal to the corresponding element of `A`.

`Y = randg(A,m)` returns an `m`-by-`m` matrix of random values chosen from gamma distributions with shape parameters `A`. `A` is either an `m`-by-`m` matrix or a scalar. If `A` is a scalar, `randg` uses that single shape parameter value to generate all elements of `Y`.

`Y = randg(A,m,n,p,...)` or `Y = randg(A,[m,n,p,...])` returns an `m`-by-`n`-by-`p`-by-`...` array of random values chosen from gamma distributions with shape parameters `A`. `A` is either an `m`-by-`n`-by-`p`-by-`...` array or a scalar.

`randg` produces pseudo-random numbers using the MATLAB functions `rand` and `randn`. The sequence of numbers generated is determined by the settings of the uniform random number generator that underlies `rand` and `randn`. Control that shared random number generator using `rng`. See the `rng` documentation for more information.

---

**Note** To generate gamma random numbers and specify both the scale and shape parameters, you should call `gamrnd`.

---

## Examples

### Example 1

Generate a 100-by-1 array of values drawn from a gamma distribution with shape parameter 3.

```
r = randg(3,100,1);
```

### Example 2

Generate a 100-by-2 array of values drawn from gamma distributions with shape parameters 3 and 2.

```
A = [ones(100,1)*3,ones(100,1)*2];  
r = randg(A,[100,2]);
```

### Example 3

To create reproducible output from `randg`, reset the random number generator used by `rand` and `randn` to its default startup settings. This way `randg` produces the same random numbers as if you restarted MATLAB.

```
rng('default')  
randg(3,1,5)  
  
ans =  
  
    6.9223    4.3369    1.0505    3.2662   11.3269
```

### Example 4

Save the settings for the random number generator used by `rand` and `randn`, generate 5 values from `randg`, restore the settings, and repeat those values.

```
s = rng; % Obtain the current state of the random stream  
r1 = randg(10,1,5)  
  
r1 =
```

```
9.4719    9.0433    15.0774    14.7763    6.3775

rng(s); % Reset the stream to the previous state
r2 = randg(10,1,5)

r2 =

9.4719    9.0433    15.0774    14.7763    6.3775
```

r2 contains exactly the same values as r1.

## Example 5

Reinitialize the random number generator used by `rand` and `randn` with a seed based on the current time. `randg` returns different values each time you do this. Note that it is usually not necessary to do this more than once per MATLAB session.

```
rng('shuffle');
randg(2,1,5);
```

## References

- [1] Marsaglia, G., and W. W. Tsang. "A Simple Method for Generating Gamma Variables." *ACM Transactions on Mathematical Software*. Vol. 26, 2000, pp. 363–372.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`gamrnd`

**Introduced before R2006a**

# random

Random numbers

## Syntax

```
Y = random(pd)
Y = random(pd, m, n, ...)
Y = random(pd, [m, n, ...])
Y = random(name, A)
Y = random(name, A, B)
Y = random(name, A, B, C)
Y = random(name, A, B, C, D)
Y = random(name, A, m, n, ...)
Y = random(name, A, [m, n, ...])
Y = random(name, A, B, m, n, ...)
Y = random(name, A, B, [m, n, ...])
Y = random(name, A, B, C, m, n, ...)
Y = random(name, A, B, C, [m, n, ...])
Y = random(name, A, B, C, D, m, n, ...)
Y = random(name, A, B, C, D, [m, n, ...])
```

## Description

`Y = random(pd)` returns a random number `Y` from the distribution specified by the probability distribution object `pd`. You can create a probability distribution object with specified parameter values using `makedist`, or fit a probability distribution object to sample data using `fitdist`.

`Y = random(pd, m, n, ...)` or `Y = random(pd, [m, n, ...])` returns an `m`-by-`n`-by... matrix of random numbers from the probability distribution specified by `pd`.

`Y = random(name, A)` where `name` is the name of a distribution that takes a single parameter, returns random numbers `Y` from the one-parameter family of distributions specified by `name`. Parameter values for the distribution are given in `A`.

$Y$  is the same size as  $A$ .

$Y = \text{random}(\text{name}, A, B)$  returns random numbers  $Y$  from a two-parameter family of distributions. Parameter values for the distribution are given in  $A$  and  $B$ .

If  $A$  and  $B$  are arrays, they must be the same size. If either  $A$  or  $B$  are scalars, they are expanded to constant matrices of the same size.

$Y = \text{random}(\text{name}, A, B, C)$  returns random numbers  $Y$  from a three-parameter family of distributions. Parameter values for the distribution are given in  $A$ ,  $B$ , and  $C$ .

If  $A$ ,  $B$ , and  $C$  are arrays, they must be the same size. If any of  $A$ ,  $B$ , or  $C$  are scalars, they are expanded to constant matrices of the same size.

$Y = \text{random}(\text{name}, A, B, C, D)$  returns random numbers  $Y$  from a four-parameter family of distributions. Parameter values for the distribution are given in  $A$ ,  $B$ ,  $C$ , and  $D$ .

If  $A$ ,  $B$ ,  $C$ , and  $D$  are arrays, they must be the same size. If any of  $A$ ,  $B$ ,  $C$ , or  $D$  are scalars, they are expanded to constant matrices of the same size.

$Y = \text{random}(\text{name}, A, m, n, \dots)$  or  $Y = \text{random}(\text{name}, A, [m, n, \dots])$  returns an  $m$ -by- $n$ -by... matrix of random numbers.

Similarly,  $Y = \text{random}(\text{name}, A, B, m, n, \dots)$  or  $Y = \text{random}(\text{name}, A, B, [m, n, \dots])$  returns an  $m$ -by- $n$ -by... matrix of random numbers for distributions that require two parameters.  $Y = \text{random}(\text{name}, A, B, C, m, n, \dots)$  or  $Y = \text{random}(\text{name}, A, B, C, [m, n, \dots])$  returns an  $m$ -by- $n$ -by... matrix of random numbers for distributions that require three parameters.  $Y = \text{random}(\text{name}, A, B, C, D, m, n, \dots)$  or  $Y = \text{random}(\text{name}, A, B, C, D, [m, n, \dots])$  returns an  $m$ -by- $n$ -by... matrix of random numbers for distributions that require four parameters.

If any of  $A$ ,  $B$ ,  $C$ , or  $D$  are arrays, then the specified dimensions must match the common dimensions of  $A$ ,  $B$ ,  $C$ , and  $D$  after any necessary scalar expansion.

The following table denotes the acceptable values for `name`, as well as the parameters for that distribution:



name	Distribution	Input Parameter A	Input Parameter B	Input Parameter C	Input Parameter D
'beta' or 'Beta'	“Beta Distribution” on page B-4	a	b	—	—
'bino' or 'Binomial'	“Binomial Distribution” on page B-9	n: number of trials	p: probability of success for each trial	—	—
'birnbaumsaunders'	“Birnbaum-Saunders Distribution” on page B-13	$\beta$	$\gamma$	—	—
'burr' or 'Burr'	“Burr Type XII Distribution” on page B-15	$\alpha$ : scale parameter	c: shape parameter	k: shape parameter	—
'chi2' or 'Chisquare'	“Chi-Square Distribution” on page B-28	$\nu$ : degrees of freedom	—	—	—
'exp' or 'Exponential'	“Exponential Distribution” on page B-34	$\mu$ : mean	—	—	—
'ev' or 'Extreme Value'	“Extreme Value Distribution” on page B-38	$\mu$ : location parameter	$\sigma$ : scale parameter	—	—
'f' or 'F'	“F Distribution” on page B-44	$\nu_1$ : numerator degrees of freedom	$\nu_2$ : denominator degrees of freedom	—	—
'gam' or 'Gamma'	“Gamma Distribution” on page B-47	a: shape parameter	b: scale parameter	—	—
'gev' or 'Generalized Extreme Value'	“Generalized Extreme Value Distribution” on page B-53	k: shape parameter	$\sigma$ : scale parameter	$\mu$ : location parameter	—

name	Distribution	Input Parameter A	Input Parameter B	Input Parameter C	Input Parameter D
'gp' or 'Generalized Pareto'	“Generalized Pareto Distribution” on page B-58	k: tail index (shape) parameter	$\sigma$ : scale parameter	$\mu$ : threshold (location) parameter	—
'geo' or 'Geometric'	“Geometric Distribution” on page B-64	p: probability parameter	—	—	—
'hn' or 'Half Normal'	“Half-Normal Distribution” on page B-73	$\mu$ : location	$\sigma$ : scale	—	—
'hyge' or 'Hypergeometric'	“Hypergeometric Distribution” on page B-79	M: size of the population	K: number of items with the desired characteristic in the population	n: number of samples drawn	—
'inversegaussian'	“Inverse Gaussian Distribution” on page B-82	$\mu$	$\lambda$	—	—
'logistic'	“Logistic Distribution” on page B-96	$\mu$	$\sigma$	—	—
'loglogistic'	“Loglogistic Distribution” on page B-98	$\mu$	$\sigma$	—	—
'logn' or 'Lognormal'	“Lognormal Distribution” on page B-100	$\mu$	$\sigma$	—	—
'nakagami'	“Nakagami Distribution” on page B-118	$\mu$	$\omega$	—	—

name	Distribution	Input Parameter A	Input Parameter B	Input Parameter C	Input Parameter D
'nbin' or 'Negative Binomial'	“Negative Binomial Distribution” on page B-120	r: number of successes	p: probability of success in a single trial	—	—
'ncf' or 'Noncentral F'	“Noncentral F Distribution” on page B-128	v1: numerator degrees of freedom	v2: denominator degrees of freedom	$\delta$ : noncentrality parameter	—
'nct' or 'Noncentral t'	“Noncentral t Distribution” on page B-131	v: degrees of freedom	$\delta$ : noncentrality parameter	—	—
'ncx2' or 'Noncentral Chi-square'	“Noncentral Chi-Square Distribution” on page B-125	v: degrees of freedom	$\delta$ : noncentrality parameter	—	—
'norm' or 'Normal'	“Normal Distribution” on page B-134	$\mu$ : mean	$\sigma$ : standard deviation	—	—
'poiss' or 'Poisson'	“Poisson Distribution” on page B-143	$\lambda$ : mean	—	—	—
'rayl' or 'Rayleigh'	“Rayleigh Distribution” on page B-146	b: scale parameter	—	—	—
'rician'	“Rician Distribution” on page B-149	s: noncentrality parameter	$\sigma$ : scale parameter	—	—
'stable'	“Stable Distribution” on page B-151	$\alpha$ : first shape parameter	$\beta$ : second shape parameter	$\gamma$ : scale parameter	$\delta$ : location parameter
't' or 'T'	“Student's t Distribution” on page B-166	v: degrees of freedom	—	—	—

name	Distribution	Input Parameter A	Input Parameter B	Input Parameter C	Input Parameter D
'tlocationscale'	“t Location-Scale Distribution” on page B-175	$\mu$ : location parameter	$\sigma$ : scale parameter	$\nu$ : shape parameter	—
'unif' or 'Uniform'	“Uniform Distribution (Continuous)” on page B-184	a: lower endpoint (minimum)	b: upper endpoint (maximum)	—	—
'unid' or 'Discrete Uniform'	“Uniform Distribution (Discrete)” on page B-190	N: maximum observable value	—	—	—
'wbl' or 'Weibull'	“Weibull Distribution” on page B-193	a: scale parameter	b: shape parameter	—	—

## Examples

### Generate Random Numbers from a Probability Distribution

Generate a 2-by-4 array of random values from the normal distribution with mean equal to 0 and standard deviation equal to 1.

```
x1 = random('Normal', 0, 1, 2, 4)
```

```
x1 =
```

```
    0.5377    -2.2588     0.3188    -0.4336
    1.8339     0.8622    -1.3077     0.3426
```

Generate a single random value from Poisson distributions with rate parameters 1, 2, ..., 6, respectively.

```
x2 = random('Poisson', 1:6, 1, 6)
```

x2 =  
4 2 3 7 4 9

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

[cdf](#) | [fitdist](#) | [icdf](#) | [makedist](#) | [mle](#) | [pdf](#)

### Topics

“Working with Probability Distributions” on page 5-3

Introduced before R2006a

## random

**Class:** CompactGeneralizedLinearModel

Simulate responses for generalized linear regression model

## Syntax

```
ysim = random mdl, Xnew  
ysim = random mdl, Xnew, Name, Value
```

## Description

`ysim = random(mdl, Xnew)` simulates responses from the `mdl` generalized linear model to the data in `Xnew`.

`ysim = random(mdl, Xnew, Name, Value)` simulates responses with additional options specified by one or more `Name, Value` pair arguments.

## Input Arguments

### **mdl**

Generalized linear model, specified as a full `GeneralizedLinearModel` object constructed using `fitglm` or `stepwiseglm`, or a compacted `CompactGeneralizedLinearModel` object constructed using `compact`.

### **Xnew**

Points at which `mdl` predicts responses.

- If `Xnew` is a table or dataset array, it must contain the predictor names in `mdl`.
- If `Xnew` is a numeric matrix, it must have the same number of variables (columns) as was used to create `mdl`. Furthermore, all variables used in creating `mdl` must be numeric.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### **BinomialSize**

The value of the binomial  $n$  parameter for each row in the training data. `BinomialSize` can be a vector the same length as `Xnew`, or a scalar that applies to each row. The default value 1 produces `ysim` values that are predicted proportions. Use `BinomialSize` only if `mdl` is fit to a binomial distribution.

**Default:** 1

### **Offset**

Value of the offset for each row in `Xnew`. `Offset` can be a vector the same length as `Xnew`, or a scalar that applies to each row. The offset is used as an additional predictor with a coefficient value fixed at 1. In other words, if `b` is the fitted coefficient vector, and `link` is the link function,

$$\text{link}(\text{ysim}) = \text{Offset} + \text{Xnew} * \text{b}.$$

**Default:** `zeros(size(Xnew,1))`

## Output Arguments

### **ysim**

Vector of simulated values at `Xnew`.

`random` generates `ysim` using random values with mean given by the fitted model, and with the distribution used in `mdl`. The values in `ysim` are independent conditional on the predictors. For binomial and Poisson fits, `random` generates `ysim` with the specified distribution with no adjustment for any estimated dispersion.

## Examples

### Generalized Linear Model Simulation

Create a generalized linear model, and simulate its response to new data.

Generate artificial data for the model, Poisson random numbers with one underlying predictors  $X$ .

```
rng('default') % reproducible
X = rand(20,1);
mu = exp(1 + 2*X);
y = poissrnd(mu);
```

Create a generalized linear regression model of Poisson data.

```
mdl = fitglm(X,y,'y ~ x1','distr','poisson');
```

Create points for prediction.

```
Xnew = (0:.05:1)';
```

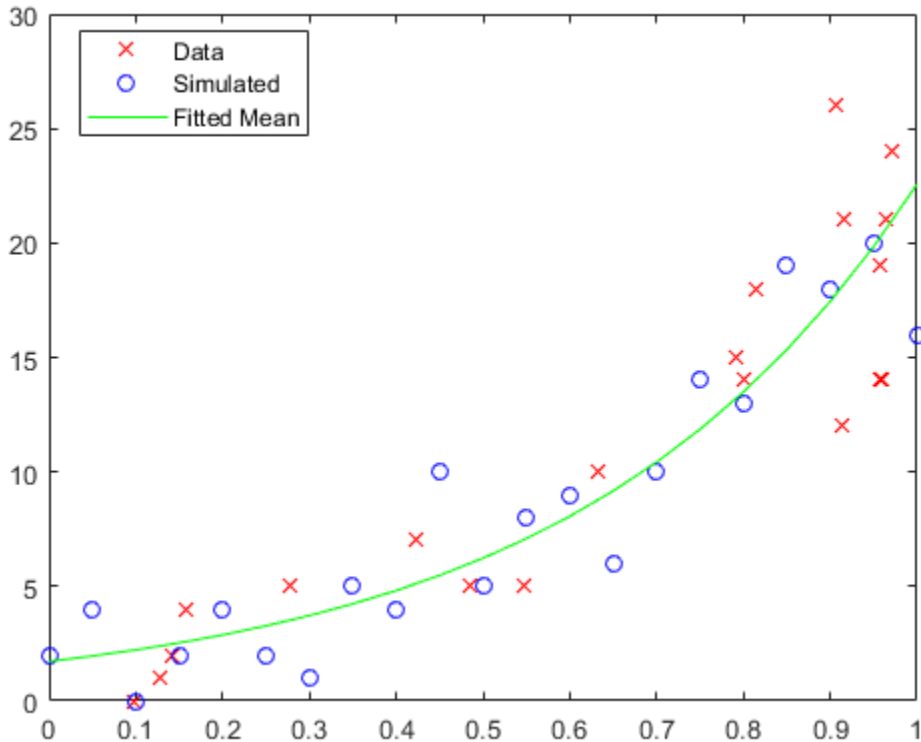
Simulate responses at the new points.

```
ysim = random(mdl,Xnew);
```

Plot the simulated values along with the original values.

```
plot(X,y,'rx',Xnew,ysim,'bo',...
      Xnew,feval(mdl,Xnew),'g-')
legend('Data','Simulated','Fitted Mean',...
       'Location','best')
```





### Generate Code From Function That Simulates Responses Given New Data

Train a generalized linear model, and then generate code from a function that draws random responses from the model given new predictor data.

Generate artificial data, specifically:

- $x \sim N(1, 0.5^2)$ .
- $\beta_0 = 1$  and  $\beta = -2$ .
- $y \sim \text{Binomial}\left(10, \frac{\exp(1+x\beta)}{1+\exp(1+x\beta)}\right)$ .

```
rng('default') % For reproducibility
x = 1 + randn(100,1)*0.5;
beta = -2;
p = exp(1 + x*beta)./(1 + exp(1 + x*beta)); % Inverse logit
n = 10;
y = binornd(n,p,100,1);
```

Create a generalized linear regression model of binomial data. Specify a binomial sample size of 10.

```
mdl = fitglm(x,y,'y ~ x1','Distribution','Binomial','BinomialSize',n);
```

Save the fitted generalized linear model to the file GLMMdl.mat.

```
saveCompactModel(mdl,'GLMMdl');
```

In your current folder, declare a function called `myrandomGLM.m` that:

- Accepts measurements with columns corresponding to those in `X` and valid name-value pair arguments
- Loads the fitted generalized linear model in `GLMMdl.mat`
- Simulates responses from the loaded GLM model

```
function y = myrandomGLM(x,varargin) %#codegen
%MYRANDOMGLM Simulate response using GLM model
% MYRANDOMGLM simulates responses for the n observations in the n-by-1
% vector x using the GLM model stored in the MAT-file GLMMdl.mat, and
% then returns the simulations in the n-by-1 vector y.
CompactMdl = loadCompactModel('GLMMdl');
narginchk(1,Inf);
y = random(CompactMdl,x,varargin{:});
end
```

Generate a MEX function from `myrandomGLM.m`. Specify the in-sample predictor data `X` and 10 for the binomial parameter `n`. Because C uses static typing, `codegen` must determine the properties of all variables in MATLAB® files at compile time. To designate arguments as compile-time constants, use `coder.Constant`.

```
codegen -config:mex myrandomGLM -args {x,coder.Constant('BinomialSize'),coder.Constant(n)}
```

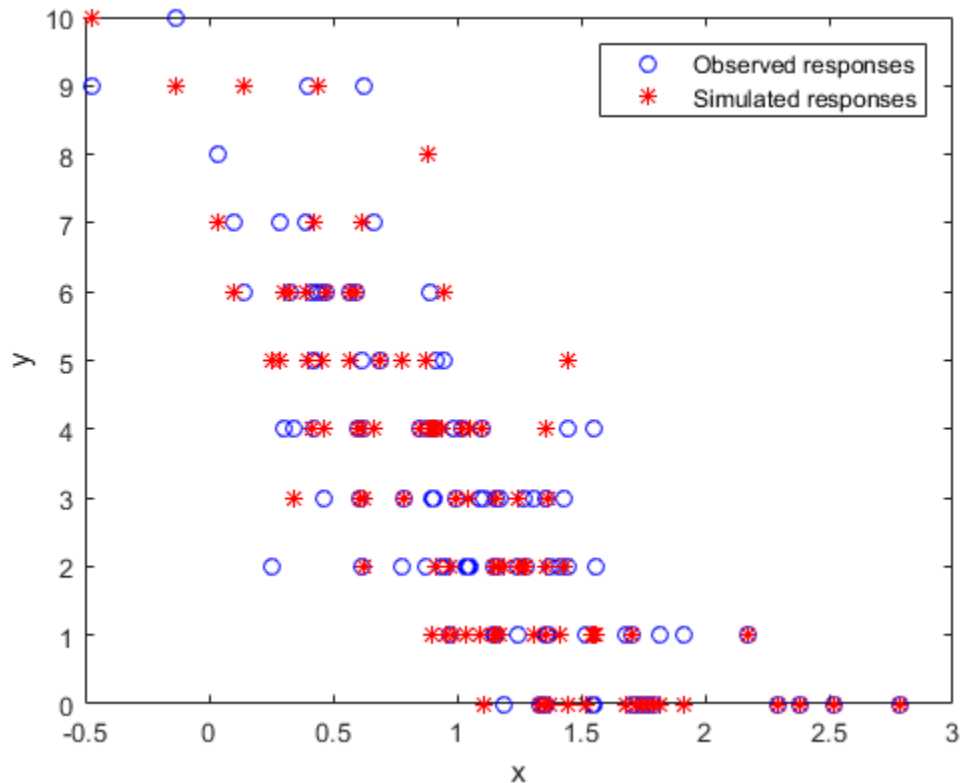
codegen generates the MEX file `myrandomGLM_mex.mexw64` in your current folder. The file extension depends on your platform.

Simulate responses using the MEX function.

```
ysim = myrandomGLM_mex(x, 'BinomialSize', n);
```

Plot the simulated values and the data in the same figure.

```
figure;  
plot(x, y, 'bo', x, ysim, 'r*');  
legend('Observed responses', 'Simulated responses');  
xlabel('x');  
ylabel('y');
```



The observed and simulated responses appear to be similarly distributed.

- “Predict or Simulate Responses to New Data” on page 12-35

## Alternatives

For predictions without random noise, use `predict` or `feval`.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- If either of the following is true, then `random` can return a different sequence of numbers than MATLAB.
  - The output is nonscalar.
  - An input parameter is invalid for the distribution.
- This table contains notes about arguments.

Argument	Notes and Limitations
<code>mdl</code>	<ul style="list-style-type: none"><li>• Save a trained model by using <code>saveCompactModel</code>, and then load the model by using <code>loadCompactModel</code> within a function that you declare.</li><li>• Must be a compile-time constant; its value cannot change in the generated code.</li><li>• For more details, see “Code Generation” on page 32-0 of the <code>CompactGeneralizedLinearModel</code> class.</li></ul>

Argument	Notes and Limitations
Xnew	<ul style="list-style-type: none"> <li>You must supply Xnew.</li> <li>Must be a single- or double-precision matrix and can be variable-sized. However, the number of columns in Xnew must be <code>numel mdl.PredictorNames</code>.</li> <li>Rows and columns must correspond to observations and predictors, respectively.</li> </ul>
Name-value pair arguments	Names in name-value pair arguments must be compile-time constants. For example, to use the 'BinomialSize' name-value pair argument in the generated code, include <code>{coder.Constant('BinomialSize'),0}</code> in the <code>-args</code> value of <code>codegen</code> .

## See Also

`CompactGeneralizedLinearModel` | `GeneralizedLinearModel` | `predict`

## Topics

“Predict or Simulate Responses to New Data” on page 12-35

“Generalized Linear Models” on page 12-12

“Code Generation Workflows” on page 31-2

Introduced in R2012a

## random

**Class:** `GeneralizedLinearMixedModel`

Generate random responses from fitted generalized linear mixed-effects model

## Syntax

```
ysim = random(glme)
ysim = random(glme, tblnew)
ysim = random( ____, Name, Value)
```

## Description

`ysim = random(glme)` returns simulated responses, `ysim`, from the fitted generalized linear mixed-effects model `glme`, at the original design points.

`ysim = random(glme, tblnew)` returns simulated responses using new input values specified in the table or dataset array, `tblnew`.

`ysim = random( ____, Name, Value)` returns simulated responses using additional options specified by one or more `Name, Value` pair arguments, using any of the previous syntaxes. For example, you can specify observation weights, binomial sizes, or offsets for the model.

## Input Arguments

**`glme` — Generalized linear mixed-effects model**

`GeneralizedLinearMixedModel` object

Generalized linear mixed-effects model, specified as a `GeneralizedLinearMixedModel` object. For properties and methods of this object, see `GeneralizedLinearMixedModel`.

**`tblnew` — New input data**

table | dataset array

New input data, which includes the response variable, predictor variables, and grouping variables on page 2-59, specified as a table or dataset array. The predictor variables can be continuous or grouping variables. `tblnew` must contain the same variables as the original table or dataset array, `tbl`, used to fit the generalized linear mixed-effects model `glme`.

Data Types: `single` | `double` | `logical` | `char`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### **BinomialSize** — Number of trials for binomial distribution

`ones(m, 1)` (default) |  $m$ -by-1 vector of positive integer values

Number of trials for binomial distribution, specified as the comma-separated pair consisting of 'BinomialSize' and an  $m$ -by-1 vector of positive integer values, where  $m$  is the number of rows in `tblnew`. The 'BinomialSize' name-value pair applies only to the binomial distribution. The value specifies the number of binomial trials when generating the random response values.

Data Types: `single` | `double`

### **Offset** — Model offset

`zeros(m, 1)` (default) | vector of scalar values

Model offset, specified as a vector of scalar values of length  $m$ , where  $m$  is the number of rows in `tblnew`. The offset is used as an additional predictor and has a coefficient value fixed at 1.

### **Weights** — Observation weights

$m$ -by-1 vector of nonnegative scalar values

Observation weights, specified as the comma-separated pair consisting of 'Weights' and an  $m$ -by-1 vector of nonnegative scalar values, where  $m$  is the number of rows in `tblnew`. If the response distribution is binomial or Poisson, then 'Weights' must be a vector of positive integers.

Data Types: `single` | `double`

## Output Arguments

### **`ysim` — Simulated response values**

*m*-by-1 vector

Simulated response values, returned as an *m*-by-1 vector, where *m* is the number of rows in `tblnew.random`. `random` creates `ysim` by first generating the random-effects vector based on its fitted prior distribution. `random` then generates `ysim` from its fitted conditional distribution given the random effects. `random` takes into account the effect of observation weights specified when fitting the model using `fitglm`, if any.

## Examples

### Simulate Random Responses From a GLME Model

Load the sample data.

```
load mfr
```

This simulated data is from a manufacturing company that operates 50 factories across the world, with each factory running a batch process to create a finished product. The company wants to decrease the number of defects in each batch, so it developed a new manufacturing process. To test the effectiveness of the new process, the company selected 20 of its factories at random to participate in an experiment: Ten factories implemented the new process, while the other ten continued to run the old process. In each of the 20 factories, the company ran five batches (for a total of 100 batches) and recorded the following data:

- Flag to indicate whether the batch used the new process (`newprocess`)
- Processing time for each batch, in hours (`time`)
- Temperature of the batch, in degrees Celsius (`temp`)
- Categorical variable indicating the supplier (A, B, or C) of the chemical used in the batch (`supplier`)



- Number of defects in the batch (`defects`)

The data also includes `time_dev` and `temp_dev`, which represent the absolute deviation of time and temperature, respectively, from the process standard of 3 hours at 20 degrees Celsius.

Fit a generalized linear mixed-effects model using `newprocess`, `time_dev`, `temp_dev`, and `supplier` as fixed-effects predictors. Include a random-effects term for intercept grouped by `factory`, to account for quality differences that might exist due to factory-specific variations. The response variable `defects` has a Poisson distribution, and the appropriate link function for this model is `log`. Use the Laplace fit method to estimate the coefficients. Specify the dummy variable encoding as `'effects'`, so the dummy variable coefficients sum to 0.

The number of defects can be modeled using a Poisson distribution

$$\text{defects}_{ij} \sim \text{Poisson}(\mu_{ij})$$

This corresponds to the generalized linear mixed-effects model

$$\log(\mu_{ij}) = \beta_0 + \beta_1 \text{newprocess}_{ij} + \beta_2 \text{time\_dev}_{ij} + \beta_3 \text{temp\_dev}_{ij} + \beta_4 \text{supplier\_C}_{ij} + \beta_5 \text{supplier\_B}_{ij} + b_i,$$

where

- `defectsij` is the number of defects observed in the batch produced by factory *i* during batch *j*.
- $\mu_{ij}$  is the mean number of defects corresponding to factory *i* (where  $i = 1, 2, \dots, 20$ ) during batch *j* (where  $j = 1, 2, \dots, 5$ ).
- `newprocessij`, `time_devij`, and `temp_devij` are the measurements for each variable that correspond to factory *i* during batch *j*. For example, `newprocessij` indicates whether the batch produced by factory *i* during batch *j* used the new process.
- `supplier_Cij` and `supplier_Bij` are dummy variables that use effects (sum-to-zero) coding to indicate whether company C or B, respectively, supplied the process chemicals for the batch produced by factory *i* during batch *j*.
- $b_i \sim N(0, \sigma_b^2)$  is a random-effects intercept for each factory *i* that accounts for factory-specific variation in quality.

```
glme = fitglme(mfr, 'defects ~ 1 + newprocess + time_dev + temp_dev + supplier + (1|fact
```

Use `random` to simulate a new response vector from the fitted model.

```
rng(0, 'twister'); % For reproducibility
ynew = random(glme);
```

Display the first 10 rows of the simulated response vector.

```
ynew(1:10)
```

```
ans =
```

```
3
3
1
7
5
8
7
9
5
9
```

Simulate a new response vector using new input values. Create a new table by copying the first 10 rows of `mfr` into `tblnew`.

```
tblnew = mfr(1:10, :);
```

The first 10 rows of `mfr` include data collected from trials 1 through 5 for factories 1 and 2. Both factories used the old process for all of their trials during the experiment, so `newprocess = 0` for all 10 observations.

Change the value of `newprocess` to 1 for the observations in `tblnew`.

```
tblnew.newprocess = ones(height(tblnew), 1);
```

Simulate new responses using the new input values in `tblnew`.

```
ynew2 = random(glme, tblnew)
```

```
ynew2 =
```

```
2
```

3  
5  
4  
2  
2  
2  
1  
2  
0

## Definitions

### Conditional Distribution Method

`random` generates random data from the fitted generalized linear mixed-effects model as follows:

- Sample  $b_{sim} \sim P(b | \theta, \sigma^2)$ , where  $P(b | \theta, \sigma^2)$  is the estimated prior distribution of random effects, and  $\hat{\theta}$  is a vector of estimated covariance parameters, and  $\hat{\sigma}^2$  is the estimated dispersion parameter.
- Given  $b_{sim}$ , for  $i = 1$  to  $m$ , sample  $y_{sim\_i} \sim P(y_{new\_i} | b_{sim}, \beta, \theta, \sigma^2)$ , where  $P(y_{new\_i} | b_{sim}, \beta, \theta, \sigma^2)$  is the conditional distribution of the  $i$ th new response  $y_{new\_i}$  given  $b_{sim}$  and the model parameters.

## See Also

`GeneralizedLinearMixedModel` | `fitglm` | `fitted` | `predict`

## random

**Class:** `gmdistribution`

Random numbers from Gaussian mixture distribution

## Syntax

```
y = random(obj)
Y = random(obj, n)
[Y, idx] = random(obj, n)
```

## Description

`y = random(obj)` generates a 1-by- $d$  vector  $y$  drawn at random from the  $d$ -dimensional Gaussian mixture distribution defined by `obj`. `obj` is an object created by `gmdistribution` or `fitgmdist`.

`Y = random(obj, n)` generates an  $n$ -by- $d$  matrix  $Y$  of  $n$   $d$ -dimensional random samples.

`[Y, idx] = random(obj, n)` also returns an  $n$ -by-1 vector `idx`, where `idx(I)` is the index of the component used to generate  $Y(I, :)$ .

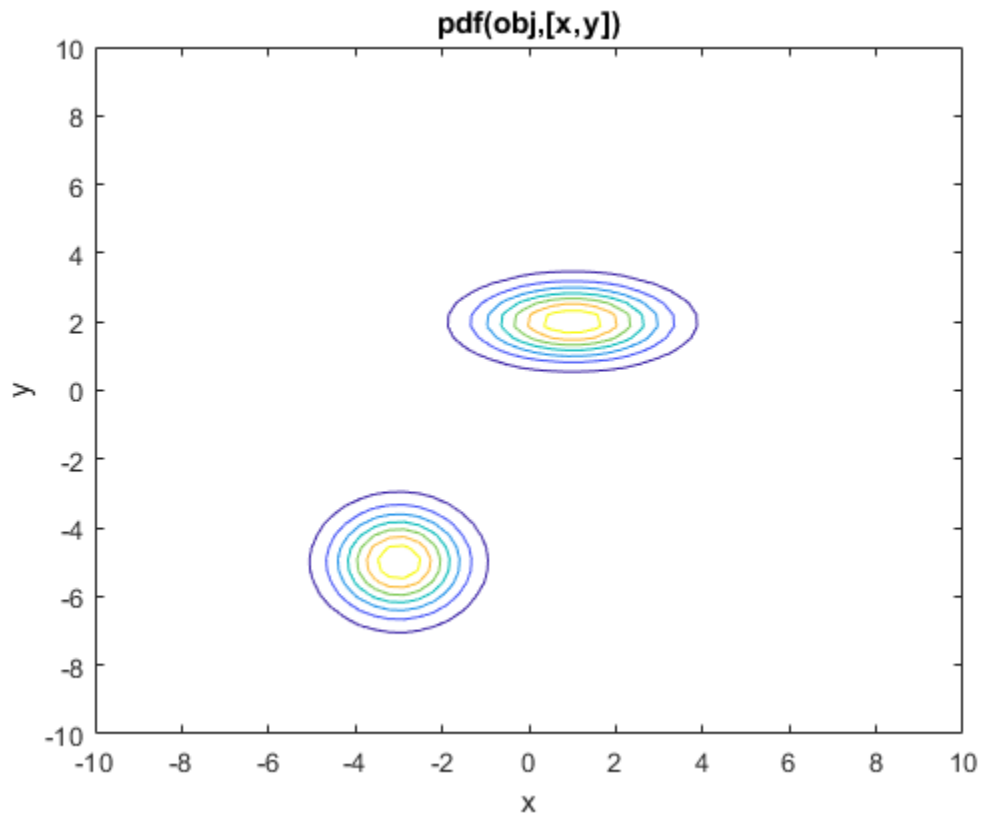
## Examples

### Generate Gaussian Mixture Variates

Create a `gmdistribution` object defining a two-component mixture of bivariate Gaussian distributions.

```
MU = [1 2; -3 -5];
SIGMA = cat(3, [2 0; 0 .5], [1 0; 0 1]);
p = ones(1, 2)/2;
obj = gmdistribution(MU, SIGMA, p);
```

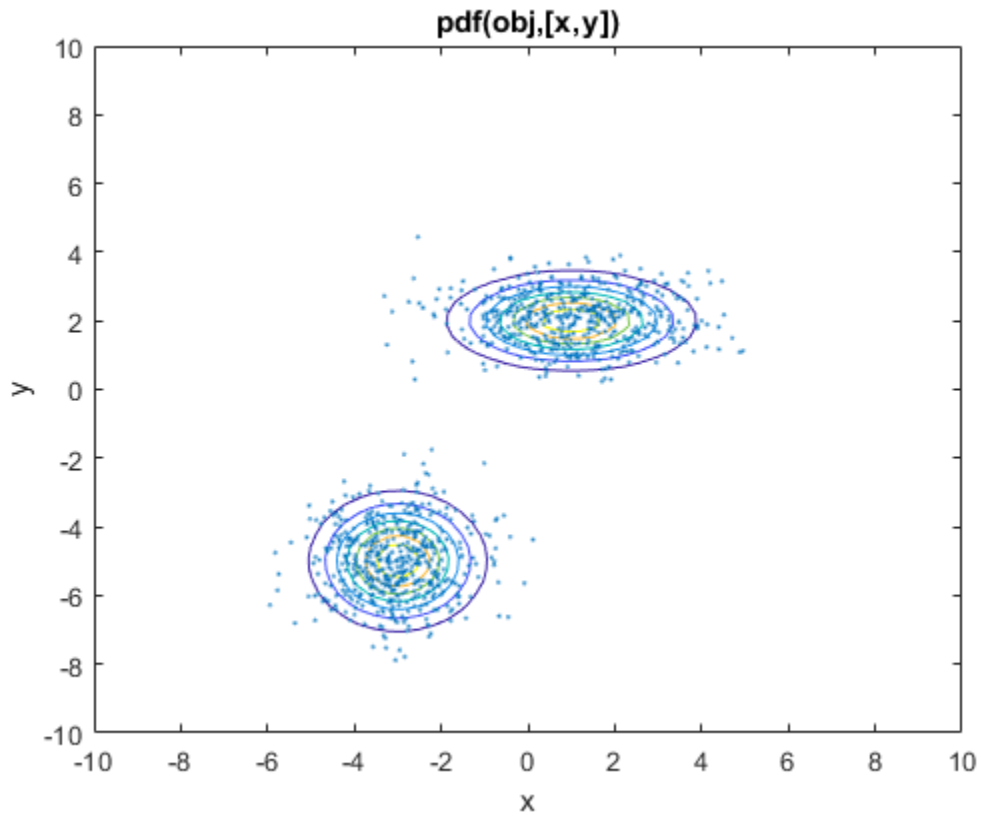
```
ezcontour(@ (x,y) pdf(obj,[x y]),[-10 10],[-10 10])  
hold on
```



Generate 1000 random values.

```
rng(1); % For reproducibility  
Y = random(obj,1000);
```

```
scatter(Y(:,1),Y(:,2),10, 'r')
```



## See Also

`fitgmdist` | `gmdistribution` | `mvnrnd`

---

# random

**Class:** CompactLinearModel

Simulate responses for linear regression model

## Syntax

```
ysim = random mdl  
ysim = random mdl, Xnew
```

## Description

`ysim = random(mdl)` simulates responses from the fitted linear model `mdl` at the original design points.

`ysim = random(mdl, Xnew)` simulates responses from the `mdl` linear model to the data in `Xnew`, adding random noise.

## Input Arguments

**mdl** — Linear model object

LinearModel object | CompactLinearModel object

Linear model object, specified as a full `LinearModel` object constructed using `fitlm` or `stepwiselm`, or a compacted `CompactLinearModel` object constructed using `compact`.

**Xnew** — New predictor input values

table | dataset array | numeric matrix

New predictor input values, specified as a table, dataset array, or numeric matrix.

- If `Xnew` is a table or dataset array, it must contain the predictor names in `mdl`.

- If `Xnew` is a numeric matrix, it must have the same number of variables (columns) as was used to create `mdl`. Furthermore, all variables used in creating `mdl` must be numeric.

## Output Arguments

### **`ysim`** — Predicted mean values

numeric vector

Predicted mean values at `Xnew`, perturbed by random noise, returned as a numeric vector. The noise is independent and normally distributed, with mean equal to zero, and variance equal to the estimated error variance of the model.

## Examples

### Simulate Response Data From Linear Regression Model

Create a model of car mileage as a function of weight, and simulate the response.

Create a quadratic model of car mileage as a function of weight from the `carsmall` data.

```
load carsmall
X = Weight;
y = MPG;
mdl = fitlm(X,y,'quadratic');
```

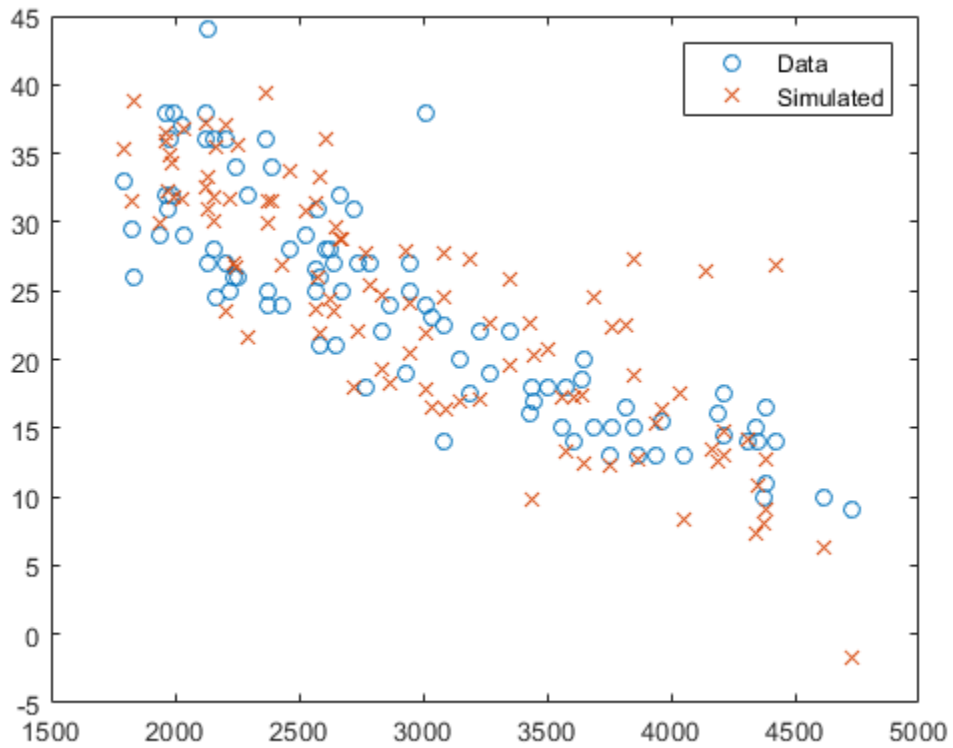
Create simulated responses to the data.

```
Xnew = X;
ysim = random(mdl,Xnew);
```

Plot the original responses and the simulated responses to see how they differ.

```
plot(X,y,'o',X,ysim,'x')
legend('Data','Simulated')
```





- “random” on page 11-40

## Alternatives

For predictions without random noise, use `predict` or `feval`.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- If either of the following is true, then `random` can return a different sequence of numbers than MATLAB.
  - The output is nonscalar.
  - An input parameter is invalid for the distribution.
- This table contains notes about arguments.

Argument	Notes and Limitations
<code>mdl</code>	<ul style="list-style-type: none"> <li>• Save a trained model by using <code>saveCompactModel</code>, and then load the model by using <code>loadCompactModel</code> within a function that you declare.</li> <li>• If you train a linear model by using <code>fitlm</code> and specifying <code>'RobustOpts'</code> as a structure with an anonymous function handle for the <code>RobustWgtFun</code> field, use <code>saveCompactModel</code> to save the model, and use <code>loadCompactModel</code> to load the model, then <code>loadCompactModel</code> cannot restore the <code>Robust</code> property into the MATLAB Workspace. However, <code>loadCompactModel</code> can load the model at compile time for code generation within a function that you declare.</li> <li>• Must be a compile-time constant; its value cannot change in the generated code.</li> <li>• For more details, see “Code Generation” on page 32-0 of the <code>CompactLinearModel</code> class.</li> </ul>

Argument	Notes and Limitations
Xnew	<ul style="list-style-type: none"><li>• You must supply Xnew.</li><li>• Must be a single- or double-precision matrix and can be variable-sized. However, the number of columns in Xnew must be <code>numel mdl.PredictorNames</code>.</li><li>• Rows and columns must correspond to observations and predictors, respectively.</li></ul>

## See Also

`CompactLinearModel` | `LinearModel` | `feval` | `predict`

## Topics

“random” on page 11-40

“Linear Regression” on page 11-11

“Code Generation Workflows” on page 31-2

**Introduced in R2012a**

## random

**Class:** LinearMixedModel

Generate random responses from fitted linear mixed-effects model

### Syntax

```
ysim = random(lme)
ysim = random(lme, tblnew)
ysim = random(lme, Xnew, Znew)
ysim = random(lme, Xnew, Znew, Gnew)
```

### Description

`ysim = random(lme)` returns a vector of simulated responses `ysim` from the fitted linear mixed-effects model `lme` at the original fixed- and random-effects design points, used to fit `lme`.

`random` simulates new random-effects vector and new observation errors. So, the simulated response is

$$y_{sim} = X\hat{\beta} + Z\hat{b} + \varepsilon,$$

where  $\hat{\beta}$  is the estimated fixed-effects coefficients,  $\hat{b}$  is the new random effects, and  $\varepsilon$  is the new observation error.

`random` also accounts for the effect of observation weights, if you use any when fitting the model.

`ysim = random(lme, tblnew)` returns a vector of simulated responses `ysim` from the fitted linear mixed-effects model `lme` at the values in the new table or dataset array `tblnew`. Use a table or dataset array for `random` if you use a table or dataset array for fitting the model `lme`.

`ysim = random(lme, Xnew, Znew)` returns a vector of simulated responses `ysim` from the fitted linear mixed-effects model `lme` at the values in the new fixed- and random-

effects design matrices,  $X_{new}$  and  $Z_{new}$ , respectively.  $Z_{new}$  can also be a cell array of matrices. Use the matrix format for `random` if you use design matrices for fitting the model `lme`.

`ysim = random(lme, Xnew, Znew, Gnew)` returns a vector of simulated responses `ysim` from the fitted linear mixed-effects model `lme` at the values in the new fixed- and random-effects design matrices,  $X_{new}$  and  $Z_{new}$ , respectively, and the grouping variable `Gnew`.

$Z_{new}$  and `Gnew` can also be cell arrays of matrices and grouping variables, respectively.

## Input Arguments

### **lme** — Linear mixed-effects model

`LinearMixedModel` object

Linear mixed-effects model, specified as a `LinearMixedModel` object constructed using `fitlme` or `fitlmematrix`.

### **tblnew** — New input data

`table` | `dataset` array

New input data, which includes the response variable, predictor variables, and grouping variables on page 2-59, specified as a table or dataset array. The predictor variables can be continuous or grouping variables. `tblnew` must have the same variables as in the original table or dataset array used to fit the linear mixed-effects model `lme`.

Data Types: `single` | `double` | `logical` | `char`

### **Xnew** — New fixed-effects design matrix

$n$ -by- $p$  matrix

New fixed-effects design matrix, specified as an  $n$ -by- $p$  matrix, where  $n$  is the number of observations and  $p$  is the number of fixed predictor variables. Each row of  $X$  corresponds to one observation and each column of  $X$  corresponds to one variable.

Data Types: `single` | `double`

### **Znew** — New random-effects design

$n$ -by- $q$  matrix | cell array of length  $R$

New random-effects design, specified as an  $n$ -by- $q$  matrix or a cell array of  $R$  design matrices  $Z\{r\}$ , where  $r = 1, 2, \dots, R$ . If  $Z_{\text{new}}$  is a cell array, then each  $Z\{r\}$  is an  $n$ -by- $q(r)$  matrix, where  $n$  is the number of observations, and  $q(r)$  is the number of random predictor variables.

Data Types: `single` | `double` | `logical` | `char` | `cell`

#### **$Z_{\text{new}}$ — New grouping variable or variables**

vector | cell array of grouping variables of length  $R$

New grouping variable or variables on page 2-59, specified as a vector or a cell array, of length  $R$ , of grouping variables used to fit the linear mixed-effects model, `lme`.

`random` treats all levels of each grouping variable as new levels. It draws an independent random effects vector for each level of each grouping variable.

Data Types: `single` | `double` | `logical` | `char` | `cell`

## Output Arguments

#### **$y_{\text{sim}}$ — Simulated response values**

$n$ -by-1 vector

Simulated response values, returned as an  $n$ -by-1 vector, where  $n$  is the number of observations.

## Examples

### **Generate Random Responses at the Original Design Values**

Load the sample data.

```
load(fullfile(matlabroot, 'examples', 'stats', 'fertilizer.mat'));
```

The dataset array includes data from a split-plot experiment, where soil is divided into three blocks based on the soil type: sandy, silty, and loamy. Each block is divided into five plots, where five different types of tomato plants (cherry, heirloom, grape, vine, and plum) are randomly assigned to these plots. The tomato plants in the plots are then

divided into subplots, where each subplot is treated by one of four fertilizers. This is simulated data.

Store the data in a dataset array called `ds`, for practical purposes, and define `Tomato`, `Soil`, and `Fertilizer` as categorical variables.

```
ds = fertilizer;
ds.Tomato = nominal(ds.Tomato);
ds.Soil = nominal(ds.Soil);
ds.Fertilizer = nominal(ds.Fertilizer);
```

Fit a linear mixed-effects model, where `Fertilizer` and `Tomato` are the fixed-effects variables, and the mean yield varies by the block (soil type), and the plots within blocks (tomato types within soil types) independently.

```
lme = fitlme(ds, 'Yield ~ Fertilizer * Tomato + (1|Soil) + (1|Soil:Tomato)');
```

Generate random response values at the original design points. Display the first five values.

```
rng(123, 'twister') % For reproducibility
ysim = random(lme);
ysim(1:5)
```

```
ans =

    114.8785
    134.2018
    154.2818
    169.7554
     84.6089
```

### Plot Randomly Generated vs. Observed Response Values

Load the sample data.

```
load carsmall
```

Fit a linear mixed-effects model, with a fixed-effects for `Weight`, and a random intercept grouped by `Model_Year`. First, store the data in a table.

```
tbl = table(MPG,Weight,Model_Year);  
lme = fitlme(tbl,'MPG ~ Weight + (1|Model_Year)');
```

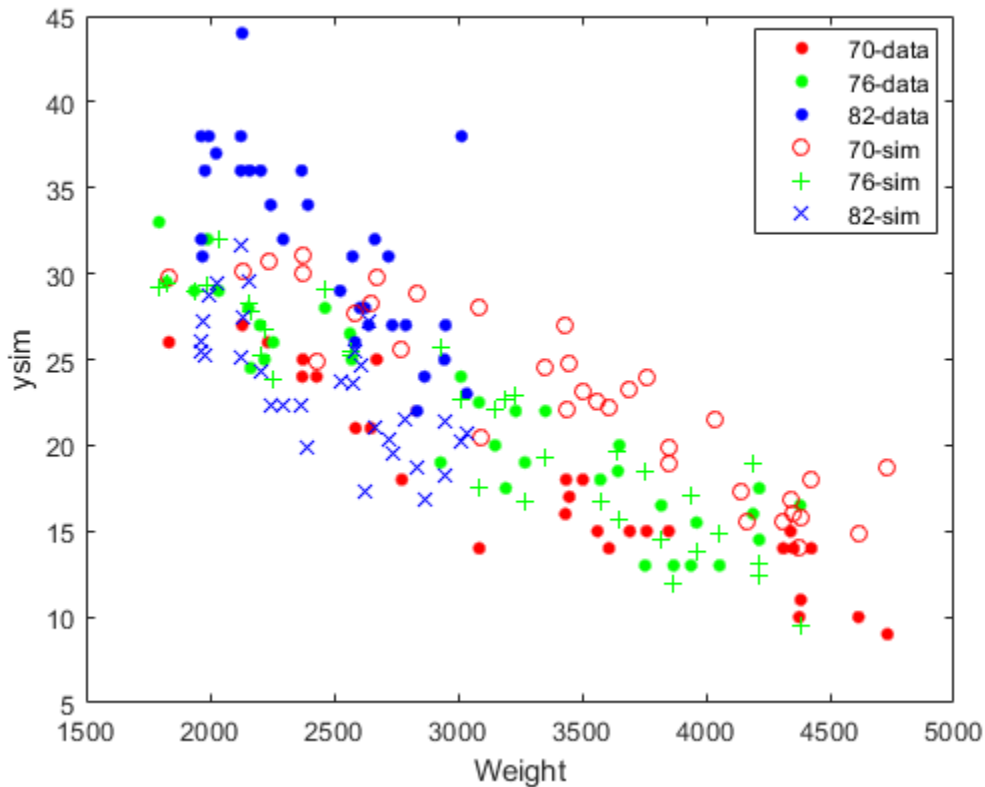
Randomly generate responses using the original data.

```
rng(123,'twister') % For reproducibility  
ysim = random(lme,tbl);
```

Plot the original and the randomly generated responses to see how they differ. Group them by model year.

```
figure()  
gscatter(Weight,MPG,Model_Year)  
hold on  
gscatter(Weight,ysim,Model_Year,[],'o+x')  
legend('70-data','76-data','82-data','70-sim','76-sim','82-sim')  
hold off
```





Note that the simulated random response values for year 82 are lower than the original data for that year. This might be due to a lower simulated random effect for year 82 than the estimated random effect in the original data.

### Generate Responses Using a New Dataset Array

Load the sample data.

```
load(fullfile(matlabroot, 'examples', 'stats', 'fertilizer.mat'));
```

The dataset array includes data from a split-plot experiment, where soil is divided into three blocks based on the soil type: sandy, silty, and loamy. Each block is divided into

five plots, where five different types of tomato plants (cherry, heirloom, grape, vine, and plum) are randomly assigned to these plots. The tomato plants in the plots are then divided into subplots, where each subplot is treated by one of four fertilizers. This is simulated data.

Store the data in a dataset array called `ds`, for practical purposes, and define `Tomato`, `Soil`, and `Fertilizer` as categorical variables.

```
ds = fertilizer;  
ds.Tomato = nominal(ds.Tomato);  
ds.Soil = nominal(ds.Soil);  
ds.Fertilizer = nominal(ds.Fertilizer);
```

Fit a linear mixed-effects model, where `Fertilizer` and `Tomato` are the fixed-effects variables, and the mean yield varies by the block (soil type), and the plots within blocks (tomato types within soil types) independently.

```
lme = fitlme(ds, 'Yield ~ Fertilizer * Tomato + (1|Soil) + (1|Soil:Tomato)');
```

Create a new dataset array with design values. The new dataset array must have the same variables as the original dataset array you use for fitting the model `lme`.

```
dsnew = dataset();  
dsnew.Soil = nominal({'Sandy'; 'Silty'; 'Silty'});  
dsnew.Tomato = nominal({'Cherry'; 'Vine'; 'Plum'});  
dsnew.Fertilizer = nominal([2;2;4]);
```

Generate random responses at the new points.

```
rng(123, 'twister') % For reproducibility  
ysim = random(lme, dsnew)
```

```
ysim =  
  
    99.6006  
   101.9911  
   161.4026
```

## Generate Random Responses Using New Design Matrices

Load the sample data.

```
load carbig
```

Fit a linear mixed-effects model for miles per gallon (MPG), with fixed effects for acceleration, horsepower, and cylinders, and potentially correlated random effect for intercept and acceleration grouped by model year.

First, prepare the design matrices for fitting the linear mixed-effects model.

```
X = [ones(406,1) Acceleration Horsepower];
Z = [ones(406,1) Acceleration];
Model_Year = nominal(Model_Year);
G = Model_Year;
```

Now, fit the model using `fitlmematrix` with the defined design matrices and grouping variables.

```
lme = fitlmematrix(X,MPG,Z,G,'FixedEffectPredictors',...
{'Intercept','Acceleration','Horsepower'},'RandomEffectPredictors',...
{{'Intercept','Acceleration'}},'RandomEffectGroups',{'Model_Year'});
```

Create the design matrices that contain the data at which to predict the response values. `Xnew` must have three columns as in `X`. The first column must be a column of 1s. And the values in the last two columns must correspond to `Acceleration` and `Horsepower`, respectively. The first column of `Znew` must be a column of 1s, and the second column must contain the same `Acceleration` values as in `Xnew`. The original grouping variable in `G` is the model year. So, `Gnew` must contain values for the model year. Note that `Gnew` must contain nominal values.

```
Xnew = [1,13.5,185; 1,17,205; 1,21.2,193];
Znew = [1,13.5; 1,17; 1,21.2];
Gnew = nominal([73 77 82]);
```

Generate random responses for the data in the new design matrices.

```
rng(123,'twister') % For reproducibility
ysim = random(lme,Xnew,Znew,Gnew)
```

```
ysim =
    15.7416
    10.6085
     6.8796
```

Now, repeat the same for a linear mixed-effects model with uncorrelated random-effects terms for intercept and acceleration. First, change the original random effects design and the random effects grouping variables. Then, fit the model.

```
Z = {ones(406,1),Acceleration};
G = {Model_Year,Model_Year};

lme = fitlmematrix(X,MPG,Z,G,'FixedEffectPredictors',...
{'Intercept','Acceleration','Horsepower'},'RandomEffectPredictors',...
{'Intercept'},{'Acceleration'}},'RandomEffectGroups',{'Model_Year','Model_Year'});
```

Now, recreate the new random effects design, `Znew`, and the grouping variable design, `Gnew`, using which to predict the response values.

```
Znew = {[1;1;1],[13.5;17;21.2]};
MY = nominal([73 77 82]);
Gnew = {MY,MY};
```

Generate random responses using the new design matrices.

```
rng(123,'twister') % For reproducibility
ysim = random(lme,Xnew,Znew,Gnew)
```

```
ysim =
    16.8280
    10.4375
     4.1027
```

## See Also

[LinearMixedModel](#) | [fitlme](#) | [fitlmematrix](#) | [predict](#)

---

# random

**Class:** NonLinearModel

Simulate responses for nonlinear regression model

## Syntax

```
ysim = random mdl
ysim = random mdl, Xnew
ysim = random mdl, Xnew, 'Weights', W
```

## Description

`ysim = random(mdl)` simulates responses from the fitted nonlinear model `mdl` at the original design points.

`ysim = random(mdl, Xnew)` simulates responses from the fitted nonlinear model `mdl` to the data in `Xnew`, adding random noise.

`ysim = random(mdl, Xnew, 'Weights', W)` simulates responses using the observation weights, `W`.

## Input Arguments

### **mdl**

Nonlinear regression model, constructed by `fitnlm`.

### **Xnew**

Points at which `mdl` predicts responses.

- If `Xnew` is a table or dataset array, it must contain the predictor names in `mdl`.

- If  $X_{\text{new}}$  is a numeric matrix, it must have the same number of variables (columns) as was used to create `mdl`. Furthermore, all variables used in creating `mdl` must be numeric.

#### **W**

Vector of real, positive value weights or a function handle.

- If you specify a vector, then it must have the same number of elements as the number of observations (or rows) in  $X_{\text{new}}$ .
- If you specify a function handle, the function must accept a vector of predicted response values as input, and returns a vector of real positive weights as output.

Given weights, `W`, `random` estimates the error variance at observation `i` by  $\text{MSE} * (1 / W(i))$ , where `MSE` is the mean squared error.

**Default:** No weights

## Output Arguments

#### **ysim**

Vector of predicted mean values at  $X_{\text{new}}$ , perturbed by random noise. The noise is independent, normally distributed, with mean zero, and variance equal to the estimated error variance of the model.

## Examples

### **Simulate Responses**

Create a nonlinear model of car mileage as a function of weight, and simulate the response.

Create an exponential model of car mileage as a function of weight from the `carsmall` data. Scale the weight by a factor of 1000 so all the variables are roughly equal in size.

```
load carsmall
X = Weight;
```

---

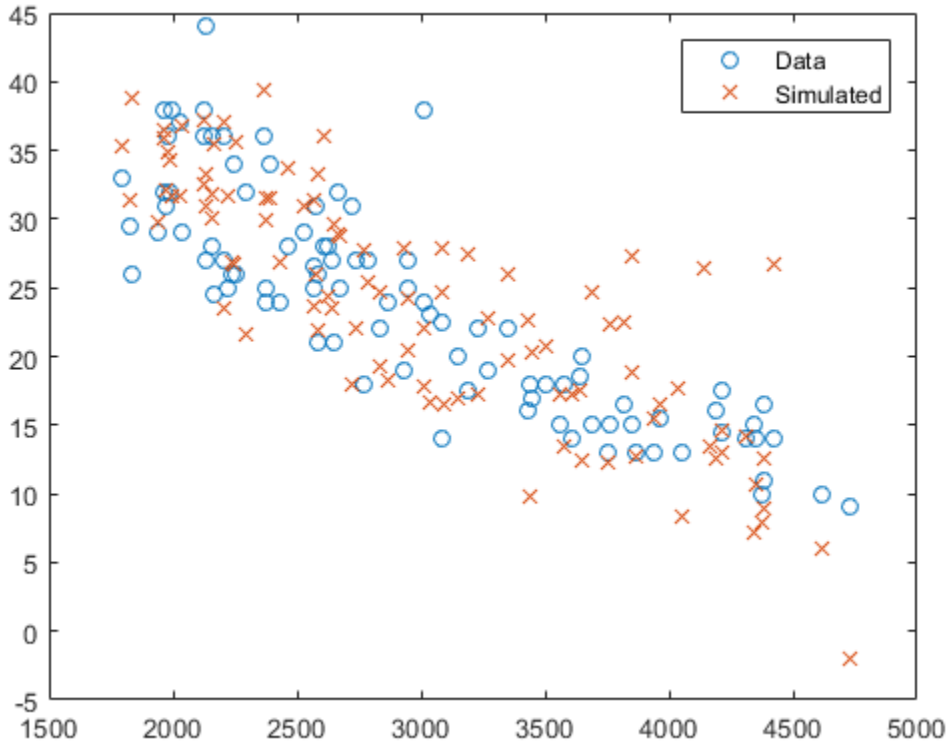
```
y = MPG;
modelfun = 'y ~ b1 + b2*exp(-b3*x/1000)';
beta0 = [1 1 1];
mdl = fitnlm(X,y,modelfun,beta0);
```

Create simulated responses to the data.

```
Xnew = X;
ysim = random(mdl,Xnew);
```

Plot the original responses and the simulated responses to see how they differ.

```
plot(X,y,'o',X,ysim,'x')
legend('Data','Simulated')
```



- “Predict or Simulate Responses Using a Nonlinear Model” on page 13-10

## Alternatives

For predictions without added noise, use `predict`.

## See Also

`NonLinearModel` | `feval` | `predict`



## Topics

“Predict or Simulate Responses Using a Nonlinear Model” on page 13-10

“Nonlinear Regression” on page 13-2

## random

**Class:** `piecewisedistribution`

Random numbers from piecewise distribution

### Syntax

```
r = random(obj)
R = random(obj, n)
R = random(obj, m, n)
R = random(obj, [m, n])
R = random(obj, m, n, p, ...)
R = random(obj, [m, n, p, ...])
```

### Description

`r = random(obj)` generates a pseudo-random number `r` drawn from the piecewise distribution object `obj`.

`R = random(obj, n)` generates an  $n$ -by- $n$  matrix of pseudo-random numbers `R`.

`R = random(obj, m, n)` or `R = random(obj, [m, n])` generates an  $m$ -by- $n$  matrix of pseudo-random numbers `R`.

`R = random(obj, m, n, p, ...)` or `R = random(obj, [m, n, p, ...])` generates an  $m$ -by- $n$ -by- $p$ -by-... array of pseudo-random numbers `R`.

### Examples

Fit Pareto tails to a  $t$  distribution at cumulative probabilities 0.1 and 0.9:

```
t = trnd(3, 100, 1);
obj = paretotails(t, 0.1, 0.9);

r = random(obj)
```

r =  
0.8285

## See Also

[cdf](#) | [icdf](#) | [paretotails](#)

## random

**Class:** ProbDist

Generate random number drawn from ProbDist object

### Syntax

```
Y = random(PD)
Y = random(PD, N)
Y = random(PD, N, M, ...)
```

### Description

`Y = random(PD)` generates a random number drawn from the distribution specified by `PD`, a ProbDist object.

`Y = random(PD, N)` generates an  $N$ -by- $N$  array of random numbers drawn from the distribution specified by `PD`, a ProbDist object.

`Y = random(PD, N, M, ...)` generates an  $N$ -by- $M$ -by... array of random numbers drawn from the distribution specified by `PD`, a ProbDist object.

### Input Arguments

<i>PD</i>	An object of the class ProbDistUnivParam or ProbDistUnivKernel.
<i>N</i>	A positive integer.
<i>M</i>	A positive integer.

### Output Arguments

<i>Y</i>	A random number drawn from the distribution specified by <code>PD</code> .
----------	--

## See Also

random

## random

**Class:** RepeatedMeasuresModel

Generate new random response values given predictor values

## Syntax

```
ysim = random(rm, tnew)
```

## Description

`ysim = random(rm, tnew)` generates random response values from the repeated measures model `rm` using the predictor variables from table `tnew`.

## Input Arguments

**rm — Repeated measures model**

RepeatedMeasuresModel object

Repeated measures model, returned as a RepeatedMeasuresModel object.

For properties and methods of this object, see RepeatedMeasuresModel.

**tnew — New data**

table used to create `rm` (default) | table

New data including the values of the response variables and the between-subject factors used as predictors in the repeated measures model, `rm`, specified as a table. `tnew` must contain all of the between-subject factors used to create `rm`.

## Output Arguments

**ysim** — Random response values

*n*-by-*r* matrix

Random response values `random` generates, returned as an *n*-by-*r* matrix, where *n* is the number of rows in `tnew`, and *r* is the number of repeated measures in `rm`.

## Examples

### Randomly Generate New Response Values

Load the sample data.

```
load fisheriris
```

The column vector `species` consists of iris flowers of three different species: `setosa`, `versicolor`, and `virginica`. The double matrix `meas` consists of four types of measurements on the flowers: the length and width of sepals and petals in centimeters, respectively.

Store the data in a table array.

```
t = table(species,meas(:,1),meas(:,2),meas(:,3),meas(:,4),...
'VariableNames',{'species','meas1','meas2','meas3','meas4'});
Meas = dataset([1 2 3 4]','VarNames',{'Measurements'});
```

Fit a repeated measures model, where the measurements are the responses and the species is the predictor variable.

```
rm = fitrm(t,'meas1-meas4~species','WithinDesign',Meas);
```

Randomly generate new response values.

```
ysim = random(rm);
```

`random` uses the predictor values in the original sample data you use to fit the repeated measures model `rm` in table `t`.

### Randomly Generate Response Values Using New Data

Load the sample data.

```
load repeatedmeas
```

The table `between` includes the between-subject variables `age`, `IQ`, `group`, `gender`, and eight repeated measures `y1` through `y8` as responses. The table `within` includes the within-subject variables `w1` and `w2`. This is simulated data.

Fit a repeated measures model, where the repeated measures `y1` through `y8` are the responses, and `age`, `IQ`, `group`, `gender`, and the group-gender interaction are the predictor variables. Also specify the within-subject design matrix.

```
rm = fitrm(between, 'y1-y8 ~ Group*Gender + Age + IQ', 'WithinDesign', within);
```

Define a table with new values for the predictor variables.

```
tnew = table(16, 93, {'B'}, {'Male'}, 'VariableNames', {'Age', 'IQ', 'Group', 'Gender'})
```

```
tnew =
```

```
1x4 table
```

Age	IQ	Group	Gender
16	93	'B'	'Male'

Randomly generate new response values using the values in the new table `tnew`.

```
ysim = random(rm, tnew)
```

```
ysim =
```

```
Columns 1 through 7
```

```
46.2252    66.8003   -40.4987   -1.9930    27.5213   -37.9809    4.8905
```

```
Column 8
```



-3.7568

## Algorithms

`random` computes `ysim` by creating predicted values and adding random noise values. For each row, the noise has a multivariate normal distribution with covariance the same as `rm.Covariance`.

## See Also

`fitrm` | `predict`

## random

**Class:** `prob.TruncatableDistribution`

**Package:** `prob`

Generate random numbers from probability distribution object

### Syntax

```
r = random(pd)
r = random(pd, sz1, ..., szN)
r = random(pd, [sz1, ..., szN])
```

### Description

`r = random(pd)` generates a random number `r` from the probability distribution `pd`.

`r = random(pd, sz1, ..., szN)` generates a `sz1-by-...-by-szN` array of random numbers from the probability distribution `pd`.

`r = random(pd, [sz1, ..., szN])` generates a `sz1-by-...-by-szN` array of random numbers from the probability distribution `pd`.

### Input Arguments

**pd — Probability distribution**

probability distribution object

Probability distribution, specified as a probability distribution object. Create a probability distribution object with specified parameter values using `makedist`. Alternatively, for fittable distributions, create a probability distribution object by fitting it to data using `fitdist` or the Distribution Fitting app.

**sz1, ..., szN — Size of each dimension**

two or more integer values | vector of integer values

Size of each dimension, specified as two or more integer values, or a vector of such values. For example, specifying `5, 3, 2` or `[5, 3, 2]` generates a 5-by-3-by-2 array of random numbers from the probability distribution `pd`.

Data Types: `single` | `double`

## Output Arguments

**r** — Random number

scalar value | array of values

Random number generated from the probability distribution, returned as a scalar value or an array of scalar values with the dimensions specified by `sz1, ..., szN`.

## Examples

### Generate One Random Number

Create a standard normal probability distribution object.

```
pd = makedist('Normal')
```

```
pd =
```

```
NormalDistribution
```

```
Normal distribution
```

```
mu = 0
```

```
sigma = 1
```

Generate one random number from the distribution.

```
r = random(pd)
```

```
r =
```

```
0.5377
```

### Generate Multiple Random Numbers

Create a Weibull probability distribution object using the default parameter values.

```
pd = makedist('Weibull')
```

```
pd =
```

```
WeibullDistribution
```

```
Weibull distribution
```

```
A = 1
```

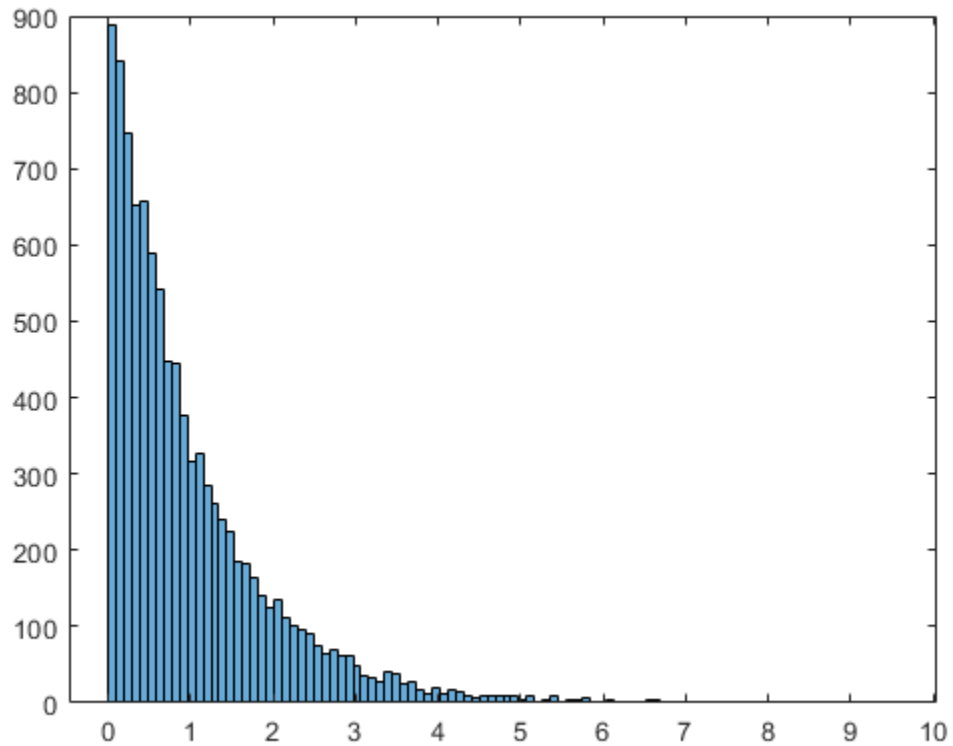
```
B = 1
```

Generate random numbers from distribution and visualize with a histogram.

```
rng default % For reproducibility
```

```
r = random(pd,10000,1);
```

```
histogram(r,100)
```



### Generate a Multidimensional Array of Random Numbers

Create a standard normal probability distribution object.

```
pd = makedist('Normal')
```

```
pd =
```

```
NormalDistribution
```

```
Normal distribution
```

```
mu = 0  
sigma = 1
```

Generate a 2-by-3-by-2 array of random numbers from the distribution.

```
r = random(pd, [2, 3, 2])
```

```
r(:, :, 1) =
```

```
0.5377    -2.2588    0.3188  
1.8339     0.8622   -1.3077
```

```
r(:, :, 2) =
```

```
-0.4336    3.5784   -1.3499  
0.3426    2.7694    3.0349
```

## See Also

[distributionFitter](#) | [fitdist](#) | [makedist](#)

# randomEffects

**Class:** GeneralizedLinearMixedModel

Estimates of random effects and related statistics

## Syntax

```
B = randomEffects(glme)
[B, BNames] = randomEffects(glme)
[B, BNames, stats] = randomEffects(glme)
[B, BNames, stats] = randomEffects(glme, Name, Value)
```

## Description

`B = randomEffects(glme)` returns the estimates of the empirical Bayes predictors (EPBs) of random effects in the generalized linear mixed-effects model `glme` conditional on the estimated covariance parameters and the observed response.

`[B, BNames] = randomEffects(glme)` also returns the names of the coefficients, `BNames`. Each name corresponds to a coefficient in `B`.

`[B, BNames, stats] = randomEffects(glme)` also returns related statistics, `stats`, for the estimated EBPs of random effects in `glme`.

`[B, BNames, stats] = randomEffects(glme, Name, Value)` returns any of the above output arguments using additional options specified by one or more `Name, Value` pair arguments. For example, you can specify the confidence interval level, or the method for computing the approximate degrees of freedom.

## Input Arguments

**glme** — Generalized linear mixed-effects model

GeneralizedLinearMixedModel object

Generalized linear mixed-effects model, specified as a `GeneralizedLinearMixedModel` object. For properties and methods of this object, see `GeneralizedLinearMixedModel`.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### **Alpha** — Confidence level

0.05 (default) | scalar value in the range [0,1]

Confidence level, specified as the comma-separated pair consisting of 'Alpha' and a scalar value in the range [0,1]. For a value  $\alpha$ , the confidence level is  $100 \times (1 - \alpha)\%$ .

For example, for 99% confidence intervals, you can specify the confidence level as follows.

Example: 'Alpha', 0.01

Data Types: `single` | `double`

### **DFMethod** — Method for computing approximate degrees of freedom

'residual' (default) | 'none'

Method for computing approximate degrees of freedom, specified as the comma-separated pair consisting of 'DFMethod' and one of the following.

Value	Description
'residual'	The degrees of freedom are assumed to be constant and equal to $n - p$ , where $n$ is the number of observations and $p$ is the number of fixed effects.
'none'	All degrees of freedom are set to infinity.

Example: 'DFMethod', 'none'



## Output Arguments

### **B** — Estimated empirical Bayes predictors for the random effects

column vector

Estimated empirical Bayes predictors (EBPs) for the random effects in the generalized linear mixed-effects model `glme`, returned as a column vector. The EBPs in `B` are approximated by the mode of the empirical posterior distribution of the random effects given the estimated covariance parameters and the observed response.

Suppose `glme` has  $R$  grouping variables  $g_1, g_2, \dots, g_R$ , with levels  $m_1, m_2, \dots, m_R$ , respectively. Also suppose  $q_1, q_2, \dots, q_R$  are the lengths of the random-effects vectors that are associated with  $g_1, g_2, \dots, g_R$ , respectively. Then, `B` is a column vector of length  $q_1 * m_1 + q_2 * m_2 + \dots + q_R * m_R$ .

`randomEffects` creates `B` by concatenating the empirical Bayes predictors of random-effects vectors corresponding to each level of each grouping variable as `[g1level1; g1level2; ...; g1levelm1; g2level1; g2level2; ...; g2levelm2; ...; gRlevel1; gRlevel2; ...; gRlevelmR]'`.

### **BNames** — Names of random-effects coefficients

table

Names of random-effects coefficients in `B`, returned as a table.

### **stats** — Estimated empirical Bayes predictors and related statistics

table

Estimated empirical Bayes predictors (EBPs) and related statistics for the random effects in the generalized linear mixed-effects model `glme`, returned as a table. `stats` has one row for each of the random effects, and one column for each of the following statistics.

Column Name	Description
Group	Grouping variable associated with the random effect
Level	Level within the grouping variable corresponding to the random effect
Name	Name of the random-effect coefficient

Column Name	Description
Estimate	Empirical Bayes predictor (EBP) of random effect
SEPred	Square root of the conditional mean squared error of prediction (CMSEP) given covariance parameters and response
tStat	<i>t</i> -statistic for a test that the random-effects coefficient is equal to 0
DF	Estimated degrees of freedom for the <i>t</i> -statistic
pValue	<i>p</i> -value for the <i>t</i> -statistic
Lower	Lower limit of a 95% confidence interval for the random-effects coefficient
Upper	Upper limit of a 95% confidence interval for the random-effects coefficient

`randomEffects` computes the confidence intervals using the conditional mean squared error of prediction (CMSEP) approach conditional on the estimated covariance parameters and the observed response. An alternative interpretation of the confidence intervals is that they are approximate Bayesian credible intervals conditional on the estimated covariance parameters and the observed response.

When fitting a GLME model using `fitglme` and one of the pseudo likelihood fit methods ('MPL' or 'REMP'), `randomEffects` computes confidence intervals and related statistics based on the fitted linear mixed-effects model from the final pseudo likelihood iteration.

## Examples

### Compute and Plot Estimated Random Effects

Load the sample data.

```
load mfr
```

This simulated data is from a manufacturing company that operates 50 factories across the world, with each factory running a batch process to create a finished product. The company wants to decrease the number of defects in each batch, so it developed a new manufacturing process. To test the effectiveness of the new process, the company selected 20 of its factories at random to participate in an experiment: Ten factories implemented the new process, while the other ten continued to run the old process. In each of the 20 factories, the company ran five batches (for a total of 100 batches) and recorded the following data:

- Flag to indicate whether the batch used the new process (`newprocess`)
- Processing time for each batch, in hours (`time`)
- Temperature of the batch, in degrees Celsius (`temp`)
- Categorical variable indicating the supplier (A, B, or C) of the chemical used in the batch (`supplier`)
- Number of defects in the batch (`defects`)

The data also includes `time_dev` and `temp_dev`, which represent the absolute deviation of time and temperature, respectively, from the process standard of 3 hours at 20 degrees Celsius.

Fit a generalized linear mixed-effects model using `newprocess`, `time_dev`, `temp_dev`, and `supplier` as fixed-effects predictors. Include a random-effects term for intercept grouped by `factory`, to account for quality differences that might exist due to factory-specific variations. The response variable `defects` has a Poisson distribution, and the appropriate link function for this model is `log`. Use the Laplace fit method to estimate the coefficients. Specify the dummy variable encoding as `'effects'`, so the dummy variable coefficients sum to 0.

The number of defects can be modeled using a Poisson distribution

$$\text{defects}_{ij} \sim \text{Poisson}(\mu_{ij})$$

This corresponds to the generalized linear mixed-effects model

$$\log(\mu_{ij}) = \beta_0 + \beta_1 \text{newprocess}_{ij} + \beta_2 \text{time\_dev}_{ij} + \beta_3 \text{temp\_dev}_{ij} + \beta_4 \text{supplier\_C}_{ij} + \beta_5 \text{supplier\_B}_{ij} + b_i,$$

where

- `defectsij` is the number of defects observed in the batch produced by factory  $i$  during batch  $j$ .

- $\mu_{ij}$  is the mean number of defects corresponding to factory  $i$  (where  $i = 1, 2, \dots, 20$ ) during batch  $j$  (where  $j = 1, 2, \dots, 5$ ).
- `newprocessij`, `time_devij`, and `temp_devij` are the measurements for each variable that correspond to factory  $i$  during batch  $j$ . For example, `newprocessij` indicates whether the batch produced by factory  $i$  during batch  $j$  used the new process.
- `supplier_Cij` and `supplier_Bij` are dummy variables that use effects (sum-to-zero) coding to indicate whether company C or B, respectively, supplied the process chemicals for the batch produced by factory  $i$  during batch  $j$ .
- $b_i \sim N(0, \sigma_b^2)$  is a random-effects intercept for each factory  $i$  that accounts for factory-specific variation in quality.

```
glme = fitglme(mfr, 'defects ~ 1 + newprocess + time_dev + temp_dev + supplier + (1|fact
```

Compute and display the names and estimated values of the empirical Bayes predictors (EBPs) for the random effects.

```
[B,BNames] = randomEffects(glme)
```

```
B =
```

```
    0.2913  
    0.1542  
   -0.2633  
   -0.4257  
    0.5453  
   -0.1069  
    0.3040  
   -0.1653  
   -0.1458  
   -0.0816  
    0.0145  
    0.1771  
    0.2487  
    0.2115  
    0.2777  
   -0.2518  
   -0.1351  
   -0.1627  
   -0.3208  
    0.0584
```

```
BNames =
```

```
20x3 table
```

Group	Level	Name
'factory'	'1'	'(Intercept)'
'factory'	'2'	'(Intercept)'
'factory'	'3'	'(Intercept)'
'factory'	'4'	'(Intercept)'
'factory'	'5'	'(Intercept)'
'factory'	'6'	'(Intercept)'
'factory'	'7'	'(Intercept)'
'factory'	'8'	'(Intercept)'
'factory'	'9'	'(Intercept)'
'factory'	'10'	'(Intercept)'
'factory'	'11'	'(Intercept)'
'factory'	'12'	'(Intercept)'
'factory'	'13'	'(Intercept)'
'factory'	'14'	'(Intercept)'
'factory'	'15'	'(Intercept)'
'factory'	'16'	'(Intercept)'
'factory'	'17'	'(Intercept)'
'factory'	'18'	'(Intercept)'
'factory'	'19'	'(Intercept)'
'factory'	'20'	'(Intercept)'

Each row of **B** contains the estimated EPB for the random-effects coefficient named in the corresponding row of **BNames**. For example, the value  $-0.2633$  in row 3 of **B** is the estimated EPB for '(Intercept) ' for level '3' of factory.

### Compute 99% Confidence Intervals for Random Effects

Load the sample data.

```
load mfr
```

This simulated data is from a manufacturing company that operates 50 factories across the world, with each factory running a batch process to create a finished product. The

company wants to decrease the number of defects in each batch, so it developed a new manufacturing process. To test the effectiveness of the new process, the company selected 20 of its factories at random to participate in an experiment: Ten factories implemented the new process, while the other ten continued to run the old process. In each of the 20 factories, the company ran five batches (for a total of 100 batches) and recorded the following data:

- Flag to indicate whether the batch used the new process (`newprocess`)
- Processing time for each batch, in hours (`time`)
- Temperature of the batch, in degrees Celsius (`temp`)
- Categorical variable indicating the supplier (A, B, or C) of the chemical used in the batch (`supplier`)
- Number of defects in the batch (`defects`)

The data also includes `time_dev` and `temp_dev`, which represent the absolute deviation of time and temperature, respectively, from the process standard of 3 hours at 20 degrees Celsius.

Fit a generalized linear mixed-effects model using `newprocess`, `time_dev`, `temp_dev`, and `supplier` as fixed-effects predictors. Include a random-effects term for intercept grouped by `factory`, to account for quality differences that might exist due to factory-specific variations. The response variable `defects` has a Poisson distribution, and the appropriate link function for this model is `log`. Use the Laplace fit method to estimate the coefficients. Specify the dummy variable encoding as `'effects'`, so the dummy variable coefficients sum to 0.

The number of defects can be modeled using a Poisson distribution  
 $\text{defects}_{ij} \sim \text{Poisson}(\mu_{ij})$

This corresponds to the generalized linear mixed-effects model

$$\log(\mu_{ij}) = \beta_0 + \beta_1 \text{newprocess}_{ij} + \beta_2 \text{time.dev}_{ij} + \beta_3 \text{temp.dev}_{ij} + \beta_4 \text{supplier.C}_{ij} + \beta_5 \text{supplier.B}_{ij} + b_i,$$

where

- $\text{defects}_{ij}$  is the number of defects observed in the batch produced by factory  $i$  during batch  $j$ .
- $\mu_{ij}$  is the mean number of defects corresponding to factory  $i$  (where  $i = 1, 2, \dots, 20$ ) during batch  $j$  (where  $j = 1, 2, \dots, 5$ ).

- $\text{newprocess}_{ij}$ ,  $\text{time\_dev}_{ij}$ , and  $\text{temp\_dev}_{ij}$  are the measurements for each variable that correspond to factory  $i$  during batch  $j$ . For example,  $\text{newprocess}_{ij}$  indicates whether the batch produced by factory  $i$  during batch  $j$  used the new process.
- $\text{supplier\_C}_{ij}$  and  $\text{supplier\_B}_{ij}$  are dummy variables that use effects (sum-to-zero) coding to indicate whether company C or B, respectively, supplied the process chemicals for the batch produced by factory  $i$  during batch  $j$ .
- $b_i \sim N(0, \sigma_b^2)$  is a random-effects intercept for each factory  $i$  that accounts for factory-specific variation in quality.

```
glme = fitglme(mfr, 'defects ~ 1 + newprocess + time_dev + temp_dev + supplier + (1|factory)
  Distribution', 'Poisson', 'Link', 'log', 'FitMethod', 'Laplace', 'DummyVarCoding', 'effects')
```

Compute and display the 99% confidence intervals for the random-effects coefficients.

```
[B, BNames, stats] = randomEffects(glme, 'Alpha', 0.01);
stats
```

```
stats =
```

```
RANDOM EFFECT COEFFICIENTS: DFMETHOD = 'RESIDUAL', ALPHA = 0.01
```

Group	Level	Name	Estimate	SEPred
'factory'	'1'	'(Intercept)'	0.29131	0.19163
'factory'	'2'	'(Intercept)'	0.15423	0.19216
'factory'	'3'	'(Intercept)'	-0.26325	0.21249
'factory'	'4'	'(Intercept)'	-0.42568	0.21667
'factory'	'5'	'(Intercept)'	0.5453	0.17963
'factory'	'6'	'(Intercept)'	-0.10692	0.20133
'factory'	'7'	'(Intercept)'	0.30404	0.18397
'factory'	'8'	'(Intercept)'	-0.16527	0.20505
'factory'	'9'	'(Intercept)'	-0.14577	0.203
'factory'	'10'	'(Intercept)'	-0.081632	0.20256
'factory'	'11'	'(Intercept)'	0.014529	0.21421
'factory'	'12'	'(Intercept)'	0.17706	0.20721
'factory'	'13'	'(Intercept)'	0.24872	0.20522
'factory'	'14'	'(Intercept)'	0.21145	0.20678
'factory'	'15'	'(Intercept)'	0.2777	0.20345
'factory'	'16'	'(Intercept)'	-0.25175	0.22568
'factory'	'17'	'(Intercept)'	-0.13507	0.22301
'factory'	'18'	'(Intercept)'	-0.1627	0.22269

'factory'	'19'	'(Intercept)'	-0.32083	0.23294
'factory'	'20'	'(Intercept)'	0.058418	0.21481
tStat	DF	pValue	Lower	Upper
1.5202	94	0.13182	-0.21251	0.79514
0.80259	94	0.42423	-0.351	0.65946
-1.2389	94	0.21846	-0.82191	0.29541
-1.9646	94	0.052408	-0.99534	0.14398
3.0356	94	0.0031051	0.073019	1.0176
-0.53105	94	0.59664	-0.63625	0.42241
1.6527	94	0.10173	-0.17964	0.78771
-0.80597	94	0.42229	-0.70438	0.37385
-0.71806	94	0.4745	-0.67949	0.38795
-0.403	94	0.68786	-0.61419	0.45093
0.067826	94	0.94607	-0.54866	0.57772
0.85446	94	0.39502	-0.36774	0.72185
1.212	94	0.22857	-0.29083	0.78827
1.0226	94	0.30913	-0.33221	0.75511
1.365	94	0.17552	-0.25719	0.81259
-1.1156	94	0.26746	-0.84509	0.34158
-0.60568	94	0.54619	-0.7214	0.45125
-0.73061	94	0.46684	-0.74817	0.42278
-1.3773	94	0.17168	-0.93325	0.29159
0.27195	94	0.78626	-0.50635	0.62319

The first three columns of `stats` contain the group name, level, and random-effects coefficient name. Column 4 contains the estimated EBP of the random-effects coefficient. The last two columns of `stats`, `Lower` and `Upper`, contain the lower and upper bounds of the 99% confidence interval, respectively. For example, for the coefficient for `'(Intercept)'` for level 3 of `factory`, the estimated EBP is -0.26325, and the 99% confidence interval is [-0.82191,0.29541].

## References

- [1] Booth, J.G., and J.P. Hobert. “Standard Errors of Prediction in Generalized Linear Mixed Models.” *Journal of the American Statistical Association*, Vol. 93, 1998, pp. 262–272.



## See Also

`GeneralizedLinearMixedModel` | `coefCI` | `coefTest` | `fixedEffects`

## randomEffects

**Class:** LinearMixedModel

Estimates of random effects and related statistics

### Syntax

```
B = randomEffects(lme)
[B, Bnames] = randomEffects(lme)
[B, Bnames, stats] = randomEffects(lme)
[B, Bnames, stats] = randomEffects(lme, Name, Value)
```

### Description

`B = randomEffects(lme)` returns the estimates of the best linear unbiased predictors (BLUPs) of random effects in the linear mixed-effects model `lme`.

`[B, Bnames] = randomEffects(lme)` also returns the names of the coefficients in `Bnames`. Each name corresponds to a coefficient in `B`.

`[B, Bnames, stats] = randomEffects(lme)` also returns the estimated BLUPs of random effects in the linear mixed-effects model `lme` and related statistics.

`[B, Bnames, stats] = randomEffects(lme, Name, Value)` also returns the BLUPs of random effects in the linear mixed-effects model `lme` and related statistics with additional options specified by one or more `Name, Value` pair arguments.

### Input Arguments

**lme** — Linear mixed-effects model

LinearMixedModel object

Linear mixed-effects model, specified as a LinearMixedModel object constructed using `fitlme` or `fitlmematrix`.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### **Alpha** — Confidence level

0.05 (default) | scalar value in the range 0 to 1

Confidence level, specified as the comma-separated pair consisting of 'Alpha' and a scalar value in the range 0 to 1. For a value  $\alpha$ , the confidence level is  $100*(1-\alpha)\%$ .

For example, for 99% confidence intervals, you can specify the confidence level as follows.

Example: 'Alpha', 0.01

Data Types: single | double

### **DFMethod** — Method for computing approximate degrees of freedom

'residual' (default) | 'satterthwaite' | 'none'

Method for computing approximate degrees of freedom for the  $t$ -statistics that test the random-effects coefficients against 0, specified as the comma-separated pair consisting of 'DFMethod' and one of the following.

'residual'

Default. The degrees of freedom are assumed to be constant and equal to  $n - p$ , where  $n$  is the number of observations and  $p$  is the number of fixed effects.

'satterthwaite'

Satterthwaite approximation.

'none'

All degrees of freedom are set to infinity.

For example, you can specify the Satterthwaite approximation as follows.

Example: 'DFMethod', 'satterthwaite'

## Output Arguments

### **B** — Estimated best linear unbiased predictors of random effects

column vector

Estimated best linear unbiased predictors of random effects of linear mixed-effects model `lme`, returned as a column vector.

Suppose `lme` has  $R$  grouping variables  $g_1, g_2, \dots, g_R$ , with levels  $m_1, m_2, \dots, m_R$ , respectively. Also suppose  $q_1, q_2, \dots, q_R$  are the lengths of the random-effects vectors that are associated with  $g_1, g_2, \dots, g_R$ , respectively. Then, `B` is a column vector of length  $q_1 * m_1 + q_2 * m_2 + \dots + q_R * m_R$ .

`randomEffects` creates `B` by concatenating the best linear unbiased predictors of random-effects vectors corresponding to each level of each grouping variable as `[g1level1; g1level2; ...; g1levelm1; g2level1; g2level2; ...; g2levelm2; ...; gRlevel1; gRlevel2; ...; gRlevelmR]'`.

### **Bnames** — Names of random-effects coefficients

table

Names of random-effects coefficients in `B`, returned as a table.

### **stats** — Estimates of random effects BLUPs and related statistics

dataset array

Estimates of random effects BLUPs and related statistics, returned as a dataset array that has one row for each of the fixed effects and one column for each of the following statistics.

Group	Grouping variable associated with the random effect
Level	Level within the grouping variable corresponding to the random effect
Name	Name of the random-effect coefficient
Estimate	Best linear unbiased predictor (BLUP) of random effect
SEPred	Standard error of the estimate (BLUP minus random effect)
tStat	$t$ -statistic for a test that the random effect is zero
DF	Estimated degrees of freedom for the $t$ -statistic
pValue	$p$ -value for the $t$ -statistic

Lower	Lower limit of a 95% confidence interval for the random effect
Upper	Upper limit of a 95% confidence interval for the random effect

## Examples

### Display Random-Effects Estimates and Coefficient Names

Load the sample data.

```
load carbig
```

Fit a linear mixed-effects model for miles per gallon (MPG), with fixed effects for acceleration and horsepower, and potentially correlated random effects for intercept and acceleration, grouped by the model year. First, store the data in a table.

```
tbl = table(Acceleration, Horsepower, Model_Year, MPG);
```

Fit the model.

```
lme = fitlme(tbl, 'MPG ~ Acceleration + Horsepower + (Acceleration|Model_Year)');
```

Compute the BLUPs of the random-effects coefficients and display the names of the corresponding random effects.

```
[B, Bnames] = randomEffects(lme)
```

```
B =
```

```

    3.1270
   -0.2426
   -1.6532
   -0.0086
    1.2075
   -0.2179
    4.4107
   -0.4887
   -1.3103
   -0.0208
```

2.8029  
 -0.3790  
 0.0865  
 -0.1280  
 0.4216  
 -0.0259  
 -2.3889  
 0.1634  
 0.9618  
 0.0117  
 -2.2345  
 0.5020  
 -2.1332  
 0.3254  
 -3.2979  
 0.5090

Bnames =

26x3 table

Group	Level	Name
'Model_Year'	'70'	'(Intercept)'
'Model_Year'	'70'	'Acceleration'
'Model_Year'	'71'	'(Intercept)'
'Model_Year'	'71'	'Acceleration'
'Model_Year'	'72'	'(Intercept)'
'Model_Year'	'72'	'Acceleration'
'Model_Year'	'73'	'(Intercept)'
'Model_Year'	'73'	'Acceleration'
'Model_Year'	'74'	'(Intercept)'
'Model_Year'	'74'	'Acceleration'
'Model_Year'	'75'	'(Intercept)'
'Model_Year'	'75'	'Acceleration'
'Model_Year'	'76'	'(Intercept)'
'Model_Year'	'76'	'Acceleration'
'Model_Year'	'77'	'(Intercept)'
'Model_Year'	'77'	'Acceleration'
'Model_Year'	'78'	'(Intercept)'
'Model_Year'	'78'	'Acceleration'
'Model_Year'	'79'	'(Intercept)'

```

'Model_Year'      '79'      'Acceleration'
'Model_Year'      '80'      '(Intercept)'
'Model_Year'      '80'      'Acceleration'
'Model_Year'      '81'      '(Intercept)'
'Model_Year'      '81'      'Acceleration'
'Model_Year'      '82'      '(Intercept)'
'Model_Year'      '82'      'Acceleration'

```

Since intercept and acceleration have potentially correlated random effects, grouped by model year of the cars, `randomEffects` creates a separate row for intercept and acceleration at each level of the grouping variable.

Compute the covariance parameters of the random effects.

```

[~,~,stats] = covarianceParameters(lme)
stats{1}

```

```
stats =
```

```
2x1 cell array
```

```

{3x7 classreg.regr.lmeutils.titleddataset}
{1x5 classreg.regr.lmeutils.titleddataset}

```

```
ans =
```

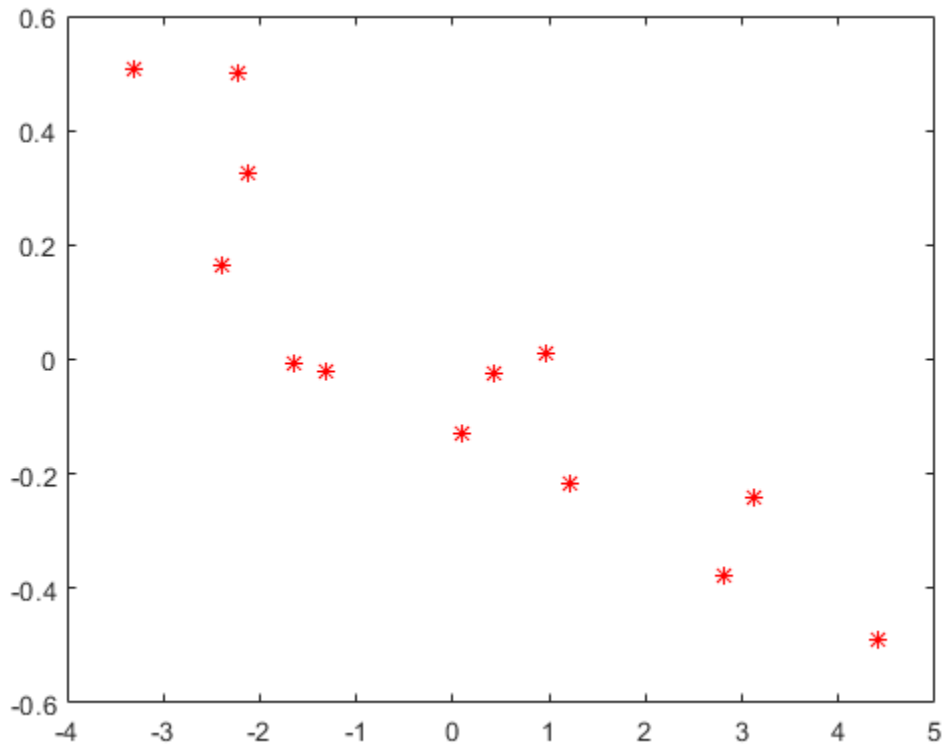
```
COVARIANCE TYPE: FULLCHOLESKY
```

Group	Name1	Name2	Type
Model_Year	'(Intercept)'	'(Intercept)'	'std'
Model_Year	'Acceleration'	'(Intercept)'	'corr'
Model_Year	'Acceleration'	'Acceleration'	'std'

Estimate	Lower	Upper
3.3475	1.2862	8.7119
-0.87971	-0.98501	-0.29675
0.33789	0.1825	0.62558

The correlation value suggests that random effects seem negatively correlated. Plot the random effects for intercept versus acceleration to confirm this.

```
plot(B(1:2:end),B(2:2:end),'r*')
```



### Compute Random-Effects Estimates and Related Statistics

Load the sample data.

```
load(fullfile(matlabroot,'examples','stats','fertilizer.mat'));
```

The dataset array includes data from a split-plot experiment, where soil is divided into three blocks based on the soil type: sandy, silty, and loamy. Each block is divided into



five plots, where five different types of tomato plants (cherry, heirloom, grape, vine, and plum) are randomly assigned to these plots. The tomato plants in the plots are then divided into subplots, where each subplot is treated by one of four fertilizers. This is simulated data.

Store the data in a dataset array called `ds`, for practical purposes, and define `Tomato`, `Soil`, and `Fertilizer` as categorical variables.

```
ds = fertilizer;
ds.Tomato = nominal(ds.Tomato);
ds.Soil = nominal(ds.Soil);
ds.Fertilizer = nominal(ds.Fertilizer);
```

Fit a linear mixed-effects model, where `Fertilizer` and `Tomato` are the fixed-effects variables, and the mean yield varies by the block (soil type), and the plots within blocks (tomato types within soil types) independently.

```
lme = fitlme(ds, 'Yield ~ Fertilizer * Tomato + (1|Soil) + (1|Soil:Tomato)');
```

Compute the BLUPs and related statistics for random effects.

```
[~,~,stats] = randomEffects(lme)
```

```
stats =
```

```
RANDOM EFFECT COEFFICIENTS: DFMETHOD = 'RESIDUAL', ALPHA = 0.05
```

Group	Level	Name	Estimate
'Soil'	'Loamy'	'(Intercept)'	1.0061
'Soil'	'Sandy'	'(Intercept)'	-1.5236
'Soil'	'Silty'	'(Intercept)'	0.51744
'Soil:Tomato'	'Loamy Cherry'	'(Intercept)'	12.46
'Soil:Tomato'	'Loamy Grape'	'(Intercept)'	-2.6429
'Soil:Tomato'	'Loamy Heirloom'	'(Intercept)'	16.681
'Soil:Tomato'	'Loamy Plum'	'(Intercept)'	-5.0172
'Soil:Tomato'	'Loamy Vine'	'(Intercept)'	-4.6874
'Soil:Tomato'	'Sandy Cherry'	'(Intercept)'	-17.393
'Soil:Tomato'	'Sandy Grape'	'(Intercept)'	-7.3679
'Soil:Tomato'	'Sandy Heirloom'	'(Intercept)'	-8.621
'Soil:Tomato'	'Sandy Plum'	'(Intercept)'	7.669
'Soil:Tomato'	'Sandy Vine'	'(Intercept)'	0.28246
'Soil:Tomato'	'Silty Cherry'	'(Intercept)'	4.9326

'Soil:Tomato'	'Silty Grape'	'(Intercept)'	10.011
'Soil:Tomato'	'Silty Heirloom'	'(Intercept)'	-8.0599
'Soil:Tomato'	'Silty Plum'	'(Intercept)'	-2.6519
'Soil:Tomato'	'Silty Vine'	'(Intercept)'	4.405

SEPred	tStat	DF	pValue	Lower	Upper
2.3374	0.43044	40	0.66918	-3.718	5.7303
2.3374	-0.65181	40	0.51825	-6.2477	3.2006
2.3374	0.22137	40	0.82593	-4.2067	5.2416
7.1765	1.7362	40	0.090224	-2.0443	26.964
7.1765	-0.36827	40	0.71461	-17.147	11.861
7.1765	2.3244	40	0.025269	2.1766	31.185
7.1765	-0.69911	40	0.48853	-19.522	9.4872
7.1765	-0.65316	40	0.51739	-19.192	9.8169
7.1765	-2.4235	40	0.019987	-31.897	-2.8882
7.1765	-1.0267	40	0.31075	-21.872	7.1364
7.1765	-1.2013	40	0.23671	-23.125	5.8833
7.1765	1.0686	40	0.29165	-6.8353	22.173
7.1765	0.039359	40	0.9688	-14.222	14.787
7.1765	0.68732	40	0.49585	-9.5718	19.437
7.1765	1.3949	40	0.17073	-4.4935	24.515
7.1765	-1.1231	40	0.2681	-22.564	6.4444
7.1765	-0.36952	40	0.71369	-17.156	11.852
7.1765	0.6138	40	0.54282	-10.099	18.909

The first three rows contain the random-effects estimates and the statistics for the three levels, Loamy, Sandy, and Silty of the grouping variable Soil. The corresponding  $P$ -values 0.66918, 0.51825, and 0.82593 indicate that these random-effects are not significantly different from 0. The following 15 rows include the BLUPS of random-effects estimates for the intercept, grouped by the variable Tomato nested in Soil, i.e. interaction of Tomato and Soil.

### Compute Confidence Intervals with Specified Options

Load the sample data.

```
load carsmall
```

Shift and Operator are nominal variables.

```
shift.Shift = nominal(shift.Shift);
shift.Operator = nominal(shift.Operator);
```

Fit a linear mixed-effects model with a random intercept grouped by operator, to assess if there is a significant difference in the performance according to the time of the shift. Use the restricted maximum likelihood method.

```
lme = fitlme(shift, 'QCDev ~ Shift + (1|Operator)');
```

Compute the 99% confidence intervals for random effects using the residuals option to compute the degrees of freedom. This is the default method.

```
[~,~,stats] = randomEffects(lme, 'alpha', 0.01)
```

```
stats =
```

```
Random effect coefficients: DFMethod = 'Residual', Alpha = 0.01
```

Group	Level	Name	Estimate	SEPred	tStat
'Operator'	'1'	'(Intercept)'	0.57753	0.90378	0.63902
'Operator'	'2'	'(Intercept)'	1.1757	0.90378	1.3009
'Operator'	'3'	'(Intercept)'	-2.1715	0.90378	-2.4027
'Operator'	'4'	'(Intercept)'	2.3655	0.90378	2.6174
'Operator'	'5'	'(Intercept)'	-1.9472	0.90378	-2.1546

Compute the 99% confidence intervals for random effects using the Satterthwaite approximation to compute the degrees of freedom.

```
[~,~,stats] = randomEffects(lme, 'DFMethod', 'satterthwaite', 'alpha', 0.01)
```

```
stats =
```

```
Random effect coefficients: DFMethod = 'Satterthwaite', Alpha = 0.01
```

Group	Level	Name	Estimate	SEPred	tStat
'Operator'	'1'	'(Intercept)'	0.57753	0.90378	0.63902
'Operator'	'2'	'(Intercept)'	1.1757	0.90378	1.3009
'Operator'	'3'	'(Intercept)'	-2.1715	0.90378	-2.4027
'Operator'	'4'	'(Intercept)'	2.3655	0.90378	2.6174
'Operator'	'5'	'(Intercept)'	-1.9472	0.90378	-2.1546

The Satterhwaite method usually produces smaller values for the degrees of freedom (DF), which results in larger  $p$ -values (`pValue`) and larger confidence intervals (`Lower` and `Upper`) for the random-effects estimates.

## See Also

`LinearMixedModel` | `coefCI` | `coefTest` | `fitlme` | `fixedEffects`

# randsample

Random sample

## Syntax

```
y = randsample(n,k)
y = randsample(population,k)
y = randsample(n,k,replacement)
y = randsample(population,k,replacement)
y = randsample(n,k,true,w)
y = randsample(population,k,true,w)
y = randsample(s,...)
```

## Description

`y = randsample(n,k)` returns a  $k$ -by-1 vector  $y$  of values sampled uniformly at random, without replacement, from the integers 1 to  $n$ .

`y = randsample(population,k)` returns a vector of  $k$  values sampled uniformly at random, without replacement, from the values in the vector `population`. The orientation of  $y$  (row or column) is the same as `population`.

`y = randsample(n,k,replacement)` or `y = randsample(population,k,replacement)` returns a sample taken with replacement if `replacement` is `true`, or without replacement if `replacement` is `false`. The default is `false`.

`y = randsample(n,k,true,w)` or `y = randsample(population,k,true,w)` returns a weighted sample taken with replacement, using a vector of positive weights  $w$ , whose length is  $n$ . The probability that the integer  $i$  is selected for an entry of  $y$  is  $w(i) / \text{sum}(w)$ . Usually,  $w$  is a vector of probabilities. `randsample` does not support weighted sampling without replacement.

`y = randstream(s, ...)` uses the stream `s` for random number generation. `s` is a member of the `RandStream` class. Default is the MATLAB default random number stream.

## Examples

Draw a single value from the integers 1 through 10:

```
n = 10;
x = randstream(n,1);
```

Draw a single value from the population 1 through `n`, where `n > 1`:

```
y = randstream(1:n,1);
```

---

**Note** If `population` is a numeric vector containing only nonnegative integer values, and `population` can have length 1, use

```
y = population(randstream(length(population),k))
```

instead of `y = randstream(population,k)`.

---

Generate a random sequence of the characters A, C, G, and T, with replacement, according to the specified probabilities.

```
R = randstream('ACGT',48,true,[0.15 0.35 0.35 0.15])
```

## Tips

- To randomly sample data, with or without replacement, use `datasample`.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- When sampling without replacement, the order of the output values might not match MATLAB.

### See Also

`RandStream` | `datasample` | `rand` | `randperm`

Introduced before R2006a

## randtool

Interactive random number generation

### Syntax

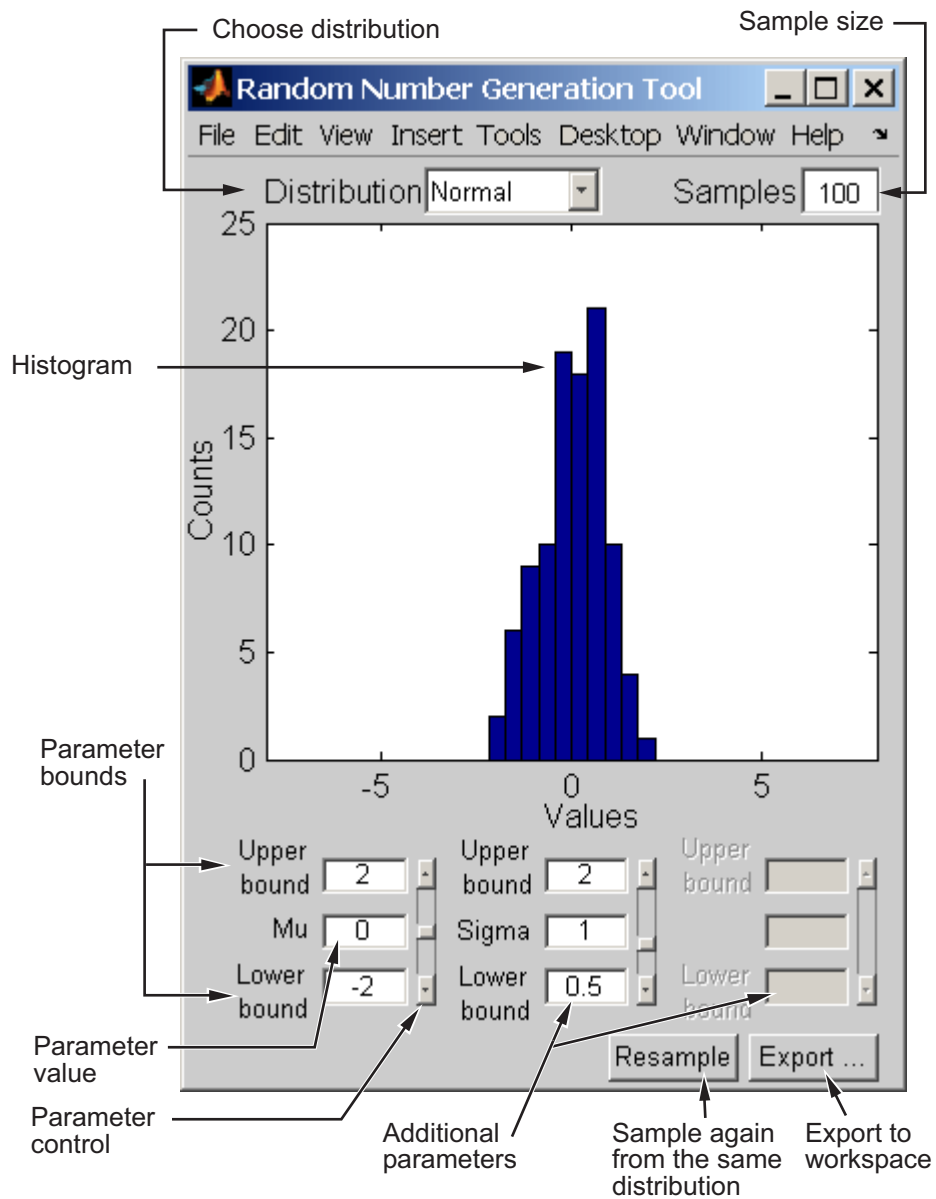
```
randtool
```

### Description

`randtool` opens the Random Number Generation Tool.

The Random Number Generation Tool is a graphical user interface that generates random samples from specified probability distributions and displays the samples as histograms. Use the tool to explore the effects of changing parameters and sample size on the distributions.





Start by selecting a distribution, then enter the desired sample size.

You can also

- Use the controls at the bottom of the window to set parameter values for the distribution and to change their upper and lower bounds.
- Draw another sample from the same distribution, with the same size and parameters.
- Export the current sample to your workspace. A dialog box enables you to provide a name for the sample.

**Introduced before R2006a**

## range

Range of values

## Syntax

```
range(X)  
y = range(X, dim)
```

## Description

`range(X)` returns the difference between the maximum and the minimum of a sample. For vectors, `range(x)` is the range of the elements. For matrices, `range(X)` is a row vector containing the range of each column of `X`. For N-dimensional arrays, `range` operates along the first nonsingleton dimension of `X`.

`y = range(X, dim)` operates along the dimension `dim` of `X`.

`range` treats NaNs as missing values and ignores them.

The range is an easily-calculated estimate of the spread of a sample. Outliers have an undue influence on this statistic, which makes it an unreliable estimator.

## Examples

The range of a large sample of standard normal random numbers is approximately six. This is the motivation for the process capability indices  $C_p$  and  $C_{pk}$  in statistical quality control applications.

```
rv = normrnd(0,1,1000,5);  
near6 = range(rv)  
near6 =  
    6.1451    6.4986    6.2909    5.8894    7.0002
```

## Extended Capabilities

### Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays” (MATLAB).

### See Also

`iqr` | `mad` | `std`

**Introduced before R2006a**

## rangesearch

Find all neighbors within specified distance using exhaustive search or *Kd*-tree

### Syntax

```
Idx = rangesearch(Mdl, Y, r)
Idx = rangesearch(Mdl, Y, r, Name, Value)
[Idx, D] = rangesearch( ___ )
```

### Description

`Idx = rangesearch(Mdl, Y, r)` searches for all neighbors (i.e., points, rows, or observations) in `Mdl.X` within radius `r` of each point (i.e., row or observation) in the query data `Y` using an exhaustive search or a *Kd*-tree. `rangesearch` returns `Idx`, which is a column vector of the indices of `Mdl.X` within `r` units.

`Idx = rangesearch(Mdl, Y, r, Name, Value)` returns the indices of the observation in `Mdl.X` within radius `r` of each observation in `Y` with additional options specified by one or more `Name, Value` pair arguments. For example, you can specify to use a different distance metric than is stored in `Mdl.Distance` or a different distance metric parameter than is stored in `Mdl.DistanceParameter`.

`[Idx, D] = rangesearch( ___ )` additionally returns the matrix `D` using any of the input arguments in the previous syntaxes. `D` contains the distances between the observations in `Mdl.X` within radius `r` of each observation in `Y`. The function arranges the columns of `D` in ascending order by closeness, with respect to the distance metric.

### Examples

#### Search for Neighbors Within A Radius Using a *Kd*-tree and Exhaustive Search

`rangesearch` accepts `ExhaustiveSearcher` or `KDTreeSearcher` model objects to search the training data for the nearest neighbors to the query data. An

ExhaustiveSearcher model invokes the exhaustive searcher algorithm, and a KDTreeSearcher model defines a  $K$  d-tree, which rangesearch uses to search for nearest neighbors.

Load Fisher's iris data set. Randomly reserve five observations from the data for query data. Focus on the petal dimensions.

```
load fisheriris
rng(1); % For reproducibility
n = size(meas,1);
idx = randsample(n,5);
X = meas(~ismember(1:n,idx),3:4); % Training data
Y = meas(idx,3:4); % Query data
```

Grow a default two-dimensional  $K$  d-tree.

```
MdlKDT = KDTreeSearcher(X)
```

```
MdlKDT =
```

```
    KDTreeSearcher with properties:
```

```
        BucketSize: 50
        Distance: 'euclidean'
        DistParameter: []
           X: [145x2 double]
```

MdlKDT is a KDTreeSearcher model object. You can alter its writable properties using dot notation.

Prepare an exhaustive nearest neighbors searcher.

```
MdlES = ExhaustiveSearcher(X)
```

```
MdlES =
```

```
    ExhaustiveSearcher with properties:
```

```
        Distance: 'euclidean'
        DistParameter: []
           X: [145x2 double]
```

MdlES is an ExhaustiveSearcher model object. It contains the options, such as the distance metric, to use to find nearest neighbors.

Alternatively, you can grow a  $K$  d-tree or prepare an exhaustive nearest neighbors searcher using `createns`.

Search training data for the nearest neighbor indices that correspond to each query observation that are within a 0.5 cm radius. Conduct both types of searches and use the default settings.

```
r = 0.15; % Search radius
IdxKDT = rangesearch(MdlKDT,Y,r);
IdxES = rangesearch(MdlES,Y,r);
[IdxKDT IdxES]
```

```
ans =
```

```
5x2 cell array

    {1x27 double}    {1x27 double}
    {[          13]}    {[          13]}
    {1x27 double}    {1x27 double}
    {1x2  double}    {1x2  double}
    {1x0  double}    {1x0  double}
```

IdxKDT and IdxES are cell arrays of vectors corresponding to the indices of X that are within 0.15 cm of the observations in Y. Each row of the index matrices corresponds to a query observation.

Compare the results between the methods.

```
cellfun(@isequal,IdxKDT,IdxES)
```

```
ans =
```

```
5x1 logical array

     1
     1
     1
     1
```

1

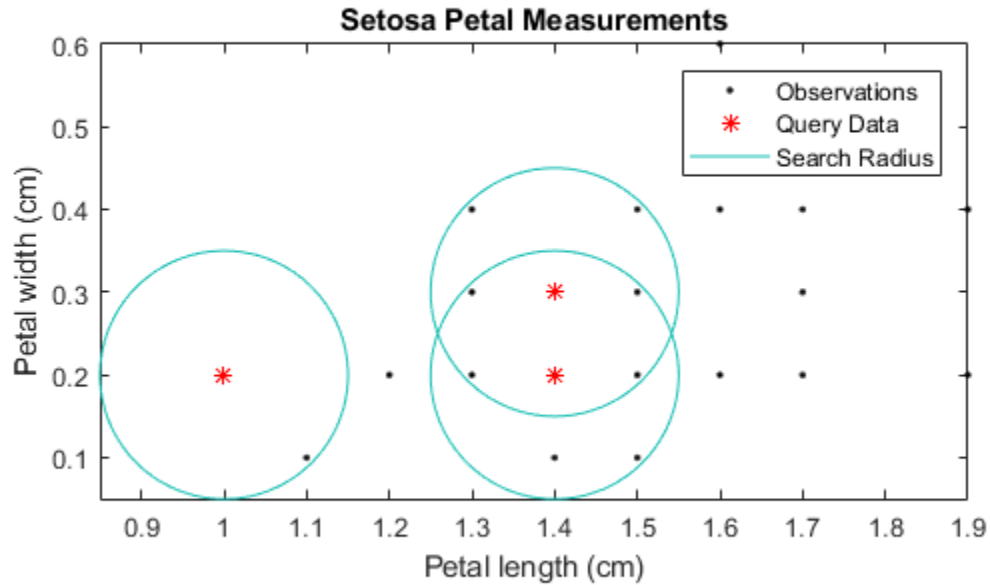
In this case, the results are the same.

Plot the results for the setosa irises.

```
setosaIdx = strcmp(species(~ismember(1:n,idx)), 'setosa');
XSetosa = X(setosaIdx, :);
ySetosaIdx = strcmp(species(idx), 'setosa');
YSetosa = Y(ySetosaIdx, :);

figure;
plot(XSetosa(:,1), XSetosa(:,2), '.k');
hold on;
plot(YSetosa(:,1), YSetosa(:,2), '*r');
for j = 1:sum(ySetosaIdx)
    c = YSetosa(j, :);
    circleFun = @(x1,x2)r^2 - (x1 - c(1)).^2 - (x2 - c(2)).^2;
    ezplot(circleFun, [c(1) + [-1 1]*r, c(2) + [-1 1]*r])
end
xlabel 'Petal length (cm)';
ylabel 'Petal width (cm)';
title 'Setosa Petal Measurements';
legend('Observations', 'Query Data', 'Search Radius');
axis equal
hold off
```





### Search for Neighbors Within a Radius Using the Mahalanobis Distance

Load Fisher's iris data set.

```
load fisheriris
```

Remove five irises randomly from the predictor data to use as a query set.

```
rng(1); % For reproducibility
n = size(meas,1); % Sample size
qIdx = randsample(n,5); % Indices of query data
X = meas(~ismember(1:n,qIdx),:);
Y = meas(qIdx,:);
```

Prepare a default exhaustive nearest neighbors searcher.

```
Mdl = ExhaustiveSearcher(X)
```

```
Mdl =
```

```
ExhaustiveSearcher with properties:
```

```
    Distance: 'euclidean'  
  DistParameter: []  
           X: [145x4 double]
```

Mdl is an ExhaustiveSearcher model.

Find the indices of the training data (X) that are within 0.15 cm of each point in the query data (Y). Specify that the distances are with respect to the Mahalanobis metric.

```
r = 1;  
Idx = rangesearch(Mdl,Y,r, 'Distance', 'mahalanobis')  
Idx{3}
```

```
Idx =
```

```
5x1 cell array  
  
{1x15 double}  
{1x5 double}  
{1x6 double}  
{[      84]}  
{[      69]}
```

```
ans =
```

```
1    34    33    22    24    2
```

Each cell of Idx corresponds to a query data observation and contains in X a vector of indices of the neighbors within 0.15cm of the query data. rangesearch arranges the

indices in ascending order by distance. For example, using the Mahalanobis distance, the second nearest neighbor of  $Y(3, :)$  is  $X(34, :)$ .

### Compute Distances of Neighbors Within a Radius

Load Fisher's iris data set.

```
load fisheriris
```

Remove five irises randomly from the predictor data to use as a query set.

```
rng(4); % For reproducibility
n = size(meas,1); % Sample size
qIdx = randsample(n,5); % Indices of query data
X = meas(~ismember(1:n,qIdx),:);
Y = meas(qIdx,:);
```

Grow a four-dimensional  $K$  d-tree using the training data. Specify to use the Minkowski distance for finding nearest neighbors later.

```
Mdl = KDTreeSearcher(X);
```

`Mdl` is a `KDTreeSearcher` model. By default, the distance metric for finding nearest neighbors is the Euclidean metric.

Find the indices of the training data ( $X$ ) that are within 0.5 cm from each point in the query data ( $Y$ ).

```
r = 0.5;
[Idx,D] = rangesearch(Mdl,Y,r);
```

`Idx` and `D` are five-element cell arrays of vectors. The vector values in `Idx` are the indices in  $X$ . The  $X$  indices represent the observations that are within 0.5 cm of the query data,  $Y$ . `D` contains the distances that correspond to the observations.

Display the results for query observation 3.

```
Idx{3}
D{3}
```

```
ans =
```

```
    127    122

ans =

    0.2646    0.4359
```

The closest observation to  $Y(3, :)$  is  $X(127, :)$ , which is 0.2646 cm away. The next closest is  $X(122, :)$ , which is 0.4359 cm away. All other observations are greater than 0.5 cm away from  $Y(5, :)$ .

## Input Arguments

### **mdl** — Nearest neighbor searcher

`ExhaustiveSearcher` model object | `KDTreeSearcher` model object

Nearest neighbor searcher, specified as an `ExhaustiveSearcher` or `KDTreeSearcher` model object, respectively. To create `mdl`, with the appropriate mode creator. You can also use `createns`.

If `mdl` is an `ExhaustiveSearcher` model, then `rangesearch` searches for nearest neighbors using an exhaustive search. Otherwise, `rangesearch` uses the grown *Kd*-tree to search for nearest neighbors.

### **Y** — Query data

numeric matrix

Query data, specified as a numeric matrix.

$Y$  is an  $m$ -by- $K$  matrix. Rows of  $Y$  correspond to observations (i.e., examples), and columns correspond to predictors (i.e., variables or features).  $Y$  must have the same number of columns as the training data stored in `mdl.X`.

Data Types: `single` | `double`

### **r** — Search radius

nonnegative scalar

Search radius around each point in the query data, specified as a nonnegative scalar.

`rangearch` finds all observations in `Mdl.X` that are within distance `r` of each observation in `Y`. The property `Mdl.Distance` stores the distance.

Data Types: `double` | `single`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'Distance', 'minkowski', 'P', 3` specifies to find all observations in `Mdl.X` within distance `r` of each observation in `Y`, using the Minkowski distance metric with exponent 3.

### For Both Nearest Neighbor Searchers

#### Distance — Distance metric

`Mdl.Distance` (default) | `'cityblock'` | `'euclidean'` | `'mahalanobis'` | `'minkowski'` | `'seuclidean'` | function handle | ...

Distance metric used to find neighbors of the training data to the query observations, specified as the comma-separated pair consisting of `'Distance'` and a character vector or function handle.

For both types of nearest neighbor searchers, `Mdl` supports these distance metrics.

Value	Description
<code>'chebychev'</code>	Chebychev distance (maximum coordinate difference)
<code>'cityblock'</code>	City block distance
<code>'euclidean'</code>	Euclidean distance
<code>'minkowski'</code>	Minkowski distance

If `Mdl` is an `ExhaustiveSearcher` model object, then `rangearch` supports these distance metrics.

Value	Description
'correlation'	One minus the sample linear correlation between observations (treated as sequences of values)
'cosine'	One minus the cosine of the included angle between observations (row vectors)
'hamming'	Hamming distance, which is the percentage of coordinates that differ.
'jaccard'	One minus the Jaccard coefficient, which is the percentage of nonzero coordinates that differ
'mahalanobis'	Mahalanobis distance
'seuclidean'	Standardized Euclidean distance
'spearman'	One minus the sample Spearman's rank correlation between observations (treated as sequences of values)

If `Mdl` is an `ExhaustiveSearcher` model object, then you can also specify a function handle for a custom distance metric using `@` (for example, `@distfun`). The custom distance function must:

- Have the form function `D2 = distfun(ZI, ZJ)`.
- Take as arguments:
  - A 1-by- $K$  vector `ZI` containing a single row from `X` or from the query points `Y`
  - An  $m$ -by- $K$  matrix `ZJ` containing multiple rows of `X` or `Y`
- Return an  $m$ -by-1 vector of distances `D2`. `D2(j)` is the distance between the observations `ZI` and `ZJ(j, :)`.

For more details, see “Distance Metrics” on page 18-16.

Example: `'Distance', 'minkowski'`

Data Types: `char` | `function_handle`

**P** — Exponent for Minkowski distance metric

2 (default) | positive scalar

Exponent for the Minkowski distance metric, specified as the comma-separated pair consisting of 'P' and a positive scalar. If you specify P and do not specify 'Distance', 'minkowski', then the software throws an error.

Example: 'P', 3

Data Types: double | single

#### **For Exhaustive, Nearest Neighbor Searchers**

##### **Cov — Covariance matrix for Mahalanobis distance metric**

nancov(X) (default) | positive definite matrix

Covariance matrix for the Mahalanobis distance metric, specified as the comma-separated pair consisting of 'Cov' and a positive definite matrix. Cov is a  $K$ -by- $K$  matrix, where  $K$  is the number of columns of X. If you specify Cov and do not specify 'Distance', 'mahalanobis', then rangesearch throws an error.

Example: 'Cov', eye(3)

Data Types: double | single

##### **Scale — Scale parameter value for standard Euclidean distance metric**

nanstd(X) (default) | nonnegative numeric vector

Scale parameter value for the standard Euclidean distance metric, specified as the comma-separated pair consisting of 'Scale' and a nonnegative numeric vector. Scale has length  $K$ , where  $K$  is the number of columns of X.

The software scales each difference between the training and query data using the corresponding element of Scale. If you specify Scale and do not specify 'Distance', 'seuclidean', then rangesearch throws an error.

Example: 'Scale', quantile(X, 0.75) - quantile(X, 0.25)

Data Types: double | single

---

**Note** If you specify 'Distance', 'Cov', 'P', or 'Scale', then Mdl.Distance and Mdl.DistanceParameter do not change value.

---

## Output Arguments

### **Idx** — Training data indices of nearest neighbors

cell array of numeric vectors

Training data indices of nearest neighbors, returned as a cell array of numeric vectors.

`Idx` is an  $m$ -by-1 cell array such that cell  $j$  (`Idx{j}`) contains an  $m_j$ -dimensional vector of indices of the observations in `Mdl.X` that are within  $r$  units to the query observation `Y(j, :)`. `rangesearch` arranges the elements of the vectors in ascending order by distance.

### **D** — Distances of nearest neighbors to the query data

cell array of numeric vectors

Distances of the neighbors to the query data, returned as a numeric matrix or cell array of numeric vectors.

`D` is an  $m$ -by-1 cell array such that cell  $j$  (`D{j}`) contains an  $m_j$ -dimensional vector of the distances that the observations in `Mdl.X` are from the query observation `Y(j, :)`. All elements of the vector are less than  $r$ . The function arranges the elements of the vectors in ascending order.

## Algorithms

For positive integer  $K$ , `knnsearch` finds the  $K$  points in `Mdl.X` that are nearest each  $Y$  point. In contrast, for positive scalar  $r$ , `rangesearch` finds all the points in `Mdl.X` that are within a distance  $r$  of each  $Y$  point.

## Alternatives

`rangesearch` is an object function of that requires an `ExhaustiveSearcher` or a `KDTreeSearcher` model object, query data, and a distance. Under equivalent conditions, `rangesearch` returns the same results as `rangesearch` when you specify the name-value pair argument `'NSMethod', 'exhaustive'` or `'NSMethod', 'kdtree'`, respectively.



## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This table contains notes about arguments.

Argument	Notes and Limitations
Mdl	<ul style="list-style-type: none"> <li>• Only the exhaustive search algorithm supports code generation. Therefore, Mdl must be an ExhaustiveSearcher model object.</li> <li>• There are two ways to use Mdl in code generation. For an example, see “Code Generation Workflow for Exhaustive Searcher” on page 31-47. <ul style="list-style-type: none"> <li>• Save a trained model by using <code>saveCompactModel</code>, and then load the model by using <code>loadCompactModel</code> within a function that you declare.</li> <li>• Include <code>coder.Constant(Mdl)</code> in the <code>-args</code> value of <code>codegen</code>.</li> </ul> </li> <li>• Must be a compile-time constant; its value cannot change in the generated code.</li> </ul>
'Distance'	<ul style="list-style-type: none"> <li>• Cannot be a custom distance function.</li> <li>• Must be a compile-time constant; its value cannot change in the generated code.</li> </ul>
Name-value pair arguments	Names in name-value pair arguments must be compile-time constants. For example, to allow a user-defined exponent for the Minkowski distance in the generated code, include <code>{coder.Constant('Distance'), coder.Constant('Minkowski'), coder.Constant('P'), 0}</code> in the <code>-args</code> value of <code>codegen</code> .
Idx	The sorted order of tied distances in the generated code can be different from the order in MATLAB due to numerical precision.

- For code generation, `rangesearch` uses `parfor` (by default) to create loops that run in parallel on supported shared-memory multicore platforms. If your compiler does not support the Open Multiprocessing (OpenMP) application interface or you disable OpenMP library, MATLAB Coder treats the `parfor`-loops as `for`-loops. To find supported compilers, see [http://www.mathworks.com/support/compilers/current\\_release/](http://www.mathworks.com/support/compilers/current_release/). To disable OpenMP library, specify the `EnableOpenMP` property of the `codegen` configuration object as `false`. For details, see `coder.CodeConfig`.

## See Also

`ExhaustiveSearcher` | `KDTreeSearcher` | `createns` | `knnsearch` | `rangesearch`

## Topics

`ExhaustiveSearcher`

`KDTreeSearcher`

“k-Nearest Neighbor Search and Radius Search” on page 18-18

“Distance Metrics” on page 18-16

“Code Generation Workflow for Exhaustive Searcher” on page 31-47

**Introduced in R2011b**

# rangesearch

Find all neighbors within specified distance

## Syntax

```
idx = rangesearch(X, Y, r)
[idx, D] = rangesearch(X, Y, r)
[idx, D] = rangesearch(X, Y, r, Name, Value)
```

## Description

`idx = rangesearch(X, Y, r)` finds all the  $X$  points that are within distance  $r$  of the  $Y$  points. Rows of  $X$  and  $Y$  correspond to observations, and columns correspond to variables.

`[idx, D] = rangesearch(X, Y, r)` returns the distances between each row of  $Y$  and the rows of  $X$  that are  $r$  or less distant.

`[idx, D] = rangesearch(X, Y, r, Name, Value)` finds nearby points with additional options specified by one or more `Name, Value` pair arguments.

## Input Arguments

### **x**

$m \times n$ -by- $n$  numeric matrix, where each row represents one  $n$ -dimensional point. The number of columns  $n$  must equal as the number of columns in  $Y$ .

### **y**

$m \times n$ -by- $n$  numeric matrix, where each row represents one  $n$ -dimensional point. The number of columns  $n$  must equal as the number of columns in  $X$ .

**r**

Search radius, a scalar. `rangesearch` finds all  $X$  points (rows) that are within distance  $r$  of each  $Y$  point. The meaning of distance depends on the `Distance` name-value pair.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### **BucketSize**

Maximum number of data points in the leaf node of the  $kd$ -tree. This argument is only meaningful when  $kd$ -tree is used for finding nearest neighbors.

**Default:** 50

### **Cov**

Positive definite matrix indicating the covariance matrix when computing the Mahalanobis distance. This argument is only valid when the `Distance` name-value pair argument is `'mahalanobis'`.

**Default:** `nancov(X)`

### **Distance**

Character vector or function handle specifying the distance metric.

Value	Description
<code>'euclidean'</code>	Euclidean distance.
<code>'seuclidean'</code>	Standardized Euclidean distance. Each coordinate difference between $X$ and a query point is scaled, meaning divided by a scale value $S$ . The default value of $S$ is the standard deviation computed from $X$ , $S = \text{nanstd}(X)$ . To specify another value for $S$ , use the <code>Scale</code> name-value pair argument.

Value	Description
'mahalanobis'	Mahalanobis distance, computed using a positive definite covariance matrix $C$ . The default value of $C$ is the sample covariance matrix of $X$ , as computed by <code>nancov(X)</code> . To specify a different value for $C$ , use the 'Cov' name-value pair argument.
'cityblock'	City block distance.
'minkowski'	Minkowski distance. The default exponent is 2. To specify a different exponent, use the 'P' name-value pair argument.
'chebychev'	Chebychev distance (maximum coordinate difference).
'cosine'	One minus the cosine of the included angle between observations (treated as vectors).
'correlation'	One minus the sample linear correlation between observations (treated as sequences of values).
'hamming'	Hamming distance, the percentage of coordinates that differ.
'jaccard'	One minus the Jaccard coefficient, the percentage of nonzero coordinates that differ.
'spearman'	One minus the sample Spearman's rank correlation between observations (treated as sequences of values).
@ <i>distfun</i>	Distance function handle. <i>distfun</i> has the form <pre>function D2 = DISTFUN(ZI,ZJ) ... </pre> <p>where</p> <ul style="list-style-type: none"> <li>• <math>ZI</math> is a 1-by-<math>N</math> vector containing one row of <math>X</math> or <math>Y</math>.</li> <li>• <math>ZJ</math> is an <math>M2</math>-by-<math>N</math> matrix containing multiple rows of <math>X</math> or <math>Y</math>.</li> <li>• <math>D2</math> is an <math>M2</math>-by-1 vector of distances, and <math>D2(k)</math> is the distance between the observations <math>ZI</math> and <math>ZJ(J,:)</math>.</li> </ul>

For definitions, see “Distance Metrics” on page 18-16.

**Default:** `NS.Distance`

**NSMethod**

Nearest neighbor search method.

Value	Meaning
'kdtree'	Creates and uses a <i>kd</i> -tree to find nearest neighbors. 'kdtree' is only valid when the distance metric is one of: <ul style="list-style-type: none"> <li>• 'chebychev'</li> <li>• 'cityblock'</li> <li>• 'euclidean'</li> <li>• 'minkowski'</li> </ul>
'exhaustive'	Uses the exhaustive search algorithm. The distances from all <i>X</i> points to each <i>Y</i> point are computed to find nearest neighbors.

**Default:** 'kdtree' when the number of columns of *X* is not greater than 10, *X* is not sparse, and the distance metric is one of the valid 'kdtree' metrics. Otherwise, the default is 'exhaustive'.

**P**

Positive scalar indicating the exponent of Minkowski distance. This argument is only valid when the `Distance` name-value pair argument is 'minkowski'.

**Default:** 2

**Scale**

Vector *S* containing nonnegative values, with length equal to the number of columns in *X*. Each coordinate difference between *X* and a query point is scaled by the corresponding element of *S*. This argument is only valid when the `Distance` name-value pair argument is 'seuclidean'.

**Default:** `nanstd(X)`

## Output Arguments

### **idx**

*my*-by-1 cell array, where *my* is the number of rows in *Y*. `idx{I}` contains the indices of points (rows) in *X* whose distances to *Y*(*I*, :) are not greater than *r*. The entries in `idx{I}` are in ascending order of distance.

### **D**

*my*-by-1 cell array, where *my* is the number of rows in *Y*. `D{I}` contains the distance values between *Y*(*I*, :) and the corresponding points in `idx{I}`.

## Examples

Find the *X* points that are within a Euclidean distance 1.5 of each *Y* point. Both *X* and *Y* are samples of 5-D normally distributed variables.

```
rng('default') % for reproducibility
X = randn(100,5);
Y = randn(10,5);
[idx, dist] = rangesearch(X,Y,1.5)
```

```
idx =
    [1x7 double]
    [1x2 double]
    [1x11 double]
    [1x2 double]
    [1x12 double]
    [1x9 double]
    [      89]
    [1x0 double]
    [1x0 double]
    [1x0 double]
```

```
dist =
    [1x7 double]
    [1x2 double]
    [1x11 double]
    [1x2 double]
    [1x12 double]
```

```
[1x9 double]
[      1.1739]
[1x0 double]
[1x0 double]
[1x0 double]
```

In this case, the last three  $Y$  points are more than 1.5 distant from any  $X$  point.  $X(89, :)$  is 1.1739 distant from  $Y(7, :)$ , and there is no other  $X$  point that is within distance 1.5 of  $Y(7, :)$ . There are 12 points in  $X$  within distance 1.5 of  $Y(5, :)$ .

## Definitions

### Distance Metrics

For definitions, see “Distance Metrics” on page 18-16.

## Tips

- For a fixed positive integer  $K$ , `knnsearch` finds  $K$  points in  $X$  that are nearest each  $Y$  point. In contrast, for a fixed positive real value  $r$ , `rangesearch` finds all the  $X$  points that are within a distance  $r$  of each  $Y$  point.

## Algorithms

For an overview of the *kd*-tree algorithm, see “k-Nearest Neighbor Search Using a Kd-Tree” on page 18-20.

The exhaustive search algorithm finds the distance of each point in  $X$  to each point in  $Y$ .

## Alternatives

`rangesearch` is the `ExhaustiveSearcher` function for distance search. It is equivalent to the `rangesearch` function with the `NSMethod` name-value pair set to `'exhaustive'`.



`rangesearch` is the `KDTreeSearcher` function for distance search. It is equivalent to the `rangesearch` function with the `NSMethod` name-value pair set to `'kdtree'`.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Only the exhaustive search algorithm supports code generation. Therefore, the default value of `'NSMethod'` is `'exhaustive'` for code generation, and you cannot change the value.
- The value of the `'Distance'` name-value pair argument must be a compile-time constant and cannot be a custom distance function.
- Names in name-value pair arguments must be compile-time constants. For example, to allow a user-defined exponent for the Minkowski distance in the generated code, include `{coder.Constant('Distance'), coder.Constant('Minkowski'), coder.Constant('P'), 0}` in the `-args` value of `codegen`.
- The sorted order of tied distances in the generated code can be different from the order in MATLAB due to numerical precision.
- For code generation, `rangesearch` uses `parfor` (by default) to create loops that run in parallel on supported shared-memory multicore platforms. If your compiler does not support the Open Multiprocessing (OpenMP) application interface or you disable OpenMP library, MATLAB Coder treats the `parfor`-loops as `for`-loops. To find supported compilers, see [http://www.mathworks.com/support/compilers/current\\_release/](http://www.mathworks.com/support/compilers/current_release/). To disable OpenMP library, specify the `EnableOpenMP` property of the `codegen` configuration object as `false`. For details, see `coder.CodeConfig`.

### See Also

`ExhaustiveSearcher` | `KDTreeSearcher` | `createns` | `knnsearch` | `pdist2`

## **Topics**

“k-Nearest Neighbor Search and Radius Search” on page 18-18

**Introduced in R2011b**

# ranksum

Wilcoxon rank sum test

## Syntax

```
p = ranksum(x, y)
[p, h] = ranksum(x, y)
[p, h, stats] = ranksum(x, y)
[ ___ ] = ranksum(x, y, Name, Value)
```

## Description

`p = ranksum(x, y)` returns the  $p$ -value of a two-sided Wilcoxon rank sum test on page 32-5419. `ranksum` tests the null hypothesis that data in `x` and `y` are samples from continuous distributions with equal medians, against the alternative that they are not. The test assumes that the two samples are independent. `x` and `y` can have different lengths.

This test is equivalent to a Mann-Whitney U-test.

`[p, h] = ranksum(x, y)` also returns a logical value indicating the test decision. The result `h = 1` indicates a rejection of the null hypothesis, and `h = 0` indicates a failure to reject the null hypothesis at the 5% significance level.

`[p, h, stats] = ranksum(x, y)` also returns the structure `stats` with information about the test statistic.

`[ ___ ] = ranksum(x, y, Name, Value)` returns any of the output arguments in the previous syntaxes, for a rank sum test with additional options specified by one or more `Name, Value` pair arguments.

## Examples

### Test for Equal Median of Two Populations

Test the hypothesis of equal medians for two independent unequal-sized samples.

Generate sample data.

```
rng('default') % for reproducibility
x = unifrnd(0,1,10,1);
y = unifrnd(0.25,1.25,15,1);
```

These samples come from populations with identical distributions except for a shift of 0.25 in the location.

Test the equality of medians of  $x$  and  $y$ .

```
p = ranksum(x,y)
```

```
p =
    0.0375
```

The  $P$ -value of 0.0375 indicates that `ranksum` rejects the null hypothesis of equal medians at the default 5% significance level.

### Statistics of the Test for Two Population Medians

Obtain the statistics of the test for the equality of two population medians.

Load the sample data.

```
load mileage
```

Test if the mileage per gallon is the same for the first and second type of cars.

```
[p,h,stats] = ranksum(mileage(:,1),mileage(:,2))

p =
    0.0043
```

```
h =  
  
    logical  
  
    1  
  
stats =  
  
    struct with fields:  
  
    ranksum: 21.5000
```

Both the  $P$ -value, 0.043, and  $h = 1$  indicate the rejection of the null hypothesis of equal medians at the default 5% significance level. Because the sample sizes are small (six each), ranksum calculates the  $P$ -value using the exact method. The structure `stats` includes only the value of the rank sum test statistic.

### Increase in the Median

Test the hypothesis of an increase in the population median.

Load the sample data.

```
load(fullfile(matlabroot, 'examples', 'stats', 'weather.mat'));
```

The weather data shows the daily high temperatures taken in the same month in two consecutive years.

Perform a left-sided test to assess the increase in the median at the 1% significance level.

```
[p,h,stats] = ranksum(year1,year2, 'alpha',0.01, ...  
    'tail', 'left')
```

```
p =  
  
    0.1271
```

```
h =  
  
    logical  
  
    0  
  
stats =  
  
    struct with fields:  
  
        zval: -1.1403  
        ranksum: 837.5000
```

Both the  $P$ -value of 0.1271 and  $h = 0$  indicate that there is not enough evidence to reject the null hypothesis and conclude that there is a positive shift in the median of observed high temperatures in the same month from year 1 to year 2 at the 1% significance level. Notice that `ranksum` uses the approximate method to calculate the  $P$ -value due to the large sample sizes.

Use the exact method to calculate the  $P$ -value.

```
[p,h,stats] = ranksum(year1,year2,'alpha',0.01,...  
    'tail','left','method','exact')  
  
p =  
  
    0.1273  
  
h =  
  
    logical  
  
    0  
  
stats =  
  
    struct with fields:  
  
        ranksum: 837.5000
```

The results of the approximate and exact methods are consistent with each other.

## Input Arguments

### **x** — Sample data

vector

Sample data, specified as a vector.

Data Types: `single` | `double`

### **y** — Sample data

vector

Sample data, specified as a vector. The length of `y` does not have to be the same as the length of `x`.

Data Types: `single` | `double`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'alpha', 0.01, 'method', 'approximate', 'tail', 'right'` specifies a right-tailed rank sum test with 1% significance level, which returns the approximate p-value.

### **alpha** — Significance level

0.05 (default) | scalar value in the range 0 to 1

Significance level of the decision of a hypothesis test, specified as the comma-separated pair consisting of `'alpha'` and a scalar value in the range 0 to 1. The significance level of `h` is `100 * alpha%`.

Example: `'alpha', 0.01`

Data Types: `double` | `single`

**method** — Computation method of the  $p$ -value`'exact' | 'approximate'`

Computation method of the  $p$ -value,  $p$ , specified as the comma-separated pair consisting of 'method' and one of the following:

<code>'exact'</code>	Exact computation of the $p$ -value, $p$ .
<code>'approximate'</code>	Normal approximation while computing the $p$ -value, $p$ .

When 'method' is unspecified, the default is:

- `'exact'` if  $\min(n_x, n_y) < 10$  and  $n_x + n_y < 20$
- `'approximate'` otherwise

$n_x$  and  $n_y$  are the sizes of the samples in  $x$  and  $y$ , respectively.

Example: `'method','exact'`

Data Types: char

**tail** — Type of test`'both' (default) | 'right' | 'left'`

Type of test, specified as the comma-separated pair consisting of 'tail' and one of the following:

<code>'both'</code>	Two-sided hypothesis test, where the alternative hypothesis states that $x$ and $y$ have different medians. Default test type if 'tail' is not specified.
<code>'right'</code>	Right-tailed hypothesis test, where the alternative hypothesis states that the median of $x$ is greater than the median of $y$ .
<code>'left'</code>	Left-tailed hypothesis test, where the alternative hypothesis states that the median of $x$ is less than the median of $y$ .

Example: `'tail','left'`

Data Types: char

## Output Arguments

**p** —  $p$ -value of the test

nonnegative scalar



$p$ -value of the test, returned as a positive scalar from 0 to 1.  $p$  is the probability of observing a test statistic as or more extreme than the observed value under the null hypothesis. `ranksum` computes the two-sided  $p$ -value by doubling the most significant one-sided value.

### **h** — Result of the hypothesis test

1 | 0

Result of the hypothesis test, returned as a logical value.

- If  $h = 1$ , this indicates rejection of the null hypothesis at the  $100 * \alpha\%$  significance level.
- If  $h = 0$ , this indicates a failure to reject the null hypothesis at the  $100 * \alpha\%$  significance level.

### **stats** — Test statistics

structure

Test statistics, returned as a structure. The test statistics stored in `stats` are:

- `ranksum`: Value of the rank sum test statistic
- `zval`: Value of the z-statistic on page 32-5420 (computed when 'method' is 'approximate')

## Definitions

### Wilcoxon Rank Sum Test

The Wilcoxon rank sum test is a nonparametric test for two populations when samples are independent. If  $X$  and  $Y$  are independent samples with different sample sizes, the test statistic which `ranksum` returns is the rank sum of the first sample.

The Wilcoxon rank sum test is equivalent to the Mann-Whitney U-test. The Mann-Whitney U-test is a nonparametric test for equality of population medians of two independent samples  $X$  and  $Y$ .

The Mann-Whitney U-test statistic,  $U$ , is the number of times a  $y$  precedes an  $x$  in an ordered arrangement of the elements in the two independent samples  $X$  and  $Y$ . It is

related to the Wilcoxon rank sum statistic in the following way: If  $X$  is a sample of size  $n_X$ , then

$$U = W - \frac{n_X(n_X + 1)}{2}.$$

## z-Statistic

For large samples, `ranksum` uses a  $z$ -statistic to compute the approximate  $p$ -value of the test.

If  $X$  and  $Y$  are two independent samples of size  $n_X$  and  $n_Y$ , where  $n_X < n_Y$  the  $z$ -statistic is

$$z = \frac{W - E(W)}{\sqrt{V(W)}} = \frac{W - \left[ \frac{n_X n_Y + n_X(n_X + 1)}{2} \right] - 0.5 * \text{sign}(W - E(W))}{\sqrt{\frac{n_X n_Y (n_X + n_Y + 1) - \text{tiescor}}{12}}},$$

with continuity correction and tie adjustment. Here *tiescor* is given by

$$\text{tiescor} = \frac{2 * \text{tieadj}}{(n_X + n_Y)(n_X + n_Y - 1)},$$

where `ranksum` uses `[ranks, tieadj] = tiedrank(x, y)` to obtain tie adjustments. The standard normal distribution gives the  $p$ -value for this  $z$ -statistic.

## Algorithms

`ranksum` treats NaNs in  $x$  and  $y$  as missing values and ignores them.

For a two-sided test of medians with unequal sample sizes, the test statistic that `ranksum` returns is the rank sum of the first sample.

## References

- [1] Gibbons, J. D., and S. Chakraborti. *Nonparametric Statistical Inference*, 5th Ed., Boca Raton, FL: Chapman & Hall/CRC Press, Taylor & Francis Group, 2011.

[2] Hollander, M., and D. A. Wolfe. *Nonparametric Statistical Methods*. Hoboken, NJ: John Wiley & Sons, Inc., 1999.

## See Also

`kruskalwallis` | `signrank` | `signtest` | `ttest2`

**Introduced before R2006a**

## ranova

**Class:** RepeatedMeasuresModel

Repeated measures analysis of variance

## Syntax

```
ranovatbl = ranova (rm)
ranovatbl = ranova (rm, 'WithinModel', WM)
[ranovatbl, A, C, D] = ranova ( ___ )
```

## Description

`ranovatbl = ranova (rm)` returns the results of repeated measures analysis of variance for a repeated measures model `rm` in table `ranovatbl`.

`ranovatbl = ranova (rm, 'WithinModel', WM)` returns the results of repeated measures analysis of variance using the responses specified by the within-subject model `WM`.

`[ranovatbl, A, C, D] = ranova ( ___ )` also returns arrays `A`, `C`, and `D` for the hypotheses tests of the form  $A*B*C = D$ , where `D` is zero.

## Input Arguments

**rm** — Repeated measures model

RepeatedMeasuresModel object

Repeated measures model, returned as a RepeatedMeasuresModel object.

For properties and methods of this object, see RepeatedMeasuresModel.

**wm** — Model specifying responses

'separatemeans' (default) |  $r$ -by- $nc$  contrast matrix | character vector that defines a model specification

Model specifying the responses, specified as one of the following:

- 'separatemeans' — Compute a separate mean for each group.
- $C$  —  $r$ -by- $nc$  contrast matrix specifying the  $nc$  contrasts among the  $r$  repeated measures. If  $Y$  represents a matrix of repeated measures, `ranova` tests the hypothesis that the means of  $Y*C$  are zero.
- A character vector that defines a model specification in the within-subject factors. You can define the model based on the rules for the terms in the `modelspec` argument of `fitrm`. Also see “Model Specification for Repeated Measures Models” on page 9-79.

For example, if there are three within-subject factors  $w1$ ,  $w2$ , and  $w3$ , then you can specify a model for the within-subject factors as follows.

Example: 'WithinModel', 'w1+w2+w2\*w3'

Data Types: single | double

## Output Arguments

**ranovatbl** — Results of repeated measures anova

table

Results of repeated measures anova, returned as a table.

`ranovatbl` includes a term representing all differences across the within-subjects factors. This term has either the name of the within-subjects factor if specified while fitting the model, or the name `Time` if the name of the within-subjects factor is not specified while fitting the model or there are more than one within-subjects factors. `ranovatbl` also includes all interactions between the terms in the within-subject model and all between-subject model terms. It contains the following columns.

Column Name	Definition
SumSq	Sum of squares.
DF	Degrees of freedom.

Column Name	Definition
MeanSq	Mean squared error.
F	<i>F</i> -statistic.
pValue	<i>p</i> -value for the corresponding <i>F</i> -statistic. A small <i>p</i> -value indicates significant term effect.
pValueGG	<i>p</i> -value with Greenhouse-Geisser adjustment.
pValueHF	<i>p</i> -value with Huynh-Feldt adjustment.
pValueLB	<i>p</i> -value with Lower bound adjustment.

The last three *p*-values are the adjusted *p*-values for use when the compound symmetry assumption is not satisfied. For details, see “Compound Symmetry Assumption and Epsilon Corrections” on page 9-81. The `mauchy` method tests for sphericity (hence, compound symmetry) and `epsilon` method returns the epsilon adjustment values.

#### **A — Specification based on between-subjects model**

`matrix` | `cell array`

Specification based on the between-subjects model, returned as a matrix or a cell array. It permits the hypothesis on the elements within given columns of `B` (within time hypothesis). If `ranovatbl` contains multiple hypothesis tests, `A` might be a cell array.

Data Types: `single` | `double` | `cell`

#### **C — Specification based on within-subjects model**

`matrix` | `cell array`

Specification based on the within-subjects model, returned as a matrix or a cell array. It permits the hypotheses on the elements within given rows of `B` (between time hypotheses). If `ranovatbl` contains multiple hypothesis tests, `C` might be a cell array.

Data Types: `single` | `double` | `cell`

#### **D — Hypothesis value**

0

Hypothesis value, returned as 0.

## Examples

### Repeated Measures Analysis of Variance

Load the sample data.

```
load fisheriris
```

The column vector `species` consists of iris flowers of three different species: `setosa`, `versicolor`, `virginica`. The double matrix `meas` consists of four types of measurements on the flowers: the length and width of sepals and petals in centimeters, respectively.

Store the data in a table array.

```
t = table(species,meas(:,1),meas(:,2),meas(:,3),meas(:,4),...
'VariableNames',{'species','meas1','meas2','meas3','meas4'});
Meas = table([1 2 3 4'],'VariableNames',{'Measurements'});
```

Fit a repeated measures model, where the measurements are the responses and the species is the predictor variable.

```
rm = fitrm(t,'meas1-meas4~species','WithinDesign',Meas);
```

Perform repeated measures analysis of variance.

```
ranovatbl = ranova(rm)
```

```
ranovatbl =
```

```
3x8 table
```

	SumSq	DF	MeanSq	F	pValue
(Intercept):Measurements	1656.3	3	552.09	6873.3	0
species:Measurements	282.47	6	47.078	586.1	1.4271e-206
Error(Measurements)	35.423	441	0.080324		

There are four measurements, three types of species, and 150 observations. So, degrees of freedom for measurements is  $(4-1) = 3$ , for species-measurements interaction it is  $(4-1)*(3-1) = 6$ , and for error it is  $(150-4)*(3-1) = 441$ . `ranova` computes the last three *P*-

values using Greenhouse-Geisser, Huynh-Feldt, and Lower bound corrections, respectively. You can check the compound symmetry (sphericity) assumption using the `mauchly` method, and display the epsilon corrections using the `epsilon` method.

### Longitudinal Data

Load the sample data.

```
load(fullfile(matlabroot, 'examples', 'stats', 'longitudinalData.mat'));
```

The matrix `Y` contains response data for 16 individuals. The response is the blood level of a drug measured at five time points (time = 0, 2, 4, 6, and 8). Each row of `Y` corresponds to an individual, and each column corresponds to a time point. The first eight subjects are female, and the second eight subjects are male. This is simulated data.

Define a variable that stores gender information.

```
Gender = ['F' 'F' 'F' 'F' 'F' 'F' 'F' 'F' 'M' 'M' 'M' 'M' 'M' 'M' 'M'];
```

Store the data in a proper table array format to do repeated measures analysis.

```
t = table(Gender, Y(:,1), Y(:,2), Y(:,3), Y(:,4), Y(:,5), ...  
         'VariableNames', {'Gender', 't0', 't2', 't4', 't6', 't8'});
```

Define the within-subjects variable.

```
Time = [0 2 4 6 8]';
```

Fit a repeated measures model, where the blood levels are the responses and gender is the predictor variable.

```
rm = fitrm(t, 't0-t8 ~ Gender', 'WithinDesign', Time);
```

Perform repeated measures analysis of variance.

```
ranovatbl = anova(rm)
```

```
ranovatbl =
```

```
3x8 table
```

	SumSq	DF	MeanSq	F	pValue	pValueGG
--	-------	----	--------	---	--------	----------



(Intercept):Time	881.7	4	220.43	37.539	3.0348e-15	4.7325e-09
Gender:Time	17.65	4	4.4125	0.75146	0.56126	0.4877
Error(Time)	328.83	56	5.872			

There are 5 time points, 2 genders, and 16 observations. So, the degrees of freedom for time is  $(5-1) = 4$ , for gender-time interaction it is  $(5-1)*(2-1) = 4$ , and for error it is  $(16-2)*(5-1) = 56$ . The small  $P$ -value of  $2.6198e-05$  indicates that there is a significant effect of time on blood pressure. The  $P$ -value of 0.40063 indicates that there is no significant gender-time interaction.

### Specify the Within-Subjects Model

Load the sample data.

```
load repeatedmeas
```

The table `between` includes the between-subject variables `age`, `IQ`, `group`, `gender`, and eight repeated measures `y1` through `y8` as responses. The table `within` includes the within-subject variables `w1` and `w2`. This is simulated data.

Fit a repeated measures model, where the repeated measures `y1` through `y8` are the responses, and `age`, `IQ`, `group`, `gender`, and the group-gender interaction are the predictor variables. Also specify the within-subject design matrix.

```
rm = fitrm(between, 'y1-y8 ~ Group*Gender + Age + IQ', 'WithinDesign', within);
```

Perform repeated measures analysis of variance.

```
ranovatbl = ranova(rm)
```

```
ranovatbl =
```

```
7x8 table
```

	SumSq	DF	MeanSq	F	pValue	pValueGG
(Intercept):Time	6645.2	7	949.31	2.2689	0.031674	0.071235

Age:Time	5824.3	7	832.05	1.9887	0.059978	0.10651
IQ:Time	5188.3	7	741.18	1.7715	0.096749	0.14492
Group:Time	15800	14	1128.6	2.6975	0.0014425	0.011884
Gender:Time	4455.8	7	636.55	1.5214	0.16381	0.20533
Group:Gender:Time	4247.3	14	303.38	0.72511	0.74677	0.663
Error (Time)	64433	154	418.39			

Specify the model for the within-subject factors. Also display the matrices used in the hypothesis test.

```
[ranovatbl,A,C,D] = ranova(rm, 'WithinModel', 'w1+w2')
```

```
ranovatbl =
```

21x8 table

	SumSq	DF	MeanSq	F	pValue	pValueGG
(Intercept)	3141.7	1	3141.7	2.5034	0.12787	0.12787
Age	537.48	1	537.48	0.42828	0.51962	0.51962
IQ	2975.9	1	2975.9	2.3712	0.13785	0.13785
Group	20836	2	10418	8.3012	0.0020601	0.0020601
Gender	3036.3	1	3036.3	2.4194	0.13411	0.13411
Group:Gender	211.8	2	105.9	0.084385	0.91937	0.91937
Error	27609	22	1255			
(Intercept):w1	146.75	1	146.75	0.23326	0.63389	0.63389
Age:w1	942.02	1	942.02	1.4974	0.23402	0.23402
IQ:w1	11.563	1	11.563	0.01838	0.89339	0.89339
Group:w1	4481.9	2	2240.9	3.562	0.045697	0.045697
Gender:w1	270.65	1	270.65	0.4302	0.51869	0.51869
Group:Gender:w1	240.37	2	120.19	0.19104	0.82746	0.82746
Error (w1)	13841	22	629.12			
(Intercept):w2	3663.8	3	1221.3	3.8381	0.013513	0.020339
Age:w2	1199.9	3	399.95	1.2569	0.2964	0.29645
IQ:w2	3650.1	3	1216.7	3.8237	0.013744	0.020636
Group:w2	5963.8	6	993.96	3.1237	0.0093493	0.015434
Gender:w2	2173.1	3	724.38	2.2765	0.087813	0.10134
Group:Gender:w2	3339.6	6	556.6	1.7492	0.12345	0.14
Error (w2)	21001	66	318.2			

```
A =
```

```
6x1 cell array
```

```
{1x8 double}
{1x8 double}
{1x8 double}
{2x8 double}
{1x8 double}
{2x8 double}
```

```
C =
```

```
1x3 cell array
```

```
{8x1 double} {8x1 double} {8x3 double}
```

```
D =
```

```
0
```

**Display the contents of A.**

```
[A{1};A{2};A{3};A{4};A{5};A{6}]
```

```
ans =
```

```
1 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0
0 0 1 0 0 0 0 0
0 0 0 1 0 0 0 0
0 0 0 0 1 0 0 0
0 0 0 0 0 1 0 0
0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 1
```

**Display the contents of C.**

```
[C{1} C{2} C{3}]
```

```
ans =
```

```
1      1      1      0      0
1      1      0      1      0
1      1      0      0      1
1      1     -1     -1     -1
1     -1      1      0      0
1     -1      0      1      0
1     -1      0      0      1
1     -1     -1     -1     -1
```

## Algorithms

`ranova` computes the regular  $p$ -value (in the `pValue` column of the `rmanova` table) using the  $F$ -statistic cumulative distribution function:

$$p\text{-value} = 1 - \text{fcdf}(F, v_1, v_2).$$

When the compound symmetry assumption is not satisfied, `ranova` uses a correction factor `epsilon`,  $\varepsilon$ , to compute the corrected  $p$ -values as follows:

$$p\text{-value\_corrected} = 1 - \text{fcdf}(F, \varepsilon * v_1, \varepsilon * v_2).$$

The `mauchly` method tests for sphericity (hence, compound symmetry) and `epsilon` method returns the epsilon adjustment values.

## See Also

`anova` | `epsilon` | `fitrm` | `manova` | `mauchly`

## Topics

“Model Specification for Repeated Measures Models” on page 9-79

“Compound Symmetry Assumption and Epsilon Corrections” on page 9-81

“Mauchly’s Test of Sphericity” on page 9-83

# raylcdf

Rayleigh cumulative distribution function

## Syntax

```
p = raylcdf(x,b)
p = raylcdf(x,b,'upper')
```

## Description

`p = raylcdf(x,b)` returns the Rayleigh cdf at each value in `x` using the corresponding scale parameter, `b`. `x` and `b` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input for `x` or `b` is expanded to a constant array with the same dimensions as the other input.

`p = raylcdf(x,b,'upper')` returns the complement of the Rayleigh cdf at each value in `x`, using an algorithm that more accurately computes the extreme upper tail probabilities.

The Rayleigh cdf is

$$y = F(x | b) = \int_0^x \frac{t}{b^2} e^{\left(\frac{-t^2}{2b^2}\right)} dt$$

## Examples

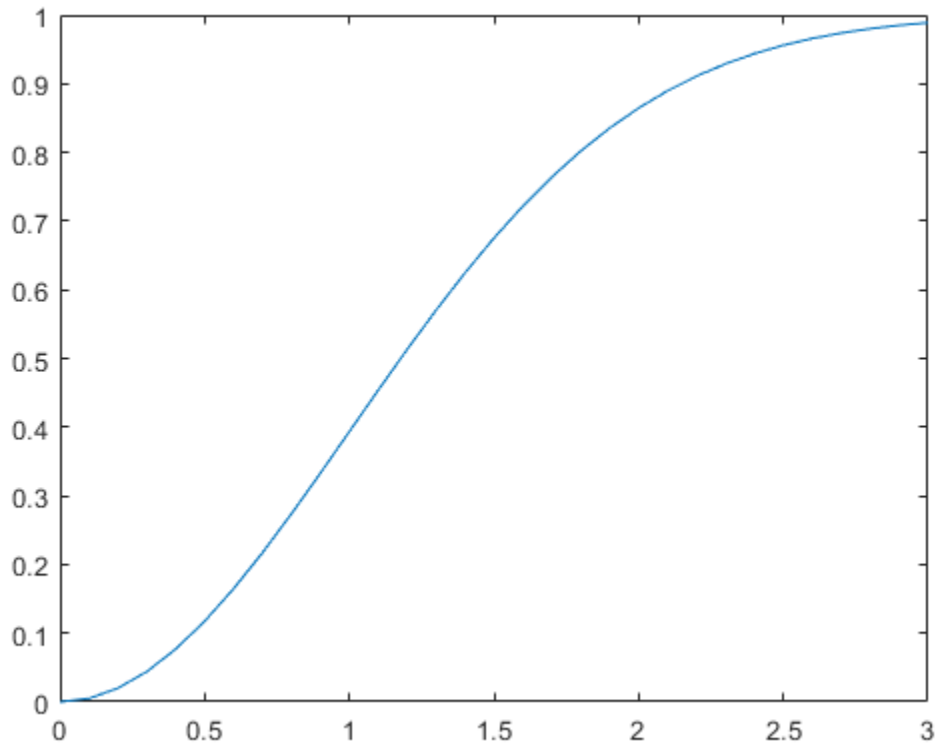
### Compute and Plot Rayleigh Distribution cdf

Compute the cdf of a Rayleigh distribution with parameter  $B = 1$ .

```
x = 0:0.1:3;
p = raylcdf(x,1);
```

Plot the cdf.

```
figure;  
plot(x,p)
```



## References

- [1] Evans, M., N. Hastings, and B. Peacock. *Statistical Distributions*. Hoboken, NJ: Wiley-Interscience, 2000. pp. 134–136.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`cdf` | `raylfit` | `raylinv` | `raylpdf` | `raylrnd` | `raylstat`

### Topics

“Rayleigh Distribution” on page B-146

Introduced before R2006a

## raylfit

Rayleigh parameter estimates

### Syntax

```
raylfit(data,alpha)
[phat,pci] = raylfit(data,alpha)
```

### Description

`raylfit(data,alpha)` returns the maximum likelihood estimates of the parameter of the Rayleigh distribution given the data in the vector `data`.

`[phat,pci] = raylfit(data,alpha)` returns the maximum likelihood estimate and  $100(1 - \text{alpha})\%$  confidence interval given the data. The default value of the optional parameter `alpha` is 0.05, corresponding to 95% confidence intervals.

### See Also

`mle` | `raylcdf` | `raylinv` | `raylpdf` | `raylrnd` | `raylstat`

### Topics

“Rayleigh Distribution” on page B-146

Introduced before R2006a



# raylinv

Rayleigh inverse cumulative distribution function

## Syntax

```
X = raylinv(P,B)
```

## Description

`X = raylinv(P,B)` returns the inverse of the Rayleigh cumulative distribution function using the corresponding scale parameter, `B` at the corresponding probabilities in `P`. `P` and `B` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input for `P` or `B` is expanded to a constant array with the same dimensions as the other input.

## Examples

```
x = raylinv(0.9,1)
x =
    2.1460
```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`raylcdf` | `raylpdf` | `raylrnd` | `raylstat`

## **Topics**

“Rayleigh Distribution” on page B-146

**Introduced before R2006a**

# raylpdf

Rayleigh probability density function

## Syntax

```
Y = raylpdf(X,B)
```

## Description

`Y = raylpdf(X,B)` computes the Rayleigh pdf at each of the values in `X` using the corresponding scale parameter, `B`. `X` and `B` can be vectors, matrices, or multidimensional arrays that all have the same size, which is also the size of `Y`. A scalar input for `X` or `B` is expanded to a constant array with the same dimensions as the other input.

The Rayleigh pdf is

$$y = f(x | b) = \frac{x}{b^2} e^{\left(\frac{-x^2}{2b^2}\right)}$$

## Examples

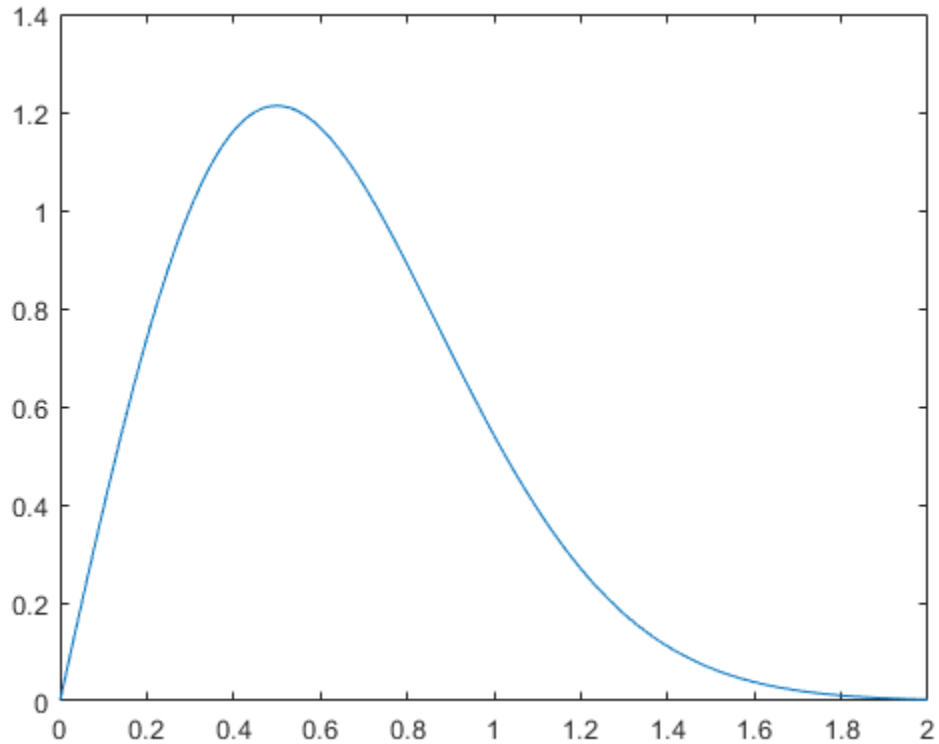
### Compute and Plot Rayleigh Distribution pdf

Compute the pdf of a Rayleigh distribution with parameter `B = 0.5`.

```
x = [0:0.01:2];  
p = raylpdf(x,0.5);
```

Plot the pdf.

```
figure;  
plot(x,p)
```



## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`pdf` | `raylcdf` | `raylfit` | `raylinv` | `raylrnd` | `raylstat`

## **Topics**

“Rayleigh Distribution” on page B-146

Introduced before R2006a

## raylrnd

Rayleigh random numbers

### Syntax

```
R = raylrnd(B)
R = raylrnd(B,v)
R = raylrnd(B,m,n)
```

### Description

`R = raylrnd(B)` returns a matrix of random numbers chosen from the Rayleigh distribution with scale parameter, `B`. `B` can be a vector, a matrix, or a multidimensional array. The size of `R` is the size of `B`.

`R = raylrnd(B,v)` returns a matrix of random numbers chosen from the Rayleigh distribution with parameter `B`, where `v` is a row vector. If `v` is a 1-by-2 vector, `R` is a matrix with `v(1)` rows and `v(2)` columns. If `v` is 1-by-`n`, `R` is an `n`-dimensional array.

`R = raylrnd(B,m,n)` returns a matrix of random numbers chosen from the Rayleigh distribution with parameter `B`, where scalars `m` and `n` are the row and column dimensions of `R`.

### Examples

```
r = raylrnd(1:5)
r =
    1.7986    0.8795    3.3473    8.9159    3.5182
```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

The generated code can return a different sequence of numbers than MATLAB if either of the following is true:

- The output is nonscalar.
- An input parameter is invalid for the distribution.

### See Also

`random` | `raylcdf` | `raylfit` | `raylinv` | `raylpdf` | `raylstat`

### Topics

“Rayleigh Distribution” on page B-146

**Introduced before R2006a**

## raylstat

Rayleigh mean and variance

### Syntax

```
[M,V] = raylstat(B)
```

### Description

`[M,V] = raylstat(B)` returns the mean of and variance for the Rayleigh distribution with scale parameter `B`.

The mean of the Rayleigh distribution with parameter  $b$  is  $b\sqrt{\pi/2}$  and the variance is  $\frac{4-\pi}{2}b^2$

### Examples

```
[mn,v] = raylstat(1)
mn =
    1.2533
v =
    0.4292
```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.



## See Also

`raylcdf` | `raylfit` | `raylinv` | `raylpdf` | `raylrnd`

## Topics

“Rayleigh Distribution” on page B-146

Introduced before R2006a

## rcoplot

Residual case order plot

### Syntax

```
rcoplot(r, rint)
```

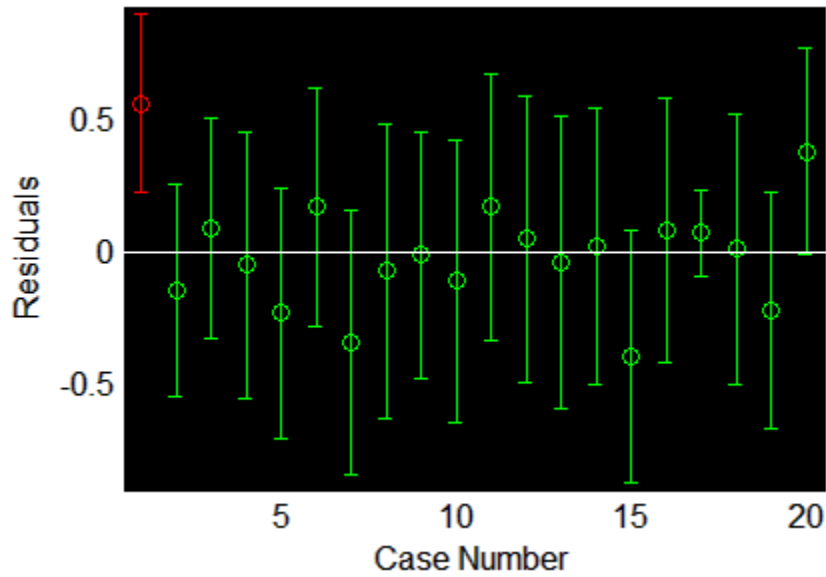
### Description

`rcoplot(r, rint)` displays an errorbar plot of the confidence intervals on the residuals from a regression. The residuals appear in the plot in case order. Inputs `r` and `rint` are outputs from the `regress` function.

### Examples

The following plots residuals and prediction intervals from a regression of a linearly additive model to the data in `moore.mat`:

```
load moore
X = [ones(size(moore,1),1) moore(:,1:5)];
y = moore(:,6);
alpha = 0.05;
[betahat,Ibeta,res,Ires,stats] = regress(y,X,alpha);
rcoplot(res,Ires)
```



The interval around the first residual, shown in red, does not contain zero. This indicates that the residual is larger than expected in 95% of new observations, and suggests the data point is an outlier.

## See Also

`regress`

Introduced before R2006a

# ReconstructionICA

Feature extraction by reconstruction ICA

## Description

`ReconstructionICA` applies reconstruction independent component analysis (RICA) to learn a transformation that maps input predictors to new predictors.

## Creation

Create a `ReconstructionICA` object by using the `rica` function.

## Properties

### **FitInfo** — Fitting history

structure

Fitting history, returned as a structure with two fields:

- `Iteration` — Iteration numbers from 0 through the final iteration.
- `Objective` — Objective function value at each corresponding iteration. Iteration 0 corresponds to the initial values, before any fitting.

Data Types: `struct`

### **InitialTransformWeights** — Initial feature transformation weights

p-by-q matrix

Initial feature transformation weights, returned as a p-by-q matrix, where p is the number of predictors passed in `X` and q is the number of features that you want. These weights are the initial weights passed to the creation function. The data type is `single` when the training data `X` is `single`.

Data Types: `single` | `double`

**ModelParameters** — Parameters for training model

structure

Parameters for training the model, returned as a structure. The structure contains a subset of the fields that correspond to the `rica` name-value pairs that were in effect during model creation:

- `IterationLimit`
- `VerbosityLevel`
- `Lambda`
- `Standardize`
- `ContrastFcn`
- `GradientTolerance`
- `StepTolerance`

For details, see the `rica` Name, Value on page 32-5936 pairs.

Data Types: `struct` **$\mu$**  — Predictor means when standardizing

p-by-1 vector

Predictor means when standardizing, returned as a p-by-1 vector. This property is nonempty when the `Standardize` name-value pair is `true` at model creation. The value is the vector of predictor means in the training data. The data type is `single` when the training data `X` is `single`.

Data Types: `single` | `double`**NonGaussianityIndicator** — Non-Gaussianity of sourceslength-q vector of  $\pm 1$ 

Non-Gaussianity of sources, returned as a length-q vector of  $\pm 1$ .

- `NonGaussianityIndicator(k) = 1` means `rica` models the *k*th source as sub-Gaussian.
- `NonGaussianityIndicator(k) = -1` means `rica` models the *k*th source as super-Gaussian, with a sharp peak at 0.

Data Types: `double`

**NumLearnedFeatures** — Number of output features

positive integer

Number of output features, returned as a positive integer. This value is the `q` argument passed to the creation function, which is the requested number of features to learn.

Data Types: double

**NumPredictors** — Number of input predictors

positive integer

Number of input predictors, returned as a positive integer. This value is the number of predictors passed in `X` to the creation function.

Data Types: double

**Sigma** — Predictor standard deviations when standardizing

p-by-1 vector

Predictor standard deviations when standardizing, returned as a p-by-1 vector. This property is nonempty when the `Standardize` name-value pair is `true` at model creation. The value is the vector of predictor standard deviations in the training data. The data type is `single` when the training data `X` is `single`.

Data Types: single | double

**TransformWeights** — Feature transformation weights

p-by-q matrix

Feature transformation weights, returned as a p-by-q matrix, where `p` is the number of predictors passed in `X` and `q` is the number of features that you want. The data type is `single` when the training data `X` is `single`.

Data Types: single | double

## Object Functions

`transform` Transform predictors into extracted features

## Examples

## Create Reconstruction ICA Object

Create a ReconstructionICA object by using the `rica` function.

Load the `caltech101` image patches.

```
data = load('caltech101patches');  
size(data.X)
```

```
ans =  
  
    100000    363
```

There are 100,000 image patches, each containing 363 features.

Extract 100 features from the data.

```
rng default % For reproducibility  
q = 100;  
Mdl = rica(data.X,q, 'IterationLimit',100)
```

Warning: Solver LBFGS was not able to converge to a solution.

```
Mdl =  
  
ReconstructionICA  
    ModelParameters: [1x1 struct]  
    NumPredictors: 363  
    NumLearnedFeatures: 100  
                Mu: []  
                Sigma: []  
                FitInfo: [1x1 struct]  
    TransformWeights: [363x100 double]  
    InitialTransformWeights: []  
    NonGaussianityIndicator: [100x1 double]
```

`rica` issues a warning because it stopped due to reaching the iteration limit, instead of reaching a step-size limit or a gradient-size limit. You can still use the learned features in the returned object by calling the `transform` function.

- “Feature Extraction Workflow” on page 15-169
- “Extract Mixed Signals” on page 15-205

## See Also

`rica` | `sparsefilt` | `transform`

## Topics

“Feature Extraction Workflow” on page 15-169

“Extract Mixed Signals” on page 15-205

“Feature Extraction” on page 15-163

**Introduced in R2017a**



# refcurve

Add reference curve to plot

## Syntax

```
refcurve(p)
refcurve
hcurve = refcurve(...)
```

## Description

`refcurve(p)` adds a polynomial reference curve with coefficients `p` to the current axes. If `p` is a vector with `n+1` elements, the curve is:

$$y = p(1)*x^n + p(2)*x^{(n-1)} + \dots + p(n)*x + p(n+1)$$

`refcurve` with no input arguments adds a line along the  $x$  axis.

`hcurve = refcurve(...)` returns the handle `hcurve` to the curve.

## Examples

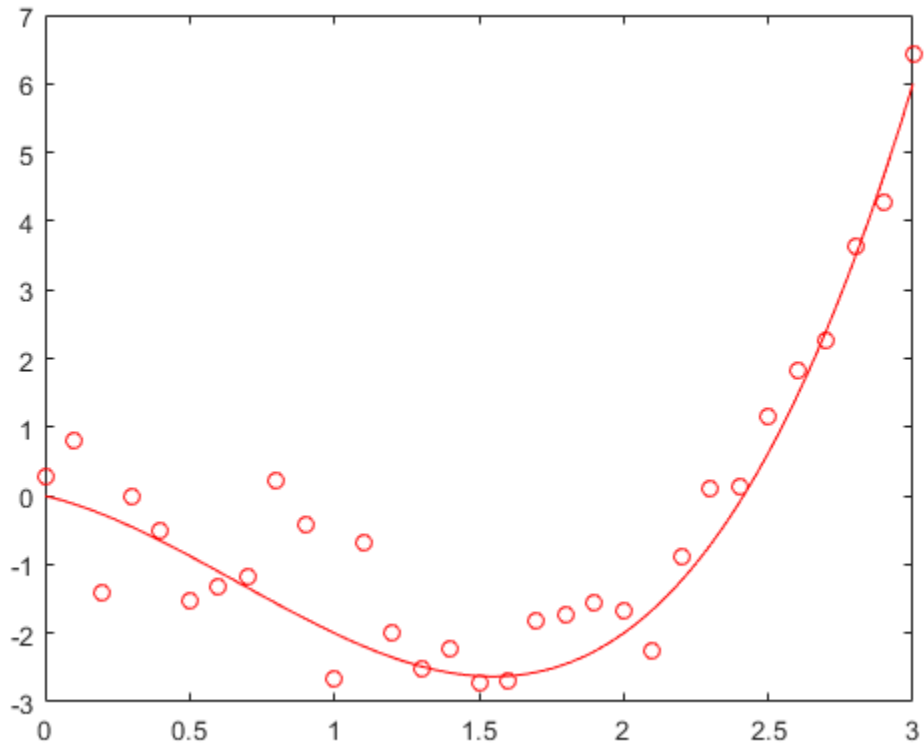
### Add Population and Fitted Mean Functions

Generate data with a polynomial trend.

```
p = [1 -2 -1 0];
t = 0:0.1:3;
rng default % For reproducibility
y = polyval(p,t) + 0.5*randn(size(t));
```

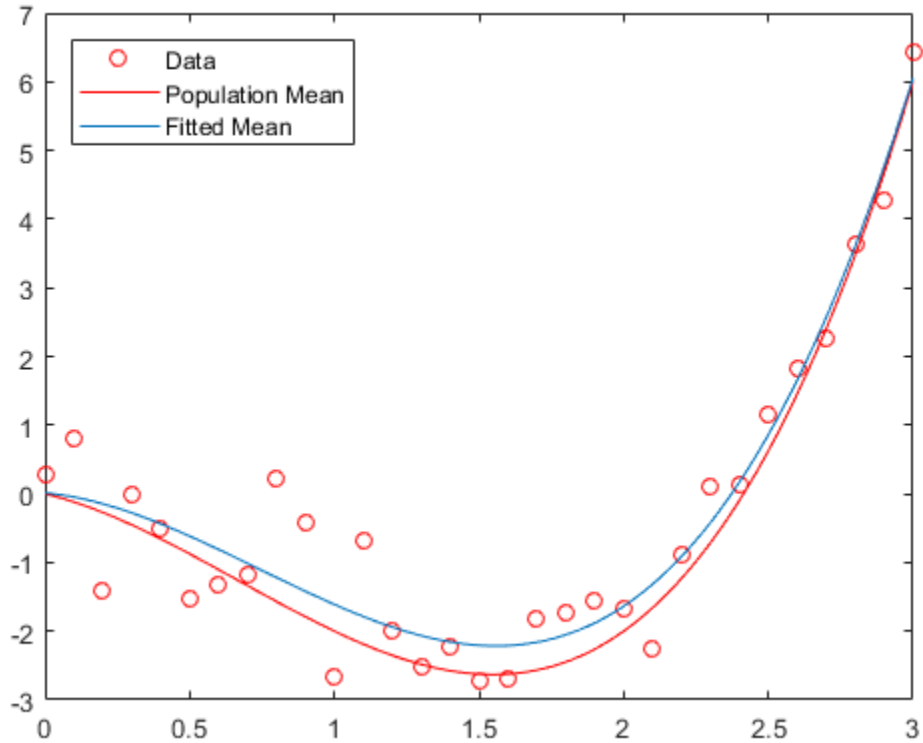
Plot data and add the population mean function using `refcurve`.

```
plot(t,y,'ro')  
h = refcurve(p);  
h.Color = 'r';
```



Also add the fitted mean function.

```
q = polyfit(t,y,3);  
refcurve(q)  
legend('Data', 'Population Mean', 'Fitted Mean', ...  
       'Location', 'NW')
```



### Plot Trajectories of a Batted Baseball Using refcurve

Introduce the relevant physical constants.

```

M = 0.145;      % Mass (kg)
R = 0.0366;    % Radius (m)
A = pi*R^2;    % Area (m^2)
rho = 1.2;     % Density of air (kg/m^3)
C = 0.5;       % Drag coefficient
D = rho*C*A/2; % Drag proportional to the square of the speed
g = 9.8;       % Acceleration due to gravity (m/s^2)

```

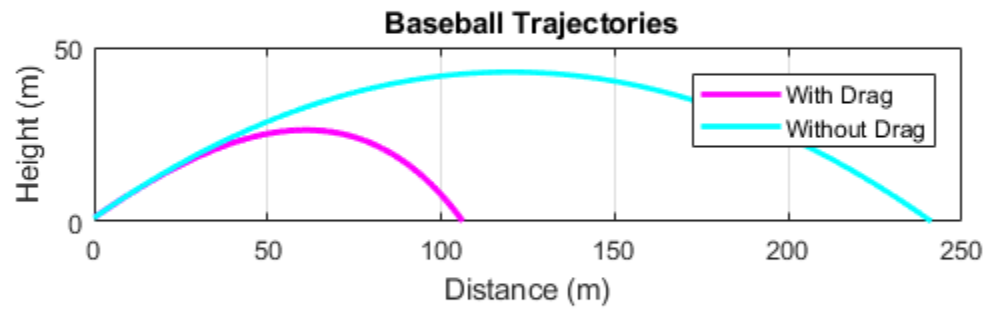
Simulate the trajectory with drag proportional to the square of the speed, assuming constant acceleration in each time interval.

```
dt = 1e-2;      % Simulation time interval (s)
r0 = [0 1];    % Initial position (m)
s0 = 50;      % Initial speed (m/s)
alpha0 = 35;  % Initial angle (deg)
v0 = s0*[cosd(alpha0) sind(alpha0)]; % Initial velocity (m/s)

r = r0;
v = v0;
trajectory = r0;
while r(2) > 0
    a = [0 -g] - (D/M)*norm(v)*v;
    v = v + a*dt;
    r = r + v*dt + (1/2)*a*(dt^2);
    trajectory = [trajectory;r];
end
```

Plot trajectory and use `refcurve` to add the drag-free parabolic trajectory (found analytically) to the plot of trajectory.

```
figure
plot(trajectory(:,1),trajectory(:,2),'m','LineWidth',2)
xlim([0,250])
h = refcurve([-g/(2*v0(1)^2),...
    (g*r0(1)/v0(1)^2) + (v0(2)/v0(1)),...
    (-g*r0(1)^2/(2*v0(1)^2) - (v0(2)*r0(1)/v0(1)) + r0(2)]);
h.Color = 'c';
h.LineWidth = 2;
axis equal
ylim([0,50])
grid on
xlabel('Distance (m)')
ylabel('Height (m)')
title('{\bf Baseball Trajectories}')
legend('With Drag','Without Drag')
```



## See Also

`gline` | `lsline` | `polyfit` | `refline`

Introduced before R2006a

## refit

**Class:** GeneralizedLinearMixedModel

Refit generalized linear mixed-effects model

## Syntax

```
glmnew = refit(glm, ynew)
```

## Description

`glmnew = refit(glm, ynew)` returns a refitted generalized linear mixed-effects model, `glmnew`, based on the input model `glm`, using a new response vector, `ynew`.

## Input Arguments

**glm** — Generalized linear mixed-effects model

GeneralizedLinearMixedModel object

Generalized linear mixed-effects model, specified as a GeneralizedLinearMixedModel object. For properties and methods of this object, see GeneralizedLinearMixedModel.

**ynew** — New response vector

$n$ -by-1 vector of scalar values

New response vector, specified as an  $n$ -by-1 vector of scalar values, where  $n$  is the number of observations used to fit `glm`.

For an observation  $i$  with prior weights  $w_i^p$  and binomial size  $n_i$  (when applicable), the response values  $y_i$  contained in `ynew` can have the following values.

Distribution	Permitted Values	Notes
Binomial	$\left\{0, \frac{1}{w_i^p n_i}, \frac{2}{w_i^p n_i}, \dots, 1\right\}$	$w_i^p$ and $n_i$ are integer values $> 0$
Poisson	$\left\{0, \frac{1}{w_i^p}, \frac{2}{w_i^p}, \dots, 1\right\}$	$w_i^p$ is an integer value $> 0$
Gamma	$(0, \infty)$	$w_i^p \geq 0$
InverseGaussian	$(0, \infty)$	$w_i^p \geq 0$
Normal	$(-\infty, \infty)$	$w_i^p \geq 0$

You can access the prior weights property  $w_i^p$  using dot notation.

```
glme.ObservationInfo.Weights
```

Data Types: `single` | `double`

## Output Arguments

### **glmnew** — Generalized linear mixed-effects model

`GeneralizedLinearMixedModel` object

Generalized linear mixed-effects model, returned as a `GeneralizedLinearMixedModel` object. `glmnew` is an updated version of the generalized linear mixed-effects model `glme`, refit to the values in the response vector `ynew`.

For properties and methods of this object, see `GeneralizedLinearMixedModel`.

## Examples

### Refit Model to New Response Vector

Load the sample data.

```
load mfr
```

This simulated data is from a manufacturing company that operates 50 factories across the world, with each factory running a batch process to create a finished product. The company wants to decrease the number of defects in each batch, so it developed a new manufacturing process. To test the effectiveness of the new process, the company selected 20 of its factories at random to participate in an experiment: Ten factories implemented the new process, while the other ten continued to run the old process. In each of the 20 factories, the company ran five batches (for a total of 100 batches) and recorded the following data:

- Flag to indicate whether the batch used the new process (`newprocess`)
- Processing time for each batch, in hours (`time`)
- Temperature of the batch, in degrees Celsius (`temp`)
- Categorical variable indicating the supplier (A, B, or C) of the chemical used in the batch (`supplier`)
- Number of defects in the batch (`defects`)

The data also includes `time_dev` and `temp_dev`, which represent the absolute deviation of time and temperature, respectively, from the process standard of 3 hours at 20 degrees Celsius.

Fit a generalized linear mixed-effects model using `newprocess`, `time_dev`, `temp_dev`, and `supplier` as fixed-effects predictors. Include a random-effects term for intercept grouped by `factory`, to account for quality differences that might exist due to factory-specific variations. The response variable `defects` has a Poisson distribution, and the appropriate link function for this model is `log`. Use the Laplace fit method to estimate the coefficients. Specify the dummy variable encoding as `'effects'`, so the dummy variable coefficients sum to 0.

The number of defects can be modeled using a Poisson distribution

$$\text{defects}_{ij} \sim \text{Poisson}(\mu_{ij})$$

This corresponds to the generalized linear mixed-effects model

$$\log(\mu_{ij}) = \beta_0 + \beta_1 \text{newprocess}_{ij} + \beta_2 \text{time.dev}_{ij} + \beta_3 \text{temp.dev}_{ij} + \beta_4 \text{supplier.C}_{ij} + \beta_5 \text{supplier.B}_{ij} + b_i,$$

where

- `defectsij` is the number of defects observed in the batch produced by factory *i* during batch *j*.



- $\mu_{ij}$  is the mean number of defects corresponding to factory  $i$  (where  $i = 1, 2, \dots, 20$ ) during batch  $j$  (where  $j = 1, 2, \dots, 5$ ).
- `newprocessij`, `time_devij`, and `temp_devij` are the measurements for each variable that correspond to factory  $i$  during batch  $j$ . For example, `newprocessij` indicates whether the batch produced by factory  $i$  during batch  $j$  used the new process.
- `supplier_Cij` and `supplier_Bij` are dummy variables that use effects (sum-to-zero) coding to indicate whether company C or B, respectively, supplied the process chemicals for the batch produced by factory  $i$  during batch  $j$ .
- $b_i \sim N(0, \sigma_b^2)$  is a random-effects intercept for each factory  $i$  that accounts for factory-specific variation in quality.

```
glme = fitglme(mfr, 'defects ~ 1 + newprocess + time_dev + temp_dev + supplier + (1|fact
```

Use random to simulate a new response vector from the fitted model.

```
rng(0, 'twister'); % For reproducibility
ynew = random(glme);
```

Refit the model using the new response vector.

```
glme = refit(glme, ynew)
```

```
glme =
```

Generalized linear mixed-effects model fit by ML

Model information:

Number of observations	100
Fixed effects coefficients	6
Random effects coefficients	20
Covariance parameters	1
Distribution	Poisson
Link	Log
FitMethod	Laplace

Formula:

```
defects ~ 1 + newprocess + time_dev + temp_dev + supplier + (1 | factory)
```

Model fit statistics:

AIC	BIC	LogLikelihood	Deviance
469.24	487.48	-227.62	455.24

Fixed effects coefficients (95% CIs):

Name	Estimate	SE	tStat	DF	pValue
'(Intercept)'	1.5738	0.18674	8.4276	94	4.0158e-13
'newprocess'	-0.21089	0.2306	-0.91455	94	0.36277
'time_dev'	-0.13769	0.77477	-0.17772	94	0.85933
'temp_dev'	0.24339	0.84657	0.2875	94	0.77436
'supplier_C'	-0.12102	0.07323	-1.6526	94	0.10175
'supplier_B'	0.098254	0.066943	1.4677	94	0.14551

Lower	Upper
1.203	1.9445
-0.66875	0.24696
-1.676	1.4006
-1.4375	1.9243
-0.26642	0.024381
-0.034662	0.23117

Random effects covariance parameters:

Group: factory (20 Levels)

Name1	Name2	Type	Estimate
'(Intercept)'	'(Intercept)'	'std'	0.46587

Group: Error

Name	Estimate
'sqrt(Dispersion)'	1

## Tips

- You can use `refit` and `random` to conduct a simulated likelihood ratio test or parametric bootstrap.

## See Also

`GeneralizedLinearMixedModel` | `designMatrix` | `fitted` | `residuals`

## refit

**Class:** FeatureSelectionNCAClassification

Refit neighborhood component analysis (NCA) model for classification

## Syntax

```
mdlrefit = refit(mdl, Name, Value)
```

## Description

`mdlrefit = refit(mdl, Name, Value)` refits the model `mdl`, with modified parameters specified by one or more `Name, Value` pair arguments.

## Input Arguments

**mdl** — Neighborhood component analysis model for classification

FeatureSelectionNCAClassification object

Neighborhood component analysis model or classification, specified as a FeatureSelectionNCAClassification object.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### Fitting Options

**FitMethod** — Method for fitting the model

`mdl.FitMethod` (default) | 'exact' | 'none' | 'average'

Method for fitting the model, specified as the comma-separated pair consisting of 'FitMethod' and one of the following.

- 'exact' — Performs fitting using all of the data.
- 'none' — No fitting. Use this option to evaluate the generalization error of the NCA model using the initial feature weights supplied in the call to `fscnca`.
- 'average' — The function divides the data into partitions (subsets), fits each partition using the `exact` method, and returns the average of the feature weights. You can specify the number of partitions using the `NumPartitions` name-value pair argument.

Example: 'FitMethod', 'none'

Data Types: `char`

### **Lambda** — Regularization parameter

`mdl.Lambda` (default) | non-negative scalar value

Regularization parameter, specified as the comma-separated pair consisting of 'Lambda' and a non-negative scalar value.

For  $n$  observations, the best `Lambda` value that minimizes the generalization error of the NCA model is expected to be a multiple of  $1/n$

Example: 'Lambda', 0.01

Data Types: `double` | `single`

### **Solver** — Solver type

`mdl.Solver` (default) | 'lbfgs' | 'sgd' | 'minibatch-lbfgs'

Solver type for estimating feature weights, specified as the comma-separated pair consisting of 'Solver' and one of the following.

- 'lbfgs' — Limited memory BFGS (Broyden-Fletcher-Goldfarb-Shanno) algorithm (LBFGS algorithm)
- 'sgd' — Stochastic gradient descent
- 'minibatch-lbfgs' — Stochastic gradient descent with LBFGS algorithm applied to mini-batches

Example: 'solver', 'minibatch-lbfgs'

Data Types: char

### **InitialFeatureWeights** — Initial feature weights

`mdl.InitialFeatureWeights` (default) |  $p$ -by-1 vector of real positive scalar values

Initial feature weights, specified as the comma-separated pair consisting of 'InitialFeatureWeights' and a  $p$ -by-1 vector of real positive scalar values.

Data Types: double | single

### **Verbose** — Indicator for verbosity level

`mdl.Verbose` (default) | 0 | 1 | >1

Indicator for verbosity level for the convergence summary display, specified as the comma-separated pair consisting of 'Verbose' and one of the following.

- 0 — No convergence summary
- 1 — Convergence summary including iteration number, norm of the gradient, and objective function value.
- >1 — More convergence information depending on the fitting algorithm

When using solver 'minibatch-lbfgs' and verbosity level >1, the convergence information includes iteration log from intermediate minibatch LBFGS fits.

Example: 'Verbose',2

Data Types: double | single

### **LBFGS or Mini-Batch LBFGS Options**

#### **GradientTolerance** — Relative convergence tolerance

`mdl.GradientTolerance` (default) | positive real scalar value

Relative convergence tolerance on the gradient norm for solver `lbfgs`, specified as the comma-separated pair consisting of 'GradientTolerance' and a positive real scalar value.

Example: 'GradientTolerance',0.00001

Data Types: double | single

### SGD or Mini-Batch LBFSG Options

#### **InitialLearningRate** — Initial learning rate for solver `sgd`

`mdl.InitialLearningRate` (default) | positive real scalar value

Initial learning rate for solver `sgd`, specified as the comma-separated pair consisting of `'InitialLearningRate'` and a positive scalar value.

When using solver type `'sgd'`, the learning rate decays over iterations starting with the value specified for `'InitialLearningRate'`.

Example: `'InitialLearningRate',0.8`

Data Types: `double` | `single`

#### **PassLimit** — Maximum number of passes for solver `'sgd'`

`mdl.PassLimit` (default) | positive integer value

Maximum number of passes for solver `'sgd'` (stochastic gradient descent), specified as the comma-separated pair consisting of `'PassLimit'` and a positive integer. Every pass processes `size(mdl.X,1)` observations.

Example: `'PassLimit',10`

Data Types: `double` | `single`

### SGD or LBFSG or Mini-Batch LBFSG Options

#### **IterationLimit** — Maximum number of iterations

`mdl.IterationLimit` (default) | positive integer value

Maximum number of iterations, specified as the comma-separated pair consisting of `'IterationLimit'` and a positive integer.

Example: `'IterationLimit',250`

Data Types: `double` | `single`

## Output Arguments

#### **mdlrefit** — Neighborhood component analysis model for classification

`FeatureSelectionNCAClassification` object

Neighborhood component analysis model for classification, returned as a `FeatureSelectionNCAClassification` object. You can either save the results as a new model or update the existing model as `mdl = refit(mdl, Name, Value)`.

## Examples

### Refit NCA Model for Classification with Modified Settings

Generate checkerboard data using the `generateCheckerBoardData.m` function.

```
rng(2016, 'twister'); % For reproducibility
pps = 1375;
[X,y] = generateCheckerBoardData(pps);
X = X + 2;
```

Plot the data.

```
figure
plot(X(y==1,1),X(y==1,2), 'rx')
hold on
plot(X(y==-1,1),X(y==-1,2), 'bx')
```

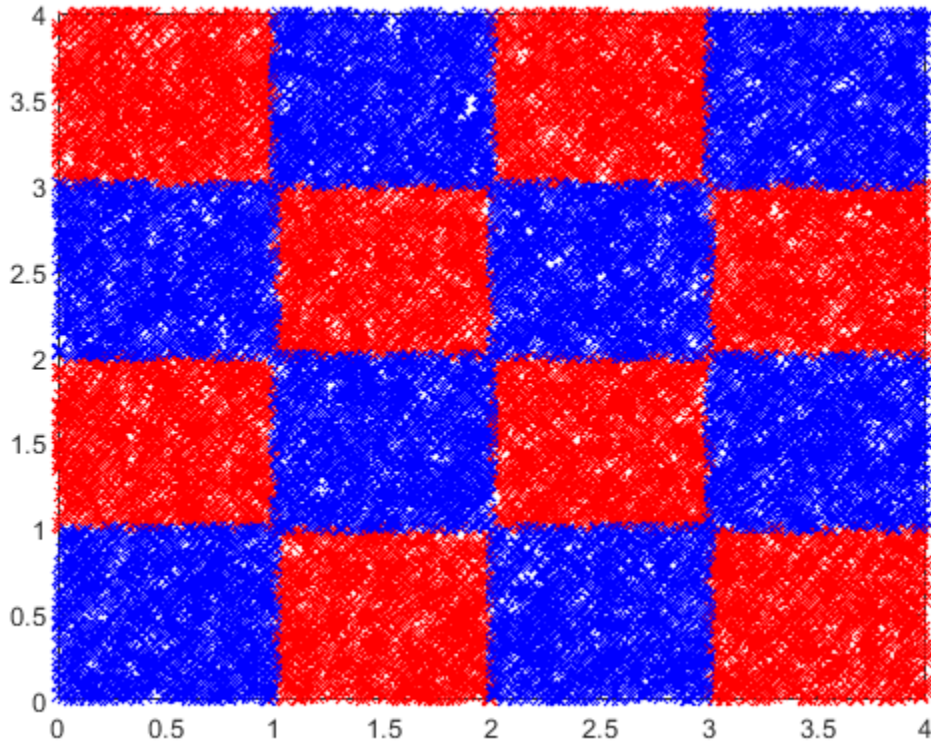
```
[n,p] = size(X)
```

```
n =
```

```
22000
```

```
p =
```

```
2
```



Add irrelevant predictors to the data.

```
Q = 98;  
Xrnd = unifrnd(0,4,n,Q);  
Xobs = [X,Xrnd];
```

This piece of code creates 98 additional predictors, all uniformly distributed between 0 and 4.

Partition the data into training and test sets. To create stratified partitions, so that each partition has similar proportion of classes, use `y` instead of `length(y)` as the partitioning criteria.

```
cvp = cvpartition(y, 'holdout', 2000);
```



cvpartition randomly chooses 2000 of the observations to add to the test set and the rest of the data to add to the training set. Create the training and validation sets using the assignments stored in the cvpartition object cvp.

```
Xtrain = Xobs(cvp.training(1),:);
ytrain = y(cvp.training(1),:);
```

```
Xval = Xobs(cvp.test(1),:);
yval = y(cvp.test(1),:);
```

Compute the misclassification error without feature selection.

```
nca = fscnca(Xtrain,ytrain,'FitMethod','none','Standardize',true, ...
    'Solver','lbfgs');
loss_nofs = loss(nca,Xval,yval)
```

```
loss_nofs =
    0.5165
```

'FitMethod','none' option uses the default weights (all 1s), which means all features are equally important.

This time, perform feature selection using neighborhood component analysis for classification, with  $\lambda = 1/n$ .

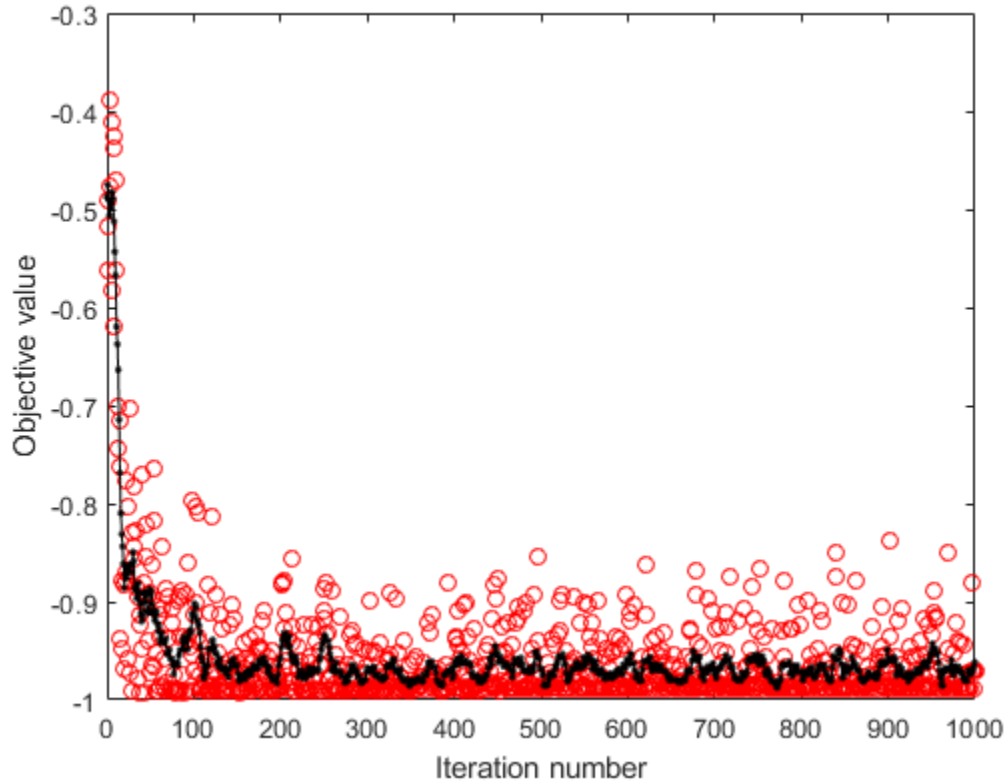
```
w0 = rand(100,1);
n = length(ytrain)
lambda = 1/n;
nca = refit(nca,'InitialFeatureWeights',w0,'FitMethod','exact', ...
    'Lambda',lambda,'solver','sgd');
```

```
n =
    20000
```

Plot the objective function value versus the iteration number.

```
figure()
plot(nca.FitInfo.Iteration,nca.FitInfo.Objective,'ro')
```

```
hold on
plot(nca.FitInfo.Iteration,movmean(nca.FitInfo.Objective,10),'k.-')
xlabel('Iteration number')
ylabel('Objective value')
```



Compute the misclassification error with feature selection.

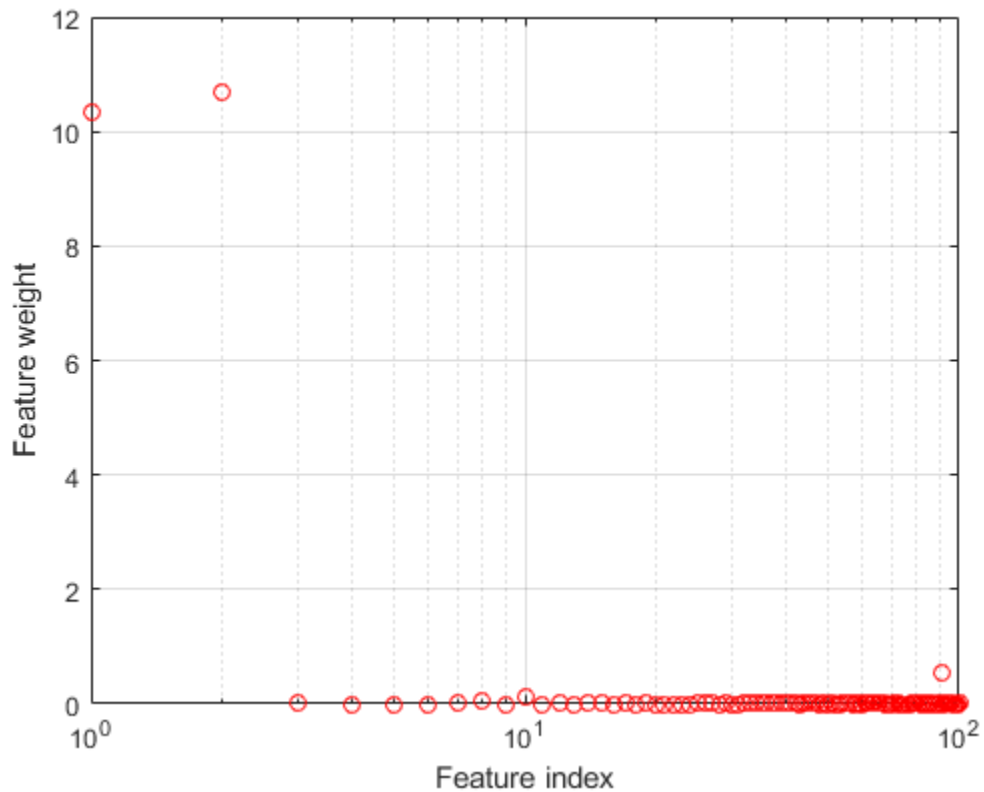
```
loss_withfs = loss(nca,Xval,yval)
```

```
loss_withfs =
```

```
0.0115
```

Plot the selected features.

```
figure
semilogx(nca.FeatureWeights, 'ro')
xlabel('Feature index')
ylabel('Feature weight')
grid on
```



Select features using the feature weights and a relative threshold.

```
tol = 0.15;
selidx = find(nca.FeatureWeights > tol*max(1,max(nca.FeatureWeights)))
```

```
selidx =
```

1  
2

Feature selection improves the results and `fscnca` detects the correct two features as relevant.

## See Also

`FeatureSelectionNCAClassification` | `fscnca` | `loss` | `predict`

**Introduced in R2016b**

## refit

**Class:** FeatureSelectionNCARegression

Refit neighborhood component analysis (NCA) model for regression

## Syntax

```
mdlrefit = refit(mdl, Name, Value)
```

## Description

`mdlrefit = refit(mdl, Name, Value)` refits the model `mdl`, with modified parameters specified by one or more `Name, Value` pair arguments.

## Input Arguments

**mdl** — Neighborhood component analysis model for regression

FeatureSelectionNCARegression object

Neighborhood component analysis model or classification, specified as a FeatureSelectionNCARegression object.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### Fitting Options

**FitMethod** — Method for fitting the model

`mdl.FitMethod` (default) | 'exact' | 'none' | 'average'

Method for fitting the model, specified as the comma-separated pair consisting of 'FitMethod' and one of the following.

- 'exact' — Performs fitting using all of the data.
- 'none' — No fitting. Use this option to evaluate the generalization error of the NCA model using the initial feature weights supplied in the call to `fsrnca`.
- 'average' — The function divides the data into partitions (subsets), fits each partition using the `exact` method, and returns the average of the feature weights. You can specify the number of partitions using the `NumPartitions` name-value pair argument.

Example: 'FitMethod', 'none'

Data Types: `char`

### **Lambda** — Regularization parameter

`mdl.Lambda` (default) | non-negative scalar value

Regularization parameter, specified as the comma-separated pair consisting of 'Lambda' and a non-negative scalar value.

For  $n$  observations, the best `Lambda` value that minimizes the generalization error of the NCA model is expected to be a multiple of  $1/n$

Example: 'Lambda', 0.01

Data Types: `double` | `single`

### **Solver** — Solver type

`mdl.Solver` (default) | 'lbfgs' | 'sgd' | 'minibatch-lbfgs'

Solver type for estimating feature weights, specified as the comma-separated pair consisting of 'Solver' and one of the following.

- 'lbfgs' — Limited memory BFGS (Broyden-Fletcher-Goldfarb-Shanno) algorithm (LBFGS algorithm)
- 'sgd' — Stochastic gradient descent
- 'minibatch-lbfgs' — Stochastic gradient descent with LBFGS algorithm applied to mini-batches

Example: 'solver', 'minibatch-lbfgs'

Data Types: char

### **InitialFeatureWeights** — Initial feature weights

`mdl.InitialFeatureWeights` (default) |  $p$ -by-1 vector of real positive scalar values

Initial feature weights, specified as the comma-separated pair consisting of 'InitialFeatureWeights' and a  $p$ -by-1 vector of real positive scalar values.

Data Types: double | single

### **Verbose** — Indicator for verbosity level

`mdl.Verbose` (default) | 0 | 1 | >1

Indicator for verbosity level for the convergence summary display, specified as the comma-separated pair consisting of 'Verbose' and one of the following.

- 0 — No convergence summary
- 1 — Convergence summary including iteration number, norm of the gradient, and objective function value.
- >1 — More convergence information depending on the fitting algorithm

When using solver 'minibatch-lbfgs' and verbosity level >1, the convergence information includes iteration log from intermediate minibatch LBFGS fits.

Example: 'Verbose',2

Data Types: double | single

### **LBFGS or Mini-Batch LBFGS Options**

#### **GradientTolerance** — Relative convergence tolerance

`mdl.GradientTolerance` (default) | positive real scalar value

Relative convergence tolerance on the gradient norm for solver `lbfgs`, specified as the comma-separated pair consisting of 'GradientTolerance' and a positive real scalar value.

Example: 'GradientTolerance',0.00001

Data Types: double | single

### SGD or Mini-Batch LBFGS Options

#### **InitialLearningRate** — Initial learning rate for solver `sgd`

`mdl.InitialLearningRate` (default) | positive real scalar value

Initial learning rate for solver `sgd`, specified as the comma-separated pair consisting of `'InitialLearningRate'` and a positive scalar value.

When using solver type `'sgd'`, the learning rate decays over iterations starting with the value specified for `'InitialLearningRate'`.

Example: `'InitialLearningRate',0.8`

Data Types: `double` | `single`

#### **PassLimit** — Maximum number of passes for solver `'sgd'`

`mdl.PassLimit` (default) | positive integer value

Maximum number of passes for solver `'sgd'` (stochastic gradient descent), specified as the comma-separated pair consisting of `'PassLimit'` and a positive integer. Every pass processes `size(mdl.X,1)` observations.

Example: `'PassLimit',10`

Data Types: `double` | `single`

### SGD or LBFGS or Mini-Batch LBFGS Options

#### **IterationLimit** — Maximum number of iterations

`mdl.IterationLimit` (default) | positive integer value

Maximum number of iterations, specified as the comma-separated pair consisting of `'IterationLimit'` and a positive integer.

Example: `'IterationLimit',250`

Data Types: `double` | `single`

## Output Arguments

#### **mdlrefit** — Neighborhood component analysis model for regression

FeatureSelectionNCARegression object



Neighborhood component analysis model or classification, returned as a FeatureSelectionNCARegression object. You can either save the results as a new model or update the existing model as `mdl = refit(mdl, Name, Value)`.

## Examples

### Refit NCA Model for Regression with Modified Settings

Load the sample data.

```
load(fullfile(matlabroot, 'examples', 'stats', 'robotarm.mat'))
```

The robotarm (pumadyn32nm) dataset is created using a robot arm simulator with 7168 training and 1024 test observations with 32 features [1], [2]. This is a preprocessed version of the original data set. Data are preprocessed by subtracting off a linear regression fit followed by normalization of all features to unit variance.

Compute the generalization error without feature selection.

```
nca = fsrnca(Xtrain, ytrain, 'FitMethod', 'none', 'Standardize', 1);
L = loss(nca, Xtest, ytest)
```

```
L =
```

```
0.9017
```

Now, refit the model and compute the prediction loss with feature selection, with  $\lambda = 0$  (no regularization term) and compare to the previous loss value, to determine feature selection seems necessary for this problem. For the settings that you do not change, refit uses the settings of the initial model nca. For example, it uses the feature weights found in nca as the initial feature weights.

```
nca2 = refit(nca, 'FitMethod', 'exact', 'Lambda', 0);
L2 = loss(nca2, Xtest, ytest)
```

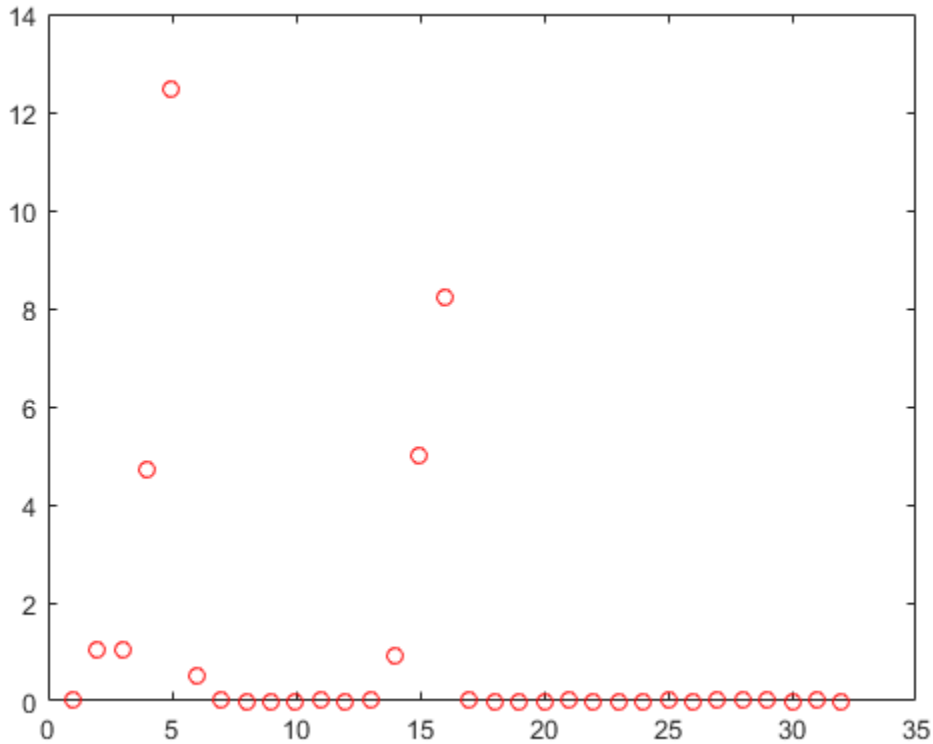
```
L2 =
```

```
0.1088
```

The decrease in the loss suggests that feature selection is necessary.

Plot the feature weights.

```
figure()  
plot(nca2.FeatureWeights, 'ro')
```



Tuning the regularization parameter usually improves the results. Suppose that, after tuning  $\lambda$  using cross-validation as in “Tune Regularization Parameter in NCA for Regression” on page 32-2757, the best  $\lambda$  value found is 0.0035. Refit the nca model using this  $\lambda$  value and stochastic gradient descent as the solver. Compute the prediction loss.

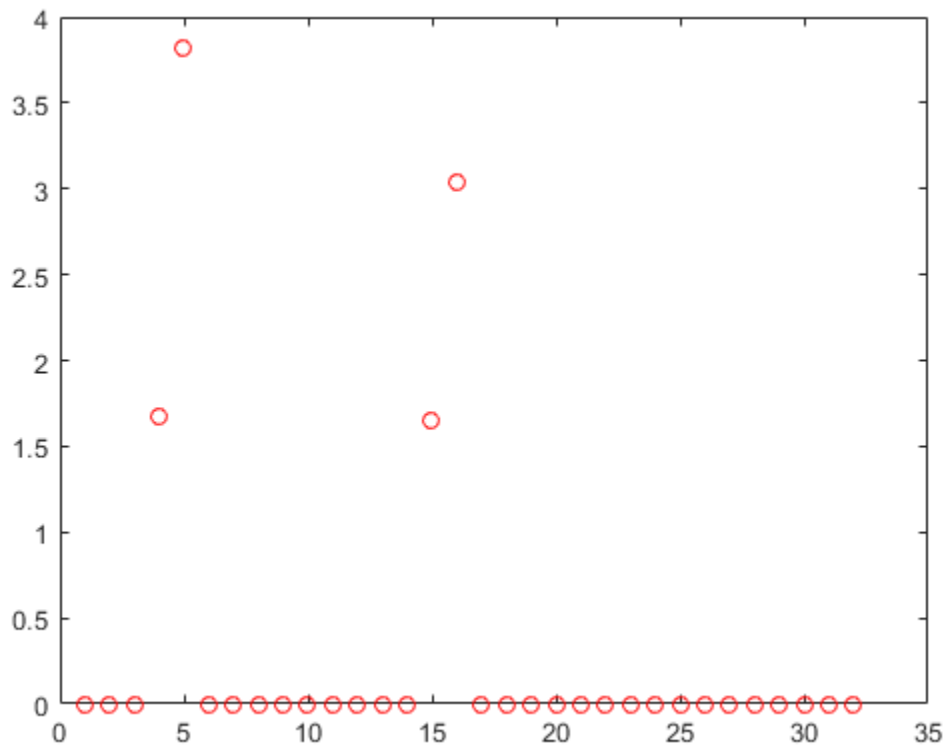
```
nca3 = refit(nca2, 'FitMethod', 'exact', 'Lambda', 0.0035, ...  
            'Solver', 'sgd');  
L3 = loss(nca3, Xtest, ytest)
```

```
L3 =
```

```
0.0573
```

Plot the feature weights.

```
figure()  
plot(nca3.FeatureWeights, 'ro')
```



After tuning the regularization parameter, the loss decreased even more and the software identified four of the features as relevant.

### References

[1] Rasmussen, C. E., R. M. Neal, G. E. Hinton, D. van Campand, M. Revow, Z. Ghahramani, R. Kustra, R. Tibshirani. The DELVE Manual, 1996, <http://mlg.eng.cam.ac.uk/pub/pdf/RasNeaHinetal96.pdf>

[2] <http://www.cs.toronto.edu/~delve/data/datasets.html>

### See Also

`FeatureSelectionNCARegression` | `fsrcna` | `loss` | `predict`

**Introduced in R2016b**

# refline

Add reference line to plot

## Syntax

```
refline(m,b)
refline(coeffs)
refline
refline(ax, ___)
hline = refline( ___ )
```

## Description

`refline(m,b)` adds a reference line with slope  $m$  and intercept  $b$  to the current axes.

`refline(coeffs)`, where `coeffs` is a two-element coefficient vector, adds the line

$$y = \text{coeffs}(1)*x + \text{coeffs}(2)$$

to the figure.

`refline` with no input arguments is equivalent to `lsline`.

`refline(ax, ___)` adds a reference line to the plot in the axis specified by `ax`, using any of the previous syntaxes.

`hline = refline( ___ )` returns the handle `hline` to the line.

## Examples

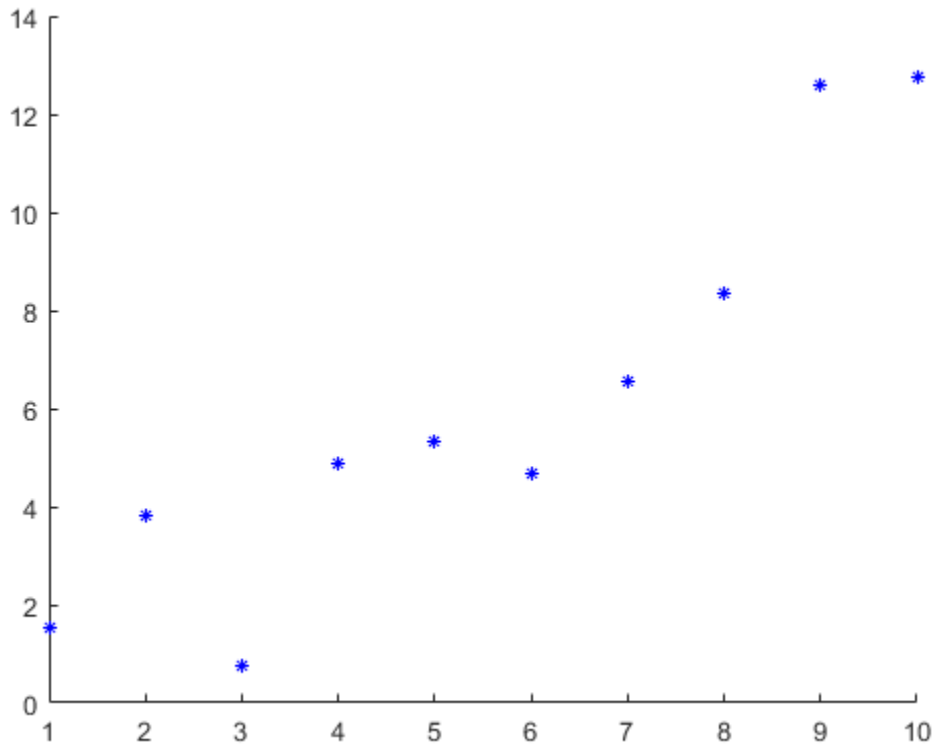
### Add a Reference Line at the Mean

Generate sample data for independent variable  $x$  and a dependent variable  $y$ .

```
x = 1:10;  
y = x + randn(1,10);
```

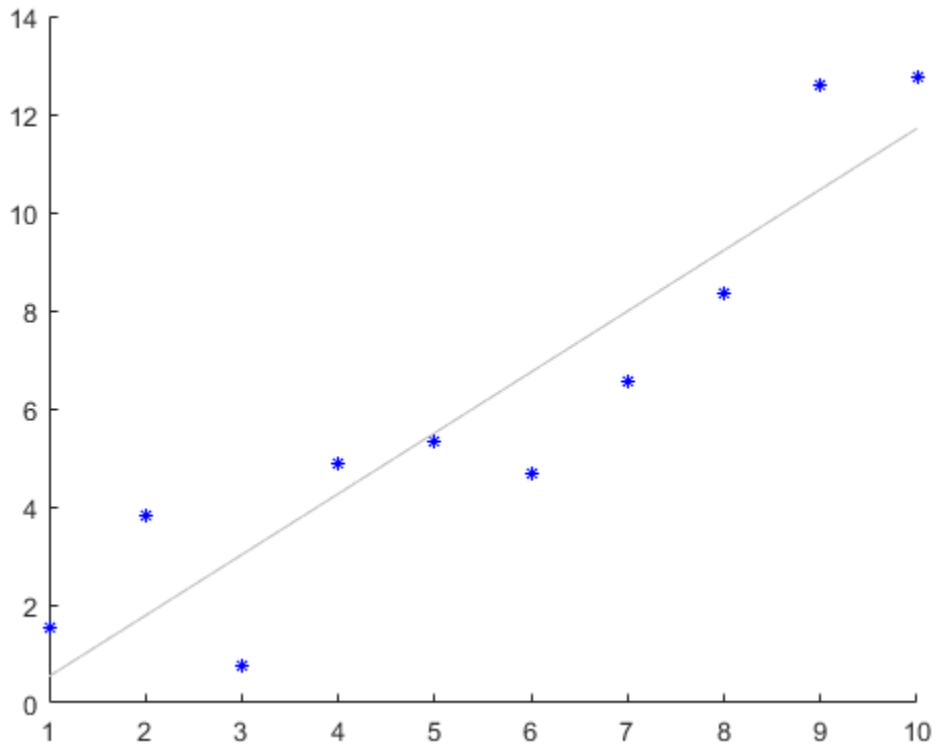
Create a scatter plot of  $x$  and  $y$ .

```
scatter(x,y,25,'b','*')
```



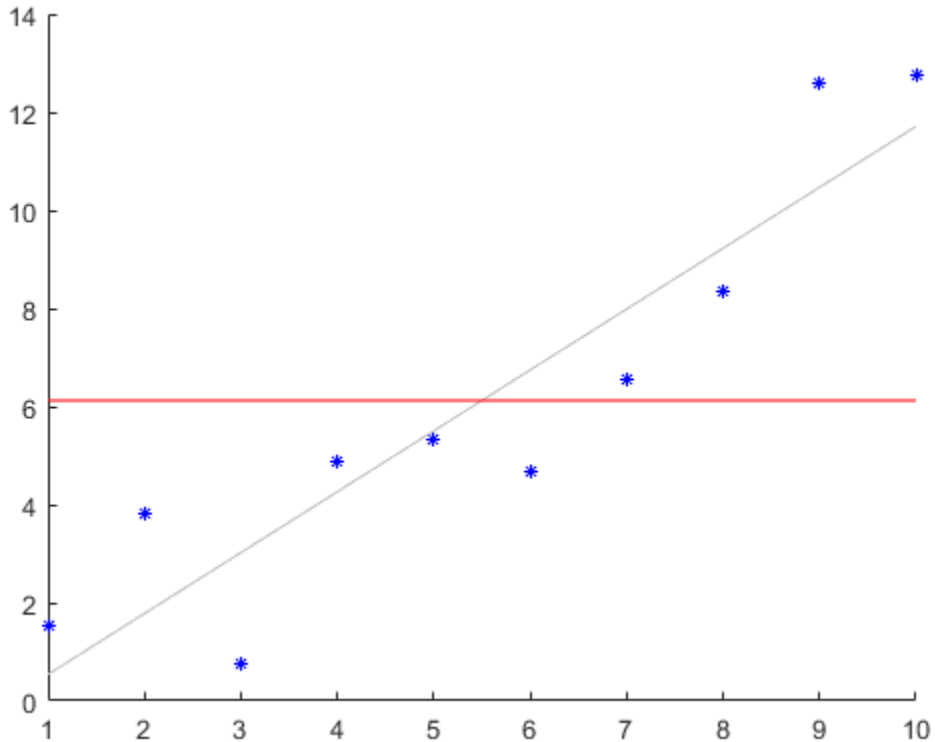
Superimpose a least-squares line on the scatter plot.

```
lsline
```



Add a reference line at the mean of the scatter and its least-squares line.

```
mu = mean(y);  
hline = reffline([0 mu]);  
hline.Color = 'r';
```



The red line shows the reference line at the mean of data.

### Specify Axes for Least-Squares and Reference Lines

Define the x-variable and two different y-variables to use for the plots.

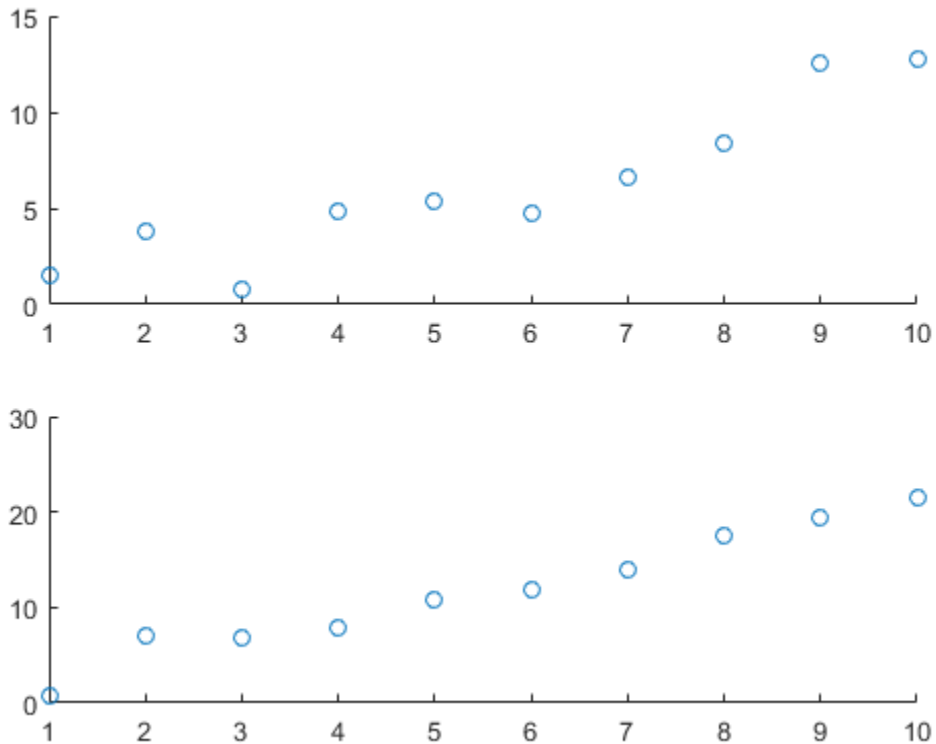
```
rng default % For reproducibility
x = 1:10;
y1 = x + randn(1,10);
y2 = 2*x + randn(1,10);
```



Define `ax1` as the top half of the figure, and `ax2` as the bottom half of the figure. Create the first scatter plot on the top axis using `y1`, and the second scatter plot on the bottom axis using `y2`.

```
figure
ax1 = subplot(2,1,1);
ax2 = subplot(2,1,2);

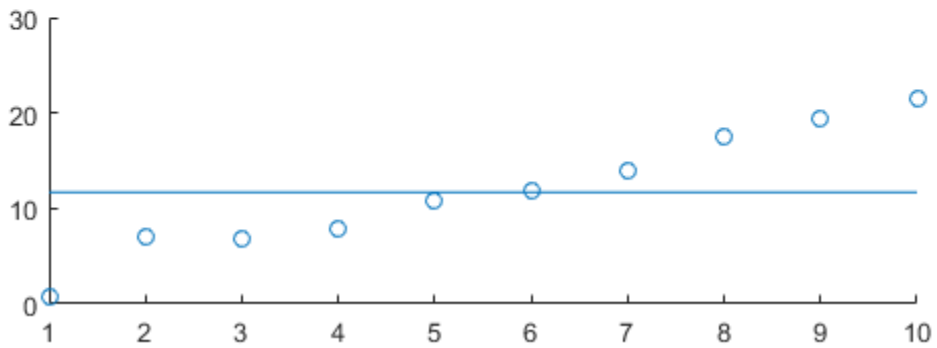
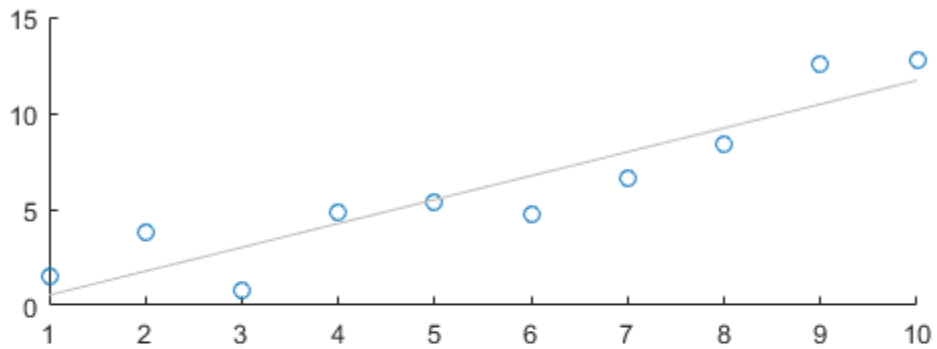
scatter(ax1,x,y1)
scatter(ax2,x,y2)
```



Superimpose a least-squares line on the top plot, and a reference line at the mean of the `y2` values in the bottom plot.

```
lsline(ax1)

mu = mean(y2);
refline(ax2,[0 mu])
```



## See Also

`gline` | `lsline` | `refcurve`

Introduced before R2006a

# regress

Multiple linear regression

## Syntax

```
b = regress(y,X)
[b,bint] = regress(y,X)
[b,bint,r] = regress(y,X)
[b,bint,r,rint] = regress(y,X)
[b,bint,r,rint,stats] = regress(y,X)
[...] = regress(y,X,alpha)
```

## Description

`b = regress(y,X)` returns a  $p$ -by-1 vector `b` of coefficient estimates for a multilinear regression of the responses in `y` on the predictors in `X`. `X` is an  $n$ -by- $p$  matrix of  $p$  predictors at each of  $n$  observations. `y` is an  $n$ -by-1 vector of observed responses.

`regress` treats NaNs in `X` or `y` as missing values, and ignores them.

If the columns of `X` are linearly dependent, `regress` obtains a basic solution by setting the maximum number of elements of `b` to zero.

`[b,bint] = regress(y,X)` returns a  $p$ -by-2 matrix `bint` of 95% confidence intervals for the coefficient estimates. The first column of `bint` contains lower confidence bounds for each of the  $p$  coefficient estimates; the second column contains upper confidence bounds.

If the columns of `X` are linearly dependent, `regress` returns zeros in elements of `bint` corresponding to the zero elements of `b`.

`[b,bint,r] = regress(y,X)` returns an  $n$ -by-1 vector `r` of residuals.

`[b,bint,r,rint] = regress(y,X)` returns an  $n$ -by-2 matrix `rint` of intervals that can be used to diagnose outliers. If the interval `rint(i,:)` for observation `i` does not

contain zero, the corresponding residual is larger than expected in 95% of new observations, suggesting an outlier.

In a linear model, observed values of  $y$  are random variables, and so are their residuals. Residuals have normal distributions with zero mean but with different variances at different values of the predictors. To put residuals on a comparable scale, they are “Studentized,” that is, they are divided by an estimate of their standard deviation that is independent of their value. Studentized residuals have  $t$  distributions with known degrees of freedom. The intervals returned in `rint` are shifts of the 95% confidence intervals of these  $t$  distributions, centered at the residuals.

`[b,bint,r,rint,stats] = regress(y,X)` returns a 1-by-4 vector `stats` that contains, in order, the  $R^2$  statistic, the  $F$  statistic and its  $p$  value, and an estimate of the error variance.

---

**Note** When computing statistics, `X` should include a column of 1s so that the model contains a constant term. The  $F$  statistic and its  $p$  value are computed under this assumption, and they are not correct for models without a constant.

The  $F$  statistic is the test statistic of the F-test on the regression model, for a significant linear regression relationship between the response variable and the predictor variables.

The  $R^2$  statistic can be negative for models without a constant, indicating that the model is not appropriate for the data.

---

`[...] = regress(y,X,alpha)` uses a  $100*(1-\text{alpha})\%$  confidence level to compute `bint` and `rint`.

## Examples

### Estimate Multiple Linear Regression Coefficients

This example shows how to estimate the coefficients of a multiple linear regression.

Load the sample data. Identify weight and horsepower as predictors, and mileage as the response.

```
load carsmall
x1 = Weight;
x2 = Horsepower; % Contains NaN data
y = MPG;
```

Compute the regression coefficients for a linear model with an interaction term.

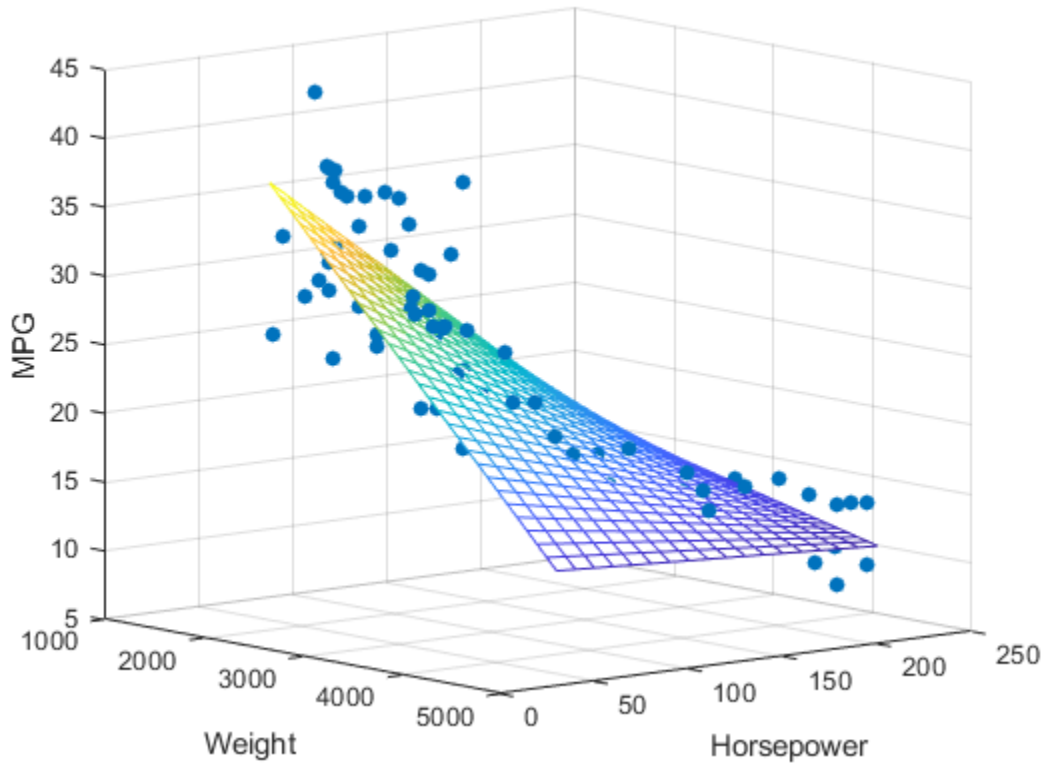
```
X = [ones(size(x1)) x1 x2 x1.*x2];
b = regress(y,X) % Removes NaN data
```

```
b =

    60.7104
   -0.0102
   -0.1882
    0.0000
```

Plot the data and the model.

```
scatter3(x1,x2,y,'filled')
hold on
x1fit = min(x1):100:max(x1);
x2fit = min(x2):10:max(x2);
[X1FIT,X2FIT] = meshgrid(x1fit,x2fit);
YFIT = b(1) + b(2)*X1FIT + b(3)*X2FIT + b(4)*X1FIT.*X2FIT;
mesh(X1FIT,X2FIT,YFIT)
xlabel('Weight')
ylabel('Horsepower')
zlabel('MPG')
view(50,10)
```



- “Interpret Linear Regression Results” on page 11-64
- “Linear Regression Workflow” on page 11-43

## References

- [1] Chatterjee, S., and A. S. Hadi. “Influential Observations, High Leverage Points, and Outliers in Linear Regression.” *Statistical Science*. Vol. 1, 1986, pp. 379–416.

## See Also

`LinearModel` | `fitlm` | `mvregress` | `rcoplot` | `stepwiselm`

## Topics

“Interpret Linear Regression Results” on page 11-64

“Linear Regression Workflow” on page 11-43

**Introduced before R2006a**

## RegressionBaggedEnsemble class

### Superclasses:

Regression ensemble grown by resampling

## Description

`RegressionBaggedEnsemble` combines a set of trained weak learner models and data on which these learners were trained. It can predict ensemble response for new data by aggregating predictions from its weak learners.

## Construction

Create a bagged regression ensemble object using `fitensemble`.

## Properties

### **CategoricalPredictors**

Indices of categorical predictors, stored as a vector of positive integers. `CategoricalPredictors` contains index values corresponding to the columns of the predictor data that contain categorical predictors. If none of the predictors are categorical, then this property is empty (`[]`).

### **CombineWeights**

A character vector describing how the ensemble combines learner predictions.

### **ExpandedPredictorNames**

Expanded predictor names, stored as a cell array of character vectors.

If the model uses encoding for categorical variables, then `ExpandedPredictorNames` includes the names that describe the expanded variables. Otherwise, `ExpandedPredictorNames` is the same as `PredictorNames`.



**FitInfo**

A numeric array of fit information. The `FitInfoDescription` property describes the content of this array.

**FitInfoDescription**

Character vector describing the meaning of the `FitInfo` array.

**FResample**

A numeric scalar between 0 and 1. `FResample` is the fraction of training data `fitrensemble` resampled at random for every weak learner when constructing the ensemble.

**HyperparameterOptimizationResults**

Description of the cross-validation optimization of hyperparameters, stored as a `BayesianOptimization` object or a table of hyperparameters and associated values. Nonempty when the `OptimizeHyperparameters` name-value pair is nonempty at creation. Value depends on the setting of the `HyperparameterOptimizationOptions` name-value pair at creation:

- 'bayesopt' (default) — Object of class `BayesianOptimization`
- 'gridsearch' or 'randomsearch' — Table of hyperparameters used, observed objective function values (cross-validation loss), and rank of observations from lowest (best) to highest (worst)

**LearnerNames**

Cell array of character vectors with names of the weak learners in the ensemble. The name of each learner appears just once. For example, if you have an ensemble of 100 trees, `LearnerNames` is {'Tree'}.

**Method**

A character vector with the name of the algorithm `fitrensemble` used for training the ensemble.

**ModelParameters**

Parameters used in training `ens`.

**NumObservations**

Numeric scalar containing the number of observations in the training data.

**NumTrained**

Number of trained learners in the ensemble, a positive scalar.

**PredictorNames**

A cell array of names for the predictor variables, in the order in which they appear in `X`.

**ReasonForTermination**

A character vector describing the reason `fitrensemble` stopped adding weak learners to the ensemble.

**Regularization**

A structure containing the result of the `regularize` method. Use `Regularization` with `shrink` to lower resubstitution error and `shrink` the ensemble.

**Replace**

Boolean flag indicating if training data for weak learners in this ensemble were sampled with replacement. `Replace` is `true` for sampling with replacement, `false` otherwise.

**ResponseName**

A character vector with the name of the response variable `Y`.

**ResponseTransform**

Function handle for transforming scores, or character vector representing a built-in transformation function. 'none' means no transformation; equivalently, 'none' means `@(x) x`.

Add or change a `ResponseTransform` function using dot notation:

```
ens.ResponseTransform = @function
```

**Trained**

The trained learners, a cell array of compact regression models.

**TrainedWeights**

A numeric vector of weights the ensemble assigns to its learners. The ensemble computes predicted response by aggregating weighted predictions from its learners.

**UseObsForLearner**

A logical matrix of size  $N$ -by- $\text{NumTrained}$ , where  $N$  is the number of rows (observations) in the training data  $X$ , and  $\text{NumTrained}$  is the number of trained weak learners.

`UseObsForLearner(I, J)` is true if observation  $I$  was used for training learner  $J$ , and is false otherwise.

**W**

The scaled `weights`, a vector with length  $n$ , the number of rows in  $X$ . The sum of the elements of  $W$  is 1.

**X**

The matrix of predictor values that trained the ensemble. Each column of  $X$  represents one variable, and each row represents one observation.

**Y**

The numeric column vector with the same number of rows as  $X$  that trained the ensemble. Each entry in  $Y$  is the response to the data in the corresponding row of  $X$ .

**Methods**

<code>oobLoss</code>	Out-of-bag regression error
<code>oobPermutedPredictorImportance</code>	Predictor importance estimates by permutation of out-of-bag predictor observations for random forest of regression trees
<code>oobPredict</code>	Predict out-of-bag response of ensemble

## Inherited Methods

<code>compact</code>	Create compact regression ensemble
<code>crossval</code>	Cross validate ensemble
<code>cvshrink</code>	Cross validate shrinking (pruning) ensemble
<code>regularize</code>	Find weights to minimize resubstitution error plus penalty term
<code>resubLoss</code>	Regression error by resubstitution
<code>resubPredict</code>	Predict response of ensemble by resubstitution
<code>resume</code>	Resume training ensemble
<code>shrink</code>	Prune ensemble
<code>loss</code>	Regression error
<code>predict</code>	Predict responses using ensemble of regression models
<code>predictorImportance</code>	Estimates of predictor importance
<code>removeLearners</code>	Remove members of compact regression ensemble

## Copy Semantics

Value. To learn how value classes affect copy operations, see [Copying Objects \(MATLAB\)](#).

## Examples

### Train Bagged Ensemble of Regression Trees

Load the `carsmall` data set. Consider a model that explains a car's fuel economy (MPG) using its weight (`Weight`) and number of cylinders (`Cylinders`).

```
load carsmall
X = [Weight Cylinders];
Y = MPG;
```

Train a bagged ensemble of 100 regression trees using all measurements.

```
rng(1) % For reproducibility
Mdl = fitensemble(X,Y,'bag',100,'Tree','Type','regression')

Mdl =

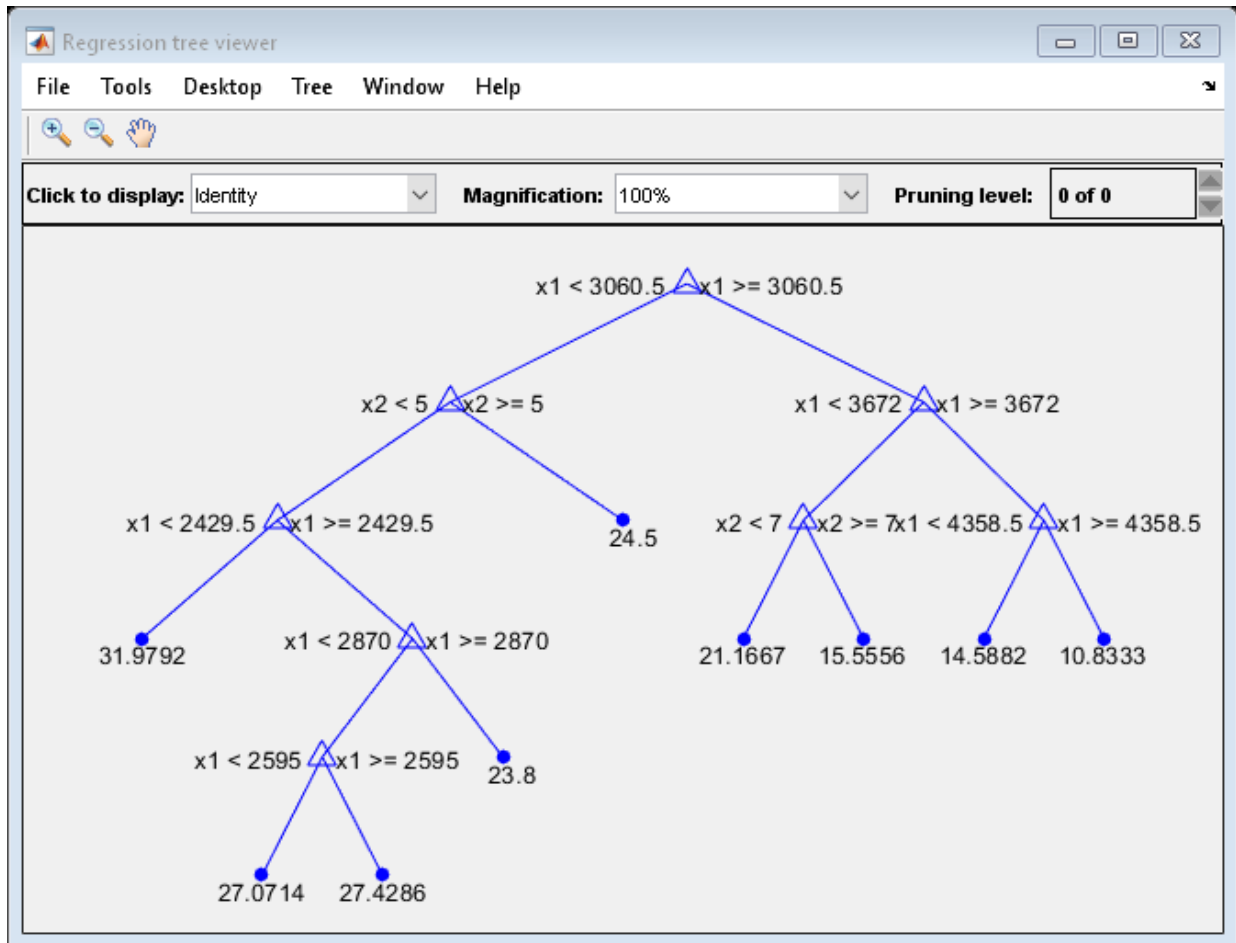
classreg.learning.regr.RegressionBaggedEnsemble
    ResponseName: 'Y'
    CategoricalPredictors: []
    ResponseTransform: 'none'
    NumObservations: 94
    NumTrained: 100
    Method: 'Bag'
    LearnerNames: {'Tree'}
    ReasonForTermination: 'Terminated normally after completing the requested number o
    FitInfo: []
    FitInfoDescription: 'None'
    Regularization: []
    FResample: 1
    Replace: 1
    UseObsForLearner: [94x100 logical]
```

Mdl is a RegressionBaggedEnsemble model object.

Mdl.Trained is the property that stores a 100-by-1 cell vector of the trained, compact regression trees (CompactRegressionTree model objects) that compose the ensemble.

Plot a graph of the first trained regression tree.

```
view(Mdl.Trained{1}, 'Mode', 'graph')
```



By default, `fitensemble` grows deep trees for bags of trees.

Estimate the in-sample mean-squared error (MSE).

```
L = resubLoss(Mdl)
```

```
L =
```

12.5791

## Tip

For a bagged ensemble of regression trees, the `Trained` property of `ens` stores a cell vector of `ens.NumTrained CompactRegressionTree` model objects. For a textual or graphical display of tree `t` in the cell vector, enter

```
view(ens.Trained{t})
```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Only the `predict` function supports code generation.
- When you train an ensemble by using `fitensemble`, code generation limitations for regression trees also apply to ensembles of regression trees. For more details, see “Code Generation” on page 32-0 of the `CompactRegressionTree` class.

### See Also

`RegressionEnsemble` | `fitensemble` | `plotPartialDependence` | `view`

Introduced in R2011a

## RegressionEnsemble class

### Superclasses:

Ensemble regression

## Description

`RegressionEnsemble` combines a set of trained weak learner models and data on which these learners were trained. It can predict ensemble response for new data by aggregating predictions from its weak learners.

## Construction

Create a regression ensemble object using `fitensemble`.

## Properties

### **CategoricalPredictors**

Indices of categorical predictors, stored as a vector of positive integers. `CategoricalPredictors` contains index values corresponding to the columns of the predictor data that contain categorical predictors. If none of the predictors are categorical, then this property is empty (`[]`).

### **CombineWeights**

A character vector describing how the ensemble combines learner predictions.

### **ExpandedPredictorNames**

Expanded predictor names, stored as a cell array of character vectors.

If the model uses encoding for categorical variables, then `ExpandedPredictorNames` includes the names that describe the expanded variables. Otherwise, `ExpandedPredictorNames` is the same as `PredictorNames`.



**FitInfo**

A numeric array of fit information. The `FitInfoDescription` property describes the content of this array.

**FitInfoDescription**

Character vector describing the meaning of the `FitInfo` array.

**LearnerNames**

Cell array of character vectors with names of the weak learners in the ensemble. The name of each learner appears just once. For example, if you have an ensemble of 100 trees, `LearnerNames` is `{ 'Tree' }`.

**HyperparameterOptimizationResults**

Description of the cross-validation optimization of hyperparameters, stored as a `BayesianOptimization` object or a table of hyperparameters and associated values. Nonempty when the `OptimizeHyperparameters` name-value pair is nonempty at creation. Value depends on the setting of the `HyperparameterOptimizationOptions` name-value pair at creation:

- `'bayesopt'` (default) — Object of class `BayesianOptimization`
- `'gridsearch'` or `'randomsearch'` — Table of hyperparameters used, observed objective function values (cross-validation loss), and rank of observations from lowest (best) to highest (worst)

**Method**

A character vector with the name of the algorithm `fitrensemble` used for training the ensemble.

**ModelParameters**

Parameters used in training `ens`.

**NumObservations**

Numeric scalar containing the number of observations in the training data.

**NumTrained**

Number of trained learners in the ensemble, a positive scalar.

**PredictorNames**

A cell array of names for the predictor variables, in the order in which they appear in X.

**ReasonForTermination**

A character vector describing the reason `fitrensemble` stopped adding weak learners to the ensemble.

**Regularization**

A structure containing the result of the `regularize` method. Use `Regularization` with `shrink` to lower resubstitution error and shrink the ensemble.

**ResponseName**

A character vector with the name of the response variable Y.

**ResponseTransform**

Function handle for transforming scores, or character vector representing a built-in transformation function. 'none' means no transformation; equivalently, 'none' means  $@(x) x$ .

Add or change a `ResponseTransform` function using dot notation:

```
ens.ResponseTransform = @function
```

**Trained**

The trained learners, a cell array of compact regression models.

**TrainedWeights**

A numeric vector of weights the ensemble assigns to its learners. The ensemble computes predicted response by aggregating weighted predictions from its learners.

**W**

The scaled `weights`, a vector with length `n`, the number of rows in `X`. The sum of the elements of `W` is 1.

**X**

The matrix of predictor values that trained the ensemble. Each column of `X` represents one variable, and each row represents one observation.

**Y**

The numeric column vector with the same number of rows as `X` that trained the ensemble. Each entry in `Y` is the response to the data in the corresponding row of `X`.

## Methods

<code>compact</code>	Create compact regression ensemble
<code>crossval</code>	Cross validate ensemble
<code>cvshrink</code>	Cross validate shrinking (pruning) ensemble
<code>regularize</code>	Find weights to minimize resubstitution error plus penalty term
<code>resubLoss</code>	Regression error by resubstitution
<code>resubPredict</code>	Predict response of ensemble by resubstitution
<code>resume</code>	Resume training ensemble
<code>shrink</code>	Prune ensemble

## Inherited Methods

<code>loss</code>	Regression error
<code>predict</code>	Predict responses using ensemble of regression models
<code>predictorImportance</code>	Estimates of predictor importance
<code>removeLearners</code>	Remove members of compact regression ensemble

## Copy Semantics

Value. To learn how value classes affect copy operations, see [Copying Objects \(MATLAB\)](#).

## Examples

### Train Boosted Regression Ensemble

Load the `ionosphere` data set.

```
load ionosphere
```

Load the `carsmall` data set. Consider a model that explains a car's fuel economy (MPG) using its weight (`Weight`) and number of cylinders (`Cylinders`).

```
load carsmall
X = [Weight Cylinders];
Y = MPG;
```

Train a boosted ensemble of 100 regression trees using the `LSBoost`. Specify that `Cylinders` is a categorical variable.

```
Mdl = fitensemble(X,Y,'LSBoost',100,'Tree','PredictorNames',{'W','C'},...
    'CategoricalPredictors',2)
```

```
Mdl =
```

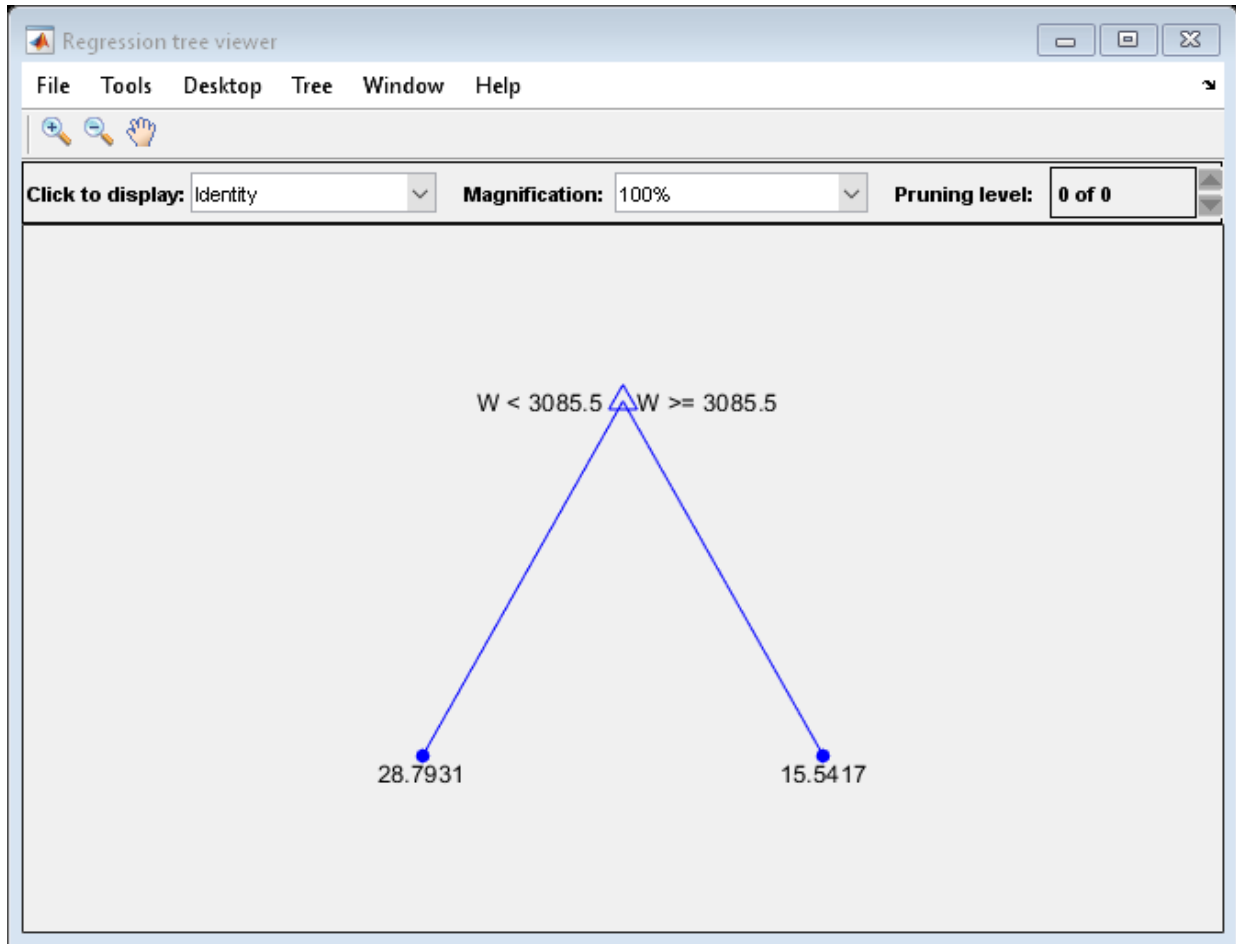
```
classreg.learning.regr.RegressionEnsemble
    PredictorNames: {'W' 'C'}
    ResponseName: 'Y'
    CategoricalPredictors: 2
    ResponseTransform: 'none'
    NumObservations: 94
    NumTrained: 100
    Method: 'LSBoost'
    LearnerNames: {'Tree'}
    ReasonForTermination: 'Terminated normally after completing the requested number of iterations'
    FitInfo: [100x1 double]
    FitInfoDescription: {2x1 cell}
    Regularization: []
```

Mdl is a RegressionEnsemble model object that contains the training data, among other things.

Mdl.Trained is the property that stores a 100-by-1 cell vector of the trained regression trees (CompactRegressionTree model objects) that compose the ensemble.

Plot a graph of the first trained regression tree.

```
view(Mdl.Trained{1}, 'Mode', 'graph')
```



By default, `fitensemble` grows stumps for boosted ensembles of trees.

Predict the fuel economy of 4,000 pound cars with 4, 6, and 8 cylinders.

```
XNew = [4000*ones(3,1) [4; 6; 8]];
mpgNew = predict(Mdl,XNew)
```

```
mpgNew =
    19.3228
```

```
16.4509  
14.5549
```

## Tip

For an ensemble of regression trees, the `Trained` property contains a cell vector of `ens.NumTrained` `CompactRegressionTree` model objects. For a textual or graphical display of tree `t` in the cell vector, enter

```
view(ens.Trained{t})
```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Only the `predict` function supports code generation.
- When you train an ensemble by using `fitensemble`, code generation limitations for regression trees also apply to ensembles of regression trees. For more details, see “Code Generation” on page 32-0 of the `CompactRegressionTree` class.

### See Also

`ClassificationEnsemble` | `CompactRegressionEnsemble` | `fitensemble` | `plotPartialDependence` | `templateTree` | `view`

Introduced in R2011a

## RegressionGP class

### Superclasses:

Gaussian process regression model class

### Description

`RegressionGP` is a Gaussian process regression (GPR) model. You can train a GPR model, using `fitrgp`. Using the trained model, you can

- Predict responses for training data using `resubPredict` or new predictor data using `predict`. You can also compute the prediction intervals.
- Compute the regression loss for training data using `resubLoss` or new data using `loss`.

### Construction

`gprMdl = fitrgp(tbl, ResponseVarName)` returns a GPR model trained using the sample data in `tbl`, where `ResponseVarName` is the name of the response variable in `tbl`.

`gprMdl = fitrgp(tbl, formula)` returns a Gaussian process regression (GPR) model, trained using the sample data in `tbl`, for the predictor variables and response variables identified by `formula`.

`gprMdl = fitrgp(tbl, y)` returns a GPR model for the predictors in table `tbl` and continuous response vector `y`.

`gprMdl = fitrgp(X, y)` returns a GPR model for predictors `X` and continuous response vector `y`.

`gprMdl = fitrgp( ____, Name, Value)` returns a GPR model for any of the input arguments in the previous syntaxes, with additional options specified by one or more `Name, Value` pair arguments.



For example, you can specify the fitting method, the prediction method, the covariance function, or the active set selection method.

You can also train a cross-validated model. If you train a cross-validated model, the output is a `RegressionPartitionedModel` object. For further analysis on the cross-validated object, use the methods of `RegressionPartitionedModel` class. For the methods of this class, see the `RegressionPartitionedModel` class page.

## Input Arguments

### **tbl** — Sample data

table

Sample data used to train the model, specified as a `table`. Each row of `tbl` corresponds to one observation, and each column corresponds to one variable. `tbl` contains the predictor variables, and optionally it can also contain one column for the response variable. Multi-column variables and cell arrays other than cell arrays of character vectors are not allowed.

- If `tbl` contains the response variable, and you want to use all the remaining variables as predictors, then specify the response variable using `ResponseVarName`.
- If `tbl` contains the response variable, and you want to use only a subset of the predictors in training the model, then specify the response variable and the predictor variables using `formula`.
- If `tbl` does not contain the response variable, then specify a response variable using `y`. The length of the response variable and the number of rows in `tbl` must be equal.

For more information on the `table` data type, see `table`.

If your predictor data contains categorical variables, then the software uses full dummy coding for these variables. The software creates one dummy variable for each level of the categorical variable.

Data Types: `table`

### **ResponseVarName** — Response variable name

name of a variable in `tbl`

Response variable name, specified as the name of a variable in `tbl`. You must specify `ResponseVarName` as a character vector. For example, if the response variable `y` is

stored in `tbl` (as `tbl.y`), then specify it as `'y'`. Otherwise, the software treats all the columns of `tbl`, including `y` as predictors when training the model.

Data Types: `char`

**formula** — Response and predictor variables to use in model training

character vector in the form of `'y~x1+x2+x3'`

Response and predictor variables to use in model training, specified as a character vector in the form of `'y~x1+x2+x3'`. In this form, `y` represents the response variable, `x1`, `x2`, `x3` represent the predictor variables to use in training the model.

Use `formula` if you want to specify a subset of variables in `tbl` as predictors to use when training the model. If you specify a formula, then any variables that do not appear in `formula` are not used to train the model.

The formula does not indicate the form of the `BasisFunction`.

Example: `'PetalLength~PetalWidth+Species'` identifies the variable `PetalLength` as the response variable, and `PetalWidth` and `Species` as the predictor variables.

**x** — Predictor data for the GPR model

*n*-by-*d* matrix

Predictor data for the GPR model, specified as an *n*-by-*d* matrix. *n* is the number of observations (rows), and *d* is the number of predictors (columns).

The length of `y` and the number of rows of `x` must be equal.

To specify the names of the predictors in the order of their appearance in `x`, use the `PredictorNames` name-value pair argument.

Data Types: `double`

**y** — Response data for the GPR model

*n*-by-1 vector

Response data for the GPR model, specified as an *n*-by-1 vector. You can omit `y` if you provide the training data in `tbl` that also includes `y`. In that case, use `ResponseVarName` to identify the response variable or use `formula` to identify the response and predictor variables.

Data Types: `double` | `char`

## Properties

### Fitting

#### FitMethod — Method used to estimate the parameters

'none' | 'exact' | 'sd' | 'sr' | 'fic'

Method used to estimate the basis function coefficients,  $\beta$ ; noise standard deviation,  $\sigma$ ; and kernel parameters,  $\theta$ , of the GPR model, stored as a character vector. It can be one of the following.

Fit Method	Description
'none'	No estimation. <code>fitrgp</code> uses the initial parameter values as the parameter values.
'exact'	Exact Gaussian process regression.
'sd'	Subset of data points approximation.
'sr'	Subset of regressors approximation.
'fic'	Fully independent conditional approximation.

#### BasisFunction — Explicit basis function

'none' | 'constant' | 'linear' | 'pureQuadratic' | function handle

Explicit basis function used in the GPR model, stored as a character vector or a function handle. It can be one of the following. If  $n$  is the number of observations, the basis function adds the term  $\mathbf{H}^*\boldsymbol{\beta}$  to the model, where  $\mathbf{H}$  is the basis matrix and  $\boldsymbol{\beta}$  is a  $p$ -by-1 vector of basis coefficients.

Explicit Basis	Basis Matrix
'none'	Empty matrix.
'constant'	$\mathbf{H} = \mathbf{1}$ ( $n$ -by-1 vector of 1s, where $n$ is the number of observations)
'linear'	$\mathbf{H} = [\mathbf{1}, \mathbf{X}]$

Explicit Basis	Basis Matrix
'pureQuadratic'	$\mathbf{H} = [\mathbf{1}, \mathbf{X}, \mathbf{X}_2],$ <p>where</p> $\mathbf{X}_2 = \begin{bmatrix} x_{11}^2 & x_{12}^2 & \cdots & x_{1d}^2 \\ x_{21}^2 & x_{22}^2 & \cdots & x_{2d}^2 \\ \vdots & \vdots & \vdots & \vdots \\ x_{n1}^2 & x_{n2}^2 & \cdots & x_{nd}^2 \end{bmatrix}.$
Function handle	<p>Function handle, <code>hfcn</code>, that <code>fitrgp</code> calls as:</p> $\mathbf{H} = \text{hfcn}(\mathbf{X}),$ <p>where <math>\mathbf{X}</math> is an <math>n</math>-by-<math>d</math> matrix of predictors and <math>\mathbf{H}</math> is an <math>n</math>-by-<math>p</math> matrix of basis functions.</p>

Data Types: `char` | `function_handle`

### Beta — Estimated coefficients

vector

Estimated coefficients for the explicit basis functions, stored as a vector. You can define the explicit basis function by using the `BasisFunction` name-value pair argument in `fitrgp`.

Data Types: `double`

### Sigma — Estimated noise standard deviation

scalar value

Estimated noise standard deviation of the GPR model, stored as a scalar value.

Data Types: `double`

### CategoricalPredictors — Indices of categorical predictors

vector of positive integers

Indices of categorical predictors, stored as a vector of positive integers. `CategoricalPredictors` contains index values corresponding to the columns of the

predictor data that contain categorical predictors. If none of the predictors are categorical, then this property is empty (`[]`).

Data Types: `single` | `double`

### **HyperparameterOptimizationResults** — Description of the cross-validation optimization of hyperparameters

BayesianOptimization object | table of hyperparameters and associated values

Description of the cross-validation optimization of hyperparameters, stored as a BayesianOptimization object or a table of hyperparameters and associated values. Nonempty when the OptimizeHyperparameters name-value pair is nonempty at creation. Value depends on the setting of the HyperparameterOptimizationOptions name-value pair at creation:

- `'bayesopt'` (default) — Object of class BayesianOptimization
- `'gridsearch'` or `'randomsearch'` — Table of hyperparameters used, observed objective function values (cross-validation loss), and rank of observations from lowest (best) to highest (worst)

Data Types: `table`

### **LogLikelihood** — Maximized marginal log likelihood

scalar value | `[]`

Maximized marginal log likelihood of the GPR model, stored as a scalar value if the FitMethod is different from `'none'`. If FitMethod is `'none'`, then LogLikelihood is empty.

If FitMethod is `'sd'`, `'sr'`, or `'fic'`, then LogLikelihood is the maximized approximation of the marginal log likelihood of the GPR model.

Data Types: `double`

### **ModelParameters** — Parameters used for training

GPParams object

Parameters used for training the GPR model, stored as a GPParams object.

**Kernel Function****KernelFunction** — Form of the covariance function

'squaredExponential' | 'matern32' | 'matern52' | 'ardsquaredexponential' | 'ardmatern32' | 'ardmatern52' | function handle

Form of the covariance function used in the GPR model, stored as a character vector containing the name of the built-in covariance function or a function handle. It can be one of the following.

Function	Description
'squaredexponential'	Squared exponential kernel.
'matern32'	Matern kernel with parameter 3/2.
'matern52'	Matern kernel with parameter 5/2.
'ardsquaredexponential'	Squared exponential kernel with a separate length scale per predictor.
'ardmatern32'	Matern kernel with parameter 3/2 and a separate length scale per predictor.
'ardmatern52'	Matern kernel with parameter 5/2 and a separate length scale per predictor.
Function handle	A function handle that <code>fitrgp</code> can call like this: $K_{mn} = \text{kfcn}(X_m, X_n, \theta)$ where $X_m$ is an $m$ -by- $d$ matrix, $X_n$ is an $n$ -by- $d$ matrix and $K_{mn}$ is an $m$ -by- $n$ matrix of kernel products such that $K_{mn}(i,j)$ is the kernel product between $X_m(i,:)$ and $X_n(j,:)$ . $\theta$ is the $r$ -by-1 unconstrained parameter vector for <code>kfcn</code> .

Data Types: char | function\_handle

**KernelInformation** — Information about the parameters of the kernel function

structure

Information about the parameters of the kernel function used in the GPR model, stored as a structure with the following fields.

Field Name	Description
Name	Name of the kernel function
KernelParameters	Vector of the estimated kernel parameters
KernelParameterNames	Names associated with the elements of KernelParameters.

Data Types: struct

### Prediction

#### **predictMethod** — Method used to make predictions

'exact' | 'bcd' | 'sd' | 'sr' | 'fic'

Method that `predict` uses to make predictions from the GPR model, stored as a character vector. It can be one of the following.

PredictMethod	Description
'exact'	Exact Gaussian process regression
'bcd'	Block Coordinate Descent
'sd'	Subset of Data points approximation
'sr'	Subset of Regressors approximation
'fic'	Fully Independent Conditional approximation

#### **Alpha** — Weights

numeric vector

Weights used to make predictions from the trained GPR model, stored as a numeric vector. `predict` computes the predictions for a new predictor matrix  $X_{new}$  by using the product

$$\mathbf{K}(\mathbf{X}_{new}, \mathbf{A}) * \alpha.$$

$\mathbf{K}(\mathbf{X}_{new}, \mathbf{A})$  is the matrix of kernel products between  $\mathbf{X}_{new}$  and active set vector  $\mathbf{A}$  and  $\alpha$  is a vector of weights.

Data Types: double

#### **BCDInformation** — Information on BCD-based computation of Alpha

structure | []

Information on block coordinate descent (BCD)-based computation of Alpha when PredictMethod is 'bcd', stored as a structure containing the following fields.

Field Name	Description
Gradient	$n$ -by-1 vector containing the gradient of the BCD objective function at convergence.
Objective	Scalar containing the BCD objective function at convergence.
SelectionCounts	$n$ -by-1 integer vector indicating the number of times each point was selected into a block during BCD.

Alpha property contains the Alpha vector computed from BCD.

If PredictMethod is not 'bcd', then BCDInformation is empty.

Data Types: struct

#### **ResponseTransform** — Transformation applied to predicted response

character vector

Transformation applied to the predicted response, stored as a character vector describing how the response values the model predicts are transformed. In RegressionGP, ResponseTransform is 'none' and RegressionGP does not use ResponseTransform when making predictions.

Data Types: char | function\_handle

#### **Active Set Selection**

##### **ActiveSetVectors** — Subset of training data

matrix

Subset of training data used to make predictions from the GPR model, stored as a matrix.

predict computes the predictions for a new predictor matrix  $X_{new}$  by using the product

$$\mathbf{K}(\mathbf{X}_{new}, \mathbf{A}) * \alpha.$$

$\mathbf{K}(\mathbf{X}_{new}, \mathbf{A})$  is the matrix of kernel products between  $\mathbf{X}_{new}$  and active set vector  $\mathbf{A}$  and  $\alpha$  is a vector of weights.



`ActiveSetVectors` is equal to the training data  $X$  for exact GPR fitting and a subset of the training data  $X$  for sparse GPR methods. When there are categorical predictors in the model, `ActiveSetVectors` contains dummy variables for the corresponding predictors.

Data Types: `double`

### **ActiveSetHistory** — History of active set selection and parameter estimation

structure

History of interleaved active set selection and parameter estimation on page 32-5519 for `FitMethod` equal to `'sd'`, `'sr'`, or `'fic'`, stored as a structure with the following fields.

Field Name	Description
<code>ParameterVector</code>	Cell array containing the parameter vectors: basis function coefficients, $\beta$ , kernel function parameters $\theta$ , and noise standard deviation $\sigma$ .
<code>ActiveSetIndices</code>	Cell array containing the active set indices.
<code>Loglikelihood</code>	Vector containing the maximized log likelihoods.
<code>CriterionProfile</code>	Cell array containing the active set selection criterion values as the active set grows from size 0 to its final size.

Data Types: `struct`

### **ActiveSetMethod** — Method used to select the active set

`'sgma'` | `'entropy'` | `'likelihood'` | `'random'`

Method used to select the active set for sparse methods (`'sd'`, `'sr'`, or `'fic'`), stored as a character vector. It can be one of the following.

ActiveSetMethod	Description
<code>'sgma'</code>	Sparse greedy matrix approximation
<code>'entropy'</code>	Differential entropy-based selection
<code>'likelihood'</code>	Subset of regressors log likelihood-based selection
<code>'random'</code>	Random selection

The selected active set is used in parameter estimation or prediction, depending on the choice of `FitMethod` and `PredictMethod` in `fitrgp`.

**ActiveSetSize — Size of the active set**

integer value

Size of the active set for sparse methods ('sd', 'sr', or 'fic'), stored as an integer value.

Data Types: double

**IsActiveSetVector — Indicators for selected active set**

logical vector

Indicators for selected active set for making predictions from the trained GPR model, stored as a logical vector. These indicators mark the subset of training data that `fitrgp` selects as the active set. For example, if `X` is the original training data, then `ActiveSetVectors = X(IsActiveSetVector, :)`.

Data Types: logical

**Training Data****NumObservations — Number of observations in training data**

scalar value

Number of observations in training data, stored as a scalar value.

Data Types: double

**x — Training data**

*n*-by-*d* table | *n*-by-*d* matrix

Training data, stored as an *n*-by-*d* table or matrix, where *n* is the number of observations and *d* is the number of predictor variables (columns) in the training data. If the GPR model is trained on a table, then `x` is a table. Otherwise, `x` is a matrix.

Data Types: double | table

**y — Observed response values**

*n*-by-1 vector

Observed response values used to train the GPR model, stored as an *n*-by-1 vector, where *n* is the number of observations.

Data Types: `double`

**PredictorNames** — Names of predictors

cell array of character vectors

Names of predictors used in the GPR model, stored as a cell array of character vectors. Each name (cell) corresponds to a column in  $X$ .

Data Types: `cell`

**ExpandedPredictorNames** — Names of expanded predictors

cell array of character vectors

Names of expanded predictors for the GPR model, stored as a cell array of character vectors. Each name (cell) corresponds to a column in `ActiveSetVectors`.

If the model uses dummy variables for categorical variables, then `ExpandedPredictorNames` includes the names that describe the expanded variables. Otherwise, `ExpandedPredictorNames` is the same as `PredictorNames`.

Data Types: `cell`

**ResponseName** — Name of the response variable

character vector

Name of the response variable in the GPR model, stored as a character vector.

Data Types: `char`

**PredictorLocation** — Means of predictors

1-by- $d$  vector | []

Means of predictors used for training the GPR model if the training data is standardized, stored as a 1-by- $d$  vector. If the training data is not standardized, `PredictorLocation` is empty.

If `PredictorLocation` is not empty, then the `predict` method centers the predictor values by subtracting the respective element of `PredictorLocation` from every column of  $X$ .

If there are categorical predictors, then `PredictorLocation` includes a 0 for each dummy variable corresponding to those predictors. The dummy variables are not centered or scaled.

Data Types: `double`

### **PredictorScale** — Standard deviations of predictors

1-by- $d$  vector | []

Standard deviations of predictors used for training the GPR model if the training data is standardized, stored as a 1-by- $d$  vector. If the training data is not standardized, `PredictorScale` is empty.

If `PredictorScale` is not empty, the `predict` method scales the predictors by dividing every column of `X` by the respective element of `PredictorScale` (after centering using `PredictorLocation`).

If there are categorical predictors, then `PredictorLocation` includes a 1 for each dummy variable corresponding to those predictors. The dummy variables are not centered or scaled.

Data Types: `double`

### **RowsUsed** — Indicators for rows used in training

logical vector | []

Indicators for rows used in training the GPR model, stored as a logical vector. If all rows are used in training the model, then `RowsUsed` is empty.

Data Types: `logical`

## Methods

<code>compact</code>	Create compact Gaussian process regression model
<code>crossval</code>	Cross-validate Gaussian process regression model
<code>postFitStatistics</code>	Compute post-fit statistics for the exact Gaussian process regression model
<code>resubLoss</code>	Resubstitution loss for a trained Gaussian process regression model
<code>resubPredict</code>	Resubstitution prediction from a trained Gaussian process regression model

## Inherited Methods

loss        Regression error for Gaussian process regression model  
 predict     Predict response of Gaussian process regression model

## Definitions

### Active Set Selection and Parameter Estimation

For subset of data, subset of regressors, or fully independent conditional approximation fitting methods (FitMethod equal to 'sd', 'sr', or 'fic'), if you do not provide the active set, fitrgp selects the active set and computes the parameter estimates in a series of iterations.

In the first iteration, the software uses the initial parameter values in vector  $\eta_0 = [\beta_0, \sigma_0, \theta_0]$  to select an active set  $A_1$ . It maximizes the GPR marginal log likelihood or its approximation using  $\eta_0$  as the initial values and  $A_1$  to compute the new parameter estimates  $\eta_1$ . Next, it computes the new log likelihood  $L_1$  using  $\eta_1$  and  $A_1$ .

In the second iteration, the software selects the active set  $A_2$  using the parameter values in  $\eta_1$ . Then, using  $\eta_1$  as the initial values and  $A_2$ , it maximizes the GPR marginal log likelihood or its approximation and estimates the new parameter values  $\eta_2$ . Then using  $\eta_2$  and  $A_2$ , computes the new log likelihood value  $L_2$ .

The following table summarizes the iterations and what is computed at each iteration.

Iteration Number	Active Set	Parameter Vector	Log Likelihood
1	$A_1$	$\eta_1$	$L_1$
2	$A_2$	$\eta_2$	$L_2$
3	$A_3$	$\eta_3$	$L_3$
...	...	...	...

The software iterates similarly for a specified number of repetitions. You can specify the number of replications for active set selection using the NumActiveSetRepeats name-value pair argument.

## Tips

- You can access the properties of this class using dot notation. For example, `KernelInformation` is a structure holding the kernel parameters and their names. Hence, to access the kernel function parameters of the trained model `gprMdl`, use `gprMdl.KernelInformation.KernelParameters`.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Only the `predict` function supports code generation.
- When you train a Gaussian process regression model by using `fitrgp`, the following restrictions apply.
  - Code generation does not support categorical predictors. You cannot supply training data in a table that contains at least one categorical predictor, and you cannot use the `'CategoricalPredictors'` name-value pair argument. To dummy-code variables that you want treated as categorical, preprocess the categorical data by using `dummyvar` before fitting the model.
  - The value of the `'ResponseTransform'` name-value pair argument cannot be an anonymous function.

### See Also

`CompactRegressionGP` | `compact` | `fitrgp` | `plotPartialDependence`

### Topics

Class Attributes (MATLAB)

Property Attributes (MATLAB)

**Introduced in R2015b**

## RegressionLinear class

Linear regression model for high-dimensional data

### Description

`RegressionLinear` is a trained linear model object for regression; the linear model is a support vector machine regression (SVM) or linear regression model. `fitrlinear` fits a `RegressionLinear` model by minimizing the objective function using techniques that reduce computation time for high-dimensional data sets (e.g., stochastic gradient descent). The regression loss plus the regularization term compose the objective function.

Unlike other regression models, and for economical memory usage, `RegressionLinear` model objects do not store the training data. However, they do store, for example, the estimated linear model coefficients, estimated coefficients, and the regularization strength.

You can use trained `RegressionLinear` models to predict responses for new data. For details, see `predict`.

### Construction

`Mdl = fitrlinear(X,Y)` returns a trained linear regression model object that contains the results of fitting a support vector machine regression model to the predictor data `X` and response data `Y`.

`Mdl = fitrlinear(___,Name,Value)` returns a trained linear regression model with additional options specified by one or more `Name,Value` pair arguments, using any of the previous syntaxes. For example, you can specify to implement logistic regression, specify to cross-validate, or specify the type of regularization.

If you specify one of the following four options, then `Mdl` is a `RegressionPartitionedLinear` model: `'CrossVal'`, `'CVPartition'`, `'Holdout'`, or `'KFold'`.



## Input Arguments

### **x** — Predictor data

full matrix | sparse matrix

Predictor data, specified as an  $n$ -by- $p$  full or sparse matrix.

The length of  $Y$  and the number of observations in  $X$  must be equal.

---

**Note** If you orient your predictor matrix so that observations correspond to columns and specify 'ObservationsIn', 'columns', then you might experience a significant reduction in optimization-execution time.

---

Data Types: `single` | `double`

### **y** — Response data

numeric vector

Response data, specified as an  $n$ -dimensional numeric vector. The length of  $Y$  and the number of observations in  $X$  must be equal.

Data Types: `single` | `double`

For more details, see `fitrlinear`.

## Properties

### Linear Regression Properties

#### **Epsilon** — Half of width of epsilon-insensitive band

nonnegative scalar

Half of the width of the epsilon-insensitive-band, specified as a nonnegative scalar.

If `Learner` is not 'svm', then `Epsilon` is an empty array (`[]`).

Data Types: `single` | `double`

#### **Lambda** — Regularization term strength

nonnegative scalar | vector of nonnegative values

Regularization term strength, specified as a nonnegative scalar or vector of nonnegative values.

Data Types: `double` | `single`

### Learner — Linear regression model type

`'leastsquares'` | `'svm'`

Linear regression model type, specified as `'leastsquares'` or `'svm'`.

In this table,  $f(x) = x\beta + b$ .

- $\beta$  is a vector of  $p$  coefficients.
- $x$  is an observation from  $p$  predictor variables.
- $b$  is the scalar bias.

Value	Algorithm	Loss function	FittedLoss Value
<code>'leastsquares'</code>	Linear regression via ordinary least squares	Mean squared error (MSE): $\ell[y, f(x)] = \frac{1}{2}[y - f(x)]^2$	<code>'mse'</code>
<code>'svm'</code>	Support vector machine regression	Epsilon-insensitive: $\ell[y, f(x)] = \max[0,  y - f(x)  - \varepsilon]$	<code>'epsiloninsensitive'</code>

Data Types: `char`

### Beta — Linear coefficient estimates

numeric vector

Linear coefficient estimates, specified as a numeric vector with length equal to the number of predictors.

Data Types: `double`

### Bias — Estimated bias term

numeric scalar

Estimated bias term or model intercept, specified as a numeric scalar.

Data Types: double

### FittedLoss — Loss function used to fit the linear model

'epsiloninsensitive' | 'mse'

Loss function used to fit the model, specified as 'epsiloninsensitive' or 'mse'.

Value	Algorithm	Loss function	Learner Value
'epsiloninsensitive'	Support vector machine regression	Epsilon-insensitive: $\ell[y, f(x)] = \max[0,  y - f(x)  - \varepsilon]$	'svm'
'mse'	Linear regression via ordinary least squares	Mean squared error (MSE): $\ell[y, f(x)] = \frac{1}{2}[y - f(x)]^2$	'leastquares'

### Regularization — Complexity penalty type

'lasso (L1)' | 'ridge (L2)'

Complexity penalty type, specified as 'lasso (L1)' or 'ridge (L2)'.

The software composes the objective function for minimization from the sum of the average loss function (see FittedLoss) and the regularization term in this table.

Value	Description
'lasso (L1)'	Lasso ( $L_1$ ) penalty: $\lambda \sum_{j=1}^p  \beta_j $
'ridge (L2)'	Ridge ( $L_2$ ) penalty: $\frac{\lambda}{2} \sum_{j=1}^p \beta_j^2$

$\lambda$  specifies the regularization term strength (see Lambda).

The software excludes the bias term ( $\beta_0$ ) from the regularization penalty.

### Other Regression Properties

#### ModelParameters — Parameters used for training model

structure

Parameters used for training the `RegressionLinear` model, specified as a structure.

Access fields of `ModelParameters` using dot notation. For example, access the relative tolerance on linear coefficients and bias term using `Mdl.ModelParameters.BetaTolerance`.

Data Types: `struct`

**PredictorNames — Predictor names**

cell array of character vectors

Predictor names in their order of appearance in predictor data (X), specified as a cell array of character vectors. `PredictorNames` has length equal to the number of columns in X.

Data Types: `cell`

**ExpandedPredictorNames — Expanded predictor names**

cell array of character vectors

Expanded predictor names, stored as a cell array of character vectors.

Because a `RegressionLinear` model does not support categorical predictors, `ExpandedPredictorNames` and `PredictorNames` are equal.

Data Types: `cell`

**ResponseName — Response variable name**

character array

Response variable name, specified as a character array.

Data Types: `char`

**ResponseTransform — Response transformation function**

'none' | function handle

Response transformation function, stored as 'none' or a function handle. `ResponseTransform` describes how the software transforms raw response values.

For a MATLAB function, or a function that you define, enter its function handle. For example, you can enter `Mdl.ResponseTransform = @function`, where *function*

accepts a numeric vector of the original responses and returns a numeric vector of the same size containing the transformed responses.

Data Types: char | function\_handle

## Methods

loss	Regression loss for linear regression models
predict	Predict response of linear regression model
selectModels	Select fitted regularized linear regression models

## Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects (MATLAB).

## Examples

### Train Linear Regression Model

Train a linear regression model using SVM, dual SGD, and ridge regularization.

Simulate 10000 observations from this model

$$y = x_{100} + 2x_{200} + e.$$

- $X = x_1, \dots, x_{10000}$  is a 10000-by-1000 sparse matrix with 10% nonzero standard normal elements.
- $e$  is random normal error with mean 0 and standard deviation 0.3.

```
rng(1) % For reproducibility
n = 1e4;
d = 1e3;
nz = 0.1;
X = sprandn(n,d,nz);
Y = X(:,100) + 2*X(:,200) + 0.3*randn(n,1);
```

Train a linear regression model. By default, `fitrlinear` uses support vector machines with a ridge penalty, and optimizes using dual SGD for SVM. Determine how well the optimization algorithm fit the model to the data by extracting a fit summary.

```
[Mdl,FitInfo] = fitrlinear(X,Y)
```

```
Mdl =
```

```
RegressionLinear
  ResponseName: 'Y'
  ResponseTransform: 'none'
      Beta: [1000x1 double]
      Bias: -0.0056
      Lambda: 1.0000e-04
  Learner: 'svm'
```

```
FitInfo =
```

```
struct with fields:

    Lambda: 1.0000e-04
    Objective: 0.2726
    PassLimit: 10
    NumPasses: 10
    BatchLimit: []
    NumIterations: 100000
    GradientNorm: NaN
    GradientTolerance: 0
    RelativeChangeInBeta: 0.4908
    BetaTolerance: 1.0000e-04
    DeltaGradient: 1.5818
    DeltaGradientTolerance: 0.1000
    TerminationCode: 0
    TerminationStatus: {'Iteration limit exceeded.'}
    Alpha: [10000x1 double]
    History: []
    FitTime: 0.2463
    Solver: {'dual'}
```

Mdl is a RegressionLinear model. You can pass Mdl and the training or new data to loss to inspect the in-sample mean-squared error. Or, you can pass Mdl and new predictor data to predict to predict responses for new observations.

FitInfo is a structure array containing, among other things, the termination status (TerminationStatus) and how long the solver took to fit the model to the data (FitTime). It is good practice to use FitInfo to determine whether optimization-termination measurements are satisfactory. In this case, fitrlinear reached the maximum number of iterations. Because training time is fast, you can retrain the model, but increase the number of passes through the data. Or, try another solver, such as LBFGS.

### Predict Responses Using Linear Regression Model

Simulate 10000 observations from this model

$$y = x_{100} + 2x_{200} + e.$$

- $X = \{x_1, \dots, x_{1000}\}$  is a 10000-by-1000 sparse matrix with 10% nonzero standard normal elements.
- $e$  is random normal error with mean 0 and standard deviation 0.3.

```
rng(1) % For reproducibility
n = 1e4;
d = 1e3;
nz = 0.1;
X = sprandn(n,d,nz);
Y = X(:,100) + 2*X(:,200) + 0.3*randn(n,1);
```

Hold out 5% of the data.

```
rng(1); % For reproducibility
cvp = cvpartition(n, 'Holdout', 0.05)
```

```
cvp =
```

```
Hold-out cross validation partition
  NumObservations: 10000
    NumTestSets: 1
      TrainSize: 9500
      TestSize: 500
```

`cvp` is a `CVPPartition` object that defines the random partition of  $n$  data into training and test sets.

Train a linear regression model using the training set. For faster training time, orient the predictor data matrix so that the observations are in columns.

```
idxTrain = training(cvp); % Extract training set indices
X = X';
Mdl = fitrlinear(X(:,idxTrain),Y(idxTrain),'ObservationsIn','columns');
```

Predict observations and the mean squared error (MSE) for the hold out sample.

```
idxTest = test(cvp); % Extract test set indices
yHat = predict(Mdl,X(:,idxTest),'ObservationsIn','columns');
L = loss(Mdl,X(:,idxTest),Y(idxTest),'ObservationsIn','columns')
```

```
L =
```

```
    0.1852
```

The hold-out sample MSE is 0.1852.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Only the `predict` function supports code generation.
- When you train a linear regression model by using `fitrlinear`, the following restrictions apply.
  - The predictor data input argument value ( $X$ ) must be a full numeric matrix.
  - You can specify only one regularization strength, either `'auto'` or a nonnegative scalar for the `'Lambda'` name-value pair argument.



- The value of the 'ResponseTransform' name-value pair argument cannot be an anonymous function.

## See Also

`RegressionPartitionedLinear` | `fitrlinear` | `plotPartialDependence` | `predict`

**Introduced in R2016a**

## RegressionPartitionedEnsemble class

### Superclasses:

Cross-validated regression ensemble

### Description

`RegressionPartitionedEnsemble` is a set of regression ensembles trained on cross-validated folds. Estimate the quality of classification by cross validation using one or more “kfold” methods: `kfoldfun`, `kfoldLoss`, or `kfoldPredict`. Every “kfold” method uses models trained on in-fold observations to predict response for out-of-fold observations. For example, suppose you cross validate using five folds. In this case, every training fold contains roughly 4/5 of the data and every test fold contains roughly 1/5 of the data. The first model stored in `Trained{1}` was trained on X and Y with the first 1/5 excluded, the second model stored in `Trained{2}` was trained on X and Y with the second 1/5 excluded, and so on. When you call `kfoldPredict`, it computes predictions for the first 1/5 of the data using the first model, for the second 1/5 of data using the second model and so on. In short, response for every observation is computed by `kfoldPredict` using the model trained without this observation.

### Construction

`cvens = crossval(ens)` creates a cross-validated ensemble from `ens`, a regression ensemble. For syntax details, see the `crossval` method reference page.

`cvens = fitrensemble(X, Y, Name, Value)` creates a cross-validated ensemble when `Name` is one of 'crossval', 'kfold', 'holdout', 'leaveout', or 'cvpartition'. For syntax details, see the `fitrensemble` function reference page.

### Input Arguments

#### **ens**

A regression ensemble constructed with `fitrensemble`.

## Properties

### **CategoricalPredictors**

Indices of categorical predictors, stored as a vector of positive integers. `CategoricalPredictors` contains index values corresponding to the columns of the predictor data that contain categorical predictors. If none of the predictors are categorical, then this property is empty (`[]`).

### **CrossValidatedModel**

Name of the cross-validated model, a character vector.

### **Kfold**

Number of folds used in a cross-validated tree, a positive integer.

### **ModelParameters**

Object holding parameters of `tree`.

### **NumObservations**

Numeric scalar containing the number of observations in the training data.

### **NTrainedPerFold**

Vector of `Kfold` elements. Each entry contains the number of trained learners in this cross-validation fold.

### **Partition**

The partition of class `cvpartition` used in creating the cross-validated ensemble.

### **PredictorNames**

A cell array of names for the predictor variables, in the order in which they appear in `X`.

### **ResponseName**

Name of the response variable `Y`, a character vector.

**ResponseTransform**

Function handle for transforming scores, or character vector representing a built-in transformation function. 'none' means no transformation; equivalently, 'none' means  $@(x) x$ .

Add or change a ResponseTransform function using dot notation:

```
ens.ResponseTransform = @function
```

**Trainable**

Cell array of ensembles trained on cross-validation folds. Every ensemble is full, meaning it contains its training data and weights.

**Trained**

Cell array of compact ensembles trained on cross-validation folds.

**w**

The scaled weights, a vector with length  $n$ , the number of rows in  $X$ .

**X**

A matrix of predictor values. Each column of  $X$  represents one variable, and each row represents one observation.

**Y**

A numeric column vector with the same number of rows as  $X$ . Each entry in  $Y$  is the response to the data in the corresponding row of  $X$ .

## Methods

kfoldLoss	Cross-validation loss of partitioned regression ensemble
resume	Resume training ensemble

## Inherited Methods

kfoldfun	Cross validate function
kfoldLoss	Cross-validation loss of partitioned regression model
kfoldPredict	Predict response for observations not used for training.

## Copy Semantics

Value. To learn how value classes affect copy operations, see [Copying Objects \(MATLAB\)](#).

## Examples

### Construct Partitioned Regression Ensemble

Construct a partitioned regression ensemble, and examine the cross-validation losses for the folds.

Load the `carsmall` data set.

```
load carsmall;
```

Create a subset of variables.

```
XX = [Cylinders Displacement Horsepower Weight];  
YY = MPG;
```

Construct the ensemble model.

```
rens = fitrensemble(XX,YY);
```

Create a cross-validated ensemble from `rens`.

```
rng(10,'twister') % For reproducibility  
cvrens = crossval(rens);
```

Examine the cross-validation losses.

```
L = kfoldLoss(cvrens,'mode','individual')
```

`L` =

```
21.4489
48.4388
28.2223
17.5354
29.9441
49.5254
43.8872
31.1757
31.6388
8.9607
```

`L` is a vector containing the cross-validation loss for each trained learner in the ensemble.

## See Also

`ClassificationPartitionedEnsemble` | `RegressionEnsemble` |  
`RegressionPartitionedModel` | `fitrtree`

# RegressionPartitionedLinear class

Cross-validated linear regression model for high-dimensional data

## Description

`RegressionPartitionedLinear` is a set of linear regression models trained on cross-validated folds. To obtain a cross-validated, linear regression model, use `fitrlinear` and specify one of the cross-validation options. You can estimate the predictive quality of the model, or how well the linear regression model generalizes, using one or more of these “kfold” methods: `kfoldPredict` and `kfoldLoss`.

Every “kfold” method uses models trained on in-fold observations to predict the response for out-of-fold observations. For example, suppose that you cross-validate using five folds. In this case, the software randomly assigns each observation into five roughly equally sized groups. The training fold contains four of the groups (that is, roughly 4/5 of the data) and the test fold contains the other group (that is, roughly 1/5 of the data). In this case, cross-validation proceeds as follows:

- 1 The software trains the first model (stored in `CVMdl.Trained{1}`) using the observations in the last four groups and reserves the observations in the first group for validation.
- 2 The software trains the second model (stored in `CVMdl.Trained{2}`) using the observations in the first group and last three groups. The software reserves the observations in the second group for validation.
- 3 The software proceeds in a similar fashion for the third through fifth models.

If you validate by calling `kfoldPredict`, it computes predictions for the observations in group 1 using the first model, group 2 for the second model, and so on. In short, the software estimates a response for every observation using the model trained without that observation.

---

**Note** Unlike other cross-validated, regression models, `RegressionPartitionedLinear` model objects do not store the predictor data set.

---

## Construction

`CVMdl = fitrlinear(X,Y,Name,Value)` creates a cross-validated, linear regression model when `Name` is either `'CrossVal'`, `'CVPartition'`, `'Holdout'`, or `'KFold'`. For more details, see `fitrlinear`.

## Properties

### Cross-Validation Properties

#### **CrossValidatedModel** — Cross-validated model name

character array

Cross-validated model name, specified as a character array.

For example, `'Linear'` specifies a cross-validated linear model for binary classification or regression.

Data Types: `char`

#### **KFold** — Number of cross-validated folds

positive integer

Number of cross-validated folds, specified as a positive integer.

Data Types: `double`

#### **ModelParameters** — Cross-validation parameter values

object

Cross-validation parameter values, e.g., the name-value pair argument values used to cross-validate the linear model, specified as an object. `ModelParameters` does not contain estimated parameters.

Access properties of `ModelParameters` using dot notation.

#### **NumObservations** — Number of observations

positive numeric scalar

Number of observations in the training data, specified as a positive numeric scalar.



Data Types: `double`

**Partition — Data partition**

`cvpartition` model

Data partition indicating how the software splits the data into cross-validation folds, specified as a `cvpartition` model.

**Trained — Linear regression models trained on cross-validation folds**

cell array of `RegressionLinear` model objects

Linear regression models trained on cross-validation folds, specified as a cell array of `RegressionLinear` models. `Trained` has  $k$  cells, where  $k$  is the number of folds.

Data Types: `cell`

**w — Observation weights**

numeric vector

Observation weights used to cross-validate the model, specified as a numeric vector. `w` has `NumObservations` elements.

The software normalizes the weights used for training so that `nansum(w)` is 1.

Data Types: `single` | `double`

**Y — Observed responses**

numeric vector

Observed responses used to cross-validate the model, specified as a numeric vector containing `NumObservations` elements.

Each row of `Y` represents the observed response of the corresponding observation in the predictor data.

Data Types: `single` | `double`

**Other Regression Properties**

**CategoricalPredictors — Indices of categorical predictors**

vector of positive integers

Indices of categorical predictors, stored as a vector of positive integers. `CategoricalPredictors` contains index values corresponding to the columns of the

predictor data that contain categorical predictors. If none of the predictors are categorical, then this property is empty (`[]`).

Data Types: `single` | `double`

#### **PredictorNames** — Predictor names

cell array of character vectors

Predictor names in their order of appearance in predictor data used to train `CVMdl`, specified as a cell array of character vectors. `PredictorNames` has length equal to the number of predictor variables in the predictor data.

Data Types: `cell`

#### **ResponseName** — Response variable name

character array

Response variable name, specified as a character array.

Data Types: `char`

#### **ResponseTransform** — Response transformation function

'none' | function handle

Response transformation function, stored as 'none' or a function handle. `ResponseTransform` describes how the software transforms raw response values.

For a MATLAB function, or a function that you define, enter its function handle. For example, you can enter `Mdl.ResponseTransform = @function`, where `function` accepts a numeric vector of the original responses and returns a numeric vector of the same size containing the transformed responses.

Data Types: `char` | `function_handle`

## Methods

<code>kfoldLoss</code>	Regression loss for observations not used in training
<code>kfoldPredict</code>	Predict responses for observations not used for training

## Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects (MATLAB).

## Examples

### Create Cross-Validated Linear Regression Model

Simulate 10000 observations from this model

$$y = x_{100} + 2x_{200} + e.$$

- $X = \{x_1, \dots, x_{1000}\}$  is a 10000-by-1000 sparse matrix with 10% nonzero standard normal elements.
- $e$  is random normal error with mean 0 and standard deviation 0.3.

```
rng(1) % For reproducibility
n = 1e4;
d = 1e3;
nz = 0.1;
X = sprandn(n,d,nz);
Y = X(:,100) + 2*X(:,200) + 0.3*randn(n,1);
```

Cross-validate a linear regression model. To increase execution speed, transpose the predictor data and specify that the observations are in columns.

```
X = X';
CVMdl = fitrlinear(X,Y,'CrossVal','on','ObservationsIn','columns');
```

CVMdl is a RegressionPartitionedLinear cross-validated model. Because fitrlinear implements 10-fold cross-validation by default, CVMdl.Trained contains a cell vector of ten RegressionLinear models. Each cell contains a linear regression model trained on nine folds, and then tested on the remaining fold.

Predict responses for out-of-fold observations and estimate the generalization error by passing CVMdl to kfoldPredict and kfoldLoss, respectively.

```
oofYHat = kfoldPredict(CVMdl);
ge = kfoldLoss(CVMdl)
```

```
ge =  
    0.1748
```

The estimated, generalization, mean squared error is 0.1748.

### Find Good Lasso Penalty Using Cross-Validation

To determine a good lasso-penalty strength for a linear regression model that uses least squares, implement 5-fold cross-validation.

Simulate 10000 observations from this model

$$y = x_{100} + 2x_{200} + e.$$

- $X = \{x_1, \dots, x_{1000}\}$  is a 10000-by-1000 sparse matrix with 10% nonzero standard normal elements.
- $e$  is random normal error with mean 0 and standard deviation 0.3.

```
rng(1) % For reproducibility  
n = 1e4;  
d = 1e3;  
nz = 0.1;  
X = sprandn(n,d,nz);  
Y = X(:,100) + 2*X(:,200) + 0.3*randn(n,1);
```

Create a set of 15 logarithmically-spaced regularization strengths from  $10^{-5}$  through  $10^{-1}$ .

```
Lambda = logspace(-5,-1,15);
```

Cross-validate the models. To increase execution speed, transpose the predictor data and specify that the observations are in columns. Optimize the objective function using SpARSA.

```
X = X';  
CVMdl = fitrlinear(X,Y,'ObservationsIn','columns','KFold',5,'Lambda',Lambda,...  
    'Learner','leastsquares','Solver','sparsa','Regularization','lasso');  
  
numCLModels = numel(CVMdl.Trained)
```

```
numCLModels =
```

```
5
```

CVMdl is a RegressionPartitionedLinear model. Because fitrlinear implements 5-fold cross-validation, CVMdl contains 5 RegressionLinear models that the software trains on each fold.

Display the first trained linear regression model.

```
Mdl1 = CVMdl.Trained{1}
```

```
Mdl1 =
```

```
RegressionLinear
  ResponseName: 'Y'
  ResponseTransform: 'none'
      Beta: [1000x15 double]
      Bias: [1x15 double]
      Lambda: [1x15 double]
  Learner: 'leastsquares'
```

Mdl1 is a RegressionLinear model object. fitrlinear constructed Mdl1 by training on the first four folds. Because Lambda is a sequence of regularization strengths, you can think of Mdl1 as 11 models, one for each regularization strength in Lambda.

Estimate the cross-validated MSE.

```
mse = kfoldLoss(CVMdl);
```

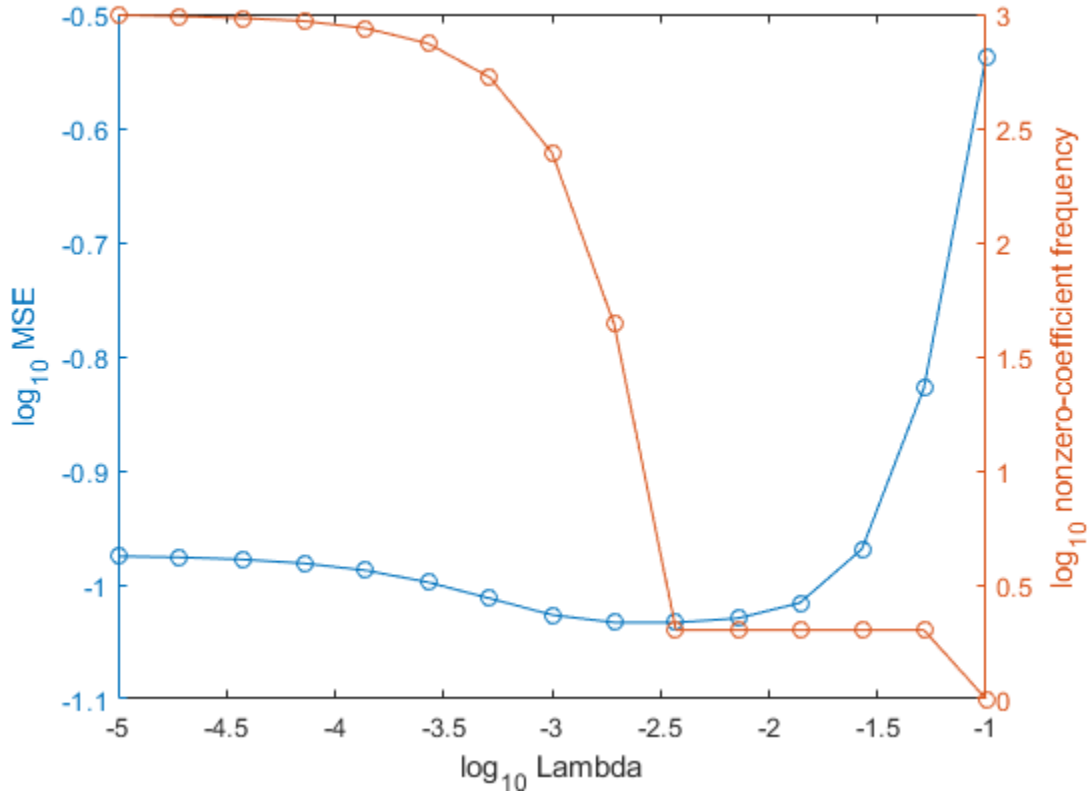
Higher values of Lambda lead to predictor variable sparsity, which is a good quality of a regression model. For each regularization strength, train a linear regression model using the entire data set and the same options as when you cross-validated the models.

Determine the number of nonzero coefficients per model.

```
Mdl = fitrlinear(X,Y,'ObservationsIn','columns','Lambda',Lambda,...
  'Learner','leastsquares','Solver','sparsa','Regularization','lasso');
numNZCoeff = sum(Mdl.Beta~=0);
```

In the same figure, plot the cross-validated MSE and frequency of nonzero coefficients for each regularization strength. Plot all variables on the log scale.

```
figure;
[h,hL1,hL2] = plotyy(log10(Lambda),log10(mse),...
    log10(Lambda),log10(numNZCoeff));
hL1.Marker = 'o';
hL2.Marker = 'o';
ylabel(h(1),'log_{10} MSE')
ylabel(h(2),'log_{10} nonzero-coefficient frequency')
xlabel('log_{10} Lambda')
hold off
```



Choose the index of the regularization strength that balances predictor variable sparsity and low MSE (for example, `Lambda(10)`).

```
idxFinal = 10;
```

Extract the model with corresponding to the minimal MSE.

```
MdlFinal = selectModels(Mdl,idxFinal)
idxNZCoeff = find(MdlFinal.Beta~=0)
EstCoeff = Mdl.Beta(idxNZCoeff)
```

```
MdlFinal =
```

```
RegressionLinear
  ResponseName: 'Y'
  ResponseTransform: 'none'
  Beta: [1000x1 double]
  Bias: -0.0050
  Lambda: 0.0037
  Learner: 'leastsquares'
```

```
idxNZCoeff =
```

```
100
200
```

```
EstCoeff =
```

```
1.0051
1.9965
```

`MdlFinal` is a `RegressionLinear` model with one regularization strength. The nonzero coefficients `EstCoeff` are close to the coefficients that simulated the data.

## See Also

[RegressionLinear](#) | [fitrlinear](#) | [kfoldLoss](#) | [kfoldPredict](#)

**Introduced in R2016a**



# RegressionPartitionedModel class

Cross-validated regression model

## Description

`RegressionPartitionedModel` is a set of regression models trained on cross-validated folds. Estimate the quality of regression by cross validation using one or more “kfold” methods: `kfoldPredict`, `kfoldLoss`, and `kfoldfun`. Every “kfold” method uses models trained on in-fold observations to predict response for out-of-fold observations. For example, suppose you cross validate using five folds. In this case, every training fold contains roughly 4/5 of the data and every test fold contains roughly 1/5 of the data. The first model stored in `Trained{1}` was trained on `X` and `Y` with the first 1/5 excluded, the second model stored in `Trained{2}` was trained on `X` and `Y` with the second 1/5 excluded, and so on. When you call `kfoldPredict`, it computes predictions for the first 1/5 of the data using the first model, for the second 1/5 of data using the second model and so on. In short, response for every observation is computed by `kfoldPredict` using the model trained without this observation.

## Construction

`cvmodel = crossval(tree)` creates a cross-validated classification model from a regression tree. For syntax details, see the `crossval` method reference page.

`cvmodel = fitrtree(X,Y,Name,Value)` creates a cross-validated model when `name` is one of 'CrossVal', 'KFold', 'Holdout', 'Leaveout', or 'CVPartition'. For syntax details, see the `fitrtree` function reference page.

## Input Arguments

### **tree**

A regression tree constructed with `fitrtree`.

## Properties

### **CategoricalPredictors**

Indices of categorical predictors, stored as a vector of positive integers. `CategoricalPredictors` contains index values corresponding to the columns of the predictor data that contain categorical predictors. If none of the predictors are categorical, then this property is empty (`[]`).

### **CrossValidatedModel**

Name of the cross-validated model, a character vector.

### **Kfold**

Number of folds used in a cross-validated tree, a positive integer.

### **ModelParameters**

Object holding parameters of `tree`.

### **Partition**

The partition of class `cvpartition` used in the cross-validated model.

### **PredictorNames**

A cell array of names for the predictor variables, in the order in which they appear in `X`.

### **ResponseName**

Name of the response variable `Y`, a character vector.

### **ResponseTransform**

Function handle for transforming the raw response values (mean squared error). The function handle should accept a matrix of response values and return a matrix of the same size. The default character vector `'none'` means  $@(x) x$ , or no transformation.

Add or change a `ResponseTransform` function using dot notation:

```
ctree.ResponseTransform = @function
```

**Trained**

The trained learners, a cell array of compact regression models.

**W**

The scaled `weights`, a vector with length `n`, the number of rows in `X`.

**X**

A matrix of predictor values. Each column of `X` represents one variable, and each row represents one observation.

**Y**

A numeric column vector with the same number of rows as `X`. Each entry in `Y` is the response to the data in the corresponding row of `X`.

## Methods

<code>kfoldfun</code>	Cross validate function
<code>kfoldLoss</code>	Cross-validation loss of partitioned regression model
<code>kfoldPredict</code>	Predict response for observations not used for training.

## Copy Semantics

Value. To learn how value classes affect copy operations, see [Copying Objects \(MATLAB\)](#).

## Examples

### Evaluate Cross-Validation Error

Load the sample data. Create a variable `X` containing the `Horsepower` and `Weight` data.

```
load carsmall
X = [Horsepower Weight];
```

Construct a regression tree using the sample data.

```
cvtree = fitrtree(X,MPG,'crossval','on');
```

Evaluate the cross-validation error of the `carsmall` data using Horsepower and Weight as predictor variables for mileage (MPG).

```
L = kfoldLoss(cvtree)
```

```
L =
```

```
    25.5338
```

## See Also

[ClassificationPartitionedModel](#) | [RegressionPartitionedEnsemble](#)

# RegressionPartitionedSVM class

## Superclasses:

Cross-validated support vector machine regression model

## Description

`RegressionPartitionedSVM` is a set of support vector machine (SVM) regression models trained on cross-validated folds.

## Construction

`CVMD1 = crossval mdl` returns a cross-validated (partitioned) support vector machine regression model, `CVMD1`, from a trained SVM regression model, `mdl`.

`CVMD1 = crossval mdl, Name, Value` returns a cross-validated model with additional options specified by one or more `Name, Value` pair arguments. `Name` can also be a property name on page 32-5551 and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name-value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

## Input Arguments

**mdl** — Full, trained SVM regression model

`RegressionSVM` model

Full, trained SVM regression model, specified as a `RegressionSVM` model returned by `fitrsvm`.

## Properties

**CategoricalPredictors** — Indices of categorical predictors

vector of positive integers

Indices of categorical predictors, stored as a vector of positive integers. `CategoricalPredictors` contains index values corresponding to the columns of the predictor data that contain categorical predictors. If none of the predictors are categorical, then this property is empty (`[]`).

Data Types: `single` | `double`

**CrossValidatedModel** — Name of the cross-validated model

character vector

Name of the cross-validated model, stored as a character vector.

Data Types: `char`

**KFold** — Number of cross-validation folds

positive integer value

Number of cross-validation folds, stored as a positive integer value.

Data Types: `single` | `double`

**ModelParameters** — Cross-validation parameters

object

Cross-validation parameters, stored as an object.

**NumObservations** — Number of observations

positive integer value

Number of observations in the training data, stored as a positive integer value.

Data Types: `single` | `double`

**Partition** — Data partition for cross-validation

`cvpartition` object

Data partition for cross-validation, stored as a `cvpartition` object.

**PredictorNames** — Predictor names

cell array of character vectors

Predictor names, stored as a cell array of character vectors containing the name of each predictor in the order in which they appear in `X`. `PredictorNames` has a length equal to the number of columns in `X`.

Data Types: `cell`

**ResponseName** — Response variable name

character vector

Response variable name, stored as a character vector.

Data Types: `char`

**ResponseTransform** — Response transformation function

'none' | function handle

Response transformation function, stored as 'none' or a function handle. ResponseTransform describes how the software transforms raw response values.

For a MATLAB function, or a function that you define, enter its function handle. For example, you can enter `Mdl.ResponseTransform = @function`, where *function* accepts a numeric vector of the original responses and returns a numeric vector of the same size containing the transformed responses.

Data Types: `char` | `function_handle`

**Trained** — Trained, compact regression models

cell array of CompactRegressionSVM models

Trained, compact regression models, stored as a cell array of CompactRegressionSVM models.

Data Types: `cell`

**w** — Observation weights

vector of numeric values

Observation weights used to train the model, stored as a numeric vector containing NumObservation number of elements. `fitrsvm` normalizes the weights used for training so that they sum to 1.

Data Types: `single` | `double`

**x** — Predictor values

matrix of numeric values | table of numeric values

Predictor values used to train the model, stored as a matrix of numeric values if the model is trained on a matrix, or a table if the model is trained on a table. X has size *n*-by-

$p$ , where  $n$  is the number of rows and  $p$  is the number of predictor variables or columns in the training data.

Data Types: `single` | `double` | `table`

### **$y$ — Observed responses**

numeric vector

Observed responses used to cross-validate the model, specified as a numeric vector containing `NumObservations` elements.

Each row of  $Y$  represents the observed classification of the corresponding row of  $X$ .

Data Types: `single` | `double`

## Methods

### Inherited Methods

<code>kfoldfun</code>	Cross validate function
<code>kfoldLoss</code>	Cross-validation loss of partitioned regression model
<code>kfoldPredict</code>	Predict response for observations not used for training.

## Examples

### Train Cross-Validated SVM Regression Model Using `crossval`

This example shows how to train a cross-validated SVM regression model using `crossval`.

This example uses the abalone data from the UCI Machine Learning Repository. Download the data and save it in your current directory with the name 'abalone.data'. Read the data into a table.

```
tbl = readtable('abalone.data', 'Filetype', 'text', 'ReadVariableNames', false);  
rng default % for reproducibility
```



The sample data contains 4177 observations. All of the predictor variables are continuous except for `sex`, which is a categorical variable with possible values 'M' (for males), 'F' (for females), and 'I' (for infants). The goal is to predict the number of rings on the abalone, and thereby determine its age, using physical measurements.

Train an SVM regression model, using a Gaussian kernel function with a kernel scale equal to 2.2. Standardize the data.

```
mdl = fitrsvm(tbl, 'Var9', 'KernelFunction', 'gaussian', 'KernelScale', 2.2, 'Standardize', true)
```

`mdl` is a trained `RegressionSVM` regression model.

Cross validate the model using 10-fold cross validation.

```
CVmdl = crossval(mdl)
```

```
CVmdl =
```

```
classreg.learning.partition.RegistrationPartitionedSVM
  CrossValidatedModel: 'SVM'
  PredictorNames: {1x8 cell}
  CategoricalPredictors: 1
  ResponseName: 'Var9'
  NumObservations: 4177
  KFold: 10
  Partition: [1x1 cvpartition]
  ResponseTransform: 'none'
```

Properties, Methods

`CVmdl` is a `RegistrationPartitionedSVM` cross-validated regression model. The software:

1. Randomly partitions the data into ten equally-sized sets.
2. Trains an SVM regression model on nine of the ten sets.
3. Repeats steps 1 and 2  $k = 10$  times. It leaves out one of the partitions each time, and trains on the other nine partitions.
4. Combines generalization statistics for each fold.

Display the first of the 10 trained models.

```
FirstModel = CVMdl.Trained{1}

FirstModel =

classreg.learning.regr.CompactRegressionSVM
    PredictorNames: {1x8 cell}
    ResponseName: 'Var9'
    ResponseTransform: 'none'
    Alpha: [3553x1 double]
    Bias: 11.0623
    KernelParameters: [1x1 struct]
        Mu: [0 0 0 0.5242 0.4080 0.1393 0.8300 0.3599 0.1811 0.2392]
        Sigma: [1 1 1 0.1205 0.0995 0.0392 0.4907 0.2217 0.1103 0.1392]
    SupportVectors: [3553x10 double]
```

Properties, Methods

FirstModel is the first of the 10 trained CompactRegressionSVM models.

### Specify Cross-Validation Holdout Proportion for SVM Regression

This example shows how to specify a holdout proportion for training a cross-validated SVM regression model.

This example uses the abalone data from the UCI Machine Learning Repository. Download the data and save it in your current directory with the name 'abalone.data'. Read the data into a table.

```
tbl = readtable('abalone.data','Filetype','text','ReadVariableNames',false);
rng default % for reproducibility
```

The sample data contains 4177 observations. All of the predictor variables are continuous except for sex, which is a categorical variable with possible values 'M' (for males), 'F' (for females), and 'I' (for infants). The goal is to predict the number of rings on the abalone, and thereby determine its age, using physical measurements.

Train an SVM regression model, using a Gaussian kernel function with a kernel scale equal to 2.2. Standardize the data.

```
mdl = fitrsvm(tbl,'Var9','KernelFunction','gaussian','KernelScale',2.2,'Standardize',tr
```

mdl is a trained RegressionSVM regression model.

Cross validate the regression model by specifying a 10% holdout sample.

```
CVMdl = crossval(mdl, 'Holdout', 0.1)

CVMdl =

classreg.learning.partition.RegistrationPartitionedSVM
  CrossValidatedModel: 'SVM'
  PredictorNames: {1x8 cell}
  CategoricalPredictors: 1
  ResponseName: 'Var9'
  NumObservations: 4177
  KFold: 1
  Partition: [1x1 cvpartition]
  ResponseTransform: 'none'
```

Properties, Methods

CVMdl is a RegressionPartitionedSVM model object.

Extract and display the trained, compact SVM regression model from CVMdl.

```
CVMdl.Trained{1}

TrainedModel =

classreg.learning.regr.CompactRegressionSVM
  PredictorNames: {1x8 cell}
  ResponseName: 'Var9'
  ResponseTransform: 'none'
  Alpha: [3530x1 double]
  Bias: 11.2646
  KernelParameters: [1x1 struct]
  Mu: [0 0 0 0.5244 0.4080 0.1393 0.8282 0.3595 0.1805 0.2386]
  Sigma: [1 1 1 0.1198 0.0989 0.0388 0.4891 0.2218 0.1093 0.1390]
  SupportVectors: [3530x10 double]
```

Properties, Methods

TrainedModel is a CompactRegressionSVM regression model that was trained using 90% of the data.

## Alternatives

You can create a RegressionPartitionedSVM model using the following techniques:

- Use the training function `fitrsvm` and specify one of the 'CrossVal', 'Holdout', 'Kfold', or 'Leaveout' name-value pairs.
- Train a model using `fitrsvm`, then cross validate the model using the `crossval` method.
- Create a cross validation partition using `cvpartition`, then pass the resulting partition object to `fitrsvm` during training using the 'CVPartition' name-value pair.

## References

- [1] Nash, W.J., T. L. Sellers, S. R. Talbot, A. J. Cawthorn, and W. B. Ford. *The Population Biology of Abalone (Haliotis species) in Tasmania. I. Blacklip Abalone (H. rubra) from the North Coast and Islands of Bass Strait*, Sea Fisheries Division, Technical Report No. 48, 1994.
- [2] Waugh, S. *Extending and benchmarking Cascade-Correlation*, Ph.D. thesis, Computer Science Department, University of Tasmania, 1995.
- [3] Clark, D., Z. Schreter, A. Adams. *A Quantitative Comparison of Dystal and Backpropagation*, submitted to the Australian Conference on Neural Networks, 1996.
- [4] Lichman, M. *UCI Machine Learning Repository*, [<http://archive.ics.uci.edu/ml>]. Irvine, CA: University of California, School of Information and Computer Science.

## See Also

RegressionSVM | crossval | cvpartition | fitrsvm

Introduced in R2015b

## RegressionSVM class

### Superclasses:

Support vector machine regression model

### Description

`RegressionSVM` is a support vector machine (SVM) regression model. Train a `RegressionSVM` model using `fitrsvm` and the sample data.

`RegressionSVM` models store data, parameter values, support vectors, and algorithmic implementation information. You can use these models to:

- Estimate resubstitution predictions. For details, see `resubPredict`.
- Predict values for new data. For details, see `predict`.
- Compute resubstitution loss. For details, see `resubLoss`.
- Compute the mean square error or epsilon-insensitive loss. For details, see `loss`.

### Construction

`Mdl = fitrsvm(Tbl, ResponseVarName)` returns a full, trained support vector machine (SVM) regression model `Mdl` trained using the predictors values in the table `Tbl` and the response values in `Tbl.ResponseVarName`.

`Mdl = fitrsvm(Tbl, formula)` returns a full, trained support vector machine (SVM) regression model `Mdl` trained using the predictor and response data in the table `Tbl`. `formula` is an explanatory model of the response and a subset of predictor variables in `Tbl` used for training.

`Mdl = fitrsvm(Tbl, Y)` returns a full, trained support vector machine (SVM) regression model `Mdl` trained using the predictors values in the table `Tbl` and the response values in the vector `Y`.

`Mdl = fitrsvm(X, Y)` returns a full, trained support vector machine (SVM) regression model `Mdl` trained using the predictors values in the matrix `X` and the response values in the vector `Y`.

`Mdl = fitrsvm(____, Name, Value)` returns an SVM regression model with additional options specified by one or more name-value pair arguments, using any of the previous syntaxes. For example, you can specify the kernel function or train a cross-validated model.

## Input Arguments

### **Tbl** — Predictor data

table

Sample data used to train the model, specified as a table. Each row of `Tbl` corresponds to one observation, and each column corresponds to one predictor variable. Optionally, `Tbl` can contain one additional column for the response variable. Multi-column variables and cell arrays other than cell arrays of character vectors are not allowed.

If `Tbl` contains the response variable, and you want to use all remaining variables in `Tbl` as predictors, then specify the response variable using `ResponseVarName`.

If `Tbl` contains the response variable, and you want to use only a subset of the remaining variables in `Tbl` as predictors, then specify a formula using `formula`.

If `Tbl` does not contain the response variable, then specify a response variable using `Y`. The length of response variable and the number of rows of `Tbl` must be equal.

If a row of `Tbl` or an element of `Y` contains at least one NaN, then `fitrsvm` removes those rows and elements from both arguments when training the model.

To specify the names of the predictors in the order of their appearance in `Tbl`, use the `PredictorNames` name-value pair argument.

Data Types: `table`

### **ResponseVarName** — Response variable name

name of variable in `Tbl`

Response variable name, specified as the name of a variable in `Tbl`. The response variable must be a numeric vector.

You must specify `ResponseVarName` as a character vector. For example, if `Tbl` stores the response variable `Y` as `Tbl.Y`, then specify it as `'Y'`. Otherwise, the software treats all columns of `Tbl`, including `Y`, as predictors when training the model.

**formula** — Explanatory model of response and subset of predictor variables

character vector

Explanatory model of the response and a subset of the predictor variables, specified as a character vector in the form of `'Y~X1+X2+X3'`. In this form, `Y` represents the response variable, and `X1`, `X2`, and `X3` represent the predictor variables. The variables must be variable names in `Tbl` (`Tbl.Properties.VariableNames`).

To specify a subset of variables in `Tbl` as predictors for training the model, use a formula. If you specify a formula, then the software does not use any variables in `Tbl` that do not appear in `formula`.

Data Types: `char`

**y** — Response data

numeric vector

Response data, specified as an  $n$ -by-1 numeric vector. The length of `Y` and the number of rows of `Tbl` or `X` must be equal.

If a row of `Tbl` or `X`, or an element of `Y`, contains at least one `NaN`, then `fitrsvm` removes those rows and elements from both arguments when training the model.

To specify the response variable name, use the `ResponseName` name-value pair argument.

Data Types: `single` | `double`

**x** — Predictor data

numeric matrix

Predictor data to which the SVM regression model is fit, specified as an  $n$ -by- $p$  numeric matrix.  $n$  is the number of observations and  $p$  is the number of predictor variables.

The length of `Y` and the number of rows of `X` must be equal.

If a row of `X` or an element of `Y` contains at least one `NaN`, then `fitrsvm` removes those rows and elements from both arguments.



To specify the names of the predictors in the order of their appearance in  $X$ , use the `PredictorNames` name-value pair argument.

Data Types: `single` | `double`

---

**Note** The software treats NaN, empty character vector (''), and `<undefined>` elements as missing values. If a row of  $X$  or an element of  $Y$  contains at least one NaN, then the software removes those rows and elements from both arguments. Such deletion decreases the effective training or cross-validation sample size.

---

## Properties

### **Alpha** — Dual problem coefficients

vector of numeric values

Dual problem coefficients, specified as a vector of numeric values. `Alpha` contains  $m$  elements, where  $m$  is the number of support vectors in the trained SVM regression model. The dual problem introduces two Lagrange multipliers for each support vector. The values of `Alpha` are the differences between the two estimated Lagrange multipliers for the support vectors. For more details, see “Understanding Support Vector Machine Regression” on page 25-2.

If you specified to remove duplicates using `RemoveDuplicates`, then, for a particular set of duplicate observations that are support vectors, `Alpha` contains one coefficient corresponding to the entire set. That is, MATLAB attributes a nonzero coefficient to one observation from the set of duplicates and a coefficient of 0 to all other duplicate observations in the set.

Data Types: `single` | `double`

### **Beta** — Primal linear problem coefficients

vector of numeric values | ' [] '

Primal linear problem coefficients, stored as a numeric vector of length  $p$ , where  $p$  is the number of predictors in the SVM regression model.

The values in `Beta` are the linear coefficients for the primal optimization problem.

If the model is obtained using a kernel function other than `'linear'`, this property is empty (' [] ').

The `predict` method computes predicted response values for the model as  $Y_{FIT} = (X/S) \times \text{Beta} + \text{Bias}$ , where  $S$  is the value of the kernel scale stored in the `KernelParameters.Scale` property.

Data Types: `single` | `double`

### **Bias** — Bias term

scalar value

Bias term in the SVM regression model, stored as a scalar value.

Data Types: `single` | `double`

### **BoxConstraints** — Box constraints for dual problem coefficients

vector of numeric values

Box constraints for dual problem alpha coefficients, stored as a numeric vector containing  $n$  elements, where  $n$  is the number of observations in  $X$  (`Mdl.NumObservations`).

The absolute value of the dual coefficient `Alpha` for observation  $i$  cannot exceed `BoxConstraints(i)`.

If you specify removing duplicates using `'RemoveDuplicates'`, then for a given set of duplicate observations, MATLAB sums the box constraints, and then attributes the sum to one observation and box constraints of 0 to all other observations in the set.

Data Types: `single` | `double`

### **CacheInfo** — Caching information

structure

Caching information, stored as a structure with the following fields.

Field	Description
<code>'Size'</code>	Positive scalar value indicating the cache size (in MB) that the software reserves to store entries of the Gram matrix. Set the cache size by using the <code>'CacheSize'</code> name-value pair argument in <code>fitrsvm</code> .

Field	Description
'Algorithm'	Character vector containing the name of the algorithm used to remove entries from the cache when its capacity is exceeded. Currently, the only available caching algorithm is 'Queue'. You cannot set the caching algorithm.

Data Types: struct

### **CategoricalPredictors** — Indices of categorical predictors

vector of positive integers

Indices of categorical predictors, stored as a vector of positive integers. `CategoricalPredictors` contains index values corresponding to the columns of the predictor data that contain categorical predictors. If none of the predictors are categorical, then this property is empty (`[]`).

Data Types: single | double

### **ConvergenceInfo** — Convergence information

structure

Convergence information, stored as a structure with the following fields.

Field	Description
Converged	Logical flag indicating whether the algorithm converged. A value of 1 indicates convergence.
ReasonForConvergence	Character vector indicating the criterion the software used to detect convergence.
Gap	Scalar feasibility gap between the dual and primal objective functions.
GapTolerance	Scalar tolerance for the feasibility gap. You can set this tolerance using the 'GapTolerance' name-value pair argument in <code>fitrsvm</code> .
DeltaGradient	Scalar gradient difference between upper and lower violators.

Field	Description
DeltaGradientTolerance	Scalar tolerance for the gradient difference. You can set this tolerance using the DeltaGradientTolerance name-value pair argument in <code>fitrsvm</code> .
LargestKKTViolation	Maximal scalar Karush-Kuhn-Tucker (KKT) violation value.
KKTTolerance	Scalar tolerance for the largest KKT violation. You can set this tolerance using the 'KKTTolerance' name-value pair argument in <code>fitrsvm</code> .
History	Structure containing convergence information recorded at periodic intervals during the model training process. This structure contains the following fields: <ul style="list-style-type: none"> <li>• NumIterations — Array of iteration indices at which the software recorded convergence criteria.</li> <li>• Gap — Gap values at these iterations.</li> <li>• DeltaGradient — DeltaGradient values at these iterations.</li> <li>• LargestKKTViolation — LargestKKTViolation values at these iterations.</li> <li>• NumSupportVectors — Number of support vectors at these iterations.</li> <li>• Objective — Objective values at these iterations.</li> </ul>
Objective	Numeric value of the dual objective.

Data Types: struct

**Epsilon** — Half the width of the epsilon-insensitive band

nonnegative scalar value

Half the width of the epsilon-insensitive band, stored as a nonnegative scalar value.

Data Types: `single` | `double`

### **ExpandedPredictorNames** — Expanded predictor names

cell array of character vectors

Expanded predictor names, stored as a cell array of character vectors.

If the model uses encoding for categorical variables, then `ExpandedPredictorNames` includes the names that describe the expanded variables. Otherwise, `ExpandedPredictorNames` is the same as `PredictorNames`.

### **Gradient** — Gradient values in training data

vector of numeric values

Gradient values in training data, stored as a numeric vector containing  $2n$  elements, where  $n$  is the number of observations in the training data.

Element  $i$  of `Gradient` contains the gradient value for the Alpha coefficient that corresponds to the upper boundary of the epsilon-insensitive band at observation  $i$  at the end of the optimization.

Element  $i + NumObservations$  of `Gradient` contains the gradient value for the Alpha coefficient that corresponds to the lower boundary of the epsilon-insensitive band at observation  $i$  at the end of the optimization.

Data Types: `single` | `double`

### **HyperparameterOptimizationResults** — Description of the cross-validation optimization of hyperparameters

`BayesianOptimization` object | table of hyperparameters and associated values

Description of the cross-validation optimization of hyperparameters, stored as a `BayesianOptimization` object or a table of hyperparameters and associated values. Nonempty when the `OptimizeHyperparameters` name-value pair is nonempty at creation. Value depends on the setting of the `HyperparameterOptimizationOptions` name-value pair at creation:

- `'bayesopt'` (default) — Object of class `BayesianOptimization`

- `'gridsearch'` or `'randomsearch'` — Table of hyperparameters used, observed objective function values (cross-validation loss), and rank of observations from lowest (best) to highest (worst)

Data Types: `table`

### **IsSupportVector** — Flag indicating whether observation is support vector

logical vector

Flag indicating whether an observation is a support vector, stored as an  $n$ -by-1 logical vector.  $n$  is the number of observations in  $X$  (see `NumObservations`). A value of 1 indicates that the corresponding observation in the training data is a support vector.

If you specify removing duplicates using `RemoveDuplicates`, then for a given set of duplicate observations that are support vectors, `IsSupportVector` flags only one as a support vector.

Data Types: `logical`

### **KernelParameters** — Kernel function parameters

structure

Kernel function parameters, stored as a structure with the following fields.

Field	Description
Function	Kernel function name (a character vector).
Scale	Numeric scale factor used to divide predictor values.

You can specify values for `KernelParameters.Function` and `KernelParameters.Scale` by using the `KernelFunction` and `KernelScale` name-value pair arguments in `fitrsvm`, respectively.

Data Types: `struct`

### **ModelParameters** — Parameter values

model parameter object

Parameter values used to train the SVM regression model, stored as a model parameter object. Access the properties of `ModelParameters` using dot notation. For example, access the value of `Epsilon` used to train the model as `Mdl.ModelParameters.Epsilon`.

**Mu — Predictor means**

vector of numeric values | ' [] '

Predictor means, stored as a vector of numeric values.

If the predictors are standardized, then `Mu` is a numeric vector of length  $p$ , where  $p$  is the number of predictors used to train the model. In this case, the `predict` method centers predictor matrix `X` by subtracting the corresponding element of `Mu` from each column.

If the predictors are not standardized, then `Mu` is empty ( ' [] ' ).

If the data contains categorical predictors, then `Mu` includes elements for the dummy variables for those predictors. The corresponding entries in `Mu` are 0 because dummy variables are not centered or scaled.

Data Types: `single` | `double`

**NumIterations — Number of iterations required for convergence**

positive integer value

Number of iterations required for the optimization routine to reach convergence, stored as a positive integer value.

To set a limit on the number of iterations, use the `'IterationLimit'` name-value pair argument of `fitrsvm`.

Data Types: `single` | `double`

**NumObservations — Number of observations**

positive integer value

Number of observations in the training data, stored as a positive integer value.

Data Types: `single` | `double`

**PredictorNames — Predictor names**

cell array of character vectors

Predictor names, stored as a cell array of character vectors containing the name of each predictor in the order they appear in `X`. `PredictorNames` has a length equal to the number of columns in `X`.

Data Types: `cell`

**OutlierFraction** — Expected fraction of outliers

scalar value in the range [0,1]

Expected fraction of outliers in the training set, stored as a scalar value in the range [0,1]. You can specify the expected fraction of outliers using the 'OutlierFraction' name-value pair argument in `fitrsvm`.

Data Types: `double`**ResponseName** — Response variable name

character vector

Response variable name, stored as a character vector.

Data Types: `char`**ResponseTransform** — Response transformation function

'none' | function handle

Response transformation function, stored as 'none' or a function handle. `ResponseTransform` describes how the software transforms raw response values.

For a MATLAB function, or a function that you define, enter its function handle. For example, you can enter `Mdl.ResponseTransform = @function`, where `function` accepts a numeric vector of the original responses and returns a numeric vector of the same size containing the transformed responses.

Data Types: `char` | `function_handle`**ShrinkagePeriod** — Number of iterations between reductions of active set

nonnegative integer value

Number of iterations between reductions of the active set during optimization, stored as a nonnegative integer value.

You can set the shrinkage period by using the 'ShrinkagePeriod' name-value pair argument in `fitrsvm`.

Data Types: `single` | `double`**Sigma** — Predictor standard deviations

vector of numeric values | '[]'

Predictor standard deviations, stored as a vector of numeric values.



If the predictors are standardized, then `Sigma` is a numeric vector of length  $p$ , where  $p$  is the number of predictors used to train the model. In this case, the `predict` method scales the predictor matrix  $X$  by dividing each column by the corresponding element of `Sigma`, after centering each element using `Mu`.

If the predictors are not standardized, then `Sigma` is empty (`[]`).

If the data contains categorical predictors, `Sigma` includes elements for the dummy variables for those predictors. The corresponding entries in `Sigma` are 1, because dummy variables are not centered or scaled.

Data Types: `single` | `double`

### **`solver`** — Name of solver algorithm

character vector

Name of the solver algorithm used to solve the optimization problem, stored as a value in this table.

Value	Description
'SMO'	Sequential Minimal Optimization
'ISDA'	Iterative Single Data Algorithm
'L1QP'	L1 soft-margin minimization by quadratic programming (requires an Optimization Toolbox license).

Data Types: `char`

### **`SupportVectors`** — Support vectors

matrix of numeric values

Support vectors, stored as an  $m$ -by- $p$  matrix of numeric values.  $m$  is the number of support vectors (`sum(Mdl.IsSupportVector)`), and  $p$  is the number of predictors in  $X$ .

If you specified to remove duplicates using `RemoveDuplicates`, then for a given set of duplicate observations that are support vectors, `SupportVectors` contains one unique support vector.

Data Types: `single` | `double`

### **`w`** — Observation weights

vector of numeric values

Observation weights used to train the model, stored as a numeric vector containing `NumObservation` number of elements. `fitrsvm` normalizes the weights used for training so that they sum to 1.

Data Types: `single` | `double`

### **x** — Predictor values

matrix of numeric values | table of numeric values

Predictor values used to train the model, stored as a matrix of numeric values if the model is trained on a matrix, or a table if the model is trained on a table. `x` has size  $n$ -by- $p$ , where  $n$  is the number of rows and  $p$  is the number of predictor variables or columns in the training data.

Data Types: `single` | `double` | `table`

### **y** — Observed response values

vector of numeric values

Observed response values, stored as a numeric vector containing `NumObservations` number of elements.

Data Types: `single` | `double`

## Methods

<code>compact</code>	Compact support vector machine regression model
<code>crossval</code>	Cross-validated support vector machine regression model
<code>resubLoss</code>	Resubstitution loss for support vector machine regression model
<code>resubPredict</code>	Predict resubstitution response of support vector machine regression model
<code>resume</code>	Resume training support vector machine regression model

## Inherited Methods

<code>discardSupportVectors</code>	Discard support vectors
<code>loss</code>	Regression error for support vector machine regression model
<code>predict</code>	Predict responses using support vector machine regression model

## Copy Semantics

Value. To learn how value classes affect copy operations, see [Copying Objects \(MATLAB\)](#).

## Examples

### Train Linear Support Vector Machine Regression Model

This example shows how to train a linear support vector machine (SVM) regression model using sample data stored in matrices.

Load the `carsmall` data set.

```
load carsmall
rng default % for reproducibility
```

Specify `Horsepower` and `Weight` as the predictor variables (`X`) and `MPG` as the response variable (`Y`).

```
X = [Horsepower,Weight];
Y = MPG;
```

Train a linear SVM regression model.

```
Mdl = fitrsvm(X,Y)
```

```
Mdl =
```

```
RegressionSVM
    ResponseName: 'Y'
```

```
CategoricalPredictors: []
  ResponseTransform: 'none'
    Alpha: [76x1 double]
    Bias: 43.1920
  KernelParameters: [1x1 struct]
  NumObservations: 93
  BoxConstraints: [93x1 double]
  ConvergenceInfo: [1x1 struct]
  IsSupportVector: [93x1 logical]
  Solver: 'SMO'
```

The Command Window shows that `Mdl` is a trained `RegressionSVM` model and a list of its properties.

Check the model for convergence.

```
Mdl.ConvergenceInfo.Converged
```

```
ans =
    logical
    0
```

0 indicates that the model did not converge.

```
MdlStd = fitrsvm(X,Y,'Standardize',true)
```

```
MdlStd =
    RegressionSVM
      ResponseName: 'Y'
  CategoricalPredictors: []
    ResponseTransform: 'none'
      Alpha: [77x1 double]
      Bias: 22.9131
    KernelParameters: [1x1 struct]
      Mu: [109.3441 2.9625e+03]
      Sigma: [45.3545 805.9668]
    NumObservations: 93
```

```
BoxConstraints: [93x1 double]
ConvergenceInfo: [1x1 struct]
IsSupportVector: [93x1 logical]
Solver: 'SMO'
```

Check the model for convergence.

```
MdlStd.ConvergenceInfo.Converged
```

```
ans =
    logical
     1
```

1 indicates that the model did converge.

Compute the resubstitution mean squared error for the new model.

```
lStd = resubLoss(MdlStd)
```

```
lStd =
    17.0256
```

## Train Support Vector Machine Regression Model

Train a support vector machine regression model using the abalone data from the UCI Machine Learning Repository.

Download the data and save it in your current folder with the name 'abalone.csv'.

```
url = 'https://archive.ics.uci.edu/ml/machine-learning-databases/abalone/abalone.data';
websave('abalone.csv',url);
```

Read the data into a table. Specify the variable names.

```
varnames = {'Sex'; 'Length'; 'Diameter'; 'Height'; 'Whole_weight';...  
           'Shucked_weight'; 'Viscera_weight'; 'Shell_weight'; 'Rings'};  
Tbl = readtable('abalone.csv', 'Filetype', 'text', 'ReadVariableNames', false);  
Tbl.Properties.VariableNames = varnames;
```

The sample data contains 4177 observations. All the predictor variables are continuous except for Sex, which is a categorical variable with possible values 'M' (for males), 'F' (for females), and 'I' (for infants). The goal is to predict the number of rings (stored in Rings) on the abalone and determine its age using physical measurements.

Train an SVM regression model, using a Gaussian kernel function with an automatic kernel scale. Standardize the data.

```
rng default % For reproducibility  
Mdl = fitcsvm(Tbl, 'Rings', 'KernelFunction', 'gaussian', 'KernelScale', 'auto', ...  
             'Standardize', true)
```

```
Mdl =
```

```
RegressionSVM  
    PredictorNames: {1×8 cell}  
    ResponseName: 'Rings'  
    CategoricalPredictors: 1  
    ResponseTransform: 'none'  
        Alpha: [3635×1 double]  
        Bias: 10.8144  
    KernelParameters: [1×1 struct]  
        Mu: [1×10 double]  
        Sigma: [1×10 double]  
    NumObservations: 4177  
    BoxConstraints: [4177×1 double]  
    ConvergenceInfo: [1×1 struct]  
    IsSupportVector: [4177×1 logical]  
    Solver: 'SMO'
```

The Command Window shows that Mdl is a trained RegressionSVM model and displays a property list.

Display the properties of Mdl using dot notation. For example, check to confirm whether the model converged and how many iterations it completed.

```
conv = Mdl.ConvergenceInfo.Converged  
iter = Mdl.NumIterations
```

```
conv =  
    logical
```

```
    1
```

```
iter =  
    2759
```

The returned results indicate that the model converged after 2759 iterations.

## References

- [1] Nash, W.J., T. L. Sellers, S. R. Talbot, A. J. Cawthorn, and W. B. Ford. *The Population Biology of Abalone (Haliotis species) in Tasmania. I. Blacklip Abalone (H. rubra) from the North Coast and Islands of Bass Strait*, Sea Fisheries Division, Technical Report No. 48, 1994.
- [2] Waugh, S. *Extending and benchmarking Cascade-Correlation*, Ph.D. thesis, Computer Science Department, University of Tasmania, 1995.
- [3] Clark, D., Z. Schreter, A. Adams. *A Quantitative Comparison of Dystal and Backpropagation*, submitted to the Australian Conference on Neural Networks, 1996.
- [4] Lichman, M. *UCI Machine Learning Repository*, [<http://archive.ics.uci.edu/ml>]. Irvine, CA: University of California, School of Information and Computer Science.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Only the `predict` function supports code generation.
- When you train an SVM regression model by using `fitrsvm`, the following restrictions apply.
  - Code generation does not support categorical predictors. You cannot supply training data in a table that contains at least one categorical predictor, and you cannot use the `'CategoricalPredictors'` name-value pair argument. To dummy-code variables that you want treated as categorical, preprocess the categorical data by using `dummyvar` before fitting the model.
  - The value of the `'ResponseTransform'` name-value pair argument cannot be an anonymous function.

### See Also

`CompactRegressionSVM` | `RegressionPartitionedSVM` | `fitrsvm` | `plotPartialDependence`

### Topics

“Understanding Support Vector Machine Regression” on page 25-2

Introduced in R2015b



# RegressionTree class

## Superclasses:

Regression tree

## Description

A decision tree with binary splits for regression. An object of class `RegressionTree` can predict responses for new data with the `predict` method. The object contains the data used for training, so can compute resubstitution predictions.

## Construction

`tree = fitrtree(tbl, y)` returns a regression tree based on the input variables (also known as predictors, features, or attributes) in the table `tbl` and output (response) contained in `tbl.ResponseVarName`. `tree` is a binary tree where each branching node is split based on the values of a column of `tbl`.

`tree = fitrtree(tbl, formula)` returns a regression tree based on the input variables contained in the table `tbl`. `formula` is a formula that identifies the response and predictor variables in TBL used for training.

`tree = fitrtree(tbl, y)` returns a regression tree based on the input variables contained in the table `tbl` and output contained in `y`.

`tree = fitrtree(x, y)` returns a regression tree based on the input variables `x` and output `y`. `tree` is a binary tree where each branching node is split based on the values of a column of `x`.

`tree = fitrtree( ____, Name, Value)` fits a tree with additional options specified by one or more `Name, Value` pair arguments. For example, you can specify observation weights or train a cross-validated model.

If you use one of the following five options, `tree` is of class `RegressionPartitionedModel`: `'CrossVal'`, `'KFold'`, `'Holdout'`, `'Leaveout'`, or `'CVPartition'`. Otherwise, `tree` is of class `RegressionTree`.

## Input Arguments

### **tbl** — Sample data

table

Sample data used to train the model, specified as a table. Each row of `tbl` corresponds to one observation, and each column corresponds to one predictor variable. Optionally, `tbl` can contain one additional column for the response variable. Multi-column variables and cell arrays other than cell arrays of character vectors are not allowed.

If `tbl` contains the response variable, and you want to use all remaining variables in `tbl` as predictors, then specify the response variable using `ResponseVarName`.

If `tbl` contains the response variable, and you want to use only a subset of the remaining variables in `tbl` as predictors, then specify a formula using `formula`.

If `tbl` does not contain the response variable, then specify a response variable using `y`. The length of response variable and the number of rows of `tbl` must be equal.

Data Types: `table`

### **x** — Predictor data

numeric matrix

Predictor data, specified as numeric matrix. Each column of `X` represents one variable, and each row represents one observation.

`fitrtree` considers NaN values in `X` as missing values. `fitrtree` does not use observations with all missing values for `X` the fit. `fitrtree` uses observations with some missing values for `X` to find splits on variables for which these observations have valid values.

Data Types: `single` | `double`

### **ResponseVarName** — Response variable name

name of variable in `tbl`

Response variable name, specified as the name of a variable in `Tbl`. The response variable must be a numeric vector.

You must specify `ResponseVarName` as a character vector. For example, if `Tbl` stores the response variable `Y` as `Tbl.Y`, then specify it as `'Y'`. Otherwise, the software treats all columns of `Tbl`, including `Y`, as predictors when training the model.

**formula — Response and predictor variables to use in model training**

character vector in the form of `'Y~X1+X2+X3'`

Response and predictor variables to use in model training, specified as a character vector in the form of `'Y~X1+X2+X3'`. In this form, `Y` represents the response variable, and `X1`, `X2`, and `X3` represent the predictor variables.

To specify a subset of variables in `tbl` as predictors for training the model, use a formula. If you specify a formula, then any variables in `tbl` that do not appear in `formula` are not used to train the model.

Data Types: `char`

**y — Response data**

numeric column vector

Response data, specified as a numeric column vector with the same number of rows as `X`. Each entry in `Y` is the response to the data in the corresponding row of `X`.

The software considers NaN values in `Y` to be missing values. `fitrtree` does not use observations with missing values for `Y` in the fit.

Data Types: `single` | `double`

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

**CategoricalPredictors — Categorical predictors list**

numeric or logical vector | cell array of character vectors | character matrix | `'all'`

Categorical predictors list, specified as the comma-separated pair consisting of `'CategoricalPredictors'` and one of the following:

- A numeric vector with indices from 1 through `p`, where `p` is the number of columns of `X`.
- A logical vector of length `p`, where a `true` entry means that the corresponding column of `X` is a categorical variable.
- A cell array of character vectors, where each element in the array is the name of a predictor variable. The names must match entries in `PredictorNames` values.
- A character matrix, where each row of the matrix is a name of a predictor variable. The names must match entries in `PredictorNames` values. Pad the names with extra blanks so each row of the character matrix has the same length.
- `'all'`, meaning all predictors are categorical.

By default, if the predictor data is in a matrix (`X`), the software assumes that none of the predictors are categorical. If the predictor data is in a table (`Tbl`), the software assumes that a variable is categorical if it contains, logical values, values of the unordered data type `categorical`, or a cell array of character vectors.

Example: `'CategoricalPredictors', 'all'`

Data Types: `single` | `double` | `char` | `logical` | `cell`

### **CrossVal** — Cross-validation flag

`'off'` (default) | `'on'`

Cross-validation flag, specified as the comma-separated pair consisting of `'CrossVal'` and either `'on'` or `'off'`.

If `'on'`, `fitrtree` grows a cross-validated decision tree with 10 folds. You can override this cross-validation setting using one of the `'Kfold'`, `'Holdout'`, `'Leaveout'`, or `'CVPartition'` name-value pair arguments. You can only use one of these four options (`'Kfold'`, `'Holdout'`, `'Leaveout'`, or `'CVPartition'`) at a time when creating a cross-validated tree.

Alternatively, cross-validate tree later using the `crossval` method.

Example: `'CrossVal', 'on'`

### **CVPartition** — Partition for cross-validation tree

`cvpartition` object

Partition for cross-validated tree, specified as the comma-separated pair consisting of `'CVPartition'` and an object created using `cvpartition`.

If you use 'CVPartition', you cannot use any of the 'KFold', 'Holdout', or 'Leaveout' name-value pair arguments.

**Holdout — Fraction of data for holdout validation**

0 (default) | scalar value in the range [0,1]

Fraction of data used for holdout validation, specified as the comma-separated pair consisting of 'Holdout' and a scalar value in the range [0,1]. Holdout validation tests the specified fraction of the data, and uses the rest of the data for training.

If you use 'Holdout', you cannot use any of the 'CVPartition', 'KFold', or 'Leaveout' name-value pair arguments.

Example: 'Holdout',0.1

Data Types: single | double

**KFold — Number of folds**

10 (default) | positive integer greater than 1

Number of folds to use in a cross-validated tree, specified as the comma-separated pair consisting of 'KFold' and a positive integer value greater than 1.

If you use 'KFold', you cannot use any of the 'CVPartition', 'Holdout', or 'Leaveout' name-value pair arguments.

Example: 'KFold',8

Data Types: single | double

**Leaveout — Leave-one-out cross-validation flag**

'off' (default) | 'on'

Leave-one-out cross-validation flag, specified as the comma-separated pair consisting of 'Leaveout' and either 'on' or 'off'. Use leave-one-out cross validation by setting to 'on'.

If you use 'Leaveout', you cannot use any of the 'CVPartition', 'Holdout', or 'KFold' name-value pair arguments.

Example: 'Leaveout', 'on'

**MaxNumSplits — Maximal number of decision splits**

size(X,1) - 1 (default) | positive integer

Maximal number of decision splits (or branch nodes), specified as the comma-separated pair consisting of `'MaxNumSplits'` and a positive integer. `RegressionTree` splits `MaxNumSplits` or fewer branch nodes. For more details on splitting behavior, see “Tree Depth Control” on page 32-2668.

Example: `'MaxNumSplits', 5`

Data Types: `single | double`

### **MergeLeaves — Leaf merge flag**

`'on'` (default) | `'off'`

Leaf merge flag, specified as the comma-separated pair consisting of `'MergeLeaves'` and `'on'` or `'off'`.

If `MergeLeaves` is `'on'`, then `RegressionTree`:

- Merges leaves that originate from the same parent node, and that yields a sum of risk values greater or equal to the risk associated with the parent node
- Estimates the optimal sequence of pruned subtrees, but does not prune the regression tree

Otherwise, `RegressionTree` does not merge leaves.

Example: `'MergeLeaves', 'off'`

### **MinLeafSize — Minimum number of leaf node observations**

1 (default) | positive integer value

Minimum number of leaf node observations, specified as the comma-separated pair consisting of `'MinLeafSize'` and a positive integer value. Each leaf has at least `MinLeafSize` observations per tree leaf. If you supply both `MinParentSize` and `MinLeafSize`, `fitrtree` uses the setting that gives larger leaves: `MinParentSize = max(MinParentSize, 2*MinLeafSize)`.

Example: `'MinLeafSize', 3`

Data Types: `single | double`

### **MinParentSize — Minimum number of branch node observations**

10 (default) | positive integer value

Minimum number of branch node observations, specified as the comma-separated pair consisting of `'MinParentSize'` and a positive integer value. Each branch node in the

tree has at least `MinParentSize` observations. If you supply both `MinParentSize` and `MinLeafSize`, `fitrtree` uses the setting that gives larger leaves: `MinParentSize = max(MinParentSize, 2*MinLeafSize)`.

Example: `'MinParentSize', 8`

Data Types: `single` | `double`

### **NumVariablesToSample** — Number of predictors for split

`'all'` (default) | positive integer value

Number of predictors to select at random for each split, specified as the comma-separated pair consisting of `'NumVariablesToSample'` and a positive integer value. You can also specify `'all'` to use all available predictors.

Example: `'NumVariablesToSample', 3`

Data Types: `single` | `double`

### **PredictorNames** — Predictor variable names

cell array of unique character vectors

Predictor variable names, specified as the comma-separated pair consisting of `'PredictorNames'` and a cell array of unique character vectors. The functionality of `'PredictorNames'` depends on the way you supply the training data.

- If you supply `X` and `Y`, then you can use `'PredictorNames'` to give the predictor variables in `X` names.
  - The order of the names in `PredictorNames` must correspond to the column order of `X`. That is, `PredictorNames{1}` is the name of `X(:, 1)`, `PredictorNames{2}` is the name of `X(:, 2)`, and so on. Also, `size(X, 2)` and `numel(PredictorNames)` must be equal.
  - By default, `PredictorNames` is `{'x1', 'x2', ...}`.
- If you supply `Tbl`, then you can use `'PredictorNames'` to choose which predictor variables to use in training. That is, `RegressionTree` uses the predictor variables in `PredictorNames` and the response only in training.
  - `PredictorNames` must be a subset of `Tbl.Properties.VariableNames` and cannot include the name of the response variable.
  - By default, `PredictorNames` contains the names of all predictor variables.

- It good practice to specify the predictors for training using one of 'PredictorNames' or formula only.

```
Example: 'PredictorNames',  
{'SepalLength', 'SepalWidth', 'PedalLength', 'PedalWidth'}
```

Data Types: cell

### **Prune — Flag to estimate optimal sequence of pruned subtrees**

'on' (default) | 'off'

Flag to estimate the optimal sequence of pruned subtrees, specified as the comma-separated pair consisting of 'Prune' and 'on' or 'off'.

If Prune is 'on', then RegressionTree grows the regression tree and estimates the optimal sequence of pruned subtrees, but does not prune the regression tree. Otherwise, RegressionTree grows the regression tree without estimating the optimal sequence of pruned subtrees.

To prune a trained regression tree, pass the regression tree to prune.

```
Example: 'Prune', 'off'
```

### **PruneCriterion — Pruning criterion**

'error' (default)

Pruning criterion, specified as the comma-separated pair consisting of 'PruneCriterion' and 'error'.

```
Example: 'PruneCriterion', 'error'
```

### **QuadraticErrorTolerance — Quadratic error tolerance**

1e-6 (default) | positive scalar value

Quadratic error tolerance per node, specified as the comma-separated pair consisting of 'QuadraticErrorTolerance' and a positive scalar value. Splitting nodes stops when quadratic error per node drops below QuadraticErrorTolerance\*QED, where QED is the quadratic error for the entire data computed before the decision tree is grown.

```
Example: 'QuadraticErrorTolerance', 1e-4
```

### **ResponseName — Response variable name**

'Y' (default) | character vector



Response variable name, specified as the comma-separated pair consisting of 'ResponseName' and a character vector.

- If you supply Y, then you can use 'ResponseName' to specify a name for the response variable.
- If you supply ResponseVarName or formula, then you cannot use 'ResponseName'.

Example: 'ResponseName', 'response'

Data Types: char

### **ResponseTransform — Response transform function**

'none' (default) | function handle

Response transform function for transforming the raw response values, specified as the comma-separated pair consisting of 'ResponseTransform' and either a function handle or 'none'. The function handle must accept a matrix of response values and return a matrix of the same size. The default is 'none', which means  $@(x) x$ , or no transformation.

Add or change a ResponseTransform function using dot notation:

```
tree.ResponseTransform = @function
```

Data Types: char | function\_handle

### **SplitCriterion — Split criterion**

'MSE' (default)

Split criterion, specified as the comma-separated pair consisting of 'SplitCriterion' and 'MSE', meaning mean squared error.

Example: 'SplitCriterion', 'MSE'

### **Surrogate — Surrogate decision splits flag**

'off' (default) | 'on' | 'all' | positive integer

Surrogate decision splits flag, specified as the comma-separated pair consisting of 'Surrogate' and 'on', 'off', 'all', or a positive integer.

- When 'on', fitrtree finds at most 10 surrogate splits at each branch node.

- When set to a positive integer, `fitrtree` finds at most the specified number of surrogate splits at each branch node.
- When set to 'all', `fitrtree` finds all surrogate splits at each branch node. The 'all' setting can use much time and memory.

Use surrogate splits to improve the accuracy of predictions for data with missing values. The setting also enables you to compute measures of predictive association between predictors.

Example: 'Surrogate', 'on'

Data Types: `single` | `double` | `char`

### **weights** — Observation weights

`ones(size(X,1),1)` (default) | vector of scalar values

Observation weights, specified as the comma-separated pair consisting of 'Weights' and a vector of scalar values. The software weights the observations in each row of `X` or `Tbl` with the corresponding value in `Weights`. The size of `Weights` must equal the number of rows in `X` or `Tbl`.

If you specify the input data as a table `Tbl`, then `Weights` can be the name of a variable in `Tbl` that contains a numeric vector. In this case, you must specify `Weights` as a character vector. For example, if weights vector `W` is stored as `Tbl.W`, then specify it as 'W'. Otherwise, the software treats all columns of `Tbl`, including `W`, as predictors when training the model.

`fitrtree` normalizes the weights in each class to add up to 1.

Data Types: `single` | `double`

## Properties

### **CategoricalPredictors**

Indices of categorical predictors, stored as a vector of positive integers. `CategoricalPredictors` contains index values corresponding to the columns of the predictor data that contain categorical predictors. If none of the predictors are categorical, then this property is empty (`[]`).

**CategoricalSplits**

An  $n$ -by-2 cell array, where  $n$  is the number of categorical splits in `tree`. Each row in `CategoricalSplits` gives left and right values for a categorical split. For each branch node with categorical split  $j$  based on a categorical predictor variable  $z$ , the left child is chosen if  $z$  is in `CategoricalSplits(j,1)` and the right child is chosen if  $z$  is in `CategoricalSplits(j,2)`. The splits are in the same order as nodes of the tree. Nodes for these splits can be found by running `cuttype` and selecting 'categorical' cuts from top to bottom.

**Children**

An  $n$ -by-2 array containing the numbers of the child nodes for each node in `tree`, where  $n$  is the number of nodes. Leaf nodes have child node 0.

**CutCategories**

An  $n$ -by-2 cell array of the categories used at branches in `tree`, where  $n$  is the number of nodes. For each branch node  $i$  based on a categorical predictor variable  $x$ , the left child is chosen if  $x$  is among the categories listed in `CutCategories{i,1}`, and the right child is chosen if  $x$  is among those listed in `CutCategories{i,2}`. Both columns of `CutCategories` are empty for branch nodes based on continuous predictors and for leaf nodes.

`CutPoint` contains the cut points for 'continuous' cuts, and `CutCategories` contains the set of categories.

**CutPoint**

An  $n$ -element vector of the values used as cut points in `tree`, where  $n$  is the number of nodes. For each branch node  $i$  based on a continuous predictor variable  $x$ , the left child is chosen if  $x < \text{CutPoint}(i)$  and the right child is chosen if  $x \geq \text{CutPoint}(i)$ . `CutPoint` is NaN for branch nodes based on categorical predictors and for leaf nodes.

**CutType**

An  $n$ -element cell array indicating the type of cut at each node in `tree`, where  $n$  is the number of nodes. For each node  $i$ , `CutType{i}` is:

- 'continuous' — If the cut is defined in the form  $x < v$  for a variable  $x$  and cut point  $v$ .

- `'categorical'` — If the cut is defined by whether a variable `x` takes a value in a set of categories.
- `''` — If `i` is a leaf node.

`CutPoint` contains the cut points for `'continuous'` cuts, and `CutCategories` contains the set of categories.

### **CutPredictor**

An  $n$ -element cell array of the names of the variables used for branching in each node in tree, where  $n$  is the number of nodes. These variables are sometimes known as *cut variables*. For leaf nodes, `CutPredictor` contains an empty character vector.

`CutPoint` contains the cut points for `'continuous'` cuts, and `CutCategories` contains the set of categories.

### **ExpandedPredictorNames**

Expanded predictor names, stored as a cell array of character vectors.

If the model uses encoding for categorical variables, then `ExpandedPredictorNames` includes the names that describe the expanded variables. Otherwise, `ExpandedPredictorNames` is the same as `PredictorNames`.

### **HyperparameterOptimizationResults**

Description of the cross-validation optimization of hyperparameters, stored as a `BayesianOptimization` object or a table of hyperparameters and associated values. Nonempty when the `OptimizeHyperparameters` name-value pair is nonempty at creation. Value depends on the setting of the `HyperparameterOptimizationOptions` name-value pair at creation:

- `'bayesopt'` (default) — Object of class `BayesianOptimization`
- `'gridsearch'` or `'randomsearch'` — Table of hyperparameters used, observed objective function values (cross-validation loss), and rank of observations from lowest (best) to highest (worst)

### **IsBranchNode**

An  $n$ -element logical vector `ib` that is `true` for each branch node and `false` for each leaf node of tree.

**ModelParameters**

Object holding parameters of `tree`.

**NumObservations**

Number of observations in the training data, a numeric scalar. `NumObservations` can be less than the number of rows of input data `X` when there are missing values in `X` or response `Y`.

**NodeError**

An  $n$ -element vector  $e$  of the errors of the nodes in `tree`, where  $n$  is the number of nodes.  $e(i)$  is the misclassification probability for node  $i$ .

**NodeMean**

An  $n$ -element numeric array with mean values in each node of `tree`, where  $n$  is the number of nodes in the tree. Every element in `NodeMean` is the average of the true `Y` values over all observations in the node.

**NodeProbability**

An  $n$ -element vector  $p$  of the probabilities of the nodes in `tree`, where  $n$  is the number of nodes. The probability of a node is computed as the proportion of observations from the original data that satisfy the conditions for the node. This proportion is adjusted for any prior probabilities assigned to each class.

**NodeRisk**

An  $n$ -element vector of the risk of the nodes in the tree, where  $n$  is the number of nodes. The risk for each node is the node error weighted by the node probability.

**NodeSize**

An  $n$ -element vector `sizes` of the sizes of the nodes in `tree`, where  $n$  is the number of nodes. The size of a node is defined as the number of observations from the data used to create the tree that satisfy the conditions for the node.

**NumNodes**

The number of nodes  $n$  in `tree`.

**Parent**

An  $n$ -element vector `p` containing the number of the parent node for each node in `tree`, where  $n$  is the number of nodes. The parent of the root node is 0.

**PredictorNames**

A cell array of names for the predictor variables, in the order in which they appear in `X`.

**PruneAlpha**

Numeric vector with one element per pruning level. If the pruning level ranges from 0 to  $M$ , then `PruneAlpha` has  $M + 1$  elements sorted in ascending order. `PruneAlpha(1)` is for pruning level 0 (no pruning), `PruneAlpha(2)` is for pruning level 1, and so on.

**PruneList**

An  $n$ -element numeric vector with the pruning levels in each node of `tree`, where  $n$  is the number of nodes. The pruning levels range from 0 (no pruning) to  $M$ , where  $M$  is the distance between the deepest leaf and the root node.

**ResponseName**

A character vector that specifies the name of the response variable ( $Y$ ).

**ResponseTransform**

Function handle for transforming the raw response values (mean squared error). The function handle must accept a matrix of response values and return a matrix of the same size. The default 'none' means  $@(x) x$ , or no transformation.

Add or change a `ResponseTransform` function using dot notation:

```
tree.ResponseTransform = @function
```

**RowsUsed**

An  $n$ -element logical vector indicating which rows of the original predictor data ( $X$ ) were used in fitting. If the software uses all rows of  $X$ , then `RowsUsed` is an empty array (`[]`).

**SurrogateCutCategories**

An  $n$ -element cell array of the categories used for surrogate splits in `tree`, where  $n$  is the number of nodes in `tree`. For each node  $k$ , `SurrogateCutCategories{k}` is a cell

array. The length of `SurrogateCutCategories{k}` is equal to the number of surrogate predictors found at this node. Every element of `SurrogateCutCategories{k}` is either an empty character vector for a continuous surrogate predictor, or is a two-element cell array with categories for a categorical surrogate predictor. The first element of this two-element cell array lists categories assigned to the left child by this surrogate split, and the second element of this two-element cell array lists categories assigned to the right child by this surrogate split. The order of the surrogate split variables at each node is matched to the order of variables in `SurrogateCutPredictor`. The optimal-split variable at this node does not appear. For nonbranch (leaf) nodes, `SurrogateCutCategories` contains an empty cell.

### **SurrogateCutFlip**

An  $n$ -element cell array of the numeric cut assignments used for surrogate splits in `tree`, where  $n$  is the number of nodes in `tree`. For each node  $k$ , `SurrogateCutFlip{k}` is a numeric vector. The length of `SurrogateCutFlip{k}` is equal to the number of surrogate predictors found at this node. Every element of `SurrogateCutFlip{k}` is either zero for a categorical surrogate predictor, or a numeric cut assignment for a continuous surrogate predictor. The numeric cut assignment can be either  $-1$  or  $+1$ . For every surrogate split with a numeric cut  $C$  based on a continuous predictor variable  $Z$ , the left child is chosen if  $Z < C$  and the cut assignment for this surrogate split is  $+1$ , or if  $Z \geq C$  and the cut assignment for this surrogate split is  $-1$ . Similarly, the right child is chosen if  $Z \geq C$  and the cut assignment for this surrogate split is  $+1$ , or if  $Z < C$  and the cut assignment for this surrogate split is  $-1$ . The order of the surrogate split variables at each node is matched to the order of variables in `SurrogateCutPredictor`. The optimal-split variable at this node does not appear. For nonbranch (leaf) nodes, `SurrogateCutFlip` contains an empty array.

### **SurrogateCutPoint**

An  $n$ -element cell array of the numeric values used for surrogate splits in `tree`, where  $n$  is the number of nodes in `tree`. For each node  $k$ , `SurrogateCutPoint{k}` is a numeric vector. The length of `SurrogateCutPoint{k}` is equal to the number of surrogate predictors found at this node. Every element of `SurrogateCutPoint{k}` is either NaN for a categorical surrogate predictor, or a numeric cut for a continuous surrogate predictor. For every surrogate split with a numeric cut  $C$  based on a continuous predictor variable  $Z$ , the left child is chosen if  $Z < C$  and `SurrogateCutFlip` for this surrogate split is  $+1$ , or if  $Z \geq C$  and `SurrogateCutFlip` for this surrogate split is  $-1$ . Similarly, the right child is chosen if  $Z \geq C$  and `SurrogateCutFlip` for this surrogate split is  $+1$ , or if  $Z < C$  and `SurrogateCutFlip` for this surrogate split is  $-1$ . The order of the surrogate split

variables at each node is matched to the order of variables returned by `SurrCutPredictor`. The optimal-split variable at this node does not appear. For nonbranch (leaf) nodes, `SurrogateCutPoint` contains an empty cell.

### **SurrogateCutType**

An  $n$ -element cell array indicating types of surrogate splits at each node in `tree`, where  $n$  is the number of nodes in `tree`. For each node  $k$ , `SurrogateCutType{k}` is a cell array with the types of the surrogate split variables at this node. The variables are sorted by the predictive measure of association with the optimal predictor in the descending order, and only variables with the positive predictive measure are included. The order of the surrogate split variables at each node is matched to the order of variables in `SurrogateCutPredictor`. The optimal-split variable at this node does not appear. For nonbranch (leaf) nodes, `SurrogateCutType` contains an empty cell. A surrogate split type can be either 'continuous' if the cut is defined in the form  $Z < V$  for a variable  $Z$  and cut point  $V$  or 'categorical' if the cut is defined by whether  $Z$  takes a value in a set of categories.

### **SurrogateCutPredictor**

An  $n$ -element cell array of the names of the variables used for surrogate splits in each node in `tree`, where  $n$  is the number of nodes in `tree`. Every element of `SurrogateCutPredictor` is a cell array with the names of the surrogate split variables at this node. The variables are sorted by the predictive measure of association with the optimal predictor in the descending order, and only variables with the positive predictive measure are included. The optimal-split variable at this node does not appear. For nonbranch (leaf) nodes, `SurrogateCutPredictor` contains an empty cell.

### **SurrogatePredictorAssociation**

An  $n$ -element cell array of the predictive measures of association for surrogate splits in `tree`, where  $n$  is the number of nodes in `tree`. For each node  $k$ , `SurrogatePredictorAssociation{k}` is a numeric vector. The length of `SurrogatePredictorAssociation{k}` is equal to the number of surrogate predictors found at this node. Every element of `SurrogatePredictorAssociation{k}` gives the predictive measure of association between the optimal split and this surrogate split. The order of the surrogate split variables at each node is the order of variables in `SurrogateCutPredictor`. The optimal-split variable at this node does not appear. For nonbranch (leaf) nodes, `SurrogatePredictorAssociation` contains an empty cell.



**w**

The scaled `weights`, a vector with length `n`, the number of rows in `X`.

**X**

A matrix of predictor values. Each column of `X` represents one variable, and each row represents one observation.

**Y**

A numeric column vector with the same number of rows as `X`. Each entry in `Y` is the response to the data in the corresponding row of `X`.

## Methods

<code>compact</code>	Compact regression tree
<code>crossval</code>	Cross-validated decision tree
<code>cvloss</code>	Regression error by cross validation
<code>prune</code>	Produce sequence of subtrees by pruning
<code>resubLoss</code>	Regression error by resubstitution
<code>resubPredict</code>	Predict resubstitution response of tree

## Inherited Methods

<code>loss</code>	Regression error
<code>surrogateAssociation</code>	Mean predictive measure of association for surrogate splits in decision tree
<code>predict</code>	Predict responses using regression tree
<code>predictorImportance</code>	Estimates of predictor importance
<code>view</code>	View tree

## Copy Semantics

Value. To learn how value classes affect copy operations, see [Copying Objects \(MATLAB\)](#).

## Examples

### Construct a Regression Tree

Load the sample data.

```
load carsmall;
```

Construct a regression tree using the sample data.

```
tree = fitrtree([Weight, Cylinders],MPG,...  
               'categoricalpredictors',2,'MinParentSize',20,...  
               'PredictorNames',{'W','C'})
```

```
tree =
```

```
RegressionTree  
  PredictorNames: {'W' 'C'}  
  ResponseName: 'Y'  
  CategoricalPredictors: 2  
  ResponseTransform: 'none'  
  NumObservations: 94
```

Predict the mileage of 4,000-pound cars with 4, 6, and 8 cylinders.

```
mileage4K = predict(tree,[4000 4; 4000 6; 4000 8])
```

```
mileage4K =
```

```
19.2778  
19.2778  
14.3889
```

- “Decision Trees” on page 19-2

## References

- [1] Breiman, L., J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Boca Raton, FL: CRC Press, 1984.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Only the `predict` function supports code generation.
- When you train a regression tree model by using `fitrtree`, the following restrictions apply.
  - Code generation does not support categorical predictors. You cannot supply training data in a table that contains at least one categorical predictor, and you cannot use the `'CategoricalPredictors'` name-value pair argument. To dummy-code variables that you want treated as categorical, preprocess the categorical data by using `dummyvar` before fitting the model..
  - The value of the `'ResponseTransform'` name-value pair argument cannot be an anonymous function.
  - You cannot use surrogate splits, that is, the value of the `'Surrogate'` name-value pair argument must be `'off'`.

### See Also

`ClassificationTree` | `CompactRegressionTree` | `RegressionEnsemble` | `plotPartialDependence` | `predict`

### Topics

“Decision Trees” on page 19-2

**Introduced in R2011a**

# regstats

Regression diagnostics

## Syntax

```
regstats(y, X, model)
stats = regstats(...)
stats = regstats(y, X, model, whichstats)
```

## Description

`regstats(y, X, model)` performs a multilinear regression of the responses in `y` on the predictors in `X`. `X` is an  $n$ -by- $p$  matrix of  $p$  predictors at each of  $n$  observations. `y` is an  $n$ -by-1 vector of observed responses.

---

**Note** By default, `regstats` adds a first column of 1s to `X`, corresponding to a constant term in the model. Do not enter a column of 1s directly into `X`.

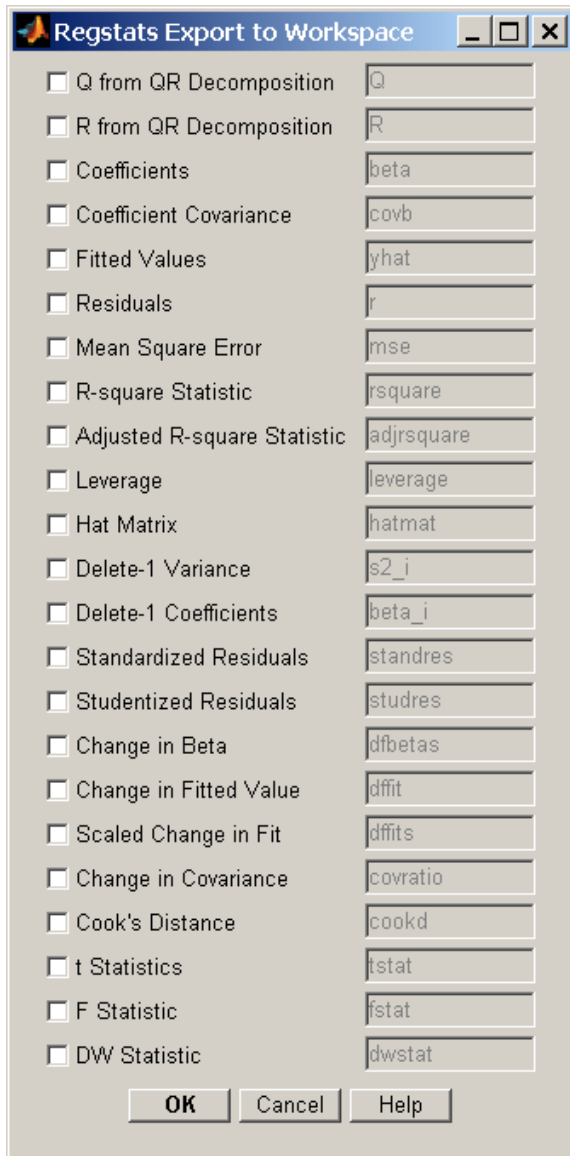
---

The optional input `model` controls the regression model. By default, `regstats` uses a linear additive model with a constant term. `model` can be any one of the following:

- 'linear' — Constant and linear terms (the default)
- 'interaction' — Constant, linear, and interaction terms
- 'quadratic' — Constant, linear, interaction, and squared terms
- 'purequadratic' — Constant, linear, and squared terms

Alternatively, `model` can be a matrix of model terms accepted by the `x2fx` function. See `x2fx` for a description of this matrix and for a description of the order in which terms appear. You can use this matrix to specify other models including ones without a constant term.

With this syntax, the function displays a graphical user interface (GUI) with a list of diagnostic statistics, as shown in the following figure.



When you select check boxes corresponding to the statistics you want to compute and click **OK**, `regstats` returns the selected statistics to the MATLAB workspace. The names of the workspace variables are displayed on the right-hand side of the interface.

You can change the name of the workspace variable to any valid MATLAB variable name.

`stats = regstats(...)` creates the structure `stats`, whose fields contain all of the diagnostic statistics for the regression. This syntax does not open the GUI. The fields of `stats` are listed in the following table.

Field	Description
<code>Q</code>	$Q$ from the $QR$ decomposition of the design matrix
<code>R</code>	$R$ from the $QR$ decomposition of the design matrix
<code>beta</code>	Regression coefficients
<code>covb</code>	Covariance of regression coefficients
<code>yhat</code>	Fitted values of the response data
<code>r</code>	Residuals
<code>mse</code>	Mean squared error
<code>rsquare</code>	$R^2$ statistic
<code>adjrsquare</code>	Adjusted $R^2$ statistic
<code>leverage</code>	Leverage
<code>hatmat</code>	Hat matrix
<code>s2_i</code>	Delete-1 variance
<code>beta_i</code>	Delete-1 coefficients
<code>standres</code>	Standardized residuals
<code>studres</code>	Studentized residuals
<code>dfbetas</code>	Scaled change in regression coefficients
<code>dffit</code>	Change in fitted values
<code>dffits</code>	Scaled change in fitted values
<code>covratio</code>	Change in covariance
<code>cookd</code>	Cook's distance
<code>tstat</code>	$t$ statistics and $p$ -values for coefficients
<code>fstat</code>	$F$ statistic and $p$ -value
<code>dwstat</code>	Durbin-Watson statistic and $p$ -value

Note that the fields names of `stats` correspond to the names of the variables returned to the MATLAB workspace when you use the GUI. For example, `stats.beta` corresponds to the variable `beta` that is returned when you select **Coefficients** in the GUI and click **OK**.

`stats = regstats(y,X,model,whichstats)` returns only the statistics that you specify in `whichstats`. `whichstats` can be a single character vector such as `'leverage'` or a cell array of character vectors such as `{'leverage' 'standres' 'studres'}`. Set `whichstats` to `'all'` to return all of the statistics.

---

**Note** The  $F$  statistic is computed under the assumption that the model contains a constant term. It is not correct for models without a constant. The  $R^2$  statistic can be negative for models without a constant, which indicates that the model is not appropriate for the data.

---

## Examples

Open the `regstats` GUI using data from `hald.mat`:

```
load hald
regstats(heat,ingredients,'linear');
```

Select **Fitted Values** and **Residuals** in the GUI:



Click **OK** to export the fitted values and residuals to the MATLAB workspace in variables named `yhat` and `r`, respectively.

You can create the same variables using the `stats` output, without opening the GUI:

```
whichstats = {'yhat','r'};
stats = regstats(heat,ingredients,'linear',whichstats);
yhat = stats.yhat;
r = stats.r;
```



## References

- [1] Belsley, D. A., E. Kuh, and R. E. Welsch. *Regression Diagnostics*. Hoboken, NJ: John Wiley & Sons, Inc., 1980.
- [2] Chatterjee, S., and A. S. Hadi. “Influential Observations, High Leverage Points, and Outliers in Linear Regression.” *Statistical Science*. Vol. 1, 1986, pp. 379–416.
- [3] Cook, R. D., and S. Weisberg. *Residuals and Influence in Regression*. New York: Chapman & Hall/CRC Press, 1983.
- [4] Goodall, C. R. “Computation Using the QR Decomposition.” *Handbook in Statistics*. Vol. 9, Amsterdam: Elsevier/North-Holland, 1993.

## See Also

`LinearModel` | `fitlm` | `stepwiselm`

## Topics

“Interpret Linear Regression Results” on page 11-64

“Linear Regression Workflow” on page 11-43

**Introduced before R2006a**

## regularize

**Class:** RegressionEnsemble

Find weights to minimize resubstitution error plus penalty term

### Syntax

```
ens1 = regularize(ens)
ens1 = regularize(ens, Name, Value)
```

### Description

`ens1 = regularize(ens)` finds optimal weights for learners in `ens` by lasso regularization. `regularize` returns a regression ensemble identical to `ens`, but with a populated `Regularization` property.

`ens1 = regularize(ens, Name, Value)` computes optimal weights with additional options specified by one or more `Name, Value` pair arguments. You can specify several name-value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### Input Arguments

**ens**

A regression ensemble, created by `fitrensemble`.

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

**lambda**

Vector of nonnegative regularization parameter values for lasso. For the default setting of `lambda`, `regularize` calculates the smallest value `lambda_max` for which all optimal weights for learners are 0. The default value of `lambda` is a vector including 0 and nine exponentially-spaced numbers from `lambda_max/1000` to `lambda_max`.

**Default:** `[0 logspace(log10(lambda_max/1000), log10(lambda_max), 9)]`

**MaxIter**

Maximum number of iterations allowed, specified as a positive integer. If the algorithm executes `MaxIter` iterations before reaching the convergence tolerance, then the function stops iterating and returns a warning message. The function can return more than one warning when either `npass` or the number of `lambda` values is greater than 1.

**Default:** `1e3`

**npass**

Maximal number of passes for lasso optimization, a positive integer.

**Default:** `10`

**reltol**

Relative tolerance on the regularized loss for lasso, a numeric positive scalar.

**Default:** `1e-3`

**verbose**

Verbosity level, either 0 or 1. When set to 1, `regularize` displays more information as it runs.

**Default:** `0`

## Output Arguments

**ens1**

A regression ensemble. Usually you set `ens1` to the same name as `ens`.

## Examples

### Regularize Ensemble of Bagged Trees

Regularize an ensemble of bagged trees.

Generate sample data.

```
rng(10, 'twister') % For reproducibility
X = rand(2000,20);
Y = repmat(-1,2000,1);
Y(sum(X(:,1:5),2)>2.5) = 1;
```

Regularize an ensemble of bagged regression trees.

```
bag = fitrensemble(X,Y, 'Method', 'Bag', 'NumLearningCycles',300);
bag = regularize(bag, 'lambda', [0.001 0.1], 'verbose', 1);
```

```
Starting lasso minimization for Lambda=0.001. Initial MSE=0.110607.
```

```
Lasso minimization completed pass 1 for Lambda=0.001
MSE = 0.0899652
Relative change in MSE = 0.229447
Number of learners with non-zero weights = 12
Lasso minimization completed pass 2 for Lambda=0.001
MSE = 0.064488
Relative change in MSE = 0.39507
Number of learners with non-zero weights = 43
Lasso minimization completed pass 3 for Lambda=0.001
MSE = 0.0608422
Relative change in MSE = 0.0599211
Number of learners with non-zero weights = 64
Lasso minimization completed pass 4 for Lambda=0.001
MSE = 0.0600689
Relative change in MSE = 0.0128732
Number of learners with non-zero weights = 82
Lasso minimization completed pass 5 for Lambda=0.001
```

```
MSE = 0.0599416
Relative change in MSE = 0.00212391
Number of learners with non-zero weights = 95
Lasso minimization completed pass 6 for Lambda=0.001
MSE = 0.0599377
Relative change in MSE = 6.56533e-05
Number of learners with non-zero weights = 108
Lasso minimization completed pass 7 for Lambda=0.001
MSE = 0.0599377
Relative change in MSE = 5.37559e-07
Number of learners with non-zero weights = 109
Lasso minimization completed pass 8 for Lambda=0.001
MSE = 0.0599376
Relative change in MSE = 4.5759e-07
Number of learners with non-zero weights = 108
Completed lasso minimization for Lambda=0.001.
Resubstitution MSE changed from 0.110607 to 0.0599376.
Number of learners reduced from 300 to 108.
Starting lasso minimization for Lambda=0.1. Initial MSE=0.110607.
Lasso minimization completed pass 1 for Lambda=0.1
MSE = 0.113013
Relative change in MSE = 0.0212885
Number of learners with non-zero weights = 10
Lasso minimization completed pass 2 for Lambda=0.1
MSE = 0.086583
Relative change in MSE = 0.30526
Number of learners with non-zero weights = 27
Lasso minimization completed pass 3 for Lambda=0.1
MSE = 0.080426
Relative change in MSE = 0.0765551
Number of learners with non-zero weights = 42
Lasso minimization completed pass 4 for Lambda=0.1
MSE = 0.0795375
Relative change in MSE = 0.0111715
Number of learners with non-zero weights = 57
Lasso minimization completed pass 5 for Lambda=0.1
MSE = 0.0792384
Relative change in MSE = 0.00377379
Number of learners with non-zero weights = 67
Lasso minimization completed pass 6 for Lambda=0.1
MSE = 0.0786909
Relative change in MSE = 0.00695864
Number of learners with non-zero weights = 75
Lasso minimization completed pass 7 for Lambda=0.1
```

```
MSE = 0.0787917
Relative change in MSE = 0.00128042
Number of learners with non-zero weights = 77
Lasso minimization completed pass 8 for Lambda=0.1
MSE = 0.0788015
Relative change in MSE = 0.000123345
Number of learners with non-zero weights = 87
Lasso minimization completed pass 9 for Lambda=0.1
MSE = 0.0788032
Relative change in MSE = 2.1627e-05
Number of learners with non-zero weights = 87
Completed lasso minimization for Lambda=0.1.
Resubstitution MSE changed from 0.110607 to 0.0788032.
Number of learners reduced from 300 to 87.
```

regularize reports on its progress.

Inspect the resulting regularization structure.

```
bag.Regularization
```

```
ans = struct with fields:
    Method: 'Lasso'
    TrainedWeights: [300x2 double]
    Lambda: [1.0000e-03 0.1000]
    ResubstitutionMSE: [0.0599 0.0788]
    CombineWeights: @classreg.learning.combiner.WeightedSum
```

Check how many learners in the regularized ensemble have positive weights. These are the learners included in a shrunken ensemble.

```
sum(bag.Regularization.TrainedWeights > 0)

ans =

    108     87
```

Shrink the ensemble using the weights from `Lambda = 0.1`.

```
cmp = shrink(bag, 'weightcolumn', 2)

cmp =
    classreg.learning.regr.CompactRegressionEnsemble
```

```

    ResponseName: 'Y'
  CategoricalPredictors: []
    ResponseTransform: 'none'
    NumTrained: 87

```

Properties, Methods

The compact ensemble contains 87 members, less than 1/3 of the original 300.

- “Ensemble Regularization” on page 18-88

## Definitions

### Lasso

The lasso algorithm finds an optimal set of learner weights  $a_t$  that minimize

$$\sum_{n=1}^N w_n g \left( \left( \sum_{t=1}^T \alpha_t h_t(x_n) \right), y_n \right) + \lambda \sum_{t=1}^T |\alpha_t|.$$

Here

- $\lambda \geq 0$  is a parameter you provide, called the lasso parameter.
- $h_t$  is a weak learner in the ensemble trained on  $N$  observations with predictors  $x_n$ , responses  $y_n$ , and weights  $w_n$ .
- $g(f,y) = (f - y)^2$  is the squared error.

### See Also

cvshrink | lasso | shrink

### Topics

“Ensemble Regularization” on page 18-88

## RegularizationValue property

**Class:** `gmdistribution`

Value of 'Regularize' parameter

### Description

The value of the parameter 'Regularize'.

---

**Note** This property applies only to `gmdistribution` objects constructed with `fitgmdist`.

---



## relieff

Importance of attributes (predictors) using ReliefF algorithm

### Syntax

```
[RANKED,WEIGHT] = relieff(X,Y,K)
[RANKED,WEIGHT] = relieff(X,Y,K,'PARAM1',val1,'PARAM2',val2,...)
```

### Description

[RANKED,WEIGHT] = `relieff(X,Y,K)` computes ranks and weights of attributes (predictors) for input data matrix `X` and response vector `Y` using the ReliefF algorithm for classification or RReliefF for regression with `K` nearest neighbors. For classification, `relieff` uses `K` nearest neighbors per class. `RANKED` are indices of columns in `X` ordered by attribute importance, meaning `RANKED(1)` is the index of the most important predictor. `WEIGHT` are attribute weights ranging from -1 to 1 with large positive weights assigned to important attributes.

If `Y` is numeric, `relieff` by default performs RReliefF analysis for regression. If `Y` is categorical, logical, a character array, or a cell array of character vectors, `relieff` by default performs ReliefF analysis for classification.

Attribute ranks and weights computed by `relieff` usually depend on `K`. If you set `K` to 1, the estimates computed by `relieff` can be unreliable for noisy data. If you set `K` to a value comparable with the number of observations (rows) in `X`, `relieff` can fail to find important attributes. You can start with `K = 10` and investigate the stability and reliability of `relieff` ranks and weights for various values of `K`.

```
[RANKED,WEIGHT] = relieff(X,Y,K,'PARAM1',val1,'PARAM2',val2,...)
```

specifies optional parameter name/value pairs.

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single

quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

**method**

Either 'regression' (default if `Y` is numeric) or 'classification' (default if `Y` is not numeric).

**prior**

Prior probabilities for each class, specified as 'empirical', 'uniform', a vector (one value for each distinct group name), or a structure `S` with two fields:

- `S.group` containing the group names as a categorical variable, character array, or cell array of character vectors
- `S.prob` containing a vector of corresponding probabilities

If the input value is 'empirical' (default), class probabilities are determined from class frequencies in `Y`. If the input value is 'uniform', all class probabilities are set equal.

**updates**

Number of observations to select at random for computing the weight of every attribute. By default all observations are used.

**categoricalx**

'on' or 'off', 'off' by default. If 'on', treat all predictors in `X` as categorical. If 'off', treat all predictors in `X` as numerical. You cannot mix numerical and categorical predictors.

**sigma**

Distance scaling factor. For observation  $i$ , influence on the attribute weight from its nearest neighbor  $j$  is multiplied by  $\exp(-\text{rank}(i,j)/\text{sigma}^2)$ , where  $\text{rank}(i,j)$  is the position of  $j$  in the list of nearest neighbors of  $i$  sorted by distance in the ascending order. Default is `Inf` (all nearest neighbors have the same influence) for classification and 50 for regression.

## Examples

### Rank Predictors by Importance

Load the sample data.

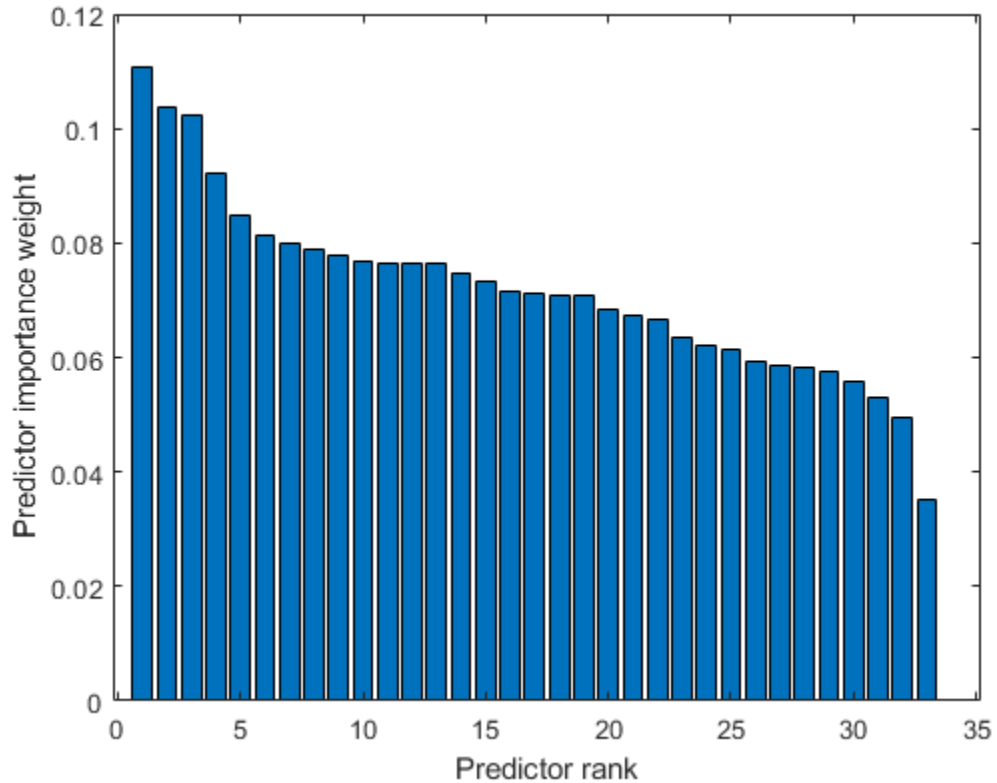
```
load ionosphere;
```

Rank the predictors based on importance.

```
[ranked,weights] = relieff(X,Y,10);
```

Create a bar plot of predictor importance weights.

```
bar(weights(ranked));  
xlabel('Predictor rank');  
ylabel('Predictor importance weight');
```



### Determine the Important Predictors

Load the sample data.

```
load fisheriris
```

Find the important predictors.

```
[ranked,weight] = relieff(meas,species,10)
```

```
ranked =
```

```
      4      3      1      2
weight =
      0.1399    0.1226    0.3590    0.3754
```

The fourth predictor is the most important, and the second predictor is the least important.

## References

- [1] Kononenko, I., Simec, E., & Robnik-Sikonja, M. (1997). *Overcoming the myopia of inductive learning algorithms with RELIEFF*. Retrieved from CiteSeerX: <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.56.4740>
- [2] Robnik-Sikonja, M., & Kononenko, I. (1997). *An adaptation of Relief for attribute estimation in regression*. Retrieved from CiteSeerX: <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.34.8381>
- [3] Robnik-Sikonja, M., & Kononenko, I. (2003). *Theoretical and empirical analysis of ReliefF and RReliefF*. *Machine Learning*, 53, 23–69.

## See Also

`knnsearch` | `pdist2` | `plotPartialDependence` | `sequentialfs`

Introduced in R2010b

## removeLearners

**Class:** CompactClassificationEnsemble

Remove members of compact classification ensemble

### Syntax

```
cens1 = removeLearners(cens, idx)
```

### Description

`cens1 = removeLearners(cens, idx)` creates a compact classification ensemble identical to `cens` only without the ensemble members in the `idx` vector.

### Input Arguments

**cens**

Compact classification ensemble, constructed with `compact`.

**idx**

Vector of positive integers with entries from 1 to `cens.NumTrained`, where `cens.NumTrained` is the number of members in `cens`. `cens1` contains all members of `cens` except those with indices in `idx`.

Typically, you set `idx = j:cens.NumTrained` for some positive integer `j`.

## Output Arguments

**cens1**

Compact classification ensemble, identical to `cens` except `cens1` does not contain those members of `cens` with indices in `idx`.

## Examples

### Remove Learners from an Ensemble

Create a compact classification ensemble. Compact it further by removing members of the ensemble.

Create a compact classification ensemble for the `ionosphere` data.

```
load ionosphere
ens = fitensemble(X,Y,'AdaBoostM1',100,'Tree');
cens = compact(ens);
```

Remove the last 50 members of the ensemble.

```
idx = cens.NumTrained-49:cens.NumTrained;
cens1 = removeLearners(cens,idx);
```

- “Classification with Imbalanced Data” on page 18-101

## Tips

- Typically, set `cens1` equal to `cens` to retain just one ensemble.
- Removing learners reduces the memory used by the ensemble and speeds up its predictions.

## See Also

`CompactClassificationEnsemble`

## **Topics**

“Classification with Imbalanced Data” on page 18-101



# removeLearners

**Class:** CompactRegressionEnsemble

Remove members of compact regression ensemble

## Syntax

```
cens1 = removeLearners(cens, idx)
```

## Description

`cens1 = removeLearners(cens, idx)` creates a compact regression ensemble identical to `cens` only without the ensemble members in the `idx` vector.

## Input Arguments

### **cens**

Compact regression ensemble, constructed with `compact`.

### **idx**

Vector of positive integers with entries from 1 to `cens.NumTrained`, where `cens.NumTrained` is the number of members in `cens`. `cens1` contains the members of `cens` except those with indices in `idx`.

Typically, you set `idx = j:cens.NumTrained` for some positive integer `j`.

## Output Arguments

**cens1**

Compact regression ensemble, identical to `cens` except `cens1` does not contain members of `cens` with indices in `idx`.

## Examples

### Remove Learners from an Ensemble

Create a compact regression ensemble. Compact it further by removing members of the ensemble.

Create a compact regression ensemble for the `carsmall` data.

```
load carsmall
X = [Weight Cylinders];
ens = fitensemble(X,MPG,'LSBoost',100,'Tree','categorical',2);
cens = compact(ens);
```

Remove the last 50 members of the ensemble.

```
idx = cens.NumTrained-49:cens.NumTrained;
cens1 = removeLearners(cens,idx);
```

## Tips

- Typically, set `cens1` equal to `cens` to retain just one ensemble.
- Removing learners reduces the memory used by the ensemble and speeds up its predictions.

## See Also

`CompactRegressionEnsemble`

# removeTerms

**Class:** GeneralizedLinearModel

Remove terms from generalized linear model

## Syntax

```
mdl1 = removeTerms(mdl, terms)
```

## Description

`mdl1 = removeTerms(mdl, terms)` returns a linear model the same as `mdl` but with fewer terms.

## Input Arguments

**mdl**

Generalized linear model, as constructed by `fitglm` or `stepwiseglm`.

**terms**

Terms to remove from the `mdl` regression model. Specify as either a:

- Text representing one or more terms to remove. For details, see “Wilkinson Notation” on page 32-5623.
- Row or rows in the terms matrix (see `modelspec` in `fitglm`). For example, if there are three variables A, B, and C:

```
[0 0 0] represents a constant term or intercept  
[0 1 0] represents B; equivalently, A^0 * B^1 * C^0  
[1 0 1] represents A*C  
[2 0 0] represents A^2  
[0 1 2] represents B*(C^2)
```

## Output Arguments

**mdl1**

Generalized linear model, the same as `mdl` but without the terms given in `terms`. You can set `mdl1` equal to `mdl` to overwrite `mdl`.

## Examples

### Remove a Term from a Generalized Linear Regression Model

This example makes a model using two predictors, then removes one.

Generate artificial data for the model, Poisson random numbers with two underlying predictors `X(1)` and `X(2)`.

```
rng('default') % for reproducibility
rndvars = randn(100,2);
X = [2+rndvars(:,1),rndvars(:,2)];
mu = exp(1 + X*[1;2]);
y = poissrnd(mu);
```

Create a generalized linear regression model of Poisson data.

```
mdl = fitglm(X,y,'y ~ x1 + x2','distr','poisson')
```

```
mdl =
```

Generalized linear regression model:

```
log(y) ~ 1 + x1 + x2
Distribution = Poisson
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
	-----	-----	-----	-----
(Intercept)	1.0405	0.022122	47.034	0
x1	0.9968	0.003362	296.49	0
x2	1.987	0.0063433	313.24	0

```

100 observations, 97 error degrees of freedom
Dispersion: 1
Chi^2-statistic vs. constant model: 2.95e+05, p-value = 0

```

Remove the second predictor from the model.

```
mdl1 = removeTerms(mdl, 'x2')
```

```
mdl1 =
```

```

Generalized linear regression model:
log(y) ~ 1 + x1
Distribution = Poisson

```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	2.7784	0.014043	197.85	0
x1	1.1732	0.0033653	348.6	0

```

100 observations, 98 error degrees of freedom
Dispersion: 1
Chi^2-statistic vs. constant model: 1.25e+05, p-value = 0

```

- “Plots to Understand Predictor Effects and How to Modify a Model” on page 12-31
- “Generalized Linear Model Workflow” on page 12-41

## Definitions

### Wilkinson Notation

Wilkinson notation describes the factors present in models. The notation relates to factors present in models, not to the multipliers (coefficients) of those factors.

Wilkinson Notation	Factors in Standard Notation
1	Constant (intercept) term
$A^k$ , where $k$ is a positive integer	$A, A^2, \dots, A^k$
$A + B$	$A, B$
$A*B$	$A, B, A*B$
$A:B$	$A*B$ only
$-B$	Do not include $B$
$A*B + C$	$A, B, C, A*B$
$A + B + C + A:B$	$A, B, C, A*B$
$A*B*C - A:B:C$	$A, B, C, A*B, A*C, B*C$
$A*(B + C)$	$A, B, C, A*B, A*C$

Statistics and Machine Learning Toolbox notation always includes a constant term unless you explicitly remove the term using  $-1$ .

For details, see Wilkinson and Rogers [1].

## Alternatives

`step` adds or removes terms from a model using a greedy one-step algorithm.

## References

[1] Wilkinson, G. N., and C. E. Rogers. *Symbolic description of factorial models for analysis of variance*. *J. Royal Statistics Society* 22, pp. 392–399, 1973.

## See Also

`GeneralizedLinearModel` | `addTerms` | `step` | `stepwiseglm`

## Topics

“Plots to Understand Predictor Effects and How to Modify a Model” on page 12-31

“Generalized Linear Model Workflow” on page 12-41  
“Generalized Linear Models” on page 12-12

## removeTerms

**Class:** LinearModel

Remove terms from linear model

### Syntax

```
mdl1 = removeTerms(mdl, terms)
```

### Description

`mdl1 = removeTerms(mdl, terms)` returns a linear regression model `mdl1` that is the same as the input model `mdl`, but with terms removed as specified by `terms`.

### Input Arguments

**mdl** — Full, fitted linear regression model

LinearModel object

Full, fitted linear regression model, specified as a LinearModel object constructed using `fitlm` or `stepwiselm`.

**terms** — Terms to remove from regression model

formula | matrix

Terms to remove from the regression model `mdl`, specified as one of the following:

- Formula representing one or more terms to remove. For details, see Wilkinson Notation on page 32-5629.
- Row or rows in the terms matrix (see the `modelspec` argument description in the fitting function `fitlm`). For example, if there are three variables A, B, and C:

```
[0 0 0] represents a constant term or intercept  
[0 1 0] represents B; equivalently, A^0 * B^1 * C^0
```



```
[1 0 1] represents A*C
[2 0 0] represents A^2
[0 1 2] represents B*(C^2)
```

## Output Arguments

### mdl1 — Regression model with fewer terms

LinearModel object

Regression model with fewer terms, returned as a `LinearModel` object. `mdl1` is the same as `mdl` but with `terms` removed. To overwrite `mdl`, set `mdl1` equal to `mdl`.

## Examples

### Remove Terms from a Linear Regression Model

Construct a default linear model of the Hald data. Remove terms with high  $P$ -values.

Load the data.

```
load hald
X = ingredients; % predictor variables
y = heat; % response
```

Fit a default linear model to the data.

```
mdl = fitlm(X,y)
```

```
mdl =
```

```
Linear regression model:
  y ~ 1 + x1 + x2 + x3 + x4
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
	-----	-----	-----	-----
(Intercept)	62.405	70.071	0.8906	0.39913

x1	1.5511	0.74477	2.0827	0.070822
x2	0.51017	0.72379	0.70486	0.5009
x3	0.10191	0.75471	0.13503	0.89592
x4	-0.14406	0.70905	-0.20317	0.84407

Number of observations: 13, Error degrees of freedom: 8  
Root Mean Squared Error: 2.45  
R-squared: 0.982, Adjusted R-Squared 0.974  
F-statistic vs. constant model: 111, p-value = 4.76e-07

Remove the x3 and x4 terms because their *P*-values are so high.

```
terms = 'x3 + x4'; % terms to remove  
mdl1 = removeTerms(mdl, terms)
```

```
mdl1 =
```

```
Linear regression model:  
y ~ 1 + x1 + x2
```

```
Estimated Coefficients:
```

	Estimate	SE	tStat	pValue
(Intercept)	52.577	2.2862	22.998	5.4566e-10
x1	1.4683	0.1213	12.105	2.6922e-07
x2	0.66225	0.045855	14.442	5.029e-08

Number of observations: 13, Error degrees of freedom: 10  
Root Mean Squared Error: 2.41  
R-squared: 0.979, Adjusted R-Squared 0.974  
F-statistic vs. constant model: 230, p-value = 4.41e-09

The new model has the same adjusted R-Squared value (0.974) as the previous model, meaning it is about as good a fit. All the terms in the new model have extremely low *P*-values.

- “Change Models” on page 11-36

## Definitions

### Wilkinson Notation

Wilkinson notation describes the factors present in models. The notation relates to factors present in models, not to the multipliers (coefficients) of those factors.

Wilkinson Notation	Factors in Standard Notation
1	Constant (intercept) term
$A^k$ , where $k$ is a positive integer	$A, A^2, \dots, A^k$
$A + B$	$A, B$
$A*B$	$A, B, A*B$
$A:B$	$A*B$ only
$-B$	Do not include $B$
$A*B + C$	$A, B, C, A*B$
$A + B + C + A:B$	$A, B, C, A*B$
$A*B*C - A:B:C$	$A, B, C, A*B, A*C, B*C$
$A*(B + C)$	$A, B, C, A*B, A*C$

Statistics and Machine Learning Toolbox notation always includes a constant term unless you explicitly remove the term using `-1`.

For details, see Wilkinson and Rogers [1].

## Alternatives

Use `stepwiselm` to select a model from a starting model, continuing until no single step is beneficial.

Use `addTerms` to add particular terms.

Use `step` to optimally improve the model by adding or removing terms.

## References

- [1] Wilkinson, G. N., and C. E. Rogers. *Symbolic description of factorial models for analysis of variance*. J. Royal Statistics Society 22, pp. 392–399, 1973.

## See Also

`LinearModel` | `addTerms` | `stepwiselm`

## Topics

“Change Models” on page 11-36

“Linear Regression” on page 11-11

# reorderlevels

Reorder levels of nominal or ordinal arrays

---

**Note** The `nominal` and `ordinal` array data types might be removed in a future release. To represent ordered and unordered discrete, nonnumeric data, use the MATLAB® `categorical` data type instead.

---

## Syntax

```
B = reorderlevels(A,newlevels)
```

## Description

`B = reorderlevels(A,newlevels)` returns a nominal or ordinal array object the same as `A` but with levels in the new order specified by `newlevels`.

For ordinal arrays, the order of the levels has significance for relational operators, minimum and maximum, and for sorting.

## Examples

- “Reorder Category Levels” on page 2-12
- “Sort Ordinal Arrays” on page 2-46

## Input Arguments

**A** — Nominal or ordinal array

`nominal array` | `ordinal array`

Nominal or ordinal array, specified as a `nominal` or `ordinal` array object created using `nominal` or `ordinal`.

**newlevels** — **New order of levels**

cell array of character vectors | 2-D character matrix

New order of levels, specified as a cell array of character vectors or 2-D character matrix. `newlevels` must be a reordering of the labels returned by `getlabels`.

Data Types: `char` | `cell`

## Output Arguments

**B** — **Nominal or ordinal array**

nominal array | ordinal array

Nominal or ordinal array, returned as a nominal or ordinal array object.

## See Also

`addlevels` | `droplevels` | `getlabels` | `nominal` | `ordinal` | `reorderlevels`

## Topics

“Reorder Category Levels” on page 2-12

“Sort Ordinal Arrays” on page 2-46

`nominal`

`ordinal`

**Introduced in R2007a**

# repartition

**Class:** cvpartition

Repartition data for cross-validation

## Syntax

```
cnew = repartition(c)
```

## Description

`cnew = repartition(c)` constructs an object `cnew` of the `cvpartition` on page 32-1292 class defining a random partition of the same type as `c`, where `c` is also an object of the `cvpartition` on page 32-1292 class.

Repartitioning is useful for Monte-Carlo repetitions of cross-validation analyses. `repartition` is called by `crossval` when the 'mcreps' parameter is specified.

## Examples

Partition and repartition 100 observations for 3-fold cross-validation:

```
c = cvpartition(100,'kfold',3)
c =
K-fold cross validation partition
      N: 100
  NumTestSets: 3
   TrainSize: 67  66  67
   TestSize: 33  34  33

cnew = repartition(c)
cnew =
K-fold cross validation partition
      N: 100
  NumTestSets: 3
```

```
TrainSize: 67 66 67
TestSize: 33 34 33
```

Check for equality of the test data in the first fold:

```
isequal(test(c,1),test(cnew,1))
ans =
    0
```

## See Also

`cvpartition`



# RepeatedMeasuresModel class

Repeated measures model class

## Description

A `RepeatedMeasuresModel` object represents a model fitted to data with multiple measurements per subject. The object comprises data, fitted coefficients, covariance parameters, design matrix, error degrees of freedom, and between- and within-subjects factor names for a repeated measures model. You can predict model responses using the `predict` method and generate random data at new design points using the `random` method.

## Construction

You can fit a repeated measures model using `fitrm(t,modelspec)`.

## Input Arguments

### `t` — Input data

table

Input data, which includes the values of the response variables and the between-subject factors to use as predictors in the repeated measures model, specified as a table.

Data Types: table

### `modelspec` — Formula for model specification

character vector of the form 'y1-yk ~ terms'

Formula for model specification, specified as a character vector of the form 'y1-yk ~ terms'. Specify the terms using Wilkinson notation. `fitrm` treats the variables used in model terms as categorical if they are categorical (nominal or ordinal), logical, character arrays, or a cell array of character vectors.

Example: 'y1-y4 ~ x1 + x2 \* x3'

## Properties

### **BetweenDesign** — Design for between-subject factors

table

Design for between-subject factors and values of repeated measures, stored as a table.

Data Types: `table`

### **BetweenModel** — Model for between-subjects factors

character vector

Model for between-subjects factors, stored as a character vector. This character vector is the text representation to the right of the tilde in the model specification you provide when fitting the repeated measures model using `fitrm`.

Data Types: `char`

### **BetweenFactorNames** — Names of variables used as between-subject factors

cell array of character vectors

Names of variables used as between-subject factors in the repeated measures model, `rm`, stored as a cell array of character vectors.

Data Types: `cell`

### **ResponseNames** — Names of variables used as response variables

cell array of character vectors

Names of variables used as response variables in the repeated measures model, `rm`, stored as a cell array of character vectors.

Data Types: `cell`

### **WithinDesign** — Values of within-subject factors

table

Values of the within-subject factors, stored as a table.

Data Types: `table`

### **WithinModel** — Model for within-subjects factors

character vector

Model for within-subjects factors, stored as a character vector.

Data Types: `char`

**WithinFactorNames** — Names of within-subject factors

cell array of character vectors

Names of the within-subject factors, stored as a cell array of character vectors.

Data Types: `cell`

**Coefficients** — Values of estimated coefficients

table

Values of the estimated coefficients for fitting the repeated measures as a function of the terms in the between-subjects model, stored as a table.

`fitrm` defines the coefficients for a categorical term using 'effects' coding, which means coefficients sum to 0. There is one coefficient for each level except the first. The implied coefficient for the first level is the sum of the other coefficients for the term.

You can display the coefficient values as a matrix rather than a table using `coef = r.Coefficients{:, :}`.

You can display marginal means for all levels using the `margmean` method.

Data Types: `table`

**Covariance** — Estimated response covariances

table

Estimated response covariances, that is, covariance of the repeated measures, stored as a table. `fitrm` computes the covariances around the mean returned by the fitted repeated measures model `rm`.

You can display the covariance values as a matrix rather than a table using `coef = r.Covariance{:, :}`.

Data Types: `table`

**DfE** — Error degrees of freedom

scalar value

Error degrees of freedom, stored as a scalar value. `DFE` is the number of observations minus the number of estimated coefficients in the between-subjects model.

Data Types: `double`

## Methods

<code>anova</code>	Analysis of variance for between-subject effects
<code>epsilon</code>	Epsilon adjustment for repeated measures anova
<code>grpstats</code>	Compute descriptive statistics of repeated measures data by group
<code>manova</code>	Multivariate analysis of variance
<code>margmean</code>	Estimate marginal means
<code>mauchly</code>	Mauchly's test for sphericity
<code>multcompare</code>	Multiple comparison of estimated marginal means
<code>plot</code>	Plot data with optional grouping
<code>plotprofile</code>	Plot expected marginal means with optional grouping
<code>predict</code>	Compute predicted values given predictor values
<code>random</code>	Generate new random response values given predictor values
<code>ranova</code>	Repeated measures analysis of variance

## Examples

### Fit a Repeated Measures Model

Load the sample data.

```
load fisheriris
```

The column vector, `species`, consists of iris flowers of three different species: `setosa`, `versicolor`, `virginica`. The double matrix `meas` consists of four types of measurements on the flowers: the length and width of sepals and petals in centimeters, respectively.

Store the data in a table array.

```
t = table(species,meas(:,1),meas(:,2),meas(:,3),meas(:,4),...
'VariableNames',{'species','meas1','meas2','meas3','meas4'});
Meas = table([1 2 3 4'],'VariableNames',{'Measurements'});
```

Fit a repeated measures model, where the measurements are the responses and the species is the predictor variable.

```
rm = fitrm(t,'meas1-meas4~species','WithinDesign',Meas)
```

```
rm =
```

```
RepeatedMeasuresModel with properties:
```

```
Between Subjects:
```

```
    BetweenDesign: [150x5 table]
```

```
    ResponseNames: {'meas1' 'meas2' 'meas3' 'meas4'}
```

```
    BetweenFactorNames: {'species'}
```

```
    BetweenModel: '1 + species'
```

```
Within Subjects:
```

```
    WithinDesign: [4x1 table]
```

```
    WithinFactorNames: {'Measurements'}
```

```
    WithinModel: 'separatemeans'
```

```
Estimates:
```

```
    Coefficients: [3x4 table]
```

```
    Covariance: [4x4 table]
```

Display the coefficients.

```
rm.Coefficients
```

```
ans =
```

```
3x4 table
```

	meas1	meas2	meas3	meas4
(Intercept)	5.8433	3.0573	3.758	1.1993
species_setosa	-0.83733	0.37067	-2.296	-0.95333
species_versicolor	0.092667	-0.28733	0.502	0.12667

`fitrm` uses the 'effects' contrasts, which means that the coefficients sum to 0. The `rm.DesignMatrix` has one column of 1s for the intercept, and two other columns `species_setosa` and `species_versicolor`, which are as follows:

$$\text{species\_setosa} = \begin{cases} 1, & \text{if } \textit{setosa} \\ 0, & \text{if } \textit{versicolor} \\ -1, & \text{if } \textit{virginica} \end{cases}$$

and

$$\text{species\_versicolor} = \begin{cases} 0, & \text{if } \textit{setosa} \\ 1, & \text{if } \textit{versicolor} \\ -1, & \text{if } \textit{virginica} \end{cases} .$$

Display the covariance matrix.

```
rm.Covariance
```

```
ans =
```

```
4x4 table
```

	meas1	meas2	meas3	meas4
meas1	0.26501	0.092721	0.16751	0.038401
meas2	0.092721	0.11539	0.055244	0.03271
meas3	0.16751	0.055244	0.18519	0.042665
meas4	0.038401	0.03271	0.042665	0.041882

Display the error degrees of freedom.

```
rm.DFE
```

```
ans =
```

```
147
```

The error degrees of freedom is the number of observations minus the number of estimated coefficients in the between-subjects model, e.g.  $150 - 3 = 147$ .

## Definitions

### Wilkinson Notation

Wilkinson notation describes the factors present in models. It does not describe the multipliers (coefficients) of those factors.

Use these rules to specify the responses in `modelspec`.

Wilkinson Notation	Description
Y1, Y2, Y3	Specific list of variables
Y1-Y5	All table variables from Y1 through Y5

Use these rules to specify terms in `modelspec`.

Wilkinson Notation	Factors in Standard Notation
1	Constant (intercept) term
$X^k$ , where $k$ is a positive integer	$X, X^2, \dots, X^k$
$X1 + X2$	$X1, X2$
$X1 * X2$	$X1, X2, X1 * X2$
$X1 : X2$	$X1 * X2$ only
$-X2$	Do not include $X2$
$X1 * X2 + X3$	$X1, X2, X3, X1 * X2$
$X1 + X2 + X3 + X1 : X2$	$X1, X2, X3, X1 * X2$
$X1 * X2 * X3 - X1 : X2 : X3$	$X1, X2, X3, X1 * X2, X1 * X3, X2 * X3$
$X1 * (X2 + X3)$	$X1, X2, X3, X1 * X2, X1 * X3$

Statistics and Machine Learning Toolbox notation always includes a constant term unless you explicitly remove the term using `-1`.

## See Also

`fitrm`

## Topics

Class Attributes (MATLAB)

Property Attributes (MATLAB)



# replacedata

**Class:** dataset

Replace dataset variables

---

**Note** The `dataset` data type might be removed in a future release. To work with heterogeneous data, use the MATLAB® `table` data type instead. See MATLAB `table` documentation for more information.

---

## Syntax

```
B = replacedata(A,X)
B = replacedata(A,X,vars)
B = replacedata(A,fun)
B = replacedata(A,fun,vars)
```

## Description

`B = replacedata(A,X)` creates a dataset array `B` with the same variables as the dataset array `A`, but with the data for those variables replaced by the data in the array `X`. `replacedata` creates each variable in `B` using one or more columns from `X`, in order. `X` must have as many columns as the total number of columns in all of the variables in `A`, and as many rows as `A` has observations.

`B = replacedata(A,X,vars)` creates a dataset array `B` with the same variables as the dataset array `A`, but with the data for the variables specified in `vars` replaced by the data in the array `X`. The remaining variables in `B` are copies of the corresponding variables in `A`. `vars` is a positive integer, a vector of positive integers, a variable name, a cell array containing one or more variable names, or a logical vector. Each variable in `B` has as many columns as the corresponding variable in `A`. `X` must have as many columns as the total number of columns in all the variables specified in `vars`.

`B = replacedata(A,fun)` or `B = replacedata(A,fun,vars)` creates a dataset array `B` by applying the function `fun` to the values in `A`'s variables. `replacedata` first

horizontally concatenates A's variables into a single array, then applies the function `fun`. The specified variables in A must have types and sizes compatible with the concatenation. `fun` is a function handle that accepts a single input array and returns an array with the same number of rows and columns as the input.

## Examples

```
data = dataset({rand(3,3),'Var1','Var2','Var3'})

% Use ZSCORE to normalize each variable in a dataset array
% separately, by explicitly extracting and transforming the
% data, and then replacing it.
X = double(data);
X = zscore(X);
data = replacedata(data,X)

% Equivalently, provide a handle to ZSCORE.
data = replacedata(data,@zscore)

% Use ZSCORE to normalize each observation in a dataset
% array separately by creating an anonymous function.
data = replacedata(data,@(x) zscore(x,[],2))
```

## See Also

`dataset`

# replaceWithMissing

**Class:** dataset

Insert missing data indicators into a dataset array

---

**Note** The `dataset` data type might be removed in a future release. To work with heterogeneous data, use the MATLAB® `table` data type instead. See MATLAB `table` documentation for more information.

---

## Syntax

```
ds2 = replaceWithMissing(ds,Name,Value)
```

## Description

`ds2 = replaceWithMissing(ds,Name,Value)` replaces specified values in a dataset array with standard missing data indicators using options specified by one or more `Name,Value` pair arguments. Use `replaceWithMissing` to specify:

- Which numeric missing value indicators to replace with `NaN`.
- Which character missing value indicators to replace with `' '`.
- Which categorical levels to replace with `<undefined>`.

## Input Arguments

**ds**

dataset array.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single

quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

**NumericValues**

Vector of numeric values that `replaceWithMissing` replaces with `NaN`.

**CategoricalLevels**

Character vector or cell array of character vectors naming the categorical levels that `replaceWithMissing` replaces with `<undefined>`.

**Strings**

Character vector or cell array of character vectors containing the text that `replaceWithMissing` replaces with ' '.

**DataVars**

Specified set of variables in `ds` for which `replaceWithMissing` replaces values. You can specify a positive integer or vector of positive integers indicating the variable column numbers, a variable name or a cell array of variables names, or a logical vector indicating which variables to replace missing values in.

**Default:** All variables in `ds`.

## Output Arguments

**ds2**

dataset array that has the specified missing value indicators, in the specified variables of `ds`, replaced with standard missing value indicators.

## Examples

**Replace Nonstandard Missing Value Indicators**

Replace nonstandard missing value indicators with standard missing value indicators.

Replace numeric missing values coded 99 with NaN, and character missing values coded '.' with ''.

```
ds = replaceWithMissing(ds, 'NumericValues', 99, 'Strings', '.');
```

- “Clean Messy and Missing Data” on page 2-123

## See Also

dataset | ismissing

## Topics

“Clean Messy and Missing Data” on page 2-123

“Dataset Arrays” on page 2-142

## reset

**Class:** grandstream

Reset state

## Syntax

```
reset(q)
```

## Description

`reset(q)` resets the state of the quasi-random number stream `q` of the `grandstream` on page 32-5248 class back to its initial state, 1. Subsequent points drawn from the stream will be the same as those drawn from a new stream. The command is equivalent to `q.State = 1`.

## Examples

Use `grandstream` to construct a 3-D Halton stream, based on a point set that skips the first 1000 values and then retains every 101st point:

```
q = grandstream('halton',3,'Skip',1e3,'Leap',1e2)
q =
  Halton quasi-random stream in 3 dimensions
  Point set properties:
      Skip : 1000
      Leap : 100
  ScrambleMethod : none

nextIdx = q.State
nextIdx =
     1
```

Use `grand` to generate two samples of size four:

```
X1 = grand(q,4)
X1 =
    0.0928    0.3475    0.0051
    0.6958    0.2035    0.2371
    0.3013    0.8496    0.4307
    0.9087    0.5629    0.6166
nextIdx = q.State
nextIdx =
     5
```

```
X2 = grand(q,4)
X2 =
    0.2446    0.0238    0.8102
    0.5298    0.7540    0.0438
    0.3843    0.5112    0.2758
    0.8335    0.2245    0.4694
nextIdx = q.State
nextIdx =
     9
```

Use `reset` to reset the stream, then generate another sample:

```
reset(q)
nextIdx = q.State
nextIdx =
     1
```

```
X = grand(q,4)
X =
    0.0928    0.3475    0.0051
    0.6958    0.2035    0.2371
    0.3013    0.8496    0.4307
    0.9087    0.5629    0.6166
```

## See Also

`grand` | `grandstream`

## residuals

**Class:** `GeneralizedLinearMixedModel`

Residuals of fitted generalized linear mixed-effects model

### Syntax

```
r = residuals(glme)
r = residuals(glme, Name, Value)
```

### Description

`r = residuals(glme)` returns the raw conditional residuals from a fitted generalized linear mixed-effects model `glme`.

`r = residuals(glme, Name, Value)` returns the residuals using additional options specified by one or more `Name, Value` pair arguments. For example, you can specify to return Pearson residuals for the model.

### Input Arguments

**glme** — Generalized linear mixed-effects model

`GeneralizedLinearMixedModel` object

Generalized linear mixed-effects model, specified as a `GeneralizedLinearMixedModel` object. For properties and methods of this object, see `GeneralizedLinearMixedModel`.

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.



**Conditional** — Indicator for conditional residuals

true (default) | false

Indicator for conditional residuals, specified as the comma-separated pair consisting of 'Conditional' and one of the following.

Value	Description
true	Contributions from both fixed effects and random effects (conditional)
false	Contribution from only fixed effects (marginal)

Conditional residuals include contributions from both fixed- and random-effects predictors. Marginal residuals include contribution from only fixed effects. To obtain marginal residual values, `residuals` computes the conditional mean of the response with the empirical Bayes predictor vector of random effects,  $b$ , set to 0.

Example: 'Conditional', false

**ResidualType** — Residual type

'raw' (default) | 'Pearson'

Residual type, specified as the comma-separated pair consisting of 'ResidualType' and one of the following.

Residual Type	Conditional	Marginal
'raw'	$r_{ci} = y_i - g^{-1}(x_i^T \hat{\beta} + z_i^T \hat{b} + \delta_i)$	$r_{mi} = y_i - g^{-1}(x_i^T \hat{\beta} + \delta_i)$
'Pearson'	$r_{ci}^{pearson} = \frac{r_{ci}}{\sqrt{\widehat{\sigma^2} v_i(\mu_i(\hat{\beta}, \hat{b}))}}$	$r_{mi}^{pearson} = \frac{r_{mi}}{\sqrt{\widehat{\sigma^2} v_i(\mu_i(\hat{\beta}, 0))}}$

In each of these equations:

- $y_i$  is the  $i$ th element of the  $n$ -by-1 response vector,  $y$ , where  $i = 1, \dots, n$ .
- $g^{-1}$  is the inverse link function for the model.
- $x_i^T$  is the  $i$ th row of the fixed-effects design matrix  $X$ .
- $z_i^T$  is the  $i$ th row of the random-effects design matrix  $Z$ .

- $\delta_i$  is the  $i$ th offset value.
- $\sigma^2$  is the dispersion parameter.
- $w_i$  is the  $i$ th observation weight.
- $v_i$  is the variance term for the  $i$ th observation.
- $\mu_i$  is the mean of the response for the  $i$ th observation.
- $\hat{\beta}$  and  $\hat{b}$  are estimated values of  $\beta$  and  $b$ .

Raw residuals from a generalized linear mixed-effects model have nonconstant variance. Pearson residuals are expected to have an approximately constant variance, and are generally used for analysis.

Example: `'ResidualType', 'Pearson'`

## Output Arguments

### **r** — Residuals

$n$ -by-1 vector

Residuals of the fitted generalized linear mixed-effects model `glme` returned as an  $n$ -by-1 vector, where  $n$  is the number of observations.

## Examples

### Plot Residuals Versus Fitted Values

Load the sample data.

```
load mfr
```

This simulated data is from a manufacturing company that operates 50 factories across the world, with each factory running a batch process to create a finished product. The company wants to decrease the number of defects in each batch, so it developed a new manufacturing process. To test the effectiveness of the new process, the company selected 20 of its factories at random to participate in an experiment: Ten factories implemented the new process, while the other ten continued to run the old process. In each of the 20

factories, the company ran five batches (for a total of 100 batches) and recorded the following data:

- Flag to indicate whether the batch used the new process (`newprocess`)
- Processing time for each batch, in hours (`time`)
- Temperature of the batch, in degrees Celsius (`temp`)
- Categorical variable indicating the supplier (A, B, or C) of the chemical used in the batch (`supplier`)
- Number of defects in the batch (`defects`)

The data also includes `time_dev` and `temp_dev`, which represent the absolute deviation of time and temperature, respectively, from the process standard of 3 hours at 20 degrees Celsius.

Fit a generalized linear mixed-effects model using `newprocess`, `time_dev`, `temp_dev`, and `supplier` as fixed-effects predictors. Include a random-effects term for intercept grouped by `factory`, to account for quality differences that might exist due to factory-specific variations. The response variable `defects` has a Poisson distribution, and the appropriate link function for this model is `log`. Use the Laplace fit method to estimate the coefficients. Specify the dummy variable encoding as `'effects'`, so the dummy variable coefficients sum to 0.

The number of defects can be modeled using a Poisson distribution

$$\text{defects}_{ij} \sim \text{Poisson}(\mu_{ij})$$

This corresponds to the generalized linear mixed-effects model

$$\log(\mu_{ij}) = \beta_0 + \beta_1 \text{newprocess}_{ij} + \beta_2 \text{time.dev}_{ij} + \beta_3 \text{temp.dev}_{ij} + \beta_4 \text{supplier.C}_{ij} + \beta_5 \text{supplier.B}_{ij} + b_i,$$

where

- `defectsij` is the number of defects observed in the batch produced by factory *i* during batch *j*.
- `μij` is the mean number of defects corresponding to factory *i* (where *i* = 1, 2, ..., 20) during batch *j* (where *j* = 1, 2, ..., 5).
- `newprocessij`, `time.devij`, and `temp.devij` are the measurements for each variable that correspond to factory *i* during batch *j*. For example, `newprocessij` indicates whether the batch produced by factory *i* during batch *j* used the new process.

- `supplier_Cij` and `supplier_Bij` are dummy variables that use effects (sum-to-zero) coding to indicate whether company C or B, respectively, supplied the process chemicals for the batch produced by factory  $i$  during batch  $j$ .
- $b_i \sim N(0, \sigma_b^2)$  is a random-effects intercept for each factory  $i$  that accounts for factory-specific variation in quality.

```
glme = fitglme(mfr, 'defects ~ 1 + newprocess + time_dev + temp_dev + supplier + (1|facto
  'Distribution', 'Poisson', 'Link', 'log', 'FitMethod', 'Laplace', 'DummyVarCoding', 'effec
```

Generate the conditional Pearson residuals and the conditional fitted values from the model.

```
r = residuals(glme, 'ResidualType', 'Pearson');
mufit = fitted(glme);
```

Display the first ten rows of the Pearson residuals.

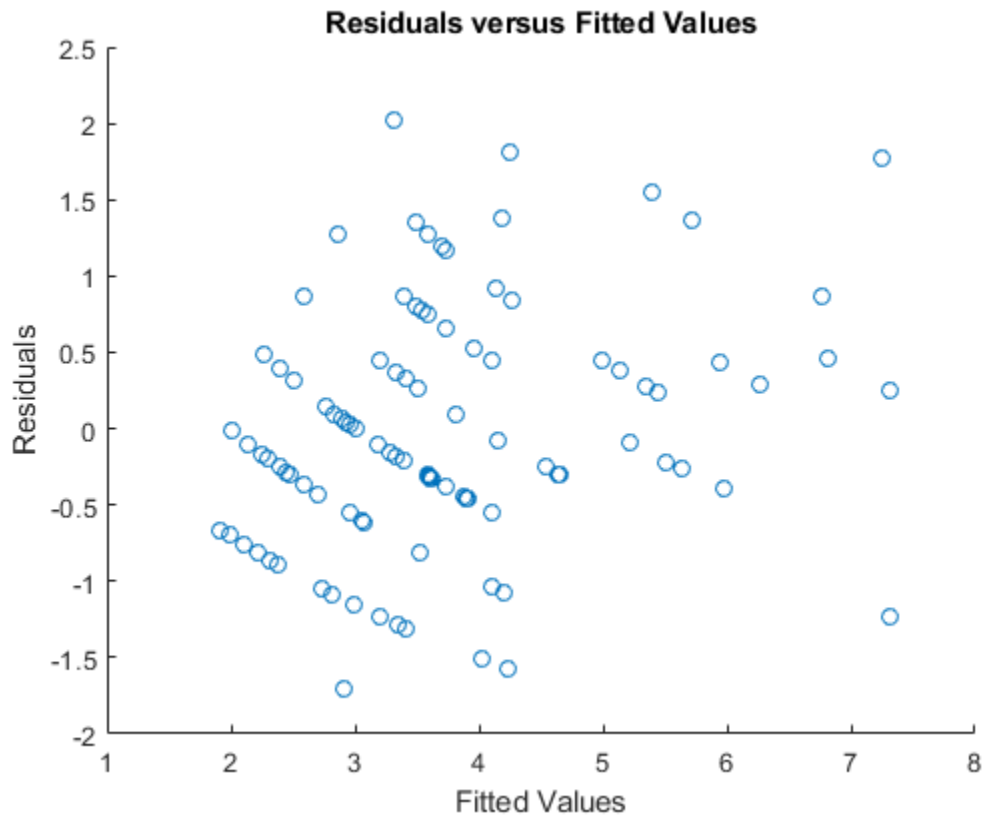
```
r(1:10)
```

```
ans =

    0.4530
    0.4339
    0.3833
   -0.2653
    0.2811
   -0.0935
   -0.2984
   -0.2509
    1.5547
   -0.3027
```

Plot the Pearson residuals versus the fitted values, to check for signs of nonconstant variance among the residuals (heteroscedasticity).

```
figure
scatter(mufit, r)
title('Residuals versus Fitted Values')
xlabel('Fitted Values')
ylabel('Residuals')
```



The plot does not show a systematic dependence on the fitted values, so there are no signs of nonconstant variance among the residuals.

## See Also

`GeneralizedLinearMixedModel` | `designMatrix` | `fitted` | `response`

## residuals

**Class:** LinearMixedModel

Residuals of fitted linear mixed-effects model

### Syntax

```
R = residuals(lme)
R = residuals(lme, Name, Value)
```

### Description

`R = residuals(lme)` returns the raw conditional residuals from a fitted linear mixed-effects model `lme`.

`R = residuals(lme, Name, Value)` returns the residuals from the linear mixed-effects model `lme` with additional options specified by one or more `Name, Value` pair arguments.

For example, you can specify Pearson or standardized residuals, or residuals with contributions from only fixed effects.

### Input Arguments

**lme** — Linear mixed-effects model

LinearMixedModel object

Linear mixed-effects model, specified as a LinearMixedModel object constructed using `fitlme` or `fitlmematrix`.

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single

quotes (' '). You can specify several name and value pair arguments in any order as Name1, Value1, . . . , NameN, ValueN.

### Conditional — Indicator for conditional residuals

True (default) | False

Indicator for conditional residuals, specified as the comma-separated pair consisting of 'Conditional' and one of the following.

True	Contribution from both fixed effects and random effects (conditional)
False	Contribution from only fixed effects (marginal)

Example: 'Conditional, 'False'

### ResidualType — Residual type

'Raw' (default) | 'Pearson' | 'Standardized'

Residual type, specified by the comma-separated pair consisting of ResidualType and one of the following.

Residual Type	Conditional	Marginal
'Raw'	$r_i^C = [y - X\hat{\beta} - Z\hat{b}]_i$	$r_i^M = [y - X\hat{\beta}]_i$
'Pearson'	$pr_i^C = \frac{r_i^C}{\sqrt{[\widehat{\text{Var}}_{y,b}(y - X\hat{\beta} - Z\hat{b})]_{ii}}}$	$pr_i^M = \frac{r_i^M}{\sqrt{[\widehat{\text{Var}}_y(y - X\hat{\beta})]_{ii}}}$
'Standardized'	$st_i^C = \frac{r_i^C}{\sqrt{[\widehat{\text{Var}}_y(r^C)]_{ii}}}$	$st_i^M = \frac{r_i^M}{\sqrt{[\widehat{\text{Var}}_y(r^M)]_{ii}}}$

For more information on the conditional and marginal residuals and residual variances, see Definitions at the end of this page.

Example: 'ResidualType', 'Standardized'

## Output Arguments

### R — Residuals

*n*-by-1 vector

Residuals of the fitted linear mixed-effects model `lme` returned as an *n*-by-1 vector, where *n* is the number of observations.

## Examples

### Plot Residuals vs. Fitted Values

Load the sample data.

```
load(fullfile(matlabroot, 'examples', 'stats', 'weight.mat'));
```

`weight` contains data from a longitudinal study, where 20 subjects are randomly assigned to 4 exercise programs, and their weight loss is recorded over six 2-week time periods. This is simulated data.

Store the data in a table. Define `Subject` and `Program` as categorical variables.

```
tbl = table(InitialWeight, Program, Subject, Week, y);  
tbl.Subject = nominal(tbl.Subject);  
tbl.Program = nominal(tbl.Program);
```

Fit a linear mixed-effects model where the initial weight, type of program, week, and the interaction between the week and type of program are the fixed effects. The intercept and week vary by subject.

```
lme = fitlme(tbl, 'y ~ InitialWeight + Program*Week + (Week|Subject)');
```

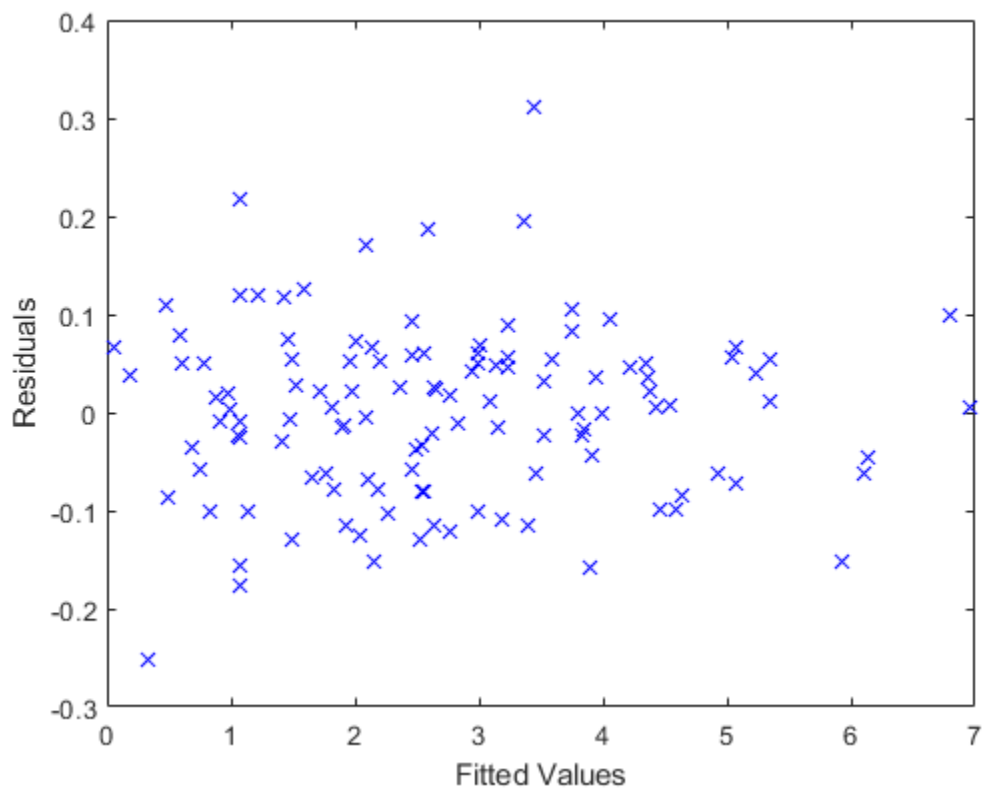
Compute the fitted values and raw residuals.

```
F = fitted(lme);  
R = residuals(lme);
```

Plot the residuals versus the fitted values.

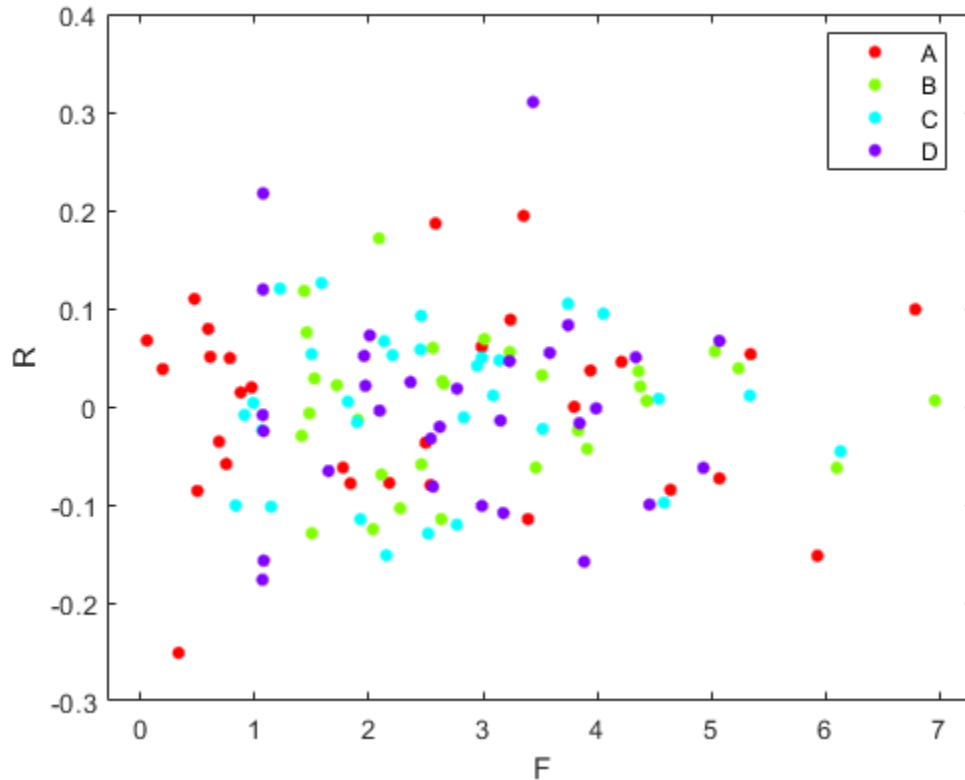


```
plot(F,R,'bx')
xlabel('Fitted Values')
ylabel('Residuals')
```



Now, plot the residuals versus the fitted values, grouped by program.

```
figure();
gscatter(F,R,Program)
```



The residuals seem to behave similarly across levels of the program as expected.

### Compute Conditional and Marginal Pearson Residuals

Load the sample data.

```
load carbig
```

Store the variables for miles per gallon (MPG), acceleration, horsepower, cylinders, and model year in a table.

```
tbl = table(MPG,Acceleration,Horsepower,Cylinders,Model_Year);
```

Fit a linear mixed-effects model for miles per gallon (MPG), with fixed effects for acceleration, horsepower, and the cylinders, and potentially correlated random effects for intercept and acceleration grouped by model year.

```
lme = fitlme(tbl, 'MPG ~ Acceleration + Horsepower + Cylinders + (Acceleration|Model_Year)')
```

Compute the conditional Pearson residuals and display the first five residuals.

```
PR = residuals(lme, 'ResidualType', 'Pearson');  
PR(1:5)
```

```
ans =
```

```
-0.0533  
 0.0652  
 0.3655  
-0.0106  
-0.3340
```

Compute the marginal Pearson residuals and display the first five residuals.

```
PRM = residuals(lme, 'ResidualType', 'Pearson', 'Conditional', false);  
PRM(1:5)
```

```
ans =
```

```
-0.1250  
 0.0130  
 0.3242  
-0.0861  
-0.3006
```

## Examine Residuals

Load the sample data.

```
load carbig
```

Store the variables for miles per gallon (MPG), acceleration, horsepower, cylinders, and model year in a table.

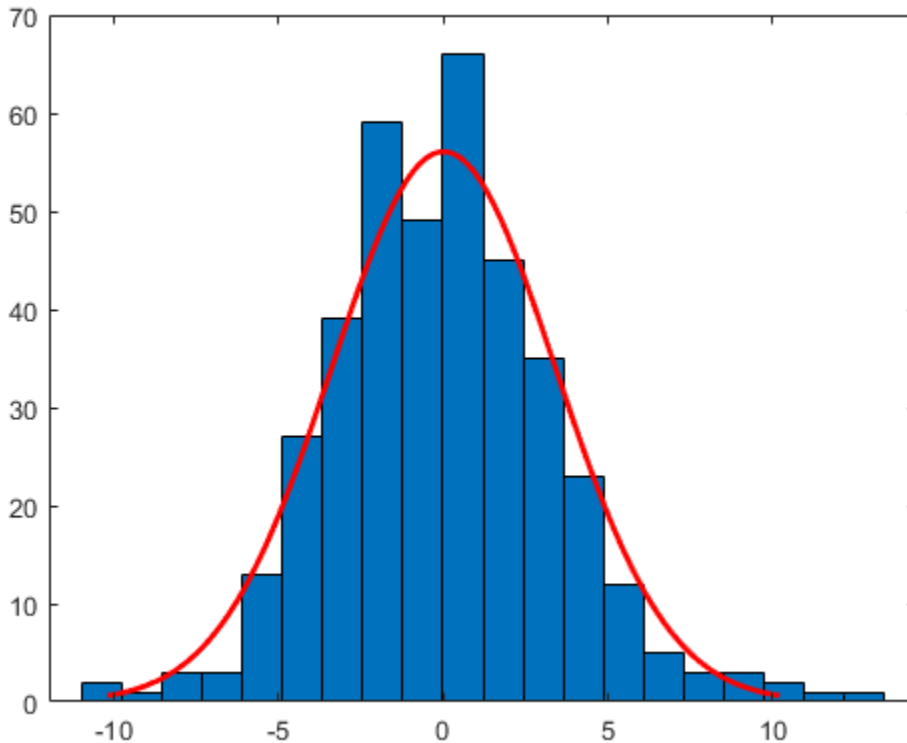
```
tbl = table(MPG,Acceleration,Horsepower,Cylinders,Model_Year);
```

Fit a linear mixed-effects model for miles per gallon (MPG), with fixed effects for acceleration, horsepower, and the cylinders, and potentially correlated random effects for intercept and acceleration grouped by model year.

```
lme = fitlme(tbl,'MPG ~ Acceleration + Horsepower + Cylinders + (Acceleration|Model_Year)');
```

Draw a histogram of the raw residuals with a normal fit.

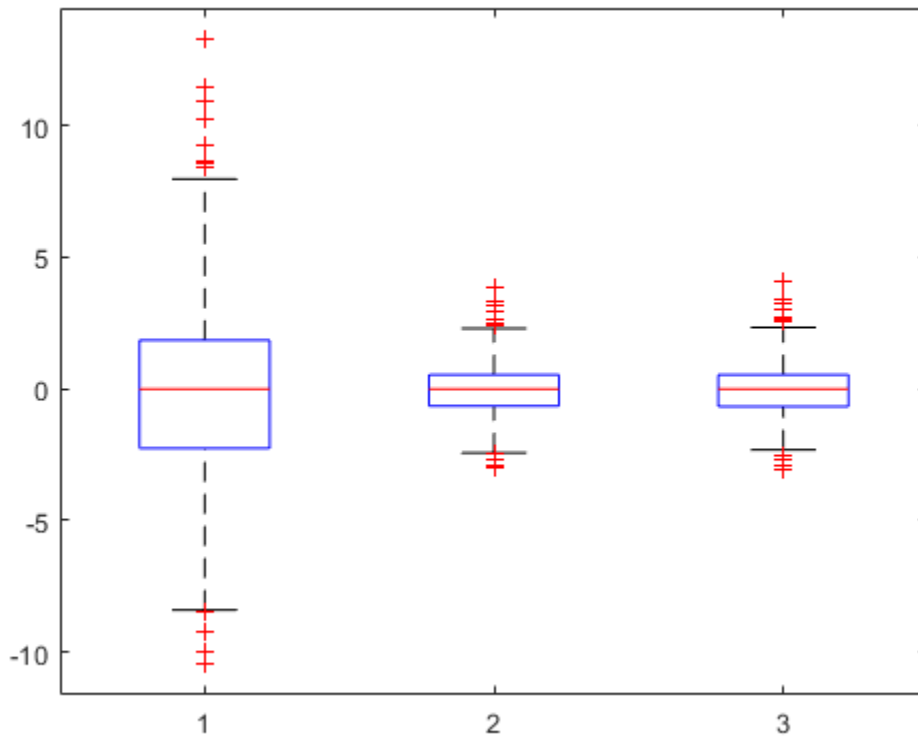
```
r = residuals(lme);  
histfit(r)
```



Normal distribution seems to be a good fit for the residuals.

Compute the conditional Pearson and standardized residuals and create box plots of all three types of residuals.

```
pr = residuals(lme, 'ResidualType', 'Pearson');
st = residuals(lme, 'ResidualType', 'Standardized');
X = [r pr st];
figure();
boxplot(X)
```



Red plus signs show the observations with residuals above or below  $q3 + 1.5(q3 - q1)$  and  $q1 - 1.5(q3 - q1)$ , where  $q1$  and  $q3$  are the 25th and 75th percentiles, respectively.

Find the observations with residuals that are 2.5 standard deviations above and below the mean.

```
find(r > nanmean(r) + 2.5*nanstd(r))
```

```
ans =
```

```
    62  
   252  
   255  
   330  
   337  
   341  
   396
```

```
find(r < nanmean(r) - 2.5*nanstd(r))
```

```
ans =
```

```
   119  
   324  
   375
```

## Definitions

### Conditional and Marginal Residuals

Conditional residuals include contributions from both fixed and random effects, whereas marginal residuals include contribution from only fixed effects.

Suppose the linear mixed-effects model `lme` has an  $n$ -by- $p$  fixed-effects design matrix  $X$  and an  $n$ -by- $q$  random-effects design matrix  $Z$ . Also, suppose the  $p$ -by-1 estimated fixed-

effects vector is  $\hat{\beta}$ , and the  $q$ -by-1 estimated best linear unbiased predictor (BLUP)

vector of random effects is  $\hat{b}$ . The fitted conditional response is

$$\hat{y}_{Cond} = X\hat{\beta} + Z\hat{b},$$

and the fitted marginal response is

$$\hat{y}_{Mar} = X\hat{\beta},$$

`residuals` can return three types of residuals: raw, Pearson, and standardized. For any type, you can compute the conditional or the marginal residuals. For example, the conditional raw residual is

$$r_{Cond} = y - X\hat{\beta} - Z\hat{b},$$

and the marginal raw residual is

$$r_{Mar} = y - X\hat{\beta}.$$

For more information on other types of residuals, see the `ResidualType` name-value pair argument.

## See Also

`LinearMixedModel` | `fitted` | `plotResiduals` | `response`

## response

**Class:** GeneralizedLinearMixedModel

Response vector of generalized linear mixed-effects model

## Syntax

```
y = response(glme)
[y,binomialsizes] = response(glme)
```

## Description

`y = response(glme)` returns the response vector `y` used to fit the generalized linear mixed effects model `glme`.

`[y,binomialsizes] = response(glme)` also returns the binomial size associated with each element of `y` if the conditional distribution of response given the random effects is binomial.

## Input Arguments

**glme** — Generalized linear mixed-effects model

GeneralizedLinearMixedModel object

Generalized linear mixed-effects model, specified as a GeneralizedLinearMixedModel object. For properties and methods of this object, see GeneralizedLinearMixedModel.

## Output Arguments

**y** — Response values

$n$ -by-1 vector

Response values, specified as an  $n$ -by-1 vector, where  $n$  is the number of observations.



For an observation  $i$  with prior weights  $w_i^p$  and binomial size  $n_i$  (when applicable), the response values  $y_i$  can have the following values.

Distribution	Permitted Values	Notes
Binomial	$\left\{0, \frac{1}{w_i^p n_i}, \frac{2}{w_i^p n_i}, \dots, 1\right\}$	$w_i^p$ and $n_i$ are integer values $> 0$
Poisson	$\left\{0, \frac{1}{w_i^p}, \frac{2}{w_i^p}, \dots\right\}$	$w_i^p$ is an integer value $> 0$
Gamma	$(0, \infty)$	$w_i^p \geq 0$
InverseGaussian	$(0, \infty)$	$w_i^p \geq 0$
normal	$(-\infty, \infty)$	$w_i^p \geq 0$

You can access the prior weights property  $w_i^p$  using dot notation. For example, to access the prior weights property for a model `glme`:

```
glme.ObservationInfo.Weights
```

### **binomialsize** — Binomial size

vector

Binomial size associated with each element of `y`, returned as an  $n$ -by-1 vector, where  $n$  is the number of observations. `response` only returns `binomialsize` if the conditional distribution of response given the random effects is binomial. `binomialsize` is empty for other distributions.

## Examples

### Plot Response Versus Fitted Values

Load the sample data.

```
load mfr
```

This simulated data is from a manufacturing company that operates 50 factories across the world, with each factory running a batch process to create a finished product. The

company wants to decrease the number of defects in each batch, so it developed a new manufacturing process. To test the effectiveness of the new process, the company selected 20 of its factories at random to participate in an experiment: Ten factories implemented the new process, while the other ten continued to run the old process. In each of the 20 factories, the company ran five batches (for a total of 100 batches) and recorded the following data:

- Flag to indicate whether the batch used the new process (`newprocess`)
- Processing time for each batch, in hours (`time`)
- Temperature of the batch, in degrees Celsius (`temp`)
- Categorical variable indicating the supplier (A, B, or C) of the chemical used in the batch (`supplier`)
- Number of defects in the batch (`defects`)

The data also includes `time_dev` and `temp_dev`, which represent the absolute deviation of time and temperature, respectively, from the process standard of 3 hours at 20 degrees Celsius.

Fit a generalized linear mixed-effects model using `newprocess`, `time_dev`, `temp_dev`, and `supplier` as fixed-effects predictors. Include a random-effects term for intercept grouped by `factory`, to account for quality differences that might exist due to factory-specific variations. The response variable `defects` has a Poisson distribution, and the appropriate link function for this model is `log`. Use the Laplace fit method to estimate the coefficients. Specify the dummy variable encoding as `'effects'`, so the dummy variable coefficients sum to 0.

The number of defects can be modeled using a Poisson distribution  
 $\text{defects}_{ij} \sim \text{Poisson}(\mu_{ij})$

This corresponds to the generalized linear mixed-effects model

$$\log(\mu_{ij}) = \beta_0 + \beta_1 \text{newprocess}_{ij} + \beta_2 \text{time.dev}_{ij} + \beta_3 \text{temp.dev}_{ij} + \beta_4 \text{supplier.C}_{ij} + \beta_5 \text{supplier.B}_{ij} + b_i,$$

where

- $\text{defects}_{ij}$  is the number of defects observed in the batch produced by factory  $i$  during batch  $j$ .
- $\mu_{ij}$  is the mean number of defects corresponding to factory  $i$  (where  $i = 1, 2, \dots, 20$ ) during batch  $j$  (where  $j = 1, 2, \dots, 5$ ).

- `newprocessij`, `time_devij`, and `temp_devij` are the measurements for each variable that correspond to factory *i* during batch *j*. For example, `newprocessij` indicates whether the batch produced by factory *i* during batch *j* used the new process.
- `supplier_Cij` and `supplier_Bij` are dummy variables that use effects (sum-to-zero) coding to indicate whether company C or B, respectively, supplied the process chemicals for the batch produced by factory *i* during batch *j*.
- $b_i \sim N(0, \sigma_b^2)$  is a random-effects intercept for each factory *i* that accounts for factory-specific variation in quality.

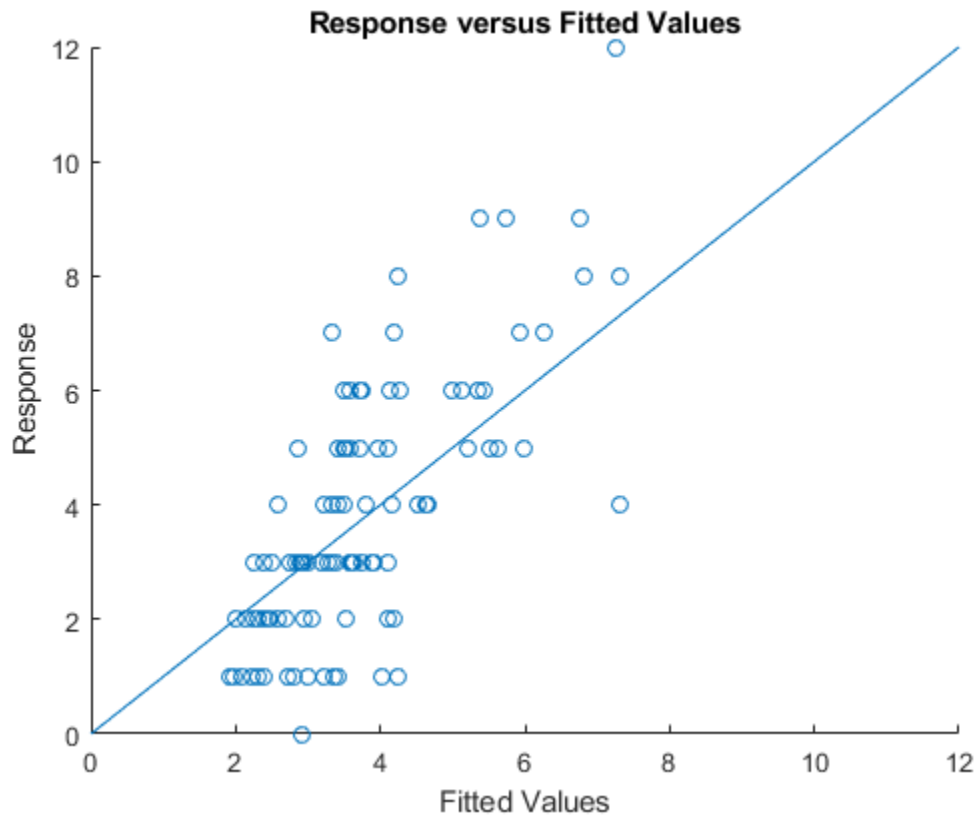
```
glme = fitglme(mfr, 'defects ~ 1 + newprocess + time_dev + temp_dev + supplier + (1|factory)
  Distribution', 'Poisson', 'Link', 'log', 'FitMethod', 'Laplace', 'DummyVarCoding', 'effects')
```

Extract the observed response values for the model, then use `fitted` to generate the fitted conditional mean values.

```
y = response(glme); % Observed response values
yfit = fitted(glme); % Fitted response values
```

Create a scatterplot of the observed response values versus fitted values. Add a reference line to improve the visualization.

```
figure
scatter(yfit, y)
xlim([0, 12])
ylim([0, 12])
refline(1, 0)
title('Response versus Fitted Values')
xlabel('Fitted Values')
ylabel('Response')
```



The plot shows a positive correlation between the fitted values and the observed response values.

## References

- [1] Hox, J. *Multilevel Analysis, Techniques and Applications*. Lawrence Erlbaum Associates, Inc., 2002.

## See Also

`GeneralizedLinearMixedModel` | `fitted` | `residuals`

## response

**Class:** LinearMixedModel

Response vector of the linear mixed-effects model

## Syntax

```
y = response(lme)
```

## Description

`y = response(lme)` returns the response vector `y` used to fit the linear mixed-effects model `lme`.

## Input Arguments

**lme** — Linear mixed-effects model

LinearMixedModel object

Linear mixed-effects model, specified as a LinearMixedModel object constructed using `fitlme` or `fitlmematrix`.

## Output Arguments

**y** — Response values

$n$ -by-1 vector

Response values, specified as an  $n$ -by-1 vector, where  $n$  is the number of observations.

Data Types: `single` | `double`

## Examples

### Plot Response versus Fitted Values

Load the sample data.

```
load(fullfile(matlabroot, 'examples', 'stats', 'weight.mat'));
```

`weight` contains data from a longitudinal study, where 20 subjects are randomly assigned to 4 exercise programs, and their weight loss is recorded over two-week time periods. This is simulated data.

Store the data in a table. Define `Subject` and `Program` as categorical variables.

```
tbl = table(InitialWeight, Program, Subject, Week, y);  
tbl.Subject = nominal(tbl.Subject);  
tbl.Program = nominal(tbl.Program);
```

Fit a linear mixed-effects model where the initial weight, type of program, week, and the interaction between the week and type of program are the fixed effects. The intercept and week vary by subject.

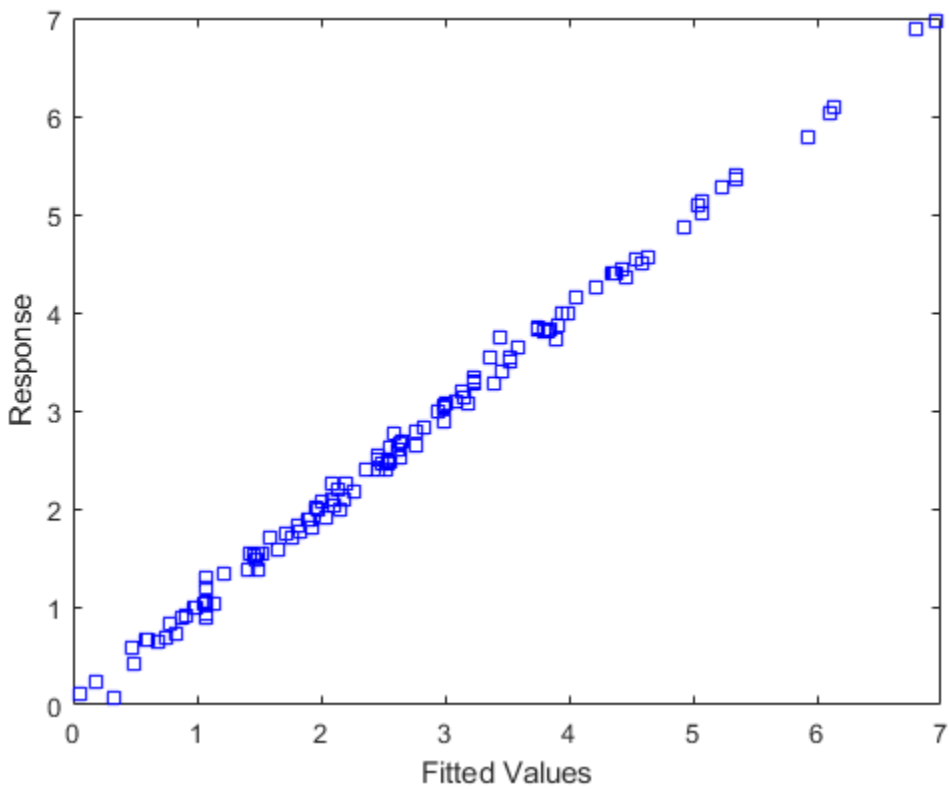
```
lme = fitlme(tbl, 'y ~ InitialWeight + Program*Week + (Week|Subject)');
```

Compute the fitted values and the response.

```
F = fitted(lme);  
y = response(lme);
```

Plot the response versus the fitted values.

```
plot(F, y, 'bs')  
xlabel('Fitted Values')  
ylabel('Response')
```



## See Also

`LinearMixedModel` | `fitted` | `residuals`

## resubEdge

**Class:** ClassificationDiscriminant

Classification edge by resubstitution

### Syntax

```
edge = resubEdge(obj)
```

### Description

`edge = resubEdge(obj)` returns the classification edge obtained by `obj` on its training data.

### Input Arguments

**obj**

Discriminant analysis classifier, produced using `fitcdiscr`.

### Output Arguments

**edge**

Classification edge obtained by resubstituting the training data into the calculation of `edge`.

### Examples



## Estimate the Resubstitution Edge of Discriminant Analysis Classifiers

Estimate the quality of a discriminant analysis classifier for Fisher's iris data by resubstitution.

Load Fisher's iris data set.

```
load fisheriris
```

Train a discriminant analysis classifier.

```
Mdl = fitcdiscr(meas, species);
```

Compute the resubstitution edge.

```
redge = resubEdge(Mdl)
```

```
redge =  
    0.9454
```

## Definitions

### Edge

The edge is the weighted mean value of the classification margin. The weights are class prior probabilities. If you supply additional weights, those weights are normalized to sum to the prior probabilities in the respective classes, and are then used to compute the weighted average.

### Margin

The classification margin is the difference between the classification score for the true class and maximal classification score for the false classes.

The classification margin is a column vector with the same number of rows as in the matrix  $X$ . A high value of margin indicates a more reliable prediction than a low value.

## Score

For discriminant analysis, the score of a classification is the posterior probability of the classification. For the definition of posterior probability in discriminant analysis, see “Posterior Probability” on page 20-8.

## See Also

`ClassificationDiscriminant` | `edge` | `fitcdiscr` | `resubMargin`

## Topics

“Discriminant Analysis Classification” on page 20-2

# resubEdge

**Class:** ClassificationECOC

Classification edge by resubstitution for multiclass, error-correcting output codes model

## Syntax

```
e = resubEdge(Mdl)
e = resubEdge(Mdl, Name, Value)
```

## Description

`e = resubEdge(Mdl)` returns the classification edge on page 32-5683 (e) for the trained, multiclass, error-correcting output codes (ECOC) model `Mdl` using the training data stored in `Mdl.X` and corresponding class labels stored in `Mdl.Y`.

`e = resubEdge(Mdl, Name, Value)` computes the resubstitution classification edge with additional options specified by one or more `Name, Value` pair arguments.

For example, specify a decoding scheme, binary learner loss function, or verbosity level.

## Input Arguments

**mdl** — Multiclass ECOC model

ClassificationECOC model

Multiclass ECOC model, specified as a `ClassificationECOC` model returned by `fitcecoc`.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single

quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

### BinaryLoss — Binary learner loss function

'hamming' | 'linear' | 'logit' | 'exponential' | 'binodeviance' | 'hinge'  
| 'quadratic' | function handle

Binary learner loss function, specified as the comma-separated pair consisting of 'BinaryLoss' and a built-in, loss-function name or function handle.

- This table contains names and descriptions of the built-in functions, where  $y_j$  is a class label for a particular binary learner (in the set  $\{-1, 1, 0\}$ ),  $s_j$  is the score for observation  $j$ , and  $g(y_j, s_j)$  is the binary loss formula.

Value	Description	Score Domain	$g(y_j, s_j)$
'binodeviance'	Binomial deviance	$(-\infty, \infty)$	$\log[1 + \exp(-2y_j s_j)] / [2\log(2)]$
'exponential'	Exponential	$(-\infty, \infty)$	$\exp(-y_j s_j) / 2$
'hamming'	Hamming	$[0, 1]$ or $(-\infty, \infty)$	$[1 - \text{sign}(y_j s_j)] / 2$
'hinge'	Hinge	$(-\infty, \infty)$	$\max(0, 1 - y_j s_j) / 2$
'linear'	Linear	$(-\infty, \infty)$	$(1 - y_j s_j) / 2$
'logit'	Logistic	$(-\infty, \infty)$	$\log[1 + \exp(-y_j s_j)] / [2\log(2)]$
'quadratic'	Quadratic	$[0, 1]$	$[1 - y_j(2s_j - 1)]^2 / 2$

The software normalizes the binary losses such that the loss is 0.5 when  $y_j = 0$ . Also, the software calculates the mean binary loss for each class.

- For a custom binary loss function, e.g., `customFunction`, specify its function handle `'BinaryLoss', @customFunction`.

`customFunction` should have this form:

```
bLoss = customFunction(M, s)
```

where:

- $M$  is the  $K$ -by- $L$  coding matrix stored in `Mdl.CodingMatrix`.
- $s$  is the 1-by- $L$  row vector of classification scores.

- `bLoss` is the classification loss. This scalar aggregates the binary losses for every learner in a particular class. For example, you can use the mean binary loss to aggregate the loss over the learners for each class.
- $K$  is the number of classes.
- $L$  is the number of binary learners.

For an example of passing a custom binary loss function, see “Predict Test-Sample Labels of ECOC Models Using Custom Binary Loss Function” on page 32-4933.

By default, if all binary learners are:

- SVMs or linear classification models of SVM learners, then `BinaryLoss` is `'hinge'`
- Ensembles trained by `AdaboostM1` or `GentleBoost`, then `BinaryLoss` is `'exponential'`
- Ensembles trained by `LogitBoost`, then `BinaryLoss` is `'binodeviance'`
- Linear classification models of logistic regression learners, or you specify to predict class posterior probabilities (i.e., set `'FitPosterior', 1` in `fitcecoc`), then `BinaryLoss` is `'quadratic'`

Otherwise, the default `BinaryLoss` is `'hamming'`. To check the default value, use dot notation to display the `BinaryLoss` property of the trained model at the command line.

Example: `'BinaryLoss', 'binodeviance'`

Data Types: `char` | `function_handle`

### Decoding — Decoding scheme

`'lossweighted'` (default) | `'lossbased'`

Decoding scheme that aggregates the binary losses on page 32-4941, specified as the comma-separated pair consisting of `'Decoding'` and `'lossweighted'` or `'lossbased'`.

Example: `'Decoding', 'lossbased'`

Data Types: `char`

### Options — Estimation options

`[]` (default) | structure array returned by `statset`

Estimation options, specified as the comma-separated pair consisting of `'Options'` and a structure array returned by `statset`.

To invoke parallel computing:

- You need a Parallel Computing Toolbox license.
- Specify 'Options', `statset('UseParallel',1)`.

### **Verbose** — Verbosity level

0 (default) | 1

Verbosity level, specified as the comma-separated pair consisting of 'Verbose' and 0 or 1. Verbose controls the amount of diagnostic messages that the software displays in the Command Window.

If `Verbose` is 0, then the software does not display diagnostic messages. Otherwise, the software displays diagnostic messages.

Example: `'Verbose',1`

Data Types: `single` | `double`

## Output Arguments

### **e** — Classification edge

numeric scalar

Classification edge, returned as a scalar. `e` represents the (weighted) mean of the classification margins on page 32-5683.

## Examples

### **Estimate the Resubstitution Edge of ECOC Models**

Load Fisher's iris data set.

```
load fisheriris
X = meas;
Y = species;
```

Train an ECOC model using SVM binary classifiers. It is good practice to standardize the predictors and define the class order. Specify to standardize the predictors using an SVM template.

```
t = templateSVM('Standardize',1);
classOrder = unique(Y)
Mdl = fitcecoc(X,Y,'Learners',t,'ClassNames',classOrder);
```

```
classOrder =

    3x1 cell array

    {'setosa'      }
    {'versicolor'}
    {'virginica'  }
```

`t` is an SVM template object. The software uses default values for empty options in `t` during training. `Mdl` is a ClassificationECOC model.

Estimate the resubstitution edge.

```
e = resubEdge(Mdl)
```

```
e =

    0.4960
```

The mean of the training sample margins is 0.4961.

### Select ECOC Model Features by Comparing In-Sample Edges

The classifier edge measures the average of the classifier margins. One way to perform feature selection is to compare training sample edges from multiple models. Based solely on this criterion, the classifier with the highest edge is the best classifier.

Load Fisher's iris data set. Define two data sets:

- `fullX` contains all four predictors.

- `partX` contains the sepal measurements.

```
load fisheriris
X = meas;
fullX = X;
partX = X(:,1:2);
Y = species;
```

Train ECOC models using SVM binary learners for each predictor set. It is good practice to define the class order. Specify to standardize the predictors using an SVM template, and to compute posterior probabilities.

```
t = templateSVM('Standardize',1);
classOrder = unique(Y)
FullMdl = fitcecoc(fullX,Y,'Learners',t,'ClassNames',classOrder,...
    'FitPosterior',1);
PartMdl = fitcecoc(partX,Y,'Learners',t,'ClassNames',classOrder,...
    'FitPosterior',1);
```

```
classOrder =
    3x1 cell array
    {'setosa'    }
    {'versicolor'}
    {'virginica' }
```

The default SVM score is distance from the decision boundary. If you specify to compute posterior probabilities, then the software uses posterior probabilities as scores.

Estimate the training sample edge for each classifier. The quadratic loss function operates on scores in the domain [0,1]. Specify to use quadratic loss when aggregating the binary learners for both models.

```
fullEdge = resubEdge(FullMdl,'BinaryLoss','quadratic')
partEdge = resubEdge(PartMdl,'BinaryLoss','quadratic')
```

```
fullEdge =
    0.9896
```



```
partEdge =  
    0.5058
```

The edge for the classifier trained on the complete data set is greater, suggesting that the classifier trained using every predictor has a better in-sample fit.

- “Quick Start Parallel Computing for Statistics and Machine Learning Toolbox” on page 30-2

## Definitions

### Classification Edge

The classification edge is the weighted mean of the classification margins.

One way to choose among multiple classifiers, e.g., to perform feature selection, is to choose the classifier that yields the highest edge.

### Classification Margin

The classification margins are, for each observation, the difference between the negative loss for the true class and maximal negative loss among the false classes. If the margins are on the same scale, then they serve as a classification confidence measure, i.e., among multiple classifiers, those that yield larger margins are better.

### Binary Loss

A binary loss is a function of the class and classification score that determines how well a binary learner classifies an observation into the class.

Let:

- $m_{kj}$  be element  $(k,j)$  of the coding design matrix  $M$  (i.e., the code corresponding to class  $k$  of binary learner  $j$ )
- $s_j$  be the score of binary learner  $j$  for an observation
- $g$  be the binary loss function

- $\hat{k}$  be the predicted class for the observation

In loss-based decoding [15], the class producing the minimum sum of the binary losses over binary learners determines the predicted class of an observation, that is,

$$\hat{k} = \operatorname{argmin}_k \sum_{j=1}^L |m_{kj}| g(m_{kj}, s_j).$$

In loss-weighted decoding [15], the class producing the minimum average of the binary losses over binary learners determines the predicted class of an observation, that is,

$$\hat{k} = \operatorname{argmin}_k \frac{\sum_{j=1}^L |m_{kj}| g(m_{kj}, s_j)}{\sum_{j=1}^L |m_{kj}|}.$$

Allwein et al. [2] suggest that loss-weighted decoding improves classification accuracy by keeping loss values for all classes in the same dynamic range.

This table summarizes the supported loss functions, where  $y_j$  is a class label for a particular binary learner (in the set  $\{-1, 1, 0\}$ ),  $s_j$  is the score for observation  $j$ , and  $g(y_j, s_j)$ .

Value	Description	Score Domain	$g(y_j, s_j)$
'binodeviance'	Binomial deviance	$(-\infty, \infty)$	$\log[1 + \exp(-2y_j s_j)] / [2\log(2)]$
'exponential'	Exponential	$(-\infty, \infty)$	$\exp(-y_j s_j) / 2$
'hamming'	Hamming	$[0, 1]$ or $(-\infty, \infty)$	$[1 - \operatorname{sign}(y_j s_j)] / 2$
'hinge'	Hinge	$(-\infty, \infty)$	$\max(0, 1 - y_j s_j) / 2$
'linear'	Linear	$(-\infty, \infty)$	$(1 - y_j s_j) / 2$
'logit'	Logistic	$(-\infty, \infty)$	$\log[1 + \exp(-y_j s_j)] / [2\log(2)]$
'quadratic'	Quadratic	$[0, 1]$	$[1 - y_j(2s_j - 1)]^2 / 2$

The software normalizes the binary losses such that the loss is 0.5 when  $y_j = 0$ , and aggregates using the average of the binary learners [2].

Do not confuse the binary loss with the overall classification loss (specified by the `LossFun` name-value pair argument of `predict` and `loss`), e.g., classification error, which measures how well an ECOC classifier performs as a whole.

## Tip

To compare margins or edges of several classifiers, use template objects to specify a common score transform function among the classifiers when you train them using `fitcecoc`.

## References

- [1] Allwein, E., R. Schapire, and Y. Singer. “Reducing multiclass to binary: A unifying approach for margin classifiers.” *Journal of Machine Learning Research*. Vol. 1, 2000, pp. 113–141.
- [2] Escalera, S., O. Pujol, and P. Radeva. “On the decoding process in ternary error-correcting output codes.” *IEEE Transactions on Pattern Analysis and Machine Intelligence*. Vol. 32, Issue 7, 2010, pp. 120–134.
- [3] Escalera, S., O. Pujol, and P. Radeva. “Separability of ternary codes for sparse designs of error-correcting output codes.” *Pattern Recogn.* Vol. 30, Issue 3, 2009, pp. 285–297.

## See Also

`ClassificationECOC` | `edge` | `fitcecoc` | `predict` | `resbuMargin` | `resubPredict`

## Topics

“Quick Start Parallel Computing for Statistics and Machine Learning Toolbox” on page 30-2

“Reproducibility in Parallel Statistical Computations” on page 30-14

“Concepts of Parallel Computing in Statistics and Machine Learning Toolbox” on page 30-8

## resubEdge

**Class:** ClassificationEnsemble

Classification edge by resubstitution

### Syntax

```
edge = resubEdge(ens)
edge = resubEdge(ens, Name, Value)
```

### Description

`edge = resubEdge(ens)` returns the classification edge obtained by `ens` on its training data.

`edge = resubEdge(ens, Name, Value)` calculates edge with additional options specified by one or more `Name, Value` pair arguments. You can specify several name-value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### Input Arguments

**ens**

A classification ensemble created with `fitcensemble`.

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

**learners**

Indices of weak learners in the ensemble ranging from 1 to NumTrained. `resubEdge` uses only these learners for calculating edge.

**Default:** `1:NumTrained`

**mode**

Character vector representing the meaning of the output edge:

- 'ensemble' — edge is a scalar value, the loss for the entire ensemble.
- 'individual' — edge is a vector with one element per trained learner.
- 'cumulative' — edge is a vector in which element  $J$  is obtained by using learners `1:J` from the input list of learners.

**Default:** 'ensemble'

## Output Arguments

**edge**

Classification edge obtained by `ens` by resubstituting the training data into the calculation of edge. Classification edge is classification margin averaged over the entire data. `edge` can be a scalar or vector, depending on the setting of the `mode` name-value pair.

## Examples

### Find Classification Edge by Resubstitution of Training Data

Find the resubstitution edge for an ensemble that classifies the Fisher iris data.

Load the sample data set.

```
load fisheriris
```

Train an ensemble of 100 boosted classification trees using `AdaBoostM2`.

```
t = templateTree('MaxNumSplits',1); % Weak learner template tree object
ens = fitcensemble(meas,species,'Method','AdaBoostM2','Learners',t);
```

Find the resubstitution edge.

```
edge = resubEdge(ens)

edge = 3.2486
```

## Definitions

### Edge

The edge is the weighted mean value of the classification margin. The weights are the class probabilities in `ens.Prior`.

### Margin

The classification margin is the difference between the classification score for the true class and maximal classification score for the false classes. Margin is a column vector with the same number of rows as in the matrix `ens.X`.

### Score (ensemble)

For ensembles, a classification score represents the confidence of a classification into a class. The higher the score, the higher the confidence.

Different ensemble algorithms have different definitions for their scores. Furthermore, the range of scores depends on ensemble type. For example:

- AdaBoostM1 scores range from  $-\infty$  to  $\infty$ .
- Bag scores range from 0 to 1.

## See Also

`resubEdge` | `resubLoss` | `resubMargin` | `resubPredict`

## resubEdge

**Class:** ClassificationKNN

Edge of  $k$ -nearest neighbor classifier by resubstitution

## Syntax

```
E = resubEdge mdl)
```

## Description

`E = resubEdge(mdl)` returns the classification edge for `mdl` with the data used to train `mdl` (see “Edge” on page 32-5691).

## Input Arguments

**mdl** — Classifier model

classifier model object

$k$ -nearest neighbor classifier model, returned as a classifier model object.

Note that using the 'CrossVal', 'Kfold', 'Holdout', 'Leaveout', or 'CVPartition' options results in a model of class `ClassificationPartitionedModel`. You cannot use a partitioned tree for prediction, so this kind of tree does not have a `predict` method.

Otherwise, `mdl` is of class `ClassificationKNN`, and you can use the `predict` method to make predictions.

## Output Arguments

### E

Classification edge, a scalar that is the mean classification margin (see “Margin” on page 32-5691).

## Examples

### Resubstitution Edge Calculation

Construct a  $k$ -nearest neighbor classifier for the Fisher iris data, where  $k = 5$ .

Load the data.

```
load fisheriris
X = meas;
Y = species;
```

Construct a classifier for 5-nearest neighbors.

```
mdl = fitcknn(X,Y,'NumNeighbors',5);
```

Examine the resubstitution edge of the classifier.

```
E = resubEdge(mdl)
```

```
E =
    0.9253
```

- “Examine Quality of KNN Classifier” on page 18-37



## Definitions

### Edge

The edge is the mean value of the classification margin.

### Margin

The classification margin is the difference between the classification score for the true class and maximal classification score for the false classes.

Margin is a column vector with the same number of rows as in the training data.

### Score

The score of a classification is the posterior probability of the classification. The posterior probability is the number of neighbors that have that classification, divided by the number of neighbors. For a more detailed definition that includes weights and prior probabilities, see “Posterior Probability” on page 32-4897.

## See Also

`ClassificationKNN` | `fitcknn` | `resubLoss` | `resubMargin` | `resubPredict`

## Topics

“Examine Quality of KNN Classifier” on page 18-37

“Classification Using Nearest Neighbors” on page 18-16

## resubEdge

**Class:** ClassificationNaiveBayes

Classification edge for naive Bayes classifiers by resubstitution

### Syntax

```
e = resubEdge(Mdl)
```

### Description

`e = resubEdge(Mdl)` returns the resubstitution classification edge on page 32-5694 (`e`) for the naive Bayes classifier `Mdl` using the training data stored in `Mdl.X` and corresponding class labels stored in `Mdl.Y`.

### Input Arguments

**Mdl** — Fully trained naive Bayes classifier

`ClassificationNaiveBayes` model

A fully trained naive Bayes classifier, specified as a `ClassificationNaiveBayes` model trained by `fitcnb`.

### Output Arguments

**e** — Classification edge

scalar

Classification edge on page 32-5694, returned as a scalar. If you passed in weights when training the classifier, then `e` is the weighted classification edge. The software normalizes the weights so that they sum to the prior probability of their respective class.

## Examples

### Estimate the Resubstitution Edge of Naive Bayes Classifiers

Load Fisher's iris data set.

```
load fisheriris
X = meas;    % Predictors
Y = species; % Response
rng(1);
```

Train a naive Bayes classifier. It is good practice to specify the class order. Assume that each predictor is conditionally, normally distributed given its label.

```
Mdl = fitcnb(X,Y,'ClassNames',{'setosa','versicolor','virginica'});
```

Mdl is a trained `ClassificationNaiveBayes` classifier.

Estimate the resubstitution edge.

```
e = resubEdge(Mdl)
```

```
e =
    0.8944
```

The mean of the training sample margins is approximately 0.9, which indicates that the classifier classifies in-sample observations with high confidence.

### Select Naive Bayes Classifier Features by Comparing In-Sample Edges

The classifier edge measures the average of the classifier margins. One way to perform feature selection is to compare training sample edges from multiple models. Based solely on this criterion, the classifier with the highest edge is the best classifier.

Load Fisher's iris data set. Define two data sets:

- `fullX` contains all predictors.

- `partX` contains the last two predictors.

```
load fisheriris
X = meas;      % Predictors
Y = species;  % Response
fullX = X;
partX = X(:,3:4);
```

Train naive Bayes classifiers for each predictor set.

```
FullMdl = fitcnb(fullX,Y);
PartMdl = fitcnb(partX,Y);
```

Estimate the training sample edge for each classifier.

```
fullEdge = resubEdge(FullMdl)
partEdge = resubEdge(PartMdl)
```

```
fullEdge =
```

```
    0.8944
```

```
partEdge =
```

```
    0.9169
```

The edge for the classifier trained on predictors 3 and 4 is greater, suggesting that the classifier trained using only those predictors has a better in-sample fit.

## Definitions

### Classification Edge

The classification edge is the weighted mean of the classification margins.

If you supply weights, then the software normalizes them to sum to the prior probability of their respective class. The software uses the normalized weights to compute the weighted mean.

One way to choose among multiple classifiers, e.g., to perform feature selection, is to choose the classifier that yields the highest edge.

## Classification Margins

The classification margins are, for each observation, the difference between the score for the true class and maximal score for the false classes. Provided that they are on the same scale, margins serve as a classification confidence measure, i.e., among multiple classifiers, those that yield larger margins are better.

## Posterior Probability

The posterior probability is the probability that an observation belongs in a particular class, given the data.

For naive Bayes, the posterior probability that a classification is  $k$  for a given observation  $(x_1, \dots, x_P)$  is

$$\hat{P}(Y = k | x_1, \dots, x_P) = \frac{P(X_1, \dots, X_P | y = k)\pi(Y = k)}{P(X_1, \dots, X_P)},$$

where:

- $P(X_1, \dots, X_P | y = k)$  is the conditional joint density of the predictors given they are in class  $k$ . `Mdl.DistributionNames` stores the distribution names of the predictors.
- $\pi(Y = k)$  is the class prior probability distribution. `Mdl.Prior` stores the prior distribution.
- $P(X_1, \dots, X_P)$  is the joint density of the predictors. The classes are discrete, so

$$P(X_1, \dots, X_P) = \sum_{k=1}^K P(X_1, \dots, X_P | y = k)\pi(Y = k).$$

## Prior Probability

The prior probability is the believed relative frequency that observations from a class occur in the population for each class.

## Score

The naive Bayes score is the class posterior probability given the observation.

## See Also

`ClassificationNaiveBayes` | `CompactClassificationNaiveBayes` | `edge` | `fitcnb` | `loss` | `margin` | `predict` | `resubEdge` | `resubLoss` | `resubLoss` | `resubMargin`

## Topics

“Naive Bayes Classification” on page 21-2

# resubEdge

**Class:** ClassificationSVM

Classification edge for support vector machine classifiers by resubstitution

## Syntax

```
e = resubEdge(SVMModel)
```

## Description

`e = resubEdge(SVMModel)` returns the resubstitution classification edge on page 32-5699 (e) for the support vector machine (SVM) classifier `SVMModel` using the training data stored in `SVMModel.X` and corresponding class labels stored in `SVMModel.Y`.

## Input Arguments

**SVMModel** — Full, trained SVM classifier

ClassificationSVM classifier

Full, trained SVM classifier, specified as a `ClassificationSVM` model trained using `fitcsvm`.

## Output Arguments

**e** — Classification edge

scalar

Classification edge, returned as a scalar. `e` represents the weighted mean of the classification margins on page 32-5700.

## Examples

### Estimate the Resubstitution Edge of SVM Classifiers

Load the `ionosphere` data set.

```
load ionosphere
```

Train an SVM classifier. It is good practice to standardize the predictors and define the class order.

```
SVMModel = fitcsvm(X,Y,'Standardize',true,'ClassNames',{'b','g'});
```

`SVMModel` is a trained `ClassificationSVM` classifier. 'b' is the negative class and 'g' is the positive class.

Estimate the resubstitution edge.

```
e = resubEdge(SVMModel)
```

```
e =
```

```
5.0998
```

The mean of the training sample margins is 5.0999.

### Select SVM Classifier Features by Comparing In-Sample Edges

The classifier edge measures the average of the classifier margins. One way to perform feature selection is to compare training sample edges from multiple models. Based solely on this criterion, the classifier with the highest edge is the best classifier.

Load the `ionosphere` data set. Define two data sets:

- `fullX` contains all predictors (except the removed column of 0s).
- `partX` contains the last 20 predictors.



```
load ionosphere
fullX = X;
partX = X(:,end-20:end);
```

Train SVM classifiers for each predictor set.

```
FullSVMModel = fitcsvm(fullX,Y);
PartSVMModel = fitcsvm(partX,Y);
```

Estimate the training sample edge for each classifier.

```
fullEdge = resubEdge(FullSVMModel)
partEdge = resubEdge(PartSVMModel)
```

```
fullEdge =
```

```
    3.3653
```

```
partEdge =
```

```
    2.0470
```

The edge for the classifier trained on the complete data set is greater, suggesting that the classifier trained using all of the predictors has a better in-sample fit.

## Definitions

### Classification Edge

The edge is the weighted mean of the classification margins.

The weights are the prior class probabilities. If you supply weights, then the software normalizes them to sum to the prior probabilities in the respective classes. The software uses the renormalized weights to compute the weighted mean.

One way to choose among multiple classifiers, e.g., to perform feature selection, is to choose the classifier that yields the highest edge.

## Classification Margins

The classification margins for binary classification are, for each observation, the difference between the classification score for the true class and the classification score for the false class.

The software defines the classification margin for binary classification as

$$m = 2yf(x).$$

$x$  is an observation. If the true label of  $x$  is the positive class, then  $y$  is 1, and  $-1$  otherwise.  $f(x)$  is the positive-class classification score for the observation  $x$ . The literature commonly defines the margin as  $m = yf(x)$ .

If the margins are on the same scale, then they serve as a classification confidence measure, i.e., among multiple classifiers, those that yield larger margins are better.

## Classification Score

The SVM classification score for classifying observation  $x$  is the signed distance from  $x$  to the decision boundary ranging from  $-\infty$  to  $+\infty$ . A positive score for a class indicates that  $x$  is predicted to be in that class. A negative score indicates otherwise.

The score for predicting  $x$  into the positive class, also the numerical, predicted response for  $x$ ,  $f(x)$ , is the trained SVM classification function

$$f(x) = \sum_{j=1}^n \alpha_j y_j G(x_j, x) + b,$$

where  $(\alpha_1, \dots, \alpha_n, b)$  are the estimated SVM parameters,  $G(x_j, x)$  is the dot product in the predictor space between  $x$  and the support vectors, and the sum includes the training set observations. The score for predicting  $x$  into the negative class is  $-f(x)$ .

If  $G(x_j, x) = x_j'x$  (the linear kernel), then the score function reduces to

$$f(x) = (x / s)' \beta + b.$$

$s$  is the kernel scale and  $\beta$  is the vector of fitted linear coefficients.

## Algorithms

For binary classification, the software defines the margin for observation  $j$ ,  $m_j$ , as

$$m_j = 2y_j f(x_j),$$

where  $y_j \in \{-1, 1\}$ , and  $f(x_j)$  is the predicted score of observation  $j$  for the positive class. However, the literature commonly uses  $m_j = y_j f(x_j)$  to define the margin.

## References

- [1] Christianini, N., and J. C. Shawe-Taylor. *An Introduction to Support Vector Machines and Other Kernel-Based Learning Methods*. Cambridge, UK: Cambridge University Press, 2000.

## See Also

[ClassificationSVM](#) | [CompactClassificationSVM](#) | [edge](#) | [fitcsvm](#) | [resubLoss](#)  
| [resubMargin](#)

## resubEdge

**Class:** ClassificationTree

Classification edge by resubstitution

## Syntax

```
edge = resubEdge(tree)
```

## Description

`edge = resubEdge(tree)` returns the classification edge obtained by `tree` on its training data.

## Input Arguments

**tree**

A classification tree created using `fitctree`.

## Output Arguments

**edge**

Classification edge obtained by resubstituting the training data into the calculation of `edge`.

## Examples

Estimate the quality of a classification tree for the Fisher iris data by resubstitution.

```
load fisheriris
tree = fitctree(meas,species);
redge = resubEdge(tree)

redge =
    0.9384
```

## Definitions

### Edge

The edge is the weighted mean value of the classification margin. The weights are the class probabilities in `tree.Prior`.

### Margin

The classification margin is the difference between the classification score for the true class and maximal classification score for the false classes. Margin is a column vector with the same number of rows as in the matrix `X`.

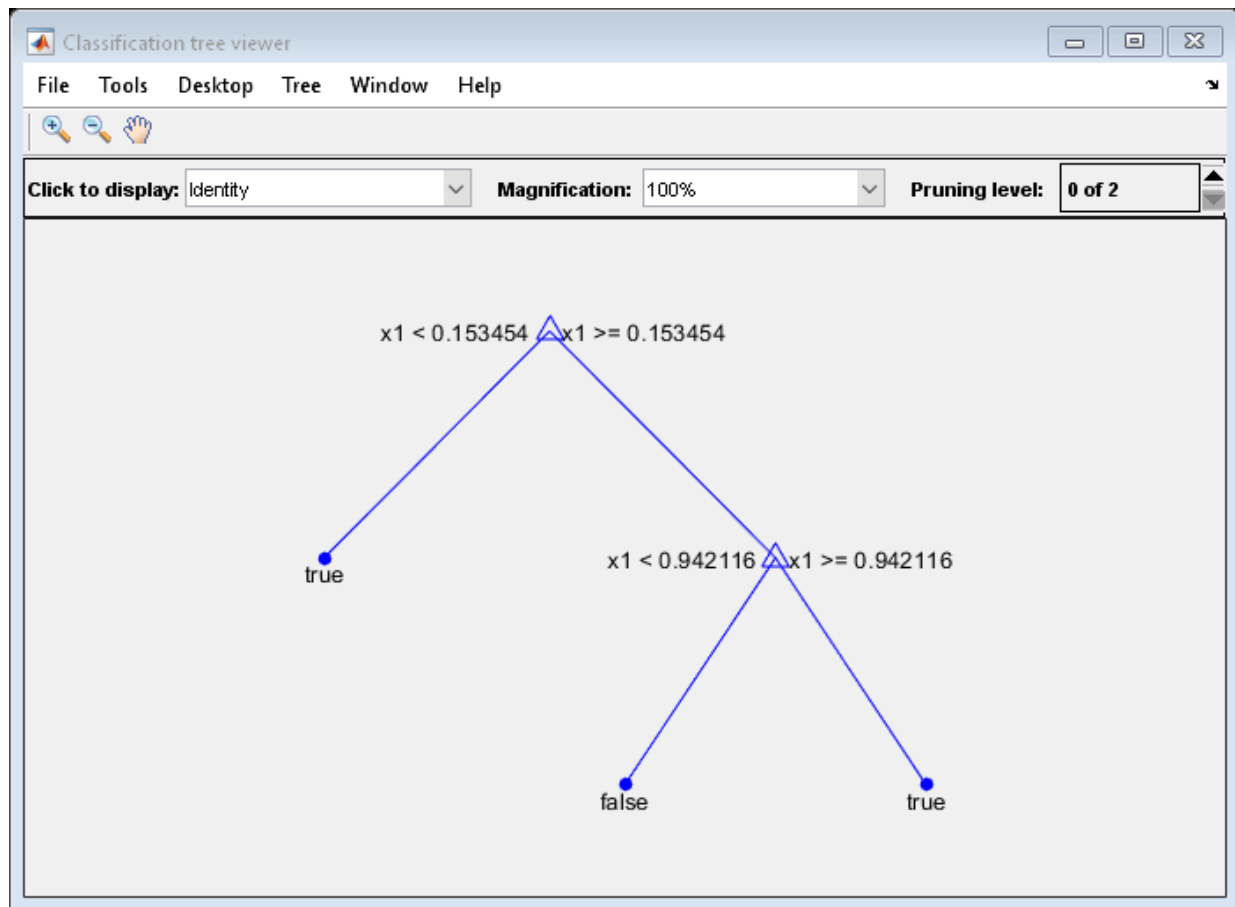
### Score (tree)

For trees, the score of a classification of a leaf node is the posterior probability of the classification at that node. The posterior probability of the classification at a node is the number of training sequences that lead to that node with the classification, divided by the number of training sequences that lead to that node.

For example, consider classifying a predictor `X` as `true` when  $X < 0.15$  or  $X > 0.95$ , and `X` is `false` otherwise.

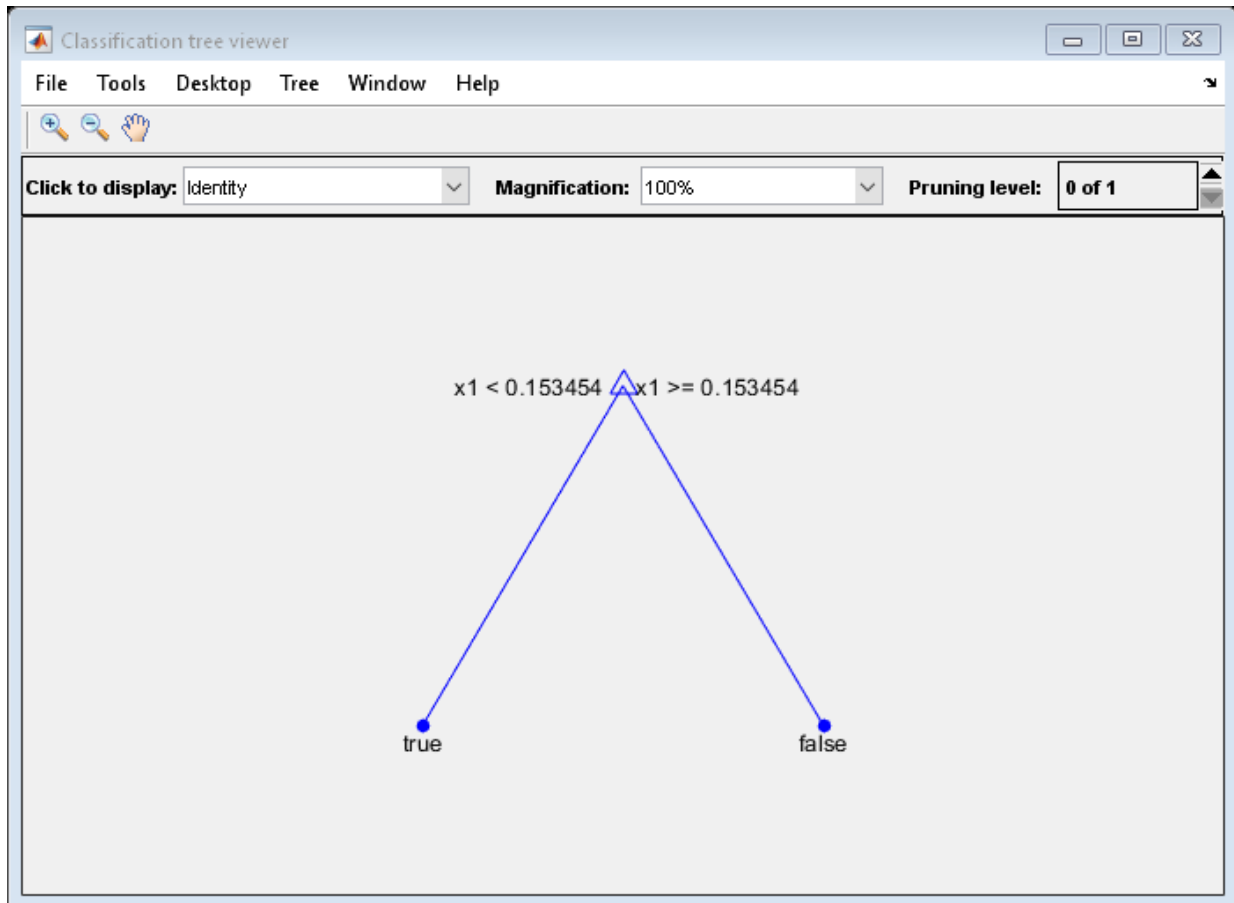
Generate 100 random points and classify them:

```
rng(0, 'twister') % for reproducibility
X = rand(100,1);
Y = (abs(X - .55) > .4);
tree = fitctree(X,Y);
view(tree, 'Mode', 'Graph')
```



Prune the tree:

```
tree1 = prune(tree, 'Level', 1);  
view(tree1, 'Mode', 'Graph')
```



The pruned tree correctly classifies observations that are less than 0.15 as `true`. It also correctly classifies observations from .15 to .94 as `false`. However, it incorrectly classifies observations that are greater than .94 as `false`. Therefore, the score for observations that are greater than .15 should be about  $.05/.85=.06$  for `true`, and about  $.8/.85=.94$  for `false`.

Compute the prediction scores for the first 10 rows of `X`:

```
[~,score] = predict(tree1,X(1:10));
[score X(1:10,:)]
```

```
ans =  
  
    0.9059    0.0941    0.8147  
    0.9059    0.0941    0.9058  
         0     1.0000    0.1270  
    0.9059    0.0941    0.9134  
    0.9059    0.0941    0.6324  
         0     1.0000    0.0975  
    0.9059    0.0941    0.2785  
    0.9059    0.0941    0.5469  
    0.9059    0.0941    0.9575  
    0.9059    0.0941    0.9649
```

Indeed, every value of  $x$  (the right-most column) that is less than 0.15 has associated scores (the left and center columns) of 0 and 1, while the other values of  $x$  have associated scores of 0.91 and 0.09. The difference (score 0.09 instead of the expected .06) is due to a statistical fluctuation: there are 8 observations in  $x$  in the range  $(.95, 1)$  instead of the expected 5 observations.

## See Also

`edge` | `fitctree` | `resubLoss` | `resubMargin` | `resubPredict`



# resubLoss

**Class:** ClassificationDiscriminant

Classification error by resubstitution

## Syntax

```
L = resubLoss(obj)
L = resubLoss(obj,Name,Value)
```

## Description

`L = resubLoss(obj)` returns the resubstitution loss, meaning the loss computed for the data that `fitcdiscr` used to create `obj`.

`L = resubLoss(obj,Name,Value)` returns loss statistics with additional options specified by one or more `Name,Value` pair arguments.

## Input Arguments

**obj**

Discriminant analysis classifier, produced using `fitcdiscr`.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

### LossFun — Loss function

'classiferror' (default) | 'binodeviance' | 'exponential' | 'hinge' | 'logit' | 'mincost' | 'quadratic' | function handle

Loss function, specified as the comma-separated pair consisting of 'LossFun' and a built-in, loss-function name or function handle.

- The following lists available loss functions. Specify one using its corresponding character vector.

Value	Description
'binodeviance'	Binomial deviance
'classiferror'	Classification error
'exponential'	Exponential
'hinge'	Hinge
'logit'	Logistic
'mincost'	Minimal expected misclassification cost (for classification scores that are posterior probabilities)
'quadratic'	Quadratic

'mincost' is appropriate for classification scores that are posterior probabilities. Discriminant analysis models return posterior probabilities as classification scores by default (see `predict`).

- Specify your own function using function handle notation.

Suppose that  $n$  be the number of observations in  $X$  and  $K$  be the number of distinct classes (`numel(obj.ClassNames)`). Your function must have this signature

```
lossvalue = lossfun(C,S,W,Cost)
```

where:

- The output argument `lossvalue` is a scalar.
- You choose the function name (`lossfun`).
- $C$  is an  $n$ -by- $K$  logical matrix with rows indicating which class the corresponding observation belongs. The column order corresponds to the class order in `obj.ClassNames`.

Construct  $C$  by setting  $C(p, q) = 1$  if observation  $p$  is in class  $q$ , for each row. Set all other elements of row  $p$  to 0.

- $S$  is an  $n$ -by- $K$  numeric matrix of classification scores. The column order corresponds to the class order in `obj.ClassNames`.  $S$  is a matrix of classification scores, similar to the output of `predict`.
- $W$  is an  $n$ -by-1 numeric vector of observation weights. If you pass  $W$ , the software normalizes them to sum to 1.
- $Cost$  is a  $K$ -by- $K$  numeric matrix of misclassification costs. For example, `Cost = ones(K) - eye(K)` specifies a cost of 0 for correct classification, and 1 for misclassification.

Specify your function using `'LossFun', @lossfun`.

For more details on loss functions, see “Classification Loss” on page 32-5710.

Data Types: `char` | `function_handle`

## Output Arguments

**L**

Classification error, a scalar. The meaning of the error depends on the values in `weights` and `lossfun`. See “Classification Loss” on page 32-5710.

## Examples

Compute the resubstituted classification error for the Fisher iris data:

```
load fisheriris
obj = fitcdiscr(meas, species);
L = resubLoss(obj)
```

```
L =
    0.0200
```

## Definitions

### Classification Loss

Classification loss functions measure the predictive inaccuracy of classification models. When comparing the same type of loss among many models, lower loss indicates a better predictive model.

Suppose that:

- $L$  is the weighted average classification loss.
- $n$  is the sample size.
- For binary classification:
  - $y_j$  is the observed class label. The software codes it as  $-1$  or  $1$  indicating the negative or positive class, respectively.
  - $f(X_j)$  is the raw classification score for observation (row)  $j$  of the predictor data  $X$ .
  - $m_j = y_j f(X_j)$  is the classification score for classifying observation  $j$  into the class corresponding to  $y_j$ . Positive values of  $m_j$  indicate correct classification and do not contribute much to the average loss. Negative values of  $m_j$  indicate incorrect classification and contribute to the average loss.
- For algorithms that support multiclass classification (that is,  $K \geq 3$ ):
  - $y_j^*$  is a vector of  $K - 1$  zeros, and a  $1$  in the position corresponding to the true, observed class  $y_j$ . For example, if the true class of the second observation is the third class and  $K = 4$ , then  $y_2^* = [0 \ 0 \ 1 \ 0]'$ . The order of the classes corresponds to the order in the `ClassNames` property of the input model.
  - $f(X_j)$  is the length  $K$  vector of class scores for observation  $j$  of the predictor data  $X$ . The order of the scores corresponds to the order of the classes in the `ClassNames` property of the input model.
  - $m_j = y_j^{*'} f(X_j)$ . Therefore,  $m_j$  is the scalar classification score that the model predicts for the true, observed class.
- The weight for observation  $j$  is  $w_j$ . The software normalizes the observation weights so that they sum to the corresponding prior class probability. The software also normalizes the prior probabilities so they sum to  $1$ . Therefore,

$$\sum_{j=1}^n w_j = 1.$$

The supported loss functions are:

- Binomial deviance, specified using 'LossFun', 'binodeviance'. Its equation is

$$L = \sum_{j=1}^n w_j \log \{1 + \exp[-2m_j]\}.$$

- Exponential loss, specified using 'LossFun', 'exponential'. Its equation is

$$L = \sum_{j=1}^n w_j \exp(-m_j).$$

- Classification error, specified using 'LossFun', 'classiferror'. It is the weighted fraction of misclassified observations, with equation

$$L = \sum_{j=1}^n w_j I\{\hat{y}_j \neq y_j\}.$$

$\hat{y}_j$  is the class label corresponding to the class with the maximal posterior probability.  $I\{x\}$  is the indicator function.

- Hinge loss, specified using 'LossFun', 'hinge'. Its equation is

$$L = \sum_{j=1}^n w_j \max\{0, 1 - m_j\}.$$

- Logit loss, specified using 'LossFun', 'logit'. Its equation is

$$L = \sum_{j=1}^n w_j \log(1 + \exp(-m_j)).$$

- Minimal cost, specified using 'LossFun', 'mincost'. The software computes the weighted minimal cost using this procedure for observations  $j = 1, \dots, n$ :

- 1 Estimate the 1-by- $K$  vector of expected classification costs for observation  $j$

$$\gamma_j = f(X_j)'C.$$

$f(X_j)$  is the column vector of class posterior probabilities for binary and multiclass classification.  $C$  is the cost matrix the input model stores in the property `Cost`.

- 2 For observation  $j$ , predict the class label corresponding to the minimum, expected classification cost:

$$\hat{y}_j = \min_{j=1,\dots,K} (\gamma_j).$$

**3** Using  $C$ , identify the cost incurred ( $c_j$ ) for making the prediction.

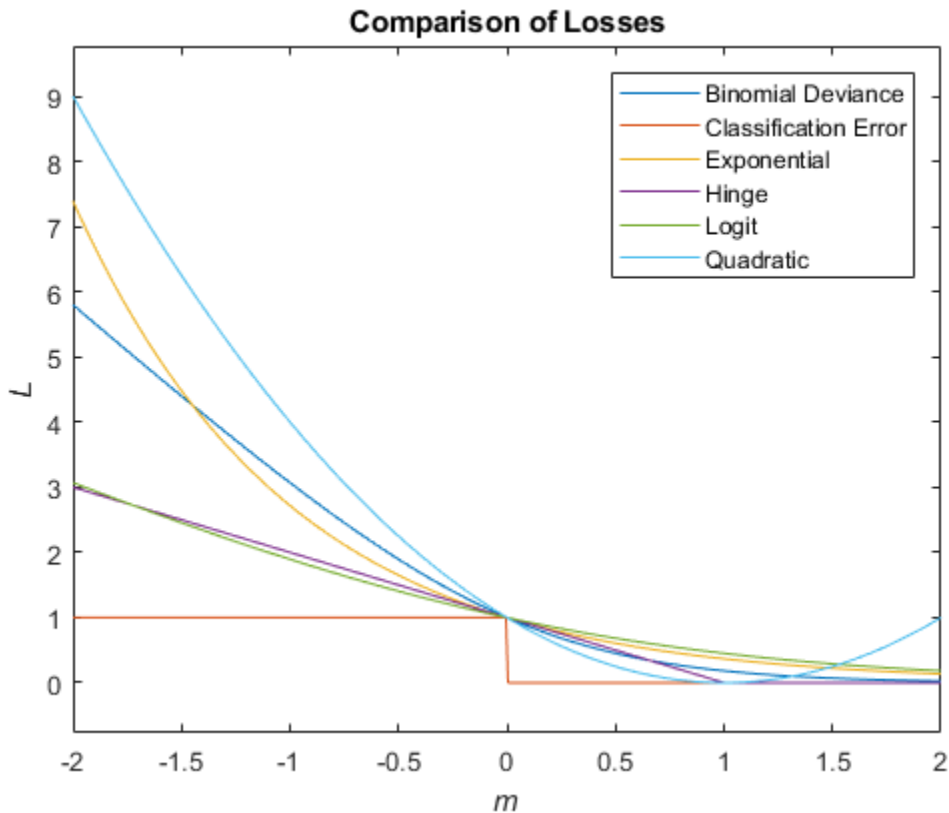
The weighted, average, minimum cost loss is

$$L = \sum_{j=1}^n w_j c_j.$$

- Quadratic loss, specified using 'LossFun', 'quadratic'. Its equation is

$$L = \sum_{j=1}^n w_j (1 - m_j)^2.$$

This figure compares some of the loss functions for one observation over  $m$  (some functions are normalized to pass through  $[0,1]$ ).



## Posterior Probability

The posterior probability that a point  $z$  belongs to class  $j$  is the product of the prior probability and the multivariate normal density. The density function of the multivariate normal with mean  $\mu_j$  and covariance  $\Sigma_j$  at a point  $z$  is

$$P(x | k) = \frac{1}{(2\pi|\Sigma_k|)^{1/2}} \exp\left(-\frac{1}{2}(x - \mu_k)^T \Sigma_k^{-1} (x - \mu_k)\right),$$

where  $|\Sigma_k|$  is the determinant of  $\Sigma_k$ , and  $\Sigma_k^{-1}$  is the inverse matrix.

Let  $P(k)$  represent the prior probability of class  $k$ . Then the posterior probability that an observation  $x$  is of class  $k$  is

$$\hat{P}(k | x) = \frac{P(x | k)P(k)}{P(x)},$$

where  $P(x)$  is a normalization constant, the sum over  $k$  of  $P(x | k)P(k)$ .

## Prior Probability

The prior probability is one of three choices:

- 'uniform' — The prior probability of class  $k$  is one over the total number of classes.
- 'empirical' — The prior probability of class  $k$  is the number of training samples of class  $k$  divided by the total number of training samples.
- Custom — The prior probability of class  $k$  is the  $k$ th element of the `prior` vector. See `fitcdiscr`.

After creating a classification model (`Mdl`) you can set the prior using dot notation:

```
Mdl.Prior = v;
```

where `v` is a vector of positive elements representing the frequency with which each element occurs. You do not need to retrain the classifier when you set a new prior.

## Cost

The matrix of expected costs per observation is defined in “Cost” on page 20-9.

## See Also

`ClassificationDiscriminant` | `fitcdiscr` | `loss`

## Topics

“Discriminant Analysis Classification” on page 20-2



## resubLoss

**Class:** ClassificationECOC

Classification loss by resubstitution for multiclass, error-correcting output codes model

### Syntax

```
L = resubLoss(Mdl)
L = resubLoss(Mdl, Name, Value)
```

### Description

`L = resubLoss(Mdl)` returns classification loss (`L`) for the trained, multiclass, error-correcting output codes (ECOC) model `Mdl` using the training data stored in `Mdl.X` and corresponding class labels stored in `Mdl.Y`.

`L = resubLoss(Mdl, Name, Value)` returns the classification loss with additional options specified by one or more `Name, Value` pair arguments.

For example, specify the loss function, decoding scheme, or verbosity level.

### Input Arguments

**mdl** — Multiclass ECOC model

ClassificationECOC model

Multiclass ECOC model, specified as a `ClassificationECOC` model returned by `fitcecoc`.

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single

quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### BinaryLoss — Binary learner loss function

'hamming' | 'linear' | 'logit' | 'exponential' | 'binodeviance' | 'hinge'  
| 'quadratic' | function handle

Binary learner loss function, specified as the comma-separated pair consisting of 'BinaryLoss' and a built-in, loss-function name or function handle.

- This table contains names and descriptions of the built-in functions, where  $y_j$  is a class label for a particular binary learner (in the set  $\{-1, 1, 0\}$ ),  $s_j$  is the score for observation  $j$ , and  $g(y_j, s_j)$  is the binary loss formula.

Value	Description	Score Domain	$g(y_j, s_j)$
'binodeviance'	Binomial deviance	$(-\infty, \infty)$	$\log[1 + \exp(-2y_j s_j)] / [2\log(2)]$
'exponential'	Exponential	$(-\infty, \infty)$	$\exp(-y_j s_j) / 2$
'hamming'	Hamming	$[0, 1]$ or $(-\infty, \infty)$	$[1 - \text{sign}(y_j s_j)] / 2$
'hinge'	Hinge	$(-\infty, \infty)$	$\max(0, 1 - y_j s_j) / 2$
'linear'	Linear	$(-\infty, \infty)$	$(1 - y_j s_j) / 2$
'logit'	Logistic	$(-\infty, \infty)$	$\log[1 + \exp(-y_j s_j)] / [2\log(2)]$
'quadratic'	Quadratic	$[0, 1]$	$[1 - y_j(2s_j - 1)]^2 / 2$

The software normalizes the binary losses such that the loss is 0.5 when  $y_j = 0$ . Also, the software calculates the mean binary loss for each class.

- For a custom binary loss function, e.g., `customFunction`, specify its function handle `'BinaryLoss', @customFunction`.

`customFunction` should have this form:

```
bLoss = customFunction(M, s)
```

where:

- $M$  is the  $K$ -by- $L$  coding matrix stored in `Mdl.CodingMatrix`.
- $s$  is the 1-by- $L$  row vector of classification scores.

- `bLoss` is the classification loss. This scalar aggregates the binary losses for every learner in a particular class. For example, you can use the mean binary loss to aggregate the loss over the learners for each class.
- $K$  is the number of classes.
- $L$  is the number of binary learners.

For an example of passing a custom binary loss function, see “Predict Test-Sample Labels of ECOC Models Using Custom Binary Loss Function” on page 32-4933.

By default, if all binary learners are:

- SVMs or linear classification models of SVM learners, then `BinaryLoss` is `'hinge'`
- Ensembles trained by `AdaboostM1` or `GentleBoost`, then `BinaryLoss` is `'exponential'`
- Ensembles trained by `LogitBoost`, then `BinaryLoss` is `'binodeviance'`
- Linear classification models of logistic regression learners, or you specify to predict class posterior probabilities (i.e., set `'FitPosterior', 1` in `fitcecoc`), then `BinaryLoss` is `'quadratic'`

Otherwise, the default `BinaryLoss` is `'hamming'`. To check the default value, use dot notation to display the `BinaryLoss` property of the trained model at the command line.

Example: `'BinaryLoss', 'binodeviance'`

Data Types: `char` | `function_handle`

### **Decoding — Decoding scheme**

`'lossweighted'` (default) | `'lossbased'`

Decoding scheme that aggregates the binary losses on page 32-4941, specified as the comma-separated pair consisting of `'Decoding'` and `'lossweighted'` or `'lossbased'`.

Example: `'Decoding', 'lossbased'`

Data Types: `char`

### **LossFun — Loss function**

`'classiferror'` (default) | `function_handle`

Loss function, specified as the comma-separated pair consisting of `'LossFun'` and a function handle or `'classiferror'`.

You can:

- Specify the built-in function `'classiferror'`, then the loss function is classification error on page 32-5722, in other words, the proportion of misclassified observations.
- Specify your own function using function handle notation.

Suppose that  $n = \text{size}(X, 1)$  is the sample size and  $k$  is the number of classes. Your function must have the signature `lossvalue = lossfun(C, S, W, Cost)`, where:

- The output argument `lossvalue` is a scalar.
- You choose the function name (`lossfun`).
- $C$  is an  $n$ -by- $k$  logical matrix with rows indicating to which class the corresponding observation belongs. The column order corresponds to the class order in `Mdl.ClassNames`.

Construct  $C$  by setting  $C(p, q) = 1$  if observation  $p$  is in class  $q$  for each row. Set all other elements of row  $p$  to 0.

- $S$  is an  $n$ -by- $k$  numeric matrix of negated loss values for classes. Each row corresponds to an observation. The column order corresponds to the class order in `CVMdl.ClassNames`.  $S$  is similar to the output argument `NegLoss` of `resubPredict`.
- $W$  is an  $n$ -by-1 numeric vector of observation weights. If you pass  $W$ , the software normalizes its elements to sum to 1.
- $Cost$  is a  $k$ -by- $k$  numeric matrix of misclassification costs. For example, `Cost = ones(K) - eye(K)` specifies a cost of 0 for correct classification, and 1 for misclassification.

Specify your function using `'LossFun', @lossfun`.

### Options — Estimation options

`[]` (default) | structure array returned by `statset`

Estimation options, specified as the comma-separated pair consisting of `'Options'` and a structure array returned by `statset`.

To invoke parallel computing:

- You need a Parallel Computing Toolbox license.

- Specify 'Options', `statset('UseParallel',1)`.

### **Verbose — Verbosity level**

0 (default) | 1

Verbosity level, specified as the comma-separated pair consisting of 'Verbose' and 0 or 1. Verbose controls the amount of diagnostic messages that the software displays in the Command Window.

If Verbose is 0, then the software does not display diagnostic messages. Otherwise, the software displays diagnostic messages.

Example: 'Verbose',1

Data Types: `single` | `double`

## Output Arguments

### **L — Classification loss**

scalar

Classification loss, returned as a scalar. L is a generalization or resubstitution quality measure. Its interpretation depends on the loss function and weighting scheme, but, in general, better classifiers yield smaller loss values.

## Examples

### **Determine the Resubstitution Loss of ECOC Models**

Load Fisher's iris data set.

```
load fisheriris
X = meas;
Y = species;
```

Train an ECOC model using SVM binary classifiers. It is good practice to standardize the predictors and define the class order. Specify to standardize the predictors using an SVM template.

```
t = templateSVM('Standardize',1);
classOrder = unique(Y)
Mdl = fitcecoc(X,Y,'Learners',t,'ClassNames',classOrder);
```

```
classOrder =

    3x1 cell array

    {'setosa'      }
    {'versicolor' }
    {'virginica'   }
```

`t` is an SVM template object. The software uses default values for empty options in `t` during training. `Mdl` is a `ClassificationECOC` model.

Estimate the resubstitution loss (i.e., the in-sample classification error).

```
L = resubLoss(Mdl)
```

```
L =

    0.0267
```

The ECOC model misclassifies 2.67% of the training sample irises.

### Determine the ECOC Model Quality Using a Custom Resubstitution Loss

Suppose that it is interesting to know how well a model classifies a particular class. This example shows how to pass such a custom loss function to `resubLoss`.

Load Fisher's iris data set.

```
load fisheriris
X = meas;
Y = categorical(species);
n = numel(Y);           % Sample size
classOrder = unique(Y) % Class order
K = numel(classOrder); % Number of classes
```

```
classOrder =
    3x1 categorical array

    setosa
    versicolor
    virginica
```

Train an ECOC model using SVM binary classifiers. It is good practice to standardize the predictors and define the class order. Specify to standardize the predictors using an SVM template.

```
t = templateSVM('Standardize',1);
Mdl = fitcecoc(X,Y,'Learners',t,'ClassNames',classOrder);
```

`t` is an SVM template object. The software uses default values for empty options in `t` during training. `Mdl` is a `ClassificationECOC` model.

Compute the negated losses for the training observations.

```
rng(1); % For reproducibility
[~,negLoss] = resubPredict(Mdl);
```

Create a function that takes the minimal loss for each observation, and then averages the minimal losses across all observations.

```
lossfun = @(C,S,~,~)mean(min(-negLoss,[],2));
```

Compute the custom loss for the training data.

```
resubLoss(Mdl,'LossFun',lossfun)
```

```
ans =
    0.0065
```

The average, minimal, binary loss in the training data is 0.0065.

- “Quick Start Parallel Computing for Statistics and Machine Learning Toolbox” on page 30-2

## Definitions

### Classification Error

The classification error is a binary classification error measure that has the form

$$L = \frac{\sum_{j=1}^n w_j e_j}{\sum_{j=1}^n w_j},$$

where:

- $w_j$  is the weight for observation  $j$ . The software renormalizes the weights to sum to 1.
- $e_j = 1$  if the predicted class of observation  $j$  differs from its true class, and 0 otherwise.

In other words, it is the proportion of observations that the classifier misclassifies.

### Binary Loss

A binary loss is a function of the class and classification score that determines how well a binary learner classifies an observation into the class.

Let:

- $m_{kj}$  be element  $(k,j)$  of the coding design matrix  $M$  (i.e., the code corresponding to class  $k$  of binary learner  $j$ )
- $s_j$  be the score of binary learner  $j$  for an observation
- $g$  be the binary loss function
- $\hat{k}$  be the predicted class for the observation

In loss-based decoding [15], the class producing the minimum sum of the binary losses over binary learners determines the predicted class of an observation, that is,

$$\hat{k} = \operatorname{argmin}_k \sum_{j=1}^L |m_{kj}| g(m_{kj}, s_j).$$



In loss-weighted decoding [15], the class producing the minimum average of the binary losses over binary learners determines the predicted class of an observation, that is,

$$\hat{k} = \underset{k}{\operatorname{argmin}} \frac{\sum_{j=1}^L |m_{kj}| g(m_{kj}, s_j)}{\sum_{j=1}^L |m_{kj}|}.$$

Allwein et al. [2] suggest that loss-weighted decoding improves classification accuracy by keeping loss values for all classes in the same dynamic range.

This table summarizes the supported loss functions, where  $y_j$  is a class label for a particular binary learner (in the set  $\{-1, 1, 0\}$ ),  $s_j$  is the score for observation  $j$ , and  $g(y_j, s_j)$ .

Value	Description	Score Domain	$g(y_j, s_j)$
'binodeviance'	Binomial deviance	$(-\infty, \infty)$	$\log[1 + \exp(-2y_j s_j)] / [2\log(2)]$
'exponential'	Exponential	$(-\infty, \infty)$	$\exp(-y_j s_j) / 2$
'hamming'	Hamming	$[0, 1]$ or $(-\infty, \infty)$	$[1 - \operatorname{sign}(y_j s_j)] / 2$
'hinge'	Hinge	$(-\infty, \infty)$	$\max(0, 1 - y_j s_j) / 2$
'linear'	Linear	$(-\infty, \infty)$	$(1 - y_j s_j) / 2$
'logit'	Logistic	$(-\infty, \infty)$	$\log[1 + \exp(-y_j s_j)] / [2\log(2)]$
'quadratic'	Quadratic	$[0, 1]$	$[1 - y_j(2s_j - 1)]^2 / 2$

The software normalizes the binary losses such that the loss is 0.5 when  $y_j = 0$ , and aggregates using the average of the binary learners [2].

Do not confuse the binary loss with the overall classification loss (specified by the `LossFun` name-value pair argument of `predict` and `loss`), e.g., classification error, which measures how well an ECOC classifier performs as a whole.

## References

- [1] Allwein, E., R. Schapire, and Y. Singer. "Reducing multiclass to binary: A unifying approach for margin classifiers." *Journal of Machine Learning Research*. Vol. 1, 2000, pp. 113–141.

- [2] Escalera, S., O. Pujol, and P. Radeva. “On the decoding process in ternary error-correcting output codes.” *IEEE Transactions on Pattern Analysis and Machine Intelligence*. Vol. 32, Issue 7, 2010, pp. 120–134.
- [3] Escalera, S., O. Pujol, and P. Radeva. “Separability of ternary codes for sparse designs of error-correcting output codes.” *Pattern Recogn.* Vol. 30, Issue 3, 2009, pp. 285–297.

## See Also

`ClassificationECOC` | `fitcecoc` | `loss` | `predict` | `resubPredict`

## Topics

“Quick Start Parallel Computing for Statistics and Machine Learning Toolbox” on page 30-2

“Reproducibility in Parallel Statistical Computations” on page 30-14

“Concepts of Parallel Computing in Statistics and Machine Learning Toolbox” on page 30-8

# resubLoss

**Class:** ClassificationEnsemble

Classification error by resubstitution

## Syntax

```
L = resubLoss(ens)
L = resubLoss(ens,Name,Value)
```

## Description

`L = resubLoss(ens)` returns the resubstitution loss, meaning the loss computed for the data that `fitcensemble` used to create `ens`.

`L = resubLoss(ens,Name,Value)` calculates loss with additional options specified by one or more `Name,Value` pair arguments. You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

## Input Arguments

### **ens**

A classification ensemble created with `fitcensemble`.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

**learners**

Indices of weak learners in the ensemble ranging from 1 to NumTrained. `resubLoss` uses only these learners for calculating loss.

**Default:** `1:NumTrained`

**lossfun**

Loss function, specified as the comma-separated pair consisting of 'LossFun' and a built-in, loss-function name or function handle.

- The following lists available loss functions. Specify one using its corresponding character vector.

Value	Description
'binodeviance'	Binomial deviance
'classiferror'	Classification error
'exponential'	Exponential
'hinge'	Hinge
'logit'	Logistic
'mincost'	Minimal expected misclassification cost (for classification scores that are posterior probabilities)
'quadratic'	Quadratic

'mincost' is appropriate for classification scores that are posterior probabilities.

- Bagged and subspace ensembles return posterior probabilities by default (`ens.Method` is 'Bag' or 'Subspace').
- If the ensemble method is 'AdaBoostM1', 'AdaBoostM2', GentleBoost, or 'LogitBoost', then, to use posterior probabilities as classification scores, you must specify the double-logit score transform by entering
 

```
ens.ScoreTransform = 'doublelogit';
```
- For all other ensemble methods, the software does not support posterior probabilities as classification scores.
- Specify your own function using function handle notation.

Suppose that  $n$  be the number of observations in  $X$  and  $K$  be the number of distinct classes (`numel(ens.ClassNames)`, `ens` is the input model). Your function must have this signature

```
lossvalue = lossfun(C,S,W,Cost)
```

where:

- The output argument `lossvalue` is a scalar.
- You choose the function name (`lossfun`).
- `C` is an  $n$ -by- $K$  logical matrix with rows indicating which class the corresponding observation belongs. The column order corresponds to the class order in `ens.ClassNames`.

Construct `C` by setting  $C(p, q) = 1$  if observation  $p$  is in class  $q$ , for each row. Set all other elements of row  $p$  to 0.

- `S` is an  $n$ -by- $K$  numeric matrix of classification scores. The column order corresponds to the class order in `ens.ClassNames`. `S` is a matrix of classification scores, similar to the output of `predict`.
- `W` is an  $n$ -by-1 numeric vector of observation weights. If you pass `W`, the software normalizes them to sum to 1.
- `Cost` is a  $K$ -by- $K$  numeric matrix of misclassification costs. For example, `Cost = ones(K) - eye(K)` specifies a cost of 0 for correct classification, and 1 for misclassification.

Specify your function using `'LossFun', @lossfun`.

For more details on loss functions, see “Classification Loss” on page 32-5728.

**Default:** `'classiferror'`

#### **mode**

Character vector representing the meaning of the output `L`:

- `'ensemble'` — `L` is a scalar value, the loss for the entire ensemble.
- `'individual'` — `L` is a vector with one element per trained learner.
- `'cumulative'` — `L` is a vector in which element  $J$  is obtained by using learners  $1:J$  from the input list of learners.

**Default:** 'ensemble'

## Output Arguments

**L**

Classification loss on page 32-5728, by default the fraction of misclassified data. **L** can be a vector, and can mean different things, depending on the name-value pair settings.

## Examples

### Estimate Classification Error for Training Observations

Load Fisher's iris data set.

```
load fisheriris
```

Boost 100 classification trees using AdaBoostM2.

```
ens = fitensemble(meas,species,'AdaBoostM2',100,'Tree');
```

Estimate the resubstitution classification error.

```
loss = resubLoss(ens)
```

```
loss =
```

```
    0.0333
```

## Definitions

### Classification Loss

Classification loss functions measure the predictive inaccuracy of classification models. When comparing the same type of loss among many models, lower loss indicates a better predictive model.

Suppose that:

- $L$  is the weighted average classification loss.
- $n$  is the sample size.
- For binary classification:
  - $y_j$  is the observed class label. The software codes it as  $-1$  or  $1$  indicating the negative or positive class, respectively.
  - $f(X_j)$  is the raw classification score for observation (row)  $j$  of the predictor data  $X$ .
  - $m_j = y_j f(X_j)$  is the classification score for classifying observation  $j$  into the class corresponding to  $y_j$ . Positive values of  $m_j$  indicate correct classification and do not contribute much to the average loss. Negative values of  $m_j$  indicate incorrect classification and contribute to the average loss.
- For algorithms that support multiclass classification (that is,  $K \geq 3$ ):
  - $y_j^*$  is a vector of  $K - 1$  zeros, and a 1 in the position corresponding to the true, observed class  $y_j$ . For example, if the true class of the second observation is the third class and  $K = 4$ , then  $y_2^* = [0 \ 0 \ 1 \ 0]'$ . The order of the classes corresponds to the order in the `ClassNames` property of the input model.
  - $f(X_j)$  is the length  $K$  vector of class scores for observation  $j$  of the predictor data  $X$ . The order of the scores corresponds to the order of the classes in the `ClassNames` property of the input model.
  - $m_j = y_j^{*'} f(X_j)$ . Therefore,  $m_j$  is the scalar classification score that the model predicts for the true, observed class.
- The weight for observation  $j$  is  $w_j$ . The software normalizes the observation weights so that they sum to the corresponding prior class probability. The software also normalizes the prior probabilities so they sum to 1. Therefore,

$$\sum_{j=1}^n w_j = 1.$$

The supported loss functions are:

- Binomial deviance, specified using `'LossFun', 'binodeviance'`. Its equation is

$$L = \sum_{j=1}^n w_j \log \left\{ 1 + \exp \left[ -2m_j \right] \right\}.$$

- Exponential loss, specified using 'LossFun', 'exponential'. Its equation is

$$L = \sum_{j=1}^n w_j \exp(-m_j).$$

- Classification error, specified using 'LossFun', 'classiferror'. It is the weighted fraction of misclassified observations, with equation

$$L = \sum_{j=1}^n w_j I\{\hat{y}_j \neq y_j\}.$$

$\hat{y}_j$  is the class label corresponding to the class with the maximal posterior probability.  $I\{x\}$  is the indicator function.

- Hinge loss, specified using 'LossFun', 'hinge'. Its equation is

$$L = \sum_{j=1}^n w_j \max\{0, 1 - m_j\}.$$

- Logit loss, specified using 'LossFun', 'logit'. Its equation is

$$L = \sum_{j=1}^n w_j \log(1 + \exp(-m_j)).$$

- Minimal cost, specified using 'LossFun', 'mincost'. The software computes the weighted minimal cost using this procedure for observations  $j = 1, \dots, n$ :

- 1 Estimate the 1-by- $K$  vector of expected classification costs for observation  $j$

$$\gamma_j = f(X_j)'C.$$

$f(X_j)$  is the column vector of class posterior probabilities for binary and multiclass classification.  $C$  is the cost matrix the input model stores in the property `Cost`.

- 2 For observation  $j$ , predict the class label corresponding to the minimum, expected classification cost:

$$\hat{y}_j = \min_{j=1, \dots, K} (\gamma_j).$$

- 3 Using  $C$ , identify the cost incurred ( $c_j$ ) for making the prediction.

The weighted, average, minimum cost loss is

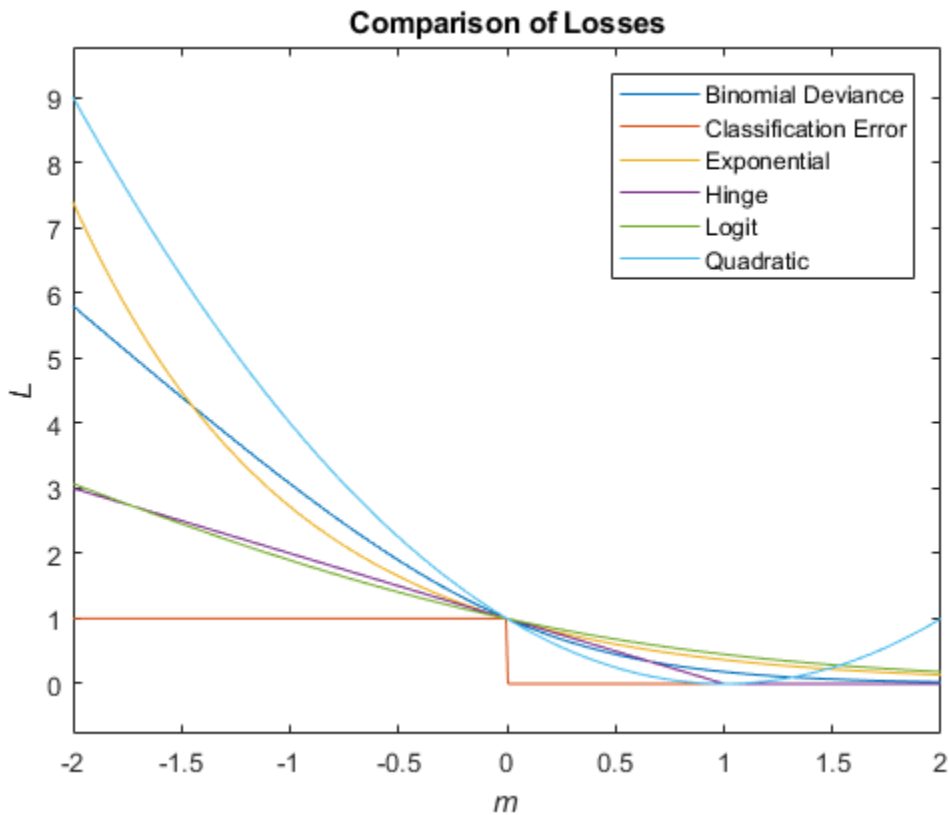


$$L = \sum_{j=1}^n w_j c_j.$$

- Quadratic loss, specified using 'LossFun', 'quadratic'. Its equation is

$$L = \sum_{j=1}^n w_j (1 - m_j)^2.$$

This figure compares some of the loss functions for one observation over  $m$  (some functions are normalized to pass through  $[0,1]$ ).



## See Also

`resubEdge` | `resubLoss` | `resubMargin` | `resubPredict`

# resubLoss

**Class:** ClassificationKNN

Loss of  $k$ -nearest neighbor classifier by resubstitution

## Syntax

```
L = resubLoss mdl
L = resubLoss mdl, Name, Value
```

## Description

`L = resubLoss(mdl)` returns the resubstitution loss, meaning the loss computed for the data that `fitcknn` used to create `mdl`.

`L = resubLoss(mdl, Name, Value)` returns loss statistics with additional options specified by one or more `Name, Value` pair arguments.

## Input Arguments

**mdl** — Classifier model

classifier model object

$k$ -nearest neighbor classifier model, returned as a classifier model object.

Note that using the 'CrossVal', 'Kfold', 'Holdout', 'Leaveout', or 'CVPartition' options results in a model of class `ClassificationPartitionedModel`. You cannot use a partitioned tree for prediction, so this kind of tree does not have a `predict` method.

Otherwise, `mdl` is of class `ClassificationKNN`, and you can use the `predict` method to make predictions.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### LossFun — Loss function

'classiferror' (default) | 'binodeviance' | 'exponential' | 'hinge' | 'logit' | 'mincost' | 'quadratic' | function handle

Loss function, specified as the comma-separated pair consisting of 'LossFun' and a built-in, loss-function name or function handle.

- The following lists available loss functions. Specify one using its corresponding character vector.

Value	Description
'binodeviance'	Binomial deviance
'classiferror'	Classification error
'exponential'	Exponential
'hinge'	Hinge
'logit'	Logistic
'mincost'	Minimal expected misclassification cost (for classification scores that are posterior probabilities)
'quadratic'	Quadratic

'mincost' is appropriate for classification scores that are posterior probabilities  $k$ -nearest neighbor models return posterior probabilities as classification scores by default (see `predict`).

- Specify your own function using function handle notation.

Suppose that  $n$  be the number of observations in  $X$  and  $K$  be the number of distinct classes (`numel(mdl.ClassNames)`). Your function must have this signature

```
lossvalue = lossfun(C,S,W,Cost)
```

where:

- The output argument `lossvalue` is a scalar.
- You choose the function name (`lossfun`).
- `C` is an  $n$ -by- $K$  logical matrix with rows indicating which class the corresponding observation belongs. The column order corresponds to the class order in `mdl.ClassNames`.

Construct `C` by setting  $C(p, q) = 1$  if observation  $p$  is in class  $q$ , for each row. Set all other elements of row  $p$  to 0.

- `S` is an  $n$ -by- $K$  numeric matrix of classification scores. The column order corresponds to the class order in `mdl.ClassNames`. `S` is a matrix of classification scores, similar to the output of `predict`.
- `W` is an  $n$ -by-1 numeric vector of observation weights. If you pass `W`, the software normalizes them to sum to 1.
- `Cost` is a  $K$ -by- $K$  numeric matrix of misclassification costs. For example, `Cost = ones(K) - eye(K)` specifies a cost of 0 for correct classification, and 1 for misclassification.

Specify your function using `'LossFun', @lossfun`.

For more details on loss functions, see “Classification Loss” on page 32-5736.

Data Types: `char` | `function_handle`

## Output Arguments

### **L**

Classification loss on page 32-5736, a scalar. The meaning of the error depends on the values in `weights` and `lossfun`.

## Examples

### Loss Calculation

Construct a  $k$ -nearest neighbor classifier for the Fisher iris data, where  $k = 5$ .

Load the data.

```
load fisheriris
```

Construct a classifier for 5-nearest neighbors.

```
mdl = fitcknn(meas, species, 'NumNeighbors', 5);
```

Examine the resubstitution loss of the classifier.

```
L = resubLoss(mdl)
```

```
L =
```

```
0.0333
```

The classifier predicts incorrect classifications for 1/30 of its training data.

- “Examine Quality of KNN Classifier” on page 18-37
- “Predict Classification Using KNN Classifier” on page 18-38
- “Modify KNN Classifier” on page 18-39

## Definitions

### Classification Loss

Classification loss functions measure the predictive inaccuracy of classification models. When comparing the same type of loss among many models, lower loss indicates a better predictive model.

Suppose that:

- $L$  is the weighted average classification loss.
- $n$  is the sample size.
- For binary classification:
  - $y_j$  is the observed class label. The software codes it as  $-1$  or  $1$  indicating the negative or positive class, respectively.

- $f(X_j)$  is the raw classification score for observation (row)  $j$  of the predictor data  $X$ .
- $m_j = y_j f(X_j)$  is the classification score for classifying observation  $j$  into the class corresponding to  $y_j$ . Positive values of  $m_j$  indicate correct classification and do not contribute much to the average loss. Negative values of  $m_j$  indicate incorrect classification and contribute to the average loss.
- For algorithms that support multiclass classification (that is,  $K \geq 3$ ):
  - $y_j^*$  is a vector of  $K - 1$  zeros, and a 1 in the position corresponding to the true, observed class  $y_j$ . For example, if the true class of the second observation is the third class and  $K = 4$ , then  $y_2^* = [0 \ 0 \ 1 \ 0]'$ . The order of the classes corresponds to the order in the `ClassNames` property of the input model.
  - $f(X_j)$  is the length  $K$  vector of class scores for observation  $j$  of the predictor data  $X$ . The order of the scores corresponds to the order of the classes in the `ClassNames` property of the input model.
  - $m_j = y_j^{*'} f(X_j)$ . Therefore,  $m_j$  is the scalar classification score that the model predicts for the true, observed class.
- The weight for observation  $j$  is  $w_j$ . The software normalizes the observation weights so that they sum to the corresponding prior class probability. The software also normalizes the prior probabilities so they sum to 1. Therefore,

$$\sum_{j=1}^n w_j = 1.$$

The supported loss functions are:

- Binomial deviance, specified using `'LossFun', 'binodeviance'`. Its equation is

$$L = \sum_{j=1}^n w_j \log \{1 + \exp[-2m_j]\}.$$

- Exponential loss, specified using `'LossFun', 'exponential'`. Its equation is

$$L = \sum_{j=1}^n w_j \exp(-m_j).$$

- Classification error, specified using `'LossFun', 'classiferror'`. It is the weighted fraction of misclassified observations, with equation

$$L = \sum_{j=1}^n w_j I\{\hat{y}_j \neq y_j\}.$$

$\hat{y}_j$  is the class label corresponding to the class with the maximal posterior probability.  $I\{x\}$  is the indicator function.

- Hinge loss, specified using 'LossFun', 'hinge'. Its equation is

$$L = \sum_{j=1}^n w_j \max\{0, 1 - m_j\}.$$

- Logit loss, specified using 'LossFun', 'logit'. Its equation is

$$L = \sum_{j=1}^n w_j \log(1 + \exp(-m_j)).$$

- Minimal cost, specified using 'LossFun', 'mincost'. The software computes the weighted minimal cost using this procedure for observations  $j = 1, \dots, n$ :

- 1 Estimate the 1-by- $K$  vector of expected classification costs for observation  $j$

$$\gamma_j = f(X_j)'C.$$

$f(X_j)$  is the column vector of class posterior probabilities for binary and multiclass classification.  $C$  is the cost matrix the input model stores in the property `Cost`.

- 2 For observation  $j$ , predict the class label corresponding to the minimum, expected classification cost:

$$\hat{y}_j = \min_{j=1, \dots, K} (\gamma_j).$$

- 3 Using  $C$ , identify the cost incurred ( $c_j$ ) for making the prediction.

The weighted, average, minimum cost loss is

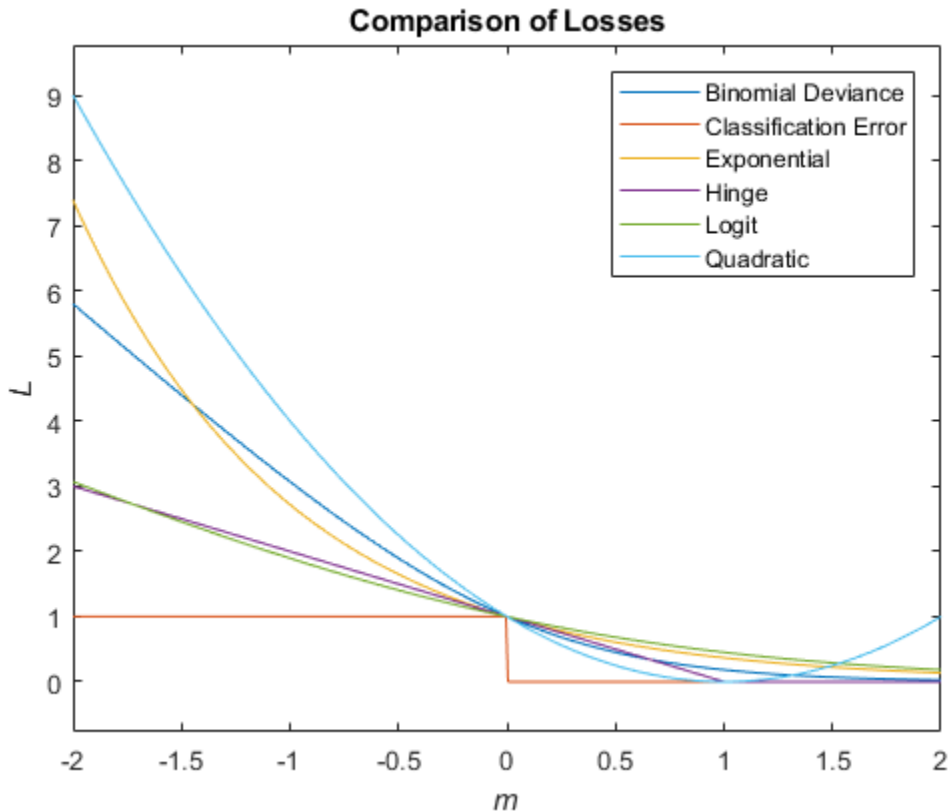
$$L = \sum_{j=1}^n w_j c_j.$$

- Quadratic loss, specified using 'LossFun', 'quadratic'. Its equation is

$$L = \sum_{j=1}^n w_j (1 - m_j)^2.$$



This figure compares some of the loss functions for one observation over  $m$  (some functions are normalized to pass through  $[0,1]$ ).



## True Misclassification Cost

There are two costs associated with KNN classification: the true misclassification cost per class, and the expected misclassification cost per observation.

You can set the true misclassification cost per class in the `Cost` name-value pair when you run `fitcknn`. `Cost(i,j)` is the cost of classifying an observation into class  $j$  if its true class is  $i$ . By default, `Cost(i,j)=1` if  $i \neq j$ , and `Cost(i,j)=0` if  $i=j$ . In other words, the cost is 0 for correct classification, and 1 for incorrect classification.

## Expected Cost

There are two costs associated with KNN classification: the true misclassification cost per class, and the expected misclassification cost per observation. The third output of `predict` is the expected misclassification cost per observation.

Suppose you have `Nobs` observations that you want to classify with a trained classifier `mdl`. Suppose you have `K` classes. You place the observations into a matrix `Xnew` with one observation per row. The command

```
[label,score,cost] = predict(mdl,Xnew)
```

returns, among other outputs, a `cost` matrix of size `Nobs-by-K`. Each row of the `cost` matrix contains the expected (average) cost of classifying the observation into each of the `K` classes. `cost(n,k)` is

$$\sum_{i=1}^K \hat{P}(i | X_{new}(n)) C(k | i),$$

where

- $K$  is the number of classes.
- $\hat{P}(i | X_{new}(n))$  is the posterior probability on page 32-4897 of class  $i$  for observation  $X_{new}(n)$ .
- $C(k | i)$  is the true misclassification cost on page 32-4898 of classifying an observation as  $k$  when its true class is  $i$ .

## See Also

`ClassificationKNN` | `fitcknn` | `resubEdge` | `resubMargin` | `resubPredict`

## Topics

“Examine Quality of KNN Classifier” on page 18-37

“Predict Classification Using KNN Classifier” on page 18-38

“Modify KNN Classifier” on page 18-39

“Classification Using Nearest Neighbors” on page 18-16

## resubLoss

**Class:** ClassificationNaiveBayes

Classification loss for naive Bayes classifiers by resubstitution

### Syntax

```
L = resubLoss(Mdl)
L = resubLoss(Mdl, Name, Value)
```

### Description

`L = resubLoss(Mdl)` returns the in-sample minimum misclassification cost loss (L), which is a scalar representing how well the trained naive Bayes classifier `Mdl` classifies the predictor data stored in `Mdl.X` as compared to the true class labels stored in `Mdl.Y`.

`L = resubLoss(Mdl, Name, Value)` returns the in-sample classification loss with additional options specified by one or more `Name, Value` pair arguments.

### Input Arguments

**Mdl** — Fully trained naive Bayes classifier

ClassificationNaiveBayes model

A fully trained naive Bayes classifier, specified as a `ClassificationNaiveBayes` model trained by `fitcnb`.

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

**LossFun — Loss function**

'classiferror' (default) | 'binodeviance' | 'exponential' | 'hinge' | 'logit' | 'mincost' | 'quadratic' | function handle

Loss function, specified as the comma-separated pair consisting of 'LossFun' and a built-in, loss-function name or function handle.

- The following lists available loss functions. Specify one using its corresponding character vector.

Value	Description
'binodeviance'	Binomial deviance
'classiferror'	Classification error
'exponential'	Exponential
'hinge'	Hinge
'logit'	Logistic
'mincost'	Minimal expected misclassification cost (for classification scores that are posterior probabilities)
'quadratic'	Quadratic

'mincost' is appropriate for classification scores that are posterior probabilities. Naive Bayes models return posterior probabilities as classification scores by default (see `predict`).

- Specify your own function using function handle notation.

Suppose that  $n$  be the number of observations in  $X$  and  $K$  be the number of distinct classes (`numel(Mdl.ClassNames)`,  $Mdl$  is the input model). Your function must have this signature

```
lossvalue = lossfun(C,S,W,Cost)
```

where:

- The output argument `lossvalue` is a scalar.
- You choose the function name (`lossfun`).
- $C$  is an  $n$ -by- $K$  logical matrix with rows indicating which class the corresponding observation belongs. The column order corresponds to the class order in `Mdl.ClassNames`.

Construct  $C$  by setting  $C(p, q) = 1$  if observation  $p$  is in class  $q$ , for each row. Set all other elements of row  $p$  to 0.

- $S$  is an  $n$ -by- $K$  numeric matrix of classification scores. The column order corresponds to the class order in `Mdl.ClassNames`.  $S$  is a matrix of classification scores, similar to the output of `predict`.
- $W$  is an  $n$ -by-1 numeric vector of observation weights. If you pass  $W$ , the software normalizes them to sum to 1.
- $Cost$  is a  $K$ -by- $K$  numeric matrix of misclassification costs. For example,  $Cost = \text{ones}(K) - \text{eye}(K)$  specifies a cost of 0 for correct classification, and 1 for misclassification.

Specify your function using `'LossFun', @lossfun`.

For more details on loss functions, see “Classification Loss” on page 32-5745.

Data Types: `char` | `function_handle`

## Output Arguments

### **L** — Classification loss

scalar

Classification loss, returned as a scalar.  $L$  is a generalization or resubstitution quality measure. Its interpretation depends on the loss function and weighting scheme, but, in general, better classifiers yield smaller loss values.

## Examples

### Determine the Resubstitution Loss of Naive Bayes Classifiers

Load Fisher's iris data set.

```
load fisheriris
X = meas;      % Predictors
Y = species;  % Response
```

Train a naive Bayes classifier. It is good practice to specify the class order. Assume that each predictor is conditionally, normally distributed given its label.

```
Mdl = fitcnb(X,Y,'ClassNames',{'setosa','versicolor','virginica'});
```

Mdl is a trained `ClassificationNaiveBayes` classifier.

Estimate the default resubstitution loss, which is the in-sample minimum misclassification cost.

```
L = resubLoss(Mdl)
```

```
L =
```

```
0.0400
```

The average, in-sample cost of classification is 0.04.

### Determine Resubstitution Classification Error of Naive Bayes Classifiers

Load Fisher's iris data set.

```
load fisheriris
X = meas;    % Predictors
Y = species; % Response
```

Train a naive Bayes classifier. It is good practice to specify the class order. Assume that each predictor is conditionally, normally distributed given its label.

```
Mdl = fitcnb(X,Y,'ClassNames',{'setosa','versicolor','virginica'});
```

Mdl is a trained `ClassificationNaiveBayes` classifier.

Estimate the in-sample proportion of misclassified observations.

```
L = resubLoss(Mdl,'LossFun','classificationerror')
```

```
L =
```

```
0.0400
```

The naive Bayes classifier misclassifies 4% of the training observations.

## Definitions

### Classification Loss

Classification loss functions measure the predictive inaccuracy of classification models. When comparing the same type of loss among many models, lower loss indicates a better predictive model.

Suppose that:

- $L$  is the weighted average classification loss.
- $n$  is the sample size.
- For binary classification:
  - $y_j$  is the observed class label. The software codes it as  $-1$  or  $1$  indicating the negative or positive class, respectively.
  - $f(X_j)$  is the raw classification score for observation (row)  $j$  of the predictor data  $X$ .
  - $m_j = y_j f(X_j)$  is the classification score for classifying observation  $j$  into the class corresponding to  $y_j$ . Positive values of  $m_j$  indicate correct classification and do not contribute much to the average loss. Negative values of  $m_j$  indicate incorrect classification and contribute to the average loss.
- For algorithms that support multiclass classification (that is,  $K \geq 3$ ):
  - $y_j^*$  is a vector of  $K - 1$  zeros, and a 1 in the position corresponding to the true, observed class  $y_j$ . For example, if the true class of the second observation is the third class and  $K = 4$ , then  $y_2^* = [0 \ 0 \ 1 \ 0]'$ . The order of the classes corresponds to the order in the `ClassNames` property of the input model.
  - $f(X_j)$  is the length  $K$  vector of class scores for observation  $j$  of the predictor data  $X$ . The order of the scores corresponds to the order of the classes in the `ClassNames` property of the input model.
  - $m_j = y_j^{*'} f(X_j)$ . Therefore,  $m_j$  is the scalar classification score that the model predicts for the true, observed class.
- The weight for observation  $j$  is  $w_j$ . The software normalizes the observation weights so that they sum to the corresponding prior class probability. The software also normalizes the prior probabilities so they sum to 1. Therefore,

$$\sum_{j=1}^n w_j = 1.$$

The supported loss functions are:

- Binomial deviance, specified using 'LossFun', 'binodeviance'. Its equation is

$$L = \sum_{j=1}^n w_j \log \{1 + \exp[-2m_j]\}.$$

- Exponential loss, specified using 'LossFun', 'exponential'. Its equation is

$$L = \sum_{j=1}^n w_j \exp(-m_j).$$

- Classification error, specified using 'LossFun', 'classiferror'. It is the weighted fraction of misclassified observations, with equation

$$L = \sum_{j=1}^n w_j I\{\hat{y}_j \neq y_j\}.$$

$\hat{y}_j$  is the class label corresponding to the class with the maximal posterior probability.  $I\{x\}$  is the indicator function.

- Hinge loss, specified using 'LossFun', 'hinge'. Its equation is

$$L = \sum_{j=1}^n w_j \max\{0, 1 - m_j\}.$$

- Logit loss, specified using 'LossFun', 'logit'. Its equation is

$$L = \sum_{j=1}^n w_j \log(1 + \exp(-m_j)).$$

- Minimal cost, specified using 'LossFun', 'mincost'. The software computes the weighted minimal cost using this procedure for observations  $j = 1, \dots, n$ :

- 1 Estimate the 1-by- $K$  vector of expected classification costs for observation  $j$

$$\gamma_j = f(X_j)'C.$$



$f(X_j)$  is the column vector of class posterior probabilities for binary and multiclass classification.  $C$  is the cost matrix the input model stores in the property `Cost`.

- 2 For observation  $j$ , predict the class label corresponding to the minimum, expected classification cost:

$$\hat{y}_j = \min_{j=1, \dots, K} (\gamma_j).$$

- 3 Using  $C$ , identify the cost incurred ( $c_j$ ) for making the prediction.

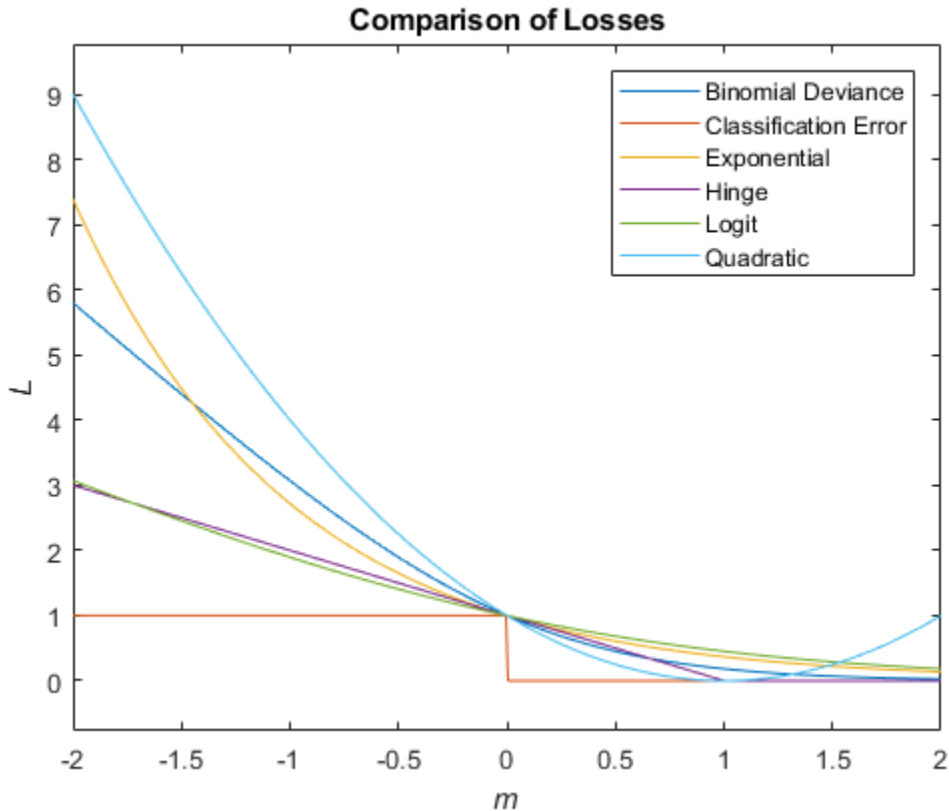
The weighted, average, minimum cost loss is

$$L = \sum_{j=1}^n w_j c_j.$$

- Quadratic loss, specified using `'LossFun', 'quadratic'`. Its equation is

$$L = \sum_{j=1}^n w_j (1 - m_j)^2.$$

This figure compares some of the loss functions for one observation over  $m$  (some functions are normalized to pass through  $[0,1]$ ).



## Posterior Probability

The posterior probability is the probability that an observation belongs in a particular class, given the data.

For naive Bayes, the posterior probability that a classification is  $k$  for a given observation  $(x_1, \dots, x_p)$  is

$$\hat{P}(Y = k | x_1, \dots, x_p) = \frac{P(X_1, \dots, X_p | y = k)\pi(Y = k)}{P(X_1, \dots, X_p)},$$

where:

- $P(X_1, \dots, X_P | y = k)$  is the conditional joint density of the predictors given they are in class  $k$ . `Mdl.DistributionNames` stores the distribution names of the predictors.
- $\pi(Y = k)$  is the class prior probability distribution. `Mdl.Prior` stores the prior distribution.
- $P(X_1, \dots, X_P)$  is the joint density of the predictors. The classes are discrete, so

$$P(X_1, \dots, X_P) = \sum_{k=1}^K P(X_1, \dots, X_P | y = k) \pi(Y = k).$$

## Prior Probability

The prior probability is the believed relative frequency that observations from a class occur in the population for each class.

## References

- [1] Hastie, T., R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*, second edition. Springer, New York, 2008.

## See Also

`ClassificationNaiveBayes` | `CompactClassificationNaiveBayes` | `fitcnb` | `loss` | `predict` | `resubPredict`

## Topics

“Naive Bayes Classification” on page 21-2

## resubLoss

**Class:** ClassificationSVM

Classification loss for support vector machine classifiers by resubstitution

### Syntax

```
L = resubLoss(SVMModel)
L = resubLoss(SVMModel, Name, Value)
```

### Description

`L = resubLoss(SVMModel)` returns the classification loss on page 32-5754 by resubstitution (`L`), the in-sample classification loss, for the support vector machine (SVM) classifier `SVMModel` using the training data stored in `SVMModel.X` and corresponding class labels stored in `SVMModel.Y`.

`L = resubLoss(SVMModel, Name, Value)` returns the classification loss by resubstitution with additional options specified by one or more `Name, Value` pair arguments.

### Input Arguments

**SVMModel** — Full, trained SVM classifier

ClassificationSVM classifier

Full, trained SVM classifier, specified as a `ClassificationSVM` model trained using `fitcsvm`.

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single

quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### LossFun — Loss function

'classiferror' (default) | 'binodeviance' | 'exponential' | 'hinge' | 'logit' | 'mincost' | 'quadratic' | function handle

Loss function, specified as the comma-separated pair consisting of 'LossFun' and a built-in, loss-function name or function handle.

- The following lists available loss functions. Specify one using its corresponding character vector.

Value	Description
'binodeviance'	Binomial deviance
'classiferror'	Classification error
'exponential'	Exponential
'hinge'	Hinge
'logit'	Logistic
'mincost'	Minimal expected misclassification cost (for classification scores that are posterior probabilities)
'quadratic'	Quadratic

'mincost' is appropriate for classification scores that are posterior probabilities.

You can specify to use posterior probabilities as classification scores for SVM models by setting 'FitPosterior', true when you cross-validate the model using `fitcsvm`.

- Specify your own function using function handle notation.

Suppose that  $n$  be the number of observations in  $X$  and  $K$  be the number of distinct classes (`numel(SVMModel.ClassNames)`, `SVMModel` is the input model). Your function must have this signature

```
lossvalue = lossfun(C,S,W,Cost)
```

where:

- The output argument `lossvalue` is a scalar.

- You choose the function name (*lossfun*).
- $C$  is an  $n$ -by- $K$  logical matrix with rows indicating which class the corresponding observation belongs. The column order corresponds to the class order in `SVMModel.ClassNames`.

Construct  $C$  by setting  $C(p, q) = 1$  if observation  $p$  is in class  $q$ , for each row. Set all other elements of row  $p$  to 0.

- $S$  is an  $n$ -by- $K$  numeric matrix of classification scores. The column order corresponds to the class order in `SVMModel.ClassNames`.  $S$  is a matrix of classification scores, similar to the output of `predict`.
- $W$  is an  $n$ -by-1 numeric vector of observation weights. If you pass  $W$ , the software normalizes them to sum to 1.
- $Cost$  is a  $K$ -by- $K$  numeric matrix of misclassification costs. For example,  $Cost = ones(K) - eye(K)$  specifies a cost of 0 for correct classification, and 1 for misclassification.

Specify your function using `'LossFun', @lossfun`.

For more details on loss functions, see “Classification Loss” on page 32-5754.

Data Types: `char` | `function_handle`

## Output Arguments

### **L** — Classification loss

scalar

Classification loss, returned as a scalar.  $L$  is a generalization or resubstitution quality measure. Its interpretation depends on the loss function and weighting scheme, but, in general, better classifiers yield smaller loss values.

## Examples

### Determine the Resubstitution Loss of SVM Classifiers

Load the `ionosphere` data set.

```
load ionosphere
```

Train an SVM classifier. It is good practice to standardize the data.

```
SVMMModel = fitcsvm(X,Y,'ClassNames',{'b','g'},'Standardize',true);
```

SVMMModel is a trained ClassificationSVM classifier. The negative class is 'b' and the positive class is 'g'.

Estimate the resubstitution loss (i.e., the in-sample classification error).

```
L = resubLoss(SVMMModel)
```

```
L =
```

```
0.0570
```

The SVM classifier misclassifies 5.7% of the training sample radar returns.

### Determine the Resubstitution Hinge Loss of SVM Classifiers

Load the ionosphere data set.

```
load ionosphere
```

Train an SVM classifier. It is good practice to standardize the data.

```
SVMMModel = fitcsvm(X,Y,'ClassNames',{'b','g'},'Standardize',true);
```

SVMMModel is a trained ClassificationSVM classifier. The negative class is 'b' and the positive class is 'g'.

Estimate the in-sample hinge loss.

```
L = resubLoss(SVMMModel,'LossFun','Hinge')
```

```
L =
```

```
0.1603
```

The hinge loss is 0.1603. Classifiers with hinge losses close to 0 are desirable.

## Definitions

### Classification Loss

Classification loss functions measure the predictive inaccuracy of classification models. When comparing the same type of loss among many models, lower loss indicates a better predictive model.

Suppose that:

- $L$  is the weighted average classification loss.
- $n$  is the sample size.
- For binary classification:
  - $y_j$  is the observed class label. The software codes it as  $-1$  or  $1$  indicating the negative or positive class, respectively.
  - $f(X_j)$  is the raw classification score for observation (row)  $j$  of the predictor data  $X$ .
  - $m_j = y_j f(X_j)$  is the classification score for classifying observation  $j$  into the class corresponding to  $y_j$ . Positive values of  $m_j$  indicate correct classification and do not contribute much to the average loss. Negative values of  $m_j$  indicate incorrect classification and contribute to the average loss.
- For algorithms that support multiclass classification (that is,  $K \geq 3$ ):
  - $y_j^*$  is a vector of  $K - 1$  zeros, and a 1 in the position corresponding to the true, observed class  $y_j$ . For example, if the true class of the second observation is the third class and  $K = 4$ , then  $y_2^* = [0\ 0\ 1\ 0]'$ . The order of the classes corresponds to the order in the `ClassNames` property of the input model.
  - $f(X_j)$  is the length  $K$  vector of class scores for observation  $j$  of the predictor data  $X$ . The order of the scores corresponds to the order of the classes in the `ClassNames` property of the input model.
  - $m_j = y_j^{*'} f(X_j)$ . Therefore,  $m_j$  is the scalar classification score that the model predicts for the true, observed class.
- The weight for observation  $j$  is  $w_j$ . The software normalizes the observation weights so that they sum to the corresponding prior class probability. The software also normalizes the prior probabilities so they sum to 1. Therefore,



$$\sum_{j=1}^n w_j = 1.$$

The supported loss functions are:

- Binomial deviance, specified using 'LossFun', 'binodeviance'. Its equation is

$$L = \sum_{j=1}^n w_j \log \{1 + \exp[-2m_j]\}.$$

- Exponential loss, specified using 'LossFun', 'exponential'. Its equation is

$$L = \sum_{j=1}^n w_j \exp(-m_j).$$

- Classification error, specified using 'LossFun', 'classiferror'. It is the weighted fraction of misclassified observations, with equation

$$L = \sum_{j=1}^n w_j I\{\hat{y}_j \neq y_j\}.$$

$\hat{y}_j$  is the class label corresponding to the class with the maximal posterior probability.  $I\{x\}$  is the indicator function.

- Hinge loss, specified using 'LossFun', 'hinge'. Its equation is

$$L = \sum_{j=1}^n w_j \max\{0, 1 - m_j\}.$$

- Logit loss, specified using 'LossFun', 'logit'. Its equation is

$$L = \sum_{j=1}^n w_j \log(1 + \exp(-m_j)).$$

- Minimal cost, specified using 'LossFun', 'mincost'. The software computes the weighted minimal cost using this procedure for observations  $j = 1, \dots, n$ :

- 1 Estimate the 1-by- $K$  vector of expected classification costs for observation  $j$

$$\gamma_j = f(X_j)'C.$$

$f(X_j)$  is the column vector of class posterior probabilities for binary and multiclass classification.  $C$  is the cost matrix the input model stores in the property `Cost`.

- 2 For observation  $j$ , predict the class label corresponding to the minimum, expected classification cost:

$$\hat{y}_j = \min_{j=1,\dots,K} (\gamma_j).$$

- 3 Using  $C$ , identify the cost incurred ( $c_j$ ) for making the prediction.

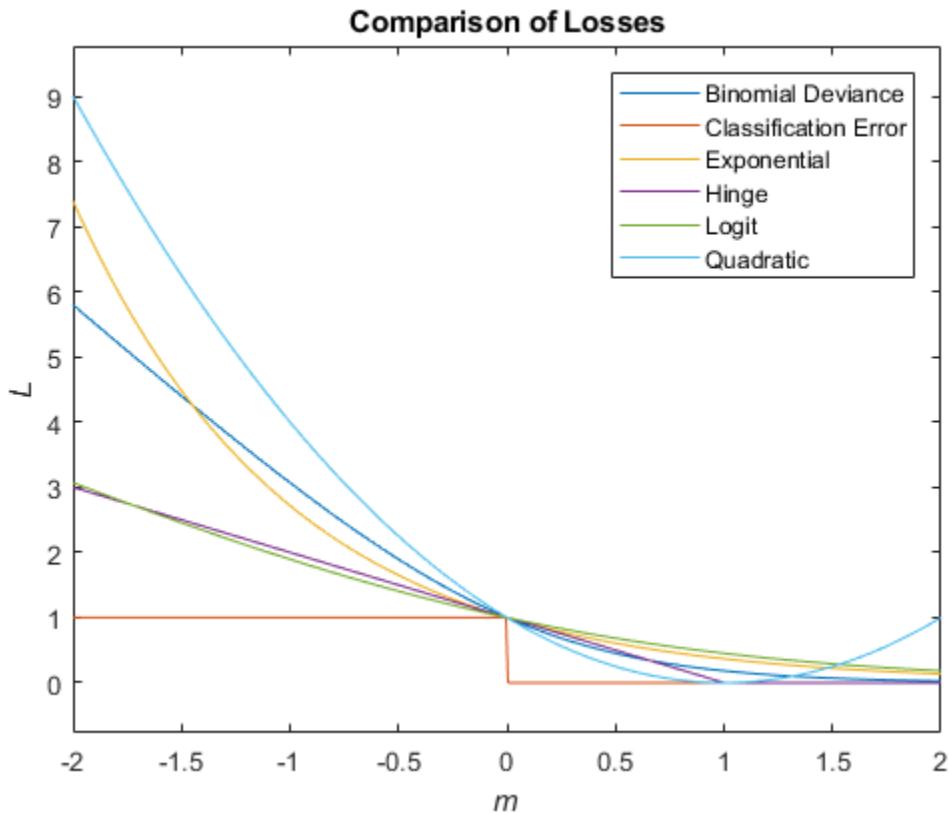
The weighted, average, minimum cost loss is

$$L = \sum_{j=1}^n w_j c_j.$$

- Quadratic loss, specified using `'LossFun', 'quadratic'`. Its equation is

$$L = \sum_{j=1}^n w_j (1 - m_j)^2.$$

This figure compares some of the loss functions for one observation over  $m$  (some functions are normalized to pass through  $[0,1]$ ).



## Score

The SVM classification score for classifying observation  $x$  is the signed distance from  $x$  to the decision boundary ranging from  $-\infty$  to  $+\infty$ . A positive score for a class indicates that  $x$  is predicted to be in that class. A negative score indicates otherwise.

The score for predicting  $x$  into the positive class, also the numerical, predicted response for  $x$ ,  $f(x)$ , is the trained SVM classification function

$$f(x) = \sum_{j=1}^n \alpha_j y_j G(x_j, x) + b,$$

where  $(\alpha_1, \dots, \alpha_n, b)$  are the estimated SVM parameters,  $G(x_j, x)$  is the dot product in the predictor space between  $x$  and the support vectors, and the sum includes the training set observations. The score for predicting  $x$  into the negative class is  $-f(x)$ .

If  $G(x_j, x) = x_j'x$  (the linear kernel), then the score function reduces to  $f(x) = (x/s)'\beta + b$ .

$s$  is the kernel scale and  $\beta$  is the vector of fitted linear coefficients.

## References

- [1] Hastie, T., R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*, second edition. Springer, New York, 2008.

## See Also

`ClassificationSVM` | `CompactClassificationSVM` | `fitsvm` | `loss` | `resubMargin` | `resubPredict`

## resubLoss

**Class:** ClassificationTree

Classification error by resubstitution

### Syntax

```
L = resubLoss(tree)
L = resubLoss(tree,Name,Value)
L = resubLoss(tree,'Subtrees',subtreevector)
[L,se] = resubLoss(tree,'Subtrees',subtreevector)
[L,se,NLeaf] = resubLoss(tree,'Subtrees',subtreevector)
[L,se,NLeaf,bestlevel] = resubLoss(tree,'Subtrees',subtreevector)
[L,...] = resubLoss(tree,'Subtrees',subtreevector,Name,Value)
```

### Description

`L = resubLoss(tree)` returns the resubstitution loss, meaning the loss computed for the data that `fitctree` used to create `tree`.

`L = resubLoss(tree,Name,Value)` returns the loss with additional options specified by one or more `Name, Value` pair arguments. You can specify several name-value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

`L = resubLoss(tree,'Subtrees',subtreevector)` returns a vector of classification errors for the trees in the pruning sequence `subtreevector`.

`[L,se] = resubLoss(tree,'Subtrees',subtreevector)` returns the vector of standard errors of the classification errors.

`[L,se,NLeaf] = resubLoss(tree,'Subtrees',subtreevector)` returns the vector of numbers of leaf nodes in the trees of the pruning sequence.

`[L,se,NLeaf,bestlevel] = resubLoss(tree,'Subtrees',subtreevector)` returns the best pruning level as defined in the `TreeSize` name-value pair. By default,

`bestlevel` is the pruning level that gives loss within one standard deviation of minimal loss.

`[L, ...] = resubLoss(tree, 'Subtrees', subtreevector, Name, Value)` returns loss statistics with additional options specified by one or more `Name, Value` pair arguments. You can specify several name-value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

## Input Arguments

### **tree**

A classification tree constructed by `fitctree`.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### **LossFun — Loss function**

'classiferror' (default) | 'binodeviance' | 'exponential' | 'hinge' | 'logit' | 'mincost' | 'quadratic' | function handle

Loss function, specified as the comma-separated pair consisting of 'LossFun' and a built-in, loss-function name or function handle.

- The following lists available loss functions. Specify one using its corresponding character vector.

Value	Description
'binodeviance'	Binomial deviance
'classiferror'	Classification error
'exponential'	Exponential
'hinge'	Hinge
'logit'	Logistic

Value	Description
'mincost'	Minimal expected misclassification cost (for classification scores that are posterior probabilities)
'quadratic'	Quadratic

'mincost' is appropriate for classification scores that are posterior probabilities. Classification trees return posterior probabilities as classification scores by default (see `predict`).

- Specify your own function using function handle notation.

Suppose that  $n$  be the number of observations in  $X$  and  $K$  be the number of distinct classes (`numel(tree.ClassNames)`). Your function must have this signature

```
lossvalue = lossfun(C,S,W,Cost)
```

where:

- The output argument `lossvalue` is a scalar.
- You choose the function name (`lossfun`).
- $C$  is an  $n$ -by- $K$  logical matrix with rows indicating which class the corresponding observation belongs. The column order corresponds to the class order in `tree.ClassNames`.

Construct  $C$  by setting  $C(p, q) = 1$  if observation  $p$  is in class  $q$ , for each row. Set all other elements of row  $p$  to 0.

- $S$  is an  $n$ -by- $K$  numeric matrix of classification scores. The column order corresponds to the class order in `tree.ClassNames`.  $S$  is a matrix of classification scores, similar to the output of `predict`.
- $W$  is an  $n$ -by-1 numeric vector of observation weights. If you pass  $W$ , the software normalizes them to sum to 1.
- $Cost$  is a  $K$ -by- $K$  numeric matrix of misclassification costs. For example, `Cost = ones(K) - eye(K)` specifies a cost of 0 for correct classification, and 1 for misclassification.

Specify your function using `'LossFun', @lossfun`.

For more details on loss functions, see “Classification Loss” on page 32-5767.

Name, Value arguments associated with pruning subtrees:

**Subtrees — Pruning level**

0 (default) | vector of nonnegative integers | 'all'

Pruning level, specified as the comma-separated pair consisting of 'Subtrees' and a vector of nonnegative integers in ascending order or 'all'.

If you specify a vector, then all elements must be at least 0 and at most `max(tree.PruneList)`. 0 indicates the full, unpruned tree and `max(tree.PruneList)` indicates the completely pruned tree (i.e., just the root node).

If you specify 'all', then `ClassificationTree.resubLoss` operates on all subtrees (i.e., the entire pruning sequence). This specification is equivalent to using `0:max(tree.PruneList)`.

`ClassificationTree.resubLoss` prunes tree to each level indicated in `Subtrees`, and then estimates the corresponding output arguments. The size of `Subtrees` determines the size of some output arguments.

To invoke `Subtrees`, the properties `PruneList` and `PruneAlpha` of `tree` must be nonempty. In other words, grow tree by setting 'Prune', 'on', or by pruning tree using `prune`.

Example: 'Subtrees', 'all'

**TreeSize — Tree size**

'se' (default) | 'min'

Tree size, specified as the comma-separated pair consisting of 'TreeSize' and one of the following character vectors:

- 'se' — `loss` returns the highest pruning level with loss within one standard deviation of the minimum ( $L+se$ , where  $L$  and  $se$  relate to the smallest value in `Subtrees`).
- 'min' — `loss` returns the element of `Subtrees` with smallest loss, usually the smallest element of `Subtrees`.



## Output Arguments

### **L**

Classification loss on page 32-5767, a vector the length of `Subtrees`. The meaning of the error depends on the values in `Weights` and `LossFun`.

### **se**

Standard error of loss, a vector the length of `Subtrees`.

### **NLeaf**

Number of leaves (terminal nodes) in the pruned subtrees, a vector the length of `Subtrees`.

### **bestlevel**

A scalar whose value depends on `TreeSize`:

- `TreeSize = 'se'` — `loss` returns the highest pruning level with loss within one standard deviation of the minimum ( $L+se$ , where `L` and `se` relate to the smallest value in `Subtrees`).
- `TreeSize = 'min'` — `loss` returns the element of `Subtrees` with smallest loss, usually the smallest element of `Subtrees`.

## Examples

### Compute the In-Sample Classification Error

Compute the resubstitution classification error for the `ionosphere` data.

```
load ionosphere
tree = fitctree(X,Y);
L = resubLoss(tree)
```

`L =`

```
0.0114
```

### Examine the Classification Error for Each Subtree

Unpruned decision trees tend to overfit. One way to balance model complexity and out-of-sample performance is to prune a tree (or restrict its growth) so that in-sample and out-of-sample performance are satisfactory.

Load Fisher's iris data set. Partition the data into training (50%) and validation (50%) sets.

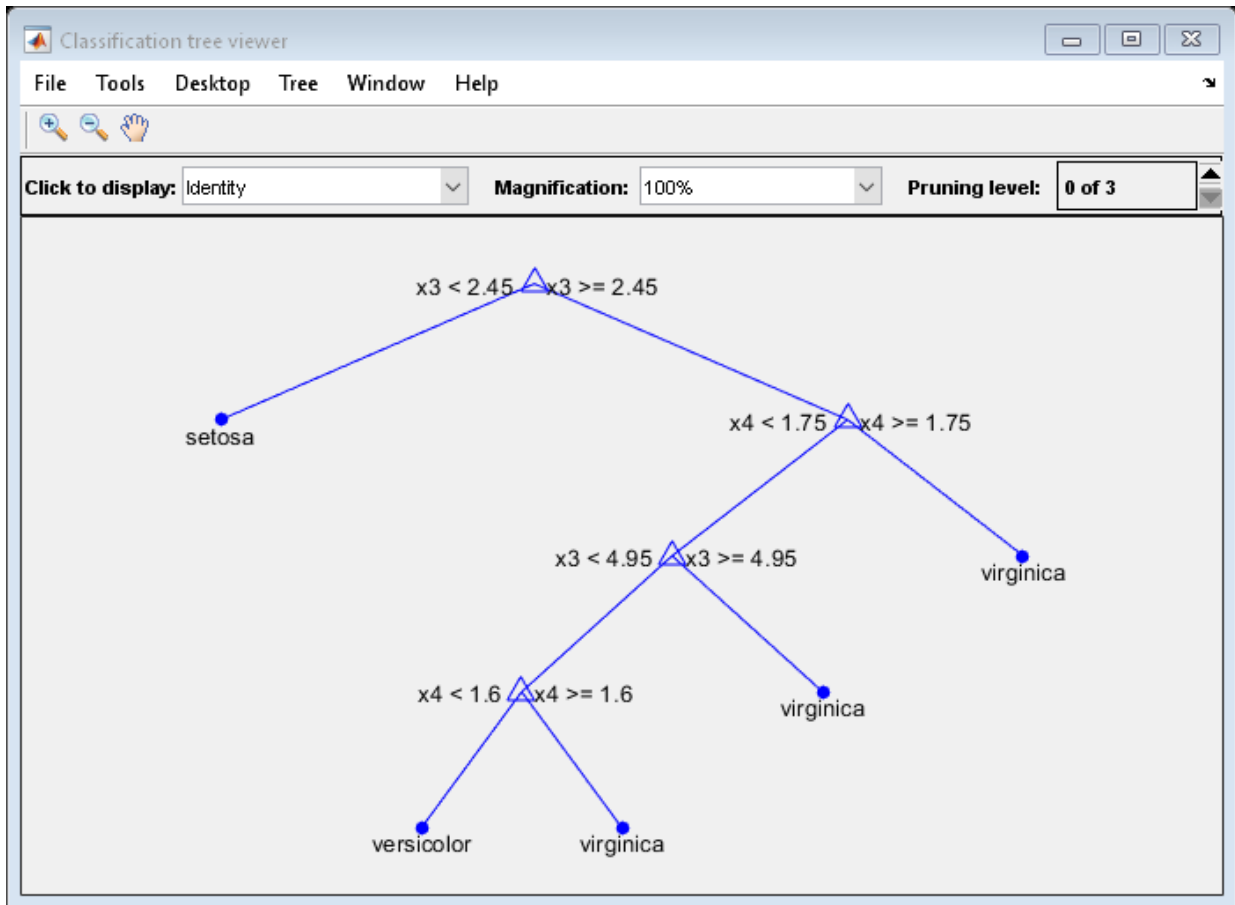
```
load fisheriris
n = size(meas,1);
rng(1) % For reproducibility
idxTrn = false(n,1);
idxTrn(randsample(n,round(0.5*n))) = true; % Training set logical indices
idxVal = idxTrn == false; % Validation set logical indices
```

Grow a classification tree using the training set.

```
Mdl = fitctree(meas(idxTrn,:), species(idxTrn));
```

View the classification tree.

```
view(Mdl, 'Mode', 'graph');
```



The classification tree has four pruning levels. Level 0 is the full, unpruned tree (as displayed). Level 3 is just the root node (i.e., no splits).

Examine the training sample classification error for each subtree (or pruning level) excluding the highest level.

```
m = max(Mdl.PruneList) - 1;
trnLoss = resubLoss(Mdl, 'SubTrees', 0:m)
```

```
trnLoss =
```

```
0.0267
0.0533
0.3067
```

- The full, unpruned tree misclassifies about 2.7% of the training observations.
- The tree pruned to level 1 misclassifies about 5.3% of the training observations.
- The tree pruned to level 2 (i.e., a stump) misclassifies about 30.6% of the training observations.

Examine the validation sample classification error at each level excluding the highest level.

```
valLoss = loss(Mdl, meas(idxVal, :), species(idxVal), 'SubTrees', 0:m)
```

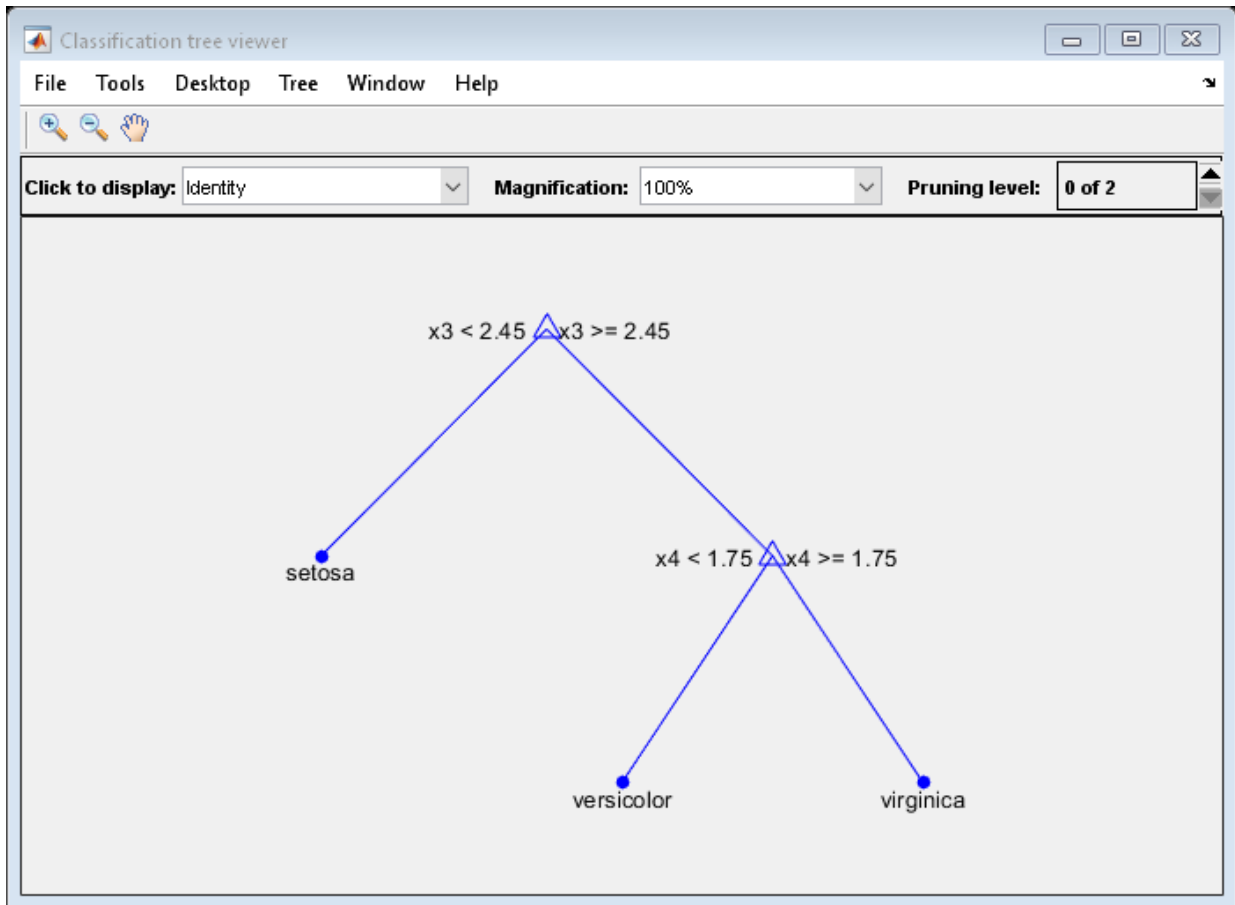
```
valLoss =

    0.0369
    0.0237
    0.3067
```

- The full, unpruned tree misclassifies about 3.7% of the validation observations.
- The tree pruned to level 1 misclassifies about 2.4% of the validation observations.
- The tree pruned to level 2 (i.e., a stump) misclassifies about 30.7% of the validation observations.

To balance model complexity and out-of-sample performance, consider pruning `Mdl` to level 1.

```
pruneMdl = prune(Mdl, 'Level', 1);
view(pruneMdl, 'Mode', 'graph')
```



## Definitions

### Classification Loss

Classification loss functions measure the predictive inaccuracy of classification models. When comparing the same type of loss among many models, lower loss indicates a better predictive model.

Suppose that:

- $L$  is the weighted average classification loss.
- $n$  is the sample size.
- For binary classification:
  - $y_j$  is the observed class label. The software codes it as  $-1$  or  $1$  indicating the negative or positive class, respectively.
  - $f(X_j)$  is the raw classification score for observation (row)  $j$  of the predictor data  $X$ .
  - $m_j = y_j f(X_j)$  is the classification score for classifying observation  $j$  into the class corresponding to  $y_j$ . Positive values of  $m_j$  indicate correct classification and do not contribute much to the average loss. Negative values of  $m_j$  indicate incorrect classification and contribute to the average loss.
- For algorithms that support multiclass classification (that is,  $K \geq 3$ ):
  - $y_j^*$  is a vector of  $K - 1$  zeros, and a 1 in the position corresponding to the true, observed class  $y_j$ . For example, if the true class of the second observation is the third class and  $K = 4$ , then  $y_2^* = [0 \ 0 \ 1 \ 0]'$ . The order of the classes corresponds to the order in the `ClassNames` property of the input model.
  - $f(X_j)$  is the length  $K$  vector of class scores for observation  $j$  of the predictor data  $X$ . The order of the scores corresponds to the order of the classes in the `ClassNames` property of the input model.
  - $m_j = y_j^{*'} f(X_j)$ . Therefore,  $m_j$  is the scalar classification score that the model predicts for the true, observed class.
- The weight for observation  $j$  is  $w_j$ . The software normalizes the observation weights so that they sum to the corresponding prior class probability. The software also normalizes the prior probabilities so they sum to 1. Therefore,

$$\sum_{j=1}^n w_j = 1.$$

The supported loss functions are:

- Binomial deviance, specified using `'LossFun', 'binodeviance'`. Its equation is

$$L = \sum_{j=1}^n w_j \log \{1 + \exp[-2m_j]\}.$$

- Exponential loss, specified using `'LossFun', 'exponential'`. Its equation is

$$L = \sum_{j=1}^n w_j \exp(-m_j).$$

- Classification error, specified using 'LossFun', 'classiferror'. It is the weighted fraction of misclassified observations, with equation

$$L = \sum_{j=1}^n w_j I\{\hat{y}_j \neq y_j\}.$$

$\hat{y}_j$  is the class label corresponding to the class with the maximal posterior probability.  $I\{x\}$  is the indicator function.

- Hinge loss, specified using 'LossFun', 'hinge'. Its equation is

$$L = \sum_{j=1}^n w_j \max\{0, 1 - m_j\}.$$

- Logit loss, specified using 'LossFun', 'logit'. Its equation is

$$L = \sum_{j=1}^n w_j \log(1 + \exp(-m_j)).$$

- Minimal cost, specified using 'LossFun', 'mincost'. The software computes the weighted minimal cost using this procedure for observations  $j = 1, \dots, n$ :

- 1 Estimate the 1-by- $K$  vector of expected classification costs for observation  $j$

$$\gamma_j = f(X_j)'C.$$

$f(X_j)$  is the column vector of class posterior probabilities for binary and multiclass classification.  $C$  is the cost matrix the input model stores in the property `Cost`.

- 2 For observation  $j$ , predict the class label corresponding to the minimum, expected classification cost:

$$\hat{y}_j = \min_{j=1, \dots, K}(\gamma_j).$$

- 3 Using  $C$ , identify the cost incurred ( $c_j$ ) for making the prediction.

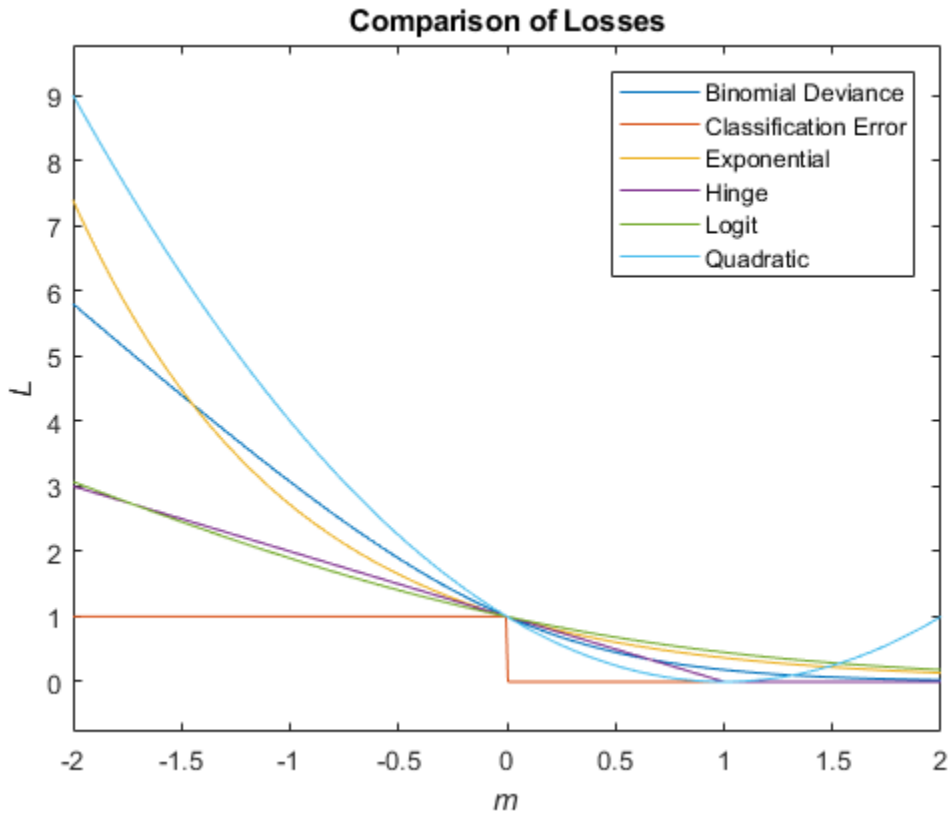
The weighted, average, minimum cost loss is

$$L = \sum_{j=1}^n w_j c_j.$$

- Quadratic loss, specified using 'LossFun', 'quadratic'. Its equation is

$$L = \sum_{j=1}^n w_j (1 - m_j)^2.$$

This figure compares some of the loss functions for one observation over  $m$  (some functions are normalized to pass through  $[0,1]$ ).





## True Misclassification Cost

There are two costs associated with classification: the true misclassification cost per class, and the expected misclassification cost per observation.

You can set the true misclassification cost per class in the `Cost` name-value pair when you create the classifier using the `fitctree` method.  $\text{Cost}(i, j)$  is the cost of classifying an observation into class  $j$  if its true class is  $i$ . By default,  $\text{Cost}(i, j) = 1$  if  $i \neq j$ , and  $\text{Cost}(i, j) = 0$  if  $i = j$ . In other words, the cost is 0 for correct classification, and 1 for incorrect classification.

## Expected Misclassification Cost

There are two costs associated with classification: the true misclassification cost per class, and the expected misclassification cost per observation.

Suppose you have `Nobs` observations that you want to classify with a trained classifier. Suppose you have  $K$  classes. You place the observations into a matrix `Xnew` with one observation per row.

The expected cost matrix `CE` has size `Nobs`-by- $K$ . Each row of `CE` contains the expected (average) cost of classifying the observation into each of the  $K$  classes.  $\text{CE}(n, k)$  is

$$\sum_{i=1}^K \hat{P}(i | X_{\text{new}}(n)) C(k | i),$$

where

- $K$  is the number of classes.
- $\hat{P}(i | X_{\text{new}}(n))$  is the posterior probability of class  $i$  for observation  $X_{\text{new}}(n)$ .
- $C(k | i)$  is the true misclassification cost of classifying an observation as  $k$  when its true class is  $i$ .

## See Also

`fitctree` | `loss` | `resubEdge` | `resubMargin` | `resubPredict`

## resubLoss

**Class:** RegressionEnsemble

Regression error by resubstitution

### Syntax

```
L = resubLoss(ens)
L = resubLoss(ens,Name,Value)
```

### Description

`L = resubLoss(ens)` returns the resubstitution loss, meaning the mean squared error computed for the data that `fitrensemble` used to create `ens`.

`L = resubLoss(ens,Name,Value)` calculates loss with additional options specified by one or more `Name,Value` pair arguments. You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

### Input Arguments

#### **ens**

A regression ensemble created with `fitrensemble`.

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

**learners**

Indices of weak learners in the ensemble ranging from 1 to NumTrained. resubLoss uses only these learners for calculating loss.

**Default:** 1:NumTrained

**lossfun**

Function handle for loss function, or 'mse', meaning mean squared error. If you pass a function handle fun, resubLoss calls it as

```
FUN(Y, Yfit, W)
```

where Y, Yfit, and W are numeric vectors of the same length. Y is the observed response, Yfit is the predicted response, and W is the observation weights.

**Default:** 'mse'

**mode**

Character vector representing the meaning of the output L:

- 'ensemble' — L is a scalar value, the loss for the entire ensemble.
- 'individual' — L is a vector with one element per trained learner.
- 'cumulative' — L is a vector in which element J is obtained by using learners 1:J from the input list of learners.

**Default:** 'ensemble'

## Output Arguments

**L**

Loss, by default the mean squared error. L can be a vector, and can mean different things, depending on the name-value pair settings.

## Examples

### Estimate Resubstitution Loss

Find the mean-squared difference between resubstitution predictions and training data.

Load the `carsmall` data set and select horsepower and vehicle weight as predictors.

```
load carsmall
X = [Horsepower Weight];
```

Train an ensemble of regression trees, and find the mean-squared difference of predictions from the training data.

```
ens = fitensemble(X,MPG);
MSE = resubLoss(ens)
```

```
MSE = 0.5836
```

## See Also

[loss](#) | [resubLoss](#) | [resubPredict](#)

# resubLoss

**Class:** RegressionGP

Resubstitution loss for a trained Gaussian process regression model

## Syntax

```
L = resubLoss(gprMdl)
L = resubLoss(gprMdl, Name, Value)
```

## Description

`L = resubLoss(gprMdl)` returns the resubstitution mean squared error for the Gaussian process regression (GPR) model, `gprMdl`.

`L = resubLoss(gprMdl, Name, Value)` returns the resubstitution loss for the GPR model, `gprMdl`, with additional options specified by one or more `Name, Value` pair arguments. For example, you can specify a custom loss function or the observation weights.

## Input Arguments

**gprMdl** — Gaussian process regression model

RegressionGP object

Gaussian process regression model, specified as a RegressionGP object.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

**lossfun — Loss function**

'mse' (default) | function handle

Loss function for the trained GPR model, specified as 'mse' or a function handle. 'mse' stands for the mean squared error.

If you pass a function handle, say `fun`, `resubLoss` calls it as `: fun(Y, Ypred, W)`, where `Y`, `Ypred`, and `W` are numeric vectors of length  $n$ , and  $n$  is the number of observations in the training data. `Y` is the observed response, `Ypred` is the predicted response, and `W` is the observation weights.

Example: `'lossfun', Fct` calls the loss function `Fct`.

Data Types: `char` | `function_handle`

**weights — Observation weights**

vector of 1s (default) |  $n$ -by-1 vector

Observation weights, specified as an  $n$ -by-1 vector, where  $n$  is the number of observations in the training data. By default, weight of each observation is 1.

Data Types: `double` | `single`

## Output Arguments

**L — Resubstitution error**

scalar value

Resubstitution error for the GPR model, returned as a scalar value.

## Examples

**Compute the Resubstitution Loss**

This example uses "Housing" data set [1] from the UCI machine learning archive [2] described in <http://archive.ics.uci.edu/ml/datasets/Housing>. Download the data and save it in your current directory as a data file named `housing.data`.

The dataset has 506 observations. The first 13 columns contain the predictor values and the last column contains the response values. The goal is to predict the median value of owner-occupied homes in the Boston suburb area as a function of 13 predictors.

Load the data and define the response vector and predictor matrix.

```
load('housing.data');  
X = housing(:,1:13);  
y = housing(:,end);
```

Fit a GPR model using subset of regressors ('sr') approximation method with Matern 3/2 ('Matern32') kernel function. Predict using the fully independent conditional ('fic') method.

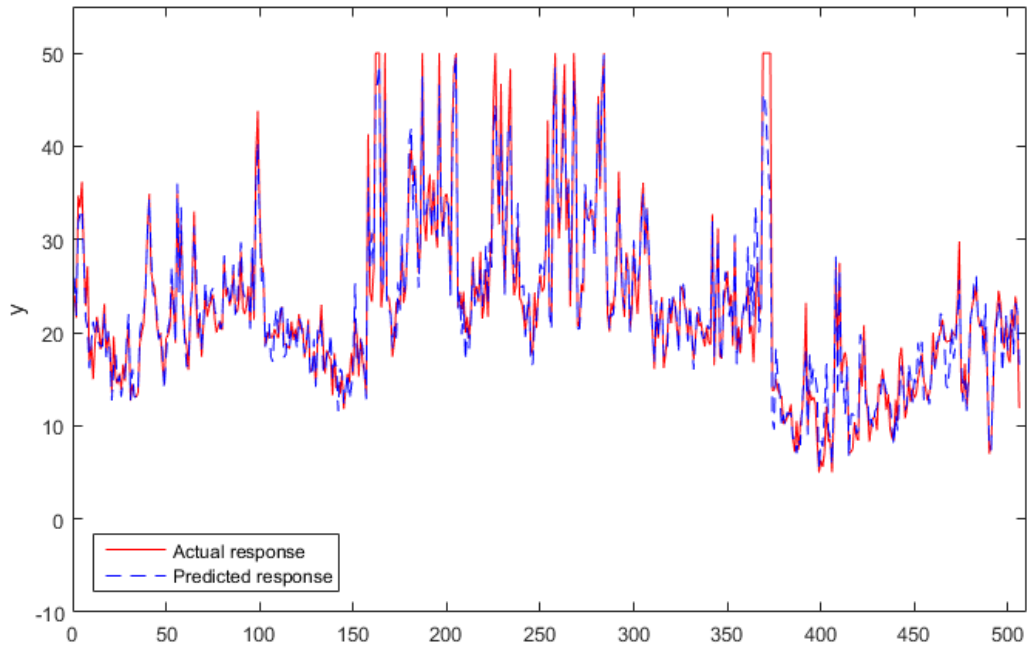
```
gprMdl = fitrgp(X,y,'KernelFunction','Matern32',...  
'FitMethod','sr','PredictMethod','fic');
```

Compute the resubstitution predictions.

```
ypred = resubPredict(gprMdl);
```

Plot the predicted response values along with the actual response values.

```
plot(y,'r'); % Plot original response values  
hold on;  
plot(ypred,'b--'); % Plot predicted response values  
ylabel('y');  
legend('Actual response','Predicted response','Location','SouthWest');  
axis([0 510 -10 55]);  
hold off
```



Compute the resubstitution loss.

```
L = resubLoss(gprMdl)
```

```
L =
```

```
4.8478
```

Manually compute the regression loss.

```
n = length(y);
```

```
L = (y-ypred)'*(y-ypred)/n
```

```
L =
```

```
4.8478
```



## Compute Custom Resubstitution Loss

Load the sample data and store in a table.

```
load fisheriris
tbl = table(meas(:,1),meas(:,2),meas(:,3),meas(:,4),species,...
'VariableNames',{'meas1','meas2','meas3','meas4','species'});
```

Fit a GPR model using the first measurement as the response and the other variables as the predictors.

```
mdl = fitrgp(tbl,'meas1');
```

Predict the responses using the trained model.

```
ypred = predict(mdl,tbl);
```

Compute the mean absolute error.

```
n = height(tbl);
y = tbl.meas1;
fun = @(y,ypred,w) sum(abs(y-ypred))/n;
L = resubLoss(mdl,'lossfun',fun)
```

```
L =
```

```
0.2345
```

## Alternatives

To compute the regression error for new data, use `loss`.

## References

- [1] Harrison, D. and D.L., Rubinfeld. "Hedonic prices and the demand for clean air." *J. Environ. Economics & Management*. Vol.5, 1978, pp. 81-102.
- [2] Lichman, M. UCI Machine Learning Repository, Irvine, CA: University of California, School of Information and Computer Science, 2013. <http://archive.ics.uci.edu/ml>.

## See Also

`RegressionGP` | `fitrgp` | `loss` | `resubPredict`

**Introduced in R2015b**

# resubLoss

**Class:** RegressionSVM

Resubstitution loss for support vector machine regression model

## Syntax

```
L = resubLoss mdl
L = resubLoss mdl, Name, Value
```

## Description

`L = resubLoss(mdl)` returns the resubstitution loss for the support vector machine (SVM) regression model `mdl`, using the training data stored in `mdl.X` and corresponding response values stored in `mdl.Y`.

`L = resubLoss(mdl, Name, Value)` returns the resubstitution loss with additional options specified by one or more `Name, Value` pair arguments. For example, you can specify the loss function or observation weights.

## Input Arguments

**mdl** — Full, trained SVM regression model

RegressionSVM model

Full, trained SVM regression model, specified as a RegressionSVM model returned by `fitrsvm`.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

**LossFun — Loss function**

'mse' (default) | 'epsiloninsensitive' | function handle

Loss function, specified as the comma-separated pair consisting of 'LossFun' and a function handle or character vector.

- The following lists available loss functions. Specify one using its corresponding character vector.

Value	Loss Function
'mse'	“Mean Squared Error” on page 32-5784
'epsiloninsensitive'	“Epsilon-Insensitive Loss Function” on page 32-5785

- Specify your own function using function handle notation.

Suppose that  $n = \text{size}(X,1)$  is the sample size. Your function must have the signature `lossvalue = lossfun(Y,Yfit,W)`, where:

- The output argument `lossvalue` is a numeric value.
- You choose the function name (*lossfun*).
- $Y$  is an  $n$ -by-1 numeric vector of observed response values.
- $Yfit$  is an  $n$ -by-1 numeric vector of predicted response values, calculated using the corresponding predictor values in  $X$  (similar to the output of `predict`).
- $W$  is an  $n$ -by-1 numeric vector of observation weights.

Specify your function using 'LossFun',@*lossfun*.

Example: 'LossFun','epsiloninsensitive'

Data Types: char | function\_handle

**Weights — Observation weights**

ones(size(X,1)) (default) | numeric vector

Observation weights, specified as the comma-separated pair consisting of 'Weights' and a numeric vector. Weights must be the same length as the number of rows in  $X$ . The software weighs the observations in each row of  $X$  using the corresponding weight value in `Weights`.

Data Types: single | double

## Output Arguments

### L — Resubstitution loss

scalar value

Resubstitution loss, returned as a scalar value.

The resubstitution loss is the loss calculated between the response training data and the model's predicted response values based on the input training data.

Resubstitution loss can be an overly optimistic estimate of the predictive error on new data. If the resubstitution loss is high, the model's predictions are not likely to be very good. However, having a low resubstitution loss does not guarantee good predictions for new data.

To better assess the predictive accuracy of your model, cross validate the model using `crossval`.

## Examples

### Resubstitution Loss for SVM Regression Model

This example shows how to train an SVM regression model, then calculate the resubstitution loss using mean square error (MSE) and epsilon-insensitive loss.

This example uses the abalone data from the UCI Machine Learning Repository. Download the data and save it in your current directory with the name 'abalone.data'.

Read the data into a table.

```
tbl = readtable('abalone.data', 'Filetype', 'text', 'ReadVariableNames', false);  
rng default % for reproducibility
```

The sample data contains 4177 observations. All of the predictor variables are continuous except for `sex`, which is a categorical variable with possible values 'M' (for males), 'F' (for females), and 'I' (for infants). The goal is to predict the number of rings on the abalone, and thereby determine its age, using physical measurements.

Train an SVM regression model to the data, using a Gaussian kernel function with an automatic kernel scale. Standardize the data.

```
mdl = fitrsvm(tbl, 'Var9', 'KernelFunction', 'gaussian', 'KernelScale', 'auto', 'Standardize')
```

Calculate the resubstitution loss using mean square error (MSE).

```
mse_loss = resubLoss(mdl)
```

```
mse_loss =
```

```
4.0603
```

Calculate the epsilon-insensitive loss.

```
eps_loss = resubLoss(mdl, 'LossFun', 'epsiloninsensitive')
```

```
eps_loss =
```

```
1.1027
```

## Definitions

### Mean Squared Error

The weighted mean squared error is calculated as follows:

$$\text{mse} = \frac{\sum_{j=1}^n w_j (f(x_j) - y_j)^2}{\sum_{j=1}^n w_j},$$

where:

- $n$  is the number of rows of data
- $x_j$  is the  $j$ th row of data
- $y_j$  is the true response to  $x_j$
- $f(x_j)$  is the response prediction of the SVM regression model `mdl` to  $x_j$

- $w$  is the vector of weights.

The weights in  $w$  are all equal to one by default. You can specify different values for weights using the 'weights' name-value pair argument. If you specify weights, each value is divided by the sum of all weights, such that the normalized weights add to one.

## Epsilon-Insensitive Loss Function

The epsilon-insensitive loss function ignores errors that are within the distance epsilon ( $\epsilon$ ) of the function value. It is formally described as:

$$Loss_{\epsilon} = \begin{cases} 0, & \text{if } |y - f(x)| \leq \epsilon \\ |y - f(x)| - \epsilon, & \text{otherwise.} \end{cases}$$

The mean epsilon-insensitive loss is calculated as follows:

$$Loss = \frac{\sum_{j=1}^n w_j \max(0, |y_j - f(x_j)| - \epsilon)}{\sum_{j=1}^n w_j},$$

## References

- [1] Nash, W.J., T. L. Sellers, S. R. Talbot, A. J. Cawthorn, and W. B. Ford. *The Population Biology of Abalone (Haliotis species) in Tasmania. I. Blacklip Abalone (H. rubra) from the North Coast and Islands of Bass Strait*, Sea Fisheries Division, Technical Report No. 48, 1994.
- [2] Waugh, S. *Extending and benchmarking Cascade-Correlation*, Ph.D. thesis, Computer Science Department, University of Tasmania, 1995.
- [3] Clark, D., Z. Schreter, A. Adams. *A Quantitative Comparison of Dystal and Backpropagation*, submitted to the Australian Conference on Neural Networks, 1996.
- [4] Lichman, M. *UCI Machine Learning Repository*, [<http://archive.ics.uci.edu/ml>]. Irvine, CA: University of California, School of Information and Computer Science.

## See Also

`RegressionSVM` | `fitrsvm` | `loss` | `resubPredict`

**Introduced in R2015b**



## resubLoss

**Class:** RegressionTree

Regression error by resubstitution

### Syntax

```
L = resubLoss(tree)
L = resubLoss(tree,Name,Value)
L = resubLoss(tree,'Subtrees',subtreevector)
[L,se] = resubLoss(tree,'Subtrees',subtreevector)
[L,se,NLeaf] = resubLoss(tree,'Subtrees',subtreevector)
[L,se,NLeaf,bestlevel] = resubLoss(tree,'Subtrees',subtreevector)
[L,...] = resubLoss(tree,'Subtrees',subtreevector,Name,Value)
```

### Description

`L = resubLoss(tree)` returns the resubstitution loss, meaning the loss computed for the data that `fitrtree` used to create `tree`.

`L = resubLoss(tree,Name,Value)` returns the loss with additional options specified by one or more `Name, Value` pair arguments. You can specify several name-value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

`L = resubLoss(tree,'Subtrees',subtreevector)` returns a vector of mean squared errors for the trees in the pruning sequence `subtreevector`.

`[L,se] = resubLoss(tree,'Subtrees',subtreevector)` returns the vector of standard errors of the classification errors.

`[L,se,NLeaf] = resubLoss(tree,'Subtrees',subtreevector)` returns the vector of numbers of leaf nodes in the trees of the pruning sequence.

`[L,se,NLeaf,bestlevel] = resubLoss(tree,'Subtrees',subtreevector)` returns the best pruning level as defined in the `TreeSize` name-value pair. By default,

`bestlevel` is the pruning level that gives loss within one standard deviation of minimal loss.

`[L, ...] = resubLoss(tree, 'Subtrees', subtreevector, Name, Value)` returns loss statistics with additional options specified by one or more `Name, Value` pair arguments. You can specify several name-value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

## Input Arguments

### **tree** — Regression tree

`RegressionTree` model object

A regression tree (`RegressionTree` model object) constructed using `fitrtree`.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### **LossFun** — Loss function

'mse' (default) | function handle

Function handle or 'mse' meaning mean squared error.

You can write your own loss function in the syntax described in “Loss Functions” on page 32-5794.

Data Types: `char` | `function_handle`

`Name, Value` arguments associated with pruning subtrees:

### **Subtrees** — Pruning level

0 (default) | vector of nonnegative integers | 'all'

Pruning level, specified as the comma-separated pair consisting of 'Subtrees' and a vector of nonnegative integers in ascending order or 'all'.

If you specify a vector, then all elements must be at least 0 and at most `max(tree.PruneList)`. 0 indicates the full, unpruned tree and `max(tree.PruneList)` indicates the completely pruned tree (i.e., just the root node).

If you specify 'all', then `RegressionTree.resubLoss` operates on all subtrees (i.e., the entire pruning sequence). This specification is equivalent to using `0:max(tree.PruneList)`.

`RegressionTree.resubLoss` prunes tree to each level indicated in `Subtrees`, and then estimates the corresponding output arguments. The size of `Subtrees` determines the size of some output arguments.

To invoke `Subtrees`, the properties `PruneList` and `PruneAlpha` of tree must be nonempty. In other words, grow tree by setting 'Prune', 'on', or by pruning tree using `prune`.

Example: 'Subtrees', 'all'

### **TreeSize — Tree size**

'se' (default) | 'min'

One of the following:

- 'se' — loss returns the highest pruning level with loss within one standard deviation of the minimum ( $L + se$ , where  $L$  and  $se$  relate to the smallest value in `Subtrees`).
- 'min' — loss returns the element of `Subtrees` with smallest loss, usually the smallest element of `Subtrees`.

Data Types: char

## Output Arguments

### **L — Regression loss**

numeric vector of positive values

Regression loss (mean squared error), a vector the length of `Subtrees`. The meaning of the error depends on the values in `Weights` and `LossFun`.

**se** — Standard error of loss

numeric vector of positive values

Standard error of loss, a vector the length of `Subtrees`.

**nLeaf** — Number of leaves

numeric vector of nonnegative integers

Number of leaves (terminal nodes) in the pruned subtrees, a vector the length of `Subtrees`.

**bestlevel** — Optimal pruning level

nonnegative numeric scalar

A scalar whose value depends on `TreeSize`:

- `TreeSize = 'se'` — `loss` returns the highest pruning level with loss within one standard deviation of the minimum ( $L + se$ , where  $L$  and  $se$  relate to the smallest value in `Subtrees`).
- `TreeSize = 'min'` — `loss` returns the element of `Subtrees` with smallest loss, usually the smallest element of `Subtrees`.

## Examples

**Compute the In-Sample MSE**

Load the `carsmall` data set. Consider `Displacement`, `Horsepower`, and `Weight` as predictors of the response `MPG`.

```
load carsmall
X = [Displacement Horsepower Weight];
```

Grow a regression tree using all observations.

```
Mdl = fitrtree(X,MPG);
```

Compute the resubstitution MSE.

```
resubLoss(Mdl)
```

```
ans =  
  
    4.8952
```

## Examine the MSE for Each Subtree

Unpruned decision trees tend to overfit. One way to balance model complexity and out-of-sample performance is to prune a tree (or restrict its growth) so that in-sample and out-of-sample performance are satisfactory.

Load the `carsmall` data set. Consider Displacement, Horsepower, and Weight as predictors of the response MPG.

```
load carsmall  
X = [Displacement Horsepower Weight];  
Y = MPG;
```

Partition the data into training (50%) and validation (50%) sets.

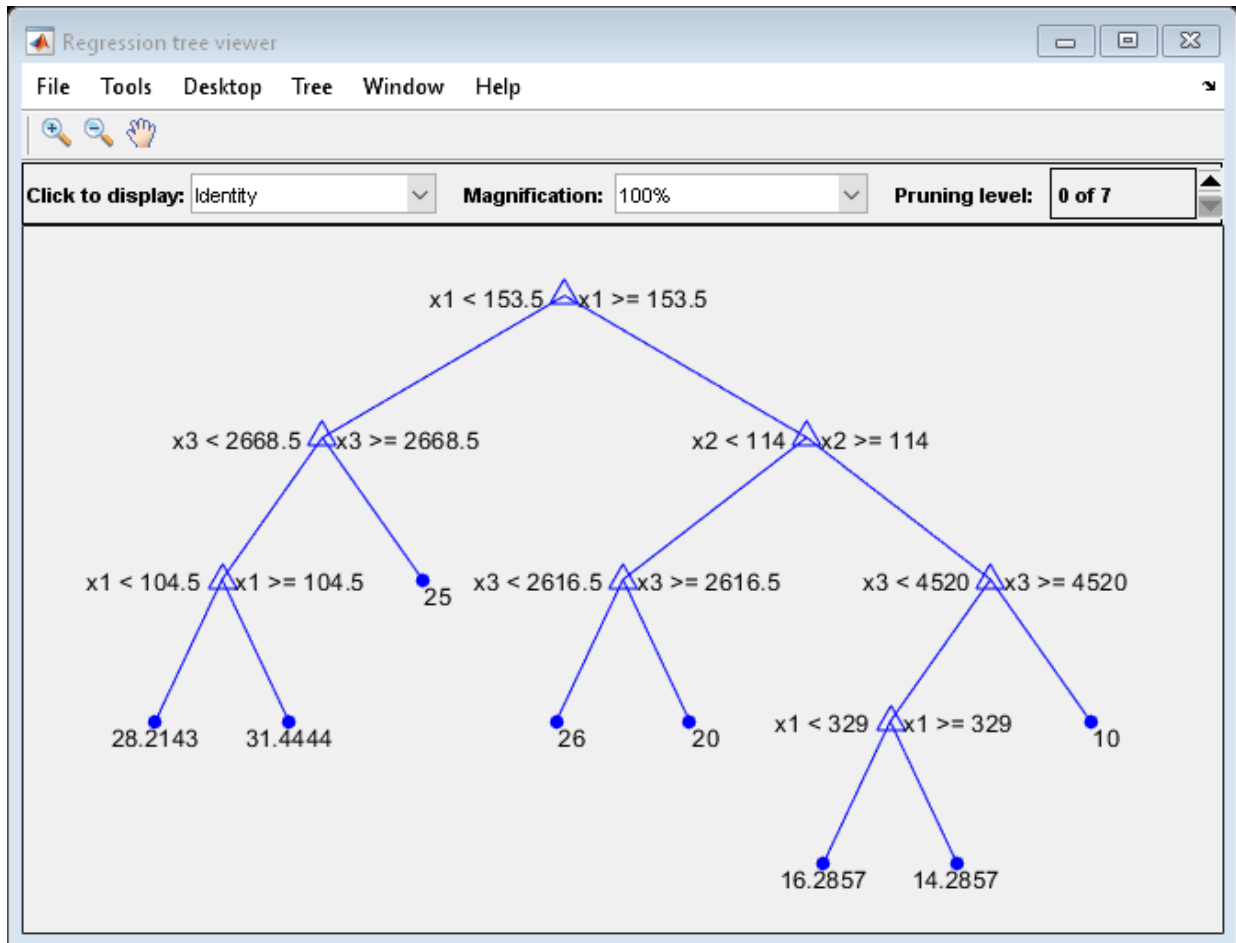
```
n = size(X,1);  
rng(1) % For reproducibility  
idxTrn = false(n,1);  
idxTrn(randsample(n,round(0.5*n))) = true; % Training set logical indices  
idxVal = idxTrn == false; % Validation set logical indices
```

Grow a regression tree using the training set.

```
Mdl = fitrtree(X(idxTrn,:),Y(idxTrn));
```

View the regression tree.

```
view(Mdl, 'Mode', 'graph');
```



The regression tree has seven pruning levels. Level 0 is the full, unpruned tree (as displayed). Level 7 is just the root node (i.e., no splits).

Examine the training sample MSE for each subtree (or pruning level) excluding the highest level.

```
m = max(Mdl.PruneList) - 1;
trnLoss = resubLoss(Mdl, 'SubTrees', 0:m)
```

```
trnLoss =
```

```
5.9789
6.2768
6.8316
7.5209
8.3951
10.7452
14.8445
```

- The MSE for the full, unpruned tree is about 6 units.
- The MSE for the tree pruned to level 1 is about 6.3 units.
- The MSE for the tree pruned to level 6 (i.e., a stump) is about 14.8 units.

Examine the validation sample MSE at each level excluding the highest level.

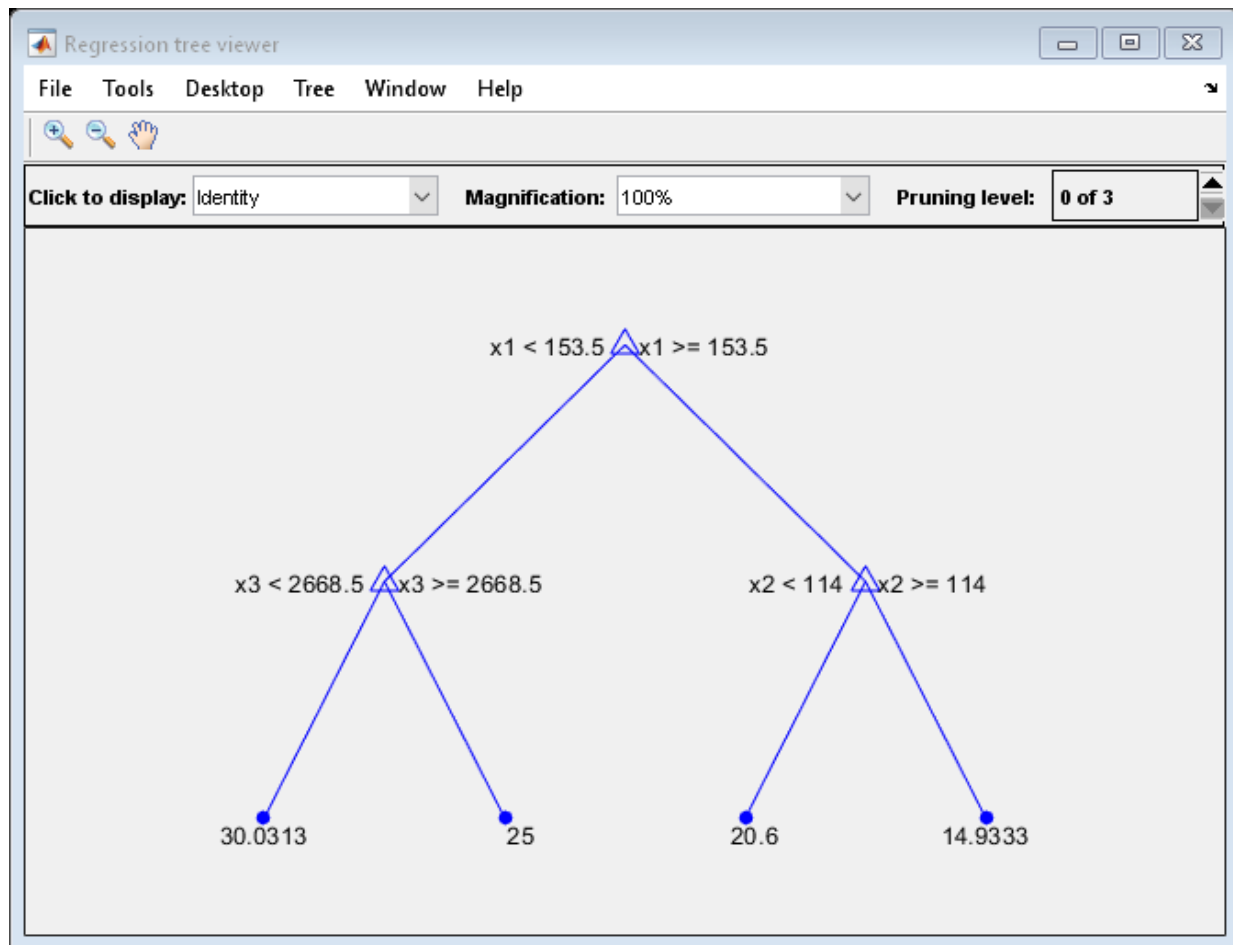
```
valLoss = loss(Mdl,X(idxVal,:),Y(idxVal),'SubTrees',0:m)
```

```
valLoss =
32.1205
31.5035
32.0541
30.8183
26.3535
30.0137
38.4695
```

- The MSE for the full, unpruned tree (level 0) is about 32.1 units.
- The MSE for the tree pruned to level 4 is about 26.4 units.
- The MSE for the tree pruned to level 5 is about 30.0 units.
- The MSE for the tree pruned to level 6 (i.e., a stump) is about 38.5 units.

To balance model complexity and out-of-sample performance, consider pruning `Mdl` to level 4.

```
pruneMdl = prune(Mdl,'Level',4);
view(pruneMdl,'Mode','graph')
```



## Definitions

### Loss Functions

The built-in loss function is 'mse', meaning mean squared error.

To write your own loss function, create a function file of the form



```
function loss = lossfun(Y,Yfit,W)
```

- `N` is the number of rows of `tree.X`.
- `Y` is an `N`-element vector representing the observed response.
- `Yfit` is an `N`-element vector representing the predicted responses.
- `W` is an `N`-element vector representing the observation weights.
- The output `loss` should be a scalar.

Pass the function handle `@lossfun` as the value of the `LossFun` name-value pair.

## See Also

`fitrtree` | `loss` | `resubPredict`

## resubMargin

**Class:** ClassificationDiscriminant

Classification margins by resubstitution

### Syntax

```
M = resubMargin(obj)
```

### Description

`M = resubMargin(obj)` returns resubstitution classification margins for `obj`.

### Input Arguments

`obj`

Discriminant analysis classifier, produced using `fitcdiscr`.

### Output Arguments

**M**

Numeric column-vector of length `size(obj.X,1)` containing the classification margins.

### Examples

#### Estimate Resubstitution Margins for Discriminant Analysis Classifiers

Find the margins for a discriminant analysis classifier for Fisher's iris data by resubstitution. Examine several entries.

Load Fisher's iris data set.

```
load fisheriris
```

Train a discriminant analysis classifier.

```
Mdl = fitcdiscr(meas,species);
```

Compute the resubstitution margins, and display several of them.

```
m = resubMargin(Mdl);  
m(1:25:end)
```

```
ans =
```

```
1.0000  
1.0000  
0.9998  
0.9998  
1.0000  
0.9946
```

## Definitions

### Margin

The classification margin is the difference between the classification score for the true class and maximal classification score for the false classes.

The classification margin is a column vector with the same number of rows as in the matrix  $X$ . A high value of margin indicates a more reliable prediction than a low value.

### Score

For discriminant analysis, the score of a classification is the posterior probability of the classification. For the definition of posterior probability in discriminant analysis, see “Posterior Probability” on page 20-8.

## See Also

`ClassificationDiscriminant` | `fitcdiscr` | `margin`

## Topics

“Discriminant Analysis Classification” on page 20-2

# resubMargin

**Class:** ClassificationECOC

Classification margins by resubstitution for multiclass, error-correcting output codes model

## Syntax

```
m = margin(Mdl)
m = margin(Mdl,Name,Value)
```

## Description

`m = margin(Mdl)` returns the classification margins on page 32-5806 (m) for the trained, multiclass, error-correcting output codes (ECOC) model `Mdl` using the training data stored in `Mdl.X` and corresponding class labels stored in `Mdl.Y`.

`m = margin(Mdl,Name,Value)` returns the classification margins with additional options specified by one or more `Name,Value` pair arguments.

For example, specify a decoding scheme, binary learner loss function, or verbosity level.

## Input Arguments

**mdl** — Multiclass ECOC model

ClassificationECOC model

Multiclass ECOC model, specified as a ClassificationECOC model returned by `fitcecoc`.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single

quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

### BinaryLoss — Binary learner loss function

'hamming' | 'linear' | 'logit' | 'exponential' | 'binodeviance' | 'hinge'  
| 'quadratic' | function handle

Binary learner loss function, specified as the comma-separated pair consisting of 'BinaryLoss' and a built-in, loss-function name or function handle.

- This table contains names and descriptions of the built-in functions, where  $y_j$  is a class label for a particular binary learner (in the set  $\{-1, 1, 0\}$ ),  $s_j$  is the score for observation  $j$ , and  $g(y_j, s_j)$  is the binary loss formula.

Value	Description	Score Domain	$g(y_j, s_j)$
'binodeviance'	Binomial deviance	$(-\infty, \infty)$	$\log[1 + \exp(-2y_j s_j)] / [2\log(2)]$
'exponential'	Exponential	$(-\infty, \infty)$	$\exp(-y_j s_j) / 2$
'hamming'	Hamming	$[0, 1]$ or $(-\infty, \infty)$	$[1 - \text{sign}(y_j s_j)] / 2$
'hinge'	Hinge	$(-\infty, \infty)$	$\max(0, 1 - y_j s_j) / 2$
'linear'	Linear	$(-\infty, \infty)$	$(1 - y_j s_j) / 2$
'logit'	Logistic	$(-\infty, \infty)$	$\log[1 + \exp(-y_j s_j)] / [2\log(2)]$
'quadratic'	Quadratic	$[0, 1]$	$[1 - y_j(2s_j - 1)]^2 / 2$

The software normalizes the binary losses such that the loss is 0.5 when  $y_j = 0$ . Also, the software calculates the mean binary loss for each class.

- For a custom binary loss function, e.g., `customFunction`, specify its function handle `'BinaryLoss', @customFunction`.

`customFunction` should have this form:

```
bLoss = customFunction(M, s)
```

where:

- $M$  is the  $K$ -by- $L$  coding matrix stored in `Mdl.CodingMatrix`.
- $s$  is the 1-by- $L$  row vector of classification scores.

- `bLoss` is the classification loss. This scalar aggregates the binary losses for every learner in a particular class. For example, you can use the mean binary loss to aggregate the loss over the learners for each class.
- $K$  is the number of classes.
- $L$  is the number of binary learners.

For an example of passing a custom binary loss function, see “Predict Test-Sample Labels of ECOC Models Using Custom Binary Loss Function” on page 32-4933.

By default, if all binary learners are:

- SVMs or linear classification models of SVM learners, then `BinaryLoss` is `'hinge'`
- Ensembles trained by `AdaboostM1` or `GentleBoost`, then `BinaryLoss` is `'exponential'`
- Ensembles trained by `LogitBoost`, then `BinaryLoss` is `'binodeviance'`
- Linear classification models of logistic regression learners, or you specify to predict class posterior probabilities (i.e., set `'FitPosterior', 1` in `fitcecoc`), then `BinaryLoss` is `'quadratic'`

Otherwise, the default `BinaryLoss` is `'hamming'`. To check the default value, use dot notation to display the `BinaryLoss` property of the trained model at the command line.

Example: `'BinaryLoss', 'binodeviance'`

Data Types: `char` | `function_handle`

### Decoding — Decoding scheme

`'lossweighted'` (default) | `'lossbased'`

Decoding scheme that aggregates the binary losses on page 32-4941, specified as the comma-separated pair consisting of `'Decoding'` and `'lossweighted'` or `'lossbased'`.

Example: `'Decoding', 'lossbased'`

Data Types: `char`

### Options — Estimation options

`[]` (default) | structure array returned by `statset`

Estimation options, specified as the comma-separated pair consisting of `'Options'` and a structure array returned by `statset`.

To invoke parallel computing:

- You need a Parallel Computing Toolbox license.
- Specify 'Options', `statset('UseParallel',1)`.

### **verbose** — Verbosity level

0 (default) | 1

Verbosity level, specified as the comma-separated pair consisting of 'Verbose' and 0 or 1. Verbose controls the amount of diagnostic messages that the software displays in the Command Window.

If `Verbose` is 0, then the software does not display diagnostic messages. Otherwise, the software displays diagnostic messages.

Example: `'Verbose',1`

Data Types: `single` | `double`

## Output Arguments

### **m** — Classification margins

numeric column vector

Classification margins, returned as a numeric column vector.

`m` has the same length as `Mdl.Y`. The software estimates each entry of `m` using the trained ECOC model `Mdl`, the corresponding row of `Mdl.X`, and the true class label `Mdl.Y`.

## Examples

### Estimate In-Sample Classification Margins of ECOC Models

Load Fisher's iris data set.

```
load fisheriris
X = meas;
Y = species;
```



Train an ECOC model using SVM binary classifiers. It is good practice to standardize the predictors and define the class order. Specify to standardize the predictors using an SVM template.

```
t = templateSVM('Standardize',1);
classOrder = unique(Y)
Mdl = fitcecoc(X,Y,'Learners',t,'ClassNames',classOrder);
```

```
classOrder =

3x1 cell array

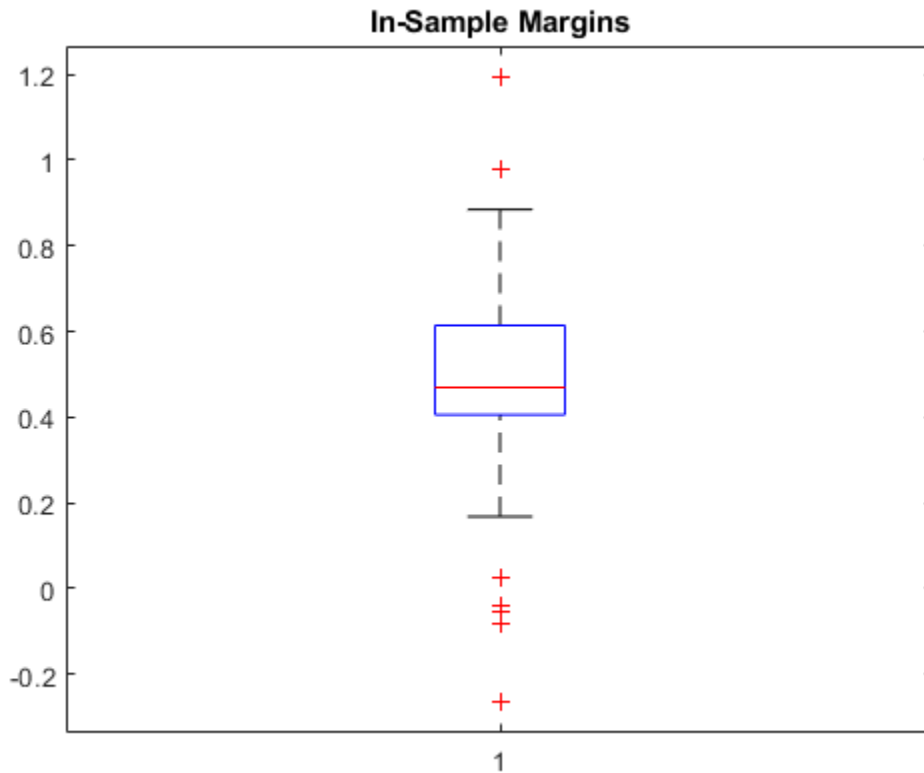
    {'setosa'    }
    {'versicolor'}
    {'virginica' }
```

`t` is an SVM template object. The software uses default values for empty options in `t` during training. `Mdl` is a ClassificationECOC model.

Estimate the in-sample classification margins. Display the distribution of the margins using a boxplot.

```
m = resubMargin(Mdl);

figure;
boxplot(m);
title 'In-Sample Margins'
```



An observation margin is the positive-class, negated loss minus the maximum negative-class, negated loss. Classifiers that yield relatively large margins are desirable.

### Select ECOC Model Features by Examining In-Sample Margins

The classifier margins measure, for each observation, the difference between the positive-class, negated loss score and the maximal negative-class, negated loss. One way to perform feature selection is to compare in-sample margins from multiple models. Based solely on this criterion, the model with the highest margins is the best model.

Load Fisher's iris data set. Define two data sets:

- fullX contains all 4 predictors.
- partX contains the sepal measurements.

```
load fisheriris
X = meas;
fullX = X;
partX = X(:,1:2);
Y = species;
```

Train ECOC models using SVM binary learners for each predictor set. It is good practice to standardize the predictors and define the class order. Specify to standardize the predictors using an SVM template, and to compute posterior probabilities.

```
t = templateSVM('Standardize',1);
classOrder = unique(Y)
FullMdl = fitcecoc(fullX,Y,'Learners',t,'ClassNames',classOrder,...
    'FitPosterior',1);
PartMdl = fitcecoc(partX,Y,'Learners',t,'ClassNames',classOrder,...
    'FitPosterior',1);
```

```
classOrder =

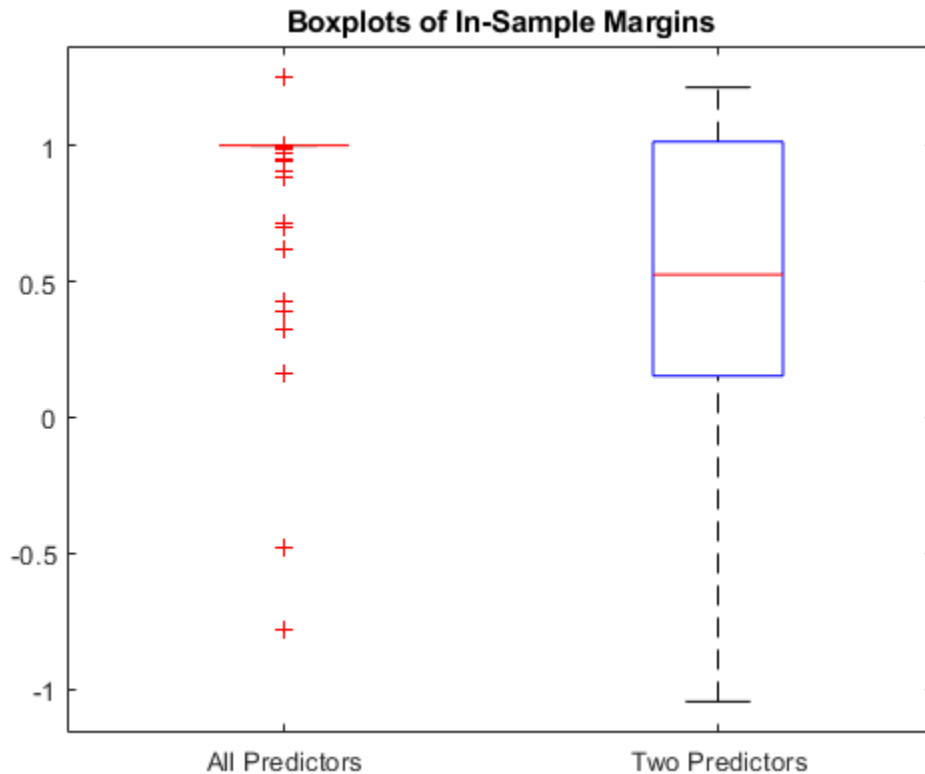
3x1 cell array

    {'setosa'   }
    {'versicolor'}
    {'virginica' }
```

Estimate the in-sample margins for each classifier. For each model, display the distribution of the margins using a boxplot.

```
fullMargins = resubMargin(FullMdl);
partMargins = resubMargin(PartMdl);

figure;
boxplot([fullMargins partMargins],'Labels',{'All Predictors','Two Predictors'});
title('Boxplots of In-Sample Margins')
```



The margin distribution of CMdl is situated higher, and with less variability than the margin distribution of PCMdl.

- “Quick Start Parallel Computing for Statistics and Machine Learning Toolbox” on page 30-2

## Definitions

### Classification Margin

The classification margins are, for each observation, the difference between the negative loss for the true class and maximal negative loss among the false classes. If the margins

are on the same scale, then they serve as a classification confidence measure, i.e., among multiple classifiers, those that yield larger margins are better.

## Binary Loss

A binary loss is a function of the class and classification score that determines how well a binary learner classifies an observation into the class.

Let:

- $m_{kj}$  be element  $(k,j)$  of the coding design matrix  $M$  (i.e., the code corresponding to class  $k$  of binary learner  $j$ )
- $s_j$  be the score of binary learner  $j$  for an observation
- $g$  be the binary loss function
- $\hat{k}$  be the predicted class for the observation

In loss-based decoding [15], the class producing the minimum sum of the binary losses over binary learners determines the predicted class of an observation, that is,

$$\hat{k} = \operatorname{argmin}_k \sum_{j=1}^L |m_{kj}| g(m_{kj}, s_j).$$

In loss-weighted decoding [15], the class producing the minimum average of the binary losses over binary learners determines the predicted class of an observation, that is,

$$\hat{k} = \operatorname{argmin}_k \frac{\sum_{j=1}^L |m_{kj}| g(m_{kj}, s_j)}{\sum_{j=1}^L |m_{kj}|}.$$

Allwein et al. [2] suggest that loss-weighted decoding improves classification accuracy by keeping loss values for all classes in the same dynamic range.

This table summarizes the supported loss functions, where  $y_j$  is a class label for a particular binary learner (in the set  $\{-1, 1, 0\}$ ),  $s_j$  is the score for observation  $j$ , and  $g(y_j, s_j)$ .

Value	Description	Score Domain	$g(y, s_j)$
'binodeviance'	Binomial deviance	$(-\infty, \infty)$	$\log[1 + \exp(-2y_j s_j)] / [2\log(2)]$
'exponential'	Exponential	$(-\infty, \infty)$	$\exp(-y_j s_j) / 2$
'hamming'	Hamming	$[0, 1]$ or $(-\infty, \infty)$	$[1 - \text{sign}(y_j s_j)] / 2$
'hinge'	Hinge	$(-\infty, \infty)$	$\max(0, 1 - y_j s_j) / 2$
'linear'	Linear	$(-\infty, \infty)$	$(1 - y_j s_j) / 2$
'logit'	Logistic	$(-\infty, \infty)$	$\log[1 + \exp(-y_j s_j)] / [2\log(2)]$
'quadratic'	Quadratic	$[0, 1]$	$[1 - y_j(2s_j - 1)]^2 / 2$

The software normalizes the binary losses such that the loss is 0.5 when  $y_j = 0$ , and aggregates using the average of the binary learners [2].

Do not confuse the binary loss with the overall classification loss (specified by the `LossFun` name-value pair argument of `predict` and `loss`), e.g., classification error, which measures how well an ECOC classifier performs as a whole.

## Tip

To compare margins or edges of several classifiers, use template objects to specify a common score transform function among the classifiers when you train them using `fitcecoc`.

## References

- [1] Allwein, E., R. Schapire, and Y. Singer. “Reducing multiclass to binary: A unifying approach for margin classifiers.” *Journal of Machine Learning Research*. Vol. 1, 2000, pp. 113–141.
- [2] Escalera, S., O. Pujol, and P. Radeva. “On the decoding process in ternary error-correcting output codes.” *IEEE Transactions on Pattern Analysis and Machine Intelligence*. Vol. 32, Issue 7, 2010, pp. 120–134.

- [3] Escalera, S., O. Pujol, and P. Radeva. “Separability of ternary codes for sparse designs of error-correcting output codes.” *Pattern Recogn.* Vol. 30, Issue 3, 2009, pp. 285–297.

## See Also

`ClassificationECOC` | `fitcecoc` | `margin` | `predict` | `resubEdge` | `resubPredict`

## Topics

“Quick Start Parallel Computing for Statistics and Machine Learning Toolbox” on page 30-2

“Reproducibility in Parallel Statistical Computations” on page 30-14

“Concepts of Parallel Computing in Statistics and Machine Learning Toolbox” on page 30-8

## resubMargin

**Class:** ClassificationEnsemble

Classification margins by resubstitution

### Syntax

```
margin = resubMargin(ens)
margin = resubMargin(ens, Name, Value)
```

### Description

`margin = resubMargin(ens)` returns the classification margin obtained by `ens` on its training data.

`margin = resubMargin(ens, Name, Value)` calculates margins with additional options specified by one or more `Name, Value` pair arguments.

### Input Arguments

#### **ens**

A classification ensemble created with `fitcensemble`.

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.



**learners**

Indices of weak learners in the ensemble ranging from 1 to NumTrained. resubMargin uses only these learners for calculating margin.

**Default:** 1:NumTrained

## Output Arguments

**margin**

A numeric column-vector of length `size(ens.X,1)` containing the classification margins.

## Examples

### Compute Resubstitution Margins for Classification Ensemble

Find the resubstitution margins for an ensemble that classifies the Fisher iris data.

Load the Fisher iris data set.

```
load fisheriris
```

Train an ensemble of 100 boosted classification trees using AdaBoostM2.

```
t = templateTree('MaxNumSplits',1); % Weak learner template tree object
ens = fitcensemble(meas,species,'Method','AdaBoostM2','Learners',t);
```

Find the resubstitution margins.

```
margin = resubMargin(ens);
[min(margin) mean(margin) max(margin)]

ans =

    -0.5674     3.2486     4.6245
```

## Definitions

### Margin

The classification margin is the difference between the classification score for the true class and maximal classification score for the false classes. Margin is a column vector with the same number of rows as in the matrix `ens.X`.

### Score (ensemble)

For ensembles, a classification score represents the confidence of a classification into a class. The higher the score, the higher the confidence.

Different ensemble algorithms have different definitions for their scores. Furthermore, the range of scores depends on ensemble type. For example:

- AdaBoostM1 scores range from  $-\infty$  to  $\infty$ .
- Bag scores range from 0 to 1.

### See Also

`resubEdge` | `resubLoss` | `resubMargin` | `resubPredict`

# resubMargin

**Class:** ClassificationKNN

Margin of  $k$ -nearest neighbor classifier by resubstitution

## Syntax

```
m = resubMargin mdl
```

## Description

`m = resubMargin(mdl)` returns the classification margins of the data used to train `mdl`. For the definition, see “Margin” on page 32-5815.

## Input Arguments

**mdl** — Classifier model

classifier model object

$k$ -nearest neighbor classifier model, returned as a classifier model object.

Note that using the 'CrossVal', 'KFold', 'Holdout', 'Leaveout', or 'CVPartition' options results in a model of class `ClassificationPartitionedModel`. You cannot use a partitioned tree for prediction, so this kind of tree does not have a `predict` method.

Otherwise, `mdl` is of class `ClassificationKNN`, and you can use the `predict` method to make predictions.

## Output Arguments

**m**

A numeric column vector of length `size mdl.X, 1`, where `mdl.X` is the training data for `mdl`. Each entry in `m` represents the margin for the corresponding row of `mdl.X` and (true class) `mdl.Y`.

## Examples

### Resubstitution Margin Calculation

Construct a  $k$ -nearest neighbor classifier for the Fisher iris data, where  $k = 5$ .

Load the data.

```
load fisheriris
X = meas;
Y = species;
```

Construct a classifier for 5-nearest neighbors.

```
mdl = fitcknn(X,Y,'NumNeighbors',5);
```

Examine some statistics of the resubstitution margin of the classifier.

```
m = resubMargin(mdl);
[max(m) min(m) mean(m)]

ans =

    1.0000    -0.6000    0.9253
```

The mean margin is over 0.9, indicating fairly high classification accuracy for resubstitution. For more reliable assessment of model accuracy, consider cross validation, such as `kfoldLoss`.

- “Examine Quality of KNN Classifier” on page 18-37

## Definitions

### Margin

The classification margin is the difference between the classification score for the true class and maximal classification score for the false classes.

Margin is a column vector with the same number of rows as in the training data.

### Score

The score of a classification is the posterior probability of the classification. The posterior probability is the number of neighbors that have that classification, divided by the number of neighbors. For a more detailed definition that includes weights and prior probabilities, see “Posterior Probability” on page 32-4897.

## See Also

`ClassificationKNN` | `fitcknn` | `resubEdge` | `resubLoss` | `resubPredict`

### Topics

“Examine Quality of KNN Classifier” on page 18-37

“Classification Using Nearest Neighbors” on page 18-16

## resubMargin

**Class:** ClassificationNaiveBayes

Classification margins for naive Bayes classifiers by resubstitution

### Syntax

```
m = resubMargin(Mdl)
```

### Description

`m = resubMargin(Mdl)` returns the resubstitution classification margins on page 32-5820 (`m`) for the naive Bayes classifier `Mdl` using the training data stored in `Mdl.X` and corresponding class labels stored in `Mdl.Y`.

### Input Arguments

**Mdl** — Fully trained naive Bayes classifier

ClassificationNaiveBayes model

A fully trained naive Bayes classifier, specified as a `ClassificationNaiveBayes` model trained by `fitcnb`.

### Output Arguments

**m** — Classification margins

numeric vector

Classification margins on page 32-5820, returned as a numeric vector.

`m` has the same length equal to `size(Mdl.X,1)`. Each entry of `m` is the classification margin of the corresponding observation (row) of `Mdl.X` and element of `Mdl.Y`.

## Examples

### Estimate In-Sample Classification Margins of Naive Bayes Classifiers

Load Fisher's iris data set.

```
load fisheriris
X = meas;    % Predictors
Y = species; % Response
```

Train a naive Bayes classifier. It is good practice to specify the class order. Assume that each predictor is conditionally, normally distributed given its label.

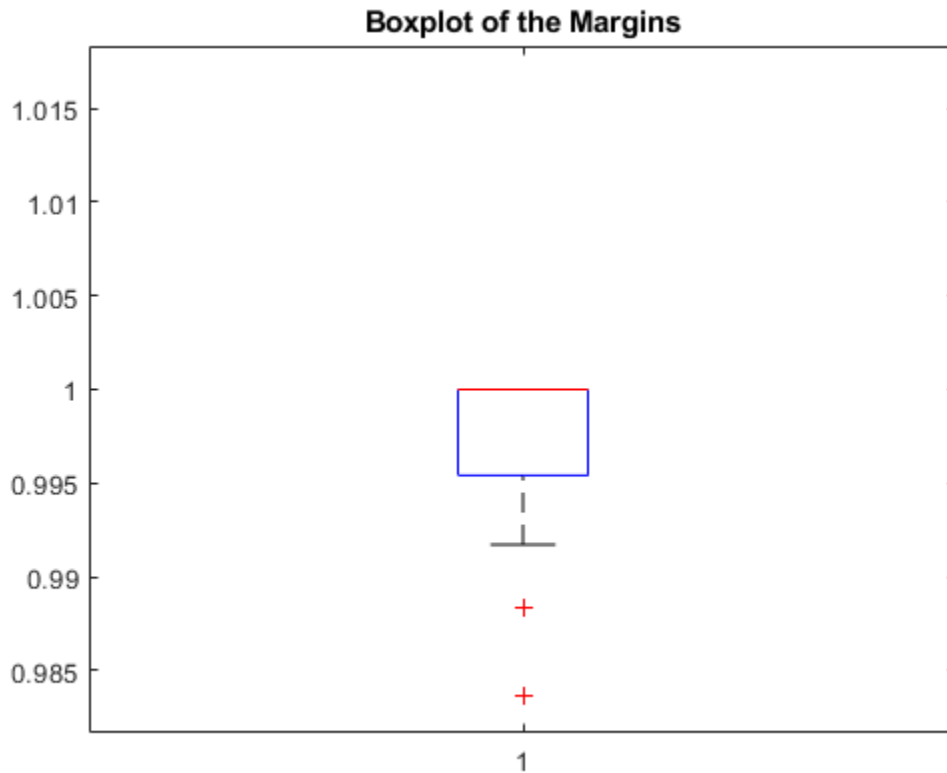
```
Mdl = fitcnb(X,Y,'ClassNames',{'setosa','versicolor','virginica'});
```

Mdl is a ClassificationNaiveBayes classifier.

Estimate the in-sample classification margins. Display the distribution of the margins using a boxplot.

```
m = resubMargin(Mdl);

figure;
boxplot(m);
h = gca;
iqr = quantile(m,0.75) - quantile(m,0.25);
h.YLim = median(m) + iqr*[-4 4];
title 'Boxplot of the Margins';
```



An observation margin is the observed (true) class score minus the maximum false class score among all scores in the respective class. Classifiers that yield relatively large margins are desirable.

### Select Naive Bayes Classifier Features by Examining In-Sample Margins

The classifier margins measure, for each observation, the difference between the true class observed score and the maximal false class score for a particular class. One way to perform feature selection is to compare in-sample margins from multiple models. Based solely on this criterion, the model with the highest margins is the best model.

Load Fisher's iris data set. Define two data sets:



- fullX contains all predictors (except the removed column of 0s).
- partX contains the last 20 predictors.

```
load fisheriris
X = meas;      % Predictors
Y = species;  % Response
fullX = X;
partX = X(:,3:4);
```

Train naive Bayes classifiers for each predictor set.

```
FullMdl = fitcnb(fullX,Y);
PartMdl  = fitcnb(partX,Y);
```

Estimate the in-sample margins for each classifier. Compute confidence intervals for each sample.

```
fullM = resubMargin(FullMdl);
partM = resubMargin(PartMdl);
n = size(X,1);
fullMCI = mean(fullM) + 2*[-std(fullM)/n std(fullM)/n]
partMCI = mean(partM) + 2*[-std(partM)/n std(partM)/n]
```

```
fullMCI =
    0.8898    0.8991
```

```
partMCI =
    0.9129    0.9209
```

The confidence intervals are tight, and mutually exclusive. The margin confidence interval of the classifier trained using just predictors 3 and 4 has higher values than that

of the full model. Therefore, the model trained on two predictors has better in-sample performance.

## Definitions

### Classification Edge

The classification edge is the weighted mean of the classification margins.

If you supply weights, then the software normalizes them to sum to the prior probability of their respective class. The software uses the normalized weights to compute the weighted mean.

One way to choose among multiple classifiers, e.g., to perform feature selection, is to choose the classifier that yields the highest edge.

### Classification Margin

The classification margins are, for each observation, the difference between the score for the true class and maximal score for the false classes. Provided that they are on the same scale, margins serve as a classification confidence measure, i.e., among multiple classifiers, those that yield larger margins are better.

### Posterior Probability

The posterior probability is the probability that an observation belongs in a particular class, given the data.

For naive Bayes, the posterior probability that a classification is  $k$  for a given observation  $(x_1, \dots, x_P)$  is

$$\hat{P}(Y = k | x_1, \dots, x_P) = \frac{P(X_1, \dots, X_P | y = k)\pi(Y = k)}{P(X_1, \dots, X_P)},$$

where:

- $P(X_1, \dots, X_P | y = k)$  is the conditional joint density of the predictors given they are in class  $k$ . `Mdl.DistributionNames` stores the distribution names of the predictors.

- $\pi(Y = k)$  is the class prior probability distribution. `Mdl.Prior` stores the prior distribution.
- $P(X_1, \dots, X_P)$  is the joint density of the predictors. The classes are discrete, so

$$P(X_1, \dots, X_P) = \sum_{k=1}^K P(X_1, \dots, X_P | y = k) \pi(Y = k).$$

## Prior Probability

The prior probability is the believed relative frequency that observations from a class occur in the population for each class.

## Score

The naive Bayes score is the class posterior probability given the observation.

## See Also

`ClassificationSVM` | `CompactClassificationSVM` | `fitsvm` | `margin` | `resubEdge` | `resubLoss`

## resubMargin

**Class:** ClassificationSVM

Classification margins for support vector machine classifiers by resubstitution

### Syntax

```
m = resubMargin(SVMModel)
```

### Description

`m = resubMargin(SVMModel)` returns the resubstitution classification margins on page 32-3931 (`m`) for the support vector machine (SVM) classifier `SVMModel` using the training data stored in `SVMModel.X` and corresponding class labels stored in `SVMModel.Y`.

### Input Arguments

**SVMModel** — Full, trained SVM classifier

ClassificationSVM classifier

Full, trained SVM classifier, specified as a `ClassificationSVM` model trained using `fitcsvm`.

### Output Arguments

**m** — Classification margins

numeric vector

Classification margins, returned as a numeric vector.

`m` has the same length as `Y`. The software estimates each entry of `m` using the trained SVM classifier `SVMModel`, the corresponding row of `X`, and the true class label `Y`.

## Examples

### Estimate In-Sample Classification Margins of SVM Classifiers

Load the `ionosphere` data set.

```
load ionosphere
```

Train an SVM classifier. It is good practice to specify the class order and standardize the data.

```
SVModel = fitcsvm(X,Y,'ClassNames',{'b','g'},'Standardize',true);
```

`SVModel` is a `ClassificationSVM` classifier. The negative class is 'b' and the positive class is 'g'.

Estimate the in-sample classification margins.

```
m = resubMargin(SVModel);  
m(10:20)
```

```
ans =
```

```
5.5622  
4.2918  
1.9993  
4.5520  
-1.4897  
3.2816  
4.0260  
4.5419  
16.4449  
2.0006  
23.3782
```

An observation margin is the observed (true) class score minus the maximum false class score among all scores in the respective class. Classifiers that yield relatively large margins are desirable.

### Select SVM Classifier Features by Examining In-Sample Margins

The classifier margins measure, for each observation, the difference between the true class observed score and the maximal false class score for a particular class. One way to perform feature selection is to compare in-sample margins from multiple models. Based solely on this criterion, the model with the highest margins is the best model.

Load the `ionosphere` data set. Define two data sets:

- `fullX` contains all predictors (except the removed column of 0s).
- `partX` contains the last 20 predictors.

```
load ionosphere
fullX = X;
partX = X(:,end-20:end);
```

Train SVM classifiers for each predictor set.

```
FullSVMModel = fitcsvm(fullX,Y);
PartSVMModel = fitcsvm(partX,Y);
```

Estimate the in-sample margins for each classifier.

```
fullMargins = resubMargin(FullSVMModel);
partMargins = resubMargin(PartSVMModel);
n = size(X,1);
p = sum(fullMargins < partMargins)/n
```

```
p =
```

```
0.2194
```

Approximately 22% of the margins from the full model are less than those from the model with fewer predictors. This suggests that the model trained using all of the predictors is better.

## Definitions

### Classification Edge

The edge is the weighted mean of the classification margins.

The weights are the prior class probabilities. If you supply weights, then the software normalizes them to sum to the prior probabilities in the respective classes. The software uses the renormalized weights to compute the weighted mean.

One way to choose among multiple classifiers, e.g., to perform feature selection, is to choose the classifier that yields the highest edge.

### Classification Margin

The classification margins for binary classification are, for each observation, the difference between the classification score for the true class and the classification score for the false class.

The software defines the classification margin for binary classification as

$$m = 2yf(x).$$

$x$  is an observation. If the true label of  $x$  is the positive class, then  $y$  is 1, and  $-1$  otherwise.  $f(x)$  is the positive-class classification score for the observation  $x$ . The literature commonly defines the margin as  $m = yf(x)$ .

If the margins are on the same scale, then they serve as a classification confidence measure, i.e., among multiple classifiers, those that yield larger margins are better.

### Classification Score

The SVM classification score for classifying observation  $x$  is the signed distance from  $x$  to the decision boundary ranging from  $-\infty$  to  $+\infty$ . A positive score for a class indicates that  $x$  is predicted to be in that class. A negative score indicates otherwise.

The score for predicting  $x$  into the positive class, also the numerical, predicted response for  $x$ ,  $f(x)$ , is the trained SVM classification function

$$f(x) = \sum_{j=1}^n \alpha_j y_j G(x_j, x) + b,$$

where  $(\alpha_1, \dots, \alpha_n, b)$  are the estimated SVM parameters,  $G(x_j, x)$  is the dot product in the predictor space between  $x$  and the support vectors, and the sum includes the training set observations. The score for predicting  $x$  into the negative class is  $-f(x)$ .

If  $G(x_j, x) = x_j'x$  (the linear kernel), then the score function reduces to  $f(x) = (x/s)' \beta + b$ .

$s$  is the kernel scale and  $\beta$  is the vector of fitted linear coefficients.

## Algorithms

For binary classification, the software defines the margin for observation  $j$ ,  $m_j$ , as

$$m_j = 2y_j f(x_j),$$

where  $y_j \in \{-1, 1\}$ , and  $f(x_j)$  is the predicted score of observation  $j$  for the positive class. However, the literature commonly uses  $m_j = y_j f(x_j)$  to define the margin.

## References

- [1] Christianini, N., and J. C. Shawe-Taylor. *An Introduction to Support Vector Machines and Other Kernel-Based Learning Methods*. Cambridge, UK: Cambridge University Press, 2000.

## See Also

ClassificationSVM | CompactClassificationSVM | fitcsvm | margin | resubEdge | resubLoss



# resubMargin

**Class:** ClassificationTree

Classification margins by resubstitution

## Syntax

```
M = resubMargin(tree)
```

## Description

`M = resubMargin(tree)` returns resubstitution classification margins for `tree`.

## Input Arguments

**tree**

A classification tree created by `fitctree`.

## Output Arguments

**M**

A numeric column-vector of length `size(tree.X,1)` containing the classification margins.

## Examples

Find the margins for a classification tree for the Fisher iris data by resubstitution.

Examine several entries:

```
load fisheriris  
tree = fitctree(meas,species);
```

```
M = resubMargin(tree);  
M(1:25:end)  
  
ans =  
    1.0000  
    1.0000  
    1.0000  
    1.0000  
    0.9565  
    0.9565
```

## Definitions

### Margin

Classification margin is the difference between classification score for the true class and maximal classification score for the false classes. A high value of margin indicates a more reliable prediction than a low value.

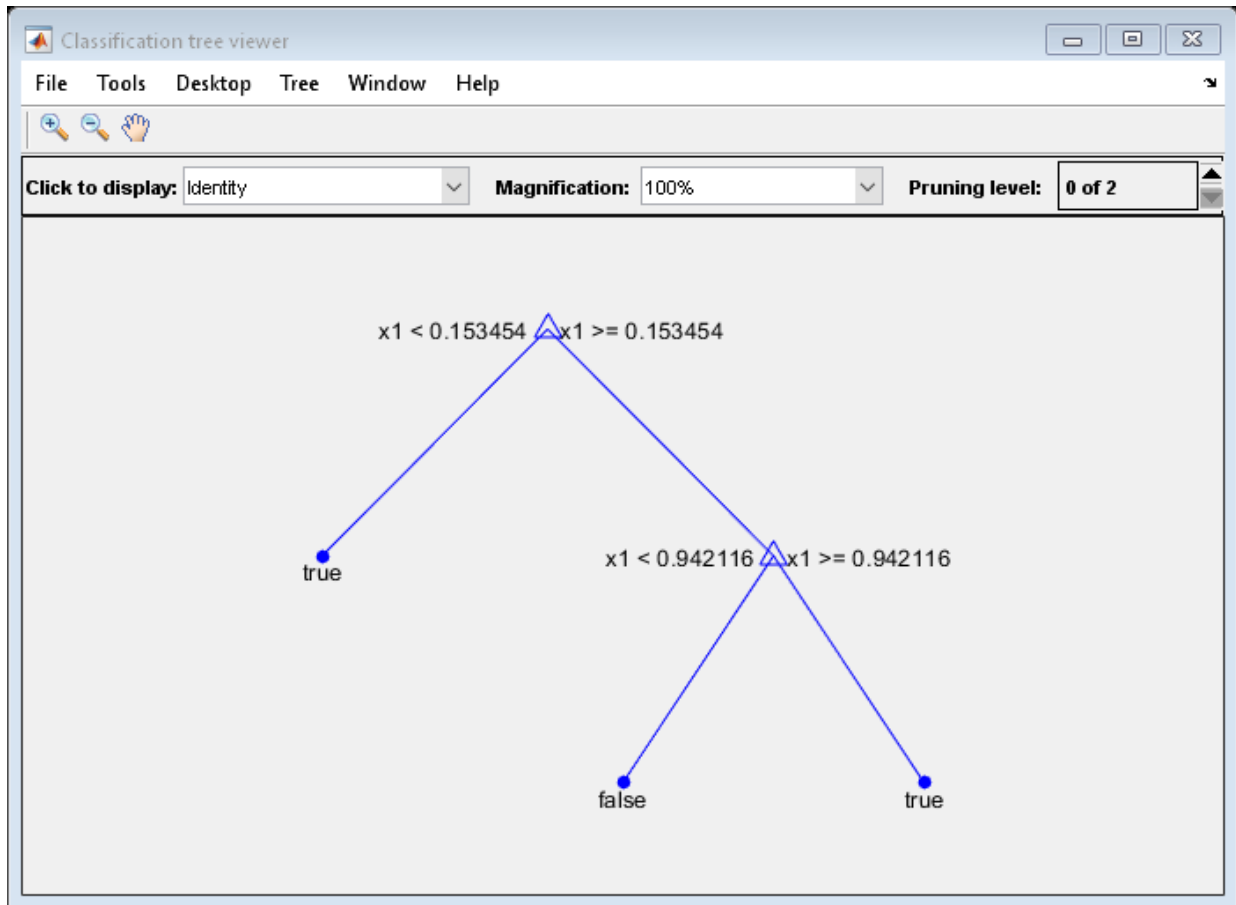
### Score (tree)

For trees, the score of a classification of a leaf node is the posterior probability of the classification at that node. The posterior probability of the classification at a node is the number of training sequences that lead to that node with the classification, divided by the number of training sequences that lead to that node.

For example, consider classifying a predictor  $X$  as `true` when  $X < 0.15$  or  $X > 0.95$ , and  $X$  is `false` otherwise.

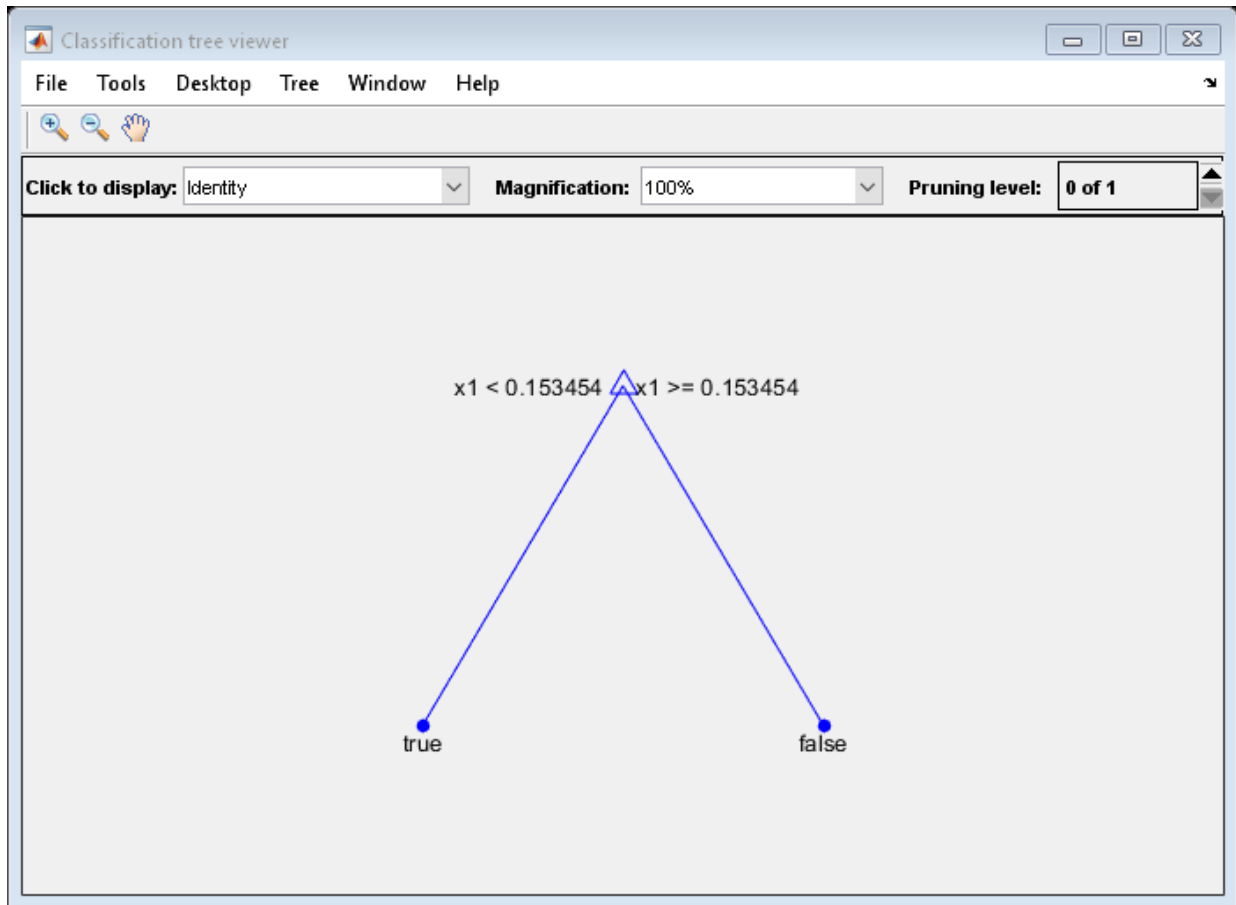
Generate 100 random points and classify them:

```
rng(0, 'twister') % for reproducibility  
X = rand(100,1);  
Y = (abs(X - .55) > .4);  
tree = fitctree(X,Y);  
view(tree, 'Mode', 'Graph')
```



Prune the tree:

```
tree1 = prune(tree, 'Level', 1);  
view(tree1, 'Mode', 'Graph')
```



The pruned tree correctly classifies observations that are less than 0.15 as `true`. It also correctly classifies observations from .15 to .94 as `false`. However, it incorrectly classifies observations that are greater than .94 as `false`. Therefore, the score for observations that are greater than .15 should be about  $.05/.85=.06$  for `true`, and about  $.8/.85=.94$  for `false`.

Compute the prediction scores for the first 10 rows of `X`:

```
[~,score] = predict(tree1,X(1:10));
[score X(1:10,:)]
```

```
ans =
    0.9059    0.0941    0.8147
    0.9059    0.0941    0.9058
         0     1.0000    0.1270
    0.9059    0.0941    0.9134
    0.9059    0.0941    0.6324
         0     1.0000    0.0975
    0.9059    0.0941    0.2785
    0.9059    0.0941    0.5469
    0.9059    0.0941    0.9575
    0.9059    0.0941    0.9649
```

Indeed, every value of  $x$  (the right-most column) that is less than 0.15 has associated scores (the left and center columns) of 0 and 1, while the other values of  $x$  have associated scores of 0.91 and 0.09. The difference (score 0.09 instead of the expected .06) is due to a statistical fluctuation: there are 8 observations in  $x$  in the range  $(.95, 1)$  instead of the expected 5 observations.

## See Also

[fitctree](#) | [margin](#) | [resubEdge](#) | [resubLoss](#) | [resubPredict](#)

## resubPredict

**Class:** ClassificationDiscriminant

Predict resubstitution response of classifier

### Syntax

```
label = resubPredict(obj)
[label,posterior] = resubPredict(obj)
[label,posterior,cost] = resubPredict(obj)
```

### Description

`label = resubPredict(obj)` returns the labels `obj` predicts for the data `obj.X`. `label` is the predictions of `obj` on the data that `fitcdiscr` used to create `obj`.

`[label,posterior] = resubPredict(obj)` returns the posterior class probabilities for the predictions.

`[label,posterior,cost] = resubPredict(obj)` returns the predicted misclassification costs per class for the resubstituted data.

### Input Arguments

**obj**

Discriminant analysis classifier, produced using `fitcdiscr`.

## Output Arguments

### **label**

Response `obj` predicts for the training data. `label` is the same data type as the training response data `obj.Y`. The predicted class labels are those with minimal expected misclassification cost; see “Prediction Using Discriminant Analysis Models” on page 20-8.

### **posterior**

N-by-K matrix of posterior probabilities for classes `obj` predicts, where N is the number of observations and K is the number of classes.

### **cost**

N-by-K matrix of predicted misclassification costs. Each cost is the average misclassification cost with respect to the posterior probability.

## Examples

Find the total number of misclassifications of the Fisher iris data for a discriminant analysis classifier:

```
load fisheriris
obj = fitcdiscr(meas,species);
Ypredict = resubPredict(obj); % the predictions
Ysame = strcmp(Ypredict,species); % true when ==
sum(~Ysame) % how many are different?

ans =
     3
```

## Definitions

### Posterior Probability

`posterior(i,k)` is the posterior probability of class `k` for observation `i`. For the mathematical definition, see “Posterior Probability” on page 20-8.

## See Also

`ClassificationDiscriminant` | `fitcdiscr` | `predict`

## Topics

“Discriminant Analysis Classification” on page 20-2



# resubPredict

**Class:** ClassificationECOC

Predict resubstitution responses for multiclass, error-correcting output codes model

## Syntax

```
label = resubPredict(Mdl)
label = resubPredict(Mdl, Name, Value)
[label, NegLoss, PBScore] = resubPredict(____)
[label, NegLoss, PBScore, Posterior] = resubPredict(____)
```

## Description

`label = resubPredict(Mdl)` returns a vector of predicted class labels for the predictor data (stored in `Mdl.X`) based on the trained, multiclass, error-correcting output codes model `Mdl`.

The software predicts the classification of an observation by assigning the observation to the class yielding the largest negated average binary loss (or, equivalently, the smallest average binary loss).

`label = resubPredict(Mdl, Name, Value)` returns predicted class labels with additional options specified by one or more `Name, Value` pair arguments.

For example, specify the posterior probability estimation method, decoding scheme, or verbosity level.

`[label, NegLoss, PBScore] = resubPredict(____)` additionally returns negated average binary loss on page 32-5850 per class (`NegLoss`) for observations, and positive-class scores (`PBScore`) for the observations classified by each binary learner.

`[label, NegLoss, PBScore, Posterior] = resubPredict(____)` additionally returns posterior class probability estimates for observations (`Posterior`).

To obtain posterior class probabilities, you must set `'FitPosterior', 1` when training the ECOC model using `fitcecoc`. Otherwise, `resubPredict` throws an error.

## Input Arguments

### **mdl** — Multiclass ECOC model

`ClassificationECOC` model

Multiclass ECOC model, specified as a `ClassificationECOC` model returned by `fitcecoc`.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### **BinaryLoss** — Binary learner loss function

'hamming' | 'linear' | 'logit' | 'exponential' | 'binodeviance' | 'hinge'  
| 'quadratic' | function handle

Binary learner loss function, specified as the comma-separated pair consisting of `'BinaryLoss'` and a built-in, loss-function name or function handle.

- This table contains names and descriptions of the built-in functions, where  $y_j$  is a class label for a particular binary learner (in the set  $\{-1, 1, 0\}$ ),  $s_j$  is the score for observation  $j$ , and  $g(y_j, s_j)$  is the binary loss formula.

Value	Description	Score Domain	$g(y_j, s_j)$
'binodeviance'	Binomial deviance	$(-\infty, \infty)$	$\log[1 + \exp(-2y_j s_j)] / [2\log(2)]$
'exponential'	Exponential	$(-\infty, \infty)$	$\exp(-y_j s_j) / 2$
'hamming'	Hamming	$[0, 1]$ or $(-\infty, \infty)$	$[1 - \text{sign}(y_j s_j)] / 2$
'hinge'	Hinge	$(-\infty, \infty)$	$\max(0, 1 - y_j s_j) / 2$
'linear'	Linear	$(-\infty, \infty)$	$(1 - y_j s_j) / 2$

Value	Description	Score Domain	$g(y, s)$
'logit'	Logistic	$(-\infty, \infty)$	$\frac{\log[1 + \exp(-y_j s_j)]}{[2\log(2)]}$
'quadratic'	Quadratic	[0,1]	$[1 - y_j(2s_j - 1)]^2/2$

The software normalizes the binary losses such that the loss is 0.5 when  $y_j = 0$ . Also, the software calculates the mean binary loss for each class.

- For a custom binary loss function, e.g., `customFunction`, specify its function handle `'BinaryLoss', @customFunction`.

`customFunction` should have this form:

```
bLoss = customFunction(M, s)
```

where:

- $M$  is the  $K$ -by- $L$  coding matrix stored in `Mdl.CodingMatrix`.
- $s$  is the 1-by- $L$  row vector of classification scores.
- `bLoss` is the classification loss. This scalar aggregates the binary losses for every learner in a particular class. For example, you can use the mean binary loss to aggregate the loss over the learners for each class.
- $K$  is the number of classes.
- $L$  is the number of binary learners.

For an example of passing a custom binary loss function, see “Predict Test-Sample Labels of ECOC Models Using Custom Binary Loss Function” on page 32-4933.

By default, if all binary learners are:

- SVMs or linear classification models of SVM learners, then `BinaryLoss` is `'hinge'`
- Ensembles trained by `AdaboostM1` or `GentleBoost`, then `BinaryLoss` is `'exponential'`
- Ensembles trained by `LogitBoost`, then `BinaryLoss` is `'binodeviance'`
- Linear classification models of logistic regression learners, or you specify to predict class posterior probabilities (i.e., set `'FitPosterior', 1` in `fitcecoc`), then `BinaryLoss` is `'quadratic'`

Otherwise, the default `BinaryLoss` is `'hamming'`. To check the default value, use dot notation to display the `BinaryLoss` property of the trained model at the command line.

Example: `'BinaryLoss', 'binodeviance'`

Data Types: `char` | `function_handle`

### **Decoding — Decoding scheme**

`'lossweighted'` (default) | `'lossbased'`

Decoding scheme that aggregates the binary losses on page 32-4941, specified as the comma-separated pair consisting of `'Decoding'` and `'lossweighted'` or `'lossbased'`.

Example: `'Decoding', 'lossbased'`

Data Types: `char`

### **NumKLInitializations — Number of random initial values**

0 (default) | nonnegative integer

Number of random initial values for fitting posterior probabilities by Kullback-Leibler divergence minimization, specified as the comma-separated pair consisting of `'NumKLInitializations'` and a nonnegative integer.

If you do not request the fourth output argument (`Posterior`) and set `'PosteriorMethod', 'kl'` (the default), then the software ignores the value of `NumKLInitializations`.

For more details, see “Posterior Estimation Using Kullback-Leibler Divergence” on page 32-4943.

Example: `'NumKLInitializations', 5`

Data Types: `single` | `double`

### **Options — Estimation options**

`[]` (default) | structure array returned by `statset`

Estimation options, specified as the comma-separated pair consisting of `'Options'` and a structure array returned by `statset`.

To invoke parallel computing:

- You need a Parallel Computing Toolbox license.

- Specify 'Options', `statset('UseParallel',1)`.

### **PosteriorMethod — Posterior probability estimation method**

'kl' (default) | 'qp'

Posterior probability estimation method, specified as the comma-separated pair consisting of 'PosteriorMethod' and 'kl' or 'qp'.

- If PosteriorMethod is 'kl', then the software estimates multiclass posterior probabilities by minimizing the Kullback-Leibler divergence between the predicted and expected posterior probabilities returned by binary learners. For details, see “Posterior Estimation Using Kullback-Leibler Divergence” on page 32-4943.
- If PosteriorMethod is 'qp', then the software estimates multiclass posterior probabilities by solving a least-squares problem using quadratic programming. You need an Optimization Toolbox license to use this option. For details, see “Posterior Estimation Using Quadratic Programming” on page 32-4945.
- If you do not request the fourth output argument (Posterior), then the software ignores the value of PosteriorMethod.

Example: 'PosteriorMethod','qp'

Data Types: char

### **Verbose — Verbosity level**

0 (default) | 1

Verbosity level, specified as the comma-separated pair consisting of 'Verbose' and 0 or 1. Verbose controls the amount of diagnostic messages that the software displays in the Command Window.

If Verbose is 0, then the software does not display diagnostic messages. Otherwise, the software displays diagnostic messages.

Example: 'Verbose',1

Data Types: single | double

## Output Arguments

### **label** — Predicted class labels

categorical array | character array | logical vector | vector of numeric values | cell array of character vectors

Predicted class labels, returned as a categorical or character array, logical or numeric vector, or cell array of character vectors.

`label` is the same data type as the `Mdl.ClassNames`, and has length equal to the number of rows of `Mdl.X`.

The software predicts the classification of an observation by assigning the observation to the class yielding the largest negated average binary loss (or, equivalently, the smallest average binary loss).

### **NegLoss** — Negated average binary losses

numeric matrix

Negated, average binary losses on page 32-5850, returned as a numeric matrix. `NegLoss` is an  $n$ -by- $K$  matrix, where  $n$  is the number of observations (`size(Mdl.X,1)`) and  $K$  is the number of unique classes (`size(Mdl.ClassNames,1)`).

### **PBScore** — Positive-class scores

numeric matrix

Positive-class scores for each binary learner, returned as a numeric matrix. `PBScore` is an  $n$ -by- $L$  matrix, where  $n$  is the number of observations (`size(Mdl.X,1)`) and  $L$  is the number of binary learners (`size(Mdl.CodingMatrix,2)`).

### **Posterior** — Posterior class probabilities

numeric matrix

Posterior class probabilities, returned as a numeric matrix. `Posterior` is an  $n$ -by- $K$  matrix, where  $n$  is the number of observations (`size(Mdl.X,1)`) and  $K$  is the number of unique classes (`size(Mdl.ClassNames,1)`).

You must set `'FitPosterior',1` when training the ECOC model using `fitcecoc` to request `Posterior`. Otherwise, the software throws an error.

## Examples

### Predict Labels of Training Data Using ECOC Models

Load Fisher's iris data set.

```
load fisheriris
X = meas;
Y = categorical(species);
n = numel(Y); % Sample size
classOrder = unique(Y);
```

Train an ECOC model using SVM binary classifiers. It is good practice to standardize the predictors and define the class order. Specify to standardize the predictors using an SVM template.

```
t = templateSVM('Standardize',1);
Mdl = fitcecoc(X,Y,'Learners',t,'ClassNames',classOrder);
```

`t` is an SVM template object. The software uses default values for empty options in `t` during training. `Mdl` is a `ClassificationECOC` model.

Predict the labels of the training data. Print a random subset of true and predicted labels.

```
labels = resubPredict(Mdl);
rng(1);
idx = randsample(n,10);
table(Y(idx),labels(idx),'VariableNames',{'TrueLabels','PredictedLabels'})
```

ans =

10x2 table

TrueLabels	PredictedLabels
setosa	setosa
versicolor	versicolor
virginica	virginica
setosa	setosa
versicolor	versicolor

```
setosa      setosa
versicolor versicolor
versicolor versicolor
setosa      setosa
setosa      setosa
```

Mdl correctly labeled the observations with indices idx.

### Predict Resubstitution Labels of ECOC Models Using a Custom Binary Loss Function

Load Fisher's iris data set.

```
load fisheriris
X = meas;
Y = categorical(species);
n = numel(Y);           % Sample size
classOrder = unique(Y); % Class order
K = numel(classOrder); % Number of classes
```

Train an ECOC model using SVM binary classifiers. It is good practice to standardize the predictors and define the class order. Specify to standardize the predictors using an SVM template.

```
t = templateSVM('Standardize',1);
Mdl = fitcecoc(X,Y,'Learners',t,'ClassNames',classOrder);
```

t is an SVM template object. The software uses default values for empty options in t during training. Mdl is a ClassificationECOC model.

SVM scores are signed distances from the observation to the decision boundary.

Therefore, the domain is  $(-\infty, \infty)$ . Create a custom binary loss function that:

- Maps the coding design matrix ( $M$ ) and positive-class classification scores ( $s$ ) for each learner to the binary loss for each observation
- Uses linear loss
- Aggregates the binary learner loss using the median

You can create a separate function for the binary loss function, and then save it on the MATLAB® path. Or, you can specify an anonymous binary loss function.



```
customBL = @(M,s)nanmedian(1 - bsxfun(@times,M,s),2)/2;
```

Predict resubstitution labels and estimate the median binary loss per class. Print the median negative binary losses per class for a random set of 10 observations.

```
[label,NegLoss] = resubPredict(Mdl,'BinaryLoss',customBL);
```

```
rng(1); % For reproducibility
```

```
idx = randsample(n,10);
```

```
classOrder
```

```
table(Y(idx),label(idx),NegLoss(idx,:),'VariableNames',...
      {'TrueLabel','PredictedLabel','NegLoss'})
```

```
classOrder =
```

```
3x1 categorical array
```

```
setosa
versicolor
virginica
```

```
ans =
```

```
10x3 table
```

TrueLabel	PredictedLabel	NegLoss		
setosa	versicolor	0.12375	1.9569	-3.5807
versicolor	versicolor	-1.017	0.62908	-1.112
virginica	virginica	-1.9083	-0.21792	0.62621
setosa	versicolor	0.43855	2.2443	-4.1828
versicolor	versicolor	-1.0733	0.39617	-0.82283
setosa	versicolor	0.26677	2.2004	-3.9672
versicolor	versicolor	-1.1235	0.6988	-1.0753
versicolor	versicolor	-1.271	0.51807	-0.7471
setosa	versicolor	0.35193	2.0678	-3.9198
setosa	versicolor	0.23363	2.1887	-3.9223

The column order corresponds to the elements of `classOrder`. The software predicts the label based on the maximum negated loss. The results seem to indicate that the median of the linear losses might not perform as well as other losses.

### Estimate Posterior Probabilities Using ECOC Classifiers

Load Fisher's iris data set. Train the classifier using the petal dimensions as predictors.

```
load fisheriris
X = meas(:,3:4);
Y = species;
rng(1); % For reproducibility
```

Create an SVM template, and specify the Gaussian kernel. It is good practice to standardize the predictors.

```
t = templateSVM('Standardize',1,'KernelFunction','gaussian');
```

`t` is an SVM template. Most of its properties are empty. When the software trains the ECOC classifier, it sets the applicable properties to their default values.

Train the ECOC classifier using the SVM template. Transform classification scores to class posterior probabilities (which are returned by `predict` or `resubPredict`) using the 'FitPosterior' name-value pair argument. Display diagnostic messages during the training using the 'Verbose' name-value pair argument. It is good practice to specify the class order.

```
Mdl = fitcecoc(X,Y,'Learners',t,'FitPosterior',1,...
    'ClassNames',{'setosa','versicolor','virginica'},...
    'Verbose',2);
```

```
Training binary learner 1 (SVM) out of 3 with 50 negative and 50 positive observations.
Negative class indices: 2
Positive class indices: 1
```

```
Fitting posterior probabilities for learner 1 (SVM).
Training binary learner 2 (SVM) out of 3 with 50 negative and 50 positive observations.
Negative class indices: 3
Positive class indices: 1
```

```
Fitting posterior probabilities for learner 2 (SVM).
Training binary learner 3 (SVM) out of 3 with 50 negative and 50 positive observations.
```

```
Negative class indices: 3
Positive class indices: 2
```

```
Fitting posterior probabilities for learner 3 (SVM).
```

Mdl is a ClassificationECOC model. The same SVM template applies to each binary learner, but you can adjust options for each binary learner by passing in a cell vector of templates.

Predict the in-sample labels and class posterior probabilities. Display diagnostic messages during the computation of labels and class posterior probabilities using the 'Verbose' name-value pair argument.

```
[label,~,~,Posterior] = resubPredict(Mdl,'Verbose',1);
Mdl.BinaryLoss
```

```
Predictions from all learners have been computed.
Loss for all observations has been computed.
Computing posterior probabilities...
```

```
ans =

    'quadratic'
```

The software assigns an observation to the class that yields the smallest average binary loss. Since all binary learners are computing posterior probabilities, the binary loss function is quadratic.

Display a random set of results.

```
idx = randsample(size(X,1),10,1);
Mdl.ClassNames
table(Y(idx),label(idx),Posterior(idx,:),...
    'VariableNames',{'TrueLabel','PredLabel','Posterior'})
```

```
ans =

    3x1 cell array

    {'setosa'    }
    {'versicolor'}
    {'virginica' }
```

```
ans =
10x3 table

   TrueLabel   PredLabel   Posterior
   _____   _____   _____
   'virginica'   'virginica'   0.0039321   0.0039869   0.99208
   'virginica'   'virginica'   0.017067   0.018263   0.96467
   'virginica'   'virginica'   0.014948   0.015856   0.9692
   'versicolor' 'versicolor'  2.2197e-14  0.87317   0.12683
   'setosa'       'setosa'      0.999     0.00025091  0.00074639
   'versicolor' 'virginica'   2.2195e-14  0.059429   0.94057
   'versicolor' 'versicolor'  2.2194e-14  0.97001   0.029986
   'setosa'       'setosa'      0.999     0.0002499   0.00074741
   'versicolor' 'versicolor'  0.0085646  0.98259   0.008849
   'setosa'       'setosa'      0.999     0.00025013  0.00074718
```

The columns of Posterior correspond to the class order of `Mdl.ClassNames`.

Define a grid of values in the observed predictor space. Predict the posterior probabilities for each instance in the grid.

```
xMax = max(X);
xMin = min(X);

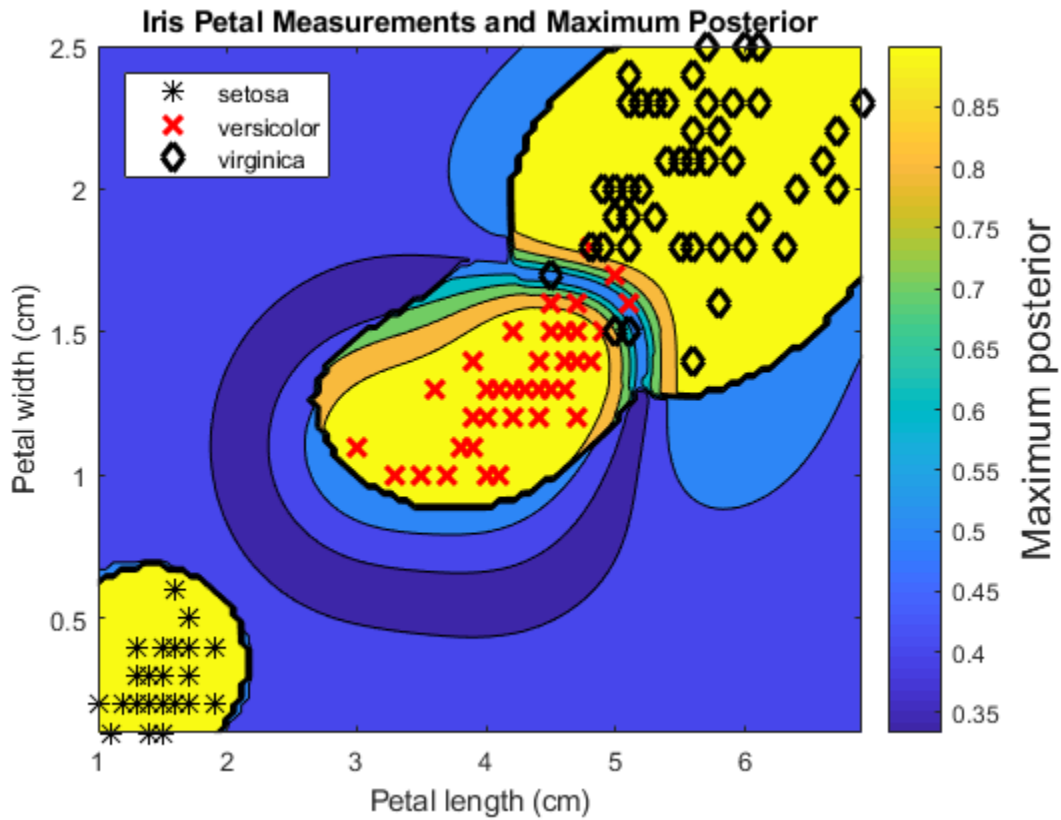
x1Pts = linspace(xMin(1), xMax(1));
x2Pts = linspace(xMin(2), xMax(2));
[x1Grid, x2Grid] = meshgrid(x1Pts, x2Pts);

[~,~,~,PosteriorRegion] = predict(Mdl, [x1Grid(:), x2Grid(:)]);
```

For each coordinate on the grid, plot the maximum class posterior probability among all classes.

```
figure;
contourf(x1Grid, x2Grid, ...
         reshape(max(PosteriorRegion, [], 2), size(x1Grid, 1), size(x1Grid, 2)));
h = colorbar;
h.YLabel.String = 'Maximum posterior';
h.YLabel.FontSize = 15;
hold on
```

```
gh = gscatter(X(:,1),X(:,2),Y,'krk','*xd',8);  
gh(2).LineWidth = 2;  
gh(3).LineWidth = 2;  
  
title 'Iris Petal Measurements and Maximum Posterior';  
xlabel 'Petal length (cm)';  
ylabel 'Petal width (cm)';  
axis tight  
legend(gh,'Location','NorthWest')  
hold off
```



### Estimate Posterior Probabilities Using Parallel Computing

Train an error-correcting output codes, multiclass model and estimate posterior probabilities using parallel computing.

Load the `arrhythmia` data set.

```
load arrhythmia
Y = categorical(Y);
tabulate(Y)
n = numel(Y);
K = numel(unique(Y));
```

Value	Count	Percent
1	245	54.20%
2	44	9.73%
3	15	3.32%
4	15	3.32%
5	13	2.88%
6	25	5.53%
7	3	0.66%
8	2	0.44%
9	9	1.99%
10	50	11.06%
14	4	0.88%
15	5	1.11%
16	22	4.87%

Several classes are not represented in the data, and many other classes have low relative frequencies.

Specify an ensemble learning template that uses the GentleBoost method and 50 weak, classification tree learners.

```
t = templateEnsemble('GentleBoost',50,'Tree');
```

`t` is a template object. Most of the options are empty (`[]`). The software uses default values for all empty options during training.

Since there are many classes, specify a sparse random coding design.

```
rng(1); % For reproducibility
Coding = designecoc(K,'sparserandom');
```

Train an ECOC model using parallel computing. Specify to fit posterior probabilities.

```

pool = parpool; % Invokes workers
options = statset('UseParallel',1);
Mdl = fitcecoc(X,Y,'Learner',t,'Options',options,'Coding',Coding,...
'FitPosterior',1);

```

Starting parallel pool (parpool) using the 'local' profile ... connected to 4 workers.

Mdl is a ClassificationECOC model. You can access its properties using dot notation. The pool invokes four workers. The number of workers might vary among systems.

Estimate posterior probabilities, and display the posterior probability of being classified as not having arrhythmia (class 1) given the data.

```

[~,~,~,posterior] = resubPredict(Mdl);
idx = randsample(n,10,1);
table(idx,Y(idx),posterior(idx,1),...
'VariableNames',{'ObservationIndex','TrueLabel','PosteriorNoArrythmia'})

```

ans =

ObservationIndex	TrueLabel	PosteriorNoArrythmia
79	1	0.91522
248	1	0.95376
398	10	0.032369
207	1	0.97965
340	1	0.93628
206	1	0.97795
345	10	0.015643
296	2	0.14796
391	1	0.96494
406	1	0.94867

- “Quick Start Parallel Computing for Statistics and Machine Learning Toolbox” on page 30-2

## Definitions

### Binary Loss

A binary loss is a function of the class and classification score that determines how well a binary learner classifies an observation into the class.

Let:

- $m_{kj}$  be element  $(k,j)$  of the coding design matrix  $M$  (i.e., the code corresponding to class  $k$  of binary learner  $j$ )
- $s_j$  be the score of binary learner  $j$  for an observation
- $g$  be the binary loss function
- $\hat{k}$  be the predicted class for the observation

In loss-based decoding [15], the class producing the minimum sum of the binary losses over binary learners determines the predicted class of an observation, that is,

$$\hat{k} = \operatorname{argmin}_k \sum_{j=1}^L |m_{kj}| g(m_{kj}, s_j).$$

In loss-weighted decoding [15], the class producing the minimum average of the binary losses over binary learners determines the predicted class of an observation, that is,

$$\hat{k} = \operatorname{argmin}_k \frac{\sum_{j=1}^L |m_{kj}| g(m_{kj}, s_j)}{\sum_{j=1}^L |m_{kj}|}.$$

Allwein et al. [2] suggest that loss-weighted decoding improves classification accuracy by keeping loss values for all classes in the same dynamic range.

This table summarizes the supported loss functions, where  $y_j$  is a class label for a particular binary learner (in the set  $\{-1,1,0\}$ ),  $s_j$  is the score for observation  $j$ , and  $g(y_j, s_j)$ .

Value	Description	Score Domain	$g(y_j, s_j)$
'binodeviance'	Binomial deviance	$(-\infty, \infty)$	$\log[1 + \exp(-2y_j s_j)] / [2\log(2)]$



Value	Description	Score Domain	$g(y_j, s_j)$
'exponential'	Exponential	$(-\infty, \infty)$	$\exp(-y_j s_j)/2$
'hamming'	Hamming	$[0, 1]$ or $(-\infty, \infty)$	$[1 - \text{sign}(y_j s_j)]/2$
'hinge'	Hinge	$(-\infty, \infty)$	$\max(0, 1 - y_j s_j)/2$
'linear'	Linear	$(-\infty, \infty)$	$(1 - y_j s_j)/2$
'logit'	Logistic	$(-\infty, \infty)$	$\log[1 + \exp(-y_j s_j)]/[2\log(2)]$
'quadratic'	Quadratic	$[0, 1]$	$[1 - y_j(2s_j - 1)]^2/2$

The software normalizes the binary losses such that the loss is 0.5 when  $y_j = 0$ , and aggregates using the average of the binary learners [2].

Do not confuse the binary loss with the overall classification loss (specified by the `LossFun` name-value pair argument of `predict` and `loss`), e.g., classification error, which measures how well an ECOC classifier performs as a whole.

## Algorithms

- The software can estimate class posterior probabilities using quadratic programming or by minimizing the Kullback-Leibler divergence. For the following descriptions of the posterior estimation algorithms, let:
  - $m_{kj}$  be the element  $(k,j)$  of the coding design matrix  $M$ .
  - $I$  be the indicator function.
  - $\hat{p}_k$  be the class posterior probability estimate for class  $k$  of an observation,  $k = 1, \dots, K$ .
  - $r_j$  be the positive-class posterior probability for binary learner  $j$ . That is,  $r_j$  is the probability that binary learner  $j$  classifies an observation into the positive class, given the training data.

## Posterior Estimation Using Kullback-Leibler Divergence

By default, the software minimizes the Kullback-Leibler divergence to estimate class posterior probabilities. The Kullback-Leibler divergence between the expected and observed positive-class posterior probabilities is

$$\Delta(r, r) = \sum_{j=1}^L w_j \left[ r_j \log \frac{r_j}{1-r_j} + (1-r_j) \log \frac{1-r_j}{r_j} \right],$$

where  $w_j = \sum_{S_j} w_i^*$  is the weight for binary learner  $j$  with  $S_j$  the set of observation indices that binary learner  $j$  is trained on and  $w_i^*$  is the weight of observation  $i$ . The software minimizes the divergence iteratively. The first step is to choose initial values

$\hat{p}_k^{(0)}$ ;  $k = 1, \dots, K$  for the class posterior probabilities.

- If you do not specify `NumKLIterations`, then the software uses both sets of deterministic initial values described next, and uses the one that minimizes  $\Delta$ .

- $\hat{p}_k^{(0)} = 1 / K$ ;  $k = 1, \dots, K$ .

- $\hat{p}_k^{(0)}$ ;  $k = 1, \dots, K$  is the solution of the system
 
$$M_{01} \hat{p}^{(0)} = r,$$

where  $M_{01}$  is  $M$  with all  $m_{kj} = -1$  replaced with 0, and  $r$  is a vector of positive-class posterior probabilities returned by the  $L$  binary learners [14]. The software uses `lsqnonneg` to solve the system.

- If you specify `'NumKLIterations', c`, where  $c$  is a natural number, then the

software does the following to choose  $\hat{p}_k^{(0)}$ ;  $k = 1, \dots, K$ , and uses the one that minimizes  $\Delta$ .

- The software chooses both sets of deterministic initial values as described previously.
- The software randomly generates  $c$  vectors of length  $K$  using `rand`, and then normalizes each vector to sum to 1.

At iteration  $t$ , the software:

- 1 Computes

$$\hat{r}_j^{(t)} = \frac{\sum_{k=1}^K \hat{p}_k^{(t)} I(m_{kj} = +1)}{\sum_{k=1}^K \hat{p}_k^{(t)} I(m_{kj} = +1 \cup m_{kj} = -1)}.$$

- 2 Estimates the next class posterior probability using

$$\hat{p}_k^{(t+1)} = \hat{p}_k^{(t)} \frac{\sum_{j=1}^L w_j [r_j I(m_{kj} = +1) + (1 - r_j) I(m_{kj} = -1)]}{\sum_{j=1}^L w_j [\hat{r}_j^{(t)} I(m_{kj} = +1) + (1 - \hat{r}_j^{(t)}) I(m_{kj} = -1)]}.$$

- 3 Normalizes  $\hat{p}_k^{(t+1)}$ ;  $k = 1, \dots, K$  so that they sum to 1.
- 4 Checks for convergence.

For more details, see [23] and [49].

## Posterior Estimation Using Quadratic Programming

Posterior probability estimation using quadratic programming requires an Optimization Toolbox license. To estimate posterior probabilities for an observation using this method, the software:

- 1 Estimates the positive-class posterior probabilities,  $r_j$ , for binary learners  $j = 1, \dots, L$ .

- 2 Using the relationship between  $r_j$  and  $\hat{p}_k$  [45], minimizes

$$\sum_{j=1}^L \left[ -r_j \sum_{k=1}^K \hat{p}_k I(m_{kj} = -1) + (1 - r_j) \sum_{k=1}^K \hat{p}_k I(m_{kj} = +1) \right]^2$$

with respect to  $\hat{p}_k$  and the restrictions

$$0 \leq \hat{p}_k \leq 1$$

$$\sum_k \hat{p}_k = 1.$$

The software performs minimization using `quadprog`.

## References

- [1] Allwein, E., R. Schapire, and Y. Singer. “Reducing multiclass to binary: A unifying approach for margin classifiers.” *Journal of Machine Learning Research*. Vol. 1, 2000, pp. 113–141.
- [2] Dietterich, T., and G. Bakiri. “Solving Multiclass Learning Problems Via Error-Correcting Output Codes.” *Journal of Artificial Intelligence Research*. Vol. 2, 1995, pp. 263–286.
- [3] Escalera, S., O. Pujol, and P. Radeva. “On the decoding process in ternary error-correcting output codes.” *IEEE Transactions on Pattern Analysis and Machine Intelligence*. Vol. 32, Issue 7, 2010, pp. 120–134.
- [4] Escalera, S., O. Pujol, and P. Radeva. “Separability of ternary codes for sparse designs of error-correcting output codes.” *Pattern Recogn.* Vol. 30, Issue 3, 2009, pp. 285–297.
- [5] Hastie, T., and R. Tibshirani. “Classification by Pairwise Coupling.” *Annals of Statistics*. Vol. 26, Issue 2, 1998, pp. 451–471.
- [6] Wu, T. F., C. J. Lin, and R. Weng. “Probability Estimates for Multi-Class Classification by Pairwise Coupling.” *Journal of Machine Learning Research*. Vol. 5, 2004, pp. 975–1005.
- [7] Zadrozny, B. “Reducing Multiclass to Binary by Coupling Probability Estimates.” *NIPS 2001: Proceedings of Advances in Neural Information Processing Systems 14*, 2001, pp. 1041–1048.

## See Also

`ClassificationECOC` | `fitcecoc` | `predict` | `quadprog` | `statset`

## Topics

“Quick Start Parallel Computing for Statistics and Machine Learning Toolbox” on page 30-2  
“Reproducibility in Parallel Statistical Computations” on page 30-14  
“Concepts of Parallel Computing in Statistics and Machine Learning Toolbox” on page 30-8

# resubPredict

**Class:** ClassificationEnsemble

Predict ensemble response by resubstitution

## Syntax

```
label = resubPredict(ens)
[label, score] = resubPredict(ens)
[label, score] = resubPredict(ens, Name, Value)
```

## Description

`label = resubPredict(ens)` returns the labels `ens` predicts for the data `ens.X`. `label` is the predictions of `ens` on the data that `fitcensemble` used to create `ens`.

`[label, score] = resubPredict(ens)` also returns scores for all classes.

`[label, score] = resubPredict(ens, Name, Value)` finds resubstitution predictions with additional options specified by one or more `Name, Value` pair arguments.

## Input Arguments

### **ens**

A classification ensemble created with `fitcensemble`.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

**learners**

Indices of weak learners in the ensemble ranging from 1 to NumTrained. `oobLoss` uses only these learners for calculating loss.

**Default:** `1:NumTrained`

## Output Arguments

**label**

The response `ens` predicts for the training data. `label` is the same data type as the training response data `ens.Y`, and has the same number of entries as the number of rows in `ens.X`.

**score**

An N-by-K matrix, where N is the number of rows in `ens.X`, and K is the number of classes in `ens`. High score value indicates that an observation likely comes from this class.

## Examples

**Find Number of Ensemble Misclassifications**

Find the total number of misclassifications of the `fisheriris` data for a classification ensemble.

Load the Fisher iris data set.

```
load fisheriris
```

Train an ensemble of 100 boosted classification trees using `AdaBoostM2`.

```
t = templateTree('MaxNumSplits',1); % Weak learner template tree object
ens = fitcensemble(meas,species,'Method','AdaBoostM2','Learners',t);
```

Find the total number of misclassifications.

```
Ypredict = resubPredict(ens); % The predictions
Ysame = strcmp(Ypredict,species); % True when Ypredict and species are equal
sum(~Ysame) % Number of different predictions

ans = 5
```

## Definitions

### Score (ensemble)

For ensembles, a classification score represents the confidence of a classification into a class. The higher the score, the higher the confidence.

Different ensemble algorithms have different definitions for their scores. Furthermore, the range of scores depends on ensemble type. For example:

- AdaBoostM1 scores range from  $-\infty$  to  $\infty$ .
- Bag scores range from 0 to 1.

### See Also

[resubEdge](#) | [resubLoss](#) | [resubMargin](#) | [resubPredict](#)

## resubPredict

**Class:** ClassificationKNN

Predict resubstitution response of  $k$ -nearest neighbor classifier

### Syntax

```
label = resubPredict mdl
[label, score] = resubPredict mdl
[label, score, cost] = resubPredict mdl
```

### Description

`label = resubPredict mdl` returns the labels `mdl` predicts for the data `mdl.X`. `label` is the predictions of `mdl` on the data that `fitcknn` used to create `mdl`.

`[label, score] = resubPredict mdl` returns the posterior class probabilities for the predictions.

`[label, score, cost] = resubPredict mdl` returns the misclassification costs.

### Input Arguments

**mdl** — Classifier model

classifier model object

$k$ -nearest neighbor classifier model, returned as a classifier model object.

Note that using the 'CrossVal', 'Kfold', 'Holdout', 'Leaveout', or 'CVPartition' options results in a model of class `ClassificationPartitionedModel`. You cannot use a partitioned tree for prediction, so this kind of tree does not have a `predict` method.

Otherwise, `mdl` is of class `ClassificationKNN`, and you can use the `predict` method to make predictions.



## Output Arguments

### **label**

Predicted class labels for the points in the training data  $X$ , a vector with length equal to the number of rows in the training data  $X$ . The label is the class with minimal expected cost (see “Expected Cost” on page 32-5861).

### **score**

Numeric matrix of size  $N$ -by- $K$ , where  $N$  is the number of observations (rows) in the training data  $X$ , and  $K$  is the number of classes (in `mdl.ClassNames`). `score(i,j)` is the posterior probability that row  $i$  of  $X$  is of class  $j$ . See “Posterior Probability” on page 32-5860.

### **cost**

Matrix of expected costs of size  $N$ -by- $K$ , where  $N$  is the number of observations (rows) in the training data  $X$ , and  $K$  is the number of classes (in `mdl.ClassNames`). `cost(i,j)` is the cost of classifying row  $i$  of  $X$  as class  $j$ . See “Expected Cost” on page 32-5861.

## Examples

### **Predict the Labels of the Training Data**

Examine the quality of a classifier by its resubstitution predictions.

Load the data.

```
load fisheriris
X = meas;
Y = species;
```

Construct a classifier for 5-nearest neighbors.

```
mdl = fitcknn(X,Y,'NumNeighbors',5);
```

Generate the resubstitution predictions.

```
label = resubPredict(mdl);
```

Calculate the number of differences between the predictions `label` and the original data `Y`.

```
mydiff = not(strcmp(Y,label)); % mydiff(i) = 1 means they differ
sum(mydiff) % Number of differences
```

```
ans =
```

```
5
```

A values of 1 in `mydiff` indicates that the observed label differs from the corresponding predicted label. There are 5 misclassifications.

## Definitions

### Posterior Probability

For a vector (single query point)  $X$  and model `mdl`, let

- $K$  be the number of nearest neighbors used in prediction, `mdl.NumNeighbors`
- $\text{nbrd}(\text{mdl}, X)$  be the  $K$  nearest neighbors to  $X$  in `mdl.X`
- $Y(\text{nbrd})$  be the classifications of the points in  $\text{nbrd}(\text{mdl}, X)$ , namely `mdl.Y(nbrd)`
- $W(\text{nbrd})$  be the weights of the points in  $\text{nbrd}(\text{mdl}, X)$
- `prior` be the priors of the classes in `mdl.Y`

If there is a vector of prior probabilities, then the observation weights  $W$  are normalized by class to sum to the priors. This might involve a calculation for the point  $X$ , because weights can depend on the distance from  $X$  to the points in `mdl.X`.

The posterior probability  $p(j|X)$  is

$$p(j|X) = \frac{\sum_{i \in \text{nbrd}} W(i) 1_{Y(X(i)=j)}}{\sum_{i \in \text{nbrd}} W(i)}.$$

Here  $1_{Y(X(i)=j)}$  means 1 when `mdl.Y(i) = j`, and 0 otherwise.

## Expected Cost

There are two costs associated with KNN classification: the true misclassification cost per class, and the expected misclassification cost per observation. The third output of `predict` is the expected misclassification cost per observation.

Suppose you have `Nobs` observations that you want to classify with a trained classifier `mdl`. Suppose you have `K` classes. You place the observations into a matrix `X` with one observation per row. The command

```
[label,score,cost] = predict(mdl,X)
```

returns, among other outputs, a `cost` matrix of size `Nobs-by-K`. Each row of the `cost` matrix contains the expected (average) cost of classifying the observation into each of the `K` classes. `cost(n,k)` is

$$\sum_{i=1}^K \hat{P}(i | X(n)) C(k | i),$$

where

- $K$  is the number of classes.
- $\hat{P}(i | X(n))$  is the posterior probability on page 32-4897 of class  $i$  for observation  $X(n)$ .
- $C(k | i)$  is the true misclassification cost on page 32-4898 of classifying an observation as  $k$  when its true class is  $i$ .

## True Misclassification Cost

There are two costs associated with KNN classification: the true misclassification cost per class, and the expected misclassification cost per observation.

You can set the true misclassification cost per class in the `Cost` name-value pair when you run `fitcknn`. `Cost(i,j)` is the cost of classifying an observation into class  $j$  if its true class is  $i$ . By default, `Cost(i,j)=1` if  $i \neq j$ , and `Cost(i,j)=0` if  $i=j$ . In other words, the cost is 0 for correct classification, and 1 for incorrect classification.

## Algorithms

If you specified to standardize the predictor data, that is, `mdl.Mu` and `mdl.Sigma` are not empty (`[]`), then `resubPredict` standardizes the predictor data before predicting labels.

## See Also

`ClassificationKNN` | `fitcknn` | `predict` | `resubEdge` | `resubLoss` | `resubMargin`

# resubPredict

**Class:** ClassificationNaiveBayes

Predict naive Bayes classifier resubstitution response

## Syntax

```
label = resubPredict(Mdl)
[label, Posterior, Cost] = predict(Mdl)
```

## Description

`label = resubPredict(Mdl)` returns a vector of predicted class labels (`label`) for the trained naive Bayes classifier `Mdl` using the predictor data `Mdl.X`.

`[label, Posterior, Cost] = predict(Mdl)` additionally returns posterior probabilities on page 32-5868 (`Posterior`) and predicted (expected) misclassification costs on page 32-5867 (`Cost`) corresponding to the observations (rows) in `Mdl.X`.

## Input Arguments

**mdl** — Fully trained naive Bayes classifier

ClassificationNaiveBayes model

A fully trained naive Bayes classifier, specified as a `ClassificationNaiveBayes` model trained by `fitcnb`.

## Output Arguments

**label** — Predicted class labels

categorical vector | character array | logical vector | numeric vector | cell array of character vectors

Predicted class labels, returned as a categorical vector, character array, logical or numeric vector, or cell array of character vectors.

label:

- Is the same data type as the observed class labels ( $Y$ ) that trained `Mdl`
- Has length equal to the number of rows of  $X$
- Is the class yielding the lowest expected misclassification cost (`Cost`)

### **Posterior** — Class posterior probabilities

numeric matrix

Class posterior probabilities on page 32-5868, returned as a numeric matrix. `Posterior` has rows equal to the number of rows of `Mdl.X` and columns equal to the number of distinct classes in the training data (`size(Mdl.ClassNames, 1)`).

`Posterior(j, k)` is the predicted posterior probability of class  $k$  (i.e., in class `Mdl.ClassNames(k)`) given the observation in row  $j$  of `Mdl.X`.

Data Types: `double`

### **Cost** — Expected misclassification costs

numeric matrix

Expected misclassification costs on page 32-5867, returned as a numeric matrix. `Cost` has rows equal to the number of rows of `Mdl.X` and columns equal to the number of distinct classes in the training data (`size(Mdl.ClassNames, 1)`).

`Cost(j, k)` is the expected misclassification cost of the observation in row  $j$  of `Mdl.X` being predicted into class  $k$  (i.e., in class `Mdl.ClassNames(k)`).

## Examples

### Label Training Sample Observations for Naive Bayes

Load Fisher's iris data set.

```
load fisheriris
X = meas; % Predictors
Y = species; % Response
```

Train a naive Bayes classifier. It is good practice to specify the class order. Assume that each predictor is conditionally, normally distributed given its label.

```
Mdl = fitcnb(X,Y,...
    'ClassNames',{'setosa','versicolor','virginica'});
```

Mdl is a ClassificationNaiveBayes classifier.

Predict the training sample labels. Display the results for the 10 observations.

```
label = resubPredict(Mdl);
rng(1); % For reproducibility
idx = randsample(size(X,1),10);
table(Y(idx),label(idx),'VariableNames',...
    {'TrueLabel','PredictedLabel'})
```

ans =

10x2 table

TrueLabel	PredictedLabel
'setosa'	'setosa'
'versicolor'	'versicolor'
'virginica'	'virginica'
'setosa'	'setosa'
'versicolor'	'versicolor'
'setosa'	'setosa'
'versicolor'	'versicolor'
'versicolor'	'versicolor'
'setosa'	'setosa'
'setosa'	'setosa'

## Estimate In-Sample Posterior Probabilities of Naive Bayes Classifiers

Load Fisher's iris data set.

```
load fisheriris
X = meas;    % Predictors
Y = species; % Response
```

Train a naive Bayes classifier. It is good practice to specify the class order. Assume that each predictor is conditionally, normally distributed given its label.

```
Mdl = fitcnb(X,Y,...
    'ClassNames',{'setosa','versicolor','virginica'});
```

Mdl is a ClassificationNaiveBayes classifier.

Estimate posterior probabilities and expected misclassification costs for the training data. Display the results for 10 observations.

```
[label,Posterior,MisclassCost] = resubPredict(Mdl);
rng(1); % For reproducibility
idx = randsample(size(X,1),10);
Mdl.ClassNames
table(Y(idx),label(idx),Posterior(idx,:), 'VariableNames',...
    {'TrueLabel','PredictedLabel','PosteriorProbability'})
MisclassCost(idx,:)
```

```
ans =
```

```
3x1 cell array
```

```
{'setosa' }
{'versicolor'}
{'virginica' }
```

```
ans =
```

```
10x3 table
```

TrueLabel	PredictedLabel	PosteriorProbability		
'setosa'	'setosa'	1	3.8821e-16	5.5878e-24
'versicolor'	'versicolor'	1.2516e-54	1	4.5001e-06
'virginica'	'virginica'	5.5646e-188	0.00058232	0.99942
'setosa'	'setosa'	1	4.5352e-20	3.1301e-27
'versicolor'	'versicolor'	5.0002e-69	0.99989	0.00010716



'setosa'	'setosa'	1	2.9813e-18	2.1524e-25
'versicolor'	'versicolor'	4.6313e-60	0.99999	7.5413e-06
'versicolor'	'versicolor'	7.9205e-100	0.94293	0.057072
'setosa'	'setosa'	1	1.799e-19	6.0606e-27
'setosa'	'setosa'	1	1.5426e-17	1.2744e-24

ans =

0.0000	1.0000	1.0000
1.0000	0.0000	1.0000
1.0000	0.9994	0.0006
0.0000	1.0000	1.0000
1.0000	0.0001	0.9999
0.0000	1.0000	1.0000
1.0000	0.0000	1.0000
1.0000	0.0571	0.9429
0.0000	1.0000	1.0000
0.0000	1.0000	1.0000

The order of the columns of `Posterior` and `MisclassCost` corresponds to the order of the classes in `Mdl.ClassNames`.

## Definitions

### Misclassification Cost

A misclassification cost is the relative severity of a classifier labeling an observation into the wrong class.

There are two types of misclassification costs: true and expected. Let  $K$  be the number of classes.

- True misclassification cost — A  $K$ -by- $K$  matrix, where element  $(i,j)$  indicates the misclassification cost of predicting an observation into class  $j$  if its true class is  $i$ . The software stores the misclassification cost in the property `Mdl.Cost`, and used in computations. By default, `Mdl.Cost(i,j) = 1` if  $i \neq j$ , and `Mdl.Cost(i,j) = 0` if  $i = j$ . In other words, the cost is 0 for correct classification, and 1 for any incorrect classification.

- Expected misclassification cost — A  $K$ -dimensional vector, where element  $k$  is the weighted average misclassification cost of classifying an observation into class  $k$ , weighted by the class posterior probabilities. In other words,

$$c_k = \sum_{j=1}^K \hat{P}(Y = j | x_1, \dots, x_P) \text{Cost}_{jk}.$$

the software classifies observations to the class corresponding with the lowest expected misclassification cost.

## Posterior Probability

The posterior probability is the probability that an observation belongs in a particular class, given the data.

For naive Bayes, the posterior probability that a classification is  $k$  for a given observation  $(x_1, \dots, x_P)$  is

$$\hat{P}(Y = k | x_1, \dots, x_P) = \frac{P(X_1, \dots, X_P | y = k) \pi(Y = k)}{P(X_1, \dots, X_P)},$$

where:

- $P(X_1, \dots, X_P | y = k)$  is the conditional joint density of the predictors given they are in class  $k$ . `Mdl.DistributionNames` stores the distribution names of the predictors.
- $\pi(Y = k)$  is the class prior probability distribution. `Mdl.Prior` stores the prior distribution.
- $P(X_1, \dots, X_P)$  is the joint density of the predictors. The classes are discrete, so

$$P(X_1, \dots, X_P) = \sum_{k=1}^K P(X_1, \dots, X_P | y = k) \pi(Y = k).$$

## Prior Probability

The prior probability is the believed relative frequency that observations from a class occur in the population for each class.

## References

- [1] Hastie, T., R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*, Second Edition. NY: Springer, 2008.

## See Also

`ClassificationNaiveBayes` | `CompactClassificationNaiveBayes` | `fitcnb` | `loss` | `predict`

## Topics

“Naive Bayes Classification” on page 21-2  
“Grouping Variables” on page 2-59

## resubPredict

**Class:** ClassificationSVM

Predict support vector machine classifier resubstitution responses

### Syntax

```
label = resubPredict(SVMModel)
[label, Score] = resubPredict(SVMModel)
```

### Description

`label = resubPredict(SVMModel)` returns a vector of predicted class labels (`label`) for the trained support vector machine (SVM) classifier `SVMModel` using the predictor data `SVMModel.X`.

`[label, Score] = resubPredict(SVMModel)` additionally returns class likelihood measures, that is, either scores or posterior probabilities.

### Input Arguments

**SVMModel** — Full, trained SVM classifier

ClassificationSVM classifier

Full, trained SVM classifier, specified as a `ClassificationSVM` model trained using `fitcsvm`.

### Output Arguments

**label** — Predicted class labels

categorical array | character array | logical vector | vector of numeric values | cell array of character vectors

Predicted class labels, returned as a categorical or character array, logical or numeric vector, or cell array of character vectors.

label:

- Is the same data type as the observed class labels (`SVMMModel.Y`)
- Has length equal to the number of rows of `SVMMModel.X`

For one-class learning, the elements of `label` are the one class represented in `SVMMModel.Y`

### **score** — Predicted class scores or posterior probabilities

numeric column vector | numeric matrix

Predicted class scores on page 32-5874 or posterior probabilities on page 32-5875, returned as a numeric column vector or numeric matrix.

- For one-class learning, `Score` is a column vector with the same number of rows as `SVMMModel.X`. The elements are the positive class scores for the corresponding observations. You cannot obtain posterior probabilities for one-class learning.
- For two-class learning, `Score` is a two column matrix with the same number of rows as `SVMMModel.X`.
  - If you fit the optimal score-to-posterior probability transformation function using `fitPosterior` or `fitSVMPosterior`, then `Score` contains class posterior probabilities. That is, if the value of `SVMMModel.ScoreTransform` is not `none`, then the elements of the first and second columns of `Score` are the negative class (`SVMMModel.ClassNames{1}`) and positive class (`SVMMModel.ClassNames{2}`) posterior probabilities for the corresponding observations, respectively.
  - Otherwise, the elements of the first column are the negative class scores and the elements of the second column are the positive class scores for the corresponding observations.

If `SVMMModel.KernelParameters.Function` is 'linear', then the software estimates the classification score for the observation  $x$  using

$$f(x) = (x/s)' \beta + b.$$

`SVMMModel` stores  $\beta$ ,  $b$ , and  $s$  in the properties `Beta`, `Bias`, and `KernelParameters.Scale`, respectively.

Data Types: `double` | `single`

## Examples

### Label Training Sample Observations for SVM Classifiers Using resubPredict

Load the ionosphere data set.

```
load ionosphere
```

Train an SVM classifier. It is good practice to specify the class order and standardize the data.

```
SVModel = fitcsvm(X,Y,'ClassNames',{'b','g'},'Standardize',true);
```

SVModel is a ClassificationSVM classifier. The positive class is 'g'.

Predict the training sample labels and scores. Display the results for the first 10 observations.

```
[label,score] = resubPredict(SVModel);  
table(Y(1:10),label(1:10),score(1:10,2),'VariableNames',...  
      {'TrueLabel','PredictedLabel','Score'})
```

```
ans =
```

```
10x3 table
```

TrueLabel	PredictedLabel	Score
'g'	'g'	1.4861
'b'	'b'	-1.0004
'g'	'g'	1.8685
'b'	'b'	-2.6458
'g'	'g'	1.2805
'b'	'b'	-1.4617
'g'	'g'	2.1672
'b'	'b'	-5.7085
'g'	'g'	2.4797
'b'	'b'	-2.7811

## Estimate In-Sample Posterior Probabilities of SVM Classifiers

Load the `ionosphere` data set.

```
load ionosphere
```

Train an SVM classifier. It is good practice to specify the class order and standardize the data.

```
SVMMModel = fitcsvm(X,Y,'ClassNames',{'b','g'},'Standardize',true);
```

`SVMMModel` is a `ClassificationSVM` classifier. The positive class is 'g'.

Fit the optimal score-to-posterior-probability transformation function.

```
rng(1); % For reproducibility
ScoreSVMMModel = fitPosterior(SVMMModel)
```

```
ScoreSVMMModel =

ClassificationSVM
    ResponseName: 'Y'
  CategoricalPredictors: []
        ClassNames: {'b' 'g'}
    ScoreTransform: '@(S) sigmoid(S,-9.481802e-01,-1.218745e-01) '
    NumObservations: 351
           Alpha: [90x1 double]
           Bias: -0.1343
  KernelParameters: [1x1 struct]
           Mu: [1x34 double]
           Sigma: [1x34 double]
    BoxConstraints: [351x1 double]
  ConvergenceInfo: [1x1 struct]
  IsSupportVector: [351x1 logical]
           Solver: 'SMO'
```

Since the classes are inseparable, the score transformation function (`ScoreSVMMModel.ScoreTransform`) is the sigmoid function.

Estimate scores and positive class posterior probabilities for the training data. Display the results for the first 10 observations.

```
[label,scores] = resubPredict(SVMModel);
[~,postProbs] = resubPredict(ScoreSVMModel);
table(Y(1:10),label(1:10),scores(1:10,2),postProbs(1:10,2),'VariableNames',...
      {'TrueLabel','PredictedLabel','Score','PosteriorProbability'})
```

```
ans =
```

```
10x4 table
```

TrueLabel	PredictedLabel	Score	PosteriorProbability
'g'	'g'	1.4861	0.82215
'b'	'b'	-1.0004	0.30436
'g'	'g'	1.8685	0.86916
'b'	'b'	-2.6458	0.084183
'g'	'g'	1.2805	0.79184
'b'	'b'	-1.4617	0.22028
'g'	'g'	2.1672	0.89814
'b'	'b'	-5.7085	0.0050122
'g'	'g'	2.4797	0.92223
'b'	'b'	-2.7811	0.074805

## Definitions

### Classification Score

The SVM classification score for classifying observation  $x$  is the signed distance from  $x$  to the decision boundary ranging from  $-\infty$  to  $+\infty$ . A positive score for a class indicates that  $x$  is predicted to be in that class. A negative score indicates otherwise.

The score for predicting  $x$  into the positive class, also the numerical, predicted response for  $x$ ,  $f(x)$ , is the trained SVM classification function

$$f(x) = \sum_{j=1}^n \alpha_j y_j G(x_j, x) + b,$$



where  $(\alpha_1, \dots, \alpha_n, b)$  are the estimated SVM parameters,  $G(x_j, x)$  is the dot product in the predictor space between  $x$  and the support vectors, and the sum includes the training set observations. The score for predicting  $x$  into the negative class is  $-f(x)$ .

If  $G(x_j, x) = x_j'x$  (the linear kernel), then the score function reduces to  $f(x) = (x/s)' \beta + b$ .

$s$  is the kernel scale and  $\beta$  is the vector of fitted linear coefficients.

## Posterior Probability

The probability that an observation belongs in a particular class, given the data.

For SVM, the posterior probability is a function of the score,  $P(s)$ , that observation  $j$  is in class  $k = \{-1, 1\}$ .

- For separable classes, the posterior probability is the step function

$$P(s_j) = \begin{cases} 0; & s < \max_{y_k=-1} s_k \\ \pi; & \max_{y_k=-1} s_k \leq s_j \leq \min_{y_k=+1} s_k \\ 1; & s_j > \min_{y_k=+1} s_k \end{cases}$$

where:

- $s_j$  is the score of observation  $j$ .
- $+1$  and  $-1$  denote the positive and negative classes, respectively.
- $\pi$  is the prior probability that an observation is in the positive class.
- For inseparable classes, the posterior probability is the sigmoid function

$$P(s_j) = \frac{1}{1 + \exp(As_j + B)},$$

where the parameters  $A$  and  $B$  are the slope and intercept parameters.

## Prior Probability

The prior probability is the believed relative frequency that observations from a class occur in the population for each class.

## Algorithms

- By default and irrespective of the model kernel function, MATLAB uses the dual representation of the score function to classify observations based on trained SVM models, specifically

$$\hat{f}(x) = \sum_{j=1}^n \hat{\alpha}_j y_j G(x, x_j) + \hat{b}.$$

This prediction method requires, among other things, the trained support vectors and  $\alpha$  coefficients (see the `SupportVectors` and `Alpha` properties of the SVM model).

If you are using a linear SVM model for classification and there are many support vectors, then this prediction method can be slow. To efficiently classify observations based on a linear SVM model, remove the support vectors from the model object using `discardSupportVectors`. The resulting model uses the simple linear score function for prediction instead, specifically

$$\hat{f}(x) = x' \hat{\beta} + \hat{b}.$$

For more details, see “Support Vector Machines for Binary Classification” on page 32-2129.

- By default, the software computes optimal posterior probabilities using Platt’s method [1]:
  - 1 Performing 10-fold cross validation
  - 2 Fitting the sigmoid function parameters to the scores returned from the cross validation
  - 3 Estimating the posterior probabilities by entering the cross-validation scores into the fitted sigmoid function
- The software incorporates prior probabilities in the SVM objective function during training.
- For SVM, `predict` classifies observations into the class yielding the largest score (i.e., the largest posterior probability). The software accounts for misclassification costs by applying the average-cost correction before training the classifier. That is, given the class prior vector  $P$ , misclassification cost matrix  $C$ , and observation weight vector  $w$ , the software defines a new vector of observation weights ( $W$ ) such that

$$W_j = w_j P_j \sum_{k=1}^K C_{jk}.$$

## References

- [1] Platt, J. “Probabilistic outputs for support vector machines and comparisons to regularized likelihood methods.” *In Advances in Large Margin Classifiers*. MIT Press, 1999, pp. 61–74.

## See Also

`ClassificationSVM` | `CompactClassificationSVM` | `fitPosterior` | `fitSVMPosterior` | `fitcsvm` | `predict`

## resubPredict

**Class:** ClassificationTree

Predict resubstitution response of tree

### Syntax

```
label = resubPredict(tree)
[label,posterior] = resubPredict(tree)
[label,posterior,node] = resubPredict(tree)
[label,posterior,node,cnum] = resubPredict(tree)
[label,...] = resubPredict(tree,Name,Value)
```

### Description

`label = resubPredict(tree)` returns the labels tree predicts for the data `tree.X`. `label` is the predictions of tree on the data that `fitctree` used to create `tree`.

`[label,posterior] = resubPredict(tree)` returns the posterior class probabilities for the predictions.

`[label,posterior,node] = resubPredict(tree)` returns the node numbers of tree for the resubstituted data.

`[label,posterior,node,cnum] = resubPredict(tree)` returns the predicted class numbers for the predictions.

`[label,...] = resubPredict(tree,Name,Value)` returns resubstitution predictions with additional options specified by one or more `Name, Value` pair arguments.

## Input Arguments

### **tree**

A classification tree constructed by `fitctree`.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### **Subtrees — Pruning level**

0 (default) | vector of nonnegative integers | 'all'

Pruning level, specified as the comma-separated pair consisting of 'Subtrees' and a vector of nonnegative integers in ascending order or 'all'.

If you specify a vector, then all elements must be at least 0 and at most `max(tree.PruneList)`. 0 indicates the full, unpruned tree and `max(tree.PruneList)` indicates the completely pruned tree (i.e., just the root node).

If you specify 'all', then `ClassificationTree.resubPredict` operates on all subtrees (i.e., the entire pruning sequence). This specification is equivalent to using `0:max(tree.PruneList)`.

`ClassificationTree.resubPredict` prunes tree to each level indicated in `Subtrees`, and then estimates the corresponding output arguments. The size of `Subtrees` determines the size of some output arguments.

To invoke `Subtrees`, the properties `PruneList` and `PruneAlpha` of `tree` must be nonempty. In other words, grow tree by setting 'Prune', 'on', or by pruning tree using `prune`.

Example: 'Subtrees', 'all'

## Output Arguments

### **label**

The response `tree` predicts for the training data. `label` is the same data type as the training response data `tree.Y`.

If the `Subtrees` name-value argument contains  $m > 1$  entries, `label` has  $m$  columns, each of which represents the predictions of the corresponding subtree. Otherwise, `label` is a vector.

### **posterior**

Matrix or array of posterior probabilities for classes `tree` predicts.

If the `Subtrees` name-value argument is a scalar or is missing, `posterior` is an  $n$ -by- $k$  matrix, where  $n$  is the number of rows in the training data `tree.X`, and  $k$  is the number of classes.

If `Subtrees` contains  $m > 1$  entries, `posterior` is an  $n$ -by- $k$ -by- $m$  array, where the matrix for each  $m$  gives posterior probabilities for the corresponding subtree.

### **node**

The node numbers of `tree` where each data row resolves.

If the `Subtrees` name-value argument is a scalar or is missing, `node` is a numeric column vector with  $n$  rows, the same number of rows as `tree.X`.

If `Subtrees` contains  $m > 1$  entries, `node` is a  $n$ -by- $m$  matrix. Each column represents the node predictions of the corresponding subtree.

### **cnum**

The class numbers that `tree` predicts for the resubstituted data.

If the `Subtrees` name-value argument is a scalar or is missing, `cnum` is a numeric column vector with  $n$  rows, the same number of rows as `tree.X`.

If `Subtrees` contains  $m > 1$  entries, `cnum` is a  $n$ -by- $m$  matrix. Each column represents the class predictions of the corresponding subtree.

## Examples

### Compute Number of Misclassified Observations

Find the total number of misclassifications of the Fisher iris data for a classification tree.

```
load fisheriris
tree = fitctree(meas,species);
Ypredict = resubPredict(tree); % The predictions
Ysame = strcmp(Ypredict,species); % True when ==
sum(~Ysame) % How many are different?
```

```
ans =
```

```
3
```

### Compare In-Sample Posterior Probabilities for Each Subtree

Load Fisher's iris data set. Partition the data into training (50%)

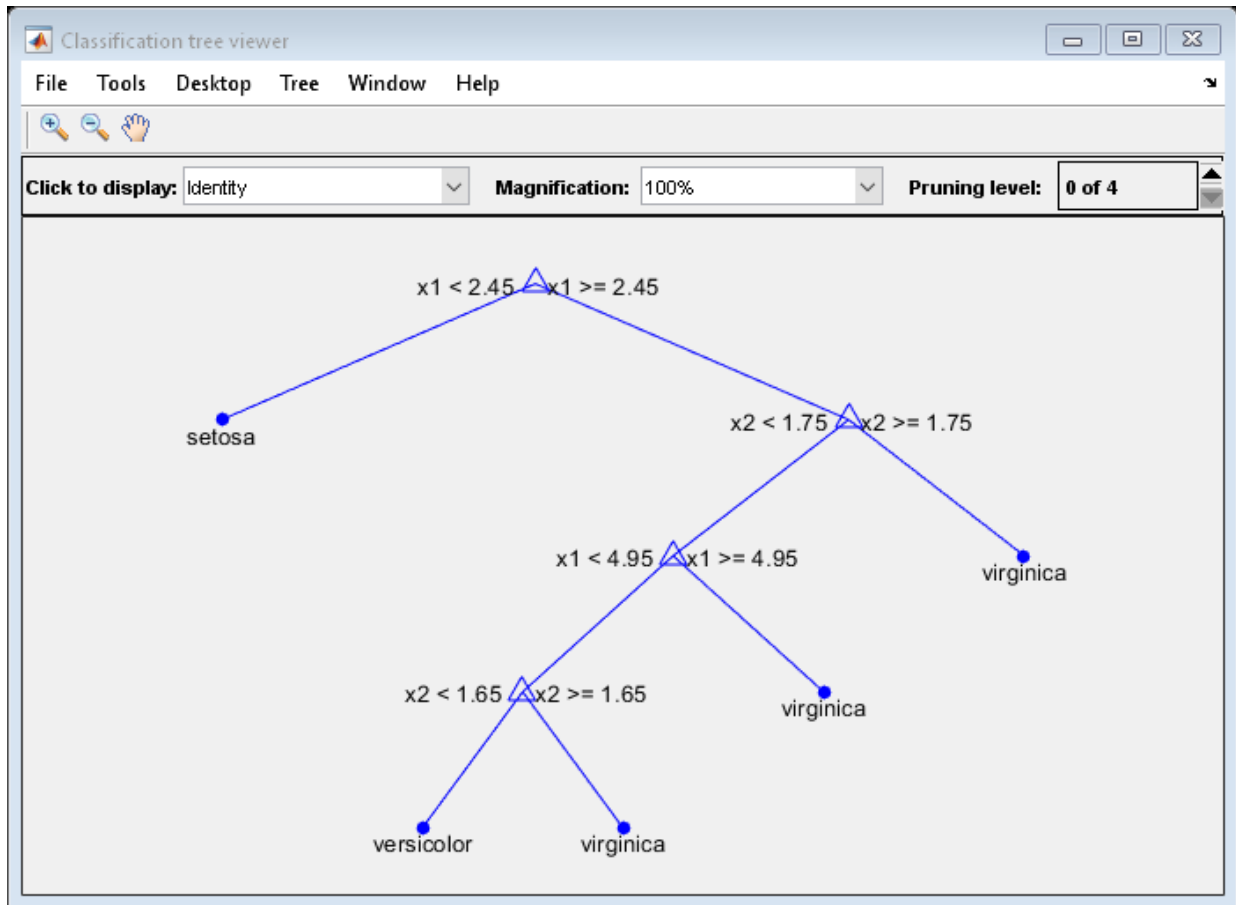
```
load fisheriris
```

Grow a classification tree using the all petal measurements.

```
Mdl = fitctree(meas(:,3:4),species);
n = size(meas,1); % Sample size
K = numel(Mdl.ClassNames); % Number of classes
```

View the classification tree.

```
view(Mdl, 'Mode', 'graph');
```



The classification tree has four pruning levels. Level 0 is the full, unpruned tree (as displayed). Level 4 is just the root node (i.e., no splits).

Estimate the posterior probabilities for each class using the subtrees pruned to levels 1 and 3.

```
[~,Posterior] = resubPredict(Mdl,'SubTrees',[1 3]);
```

Posterior is an n-by- K-by- 2 array of posterior probabilities. Rows of Posterior correspond to observations, columns correspond to the classes with order `Mdl.ClassNames`, and pages correspond to pruning level.



Display the class posterior probabilities for iris 125 using each subtree.

```
Posterior(125, :, :)

ans(:, :, 1) =

    0    0.0217    0.9783

ans(:, :, 2) =

    0    0.5000    0.5000
```

The decision stump (page 2 of Posterior) has trouble predicting whether iris 125 is versicolor or virginica.

## Definitions

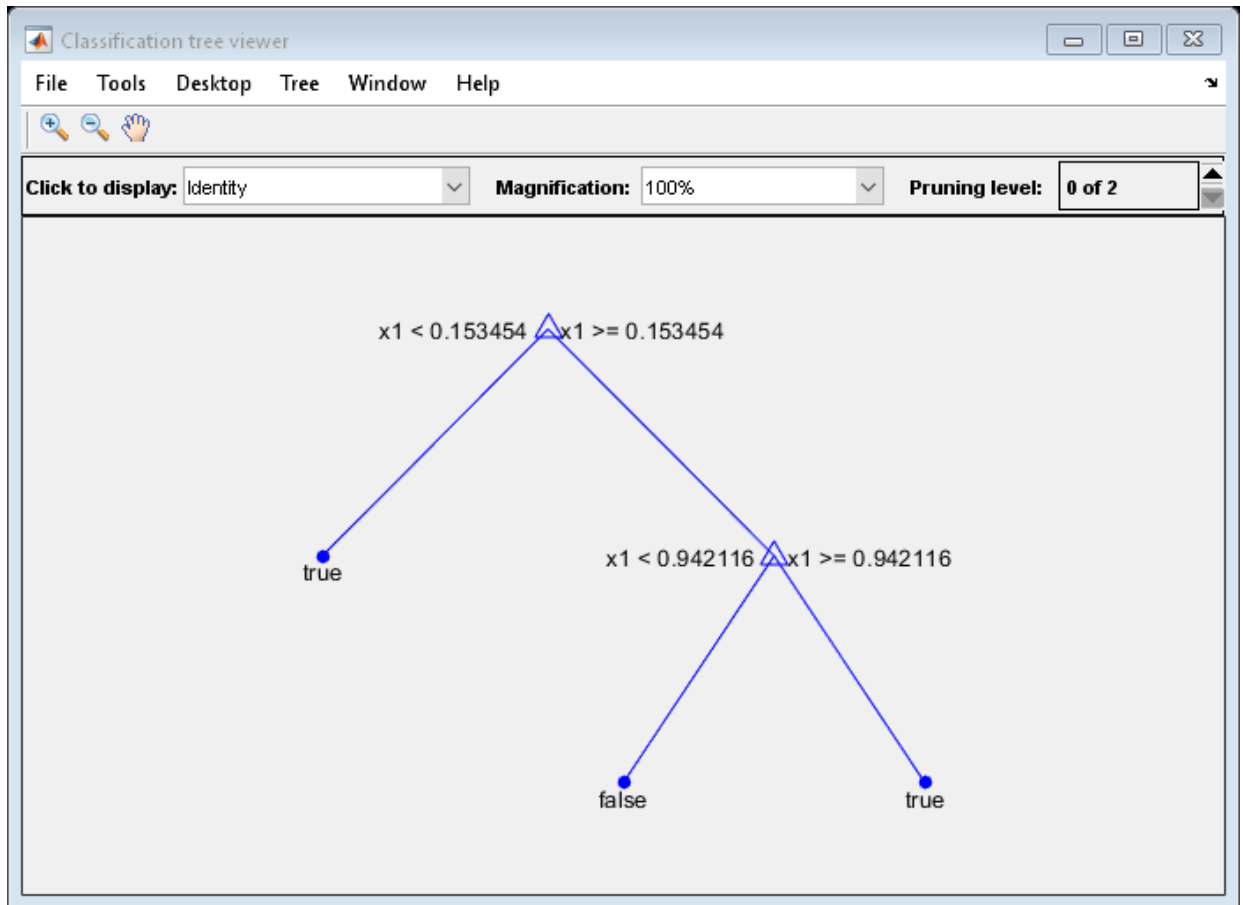
### Posterior Probability

The posterior probability of the classification at a node is the number of training sequences that lead to that node with this classification, divided by the number of training sequences that lead to that node.

For example, consider classifying a predictor  $X$  as true when  $X < 0.15$  or  $X > 0.95$ , and  $X$  is false otherwise.

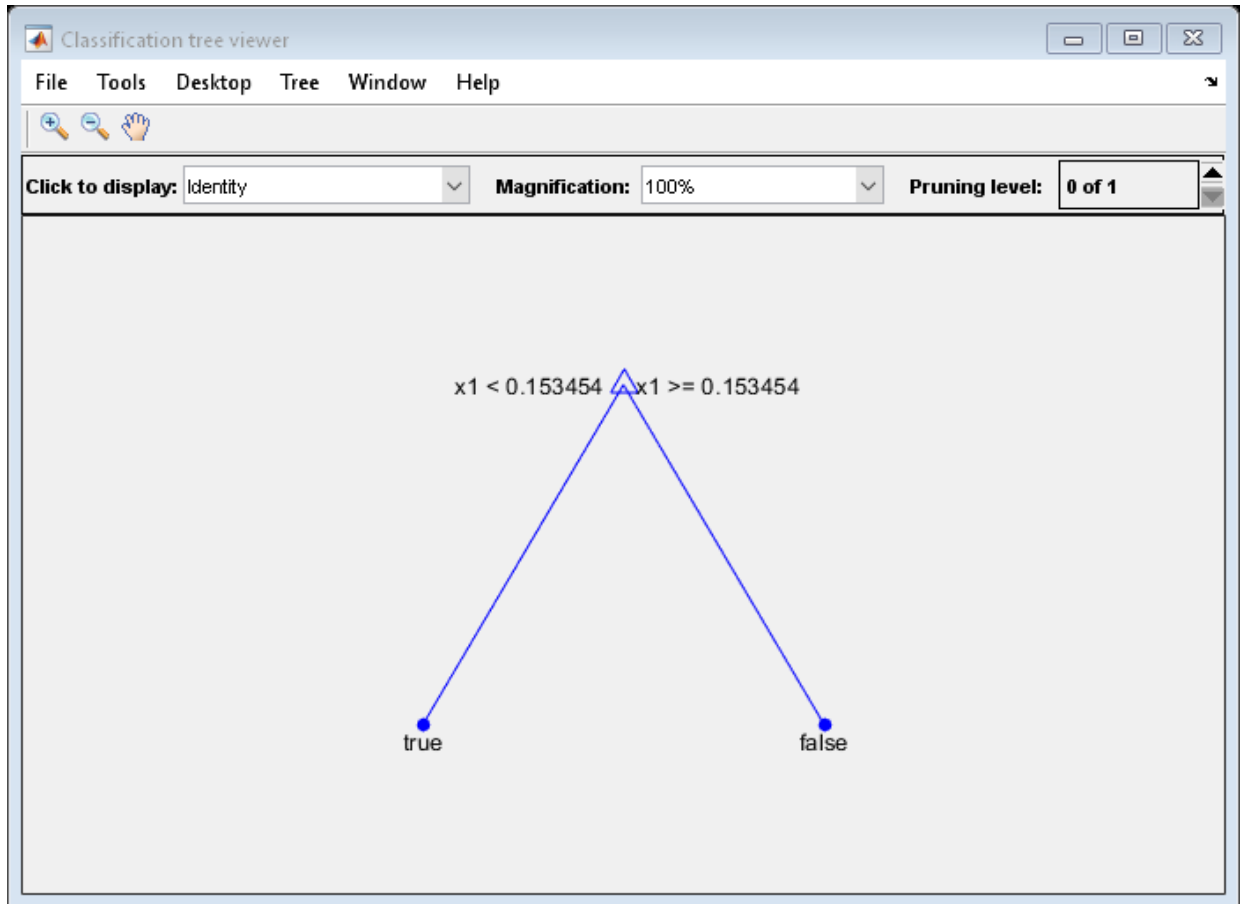
- 1 Generate 100 random points and classify them:

```
rng(0) % For reproducibility
X = rand(100,1);
Y = (abs(X - .55) > .4);
tree = fitctree(X,Y);
view(tree, 'Mode', 'graph')
```



2 Prune the tree:

```
tree1 = prune(tree, 'Level', 1);  
view(tree1, 'Mode', 'graph')
```



The pruned tree correctly classifies observations that are less than 0.15 as *true*. It also correctly classifies observations between .15 and .94 as *false*. However, it incorrectly classifies observations that are greater than .94 as *false*. Therefore the score for observations that are greater than .15 should be about  $.05/.85=.06$  for *true*, and about  $.8/.85=.94$  for *false*.

- 3 Compute the prediction scores for the first 10 rows of  $X$ :

```
[~,score] = predict(tree1,X(1:10));
[score X(1:10,:)]
```

```
ans =  
  
    0.9059    0.0941    0.8147  
    0.9059    0.0941    0.9058  
         0     1.0000    0.1270  
    0.9059    0.0941    0.9134  
    0.9059    0.0941    0.6324  
         0     1.0000    0.0975  
    0.9059    0.0941    0.2785  
    0.9059    0.0941    0.5469  
    0.9059    0.0941    0.9575  
    0.9059    0.0941    0.9649
```

Indeed, every value of  $X$  (the rightmost column) that is less than 0.15 has associated scores (the left and center columns) of 0 and 1, while the other values of  $X$  have associated scores of 0.94 and 0.06.

## See Also

`fitctree` | `predict` | `resubEdge` | `resubLoss` | `resubMargin`

# resubPredict

**Class:** RegressionEnsemble

Predict response of ensemble by resubstitution

## Syntax

```
Yfit = resubPredict(ens)
Yfit = resubPredict(ens,Name,Value)
```

## Description

`Yfit = resubPredict(ens)` returns the response `ens` predicts for the data `ens.X`. `Yfit` is the predictions of `ens` on the data that `fitrensemble` used to create `ens`.

`Yfit = resubPredict(ens,Name,Value)` predicts responses with additional options specified by one or more `Name,Value` pair arguments.

## Input Arguments

### **ens**

A regression ensemble created with `fitrensemble`.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

**learners**

Indices of weak learners in the ensemble ranging from 1 to NumTrained. `oobLoss` uses only these learners for calculating loss.

**Default:** `1:NumTrained`

## Output Arguments

**Yfit**

A vector of predicted responses to the training data, with `ens.X` elements.

## Examples

**Find Mean-Squared Error of Resubstitution Predictions**

Find the resubstitution predictions of mileage from the `carsmall` data, and look at their mean-squared difference from the training data.

Load the `carsmall` data set and select horsepower and vehicle weight as predictors.

```
load carsmall
X = [Horsepower Weight];
```

Train an ensemble of regression trees.

```
ens = fitensemble(X,MPG,'Method','LSBoost','Learners','Tree');
```

Find the resubstitution predictions of MPG.

```
Yfit = resubPredict(ens);
```

Calculate the mean-squared difference of the resubstitution predictions from the training data.

```
MSE = mean((Yfit - ens.Y).^2)
```

```
MSE = 0.5836
```

Confirm that the result is the same as the result of `resubLoss`.

```
resubLoss(ens)
```

```
ans = 0.5836
```

## See Also

[predict](#) | [resubLoss](#) | [resubPredict](#)

## resubPredict

**Class:** RegressionGP

Resubstitution prediction from a trained Gaussian process regression model

### Syntax

```
ypred = resubPredict(gprMdl)
[ypred, ysd] = resubPredict(gprMdl)
[ypred, ysd, yint] = predict(gprMdl)
[ypred, ysd, yint] = predict(gprMdl, Name, Value)
```

### Description

`ypred = resubPredict(gprMdl)` returns the predicted responses, `ypred`, for the trained Gaussian process regression (GPR) model, `gprMdl`.

`[ypred, ysd] = resubPredict(gprMdl)` also returns the estimated standard deviations of the predicted responses corresponding to the rows of `gprMdl.X`.

`[ypred, ysd, yint] = predict(gprMdl)` also returns the 95% prediction intervals, `yint`, for the true responses corresponding to each row of training data, `gprMdl.X`.

`[ypred, ysd, yint] = predict(gprMdl, Name, Value)` returns the prediction intervals with additional options, specified by one or more `Name, Value` pair arguments. For example, you can specify the confidence level of the prediction interval.

### Input Arguments

**gprMdl** — Gaussian process regression model

RegressionGP object

Gaussian process regression model, specified as a RegressionGP object.



## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### **Alpha** — Confidence level

0.05 (default) | scalar value in the range from 0 to 1

Confidence level for the prediction intervals, specified as the comma-separated pair consisting of 'Alpha' and a scalar value in the range from 0 to 1.

Example: 'Alpha', 0.01 specifies 99% prediction intervals.

Data Types: single | double

## Output Arguments

### **ypred** — Predicted response values

$n$ -by-1 vector

Predicted response values, returned as an  $n$ -by-1 vector, where  $n$  is the number of observations in the training data.

### **ysd** — Standard deviation of the predicted response values

$n$ -by-1 vector

Standard deviation of the predicted response values corresponding to the rows of `gprMdl.X`, returned as an  $n$ -by-1 vector. `ysd(i)`,  $i = 1, 2, \dots, n$ , contains the estimated standard deviation of the new response corresponding to the predictor values at the  $i^{\text{th}}$  observation in the training data.

### **yint** — Prediction intervals for the true response values

$n$ -by-2 matrix

Prediction intervals for the true response values corresponding to the rows of `gprMdl.X`, returned as an  $n$ -by-2 matrix, where  $n$  is the number of observations in the training data. The first column of `yint` contains the lower limits and the second column contains the upper limits of the prediction intervals.

## Examples

### Plot Predicted Response and Prediction Intervals

This example uses "Housing" data set [1] from the UCI machine learning archive [2] described in <http://archive.ics.uci.edu/ml/datasets/Housing>. Download the data and save it in your current directory as a data file named `housing.data`.

The dataset has 506 observations. The first 13 columns contain the predictor values and the last column contains the response values. The goal is to predict the median value of owner-occupied homes in the Boston suburb area as a function of 13 predictors.

Load the data and define the response vector and predictor matrix.

```
load('housing.data');
X = housing(:,1:13);
y = housing(:,end);
```

Train a GPR model using subset of regressors ('sr') approximation method with Matern 3/2 ('Matern32') kernel function. Predict using the fully independent conditional ('fic') method.

```
gprMdl = fitrgp(X,y,'KernelFunction','Matern32',...
'FitMethod','sr','PredictMethod','fic');
```

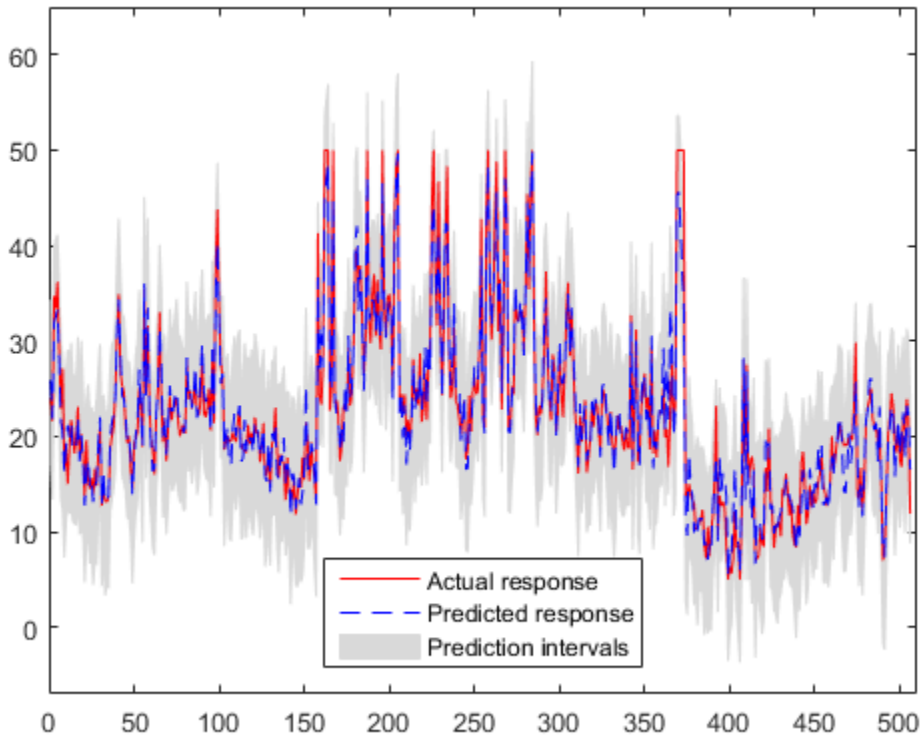
Predict the responses using the trained GPR model. Compute the 99% prediction intervals.

```
[ypred,~,yint] = resubPredict(gprMdl,'Alpha',0.01);
```

Plot the actual response values along with predictions from the GPR model.

```
figure;
h1 = area([yint(:,1) yint(:,2)-yint(:,1)],-8,...
'FaceColor',[0.85,0.85,0.85],'EdgeColor',[0.85,0.85,0.85]);
hold on;
h1(1).FaceColor = 'none'; % remove color from bottom area
h1(1).EdgeColor = 'none';
h2 = plot(y,'r'); % Plot original response values
h3 = plot(ypred,'b--'); % Plot predicted response values
legend([h2 h3 h1(2)],'Actual response','Predicted response',...
'Prediction intervals','Location','South');
```

```
axis([0 510 -7 65]);  
hold off
```



The gray area shows the 99% prediction intervals.

## Tips

- You can choose the prediction method while training the GPR model using the `PredictMethod` name-value pair argument in `fitrgp`. The default prediction method is 'exact' for  $n \leq 10000$ , where  $n$  is the number of observations in the training data, and 'bcd' (block coordinate descent), otherwise.

- Computation of standard deviations, `ysd`, and prediction intervals, `yint`, is not supported when `PredictMethod` is `'bcd'`.

## Alternatives

To compute the predicted responses for new data, use `predict`.

## References

- [1] Harrison, D. and D.L., Rubinfeld. "Hedonic prices and the demand for clean air." *J. Environ. Economics & Management*. Vol.5, 1978, pp. 81-102.
- [2] Lichman, M. UCI Machine Learning Repository, Irvine, CA: University of California, School of Information and Computer Science, 2013. <http://archive.ics.uci.edu/ml>.

## See Also

`RegressionGP` | `fitrgp` | `predict` | `resubLoss`

**Introduced in R2015b**

# resubPredict

**Class:** RegressionSVM

Predict resubstitution response of support vector machine regression model

## Syntax

```
yfit = resubPredict mdl
```

## Description

`yfit = resubPredict(mdl)` returns a vector of predicted response values, `yfit`, for the trained support vector machine (SVM) regression model `mdl` using the predictor data stored in `mdl.X`.

## Input Arguments

**mdl** — Full, trained SVM regression model

RegressionSVM model

Full, trained SVM regression model, specified as a `RegressionSVM` model returned by `fitrsvm`.

## Output Arguments

**yfit** — Predicted response

vector of numeric values

Predicted responses, returned as a vector of numeric values. The length of `yfit` is equal to the number of observations in the training data, `mdl.NumObservations`.

For details about how to predict responses, see “Equation 25-1” on page 25-4 and “Equation 25-2” on page 25-6 in “Understanding Support Vector Machine Regression” on page 25-2.

## Examples

### Resubstitution Predictions for SVM Regression Model

This example shows how to train an SVM regression model, then use the model to generate predicted response values from the training data.

This example uses the abalone data from the UCI Machine Learning Repository. Download the data and save it in your current directory with the name 'abalone.data'. Read the data into a table.

```
tbl = readtable('abalone.data', 'Filetype', 'text', 'ReadVariableNames', false);  
rng default % for reproducibility
```

The sample data contains 4177 observations. All of the predictor variables are continuous except for sex, which is a categorical variable with possible values 'M' (for males), 'F' (for females), and 'I' (for infants). The goal is to predict the number of rings on the abalone, and thereby determine its age, using physical measurements.

Train an SVM regression model to the data, using a Gaussian kernel function with an automatic kernel scale. Standardize the data.

```
mdl = fitrsvm(tbl, 'Var9', 'KernelFunction', 'gaussian', 'KernelScale', 'auto', 'Standardize'
```

Use the trained model to predict response values based on the original data.

```
yfit = resubPredict(mdl);
```

Display the first ten predicted responses alongside the actual response values.

```
[mdl.Y(1:10), yfit(1:10)]
```

```
ans =
```

```
15.0000    8.1836  
 7.0000    8.3545  
 9.0000   10.9383
```

10.0000	9.3446
7.0000	6.4042
8.0000	7.7910
20.0000	13.8275
16.0000	11.7959
9.0000	9.5724
19.0000	13.6909

The left column shows the actual response and the right column shows the corresponding predicted response.

## References

- [1] Nash, W.J., T. L. Sellers, S. R. Talbot, A. J. Cawthorn, and W. B. Ford. *The Population Biology of Abalone (Haliotis species) in Tasmania. I. Blacklip Abalone (H. rubra) from the North Coast and Islands of Bass Strait*, Sea Fisheries Division, Technical Report No. 48, 1994.
- [2] Waugh, S. *Extending and benchmarking Cascade-Correlation*, Ph.D. thesis, Computer Science Department, University of Tasmania, 1995.
- [3] Clark, D., Z. Schreter, A. Adams. *A Quantitative Comparison of Dystal and Backpropagation*, submitted to the Australian Conference on Neural Networks, 1996.
- [4] Lichman, M. *UCI Machine Learning Repository*, [<http://archive.ics.uci.edu/ml>]. Irvine, CA: University of California, School of Information and Computer Science.

## See Also

RegressionSVM | fitrsvm | predict | resubLoss

Introduced in R2015b

## resubPredict

**Class:** RegressionTree

Predict resubstitution response of tree

### Syntax

```
Yfit = resubPredict(tree)
[Yfit,node] = resubPredict(tree)
[Yfit,node] = resubPredict(tree,Name,Value)
```

### Description

`Yfit = resubPredict(tree)` returns the responses `tree` predicts for the data `tree.X`. `Yfit` is the predictions of `tree` on the data that `fitrtree` used to create `tree`.

`[Yfit,node] = resubPredict(tree)` returns the node numbers of `tree` for the resubstituted data.

`[Yfit,node] = resubPredict(tree,Name,Value)` predicts with additional options specified by one or more `Name,Value` pair arguments.

### Input Arguments

**tree**

A regression tree constructed using `fitrtree`.

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.



**Subtrees — Pruning level**

0 (default) | vector of nonnegative integers | 'all'

Pruning level, specified as the comma-separated pair consisting of 'Subtrees' and a vector of nonnegative integers in ascending order or 'all'.

If you specify a vector, then all elements must be at least 0 and at most `max(tree.PruneList)`. 0 indicates the full, unpruned tree and `max(tree.PruneList)` indicates the completely pruned tree (i.e., just the root node).

If you specify 'all', then `RegressionTree.resubPredict` operates on all subtrees (i.e., the entire pruning sequence). This specification is equivalent to using `0:max(tree.PruneList)`.

`RegressionTree.resubPredict` prunes tree to each level indicated in `Subtrees`, and then estimates the corresponding output arguments. The size of `Subtrees` determines the size of some output arguments.

To invoke `Subtrees`, the properties `PruneList` and `PruneAlpha` of tree must be nonempty. In other words, grow tree by setting 'Prune', 'on', or by pruning tree using `prune`.

Example: 'Subtrees', 'all'

## Output Arguments

**yfit**

The response tree predicts for the training data.

If the `Subtrees` name-value argument is a scalar or is missing, label is the same data type as the training response data `tree.Y`.

If `Subtrees` contains  $m > 1$  entries, label has  $m$  columns, each of which represents the predictions of the corresponding subtree.

**node**

The tree node numbers where tree sends each data row.

If the `Subtrees` name-value argument is a scalar or is missing, `node` is a numeric column vector with `n` rows, the same number of rows as `tree.X`.

If `Subtrees` contains `m>1` entries, `node` is a `n-by-m` matrix. Each column represents the node predictions of the corresponding subtree.

## Examples

### Compute the In-Sample MSE

Load the `carsmall` data set. Consider `Displacement`, `Horsepower`, and `Weight` as predictors of the response `MPG`.

```
load carsmall
X = [Displacement Horsepower Weight];
```

Grow a regression tree using all observations.

```
Mdl = fitrtree(X,MPG);
```

Compute the resubstitution MSE.

```
Yfit = resubPredict(Mdl);
mean((Yfit - Mdl.Y).^2)
```

```
ans =
```

```
4.8952
```

You can get the same result using `RegressionTree.resubLoss`.

```
resubLoss(Mdl)
```

```
ans =
```

```
4.8952
```

## Estimate In-Sample Responses For Each Subtree

Load the `carsmall` data set. Consider `Weight` as a predictor of the response `MPG`.

```
load carsmall
idxNaN = isnan(MPG + Weight);
X = Weight(~idxNaN);
Y = MPG(~idxNaN);
n = numel(X);
```

Grow a regression tree using all observations.

```
Mdl = fitrtree(X,Y);
```

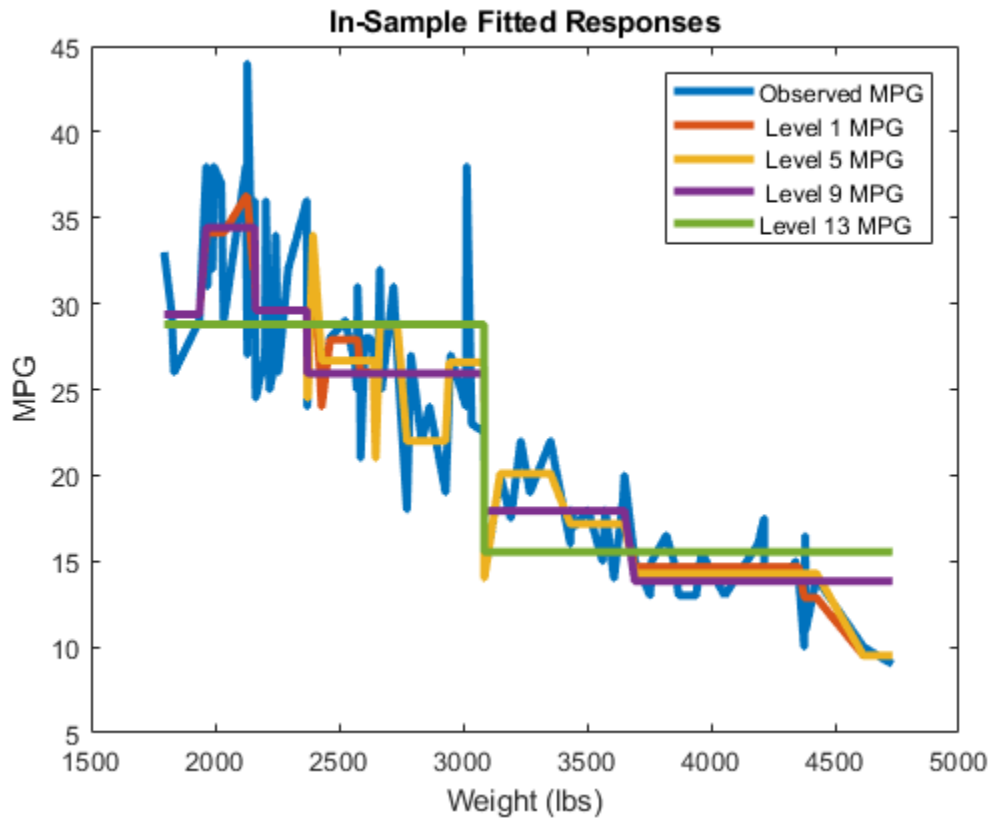
Compute resubstitution fitted values for the subtrees at several pruning levels.

```
m = max(Mdl.PruneList);
pruneLevels = 1:4:m; % Pruning levels to consider
z = numel(pruneLevels);
Yfit = resubPredict(Mdl, 'SubTrees',pruneLevels);
```

`Yfit` is an `n`-by- `z` matrix of fitted values in which the rows correspond to observations and the columns correspond to a subtree.

Plot several columns of `Yfit` and `Y` against `X`.

```
figure;
sortDat = sortrows([X Y Yfit],1); % Sort all data with respect to X
plot(repmat(sortDat(:,1),1,size(Yfit,2) + 1),sortDat(:,2:end))...
    % Vectorize for efficiency
lev = cellstr(num2str((pruneLevels)', 'Level %d MPG'));
legend(['Observed MPG'; lev])
title 'In-Sample Fitted Responses'
xlabel 'Weight (lbs)';
ylabel 'MPG';
h = findobj(gcf);
set(h(4:end), 'LineWidth', 3) % Widen all lines
```



The values of  $\hat{y}_{fit}$  for lower pruning levels tend to follow the data more closely than higher levels. Higher pruning levels tend to be flat for large  $x$  intervals.

## See Also

`fitrtree` | `predict` | `resubLoss`

# resume

Resume a Bayesian optimization

## Syntax

```
newresults = resume(results, Name, Value)
```

## Description

`newresults = resume(results, Name, Value)` resumes the optimization that produced `results` with additional options specified by one or more `Name, Value` pair arguments.

## Examples

### Resume a Bayesian Optimization

This example shows how to resume a Bayesian optimization. The optimization is for a deterministic function known as Rosenbrock's function, which is a well-known test case for nonlinear optimization. The function has a global minimum value of 0 at the point `[1, 1]`.

Create two real variables bounded by -5 and 5.

```
x1 = optimizableVariable('x1', [-5, 5]);  
x2 = optimizableVariable('x2', [-5, 5]);  
vars = [x1, x2];
```

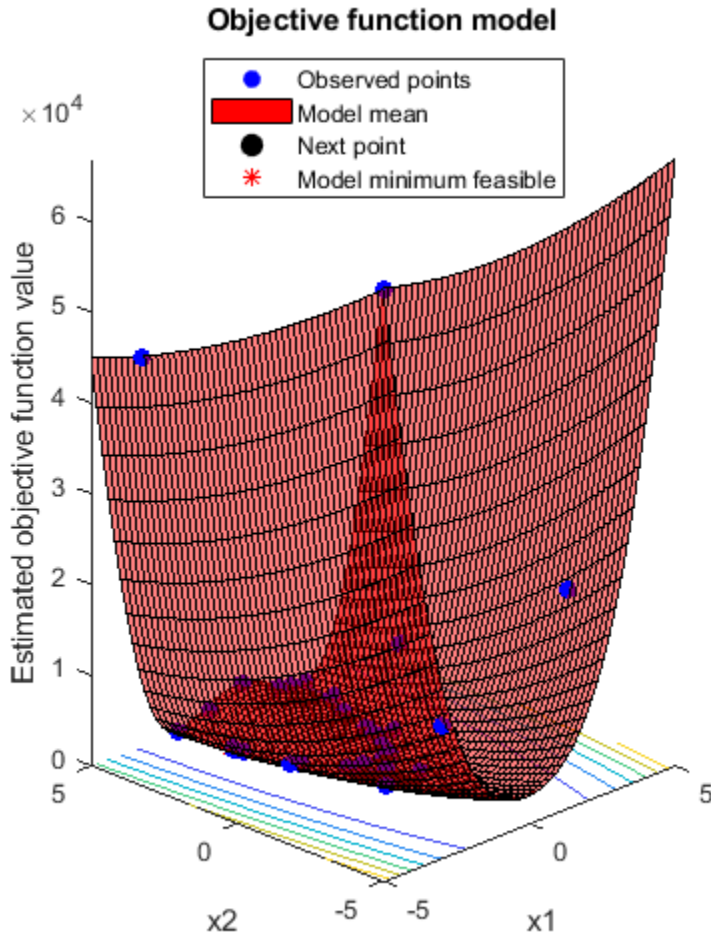
Create the objective function.

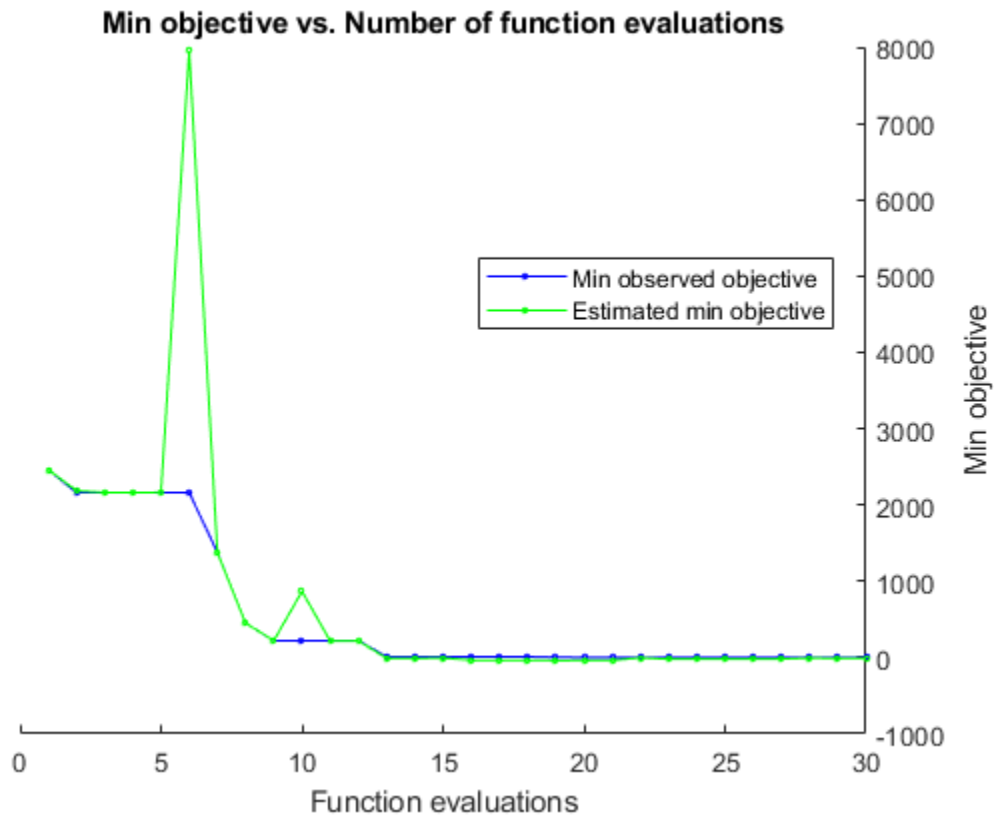
```
function f = rosenbrocks(x)  
  
f = 100*(x.x2 - x.x1^2)^2 + (1 - x.x1)^2;
```

```
fun = @rosenbrocks;
```

For reproducibility, set the random seed, and set the acquisition function to 'expected-improvement-plus' in the optimization.

```
rng default
results = bayesopt(fun, vars, 'Verbose', 0, ...
    'AcquisitionFunctionName', 'expected-improvement-plus');
```





View the best point found and the best modeled objective.

```
results.XAtMinObjective
results.MinEstimatedObjective
```

```
ans =
```

```
1x2 table
```

x1	x2
1.2421	1.5299

```
ans =  
  
-9.5402
```

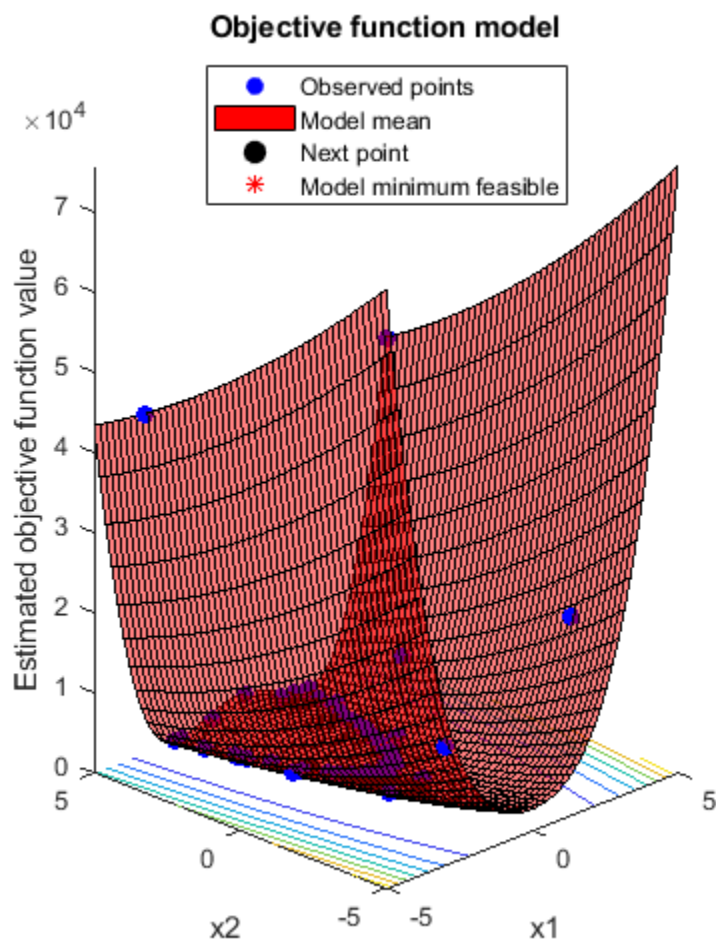
The best point is somewhat close to the optimum, but the function model is inaccurate. Resume the optimization for 30 more points (a total of 60 points), this time telling the optimizer that the objective function is deterministic.

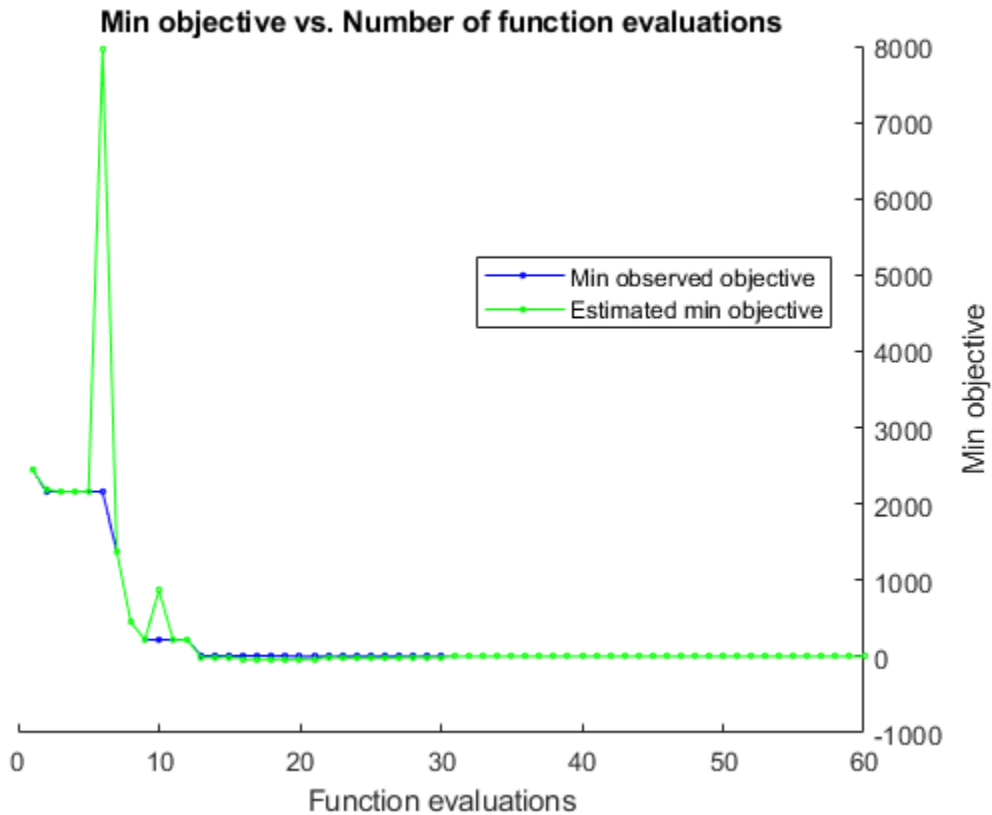
```
newresults = resume(results, 'IsObjectiveDeterministic', true, 'MaxObjectiveEvaluations', 30,  
newresults.XAtMinObjective  
newresults.MinEstimatedObjective
```

```
ans =  
  
1x2 table  
  
    x1    x2  
-----  
1.0511  1.1095
```

```
ans =  
  
-0.0226
```







The objective function model is much closer to the true function this time. The best point is closer to the true optimum.

## Input Arguments

### **results** — Bayesian optimization results

`BayesianOptimization` object

Bayesian optimization results, specified as a `BayesianOptimization` object.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

You can use any name-value pair accepted by `bayesopt` except for those beginning with `Initial`. See the `bayesopt` “Input Arguments” on page 32-159.

---

**Note** The `MaxObjectiveEvaluations` and `MaxTime` name-value pairs mean *additional* time or evaluations, above the numbers stored in `results`. So, for example, the default number of evaluations is 30 in addition to the original specification.

---

Additionally, you can use the following name-value pair.

Example: `resume(results, 'MaxObjectiveEvaluations', 60)`

### VariableDescriptions — Modify variable

`OptimizableVariable` object

Modify variable, specified as an `OptimizableVariable` object.

You can change only the following properties of a variable in an optimization.

- Range of real or integer variables. For example,

```
xvar = optimizableVariable('x', [-10, 10]);
% Modify the range:
xvar.Range = [1, 5];
```

- Type between 'integer' and 'real'. For example,

```
xvar.Type = 'integer';
```

- Transform of real or integer variables between 'log' and 'none'. For example,

```
xvar.Transform = 'log';
```

## Output Arguments

**newresults** — Optimization results

`BayesianOptimization` object

Optimization results, returned as a `BayesianOptimization` object.

## See Also

`BayesianOptimization` | `bayesopt`

**Introduced in R2016b**

## resume

**Class:** ClassificationEnsemble

Resume training ensemble

## Syntax

```
ens1 = resume(ens, nlearn)
ens1 = resume(ens, nlearn, Name, Value)
```

## Description

`ens1 = resume(ens, nlearn)` trains `ens` for `nlearn` more cycles. `resume` uses the same training options `fitcensemble` used to create `ens`.

---

**Note** You cannot resume training when `ens` is a Subspace ensemble created with 'AllPredictorCombinations' number of learners.

---

`ens1 = resume(ens, nlearn, Name, Value)` trains `ens` with additional options specified by one or more `Name, Value` pair arguments.

## Input Arguments

**ens**

A classification ensemble, created with `fitcensemble`.

**nlearn**

A positive integer, the number of cycles for additional training of `ens`.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### **nprint**

Printout frequency, a positive integer scalar or 'off' (no printouts). Returns to the command line the number of weak learners trained so far. Useful when you train ensembles with many learners on large data sets.

**Default:** 'off'

## Output Arguments

### **ens1**

The classification ensemble `ens`, augmented with additional training.

## Examples

### Train Classification Ensemble for Additional Cycles

Train a classification ensemble for three cycles, and compare the resubstitution error obtained after training the ensemble for more cycles.

Load the `ionosphere` data set.

```
load ionosphere
```

Train a classification ensemble for three cycles and examine the resubstitution error.

```
ens = fitcensemble(X, Y, 'Method', 'GentleBoost', 'NumLearningCycles', 3);  
L = resubLoss(ens)
```

```
L = 0.0114
```

Train for three more cycles and examine the new resubstitution error.

```
ens1 = resume(ens,3);  
L = resubLoss(ens1)
```

```
L = 0
```

The resubstitution error is much lower in the new ensemble than the original.

## See Also

`fitcensemble`

## resume

**Class:** ClassificationPartitionedEnsemble

Resume training learners on cross-validation folds

## Syntax

```
ens1 = resume(ens, nlearn)
ens1 = resume(ens, nlearn, Name, Value)
```

## Description

`ens1 = resume(ens, nlearn)` trains `ens` in every fold for `nlearn` more cycles. `resume` uses the same training options `fitcensemble` used to create `ens`.

`ens1 = resume(ens, nlearn, Name, Value)` trains `ens` with additional options specified by one or more `Name, Value` pair arguments.

## Input Arguments

### **ens**

A cross-validated classification ensemble. `ens` is the result of either:

- The `fitcensemble` function with a cross-validation name-value pair. The names are 'crossval', 'kfold', 'holdout', 'leaveout', or 'cvpartition'.
- The `crossval` method applied to a classification ensemble.

### **nlearn**

A positive integer, the number of cycles for additional training of `ens`.



## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### **nprint**

Printout frequency, a positive integer scalar or 'off' (no printouts). Returns to the command line the number of weak learners trained so far. Useful when you train ensembles with many learners on large data sets.

**Default:** 'off'

## Output Arguments

### **ens1**

The cross-validated classification ensemble `ens`, augmented with additional training.

## Examples

### Train Partitioned Classification Ensemble for More Cycles

Train a partitioned classification ensemble for 10 cycles, and compare the classification loss obtained after training the ensemble for more cycles.

Load the `ionosphere` data set.

```
load ionosphere
```

Train a partitioned classification ensemble for 10 cycles and examine the error.

```
t = templateTree('MaxNumSplits',1); % Weak learner template tree object
cvens = fitensemble(X,Y,'Method','GentleBoost','NumLearningCycles',10,'Learners',t,'cross-validation',10,'twister') % For reproducibility
L = kfoldLoss(cvens)
```

```
L = 0.0940
```

Train for 10 more cycles and examine the new error.

```
cvens = resume(cvens,10);  
L = kfoldLoss(cvens)
```

```
L = 0.0712
```

The cross-validation error is lower in the ensemble after training for 10 more cycles.

## See Also

[kfoldEdge](#) | [kfoldLoss](#) | [kfoldMargin](#) | [kfoldPredict](#)

## resume

**Class:** ClassificationSVM

Resume training support vector machine classifier

## Syntax

```
UpdatedSVMModel = resume (SVMModel, numIter)
UpdatedSVMModel = resume (SVMModel, numIter, Name, Value)
```

## Description

`UpdatedSVMModel = resume (SVMModel, numIter)` returns an updated support vector machine (SVM) classifier (`UpdatedSVMModel`) by training the support vector machine classifier `SVMModel` for `numIter` more iterations.

`resume` continues applying the training options that you set for `fitcsvm` to train `SVMModel`.

`UpdatedSVMModel = resume (SVMModel, numIter, Name, Value)` returns an updated support vector machine classifier (`UpdatedSVMModel`) with additional options specified by one or more `Name, Value` pair arguments.

## Input Arguments

**SVMModel** — Full, trained SVM classifier

ClassificationSVM classifier

Full, trained SVM classifier, specified as a `ClassificationSVM` model trained using `fitcsvm`.

**numIter** — Number of iterations

Positive integer

Number of iterations to continue training the SVM classifier, specified as a positive integer.

Data Types: `double`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### **Verbose** — Verbosity level

0 | 1 | 2

Verbosity level, specified as the comma-separated pair consisting of 'Verbose' and either 0, 1, or 2. `Verbose` controls the amount of optimization information that the software displays to the Command Window and is saved as a structure to `SVMModel.ConvergenceInfo.History`.

This table summarizes the available verbosity level options.

Value	Description
0	The software does not display or save convergence information.
1	The software displays diagnostic messages and saves convergence criteria every <i>numprint</i> iterations, where <i>numprint</i> is the value of the name-value pair argument 'NumPrint'.
2	The software displays diagnostic messages and saves convergence criteria at every iteration.

By default, `Verbose` is the value that `fitcsvm` used to train `SVMModel`.

Example: 'Verbose', 1

Data Types: `double`

**NumPrint** — Number of iterations between diagnostic message printouts

nonnegative integer

Number of iterations between diagnostic message printouts, specified as the comma-separated pair consisting of 'NumPrint' and a nonnegative integer.

If you set 'Verbose', 1 and 'NumPrint', *numprint*, then the software displays all optimization diagnostic messages from SMO [1] and ISDA [2] every *numprint* iterations to the Command Window.

By default, NumPrint is the value that `fitcsvm` used to train `SVModel`.

Example: 'NumPrint', 500

Data Types: double

## Output Arguments

**UpdatedSVModel** — Updated SVM classifier

ClassificationSVM classifier

Updated SVM classifier, returned as a `ClassificationSVM` classifier.

## Examples

**Resume Training an SVM Classifier**

If you trained an SVM classifier, and the solver failed to converge onto a solution, then you can resume training the classifier without having to restart the entire learning process.

Load the ionosphere data set.

```
load ionosphere
rng(1); % For reproducibility
```

Train an SVM classifier. For illustration, specify that the optimization routine uses at most 50 iterations.

```
SVMModel = fitcsvm(X,Y,'IterationLimit',50);  
DidConverge = SVMModel.ConvergenceInfo.Converged  
Reason = SVMModel.ConvergenceInfo.ReasonForConvergence
```

```
DidConverge =
```

```
    logical
```

```
        0
```

```
Reason =
```

```
    'NoConvergence'
```

DidConverge = 0 indicates that the optimization routine did not converge onto a solution. Reason states the reason why the routine did not converge. Therefore, SVMModel is a partially trained, SVM classifier.

**Resume training the SVM classifier for another 1500 iterations.**

```
UpdatedSVMModel = resume(SVMModel,1500);  
DidConverge = UpdatedSVMModel.ConvergenceInfo.Converged  
Reason = UpdatedSVMModel.ConvergenceInfo.ReasonForConvergence
```

```
DidConverge =
```

```
    logical
```

```
        1
```

```
Reason =
```

```
    'DeltaGradient'
```

DidConverge indicates that the optimization routine converged onto a solution. Reason indicates that the gradient difference (DeltaGradient) reached its tolerance level (DeltaGradientTolerance). Therefore, SVMModel is a fully trained SVM classifier.

## Monitor Training of an SVM Classifier

Load the ionosphere data set.

```
load ionosphere
```

Train an SVM classifier. For illustration, specify that the optimization routine uses at most 100 iterations. Monitor the algorithm specifying that the software prints diagnostic information every 50 iterations.

```
SVMMModel = fitcsvm(X,Y,'IterationLimit',100,'Verbose',1,'NumPrint',50);
```

```

=====
| Iteration | Set | Set Size | Feasibility | Delta | KKT |
|           |    |         | Gap         | Gradient | Violation |
=====
|           0 | active | 351 | 9.971591e-01 | 2.000000e+00 | 1.000000e+00 |
|           50 | active | 351 | 8.064425e-01 | 3.736929e+00 | 2.161317e+00 |
=====

```

SVM optimization did not converge to the required tolerance.

The software prints an iterative display to the Command Window. The printout indicates that the optimization routine has not converged onto a solution.

Estimate the resubstitution loss of the partially trained SVM classifier.

```
partialLoss = resubLoss(SVMMModel)
```

```
partialLoss =
```

```
0.1054
```

The training sample misclassification error is approximately 11%.

Resume training the classifier for another 1500 iterations. Specify that the software print diagnostic information every 250 iterations.

```
UpdatedSVMMModel = resume(SVMMModel,1500,'NumPrint',250)
```

```

=====
| Iteration | Set | Set Size | Feasibility | Delta | KKT |
|           |    |         | Gap         | Gradient | Violation |
=====

```

```
|           250 |active|           351 | 1.441556e-01 | 1.701201e+00 | 1.015454e+00 |
|           500 |active|           351 | 3.277736e-03 | 9.155364e-02 | 4.830095e-02 |
|           750 |active|           351 | 3.928360e-04 | 1.367091e-02 | 9.155316e-03 |
|          1000 |active|           351 | 4.802547e-05 | 1.551900e-03 | 7.765843e-04 |
|          1044 |active|           351 | 3.602828e-05 | 9.382457e-04 | 5.182592e-04 |
```

```
Exiting Active Set upon convergence due to DeltaGradient.
```

```
UpdatedSVMModel =
```

```
ClassificationSVM
  ResponseName: 'Y'
  CategoricalPredictors: []
  ClassNames: {'b' 'g'}
  ScoreTransform: 'none'
  NumObservations: 351
  Alpha: [103x1 double]
  Bias: -3.8828
  KernelParameters: [1x1 struct]
  BoxConstraints: [351x1 double]
  ConvergenceInfo: [1x1 struct]
  IsSupportVector: [351x1 logical]
  Solver: 'SMO'
```

The software resumes at iteration 1000, and uses the same verbosity level as you set when you trained the model using `fitcsvm`. The printout indicates that the algorithm converged. Therefore, `UpdatedSVMModel` is a fully trained `ClassificationSVM` classifier.

```
updatedLoss = resubLoss(UpdatedSVMModel)
```

```
updatedLoss =
```

```
0.0769
```



The training sample misclassification error of the fully trained classifier is approximately 8%.

## Tips

If optimization has not converged and the solver is 'SMO' or 'ISDA', then try to resume training the SVM classifier.

## References

- [1] Fan, R.-E., P.-H. Chen, and C.-J. Lin. “Working set selection using second order information for training support vector machines.” *Journal of Machine Learning Research*, Vol 6, 2005, pp. 1889–1918.
- [2] Kecman V., T. -M. Huang, and M. Vogt. “Iterative Single Data Algorithm for Training Kernel Machines from Huge Data Sets: Theory and Performance.” In *Support Vector Machines: Theory and Applications*. Edited by Lipo Wang, 255–274. Berlin: Springer-Verlag, 2005.

## See Also

`ClassificationSVM` | `fitcsvm`

## resume

**Class:** RegressionEnsemble

Resume training ensemble

## Syntax

```
ens1 = resume(ens, nlearn)
ens1 = resume(ens, nlearn, Name, Value)
```

## Description

`ens1 = resume(ens, nlearn)` trains `ens` for `nlearn` more cycles. `resume` uses the same training options `fitrensemble` used to create `ens`.

`ens1 = resume(ens, nlearn, Name, Value)` trains `ens` with additional options specified by one or more `Name, Value` pair arguments.

## Input Arguments

### **ens**

A regression ensemble, created with `fitrensemble`.

### **nlearn**

A positive integer, the number of cycles for additional training of `ens`.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

**nprint**

Printout frequency, a positive integer scalar or 'off' (no printouts). Returns to the command line the number of weak learners trained so far. Useful when you train ensembles with many learners on large data sets.

**Default:** 'off'

## Output Arguments

**ens1**

The regression ensemble `ens`, augmented with additional training.

## Examples

### Train Regression Ensemble for Additional Cycles

Train a regression ensemble for 50 cycles, and compare the resubstitution error obtained after training the ensemble for more cycles.

Load the `carsmall` data set and select displacement, horsepower, and vehicle weight as predictors.

```
load carsmall
X = [Displacement Horsepower Weight];
```

Train a regression ensemble for 50 cycles and examine the resubstitution error.

```
ens = fitensemble(X,MPG,'NumLearningCycles',50);
L = resubLoss(ens)
```

```
L = 0.5563
```

Train for 50 more cycles and examine the new resubstitution error.

```
ens = resume(ens,50);
L = resubLoss(ens)
```

```
L = 0.3463
```

The resubstitution error is lower in the new ensemble than in the original.

## See Also

`fitensemble`

## resume

**Class:** RegressionPartitionedEnsemble

Resume training ensemble

## Syntax

```
ens1 = resume(ens, nlearn)
ens1 = resume(ens, nlearn, Name, Value)
```

## Description

`ens1 = resume(ens, nlearn)` trains `ens` in every fold for `nlearn` more cycles. `resume` uses the same training options `fitensemble` used to create `ens`.

`ens1 = resume(ens, nlearn, Name, Value)` trains `ens` with additional options specified by one or more `Name, Value` pair arguments.

## Input Arguments

### **ens**

A cross-validated regression ensemble. `ens` is the result of either:

- The `fitensemble` function with a cross-validation name-value pair. The names are 'crossval', 'kfold', 'holdout', 'leaveout', or 'cvpartition'.
- The `crossval` method applied to a regression ensemble.

### **nlearn**

A positive integer, the number of cycles for additional training of `ens`.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### **nprint**

Printout frequency, a positive integer scalar or 'off' (no printouts). Returns to the command line the number of weak learners trained so far. Useful when you train ensembles with many learners on large data sets.

**Default:** 'off'

## Output Arguments

### **ens1**

The cross-validated regression ensemble `ens`, augmented with additional training.

## Examples

### **Cross-Validate Regression Ensemble Augmented with Additional Training**

Examine the cross-validation error after training a regression ensemble for more cycles.

Load the `carsmall` data set and select displacement, horsepower, and vehicle weight as predictors.

```
load carsmall
X = [Displacement Horsepower Weight];
```

Train a regression ensemble for 50 cycles.

```
ens = fitensemble(X,MPG, 'NumLearningCycles', 50);
```

Cross-validate the ensemble and examine the cross-validation error.

```
rng(10, 'twister') % For reproducibility
cvens = crossval(ens);
L = kfoldLoss(cvens)

L = 27.9435
```

Train for 50 more cycles and examine the new cross-validation error.

```
cvens = resume(cvens, 50);
L = kfoldLoss(cvens)

L = 28.7077
```

The additional training did not improve the cross-validation error.

## See Also

`fitrensemble` | `kfoldLoss`

## resume

**Class:** RegressionSVM

Resume training support vector machine regression model

## Syntax

```
updatedMdl = resume (mdl, numIter)
updatedMdl = resume (mdl, numIter, Name, Value)
```

## Description

`updatedMdl = resume (mdl, numIter)` returns an updated support vector machine (SVM) regression model, `updatedMdl`, by training the model for an additional number of iterations as specified by `numIter`.

`resume` applies the same training options to `updatedMdl` that you set when using `fitrsvm` to train `mdl`.

`updatedMdl = resume (mdl, numIter, Name, Value)` returns an updated SVM regression model with additional options specified by one or more `Name, Value` pair arguments.

## Input Arguments

**mdl** — Full, trained SVM regression model

RegressionSVM model

Full, trained SVM regression model, specified as a RegressionSVM model trained using `fitrsvm`.

**numIter** — Number of iterations

positive integer value



Number of iterations to continue training the SVM regression model, specified as a positive integer value.

Data Types: `single` | `double`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### **Verbose** — Verbosity level

0 | 1 | 2

Verbosity level, specified as the comma-separated pair consisting of `'Verbose'` and either 0, 1, or 2. `Verbose` controls the amount of optimization information that the software displays to the Command Window and is saved in the model as `mdl.ModelParameters.VerbosityLevel`.

By default, `Verbose` is the value that `fitrsvm` used to train `mdl`.

Example: `'Verbose', 1`

Data Types: `single` | `double`

### **NumPrint** — Number of iterations between diagnostic message printouts

nonnegative integer value

Number of iterations between diagnostic message printouts, specified as the comma-separated pair consisting of `'NumPrint'` and a nonnegative integer.

If you set `'Verbose', 1` and `'NumPrint', numprint`, then the software displays optimization diagnostic messages to the Command Window every `numprint` number of iterations .

By default, `NumPrint` is the value that `fitrsvm` used to train `mdl`.

Example: `'NumPrint', 500`

Data Types: `single` | `double`

## Output Arguments

### **updatedMdl** — Updated SVM regression model

RegressionSVM model

Updated SVM regression model, returned as a `RegressionSVM` model.

## Examples

### Resume Training an SVM Regression Model

This example shows how to resume training an SVM regression model that failed to converge without restarting the entire learning process.

Load the `carsmall` data set.

```
load carsmall
rng default % for reproducibility
```

Specify `Acceleration`, `Cylinders`, `Displacement`, `Horsepower`, and `Weight` as the predictor variables (X) and `MPG` as the response variable (Y).

```
X = [Acceleration,Cylinders,Displacement,Horsepower,Weight];
Y = MPG;
```

Train a linear SVM regression model. For illustration purposes, set the iteration limit to 50. Standardize the data.

```
mdl = fitrsvm(X,Y,'IterationLimit',50,'Standardize',true);
```

Check to confirm whether the model converged.

```
mdl.ConvergenceInfo.Converged
```

```
ans =
```

```
0
```

The returned value of 0 indicates that the model did not converge.

Resume training the model for up to an additional 100 iterations.

```
updatedMdl = resume(mdl,100);
```

Check to confirm whether the updated model converged.

```
updatedMdl.ConvergenceInfo.Converged  
ans =  
    1
```

The returned value of 1 indicates that the updated model did converge.

Check the reason for convergence and the total number of iterations required.

```
updatedMdl.ConvergenceInfo.ReasonForConvergence  
updatedMdl.NumIterations  
ans =  
FeasibilityGap  
  
ans =  
    97
```

The model converged because the feasibility gap reached its tolerance value after 97 iterations.

## Tips

If optimization has not converged and 'Solver' is set to 'SMO' or 'ISDA', then try to resume training the SVM regression model.

## See Also

RegressionSVM | fitrsvm

Introduced in R2015b

## rica

Feature extraction by using reconstruction ICA

### Syntax

```
Mdl = rica(X,q)
Mdl = rica(X,q,Name,Value)
```

### Description

`Mdl = rica(X,q)` returns a reconstruction independent component analysis (RICA) model object that contains the results from applying RICA to the table or matrix of predictor data  $X$  containing  $p$  variables.  $q$  is the number of features to extract from  $X$ , therefore `rica` learns a  $p$ -by- $q$  matrix of transformation weights. For undercomplete or overcomplete feature representations,  $q$  can be less than or greater than the number of predictor variables, respectively.

- To access the learned transformation weights, use `Mdl.TransformWeights`.
- To transform  $X$  to the new set of features by using the learned transformation, pass `Mdl` and  $X$  to `transform`.

`Mdl = rica(X,q,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments. For example, you can standardize the predictor data or specify the value of the penalty coefficient in the reconstruction term of the objective function.

### Examples

#### Create Reconstruction ICA Object

Create a `ReconstructionICA` object by using the `rica` function.

Load the `caltech101` image patches.

```
data = load('caltech101patches');
size(data.X)
```

```
ans =
    100000    363
```

There are 100,000 image patches, each containing 363 features.

Extract 100 features from the data.

```
rng default % For reproducibility
q = 100;
Mdl = rica(data.X,q,'IterationLimit',100)
```

```
Warning: Solver LBFGS was not able to converge to a solution.
```

```
Mdl =
ReconstructionICA
    ModelParameters: [1x1 struct]
    NumPredictors: 363
    NumLearnedFeatures: 100
        Mu: []
        Sigma: []
    FitInfo: [1x1 struct]
    TransformWeights: [363x100 double]
    InitialTransformWeights: []
    NonGaussianityIndicator: [100x1 double]
```

`rica` issues a warning because it stopped due to reaching the iteration limit, instead of reaching a step-size limit or a gradient-size limit. You can still use the learned features in the returned object by calling the `transform` function.

- “Feature Extraction Workflow” on page 15-169
- “Extract Mixed Signals” on page 15-205

## Input Arguments

### **x** — Predictor data

numeric matrix | table

Predictor data, specified as an  $n$ -by- $p$  numeric matrix or table. Rows correspond to individual observations and columns correspond to individual predictor variables. If `X` is a table, then all of its variables must be numeric vectors.

Data Types: `single` | `double` | `table`

### **q** — Number of features to extract

positive integer

Number of features to extract from the predictor data, specified as a positive integer.

`rica` stores a  $p$ -by- $q$  transform weight matrix in `Mdl.TransformWeights`. Therefore, setting very large values for `q` can result in greater memory consumption and increased computation time.

Data Types: `single` | `double`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `Mdl = rica(X,q, 'IterationLimit',200, 'Standardize',true)` runs `rica` with optimization iterations limited to 200 and standardized predictor data.

### **IterationLimit** — Maximum number of iterations

1000 (default) | positive integer

Maximum number of iterations, specified as the comma-separated pair consisting of 'IterationLimit' and a positive integer.

Example: 'IterationLimit',1e6

Data Types: `single` | `double`

**VerbosityLevel** — Verbosity level

0 (default) | nonnegative integer

Verbosity level for monitoring algorithm convergence, specified as the comma-separated pair consisting of 'VerbosityLevel' and a value in this table.

Value	Description
0	rica does not display convergence information at the command line.
Positive integer	rica displays convergence information at the command line.

**Convergence Information**

Heading	Meaning
FUN VALUE	Objective function value.
NORM GRAD	Norm of the gradient of the objective function.
NORM STEP	Norm of the iterative step, meaning the distance between the previous point and the current point.
CURV	OK means the weak Wolfe condition is satisfied. This condition is a combination of sufficient decrease of the objective function and a curvature condition.
GAMMA	Inner product of the step times the gradient difference, divided by the inner product of the gradient difference with itself. The gradient difference is the gradient at the current point minus the gradient at the previous point. Gives diagnostic information on the objective function curvature.
ALPHA	Step direction multiplier, which differs from 1 when the algorithm performed a line search.
ACCEPT	YES means the algorithm found an acceptable step to take.

Example: 'VerbosityLevel', 1

Data Types: single | double

**Lambda** — Regularization coefficient value

1 (default) | positive numeric scalar

Regularization coefficient value for the transform weight matrix, specified as the comma-separated pair consisting of 'Lambda' and a positive numeric scalar. If you specify 0, then there is no regularization term in the objective function.

Example: 'Lambda', 0.1

Data Types: single | double

### **Standardize** — Flag to standardize predictor data

false (default) | true

Flag to standardize the predictor data, specified as the comma-separated pair consisting of 'Standardize' and true (1) or false (0).

If Standardize is true, then:

- `rica` centers and scales each column of the predictor data (X) by the column mean and standard deviation, respectively.
- `rica` extracts new features by using the standardized predictor matrix, and stores the predictor variable means and standard deviations in properties `Mu` and `Sigma` of `Mdl`.

Example: 'Standardize', true

Data Types: logical

### **ContrastFcn** — Contrast function

'logcosh' (default) | 'exp' | 'sqrt'

Contrast function, specified as 'logcosh', 'exp', or 'sqrt'. The contrast function is a smooth function that is similar to an absolute value function. The `rica` objective function contains a term

$$\sum_{j=1}^q \frac{1}{n} \sum_{i=1}^n g(w_j^T \tilde{x}_i),$$

where  $g$  represents the contrast function, the  $w_j$  are the variables over which the optimization takes place, and the  $\tilde{x}_i$  are data.

The three available contrast functions are:



- 'logcosh' —  $g = \frac{1}{2} \log(\cosh(2x))$
- 'exp' —  $g = -\exp\left(-\frac{x^2}{2}\right)$
- 'sqrt' —  $g = \sqrt{x^2 + 10^{-8}}$

Example: 'ContrastFcn', 'exp'

Data Types: char

### **InitialTransformWeights** — Transformation weights that initialize optimization

randn(p,q) (default) | numeric matrix

Transformation weights that initialize optimization, specified as the comma-separated pair consisting of 'InitialTransformWeights' and a  $p$ -by- $q$  numeric matrix.  $p$  must be the number of columns or variables in X and  $q$  is the value of  $q$ .

---

**Tip** You can continue optimizing a previously returned transform weight matrix by passing it as an initial value in another call to `rica`. The output model object `Mdl` stores a learned transform weight matrix in the `TransformWeights` property.

---

Example: 'InitialTransformWeights', Mdl.TransformWeights

Data Types: single | double

### **NonGaussianityIndicator** — Non-Gaussianity of sources

ones(q,1) (default) | length-q vector of ±1

Non-Gaussianity of sources, specified as a length- $q$  vector of ±1.

- `NonGaussianityIndicator(k) = 1` means `rica` models the  $k$ th source as super-Gaussian, with a sharp peak at 0.
- `NonGaussianityIndicator(k) = -1` means `rica` models the  $k$ th source as sub-Gaussian.

Data Types: single | double

**GradientTolerance** — Relative convergence tolerance on gradient norm`1e-6` (default) | positive numeric scalar

Relative convergence tolerance on gradient norm, specified as the comma-separated pair consisting of 'GradientTolerance' and a positive numeric scalar. This gradient is the gradient of the objective function.

Example: 'GradientTolerance', 1e-4

Data Types: `single` | `double`

**StepTolerance** — Absolute convergence tolerance on step size`1e-6` (default) | positive numeric scalar

Absolute convergence tolerance on the step size, specified as the comma-separated pair consisting of 'StepTolerance' and a positive numeric scalar.

Example: 'StepTolerance', 1e-4

Data Types: `single` | `double`

## Output Arguments

**Mdl** — Learned reconstruction ICA model`ReconstructionICA` model object

Learned reconstruction ICA model, returned as a `ReconstructionICA` model object.

To access properties of `Mdl`, use dot notation. For example:

- To access the learned transform weights, use `Mdl.TransformWeights`.
- To access the structure of fitting information, use `Mdl.FitInfo`.

## Algorithms

The `rica` function creates a linear transformation of input features to output features. The transformation is based on optimizing a nonlinear objective function that roughly balances statistical independence of the output features versus the ability to reconstruct the input data using the output features.

For details, see “Reconstruction ICA Algorithm” on page 15-166.

## See Also

`ReconstructionICA` | `sparsefilt` | `transform`

## Topics

“Feature Extraction Workflow” on page 15-169

“Extract Mixed Signals” on page 15-205

“Feature Extraction” on page 15-163

**Introduced in R2017a**

## ridge

Ridge regression

### Syntax

```
b = ridge(y, X, k)
b = ridge(y, X, k, scaled)
```

### Description

`b = ridge(y, X, k)` returns a vector `b` of coefficient estimates for a multilinear ridge regression of the responses in `y` on the predictors in `X`. `X` is an  $n$ -by- $p$  matrix of  $p$  predictors at each of  $n$  observations. `y` is an  $n$ -by-1 vector of observed responses. `k` is a vector of ridge parameters. If `k` has  $m$  elements, `b` is  $p$ -by- $m$ . By default, `b` is computed after centering and scaling the predictors to have mean 0 and standard deviation 1. The model does not include a constant term, and `X` should not contain a column of 1s.

`b = ridge(y, X, k, scaled)` uses the  $\{0,1\}$ -valued flag `scaled` to determine if the coefficient estimates in `b` are restored to the scale of the original data. `ridge(y, X, k, 0)` performs this additional transformation. In this case, `b` contains  $p+1$  coefficients for each value of `k`, with the first row corresponding to a constant term in the model.

`ridge(y, X, k, 1)` is the same as `ridge(y, X, k)`. In this case, `b` contains  $p$  coefficients, without a coefficient for a constant term.

The relationship between `b0 = ridge(y, X, k, 0)` and `b1 = ridge(y, X, k, 1)` is given by

```
m = mean(X);
s = std(X, 0, 1)';
b1_scaled = b1./s;
b0 = [mean(y)-m*b1_scaled; b1_scaled]
```

This can be seen by replacing the  $x_i$  ( $i = 1, \dots, n$ ) in the multilinear model  $y = b_0^0 + b_1^0 x_1 + \dots + b_n^0 x_n$  with the  $z$ -scores  $z_i = (x_i - \mu_i)/\sigma_i$ , and replacing  $y$  with  $y - \mu_y$ .

In general, `b1` is more useful for producing plots in which the coefficients are to be displayed on the same scale, such as a *ridge trace* (a plot of the regression coefficients as a function of the ridge parameter). `b0` is more useful for making predictions.

Coefficient estimates for multiple linear regression models rely on the independence of the model terms. When terms are correlated and the columns of the design matrix  $X$  have an approximate linear dependence, the matrix  $(X^T X)^{-1}$  becomes close to singular. As a result, the least-squares estimate

$$\hat{\beta} = (X^T X)^{-1} X^T y$$

becomes highly sensitive to random errors in the observed response  $y$ , producing a large variance. This situation of multicollinearity can arise, for example, when data are collected without an experimental design.

Ridge regression addresses the problem by estimating regression coefficients using

$$\hat{\beta} = (X^T X + kI)^{-1} X^T y$$

where  $k$  is the ridge parameter and  $I$  is the identity matrix. Small positive values of  $k$  improve the conditioning of the problem and reduce the variance of the estimates. While biased, the reduced variance of ridge estimates often result in a smaller mean square error when compared to least-squares estimates.

## Examples

### Ridge Regression

Load the sample data.

```
load acetylene
```

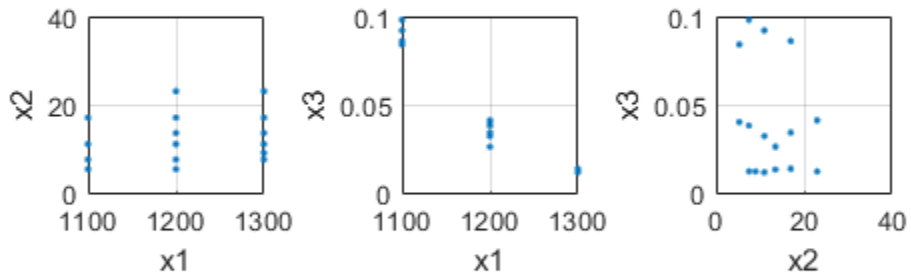
`acetylene` has observations for the predictor variables `x1`, `x2`, `x3`, and the response variable `y`.

Plot the predictor variables against each other.

```
subplot(1,3,1)
plot(x1,x2,'.')
xlabel('x1'); ylabel('x2'); grid on; axis square
```

```
subplot(1,3,2)
plot(x1,x3, '.')
xlabel('x1'); ylabel('x3'); grid on; axis square

subplot(1,3,3)
plot(x2,x3, '.')
xlabel('x2'); ylabel('x3'); grid on; axis square
```



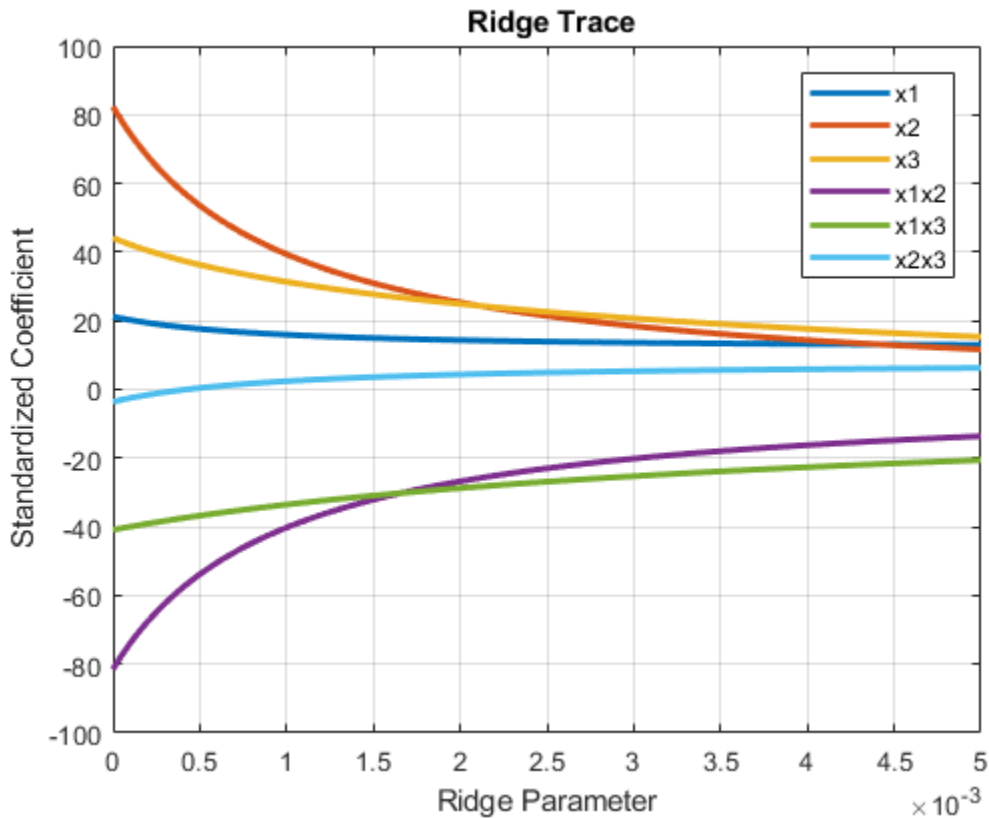
Note the correlation between  $x_1$  and the other two predictor variables.

Compute coefficient estimates for a multilinear model with interaction terms, for a range of ridge parameters using `ridge` and `x2fx`.

```
X = [x1 x2 x3];
D = x2fx(X, 'interaction');
D(:,1) = []; % No constant term
k = 0:1e-5:5e-3;
b = ridge(y,D,k);
```

Plot the ridge trace.

```
figure
plot(k,b, 'LineWidth',2)
ylim([-100 100])
grid on
xlabel('Ridge Parameter')
ylabel('Standardized Coefficient')
title('\bf Ridge Trace')
legend('x1', 'x2', 'x3', 'x1x2', 'x1x3', 'x2x3')
```



The estimates stabilize to the right of the plot. Note that the coefficient of the  $x_2x_3$  interaction term changes sign at a value of the ridge parameter  $\approx 5 \times 10^{-4}$ .

## References

- [1] Hoerl, A. E., and R. W. Kennard. "Ridge Regression: Biased Estimation for Nonorthogonal Problems." *Technometrics*. Vol. 12, No. 1, 1970, pp. 55–67.
- [2] Hoerl, A. E., and R. W. Kennard. "Ridge Regression: Applications to Nonorthogonal Problems." *Technometrics*. Vol. 12, No. 1, 1970, pp. 69–82.



- [3] Marquardt, D.W. “Generalized Inverses, Ridge Regression, Biased Linear Estimation, and Nonlinear Estimation.” *Technometrics*. Vol. 12, No. 3, 1970, pp. 591–612.
- [4] Marquardt, D. W., and R.D. Snee. “Ridge Regression in Practice.” *The American Statistician*. Vol. 29, No. 1, 1975, pp. 3–20.

## See Also

regress | stepwise

Introduced before R2006a

## risk

**Class:** `classregtree`

Node risks

---

**Note** `classregtree` will be removed in a future release. See `fitctree`, `fitrtree`, `ClassificationTree`, or `RegressionTree` instead.

---

## Syntax

```
r = risk(t)
r = risk(t,nodes)
```

## Description

`r = risk(t)` returns an  $n$ -element vector  $r$  of the risk of the nodes in the tree  $t$ , where  $n$  is the number of nodes. The risk  $r(i)$  for node  $i$  is the node error  $e(i)$  (computed by `nodeerr`) weighted by the node probability  $p(i)$  (computed by `nodeprob`).

`r = risk(t,nodes)` takes a vector `nodes` of node numbers and returns the risk values for the specified nodes.

## Examples

Create a classification tree for Fisher's iris data:

```
load fisheriris;

t = classregtree(meas,species,...
                'names',{'SL' 'SW' 'PL' 'PW'})

t =
Decision tree for classification
1  if PL<2.45 then node 2 elseif PL>=2.45 then node 3 else setosa
2  class = setosa
3  if PW<1.75 then node 4 elseif PW>=1.75 then node 5 else versicolor
4  if PL<4.95 then node 6 elseif PL>=4.95 then node 7 else versicolor
```

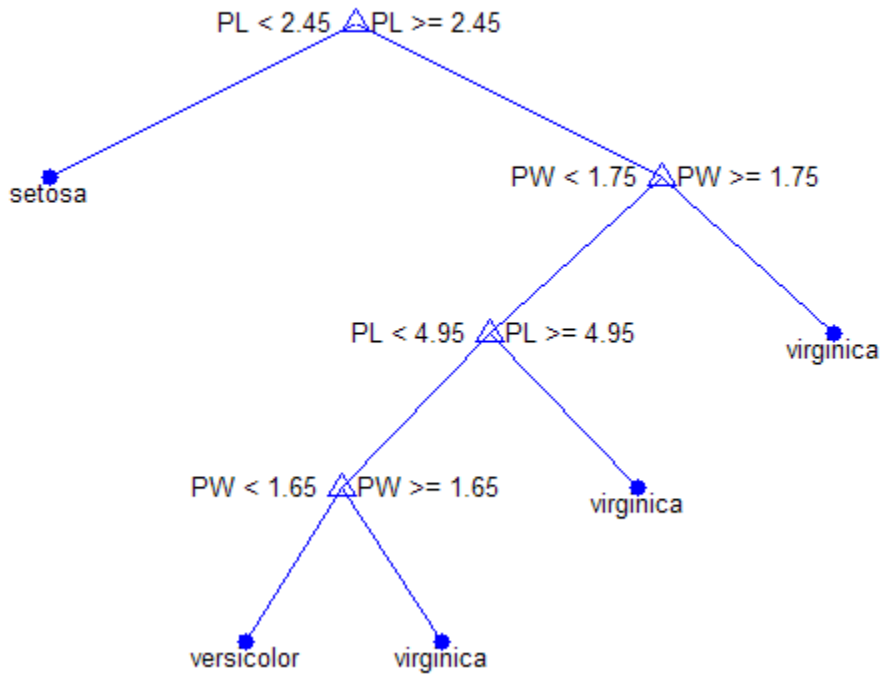
```

5 class = virginica
6 if PW<1.65 then node 8 elseif PW>=1.65 then node 9 else versicolor
7 class = virginica
8 class = versicolor
9 class = virginica

```

```
view(t)
```

Click to display:  Magnification:  Pruning level:



```

e = nodeerr(t);
p = nodeprob(t);
r = risk(t);

```

```

r
r =
    0.6667
    0

```

```
0.3333
0.0333
0.0067
0.0067
0.0133
  0
  0

e.*p
ans =
  0.6667
  0
  0.3333
  0.0333
  0.0067
  0.0067
  0.0133
  0
  0
```

## References

- [1] Breiman, L., J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Boca Raton, FL: CRC Press, 1984.

## See Also

`classregtree` | `nodeerr` | `nodeprob`

# robustcov

Robust multivariate covariance and mean estimate

## Syntax

```
sig = robustcov(x)
[ sig,mu ] = robustcov(x)
[ sig,mu,mah ] = robustcov(x)
[ sig,mu,mah,outliers ] = robustcov(x)
[ sig,mu,mah,outliers,s ] = robustcov(x)
[ ____ ] = robustcov(x,Name,Value)
```

## Description

`sig = robustcov(x)` returns the robust covariance estimate `sig` of the multivariate data contained in `x`.

`[ sig,mu ] = robustcov(x)` also returns an estimate of the robust Minimum Covariance Determinant (MCD) mean, `mu`.

`[ sig,mu,mah ] = robustcov(x)` also returns the robust distances `mah`, computed as the Mahalanobis distances of the observations using the robust estimates of the mean and covariance.

`[ sig,mu,mah,outliers ] = robustcov(x)` also returns the indices of the observations retained as outliers in the sample data, `outliers`.

`[ sig,mu,mah,outliers,s ] = robustcov(x)` also returns a structure `s` that contains information about the estimate.

`[ ____ ] = robustcov(x,Name,Value)` returns any of the arguments shown in the previous syntaxes, using additional options specified by one or more `Name,Value` pair arguments. For example, you can specify which robust estimator to use or the start method to use for the attractors.

## Examples

### Detect Outliers Using Distance-Distance Plots

Use a Gaussian copula to generate random data points from a bivariate distribution.

```
rng default
rho = [1,0.05;0.05,1];
u = copularnd('Gaussian',rho,50);
```

Modify 5 randomly selected observations to be outliers.

```
noise = randperm(50,5);
u(noise,1) = u(noise,1)*5;
```

Calculate the robust covariance matrices using the three available methods: Fast-MCD, Orthogonalized Gnanadesikan-Kettenring (OGK), and Olive-Hawkins.

```
[Sfmc, Mfmc, dfmc, Outfmc] = robustcov(u);
[Sogk, Mogk, dogk, Outogk] = robustcov(u,'Method','ogk');
[Soh, Moh, doh, Outoh] = robustcov(u,'Method','olivehawkins');
```

Calculate the classical distance values for the sample data using the Mahalanobis measure.

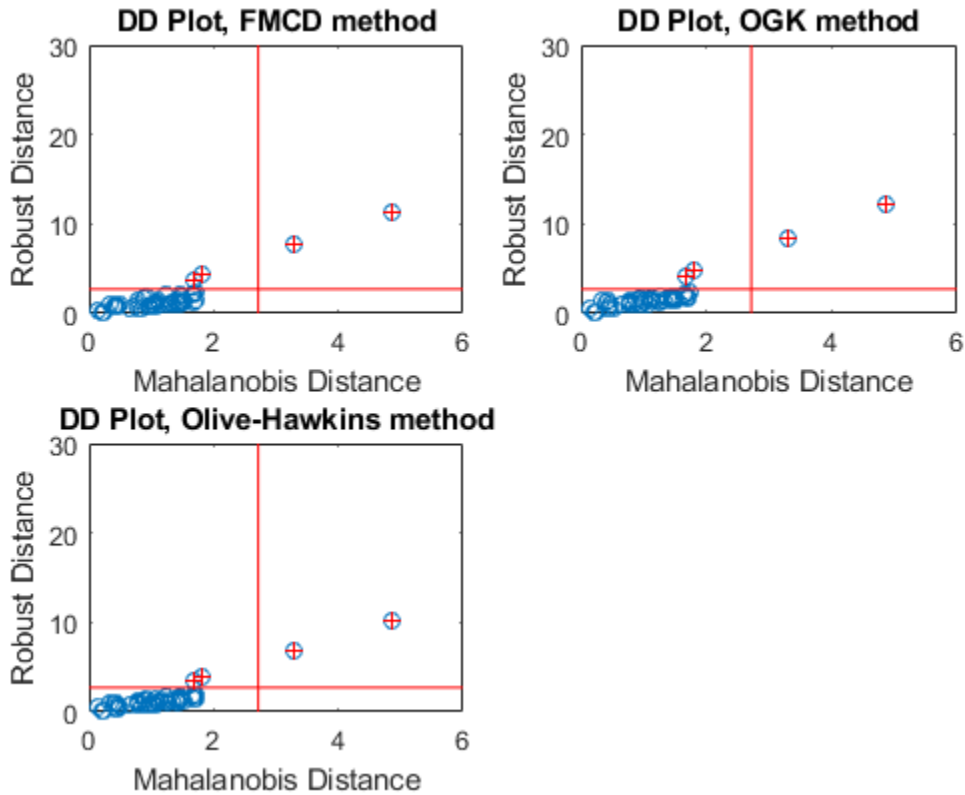
```
d_classical = pdist2(u, mean(u), 'mahal');
p = size(u,2);
chi2quantile = sqrt(chi2inv(0.975,p));
```

Create DD Plots for each robust covariance calculation method.

```
figure
subplot(2,2,1)
plot(d_classical, dfmc, 'o')
line([chi2quantile, chi2quantile], [0, 30], 'color', 'r')
line([0, 6], [chi2quantile, chi2quantile], 'color', 'r')
hold on
plot(d_classical(Outfmc), dfmc(Outfmc), 'r+')
xlabel('Mahalanobis Distance')
ylabel('Robust Distance')
title('DD Plot, FMCD method')
hold off
```

```
subplot(2,2,2)
plot(d_classical, dogk, 'o')
line([chi2quantile, chi2quantile], [0, 30], 'color', 'r')
line([0, 6], [chi2quantile, chi2quantile], 'color', 'r')
hold on
plot(d_classical(Outogk), dogk(Outogk), 'r+')
xlabel('Mahalanobis Distance')
ylabel('Robust Distance')
title('DD Plot, OGK method')
hold off

subplot(2,2,3)
plot(d_classical, doh, 'o')
line([chi2quantile, chi2quantile], [0, 30], 'color', 'r')
line([0, 6], [chi2quantile, chi2quantile], 'color', 'r')
hold on
plot(d_classical(Outoh), doh(Outoh), 'r+')
xlabel('Mahalanobis Distance')
ylabel('Robust Distance')
title('DD Plot, Olive-Hawkins method')
hold off
```



In a DD plot, the data points tend to cluster in a straight line that passes through the origin. Points that are far removed from this line are generally considered outliers. In each of the previous plots, the red '+' symbol indicates the data points that `robustcov` considers to be outliers.

### Evaluate Data for Multivariate Normal Distribution

This example shows how to use `robustcov` to evaluate sample data for multivariate normal or other elliptically-contoured (EC) distributions.



Generate random sample data from a multivariate normal distribution. Calculate the Mahalanobis distances for the robust covariance estimates (using the Olive-Hawkins method) and the classical covariance estimates.

```
rng default
x1 = mvnrnd(zeros(1,3),eye(3),200);
[~, ~, d1] = robustcov(x1, 'Method', 'olivehawkins');
d_classical1 = pdist2(x1, mean(x1), 'mahalanobis');
```

Generate random sample data from an elliptically-countoured (EC) distribution. Calculate the Mahalanobis distances for the robust covariance estimates (using the Olive-Hawkins method) and the classical covariance estimates.

```
mu1 = [0 0 0];
sig1 = eye(3);
mu2 = [0 0 0];
sig2 = 25*eye(3);
x2 = [mvnrnd(mu1, sig1, 120); mvnrnd(mu2, sig2, 80)];
[~, ~, d2] = robustcov(x2, 'Method', 'olivehawkins');
d_classical2 = pdist2(x2, mean(x2), 'mahalanobis');
```

Generate random sample data from a multivariate lognormal distribution, which is neither multivariate normal or elliptically-contoured. Calculate the Mahalanobis distances for the robust covariance estimates (using the Olive-Hawkins method) and the classical covariance estimates.

```
x3 = exp(x1);
[~, ~, d3] = robustcov(x3, 'Method', 'olivehawkins');
d_classical3 = pdist2(x3, mean(x3), 'mahalanobis');
```

Create a D-D Plot for each of the three sets of sample data to compare.

```
figure
subplot(2,2,1)
plot(d_classical1, d1, 'o')
line([0 4.5], [0, 4.5])
xlabel('Mahalanobis Distance')
ylabel('Robust Distance')
title('DD Plot, Multivariate Normal')

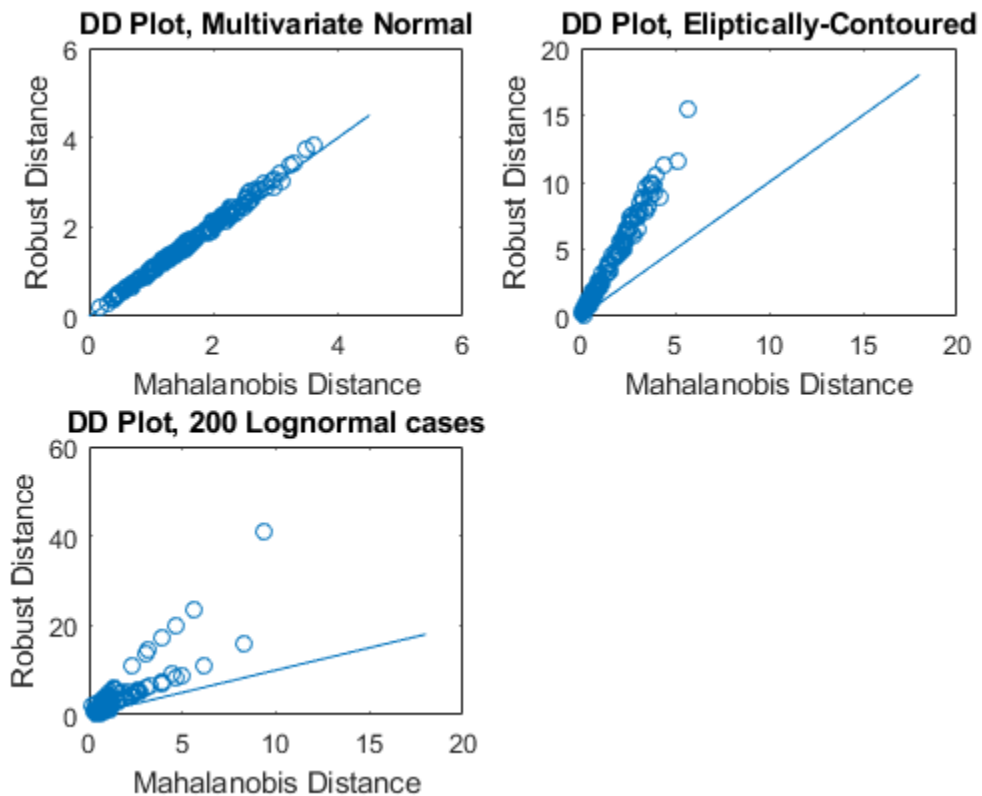
subplot(2,2,2)
plot(d_classical2, d2, 'o')
line([0 18], [0, 18])
xlabel('Mahalanobis Distance')
```

```

ylabel('Robust Distance')
title('DD Plot, Elliptically-Contoured')

subplot(2,2,3)
plot(d_classical3, d3, 'o')
line([0 18], [0, 18])
xlabel('Mahalanobis Distance')
ylabel('Robust Distance')
title('DD Plot, 200 Lognormal cases')

```



For data with a multivariate normal distribution (as shown in the upper left), the plotted points follow a straight, 45-degree line extending from the origin. For data with an elliptically-contoured distribution (as shown in the upper right), the plotted points follow

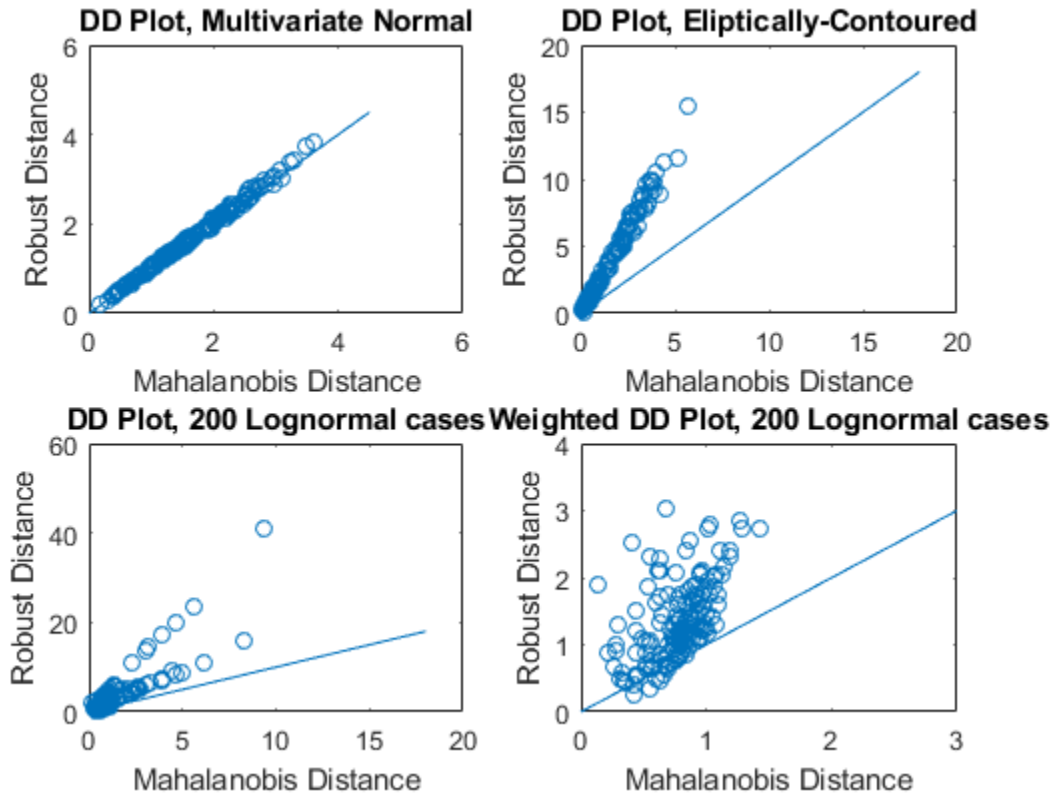
a straight line, but are not at a 45-degree angle to the origin. For the lognormal distribution (as show in in the lower left), the plotted points do not follow a straight line.

It is difficult to identify any pattern in the lognormal distribution plot because most of the points are in the lower left of the plot. Use a weighted DD plot to magnify this corner and reveal features that are obscured when large robust distances exist.

```
d3_weighted = d3(d3 < sqrt(chi2inv(0.975,3)));  
d_classical_weighted = d_classical3(d3 < sqrt(chi2inv(0.975,3)));
```

Add a fourth subplot to the figure to show the results of the weighting process on the lognormally distributed data.

```
subplot(2,2,4)  
plot(d_classical_weighted, d3_weighted, 'o')  
line([0 3], [0, 3])  
xlabel('Mahalanobis Distance')  
ylabel('Robust Distance')  
title('Weighted DD Plot, 200 Lognormal cases')
```



The scale on this plot indicates that it represents a magnified view of the original DD plot for the lognormal data. This view more clearly shows the lack of pattern to the plot, which indicates that the data is neither multivariate normal nor elliptically contoured.

### Compute Robust Covariance and Plot the Outliers

Use a Gaussian copula to generate random data points from a bivariate distribution.

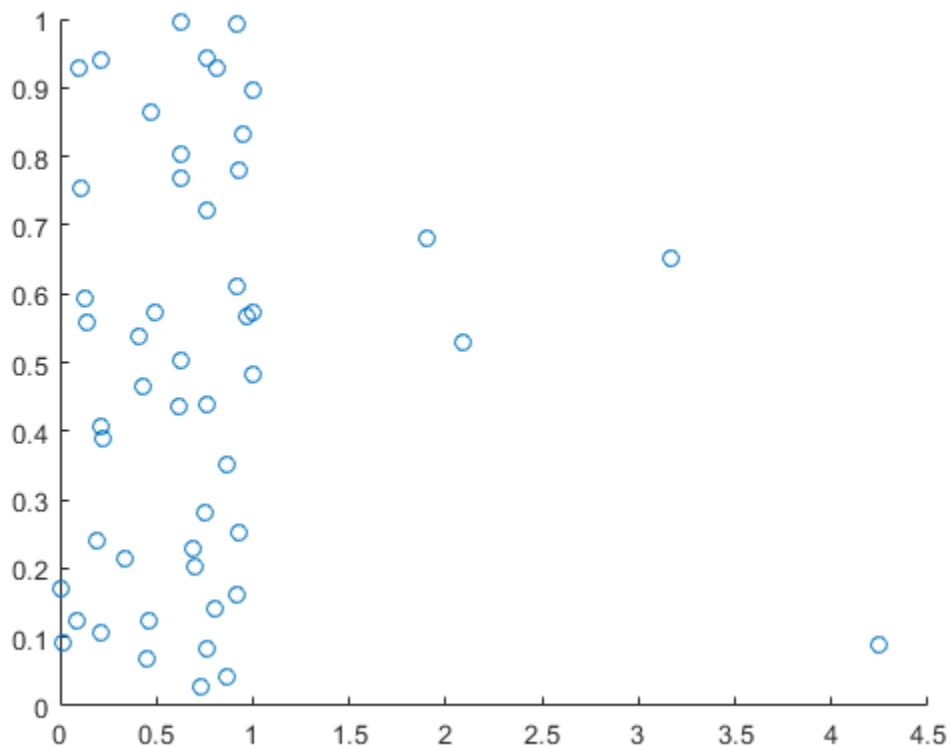
```
rng default
rho = [1,0.05;0.05,1];
u = copularnd('Gaussian',rho,50);
```

Modify 5 randomly selected observations to be outliers.

```
noise = randperm(50,5);  
u(noise,1) = u(noise,1)*5;
```

Visualize the bivariate data using a scatter plot.

```
figure  
scatter(u(:,1),u(:,2))
```



Most of the data points appear on the left side of the plot. However, some of the data points appear further to the right. These points are possible outliers that could affect the covariance matrix calculation.

Compare the classical and robust covariance matrices.

```
c = cov(u)
rc = robustcov(u)
```

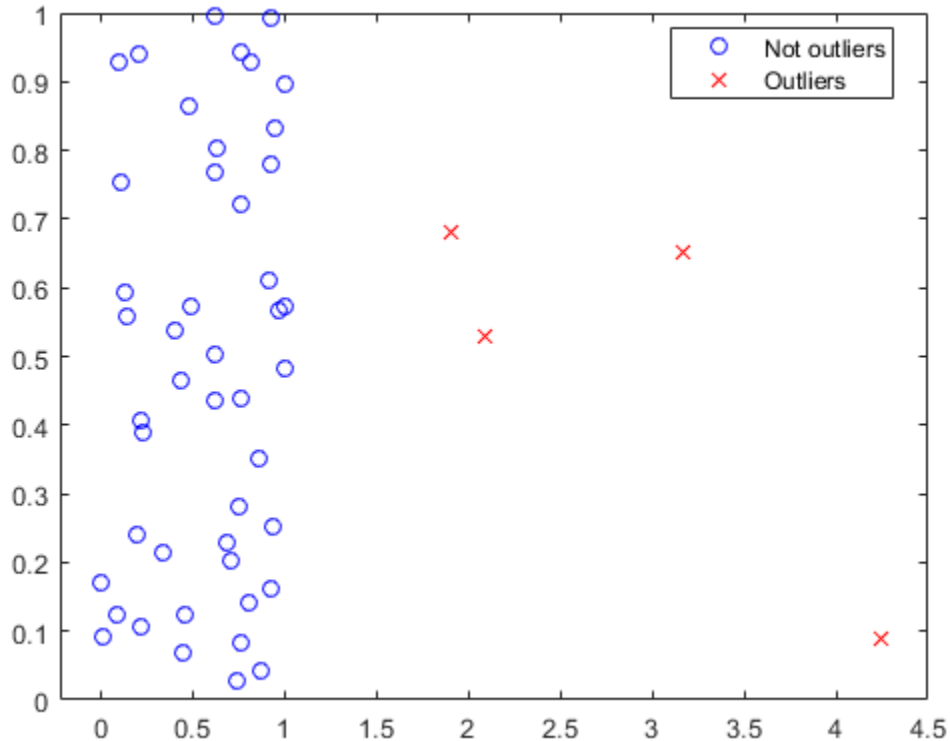
```
c =
    0.5523    0.0000
    0.0000    0.0913
```

```
rc =
    0.1117    0.0364
    0.0364    0.1695
```

The classical and robust covariance matrices differ because the outliers present in the sample data influence the results.

Identify and plot the data points that `robustcov` considers outliers.

```
[sig,mu,mah,outliers] = robustcov(u);
figure
gscatter(u(:,1),u(:,2),outliers,'br','ox')
legend({'Not outliers','Outliers'})
```



`robustcov` identifies the data points on the right side of the plot as potential outliers, and treats them accordingly when calculating the robust covariance matrix.

## Input Arguments

### **x** — Sample data

(default) | matrix of numeric values

Sample data used to estimate the robust covariance matrix, specified as a matrix of numeric values.  $x$  is an  $n$ -by- $p$  matrix where each row is an observation and each column is a variable.

`robustcov` removes any rows with missing predictor values when calculating the robust covariance matrix.

Data Types: `single` | `double`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1`, `Value1`, ..., `NameN`, `ValueN`.

Example: `'Method', 'ogk', 'NumOGKIterations', 1` specifies the robust estimator as the Orthogonalized Gnanadesikan-Kettenring method and sets the number of orthogonalization iterations 1.

### For All Estimators

#### Method — Robust estimator

`'fmcd'` (default) | `'ogk'` | `'olivehawkins'`

Robust estimator, specified as one of the following.

Name	Value
<code>'fmcd'</code>	FAST-MCD (Minimum Covariance Determinant) on page 32-5967 method
<code>'ogk'</code>	Orthogonalized Gnanadesikan-Kettenring (OGK) on page 32-5968 estimate
<code>'olivehawkins'</code>	Concentration algorithm on page 32-5968 techniques, a family of fast, consistent and highly outlier-resistant methods

Example: `'Method', 'ogk'`

### For 'FMCD' and 'OliveHawkins' Estimators Only

#### OutlierFraction — Outlier fraction

0.5 (default) | numeric value in the range [0,0.5]

Outlier fraction, specified as the comma-separated pair consisting of `'OutlierFraction'` and a numeric value in the range [0,0.5]. The value 1 –



*OutlierFraction* specifies the fraction of observations over which to minimize the covariance determinant.

The algorithm chooses a subsample of size  $h = \text{ceiling}(n + p + 1) / 2$ , where  $n$  is the number of observations and  $p$  is the number of dimensions. *OutlierFraction* is the value for which the maximum possible breakdown is achieved, and controls the size of the subsets  $h$  over which the covariance determinant is minimized. The algorithm then chooses  $h$  to approximately equal  $(1 - \text{OutlierFraction}) \times n$  observations per subset.

Example: 'OutlierFraction', 0.25

Data Types: single | double

### **NumTrials — Number of trials**

positive integer value

Number of trials, specified as the comma-separated pair consisting of 'NumTrials' and a positive integer value.

If 'Method' is 'fmcd', then *NumTrials* is the number of random subsamples of size  $(p + 1)$  drawn from the sample data as starting points in the algorithm.  $p$  is the number of dimensions in the sample data. In this case, the default value for *NumTrials* is 500.

If 'Method' is 'olivehawkins', then *NumTrials* is the number of trial fits, or attractors, to be used. In this case, the default value for *NumTrials* is 2. This option is only useful for non-deterministic starts.

Example: 'NumTrials', 300

Data Types: single | double

### **For the FMCD Method Only**

### **BiasCorrection — Flag to apply small-sample correction factor**

1 (default) | 0

Flag to apply small-sample correction factor, specified as the comma-separated pair consisting of 'BiasCorrection' and either 1 or 0. A 1 value indicates that *robustcov* corrects for bias in the covariance estimate for small samples. A 0 value indicates that *robustcov* does not apply this correction.

Example: 'BiasCorrection', 0

Data Types: logical

**For the OGK Method Only****NumOGKIterations** — Number of orthogonalization iterations

2 (default) | positive integer value

Number of orthogonalization iterations, specified as the comma-separated pair consisting of 'NumOGKIterations' and a positive integer value. Generally, this value is set to 1 or 2, and further steps are unlikely to improve the estimation.

Example: 'NumIter',1

Data Types: single | double

**UnivariateEstimator** — Function for computing univariate robust estimates

'tauscale' (default) | 'qn'

Function for computing univariate robust estimates, specified as the comma-separated pair consisting of 'UnivariateEstimator' and one of the following.

Name	Value
'tauscale'	Use the “tau-scale” estimate of Yohai and Zamar, which is a truncated standard deviation and a weighted mean.
'qn'	Use the Qn scale estimate of Croux and Rousseuw.

Example: 'UnivariateEstimator', 'qn'

Data Types: char

**For the OliveHawkins method only****ReweightingMethod** — Method for reweighting

'rfch' (default) | 'rmvn'

Method for reweighting in the efficiency step, specified as the comma-separated pair consisting of 'ReweightingMethod' and one of the following.

Name	Value
'rfch'	Uses two reweighting steps. This is a standard method of reweighting to improve efficiency.

Name	Value
'rmvn'	Reweighted multivariate normal. Uses two reweighting steps that can be useful for estimating the true covariance matrix under a variety of outlier configurations when the clean data are multivariate normal.

Example: 'ReweightingMethod', 'rmvn'

#### **NumConcentrationSteps** — Number of concentration steps

10 (default) | positive integer value

Number of concentration steps, specified as the comma-separated pair consisting of 'NumConcentrationSteps' and a positive integer value.

Example: 'NumConcentrationSteps', 8

Data Types: single | double | cell

#### **StartMethod** — Start method for each attractor

'classical' (default) | 'medianball' | 'elemental' | function handle | cell array

Start method for each attractor, specified as the comma-separated pair consisting of 'Start' and one of the following.

Name	Value
'classical'	Use the classical estimator as the start. This is the DGK attractor which, used on its own, is known as the DGK estimator.
'medianball'	Use the Median Ball as the start. The Median Ball is $(\text{med}(x), \text{eye}(p))$ . So 50% of cases furthest in Euclidean distance from the sample median are trimmed for computing the MB start. This is the MB attractor which, used on its own, is known as the MB estimator.

Name	Value
'elemental'	The attractor is generated by concentration where the start is a randomly selected elemental start: the classical estimator applied to a randomly selected “elemental set” of $p + 1$ cases. This “elemental” attractor is computationally efficient, but suffers from theoretical drawbacks, as it is inconsistent and zero breakdown.

By default, the attractor is chosen as follows: If one of the attractors is 'medianball', then any attractor whose location estimate has greater Euclidean distance from `median(X)` than half the data (in other words, is outside the median ball) is not used. Then the final attractor is chosen based on the MCD criterion.

You can also specify a function handle for a function that returns two output arguments used for computing the initial location and scatter estimates..

You can also specify a cell array containing any combination of the options given in the previous table and function handles. The number of attractors used is equal to the length of the cell array. This option allows more control over the algorithm and the ability to specify a custom number of attractors and starts.

Example: 'StartMethod', 'medianball'

Data Types: `function_handle`

## Output Arguments

### **sig** — Robust covariance matrix estimates

numeric matrix

Robust covariance matrix estimates, returned as a  $p$ -by- $p$  numeric matrix.  $p$  is the number of predictors contained in the sample data.

### **mu** — Robust mean estimates

array of numeric values

Robust mean estimates, returned as a 1-by- $p$  array of numeric values.  $p$  is the number of predictors contained in the sample data.

**mah — Robust distances**

array of numeric values

Robust distances, returned as a 1-by- $n$  array of numeric values. `robustcov` removes any rows of  $x$  that contain missing data, so the number of rows of `mah` might be smaller than the number of rows in  $x$ .

**outliers — Indices of outliers**

array of logical values

Indices of observations retained as outliers in the sample data  $x$ , returned as a 1-by- $n$  array of logical values. A 0 value indicates that the observation is not an outlier. A 1 value indicates that the observation is an outlier.

`robustcov` removes any rows of  $x$  that contain missing data, so the number of rows of `outliers` might be smaller than the number of rows in  $x$ .

**s — Structure containing estimate information**

structure

Structure containing estimate information, returned as a structure.

## Algorithms

### Minimum Covariance Determinant Estimate

*Minimum covariance determinant* (MCD) is the fastest estimator of multivariate location and scatter that is both consistent and robust. However, an exact evaluation of the MCD is impractical because it is computationally expensive to evaluate all possible subsets of the sample data. `robustcov` uses the FAST-MCD method to implement MCD [3]

The FAST-MCD method selects  $h$  observations out of  $n$  (where  $n/2 < h \leq n$ ) whose classical covariance matrix has the lowest possible determinant. The MCD mean is the mean of the  $h$  selected observations.

The MCD covariance is the covariance matrix of the  $h$  selected points, multiplied by a consistency factor to obtain consistency at the multivariate normal distribution, and by a correction factor to correct for bias at small sample sizes.

## Orthogonalized Gnanadesikan-Kettenring Estimate

*Orthogonalized Gnanadesikan-Kettenring* (OGK) estimate is a positive definite estimate of the scatter starting from the Gnanadesikan and Kettering (GK) estimator, a pairwise robust scatter matrix that may be non-positive definite [1]. The estimate uses a form of principal components called an orthogonalization iteration on the pairwise scatter matrix, replacing its eigenvalues, which could be negative, with robust variances. This procedure can be iterated for improved results, and convergence is usually obtained after 2 or 3 iterations.

## Olive Hawkins Estimate

The Olive-Hawkins estimate uses the “concentration algorithm” techniques proposed by Olive and Hawkins. This is a family of fast, consistent, and highly outlier-resistant methods. The estimate is a robust root  $n$ -consistent estimator of covariance for elliptically contoured distributions with fourth moments. This estimate is obtained by first generating trial estimates, or starts, and then using the concentration technique from each trial fit to obtain attractors.

Suppose  $(T_{0j}, C_{0j})$  is a start, then at the next iteration the classical mean and covariance estimators are computed from the approximately  $n / 2$  cases (where  $n$  is the number of observations) with the smallest Mahalanobis distances based on the estimates from the previous iteration. This iteration can be continued for a fixed number of steps  $k$ , with the estimate at the last step,  $k$ , being the attractor. The final estimate is chosen based on a given criterion.

By default, two attractors are used. The first attractor is the Devlin-Gnanadesikan-Kettering (DGK) attractor, where the start used is the classical estimator. The second attractor is the Median Ball (MB) attractor, where the start used is  $(\text{median}(x), \text{eye}(p))$ , in other words the half set of data closest to  $\text{median}(x)$  in Euclidean distance. The MB attractor is used if the location estimator of the DGK attractor is outside of the median ball, and the attractor with the smallest determinant is used otherwise. The final mean estimate is the mean estimate of the chosen attractor, and the final covariance estimate is the covariance estimate of the chosen attractor, multiplied by a scaling factor to make the estimate consistent at the normal distribution.

## References

- [1] Maronna, R. and Zamar, R.H.. “Robust estimates of location and dispersion for high dimensional datasets.” *Technometrics*, Vol. 50, 2002.

- [2] Pison, S. Van Aelst and G. Willems. “Small Sample Corrections for LTS and MCD.” *Metrika*, Vol. 55, 2002.
- [3] Rousseuw, P.J. and Van Driessen, K. “A fast algorithm for the minimum covariance determinant estimator.” *Technometrics*, Vol. 41, 1999.
- [4] Olive, D.J. “A resistant estimator of multivariate location and dispersion.” *Computational Statistics and Data Analysis*, Vol. 46, pp. 99–102, 2004.

## See Also

COV

Introduced in R2016a

## robustdemo

Interactive robust regression

### Syntax

```
robustdemo  
robustdemo(x, y)
```

### Description

`robustdemo` shows the difference between ordinary least squares and robust regression for data with a single predictor. With no input arguments, `robustdemo` displays a scatter plot of a sample of roughly linear data with one outlier. The bottom of the figure displays equations of lines fitted to the data using ordinary least squares and robust methods, together with estimates of the root mean squared errors.

Use the right mouse button to click on a point and view its least-squares leverage and robust weight.

Use the left mouse button to click-and-drag a point. The displays will update.

`robustdemo(x, y)` uses `x` and `y` data vectors you supply, in place of the sample data supplied with the function.

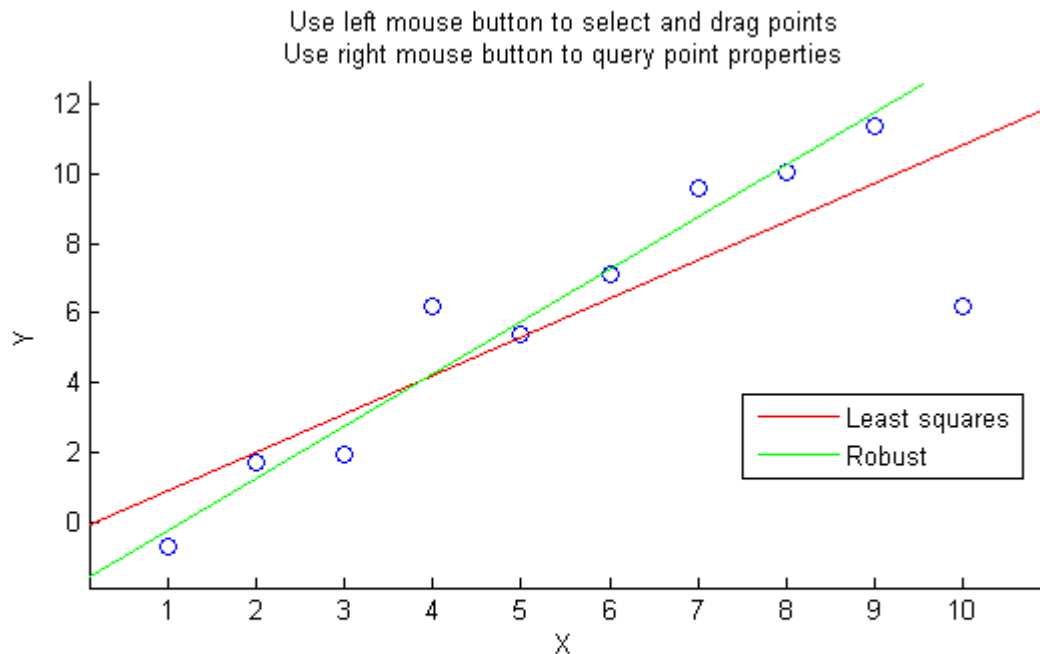
### Examples

The following steps show you how to use `robustdemo`.

- 1 **Start the example.** To begin using `robustdemo` with the built-in data, simply type the function name:

```
robustdemo
```

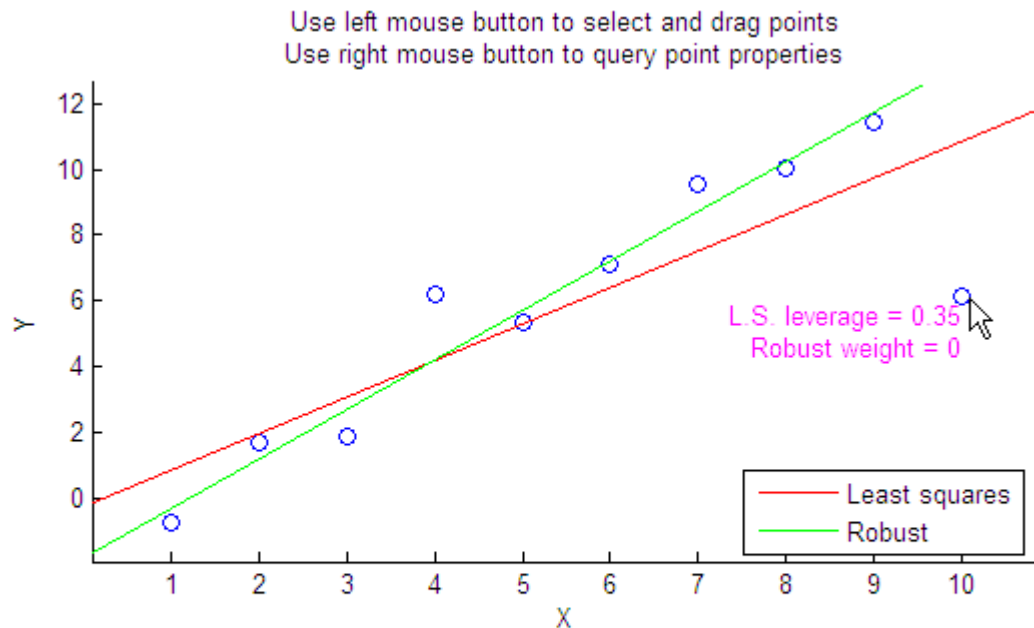




Least squares:	$Y = -0.188327 + 1.10351 * X$	RMS error = 2.21375
Robust:	$Y = -1.77278 + 1.50415 * X$	RMS error = 1.43663

The resulting figure shows a scatter plot with two fitted lines. The red line is the fit using ordinary least-squares regression. The green line is the fit using robust regression. At the bottom of the figure are the equations for the fitted lines, together with the estimated root mean squared errors for each fit.

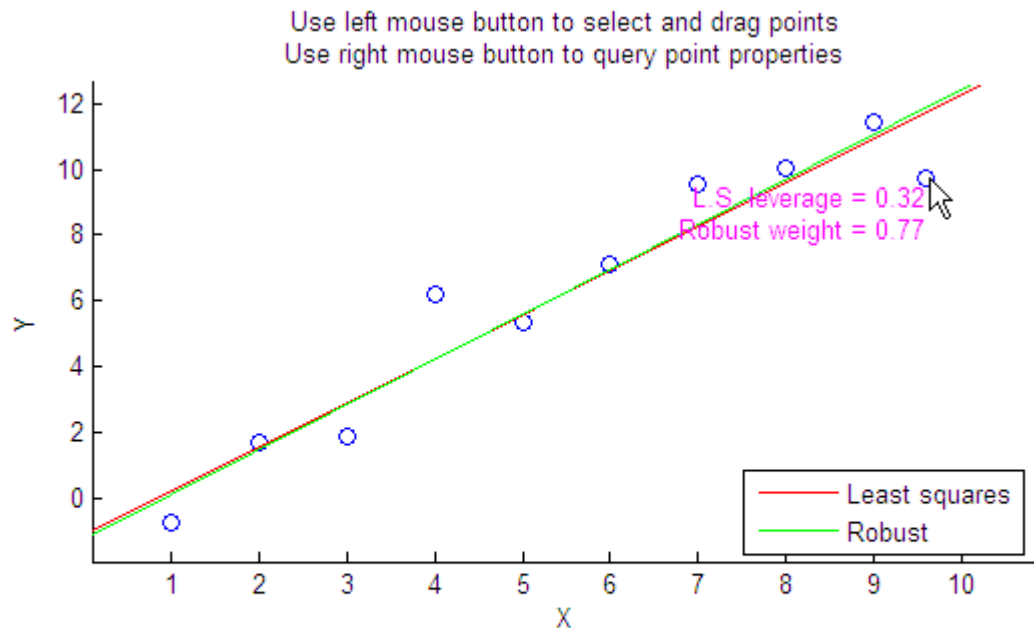
- 2 View leverages and robust weights.** Right-click on any data point to see its least-squares leverage and robust weight:



Least squares:	$Y = -0.188327 + 1.10351 * X$	RMS error = 2.21375
Robust:	$Y = -1.77278 + 1.50415 * X$	RMS error = 1.43663

In the built-in data, the right-most point has a relatively high leverage of 0.35. The point exerts a large influence on the least-squares fit, but its small robust weight shows that it is effectively excluded from the robust fit.

- 3 See how changes in the data affect the fits.** With the left mouse button, click and hold on any data point and drag it to a new location. When you release the mouse button, the displays update:



Least squares:	$Y = -1.0661 + 1.33785 \cdot X$	RMS error = 1.21477
Robust:	$Y = -1.18916 + 1.36459 \cdot X$	RMS error = 1.27697

Bringing the right-most data point closer to the least-squares line makes the two fitted lines nearly identical. The adjusted right-most data point has significant weight in the robust fit.

## See Also

[leverage](#) | [robustfit](#)

Introduced before R2006a

## robustfit

Robust regression

### Syntax

```
b = robustfit(X,y)
b = robustfit(X,y,wfun,tune)
b = robustfit(X,y,wfun,tune,const)
[b,stats] = robustfit(...)
```

### Description

`b = robustfit(X,y)` returns a  $(p + 1)$ -by-1 vector `b` of coefficient estimates for a robust multilinear regression of the responses in `y` on the predictors in `X`. `X` is an  $n$ -by- $p$  matrix of  $p$  predictors at each of  $n$  observations. `y` is an  $n$ -by-1 vector of observed responses. By default, the algorithm uses iteratively reweighted least squares with a bisquare weighting function.

---

**Note** By default, `robustfit` adds a first column of 1s to `X`, corresponding to a constant term in the model. Do not enter a column of 1s directly into `X`. You can change the default behavior of `robustfit` using the input `const`, below.

---

`robustfit` treats NaNs in `X` or `y` as missing values, and removes them.

`b = robustfit(X,y,wfun,tune)` specifies a weighting function `wfun`. `tune` is a tuning constant that is divided into the residual vector before computing weights.

The weighting function `wfun` can be any one of the following:

Weight Function	Equation	Default Tuning Constant
'andrews'	$w = (\text{abs}(r) < \pi) \cdot \sin(r) \cdot / r$	1.339

Weight Function	Equation	Default Tuning Constant
'bisquare' (default)	$w = (\text{abs}(r) < 1) .* (1 - r.^2).^2$	4.685
'cauchy'	$w = 1 ./ (1 + r.^2)$	2.385
'fair'	$w = 1 ./ (1 + \text{abs}(r))$	1.400
'huber'	$w = 1 ./ \max(1, \text{abs}(r))$	1.345
'logistic'	$w = \tanh(r) ./ r$	1.205
'ols'	Ordinary least squares (no weighting function)	None
'talwar'	$w = 1 * (\text{abs}(r) < 1)$	2.795
'welsch'	$w = \exp(-r.^2)$	2.985

If `tune` is unspecified, the default value in the table is used. Default tuning constants give coefficient estimates that are approximately 95% as statistically efficient as the ordinary least-squares estimates, provided the response has a normal distribution with no outliers. Decreasing the tuning constant increases the downweight assigned to large residuals; increasing the tuning constant decreases the downweight assigned to large residuals.

The value `r` in the weight functions is

$$r = \text{resid} / (\text{tune} * s * \sqrt{1-h})$$

where `resid` is the vector of residuals from the previous iteration, `h` is the vector of leverage values from a least-squares fit, and `s` is an estimate of the standard deviation of the error term given by

$$s = \text{MAD} / 0.6745$$

Here `MAD` is the median absolute deviation of the residuals from their median. The constant 0.6745 makes the estimate unbiased for the normal distribution. If there are  $p$  columns in  $X$ , the smallest  $p$  absolute deviations are excluded when computing the median.

You can write your own weight function. The function must take a vector of scaled residuals as input and produce a vector of weights as output. In this case, `wfun` is specified using a function handle `@` (as in `@myfun`), and the input `tune` is required.

`b = robustfit(X, y, wfun, tune, const)` controls whether or not the model will include a constant term. `const` is 'on' to include the constant term (the default), or 'off' to omit it. When `const` is 'on', `robustfit` adds a first column of 1s to `X` and `b` becomes a  $(p + 1)$ -by-1 vector. When `const` is 'off', `robustfit` does not alter `X`, then `b` is a  $p$ -by-1 vector.

`[b, stats] = robustfit(...)` returns the structure `stats`, whose fields contain diagnostic statistics from the regression. The fields of `stats` are:

- `ols_s` — Sigma estimate (RMSE) from ordinary least squares
- `robust_s` — Robust estimate of sigma
- `mad_s` — Estimate of sigma computed using the median absolute deviation of the residuals from their median; used for scaling residuals during iterative fitting
- `s` — Final estimate of sigma, the larger of `robust_s` and a weighted average of `ols_s` and `robust_s`
- `resid` — Residual
- `rstud` — Studentized residual (see `regress` for more information)
- `se` — Standard error of coefficient estimates
- `covb` — Estimated covariance matrix for coefficient estimates
- `coeffcorr` — Estimated correlation of coefficient estimates
- `t` — Ratio of `b` to `se`
- `p` —  $p$ -values for `t`
- `w` — Vector of weights for robust fit
- `R` —  $R$  factor in  $QR$  decomposition of `X`
- `dfe` — Degrees of freedom for error
- `h` — Vector of leverage values for least-squares fit

The `robustfit` function estimates the variance-covariance matrix of the coefficient estimates using `inv(X'*X)*stats.s^2`. Standard errors and correlations are derived from this estimate.

## Examples

## Compare Robust and Least-Squares Regression

Generate data with the trend  $y = 10 - 2x$ , then change one value to simulate an outlier.

```
x = (1:10)';  
rng default; % For reproducibility  
y = 10 - 2*x + randn(10,1);  
y(10) = 0;
```

Fit a straight line using ordinary least squares regression.

```
bls = regress(y, [ones(10,1) x])
```

```
bls =  
  
    7.8518  
   -1.3644
```

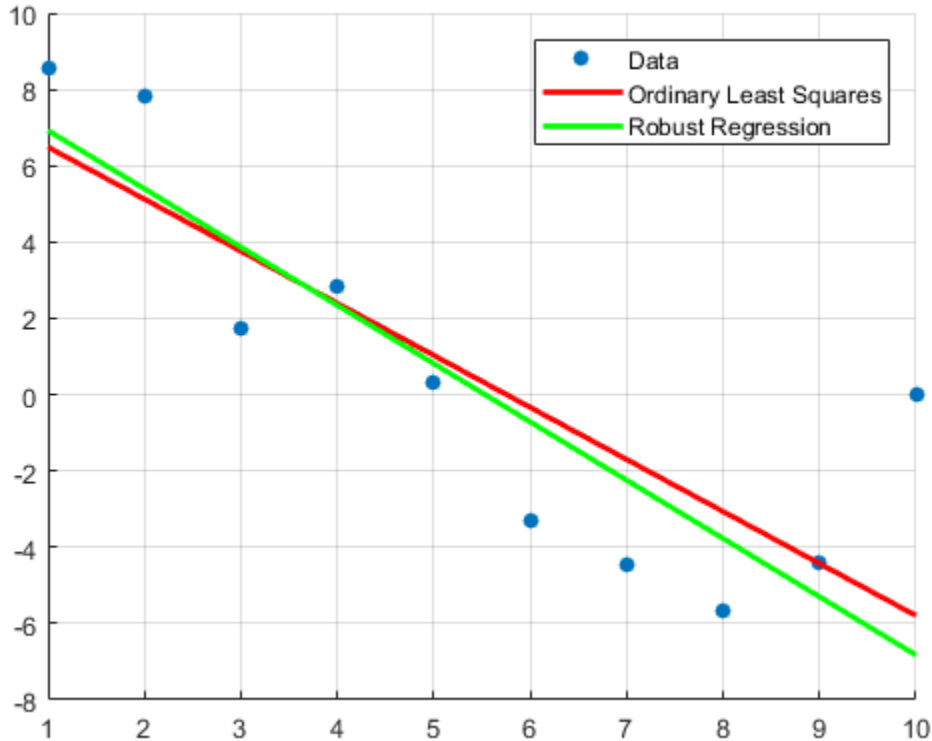
Now use robust regression to estimate a straight-line fit.

```
brob = robustfit(x,y)
```

```
brob =  
  
    8.4504  
   -1.5278
```

Create scatter plot of the data together with the fits.

```
scatter(x,y,'filled'); grid on; hold on  
plot(x,bls(1)+bls(2)*x,'r','LineWidth',2);  
plot(x,brob(1)+brob(2)*x,'g','LineWidth',2)  
legend('Data','Ordinary Least Squares','Robust Regression')
```



The robust fit is less influenced by the outlier than the least-squares fit.

## References

- [1] DuMouchel, W. H., and F. L. O'Brien. "Integrating a Robust Option into a Multiple Regression Computing Environment." *Computer Science and Statistics: Proceedings of the 21st Symposium on the Interface*. Alexandria, VA: American Statistical Association, 1989.
- [2] Holland, P. W., and R. E. Welsch. "Robust Regression Using Iteratively Reweighted Least-Squares." *Communications in Statistics: Theory and Methods*, A6, 1977, pp. 813–827.



[3] Huber, P. J. *Robust Statistics*. Hoboken, NJ: John Wiley & Sons, Inc., 1981.

[4] Street, J. O., R. J. Carroll, and D. Ruppert. “A Note on Computing Robust Regression Estimates via Iteratively Reweighted Least Squares.” *The American Statistician*. Vol. 42, 1988, pp. 152–154.

## See Also

`regress` | `robustdemo`

Introduced before R2006a

## rotatefactors

Rotate factor loadings

### Syntax

```
B = rotatefactors(A)
B = rotatefactors(A, 'Method', 'orthomax', 'Coeff', gamma)
B = rotatefactors(A, 'Method', 'procrustes', 'Target', target)
B = rotatefactors(A, 'Method', 'pattern', 'Target', target)
B = rotatefactors(A, 'Method', 'promax')
[B,T] = rotatefactors(A,...)
```

### Description

`B = rotatefactors(A)` rotates the  $d$ -by- $m$  loadings matrix  $A$  to maximize the varimax criterion, and returns the result in  $B$ . Rows of  $A$  and  $B$  correspond to variables and columns correspond to factors, for example, the  $(i, j)$ th element of  $A$  is the coefficient for the  $i$ th variable on the  $j$ th factor. The matrix  $A$  usually contains principal component coefficients created with `pca` or `pcacov`, or factor loadings estimated with `factoran`.

`B = rotatefactors(A, 'Method', 'orthomax', 'Coeff', gamma)` rotates  $A$  to maximize the orthomax criterion with the coefficient  $\gamma$ , i.e.,  $B$  is the orthogonal rotation of  $A$  that maximizes

$$\text{sum}(D * \text{sum}(B.^4, 1) - \text{GAMMA} * \text{sum}(B.^2, 1).^2)$$

The default value of 1 for  $\gamma$  corresponds to varimax rotation. Other possibilities include  $\gamma = 0$ ,  $m/2$ , and  $d(m - 1)/(d + m - 2)$ , corresponding to quartimax, equamax, and parsimax. You can also supply 'varimax', 'quartimax', 'equamax', or 'parsimax' for the 'method' parameter and omit the 'Coeff' parameter.

If 'Method' is 'orthomax', 'varimax', 'quartimax', 'equamax', or 'parsimax', then additional parameters are

- 'Normalize' — Flag indicating whether the loadings matrix should be row-normalized for rotation. If 'on' (the default), rows of  $A$  are normalized prior to

rotation to have unit Euclidean norm, and unnormalized after rotation. If 'off', the raw loadings are rotated and returned.

- 'Reltol' — Relative convergence tolerance in the iterative algorithm used to find  $T$ . The default is `sqrt(eps)`.
- 'Maxit' — Iteration limit in the iterative algorithm used to find  $T$ . The default is 250.

`B = rotatefactors(A, 'Method', 'procrustes', 'Target', target)` performs an oblique procrustes rotation of  $A$  to the  $d$ -by- $m$  target loadings matrix `target`.

`B = rotatefactors(A, 'Method', 'pattern', 'Target', target)` performs an oblique rotation of the loadings matrix  $A$  to the  $d$ -by- $m$  target pattern matrix `target`, and returns the result in  $B$ . `target` defines the "restricted" elements of  $B$ , i.e., elements of  $B$  corresponding to zero elements of `target` are constrained to have small magnitude, while elements of  $B$  corresponding to nonzero elements of `target` are allowed to take on any magnitude.

If 'Method' is 'procrustes' or 'pattern', an additional parameter is 'Type', the type of rotation. If 'Type' is 'orthogonal', the rotation is orthogonal, and the factors remain uncorrelated. If 'Type' is 'oblique' (the default), the rotation is oblique, and the rotated factors might be correlated.

When 'Method' is 'pattern', there are restrictions on `target`. If  $A$  has  $m$  columns, then for orthogonal rotation, the  $j$ th column of `target` must contain at least  $m - j$  zeros. For oblique rotation, each column of `target` must contain at least  $m - 1$  zeros.

`B = rotatefactors(A, 'Method', 'promax')` rotates  $A$  to maximize the promax criterion, equivalent to an oblique Procrustes rotation with a target created by an orthomax rotation. Use the four orthomax parameters to control the orthomax rotation used internally by promax.

An additional parameter for 'promax' is 'Power', the exponent for creating promax target matrix. 'Power' must be 1 or greater. The default is 4.

`[B, T] = rotatefactors(A, ...)` returns the rotation matrix  $T$  used to create  $B$ , that is,  $B = A * T$ . You can find the correlation matrix of the rotated factors by using `inv(T' * T)`. For orthogonal rotation, this is the identity matrix, while for oblique rotation, it has unit diagonal elements but nonzero off-diagonal elements.

## Examples

```
rng('default') % for reproducibility
X = randn(100,10);

% Default (normalized varimax) rotation:
% first three principal components.
LPC = pca(X);
[L1,T] = rotatefactors(LPC(:,1:3));

% Equamax rotation:
% first three principal components.
[L2,T] = rotatefactors(LPC(:,1:3),...
    'method','equamax');

% Promax rotation:
% first three factors.
LFA = factoran(X,3,'Rotate','none');
[L3,T] = rotatefactors(LFA(:,1:3),...
    'method','promax',...
    'power',2);

% Pattern rotation:
% first three factors.
Tgt = [1 1 1 1 1 0 1 0 1 1; ...
    0 0 0 1 1 1 0 0 0 0; ...
    1 0 0 1 0 1 1 1 1 0]';
[L4,T] = rotatefactors(LFA(:,1:3),...
    'method','pattern',...
    'target',Tgt);
inv(T'*T) % Correlation matrix of the rotated factors
ans =

    1.0000    -0.9593   -0.7098
   -0.9593    1.0000    0.5938
   -0.7098    0.5938    1.0000
```

## References

- [1] Harman, H. H. *Modern Factor Analysis*. 3rd ed. Chicago: University of Chicago Press, 1976.

[2] Lawley, D. N., and A. E. Maxwell. *Factor Analysis as a Statistical Method*. 2nd ed. New York: American Elsevier Publishing, 1971.

## See Also

biplot | factoran | pca | pcacov | procrustes

**Introduced before R2006a**

## rowexch

Row exchange

### Syntax

```
dRE = rowexch(nfactors, nruns)
[dRE, X] = rowexch(nfactors, nruns)
[dRE, X] = rowexch(nfactors, nruns, model)
[dRE, X] = rowexch(..., param1, val1, param2, val2, ...)
```

### Description

`dRE = rowexch(nfactors, nruns)` uses a row-exchange algorithm to generate a  $D$ -optimal design `dRE` with `nruns` runs (the rows of `dRE`) for a linear additive model with `nfactors` factors (the columns of `dRE`). The model includes a constant term.

`[dRE, X] = rowexch(nfactors, nruns)` also returns the associated design matrix `X`, whose columns are the model terms evaluated at each treatment (row) of `dRE`.

`[dRE, X] = rowexch(nfactors, nruns, model)` uses the linear regression model specified in `model`. `model` is one of the following:

- 'linear' — Constant and linear terms. This is the default.
- 'interaction' — Constant, linear, and interaction terms
- 'quadratic' — Constant, linear, interaction, and squared terms
- 'purequadratic' — Constant, linear, and squared terms

The order of the columns of `X` for a full quadratic model with  $n$  terms is:

- 1 The constant term
- 2 The linear terms in order 1, 2, ...,  $n$
- 3 The interaction terms in order (1, 2), (1, 3), ..., (1,  $n$ ), (2, 3), ..., ( $n-1$ ,  $n$ )
- 4 The squared terms in order 1, 2, ...,  $n$

Other models use a subset of these terms, in the same order.

Alternatively, *model* can be a matrix specifying polynomial terms of arbitrary order. In this case, *model* should have one column for each factor and one row for each term in the model. The entries in any row of *model* are powers for the factors in the columns. For example, if a model has factors  $X_1$ ,  $X_2$ , and  $X_3$ , then a row  $[0 \ 1 \ 2]$  in *model* specifies the term  $(X_1.^0) .* (X_2.^1) .* (X_3.^2)$ . A row of all zeros in *model* specifies a constant term, which can be omitted.

`[dRE,X] = rowexch(...,param1,val1,param2,val2,...)` specifies additional parameter/value pairs for the design. Valid parameters and their values are listed in the following table.

Parameter	Value
'bounds'	Lower and upper bounds for each factor, specified as a 2-by- <code>nfactors</code> matrix. Alternatively, this value can be a cell array containing <code>nfactors</code> elements, each element specifying the vector of allowable values for the corresponding factor.
'categorical'	Indices of categorical predictors.
'display'	Either 'on' or 'off' to control display of the iteration counter. The default is 'on'.
'excldefun'	Handle to a function that excludes undesirable runs. If the function is $f$ , it must support the syntax $b = f(S)$ , where $S$ is a matrix of treatments with <code>nfactors</code> columns and $b$ is a vector of Boolean values with the same number of rows as $S$ . $b(i)$ is true if the $i$ th row $S$ should be excluded.
'init'	Initial design as an <code>nruns</code> -by- <code>nfactors</code> matrix. The default is a randomly selected set of points.
'levels'	Vector of number of levels for each factor.
'maxiter'	Maximum number of iterations. The default is 10.

Parameter	Value
options	<p>A structure that specifies whether to run in parallel, and specifies the random stream or streams. Create the options structure with <code>statset</code>. Option fields:</p> <ul style="list-style-type: none"> <li>• <code>UseParallel</code> — Set to <code>true</code> to compute in parallel. Default is <code>false</code>.</li> <li>• <code>UseSubstreams</code> — Set to <code>true</code> to compute in parallel in a reproducible fashion. Default is <code>false</code>. To compute reproducibly, set <code>Streams</code> to a type allowing substreams: <code>'mlfg6331_64'</code> or <code>'mrg32k3a'</code>.</li> <li>• <code>Streams</code> — A <code>RandStream</code> object or cell array of such objects. If you do not specify <code>Streams</code>, <code>rowexch</code> uses the default stream or streams. If you choose to specify <code>Streams</code>, use a single object except in the case <ul style="list-style-type: none"> <li>• You have an open Parallel pool</li> <li>• <code>UseParallel</code> is <code>true</code></li> <li>• <code>UseSubstreams</code> is <code>false</code></li> </ul> <p>In that case, use a cell array the same size as the Parallel pool.</p> </li> </ul>
'tries'	<p>Number of times to try to generate a design from a new starting point. The algorithm uses random points for each try, except possibly the first. The default is 1.</p>

## Examples

Suppose you want a design to estimate the parameters in the following three-factor, seven-term interaction model:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + \beta_{12} x_1 x_2 + \beta_{13} x_1 x_3 + \beta_{23} x_2 x_3 + \varepsilon$$

Use `rowexch` to generate a  $D$ -optimal design with seven runs:

```
nfactors = 3;
nruns = 7;
[dRE,X] = rowexch(nfactors,nruns,'interaction','tries',10)
```



```

dRE =
  -1  -1   1
   1  -1   1
   1  -1  -1
   1   1   1
  -1  -1  -1
  -1   1  -1
  -1   1   1

X =
   1  -1  -1   1   1  -1  -1
   1   1  -1   1  -1   1  -1
   1   1  -1  -1  -1  -1   1
   1   1   1   1   1   1   1
   1  -1  -1  -1  -1   1   1
   1  -1   1  -1  -1   1  -1
   1  -1   1   1  -1  -1   1

```

Columns of the design matrix  $X$  are the model terms evaluated at each row of the design  $dRE$ . The terms appear in order from left to right: constant term, linear terms (1, 2, 3), interaction terms (12, 13, 23). Use  $X$  to fit the model, as described in “Linear Regression” on page 11-11, to response data measured at the design points in  $dRE$ .

## Algorithms

Both `cordexch` and `rowexch` use iterative search algorithms. They operate by incrementally changing an initial design matrix  $X$  to increase  $D = |X^T X|$  at each step. In both algorithms, there is randomness built into the selection of the initial design and into the choice of the incremental changes. As a result, both algorithms may return locally, but not globally,  $D$ -optimal designs. Run each algorithm multiple times and select the best result for your final design. Both functions have a `'tries'` parameter that automates this repetition and comparison.

At each step, the row-exchange algorithm exchanges an entire row of  $X$  with a row from a design matrix  $C$  evaluated at a candidate set of feasible treatments. The `rowexch` function automatically generates a  $C$  appropriate for a specified model, operating in two steps by calling the `candgen` and `candexch` functions in sequence. Provide your own  $C$  by calling `candexch` directly. In either case, if  $C$  is large, its static presence in memory can affect computation.

## See Also

`candexch` | `candgen` | `cordexch`

**Introduced before R2006a**

# rsmdemo

Interactive response surface demonstration

## Syntax

rsmdemo

## Description

rsmdemo opens a group of three graphical user interfaces for interactively investigating response surface methodology (RSM), nonlinear fitting, and the design of experiments.

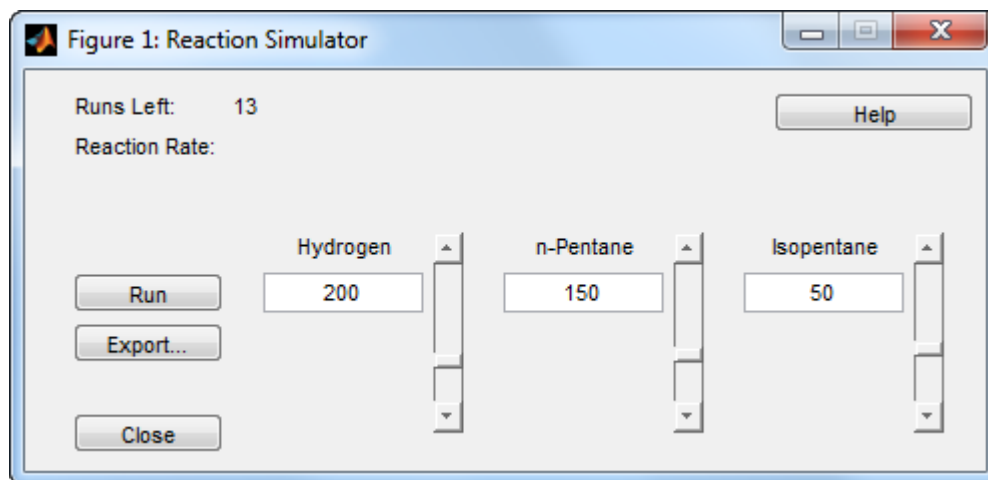
The interfaces allow you to collect and model data from a simulated chemical reaction. Experimental predictors are concentrations of three reactants (hydrogen, *n*-Pentane, and isopentane) and the response is the reaction rate. The reaction rate is simulated by a Hougen-Watson model (Bates and Watts, [2] on page C-2, pp. 271–272):

$$rate = \frac{\beta_1 x_2 - x_3 / \beta_5}{1 + \beta_2 x_1 + \beta_3 x_2 + \beta_4 x_3}$$

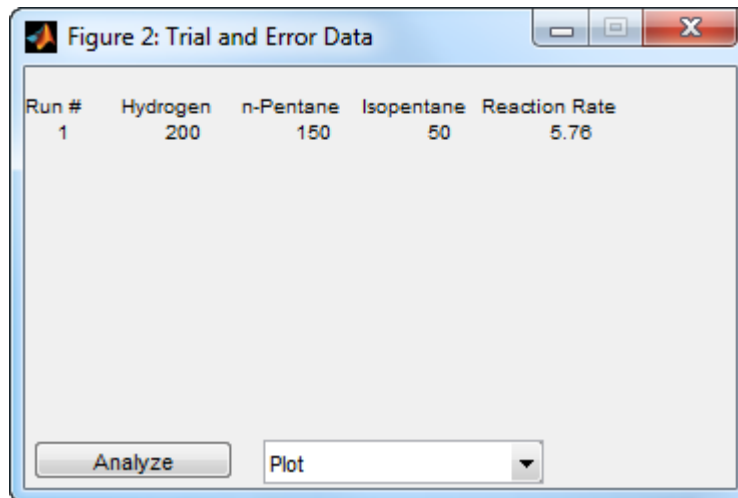
where *rate* is the reaction rate,  $x_1$ ,  $x_2$ , and  $x_3$  are the concentrations of hydrogen, *n*-pentane, and isopentane, respectively, and  $\beta_1, \beta_2, \dots, \beta_5$  are fixed parameters. Random errors are used to perturb the reaction rate for each combination of reactants.

Collect data using one of two methods:

- 1 Manually set reactant concentrations in the **Reaction Simulator** interface by editing the text boxes or by adjusting the associated sliders.

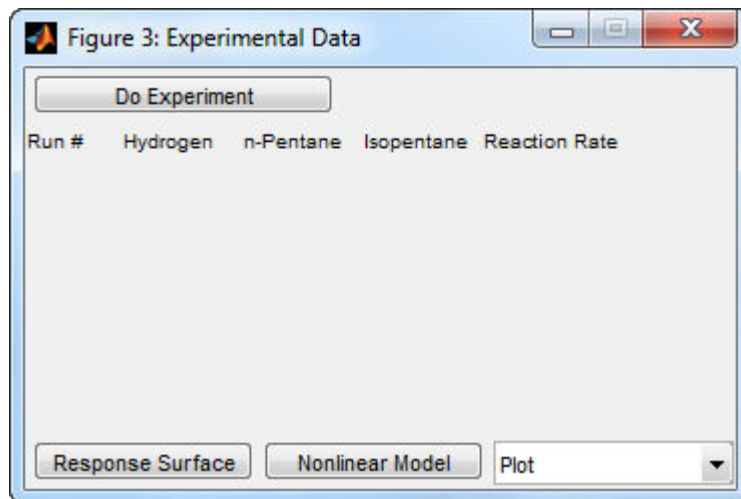


When you click **Run**, the concentrations and simulated reaction rate are recorded on the **Trial and Error Data** interface.



You are allowed up to 13 independent experimental runs for data collection.

- 2 Use a designed experiment to set reactant concentrations in the **Experimental Data** interface by clicking the **Do Experiment** button.



A 13-run *D*-optimal design for a full quadratic model is generated by the `cordexch` function, and the concentrations and simulated reaction rates are recorded on the same interface.

Run #	Hydrogen	n-Pentane	Isopentane	Reaction Rate
1	100	300	120	7.63
2	100	80	10	7.84
3	100	80	120	0.04
4	100	300	10	17.26
5	470	300	10	8.38
6	100	190	65	8.54
7	470	80	10	2.49
8	285	80	120	0.03
9	470	300	120	4.68
10	285	300	65	8.66
11	470	190	120	2.42
12	285	190	10	8.28
13	470	80	65	1.12

Once data is collected, scatter plots of reaction rates vs. individual predictors are generated by selecting one of the following from the **Plot** pop-up menu below the recorded data:

- **Hydrogen vs. Rate**
- **n-Pentane vs. Rate**
- **Isopentane vs. Rate**

Fit a response surface model to the data by clicking the **Analyze** button below the trial-and-error data or the **Response Surface** button below the experimental data. Both buttons load the data into the Response Surface Tool `rstool`. By default, trial-and-error data is fit with a linear additive model and experimental data is fit with a full quadratic model, but the models can be adjusted in the Response Surface Tool.

For experimental data, you have the additional option of fitting a Hougen-Watson model. Click the **Nonlinear Model** button to load the data and the model in `hougen` into the Nonlinear Fitting Tool `nlintool`.

## See Also

`cordexch` | `hougen` | `nlintool` | `rstool`

**Introduced before R2006a**

# **rstool**

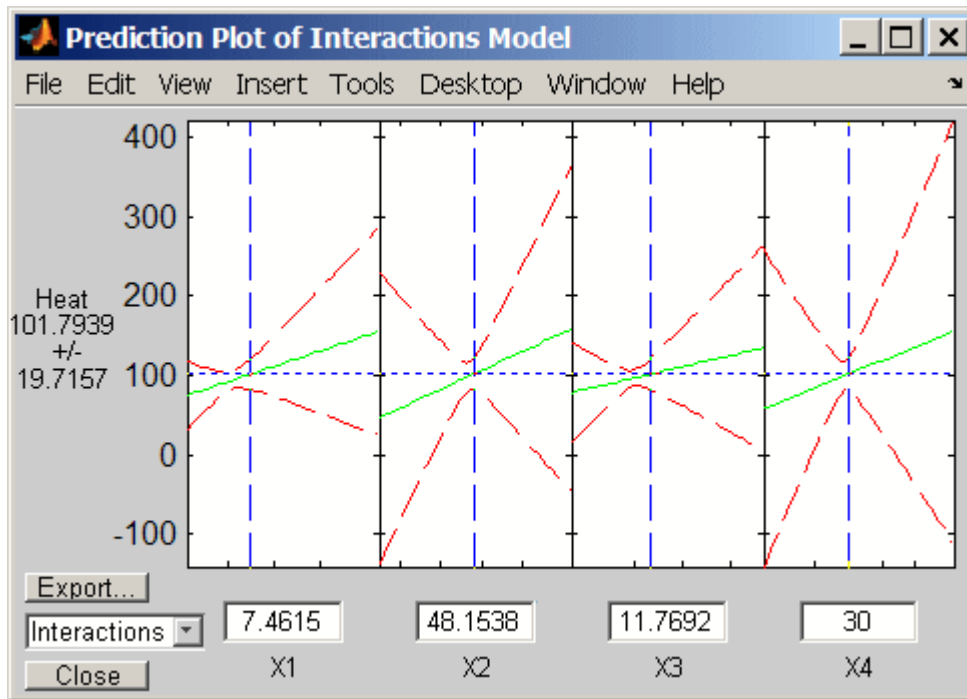
Interactive response surface modeling

## **Syntax**

```
rstool  
rstool(X,Y,model)  
rstool(x,y,model,alpha)  
rstool(x,y,model,alpha,xname,yname)
```

## **Description**

`rstool` opens a graphical user interface for interactively investigating one-dimensional contours of multidimensional response surface models.



By default, the interface opens with the data from `hald.mat` and a fitted response surface with constant, linear, and interaction terms.

A sequence of plots is displayed, each showing a contour of the response surface against a single predictor, with all other predictors held fixed. `rstool` plots a 95% simultaneous confidence band for the fitted response surface as two red curves. Predictor values are displayed in the text boxes on the horizontal axis and are marked by vertical dashed blue lines in the plots. Predictor values are changed by editing the text boxes or by dragging the dashed blue lines. When you change the value of a predictor, all plots update to show the new point in predictor space.

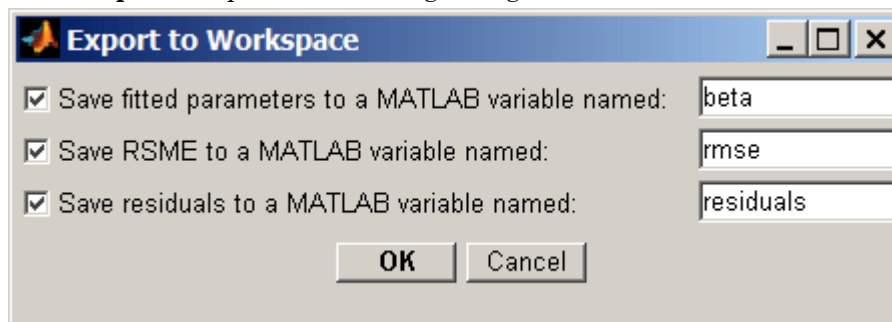
The pop-up menu at the lower left of the interface allows you to choose among the following models:

- Linear — Constant and linear terms (the default)
- Pure Quadratic — Constant, linear, and squared terms



- `Interactions` — Constant, linear, and interaction terms
- `Full Quadratic` — Constant, linear, interaction, and squared terms

Click **Export** to open the following dialog box:



The dialog allows you to save information about the fit to MATLAB workspace variables with valid names.

`rstool(X, Y, model)` opens the interface with the predictor data in `X`, the response data in `Y`, and the fitted model `model`. Distinct predictor variables should appear in different columns of `X`. `Y` can be a vector, corresponding to a single response, or a matrix, with columns corresponding to multiple responses. `Y` must have as many elements (or rows, if it is a matrix) as `X` has rows.

The optional input `model` can be any one of the following:

- `'linear'` — Constant and linear terms (the default)
- `'purequadratic'` — Constant, linear, and squared terms
- `'interaction'` — Constant, linear, and interaction terms
- `'quadratic'` — Constant, linear, interaction, and squared terms

To specify a polynomial model of arbitrary order, or a model without a constant term, use a matrix for `model` as described in `x2fx`.

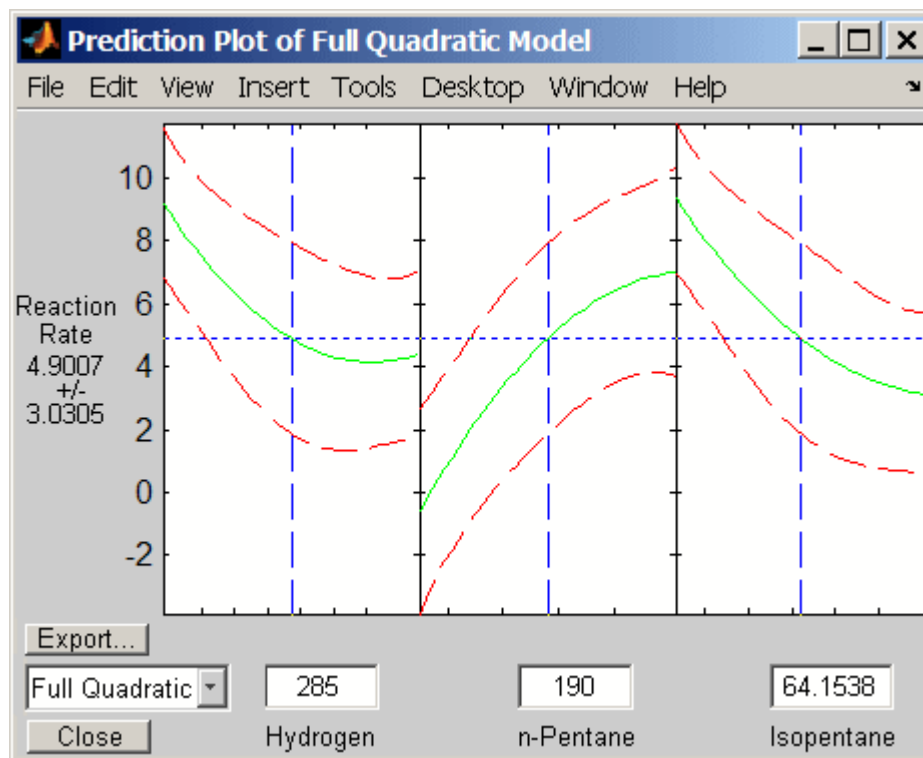
`rstool(x, y, model, alpha)` uses  $100(1-\alpha)\%$  global confidence intervals for new observations in the plots.

`rstool(x, y, model, alpha, xname, yname)` labels the axes using `xname` and `yname`. To label each subplot differently, `xname` and `yname` can be cell arrays of character vectors.

## Examples

The following uses `rstool` to visualize a quadratic response surface model of the 3-D chemical reaction data in `reaction.mat`:

```
load reaction
alpha = 0.01; % Significance level
rstool(reactants,rate,'quadratic',alpha,xn,yn)
```



The `rstool` interface is used by `rsmdemo` to visualize the results of simulated experiments with data like that in `reaction.mat`. As described in “Response Surface Designs” on page 27-9, `rsmdemo` uses a response surface model to generate simulated data at combinations of predictors specified by either the user or by a designed experiment.

## See Also

`nlintool` | `rsmdemo` | `x2fx`

**Introduced before R2006a**

## runstest

Run test for randomness

### Syntax

```
h = runstest(x)
h = runstest(x,v)
h = runstest(x,'ud')
h = runstest( ____,Name,Value)
[h,p,stats] = runstest( ____,Name,Value)
```

### Description

`h = runstest(x)` returns a test decision for the null hypothesis that the values in the data vector `x` come in random order, against the alternative that they do not. The test is based on the number of runs of consecutive values above or below the mean of `x`. The result `h` is 1 if the test rejects the null hypothesis at the 5% significance level, or 0 otherwise.

`h = runstest(x,v)` returns a test decision based on the number of runs of consecutive values above or below the specified reference value `v`. Values exactly equal to `v` are discarded.

`h = runstest(x,'ud')` returns a test decision based on the number of runs up or down. Too few runs indicate a trend, while too many runs indicate an oscillation. Values exactly equal to the preceding value are discarded.

`h = runstest( ____,Name,Value)` returns a test decision using additional options specified by one or more name-value pair arguments. For example, you can change the significance level of the test, specify the algorithm used to calculate the *p*-value, or conduct a one-sided test.

`[h,p,stats] = runstest( ____,Name,Value)` also returns the *p*-value of the test `p`, and a structure `stats` containing additional data about the test.

## Examples

### Test Data for Randomness Using Sample Median

Generate a vector of 40 random numbers from a standard normal distribution.

```
rng default; % for reproducibility
x = randn(40,1);
```

Test whether the values in `x` appear in random order, using the sample median as the reference value.

```
[h,p] = runstest(x,median(x))
```

```
h =
```

```
0
```

```
p =
```

```
0.8762
```

The returned value of `h = 0` indicates that `runstest` does not reject the null hypothesis that the values in `x` are in random order at the default 5% significance level.

## Input Arguments

### **x** — Data vector

vector of scalar values

Data vector, specified as a vector of scalar values. `runstest` treats NaN values in `x` as missing values, and ignores them.

Data Types: `single` | `double`

### **v** — Reference value

mean of `x` (default) | scalar value

Reference value, specified as a scalar value. If you specify a value for `v`, then `runstest` performs the hypothesis test based on the number of runs of consecutive values above or below `v`. `runstest` discards values exactly equal to `v`.

Data Types: `single` | `double`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example:

### **Alpha** — Significance level

`0.05` (default) | scalar value in the range (0,1)

Significance level of the hypothesis test, specified as the comma-separated pair consisting of 'Alpha' and a scalar value in the range (0,1).

Example: `'Alpha', 0.01`

Data Types: `single` | `double`

### **Method** — Method used to compute $p$ -value

`'exact'` | `'approximate'`

Method used to compute  $p$ -value, specified as the comma-separated pair consisting of 'Method' and either 'exact' to use an exact algorithm, or 'approximate' to use a normal approximation. The default is 'exact' for runs above/below, and for runs up/down when the length of `x` is less than or equal to 50. If `runstest` tests for runs up/down and the length of `x` is greater than 50, then the default is 'approximate', and the 'exact' method is not available.

Example: `'Method', 'approximate'`

### **Tail** — Type of alternative hypothesis

`'both'` (default) | `'right'` | `'left'`

Type of alternative hypothesis, specified as the comma-separated pair consisting of 'Tail' and one of the following.

<code>'both'</code>	Two-tailed test (sequence is not random)
<code>'right'</code>	Right-tailed test (like values separate for runs above/below, direction alternates for runs up/down)
<code>'left'</code>	Left-tailed test (like values cluster for runs above/below, values trend for runs up/down)

Example: `'Tail', 'right'`

## Output Arguments

### **h** — Hypothesis test result

1 | 0

Hypothesis test result, returned as a logical value.

- If  $h = 1$ , then `runstest` rejects the null hypothesis at the Alpha significance level.
- If  $h = 0$ , then `runstest` fails to reject the null hypothesis at the Alpha significance level.

The result in `runstest` is based on the number of runs of consecutive values above or below the mean of  $x$ . Too few runs indicate a tendency for high and low values to cluster. Too many runs indicate a tendency for high and low values to alternate.

`runstest` uses a test statistic which is the difference between the number of runs and its mean, divided by its standard deviation. The test statistic is approximately normally distributed when the null hypothesis is true.

### **p** — *p*-value

scalar value in the range [0,1]

*p*-value of the test, returned as a scalar value in the range [0,1]. *p* is the probability of observing a test statistic as extreme as, or more extreme than, the observed value under the null hypothesis. Small values of *p* cast doubt on the validity of the null hypothesis.

*p* is computed from either the test statistic or the exact distribution of the number of runs, depending on the value specified for the `'Method'` name-value pair argument.

### **stats** — Test data

structure

Test data, returned as a structure with the following fields.

- `nruns` — The number of runs
- `n1` — The number of values above `v`
- `n0` — The number of values below `v`
- `z` — The test statistic

## See Also

`signrank` | `signtest`

**Introduced before R2006a**



## SampleWithReplacement property

**Class:** TreeBagger

Flag to sample with replacement

### Description

The `SampleWithReplacement` property is a logical flag specifying if data are sampled for each decision tree with replacement. True if `TreeBagger` samples data with replacement and false otherwise. True by default.

## sampsizepwr

Sample size and power of test

`sampsizepwr` computes the sample size, power, or alternative parameter value for a hypothesis test, given the other two values. For example, you can compute the sample size required to obtain a particular power for a hypothesis test, given the parameter value of the alternative hypothesis.

### Syntax

```
nout = sampsizepwr(testtype,p0,p1)
nout = sampsizepwr(testtype,p0,p1,pwr)

pwrout = sampsizepwr(testtype,p0,p1,[],n)

plout = sampsizepwr(testtype,p0,[],pwr,n)

___ = sampsizepwr(testtype,p0,p1,pwr,n,Name,Value)
```

### Description

`nout = sampsizepwr(testtype,p0,p1)` returns the sample size, `nout`, required for a two-sided test of the type specified by `testtype` to have a power (probability of rejecting the null hypothesis when the alternative hypothesis is true) of 0.90 when the significance level (probability of rejecting the null hypothesis when the null hypothesis is true) is 0.05. `p0` specifies parameter values under the null hypothesis. `p1` specifies the value, or an array of values, of the single parameter being tested under the alternative hypothesis.

`nout = sampsizepwr(testtype,p0,p1,pwr)` returns the sample size, `nout`, that corresponds to the specified power, `pwr`, and the parameter value under the alternative hypothesis, `p1`.

`pwrout = sampsizepwr(testtype,p0,p1,[],n)` returns the power achieved for a sample size of `n` when the true parameter value is `p1`.

`p1out = sampsizewr(testtype,p0,[],pwr,n)` returns the parameter value detectable with the specified sample size, `n`, and the specified power, `pwr`.

`___ = sampsizewr(testtype,p0,p1,pwr,n,Name,Value)` returns any of the previous arguments using one or more name-value pair arguments. For example, you can change the significance level of the test, or specify a right- or left-tailed test. The name-value pairs can appear in any order but must begin in the sixth argument position.

## Examples

### Compute Sample Size for Selected Power Value

A company runs a manufacturing process that fills empty bottles with 100 mL of liquid. To monitor quality, the company randomly selects several bottles and measures the volume of liquid inside.

Determine the sample size the company must use if it wants to detect a difference between 100 mL and 102 mL with a power of 0.80. Assume that prior evidence indicates a standard deviation of 5 mL.

```
nout = sampsizewr('t',[100 5],102,0.80)
```

```
nout =
```

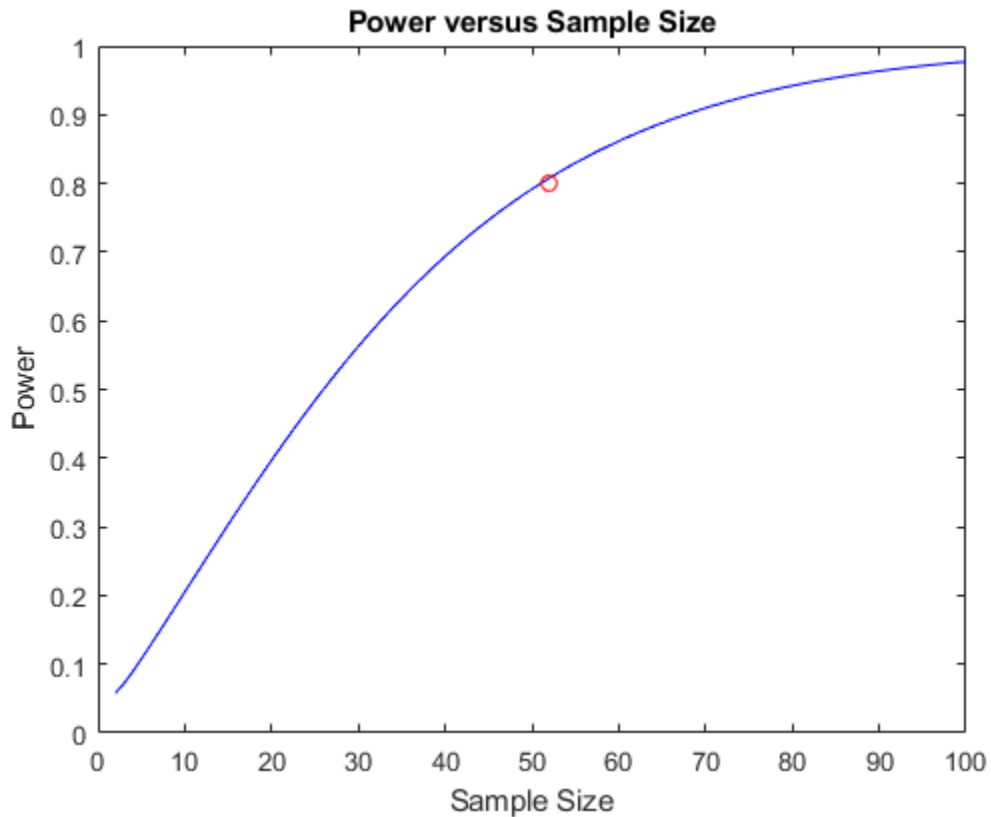
```
    52
```

The company must test 52 bottles to detect the difference between a mean volume of 100 mL and 102 mL with a power of 0.80.

Generate a power curve to visualize how the sample size affects the power of the test.

```
nn = 1:100;  
pwROUT = sampsizewr('t',[100 5],102,[],nn);
```

```
figure;  
plot(nn,pwROUT,'b-',nout,0.8,'ro')  
title('Power versus Sample Size')  
xlabel('Sample Size')  
ylabel('Power')
```



### Compute Power and Sample Size for One-Sided Test

An employee wants to buy a house near her office. She decides to eliminate from consideration any house that has a mean morning commute time greater than 20 minutes. The null hypothesis for this right-sided test is  $H_0: \mu = 20$ , and the alternative hypothesis is  $H_A: \mu > 20$ . The selected significance level is 0.05.

To determine the mean commute time, the employee takes a test drive from the house to her office during rush hour every morning for one week, so her total sample size is 5. She assumes that the standard deviation,  $\sigma$ , is equal to 5.

The employee decides that a true mean commute time of 25 minutes is too different from her targeted 20-minute limit, so she wants to detect a significant departure if the true mean is 25 minutes. Find the probability of incorrectly concluding that the mean commute time is no greater than 20 minutes.

Compute the power of the test, and then subtract the power from 1 to obtain  $\beta$ .

```
power = sampsizewr('t', [20 5], 25, [], 5, 'Tail', 'right');
beta = 1 - power
```

```
beta =
    0.4203
```

The  $\beta$  value indicates a probability of 0.4203 that the employee concludes incorrectly that the morning commute is not greater than 20 minutes.

The employee decides that this risk is too high, and she wants no more than a 0.01 probability of reaching an incorrect conclusion. Calculate the number of test drives the employee must take to obtain a power of 0.99.

```
nout = sampsizewr('t', [20 5], 25, 0.99, [], 'Tail', 'right')

nout =
    18
```

The results indicate that she must take 18 test drives from a candidate house to achieve this power level.

The employee decides that she only has time to take 10 test drives. She also accepts a 0.05 probability of making an incorrect conclusion. Calculate the smallest true parameter value that produces a detectable difference in mean commute time.

```
plout = sampsizewr('t', [20 5], [], 0.95, 10, 'Tail', 'right')

plout =
```

```
25.6532
```

Given the employee's target power level and sample size, her test detects a significant difference from a mean commute time of at least 25.6532 minutes.

### Compute Sample Size for a Binomial Test

Compute the sample size,  $n$ , required to distinguish  $p = 0.30$  from  $p = 0.36$ , using a binomial test with a power of 0.8.

```
napprox = sampsizepwr('p',0.30,0.36,0.8)
```

```
Warning: Values N>200 are approximate. Plotting the power as a function  
of N may reveal lower N values that have the required power.
```

```
napprox =
```

```
485
```

The result indicates that a power of 0.8 requires a sample size of 485. However, this result is approximate.

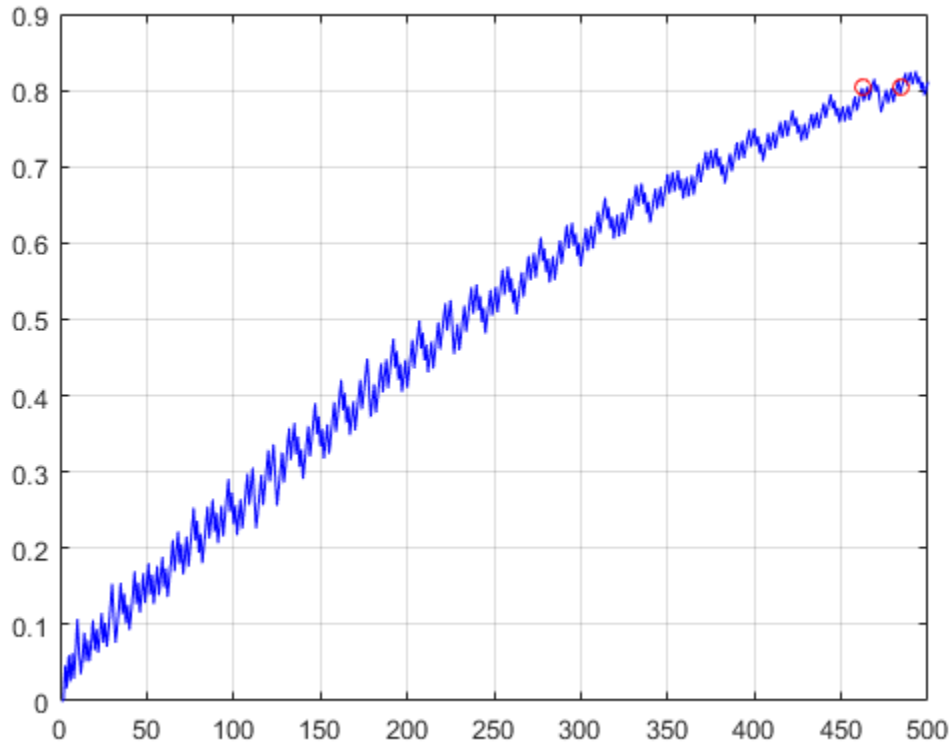
Make a plot to see if any smaller  $n$  values provide the required power of 0.8.

```
nn = 1:500;  
pwrou = sampsizepwr('p',0.3,0.36,[],nn);  
nexact = min(nn(pwrou>=0.8))
```

```
figure  
plot(nn,pwrou,'b-',[napprox nexact],pwrou([napprox nexact]),'ro')  
grid on
```

```
nexact =
```

```
462
```



The result indicates that a sample size of 462 also provides a power of 0.8 for this test.

### Compute Power for a Two-Sample t-Test

A farmer wants to test the impact of two different types of fertilizer on the yield of his bean crops. He currently uses Fertilizer A, but believes that Fertilizer B might improve crop yield. Because Fertilizer B is more expensive than Fertilizer A, the farmer wants to limit the number of plants he treats with Fertilizer B in this experiment.

The farmer uses a 2:1 ratio of plants in each treatment group. He tests 10 plants with Fertilizer A, and 5 plants with Fertilizer B. The mean yield using Fertilizer A is 1.4 kg

per plant, with a standard deviation of 0.2. The mean yield using Fertilizer B is 1.7 kg per plant. The significance level of the test is 0.05.

Compute the power of the test.

```
pwr = sampsizepwr('t2', [1.4 0.2], 1.7, [], 5, 'Ratio', 2)
```

```
pwr =
```

```
0.7165
```

The farmer wants to increase the power of the test to 0.90. Calculate how many plants he must treat with each type of fertilizer.

```
n = sampsizepwr('t2', [1.4 0.2], 1.7, 0.9, [])
```

```
n =
```

```
11
```

To increase the power of the test to 0.90, the farmer must test 11 plants with each type of fertilizer.

The farmer wants to reduce the number of plants he must treat with Fertilizer B, but keep the power of the test at 0.90. but maintain the initial 2:1 ratio of plants in each treatment group

Using a 2:1 ratio of plants in each treatment group, calculate how many plants the farmer must test to obtain a power of 0.90. Use the mean and standard deviation values obtained in the previous test.

```
[n1out, n2out] = sampsizepwr('t2', [1.4, 0.2], 1.7, 0.9, [], 'Ratio', 2)
```

```
n1out =
```

```
8
```

```
n2out =
```



To obtain a power of 0.90, the farmer must treat 16 plants with Fertilizer A and 8 plants with Fertilizer B.

## Input Arguments

### **testtype** — Test type

'z' | 't' | 't2' | 'var' | 'p'

Test type, specified as one of the following.

- 'z' —  $z$ -test for normally distributed data with known standard deviation.
- 't' —  $t$ -test for normally distributed data with unknown standard deviation.
- 't2' — Two-sample pooled  $t$ -test for normally distributed data with unknown standard deviation and equal variances.
- 'var' — Chi-square test of variance for normally distributed data.
- 'p' — Test of the  $p$  parameter (success probability) for a binomial distribution. The 'p' test is a discrete test for which increasing the sample size does not always increase the power. For  $n$  values larger than 200, there may exist values smaller than the returned  $n$  value that also produce the specified power.

### **p0** — Parameter value under null hypothesis

scalar value | two-element array of scalar values

Parameter value under the null hypothesis, specified as a scalar value or a two-element array of scalar values.

- If `testtype` is 'z' or 't', then `p0` is a two-element array `[mu0, sigma0]` of the mean and standard deviation, respectively, under the null hypothesis.
- If `testtype` is 't2', then `p0` is a two-element array `[mu0, sigma0]` of the mean and standard deviation, respectively, of the first sample under the null and alternative hypotheses.
- If `testtype` is 'var', then `p0` is the variance under the null hypothesis.
- If `testtype` is 'p', then `p0` is the value of  $p$  under the null hypothesis.

Data Types: single | double

**p1 — Parameter value under alternative hypothesis**

scalar value | array of scalar values | []

Parameter value under the alternative hypothesis, specified as a scalar value or as an array of scalar values.

- If `testtype` is 'z' or 't', then `p1` is the value of the mean under the alternative hypothesis.
- If `testtype` is 't2', then `p1` is the value of the mean of the second sample under the alternative hypothesis.
- If `testtype` is 'var', then `p1` is the variance under the alternative hypothesis.
- If `testtype` is 'p', then `p1` is the value of  $p$  under the alternative hypothesis.

If you specify `p1` as an array, then `sampsizepwr` returns an array for `nout` or `pwrout` that is the same length as `p1`.

To return the alternative parameter value, `p1out`, specify `p1` using empty brackets ([]), as shown in the syntax description on page 32-0 .

Data Types: single | double

**pwr — Power of the test**

0.90 (default) | scalar value in the range (0,1) | array of scalar values in the range (0,1) | []

Power of the test, specified as a scalar value in the range (0,1) or as an array of scalar values in the range (0,1). The power of a test is the probability of rejecting the null hypothesis when the alternative hypothesis is true, given a particular significance level.

If you specify `pwr` as an array, then `sampsizepwr` returns an array for `nout` or `p1out` that is the same length as `pwr`.

To return a power value, `pwrout`, specify `pwr` using empty brackets ([]), as shown in the syntax description on page 32-0 .

Data Types: single | double

**n — Sample size**

positive integer value | array of positive integer values

Sample size, specified as a positive integer value or as an array of positive integer values.

If `testtype` is `'t2'`, then `samsizepwr` assumes that the two sample sizes are equal. For unequal sample sizes, specify `n` as the smaller of the two sample sizes, and use the `'Ratio'` name-value pair argument to indicate the sample size ratio. For example, if the smaller sample size is 5 and the larger sample size is 10, specify `n` as 5, and the `'Ratio'` name-value pair as 2.

If you specify `n` as an array, then `samsizepwr` returns an array for `pwrout` or `plout` that is the same length as `n`.

Data Types: `single` | `double`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'Alpha', 0.01, 'Tail', 'right'` specifies a right-tailed test with a 0.01 significance level.

### **Alpha** — Significance level

0.05 (default) | scalar value in the range (0,1)

Significance value of the test, specified as the comma-separated pair consisting of `'Alpha'` and a scalar value in the range (0,1).

Example: `'Alpha', 0.01`

Data Types: `single` | `double`

### **Ratio** — Sample size ratio

1 (default) | scalar value greater than or equal to 1

Sample size ratio for a two-sample *t*-test, specified as the comma-separated pair consisting of `'Ratio'` and a scalar value greater than or equal to 1. The value of `Ratio` is equal to  $n_2/n_1$ , where  $n_2$  is the larger sample size, and  $n_1$  is the smaller sample size.

To return the power, `pwrout`, or alternative parameter value, `plout`, specify the smaller of the two sample sizes for `n`, and use `'Ratio'` to indicate the sample size ratio.

Example: `'Ratio', 2`

**Tail — Test type**`'both' (default) | 'right' | 'left'`

Test type, specified as the comma-separated pair consisting of 'Tail' and one of the following:

- 'both' — Two-sided test for an alternative not equal to  $p_0$
- 'right' — One-sided test for an alternative larger than  $p_0$
- 'left' — One-sided test for an alternative smaller than  $p_0$

Example: 'Tail', 'right'

## Output Arguments

**nout — Sample size**`positive integer value | array of positive integer values`

Sample size, returned as a positive integer value or as an array of positive integer values.

If `testtype` is `t2`, and you use the 'Ratio' name-value pair argument to specify the ratio of the two unequal sample sizes, then `nout` returns the smaller of the two sample sizes.

Alternatively, to return both sample sizes, specify this argument as `[n1out, n2out]`. In this case, `sampsizepwr` returns the smaller sample size as `n1out`, and the larger sample size as `n2out`.

If you specify `pwr` or `p1` as an array, then `sampsizepwr` returns an array for `nout` that is the same length as `pwr` or `p1`.

**pwrout — Power**`scalar value in the range (0,1) | array of scalar values in the range (0,1)`

Power achieved by the test, returned as a scalar value in the range (0,1) or as an array of scalar values in the range (0,1).

If you specify `n` or `p1` as an array, then `sampsizepwr` returns an array for `pwrout` that is the same length as `n` or `p1`.

**p1out — Parameter value for the alternative hypothesis**

scalar value | array of scalar values

Parameter value for the alternative hypothesis, returned as a scalar value or as an array of scalar values.

When computing `p1out` for the 'p' test, if no alternative can be rejected for a given null hypothesis and significance level, the function displays a warning message and returns `NaN`.

**See Also**`binocdf` | `ttest` | `ttest2` | `vartest` | `ztest`**Introduced in R2006b**

## saveCompactModel

Save model object in file for code generation

To generate C code, you must have a MATLAB Coder license.

### Syntax

```
saveCompactModel(Mdl, filename)
```

### Description

`saveCompactModel(Mdl, filename)` prepares a compact classification model, compact regression model, or nearest neighbor searcher (`Mdl`) for code generation by reducing its memory footprint, and then saving it in the MATLAB formatted binary file (MAT-file) `filename`. You can pass `filename` to `loadCompactModel` for code generation.

### Examples

#### Save SVM Model for Code Generation

To generate C code that classifies new observations based on a trained SVM model, you must first save the trained model to disk. This example shows how to perform this first step.

Load the `ionosphere` data set.

```
load ionosphere
```

Train an SVM classification model using the entire data set. Specify to standardize the data.

```
Mdl = fitcsvm(X, Y, 'Standardize', true);
```

`Mdl` is a `ClassificationSVM` model.

Save the SVM classification model to the file 'SVMIonosphere.mat'.

```
saveCompactModel(Mdl, 'SVMIonosphere');
```

'SVMIonosphere.mat' appears in your present working directory. `saveCompactModel` reduces the memory footprint of the model by removing properties that are not needed for prediction, for example, the training data. Then, `saveCompactModel` saves a structure array that characterizes `Mdl` in 'SVMIonosphere.mat'.

In the function that you declare that classifies new data using the trained SVM model, load the structure array in 'SVMIonosphere.mat'.

## Generate Code From Function That Classifies New Data

Declare a function that classifies new observations using a trained classification model. This example requires a MATLAB® Coder™ license.

Load Fisher's iris data set.

```
load fisheriris
```

Train an ECOC model composed of SVM binary learners. Using an SVM template, standardize the data, hold out 30% of the data from training, and specify the Gaussian kernel.

```
t = templateSVM('Standardize',true,'KernelFunction','gaussian');  
CVMdl = fitcecoc(meas,species,'Learners',t,'Holdout',0.30);
```

`CVMdl` is a `ClassificationPartitionedECOC` model, which stores, among other things:

- A compact ECOC model, which `fitcecoc` trained using the training set.
- The data partition object, which specifies how `fitcecoc` randomly split the data into training and holdout sets.

Extract the compact ECOC classification model and the holdout set indices using dot notation. Identify the holdout observations.

```
CompactMdl = CVMdl.Trained{1};  
cvp = CVMdl.Partition;
```

```
idxTest = test(cvp);  
measHO = meas(idxTest,:);
```

CompactMdl is a CompactClassificationECOC model equipped to classify new observations.

Save the compact ECOC classification model to the file ECOCIris.mat.

```
saveCompactModel(CompactMdl, 'ECOCIris');
```

Declare a function in your current working folder called classifyIris.m that:

- Accepts iris-flower measurements commensurate with meas, and returns predicted labels
- Loads the compact ECOC classification model
- Passes the loaded model and iris-flower measurements to predict

```
function label = classifyIris(X) %#codegen  
%CLASSIFYIRISES Classify iris species using ECOC Model  
% CLASSIFYIRISES classifies the iris-flower measurements in X  
% using the compact ECOC model in the file ECOCIris.mat, and then  
% returns class labels in label.  
CompactMdl = loadCompactModel('ECOCIris');  
label = predict(CompactMdl, X);  
  
end
```

The %#codegen compilation directive indicates that the MATLAB® code is intended for code generation.

Generate a MEX function from classifyIris.m. Because C uses static typing, codegen must determine the properties of all variables in MATLAB® files at compile time. To ensure that the MEX function can use the same input, specify the holdout observations as arguments to the function using the '-args' option.

```
codegen classifyIris.m -args {measHO}
```

The MEX file classifyIris\_mex.mexw64 generates to your present working directory. The file extension depends on your platform.



Compare labels predicted using `predict` at the command line, `classifyIris.m`, and `classifyIris_mex`.

```
label1 = predict(CompactMdl,measHO);
label2 = classifyIris(meadHO);
label3 = classifyIris_mex(meadHO);

comp12 = cellfun(@strcmp,label1,label2);
comp23 = cellfun(@strcmp,label2,label3);
agree12 = sum(comp12) == numel(label1)
agree23 = sum(comp23) == numel(label1)

agree12 =

    logical

     1

agree23 =

    logical

     1
```

The labels predicted all three ways are the same.

- “C Code Generation and Classification Learner” on page 31-8
- “Predict Class Labels Using MATLAB Function Block” on page 31-18
- “C Code Generation for Image Classifier”

## Input Arguments

**mdl** — Compact classification model, compact regression model, or nearest neighbor searcher

full model object | compact model object

Compact classification model, compact regression model, or nearest neighbor searcher, specified as a full or compact model object, as given in the following tables of supported models.

- If `mdl` is a full classification or regression model object, then `saveCompactModel` saves its corresponding compact model object.

### Classification Model Object

Model	Problem	Full Model Object	Compact Model Object	Training Function
Classification tree	Multiclass classification	<code>ClassificationTree</code>	<code>CompactClassificationTree</code>	<code>fitctree</code>
Discriminant Analysis	Multiclass classification	<code>ClassificationDiscriminant</code>	<code>CompactClassificationDiscriminant</code>	<code>fitcdiscr</code>
Ensemble of classification trees	Multiclass classification	<code>ClassificationEnsemble</code> , <code>ClassificationBaggedEnsemble</code>	<code>CompactClassificationEnsemble</code>	<code>fitcensemble</code>
<i>k</i> NN	Multiclass classification	<code>ClassificationKNN</code>	N/A	<code>fitcknn</code>
Linear classification for high-dimensional data	Binary classification	N/A	<code>ClassificationLinear</code>	<code>fitclinear</code>
Logistic regression or SVM	Multiclass classification	<code>ClassificationECOC</code>	<code>CompactClassificationECOC</code>	<code>fitcecoc</code>
SVM	Binary classification	<code>ClassificationSVM</code>	<code>CompactClassificationSVM</code>	<code>fitsvm</code>

- A `ClassificationLinear` model object is a compact object that does not have a corresponding full object.
- A `ClassificationKNN` model object is a full object that does not have a corresponding compact object. `saveCompactModel` saves a compact version that does not include cross-validation parameters and hyperparameters.

**Regression Model Object**

Model	Full Model Object	Compact Model Object	Training Function
Ensemble of regression trees	RegressionEnsemble, RegressionBaggedEnsemble	CompactRegressionEnsemble	fitrensemble
Gaussian process regression	RegressionGP	CompactRegressionGP	fitrgp
Generalized linear model	GeneralizedLinearModel	CompactGeneralizedLinearModel	fitglm, stepwiseglm
Linear model	LinearModel	CompactLinearModel	fitlm, stepwiselm
Linear regression for high-dimensional data	N/A	RegressionLinear	fitrlinear
Regression tree	RegressionTree	CompactRegressionTree	fitrtree
SVM	RegressionSVM	CompactRegressionSVM	fitrsvm

- A `RegressionLinear` model object is a compact object that does not have a corresponding full object.
- If `Mdl` is a nearest neighbor searcher, then `saveCompactModel` saves the model as it is. A nearest neighbor searcher object is a full object that does not have a corresponding compact object.

**Nearest Neighbor Searcher Object**

Model	Model Object	Training Function
Exhaustive nearest neighbor searcher	ExhaustiveSearcher	ExhaustiveSearcher, createns

**filename** — Name of file  
character vector

Name of file, specified as a character vector. `filename` must be writable.

If `filename` exists, then `saveCompactModel` overwrites the file.

If `filename` has no extension (that is, no period followed by text), then `saveCompactModel` appends `.mat`. Otherwise, the extension must be `.mat`.

If `filename` does not include a full path, then `saveCompactModel` saves to the present working directory.

Example: `'SVMmdl'`

## Algorithms

`saveCompactModel` reduces the memory footprint of full classification or regression models (`Mdl`) by removing properties that are not required for prediction. That is, before saving the model, `saveCompactModel` applies the appropriate `compact` method to `Mdl`.

## See Also

`loadCompactModel`

## Topics

“C Code Generation and Classification Learner” on page 31-8

“Predict Class Labels Using MATLAB Function Block” on page 31-18

“C Code Generation for Image Classifier”

“Code Generation Workflows” on page 31-2

“Code Generation Support, Usage Notes, and Limitations” on page 31-24

**Introduced in R2016b**

# scatterhist

Scatter plot with marginal histograms

## Syntax

```
scatterhist(x,y)  
scatterhist(x,y,Name,Value)
```

```
h = scatterhist(____)
```

## Description

`scatterhist(x,y)` creates a 2-D scatter plot of the data in vectors `x` and `y`, and displays the marginal distributions of `x` and `y` as univariate histograms on the horizontal and vertical axes of the scatter plot, respectively.

`scatterhist(x,y,Name,Value)` creates the plot using additional options specified by one or more name-value pair arguments. For example, you can specify a grouping variable or change the display options.

`h = scatterhist(____)` returns a vector of three axis handles for the scatter plot, the histogram along the horizontal axis, and the histogram along the vertical axis, respectively, using any of the input arguments in the previous syntaxes.

## Examples

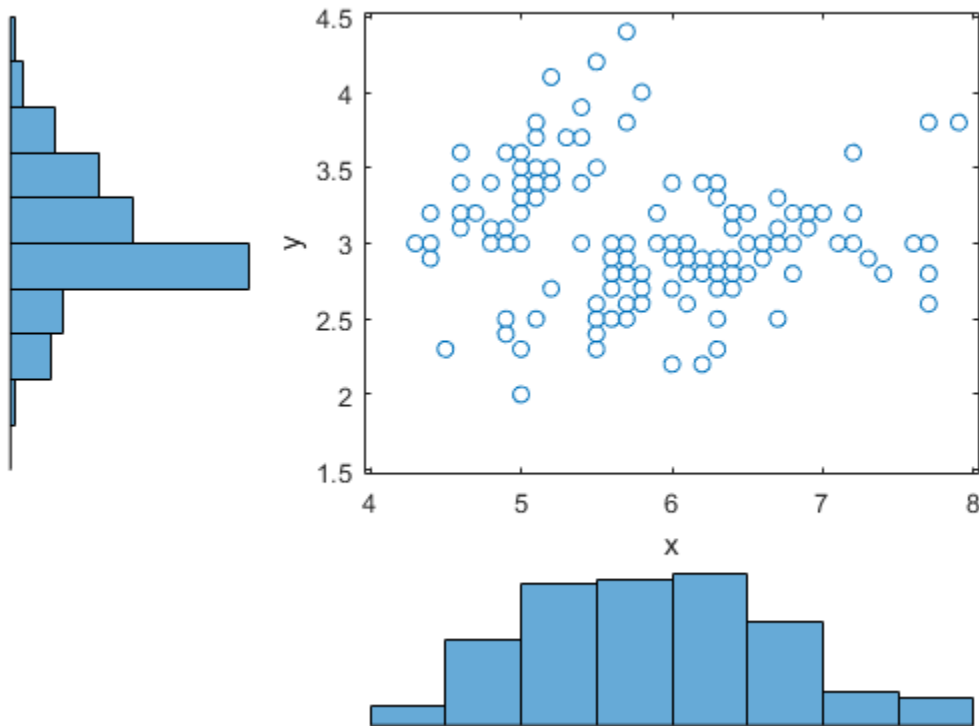
### Create a Scatterhist Plot


Load the sample data. Create data vector `x` from the first column of the data matrix, which contains sepal length measurements from iris flowers. Create data vector `y` from the second column of the data matrix, which contains sepal width measurements from the same flowers.

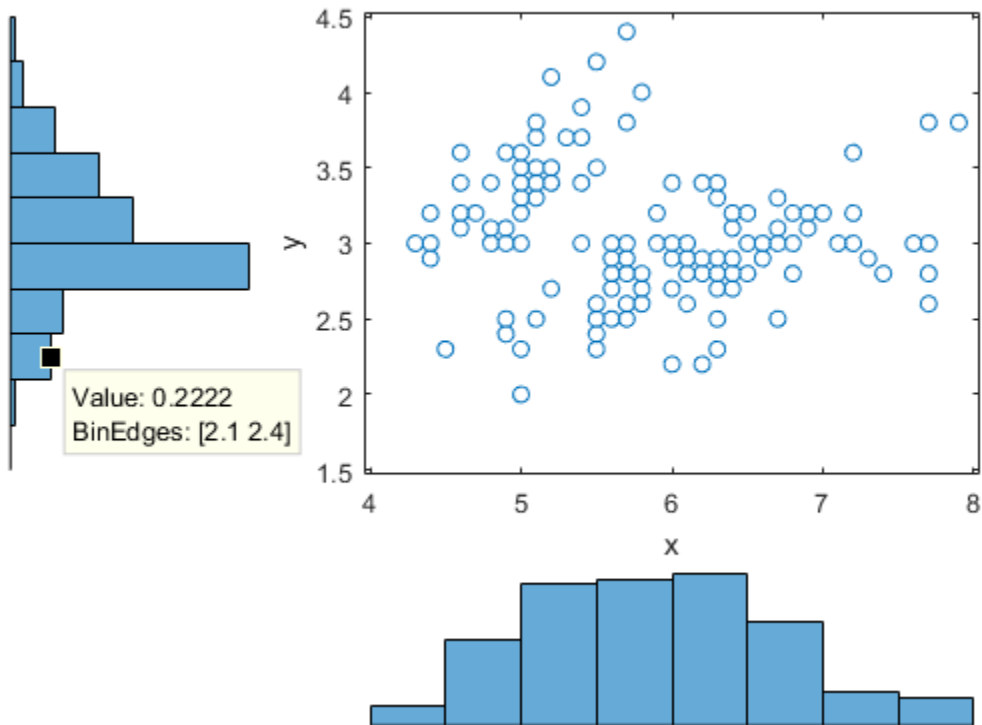
```
load fisheriris.mat;  
x = meas(:,1);  
y = meas(:,2);
```

Create a scatter plot and two marginal histograms to visualize the relationship between sepal length and sepal width.

```
scatterhist(x,y)
```



Display a data tip for a bin in a histogram. First, click the data cursor icon  on the figure toolbar or select **Tools > Data Cursor**. Then, click a bin in a histogram to select the bin and open its data tip.



The data tip displays the probability density function estimate of the selected bin and the lower and upper values for the bin edges.

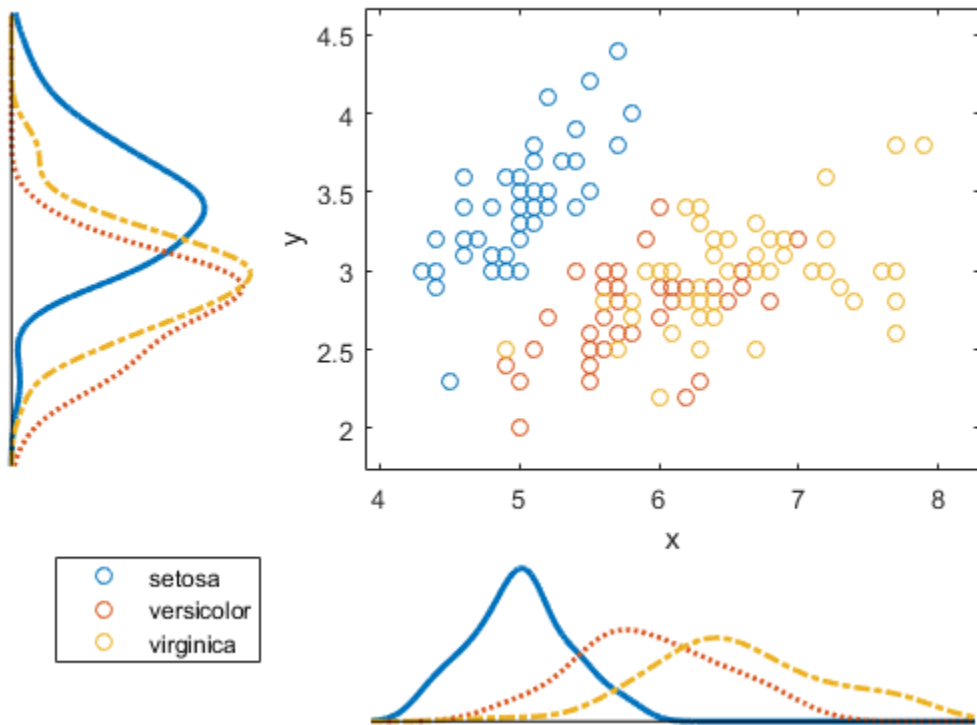
### Plot Grouped Data

Load the sample data. Create data vector  $x$  from the first column of the data matrix, which contains sepal length measurements from three species of iris flowers. Create data vector  $y$  from the second column of the data matrix, which contains sepal width measurements from the same flowers.

```
load fisheriris.mat;  
x = meas(:,1);  
y = meas(:,2);
```

Create a scatter plot and six kernel density plots to visualize the relationship between sepal length and sepal width, grouped by species.

```
scatterhist(x,y,'Group',species,'Kernel','on')
```



The plot shows that the relationship between sepal length and width varies depending on the flower species.



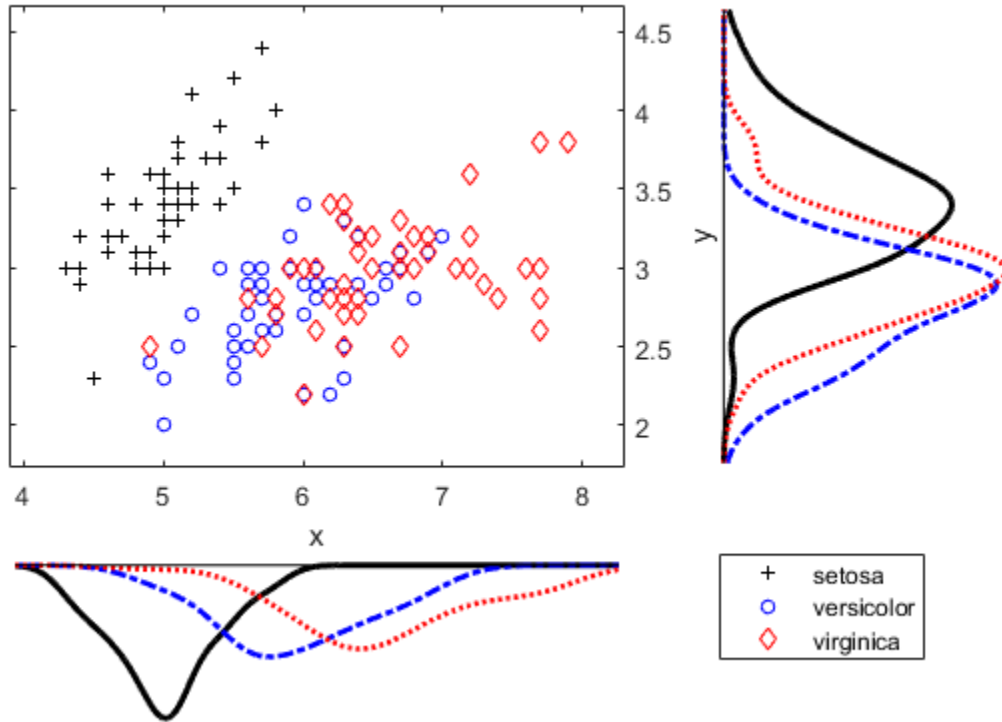
## Customize the Plot Display

Load the sample data. Create data vector  $x$  from the first column of the data matrix, which contains sepal length measurements from three different species of iris flowers. Create data vector  $y$  from the second column of the data matrix, which contains sepal width measurements from the same flowers.

```
load fisheriris.mat;  
x = meas(:,1);  
y = meas(:,2);
```

Create a scatter plot and six kernel density plots to visualize the relationship between sepal length and sepal width as measured on three species of iris flowers, grouped by species. Customize the appearance of the plots.

```
scatterhist(x,y,'Group',species,'Kernel','on','Location','SouthEast',...  
            'Direction','out','Color','kbr','LineStyle',{'-','-.'},...  
            'LineWidth',[2,2,2],'Marker','+od','MarkerSize',[4,5,6]);
```



### Customize Plots Using Axes Handles

Load the sample data. Create data vector `x` from the first column of the data matrix, which contains sepal length measurements from three species of iris flowers. Create data vector `y` from the second column of the data matrix, which contains sepal width measurements from the same flowers.

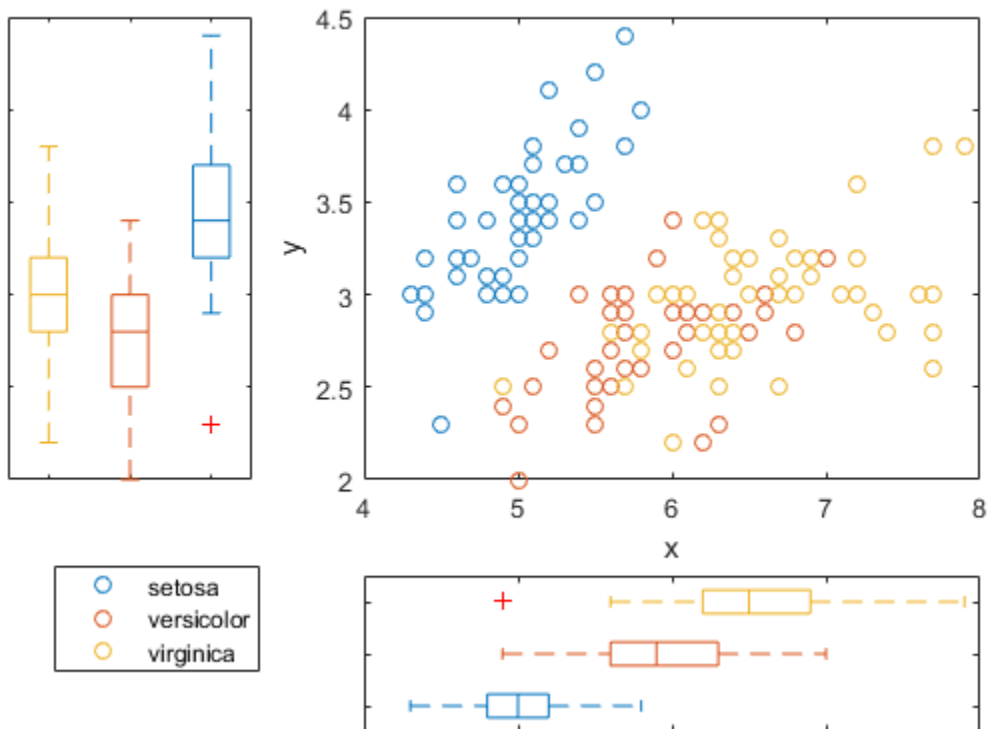
```
load fisheriris.mat;
x = meas(:,1);
y = meas(:,2);
```

Use axis handles to replace the marginal histograms with box plots.

```

h = scatterhist(x,y,'Group',species);
hold on;
clr = get(h(1),'colororder');
boxplot(h(2),x,species,'orientation','horizontal',...
        'label',{' ',' ',' '},'color',clr);
boxplot(h(3),y,species,'orientation','horizontal',...
        'label',{' ',' ',' '},'color',clr);
set(h(2:3),'XTickLabel','');
view(h(3),[270,90]); % Rotate the Y plot
axis(h(1),'auto'); % Sync axes
hold off;

```



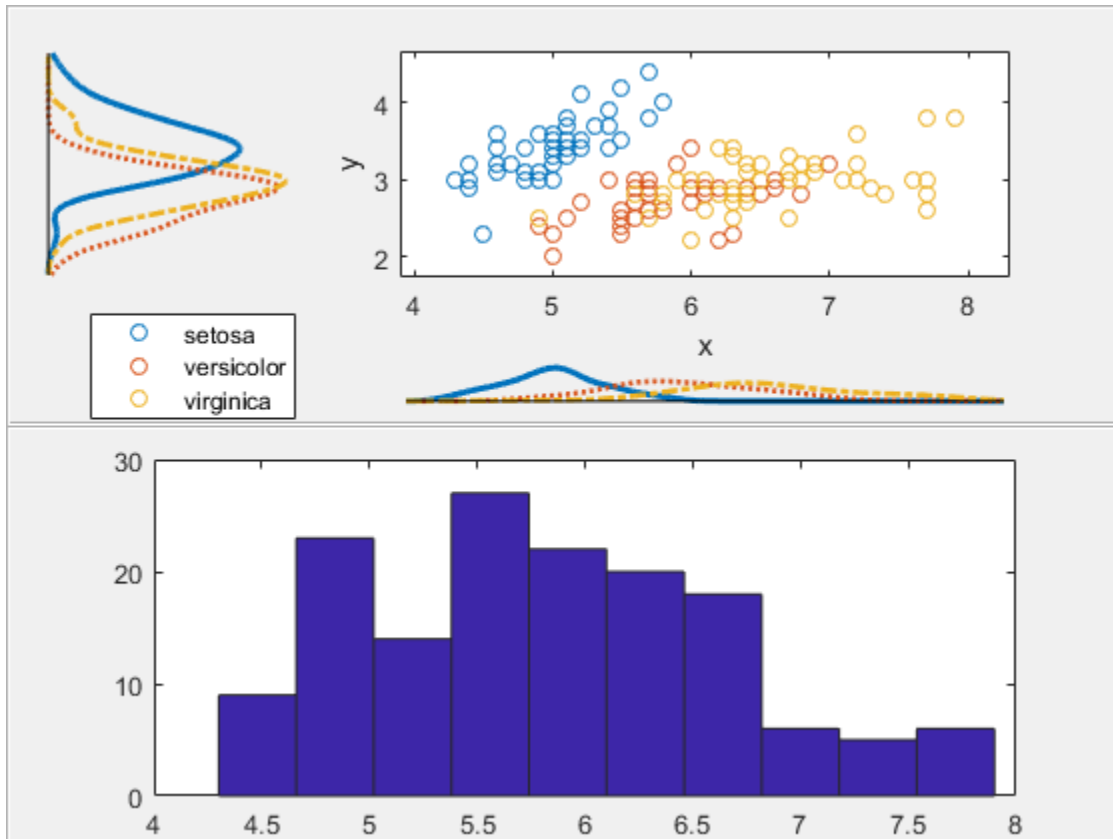
### Create a scatterhist Plot in a Specified Parent Container

Load the sample data. Create data vector `x` from the first column of the data matrix, which contains sepal length measurements from iris flowers. Create data vector `y` from the second column of the data matrix, which contains sepal width measurements from the same flowers.

```
load fisheriris
x = meas(:,1);
y = meas(:,2);
```

Create a new figure and define two uipanel objects to divide the figure into two parts. In the upper half of the figure, plot the sample data using `scatterhist`. Include marginal kernel density plots grouped by species. In the lower half of the figure, plot a histogram of the sepal length measurements contained in `x`.

```
figure
hp1 = uipanel('position',[0 .5 1 .5]);
hp2 = uipanel('position',[0 0 1 .5]);
scatterhist(x,y,'Group',species,'Kernel','on','Parent',hp1);
axes('Parent',hp2);
hist(x);
```



## Input Arguments

**x** — Sample data

vector

Sample data, specified as a vector. The data vectors  $x$  and  $y$  must be the same length.

If  $x$  or  $y$  contain NaN values, then `scatterhist`:

- Removes rows with NaN values in either  $x$  or  $y$  from both data vectors when generating the scatter plot

- Removes rows with NaN values only from the corresponding  $x$  or  $y$  data vector when generating the marginal histograms

Data Types: `single` | `double`

### **$y$ — Sample data**

vector

Sample data, specified as a vector. The data vectors  $x$  and  $y$  must be the same length.

If  $x$  or  $y$  contain NaN values, then `scatterhist`:

- Removes rows with NaN values in either  $x$  or  $y$  from both data vectors when generating the scatter plot
- Removes rows with NaN values only from the corresponding  $x$  or  $y$  data vector when generating the marginal histograms

Data Types: `single` | `double`

## **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'Location', 'SouthEast', 'Direction', 'out'` specifies a plot with histograms located below and to the right of the scatter plot, with the bars directed away from the scatter plot.

### **`NBins` — Number of bins for histograms**

positive integer value | vector

Number of bins for histograms, specified as the comma-separated pair consisting of `'NBins'` and a positive integer value greater than or equal to 2, or vector of two such values. If the number of bins is specified as a positive integer value, that value is the number of bins for both the  $x$  and  $y$  histograms. If the number of bins is specified by a vector, the first value is the number of bins for the  $x$  data, and the second value is the number of bins for the  $y$  data. By default, the number of bins is computed based on the sample standard deviation using Scott's rule.

Example: `'NBins', [5, 7]`

Data Types: `single` | `double`

### **Location — Location of marginal histograms**

`'SouthWest'` (default) | `'SouthEast'` | `'NorthEast'` | `'NorthWest'`

Location of the marginal histograms in the figure, specified as the comma-separated pair consisting of `'Location'` and one of the following.

`'SouthWest'` Plot the histograms below and to the left of the scatter plot.

`'SouthEast'` Plot the histograms below and to the right of the scatter plot.

`'NorthEast'` Plot the histograms above and to the right of the scatter plot.

`'NorthWest'` Plot the histograms above and to the left of the scatter plot.

Example: `'Location', 'SouthEast'`

### **Direction — Direction of marginal histograms**

`'in'` (default) | `'out'`

Direction of the marginal histograms, specified as the comma-separated pair consisting of `'Direction'` and one of the following.

`'in'` Plot the histograms with the bars directed toward the scatter plot.

`'out'` Plot the histograms with the bars directed away from the scatter plot.

Example: `'Direction', 'out'`

### **Group — Grouping variable**

categorical array | logical or numeric vector | cell array of character vectors

Grouping variable, specified as the comma-separated pair consisting of `'Group'` and a categorical array, logical or numeric vector, or cell array of character vectors. Each unique value in a grouping variable defines a group.

For example, if `Gender` is a cell array of character vectors with values `'Male'` and `'Female'`, you can use `Gender` as a grouping variable to plot your data by gender.

Multiple grouping variables can be used by specifying a cell array of grouping variable names. Observations are placed in the same group if they have common values of all specified grouping variables.

For example, if `Smoker` is a logical vector with values 0 for nonsmokers and 1 for smokers, then specifying the cell array `{Gender, Smoker}` divides observations into four groups: Male Smoker, Male Nonsmoker, Female Smoker, and Female Nonsmoker.

Example: `'Group', {Gender, Smoker}`

Data Types: `single | double | logical | cell | char`

### **PlotGroup — Grouped plot indicator**

`'on' | 'off'`

Grouped plot indicator, specified as the comma-separated pair consisting of `'PlotGroup'` and one of the following.

- `'on'`      Display grouped histograms or grouped kernel density plots. This is the default if a `Group` parameter is specified.
- `'off'`     Display histograms or kernel density plots of the whole data set. This is the default if a `Group` parameter is not specified.

Example: `'PlotGroup', 'off'`

### **style — Histogram display style**

`'stairs' | 'bar'`

Histogram display style, specified as the comma-separated pair consisting of `'PlotGroup'` and one of the following.

- `'stairs'`    Display a staircase plot that shows the outline of the histogram without filling the bars. This is the default if you specify a grouping variable that contains more than one group.
- `'bar'`      Display a histogram bar plot. This is the default if you specify a grouping variable that contains only one group or if `PlotGroup` is specified as `'off'`.

Example: `'Style', 'bar'`

### **kernel — Kernel density plot indicator**

`'off' (default) | 'on' | 'overlay'`



Kernel density plot indicator, specified as the comma-separated pair consisting of 'Kernel' and one of the following.

- 'off'        Display the marginal distributions as histograms.
- 'on'         Display the marginal distributions as kernel density plots.
- 'overlay'   Display the marginal distributions as kernel density plots overlaid onto histograms, similar to `histfit`.

Example: `'Kernel', 'overlay'`

#### **Bandwidth** — Bandwidth of kernel smoothing window matrix

Bandwidth of kernel smoothing window, specified as the comma-separated pair consisting of 'Bandwidth' and a matrix of size 2-by- $K$ , where  $K$  is the number of unique groups. The first row of the matrix gives the bandwidth of each group in  $x$ , and the second row gives the bandwidth of each group in  $y$ . By default, `scatterhist` finds the optimal bandwidth for estimating normal densities. Specifying a different bandwidth value changes the smoothing characteristics of the resulting kernel density plot. The value specified is a scaling factor for the normal distribution used to generate the kernel density plot.

Example: `'Bandwidth', [.5, .2, .1; .15, .25, .35]`

Data Types: `single` | `double`

#### **Legend** — Legend visibility indicator

'on' | 'off'

Legend visibility indicator, specified as the comma-separated pair consisting of 'Legend' and one of the following.

- 'on'         Set legend visible. This is the default if a `Group` parameter is specified.
- 'off'        Set legend invisible. This is the default if a `Group` parameter is not specified.

Example: `'Legend', 'on'`

#### **Parent** — Parent container of the plot

`ipanel` container object | `figure` container object

Parent container for the plot, specified as a `uipanel` container object or `figure` container object. You can create panel container objects using `uipanel` or `figure`, respectively.

For example, if `h1` is a panel container object, specify the parent container of the plot as follows.

Example: `'Parent', h1`

### **LineStyle** — Style of kernel density plot line

valid line style | cell array of line styles

Style of kernel density plot line, specified as the comma-separated pair consisting of `'LineStyle'` and a valid line style or a cell array of valid line styles. See `plot` for valid line styles. The default is a solid line. Use a cell array to specify different line styles for each group. When the total number of groups exceeds the number of specified values, `scatterhist` cycles through the specified values.

Example: `'LineStyle', {'-', ':', '-.'}`

### **LineWidth** — Width of kernel density plot line

0.5 (default) | nonnegative scalar value | vector

Width of kernel density plot line, specified as the comma-separated pair consisting of `'LineWidth'` and a nonnegative scalar value or vector of nonnegative scalar values. The specified value is the size of the kernel density plot line measured in points. The default size is 0.5 points. Use a vector to specify different line widths for each group. When the total number of groups is greater than the number of specified values, `scatterhist` cycles through the specified values.

Example: `'LineWidth', [0.5, 1, 2]`

Data Types: `single` | `double`

### **Color** — Marker color for each scatter plot group

character vector of color names | matrix of RGB values

Marker color for each scatter plot group, specified as the comma-separated pair consisting of `'Color'` and a character vector of color names, or a three-column matrix of RGB values in the range `[0,1]`. See `ColorSpec` for predefined colors and their RGB equivalents. If colors are specified using a matrix, each row of the matrix represents a group, and the three columns represent the R value, G value, and B value, respectively.

When the total number of groups exceeds the number of specified colors, `scatterhist` cycles through the specified colors.

Example: `'Color', 'kcm'`

Example: `'Color', [.5,0,1;0,.5,.5]`

Data Types: `single` | `double` | `char`

### **Marker** — Marker symbol for each scatterplot group

`'o'` (default) | character vector

Marker symbol for each scatter plot group, specified as the comma-separated pair consisting of `'Marker'` and a character vector of one or more valid marker symbols. See `plot` for valid symbols. The default is `'o'`, a circle. When the total number of groups exceeds the number of specified symbols, `scatterhist` cycles through the specified symbols.

Example: `'Marker', '+do'`

### **MarkerSize** — Marker size for each scatter plot group

6 (default) | nonnegative scalar value | vector

Marker size for each scatter plot group, specified as the comma-separated pair consisting of `'MarkerSize'` and a nonnegative scalar value or a vector of nonnegative scalar values, measured in points. When the total number of groups exceeds the number of specified values, `scatterhist` cycles through the specified values.

Example: `'MarkerSize', 10`

Data Types: `single` | `double`

## Output Arguments

### **h** — Axes handles

vector

Axes handles for the three plots, returned as a vector. The vector contains the handles for the scatter plot, the histogram along the horizontal axis, and the histogram along the vertical axis, respectively.

## See Also

`gscatter` | `histogram`

## Topics

“Grouping Variables” on page 2-59

**Introduced in R2007a**

# scramble

**Class:** grandset

Scramble quasi-random point set

## Syntax

```
ps = scramble(p, type)
ps = scramble(p, 'clear')
ps = scramble(p)
```

## Description

`ps = scramble(p, type)` returns a scrambled copy `ps` of the point set `p` of the `grandset` on page 32-5245 class, created using the scramble type specified in the character vector `type`. Point sets from different subclasses of `grandset` support different scramble types, as indicated in the following table.

Subclass	Scramble Types
haltonset on page 32-2987	'RR2' — A permutation of the radical inverse coefficients derived by applying a reverse-radix operation to all of the possible coefficient values. The scramble is described in [1].
sobolset on page 32-6138	'MatousekAffineOwen' — A random linear scramble combined with a random digital shift. The scramble is described in [2].

`ps = scramble(p, 'clear')` removes all scramble settings from `p` and returns the result in `ps`.

`ps = scramble(p)` removes all scramble settings from `p` and then adds them back in the order they were originally applied. This typically results in a different point set because of the randomness of the scrambling algorithms.

## Examples

Use `haltonset` to generate a 3-D Halton point set, skip the first 1000 values, and then retain every 101st point:

```
p = haltonset(3, 'Skip', 1e3, 'Leap', 1e2)
p =
    Halton point set in 3 dimensions (8.918019e+013 points)
    Properties:
        Skip : 1000
        Leap : 100
    ScrambleMethod : none
```

Use `scramble` to apply reverse-radix scrambling:

```
p = scramble(p, 'RR2')
p =
    Halton point set in 3 dimensions (8.918019e+013 points)
    Properties:
        Skip : 1000
        Leap : 100
    ScrambleMethod : RR2
```

Use `net` to generate the first four points:

```
X0 = net(p, 4)
X0 =
    0.0928    0.6950    0.0029
    0.6958    0.2958    0.8269
    0.3013    0.6497    0.4141
    0.9087    0.7883    0.2166
```

Use parenthesis indexing to generate every third point, up to the 11th point:

```
X = p(1:3:11, :)
X =
    0.0928    0.6950    0.0029
    0.9087    0.7883    0.2166
    0.3843    0.9840    0.9878
    0.6831    0.7357    0.7923
```

## References

- [1] Kocis, L., and W. J. Whiten. “Computational Investigations of Low-Discrepancy Sequences.” *ACM Transactions on Mathematical Software*. Vol. 23, No. 2, 1997, pp. 266–294.
- [2] Matousek, J. “On the L2-Discrepancy for Anchored Boxes.” *Journal of Complexity*. Vol. 14, No. 4, 1998, pp. 527–556.

## See Also

haltonset | sobolset

## ScrambleMethod property

**Class:** grandset

Settings that control scrambling

### Description

The `ScrambleMethod` property contains a structure that defines which scrambles to apply to the sequence. The structure consists of two fields:

- `Type`: A character vector containing the name of the scramble.
- `Options`: A cell array of parameter values for the scramble.

Different point sets support different scramble types as outlined in the help for each point set class. An error occurs if you set an invalid scramble type for a given point set.

The `ScrambleMethod` property also accepts an empty matrix as a value. This will clear all scrambling and set the property to contain a (0×0) structure.

The `scramble` method provides an alternative, easier way to set scrambles.

### Examples

Apply a random linear scramble combined with a random digital shift to a `sobolset` point set class:

```
P = sobolset(5);  
P = scramble(P, 'MatousekAffineOwen');  
P.ScrambleMethod
```

### See Also

`scramble` | `sobolset`



# segment

**Class:** `piecewisedistribution`

Segments containing values

## Syntax

```
S = segment(obj, X, P)
```

## Description

`S = segment(obj, X, P)` returns an array `S` of integers indicating which segment of the piecewise distribution object `obj` contains each value of `X` or, alternatively, `P`. One of `X` and `P` must be empty (`[]`). If `X` is nonempty, `S` is determined by comparing `X` with the quantile boundary values defined for `obj`. If `P` is nonempty, `S` is determined by comparing `P` with the probability boundary values.

## Examples

Fit Pareto tails to a  $t$  distribution at cumulative probabilities 0.1 and 0.9:

```
t = trnd(3,100,1);
obj = paretotails(t,0.1,0.9);

pvals = 0:0.2:1;
s = segment(obj,[],pvals)
s =
     1     2     2     2     2     3
```

## See Also

`boundary` | `nsegments` | `paretotails`

## selectModels

**Class:** ClassificationLinear

Choose subset of regularized, binary linear classification models

### Syntax

```
SubMdl = selectModels(Mdl, idx)
```

### Description

`SubMdl = selectModels(Mdl, idx)` returns a subset of trained, binary linear classification models from a set of binary linear classification models (`Mdl`) trained using various regularization strengths. The indices (`idx`) correspond to the regularization strengths in `Mdl.Lambda`, and specify which models to return.

### Input Arguments

**mdl** — Binary linear classification models trained using various regularization strengths

ClassificationLinear model object

Binary linear classification models trained using various regularization strengths, specified as a ClassificationLinear model object. You can create a ClassificationLinear model object using `fitlinear`.

Although `Mdl` is one model object, if `numel(Mdl.Lambda) = L ≥ 2`, then you can think of `Mdl` as  $L$  trained models.

**idx** — Indices corresponding to regularization strengths

numeric vector of positive integers

Indices corresponding to regularization strengths, specified as a numeric vector of positive integers. Values of `idx` must be in the interval  $[1, L]$ , where  $L = \text{numel}(Mdl.Lambda)$ .

Data Types: `double` | `single`

## Output Arguments

**subMdl** — Subset of binary linear classification models trained using various regularization strengths

ClassificationLinear model object

Subset of binary linear classification models trained using various regularization strengths, returned as a ClassificationLinear model object.

## Examples

### Find Good Lasso Penalty Using Classification Loss

To determine a good lasso-penalty strength for a linear classification model that uses a logistic regression learner, compare test-sample classification error rates.

Load the NLP data set. Preprocess the data as in “Specify Custom Classification Loss” on page 32-3663.

```
load nlpdata
Ystats = Y == 'stats';
X = X';

rng(10); % For reproducibility
Partition = cvpartition(Ystats,'Holdout',0.30);
testIdx = test(Partition);
XTest = X(:,testIdx);
YTest = Ystats(testIdx);
```

Create a set of 11 logarithmically-spaced regularization strengths from  $10^{-6}$  through  $10^{-0.5}$ .

```
Lambda = logspace(-6,-0.5,11);
```

Train binary, linear classification models that use each of the regularization strengths. Optimize the objective function using SpARSA. Lower the tolerance on the gradient of the objective function to  $1e-8$ .

```
CVMdl = fitclinear(X,Ystats,'ObservationsIn','columns',...  
    'CVPartition',Partition,'Learner','logistic','Solver','sparsa',...  
    'Regularization','lasso','Lambda',Lambda,'GradientTolerance',1e-8)
```

```
CVMdl =
```

```
classreg.learning.partition.ClassificationPartitionedLinear  
  CrossValidatedModel: 'Linear'  
    ResponseName: 'Y'  
  NumObservations: 31572  
        KFold: 1  
    Partition: [1x1 cvpartition]  
    ClassNames: [0 1]  
    ScoreTransform: 'none'
```

Extract the trained linear classification model.

```
Mdl = CVMdl.Trained{1}
```

```
Mdl =
```

```
ClassificationLinear  
  ResponseName: 'Y'  
    ClassNames: [0 1]  
  ScoreTransform: 'logit'  
        Beta: [34023x11 double]  
        Bias: [1x11 double]  
        Lambda: [1x11 double]  
    Learner: 'logistic'
```

Mdl is a `ClassificationLinear` model object. Because `Lambda` is a sequence of regularization strengths, you can think of Mdl as 11 models, one for each regularization strength in `Lambda`.

Estimate the test-sample classification error.

```
ce = loss(Mdl,X(:,testIdx),Ystats(testIdx),'ObservationsIn','columns');
```

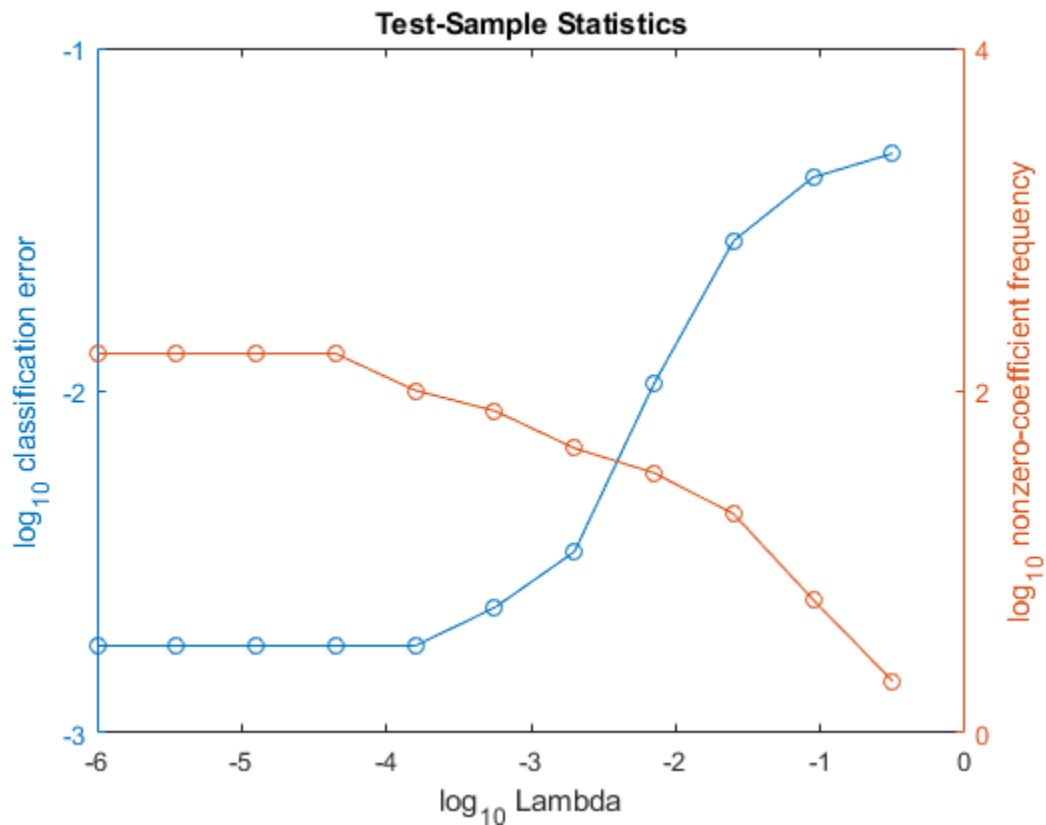
Because there are 11 regularization strengths, `ce` is a 1-by-11 vector of classification error rates.

Higher values of `Lambda` lead to predictor variable sparsity, which is a good quality of a classifier. For each regularization strength, train a linear classification model using the entire data set and the same options as when you cross-validated the models. Determine the number of nonzero coefficients per model.

```
Mdl = fitclinear(X,Ystats,'ObservationsIn','columns',...  
    'Learner','logistic','Solver','sparsa','Regularization','lasso',...  
    'Lambda',Lambda,'GradientTolerance',1e-8);  
numNZCoeff = sum(Mdl.Beta~=0);
```

In the same figure, plot the test-sample error rates and frequency of nonzero coefficients for each regularization strength. Plot all variables on the log scale.

```
figure;  
[h,hL1,hL2] = plotyy(log10(Lambda),log10(ce),...  
    log10(Lambda),log10(numNZCoeff + 1));  
hL1.Marker = 'o';  
hL2.Marker = 'o';  
ylabel(h(1),'log_{10} classification error')  
ylabel(h(2),'log_{10} nonzero-coefficient frequency')  
xlabel('log_{10} Lambda')  
title('Test-Sample Statistics')  
hold off
```



Choose the index of the regularization strength that balances predictor variable sparsity and low classification error. In this case, a value between  $10^{-4}$  to  $10^{-1}$  should suffice.

```
idxFinal = 7;
```

Select the model from `Mdl` with the chosen regularization strength.

```
MdlFinal = selectModels(Mdl,idxFinal);
```

`MdlFinal` is a `ClassificationLinear` model containing one regularization strength. To estimate labels for new observations, pass `MdlFinal` and the new data to `predict`.

## Tip

One way to build several predictive, binary linear classification models is:

- 1 Hold out a portion of the data for testing.
- 2 Train a binary, linear classification model using `fitcllinear`. Specify a grid of regularization strengths using the 'Lambda' name-value pair argument and supply the training data. `fitcllinear` returns one `ClassificationLinear` model object, but it contains a model for each regularization strength.
- 3 To determine the quality of each regularized model, pass the returned model object and the held-out data to, for example, `loss`.
- 4 Identify the indices (`idx`) of a satisfactory subset of regularized models, and then pass the returned model and the indices to `selectModels`. `selectModels` returns one `ClassificationLinear` model object, but it contains `numel(idx)` regularized models.
- 5 To predict class labels for new data, pass the data and the subset of regularized models to `predict`.

## See Also

`ClassificationLinear` | `fitcllinear` | `loss` | `predict`

Introduced in R2016a

## selectModels

**Class:** CompactClassificationECOC

Choose subset of regularized linear classification models

### Syntax

```
SubMdl = selectModels(Mdl, idx)
```

### Description

`SubMdl = selectModels(Mdl, idx)` returns a subset of trained ECOC models composed of binary linear classification models from a set of multiclass ECOC models (`Mdl`) trained using various regularization strengths. The indices (`idx`) correspond to the regularization strengths in `Mdl.BinaryLearners{1}.Lambda`, and specify which models to return.

### Input Arguments

**mdl** — Multiclass ECOC model composed of linear classification models, trained using various regularization strengths

CompactClassificationECOC model object

Multiclass ECOC model composed of linear classification models, trained using various regularization strengths, specified as a CompactClassificationECOC model object.

When creating `Mdl`, you must:

- Use `fitcecoc`
- Specify linear-classification-model binary learners (see `Learners`)
- Specify the same regularization strengths for each linear-classification-model binary learner



Although `Mdl` is one model object, if `numel(Mdl.BinaryLearners{1}.Lambda) = L ≥ 2`, then you can think of `Mdl` as  $L$  trained models.

**`idx` — Indices corresponding to regularization strengths**

numeric vector of positive integers

Indices corresponding to regularization strengths, specified as a numeric vector of positive integers. Values of `idx` must be in the interval  $[1, L]$ , where  $L = \text{numel}(\text{Mdl.BinaryLearners}\{1\}.\text{Lambda})$ .

Data Types: `double` | `single`

## Output Arguments

**`SubMdl` — Subset of ECOC models composed of binary linear classification models, trained using various regularization strengths**

`CompactClassificationECOC` model object

Subset of ECOC models composed of binary linear classification models, trained using various regularization strengths, returned as a `CompactClassificationECOC` model object.

## Examples

### Select Best Regularized Models

Load the NLP data set.

```
load nlpdata
```

`X` is a sparse matrix of predictor data, and `Y` is a categorical vector of class labels.

Create a set of 11 logarithmically-spaced regularization strengths from  $10^{-8}$  through  $10^{-1}$ .

```
Lambda = logspace(-8, -1, 11);
```

Create a linear-classification-model template that specifies optimizing the objective function using `Sparse`, and use lasso penalties with the strengths specified in `Lambda`.

```
t = templateLinear('Solver','sparsa','Regularization','lasso',...
    'Lambda',Lambda);
```

Hold out 30% of the data for testing. Identify the test-sample indices.

```
rng(1); % For reproducibility
cvp = cvpartition(Y,'Holdout',0.30);
idxTest = test(cvp);
```

Train an ECOC model composed of linear classification models. For quicker execution time, orient the predictor data so that individual observations correspond to columns.

```
X = X';
CVMdl = fitcecoc(X,Y,'Learners',t,'ObservationsIn','columns','CVPartition',cvp);
Mdl = CVMdl.Trained{1};
numel(Mdl.BinaryLearners{1}.Lambda)
```

```
ans =
```

```
11
```

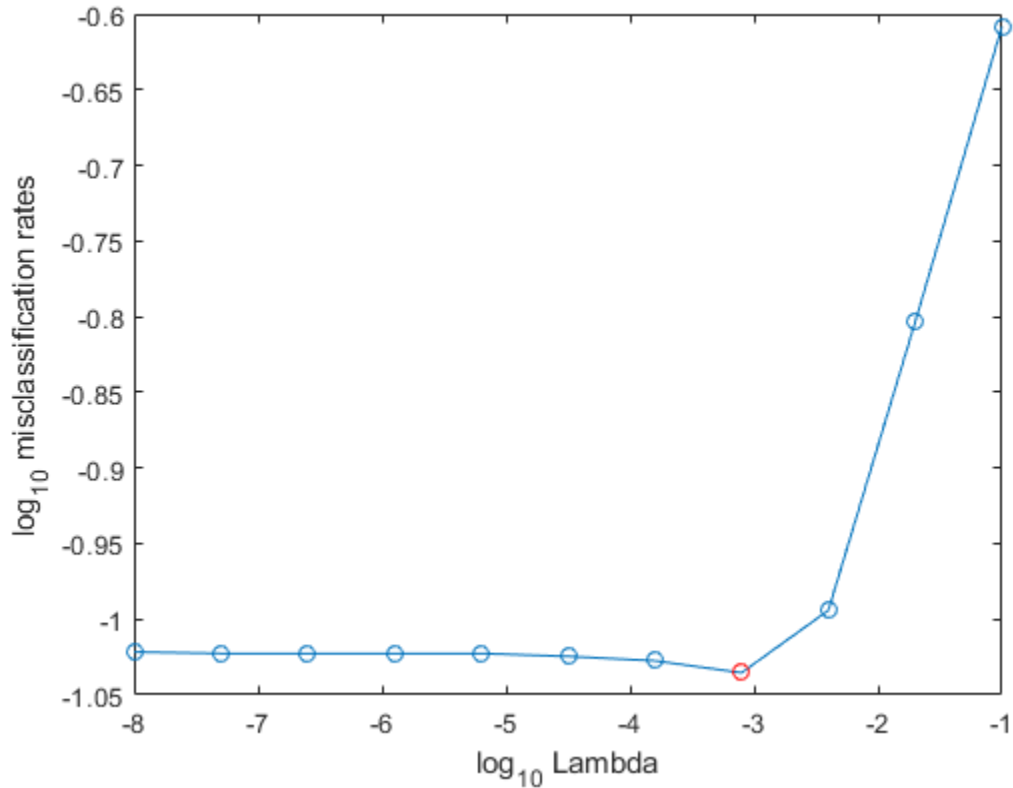
Mdl is a CompactClassificationECOC model object. Because Lambda is an 11-dimensional vector of regularization strengths, you can think of Mdl as eleven trained models, one for each regularization strength.

Estimate the test-sample misclassification rates for each regularized model.

```
ce = loss(Mdl,X(:,idxTest),Y(idxTest),'ObservationsIn','columns');
```

Plot the misclassification rates with respect to regularization strength on the log scale.

```
figure;
plot(log10(Lambda),log10(ce),'-o')
ylabel('log_{10} misclassification rates')
xlabel('log_{10} Lambda')
[~,minCEIdx] = min(ce);
minLambda = Lambda(minCEIdx);
hold on
plot(log10(minLambda),log10(ce(minCEIdx)),'ro');
hold off
```



Several values of  $\text{Lambda}$  yield similarly small classification error values. Higher values of  $\text{lambda}$  lead to predictor variable sparsity, which is a good quality of a classifier.

Select the four models with the regularization strengths that occur around where the classification error starts increasing.

```
idx = 7:10;
MdlFinal = selectModels(Mdl,idx)
LambdaFinal = MdlFinal.BinaryLearners{1}.Lambda

MdlFinal =

classreg.learning.classif.CompactClassificationECOC
```

```
ResponseName: 'Y'  
ClassNames: [1x13 categorical]  
ScoreTransform: 'none'  
BinaryLearners: {78x1 cell}  
CodingMatrix: [13x78 double]
```

```
LambdaFinal =  
  
    0.0002    0.0008    0.0040    0.0200
```

`MdlFinal` is a `CompactClassificationECOC` model object. You can think of it as three models trained using the four regularization strengths in `LambdaFinal`.

## Tip

One way to build several predictive ECOC models composed of binary linear classification models is:

- 1 Create a binary-linear-classification-model template and specify a grid of regularization strengths using the 'Lambda' name-value pair argument.
- 2 Hold out a portion of the data for testing.
- 3 Train an ECOC model using `fitcecoc`. Specify the template using the 'Learners' name-value pair argument and supply the training data. `fitcecoc` returns one `CompactClassificationECOC` model object containing `ClassificationLinear` binary learners, but all binary learners contain a model for each regularization strength.
- 4 To determine the quality of each regularized model, pass the returned model object and the held-out data to, for example, `loss`.
- 5 Identify the indices (`idx`) of a satisfactory subset of regularized models, and then pass the returned model and the indices to `selectModels`. `selectModels` returns one `CompactClassificationECOC` model object, but it contains `numel(idx)` regularized models.
- 6 To predict class labels for new data, pass the data and the subset of regularized models to `predict`.

## See Also

`ClassificationLinear` | `ClassificationLinearECOC` | `fitcecoc` | `loss` |  
`predict` | `templateLinear`

**Introduced in R2016a**

## selectModels

**Class:** RegressionLinear

Select fitted regularized linear regression models

## Syntax

```
SubMdl = selectModels(Mdl, idx)
```

## Description

`SubMdl = selectModels(Mdl, idx)` returns a subset of trained linear regression models from a set of linear regression models (`Mdl`) trained using various regularization strengths. The indices `idx` correspond to the regularization strengths in `Mdl.Lambda`, and specify which models to return.

## Input Arguments

**`mdl` — Linear regression models trained using various regularization strengths**

RegressionLinear model object

Linear regression models trained using various regularization strengths, specified as a RegressionLinear model object. You can create a RegressionLinear model object using `fitrlinear`.

Although `Mdl` is one model object, if `numel(Mdl.Lambda) = L ≥ 2`, then you can think of `Mdl` as  $L$  trained models.

**`idx` — Indices corresponding to regularization strengths**

numeric vector of positive integers

Indices corresponding to regularization strengths, specified as a numeric vector of positive integers. Values of `idx` must be in the interval  $[1, L]$ , where  $L = \text{numel}(Mdl.Lambda)$ .

Data Types: `double` | `single`

## Output Arguments

**submdl** — Subset of linear regression models trained using various regularization strengths  
`RegressionLinear` model object

Subset of linear regression models trained using various regularization strengths, returned as a `RegressionLinear` model object.

## Examples

### Find Good Lasso Penalty Using Regression Loss

Simulate 10000 observations from this model

$$y = x_{100} + 2x_{200} + e.$$

- $X = \{x_1, \dots, x_{1000}\}$  is a 10000-by-1000 sparse matrix with 10% nonzero standard normal elements.
- $e$  is random normal error with mean 0 and standard deviation 0.3.

```
rng(1) % For reproducibility
n = 1e4;
d = 1e3;
nz = 0.1;
X = sprandn(n,d,nz);
Y = X(:,100) + 2*X(:,200) + 0.3*randn(n,1);
```

Create a set of 15 logarithmically-spaced regularization strengths from  $10^{-4}$  through  $10^{-1}$ .

```
Lambda = logspace(-4,-1,15);
```

Hold out 30% of the data for testing. Identify the test-sample indices.

```
cvp = cvpartition(numel(Y), 'Holdout', 0.30);
idxTest = test(cvp);
```

Train a linear regression model using lasso penalties with the strengths in `Lambda`. Specify the regularization strengths, optimizing the objective function using `SpaRSA`, and the data partition. To increase execution speed, transpose the predictor data and specify that the observations are in columns.

```
X = X';
CVMdl = fitrlinear(X,Y,'ObservationsIn','columns','Lambda',Lambda,...
    'Solver','sparsa','Regularization','lasso','CVPartition',cvp);
Mdl1 = CVMdl.Trained{1};
numel(Mdl1.Lambda)
```

```
ans =
```

```
15
```

`Mdl1` is a `RegressionLinear` model. Because `Lambda` is a 15-dimensional vector of regularization strengths, you can think of `Mdl1` as 15 trained models, one for each regularization strength.

Estimate the test-sample mean squared error for each regularized model.

```
mse = loss(Mdl1,X(:,idxTest),Y(idxTest),'ObservationsIn','columns');
```

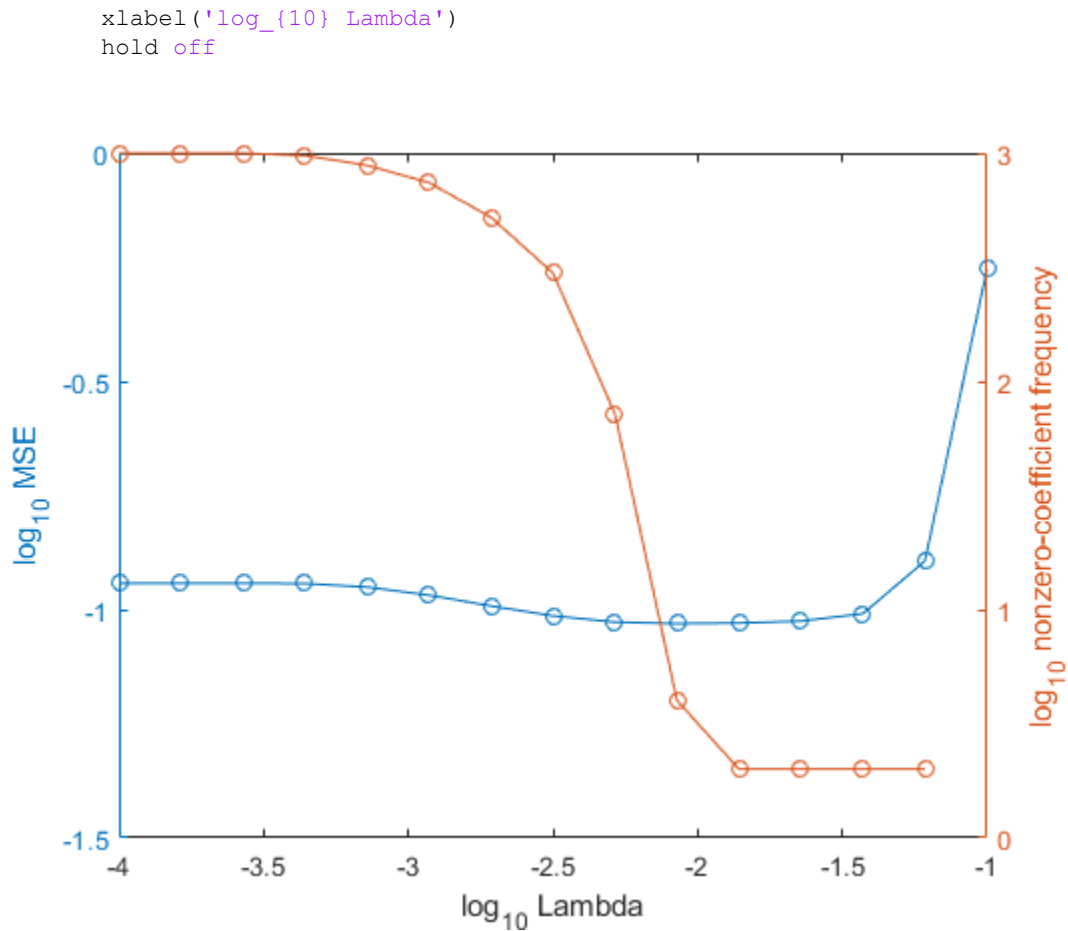
Higher values of `Lambda` lead to predictor variable sparsity, which is a good quality of a regression model. Retrain the model using the entire data set and all options used previously, except the data-partition specification. Determine the number of nonzero coefficients per model.

```
Mdl = fitrlinear(X,Y,'ObservationsIn','columns','Lambda',Lambda,...
    'Solver','sparsa','Regularization','lasso');
numNZCoeff = sum(Mdl.Beta~=0);
```

In the same figure, plot the MSE and frequency of nonzero coefficients for each regularization strength. Plot all variables on the log scale.

```
figure;
[h,hL1,hL2] = plotyy(log10(Lambda),log10(mse),...
    log10(Lambda),log10(numNZCoeff));
hL1.Marker = 'o';
hL2.Marker = 'o';
ylabel(h(1),'log_{10} MSE')
ylabel(h(2),'log_{10} nonzero-coefficient frequency')
```





Select the index or indices of  $\text{Lambda}$  that balance minimal classification error and predictor-variable sparsity (for example,  $\text{Lambda}(11)$ ).

```
idx = 11;  
MdlFinal = selectModels(Mdl,idx);
```

`MdlFinal` is a trained `RegressionLinear` model object that uses `Lambda(11)` as a regularization strength.

## Tip

One way to build several predictive linear regression models is:

- 1 Hold out a portion of the data for testing.
- 2 Train a linear regression model using `fitrlinear`. Specify a grid of regularization strengths using the 'Lambda' name-value pair argument and supply the training data. `fitrlinear` returns one `RegressionLinear` model object, but it contains a model for each regularization strength.
- 3 To determine the quality of each regularized model, pass the returned model object and the held-out data to, for example, `loss`.
- 4 Identify the indices (`idx`) of a satisfactory subset of regularized models, and then pass the returned model and the indices to `selectModels`. `selectModels` returns one `RegressionLinear` model object, but it contains `numel(idx)` regularized models.
- 5 To predict class labels for new data, pass the data and the subset of regularized models to `predict`.

## See Also

`RegressionLinear` | `fitrlinear` | `loss` | `predict`

**Introduced in R2016a**

# sequentialfs

Sequential feature selection

## Syntax

```
inmodel = sequentialfs(fun,X,y)
inmodel = sequentialfs(fun,X,Y,Z,...)
[inmodel,history] = sequentialfs(fun,X,...)
[] = sequentialfs(...,param1, val1,param2, val2,...)
```

## Description

`inmodel = sequentialfs(fun,X,y)` selects a subset of features from the data matrix `X` that best predict the data in `y` by sequentially selecting features until there is no improvement in prediction. Rows of `X` correspond to observations; columns correspond to variables or features. `y` is a column vector of response values or class labels for each observation in `X`. `X` and `y` must have the same number of rows. `fun` is a function handle to a function that defines the criterion used to select features and to determine when to stop. The output `inmodel` is a logical vector indicating which features are finally chosen.

Starting from an empty feature set, `sequentialfs` creates candidate feature subsets by sequentially adding each of the features not yet selected. For each candidate feature subset, `sequentialfs` performs 10-fold cross-validation by repeatedly calling `fun` with different training subsets of `X` and `y`, `XTRAIN` and `ytrain`, and test subsets of `X` and `y`, `XTEST` and `ytest`, as follows:

```
criterion = fun(XTRAIN,ytrain,XTEST,ytest)
```

`XTRAIN` and `ytrain` contain the same subset of rows of `X` and `Y`, while `XTEST` and `ytest` contain the complementary subset of rows. `XTRAIN` and `XTEST` contain the data taken from the columns of `X` that correspond to the current candidate feature set.

Each time it is called, `fun` must return a scalar value `criterion`. Typically, `fun` uses `XTRAIN` and `ytrain` to train or fit a model, then predicts values for `XTEST` using that model, and finally returns some measure of distance, or loss, of those predicted values

from `ytest`. In the cross-validation calculation for a given candidate feature set, `sequentialfs` sums the values returned by `fun` and divides that sum by the total number of test observations. It then uses that mean value to evaluate each candidate feature subset.

Typical loss measures include sum of squared errors for regression models (`sequentialfs` computes the mean-squared error in this case), and the number of misclassified observations for classification models (`sequentialfs` computes the misclassification rate in this case).

---

**Note** `sequentialfs` divides the sum of the values returned by `fun` across all test sets by the total number of test observations. Accordingly, `fun` should not divide its output value by the number of test observations.

---

After computing the mean `criterion` values for each candidate feature subset, `sequentialfs` chooses the candidate feature subset that minimizes the mean `criterion` value. This process continues until adding more features does not decrease the `criterion`.

`inmodel = sequentialfs(fun,X,Y,Z,...)` allows any number of input variables `X`, `Y`, `Z`, ... `sequentialfs` chooses features (columns) only from `X`, but otherwise imposes no interpretation on `X`, `Y`, `Z`, ... All data inputs, whether column vectors or matrices, must have the same number of rows. `sequentialfs` calls `fun` with training and test subsets of `X`, `Y`, `Z`, ... as follows:

```
criterion = fun(XTRAIN,YTRAIN,ZTRAIN,...,  
              XTEST,YTEST,ZTEST,...)
```

`sequentialfs` creates `XTRAIN`, `YTRAIN`, `ZTRAIN`, ... , `XTEST`, `YTEST`, `ZTEST`, ... by selecting subsets of the rows of `X`, `Y`, `Z`, ... `fun` must return a scalar value `criterion`, but may compute that value in any way. Elements of the logical vector `inmodel` correspond to columns of `X` and indicate which features are finally chosen.

`[inmodel,history] = sequentialfs(fun,X,...)` returns information on which feature is chosen at each step. `history` is a scalar structure with the following fields:

- `Crit` — A vector containing the `criterion` values computed at each step.
- `In` — A logical matrix in which row `i` indicates the features selected at step `i`.

[ ] = sequentialfs(..., param1, val1, param2, val2, ...) specifies optional parameter name/value pairs from the following table.

Parameter	Value
'cv'	<p>The validation method used to compute the criterion for each candidate feature subset.</p> <ul style="list-style-type: none"> <li>• When the value is a positive integer <i>k</i>, sequentialfs uses <i>k</i>-fold cross-validation without stratification.</li> <li>• When the value is an object of the <code>cvpartition</code> on page 32-1292 class, other forms of cross-validation can be specified.</li> <li>• When the value is 'resubstitution', the original data are passed to <code>fun</code> as both the training and test data to compute the criterion.</li> <li>• When the value is 'none', sequentialfs calls <code>fun</code> as <code>criterion = fun(X, Y, Z, ...)</code>, without separating test and training sets.</li> </ul> <p>The default value is 10, that is, 10-fold cross-validation without stratification.</p> <p>So-called wrapper methods use a function <code>fun</code> that implements a learning algorithm. These methods usually apply cross-validation to select features. So-called filter methods use a function <code>fun</code> that measures characteristics of the data (such as correlation) to select features.</p>
'mcreps'	<p>A positive integer indicating the number of Monte-Carlo repetitions for cross-validation. The default value is 1. The value must be 1 if the value of 'cv' is 'resubstitution' or 'none'.</p>
'direction'	<p>The direction of the sequential search. The default is 'forward'. A value of 'backward' specifies an initial candidate set including all features and an algorithm that removes features sequentially until the criterion increases.</p>
'keepin'	<p>A logical vector or a vector of column numbers specifying features that must be included. The default is empty.</p>

Parameter	Value
'keepout'	A logical vector or a vector of column numbers specifying features that must be excluded. The default is empty.
'nfeatures'	The number of features at which <code>sequentialfs</code> should stop. <code>inmodel</code> includes exactly this many features. The default value is empty, indicating that <code>sequentialfs</code> should stop when a local minimum of the criterion is found. A nonempty value overrides values of 'MaxIter' and 'TolFun' in 'options'.
'nullmodel'	A logical value, indicating whether or not the null model (containing no features from X) should be included in feature selection and in the <code>history</code> output. The default is <code>false</code> .
'options'	Options structure for the iterative sequential search algorithm, as created by <code>statset.sequentialfs</code> uses the following <code>statset</code> parameters: <ul style="list-style-type: none"> <li>• <code>Display</code> — Amount of information displayed by the algorithm. The default is 'off'.</li> <li>• <code>MaxIter</code> — Maximum number of iterations allowed. The default is <code>Inf</code>.</li> <li>• <code>TolFun</code> — Termination tolerance for the objective function value. The default is <code>1e-6</code> if 'direction' is 'forward'; <code>0</code> if 'direction' is 'backward'.</li> <li>• <code>TolTypeFun</code> — Use absolute or relative objective function tolerances. The default is 'rel'.</li> <li>• <code>UseParallel</code> — Set to <code>true</code> to compute in parallel. Default is <code>false</code>.</li> <li>• <code>UseSubstreams</code> — Set to <code>true</code> to compute in parallel in a reproducible fashion. Default is <code>false</code>. To compute reproducibly, set <code>Streams</code> to a type allowing substreams: 'mlfg6331_64' or 'mrg32k3a'.</li> <li>• <code>Streams</code> — A <code>RandStream</code> object or cell array consisting of one such object. If you do not specify <code>Streams</code>, <code>sequentialfs</code> uses the default stream.</li> </ul>

## Examples

Perform sequential feature selection for classification of noisy features:

```
load fisheriris;
X = randn(150,10);
X(:,[1 3 5 7])= meas;
y = species;

c = cvpartition(y,'k',10);
opts = statset('display','iter');
fun = @(XT,yT,Xt,yt)...
      (sum(~strcmp(yt,classify(Xt,XT,yT,'quadratic'))));

[fs,history] = sequentialfs(fun,X,y,'cv',c,'options',opts)
```

```
Start forward sequential feature selection:
Initial columns included: none
Columns that cannot be included: none
Step 1, added column 7, criterion value 0.04
Step 2, added column 5, criterion value 0.0266667
Final columns included: 5 7
```

```
fs =
    0  0  0  0  1  0  1  0  0  0
history =
    In: [2x10 logical]
    Crit: [0.0400 0.0267]
```

```
history.In
ans =
    0  0  0  0  0  0  1  0  0  0
    0  0  0  0  1  0  1  0  0  0
```

## See Also

[crossval](#) | [cvpartition](#) | [relieff](#) | [statset](#) | [stepwisefit](#)

## Topics

“Select Subset of Features with Comparative Predictive Power” on page 15-75

“Sequential Feature Selection” on page 15-73

**Introduced in R2008a**



# set

**Class:** dataset

Set and display properties

---

**Note** The `dataset` data type might be removed in a future release. To work with heterogeneous data, use the MATLAB® `table` data type instead. See MATLAB `table` documentation for more information.

---

## Syntax

```
set(A)
set(A,PropertyName)
A = set(A,PropertyName,PropertyValue,...)
B = set(A,PropertyName,value)
```

## Description

`set(A)` displays all properties of the dataset array `A` and their possible values.

`set(A,PropertyName)` displays possible values for the property specified by `PropertyName`.

`A = set(A,PropertyName,PropertyValue,...)` sets property name/value pairs.

`B = set(A,PropertyName,value)` returns a dataset array `B` that is a copy of `A`, but with the property `'PropertyName'` set to the value `value`.

---

**Note** Using `set(A,'PropertyName',value)` without assigning to a variable does not modify `A`'s properties. Use `A = set(A,'PropertyName',value)` to modify `A`.

---

## Examples

Create a dataset array from Fisher's iris data and add a description:

```
load fisheriris
NumObs = size(meas,1);
NameObs = strcat({'Obs'},num2str((1:NumObs)','%-d'));
iris = dataset({nominal(species),'species'},...
              {meas,'SL','SW','PL','PW'},...
              'ObsNames',NameObs);
iris = set(iris,'Description','Fisher''s Iris Data');
get(iris)
  Description: 'Fisher's Iris Data'
  Units: {}
  DimNames: {'Observations' 'Variables'}
  UserData: []
  ObsNames: {150x1 cell}
  VarNames: {'species' 'SL' 'SW' 'PL' 'PW'}
```

## See Also

[get](#) | [summary](#)

## setDefaultYfit

**Class:** CompactTreeBagger

Set default value for predict

### Syntax

```
B = setDefaultYfit(B, Yfit)
```

### Description

`B = setDefaultYfit(B, Yfit)` sets the default prediction for ensemble `B` to `Yfit`. The default prediction must be a character variable for classification or a numeric scalar for regression. This setting controls what predicted value `CompactTreeBagger` returns when no prediction is possible, for example when the `predict` method needs to predict for an observation which has only false values in the matrix supplied through `'UseInstanceForTree'` argument.

### See Also

`TreeBagger.DefaultYfit` | `predict`

## setdiff

**Class:** dataset

Set difference for dataset array observations

---

**Note** The `dataset` data type might be removed in a future release. To work with heterogeneous data, use the MATLAB® `table` data type instead. See MATLAB `table` documentation for more information.

---

## Syntax

```
C = setdiff(A,B)
C = setdiff(A,B,vars)
C = setxor(A,B,vars,setOrder)
[C,iA] = setxor(____)
```

## Description

`C = setdiff(A,B)` for dataset arrays `A` and `B` returns the set of observations that are in `A` but not `B`, with repetitions removed. The observations in the dataset array `C` are sorted.

`C = setdiff(A,B,vars)` returns the set of observations that are in `A` but not `B`, considering only the variables specified in `vars`, with repetitions removed. The observations in the dataset array `C` are sorted by these variables. The values for variables not specified in `vars` for each observation in `C` are taken from the corresponding observation in `A`. If there are multiple observations in `A` that correspond to an observation in `C`, those values are taken from the first occurrence.

`C = setxor(A,B,vars,setOrder)` returns the observations in `C` in the order specified by `setOrder`.

`[C,iA] = setxor(____)` also returns the index vector `iA` such that `C = A(iA,:)`. If there are repeated observations in `A`, then `setxor` returns the index of the first occurrence. You can use any of the previous input arguments.

## Input Arguments

**A,B**

Input dataset arrays.

**vars**

Cell array of character vectors containing variable names or a vector of integers containing variable column numbers, indicating the variables that `setdiff` considers.

Specify `vars` as `[]` to use its default value of all variables.

**setOrder**

Flag indicating the sorting order for the observations in C. The possible values of `setOrder` are:

'sorted'	Observations in C are in sorted order (default).
'stable'	Observations in C are in the same order that they appear in A.

## Output Arguments

**C**

Dataset array with the observations that are in A but not B, with repetitions removed. C is in sorted order (by default), or the order specified by `setOrder`.

**iA**

Index vector, indicating the observations from A that are in C. The vector `iA` contains the index to the first occurrence of any repeated observations in A.

## Examples

### Set Difference of Two Dataset Arrays

Create a scalar structure array, and then convert it into two dataset arrays.

```
S(1,1).Name = 'CLARK';  
S(1,1).Gender = 'M';  
S(1,1).SystolicBP = 124;  
S(1,1).DiastolicBP = 93;
```

```
S(2,1).Name = 'BROWN';  
S(2,1).Gender = 'F';  
S(2,1).SystolicBP = 122;  
S(2,1).DiastolicBP = 80;
```

```
S(3,1).Name = 'MARTIN';  
S(3,1).Gender = 'M';  
S(3,1).SystolicBP = 130;  
S(3,1).DiastolicBP = 92;
```

```
A = struct2dataset(S(1:2));  
B = struct2dataset(S(2:3));
```

The intersection of A and B is the second observation, with last name BROWN.

Return the set difference of A and B.

```
[C,iA] = setdiff(A,B)
```

```
C =
```

Name	Gender	SystolicBP	DiastolicBP
'CLARK'	'M'	124	93

```
iA =
```

```
1
```

The first observation in A is not present in B.

## See Also

`dataset` | `intersect` | `ismember` | `setxor` | `sortrows` | `union` | `unique`

## Topics

“Dataset Arrays” on page 2-142

## setlabels

Assign labels to levels of nominal or ordinal arrays

---

**Note** The `nominal` and `ordinal` array data types might be removed in a future release. To represent ordered and unordered discrete, nonnumeric data, use the MATLAB® `categorical` data type instead.

---

### Syntax

```
B = setlabels(A, labels)
B = setlabels(A, labels, levels)
```

### Description

`B = setlabels(A, labels)` returns a nominal or ordinal array object the same as `A` but with levels labeled in the order specified by `labels`.

`B = setlabels(A, labels, levels)` labels only the levels specified in `levels`.

### Examples

- “Change Category Labels” on page 2-10

### Input Arguments

**A** — Nominal or ordinal array

nominal array | ordinal array

Nominal or ordinal array, specified as a nominal or ordinal array object created using `nominal` or `ordinal`.



**labels — Labels to assign**

cell array of character vectors | 2-D character matrix

Labels to assign to levels, specified as a cell array of character vectors or 2-D character matrix.

Data Types: char | cell

**levels — Levels to assign labels**

cell array of character vectors | 2-D character matrix

Level to assign labels to, specified as a cell array of character vectors or 2-D character matrix.

Data Types: char | cell

## Output Arguments

**B — Nominal or ordinal array**

nominal array | ordinal array

Nominal or ordinal array, returned as a nominal or ordinal array object.

## See Also

getlabels | nominal | ordinal

## Topics

“Change Category Labels” on page 2-10

nominal

ordinal

Introduced in R2007a

## setxor

**Class:** dataset

Set exclusive or for dataset array observations

---

**Note** The `dataset` data type might be removed in a future release. To work with heterogeneous data, use the MATLAB® `table` data type instead. See MATLAB `table` documentation for more information.

---

## Syntax

```
C = setxor(A,B)
C = setxor(A,B,vars)
C = setxor(A,B,vars,setOrder)
[C,iA,iB] = setxor(____)
```

## Description

`C = setxor(A,B)` for dataset arrays `A` and `B` returns the set of observations that are not in the intersection of the two arrays, with repetitions removed. The observations in the dataset array `C` are sorted.

`C = setxor(A,B,vars)` returns the set of observations that are not in the intersection of the two arrays, considering only the variables specified in `vars`, with repetitions removed. The observations in the dataset array `C` are sorted by these variables. The values for variables not specified in `vars` for each observation in `C` are taken from the corresponding observation in `A` or `B`. If there are multiple observations in `A` or `B` that correspond to an observation in `C`, those values are taken from the first occurrence.

`C = setxor(A,B,vars,setOrder)` returns the observations in `C` in the order specified by `setOrder`.

`[C,iA,iB] = setxor(____)` also returns index vectors `iA` and `iB` such that `C` is a sorted combination of the values `A(iA,:)` and `B(iB,:)`. If there are repeated

observations in A or B, then `setxor` returns the index of the first occurrence. You can use any of the previous input arguments.

## Input Arguments

**A,B**

Input dataset arrays.

**vars**

Cell array of character vectors containing variable names or a vector of integers containing variable column numbers, indicating the variables in A and B that `setxor` considers.

Specify `vars` as `[]` to use its default value of all variables.

**setOrder**

Flag indicating the sorting order for the observations in C. The possible values of `setOrder` are:

'sorted'	Observations in C are in sorted order (default).
'stable'	Observations in C are in the same order that they appear in A, then B.

## Output Arguments

**C**

Dataset array with the observations not in the intersection of A and B, with repetitions removed. C is in sorted order (by default), or the order specified by `setOrder`.

**iA**

Index vector, indicating the observations from A that are in C. The vector `iA` contains the index to the first occurrence of any repeated observations in A.

**iB**

Index vector, indicating the observations from B that are in C. The vector `iB` contains the index to the first occurrence of any repeated observations in B.

## Examples

### Symmetric Difference of Two Dataset Arrays

Create a scalar structure array, and then convert it into two dataset arrays.

```
S(1,1).Name = 'CLARK';  
S(1,1).Gender = 'M';  
S(1,1).SystolicBP = 124;  
S(1,1).DiastolicBP = 93;  
  
S(2,1).Name = 'BROWN';  
S(2,1).Gender = 'F';  
S(2,1).SystolicBP = 122;  
S(2,1).DiastolicBP = 80;  
  
S(3,1).Name = 'MARTIN';  
S(3,1).Gender = 'M';  
S(3,1).SystolicBP = 130;  
S(3,1).DiastolicBP = 92;  
  
A = struct2dataset(S(1:2));  
B = struct2dataset(S(2:3));
```

The intersection of A and B is the second observation, with last name BROWN.

Return the symmetric difference of A and B.

```
[C,iA,iB] = setxor(A,B);  
C
```

C =

Name	Gender	SystolicBP	DiastolicBP
'CLARK'	'M'	124	93

```
'MARTIN'      'M'      130      92

[iA iB]

ans =

     1     2
```

The symmetric difference contains the first observation from A, and the second observation from B.

## See Also

`dataset` | `intersect` | `ismember` | `setdiff` | `sortrows` | `union` | `unique`

## Topics

“Dataset Arrays” on page 2-142

## SharedCovariance property

**Class:** `gmdistribution`

`true` if all covariance matrices are restricted to be the same

### Description

Logical `true` if all the covariance matrices are restricted to be the same (pooled estimate); logical `false` otherwise.

# shrink

**Class:** RegressionEnsemble

Prune ensemble

## Syntax

```
cmp = shrink(ens)
cmp = shrink(ens, Name, Value)
```

## Description

`cmp = shrink(ens)` returns a compact shrunken version of `ens`, a regularized ensemble. `cmp` retains only learners with weights above a threshold.

`cmp = shrink(ens, Name, Value)` returns an ensemble with additional options specified by one or more `Name, Value` pair arguments. You can specify several name-value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

## Input Arguments

### **ens**

A regression ensemble created with `fitrensemble`.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

**lambda**

Vector of nonnegative regularization parameter values for lasso. If `ens.Regularization` is nonempty (populate it with `regularize`), `shrink` regularizes `ens` using `lambda`. If `ens` contains a `Regularization` structure, you cannot pass `lambda`.

**Default:** []

**threshold**

Lower cutoff on weights for weak learners, a numeric nonnegative scalar. `shrink` creates `cmp` from those learners with weights above `threshold`.

**Default:** 0

**weightcolumn**

Column index of `ens.Regularization.TrainedWeights`, a positive integer. `shrink` creates `cmp` with learner weights from this column.

**Default:** 1

## Output Arguments

**cmp**

A regression ensemble of class `CompactRegressionEnsemble`. Use `cmp` for making predictions exactly as you use `ens`, with the `predict` method.

`shrink` orders the members of `cmp` from largest to smallest.

## Examples

### Shrink Bagged Regression Ensemble

Shrink a 300-member bagged regression ensemble, and view the number of members of the resulting ensemble.



Generate sample data.

```
rng(10, 'twister') % For reproducibility
X = rand(2000,20);
Y = repmat(-1,2000,1);
Y(sum(X(:,1:5),2)>2.5) = 1;
```

Shrink a 300-member bagged regression ensemble using 0.1 for the parameter lambda.

```
bag = fitensemble(X,Y, 'Method', 'Bag', 'NumLearningCycles', 300, 'Learners', 'Tree');
cmp = shrink(bag, 'lambda', 0.1);
```

View the number of members of the resulting ensemble.

```
cmp.NumTrained
ans = 87
```

- “Ensemble Regularization” on page 18-88

## See Also

cvshrink | predict | regularize

## Topics

“Ensemble Regularization” on page 18-88

## Sigma property

**Class:** gmdistribution

Input array of covariances

## Description

Input array of covariances SIGMA.

# signrank

Wilcoxon signed rank test

## Syntax

```
p = signrank(x)
p = signrank(x,y)
p = signrank(x,y,Name,Value)
[p,h] = signrank(____)
[p,h,stats] = signrank(____)

[____] = signrank(x,m)
[____] = signrank(x,m,Name,Value)
```

## Description

`p = signrank(x)` returns the  $p$ -value of a two-sided Wilcoxon signed rank test on page 32-6094.

`signrank` tests the null hypothesis that data in the vector `x` come from a distribution whose median is zero at the 5% significance level. The test assumes that the data in `x` come from a continuous distribution symmetric about its median.

`p = signrank(x,y)` returns the  $p$ -value of a paired, two-sided test for the null hypothesis that  $x - y$  comes from a distribution with zero median.

`p = signrank(x,y,Name,Value)` returns the  $p$ -value for the sign test with additional options specified by one or more `Name,Value` pair arguments.

`[p,h] = signrank(____)` also returns a logical value indicating the test decision. `h = 1` indicates a rejection of the null hypothesis, and `h = 0` indicates a failure to reject the null hypothesis at the 5% significance level. You can use any of the input arguments in the previous syntaxes.

`[p,h,stats] = signrank(____)` also returns the structure `stats` with information about the test statistic.

[ \_\_\_ ] = `signrank(x,m)` returns any of the output arguments in the previous syntaxes for the null hypothesis that the data in `x` are observations from a distribution with median `m`.

[ \_\_\_ ] = `signrank(x,m,Name,Value)` returns any of the output arguments in the previous syntaxes for the signed rank test with additional options specified by one or more `Name,Value` pair arguments.

## Examples

### Test for Zero Median of a Single Population

Test the hypothesis of zero median.

Generate the sample data.

```
rng('default') % for reproducibility
x = randn(1,25) + 1.30;
```

Test the hypothesis that the data in `x` has zero median.

```
[p,h] = signrank(x)
```

```
p =
    3.2229e-05
```

```
h =
    logical
     1
```

At the default 5% significance level, the value `h = 1` indicates that the test rejects the null hypothesis of zero median.

## Test the Median of Differences of Paired Samples

Test the hypothesis of zero median for the difference between paired samples.

Generate the sample data.

```
rng('default') % for reproducibility
x = lognrnd(2, .25, 10, 1);
y = x + trnd(2, 10, 1);
```

Test the hypothesis that  $x - y$  has zero median.

```
[p,h] = signrank(x,y)
```

```
p =
    0.3223
```

```
h =
    logical
    0
```

The results indicate that the test fails to reject the null hypothesis of zero median in the difference at the default 5% significance level.

## Signed Rank Test for Large Samples

Conduct a -sided test on a large sample using approximation.

Load the sample data.

```
load(fullfile(matlabroot, 'examples', 'stats', 'gradespaired.mat'));
```

Test the null hypothesis that the median of the grade differences of students before and after participating in a tutoring program is 0 against the alternate that it is less than 0.

```
[p,h,stats] = signrank(gradespaired(:,1), ...
    gradespaired(:,2), 'tail', 'left')
```

```
p =  
    0.0047  
  
h =  
    logical  
    1  
  
stats =  
    struct with fields:  
        zval: -2.5982  
        signedrank: 2.0175e+03
```

Because the sample size is greater than 15, `signrank` uses an approximate method to calculate the  $P$ -value and also returns the value of the  $z$ -statistic. The value `h = 1` indicates that the test rejects the null hypothesis that there is no difference between the grade medians at the 5% significance level. There is enough statistical evidence to conclude that the median grade before the tutoring program is less than the median grade after the tutoring program.

Repeat the test using the exact method.

```
[p,h,stats] = signrank(gradespaired(:,1),gradespaired(:,2),...  
    'tail','left','method','exact')  
  
p =  
    0.0045  
  
h =  
    logical  
    1
```

```
stats =  
  struct with fields:  
    signedrank: 2.0175e+03
```

The results obtained using the approximate method are consistent with the exact method.

### Two-Sided Test for the Median of a Single Population

Load the sample data.

```
load mileage
```

The data contains the mileages per gallon for three different types of cars in columns 1 to 3.

Test the hypothesis that the median mileage for the type of cars in the second column differs from 33.

```
[p,h,stats] = signrank(mileage(:,2),33)
```

```
p =  
  0.0313
```

```
h =  
  logical  
  1
```

```
stats =  
  struct with fields:
```

```
signedrank: 21
```

At the 5% significance level, the results indicate that the median mileage for the second type of cars differs from 33. Note that `signrank` uses an exact method to calculate the  $P$ -value for small samples and does not return the  $Z$ -statistic.

### Right-Sided Test for the Median of a Single Population

Use the name-value pair arguments in `signrank`.

Load the sample data.

```
load mileage
```

The data contains the mileage per gallon for three different types of cars in columns 1 to 3.

Test the hypothesis that the median mileage for the type of cars in the second row are larger than 33.

```
[p,h,stats] = signrank(mileage(:,2),33,'tail','right')
```

```
p =
```

```
0.0156
```

```
h =
```

```
logical
```

```
1
```

```
stats =
```

```
struct with fields:
```

```
signedrank: 21
```



Repeat the same test at the 1% significance level using the approximate method.

```
[p,h,stats] = signrank(mileage(:,2),33,'tail','right',...  
'alpha',0.01,'method','approximate')
```

```
p =
```

```
    0.0180
```

```
h =
```

```
logical
```

```
    0
```

```
stats =
```

```
struct with fields:
```

```
    zval: 2.0966
```

```
 signedrank: 21
```

This result,  $h = 0$ , indicates that the null hypothesis cannot be rejected at the 1% significance level.

## Input Arguments

### **x** — Sample data

vector

Sample data, specified as a vector.

Data Types: `single` | `double`

### **y** — Sample data

vector

Sample data, specified as a vector.  $y$  must be the same length as  $x$ .

Data Types: `single` | `double`

**m — Hypothesized value of the median**

scalar

Hypothesized value of the median, specified as a scalar.

Example: `signrank(x,10)`

Data Types: `single` | `double`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'alpha', 0.01, 'method', 'approximate', 'tail', 'right'` specifies a right-tailed signed rank test with 1% significance level, which returns the approximate  $p$ -value.

**alpha — Significance level**

0.05 (default) | scalar value in the range 0 to 1

Significance level of the decision of a hypothesis test, specified as the comma-separated pair consisting of `'alpha'` and a scalar value in the range 0 to 1. Significance level of  $h$  is  $100 * \text{alpha}\%$ .

Example: `'alpha', 0.01`

Data Types: `double` | `single`

**method — Computation method of  $p$** 

`'exact'` | `'approximate'`

Computation method of  $p$ , specified as the comma-separated pair consisting of `'method'` and one of the following.

<code>'exact'</code>	Exact computation of the $p$ -value, $p$ . Default value for 15 or fewer observations in $x$ , $x - m$ , or $x - y$ when <code>method</code> is unspecified.
----------------------	--

'approximate'	Normal approximation while computing the $p$ -value, $p$ . Default value for more than 15 observations in $x$ , $x - m$ , or $x - y$ when 'method' is unspecified because the exact method can be slow on large samples.
---------------	--

Example: 'method','exact'

Data Types: char

### tail — Type of test

'both' (default) | 'right' | 'left'

Type of test, specified as the comma-separated pair consisting of 'tail' and one of the following:

'both'	Two-sided hypothesis test, which is the default test type. <ul style="list-style-type: none"> <li>For a one-sample test, the alternate hypothesis states that the data in <math>x</math> come from a continuous distribution with median different than 0 or <math>m</math>.</li> <li>For a two-sample test, the alternate hypothesis states that the data in <math>x - y</math> come from a distribution with median different than 0.</li> </ul>
'right'	Right-tailed hypothesis test. <ul style="list-style-type: none"> <li>For a one-sample test, the alternate hypothesis states that the data in <math>x</math> come from a continuous distribution with median greater than 0 or <math>m</math>.</li> <li>For a two-sample test, the alternate hypothesis states the data in <math>x - y</math> come from a distribution with median greater than 0.</li> </ul>
'left'	Left-tailed hypothesis test. <ul style="list-style-type: none"> <li>For a one-sample test, the alternate hypothesis states that the data in <math>x</math> come from a continuous distribution with median less than 0 or <math>m</math>.</li> <li>For a two-sample test, the alternate hypothesis states the data in <math>x - y</math> come from a distribution with median less than 0.</li> </ul>

Example: 'tail','left'

## Output Arguments

**p** —  $p$ -value of the test

nonnegative scalar

$p$ -value of the test, returned as a nonnegative scalar from 0 to 1.  $p$  is the probability of observing a test statistic as or more extreme than the observed value under the null hypothesis. `signrank` computes the two-sided  $p$ -value by doubling the most significant one-sided value.

#### **h** — Result of the hypothesis test

1 | 0

Result of the hypothesis test, returned as a logical value.

- If  $h = 1$ , this indicates the rejection of the null hypothesis at the  $100 * \alpha\%$  significance level.
- If  $h = 0$ , this indicates a failure to reject the null hypothesis at the  $100 * \alpha\%$  significance level.

#### **stats** — Test statistics

structure

Test statistics, returned as a structure. The test statistics stored in `stats` are:

- `signrank`: Value of the sign rank test statistic.
- `zval`: Value of the  $z$ - statistic on page 32-6095 (computed when 'method' is 'approximate').

## Definitions

### Wilcoxon Signed Rank Test

The Wilcoxon signed rank test is a nonparametric test for two populations when the observations are paired. In this case, the test statistic,  $W$ , is the sum of the ranks of positive differences between the observations in the two samples (that is,  $x - y$ ). When you use the test for one sample, then  $W$  is the sum of the ranks of positive differences between the observations and the hypothesized median value  $M_0$  (which is 0 when you use `signrank(x)` and  $m$  when you use `signrank(x, m)`).

## z-Statistic

For large samples, or when method is approximate, the signrank function calculates the  $p$ -value using the  $z$ -statistic, given by

$$z = \frac{(W - n(n+1)/4)}{\sqrt{\frac{n(n+1)(2n+1) - tieadj}{24}}},$$

where  $n$  is the sample size of the difference  $x - y$  or  $x - m$ . For the two-sample case, signrank uses `[tie_rank, tieadj] = tiedrank(abs(diffxy), 0, 0, epsdiff)` to obtain the tie adjustment value `tieadj`.

## Algorithms

signrank treats NaNs in  $x$  and  $y$  as missing values and ignores them.

For the two-sample case, signrank uses a tolerance based on the values `epsdiff = eps(x) + eps(y)`. The signrank function treats any pair of values with difference `d(i) = x(i) - y(i)` that differ by no more than the sum of their two `eps` values (`abs(d(i)) < epsdiff(i)`) as ties.

## References

- [1] Gibbons, J. D., and S. Chakraborti. *Nonparametric Statistical Inference*, 5th Ed., Boca Raton, FL: Chapman & Hall/CRC Press, Taylor & Francis Group, 2011.
- [2] Hollander, M., and D. A. Wolfe. *Nonparametric Statistical Methods*. Hoboken, NJ: John Wiley & Sons, Inc., 1999.

## See Also

`ranksum` | `signtest` | `ttest` | `ztest`

Introduced before R2006a

## signtest

Sign test

### Syntax

```
p = signtest(x)
p = signtest(x,y)
p = signtest(x,y,Name,Value)
[p,h] = signtest(____)
[p,h,stats] = signtest(____)

[____] = signtest(x,m)
[____] = signtest(x,m,Name,Value)
```

### Description

`p = signtest(x)` returns the  $p$ -value for a two-sided sign test on page 32-6106.

`signtest` tests the hypothesis that data in `x` has a continuous distribution with zero median against the alternative that the distribution does not have zero median at the 5% significance level.

`p = signtest(x,y)` returns the  $p$ -value of a two-sided sign test on page 32-6106. Here, `signtest` tests for the hypothesis that the data in `x - y` has a distribution with zero median against the alternative that the distribution does not have zero median. Note that a hypothesis of zero median for `x - y` is not equivalent to a hypothesis of equal median for `x` and `y`.

`p = signtest(x,y,Name,Value)` returns the  $p$ -value for the sign test with additional options specified by one or more `Name,Value` pair arguments.

`[p,h] = signtest(____)` also returns a logical value indicating the test decision. The value `h = 1` indicates a rejection of the null hypothesis, and `h = 0` indicates a failure to reject the null hypothesis at the 5% significance level. You can use any of the input arguments in the previous syntaxes.

`[p,h,stats] = signtest(____)` also returns the structure `stats` containing information about the test statistic.

`[____] = signtest(x,m)` returns any of the output arguments in the previous syntaxes for the test whether the data in `x` are observations from a distribution with median `m` against the alternative that the median is different from `m`.

`[____] = signtest(x,m,Name,Value)` returns any of the output arguments in the previous syntaxes for the sign test with additional options specified by one or more `Name,Value` pair arguments.

## Examples

### Test for Zero Median of a Single Population

Test the hypothesis of zero median.

Generate the sample data.

```
rng('default') % for reproducibility
x = randn(1,25);
```

The sampling distribution of `x` is symmetric with zero median.

Test the null hypothesis that `x` comes from a distribution with a median different from zero median.

```
[p,h,stats] = signtest(x,0)
```

```
p =
    0.1078
```

```
h =
    logical
    0
```

```
stats =  
  
    struct with fields:  
  
    zval: NaN  
    sign: 17
```

At the default 5% significance level, the result `h = 0` indicates that `signtest` fails to reject to the null hypothesis of zero median. `signtest` calculates the  $p$ -value using the exact method, hence it does not calculate `zval` and returns it as a NaN.

### Test for Zero Median for the Difference of Paired Samples

Test the hypothesis of zero median for the difference between paired samples.

Generate the sample data.

```
rng('default') % for reproducibility  
before = lognrnd(2, .25, 10, 1);  
after = before + (lognrnd(0, .5, 10, 1) - 1);
```

The sampling distribution of the difference between `before` and `after` is symmetric with zero median.

Test the null hypothesis that the difference of `before` and `after` has zero median.

```
[p,h] = signtest(before,after)
```

```
p =  
  
    0.7539
```

```
h =  
  
    logical  
  
    0
```



At the default 5% significance level, the value  $h = 0$  indicates that `signtest` fails to reject to the null hypothesis of zero median in the difference.

### Medians of Paired Samples

Test the hypothesis of zero median for the difference between two paired samples using the exact and approximate methods.

Generate the sample data.

```
rng('default') % for reproducibility
x = lognrnd(2, .25, 15, 1);
y = x + trnd(2, 15, 1);
display([x y])
```

```
8.4521    7.8047
11.6869   11.4094
4.2009    5.1133
9.1664   12.1655
8.0020   10.0300
5.3285    6.0153
6.6300    5.1235
8.0499    8.6737
18.0763   19.2164
14.7665   15.3380
5.2726    8.4187
15.7798   16.2093
8.8583    8.5575
7.2735    7.4783
8.8347    7.8894
```

Test the hypothesis that  $x - y$  has zero median.

```
[p,h,stats] = signtest(x,y)
```

```
p =
```

```
0.3018
```

```
h =
```

```
logical
    0

stats =

    struct with fields:

        zval: NaN
        sign: 5
```

At the default 5% significance level, the value  $h = 0$  indicates that the test fails to reject the null hypothesis of zero median in the difference.

Repeat the test using the approximate method.

```
[p,h,stats] = signtest(x,y,'method','approximate')

p =

    0.3017

h =

    logical
    0

stats =

    struct with fields:

        zval: -1.0328
        sign: 5
```

The approximate  $P$ -value, which `signtest` obtains using the  $z$ -statistic, is really close to the exact  $P$ -value.

### Test for Large Samples

Perform a left-sided sign test for large samples.

Load the sample data.

```
load(fullfile(matlabroot, 'examples', 'stats', 'gradespaired.mat'));
```

Test the null hypothesis that the median of the grade differences before and after the tutoring program is 0 against the alternate that it is less than 0.

```
[p,h,stats] = signtest(gradespaired(:,1),gradespaired(:,2),...  
'tail','left')
```

```
p =
```

```
0.0013
```

```
h =
```

```
logical
```

```
1
```

```
stats =
```

```
struct with fields:
```

```
zval: -3.0110
```

```
sign: 37
```

Because the sample size is large (greater than 100), `signtest` uses an approximate method to calculate the  $P$ -value and also returns the value of the  $z$ -statistic. The test

rejects the null hypothesis that there is no difference between the grade medians at the 5% significance level.

### Test for Median of a Single Population

Test the hypothesis that the population median is different from a specified value.

Load the sample data.

```
load lawdata
```

The data set has 15 observations for variables `gpa` and `lsat`.

Test the hypothesis that the median `lsat` score is higher than 570.

```
[p,h,stats] = signtest(lsat,570,'tail','right')
```

```
p =
```

```
    0.0176
```

```
h =
```

```
logical
```

```
    1
```

```
stats =
```

```
struct with fields:
```

```
    zval: NaN
```

```
    sign: 12
```

Both the  $P$ -value, 0.0176, and  $h = 1$  indicate that at the 5% significance level the test concludes in favor of the alternate hypothesis.

## Input Arguments

### **x** — Sample data

vector

Sample data, specified as a vector.

Data Types: `single` | `double`

### **y** — Sample data

vector

Sample data, specified as a vector.  $y$  must be the same length as  $x$ .

Data Types: `single` | `double`

### **m** — Hypothesized value of the median

scalar

Hypothesized value of the median, specified as a scalar.

Example: `signtest(x, 35)`

Data Types: `single` | `double`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'alpha', 0.01, 'method', 'approximate', 'tail', 'right'` specifies a right-tailed sign test with 1% significance level, which returns the approximate  $p$ -value.

### **alpha** — Significance level

0.05 (default) | scalar value in the range 0 to 1

Significance level of the hypothesis test, specified as the comma-separated pair consisting of 'alpha' and a scalar value in the range 0 to 1. The default value of alpha is 0.05.

Significance level of h is  $100 * \alpha\%$ .

Example: 'alpha', 0.01

Data Types: double | single

#### method — *p*-value computation method

'exact' | 'approximate'

*p*-value computation method, specified as the comma-separated pair consisting of 'method' and one of the following:

'exact'	Exact computation of the <i>p</i> -value, p.
'approximate'	Normal approximation for computing the <i>p</i> -value, p.

The default computation method is 'exact', if there are fewer than 100 observations and 'approximate' if there are 100 observations or more.

Example: 'method','exact'

Data Types: char

#### tail — Type of test

'both' (default) | 'right' | 'left'

Type of test, specified as the comma-separated pair consisting of 'tail' and one of the following:

'both'	Two-sided hypothesis test, which is the default test type. <ul style="list-style-type: none"> <li>For a one-sample test, the alternate hypothesis states that the data in <i>x</i> come from a continuous distribution with median different than zero (or <i>m</i>).</li> <li>For a two-sample test, the alternate hypothesis states that the data in <i>x-y</i> come from a distribution with median different than zero.</li> </ul>
'right'	Right-tailed hypothesis test. <ul style="list-style-type: none"> <li>For a one-sample test, the alternate hypothesis states that the data in <i>x</i> come from a continuous distribution with median greater than zero (or <i>m</i>).</li> <li>For a two-sample test, the alternate hypothesis states the data in <i>x-y</i> come from a distribution with median greater than zero.</li> </ul>

'left'	<p>Left-tailed hypothesis test.</p> <ul style="list-style-type: none"> <li>• For a one-sample test, the alternate hypothesis states that the data in <math>x</math> come from a continuous distribution with median less than zero (or <math>m</math>).</li> <li>• For a two-sample test, the alternative hypothesis states the data in <math>x-y</math> come from a distribution with median less than zero.</li> </ul>
--------	--

Example: 'tail','left'

## Output Arguments

### **p** — *p*-value of the test

nonnegative scalar

*p*-value of the test, returned as a nonnegative scalar from 0 to 1. *p* is the probability of observing a test statistic as or more extreme than the observed value under the null hypothesis. `signtest` computes the two-sided *p*-value by doubling the most significant one-sided value.

### **h** — Result of the hypothesis test

1 | 0

Result of the hypothesis test, returned as a logical value.

- If  $h = 1$ , this indicates rejection of the null hypothesis at the  $100 * \alpha\%$  significance level.
- If  $h = 0$ , this indicates a failure to reject the null hypothesis at the  $100 * \alpha\%$  significance level.

### **stats** — Test statistics

structure

Test statistics, returned as a structure. The test statistics stored in `stats` are:

- `sign`: Value of the sign test statistic.
- `zval`: Value of the z-statistic on page 32-6106 (computed only for large samples).

## Definitions

### Sign Test

The sign test is a nonparametric test for the median of a population or median of the difference of two populations.

For example, for tests on a single population median:

- If the test is two-sided, then the test statistic,  $S$ , is the minimum of the number of observations that are smaller or larger than the hypothesized median value,  $M_0$ .
- If the test is right-sided, then  $S$  is the number of observations that are larger than the hypothesized median value  $M_0$ .
- If the test is left-sided, then  $S$  is the number of observations that are smaller than the hypothesized median value  $M_0$ .

### z-Statistic

For a large sample, `signtest` uses the  $z$ -statistic to approximate the  $p$ -value.

The `signtest` test statistic is the number of elements that are greater than 0 (for `signtest(x)` or `signtest(x-y)`), or  $m$  (for `signtest(x, m)`). Hence, the  $z$ -statistic of the sign test, with the continuity correction, is:

$$z = \frac{(S - E(S))}{\sqrt{V(S)}} = \frac{(S - (0.5)n - 0.5\text{sign}(npos - nneg))}{\sqrt{(0.5)(0.5)n}},$$

where  $npos$  and  $nneg$  are the number of positive and negative differences from the hypothesized median value, respectively.

## Algorithms

For a one-sample test, `signtest` omits values in  $x$  that are zero or NaN.

For a two-sample test, `signtest` omits values in  $x - y$  that are zero or NaN.



## References

- [1] Gibbons, J. D., and S. Chakraborti. *Nonparametric Statistical Inference*, 5th Ed. Boca Raton, FL: Chapman & Hall/CRC Press, Taylor & Francis Group, 2011.
- [2] Hollander, M., and D. A. Wolfe. *Nonparametric Statistical Methods*. Hoboken, NJ: John Wiley & Sons, Inc., 1999.

## See Also

ranksum | signrank | ttest | ztest

**Introduced before R2006a**

## silhouette

Silhouette plot

### Syntax

```
silhouette(X, clust)
s = silhouette(X, clust)
[s, h] = silhouette(X, clust)
[...] = silhouette(X, clust, metric)
[...] = silhouette(X, clust, distfun, p1, p2, ...)
```

### Description

`silhouette(X, clust)` plots cluster silhouettes for the  $n$ -by- $p$  data matrix  $X$ , with clusters defined by `clust`. Rows of  $X$  correspond to points, columns correspond to coordinates. `clust` can be a categorical variable, numeric vector, character matrix, or cell array of character vectors containing a cluster name for each point. `silhouette` treats NaNs or empty character vectors in `clust` as missing values, and ignores the corresponding rows of  $X$ . By default, `silhouette` uses the squared Euclidean distance between points in  $X$ .

`s = silhouette(X, clust)` returns the silhouette values in the  $n$ -by-1 vector  $s$ , but does not plot the cluster silhouettes.

`[s, h] = silhouette(X, clust)` plots the silhouettes, and returns the silhouette values in the  $n$ -by-1 vector  $s$ , and the figure handle in  $h$ .

`[...] = silhouette(X, clust, metric)` plots the silhouettes using the inter-point distance function specified in `metric`. Choices for `metric` are given in the following table.

Metric	Description
'Euclidean'	Euclidean distance
'sqEuclidean'	Squared Euclidean distance (default)

Metric	Description
'cityblock'	Sum of absolute differences
'cosine'	One minus the cosine of the included angle between points (treated as vectors)
'correlation'	One minus the sample correlation between points (treated as sequences of values)
'Hamming'	Percentage of coordinates that differ
'Jaccard'	Percentage of nonzero coordinates that differ
Vector	A numeric distance matrix in upper triangular vector form, such as is created by <code>pdist</code> . <code>X</code> is not used in this case, and can safely be set to <code>[]</code> .

For more information on each metric, see “Distance Metrics” on page 18-16.

`[...] = silhouette(X, clust, distfun, p1, p2, ...)` accepts a function handle `distfun` to a metric of the form

```
d = distfun(X0, X, p1, p2, ...)
```

where `X0` is a 1-by-`p` point, `X` is an `n`-by-`p` matrix of points, and `p1, p2, ...` are optional additional arguments. The function `distfun` returns an `n`-by-1 vector `d` of distances between `X0` and each point (row) in `X`. The arguments `p1, p2, ...` are passed directly to the function `distfun`.

## Examples

### Create Silhouette Plot

Create a silhouette plot from clustered data.

Generate random sample data.

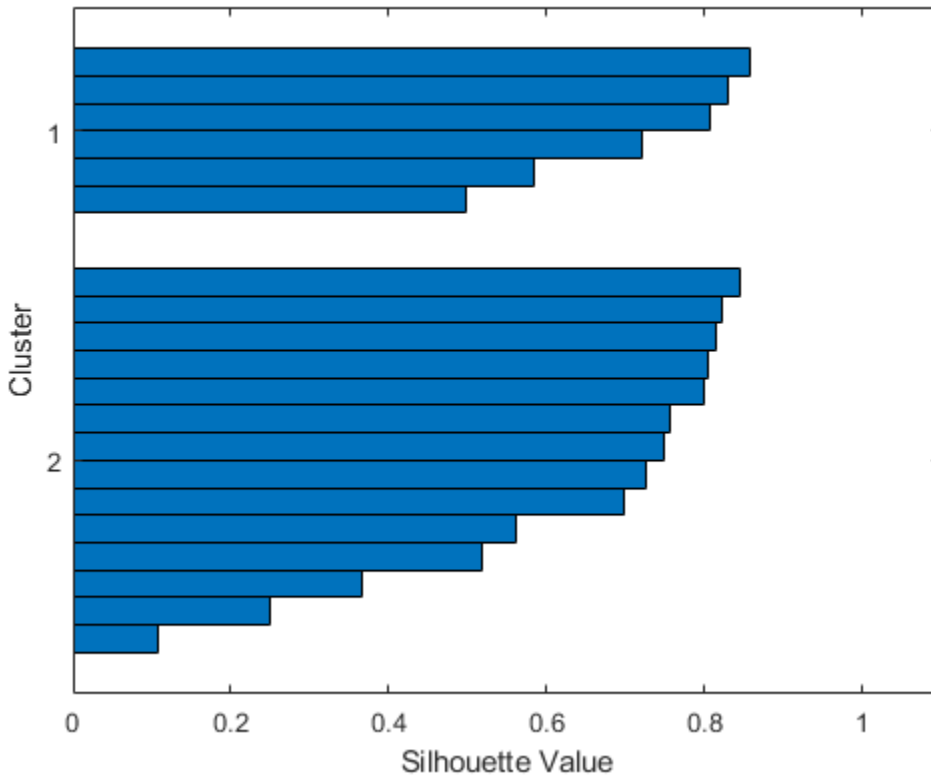
```
rng default % For reproducibility
X = [randn(10,2)+ones(10,2); randn(10,2)-ones(10,2)];
```

Cluster the data in `X` using `kmeans`.

```
cidx = kmeans(X,2);
```

Create a silhouette plot from the clustered data.

```
silhouette(X, cidx)
```



### Compute Silhouette Values

Compute the silhouette values from clustered data.

Generate random sample data.

```
rng default % For reproducibility  
X = [randn(10,2)+ones(10,2);randn(10,2)-ones(10,2)];
```

Use `kmeans` to cluster the data in `X` based on the sum of absolute differences in distance.

```
cidx = kmeans(X,2,'distance','cityblock');
```

Compute the silhouette values from the clustered data. Specify `metric` as `'cityblock'` to indicate that the `kmeans` clustering is based on the sum of absolute differences.

```
s = silhouette(X,cidx,'cityblock')
```

```
s =
```

```
0.0816  
0.5848  
0.1906  
0.2781  
0.3954  
0.4050  
0.0897  
0.5416  
0.6203  
0.6664  
0.5814  
0.6022  
0.6540  
0.5223  
0.5566  
0.4227  
0.6225  
0.6558  
0.5284  
0.6034
```

## Definitions

### Silhouette Value

The silhouette value for each point is a measure of how similar that point is to points in its own cluster, when compared to points in other clusters. The silhouette value for the  $i$ th point,  $S_i$ , is defined as

$$S_i = (b_i - a_i) / \max(a_i, b_i)$$

where  $a_i$  is the average distance from the  $i$ th point to the other points in the same cluster as  $i$ , and  $b_i$  is the minimum average distance from the  $i$ th point to points in a different cluster, minimized over clusters.

The silhouette value ranges from -1 to +1. A high silhouette value indicates that  $i$  is well-matched to its own cluster, and poorly-matched to neighboring clusters. If most points have a high silhouette value, then the clustering solution is appropriate. If many points have a low or negative silhouette value, then the clustering solution may have either too many or too few clusters. The silhouette clustering evaluation criterion can be used with any distance metric.

## References

- [1] Kaufman L., and P. J. Rousseeuw. *Finding Groups in Data: An Introduction to Cluster Analysis*. Hoboken, NJ: John Wiley & Sons, Inc., 1990.

## See Also

dendrogram | evalclusters | kmeans | linkage | pdist

## Topics

“Grouping Variables” on page 2-59

**Introduced before R2006a**

# clustering.evaluation.SilhouetteEvaluation class

**Package:** clustering.evaluation

**Superclasses:**

Silhouette criterion clustering evaluation object

## Description

`clustering.evaluation.SilhouetteEvaluation` is an object consisting of sample data, clustering data, and silhouette criterion values used to evaluate the optimal number of data clusters. Create a silhouette criterion clustering evaluation object using `evalclusters`.

## Construction

`eva = evalclusters(x, clust, 'Silhouette')` creates a silhouette criterion clustering evaluation object.

`eva = evalclusters(x, clust, 'Silhouette', Name, Value)` creates a silhouette criterion clustering evaluation object using additional options specified by one or more name-value pair arguments.

## Input Arguments

**x** — Input data

matrix

Input data, specified as an  $N$ -by- $P$  matrix.  $N$  is the number of observations, and  $P$  is the number of variables.

Data Types: `single` | `double`

**clust** — Clustering algorithm

'kmeans' | 'linkage' | 'gmdistribution' | matrix of clustering solutions | function handle

Clustering algorithm, specified as one of the following.

'kmeans'	Cluster the data in $x$ using the <code>kmeans</code> clustering algorithm, with 'EmptyAction' set to 'singleton' and 'Replicates' set to 5.
'linkage'	Cluster the data in $x$ using the <code>clusterdata</code> agglomerative clustering algorithm, with 'Linkage' set to 'ward'.
'gmdistribution'	Cluster the data in $x$ using the <code>gmdistribution</code> Gaussian mixture distribution algorithm, with 'SharedCov' set to <code>true</code> and 'Replicates' set to 5.

If `Criterion` is 'CalinskiHarabasz', 'DaviesBouldin', or 'silhouette', you can specify a clustering algorithm using a function handle (MATLAB). The function must be of the form `C = clustfun(DATA, K)`, where `DATA` is the data to be clustered, and `K` is the number of clusters. The output of `clustfun` must be one of the following:

- A vector of integers representing the cluster index for each observation in `DATA`. There must be `K` unique values in this vector.
- A numeric  $n$ -by- $K$  matrix of score for  $n$  observations and  $K$  classes. In this case, the cluster index for each observation is determined by taking the largest score value in each row.

If `Criterion` is 'CalinskiHarabasz', 'DaviesBouldin', or 'silhouette', you can also specify `clust` as a  $n$ -by- $K$  matrix containing the proposed clustering solutions.  $n$  is the number of observations in the sample data, and  $K$  is the number of proposed clustering solutions. Column  $j$  contains the cluster indices for each of the  $N$  points in the  $j$ th clustering solution.

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'KList', [1:5], 'Distance', 'cityblock'` specifies to test 1, 2, 3, 4, and 5 clusters using the sum of absolute differences distance measure.

### **ClusterPriors** — Prior probabilities for each cluster

'empirical' (default) | 'equal'



Prior probabilities for each cluster, specified as the comma-separated pair consisting of 'ClusterPriors' and one of the following.

'empirical'	Compute the overall silhouette value for the clustering solution by averaging the silhouette values for all points. Each cluster contributes to the overall silhouette value proportionally to its size.
'equal'	Compute the overall silhouette value for the clustering solution by averaging the silhouette values for all points within each cluster, and then averaging those values across all clusters. Each cluster contributes equally to the overall silhouette value, regardless of its size.

Example: 'ClusterPriors', 'empirical'

#### Distance — Distance metric

'sqEuclidean' (default) | 'Euclidean' | 'cityblock' | vector | function | ...

Distance metric used for computing the criterion values, specified as the comma-separated pair consisting of 'Distance' and one of the following.

'sqEuclidean'	Squared Euclidean distance
'Euclidean'	Euclidean distance. This option is not valid for the kmeans clustering algorithm.
'cityblock'	Sum of absolute differences
'cosine'	One minus the cosine of the included angle between points (treated as vectors)
'correlation'	One minus the sample correlation between points (treated as sequences of values)
'Hamming'	Percentage of coordinates that differ. This option is only valid for the Silhouette criterion.
'Jaccard'	Percentage of nonzero coordinates that differ. This option is only valid for the Silhouette criterion.

For detailed information about each distance metric, see pdist.

You can also specify a function for the distance metric using a function handle (MATLAB). The distance function must be of the form  $d2 = \text{distfun}(XI, XJ)$ , where  $XI$  is a 1-by- $n$  vector corresponding to a single row of the input matrix  $X$ , and  $XJ$  is an  $m_2$ -

by- $n$  matrix corresponding to multiple rows of  $X$ . `distfun` must return an  $m_2$ -by-1 vector of distances  $d_2$ , whose  $k$ th element is the distance between  $X_I$  and  $X_J(k, :)$ .

If `Criterion` is 'silhouette', you can also specify `Distance` as the output vector output created by the function `pdist`.

When `Clust` a character vector representing a built-in clustering algorithm, `evalclusters` uses the distance metric specified for `Distance` to cluster the data, except for the following:

- If `Clust` is 'linkage', and `Distance` is either 'sqEuclidean' or 'Euclidean', then the clustering algorithm uses Euclidean distance and Ward linkage.
- If `Clust` is 'linkage' and `Distance` is any other metric, then the clustering algorithm uses the specified distance metric and average linkage.

In all other cases, the distance metric specified for `Distance` must match the distance metric used in the clustering algorithm to obtain meaningful results.

`Distance` only accepts a function handle if the clustering algorithm `clust` accepts a function handle as the distance metric. For example, the `kmeans` clustering algorithm does not accept a function handle as the distance metric. Therefore, if you use the `kmeansalgorithm` and then specify a function handle for `Distance`, the software errors.

Example: 'Distance', 'Euclidean'

**KList** — List of number of clusters to evaluate  
vector

List of number of clusters to evaluate, specified as the comma-separated pair consisting of 'KList' and a vector of positive integer values. You must specify `KList` when `clust` is a clustering algorithm name or a function handle. When `criterion` is 'gap', `clust` must be a character vector or a function handle, and you must specify `KList`.

Example: 'KList', [1:6]

## Properties

### **ClusteringFunction**

Clustering algorithm used to cluster the input data, stored as a valid clustering algorithm name or function handle. If the clustering solutions are provided in the input, `ClusteringFunction` is empty.

### **ClusterPriors**

Prior probabilities for each cluster, stored as valid prior probability name.

### **ClusterSilhouettes**

Silhouette values corresponding to each proposed number of clusters in `InspectedK`, stored as a cell array of vectors.

### **CriterionName**

Name of the criterion used for clustering evaluation, stored as a valid criterion name.

### **CriterionValues**

Criterion values corresponding to each proposed number of clusters in `InspectedK`, stored as a vector of numerical values.

### **Distance**

Distance measure used for clustering data, stored as a valid distance measure name.

### **InspectedK**

List of the number of proposed clusters for which to compute criterion values, stored as a vector of positive integer values.

### **Missing**

Logical flag for excluded data, stored as a column vector of logical values. If `Missing` equals `true`, then the corresponding value in the data matrix  $x$  is not used in the clustering solution.

**NumObservations**

Number of observations in the data matrix  $X$ , minus the number of missing (NaN) values in  $X$ , stored as a positive integer value.

**OptimalK**

Optimal number of clusters, stored as a positive integer value.

**OptimalY**

Optimal clustering solution corresponding to `OptimalK`, stored as a column vector of positive integer values. If the clustering solutions are provided in the input, `OptimalY` is empty.

**x**

Data used for clustering, stored as a matrix of numerical values.

## Methods

### Inherited Methods

<code>addK</code>	Evaluate additional numbers of clusters
<code>plot</code>	Plot clustering evaluation object criterion values
<code>compact</code>	Compact clustering evaluation object

## Examples

### Evaluate the Clustering Solution Using Silhouette Criterion

Evaluate the optimal number of clusters using the silhouette clustering evaluation criterion.

Generate sample data containing random numbers from three multivariate distributions with different parameter values.

```
rng('default'); % For reproducibility
mu1 = [2 2];
sigma1 = [0.9 -0.0255; -0.0255 0.9];

mu2 = [5 5];
sigma2 = [0.5 0 ; 0 0.3];

mu3 = [-2, -2];
sigma3 = [1 0 ; 0 0.9];

N = 200;

X = [mvnrnd(mu1, sigma1, N); ...
     mvnrnd(mu2, sigma2, N); ...
     mvnrnd(mu3, sigma3, N)];
```

Evaluate the optimal number of clusters using the silhouette criterion. Cluster the data using `kmeans`.

```
E = evalclusters(X, 'kmeans', 'silhouette', 'klist', [1:6])
```

```
E =
```

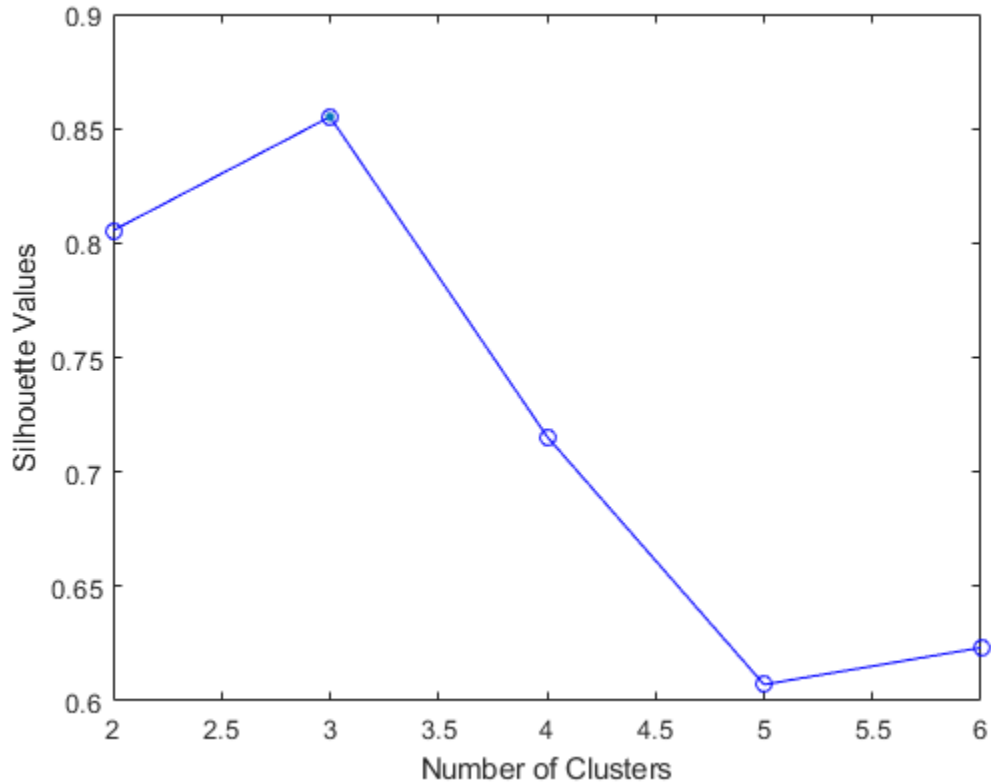
```
SilhouetteEvaluation with properties:
```

```
NumObservations: 600
  InspectedK: [1 2 3 4 5 6]
 CriterionValues: [NaN 0.8055 0.8551 0.7155 0.6071 0.6232]
   OptimalK: 3
```

The `OptimalK` value indicates that, based on the silhouette criterion, the optimal number of clusters is three.

Plot the silhouette criterion values for each number of clusters tested.

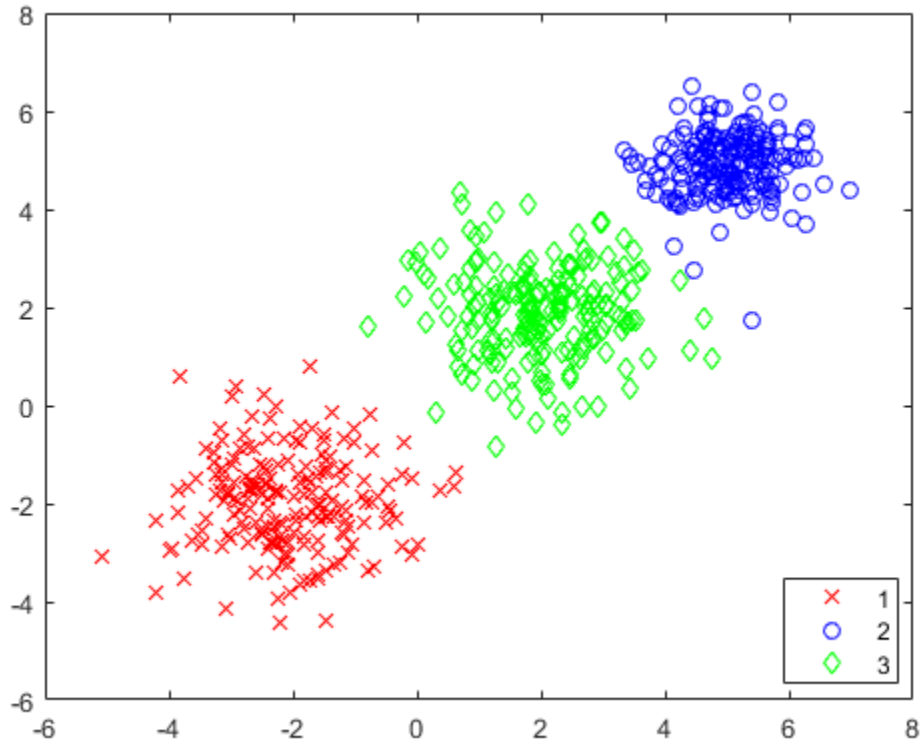
```
figure;
plot(E)
```



The plot shows that the highest silhouette value occurs at three clusters, suggesting that the optimal number of clusters is three.

Create a grouped scatter plot to visually examine the suggested clusters.

```
figure;  
gscatter(X(:,1),X(:,2),E.OptimalY,'rbg','xod')
```



The plot shows three distinct clusters within the data: Cluster 1 is in the lower-left corner, cluster 2 is near the center of the plot, and cluster 3 is in the upper-right corner.

## Definitions

### Silhouette Value

The silhouette value for each point is a measure of how similar that point is to points in its own cluster, when compared to points in other clusters. The silhouette value for the  $i$ th point,  $S_i$ , is defined as

$$S_i = (b_i - a_i) / \max(a_i, b_i)$$

where  $a_i$  is the average distance from the  $i$ th point to the other points in the same cluster as  $i$ , and  $b_i$  is the minimum average distance from the  $i$ th point to points in a different cluster, minimized over clusters.

The silhouette value ranges from -1 to +1. A high silhouette value indicates that  $i$  is well-matched to its own cluster, and poorly-matched to neighboring clusters. If most points have a high silhouette value, then the clustering solution is appropriate. If many points have a low or negative silhouette value, then the clustering solution may have either too many or too few clusters. The silhouette clustering evaluation criterion can be used with any distance metric.

## References

- [1] Kaufman L. and P. J. Rouseeuw. *Finding Groups in Data: An Introduction to Cluster Analysis*. Hoboken, NJ: John Wiley & Sons, Inc., 1990.
- [2] Rouseeuw, P. J. “Silhouettes: a graphical aid to the interpretation and validation of cluster analysis.” *Journal of Computational and Applied Mathematics*. Vol. 20, No. 1, 1987, pp. 53–65.

## See Also

`clustering.evaluation.CalinskiHarabaszEvaluation` |  
`clustering.evaluation.DaviesBouldinEvaluation` |  
`clustering.evaluation.GapEvaluation` | `evalclusters` | `silhouette`

## Topics

Class Attributes (MATLAB)  
Property Attributes (MATLAB)



# single

**Class:** dataset

Convert dataset variables to single array

---

**Note** The `dataset` data type might be removed in a future release. To work with heterogeneous data, use the MATLAB® `table` data type instead. See MATLAB `table` documentation for more information.

---

## Syntax

```
B = single(A)
B = single(A,vars)
```

## Description

`B = single(A)` returns the contents of the dataset `A`, converted to one single array. The classes of the variables in the dataset must support the conversion.

`B = single(A,vars)` returns the contents of the dataset variables specified by `vars`. `vars` is a positive integer, a vector of positive integers, a variable name, a cell array containing one or more variable names, or a logical vector.

## See Also

`dataset` | `double` | `replacedata`

## size

**Class:** dataset

Size of dataset array

---

**Note** The dataset data type might be removed in a future release. To work with heterogeneous data, use the MATLAB® table data type instead. See MATLAB table documentation for more information.

---

## Syntax

```
D = SIZE(A)
[NOBS,NVARS] = SIZE(A)
[M1,M2,M3,...,MN] = SIZE(A)
M = size(A,dim)
```

## Description

`D = SIZE(A)` returns the two-element row vector `D = [NOBS,NVARS]` containing the number of observations and number of variables in the dataset `A`. A dataset array always has two dimensions.

`[NOBS,NVARS] = SIZE(A)` returns the numbers of observations and variables in the dataset `A` as separate output variables.

`[M1,M2,M3,...,MN] = SIZE(A)`, for `N > 2`, returns `M1 = NOBS`, `M2 = NVARS`, and `M3,...,MN = 1`.

`M = size(A,dim)` returns the length of the dimension specified by the scalar `dim`:

- `M = size(A,1)` returns `NOBS`
- `M = size(A,2)` returns `NVARS`
- `M = size(A,k)` returns 1 for `k > 2`

## See Also

`length` | `ndims` | `numel`

## size

**Class:** grandset

Number of dimensions in matrix

## Syntax

```
d = size(p)
[m,n] = size(p)
m = size(p,dim)
```

## Description

`d = size(p)` returns the two-element row vector `d = [m,n]` containing the number of points in the point set and the number of dimensions the points are in, for the point set `p`. These correspond to the number of rows and columns in the matrix that would be produced by the expression `p(:, :)`.

`[m,n] = size(p)` returns the number of points and dimensions for `p` as separate output variables.

`m = size(p,dim)` returns the length of the dimension specified by the scalar `dim`. For example, `size(p,1)` returns the number of rows (points in the point set). If `dim` is greater than 2, `m` will be 1.

## Examples

The commands

```
P = sobolset(12);
d = size(P)

return

d = [9.0072e+015 12]
```

**The command**

```
[m,n] = size(P)
```

**returns**

```
m = 9.0072e+015  
n = 12
```

**The command**

```
m2 = size(P, 2)
```

**returns**

```
m2 = 12
```

**See Also**

`length` | `ndims` | `grandset`

## slicesample

Slice sampler

### Syntax

```
rnd = slicesample(initial, nsamples, 'pdf', pdf)
rnd = slicesample(initial, nsamples, 'logpdf', logpdf)
[rnd, neval] = slicesample(initial, ...)
[rnd, neval] = slicesample(initial, ..., Name, Value)
```

### Description

`rnd = slicesample(initial, nsamples, 'pdf', pdf)` generates `nsamples` random samples using the slice sampling method (see “Algorithms” on page 32-6133). `pdf` gives the target probability density function (pdf). `initial` is a row vector or scalar containing the initial value of the random sample sequences.

`rnd = slicesample(initial, nsamples, 'logpdf', logpdf)` generates samples using the logarithm of the pdf.

`[rnd, neval] = slicesample(initial, ...)` returns the average number of function evaluations that occurred in the slice sampling.

`[rnd, neval] = slicesample(initial, ..., Name, Value)` generates random samples with additional options specified by one or more `Name, Value` pair arguments.

### Input Arguments

#### **initial**

Initial point, a scalar or row vector. Set `initial` so `pdf(initial)` is a strictly positive scalar. `length(initial)` is the number of dimensions of each sample.

**nsamples**

Positive integer, the number of samples that `slicesample` generates.

**pdf**

Handle to a function that generates the probability density function, specified with `@`. `pdf` can be unnormalized, meaning it need not integrate to 1.

**logpdf**

Handle to a function that generates the logarithm of the probability density function, specified with `@`. `logpdf` can be the logarithm of an unnormalized pdf.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

**burnin**

Nonnegative integer, the number of samples to generate and discard before generating the samples to return. The slice sampling algorithm is a Markov chain whose stationary distribution is proportional to that of the `pdf` argument. Set `burnin` to a high enough value that you believe the Markov chain approximately reaches stationarity after `burnin` samples.

**Default:** 0

**thin**

Positive integer, where `slicesample` discards every `thin - 1` samples and returns the next. The slice sampling algorithm is a Markov chain, so the samples are serially correlated. To reduce the serial correlation, choose a larger value of `thin`.

**Default:** 1

**width**

Width of the interval around the current sample, a scalar or vector of positive values. `slicesample` begins with this interval and searches for an appropriate region containing the points of `pdf` that evaluate to a large enough value.

- If `width` is a scalar and the samples have multiple dimensions, `slicesample` uses `width` for each dimension.
- If `width` is a vector, it should have the same length as `initial`.

**Default:** 10

## Output Arguments

**rnd**

`nsamples-by-length(initial)` matrix, where each row is one sample.

**neval**

Scalar, the mean number of function evaluations per sample. `neval` includes the `burnin` and `thin` evaluations, not just the evaluations of samples returned in `rnd`. Therefore the total number of function evaluations is

```
neval*(nsamples*thin + burnin).
```

## Examples

### Generate Random Samples From a Multimodal Density

This example shows how to generate random samples from a multimodal density using `slicesample`.

Define a function proportional to a multimodal density.

```
rng default % For reproducibility
f = @(x) exp(-x.^2/2) .* (1 + (sin(3*x)).^2) .* ...
```



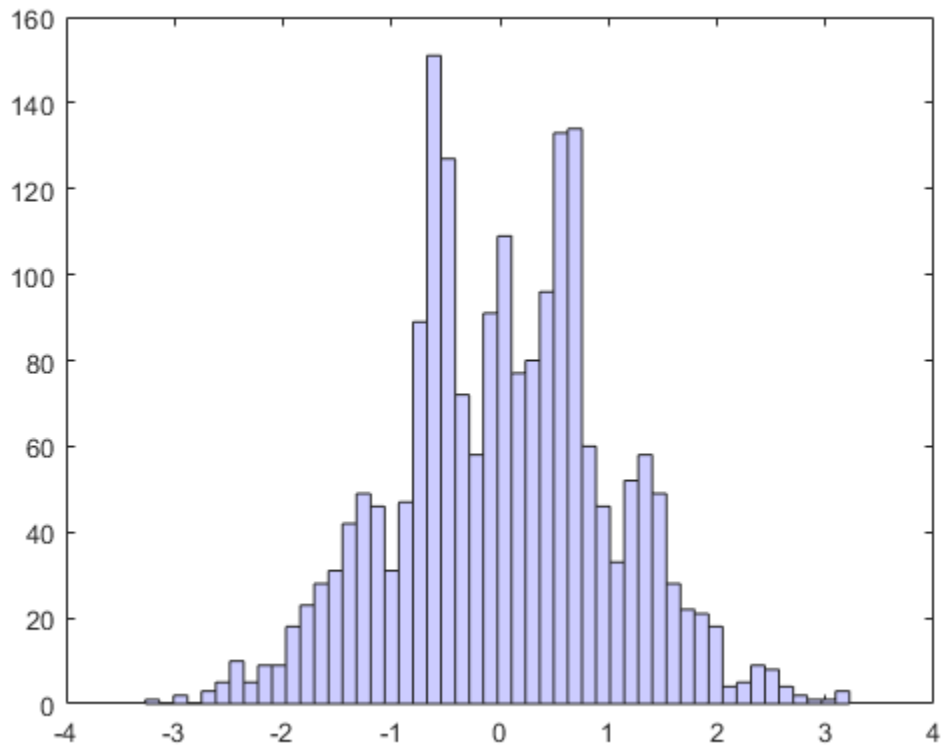
```
(1 + (cos(5*x).^2));  
area = integral(f,-5,5);
```

Generate 2000 samples from the density, using a burn-in period of 1000, and keeping one in five samples.

```
N = 2000;  
x = slicesample(1,N, 'pdf', f, 'thin', 5, 'burnin', 1000);
```

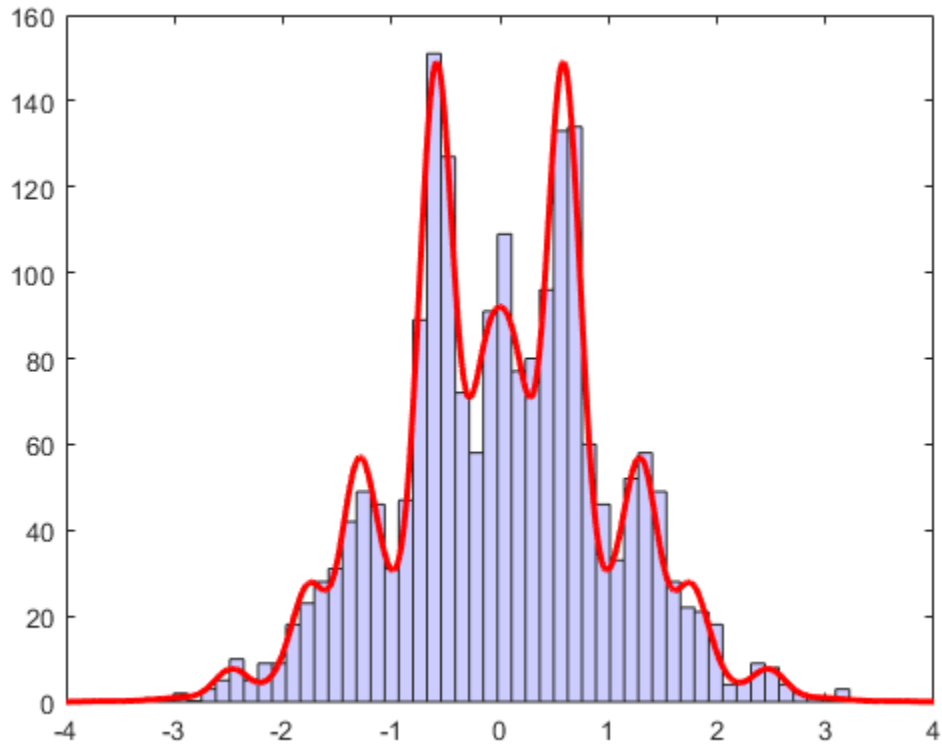
Plot a histogram of the sample.

```
[binheight,bincenter] = hist(x,50);  
h = bar(bincenter,binheight, 'hist');  
h.FaceColor = [.8 .8 1];
```



Scale the density to have the same area as the histogram, and superimpose it on the histogram.

```
hold on
h = gca;
xd = h.XLim;
xgrid = linspace(xd(1),xd(2),1000);
binwidth = (bincenter(2)-bincenter(1));
y = (N*binwidth/area) * f(xgrid);
plot(xgrid,y,'r','LineWidth',2)
hold off
```



The samples seem to fit the theoretical distribution well, so the `burnin` value seems adequate.

## Tips

- There are no definitive suggestions for choosing appropriate values for `burnin`, `thin`, or `width`. Choose starting values of `burnin` and `thin`, and increase them, if necessary, to give the requisite independence and marginal distributions. See Neal [1] for details of the effect of adjusting `width`.

## Algorithms

At each point in the sequence of random samples, `slicesample` selects the next point by “slicing” the density to form a neighborhood around the previous point where the density is above some value. Consequently, the sample points are not independent. Nearby points in the sequence tend to be closer together than they would be from a sample of independent values. For many purposes, the entire set of points can be used as a sample from the target distribution. However, when this type of serial correlation is a problem, the `burnin` and `thin` parameters can help reduce that correlation.

`slicesample` uses the slice sampling algorithm of Neal [1]. For numerical stability, it converts a `pdf` function into a `logpdf` function. The algorithm to resize the support region for each level, called “stepping-out” and “stepping-in,” was suggested by Neal.

## References

- [1] Neal, Radford M. *Slice Sampling*. Ann. Stat. Vol. 31, No. 3, pp. 705–767, 2003.  
Available at Project Euclid.

## See Also

`mhsample` | `rand` | `randsample`

Introduced in R2006a

## skewness

Skewness

### Syntax

```
y = skewness(X)
y = skewness(X, flag)
y = skewness(X, flag, dim)
```

### Description

`y = skewness(X)` returns the sample skewness of `X`. For vectors, `skewness(x)` is the skewness of the elements of `x`. For matrices, `skewness(X)` is a row vector containing the sample skewness of each column. For N-dimensional arrays, `skewness` operates along the first nonsingleton dimension of `X`.

`y = skewness(X, flag)` specifies whether to correct for bias (`flag = 0`) or not (`flag = 1`, the default). When `X` represents a sample from a population, the skewness of `X` is biased; that is, it will tend to differ from the population skewness by a systematic amount that depends on the size of the sample. You can set `flag = 0` to correct for this systematic bias.

`y = skewness(X, flag, dim)` takes the skewness along dimension `dim` of `X`.

`skewness` treats NaNs as missing values and removes them.

### Examples

```
X = randn([5 4])
X =
    1.1650    1.6961   -1.4462   -0.3600
    0.6268    0.0591   -0.7012   -0.1356
    0.0751    1.7971    1.2460   -1.3493
    0.3516    0.2641   -0.6390   -1.2704
```

```
-0.6965  0.8717  0.5774  0.9846
```

```
y = skewness(X)
```

```
y =
```

```
-0.2933  0.0482  0.2735  0.4641
```

## Algorithms

Skewness is a measure of the asymmetry of the data around the sample mean. If skewness is negative, the data are spread out more to the left of the mean than to the right. If skewness is positive, the data are spread out more to the right. The skewness of the normal distribution (or any perfectly symmetric distribution) is zero.

The skewness of a distribution is defined as

$$s = \frac{E(x - \mu)^3}{\sigma^3}$$

where  $\mu$  is the mean of  $x$ ,  $\sigma$  is the standard deviation of  $x$ , and  $E(t)$  represents the expected value of the quantity  $t$ . `skewness` computes a sample version of this population value.

When you set `flag` to 1, the following equation applies:

$$s_1 = \frac{\frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})^3}{\left( \sqrt{\frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})^2} \right)^3}$$

When you set `flag` to 0, the following equation applies:

$$s_0 = \frac{\sqrt{n(n-1)}}{n-2} s_1$$

This bias-corrected formula requires that `X` contain at least three elements.

## Extended Capabilities

### Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays” (MATLAB).

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`kurtosis` | `mean` | `moment` | `std` | `var`

Introduced before R2006a

# Skip property

**Class:** grandset

Number of initial points to omit from sequence

## Description

The `Skip` property of a point set contains a positive integer which specifies the number of initial points in the sequence to omit from the point set. The default `Skip` value is 0.

Initial points of a sequence sometimes exhibit undesirable properties, for example the first point is often  $(0, 0, 0, \dots)$  and this may "unbalance" the sequence since its counterpart,  $(1, 1, 1, \dots)$ , never appears. Another common reason is that initial points often exhibit correlations among different dimensions which disappear later in the sequence.

## Examples

Examine the difference between skipping and not skipping points:

```
% No skipping produces the standard Sobol sequence.  
P = sobolset(5);  
P(1:3, :)
```

```
% Skip the first point of the sequence. The point set now  
% starts at the second point of the basic Sobol sequence.  
P.Skip = 1;  
P(1:3, :)
```

## See Also

[Leap](#) | [net](#) | [grandset](#) | [subsref](#)

## sobolset class

### Superclasses:

Sobol quasi-random point sets

## Description

`sobolset` is a quasi-random point set class that produces points from the Sobol sequence. The Sobol sequence is a base-2 digital sequence that fills space in a highly uniform manner.

## Construction

`sobolset`          Construct Sobol quasi-random point set

## Methods

### Inherited Methods

Methods in the following table are inherited from `grandset` on page 32-5245.

<code>disp</code>	Display <code>grandset</code> object
<code>end</code>	Last index in indexing expression for point set
<code>length</code>	Length of point set
<code>ndims</code>	Number of dimensions in matrix
<code>net</code>	Generate quasi-random point set
<code>scramble</code>	Scramble quasi-random point set
<code>size</code>	Number of dimensions in matrix
<code>suboref</code>	Subscripted reference for <code>grandset</code>



## Properties

PointOrder                      Point generation method

## Inherited Properties

Properties in the following table are inherited from `grandset` on page 32-5245.

Dimensions	Number of dimensions
Leap	Interval between points
ScrambleMethod	Settings that control scrambling
Skip	Number of initial points to omit from sequence
Type	Name of sequence on which point set $P$ is based

## Copy Semantics

Value. To learn how this affects your use of the class, see [Comparing Handle and Value Classes \(MATLAB\)](#) in the MATLAB Object-Oriented Programming documentation.

## References

- [1] Bratley, P., and B. L. Fox, "ALGORITHM 659 Implementing Sobol's Quasirandom Sequence Generator," *ACM Transactions on Mathematical Software*, Vol. 14, No. 1, pp. 88-100, 1988.
- [2] Joe, S., and F. Y. Kuo, "Remark on Algorithm 659: Implementing Sobol's Quasirandom Sequence Generator," *ACM Transactions on Mathematical Software*, Vol. 29, No. 1, pp. 49-57, 2003.
- [3] Hong, H. S., and F. J. Hickernell, "ALGORITHM 823: Implementing Scrambled Digital Sequences," *ACM Transactions on Mathematical Software*, Vol. 29, No. 2, pp. 95-109, 2003.
- [4] Matousek, J., "On the L2-discrepancy for anchored boxes," *Journal of Complexity*, Vol. 14, pp. 527-556, 1998.

## See Also

haltonset

## Topics

“Quasi-Random Point Sets” on page 7-19

# sobolset

**Class:** sobolset

Construct Sobol quasi-random point set

## Syntax

```
p = sobolset(d)
p = sobolset(d, prop1, val1, prop2, val2, ...)
```

## Description

`p = sobolset(d)` constructs a  $d$ -dimensional point set `p` of the `sobolset` on page 32-6138 class, with default property settings.

`p = sobolset(d, prop1, val1, prop2, val2, ...)` specifies property name/value pairs used to construct `p`.

The object `p` returned by `sobolset` encapsulates properties of a specified quasi-random sequence. The point set is finite, with a length determined by the `Skip` and `Leap` properties and by limits on the size of point set indices (maximum value of  $2^{53}$ ). Values of the point set are not generated and stored in memory until you access `p` using `net` or parenthesis indexing.

## Examples

Generate a 3-D Sobol point set, skip the first 1000 values, and then retain every 101st point:

```
p = sobolset(3, 'Skip', 1e3, 'Leap', 1e2)
p =
    Sobol point set in 3 dimensions (8.918019e+013 points)
    Properties:
        Skip : 1000
        Leap : 100
```

```
ScrambleMethod : none
PointOrder : standard
```

Use `scramble` to apply a random linear scramble combined with a random digital shift:

```
p = scramble(p, 'MatousekAffineOwen')
p =
  Sobol point set in 3 dimensions (8.918019e+013 points)
  Properties:
    Skip : 1000
    Leap : 100
  ScrambleMethod : MatousekAffineOwen
  PointOrder : standard
```

Use `net` to generate the first four points:

```
X0 = net(p, 4)
X0 =
  0.7601    0.5919    0.9529
  0.1795    0.0856    0.0491
  0.5488    0.0785    0.8483
  0.3882    0.8771    0.8755
```

Use parenthesis indexing to generate every third point, up to the 11th point:

```
X = p(1:3:11, :)
X =
  0.7601    0.5919    0.9529
  0.3882    0.8771    0.8755
  0.6905    0.4951    0.8464
  0.1955    0.5679    0.3192
```

## References

- [1] Bratley, P., and B. L. Fox. “Algorithm 659 Implementing Sobol's Quasirandom Sequence Generator.” *ACM Transactions on Mathematical Software*. Vol. 14, No. 1, 1988, pp. 88–100.
- [2] Joe, S., and F. Y. Kuo. “Remark on Algorithm 659: Implementing Sobol's Quasirandom Sequence Generator.” *ACM Transactions on Mathematical Software*. Vol. 29, No. 1, 2003, pp. 49–57.

- [3] Hong, H. S., and F. J. Hickernell. “Algorithm 823: Implementing Scrambled Digital Sequences.” *ACM Transactions on Mathematical Software*. Vol. 29, No. 2, 2003, pp. 95–109.
- [4] Matousek, J. “On the L2-Discrepancy for Anchored Boxes.” *Journal of Complexity*. Vol. 14, No. 4, 1998, pp. 527–556.

## See Also

haltonset | net | scramble

**Introduced in R2008a**

## sortrows

**Class:** dataset

Sort rows of dataset array

---

**Note** The `dataset` data type might be removed in a future release. To work with heterogeneous data, use the MATLAB® `table` data type instead. See MATLAB `table` documentation for more information.

---

## Syntax

```
B = sortrows(A)
B = sortrows(A,vars)
B = sortrows(A,'obsnames')
B = sortrows(A,vars,mode)
[B,idx] = sortrows(A)
```

## Description

`B = sortrows(A)` returns a copy of the dataset array `A`, with the observations sorted in ascending order by all of the variables in `A`. The observations in `B` are sorted first by the first variable, next by the second variable, and so on. Each variable in `A` must be a valid input to `sort`, or, if a variable has multiple columns, to the MATLAB `sortrows` function or to its `ownsorrows` method.

`B = sortrows(A,vars)` sorts the observations in `A` by the variables specified by `vars`. `vars` is a positive integer, a vector of positive integers, variable names, a cell array containing one or more variable names, or a logical vector.

`B = sortrows(A,'obsnames')` sorts the observations in `A` by observation name.

`B = sortrows(A,vars,mode)` sorts in the direction specified by `mode`. When `mode` is `'ascend'` (the default) or `'descend'`, `sortrows` sorts `A` by the variables specified by `vars` in ascending or descending order, respectively. `mode` can also be a cell array

containing 'ascend' or 'descend', to specify a different sorting direction for each variable in vars. Specify [] for vars to sort using all variables.

[B,idx] = sortrows(A) also returns an index vector idx such that B = A(idx,:).

## Examples

Sort the data in hospital.mat by age and then by last name:

```
load hospital
hospital(1:5,1:3)
ans =
```

	LastName	Sex	Age
YPL-320	'SMITH'	Male	38
GLI-532	'JOHNSON'	Male	43
PNI-258	'WILLIAMS'	Female	38
MIJ-579	'JONES'	Female	40
XLK-030	'BROWN'	Female	49

```
hospital = sortrows(hospital,{'Age','LastName'});
hospital(1:5,1:3)
ans =
```

	LastName	Sex	Age
REV-997	'ALEXANDER'	Male	25
FZR-250	'HALL'	Male	25
LIM-480	'HILL'	Female	25
XUE-826	'JACKSON'	Male	25
SCQ-914	'JAMES'	Male	25

Sort the data in hospital by gender in ascending order, and age in descending order.

```
hospital = sortrows(hospital,{'Sex','Age'},{'ascend','descend'});
hospital(1:5,1:3)
ans =
```

	LastName	Sex	Age
XLK-030	'BROWN'	Female	49
GGU-691	'HUGHES'	Female	49
KKL-155	'ADAMS'	Female	48
HQO-561	'BRYANT'	Female	48
BKD-785	'CLARK'	Female	48

```
hospital(end-4:end,1:3)
ans =
```

	LastName	Sex	Age
VNL-702	'MOORE'	Male	28
REV-997	'ALEXANDER'	Male	25
FZR-250	'HALL'	Male	25
XUE-826	'JACKSON'	Male	25
SCQ-914	'JAMES'	Male	25

## See Also

dataset | unique

## Topics

“Sort Observations in Dataset Arrays” on page 2-103

“Dataset Arrays” on page 2-142



# sparsefilt

Feature extraction by using sparse filtering

## Syntax

```
Mdl = sparsefilt(X,q)
Mdl = sparsefilt(X,q,Name,Value)
```

## Description

`Mdl = sparsefilt(X,q)` returns a sparse filtering model object that contains the results from applying sparse filtering to the table or matrix of predictor data `X` containing  $p$  variables. `q` is the number of features to extract from `X`, therefore `sparsefilt` learns a  $p$ -by- $q$  matrix of transformation weights. For undercomplete or overcomplete feature representations, `q` can be less than or greater than the number of predictor variables, respectively.

- To access the learned transformation weights, use `Mdl.TransformWeights`.
- To transform `X` to the new set of features by using the learned transformation, pass `Mdl` and `X` to `transform`.

`Mdl = sparsefilt(X,q,Name,Value)` uses additional options specified by one or more `Name, Value` pair arguments. For example, you can standardize the predictor data or apply  $L^2$  regularization.

## Examples

### Create Sparse Filter

Create a `SparseFiltering` object by using the `sparsefilt` function.

Load the `caltech101` image patches.

```
data = load('caltech101patches');
size(data.X)
```

```
ans =

    100000    363
```

There are 100,000 image patches, each containing 363 features.

Extract 100 features from the data.

```
rng default % For reproducibility
Q = 100;
obj = sparsefilt(data.X,Q,'IterationLimit',100)
```

Warning: Solver LBFGS was not able to converge to a solution.

```
obj =

    SparseFiltering
      ModelParameters: [1x1 struct]
      NumPredictors: 363
      NumLearnedFeatures: 100
              Mu: []
              Sigma: []
      FitInfo: [1x1 struct]
      TransformWeights: [363x100 double]
  InitialTransformWeights: []
```

`sparsefilt` issues a warning because it stopped due to reaching the iteration limit, instead of reaching a step-size limit or a gradient-size limit. You can still use the learned features in the returned object by calling the `transform` function.

### Restart `sparsefilt`

Continue optimizing a sparse filter.

Load the `caltech101` image patches.

```
data = load('caltech101patches');
size(data.X)
```

```
ans =
    100000    363
```

There are 100,000 image patches, each containing 363 features.

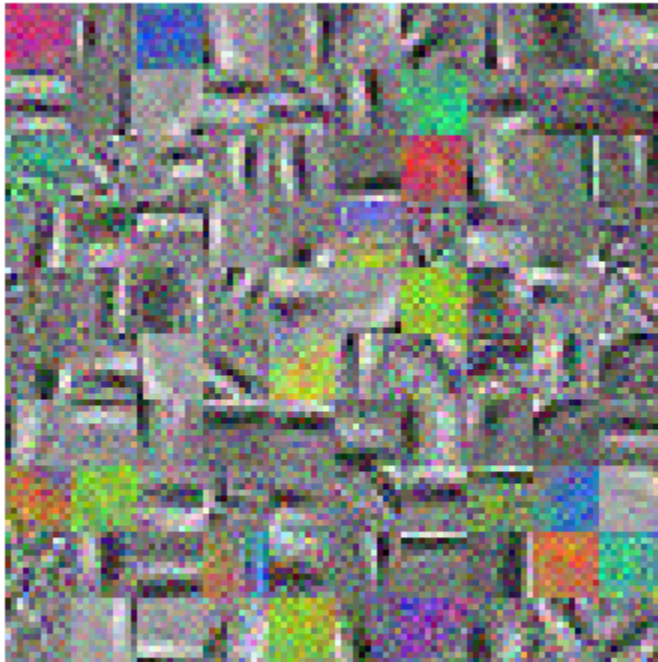
Extract 100 features from the data and use an iteration limit of 20.

```
rng default % For reproducibility
q = 100;
Mdl = sparsefilt(data.X,q,'IterationLimit',20);
```

Warning: Solver LBFGS was not able to converge to a solution.

View the resulting transformation matrix as image patches.

```
wts = Mdl.TransformWeights;
W = reshape(wts, [11,11,3,q]);
[dx,dy,~,~] = size(W);
for f = 1:q
    Wvec = W(:,:,,f);
    Wvec = Wvec(:);
    Wvec = (Wvec - min(Wvec)) / (max(Wvec) - min(Wvec));
    W(:,:,,f) = reshape(Wvec,dx,dy,3);
end
m = ceil(sqrt(q));
n = m;
img = zeros(m*dx,n*dy,3);
f = 1;
for i = 1:m
    for j = 1:n
        if (f <= q)
            img((i-1)*dx+1:i*dx, (j-1)*dy+1:j*dy,:) = W(:,:,,f);
            f = f+1;
        end
    end
end
end
imshow(img,'InitialMagnification',300);
```



The image patches appear noisy. To clean up the noise, try more iterations. Restart the optimization from where it stopped for another 40 iterations.

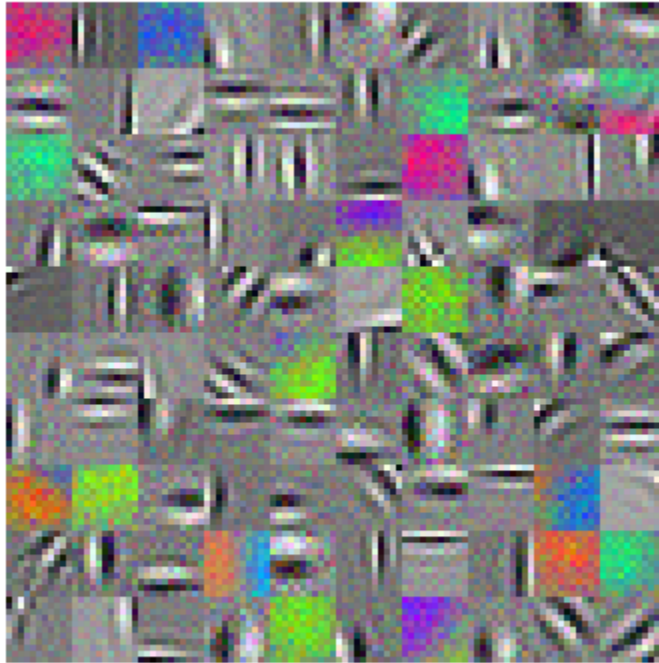
```
Mdl = sparsefilt(data.X,q,'IterationLimit',40,'InitialTransformWeights',wts);
```

```
Warning: Solver LBFGS was not able to converge to a solution.
```

View the updated transformation matrix as image patches.

```
wts = Mdl.TransformWeights;  
W = reshape(wts,[11,11,3,q]);  
[dx,dy,~,~] = size(W);  
for f = 1:q  
    Wvec = W(:, :, :, f);
```

```
Wvec = Wvec(:);
Wvec = (Wvec - min(Wvec)) / (max(Wvec) - min(Wvec));
W(:,:,,f) = reshape(Wvec,dx,dy,3);
end
m = ceil(sqrt(q));
n = m;
img = zeros(m*dx,n*dy,3);
f = 1;
for i = 1:m
    for j = 1:n
        if (f <= q)
            img((i-1)*dx+1:i*dx,(j-1)*dy+1:j*dy,:) = W(:,:,,f);
            f = f+1;
        end
    end
end
end
imshow(img,'InitialMagnification',300);
```



These images are much less noisy.

- “Feature Extraction Workflow” on page 15-169

## Input Arguments

### **x** — Predictor data

numeric matrix | table

Predictor data, specified as an  $n$ -by- $p$  numeric matrix or table. Rows correspond to individual observations and columns correspond to individual predictor variables. If  $x$  is a table, then all of its variables must be numeric vectors.

Data Types: `single` | `double` | `table`

### **q** — Number of features to extract

positive integer

Number of features to extract from the predictor data, specified as a positive integer.

`sparsefilt` stores a  $p$ -by- $q$  transform weight matrix in `Mdl.TransformWeights`. Therefore, setting very large values for  $q$  can result in greater memory consumption and increased computation time.

Data Types: `single` | `double`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'Standardize', true, 'Lambda', 1` standardizes the predictor data and applies a penalty of 1 to the transform weight matrix.

### **IterationLimit** — Maximum number of iterations

1000 (default) | positive integer

Maximum number of iterations, specified as the comma-separated pair consisting of `'IterationLimit'` and a positive integer.

Example: `'IterationLimit', 1e6`

Data Types: `single` | `double`

### **VerbosityLevel** — Verbosity level

0 (default) | nonnegative integer

Verbosity level for monitoring algorithm convergence, specified as the comma-separated pair consisting of `'VerbosityLevel'` and a value in this table.

Value	Description
0	<code>sparsefilt</code> does not display convergence information at the command line.

Value	Description
Positive integer	<code>sparsefilt</code> displays convergence information at the command line.

**Convergence Information**

Heading	Meaning
FUN VALUE	Objective function value.
NORM GRAD	Norm of the gradient of the objective function.
NORM STEP	Norm of the iterative step, meaning the distance between the previous point and the current point.
CURV	OK means the weak Wolfe condition is satisfied. This condition is a combination of sufficient decrease of the objective function and a curvature condition.
GAMMA	Inner product of the step times the gradient difference, divided by the inner product of the gradient difference with itself. The gradient difference is the gradient at the current point minus the gradient at the previous point. Gives diagnostic information on the objective function curvature.
ALPHA	Step direction multiplier, which differs from 1 when the algorithm performed a line search.
ACCEPT	YES means the algorithm found an acceptable step to take.

Example: `'VerbosityLevel', 1`

Data Types: `single` | `double`

**Lambda** —  $L^2$  regularization coefficient value

0 (default) | positive numeric scalar

$L^2$  regularization coefficient value for the transform weight matrix, specified as the comma-separated pair consisting of `'Lambda'` and a positive numeric scalar. If you specify 0, the default, then there is no regularization term in the objective function.

Example: `'Lambda', 0.1`

Data Types: `single` | `double`

**Standardize** — Flag to standardize predictor data

`false` (default) | `true`



Flag to standardize the predictor data, specified as the comma-separated pair consisting of 'Standardize' and true (1) or false (0).

If Standardize is true, then:

- sparsefilt centers and scales each column of the predictor data ( $X$ ) by the column mean and standard deviation, respectively.
- sparsefilt extracts new features by using the standardized predictor matrix, and stores the predictor variable means and standard deviations in properties Mu and Sigma of Mdl.

Example: 'Standardize', true

Data Types: logical

#### **InitialTransformWeights** — Transformation weights that initialize optimization

randn(p,q) (default) | numeric matrix

Transformation weights that initialize optimization, specified as the comma-separated pair consisting of 'InitialTransformWeights' and a  $p$ -by- $q$  numeric matrix.  $p$  must be the number of columns or variables in  $X$  and  $q$  is the value of  $q$ .

---

**Tip** You can continue optimizing a previously returned transform weight matrix by passing it as an initial value in another call to sparsefilt. The output model object Mdl stores a learned transform weight matrix in the TransformWeights property.

---

Example: 'InitialTransformWeights', Mdl.TransformWeights

Data Types: single | double

#### **GradientTolerance** — Relative convergence tolerance on gradient norm

1e-6 (default) | positive numeric scalar

Relative convergence tolerance on gradient norm, specified as the comma-separated pair consisting of 'GradientTolerance' and a positive numeric scalar. This gradient is the gradient of the objective function.

Example: 'GradientTolerance', 1e-4

Data Types: single | double

**StepTolerance** — Absolute convergence tolerance on step size`1e-6` (default) | positive numeric scalar

Absolute convergence tolerance on the step size, specified as the comma-separated pair consisting of 'StepTolerance' and a positive numeric scalar.

Example: 'StepTolerance', 1e-4

Data Types: `single` | `double`

## Output Arguments

**Mdl** — Learned sparse filtering model`SparseFiltering` model object

Learned sparse filtering model, returned as a `SparseFiltering` model object.

To access properties of `Mdl`, use dot notation. For example:

- To access the learned transform weights, use `Mdl.TransformWeights`.
- To access the fitting information structure, use `Mdl.FitInfo`.

To find sparse filtering coefficients for new data, use the `transform` function.

## Algorithms

The `sparsefilt` function creates a nonlinear transformation of input features to output features. The transformation is based on optimizing an objective function that encourages the representation of each example by as few output features as possible while at the same time keeping the output features equally active across examples.

For details, see “Sparse Filtering Algorithm” on page 15-163.

## See Also

`SparseFiltering` | `rica` | `transform`

## Topics

“Feature Extraction Workflow” on page 15-169

“Feature Extraction” on page 15-163

**Introduced in R2017a**

# SparseFiltering

Feature extraction by sparse filtering

## Description

`SparseFiltering` uses sparse filtering to learn a transformation that maps input predictors to new predictors.

## Creation

Create a `SparseFiltering` object using the `sparsefilt` function.

## Properties

### `FitInfo` — Fitting history

structure

Fitting history, returned as a structure with two fields:

- `Iteration` — Iteration numbers from 0 through the final iteration.
- `Objective` — Objective function value at each corresponding iteration. Iteration 0 corresponds to the initial values, before any fitting.

Data Types: `struct`

### `InitialTransformWeights` — Initial feature transformation weights

p-by-q matrix

Initial feature transformation weights, returned as a p-by-q matrix, where p is the number of predictors passed in `X` and q is the number of features that you want. These weights are the initial weights passed to the creation function. The data type is `single` when the training data `X` is `single`.

Data Types: `single` | `double`

**ModelParameters** — Parameters used for training model

structure

Parameters used for training the model, returned as a structure. The structure contains a subset of the fields that corresponds to the `sparsefilt` name-value pairs that were in effect during model creation:

- `IterationLimit`
- `VerbosityLevel`
- `Lambda`
- `Standardize`
- `GradientTolerance`
- `StepTolerance`

For details, see the `sparsefilt` name-value pairs in the documentation.

Data Types: `struct` **$\mu$**  — Predictor means when standardizing $p$ -by-1 vector

Predictor means when standardizing, returned as a  $p$ -by-1 vector. This property is nonempty when the `Standardize` name-value pair is `true` at model creation. The value is the vector of predictor means in the training data. The data type is `single` when the training data  $X$  is `single`.

Data Types: `single` | `double`**NumLearnedFeatures** — Number of output features

positive integer

Number of output features, returned as a positive integer. This value is the `q` argument passed to the creation function, which is the requested number of features to learn.

Data Types: `double`**NumPredictors** — Number of input predictors

positive integer

Number of input predictors, returned as a positive integer. This value is the number of predictors passed in  $X$  to the creation function.

Data Types: `double`

**Sigma** — Predictor standard deviations when standardizing

`p-by-1` vector

Predictor standard deviations when standardizing, returned as a `p-by-1` vector. This property is nonempty when the `Standardize` name-value pair is `true` at model creation. The value is the vector of predictor standard deviations in the training data. The data type is `single` when the training data `X` is `single`.

Data Types: `single` | `double`

**TransformWeights** — Feature transformation weights

`p-by-q` matrix

Feature transformation weights, returned as a `p-by-q` matrix, where `p` is the number of predictors passed in `X` and `q` is the number of features that you want. The data type is `single` when the training data `X` is `single`.

Data Types: `single` | `double`

## Object Functions

`transform` Transform predictors into extracted features

## Examples

### Create Sparse Filter

Create a `SparseFiltering` object by using the `sparsefilt` function.

Load the `caltech101` image patches.

```
data = load('caltech101patches');  
size(data.X)
```

```
ans =
```

```
100000      363
```

There are 100,000 image patches, each containing 363 features.

Extract 100 features from the data.

```
rng default % For reproducibility
Q = 100;
obj = sparsefilt(data.X,Q, 'IterationLimit',100)
```

Warning: Solver LBFGS was not able to converge to a solution.

```
obj =
```

```

SparseFiltering
  ModelParameters: [1x1 struct]
  NumPredictors: 363
  NumLearnedFeatures: 100
  Mu: []
  Sigma: []
  FitInfo: [1x1 struct]
  TransformWeights: [363x100 double]
  InitialTransformWeights: []
```

`sparsefilt` issues a warning because it stopped due to reaching the iteration limit, instead of reaching a step-size limit or a gradient-size limit. You can still use the learned features in the returned object by calling the `transform` function.

- “Feature Extraction Workflow” on page 15-169

## See Also

`rica` | `sparsefilt` | `transform`

## Topics

“Feature Extraction Workflow” on page 15-169

“Feature Extraction” on page 15-163

**Introduced in R2017a**



# squareform

Format distance matrix

## Syntax

```
Z = squareform(y)
y = squareform(Z)
Z = squareform(y, 'tovector')
Y = squareform(Z, 'tomatrix')
```

## Description

`Z = squareform(y)`, where `y` is a vector as created by the `pdist` function, converts `y` into a square, symmetric format `Z`, in which `Z(i, j)` denotes the distance between the `i`th and `j`th objects in the original data.

`y = squareform(Z)`, where `Z` is a square, symmetric matrix with zeros along the diagonal, creates a vector `y` containing the `Z` elements below the diagonal. `y` has the same format as the output from the `pdist` function.

`Z = squareform(y, 'tovector')` forces `squareform` to treat `y` as a vector.

`Y = squareform(Z, 'tomatrix')` forces `squareform` to treat `Z` as a matrix.

The last two formats are useful if the input has a single element, so that it is ambiguous whether the input is a vector or square matrix.

## Examples

```
y = 1:6
y =
     1     2     3     4     5     6

X = [0 1 2 3; 1 0 4 5; 2 4 0 6; 3 5 6 0]
```

```
X =  
  0  1  2  3  
  1  0  4  5  
  2  4  0  6  
  3  5  6  0
```

Then `squareform(y) = X` and `squareform(X) = y`.

## See Also

`pdist`

**Introduced before R2006a**

# stack

**Class:** dataset

Stack data from multiple variables into single variable

---

**Note** The `dataset` data type might be removed in a future release. To work with heterogeneous data, use the MATLAB® `table` data type instead. See MATLAB `table` documentation for more information.

---

## Syntax

```
B = stack(A,datavars)
[B,iA] = stack(A,datavars)
B = stack(A,datavars,Parameter,value)
```

## Description

`B = stack(A,datavars)` stacks multiple variables in dataset array `A` into a single variable in `B`. In general, `B` contains fewer variables but more observations than `A`.

`datavars` specifies a group of `m` data variables in `A`. `stack` creates a single data variable in `B` by interleaving their values, and if `A` has `n` observations, then `B` has `m-by-n` observations. In other words, `stack` takes the `m` data values from each observation in `A` and stacks them up to create `m` observations in `B`. `datavars` is a positive integer, a vector of positive integers, a variable name, a cell array containing one or more variable names, or a logical vector. `stack` also creates a grouping variable in `B` to indicate which of the `m` data variables in `A` each observation in `B` corresponds to.

`stack` assigns values for the "per-variable properties (e.g., `Units` and `VarDescription`) for the new data variable in `B` from the corresponding property values for the first variable listed in `datavars`.

`stack` copies the remaining variables from `A` to `B` without stacking, by replicating each of their values `m` times. These variables are typically grouping variables. Because their

values are constant across each group of  $m$  observations in  $B$ , they identify which observation in  $A$  an observation in  $B$  came from.

`[B, iA] = stack(A, datavars)` returns an index vector  $iA$  indicating the correspondence between observations in  $B$  and those in  $A$ . `stack` creates  $B(j, :)$  using `A(iA(j), datavarss)`.

For more information on grouping variables, see “Grouping Variables” on page 2-59.

## Input Arguments

`B = stack(A, datavars, Parameter, value)` uses the following parameter name/value pairs to control how `stack` converts variables in  $A$  to variables in  $B$ :

'ConstVars'	Variables in $A$ to copy to $B$ without stacking. <code>ConstVars</code> is a positive integer, a vector of positive integers, a variable name, a cell array containing one or more variable names, or a logical vector. The default is all variables in $A$ not specified in <code>datavars</code> .
'NewDataVarName'	A name for the data variable to be created in $B$ . The default is a concatenation of the names of the $m$ variables that are stacked up.
'IndVarName'	A name for the grouping variable to create in $B$ to indicate the source of each value in the new data variable. The default is based on the 'NewDataVarName' parameter.

You can also specify multiple groups of data variables in  $A$ , each of which becomes a variable in  $B$ . All groups must contain the same number of variables. Use a cell array to contain multiple parameter values for `datavars`, and a cell array of character vectors to contain multiple 'NewDataVarName'.

## Examples

Combine several variables for estimated influenza rates into a single variable. Then unstack the estimated influenza rates by date.

```
load flu

% FLU has a 'Date' variable, and 10 variables for estimated influenza rates
% (in 9 different regions, estimated from Google searches, plus a
% nationwide estimate from the CDC). Combine those 10 variables into an
% array that has a single data variable, 'FluRate', and an indicator
% variable, 'Region', that says which region each estimate is from.
[flu2,iflu] = stack(flu, 2:11, 'NewDataVarName','FluRate', ...
    'IndVarName','Region')

% The second observation in FLU is for 10/16/2005. Find the observations
% in FLU2 that correspond to that date.
flu(2,:)
flu2(iflu==2,:)

% Use the 'Date' variable from that array to split 'FluRate' into 52
% separate variables, each containing the estimated influenza rates for
% each unique date. The new array has one observation for each region. In
% effect, this is the original array FLU "on its side".
dateNames = cellstr(datestr(flu.Date,'mmm_DD_YYYY'));
[flu3,iflu2] = unstack(flu2, 'FluRate', 'Date', ...
    'NewDataVarNames',dateNames)

% Since observations in FLU3 represent regions, IFLU2 indicates the first
% occurrence in FLU2 of each region.
flu2(iflu2,:)
```

## See Also

`dataset.join` | `dataset.unstack`

## Topics

“Grouping Variables” on page 2-59

## State property

**Class:** grandstream

Current state of the stream

## Description

The `State` property of a quasi-random stream contains the index into the associated point set of the next point to draw in the stream. Getting and resetting the `State` property allows you to return a stream to a previous state. The initial value of `State` is 1.

## Examples

```
Q = grandstream('sobol', 5);
s = Q.State;
u1 = grand(Q, 10)
Q.State = s;
u2 = grand(Q, 10) % contains exactly the same values as u1
```

## See Also

`grand`

# statget

Access values in statistics options structure

## Syntax

```
val = statget(options,param)
val = statget(options,param,default)
```

## Description

`val = statget(options,param)` returns the value of the parameter specified by the character vector `param` in the statistics options structure `options`. If the parameter is undefined in `options`, `statget` returns `[]`. You need to type only enough leading characters to define the parameter name uniquely. `statget` ignores case for parameter names. For available options, see [Inputs](#).

`val = statget(options,param,default)` returns `default` if the specified parameter is undefined in the optimization options structure `options`.

## Input Arguments

### **DerivStep**

Relative difference used in finite difference derivative calculations. A positive scalar, or a vector of positive scalars the same size as the vector of parameters estimated by the Statistics and Machine Learning Toolbox function using the options structure.

### **Display**

Amount of information displayed by the algorithm.

- 'off' — Displays no information.
- 'final' — Displays the final output.

- `'iter'` — Displays iterative output to the command window for some functions; otherwise displays the final output.

**FunValCheck**

Check for invalid values, such as NaN or Inf, from the objective function.

- `'off'`
- `'on'`

**GradObj**

Flags whether the objective function returns a gradient vector as a second output.

- `'off'`
- `'on'`

**Jacobian**

Flags whether the objective function returns a Jacobian as a second output.

- `'off'`
- `'on'`

**MaxFunEvals**

Maximum number of objective function evaluations allowed. Positive integer.

**MaxIter**

Maximum number of iterations allowed. Positive integer.

**OutputFcn**

The solver calls all output functions after each iteration.

- Function handle specified using `@`
- a cell array with function handles
- an empty array (default)



**Robust**

Invoke robust fitting option.

- 'off'
- 'on'

**RobustWgtFun**

A weight function for robust fitting. Valid only when `Robust` is 'on'. Can also be a function handle that accepts a normalized residual as input and returns the robust weights as output.

- 'bisquare'
- 'andrews'
- 'cauchy'
- 'fair'
- 'huber'
- 'logistic'
- 'talwar'
- 'welsch'

**Streams**

A single instance of the `RandStream` class, or a cell array of `RandStream` instances. The `Streams` option is accepted by some functions to govern what stream(s) to use in generating random numbers within the function. If 'UseSubstreams' is true, the `Streams` value must be a scalar, or must be empty. If 'UseParallel' is true and 'UseSubstreams' is false, then the `Streams` argument must either be empty, or its length must match the number of processors used in the computation: equal to the *parpool* size if a *parpool* is open, a scalar otherwise.

**TolBnd**

Parameter bound tolerance. Positive scalar.

**TolFun**

Termination tolerance for the objective function value. Positive scalar.

**TolTypeFun**

Use `TolFun` for absolute or relative objective function tolerances.

- 'abs'
- 'rel'

**TolTypeX**

Use `TolX` for absolute or relative parameter tolerances.

- 'abs'
- 'rel'

**TolX**

Termination tolerance for the parameters. Positive scalar.

**Tune**

The tuning constant used in robust fitting to normalize the residuals before applying the weight function. The default value depends upon the weight function. This parameter is necessary if you specify the weight function as a function handle. Positive scalar.

**UseParallel**

Flag indicating whether eligible functions should use capabilities of the Parallel Computing Toolbox (PCT), if the capabilities are available. That is, if the PCT is installed, and a PCT parpool is in effect. Valid values are `false` (the default), for serial computation, and `true`, for parallel computation.

**UseSubstreams**

Flag indicating whether the random number generator in eligible functions should use `Substream` property of the `RandStream` class. `false` (default) or `true`. When `true`, high level iterations within the function will set the `Substream` property to the value of the iteration. This behavior helps to generate reproducible random number streams in parallel and/or serial mode computation.

### **WgtFun**

A weight function for robust fitting. Valid only when `Robust` is 'on'. Can also be a function handle that accepts a normalized residual as input and returns the robust weights as output.

- 'bisquare'
- 'andrews'
- 'cauchy'
- 'fair'
- 'huber'
- 'logistic'
- 'talwar'
- 'welsch'

## **Examples**

This statement returns the value of the `Display` statistics options parameter from the structure called `my_options`.

```
val = statget(my_options, 'Display')
```

Return the value of the `Display` statistics options parameter from the structure called `my_options` (as in the previous example). If the `Display` parameter is undefined, `statget` returns the value 'final'.

```
optnew = statget(my_options, 'Display', 'final');
```

## **See Also**

`statset`

**Introduced before R2006a**

## statset

Create statistics options structure

### Syntax

```
statset
statset(statfun)
options = statset(...)
options = statset(fieldname1, val1, fieldname2, val2, ...)
options = statset(olddopts, fieldname1, val1, fieldname2, val2, ...)
options = statset(olddopts, newopts)
```

### Description

`statset` with no input arguments and no output arguments displays all fields of a statistics options structure and their possible values.

`statset(statfun)` displays fields and default values used by the Statistics and Machine Learning Toolbox function `statfun`. Specify `statfun` using a character vector or a function handle.

`options = statset(...)` creates a statistics options structure `options`. With no input arguments, all fields of the options structure are an empty array (`[]`). With a specified `statfun`, function-specific fields are default values and the remaining fields are `[]`. Function-specific fields set to `[]` indicate that the function is to use its default value for that parameter. For available options, see [Inputs](#).

`options = statset(fieldname1, val1, fieldname2, val2, ...)` creates an options structure in which the named fields have the specified values. Any unspecified values are `[]`. Use character vectors for field names. For fields that are character-valued, you must input the complete character vector for the value. If you provide an invalid character vector for a value, `statset` uses the default.

`options = statset(olddopts, fieldname1, val1, fieldname2, val2, ...)` creates a copy of `olddopts` with the named parameters changed to the specified values.

`options = statset(oldopts,newopts)` combines an existing options structure, `oldopts`, with a new options structure, `newopts`. Any parameters in `newopts` with nonempty values overwrite corresponding parameters in `oldopts`.

## Input Arguments

### **DerivStep**

Relative difference used in finite difference derivative calculations. A positive scalar, or a vector of positive scalars the same size as the vector of parameters estimated by the Statistics and Machine Learning Toolbox function using the options structure.

### **Display**

Amount of information displayed by the algorithm.

- 'off' — Displays no information.
- 'final' — Displays the final output.
- 'iter' — Displays iterative output to the command window for some functions; otherwise displays the final output.

### **FunValCheck**

Check for invalid values, such as NaN or Inf, from the objective function.

- 'off'
- 'on'

### **GradObj**

Flags whether the objective function returns a gradient vector as a second output.

- 'off'
- 'on'

### **Jacobian**

Flags whether the objective function returns a Jacobian as a second output.

- 'off'
- 'on'

**MaxFunEvals**

Maximum number of objective function evaluations allowed. Positive integer.

**MaxIter**

Maximum number of iterations allowed. Positive integer.

**OutputFcn**

The solver calls all output functions after each iteration.

- Function handle specified using @
- a cell array with function handles
- an empty array (default)

**Robust**

Invoke robust fitting option.

- 'off'
- 'on'

Robust will be removed in a future software release. Use `RobustWgtFun` for robust fitting.

**RobustWgtFun**

Weight function for robust fitting. Can also be a function handle that accepts a normalized residual as input and returns the robust weights as output. If you use a function handle, give a `Tune` constant. See “Robust Options” on page 32-6179.

**Streams**

A single instance of the `RandStream` class, or a cell array of `RandStream` instances. The `Streams` option is accepted by some functions to govern what stream(s) to use in generating random numbers within the function. If `'UseSubstreams'` is true, the `Streams` value must be a scalar, or must be empty. If `'UseParallel'` is true and

---

'UseSubstreams' is *false*, then the `Streams` argument must either be empty, or its length must match the number of processors used in the computation: equal to the *parpool* size if a *parpool* is open, a scalar otherwise.

**TolBnd**

Parameter bound tolerance. Positive scalar.

**TolFun**

Termination tolerance for the objective function value. Positive scalar.

**TolTypeFun**

Use `TolFun` for absolute or relative objective function tolerances.

- 'abs'
- 'rel'

**TolTypeX**

Use `TolX` for absolute or relative parameter tolerances.

- 'abs'
- 'rel'

**TolX**

Termination tolerance for the parameters. Positive scalar.

**Tune**

Tuning constant used in robust fitting to normalize the residuals before applying the weight function. The default value depends upon the weight function. This parameter is necessary if you specify the weight function as a function handle. Positive scalar. See “Robust Options” on page 32-6179.

**UseParallel**

Flag indicating whether eligible functions should use capabilities of the Parallel Computing Toolbox (PCT), if the capabilities are available. That is, if the PCT is installed, and a PCT *parpool* is in effect. Valid values are *false* (the default), for serial computation, and *true*, for parallel computation.

**UseSubstreams**

Flag indicating whether the random number generator in eligible functions should use `Substream` property of the `RandStream` class. `false` (default) or `true`. When `true`, high level iterations within the function will set the `Substream` property to the value of the iteration. This behavior helps to generate reproducible random number streams in parallel and/or serial mode computation.

**WgtFun**

Weight function for robust fitting. Valid only when `Robust` is `'on'`. Can also be a function handle that accepts a normalized residual as input and returns the robust weights as output. See “Robust Options” on page 32-6179.

`WgtFun` will be removed in a future software release. Use `RobustWgtFun` instead.

## Examples

Suppose you want to change the default parameter values for the function `evfit`, which fits an extreme value distribution to data. The defaults parameter values are:

```
statset('evfit')
ans =
    Display: 'off'
    MaxFunEvals: []
    MaxIter: []
    TolBnd: []
    TolFun: []
    TolTypeFun: []
    TolX: 1.0000e-06
    TolTypeX: []
    GradObj: []
    Jacobian: []
    DerivStep: []
    FunValCheck: []
    Robust: []
    RobustWgtFun: []
    WgtFun: []
    Tune: []
    UseParallel: []
    UseSubstreams: []
```



```
Streams: []
OutputFcn: []
```

The only parameters that `evfit` uses are `Display` and `TolX`. To create an options structure with the value of `TolX` set to  $1e-8$ , enter:

```
options = statset('TolX',1e-8)
% Pass options to evfit:
mu = 1;
sigma = 1;
data = evrnd(mu,sigma,1,100);

paramhat = evfit(data,[],[],[],options)
```

## Definitions

### Robust Options

Weight Function	Equation	Default Tuning Constant
'andrews'	$w = (\text{abs}(r) < \pi) .* \sin(r) ./ r$	1.339
'bisquare' (default)	$w = (\text{abs}(r) < 1) .* (1 - r.^2).^2$	4.685
'cauchy'	$w = 1 ./ (1 + r.^2)$	2.385
'fair'	$w = 1 ./ (1 + \text{abs}(r))$	1.400
'huber'	$w = 1 ./ \max(1, \text{abs}(r))$	1.345
'logistic'	$w = \tanh(r) ./ r$	1.205
'talwar'	$w = 1 * (\text{abs}(r) < 1)$	2.795
'welsch'	$w = \exp(-r.^2)$	2.985
[]	No robust fitting	—

## See Also

`statget`

**Introduced before R2006a**

## std

Standard deviation of probability distribution

## Syntax

```
s = std(pd)
```

## Description

`s = std(pd)` returns the standard deviation `s` of the probability distribution `pd`.

## Examples

### Standard Deviation of a Fitted Distribution

Load the sample data. Create a vector containing the first column of students' exam grade data.

```
load examgrades  
x = grades(:,1);
```

Fit a normal distribution object to the data.

```
pd = fitdist(x, 'Normal')
```

```
pd =
```

```
NormalDistribution
```

```
Normal distribution
```

```
mu = 75.0083 [73.4321, 76.5846]  
sigma = 8.7202 [7.7391, 9.98843]
```

Compute the standard deviation of the fitted distribution.

```
s = std(pd)
```

```
s =
```

```
8.7202
```

For a normal distribution, the standard deviation is equal to the parameter `sigma`.

### Standard Deviation of a Skewed Distribution

Create a Weibull probability distribution object

```
pd = makedist('Weibull', 'a', 5, 'b', 2)
```

```
pd =
```

```
WeibullDistribution
```

```
Weibull distribution
```

```
A = 5
```

```
B = 2
```

Compute the standard deviation of the distribution.

```
s = std(pd)
```

```
s =
```

```
2.3163
```

## Input Arguments

**pd** — Probability distribution  
probability distribution object

Probability distribution, specified as a probability distribution object. Create a probability distribution object with specified parameter values using `makedist`. Alternatively, create a probability distribution object by fitting it to data using `fitdist` or the Distribution Fitter app.

## Output Arguments

### **s** — Standard deviation

nonnegative scalar value

Standard deviation of the probability distribution, returned as a nonnegative scalar value.

## See Also

`distributionFitter` | `fitdist` | `makedist`

## Topics

“Working with Probability Distributions” on page 5-3

Introduced before R2006a

## std

**Class:** ProbDistUnivParam

Return standard deviation of ProbDistUnivParam object

## Syntax

$S = \text{std}(PD)$

## Description

$S = \text{std}(PD)$  returns  $S$ , the standard deviation of the ProbDistUnivParam object  $PD$ .

## Input Arguments

$PD$  An object of the class ProbDistUnivParam.

## Output Arguments

$S$  The standard deviation of the ProbDistUnivParam object  $PD$ .

## See Also

std

## std

**Class:** prob.ParametricTruncatableDistribution

**Package:** prob

Standard deviation of probability distribution object

## Syntax

```
s = std(pd)
```

## Description

`s = std(pd)` returns the standard deviation `s` of the probability distribution `pd`.

## Input Arguments

**pd** — Probability distribution

probability distribution object

Probability distribution, specified as a probability distribution object. Create a probability distribution object with specified parameter values using `makedist`.

## Output Arguments

**s** — Standard deviation

nonnegative scalar value

Standard deviation of the probability distribution, returned as a nonnegative scalar value.

## Examples

### Standard Deviation of a Triangular Distribution

Create a triangular distribution object.

```
pd = makedist('Triangular', 'a', -3, 'b', 1, 'c', 3)
```

```
pd =
```

```
    TriangularDistribution
```

```
A = -3, B = 1, C = 3
```

Compute the standard deviation of the distribution.

```
s = std(pd)
```

```
s =
```

```
    1.2472
```

### See Also

`makedist`



## std

**Class:** prob.ToolboxFittableParametricDistribution

**Package:** prob

Standard deviation of probability distribution object

## Syntax

```
s = std(pd)
```

## Description

`s = std(pd)` returns the standard deviation `s` of the probability distribution `pd`.

## Input Arguments

**pd** — Probability distribution

probability distribution object

Probability distribution, specified as a probability distribution object. Create a probability distribution object with specified parameter values using `makedist`. Alternatively, create a probability distribution object by fitting it to data using `fitdist` or the Distribution Fitter app.

## Output Arguments

**s** — Standard deviation

nonnegative scalar value

Standard deviation of the probability distribution, returned as a nonnegative scalar value.

## Examples

### Standard Deviation of a Fitted Distribution

Load the sample data. Create a vector containing the first column of students' exam grade data.

```
load examgrades;  
x = grades(:,1);
```

Fit a normal distribution object to the data.

```
pd = fitdist(x, 'Normal')  
  
pd =  
  
NormalDistribution  
  
Normal distribution  
    mu = 75.0083    [73.4321, 76.5846]  
    sigma = 8.7202    [7.7391, 9.98843]
```

Compute the standard deviation of the fitted distribution.

```
s = std(pd)  
  
s =  
  
8.7202
```

For a normal distribution, the standard deviation is equal to the parameter `sigma`.

### Standard Deviation of a Skewed Distribution

Create a Weibull probability distribution object

```
pd = makedist('Weibull', 'a', 5, 'b', 2)
```

```
pd =  
  
    WeibullDistribution  
  
Weibull distribution  
  A = 5  
  B = 2
```

Compute the standard deviation of the distribution.

```
s = std(pd)
```

```
s =  
  
    2.3163
```

## See Also

[distributionFitter](#) | [fitdist](#) | [makedist](#)

## step

**Class:** GeneralizedLinearModel

Improve generalized linear regression model by adding or removing terms

## Syntax

```
mdl1 = step(mdl)
mdl1 = step(mdl, Name, Value)
```

## Description

`mdl1 = step(mdl)` returns an improved generalized linear model based on `mdl`, with one predictor added or removed.

`mdl1 = step(mdl, Name, Value)` improves a generalized linear model with additional options specified by one or more `Name, Value` pair arguments.

## Input Arguments

**mdl** — Generalized linear model

GeneralizedLinearModel object

Generalized linear model representing a least-squares fit of the link of the response to the data, returned as a GeneralizedLinearModel object.

For properties and methods of the generalized linear model object, `mdl`, see the GeneralizedLinearModel class page.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single

quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

**Criterion — Criterion to add or remove terms**

`'sse'` (default) | `'aic'` | `'bic'` | `'rsquared'` | `'adjrsquared'`

Criterion to add or remove terms, specified as the comma-separated pair consisting of `'Criterion'` and one of the following:

- `'sse'` — Default for `stepwiselm`.  $p$ -value for an  $F$ -test of the change in the sum of squared error by adding or removing the term.
- `'aic'` — Change in the value of Akaike information criterion (AIC).
- `'bic'` — Change in the value of Bayesian information criterion (BIC).
- `'rsquared'` — Increase in the value of  $R^2$ .
- `'adjrsquared'` — Increase in the value of adjusted  $R^2$ .

Example: `'Criterion', 'bic'`

**Lower — Model specification describing terms that cannot be removed from model**

`'constant'` (default)

Model specification describing terms that cannot be removed from the model, specified as the comma-separated pair consisting of `'Lower'` and one of the options for `modelspec` naming the model.

Example: `'Lower', 'linear'`

**nSteps — Number of steps to take**

no limit (default) | positive integer

Number of steps to take, specified as the comma-separated pair consisting of `'NSteps'` and a positive integer.

Data Types: `single` | `double`

**PEnter — Improvement measure for adding term**

scalar value

Improvement measure for adding a term, specified as the comma-separated pair consisting of `'PEnter'` and a scalar value. The default values are below.

Criterion	Default value	Decision
'Deviance'	0.05	If the $p$ -value of $F$ or chi-squared statistic is smaller than <code>PEnter</code> , add the term to the model.
'SSE'	0.05	If the SSE of the model is smaller than <code>PEnter</code> , add the term to the model.
'AIC'	0	If the change in the AIC of the model is smaller than <code>PEnter</code> , add the term to the model.
'BIC'	0	If the change in the BIC of the model is smaller than <code>PEnter</code> , add the term to the model.
'Rsquared'	0.1	If the increase in the R-squared of the model is larger than <code>PEnter</code> , add the term to the model.
'AdjRsquared'	0	If the increase in the adjusted R-squared of the model is larger than <code>PEnter</code> , add the term to the model.

For more information on the criteria, see `Criterion` name-value pair argument.

Example: `'PEnter', 0.075`

#### **PRemove** — Improvement measure for removing term scalar value

Improvement measure for removing a term, specified as the comma-separated pair consisting of `'PRemove'` and a scalar value.

Criterion	Default value	Decision
'Deviance'	0.10	If the $p$ -value of $F$ or chi-squared statistic is larger than <code>PRemove</code> , remove the term from the model.
'SSE'	0.10	If the $p$ -value of the $F$ statistic is larger than <code>PRemove</code> , remove the term from the model.
'AIC'	0.01	If the change in the AIC of the model is larger than <code>PRemove</code> , remove the term from the model.
'BIC'	0.01	If the change in the BIC of the model is larger than <code>PRemove</code> , remove the term from the model.
'Rsquared'	0.05	If the increase in the R-squared value of the model is smaller than <code>PRemove</code> , remove the term from the model.
'AdjRsquared'	-0.05	If the increase in the adjusted R-squared value of the model is smaller than <code>PRemove</code> , remove the term from the model.

At each step, stepwise algorithm also checks whether any term is redundant (linearly dependent) with other terms in the current model. When any term is linearly dependent with other terms in the current model, it is removed, regardless of the criterion value.

For more information on the criteria, see `Criterion` name-value pair argument.

Example: `'PRemove', 0.05`

**Upper** — Model specification describing largest set of terms in fit

`'interactions'` (default) | character vector

Model specification describing the largest set of terms in the fit, specified as the comma-separated pair consisting of 'Upper' and one of the character vector options for `modelspec` naming the model.

Example: `'Upper', 'quadratic'`

### **verbose** — Control for display of information

1 (default) | 0 | 2

Control for display of information, specified as the comma-separated pair consisting of 'Verbose' and one of the following:

- 0 — Suppress all display.
- 1 — Display the action taken at each step.
- 2 — Also display the actions evaluated at each step.

Example: `'Verbose', 2`

## Output Arguments

### **mdl1** — Regression model with additional terms

`LinearModel` object

Regression model with additional terms, returned as a `LinearModel` object. `mdl1` is the same as `mdl` but includes the additional terms specified in `terms`. To overwrite `mdl`, set `mdl1` equal to `mdl`.

## Examples

### **Add Predictors One at a Time**

Fit a Poisson regression model using random data and a single predictor, then step in other predictors.

Generate artificial data with 20 predictors, using three of the predictors for the responses.



```
rng('default') % for reproducibility
X = randn(100,20);
mu = exp(X(:, [5 10 15])*[.4;.2;.3] + 1);
y = poissrnd(mu);
```

Construct a generalized linear model using  $X(:, 1)$  as the only predictor.

```
mdl = fitglm(X,y,...
    'y ~ x1','Distribution','poisson')
```

```
mdl =
```

Generalized linear regression model:

```
log(y) ~ 1 + x1
Distribution = Poisson
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
	-----	-----	-----	-----
(Intercept)	1.1278	0.057487	19.618	1.0904e-85
x1	0.061287	0.04848	1.2642	0.20617

100 observations, 98 error degrees of freedom

Dispersion: 1

Chi<sup>2</sup>-statistic vs. constant model: 1.59, p-value = 0.208

Add a variable to the model using step.

```
mdl1 = step(mdl)
```

1. Adding x5, Deviance = 134.2976, Chi2Stat = 50.80176, PValue = 1.021821e-12

```
mdl1 =
```

Generalized linear regression model:

```
log(y) ~ 1 + x1 + x5
Distribution = Poisson
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
--	----------	----	-------	--------

	Estimate	SE	tStat	pValue
(Intercept)	1.0418	0.062341	16.712	1.07e-62
x1	0.018803	0.049916	0.37671	0.70639
x5	0.47881	0.067875	7.0542	1.7357e-12

100 observations, 97 error degrees of freedom  
 Dispersion: 1  
 Chi^2-statistic vs. constant model: 52.4, p-value = 4.21e-12

**Add another variable to the model using step.**

```
mdl1 = step(mdl1)
```

2. Adding x15, Deviance = 105.9973, Chi2Stat = 28.30027, PValue = 1.038814e-07

```
mdl1 =
```

Generalized linear regression model:  
 $\log(y) \sim 1 + x1 + x5 + x15$   
 Distribution = Poisson

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	1.0459	0.0627	16.681	1.7975e-62
x1	0.026907	0.05003	0.53782	0.5907
x5	0.3983	0.068376	5.8251	5.7073e-09
x15	0.28949	0.053992	5.3618	8.2375e-08

100 observations, 96 error degrees of freedom  
 Dispersion: 1  
 Chi^2-statistic vs. constant model: 80.7, p-value = 2.18e-17

- “Plots to Understand Predictor Effects and How to Modify a Model” on page 12-31

## Tips

- Use `addTerms` or `removeTerms` to control exactly which terms enter or leave the model.

## Algorithms

Stepwise regression is a systematic method for adding and removing terms from a linear or generalized linear model based on their statistical significance in explaining the response variable. The method begins with an initial model, specified using `modelspec`, and then compares the explanatory power of incrementally larger and smaller models.

MATLAB uses forward and backward stepwise regression to determine a final model. At each step, the method searches for terms to add to or remove from the model based on the value of the 'Criterion' argument. The default value of 'Criterion' is 'sse', and in this case, `stepwiselm` uses the  $p$ -value of an  $F$ -statistic to test models with and without a potential term at each step. If a term is not currently in the model, the null hypothesis is that the term would have a zero coefficient if added to the model. If there is sufficient evidence to reject the null hypothesis, the term is added to the model. Conversely, if a term is currently in the model, the null hypothesis is that the term has a zero coefficient. If there is insufficient evidence to reject the null hypothesis, the term is removed from the model.

Here is how stepwise proceeds when 'Criterion' is 'sse':

- 1 Fit the initial model.
- 2 Examine a set of available terms not in the model. If any of these terms have  $p$ -values less than an entrance tolerance (that is, if it is unlikely that they would have zero coefficient if added to the model), add the one with the smallest  $p$ -value and repeat this step; otherwise, go to step 3.
- 3 If any of the available terms in the model have  $p$ -values greater than an exit tolerance (that is, the hypothesis of a zero coefficient cannot be rejected), remove the one with the largest  $p$ -value and go to step 2; otherwise, end.

At any stage, the function will not add a higher-order term if the model does not also include all lower-order terms that are subsets of it. For example, it will not try to add the term  $X1 : X2^2$  unless both  $X1$  and  $X2^2$  are already in the model. Similarly, the function will not remove lower-order terms that are subsets of higher-order terms that remain in

the model. For example, it will not examine to remove  $X_1$  or  $X_2^2$  if  $X_1 : X_2^2$  stays in the model.

The default for `stepwiseglm` is 'Deviance' and it follows a similar procedure for adding or removing terms.

There are several other criteria available, which you can specify using the 'Criterion' argument. You can use the change in the value of the Akaike information criterion, Bayesian information criterion, R-squared, adjusted R-squared as a criterion to add or remove terms.

Depending on the terms included in the initial model and the order in which terms are moved in and out, the method might build different models from the same set of potential terms. The method terminates when no single step improves the model. There is no guarantee, however, that a different initial model or a different sequence of steps will not lead to a better fit. In this sense, stepwise models are locally optimal, but might not be globally optimal.

## Alternatives

Use `stepwiseglm` to select a model from a starting model, continuing until no single step is beneficial.

Use `addTerms` or `removeTerms` to add or remove particular terms.

## See Also

`GeneralizedLinearModel` | `addTerms` | `removeTerms` | `stepwiseglm`

## Topics

“Plots to Understand Predictor Effects and How to Modify a Model” on page 12-31

“Generalized Linear Models” on page 12-12

---

## step

**Class:** LinearModel

Improve linear regression model by adding or removing terms

## Syntax

```
mdl1 = step(mdl)
mdl1 = step(mdl, Name, Value)
```

## Description

`mdl1 = step(mdl)` returns an improved linear model based on `mdl`, with one predictor added or removed.

---

**Note** You can use `step` only if `mdl.Robust = []`. This holds when you create `mdl` with `fitlm` having the `RobustOpts` name-value pair set to the default `'off'`.

---

`mdl1 = step(mdl, Name, Value)` returns an improved linear model using additional options specified by one or more `Name, Value` pair arguments. For example, you can specify the criterion to use to add or remove terms.

## Input Arguments

**mdl** — Full, fitted linear regression model

LinearModel object

Full, fitted linear regression model, specified as a `LinearModel` object constructed using `fitlm` or `stepwiselm`.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### Criterion — Criterion for selecting terms to add or remove

'SSE' (default)

Criterion for selecting terms to add or remove, specified as the comma-separated pair consisting of 'Criterion' and one of the following.

Criterion	PEnter	PRemove	Compared Against
'SSE'	0.05	< 0.1	$p$ -value for $F$ test
'AIC'	0	< 0.01	Change in AIC
'BIC'	0	< 0.01	Change in BIC
'Rsquared'	0.1	> 0.05	Increase in R-squared
'AdjRsquared'	0	> -0.05	Increase in adjusted R-squared

Example: 'Criterion', 'BIC'

### Lower — Model specification describing terms that cannot be removed from model

'constant' (default)

Model specification describing terms that cannot be removed from the model, specified as the comma-separated pair consisting of 'Lower' and one of the options for `modelspec` naming the model.

Example: 'Lower', 'linear'

### NSteps — Number of steps to take

no limit (default) | positive integer

Number of steps to take, specified as the comma-separated pair consisting of 'NSteps' and a positive integer.

Data Types: `single` | `double`

### PEnter — Improvement measure for adding term

scalar value

Improvement measure for adding a term, specified as the comma-separated pair consisting of 'PEnter' and a scalar value. The default values are below.

Criterion	Default value	Decision
'Deviance'	0.05	If the $p$ -value of $F$ or chi-squared statistic is smaller than PEnter, add the term to the model.
'SSE'	0.05	If the SSE of the model is smaller than PEnter, add the term to the model.
'AIC'	0	If the change in the AIC of the model is smaller than PEnter, add the term to the model.
'BIC'	0	If the change in the BIC of the model is smaller than PEnter, add the term to the model.
'Rsquared'	0.1	If the increase in the R-squared of the model is larger than PEnter, add the term to the model.
'AdjRsquared'	0	If the increase in the adjusted R-squared of the model is larger than PEnter, add the term to the model.

For more information on the criteria, see Criterion name-value pair argument.

Example: 'PEnter', 0.075

**PRemove** — Improvement measure for removing term  
scalar value

Improvement measure for removing a term, specified as the comma-separated pair consisting of 'PRemove' and a scalar value.

Criterion	Default value	Decision
'Deviance'	0.10	If the $p$ -value of $F$ or chi-squared statistic is larger than <code>PRemove</code> , remove the term from the model.
'SSE'	0.10	If the $p$ -value of the $F$ statistic is larger than <code>PRemove</code> , remove the term from the model.
'AIC'	0.01	If the change in the AIC of the model is larger than <code>PRemove</code> , remove the term from the model.
'BIC'	0.01	If the change in the BIC of the model is larger than <code>PRemove</code> , remove the term from the model.
'Rsquared'	0.05	If the increase in the R-squared value of the model is smaller than <code>PRemove</code> , remove the term from the model.
'AdjRsquared'	-0.05	If the increase in the adjusted R-squared value of the model is smaller than <code>PRemove</code> , remove the term from the model.

At each step, stepwise algorithm also checks whether any term is redundant (linearly dependent) with other terms in the current model. When any term is linearly dependent with other terms in the current model, it is removed, regardless of the criterion value.

For more information on the criteria, see `Criterion` name-value pair argument.

Example: `'PRemove', 0.05`

### Upper — Model specification describing largest set of terms in fit

`'interactions'` (default) | character vector



Model specification describing the largest set of terms in the fit, specified as the comma-separated pair consisting of 'Upper' and one of the character vector options for `modelspec` naming the model.

Example: `'Upper', 'quadratic'`

### **verbose** — Control for display of information

1 (default) | 0 | 2

Control for display of information, specified as the comma-separated pair consisting of 'Verbose' and one of the following:

- 0 — Suppress all display.
- 1 — Display the action taken at each step.
- 2 — Also display the actions evaluated at each step.

Example: `'Verbose', 2`

## Output Arguments

### **mdl1** — Improved linear regression model

`LinearModel` object

Improved linear regression model, returned as a `LinearModel` object. To overwrite `mdl`, set `mdl1` equal to `mdl`.

## Examples

### **Modify a Linear Regression Model Using `step`**

Fit a linear model to car data. Use `step` to evaluate whether a quadratic model improves the fit quality.

Load `carsmall` data, and create a table using weight and model year predictors with MPG response.

```
load carsmall
tbl = table(MPG,Weight);
tbl.Year = categorical(Model_Year);
```

Make a linear model of MPG as a function of Year and Weight.

```
mdl = fitlm(tbl,'MPG ~ Year + Weight')
```

```
mdl =
```

```
Linear regression model:
    MPG ~ 1 + Weight + Year
```

```
Estimated Coefficients:
```

	Estimate	SE	tStat	pValue
(Intercept)	40.11	1.5418	26.016	1.2024e-43
Weight	-0.0066475	0.00042802	-15.531	3.3639e-27
Year_76	1.9291	0.74761	2.5804	0.011488
Year_82	7.9093	0.84975	9.3078	7.8681e-15

```
Number of observations: 94, Error degrees of freedom: 90
Root Mean Squared Error: 2.92
R-squared: 0.873, Adjusted R-Squared 0.868
F-statistic vs. constant model: 206, p-value = 3.83e-40
```

Use step to adjust the model to potentially include full quadratic terms.

```
mdl1 = step(mdl,'upper','quadratic')
```

```
1. Adding Weight^2, FStat = 9.9164, pValue = 0.0022303
```

```
mdl1 =
```

```
Linear regression model:
    MPG ~ 1 + Weight + Year + Weight^2
```

```
Estimated Coefficients:
```

	Estimate	SE	tStat	pValue
--	----------	----	-------	--------

(Intercept)	54.206	4.7117	11.505	2.6648e-19
Weight	-0.016404	0.0031249	-5.2493	1.0283e-06
Year_76	2.0887	0.71491	2.9215	0.0044137
Year_82	8.1864	0.81531	10.041	2.6364e-16
Weight^2	1.5573e-06	4.9454e-07	3.149	0.0022303

Number of observations: 94, Error degrees of freedom: 89  
 Root Mean Squared Error: 2.78  
 R-squared: 0.885, Adjusted R-Squared 0.88  
 F-statistic vs. constant model: 172, p-value = 5.52e-41

- “Linear Regression Workflow” on page 11-43
- “Change Models” on page 11-36

## Algorithms

### Stepwise Regression

Stepwise regression is a systematic method for adding and removing terms from a linear or generalized linear model based on their statistical significance in explaining the response variable. The method begins with an initial model, specified using `modelspec`, and then compares the explanatory power of incrementally larger and smaller models.

MATLAB uses forward and backward stepwise regression to determine a final model. At each step, the method searches for terms to add to or remove from the model based on the value of the 'Criterion' argument. The default value of 'Criterion' is 'sse', and in this case, `stepwiselm` uses the  $p$ -value of an  $F$ -statistic to test models with and without a potential term at each step. If a term is not currently in the model, the null hypothesis is that the term would have a zero coefficient if added to the model. If there is sufficient evidence to reject the null hypothesis, the term is added to the model. Conversely, if a term is currently in the model, the null hypothesis is that the term has a zero coefficient. If there is insufficient evidence to reject the null hypothesis, the term is removed from the model.

Here is how stepwise proceeds when 'Criterion' is 'sse':

- 1 Fit the initial model.

- 2 Examine a set of available terms not in the model. If any of these terms have  $p$ -values less than an entrance tolerance (that is, if it is unlikely that they would have zero coefficient if added to the model), add the one with the smallest  $p$ -value and repeat this step; otherwise, go to step 3.
- 3 If any of the available terms in the model have  $p$ -values greater than an exit tolerance (that is, the hypothesis of a zero coefficient cannot be rejected), remove the one with the largest  $p$ -value and go to step 2; otherwise, end.

At any stage, the function will not add a higher-order term if the model does not also include all lower-order terms that are subsets of it. For example, it will not try to add the term  $X1 : X2^2$  unless both  $X1$  and  $X2^2$  are already in the model. Similarly, the function will not remove lower-order terms that are subsets of higher-order terms that remain in the model. For example, it will not examine to remove  $X1$  or  $X2^2$  if  $X1 : X2^2$  stays in the model.

The default for `stepwiseglm` is 'Deviance' and it follows a similar procedure for adding or removing terms.

There are several other criteria available, which you can specify using the 'Criterion' argument. You can use the change in the value of the Akaike information criterion, Bayesian information criterion, R-squared, adjusted R-squared as a criterion to add or remove terms.

Depending on the terms included in the initial model and the order in which terms are moved in and out, the method might build different models from the same set of potential terms. The method terminates when no single step improves the model. There is no guarantee, however, that a different initial model or a different sequence of steps will not lead to a better fit. In this sense, stepwise models are locally optimal, but might not be globally optimal.

## Alternatives

Use `stepwiselm` to select a model from a starting model, continuing until no single step is beneficial.

Use `addTerms` or `removeTerms` to add or remove particular terms.

## See Also

`LinearModel` | `addTerms` | `removeTerms` | `stepwiselm`

## Topics

“Linear Regression Workflow” on page 11-43

“Change Models” on page 11-36

“Linear Regression” on page 11-11

## stepwise

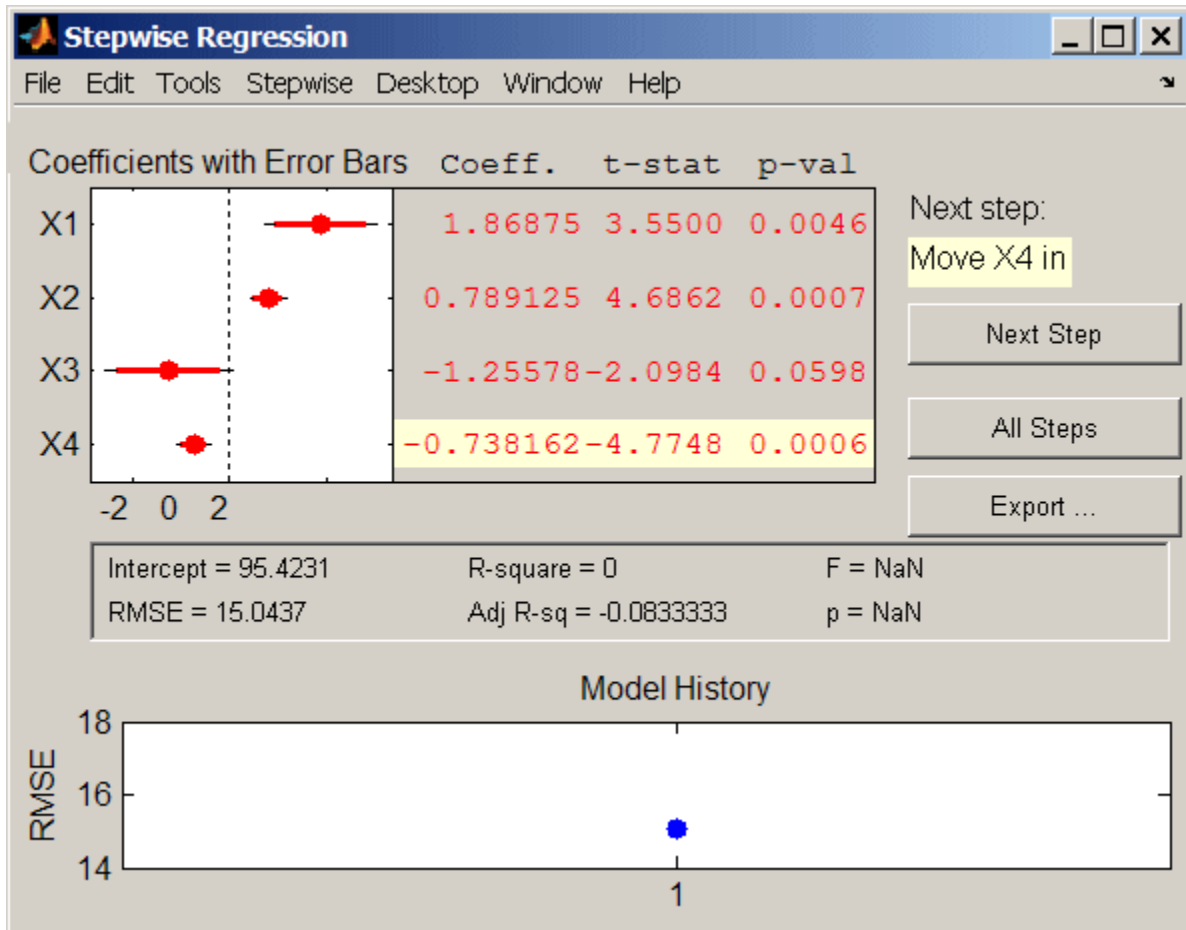
Interactive stepwise regression

### Syntax

```
stepwise  
stepwise(X, y)  
stepwise(X, y, inmodel, penter, premove)
```

### Description

`stepwise` uses the sample data in `hald.mat` to display a graphical user interface for performing stepwise regression of the response values in `heat` on the predictive terms in `ingredients`.



The upper left of the interface displays estimates of the coefficients for all potential terms, with horizontal bars indicating 90% (colored) and 95% (grey) confidence intervals. The red color indicates that, initially, the terms are not in the model. Values displayed in the table are those that would result if the terms were added to the model.

The middle portion of the interface displays summary statistics for the entire model. These statistics are updated with each step.

The lower portion of the interface, **Model History**, displays the RMSE for the model. The plot tracks the RMSE from step to step, so you can compare the optimality of

different models. Hover over the blue dots in the history to see which terms were in the model at a particular step. Click on a blue dot in the history to open a copy of the interface initialized with the terms in the model at that step.

Initial models, as well as entrance/exit tolerances for the  $p$ -values of  $F$ -statistics, are specified using additional input arguments to `stepwise`. Defaults are an initial model with no terms, an entrance tolerance of 0.05, and an exit tolerance of 0.10.

To center and scale the input data (compute  $z$ -scores) to improve conditioning of the underlying least-squares problem, select `Scale Inputs` from the **Stepwise** menu.

You proceed through a stepwise regression in one of two ways:

- 1 Click **Next Step** to select the recommended next step. The recommended next step either adds the most significant term or removes the least significant term. When the regression reaches a local minimum of RMSE, the recommended next step is “Move no terms.” You can perform all of the recommended steps at once by clicking **All Steps**.
- 2 Click a line in the plot or in the table to toggle the state of the corresponding term. Clicking a red line, corresponding to a term not currently in the model, adds the term to the model and changes the line to blue. Clicking a blue line, corresponding to a term currently in the model, removes the term from the model and changes the line to red.

To call `addedvarplot` and produce an added variable plot from the `stepwise` interface, select **Added Variable Plot** from the **Stepwise** menu. A list of terms is displayed. Select the term you want to add, and then click **OK**.

Click **Export** to display a dialog box that allows you to select information from the interface to save to the MATLAB workspace. Check the information you want to export and, optionally, change the names of the workspace variables to be created. Click **OK** to export the information.

`stepwise(X, y)` displays the interface using the  $p$  predictive terms in the  $n$ -by- $p$  matrix  $X$  and the response values in the  $n$ -by-1 vector  $y$ . Distinct predictive terms should appear in different columns of  $X$ .

---

**Note** `stepwise` automatically includes a constant term in all models. Do not enter a column of 1s directly into  $X$ .

---



stepwise treats NaN values in either  $X$  or  $y$  as missing values, and ignores them.

`stepwise(X, y, inmodel, penter, premove)` additionally specifies the initial model (`inmodel`) and the entrance (`penter`) and exit (`prremove`) tolerances for the  $p$ -values of  $F$ -statistics. `inmodel` is either a logical vector with length equal to the number of columns of  $X$ , or a vector of indices, with values ranging from 1 to the number of columns in  $X$ . The value of `penter` must be less than or equal to the value of `prremove`.

## Algorithms

Stepwise regression is a systematic method for adding and removing terms from a multilinear model based on their statistical significance in a regression. The method begins with an initial model and then compares the explanatory power of incrementally larger and smaller models. At each step, the  $p$  value of an  $F$ -statistic is computed to test models with and without a potential term. If a term is not currently in the model, the null hypothesis is that the term would have a zero coefficient if added to the model. If there is sufficient evidence to reject the null hypothesis, the term is added to the model. Conversely, if a term is currently in the model, the null hypothesis is that the term has a zero coefficient. If there is insufficient evidence to reject the null hypothesis, the term is removed from the model. The method proceeds as follows:

- 1 Fit the initial model.
- 2 If any terms not in the model have  $p$ -values less than an entrance tolerance (that is, if it is unlikely that they would have zero coefficient if added to the model), add the one with the smallest  $p$  value and repeat this step; otherwise, go to step 3.
- 3 If any terms in the model have  $p$ -values greater than an exit tolerance (that is, if it is unlikely that the hypothesis of a zero coefficient can be rejected), remove the one with the largest  $p$  value and go to step 2; otherwise, end.

Depending on the terms included in the initial model and the order in which terms are moved in and out, the method may build different models from the same set of potential terms. The method terminates when no single step improves the model. There is no guarantee, however, that a different initial model or a different sequence of steps will not lead to a better fit. In this sense, stepwise models are locally optimal, but may not be globally optimal.

## See Also

`addedvarplot` | `regress` | `stepwisefit`

**Introduced before R2006a**

# GeneralizedLinearModel.stepwise

**Class:** GeneralizedLinearModel

Create generalized linear regression model by stepwise regression

---

**Note** GeneralizedLinearModel.stepwise will be removed in a future release. Use stepwiseglm instead.

---

## Syntax

```
mdl = GeneralizedLinearModel.stepwise(tbl,modelspec)
mdl = GeneralizedLinearModel.stepwise(X,y,modelspec)
mdl = GeneralizedLinearModel.stepwise(...,modelspec,Name,Value)
```

## Description

mdl = GeneralizedLinearModel.stepwise(tbl,modelspec) creates a generalized linear model of a table or dataset array tbl, using stepwise regression to add or remove predictors. modelspec is the starting model for the stepwise procedure.

mdl = GeneralizedLinearModel.stepwise(X,y,modelspec) creates a generalized linear model of the responses y to a data matrix X, using stepwise regression to add or remove predictors.

mdl = GeneralizedLinearModel.stepwise(...,modelspec,Name,Value) creates a generalized linear model with additional options specified by one or more Name,Value pair arguments.

## Input Arguments

**tbl** — Input data

table | dataset array

Input data, specified as a table or dataset array. When `modelspec` is a formula, it specifies the variables to be used as the predictors and response. Otherwise, if you do not specify the predictor and response variables, the last variable is the response variable and the others are the predictor variables by default.

Predictor variables can be numeric, or any grouping variable type, such as logical or categorical (see “Grouping Variables” on page 2-59). The response must be numeric or logical.

To set a different column as the response variable, use the `ResponseVar` name-value pair argument. To use a subset of the columns as predictors, use the `PredictorVars` name-value pair argument.

### **x — Predictor variables**

matrix

Predictor variables, specified as an  $n$ -by- $p$  matrix, where  $n$  is the number of observations and  $p$  is the number of predictor variables. Each column of `X` represents one variable, and each row represents one observation.

By default, there is a constant term in the model, unless you explicitly remove it, so do not include a column of 1s in `X`.

Data Types: `single` | `double` | `logical`

### **y — Response variable**

vector

Response variable, specified as an  $n$ -by-1 vector, where  $n$  is the number of observations. Each entry in `y` is the response for the corresponding row of `X`.

Data Types: `single` | `double` | `logical`

### **modelspec — Starting model**

character vector specifying the model |  $t$ -by- $(p+1)$  terms matrix | character vector of the form `'Y ~ terms'`

Starting model for `stepwiseglm`, specified as one of the following:

- Character vector specifying the type of model.

Character Vector	Model Type
'constant'	Model contains only a constant (intercept) term.
'linear'	Model contains an intercept and linear terms for each predictor.
'interactions'	Model contains an intercept, linear terms, and all products of pairs of distinct predictors (no squared terms).
'purequadratic'	Model contains an intercept, linear terms, and squared terms.
'quadratic'	Model contains an intercept, linear terms, interactions, and squared terms.
'polyijk'	Model is a polynomial with all terms up to degree $i$ in the first predictor, degree $j$ in the second predictor, etc. Use numerals 0 through 9. For example, 'poly2111' has a constant plus all linear and product terms, and also contains terms with predictor 1 squared.

- $t$ -by- $(p+1)$  matrix, namely terms matrix on page 32-6225, specifying terms to include in model, where  $t$  is the number of terms and  $p$  is the number of predictor variables, and plus one is for the response variable.
- Character vector representing a formula on page 32-6228 in the form

`'Y ~ terms'`,

where the `terms` are in “Wilkinson Notation” on page 32-6229.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### **BinomialSize** — Number of trials for binomial distribution

1 (default) | scalar value | vector

Number of trials for binomial distribution, that is the sample size, specified as the comma-separated pair consisting of a scalar value or a vector of the same length as the

response. This is the parameter `n` for the fitted binomial distribution. `BinomialSize` applies only when the `Distribution` parameter is `'binomial'`.

If `BinomialSize` is a scalar value, that means all observations have the same number of trials.

As an alternative to `BinomialSize`, you can specify the response as a two-column vector with counts in column 1 and `BinomialSize` in column 2.

Data Types: `single` | `double`

### **CategoricalVars** — Categorical variable list

cell array of character vectors | logical or numeric index vector

Categorical variable list, specified as the comma-separated pair consisting of `'CategoricalVars'` and either a cell array of character vectors containing categorical variable names in the table or dataset array `tbl`, or a logical or numeric index vector indicating which columns are categorical.

- If data is in a table or dataset array `tbl`, then, by default, `GeneralizedLinearModel.stepwise` treats all categorical values, logical values, character arrays, and cell arrays of character vectors as categorical variables.
- If data is in matrix `X`, then the default value of `'CategoricalVars'` is an empty matrix `[]`. That is, no variable is categorical unless you specify it as categorical.

For example, you can specify the observations 2 and 3 out of 6 as categorical using either of the following examples.

Example: `'CategoricalVars', [2,3]`

Example: `'CategoricalVars', logical([0 1 1 0 0 0])`

Data Types: `single` | `double` | `logical` | `cell`

### **Criterion** — Criterion to add or remove terms

`'sse'` (default) | `'aic'` | `'bic'` | `'rsquared'` | `'adjrsquared'`

Criterion to add or remove terms, specified as the comma-separated pair consisting of `'Criterion'` and one of the following:

- `'sse'` — Default for `stepwiselm`.  $p$ -value for an  $F$ -test of the change in the sum of squared error by adding or removing the term.

- 'aic' — Change in the value of Akaike information criterion (AIC).
- 'bic' — Change in the value of Bayesian information criterion (BIC).
- 'rsquared' — Increase in the value of  $R^2$ .
- 'adjrsquared' — Increase in the value of adjusted  $R^2$ .

Example: 'Criterion', 'bic'

#### **DispersionFlag** — Indicator to compute dispersion parameter

false for 'binomial' and 'poisson' distributions (default) | true

Indicator to compute dispersion parameter for 'binomial' and 'poisson' distributions, specified as the comma-separated pair consisting of 'DispersionFlag' and one of the following.

true	Estimate a dispersion parameter when computing standard errors
false	Default. Use the theoretical value when computing standard errors

The fitting function always estimates the dispersion for other distributions.

Example: 'DispersionFlag', true

#### **Distribution** — Distribution of the response variable

'normal' (default) | 'binomial' | 'poisson' | 'gamma' | 'inverse gaussian'

Distribution of the response variable, specified as the comma-separated pair consisting of 'Distribution' and one of the following.

'normal'	Normal distribution
'binomial'	Binomial distribution
'poisson'	Poisson distribution
'gamma'	Gamma distribution
'inverse gaussian'	Inverse Gaussian distribution

Example: 'Distribution', 'gamma'

#### **Exclude** — Observations to exclude

logical or numeric index vector

Observations to exclude from the fit, specified as the comma-separated pair consisting of 'Exclude' and a logical or numeric index vector indicating which observations to exclude from the fit.

For example, you can exclude observations 2 and 3 out of 6 using either of the following examples.

Example: `'Exclude', [2,3]`

Example: `'Exclude', logical([0 1 1 0 0 0])`

Data Types: `single` | `double` | `logical`

### Intercept — Indicator for constant term

`true` (default) | `false`

Indicator for the constant term (intercept) in the fit, specified as the comma-separated pair consisting of 'Intercept' and either `true` to include or `false` to remove the constant term from the model.

Use 'Intercept' only when specifying the model using a character vector, not a formula or matrix.

Example: `'Intercept', false`

### Link — Link function

The canonical link function (default) | scalar value | structure

Link function to use in place of the canonical link function, specified as the comma-separated pair consisting of 'Link' and one of the following.

Link Function Name	Link Function	Mean (Inverse) Function
'identity'	$f(\mu) = \mu$	$\mu = Xb$
'log'	$f(\mu) = \log(\mu)$	$\mu = \exp(Xb)$
'logit'	$f(\mu) = \log(\mu/(1-\mu))$	$\mu = \exp(Xb) / (1 + \exp(Xb))$
'probit'	$f(\mu) = \Phi^{-1}(\mu)$	$\mu = \Phi(Xb)$
'compploglog'	$f(\mu) = \log(-\log(1 - \mu))$	$\mu = 1 - \exp(-\exp(Xb))$
'reciprocal'	$f(\mu) = 1/\mu$	$\mu = 1/(Xb)$
p (a number)	$f(\mu) = \mu^p$	$\mu = Xb^{1/p}$



Link Function Name	Link Function	Mean (Inverse) Function
<p>S (a structure) with three fields. Each field holds a function handle that accepts a vector of inputs and returns a vector of the same size:</p> <ul style="list-style-type: none"> <li>• <code>S.Link</code> — The link function</li> <li>• <code>S.Inverse</code> — The inverse link function</li> <li>• <code>S.Derivative</code> — The derivative of the link function</li> </ul>	$f(\mu) = S.Link(\mu)$	$\mu = S.Inverse(Xb)$

The link function defines the relationship  $f(\mu) = X*b$  between the mean response  $\mu$  and the linear combination of predictors  $X*b$ .

For more information on the canonical link functions, see [Definitions](#).

Example: `'Link', 'probit'`

**Lower** — Model specification describing terms that cannot be removed from model  
`'constant'` (default)

Model specification describing terms that cannot be removed from the model, specified as the comma-separated pair consisting of `'Lower'` and one of the options for `modelspec` naming the model.

Example: `'Lower', 'linear'`

**Offset** — Offset variable  
`[]` (default) | vector | character vector

Offset variable in the fit, specified as the comma-separated pair consisting of `'Offset'` and a vector or name of a variable with the same length as the response.

`fitglm` and `stepwiseglm` use `Offset` as an additional predictor, with a coefficient value fixed at 1.0. In other words, the formula for fitting is

$\mu \sim \text{Offset} + (\text{terms involving real predictors})$

with the `Offset` predictor having coefficient 1.

For example, consider a Poisson regression model. Suppose the number of counts is known for theoretical reasons to be proportional to a predictor `A`. By using the log link function and by specifying `log(A)` as an offset, you can force the model to satisfy this theoretical constraint.

Data Types: `single` | `double` | `char`

### **PEnter** — Improvement measure for adding term

scalar value

Improvement measure for adding a term, specified as the comma-separated pair consisting of `'PEnter'` and a scalar value. The default values are below.

Criterion	Default value	Decision
'Deviance'	0.05	If the $p$ -value of $F$ or chi-squared statistic is smaller than <code>PEnter</code> , add the term to the model.
'SSE'	0.05	If the SSE of the model is smaller than <code>PEnter</code> , add the term to the model.
'AIC'	0	If the change in the AIC of the model is smaller than <code>PEnter</code> , add the term to the model.
'BIC'	0	If the change in the BIC of the model is smaller than <code>PEnter</code> , add the term to the model.
'Rsquared'	0.1	If the increase in the R-squared of the model is larger than <code>PEnter</code> , add the term to the model.

Criterion	Default value	Decision
'AdjRsquared'	0	If the increase in the adjusted R-squared of the model is larger than PEnter, add the term to the model.

For more information on the criteria, see `Criterion` name-value pair argument.

Example: `'PEnter', 0.075`

### **PredictorVars** — Predictor variables

cell array of character vectors | logical or numeric index vector

Predictor variables to use in the fit, specified as the comma-separated pair consisting of `'PredictorVars'` and either a cell array of character vectors of the variable names in the table or dataset array `tbl`, or a logical or numeric index vector indicating which columns are predictor variables.

The character vectors should be among the names in `tbl`, or the names you specify using the `'VarNames'` name-value pair argument.

The default is all variables in `X`, or all variables in `tbl` except for `ResponseVar`.

For example, you can specify the second and third variables as the predictor variables using either of the following examples.

Example: `'PredictorVars', [2,3]`

Example: `'PredictorVars', logical([0 1 1 0 0 0])`

Data Types: `single` | `double` | `logical` | `cell`

### **PRemove** — Improvement measure for removing term

scalar value

Improvement measure for removing a term, specified as the comma-separated pair consisting of `'PRemove'` and a scalar value.

Criterion	Default value	Decision
'Deviance'	0.10	If the $p$ -value of $F$ or chi-squared statistic is larger than <code>PRemove</code> , remove the term from the model.
'SSE'	0.10	If the $p$ -value of the $F$ statistic is larger than <code>PRemove</code> , remove the term from the model.
'AIC'	0.01	If the change in the AIC of the model is larger than <code>PRemove</code> , remove the term from the model.
'BIC'	0.01	If the change in the BIC of the model is larger than <code>PRemove</code> , remove the term from the model.
'Rsquared'	0.05	If the increase in the R-squared value of the model is smaller than <code>PRemove</code> , remove the term from the model.
'AdjRsquared'	-0.05	If the increase in the adjusted R-squared value of the model is smaller than <code>PRemove</code> , remove the term from the model.

At each step, stepwise algorithm also checks whether any term is redundant (linearly dependent) with other terms in the current model. When any term is linearly dependent with other terms in the current model, it is removed, regardless of the criterion value.

For more information on the criteria, see `Criterion` name-value pair argument.

Example: `'PRemove', 0.05`

**ResponseVar — Response variable**

last column in `tbl` (default) | character vector containing variable name | logical or numeric index vector

Response variable to use in the fit, specified as the comma-separated pair consisting of `'ResponseVar'` and either a character vector containing the variable name in the table or dataset array `tbl`, or a logical or numeric index vector indicating which column is the response variable. You typically need to use `'ResponseVar'` when fitting a table or dataset array `tbl`.

For example, you can specify the fourth variable, say `yield`, as the response out of six variables, in one of the following ways.

Example: `'ResponseVar', 'yield'`

Example: `'ResponseVar', [4]`

Example: `'ResponseVar', logical([0 0 0 1 0 0])`

Data Types: `single` | `double` | `logical` | `char`

**Upper — Model specification describing largest set of terms in fit**

`'interactions'` (default) | character vector

Model specification describing the largest set of terms in the fit, specified as the comma-separated pair consisting of `'Upper'` and one of the character vector options for `modelspec` naming the model.

Example: `'Upper', 'quadratic'`

**VarNames — Names of variables in fit**

`{'x1', 'x2', ..., 'xn', 'y'}` (default) | cell array of character vectors

Names of variables in fit, specified as the comma-separated pair consisting of `'VarNames'` and a cell array of character vectors including the names for the columns of `X` first, and the name for the response variable `y` last.

`'VarNames'` is not applicable to variables in a table or dataset array, because those variables already have names.

For example, if in your data, horsepower, acceleration, and model year of the cars are the predictor variables, and miles per gallon (MPG) is the response variable, then you can name the variables as follows.

Example: 'VarNames', {'Horsepower', 'Acceleration', 'Model\_Year', 'MPG'}

Data Types: cell

### **weights** — Observation weights

`ones(n,1)` (default) |  $n$ -by-1 vector of nonnegative scalar values

Observation weights, specified as the comma-separated pair consisting of 'Weights' and an  $n$ -by-1 vector of nonnegative scalar values, where  $n$  is the number of observations.

Data Types: single | double

## Output Arguments

### **mdl** — Generalized linear model

GeneralizedLinearModel object

Generalized linear model representing a least-squares fit of the link of the response to the data, returned as a GeneralizedLinearModel object.

For properties and methods of the generalized linear model object, mdl, see the GeneralizedLinearModel class page.

## Examples

### **Create a Generalized Linear Model Stepwise**

Create response data using just three of 20 predictors, and create a generalized linear model stepwise to see if it uses just the correct predictors.

Create data with 20 predictors, and Poisson response using just three of the predictors, plus a constant.

```
rng default % for reproducibility
X = randn(100,20);
mu = exp(X(:, [5 10 15]) * [.4; .2; .3] + 1);
y = poissrnd(mu);
```

Fit a generalized linear model using the Poisson distribution.

```
mdl = stepwiseglm(X,y,...
    'constant','upper','linear','Distribution','poisson')
1. Adding x5, Deviance = 134.439, Chi2Stat = 52.24814, PValue = 4.891229e-13
2. Adding x15, Deviance = 106.285, Chi2Stat = 28.15393, PValue = 1.1204e-07
3. Adding x10, Deviance = 95.0207, Chi2Stat = 11.2644, PValue = 0.000790094
```

```
mdl =
```

```
Generalized linear regression model:
log(y) ~ 1 + x5 + x10 + x15
Distribution = Poisson
```

```
Estimated Coefficients:
```

	Estimate	SE	tStat	pValue
(Intercept)	1.0115	0.064275	15.737	8.4217e-56
x5	0.39508	0.066665	5.9263	3.0977e-09
x10	0.18863	0.05534	3.4085	0.0006532
x15	0.29295	0.053269	5.4995	3.8089e-08

```
100 observations, 96 error degrees of freedom
Dispersion: 1
Chi^2-statistic vs. constant model: 91.7, p-value = 9.61e-20
```

- “Compare large and small stepwise models” on page 11-127

## Definitions

### Terms Matrix

A terms matrix is a  $t$ -by- $(p + 1)$  matrix specifying terms in a model, where  $t$  is the number of terms,  $p$  is the number of predictor variables, and plus one is for the response variable.

The value of  $T(i, j)$  is the exponent of variable  $j$  in term  $i$ . Suppose there are three predictor variables A, B, and C:

```
[0 0 0 0] % Constant term or intercept
[0 1 0 0] % B; equivalently, A^0 * B^1 * C^0
```

```
[1 0 1 0] % A*C
[2 0 0 0] % A^2
[0 1 2 0] % B*(C^2)
```

The 0 at the end of each term represents the response variable. In general,

- If you have the variables in a table or dataset array, then 0 must represent the response variable depending on the position of the response variable. The following example illustrates this.

Load the sample data and define the dataset array.

```
load hospital
dsa = dataset(hospital.Sex,hospital.BloodPressure(:,1),hospital.Age,...
hospital.Smoker,'VarNames',{'Sex','BloodPressure','Age','Smoker'});
```

Represent the linear model 'BloodPressure ~ 1 + Sex + Age + Smoker' in a terms matrix. The response variable is in the second column of the dataset array, so there must be a column of 0s for the response variable in the second column of the terms matrix.

```
T = [0 0 0 0;1 0 0 0;0 0 1 0;0 0 0 1]
```

```
T =
```

```
    0    0    0    0
    1    0    0    0
    0    0    1    0
    0    0    0    1
```

Redefine the dataset array.

```
dsa = dataset(hospital.BloodPressure(:,1),hospital.Sex,hospital.Age,...
hospital.Smoker,'VarNames',{'BloodPressure','Sex','Age','Smoker'});
```

Now, the response variable is the first term in the dataset array. Specify the same linear model, 'BloodPressure ~ 1 + Sex + Age + Smoker', using a terms matrix.

```
T = [0 0 0 0;0 1 0 0;0 0 1 0;0 0 0 1]
```

```
T =
```

```
    0    0    0    0
    0    1    0    0
```



```

0      0      1      0
0      0      0      1

```

- If you have the predictor and response variables in a matrix and column vector, then you must include 0 for the response variable at the end of each term. The following example illustrates this.

Load the sample data and define the matrix of predictors.

```

load carsmall
X = [Acceleration,Weight];

```

Specify the model 'MPG ~ Acceleration + Weight + Acceleration:Weight + Weight^2' using a term matrix and fit the model to the data. This model includes the main effect and two-way interaction terms for the variables, Acceleration and Weight, and a second-order term for the variable, Weight.

```
T = [0 0 0;1 0 0;0 1 0;1 1 0;0 2 0]
```

```
T =
```

```

0      0      0
1      0      0
0      1      0
1      1      0
0      2      0

```

Fit a linear model.

```
mdl = fitlm(X,MPG,T)
```

```
mdl =
```

```
Linear regression model:
```

```
y ~ 1 + x1*x2 + x2^2
```

```
Estimated Coefficients:
```

	Estimate	SE	tStat	pValue
(Intercept)	48.906	12.589	3.8847	0.00019665
x1	0.54418	0.57125	0.95261	0.34337
x2	-0.012781	0.0060312	-2.1192	0.036857
x1:x2	-0.00010892	0.00017925	-0.6076	0.545
x2^2	9.7518e-07	7.5389e-07	1.2935	0.19917

```
Number of observations: 94, Error degrees of freedom: 89
```

```
Root Mean Squared Error: 4.1
R-squared: 0.751, Adjusted R-Squared 0.739
F-statistic vs. constant model: 67, p-value = 4.99e-26
```

Only the intercept and `x2` term, which correspond to the `Weight` variable, are significant at the 5% significance level.

Now, perform a stepwise regression with a constant model as the starting model and a linear model with interactions as the upper model.

```
T = [0 0 0;1 0 0;0 1 0;1 1 0];
mdl = stepwiselm(X,MPG,[0 0 0], 'upper', T)
```

```
1. Adding x2, FStat = 259.3087, pValue = 1.643351e-28
```

```
mdl =
```

```
Linear regression model:
```

```
y ~ 1 + x2
```

```
Estimated Coefficients:
```

	Estimate	SE	tStat	pValue
(Intercept)	49.238	1.6411	30.002	2.7015e-49
x2	-0.0086119	0.0005348	-16.103	1.6434e-28

```
Number of observations: 94, Error degrees of freedom: 92
```

```
Root Mean Squared Error: 4.13
```

```
R-squared: 0.738, Adjusted R-Squared 0.735
```

```
F-statistic vs. constant model: 259, p-value = 1.64e-28
```

The results of the stepwise regression are consistent with the results of `fitlm` in the previous step.

## Formula

A formula for model specification is a character vector of the form `'Y ~ terms'`

where

- `Y` is the response name.
- `terms` contains

- Variable names
- + means include the next variable
- - means do not include the next variable
- : defines an interaction, a product of terms
- \* defines an interaction **and all lower-order terms**
- ^ raises the predictor to a power, exactly as in \* repeated, so ^ includes lower order terms as well
- () groups terms

---

**Note** Formulas include a constant (intercept) term by default. To exclude a constant term from the model, include -1 in the formula.

---

For example,

'Y ~ A + B + C' means a three-variable linear model with intercept.

'Y ~ A + B + C - 1' is a three-variable linear model without intercept.

'Y ~ A + B + C + B^2' is a three-variable model with intercept and a B^2 term.

'Y ~ A + B^2 + C' is the same as the previous example because B^2 includes a B term.

'Y ~ A + B + C + A:B' includes an A\*B term.

'Y ~ A\*B + C' is the same as the previous example because  $A*B = A + B + A:B$ .

'Y ~ A\*B\*C - A:B:C' has all interactions among A, B, and C, except the three-way interaction.

'Y ~ A\*(B + C + D)' has all linear terms, plus products of A with each of the other variables.

## Wilkinson Notation

Wilkinson notation describes the factors present in models. The notation relates to factors present in models, not to the multipliers (coefficients) of those factors.

Wilkinson Notation	Factors in Standard Notation
1	Constant (intercept) term
A^k, where k is a positive integer	A, A <sup>2</sup> , ..., A <sup>k</sup>

Wilkinson Notation	Factors in Standard Notation
A + B	A, B
A*B	A, B, A*B
A:B	A*B only
-B	Do not include B
A*B + C	A, B, C, A*B
A + B + C + A:B	A, B, C, A*B
A*B*C - A:B:C	A, B, C, A*B, A*C, B*C
A* (B + C)	A, B, C, A*B, A*C

Statistics and Machine Learning Toolbox notation always includes a constant term unless you explicitly remove the term using `-1`.

## Canonical Link Function

The default link function for a generalized linear model is the canonical link function.

### Canonical Link Functions for Generalized Linear Models

Distribution	Link Function Name	Link Function	Mean (Inverse) Function
'normal'	'identity'	$f(\mu) = \mu$	$\mu = Xb$
'binomial'	'logit'	$f(\mu) = \log(\mu/(1-\mu))$	$\mu = \exp(Xb) / (1 + \exp(Xb))$
'poisson'	'log'	$f(\mu) = \log(\mu)$	$\mu = \exp(Xb)$
'gamma'	-1	$f(\mu) = 1/\mu$	$\mu = 1/(Xb)$
'inverse gaussian'	-2	$f(\mu) = 1/\mu^2$	$\mu = (Xb)^{-1/2}$

## Tips

- The generalized linear model `mdl` is a standard linear model unless you specify otherwise with the `Distribution` name-value pair.
- For other methods such as `devianceTest`, or properties of the `GeneralizedLinearModel` object, see `GeneralizedLinearModel`.

## Algorithms

Stepwise regression is a systematic method for adding and removing terms from a linear or generalized linear model based on their statistical significance in explaining the response variable. The method begins with an initial model, specified using `modelspec`, and then compares the explanatory power of incrementally larger and smaller models.

MATLAB uses forward and backward stepwise regression to determine a final model. At each step, the method searches for terms to add to or remove from the model based on the value of the 'Criterion' argument. The default value of 'Criterion' is 'sse', and in this case, `stepwiselm` uses the  $p$ -value of an  $F$ -statistic to test models with and without a potential term at each step. If a term is not currently in the model, the null hypothesis is that the term would have a zero coefficient if added to the model. If there is sufficient evidence to reject the null hypothesis, the term is added to the model. Conversely, if a term is currently in the model, the null hypothesis is that the term has a zero coefficient. If there is insufficient evidence to reject the null hypothesis, the term is removed from the model.

Here is how stepwise proceeds when 'Criterion' is 'sse':

- 1 Fit the initial model.
- 2 Examine a set of available terms not in the model. If any of these terms have  $p$ -values less than an entrance tolerance (that is, if it is unlikely that they would have zero coefficient if added to the model), add the one with the smallest  $p$ -value and repeat this step; otherwise, go to step 3.
- 3 If any of the available terms in the model have  $p$ -values greater than an exit tolerance (that is, the hypothesis of a zero coefficient cannot be rejected), remove the one with the largest  $p$ -value and go to step 2; otherwise, end.

At any stage, the function will not add a higher-order term if the model does not also include all lower-order terms that are subsets of it. For example, it will not try to add the term  $X1 : X2^2$  unless both  $X1$  and  $X2^2$  are already in the model. Similarly, the function will not remove lower-order terms that are subsets of higher-order terms that remain in the model. For example, it will not examine to remove  $X1$  or  $X2^2$  if  $X1 : X2^2$  stays in the model.

The default for `stepwiseglm` is 'Deviance' and it follows a similar procedure for adding or removing terms.

There are several other criteria available, which you can specify using the 'Criterion' argument. You can use the change in the value of the Akaike information criterion, Bayesian information criterion, R-squared, adjusted R-squared as a criterion to add or remove terms.

Depending on the terms included in the initial model and the order in which terms are moved in and out, the method might build different models from the same set of potential terms. The method terminates when no single step improves the model. There is no guarantee, however, that a different initial model or a different sequence of steps will not lead to a better fit. In this sense, stepwise models are locally optimal, but might not be globally optimal.

## Alternatives

You can also create a stepwise generalized linear model using `stepwiseglm`.

Use `fitglm` to create a model with a fixed specification. Use `step`, `addTerms`, or `removeTerms` to adjust a fitted model.

## References

- [1] Collett, D. *Modeling Binary Data*. New York: Chapman & Hall, 2002.
- [2] Dobson, A. J. *An Introduction to Generalized Linear Models*. New York: Chapman & Hall, 1990.
- [3] McCullagh, P., and J. A. Nelder. *Generalized Linear Models*. New York: Chapman & Hall, 1990.

## See Also

`GeneralizedLinearModel` | `fitglm` | `stepwiseglm`

## Topics

“Compare large and small stepwise models” on page 11-127

“Generalized Linear Models” on page 12-12

# LinearModel.stepwise

**Class:** LinearModel

Create linear regression model by stepwise regression

---

**Note** `LinearModel.stepwise` will be removed in a future release. Use `stepwiselm` instead.

---

## Syntax

```
mdl = LinearModel.stepwise(tbl,modelspec)
mdl = LinearModel.stepwise(X,y,modelspec)
mdl = LinearModel.stepwise(___,modelspec,Name,Value)
```

## Description

`mdl = LinearModel.stepwise(tbl,modelspec)` returns a linear model of a table or dataset array `tbl`, using stepwise regression to add or remove predictors. `modelspec` is the starting model for the stepwise procedure.

`mdl = LinearModel.stepwise(X,y,modelspec)` creates a linear model of the responses `y` to a data matrix `X`, using stepwise regression to add or remove predictors. `modelspec` is the starting model for the stepwise procedure.

`mdl = LinearModel.stepwise(___,modelspec,Name,Value)` creates a linear model for any of the inputs in the previous syntaxes, with additional options specified by one or more `Name,Value` pair arguments.

For example, you can specify the categorical variables, the smallest or largest set of terms to use in the model, the maximum number of steps to take, or the criterion `LinearModel.stepwise` uses to add or remove terms.

## Input Arguments

### **tbl** — Input data

table | dataset array

Input data, specified as a table or dataset array. When `modelspec` is a formula, it specifies the variables to be used as the predictors and response. Otherwise, if you do not specify the predictor and response variables, the last variable is the response variable and the others are the predictor variables by default.

Predictor variables can be numeric, or any grouping variable type, such as logical or categorical (see “Grouping Variables” on page 2-59). The response must be numeric or logical.

To set a different column as the response variable, use the `ResponseVar` name-value pair argument. To use a subset of the columns as predictors, use the `PredictorVars` name-value pair argument.

### **X** — Predictor variables

matrix

Predictor variables, specified as an  $n$ -by- $p$  matrix, where  $n$  is the number of observations and  $p$  is the number of predictor variables. Each column of `X` represents one variable, and each row represents one observation.

By default, there is a constant term in the model, unless you explicitly remove it, so do not include a column of 1s in `X`.

Data Types: `single` | `double` | `logical`

### **y** — Response variable

vector

Response variable, specified as an  $n$ -by-1 vector, where  $n$  is the number of observations. Each entry in `y` is the response for the corresponding row of `X`.

Data Types: `single` | `double` | `logical`

### **modelspec** — Starting model

character vector specifying the model |  $t$ -by- $(p+1)$  terms matrix | character vector of the form `'Y ~ terms'`



Starting model for the stepwise regression, specified as one of the following:

- Character vector specifying the type of starting model.

Character Vector	Model Type
'constant'	Model contains only a constant (intercept) term.
'linear'	Model contains an intercept and linear terms for each predictor.
'interactions'	Model contains an intercept, linear terms, and all products of pairs of distinct predictors (no squared terms).
'purequadratic'	Model contains an intercept, linear terms, and squared terms.
'quadratic'	Model contains an intercept, linear terms, interactions, and squared terms.
'polyijk'	Model is a polynomial with all terms up to degree $i$ in the first predictor, degree $j$ in the second predictor, etc. Use numerals 0 through 9. For example, 'poly2111' has a constant plus all linear and product terms, and also contains terms with predictor 1 squared.

If you want to specify the smallest or largest set of terms in the model, use the `Lower` and `Upper` name-value pair arguments.

- $t$ -by- $(p+1)$  matrix, namely a terms matrix on page 32-6245, specifying terms to include in model, where  $t$  is the number of terms and  $p$  is the number of predictor variables, and plus one is for the response variable.
- Character vector representing a formula on page 32-6248 in the form

```
'Y ~ terms',
```

where the `terms` are in “Wilkinson Notation” on page 32-6249.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

**CategoricalVars — Categorical variable list**

cell array of character vectors | logical or numeric index vector

Categorical variable list, specified as the comma-separated pair consisting of 'CategoricalVars' and either a cell array of character vectors containing categorical variable names in the table or dataset array `tbl`, or a logical or numeric index vector indicating which columns are categorical.

- If data is in a table or dataset array `tbl`, then, by default, `LinearModel.stepwise` treats all categorical values, logical values, character arrays, and cell arrays of character vectors as categorical variables.
- If data is in matrix `X`, then the default value of 'CategoricalVars' is an empty matrix `[]`. That is, no variable is categorical unless you specify it as categorical.

For example, you can specify the observations 2 and 3 out of 6 as categorical using either of the following examples.

```
Example: 'CategoricalVars', [2,3]
```

```
Example: 'CategoricalVars', logical([0 1 1 0 0 0])
```

Data Types: `single` | `double` | `logical` | `cell`

**Criterion — Criterion to add or remove terms**`'sse'` (default) | `'aic'` | `'bic'` | `'rsquared'` | `'adjrsquared'`

Criterion to add or remove terms, specified as the comma-separated pair consisting of 'Criterion' and one of the following:

- `'sse'` — Default for `stepwiselm`.  $p$ -value for an  $F$ -test of the change in the sum of squared error by adding or removing the term.
- `'aic'` — Change in the value of Akaike information criterion (AIC).
- `'bic'` — Change in the value of Bayesian information criterion (BIC).
- `'rsquared'` — Increase in the value of  $R^2$ .
- `'adjrsquared'` — Increase in the value of adjusted  $R^2$ .

```
Example: 'Criterion', 'bic'
```

**Exclude — Observations to exclude**

logical or numeric index vector

Observations to exclude from the fit, specified as the comma-separated pair consisting of 'Exclude' and a logical or numeric index vector indicating which observations to exclude from the fit.

For example, you can exclude observations 2 and 3 out of 6 using either of the following examples.

```
Example: 'Exclude', [2, 3]
```

```
Example: 'Exclude', logical([0 1 1 0 0 0])
```

```
Data Types: single | double | logical
```

### **Intercept** — Indicator for constant term

```
true (default) | false
```

Indicator for the constant term (intercept) in the fit, specified as the comma-separated pair consisting of 'Intercept' and either `true` to include or `false` to remove the constant term from the model.

Use 'Intercept' only when specifying the model using a character vector, not a formula or matrix.

```
Example: 'Intercept', false
```

### **Lower** — Model specification describing terms that cannot be removed from model

```
'constant' (default)
```

Model specification describing terms that cannot be removed from the model, specified as the comma-separated pair consisting of 'Lower' and one of the options for `modelspec` naming the model.

```
Example: 'Lower', 'linear'
```

### **NSteps** — Number of steps to take

```
no limit (default) | positive integer
```

Number of steps to take, specified as the comma-separated pair consisting of 'NSteps' and a positive integer.

```
Data Types: single | double
```

### **PEnter** — Improvement measure for adding term

```
scalar value
```

Improvement measure for adding a term, specified as the comma-separated pair consisting of 'PEnter' and a scalar value. The default values are below.

Criterion	Default value	Decision
'Deviance'	0.05	If the $p$ -value of $F$ or chi-squared statistic is smaller than PEnter, add the term to the model.
'SSE'	0.05	If the SSE of the model is smaller than PEnter, add the term to the model.
'AIC'	0	If the change in the AIC of the model is smaller than PEnter, add the term to the model.
'BIC'	0	If the change in the BIC of the model is smaller than PEnter, add the term to the model.
'Rsquared'	0.1	If the increase in the R-squared of the model is larger than PEnter, add the term to the model.
'AdjRsquared'	0	If the increase in the adjusted R-squared of the model is larger than PEnter, add the term to the model.

For more information on the criteria, see Criterion name-value pair argument.

Example: 'PEnter', 0.075

### **PredictorVars** — Predictor variables

cell array of character vectors | logical or numeric index vector

Predictor variables to use in the fit, specified as the comma-separated pair consisting of 'PredictorVars' and either a cell array of character vectors of the variable names in

the table or dataset array `tbl`, or a logical or numeric index vector indicating which columns are predictor variables.

The character vectors should be among the names in `tbl`, or the names you specify using the `'VarNames'` name-value pair argument.

The default is all variables in `X`, or all variables in `tbl` except for `ResponseVar`.

For example, you can specify the second and third variables as the predictor variables using either of the following examples.

Example: `'PredictorVars', [2,3]`

Example: `'PredictorVars', logical([0 1 1 0 0 0])`

Data Types: `single` | `double` | `logical` | `cell`

**`PRemove`** — Improvement measure for removing term  
scalar value

Improvement measure for removing a term, specified as the comma-separated pair consisting of `'PRemove'` and a scalar value.

Criterion	Default value	Decision
'Deviance'	0.10	If the $p$ -value of $F$ or chi-squared statistic is larger than <code>PRemove</code> , remove the term from the model.
'SSE'	0.10	If the $p$ -value of the $F$ statistic is larger than <code>PRemove</code> , remove the term from the model.
'AIC'	0.01	If the change in the AIC of the model is larger than <code>PRemove</code> , remove the term from the model.
'BIC'	0.01	If the change in the BIC of the model is larger than <code>PRemove</code> , remove the term from the model.

Criterion	Default value	Decision
'Rsquared'	0.05	If the increase in the R-squared value of the model is smaller than <code>PRemove</code> , remove the term from the model.
'AdjRsquared'	-0.05	If the increase in the adjusted R-squared value of the model is smaller than <code>PRemove</code> , remove the term from the model.

At each step, stepwise algorithm also checks whether any term is redundant (linearly dependent) with other terms in the current model. When any term is linearly dependent with other terms in the current model, it is removed, regardless of the criterion value.

For more information on the criteria, see `Criterion` name-value pair argument.

Example: `'PRemove', 0.05`

### **ResponseVar** — Response variable

last column in `tbl` (default) | character vector containing variable name | logical or numeric index vector

Response variable to use in the fit, specified as the comma-separated pair consisting of `'ResponseVar'` and either a character vector containing the variable name in the table or dataset array `tbl`, or a logical or numeric index vector indicating which column is the response variable. You typically need to use `'ResponseVar'` when fitting a table or dataset array `tbl`.

For example, you can specify the fourth variable, say `yield`, as the response out of six variables, in one of the following ways.

Example: `'ResponseVar', 'yield'`

Example: `'ResponseVar', [4]`

Example: `'ResponseVar', logical([0 0 0 1 0 0])`

Data Types: `single` | `double` | `logical` | `char`

**Upper** — Model specification describing largest set of terms in fit

'interactions' (default) | character vector

Model specification describing the largest set of terms in the fit, specified as the comma-separated pair consisting of 'Upper' and one of the character vector options for `modelspec` naming the model.

Example: 'Upper', 'quadratic'

**VarNames** — Names of variables in fit

{'x1', 'x2', ..., 'xn', 'y'} (default) | cell array of character vectors

Names of variables in fit, specified as the comma-separated pair consisting of 'VarNames' and a cell array of character vectors including the names for the columns of *X* first, and the name for the response variable *y* last.

'VarNames' is not applicable to variables in a table or dataset array, because those variables already have names.

For example, if in your data, horsepower, acceleration, and model year of the cars are the predictor variables, and miles per gallon (MPG) is the response variable, then you can name the variables as follows.

Example: 'VarNames', {'Horsepower', 'Acceleration', 'Model\_Year', 'MPG'}

Data Types: `cell`

**verbose** — Control for display of information

1 (default) | 0 | 2

Control for display of information, specified as the comma-separated pair consisting of 'Verbose' and one of the following:

- 0 — Suppress all display.
- 1 — Display the action taken at each step.
- 2 — Also display the actions evaluated at each step.

Example: 'Verbose', 2

**weights** — Observation weights

`ones(n, 1)` (default) | *n*-by-1 vector of nonnegative scalar values

Observation weights, specified as the comma-separated pair consisting of 'Weights' and an  $n$ -by-1 vector of nonnegative scalar values, where  $n$  is the number of observations.

Data Types: `single` | `double`

## Output Arguments

### **mdl** — Linear model

`LinearModel` object

Linear model representing a least-squares fit of the response to the data, returned as a `LinearModel` object.

For the properties and methods of the linear model object, `mdl`, see the `LinearModel` class page.

## Examples

### Fit a Linear Model Using Stepwise Regression

Load the sample data.

```
load hald
```

`hald` contains hardening data for 13 different concrete compositions. `heat` is the heat of hardening after 180 days. `ingredients` is the percentage of each different ingredient in the cement sample.

Fit a linear model to the data. Set the criterion value to enter the model as 0.06.

```
mdl = stepwiselm(ingredients,heat,'PEnter',0.06)
```

1. Adding `x4`, `FStat` = 22.7985, `pValue` = 0.000576232
2. Adding `x1`, `FStat` = 108.2239, `pValue` = 1.105281e-06
3. Adding `x2`, `FStat` = 5.0259, `pValue` = 0.051687
4. Removing `x4`, `FStat` = 1.8633, `pValue` = 0.2054

```
mdl =
```



```
Linear regression model:
y ~ 1 + x1 + x2
```

```
Estimated Coefficients:
```

	Estimate	SE	tStat	pValue
(Intercept)	52.577	2.2862	22.998	5.4566e-10
x1	1.4683	0.1213	12.105	2.6922e-07
x2	0.66225	0.045855	14.442	5.029e-08

```
Number of observations: 13, Error degrees of freedom: 10
Root Mean Squared Error: 2.41
R-squared: 0.979, Adjusted R-Squared 0.974
F-statistic vs. constant model: 230, p-value = 4.41e-09
```

By default, the starting model is the constant model. `stepwiselm` performs forward selection and `x4`, `x1`, and `x2`, respectively, as the corresponding  $P$ -values are less than the `PEnter` value of 0.06. `stepwiselm` later uses backward elimination and eliminates `x4` from the model because, once `x2` is in the model, the  $P$ -value of `x4` is higher than the default value of `PRemove`, 0.1.

## Stepwise Regression Using Specified Model Formula and Variables

Perform stepwise regression using variables stored in a dataset array. Specify the starting model using Wilkinson notation, and identify the response and predictor variables using optional arguments.

Load the sample data.

```
load hospital
```

The hospital dataset array includes the gender, age, weight, and smoking status of patients.

Fit a linear model with a starting model of a constant term and `Smoker` as the predictor variable. Specify the response variable, `Weight`, and categorical predictor variables, `Sex`, `Age`, and `Smoker`.

```
mdl = stepwiselm(hospital, 'Weight~1+Smoker', ...  
'ResponseVar', 'Weight', 'PredictorVars', {'Sex', 'Age', 'Smoker'}, ...  
'CategoricalVar', {'Sex', 'Smoker'})
```

```
1. Adding Sex, FStat = 770.0158, pValue = 6.262758e-48  
2. Removing Smoker, FStat = 0.21224, pValue = 0.64605
```

```
mdl =
```

```
Linear regression model:  
Weight ~ 1 + Sex
```

```
Estimated Coefficients:
```

	Estimate	SE	tStat	pValue
(Intercept)	130.47	1.1995	108.77	5.2762e-104
Sex_Male	50.06	1.7496	28.612	2.2464e-49

```
Number of observations: 100, Error degrees of freedom: 98  
Root Mean Squared Error: 8.73  
R-squared: 0.893, Adjusted R-Squared 0.892  
F-statistic vs. constant model: 819, p-value = 2.25e-49
```

At each step, `stepwiselm` searches for terms to add and remove. At first step, stepwise algorithm adds `Sex` to the model with a  $P$ -value of  $6.26e-48$ . Then, removes `Smoker` from the model, since given `Sex` in the model, the variable `Smoker` becomes redundant. `stepwiselm` only includes `Sex` in the final linear model. The weight of the patients do not seem to differ significantly according to age or the status of smoking.

- “Compare large and small stepwise models” on page 11-127
- “Linear Regression” on page 11-11

## Definitions

### Terms Matrix

A terms matrix is a  $t$ -by- $(p + 1)$  matrix specifying terms in a model, where  $t$  is the number of terms,  $p$  is the number of predictor variables, and plus one is for the response variable.

The value of  $T(i, j)$  is the exponent of variable  $j$  in term  $i$ . Suppose there are three predictor variables A, B, and C:

```
[0 0 0 0] % Constant term or intercept
[0 1 0 0] % B; equivalently, A^0 * B^1 * C^0
[1 0 1 0] % A*C
[2 0 0 0] % A^2
[0 1 2 0] % B*(C^2)
```

The 0 at the end of each term represents the response variable. In general,

- If you have the variables in a table or dataset array, then 0 must represent the response variable depending on the position of the response variable. The following example illustrates this.

Load the sample data and define the dataset array.

```
load hospital
dsa = dataset(hospital.Sex,hospital.BloodPressure(:,1),hospital.Age,...
hospital.Smoker, 'VarNames', {'Sex', 'BloodPressure', 'Age', 'Smoker'});
```

Represent the linear model 'BloodPressure ~ 1 + Sex + Age + Smoker' in a terms matrix. The response variable is in the second column of the dataset array, so there must be a column of 0s for the response variable in the second column of the terms matrix.

```
T = [0 0 0 0;1 0 0 0;0 0 1 0;0 0 0 1]
```

```
T =
```

```

0     0     0     0
1     0     0     0
0     0     1     0
0     0     0     1
```

Redefine the dataset array.

```
dsa = dataset(hospital.BloodPressure(:,1),hospital.Sex,hospital.Age,...  
hospital.Smoker,'VarNames',{'BloodPressure','Sex','Age','Smoker'});
```

Now, the response variable is the first term in the dataset array. Specify the same linear model, 'BloodPressure ~ 1 + Sex + Age + Smoker', using a terms matrix.

```
T = [0 0 0 0;0 1 0 0;0 0 1 0;0 0 0 1]
```

```
T =
```

```
    0    0    0    0  
    0    1    0    0  
    0    0    1    0  
    0    0    0    1
```

- If you have the predictor and response variables in a matrix and column vector, then you must include 0 for the response variable at the end of each term. The following example illustrates this.

Load the sample data and define the matrix of predictors.

```
load carsmall  
X = [Acceleration,Weight];
```

Specify the model 'MPG ~ Acceleration + Weight + Acceleration:Weight + Weight^2' using a term matrix and fit the model to the data. This model includes the main effect and two-way interaction terms for the variables, Acceleration and Weight, and a second-order term for the variable, Weight.

```
T = [0 0 0;1 0 0;0 1 0;1 1 0;0 2 0]
```

```
T =
```

```
    0    0    0  
    1    0    0  
    0    1    0  
    1    1    0  
    0    2    0
```

Fit a linear model.

```
mdl = fitlm(X,MPG,T)
```

```
mdl =
```

```
Linear regression model:
```

```
y ~ 1 + x1*x2 + x2^2
```

```
Estimated Coefficients:
```

	Estimate	SE	tStat	pValue
(Intercept)	48.906	12.589	3.8847	0.00019665
x1	0.54418	0.57125	0.95261	0.34337
x2	-0.012781	0.0060312	-2.1192	0.036857
x1:x2	-0.00010892	0.00017925	-0.6076	0.545
x2^2	9.7518e-07	7.5389e-07	1.2935	0.19917

```
Number of observations: 94, Error degrees of freedom: 89
```

```
Root Mean Squared Error: 4.1
```

```
R-squared: 0.751, Adjusted R-Squared 0.739
```

```
F-statistic vs. constant model: 67, p-value = 4.99e-26
```

Only the intercept and x2 term, which correspond to the weight variable, are significant at the 5% significance level.

Now, perform a stepwise regression with a constant model as the starting model and a linear model with interactions as the upper model.

```
T = [0 0 0;1 0 0;0 1 0;1 1 0];
```

```
mdl = stepwiselm(X,MPG,[0 0 0], 'upper', T)
```

```
1. Adding x2, FStat = 259.3087, pValue = 1.643351e-28
```

```
mdl =
```

```
Linear regression model:
```

```
y ~ 1 + x2
```

```
Estimated Coefficients:
```

	Estimate	SE	tStat	pValue
(Intercept)	49.238	1.6411	30.002	2.7015e-49
x2	-0.0086119	0.0005348	-16.103	1.6434e-28

```
Number of observations: 94, Error degrees of freedom: 92
```

```
Root Mean Squared Error: 4.13
```

```
R-squared: 0.738, Adjusted R-Squared 0.735
```

```
F-statistic vs. constant model: 259, p-value = 1.64e-28
```

The results of the stepwise regression are consistent with the results of `fitlm` in the previous step.

## Formula

A formula for model specification is a character vector of the form `'Y ~ terms'`

where

- `Y` is the response name.
- `terms` contains
  - Variable names
  - `+` means include the next variable
  - `-` means do not include the next variable
  - `:` defines an interaction, a product of terms
  - `*` defines an interaction **and all lower-order terms**
  - `^` raises the predictor to a power, exactly as in `*` repeated, so `^` includes lower order terms as well
  - `()` groups terms

---

**Note** Formulas include a constant (intercept) term by default. To exclude a constant term from the model, include `-1` in the formula.

---

For example,

`'Y ~ A + B + C'` means a three-variable linear model with intercept.

`'Y ~ A + B + C - 1'` is a three-variable linear model without intercept.

`'Y ~ A + B + C + B^2'` is a three-variable model with intercept and a `B^2` term.

`'Y ~ A + B^2 + C'` is the same as the previous example because `B^2` includes a `B` term.

`'Y ~ A + B + C + A:B'` includes an `A*B` term.

`'Y ~ A*B + C'` is the same as the previous example because `A*B = A + B + A:B`.

`'Y ~ A*B*C - A:B:C'` has all interactions among `A`, `B`, and `C`, except the three-way interaction.

' $Y \sim A*(B + C + D)$ ' has all linear terms, plus products of A with each of the other variables.

## Wilkinson Notation

Wilkinson notation describes the factors present in models. The notation relates to factors present in models, not to the multipliers (coefficients) of those factors.

Wilkinson Notation	Factors in Standard Notation
1	Constant (intercept) term
$A^k$ , where k is a positive integer	A, $A^2$ , ..., $A^k$
$A + B$	A, B
$A*B$	A, B, $A*B$
$A:B$	$A*B$ only
$-B$	Do not include B
$A*B + C$	A, B, C, $A*B$
$A + B + C + A:B$	A, B, C, $A*B$
$A*B*C - A:B:C$	A, B, C, $A*B$ , $A*C$ , $B*C$
$A*(B + C)$	A, B, C, $A*B$ , $A*C$

Statistics and Machine Learning Toolbox notation always includes a constant term unless you explicitly remove the term using `-1`.

## Tips

- You cannot use robust regression with stepwise regression. Check your data for outliers before using `LinearModel.stepwise`.
- For other methods or properties of the `LinearModel` object, see `LinearModel`.

## Algorithms

Stepwise regression is a systematic method for adding and removing terms from a linear or generalized linear model based on their statistical significance in explaining the

response variable. The method begins with an initial model, specified using `modelspec`, and then compares the explanatory power of incrementally larger and smaller models.

MATLAB uses forward and backward stepwise regression to determine a final model. At each step, the method searches for terms to add to or remove from the model based on the value of the 'Criterion' argument. The default value of 'Criterion' is 'sse', and in this case, `stepwiselm` uses the  $p$ -value of an  $F$ -statistic to test models with and without a potential term at each step. If a term is not currently in the model, the null hypothesis is that the term would have a zero coefficient if added to the model. If there is sufficient evidence to reject the null hypothesis, the term is added to the model. Conversely, if a term is currently in the model, the null hypothesis is that the term has a zero coefficient. If there is insufficient evidence to reject the null hypothesis, the term is removed from the model.

Here is how stepwise proceeds when 'Criterion' is 'sse':

- 1 Fit the initial model.
- 2 Examine a set of available terms not in the model. If any of these terms have  $p$ -values less than an entrance tolerance (that is, if it is unlikely that they would have zero coefficient if added to the model), add the one with the smallest  $p$ -value and repeat this step; otherwise, go to step 3.
- 3 If any of the available terms in the model have  $p$ -values greater than an exit tolerance (that is, the hypothesis of a zero coefficient cannot be rejected), remove the one with the largest  $p$ -value and go to step 2; otherwise, end.

At any stage, the function will not add a higher-order term if the model does not also include all lower-order terms that are subsets of it. For example, it will not try to add the term  $X1 : X2^2$  unless both  $X1$  and  $X2^2$  are already in the model. Similarly, the function will not remove lower-order terms that are subsets of higher-order terms that remain in the model. For example, it will not examine to remove  $X1$  or  $X2^2$  if  $X1 : X2^2$  stays in the model.

The default for `stepwiseglm` is 'Deviance' and it follows a similar procedure for adding or removing terms.

There are several other criteria available, which you can specify using the 'Criterion' argument. You can use the change in the value of the Akaike information criterion, Bayesian information criterion, R-squared, adjusted R-squared as a criterion to add or remove terms.



Depending on the terms included in the initial model and the order in which terms are moved in and out, the method might build different models from the same set of potential terms. The method terminates when no single step improves the model. There is no guarantee, however, that a different initial model or a different sequence of steps will not lead to a better fit. In this sense, stepwise models are locally optimal, but might not be globally optimal.

## Alternatives

You can also construct a stepwise linear model using `stepwiselm`.

You can construct a model using `fitlm`, then manually adjust the model using `step`, `addTerms`, or `removeTerms`. Use `fitlm` for robust regression. You cannot use robust regression and stepwise regression together.

## References

[1] Draper, N. R., and H. Smith. *Applied Regression Analysis*. Hoboken, NJ: Wiley-Interscience, pp. 307–312, 1998.

## See Also

`LinearModel` | `fitlm` | `step`

## Topics

“Compare large and small stepwise models” on page 11-127

“Linear Regression” on page 11-11

“Stepwise Regression” on page 11-127

## stepwiseglm

Create generalized linear regression model by stepwise regression

### Syntax

```
mdl = stepwiseglm(tbl,modelspec)
mdl = stepwiseglm(X,y,modelspec)
mdl = stepwiseglm(...,modelspec,Name,Value)
```

### Description

`mdl = stepwiseglm(tbl,modelspec)` creates a generalized linear model of a table or dataset array `tbl`, using stepwise regression to add or remove predictors. `modelspec` is the starting model for the stepwise procedure.

`mdl = stepwiseglm(X,y,modelspec)` creates a generalized linear model of the responses `y` to a data matrix `X`, using stepwise regression to add or remove predictors.

`mdl = stepwiseglm(...,modelspec,Name,Value)` creates a generalized linear model with additional options specified by one or more `Name, Value` pair arguments.

### Examples

#### Generalized Linear Model Using Stepwise Algorithm

Create response data using just three of 20 predictors, and create a generalized linear model using stepwise algorithm to see if it uses just the correct predictors.

Create data with 20 predictors, and Poisson response using just three of the predictors, plus a constant.

```
rng('default') % for reproducibility
X = randn(100,20);
```

```
mu = exp(X(:, [5 10 15])*[.4;.2;.3] + 1);
y = poissrnd(mu);
```

Fit a generalized linear model using the Poisson distribution.

```
mdl = stepwiseglm(X,y,...
    'constant','upper','linear','Distribution','poisson')
```

```
1. Adding x5, Deviance = 134.439, Chi2Stat = 52.24814, PValue = 4.891229e-13
2. Adding x15, Deviance = 106.285, Chi2Stat = 28.15393, PValue = 1.1204e-07
3. Adding x10, Deviance = 95.0207, Chi2Stat = 11.2644, PValue = 0.000790094
```

```
mdl =
```

Generalized linear regression model:

```
log(y) ~ 1 + x5 + x10 + x15
Distribution = Poisson
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	1.0115	0.064275	15.737	8.4217e-56
x5	0.39508	0.066665	5.9263	3.0977e-09
x10	0.18863	0.05534	3.4085	0.0006532
x15	0.29295	0.053269	5.4995	3.8089e-08

100 observations, 96 error degrees of freedom

Dispersion: 1

Chi<sup>2</sup>-statistic vs. constant model: 91.7, p-value = 9.61e-20

The starting model is the constant model. `stepwiseglm` by default uses deviance of the model as the criterion. It first adds `x5` into the model, as the *P*-value for the test statistic, deviance (the differences in the deviances of the two models), is less than the default threshold value 0.05. Then, it adds `x15` because given `x5` is in the model, when `x15` is added, the *P*-value for chi-squared test is smaller than 0.05. It then adds `x10` because given `x5` and `x15` are in the model, when `x10` is added, the *P*-value for the chi-square test statistic is again less than 0.05.

- “Compare large and small stepwise models” on page 11-127

## Input Arguments

### **tbl** — Input data

table | dataset array

Input data, specified as a table or dataset array. When `modelspec` is a formula, it specifies the variables to be used as the predictors and response. Otherwise, if you do not specify the predictor and response variables, the last variable is the response variable and the others are the predictor variables by default.

Predictor variables can be numeric, or any grouping variable type, such as logical or categorical (see “Grouping Variables” on page 2-59). The response must be numeric or logical.

To set a different column as the response variable, use the `ResponseVar` name-value pair argument. To use a subset of the columns as predictors, use the `PredictorVars` name-value pair argument.

### **X** — Predictor variables

matrix

Predictor variables, specified as an  $n$ -by- $p$  matrix, where  $n$  is the number of observations and  $p$  is the number of predictor variables. Each column of `X` represents one variable, and each row represents one observation.

By default, there is a constant term in the model, unless you explicitly remove it, so do not include a column of 1s in `X`.

Data Types: `single` | `double` | `logical`

### **y** — Response variable

vector

Response variable, specified as an  $n$ -by-1 vector, where  $n$  is the number of observations. Each entry in `y` is the response for the corresponding row of `X`.

Data Types: `single` | `double` | `logical`

### **modelspec** — Starting model

character vector specifying the model |  $t$ -by- $(p+1)$  terms matrix | character vector of the form `'Y ~ terms'`

Starting model for `stepwiseglm`, specified as one of the following:

- Character vector specifying the type of the starting model.

Character Vector	Starting Model Type
'constant'	Model contains only a constant (intercept) term.
'linear'	Model contains an intercept and linear terms for each predictor.
'interactions'	Model contains an intercept, linear terms for each predictor, and all products of pairs of distinct predictors (no squared terms).
'purequadratic'	Model contains an intercept, linear terms, and squared terms for each predictor.
'quadratic'	Model contains an intercept, linear terms, interactions, and squared terms for each predictor.
'polyijk'	Model is a polynomial with all terms up to degree $i$ in the first predictor, degree $j$ in the second predictor, etc. Use numerals 0 through 9. For example, 'poly2111' has a constant plus all linear* and product terms, and also contains terms with predictor 1 squared.

If you want to specify the smallest or largest set of terms in the model that `stepwiseglm` fits, use the `Lower` and `Upper` name-value pair arguments.

- $t$ -by- $(p+1)$  matrix, namely terms matrix on page 32-6264, specifying terms to include in model, where  $t$  is the number of terms and  $p$  is the number of predictor variables, and plus one is for the response variable.
- Character vector representing a formula on page 32-6267 in the form

```
'Y ~ terms',
```

where the `terms` are in “Wilkinson Notation” on page 32-6268.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example:

`'Criterion', 'aic', 'Distribution', 'poisson', 'Upper', 'interactions'` specifies Akaike Information Criterion as the criterion to add or remove variables to the model, Poisson distribution as the distribution of the response variable, and a model with all possible interactions as the largest model to consider as the fit.

### **BinomialSize** — Number of trials for binomial distribution

1 (default) | scalar value | vector

Number of trials for binomial distribution, that is the sample size, specified as the comma-separated pair consisting of a scalar value or a vector of the same length as the response. This is the parameter `n` for the fitted binomial distribution. `BinomialSize` applies only when the `Distribution` parameter is `'binomial'`.

If `BinomialSize` is a scalar value, that means all observations have the same number of trials.

As an alternative to `BinomialSize`, you can specify the response as a two-column vector with counts in column 1 and `BinomialSize` in column 2.

Data Types: `single` | `double`

### **CategoricalVars** — Categorical variable list

cell array of character vectors | logical or numeric index vector

Categorical variable list, specified as the comma-separated pair consisting of `'CategoricalVars'` and either a cell array of character vectors containing categorical variable names in the table or dataset array `tbl`, or a logical or numeric index vector indicating which columns are categorical.

- If data is in a table or dataset array `tbl`, then, by default, `stepwiseglm` treats all categorical values, logical values, character arrays, and cell arrays of character vectors as categorical variables.
- If data is in matrix `X`, then the default value of `'CategoricalVars'` is an empty matrix `[]`. That is, no variable is categorical unless you specify it as categorical.

For example, you can specify the observations 2 and 3 out of 6 as categorical using either of the following examples.

Example: `'CategoricalVars', [2,3]`

Example: `'CategoricalVars', logical([0 1 1 0 0 0])`

Data Types: `single` | `double` | `logical` | `cell`

### Criterion — Criterion to add or remove terms

`'Deviance'` (default) | `'sse'` | `'aic'` | `'bic'` | `'rsquared'` | `'adjrsquared'`

Criterion to add or remove terms, specified as the comma-separated pair consisting of `'Criterion'` and one of the following:

- `'Deviance'` — Default for `stepwiseglm`.  $p$ -value for  $F$  or chi-squared test of the change in the deviance by adding or removing the term.  $F$ -test is for testing a single model. Chi-squared test is for comparing two different models. This option is not valid for `stepwiselm`.
- `'sse'` — Default for `stepwiselm`.  $p$ -value for an  $F$ -test of the change in the sum of squared error by adding or removing the term.
- `'aic'` — Change in the value of Akaike information criterion (AIC).
- `'bic'` — Change in the value of Bayesian information criterion (BIC).
- `'rsquared'` — Increase in the value of  $R^2$ .
- `'adjrsquared'` — Increase in the value of adjusted  $R^2$ .

Example: `'Criterion','bic'`

### DispersionFlag — Indicator to compute dispersion parameter

`false` for `'binomial'` and `'poisson'` distributions (default) | `true`

Indicator to compute dispersion parameter for `'binomial'` and `'poisson'` distributions, specified as the comma-separated pair consisting of `'DispersionFlag'` and one of the following.

<code>true</code>	Estimate a dispersion parameter when computing standard errors
<code>false</code>	Default. Use the theoretical value when computing standard errors

The fitting function always estimates the dispersion for other distributions.

Example: `'DispersionFlag',true`

### Distribution — Distribution of the response variable

`'normal'` (default) | `'binomial'` | `'poisson'` | `'gamma'` | `'inverse gaussian'`

Distribution of the response variable, specified as the comma-separated pair consisting of 'Distribution' and one of the following.

'normal'	Normal distribution
'binomial'	Binomial distribution
'poisson'	Poisson distribution
'gamma'	Gamma distribution
'inverse gaussian'	Inverse Gaussian distribution

Example: `'Distribution', 'gamma'`

#### **Exclude** — Observations to exclude

logical or numeric index vector

Observations to exclude from the fit, specified as the comma-separated pair consisting of 'Exclude' and a logical or numeric index vector indicating which observations to exclude from the fit.

For example, you can exclude observations 2 and 3 out of 6 using either of the following examples.

Example: `'Exclude', [2,3]`

Example: `'Exclude', logical([0 1 1 0 0 0])`

Data Types: `single` | `double` | `logical`

#### **Intercept** — Indicator for constant term

`true` (default) | `false`

Indicator for the constant term (intercept) in the fit, specified as the comma-separated pair consisting of 'Intercept' and either `true` to include or `false` to remove the constant term from the model.

Use 'Intercept' only when specifying the model using a character vector, not a formula or matrix.

Example: `'Intercept', false`

#### **Link** — Link function

The canonical link function (default) | scalar value | structure



Link function to use in place of the canonical link function, specified as the comma-separated pair consisting of 'Link' and one of the following.

Link Function Name	Link Function	Mean (Inverse) Function
'identity'	$f(\mu) = \mu$	$\mu = Xb$
'log'	$f(\mu) = \log(\mu)$	$\mu = \exp(Xb)$
'logit'	$f(\mu) = \log(\mu/(1-\mu))$	$\mu = \exp(Xb) / (1 + \exp(Xb))$
'probit'	$f(\mu) = \Phi^{-1}(\mu)$	$\mu = \Phi(Xb)$
'comploglog'	$f(\mu) = \log(-\log(1 - \mu))$	$\mu = 1 - \exp(-\exp(Xb))$
'reciprocal'	$f(\mu) = 1/\mu$	$\mu = 1/(Xb)$
p (a number)	$f(\mu) = \mu^p$	$\mu = Xb^{1/p}$
S (a structure) with three fields. Each field holds a function handle that accepts a vector of inputs and returns a vector of the same size: <ul style="list-style-type: none"> <li>• S.Link — The link function</li> <li>• S.Inverse — The inverse link function</li> <li>• S.Derivative — The derivative of the link function</li> </ul>	$f(\mu) = S.Link(\mu)$	$\mu = S.Inverse(Xb)$

The link function defines the relationship  $f(\mu) = X^*b$  between the mean response  $\mu$  and the linear combination of predictors  $X^*b$ .

For more information on the canonical link functions, see [Definitions](#).

Example: 'Link', 'probit'

**Lower** — Model specification describing terms that cannot be removed from model  
'constant' (default)

Model specification describing terms that cannot be removed from the model, specified as the comma-separated pair consisting of 'Lower' and one of the options for `modelspec` naming the model.

Example: 'Lower', 'linear'

### **Offset** — Offset variable

[] (default) | vector | character vector

Offset variable in the fit, specified as the comma-separated pair consisting of 'Offset' and a vector or name of a variable with the same length as the response.

`fitglm` and `stepwiseglm` use `Offset` as an additional predictor, with a coefficient value fixed at 1.0. In other words, the formula for fitting is

$$\mu \sim \text{Offset} + (\text{terms involving real predictors})$$

with the `Offset` predictor having coefficient 1.

For example, consider a Poisson regression model. Suppose the number of counts is known for theoretical reasons to be proportional to a predictor  $A$ . By using the log link function and by specifying `log(A)` as an offset, you can force the model to satisfy this theoretical constraint.

Data Types: `single` | `double` | `char`

### **PEnter** — Improvement measure for adding term

scalar value

Improvement measure for adding a term, specified as the comma-separated pair consisting of 'PEnter' and a scalar value. The default values are below.

Criterion	Default value	Decision
'Deviance'	0.05	If the $p$ -value of $F$ or chi-squared statistic is smaller than <code>PEnter</code> , add the term to the model.
'SSE'	0.05	If the SSE of the model is smaller than <code>PEnter</code> , add the term to the model.

Criterion	Default value	Decision
'AIC'	0	If the change in the AIC of the model is smaller than <code>PEnter</code> , add the term to the model.
'BIC'	0	If the change in the BIC of the model is smaller than <code>PEnter</code> , add the term to the model.
'Rsquared'	0.1	If the increase in the R-squared of the model is larger than <code>PEnter</code> , add the term to the model.
'AdjRsquared'	0	If the increase in the adjusted R-squared of the model is larger than <code>PEnter</code> , add the term to the model.

For more information on the criteria, see `Criterion` name-value pair argument.

Example: `'PEnter', 0.075`

### **PredictorVars** — Predictor variables

cell array of character vectors | logical or numeric index vector

Predictor variables to use in the fit, specified as the comma-separated pair consisting of `'PredictorVars'` and either a cell array of character vectors of the variable names in the table or dataset array `tbl`, or a logical or numeric index vector indicating which columns are predictor variables.

The character vectors should be among the names in `tbl`, or the names you specify using the `'VarNames'` name-value pair argument.

The default is all variables in `X`, or all variables in `tbl` except for `ResponseVar`.

For example, you can specify the second and third variables as the predictor variables using either of the following examples.

Example: `'PredictorVars', [2, 3]`

Example: `'PredictorVars', logical([0 1 1 0 0 0])`

Data Types: `single` | `double` | `logical` | `cell`

### **PRemove** — Improvement measure for removing term

scalar value

Improvement measure for removing a term, specified as the comma-separated pair consisting of 'PRemove' and a scalar value.

Criterion	Default value	Decision
'Deviance'	0.10	If the $p$ -value of $F$ or chi-squared statistic is larger than PRemove, remove the term from the model.
'SSE'	0.10	If the $p$ -value of the $F$ statistic is larger than PRemove, remove the term from the model.
'AIC'	0.01	If the change in the AIC of the model is larger than PRemove, remove the term from the model.
'BIC'	0.01	If the change in the BIC of the model is larger than PRemove, remove the term from the model.
'Rsquared'	0.05	If the increase in the R-squared value of the model is smaller than PRemove, remove the term from the model.
'AdjRsquared'	-0.05	If the increase in the adjusted R-squared value of the model is smaller than PRemove, remove the term from the model.

At each step, stepwise algorithm also checks whether any term is redundant (linearly dependent) with other terms in the current model. When any term is linearly dependent with other terms in the current model, it is removed, regardless of the criterion value.

For more information on the criteria, see `Criterion` name-value pair argument.

Example: `'PRemove', 0.05`

### **ResponseVar** — Response variable

last column in `tbl` (default) | character vector containing variable name | logical or numeric index vector

Response variable to use in the fit, specified as the comma-separated pair consisting of `'ResponseVar'` and either a character vector containing the variable name in the table or dataset array `tbl`, or a logical or numeric index vector indicating which column is the response variable. You typically need to use `'ResponseVar'` when fitting a table or dataset array `tbl`.

For example, you can specify the fourth variable, say `yield`, as the response out of six variables, in one of the following ways.

Example: `'ResponseVar', 'yield'`

Example: `'ResponseVar', [4]`

Example: `'ResponseVar', logical([0 0 0 1 0 0])`

Data Types: `single` | `double` | `logical` | `char`

### **Upper** — Model specification describing largest set of terms in fit

`'interactions'` (default) | character vector

Model specification describing the largest set of terms in the fit, specified as the comma-separated pair consisting of `'Upper'` and one of the character vector options for `modelspec` naming the model.

Example: `'Upper', 'quadratic'`

### **VarNames** — Names of variables in fit

`{'x1', 'x2', ..., 'xn', 'y'}` (default) | cell array of character vectors

Names of variables in fit, specified as the comma-separated pair consisting of `'VarNames'` and a cell array of character vectors including the names for the columns of `X` first, and the name for the response variable `y` last.

'VarNames' is not applicable to variables in a table or dataset array, because those variables already have names.

For example, if in your data, horsepower, acceleration, and model year of the cars are the predictor variables, and miles per gallon (MPG) is the response variable, then you can name the variables as follows.

Example: `'VarNames', {'Horsepower', 'Acceleration', 'Model_Year', 'MPG'}`

Data Types: `cell`

### **weights** — Observation weights

`ones(n,1)` (default) |  $n$ -by-1 vector of nonnegative scalar values

Observation weights, specified as the comma-separated pair consisting of 'Weights' and an  $n$ -by-1 vector of nonnegative scalar values, where  $n$  is the number of observations.

Data Types: `single` | `double`

## Output Arguments

### **mdl** — Generalized linear model

`GeneralizedLinearModel` object

Generalized linear model representing a least-squares fit of the link of the response to the data, returned as a `GeneralizedLinearModel` object.

For properties and methods of the generalized linear model object, `mdl`, see the `GeneralizedLinearModel` class page.

## Definitions

### Terms Matrix

A terms matrix is a  $t$ -by- $(p + 1)$  matrix specifying terms in a model, where  $t$  is the number of terms,  $p$  is the number of predictor variables, and plus one is for the response variable.

The value of  $T(i, j)$  is the exponent of variable  $j$  in term  $i$ . Suppose there are three predictor variables  $A$ ,  $B$ , and  $C$ :

```
[0 0 0 0] % Constant term or intercept
[0 1 0 0] % B; equivalently, A^0 * B^1 * C^0
[1 0 1 0] % A*C
[2 0 0 0] % A^2
[0 1 2 0] % B*(C^2)
```

The 0 at the end of each term represents the response variable. In general,

- If you have the variables in a table or dataset array, then 0 must represent the response variable depending on the position of the response variable. The following example illustrates this.

Load the sample data and define the dataset array.

```
load hospital
dsa = dataset(hospital.Sex,hospital.BloodPressure(:,1),hospital.Age,...
hospital.Smoker,'VarNames',{'Sex','BloodPressure','Age','Smoker'});
```

Represent the linear model 'BloodPressure ~ 1 + Sex + Age + Smoker' in a terms matrix. The response variable is in the second column of the dataset array, so there must be a column of 0s for the response variable in the second column of the terms matrix.

```
T = [0 0 0 0;1 0 0 0;0 0 1 0;0 0 0 1]
```

```
T =
```

```
    0    0    0    0
    1    0    0    0
    0    0    1    0
    0    0    0    1
```

Redefine the dataset array.

```
dsa = dataset(hospital.BloodPressure(:,1),hospital.Sex,hospital.Age,...
hospital.Smoker,'VarNames',{'BloodPressure','Sex','Age','Smoker'});
```

Now, the response variable is the first term in the dataset array. Specify the same linear model, 'BloodPressure ~ 1 + Sex + Age + Smoker', using a terms matrix.

```
T = [0 0 0 0;0 1 0 0;0 0 1 0;0 0 0 1]
```

```
T =  
  
    0    0    0    0  
    0    1    0    0  
    0    0    1    0  
    0    0    0    1
```

- If you have the predictor and response variables in a matrix and column vector, then you must include 0 for the response variable at the end of each term. The following example illustrates this.

Load the sample data and define the matrix of predictors.

```
load carsmall  
X = [Acceleration,Weight];
```

Specify the model 'MPG ~ Acceleration + Weight + Acceleration:Weight + Weight^2' using a term matrix and fit the model to the data. This model includes the main effect and two-way interaction terms for the variables, Acceleration and Weight, and a second-order term for the variable, Weight.

```
T = [0 0 0;1 0 0;0 1 0;1 1 0;0 2 0]
```

```
T =  
  
    0    0    0  
    1    0    0  
    0    1    0  
    1    1    0  
    0    2    0
```

Fit a linear model.

```
mdl = fitlm(X,MPG,T)
```

```
mdl =
```

```
Linear regression model:  
y ~ 1 + x1*x2 + x2^2
```

```
Estimated Coefficients:
```

	Estimate	SE	tStat	pValue
(Intercept)	48.906	12.589	3.8847	0.00019665
x1	0.54418	0.57125	0.95261	0.34337
x2	-0.012781	0.0060312	-2.1192	0.036857



x1:x2	-0.00010892	0.00017925	-0.6076	0.545
x2^2	9.7518e-07	7.5389e-07	1.2935	0.19917

Number of observations: 94, Error degrees of freedom: 89  
 Root Mean Squared Error: 4.1  
 R-squared: 0.751, Adjusted R-Squared 0.739  
 F-statistic vs. constant model: 67, p-value = 4.99e-26

Only the intercept and x2 term, which correspond to the Weight variable, are significant at the 5% significance level.

Now, perform a stepwise regression with a constant model as the starting model and a linear model with interactions as the upper model.

```
T = [0 0 0;1 0 0;0 1 0;1 1 0];
mdl = stepwiselm(X,MPG,[0 0 0], 'upper', T)
```

1. Adding x2, FStat = 259.3087, pValue = 1.643351e-28

```
mdl =
```

```
Linear regression model:
y ~ 1 + x2
```

```
Estimated Coefficients:
```

	Estimate	SE	tStat	pValue
(Intercept)	49.238	1.6411	30.002	2.7015e-49
x2	-0.0086119	0.0005348	-16.103	1.6434e-28

Number of observations: 94, Error degrees of freedom: 92  
 Root Mean Squared Error: 4.13  
 R-squared: 0.738, Adjusted R-Squared 0.735  
 F-statistic vs. constant model: 259, p-value = 1.64e-28

The results of the stepwise regression are consistent with the results of fitlm in the previous step.

## Formula

A formula for model specification is a character vector of the form ' $Y \sim terms$ '

where

- $Y$  is the response name.
- `terms` contains
  - Variable names
  - `+` means include the next variable
  - `-` means do not include the next variable
  - `:` defines an interaction, a product of terms
  - `*` defines an interaction **and all lower-order terms**
  - `^` raises the predictor to a power, exactly as in `*` repeated, so `^` includes lower order terms as well
  - `()` groups terms

---

**Note** Formulas include a constant (intercept) term by default. To exclude a constant term from the model, include `-1` in the formula.

---

For example,

' $Y \sim A + B + C$ ' means a three-variable linear model with intercept.

' $Y \sim A + B + C - 1$ ' is a three-variable linear model without intercept.

' $Y \sim A + B + C + B^2$ ' is a three-variable model with intercept and a  $B^2$  term.

' $Y \sim A + B^2 + C$ ' is the same as the previous example because  $B^2$  includes a  $B$  term.

' $Y \sim A + B + C + A:B$ ' includes an  $A*B$  term.

' $Y \sim A*B + C$ ' is the same as the previous example because  $A*B = A + B + A:B$ .

' $Y \sim A*B*C - A:B:C$ ' has all interactions among  $A$ ,  $B$ , and  $C$ , except the three-way interaction.

' $Y \sim A*(B + C + D)$ ' has all linear terms, plus products of  $A$  with each of the other variables.

## Wilkinson Notation

Wilkinson notation describes the factors present in models. The notation relates to factors present in models, not to the multipliers (coefficients) of those factors.

Wilkinson Notation	Factors in Standard Notation
1	Constant (intercept) term
$A^k$ , where $k$ is a positive integer	$A, A^2, \dots, A^k$
$A + B$	$A, B$
$A*B$	$A, B, A*B$
$A:B$	$A*B$ only
$-B$	Do not include $B$
$A*B + C$	$A, B, C, A*B$
$A + B + C + A:B$	$A, B, C, A*B$
$A*B*C - A:B:C$	$A, B, C, A*B, A*C, B*C$
$A*(B + C)$	$A, B, C, A*B, A*C$

Statistics and Machine Learning Toolbox notation always includes a constant term unless you explicitly remove the term using `-1`.

## Canonical Function

The default link function for a generalized linear model is the canonical link function.

### Canonical Link Functions for Generalized Linear Models

Distribution	Link Function Name	Link Function	Mean (Inverse) Function
'normal'	'identity'	$f(\mu) = \mu$	$\mu = Xb$
'binomial'	'logit'	$f(\mu) = \log(\mu/(1-\mu))$	$\mu = \exp(Xb) / (1 + \exp(Xb))$
'poisson'	'log'	$f(\mu) = \log(\mu)$	$\mu = \exp(Xb)$
'gamma'	-1	$f(\mu) = 1/\mu$	$\mu = 1/(Xb)$
'inverse gaussian'	-2	$f(\mu) = 1/\mu^2$	$\mu = (Xb)^{-1/2}$

## Tips

- The generalized linear model `mdl` is a standard linear model unless you specify otherwise with the `Distribution` name-value pair.

- For other methods such as `devianceTest`, or properties of the `GeneralizedLinearModel` object, see `GeneralizedLinearModel`.
- After training a `GeneralizedLinearModel` model object by using `stepwiseglm`, use the function `predict` and the trained model object to generate C code that predicts responses for new data. For details, see “Code Generation” on page 32-0 .

## Algorithms

Stepwise regression is a systematic method for adding and removing terms from a linear or generalized linear model based on their statistical significance in explaining the response variable. The method begins with an initial model, specified using `modelspec`, and then compares the explanatory power of incrementally larger and smaller models.

MATLAB uses forward and backward stepwise regression to determine a final model. At each step, the method searches for terms to add to or remove from the model based on the value of the 'Criterion' argument. The default value of 'Criterion' is 'sse', and in this case, `stepwiselm` uses the  $p$ -value of an  $F$ -statistic to test models with and without a potential term at each step. If a term is not currently in the model, the null hypothesis is that the term would have a zero coefficient if added to the model. If there is sufficient evidence to reject the null hypothesis, the term is added to the model. Conversely, if a term is currently in the model, the null hypothesis is that the term has a zero coefficient. If there is insufficient evidence to reject the null hypothesis, the term is removed from the model.

Here is how stepwise proceeds when 'Criterion' is 'sse':

- 1 Fit the initial model.
- 2 Examine a set of available terms not in the model. If any of these terms have  $p$ -values less than an entrance tolerance (that is, if it is unlikely that they would have zero coefficient if added to the model), add the one with the smallest  $p$ -value and repeat this step; otherwise, go to step 3.
- 3 If any of the available terms in the model have  $p$ -values greater than an exit tolerance (that is, the hypothesis of a zero coefficient cannot be rejected), remove the one with the largest  $p$ -value and go to step 2; otherwise, end.

At any stage, the function will not add a higher-order term if the model does not also include all lower-order terms that are subsets of it. For example, it will not try to add the term `X1:X2^2` unless both `X1` and `X2^2` are already in the model. Similarly, the function

will not remove lower-order terms that are subsets of higher-order terms that remain in the model. For example, it will not examine to remove  $X_1$  or  $X_2^2$  if  $X_1 : X_2^2$  stays in the model.

The default for `stepwiseglm` is 'Deviance' and it follows a similar procedure for adding or removing terms.

There are several other criteria available, which you can specify using the 'Criterion' argument. You can use the change in the value of the Akaike information criterion, Bayesian information criterion, R-squared, adjusted R-squared as a criterion to add or remove terms.

Depending on the terms included in the initial model and the order in which terms are moved in and out, the method might build different models from the same set of potential terms. The method terminates when no single step improves the model. There is no guarantee, however, that a different initial model or a different sequence of steps will not lead to a better fit. In this sense, stepwise models are locally optimal, but might not be globally optimal.

## Alternatives

Use `fitglm` to create a model with a fixed specification. Use `step`, `addTerms`, or `removeTerms` to adjust a fitted model.

## References

- [1] Collett, D. *Modeling Binary Data*. New York: Chapman & Hall, 2002.
- [2] Dobson, A. J. *An Introduction to Generalized Linear Models*. New York: Chapman & Hall, 1990.
- [3] McCullagh, P., and J. A. Nelder. *Generalized Linear Models*. New York: Chapman & Hall, 1990.

## See Also

`GeneralizedLinearModel` | `fitglm` | `predict`

## **Topics**

“Compare large and small stepwise models” on page 11-127

“Generalized Linear Models” on page 12-12

**Introduced in R2013b**

# stepwiselm

Create linear regression model using stepwise regression

## Syntax

```
mdl = stepwiselm(tbl,modelspec)
mdl = stepwiselm(X,y,modelspec)
mdl = stepwiselm( ____,Name,Value)
```

## Description

`mdl = stepwiselm(tbl,modelspec)` returns a linear model for the variables in the table or dataset array `tbl` using stepwise regression to add or remove predictors. `stepwiselm` uses forward and backward stepwise regression to determine a final model. At each step, the function searches for terms to add to or remove from the model based on the value of the 'Criterion' argument. `modelspec` is the starting model for the stepwise procedure.

`mdl = stepwiselm(X,y,modelspec)` creates a linear model of the responses `y` to the predictor variables in the data matrix `X`, using stepwise regression to add or remove predictors. `modelspec` is the starting model for the stepwise procedure.

`mdl = stepwiselm( ____,Name,Value)` creates a linear model for any of the inputs in the previous syntaxes, with additional options specified by one or more `Name,Value` pair arguments.

For example, you can specify the categorical variables, the smallest or largest set of terms to use in the model, the maximum number of steps to take, or the criterion `stepwiselm` uses to add or remove terms.

## Examples

### Fit a Linear Model Using Stepwise Regression

Load the sample data.

```
load hald
```

`hald` contains hardening data for 13 different concrete compositions. `heat` is the heat of hardening after 180 days. `ingredients` is the percentage of each different ingredient in the cement sample.

Fit a linear model to the data. Set the criterion value to enter the model as 0.06.

```
mdl = stepwiselm(ingredients,heat,'PEnter',0.06)
```

1. Adding x4, FStat = 22.7985, pValue = 0.000576232
2. Adding x1, FStat = 108.2239, pValue = 1.105281e-06
3. Adding x2, FStat = 5.0259, pValue = 0.051687
4. Removing x4, FStat = 1.8633, pValue = 0.2054

```
mdl =
```

```
Linear regression model:
```

```
y ~ 1 + x1 + x2
```

```
Estimated Coefficients:
```

	Estimate	SE	tStat	pValue
(Intercept)	52.577	2.2862	22.998	5.4566e-10
x1	1.4683	0.1213	12.105	2.6922e-07
x2	0.66225	0.045855	14.442	5.029e-08

```
Number of observations: 13, Error degrees of freedom: 10
```

```
Root Mean Squared Error: 2.41
```

```
R-squared: 0.979, Adjusted R-Squared 0.974
```

```
F-statistic vs. constant model: 230, p-value = 4.41e-09
```

By default, the starting model is the constant model. `stepwiselm` performs forward selection and `x4`, `x1`, and `x2`, respectively, as the corresponding *P*-values are less than the `PEnter` value of 0.06. `stepwiselm` later uses backward elimination and eliminates



x4 from the model because, once x2 is in the model, the  $P$ -value of x4 is higher than the default value of  $P_{\text{Remove}}$ , 0.1.

### Stepwise Regression Using Specified Model Formula and Variables

Perform stepwise regression using variables stored in a dataset array. Specify the starting model using Wilkinson notation, and identify the response and predictor variables using optional arguments.

Load the sample data.

```
load hospital
```

The hospital dataset array includes the gender, age, weight, and smoking status of patients.

Fit a linear model with a starting model of a constant term and Smoker as the predictor variable. Specify the response variable, Weight, and categorical predictor variables, Sex, Age, and Smoker.

```
mdl = stepwiselm(hospital, 'Weight~1+Smoker', ...
    'ResponseVar', 'Weight', 'PredictorVars', {'Sex', 'Age', 'Smoker'}, ...
    'CategoricalVar', {'Sex', 'Smoker'})
```

1. Adding Sex, FStat = 770.0158, pValue = 6.262758e-48
2. Removing Smoker, FStat = 0.21224, pValue = 0.64605

```
mdl =
```

```
Linear regression model:
```

```
Weight ~ 1 + Sex
```

```
Estimated Coefficients:
```

	Estimate	SE	tStat	pValue
	-----	-----	-----	-----
(Intercept)	130.47	1.1995	108.77	5.2762e-104
Sex_Male	50.06	1.7496	28.612	2.2464e-49

```
Number of observations: 100, Error degrees of freedom: 98
```

```
Root Mean Squared Error: 8.73
R-squared: 0.893, Adjusted R-Squared 0.892
F-statistic vs. constant model: 819, p-value = 2.25e-49
```

At each step, `stepwiselm` searches for terms to add and remove. At first step, stepwise algorithm adds `Sex` to the model with a  $P$ -value of  $6.26e-48$ . Then, removes `Smoker` from the model, since given `Sex` in the model, the variable `Smoker` becomes redundant. `stepwiselm` only includes `Sex` in the final linear model. The weight of the patients do not seem to differ significantly according to age or the status of smoking.

- “Compare large and small stepwise models” on page 11-127
- “Linear Regression” on page 11-11

## Input Arguments

### **tbl** — Input data

table | dataset array

Input data, specified as a table or dataset array. When `modelspec` is a formula, it specifies the variables to be used as the predictors and response. Otherwise, if you do not specify the predictor and response variables, the last variable is the response variable and the others are the predictor variables by default.

Predictor variables can be numeric, or any grouping variable type, such as logical or categorical (see “Grouping Variables” on page 2-59). The response must be numeric or logical.

To set a different column as the response variable, use the `ResponseVar` name-value pair argument. To use a subset of the columns as predictors, use the `PredictorVars` name-value pair argument.

### **x** — Predictor variables

matrix

Predictor variables, specified as an  $n$ -by- $p$  matrix, where  $n$  is the number of observations and  $p$  is the number of predictor variables. Each column of `x` represents one variable, and each row represents one observation.

By default, there is a constant term in the model, unless you explicitly remove it, so do not include a column of 1s in `x`.

Data Types: `single` | `double` | `logical`

### **y** — Response variable

vector

Response variable, specified as an  $n$ -by-1 vector, where  $n$  is the number of observations. Each entry in  $y$  is the response for the corresponding row of  $X$ .

Data Types: `single` | `double` | `logical`

### **modelspec** — Starting model

character vector specifying the model |  $t$ -by- $(p+1)$  terms matrix | character vector of the form 'Y ~ terms'

Starting model for the stepwise regression, specified as one of the following:

- Character vector specifying the type of starting model.

Character Vector	Starting Model Type
'constant'	Model contains only a constant (intercept) term.
'linear'	Model contains an intercept and linear terms for each predictor.
'interactions'	Model contains an intercept, linear terms for each predictor, and all products of pairs of distinct predictors (no squared terms).
'purequadratic'	Model contains an intercept, linear terms, and squared terms for each predictor.
'quadratic'	Model contains an intercept, linear terms, interactions, and squared terms for each predictor.
'polyijk'	Model is a polynomial with all terms up to degree $i$ in the first predictor, degree $j$ in the second predictor, etc. Use numerals 0 through 9. For example, 'poly2111' has a constant plus all linear* and product terms, and also contains terms with predictor 1 squared.

If you want to specify the smallest or largest set of terms in the model that `stepwiselm` fits, use the `Lower` and `Upper` name-value pair arguments.

- $t$ -by- $(p+1)$  matrix, namely a terms matrix on page 32-6285, specifying terms to include in model, where  $t$  is the number of terms and  $p$  is the number of predictor variables, and plus one is for the response variable.

- Character vector representing a formula on page 32-6288 in the form

```
'Y ~ terms',
```

where the `terms` are in “Wilkinson Notation” on page 32-6289.

---

**Note** All the terms `stepwiselm` returns are linear in the coefficients.

---

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'Criterion', 'aic', 'Upper', interactions, 'Verbose', 1` instructs `stepwiselm` to use the Akaike information criterion, display the action it takes at each step, and include at most the interaction terms in the model.

### **CategoricalVars** — Categorical variable list

cell array of character vectors | logical or numeric index vector

Categorical variable list, specified as the comma-separated pair consisting of `'CategoricalVars'` and either a cell array of character vectors containing categorical variable names in the table or dataset array `tbl`, or a logical or numeric index vector indicating which columns are categorical.

- If data is in a table or dataset array `tbl`, then, by default, `stepwiselm` treats all categorical values, logical values, character arrays, and cell arrays of character vectors as categorical variables.
- If data is in matrix `X`, then the default value of `'CategoricalVars'` is an empty matrix `[]`. That is, no variable is categorical unless you specify it as categorical.

For example, you can specify the observations 2 and 3 out of 6 as categorical using either of the following examples.

Example: `'CategoricalVars', [2,3]`

Example: `'CategoricalVars', logical([0 1 1 0 0 0])`

Data Types: `single` | `double` | `logical` | `cell`

### **Criterion** — Criterion to add or remove terms

'sse' (default) | 'aic' | 'bic' | 'rsquared' | 'adjrsquared'

Criterion to add or remove terms, specified as the comma-separated pair consisting of 'Criterion' and one of the following:

- 'sse' — Default for `stepwiselm`.  $p$ -value for an  $F$ -test of the change in the sum of squared error by adding or removing the term.
- 'aic' — Change in the value of Akaike information criterion (AIC).
- 'bic' — Change in the value of Bayesian information criterion (BIC).
- 'rsquared' — Increase in the value of  $R^2$ .
- 'adjrsquared' — Increase in the value of adjusted  $R^2$ .

Example: 'Criterion', 'bic'

### **Exclude** — Observations to exclude

logical or numeric index vector

Observations to exclude from the fit, specified as the comma-separated pair consisting of 'Exclude' and a logical or numeric index vector indicating which observations to exclude from the fit.

For example, you can exclude observations 2 and 3 out of 6 using either of the following examples.

Example: 'Exclude', [2,3]

Example: 'Exclude', logical([0 1 1 0 0 0])

Data Types: `single` | `double` | `logical`

### **Intercept** — Indicator for constant term

true (default) | false

Indicator for the constant term (intercept) in the fit, specified as the comma-separated pair consisting of 'Intercept' and either `true` to include or `false` to remove the constant term from the model.

Use 'Intercept' only when specifying the model using a character vector, not a formula or matrix.

Example: `'Intercept', false`

**Lower** — Model specification describing terms that cannot be removed from model

`'constant'` (default)

Model specification describing terms that cannot be removed from the model, specified as the comma-separated pair consisting of `'Lower'` and one of the options for `modelspec` naming the model.

Example: `'Lower', 'linear'`

**NSteps** — Number of steps to take

no limit (default) | positive integer

Number of steps to take, specified as the comma-separated pair consisting of `'NSteps'` and a positive integer.

Data Types: `single` | `double`

**PEnter** — Improvement measure for adding term

scalar value

Improvement measure for adding a term, specified as the comma-separated pair consisting of `'PEnter'` and a scalar value. The default values are below.

Criterion	Default value	Decision
<code>'Deviance'</code>	0.05	If the $p$ -value of $F$ or chi-squared statistic is smaller than <code>PEnter</code> , add the term to the model.
<code>'SSE'</code>	0.05	If the SSE of the model is smaller than <code>PEnter</code> , add the term to the model.
<code>'AIC'</code>	0	If the change in the AIC of the model is smaller than <code>PEnter</code> , add the term to the model.

Criterion	Default value	Decision
'BIC'	0	If the change in the BIC of the model is smaller than <code>PEnter</code> , add the term to the model.
'Rsquared'	0.1	If the increase in the R-squared of the model is larger than <code>PEnter</code> , add the term to the model.
'AdjRsquared'	0	If the increase in the adjusted R-squared of the model is larger than <code>PEnter</code> , add the term to the model.

For more information on the criteria, see `Criterion` name-value pair argument.

Example: `'PEnter',0.075`

### **PredictorVars** — Predictor variables

cell array of character vectors | logical or numeric index vector

Predictor variables to use in the fit, specified as the comma-separated pair consisting of `'PredictorVars'` and either a cell array of character vectors of the variable names in the table or dataset array `tbl`, or a logical or numeric index vector indicating which columns are predictor variables.

The character vectors should be among the names in `tbl`, or the names you specify using the `'VarNames'` name-value pair argument.

The default is all variables in `X`, or all variables in `tbl` except for `ResponseVar`.

For example, you can specify the second and third variables as the predictor variables using either of the following examples.

Example: `'PredictorVars',[2,3]`

Example: `'PredictorVars',logical([0 1 1 0 0 0])`

Data Types: single | double | logical | cell

**PRemove** — Improvement measure for removing term

scalar value

Improvement measure for removing a term, specified as the comma-separated pair consisting of 'PRemove' and a scalar value.

Criterion	Default value	Decision
'Deviance'	0.10	If the $p$ -value of $F$ or chi-squared statistic is larger than PRemove, remove the term from the model.
'SSE'	0.10	If the $p$ -value of the $F$ statistic is larger than PRemove, remove the term from the model.
'AIC'	0.01	If the change in the AIC of the model is larger than PRemove, remove the term from the model.
'BIC'	0.01	If the change in the BIC of the model is larger than PRemove, remove the term from the model.
'Rsquared'	0.05	If the increase in the R-squared value of the model is smaller than PRemove, remove the term from the model.
'AdjRsquared'	-0.05	If the increase in the adjusted R-squared value of the model is smaller than PRemove, remove the term from the model.

At each step, stepwise algorithm also checks whether any term is redundant (linearly dependent) with other terms in the current model. When any term is linearly dependent with other terms in the current model, it is removed, regardless of the criterion value.

For more information on the criteria, see Criterion name-value pair argument.



Example: `'PRemove',0.05`

### **ResponseVar** — Response variable

last column in `tbl` (default) | character vector containing variable name | logical or numeric index vector

Response variable to use in the fit, specified as the comma-separated pair consisting of `'ResponseVar'` and either a character vector containing the variable name in the table or dataset array `tbl`, or a logical or numeric index vector indicating which column is the response variable. You typically need to use `'ResponseVar'` when fitting a table or dataset array `tbl`.

For example, you can specify the fourth variable, say `yield`, as the response out of six variables, in one of the following ways.

Example: `'ResponseVar','yield'`

Example: `'ResponseVar',[4]`

Example: `'ResponseVar',logical([0 0 0 1 0 0])`

Data Types: `single` | `double` | `logical` | `char`

### **Upper** — Model specification describing largest set of terms in fit

`'interactions'` (default) | character vector

Model specification describing the largest set of terms in the fit, specified as the comma-separated pair consisting of `'Upper'` and one of the character vector options for `modelspec` naming the model.

Example: `'Upper','quadratic'`

### **VarNames** — Names of variables in fit

`{'x1','x2',...,'xn','y'}` (default) | cell array of character vectors

Names of variables in fit, specified as the comma-separated pair consisting of `'VarNames'` and a cell array of character vectors including the names for the columns of `X` first, and the name for the response variable `y` last.

`'VarNames'` is not applicable to variables in a table or dataset array, because those variables already have names.

For example, if in your data, horsepower, acceleration, and model year of the cars are the predictor variables, and miles per gallon (MPG) is the response variable, then you can name the variables as follows.

Example: `'VarNames', {'Horsepower', 'Acceleration', 'Model_Year', 'MPG'}`

Data Types: `cell`

### **Verbose** — Control for display of information

1 (default) | 0 | 2

Control for display of information, specified as the comma-separated pair consisting of `'Verbose'` and one of the following:

- 0 — Suppress all display.
- 1 — Display the action taken at each step.
- 2 — Also display the actions evaluated at each step.

Example: `'Verbose', 2`

### **Weights** — Observation weights

`ones(n,1)` (default) |  $n$ -by-1 vector of nonnegative scalar values

Observation weights, specified as the comma-separated pair consisting of `'Weights'` and an  $n$ -by-1 vector of nonnegative scalar values, where  $n$  is the number of observations.

Data Types: `single` | `double`

## Output Arguments

### **mdl** — Linear model

`LinearModel` object

Linear model representing a least-squares fit of the response to the data, returned as a `LinearModel` object.

For the properties and methods of the linear model object, `mdl`, see the `LinearModel` class page.

## Definitions

### Terms Matrix

A terms matrix is a  $t$ -by- $(p + 1)$  matrix specifying terms in a model, where  $t$  is the number of terms,  $p$  is the number of predictor variables, and plus one is for the response variable.

The value of  $T(i, j)$  is the exponent of variable  $j$  in term  $i$ . Suppose there are three predictor variables A, B, and C:

```
[0 0 0 0] % Constant term or intercept
[0 1 0 0] % B; equivalently, A^0 * B^1 * C^0
[1 0 1 0] % A*C
[2 0 0 0] % A^2
[0 1 2 0] % B*(C^2)
```

The 0 at the end of each term represents the response variable. In general,

- If you have the variables in a table or dataset array, then 0 must represent the response variable depending on the position of the response variable. The following example illustrates this.

Load the sample data and define the dataset array.

```
load hospital
dsa = dataset(hospital.Sex,hospital.BloodPressure(:,1),hospital.Age,...
hospital.Smoker, 'VarNames', {'Sex', 'BloodPressure', 'Age', 'Smoker'});
```

Represent the linear model 'BloodPressure ~ 1 + Sex + Age + Smoker' in a terms matrix. The response variable is in the second column of the dataset array, so there must be a column of 0s for the response variable in the second column of the terms matrix.

```
T = [0 0 0 0;1 0 0 0;0 0 1 0;0 0 0 1]
```

```
T =
```

```

0     0     0     0
1     0     0     0
0     0     1     0
0     0     0     1
```

Redefine the dataset array.

```
dsa = dataset(hospital.BloodPressure(:,1),hospital.Sex,hospital.Age,...  
hospital.Smoker,'VarNames',{'BloodPressure','Sex','Age','Smoker'});
```

Now, the response variable is the first term in the dataset array. Specify the same linear model, 'BloodPressure ~ 1 + Sex + Age + Smoker', using a terms matrix.

```
T = [0 0 0 0;0 1 0 0;0 0 1 0;0 0 0 1]
```

```
T =
```

```
    0    0    0    0  
    0    1    0    0  
    0    0    1    0  
    0    0    0    1
```

- If you have the predictor and response variables in a matrix and column vector, then you must include 0 for the response variable at the end of each term. The following example illustrates this.

Load the sample data and define the matrix of predictors.

```
load carsmall  
X = [Acceleration,Weight];
```

Specify the model 'MPG ~ Acceleration + Weight + Acceleration:Weight + Weight^2' using a term matrix and fit the model to the data. This model includes the main effect and two-way interaction terms for the variables, Acceleration and Weight, and a second-order term for the variable, Weight.

```
T = [0 0 0;1 0 0;0 1 0;1 1 0;0 2 0]
```

```
T =
```

```
    0    0    0  
    1    0    0  
    0    1    0  
    1    1    0  
    0    2    0
```

Fit a linear model.

```
mdl = fitlm(X,MPG,T)
```

```
mdl =
```

```
Linear regression model:
```

```
y ~ 1 + x1*x2 + x2^2
```

```
Estimated Coefficients:
```

	Estimate	SE	tStat	pValue
(Intercept)	48.906	12.589	3.8847	0.00019665
x1	0.54418	0.57125	0.95261	0.34337
x2	-0.012781	0.0060312	-2.1192	0.036857
x1:x2	-0.00010892	0.00017925	-0.6076	0.545
x2^2	9.7518e-07	7.5389e-07	1.2935	0.19917

```
Number of observations: 94, Error degrees of freedom: 89
```

```
Root Mean Squared Error: 4.1
```

```
R-squared: 0.751, Adjusted R-Squared 0.739
```

```
F-statistic vs. constant model: 67, p-value = 4.99e-26
```

Only the intercept and x2 term, which correspond to the Weight variable, are significant at the 5% significance level.

Now, perform a stepwise regression with a constant model as the starting model and a linear model with interactions as the upper model.

```
T = [0 0 0;1 0 0;0 1 0;1 1 0];
```

```
mdl = stepwiselm(X,MPG,[0 0 0], 'upper', T)
```

```
1. Adding x2, FStat = 259.3087, pValue = 1.643351e-28
```

```
mdl =
```

```
Linear regression model:
```

```
y ~ 1 + x2
```

```
Estimated Coefficients:
```

	Estimate	SE	tStat	pValue
(Intercept)	49.238	1.6411	30.002	2.7015e-49
x2	-0.0086119	0.0005348	-16.103	1.6434e-28

```
Number of observations: 94, Error degrees of freedom: 92
```

```
Root Mean Squared Error: 4.13
```

```
R-squared: 0.738, Adjusted R-Squared 0.735
```

```
F-statistic vs. constant model: 259, p-value = 1.64e-28
```

The results of the stepwise regression are consistent with the results of `fitlm` in the previous step.

## Formula

A formula for model specification is a character vector of the form ' $Y \sim terms$ '

where

- $Y$  is the response name.
- $terms$  contains
  - Variable names
  - $+$  means include the next variable
  - $-$  means do not include the next variable
  - $:$  defines an interaction, a product of terms
  - $*$  defines an interaction **and all lower-order terms**
  - $^$  raises the predictor to a power, exactly as in  $*$  repeated, so  $^$  includes lower order terms as well
  - $()$  groups terms

---

**Note** Formulas include a constant (intercept) term by default. To exclude a constant term from the model, include  $-1$  in the formula.

---

For example,

' $Y \sim A + B + C$ ' means a three-variable linear model with intercept.

' $Y \sim A + B + C - 1$ ' is a three-variable linear model without intercept.

' $Y \sim A + B + C + B^2$ ' is a three-variable model with intercept and a  $B^2$  term.

' $Y \sim A + B^2 + C$ ' is the same as the previous example because  $B^2$  includes a  $B$  term.

' $Y \sim A + B + C + A:B$ ' includes an  $A*B$  term.

' $Y \sim A*B + C$ ' is the same as the previous example because  $A*B = A + B + A:B$ .

' $Y \sim A*B*C - A:B:C$ ' has all interactions among  $A$ ,  $B$ , and  $C$ , except the three-way interaction.

' $Y \sim A*(B + C + D)$ ' has all linear terms, plus products of A with each of the other variables.

## Wilkinson Notation

Wilkinson notation describes the factors present in models. The notation relates to factors present in models, not to the multipliers (coefficients) of those factors.

Wilkinson Notation	Factors in Standard Notation
1	Constant (intercept) term
$A^k$ , where k is a positive integer	$A, A^2, \dots, A^k$
$A + B$	$A, B$
$A*B$	$A, B, A*B$
$A:B$	$A*B$ only
$-B$	Do not include B
$A*B + C$	$A, B, C, A*B$
$A + B + C + A:B$	$A, B, C, A*B$
$A*B*C - A:B:C$	$A, B, C, A*B, A*C, B*C$
$A*(B + C)$	$A, B, C, A*B, A*C$

Statistics and Machine Learning Toolbox notation always includes a constant term unless you explicitly remove the term using `-1`.

## Tips

- You cannot use robust regression with stepwise regression. Check your data for outliers before using `stepwiselm`.
- For other methods such as `anova`, or properties of the `LinearModel` object, see `LinearModel`.
- After training a `LinearModel` model object by using `stepwiselm`, use the function `predict` and the trained model object to generate C code that predicts responses for new data. For details, see “Code Generation” on page 32-0 .

## Algorithms

Stepwise regression is a systematic method for adding and removing terms from a linear or generalized linear model based on their statistical significance in explaining the response variable. The method begins with an initial model, specified using `modelspec`, and then compares the explanatory power of incrementally larger and smaller models.

MATLAB uses forward and backward stepwise regression to determine a final model. At each step, the method searches for terms to add to or remove from the model based on the value of the 'Criterion' argument. The default value of 'Criterion' is 'sse', and in this case, `stepwiselm` uses the  $p$ -value of an  $F$ -statistic to test models with and without a potential term at each step. If a term is not currently in the model, the null hypothesis is that the term would have a zero coefficient if added to the model. If there is sufficient evidence to reject the null hypothesis, the term is added to the model. Conversely, if a term is currently in the model, the null hypothesis is that the term has a zero coefficient. If there is insufficient evidence to reject the null hypothesis, the term is removed from the model.

Here is how stepwise proceeds when 'Criterion' is 'sse':

- 1 Fit the initial model.
- 2 Examine a set of available terms not in the model. If any of these terms have  $p$ -values less than an entrance tolerance (that is, if it is unlikely that they would have zero coefficient if added to the model), add the one with the smallest  $p$ -value and repeat this step; otherwise, go to step 3.
- 3 If any of the available terms in the model have  $p$ -values greater than an exit tolerance (that is, the hypothesis of a zero coefficient cannot be rejected), remove the one with the largest  $p$ -value and go to step 2; otherwise, end.

At any stage, the function will not add a higher-order term if the model does not also include all lower-order terms that are subsets of it. For example, it will not try to add the term  $X1 : X2^2$  unless both  $X1$  and  $X2^2$  are already in the model. Similarly, the function will not remove lower-order terms that are subsets of higher-order terms that remain in the model. For example, it will not examine to remove  $X1$  or  $X2^2$  if  $X1 : X2^2$  stays in the model.

The default for `stepwiseglm` is 'Deviance' and it follows a similar procedure for adding or removing terms.



There are several other criteria available, which you can specify using the 'Criterion' argument. You can use the change in the value of the Akaike information criterion, Bayesian information criterion, R-squared, adjusted R-squared as a criterion to add or remove terms.

Depending on the terms included in the initial model and the order in which terms are moved in and out, the method might build different models from the same set of potential terms. The method terminates when no single step improves the model. There is no guarantee, however, that a different initial model or a different sequence of steps will not lead to a better fit. In this sense, stepwise models are locally optimal, but might not be globally optimal.

## Alternative Functionality

You can construct a model using `fitlm`, and then manually adjust the model using `step`, `addTerms`, or `removeTerms`.

## See Also

`LinearModel` | `fitlm` | `step`

## Topics

“Compare large and small stepwise models” on page 11-127

“Linear Regression” on page 11-11

“Stepwise Regression” on page 11-127

**Introduced in R2013b**

## stepwisefit

Stepwise regression

### Syntax

```
b = stepwisefit(X,y)
[b,se,pval,inmodel,stats,nextstep,history] = stepwisefit(...)
[...] = stepwisefit(X,y,param1, val1,param2, val2,...)
```

### Description

`b = stepwisefit(X,y)` uses a stepwise method to perform a multilinear regression of the response values in the  $n$ -by-1 vector  $y$  on the  $p$  predictive terms in the  $n$ -by- $p$  matrix  $X$ . Distinct predictive terms should appear in different columns of  $X$ .

$b$  is a  $p$ -by-1 vector of estimated coefficients for all of the terms in  $X$ . The `stepwisefit` function calculates the coefficient estimate values in  $b$  as follows:

- If a term is not in the final model, then the corresponding coefficient estimate in  $b$  results from adding only that term to the predictors in the final model.
- If a term is in the final model, then the coefficient estimate in  $b$  for that term is a result of the final model, that is `stepwise` does not consider the terms it excluded from the model while computing these values.

---

**Note** `stepwisefit` automatically includes a constant term in all models. Do not enter a column of 1s directly into  $X$ .

---

`stepwisefit` treats NaN values in either  $X$  or  $y$  as missing values, and ignores them.

`[b,se,pval,inmodel,stats,nextstep,history] = stepwisefit(...)` returns the following additional information:

- `se` — A vector of standard errors for  $b$

- `pval` — A vector of  $p$ -values for testing whether elements of `b` are 0
- `inmodel` — A logical vector, with length equal to the number of columns in `X`, specifying which terms are in the final model
- `stats` — A structure of additional statistics with the following fields. All statistics pertain to the final model except where noted.
  - `source` — The character vector `'stepwisefit'`
  - `dfe` — Degrees of freedom for error
  - `df0` — Degrees of freedom for the regression
  - `SStotal` — Total sum of squares of the response
  - `SSresid` — Sum of squares of the residuals
  - `fstat` —  $F$ -statistic for testing the final model vs. no model (mean only)
  - `pval` —  $p$  value of the  $F$ -statistic
  - `rmse` — Root mean square error
  - `xr` — Residuals for predictors not in the final model, after removing the part of them explained by predictors in the model
  - `yr` — Residuals for the response using predictors in the final model
  - `B` — Coefficients for terms in final model, with values for a term not in the model set to the value that would be obtained by adding that term to the model
  - `SE` — Standard errors for coefficient estimates
  - `TSTAT` —  $t$  statistics for coefficient estimates
  - `PVAL` —  $p$ -values for coefficient estimates
  - `intercept` — Estimated intercept
  - `wasnan` — Indicates which rows in the data contained NaN values
- `nextstep` — The recommended next step—either the index of the next term to move in or out of the model, or 0 if no further steps are recommended
- `history` — Structure containing information on steps taken, with the following fields:
  - `B` — Matrix of regression coefficients, where each column is one step, and each row is one coefficient.
  - `rmse` — Root mean square errors for the model at each step.

- `df0` — Degrees of freedom for the regression at each step.
- `in` — Logical array indicating which predictors are in the model at each step, where each row is one step, and each column is one predictor.

`[...] = stepwisefit(X, y, param1, val1, param2, val2, ...)` specifies one or more of the name/value pairs described in the following table.

Parameter	Value
'inmodel'	A logical vector specifying terms to include in the initial fit. The default is to specify no terms.
'penter'	The maximum $p$ value for a term to be added. The default is 0.05.
'premove'	The minimum $p$ value for a term to be removed. The default is the maximum of the value of 'penter' and 0.10.
'display'	'on' displays information about each step in the command window. This is the default.  'off' omits the display.
'maxiter'	The maximum number of steps in the regression. The default is Inf.
'keep'	A logical vector specifying terms to keep in their initial state. The default is to specify no terms.
'scale'	'on' centers and scales each column of $X$ (computes $z$ -scores) before fitting.  'off' does not scale the terms. This is the default.

## Examples

Load the data in `hald.mat`, which contains observations of the heat of reaction of various cement mixtures:

```
load hald
whos
  Name           Size      Bytes   Class   Attributes
  Description    22x58     2552    char
  hald           13x5       520     double
  heat           13x1       104     double
  ingredients    13x4       416     double
```

The response (heat) depends on the quantities of the four predictors (the columns of ingredients).

Use `stepwisefit` to carry out the stepwise regression algorithm, beginning with no terms in the model and using entrance/exit tolerances of 0.05/0.10 on the  $p$ -values:

```
stepwisefit(ingredients,heat,...
            'penter',0.05,'premove',0.10);
Initial columns included: none
Step 1, added column 4, p=0.000576232
Step 2, added column 1, p=1.10528e-006
Final columns included: 1 4
   'Coeff'      'Std.Err.'    'Status'    'P'
   [ 1.4400]    [ 0.1384]    'In'       [1.1053e-006]
   [ 0.4161]    [ 0.1856]    'Out'      [ 0.0517]
   [-0.4100]    [ 0.1992]    'Out'      [ 0.0697]
   [-0.6140]    [ 0.0486]    'In'       [1.8149e-007]
```

`stepwisefit` automatically includes an intercept term in the model, so you do not add it explicitly to `ingredients` as you would for `regress`. For terms not in the model, coefficient estimates and their standard errors are those that result by adding the corresponding term to the final model.

The `inmodel` parameter is used to specify terms in an initial model:

```
initialModel = ...
            [false true false false]; % Force in 2nd term
stepwisefit(ingredients,heat,...
            'inmodel',initialModel,...
            'penter',.05,'premove',0.10);
Initial columns included: 2
Step 1, added column 1, p=2.69221e-007
Final columns included: 1 2
   'Coeff'      'Std.Err.'    'Status'    'P'
   [ 1.4683]    [ 0.1213]    'In'       [2.6922e-007]
   [ 0.6623]    [ 0.0459]    'In'       [5.0290e-008]
   [ 0.2500]    [ 0.1847]    'Out'      [ 0.2089]
   [-0.2365]    [ 0.1733]    'Out'      [ 0.2054]
```

The preceding two models, built from different initial models, use different subsets of the predictive terms. Terms 2 and 4, swapped in the two models, are highly correlated:

```
term2 = ingredients(:,2);
term4 = ingredients(:,4);
```

```
R = corrcoef(term2,term4)
R =
    1.0000    -0.9730
   -0.9730    1.0000
```

To compare the models, use the stats output of stepwisefit:

```
[betahat1,se1,pval1,inmodel1,stats1] = ...
    stepwisefit(ingredients,heat,...
    'penter',.05,'premove',0.10,...
    'display','off');
[betahat2,se2,pval2,inmodel2,stats2] = ...
    stepwisefit(ingredients,heat,...
    'inmodel',initialModel,...
    'penter',.05,'premove',0.10,...
    'display','off');

RMSE1 = stats1.rmse
RMSE1 =
    2.7343
RMSE2 = stats2.rmse
RMSE2 =
    2.4063
```

The second model has a lower Root Mean Square Error (RMSE).

## Algorithms

Stepwise regression is a systematic method for adding and removing terms from a multilinear model based on their statistical significance in a regression. The method begins with an initial model and then compares the explanatory power of incrementally larger and smaller models. At each step, the  $p$  value of an  $F$ -statistic is computed to test models with and without a potential term. If a term is not currently in the model, the null hypothesis is that the term would have a zero coefficient if added to the model. If there is sufficient evidence to reject the null hypothesis, the term is added to the model. Conversely, if a term is currently in the model, the null hypothesis is that the term has a zero coefficient. If there is insufficient evidence to reject the null hypothesis, the term is removed from the model. The method proceeds as follows:

- 1 Fit the initial model.
- 2 If any terms not in the model have  $p$ -values less than an entrance tolerance (that is, if it is unlikely that they would have zero coefficient if added to the model), add the one with the smallest  $p$  value and repeat this step; otherwise, go to step 3.

- 3 If any terms in the model have  $p$ -values greater than an exit tolerance (that is, if it is unlikely that the hypothesis of a zero coefficient can be rejected), remove the one with the largest  $p$  value and go to step 2; otherwise, end.

Depending on the terms included in the initial model and the order in which terms are moved in and out, the method may build different models from the same set of potential terms. The method terminates when no single step improves the model. There is no guarantee, however, that a different initial model or a different sequence of steps will not lead to a better fit. In this sense, stepwise models are locally optimal, but may not be globally optimal.

## References

- [1] Draper, N. R., and H. Smith. *Applied Regression Analysis*. Hoboken, NJ: Wiley-Interscience, 1998. pp. 307–312.

## See Also

`addedvarplot` | `regress` | `stepwise`

Introduced before R2006a

## subsasgn

**Class:** `classregtree`

Subscripted reference for `classregtree` object

---

**Note** `classregtree` will be removed in a future release. See `fitctree`, `fitrtree`, `ClassificationTree`, or `RegressionTree` instead.

---

## Syntax

## Description

Subscript assignment is not allowed for a `classregtree` object.

## See Also

`classregtree`



# subsasgn

**Class:** dataset

Subscripted assignment to dataset array

---

**Note** The `dataset` data type might be removed in a future release. To work with heterogeneous data, use the MATLAB® `table` data type instead. See MATLAB `table` documentation for more information.

---

## Description

`A = subsasgn(A,S,B)` is called for the syntax `A(i,j)=B`, `A{i,j}=B`, or `A.var=B` when `A` is a dataset array. `S` is a structure array with the fields:

<code>type</code>	'()', '{}', or '.' specifying the subscript type.
<code>subs</code>	Cell array or character vector containing the actual subscripts.

`A(i,j) = B` assigns the contents of the dataset array `B` to a subset of the observations and variables in the dataset array `A`. `i` and `j` are one of the following types:

- positive integers
- vectors of positive integers
- observation/variable names
- cell arrays containing one or more observation/variable names
- logical vectors

The assignment does not use observation names, variable names, or any other properties of `B` to modify properties of `A`; however properties of `A` are extended with default values if the assignment expands the number of observations or variables in `A`. Elements of `B` are assigned into `A` by position, not by matching names.

`A{i,j} = B` assigns the value `B` into an element of the dataset array `A`. `i` and `J` are positive integers, or logical vectors. Cell indexing cannot assign into multiple dataset

elements, that is, the subscripts  $i$  and  $j$  must each refer to only a single observation or variable.  $B$  is cast to the type of the target variable if necessary. If the dataset element already exists,  $A\{i, j\}$  may also be followed by further subscripting as supported by the variable.

For dataset variables that are cell arrays, assignments such as  $A\{1, 'CellVar'\} = B$  assign into the contents of the target dataset element in the same way that  $\{\}$ -indexing of an ordinary cell array does.

For dataset variables that are n-D arrays, i.e., each observation is a matrix or array, an assignment such as  $A\{1, 'ArrayVar'\} = B$  assigns into the second and following dimensions of the target dataset element, i.e., the assignment adds a leading singleton dimension to  $B$  to account for the observation dimension of the dataset variable.

$A.var = B$  or  $A.(varname) = B$  assigns  $B$  to a dataset variable.  $var$  is a variable name literal, or  $varname$  is a character variable containing a variable name. If the dataset variable already exists, the assignment completely replaces that variable. To assign into an element of the variable,  $A.var$  or  $A.(varname)$  may be followed by further subscripting as supported by the variable. In particular,  $A.var(obsnames, \dots) = B$  and  $A.var\{obsnames, \dots\} = B$  (when supported by `var`) provide assignment into a dataset variable using observation names.

$A.properties.propertyname = P$  assigns to a dataset property.  $propertyname$  is one of the following:

- 'ObsNames'
- 'VarNames'
- 'Description'
- 'Units'
- 'DimNames'
- 'UserData'
- 'VarDescription'

To assign into an element of the property,  $A.properties.propertyname$  may also be followed by further subscripting as supported by the property.

You cannot assign multiple values into dataset variables or properties using assignments such as  $[A.CellVar\{1:2\}] = B$ ,  $[A.StructVar(1:2).field] = B$ , or

`[A.Properties.ObsNames{1:2}] = B`. Use multiple assignments of the form `A.CellVar{1} = B` instead.

Similarly, if a dataset variable is a cell array with multiple columns or is an n-D cell array, then the contents of that variable for a single observation consists of multiple cells, and you cannot assign to all of them using the syntax `A{1, 'CellVar'} = B`. Use multiple assignments of the form `[A.CellVar{1,1}] = B` instead.

## See Also

`dataset` | `set` | `subsref`

## subsasgn

**Class:** `gmdistribution`

Subscripted reference for Gaussian mixture distribution object

## Description

Subscript assignment is not allowed for `gmdistribution` objects.

## See Also

`gmdistribution`

## subsasgn

**Class:** NaiveBayes

Subscripted reference for NaiveBayes object

## Description

Subscript assignment is not allowed for a NaiveBayes object.

## subsref

**Class:** `classregtree`

Subscripted reference for `classregtree` object

---

**Note** `classregtree` will be removed in a future release. See `fitctree`, `fitrtree`, `ClassificationTree`, or `RegressionTree` instead.

---

## Syntax

`B = subsref(T,S)`

## Description

`B = subsref(T,S)` is called for the syntax `T(X)` when `T` is a `classregtree` object. `S` is a structure array with the fields:

<code>type</code>	Character vector containing '()', '{}', or '.' specifying the subscript type.
<code>subs</code>	Cell array or character vector containing the actual subscripts.

`[...] = T(...)` invokes the `eval` method for the tree `T`.

## See Also

`classregtree` | `eval`

# subsref

**Class:** dataset

Subscripted reference for dataset array

---

**Note** The `dataset` data type might be removed in a future release. To work with heterogeneous data, use the MATLAB® `table` data type instead. See MATLAB `table` documentation for more information.

---

## Syntax

```
B = subsref(A,S)
```

## Description

`B = subsref(A,S)` is called for the syntax `A(i,j)`, `A{i,j}`, or `A.var` when `A` is a dataset array. `S` is a structure array with the fields:

<code>type</code>	Character vector containing '()', '{}', or '.' specifying the subscript type.
<code>subs</code>	Cell array or character vector containing the actual subscripts.

`B = A(i,j)` returns a dataset array that contains a subset of the observations and variables in the dataset array `A`. `i` and `j` are one of the following types:

- positive integers
- vectors of positive integers
- observation/variable names
- cell arrays containing one or more observation/variable names
- logical vectors

`B` contains the same property values as `A`, subsetted for observations or variables where appropriate.

`B = A{i, j}` returns an element of a dataset variable. `i` and `j` are positive integers, or logical vectors. Cell indexing cannot return multiple dataset elements, that is, the subscripts `i` and `j` must each refer to only a single observation or variable. `A{i, j}` may also be followed by further subscripting as supported by the variable.

For dataset variables that are cell arrays, expressions such as `A{1, 'CellVar'}` return the contents of the referenced dataset element in the same way that `{}`-indexing on an ordinary cell array does. If the dataset variable is a single column of cells, the contents of a single cell is returned. If the dataset variable has multiple columns or is n-D, multiple outputs containing the contents of multiple cells are returned.

For dataset variables that are n-D arrays, i.e., each observation is a matrix or an array, expressions such as `A{1, 'ArrayVar'}` return `A.ArrayVar(1, :, ...)` with the leading singleton dimension squeezed out.

`B = A.var` or `A.(varname)` returns a dataset variable. `var` is a variable name literal, or `varname` is a character variable containing a variable name. `A.var` or `A.(varname)` may also be followed by further subscripting as supported by the variable. In particular, `A.var(obsnames, ...)` and `A.var{obsnames, ...}` (when supported by `var`) provide subscripting into a dataset variable using observation names.

`P = A.Properties.propertyname` returns a dataset property. `propertyname` is one of the following:

- 'ObsNames'
- 'VarNames'
- 'Description'
- 'Units'
- 'DimNames'
- 'UserData'
- 'VarDescription'

`A.properties.propertyname` may also be followed by further subscripting as supported by the property.

## Limitations

Subscripting expressions such as `A.CellVar{1:2}`, `A.StructVar(1:2).field`, or `A.Properties.ObsNames{1:2}` are valid, but result in `subsref` returning multiple



outputs in the form of a comma-separated list. If you explicitly assign to output arguments on the left-hand side of an assignment, for example, `[cellval1, cellval2] = A.CellVar{1:2}`, those variables will receive the corresponding values. However, if there are no output arguments, only the first output in the comma-separated list is returned.

Similarly, if a dataset variable is a cell array with multiple columns or is an n-D cell array, then subscripting expressions such as `A{1, 'CellVar'}` result in `subsref` returning the contents of multiple cells. You should explicitly assign to output arguments on the left-hand side of an assignment, for example, `[cellval1, cellval2] = A{1, 'CellVar'}`.

## See Also

`dataset` | `set` | `subsasgn`

## subsref

**Class:** `gmdistribution`

Subscripted reference for Gaussian mixture distribution object

## Syntax

```
B = subsref(T,S)
```

## Description

`B = subsref(T,S)` is called for the syntax `T(X)` when `T` is a `gmdistribution` object. `S` is a structure array with the following fields:

<code>type</code>	Character vector containing '()', '{}', or '.' specifying the subscript type.
<code>subs</code>	Cell array or character vector containing the actual subscripts.

## See Also

`gmdistribution`

## subsref

**Class:** NaiveBayes

Subscripted reference for NaiveBayes object

## Syntax

```
b = subsref(nb,s)
```

## Description

`b = subsref(nb,s)` is called for the syntax `nb(s)` when `nb` is a NaiveBayes object. `S` is a structure array with the fields:

<code>type</code>	Character vector containing '()', '{}', or '!' specifying the subscript type.
<code>subs</code>	Cell array or character vector containing the actual subscripts.

## subsref

**Class:** grandset

Subscripted reference for grandset

## Syntax

```
x = p(i, j)
x = subsref(p, s)
```

## Description

`x = p(i, j)` returns a matrix that contains a subset of the points from the point set `p`. The indices in `i` select points from the set and the indices in `j` select columns from those points. `i` and `j` are vector of positive integers or logical vectors. A colon used as a subscript, as in `p(i, :)`, indicates the entire row (or column).

`x = subsref(p, s)` is called for the syntax `p(i)`, `p{i}`, or `p.i`. `s` is a structure array with the fields:

<code>type</code>	Character vector containing '()', '{}', or '.' specifying the subscript type.
<code>subs</code>	Cell array or character vector containing the actual subscripts.

## Examples

Command	Returns
<code>p = sobolset(5);</code>	The fifth point
<code>x = p(1:10, :)</code>	All columns of the first 10 points
<code>x = p(end, 1)</code>	The first column of the last point
<code>x = p([1, 4, 5], :)</code>	Points 1, 4, and 5

## See Also

qrandset

## summary

**Class:** dataset

Print summary of dataset array

---

**Note** The `dataset` data type might be removed in a future release. To work with heterogeneous data, use the MATLAB® `table` data type instead. See MATLAB `table` documentation for more information.

---

## Syntax

```
summary(A)  
s = summary(A)
```

## Description

`summary(A)` prints a summary of a dataset array and the variables that it contains.

`s = summary(A)` returns a scalar structure `s` that contains a summary of the dataset `A` and the variables that `A` contains. For more information on the fields in `s`, see [Outputs](#).

Summary information depends on the type of the variables in the data set:

- For numerical variables, `summary` computes a five-number summary of the data, giving the minimum, the first quartile, the median, the third quartile, and the maximum.
- For logical variables, `summary` counts the number of `true`s and `false`s in the data.
- For categorical variables, `summary` counts the number of data at each level.

## Output Arguments

The following list describes the fields in the structure `s`:

- `Description` — A character array containing the dataset description.
- `Variables` — A structure array with one element for each dataset variable in `A`. Each element has the following fields:
  - `Name` — A character vector containing the name of the variable.
  - `Description` — A character vector containing the variable's description.
  - `Units` — A character vector containing the variable's units.
  - `Size` — A numeric vector containing the size of the variable.
  - `Class` — A character vector containing the class of the variable.
  - `Data` — A scalar structure containing the following fields.

For numeric variables:

- `Probabilities` — A numeric vector containing the probabilities [0.0 .25 .50 .75 1.0] and NaN (if any are present in the corresponding dataset variable).
- `Quantiles` — A numeric vector containing the values that correspond to 'Probabilities' for the corresponding dataset variable, and a count of NaNs (if any are present).

For logical variables:

- `Values` — The logical vector [true false].
- `Counts` — A numeric vector of counts for each logical value.

For categorical variables:

- `Levels` — A cell array containing the labels for each level of the corresponding dataset variable.
- `Counts` — A numeric vector of counts for each level.

'Data' is empty if variable is not numeric, categorical, or logical. If a dataset variable has more than one column, then the corresponding 'Quantiles' or 'Counts' field is a matrix or an array.

## Examples

Summarize Fisher's iris data:

```
load fisheriris
species = nominal(species);
data = dataset(species,meas);
summary(data)
species: [150x1 nominal]
   setosa   versicolor   virginica
      50         50         50
meas: [150x4 double]
   min      4.3000      2      1      0.1000
   1st Q    5.1000    2.8000    1.6000    0.3000
   median  5.8000      3    4.3500    1.3000
   3rd Q   6.4000    3.3000    5.1000    1.8000
   max     7.9000    4.4000    6.9000    2.5000
```

Summarize the data in hospital.mat:

```
load hospital
summary(hospital)
```

Dataset array created from the data file hospital.dat.

The first column of the file ("id") is used for observation names. Other columns ("sex" and "smoke") have been converted from their original coded values into categorical and logical variables. Two sets of columns ("sys" and "dia", "trial1" through "trial4") have been combined into single variables with multivariate observations. Column headers have been replaced with more descriptive variable names. Units have been added where appropriate.

```
LastName: [100x1 cell array of character vectors]
```

```
Sex: [100x1 nominal]
```

```
   Female   Male
      53     47
```

```
Age: [100x1 double, Units = Yrs]
```

```
   min      1st Q      median      3rd Q      max
    25       32       39       44       50
```

```
Weight: [100x1 double, Units = Lbs]
```

```
   min      1st Q      median      3rd Q      max
   111    130.5000    142.5000    180.5000    202
```

```
Smoker: [100x1 logical]
```



---

true	false
34	66

BloodPressure: [100x2 double, Units = mm Hg]

Systolic/Diastolic

min	109	68
1st Q	117.5000	77.5000
median	122	81.5000
3rd Q	127.5000	89
max	138	99

Trials: [100x1 cell, Units = Counts]

From zero to four measurement trials performed

## See Also

get | grpstats | set

## Support property

**Class:** ProbDist

Read-only structure containing information about support of ProbDist object

## Description

`Support` is a read-only property of the `ProbDist` class. `Support` is a structure containing information about the support of a `ProbDist` object. It includes the following fields:

- `range`
- `closedbound`
- `iscontinuous`

## Values

The values for the three fields in the structure are:

- `range` — A two-element vector `[L, U]`, such that all of the probability is contained from `L` to `U`.
- `closedbound` — A two-element logical vector indicating whether the corresponding range endpoint is included. Possible values for each endpoint are 1 (`true`) or 0 (`false`).
- `iscontinuous` — A logical value indicates if the distribution takes values on the entire interval from `L` to `U` (`true`), or if it takes only integer values within this range (`false`). Possible values are 1 (`true`) or 0 (`false`).

Use this information to view and compare information about the support of distributions.

# struct2dataset

Convert structure array to dataset array

---

**Note** The `dataset` data type might be removed in a future release. To work with heterogeneous data, use the MATLAB® `table` data type instead. See MATLAB `table` documentation for more information.

---

## Syntax

```
ds = struct2dataset(S)
ds = struct2dataset(S,Name,Value)
```

## Description

`ds = struct2dataset(S)` converts a structure array to a dataset array.

`ds = struct2dataset(S,Name,Value)` performs the conversion using additional options specified by one or more `Name,Value` pair arguments.

## Examples

### Convert Scalar Structure Array to Dataset Array

Convert a scalar structure array to a dataset array using the default options.

Create a structure array to convert.

```
S.Name = {'CLARK';'BROWN';'MARTIN'};
S.Gender = {'M';'F';'M'};
S.SystolicBP = [124;122;130];
S.DiastolicBP = [93;80;92];
S
```

```
S =  
  
    struct with fields:  
  
        Name: {3x1 cell}  
        Gender: {3x1 cell}  
        SystolicBP: [3x1 double]  
        DiastolicBP: [3x1 double]
```

The scalar structure array has four fields, each with three rows.

Convert the structure array to a dataset array.

```
ds = struct2dataset(S)
```

```
ds =  
  
    Name           Gender   SystolicBP   DiastolicBP  
    'CLARK'        'M'         124          93  
    'BROWN'        'F'         122          80  
    'MARTIN'       'M'         130          92
```

The structure field names in `S` become the variable names in the output dataset array. The size of `ds` is 3-by-4.

### Convert Nonscalar Structure Array to Dataset Array

Convert a nonscalar structure array to a dataset array, using one of the structure fields for observation names.

Create a nonscalar structure array to convert.

```
S(1,1).Name = 'CLARK';  
S(1,1).Gender = 'M';  
S(1,1).SystolicBP = 124;  
S(1,1).DiastolicBP = 93;  
  
S(2,1).Name = 'BROWN';  
S(2,1).Gender = 'F';
```

```
S(2,1).SystolicBP = 122;
S(2,1).DiastolicBP = 80;
```

```
S(3,1).Name = 'MARTIN';
S(3,1).Gender = 'M';
S(3,1).SystolicBP = 130;
S(3,1).DiastolicBP = 92;
```

```
S
```

```
S =
```

```
3x1 struct array with fields:
```

```
    Name
    Gender
    SystolicBP
    DiastolicBP
```

This is a 3-by-1 structure array with 4 fields.

Convert the structure array to a dataset array, using the Name field for observation names.

```
ds = struct2dataset(S, 'ReadObsNames', 'Name')
```

```
ds =
```

	Gender	SystolicBP	DiastolicBP
CLARK	'M'	124	93
BROWN	'F'	122	80
MARTIN	'M'	130	92

The size of ds is 3-by-3 because the structure field Name is used for observation names, and not as a dataset array variable.

```
ds.Properties.DimNames
```

```
ans =
```

```
1x2 cell array

    {'Name'}    {'Variables'}

ds.Properties.ObsNames

ans =

3x1 cell array

    {'CLARK' }
    {'BROWN' }
    {'MARTIN' }
```

- “Create a Dataset Array from Workspace Variables” on page 2-70
- “Create a Dataset Array from a File” on page 2-77

## Input Arguments

### **s** — Input structure array

structure array

Input structure array to convert to a dataset array, specified as a scalar structure array with  $N$  fields, each with  $M$  rows, or a nonscalar  $M$ -by-1 structure array with  $N$  fields.

Data Types: `struct`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'ReadObsNames', 'myField'` specifies that the structure field, `myField`, contains observation names.

**ReadObsNames** — Name of structure field containing observation names for dataset array

`false` (default) | character vector

Name of structure field containing observation names for the output dataset array, specified as the comma-separated pair consisting of 'ReadObsNames' and a character vector containing a field name from the input structure array, *S*. When you specify a field name, `struct2dataset` uses that field to create observation names, and sets `ds.Properties.DimNames` equal to `{ReadObsNames, 'Variables'}`.

For example, to specify that observation names are in the structure field, `Names`, use

Example: 'ReadObsNames', 'Names'

By default, or if `ReadObsNames` is equal to `false`, `struct2dataset` does not create observation names unless you specify names using the name-value pair argument `ObsNames`.

#### **ObsNames** — Observation names for dataset array

cell array of character vectors

Observation names for the output dataset array, specified as the comma-separated pair consisting of 'ObsNames' and a cell array of character vectors containing observation names. The names do not need to be valid MATLAB identifiers, but they must be unique.

#### **AsScalar** — Indicator for how to treat scalar structure

false | true

Indicator for how to treat a scalar input structure array, specified as the comma-separated pair consisting of 'AsScalar' and either `true` or `false`. The default value is `true` if *S* is a scalar structure array, and `false` otherwise.

By default, `struct2dataset` converts a scalar structure array with *N* fields, each with *M* rows, into an *M*-by-*N* dataset array.

If instead you set `AsScalar` equal to `false` for a scalar input structure array, then `struct2dataset` converts *S* to a dataset array with *N* observations.

## Output Arguments

#### **ds** — Output dataset array

dataset array

Output dataset array, returned by default with *M* observations and *N* variables.

- If  $S$  is a scalar structure array with  $N$  fields, each with  $M$  rows, then `ds` is an  $M$ -by- $N$  dataset array.
- If  $S$  is a nonscalar  $M$ -by-1 structure array with  $N$  fields, then `ds` is an  $M$ -by- $N$  dataset array.
- If  $S$  is a scalar structure array with  $N$  fields, each with  $M$  rows, and `AsScalar` is set equal to `false`, then `ds` is a dataset array with  $N$  observations.

## See Also

`cell2dataset` | `dataset` | `dataset2struct`

## Topics

“Create a Dataset Array from Workspace Variables” on page 2-70

“Create a Dataset Array from a File” on page 2-77

“Dataset Arrays” on page 2-142

**Introduced in R2012b**



# surfht

Interactive contour plot

## Syntax

```
surfht(z)  
surfht(x, y, z)
```

## Description

`surfht(z)` creates an interactive contour plot of the data in matrix `z`. `surfht` treats the values in `z` as the height above the plane.

`surfht(x, y, z)` creates an interactive contour plot of the data in matrix `z`, using the  $x$ -axis values contained in `x` and the  $y$ -axis values contained in `y`.

## Examples

### Create an Interactive Contour Plot

This example shows how to use `surfht` to create an interactive contour plot.

Create a grid of the  $(x,y)$  domain from  $(-2,-2)$  to  $(2,2)$  using `meshgrid`.

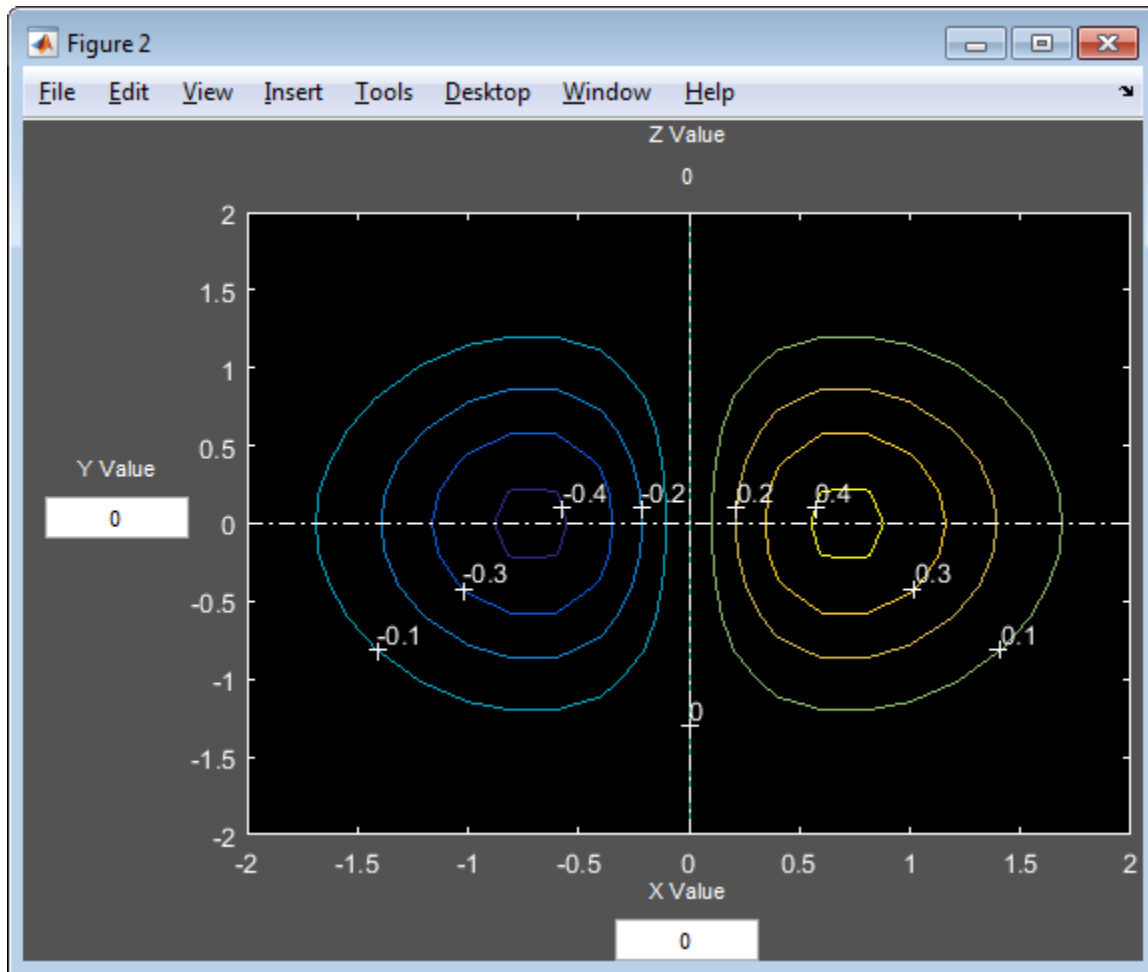
```
[x,y] = meshgrid(-2:0.2:2,-2:0.2:2);
```

Evaluate the function  $z(x,y) = x \times \exp(-x^2 - y^2)$  over this domain.

```
z = x.*exp(-x.^2 - y.^2);
```

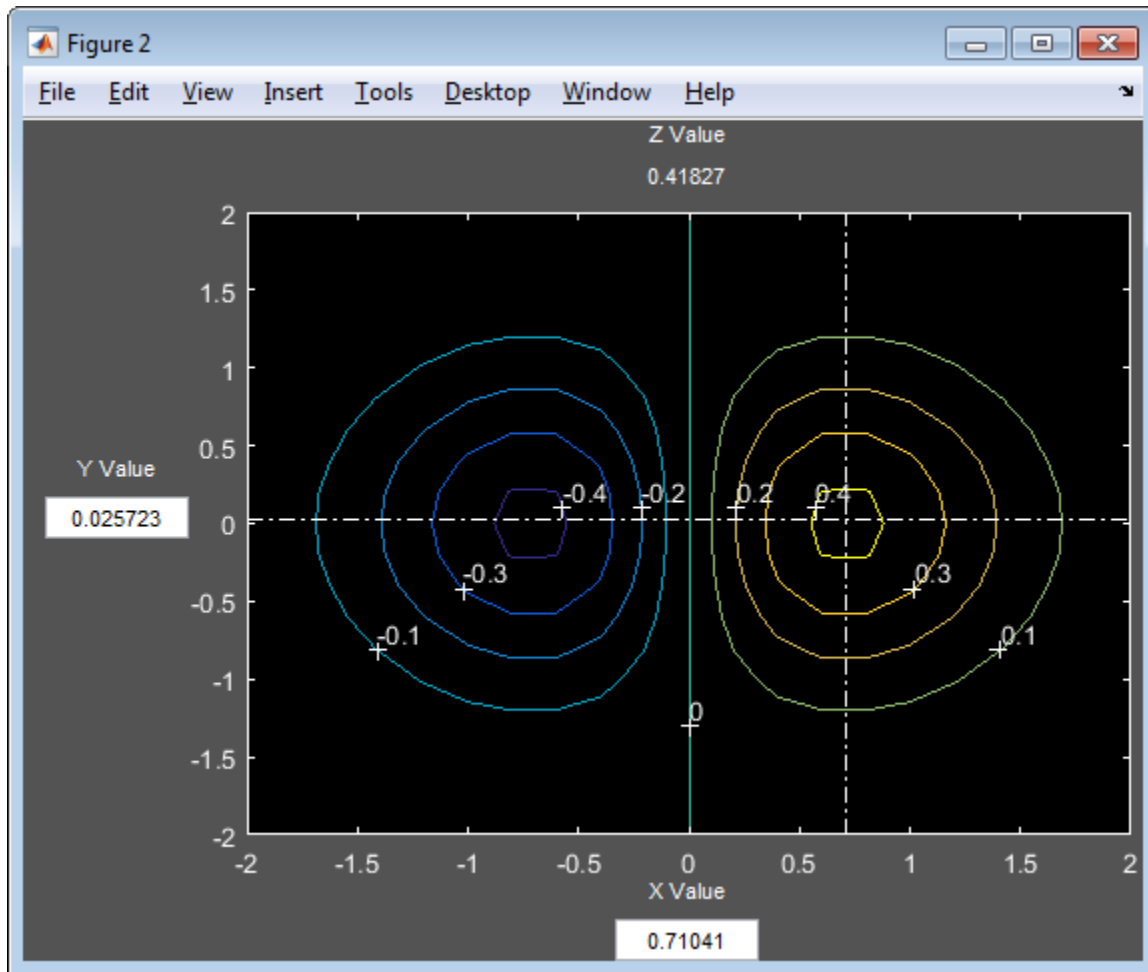
Open an interactive contour plot. Since `meshgrid` creates a grid of the  $x$  and  $y$  values, open the plot using the first row of `x` and the first column of `y`.

```
surfht(x(1,:),y(:,1),z)
```



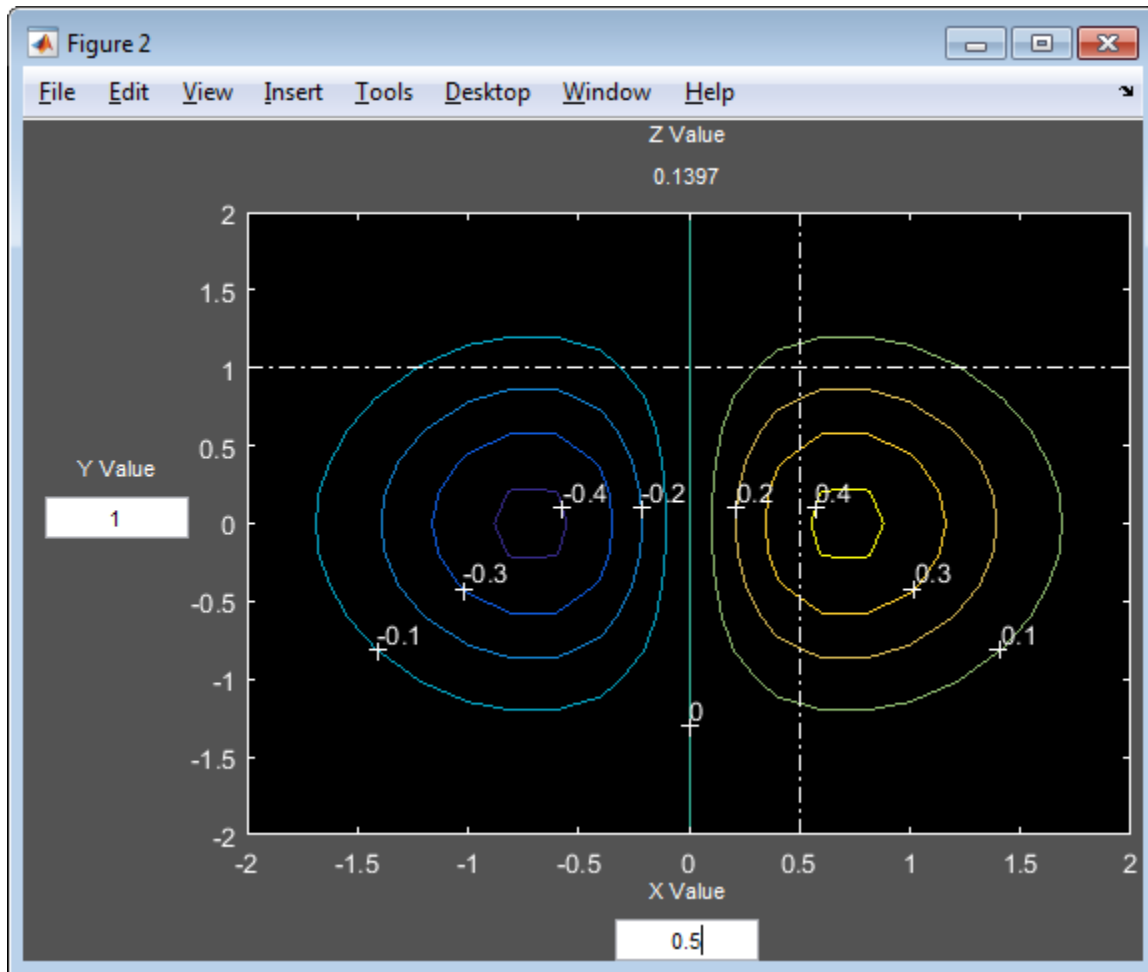
The figure shows a contour plot of the  $z$  values along the specified  $x$ - and  $y$ -axes.

Click the plot to evaluate  $z$  at the  $(x,y)$  coordinates indicated by the intersecting white lines.



For example, at  $x = 0.71041$  and  $y = 0.025723$ , the value of  $z$  is 0.41827.

Alternatively, enter values in the fields labeled  $x$  Value and  $y$  Value to evaluate  $z$  at the specified coordinates. For example, evaluate  $z$  at  $x = 0.5$  and  $y = 1$ .



The value of  $z$  is 0.1397.

## Input Arguments

**$z$**  — **z-axis values for contour plot**  
 numeric matrix

$z$ -axis values for contour plot, specified as a numeric matrix.

`surfht` treats the values in `z` as the height above the plane. By default, the  $x$ -axis values of the plot are the column indices of `z`, and the  $y$ -axis values of the plot are the row indices of `z`. To change the  $x$ - and  $y$ -axis values, specify `x` and `y`, respectively.

Data Types: `single` | `double`

### **`x` — $x$ -axis values for contour plot**

column indices of `z` (default) | numeric vector

$x$ -axis values for contour plot, specified as a numeric vector. The length of `x` must match the number of columns in `z`.

Data Types: `single` | `double`

### **`y` — $y$ -axis values for contour plot**

row indices of `z` (default) | numeric vector

$y$ -axis values for contour plot, specified as a numeric vector. The length of `y` must match the number of rows in `z`.

Data Types: `single` | `double`

## Tips

- The intersection of the vertical and horizontal reference lines on the interactive plot defines the current  $x$  value and  $y$  value.
- Drag the dotted white reference lines to watch the interpolated  $z$  value (at the top of the plot) update simultaneously.
- Alternatively, obtain a specific interpolated  $z$  value by typing the  $x$  value and  $y$  value into editable text fields on the  $x$ -axis and  $y$ -axis, respectively.

## See Also

`contour` | `meshgrid` | `surf`

Introduced before R2006a

## surrctcategories

**Class:** classregtree

Categories used for surrogate splits in decision tree

---

**Note** `classregtree` will be removed in a future release. See `fitctree`, `fitrtree`, `ClassificationTree`, or `RegressionTree` instead.

---

## Syntax

```
C = surrctcategories(T)
C = surrctcategories(T, J)
```

## Description

`C = surrctcategories(T)` returns an  $n$ -element cell array `C` of the categories used for surrogate splits in the decision tree `T`, where  $n$  is the number of nodes in the tree. For each node  $K$ , `C{K}` is a cell array. The length of `C{K}` is equal to the number of surrogate predictors found at this node. Every element of `C{K}` is either an empty character vector for a continuous surrogate predictor or a two-element cell array with categories for a categorical surrogate predictor. The first element of this two-element cell array lists categories assigned to the left child by this surrogate split and the second element of this two-element cell array lists categories assigned to the right child by this surrogate split. The order of the surrogate split variables at each node is matched to the order of variables returned by `surrctvar`. The optimal-split variable at this node is not included. For non-branch (leaf) nodes, `C` contains an empty cell.

`C = surrctcategories(T, J)` takes an array `J` of node numbers and returns the categories for the specified nodes.

## See Also

`classregtree` | `cutcategories` | `surrctpoint` | `surrcttype` | `surrctvar`

## surrctflip

**Class:** classregtree

Numeric cutpoint assignments used for surrogate splits in decision tree

---

**Note** classregtree will be removed in a future release. See fitctree, fitrtree, ClassificationTree, or RegressionTree instead.

---

## Syntax

```
V = surrctflip(T)
V = surrctflip(T,J)
```

## Description

`V = surrctflip(T)` returns an  $n$ -element cell array  $V$  of the numeric cut assignments used for surrogate splits in the decision tree  $T$ , where  $n$  is the number of nodes in the tree. For each node  $K$ ,  $V\{K\}$  is a numeric vector. The length of  $V\{K\}$  is equal to the number of surrogate predictors found at this node. Every element of  $V\{K\}$  is either zero for a categorical surrogate predictor or a numeric cut assignment for a continuous surrogate predictor. The numeric cut assignment can be either -1 or +1. For every surrogate split with a numeric cut  $C$  based on a continuous predictor variable  $Z$ , the left child is chosen if  $Z < C$  and the cut assignment for this surrogate split is +1, or if  $Z \geq C$  and the cut assignment for this surrogate split is -1. Similarly, the right child is chosen if  $Z \geq C$  and the cut assignment for this surrogate split is +1, or if  $Z < C$  and the cut assignment for this surrogate split is -1. The order of the surrogate split variables at each node is matched to the order of variables returned by `surrctvar`. The optimal-split variable at this node is not included. For non-branch (leaf) nodes,  $V$  contains an empty array.

`V = surrctflip(T,J)` takes an array  $J$  of node numbers and returns the cutpoint assignments for the specified nodes.

## See Also

`classregtree` | `cutpoint` | `surrctcategories` | `surrctpoint` | `surrcttype`  
| `surrctvar`



# surrucutpoint

**Class:** classregtree

Cutpoints used for surrogate splits in decision tree

---

**Note** classregtree will be removed in a future release. See fitctree, fitrtree, ClassificationTree, or RegressionTree instead.

---

## Syntax

```
V = surrucutpoint(T)
V = surrucutpoint(T,J)
```

## Description

`V = surrucutpoint(T)` returns an  $n$ -element cell array `V` of the numeric values used for surrogate splits in the decision tree `T`, where  $n$  is the number of nodes in the tree. For each node  $K$ , `V{K}` is a numeric vector. The length of `V{K}` is equal to the number of surrogate predictors found at this node. Every element of `V{K}` is either either NaN for a categorical surrogate predictor or a numeric cut for a continuous surrogate predictor. For every surrogate split with a numeric cut  $C$  based on a continuous predictor variable  $Z$ , the left child is chosen if  $Z < C$  and `surrucutflip` for this surrogate split is -1. Similarly, the right child is chosen if  $Z \geq C$  and `surrucutflip` for this surrogate split is +1, or if  $Z < C$  and `surrucutflip` for this surrogate split is -1. The order of the surrogate split variables at each node is matched to the order of variables returned by `surrucutvar`. The optimal-split variable at this node is not included. For non-branch (leaf) nodes, `V` contains an empty cell.

`V = surrucutpoint(T,J)` takes an array `J` of node numbers and returns the cutpoint assignments for the specified nodes.

## See Also

`classregtree` | `cutpoint` | `surrctcategories` | `surrctflip` | `surrcttype` | `surrctvar`

# surrctype

**Class:** classregtree

Types of surrogate splits used at branches in decision tree

---

**Note** `classregtree` will be removed in a future release. See `fitctree`, `fitrtree`, `ClassificationTree`, or `RegressionTree` instead.

---

## Syntax

```
C = surrctype(T)
C = surrctype(T, J)
```

## Description

`C = surrctype(T)` returns an  $n$ -element cell array `C` indicating types of surrogate splits at each node in the tree `T`, where  $n$  is the number of nodes in the tree. For each node  $K$ , `C{K}` is a cell array with the types of the surrogate split variables at this node. The variables are sorted by the predictive measure of association with the optimal predictor in the descending order, and only variables with the positive predictive measure are included. The order of the surrogate split variables at each node is matched to the order of variables returned by `surrctvar`. The optimal-split variable at this node is not included. For non-branch (leaf) nodes, `C` contains an empty cell. A surrogate split type can be either `'continuous'` if the cut is defined in the form  $Z < V$  for a variable  $Z$  and cutpoint  $V$  or `'categorical'` if the cut is defined by whether  $Z$  takes a value in a set of categories.

`C = surrctype(T, J)` takes an array `J` of node numbers and returns the cut types for the specified nodes.

## See Also

`classregtree` | `cuttype` | `numnodes` | `surrctvar`

## surrctvar

**Class:** classregtree

Variables used for surrogate splits in decision tree

---

**Note** `classregtree` will be removed in a future release. See `fitctree`, `fitrtree`, `ClassificationTree`, or `RegressionTree` instead.

---

## Syntax

```
V = surrctvar(T)
V = surrctvar(T,J)
[V,NUM] = surrctvar(...)
```

## Description

`V = surrctvar(T)` returns an  $n$ -element cell array `V` of the names of the variables used for surrogate splits in each node of the tree `T`, where  $n$  is the number of nodes in the tree. Every element of `V` is a cell array with the names of the surrogate split variables at this node. The variables are sorted by the predictive measure of association with the optimal predictor in the descending order, and only variables with the positive predictive measure are included. The optimal-split variable at this node is not included. For non-branch (leaf) nodes, `V` contains an empty cell.

`V = surrctvar(T,J)` takes an array `J` of node numbers and returns the cut types for the specified nodes.

`[V,NUM] = surrctvar(...)` also returns a cell array `NUM` with indices for each variable.

## See Also

`children` | `classregtree` | `cutvar` | `numnodes`

## surrvarassoc

**Class:** classregtree

Predictive measure of association for surrogate splits in decision tree

---

**Note** classregtree will be removed in a future release. See fitctree, fitrtree, ClassificationTree, or RegressionTree instead.

---

### Syntax

```
A = surrvarassoc(T)
A = surrvarassoc(T,J)
```

### Description

`A = surrvarassoc(T)` returns an  $n$ -element cell array `A` of the predictive measures of association for surrogate splits in the decision tree `T`, where  $n$  is the number of nodes in the tree. For each node  $K$ , `A{K}` is a numeric vector. The length of `A{K}` is equal to the number of surrogate predictors found at this node. Every element of `A{K}` gives the predictive measure of association between the optimal split and this surrogate split. The order of the surrogate split variables at each node is matched to the order of variables returned by `surrvar`. The optimal-split variable at this node is not included. For non-branch (leaf) nodes, `V` contains an empty cell.

`A = surrvarassoc(T,J)` takes an array `J` of node numbers and returns the predictive measure of association for the specified nodes.

### See Also

classregtree | surrcutcategories | surrcutflip | surrcutpoint |  
surrcuttype | surrcutvar

## svmclassify

Classify using support vector machine (SVM)

---

**Note** `svmclassify` will be removed in a future release. See `fitcsvm`, `ClassificationSVM`, and `CompactClassificationSVM` instead.

---

## Syntax

```
Group = svmclassify(SVMStruct, Sample)
Group = svmclassify(SVMStruct, Sample, 'Showplot', true)
```

## Description

`Group = svmclassify(SVMStruct, Sample)` classifies each row of the data in `Sample`, a matrix of data, using the information in a support vector machine classifier structure `SVMStruct`, created using the `svmtrain` function. Like the training data used to create `SVMStruct`, `Sample` is a matrix where each row corresponds to an observation or replicate, and each column corresponds to a feature or variable. Therefore, `Sample` must have the same number of columns as the training data. This is because the number of columns defines the number of features. `Group` indicates the group to which each row of `Sample` has been assigned.

`Group = svmclassify(SVMStruct, Sample, 'Showplot', true)` plots the `Sample` data in the figure created using the `Showplot` property with the `svmtrain` function. This plot appears only when the data is two-dimensional.

## Input Arguments

### **SVMStruct**

Support vector machine classifier structure created using the `svmtrain` function.

### Sample

A matrix where each row corresponds to an observation or replicate, and each column corresponds to a feature or variable. Therefore, `Sample` must have the same number of columns as the training data. This is because the number of columns defines the dimensionality of the data space.

### Showplot

Describes whether to display a plot of the classification. Displays only for 2-D problems. Follow with a Boolean argument: `true` to display the plot, `false` to give no display.

## Output Arguments

### Group

Column vector with the same number of rows as `Sample`. Each entry (row) in `Group` represents the class of the corresponding row of `Sample`.

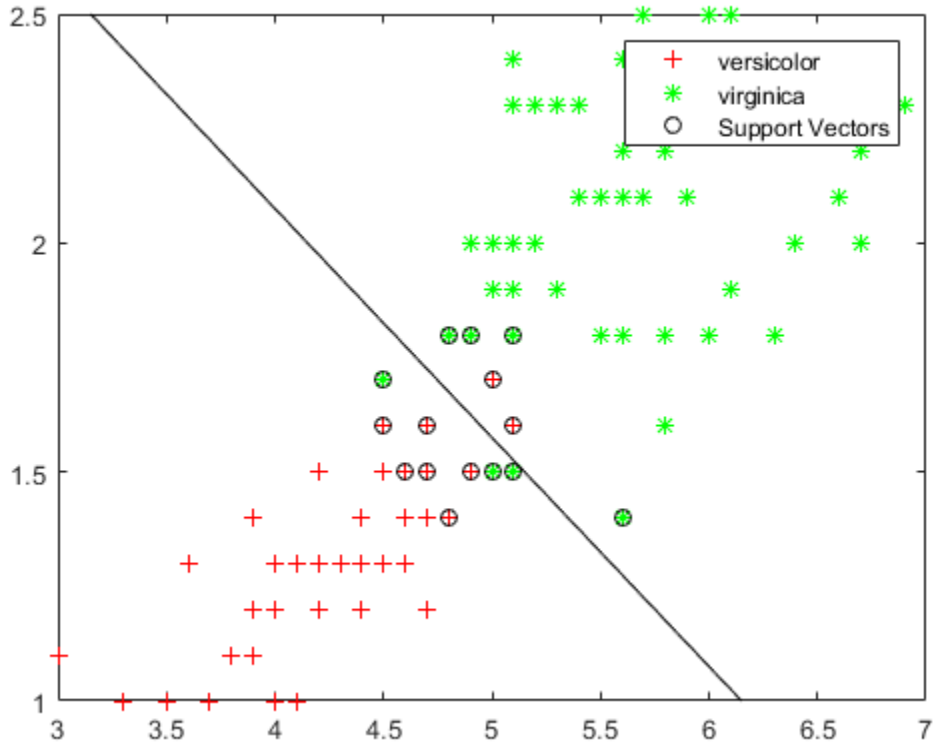
## Examples

### Classify an Observation Using a Trained SVM Classifier.

Find a line separating the Fisher iris data on `versicolor` and `virginica` species, according to the petal length and petal width measurements. These two species are in rows 51 and higher of the data set, and the petal length and width are the third and fourth columns.

```
load fisheriris
xdata = meas(51:end,3:4);
group = species(51:end);
figure;
svmStruct = svmtrain(xdata,group,'ShowPlot',true);
```

Warning: `svmtrain` will be removed in a future release. Use `fitcsvm` instead.



Classify two new flowers, one with petal length 5 and petal width 2 and the other with petal length 4 and petal width 1.5.

```
Xnew = [5 2; 4 1.5];
species = svmclassify(svmStruct,Xnew,'ShowPlot',true)
hold on;
plot(Xnew(:,1),Xnew(:,2),'ro','MarkerSize',12);
hold off
```

Warning: svmclassify will be removed in a future release. Use the predict method of an object returned by fitcsvm instead.

```
species =
```

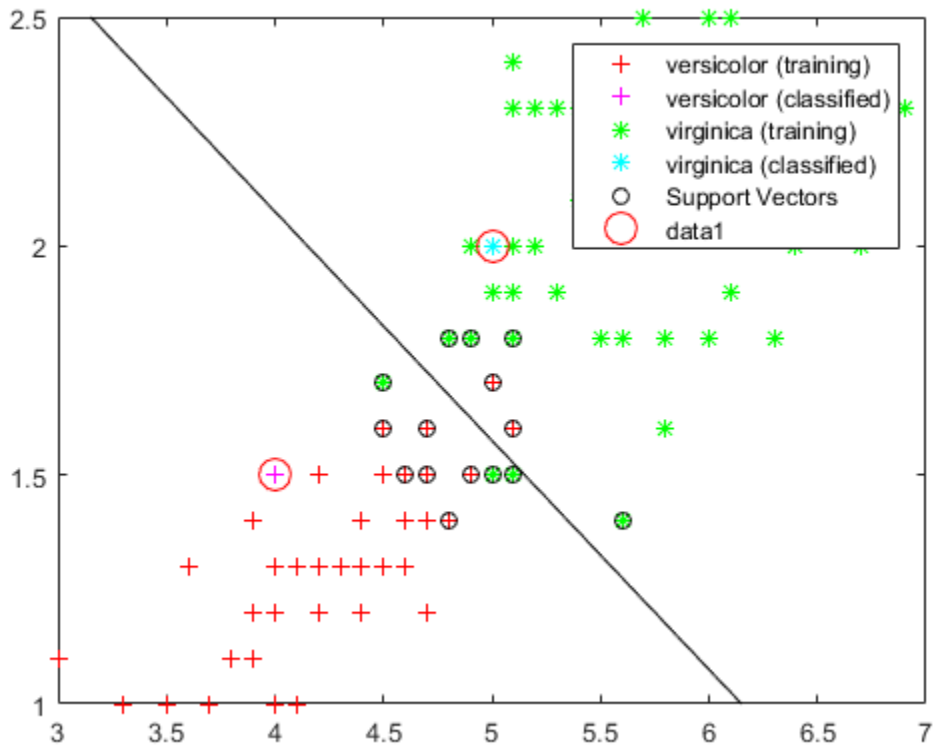


```

2x1 cell array

    {'virginica' }
    {'versicolor'}

```



## Algorithms

The `svmclassify` function uses results from `svmtrain` to classify vectors  $x$  according to the following equation:

$$c = \sum_i \alpha_i k(s_i, x) + b,$$

where  $s_i$  are the support vectors,  $\alpha_i$  are the weights,  $b$  is the bias, and  $k$  is a kernel function. In the case of a linear kernel,  $k$  is the dot product. If  $c \geq 0$ , then  $x$  is classified as a member of the first group, otherwise it is classified as a member of the second group.

## References

- [1] Kecman, V., Learning and Soft Computing, MIT Press, Cambridge, MA. 2001.
- [2] Suykens, J.A.K., Van Gestel, T., De Brabanter, J., De Moor, B., and Vandewalle, J., Least Squares Support Vector Machines, World Scientific, Singapore, 2002.
- [3] Scholkopf, B., and Smola, A.J., Learning with Kernels, MIT Press, Cambridge, MA. 2002.
- [4] Cristianini, N., and Shawe-Taylor, J. (2000). An Introduction to Support Vector Machines and Other Kernel-based Learning Methods, First Edition (Cambridge: Cambridge University Press). <http://www.support-vector.net/>

## See Also

`svmtrain`

## Topics

“Support Vector Machines for Binary Classification” on page 18-186

Introduced in R2013a

## svmtrain

Train support vector machine classifier

---

**Note** `svmtrain` will be removed in a future release. See `fitcsvm`, `ClassificationSVM`, and `CompactClassificationSVM` instead.

---

## Syntax

```
SVMStruct = svmtrain(Training, Group)
SVMStruct = svmtrain(Training, Group, Name, Value)
```

## Description

`SVMStruct = svmtrain(Training, Group)` returns a structure, `SVMStruct`, containing information about the trained support vector machine (SVM) classifier.

`SVMStruct = svmtrain(Training, Group, Name, Value)` returns a structure with additional options specified by one or more `Name, Value` pair arguments.

## Input Arguments

### Training

Matrix of training data, where each row corresponds to an observation or replicate, and each column corresponds to a feature or variable. `svmtrain` treats NaNs or empty character vectors in `Training` as missing values and ignores the corresponding rows of `Group`.

### Group

Grouping variable, which can be a categorical, numeric, or logical vector, a cell array of character vectors, or a character matrix with each row representing a class label. Each element of `Group` specifies the group of the corresponding row of `Training`. `Group`

should divide `Training` into two groups. `Group` has the same number of elements as there are rows in `Training`. `svmtrain` treats each `NaN`, empty character vector, or 'undefined' in `Group` as a missing value, and ignores the corresponding row of `Training`.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### **autoscale**

Boolean specifying whether `svmtrain` automatically centers the data points at their mean, and scales them to have unit standard deviation, before training.

**Default:** `true`

### **boxconstraint**

Value of the box constraint `C` for the soft margin. `C` can be a scalar, or a vector of the same length as the training data.

If `C` is a scalar, it is automatically rescaled by  $N / (2 * N1)$  for the data points of group one and by  $N / (2 * N2)$  for the data points of group two, where  $N1$  is the number of elements in group one,  $N2$  is the number of elements in group two, and  $N = N1 + N2$ . This rescaling is done to take into account unbalanced groups, that is cases where  $N1$  and  $N2$  have very different values.

If `C` is an array, then each array element is taken as a box constraint for the data point with the same index.

**Default:** `1`

### **kernelcachelimit**

Value that specifies the size of the kernel matrix cache for the SMO training method. The algorithm keeps a matrix with up to `kernelcachelimit` × `kernelcachelimit` double-precision, floating-point numbers in memory.

**Default:** 5000

### **kernel\_function**

Kernel function `svmtrain` uses to map the training data into kernel space. The default kernel function is the dot product. The kernel function can be one of the following character vectors or a function handle:

- 'linear' — Linear kernel, meaning dot product.
- 'quadratic' — Quadratic kernel.
- 'polynomial' — Polynomial kernel (default order 3). Specify another order with the `polyorder` name-value pair.
- 'rbf' — Gaussian Radial Basis Function kernel with a default scaling factor, `sigma`, of 1. Specify another value for `sigma` with the `rbf_sigma` name-value pair.
- 'mlp' — Multilayer Perceptron kernel with default scale `[1 -1]`. Specify another scale with the `mlp_params` name-value pair.
- `@kfun` — Function handle to a kernel function. A kernel function must be of the form

```
function K = kfun(U, V)
```

The returned value, `K`, is a matrix of size `M-by-N`, where `U` and `V` have `M` and `N` rows respectively.

If `kfun` has extra parameters, include the extra parameters via an anonymous function. For example, suppose that your kernel function is:

```
function k = kfun(u,v,p1,p2)
k = tanh(p1*(u*v')+p2);
```

Set values for `p1` and `p2`, and then use an anonymous function:

```
@(u,v) kfun(u,v,p1,p2)
```

**Default:** 'linear'

### **kktviolationlevel**

Value that specifies the fraction of variables allowed to violate the Karush-Kuhn-Tucker (KKT) conditions for the SMO training method. Set any value in `[0,1)`. For example, if you set `kktviolationlevel` to `0.05`, then 5% of the variables are allowed to violate the KKT conditions.

---

**Tip** Set this option to a positive value to help the algorithm converge if it is fluctuating near a good solution.

---

For more information on KKT conditions, see Cristianini and Shawe-Taylor [4].

**Default:** 0

**method**

Method used to find the separating hyperplane. Options are:

- 'QP' — Quadratic programming (requires an Optimization Toolbox license). The classifier is a 2-norm soft-margin support vector machine. Give quadratic programming options with the `options` name-value pair, and create options with `optimset`.
- 'SMO' — Sequential Minimal Optimization. Give SMO options with the `options` name-value pair, and create options with `statset`.
- 'LS' — Least squares.

**Default:** SMO

**mlp\_params**

Parameters of the Multilayer Perceptron (mlp) kernel. The mlp kernel requires two parameters,  $[P1 \ P2]$ . The kernel  $K = \tanh(P1*U*V' + P2)$ , where  $P1 > 0$  and  $P2 < 0$ .

**Default:** [1 -1]

**options**

Options structure for training.

- When you set 'method' to 'SMO' (default), create the options structure using `statset`. Options are:

Display	<p>Character vector that specifies the level of information about the optimization iterations that is displayed as the algorithm runs. Choices are:</p> <ul style="list-style-type: none"> <li>• <code>off</code> (default) — Reports nothing.</li> <li>• <code>iter</code> — Reports every 500 iterations.</li> <li>• <code>final</code> — Reports only when the algorithm finishes.</li> </ul>
MaxIter	<p>Integer that specifies the maximum number of iterations of the main loop. If this limit is exceeded before the algorithm converges, then the algorithm stops and returns an error. Default is 15000.</p>

The other name-value pairs that relate specifically to the 'SMO' method are `kernelcachelimit`, `kktviolationlevel`, and `tolkkt`.

- When you set `method` to 'QP', create the options structure using `optimset`. For details of applicable option choices, see `quadprog` options. SVM uses a convex quadratic program, so you can choose the 'interior-point-convex' `quadprog` algorithm. In limited testing, the 'interior-point-convex' algorithm was the best `quadprog` option for `svmtrain`, in both speed and memory utilization.

### **polyorder**

Order of the polynomial kernel.

**Default:** 3

### **rbf\_sigma**

Scaling factor (sigma) in the radial basis function kernel.

**Default:** 1

### **showplot**

Boolean indicating whether to plot the grouped data and separating line. Creates a plot only when the data has two columns (features).

**Default:** `false`

**tolkkt**

Value that specifies the tolerance with which the Karush-Kuhn-Tucker (KKT) conditions are checked for the SMO training method. For a definition of KKT conditions, see “Karush-Kuhn-Tucker (KKT) Conditions” on page 32-6348.

**Default:** 1e-3

## Output Arguments

**SVMStruct**

Structure containing information about the trained SVM classifier in the following fields:

- **SupportVectors** — Matrix of data points with each row corresponding to a support vector in the normalized data space. This matrix is a subset of the *Training* input data matrix, after normalization has been applied according to the 'AutoScale' argument.
- **Alpha** — Vector of weights for the support vectors. The sign of the weight is positive for support vectors belonging to the first group, and negative for the second group.
- **Bias** — Intercept of the hyperplane that separates the two groups in the normalized data space (according to the 'AutoScale' argument).
- **KernelFunction** — Handle to the function that maps the training data into kernel space.
- **KernelFunctionArgs** — Cell array of any additional arguments required by the kernel function.
- **GroupNames** — Categorical, numeric, or logical vector, a cell array of character vectors, or a character matrix with each row representing a class label. Specifies the group identifiers for the support vectors. It has the same number of elements as there are rows in **SupportVectors**. Each element specifies the group to which the corresponding row in **SupportVectors** belongs.
- **SupportVectorIndices** — Vector of indices that specify the rows in **Training**, the training data, that were selected as support vectors after the data was normalized, according to the **AutoScale** argument.
- **ScaleData** — Field containing normalization factors. When 'AutoScale' is set to false, it is empty. When **AutoScale** is set to true, it is a structure containing two fields:



- `shift` — Row vector of values. Each value is the negative of the mean across an observation in *Training*, the training data.
- `scaleFactor` — Row vector of values. Each value is 1 divided by the standard deviation of an observation in *Training*, the training data.

Both `svmtrain` and `svmclassify` apply the scaling in `ScaleData`.

- `FigureHandles` — Vector of figure handles created by `svmtrain` when using the `'Showplot'` argument.

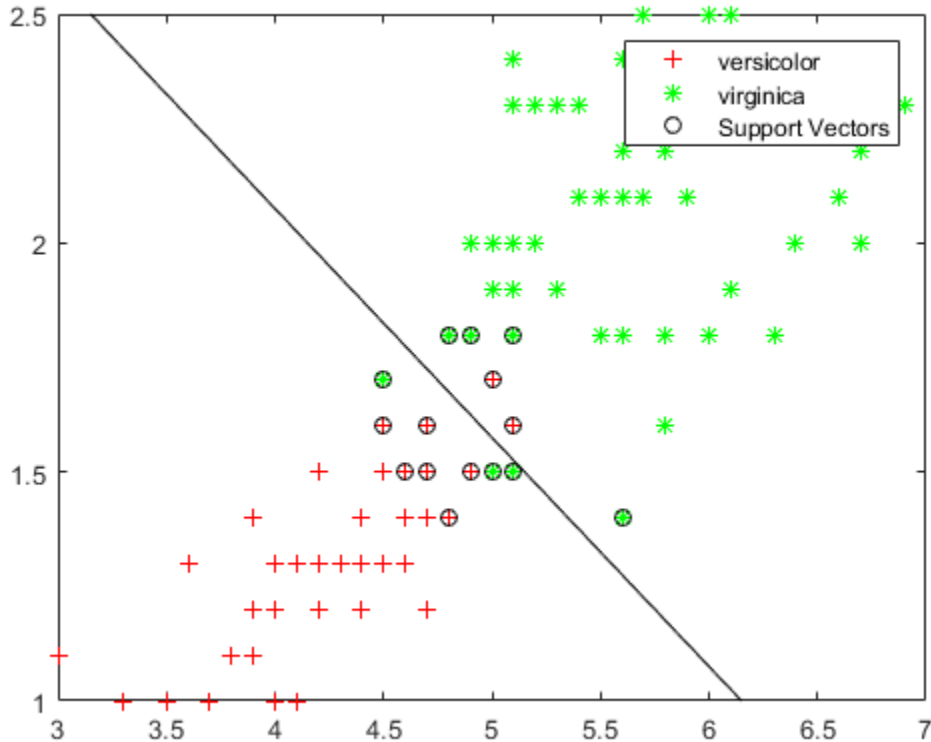
## Examples

### Train an SVM Classifier

Find a line separating the Fisher iris data on *versicolor* and *virginica* species, according to the petal length and petal width measurements. These two species are in rows 51 and higher of the data set, and the petal length and width are the third and fourth columns.

```
load fisheriris
xdata = meas(51:end,3:4);
group = species(51:end);
svmStruct = svmtrain(xdata,group,'ShowPlot',true);
```

Warning: `svmtrain` will be removed in a future release. Use `fitcsvm` instead.



## Definitions

### Karush-Kuhn-Tucker (KKT) Conditions

The Karush-Kuhn-Tucker (KKT) conditions are analogous to the condition that the gradient must be zero at a minimum, modified to take constraints into account. The difference is that the KKT conditions hold for constrained problems. The KKT conditions use the auxiliary Lagrangian function:

$$L(x, \lambda) = f(x) + \sum \lambda_{g,i} g_i(x) + \sum \lambda_{h,i} h_i(x).$$

Here  $f(x)$  is the objective function,  $g(x)$  is a vector of constraint functions  $g(x) \leq 0$ , and  $h(x)$  is a vector of constraint functions  $h(x) = 0$ . The vector  $\lambda$ , which is the concatenation of  $\lambda_g$  and  $\lambda_h$ , is the Lagrange multiplier vector. Its length is the total number of constraints.

The KKT conditions are:

$$\begin{aligned}\nabla_x L(x, \lambda) &= 0 \\ \lambda_{g,i} g_i(x) &= 0 \quad \forall i \\ g(x) &\leq 0 \\ h(x) &= 0 \\ \lambda_{g,i} &\geq 0.\end{aligned}$$

For more information, see Karush-Kuhn-Tucker conditions.

## Tips

- To classify new data, use the result of training, `SVMStruct`, with the `svmclassify` function.

## Algorithms

The `svmtrain` function uses an optimization method to identify support vectors  $s_i$ , weights  $a_i$ , and bias  $b$  that are used to classify vectors  $x$  according to the following equation:

$$c = \sum_i \alpha_i k(s_i, x) + b,$$

where  $k$  is a kernel function. In the case of a linear kernel,  $k$  is the dot product. If  $c \geq 0$ , then  $x$  is classified as a member of the first group, otherwise it is classified as a member of the second group.

## Memory Usage and Out of Memory Error

When you set 'Method' to 'QP', the `svmtrain` function operates on a data set containing  $N$  elements, and it creates an  $(N+1)$ -by- $(N+1)$  matrix to find the separating hyperplane. This matrix needs at least  $8 * (n+1)^2$  bytes of contiguous memory. If this

size of contiguous memory is not available, the software displays an “out of memory” error message.

When you set 'Method' to 'SMO' (default), memory consumption is controlled by the `kernelcachelimit` option. The SMO algorithm stores only a submatrix of the kernel matrix, limited by the size specified by the `kernelcachelimit` option. However, if the number of data points exceeds the size specified by the `kernelcachelimit` option, the SMO algorithm slows down because it has to recalculate the kernel matrix elements.

When using `svmtrain` on large data sets, and you run out of memory or the optimization step is very time consuming, try either of the following:

- Use a smaller number of samples and use cross-validation to test the performance of the classifier.
- Set 'Method' to 'SMO', and set the `kernelcachelimit` option as large as your system permits.

## References

- [1] Kecman, V., *Learning and Soft Computing*, MIT Press, Cambridge, MA. 2001.
- [2] Suykens, J.A.K., Van Gestel, T., De Brabanter, J., De Moor, B., and Vandewalle, J., *Least Squares Support Vector Machines*, World Scientific, Singapore, 2002.
- [3] Scholkopf, B., and Smola, A.J., *Learning with Kernels*, MIT Press, Cambridge, MA. 2002.
- [4] Cristianini, N., and Shawe-Taylor, J. (2000). *An Introduction to Support Vector Machines and Other Kernel-based Learning Methods*, First Edition (Cambridge: Cambridge University Press). <http://www.support-vector.net/>

## See Also

`classify` | `svmclassify`

## Topics

“Support Vector Machines for Binary Classification” on page 18-186

“Grouping Variables” on page 2-59

Introduced in R2013a

## table2dataset

Convert table to dataset array

---

**Note** The `dataset` data type might be removed in a future release. To work with heterogeneous data, use the MATLAB® `table` data type instead. See MATLAB `table` documentation for more information.

---

### Syntax

```
ds = table2dataset(t)
```

### Description

`ds = table2dataset(t)` converts a table to a dataset array.

### Examples

#### Convert a Table to a Dataset Array

Load the sample data, which contains nutritional information for 77 cereals.

```
load cereal;
```

Create a table containing the calorie, protein, fat, and name data for the first five cereals. Label the variables.

```
Calories = Calories(1:5);  
Protein = Protein(1:5);  
Fat = Fat(1:5);  
Name = Name(1:5);  
  
cereal = table(Calories,Protein,Fat,'RowNames',Name)  
  
cereal =
```

5x3 table

	Calories	Protein	Fat
	-----	-----	-----
100% Bran	70	4	1
100% Natural Bran	120	3	5
All-Bran	70	4	1
All-Bran with Extra Fiber	50	4	0
Almond Delight	110	2	2

Convert the table to a dataset array.

```
ds = table2dataset(cereal)
```

ds =

	Calories	Protein	Fat
100% Bran	70	4	1
100% Natural Bran	120	3	5
All-Bran	70	4	1
All-Bran with Extra Fiber	50	4	0
Almond Delight	110	2	2

## Input Arguments

**t** — Input table

table

Input table to convert to a dataset array, specified as a table. Each variable in **t** becomes a variable in the output dataset array **ds**.

Data Types: table

## Output Arguments

### **ds** — Output dataset array

dataset array

Output dataset array, returned as a dataset array containing the variables from the input table `t`.

## See Also

`dataset` | `table`

## Topics

“Dataset Arrays” on page 2-142

“Tables” (MATLAB)

**Introduced in R2013b**



# tabulate

Frequency table

## Syntax

```
tbl = tabulate(x)
tabulate(x)
```

## Description

`tbl = tabulate(x)` creates a frequency table of data in vector `x`. Information in `tbl` is arranged as follows:

- 1st column — The unique values of `x`
- 2nd column — The number of instances of each value
- 3rd column — The percentage of each value

If `x` is a numeric array, `tbl` is a numeric matrix. If the elements of `x` are positive integers, `tbl` includes 0 counts for integers between 1 and `max(x)` that do not appear in `x`.

If `x` is a categorical variable, character array, or cell array of character vectors, `tbl` is a cell array.

`tabulate(x)` with no output arguments displays the table in the command window.

## Examples

```
tabulate([1 2 4 4 3 4])
```

Value	Count	Percent
1	1	16.67%
2	1	16.67%
3	1	16.67%
4	3	50.00%

## Extended Capabilities

### Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays” (MATLAB).

### See Also

`pareto`

### Topics

“Grouping Variables” on page 2-59

Introduced before R2006a

## tblread

Read tabular data from file

### Syntax

```
[data,varnames,casenames] = tblread
[data,varnames,casenames] = tblread(filename)
[data,varnames,casenames] = tblread(filename,delimiter)
```

### Description

`[data,varnames,casenames] = tblread` displays the File Open dialog box for interactive selection of a tabular data file. The file format has variable names in the first row, case names in the first column and data starting in the (2, 2) position. Outputs are:

- `data` — Numeric matrix with a value for each variable-case pair
- `varnames` — Character matrix containing the variable names in the first row of the file
- `casenames` — Character matrix containing the names of each case in the first column of the file

`[data,varnames,casenames] = tblread(filename)` allows command line specification of the name of a file in the current folder, or the complete path name of any file, using the character vector `filename`.

`[data,varnames,casenames] = tblread(filename,delimiter)` reads from the file using `delimiter` as the delimiting character. Accepted values for `delimiter` are:

- ' ' or 'space'
- '\t' or 'tab'
- ',' or 'comma'
- ';' or 'semi'
- '|' or 'bar'

The default value of *delimiter* is 'space'.

## Examples

```
[data,varnames,casenames] = tblread('sat.dat')
data =
  470  530
  520  480

varnames =
Male
Female

casenames =
Verbal
Quantitative
```

## See Also

[caseread](#) | [readtable](#) | [tblwrite](#) | [tdfread](#)

**Introduced before R2006a**

## tblwrite

Write tabular data to file

### Syntax

```
tblwrite(data, varnames, casenames)
tblwrite(data, varnames, casenames, filename)
tblwrite(data, varnames, casenames, filename, delimiter)
```

### Description

`tblwrite(data, varnames, casenames)` displays the **File Open** dialog box for interactive specification of the tabular data output file. The file format has variable names in the first row, case names in the first column and data starting in the (2,2) position.

`varnames` is a character matrix containing the variable names. `casenames` is a character matrix containing the names of each case in the first column. `data` is a character matrix with a value for each variable-case pair.

`tblwrite(data, varnames, casenames, filename)` specifies a file in the current folder, or the complete path name of any file in the character vector `filename`.

`tblwrite(data, varnames, casenames, filename, delimiter)` writes to the file using `delimiter` as the delimiting character. The following table lists the accepted delimiting character for `delimiter` and their equivalent names.

Character	Name
' '	'space'
'\t'	'tab'
','	'comma'
';'	'semi'
' '	'bar'

The default value of `delimiter` is 'space'.

## Examples

Continuing the example from `tblread`:

```
tblwrite(data, varnames, casenames, 'satteest.dat')
type satteest.dat
```

	Male	Female
Verbal	470	530
Quantitative	520	480

## See Also

`casewrite` | `tblread` | `writetable`

**Introduced before R2006a**

## tcdf

Student's  $t$  cumulative distribution function

### Syntax

```
p = tcdf(x,nu)
p = tcdf(x,nu,'upper')
```

### Description

`p = tcdf(x,nu)` returns the cumulative distribution function (cdf) of the Student's  $t$  distribution at each of the values in `x` using the corresponding degrees of freedom in `nu`. `x` and `nu` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array with the same dimensions as the other inputs.

`p = tcdf(x,nu,'upper')` returns the complement of the Student's  $t$  cdf at each value in `x`, using an algorithm that more accurately computes the extreme upper tail probabilities.

### Examples

#### Compute Student's $t$ cdf

```
mu = 1;      % Population mean
sigma = 2;   % Population standard deviation
n = 100;     % Sample size

rng default % For reproducibility
x = normrnd(mu,sigma,n,1); % Random sample from population

xbar = mean(x); % Sample mean
s = std(x);     % Sample standard deviation
t = (xbar - mu)/(s/sqrt(n))
```

```
t =  
    1.0589  
  
p = 1-tcdf(t,n-1) % Probability of larger t-statistic  
  
p =  
    0.1461
```

This probability is the same as the  $p$  value returned by a  $t$  test of the null hypothesis that the sample comes from a normal population with mean  $\mu$

```
[h,ptest] = ttest(x,mu,0.05,'right')  
  
h =  
    0  
  
ptest =  
    0.1461
```

## Definitions

### Student's $t$ cdf

The cumulative distribution function (cdf) of Student's  $t$  distribution is

$$p = F(x | \nu) = \int_{-\infty}^x \frac{\Gamma\left(\frac{\nu+1}{2}\right)}{\Gamma\left(\frac{\nu}{2}\right)} \frac{1}{\sqrt{\nu\pi}} \frac{1}{\left(1 + \frac{t^2}{\nu}\right)^{\frac{\nu+1}{2}}} dt$$



where  $\nu$  is the degrees of freedom and  $\Gamma(\cdot)$  is the Gamma function. The result  $p$  is the probability that a single observation from the  $t$  distribution with  $\nu$  degrees of freedom will fall in the interval  $[-\infty, x]$ .

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`cdf` | `tinu` | `tpdf` | `trnd` | `tstat`

### Topics

“Student's  $t$  Distribution” on page B-166

Introduced before R2006a

## tdfread

Read tab-delimited file

### Syntax

```
tdfread
tdfread(filename)
tdfread(filename, delimiter)
s = tdfread(filename, ...)
```

### Description

`tdfread` displays the **File Open** dialog box for interactive selection of a data file, then reads data from the file. The file should have variable names separated by tabs in the first row, and data values separated by tabs in the remaining rows. `tdfread` creates variables in the workspace, one for each column of the file. The variable names are taken from the first row of the file. If a column of the file contains only numeric data in the second and following rows, `tdfread` creates a double variable. Otherwise, `tdfread` creates a char variable. After all values are imported, `tdfread` displays information about the imported values using the format of the `tdfread` command.

`tdfread(filename)` allows command line specification of the name of a file in the current folder, or the complete path name of any file, using *filename*.

`tdfread(filename, delimiter)` indicates that the character specified by *delimiter* separates columns in the file. Accepted values for *delimiter* are:

- ' ' or 'space'
- '\t' or 'tab'
- ',' or 'comma'
- ';' or 'semi'
- '|' or 'bar'

The default delimiter is 'tab'.

`s = tdfread(filename, ...)` returns a scalar structure `s` whose fields each contain a variable.

## Examples

The following displays the contents of the file `sat2.dat`:

```
type sat2.dat

Test,Gender,Score
Verbal,Male,470
Verbal,Female,530
Quantitative,Male,520
Quantitative,Female,480
```

The following creates the variables `Gender`, `Score`, and `Test` from the file `sat2.dat` and displays the contents of the MATLAB workspace:

```
tdfread('sat2.dat','')

Name      Size      Bytes      Class      Attributes
Gender    4x6        48         char
Score     4x1        32         double
Test      4x12       96         char
```

## See Also

[caseread](#) | [dlmread](#) | [readtable](#) | [tblread](#) | [textscan](#)

Introduced before R2006a

## ClassificationDiscriminant.template

**Class:** ClassificationDiscriminant

Discriminant analysis classifier template for ensemble (to be removed)

---

**Note** `ClassificationDiscriminant.template` will be removed in a future release. Use `templateDiscriminant` instead.

---

### Syntax

```
t = ClassificationDiscriminant.template()  
t = ClassificationDiscriminant.template(Name,Value)
```

### Description

`t = ClassificationDiscriminant.template()` returns a learner template suitable to use in the `fitcensemble` function.

`t = ClassificationDiscriminant.template(Name,Value)` creates a template with additional options specified by one or more `Name,Value` pair arguments.

### Input Arguments

#### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

#### **Delta** — Linear coefficient threshold

0 (default) | nonnegative scalar value

Linear coefficient threshold, specified as the comma-separated pair consisting of 'Delta' and a nonnegative scalar value. If a coefficient of `Mdl` has magnitude smaller than `Delta`, `Mdl` sets this coefficient to 0, and you can eliminate the corresponding predictor from the model. Set `Delta` to a higher value to eliminate more predictors.

`Delta` must be 0 for quadratic discriminant models.

Data Types: `single` | `double`

### **DiscrimType** — Discriminant type

'linear' (default) | 'quadratic' | 'diaglinear' | 'diagquadratic' | 'pseudolinear' | 'pseudoquadratic'

Discriminant type, specified as the comma-separated pair consisting of 'DiscrimType' and a character vector in this table.

Value	Description	Predictor Covariance Treatment
'linear'	Regularized linear discriminant analysis (LDA)	<ul style="list-style-type: none"> <li>All classes have the same covariance matrix.</li> <li><math>\hat{\Sigma}_\gamma = (1 - \gamma)\hat{\Sigma} + \gamma \text{diag}(\hat{\Sigma})</math>.</li> </ul> <p><math>\hat{\Sigma}</math> is the empirical, pooled covariance matrix and <math>\gamma</math> is the amount of regularization.</p>
'diaglinear'	LDA	All classes have the same, diagonal covariance matrix.
'pseudolinear'	LDA	All classes have the same covariance matrix. The software inverts the covariance matrix using the pseudo inverse.
'quadratic'	Quadratic discriminant analysis (QDA)	The covariance matrices can vary among classes.
'diagquadratic'	QDA	The covariance matrices are diagonal and can vary among classes.

Value	Description	Predictor Covariance Treatment
'pseudoquadratic'	QDA	The covariance matrices can vary among classes. The software inverts the covariance matrix using the pseudo inverse.

---

**Note** To use regularization, you must specify 'linear'. To specify the amount of regularization, use the `Gamma` name-value pair argument.

---

Example: `'DiscrimType', 'quadratic'`

#### **FillCoeffs — Coeffs property flag**

'on' | 'off'

`Coeffs` property flag, specified as the comma-separated pair consisting of 'FillCoeffs' and 'on' or 'off'. Setting the flag to 'on' populates the `Coeffs` property in the classifier object. This can be computationally intensive, especially when cross validating. The default is 'on', unless you specify a cross validation name-value pair, in which case the flag is set to 'off' by default.

Example: `'FillCoeffs', 'off'`

#### **Gamma — Amount of regularization**

scalar value in the interval [0,1]

Amount of regularization to apply when estimating the covariance matrix of the predictors, specified as the comma-separated pair consisting of 'Gamma' and a scalar value in the interval [0,1]. `Gamma` provides finer control over the covariance matrix structure than `DiscrimType`.

- If you specify 0, then the software does not use regularization to adjust the covariance matrix. That is, the software estimates and uses the unrestricted, empirical covariance matrix.
- For linear discriminant analysis, if the empirical covariance matrix is singular, then the software automatically applies the minimal regularization required to

invert the covariance matrix. You can display the chosen regularization amount by entering `Mdl.Gamma` at the command line.

- For quadratic discriminant analysis, if at least one class has an empirical covariance matrix that is singular, then the software throws an error.
- If you specify a value in the interval (0,1), then you must implement linear discriminant analysis, otherwise the software throws an error. Consequently, the software sets `DiscrimType` to 'linear'.
- If you specify 1, then the software uses maximum regularization for covariance matrix estimation. That is, the software restricts the covariance matrix to be diagonal. Alternatively, you can set `DiscrimType` to 'diagLinear' or 'diagQuadratic' for diagonal covariance matrices.

Example: `'Gamma',1`

Data Types: `single | double`

### **SaveMemory** — Flag to save covariance matrix

`'off'` (default) | `'on'`

Flag to save covariance matrix, specified as the comma-separated pair consisting of 'SaveMemory' and either 'on' or 'off'. If you specify 'on', then `fitcdiscr` does not store the full covariance matrix, but instead stores enough information to compute the matrix. The `predict` method computes the full covariance matrix for prediction, and does not store the matrix. If you specify 'off', then `fitcdiscr` computes and stores the full covariance matrix in `Mdl`.

Specify `SaveMemory` as 'on' when the input matrix contains thousands of predictors.

Example: `'SaveMemory','on'`

## Output Arguments

### **t** — Discriminant analysis classification template

classification template object

Discriminant analysis classification template suitable to use in the `fitcensemble` function, returned as a classification template object. In an ensemble, `t` specifies how to create the discriminant analysis classifier.

## Examples

### Discriminant Analysis Template for Nondefault Options

Create a nondefault discriminant analysis template for use in `fitensemble`.

Create a template for pseudolinear discriminant analysis.

```
t = ClassificationDiscriminant.template('discrimType','pseudoLinear')
```

```
t =
```

```
Fit template for classification Discriminant.
```

```
DiscrimType: 'pseudoLinear'  
Gamma: []  
Delta: []  
FillCoeffs: []  
SaveMemory: []  
Version: 1  
Method: 'Discriminant'  
Type: 'classification'
```

You can use `t` for ensemble learning.

## See Also

`ClassificationDiscriminant` | `fitcensemble` | `templateDiscriminant`



# ClassificationKNN.template

**Class:** ClassificationKNN

*k*-nearest neighbor classifier template for ensemble (to be removed)

---

**Note** ClassificationKNN.template will be removed in a future release. Use templateKNN instead.

---

## Syntax

```
t = ClassificationKNN.template()  
t = ClassificationKNN.template(Name, Value)
```

## Description

t = ClassificationKNN.template() returns a learner template suitable to use in the fitcensemble function.

t = ClassificationKNN.template(Name, Value) creates a template with additional options specified by one or more Name, Value pair arguments.

## Input Arguments

### Name-Value Pair Arguments

Specify optional comma-separated pairs of Name, Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as Name1, Value1, ..., NameN, ValueN.

#### **BreakTies** — Tie-breaking algorithm

'smallest' (default) | 'nearest' | 'random'

Tie-breaking algorithm used by the `predict` method if multiple classes have the same smallest cost, specified as the comma-separated pair consisting of `'BreakTies'` and one of the following:

- `'smallest'` — Use the smallest index among tied groups.
- `'nearest'` — Use the class with the nearest neighbor among tied groups.
- `'random'` — Use a random tiebreaker among tied groups.

By default, ties occur when multiple classes have the same number of nearest points among the  $K$  nearest neighbors.

Example: `'BreakTies', 'nearest'`

### **BucketSize** — Maximum data points in node

50 (default) | positive integer value

Maximum number of data points in the leaf node of the  $kd$ -tree, specified as the comma-separated pair consisting of `'BucketSize'` and a positive integer value. This argument is meaningful only when `NSMethod` is `'kdtree'`.

Example: `'BucketSize', 40`

Data Types: `single` | `double`

### **Cov** — Covariance matrix

`nancov(X)` (default) | positive definite matrix of scalar values

Covariance matrix, specified as the comma-separated pair consisting of `'Cov'` and a positive definite matrix of scalar values representing the covariance matrix when computing the Mahalanobis distance. This argument is only valid when `'Distance'` is `'mahalanobis'`.

You cannot simultaneously specify `'Standardize'` and either of `'Scale'` or `'Cov'`.

Data Types: `single` | `double`

### **Distance** — Distance metric

character vector | function handle

Distance metric, specified as the comma-separated pair consisting of `'Distance'` and a valid distance metric name or function handle. The allowable distance metric names depend on your choice of a neighbor-searcher method (see `NSMethod`).

NSMethod	Distance Metric Names
exhaustive	Any distance metric of ExhaustiveSearcher
kdtree	'cityblock', 'chebychev', 'euclidean', or 'minkowski'

This table includes valid distance metrics of ExhaustiveSearcher.

Distance Metric Names	Description
'cityblock'	City block distance.
'chebychev'	Chebychev distance (maximum coordinate difference).
'correlation'	One minus the sample linear correlation between observations (treated as sequences of values).
'cosine'	One minus the cosine of the included angle between observations (treated as vectors).
'euclidean'	Euclidean distance.
'hamming'	Hamming distance, percentage of coordinates that differ.
'jaccard'	One minus the Jaccard coefficient, the percentage of nonzero coordinates that differ.
'mahalanobis'	Mahalanobis distance, computed using a positive definite covariance matrix $C$ . The default value of $C$ is the sample covariance matrix of $X$ , as computed by <code>nancov(X)</code> . To specify a different value for $C$ , use the 'Cov' name-value pair argument.
'minkowski'	Minkowski distance. The default exponent is 2. To specify a different exponent, use the 'Exponent' name-value pair argument.
'seuclidean'	Standardized Euclidean distance. Each coordinate difference between $X$ and a query point is scaled, meaning divided by a scale value $S$ . The default value of $S$ is the standard deviation computed from $X$ , $S = \text{nanstd}(X)$ . To specify another value for $S$ , use the Scale name-value pair argument.
'spearman'	One minus the sample Spearman's rank correlation between observations (treated as sequences of values).

Distance Metric Names	Description
@ <i>distfun</i>	<p>Distance function handle. <i>distfun</i> has the form</p> <pre>function D2 = DISTFUN(ZI,ZJ) % calculation of distance ...</pre> <p>where</p> <ul style="list-style-type: none"> <li>• ZI is a 1-by-N vector containing one row of X or Y.</li> <li>• ZJ is an M2-by-N matrix containing multiple rows of X or Y.</li> <li>• D2 is an M2-by-1 vector of distances, and D2(k) is the distance between observations ZI and ZJ(k, :).</li> </ul>

If you specify `CategoricalPredictors` as 'all', then the default distance metric is 'hamming'. Otherwise, the default distance metric is 'euclidean'.

For definitions, see “Distance Metrics” on page 18-16.

Example: 'Distance', 'minkowski'

Data Types: `function_handle` | `char`

### DistanceWeight — Distance weighting function

'equal' (default) | 'inverse' | 'squaredinverse' | function handle

Distance weighting function, specified as the comma-separated pair consisting of 'DistanceWeight' and either a function handle or one of the values in this table.

Value	Description
'equal'	No weighting
'inverse'	Weight is 1/distance
'squaredinverse'	Weight is 1/distance <sup>2</sup>
@ <i>fcn</i>	<i>fcn</i> is a function that accepts a matrix of nonnegative distances, and returns a matrix the same size containing nonnegative distance weights. For example, 'squaredinverse' is equivalent to @ (d) d.^(-2).

Example: 'DistanceWeight', 'inverse'

Data Types: `function_handle` | `char`

### **Exponent** — Minkowski distance exponent

2 (default) | positive scalar value

Minkowski distance exponent, specified as the comma-separated pair consisting of 'Exponent' and a positive scalar value. This argument is only valid when 'Distance' is 'minkowski'.

Example: `'Exponent', 3`

Data Types: `single` | `double`

### **IncludeTies** — Tie inclusion flag

false (default) | true

Tie inclusion flag, specified as the comma-separated pair consisting of 'IncludeTies' and a logical value indicating whether `predict` includes all the neighbors whose distance values are equal to the Kth smallest distance. If `IncludeTies` is true, `predict` includes all these neighbors. Otherwise, `predict` uses exactly K neighbors.

Example: `'IncludeTies', true`

Data Types: `logical`

### **NSMethod** — Nearest neighbor search method

'kdtree' | 'exhaustive'

Nearest neighbor search method, specified as the comma-separated pair consisting of 'NSMethod' and 'kdtree' or 'exhaustive'.

- 'kdtree' — Create and use a *kd*-tree to find nearest neighbors. 'kdtree' is valid when the distance metric is one of the following:
  - 'euclidean'
  - 'cityblock'
  - 'minkowski'
  - 'chebychev'
- 'exhaustive' — Use the exhaustive search algorithm. The distance values from all points in X to each point in Y are computed to find nearest neighbors.

The default is 'kdtree' when  $X$  has 10 or fewer columns,  $X$  is not sparse, and the distance metric is a 'kdtree' type; otherwise, 'exhaustive'.

Example: 'NSMethod', 'exhaustive'

### **NumNeighbors** — Number of nearest neighbors to find

1 (default) | positive integer value

Number of nearest neighbors in  $X$  to find for classifying each point when predicting, specified as the comma-separated pair consisting of 'NumNeighbors' and a positive integer value.

Example: 'NumNeighbors', 3

Data Types: single | double

### **Scale** — Distance scale

nanstd( $X$ ) (default) | vector of nonnegative scalar values

Distance scale, specified as the comma-separated pair consisting of 'Scale' and a vector containing nonnegative scalar values with length equal to the number of columns in  $X$ . Each coordinate difference between  $X$  and a query point is scaled by the corresponding element of *Scale*. This argument is only valid when 'Distance' is 'seuclidean'.

You cannot simultaneously specify 'Standardize' and either of 'Scale' or 'Cov'.

Data Types: single | double

## Output Arguments

### **$\tau$** — Classification template

classification template object

$K$ -nearest neighbor classification template suitable to use in the `fitcensemble` function. In an ensemble,  $\tau$  specifies how to create the KNN classifier.

## Examples

## KNN template for nondefault options

Create a nondefault  $k$ -nearest neighbor template for use in `fitensemble`.

Create a template for 5-nearest neighbor search.

```
t = ClassificationKNN.template('NumNeighbors',5)
```

```
t =
```

```
Fit template for classification KNN.
```

```
    NumNeighbors: 5
      NSMethod: ''
      Distance: ''
      BucketSize: ''
    IncludeTies: []
  DistanceWeight: []
      BreakTies: []
      Exponent: []
          Cov: []
          Scale: []
StandardizeData: []
      Version: 1
      Method: 'KNN'
      Type: 'classification'
```

You can use `t` for ensemble learning.

- “Random Subspace Classification” on page 18-135

## See Also

`ClassificationKNN` | `fitensemble`

## Topics

“Random Subspace Classification” on page 18-135

## ClassificationTree.template

**Class:** ClassificationTree

Create classification template (to be removed)

---

**Note** ClassificationTree.template will be removed in a future release. Use templateTree instead.

---

### Syntax

```
t = ClassificationTree.template
t = ClassificationTree.template(Name, Value)
```

### Description

t = ClassificationTree.template returns a learner template suitable to use in the fitcensemble function.

t = ClassificationTree.template(Name, Value) creates a template with additional options specified by one or more Name, Value pair arguments. You can specify several name-value pair arguments in any order as Name1, Value1, ..., NameN, ValueN.

### Input Arguments

#### Name-Value Pair Arguments

Specify optional comma-separated pairs of Name, Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as Name1, Value1, ..., NameN, ValueN.



**AlgorithmForCategorical**

Algorithm to find the best split on a categorical predictor for data with  $K = 3$  or more classes. The available algorithms are:

'Exact'	For a categorical predictor with $C$ categories, consider all $2^{C-1} - 1$ combinations.
'PullLeft'	Start with all $C$ categories on the right branch. Consider moving each category to the left branch as it achieves the minimum impurity for the $K$ classes among the remaining categories. Out of this sequence, choose the split that has the lowest impurity.
'PCA'	Compute a score for each category using the inner product between the first principal component of a weighted covariance matrix (of the centered class probability matrix) and the vector of class probabilities for that category. Sort the scores in ascending order, and consider all $C - 1$ splits.
'OVAbyClass'	Start with all $C$ categories on the right branch. For each class, order the categories based on their probability for that class. For the first class, consider moving each category to the left branch in order, recording the impurity criterion at each move. Repeat for the remaining classes. Out of this sequence, choose the split that has the minimum impurity.

**Default:** ClassificationTree selects the optimal subset of algorithms for each split using the known number of classes and levels of a categorical predictor. For two classes, ClassificationTree always performs the exact search.

**MaxCat**

`ClassificationTree` splits a categorical predictor using the exact search algorithm if the predictor has at most `MaxCat` levels in the split node. Otherwise, `ClassificationTree` finds the best categorical split using one of the inexact algorithms.

Specify `MaxCat` as a numeric nonnegative scalar value. Passing a small value can lead to long computation time and memory overload.

**Default:** 10

**MergeLeaves**

Character vector that specifies whether to merge leaves after the tree is grown. Values are 'on' or 'off'.

When 'on', `ClassificationTree` merges leaves that originate from the same parent node, and that give a sum of risk values greater or equal to the risk associated with the parent node. When 'off', `ClassificationTree` does not merge leaves.

**Default:** 'off'

**MinLeaf**

Each leaf has at least `MinLeaf` observations per tree leaf. If you supply both `MinParent` and `MinLeaf`, `ClassificationTree` uses the setting that gives larger leaves:  $\text{MinParent} = \max(\text{MinParent}, 2 * \text{MinLeaf})$ .

**Default:** Half the number of training observations for boosting, 1 for bagging

**MinParent**

Each branch node in the tree has at least `MinParent` observations. If you supply both `MinParent` and `MinLeaf`, `ClassificationTree` uses the setting that gives larger leaves:  $\text{MinParent} = \max(\text{MinParent}, 2 * \text{MinLeaf})$ .

**Default:** Number of training observations for boosting, 2 for bagging

**NVarToSample**

Number of predictors to select at random for each split. Can be a positive integer or 'all', which means use all available predictors.

**Default:** 'all' for boosting, square root of number of predictors for bagging

**Prune**

When 'on', ClassificationTree grows the classification tree and computes the optimal sequence of pruned subtrees. When 'off' ClassificationTree grows the tree without pruning.

**Default:** 'off'

**PruneCriterion**

Character vector with the pruning criterion, either 'error' or 'impurity'.

**Default:** 'error'

**SplitCriterion**

Criterion for choosing a split. One of 'gdi' (Gini's diversity index), 'twoing' for the twoing rule, or 'deviance' for maximum deviance reduction (also known as cross entropy).

**Default:** 'gdi'

**Surrogate**

Character vector describing whether to find surrogate decision splits at each branch node. Specify as 'on', 'off', 'all', or a positive scalar value.

- When 'on', ClassificationTree finds at most 10 surrogate splits at each branch node.
- When set to a positive integer value, ClassificationTree finds at most the specified number of surrogate splits at each branch node.
- When set to 'all', ClassificationTree finds all surrogate splits at each branch node. The 'all' setting can use much time and memory.

Use surrogate splits to improve the accuracy of predictions for data with missing values. The setting also enables you to compute measures of predictive association between predictors.

**Default:** 'off'

## Output Arguments

`t`

Classification tree template suitable to use in the `fitcensemble` function. In an ensemble, `t` specifies how to grow the classification trees.

## Examples

### Construct a Classification Template with Surrogate Splits

Create a classification template with surrogate splits, and train an ensemble for the Fisher iris model with the template.

```
t = ClassificationTree.template('surrogate','on');
load fisheriris
ens = fitcensemble(meas,species,'AdaBoostM2',100,t);
```

How well does the ensemble classify the training data? This can be an underestimate of the loss on new data, but is nonetheless interesting.

```
loss = resubLoss(ens)
```

```
loss =
```

0.0333

## References

- [1] Coppersmith, D., S. J. Hong, and J. R. M. Hosking. “Partitioning Nominal Attributes in Decision Trees.” *Data Mining and Knowledge Discovery*, Vol. 3, 1999, pp. 197–217.

## See Also

`ClassificationTree` | `fitcensemble` | `fitctree` | `templateTree`

## RegressionTree.template

**Class:** RegressionTree

Create regression template (to be removed)

---

**Note** `RegressionTree.template` will be removed in a future release. Use `templateTree` instead.

---

### Syntax

```
t = RegressionTree.template
t = RegressionTree.template(Name, Value)
```

### Description

`t = RegressionTree.template` returns a learner template suitable to use in the `fitrensemble` function.

`t = RegressionTree.template(Name, Value)` creates a template with additional options specified by one or more `Name, Value` pair arguments. You can specify several name-value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### Input Arguments

#### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

**MergeLeaves**

Character vector that specifies whether to merge leaves after the tree is grown. Values are 'on' or 'off'.

When 'on', `RegressionTree` merges leaves that originate from the same parent node, and that give a sum of risk values greater or equal to the risk associated with the parent node. When 'off', `RegressionTree` does not merge leaves.

**Default:** 'off'

**MinLeaf**

Each leaf has at least `MinLeaf` observations per tree leaf. If you supply both `MinParent` and `MinLeaf`, `RegressionTree` uses the setting that gives larger leaves:  
 $\text{MinParent} = \max(\text{MinParent}, 2 * \text{MinLeaf})$ .

**Default:** Half the number of training observations for boosting, 5 for bagging

**MinParent**

Each branch node in the tree has at least `MinParent` observations. If you supply both `MinParent` and `MinLeaf`, `RegressionTree` uses the setting that gives larger leaves:  
 $\text{MinParent} = \max(\text{MinParent}, 2 * \text{MinLeaf})$ .

**Default:** Number of training observations for boosting, 10 for bagging

**NVarToSample**

Number of predictors to select at random for each split. Can be a positive integer or 'all', which means use all available predictors.

**Default:** 'all' for boosting, one third of the number of predictors for bagging

**Prune**

When 'on', `RegressionTree` grows the regression tree and computes the optimal sequence of pruned subtrees. When 'off' `RegressionTree` grows the tree without pruning.

**Default:** 'off'

### Surrogate

Character vector describing whether to find surrogate decision splits at each branch node. Specify as 'on', 'off', 'all', or a positive scalar value.

- When 'on', `RegressionTree` finds at most 10 surrogate splits at each branch node.
- When set to a positive integer value, `RegressionTree` finds at most the specified number of surrogate splits at each branch node.
- When set to 'all', `RegressionTree` finds all surrogate splits at each branch node. The 'all' setting can use much time and memory.

Use surrogate splits to improve the accuracy of predictions for data with missing values. The setting also enables you to compute measures of predictive association between predictors.

**Default:** 'off'

## Output Arguments

**t**

Regression tree template suitable to use in the `fitrensemble` function. In an ensemble, `t` specifies how to grow the regression trees.

## Examples

### Create a Regression Template

Create a regression template with surrogate splits, and train an ensemble for the `carsmall` data with the template.

```
load carsmall
t = RegressionTree.template('surrogate','on');
X = [Acceleration Displacement Horsepower Weight];
Y = MPG;
ens = fitrensemble(X,Y,'Learners',t);
```



## See Also

RegressionTree | fitrensemble | fitrtree

## templateDiscriminant

Discriminant analysis classifier template

### Syntax

```
t = templateDiscriminant()  
t = templateDiscriminant(Name,Value)
```

### Description

`t = templateDiscriminant()` returns a discriminant analysis learner template suitable for training ensembles or error-correcting output code (ECOC) multiclass models.

If you specify a default template, then the software uses default values for all input arguments during training.

Specify `t` as a learner in `fitcensemble` or `fitcecoc`.

`t = templateDiscriminant(Name,Value)` creates a template with additional options specified by one or more name-value pair arguments.

For example, you can specify the discriminant type or the regularization parameter.

If you display `t` in the Command Window, then all options appear empty (`[]`), except those that you specify using name-value pair arguments. During training, the software uses default values for empty options.

### Examples

#### Create a Discriminant Analysis Template for Ensemble Learning

Create a nondefault discriminant analysis template for use in `fitensemble`.

Load Fisher's iris data set.

```
load fisheriris
```

Create a template for pseudolinear discriminant analysis.

```
t = templateDiscriminant('DiscrimType','pseudoLinear')
```

```
t =
```

```
Fit template for classification Discriminant.
```

```
DiscrimType: 'pseudoLinear'  
    Gamma: []  
    Delta: []  
  FillCoeffs: []  
  SaveMemory: []  
    Version: 1  
    Method: 'Discriminant'  
    Type: 'classification'
```

All properties of the template object are empty except for `DiscrimType`, `Method`, and `Type`. When trained on, the software fills in the empty properties with their respective default values.

Specify `t` as a weak learner for a classification ensemble.

```
Mdl = fitensemble(meas,species,'Subspace',100,t);
```

Display the in-sample (resubstitution) misclassification error.

```
L = resubLoss(Mdl)
```

```
L =
```

0.0400

## Input Arguments

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'DiscrimType', 'pseudoLinear', 'SaveMemory', 'on'` specifies a template for pseudolinear discriminant analysis that does not store the full covariance matrix.

#### **Delta** — Linear coefficient threshold

0 (default) | nonnegative scalar value

Linear coefficient threshold, specified as the comma-separated pair consisting of `'Delta'` and a nonnegative scalar value. If a coefficient of `Mdl` has magnitude smaller than `Delta`, `Mdl` sets this coefficient to 0, and you can eliminate the corresponding predictor from the model. Set `Delta` to a higher value to eliminate more predictors.

`Delta` must be 0 for quadratic discriminant models.

Data Types: `single` | `double`

#### **DiscrimType** — Discriminant type

`'linear'` (default) | `'quadratic'` | `'diaglinear'` | `'diagquadratic'` | `'pseudolinear'` | `'pseudoquadratic'`

Discriminant type, specified as the comma-separated pair consisting of `'DiscrimType'` and a character vector in this table.

Value	Description	Predictor Covariance Treatment
'linear'	Regularized linear discriminant analysis (LDA)	<ul style="list-style-type: none"> <li>All classes have the same covariance matrix.</li> <li><math>\hat{\Sigma}_\gamma = (1 - \gamma)\hat{\Sigma} + \gamma \text{diag}(\hat{\Sigma})</math>.</li> </ul> <p><math>\hat{\Sigma}</math> is the empirical, pooled covariance matrix and <math>\gamma</math> is the amount of regularization.</p>
'diaglinear'	LDA	All classes have the same, diagonal covariance matrix.
'pseudolinear'	LDA	All classes have the same covariance matrix. The software inverts the covariance matrix using the pseudo inverse.
'quadratic'	Quadratic discriminant analysis (QDA)	The covariance matrices can vary among classes.
'diagquadratic'	QDA	The covariance matrices are diagonal and can vary among classes.
'pseudoquadratic'	QDA	The covariance matrices can vary among classes. The software inverts the covariance matrix using the pseudo inverse.

---

**Note** To use regularization, you must specify 'linear'. To specify the amount of regularization, use the Gamma name-value pair argument.

---

Example: 'DiscrimType', 'quadratic'

**FillCoeffs** — Coeffs property flag

'on' | 'off'

`Coeffs` property flag, specified as the comma-separated pair consisting of `'FillCoeffs'` and `'on'` or `'off'`. Setting the flag to `'on'` populates the `Coeffs` property in the classifier object. This can be computationally intensive, especially when cross validating. The default is `'on'`, unless you specify a cross validation name-value pair, in which case the flag is set to `'off'` by default.

Example: `'FillCoeffs', 'off'`

### **Gamma** — Amount of regularization

scalar value in the interval [0,1]

Amount of regularization to apply when estimating the covariance matrix of the predictors, specified as the comma-separated pair consisting of `'Gamma'` and a scalar value in the interval [0,1]. `Gamma` provides finer control over the covariance matrix structure than `DiscrimType`.

- If you specify 0, then the software does not use regularization to adjust the covariance matrix. That is, the software estimates and uses the unrestricted, empirical covariance matrix.
  - For linear discriminant analysis, if the empirical covariance matrix is singular, then the software automatically applies the minimal regularization required to invert the covariance matrix. You can display the chosen regularization amount by entering `Mdl.Gamma` at the command line.
  - For quadratic discriminant analysis, if at least one class has an empirical covariance matrix that is singular, then the software throws an error.
- If you specify a value in the interval (0,1), then you must implement linear discriminant analysis, otherwise the software throws an error. Consequently, the software sets `DiscrimType` to `'linear'`.
- If you specify 1, then the software uses maximum regularization for covariance matrix estimation. That is, the software restricts the covariance matrix to be diagonal. Alternatively, you can set `DiscrimType` to `'diagLinear'` or `'diagQuadratic'` for diagonal covariance matrices.

Example: `'Gamma', 1`

Data Types: `single` | `double`

### **SaveMemory** — Flag to save covariance matrix

`'off'` (default) | `'on'`

Flag to save covariance matrix, specified as the comma-separated pair consisting of 'SaveMemory' and either 'on' or 'off'. If you specify 'on', then `fitcdiscr` does not store the full covariance matrix, but instead stores enough information to compute the matrix. The `predict` method computes the full covariance matrix for prediction, and does not store the matrix. If you specify 'off', then `fitcdiscr` computes and stores the full covariance matrix in `Mdl`.

Specify `SaveMemory` as 'on' when the input matrix contains thousands of predictors.

Example: 'SaveMemory', 'on'

## Output Arguments

**t** — Discriminant analysis classification template

template object

Discriminant analysis classification template suitable for training ensembles or error-correcting output code (ECOC) multiclass models, returned as a template object. Pass `t` to `fitcensemble` or `fitcecoc` to specify how to create the discriminant analysis classifier for the ensemble or ECOC model, respectively.

If you display `t` to the Command Window, then all unspecified options appear empty (`[]`). However, the software replaces empty options with their corresponding default values during training.

## See Also

`ClassificationDiscriminant` | `fitcecoc` | `fitcensemble` | `predict`

Introduced in R2014a

## templateECOC

Error-correcting output codes learner template

### Syntax

```
t = templateECOC()  
t = templateECOC(Name,Value)
```

### Description

`t = templateECOC()` returns an error-correcting output codes (ECOC) classification learner template.

If you specify a default template, then the software uses default values for all input arguments during training.

`t = templateECOC(Name,Value)` returns a template with additional options specified by one or more name-value pair arguments.

For example, you can specify a coding design, whether to fit posterior probabilities, or the types of binary learners.

If you display `t` in the Command Window, then all options appear empty (`[]`), except those that you specify using name-value pair arguments. During training, the software uses default values for empty options.

### Examples

#### Create a Default ECOC Classification Learner Template

Use `templateECOC` to create a default ECOC template.

```
t = templateECOC()
```



```

t =

Fit template for classification ECOC.

    BinaryLearners: ''
           Coding: ''
    FitPosterior: []
           Options: []
    VerbosityLevel: []
           Version: 1
           Method: 'ECOC'
           Type: 'classification'

```

All properties of the template object are empty except for `Method` and `Type`. When you pass `t` to `testckfold`, the software fills in the empty properties with their respective default values. For example, the software fills the `BinaryLearners` property with 'SVM'. For details on other default values, see `fitcecoc`.

`t` is a plan for an ECOC learner. When you create it, no computation occurs. You can pass `t` to `testckfold` to specify a plan for an ECOC classification model to statistically compare with another model.

### Statistically Compare Performance of Two ECOC Classification Models

One way to select predictors or features is to train two models where one that uses a subset of the predictors that trained the other. Statistically compare the predictive performances of the models. If there is sufficient evidence that model trained on fewer predictors performs better than the model trained using more of the predictors, then you can proceed with a more efficient model.

Load Fisher's iris data set. Plot all 2-dimensional combinations of predictors.

```

load fisheriris
d = size(meas,2); % Number of predictors
pairs = combnk(1:d,2);

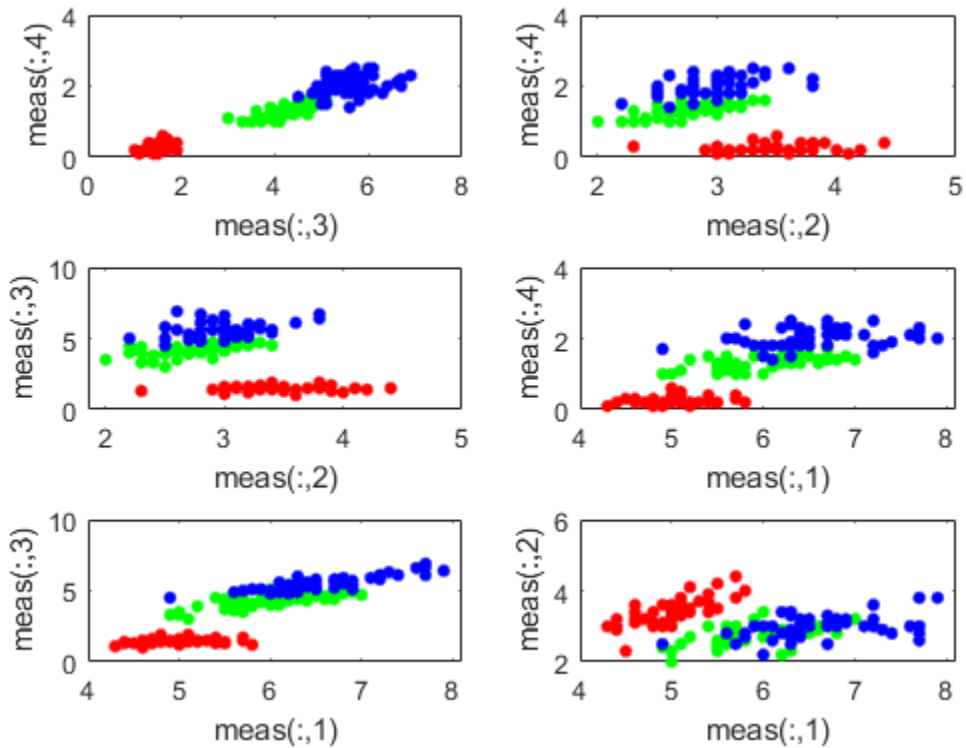
figure;
for j = 1:size(pairs,1);
    subplot(3,2,j);

```

```

gscatter(meas(:,pairs(j,1)),meas(:,pairs(j,2)),species);
xlabel(sprintf('meas(:,%d)',pairs(j,1)));
ylabel(sprintf('meas(:,%d)',pairs(j,2)));
legend off;
end

```



Based on the scatterplot, `meas(:,3)` and `meas(:,4)` seem like they separate the groups well.

Create an ECOC template. Specify to use a one-versus-all coding design.

```
t = templateECOC('Coding','onevsall');
```

By default, the ECOC model uses linear SVM binary learners. You can choose other, supported algorithms by specifying them using the 'Learners' name-value pair argument.

Test whether an ECOC model that is just trained using predictors 3 and 4 performs at most as well as an ECOC model that is trained using all predictors. Rejecting this null hypothesis means that the ECOC model trained using predictors 3 and 4 performs better than the ECOC model trained using all predictors. Suppose  $C_1$  represents the classification error of the ECOC model trained using predictors 3 and 4 and  $C_2$  represents the classification error of the ECOC model trained using all predictors, then the test is:

$$H_0 : C_1 \geq C_2$$

$$H_1 : C_1 < C_2$$

By default, `testckfold` conducts a 5-by-2  $k$ -fold  $F$  test, which is not appropriate as a one-tailed test. Specify to conduct a 5-by-2  $k$ -fold  $t$  test.

```
rng(1); % For reproducibility
[h,pValue] = testckfold(t,t,meas(:,pairs(1,:)),meas,species,...
    'Alternative','greater','Test','5x2t')
```

```
h =
```

```
    logical
```

```
    0
```

```
pValue =
```

```
    0.8940
```

The  $h = 0$  indicates that there is not enough evidence to suggest that the model trained using predictors 3 and 4 is more accurate than the model trained using all predictors.

## Input Arguments

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example:

`'Coding', 'ternarycomplete', 'FitPosterior', true, 'Learners', 'tree'` specifies a ternary complete coding design, to transform scores to posterior probabilities, and to grow classification trees for all binary learners.

#### Coding — Coding design

`'onevsone' (default) | 'allpairs' | 'binarycomplete' | 'denserandom' | 'onevsall' | 'ordinal' | 'sparserandom' | 'ternarycomplete' | numeric matrix`

Coding design name, specified as the comma-separated pair consisting of `'Coding'` and a numeric matrix or a value in this table.

Value	Number of Binary Learners	Description
<code>'allpairs'</code> and <code>'onevsone'</code>	$K(K-1)/2$	For each binary learner, one class is positive, another is negative, and the software ignores the rest. This design exhausts all combinations of class pair assignments.

Value	Number of Binary Learners	Description
'binarycomplete'	$2^{(K-1)} - 1$	This design partitions the classes into all binary combinations, and does not ignore any classes. For each binary learner, all class assignments are -1 and 1 with at least one positive and negative class in the assignment.
'denserandom'	Random, but approximately $10 \log_2 K$	For each binary learner, the software randomly assigns classes into positive or negative classes, with at least one of each type. For more details, see “Random Coding Design Matrices” on page 32-1923.
'onevsall'	$K$	For each binary learner, one class is positive and the rest are negative. This design exhausts all combinations of positive class assignments.
'ordinal'	$K - 1$	For the first binary learner, the first class is negative, and the rest positive. For the second binary learner, the first two classes are negative, the rest positive, and so on.

Value	Number of Binary Learners	Description
'sparsestrandom'	Random, but approximately $15 \log_2 K$	For each binary learner, the software randomly assigns classes as positive or negative with probability 0.25 for each, and ignores classes with probability 0.5. For more details, see “Random Coding Design Matrices” on page 32-1923.
'ternarycomplete'	$(3^K - 2^{(K+1)} + 1) / 2$	This design partitions the classes into all ternary combinations. All class assignments are 0, -1, and 1 with at least one positive and one negative class in the assignment.

You can also specify a coding design using a custom coding matrix. The custom coding matrix is a  $K$ -by- $L$  matrix. Each row corresponds to a class and each column corresponds to a binary learner. The class order (rows) corresponds to the order in `ClassNames`. Compose the matrix by following these guidelines:

- Every element of the custom coding matrix must be -1, 0, or 1, and the value must correspond to a dichotomous class assignment. This table describes the meaning of `Coding(i, j)`, that is, the class that learner  $j$  assigns to observations in class  $i$ .

Value	Dichotomous Class Assignment
-1	Negative class
0	Before training, learner $j$ removes observations in class $i$ from the data set.
1	Positive class

- Every column must contain at least one -1 or 1.
- For all column indices  $i, j$  such that  $i \neq j$ , `Coding(:, i)` cannot equal `Coding(:, j)` and `Coding(:, i)` cannot equal `-Coding(:, j)`.
- All rows of the custom coding matrix must be different.

For more details on the form of custom coding design matrices, see “Custom Coding Design Matrices” on page 32-1921.

Example: `'Coding', 'ternarycomplete'`

Data Types: `char | double | single | int16 | int32 | int64 | int8`

### **FitPosterior** — Flag indicating whether to transform scores to posterior probabilities

`false` or 0 (default) | `true` or 1

Flag indicating whether to transform scores to posterior probabilities, specified as the comma-separated pair consisting of `'FitPosterior'` and a `true` (1) or `false` (0).

If `FitPosterior` is `true`, then the software transforms binary-learner classification scores to posterior probabilities. You can obtain posterior probabilities by using `kfoldPredict`, `predict`, or `resubPredict`.

`fitcecoc` does not support fitting posterior probabilities:

- For the ensemble methods `AdaBoostM2`, `LPBoost`, `RUSBoost`, `RobustBoost`, and `TotalBoost`.
- If the binary learners (`Learners`) are linear classification models that implement SVM. To obtain posterior probabilities for linear classification models, implement logistic regression instead.

Example: `'FitPosterior', true`

Data Types: `logical`

### **Learners** — Binary learner templates

`'svm'` (default) | `'discriminant'` | `'knn'` | `'linear'` | `'naivebayes'` | `'tree'` | template object | cell vector of template objects

Binary learner templates, specified as the comma-separated pair consisting of `'Learners'` and a character vector, template object, or cell vector of template objects. Specifically, you can specify binary classifiers such as SVM, and the ensembles that use `GentleBoost`, `LogitBoost`, and `RobustBoost`, to solve multiclass problems. However, `fitcecoc` also supports multiclass models as binary classifiers.

- If `Learners` is a character vector, then the software trains each binary learner using the default values of the specified algorithm. This table summarizes the available algorithms.

Value	Description
'discriminant'	Discriminant analysis. For default options, see <code>templateDiscriminant</code> .
'knn'	$k$ -nearest neighbors. For default options, see <code>templateKNN</code> .
'linear'	Linear classification model. For default options, see <code>templateLinear</code> .
'naivebayes'	Naive Bayes. For default options, see <code>templateNaiveBayes</code> .
'svm'	SVM. For default options, see <code>templateSVM</code> .
'tree'	Classification trees. For default options, see <code>templateTree</code> .

- If `Learners` is a template object, then each binary learner trains according to the stored options. You can create a template object using:
  - `templateDiscriminant`, for discriminant analysis.
  - `templateEnsemble`, for ensemble learning. You must at least specify the learning method (`Method`), the number of learners (`NLearn`), and the type of learner (`Learners`). You cannot use the `AdaBoostM2` ensemble method for binary learning.
  - `templateKNN`, for  $k$ -nearest neighbors.
  - `templateLinear`, for linear classification.
  - `templateNaiveBayes`, for naive Bayes.
  - `templateSVM`, for SVM.
  - `templateTree`, for classification trees.
- If `Learners` is cell vector of template objects, then:
  - Cell  $j$  corresponds to binary learner  $j$  (in other words, column  $j$  of the coding design matrix), and the cell vector must have length  $L$ .  $L$  is the number of columns in the coding design matrix. For details, see `Coding`.
  - To use one of the built-in loss functions for prediction, then all binary learners must return a score in the same range. For example, you cannot include default SVM binary learners with default naive Bayes binary learners. The former returns



a score in the range  $(-\infty, \infty)$ , and the latter returns a posterior probability as a score. Otherwise, you must provide a custom loss as a function handle to functions such as `predict` and `loss`.

- You cannot specify linear classification model learner templates with any other template in the cell array.

By default, the software trains learners using default SVM templates.

Example: `'Learners', 'tree'`

Data Types: `char` | `cell`

## Output Arguments

**t** — ECOC classification template

template object

ECOC classification template, returned as a template object. Pass `t` to `testckfold` to specify how to create an ECOC classifier whose predictive performance you want to compare with another classifier.

If you display `t` to the Command Window, then all, unspecified options appear empty (`[]`). However, the software replaces empty options with their corresponding default values during training.

## See Also

`ClassificationECOC` | `designecoc` | `fitcecoc` | `predict` |  
`templateDiscriminant` | `templateEnsemble` | `templateKNN` | `templateSVM` |  
`templateTree` | `testckfold`

Introduced in R2015a

## templateEnsemble

Ensemble learning template

### Syntax

```
t = templateEnsemble(Method, NLearn, Learners)
t = templateEnsemble(Method, NLearn, Learners, Name, Value)
```

### Description

`t = templateEnsemble(Method, NLearn, Learners)` returns an ensemble learning template that specifies to use the ensemble-aggregation method `Method`, `NLearn` learning cycles, and weak learners `Learners`.

All other options of the template (`t`) specific to ensemble learning appear empty, but the software uses their corresponding default values during training.

`t = templateEnsemble(Method, NLearn, Learners, Name, Value)` returns an ensemble template with additional options specified by one or more name-value pair arguments.

For example, you can specify the number of predictors in each random subspace learner, learning rate for shrinkage, or the target classification error for `RobustBoost`.

If you display `t` in the Command Window, then all options appear empty (`[]`), except those options that you specify using name-value pair arguments. During training, the software uses default values for empty options.

### Examples

#### Create an Ensemble Learning Template

Use `templateEnsemble` to specify an ensemble learning template. You must specify the ensemble method, the number of learning cycles, and the type of weak learners. For this

example, specify the AdaBoostM1 method, 100 learners, and classification tree weak learners.

```
t = templateEnsemble('AdaBoostM1',100,'tree')
```

```
t =
```

```
Fit template for classification AdaBoostM1.
```

```

      Type: 'classification'
      Method: 'AdaBoostM1'
LearnerTemplates: 'Tree'
      NLearn: 100
      LearnRate: []

```

All properties of the template object are empty except for Method, Type, LearnerTemplates, and NLearn. When trained on, the software fills in the empty properties with their respective default values. For example, the software fills the LearnRate property with 1.

t is a plan for an ensemble learner, and no computation takes place when you specify it. You can pass t to fitcecoc to specify ensemble binary learners for ECOC multiclass learning.

## Create an Ensemble Template for ECOC Multiclass Learning

Create an ensemble template for use in fitcecoc.

Load the arrhythmia data set.

```
load arrhythmia
tabulate(categorical(Y));
rng(1); % For reproducibility
```

Value	Count	Percent
1	245	54.20%
2	44	9.73%
3	15	3.32%
4	15	3.32%
5	13	2.88%

6	25	5.53%
7	3	0.66%
8	2	0.44%
9	9	1.99%
10	50	11.06%
14	4	0.88%
15	5	1.11%
16	22	4.87%

Some classes have small relative frequencies in the data.

Create a template for a GentleBoost ensemble of classification trees, and specify to use a 100 learners and a shrinkage of 0.1. By default, boosting grows stumps (i.e., one node having a set of leaves). Since there are classes with small frequencies, the trees must be leafy enough to be sensitive to the minority classes. Specify the minimum number of leaf node observations to 3.

```
tTree = templateTree('MinLeafSize',20);  
t = templateEnsemble('AdaBoostM1',100,tTree,'LearnRate',0.1);
```

All properties of the template objects are empty except for `Method` and `Type`, and the corresponding properties of the name-value pair argument values in the function calls. When you pass `t` to the training function, the software fills in the empty properties with their respective default values.

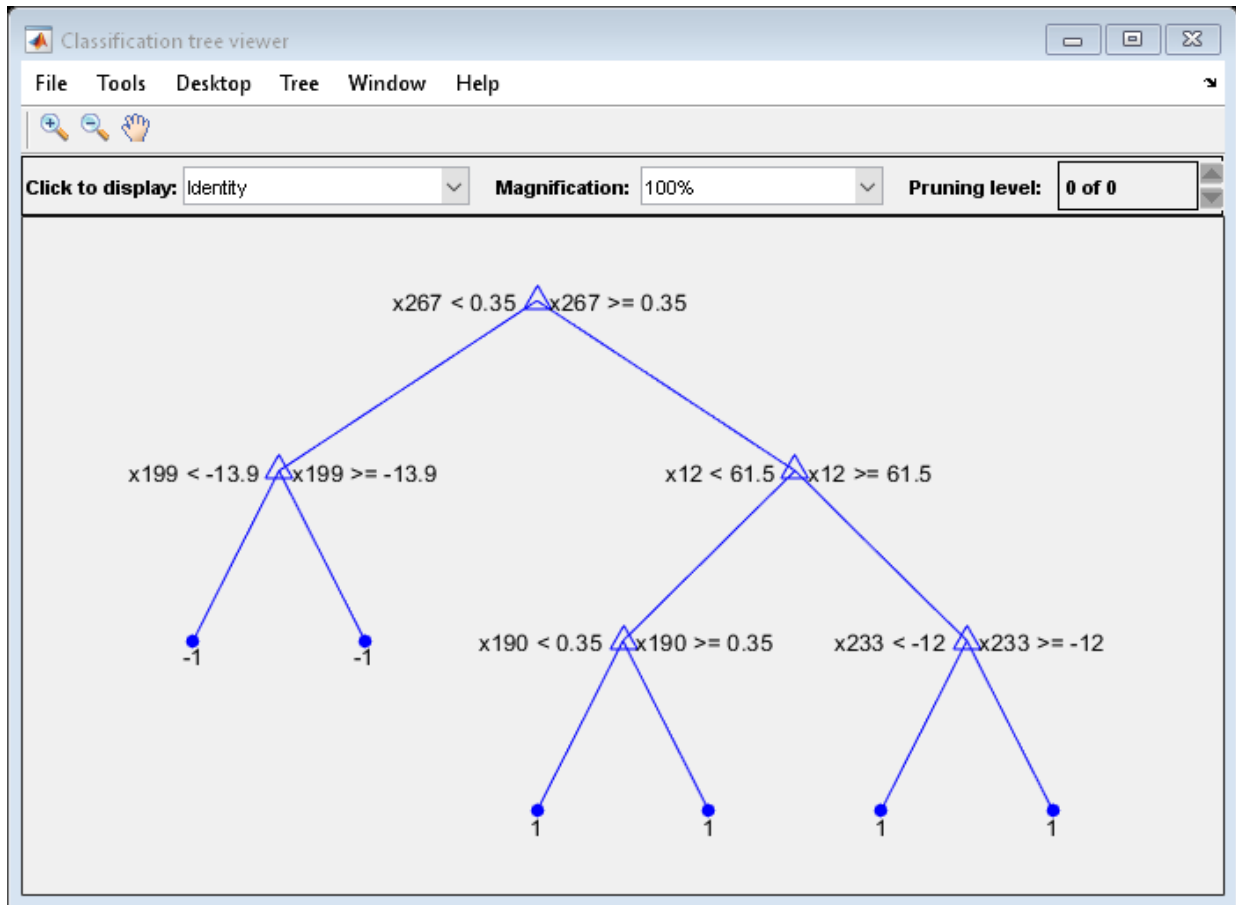
Specify `t` as a binary learner for an ECOC multiclass model. Train using the default one-versus-one coding design.

```
Mdl = fitcecoc(X,Y,'Learners',t);
```

- `Mdl` is a `ClassificationECOC` multiclass model.
- `Mdl.BinaryLearners` is a 78-by-1 cell array of `CompactClassificationEnsemble` models.
- `Mdl.BinaryLearners{j}.Trained` is a 100-by-1 cell array of `CompactClassificationTree` models, for  $j = 1, \dots, 78$ .

You can verify that one of the binary learners contains a weak learner that isn't a stump by using `view`.

```
view(Mdl.BinaryLearners{1}.Trained{1},'Mode','graph')
```



Display the in-sample (resubstitution) misclassification error.

```
L = resubLoss(Mdl, 'LossFun', 'classiferror')
```

```
L =
```

```
0.0597
```

- “Supervised Learning Workflow and Algorithms” on page 18-2

- “Framework for Ensemble Learning” on page 18-41

## Input Arguments

### **Method — Ensemble-aggregation method**

character vector

Ensemble-aggregation method, specified as one of the method names in this list.

- For classification with two classes:
  - 'AdaBoostM1'
  - 'LogitBoost'
  - 'GentleBoost'
  - 'RobustBoost' (requires Optimization Toolbox)
  - 'LPBoost' (requires Optimization Toolbox)
  - 'TotalBoost' (requires Optimization Toolbox)
  - 'RUSBoost'
  - 'Subspace'
  - 'Bag'
- For classification with three or more classes:
  - 'AdaBoostM2'
  - 'LPBoost' (requires Optimization Toolbox)
  - 'TotalBoost' (requires Optimization Toolbox)
  - 'RUSBoost'
  - 'Subspace'
  - 'Bag'
- For regression:
  - 'LSBoost'
  - 'Bag'

Because you can specify 'Bag' for classification and regression problems, specify the problem type using the `Type` name-value pair argument.

Data Types: char

### **NLearn** — Number of ensemble learning cycles

positive integer | 'AllPredictorCombinations'

Number of ensemble learning cycles, specified as a positive integer or 'AllPredictorCombinations'.

- If you specify a positive integer, then, at every learning cycle, the software trains one weak learner for every template object in `Learners`. Consequently, the software trains  $NLearn * \text{numel}(Learners)$  learners.
- If you specify 'AllPredictorCombinations', then set `Method` to 'Subspace' and specify one learner only in `Learners`. With these settings, the software trains learners for all possible combinations of predictors taken `NPredToSample` at a time. Consequently, the software trains  $nchoosek(\text{size}(X, 2), NPredToSample)$  learners.

For more details, see “Tips” on page 32-6415.

### **Learners** — Weak learners to use in ensemble

weak-learner name | weak-learner template object | cell vector of weak-learner template objects

Weak learners to use in the ensemble, specified as a weak-learner name, weak-learner template object, or cell array of weak-learner template objects.

Weak Learner	Weak-Learner Name	Template Object Creation Function	Method Settings
Discriminant analysis	'Discriminant'	templateDiscriminant	Recommended for 'Subspace'
$k$ nearest neighbors	'KNN'	templateKNN	For 'Subspace' only
Decision tree	'Tree'	templateTree	All methods except 'Subspace'

For more details, see `NLearn` and “Tips” on page 32-6415.

Example: For an ensemble composed of two types of classification trees, supply `{t1 t2}`, where `t1` and `t2` are classification tree templates.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

Example: `'LearningRate', 0.05, 'NPrint', 5` specifies to use 0.05 as the learning rate and to display a message to the command line every time it trains 5 learners.

### General Ensemble Options

#### **NPrint** — Printout frequency

`'off'` (default) | positive integer

Printout frequency, specified as the comma-separated pair consisting of `'NPrint'` and a positive integer or `'off'`.

To track the number of *weak learners* or *folds* that the software trained so far, specify a positive integer. That is, if you specify the positive integer  $m$  and:

- Do not specify any cross-validation option of the fitting function (for example, `CrossVal`), then the software displays a message to the command line every time it completes training  $m$  weak learners.
- A cross-validation option, then the software displays a message to the command line every time it finishes training  $m$  folds.

If you specify `'off'`, then the software does not display a message when it completes training weak learners.

---

**Tip** When training an ensemble of many weak learners on a large data set, specify a positive integer for `NPrint`.

---

Example: `'NPrint', 5`

Data Types: `single` | `double` | `char`

#### **Type** — Supervised learning type

`'classification'` | `'regression'`



Supervised learning type, specified as the comma-separated pair consisting of 'Type' and 'classification' or 'regression'.

- If Method is 'bag', then the supervised learning type is ambiguous. Therefore, specify Type when bagging.
- Otherwise, the value of Method determines the supervised learning type.

Example: 'Type', 'classification'

Data Types: char

### Sampling Options for Boosting Methods and Bagging

#### **FResample** — Fraction of training set to resample

1 (default) | positive scalar in (0,1]

Fraction of the training set to resample for every weak learner, specified as the comma-separated pair consisting of 'FResample' and a positive scalar in (0,1].

To use 'FResample', specify 'bag' for Method or set Resample to 'on'.

Example: 'FResample', 0.75

Data Types: single | double

#### **Replace** — Flag indicating to sample with replacement

'on' (default) | 'off'

Flag indicating sampling with replacement, specified as the comma-separated pair consisting of 'Replace' and 'off' or 'on'.

- For 'on', the software samples the training observations with replacement.
- For 'off', the software samples the training observations without replacement. If you set Resample to 'on', then the software samples training observations assuming uniform weights. If you also specify a boosting method, then the software boosts by reweighting observations.

Unless you set Method to 'bag' or set Resample to 'on', Replace has no effect.

Example: 'Replace', 'off'

Data Types: char

**Resample** — Flag indicating to resample`'off' | 'on'`

Flag indicating to resample, specified as the comma-separated pair consisting of 'Resample' and 'off' or 'on'.

- If `Method` is any boosting method, then:
  - 'Resample', 'on' specifies to sample training observations using updated weights as the multinomial sampling probabilities.
  - 'Resample', 'off' specifies to reweight observations at every learning iteration. This setting is the default.
- If `Method` is 'bag', then 'Resample' must be 'on'. The software resamples a fraction of the training observations (see `FResample`) with or without replacement (see `Replace`).

**AdaBoostM1, AdaBoostM2, LogitBoost, GentleBoost, and LSBoost Method Options****LearnRate** — Learning rate for shrinkage`1 (default) | numeric scalar in (0,1]`

Learning rate for shrinkage, specified as the comma-separated pair consisting of a numeric scalar in the interval (0,1].

To train an ensemble using shrinkage, set `LearnRate` to a value less than 1, for example, 0.1 is a popular choice. Training an ensemble using shrinkage requires more learning iterations, but often achieves better accuracy.

Example: `'LearnRate', 0.1`

Data Types: `single | double`

**RUSBoost Method Options****LearnRate** — Learning rate for shrinkage`1 (default) | numeric scalar in (0,1]`

Learning rate for shrinkage, specified as the comma-separated pair consisting of a numeric scalar in the interval (0,1].

To train an ensemble using shrinkage, set `LearnRate` to a value less than 1, for example, 0.1 is a popular choice. Training an ensemble using shrinkage requires more learning iterations, but often achieves better accuracy.

Example: `'LearnRate', 0.1`

Data Types: `single` | `double`

### **RatioToSmallest** — Sampling proportion with respect to lowest-represented class

positive numeric scalar | numeric vector of positive values

Sampling proportion with respect to the lowest-represented class, specified as the comma-separated pair consisting of `'RatioToSmallest'` and a numeric scalar or numeric vector of positive values with length equal to the number of distinct classes in the training data.

Suppose that there are  $K$  classes in the training data and the lowest-represented class has  $m$  observations in the training data.

- If you specify the positive numeric scalar  $s$ , then the software samples  $s*m$  observations from each class, that is, it uses the same sampling proportion for each class. For more details, see “Algorithms” on page 32-6417.
- If you specify the numeric vector  $[s_1, s_2, \dots, s_K]$ , then the software samples  $s_i*m$  observations from class  $i$ ,  $i = 1, \dots, K$ . The elements of `RatioToSmallest` correspond to the order of the class names specified using the `ClassNames` name-value pair argument of the fitting function (see “Tips” on page 32-6415).

The default value is `ones(K, 1)`, which specifies to sample  $m$  observations from each class.

Example: `'RatioToSmallest', [2, 1]`

Data Types: `single` | `double`

### **LPBoost and TotalBoost Method Options**

#### **MarginPrecision** — Margin precision to control convergence speed

0.1 (default) | numeric scalar in [0,1]

Margin precision to control convergence speed, specified as the comma-separated pair consisting of `'MarginPrecision'` and a numeric scalar in the interval [0,1].

`MarginPrecision` affects the number of boosting iterations required for convergence.

---

**Tip** To train an ensemble using many learners, specify a small value for `MarginPrecision`. For training using a few learners, specify a large value.

---

Example: `'MarginPrecision', 0.5`

Data Types: `single` | `double`

### **RobustBoost Method Options**

#### **RobustErrorGoal — Target classification error**

0.1 (default) | nonnegative numeric scalar

Target classification error, specified as the comma-separated pair consisting of `'RobustErrorGoal'` and a nonnegative numeric scalar. The upper bound on possible values depends on the values of `RobustMarginSigma` and `RobustMaxMargin`. However, the upper bound cannot exceed 1.

---

**Tip** For a particular training set, usually there is an optimal range for `RobustErrorGoal`. If you set it too low or too high, then the software can produce a model with poor classification accuracy. Try cross-validating to search for the appropriate value.

---

Example: `'RobustErrorGoal', 0.05`

Data Types: `single` | `double`

#### **RobustMarginSigma — Classification margin distribution spread**

0.1 (default) | positive numeric scalar

Classification margin distribution spread over the training data, specified as the comma-separated pair consisting of `'RobustMarginSigma'` and a positive numeric scalar. Before specifying `RobustMarginSigma`, consult the literature on `RobustBoost`, for example, [19].

Example: `'RobustMarginSigma', 0.5`

Data Types: `single` | `double`

#### **RobustMaxMargin — Maximal classification margin**

0 (default) | nonnegative numeric scalar

Maximal classification margin in the training data, specified as the comma-separated pair consisting of 'RobustMaxMargin' and a nonnegative numeric scalar. The software minimizes the number of observations in the training data having classification margins below RobustMaxMargin.

Example: 'RobustMaxMargin',1

Data Types: single | double

### Random Subspace Method Options

#### **NPredToSample** — Number of predictors to sample

1 (default) | positive integer

Number of predictors to sample for each random subspace learner, specified as the comma-separated pair consisting of 'NPredToSample' and a positive integer in the interval  $1, \dots, p$ , where  $p$  is the number of predictor variables (`size(X,2)` or `size(Tbl,2)`).

Data Types: single | double

## Output Arguments

#### **t** — Classification template for ensemble learning

classification template object

Classification template for ensemble learning, returned as a template object. You can pass `t` to, for example, `fitcecoc` to specify how to create the ensemble learning classifier for the ECOC model.

If you display `t` in the Command Window, then all, unspecified options appear empty (`[]`). However, the software replaces empty options with their corresponding default values during training.

## Tips

- `NLearn` can vary from a few dozen to a few thousand. Usually, an ensemble with good predictive power requires from a few hundred to a few thousand weak learners. However, you do not have to train an ensemble for that many cycles at once. You can

start by growing a few dozen learners, inspect the ensemble performance and then, if necessary, train more weak learners using `resume` for classification problems, or `resume` for regression problems.

- Ensemble performance depends on the ensemble setting and the setting of the weak learners. That is, if you specify weak learners with default parameters, then the ensemble can perform. Therefore, like ensemble settings, it is good practice to adjust the parameters of the weak learners using templates, and to choose values that minimize generalization error.
- If you specify to resample using `Resample`, then it is good practice to resample to entire data set. That is, use the default setting of 1 for `FResample`.
- In classification problems (that is, `Type` is 'classification'):
  - If the ensemble-aggregation method (`Method`) is 'bag' and:
    - The misclassification cost is highly imbalanced, then, for in-bag samples, the software oversamples unique observations from the class that has a large penalty.
    - The class prior probabilities are highly skewed, the software oversamples unique observations from the class that has a large prior probability.

For smaller sample sizes, these combinations can result in a very low relative frequency of out-of-bag observations from the class that has a large penalty or prior probability. Consequently, the estimated out-of-bag error is highly variable and it might be difficult to interpret. To avoid large estimated out-of-bag error variances, particularly for small sample sizes, set a more balanced misclassification cost matrix using the `Cost` name-value pair argument of the fitting function, or a less skewed prior probability vector using `Prior` name-value pair argument of the fitting function.

- Because the order of some input and output arguments correspond to the distinct classes in the training data, it is good practice to specify the class order using the `ClassNames` name-value pair argument of the fitting function.
  - To quickly determine the class order, remove all observations from the training data that are unclassified (that is, have a missing label), obtain and display an array of all the distinct classes, and then specify the array for `ClassNames`. For example, suppose the response variable (`Y`) is a cell array of labels. This code specifies the class order in the variable `classNames`.

```
Ycat = categorical(Y);  
classNames = categories(Ycat)
```

`categorical` assigns `<undefined>` to unclassified observations and `categories` excludes `<undefined>` from its output. Therefore, if you use this code for cell arrays of labels or similar code for categorical arrays, then you do not have to remove observations with missing labels to obtain a list of the distinct classes.

- To specify that order should be from lowest-represented label to most-represented, then quickly determine the class order (as in the previous bullet), but arrange the classes in the list by frequency before passing the list to `ClassNames`. Following from the previous example, this code specifies the class order from lowest- to most-represented in `classNamesLH`.

```
Ycat = categorical(Y);
classNames = categories(Ycat);
freq = countcats(Ycat);
[~,idx] = sort(freq);
classNamesLH = classNames(idx);
```

## Algorithms

- For details of ensemble-aggregation algorithms, see “Ensemble Algorithms” on page 18-51.
- If you specify `Method` to be a boosting algorithm and `Learners` to be decision trees, then the software grows stumps by default. A decision stump is one root node connected to two terminal, leaf nodes. You can adjust tree depth by specifying the `MaxNumSplits`, `MinLeafSize`, and `MinParentSize` name-value pair arguments using `templateTree`.
- The software generates in-bag samples by oversampling classes with large misclassification costs and undersampling classes with small misclassification costs. Consequently, out-of-bag samples have fewer observations from classes with large misclassification costs and more observations from classes with small misclassification costs. If you train a classification ensemble using a small data set and a highly skewed cost matrix, then the number of out-of-bag observations per class might be very low. Therefore, the estimated out-of-bag error might have a large variance and might be difficult to interpret. The same phenomenon can occur for classes with large prior probabilities.
- For the RUSBoost ensemble-aggregation method (`Method`), the name-value pair argument `RatioToSmallest` specifies the sampling proportion for each class with respect to the lowest-represented class. For example, suppose that there are 2 classes

in the training data,  $A$  and  $B$ .  $A$  have 100 observations and  $B$  have 10 observations. and that the lowest-represented class has  $m$  observations in the training data.

- If you set `'RatioToSmallest', 2`, then  $s*m = 2*10 = 20$ . Consequently, the software trains every learner using 20 observations from class  $A$  and 20 observations from class  $B$ . If you set `'RatioToSmallest', [2 2]`, then you will obtain the same result.
- If you set `'RatioToSmallest', [2, 1]`, then  $s1*m = 2*10 = 20$  and  $s2*m = 1*10 = 10$ . Consequently, the software trains every learner using 20 observations from class  $A$  and 10 observations from class  $B$ .
- For ensembles of decision trees, and for dual-core systems and above, `fitcensemble` and `fitrensemble` parallelize training using Intel Threading Building Blocks (TBB). For details on Intel TBB, see <https://software.intel.com/en-us/intel-tbb>.

## See Also

`ClassificationECOC` | `ClassificationEnsemble` | `fitcecoc` | `fitcensemble` | `fitrensemble` | `templateDiscriminant` | `templateKNN` | `templateTree`

## Topics

“Supervised Learning Workflow and Algorithms” on page 18-2

“Framework for Ensemble Learning” on page 18-41

**Introduced in R2014b**



# templateKNN

*k*-nearest neighbor classifier template

## Syntax

```
t = templateKNN()  
t = templateKNN(Name,Value)
```

## Description

`t = templateKNN()` returns a *k*-nearest neighbor (KNN) learner template suitable for training ensembles or error-correcting output code (ECOC) multiclass models.

If you specify a default template, then the software uses default values for all input arguments during training.

Specify `t` as a learner in `fitcensemble` or `fitcecoc`.

`t = templateKNN(Name,Value)` creates a template with additional options specified by one or more name-value pair arguments.

For example, you can specify the nearest neighbor search method, the number of nearest neighbors to find, or the distance metric.

If you display `t` in the Command Window, then all options appear empty (`[]`), except those that you specify using name-value pair arguments. During training, the software uses default values for empty options.

## Examples

### Create a *k*-Nearest Neighbors Template for Ensemble

Create a nondefault *k*-nearest neighbor template for use in `fitensemble`.

Load Fisher's iris data set.

```
load fisheriris
```

Create a template for a 5-nearest neighbor search, and specify to standardize the predictors.

```
t = templateKNN('NumNeighbors',5,'Standardize',1)
```

```
t =
```

Fit template for classification KNN.

```
    NumNeighbors: 5
      NSMethod: ''
      Distance: ''
      BucketSize: ''
      IncludeTies: []
      DistanceWeight: []
      BreakTies: []
      Exponent: []
          Cov: []
          Scale: []
StandardizeData: 1
      Version: 1
      Method: 'KNN'
      Type: 'classification'
```

All properties of the template object are empty except for `NumNeighbors`, `Method`, `StandardizeData`, and `Type`. When you specify `t` as a learner, the software fills in the empty properties with their respective default values.

Specify `t` as a weak learner for a classification ensemble.

```
Mdl = fitensemble(meas,species,'Subspace',100,t);
```

Display the in-sample (resubstitution) misclassification error.

```
L = resubLoss(Mdl)
```

```
L =
```

```
0.0600
```

### Create a $k$ -Nearest Neighbors Template for ECOC Multiclass Learning

Create a nondefault  $k$ -nearest neighbor template for use in `fitcecoc`.

Load Fisher's iris data set.

```
load fisheriris
```

Create a template for a 5-nearest neighbor search, and specify to standardize the predictors.

```
t = templateKNN('NumNeighbors',5,'Standardize',1)
```

```
t =
```

```
Fit template for classification KNN.
```

```

    NumNeighbors: 5
      NSMethod: ''
      Distance: ''
    BucketSize: ''
  IncludeTies: []
DistanceWeight: []
    BreakTies: []
      Exponent: []
          Cov: []
          Scale: []
StandardizeData: 1
      Version: 1
      Method: 'KNN'
      Type: 'classification'

```

All properties of the template object are empty except for `NumNeighbors`, `Method`, `StandardizeData`, and `Type`. When you specify `t` as a learner, the software fills in the empty properties with their respective default values.

Specify `t` as a binary learner for an ECOC multiclass model.

```
Mdl = fitcecoc(meas,species,'Learners',t);
```

By default, the software trains `Mdl` using the one-versus-one coding design.

Display the in-sample (resubstitution) misclassification error.

```
L = resubLoss(Mdl,'LossFun','classiferror')
```

```
L =
```

```
    0.0467
```

- “Random Subspace Classification” on page 18-135

## Input Arguments

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'NumNeighbors',4,'Distance','minkowski'` specifies a 4-nearest neighbor classifier template using the Minkowski distance measure.

#### **BreakTies** — Tie-breaking algorithm

```
'smallest' (default) | 'nearest' | 'random'
```

Tie-breaking algorithm used by the `predict` method if multiple classes have the same smallest cost, specified as the comma-separated pair consisting of `'BreakTies'` and one of the following:

- `'smallest'` — Use the smallest index among tied groups.
- `'nearest'` — Use the class with the nearest neighbor among tied groups.
- `'random'` — Use a random tiebreaker among tied groups.

By default, ties occur when multiple classes have the same number of nearest points among the  $K$  nearest neighbors.

Example: 'BreakTies', 'nearest'

### **BucketSize** — Maximum data points in node

50 (default) | positive integer value

Maximum number of data points in the leaf node of the  $kd$ -tree, specified as the comma-separated pair consisting of 'BucketSize' and a positive integer value. This argument is meaningful only when NSMethod is 'kdtree'.

Example: 'BucketSize', 40

Data Types: single | double

### **Cov** — Covariance matrix

nancov(X) (default) | positive definite matrix of scalar values

Covariance matrix, specified as the comma-separated pair consisting of 'Cov' and a positive definite matrix of scalar values representing the covariance matrix when computing the Mahalanobis distance. This argument is only valid when 'Distance' is 'mahalanobis'.

You cannot simultaneously specify 'Standardize' and either of 'Scale' or 'Cov'.

Data Types: single | double

### **Distance** — Distance metric

character vector | function handle

Distance metric, specified as the comma-separated pair consisting of 'Distance' and a valid distance metric name or function handle. The allowable distance metric names depend on your choice of a neighbor-searcher method (see NSMethod).

NSMethod	Distance Metric Names
exhaustive	Any distance metric of ExhaustiveSearcher
kdtree	'cityblock', 'chebychev', 'euclidean', or 'minkowski'

This table includes valid distance metrics of ExhaustiveSearcher.

Distance Metric Names	Description
'cityblock'	City block distance.
'chebychev'	Chebychev distance (maximum coordinate difference).
'correlation'	One minus the sample linear correlation between observations (treated as sequences of values).
'cosine'	One minus the cosine of the included angle between observations (treated as vectors).
'euclidean'	Euclidean distance.
'hamming'	Hamming distance, percentage of coordinates that differ.
'jaccard'	One minus the Jaccard coefficient, the percentage of nonzero coordinates that differ.
'mahalanobis'	Mahalanobis distance, computed using a positive definite covariance matrix $C$ . The default value of $C$ is the sample covariance matrix of $X$ , as computed by <code>nancov(X)</code> . To specify a different value for $C$ , use the 'Cov' name-value pair argument.
'minkowski'	Minkowski distance. The default exponent is 2. To specify a different exponent, use the 'Exponent' name-value pair argument.
'seuclidean'	Standardized Euclidean distance. Each coordinate difference between $X$ and a query point is scaled, meaning divided by a scale value $S$ . The default value of $S$ is the standard deviation computed from $X$ , $S = \text{nanstd}(X)$ . To specify another value for $S$ , use the <code>Scale</code> name-value pair argument.
'spearman'	One minus the sample Spearman's rank correlation between observations (treated as sequences of values).

Distance Metric Names	Description
@ <i>distfun</i>	<p>Distance function handle. <i>distfun</i> has the form</p> <pre>function D2 = DISTFUN(ZI,ZJ) % calculation of distance ...</pre> <p>where</p> <ul style="list-style-type: none"> <li>• ZI is a 1-by-N vector containing one row of X or Y.</li> <li>• ZJ is an M2-by-N matrix containing multiple rows of X or Y.</li> <li>• D2 is an M2-by-1 vector of distances, and D2(k) is the distance between observations ZI and ZJ(k, :).</li> </ul>

If you specify `CategoricalPredictors` as 'all', then the default distance metric is 'hamming'. Otherwise, the default distance metric is 'euclidean'.

For definitions, see “Distance Metrics” on page 18-16.

Example: 'Distance', 'minkowski'

Data Types: `function_handle` | `char`

#### DistanceWeight — Distance weighting function

'equal' (default) | 'inverse' | 'squaredinverse' | function handle

Distance weighting function, specified as the comma-separated pair consisting of 'DistanceWeight' and either a function handle or one of the values in this table.

Value	Description
'equal'	No weighting
'inverse'	Weight is 1/distance
'squaredinverse'	Weight is 1/distance <sup>2</sup>
@ <i>fcn</i>	<i>fcn</i> is a function that accepts a matrix of nonnegative distances, and returns a matrix the same size containing nonnegative distance weights. For example, 'squaredinverse' is equivalent to @ (d) d.^(-2).

Example: 'DistanceWeight', 'inverse'

Data Types: `function_handle` | `char`

### **Exponent** — Minkowski distance exponent

2 (default) | positive scalar value

Minkowski distance exponent, specified as the comma-separated pair consisting of 'Exponent' and a positive scalar value. This argument is only valid when 'Distance' is 'minkowski'.

Example: `'Exponent', 3`

Data Types: `single` | `double`

### **IncludeTies** — Tie inclusion flag

`false` (default) | `true`

Tie inclusion flag, specified as the comma-separated pair consisting of 'IncludeTies' and a logical value indicating whether `predict` includes all the neighbors whose distance values are equal to the Kth smallest distance. If `IncludeTies` is `true`, `predict` includes all these neighbors. Otherwise, `predict` uses exactly K neighbors.

Example: `'IncludeTies', true`

Data Types: `logical`

### **NSMethod** — Nearest neighbor search method

`'kdtree'` | `'exhaustive'`

Nearest neighbor search method, specified as the comma-separated pair consisting of 'NSMethod' and 'kdtree' or 'exhaustive'.

- `'kdtree'` — Create and use a *kd*-tree to find nearest neighbors. 'kdtree' is valid when the distance metric is one of the following:
  - `'euclidean'`
  - `'cityblock'`
  - `'minkowski'`
  - `'chebychev'`
- `'exhaustive'` — Use the exhaustive search algorithm. The distance values from all points in X to each point in Y are computed to find nearest neighbors.



The default is 'kdtree' when  $X$  has 10 or fewer columns,  $X$  is not sparse, and the distance metric is a 'kdtree' type; otherwise, 'exhaustive'.

Example: 'NSMethod', 'exhaustive'

### **NumNeighbors** — Number of nearest neighbors to find

1 (default) | positive integer value

Number of nearest neighbors in  $X$  to find for classifying each point when predicting, specified as the comma-separated pair consisting of 'NumNeighbors' and a positive integer value.

Example: 'NumNeighbors', 3

Data Types: single | double

### **Scale** — Distance scale

nanstd( $X$ ) (default) | vector of nonnegative scalar values

Distance scale, specified as the comma-separated pair consisting of 'Scale' and a vector containing nonnegative scalar values with length equal to the number of columns in  $X$ . Each coordinate difference between  $X$  and a query point is scaled by the corresponding element of `Scale`. This argument is only valid when 'Distance' is 'seuclidean'.

You cannot simultaneously specify 'Standardize' and either of 'Scale' or 'Cov'.

Data Types: single | double

### **Standardize** — Flag to standardize predictors

false (default) | true

Flag to standardize the predictors, specified as the comma-separated pair consisting of 'Standardize' and true (1) or false (0).

If you set 'Standardize', true, then the software centers and scales each column of the predictor data ( $X$ ) by the column mean and standard deviation, respectively.

The software does not standardize categorical predictors, and throws an error if all predictors are categorical.

You cannot simultaneously specify 'Standardize', 1 and either of 'Scale' or 'Cov'.

It is good practice to standardize the predictor data.

Example: `'Standardize', true`

Data Types: `logical`

## Output Arguments

**t** — *k*NN classification template

template object

*k*NN classification template suitable for training ensembles or error-correcting output code (ECOC) multiclass models, returned as a template object. Pass `t` to `fitcensemble` or `fitcecoc` to specify how to create the *k*NN classifier for the ensemble or ECOC model, respectively.

If you display `t` to the Command Window, then all, unspecified options appear empty (`[]`). However, the software replaces empty options with their corresponding default values during training.

## See Also

`ClassificationKNN` | `ExhaustiveSearcher` | `fitcecoc` | `fitcensemble`

## Topics

“Random Subspace Classification” on page 18-135

**Introduced in R2014a**

# templateLinear

Linear classification learner template

`templateLinear` creates a template suitable for fitting a linear classification model to high-dimensional data for multiclass problems.

The template specifies the binary learner model, regularization type and strength, and solver, among other things. After creating the template, train the model by passing the template and data to `fitcecoc`.

## Syntax

```
t = templateLinear()  
t = templateLinear(Name,Value)
```

## Description

`t = templateLinear()` returns a linear classification learner template.

If you specify a default template, then the software uses default values for all input arguments during training.

`t = templateLinear(Name,Value)` returns a template with additional options specified by one or more name-value pair arguments. For example, you can specify to implement logistic regression, specify the regularization type or strength, or specify the solver to use for objective-function minimization.

If you display `t` in the Command Window, then all options appear empty (`[]`) except options that you specify using name-value pair arguments. During training, the software uses default values for empty options.

## Examples

### Train Multiclass Linear Classification Model

Train an ECOC model composed of multiple binary, linear classification models.

Load the NLP data set.

```
load nlpdata
```

X is a sparse matrix of predictor data, and Y is a categorical vector of class labels. There are more than two classes in the data.

Create a default linear-classification-model template.

```
t = templateLinear();
```

To adjust the default values, see `templateLinear`.

Train an ECOC model composed of multiple binary, linear classification models that can identify the product given the frequency distribution of words on a documentation web page. For faster training time, transpose the predictor data, and specify that observations correspond to columns.

```
X = X';  
rng(1); % For reproducibility  
Mdl = fitcecoc(X,Y,'Learners',t,'ObservationsIn','columns')
```

```
Mdl =
```

```
classreg.learning.classif.CompactClassificationECOC  
  ResponseName: 'Y'  
  ClassNames: [1x13 categorical]  
  ScoreTransform: 'none'  
  BinaryLearners: {78x1 cell}  
  CodingMatrix: [13x78 double]
```

Alternatively, you can train an ECOC model composed of default linear classification models using `'Learners','Linear'`.

To conserve memory, `fitcecoc` returns trained ECOC models composed of linear classification learners in `CompactClassificationECOC` model objects.

## Input Arguments

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'Learner', 'logistic', 'Regularization', 'lasso', 'CrossVal', 'on'` specifies to implement logistic regression with a lasso penalty, and to implement 10-fold cross-validation.

### Linear Classification Options

#### **Lambda** — Regularization term strength

'auto' (default) | nonnegative scalar | vector of nonnegative values

Regularization term strength, specified as the comma-separated pair consisting of 'Lambda' and 'auto', a nonnegative scalar, or a vector of nonnegative values.

- For 'auto',  $\text{Lambda} = 1/n$ .
  - If you specify a cross-validation, name-value pair argument (e.g., `CrossVal`), then  $n$  is the number of in-fold observations.
  - Otherwise,  $n$  is the training sample size.
- For a vector of nonnegative values, the software sequentially optimizes the objective function for each distinct value in `Lambda` in ascending order.
  - If `Solver` is 'sgd' or 'asgd' and `Regularization` is 'lasso', then the software does not use the previous coefficient estimates as a warm start on page 32-2043 for the next optimization iteration. Otherwise, the software uses warm starts.
  - If `Regularization` is 'lasso', then any coefficient estimate of 0 retains its value when the software optimizes using subsequent values in `Lambda`.

Returns coefficient estimates for all optimization iterations.

Example: `'Lambda', 10.^(-(10:-2:2))`

Data Types: `char` | `double` | `single`

### Learner — Linear classification model type

`'svm'` (default) | `'logistic'`

Linear classification model type, specified as the comma-separated pair consisting of `'Learner'` and `'svm'` or `'logistic'`.

In this table,  $f(x) = x\beta + b$ .

- $\beta$  is a vector of  $p$  coefficients.
- $x$  is an observation from  $p$  predictor variables.
- $b$  is the scalar bias.

Value	Algorithm	Response Range	Loss Function
<code>'svm'</code>	Support vector machine	$y \in \{-1, 1\}$ ; 1 for the positive class, and -1 otherwise.	Hinge: $\ell[y, f(x)] = \max[0, 1 - yf(x)]$
<code>'logistic'</code>	Logistic regression	Same as <code>'svm'</code>	Deviance (logistic): $\ell[y, f(x)] = \log\{1 + \exp[-yf(x)]\}$

Example: `'Learner', 'logistic'`

Data Types: `char`

### Regularization — Complexity penalty type

`'lasso'` | `'ridge'`

Complexity penalty type, specified as the comma-separated pair consisting of `'Regularization'` and `'lasso'` or `'ridge'`.

The software composes the objective function for minimization from the sum of the average loss function (see `Learner`) and the regularization term in this table.

Value	Description
'lasso'	Lasso (L1) penalty: $\lambda \sum_{j=1}^p  \beta_j $
'ridge'	Ridge (L2) penalty: $\frac{\lambda}{2} \sum_{j=1}^p \beta_j^2$

To specify the regularization term strength, which is  $\lambda$  in the expressions, use `Lambda`.

The software excludes the bias term ( $\beta_0$ ) from the regularization penalty.

If `Solver` is 'sparsa', then the default value of `Regularization` is 'lasso'. Otherwise, the default is 'ridge'.

---

### Tip

- For predictor variable selection, specify 'lasso'.
  - For optimization accuracy, specify 'ridge'.
- 

Example: 'Regularization', 'lasso'

Data Types: char

### **solver** — Objective function minimization technique

'sgd' | 'asgd' | 'dual' | 'bfgs' | 'lbfgs' | 'sparsa' | cell array of character vectors

Objective function minimization technique, specified as the comma-separated pair consisting of 'Solver' and a character vector or cell array of character vectors from this table.

Value	Description	Restrictions
'sgd'	Stochastic gradient descent (SGD) [5][3]	

Value	Description	Restrictions
'asgd'	Average stochastic gradient descent (ASGD) [8]	
'dual'	Dual SGD for SVM [2][7]	Regularization must be 'ridge' and Learner must be 'svm'.
'bfgs'	Broyden-Fletcher-Goldfarb-Shanno quasi-Newton algorithm (BFGS) [4]	Inefficient if X is very high-dimensional.
'lbfgs'	Limited-memory BFGS (LBFGS) [4]	Regularization must be 'ridge'.
'sparsa'	Sparse Reconstruction by Separable Approximation (SpaRSA) [6]	Regularization must be 'lasso'.

If you specify:

- A ridge penalty (see Regularization) and the predictor data set contains 100 or fewer predictor variables, then the default solver is 'bfgs'.
- An SVM model (see Learner), a ridge penalty, and the predictor data set contains more than 100 predictor variables, then the default solver is 'dual'.
- A lasso penalty and the predictor data set contains 100 or fewer predictor variables, then the default solver is 'sparsa'.

Otherwise, the default solver is 'sgd'.

For more details on which solver to choose, see “Tips” on page 32-6445.

Example: 'Solver', {'sgd', 'lbfgs'}

Data Types: char | cell

### Beta — Initial linear coefficient estimates

zeros( $p, 1$ ) (default) | numeric vector | numeric matrix

Initial linear coefficient estimates ( $\beta$ ), specified as the comma-separated pair consisting of 'Beta' and a  $p$ -dimensional numeric vector or a  $p$ -by- $L$  numeric matrix.  $p$  is the number of predictor variables in X and  $L$  is the number of regularization-strength values (for more details, see Lambda).



- If you specify a  $p$ -dimensional vector, then the software optimizes the objective function  $L$  times using this process.
  - 1 The software optimizes using `Beta` as the initial value and the minimum value of `Lambda` as the regularization strength.
  - 2 The software optimizes again using the resulting estimate from the previous optimization as a warm start on page 32-2043, and the next smallest value in `Lambda` as the regularization strength.
  - 3 The software implements step 2 until it exhausts all values in `Lambda`.
- If you specify a  $p$ -by- $L$  matrix, then the software optimizes the objective function  $L$  times. At iteration  $j$ , the software uses `Beta(:, j)` as the initial value and, after it sorts `Lambda` in ascending order, uses `Lambda(j)` as the regularization strength.

If you set `'Solver', 'dual'`, then the software ignores `Beta`.

Data Types: `single` | `double`

### **Bias** — Initial intercept estimate

numeric scalar | numeric vector

Initial intercept estimate ( $b$ ), specified as the comma-separated pair consisting of `'Bias'` and a numeric scalar or an  $L$ -dimensional numeric vector.  $L$  is the number of regularization-strength values (for more details, see `Lambda`).

- If you specify a scalar, then the software optimizes the objective function  $L$  times using this process.
  - 1 The software optimizes using `Bias` as the initial value and the minimum value of `Lambda` as the regularization strength.
  - 2 The uses the resulting estimate as a warm start to the next optimization iteration, and uses the next smallest value in `Lambda` as the regularization strength.
  - 3 The software implements step 2 until it exhausts all values in `Lambda`.
- If you specify an  $L$ -dimensional vector, then the software optimizes the objective function  $L$  times. At iteration  $j$ , the software uses `Bias(j)` as the initial value and, after it sorts `Lambda` in ascending order, uses `Lambda(j)` as the regularization strength.
- By default:

- If `Learner` is `'logistic'`, then let  $g_j$  be 1 if  $Y(j)$  is the positive class, and -1 otherwise. `Bias` is the weighted average of the  $g$  for training or, for cross-validation, in-fold observations.
- If `Learner` is `'svm'`, then `Bias` is 0.

Data Types: `single` | `double`

### **FitBias** — Linear model intercept inclusion flag

`true` (default) | `false`

Linear model intercept inclusion flag, specified as the comma-separated pair consisting of `'FitBias'` and `true` or `false`.

Value	Description
<code>true</code>	The software includes the bias term $b$ in the linear model, and then estimates it.
<code>false</code>	The software sets $b = 0$ during estimation.

Example: `'FitBias', false`

Data Types: `logical`

### **PostFitBias** — Flag to fit linear model intercept after optimization

`false` (default) | `true`

Flag to fit the linear model intercept after optimization, specified as the comma-separated pair consisting of `'PostFitBias'` and `true` or `false`.

Value	Description
<code>false</code>	The software estimates the bias term $b$ and the coefficients $\beta$ during optimization.
<code>true</code>	To estimate $b$ , the software: <ol style="list-style-type: none"> <li><b>1</b> Estimates <math>\beta</math> and <math>b</math> using the model</li> <li><b>2</b> Estimates classification scores</li> <li><b>3</b> Refits <math>b</math> by placing the threshold on the classification scores that attains maximum accuracy</li> </ol>

If you specify `true`, then `FitBias` must be `true`.

Example: 'PostFitBias', true

Data Types: logical

### SGD and ASGD Solver Options

#### BatchSize — Mini-batch size

positive integer

Mini-batch size, specified as the comma-separated pair consisting of 'BatchSize' and a positive integer. At each iteration, the software estimates the gradient using BatchSize observations from the training data.

- If the predictor data is a numeric matrix, then the default value is 10.
- If the predictor data is a sparse matrix, then the default value is  $\max([10, \text{ceil}(\text{sqrt}(\text{ff}))])$ , where  $\text{ff} = \text{numel}(X) / \text{nnz}(X)$ , that is, the fullness factor of X.

Example: 'BatchSize', 100

Data Types: single | double

#### LearnRate — Learning rate

positive scalar

Learning rate, specified as the comma-separated pair consisting of 'LearnRate' and a positive scalar. LearnRate specifies how many steps to take per iteration. At each iteration, the gradient specifies the direction and magnitude of each step.

- If Regularization is 'ridge', then LearnRate specifies the initial learning rate  $\gamma_0$ . The software determines the learning rate for iteration  $t$ ,  $\gamma_t$ , using

$$\gamma_t = \frac{\gamma_0}{(1 + \lambda \gamma_0 t)^c}$$

- $\lambda$  is the value of Lambda.
- If Solver is 'sgd', then  $c = 1$ .
- If Solver is 'asgd', then  $c$  is 0.75 [7].
- If Regularization is 'lasso', then, for all iterations, LearnRate is constant.

By default, `LearnRate` is  $1/\sqrt{1+\max(\text{sum}(X.^2, \text{obsDim}))}$ , where `obsDim` is 1 if the observations compose the columns of the predictor data `X`, and 2 otherwise.

Example: `'LearnRate', 0.01`

Data Types: `single` | `double`

### **OptimizeLearnRate — Flag to decrease learning rate**

`true` (default) | `false`

Flag to decrease the learning rate when the software detects divergence (that is, overstepping the minimum), specified as the comma-separated pair consisting of `'OptimizeLearnRate'` and `true` or `false`.

If `OptimizeLearnRate` is `'true'`, then:

- 1 For the few optimization iterations, the software starts optimization using `LearnRate` as the learning rate.
- 2 If the value of the objective function increases, then the software restarts and uses half of the current value of the learning rate.
- 3 The software iterates step 2 until the objective function decreases.

Example: `'OptimizeLearnRate', true`

Data Types: `logical`

### **TruncationPeriod — Number of mini-batches between lasso truncation runs**

10 (default) | positive integer

Number of mini-batches between lasso truncation runs, specified as the comma-separated pair consisting of `'TruncationPeriod'` and a positive integer.

After a truncation run, the software applies a soft threshold to the linear coefficients. That is, after processing  $k = \text{TruncationPeriod}$  mini-batches, the software truncates the estimated coefficient  $j$  using

$$\hat{\beta}_j^* = \begin{cases} \hat{\beta}_j - u_t & \text{if } \hat{\beta}_j > u_t, \\ 0 & \text{if } |\hat{\beta}_j| \leq u_t, \\ \hat{\beta}_j + u_t & \text{if } \hat{\beta}_j < -u_t. \end{cases}$$

- For SGD,  $\hat{\beta}_j$  is the estimate of coefficient  $j$  after processing  $k$  mini-batches.  $u_t = k\gamma_t\lambda$ .  $\gamma_t$  is the learning rate at iteration  $t$ .  $\lambda$  is the value of Lambda.
- For ASGD,  $\hat{\beta}_j$  is the averaged estimate coefficient  $j$  after processing  $k$  mini-batches,  $u_t = k\lambda$ .

If Regularization is 'ridge', then the software ignores TruncationPeriod.

Example: 'TruncationPeriod', 100

Data Types: single | double

### SGD and ASGD Convergence Controls

#### BatchLimit — Maximal number of batches

positive integer

Maximal number of batches to process, specified as the comma-separated pair consisting of 'BatchLimit' and a positive integer. When the software processes BatchLimit batches, it terminates optimization.

- By default:
  - The software passes through the data PassLimit times.
  - If you specify multiple solvers, and use (A)SGD to get an initial approximation for the next solver, then the default value is  $\text{ceil}(1e6/\text{BatchSize})$ . BatchSize is the value of the 'BatchSize' name-value pair argument.
- If you specify 'BatchLimit' and 'PassLimit', then the software chooses the argument that results in processing the fewest observations.
- If you specify 'BatchLimit' but not 'PassLimit', then the software processes enough batches to complete up to one entire pass through the data.

Example: 'BatchLimit', 100

Data Types: single | double

#### BetaTolerance — Relative tolerance on linear coefficients and bias term

1e-4 (default) | nonnegative scalar

Relative tolerance on the linear coefficients and the bias term (intercept), specified as the comma-separated pair consisting of 'BetaTolerance' and a nonnegative scalar.

Let  $B_t = \begin{bmatrix} \beta_t' & b_t \end{bmatrix}$ , that is, the vector of the coefficients and the bias term at optimization

iteration  $t$ . If  $\left\| \frac{B_t - B_{t-1}}{B_t} \right\|_2 < \text{BetaTolerance}$ , then optimization terminates.

If the software converges for the last solver specified in `Solver`, then optimization terminates. Otherwise, the software uses the next solver specified in `Solver`.

Example: 'BetaTolerance', 1e-6

Data Types: single | double

**NumCheckConvergence** — Number of batches to process before next convergence check  
positive integer

Number of batches to process before next convergence check, specified as the comma-separated pair consisting of 'NumCheckConvergence' and a positive integer.

To specify the batch size, see `BatchSize`.

The software checks for convergence about 10 times per pass through the entire data set by default.

Example: 'NumCheckConvergence', 100

Data Types: single | double

**PassLimit** — Maximal number of passes  
1 (default) | positive integer

Maximal number of passes through the data, specified as the comma-separated pair consisting of 'PassLimit' and a positive integer.

The software processes all observations when it completes one pass through the data.

When the software passes through the data `PassLimit` times, it terminates optimization.

If you specify 'BatchLimit' and PassLimit, then the software chooses the argument that results in processing the fewest observations.

Example: 'PassLimit', 5

Data Types: single | double

### Dual SGD Convergence Controls

#### **BetaTolerance** — Relative tolerance on linear coefficients and bias term

1e-4 (default) | nonnegative scalar

Relative tolerance on the linear coefficients and the bias term (intercept), specified as the comma-separated pair consisting of 'BetaTolerance' and a nonnegative scalar.

Let  $B_t = \begin{bmatrix} \beta_t' & b_t \end{bmatrix}$ , that is, the vector of the coefficients and the bias term at optimization

iteration  $t$ . If  $\left\| \frac{B_t - B_{t-1}}{B_t} \right\|_2 < \text{BetaTolerance}$ , then optimization terminates.

If you also specify DeltaGradientTolerance, then optimization terminates when the software satisfies either stopping criterion.

If the software converges for the last solver specified in Solver, then optimization terminates. Otherwise, the software uses the next solver specified in Solver.

Example: 'BetaTolerance', 1e-6

Data Types: single | double

#### **DeltaGradientTolerance** — Gradient-difference tolerance

1 (default) | nonnegative scalar

Gradient-difference tolerance between upper and lower pool Karush-Kuhn-Tucker (KKT) complementarity conditions on page 32-2128 violators, specified as the comma-separated pair consisting of 'DeltaGradientTolerance' and a nonnegative scalar.

- If the magnitude of the KKT violators is less than DeltaGradientTolerance, then the software terminates optimization.
- If the software converges for the last solver specified in Solver, then optimization terminates. Otherwise, the software uses the next solver specified in Solver.

Example: `'DeltaGapTolerance', 1e-2`

Data Types: `double` | `single`

**NumCheckConvergence** — Number of passes through entire data set to process before next convergence check

5 (default) | positive integer

Number of passes through entire data set to process before next convergence check, specified as the comma-separated pair consisting of `'NumCheckConvergence'` and a positive integer.

Example: `'NumCheckConvergence', 100`

Data Types: `single` | `double`

**PassLimit** — Maximal number of passes

10 (default) | positive integer

Maximal number of passes through the data, specified as the comma-separated pair consisting of `'PassLimit'` and a positive integer.

When the software completes one pass through the data, it has processed all observations.

When the software passes through the data `PassLimit` times, it terminates optimization.

Example: `'PassLimit', 5`

Data Types: `single` | `double`

**BFGS, LBFGS, and SpaRSA Convergence Controls**

**BetaTolerance** — Relative tolerance on linear coefficients and bias term

1e-4 (default) | nonnegative scalar

Relative tolerance on the linear coefficients and the bias term (intercept), specified as the comma-separated pair consisting of `'BetaTolerance'` and a nonnegative scalar.



Let  $B_t = \begin{bmatrix} \beta_t' & b_t \end{bmatrix}$ , that is, the vector of the coefficients and the bias term at optimization

iteration  $t$ . If  $\left\| \frac{B_t - B_{t-1}}{B_t} \right\|_2 < \text{BetaTolerance}$ , then optimization terminates.

If you also specify `GradientTolerance`, then optimization terminates when the software satisfies either stopping criterion.

If the software converges for the last solver specified in `Solver`, then optimization terminates. Otherwise, the software uses the next solver specified in `Solver`.

Example: `'BetaTolerance', 1e-6`

Data Types: `single` | `double`

#### **GradientTolerance — Absolute gradient tolerance**

`1e-6` (default) | nonnegative scalar

Absolute gradient tolerance, specified as the comma-separated pair consisting of `'GradientTolerance'` and a nonnegative scalar.

Let  $\nabla L_t$  be the gradient vector of the objective function with respect to the coefficients

and bias term at optimization iteration  $t$ . If  $\|\nabla L_t\|_\infty = \max |\nabla L_t| < \text{GradientTolerance}$ , then optimization terminates.

If you also specify `BetaTolerance`, then optimization terminates when the software satisfies either stopping criterion.

If the software converges for the last solver specified in the software, then optimization terminates. Otherwise, the software uses the next solver specified in `Solver`.

Example: `'GradientTolerance', 1e-5`

Data Types: `single` | `double`

#### **HessianHistorySize — Size of history buffer for Hessian approximation**

`15` (default) | positive integer

Size of history buffer for Hessian approximation, specified as the comma-separated pair consisting of `'HessianHistorySize'` and a positive integer. That is, at each iteration,

the software composes the Hessian using statistics from the latest `HessianHistorySize` iterations.

The software does not support `'HessianHistorySize'` for `SpaRSA`.

Example: `'HessianHistorySize', 10`

Data Types: `single` | `double`

### **IterationLimit** — Maximal number of optimization iterations

1000 (default) | positive integer

Maximal number of optimization iterations, specified as the comma-separated pair consisting of `'IterationLimit'` and a positive integer. `IterationLimit` applies to these values of `Solver`: `'bfgs'`, `'lbfgs'`, and `'sparsa'`.

Example: `'IterationLimit', 500`

Data Types: `single` | `double`

## Output Arguments

### **t** — Linear classification model learner template

template object

Linear classification model learner template, returned as a template object. To train a linear classification model using high-dimensional data for multiclass problems, pass `t` to `fitcecoc`.

If you display `t` to the Command Window, then all, unspecified options appear empty (`[]`). However, the software replaces empty options with their corresponding default values during training.

## Definitions

### Warm Start

A warm start is initial estimates of the beta coefficients and bias term supplied to an optimization routine for quicker convergence.

## Tips

- It is a best practice to orient your predictor matrix so that observations correspond to columns and to specify 'ObservationsIn', 'columns'. As a result, you can experience a significant reduction in optimization-execution time.
- For better optimization accuracy if the predictor data is high-dimensional and Regularization is 'ridge', set any of these combinations for Solver:
  - 'sgd'
  - 'asgd'
  - 'dual' if Learner is 'svm'
  - {'sgd', 'lbfgs'}
  - {'asgd', 'lbfgs'}
  - {'dual', 'lbfgs'} if Learner is 'svm'

Other combinations can result in poor optimization accuracy.

- For better optimization accuracy if the predictor data is moderate- through low-dimensional and Regularization is 'ridge', set Solver to 'bfgs'.
- If Regularization is 'lasso', set any of these combinations for Solver:
  - 'sgd'
  - 'asgd'
  - 'sparsa'
  - {'sgd', 'sparsa'}
  - {'asgd', 'sparsa'}
- When choosing between SGD and ASGD, consider that:
  - SGD takes less time per iteration, but requires more iterations to converge.
  - ASGD requires fewer iterations to converge, but takes more time per iteration.
- If the predictor data has few observations, but many predictor variables, then:
  - Specify 'PostFitBias', true.
  - For SGD or ASGD solvers, set PassLimit to a positive integer that is greater than 1, for example, 5 or 10. This setting often results in better accuracy.

- For SGD and ASGD solvers, `BatchSize` affects the rate of convergence.
  - If `BatchSize` is too small, then the software achieves the minimum in many iterations, but computes the gradient per iteration quickly.
  - If `BatchSize` is too large, then the software achieves the minimum in fewer iterations, but computes the gradient per iteration slowly.
- Large learning rate (see `LearnRate`) speed-up convergence to the minimum, but can lead to divergence (that is, over-stepping the minimum). Small learning rates ensure convergence to the minimum, but can lead to slow termination.
- If `Regularization` is 'lasso', then experiment with various values of `TruncationPeriod`. For example, set `TruncationPeriod` to 1, 10, and then 100.
- For efficiency, the software does not standardize predictor data. To standardize the predictor data (`X`), enter

```
X = bsxfun(@rdivide,bsxfun(@minus,X,mean(X,2)),std(X,0,2));
```

The code requires that you orient the predictors and observations as the rows and columns of `X`, respectively. Also, for memory-usage economy, the code replaces the original predictor data the standardized data.

## References

- [1] Hsieh, C. J., K. W. Chang, C. J. Lin, S. S. Keerthi, and S. Sundararajan. “A Dual Coordinate Descent Method for Large-Scale Linear SVM.” *Proceedings of the 25th International Conference on Machine Learning, ICML '08*, 2001, pp. 408–415.
- [2] Langford, J., L. Li, and T. Zhang. “Sparse Online Learning Via Truncated Gradient.” *J. Mach. Learn. Res.*, Vol. 10, 2009, pp. 777–801.
- [3] Nocedal, J. and S. J. Wright. *Numerical Optimization*, 2nd ed., New York: Springer, 2006.
- [4] Shalev-Shwartz, S., Y. Singer, and N. Srebro. “Pegasos: Primal Estimated Sub-Gradient Solver for SVM.” *Proceedings of the 24th International Conference on Machine Learning, ICML '07*, 2007, pp. 807–814.
- [5] Wright, S. J., R. D. Nowak, and M. A. T. Figueiredo. “Sparse Reconstruction by Separable Approximation.” *Trans. Sig. Proc.*, Vol. 57, No 7, 2009, pp. 2479–2493.

- [6] Xiao, Lin. “Dual Averaging Methods for Regularized Stochastic Learning and Online Optimization.” *J. Mach. Learn. Res.*, Vol. 11, 2010, pp. 2543–2596.
- [7] Xu, Wei. “Towards Optimal One Pass Large Scale Learning with Averaged Stochastic Gradient Descent.” *CoRR*, abs/1107.2490, 2011.

## See Also

`fitcecoc` | `fitclinear` | `fitrlinear`

**Introduced in R2016a**

## templateNaiveBayes

Naive Bayes classifier template

### Syntax

```
t = templateNaiveBayes()  
t = templateNaiveBayes(Name,Value)
```

### Description

`t = templateNaiveBayes()` returns a naive Bayes on page 32-6454 template suitable for training error-correcting output code (ECOC) multiclass models.

If you specify a default template, then the software uses default values for all input arguments during training.

Specify `t` as a learner in `fitcecoc`.

`t = templateNaiveBayes(Name,Value)` returns a template with additional options specified by one or more name-value pair arguments. All properties of `t` are empty, except those you specify using `Name,Value` pair arguments.

For example, you can specify distributions for the predictors.

If you display `t` in the Command Window, then all options appear empty (`[]`), except those that you specify using name-value pair arguments. During training, the software uses default values for empty options.

### Examples

#### Create a Default Naive Bayes Template

Use `templateNaiveBayes` to specify a default naive Bayes template.

```
t = templateNaiveBayes()

t =

Fit template for classification NaiveBayes.

    DistributionNames: [1x0 double]
           Kernel: []
           Support: []
           Width: []
           Version: 1
           Method: 'NaiveBayes'
           Type: 'classification'
```

All properties of the template object are empty except for `Method` and `Type`. When you pass `t` to the training function, the software fills in the empty properties with their respective default values. For example, the software fills the `DistributionNames` property with a 1-by-`D` cell array of character vectors with 'normal' in each cell, where `D` is the number of predictors. For details on other default values, see `fitcnb`.

`t` is a plan for a naive Bayes learner, and no computation occurs when you specify it. You can pass `t` to `fitcecoc` to specify naive Bayes binary learners for ECOC multiclass learning.

### Create a Naive Bayes Template for ECOC Multiclass Learning

Create a nondefault naive Bayes template for use in `fitcecoc`.

Load Fisher's iris data set.

```
load fisheriris
```

Create a template for naive Bayes binary classifiers, and specify kernel distributions for all predictors.

```
t = templateNaiveBayes('DistributionNames','kernel')
```

```
t =
```

```
Fit template for classification NaiveBayes.
```

```
DistributionNames: 'kernel'  
  Kernel: []  
  Support: []  
  Width: []  
Version: 1  
  Method: 'NaiveBayes'  
  Type: 'classification'
```

All properties of the template object are empty except for `DistributionNames`, `Method`, and `Type`. When you pass `t` to the training function, the software fills in the empty properties with their respective default values.

Specify `t` as a binary learner for an ECOC multiclass model.

```
Mdl = fitcecoc(meas,species,'Learners',t);
```

By default, the software trains `Mdl` using the one-versus-one coding design.

Display the in-sample (resubstitution) misclassification error.

```
L = resubLoss(Mdl,'LossFun','classiferror')
```

```
L =
```

```
0.0333
```

## Input Arguments

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'DistributionNames','mn'` specifies to treat all predictors as token counts for a multinomial model.



**DistributionNames — Data distributions**

'kernel' | 'mn' | 'mvnm' | 'normal' | cell array of character vectors

Data distributions `fitcnb` uses to model the data, specified as the comma-separated pair consisting of 'DistributionNames' and a value in this table.

Value	Description
'kernel'	Kernel smoothing density estimate.
'mn'	Multinomial distribution. If you specify <code>mn</code> , then all features are components of a multinomial distribution. Therefore, you cannot include 'mn' as an element of a cell array of character vectors. For details, see “Algorithms” on page 32-2083.
'mvnm'	Multivariate multinomial distribution. For details, see “Algorithms” on page 32-2083.
'normal'	Normal (Gaussian) distribution.

If you specify a character vector, then the software models all the features using that distribution. If you specify a 1-by- $P$  cell array of character vectors, then the software models feature  $j$  using the distribution in element  $j$  of the cell array.

By default, the software sets all predictors specified as categorical predictors (using the `CategoricalPredictors` name-value pair argument) to 'mvnm'. Otherwise, the default distribution is 'normal'.

You must specify that at least one predictor has distribution 'kernel' to additionally specify `Kernel`, `Support`, or `Width`.

Example: 'Distribution', 'mn'

Data Types: cell | char

**Kernel — Kernel smoother type**

'normal' (default) | 'box' | 'epanechnikov' | 'triangle' | cell array of character vectors

Kernel smoother type, specified as the comma-separated pair consisting of 'Kernel' and a character vector or cell array of character vectors.

This table summarizes the available options for setting the kernel smoothing density region. Let  $I\{u\}$  denote the indicator function.

Value	Kernel	Formula
'box'	Box (uniform)	$f(x) = 0.5I\{ x  \leq 1\}$
'epanechnikov'	Epanechnikov	$f(x) = 0.75(1 - x^2)I\{ x  \leq 1\}$
'normal'	Gaussian	$f(x) = \frac{1}{\sqrt{2\pi}} \exp(-0.5x^2)$
'triangle'	Triangular	$f(x) = (1 -  x )I\{ x  \leq 1\}$

If you specify a 1-by- $P$  cell array, with each cell containing any value in the table, then the software trains the classifier using the kernel smoother type in cell  $j$  for feature  $j$  in  $x$ . The software ignores cells of `Kernel` not corresponding to a predictor whose distribution is 'kernel'.

You must specify that at least one predictor has distribution 'kernel' to additionally specify `Kernel`, `Support`, or `Width`.

Example: 'Kernel', {'epanechnikov', 'normal'}

Data Types: cell | char

### Support — Kernel smoothing density support

'unbounded' (default) | 'positive' | cell array | numeric row vector

Kernel smoothing density support, specified as the comma-separated pair consisting of 'Support' and 'positive', 'unbounded', a cell array, or a numeric row vector. The software applies the kernel smoothing density to the specified region.

This table summarizes the available options for setting the kernel smoothing density region.

Value	Description
1-by-2 numeric row vector	For example, $[L, U]$ , where $L$ and $U$ are the finite lower and upper bounds, respectively, for the density support.
'positive'	The density support is all positive real values.

Value	Description
'unbounded'	The density support is all real values.

If you specify a 1-by- $P$  cell array, with each cell containing any value in the table, then the software trains the classifier using the kernel support in cell  $j$  for feature  $j$  in  $X$ . The software ignores cells of `Kernel` not corresponding to a predictor whose distribution is 'kernel'.

You must specify that at least one predictor has distribution 'kernel' to additionally specify `Kernel`, `Support`, or `Width`.

Example: 'KSSupport', {[-10,20], 'unbounded'}

Data Types: cell | char | double

#### **width** — Kernel smoothing window width

matrix of numeric values | numeric column vector | numeric row vector | scalar

Kernel smoothing window width, specified as the comma-separated pair consisting of 'width' and a matrix of numeric values, numeric column vector, numeric row vector, or scalar.

Suppose there are  $K$  class levels and  $P$  predictors. This table summarizes the available options for setting the kernel smoothing window width.

Value	Description
$K$ -by- $P$ matrix of numeric values	Element $(k,j)$ specifies the width for predictor $j$ in class $k$ .
$K$ -by-1 numeric column vector	Element $k$ specifies the width for all predictors in class $k$ .
1-by- $P$ numeric row vector	Element $j$ specifies the width in all class levels for predictor $j$ .
scalar	Specifies the bandwidth for all features in all classes.

By default, the software selects a default width automatically for each combination of predictor and class by using a value that is optimal for a Gaussian distribution. If you specify `Width` and it contains NaNs, then the software selects widths for the elements containing NaNs.

You must specify that at least one predictor has distribution 'kernel' to additionally specify `Kernel`, `Support`, or `Width`.

Example: 'width', [NaN NaN]

Data Types: double | struct

## Output Arguments

### **$\tau$** — Naive Bayes classification template

template object

Naive Bayes classification template suitable for training error-correcting output code (ECOC) multiclass models, returned as a template object. Pass  $\tau$  to `fitcecoc` to specify how to create the naive Bayes classifier for the ECOC model.

If you display  $\tau$  to the Command Window, then all, unspecified options appear empty (`[]`). However, the software replaces empty options with their corresponding default values during training.

## Definitions

### Naive Bayes

Naive Bayes is a classification algorithm that applies density estimation to the data.

The algorithm leverages Bayes theorem, and (naively) assumes that the predictors are conditionally independent, given the class. Though the assumption is usually violated in practice, naive Bayes classifiers tend to yield posterior distributions that are robust to biased class density estimates, particularly where the posterior is 0.5 (the decision boundary) [1].

Naive Bayes classifiers assign observations to the most probable class (in other words, the maximum a posteriori decision rule). Explicitly, the algorithm:

- 1 Estimates the densities of the predictors within each class.
- 2 Models posterior probabilities according to Bayes rule. That is, for all  $k = 1, \dots, K$ ,

$$\hat{P}(Y = k | X_1, \dots, X_P) = \frac{\pi(Y = k) \prod_{j=1}^P P(X_j | Y = k)}{\sum_{k=1}^K \pi(Y = k) \prod_{j=1}^P P(X_j | Y = k)},$$

where:

- $Y$  is the random variable corresponding to the class index of an observation.
  - $X_1, \dots, X_P$  are the random predictors of an observation.
  - $\pi(Y = k)$  is the prior probability that a class index is  $k$ .
- 3** Classifies an observation by estimating the posterior probability for each class, and then assigns the observation to the class yielding the maximum posterior probability.

If the predictors compose a multinomial distribution, then the posterior probability

$\hat{P}(Y = k | X_1, \dots, X_P) \propto \pi(Y = k) P_{mn}(X_1, \dots, X_P | Y = k)$ , where  $P_{mn}(X_1, \dots, X_P | Y = k)$  is the probability mass function of a multinomial distribution.

## Algorithms

- If you specify 'Distribution', 'mn' when training Mdl using `fitcnb`, then the software fits a multinomial distribution using the bag-of-tokens model on page 32-2082. The software stores the probability that token  $j$  appears in class  $k$  in the property `DistributionParameters{k, j}`. Using additive smoothing [2], the estimated probability is

$$P(\text{token } j | \text{class } k) = \frac{1 + c_{jk}}{P + c_k},$$

where:

•

$c_{j|k} = n_k \frac{\sum_{i: y_i \in \text{class } k} x_{ij} w_i}{\sum_{i \in \text{class } k} w_i}$ ; which is the weighted number of occurrences of token  $j$  in class  $k$ .

•  $n_k$  is the number of observations in class  $k$ .

•

$w_i$  is the weight for observation  $i$ . The software normalizes weights within a class such that they sum to the prior probability for that class.

•

$c_k = \sum_{j=1}^P c_{j|k}$ ; which is the total weighted number of occurrences of all tokens in class  $k$ .

• If you specify 'Distribution', 'mvnmn' when training Mdl using fitcnb, then:

- 1 For each predictor, the software collects a list of the unique levels, stores the sorted list in CategoricalLevels, and considers each level a bin. Each predictor/class combination is a separate, independent multinomial random variable.
- 2 For predictor  $j$  in class  $k$ , the software counts instances of each categorical level using the list stored in CategoricalLevels{ $j$ }.
- 3 The software stores the probability that predictor  $j$ , in class  $k$ , has level  $L$  in the property DistributionParameters{ $k, j$ }, for all levels in CategoricalLevels{ $j$ }. Using additive smoothing [2], the estimated probability is

$$P(\text{predictor } j = L \mid \text{class } k) = \frac{1 + m_{j|k}(L)}{m_j + m_k},$$

where:

•

$m_{j|k}(L) = n_k \frac{\sum_{i: y_i \in \text{class } k} I\{x_{ij} = L\} w_i}{\sum_{i: y_i \in \text{class } k} w_i}$ ; which is the weighted number of observations for which predictor  $j$  equals  $L$  in class  $k$ .

- $n_k$  is the number of observations in class  $k$ .

•

$I\{x_{ij} = L\} = 1$  if  $x_{ij} = L$ , 0 otherwise.

•

$w_i$  is the weight for observation  $i$ . The software normalizes weights within a class such that they sum to the prior probability for that class.

- $m_j$  is the number of distinct levels in predictor  $j$ .

- $m_k$  is the weighted number of observations in class  $k$ .

## References

- [1] Hastie, T., R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*, Second Edition. NY: Springer, 2008.
- [2] Manning, C. D., P. Raghavan, and M. Schütze. *Introduction to Information Retrieval*, NY: Cambridge University Press, 2008.

## See Also

ClassificationECOC | ClassificationNaiveBayes | fitcecoc | fitcnb

Introduced in R2014b

## templateSVM

Support vector machine template

### Syntax

```
t = templateSVM()  
t = templateSVM(Name, Value)
```

### Description

`t = templateSVM()` returns a support vector machine (SVM) learner template suitable for training error-correcting output code (ECOC) multiclass models.

If you specify a default template, then the software uses default values for all input arguments during training.

Specify `t` as a binary learner, or one in a set of binary learners, in `fitcecoc` to train an ECOC multiclass classifier.

`t = templateSVM(Name, Value)` returns a template with additional options specified by one or more name-value pair arguments.

For example, you can specify the box constraint, the kernel function, or whether to standardize the predictors.

If you display `t` in the Command Window, then all options appear empty (`[]`), except those that you specify using name-value pair arguments. During training, the software uses default values for empty options.

### Examples

#### Create a Default Support Vector Machine Template

Use `templateSVM` to specify a default SVM template.



```
t = templateSVM()

t =

Fit template for classification SVM.

        Alpha: [0x1 double]
    BoxConstraint: []
        CacheSize: []
    CachingMethod: ''
        ClipAlphas: []
DeltaGradientTolerance: []
        Epsilon: []
    GapTolerance: []
    KKTolerance: []
    IterationLimit: []
    KernelFunction: ''
        KernelScale: []
        KernelOffset: []
KernelPolynomialOrder: []
        NumPrint: []
        Nu: []
    OutlierFraction: []
    RemoveDuplicates: []
    ShrinkagePeriod: []
        Solver: ''
    StandardizeData: []
    SaveSupportVectors: []
    VerbosityLevel: []
        Version: 2
        Method: 'SVM'
        Type: 'classification'
```

All properties of the template object are empty except for `Method` and `Type`. When you pass `t` to the training function, the software fills in the empty properties with their respective default values. For example, the software fills the `KernelFunction` property with `'linear'`. For details on other default values, see `fitcsvm`.

`t` is a plan for an SVM learner, and no computation occurs when you specify it. You can pass `t` to `fitcecoc` to specify SVM binary learners for ECOC multiclass learning. However, by default, `fitcecoc` uses default SVM binary learners.

### Create an SVM Template for ECOC Multiclass Learning

Create a nondefault SVM template for use in `fitcecoc`.

Load Fisher's iris data set.

```
load fisheriris
```

Create a template for SVM binary classifiers, and specify to use a Gaussian kernel function.

```
t = templateSVM('KernelFunction','gaussian')
```

```
t =
```

Fit template for classification SVM.

```
          Alpha: [0x1 double]
    BoxConstraint: []
          CacheSize: []
    CachingMethod: ''
          ClipAlphas: []
DeltaGradientTolerance: []
          Epsilon: []
          GapTolerance: []
          KKTolerance: []
    IterationLimit: []
    KernelFunction: 'gaussian'
          KernelScale: []
          KernelOffset: []
KernelPolynomialOrder: []
          NumPrint: []
          Nu: []
    OutlierFraction: []
RemoveDuplicates: []
    ShrinkagePeriod: []
          Solver: ''
```

```

StandardizeData: []
SaveSupportVectors: []
VerbosityLevel: []
Version: 2
Method: 'SVM'
Type: 'classification'

```

All properties of the template object are empty except for `DistributionNames`, `Method`, and `Type`. When trained on, the software fills in the empty properties with their respective default values.

Specify `t` as a binary learner for an ECOC multiclass model.

```
Mdl = fitcecoc(meas,species,'Learners',t);
```

`Mdl` is a `ClassificationECOC` multiclass classifier. By default, the software trains `Mdl` using the one-versus-one coding design.

Display the in-sample (resubstitution) misclassification error.

```
L = resubLoss(Mdl,'LossFun','classiferror')
```

```
L =
```

```
0.0200
```

## Retain and Discard Support Vectors of SVM Binary Learners

By default, `fitcecoc` empties the `Alpha`, `SupportVectorLabels`, and `SupportVectors` properties of the linear, SVM binary learners stored in the `BinaryLearners` property of the trained ECOC model. You can retain the support vectors and related values, and then discard them from the model.

Load Fisher's iris data set.

```
load fisheriris
rng(1); % For reproducibility
```

Train an ECOC model using the entire data set. Specify retaining the support vectors by passing in the appropriate SVM template.

```
t = templateSVM('SaveSupportVectors',true);
MdlSV = fitcecoc(meas,species,'Learners',t);
```

Mdl is a trained `ClassificationECOC` model. By default, `fitcecoc` uses linear, SVM binary learners. It implements a one-versus-one coding design, which requires three binary learners for three-class learning.

Access the estimated  $\alpha$  values using dot notation.

```
alpha = cell(3,1);
alpha{1} = MdlSV.BinaryLearners{1}.Alpha;
alpha{2} = MdlSV.BinaryLearners{2}.Alpha;
alpha{3} = MdlSV.BinaryLearners{3}.Alpha;
alpha
```

```
alpha =

    3x1 cell array

    { 3x1 double}
    { 3x1 double}
    {23x1 double}
```

alpha is a 3-by-1 cell array that stores the estimated values of  $\alpha$ .

Discard the support vectors and related values from the ECOC model.

```
Mdl = discardSupportVectors(MdlSV);
```

Mdl is similar to MdlSV, except that the Alpha, SupportVectorLabels, and SupportVectors of all linear SVM binary learners are empty ( []).

```
areAllEmpty = @(x) isempty([x.Alpha x.SupportVectors x.SupportVectorLabels]);
cellfun(areAllEmpty,Mdl.BinaryLearners)
```

```
ans =

    3x1 logical array
```

```
1
1
1
```

Compare the sizes of the two ECOC models.

```
vars = whos('MdlSV', 'Mdl');
100*(1 - vars(1).bytes/vars(2).bytes)
```

```
ans =
```

```
5.1575
```

Mdl is about 5% smaller than MdlSV.

Reduce your memory footprint by compacting Mdl, and then clearing MdlSV and Mdl from the workspace.

```
CMdl = compact(Mdl);
clear MdlSV Mdl;
```

Predict the label for a random row of the training data using the more efficient SVM model.

```
idx = randsample(size(meas,1),1)
predictedLabel = predict(CMdl,meas(idx,:))
trueLabel = species(idx)
```

```
idx =
```

```
63
```

```
predictedLabel =
```

```
1x1 cell array
```

```
 {'versicolor'}
```

```
trueLabel =
```

```
1x1 cell array  
  
{'versicolor'}
```

## Input Arguments

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'BoxConstraint', 0.1, 'KernelFunction', 'gaussian', 'Standardize', 1` specifies a box constraint of 0.1, to use the Gaussian (RBF) kernel, and to standardize the predictors.

#### **BoxConstraint** — Box constraint

1 (default) | positive scalar

Box constraint on page 32-2127, specified as the comma-separated pair consisting of `'BoxConstraint'` and a positive scalar.

For one-class learning, the software always sets the box constraint to 1.

For more details on the relationships and algorithmic behavior of `BoxConstraint`, `Cost`, `Prior`, `Standardize`, and `Weights`, see “Algorithms” on page 32-2132.

Example: `'BoxConstraint', 100`

Data Types: `double` | `single`

#### **CacheSize** — Cache size

1000 (default) | `'maximal'` | positive scalar

Cache size, specified as the comma-separated pair consisting of `'CacheSize'` and `'maximal'` or a positive scalar.

If `CacheSize` is `'maximal'`, then the software reserves enough memory to hold the entire  $n$ -by- $n$  Gram matrix on page 32-2127.

If `CacheSize` is a positive scalar, then the software reserves `CacheSize` megabytes of memory for training the model.

Example: `'CacheSize', 'maximal'`

Data Types: `double` | `char` | `single`

### **ClipAlphas** — Flag to clip alpha coefficients

`true` (default) | `false`

Flag to clip alpha coefficients, specified as the comma-separated pair consisting of `'ClipAlphas'` and either `true` or `false`.

Suppose that the alpha coefficient for observation  $j$  is  $\alpha_j$  and the box constraint of observation  $j$  is  $C_j$ ,  $j = 1, \dots, n$ .  $n$  is the training sample size.

Value	Description
<code>true</code>	At each iteration, if $\alpha_j$ is near 0 or near $C_j$ , then MATLAB sets $\alpha_j$ to 0 or to $C_j$ , respectively.
<code>false</code>	MATLAB does not change the alpha coefficients during optimization.

MATLAB stores the final values of  $\alpha$  in the `Alpha` property of the trained SVM model object.

`ClipAlphas` can affect SMO and ISDA convergence.

Example: `'ClipAlphas', false`

Data Types: `logical`

### **DeltaGradientTolerance** — Tolerance for gradient difference

nonnegative scalar

Tolerance for the gradient difference between upper and lower violators obtained by Sequential Minimal Optimization (SMO) or Iterative Single Data Algorithm (ISDA), specified as the comma-separated pair consisting of `'DeltaGradientTolerance'` and a nonnegative scalar.

If `DeltaGradientTolerance` is 0, then the software does not use the tolerance for the gradient difference to check for optimization convergence.

The defaults are:

- `1e-3` if the solver is SMO (for example, you set `'Solver', 'SMO'`)
- `0` if the solver is ISDA (for example, you set `'Solver', 'ISDA'`)

Example: `'DeltaGapTolerance', 1e-2`

Data Types: `double` | `single`

### **GapTolerance** — Feasibility gap tolerance

`0` (default) | nonnegative scalar

Feasibility gap tolerance obtained by SMO or ISDA, specified as the comma-separated pair consisting of `'GapTolerance'` and a nonnegative scalar.

If `GapTolerance` is 0, then the software does not use the feasibility gap tolerance to check for optimization convergence.

Example: `'GapTolerance', 1e-2`

Data Types: `double` | `single`

### **IterationLimit** — Maximal number of numerical optimization iterations

`1e6` (default) | positive integer

Maximal number of numerical optimization iterations, specified as the comma-separated pair consisting of `'IterationLimit'` and a positive integer.

The software returns a trained model regardless of whether the optimization routine successfully converges. `Mdl.ConvergenceInfo` contains convergence information.

Example: `'IterationLimit', 1e8`

Data Types: `double` | `single`

### **KernelFunction** — Kernel function

`'linear'` (default) | `'gaussian'` | `'rbf'` | `'polynomial'` | function name

Kernel function used to compute the elements of the Gram matrix on page 32-2127, specified as the comma-separated pair consisting of `'KernelFunction'` and a kernel



function name. Suppose  $G(x_j, x_k)$  is element  $(j, k)$  of the Gram matrix, where  $x_j$  and  $x_k$  are  $p$ -dimensional vectors representing observations  $j$  and  $k$  in  $X$ . This table contains supported kernel function names and their functional forms.

Kernel Function Name	Description	Formula
'gaussian' or 'rbf'	Gaussian or Radial Basis Function (RBF) kernel, default for one-class learning	$G(x_j, x_k) = \exp\left(-\ x_j - x_k\ ^2\right)$
'linear'	Linear kernel, default for two-class learning	$G(x_j, x_k) = x_j'x_k$
'polynomial'	Polynomial kernel. Use 'PolynomialOrder', $q$ to specify a polynomial kernel of order $q$ .	$G(x_j, x_k) = (1 + x_j'x_k)^q$

You can set your own kernel function, for example, `kernel`, by setting 'KernelFunction', 'kernel'. `kernel` must have the following form:

```
function G = kernel(U,V)
```

where:

- $U$  is an  $m$ -by- $p$  matrix. Columns correspond to predictor variables and rows correspond to observations.
- $V$  is an  $n$ -by- $p$  matrix. Columns correspond to predictor variables and rows correspond to observations.
- $G$  is an  $m$ -by- $n$  Gram matrix on page 32-2127 of the rows of  $U$  and  $V$ .

`kernel.m` must be on the MATLAB path.

It is good practice to avoid using generic names for kernel functions. For example, call a sigmoid kernel function 'mysigmoid' rather than 'sigmoid'.

Example: 'KernelFunction', 'gaussian'

Data Types: char

#### **KernelOffset** — Kernel offset parameter

nonnegative scalar

Kernel offset parameter, specified as the comma-separated pair consisting of `'KernelOffset'` and a nonnegative scalar.

The software adds `KernelOffset` to each element of the Gram matrix.

The defaults are:

- 0 if the solver is SMO (that is, you set `'Solver', 'SMO'`)
- 0.1 if the solver is ISDA (that is, you set `'Solver', 'ISDA'`)

Example: `'KernelOffset', 0`

Data Types: `double` | `single`

### **kernelScale** — Kernel scale parameter

1 (default) | `'auto'` | positive scalar

Kernel scale parameter, specified as the comma-separated pair consisting of `'KernelScale'` and `'auto'` or a positive scalar. The software divides all elements of the predictor matrix  $X$  by the value of `KernelScale`. Then, the software applies the appropriate kernel norm to compute the Gram matrix.

- If you specify `'auto'`, then the software selects an appropriate scale factor using a heuristic procedure. This heuristic procedure uses subsampling, so estimates can vary from one call to another. Therefore, to reproduce results, set a random number seed using `rng` before training.
- If you specify `KernelScale` and your own kernel function, for example, `kernel`, using `'KernelFunction', 'kernel'`, then the software throws an error. You must apply scaling within `kernel`.

Example: `'KernelScale', 'auto'`

Data Types: `double` | `single` | `char`

### **KKTolerance** — Karush-Kuhn-Tucker complementarity conditions violation tolerance

nonnegative scalar

Karush-Kuhn-Tucker (KKT) complementarity conditions on page 32-2128 violation tolerance, specified as the comma-separated pair consisting of `'KKTolerance'` and a nonnegative scalar.

If `KKTTolerance` is 0, then the software does not use the KKT complementarity conditions violation tolerance to check for optimization convergence.

The defaults are:

- 0 if the solver is SMO (for example, you set `'Solver', 'SMO'`)
- $1e-3$  if the solver is ISDA (for example, you set `'Solver', 'ISDA'`)

Example: `'KKTTolerance', 1e-2`

Data Types: `double` | `single`

### **Nu** — $\nu$ parameter for one-class learning

0.5 (default) | positive scalar

$\nu$  parameter for one-class learning on page 32-2128, specified as the comma-separated pair consisting of `'Nu'` and a positive scalar. Nu must be greater than 0 and at most 1.

Set Nu to control the tradeoff between ensuring most training examples are in the positive class and minimizing the weights in the score function.

Example: `'Nu', 0.25`

Data Types: `double` | `single`

### **NumPrint** — Number of iterations between optimization diagnostic message output

1000 (default) | nonnegative integer

Number of iterations between optimization diagnostic message output, specified as the comma-separated pair consisting of `'NumPrint'` and a nonnegative integer.

If you use `'Verbose', 1` and `'NumPrint', numprint`, then the software displays all optimization diagnostic messages from SMO and ISDA every numprint iterations in the Command Window.

Example: `'NumPrint', 500`

Data Types: `double` | `single`

### **OutlierFraction** — Expected proportion of outliers in training data

0 (default) | numeric scalar in the interval [0,1)

Expected proportion of outliers in the training data, specified as the comma-separated pair consisting of `'OutlierFraction'` and a numeric scalar in the interval [0,1).

If you set `'OutlierFraction', outlierfraction`, where `outlierfraction` is a value greater than 0, then:

- For two-class learning, the software implements robust learning. In other words, the software attempts to remove  $100 * \text{outlierfraction} \%$  of the observations when the optimization algorithm converges. The removed observations correspond to gradients that are large in magnitude.
- For one-class learning, the software finds an appropriate bias term such that `outlierfraction` of the observations in the training set have negative scores.

Example: `'OutlierFraction', 0.01`

Data Types: `double` | `single`

### **PolynomialOrder** — Polynomial kernel function order

3 (default) | positive integer

Polynomial kernel function order, specified as the comma-separated pair consisting of `'PolynomialOrder'` and a positive integer.

If you set `'PolynomialOrder'` and `KernelFunction` is not `'polynomial'`, then the software throws an error.

Example: `'PolynomialOrder', 2`

Data Types: `double` | `single`

### **SaveSupportVectors** — Store support vectors, their labels, and the estimated $\alpha$ coefficients

`true` | `false`

Store support vectors, their labels, and the estimated  $\alpha$  coefficients as properties of the resulting model, specified as the comma-separated pair consisting of `'SaveSupportVectors'` and `true` or `false`.

If `SaveSupportVectors` is `true`, the resulting model stores the support vectors in the `SupportVectors` property, their labels in the `SupportVectorLabels` property, and the estimated  $\alpha$  coefficients in the `Alpha` property of the compact, SVM learners.

If `SaveSupportVectors` is `false` and `KernelFunction` is `'linear'`, the resulting model does not store the support vectors and the related estimates.

To reduce memory consumption by compact SVM models, specify `SaveSupportVectors`.

For linear, SVM binary learners in an ECOC model, the default value is `false`. Otherwise, the default value is `true`.

Example: `'SaveSupportVectors', true`

Data Types: `logical`

**ShrinkagePeriod** — Number of iterations between movement of observations from active to inactive set

0 (default) | nonnegative integer

Number of iterations between the movement of observations from the active to inactive set, specified as the comma-separated pair consisting of `'ShrinkagePeriod'` and a nonnegative integer.

If you set `'ShrinkagePeriod', 0`, then the software does not shrink the active set.

Example: `'ShrinkagePeriod', 1000`

Data Types: `double` | `single`

**solver** — Optimization routine

`'ISDA'` | `'L1QP'` | `'SMO'`

Optimization routine, specified as the comma-separated pair consisting of `'Solver'` and a value in this table.

Value	Description
<code>'ISDA'</code>	Iterative Single Data Algorithm (see [30])
<code>'L1QP'</code>	Uses <code>quadprog</code> to implement <i>L1</i> soft-margin minimization by quadratic programming. This option requires an Optimization Toolbox license. For more details, see “Quadratic Programming Definition” (Optimization Toolbox).
<code>'SMO'</code>	Sequential Minimal Optimization (see [17])

The defaults are:

- `'ISDA'` if you set `'OutlierFraction'` to a positive value and for two-class learning
- `'SMO'` otherwise

Example: 'Solver', 'ISDA'

Data Types: char

### **Standardize** — Flag to standardize predictor data

false (default) | true

Flag to standardize the predictor data, specified as the comma-separated pair consisting of 'Standardize' and true (1) or false (0).

If you set 'Standardize', true:

- The software centers and scales each column of the predictor data (X) by the weighted column mean and standard deviation, respectively (for details on weighted standardizing, see “Algorithms” on page 32-2132). MATLAB does not standardize the data contained in the dummy variable columns generated for categorical predictors.
- The software trains the classifier using the standardized predictor matrix, but stores the unstandardized data in the classifier property X.

Example: 'Standardize', true

Data Types: logical

### **Verbose** — Verbosity level

0 (default) | 1 | 2

Verbosity level, specified as the comma-separated pair consisting of 'Verbose' and either 0, 1, or 2. Verbose controls the amount of optimization information that the software displays in the Command Window and saves as a structure to `Mdl.ConvergenceInfo.History`.

This table summarizes the available verbosity level options.

Value	Description
0	The software does not display or save convergence information.
1	The software displays diagnostic messages and saves convergence criteria every <code>numprint</code> iterations, where <code>numprint</code> is the value of the name-value pair argument 'NumPrint'.

Value	Description
2	The software displays diagnostic messages and saves convergence criteria at every iteration.

Example: 'Verbose', 1

Data Types: double | single

## Output Arguments

**t** — SVM classification template

template object

SVM classification template suitable for training error-correcting output code (ECOC) multiclass models, returned as a template object. Pass **t** to `fitcecoc` to specify how to create the SVM classifier for the ECOC model.

If you display **t** to the Command Window, then all, unspecified options appear empty (`[]`). However, the software replaces empty options with their corresponding default values during training.

## Tip

For linear, SVM binary learners, and for efficiency, `fitcecoc` empties the properties `Alpha`, `SupportVectorLabels`, and `SupportVectors`. `fitcecoc` lists `Beta`, rather than `Alpha`, in the model display.

To store `Alpha`, `SupportVectorLabels`, and `SupportVectors`, pass a linear, SVM template that specifies storing support vectors to `fitcecoc`. For example, enter:

```
t = templateSVM('SaveSupportVectors','on')
Mdl = fitcecoc(X,Y,'Learners',t);
```

You can remove the support vectors and related values by passing the resulting `ClassificationECOC` model to `discardSupportVectors`.

## References

- [1] Christianini, N., and J. C. Shawe-Taylor. *An Introduction to Support Vector Machines and Other Kernel-Based Learning Methods*. Cambridge, UK: Cambridge University Press, 2000.
- [2] Fan, R.-E., P.-H. Chen, and C.-J. Lin. “Working set selection using second order information for training support vector machines.” *Journal of Machine Learning Research*, Vol 6, 2005, pp. 1889–1918.
- [3] Hastie, T., R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*, Second Edition. NY: Springer, 2008.
- [4] Kecman V., T. -M. Huang, and M. Vogt. “Iterative Single Data Algorithm for Training Kernel Machines from Huge Data Sets: Theory and Performance.” In *Support Vector Machines: Theory and Applications*. Edited by Lipo Wang, 255–274. Berlin: Springer-Verlag, 2005.
- [5] Scholkopf, B., J. C. Platt, J. C. Shawe-Taylor, A. J. Smola, and R. C. Williamson. “Estimating the Support of a High-Dimensional Distribution.” *Neural Comput.*, Vol. 13, Number 7, 2001, pp. 1443–1471.
- [6] Scholkopf, B., and A. Smola. *Learning with Kernels: Support Vector Machines, Regularization, Optimization and Beyond, Adaptive Computation and Machine Learning*. Cambridge, MA: The MIT Press, 2002.

## See Also

`ClassificationECOC` | `ClassificationSVM` | `fitcecoc` | `fitcsvm`

**Introduced in R2014b**



# templateTree

Create decision tree template

## Syntax

```
t = templateTree
t = templateTree(Name,Value)
```

## Description

`t = templateTree` returns a default decision tree learner template suitable for training ensembles or error-correcting output code (ECOC) multiclass models. Specify `t` as a learner using:

- `fitcensemble` for classification ensembles
- `fitrensemble` for regression ensembles
- `fitcecoc` for ECOC model classification

If you specify a default decision tree template, then the software uses default values for all input arguments during training. It is good practice to specify the type of decision tree, e.g., for a classification tree template, specify `'Type', 'classification'`. If you specify the type of decision tree and display `t` in the Command Window, then all options except `Type` appear empty (`[]`).

`t = templateTree(Name,Value)` creates a template with additional options specified by one or more name-value pair arguments.

For example, you can specify the algorithm used to find the best split on a categorical predictor, the split criterion, or the number of predictors selected for each split.

If you display `t` in the Command Window, then all options appear empty (`[]`), except those that you specify using name-value pair arguments. During training, the software uses default values for empty options.

## Examples

### Create a Classification Template with Surrogate Splits

Create a decision tree template with surrogate splits, and use the template to train an ensemble using sample data.

Load Fisher's iris data set.

```
load fisheriris
```

Create a decision tree template with surrogate splits.

```
t = templateTree('Surrogate','on')
```

```
t =
```

```
Fit template for Tree.  
Surrogate: 'on'
```

All options of the template object are empty except for `Surrogate`. When you pass `t` to the training function, the software fills in the empty options with their respective default values.

Specify `t` as a weak learner for a classification ensemble.

```
Mdl = fitensemble(meas,species,'AdaBoostM2',100,t)
```

```
Mdl =
```

```
classreg.learning.classif.ClassificationEnsemble  
    ResponseName: 'Y'  
    CategoricalPredictors: []  
        ClassNames: {'setosa' 'versicolor' 'virginica'}  
    ScoreTransform: 'none'  
    NumObservations: 150  
    NumTrained: 100  
        Method: 'AdaBoostM2'  
    LearnerNames: {'Tree'}  
    ReasonForTermination: 'Terminated normally after completing the requested number o
```

```
FitInfo: [100x1 double]
FitInfoDescription: {2x1 cell}
```

Display the in-sample (resubstitution) misclassification error.

```
L = resubLoss(Mdl)
```

```
L =
```

```
0.0333
```

## Train Regression Ensemble

Use a trained, boosted regression tree ensemble to predict the fuel economy of a car. Choose the number of cylinders, volume displaced by the cylinders, horsepower, and weight as predictors. Then, train an ensemble using fewer predictors and compare its in-sample predictive accuracy against the first ensemble.

Load the `carsmall` data set. Store the training data in a table.

```
load carsmall
Tbl = table(Cylinders,Displacement,Horsepower,Weight,MPG);
```

Specify a regression tree template that uses surrogate splits to improve predictive accuracy in the presence of NaN values.

```
t = templateTree('Surrogate','On');
```

Train the regression tree ensemble using LSBoost and 100 learning cycles.

```
Mdl1 = fitensemble(Tbl,'MPG','LSBoost',100,t);
```

`Mdl1` is a trained `RegressionEnsemble` regression ensemble. Because `MPG` is a variable in the MATLAB® Workspace, you can obtain the same result by entering

```
Mdl1 = fitensemble(Tbl,MPG,'LSBoost',100,t);
```

Use the trained regression ensemble to predict the fuel economy for a four-cylinder car with a 200-cubic inch displacement, 150 horsepower, and weighing 3000 lbs.

```
predMPG = predict(Mdl1,[4 200 150 3000])
```

```
predMPG =  
    22.8462
```

The average fuel economy of a car with these specifications is 21.78 mpg.

Train a new ensemble using all predictors in `Tbl` except Displacement.

```
formula = 'MPG ~ Cylinders + Horsepower + Weight';  
Mdl2 = fitensemble(Tbl,formula,'LSBoost',100,t);
```

Compare the resubstitution MSEs between `Mdl1` and `Mdl2`.

```
mse1 = resubLoss(Mdl1)  
mse2 = resubLoss(Mdl2)
```

```
mse1 =  
    6.4721
```

```
mse2 =  
    7.8599
```

The in-sample MSE for the ensemble that trains on all predictors is lower.

### Find the Optimal Number of Splits and Trees for an Ensemble

You can control the depth of the trees in an ensemble of decision trees. You can also control the tree depth in an ECOC model containing decision tree binary learners using the `MaxNumSplits`, `MinLeafSize`, or `MinParentSize` name-value pair parameters.

- When bagging decision trees, `fitensemble` grows deep decision trees by default. You can grow shallower trees to reduce model complexity or computation time.
- When boosting decision trees, `fitensemble` grows stumps (a tree with one split) by default. You can grow deeper trees for better accuracy.

Load the `carsmall` data set. Specify the variables `Acceleration`, `Displacement`, `Horsepower`, and `Weight` as predictors, and `MPG` as the response.

```
load carsmall
X = [Acceleration Displacement Horsepower Weight];
Y = MPG;
```

The default values of the tree depth controllers for boosting regression trees are:

- 1 for `MaxNumSplits`. This option grows stumps.
- 5 for `MinLeafSize`
- 10 for `MinParentSize`

To search for the optimal number of splits:

- 1 Train a set of ensembles. Exponentially increase the maximum number of splits for subsequent ensembles from stump to at most  $n - 1$  splits, where  $n$  is the training sample size. Also, decrease the learning rate for each ensemble from 1 to 0.1.
- 2 Cross validate the ensembles.
- 3 Estimate the cross-validated mean-squared error (MSE) for each ensemble.
- 4 Compare the cross-validated MSEs. The ensemble with the lowest one performs the best, and indicates the optimal maximum number of splits, number of trees, and learning rate for the data set.

Grow and cross validate a deep regression tree and a stump. Specify to use surrogate splits because the data contain missing values. These serve as benchmarks.

```
MdlDeep = fitrtree(X,Y,'CrossVal','on','MergeLeaves','off',...
    'MinParentSize',1,'Surrogate','on');
MdlStump = fitrtree(X,Y,'MaxNumSplits',1,'CrossVal','on','Surrogate','on');
```

Train the boosting ensembles using 150 regression trees. Cross validate the ensemble using 5-fold cross validation. Vary the maximum number of splits using the values in the sequence  $\{2^0, 2^1, \dots, 2^m\}$ , where  $m$  is such that  $2^m$  is no greater than  $n - 1$ , where  $n$  is the training sample size. For each variant, adjust the learning rate to each value in the set  $\{0.1, 0.25, 0.5, 1\}$ ;

```
n = size(X,1);
m = floor(log2(n - 1));
lr = [0.1 0.25 0.5 1];
maxNumSplits = 2.^(0:m);
```

```

numTrees = 150;
Mdl = cell(numel(maxNumSplits),numel(lr));
rng(1); % For reproducibility
for k = 1:numel(lr);
    for j = 1:numel(maxNumSplits);
        t = templateTree('MaxNumSplits',maxNumSplits(j),'Surrogate','on');
        Mdl{j,k} = fitensemble(X,Y,'LSBoost',numTrees,t,...
            'Type','regression','KFold',5,'LearnRate',lr(k));
    end;
end;

```

Compute the cross-validated MSE for each ensemble.

```

kflAll = @(x) kfoldLoss(x,'Mode','cumulative');
errorCell = cellfun(kflAll,Mdl,'Uniform',false);
error = reshape(cell2mat(errorCell),[numTrees numel(maxNumSplits) numel(lr)]);
errorDeep = kfoldLoss(MdlDeep);
errorStump = kfoldLoss(MdlStump);

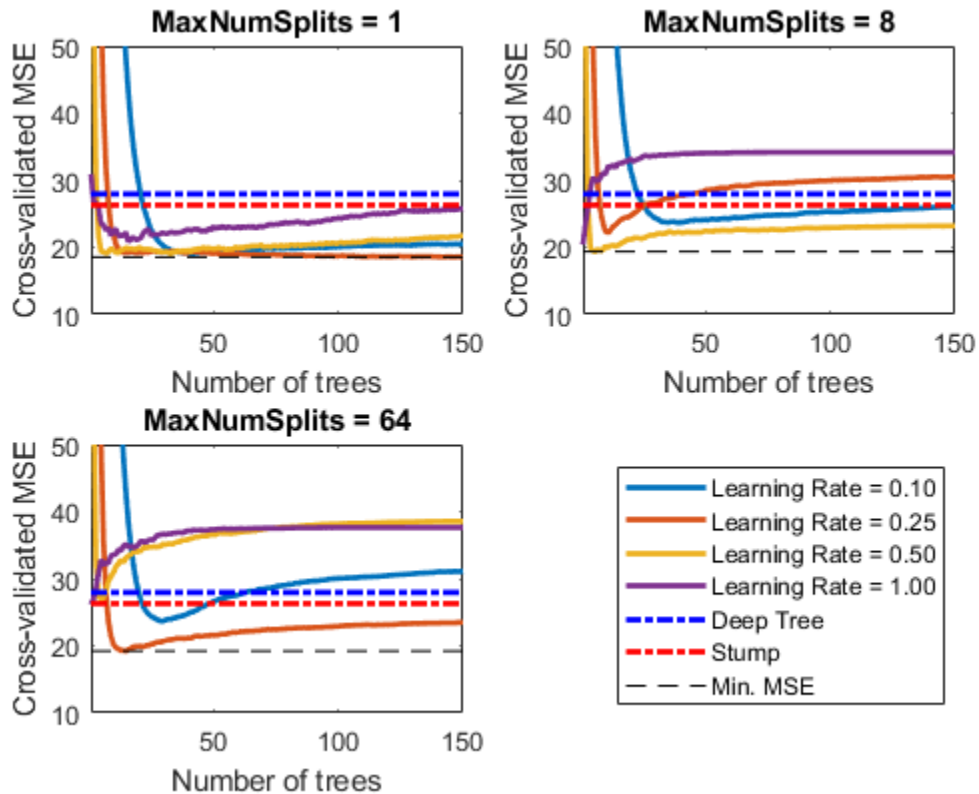
```

Plot how the cross-validated MSE behaves as the number of trees in the ensemble increases for a few of the ensembles, the deep tree, and the stump. Plot the curves with respect to learning rate in the same plot, and plot separate plots for varying tree complexities. Choose a subset of tree complexity levels.

```

mnsPlot = [1 round(numel(maxNumSplits)/2) numel(maxNumSplits)];
figure;
for k = 1:3;
    subplot(2,2,k);
    plot(squeeze(error(:,mnsPlot(k),:)), 'LineWidth',2);
    axis tight;
    hold on;
    h = gca;
    plot(h.XLim,[errorDeep errorDeep], '-.b', 'LineWidth',2);
    plot(h.XLim,[errorStump errorStump], '-.r', 'LineWidth',2);
    plot(h.XLim,min(min(error(:,mnsPlot(k),:)).*[1 1], '--k'));
    h.YLim = [10 50];
    xlabel 'Number of trees';
    ylabel 'Cross-validated MSE';
    title(sprintf('MaxNumSplits = %0.3g', maxNumSplits(mnsPlot(k))));
    hold off;
end;
hL = legend([cellstr(num2str(lr),'Learning Rate = %0.2f');...
    'Deep Tree'; 'Stump'; 'Min. MSE']);
hL.Position(1) = 0.6;

```



Each curve contains a minimum cross-validated MSE occurring at the optimal number of trees in the ensemble.

Identify the maximum number of splits, number of trees, and learning rate that yields the lowest MSE overall.

```
[minErr,minErrIdxLin] = min(error(:));
[idxNumTrees,idxMNS,idxLR] = ind2sub(size(error),minErrIdxLin);

fprintf('\nMin. MSE = %0.5f',minErr)
fprintf('\nOptimal Parameter Values:\nNum. Trees = %d',idxNumTrees);
fprintf('\nMaxNumSplits = %d\nLearning Rate = %0.2f\n',...
        maxNumSplits(idxMNS),lr(idxLR))
```

```
Min. MSE = 18.42979
Optimal Parameter Values:
Num. Trees = 1
MaxNumSplits = 4
Learning Rate = 1.00
```

For a different approach to optimizing this ensemble, see “Optimize a Boosted Regression Ensemble” on page 10-75.

### Unbiased Estimates of Predictor Importance Using Parallel Computing

This example requires a Parallel Computing Toolbox™ license.

Load the `carsmall` data set. Consider a model that predicts the mean fuel economy of a car given its acceleration, number of cylinders, engine displacement, horsepower, manufacturer, model year, and weight. Consider `Cylinders`, `Mfg`, and `Model_Year` as categorical variables.

```
load carsmall
Cylinders = categorical(Cylinders);
Mfg = categorical(cellstr(Mfg));
Model_Year = categorical(Model_Year);
X = table(Acceleration,Cylinders,Displacement,Horsepower,Mfg,...
    Model_Year,Weight,MPG);
rng('default'); % For reproducibility
```

Display the number of categories represented in the categorical variables.

```
numCylinders = numel(categories(Cylinders))
numMfg = numel(categories(Mfg))
numModelYear = numel(categories(Model_Year))
```

```
numCylinders =
```

```
3
```

```
numMfg =
```

```
28
```



```
numModelYear =
    3
```

Because there are 3 categories only in `Cylinders` and `Model_Year`, the standard CART, predictor-splitting algorithm prefers splitting a continuous predictor over these two variables.

Train a random forest of 500 regression trees using the entire data set. To grow unbiased trees, specify usage of the curvature test for splitting predictors. Because there are missing values in the data, specify usage of surrogate splits.

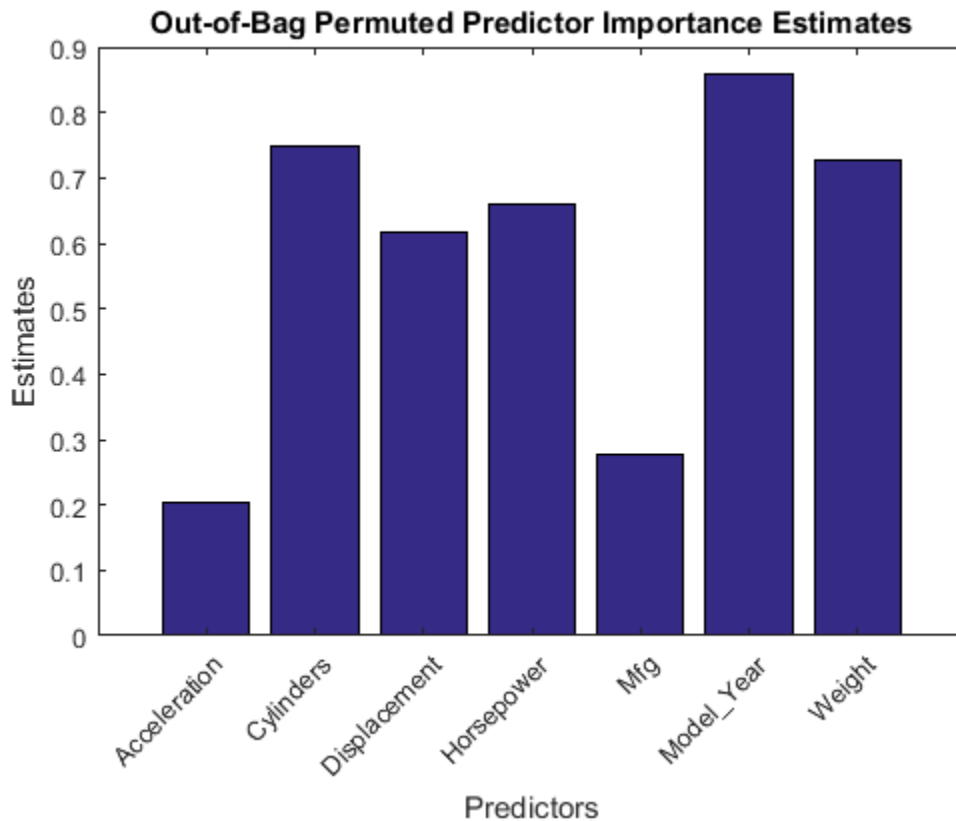
```
t = templateTree('PredictorSelection','curvature','Surrogate','on');
Mdl = fitensemble(X, 'MPG', 'Method', 'bag', 'NumLearningCycles', 500, ...
    'Learners', t);
```

Estimate predictor importance measures by permuting out-of-bag observations. Perform calculations in parallel. Compare the estimates using a bar graph.

```
options = statset('UseParallel', true);
imp = oobPermutedPredictorImportance(Mdl, 'Options', options);

figure;
bar(imp);
title('Out-of-Bag Permuted Predictor Importance Estimates');
ylabel('Estimates');
xlabel('Predictors');
h = gca;
h.XTickLabel = Mdl.PredictorNames;
h.XTickLabelRotation = 45;
h.TickLabelInterpreter = 'none';
```

```
Starting parallel pool (parpool) using the 'local' profile ... connected to 4 workers.
```



In this case, `Model_Year` is the most important predictor, followed by `Cylinders`. Compare these results to the results in “Estimate Importance of Predictors” on page 32-4475.

## Input Arguments

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single

quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'Surrogate', 'on', 'NVarToSample', 'all'` specifies a template with surrogate splits, and uses all available predictors at each split.

### For Classification Trees and Regression Trees

#### **MaxNumSplits** — Maximal number of decision splits

positive integer

Maximal number of decision splits (or branch nodes) per tree, specified as the comma-separated pair consisting of `'MaxNumSplits'` and a positive integer. `templateTree` splits `MaxNumSplits` or fewer branch nodes. For more details on splitting behavior, see “Algorithms” on page 32-6492.

For bagged decision trees and decision tree binary learners in ECOC models, the default is  $n - 1$ , where  $n$  is the number of observations in the training sample. For boosted decision trees, the default is 1.

Example: `'MaxNumSplits', 5`

Data Types: `single` | `double`

#### **MergeLeaves** — Leaf merge flag

`'off'` | `'on'`

Leaf merge flag, specified as the comma-separated pair consisting of `'MergeLeaves'` and either `'on'` or `'off'`.

When `'on'`, the decision tree merges leaves that originate from the same parent node, and that provide a sum of risk values greater or equal to the risk associated with the parent node. When `'off'`, the decision tree does not merge leaves.

For ensembles models, the default is `'off'`. For decision tree binary learners in ECOC models, the default is `'on'`.

Example: `'MergeLeaves', 'on'`

#### **MinLeafSize** — Minimum observations per leaf

positive integer value

Minimum observations per leaf, specified as the comma-separated pair consisting of `'MinLeafSize'` and a positive integer value. Each leaf has at least `MinLeafSize`

observations per tree leaf. If you supply both `MinParentSize` and `MinLeafSize`, the decision tree uses the setting that gives larger leaves: `MinParentSize = max(MinParentSize, 2*MinLeafSize)`.

For boosted and bagged decision trees, the defaults are 1 for classification and 5 for regression. For decision tree binary learners in ECOC models, the default is 1.

Example: `'MinLeafSize', 2`

### **MinParentSize** — Minimum observations per branch node

positive integer value

Minimum observations per branch node, specified as the comma-separated pair consisting of `'MinParentSize'` and a positive integer value. Each branch node in the tree has at least `MinParentSize` observations. If you supply both `MinParentSize` and `MinLeafSize`, the decision tree uses the setting that gives larger leaves: `MinParentSize = max(MinParentSize, 2*MinLeafSize)`.

For boosted and bagged decision trees, the defaults are 2 for classification and 10 for regression. For decision tree binary learners in ECOC models, the default is 10.

Example: `'MinParentSize', 4`

### **NumVariablesToSample** — Number of predictors to select at random for each split

positive integer value | `'all'`

Number of predictors to select at random for each split, specified as the comma-separated pair consisting of `'NumVariablesToSample'` and a positive integer value. Alternatively, you can specify `'all'` to use all available predictors.

For boosted decision trees and decision tree binary learners in ECOC models models, the default is `'all'`. The default for bagged decision trees is the square root of the number of predictors for classification, or one third of predictors for regression.

Example: `'NumVariablesToSample', 3`

### **PredictorSelection** — Algorithm used to select the best split predictor

`'allsplits'` (default) | `'curvature'` | `'interaction-curvature'`

Algorithm used to select the best split predictor at each node, specified as the comma-separated pair consisting of `'PredictorSelection'` and a value in this table.

Value	Description
'allsplits'	Standard CART — Selects the split predictor that maximizes the split-criterion gain over all possible splits of all predictors [1].
'curvature'	Curvature test — Selects the split predictor that minimizes the $p$ -value of chi-square tests of independence between each predictor and the response [3][4]. Training speed is similar to standard CART.
'interaction-curvature'	Interaction test — Chooses the split predictor that minimizes the $p$ -value of chi-square tests of independence between each predictor and the response, and that minimizes the $p$ -value of a chi-square test of independence between each pair of predictors and response [3]. Training speed can be slower than standard CART.

For 'curvature' and 'interaction-curvature', if all tests yield  $p$ -values greater than 0.05, then MATLAB stops splitting nodes.

---

### Tip

- The curvature and interaction tests are not recommended for boosting decision trees. To train an ensemble of boosted trees that has greater accuracy, use standard CART instead.
- Standard CART tends to select split predictors containing many distinct values, e.g., continuous variables, over those containing few distinct values, e.g., categorical variables [4]. If the predictor data set is heterogeneous, or if there are predictors that have relatively fewer distinct values than other variables, then consider specifying the curvature or interaction test.
  - If there are predictors that have relatively fewer distinct values than other predictors, for example, if the predictor data set is heterogeneous.
  - If an analysis of predictor importance is your goal. For more on predictor importance estimation, see `oobPermutedPredictorImportance` for classification problems, and `oobPermutedPredictorImportance` for regression problems.
- Trees grown using standard CART are not sensitive to predictor variable interactions. Also, such trees are less likely to identify important variables in the presence of many irrelevant predictors than the application of the interaction test. Therefore, to account

for predictor interactions and identify importance variables in the presence of many irrelevant variables, specify the interaction test [3].

- Prediction speed is unaffected by the value of 'PredictorSelection'.
- 

Example: 'PredictorSelection', 'curvature'

Data Types: char

### **Prune** — Flag to estimate optimal sequence of pruned subtrees

'off' (default) | 'on'

Flag to estimate the optimal sequence of pruned subtrees, specified as the comma-separated pair consisting of 'Prune' and 'on' or 'off'.

If `Prune` is 'on', then the software trains the classification tree learners without pruning them, but estimates the optimal sequence of pruned subtrees for each learner in the ensemble or decision tree binary learner in ECOC models. Otherwise, the software trains the classification tree learners without estimating the optimal sequence of pruned subtrees.

For ensembles, the default is 'off'.

For decision tree binary learners in ECOC models, then the default is 'on'.

Example: 'Prune', 'on'

### **PruneCriterion** — Pruning criterion

'error' | 'impurity' | 'mse'

Pruning criterion, specified as the comma-separated pair consisting of 'PruneCriterion' and a pruning criterion valid for the tree type.

- For classification trees, you can specify 'error' (default) or 'impurity'.
- For regression, you can only specify 'mse' (default).

Example: 'PruneCriterion', 'impurity'

### **SplitCriterion** — Split criterion

'gdi' | 'twoing' | 'deviance' | 'mse'

Split criterion, specified as the comma-separated pair consisting of `'SplitCriterion'` and a split criterion valid for the tree type.

- For classification trees:
  - `'gdi'` for Gini's diversity index (default)
  - `'twoing'` for the twoing rule
  - `'deviance'` for maximum deviance reduction (also known as cross entropy)
- For regression trees:
  - `'mse'` for mean squared error (default)

Example: `'SplitCriterion', 'deviance'`

#### **Surrogate — Surrogate decision splits**

`'off'` (default) | `'on'` | `'all'` | positive integer value

Surrogate decision splits flag, specified as the comma-separated pair consisting of `'Surrogate'` and one of `'off'`, `'on'`, `'all'`, or a positive integer value.

- When `'off'`, the decision tree does not find surrogate splits at the branch nodes.
- When `'on'`, the decision tree finds at most 10 surrogate splits at each branch node.
- When set to `'all'`, the decision tree finds all surrogate splits at each branch node. The `'all'` setting can consume considerable time and memory.
- When set to a positive integer value, the decision tree finds at most the specified number of surrogate splits at each branch node.

Use surrogate splits to improve the accuracy of predictions for data with missing values. This setting also lets you compute measures of predictive association between predictors.

Example: `'Surrogate', 'on'`

#### **For Classification Trees Only**

#### **AlgorithmForCategorical — Algorithm for best categorical predictor split**

`'Exact'` | `'PullLeft'` | `'PCA'` | `'OVByClass'`

Algorithm to find the best split on a categorical predictor for data with  $C$  categories for data and  $K \geq 3$  classes, specified as the comma-separated pair consisting of `'AlgorithmForCategorical'` and one of the following.

'Exact'	Consider all $2^{C-1} - 1$ combinations.
'PullLeft'	Start with all $C$ categories on the right branch. Consider moving each category to the left branch as it achieves the minimum impurity for the $K$ classes among the remaining categories. From this sequence, choose the split that has the lowest impurity.
'PCA'	Compute a score for each category using the inner product between the first principal component of a weighted covariance matrix (of the centered class probability matrix) and the vector of class probabilities for that category. Sort the scores in ascending order, and consider all $C - 1$ splits.
'OVAbyClass'	Start with all $C$ categories on the right branch. For each class, order the categories based on their probability for that class. For the first class, consider moving each category to the left branch in order, recording the impurity criterion at each move. Repeat for the remaining classes. From this sequence, choose the split that has the minimum impurity.

`ClassificationTree` selects the optimal subset of algorithms for each split using the known number of classes and levels of a categorical predictor. For two classes, `ClassificationTree` always performs the exact search. Use the `'AlgorithmForCategorical'` name-value pair argument to specify a particular algorithm.

Example: `'AlgorithmForCategorical', 'PCA'`

#### **MaxNumCategories** — Maximum category levels in split node

10 (default) | nonnegative scalar value

Maximum category levels in the split node, specified as the comma-separated pair consisting of `'MaxNumCategories'` and a nonnegative scalar value.



`ClassificationTree` splits a categorical predictor using the exact search algorithm if the predictor has at most `MaxNumCategories` levels in the split node. Otherwise, `ClassificationTree` finds the best categorical split using one of the inexact algorithms. Note that passing a small value can increase computation time and memory overload.

Example: `'MaxNumCategories',8`

### For Regression Trees Only

#### **QuadraticErrorTolerance** — Quadratic error tolerance

`1e-6` (default) | nonnegative scalar value

Quadratic error tolerance per node, specified as the comma —separated pair consisting of `'QuadraticErrorTolerance'` and a nonnegative scalar value. `RegressionTree` stops splitting nodes when the quadratic error per node drops below `QuadraticErrorTolerance*QED`, where `QED` is the quadratic error for the entire data computed before the decision tree is grown.  $QED = \text{NORM}(Y - YBAR)$ , where `YBAR` is estimated as the average of the input array `Y`.

Example: `'QuadraticErrorTolerance',1e-4`

## Output Arguments

### **t** — Decision tree template for classification or regression

template object

Decision tree template for classification or regression suitable for training ensembles or error-correcting output code (ECOC) multiclass models, returned as a template object. Pass `t` to `fitcensemble`, or `fitrensemble`, or `fitcecoc` to specify how to create the decision tree for the classification ensemble, regression ensemble, or ECOC model, respectively.

If you display `t` in the Command Window, then all unspecified options appear empty (`[]`). However, the software replaces empty options with their corresponding default values during training.

## Algorithms

- To accommodate `MaxNumSplits`, the software splits all nodes in the current layer, and then counts the number of branch nodes. A layer is the set of nodes that are equidistant from the root node. If the number of branch nodes exceeds `MaxNumSplits`, then the software follows this procedure.
  - 1 Determine how many branch nodes in the current layer need to be unsplit so that there would be at most `MaxNumSplits` branch nodes.
  - 2 Sort the branch nodes by their impurity gains.
  - 3 Unsplit the desired number of least successful branches.
  - 4 Return the decision tree grown so far.

This procedure aims at producing maximally balanced trees.

- The software splits branch nodes layer by layer until at least one of these events occurs.
  - There are `MaxNumSplits + 1` branch nodes.
  - A proposed split causes the number of observations in at least one branch node to be fewer than `MinParentSize`.
  - A proposed split causes the number of observations in at least one leaf node to be fewer than `MinLeafSize`.
  - The algorithm cannot find a good split within a layer (i.e., the pruning criterion (see `PruneCriterion`), does not improve for all proposed splits in a layer). A special case of this event is when all nodes are pure (i.e., all observations in the node have the same class).
  - For values `'curvature'` or `'interaction-curvature'` of `PredictorSelection`, all tests yield *p*-values greater than 0.05.

`MaxNumSplits` and `MinLeafSize` do not affect splitting at their default values. Therefore, if you set `'MaxNumSplits'`, then splitting might stop due to the value of `MinParentSize` before `MaxNumSplits` splits occur.

- For details on selecting split predictors and node-splitting algorithms when growing decision trees, see “Algorithms” on page 32-2174 for classification trees and “Algorithms” on page 32-2665 for regression trees.

## References

- [1] Breiman, L., J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Boca Raton, FL: CRC Press, 1984.
- [2] Coppersmith, D., S. J. Hong, and J. R. M. Hosking. “Partitioning Nominal Attributes in Decision Trees.” *Data Mining and Knowledge Discovery*, Vol. 3, 1999, pp. 197–217.
- [3] Loh, W.Y. “Regression Trees with Unbiased Variable Selection and Interaction Detection.” *Statistica Sinica*, Vol. 12, 2002, pp. 361–386.
- [4] Loh, W.Y. and Y.S. Shih. “Split Selection Methods for Classification Trees.” *Statistica Sinica*, Vol. 7, 1997, pp. 815–840.

## See Also

`ClassificationTree` | `RegressionTree` | `fitcecoc` | `fitcensemble` | `fitctree`  
| `fitrensemble`

**Introduced in R2014a**

## test

**Class:** `classregtree`

Error rate

---

**Note** `classregtree` will be removed in a future release. See `fitctree`, `fitrtree`, `ClassificationTree`, or `RegressionTree` instead.

---

## Syntax

```
cost = test(t, 'resubstitution')
cost = test(t, 'test', X, y)
cost = test(t, 'crossvalidate', X, y)
[cost, secost, ntnodes, bestlevel] = test(...)
[...] = test(..., param1, val1, param2, val2, ...)
```

## Description

`cost = test(t, 'resubstitution')` computes the cost of the tree `t` using a resubstitution method. `t` is a decision tree as created by `classregtree`. The cost of the tree is the sum over all terminal nodes of the estimated probability of a node times the cost of a node. If `t` is a classification tree, the cost of a node is the sum of the misclassification costs of the observations in that node. If `t` is a regression tree, the cost of a node is the average squared error over the observations in that node. `cost` is a vector of cost values for each subtree in the optimal pruning sequence for `t`. The resubstitution cost is based on the same sample that was used to create the original tree, so it under estimates the likely cost of applying the tree to new data.

`cost = test(t, 'test', X, y)` uses the matrix of predictors `X` and the response vector `y` as a test sample, applies the decision tree `t` to that sample, and returns a vector `cost` of cost values computed for the test sample. `X` and `y` should not be the same as the learning sample, that is, the sample that was used to fit the tree `t`.

`cost = test(t, 'crossvalidate', X, y)` uses 10-fold cross-validation to compute the cost vector. `X` and `y` should be the learning sample, that is, the sample that was used to

fit the tree `t`. The function partitions the sample into 10 subsamples, chosen randomly but with roughly equal size. For classification trees, the subsamples also have roughly the same class proportions. For each subsample, `test` fits a tree to the remaining data and uses it to predict the subsample. It pools the information from all subsamples to compute the cost for the whole sample.

`[cost, secost, ntnodes, bestlevel] = test(...)` also returns the vector `secost` containing the standard error of each cost value, the vector `ntnodes` containing the number of terminal nodes for each subtree, and the scalar `bestlevel` containing the estimated best level of pruning. A `bestlevel` of 0 means no pruning. The best level is the one that produces the smallest tree that is within one standard error of the minimum-cost subtree.

`[...] = test(..., param1, val1, param2, val2, ...)` specifies optional parameter name/value pairs for methods other than 'resubstitution', chosen from the following:

- 'weights' — Observation weights.
- 'nsamples' — The number of cross-validation samples (default is 10).
- 'treesize' — Either 'se' (default) to choose the smallest tree whose cost is within one standard error of the minimum cost, or 'min' to choose the minimal cost tree.

## Examples

### Compute the Cost of a Decision Tree

Find the best tree for Fisher's iris data using cross-validation.

Grow a large tree:

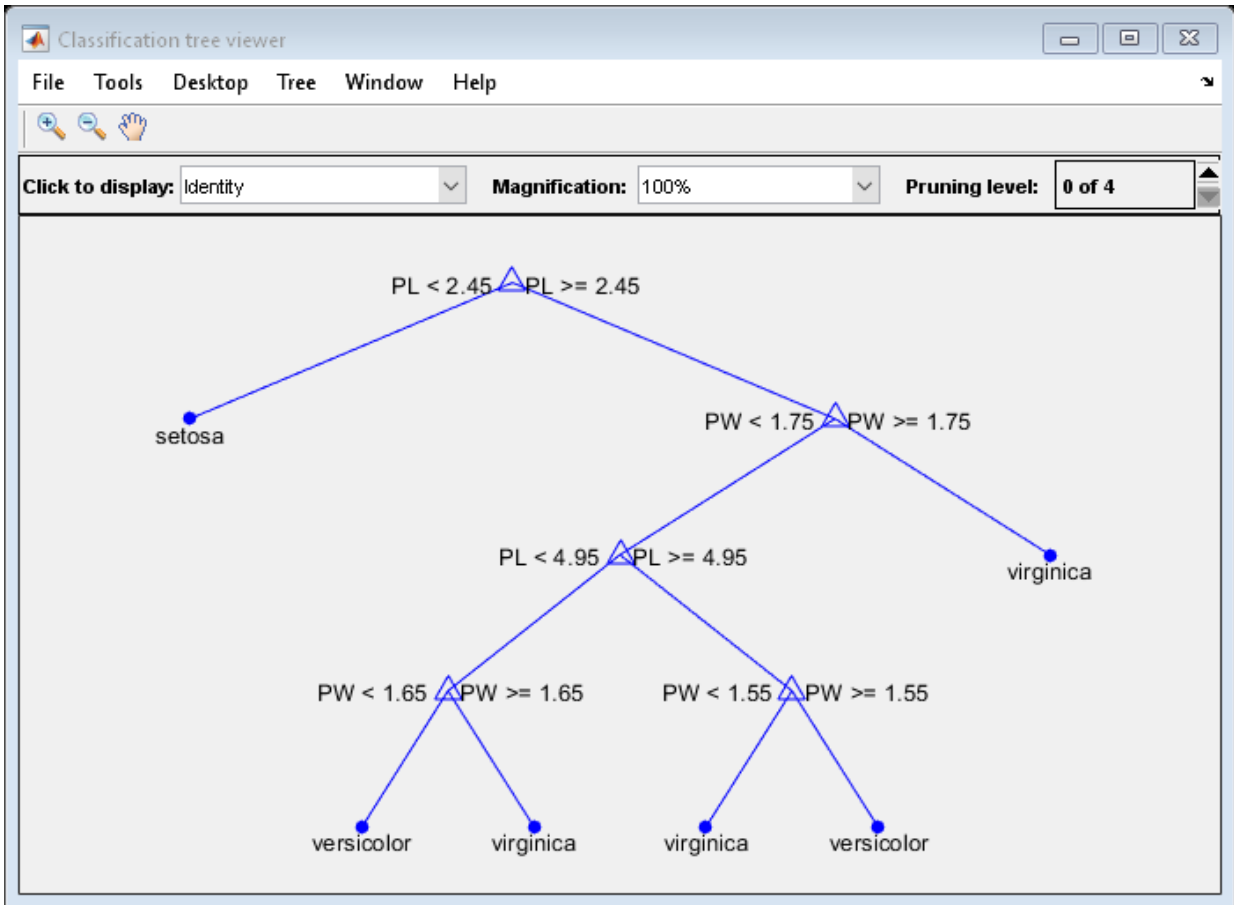
```
load fisheriris;
t = classregtree(meas, species, ...
                'names', {'SL' 'SW' 'PL' 'PW'}, ...
                'minparent', 5)
view(t)
```

Warning: `classregtree` will be removed in a future release. Use `fitctree` or `fitrtree` instead.

t =

Decision tree for classification

```
1  if PL<2.45 then node 2 elseif PL>=2.45 then node 3 else setosa
2  class = setosa
3  if PW<1.75 then node 4 elseif PW>=1.75 then node 5 else versicolor
4  if PL<4.95 then node 6 elseif PL>=4.95 then node 7 else versicolor
5  class = virginica
6  if PW<1.65 then node 8 elseif PW>=1.65 then node 9 else versicolor
7  if PW<1.55 then node 10 elseif PW>=1.55 then node 11 else virginica
8  class = versicolor
9  class = virginica
10 class = virginica
11 class = versicolor
```



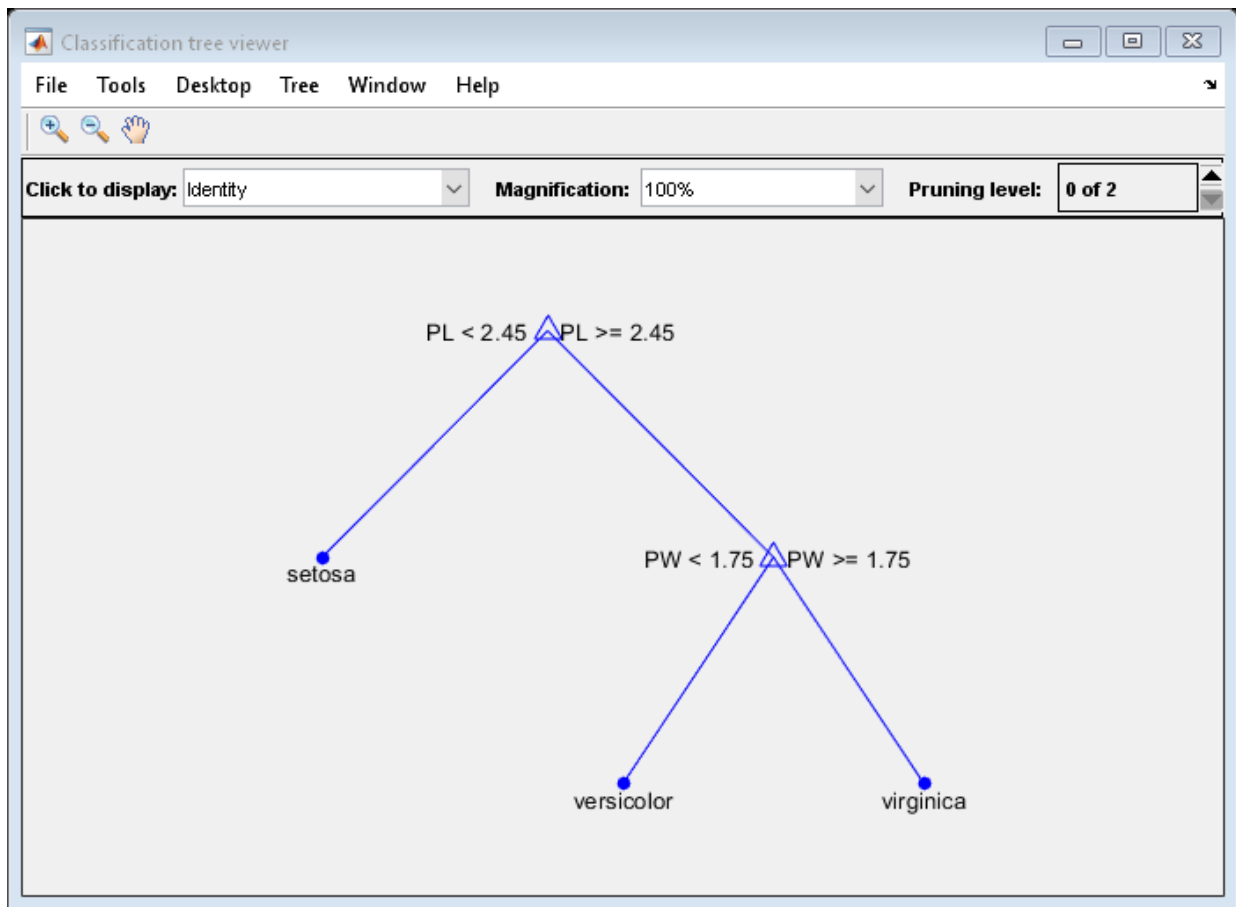
Find the minimum-cost tree:

```
rng(1); % For reproducibility
[c,s,n,best] = test(t,'crossvalidate',meas,species);
tmin = prune(t,'level',best)
view(tmin)
```

Warning: classregtree will be removed in a future release. Use one of the following methods, loss, cvloss, and resubLoss, of an object returned by fitctree or fitctree instead.

Warning: classregtree will be removed in a future release. Use the prune method of an object returned by fitctree or fitctree instead.

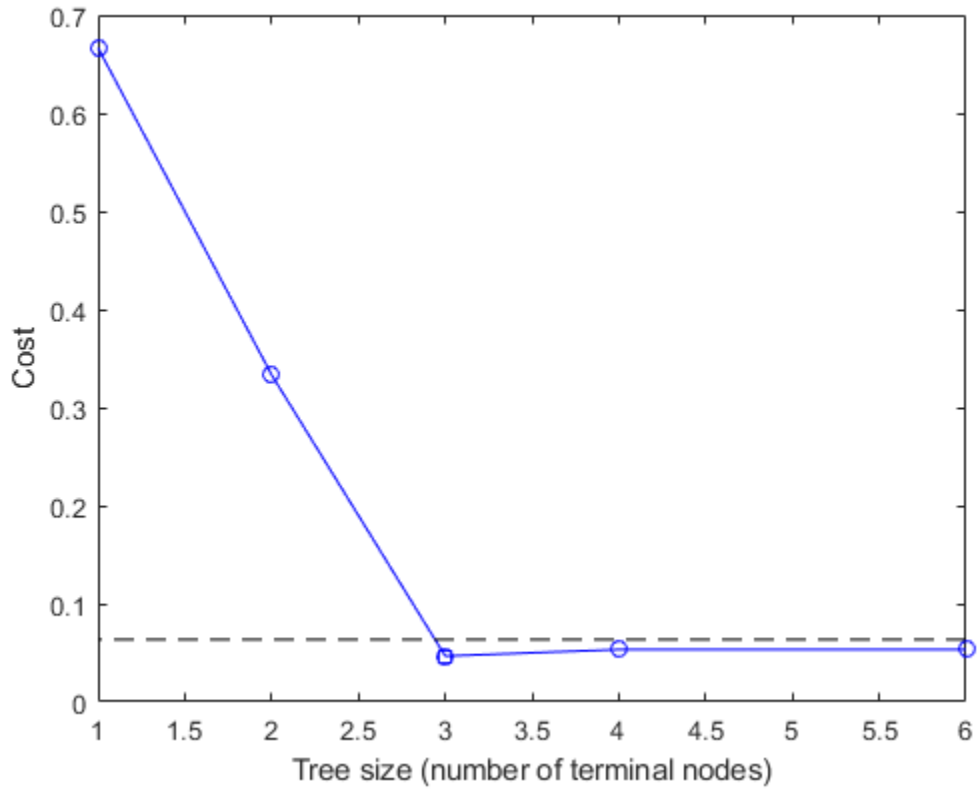
```
tmin =  
  
Decision tree for classification  
1  if PL<2.45 then node 2 elseif PL>=2.45 then node 3 else setosa  
2  class = setosa  
3  if PW<1.75 then node 4 elseif PW>=1.75 then node 5 else versicolor  
4  class = versicolor  
5  class = virginica
```



Plot the smallest tree within one standard error of the minimum cost tree:



```
[mincost,minloc] = min(c);  
plot(n,c,'b-o',...  
      n(best+1),c(best+1),'bs',...  
      n,(mincost+s(minloc))*ones(size(n)),'k--')  
xlabel('Tree size (number of terminal nodes)')  
ylabel('Cost')
```



The solid line shows the estimated cost for each tree size, the dashed line marks one standard error above the minimum, and the square marks the smallest tree under the dashed line.

## References

- [1] Breiman, L., J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Boca Raton, FL: CRC Press, 1984.

## See Also

`classregtree` | `eval` | `prune` | `view`

## test

**Class:** cvpartition

Test indices for cross-validation

## Syntax

```
idx = test(c)
idx = test(c,i)
```

## Description

`idx = test(c)` returns the logical vector `idx` of test indices for an object `c` of the `cvpartition` on page 32-1292 class of type 'holdout' or 'resubstitution'.

If `c.Type` is 'holdout', `idx` specifies the observations in the test set.

If `c.Type` is 'resubstitution', `idx` specifies all observations.

`idx = test(c,i)` returns the logical vector `idx` of test indices for repetition `i` of an object `c` of the `cvpartition` on page 32-1292 class of type 'kfold' or 'leaveout'.

If `c.Type` is 'kfold', `idx` specifies the observations in the test set in fold `i`.

If `c.Type` is 'leaveout', `idx` specifies the observation left out at repetition `i`.

## Examples

Identify the test indices in the first fold of a partition of 10 observations for 3-fold cross-validation:

```
c = cvpartition(10,'kfold',3)
c =
K-fold cross validation partition
      N: 10
```

```
NumTestSets: 3
  TrainSize: 7 6 7
  TestSize: 3 4 3

test(c,1)
ans =
  1
  1
  0
  0
  0
  0
  0
  0
  0
  1
  0
```

## See Also

[cvpartition](#) | [training](#)

# testcholdout

Compare predictive accuracies of two classification models

`testcholdout` statistically assesses the accuracies of two classification models. The function first compares their predicted labels against the true labels, and then it detects whether the difference between the misclassification rates is statistically significant.

You can assess whether the accuracies of the classification models are different, or whether one classification model performs better than another. `testcholdout` can conduct several McNemar test on page 32-6517 variations, including the asymptotic test, the exact-conditional test, and the mid- $p$ -value test. For cost-sensitive assessment on page 32-6515, available tests include a chi-square test (requires an Optimization Toolbox license) and a likelihood ratio test.

## Syntax

```
h = testcholdout(YHat1, YHat2, Y)
h = testcholdout(YHat1, YHat2, Y, Name, Value)
[h, p, e1, e2] = testcholdout( ___ )
```

## Description

`h = testcholdout(YHat1, YHat2, Y)` returns the test decision, by conducting the mid- $p$ -value McNemar test on page 32-6517, from testing the null hypothesis that the predicted class labels `YHat1` and `YHat2` have equal accuracy for predicting the true class labels `Y`. The alternative hypothesis is that the labels have unequal accuracy.

`h = 1` indicates to reject the null hypothesis at the 5% significance level. `h = 0` indicates to not reject the null hypothesis at 5% level.

`h = testcholdout(YHat1, YHat2, Y, Name, Value)` returns the result of the hypothesis test with additional options specified by one or more `Name, Value` pair arguments. For example, you can specify the type of alternative hypothesis, specify the type of test, or supply a cost matrix.

`[h,p,e1,e2] = testcholdout(____)` returns the  $p$ -value for the hypothesis test ( $p$ ) and the respective classification loss on page 32-6520 of each set of predicted class labels ( $e1$  and  $e2$ ) using any of the input arguments in the previous syntaxes.

## Examples

### Compare Accuracies of Two Different Classification Models

Train two classification models using different algorithms. Conduct a statistical test comparing the misclassification rates of the two models on a held-out set.

Load the `ionosphere` data set.

```
load ionosphere;
```

Create a partition that evenly splits the data into training and testing sets.

```
rng(1); % For reproducibility
CVP = cvpartition(Y,'holdout',0.5);
idxTrain = training(CVP); % Training-set indices
idxTest = test(CVP); % Test-set indices
```

CVP is a cross-validation partition object that specifies the training and test sets.

Train an SVM model and an ensemble of 100 bagged classification trees. For the SVM model, specify to use the radial basis function kernel and a heuristic procedure to determine the kernel scale.

```
MdlSVM = fitcsvm(X(idxTrain,:),Y(idxTrain),'Standardize',true,...
    'KernelFunction','RBF','KernelScale','auto');
MdlBag = fitensemble(X(idxTrain,:),Y(idxTrain),'Bag',100,'Tree',...
    'Type','classification');
```

MdlSVM is a trained `ClassificationSVM` model. MdlBag is a trained `ClassificationBaggedEnsemble` model.

Label the test-set observations using the trained models.

```
YhatSVM = predict(MdlSVM,X(idxTest,:));
YhatBag = predict(MdlBag,X(idxTest,:));
```

`YhatSVM` and `YhatBag` are vectors containing the predicted class labels of the respective models.

Test whether the two models have equal predictive accuracies.

```
h = testcholdout(YhatSVM, YhatBag, Y(idxTest))
```

```
h =  
    logical  
    0
```

`h = 0` indicates to not reject the null hypothesis that the two models have equal predictive accuracies.

### Assess Whether One Classification Model Classifies Better Than Another

Train two classification models using the same algorithm, but adjust a hyperparameter to make the algorithm more complex. Conduct a statistical test to assess whether the simpler model has better accuracy in held-out data than the more complex model.

Load the `ionosphere` data set.

```
load ionosphere;
```

Create a partition that evenly splits the data into training and testing sets.

```
rng(1); % For reproducibility  
CVP = cvpartition(Y, 'holdout', 0.5);  
idxTrain = training(CVP); % Training-set indices  
idxTest = test(CVP); % Test-set indices
```

`CVP` is a cross-validation partition object that specifies the training and test sets.

Train two SVM models: one that uses a linear kernel (the default for binary classification) and one that uses the radial basis function kernel. Use the default kernel scale of 1.

```
MdlLinear = fitcsvm(X(idxTrain,:),Y(idxTrain),'Standardize',true);
MdlRBF = fitcsvm(X(idxTrain,:),Y(idxTrain),'Standardize',true,...
    'KernelFunction','RBF');
```

MdlLinear and MdlRBF are trained ClassificationSVM models.

Label the test-set observations using the trained models.

```
YhatLinear = predict(MdlLinear,X(idxTest,:));
YhatRBF = predict(MdlRBF,X(idxTest,:));
```

YhatLinear and YhatRBF are vectors continuing the predicted class labels of the respective models.

Test the null hypothesis that the simpler model (MdlLinear) is at most as accurate as the more complex model (MdlRBF). Because the test-set size is large, conduct the asymptotic McNemar test, and compare the results with the mid-  $p$ -value test (the cost-insensitive testing default). Request to return  $p$ -values and misclassification rates.

```
Asymp = zeros(4,1); % Preallocation
MidP = zeros(4,1);

[Asymp(1),Asymp(2),Asymp(3),Asymp(4)] = testcholdout(YhatLinear,YhatRBF,Y(idxTest),...
    'Alternative','greater','Test','asymptotic');
[MidP(1),MidP(2),MidP(3),MidP(4)] = testcholdout(YhatLinear,YhatRBF,Y(idxTest),...
    'Alternative','greater');
table(Asymp,MidP,'RowNames',{'h' 'p' 'e1' 'e2'})
```

```
ans =
```

```
4x2 table
```

	Asymp	MidP
h	1	1
p	7.2801e-09	2.7649e-10
e1	0.13714	0.13714
e2	0.33143	0.33143

The  $p$ -value is close to zero for both tests, which indicates strong evidence to reject the null hypothesis that the simpler model is less accurate than the more complex model. No



matter what test you specify, `testcholdout` returns the same type of misclassification measure for both models.

### Conduct a Cost-Sensitive Comparison of Two Classification Models

For data sets with imbalanced class representations, or if the false-positive and false-negative costs are imbalanced, you can statistically compare the predictive performances of two classification models by including a cost matrix in the analysis.

Load the `arrhythmia` data set. Determine the class representations in the data.

```
load arrhythmia;
Y = categorical(Y);
tabulate(Y);
```

Value	Count	Percent
1	245	54.20%
2	44	9.73%
3	15	3.32%
4	15	3.32%
5	13	2.88%
6	25	5.53%
7	3	0.66%
8	2	0.44%
9	9	1.99%
10	50	11.06%
14	4	0.88%
15	5	1.11%
16	22	4.87%

There are 16 classes, however some are not represented in the data set. Most observations are classified as not having arrhythmia (class 1). To summarize, the data set is highly discrete with imbalanced classes.

Combine all observations with arrhythmia (classes 2 through 15) into one class. Remove those observations with unknown arrhythmia status from the data set.

```
Y = Y(Y ~= '16');
Y(Y ~= '1') = '2';
X = X(Y ~= '16', :);
```

Create a partition that evenly splits the data into training and testing sets.

```
rng(1); % For reproducibility
CVP = cvpartition(Y, 'holdout', 0.5);
idxTrain = training(CVP); % Training-set indices
idxTest = test(CVP); % Test-set indices
```

CVP is a cross-validation partition object that specifies the training and test sets.

Create a cost matrix such that misclassifying an arrhythmic patient into the no arrhythmia class is five times worse than misclassifying a patient without arrhythmia into the arrhythmia class. Classifying correctly incurs no cost. The rows indicate the true class and the columns indicate predicted class. When conducting a cost-sensitive analysis, it is a good practice to specify the order of the classes.

```
Cost = [0 1; 5 0];
ClassNames = categorical([2 1]);
```

Train two boosting ensembles of 50 classification trees, one that uses AdaBoostM1 and the other that uses LogitBoost. Because there are missing values, specify to use surrogate splits. Train the models using the cost matrix.

```
t = templateTree('Surrogate', 'on');
numTrees = 50;
MdlAda = fitensemble(X(idxTrain, :), Y(idxTrain), 'AdaBoostM1', numTrees, t, ...
    'Cost', Cost, 'ClassNames', ClassNames);
MdlLogit = fitensemble(X(idxTrain, :), Y(idxTrain), 'LogitBoost', numTrees, t, ...
    'Cost', Cost, 'ClassNames', ClassNames);
```

MdlAda and MdlLogit are trained ClassificationEnsemble models.

Label the test-set observations using the trained models.

```
YhatAda = predict(MdlAda, X(idxTest, :));
YhatLogit = predict(MdlLogit, X(idxTest, :));
```

YhatLinear and YhatRBF are vectors containing the predicted class labels of the respective models.

Test whether the AdaBoostM1 ensemble (MdlAda) and the LogitBoost ensemble (MdlLogit) have equal predictive accuracy. Supply the cost matrix. Conduct the asymptotic, likelihood ratio, cost-sensitive test (the default when you pass in a cost matrix). Request to return  $p$ -values and misclassification costs.

```
[h, p, e1, e2] = testcholdout(YhatAda, YhatLogit, Y(idxTest), 'Cost', Cost)
```

```
h =  
  logical  
  0  
  
p =  
  0.2300  
  
e1 =  
  2.0837  
  
e2 =  
  1.9581
```

`h = 0` indicates to not reject the null hypothesis that the two models have equal predictive accuracies.

## Input Arguments

### **`YHat1` — Predicted class labels**

categorical array | character array | logical vector | vector of numeric values | cell array of character vectors

Predicted class labels of the first classification model, specified as a categorical or character array, logical or numeric vector, or cell array of character vectors.

If `YHat1` is a character array, then each element must correspond to one row of the array.

`YHat1`, `YHat2`, and `Y` must have equal lengths.

It is a best practice for `YHat1`, `YHat2`, and `Y` to share the same data type.

**`YHat2` — Predicted class labels**

categorical array | character array | logical vector | vector of numeric values | cell array of character vectors

Predicted class labels of the second classification model, specified as a categorical or character array, logical or numeric vector, or cell array of character vectors.

If `YHat2` is a character array, then each element must correspond to one row of the array.

`YHat1`, `YHat2`, and `Y` must have equal lengths.

It is a best practice for `YHat1`, `YHat2`, and `Y` to share the same data type.

**`Y` — True class labels**

categorical array | character array | logical vector | vector of numeric values | cell array of character vectors

True class labels, specified as a categorical or character array, logical or numeric vector, or cell array of character vectors.

If `Y` is a character array, then each element must correspond to one row of the array.

`YHat1`, `YHat2`, and `Y` must have equal lengths.

It is a best practice for `YHat1`, `YHat2`, and `Y` to share the same data type.

Data Types: categorical | char | logical | single | double | cell

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

Example: `'Alternative', 'greater', 'Test', 'asymptotic', 'Cost', [0 2;1 0]` specifies to test whether the first set of first predicted class labels is more accurate than the second set, to conduct the asymptotic McNemar test, and to penalize misclassifying observations with the true label `ClassNames{1}` twice as much as for misclassifying observations with the true label `ClassNames{2}`.

**Alpha — Hypothesis test significance level**

0.05 (default) | scalar value in the interval (0,1)

Hypothesis test significance level, specified as the comma-separated pair consisting of 'Alpha' and a scalar value in the interval (0,1).

Example: 'Alpha',0.1

Data Types: single | double

**Alternative — Alternative hypothesis to assess**

'unequal' (default) | 'greater' | 'less'

Alternative hypothesis to assess, specified as the comma-separated pair consisting of 'Alternative' and one of the values listed in the table.

Value	Alternative hypothesis
'unequal' (default)	For predicting Y, YHat1 and YHat2 have unequal accuracies.
'greater'	For predicting Y, YHat1 is more accurate than YHat2.
'less'	For predicting Y, YHat1 is less accurate than YHat2.

Example: 'Alternative', 'greater'

Data Types: char

**ClassNames — Class names**

categorical vector | cell array of character vectors | character array | logical vector | numeric vector

Class names, specified as the comma-separated pair consisting of 'ClassNames' and a categorical or character array, logical or numeric vector, or cell array of character vectors. You must set ClassNames using the data type of Y.

If ClassNames is a character array, then each element must correspond to one *row* of the array.

Use ClassNames to:

- Specify the order of any input argument dimension that corresponds to class order. For example, use ClassNames to specify the order of the dimensions of Cost.

- Select a subset of classes for testing. For example, suppose that the set of all distinct class names in  $Y$  is `{'a', 'b', 'c'}`. To train and test models using observations from classes 'a' and 'c' only, specify `'ClassNames', {'a', 'c'}`.

The default is the set of all distinct class names in  $Y$ .

Example: `'ClassNames', {'b', 'g'}`

Data Types: `categorical | char | logical | single | double | cell`

### **Cost** — Misclassification cost

`square matrix | structure array`

Misclassification cost, specified as the comma-separated pair consisting of `'Cost'` and a square matrix or structure array. If you specify:

- If you specify the square matrix `Cost`, then `Cost(i, j)` is the cost of classifying a point into class  $j$  if its true class is  $i$ . That is, the rows correspond to the true class and the columns correspond to the predicted class. To specify the class order for the corresponding rows and columns of `Cost`, additionally specify the `ClassNames` name-value pair argument.
- If you specify the structure `S`, then `S` must have two fields:
  - `S.ClassNames`, which contains the class names as a variable of the same data type as  $Y$ . You can use this field to specify the order of the classes.
  - `S.ClassificationCosts`, which contains the cost matrix, with rows and columns ordered as in `S.ClassNames`

If you specify `Cost`, then `testcholdout` cannot conduct one-sided, exact, or mid- $p$  tests. You must also specify `'Alternative', 'unequal', 'Test', 'asymptotic'`. For cost-sensitive testing options, see the `CostTest` name-value pair argument.

It is a best practice to supply the same cost matrix used to train the classification models.

The default is `Cost(i, j) = 1` if  $i \neq j$ , and `Cost(i, j) = 0` if  $i = j$ .

Example: `'Cost', [0 1 2 ; 1 0 2; 2 2 0]`

Data Types: `double | single | struct`

### **CostTest** — Cost-sensitive test type

`'likelihood' (default) | 'chisquare'`

Cost-sensitive test type, specified as the comma-separated pair consisting of 'CostTest' and 'chisquare' or 'likelihood'. Unless you specify a cost matrix using the Cost name-value pair argument, testcholdout ignores CostTest.

This table summarizes the available options for cost-sensitive testing.

Value	Asymptotic test type	Requirements
'chisquare'	Chi-square test	Optimization Toolbox license to implement quadprog
'likelihood'	Likelihood ratio test	None

For more details, see “Cost-Sensitive Testing” on page 32-6515.

Example: 'CostTest', 'chisquare'

Data Types: char

#### Test — Test to conduct

'asymptotic' | 'exact' | 'midp'

Test to conduct, specified as the comma-separated pair consisting of 'Test' and 'asymptotic', 'exact', and 'midp'. This table summarizes the available options for cost-insensitive testing.

Value	Description
'asymptotic'	Asymptotic McNemar test
'exact'	Exact-conditional McNemar test
'midp' (default)	Mid- <i>p</i> -value McNemar test

For more details, see “McNemar Tests” on page 32-6517.

For cost-sensitive testing, Test must be 'asymptotic'. When you specify the Cost name-value pair argument, and choose a cost-sensitive test using the CostTest name-value pair argument, 'asymptotic' is the default.

Example: 'Test', 'asymptotic'

Data Types: char

**Note** NaNs, <undefined> values, and empty character vectors ( ' ' ) indicate missing values. `testcholdout`:

- Treats missing values in `YHat1` and `YHat2` as misclassified observations.
  - Removes missing values in `Y` and the corresponding values of `YHat1` and `YHat2`
- 

## Output Arguments

### **h** — Hypothesis test result

1 | 0

Hypothesis test result, returned as a logical value.

`h = 1` indicates the rejection of the null hypothesis at the Alpha significance level.

`h = 0` indicates failure to reject the null hypothesis at the Alpha significance level.

### **p** — *p*-value

scalar in the interval [0,1]

*p*-value of the test, returned as a scalar in the interval [0,1]. *p* is the probability that a random test statistic is at least as extreme as the observed test statistic, given that the null hypothesis is true.

`testcholdout` estimates *p* using the distribution of the test statistic, which varies with the type of test. For details on test statistics derived from the available variants of the McNemar test, see “McNemar Tests” on page 32-6517. For details on test statistics derived from cost-sensitive tests, see “Cost-Sensitive Testing” on page 32-6515.

### **e1** — Classification loss

scalar

Classification loss on page 32-6520 that summarizes the accuracy of the first set of class labels (`YHat1`) predicting the true class labels (`Y`), returned as a scalar.

For cost-insensitive testing, `e1` is the misclassification rate. That is, `e1` is the proportion of misclassified observations, which is a scalar in the interval [0,1].



For cost-sensitive testing,  $e_1$  is the misclassification cost. That is,  $e_1$  is the weighted average of the misclassification costs, in which the weights are the respective estimated proportions of misclassified observations.

### **$e_2$ — Classification loss**

scalar

Classification loss on page 32-6520 that summarizes the accuracy of the second set of class labels ( $Y_{\text{Hat}2}$ ) predicting the true class labels ( $Y$ ), returned as a scalar.

For cost-insensitive testing,  $e_2$  is the misclassification rate. That is,  $e_2$  is the proportion of misclassified observations, which is a scalar in the interval  $[0,1]$ .

For cost-sensitive testing,  $e_2$  is the misclassification cost. That is,  $e_2$  is the weighted average of the costs of misclassification, in which the weights are the respective estimated proportions of misclassified observations.

## Definitions

### Cost-Sensitive Testing

Conduct cost-sensitive testing when the cost of misclassification is imbalanced. When conducting a cost-sensitive analysis, you can account for the cost imbalance in training the classification models, and then in statistically comparing them.

If the cost of misclassification is imbalanced, then the misclassification rate tends to be a poorly performing classification loss on page 32-6520. Use misclassification cost instead to compare classification models.

Misclassification costs are often unbalanced in applications. For example, consider classifying subjects based on a set of predictors into two categories: healthy and sick. Misclassifying a sick subject as healthy poses a danger to the subject's life. However, misclassifying a healthy subject as sick can cause some inconvenience, but does not pose any danger. In this situation, you assign misclassification costs such that misclassifying a sick subject as healthy is more costly than misclassifying a healthy subject as sick.

The definitions that follow summarize the cost-sensitive tests. In the definitions:

- $n_{ijk}$  and  $\hat{\pi}_{ijk}$  are the number and estimated proportion of test-sample observations with true class  $k$  that the first classification model assigns label  $i$ . The second

classification model assigns label  $j$ . The unknown, true value of  $\hat{\pi}_{ijk}$  is  $\pi_{ijk}$ . The test-

set sample size is  $\sum_{i,j,k} n_{ijk} = n_{test}$ .  $\sum_{i,j,k} \pi_{ijk} = \sum_{i,j,k} \pi_{ijk} = 1$ .

- $c_{ij}$  is the relative cost of assigning label  $j$  to an observation with true class  $i$ .  $c_{ii} = 0$ ,  $c_{ij} \geq 0$ , and, for at least one  $(i,j)$  pair,  $c_{ij} > 0$ .
- All subscripts take on integer values from 1 through  $K$ , which is the number of classes.
- The expected difference in the misclassification costs of the two classification models is

$$\delta = \sum_{i=1}^K \sum_{j=1}^K \sum_{k=1}^K (c_{ki} - c_{kj}) \pi_{ijk}.$$

- The hypothesis test is

$$H_0 : \delta = 0$$

$$H_1 : \delta \neq 0$$

The available cost-sensitive tests are appropriate for two-tailed testing.

Available, asymptotic tests that address imbalanced costs are a chi-square test and a likelihood ratio test.

- Chi-square test — The chi-square test statistic is based on the Pearson and Neyman chi-square test statistics, but with a Laplace correction factor to account for any  $n_{ijk} = 0$ . The test statistic is

$$t_{\chi^2}^* = \sum_{i \neq j} \sum_k \frac{(n_{ijk} + 1 - (n_{test} + K^3) \hat{\pi}_{ijk}^{(1)})^2}{n_{ijk} + 1}.$$

If  $1 - F_{\chi^2}(t_{\chi^2}^*; 1) < \alpha$ , then reject  $H_0$ .

•

$\hat{\pi}_{ijk}^{(1)}$  are estimated by minimizing  $t_{\chi^2}^*$  under the constraint that  $\delta = 0$ .

•

$F_{\chi^2}(x; 1)$  is the  $\chi^2$  C.D.F. with one degree of freedom evaluated at  $x$ .

- Likelihood ratio test — The likelihood ratio test is based on  $N_{ijk}$ , which are binomial random variables having sample size  $n_{test}$  and success probability  $\pi_{ijk}$ . They represent

the random number of observations with true class  $k$  that the first classification model assigns label  $i$ . The second classification model assigns label  $j$ . Jointly, their distribution is multinomial.

The test statistic is

$$t_{LRT}^* = 2 \log \left[ \frac{P\left(\bigcap_{i,j,k} N_{ijk} = n_{ijk}; n_{test}, \pi_{ijk} = \pi_{ijk}^{(2)}\right)}{P\left(\bigcap_{i,j,k} N_{ijk} = n_{ijk}; n_{test}, \pi_{ijk} = \pi_{ijk}^{(3)}\right)} \right].$$

If  $1 - F_{\chi^2}(t_{LRT}^*; 1) < \alpha$ , then reject  $H_0$ .

•

$\hat{\pi}_{ijk}^{(2)} = \frac{n_{ijk}}{n_{test}}$  is the unrestricted MLE of  $\pi_{ijk}$ .

•

$\hat{\pi}_{ijk}^{(3)} = \frac{n_{ijk}}{n_{test} + \lambda(c_{ki} - c_{kj})}$  is the MLE under the null hypothesis that  $\delta = 0$ .  $\lambda$  is the solution to

$$\sum_{i,j,k} \frac{n_{ijk}(c_{ki} - c_{kj})}{n_{test} + \lambda(c_{ki} - c_{kj})} = 0.$$

•

$F_{\chi^2}(x; 1)$  is the  $\chi^2$  C.D.F. with one degree of freedom evaluated at  $x$ .

## McNemar Tests

McNemar Tests are hypothesis tests that compare two population proportions while addressing the issues resulting from two dependent, matched-pair samples.

One way to compare the predictive accuracies of two classification models is:

- 1 Partition the data into training and test sets.
- 2 Train both classification models using the training set.
- 3 Predict class labels using the test set.
- 4 Summarize the results in a two-by-two table resembling this figure.

		Model 2		
		Correct	Incorrect	
Model 1	Correct	$n_{11}$	$n_{12}$	$n_{1\bullet}$
	Incorrect	$n_{21}$	$n_{22}$	$n_{2\bullet}$
		$n_{\bullet 1}$	$n_{\bullet 2}$	$n_{test}$

$n_{ii}$  are the number of concordant pairs, that is, the number of observations that both models classify the same way (correctly or incorrectly).  $n_{ij}$ ,  $i \neq j$ , are the number of discordant pairs, that is, the number of observations that models classify differently (correctly or incorrectly).

The misclassification rates for Models 1 and 2 are

$$\hat{\pi}_{2\bullet} = n_{2\bullet} / n$$

and  $\hat{\pi}_{\bullet 2} = n_{\bullet 2} / n$ , respectively. A two-sided test for comparing the accuracy of the two models is

$$H_0 : \pi_{\bullet 2} = \pi_{2\bullet}$$

$$H_1 : \pi_{\bullet 2} \neq \pi_{2\bullet}$$

The null hypothesis suggests that the population exhibits marginal homogeneity, which

reduces the null hypothesis to  $H_0 : \pi_{12} = \pi_{21}$ . Also, under the null hypothesis,  $N_{12} \sim \text{Binomial}(n_{12} + n_{21}, 0.5)$  [1].

These facts are the basis for these, available McNemar test variants: the asymptotic, exact-conditional and mid-p-value McNemar tests. The definitions that follow summarize the available variants.

- Asymptotic — The asymptotic McNemar test statistics and rejection regions (for significance-level  $\alpha$ ) are:

- For one-sided tests, the test statistic

$$t_{a1}^* = \frac{n_{12} - n_{21}}{\sqrt{n_{12} + n_{21}}}.$$

If  $1 - \Phi\left(\left|t_1^*\right|\right) < \alpha$ , where  $\Phi$  is the standard Gaussian C.D.F., then reject  $H_0$ .

- For two-sided tests, the test statistic

$$t_{a2}^* = \frac{(n_{12} - n_{21})^2}{n_{12} + n_{21}}.$$

If  $1 - F_{\chi^2}(t_2^*; m) < \alpha$ , where  $F_{\chi^2}(x; m)$  is the  $\chi_m^2$  C.D.F. evaluated at  $x$ , then reject  $H_0$ .

This variant requires large-sample theory, specifically, the Gaussian approximation to the binomial distribution. Therefore:

- The total number of discordant pairs,  $n_d = n_{12} + n_{21}$  must be greater than 10 ([1], Ch. 10.1.4).
- In general, asymptotic tests do not guarantee nominal coverage. The observed probability of falsely rejecting the null hypothesis can exceed  $\alpha$ . Simulation studies in [18] suggest this, but the asymptotic McNemar test performs well in terms of statistical power.
- Exact Conditional — The exact-conditional McNemar test statistics and rejection regions (for significance-level  $\alpha$ ) are ([36], [38]):

- For one-sided tests, the test statistic

$$t_1^* = n_{12}$$

If  $F_{\text{Bin}}(t_1^*; n_d, 0.5) < \alpha$ , where  $F_{\text{Bin}}(x; n, p)$  is the binomial C.D.F. with sample size  $n$  and success probability  $p$  evaluated at  $x$ , then reject  $H_0$ .

- For two-sided tests, the test statistic

$$t_2^* = \min(n_{12}, n_{21})$$

If  $F_{\text{Bin}}(t_2^*; n_d, 0.5) < \alpha / 2$ , then reject  $H_0$ .

The exact conditional test always attains nominal coverage. Simulation studies in [18] suggest that the test is conservative, and then show that the test lacks statistical power compared to other variants. For small or highly discrete test samples, consider using the mid- $p$ -value test ([1], Ch. 3.6.3). For details, see `Test` and “McNemar Tests” on page 32-6517.

- Mid- $p$ -value test — The mid- $p$ -value McNemar test statistics and rejection regions (for significance-level  $\alpha$ ) are ([32]):

- For one-sided tests, the test statistic

$$t_1^* = n_{12}$$

If  $F_{\text{Bin}}(t_1^* - 1; n_{12} + n_{21}, 0.5) + 0.5f_{\text{Bin}}(t_1^*; n_{12} + n_{21}, 0.5) < \alpha$ , where  $F_{\text{Bin}}(x; n, p)$  and  $f_{\text{Bin}}(x; n, p)$  are the binomial C.D.F. and P.D.F, respectively, with sample size  $n$  and success probability  $p$  evaluated at  $x$ , then reject  $H_0$ .

- For two-sided tests, the test statistic

$$t_2^* = \min(n_{12}, n_{21})$$

If  $F_{\text{Bin}}(t_2^* - 1; n_{12} + n_{21} - 1, 0.5) + 0.5f_{\text{Bin}}(t_2^*; n_{12} + n_{21}, 0.5) < \alpha / 2$ , then reject  $H_0$ .

The mid- $p$ -value test addresses the over-conservative behavior of the exact conditional test. The simulation studies in [18] demonstrate that this test attains nominal coverage, and has good statistical power.

## Classification Loss

Classification losses indicate the accuracy of a classification model or set of predicted labels. Two classification losses are misclassification rate and cost.

`testcholdout` returns the classification losses (see `e1` and `e2`) under the alternative hypothesis (i.e., the unrestricted classification losses).  $n_{ijk}$  is the number of test-sample observations with true class  $k$  that the first classification model assigns label  $i$  and the second classification model assigns label  $j$ , and the corresponding estimated proportion is

$\hat{\pi}_{ijk} = \frac{n_{ijk}}{n_{test}}$ . The test-set sample size is  $\sum_{i,j,k} n_{ijk} = n_{test}$ . The indices are taken from 1 through  $K$ , the number of classes.

- Misclassification rate, or classification error, is a scalar in the interval  $[0,1]$  representing the proportion of misclassified observations. That is, the misclassification rate for the first classification model is

$$e_1 = \sum_{j=1}^K \sum_{k=1}^K \sum_{i \neq k} \hat{\pi}_{ijk}.$$

For the misclassification rate of the second classification model ( $e_2$ ), switch the indices  $i$  and  $j$  in the formula.

Classification accuracy decreases as the misclassification rate increases to 1.

- Misclassification cost is a nonnegative scalar and is a measure of classification quality relative to the values the specified cost matrix elements. Its interpretation depends on the specified costs of misclassification. Misclassification cost is the weighted average of the costs of misclassification (specified in a cost matrix,  $C$ ) in which the weights are the respective, estimated proportions of misclassified observations. The misclassification cost for the first classification model is

$$e_1 = \sum_{j=1}^K \sum_{k=1}^K \sum_{i \neq k} \hat{\pi}_{ijk} c_{ki},$$

where  $c_{kj}$  is the cost of classifying an observation into class  $j$  if its true class is  $k$ . For the misclassification cost of the second classification model ( $e_2$ ), switch the indices  $i$  and  $j$  in the formula.

In general, for a fixed cost matrix, classification accuracy decreases as misclassification cost increases.

## Tips

- It is a good practice to obtain predicted class labels by passing any trained classification model and new predictor data to the `predict` method. For example, for predicted labels from an SVM model, see `predict`.
- Cost-sensitive tests perform numerical optimization, which requires additional computational resources. The likelihood ratio test conducts numerical optimization indirectly by finding the root of a Lagrange multiplier in an interval. For some data sets, if the root lies close to the boundaries of the interval, then the method can fail. Therefore, if you have an Optimization Toolbox license, consider conducting the cost-sensitive chi-square test instead. For more details, see `CostTest` and “Cost-Sensitive Testing” on page 32-6515.

## References

- [1] Agresti, A. *Categorical Data Analysis*, 2nd Ed. John Wiley & Sons, Inc.: Hoboken, NJ, 2002.
- [2] Fagerlan, M.W., S Lydersen, P. Laake. “The McNemar Test for Binary Matched-Pairs Data: Mid-p and Asymptotic Are Better Than Exact Conditional.” *BMC Medical Research Methodology*. Vol. 13, 2013, pp. 1–8.
- [3] Lancaster, H.O. “Significance Tests in Discrete Distributions.” *JASA*, Vol. 56, Number 294, 1961, pp. 223–234.
- [4] McNemar, Q. “Note on the Sampling Error of the Difference Between Correlated Proportions or Percentages.” *Psychometrika*, Vol. 12, Number 2, 1947, pp. 153–157.
- [5] Mosteller, F. “Some Statistical Problems in Measuring the Subjective Response to Drugs.” *Biometrics*, Vol. 8, Number 3, 1952, pp. 220–226.

## See Also

testckfold

## Topics

“Hypothesis Tests”

**Introduced in R2015a**



## testckfold

Compare accuracies of two classification models by repeated cross validation

`testckfold` statistically assesses the accuracies of two classification models by repeatedly cross validating the two models, determining the differences in the classification loss, and then formulating the test statistic by combining the classification loss differences. This type of test is particularly appropriate when sample size is limited.

You can assess whether the accuracies of the classification models are different, or whether one classification model performs better than another. Available tests include a 5-by-2 paired  $t$  test, a 5-by-2 paired  $F$  test, and a 10-by-10 repeated cross-validation  $t$  test. For more details, see “Repeated Cross-Validation Tests” on page 32-6546. To speed up computations, `testckfold` supports parallel computing (requires a Parallel Computing Toolbox license).

## Syntax

```
h = testckfold(C1,C2,X1,X2)
h = testckfold(C1,C2,X1,X2,Y)
h = testckfold( ___,Name,Value)
[h,p,e1,e2] = testckfold( ___)
```

## Description

`h = testckfold(C1,C2,X1,X2)` returns the test decision that results from conducting a 5-by-2 paired  $F$  cross-validation test. The null hypothesis is the classification models `C1` and `C2` have equal accuracy in predicting the true class labels using the predictor and response data in the tables `X1` and `X2`. `h = 1` indicates to reject the null hypothesis at the 5% significance level.

`testckfold` conducts the cross-validation test by applying `C1` and `C2` to all predictor variables in `X1` and `X2`, respectively. The true class labels in `X1` and `X2` must be the same. The response variable names in `X1`, `X2`, `C1.ResponseName`, and `C2.ResponseName` must be the same.

For examples of ways to compare models, see “Tips” on page 32-6550.

`h = testckfold(C1,C2,X1,X2,Y)` applies the full classification model or classification templates `C1` and `C2` to all predictor variables in the tables or matrices of data `X1` and `X2`, respectively. `Y` is the table variable name corresponding to the true class labels, or an array of true class labels.

`h = testckfold(____,Name,Value)` uses any of the input arguments in the previous syntaxes and additional options specified by one or more `Name, Value` pair arguments. For example, you can specify the type of alternative hypothesis, the type of test, or the use of parallel computing.

`[h,p,e1,e2] = testckfold(____)` also returns the  $p$ -value for the hypothesis test (`p`) and the respective classification losses on page 32-6549 for each cross-validation run and fold (`e1` and `e2`).

## Examples

### Compare Classification Tree Predictor-Selection Algorithms

At each node, `fitctree` chooses the best predictor to split using an exhaustive search by default. Alternatively, you can choose to split the predictor that shows the most evidence of dependence with the response by conducting curvature tests. This example statistically compares classification trees grown via exhaustive search for the best splits and grown by conducting curvature tests with interaction.

Load the `census1994` data set.

```
load census1994.mat
rng(1) % For reproducibility
```

Grow a default classification tree using the training set, `adultdata`, which is a table. The response-variable name is `'salary'`.

```
C1 = fitctree(adultdata,'salary')
```

```
C1 =
```

```
ClassificationTree
```

```

    PredictorNames: {1x14 cell}
    ResponseName: 'salary'
CategoricalPredictors: [2 4 6 7 8 9 10 14]
    ClassNames: [<=50K    >50K]
    ScoreTransform: 'none'
NumObservations: 32561

```

C1 is a full ClassificationTree model. Its ResponseName property is 'salary'. C1 uses an exhaustive search to find the best predictor to split on based on maximal splitting gain.

Grow another classification tree using the same data set, but specify to find the best predictor to split using the curvature test with interaction.

```
C2 = fitctree(adultdata, 'salary', 'PredictorSelection', 'interaction-curvature')
```

```
C2 =
```

```

ClassificationTree
    PredictorNames: {1x14 cell}
    ResponseName: 'salary'
CategoricalPredictors: [2 4 6 7 8 9 10 14]
    ClassNames: [<=50K    >50K]
    ScoreTransform: 'none'
NumObservations: 32561

```

C2 also is a full ClassificationTree model with ResponseName equal to 'salary'.

Conduct a 5-by-2 paired  $F$  test to compare the accuracies of the two models using the training set. Because the response-variable names in the data sets and the ResponseName properties are all equal, and the response data in both sets are equal, you can omit supplying the response data.

```
h = testckfold(C1,C2,adultdata,adultdata)
```

```
h =
```

```
logical
```

0

$h = 0$  indicates to not reject the null hypothesis that C1 and C2 have the same accuracies at 5% level.

### Compare Accuracies of Two Different Classification Models

Conduct a statistical test comparing the misclassification rates of the two models using a 5-by-2 paired  $F$  test.

Load Fisher's iris data set.

```
load fisheriris;
```

Create a naive Bayes template and a classification tree template using default options.

```
C1 = templateNaiveBayes;  
C2 = templateTree;
```

C1 and C2 are template objects corresponding to the naive Bayes and classification tree algorithms, respectively.

Test whether the two models have equal predictive accuracies. Use the same predictor data for each model. `testckfold` conducts a 5-by-2, two-sided, paired  $F$  test by default.

```
rng(1); % For reproducibility  
h = testckfold(C1,C2,meas,meas,species)
```

$h =$

```
logical
```

```
0
```

$h = 0$  indicates to not reject the null hypothesis that the two models have equal predictive accuracies.

## Compare Classification Accuracies of Simple and Complex Models

Conduct a statistical test to assess whether a simpler model has better accuracy than a more complex model using a 10-by-10 repeated cross-validation *t* test.

Load Fisher's iris data set. Create a cost matrix that penalizes misclassifying a setosa iris twice as much as misclassifying a virginica iris as a versicolor.

```
load fisheriris;
tabulate(species)
Cost = [0 2 2; 2 0 1; 2 1 0];
ClassNames = {'setosa' 'versicolor' 'virginica'};...
    % Specifies the order of the rows and columns in Cost
```

Value	Count	Percent
setosa	50	33.33%
versicolor	50	33.33%
virginica	50	33.33%

The empirical distribution of the classes is uniform, and the classification cost is slightly imbalanced.

Create two ECOC templates: one that uses linear SVM binary learners and one that uses SVM binary learners equipped with the RBF kernel.

```
tSVMLinear = templateSVM('Standardize',true); % Linear SVM by default
tSVMRBF = templateSVM('KernelFunction','RBF','Standardize',true);
C1 = templateECOC('Learners',tSVMLinear);
C2 = templateECOC('Learners',tSVMRBF);
```

C1 and C2 are ECOC template objects. C1 is prepared for linear SVM. C2 is prepared for SVM with an RBF kernel training.

Test the null hypothesis that the simpler model (C1) is at most as accurate as the more complex model (C2) in terms of classification costs. Conduct the 10-by-10 repeated cross-validation test. Request to return *p*-values and misclassification costs.

```
rng(1); % For reproducibility
[h,p,e1,e2] = testckfold(C1,C2,meas,meas,species,...
    'Alternative','greater','Test','10x10t','Cost',Cost,...
    'ClassNames',ClassNames)
```

```
h =
```

logical

0

p =

0.1077

e1 =

Columns 1 through 7

0	0	0	0.0667	0	0.0667	0.1333
0.0667	0.0667	0	0	0	0	0.0667
0	0	0	0	0	0.0667	0.0667
0.0667	0.0667	0	0.0667	0	0.0667	0
0.0667	0.0667	0.0667	0	0.0667	0.0667	0
0	0	0.1333	0	0	0.0667	0
0.0667	0.0667	0	0	0.0667	0	0
0.0667	0	0.0667	0.0667	0	0.1333	0
0	0.0667	0.1333	0.0667	0.0667	0	0
0	0.0667	0.0667	0.0667	0.0667	0	0

Columns 8 through 10

0	0.1333	0
0	0.0667	0.0667
0.0667	0.0667	0.0667
0	0.0667	0
0	0	0
0	0.0667	0.0667
0.0667	0	0.0667
0.0667	0	0
0	0	0
0.0667	0	0

e2 =

Columns 1 through 7

0	0	0	0.1333	0	0.0667	0.1333
0.0667	0.0667	0	0.1333	0	0	0
0.1333	0.1333	0	0	0	0.0667	0
0	0.1333	0	0.0667	0.1333	0.1333	0
0.0667	0.0667	0.0667	0	0.0667	0.1333	0.1333
0.0667	0	0.0667	0.0667	0	0.0667	0.1333
0.2000	0.0667	0	0	0.0667	0	0
0.2000	0	0	0.1333	0	0.1333	0
0	0.0667	0.0667	0.0667	0.1333	0	0.2000
0.0667	0.0667	0	0.0667	0.1333	0	0

Columns 8 through 10

0	0.2667	0
0.1333	0.1333	0.0667
0.0667	0.0667	0.0667
0	0.0667	0
0	0	0.0667
0	0.0667	0.0667
0.1333	0	0.0667
0.0667	0	0
0	0	0
0.0667	0.1333	0.0667

The  $p$ -value is slightly greater than 0.10, which indicates to retain the null hypothesis that the simpler model is at most as accurate as the more complex model. This result is consistent for any significance level ( $\text{Alpha}$ ) that is at most 0.10.

$e_1$  and  $e_2$  are 10-by-10 matrices containing misclassification costs. Row  $r$  corresponds to run  $r$  of the repeated cross validation. Column  $k$  corresponds to test-set fold  $k$  within a particular cross-validation run. For example, element (2,4) of  $e_2$  is 0.1333. This value means that in cross-validation run 2, when the test set is fold 4, the estimated test-set misclassification cost is 0.1333.

### Select Features Using Statistical Accuracy Comparison

Reduce classification model complexity by selecting a subset of predictor variables (features) from a larger set. Then, statistically compare the accuracy between the two models.

Load the ionosphere data set.

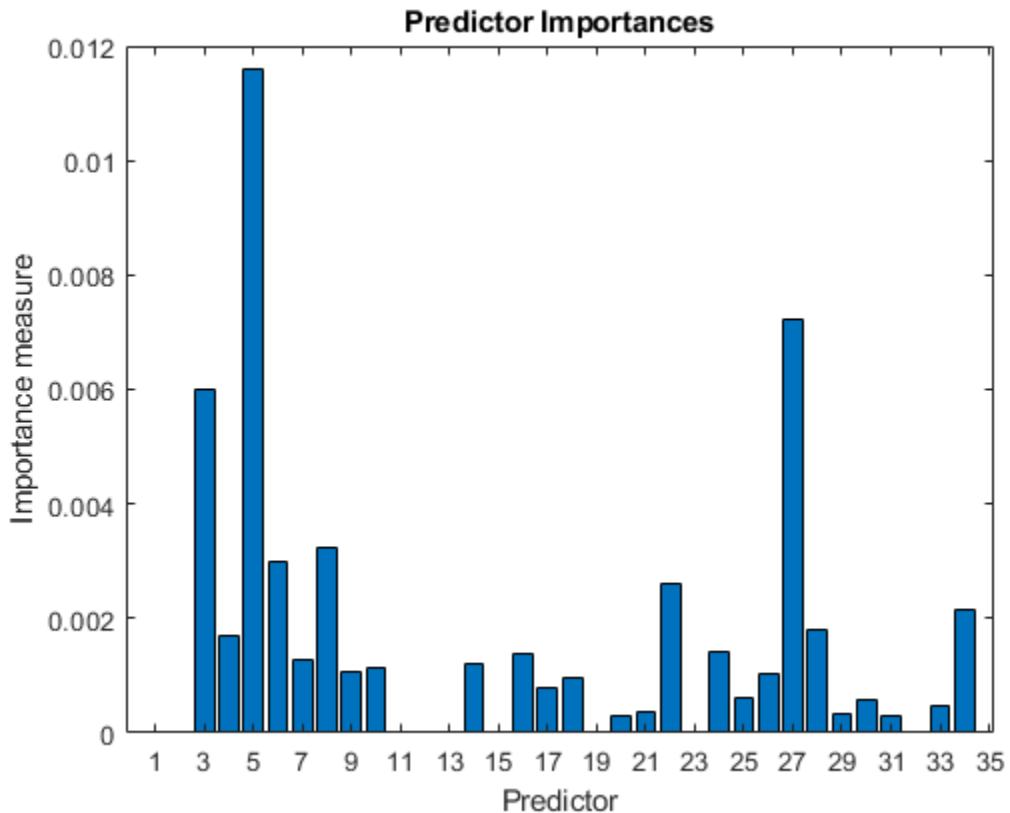
```
load ionosphere;
```

Train an ensemble of 100 boosted classification trees using AdaBoostM1 and the entire set of predictors. Inspect the importance measure for each predictor.

```
t = templateTree('MaxNumSplits',1); % Weak-learner template tree object
C = fitcensemble(X,Y,'Method','AdaBoostM1','Learners',t);
predImp = predictorImportance(C);

figure;
bar(predImp);
h = gca;
h.XTick = 1:2:h.XLim(2);
title('Predictor Importances');
xlabel('Predictor');
ylabel('Importance measure');
```





Identify the top five predictors in terms of their importance.

```
[~,idxSort] = sort(predImp, 'descend');
idx5 = idxSort(1:5);
```

Test whether the two models have equal predictive accuracies. Specify the reduced data set and then the full predictor data. Use parallel computing to speed up computations.

```
s = RandStream('mlfg6331_64');
Options = statset('UseParallel',true,'Streams',s,'UseSubstreams',true);
```

```
[h,p,e1,e2] = testckfold(C,C,X(:,idx5),X,Y,'Options',Options)
```

```
Starting parallel pool (parpool) using the 'local' profile ...
connected to 12 workers.
```

```
h = logical
    0

p = 0.4161

e1 =
    0.0686    0.0795
    0.0800    0.0625
    0.0914    0.0568
    0.0400    0.0739
    0.0914    0.0966

e2 =
    0.0914    0.0625
    0.1257    0.0682
    0.0971    0.0625
    0.0800    0.0909
    0.0914    0.1193
```

`testckfold` treats trained classification models as templates, and so it ignores all fitted parameters in `C`. That is, `testckfold` cross validates `C` using only the specified options and the predictor data to estimate the out-of-fold classification losses.

`h = 0` indicates to not reject the null hypothesis that the two models have equal predictive accuracies. This result favors the simpler ensemble.

## Input Arguments

### **c1** — Classification model template or trained classification model

classification model template object | trained classification model object

Classification model template or trained classification model, specified as any classification model template object or trained classification model object described in these tables.

Template Type		Returned By
Classification tree		templateTree
Discriminant analysis		templateDiscriminant
Ensemble (boosting, bagging, and random subspace)		templateEnsemble
Error-correcting output codes (ECOC), multiclass classification model		templateECOC
$k$ NN		templateKNN
Naive Bayes		templateNaiveBayes
Support Vector Machine (SVM)		templateSVM
Trained Model Type	Model Object	Returned By
Classification tree	ClassificationTree	fitctree
Discriminant analysis	ClassificationDiscriminant	fitcdiscr
Ensemble of bagged classification models	ClassificationBaggedEnsemble	fitcensemble
Ensemble of classification models	ClassificationEnsemble	fitcensemble
ECOC model	ClassificationECOC	fitcecoc
$k$ NN	ClassificationKNN	fitcknn
Naive Bayes	ClassificationNaiveBayes	fitcnb
SVM	ClassificationSVM	fitsvm

For efficiency, supply a classification model template object instead of a trained classification model object.

## c2 — Classification model template or trained model

classification model template object | trained classification model object

Classification model template or trained classification model, specified as any classification model template object or trained classification model object described in these tables.

Template Type	Returned By
Classification tree	templateTree

Template Type		Returned By
Discriminant analysis		templateDiscriminant
Ensemble (boosting, bagging, and random subspace)		templateEnsemble
Error-correcting output codes (ECOC), multiclass classification model		templateECOC
$k$ NN		templateKNN
Naive Bayes		templateNaiveBayes
Support Vector Machine (SVM)		templateSVM
Trained Model Type	Model Object	Returned By
Classification tree	ClassificationTree	fitctree
Discriminant analysis	ClassificationDiscriminant	fitcdiscr
Ensemble of bagged classification models	ClassificationBaggedEnsemble	fitcensemble
Ensemble of classification models	ClassificationEnsemble	fitcensemble
ECOC model	ClassificationECOC	fitcecoc
$k$ NN	ClassificationKNN	fitcknn
Naive Bayes	ClassificationNaiveBayes	fitcnb
SVM	ClassificationSVM	fitsvm

For efficiency, supply a classification model template object instead of a trained classification model object.

### **x1 — Data used to apply to first full classification model or template**

numeric matrix | table

Data used to apply to the first full classification model or template, C1, specified as a numeric matrix or table.

Each row of X1 corresponds to one observation, and each column corresponds to one variable. `testckfold` does not support multi-column variables and cell arrays other than cell arrays of character vectors.

X1 and X2 must be of the same data type, and X1, X2, Y must have the same number of observations.

If you specify Y as an array, then `testckfold` treats all columns of X1 as separate predictor variables.

Data Types: `double` | `single` | `table`

### **x2 — Data used to apply to second full classification model or template**

numeric matrix | table

Data used to apply to the second full classification model or template, C2, specified as a numeric matrix or table.

Each row of X2 corresponds to one observation, and each column corresponds to one variable. `testckfold` does not support multi-column variables and cell arrays other than cell arrays of character vectors.

X1 and X2 must be of the same data type, and X1, X2, Y must have the same number of observations.

If you specify Y as an array, then `testckfold` treats all columns of X2 as separate predictor variables.

Data Types: `double` | `single` | `table`

### **y — True class labels**

categorical array | character array | logical vector | vector of numeric values | cell array of character vectors | character vector

True class labels, specified as a categorical or character array, a logical or numeric vector, a cell array of character vectors, or a character vector.

- For a character vector, X1 and X2 must be tables, their response variables must have the same name and values, and Y must be the common variable name. For example, if `X1.Labels` and `X2.Labels` are the response variables, then Y is `'Labels'` and `X1.Labels` and `X2.Labels` must be equivalent.
- For all other supported data types, Y is an array of true class labels.
  - If Y is a character array, then each element must correspond to one row of the array.

- `X1`, `X2`, `Y` must have the same number of observations (rows).
- If both of these statements are true, then you can omit supplying `Y`.
  - `X1` and `X2` are tables containing the same response variable (values and name).
  - `C1` and `C2` are full classification models containing `ResponseName` properties specifying the response variable names in `X1` and `X2`.

Consequently, `testckfold` uses the common response variable in the tables. For example, if the response variables in the tables are `X1.Labels` and `X2.Labels`, and the values of `C1.ResponseName` and `C2.ResponseName` are `'Labels'`, then you do not have to supply `Y`.

Data Types: `categorical` | `char` | `logical` | `single` | `double` | `cell`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example:

```
'Alternative', 'greater', 'Test', '10x10t', 'Options', statsset('UseParallel', true)
```

 specifies to test whether the first set of first predicted class labels is more accurate than the second set, to conduct the 10-by-10 t test, and to use parallel computing for cross validation.

### **Alpha** — Hypothesis test significance level

0.05 (default) | scalar value in the interval (0,1)

Hypothesis test significance level, specified as the comma-separated pair consisting of `'Alpha'` and a scalar value in the interval (0,1).

Example: `'Alpha', 0.1`

Data Types: `single` | `double`

### **Alternative** — Alternative hypothesis to assess

`'unequal'` (default) | `'greater'` | `'less'`

Alternative hypothesis to assess, specified as the comma-separated pair consisting of 'Alternative' and one of the values listed in the table.

Value	Alternative Hypothesis Description	Supported Tests
'unequal' (default)	For predicting $Y$ , the set of predictions resulting from $C1$ applied to $X1$ and $C2$ applied to $X2$ have unequal accuracies.	'5x2F', '5x2t', and '10x10t'
'greater'	For predicting $Y$ , the set of predictions resulting from $C1$ applied to $X1$ is more accurate than $C2$ applied to $X2$ .	'5x2t' and '10x10t'
'less'	For predicting $Y$ , the set of predictions resulting from $C1$ applied to $X1$ is less accurate than $C2$ applied to $X2$ .	'5x2t' and '10x10t'

For details on supported tests, see `Test`.

Example: 'Alternative', 'greater'

Data Types: char

### **X1CategoricalPredictors** — Flag identifying categorical predictors

[ ] (default) | logical vector | numeric vector | 'all'

Flag identifying categorical predictors in the first test-set predictor data ( $X1$ ), specified as the comma-separated pair consisting of 'X1CategoricalPredictors' and one of the following:

- A numeric vector with indices from 1 through  $p$ , where  $p$  is the number of columns of  $X1$ .
- A logical vector of length  $p$ , where a `true` entry means that the corresponding column of  $X1$  is a categorical variable.
- 'all', meaning all predictors are categorical.

Specification of `X1CategoricalPredictors` is appropriate if:

- At least one predictor is categorical and  $C1$  is a classification tree, an ensemble of classification trees, an ECOC model, or a naive Bayes classification model.
- All predictors are categorical and  $C1$  is a  $k$ NN classification model.

If you specify `X1CategoricalPredictors` for any other case, then `testckfold` throws an error. For example, the function cannot train SVM learners using categorical predictors.

The default is `[]`, which indicates that there are no categorical predictors.

Example: `'X1CategoricalPredictors', 'all'`

Data Types: `single` | `double`

### **X2CategoricalPredictors** — Flag identifying categorical predictors

`[]` (default) | logical vector | numeric vector | `'all'`

Flag identifying categorical predictors in the second test-set predictor data (`X2`), specified as the comma-separated pair consisting of `'X2CategoricalPredictors'` and one of the following:

- A numeric vector with indices from 1 through `p`, where `p` is the number of columns of `X2`.
- A logical vector of length `p`, where a `true` entry means that the corresponding column of `X2` is a categorical variable.
- `'all'`, meaning all predictors are categorical.

Specification of `X2CategoricalPredictors` is appropriate if:

- At least one predictor is categorical and `C2` is a classification tree, an ensemble of classification trees, an ECOC model, or a naive Bayes classification model.
- All predictors are categorical and `C2` is a  $k$ NN classification model.

If you specify `X2CategoricalPredictors` for any other case, then `testckfold` throws an error. For example, the function cannot train SVM learners using categorical predictors.

The default is `[]`, which indicates that there are no categorical predictors.

Example: `'X2CategoricalPredictors', 'all'`

Data Types: `single` | `double`

### **ClassNames** — Class names

categorical vector | cell array of character vectors | character array | logical vector | numeric vector



Class names, specified as the comma-separated pair consisting of 'ClassNames' and a categorical or character array, logical or numeric vector, or cell array of character vectors. You must set `ClassNames` using the data type of `Y`.

If `ClassNames` is a character array, then each element must correspond to one *row* of the array.

Use `ClassNames` to:

- Specify the order of any input argument dimension that corresponds to class order. For example, use `ClassNames` to specify the order of the dimensions of `Cost`.
- Select a subset of classes for testing. For example, suppose that the set of all distinct class names in `Y` is {'a', 'b', 'c'}. To train and test models using observations from classes 'a' and 'c' only, specify 'ClassNames', {'a', 'c'}.

The default is the set of all distinct class names in `Y`.

Example: 'ClassNames', {'b', 'g'}

Data Types: categorical | char | logical | single | double | cell

### **Cost — Classification cost**

square matrix | structure array

Classification cost, specified as the comma-separated pair consisting of 'Cost' and a square matrix or structure array.

- If you specify the square matrix `Cost`, then `Cost(i, j)` is the cost of classifying a point into class `j` if its true class is `i`. That is, the rows correspond to the true class and the columns correspond to the predicted class. To specify the class order for the corresponding rows and columns of `Cost`, additionally specify the `ClassNames` name-value pair argument.
- If you specify the structure `S`, then `S` must have two fields:
  - `S.ClassNames`, which contains the class names as a variable of the same data type as `Y`. You can use this field to specify the order of the classes.
  - `S.ClassificationCosts`, which contains the cost matrix, with rows and columns ordered as in `S.ClassNames`

For cost-sensitive testing use, `testcholdout`.

It is a best practice to supply the same cost matrix used to train the classification models.

The default is  $\text{Cost}(i, j) = 1$  if  $i \neq j$ , and  $\text{Cost}(i, j) = 0$  if  $i = j$ .

Example: 'Cost', [0 1 2 ; 1 0 2; 2 2 0]

Data Types: double | single | struct

### LossFun — Loss function

'classiferror' (default) | 'binodeviance' | 'exponential' | 'hinge' | function handle

Loss function, specified as the comma-separated pair consisting of 'LossFun' and 'classiferror', 'binodeviance', 'exponential', 'hinge', or a function handle.

- The following lists available loss functions.

Value	Loss Function
'binodeviance'	Binomial deviance
'classiferror'	Classification error
'exponential'	Exponential loss
'hinge'	Hinge loss

- Specify your own function using function handle notation.

Suppose that  $n = \text{size}(X, 1)$  is the sample size and there are  $K$  unique classes. Your function must have the signature  $\text{lossvalue} = \text{lossfun}(C, S, W, \text{Cost})$ , where:

- The output argument `lossvalue` is a scalar.
- `lossfun` is the name of your function.
- `C` is an  $n$ -by- $K$  logical matrix with rows indicating which class the corresponding observation belongs to. The column order corresponds to the class order in the `ClassNames` name-value pair argument.

Construct `C` by setting  $C(p, q) = 1$  if observation  $p$  is in class  $q$ , for each row. Set all other elements of row  $p$  to 0.

- `S` is an  $n$ -by- $K$  numeric matrix of classification scores. The column order corresponds to the class order in the `ClassNames` name-value pair argument. `S` is a matrix of classification scores.

- $\bar{w}$  is an  $n$ -by-1 numeric vector of observation weights. If you pass  $\bar{w}$ , the software normalizes the weights to sum to 1.
- $\text{Cost}$  is a  $K$ -by- $K$  numeric matrix of classification costs. For example,  $\text{Cost} = \text{ones}(K) - \text{eye}(K)$  specifies a cost of 0 for correct classification and a cost of 1 for misclassification.

Specify your function using `'LossFun', @lossfun`.

Data Types: `char` | `function_handle`

### Options — Parallel computing options

`[]` (default) | structure array returned by `statset`

Parallel computing options, specified as the comma-separated pair consisting of `'Options'` and a structure array returned by `statset`. These options require Parallel Computing Toolbox. `testckfold` uses `'Streams'`, `'UseParallel'`, and `'UseSubstreams'` fields.

This table summarizes the available options.

Option	Description
<code>'Streams'</code>	<p>A <code>RandStream</code> object or cell array of such objects. If you do not specify <code>Streams</code>, the software uses the default stream or streams. If you specify <code>Streams</code>, use a single object except when the following are true:</p> <ul style="list-style-type: none"> <li>• You have an open parallel pool.</li> <li>• <code>UseParallel</code> is <code>true</code>.</li> <li>• <code>UseSubstreams</code> is <code>false</code>.</li> </ul> <p>In that case, use a cell array of the same size as the parallel pool. If a parallel pool is not open, then the software tries to open one (depending on your preferences), and <code>Streams</code> must supply a single random number stream.</p>

Option	Description
'UseParallel'	If you have Parallel Computing Toolbox, then you can invoke a pool of workers by setting 'UseParallel', true.
'UseSubstreams'	Set to true to compute in parallel using the stream specified by 'Streams'. Default is false. For example, set Streams to a type allowing substreams, such as 'mlfg6331_64' or 'mrg32k3a'.

A best practice to ensure more predictable results is to use `parpool` and explicitly create a parallel pool before you invoke parallel computing using `testcckfold`.

Example: `'Options', statset('UseParallel', true)`

Data Types: `struct`

### Prior — Prior probabilities

'empirical' (default) | 'uniform' | numeric vector | structure

Prior probabilities for each class, specified as the comma-separated pair consisting of 'Prior' and 'empirical', 'uniform', a numeric vector, or a structure.

This table summarizes the available options for setting prior probabilities.

Value	Description
'empirical'	The class prior probabilities are the class relative frequencies in $Y$ .
'uniform'	All class prior probabilities are equal to $1/K$ , where $K$ is the number of classes.
numeric vector	Each element is a class prior probability. Specify the order using the <code>ClassNames</code> name-value pair argument. The software normalizes the elements such that they sum to 1.

Value	Description
structure	<p>A structure <i>S</i> with two fields:</p> <ul style="list-style-type: none"> <li>• <i>S.ClassNames</i> contains the class names as a variable of the same type as <i>Y</i>.</li> <li>• <i>S.ClassProbs</i> contains a vector of corresponding prior probabilities. The software normalizes the elements such that they sum to 1.</li> </ul>

Example: `'Prior', struct('ClassNames',  
{'setosa', 'versicolor'}), 'ClassProbs', [1, 2])`

#### Test — Test to conduct

'5x2F' (default) | '5x2t' | '10x10t'

Test to conduct, specified as the comma-separated pair consisting of 'Test' and one of the following: '5x2F', '5x2t', '10x10t'.

Value	Description	Supported Alternative Hypothesis
'5x2F' (default)	5-by-2 paired <i>F</i> test. Appropriate for two-sided testing only.	'unequal'
'5x2t'	5-by-2 paired <i>t</i> test	'unequal', 'less', 'greater'
'10x10t'	10-by-10 repeated cross validation <i>t</i> test	'unequal', 'less', 'greater'

For details on the available tests, see “Repeated Cross-Validation Tests” on page 32-6546. For details on supported alternative hypotheses, see *Alternative*.

Example: `'Test', '10x10t'`

#### Verbose — Verbosity level

0 (default) | 1 | 2

Verbosity level, specified as the comma-separated pair consisting of 'Verbose' and 0, 1, or 2. Verbose controls the amount of diagnostic information that the software displays in the Command Window during training of each cross-validation fold.

This table summarizes the available verbosity level options.

Value	Description
0	The software does not display diagnostic information.
1	The software displays diagnostic messages every time it implements a new cross validation run.
2	The software displays diagnostic messages every time it implements a new cross validation run, and every time it trains on a particular fold.

Example: 'Verbose', 1

Data Types: double | single

### **weights** — Observation weights

`ones(size(X,1),1)` (default) | numeric vector

Observation weights, specified as the comma-separated pair consisting of 'Weights' and a numeric vector.

The size of `Weights` must equal the number of rows of `X1`. The software weighs the observations in each row of `X` with the corresponding weight in `Weights`.

The software normalizes `Weights` to sum up to the value of the prior probability in the respective class.

Data Types: double | single

---

### **Notes:**

- `testckfold` treats trained classification models as templates. Therefore, it ignores all fitted parameters in the model. That is, `testckfold` cross validates using only the options specified in the model and the predictor data.

- The repeated cross-validation tests depend on the assumption that the test statistics are asymptotically normal under the null hypothesis. Highly imbalanced cost matrices (for example,  $\text{Cost} = [0 \ 100; 1 \ 0]$ ) and highly discrete response distributions (that is, most of the observations are in a small number of classes) might violate the asymptotic normality assumption. For cost-sensitive testing, use `testcholdout`.
- NaNs, `<undefined>` values, and empty character vectors ( ' ') indicate missing values.
  - For the treatment of missing values in `X1` and `X2`, see the appropriate classification model training function reference page: `fitctree`, `fitcdiscr`, `fitcensemble`, `fitcecoc`, `fitcknn`, `fitcnb`, or `fitcsvm`.
  - `testckfold` removes missing values in `Y` and the corresponding rows of `X1` and `X2`.

---

## Output Arguments

### **h** — Hypothesis test result

1 | 0

Hypothesis test result, returned as a logical value.

`h = 1` indicates the rejection of the null hypothesis at the Alpha significance level.

`h = 0` indicates failure to reject the null hypothesis at the Alpha significance level.

### **p** — *p*-value

scalar in the interval [0,1]

*p*-value of the test, returned as a scalar in the interval [0,1]. *p* is the probability that a random test statistic is at least as extreme as the observed test statistic, given that the null hypothesis is true.

`testckfold` estimates *p* using the distribution of the test statistic, which varies with the type of test. For details on test statistics, see “Repeated Cross-Validation Tests” on page 32-6546.

### **e1** — Classification losses

numeric matrix

Classification losses on page 32-6549, returned as a numeric matrix. The rows of  $e_1$  correspond to the cross-validation run and the columns correspond to the test fold.

`testckfold` applies the first test-set predictor data ( $X_1$ ) to the first classification model ( $C_1$ ) to estimate the first set of class labels.

$e_1$  summarizes the accuracy of the first set of class labels predicting the true class labels ( $Y$ ) for each cross-validation run and fold. The meaning of the elements of  $e_1$  depends on the type of classification loss.

### **$e_2$ — Classification losses**

numeric matrix

Classification losses on page 32-6549, returned as a numeric matrix. The rows of  $e_2$  correspond to the cross-validation run and the columns correspond to the test fold.

`testckfold` applies the first test-set predictor data ( $X_2$ ) to the first classification model ( $C_2$ ) to estimate the first set of class labels.

$e_2$  summarizes the accuracy of the first set of class labels predicting the true class labels ( $Y$ ) for each cross-validation run and fold. The meaning of the elements of  $e_2$  depends on the type of classification loss.

## Definitions

### Repeated Cross-Validation Tests

Repeated cross-validation tests form the test statistic for comparing the accuracies of two classification models by combining the classification loss differences resulting from repeatedly cross validating the data. Repeated cross-validation tests are useful when sample size is limited.

To conduct an  $R$ -by- $K$  test:

- 1 Randomly divide (stratified by class) the predictor data sets and true class labels into  $K$  sets,  $R$  times. Each division is called a run and each set within a run is called a fold. Each run contains the complete, but divided, data sets.
- 2 For runs  $r = 1$  through  $R$ , repeat these steps for  $k = 1$  through  $K$ :



- a Reserve fold  $k$  as a test set, and train the two classification models using their respective predictor data sets on the remaining  $K - 1$  folds.
- b Predict class labels using the trained models and their respective fold  $k$  predictor data sets.
- c Estimate the classification loss by comparing the two sets of estimated labels to the true labels. Denote  $e_{crk}$  as the classification loss when the test set is fold  $k$  in run  $r$  of classification model  $c$ .
- d Compute the difference between the classification losses of the two models:
 
$$\hat{\delta}_{rk} = e_{1rk} - e_{2rk}.$$

At the end of a run, there are  $K$  classification losses per classification model.

**3** Combine the results of step 2. For each  $r = 1$  through  $R$ :

- Estimate the within-fold averages of the differences and their average:

$$\bar{\delta}_r = \frac{1}{K} \sum_{k=1}^K \hat{\delta}_{rk}.$$

•

Estimate the overall average of the differences:  $\bar{\delta} = \frac{1}{KR} \sum_{r=1}^R \sum_{k=1}^K \hat{\delta}_{rk}.$

•

Estimate the within-fold variances of the differences:  $s_r^2 = \frac{1}{K} \sum_{k=1}^K (\hat{\delta}_{rk} - \bar{\delta}_r)^2.$

•

Estimate the average of the within-fold differences:  $\bar{s}^2 = \frac{1}{R} \sum_{r=1}^R s_r^2.$

- Estimate the overall sample variance of the differences:

$$S^2 = \frac{1}{KR - 1} \sum_{r=1}^R \sum_{k=1}^K (\hat{\delta}_{rk} - \bar{\delta})^2.$$

Compute the test statistic. All supported tests described here assume that, under  $H_0$ , the estimated differences are independent and approximately normally distributed, with mean 0 and a finite, common standard deviation. However, these tests violate the independence assumption, and so the test-statistic distributions are approximate.

- For  $R = 2$ , the test is a paired test. The two supported tests are a paired  $t$  and  $F$  test.

- The test statistic for the paired  $t$  test is

$$t_{paired}^* = \frac{\hat{\delta}_{11}}{\sqrt{s^2}}.$$

$t_{paired}^*$  has a  $t$ -distribution with  $R$  degrees of freedom under the null hypothesis.

To reduce the effects of correlation between the estimated differences, the quantity  $\hat{\delta}_{11}$  occupies the numerator rather than  $\bar{\delta}$ .

5-by-2 paired  $t$  tests can be slightly conservative [4].

- The test statistic for the paired  $F$  test is

$$F_{paired}^* = \frac{\frac{1}{RK} \sum_{r=1}^R \sum_{k=1}^K (\hat{\delta}_{rk})^2}{s^2}.$$

$F_{paired}^*$  has an  $F$  distribution with  $RK$  and  $R$  degrees of freedom.

A 5-by-2 paired  $F$  test has comparable power to the 5-by-2  $t$  test, but is more conservative [1].

- For  $R > 2$ , the test is a repeated cross-validation test. The test statistic is

$$t_{CV}^* = \frac{\bar{\delta}}{S/\sqrt{\nu+1}}.$$

$t_{CV}^*$  has a  $t$  distribution with  $\nu$  degrees of freedom. If the differences were truly independent, then  $\nu = RK - 1$ . In this case, the degrees of freedom parameter must be optimized.

For a 10-by-10 repeated cross-validation  $t$  test, the optimal degrees of freedom between 8 and 11 ([2] and [3]). `testckfold` uses  $\nu = 10$ .

The advantage of repeated cross validation tests over paired tests is that the results are more repeatable [3]. The disadvantage is that they require high computational resources.

## Classification Loss

Classification losses indicate the accuracy of a classification model or set of predicted labels. In general, for a fixed cost matrix, classification accuracy decreases as classification loss increases.

`testckfold` returns the classification losses (see  $e_1$  and  $e_2$ ) under the alternative hypothesis (that is, the unrestricted classification losses). In the definitions that follow:

- The classification losses focus on the first classification model. The classification losses for the second model are similar.
- $n_{test}$  is the test-set sample size.
- $I(x)$  is the indicator function. If  $x$  is a true statement, then  $I(x) = 1$ . Otherwise,  $I(x) = 0$ .
- $\hat{p}_{1j}$  is the predicted class assignment of classification model 1 for observation  $j$ .
- $y_j$  is the true class label of observation  $j$ .
- Binomial deviance has the form

$$e_1 = \frac{\sum_{j=1}^{n_{test}} w_j \log\left(1 + \exp\left(-2y'_j f(X_j)\right)\right)}{\sum_{j=1}^{n_{test}} w_j}$$

where:

- $y_j = 1$  for the positive class and  $-1$  for the negative class.
- $f(X_j)$  is the classification score.

The binomial deviance has connections to the maximization of the binomial likelihood function. For details on binomial deviance, see [5].

- Exponential loss is similar to binomial deviance and has the form

$$e_1 = \frac{\sum_{j=1}^{n_{test}} w_j \exp\left(-y_j f(X_j)\right)}{\sum_{j=1}^{n_{test}} w_j}.$$

$y_j$  and  $f(X_j)$  take the same forms here as in the binomial deviance formula.

- Hinge loss has the form

$$e_1 = \frac{\sum_{j=1}^n w_j \max\{0, 1 - y_j' f(X_j)\}}{\sum_{j=1}^n w_j},$$

$y_j$  and  $f(X_j)$  take the same forms here as in the binomial deviance formula.

Hinge loss linearly penalizes for misclassified observations and is related to the SVM objective function used for optimization. For more details on hinge loss, see [5].

- Misclassification rate, or classification error, is a scalar in the interval  $[0,1]$  representing the proportion of misclassified observations. That is, the misclassification rate for the first classification model is

$$e_1 = \frac{\sum_{j=1}^{n_{test}} w_j I(p_{1j} \neq y_j)}{\sum_{j=1}^{n_{test}} w_j}.$$

## Tips

- Examples of ways to compare models include:
  - Compare the accuracies of a simple classification model and a more complex model by passing the same set of predictor data.
  - Compare the accuracies of two different models using two different sets of predictors.
  - Perform various types of feature selection. For example, you can compare the accuracy of a model trained using a set of predictors to the accuracy of one trained on a subset or different set of predictors. You can arbitrarily choose the set of predictors, or use a feature selection technique like PCA or sequential feature selection (see `pca` and `sequentialfs`).

- If both of these statements are true, then you can omit supplying  $Y$ .
  - $X1$  and  $X2$  are tables containing the response variable and use the same response variable name.
  - $C1$  and  $C2$  are full classification models containing equal `ResponseName` properties (e.g. `strcmp(C1.ResponseName, C2.ResponseName) = 1`).

Consequently, `testckfold` uses the common response variable in the tables.

- One way to perform cost-insensitive feature selection is:
  - 1 Create a classification model template that characterizes the first classification model ( $C1$ ).
  - 2 Create a classification model template that characterizes the second classification model ( $C2$ ).
  - 3 Specify two predictor data sets. For example, specify  $X1$  as the full predictor set and  $X2$  as a reduced set.
  - 4 Enter `testckfold(C1, C2, X1, X2, Y, 'Alternative', 'less')`. If `testckfold` returns 1, then there is enough evidence to suggest that the classification model that uses fewer predictors performs better than the model that uses the full predictor set.

Alternatively, you can assess whether there is a significant difference between the accuracies of the two models. To perform this assessment, remove the `'Alternative', 'less'` specification in step 4. `testckfold` conducts a two-sided test, and  $h = 0$  indicates that there is not enough evidence to suggest a difference in the accuracy of the two models.

- The tests are appropriate for the misclassification rate classification loss on page 32-6549, but you can specify other loss functions (see `LOSSFUN`). The key assumptions are that the estimated classification losses are independent and normally distributed with mean 0 and finite common variance under the two-sided null hypothesis. Classification losses other than the misclassification rate can violate this assumption.
- Highly discrete data, imbalanced classes, and highly imbalanced cost matrices can violate the normality assumption of classification loss differences.

## Algorithms

If you specify to conduct the 10-by-10 repeated cross-validation  $t$  test using 'Test', '10x10t', then `testckfold` uses 10 degrees of freedom for the  $t$  distribution to find the critical region and estimate the  $p$ -value. For more details, see [2] and [3].

## Alternatives

Use `testcholdout`:

- For test sets with larger sample sizes
- To implement variants of the McNemar test to compare two classification model accuracies
- For cost-sensitive testing using a chi-square or likelihood ratio test. The chi-square test uses `quadprog`, which requires an Optimization Toolbox license.

## References

- [1] Alpaydin, E. “Combined 5 x 2 CV F Test for Comparing Supervised Classification Learning Algorithms.” *Neural Computation*, Vol. 11, No. 8, 1999, pp. 1885–1992.
- [2] Bouckaert, R. “Choosing Between Two Learning Algorithms Based on Calibrated Tests.” *International Conference on Machine Learning*, 2003, pp. 51–58.
- [3] Bouckaert, R., and E. Frank. “Evaluating the Replicability of Significance Tests for Comparing Learning Algorithms.” *Advances in Knowledge Discovery and Data Mining, 8th Pacific-Asia Conference*, 2004, pp. 3–12.
- [4] Dietterich, T. “Approximate statistical tests for comparing supervised classification learning algorithms.” *Neural Computation*, Vol. 10, No. 7, 1998, pp. 1895–1923.
- [5] Hastie, T., R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*, 2nd Ed. New York: Springer, 2008.

## See Also

`templateDiscriminant` | `templateECOC` | `templateEnsemble` | `templateKNN` | `templateNaiveBayes` | `templateSVM` | `templateTree` | `testcholdout`

## **Topics**

“Hypothesis Tests”

**Introduced in R2015a**

## TestSize property

**Class:** cvpartition

Size of each test set

### Description

Value is a vector in partitions of type 'kfold' and 'leaveout'.

Value is a scalar in partitions of type 'holdout' and 'resubstitution'.



# tiedrank

Rank adjusted for ties

## Syntax

```
[R, TIEADJ] = tiedrank(X)
[R, TIEADJ] = tiedrank(X, 1)
[R, TIEADJ] = tiedrank(X, 0, 1)
```

## Description

`[R, TIEADJ] = tiedrank(X)` computes the ranks of the values in the vector `X`. If any `X` values are tied, `tiedrank` computes their average rank. The return value `TIEADJ` is an adjustment for ties required by the nonparametric tests `signrank` and `ranksum`, and for the computation of Spearman's rank correlation.

`[R, TIEADJ] = tiedrank(X, 1)` computes the ranks of the values in the vector `X`. `TIEADJ` is a vector of three adjustments for ties required in the computation of Kendall's tau. `tiedrank(X, 0)` is the same as `tiedrank(X)`.

`[R, TIEADJ] = tiedrank(X, 0, 1)` computes the ranks from each end, so that the smallest and largest values get rank 1, the next smallest and largest get rank 2, etc. These ranks are used in the Ansari-Bradley test.

## Examples

Counting from smallest to largest, the two 20 values are 2nd and 3rd, so they both get rank 2.5 (average of 2 and 3):

```
tiedrank([10 20 30 40 20])
ans =
    1.0000    2.5000    4.0000    5.0000    2.5000
```

## Algorithms

`tiedrank` treats NaNs in `X` as missing values and ignores them. The rank of NaNs in the output argument `R` is NaN.

## See Also

`ansaribradley` | `corr` | `partialcorr` | `ranksum` | `signrank`

**Introduced before R2006a**

## tinv

Student's  $t$  inverse cumulative distribution function

### Syntax

```
x = tinv(p,nu)
```

### Description

`x = tinv(p,nu)` returns the inverse of Student's  $t$  cdf using the degrees of freedom in `nu` for the corresponding probabilities in `p`. `p` and `nu` can be vectors, matrices, or multidimensional arrays that are the same size. A scalar input is expanded to a constant array with the same dimensions as the other inputs. The values in `p` must lie on the interval  $[0\ 1]$ .

The  $t$  inverse function in terms of the  $t$  cdf is

$$x = F^{-1}(p | \nu) = \{x : F(x | \nu) = p\}$$

where

$$p = F(x | \nu) = \int_{-\infty}^x \frac{\Gamma\left(\frac{\nu+1}{2}\right)}{\Gamma\left(\frac{\nu}{2}\right)} \frac{1}{\sqrt{\nu\pi}} \frac{1}{\left(1 + \frac{t^2}{\nu}\right)^{\frac{\nu+1}{2}}} dt$$

The result,  $x$ , is the solution of the cdf integral with parameter  $\nu$ , where you supply the desired probability  $p$ .

### Examples

### Compute Student's *t* icdf

What is the 99th percentile of the Student's *t* distribution for one to six degrees of freedom?

```
percentile = tinv(0.99,1:6)
```

```
percentile =
```

```
    31.8205    6.9646    4.5407    3.7469    3.3649    3.1427
```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

[icdf](#) | [tcdf](#) | [tpdf](#) | [trnd](#) | [tstat](#)

### Topics

“Student's *t* Distribution” on page B-166

Introduced before R2006a

# prob.ToolboxFittableParametricDistribution class

**Package:** prob

**Superclasses:**

Toolbox-integrated fittable parametric probability distribution object

## Description

Create a probability distribution object with specified parameter values using `makedist`. Alternatively, fit a distribution to data using `fitdist` or the Distribution Fitter app.

## Methods

<code>mean</code>	Mean of probability distribution object
<code>negloglik</code>	Negative log likelihood of probability distribution object
<code>paramci</code>	Confidence intervals for probability distribution parameters
<code>proflik</code>	Profile likelihood function for probability distribution object
<code>std</code>	Standard deviation of probability distribution object
<code>var</code>	Variance of probability distribution object

## Inherited Methods

<code>cdf</code>	Cumulative distribution function of probability distribution object
<code>icdf</code>	Inverse cumulative distribution function of probability distribution object
<code>iqr</code>	Interquartile range of probability distribution object
<code>median</code>	Median of probability distribution object
<code>pdf</code>	Probability density function of probability distribution object
<code>random</code>	Generate random numbers from probability distribution object
<code>truncate</code>	Truncate probability distribution object

## See Also

`distributionFitter` | `fitdist` | `makedist`

## Topics

Class Attributes (MATLAB)

Property Attributes (MATLAB)

## tpdf

Student's  $t$  probability density function

### Syntax

```
y = tpdf(x,nu)
```

### Description

`y = tpdf(x,nu)` returns the probability density function (pdf) of the Student's  $t$  distribution at each of the values in `x` using the corresponding degrees of freedom in `nu`. `x` and `nu` can be vectors, matrices, or multidimensional arrays that have the same size. A scalar input is expanded to a constant array with the same dimensions as the other inputs.

### Examples

#### Compute Student's $t$ pdf

The mode of the Student's  $t$  distribution is at  $x = 0$ . This example shows that the value of the function at the mode is an increasing function of the degrees of freedom.

```
tpdf(0,1:6)
```

```
ans =
```

```
0.3183    0.3536    0.3676    0.3750    0.3796    0.3827
```

The  $t$  distribution converges to the standard normal distribution as the degrees of freedom approaches infinity. How good is the approximation for  $\nu$  equal to 30?

```
difference = tpdf(-2.5:2.5,30)-normpdf(-2.5:2.5)
```

difference =

0.0035   -0.0006   -0.0042   -0.0042   -0.0006   0.0035

## Definitions

### Student's *t* pdf

The probability density function (pdf) of the Student's *t* distribution is

$$y = f(x | \nu) = \frac{\Gamma\left(\frac{\nu+1}{2}\right)}{\Gamma\left(\frac{\nu}{2}\right)} \frac{1}{\sqrt{\nu\pi}} \frac{1}{\left(1 + \frac{x^2}{\nu}\right)^{\frac{\nu+1}{2}}}$$

where  $\nu$  is the degrees of freedom and  $\Gamma(\cdot)$  is the Gamma function. The result  $y$  is the probability of observing a particular value of  $x$  from a Student's *t* distribution with  $\nu$  degrees of freedom.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

[pdf](#) | [tcdf](#) | [tinv](#) | [trnd](#) | [tstat](#)

### Topics

“Student's *t* Distribution” on page B-166



**Introduced before R2006a**

## training

**Class:** cvpartition

Training indices for cross-validation

### Syntax

```
idx = training(c)
idx = training(c,i)
```

### Description

`idx = training(c)` returns the logical vector `idx` of training indices for an object `c` of the `cvpartition` on page 32-1292 class of type 'holdout' or 'resubstitution'.

If `c.Type` is 'holdout', `idx` specifies the observations in the training set.

If `c.Type` is 'resubstitution', `idx` specifies all observations.

`idx = training(c,i)` returns the logical vector `idx` of training indices for repetition `i` of an object `c` of the `cvpartition` on page 32-1292 class of type 'kfold' or 'leaveout'.

If `c.Type` is 'kfold', `idx` specifies the observations in the training set in fold `i`.

If `c.Type` is 'leaveout', `idx` specifies the observations left in at repetition `i`.

### Examples

Identify the training indices in the first fold of a partition of 10 observations for 3-fold cross-validation:

```
c = cvpartition(10,'kfold',3)
c =
K-fold cross validation partition
```

```
      N: 10
NumTestSets: 3
  TrainSize: 7 6 7
  TestSize: 3 4 3

training(c,1)
ans =
  0
  0
  1
  1
  1
  1
  1
  1
  1
  0
  1
```

## See Also

`cvpartition` | `test`

## TrainSize property

**Class:** cvpartition

Size of each training set

### Description

Value is a vector in partitions of type 'kfold' and 'leaveout'.

Value is a scalar in partitions of type 'holdout' and 'resubstitution'.

### See Also

type

# transform

Transform predictors into extracted features

## Syntax

```
z = transform(Mdl,x)
```

## Description

`z = transform(Mdl,x)` transforms the data `x` into the features `z` via the model `Mdl`.

## Examples

### Transform Data to Learned Features

Create a feature transformation model with 100 features from the `caltech101patches` data.

```
rng default % For reproducibility
data = load('caltech101patches');
q = 100;
X = data.X;
Mdl = sparsefilt(X,q)
```

Warning: Solver LBFGS was not able to converge to a solution.

Mdl =

```
  SparseFiltering
    ModelParameters: [1x1 struct]
      NumPredictors: 363
    NumLearnedFeatures: 100
                Mu: []
                Sigma: []
          FitInfo: [1x1 struct]
```

```
TransformWeights: [363x100 double]
InitialTransformWeights: []
```

`sparsefilt` issues a warning because it stopped due to reaching the iteration limit, instead of reaching a step-size limit or a gradient-size limit. You can still use the learned features in the returned object by calling the `transform` function.

Transform the first five rows of the input data `X` to the new feature space.

```
y = transform(Mdl,X(1:5,:));
size(y)
```

```
ans =
     5    100
```

- “Feature Extraction Workflow” on page 15-169

## Input Arguments

### **Mdl** — Feature extraction model

`SparseFiltering` object | `ReconstructionICA` object

Feature extraction model, specified as a `SparseFiltering` object or as a `ReconstructionICA` object. Create `Mdl` by using the `sparsefilt` function or the `rica` function.

### **x** — Predictor data

matrix with `p` columns | table of numeric values with `p` columns

Predictor data, specified as a matrix with `p` columns or as a table of numeric values with `p` columns. Here, `p` is the number of predictors in the model, which is `Mdl.NumPredictors`. Each row of the input matrix or table represents one data point to transform.

Data Types: `single` | `double` | `table`

## Output Arguments

### **z** — Transformed data

n-by-q matrix

Transformed data, returned as an n-by-q matrix. Here, n is the number of rows in the input data  $x$ , and q is the number of features, which is `Mdl.NumLearnedFeatures`.

## Algorithms

`transform` converts data to predicted features by using the learned weight matrix  $W$  to map input predictors to output features.

- For `rica`, input data  $X$  maps linearly to output features  $XW$ . See “Reconstruction ICA Algorithm” on page 15-166.
- For `sparsefilt`, input data maps nonlinearly to output features  $\hat{F}(X,W)$ . See “Sparse Filtering Algorithm” on page 15-163.

---

**Caution** The result of `transform` for sparse filtering depends on the number of data points. In particular, the result of applying `transform` to each row of a matrix separately differs from the result of applying `transform` to the entire matrix at once.

---

## See Also

`ReconstructionICA` | `SparseFiltering` | `rica` | `sparsefilt`

## Topics

“Feature Extraction Workflow” on page 15-169

“Feature Extraction” on page 15-163

Introduced in R2017a

## TreeArgs property

**Class:** `TreeBagger`

Cell array of arguments for `fitctree` or `fitrtree`

### Description

The `TreeArgs` property is a cell array of arguments for `fitctree` or `fitrtree`. `TreeBagger` uses these arguments in growing new trees for the ensemble.

### See Also

`ClassificationTree` | `RegressionTree` | `TreeBagger` | `fitctree` | `fitrtree`



# TreeBagger class

Bag of decision trees

## Description

TreeBagger bags an ensemble of decision trees for either classification or regression. Bagging stands for bootstrap aggregation. Every tree in the ensemble is grown on an independently drawn bootstrap replica of input data. Observations not included in this replica are "out of bag" for this tree.

TreeBagger relies on the `ClassificationTree` and `RegressionTree` functionality for growing individual trees. In particular, `ClassificationTree` and `RegressionTree` accepts the number of features selected at random for each decision split as an optional input argument. That is, TreeBagger implements the random forest algorithm [1].

For regression problems, TreeBagger supports mean and quantile regression (that is, quantile regression forest [2]).

- To predict mean responses or estimate the mean-squared error given data, pass a TreeBagger model and the data to `predict` or `error`, respectively. To perform similar operations for out-of-bag observations, use `oobPredict` or `oobError`.
- To estimate quantiles of the response distribution or the quantile error given data, pass a TreeBagger model and the data to `quantilePredict` or `quantileError`, respectively. To perform similar operations for out-of-bag observations, use `oobQuantilePredict` or `oobError`.

## Construction

TreeBagger

Create bag of decision trees

## Methods

<code>append</code>	Append new trees to ensemble
<code>compact</code>	Compact ensemble of decision trees
<code>error</code>	Error (misclassification probability or MSE)
<code>fillprox</code>	Proximity matrix for training data
<code>growTrees</code>	Train additional trees and add to ensemble
<code>margin</code>	Classification margin
<code>mdsprox</code>	Multidimensional scaling of proximity matrix
<code>meanMargin</code>	Mean classification margin
<code>oobError</code>	Out-of-bag error
<code>oobMargin</code>	Out-of-bag margins
<code>oobMeanMargin</code>	Out-of-bag mean margins
<code>oobPredict</code>	Ensemble predictions for out-of-bag observations
<code>oobQuantileError</code>	Out-of-bag quantile loss of bag of regression trees
<code>oobQuantilePredict</code>	Quantile predictions for out-of-bag observations from bag of regression trees
<code>predict</code>	Predict responses using ensemble of bagged decision trees
<code>quantileError</code>	Quantile loss using bag of regression trees
<code>quantilePredict</code>	Predict response quantile using bag of regression trees

## Properties

### **ClassNames**

A cell array containing the class names for the response variable  $Y$ . This property is empty for regression trees.

### **ComputeOOBPrediction**

A logical flag specifying whether out-of-bag predictions for training observations should be computed. The default is `false`.

If this flag is `true`, the following properties are available:

- `OOBIndices`
- `OOBInstanceWeight`

If this flag is true, the following methods can be called:

- `oobError`
- `oobMargin`
- `oobMeanMargin`

#### **ComputeOOBPredictorImportance**

A logical flag specifying whether out-of-bag estimates of variable importance should be computed. The default is false. If this flag is true, then `ComputeOOBPrediction` is true as well.

If this flag is true, the following properties are available:

- `OOBPermutedPredictorDeltaError`
- `OOBPermutedPredictorDeltaMeanMargin`
- `OOBPermutedPredictorCountRaiseMargin`

#### **Cost**

Square matrix, where `Cost(i, j)` is the cost of classifying a point into class `j` if its true class is `i` (i.e., the rows correspond to the true class and the columns correspond to the predicted class). The order of the rows and columns of `Cost` corresponds to the order of the classes in `ClassNames`. The number of rows and columns in `Cost` is the number of unique classes in the response.

This property is:

- read-only
- empty (`[]`) for ensembles of regression trees

#### **DefaultYfit**

Default value returned by `predict` and `oobPredict`. The `DefaultYfit` property controls what predicted value is returned when no prediction is possible. For example, when `oobPredict` needs to predict for an observation that is in-bag for all trees in the ensemble.

- For classification, you can set this property to either '' or 'MostPopular'. If you choose 'MostPopular' (the default), the property value becomes the name of the most probably class in the training data. If you choose '', the in-bag observations are excluded from computation of the out-of-bag error and margin.
- For regression, you can set this property to any numeric scalar. The default value is the mean of the response for the training data. If you set this property to NaN, the in-bag observations are excluded from computation of the out-of-bag error and margin.

**DeltaCriterionDecisionSplit**

A numeric array of size 1-by-*Nvars* of changes in the split criterion summed over splits on each variable, averaged across the entire ensemble of grown trees.

**InBagFraction**

Fraction of observations that are randomly selected with replacement for each bootstrap replica. The size of each replica is  $Nobs \times InBagFraction$ , where *Nobs* is the number of observations in the training set. The default value is 1.

**MergeLeaves**

A logical flag specifying whether decision tree leaves with the same parent are merged for splits that do not decrease the total risk. The default value is `false`.

**Method**

Method used by trees. The possible values are 'classification' for classification ensembles, and 'regression' for regression ensembles.

**MinLeafSize**

Minimum number of observations per tree leaf. By default, `MinLeafSize` is 1 for classification and 5 for regression. For decision tree training, the `MinParent` value is set equal to  $2 * MinLeafSize$ .

**NumTrees**

Scalar value equal to the number of decision trees in the ensemble.

**NumPredictorSplit**

A numeric array of size 1-by-*Nvars*, where every element gives a number of splits on this predictor summed over all trees.

**NumPredictorsToSample**

Number of predictor or feature variables to select at random for each decision split. By default, `NumPredictorsToSample` is equal to the square root of the total number of variables for classification, and one third of the total number of variables for regression.

**OOBIndices**

Logical array of size *Nobs*-by-*NumTrees*, where *Nobs* is the number of observations in the training data and *NumTrees* is the number of trees in the ensemble. A `true` value for the  $(i,j)$  element indicates that observation *i* is out-of-bag for tree *j*. In other words, observation *i* was not selected for the training data used to grow tree *j*.

**OOBInstanceWeight**

Numeric array of size *Nobs*-by-1 containing the number of trees used for computing the out-of-bag response for each observation. *Nobs* is the number of observations in the training data used to create the ensemble.

**OOBPermutedPredictorCountRaiseMargin**

A numeric array of size 1-by-*Nvars* containing a measure of variable importance for each predictor variable (feature). For any variable, the measure is the difference between the number of raised margins and the number of lowered margins if the values of that variable are permuted across the out-of-bag observations. This measure is computed for every tree, then averaged over the entire ensemble and divided by the standard deviation over the entire ensemble. This property is empty for regression trees.

**OOBPermutedPredictorDeltaError**

A numeric array of size 1-by-*Nvars* containing a measure of importance for each predictor variable (feature). For any variable, the measure is the increase in prediction error if the values of that variable are permuted across the out-of-bag observations. This measure is computed for every tree, then averaged over the entire ensemble and divided by the standard deviation over the entire ensemble.

**OOBPermutedPredictorDeltaMeanMargin**

A numeric array of size 1-by-*Nvars* containing a measure of importance for each predictor variable (feature). For any variable, the measure is the decrease in the classification margin if the values of that variable are permuted across the out-of-bag observations. This measure is computed for every tree, then averaged over the entire ensemble and divided by the standard deviation over the entire ensemble. This property is empty for regression trees.

**OutlierMeasure**

A numeric array of size *Nobs*-by-1, where *Nobs* is the number of observations in the training data, containing outlier measures for each observation.

**Prior**

Numeric vector of prior probabilities for each class. The order of the elements of `Prior` corresponds to the order of the classes in `ClassNames`.

This property is:

- read-only
- empty ([]) for ensembles of regression trees

**Proximity**

A numeric matrix of size *Nobs*-by-*Nobs*, where *Nobs* is the number of observations in the training data, containing measures of the proximity between observations. For any two observations, their proximity is defined as the fraction of trees for which these observations land on the same leaf. This is a symmetric matrix with 1s on the diagonal and off-diagonal elements ranging from 0 to 1.

**Prune**

The `Prune` property is true if decision trees are pruned and false if they are not. Pruning decision trees is not recommended for ensembles. The default value is false.

**SampleWithReplacement**

A logical flag specifying if data are sampled for each decision tree with replacement. This property is true if `TreeBagger` samples data with replacement and false otherwise. Default value is true.

**TreeArguments**

Cell array of arguments for `fitctree` or `fitrtree`. These arguments are used by `TreeBagger` when growing new trees for the ensemble.

**Trees**

A cell array of size *NumTrees*-by-1 containing the trees in the ensemble.

**SurrogateAssociation**

A matrix of size *Nvars*-by-*Nvars* with predictive measures of variable association, averaged across the entire ensemble of grown trees. If you grew the ensemble setting 'surrogate' to 'on', this matrix for each tree is filled with predictive measures of association averaged over the surrogate splits. If you grew the ensemble setting 'surrogate' to 'off' (default), `SurrogateAssociation` is diagonal.

**PredictorNames**

A cell array containing the names of the predictor variables (features). `TreeBagger` takes these names from the optional 'names' parameter. The default names are 'x1', 'x2', etc.

**W**

Numeric vector of weights of length *Nobs*, where *Nobs* is the number of observations (rows) in the training data. `TreeBagger` uses these weights for growing every decision tree in the ensemble. The default `W` is `ones(Nobs, 1)`.

**X**

A table or numeric matrix of size *Nobs*-by-*Nvars*, where *Nobs* is the number of observations (rows) and *Nvars* is the number of variables (columns) in the training data. If you train the ensemble using a table of predictor values, then `X` is a table. If you train the ensemble using a matrix of predictor values, then `X` is a matrix. This property contains the predictor (or feature) values.

**Y**

A size *Nobs* array of response data. Elements of `Y` correspond to the rows of `X`. For classification, `Y` is the set of true class labels. Labels can be any grouping variable on page 2-59, that is, a numeric or logical vector, character matrix, cell vector of character

vectors, or categorical vector. `TreeBagger` converts labels to a cell array of character vectors for classification. For regression, `Y` is a numeric vector.

## Examples

### Train Ensemble of Bagged Classification Trees

Load Fisher's iris data set.

```
load fisheriris
```

Train an ensemble of bagged classification trees using the entire data set. Specify 50 weak learners. Store which observations are out of bag for each tree.

```
rng(1); % For reproducibility
Mdl = TreeBagger(50, meas, species, 'OOBPrediction', 'On', ...
    'Method', 'classification')
```

```
Mdl =
```

```
    TreeBagger
Ensemble with 50 bagged decision trees:
           Training X:           [150x4]
           Training Y:           [150x1]
           Method:           classification
           NumPredictors:           4
           NumPredictorsToSample:           2
           MinLeafSize:           1
           InBagFraction:           1
           SampleWithReplacement:           1
           ComputeOOBPrediction:           1
           ComputeOOBPredictorImportance:           0
           Proximity:           []
           ClassNames:           'setosa'   'versicolor'   'virginica'
```

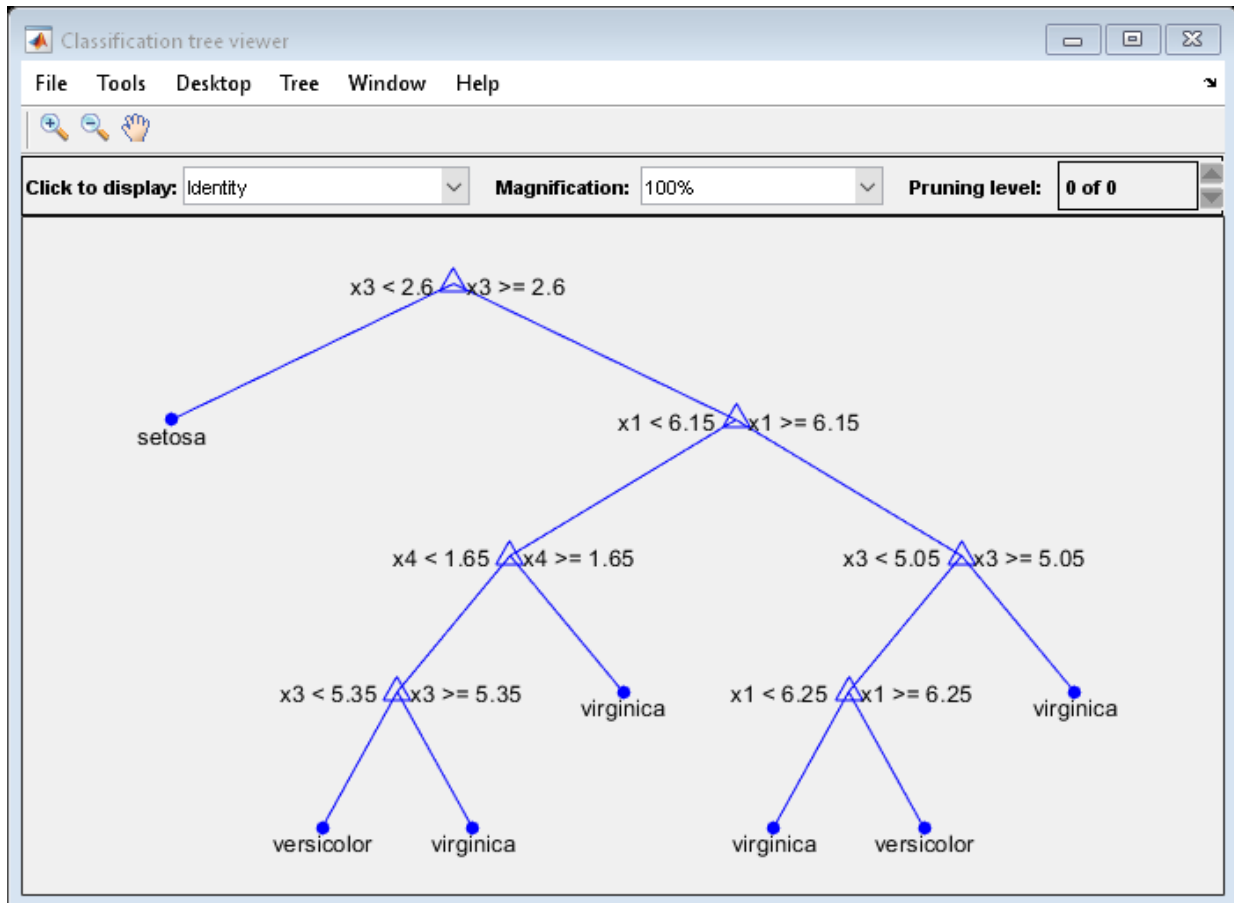
`Mdl` is a `TreeBagger` ensemble.

`Mdl.Trees` stores a 50-by-1 cell vector of the trained classification trees (`CompactClassificationTree` model objects) that compose the ensemble.



Plot a graph of the first trained classification tree.

```
view(Mdl.Trees{1}, 'Mode', 'graph')
```



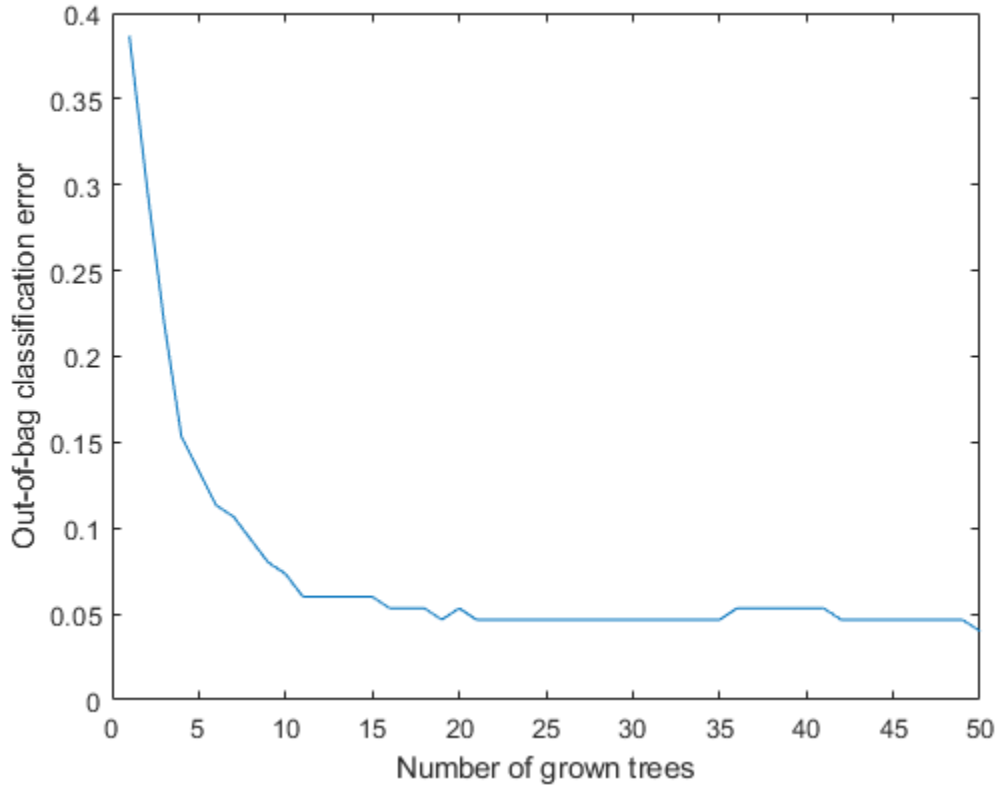
By default, `TreeBagger` grows deep trees.

`Mdl.OOBIndices` stores the out-of-bag indices as a matrix of logical values.

Plot the out-of-bag error over the number of grown classification trees.

```
figure;  
oobErrorBaggedEnsemble = oobError(Mdl);
```

```
plot(oobErrorBaggedEnsemble)
xlabel 'Number of grown trees';
ylabel 'Out-of-bag classification error';
```



The out-of-bag error decreases with the number of grown trees.

To label out-of-bag observations, pass `Mdl` to `oobPredict`.

### Train Ensemble of Bagged Regression Trees

Load the `carsmall` data set. Consider a model that predicts the fuel economy of a car given its engine displacement.

```
load carsmall
```

Train an ensemble of bagged regression trees using the entire data set. Specify 100 weak learners.

```
rng(1); % For reproducibility
Mdl = TreeBagger(100, Displacement, MPG, 'Method', 'regression');
```

Mdl is a TreeBagger ensemble.

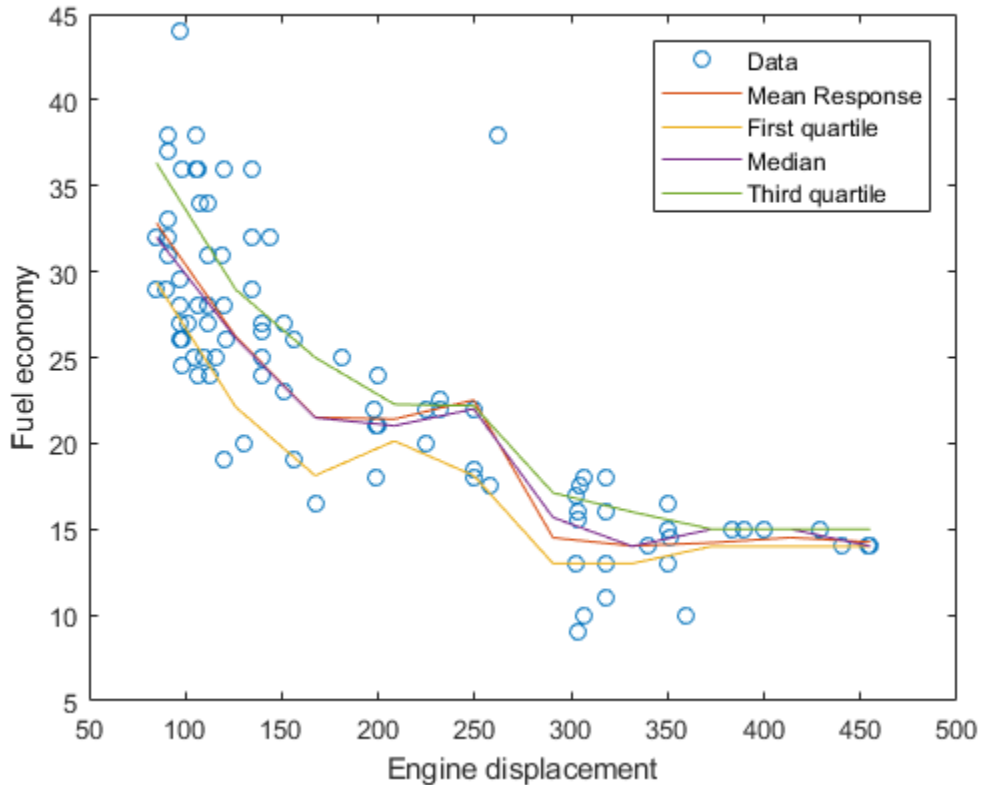
Using a trained bag of regression trees, you can estimate conditional mean responses or perform quantile regression to predict conditional quantiles.

For ten equally-spaced engine displacements between the minimum and maximum in-sample displacement, predict conditional mean responses and conditional quartiles.

```
predX = linspace(min(Displacement), max(Displacement), 10)';
mpgMean = predict(Mdl, predX);
mpgQuartiles = quantilePredict(Mdl, predX, 'Quantile', [0.25, 0.5, 0.75]);
```

Plot the observations, and estimated mean responses and quartiles in the same figure.

```
figure;
plot(Displacement, MPG, 'o');
hold on
plot(predX, mpgMean);
plot(predX, mpgQuartiles);
ylabel('Fuel economy');
xlabel('Engine displacement');
legend('Data', 'Mean Response', 'First quartile', 'Median', 'Third quartile');
```



### Unbiased Predictor Importance Estimates

Load the `carsmall` data set. Consider a model that predicts the mean fuel economy of a car given its acceleration, number of cylinders, engine displacement, horsepower, manufacturer, model year, and weight. Consider `Cylinders`, `Mfg`, and `Model_Year` as categorical variables.

```
load carsmall
Cylinders = categorical(Cylinders);
Mfg = categorical(cellstr(Mfg));
Model_Year = categorical(Model_Year);
X = table(Acceleration,Cylinders,Displacement,Horsepower,Mfg,...
```

```
Model_Year,Weight,MPG);  
rng('default'); % For reproducibility
```

Display the number of categories represented in the categorical variables.

```
numCylinders = numel(categories(Cylinders))  
numMfg = numel(categories(Mfg))  
numModelYear = numel(categories(Model_Year))
```

```
numCylinders =
```

```
3
```

```
numMfg =
```

```
28
```

```
numModelYear =
```

```
3
```

Because there are 3 categories only in `Cylinders` and `Model_Year`, the standard CART, predictor-splitting algorithm prefers splitting a continuous predictor over these two variables.

Train a random forest of 200 regression trees using the entire data set. To grow unbiased trees, specify usage of the curvature test for splitting predictors. Because there are missing values in the data, specify usage of surrogate splits. Store the out-of-bag information for predictor importance estimation.

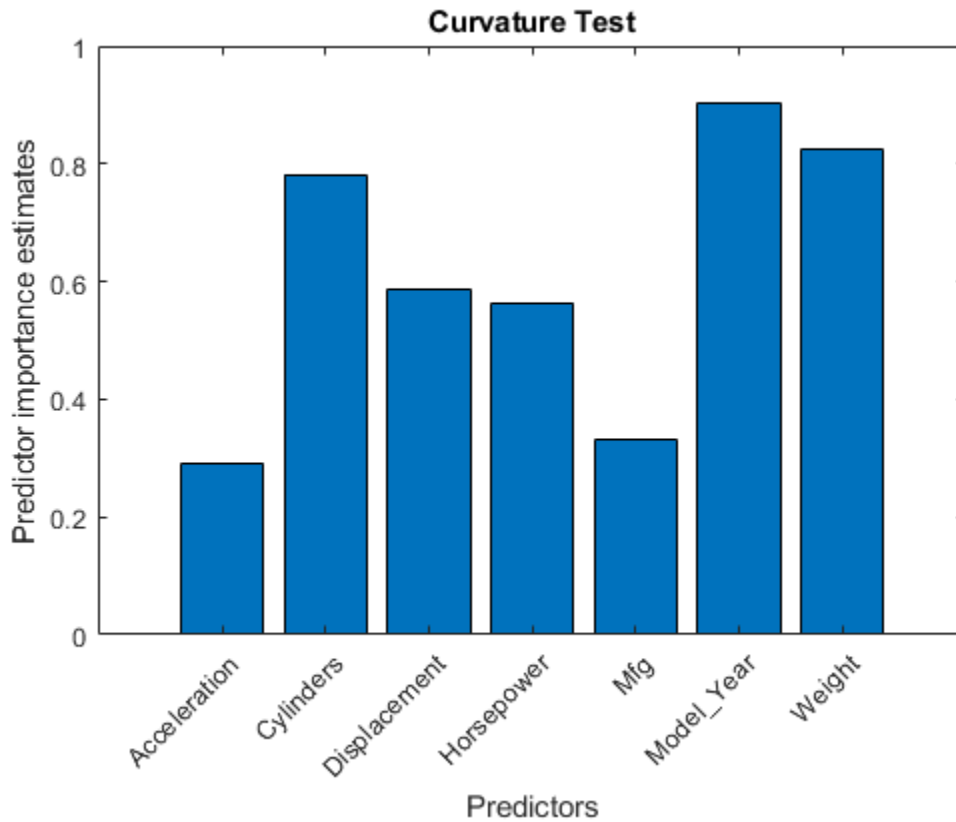
```
Mdl = TreeBagger(200,X,'MPG','Method','regression','Surrogate','on',...  
    'PredictorSelection','curvature','OOBPredictorImportance','on');
```

`TreeBagger` stores predictor importance estimates in the property `OOBPermutedPredictorDeltaError`. Compare the estimates using a bar graph.

```
imp = Mdl.OOBPermutedPredictorDeltaError;
```

```
figure;  
bar(imp);
```

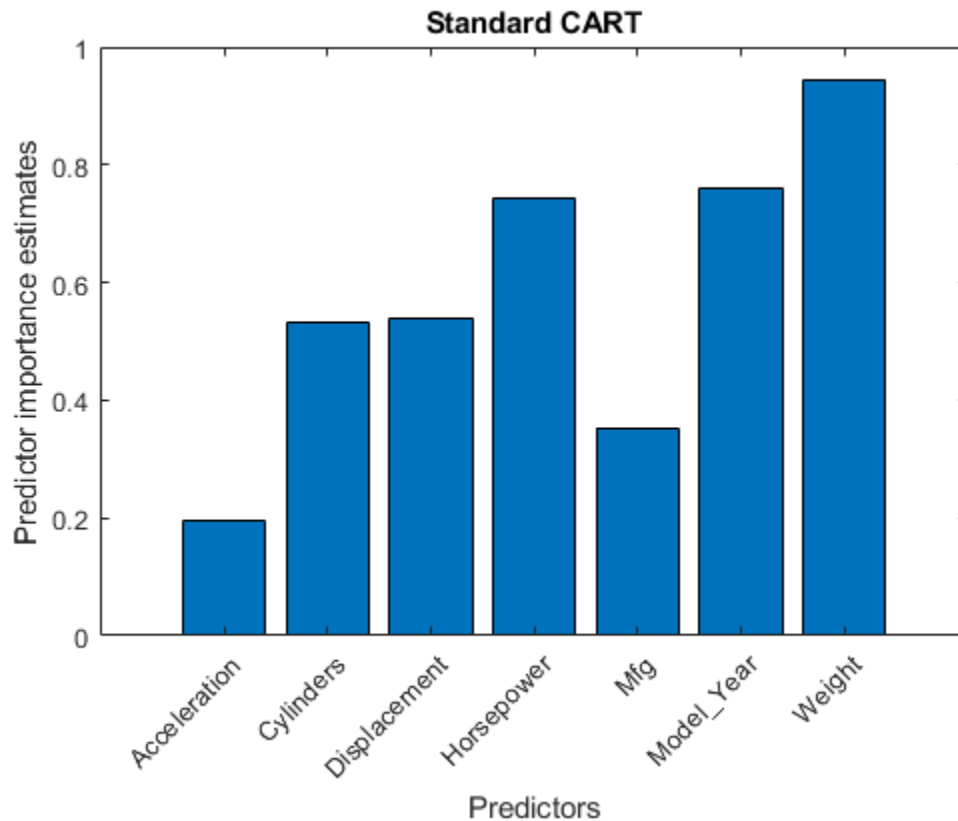
```
title('Curvature Test');  
ylabel('Predictor importance estimates');  
xlabel('Predictors');  
h = gca;  
h.XTickLabel = Mdl.PredictorNames;  
h.XTickLabelRotation = 45;  
h.TickLabelInterpreter = 'none';
```



In this case, `Model_Year` is the most important predictor, followed by `Weight`.

Compare the `imp` to predictor importance estimates computed from a random forest that grows trees using standard CART.

```
MdlCART = TreeBagger(200,X,'MPG','Method','regression','Surrogate','on',...  
    'OOBPredictorImportance','on');  
  
impCART = MdlCART.OOBPermutedPredictorDeltaError;  
  
figure;  
bar(impCART);  
title('Standard CART');  
ylabel('Predictor importance estimates');  
xlabel('Predictors');  
h = gca;  
h.XTickLabel = Mdl.PredictorNames;  
h.XTickLabelRotation = 45;  
h.TickLabelInterpreter = 'none';
```



In this case, `Weight`, a continuous predictor, is the most important. The next two most importance predictor are `Model_Year` followed closely by `Horsepower`, which is a continuous predictor.

- “Bootstrap Aggregation (Bagging) of Regression Trees” on page 18-141
- “Bootstrap Aggregation (Bagging) of Classification Trees” on page 18-154
- “Framework for Ensemble Learning” on page 18-41
- “Decision Trees” on page 19-2
- “Grouping Variables” on page 2-59

## Copy Semantics

Value. To learn how this affects your use of the class, see [Comparing Handle and Value Classes \(MATLAB\)](#) in the MATLAB Object-Oriented Programming documentation.

## Tip

For a `TreeBagger` model object `B`, the `Trees` property stores a cell vector of `B.NumTrees` `CompactClassificationTree` or `CompactRegressionTree` model objects. For a textual or graphical display of tree `t` in the cell vector, enter

```
view(B.Trees{t})
```

## References

- [1] Breiman, L. *Random Forests*. *Machine Learning* 45, pp. 5–32, 2001.
- [2] Meinshausen, N. “Quantile Regression Forests.” *Journal of Machine Learning Research*, Vol. 7, 2006, pp. 983–999.

## See Also

`CompactTreeBagger` | `TreeBagger` | `compact` | `error` | `oobError` | `oobPredict` | `plotPartialDependence` | `predict` | `view` | `view`



## Topics

“Bootstrap Aggregation (Bagging) of Regression Trees” on page 18-141

“Bootstrap Aggregation (Bagging) of Classification Trees” on page 18-154

“Framework for Ensemble Learning” on page 18-41

“Decision Trees” on page 19-2

“Grouping Variables” on page 2-59

## TreeBagger

**Class:** TreeBagger

Create bag of decision trees

Individual decision trees tend to overfit. Bootstrap-aggregated (bagged) decision trees combine the results of many decision trees, which reduces the effects of overfitting and improves generalization. `TreeBagger` grows the decision trees in the ensemble using a bootstrap samples of the data. Also, `TreeBagger` selects a random subset of predictors to use at each decision split as in the random forest algorithm [1].

By default, `TreeBagger` bags classification trees. To bag regression trees instead, specify `'Method', 'regression'`.

For regression problems, `TreeBagger` supports mean and quantile regression (that is, quantile regression forest [5]).

## Syntax

```
Mdl = TreeBagger (NumTrees, Tbl, ResponseVarName)
Mdl = TreeBagger (NumTrees, Tbl, formula)
Mdl = TreeBagger (NumTrees, Tbl, Y)
B = TreeBagger (NumTrees, X, Y)
B = TreeBagger (NumTrees, X, Y, Name, Value)
```

## Description

`Mdl = TreeBagger (NumTrees, Tbl, ResponseVarName)` returns a ensemble of `NumTrees` bagged classification trees trained using the sample data in the table `Tbl`. `ResponseVarName` is the name of the response variable in `Tbl`.

`Mdl = TreeBagger (NumTrees, Tbl, formula)` returns an ensemble of bagged classification trees trained using the sample data in the table `Tbl`. `formula` is an explanatory model of the response and a subset of predictor variables in `Tbl` used to fit `Mdl`. Specify `Formula` using Wilkinson notation. For more information, see “Wilkinson Notation” on page 11-117.

`Mdl = TreeBagger(NumTrees, Tbl, Y)` returns an ensemble of classification trees using the predictor variables in table `Tbl` and class labels in vector `Y`.

`Y` is an array of response data. Elements of `Y` correspond to the rows of `Tbl`. For classification, `Y` is the set of true class labels. Labels can be any grouping variable on page 2-59, that is, a numeric or logical vector, character matrix, cell vector of character vectors, or categorical vector. `TreeBagger` converts labels to a cell array of character vectors. For regression, `Y` is a numeric vector. To grow regression trees, you must specify the name-value pair `'Method', 'regression'`.

`B = TreeBagger(NumTrees, X, Y)` creates an ensemble `B` of `NumTrees` decision trees for predicting response `Y` as a function of predictors in the numeric matrix of training data, `X`. Each row in `X` represents an observation and each column represents a predictor or feature.

`B = TreeBagger(NumTrees, X, Y, Name, Value)` specifies optional parameter name-value pairs:

'InBagFraction'	Fraction of input data to sample with replacement from the input data for growing each new tree. Default value is 1.
-----------------	--

'Cost'	<p>Square matrix <math>C</math>, where <math>C(i, j)</math> is the cost of classifying a point into class <math>j</math> if its true class is <math>i</math> (i.e., the rows correspond to the true class and the columns correspond to the predicted class). The order of the rows and columns of <code>Cost</code> corresponds to the order of the classes in the <code>ClassNames</code> property of the trained <code>TreeBagger</code> model <code>B</code>.</p> <p>Alternatively, <code>cost</code> can be a structure <code>S</code> having two fields:</p> <ul style="list-style-type: none"> <li>• <code>S.ClassNames</code> containing the group names as a categorical variable, character array, or cell vector of character vectors</li> <li>• <code>S.ClassificationCosts</code> containing the cost matrix <code>C</code></li> </ul> <p>The default value is <math>C(i, j) = 1</math> if <math>i \neq j</math>, and <math>C(i, j) = 0</math> if <math>i = j</math>.</p> <p>If <code>Cost</code> is highly skewed, then, for in-bag samples, the software oversamples unique observations from the class that has a large penalty. For smaller sample sizes, this might cause a very low relative frequency of out-of-bag observations from the class that has a large penalty. Therefore, the estimated out-of-bag error is highly variable, and might be difficult to interpret.</p>
'SampleWithReplacement'	<p>'on' to sample with replacement or 'off' to sample without replacement. If you sample without replacement, you need to set <code>'InBagFraction'</code> to a value less than one. Default is 'on'.</p>
'OOBPrediction'	<p>'on' to store info on what observations are out of bag for each tree. This info can be used by <code>oobPrediction</code> to compute the predicted class probabilities for each tree in the ensemble. Default is 'off'.</p>
'OOBPredictorImportance'	<p>'on' to store out-of-bag estimates of feature importance in the ensemble. Default is 'off'. Specifying 'on' also sets the <code>'OOBPrediction'</code> value to 'on'. If an analysis of predictor importance is your goal, then also specify <code>'PredictorSelection'</code>, <code>'curvature'</code> or <code>'PredictorSelection','interaction-curvature'</code>. For more details, see <code>fitctree</code> or <code>fitrtree</code>.</p>

---

'Method'	Either 'classification' or 'regression'. Regression requires a numeric Y.
'NumPredictorsToSample'	Number of variables to select at random for each decision split. Default is the square root of the number of variables for classification and one third of the number of variables for regression. Valid values are 'all' or a positive integer. Setting this argument to any valid value but 'all' invokes Breiman's random forest algorithm [1].
'NumPrint'	Number of training cycles (grown trees) after which TreeBagger displays a diagnostic message showing training progress. Default is no diagnostic messages.
'MinLeafSize'	Minimum number of observations per tree leaf. Default is 1 for classification and 5 for regression.

'Options'	<p>A structure that specifies options that govern the computation when growing the ensemble of decision trees. One option requests that the computation of decision trees on multiple bootstrap replicates uses multiple processors, if the Parallel Computing Toolbox is available. Two options specify the random number streams to use in selecting bootstrap replicates. You can create this argument with a call to <code>statset</code>. You can retrieve values of the individual fields with a call to <code>statget</code>. Applicable <code>statset</code> parameters are:</p> <ul style="list-style-type: none"><li>• <code>'UseParallel'</code> — If <code>true</code> and if a <code>parpool</code> of the Parallel Computing Toolbox is open, compute decision trees drawn on separate bootstrap replicates in parallel. If the Parallel Computing Toolbox is not installed, or a <code>parpool</code> is not open, computation occurs in serial mode. Default is <code>false</code>, or serial computation.</li><li>• <code>'UseSubstreams'</code> — If <code>true</code> select each bootstrap replicate using a separate <code>Substream</code> of the random number generator (aka <code>Stream</code>). This option is available only with <code>RandStream</code> types that support <code>Substreams</code>: <code>'mlfg6331_64'</code> or <code>'mrg32k3a'</code>. Default is <code>false</code>, do not use a different <code>Substream</code> to compute each bootstrap replicate.</li><li>• <code>Streams</code> — A <code>RandStream</code> object or cell array of such objects. If you do not specify <code>Streams</code>, <code>TreeBagger</code> uses the default stream or streams. If you choose to specify <code>Streams</code>, use a single object except in the case<ul style="list-style-type: none"><li>• You have an open <code>Parallel</code> pool</li><li>• <code>UseParallel</code> is <code>true</code></li><li>• <code>UseSubstreams</code> is <code>false</code></li></ul></li></ul> <p>In that case, use a cell array the same size as the <code>Parallel</code> pool.</p>
-----------	--

'Prior'	<p>Prior probabilities for each class. Specify as one of:</p> <ul style="list-style-type: none"><li>• A character vector:<ul style="list-style-type: none"><li>• 'Empirical' determines class probabilities from class frequencies in <math>Y</math>. If you pass observation weights, they are used to compute the class probabilities. This is the default.</li><li>• 'Uniform' sets all class probabilities equal.</li></ul></li><li>• A vector (one scalar value for each class). The order of the elements <code>Prior</code> corresponds to the order of the classes in the <code>ClassNames</code> property of the trained <code>TreeBagger</code> model <code>B</code>.</li><li>• A structure <code>S</code> with two fields:<ul style="list-style-type: none"><li>• <code>S.ClassNames</code> containing the class names as a categorical variable, character array, or cell vector of character vectors</li><li>• <code>S.ClassProbs</code> containing a vector of corresponding probabilities</li></ul></li></ul> <p>If you set values for both <code>Weights</code> and <code>Prior</code>, the weights are renormalized to add up to the value of the prior probability in the respective class.</p> <p>If <code>Prior</code> is highly skewed, then, for in-bag samples, the software oversamples unique observations from the class that has a large prior probability. For smaller sample sizes, this might cause a very low relative frequency of out-of-bag observations from the class that has a large prior probability. Therefore, the estimated out-of-bag error is highly variable, and might be difficult to interpret.</p>
---------	--

'PredictorNames'	<p>Predictor variable names, specified as the comma-separated pair consisting of 'PredictorNames' and a cell array of unique character vectors. The functionality of 'PredictorNames' depends on the way you supply the training data.</p> <ul style="list-style-type: none"><li>• If you supply <math>X</math> and <math>Y</math>, then you can use 'PredictorNames' to give the predictor variables in <math>X</math> names.<ul style="list-style-type: none"><li>• The order of the names in <code>PredictorNames</code> must correspond to the column order of <math>X</math>. That is, <code>PredictorNames{1}</code> is the name of <math>X(:, 1)</math>, <code>PredictorNames{2}</code> is the name of <math>X(:, 2)</math>, and so on. Also, <code>size(X, 2)</code> and <code>numel(PredictorNames)</code> must be equal.</li><li>• By default, <code>PredictorNames</code> is <code>{'x1', 'x2', ...}</code>.</li></ul></li><li>• If you supply <code>Tbl</code>, then you can use 'PredictorNames' to choose which predictor variables to use in training. That is, <code>TreeBagger</code> uses the predictor variables in <code>PredictorNames</code> and the response only in training.<ul style="list-style-type: none"><li>• <code>PredictorNames</code> must be a subset of <code>Tbl.Properties.VariableNames</code> and cannot include the name of the response variable.</li><li>• By default, <code>PredictorNames</code> contains the names of all predictor variables.</li><li>• It good practice to specify the predictors for training using one of 'PredictorNames' or formula only.</li></ul></li></ul>
------------------	--



'CategoricalPredictors'	<p>Categorical predictors list, specified as the comma-separated pair consisting of 'CategoricalPredictors' and one of the following.</p> <ul style="list-style-type: none"> <li>• A numeric vector with indices from 1 to p, where p is the number of columns of X.</li> <li>• A logical vector of length p, where a true entry means that the corresponding column of X is a categorical variable.</li> <li>• A cell vector of character vectors, where each element in the array is the name of a predictor variable. The names must match entries in PredictorNames values.</li> <li>• A character matrix, where each row of the matrix is a name of a predictor variable. The names must match entries in PredictorNames values. Pad the names with extra blanks so each row of the character matrix has the same length.</li> <li>• 'all', meaning all predictors are categorical.</li> </ul>
'ChunkSize'	<p>Chunk size, specified as the comma-separated pair consisting of 'ChunkSize' and a positive integer. The chunk size specifies the number of observations in each chunk of data. The default value is 50000.</p> <p><b>Note</b> This option only applies when using TreeBagger on tall arrays. See “Extended Capabilities” on page 32-0 for more information.</p>

In addition to the optional arguments above, TreeBagger accepts these optional fitctree and fitrtree arguments.

Supported fitctree arguments	Supported fitrtree arguments
AlgorithmForCategorical	MaxNumSplits
MaxNumCategories	MergeLeaves
MaxNumSplits	PredictorSelection
MergeLeaves	Prune
PredictorSelection	PruneCriterion
Prune	QuadraticErrorTolerance

Supported <code>fitctree</code> arguments	Supported <code>fitrtree</code> arguments
PruneCriterion	SplitCriterion
SplitCriterion	Surrogate
Surrogate	Weights
Weights	

## Examples

### Train Ensemble of Bagged Classification Trees

Load Fisher's iris data set.

```
load fisheriris
```

Train an ensemble of bagged classification trees using the entire data set. Specify 50 weak learners. Store which observations are out of bag for each tree.

```
rng(1); % For reproducibility
Mdl = TreeBagger(50, meas, species, 'OOBPrediction', 'On', ...
    'Method', 'classification')
```

```
Mdl =
```

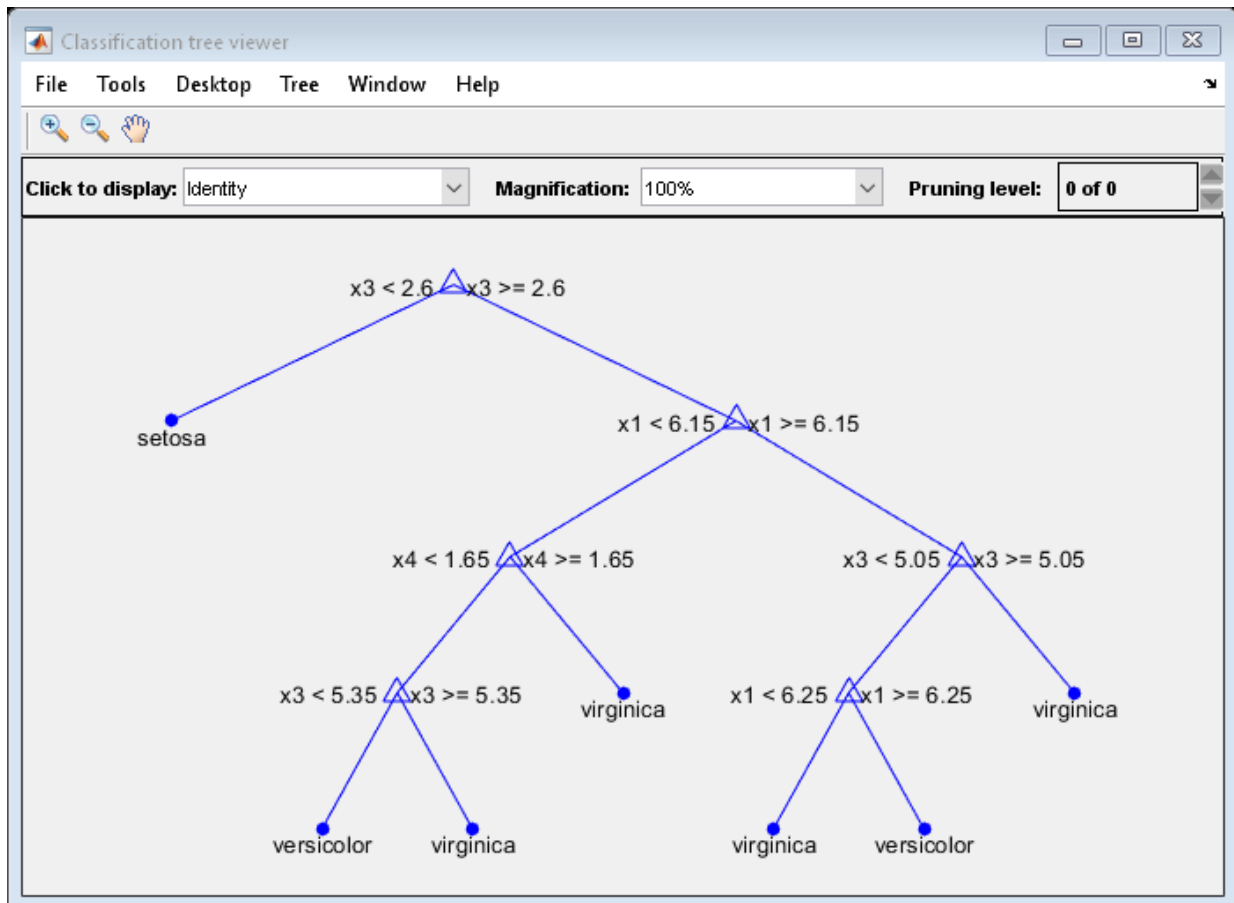
```
TreeBagger
Ensemble with 50 bagged decision trees:
    Training X:          [150x4]
    Training Y:          [150x1]
    Method:              classification
    NumPredictors:       4
    NumPredictorsToSample: 2
    MinLeafSize:         1
    InBagFraction:       1
    SampleWithReplacement: 1
    ComputeOOBPrediction: 1
    ComputeOOBPredictorImportance: 0
    Proximity:           []
    ClassNames:          'setosa'   'versicolor'   'virginica'
```

Mdl is a TreeBagger ensemble.

Mdl.Trees stores a 50-by-1 cell vector of the trained classification trees (CompactClassificationTree model objects) that compose the ensemble.

Plot a graph of the first trained classification tree.

```
view(Mdl.Trees{1}, 'Mode', 'graph')
```

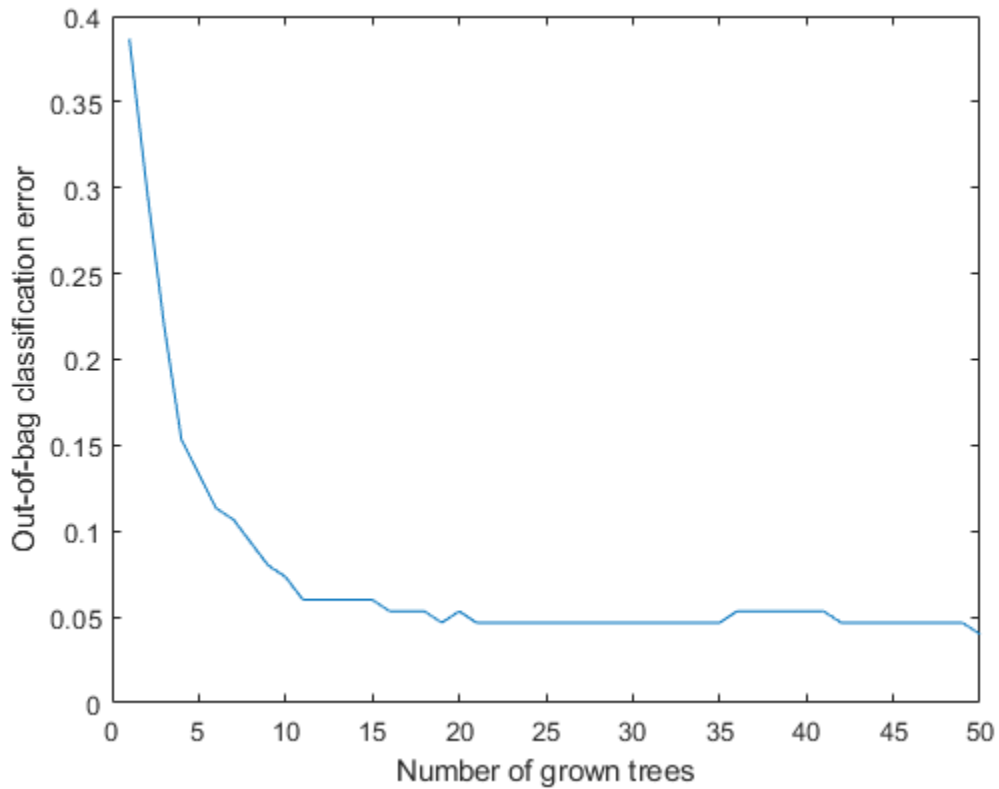


By default, TreeBagger grows deep trees.

Mdl.OOBIndices stores the out-of-bag indices as a matrix of logical values.

Plot the out-of-bag error over the number of grown classification trees.

```
figure;  
oobErrorBaggedEnsemble = oobError(Mdl);  
plot(oobErrorBaggedEnsemble)  
xlabel 'Number of grown trees';  
ylabel 'Out-of-bag classification error';
```



The out-of-bag error decreases with the number of grown trees.

To label out-of-bag observations, pass `Mdl` to `oobPredict`.

## Train Ensemble of Bagged Regression Trees

Load the `carsmall` data set. Consider a model that predicts the fuel economy of a car given its engine displacement.

```
load carsmall
```

Train an ensemble of bagged regression trees using the entire data set. Specify 100 weak learners.

```
rng(1); % For reproducibility
Mdl = TreeBagger(100, Displacement, MPG, 'Method', 'regression');
```

`Mdl` is a `TreeBagger` ensemble.

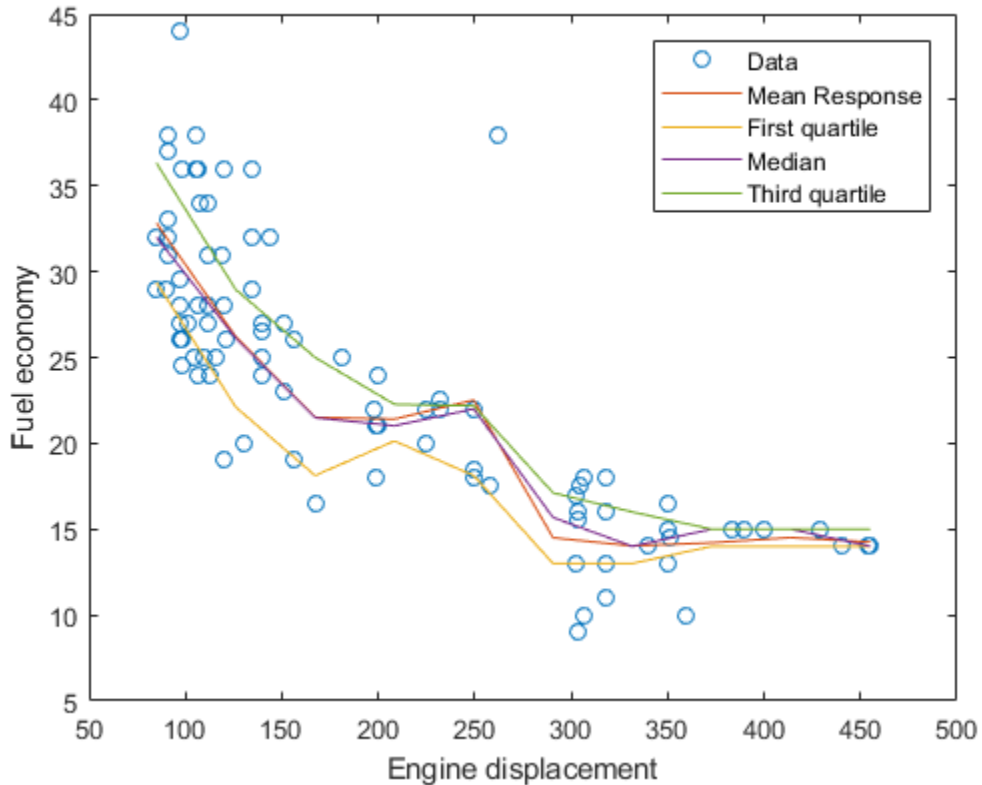
Using a trained bag of regression trees, you can estimate conditional mean responses or perform quantile regression to predict conditional quantiles.

For ten equally-spaced engine displacements between the minimum and maximum in-sample displacement, predict conditional mean responses and conditional quantiles.

```
predX = linspace(min(Displacement), max(Displacement), 10)';
mpgMean = predict(Mdl, predX);
mpgQuartiles = quantilePredict(Mdl, predX, 'Quantile', [0.25, 0.5, 0.75]);
```

Plot the observations, and estimated mean responses and quartiles in the same figure.

```
figure;
plot(Displacement, MPG, 'o');
hold on
plot(predX, mpgMean);
plot(predX, mpgQuartiles);
ylabel('Fuel economy');
xlabel('Engine displacement');
legend('Data', 'Mean Response', 'First quartile', 'Median', 'Third quartile');
```



### Unbiased Predictor Importance Estimates

Load the `carsmall` data set. Consider a model that predicts the mean fuel economy of a car given its acceleration, number of cylinders, engine displacement, horsepower, manufacturer, model year, and weight. Consider `Cylinders`, `Mfg`, and `Model_Year` as categorical variables.

```
load carsmall
Cylinders = categorical(Cylinders);
Mfg = categorical(cellstr(Mfg));
Model_Year = categorical(Model_Year);
X = table(Acceleration,Cylinders,Displacement,Horsepower,Mfg,...
```

```

    Model_Year,Weight,MPG);
rng('default'); % For reproducibility

```

Display the number of categories represented in the categorical variables.

```

numCylinders = numel(categories(Cylinders))
numMfg = numel(categories(Mfg))
numModelYear = numel(categories(Model_Year))

```

```

numCylinders =

```

```

    3

```

```

numMfg =

```

```

    28

```

```

numModelYear =

```

```

    3

```

Because there are 3 categories only in `Cylinders` and `Model_Year`, the standard CART, predictor-splitting algorithm prefers splitting a continuous predictor over these two variables.

Train a random forest of 200 regression trees using the entire data set. To grow unbiased trees, specify usage of the curvature test for splitting predictors. Because there are missing values in the data, specify usage of surrogate splits. Store the out-of-bag information for predictor importance estimation.

```

Mdl = TreeBagger(200,X,'MPG','Method','regression','Surrogate','on',...
    'PredictorSelection','curvature','OOBPredictorImportance','on');

```

`TreeBagger` stores predictor importance estimates in the property `OOBPermutedPredictorDeltaError`. Compare the estimates using a bar graph.

```

imp = Mdl.OOBPermutedPredictorDeltaError;

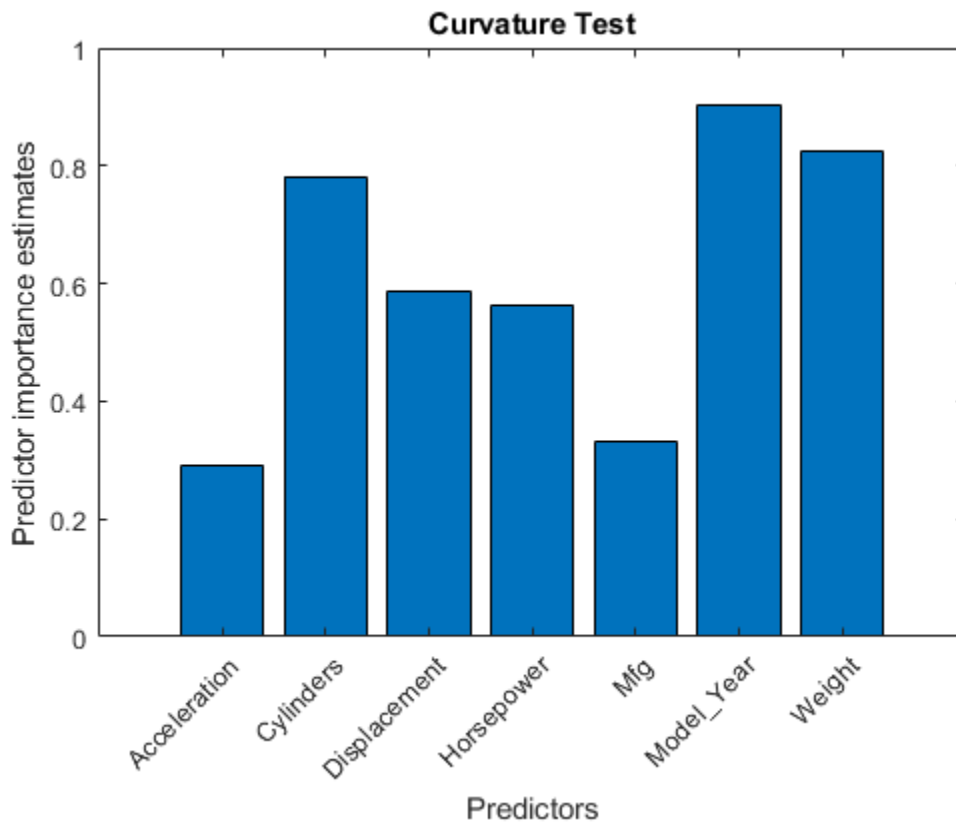
```

```

figure;
bar(imp);

```

```
title('Curvature Test');  
ylabel('Predictor importance estimates');  
xlabel('Predictors');  
h = gca;  
h.XTickLabel = Mdl.PredictorNames;  
h.XTickLabelRotation = 45;  
h.TickLabelInterpreter = 'none';
```

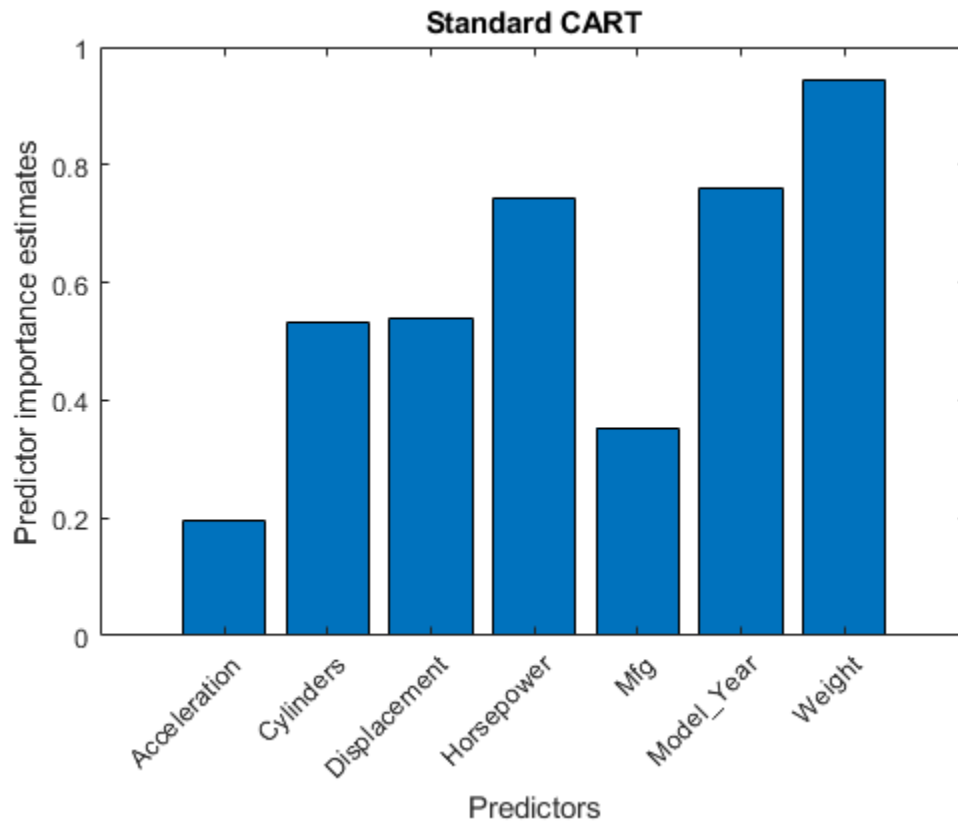


In this case, `Model_Year` is the most important predictor, followed by `Weight`.

Compare the `imp` to predictor importance estimates computed from a random forest that grows trees using standard CART.



```
MdlCART = TreeBagger(200,X,'MPG','Method','regression','Surrogate','on',...  
    'OOBPredictorImportance','on');  
  
impCART = MdlCART.OOBPermutedPredictorDeltaError;  
  
figure;  
bar(impCART);  
title('Standard CART');  
ylabel('Predictor importance estimates');  
xlabel('Predictors');  
h = gca;  
h.XTickLabel = Mdl.PredictorNames;  
h.XTickLabelRotation = 45;  
h.TickLabelInterpreter = 'none';
```



In this case, `Weight`, a continuous predictor, is the most important. The next two most importance predictor are `Model_Year` followed closely by `Horsepower`, which is a continuous predictor.

- “Bootstrap Aggregation (Bagging) of Regression Trees” on page 18-141
- “Bootstrap Aggregation (Bagging) of Classification Trees” on page 18-154
- “Framework for Ensemble Learning” on page 18-41
- “Decision Trees” on page 19-2
- “Grouping Variables” on page 2-59

## Tips

- Avoid large estimated out-of-bag error variances by setting a more balanced misclassification cost matrix or a less skewed prior probability vector.
- The `Trees` property of `B` stores a cell vector of `B.NumTrees` `CompactClassificationTree` or `CompactRegressionTree` model objects. For a textual or graphical display of tree `t` in the cell vector, enter

```
view(B.Trees{t})
```

- Standard CART tends to select split predictors containing many distinct values, e.g., continuous variables, over those containing few distinct values, e.g., categorical variables [4]. Consider specifying the curvature or interaction test if any of the following are true:
  - If there are predictors that have relatively fewer distinct values than other predictors, for example, if the predictor data set is heterogeneous.
  - If an analysis of predictor importance is your goal. `TreeBagger` stores predictor importance estimates in the `OOBPermutedPredictorDeltaError` property of `Mdl`.

For more information on predictor selection, see `PredictorSelection` for classification trees or `PredictorSelection` for regression trees.

## Algorithms

- `TreeBagger` generates in-bag samples by oversampling classes with large misclassification costs and undersampling classes with small misclassification costs. Consequently, out-of-bag samples have fewer observations from classes with large misclassification costs and more observations from classes with small misclassification costs. If you train a classification ensemble using a small data set and a highly skewed cost matrix, then the number of out-of-bag observations per class might be very low. Therefore, the estimated out-of-bag error might have a large variance and might be difficult to interpret. The same phenomenon can occur for classes with large prior probabilities.
- For details on selecting split predictors and node-splitting algorithms when growing decision trees, see “Algorithms” on page 32-2174 for classification trees and “Algorithms” on page 32-2174 for regression trees.

## References

- [1] Breiman, L. *Random Forests*. Machine Learning 45, pp. 5–32, 2001.
- [2] Breiman, L., J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Boca Raton, FL: CRC Press, 1984.
- [3] Loh, W.Y. “Regression Trees with Unbiased Variable Selection and Interaction Detection.” *Statistica Sinica*, Vol. 12, 2002, pp. 361–386.
- [4] Loh, W.Y. and Y.S. Shih. “Split Selection Methods for Classification Trees.” *Statistica Sinica*, Vol. 7, 1997, pp. 815–840.
- [5] Meinshausen, N. “Quantile Regression Forests.” *Journal of Machine Learning Research*, Vol. 7, 2006, pp. 983–999.

## Extended Capabilities

### Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function supports tall arrays for out-of-memory data with the limitations:

- Supported syntaxes for tall `X`, `Y`, `Tbl` are:
  - `B = TreeBagger (NumTrees, Tbl, Y)`
  - `B = TreeBagger (NumTrees, X, Y)`
  - `B = TreeBagger (___, Name, Value)`
- For tall arrays, `TreeBagger` supports classification. Regression is not supported.
- Supported name-value pairs are:
  - `'NumPredictorsToSample'` — Default value is the square root of the number of variables for classification.
  - `'MinLeafSize'` — Default value is 1 if the number of observations is less than 50,000. If the number of observations is larger than 50,000, then the default value is `max(1, min(5, floor(0.01*NobsChunk)))`.
  - `'ChunkSize'` (only for tall arrays) — Default value is 50000.

In addition, `TreeBagger` supports these optional arguments of `fitctree`:

- `'AlgorithmForCategorical'`
- `'CategoricalPredictors'`
- `'MaxNumCategories'`
- `'MergeLeaves'`
- `'PredictorNames'`
- `'PredictorSelection'`
- `'Prune'`
- `'PruneCriterion'`
- `'Surrogate'`
- `'MaxNumSplits'`
- `'SplitCriterion'`
- For tall data, `TreeBagger` returns a `CompactTreeBagger` object that contains most of the same properties as a full `TreeBagger` object. The main difference is that the compact object is more memory efficient. The compact object does not include properties that include the data, or that include an array of the same size as the data.
- Supported `CompactTreeBagger` methods are:

- `combine`
- `error`
- `margin`
- `meanMargin`
- `predict`
- `setDefaultYfit`

The `error`, `margin`, `meanMargin`, and `predict` methods do not support the name-value pairs `'Trees'`, `'TreeWeights'`, or `'UseInstanceForTree'`. The `error` and `meanMargin` methods additionally do not support `'Weights'`.

- `TreeBagger` creates a random forest by generating trees on disjoint chunks of the data. When more data is available than is required to create the random forest, the data is subsampled. For a similar example, see *Random Forests for Big Data* (Genuer, Poggi, Tuleau-Malot, Villa-Vialaneix 2015).

Depending on how the data is stored, it is possible that some chunks of data contain observations from only a few classes out of all the classes. In this case, `TreeBagger` might produce inferior results compared to the case where each chunk of data contains observations from most of the classes.

For more information, see “Tall Arrays” (MATLAB).

## See Also

`TreeBagger` | `compact` | `error` | `fitctree` | `fitrtree` | `oobError` | `oobPredict` | `predict` | `statset` | `view` | `view`

## Topics

“Bootstrap Aggregation (Bagging) of Regression Trees” on page 18-141

“Bootstrap Aggregation (Bagging) of Classification Trees” on page 18-154

“Framework for Ensemble Learning” on page 18-41

“Decision Trees” on page 19-2

“Grouping Variables” on page 2-59

**Introduced in R2009a**

## Trees property

**Class:** TreeBagger

Decision trees in ensemble

### Description

The `Trees` property is a cell array of size `NTrees-by-1` containing the trees in the ensemble.

### See Also

`NTrees`

# trimmean

Mean excluding outliers

## Syntax

```
m = trimmean(X,percent)
trimmean(X,percent,dim)
m = trimmean(X,percent,flag)
m = trimmean(x,percent,flag,dim)
```

## Description

`m = trimmean(X,percent)` calculates the trimmed mean of the values in `X`. For a vector input, `m` is the mean of `X`, excluding the highest and lowest `k` data values, where  $k = n * (\text{percent}/100) / 2$  and where `n` is the number of values in `X`. For a matrix input, `m` is a row vector containing the trimmed mean of each column of `X`. For `n`-D arrays, `trimmean` operates along the first non-singleton dimension. `percent` is a scalar between 0 and 100.

`trimmean(X,percent,dim)` takes the trimmed mean along dimension `dim` of `X`.

`m = trimmean(X,percent,flag)` controls how to trim when `k` is not an integer. `flag` can be chosen from the following:

'round'	Round <code>k</code> to the nearest integer (round to a smaller integer if <code>k</code> is a half integer). This is the default.
'floor'	Round <code>k</code> down to the next smaller integer.
'weight'	If $k = i + f$ where <code>i</code> is the integer part and <code>f</code> is the fraction, compute a weighted mean with weight $(1 - f)$ for the $(i + 1)$ th and $(n - i)$ th values, and full weight for the values between them.

`m = trimmean(x,percent,flag,dim)` takes the trimmed mean along dimension `dim` of `x`.

## Examples

### Efficiency of the Trimmed Mean

Generate a 100-by-100 matrix of random numbers from the standard normal distribution. This represents 100 samples, each containing 100 data points.

```
rng default; % For reproducibility
x = normrnd(0,1,100,100);
```

Compute the sample mean and the 10% trimmed mean for each column of the data matrix.

```
m = mean(x);
trim = trimmean(x,10);
```

Compute the efficiency of the 10% trimmed mean relative to the sample mean for the data.

```
sm = std(m);
strim = std(trim);
efficiency = (sm/strim).^2
```

```
efficiency =
    0.9663
```

### Trimmed Mean for Distributions with Outliers

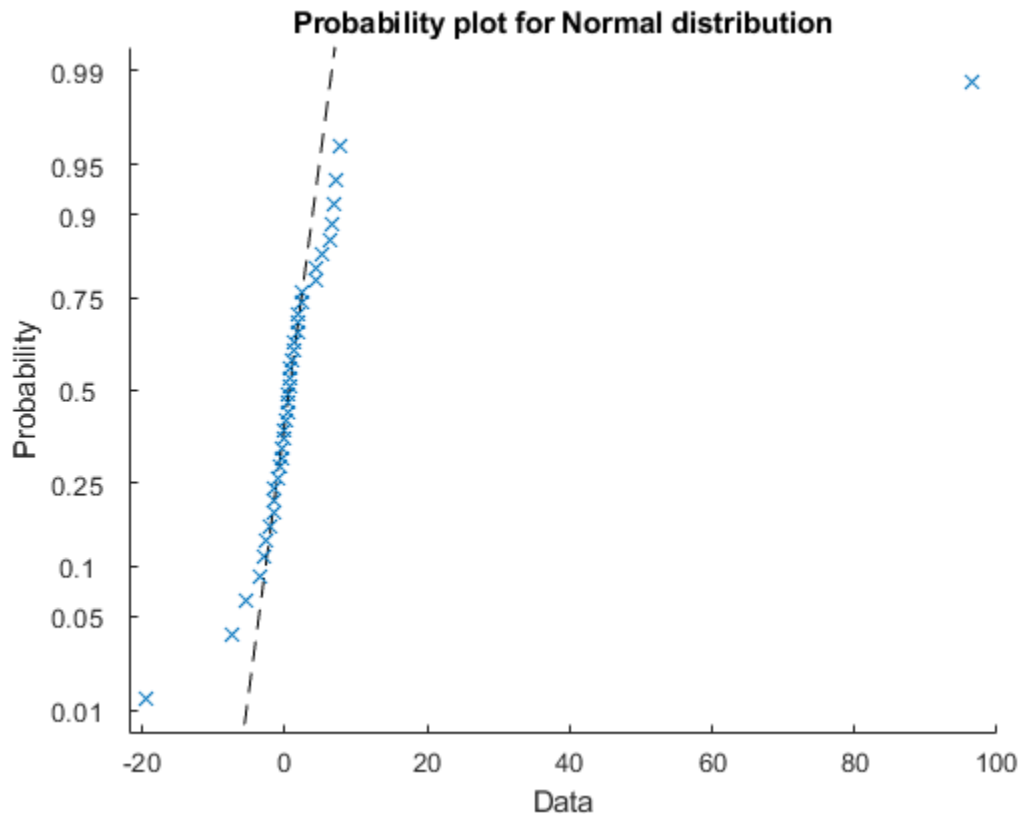
Generate random data from the  $t$  location-scale distribution, which tends to have outliers.

```
rng default; % For reproducibility
x = trnd(1,40,1);
```

Visualize the distribution using a normal probability plot.

```
probplot(x)
```





Although the distribution is symmetric around zero, there are several outliers which will affect the mean. The trimmed mean is closer to zero, which is more representative of the data.

```
mean = mean(x)
tmean = trimmean(x,25)
```

```
mean =
```

```
2.7991
```

```
tmean =
```

0.8797

## Tips

The trimmed mean is a robust estimate of the location of a sample. If there are outliers in the data, the trimmed mean is a more representative estimate of the center of the body of the data than the mean. However, if the data is all from the same probability distribution, then the trimmed mean is less efficient than the sample mean as an estimator of the location of the data.

## See Also

`geomean` | `harmmean` | `mean` | `median`

**Introduced before R2006a**

## trnd

Student's  $t$  random numbers

### Syntax

```
r = trnd(nu)
r = trnd(nu,m,n,...)
r = trnd(nu,[m,n,...])
```

### Description

`r = trnd(nu)` generates random numbers from Student's  $t$  distribution with `nu` degrees of freedom. `nu` can be a vector, a matrix, or a multidimensional array. The size of `r` is equal to the size of `nu`.

`r = trnd(nu,m,n,...)` or `r = trnd(nu,[m,n,...])` generates an `m`-by-`n`-by-... array. The `nu` parameter can be a scalar or an array of the same size as `r`.

### Examples

#### Generate Student's $t$ Distribution Random Numbers

```
r1 = trnd(ones(1,6))
```

```
r1 =
```

```
    0.2108    7.8450  -11.0511    0.4134    4.3293   -0.8323
```

```
r2 = trnd(1:6,[1 6])
```

```
r2 =
```

```
8.9290 -0.1908 0.3496 -0.7658 1.3234 -1.2808
```

```
r3 = trnd(3,2,6)
```

```
r3 =
```

```
1.3157 0.7010 0.1591 -1.3840 4.1354 0.2442  
0.9789 -2.4700 -1.8884 -0.0116 -0.9496 -0.2340
```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

The generated code can return a different sequence of numbers than MATLAB if either of the following is true:

- The output is nonscalar.
- An input parameter is invalid for the distribution.

### See Also

`random` | `tcdf` | `tinv` | `tpdf` | `tstat`

### Topics

“Student's t Distribution” on page B-166

**Introduced before R2006a**

# prob.TruncatableDistribution class

**Package:** prob

Truncatable probability distribution object

## Description

Create a probability distribution object with specified parameter values using `makedist`. Alternatively, fit a distribution to data using `fitdist` or the Distribution Fitter app.

## Methods

`cdf` Cumulative distribution function of probability distribution object  
`icdf` Inverse cumulative distribution function of probability distribution object  
`iqr` Interquartile range of probability distribution object  
`median` Median of probability distribution object  
`pdf` Probability density function of probability distribution object  
`random` Generate random numbers from probability distribution object  
`truncate` Truncate probability distribution object

## See Also

`distributionFitter` | `fitdist` | `makedist`

## Topics

Class Attributes (MATLAB)

Property Attributes (MATLAB)

## truncate

Truncate probability distribution object

### Syntax

```
t = truncate(pd, lower, upper)
```

### Description

`t = truncate(pd, lower, upper)` returns a probability distribution `t`, which is the probability distribution `pd` truncated to the specified interval with lower limit, `lower`, and upper limit, `upper`.

### Examples

#### Truncate a Probability Distribution

Create a standard normal probability distribution object.

```
pd = makedist('Normal')
```

```
pd =
```

```
NormalDistribution
```

```
Normal distribution
```

```
mu = 0
```

```
sigma = 1
```

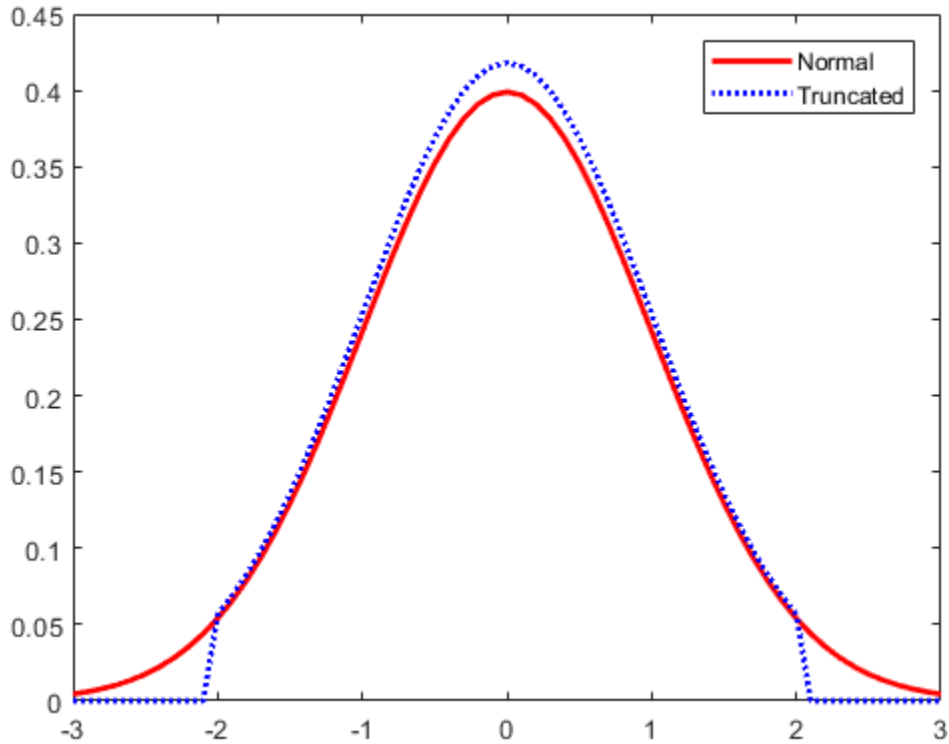
Truncate the distribution to have a lower limit of -2 and an upper limit of 2.

```
t = truncate(pd, -2, 2)
```

```
t =  
  
NormalDistribution  
  
Normal distribution  
    mu = 0  
    sigma = 1  
Truncated to the interval [-2, 2]
```

Plot the pdf of the original and truncated distributions for a visual comparison.

```
x = -3:.1:3;  
figure;  
plot(x,pdf(pd,x), 'Color','red', 'LineWidth',2)  
hold on;  
plot(x,pdf(t,x), 'Color','blue', 'LineWidth',2, 'LineStyle',':')  
legend({'Normal', 'Truncated'}, 'Location', 'NE')  
hold off;
```



### Generate Random Numbers from a Truncated Distribution

Create a standard normal probability distribution object.

```
pd = makedist('Normal')
```

```
pd =
```

```
NormalDistribution
```

```
Normal distribution
```



```
mu = 0
sigma = 1
```

Truncate the distribution by restricting it to positive values. Set the lower limit to 0 and the upper limit to infinity.

```
t = truncate(pd, 0, inf)
```

```
t =
```

```
NormalDistribution
```

```
Normal distribution
```

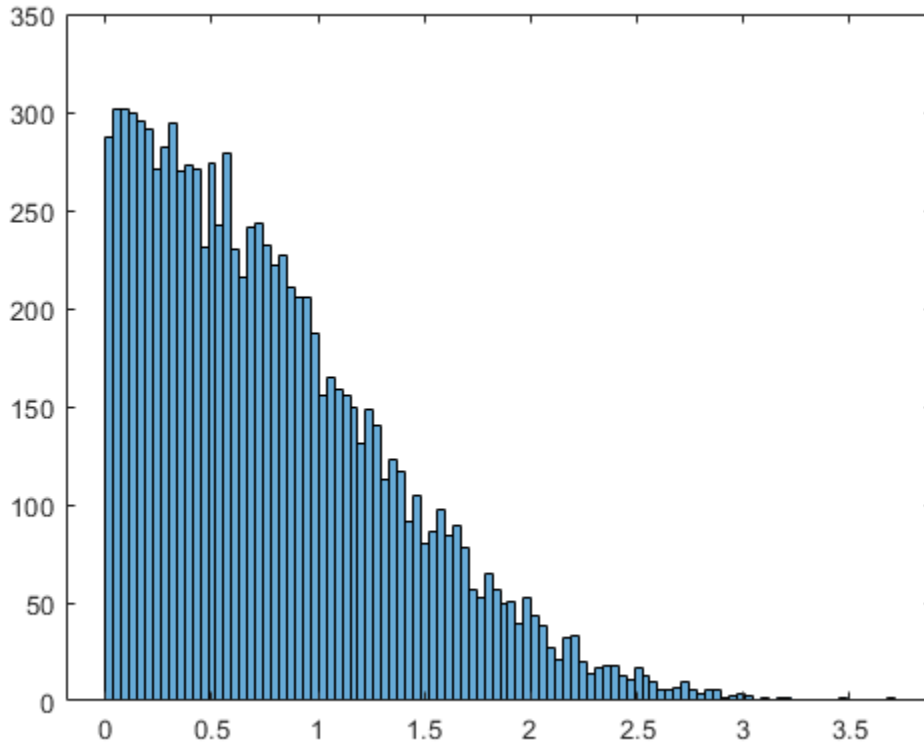
```
mu = 0
```

```
sigma = 1
```

```
Truncated to the interval [0, Inf]
```

Generate random numbers from the truncated distribution and visualize with a histogram.

```
r = random(t, 10000, 1);
histogram(r, 100)
```



## Input Arguments

### **pd** — Probability distribution

probability distribution object

Probability distribution, specified as a probability distribution object. Create a probability distribution object with specified parameter values using `makedist`.

Alternatively, for fittable distributions, create a probability distribution object by fitting it to data using `fitdist` or the Distribution Fitting app.

**lower** — Lower truncation limit

scalar value

Lower truncation limit, specified as a scalar value.

Data Types: `single` | `double`

**upper** — Upper truncation limit

scalar value

Upper truncation limit, specified as a scalar value.

Data Types: `single` | `double`

## Output Arguments

 **$\tau$**  — Truncated distribution

probability distribution object

Truncated distribution, returned as a probability distribution object. The probability distribution function (pdf) of  $\tau$  is 0 outside the truncation interval. Inside the truncation interval, the pdf of  $\tau$  is equal to the pdf of `pd`, but divided by the probability assigned to that interval by `pd`.

## See Also

`distributionFitter` | `fitdist` | `makedist`

## Topics

“Working with Probability Distributions” on page 5-3

Introduced in R2013b

# truncate

**Class:** prob.TruncatableDistribution

**Package:** prob

Truncate probability distribution object

## Syntax

```
t = truncate(pd, lower, upper)
```

## Description

`t = truncate(pd, lower, upper)` returns a probability distribution `t`, which is the probability distribution `pd` truncated to the specified interval with lower limit, `lower`, and upper limit, `upper`.

## Input Arguments

**pd — Probability distribution**

probability distribution object

Probability distribution, specified as a probability distribution object. Create a probability distribution object with specified parameter values using `makedist`. Alternatively, for fittable distributions, create a probability distribution object by fitting it to data using `fitdist` or the Distribution Fitting app.

**lower — Lower truncation limit**

scalar value

Lower truncation limit, specified as a scalar value.

Data Types: `single` | `double`

**upper — Upper truncation limit**

scalar value

Upper truncation limit, specified as a scalar value.

Data Types: `single` | `double`

## Output Arguments

### **$\tau$** — Truncated distribution

probability distribution object

Truncated distribution, returned as a probability distribution object. The pdf of  $\tau$  is 0 outside the truncation interval. Inside the truncation interval, the pdf of  $\tau$  is equal to the pdf of `pd`, but divided by the probability assigned to that interval by `pd`.

## Examples

### Truncate a Probability Distribution

Create a standard normal probability distribution object.

```
pd = makedist('Normal')
```

```
pd =
```

```
NormalDistribution
```

```
Normal distribution
```

```
mu = 0
```

```
sigma = 1
```

Truncate the distribution to have a lower limit of -2 and an upper limit of 2.

```
t = truncate(pd,-2,2)
```

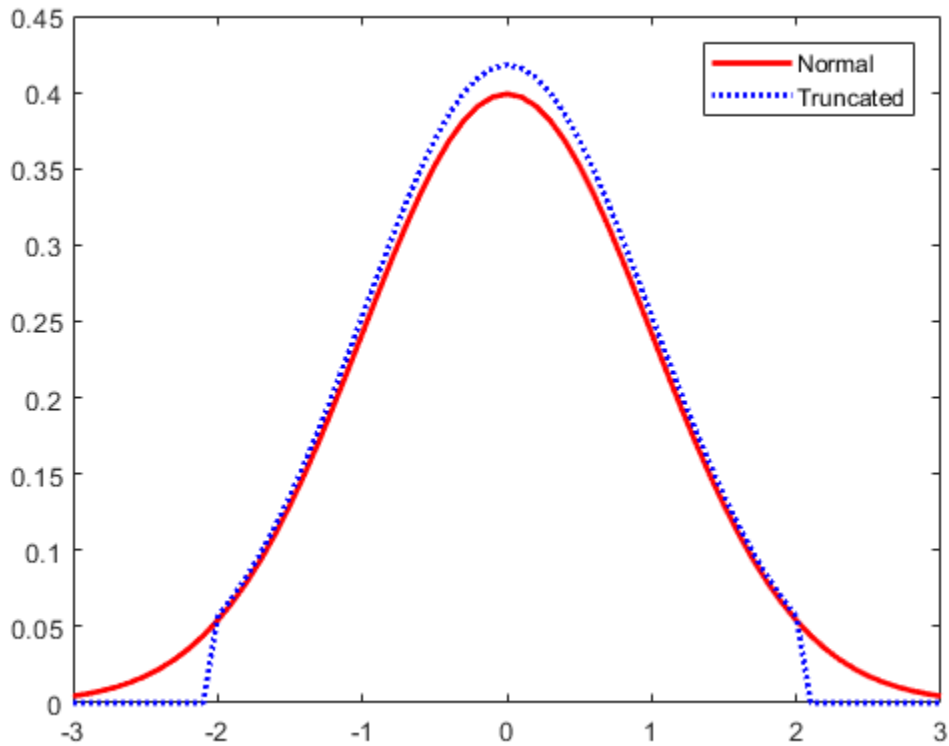
```
t =
```

```
NormalDistribution
```

```
Normal distribution
    mu = 0
    sigma = 1
Truncated to the interval [-2, 2]
```

Plot the pdf of the original and truncated distributions for a visual comparison.

```
x = -3:.1:3;
figure;
plot(x,pdf(pd,x),'Color','red','LineWidth',2)
hold on;
plot(x,pdf(t,x),'Color','blue','LineWidth',2,'LineStyle',':')
legend({'Normal','Truncated'},'Location','NE')
hold off;
```



### Generate Random Numbers from a Truncated Distribution

Create a standard normal probability distribution object.

```
pd = makedist('Normal')
```

```
pd =
```

```
NormalDistribution
```

```
Normal distribution
```

```
mu = 0  
sigma = 1
```

Truncate the distribution by restricting it to positive values. Set the lower limit to 0 and the upper limit to infinity.

```
t = truncate(pd, 0, inf)
```

```
t =
```

```
NormalDistribution
```

```
Normal distribution
```

```
mu = 0
```

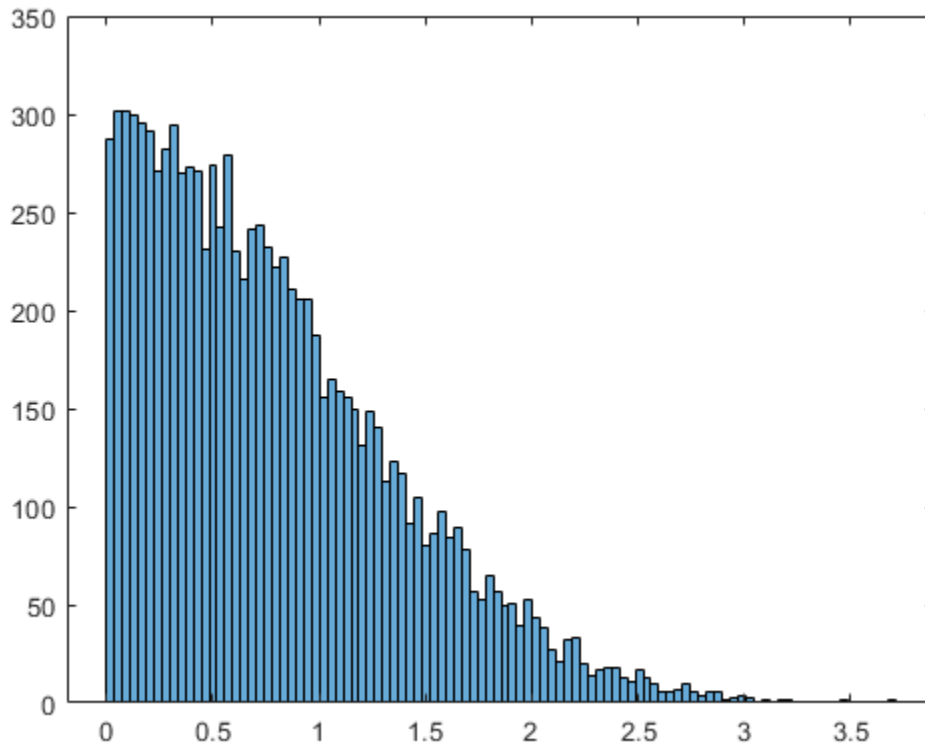
```
sigma = 1
```

```
Truncated to the interval [0, Inf]
```

Generate random numbers from the truncated distribution and visualize with a histogram.

```
r = random(t, 10000, 1);  
histogram(r, 100)
```





## See Also

`distributionFitter` | `fitdist` | `makedist`

## Topics

“Working with Probability Distributions” on page 5-3

“Supported Distributions” on page 5-17

## tsne

t-Distributed Stochastic Neighbor Embedding

### Syntax

```
Y = tsne(X)
Y = tsne(X,Name,Value)
[Y,loss] = tsne(____)
```

### Description

`Y = tsne(X)` returns a matrix of two-dimensional embeddings of the high-dimensional rows of `X`.

`Y = tsne(X,Name,Value)` modifies the embeddings using options specified by one or more name-value pair arguments.

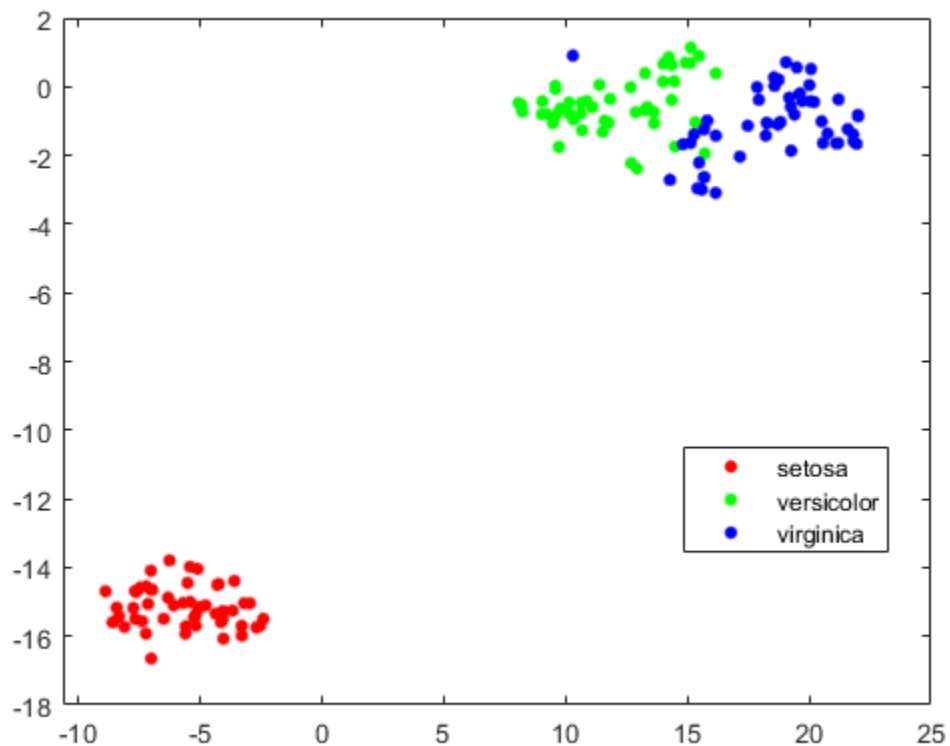
`[Y,loss] = tsne(____)`, for any input arguments, also returns the Kullback-Leibler divergence between the joint distributions that model the data `X` and the embedding `Y`.

### Examples

#### Visualize Fisher Iris Data

The Fisher iris data set has four-dimensional measurements of irises, and corresponding classification into species. Visualize this data by reducing the dimension using `tsne`.

```
load fisheriris
rng default % for reproducibility
Y = tsne(meas);
gscatter(Y(:,1),Y(:,2),species)
```



### Compare Distance Measures

Use various distance measures to try to obtain a better separation between species in the Fisher iris data.

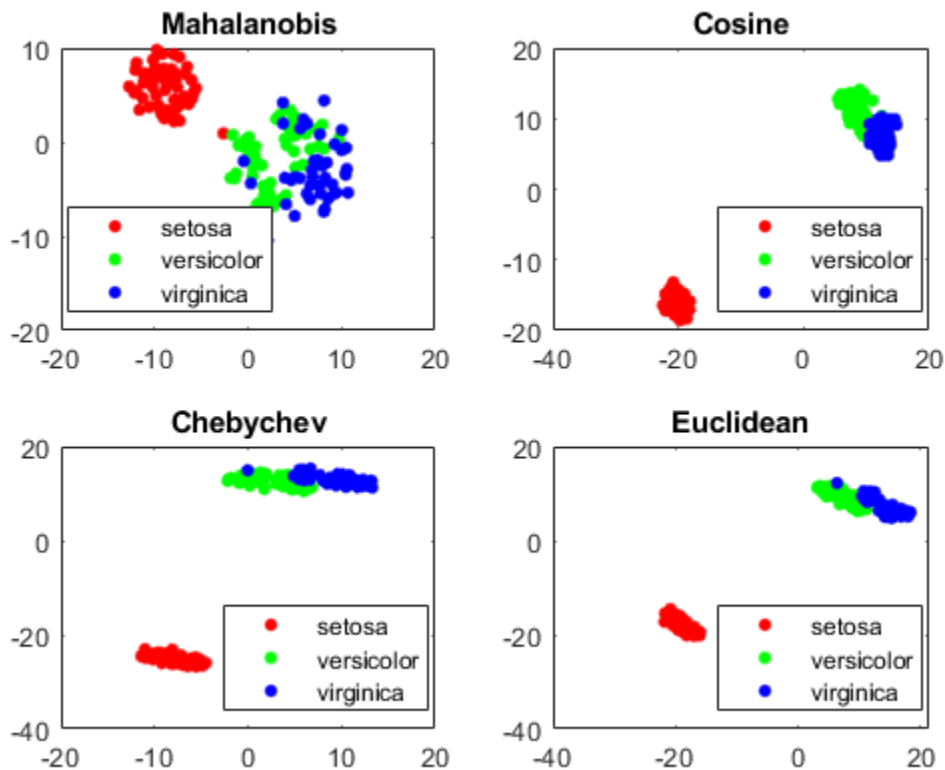
```
load fisheriris

rng default % for reproducibility
Y = tsne(meas, 'Algorithm', 'exact', 'Distance', 'mahalanobis');
subplot(2,2,1)
gscatter(Y(:,1), Y(:,2), species)
title('Mahalanobis')
```

```
rng default % for fair comparison
Y = tsne(meas, 'Algorithm', 'exact', 'Distance', 'cosine');
subplot(2,2,2)
gscatter(Y(:,1),Y(:,2),species)
title('Cosine')

rng default % for fair comparison
Y = tsne(meas, 'Algorithm', 'exact', 'Distance', 'chebychev');
subplot(2,2,3)
gscatter(Y(:,1),Y(:,2),species)
title('Chebychev')

rng default % for fair comparison
Y = tsne(meas, 'Algorithm', 'exact', 'Distance', 'euclidean');
subplot(2,2,4)
gscatter(Y(:,1),Y(:,2),species)
title('Euclidean')
```



In this case, the cosine, Chebychev, and Euclidean distance measures give reasonably good separation of clusters. But the Mahalanobis distance measure does not give a good separation.

### Plot Results with NaN Input Data

`tsne` removes input data rows that contain any NaN entries. Therefore, you must remove any such rows from your classification data before plotting.

For example, change a few random entries in the Fisher iris data to NaN.

```
load fisheriris
rng default % for reproducibility
meas(rand(size(meas)) < 0.05) = NaN;
```

Embed the four-dimensional data into two dimensions using `tsne`.

```
Y = tsne(meas, 'Algorithm', 'exact');
```

Warning: Rows with NaN missing values in X or 'InitialY' values are removed.

Determine how many rows were eliminated from the embedding.

```
length(species) - length(Y)
```

```
ans =
```

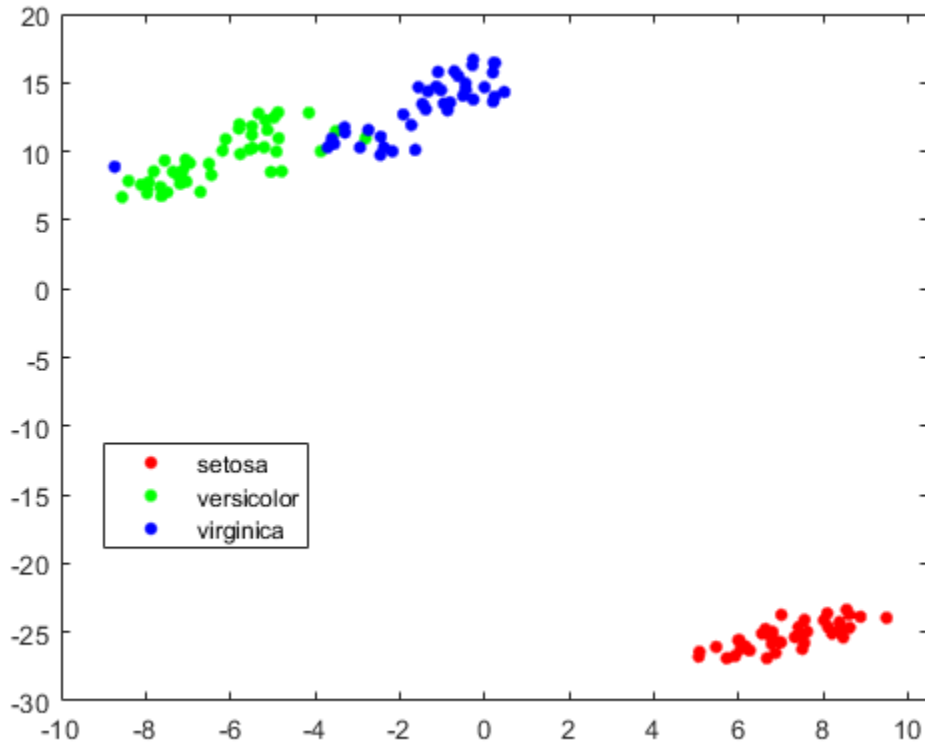
```
22
```

Prepare to plot the result by locating the rows of `meas` that have no NaN values.

```
goodrows = not(any(isnan(meas), 2));
```

Plot the results using only the rows of `species` that correspond to rows of `meas` with no NaN values.

```
gscatter(Y(:,1), Y(:,2), species(goodrows))
```



### Compare t-SNE Loss

Find both 2-D and 3-D embeddings of the Fisher iris data, and compare the loss for each embedding. It is likely that the loss is lower for a 3-D embedding, because this embedding has more freedom to match the original data.

```
load fisheriris
rng default % for reproducibility
[Y,loss] = tsne(meas,'Algorithm','exact');
rng default % for fair comparison
[Y2,loss2] = tsne(meas,'Algorithm','exact','NumDimensions',3);
fprintf('2-D embedding has loss %g, and 3-D embedding has loss %g.\n',loss,loss2)
```

2-D embedding has loss 0.122669, and 3-D embedding has loss 0.0985112.

As expected, the 3-D embedding has lower loss.

View the embeddings. Use RGB colors [1 0 0], [0 1 0], and [0 0 1].

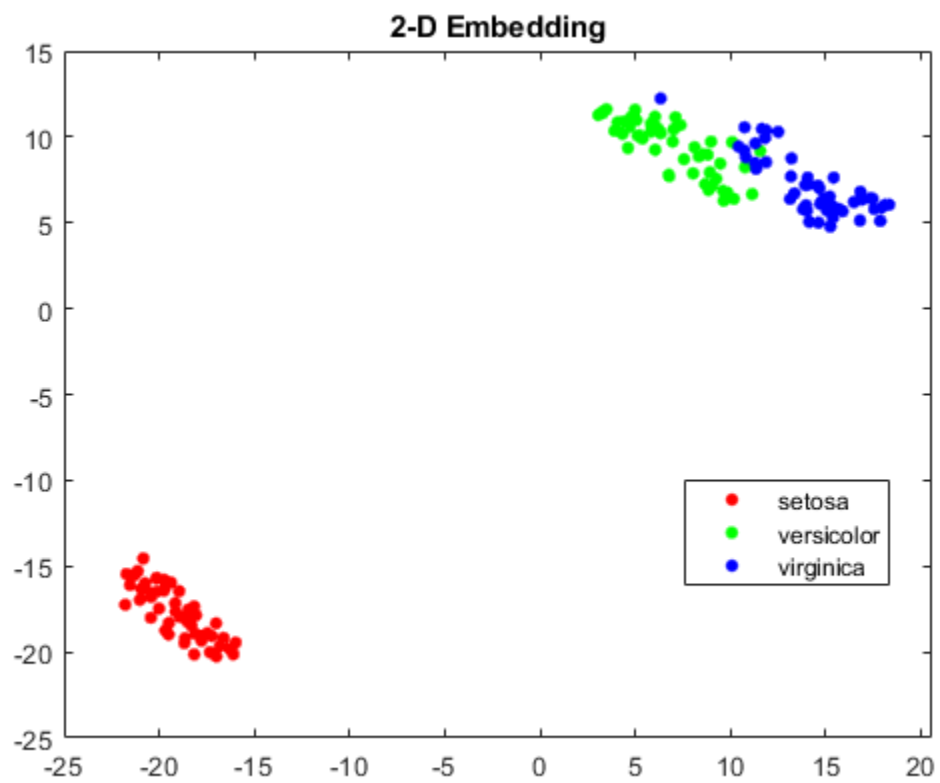
For the 3-D plot, convert the species to numeric values using the `categorical` command, then convert the numeric values to RGB colors using the `sparse` function as follows. If `v` is a vector of positive integers 1, 2, or 3, corresponding to the species data, then the command

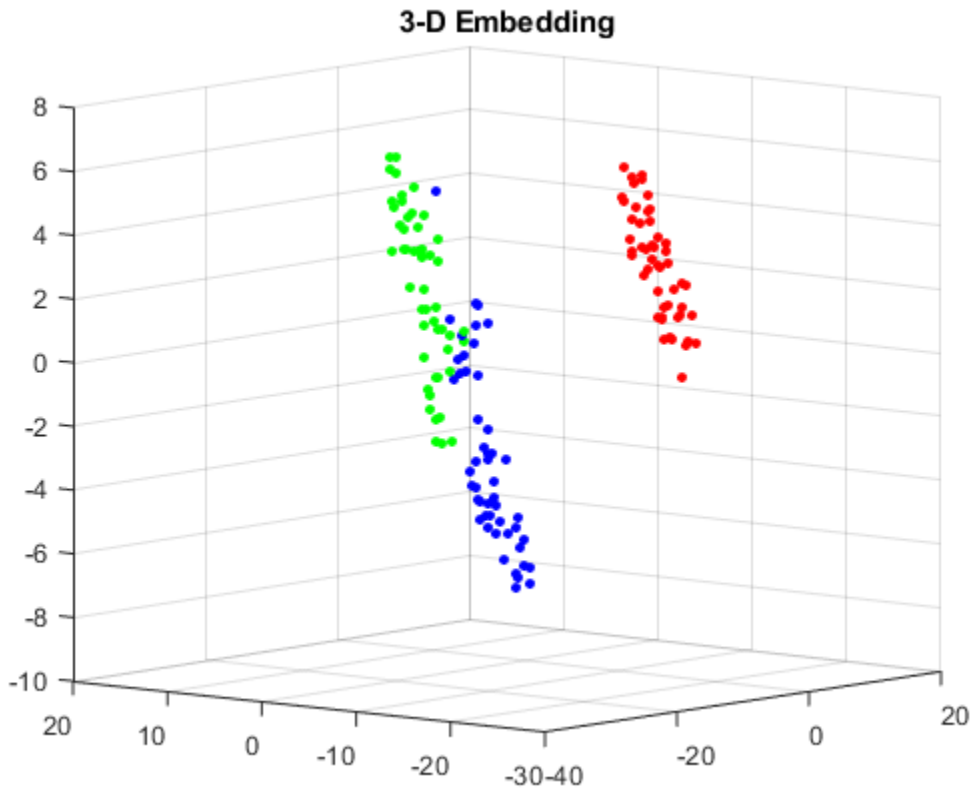
```
sparse(1:numel(v),v,ones(size(v)))
```

is a sparse matrix whose rows are the RGB colors of the species.

```
gscatter(Y(:,1),Y(:,2),species,eye(3))
title('2-D Embedding')
figure
v = double(categorical(species));
c = full(sparse(1:numel(v),v,ones(size(v)),numel(v),3));
scatter3(Y2(:,1),Y2(:,2),Y2(:,3),15,c,'filled')
title('3-D Embedding')
view(-50,8)
```







- “Visualize High-Dimensional Data Using t-SNE” on page 15-144
- “t-SNE Custom Output Function” on page 15-141
- “tsne Settings” on page 15-149

## Input Arguments

### **x** — Data points

*n*-by-*m* matrix

Data points, specified as an *n*-by-*m* matrix, where each row is one *m*-dimensional point.

`tsne` removes rows of `X` that contain any NaN values before creating an embedding. See “Plot Results with NaN Input Data” on page 32-6631.

Data Types: `single` | `double`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `Y = tsne(X, 'Algorithm', 'Exact', 'NumPCAComponents', 50)`

### Algorithm Control

#### Algorithm — `tsne` algorithm

'barneshut' (default) | 'exact'

`tsne` algorithm, specified as 'barneshut' or 'exact'. The 'exact' algorithm optimizes the Kullback-Leibler divergence of distributions between the original space and the embedded space. The 'barneshut' algorithm performs an approximate optimization that is faster and uses less memory when the number of data rows is large.

---

**Note** For the 'barneshut' algorithm, `tsne` uses `knnsearch` to find the nearest neighbors.

---

Example: 'exact'

Data Types: `char`

#### Distance — Distance metric

'euclidean' (default) | 'seuclidean' | 'cityblock' | 'chebychev' | 'minkowski' | 'mahalanobis' | 'cosine' | 'correlation' | 'spearman' | 'hamming' | 'jaccard' | function handle

Distance metric, specified by one of the following. For definitions of the distance metrics, see `pdist`.

- 'euclidean' — Euclidean distance.

- 'seuclidean' — Standardized Euclidean distance. Each coordinate difference between rows in  $X$  and the query matrix is scaled by dividing by the corresponding element of the standard deviation computed from  $S = \text{nanstd}(X)$ .
- 'cityblock' — City block distance.
- 'chebychev' — Chebychev distance, which is the maximum coordinate difference.
- 'minkowski' — Minkowski distance with exponent 2. This is the same as Euclidean distance.
- 'mahalanobis' — Mahalanobis distance, computed using the positive definite covariance matrix  $\text{nancov}(X)$ .
- 'cosine' — 1 minus the cosine of the included angle between observations (treated as vectors).
- 'correlation' — One minus the sample linear correlation between observations (treated as sequences of values).
- 'spearman' — One minus the sample Spearman's rank correlation between observations (treated as sequences of values).
- 'hamming' — Hamming distance, which is the percentage of coordinates that differ.
- 'jaccard' — One minus the Jaccard coefficient, which is the percentage of nonzero coordinates that differ.
- custom distance function — A distance function specified using @ (for example, @distfun). For details, see “Definitions” on page 32-6642.

In all cases, `tsne` uses squared pairwise distances to calculate the Gaussian kernel in the joint distribution of  $X$ .

Example: 'mahalanobis'

Data Types: char | function\_handle

### **Exaggeration — Size of natural clusters in data**

4 (default) | scalar value 1 or greater

Size of natural clusters in data, specified as a scalar value 1 or greater.

A large exaggeration makes `tsne` learn larger joint probabilities of  $Y$  and creates relatively more space between clusters in  $Y$ . `tsne` uses exaggeration in the first 99 optimization iterations.

If the value of Kullback-Leibler divergence increases in the early stage of the optimization, try reducing the exaggeration. See “tsne Settings” on page 15-149.

Example: 10

Data Types: `single` | `double`

### **NumDimensions** — Dimension of the output $Y$

2 (default) | positive integer

Dimension of the output  $Y$ , specified as a positive integer. Generally, set `NumDimensions` to 2 or 3.

Example: 3

Data Types: `single` | `double`

### **NumPCAComponents** — PCA dimension reduction

0 (default) | nonnegative integer

PCA dimension reduction, specified as a nonnegative integer. Before `tsne` embeds the high-dimensional data, it first reduces the dimensionality of the data to `NumPCAComponents` using the `pca` function. When `NumPCAComponents` is 0, `tsne` does not use PCA.

Example: 50

Data Types: `single` | `double`

### **Perplexity** — Effective number of local neighbors of each point

30 (default) | positive scalar

Effective number of local neighbors of each point, specified as a positive scalar. See “t-SNE Algorithm” on page 15-133.

Larger perplexity causes `tsne` to use more points as nearest neighbors. Use a larger value of `Perplexity` for a large dataset. Typical `Perplexity` values are from 5 to 50. In the Barnes-Hut algorithm, `tsne` uses  $\min(3 * \text{Perplexity}, N - 1)$  as the number of nearest neighbors. See “tsne Settings” on page 15-149.

Example: 10

Data Types: `single` | `double`

**Standardize — Normalize input data**`false (default) | true`

Normalize input data, specified as `false` or `true`. When `true`, `tsne` centers and scales `X` by dividing the columns by their standard deviations.

When features in `X` are on different scales, set 'Standardize' to `true`. Do this because the learning process is based on nearest neighbors, so features with large scales can override the contribution of features with small scales.

Example: `true`

Data Types: `logical`

**Optimization Control****InitialY — Initial embedded points**`1e-4*randn(N, NumDimensions) (default) | n-by-NumDimensions real matrix`

Initial embedded points, specified as an `n-by-NumDimensions` real matrix, where `n` is the number of rows of `X`. The `tsne` optimization algorithm uses these points as initial values.

Data Types: `single` | `double`

**LearnRate — Learning rate for optimization process**`500 (default) | positive scalar`

Learning rate for optimization process, specified as a positive scalar. Typically, set values from 100 through 1000.

When `LearnRate` is too small, `tsne` can converge to a poor local minimum. When `LearnRate` is too large, the optimization can initially have the Kullback-Leibler divergence increase rather than decrease. See “`tsne` Settings” on page 15-149.

Example: `1000`

Data Types: `single` | `double`

**NumPrint — Iterative display frequency**`20 (default) | positive integer`

Iterative display frequency, specified as a positive integer. When the `Verbose` name-value pair is not 0, `tsne` returns iterative display after every `NumPrint` iterations. If the

Options name-value pair contains a nonempty 'OutputFcn' entry, then output functions run after every NumPrint iterations.

Example: 20

Data Types: single | double

### Options — Optimization options

Structure containing the fields 'MaxIter', 'OutputFcn', and 'TolFun'

Optimization options, specified as a structure containing the fields 'MaxIter', 'OutputFcn', and 'TolFun'. Create 'Options' using `statset` or `struct`.

- 'MaxIter' — Positive integer specifying the maximum number of optimization iterations. Default: 1000.
- 'OutputFcn' — Function handle or cell array of function handles specifying one or more functions to call after every NumPrint optimization iterations. For syntax details, see “t-SNE Output Function” on page 15-140. Default: [].
- 'TolFun' — Stopping criterion for the optimization. The optimization exits when the norm of the gradient of the Kullback-Leibler divergence is less than 'TolFun'. Default: 1e-10.

Example: `options = statset('MaxIter',500)`

Data Types: struct

### Theta — Barnes-Hut tradeoff parameter

0.5 (default) | scalar from 0 through 1

Barnes-Hut tradeoff parameter, specified as a scalar from 0 through 1. Higher values give a faster but less accurate optimization. Applies only when Algorithm is 'barneshut'.

Example: 0.1

Data Types: single | double

### Verbose — Iterative display

0 (default) | 1 | 2

Iterative display, specified as 0, 1, or 2. When `Verbose` is not 0, `tsne` prints a summary table of the Kullback-Leibler divergence and the norm of its gradient every `NumPrint` iterations.

When `Verbose` is 2, `tsne` also prints the variances of Gaussian kernels. `tsne` uses these kernels in its computation of the joint probability of  $x$ . If you see a large difference in the scales of the minimum and maximum variances, you can sometimes get more suitable results by rescaling  $x$ .

Example: 2

Data Types: `single` | `double`

## Output Arguments

### **`y`** — Embedded points

`n`-by-`NumDimensions` matrix

Embedded points, returned as an `n`-by-`NumDimensions` matrix. Each row represents one embedded point. `n` is the number of rows of data  $X$  that do not contain any NaN entries. See “Plot Results with NaN Input Data” on page 32-6631.

### **`loss`** — Kullback-Leibler divergence

nonnegative scalar

Kullback-Leibler divergence between modeled input and output distributions, returned as a nonnegative scalar. For details, see “t-SNE Algorithm” on page 15-133.

## Definitions

### Custom Distance Function

The syntax of a custom distance function is as follows.

```
function D2 = distfun(ZI,ZJ)
```

`tsne` passes `ZI` and `ZJ` to your function, and your function computes the distance.



- $Z_I$  is a 1-by- $n$  vector containing a single row from  $X$  or  $Y$ .
- $Z_J$  is an  $m$ -by- $n$  matrix containing multiple rows of  $X$  or  $Y$ .

Your function returns  $D_2$ , which is an  $m$ -by-1 vector of distances. The  $j$ th element of  $D_2$  is the distance between the observations  $Z_I$  and  $Z_J(j, :)$ .

---

**Tip** If your data are not sparse, then usually the built-in distance functions are faster than a function handle.

---

## Algorithms

`tsne` constructs a set of embedded points in a low-dimensional space whose relative similarities mimic those of the original high-dimensional points. The embedded points show the clustering in the original data.

Roughly, the algorithm models the original points as coming from a Gaussian distribution, and the embedded points as coming from a Student's  $t$  distribution. The algorithm tries to minimize the Kullback-Leibler divergence between these two distributions by moving the embedded points.

For details, see “t-SNE” on page 15-132.

## See Also

`gscatter` | `knnsearch` | `pca` | `pdist` | `statset`

## Topics

“Visualize High-Dimensional Data Using t-SNE” on page 15-144

“t-SNE Custom Output Function” on page 15-141

“tsne Settings” on page 15-149

“t-SNE” on page 15-132

“Dimensionality Reduction and Feature Extraction”

**Introduced in R2017a**

## tstat

Student's  $t$  mean and variance

## Syntax

```
[m,v] = tstat(nu)
```

## Description

`[m,v] = tstat(nu)` returns the mean of and variance for Student's  $t$  distribution using the degrees of freedom in `nu`. `nu` can a vectors, matrix, or multidimensional array. The returned values for `m` and `v` are the same size as `nu`.

## Examples

### Compute Student's $t$ Mean and Variance

Compute the mean and variance for Student's  $t$  distributions with degrees of freedom `nu` equal to 1 through 30.

```
nu = reshape(1:30,6,5);  
[m,v] = tstat(nu)
```

`m =`

```
NaN    0    0    0    0  
    0    0    0    0    0  
    0    0    0    0    0  
    0    0    0    0    0  
    0    0    0    0    0  
    0    0    0    0    0
```

`v =`

NaN	1.4000	1.1818	1.1176	1.0870
NaN	1.3333	1.1667	1.1111	1.0833
3.0000	1.2857	1.1538	1.1053	1.0800
2.0000	1.2500	1.1429	1.1000	1.0769
1.6667	1.2222	1.1333	1.0952	1.0741
1.5000	1.2000	1.1250	1.0909	1.0714

Note that the variance does not exist for one and two degrees of freedom.

## Definitions

### Student's $t$ Mean and Variance

The mean of the Student's  $t$  distribution is  
mean = 0

for degrees of freedom  $\nu$  greater than 1. If  $\nu$  equals 1, then the mean is undefined.

The variance of the Student's  $t$  distribution is

$$\text{var} = \frac{\nu}{\nu - 2}$$

for degrees of freedom  $\nu$  greater than 2. If  $\nu$  is less than or equal to 2, then the variance is undefined.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`tcdf` | `tinvs` | `tpdf` | `trnd`

## **Topics**

“Student's t Distribution” on page B-166

**Introduced before R2006a**

# ttest

One-sample and paired-sample *t*-test

## Syntax

```
h = ttest(x)
```

```
h = ttest(x,y)
```

```
h = ttest(x,y,Name,Value)
```

```
h = ttest(x,m)
```

```
h = ttest(x,m,Name,Value)
```

```
[h,p] = ttest(____)
```

```
[h,p,ci,stats] = ttest(____)
```

## Description

`h = ttest(x)` returns a test decision for the null hypothesis that the data in `x` comes from a normal distribution with mean equal to zero and unknown variance, using the one-sample *t*-test on page 32-6655. The alternative hypothesis is that the population distribution does not have a mean equal to zero. The result `h` is 1 if the test rejects the null hypothesis at the 5% significance level, and 0 otherwise.

`h = ttest(x,y)` returns a test decision for the null hypothesis that the data in `x - y` comes from a normal distribution with mean equal to zero and unknown variance, using the paired-sample *t*-test.

`h = ttest(x,y,Name,Value)` returns a test decision for the paired-sample *t*-test with additional options specified by one or more name-value pair arguments. For example, you can change the significance level or conduct a one-sided test.

`h = ttest(x,m)` returns a test decision for the null hypothesis that the data in `x` comes from a normal distribution with mean `m` and unknown variance. The alternative hypothesis is that the mean is not `m`.

`h = ttest(x,m,Name,Value)` returns a test decision for the one-sample *t*-test with additional options specified by one or more name-value pair arguments. For example, you can change the significance level or conduct a one-sided test.

`[h,p] = ttest(____)` also returns the *p*-value, *p*, of the test, using any of the input arguments from the previous syntax groups.

`[h,p,ci,stats] = ttest(____)` also returns the confidence interval *ci* for the mean of *x*, or of *x* - *y* for the paired *t*-test, and the structure *stats* containing information about the test statistic.

## Examples

### t-Test for Mean Equal to Zero

Load the sample data. Create a vector containing the third column of the stock returns data.

```
load stockreturns
x = stocks(:,3);
```

Test the null hypothesis that the sample data comes from a population with mean equal to zero.

```
[h,p,ci,stats] = ttest(x)
```

```
h =
```

```
1
```

```
p =
```

```
0.0106
```

```
ci =
```

```
-0.7357
```

```
-0.0997
```

```
stats =  
  
    struct with fields:  
  
        tstat: -2.6065  
         df: 99  
         sd: 1.6027
```

The returned value  $h = 1$  indicates that `ttest` rejects the null hypothesis at the 5% significance level.

### t-Test at Different Significance Level

Load the sample data. Create a vector containing the third column of the stock returns data.

```
load stockreturns  
x = stocks(:,3);
```

Test the null hypothesis that the sample data are from a population with mean equal to zero at the 1% significance level.

```
h = ttest(x,0,'Alpha',0.01)
```

```
h =  
  
    0
```

The returned value  $h = 0$  indicates that `ttest` does not reject the null hypothesis at the 1% significance level.

### Paired-Sample t-Test

Load the sample data. Create vectors containing the first and second columns of the data matrix to represent students' grades on two exams.

```
load examgrades
x = grades(:,1);
y = grades(:,2);
```

Test the null hypothesis that the pairwise difference between data vectors  $x$  and  $y$  has a mean equal to zero.

```
[h,p] = ttest(x,y)
```

```
h =
    0
```

```
p =
    0.9805
```

The returned value of  $h = 0$  indicates that `ttest` does not reject the null hypothesis at the default 5% significance level.

### Paired-Sample t-Test at Different Significance Level

Load the sample data. Create vectors containing the first and second columns of the data matrix to represent students' grades on two exams.

```
load examgrades
x = grades(:,1);
y = grades(:,2);
```

Test the null hypothesis that the pairwise difference between data vectors  $x$  and  $y$  has a mean equal to zero at the 1% significance level.

```
[h,p] = ttest(x,y,'Alpha',0.01)
```

```
h =
    0
```



```
p =  
    0.9805
```

The returned value of  $h = 0$  indicates that `ttest` does not reject the null hypothesis at the 1% significance level.

### t-Test for a Hypothesized Mean

Load the sample data. Create a vector containing the first column of the students' exam grades data.

```
load examgrades  
x = grades(:,1);
```

Test the null hypothesis that sample data comes from a distribution with mean  $m = 75$ .

```
h = ttest(x,75)
```

```
h =  
    0
```

The returned value of  $h = 0$  indicates that `ttest` does not reject the null hypothesis at the 5% significance level.

### One-Sided t-Test

Load the sample data. Create a vector containing the first column of the students' exam grades data.

```
load examgrades  
x = grades(:,1);
```

Test the null hypothesis that the data comes from a population with mean equal to 65, against the alternative that the mean is greater than 65.

```
h = ttest(x,65,'Tail','right')
```

```
h =  
  
    1
```

The returned value of `h = 1` indicates that `ttest` rejects the null hypothesis at the 5% significance level, in favor of the alternate hypothesis that the data comes from a population with a mean greater than 65.

## Input Arguments

### **x** — Sample data

vector | matrix | multidimensional array

Sample data, specified as a vector, matrix, or multidimensional array on page 32-6655. `ttest` performs a separate *t*-test along each column and returns a vector of results. If `y` sample data is specified, `x` and `y` must be the same size.

Data Types: `single` | `double`

### **y** — Sample data

vector | matrix | multidimensional array

Sample data, specified as a vector, matrix, or multidimensional array on page 32-6655. If `y` sample data is specified, `x` and `y` must be the same size.

Data Types: `single` | `double`

### **m** — Hypothesized population mean

0 (default) | scalar value

Hypothesized population mean, specified as a scalar value.

Data Types: `single` | `double`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single

quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'Tail', 'right', 'Alpha', 0.01` conducts a right-tailed hypothesis test at the 1% significance level.

### **Alpha — Significance level**

`0.05` (default) | scalar value in the range (0,1)

Significance level of the hypothesis test, specified as the comma-separated pair consisting of `'Alpha'` and a scalar value in the range (0,1).

Example: `'Alpha', 0.01`

Data Types: `single` | `double`

### **Dim — Dimension**

first nonsingleton dimension (default) | positive integer value

Dimension of the input matrix along which to test the means, specified as the comma-separated pair consisting of `'Dim'` and a positive integer value. For example, specifying `'Dim', 1` tests the column means, while `'Dim', 2` tests the row means.

Example: `'Dim', 2`

Data Types: `single` | `double`

### **Tail — Type of alternative hypothesis**

`'both'` (default) | `'right'` | `'left'`

Type of alternative hypothesis to evaluate, specified as the comma-separated pair consisting of `'Tail'` and one of the following.

<code>'both'</code>	Test the alternative hypothesis that the population mean is not $m$ .
<code>'right'</code>	Test the alternative hypothesis that the population mean is greater than $m$ .
<code>'left'</code>	Test the alternative hypothesis that the population mean is less than $m$ .

Example: `'Tail', 'right'`

## Output Arguments

### **h** — Hypothesis test result

1 | 0

Hypothesis test result, returned as a logical value.

- If  $h = 1$ , this indicates the rejection of the null hypothesis at the Alpha significance level.
- If  $h = 0$ , this indicates a failure to reject the null hypothesis at the Alpha significance level.

### **p** — *p*-value

scalar value in the range [0,1]

*p*-value of the test, returned as a scalar value in the range [0,1]. *p* is the probability of observing a test statistic as extreme as, or more extreme than, the observed value under the null hypothesis. Small values of *p* cast doubt on the validity of the null hypothesis.

### **ci** — Confidence interval

vector

Confidence interval for the true population mean, returned as a two-element vector containing the lower and upper boundaries of the  $100 \times (1 - \text{Alpha})\%$  confidence interval.

### **stats** — Test statistics

structure

Test statistics, returned as a structure containing the following:

- `tstat` — Value of the test statistic.
- `df` — Degrees of freedom of the test.
- `sd` — Estimated population standard deviation. For a paired *t*-test, *sd* is the standard deviation of  $x - y$ .

## Definitions

### One-Sample t-Test

The one-sample  $t$ -test is a parametric test of the location parameter when the population standard deviation is unknown.

The test statistic is

$$t = \frac{\bar{x} - \mu}{s / \sqrt{n}},$$

where  $\bar{x}$  is the sample mean,  $\mu$  is the hypothesized population mean,  $s$  is the sample standard deviation, and  $n$  is the sample size. Under the null hypothesis, the test statistic has Student's  $t$  distribution with  $n - 1$  degrees of freedom.

### Multidimensional Array

A multidimensional array has more than two dimensions. For example, if  $x$  is a 1-by-3-by-4 array, then  $x$  is a three-dimensional array.

### First Nonsingleton Dimension

The first nonsingleton dimension is the first dimension of an array whose size is not equal to 1. For example, if  $x$  is a 1-by-2-by-3-by-4 array, then the second dimension is the first nonsingleton dimension of  $x$ .

## Tips

- Use `sampsizepwr` to calculate:
  - The sample size that corresponds to specified power and parameter values;
  - The power achieved for a particular sample size, given the true parameter value;
  - The parameter value detectable with the specified sample size and power.

## See Also

`sampsizepwr` | `ttest2` | `ztest`

**Introduced before R2006a**

## ttest2

Two-sample *t*-test

### Syntax

```
h = ttest2(x,y)
h = ttest2(x,y,Name,Value)
[h,p] = ttest2(____)
[h,p,ci,stats] = ttest2(____)
```

### Description

`h = ttest2(x,y)` returns a test decision for the null hypothesis that the data in vectors `x` and `y` comes from independent random samples from normal distributions with equal means and equal but unknown variances, using the two-sample *t*-test on page 32-6663. The alternative hypothesis is that the data in `x` and `y` comes from populations with unequal means. The result `h` is 1 if the test rejects the null hypothesis at the 5% significance level, and 0 otherwise.

`h = ttest2(x,y,Name,Value)` returns a test decision for the two-sample *t*-test with additional options specified by one or more name-value pair arguments. For example, you can change the significance level or conduct the test without assuming equal variances.

`[h,p] = ttest2(____)` also returns the *p*-value, `p`, of the test, using any of the input arguments in the previous syntaxes.

`[h,p,ci,stats] = ttest2(____)` also returns the confidence interval on the difference of the population means, `ci`, and the structure `stats` containing information about the test statistic.

### Examples

### Two-Sample t-Test for Equal Means

Load the data set. Create vectors containing the first and second columns of the data matrix to represent students' grades on two exams.

```
load examgrades
x = grades(:,1);
y = grades(:,2);
```

Test the null hypothesis that the two data samples are from populations with equal means.

```
[h,p,ci,stats] = ttest2(x,y)
```

```
h =
```

```
    0
```

```
p =
```

```
    0.9867
```

```
ci =
```

```
   -1.9438
```

```
    1.9771
```

```
stats =
```

```
struct with fields:
```

```
    tstat: 0.0167
```

```
       df: 238
```

```
       sd: 7.7084
```

The returned value of  $h = 0$  indicates that `ttest2` does not reject the null hypothesis at the default 5% significance level.



## t-Test for Equal Means Without Assuming Equal Variances

Load the data set. Create vectors containing the first and second columns of the data matrix to represent students' grades on two exams.

```
load examgrades
x = grades(:,1);
y = grades(:,2);
```

Test the null hypothesis that the two data vectors are from populations with equal means, without assuming that the populations also have equal variances.

```
[h,p] = ttest2(x,y,'Vartype','unequal')
```

```
h =
    0

p =
    0.9867
```

The returned value of  $h = 0$  indicates that `ttest2` does not reject the null hypothesis at the default 5% significance level even if equal variances are not assumed.

## Input Arguments

### **x** — Sample data

vector | matrix | multidimensional array

Sample data, specified as a vector, matrix, or multidimensional array. `ttest2` treats NaN values as missing data and ignores them.

- If  $x$  and  $y$  are specified as vectors, they do not need to be the same length.
- If  $x$  and  $y$  are specified as matrices, they must have the same number of columns. `ttest2` performs a separate  $t$ -test along each column and returns a vector of results.
- If  $x$  and  $y$  are specified as multidimensional arrays on page 32-6663, they must have the same size along all but the first nonsingleton dimension on page 32-6663.

Data Types: `single` | `double`

### **y** — Sample data

vector | matrix | multidimensional array

Sample data, specified as a vector, matrix, or multidimensional array. `ttest2` treats NaN values as missing data and ignores them.

- If `x` and `y` are specified as vectors, they do not need to be the same length.
- If `x` and `y` are specified as matrices, they must have the same number of columns. `ttest2` performs a separate *t*-test along each column and returns a vector of results.
- If `x` and `y` are specified as multidimensional arrays on page 32-6663, they must have the same size along all but the first nonsingleton dimension on page 32-6663. `ttest2` works along the first nonsingleton dimension.

Data Types: `single` | `double`

## **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

Example: `'Tail', 'right', 'Alpha', 0.01, 'Vartype', 'unequal'` specifies a right-tailed test at the 1% significance level, and does not assume that `x` and `y` have equal population variances.

### **Alpha** — Significance level

0.05 (default) | scalar value in the range (0,1)

Significance level of the hypothesis test, specified as the comma-separated pair consisting of `'Alpha'` and a scalar value in the range (0,1).

Example: `'Alpha', 0.01`

Data Types: `single` | `double`

### **Dim** — Dimension

first nonsingleton dimension (default) | positive integer value

Dimension of the input matrix along which to test the means, specified as the comma-separated pair consisting of 'Dim' and a positive integer value. For example, specifying 'Dim', 1 tests the column means, while 'Dim', 2 tests the row means.

Example: 'Dim', 2

Data Types: single | double

### **Tail** — Type of alternative hypothesis

'both' (default) | 'right' | 'left'

Type of alternative hypothesis to evaluate, specified as the comma-separated pair consisting of 'Tail' and one of the following.

'both'	Test the alternative hypothesis that the population means are not equal.
'right'	Test the alternative hypothesis that the population mean of $x$ is greater than the population mean of $y$ .
'left'	Test the alternative hypothesis that the population mean of $x$ is less than the population mean of $y$ .

Example: 'Tail', 'right'

### **vartype** — Variance type

'equal' (default) | 'unequal'

Variance type, specified as the comma-separated pair consisting of 'Vartype' and one of the following.

'equal'	Conduct test using the assumption that $x$ and $y$ are from normal distributions with unknown but equal variances.
'unequal'	Conduct test using the assumption that $x$ and $y$ are from normal distributions with unknown and unequal variances. This is called the Behrens-Fisher problem. <code>ttest2</code> uses Satterthwaite's approximation for the effective degrees of freedom.

Vartype must be a single variance type, even when  $x$  is a matrix or a multidimensional array.

Example: 'Vartype', 'unequal'

## Output Arguments

### **h** — Hypothesis test result

1 | 0

Hypothesis test result, returned as a logical value.

- If  $h = 1$ , this indicates the rejection of the null hypothesis at the `Alpha` significance level.
- If  $h = 0$ , this indicates a failure to reject the null hypothesis at the `Alpha` significance level.

### **p** — *p*-value

scalar value in the range [0,1]

*p*-value of the test, returned as a scalar value in the range [0,1]. *p* is the probability of observing a test statistic as extreme as, or more extreme than, the observed value under the null hypothesis. Small values of *p* cast doubt on the validity of the null hypothesis.

### **ci** — Confidence interval

vector

Confidence interval for the difference in population means of *x* and *y*, returned as a two-element vector containing the lower and upper boundaries of the  $100 \times (1 - \text{Alpha})\%$  confidence interval.

### **stats** — Test statistics

structure

Test statistics for the two-sample *t*-test, returned as a structure containing the following:

- `tstat` — Value of the test statistic.
- `df` — Degrees of freedom of the test.
- `sd` — Pooled estimate of the population standard deviation (for the equal variance case) or a vector containing the unpooled estimates of the population standard deviations (for the unequal variance case).

## Definitions

### Two-Sample *t*-test

The two-sample *t*-test is a parametric test that compares the location parameter of two independent data samples.

The test statistic is

$$t = \frac{\bar{x} - \bar{y}}{\sqrt{\frac{s_x^2}{n} + \frac{s_y^2}{m}}},$$

where  $\bar{x}$  and  $\bar{y}$  are the sample means,  $s_x$  and  $s_y$  are the sample standard deviations, and  $n$  and  $m$  are the sample sizes.

In the case where it is assumed that the two data samples are from populations with equal variances, the test statistic under the null hypothesis has Student's *t* distribution with  $n + m - 2$  degrees of freedom, and the sample standard deviations are replaced by the pooled standard deviation

$$s = \sqrt{\frac{(n-1)s_x^2 + (m-1)s_y^2}{n+m-2}}.$$

In the case where it is not assumed that the two data samples are from populations with equal variances, the test statistic under the null hypothesis has an approximate Student's *t* distribution with a number of degrees of freedom given by Satterthwaite's approximation. This test is sometimes called Welch's *t*-test.

### Multidimensional Array

A multidimensional array has more than two dimensions. For example, if  $x$  is a 1-by-3-by-4 array, then  $x$  is a three-dimensional array.

### First Nonsingleton Dimension

The first nonsingleton dimension is the first dimension of an array whose size is not equal to 1. For example, if  $x$  is a 1-by-2-by-3-by-4 array, then the second dimension is the first nonsingleton dimension of  $x$ .

## Tips

- Use `sampsizepwr` to calculate:
  - The sample size that corresponds to specified power and parameter values;
  - The power achieved for a particular sample size, given the true parameter value;
  - The parameter value detectable with the specified sample size and power.

## See Also

`sampsizepwr` | `ttest` | `ztest`

Introduced before R2006a

# type

**Class:** classregtree

Tree type

---

**Note** classregtree will be removed in a future release. See fitctree, fitrtree, ClassificationTree, or RegressionTree instead.

---

## Syntax

```
ttype = type(t)
```

## Description

ttype = type(t) returns the type of the tree t. ttype is 'regression' for regression trees and 'classification' for classification trees.

## Examples

Create a classification tree for Fisher's iris data:

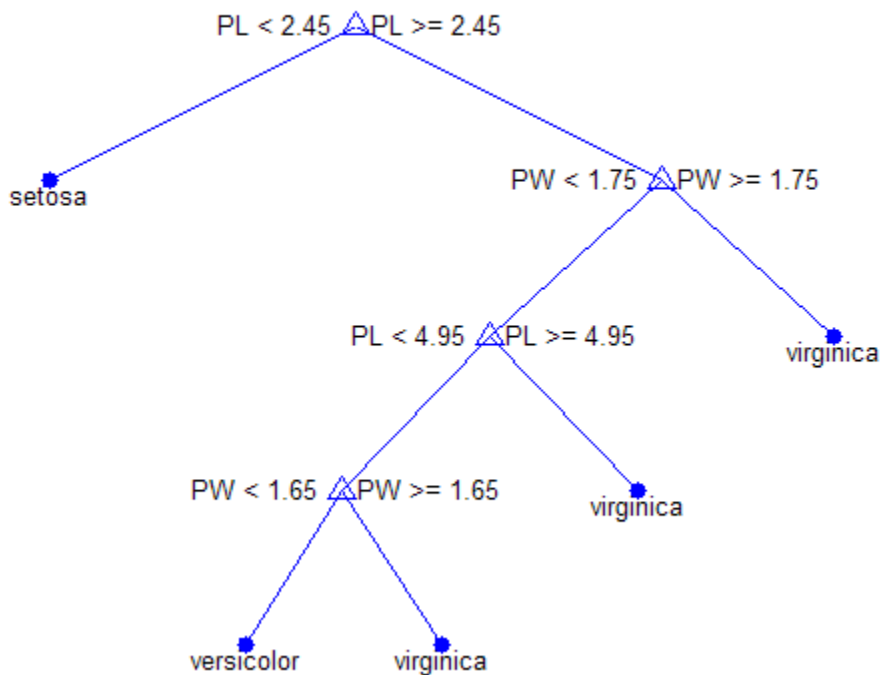
```
load fisheriris;

t = classregtree(meas, species, ...
                'names', {'SL' 'SW' 'PL' 'PW'})

t =
Decision tree for classification
1  if PL<2.45 then node 2 elseif PL>=2.45 then node 3 else setosa
2  class = setosa
3  if PW<1.75 then node 4 elseif PW>=1.75 then node 5 else versicolor
4  if PL<4.95 then node 6 elseif PL>=4.95 then node 7 else versicolor
5  class = virginica
6  if PW<1.65 then node 8 elseif PW>=1.65 then node 9 else versicolor
7  class = virginica
8  class = versicolor
9  class = virginica

view(t)
```

Click to display:  Magnification:  Pruning level:



```
ttype = type(t)
ttype =
classification
```

## References

- [1] Breiman, L., J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Boca Raton, FL: CRC Press, 1984.



## See Also

`classregtree`

## Type property

**Class:** cvpartition

Type of partition

## Description

The type of validation partition. It is 'kfold', 'holdout', 'leaveout', or 'resubstitution'.

## See Also

trainsize

## Type property

**Class:** grandset

Name of sequence on which point set  $P$  is based

## Description

$P$ .Type returns a character vector that contains the name of the sequence on which the point set  $P$  is based, for example 'Sobol'. You cannot change the Type property for a point set.

## BetaDistribution

Beta probability distribution object

### Description

A `BetaDistribution` object consist of parameters, a model description, and sample data for a beta probability distribution.

The beta distribution describes a family of curves that are unique in that they are nonzero only on the interval (0,1). A more general version of the distribution assigns parameters to the endpoints of the interval.

The beta distribution uses the following parameters.

Parameter	Description	Support
a	First shape parameter	$a > 0$
b	Second shape parameter	$b > 0$

### Creation

There are several ways to create a `BetaDistribution` probability distribution object.

- Create a distribution with specified parameter values using `makedist`.
- Fit a distribution to data using `fitdist`.
- Interactively fit a distribution to data using the **Distribution Fitter** app.

### Properties

#### Distribution Parameters

**a** — First shape parameter  
positive scalar value

First shape parameter of the beta distribution, specified as a positive scalar value.

Data Types: `single` | `double`

**b** — **Second shape parameter**

positive scalar value

Second shape parameter of the beta distribution, specified as a positive scalar value.

Data Types: `single` | `double`

## Distribution Characteristics

**IsTruncated** — **Logical flag for truncated distribution**

0 | 1

This property is read-only.

Logical flag for truncated distribution, specified as a logical value. If `IsTruncated` equals 0, the distribution is not truncated. If `IsTruncated` equals 1, the distribution is truncated.

Data Types: `logical`

**NumParameters** — **Number of parameters**

positive integer value

This property is read-only.

Number of parameters for the probability distribution, specified as a positive integer value.

Data Types: `single` | `double`

**ParameterCovariance** — **Covariance matrix of the parameter estimates**

matrix of scalar values

This property is read-only.

Covariance matrix of the parameter estimates, specified as a  $p$ -by- $p$  matrix, where  $p$  is the number of parameters in the distribution. The  $(i,j)$  element is the covariance between the estimates of the  $i$ th parameter and the  $j$ th parameter. The  $(i,i)$  element is the estimated variance of the  $i$ th parameter. If parameter  $i$  is fixed rather than

estimated by fitting the distribution to data, then the  $(i,i)$  elements of the covariance matrix are 0.

Data Types: `single` | `double`

**ParameterIsFixed** — Logical flag for fixed parameters

array of logical values

This property is read-only.

Logical flag for fixed parameters, specified as an array of logical values. If 0, the corresponding parameter in the `ParameterNames` array is not fixed. If 1, the corresponding parameter in the `ParameterNames` array is fixed.

Data Types: `logical`

**ParameterValues** — Distribution parameter values

vector of scalar values

This property is read-only.

Distribution parameter values, specified as a vector.

Data Types: `single` | `double`

**Truncation** — Truncation interval

vector of scalar values

This property is read-only.

Truncation interval for the probability distribution, specified as a vector containing the lower and upper truncation boundaries.

Data Types: `single` | `double`

## Other Object Properties

**DistributionName** — Probability distribution name

character vector

This property is read-only.

Probability distribution name, specified as a character vector.

Data Types: `char`

**InputData** — Data used for distribution fitting  
structure

This property is read-only.

Data used for distribution fitting, specified as a structure containing the following:

- `data`: Data vector used for distribution fitting.
- `cens`: Censoring vector, or empty if none.
- `freq`: Frequency vector, or empty if none.

Data Types: `struct`

**ParameterDescription** — Distribution parameter descriptions  
cell array of character vectors

This property is read-only.

Distribution parameter descriptions, specified as a cell array of character vectors. Each cell contains a short description of one distribution parameter.

Data Types: `char`

**ParameterNames** — Distribution parameter names  
cell array of character vectors

This property is read-only.

Distribution parameter names, specified as a cell array of character vectors.

Data Types: `char`

## Object Functions

<code>cdf</code>	Cumulative distribution functions
<code>icdf</code>	Inverse cumulative distribution functions
<code>iqr</code>	Interquartile range
<code>mean</code>	Mean of probability distribution
<code>median</code>	Median of probability distribution

negloglik	Negative log likelihood of probability distribution
paramci	Confidence intervals for probability distribution parameters
pdf	Probability density functions
proflik	Profile likelihood function for probability distribution
random	Random numbers
std	Standard deviation of probability distribution
truncate	Truncate probability distribution object
var	Variance of probability distribution

## Examples

### Create a Beta Distribution Object Using Default Parameters

Create a beta distribution object using the default parameter values.

```
pd = makedist('Beta')
```

```
pd =  
  
BetaDistribution  
  
Beta distribution  
a = 1  
b = 1
```

### Create a Beta Distribution Object Using Specified Parameters

Create a beta distribution object by specifying the parameter values.

```
pd = makedist('Beta', 'a', 2, 'b', 4)
```

```
pd =  
  
BetaDistribution  
  
Beta distribution  
a = 2  
b = 4
```



Compute the mean of the distribution.

```
m = mean(pd)
```

```
m =
```

```
0.3333
```

## See Also

**Distribution Fitter** | `fitdist` | `makedist`

## Topics

“Beta Distribution” on page B-4

**Introduced in R2013a**

## BinomialDistribution

Binomial probability distribution object

### Description

A `BinomialDistribution` object consists of parameters, a model description, and sample data for a binomial probability distribution

The binomial distribution models the total number of successes in repeated trials from an infinite population under the following conditions:

- Only two outcomes are possible for each of  $n$  trials.
- The probability of success for each trial is constant.
- All trials are independent of each other.

The binomial distribution uses the following parameters.

Parameter	Description	Support
$N$	Number of trials	positive integer
$p$	Probability of success	$0 \leq p \leq 1$

### Creation

There are several ways to create a `BinomialDistribution` probability distribution object.

- Create a distribution with specified parameter values using `makedist`.
- Fit a distribution to data using `fitdist`.
- Interactively fit a distribution to data using the **Distribution Fitter** app.

## Properties

### Distribution Parameters

**n** — Number of trials

positive integer value

Number of trials for the binomial distribution, specified as a positive integer value.

Data Types: `single` | `double`

**p** — Probability of success

positive scalar value in the range  $[0, 1]$

Probability of success of any individual trial for the binomial distribution, specified as a positive scalar value in the range  $[0, 1]$ .

Data Types: `single` | `double`

### Distribution Characteristics

**IsTruncated** — Logical flag for truncated distribution

0 | 1

This property is read-only.

Logical flag for truncated distribution, specified as a logical value. If `IsTruncated` equals 0, the distribution is not truncated. If `IsTruncated` equals 1, the distribution is truncated.

Data Types: `logical`

**NumParameters** — Number of parameters

positive integer value

This property is read-only.

Number of parameters for the probability distribution, specified as a positive integer value.

Data Types: `single` | `double`

**ParameterCovariance** — Covariance matrix of the parameter estimates

matrix of scalar values

This property is read-only.

Covariance matrix of the parameter estimates, specified as a  $p$ -by- $p$  matrix, where  $p$  is the number of parameters in the distribution. The  $(i,j)$  element is the covariance between the estimates of the  $i$ th parameter and the  $j$ th parameter. The  $(i,i)$  element is the estimated variance of the  $i$ th parameter. If parameter  $i$  is fixed rather than estimated by fitting the distribution to data, then the  $(i,i)$  elements of the covariance matrix are 0.

Data Types: `single` | `double`

**ParameterIsFixed** — Logical flag for fixed parameters

array of logical values

This property is read-only.

Logical flag for fixed parameters, specified as an array of logical values. If 0, the corresponding parameter in the `ParameterNames` array is not fixed. If 1, the corresponding parameter in the `ParameterNames` array is fixed.

Data Types: `logical`

**ParameterValues** — Distribution parameter values

vector of scalar values

This property is read-only.

Distribution parameter values, specified as a vector.

Data Types: `single` | `double`

**Truncation** — Truncation interval

vector of scalar values

This property is read-only.

Truncation interval for the probability distribution, specified as a vector containing the lower and upper truncation boundaries.

Data Types: `single` | `double`

## Other Object Properties

### **DistributionName** — Probability distribution name

character vector

This property is read-only.

Probability distribution name, specified as a character vector.

Data Types: `char`

### **InputData** — Data used for distribution fitting

structure

This property is read-only.

Data used for distribution fitting, specified as a structure containing the following:

- `data`: Data vector used for distribution fitting.
- `cens`: Censoring vector, or empty if none.
- `freq`: Frequency vector, or empty if none.

Data Types: `struct`

### **ParameterDescription** — Distribution parameter descriptions

cell array of character vectors

This property is read-only.

Distribution parameter descriptions, specified as a cell array of character vectors. Each cell contains a short description of one distribution parameter.

Data Types: `char`

### **ParameterNames** — Distribution parameter names

cell array of character vectors

This property is read-only.

Distribution parameter names, specified as a cell array of character vectors.

Data Types: `char`

## Object Functions

<code>cdf</code>	Cumulative distribution functions
<code>icdf</code>	Inverse cumulative distribution functions
<code>iqr</code>	Interquartile range
<code>mean</code>	Mean of probability distribution
<code>median</code>	Median of probability distribution
<code>negloglik</code>	Negative log likelihood of probability distribution
<code>paramci</code>	Confidence intervals for probability distribution parameters
<code>pdf</code>	Probability density functions
<code>proflik</code>	Profile likelihood function for probability distribution
<code>random</code>	Random numbers
<code>std</code>	Standard deviation of probability distribution
<code>truncate</code>	Truncate probability distribution object
<code>var</code>	Variance of probability distribution

## Examples

### Create a Negative Binomial Distribution Object Using Default Parameters

Create a negative binomial distribution object using the default parameter values.

```
pd = makedist('NegativeBinomial')
```

```
pd =
```

```
  NegativeBinomialDistribution
```

```
  Negative Binomial distribution
```

```
  R = 1
```

```
  P = 0.5
```

### Create a Binomial Distribution Object Using Specified Parameters

Create a binomial distribution object by specifying the parameter values.

```
pd = makedist('Binomial', 'N', 30, 'p', 0.25)
```

```
pd =
```

```
BinomialDistribution
```

```
Binomial distribution
```

```
N = 30
```

```
p = 0.25
```

Compute the mean of the distribution.

```
m = mean(pd)
```

```
m =
```

```
7.5000
```

## See Also

**Distribution Fitter** | `fitdist` | `makedist`

## Topics

“Binomial Distribution” on page B-9

“Bernoulli Distribution” on page B-2

**Introduced in R2013a**

## BirnbaumSaundersDistribution

Birnbaum-Saunders probability distribution object

### Description

A `BirnbaumSaundersDistribution` object consists of parameters, a model description, and sample data for a Birnbaum-Saunders probability distribution.

The Birnbaum-Saunders distribution was originally proposed as a lifetime model for materials subject to cyclic patterns of stress and strain, where the ultimate failure of the material comes from the growth of a prominent flaw. In materials science, Miner's Rule suggests that the damage occurring after  $n$  cycles, at a stress level with an expected lifetime of  $N$  cycles, is proportional to  $n / N$ . Whenever Miner's Rule applies, the Birnbaum-Saunders model is a reasonable choice for a lifetime distribution model.

The Birnbaum-Saunders distribution uses the following parameters.

Parameter	Description	Support
beta	scale parameter	$\beta > 0$
gamma	shape parameter	$\gamma > 0$

### Creation

There are several ways to create a `BirnbaumSaundersDistribution` probability distribution object.

- Create a distribution with specified parameter values using `makedist`.
- Fit a distribution to data using `fitdist`.
- Interactively fit a distribution to data using the **Distribution Fitter** app.



## Properties

### Distribution Parameters

**beta** — Scale parameter

positive scalar value

Scale parameter of the Birnbaum-Saunders distribution, specified as a positive scalar value.

Data Types: `single` | `double`**gamma** — Shape parameter

positive scalar value

Shape parameter of the Birnbaum-Saunders distribution, specified as a positive scalar value.

Data Types: `single` | `double`

### Distribution Characteristics

**IsTruncated** — Logical flag for truncated distribution

0 | 1

This property is read-only.

Logical flag for truncated distribution, specified as a logical value. If `IsTruncated` equals 0, the distribution is not truncated. If `IsTruncated` equals 1, the distribution is truncated.

Data Types: `logical`**NumParameters** — Number of parameters

positive integer value

This property is read-only.

Number of parameters for the probability distribution, specified as a positive integer value.

Data Types: `single` | `double`

**ParameterCovariance** — Covariance matrix of the parameter estimates

matrix of scalar values

This property is read-only.

Covariance matrix of the parameter estimates, specified as a  $p$ -by- $p$  matrix, where  $p$  is the number of parameters in the distribution. The  $(i,j)$  element is the covariance between the estimates of the  $i$ th parameter and the  $j$ th parameter. The  $(i,i)$  element is the estimated variance of the  $i$ th parameter. If parameter  $i$  is fixed rather than estimated by fitting the distribution to data, then the  $(i,i)$  elements of the covariance matrix are 0.

Data Types: `single` | `double`

**ParameterIsFixed** — Logical flag for fixed parameters

array of logical values

This property is read-only.

Logical flag for fixed parameters, specified as an array of logical values. If 0, the corresponding parameter in the `ParameterNames` array is not fixed. If 1, the corresponding parameter in the `ParameterNames` array is fixed.

Data Types: `logical`

**ParameterValues** — Distribution parameter values

vector of scalar values

This property is read-only.

Distribution parameter values, specified as a vector.

Data Types: `single` | `double`

**Truncation** — Truncation interval

vector of scalar values

This property is read-only.

Truncation interval for the probability distribution, specified as a vector containing the lower and upper truncation boundaries.

Data Types: `single` | `double`

## Other Object Properties

### **DistributionName** — Probability distribution name

character vector

This property is read-only.

Probability distribution name, specified as a character vector.

Data Types: `char`

### **InputData** — Data used for distribution fitting

structure

This property is read-only.

Data used for distribution fitting, specified as a structure containing the following:

- `data`: Data vector used for distribution fitting.
- `cens`: Censoring vector, or empty if none.
- `freq`: Frequency vector, or empty if none.

Data Types: `struct`

### **ParameterDescription** — Distribution parameter descriptions

cell array of character vectors

This property is read-only.

Distribution parameter descriptions, specified as a cell array of character vectors. Each cell contains a short description of one distribution parameter.

Data Types: `char`

### **ParameterNames** — Distribution parameter names

cell array of character vectors

This property is read-only.

Distribution parameter names, specified as a cell array of character vectors.

Data Types: `char`

## Object Functions

<code>cdf</code>	Cumulative distribution functions
<code>icdf</code>	Inverse cumulative distribution functions
<code>iqr</code>	Interquartile range
<code>mean</code>	Mean of probability distribution
<code>median</code>	Median of probability distribution
<code>negloglik</code>	Negative log likelihood of probability distribution
<code>paramci</code>	Confidence intervals for probability distribution parameters
<code>pdf</code>	Probability density functions
<code>proflk</code>	Profile likelihood function for probability distribution
<code>random</code>	Random numbers
<code>std</code>	Standard deviation of probability distribution
<code>truncate</code>	Truncate probability distribution object
<code>var</code>	Variance of probability distribution

## Examples

### Create a Birnbaum-Saunders Distribution Object Using Default Parameters

Create a Birnbaum-Saunders distribution object using the default parameter values.

```
pd = makedist('BirnbaumSaunders')
```

```
pd =
```

```
BirnbaumSaundersDistribution  
  
Birnbaum-Saunders distribution  
  beta = 1  
  gamma = 1
```

### Create a Birnbaum-Saunders Distribution Object Using Specified Parameter Values

Create a Birnbaum-Saunders distribution object by specifying the parameter values.

```
pd = makedist('BirnbaumSaunders', 'beta', 2, 'gamma', 5)
```

```
pd =
```

```
BirnbauSaundersDistribution
```

```
Birnbau-Saunders distribution  
  beta = 2  
  gamma = 5
```

Compute the mean of the distribution.

```
m = mean(pd)
```

```
m =
```

```
  27
```

## See Also

**Distribution Fitter** | `fitdist` | `makedist`

## Topics

“Birnbau-Saunders Distribution” on page B-13

**Introduced in R2013a**

## BurrDistribution

Burr probability distribution object

### Description

A `BurrDistribution` object consists of parameters, a model description, and sample data for a Burr probability distribution.

The Burr distribution is a three-parameter family of distributions on the positive real line. It can fit a wide range of empirical data, and is used in various fields such as finance, hydrology, and reliability to model a variety of data types.

The Burr distribution uses the following parameters.

Parameter	Description	Support
alpha	Scale parameter	$\alpha > 0$
c	First shape parameter	$c > 0$
k	Second shape parameter	$k > 0$

### Creation

There are several ways to create a `BurrDistribution` probability distribution object.

- Create a distribution with specified parameter values using `makedist`.
- Fit a distribution to data using `fitdist`.
- Interactively fit a distribution to data using the **Distribution Fitter** app.

## Properties

### Distribution Parameters

**alpha** — Scale parameter

positive scalar value

Scale parameter of the Burr distribution, specified as a positive scalar value.

Data Types: `single` | `double`

**c** — First shape parameter

positive scalar value

First shape parameter of the Burr distribution, specified as a positive scalar value.

Data Types: `single` | `double`

**k** — Second shape parameter

positive scalar value

Second shape parameter of the Burr distribution, specified as a positive scalar value.

Data Types: `single` | `double`

### Distribution Characteristics

**IsTruncated** — Logical flag for truncated distribution

0 | 1

This property is read-only.

Logical flag for truncated distribution, specified as a logical value. If `IsTruncated` equals 0, the distribution is not truncated. If `IsTruncated` equals 1, the distribution is truncated.

Data Types: `logical`

**NumParameters** — Number of parameters

positive integer value

This property is read-only.

Number of parameters for the probability distribution, specified as a positive integer value.

Data Types: `single` | `double`

**ParameterCovariance** — Covariance matrix of the parameter estimates

matrix of scalar values

This property is read-only.

Covariance matrix of the parameter estimates, specified as a  $p$ -by- $p$  matrix, where  $p$  is the number of parameters in the distribution. The  $(i,j)$  element is the covariance between the estimates of the  $i$ th parameter and the  $j$ th parameter. The  $(i,i)$  element is the estimated variance of the  $i$ th parameter. If parameter  $i$  is fixed rather than estimated by fitting the distribution to data, then the  $(i,i)$  elements of the covariance matrix are 0.

Data Types: `single` | `double`

**ParameterIsFixed** — Logical flag for fixed parameters

array of logical values

This property is read-only.

Logical flag for fixed parameters, specified as an array of logical values. If 0, the corresponding parameter in the `ParameterNames` array is not fixed. If 1, the corresponding parameter in the `ParameterNames` array is fixed.

Data Types: `logical`

**ParameterValues** — Distribution parameter values

vector of scalar values

This property is read-only.

Distribution parameter values, specified as a vector.

Data Types: `single` | `double`

**Truncation** — Truncation interval

vector of scalar values

This property is read-only.



Truncation interval for the probability distribution, specified as a vector containing the lower and upper truncation boundaries.

Data Types: `single` | `double`

## Other Object Properties

### **DistributionName** — Probability distribution name

character vector

This property is read-only.

Probability distribution name, specified as a character vector.

Data Types: `char`

### **InputData** — Data used for distribution fitting

structure

This property is read-only.

Data used for distribution fitting, specified as a structure containing the following:

- `data`: Data vector used for distribution fitting.
- `cens`: Censoring vector, or empty if none.
- `freq`: Frequency vector, or empty if none.

Data Types: `struct`

### **ParameterDescription** — Distribution parameter descriptions

cell array of character vectors

This property is read-only.

Distribution parameter descriptions, specified as a cell array of character vectors. Each cell contains a short description of one distribution parameter.

Data Types: `char`

### **ParameterNames** — Distribution parameter names

cell array of character vectors

This property is read-only.

Distribution parameter names, specified as a cell array of character vectors.

Data Types: `char`

## Object Functions

<code>cdf</code>	Cumulative distribution functions
<code>icdf</code>	Inverse cumulative distribution functions
<code>iqr</code>	Interquartile range
<code>mean</code>	Mean of probability distribution
<code>median</code>	Median of probability distribution
<code>negloglik</code>	Negative log likelihood of probability distribution
<code>paramci</code>	Confidence intervals for probability distribution parameters
<code>pdf</code>	Probability density functions
<code>proflik</code>	Profile likelihood function for probability distribution
<code>random</code>	Random numbers
<code>std</code>	Standard deviation of probability distribution
<code>truncate</code>	Truncate probability distribution object
<code>var</code>	Variance of probability distribution

## Examples

### Create a Burr Distribution Object Using Default Parameters

Create a Burr distribution object using the default parameter values.

```
pd = makedist('Burr')
```

```
pd =
```

```
    BurrDistribution
```

```
    Burr distribution
```

```
    alpha = 1
```

```
    c = 1
```

```
k = 1
```

### Create a Burr Distribution Object Using Specified Parameters

Create a Burr distribution object by specifying parameter values.

```
pd = makedist('Burr', 'alpha', 1, 'c', 2, 'k', 5)
```

```
pd =
```

```
BurrDistribution  
  
Burr distribution  
  alpha = 1  
    c = 2  
    k = 5
```

Compute the mean of the distribution.

```
m = mean(pd)
```

```
m =
```

```
0.4295
```

## See Also

**Distribution Fitter** | `fitdist` | `makedist`

## Topics

“Burr Type XII Distribution” on page B-15

**Introduced in R2013a**

# ExponentialDistribution

Exponential probability distribution object

## Description

An `ExponentialDistribution` object consists of parameters, a model description, and sample data for an exponential probability distribution.

The exponential distribution is used to model events that occur randomly over time, and its main application area is studies of lifetimes. It is a special case of the gamma distribution with the shape parameter  $\alpha = 1$ .

The exponential distribution uses the following parameters.

Parameter	Description	Support
<code>mu</code>	Mean	$\mu > 0$

## Creation

There are several ways to create a `ExponentialDistribution` probability distribution object.

- Create a distribution with specified parameter values using `makedist`.
- Fit a distribution to data using `fitdist`.
- Interactively fit a distribution to data using the **Distribution Fitter** app.

## Properties

### Distribution Parameter

**`mu`** — Mean

positive scalar value

Mean of the exponential distribution, specified as a positive scalar value.

Data Types: `single` | `double`

## Distribution Characteristics

### **IsTruncated** — Logical flag for truncated distribution

0 | 1

This property is read-only.

Logical flag for truncated distribution, specified as a logical value. If `IsTruncated` equals 0, the distribution is not truncated. If `IsTruncated` equals 1, the distribution is truncated.

Data Types: `logical`

### **NumParameters** — Number of parameters

positive integer value

This property is read-only.

Number of parameters for the probability distribution, specified as a positive integer value.

Data Types: `single` | `double`

### **ParameterCovariance** — Covariance matrix of the parameter estimates

matrix of scalar values

This property is read-only.

Covariance matrix of the parameter estimates, specified as a  $p$ -by- $p$  matrix, where  $p$  is the number of parameters in the distribution. The  $(i,j)$  element is the covariance between the estimates of the  $i$ th parameter and the  $j$ th parameter. The  $(i,i)$  element is the estimated variance of the  $i$ th parameter. If parameter  $i$  is fixed rather than estimated by fitting the distribution to data, then the  $(i,i)$  elements of the covariance matrix are 0.

Data Types: `single` | `double`

### **ParameterIsFixed** — Logical flag for fixed parameters

array of logical values

This property is read-only.

Logical flag for fixed parameters, specified as an array of logical values. If 0, the corresponding parameter in the `ParameterNames` array is not fixed. If 1, the corresponding parameter in the `ParameterNames` array is fixed.

Data Types: `logical`

#### **ParameterValues** — Distribution parameter values

vector of scalar values

This property is read-only.

Distribution parameter values, specified as a vector.

Data Types: `single` | `double`

#### **Truncation** — Truncation interval

vector of scalar values

This property is read-only.

Truncation interval for the probability distribution, specified as a vector containing the lower and upper truncation boundaries.

Data Types: `single` | `double`

## Other Object Properties

#### **DistributionName** — Probability distribution name

character vector

This property is read-only.

Probability distribution name, specified as a character vector.

Data Types: `char`

#### **InputData** — Data used for distribution fitting

structure

This property is read-only.

Data used for distribution fitting, specified as a structure containing the following:

- `data`: Data vector used for distribution fitting.
- `cens`: Censoring vector, or empty if none.
- `freq`: Frequency vector, or empty if none.

Data Types: `struct`

#### **ParameterDescription** — Distribution parameter descriptions

cell array of character vectors

This property is read-only.

Distribution parameter descriptions, specified as a cell array of character vectors. Each cell contains a short description of one distribution parameter.

Data Types: `char`

#### **ParameterNames** — Distribution parameter names

cell array of character vectors

This property is read-only.

Distribution parameter names, specified as a cell array of character vectors.

Data Types: `char`

## Object Functions

<code>cdf</code>	Cumulative distribution functions
<code>icdf</code>	Inverse cumulative distribution functions
<code>iqr</code>	Interquartile range
<code>mean</code>	Mean of probability distribution
<code>median</code>	Median of probability distribution
<code>negloglik</code>	Negative log likelihood of probability distribution
<code>paramci</code>	Confidence intervals for probability distribution parameters
<code>pdf</code>	Probability density functions
<code>proflik</code>	Profile likelihood function for probability distribution
<code>random</code>	Random numbers
<code>std</code>	Standard deviation of probability distribution
<code>truncate</code>	Truncate probability distribution object
<code>var</code>	Variance of probability distribution

## Examples

### Create an Exponential Distribution Object Using Default Parameters

Create an exponential distribution object using the default parameter values.

```
pd = makedist('Exponential')
```

```
pd =  
  
ExponentialDistribution  
  
Exponential distribution  
mu = 1
```

### Create an Exponential Distribution Object Using Specified Parameters

Create an exponential distribution object by specifying the parameter values.

```
pd = makedist('Exponential', 'mu', 2)
```

```
pd =  
  
ExponentialDistribution  
  
Exponential distribution  
mu = 2
```

Compute the variance of the distribution.

```
v = var(pd)
```

```
v =
```



## See Also

**Distribution Fitter** | `fitdist` | `makedist`

## Topics

“Exponential Distribution” on page B-34

**Introduced in R2013a**

# ExtremeValueDistribution

Extreme value probability distribution object

## Description

An `ExtremeValueDistribution` object consists of parameters, a model description, and sample data for an extreme value probability distribution.

The extreme value distribution is appropriate for modeling the smallest value from a distribution whose tails decay exponentially fast, for example, the normal distribution. It can also model the largest value from a distribution, such as the normal or exponential distributions, by using the negative of the original values.

The extreme value distribution uses the following parameters.

Parameter	Description	Support
<code>mu</code>	Location parameter	$-\infty < \mu < \infty$
<code>sigma</code>	Scale parameter	$\sigma \geq 0$

## Creation

There are several ways to create a `ExtremeValueDistribution` probability distribution object.

- Create a distribution with specified parameter values using `makedist`.
- Fit a distribution to data using `fitdist`.
- Interactively fit a distribution to data using the **Distribution Fitter** app.

## Properties

### Distribution Parameters

**mu** — Location parameter

scalar value

Location parameter of the extreme value distribution, specified as a scalar value.

Data Types: `single` | `double`

**sigma** — Scale parameter

nonnegative scalar value

Scale parameter of the extreme value distribution, specified as a nonnegative scalar value.

Data Types: `single` | `double`

### Distribution Characteristics

**IsTruncated** — Logical flag for truncated distribution

0 | 1

This property is read-only.

Logical flag for truncated distribution, specified as a logical value. If `IsTruncated` equals 0, the distribution is not truncated. If `IsTruncated` equals 1, the distribution is truncated.

Data Types: `logical`

**NumParameters** — Number of parameters

positive integer value

This property is read-only.

Number of parameters for the probability distribution, specified as a positive integer value.

Data Types: `single` | `double`

**ParameterCovariance** — Covariance matrix of the parameter estimates

matrix of scalar values

This property is read-only.

Covariance matrix of the parameter estimates, specified as a  $p$ -by- $p$  matrix, where  $p$  is the number of parameters in the distribution. The  $(i,j)$  element is the covariance between the estimates of the  $i$ th parameter and the  $j$ th parameter. The  $(i,i)$  element is the estimated variance of the  $i$ th parameter. If parameter  $i$  is fixed rather than estimated by fitting the distribution to data, then the  $(i,i)$  elements of the covariance matrix are 0.

Data Types: `single` | `double`

**ParameterIsFixed** — Logical flag for fixed parameters

array of logical values

This property is read-only.

Logical flag for fixed parameters, specified as an array of logical values. If 0, the corresponding parameter in the `ParameterNames` array is not fixed. If 1, the corresponding parameter in the `ParameterNames` array is fixed.

Data Types: `logical`

**ParameterValues** — Distribution parameter values

vector of scalar values

This property is read-only.

Distribution parameter values, specified as a vector.

Data Types: `single` | `double`

**Truncation** — Truncation interval

vector of scalar values

This property is read-only.

Truncation interval for the probability distribution, specified as a vector containing the lower and upper truncation boundaries.

Data Types: `single` | `double`

## Other Object Properties

### **DistributionName** — Probability distribution name

character vector

This property is read-only.

Probability distribution name, specified as a character vector.

Data Types: `char`

### **InputData** — Data used for distribution fitting

structure

This property is read-only.

Data used for distribution fitting, specified as a structure containing the following:

- `data`: Data vector used for distribution fitting.
- `cens`: Censoring vector, or empty if none.
- `freq`: Frequency vector, or empty if none.

Data Types: `struct`

### **ParameterDescription** — Distribution parameter descriptions

cell array of character vectors

This property is read-only.

Distribution parameter descriptions, specified as a cell array of character vectors. Each cell contains a short description of one distribution parameter.

Data Types: `char`

### **ParameterNames** — Distribution parameter names

cell array of character vectors

This property is read-only.

Distribution parameter names, specified as a cell array of character vectors.

Data Types: `char`

## Object Functions

<code>cdf</code>	Cumulative distribution functions
<code>icdf</code>	Inverse cumulative distribution functions
<code>iqr</code>	Interquartile range
<code>mean</code>	Mean of probability distribution
<code>median</code>	Median of probability distribution
<code>negloglik</code>	Negative log likelihood of probability distribution
<code>paramci</code>	Confidence intervals for probability distribution parameters
<code>pdf</code>	Probability density functions
<code>proflik</code>	Profile likelihood function for probability distribution
<code>random</code>	Random numbers
<code>std</code>	Standard deviation of probability distribution
<code>truncate</code>	Truncate probability distribution object
<code>var</code>	Variance of probability distribution

## Examples

### Create an Extreme Value Distribution Object Using Default Parameters

Create an extreme value distribution object using the default parameter values.

```
pd = makedist('ExtremeValue')
```

```
pd =
```

```
ExtremeValueDistribution  
  
Extreme Value distribution  
    mu = 0  
    sigma = 1
```

### Create an Extreme Value Distribution Object Using Specified Parameters

Create an extreme value distribution object by specifying the parameter values.

```
pd = makedist('ExtremeValue', 'mu', -1, 'sigma', 2)
```

```
pd =
```

```
ExtremeValueDistribution  
  
Extreme Value distribution  
  mu = -1  
  sigma = 2
```

Compute the standard deviation for the distribution.

```
s = std(pd)
```

```
s =
```

```
2.5651
```

## See Also

**Distribution Fitter** | `fitdist` | `makedist`

## Topics

“Extreme Value Distribution” on page B-38

**Introduced in R2013a**

# GammaDistribution

Gamma probability distribution object

## Description

A `GammaDistribution` object consists of parameters, a model description, and sample data for a gamma probability distribution.

The gamma distribution is a two-parameter family of distributions used to model sums of exponentially distributed random variables. The chi-square and the exponential distributions, which are special cases of the gamma distribution, are one-parameter distributions that fix one of the two gamma parameters.

The gamma distribution uses the following parameters.

Parameter	Description	Support
a	Shape parameter	$a > 0$
b	Scale parameter	$b > 0$

## Creation

There are several ways to create a `GammaDistribution` probability distribution object.

- Create a distribution with specified parameter values using `makedist`.
- Fit a distribution to data using `fitdist`.
- Interactively fit a distribution to data using the **Distribution Fitter** app.



## Properties

### Distribution Parameters

**a** — Shape parameter

positive scalar value

Shape parameter for the gamma distribution, specified as a positive scalar value.

Data Types: `single` | `double`**b** — Scale parameter

nonnegative scalar value

Scale parameter for the gamma distribution, specified as a nonnegative scalar value.

Data Types: `single` | `double`

### Distribution Characteristics

**IsTruncated** — Logical flag for truncated distribution

0 | 1

This property is read-only.

Logical flag for truncated distribution, specified as a logical value. If `IsTruncated` equals 0, the distribution is not truncated. If `IsTruncated` equals 1, the distribution is truncated.

Data Types: `logical`**NumParameters** — Number of parameters

positive integer value

This property is read-only.

Number of parameters for the probability distribution, specified as a positive integer value.

Data Types: `single` | `double`

**ParameterCovariance** — Covariance matrix of the parameter estimates

matrix of scalar values

This property is read-only.

Covariance matrix of the parameter estimates, specified as a  $p$ -by- $p$  matrix, where  $p$  is the number of parameters in the distribution. The  $(i,j)$  element is the covariance between the estimates of the  $i$ th parameter and the  $j$ th parameter. The  $(i,i)$  element is the estimated variance of the  $i$ th parameter. If parameter  $i$  is fixed rather than estimated by fitting the distribution to data, then the  $(i,i)$  elements of the covariance matrix are 0.

Data Types: `single` | `double`

**ParameterIsFixed** — Logical flag for fixed parameters

array of logical values

This property is read-only.

Logical flag for fixed parameters, specified as an array of logical values. If 0, the corresponding parameter in the `ParameterNames` array is not fixed. If 1, the corresponding parameter in the `ParameterNames` array is fixed.

Data Types: `logical`

**ParameterValues** — Distribution parameter values

vector of scalar values

This property is read-only.

Distribution parameter values, specified as a vector.

Data Types: `single` | `double`

**Truncation** — Truncation interval

vector of scalar values

This property is read-only.

Truncation interval for the probability distribution, specified as a vector containing the lower and upper truncation boundaries.

Data Types: `single` | `double`

## Other Object Properties

### **DistributionName** — Probability distribution name

character vector

This property is read-only.

Probability distribution name, specified as a character vector.

Data Types: `char`

### **InputData** — Data used for distribution fitting

structure

This property is read-only.

Data used for distribution fitting, specified as a structure containing the following:

- `data`: Data vector used for distribution fitting.
- `cens`: Censoring vector, or empty if none.
- `freq`: Frequency vector, or empty if none.

Data Types: `struct`

### **ParameterDescription** — Distribution parameter descriptions

cell array of character vectors

This property is read-only.

Distribution parameter descriptions, specified as a cell array of character vectors. Each cell contains a short description of one distribution parameter.

Data Types: `char`

### **ParameterNames** — Distribution parameter names

cell array of character vectors

This property is read-only.

Distribution parameter names, specified as a cell array of character vectors.

Data Types: `char`

## Object Functions

<code>cdf</code>	Cumulative distribution functions
<code>icdf</code>	Inverse cumulative distribution functions
<code>iqr</code>	Interquartile range
<code>mean</code>	Mean of probability distribution
<code>median</code>	Median of probability distribution
<code>negloglik</code>	Negative log likelihood of probability distribution
<code>paramci</code>	Confidence intervals for probability distribution parameters
<code>pdf</code>	Probability density functions
<code>proflik</code>	Profile likelihood function for probability distribution
<code>random</code>	Random numbers
<code>std</code>	Standard deviation of probability distribution
<code>truncate</code>	Truncate probability distribution object
<code>var</code>	Variance of probability distribution

## Examples

### Create a Gamma Distribution Object Using Default Parameters

Create a gamma distribution object using the default parameter values.

```
pd = makedist('Gamma')
```

```
pd =
```

```
GammaDistribution
```

```
Gamma distribution
```

```
  a = 1
```

```
  b = 1
```

### Create a Gamma Distribution Object Using Specified Parameters

Create a gamma distribution object by specifying the parameter values.

```
pd = makedist('Gamma', 'a', 2, 'b', 4)
```

```
pd =  
  
GammaDistribution  
  
Gamma distribution  
  a = 2  
  b = 4
```

Compute the mean of the distribution.

```
m = mean(pd)
```

```
m =  
  
      8
```

## See Also

**Distribution Fitter** | `fitdist` | `makedist`

## Topics

“Gamma Distribution” on page B-47

**Introduced in R2013a**

# GeneralizedExtremeValueDistribution

Generalized extreme value probability distribution object

## Description

A `GeneralizedExtremeValueDistribution` object consists of parameters, a model description, and sample data for a generalized extreme value probability distribution.

The generalized extreme value distribution is often used to model the smallest or largest value among a large set of independent, identically distributed random values representing measurements or observations. It combines three simpler distributions into a single form, allowing a continuous range of possible shapes that include all three of the simpler distributions.

The three distribution types correspond to the limiting distribution of block maxima from different classes of underlying distributions:

- Type 1 — Distributions whose tails decrease exponentially, such as the normal distribution
- Type 2 — Distributions whose tails decrease as a polynomial, such as Student's  $t$  distribution
- Type 3 — Distributions whose tails are finite, such as the beta distribution

The generalized extreme value distribution uses the following parameters.

Parameter	Description	Support
<code>k</code>	Shape parameter	$-\infty \leq k \leq \infty$
<code>sigma</code>	Scale parameter	$\sigma \geq 0$
<code>mu</code>	Location parameter	$-\infty \leq \mu \leq \infty$

## Creation

There are several ways to create a `GeneralizedExtremeValueDistribution` probability distribution object.

- Create a distribution with specified parameter values using `makedist`.
- Fit a distribution to data using `fitdist`.
- Interactively fit a distribution to data using the **Distribution Fitter** app.

## Properties

### Distribution Parameters

#### **k** — Shape parameter

scalar value

Shape parameter of the generalized extreme value distribution, specified as a scalar value.

Data Types: `single` | `double`

#### **sigma** — Scale parameter

nonnegative scalar value

Scale parameter of the generalized extreme value distribution, specified as a nonnegative scalar value.

Data Types: `single` | `double`

#### **mu** — Location parameter

scalar value

Location parameter of the generalized extreme value distribution, specified as a scalar value.

Data Types: `single` | `double`

### Distribution Characteristics

#### **IsTruncated** — Logical flag for truncated distribution

0 | 1

This property is read-only.

Logical flag for truncated distribution, specified as a logical value. If `IsTruncated` equals 0, the distribution is not truncated. If `IsTruncated` equals 1, the distribution is truncated.

Data Types: `logical`

**NumParameters** — Number of parameters

positive integer value

This property is read-only.

Number of parameters for the probability distribution, specified as a positive integer value.

Data Types: `single` | `double`

**ParameterCovariance** — Covariance matrix of the parameter estimates

matrix of scalar values

This property is read-only.

Covariance matrix of the parameter estimates, specified as a  $p$ -by- $p$  matrix, where  $p$  is the number of parameters in the distribution. The  $(i,j)$  element is the covariance between the estimates of the  $i$ th parameter and the  $j$ th parameter. The  $(i,i)$  element is the estimated variance of the  $i$ th parameter. If parameter  $i$  is fixed rather than estimated by fitting the distribution to data, then the  $(i,i)$  elements of the covariance matrix are 0.

Data Types: `single` | `double`

**ParameterIsFixed** — Logical flag for fixed parameters

array of logical values

This property is read-only.

Logical flag for fixed parameters, specified as an array of logical values. If 0, the corresponding parameter in the `ParameterNames` array is not fixed. If 1, the corresponding parameter in the `ParameterNames` array is fixed.

Data Types: `logical`

**ParameterValues** — Distribution parameter values

vector of scalar values



This property is read-only.

Distribution parameter values, specified as a vector.

Data Types: `single` | `double`

#### **Truncation — Truncation interval**

vector of scalar values

This property is read-only.

Truncation interval for the probability distribution, specified as a vector containing the lower and upper truncation boundaries.

Data Types: `single` | `double`

## Other Object Properties

#### **DistributionName — Probability distribution name**

character vector

This property is read-only.

Probability distribution name, specified as a character vector.

Data Types: `char`

#### **InputData — Data used for distribution fitting**

structure

This property is read-only.

Data used for distribution fitting, specified as a structure containing the following:

- `data`: Data vector used for distribution fitting.
- `cens`: Censoring vector, or empty if none.
- `freq`: Frequency vector, or empty if none.

Data Types: `struct`

#### **ParameterDescription — Distribution parameter descriptions**

cell array of character vectors

This property is read-only.

Distribution parameter descriptions, specified as a cell array of character vectors. Each cell contains a short description of one distribution parameter.

Data Types: `char`

**ParameterNames** — Distribution parameter names

cell array of character vectors

This property is read-only.

Distribution parameter names, specified as a cell array of character vectors.

Data Types: `char`

## Object Functions

<code>cdf</code>	Cumulative distribution functions
<code>icdf</code>	Inverse cumulative distribution functions
<code>iqr</code>	Interquartile range
<code>mean</code>	Mean of probability distribution
<code>median</code>	Median of probability distribution
<code>negloglik</code>	Negative log likelihood of probability distribution
<code>paramci</code>	Confidence intervals for probability distribution parameters
<code>pdf</code>	Probability density functions
<code>proflik</code>	Profile likelihood function for probability distribution
<code>random</code>	Random numbers
<code>std</code>	Standard deviation of probability distribution
<code>truncate</code>	Truncate probability distribution object
<code>var</code>	Variance of probability distribution

## Examples

### Create a Generalized Extreme Value Distribution Object Using Default Parameters

Create a generalized extreme value distribution object using the default parameter values.

```
pd = makedist('GeneralizedExtremeValue')
```

```
pd =  
  
GeneralizedExtremeValueDistribution  
  
Generalized Extreme Value distribution  
    k = 0  
    sigma = 1  
    mu = 0
```

### Create a Generalized Extreme Value Distribution Object Using Specified Parameters

Create a generalized extreme value distribution object by specifying values for the parameters.

```
pd = makedist('GeneralizedExtremeValue', 'k', 0, 'sigma', 2, 'mu', 1)
```

```
pd =  
  
GeneralizedExtremeValueDistribution  
  
Generalized Extreme Value distribution  
    k = 0  
    sigma = 2  
    mu = 1
```

Compute the mean of the distribution.

```
m = mean(pd)
```

```
m =  
  
    2.1544
```

## See Also

**Distribution Fitter** | `fitdist` | `makedist`

## **Topics**

“Generalized Extreme Value Distribution” on page B-53

**Introduced in R2013a**

# GeneralizedParetoDistribution

Generalized Pareto probability distribution object

## Description

A `GeneralizedParetoDistribution` object consists of parameters, a model description, and sample data for a generalized Pareto probability distribution.

The generalized Pareto distribution is used to model the tails of another distribution. It allows a continuous range of possible shapes that include both the exponential and Pareto distributions as special cases. It has three basic forms, each corresponding to a limiting distribution of exceedence data from a different class of underlying distributions.

- Distributions whose tails decrease exponentially, such as the normal, lead to a generalized Pareto shape parameter of zero.
- Distributions whose tails decrease polynomially, such as the Student's  $t$ , lead to a positive shape parameter.
- Distributions whose tails are finite, such as the beta, lead to a negative shape parameter.

The generalized Pareto distribution uses the following parameters.

Parameter	Description	Support
<code>k</code>	Shape parameter	$-\infty < k < \infty$
<code>sigma</code>	Scale parameter	$\sigma \geq 0$
<code>theta</code>	Location parameter	$-\infty < \theta < \infty$

## Creation

There are several ways to create a `GeneralizedParetoDistribution` probability distribution object.

- Create a distribution with specified parameter values using `makedist`.

- Fit a distribution to data using `fitdist`.
- Interactively fit a distribution to data using the **Distribution Fitter** app.

## Properties

### Distribution Parameters

#### **`k`** — Shape parameter

scalar value

Shape parameter for the generalized Pareto distribution, specified as a scalar value.

Data Types: `single` | `double`

#### **`sigma`** — Scale parameter

nonnegative scalar value

Scale parameter for the generalized Pareto distribution, specified as a nonnegative scalar value.

Data Types: `single` | `double`

#### **`theta`** — Location parameter

scalar value

Location parameter for the generalized Pareto distribution, specified as a scalar value.

Data Types: `single` | `double`

### Distribution Characteristics

#### **`IsTruncated`** — Logical flag for truncated distribution

0 | 1

This property is read-only.

Logical flag for truncated distribution, specified as a logical value. If `IsTruncated` equals 0, the distribution is not truncated. If `IsTruncated` equals 1, the distribution is truncated.

Data Types: `logical`

**NumParameters** — Number of parameters

positive integer value

This property is read-only.

Number of parameters for the probability distribution, specified as a positive integer value.

Data Types: `single` | `double`

**ParameterCovariance** — Covariance matrix of the parameter estimates

matrix of scalar values

This property is read-only.

Covariance matrix of the parameter estimates, specified as a  $p$ -by- $p$  matrix, where  $p$  is the number of parameters in the distribution. The  $(i,j)$  element is the covariance between the estimates of the  $i$ th parameter and the  $j$ th parameter. The  $(i,i)$  element is the estimated variance of the  $i$ th parameter. If parameter  $i$  is fixed rather than estimated by fitting the distribution to data, then the  $(i,i)$  elements of the covariance matrix are 0.

Data Types: `single` | `double`

**ParameterIsFixed** — Logical flag for fixed parameters

array of logical values

This property is read-only.

Logical flag for fixed parameters, specified as an array of logical values. If 0, the corresponding parameter in the `ParameterNames` array is not fixed. If 1, the corresponding parameter in the `ParameterNames` array is fixed.

Data Types: `logical`

**ParameterValues** — Distribution parameter values

vector of scalar values

This property is read-only.

Distribution parameter values, specified as a vector.

Data Types: `single` | `double`

**Truncation** — Truncation interval

vector of scalar values

This property is read-only.

Truncation interval for the probability distribution, specified as a vector containing the lower and upper truncation boundaries.

Data Types: `single` | `double`

## Other Object Properties

**DistributionName** — Probability distribution name

character vector

This property is read-only.

Probability distribution name, specified as a character vector.

Data Types: `char`

**InputData** — Data used for distribution fitting

structure

This property is read-only.

Data used for distribution fitting, specified as a structure containing the following:

- `data`: Data vector used for distribution fitting.
- `cens`: Censoring vector, or empty if none.
- `freq`: Frequency vector, or empty if none.

Data Types: `struct`

**ParameterDescription** — Distribution parameter descriptions

cell array of character vectors

This property is read-only.



Distribution parameter descriptions, specified as a cell array of character vectors. Each cell contains a short description of one distribution parameter.

Data Types: `char`

#### **ParameterNames** — Distribution parameter names

cell array of character vectors

This property is read-only.

Distribution parameter names, specified as a cell array of character vectors.

Data Types: `char`

## Object Functions

<code>cdf</code>	Cumulative distribution functions
<code>icdf</code>	Inverse cumulative distribution functions
<code>iqr</code>	Interquartile range
<code>mean</code>	Mean of probability distribution
<code>median</code>	Median of probability distribution
<code>negloglik</code>	Negative log likelihood of probability distribution
<code>paramci</code>	Confidence intervals for probability distribution parameters
<code>pdf</code>	Probability density functions
<code>proflik</code>	Profile likelihood function for probability distribution
<code>random</code>	Random numbers
<code>std</code>	Standard deviation of probability distribution
<code>truncate</code>	Truncate probability distribution object
<code>var</code>	Variance of probability distribution

## Examples

### Create a Generalized Pareto Distribution Object Using Default Parameters

Create a generalized Pareto distribution object using the default parameter values.

```
pd = makedist('GeneralizedPareto')
```

```
pd =
```

```
GeneralizedParetoDistribution

Generalized Pareto distribution
    k = 1
    sigma = 1
    theta = 1
```

### Create a Generalized Pareto Distribution Object Using Specified Parameters

Create a generalized Pareto distribution object by specifying parameter values.

```
pd = makedist('GeneralizedPareto', 'k', 0, 'sigma', 2, 'theta', 1)
```

```
pd =

GeneralizedParetoDistribution

Generalized Pareto distribution
    k = 0
    sigma = 2
    theta = 1
```

Compute the mean of the distribution.

```
m = mean(pd)
```

```
m =

    3
```

## See Also

**Distribution Fitter** | `fitdist` | `makedist`

## Topics

“Generalized Pareto Distribution” on page B-58

“Nonparametric and Empirical Probability Distributions” on page 5-37

Introduced in R2013a

# HalfNormalDistribution

Half-normal probability distribution object

## Description

A `HalfNormalDistribution` object consists of parameters, a model description, and sample data for a half-normal probability distribution.

The half-normal distribution is a special case of the folded normal and truncated normal distribution. Applications of the half-normal distribution include modeling measurement data and lifetime data.

The half-normal distribution uses the following parameters:

Parameter	Description	Support
<code>mu</code>	Location	$-\infty < \mu < \infty$
<code>sigma</code>	Scale	$\sigma \geq 0$

For more information about the half-normal distribution, see “Half-Normal Distribution” on page B-73.

## Creation

There are several ways to create a `HalfNormalDistribution` probability distribution object.

- Create a distribution with specified parameter values using `makedist`.
- Fit a distribution to data using `fitdist`.
- Interactively fit a distribution to data using the **Distribution Fitter** app.

## Properties

### Distribution Parameters

**mu** — Location parameter

scalar value

Location parameter of the half-normal distribution, specified as a scalar value. The `mu` parameter is also the lower limit of the half-normal distribution.

The Statistics and Machine Learning Toolbox implementation of the half-normal distribution assumes a fixed value for the location parameter  $\mu$ . You can specify a value for the  $\mu$  parameter when creating a `HalfNormalDistribution` object.

Data Types: `single` | `double`**sigma** — Scale parameter

nonnegative scalar value

Scale parameter of the half-normal distribution, specified as a nonnegative scalar value.

Data Types: `single` | `double`

### Distribution Characteristics

**IsTruncated** — Logical flag for truncated distribution

0 | 1

This property is read-only.

Logical flag for truncated distribution, specified as a logical value. If `IsTruncated` equals 0, the distribution is not truncated. If `IsTruncated` equals 1, the distribution is truncated.

Data Types: `logical`**NumParameters** — Number of parameters

positive integer value

This property is read-only.

Number of parameters for the probability distribution, specified as a positive integer value.

Data Types: `single` | `double`

**ParameterCovariance** — Covariance matrix of the parameter estimates

matrix of scalar values

This property is read-only.

Covariance matrix of the parameter estimates, specified as a  $p$ -by- $p$  matrix, where  $p$  is the number of parameters in the distribution. The  $(i,j)$  element is the covariance between the estimates of the  $i$ th parameter and the  $j$ th parameter. The  $(i,i)$  element is the estimated variance of the  $i$ th parameter. If parameter  $i$  is fixed rather than estimated by fitting the distribution to data, then the  $(i,i)$  elements of the covariance matrix are 0.

Data Types: `single` | `double`

**ParameterIsFixed** — Logical flag for fixed parameters

array of logical values

This property is read-only.

Logical flag for fixed parameters, specified as an array of logical values. If 0, the corresponding parameter in the `ParameterNames` array is not fixed. If 1, the corresponding parameter in the `ParameterNames` array is fixed.

Data Types: `logical`

**ParameterValues** — Distribution parameter values

vector of scalar values

This property is read-only.

Distribution parameter values, specified as a vector.

Data Types: `single` | `double`

**Truncation** — Truncation interval

vector of scalar values

This property is read-only.

Truncation interval for the probability distribution, specified as a vector containing the lower and upper truncation boundaries.

Data Types: `single` | `double`

## Other Object Properties

### **DistributionName** — Probability distribution name

character vector

This property is read-only.

Probability distribution name, specified as a character vector.

Data Types: `char`

### **InputData** — Data used for distribution fitting

structure

This property is read-only.

Data used for distribution fitting, specified as a structure containing the following:

- `data`: Data vector used for distribution fitting.
- `cens`: Censoring vector, or empty if none.
- `freq`: Frequency vector, or empty if none.

Data Types: `struct`

### **ParameterDescription** — Distribution parameter descriptions

cell array of character vectors

This property is read-only.

Distribution parameter descriptions, specified as a cell array of character vectors. Each cell contains a short description of one distribution parameter.

Data Types: `char`

### **ParameterNames** — Distribution parameter names

cell array of character vectors

This property is read-only.

Distribution parameter names, specified as a cell array of character vectors.

Data Types: `char`

## Object Functions

<code>cdf</code>	Cumulative distribution functions
<code>icdf</code>	Inverse cumulative distribution functions
<code>iqr</code>	Interquartile range
<code>mean</code>	Mean of probability distribution
<code>median</code>	Median of probability distribution
<code>negloglik</code>	Negative log likelihood of probability distribution
<code>paramci</code>	Confidence intervals for probability distribution parameters
<code>pdf</code>	Probability density functions
<code>proflik</code>	Profile likelihood function for probability distribution
<code>random</code>	Random numbers
<code>std</code>	Standard deviation of probability distribution
<code>truncate</code>	Truncate probability distribution object
<code>var</code>	Variance of probability distribution

## Examples

### Create a Half-Normal Distribution Object Using Default Parameters

```
pd = makedist('HalfNormal')
```

```
pd =
```

```
HalfNormalDistribution
```

```
Half Normal distribution
```

```
mu = 0
```

```
sigma = 1
```



### Create a Half-Normal Distribution Object Using Specified Parameters

Create a half-normal distribution object. Specify `mu` equal to 0 and `sigma` equal to 1.5.

```
pd = makedist('HalfNormal','mu',0,'sigma',1.5)
```

```
pd =
```

```
HalfNormalDistribution  
  
Half Normal distribution  
    mu = 0  
    sigma = 1.5
```

Compute the mean and standard deviation of the distribution.

```
m = mean(pd)  
s = std(pd)
```

```
m =
```

```
1.1968
```

```
s =
```

```
0.9042
```

### Fit a Half-Normal Distribution Object

Generate 100 random numbers from a standard normal distribution and compute their absolute value.

```
rng default % For reproducibility  
x = abs(random(makedist('Normal'),100,1));
```

Fit a half-normal distribution object to the sample data.

```
pd = fitdist(x,'HalfNormal')
```

```
pd =  
  
HalfNormalDistribution  
  
Half Normal distribution  
mu = 0  
sigma = 1.1631 [1.02184, 1.35006]
```

Calculate the mean of the fitted half-normal distribution using the probability distribution object.

```
m = mean(pd)
```

```
m =  
  
0.9280
```

Calculate the mean of the half-normal distribution by substituting the fitted `mu` and `sigma` parameter values into the formula

$$mean = \mu + \sigma \sqrt{\frac{2}{\pi}}.$$

```
mcalc = pd.mu + pd.sigma*(sqrt(2/pi))
```

```
mcalc =  
  
0.9280
```

## References

- [1] Cooray, K. and M.M.A. Ananda. “A Generalization of the Half-Normal Distribution with Applications to Lifetime Data.” *Communications in Statistics – Theory and Methods*. Vol. 37, Number 9, 2008, pp. 1323–1337.
- [2] Pewsey, A. *Large-Sample Inference for the General Half-Normal Distribution*. *Communications in Statistics – Theory and Methods*. Vol. 31, Number 7, 2002, pp. 1045–1054.

## See Also

**Distribution Fitter** | `fitdist` | `makedist`

## Topics

“Half-Normal Distribution” on page B-73

**Introduced in R2016a**

# InverseGaussianDistribution

Inverse Gaussian probability distribution object

## Description

An `InverseGaussianDistribution` object consists of parameters, a model description, and sample data for an inverse Gaussian probability distribution.

Also known as the Wald distribution, the inverse Gaussian is used to model nonnegative positively skewed data. Inverse Gaussian distributions have many similarities to standard Gaussian (normal) distributions, which lead to applications in inferential statistics.

The inverse Gaussian distribution uses the following parameters.

Parameter	Description	Support
<code>mu</code>	Scale parameter	$\mu > 0$
<code>lambda</code>	Shape parameter	$\lambda > 0$

## Creation

There are several ways to create a `InverseGaussianDistribution` probability distribution object.

- Create a distribution with specified parameter values using `makedist`.
- Fit a distribution to data using `fitdist`.
- Interactively fit a distribution to data using the **Distribution Fitter** app.

## Properties

### Distribution Parameters

**mu** — Scale parameter

positive scalar value

Scale parameter for the inverse Gaussian distribution, specified as a positive scalar value.

Data Types: `single` | `double`**lambda** — Shape parameter

positive scalar value

Shape parameter for the inverse Gaussian distribution, specified as a positive scalar value.

Data Types: `single` | `double`

### Distribution Characteristics

**IsTruncated** — Logical flag for truncated distribution

0 | 1

This property is read-only.

Logical flag for truncated distribution, specified as a logical value. If `IsTruncated` equals 0, the distribution is not truncated. If `IsTruncated` equals 1, the distribution is truncated.

Data Types: `logical`**NumParameters** — Number of parameters

positive integer value

This property is read-only.

Number of parameters for the probability distribution, specified as a positive integer value.

Data Types: `single` | `double`

**ParameterCovariance** — Covariance matrix of the parameter estimates

matrix of scalar values

This property is read-only.

Covariance matrix of the parameter estimates, specified as a  $p$ -by- $p$  matrix, where  $p$  is the number of parameters in the distribution. The  $(i,j)$  element is the covariance between the estimates of the  $i$ th parameter and the  $j$ th parameter. The  $(i,i)$  element is the estimated variance of the  $i$ th parameter. If parameter  $i$  is fixed rather than estimated by fitting the distribution to data, then the  $(i,i)$  elements of the covariance matrix are 0.

Data Types: `single` | `double`

**ParameterIsFixed** — Logical flag for fixed parameters

array of logical values

This property is read-only.

Logical flag for fixed parameters, specified as an array of logical values. If 0, the corresponding parameter in the `ParameterNames` array is not fixed. If 1, the corresponding parameter in the `ParameterNames` array is fixed.

Data Types: `logical`

**ParameterValues** — Distribution parameter values

vector of scalar values

This property is read-only.

Distribution parameter values, specified as a vector.

Data Types: `single` | `double`

**Truncation** — Truncation interval

vector of scalar values

This property is read-only.

Truncation interval for the probability distribution, specified as a vector containing the lower and upper truncation boundaries.

Data Types: `single` | `double`

## Other Object Properties

### **DistributionName** — Probability distribution name

character vector

This property is read-only.

Probability distribution name, specified as a character vector.

Data Types: `char`

### **InputData** — Data used for distribution fitting

structure

This property is read-only.

Data used for distribution fitting, specified as a structure containing the following:

- `data`: Data vector used for distribution fitting.
- `cens`: Censoring vector, or empty if none.
- `freq`: Frequency vector, or empty if none.

Data Types: `struct`

### **ParameterDescription** — Distribution parameter descriptions

cell array of character vectors

This property is read-only.

Distribution parameter descriptions, specified as a cell array of character vectors. Each cell contains a short description of one distribution parameter.

Data Types: `char`

### **ParameterNames** — Distribution parameter names

cell array of character vectors

This property is read-only.

Distribution parameter names, specified as a cell array of character vectors.

Data Types: `char`

## Object Functions

<code>cdf</code>	Cumulative distribution functions
<code>icdf</code>	Inverse cumulative distribution functions
<code>iqr</code>	Interquartile range
<code>mean</code>	Mean of probability distribution
<code>median</code>	Median of probability distribution
<code>negloglik</code>	Negative log likelihood of probability distribution
<code>paramci</code>	Confidence intervals for probability distribution parameters
<code>pdf</code>	Probability density functions
<code>proflik</code>	Profile likelihood function for probability distribution
<code>random</code>	Random numbers
<code>std</code>	Standard deviation of probability distribution
<code>truncate</code>	Truncate probability distribution object
<code>var</code>	Variance of probability distribution

## Examples

### Create an Inverse Gaussian Distribution Object Using Default Parameters

Create an inverse Gaussian distribution object using the default parameter values.

```
pd = makedist('InverseGaussian')
```

```
pd =
```

```
InverseGaussianDistribution
```

```
Inverse Gaussian distribution
```

```
mu = 1
```

```
lambda = 1
```

### Create an Inverse Gaussian Distribution Object Using Specified Parameters

Create an inverse Gaussian distribution object by specifying parameter values.

```
pd = makedist('InverseGaussian', 'mu', 2, 'lambda', 4)
```

```
pd =
```



```
InverseGaussianDistribution  
  
Inverse Gaussian distribution  
  mu = 2  
  lambda = 4
```

Compute the standard deviation of the distribution.

```
s = std(pd)
```

```
s =
```

```
  1.4142
```

## See Also

**Distribution Fitter** | `fitdist` | `makedist`

## Topics

“Inverse Gaussian Distribution” on page B-82

**Introduced in R2013a**

# KernelDistribution

Kernel probability distribution object

## Description

A `KernelDistribution` object consists of parameters, a model description, and sample data for a nonparametric kernel-smoothing distribution.

The kernel distribution is a nonparametric estimation of the probability density function (pdf) of a random variable.

The kernel distribution uses the following options.

Option	Description	Possible Values
Kernel	Kernel function type	normal, box, triangle, epanechnikov
BandWidth	Kernel smoothing parameter	BandWidth > 0

## Creation

There are several ways to create a `KernelDistribution` probability distribution object.

- Fit a distribution to data using `fitdist`.
- Interactively fit a distribution to data using the **Distribution Fitter** app.

## Properties

### Distribution Parameters

**Kernel** — Kernel smoother type

```
'normal' | 'box' | 'triangle' | 'epanechnikov'
```

Kernel function type, specified as a valid kernel function type name.

**Bandwidth** — Bandwidth of kernel smoothing window

positive scalar value

Bandwidth of the kernel smoothing window, specified as a positive scalar value.

Data Types: `single` | `double`

## Distribution Characteristics

**IsTruncated** — Logical flag for truncated distribution

0 | 1

This property is read-only.

Logical flag for truncated distribution, specified as a logical value. If `IsTruncated` equals 0, the distribution is not truncated. If `IsTruncated` equals 1, the distribution is truncated.

Data Types: `logical`**Truncation** — Truncation interval

vector of scalar values

This property is read-only.

Truncation interval for the probability distribution, specified as a vector containing the lower and upper truncation boundaries.

Data Types: `single` | `double`

## Other Object Properties

**DistributionName** — Probability distribution name

character vector

This property is read-only.

Probability distribution name, specified as a character vector.

Data Types: `char`**InputData** — Data used for distribution fitting

structure

This property is read-only.

Data used for distribution fitting, specified as a structure containing the following:

- `data`: Data vector used for distribution fitting.
- `cens`: Censoring vector, or empty if none.
- `freq`: Frequency vector, or empty if none.

Data Types: `struct`

## Object Functions

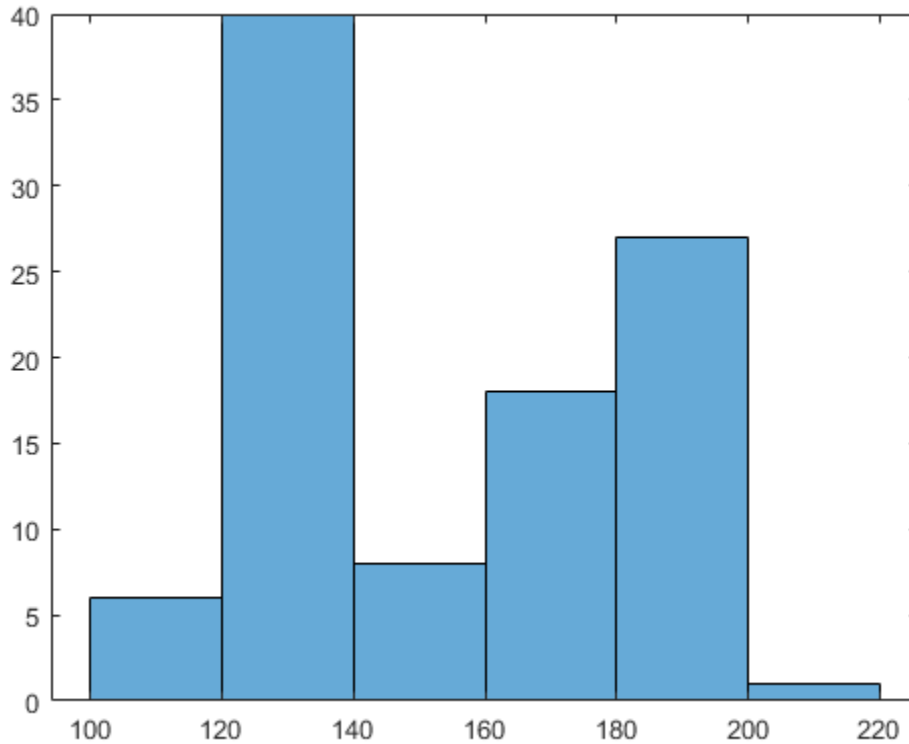
<code>cdf</code>	Cumulative distribution functions
<code>icdf</code>	Inverse cumulative distribution functions
<code>iqr</code>	Interquartile range
<code>mean</code>	Mean of probability distribution
<code>median</code>	Median of probability distribution
<code>negloglik</code>	Negative log likelihood of probability distribution
<code>pdf</code>	Probability density functions
<code>random</code>	Random numbers
<code>std</code>	Standard deviation of probability distribution
<code>truncate</code>	Truncate probability distribution object
<code>var</code>	Variance of probability distribution

## Examples

### Fit a Kernel Distribution Object to Data

Load the sample data. Visualize the patient weight data using a histogram.

```
load hospital
histogram(hospital.Weight)
```



The histogram shows that the data has two modes, one for female patients and one for male patients.

Create a probability distribution object by fitting a kernel distribution to the patient weight data.

```
pd_kernel = fitdist(hospital.Weight, 'Kernel')
```

```
pd_kernel =
```

```
KernelDistribution
```

```
Kernel = normal
```

```
Bandwidth = 14.3792  
Support = unbounded
```

For comparison, create another probability distribution object by fitting a normal distribution to the patient weight data.

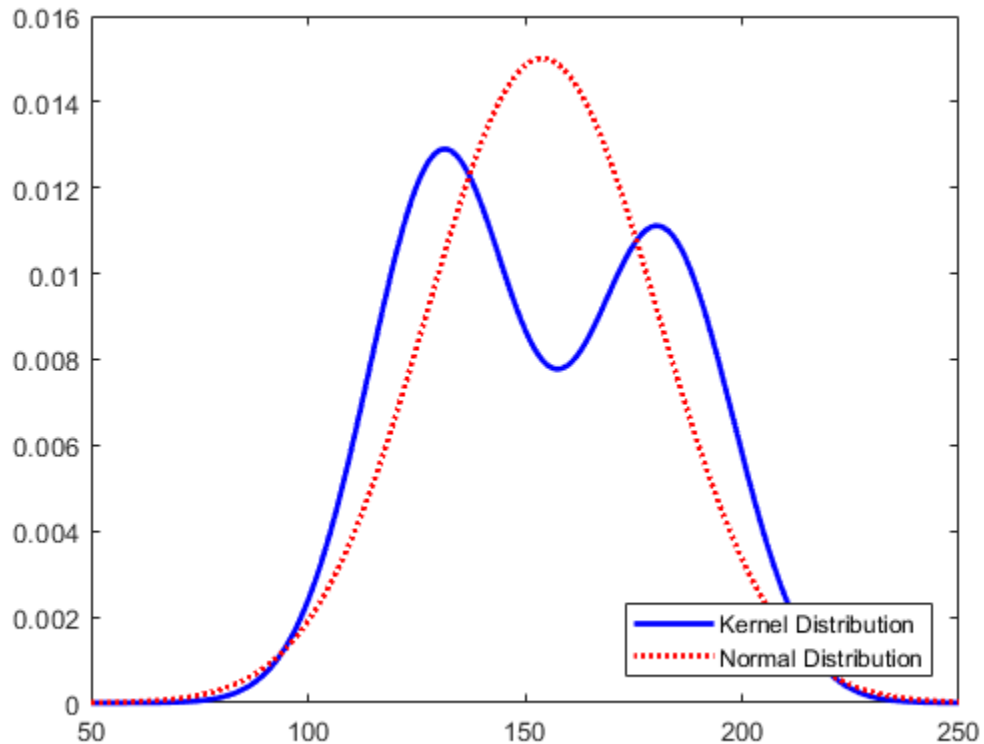
```
pd_normal = fitdist(hospital.Weight, 'Normal')  
  
pd_normal =  
  
NormalDistribution  
  
Normal distribution  
mu = 154 [148.728, 159.272]  
sigma = 26.5714 [23.3299, 30.8674]
```

Define the x values and compute the pdf of each distribution.

```
x = 50:1:250;  
pdf_kernel = pdf(pd_kernel, x);  
pdf_normal = pdf(pd_normal, x);
```

Plot the pdf of each distribution.

```
plot(x, pdf_kernel, 'Color', 'b', 'LineWidth', 2);  
hold on;  
plot(x, pdf_normal, 'Color', 'r', 'LineStyle', ':', 'LineWidth', 2);  
legend('Kernel Distribution', 'Normal Distribution', 'Location', 'SouthEast');  
hold off;
```



Fitting a kernel distribution instead of a unimodal distribution such as the normal reveals the separate modes for the female and male patients.

- “Fit Kernel Distribution Object to Data” on page 5-46
- “Fit Probability Distribution Objects to Grouped Data” on page 5-121
- “Compare Multiple Distribution Fits” on page 5-114

## See Also

**Distribution Fitter** | `fitdist`

## **Topics**

“Fit Kernel Distribution Object to Data” on page 5-46

“Fit Probability Distribution Objects to Grouped Data” on page 5-121

“Compare Multiple Distribution Fits” on page 5-114

“Kernel Distribution” on page B-86

“Nonparametric and Empirical Probability Distributions” on page 5-37

**Introduced in R2013a**



# LogisticDistribution

Logistic probability distribution object

## Description

A `LogisticDistribution` object consists of parameters, a model description, and sample data for a logistic probability distribution.

The logistic distribution is used for growth models and in logistic regression. It has longer tails and a higher kurtosis than the normal distribution.

The logistic distribution uses the following parameters.

Parameter	Description	Support
<code>mu</code>	Mean	$-\infty < \mu < \infty$
<code>sigma</code>	Scale parameter	$\sigma \geq 0$

## Creation

There are several ways to create a `LogisticDistribution` probability distribution object.

- Create a distribution with specified parameter values using `makedist`.
- Fit a distribution to data using `fitdist`.
- Interactively fit a distribution to data using the **Distribution Fitter** app.

## Properties

### Distribution Parameters

**mu** — Mean  
scalar value

Mean of the logistic distribution, specified as a scalar value.

Data Types: `single` | `double`

**sigma** — Scale parameter

nonnegative scalar value

Scale parameter of the logistic distribution, specified as a nonnegative scalar value.

Data Types: `single` | `double`

## Distribution Characteristics

**IsTruncated** — Logical flag for truncated distribution

0 | 1

This property is read-only.

Logical flag for truncated distribution, specified as a logical value. If `IsTruncated` equals 0, the distribution is not truncated. If `IsTruncated` equals 1, the distribution is truncated.

Data Types: `logical`

**NumParameters** — Number of parameters

positive integer value

This property is read-only.

Number of parameters for the probability distribution, specified as a positive integer value.

Data Types: `single` | `double`

**ParameterCovariance** — Covariance matrix of the parameter estimates

matrix of scalar values

This property is read-only.

Covariance matrix of the parameter estimates, specified as a  $p$ -by- $p$  matrix, where  $p$  is the number of parameters in the distribution. The  $(i,j)$  element is the covariance between the estimates of the  $i$ th parameter and the  $j$ th parameter. The  $(i,i)$  element is the estimated variance of the  $i$ th parameter. If parameter  $i$  is fixed rather than

estimated by fitting the distribution to data, then the  $(i,i)$  elements of the covariance matrix are 0.

Data Types: `single` | `double`

**ParameterIsFixed** — Logical flag for fixed parameters

array of logical values

This property is read-only.

Logical flag for fixed parameters, specified as an array of logical values. If 0, the corresponding parameter in the `ParameterNames` array is not fixed. If 1, the corresponding parameter in the `ParameterNames` array is fixed.

Data Types: `logical`

**ParameterValues** — Distribution parameter values

vector of scalar values

This property is read-only.

Distribution parameter values, specified as a vector.

Data Types: `single` | `double`

**Truncation** — Truncation interval

vector of scalar values

This property is read-only.

Truncation interval for the probability distribution, specified as a vector containing the lower and upper truncation boundaries.

Data Types: `single` | `double`

## Other Object Properties

**DistributionName** — Probability distribution name

character vector

This property is read-only.

Probability distribution name, specified as a character vector.

Data Types: `char`

**InputData** — Data used for distribution fitting  
structure

This property is read-only.

Data used for distribution fitting, specified as a structure containing the following:

- `data`: Data vector used for distribution fitting.
- `cens`: Censoring vector, or empty if none.
- `freq`: Frequency vector, or empty if none.

Data Types: `struct`

**ParameterDescription** — Distribution parameter descriptions  
cell array of character vectors

This property is read-only.

Distribution parameter descriptions, specified as a cell array of character vectors. Each cell contains a short description of one distribution parameter.

Data Types: `char`

**ParameterNames** — Distribution parameter names  
cell array of character vectors

This property is read-only.

Distribution parameter names, specified as a cell array of character vectors.

Data Types: `char`

## Object Functions

<code>cdf</code>	Cumulative distribution functions
<code>icdf</code>	Inverse cumulative distribution functions
<code>iqr</code>	Interquartile range
<code>mean</code>	Mean of probability distribution
<code>median</code>	Median of probability distribution

negloglik	Negative log likelihood of probability distribution
paramci	Confidence intervals for probability distribution parameters
pdf	Probability density functions
proflik	Profile likelihood function for probability distribution
random	Random numbers
std	Standard deviation of probability distribution
truncate	Truncate probability distribution object
var	Variance of probability distribution

## Examples

### Create a Logistic Distribution Object Using Default Parameters

Create a logistic distribution object using the default parameter values.

```
pd = makedist('Logistic')
```

```
pd =  
  
LogisticDistribution  
  
Logistic distribution  
  mu = 0  
  sigma = 1
```

### Create a Logistic Distribution Object Using Specified Parameters

Create a logistic distribution object by specifying parameter values.

```
pd = makedist('Logistic', 'mu', 2, 'sigma', 4)
```

```
pd =  
  
LogisticDistribution  
  
Logistic distribution  
  mu = 2  
  sigma = 4
```

Compute the standard deviation of the distribution.

```
s = std(pd)
```

```
s =
```

```
7.2552
```

- “Compare Multiple Distribution Fits” on page 5-114

## See Also

**Distribution Fitter** | `fitdist` | `makedist`

## Topics

“Compare Multiple Distribution Fits” on page 5-114

“Logistic Distribution” on page B-96

**Introduced in R2013a**

# LoglogisticDistribution

Loglogistic probability distribution object

## Description

A `LoglogisticDistribution` object consists of parameters, a model description, and sample data for a loglogistic probability distribution.

The loglogistic distribution is closely related to the logistic distribution. If  $x$  is distributed loglogistically with parameters  $\mu$  and  $\sigma$ , then  $\log(x)$  is distributed logistically with mean and standard deviation. This distribution is often used in survival analysis to model events that experience an initial rate increase, followed by a rate decrease.

The loglogistic distribution uses the following parameters.

Parameter	Description	Support
mu	Log mean	$\mu > 0$
sigma	Log scale parameter	$\sigma > 0$

## Creation

There are several ways to create a `LoglogisticDistribution` probability distribution object.

- Create a distribution with specified parameter values using `makedist`.
- Fit a distribution to data using `fitdist`.
- Interactively fit a distribution to data using the **Distribution Fitter** app.

## Properties

### Distribution Parameters

**mu — Log mean**

positive scalar value

Log mean for the loglogistic distribution, specified as a positive scalar value.

Data Types: `single` | `double`

**sigma — Log scale parameter**

positive scalar value

Log scale parameter for the loglogistic distribution, specified as a positive scalar value.

Data Types: `single` | `double`

### Distribution Characteristics

**IsTruncated — Logical flag for truncated distribution**

0 | 1

This property is read-only.

Logical flag for truncated distribution, specified as a logical value. If `IsTruncated` equals 0, the distribution is not truncated. If `IsTruncated` equals 1, the distribution is truncated.

Data Types: `logical`

**NumParameters — Number of parameters**

positive integer value

This property is read-only.

Number of parameters for the probability distribution, specified as a positive integer value.

Data Types: `single` | `double`



**ParameterCovariance** — Covariance matrix of the parameter estimates

matrix of scalar values

This property is read-only.

Covariance matrix of the parameter estimates, specified as a  $p$ -by- $p$  matrix, where  $p$  is the number of parameters in the distribution. The  $(i,j)$  element is the covariance between the estimates of the  $i$ th parameter and the  $j$ th parameter. The  $(i,i)$  element is the estimated variance of the  $i$ th parameter. If parameter  $i$  is fixed rather than estimated by fitting the distribution to data, then the  $(i,i)$  elements of the covariance matrix are 0.

Data Types: `single` | `double`

**ParameterIsFixed** — Logical flag for fixed parameters

array of logical values

This property is read-only.

Logical flag for fixed parameters, specified as an array of logical values. If 0, the corresponding parameter in the `ParameterNames` array is not fixed. If 1, the corresponding parameter in the `ParameterNames` array is fixed.

Data Types: `logical`

**ParameterValues** — Distribution parameter values

vector of scalar values

This property is read-only.

Distribution parameter values, specified as a vector.

Data Types: `single` | `double`

**Truncation** — Truncation interval

vector of scalar values

This property is read-only.

Truncation interval for the probability distribution, specified as a vector containing the lower and upper truncation boundaries.

Data Types: `single` | `double`

## Other Object Properties

### **DistributionName** — Probability distribution name

character vector

This property is read-only.

Probability distribution name, specified as a character vector.

Data Types: `char`

### **InputData** — Data used for distribution fitting

structure

This property is read-only.

Data used for distribution fitting, specified as a structure containing the following:

- `data`: Data vector used for distribution fitting.
- `cens`: Censoring vector, or empty if none.
- `freq`: Frequency vector, or empty if none.

Data Types: `struct`

### **ParameterDescription** — Distribution parameter descriptions

cell array of character vectors

This property is read-only.

Distribution parameter descriptions, specified as a cell array of character vectors. Each cell contains a short description of one distribution parameter.

Data Types: `char`

### **ParameterNames** — Distribution parameter names

cell array of character vectors

This property is read-only.

Distribution parameter names, specified as a cell array of character vectors.

Data Types: `char`

## Object Functions

<code>cdf</code>	Cumulative distribution functions
<code>icdf</code>	Inverse cumulative distribution functions
<code>iqr</code>	Interquartile range
<code>mean</code>	Mean of probability distribution
<code>median</code>	Median of probability distribution
<code>negloglik</code>	Negative log likelihood of probability distribution
<code>paramci</code>	Confidence intervals for probability distribution parameters
<code>pdf</code>	Probability density functions
<code>proflik</code>	Profile likelihood function for probability distribution
<code>random</code>	Random numbers
<code>std</code>	Standard deviation of probability distribution
<code>truncate</code>	Truncate probability distribution object
<code>var</code>	Variance of probability distribution

## Examples

### Create a Loglogistic Distribution Object Using Default Parameters

Create a loglogistic distribution object using the default parameter values.

```
pd = makedist('Loglogistic')
```

```
pd =
```

```
LoglogisticDistribution  
  
Log-Logistic distribution  
  mu = 0  
  sigma = 1
```

### Create a Loglogistic Distribution Object Using Specified Parameters

Create a loglogistic distribution object by specifying the parameter values.

```
pd = makedist('Loglogistic', 'mu', 5, 'sigma', 2)
```

```
pd =
```

```
LoglogisticDistribution  
  
Log-Logistic distribution  
    mu = 5  
    sigma = 2
```

Generate random numbers from the loglogistic distribution and compute their log values.

```
rng(19) % for reproducibility  
x = random(pd,10000,1);  
logx = log(x);
```

Compute the mean of the log values.

```
m = mean(logx)
```

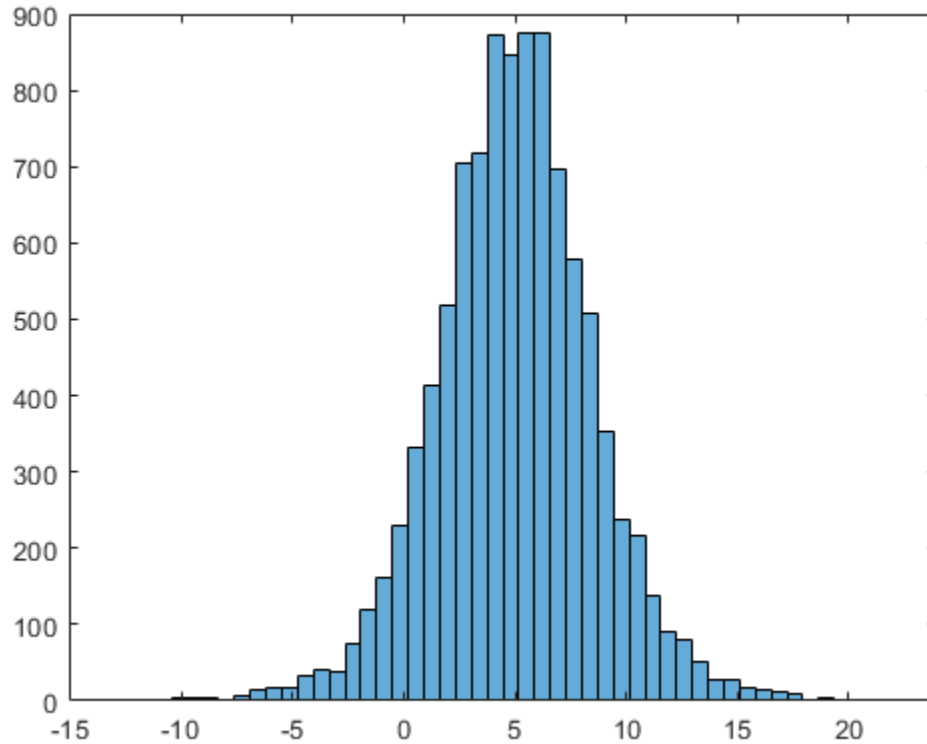
```
m =
```

```
4.9828
```

The mean of the log of  $x$  is equal to the  $\mu$  parameter of  $x$ , since  $x$  has a loglogistic distribution.

Plot  $\log x$ .

```
histogram(logx,50)
```



The plot shows that the log values of  $x$  have a logistic distribution.

## See Also

**Distribution Fitter** | `fitdist` | `makedist`

## Topics

“Loglogistic Distribution” on page B-98

Introduced in R2013a

# LognormalDistribution

Lognormal probability distribution object

## Description

A `LognormalDistribution` object consists of parameters, a model description, and sample data for a lognormal probability distribution.

The lognormal distribution is a probability distribution whose logarithm has a normal distribution. It is sometimes called the Galton distribution. The lognormal distribution is applicable when the quantity of interest must be positive, since  $\log(x)$  exists only when  $x$  is positive.

The lognormal distribution uses the following parameters.

Parameter	Description	Support
<code>mu</code>	Log mean	$-\infty < \mu < \infty$
<code>sigma</code>	Log standard deviation	$\sigma \geq 0$

## Creation

There are several ways to create a `LognormalDistribution` probability distribution object.

- Create a distribution with specified parameter values using `makedist`.
- Fit a distribution to data using `fitdist`.
- Interactively fit a distribution to data using the **Distribution Fitter** app.

## Properties

### Distribution Parameters

**mu — Log mean**

scalar value

Log mean for the lognormal distribution, specified as a scalar value.

Data Types: `single` | `double`

**sigma — Log standard deviation**

nonnegative scalar value

Log standard deviation for the lognormal distribution, specified as a nonnegative scalar value.

Data Types: `single` | `double`

### Distribution Characteristics

**IsTruncated — Logical flag for truncated distribution**

0 | 1

This property is read-only.

Logical flag for truncated distribution, specified as a logical value. If `IsTruncated` equals 0, the distribution is not truncated. If `IsTruncated` equals 1, the distribution is truncated.

Data Types: `logical`

**NumParameters — Number of parameters**

positive integer value

This property is read-only.

Number of parameters for the probability distribution, specified as a positive integer value.

Data Types: `single` | `double`

**ParameterCovariance** — Covariance matrix of the parameter estimates

matrix of scalar values

This property is read-only.

Covariance matrix of the parameter estimates, specified as a  $p$ -by- $p$  matrix, where  $p$  is the number of parameters in the distribution. The  $(i,j)$  element is the covariance between the estimates of the  $i$ th parameter and the  $j$ th parameter. The  $(i,i)$  element is the estimated variance of the  $i$ th parameter. If parameter  $i$  is fixed rather than estimated by fitting the distribution to data, then the  $(i,i)$  elements of the covariance matrix are 0.

Data Types: `single` | `double`

**ParameterIsFixed** — Logical flag for fixed parameters

array of logical values

This property is read-only.

Logical flag for fixed parameters, specified as an array of logical values. If 0, the corresponding parameter in the `ParameterNames` array is not fixed. If 1, the corresponding parameter in the `ParameterNames` array is fixed.

Data Types: `logical`

**ParameterValues** — Distribution parameter values

vector of scalar values

This property is read-only.

Distribution parameter values, specified as a vector.

Data Types: `single` | `double`

**Truncation** — Truncation interval

vector of scalar values

This property is read-only.

Truncation interval for the probability distribution, specified as a vector containing the lower and upper truncation boundaries.

Data Types: `single` | `double`



## Other Object Properties

### **DistributionName** — Probability distribution name

character vector

This property is read-only.

Probability distribution name, specified as a character vector.

Data Types: `char`

### **InputData** — Data used for distribution fitting

structure

This property is read-only.

Data used for distribution fitting, specified as a structure containing the following:

- `data`: Data vector used for distribution fitting.
- `cens`: Censoring vector, or empty if none.
- `freq`: Frequency vector, or empty if none.

Data Types: `struct`

### **ParameterDescription** — Distribution parameter descriptions

cell array of character vectors

This property is read-only.

Distribution parameter descriptions, specified as a cell array of character vectors. Each cell contains a short description of one distribution parameter.

Data Types: `char`

### **ParameterNames** — Distribution parameter names

cell array of character vectors

This property is read-only.

Distribution parameter names, specified as a cell array of character vectors.

Data Types: `char`

## Object Functions

<code>cdf</code>	Cumulative distribution functions
<code>icdf</code>	Inverse cumulative distribution functions
<code>iqr</code>	Interquartile range
<code>mean</code>	Mean of probability distribution
<code>median</code>	Median of probability distribution
<code>negloglik</code>	Negative log likelihood of probability distribution
<code>paramci</code>	Confidence intervals for probability distribution parameters
<code>pdf</code>	Probability density functions
<code>proflik</code>	Profile likelihood function for probability distribution
<code>random</code>	Random numbers
<code>std</code>	Standard deviation of probability distribution
<code>truncate</code>	Truncate probability distribution object
<code>var</code>	Variance of probability distribution

## Examples

### Create a Lognormal Distribution Object Using Default Parameters

Create a lognormal distribution object using the default parameter values.

```
pd = makedist('Lognormal')
```

```
pd =  
  
LognormalDistribution  
  
Lognormal distribution  
  mu = 0  
  sigma = 1
```

### Create a Lognormal Distribution Object Using Specified Parameters

Create a lognormal distribution object by specifying the parameter values.

```
pd = makedist('Lognormal', 'mu', 5, 'sigma', 2)
```

```
pd =
```

```
LognormalDistribution  
  
Lognormal distribution  
    mu = 5  
    sigma = 2
```

Compute the mean of the lognormal distribution.

```
mean(pd)
```

```
ans =  
  
    1.0966e+03
```

The mean of the lognormal distribution is not equal to the mu parameter.

Generate random numbers from the lognormal distribution and compute their log values.

```
rng(47); % for reproducibility  
x = random(pd,10000,1);  
logx = log(x);
```

Compute the mean of the log values.

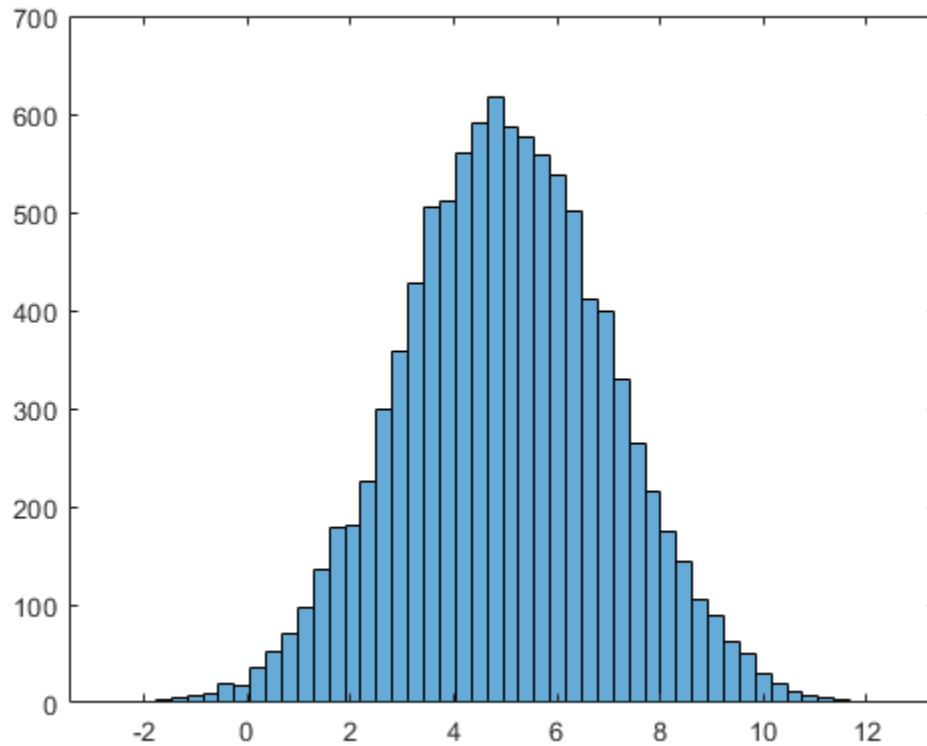
```
m = mean(logx)
```

```
m =  
  
    5.0156
```

The mean of the log of x is equal to the mu parameter of x, since x has a lognormal distribution.

Plot logx.

```
histogram(logx,50)
```



The plot shows that the log values of  $x$  are normally distributed with a mean equal to 5 and a standard deviation equal to 2.

## See Also

**Distribution Fitter** | `fitdist` | `makedist`

## Topics

“Lognormal Distribution” on page B-100

Introduced in R2013a

# MultinomialDistribution

Multinomial probability distribution object

## Description

A `MultinomialDistribution` object consists of parameters and a model description for a multinomial probability distribution.

The multinomial distribution is a generalization of the binomial distribution. While the binomial distribution gives the probability of the number of “successes” in  $n$  independent trials of a two-outcome process, the multinomial distribution gives the probability of each combination of outcomes in  $n$  independent trials of a  $k$ -outcome process. The probability of each outcome in any one trial is given by the fixed probabilities  $p_1, \dots, p_k$ .

The multinomial distribution uses the following parameters.

Parameter	Description	Support
probabilities	Outcome probabilities	$0 \leq \text{probabilities}(i) \leq 1$ ; $\sum_{\text{all}(i)} \text{probabilities}(i) = 1$

## Creation

Create a `MultinomialDistribution` probability distribution with specified parameter values object using `makedist`.

## Properties

### Distribution Parameter

**Probabilities** — outcome probabilities

vector of scalar values in the range  $[0, 1]$

Outcome probabilities for the multinomial distribution, stored as a vector of scalar values in the range  $[0, 1]$ . The values in `probabilities` must sum to 1.

Data Types: `single` | `double`

## Distribution Characteristics

### **IsTruncated** — Logical flag for truncated distribution

0 | 1

This property is read-only.

Logical flag for truncated distribution, specified as a logical value. If `IsTruncated` equals 0, the distribution is not truncated. If `IsTruncated` equals 1, the distribution is truncated.

Data Types: `logical`

### **NumParameters** — Number of parameters

positive integer value

This property is read-only.

Number of parameters for the probability distribution, specified as a positive integer value.

Data Types: `single` | `double`

### **ParameterValues** — Distribution parameter values

vector of scalar values

This property is read-only.

Distribution parameter values, specified as a vector.

Data Types: `single` | `double`

### **Truncation** — Truncation interval

vector of scalar values

This property is read-only.

Truncation interval for the probability distribution, specified as a vector containing the lower and upper truncation boundaries.

Data Types: `single` | `double`

## Other Object Properties

### **DistributionName** — Probability distribution name

character vector

This property is read-only.

Probability distribution name, specified as a character vector.

Data Types: `char`

### **ParameterDescription** — Distribution parameter descriptions

cell array of character vectors

This property is read-only.

Distribution parameter descriptions, specified as a cell array of character vectors. Each cell contains a short description of one distribution parameter.

Data Types: `char`

### **ParameterNames** — Distribution parameter names

cell array of character vectors

This property is read-only.

Distribution parameter names, specified as a cell array of character vectors.

Data Types: `char`

## Object Functions

<code>cdf</code>	Cumulative distribution functions
<code>icdf</code>	Inverse cumulative distribution functions
<code>iqr</code>	Interquartile range
<code>mean</code>	Mean of probability distribution
<code>median</code>	Median of probability distribution

pdf	Probability density functions
random	Random numbers
std	Standard deviation of probability distribution
truncate	Truncate probability distribution object
var	Variance of probability distribution

## Examples

### Create a Multinomial Distribution Object Using Default Parameters

Create a multinomial distribution object using the default parameter values.

```
pd = makedist('Multinomial')
```

```
pd =
```

```
  MultinomialDistribution
```

```
  Probabilities:
```

```
    0.5000    0.5000
```

### Create a Multinomial Distribution Object Using Specified Parameters

Create a multinomial distribution object for a distribution with three possible outcomes. Outcome 1 has a probability of 1/2, outcome 2 has a probability of 1/3, and outcome 3 has a probability of 1/6.

```
pd = makedist('Multinomial', 'probabilities', [1/2 1/3 1/6])
```

```
pd =
```

```
  MultinomialDistribution
```

```
  Probabilities:
```

```
    0.5000    0.3333    0.1667
```

Generate a random outcome from the distribution.



```
rng('default'); % for reproducibility
r = random(pd)
```

```
r =
     2
```

The result of this trial is outcome 2. By default, the number of trials in each experiment,  $n$ , equals 1.

Generate random outcomes from the distribution when the number of trials in each experiment,  $n$ , equals 1, and the experiment is repeated ten times.

```
rng('default'); % for reproducibility
r = random(pd,10,1)
```

```
r =
     2
     3
     1
     3
     2
     1
     1
     2
     3
     3
```

Each element in the array is the outcome of an individual experiment that contains one trial.

Generate random outcomes from the distribution when the number of trials in each experiment,  $n$ , equals 5, and the experiment is repeated ten times.

```
rng('default'); % for reproducibility
r = random(pd,10,5)
```

```
r =
```

2	1	2	2	1
3	3	1	1	1
1	3	3	1	2
3	1	3	1	2
2	2	2	1	1
1	1	2	2	1
1	1	2	2	1
2	3	1	1	2
3	2	2	3	2
3	3	1	1	2

Each element in the resulting matrix is the outcome of one trial. The columns correspond to the five trials in each experiment, and the rows correspond to the ten experiments. For example, in the first experiment (corresponding to the first row), 2 of the 5 trials resulted in outcome 1, and 3 of the 5 trials resulted in outcome 2.

- “Multinomial Probability Distribution Objects” on page 5-125
- “Multinomial Probability Distribution Functions” on page 5-129

## See Also

`makedist`

## Topics

- “Multinomial Probability Distribution Objects” on page 5-125
- “Multinomial Probability Distribution Functions” on page 5-129
- “Multinomial Distribution” on page B-103

Introduced in R2013a

# NakagamiDistribution

Nakagami probability distribution object

## Description

A `NakagamiDistribution` object consists of parameters, a model description, and sample data for a Nakagami probability distribution.

The Nakagami distribution is commonly used in communication theory to model scattered signals that reach a receiver using multiple paths.

The Nakagami distribution uses the following parameters.

Parameter	Description	Support
<code>mu</code>	Shape parameter	$\mu > 0$
<code>omega</code>	Scale parameter	$\omega > 0$

## Creation

There are several ways to create a `NakagamiDistribution` probability distribution object.

- Create a distribution with specified parameter values using `makedist`.
- Fit a distribution to data using `fitdist`.
- Interactively fit a distribution to data using the **Distribution Fitter** app.

## Properties

### Distribution Parameters

**mu** — Shape parameter  
positive scalar value

Shape parameter for the Nakagami distribution, specified as a positive scalar value.

Data Types: `single` | `double`

**omega** — Scale parameter

positive scalar value

Scale parameter for the Nakagami distribution, specified as a positive scalar value.

Data Types: `single` | `double`

## Distribution Characteristics

**IsTruncated** — Logical flag for truncated distribution

0 | 1

This property is read-only.

Logical flag for truncated distribution, specified as a logical value. If `IsTruncated` equals 0, the distribution is not truncated. If `IsTruncated` equals 1, the distribution is truncated.

Data Types: `logical`

**NumParameters** — Number of parameters

positive integer value

This property is read-only.

Number of parameters for the probability distribution, specified as a positive integer value.

Data Types: `single` | `double`

**ParameterCovariance** — Covariance matrix of the parameter estimates

matrix of scalar values

This property is read-only.

Covariance matrix of the parameter estimates, specified as a  $p$ -by- $p$  matrix, where  $p$  is the number of parameters in the distribution. The  $(i,j)$  element is the covariance between the estimates of the  $i$ th parameter and the  $j$ th parameter. The  $(i,i)$  element is the estimated variance of the  $i$ th parameter. If parameter  $i$  is fixed rather than

estimated by fitting the distribution to data, then the  $(i,i)$  elements of the covariance matrix are 0.

Data Types: `single` | `double`

**ParameterIsFixed** — Logical flag for fixed parameters

array of logical values

This property is read-only.

Logical flag for fixed parameters, specified as an array of logical values. If 0, the corresponding parameter in the `ParameterNames` array is not fixed. If 1, the corresponding parameter in the `ParameterNames` array is fixed.

Data Types: `logical`

**ParameterValues** — Distribution parameter values

vector of scalar values

This property is read-only.

Distribution parameter values, specified as a vector.

Data Types: `single` | `double`

**Truncation** — Truncation interval

vector of scalar values

This property is read-only.

Truncation interval for the probability distribution, specified as a vector containing the lower and upper truncation boundaries.

Data Types: `single` | `double`

## Other Object Properties

**DistributionName** — Probability distribution name

character vector

This property is read-only.

Probability distribution name, specified as a character vector.

Data Types: `char`

**InputData** — Data used for distribution fitting  
structure

This property is read-only.

Data used for distribution fitting, specified as a structure containing the following:

- `data`: Data vector used for distribution fitting.
- `cens`: Censoring vector, or empty if none.
- `freq`: Frequency vector, or empty if none.

Data Types: `struct`

**ParameterDescription** — Distribution parameter descriptions  
cell array of character vectors

This property is read-only.

Distribution parameter descriptions, specified as a cell array of character vectors. Each cell contains a short description of one distribution parameter.

Data Types: `char`

**ParameterNames** — Distribution parameter names  
cell array of character vectors

This property is read-only.

Distribution parameter names, specified as a cell array of character vectors.

Data Types: `char`

## Object Functions

<code>cdf</code>	Cumulative distribution functions
<code>icdf</code>	Inverse cumulative distribution functions
<code>iqr</code>	Interquartile range
<code>mean</code>	Mean of probability distribution
<code>median</code>	Median of probability distribution

---

negloglik	Negative log likelihood of probability distribution
paramci	Confidence intervals for probability distribution parameters
pdf	Probability density functions
proflik	Profile likelihood function for probability distribution
random	Random numbers
std	Standard deviation of probability distribution
truncate	Truncate probability distribution object
var	Variance of probability distribution

## Examples

### Create a Nakagami Distribution Object Using Default Parameters

Create a Nakagami distribution object using the default parameter values.

```
pd = makedist('Nakagami')
```

```
pd =  
  
NakagamiDistribution  
  
Nakagami distribution  
mu = 1  
omega = 1
```

### Create a Nakagami Distribution Object Using Specified Parameters

Create a Nakagami distribution object by specifying parameter values.

```
pd = makedist('Nakagami', 'mu', 5, 'omega', 2)
```

```
pd =  
  
NakagamiDistribution  
  
Nakagami distribution  
mu = 5  
omega = 2
```

Compute the mean of the distribution.

```
m = mean(pd)
```

```
m =
```

```
1.3794
```

## See Also

**Distribution Fitter** | `fitdist` | `makedist`

## Topics

“Nakagami Distribution” on page B-118

**Introduced in R2013a**



# NegativeBinomialDistribution

Negative binomial distribution object

## Description

A `NegativeBinomialDistribution` object consists of parameters, a model description, and sample data for a negative binomial probability distribution.

The negative binomial distribution models the number of failures  $x$  before a specified number of successes,  $R$ , is reached in a series of independent, identical trials. This distribution can also model count data, in which case  $R$  does not need to be an integer value.

The negative binomial distribution uses the following parameters.

Parameter	Description	Support
$R$	Number of successes	$r > 0$
$p$	Probability of success	$0 < p \leq 1$

## Creation

There are several ways to create a `NegativeBinomialDistribution` probability distribution object.

- Create a distribution with specified parameter values using `makedist`.
- Fit a distribution to data using `fitdist`.
- Interactively fit a distribution to data using the **Distribution Fitter** app.

## Properties

### Distribution Parameters

**r** — Number of successes

positive scalar value

Number of successes for the negative binomial distribution, specified as a positive scalar value.

Data Types: `single` | `double`

**p** — Probability of success

positive scalar value in the range (0,1]

Probability of success of any individual trial for the negative binomial distribution, specified as a positive scalar value in the range (0,1].

Data Types: `single` | `double`

### Distribution Characteristics

**IsTruncated** — Logical flag for truncated distribution

0 | 1

This property is read-only.

Logical flag for truncated distribution, specified as a logical value. If `IsTruncated` equals 0, the distribution is not truncated. If `IsTruncated` equals 1, the distribution is truncated.

Data Types: `logical`

**NumParameters** — Number of parameters

positive integer value

This property is read-only.

Number of parameters for the probability distribution, specified as a positive integer value.

Data Types: `single` | `double`

**ParameterCovariance** — Covariance matrix of the parameter estimates

matrix of scalar values

This property is read-only.

Covariance matrix of the parameter estimates, specified as a  $p$ -by- $p$  matrix, where  $p$  is the number of parameters in the distribution. The  $(i,j)$  element is the covariance between the estimates of the  $i$ th parameter and the  $j$ th parameter. The  $(i,i)$  element is the estimated variance of the  $i$ th parameter. If parameter  $i$  is fixed rather than estimated by fitting the distribution to data, then the  $(i,i)$  elements of the covariance matrix are 0.

Data Types: `single` | `double`

**ParameterIsFixed** — Logical flag for fixed parameters

array of logical values

This property is read-only.

Logical flag for fixed parameters, specified as an array of logical values. If 0, the corresponding parameter in the `ParameterNames` array is not fixed. If 1, the corresponding parameter in the `ParameterNames` array is fixed.

Data Types: `logical`

**ParameterValues** — Distribution parameter values

vector of scalar values

This property is read-only.

Distribution parameter values, specified as a vector.

Data Types: `single` | `double`

**Truncation** — Truncation interval

vector of scalar values

This property is read-only.

Truncation interval for the probability distribution, specified as a vector containing the lower and upper truncation boundaries.

Data Types: `single` | `double`

## Other Object Properties

### **DistributionName** — Probability distribution name

character vector

This property is read-only.

Probability distribution name, specified as a character vector.

Data Types: `char`

### **InputData** — Data used for distribution fitting

structure

This property is read-only.

Data used for distribution fitting, specified as a structure containing the following:

- `data`: Data vector used for distribution fitting.
- `cens`: Censoring vector, or empty if none.
- `freq`: Frequency vector, or empty if none.

Data Types: `struct`

### **ParameterDescription** — Distribution parameter descriptions

cell array of character vectors

This property is read-only.

Distribution parameter descriptions, specified as a cell array of character vectors. Each cell contains a short description of one distribution parameter.

Data Types: `char`

### **ParameterNames** — Distribution parameter names

cell array of character vectors

This property is read-only.

Distribution parameter names, specified as a cell array of character vectors.

Data Types: `char`

## Object Functions

<code>cdf</code>	Cumulative distribution functions
<code>icdf</code>	Inverse cumulative distribution functions
<code>iqr</code>	Interquartile range
<code>mean</code>	Mean of probability distribution
<code>median</code>	Median of probability distribution
<code>negloglik</code>	Negative log likelihood of probability distribution
<code>paramci</code>	Confidence intervals for probability distribution parameters
<code>pdf</code>	Probability density functions
<code>proflik</code>	Profile likelihood function for probability distribution
<code>random</code>	Random numbers
<code>std</code>	Standard deviation of probability distribution
<code>truncate</code>	Truncate probability distribution object
<code>var</code>	Variance of probability distribution

## Examples

### Create a Negative Binomial Distribution Object Using Default Parameters

Create a negative binomial distribution object using the default parameter values.

```
pd = makedist('NegativeBinomial')
```

```
pd =
```

```
NegativeBinomialDistribution
Negative Binomial distribution
R = 1
P = 0.5
```

### Create a Negative Binomial Distribution Object Using Specified Parameters

Create a negative binomial distribution object by specifying the parameter values.

```
pd = makedist('NegativeBinomial', 'R', 5, 'p', .1)
```

```
pd =
```

```
NegativeBinomialDistribution
```

```
Negative Binomial distribution
```

```
R = 5
```

```
P = 0.1
```

Compute the mean of the distribution.

```
m = mean(pd)
```

```
m =
```

```
45
```

## See Also

**Distribution Fitter** | `fitdist` | `makedist`

## Topics

“Negative Binomial Distribution” on page B-120

**Introduced in R2013a**

# NormalDistribution

Normal probability distribution object

## Description

A `NormalDistribution` object consists of parameters, a model description, and sample data for a normal probability distribution.

The normal distribution, sometimes called the Gaussian distribution, is a two-parameter family of curves. The usual justification for using the normal distribution for modeling is the Central Limit theorem, which states (roughly) that the sum of independent samples from any distribution with finite mean and variance converges to the normal distribution as the sample size goes to infinity.

The normal distribution uses the following parameters.

Parameter	Description	Support
<code>mu</code>	Mean	$-\infty < \mu < \infty$
<code>sigma</code>	Standard deviation	$\sigma \geq 0$

## Creation

There are several ways to create a `NormalDistribution` probability distribution object.

- Create a distribution with specified parameter values using `makedist`.
- Fit a distribution to data using `fitdist`.
- Interactively fit a distribution to data using the **Distribution Fitter** app.

## Properties

### Distribution Parameters

**mu — Mean**

scalar value

Mean of the normal distribution, specified as a scalar value.

Data Types: `single` | `double`

**sigma — Standard deviation**

nonnegative scalar value

Standard deviation of the normal distribution, specified as a nonnegative scalar value.

Data Types: `single` | `double`

### Distribution Characteristics

**IsTruncated — Logical flag for truncated distribution**

0 | 1

This property is read-only.

Logical flag for truncated distribution, specified as a logical value. If `IsTruncated` equals 0, the distribution is not truncated. If `IsTruncated` equals 1, the distribution is truncated.

Data Types: `logical`

**NumParameters — Number of parameters**

positive integer value

This property is read-only.

Number of parameters for the probability distribution, specified as a positive integer value.

Data Types: `single` | `double`



---

**ParameterCovariance** — Covariance matrix of the parameter estimates  
matrix of scalar values

This property is read-only.

Covariance matrix of the parameter estimates, specified as a  $p$ -by- $p$  matrix, where  $p$  is the number of parameters in the distribution. The  $(i,j)$  element is the covariance between the estimates of the  $i$ th parameter and the  $j$ th parameter. The  $(i,i)$  element is the estimated variance of the  $i$ th parameter. If parameter  $i$  is fixed rather than estimated by fitting the distribution to data, then the  $(i,i)$  elements of the covariance matrix are 0.

Data Types: `single` | `double`

**ParameterIsFixed** — Logical flag for fixed parameters  
array of logical values

This property is read-only.

Logical flag for fixed parameters, specified as an array of logical values. If 0, the corresponding parameter in the `ParameterNames` array is not fixed. If 1, the corresponding parameter in the `ParameterNames` array is fixed.

Data Types: `logical`

**ParameterValues** — Distribution parameter values  
vector of scalar values

This property is read-only.

Distribution parameter values, specified as a vector.

Data Types: `single` | `double`

**Truncation** — Truncation interval  
vector of scalar values

This property is read-only.

Truncation interval for the probability distribution, specified as a vector containing the lower and upper truncation boundaries.

Data Types: `single` | `double`

## Other Object Properties

### **DistributionName** — Probability distribution name

character vector

This property is read-only.

Probability distribution name, specified as a character vector.

Data Types: `char`

### **InputData** — Data used for distribution fitting

structure

This property is read-only.

Data used for distribution fitting, specified as a structure containing the following:

- `data`: Data vector used for distribution fitting.
- `cens`: Censoring vector, or empty if none.
- `freq`: Frequency vector, or empty if none.

Data Types: `struct`

### **ParameterDescription** — Distribution parameter descriptions

cell array of character vectors

This property is read-only.

Distribution parameter descriptions, specified as a cell array of character vectors. Each cell contains a short description of one distribution parameter.

Data Types: `char`

### **ParameterNames** — Distribution parameter names

cell array of character vectors

This property is read-only.

Distribution parameter names, specified as a cell array of character vectors.

Data Types: `char`

## Object Functions

<code>cdf</code>	Cumulative distribution functions
<code>icdf</code>	Inverse cumulative distribution functions
<code>iqr</code>	Interquartile range
<code>mean</code>	Mean of probability distribution
<code>median</code>	Median of probability distribution
<code>negloglik</code>	Negative log likelihood of probability distribution
<code>paramci</code>	Confidence intervals for probability distribution parameters
<code>pdf</code>	Probability density functions
<code>proflik</code>	Profile likelihood function for probability distribution
<code>random</code>	Random numbers
<code>std</code>	Standard deviation of probability distribution
<code>truncate</code>	Truncate probability distribution object
<code>var</code>	Variance of probability distribution

## Examples

### Create a Normal Distribution Object Using Default Parameters

Create a normal distribution object using the default parameter values.

```
pd = makedist('Normal')
```

```
pd =
```

```
NormalDistribution
```

```
Normal distribution
```

```
mu = 0
```

```
sigma = 1
```

### Create a Normal Distribution Object Using Specified Parameters

Create a normal distribution object by specifying the parameter values.

```
pd = makedist('Normal', 'mu', 75, 'sigma', 10)
```

```
pd =
```

```
NormalDistribution

Normal distribution
  mu = 75
  sigma = 10
```

Compute the interquartile range of the distribution.

```
r = iqr(pd)
```

```
r =
```

```
13.4898
```

### Fit a Normal Distribution Object

Load the sample data. Create a vector containing the first column of students' exam grades data.

```
load examgrades;
x = grades(:,1);
```

Create a normal distribution object by fitting it to the data.

```
pd = fitdist(x, 'Normal')
```

```
pd =
```

```
NormalDistribution

Normal distribution
  mu = 75.0083    [73.4321, 76.5846]
  sigma = 8.7202    [7.7391, 9.98843]
```

- “Compare Multiple Distribution Fits” on page 5-114

## See Also

**Distribution Fitter** | `fitdist` | `makedist`

## **Topics**

“Compare Multiple Distribution Fits” on page 5-114

“Normal Distribution” on page B-134

**Introduced in R2013a**

# PiecewiseLinearDistribution

Piecewise linear probability distribution object

## Description

A `PiecewiseLinearDistribution` object consists of a model description for a piecewise linear probability distribution.

The piecewise linear distribution is a nonparametric probability distribution created using a piecewise linear representation of the cumulative distribution function (cdf). The options specified for the piecewise linear distribution specify the form of the cdf. The probability density function (pdf) is a step function.

The piecewise linear distribution uses the following parameters.

Parameter	Description
<code>x</code>	Vector of <code>x</code> values at which the cdf changes slope
<code>Fx</code>	Vector of cdf values that correspond to each value in <code>x</code>

## Creation

Create a `PiecewiseLinearDistribution` probability distribution with specified parameter values object using `makedist`.

## Properties

### Distribution Parameters

**`x`** — Data values

vector of scalar values

Data values at which the cumulative distribution function (cdf) changes slope, specified as a vector of scalar values.

Data Types: `single` | `double`

**Fx — cdf value**

vector of scalar values

cdf value at each value in `x`, specified as a vector of scalar values.

Data Types: `single` | `double`

## Distribution Characteristics

**IsTruncated — Logical flag for truncated distribution**

0 | 1

This property is read-only.

Logical flag for truncated distribution, specified as a logical value. If `IsTruncated` equals 0, the distribution is not truncated. If `IsTruncated` equals 1, the distribution is truncated.

Data Types: `logical`

**NumParameters — Number of parameters**

positive integer value

This property is read-only.

Number of parameters for the probability distribution, specified as a positive integer value.

Data Types: `single` | `double`

**ParameterValues — Distribution parameter values**

vector of scalar values

This property is read-only.

Distribution parameter values, specified as a vector.

Data Types: `single` | `double`

**Truncation** — Truncation interval

vector of scalar values

This property is read-only.

Truncation interval for the probability distribution, specified as a vector containing the lower and upper truncation boundaries.

Data Types: `single` | `double`

## Other Object Properties

**DistributionName** — Probability distribution name

character vector

This property is read-only.

Probability distribution name, specified as a character vector.

Data Types: `char`

**ParameterDescription** — Distribution parameter descriptions

cell array of character vectors

This property is read-only.

Distribution parameter descriptions, specified as a cell array of character vectors. Each cell contains a short description of one distribution parameter.

Data Types: `char`

**ParameterNames** — Distribution parameter names

cell array of character vectors

This property is read-only.

Distribution parameter names, specified as a cell array of character vectors.

Data Types: `char`



## Object Functions

### Examples

#### Create a Piecewise Linear Distribution Object Using Default Parameters

Create a piecewise linear distribution object using the default parameter values.

```
pd = makedist('PiecewiseLinear')
```

```
pd =
```

```
    PiecewiseLinearDistribution
```

```
F(0) = 0
```

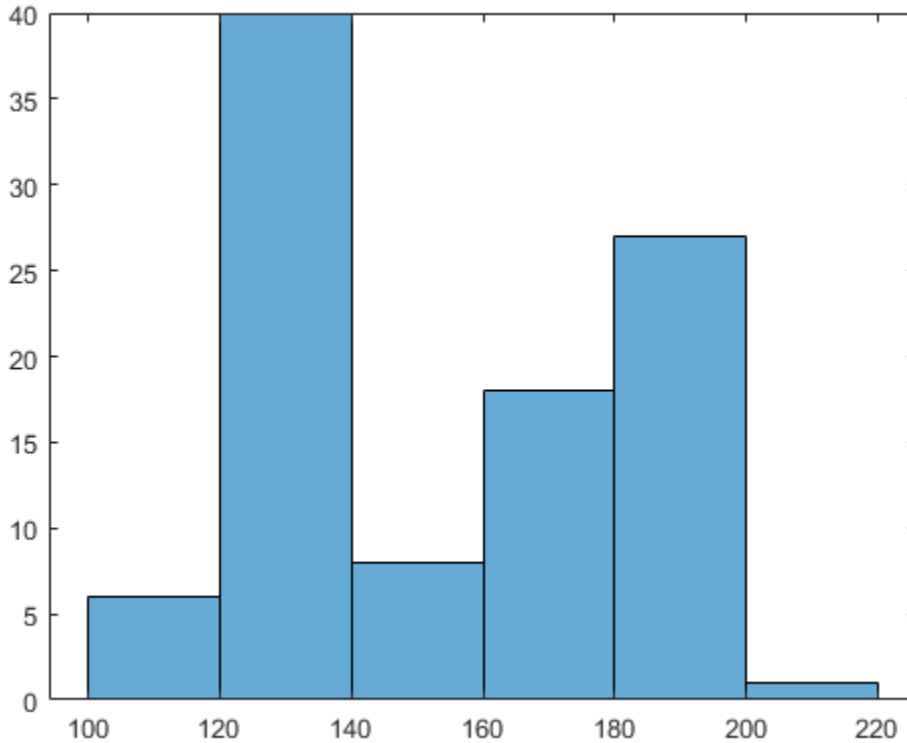
```
F(1) = 1
```

#### Create a Piecewise Linear Distribution Object Using Specified Parameters

Load the sample data. Visualize the patient weight data using a histogram.

```
load hospital
```

```
histogram(hospital.Weight)
```



The histogram shows that the data has two modes, one for female patients and one for male patients.

Compute the empirical cumulative distribution function (ecdf) for the data.

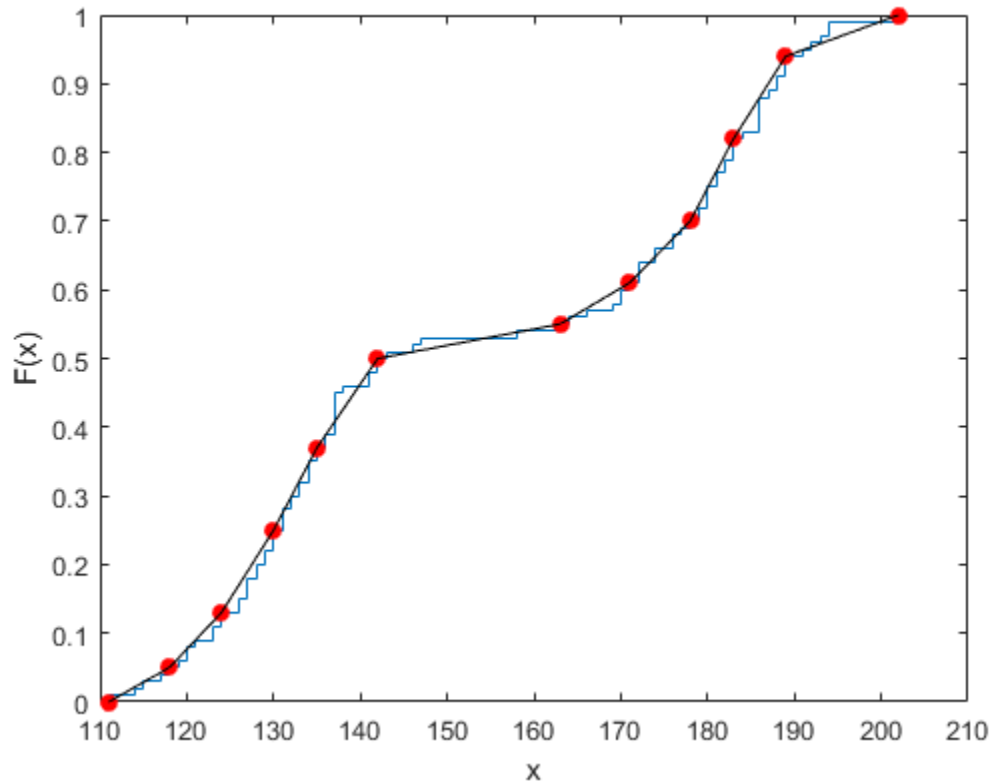
```
[f,x] = ecdf(hospital.Weight);
```

Construct a piecewise linear approximation to the ecdf and plot both functions.

```
f = f(1:5:end); % keep a less dense grid of points  
x = x(1:5:end);
```

```
figure;  
ecdf(hospital.Weight)
```

```
hold on
plot(x,f,'ro','MarkerFace','r') % overlay grid
plot(x,f,'k') % show interpolation
```



Create a piecewise linear probability distribution object using the piecewise approximation of the ecdf.

```
pd = makedist('PiecewiseLinear', 'x', x, 'Fx', f)
```

```
pd =
```

```
PiecewiseLinearDistribution
```

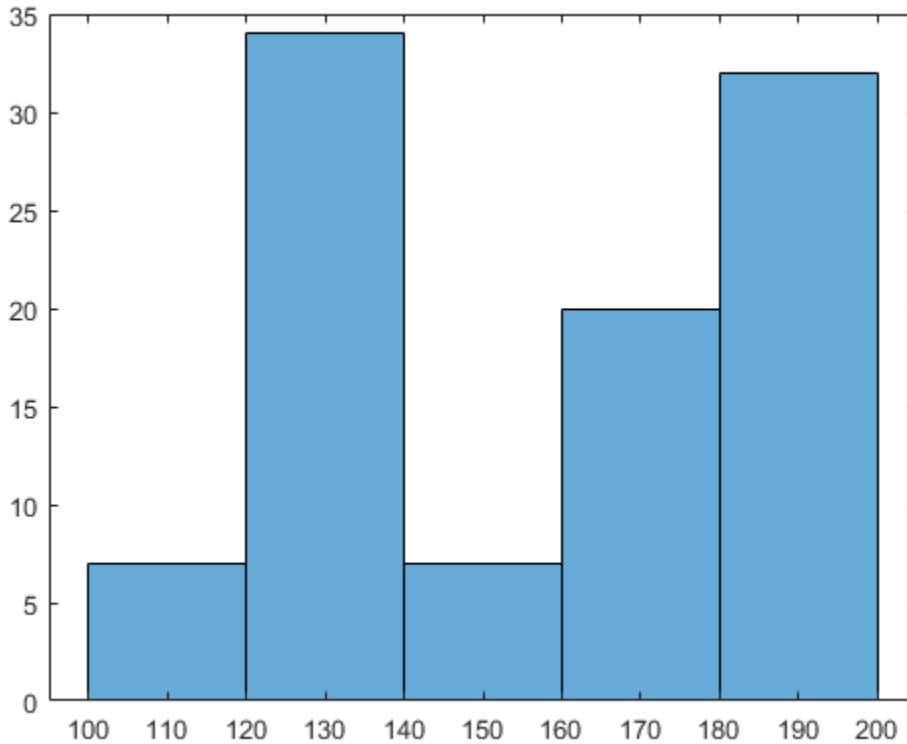
```
F(111) = 0
F(118) = 0.05
F(124) = 0.13
F(130) = 0.25
F(135) = 0.37
F(142) = 0.5
F(163) = 0.55
F(171) = 0.61
F(178) = 0.7
F(183) = 0.82
F(189) = 0.94
F(202) = 1
```

Generate 100 random numbers from the distribution.

```
rw = random(pd, 100, 1);
```

Plot the random numbers to visually compare their distribution to the original data.

```
figure;
histogram(rw)
```



The random numbers generated from the piecewise linear distribution have the same bimodal distribution as the original data.

## See Also

`makedist`

## Topics

“Nonparametric and Empirical Probability Distributions” on page 5-37

“Piecwise Linear Distribution” on page B-141

**Introduced in R2013a**

# PoissonDistribution

Poisson probability distribution object

## Description

A `PoissonDistribution` object consists of parameters, a model description, and sample data for a Poisson probability distribution.

The Poisson distribution is appropriate for applications that involve counting the number of times a random event occurs in a given amount of time, distance, area, etc. If the number of counts follows the Poisson distribution, then the interval between individual counts follows the exponential distribution.

The Poisson distribution uses the following parameters.

Parameter	Description	Support
<code>lambda</code>	Mean	$\lambda \geq 0$

## Creation

There are several ways to create a `PoissonDistribution` probability distribution object.

- Create a distribution with specified parameter values using `makedist`.
- Fit a distribution to data using `fitdist`.
- Interactively fit a distribution to data using the **Distribution Fitter** app.

## Properties

### Distribution Parameter

`lambda` — Mean

nonnegative scalar value

Mean of the Poisson distribution, stored as a nonnegative scalar value.

Data Types: `single` | `double`

## Distribution Characteristics

### **IsTruncated** — Logical flag for truncated distribution

0 | 1

This property is read-only.

Logical flag for truncated distribution, specified as a logical value. If `IsTruncated` equals 0, the distribution is not truncated. If `IsTruncated` equals 1, the distribution is truncated.

Data Types: `logical`

### **NumParameters** — Number of parameters

positive integer value

This property is read-only.

Number of parameters for the probability distribution, specified as a positive integer value.

Data Types: `single` | `double`

### **ParameterCovariance** — Covariance matrix of the parameter estimates

matrix of scalar values

This property is read-only.

Covariance matrix of the parameter estimates, specified as a  $p$ -by- $p$  matrix, where  $p$  is the number of parameters in the distribution. The  $(i,j)$  element is the covariance between the estimates of the  $i$ th parameter and the  $j$ th parameter. The  $(i,i)$  element is the estimated variance of the  $i$ th parameter. If parameter  $i$  is fixed rather than estimated by fitting the distribution to data, then the  $(i,i)$  elements of the covariance matrix are 0.

Data Types: `single` | `double`

### **ParameterIsFixed** — Logical flag for fixed parameters

array of logical values



This property is read-only.

Logical flag for fixed parameters, specified as an array of logical values. If 0, the corresponding parameter in the `ParameterNames` array is not fixed. If 1, the corresponding parameter in the `ParameterNames` array is fixed.

Data Types: `logical`

**ParameterValues — Distribution parameter values**

vector of scalar values

This property is read-only.

Distribution parameter values, specified as a vector.

Data Types: `single` | `double`

**Truncation — Truncation interval**

vector of scalar values

This property is read-only.

Truncation interval for the probability distribution, specified as a vector containing the lower and upper truncation boundaries.

Data Types: `single` | `double`

## Other Object Properties

**DistributionName — Probability distribution name**

character vector

This property is read-only.

Probability distribution name, specified as a character vector.

Data Types: `char`

**InputData — Data used for distribution fitting**

structure

This property is read-only.

Data used for distribution fitting, specified as a structure containing the following:

- `data`: Data vector used for distribution fitting.
- `cens`: Censoring vector, or empty if none.
- `freq`: Frequency vector, or empty if none.

Data Types: `struct`

#### **ParameterDescription** — Distribution parameter descriptions

cell array of character vectors

This property is read-only.

Distribution parameter descriptions, specified as a cell array of character vectors. Each cell contains a short description of one distribution parameter.

Data Types: `char`

#### **ParameterNames** — Distribution parameter names

cell array of character vectors

This property is read-only.

Distribution parameter names, specified as a cell array of character vectors.

Data Types: `char`

## Object Functions

<code>cdf</code>	Cumulative distribution functions
<code>icdf</code>	Inverse cumulative distribution functions
<code>iqr</code>	Interquartile range
<code>mean</code>	Mean of probability distribution
<code>median</code>	Median of probability distribution
<code>negloglik</code>	Negative log likelihood of probability distribution
<code>paramci</code>	Confidence intervals for probability distribution parameters
<code>pdf</code>	Probability density functions
<code>proflik</code>	Profile likelihood function for probability distribution
<code>random</code>	Random numbers
<code>std</code>	Standard deviation of probability distribution
<code>truncate</code>	Truncate probability distribution object
<code>var</code>	Variance of probability distribution

## Examples

### Create a Poisson Distribution Object Using Default Parameters

Create a Poisson distribution object using the default parameter values.

```
pd = makedist('Poisson')
```

```
pd =  
  
PoissonDistribution  
  
Poisson distribution  
lambda = 1
```

### Create a Poisson Distribution Object Using Specified Parameters

Create a Poisson distribution object by specifying the parameter values.

```
pd = makedist('Poisson', 'lambda', 5)
```

```
pd =  
  
PoissonDistribution  
  
Poisson distribution  
lambda = 5
```

Compute the variance of the distribution.

```
v = var(pd)
```

```
v =  
  
5
```

For the Poisson distribution, both the mean and variance are equal to the parameter  $\lambda$ .

## See Also

**Distribution Fitter** | `fitdist` | `makedist`

## Topics

“Poisson Distribution” on page B-143

**Introduced in R2013a**

# RayleighDistribution

Rayleigh probability distribution object

## Description

A `RayleighDistribution` object consists of parameters, a model description, and sample data for a normal probability distribution.

The Rayleigh distribution is a special case of the Weibull distribution. It is often used in communication theory to model scattered signals that reach a receiver by multiple paths.

The Rayleigh distribution uses the following parameter.

Parameter	Description	Support
$b$	Defining parameter	$b > 0$

## Creation

There are several ways to create a `RayleighDistribution` probability distribution object.

- Create a distribution with specified parameter values using `makedist`.
- Fit a distribution to data using `fitdist`.
- Interactively fit a distribution to data using the **Distribution Fitter** app.

## Properties

### Distribution Parameter

$b$  — **Defining parameter**  
positive scalar value

Defining parameter for the Rayleigh distribution, specified as a positive scalar value.

Data Types: `single` | `double`

## Distribution Characteristics

### **IsTruncated** — Logical flag for truncated distribution

0 | 1

This property is read-only.

Logical flag for truncated distribution, specified as a logical value. If `IsTruncated` equals 0, the distribution is not truncated. If `IsTruncated` equals 1, the distribution is truncated.

Data Types: `logical`

### **NumParameters** — Number of parameters

positive integer value

This property is read-only.

Number of parameters for the probability distribution, specified as a positive integer value.

Data Types: `single` | `double`

### **ParameterCovariance** — Covariance matrix of the parameter estimates

matrix of scalar values

This property is read-only.

Covariance matrix of the parameter estimates, specified as a  $p$ -by- $p$  matrix, where  $p$  is the number of parameters in the distribution. The  $(i,j)$  element is the covariance between the estimates of the  $i$ th parameter and the  $j$ th parameter. The  $(i,i)$  element is the estimated variance of the  $i$ th parameter. If parameter  $i$  is fixed rather than estimated by fitting the distribution to data, then the  $(i,i)$  elements of the covariance matrix are 0.

Data Types: `single` | `double`

### **ParameterIsFixed** — Logical flag for fixed parameters

array of logical values

This property is read-only.

Logical flag for fixed parameters, specified as an array of logical values. If 0, the corresponding parameter in the `ParameterNames` array is not fixed. If 1, the corresponding parameter in the `ParameterNames` array is fixed.

Data Types: `logical`

**ParameterValues** — Distribution parameter values

vector of scalar values

This property is read-only.

Distribution parameter values, specified as a vector.

Data Types: `single` | `double`

**Truncation** — Truncation interval

vector of scalar values

This property is read-only.

Truncation interval for the probability distribution, specified as a vector containing the lower and upper truncation boundaries.

Data Types: `single` | `double`

## Other Object Properties

**DistributionName** — Probability distribution name

character vector

This property is read-only.

Probability distribution name, specified as a character vector.

Data Types: `char`

**InputData** — Data used for distribution fitting

structure

This property is read-only.

Data used for distribution fitting, specified as a structure containing the following:

- `data`: Data vector used for distribution fitting.
- `cens`: Censoring vector, or empty if none.
- `freq`: Frequency vector, or empty if none.

Data Types: `struct`

#### **ParameterDescription** — Distribution parameter descriptions

cell array of character vectors

This property is read-only.

Distribution parameter descriptions, specified as a cell array of character vectors. Each cell contains a short description of one distribution parameter.

Data Types: `char`

#### **ParameterNames** — Distribution parameter names

cell array of character vectors

This property is read-only.

Distribution parameter names, specified as a cell array of character vectors.

Data Types: `char`

## Object Functions

<code>cdf</code>	Cumulative distribution functions
<code>icdf</code>	Inverse cumulative distribution functions
<code>iqr</code>	Interquartile range
<code>mean</code>	Mean of probability distribution
<code>median</code>	Median of probability distribution
<code>negloglik</code>	Negative log likelihood of probability distribution
<code>paramci</code>	Confidence intervals for probability distribution parameters
<code>pdf</code>	Probability density functions
<code>proflik</code>	Profile likelihood function for probability distribution
<code>random</code>	Random numbers
<code>std</code>	Standard deviation of probability distribution
<code>truncate</code>	Truncate probability distribution object
<code>var</code>	Variance of probability distribution



## Examples

### Create a Rayleigh Distribution Object Using Default Parameters

Create a Rayleigh distribution object using the default parameter values.

```
pd = makedist('Rayleigh')
```

```
pd =  
  
RayleighDistribution  
  
Rayleigh distribution  
B = 1
```

### Create a Rayleigh Distribution Object Using Specified Parameters

Create a Rayleigh distribution object by specifying the parameter values.

```
pd = makedist('Rayleigh', 'b', 3)
```

```
pd =  
  
RayleighDistribution  
  
Rayleigh distribution  
B = 3
```

Compute the mean of the distribution.

```
m = mean(pd)
```

```
m =
```

3.7599

## See Also

**Distribution Fitter** | `fitdist` | `makedist`

## Topics

“Rayleigh Distribution” on page B-146

**Introduced in R2013a**

# RicianDistribution

Rician probability distribution object

## Description

A `RicianDistribution` object consists of parameters, a model description, and sample data for a Rician probability distribution.

The Rician distribution is used in communications theory to model scattered signals that reach a receiver using multiple paths.

The Rician distribution uses the following parameters.

Name	Description	Support
<code>s</code>	Noncentrality parameter	$s \geq 0$
<code>sigma</code>	Scale parameter	$\sigma > 0$

## Creation

There are several ways to create a `RicianDistribution` probability distribution object.

- Create a distribution with specified parameter values using `makedist`.
- Fit a distribution to data using `fitdist`.
- Interactively fit a distribution to data using the **Distribution Fitter** app.

## Properties

### Distribution Parameters

**`s`** — Noncentrality parameter  
nonnegative scalar value

Noncentrality parameter of the Rician distribution, specified as a nonnegative scalar value.

Data Types: `single` | `double`

**sigma — scale parameter**

positive scalar value

Scale parameter for the Rician distribution, specified as a positive scalar value.

Data Types: `single` | `double`

## Distribution Characteristics

**IsTruncated — Logical flag for truncated distribution**

0 | 1

This property is read-only.

Logical flag for truncated distribution, specified as a logical value. If `IsTruncated` equals 0, the distribution is not truncated. If `IsTruncated` equals 1, the distribution is truncated.

Data Types: `logical`

**NumParameters — Number of parameters**

positive integer value

This property is read-only.

Number of parameters for the probability distribution, specified as a positive integer value.

Data Types: `single` | `double`

**ParameterCovariance — Covariance matrix of the parameter estimates**

matrix of scalar values

This property is read-only.

Covariance matrix of the parameter estimates, specified as a  $p$ -by- $p$  matrix, where  $p$  is the number of parameters in the distribution. The  $(i,j)$  element is the covariance between the estimates of the  $i$ th parameter and the  $j$ th parameter. The  $(i,i)$  element is

the estimated variance of the  $i$ th parameter. If parameter  $i$  is fixed rather than estimated by fitting the distribution to data, then the  $(i,i)$  elements of the covariance matrix are 0.

Data Types: `single` | `double`

**ParameterIsFixed** — Logical flag for fixed parameters

array of logical values

This property is read-only.

Logical flag for fixed parameters, specified as an array of logical values. If 0, the corresponding parameter in the `ParameterNames` array is not fixed. If 1, the corresponding parameter in the `ParameterNames` array is fixed.

Data Types: `logical`

**ParameterValues** — Distribution parameter values

vector of scalar values

This property is read-only.

Distribution parameter values, specified as a vector.

Data Types: `single` | `double`

**Truncation** — Truncation interval

vector of scalar values

This property is read-only.

Truncation interval for the probability distribution, specified as a vector containing the lower and upper truncation boundaries.

Data Types: `single` | `double`

## Other Object Properties

**DistributionName** — Probability distribution name

character vector

This property is read-only.

Probability distribution name, specified as a character vector.

Data Types: `char`

**InputData** — Data used for distribution fitting

structure

This property is read-only.

Data used for distribution fitting, specified as a structure containing the following:

- `data`: Data vector used for distribution fitting.
- `cens`: Censoring vector, or empty if none.
- `freq`: Frequency vector, or empty if none.

Data Types: `struct`

**ParameterDescription** — Distribution parameter descriptions

cell array of character vectors

This property is read-only.

Distribution parameter descriptions, specified as a cell array of character vectors. Each cell contains a short description of one distribution parameter.

Data Types: `char`

**ParameterNames** — Distribution parameter names

cell array of character vectors

This property is read-only.

Distribution parameter names, specified as a cell array of character vectors.

Data Types: `char`

## Object Functions

<code>cdf</code>	Cumulative distribution functions
<code>icdf</code>	Inverse cumulative distribution functions
<code>iqr</code>	Interquartile range
<code>mean</code>	Mean of probability distribution

---

median	Median of probability distribution
negloglik	Negative log likelihood of probability distribution
paramci	Confidence intervals for probability distribution parameters
pdf	Probability density functions
proflik	Profile likelihood function for probability distribution
random	Random numbers
std	Standard deviation of probability distribution
truncate	Truncate probability distribution object
var	Variance of probability distribution

## Examples

### Create a Rician Distribution Object Using Default Parameters

Create a Rician distribution object using the default parameter values.

```
pd = makedist('Rician')
```

```
pd =
```

```
RicianDistribution
```

```
Rician distribution
```

```
    s = 1
```

```
    sigma = 1
```

### Create a Rician Distribution Object Using Specified Parameters

Create a Rician distribution object by specifying the parameter values.

```
pd = makedist('Rician', 's', 0, 'sigma', 2)
```

```
pd =
```

```
RicianDistribution
```

```
Rician distribution
```

```
    s = 0
```

```
    sigma = 2
```

Compute the mean of the distribution.

```
m = mean(pd)
```

```
m =
```

```
2.5066
```

## See Also

**Distribution Fitter** | `fitdist` | `makedist`

## Topics

“Rician Distribution” on page B-149

**Introduced in R2013a**



# StableDistribution

Stable probability distribution object

## Description

`StableDistribution` is an object consisting of parameters, a model description, and sample data for a stable probability distribution.

The stable distribution uses the following parameters.

Parameter	Description	Support
<code>alpha</code>	First shape parameter	$0 < \alpha \leq 2$
<code>beta</code>	Second shape parameter	$-1 \leq \beta \leq 1$
<code>gam</code>	Scale parameter	$0 < \gamma < \infty$
<code>delta</code>	Location parameter	$-\infty < \delta < \infty$

## Creation

There are several ways to create a `StableDistribution` probability distribution object.

- Create a distribution with specified parameter values using `makedist`.
- Fit a distribution to data using `fitdist`.
- Interactively fit a distribution to data using the **Distribution Fitter** app.

## Properties

### Distribution Parameters

`alpha` — First shape parameter  
scalar value in the range (0,2]

First shape parameter of the stable distribution, specified as a scalar value in the range (0,2].

Data Types: `single` | `double`

**beta** — Second shape parameter

scalar value in the range [-1,1]

Second shape parameter of the stable distribution, specified as a scalar value in the range [-1,1].

Data Types: `single` | `double`

**gam** — Scale parameter

scalar value in the range (0,∞)

Scale parameter of the stable distribution, specified as a scalar value in the range (0,∞).

Data Types: `single` | `double`

**delta** — Location parameter

scalar value

Location parameter of the stable distribution, specified as a scalar value.

Data Types: `single` | `double`

## Distribution Characteristics

**IsTruncated** — Logical flag for truncated distribution

0 | 1

This property is read-only.

Logical flag for truncated distribution, specified as a logical value. If `IsTruncated` equals 0, the distribution is not truncated. If `IsTruncated` equals 1, the distribution is truncated.

Data Types: `logical`

**NumParameters** — Number of parameters

positive integer value

This property is read-only.

Number of parameters for the probability distribution, specified as a positive integer value.

Data Types: `single` | `double`

**ParameterCovariance** — Covariance matrix of the parameter estimates

matrix of scalar values

This property is read-only.

Covariance matrix of the parameter estimates, specified as a  $p$ -by- $p$  matrix, where  $p$  is the number of parameters in the distribution. The  $(i,j)$  element is the covariance between the estimates of the  $i$ th parameter and the  $j$ th parameter. The  $(i,i)$  element is the estimated variance of the  $i$ th parameter. If parameter  $i$  is fixed rather than estimated by fitting the distribution to data, then the  $(i,i)$  elements of the covariance matrix are 0.

Data Types: `single` | `double`

**ParameterIsFixed** — Logical flag for fixed parameters

array of logical values

This property is read-only.

Logical flag for fixed parameters, specified as an array of logical values. If 0, the corresponding parameter in the `ParameterNames` array is not fixed. If 1, the corresponding parameter in the `ParameterNames` array is fixed.

Data Types: `logical`

**ParameterValues** — Distribution parameter values

vector of scalar values

This property is read-only.

Distribution parameter values, specified as a vector.

Data Types: `single` | `double`

**Truncation** — Truncation interval

vector of scalar values

This property is read-only.

Truncation interval for the probability distribution, specified as a vector containing the lower and upper truncation boundaries.

Data Types: `single` | `double`

## Other Object Properties

### **DistributionName** — Probability distribution name

character vector

This property is read-only.

Probability distribution name, specified as a character vector.

Data Types: `char`

### **InputData** — Data used for distribution fitting

structure

This property is read-only.

Data used for distribution fitting, specified as a structure containing the following:

- `data`: Data vector used for distribution fitting.
- `cens`: Censoring vector, or empty if none.
- `freq`: Frequency vector, or empty if none.

Data Types: `struct`

### **ParameterDescription** — Distribution parameter descriptions

cell array of character vectors

This property is read-only.

Distribution parameter descriptions, specified as a cell array of character vectors. Each cell contains a short description of one distribution parameter.

Data Types: `char`

### **ParameterNames** — Distribution parameter names

cell array of character vectors

This property is read-only.

Distribution parameter names, specified as a cell array of character vectors.

Data Types: `char`

## Object Functions

<code>cdf</code>	Cumulative distribution functions
<code>icdf</code>	Inverse cumulative distribution functions
<code>iqr</code>	Interquartile range
<code>mean</code>	Mean of probability distribution
<code>median</code>	Median of probability distribution
<code>negloglik</code>	Negative log likelihood of probability distribution
<code>paramci</code>	Confidence intervals for probability distribution parameters
<code>pdf</code>	Probability density functions
<code>proflik</code>	Profile likelihood function for probability distribution
<code>random</code>	Random numbers
<code>std</code>	Standard deviation of probability distribution
<code>truncate</code>	Truncate probability distribution object
<code>var</code>	Variance of probability distribution

## Examples

### Create a Stable Distribution Object Using Default Parameters

Create a stable distribution object using the default parameter values.

```
pd = makedist('Stable')
```

```
pd =
```

```
StableDistribution
```

```
Stable distribution
```

```
alpha = 2
```

```
beta = 0
```

```
gam = 1
```

```
delta = 0
```

### Create a Stable Distribution Object Using Specified Parameters

Create a stable distribution object by specifying the parameter values  $\alpha = 0.5$ ,  $\beta = 0$ ,  $\gamma = 1$ , and  $\delta = 0$ .

```
pd = makedist('Stable', 'alpha', 0.5, 'beta', 0, 'gam', 1, 'delta', 0);
```

Calculate the mean of the distribution.

```
m = mean(pd)
```

```
m =
```

```
NaN
```

The mean of the stable distribution is undefined for values of  $\alpha$  less than or equal to 1.

## See Also

**Distribution Fitter** | `fitdist` | `makedist`

## Topics

“Stable Distribution” on page B-151

**Introduced in R2016a**

# tLocationScaleDistribution

*t* Location-Scale probability distribution object

## Description

A `tLocationScaleDistribution` object consists of parameters, a model description, and sample data for a *t* location-scale probability distribution.

The *t* location-scale distribution is useful for modeling data distributions with heavier tails (more prone to outliers) than the normal distribution. It approaches the normal distribution as  $\nu$  approaches infinity, and smaller values of  $\nu$  yield heavier tails.

The *t* location-scale distribution uses the following parameters.

Parameter	Description	Support
<code>mu</code>	Location parameter	$-\infty < \mu < \infty$
<code>sigma</code>	Scale parameter	$\sigma > 0$
<code>nu</code>	Shape parameter	$\nu > 0$

## Creation

There are several ways to create a `tLocationScaleDistribution` probability distribution object.

- Create a distribution with specified parameter values using `makedist`.
- Fit a distribution to data using `fitdist`.
- Interactively fit a distribution to data using the **Distribution Fitter** app.

## Properties

### Distribution Parameters

**mu** — Location parameter

scalar value

Location parameter of the  $t$  location-scale distribution, specified as a scalar value.

Data Types: `single` | `double`

**sigma** — Scale parameter

positive scalar value

Scale parameter of the  $t$  location-scale distribution, specified as a positive scalar value.

Data Types: `single` | `double`

**nu** — Degrees of freedom

positive scalar value

Degrees of freedom of the  $t$  location-scale distribution, specified as a positive scalar value.

Data Types: `single` | `double`

### Distribution Characteristics

**IsTruncated** — Logical flag for truncated distribution

0 | 1

This property is read-only.

Logical flag for truncated distribution, specified as a logical value. If `IsTruncated` equals 0, the distribution is not truncated. If `IsTruncated` equals 1, the distribution is truncated.

Data Types: `logical`

**NumParameters** — Number of parameters

positive integer value

This property is read-only.



Number of parameters for the probability distribution, specified as a positive integer value.

Data Types: `single` | `double`

**ParameterCovariance** — Covariance matrix of the parameter estimates

matrix of scalar values

This property is read-only.

Covariance matrix of the parameter estimates, specified as a  $p$ -by- $p$  matrix, where  $p$  is the number of parameters in the distribution. The  $(i,j)$  element is the covariance between the estimates of the  $i$ th parameter and the  $j$ th parameter. The  $(i,i)$  element is the estimated variance of the  $i$ th parameter. If parameter  $i$  is fixed rather than estimated by fitting the distribution to data, then the  $(i,i)$  elements of the covariance matrix are 0.

Data Types: `single` | `double`

**ParameterIsFixed** — Logical flag for fixed parameters

array of logical values

This property is read-only.

Logical flag for fixed parameters, specified as an array of logical values. If 0, the corresponding parameter in the `ParameterNames` array is not fixed. If 1, the corresponding parameter in the `ParameterNames` array is fixed.

Data Types: `logical`

**ParameterValues** — Distribution parameter values

vector of scalar values

This property is read-only.

Distribution parameter values, specified as a vector.

Data Types: `single` | `double`

**Truncation** — Truncation interval

vector of scalar values

This property is read-only.

Truncation interval for the probability distribution, specified as a vector containing the lower and upper truncation boundaries.

Data Types: `single` | `double`

## Other Object Properties

### **DistributionName** — Probability distribution name

character vector

This property is read-only.

Probability distribution name, specified as a character vector.

Data Types: `char`

### **InputData** — Data used for distribution fitting

structure

This property is read-only.

Data used for distribution fitting, specified as a structure containing the following:

- `data`: Data vector used for distribution fitting.
- `cens`: Censoring vector, or empty if none.
- `freq`: Frequency vector, or empty if none.

Data Types: `struct`

### **ParameterDescription** — Distribution parameter descriptions

cell array of character vectors

This property is read-only.

Distribution parameter descriptions, specified as a cell array of character vectors. Each cell contains a short description of one distribution parameter.

Data Types: `char`

### **ParameterNames** — Distribution parameter names

cell array of character vectors

This property is read-only.

Distribution parameter names, specified as a cell array of character vectors.

Data Types: char

## Object Functions

cdf	Cumulative distribution functions
icdf	Inverse cumulative distribution functions
iqr	Interquartile range
mean	Mean of probability distribution
median	Median of probability distribution
negloglik	Negative log likelihood of probability distribution
paramci	Confidence intervals for probability distribution parameters
pdf	Probability density functions
proflik	Profile likelihood function for probability distribution
random	Random numbers
std	Standard deviation of probability distribution
truncate	Truncate probability distribution object
var	Variance of probability distribution

## Examples

### Create a t Location-Scale Distribution Object Using Default Parameters

Create a `t` location scale distribution object using the default parameter values.

```
pd = makedist('tLocationScale')
```

```
pd =
```

```
tLocationScaleDistribution

t Location-Scale distribution
    mu = 0
    sigma = 1
```

```
nu = 5
```

### Create a t Location-Scale Distribution Object Using Specified Parameters

Create a `t` location-scale distribution object by specifying the parameter values.

```
pd = makedist('tLocationScale', 'mu', -2, 'sigma', 1, 'nu', 20)
```

```
pd =
```

```
tLocationScaleDistribution  
  
t Location-Scale distribution  
  mu = -2  
  sigma = 1  
  nu = 20
```

Compute the interquartile range of the distribution.

```
r = iqr(pd)
```

```
r =
```

```
1.3739
```

- “Represent Cauchy Distribution Using t Location-Scale” on page 5-136

## See Also

**Distribution Fitter** | `fitdist` | `makedist`

## Topics

“Represent Cauchy Distribution Using t Location-Scale” on page 5-136

“t Location-Scale Distribution” on page B-175

**Introduced in R2013a**

# TriangularDistribution

Triangular probability distribution object

## Description

A `TriangularDistribution` object consists of parameters and a model description for a triangular probability distribution.

The triangular distribution is frequently used in simulations when limited sample data is available. The lower and upper limits represent the smallest and largest values, and the location of the peak represents an estimate of the mode.

The triangular distribution uses the following parameters.

Parameter	Description	Support
<i>a</i>	Lower limit	$a \leq b$
<i>b</i>	Peak location	$a \leq b \leq c$
<i>c</i>	Upper limit	$c \geq b$

## Creation

Create a `TriangularDistribution` probability distribution with specified parameter values object using `makedist`.

## Properties

### Distribution Parameters

#### **a** — Lower limit

scalar value

Lower limit for the triangular distribution, specified as a scalar value.

Data Types: `single` | `double`

**b — Peak location**

scalar value

Location of the peak for the triangular distribution, specified as a scalar value greater than or equal to `a`.

Data Types: `single` | `double`

**c — Upper limit**

scalar value

Upper limit for the triangular distribution, specified as a scalar value greater than or equal to `b`.

Data Types: `single` | `double`

## Distribution Characteristics

**IsTruncated — Logical flag for truncated distribution**

0 | 1

This property is read-only.

Logical flag for truncated distribution, specified as a logical value. If `IsTruncated` equals 0, the distribution is not truncated. If `IsTruncated` equals 1, the distribution is truncated.

Data Types: `logical`

**NumParameters — Number of parameters**

positive integer value

This property is read-only.

Number of parameters for the probability distribution, specified as a positive integer value.

Data Types: `single` | `double`

**ParameterValues — Distribution parameter values**

vector of scalar values

This property is read-only.

Distribution parameter values, specified as a vector.

Data Types: `single` | `double`

**Truncation — Truncation interval**

vector of scalar values

This property is read-only.

Truncation interval for the probability distribution, specified as a vector containing the lower and upper truncation boundaries.

Data Types: `single` | `double`

## Other Object Properties

**DistributionName — Probability distribution name**

character vector

This property is read-only.

Probability distribution name, specified as a character vector.

Data Types: `char`

**ParameterDescription — Distribution parameter descriptions**

cell array of character vectors

This property is read-only.

Distribution parameter descriptions, specified as a cell array of character vectors. Each cell contains a short description of one distribution parameter.

Data Types: `char`

**ParameterNames — Distribution parameter names**

cell array of character vectors

This property is read-only.

Distribution parameter names, specified as a cell array of character vectors.

Data Types: char

## Object Functions

<code>cdf</code>	Cumulative distribution functions
<code>icdf</code>	Inverse cumulative distribution functions
<code>iqr</code>	Interquartile range
<code>mean</code>	Mean of probability distribution
<code>median</code>	Median of probability distribution
<code>pdf</code>	Probability density functions
<code>random</code>	Random numbers
<code>std</code>	Standard deviation of probability distribution
<code>truncate</code>	Truncate probability distribution object
<code>var</code>	Variance of probability distribution

## Examples

### Create a Triangular Distribution Object Using Default Parameters

Create a triangular distribution object using the default parameter values.

```
pd = makedist('Triangular')
```

```
pd =
```

```
    TriangularDistribution
```

```
A = 0, B = 0.5, C = 1
```

### Create a Triangular Distribution Object Using Specified Parameters

Create a triangular distribution object by specifying parameter values.

```
pd = makedist('Triangular', 'a', -2, 'b', 1, 'c', 5)
```

```
pd =
```

```
    TriangularDistribution
```



$A = -2, B = 1, C = 5$

Compute the mean of the distribution.

```
m = mean(pd)
```

```
m =
```

```
1.3333
```

- “Generate Random Numbers Using the Triangular Distribution” on page 5-62

## See Also

`makedist`

## Topics

“Generate Random Numbers Using the Triangular Distribution” on page 5-62

“Triangular Distribution” on page B-178

“Nonparametric and Empirical Probability Distributions” on page 5-37

**Introduced in R2013a**

# UniformDistribution

Uniform probability distribution object

## Description

A `UniformDistribution` object consists of parameters and a model description for a uniform probability distribution.

The uniform distribution has a constant probability density function between its two parameters, `lower` (the minimum) and `upper` (the maximum). This distribution is appropriate for representing round-off errors in values tabulated to a particular number of decimal places.

The uniform distribution uses the following parameters.

Parameter	Description	Support
<code>lower</code>	Lower parameter	$-\infty < lower < upper$
<code>upper</code>	Upper parameter	$lower < upper < \infty$

## Creation

Create a `UniformDistribution` probability distribution with specified parameter values object using `makedist`.

## Properties

### Distribution Parameters

**`lower`** — Lower parameter  
scalar value

Lower parameter for the uniform distribution, specified as a scalar value.

Data Types: `single` | `double`

**upper** — Upper parameter

scalar value

Upper parameter for the uniform distribution, specified as a scalar value greater than lower.

Data Types: `single` | `double`

## Distribution Characteristics

**IsTruncated** — Logical flag for truncated distribution

0 | 1

This property is read-only.

Logical flag for truncated distribution, specified as a logical value. If `IsTruncated` equals 0, the distribution is not truncated. If `IsTruncated` equals 1, the distribution is truncated.

Data Types: `logical`

**NumParameters** — Number of parameters

positive integer value

This property is read-only.

Number of parameters for the probability distribution, specified as a positive integer value.

Data Types: `single` | `double`

**ParameterValues** — Distribution parameter values

vector of scalar values

This property is read-only.

Distribution parameter values, specified as a vector.

Data Types: `single` | `double`

**Truncation** — Truncation interval

vector of scalar values

This property is read-only.

Truncation interval for the probability distribution, specified as a vector containing the lower and upper truncation boundaries.

Data Types: `single` | `double`

## Other Object Properties

**DistributionName** — Probability distribution name

character vector

This property is read-only.

Probability distribution name, specified as a character vector.

Data Types: `char`

**ParameterDescription** — Distribution parameter descriptions

cell array of character vectors

This property is read-only.

Distribution parameter descriptions, specified as a cell array of character vectors. Each cell contains a short description of one distribution parameter.

Data Types: `char`

**ParameterNames** — Distribution parameter names

cell array of character vectors

This property is read-only.

Distribution parameter names, specified as a cell array of character vectors.

Data Types: `char`

## Object Functions

### Examples

#### Create a Uniform Distribution Object Using Default Parameters

Create a uniform distribution object using the default parameter values.

```
pd = makedist('Uniform')
```

```
pd =
```

```
UniformDistribution  
  
Uniform distribution  
Lower = 0  
Upper = 1
```

#### Create a Uniform Distribution Object Using Specified Parameters

Create a uniform distribution object by specifying parameter values.

```
pd = makedist('Uniform', 'Lower', -4, 'Upper', 2)
```

```
pd =
```

```
UniformDistribution  
  
Uniform distribution  
Lower = -4  
Upper = 2
```

Compute the interquartile range of the distribution

```
r = iqr(pd)
```

```
r =
```

3

- “Generate Random Numbers Using Uniform Distribution Inversion” on page 5-132

## See Also

`makedist`

## Topics

- “Generate Random Numbers Using Uniform Distribution Inversion” on page 5-132
- “Uniform Distribution (Continuous)” on page B-184

**Introduced in R2013a**

# WeibullDistribution

Weibull probability distribution object

## Description

A `WeibullDistribution` object consists of parameters, a model description, and sample data for a Weibull probability distribution.

The Weibull distribution is used in reliability and lifetime modeling, and to model the breaking strength of materials.

The Weibull distribution uses the following parameters.

Parameter	Description	Support
a	Scale parameter	$a > 0$
b	Shape parameter	$b > 0$

## Creation

There are several ways to create a `WeibullDistribution` probability distribution object.

- Create a distribution with specified parameter values using `makedist`.
- Fit a distribution to data using `fitdist`.
- Interactively fit a distribution to data using the **Distribution Fitter** app.

## Properties

### Distribution Parameters

**a** — Scale parameter  
positive scalar value

Scale parameter of the Weibull distribution, specified as a positive scalar value.

Data Types: `single` | `double`

**b** — **Shape parameter**

positive scalar value

Shape parameter of the Weibull distribution, specified as a positive scalar value.

Data Types: `single` | `double`

## Distribution Characteristics

**IsTruncated** — **Logical flag for truncated distribution**

0 | 1

This property is read-only.

Logical flag for truncated distribution, specified as a logical value. If `IsTruncated` equals 0, the distribution is not truncated. If `IsTruncated` equals 1, the distribution is truncated.

Data Types: `logical`

**NumParameters** — **Number of parameters**

positive integer value

This property is read-only.

Number of parameters for the probability distribution, specified as a positive integer value.

Data Types: `single` | `double`

**ParameterCovariance** — **Covariance matrix of the parameter estimates**

matrix of scalar values

This property is read-only.

Covariance matrix of the parameter estimates, specified as a  $p$ -by- $p$  matrix, where  $p$  is the number of parameters in the distribution. The  $(i,j)$  element is the covariance between the estimates of the  $i$ th parameter and the  $j$ th parameter. The  $(i,i)$  element is the estimated variance of the  $i$ th parameter. If parameter  $i$  is fixed rather than



estimated by fitting the distribution to data, then the  $(i,i)$  elements of the covariance matrix are 0.

Data Types: `single` | `double`

**ParameterIsFixed** — Logical flag for fixed parameters

array of logical values

This property is read-only.

Logical flag for fixed parameters, specified as an array of logical values. If 0, the corresponding parameter in the `ParameterNames` array is not fixed. If 1, the corresponding parameter in the `ParameterNames` array is fixed.

Data Types: `logical`

**ParameterValues** — Distribution parameter values

vector of scalar values

This property is read-only.

Distribution parameter values, specified as a vector.

Data Types: `single` | `double`

**Truncation** — Truncation interval

vector of scalar values

This property is read-only.

Truncation interval for the probability distribution, specified as a vector containing the lower and upper truncation boundaries.

Data Types: `single` | `double`

## Other Object Properties

**DistributionName** — Probability distribution name

character vector

This property is read-only.

Probability distribution name, specified as a character vector.

Data Types: `char`

**InputData** — Data used for distribution fitting  
structure

This property is read-only.

Data used for distribution fitting, specified as a structure containing the following:

- `data`: Data vector used for distribution fitting.
- `cens`: Censoring vector, or empty if none.
- `freq`: Frequency vector, or empty if none.

Data Types: `struct`

**ParameterDescription** — Distribution parameter descriptions  
cell array of character vectors

This property is read-only.

Distribution parameter descriptions, specified as a cell array of character vectors. Each cell contains a short description of one distribution parameter.

Data Types: `char`

**ParameterNames** — Distribution parameter names  
cell array of character vectors

This property is read-only.

Distribution parameter names, specified as a cell array of character vectors.

Data Types: `char`

## Object Functions

<code>cdf</code>	Cumulative distribution functions
<code>icdf</code>	Inverse cumulative distribution functions
<code>iqr</code>	Interquartile range
<code>mean</code>	Mean of probability distribution
<code>median</code>	Median of probability distribution

negloglik	Negative log likelihood of probability distribution
paramci	Confidence intervals for probability distribution parameters
pdf	Probability density functions
proflik	Profile likelihood function for probability distribution
random	Random numbers
std	Standard deviation of probability distribution
truncate	Truncate probability distribution object
var	Variance of probability distribution

## Examples

### Create a Weibull Distribution Object Using Default Parameters

Create a Weibull distribution object using the default parameter values.

```
pd = makedist('Weibull')
```

```
pd =  
  
WeibullDistribution  
  
Weibull distribution  
A = 1  
B = 1
```

### Create a Weibull Distribution Object Using Specified Parameter Values

Create a Weibull distribution object by specifying the parameter values.

```
pd = makedist('Weibull', 'a', 2, 'b', 5)
```

```
pd =  
  
WeibullDistribution  
  
Weibull distribution  
A = 2  
B = 5
```

Compute the mean of the distribution.

```
m = mean(pd)
```

```
m =
```

```
1.8363
```

- “Fit Probability Distribution Objects to Grouped Data” on page 5-121
- “Compare Multiple Distribution Fits” on page 5-114
- “Estimate Parameters of a Three-Parameter Weibull Distribution” on page B-195

## See Also

**Distribution Fitter** | `fitdist` | `makedist`

## Topics

“Fit Probability Distribution Objects to Grouped Data” on page 5-121

“Compare Multiple Distribution Fits” on page 5-114

“Estimate Parameters of a Three-Parameter Weibull Distribution” on page B-195

“Weibull Distribution” on page B-193

**Introduced in R2013a**

## union

**Class:** dataset

Set union for dataset array observations

---

**Note** The `dataset` data type might be removed in a future release. To work with heterogeneous data, use the MATLAB® `table` data type instead. See MATLAB `table` documentation for more information.

---

## Syntax

```
C = union(A,B)
C = union(A,B,vars)
C = union(A,B,vars,setOrder)
[C,iA,iB] = union(____)
```

## Description

`C = union(A,B)` for dataset arrays `A` and `B` returns the combined set of observations from the two arrays, with repetitions removed. The observations in the dataset array `C` are sorted.

`C = union(A,B,vars)` returns the combined set of observations from the two arrays, with repetitions of unique combinations of the variables specified in `vars` removed. The observations in the dataset array `C` are sorted by those variables.

The values for variables not specified in `vars` for each observation in `C` are taken from the corresponding observation in `A` or `B`, or from `A` if there are common observations in both `A` and `B`. If there are multiple observations in `A` or `B` that correspond to an observation in `C`, those values are taken from the first occurrence.

`C = union(A,B,vars,setOrder)` returns the observations in `C` in the order specified by `setOrder`.

`[C, iA, iB] = union( ___ )` also returns index vectors `iA` and `iB` such that `C` is a sorted combination of the values `A(iA, :)` and `B(iB, :)`. If there are common observations in `A` and `B`, then `union` returns only the index from `A`, in `iA`. If there are repeated observations in `A` or `B`, then the index of the first occurrence is returned. You can use any of the previous input arguments.

## Input Arguments

### **A, B**

Input dataset arrays.

### **vars**

Cell array of character vectors containing variable names or a vector of integers containing variable column numbers, indicating the variables for which `union` removes repetitions of unique combinations of the variables.

Specify `vars` as `[]` to use its default value of all variables.

### **setOrder**

Flag indicating the sorting order for the observations in `C`. The possible values of `setOrder` are:

'sorted'	Observations in <code>C</code> are in sorted order (default).
'stable'	Observations in <code>C</code> are in the same order that they appear in <code>A</code> , then <code>B</code> .

## Output Arguments

### **C**

Dataset array with the combined observations of `A` and `B`, with repetitions removed. `C` is in sorted order (by default), or the order specified by `setOrder`.

**iA**

Index vector, indicating the observations in A that contribute to the union. `iA` contains the index to the first occurrence of any repeated observations in A.

**iB**

Index vector, indicating the observations in B that contribute to the union. If there are common observations in A and B, then `union` returns only the index from A, in `iA`. `iB` contains the index to the first occurrence of any repeated observations in B.

## Examples

### Union of Two Dataset Arrays

Navigate to the folder containing sample data, and load sample data.

```
cd(matlabroot)
cd('help/toolbox/stats/examples')

A = dataset('XLSFile','hospitalSmall.xlsx');
B = dataset('XLSFile','hospitalSmall.xlsx','Sheet',2);
[length(A) length(B)]

ans =

    14     8
```

The first dataset array, A, has 14 observations. The second dataset array, B, has 8 observations.

Return the union.

```
C = union(A,B);
length(C)

ans =

    21
```

The union of the two dataset arrays has 21 observations, indicating that there was one observation replicated in A and B.

- “Create a Dataset Array from a File” on page 2-77
- “Merge Dataset Arrays” on page 2-107

## See Also

`dataset` | `intersect` | `ismember` | `setdiff` | `setxor` | `sortrows` | `unique`

## Topics

“Create a Dataset Array from a File” on page 2-77

“Merge Dataset Arrays” on page 2-107

“Dataset Arrays” on page 2-142



# unique

**Class:** dataset

Unique observations in dataset array

---

**Note** The `dataset` data type might be removed in a future release. To work with heterogeneous data, use the MATLAB® `table` data type instead. See MATLAB `table` documentation for more information.

---

## Syntax

```
C = unique(A)
[C,ia,ic] = unique(A)
C = unique(A,vars)
[C,ia,ic] = unique(A,vars)
[...] = unique(A,vars,occurrence)
[...] = unique(...,'R2012a')
[...] = unique(...,'legacy')
[...] = unique(A,vars,setOrder)
```

## Description

---

**Note** The behavior of `dataset.unique` is consistent with the MATLAB function `unique`. For a demonstration of using the `'legacy'` flag to preserve the behavior from R2012b and prior in your existing code, see the documentation for `unique`.

---

`C = unique(A)` returns a copy of the dataset `A` that contains only the sorted unique observations. `A` must contain only variables whose class has a `unique` method, including:

- numeric
- character

- logical
- categorical
- cell arrays of character vectors

For a variable with multiple columns, its class's unique method must support the 'rows' flag.

`[C,ia,ic] = unique(A)` also returns index vectors `ia` and `ic` such that `C = A(ia,:)` and `A = C(ic,:)`.

`C = unique(A,vars)` returns a dataset that contains only one observation for each unique combination of values for the variables in `A` specified in `vars`. `vars` is a positive integer, a vector of positive integers, a variable name, a cell array containing one or more variable names, or a logical vector. `C` includes all variables from `A`. The values in `C` for the variables not specified in `vars` are taken from the last occurrence among observations in `A` with each unique combination of values for the variables specified in `vars`.

`[C,ia,ic] = unique(A,vars)` also returns index vectors `ia` and `ic` such that `C = A(ia,:)` and `A(:,vars) = C(ic,vars)`.

`[...] = unique(A,vars,occurrence)` specifies which index is returned in `ia` in the case of repeated observations in `A`. The default value is `occurrence='first'`, which returns the index of the first occurrence of each repeated observation in `A`. `occurrence='last'` returns the index of the last occurrence of each repeated observation in `A`. The values in `C` for variables not specified in `vars` are taken from the observations `A(ia,:)`. Specify `vars` as `[]` to use the default value of all variables.

`[...] = unique(...,'R2012a')` adopts the future behavior of `unique`. You can specify the flag as the final argument with any previous syntax that accepts `A`, `vars`, or `occurrence`.

`[...] = unique(...,'legacy')` preserves the current behavior of `unique`. You can specify the flag as the final argument with any previous syntax that accepts `A`, `vars`, or `occurrence`.

`[...] = unique(A,vars,setOrder)` returns the observations of `C` in a specific order. `setOrder='sorted'` returns the values of `C` in sorted order. `setOrder='stable'` returns the values of `C` in the same order as `A`. If there are repeated observations in `A`, then `ia` returns the index of the first occurrence of each repeated observation. Specify `vars` as `[]` to use the default value of all variables.

## See Also

dataset | set | subsasgn

## Units property

**Class:** dataset

Units of variables in data set

---

**Note** The `dataset` data type might be removed in a future release. To work with heterogeneous data, use the MATLAB® `table` data type instead. See MATLAB `table` documentation for more information.

---

## Description

A cell array of character vectors giving the units of the variables in the data set. This property may be empty, but if not empty, the number of character vectors must equal the number of variables. Any individual character vector may be empty for a variable that does not have units defined. The default is an empty cell array.

# unidcdf

Discrete uniform cumulative distribution function

## Syntax

```
p = unidcdf(x,N)
p = unidcdf(x,N,'upper')
```

## Description

`p = unidcdf(x,N)` returns the discrete uniform cdf at each value in `x` using the corresponding maximum observable value in `N`. `x` and `N` can be vectors, matrices, or multidimensional arrays that have the same size. A scalar input is expanded to a constant array with the same dimensions as the other inputs. The maximum observable values in `N` must be positive integers.

`p = unidcdf(x,N,'upper')` returns the complement of the discrete uniform cdf at each value in `x`, using an algorithm that more accurately computes the extreme upper tail probabilities.

The discrete uniform cdf is

$$p = F(x | N) = \frac{\text{floor}(x)}{N} I_{(1, \dots, N)}(x)$$

The result,  $p$ , is the probability that a single observation from the discrete uniform distribution with maximum  $N$  will be a positive integer less than or equal to  $x$ . The values  $x$  do not need to be integers.

## Examples

**Compute Discrete Uniform Distribution `cdf`**

What is the probability of drawing a number 20 or less from a hat with the numbers from 1 to 50 inside?

```
probability = unidcdf(20,50)
```

```
probability =
```

```
    0.4000
```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`cdf` | `mle` | `unidinv` | `unidpdf` | `unidrnd` | `unidstat`

### Topics

“Uniform Distribution (Discrete)” on page B-190

**Introduced before R2006a**

## unidinv

Discrete uniform inverse cumulative distribution function

### Syntax

```
X = unidinv(P,N)
```

### Description

`X = unidinv(P,N)` returns the smallest positive integer  $X$  such that the discrete uniform cdf evaluated at  $X$  is equal to or exceeds  $P$ . You can think of  $P$  as the probability of drawing a number as large as  $X$  out of a hat with the numbers 1 through  $N$  inside.

$P$  and  $N$  can be vectors, matrices, or multidimensional arrays that have the same size, which is also the size of  $X$ . A scalar input for  $N$  or  $P$  is expanded to a constant array with the same dimensions as the other input. The values in  $P$  must lie on the interval  $[0\ 1]$  and the values in  $N$  must be positive integers.

### Examples

```
x = unidinv(0.7,20)
x =
    14
```

```
y = unidinv(0.7 + eps,20)
y =
    15
```

A small change in the first parameter produces a large jump in output. The cdf and its inverse are both step functions. The example shows what happens at a step.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`icdf` | `unidcdf` | `unidpdf` | `unidrnd` | `unidstat`

### Topics

“Uniform Distribution (Discrete)” on page B-190

Introduced before R2006a



# unidpdf

Discrete uniform probability density function

## Syntax

```
Y = unidpdf(X,N)
```

## Description

`Y = unidpdf(X,N)` computes the discrete uniform pdf at each of the values in `X` using the corresponding maximum observable value in `N`. `X` and `N` can be vectors, matrices, or multidimensional arrays that have the same size. A scalar input is expanded to a constant array with the same dimensions as the other inputs. The parameters in `N` must be positive integers.

The discrete uniform pdf is

$$y = f(x | N) = \frac{1}{N} I_{(1, \dots, N)}(x)$$

You can think of `y` as the probability of observing any one number between 1 and `n`.

## Examples

For fixed `n`, the uniform discrete pdf is a constant.

```
y = unidpdf(1:6,10)
y =
    0.1000    0.1000    0.1000    0.1000    0.1000    0.1000
```

Now fix `x`, and vary `n`.

```
likelihood = unidpdf(5,4:9)
likelihood =
    0    0.2000    0.1667    0.1429    0.1250    0.1111
```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`pdf` | `unidcdf` | `unidinv` | `unidrnd` | `unidstat`

### Topics

“Uniform Distribution (Discrete)” on page B-190

Introduced before R2006a

# unidrnd

Discrete uniform random numbers

## Syntax

```
R = unidrnd(N)
R = unidrnd(N,m,n,...)
R = unidrnd(N,[m,n,...])
```

## Description

`R = unidrnd(N)` generates random numbers for the discrete uniform distribution with maximum `N`. The parameters in `N` must be positive integers. `N` can be a vector, a matrix, or a multidimensional array. The size of `R` is the size of `N`. The discrete uniform distribution arises from experiments equivalent to drawing a number from one to `N` out of a hat.

`R = unidrnd(N,m,n,...)` or `R = unidrnd(N,[m,n,...])` generates an `m`-by-`n`-by-... array. The `N` parameter can be a scalar or an array of the same size as `R`.

## Examples

In the Massachusetts lottery, a player chooses a four-digit number. Generate random numbers for Monday through Saturday.

```
numbers = unidrnd(10000,1,6)-1
numbers =
    4564    185    8214    4447    6154    7919
```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

The generated code can return a different sequence of numbers than MATLAB if either of the following is true:

- The output is nonscalar.
- An input parameter is invalid for the distribution.

### See Also

`random` | `unidcdf` | `unidinv` | `unidpdf` | `unidstat`

### Topics

“Uniform Distribution (Discrete)” on page B-190

**Introduced before R2006a**

## unidstat

Discrete uniform mean and variance

### Syntax

```
[M,V] = unidstat(N)
```

### Description

`[M,V] = unidstat(N)` returns the mean and variance of the discrete uniform distribution with minimum value 1 and maximum value  $N$ .

The mean of the discrete uniform distribution with parameter  $N$  is  $(N + 1)/2$ . The variance is  $(N^2 - 1)/12$ .

### Examples

```
[m,v] = unidstat(1:6)
m =
    1.0000    1.5000    2.0000    2.5000    3.0000    3.5000
v =
    0    0.2500    0.6667    1.2500    2.0000    2.9167
```

### Extended Capabilities

#### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

#### See Also

`unidcdf` | `unidinv` | `unidpdf` | `unidrnd`

## **Topics**

“Uniform Distribution (Discrete)” on page B-190

**Introduced before R2006a**

# unifcdf

Continuous uniform cumulative distribution function

## Syntax

```
p = unifcdf(x,a,b)
p = unifcdf(x,a,b,'upper')
```

## Description

`p = unifcdf(x,a,b)` returns the uniform cdf at each value in `x` using the corresponding lower endpoint (minimum), `a` and upper endpoint (maximum), `b`. `x`, `a`, and `b` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant matrix with the same dimensions as the other inputs.

`p = unifcdf(x,a,b,'upper')` returns the complement of the uniform cdf at each value in `x`, using an algorithm that more accurately computes the extreme upper tail probabilities.

The uniform cdf is

$$p = F(x | a, b) = \frac{x - a}{b - a} I_{[a, b]}(x)$$

The standard uniform distribution has `a = 0` and `b = 1`.

## Examples

### Compute Uniform Distribution cdf

What is the probability that an observation from a standard uniform distribution will be less than 0.75?

```
probability = unifcdf(0.75)
```

```
probability =
```

```
    0.7500
```

What is the probability that an observation from a uniform distribution with  $a = -1$  and  $b = 1$  will be less than 0.75?

```
probability = unifcdf(0.75,-1,1)
```

```
probability =
```

```
    0.8750
```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

[cdf](#) | [unifinv](#) | [unifit](#) | [unifpdf](#) | [unifrnd](#) | [unifstat](#)

### Topics

“Uniform Distribution (Continuous)” on page B-184

**Introduced before R2006a**



## unifinv

Continuous uniform inverse cumulative distribution function

### Syntax

```
X = unifinv(P,A,B)
```

### Description

`X = unifinv(P,A,B)` computes the inverse of the uniform cdf with parameters A and B (the minimum and maximum values, respectively) at the corresponding probabilities in P. P, A, and B can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array with the same dimensions as the other inputs.

The inverse of the uniform cdf is

$$x = F^{-1}(p | a, b) = a + p(a - b)I_{[0,1]}(p)$$

The standard uniform distribution has  $A = 0$  and  $B = 1$ .

### Examples

What is the median of the standard uniform distribution?

```
median_value = unifinv(0.5)
median_value =
    0.5000
```

What is the 99th percentile of the uniform distribution between -1 and 1?

```
percentile = unifinv(0.99, -1, 1)
percentile =
    0.9800
```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`icdf` | `unifcdf` | `unifit` | `unifpdf` | `unifrnd` | `unifstat`

### Topics

“Uniform Distribution (Continuous)” on page B-184

Introduced before R2006a

# unifit

Continuous uniform parameter estimates

## Syntax

```
[ahat,bhat] = unifit(data)
[ahat,bhat,ACI,BCI] = unifit(data)
[ahat,bhat,ACI,BCI] = unifit(data,alpha)
```

## Description

`[ahat,bhat] = unifit(data)` returns the maximum likelihood estimates (MLEs) of the parameters of the uniform distribution given the data in `data`.

`[ahat,bhat,ACI,BCI] = unifit(data)` also returns 95% confidence intervals, `ACI` and `BCI`, which are matrices with two rows. The first row contains the lower bound of the interval for each column of the matrix `data`. The second row contains the upper bound of the interval.

`[ahat,bhat,ACI,BCI] = unifit(data,alpha)` enables you to control of the confidence level `alpha`. For example, if `alpha = 0.01` then `ACI` and `BCI` are 99% confidence intervals.

## Examples

```
r = unifrnd(10,12,100,2);
[ahat,bhat,aci,bci] = unifit(r)
ahat =
    10.0154    10.0060
bhat =
    11.9989    11.9743
aci =
     9.9551     9.9461
    10.0154    10.0060
bci =
```

11.9989 11.9743  
12.0592 12.0341

## See Also

mle | unifcdf | unifinv | unifpdf | unifrnd | unifstat

## Topics

“Uniform Distribution (Continuous)” on page B-184

**Introduced before R2006a**

# unifpdf

Continuous uniform probability density function

## Syntax

```
Y = unifpdf(X,A,B)
```

## Description

`Y = unifpdf(X,A,B)` computes the continuous uniform pdf at each of the values in `X` using the corresponding lower endpoint (minimum), `A` and upper endpoint (maximum), `B`. `X`, `A`, and `B` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array with the same dimensions as the other inputs. The parameters in `B` must be greater than those in `A`.

The continuous uniform distribution pdf is

$$y = f(x | a, b) = \frac{1}{b - a} I_{[a, b]}(x)$$

The standard uniform distribution has `A = 0` and `B = 1`.

## Examples

For fixed `a` and `b`, the uniform pdf is constant.

```
x = 0.1:0.1:0.6;
y = unifpdf(x)
y =
    1    1    1    1    1    1
```

What if `x` is not between `a` and `b`?

```
y = unifpdf(-1,0,1)
y =
    0
```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`pdf` | `unifcdf` | `unifinv` | `unifit` | `unifrnd` | `unifstat`

### Topics

“Uniform Distribution (Continuous)” on page B-184

Introduced before R2006a

# unifrnd

Continuous uniform random numbers

## Syntax

```
R = unifrnd(A,B)
R = unifrnd(A,B,m,n,...)
R = unifrnd(A,B,[m,n,...])
```

## Description

`R = unifrnd(A,B)` returns an array `R` of random numbers generated from the continuous uniform distributions with lower and upper endpoints specified by `A` and `B`, respectively. If `A` and `B` are arrays, `R(i,j)` is generated from the distribution specified by the corresponding elements of `A` and `B`. If either `A` or `B` is a scalar, it is expanded to the size of the other input.

`R = unifrnd(A,B,m,n,...)` or `R = unifrnd(A,B,[m,n,...])` returns an `m`-by-`n`-by-... array. If `A` and `B` are scalars, all elements of `R` are generated from the same distribution. If either `A` or `B` is an array, they must be `m`-by-`n`-by-... .

## Examples

Generate one random number each from the continuous uniform distributions on the intervals (0,1), (0,2), ..., (0,5):

```
a = 0; b = 1:5;
r1 = unifrnd(a,b)
r1 =
    0.8147    1.8116    0.3810    3.6535    3.1618
```

Generate five random numbers each from the same distributions:

```
B = repmat(b,5,1);
R = unifrnd(a,B)
```

```
R =  
    0.0975    0.3152    0.4257    2.6230    3.7887  
    0.2785    1.9412    1.2653    0.1428    3.7157  
    0.5469    1.9143    2.7472    3.3965    1.9611  
    0.9575    0.9708    2.3766    3.7360    3.2774  
    0.9649    1.6006    2.8785    2.7149    0.8559
```

Generate five random numbers from the continuous uniform distribution on (0,2):

```
r2 = unifrnd(a,b(2),1,5)  
r2 =  
    1.4121    0.0637    0.5538    0.0923    0.1943
```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

The generated code can return a different sequence of numbers than MATLAB if either of the following is true:

- The output is nonscalar.
- An input parameter is invalid for the distribution.

### See Also

[rand](#) | [random](#) | [unifcdf](#) | [unifinv](#) | [unifit](#) | [unifpdf](#) | [unifstat](#)

### Topics

“Uniform Distribution (Continuous)” on page B-184

**Introduced before R2006a**



# unifstat

Continuous uniform mean and variance

## Syntax

```
[M,V] = unifstat(A,B)
```

## Description

`[M,V] = unifstat(A,B)` returns the mean of and variance for the continuous uniform distribution using the corresponding lower endpoint (minimum), `A` and upper endpoint (maximum), `B`. Vector or matrix inputs for `A` and `B` must have the same size, which is also the size of `M` and `V`. A scalar input for `A` or `B` is expanded to a constant matrix with the same dimensions as the other input.

The mean of the continuous uniform distribution with parameters  $a$  and  $b$  is  $(a + b)/2$ , and the variance is  $(a - b)^2/12$ .

## Examples

```
a = 1:6;  
b = 2.*a;  
[m,v] = unifstat(a,b)  
m =  
    1.5000    3.0000    4.5000    6.0000    7.5000    9.0000  
v =  
    0.0833    0.3333    0.7500    1.3333    2.0833    3.0000
```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`unifcdf` | `unifinv` | `unifit` | `unifpdf` | `unifrnd`

### Topics

“Uniform Distribution (Continuous)” on page B-184

Introduced before R2006a

# unstack

**Class:** dataset

Unstack data from single variable into multiple variables

---

**Note** The `dataset` data type might be removed in a future release. To work with heterogeneous data, use the MATLAB® `table` data type instead. See MATLAB `table` documentation for more information.

---

## Syntax

```
A = unstack(B,datavar,indvar)
[A,iB] = unstack(B,datavar,indvar)
A = unstack(B,datavar,indvar,'Parameter',value)
```

## Description

`A = unstack(B,datavar,indvar)` unstacks a single variable in dataset array `B` into multiple variables in `A`. In general `A` contains more variables, but fewer observations, than `B`.

`datavar` specifies the data variable in `B` to unstack. `indvar` specifies an indicator variable in `B` that determines which variable in `A` each value in `datavar` is unstacked into. `unstack` treats the remaining variables in `B` as grouping variables. Each unique combination of their values defines a group of observations in `B` that will be unstacked into a single observation in `A`.

`unstack` creates `m` data variables in `A`, where `m` is the number of group levels in `indvar`. The values in `indvar` indicate which of those `m` variables receive which values from `datavar`. The `j`-th data variable in `A` contains the values from `datavar` that correspond to observations whose `indvar` value was the `j`-th of the `m` possible levels. Elements of those `m` variables for which no corresponding data value in `B` exists contain a default value.

`datavar` is a positive integer, a variable name, or a logical vector containing a single true value. `indvar` is a positive integer, a variable name, or a logical vector containing a single true value.

`[A, iB] = unstack(B, datavar, indvar)` returns an index vector `iB` indicating the correspondence between observations in `A` and those in `B`. For each observation in `A`, `iB` contains the index of the first in the corresponding group of observations in `B`.

For more information on grouping variables, see “Grouping Variables” on page 2-59.

## Input Arguments

`A = unstack(B, datavar, indvar, 'Parameter', value)` uses the following parameter name/value pairs to control how `unstack` converts variables in `B` to variables in `A`:

'GroupVars '	Grouping variables in <code>B</code> that define groups of observations. <code>groupvars</code> is a positive integer, a vector of positive integers, a variable name, a cell array containing one or more variable names, or a logical vector. The default is all variables in <code>B</code> not listed in <code>datavar</code> or <code>indvar</code> .
'NewDataVarNames '	A cell array of character vectors containing names for the data variables <code>unstack</code> should create in <code>A</code> . Default is the group names of the grouping variable specified in <code>indvar</code> .
'AggregationFun '	A function handle that accepts a subset of values from <code>datavar</code> and returns a single value. <code>stack</code> applies this function to observations from the same group that have the same value of <code>indvar</code> . The function must aggregate the data values into a single value, and in such cases it is not possible to recover <code>B</code> from <code>A</code> using <code>stack</code> . The default is <code>@sum</code> for numeric data variables. For non-numeric variables, there is no default, and you must specify <code>'AggregationFun '</code> if multiple observations in the same group have the same values of <code>indvar</code> .

'ConstVars'	Variables in B to copy to A without unstacking. The values for these variables in A are taken from the first observation in each group in B, so these variables should typically be constant within each group. ConstVars is a positive integer, a vector of positive integers, a variable name, a cell array containing one or more variable names, or a logical vector. The default is no variables.
-------------	--

You can also specify more than one data variable in B, each of which becomes a set of  $m$  variables in A. In this case, specify `datavar` as a vector of positive integers, a cell array containing variable names, or a logical vector. You may specify only one variable with `indvar`. The names of each set of data variables in A are the name of the corresponding data variable in B concatenated with the names specified in `'NewDataVarNames'`. The function specified in `'AggregationFun'` must return a value with a single row.

## Examples

Combine several variables for estimated influenza rates into a single variable. Then unstack the estimated influenza rates by date.

```
load flu

% FLU has a 'Date' variable, and 10 variables for estimated influenza rates
% (in 9 different regions, estimated from Google searches, plus a
% nationwide estimate from the CDC). Combine those 10 variables into an
% array that has a single data variable, 'FluRate', and an indicator
% variable, 'Region', that says which region each estimate is from.
[flu2,iflu] = stack(flu, 2:11, 'NewDataVarName','FluRate', ...
    'IndVarName','Region')

% The second observation in FLU is for 10/16/2005. Find the observations
% in FLU2 that correspond to that date.
flu(2,:)
flu2(iflu==2,:)

% Use the 'Date' variable from that array to split 'FluRate' into 52
% separate variables, each containing the estimated influenza rates for
% each unique date. The new array has one observation for each region. In
% effect, this is the original array FLU "on its side".
dateNames = cellstr(datestr(flu.Date,'mmm DD YYYY'));
[flu3,iflu2] = unstack(flu2, 'FluRate', 'Date', ...
```

```
'NewDataVarNames',dateNames)  
  
% Since observations in FLU3 represent regions, IFLU2 indicates the first  
% occurrence in FLU2 of each region.  
flu2(iflu2,:)
```

## See Also

`dataset.join` | `dataset.stack`

## Topics

“Grouping Variables” on page 2-59

# upperparams

**Class:** paretotails

Upper Pareto tails parameters

## Syntax

```
params = upperparams(obj)
```

## Description

`params = upperparams(obj)` returns the 2-element vector `params` of shape and scale parameters, respectively, of the upper tail of the Pareto tails object `obj`. `upperparams` does not return a location parameter.

## Examples

Fit Pareto tails to a  $t$  distribution at cumulative probabilities 0.1 and 0.9:

```
t = trnd(3,100,1);  
obj = paretotails(t,0.1,0.9);
```

```
lowerparams(obj)  
ans =  
    -0.1901    1.1898  
upperparams(obj)  
ans =  
    0.3646    0.5103
```

## See Also

`lowerparams` | `paretotails`

## UserData property

**Class:** dataset

Variable containing additional information associated with data set

---

**Note** The `dataset` data type might be removed in a future release. To work with heterogeneous data, use the MATLAB® `table` data type instead. See MATLAB `table` documentation for more information.

---

## Description

Any variable containing additional information to be associated with the data set. The default is an empty array.



## var

Variance of probability distribution

## Syntax

```
v = var(pd)
```

## Description

`v = var(pd)` returns the variance `v` of the probability distribution `pd`.

## Examples

### Variance of a Fitted Distribution

Load the sample data. Create a vector containing the first column of students' exam grade data.

```
load examgrades  
x = grades(:,1);
```

Fit a normal distribution object to the data.

```
pd = fitdist(x, 'Normal')
```

```
pd =
```

```
NormalDistribution
```

```
Normal distribution
```

```
mu = 75.0083 [73.4321, 76.5846]  
sigma = 8.7202 [7.7391, 9.98843]
```

Compute the variance of the fitted distribution.

```
v = var(pd)
```

```
v =
```

```
76.0419
```

For a normal distribution, the variance is equal to the square of the parameter  $\sigma$ .

### Variance of a Skewed Distribution

Create a Weibull probability distribution object.

```
pd = makedist('Weibull', 'a', 5, 'b', 2)
```

```
pd =
```

```
WeibullDistribution
```

```
Weibull distribution
```

```
A = 5
```

```
B = 2
```

Compute the variance of the distribution.

```
v = var(pd)
```

```
v =
```

```
5.3650
```

## Input Arguments

**pd** — Probability distribution

probability distribution object

Probability distribution, specified as a probability distribution object. Create a probability distribution object with specified parameter values using `makedist`. Alternatively, create a probability distribution object by fitting it to data using `fitdist` or the Distribution Fitter app.

## Output Arguments

### **v** — Variance

nonnegative scalar value

Variance of the probability distribution, returned as a nonnegative scalar value.

## See Also

`distributionFitter` | `fitdist` | `makedist`

## Topics

“Working with Probability Distributions” on page 5-3

Introduced before R2006a

## var

**Class:** ProbDistUnivParam

Return variance of ProbDistUnivParam object

## Syntax

$V = \text{var}(PD)$

## Description

$V = \text{var}(PD)$  returns  $V$ , the variance of the ProbDistUnivParam object  $PD$ .

## Input Arguments

$PD$  An object of the class ProbDistUnivParam.

## Output Arguments

$V$  The variance of the ProbDistUnivParam object  $PD$ .

## See Also

var

## var

**Class:** prob.ParametricTruncatableDistribution

**Package:** prob

Variance of probability distribution object

## Syntax

```
v = var(pd)
```

## Description

`v = var(pd)` returns the variance `v` of the probability distribution `pd`.

## Input Arguments

**pd — Probability distribution**

probability distribution object

Probability distribution, specified as a probability distribution object. Create a probability distribution object with specified parameter values using `makedist`.

## Output Arguments

**v — Variance**

nonnegative scalar value

Variance of the probability distribution, returned as a nonnegative scalar value.

## Examples

### Variance of a Triangular Distribution

Create a triangular distribution object.

```
pd = makedist('Triangular', 'a', -3, 'b', 1, 'c', 3)
```

```
pd =
```

```
    TriangularDistribution
```

```
A = -3, B = 1, C = 3
```

Compute the variance of the distribution.

```
v = var(pd)
```

```
v =
```

```
    1.5556
```

## See Also

`makedist`

## var

**Class:** prob.ToolboxFittableParametricDistribution

**Package:** prob

Variance of probability distribution object

## Syntax

`v = var(pd)`

## Description

`v = var(pd)` returns the variance `v` of the probability distribution `pd`.

## Input Arguments

**pd** — **Probability distribution**

probability distribution object

Probability distribution, specified as a probability distribution object. Create a probability distribution object with specified parameter values using `makedist`. Alternatively, create a probability distribution object by fitting it to data using `fitdist` or the Distribution Fitter app.

## Output Arguments

**v** — **Variance**

nonnegative scalar value

Variance of the probability distribution, returned as a nonnegative scalar value.

## Examples

### Variance of a Fitted Distribution

Load the sample data. Create a vector containing the first column of students' exam grade data.

```
load examgrades;  
x = grades(:,1);
```

Fit a normal distribution object to the data.

```
pd = fitdist(x, 'Normal')  
  
pd =  
  
NormalDistribution  
  
Normal distribution  
    mu = 75.0083    [73.4321, 76.5846]  
    sigma = 8.7202    [7.7391, 9.98843]
```

Compute the variance of the fitted distribution.

```
v = var(pd)
```

```
v =  
  
76.0419
```

For a normal distribution, the variance is equal to the square of the parameter `sigma`.

### Variance of a Skewed Distribution

Create a Weibull probability distribution object.

```
pd = makedist('Weibull', 'a', 5, 'b', 2)
```



```
pd =  
  
WeibullDistribution  
  
Weibull distribution  
  A = 5  
  B = 2
```

Compute the variance of the distribution.

```
v = var(pd)
```

```
v =  
  
5.3650
```

## See Also

[distributionFitter](#) | [fitdist](#) | [makedist](#)

## VarDescription property

**Class:** dataset

Cell array of character vectors giving descriptions of variables in data set

---

**Note** The `dataset` data type might be removed in a future release. To work with heterogeneous data, use the MATLAB® `table` data type instead. See MATLAB `table` documentation for more information.

---

## Description

A cell array of character vectors giving the descriptions of the variables in the data set. This property may be empty, but if not empty, the number of character vectors must equal the number of variables. Any individual character vector may be empty for a variable that does not have a description defined. The default is an empty cell array.

# varimportance

**Class:** classregtree

Compute embedded estimates of input feature importance

---

**Note** `classregtree` will be removed in a future release. See `fitctree`, `fitrtree`, `ClassificationTree`, or `RegressionTree` instead.

---

## Syntax

```
imp = varimportance(t)
```

## Description

`imp = varimportance(t)` computes estimates of input feature importance for tree `t` by summing changes in the risk due to splits on every feature. The returned vector `imp` has one element for each input variable in the data used to train this tree. At each node, the risk is estimated as node impurity if impurity was used to split nodes and node error otherwise. This risk is weighted by the node probability. Variable importance associated with this split is computed as the difference between the risk for the parent node and the total risk for the two children.

## See Also

`risk`

## VarNames property

**Class:** dataset

Cell array giving names of variables in data set

---

**Note** The `dataset` data type might be removed in a future release. To work with heterogeneous data, use the MATLAB® `table` data type instead. See MATLAB `table` documentation for more information.

---

## Description

A cell array of nonempty, distinct character vectors giving the names of the variables in the data set. The number of character vectors must equal the number of variables. The default is the cell array of names for the variables used to create the data set.

## VarNames property

**Class:** TreeBagger

Variable names

### Description

The `VarNames` property is a cell array containing the names of the predictor variables (features). `TreeBagger` takes these names from the optional `'names'` parameter. The default names are `'x1'`, `'x2'`, etc.

## vartest

Chi-square variance test

### Syntax

```
h = vartest(x,v)
h = vartest(x,v,Name,Value)
[h,p] = vartest(____)
[h,p,ci,stats] = vartest(____)
```

### Description

`h = vartest(x,v)` returns a test decision for the null hypothesis that the data in vector `x` comes from a normal distribution with variance `v`, using the chi-square variance test on page 32-6901. The alternative hypothesis is that `x` comes from a normal distribution with a different variance. The result `h` is 1 if the test rejects the null hypothesis at the 5% significance level, and 0 otherwise.

`h = vartest(x,v,Name,Value)` performs the chi-square variance test with additional options specified by one or more name-value pair arguments. For example, you can change the significance level or conduct a one-sided test.

`[h,p] = vartest(____)` also returns the  $p$ -value of the test, `p`, using any of the input arguments in the previous syntaxes.

`[h,p,ci,stats] = vartest(____)` also returns the confidence interval for the true variance, `ci`, and the structure `stats` containing information about the test statistic.

### Examples

## Chi-Squared Test for Specified Variance

Load the sample data. Create a vector containing the first column of the students' exam grades matrix.

```
load examgrades
x = grades(:,1);
```

Test the null hypothesis that the data comes from a distribution with a variance of 25.

```
[h,p,ci,stats] = vartest(x,25)
```

```
h =
```

```
    1
```

```
p =
```

```
    0
```

```
ci =
```

```
    59.8936
```

```
    99.7688
```

```
stats =
```

```
    struct with fields:
```

```
    chisqstat: 361.9597
```

```
    df: 119
```

The returned value  $h = 1$  indicates that `vartest` rejects the null hypothesis at the default 5% significance level. `ci` shows the lower and upper boundaries of the 95% confidence interval for the true variance, and suggests that the true variance is greater than 25.

### Chi-Squared Test Using One-Sided Hypothesis

Load the sample data. Create a vector containing the first column of the students' exam grades matrix.

```
load examgrades
x = grades(:,1);
```

Test the null hypothesis that the data comes from a distribution with a variance of 25, against the alternative hypothesis that the variance is greater than 25.

```
[h,p] = vartest(x,25,'Tail','right')
```

```
h =
```

```
    1
```

```
p =
```

```
 2.4269e-26
```

The returned value of `h = 1` indicates that `vartest` rejects the null hypothesis at the default 5% significance level, in favor of the alternative hypothesis that the variance is greater than 25.

## Input Arguments

### **x** — Sample data

vector | matrix | multidimensional array

Sample data, specified as a vector, matrix, or multidimensional array. For matrices, `vartest` performs separate tests along each column of `x`, and returns a row vector of results. For multidimensional arrays on page 32-6901, `vartest` works along the first nonsingleton dimension on page 32-6901 of `x`.

Data Types: `single` | `double`

### **v** — Hypothesized variance

nonnegative scalar value



Hypothesized variance, specified as a nonnegative scalar value.

Data Types: `single` | `double`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'Tail', 'right', 'Alpha', 0.01` specifies a right-tailed hypothesis test at the 1% significance level.

### **Alpha** — Significance level

`0.05` (default) | scalar value in the range (0,1)

Significance level of the hypothesis test, specified as the comma-separated pair consisting of `'Alpha'` and a scalar value in the range (0,1).

Example: `'Alpha', 0.01`

Data Types: `single` | `double`

### **Dim** — Dimension

first nonsingleton dimension (default) | positive integer value

Dimension of the input matrix to test along, specified as the comma-separated pair consisting of `'Dim'` and a positive integer value. For example, specifying `'Dim', 1` tests the data in each column for equality to the hypothesized variance, while `'Dim', 2` tests the data in each row.

Example: `'Dim', 2`

Data Types: `single` | `double`

### **Tail** — Type of alternative hypothesis

`'both'` (default) | `'right'` | `'left'`

Type of alternative hypothesis to evaluate, specified as the comma-separated pair consisting of `'Tail'` and one of the following.

'both'	Test the alternative hypothesis that the population variance is not $v$ .
'right'	Test the alternative hypothesis that the population variance is greater than $v$ .
'left'	Test the alternative hypothesis that the population variance is less than $v$ .

Example: 'Tail', 'right'

## Output Arguments

### **h** — Hypothesis test result

1 | 0

Hypothesis test result, returned as a logical value.

- If  $h = 1$ , this indicates the rejection of the null hypothesis at the Alpha significance level.
- If  $h = 0$ , this indicates a failure to reject the null hypothesis at the Alpha significance level.

### **p** — *p*-value

scalar value in the range [0,1]

*p*-value of the test, returned as a scalar value in the range [0,1]. *p* is the probability of observing a test statistic as extreme as, or more extreme than, the observed value under the null hypothesis. Small values of *p* cast doubt on the validity of the null hypothesis.

### **ci** — Confidence interval

vector

Confidence interval for the true variance, returned as a two-element vector containing the lower and upper boundaries of the  $100 \times (1 - \text{Alpha})\%$  confidence interval.

### **stats** — Test statistics

structure

Test statistics for the chi-square variance test, returned as a structure containing:

- `chisqstat` — Value of the test statistic.
- `df` — Degrees of freedom of the test.

## Definitions

### Chi-Square Variance Test

The chi-square variance test is used to test whether the variance of a population is equal to a hypothesized value.

The test statistic is

$$T = (n - 1) \left( \frac{s}{\sigma_0} \right)^2,$$

where  $n$  is the sample size,  $s$  is the sample standard deviation, and  $\sigma_0$  is the hypothesized standard deviation. The denominator is the ratio of the sample standard deviation to the hypothesized standard deviation. The further this ratio deviates from 1, the more likely you are to reject the null hypothesis. The test statistic  $T$  has a chi-square distribution with  $n - 1$  degrees of freedom under the null hypothesis.

### Multidimensional Array

A multidimensional array has more than two dimensions. For example, if `x` is a 1-by-3-by-4 array, then `x` is a three-dimensional array.

### First Nonsingleton Dimension

The first nonsingleton dimension is the first dimension of an array whose size is not equal to 1. For example, if `x` is a 1-by-2-by-3-by-4 array, then the second dimension is the first nonsingleton dimension of `x`.

## Tips

- Use `sampsizepwr` to calculate:

- The sample size that corresponds to specified power and parameter values;
- The power achieved for a particular sample size, given the true parameter value;
- The parameter value detectable with the specified sample size and power.

## See Also

`sampsizepwr` | `vartest2` | `vartestn`

**Introduced before R2006a**

## vartest2

Two-sample  $F$ -test for equal variances

### Syntax

```
h = vartest2(x,y)
h = vartest2(x,y,Name,Value)
[h,p] = vartest2(____)
[h,p,ci,stats] = vartest2(____)
```

### Description

`h = vartest2(x,y)` returns a test decision for the null hypothesis that the data in vectors `x` and `y` comes from normal distributions with the same variance, using the two-sample  $F$ -test on page 32-6908. The alternative hypothesis is that they come from normal distributions with different variances. The result `h` is 1 if the test rejects the null hypothesis at the 5% significance level, and 0 otherwise.

`h = vartest2(x,y,Name,Value)` returns a test decision for the two-sample  $F$ -test with additional options specified by one or more name-value pair arguments. For example, you can change the significance level or conduct a one-sided test.

`[h,p] = vartest2(____)` also returns the  $p$ -value of the test, `p`, using any of the input arguments in the previous syntaxes.

`[h,p,ci,stats] = vartest2(____)` also returns the confidence interval for the true variance ratio, `ci`, and the structure `stats` containing information about the test statistic.

### Examples

### Test for Equal Variances

Load the sample data. Create vectors containing the first and second columns of the data matrix to represent students' grades on two exams.

```
load examgrades;  
x = grades(:,1);  
y = grades(:,2);
```

Test the null hypothesis that the data in `x` and `y` comes from distributions with the same variance.

```
[h,p,ci,stats] = vartest2(x,y)
```

```
h =
```

```
    1
```

```
p =
```

```
    0.0019
```

```
ci =
```

```
    1.2383
```

```
    2.5494
```

```
stats =
```

```
    struct with fields:
```

```
    fstat: 1.7768
```

```
    df1: 119
```

```
    df2: 119
```

The returned result `h = 1` indicates that `vartest2` rejects the null hypothesis at the default 5% significance level. `ci` contains the lower and upper boundaries of the 95%

confidence interval for the true variance ratio. `stats` contains the value of the test statistic for the  $F$ -test and the numerator and denominator degrees of freedom.

## One-Sided Hypothesis Test

Load the sample data. Create vectors containing the first and second columns of the data matrix to represent students' grades on two exams.

```
load examgrades;  
x = grades(:,1);  
y = grades(:,2);
```

Test the null hypothesis that the data in `x` and `y` comes from distributions with the same variance, against the alternative that the population variance of `x` is greater than that of `y`.

```
vartest2(x,y,'Tail','right')
```

```
ans =
```

```
1
```

The returned result `h = 1` indicates that `vartest2` rejects the null hypothesis at the default 5% significance level, in favor of the alternative hypothesis that the population variance of `x` is greater than that of `y`.

## Input Arguments

### **x** — Sample data

vector | matrix | multidimensional array

Sample data, specified as a vector, matrix, or multidimensional array on page 32-6909.

- If `x` and `y` are vectors, they do not need to be the same length.
- If `x` and `y` are matrices, they must have the same number of columns, but do not need to have the same number of rows. `vartest2` performs separate tests along each column and returns a vector of the results.

- If  $x$  and  $y$  are multidimensional arrays, they must have the same number of dimensions, and the same size along all but the first nonsingleton dimension on page 32-6909.

Data Types: `single` | `double`

### **$y$ — Sample data**

vector | matrix | multidimensional array

Sample data, specified as a vector, matrix, or multidimensional array on page 32-6909.

- If  $x$  and  $y$  are vectors, they do not need to be the same length.
- If  $x$  and  $y$  are matrices, they must have the same number of columns, but do not need to have the same number of rows. `vartest2` performs separate tests along each column and returns a vector of the results.
- If  $x$  and  $y$  are multidimensional arrays, they must have the same number of dimensions, and the same size along all but the first nonsingleton dimension on page 32-6909.

Data Types: `single` | `double`

## **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'Tail', 'right', 'Alpha', 0.01` specifies a right-tailed hypothesis test at the 1% significance level.

### **Alpha — Significance level**

0.05 (default) | scalar value in the range (0,1)

Significance level of the hypothesis test, specified as the comma-separated pair consisting of `'Alpha'` and a scalar value in the range (0,1).

Example: `'Alpha', 0.01`

Data Types: `single` | `double`



**Dim — Dimension**

first nonsingleton dimension (default) | positive integer value

Dimension of the input matrix to test along, specified as the comma-separated pair consisting of 'Dim' and a positive integer value. For example, specifying 'Dim', 1 tests the data in each column for variance equality, while 'Dim', 2 tests the data in each row.

Example: 'Dim', 2

Data Types: single | double

**Tail — Type of alternative hypothesis**

'both' (default) | 'right' | 'left'

Type of alternative hypothesis to evaluate using the  $F$ -test, specified as the comma-separated pair consisting of 'Tail' and one of the following.

'both'	Test the alternative hypothesis that the population variances are not equal.
'right'	Test the alternative hypothesis that the population variance of $x$ is greater than that of $y$ .
'left'	Test the alternative hypothesis that the population variance of $x$ is less than that of $y$ .

Example: 'Tail', 'right'

## Output Arguments

**h — Hypothesis test result**

1 | 0

Hypothesis test result, returned as a logical value.

- If  $h = 1$ , this indicates the rejection of the null hypothesis at the Alpha significance level.
- If  $h = 0$ , this indicates a failure to reject the null hypothesis at the Alpha significance level.

**p —  $p$ -value**

scalar value in the range [0,1]

$p$ -value of the test, returned as a scalar value in the range [0,1].  $p$  is the probability of observing a test statistic as extreme as, or more extreme than, the observed value under the null hypothesis. Small values of  $p$  cast doubt on the validity of the null hypothesis.

**ci — Confidence interval**

vector

Confidence interval for the true ratio of the population variances, returned as a two-element vector containing the lower and upper boundaries of the  $100 \times (1 - \text{Alpha})\%$  confidence interval.

**stats — Test statistics**

structure

Test statistics for the hypothesis test, returned as a structure containing:

- `fstat` — Value of the test statistic.
- `df1` — Numerator degrees of freedom of the test.
- `df2` — Denominator degrees of freedom of the test.

## Definitions

### Two-Sample $F$ -Test

The two-sample  $F$ -test is used to test if the variances of two populations are equal.

The test statistic is

$$F = \frac{s_1^2}{s_2^2},$$

where  $s_1$  and  $s_2$  are the sample standard deviations. The test statistic is a ratio of the two sample variances. The further this ratio deviates from 1, the more likely you are to reject the null hypothesis. Under the null hypothesis, the test statistic  $F$  has a  $F$ -distribution with numerator degrees of freedom equal to  $N_1 - 1$  and denominator degrees of freedom equal to  $N_2 - 1$ , where  $N_1$  and  $N_2$  are the sample sizes of the two data sets.

## Multidimensional Array

A multidimensional array has more than two dimensions. For example, if `x` is a 1-by-3-by-4 array, then `x` is a three-dimensional array.

## First Nonsingleton Dimension

The first nonsingleton dimension is the first dimension of an array whose size is not equal to 1. For example, if `x` is a 1-by-2-by-3-by-4 array, then the second dimension is the first nonsingleton dimension of `x`.

## See Also

`vartest` | `vartestn`

Introduced before R2006a

## vartestn

Multiple-sample tests for equal variances

### Syntax

```
vartestn(x)
vartestn(x,Name,Value)

vartestn(x,group)
vartestn(x,group,Name,Value)

p = vartestn(____)
[p,stats] = vartestn(____)
```

### Description

`vartestn(x)` returns a summary table of statistics and a box plot for a Bartlett test of the null hypothesis that the columns of data vector `x` come from normal distributions with the same variance. The alternative hypothesis is that not all columns of data have the same variance.

`vartestn(x,Name,Value)` returns a summary table of statistics and a box plot for a test of unequal variances with additional options specified by one or more name-value pair arguments. For example, you can specify a different type of hypothesis test or change the display settings for the test results.

`vartestn(x,group)` returns a summary table of statistics and a box plot for a Bartlett test of the null hypothesis that the data in each categorical group comes from normal distributions with the same variance. The alternative hypothesis is that not all groups have the same variance.

`vartestn(x,group,Name,Value)` returns a summary table of statistics and a box plot for a test of unequal variances with additional options specified by one or more name-value pair arguments. For example, you can specify a different type of hypothesis test or change the display settings for the test results.

`p = vartestn( ___ )` also returns the  $p$ -value of the test, `p`, using any of the input arguments in the previous syntaxes.

`[p, stats] = vartestn( ___ )` also returns the structure `stats` containing information about the test statistic.

## Examples

### Test Data for Equal Variances

Load the sample data.

```
load examgrades
```

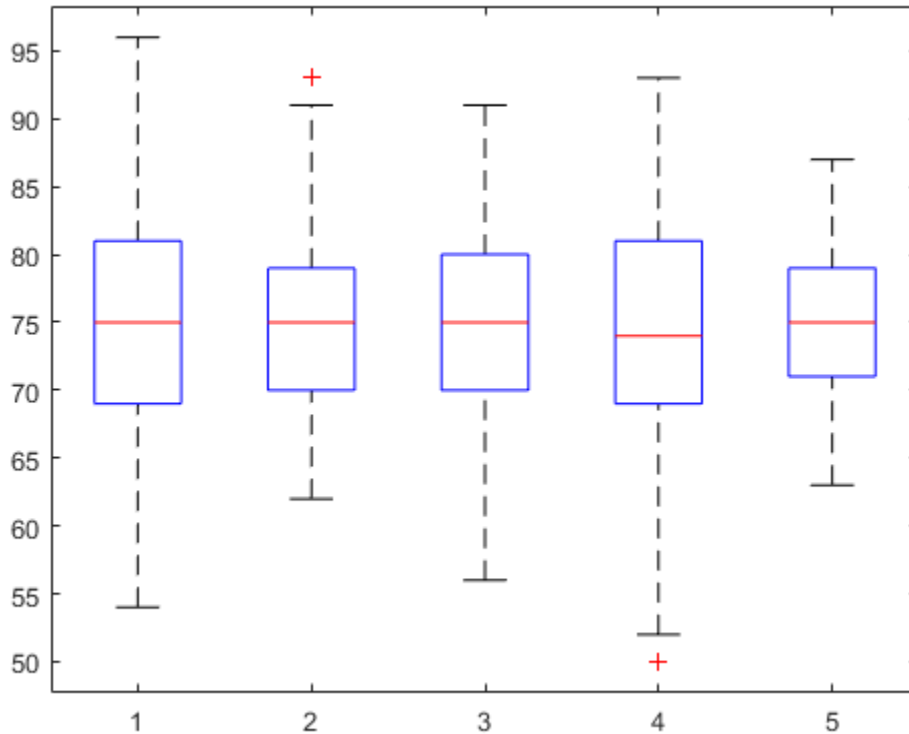
Test the null hypothesis that the variances are equal across the five columns of data in the students' exam grades matrix, `grades`.

```
vartestn(grades)
```

```
ans =
```

```
7.9086e-08
```

Group Summary Table			
Group	Count	Mean	Std Dev
1	120	75.0083	8.7202
2	120	74.9917	6.54204
3	120	74.9917	7.43091
4	120	75.0333	8.60128
5	120	74.9917	5.25884
Pooled	600	75.0033	7.42558
Bartlett's statistic	38.7332		
Degrees of freedom	4		
p-value	0		



The low  $P$ -value,  $p = 0$ , indicates that `vartestn` rejects the null hypothesis that the variances are equal across all five columns, in favor of the alternative hypothesis that at least one column has a different variance.

### Test Grouped Data for Equal Variances

Load the sample data.

```
load carsmall
```

Test the null hypothesis that the variances in miles per gallon (MPG) are equal across different model years.

```
varTestn(MPG, Model_Year)
```

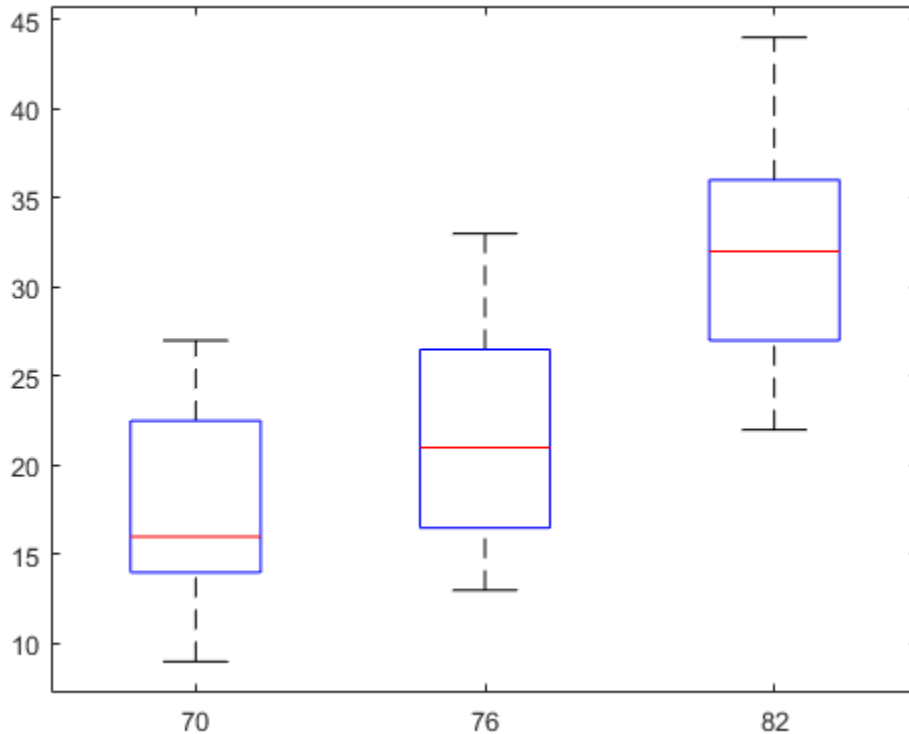
```
ans =
```

```
0.8327
```

### Group Summary Table

Group	Count	Mean	Std Dev
70	29	17.6897	5.33923
76	34	21.5735	5.8893
82	31	31.7097	5.39255
Pooled	94	23.7181	5.562
Bartlett's statistic	0.36619		
Degrees of freedom	2		
p-value	0.83269		





The high  $P$ -value,  $p = 0.83269$ , indicates that `vartestn` does not reject the null hypothesis that the variances in miles per gallon (MPG) are equal across different model years.

### Test for Equal Variances Using Levene's Test

Load the sample data.

```
load carsmall
```

Use Levene's test to test the null hypothesis that the variances in miles per gallon (MPG) are equal across different model years.

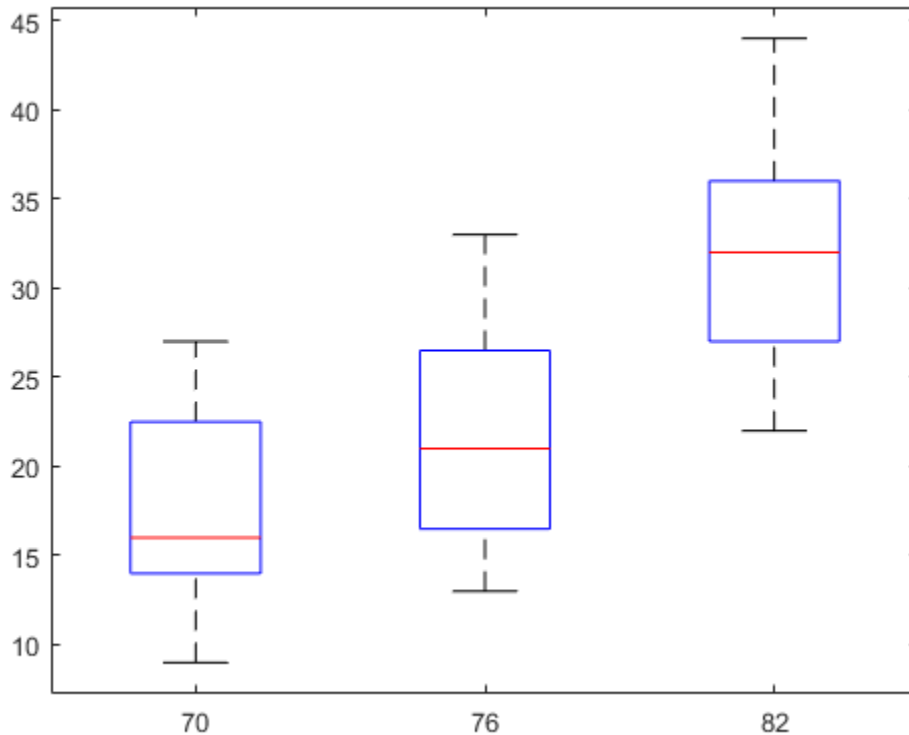
```
p = vartestn(MPG,Model_Year,'TestType','LeveneAbsolute')
```

```
p =
```

```
0.6320
```

### Group Summary Table

Group	Count	Mean	Std Dev
70	29	17.6897	5.33923
76	34	21.5735	5.8893
82	31	31.7097	5.39255
Pooled	94	23.7181	5.562
Levene's statistic (absolute)	0.46126		
Degrees of freedom	2, 91		
p-value	0.63195		



The high  $P$ -value,  $p = 0.63195$ , indicates that `vartestn` does not reject the null hypothesis that the variances in miles per gallon (MPG) are equal across different model years.

### Test for Equal Variances Using the Brown-Forsythe Test

Load the sample data.

```
load examgrades
```

Test the null hypothesis that the variances are equal across the five columns of data in the students' exam grades matrix, `grades`, using the Brown-Forsythe test. Suppress the display of the summary table of statistics and the box plot.

```
[p,stats] = vartestn(grades,'TestType','BrownForsythe','Display','off')
```

```
p =
```

```
1.3121e-06
```

```
stats =
```

```
struct with fields:
```

```
fstat: 8.4160
```

```
df: [4 595]
```

The small *P*-value,  $p = 1.3121e-06$ , indicates that `vartestn` rejects the null hypothesis that the variances are equal across all five columns, in favor of the alternative hypothesis that at least one column has a different variance.

## Input Arguments

### **x** — Sample data

matrix | vector

Sample data, specified as a matrix or vector. If a grouping variable `group` is specified, `x` must be a vector. If a grouping variable is not specified, `x` must be a matrix. In either case, `vartestn` treats NaN values as missing values and ignores them.

Data Types: `single` | `double`

### **group** — Grouping variable

categorical array | logical or numeric vector | cell array of character vectors

Grouping variable, specified as a categorical array, logical or numeric vector, or cell array of character vectors with one row for each element of `x`. Each unique value in a grouping variable defines a group.

For example, if `Gender` is a cell array of character vectors with values `'Male'` and `'Female'`, you can use `Gender` as a grouping variable to test your data by gender.

Example: `Gender`

Data Types: `single` | `double` | `logical` | `cell` | `char`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'TestType', 'BrownForsythe', 'Display', 'off'` specifies a Brown-Forsythe test and omits the plot of the results.

### **Display** — Display settings for test results

`'on'` (default) | `'off'`

Display settings for test results, specified as the comma-separated pair consisting of `'Display'` and one of the following.

`'on'`                    Display a box plot and table of summary statistics.

`'off'`                    Do not display a box plot and table of summary statistics.

Example: `'display', 'off'`

### **TestType** — Type of hypothesis test

`'Bartlett'` (default) | `'LeveneQuadratic'` | `'LeveneAbsolute'` |  
`'BrownForsythe'` | `'OBrien'`

Type of hypothesis test to perform, specified as the comma-separated pair consisting of `'TestType'` and one of the following.

`'Bartlett'`                    Bartlett's test.

`'LeveneQuadratic'`        Levene's test computed by performing ANOVA on the squared deviations of the data values from their group means.

`'LeveneAbsolute'`        Levene's test computed by performing ANOVA on the absolute deviations of the data values from their group means.

'BrownForsythe'      Brown-Forsythe test computed by performing ANOVA on the absolute deviations of the data values from the group medians.

'OBrien'              O'Brien's modification of Levene's test with  $\bar{w} = 0.5$ .

Example: 'TestType', 'OBrien'

## Output Arguments

### **p** — *p*-value

scalar value in the range [0,1]

*p*-value of the test, returned as a scalar value in the range [0,1]. *p* is the probability of observing a test statistic as extreme as, or more extreme than, the observed value under the null hypothesis. Small values of *p* cast doubt on the validity of the null hypothesis.

### **stats** — Test statistics

structure

Test statistics for the hypothesis test, returned as a structure containing:

- **chistat**: Value of the test statistic.
- **df**: Degrees of freedom of the test.

## Definitions

### Bartlett's Test

Bartlett's test is used to test whether multiple data samples have equal variances, against the alternative that at least two of the data samples do not have equal variances.

The test statistic is

$$T = \frac{(N - k) \ln s_p^2 - \sum_{i=1}^k (N_i - 1) \ln s_i^2}{1 + \left(1 / (3(k - 1))\right) \left( \left( \sum_{i=1}^k 1 / (N_i - 1) \right) - 1 / (N - k) \right)},$$

where  $s_i^2$  is the variance of the  $i$ th group,  $N$  is the total sample size,  $N_i$  is the sample size of the  $i$ th group,  $k$  is the number of groups, and  $s_p^2$  is the pooled variance. The pooled variance is defined as

$$s_p^2 = \sum_{i=1}^k (N_i - 1) s_i^2 / (N - k).$$

The test statistic has a chi-square distribution with  $k - 1$  degrees of freedom under the null hypothesis.

Bartlett's test is sensitive to departures from normality. If your data comes from a nonnormal distribution, Levene's test could provide a more accurate result.

## Levene, Brown-Forsythe, and O'Brien Tests

The Levene, Brown-Forsythe, and O'Brien tests are used to test if multiple data samples have equal variances, against the alternative that at least two of the data samples do not have equal variances.

The test statistic is

$$W = \frac{(N - k) \sum_{i=1}^k N_i (\bar{Z}_{i.} - \bar{Z}_{..})^2}{(k - 1) \sum_{i=1}^k \sum_{j=1}^{N_i} (Z_{ij} - \bar{Z}_{i.})^2},$$

where  $N_i$  is the sample size of the  $i$ th group, and  $k$  is the number of groups. Depending on the type of test specified with the `TestType` name-value pair arguments,  $Z_{ij}$  can have one of four definitions:

- If you specify `LeveneAbsolute`, `vartestn` uses  $Z_{ij} = |Y_{ij} - \bar{Y}_{i.}|$ , where  $\bar{Y}_{i.}$  is the mean of the  $i$ th subgroup.
- If you specify `LeveneQuadratic`, `vartestn` uses  $Z_{ij}^2 = (Y_{ij} - \bar{Y}_{i.})^2$ , where  $\bar{Y}_{i.}$  is the mean of the  $i$ th subgroup.

- If you specify `BrownForsythe`, `vartestn` uses  $Z_{ij} = |Y_{ij} - \tilde{Y}_{i.}|$ , where  $\tilde{Y}_{i.}$  is the median of the  $i$ th subgroup.
- If you specify `OBrien`, `vartestn` uses

$$Z_{ij} = \frac{(0.5 + n_i - 2)n_i (y_{ij} - \bar{y}_i)^2 - 0.5(n_i - 1)\sigma_i^2}{(n_i - 1)(n_i - 2)},$$

where  $n_i$  is the size of the  $i$ th group,  $\sigma_i^2$  is its sample variance.

In all cases, the test statistic has an  $F$ -distribution with  $k - 1$  numerator degrees of freedom, and  $N - k$  denominator degrees of freedom.

The Levene, Brown-Forsythe, and O'Brien tests are less sensitive to departures from normality than Bartlett's test, so they are useful alternatives if you suspect the samples come from nonnormal distributions.

## See Also

`anova1` | `vartest` | `vartest2`

Introduced before R2006a



# vertcat

**Class:** dataset

Vertical concatenation for dataset arrays

---

**Note** The `dataset` data type might be removed in a future release. To work with heterogeneous data, use the MATLAB® `table` data type instead. See MATLAB `table` documentation for more information.

---

## Syntax

```
ds = vertcat(ds1, ds2, ...)
```

## Description

`ds = vertcat(ds1, ds2, ...)` vertically concatenates the dataset arrays `ds1`, `ds2`, ... . Observation names, when present, must be unique across datasets. `vertcat` fills in default observation names for the output when some of the inputs have names and some do not.

Variable names for all dataset arrays must be identical except for order. `vertcat` concatenates by matching variable names. `vertcat` assigns values for the "per-variable" properties (e.g., `Units` and `VarDescription`) in `ds` from the corresponding property values in `ds1`.

## See Also

`cat` | `horzcat`

## view

**Class:** `classregtree`

Plot tree

---

**Note** `classregtree` will be removed in a future release. See `fitctree`, `fitrtree`, `ClassificationTree`, or `RegressionTree` instead.

---

## Syntax

```
view(t)
view(t,param1, val1,param2, val2, ...)
```

## Description

`view(t)` displays the decision tree `t` as computed by `classregtree` in a figure window. Each branch in the tree is labeled with its decision rule, and each terminal node is labeled with the predicted value for that node. Click any node to get more information about it. The information displayed is specified by the **Click to display** pop-up menu at the top of the figure.

`view(t,param1, val1,param2, val2, ...)` specifies optional parameter name/value pairs:

- `'names'` — A cell array of names for the predictor variables, in the order in which they appear in the matrix `X` from which the tree was created. (See `classregtree`.)
- `'prunelevel'` — Initial pruning level to display.

For each branch node, the left child node corresponds to the points that satisfy the condition, and the right child node corresponds to the points that do not satisfy the condition.

## Examples

Create a classification tree for Fisher's iris data:

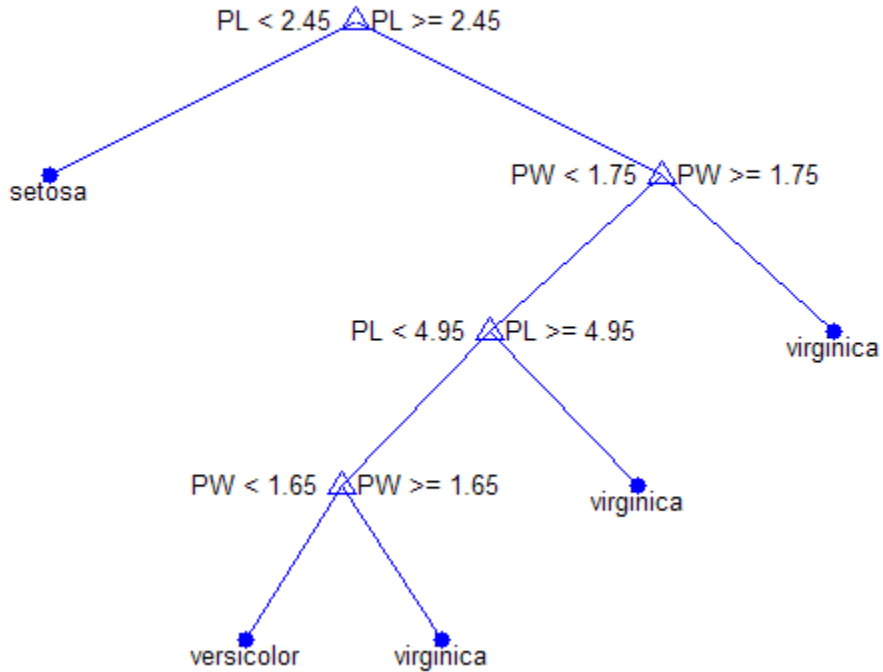
```
load fisheriris;

t = classregtree(meas,species,...
                'names',{'SL' 'SW' 'PL' 'PW'})

t =
Decision tree for classification
1  if PL<2.45 then node 2 elseif PL>=2.45 then node 3 else setosa
2  class = setosa
3  if PW<1.75 then node 4 elseif PW>=1.75 then node 5 else versicolor
4  if PL<4.95 then node 6 elseif PL>=4.95 then node 7 else versicolor
5  class = virginica
6  if PW<1.65 then node 8 elseif PW>=1.65 then node 9 else versicolor
7  class = virginica
8  class = versicolor
9  class = virginica

view(t)
```

Click to display:  Magnification:  Pruning level:



## References

- [1] Breiman, L., J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Boca Raton, FL: CRC Press, 1984.

## See Also

`classregtree` | `eval` | `prune` | `test`

# compact

**Class:** clustering.evaluation.ClusterCriterion

**Package:** clustering.evaluation

Compact clustering evaluation object

## Syntax

```
c = compact(eva)
```

## Description

`c = compact(eva)` returns a compact clustering evaluation object, which contains a subset of information about the clustering solution in `eva`. Compacting a clustering evaluation object reduces the memory requirements of the object, which is useful when clustering a large data set.

## Input Arguments

**eva** — Clustering evaluation data

clustering evaluation object

Clustering evaluation data, specified as a clustering evaluation object. Create a clustering evaluation object using `evalclusters`.

## Output Arguments

**c** — Compact clustering evaluation object

clustering evaluation object

Compact clustering evaluation object, returned as a clustering evaluation object. The compact object includes the clustering evaluation results. In the compact object, the

properties for the input data `X`, optimal clustering solution `OptimalY`, and the list of excluded data `Missing` are empty.

## Examples

### Create a Compact Clustering Evaluation Object

Create a compact clustering evaluation object from a full clustering evaluation object.

Load the sample data.

```
load fisheriris;
```

The data contains length and width measurements from the sepals and petals of three species of iris flowers.

Create a clustering evaluation object. Cluster the data using `kmeans`, and evaluate the optimal number of clusters using the gap criterion.

```
rng('default'); % For reproducibility
eva = evalclusters(meas, 'kmeans', 'Gap', 'KList', [1:6])
```

```
eva =
```

```
GapEvaluation with properties:
```

```
NumObservations: 150
InspectedK: [1 2 3 4 5 6]
CriterionValues: [0.0720 0.5928 0.8762 1.0114 1.0534 1.0720]
OptimalK: 5
```

Create a compact clustering evaluation object from `eva`.

```
c = compact(eva)
```

```
c =
```

```
GapEvaluation with properties:
```

```
NumObservations: 150
  InspectedK: [1 2 3 4 5 6]
CriterionValues: [0.0720 0.5928 0.8762 1.0114 1.0534 1.0720]
  OptimalK: 5
```

The displayed output of the compact object `c` is the same as the original object `eva`, but some properties not shown in the display are different. For example, in the compact object, the properties `x`, `OptimalY`, and `Missing` are empty.

Display the optimal clustering solution `OptimalY` for `c`.

```
c.OptimalY
```

```
ans =
```

```
 []
```

## See Also

`evalclusters`

## view

**Class:** CompactClassificationTree

View tree

## Syntax

```
view(tree)
view(tree,Name,Value)
```

## Description

`view(tree)` returns a text description of `tree`, a decision tree.

`view(tree,Name,Value)` describes `tree` with additional options specified by one or more `Name,Value` pair arguments.

## Input Arguments

**tree**

A classification tree or compact classification tree created by `fitctree` or `compact`.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.



**Mode**

Value describing the display of tree, either 'graph' or 'text'. 'graph' opens a user interface displaying tree, and containing controls for querying the tree. 'text' sends output to the Command Window describing tree.

**Default:** 'text'

## Examples

### View Trained Classification Tree

View textual and graphical displays of a trained classification tree.

Load Fisher's iris data set.

```
load fisheriris
```

Train a classification tree using all measurements.

```
Mdl = fitctree(meas,species);
```

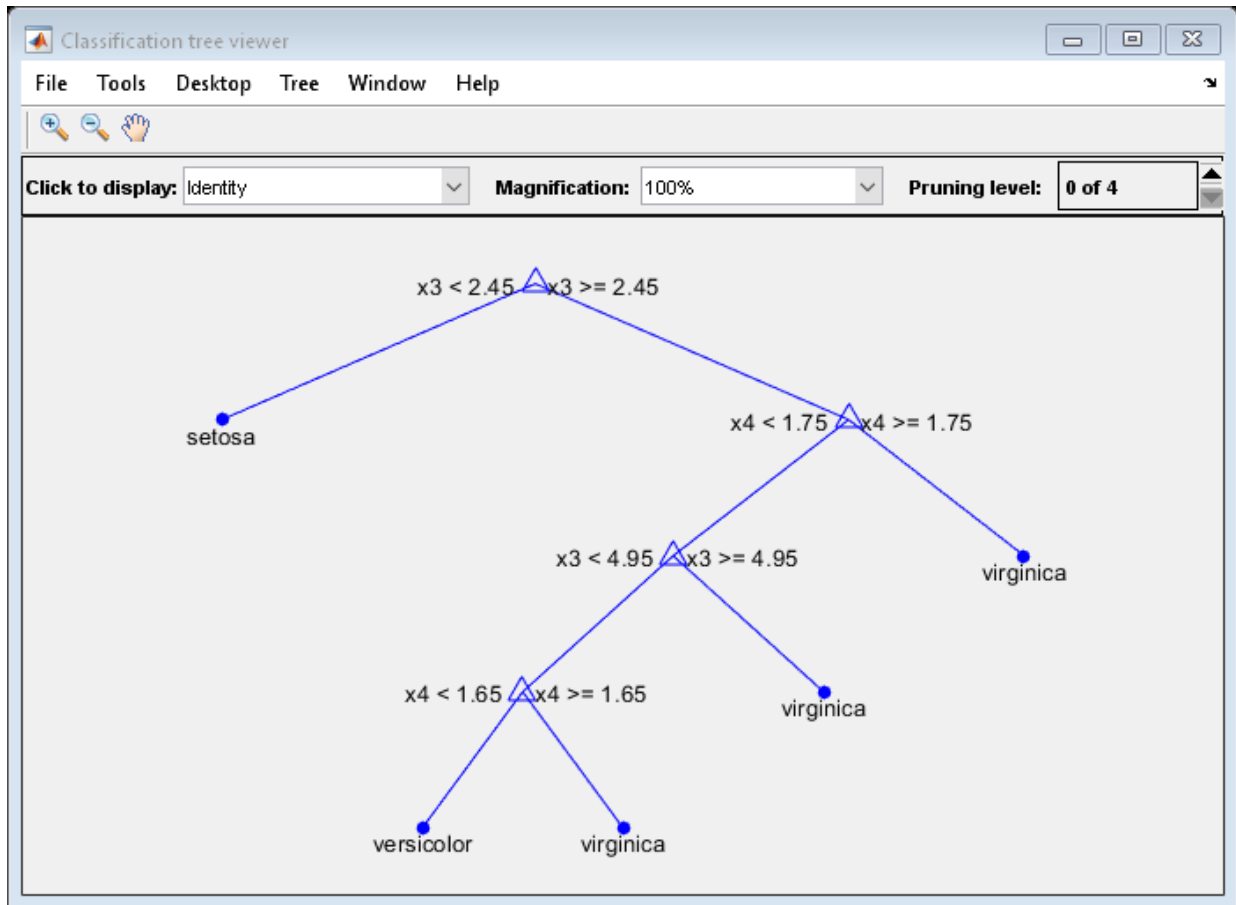
View textual display of the trained classification tree.

```
view(Mdl)
```

```
Decision tree for classification
1  if x3<2.45 then node 2 elseif x3>=2.45 then node 3 else setosa
2  class = setosa
3  if x4<1.75 then node 4 elseif x4>=1.75 then node 5 else versicolor
4  if x3<4.95 then node 6 elseif x3>=4.95 then node 7 else versicolor
5  class = virginica
6  if x4<1.65 then node 8 elseif x4>=1.65 then node 9 else versicolor
7  class = virginica
8  class = versicolor
9  class = virginica
```

View graphical display of the trained classification tree.

```
view(Mdl, 'Mode', 'graph');
```



### View Tree from Bag of Trees

Load Fisher's iris data set.

```
load fisheriris
```

Grow a bag of 100 classification trees using all measurements.

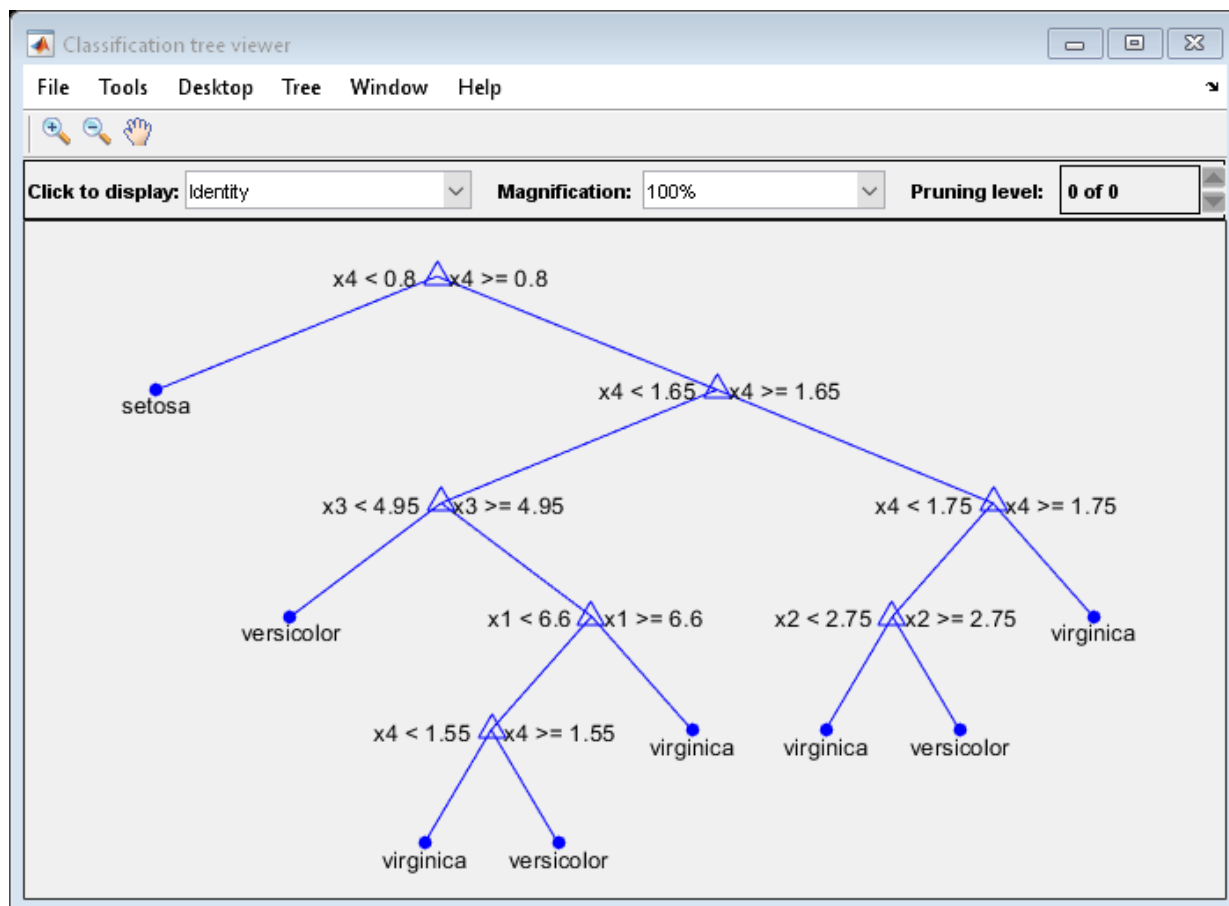
```
rng(1) % For reproducibility
Mdl = TreeBagger(100, meas, species);
```

Alternatively, you can use `fitensemble` to grow a bag of classification trees.

`Mdl` is a `TreeBagger` model object. `Mdl.Trees` stores the bag of 100 trained classification trees in a 100-by-1 cell array. That is, each cell in `Mdl.Trees` contains a `CompactClassificationTree` model object.

View a graph of the 10th classification tree in the bag.

```
Tree10 = Mdl.Trees{10};
view(Tree10, 'Mode', 'graph');
```



By default, the software grows deep trees for bags of trees.

### View Tree from Boosted Ensemble

Load Fisher's iris data set.

```
load fisheriris
```

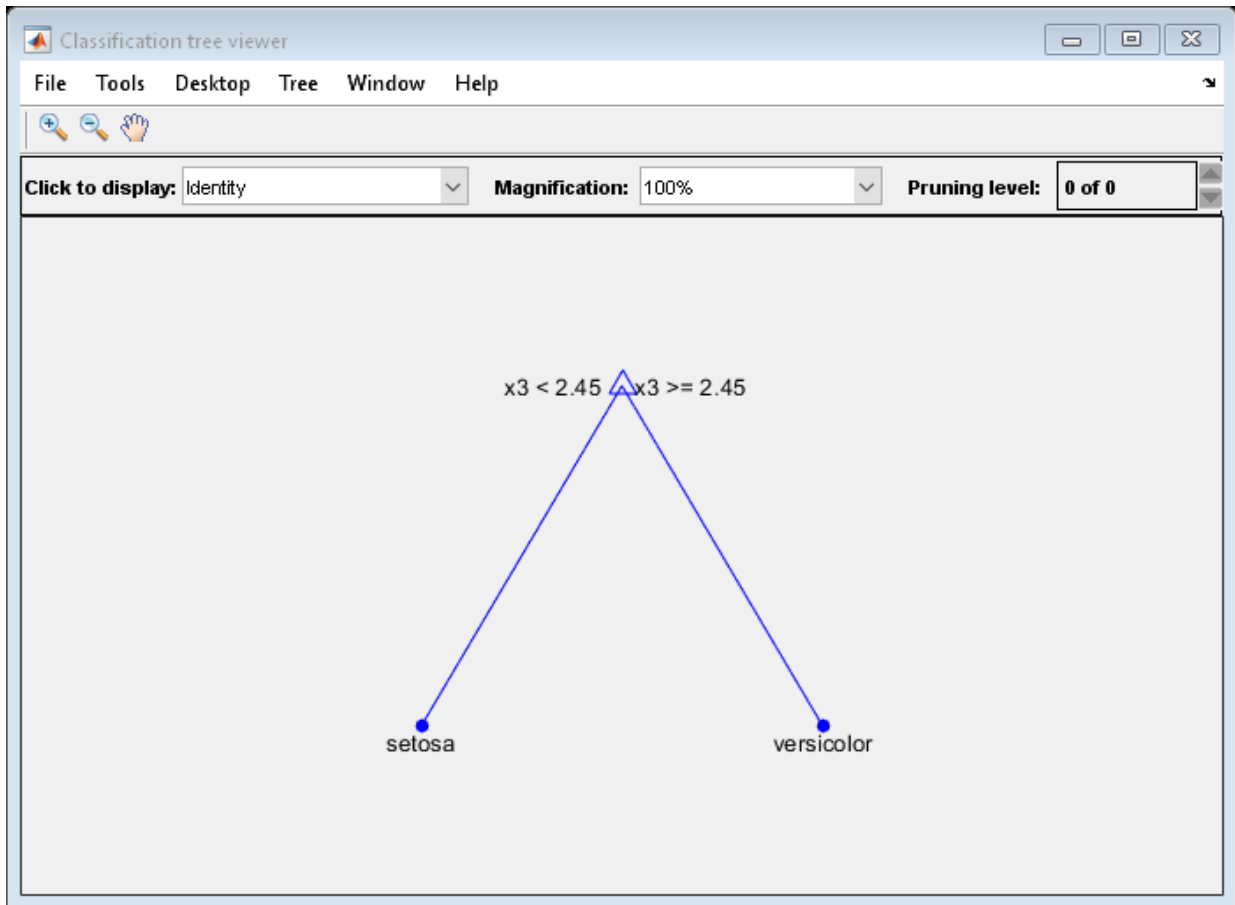
Boost an ensemble of 100 classification trees using all measurements.

```
Mdl = fitensemble(meas,species,'AdaBoostM2',100,'Tree');
```

Mdl is a `ClassificationEnsemble` model object. `Mdl.Trained` stores the ensemble of 100 trained classification trees in a 100-by-1 cell array. That is, each cell in `Mdl.Trained` contains a `CompactClassificationTree` model object.

View a graph of the 10th classification tree in the ensemble.

```
Tree10 = Mdl.Trained{10};  
view(Tree10,'Mode','graph');
```



By default, `fitensemble` grows shallow trees for boosted ensembles of trees.

## Tip

To view tree  $t$  from an ensemble of trees, enter one of these lines of code

```
view(Ens.Trained{t})  
view(Bag.Trees{t})
```

- `Ens` is a full ensemble returned by `fitcensemble` or a compact ensemble returned by `compact`.
- `Bag` is a full bag of trees returned by `TreeBagger` or a compact bag of trees returned by `compact`.

To save `tree` in the Command Window, get a figure handle by using the `findall` and `setdiff` functions, and then save `tree` using the function `saveas`.

```
before = findall(groot,'Type','figure'); % Find all figures
view(Mdl,'Mode','graph')
after = findall(groot,'Type','figure');
h = setdiff(after,before); % Get the figure handle of the tree viewer
saveas(h,'a.png')
```

## See Also

`ClassificationTree` | `fitctree`

## view

**Class:** CompactRegressionTree

View tree

## Syntax

```
view(tree)
view(tree, Name, Value)
```

## Description

`view(tree)` returns a text description of `tree`, a decision tree.

`view(tree, Name, Value)` describes `tree` with additional options specified by one or more `Name, Value` pair arguments.

## Input Arguments

**tree**

A regression tree or compact regression tree created by `fitrtree` or `compact`.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

**Mode**

Display of tree, either 'graph' or 'text'. 'graph' opens a GUI displaying tree, and containing controls for querying the tree. 'text' sends output to the Command Window describing tree.

**Default:** 'text'

## Examples

### View Trained Regression Tree

View textual and graphical displays of a trained regression tree.

Load the `carsmall` data set. Consider a model that explains a car's fuel economy (MPG) using its weight (`Weight`) and number of cylinders (`Cylinders`).

```
load carsmall
X = [Weight Cylinders];
Y = MPG;
```

Train a regression tree using all measurements.

```
Mdl = fitrtree(X,Y);
```

View textual display of the trained regression tree.

```
view(Mdl)
```

```
Decision tree for regression
```

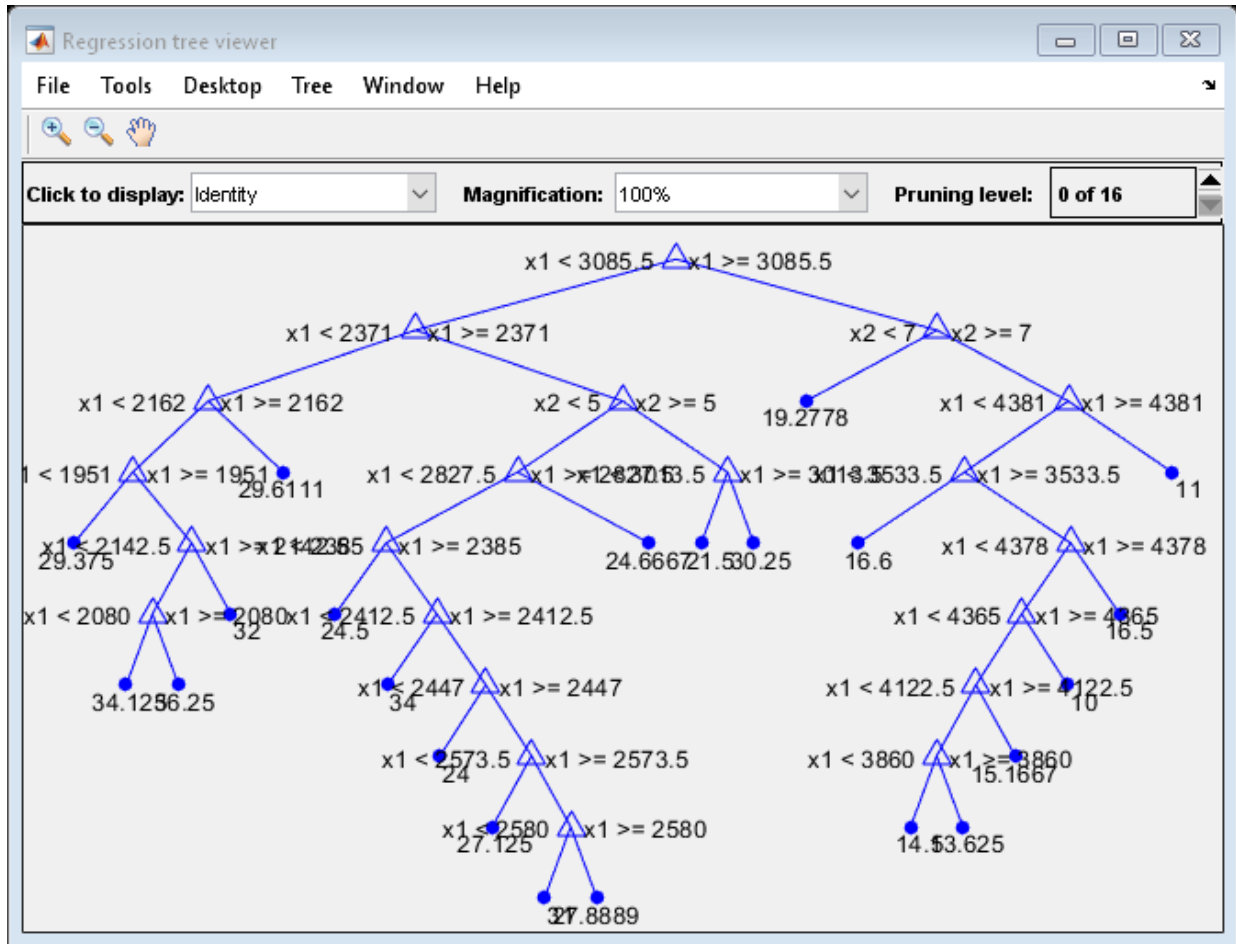
```
1  if x1<3085.5 then node 2 elseif x1>=3085.5 then node 3 else 23.7181
2  if x1<2371 then node 4 elseif x1>=2371 then node 5 else 28.7931
3  if x2<7 then node 6 elseif x2>=7 then node 7 else 15.5417
4  if x1<2162 then node 8 elseif x1>=2162 then node 9 else 32.0741
5  if x2<5 then node 10 elseif x2>=5 then node 11 else 25.9355
6  fit = 19.2778
7  if x1<4381 then node 12 elseif x1>=4381 then node 13 else 14.2963
8  if x1<1951 then node 14 elseif x1>=1951 then node 15 else 33.3056
9  fit = 29.6111
10 if x1<2827.5 then node 16 elseif x1>=2827.5 then node 17 else 27.2143
```



```
11 if x1<3013.5 then node 18 elseif x1>=3013.5 then node 19 else 23.25
12 if x1<3533.5 then node 20 elseif x1>=3533.5 then node 21 else 14.8696
13 fit = 11
14 fit = 29.375
15 if x1<2142.5 then node 22 elseif x1>=2142.5 then node 23 else 34.4286
16 if x1<2385 then node 24 elseif x1>=2385 then node 25 else 27.6389
17 fit = 24.6667
18 fit = 21.5
19 fit = 30.25
20 fit = 16.6
21 if x1<4378 then node 26 elseif x1>=4378 then node 27 else 14.3889
22 if x1<2080 then node 28 elseif x1>=2080 then node 29 else 34.8333
23 fit = 32
24 fit = 24.5
25 if x1<2412.5 then node 30 elseif x1>=2412.5 then node 31 else 28.0313
26 if x1<4365 then node 32 elseif x1>=4365 then node 33 else 14.2647
27 fit = 16.5
28 fit = 34.125
29 fit = 36.25
30 fit = 34
31 if x1<2447 then node 34 elseif x1>=2447 then node 35 else 27.6333
32 if x1<4122.5 then node 36 elseif x1>=4122.5 then node 37 else 14.5313
33 fit = 10
34 fit = 24
35 if x1<2573.5 then node 38 elseif x1>=2573.5 then node 39 else 27.8929
36 if x1<3860 then node 40 elseif x1>=3860 then node 41 else 14.15
37 fit = 15.1667
38 fit = 27.125
39 if x1<2580 then node 42 elseif x1>=2580 then node 43 else 28.2
40 fit = 14.5
41 fit = 13.625
42 fit = 31
43 fit = 27.8889
```

View graphical display of the trained regression tree.

```
view(Mdl, 'Mode', 'graph');
```



### View Tree from Bag of Trees

Load the `carsmall` data set. Consider a model that explains a car's fuel economy (MPG) using its weight (`Weight`) and number of cylinders (`Cylinders`).

```
load carsmall
X = [Weight Cylinders];
Y = MPG;
```

Grow a bag of 100 regression trees using all measurements.

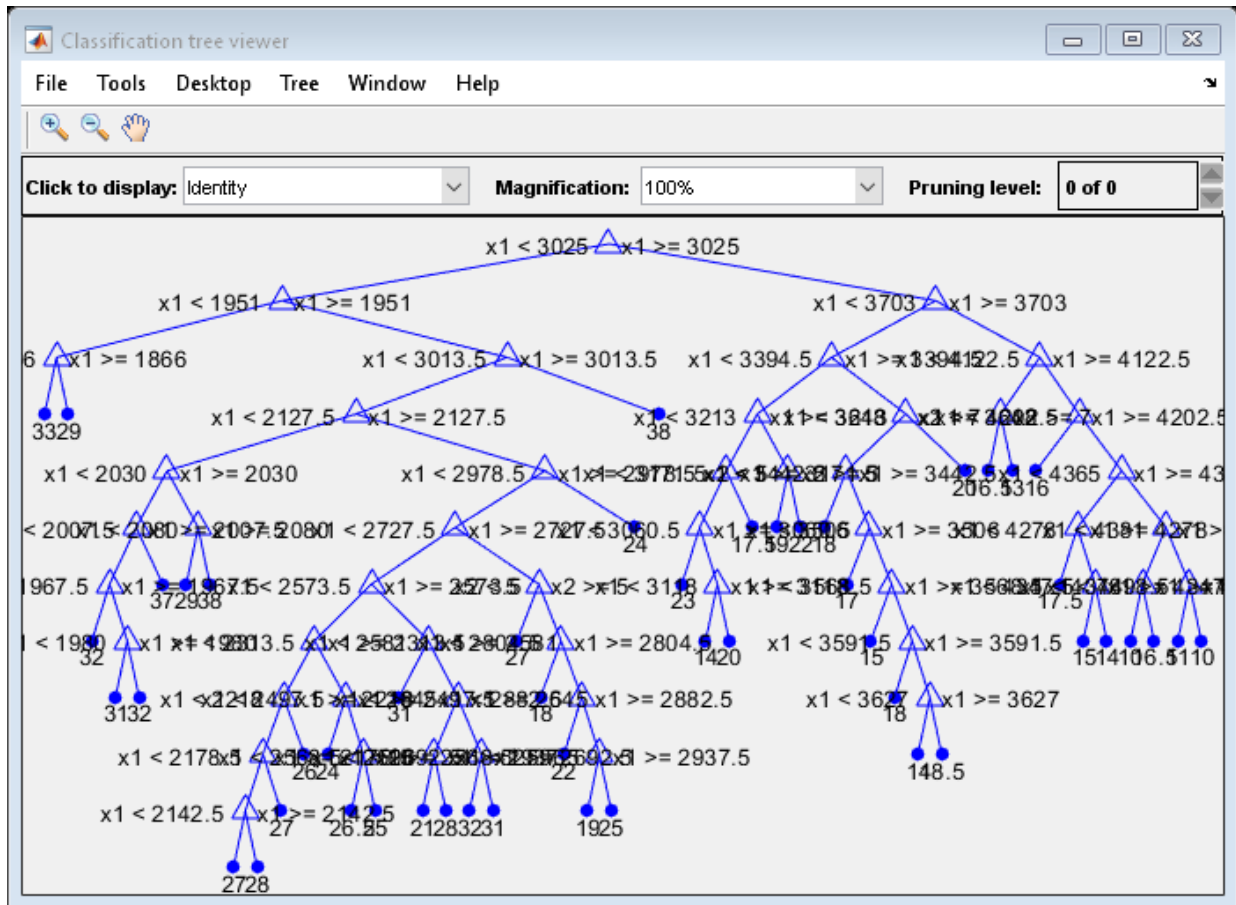
```
rng(1) % For reproducibility
Mdl = TreeBagger(100,X,Y);
```

Alternatively, you can use `fitensemble` to grow a bag of regression trees.

`Mdl` is a `TreeBagger` model object. `Mdl.Trees` stores the bag of 100 trained regression trees in a 100-by-1 cell array. That is, each cell in `Mdl.Trees` contains a `CompactRegressionTree` model object.

View a graph of the 10th regression tree in the bag.

```
Tree10 = Mdl.Trees{10};
view(Tree10, 'Mode', 'graph');
```



By default, the software grows deep trees for bags of trees.

### View Tree from Boosted Ensemble

Load the `carsmall` data set. Consider a model that explains a car's fuel economy (MPG) using its weight (`Weight`) and number of cylinders (`Cylinders`).

```
load carsmall
X = [Weight Cylinders];
Y = MPG;
```

---

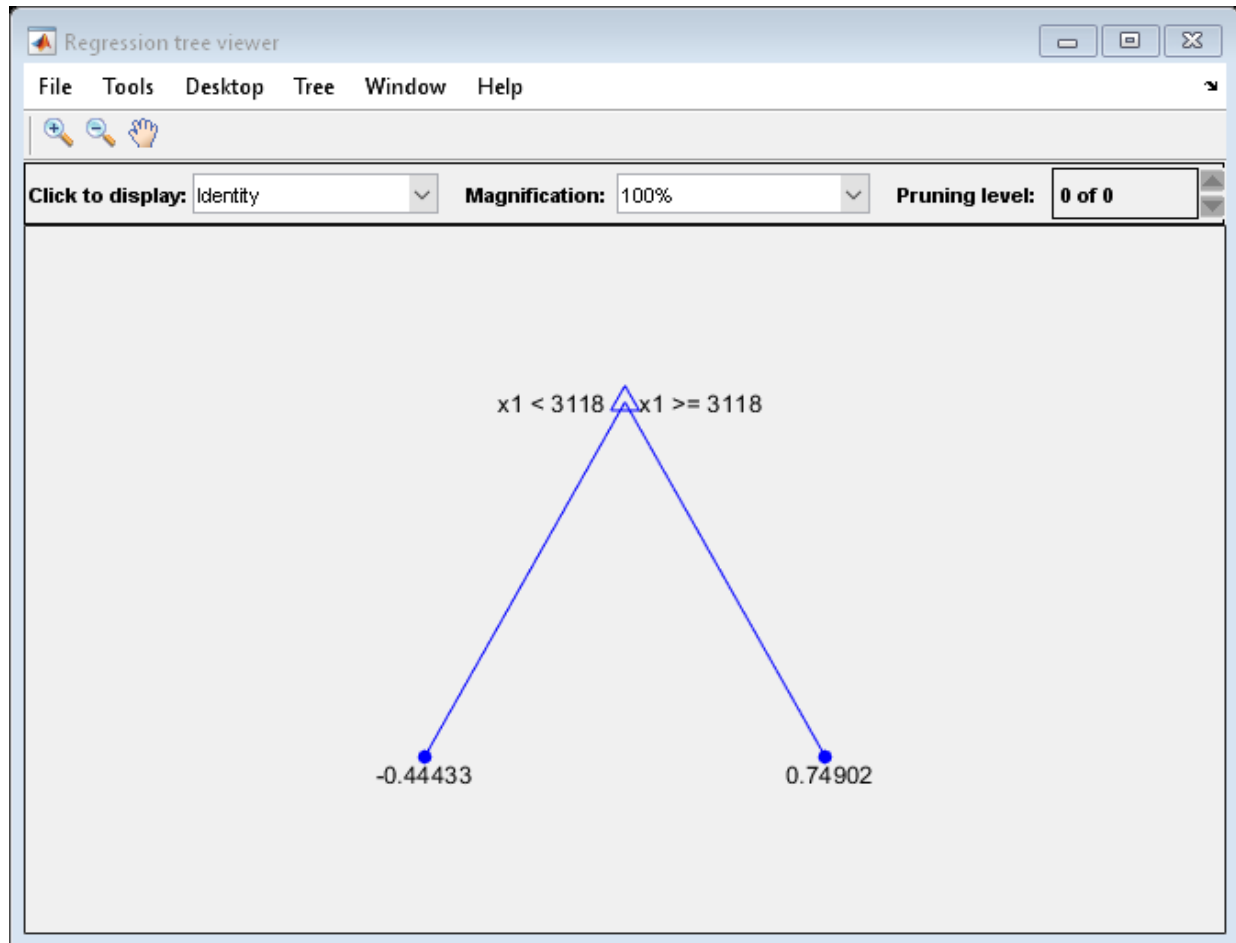
Boost an ensemble of 100 regression trees using all measurements.

```
Mdl = fitensemble(X,Y, 'LSBoost',100, 'Tree');
```

Mdl is a RegressionEnsemble model object. Mdl.Trained stores the ensemble of 100 trained regression trees in a 100-by-1 cell array. That is, each cell in Mdl.Trained contains a CompactRegressionTree model object.

View a graph of the 10th regression tree in the ensemble.

```
Tree10 = Mdl.Trained{10};  
view(Tree10, 'Mode', 'graph');
```



By default, `fitensemble` grows shallow trees for boosted ensembles of trees.

## Tip

To view tree `t` from an ensemble of trees, enter one of these lines of code

```
view(Ens.Trained{t})  
view(Bag.Trees{t})
```

- `Ens` is a full ensemble returned by `fitrensemble` or a compact ensemble returned by `compact`.
- `Bag` is a full bag of trees returned by `TreeBagger` or a compact bag of trees returned by `compact`.

To save `tree` in the Command Window, get a figure handle by using the `findall` and `setdiff` functions, and then save `tree` using the function `saveas`.

```
before = findall(groot, 'Type', 'figure'); % Find all figures
view(Mdl, 'Mode', 'graph')
after = findall(groot, 'Type', 'figure');
h = setdiff(after, before); % Get the figure handle of the tree viewer
saveas(h, 'a.png')
```

## See Also

RegressionTree | `fitrtree`

## wblcdf

Weibull cumulative distribution function

### Syntax

```
p = wblcdf(x, a, b)
[p, plo, pup] = wblcdf(x, a, b, pcov, alpha)
[p, plo, pup] = wblcdf( ____, 'upper')
```

### Description

`p = wblcdf(x, a, b)` returns the cdf of the Weibull distribution with scale parameter `a` and shape parameter `b`, at each value in `x`. `x`, `a`, and `b` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array of the same size as the other inputs. The default values for `a` and `b` are both 1. The parameters `a` and `b` must be positive.

`[p, plo, pup] = wblcdf(x, a, b, pcov, alpha)` returns confidence bounds for `p` when the input parameters `a` and `b` are estimates. `pcov` is the 2-by-2 covariance matrix of the estimated parameters. `alpha` has a default value of 0.05, and specifies  $100(1 - \text{alpha})\%$  confidence bounds. `plo` and `pup` are arrays of the same size as `p` containing the lower and upper confidence bounds.

`[p, plo, pup] = wblcdf( ____, 'upper')` returns the complement of the Weibull cdf for each value in `x`, using an algorithm that more accurately computes the extreme upper tail probabilities. You can use 'upper' with any of the previous syntaxes.

The function `wblcdf` computes confidence bounds for `p` using a normal approximation to the distribution of the estimate

$$\hat{b}(\log x - \log \hat{a})$$

and then transforms those bounds to the scale of the output `p`. The computed bounds give approximately the desired confidence level when you estimate `mu`, `sigma`, and `pcov` from



large samples, but in smaller samples other methods of computing the confidence bounds might be more accurate.

The Weibull cdf is

$$p = F(x | a, b) = \int_0^x b a^{-b} t^{b-1} e^{-\left(\frac{t}{a}\right)^b} dt = 1 - e^{-\left(\frac{x}{a}\right)^b} I_{(0, \infty)}(x)$$

## Examples

### Weibull Distribution cdf

What is the probability that a value from a Weibull distribution with parameters  $a = 0.15$  and  $b = 0.8$  is less than 0.5?

```
probability = wblcdf(0.5, 0.15, 0.8)
```

```
probability =
```

```
    0.9272
```

How sensitive is this result to small changes in the parameters?

```
[A, B] = meshgrid(0.1:0.05:0.2, 0.2:0.05:0.3);
probability = wblcdf(0.5, A, B)
```

```
probability =
```

```
    0.7484    0.7198    0.6991
    0.7758    0.7411    0.7156
```

0.8022    0.7619    0.7319

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

[cdf](#) | [wblfit](#) | [wblinv](#) | [wbllike](#) | [wblpdf](#) | [wblplot](#) | [wblrnd](#) | [wblstat](#)

### Topics

“Weibull Distribution” on page B-193

**Introduced before R2006a**

## wblfit

Weibull parameter estimates

### Syntax

```
parmhat = wblfit(data)
[parmhat,parmci] = wblfit(data)
[parmhat,parmci] = wblfit(data,alpha)
[...] = wblfit(data,alpha,censoring)
[...] = wblfit(data,alpha,censoring,freq)
[...] = wblfit(...,options)
```

### Description

`parmhat = wblfit(data)` returns the maximum likelihood estimates, `parmhat`, of the parameters of the Weibull distribution given the values in the vector `data`, which must be positive. `parmhat` is a two-element row vector: `parmhat(1)` estimates the Weibull parameter  $a$ , and `parmhat(2)` estimates the Weibull parameter  $b$ , in the pdf

$$y = f(x|a,b) = ba^{-b}x^{b-1}e^{-\left(\frac{x}{a}\right)^b}I_{(0,\infty)}(x)$$

`[parmhat,parmci] = wblfit(data)` returns 95% confidence intervals for the estimates of  $a$  and  $b$  in the 2-by-2 matrix `parmci`. The first row contains the lower bounds of the confidence intervals for the parameters, and the second row contains the upper bounds of the confidence intervals.

`[[parmhat,parmci] = wblfit(data,alpha)` returns  $100(1 - \text{alpha})\%$  confidence intervals for the parameter estimates.

`[...] = wblfit(data,alpha,censoring)` accepts a Boolean vector, `censoring`, of the same size as `data`, which is 1 for observations that are right-censored and 0 for observations that are observed exactly.

`[...] = wblfit(data,alpha,censoring,freq)` accepts a frequency vector, `freq`, of the same size as `data`. The vector `freq` typically contains integer frequencies for the

corresponding elements in `data`, but can contain any non-negative values. Pass in `[]` for `alpha`, `censoring`, or `freq` to use their default values.

`[...] = wblfit(..., options)` accepts a structure, `options`, that specifies control parameters for the iterative algorithm the function uses to compute maximum likelihood estimates. The Weibull fit function accepts an `options` structure that can be created using the function `statset`. Enter `statset('wblfit')` to see the names and default values of the parameters that `lognfit` accepts in the `options` structure. See the reference page for `statset` for more information about these options.

## Examples

```
data = wblrnd(0.5,0.8,100,1);
[parmhat, parmci] = wblfit(data)
parmhat =
    0.5861    0.8567
parmci =
    0.4606    0.7360
    0.7459    0.9973
```

## See Also

`mle` | `wblcdf` | `wblinv` | `wbllike` | `wblpdf` | `wblplot` | `wblrnd` | `wblstat`

## Topics

“Weibull Distribution” on page B-193

**Introduced before R2006a**

# wblinv

Weibull inverse cumulative distribution function

## Syntax

```
X = wblinv(P,A,B)
[X,XLO,XUP] = wblinv(P,A,B,PCOV,alpha)
```

## Description

`X = wblinv(P,A,B)` returns the inverse cumulative distribution function (cdf) for a Weibull distribution with scale parameter `A` and shape parameter `B`, evaluated at the values in `P`. `P`, `A`, and `B` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array of the same size as the other inputs. The default values for `A` and `B` are both 1.

`[X,XLO,XUP] = wblinv(P,A,B,PCOV,alpha)` returns confidence bounds for `X` when the input parameters `A` and `B` are estimates. `PCOV` is a 2-by-2 matrix containing the covariance matrix of the estimated parameters. `alpha` has a default value of 0.05, and specifies 100(1 - `alpha`)% confidence bounds. `XLO` and `XUP` are arrays of the same size as `X` containing the lower and upper confidence bounds.

The function `wblinv` computes confidence bounds for `X` using a normal approximation to the distribution of the estimate

$$\log a + \frac{\log q}{b}$$

where  $q$  is the  $P$ th quantile from a Weibull distribution with scale and shape parameters both equal to 1. The computed bounds give approximately the desired confidence level when you estimate `mu`, `sigma`, and `PCOV` from large samples, but in smaller samples other methods of computing the confidence bounds might be more accurate.

The inverse of the Weibull cdf is

$$x = F^{-1}(p | a, b) = -a [\ln(1 - p)]^{1/b} I_{[0,1]}(p)$$

## Examples

The lifetimes (in hours) of a batch of light bulbs has a Weibull distribution with parameters  $a = 200$  and  $b = 6$ .

Find the median lifetime of the bulbs:

```
life = wblinv(0.5, 200, 6)
life =
    188.1486
```

Generate 100 random values from this distribution, and estimate the 90th percentile (with confidence bounds) from the random sample

```
x = wblrnd(200,6,100,1);
p = wblfit(x)
[nlogl,pcov] = wbllike(p,x)
[q90,q90lo,q90up] = wblinv(0.9,p(1),p(2),pcov)
p =
```

```
    204.8918    6.3920
```

```
nlogl =
```

```
    496.8915
```

```
pcov =
```

```
    11.3392    0.5233
    0.5233    0.2573
```

```
q90 =
```

```
    233.4489
```

```
q90lo =
```

```
    226.0092
```

```
q90up =  
    241.1335
```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

[icdf](#) | [wblcdf](#) | [wblfit](#) | [wbllike](#) | [wblpdf](#) | [wblplot](#) | [wblrnd](#) | [wblstat](#)

### Topics

“Weibull Distribution” on page B-193

**Introduced before R2006a**

## wbllike

Weibull negative log-likelihood

### Syntax

```
nlogL = wbllike(params, data)
[logL, AVAR] = wbllike(params, data)
[...] = wbllike(params, data, censoring)
[...] = wbllike(params, data, censoring, freq)
```

### Description

`nlogL = wbllike(params, data)` returns the Weibull log-likelihood. `params(1)` is the scale parameter,  $A$ , and `params(2)` is the shape parameter,  $B$ .

`[logL, AVAR] = wbllike(params, data)` also returns `AVAR`, which is the asymptotic variance-covariance matrix of the parameter estimates if the values in `params` are the maximum likelihood estimates. `AVAR` is the inverse of Fisher's information matrix. The diagonal elements of `AVAR` are the asymptotic variances of their respective parameters.

`[...] = wbllike(params, data, censoring)` accepts a Boolean vector, `censoring`, of the same size as `data`, which is 1 for observations that are right-censored and 0 for observations that are observed exactly.

`[...] = wbllike(params, data, censoring, freq)` accepts a frequency vector, `freq`, of the same size as `data`. `freq` typically contains integer frequencies for the corresponding elements in `data`, but can contain any nonnegative values. Pass in `[]` for `censoring` to use its default value.

The Weibull negative log-likelihood for uncensored data is

$$(-\log L) = -\log \prod_{i=1} f(a, b | x_i) = -\sum_{i=1}^n \log f(a, b | x_i)$$

where  $f$  is the Weibull pdf.



wbllike is a utility function for maximum likelihood estimation.

## Examples

This example continues the example from `wblfit`.

```
r = wblrnd(0.5,0.8,100,1);
[logL, AVAR] = wbllike(wblfit(r),r)
logL =
    47.3349
AVAR =
    0.0048    0.0014
    0.0014    0.0040
```

## References

- [1] Patel, J. K., C. H. Kapadia, and D. B. Owen. *Handbook of Statistical Distributions*. New York: Marcel Dekker, 1976.

## See Also

`wblcdf` | `wblfit` | `wblinv` | `wblpdf` | `wblplot` | `wblrnd` | `wblstat`

## Topics

“Weibull Distribution” on page B-193

Introduced before R2006a

## wblpdf

Weibull probability density function

### Syntax

```
Y = wblpdf(X,A,B)
```

### Description

`Y = wblpdf(X,A,B)` computes the Weibull pdf at each of the values in `X` using the corresponding scale parameter, `A` and shape parameter, `B`. `X`, `A`, and `B` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array of the same size as the other inputs. The parameters in `A` and `B` must be positive.

The Weibull pdf is

$$f(x|a,b) = \frac{b}{a} \left(\frac{x}{a}\right)^{b-1} e^{-(x/a)^b}.$$

Some references refer to the Weibull distribution with a single parameter. This corresponds to `wblpdf` with `A = 1`.

### Examples

The exponential distribution is a special case of the Weibull distribution.

```
lambda = 1:6;
y = wblpdf(0.1:0.1:0.6,lambda,1)
y =
    0.9048    0.4524    0.3016    0.2262    0.1810    0.1508

y1 = exppdf(0.1:0.1:0.6,lambda)
y1 =
    0.9048    0.4524    0.3016    0.2262    0.1810    0.1508
```

## References

- [1] Devroye, L. *Non-Uniform Random Variate Generation*. New York: Springer-Verlag, 1986.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`pdf` | `wblcdf` | `wblfit` | `wblinv` | `wbllike` | `wblplot` | `wblrnd` | `wblstat`

### Topics

“Weibull Distribution” on page B-193

**Introduced before R2006a**

## wblplot

Weibull probability plot

### Syntax

```
wblplot(X)  
h = wblplot(X)
```

### Description

`wblplot(X)` displays a Weibull probability plot of the data in `X`. If `X` is a matrix, `wblplot` displays a plot for each column. Use a Weibull probability plot to assess visually whether the sample data in `X` comes from a population with a Weibull distribution. If the sample data has a Weibull distribution, then the data appears along the reference line. Distributions other than Weibull can introduce curvature in the plot.

`h = wblplot(X)` returns handles to the plotted lines.

### Examples

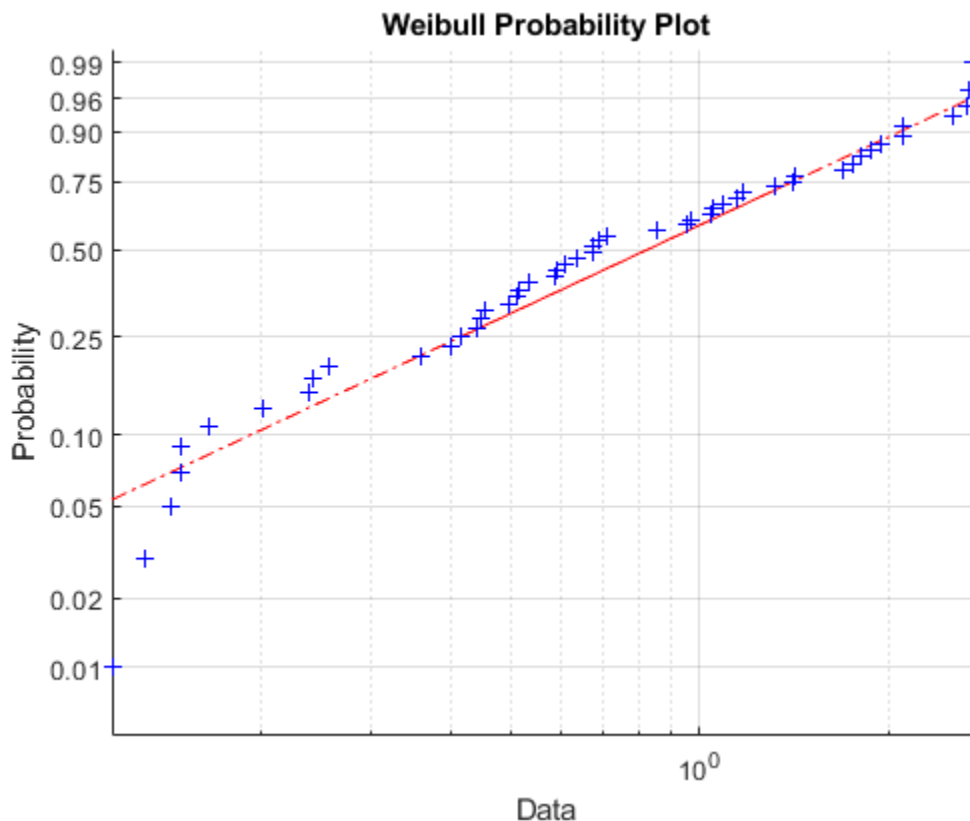
#### Create a Weibull Probability Plot

Generate a vector `r` containing 50 random numbers from a Weibull distribution with parameters `A = 1.2` and `B = 1.5`.

```
rng default; % For reproducibility  
r = wblrnd(1.2,1.5,50,1);
```

Create a Weibull probability plot to visually determine if the data comes from a Weibull distribution.

```
figure;  
wblplot(r)
```



The plot indicates that the data likely comes from a Weibull distribution.

## Algorithms

`wblplot` matches the quantiles of sample data to the quantiles of a Weibull distribution. The sample data is sorted, scaled logarithmically, and plotted on the x-axis. The y-axis represents the quantiles of the Weibull distribution, converted into probability values. Therefore, the y-axis scaling is not linear.

Where the x-axis value is the  $i$ th sorted value from a sample of size  $N$ , the y-axis value is the midpoint between evaluation points of the empirical cumulative distribution function

of the data. In the case of uncensored data, the midpoint is equal to  $\frac{(i-0.5)}{N}$ . When the data includes censored observations, use `probplot` instead.

`wblplot` superimposes a reference line to assess the linearity of the plot. The line goes through the first and third quartiles of the data.

## See Also

`ecdf` | `normplot` | `probplot` | `wblcdf` | `wblfit` | `wblinv` | `wbllike` | `wblpdf` | `wblrnd` | `wblstat`

## Topics

“Weibull Distribution” on page B-193

**Introduced before R2006a**

# wblrnd

Weibull random numbers

## Syntax

```
R = wblrnd(A,B)
R = wblrnd(A,B,m,n,...)
R = wblrnd(A,B,[m,n,...])
```

## Description

`R = wblrnd(A,B)` generates random numbers for the Weibull distribution with scale parameter, `A` and shape parameter, `B`. The input arguments `A` and `B` can be either scalars or matrices. `A` and `B`, can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array of the same size as the other input.

`R = wblrnd(A,B,m,n,...)` or `R = wblrnd(A,B,[m,n,...])` generates an `m`-by-`n`-by-... array. The `A`, `B` parameters can each be scalars or arrays of the same size as `R`.

Devroye [1] refers to the Weibull distribution with a single parameter; this is `wblrnd` with `A = 1`.

## Examples

```
n1 = wblrnd(0.5:0.5:2,0.5:0.5:2)
n1 =
    0.0178    0.0860    2.5216    0.9124

n2 = wblrnd(1/2,1/2,[1 6])
n2 =
    0.0046    1.7214    2.2108    0.0367    0.0531    0.0917
```

## References

[1] Devroye, L. *Non-Uniform Random Variate Generation*. New York: Springer-Verlag, 1986.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

The generated code can return a different sequence of numbers than MATLAB if either of the following is true:

- The output is nonscalar.
- An input parameter is invalid for the distribution.

### See Also

`random` | `wblcdf` | `wblfit` | `wblinv` | `wbllike` | `wblpdf` | `wblplot` | `wblstat`

### Topics

“Weibull Distribution” on page B-193

**Introduced before R2006a**



# wblstat

Weibull mean and variance

## Syntax

```
[M,V] = wblstat(A,B)
```

## Description

`[M,V] = wblstat(A,B)` returns the mean of and variance for the Weibull distribution with scale parameter, *A* and shape parameter, *B*. Vector or matrix inputs for *A* and *B* must have the same size, which is also the size of *M* and *V*. A scalar input for *A* or *B* is expanded to a constant matrix with the same dimensions as the other input.

The mean of the Weibull distribution with parameters *a* and *b* is

$$a \left[ \Gamma(1 + b^{-1}) \right]$$

and the variance is

$$a^2 \left[ \Gamma(1 + 2b^{-1}) - \Gamma(1 + b^{-1})^2 \right]$$

## Examples

```
[m,v] = wblstat(1:4,1:4)
m =
    1.0000    1.7725    2.6789    3.6256
v =
    1.0000    0.8584    0.9480    1.0346

wblstat(0.5,0.7)
ans =
    0.6329
```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`wblcdf` | `wblfit` | `wblinv` | `wbllike` | `wblpdf` | `wblplot` | `wblrnd`

### Topics

“Weibull Distribution” on page B-193

Introduced before R2006a

# wishrnd

Wishart random numbers

## Syntax

```
W = wishrnd(Sigma,df)
W = wishrnd(Sigma,df,D)
[W,D] = wishrnd(Sigma,df)
```

## Description

`W = wishrnd(Sigma,df)` generates a random matrix `W` having the Wishart distribution with covariance matrix `Sigma` and with `df` degrees of freedom. The inverse of `W` has the Inverse Wishart distribution with parameters `Tau = inv(Sigma)` and `df` degrees of freedom.

`W = wishrnd(Sigma,df,D)` expects `D` to be the Cholesky factor of `Sigma`. If you call `wishrnd` multiple times using the same value of `Sigma`, it's more efficient to supply `D` instead of computing it each time.

`[W,D] = wishrnd(Sigma,df)` returns `D` so you can provide it as input in future calls to `wishrnd`.

This function defines the parameter `Sigma` so that the mean of the output matrix is `Sigma*df`

## See Also

`iwishrnd`

## Topics

“Wishart Distribution” on page B-198

**Introduced before R2006a**

## X property

**Class:** TreeBagger

X data used to create ensemble

### Description

The X property is a numeric matrix of size Nobs-by-Nvars, where Nobs is the number of observations (rows) and Nvars is the number of variables (columns) in the training data. This matrix contains the predictor (or feature) values.

## xptread

Create table from data stored in SAS XPORT format file

### Syntax

```
data = xptread
data = xptread(filename)
[data,missing] = xptread(filename)
xptread(..., 'ReadObsNames', true)
```

### Description

`data = xptread` displays a dialog box for selecting a file, then reads data from the file into a table. The file must be in the SAS XPORT format.

`data = xptread(filename)` retrieves data from a SAS XPORT format file `filename`. For example, to open a SAS XPORT file named `sample.xpt`, type `data = xptread('sample.xpt')` at the command prompt.

`[data,missing] = xptread(filename)` returns a nominal array, `missing`, that contains the missing data type information from the XPORT format file.

The XPORT format allows for 28 missing data types, represented in the file by an upper case letter or the characters '.' or '\_'. If the XPORT file contains missing values, `xptread` converts them to NaN values in the output table, `data`. However, if you need the specific missing types, you can recover this information by specifying a second output, `missing`. The entries in `missing` are one of the 28 missing data type values from the XPORT format file ('.', '\_', 'A', ..., 'Z'), or are undefined for values that are not present at all in the XPORT format file. The outputs `missing` and `data` are the same size.

`xptread(..., 'ReadObsNames', true)` treats the first variable in the file as observation names. The default value is `false`.

xptread only supports single data sets per file. xptread does not support compressed files.

## See Also

table

**Introduced in R2009b**

## x2fx

Convert predictor matrix to design matrix

### Syntax

```
D = x2fx(X, model)
D = x2fx(X, model, categ)
D = x2fx(X, model, categ, catlevels)
```

### Description

`D = x2fx(X, model)` converts a matrix of predictors  $X$  to a design matrix  $D$  for regression analysis. Distinct predictor variables should appear in different columns of  $X$ .

The optional input `model` controls the regression model. By default, `x2fx` returns the design matrix for a linear additive model with a constant term. `model` is one of the following:

- `'linear'` — Constant and linear terms. This is the default.
- `'interaction'` — Constant, linear, and interaction terms
- `'quadratic'` — Constant, linear, interaction, and squared terms
- `'purequadratic'` — Constant, linear, and squared terms

If  $X$  has  $n$  columns, the order of the columns of  $D$  for a full quadratic model is:

- 1 The constant term
- 2 The linear terms (the columns of  $X$ , in order 1, 2, ...,  $n$ )
- 3 The interaction terms (pairwise products of the columns of  $X$ , in order (1, 2), (1, 3), ..., (1,  $n$ ), (2, 3), ..., ( $n-1$ ,  $n$ ))
- 4 The squared terms (in order 1, 2, ...,  $n$ )

Other models use a subset of these terms, in the same order.



Alternatively, *model* can be a matrix specifying polynomial terms of arbitrary order. In this case, *model* should have one column for each column in *X* and one row for each term in the model. The entries in any row of *model* are powers for the corresponding columns of *X*. For example, if *X* has columns *X1*, *X2*, and *X3*, then a row `[0 1 2]` in *model* specifies the term  $(X1.^0) .* (X2.^1) .* (X3.^2)$ . A row of all zeros in *model* specifies a constant term, which can be omitted.

`D = x2fx(X, model, categ)` treats columns with numbers listed in the vector *categ* as categorical variables. Terms involving categorical variables produce dummy variable columns in *D*. Dummy variables are computed under the assumption that possible categorical levels are completely enumerated by the unique values that appear in the corresponding column of *X*.

`D = x2fx(X, model, categ, catlevels)` accepts a vector *catlevels* the same length as *categ*, specifying the number of levels in each categorical variable. In this case, values in the corresponding column of *X* must be integers in the range from 1 to the specified number of levels. Not all of the levels need to appear in *X*.

## Examples

### Example 1

The following converts 2 predictors *X1* and *X2* (the columns of *X*) into a design matrix for a full quadratic model with terms constant, *X1*, *X2*, *X1.\*X2*, *X1.^2*, and *X2.^2*.

```
X = [1 10
     2 20
     3 10
     4 20
     5 15
     6 15];
```

```
D = x2fx(X, 'quadratic')
```

```
D =
     1     1    10    10     1    100
     1     2    20    40     4    400
     1     3    10    30     9    100
     1     4    20    80    16    400
     1     5    15    75    25    225
     1     6    15    90    36    225
```

## Example 2

The following converts 2 predictors  $X_1$  and  $X_2$  (the columns of  $X$ ) into a design matrix for a quadratic model with terms `constant`,  $X_1$ ,  $X_2$ ,  $X_1 \cdot X_2$ , and  $X_1.^2$ .

```
X = [1 10
      2 20
      3 10
      4 20
      5 15
      6 15];
model = [0 0
         1 0
         0 1
         1 1
         2 0];

D = x2fx(X,model)
D =
     1     1    10    10     1
     1     2    20    40     4
     1     3    10    30     9
     1     4    20    80    16
     1     5    15    75    25
     1     6    15    90    36
```

## See Also

`candexch` | `candgen` | `cordexch` | `regstats` | `rowexch` | `rstool`

**Introduced before R2006a**

## Y property

**Class:** TreeBagger

Y data used to create ensemble

### Description

The Y property is an array of true class labels for classification, or response values for regression. Y can be a numeric column vector, a character matrix, or a cell array of character vectors.

## zscore

Standardized *z*-scores

### Syntax

```
Z = zscore(X)
Z = zscore(X, flag)
Z = zscore(X, flag, dim)
[Z, mu, sigma] = zscore(____)
```

### Description

`Z = zscore(X)` returns the *z*-score on page 32-6982 for each element of *X* such that columns of *X* are centered to have mean 0 and scaled to have standard deviation 1. *Z* is the same size as *X*.

- If *X* is a vector, then *Z* is a vector of *z*-scores.
- If *X* is a matrix, then *Z* is a matrix of the same size as *X*, and each column of *Z* has mean 0 and standard deviation 1.
- For multidimensional arrays on page 32-6983, *z*-scores in *Z* are computed along the first nonsingleton dimension on page 32-6983 of *X*.

`Z = zscore(X, flag)` scales *X* using the standard deviation indicated by *flag*.

- If *flag* is 0 (default), then `zscore` scales *X* using the sample standard deviation on page 32-6983, with  $n - 1$  in the denominator of the standard deviation formula. `zscore(X, 0)` is the same as `zscore(X)`.
- If *flag* is 1, then `zscore` scales *X* using the population standard deviation on page 32-6983, with  $n$  in the denominator of standard deviation formula.

`Z = zscore(X, flag, dim)` standardizes *X* along dimension *dim*. For example, for a matrix *X*, if *dim* = 1, then `zscore` uses the means and standard deviations along the columns of *X*, if *dim* = 2, then `zscore` uses the means and standard deviations along the rows of *X*.

`[Z,mu,sigma] = zscore(____)` also returns the means and standard deviations used for centering and scaling, `mu` and `sigma`, respectively. You can use any of the input arguments in the previous syntaxes.

## Examples

### Z-Scores of Two Data Vectors

Compute and plot the *z*-scores of two data vectors, and then compare the results.

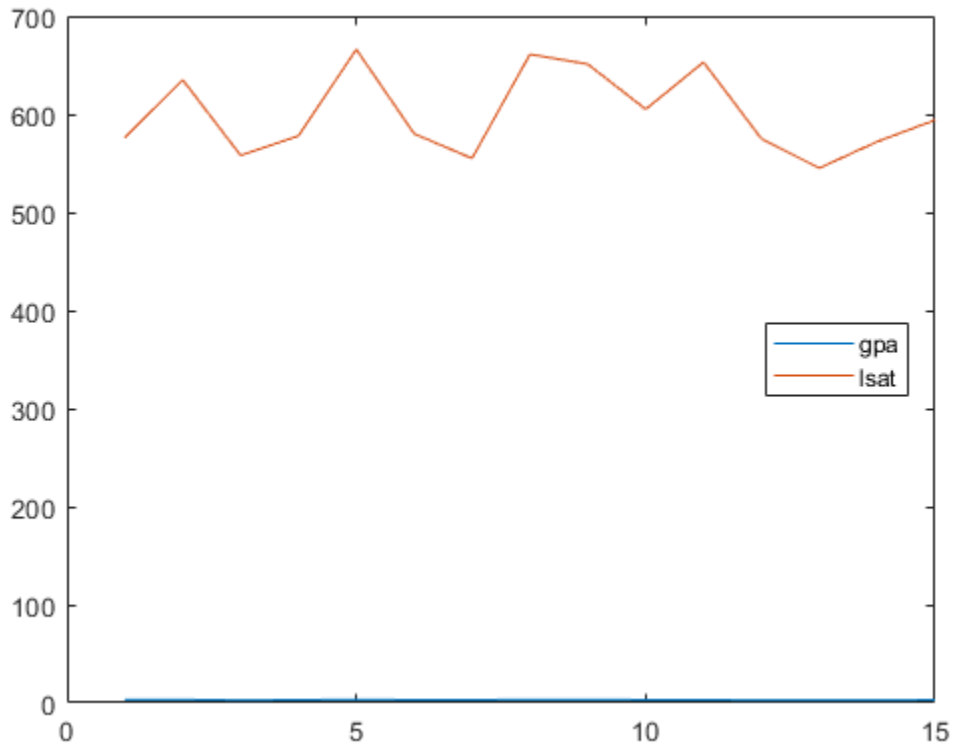
Load the sample data.

```
load lawdata
```

Two variables load into the workspace: `gpa` and `lsat`.

Plot both variables on the same axes.

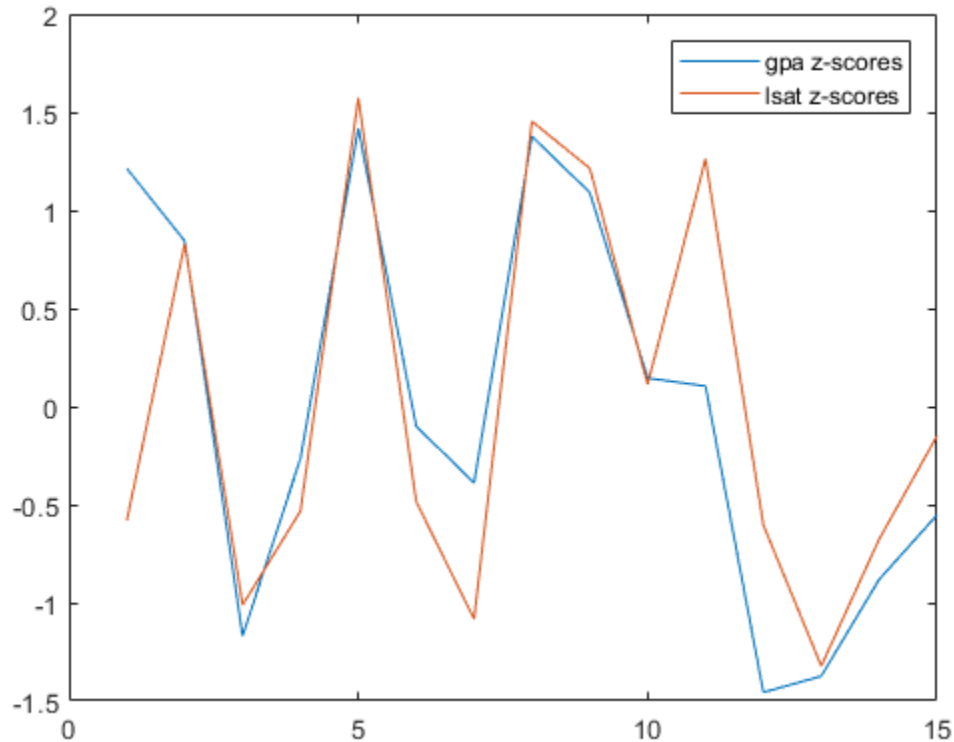
```
plot([gpa,lsat])  
legend('gpa','lsat','Location','East')
```



It is difficult to compare these two measures because they are on a very different scale.

Plot the  $z$ -scores of `gpa` and `lsat` on the same axes.

```
Zgpa = zscore(gpa);  
Zlsat = zscore(lsat);  
plot([Zgpa, Zlsat])  
legend('gpa z-scores', 'lsat z-scores', 'Location', 'Northeast')
```



Now, you can see the relative performance of individuals with respect to both their `gpa` and `lsat` results. For example, the third individual's `gpa` and `lsat` results are both one standard deviation below the sample mean. The eleventh individual's `gpa` is around the sample mean but has an `lsat` score almost 1.25 standard deviations above the sample average.

Check the mean and standard deviation of the `z`-scores you created.

```
mean([Zgpa, Zlsat])
```

```
ans =
```

```
1.0e-14 *
```

```
-0.1088    0.0357

std([Zgpa,Zlsat])

ans =

    1     1
```

By definition,  $z$ -scores of `gpa` and `lsat` have mean 0 and standard deviation 1.

### Z-Scores for a Population vs. Sample

Load the sample data.

```
load lawdata
```

Two variables load into the workspace: `gpa` and `lsat`.

Compute the  $z$ -scores of `gpa` using the population formula for standard deviation.

```
Z1 = zscore(gpa,1); % population formula
Z0 = zscore(gpa,0); % sample formula
disp([Z1 Z0])
```

```
1.2554    1.2128
0.8728    0.8432
-1.2100   -1.1690
-0.2749   -0.2656
1.4679    1.4181
-0.1049   -0.1013
-0.4024   -0.3888
1.4254    1.3771
1.1279    1.0896
0.1502    0.1451
0.1077    0.1040
-1.5076   -1.4565
-1.4226   -1.3743
-0.9125   -0.8815
-0.5724   -0.5530
```



For a sample from a population, the population standard deviation formula with  $n$  in the denominator corresponds to the maximum likelihood estimate of the population standard deviation, and might be biased. The sample standard deviation formula, on the other hand, is the unbiased estimator of the population standard deviation for a sample.

### Z-Scores of a Data Matrix

Compute  $z$ -scores using the mean and standard deviation computed along the columns or rows of a data matrix.

Load the sample data.

```
load flu
```

The dataset array `flu` is loaded in the workspace. `flu` has 52 observations on 11 variables. The first variable contains dates (in weeks). The other variables contain the flu estimates for different regions in the U.S.

Convert the dataset array to a data matrix.

```
flu2 = double(flu(:,2:end));
```

The new data matrix, `flu2`, is a 52-by-10 double data matrix. The rows correspond to the weeks and the columns correspond to the U.S. regions in the data set array `flu`.

Standardize the flu estimate for each region (the *columns* of `flu2`).

```
Z1 = zscore(flu2, [ ], 1);
```

You can see the  $z$ -scores in the variable editor by double-clicking on the matrix `Z1` created in the workspace.

Standardize the flu estimate for each week (the *rows* of `flu2`).

```
Z2 = zscore(flu2, [ ], 2);
```

### Z-Scores, Mean, and Standard Deviation

Return the mean and standard deviation used to compute the  $z$ -scores.

Load the sample data.

```
load lawdata
```

Two variables load into the workspace: `gpa` and `lsat`.

Return the  $z$ -scores, mean, and standard deviation of `gpa`.

```
[Z, gpamean, gpastdev] = zscore(gpa)
```

```
Z =
```

```
    1.2128  
    0.8432  
   -1.1690  
   -0.2656  
    1.4181  
   -0.1013  
   -0.3888  
    1.3771  
    1.0896  
    0.1451  
    0.1040  
   -1.4565  
   -1.3743  
   -0.8815  
   -0.5530
```

```
gpamean =
```

```
    3.0947
```

```
gpastdev =
```

---

0.2435

## Input Arguments

### **x** — Input data

vector | matrix | multidimensional array

Input data, specified as a vector, matrix, or multidimensional array.

Data Types: `double` | `single`

### **flag** — Indicator for the standard deviation

0 (default) | 1

Indicator for the standard deviation used to compute the *z*-scores, specified as 0 or 1.

- If `flag` is 0 (default), then `zscore` scales *X* using the sample standard deviation on page 32-6983. `zscore(X, 0)` is the same as `zscore(X)`.
- If `flag` is 1, then `zscore` scales *X* using the population standard deviation on page 32-6983.

### **dim** — Dimension

1 (default) | positive integer

Dimension along which to calculate the *z*-scores of *X*, specified as a positive integer. For example, for a matrix *X*, if `dim = 1`, then `zscore` uses the means and standard deviations along the columns of *X*, if `dim = 2`, then `zscore` uses the means and standard deviations along the rows of *X*.

## Output Arguments

### **z** — *z*-scores

vector | matrix | multidimensional array

*z*-scores, returned as a vector, matrix, or multidimensional array. A vector of *z*-scores has mean 0 and variance 1.

- If  $X$  is a vector, then  $Z$  is a vector of  $z$ -scores.
- If  $X$  is an array, then `zscore` is an array, with each column or row standardized to have mean 0 and variance 1 (depending on `dim`). If `dim` is not specified, `zscore` standardizes along the first nonsingleton dimension on page 32-6983 of  $X$ .

**mu — Mean**

scalar | vector

Mean of  $X$  used to compute the  $z$ -scores, returned as a scalar or vector.

- If  $X$  is a vector, then `mu` is a scalar.
- If  $X$  is a matrix, then `mu` is a row vector if `zscore` calculates the means along the columns of  $X$  (`dim = 1`), and a column vector if `zscore` calculates the means along the rows of  $X$  (`dim = 2`).

**sigma — Standard deviation**

scalar | vector

Standard deviation of  $X$  used to compute the  $z$ -scores, returned as a scalar or vector.

- If  $X$  is a vector, then `sigma` is a scalar.
- If  $X$  is a matrix, then `sigma` is a row vector if `zscore` calculates the standard deviations along the columns of  $X$  (`dim = 1`), and a column vector if `zscore` calculates the standard deviations along the rows of  $X$  (`dim = 2`).

## Definitions

### Z-Score

For a random variable  $X$  with mean  $\mu$  and standard deviation  $\sigma$ , the  $z$ -score of a value  $x$  is

$$z = \frac{(x - \mu)}{\sigma}.$$

For sample data with mean  $\bar{X}$  and standard deviation  $S$ , the  $z$ -score of a data point  $x$  is

$$z = \frac{(x - \bar{X})}{S}.$$

$z$ -scores measure the distance of a data point from the mean in terms of the standard deviation. This is also called *standardization* of data. The standardized data set has mean 0 and standard deviation 1, and retains the shape properties of the original data set (same skewness and kurtosis).

You can use  $z$ -scores to put data on the same scale before further analysis. This lets you to compare two or more data sets with different units.

## Multidimensional Array

A *multidimensional array* is an array with more than two dimensions. For example, if  $X$  is a 1-by-3-by-4 array, then  $X$  is a three-dimensional array.

## First Nonsingleton Dimension

A *first nonsingleton dimension* is the first dimension of an array whose size is not equal to 1. For example, if  $X$  is a 1-by-2-by-3-by-4 array, then the second dimension is the first nonsingleton dimension of  $X$ .

## Sample Standard Deviation

The *sample standard deviation*,  $S$ , is given by

$$S = \sqrt{\frac{\sum_{i=1}^n (x_i - \bar{X})^2}{n - 1}}.$$

$S$  is the square root of an unbiased estimator of the variance of the population from which  $X$  is drawn, as long as  $X$  consists of independent, identically distributed samples.

Notice that the denominator in this variance formula is  $n - 1$ .

## Population Standard Deviation

If the data is the entire population of values, then you can use the *population standard deviation*,

$$\sigma = \sqrt{\frac{\sum_{i=1}^n (x_i - \mu)^2}{n}}.$$

If  $X$  is a random sample from a population, then  $\mu$  is estimated by the sample mean, and  $\sigma$  is the biased maximum likelihood estimator of the population standard deviation.

Notice that the denominator in this variance formula is  $n$ .

## Algorithms

`zscore` returns NaNs for any sample containing NaNs.

`zscore` returns 0s for any sample that is constant (all values are the same). For example, if  $X$  is a vector of the same numeric value, then  $Z$  is a vector of 0s. If  $X$  is a matrix with a column of consisting of the same value, then that column of  $Z$  consists of 0s.

## Extended Capabilities

### Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays” (MATLAB).

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`mean` | `std`

Introduced before R2006a

## ztest

z-test

### Syntax

```
h = ztest(x,m,sigma)
h= ztest(x,m,sigma,Name,Value)
[h,p] = ztest(____)
[h,p,ci,zval] = ztest(____)
```

### Description

`h = ztest(x,m,sigma)` returns a test decision for the null hypothesis that the data in the vector `x` comes from a normal distribution with mean `m` and a standard deviation `sigma`, using the *z*-test on page 32-6990. The alternative hypothesis is that the mean is not `m`. The result `h` is 1 if the test rejects the null hypothesis at the 5% significance level, and 0 otherwise.

`h= ztest(x,m,sigma,Name,Value)` returns a test decision for the *z*-test with additional options specified by one or more name-value pair arguments. For example, you can change the significance level or conduct a one-sided test.

`[h,p] = ztest(____)` also returns the *p*-value of the test, using any of the input arguments from previous syntaxes.

`[h,p,ci,zval] = ztest(____)` also returns the confidence interval of the population mean, `ci`, and the value of the test statistic, `zval`.

### Examples

### z-Test for a Hypothesized Mean

Load the sample data. Create a vector containing the first column of the students' exam grades data.

```
load examgrades  
x = grades(:,1);
```

Test the null hypothesis that the data comes from a normal distribution with mean  $m = 75$  and standard deviation  $\sigma = 10$ .

```
[h,p,ci,zval] = ztest(x,75,10)
```

```
h =  
  
    0
```

```
p =  
  
    0.9927
```

```
ci =  
  
    73.2191  
    76.7975
```

```
zval =  
  
    0.0091
```

The returned value of  $h = 0$  indicates that `ztest` does not reject the null hypothesis at the default 5% significance level.

### One-Sided z-Test

Load the sample data. Create a vector containing the first column of the students' exam grades data.



```
load examgrades
x = grades(:,1);
```

Test the null hypothesis that the data comes from a normal distribution with mean  $m = 65$  and standard deviation  $\sigma = 10$ , against the alternative that the mean is greater than 65.

```
[h,p] = ztest(x,65,10,'Tail','right')
```

```
h =
```

```
1
```

```
p =
```

```
2.8596e-28
```

The returned value of  $h = 1$  indicates that `ztest` rejects the null hypothesis at the default 5% significance level, in favor of the alternative hypothesis that the population mean is greater than 65.

## Input Arguments

### **x** — Sample data

vector | matrix | multidimensional array

Sample data, specified as a vector, matrix, or multidimensional array.

- If  $x$  is specified as a vector, `ztest` returns a single value for each output argument.
- If  $x$  is specified as a matrix, `ztest` performs a separate  $z$ -test along each column of  $x$  and returns a vector of results.
- If  $x$  is specified as a multidimensional array on page 32-6990, `ztest` works along the first nonsingleton dimension on page 32-6990 of  $x$ .

In all cases, `ztest` treats NaN values as missing data and ignores them.

Data Types: `single` | `double`

**m — Hypothesized mean**

scalar value

Hypothesized mean, specified as a scalar value.

Data Types: `single` | `double`**sigma — Population standard deviation**

scalar value

Population standard deviation, specified as a scalar value.

Data Types: `single` | `double`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'Tail', 'right', 'Alpha', 0.01` specifies a right-tailed hypothesis test at the 1% significance level.

**Alpha — Significance level**

0.05 (default) | scalar value in the range (0,1)

Significance level of the hypothesis test, specified as the comma-separated pair consisting of `'Alpha'` and a scalar value in the range (0,1).

Example: `'Alpha', 0.01`

Data Types: `single` | `double`**Dim — Dimension**

first nonsingleton dimension (default) | positive integer value

Dimension of the input matrix along which to test the means, specified as the comma-separated pair consisting of `'Dim'` and a positive integer value. For example, specifying `'Dim', 1` tests the column means, while `'Dim', 2` tests the row means.

Example: `'Dim', 2`

Data Types: `single` | `double`

**Tail — Type of alternative hypothesis**`'both' (default) | 'right' | 'left'`

Type of alternative hypothesis to evaluate, specified as the comma-separated pair consisting of 'Tail' and one of the following.

<code>'both'</code>	Test the alternative hypothesis that the population mean is not equal to $m$ .
<code>'right'</code>	Test the alternative hypothesis that the population mean is greater than $m$ .
<code>'left'</code>	Test the alternative hypothesis that the population mean is less than $m$ .

Example: `'Tail', 'right'`

## Output Arguments

**h — Hypothesis test result**`1 | 0`

Hypothesis test result, returned as a logical value.

- If  $h = 1$ , this indicates the rejection of the null hypothesis at the Alpha significance level.
- If  $h = 0$ , this indicates a failure to reject the null hypothesis at the Alpha significance level.

**p —  $p$ -value**

scalar value in the range [0,1]

$p$ -value of the test, returned as a scalar value in the range [0,1].  $p$  is the probability of observing a test statistic as extreme as, or more extreme than, the observed value under the null hypothesis. Small values of  $p$  cast doubt on the validity of the null hypothesis.

**ci — Confidence interval**

vector

Confidence interval for the true population mean, returned as a two-element vector containing the lower and upper boundaries of the  $100 \times (1 - \text{Alpha})\%$  confidence interval.

**zval** — Test statistic

nonnegative scalar value

Test statistic, returned as a nonnegative scalar value.

## Definitions

### z-Test

The *z*-test is a parametric hypothesis test used to determine whether a sample data set comes from a population with a particular mean. The test assumes that the sample data comes from a population with a normal distribution and a known standard deviation.

The test statistic is

$$z = \frac{\bar{x} - \mu}{\sigma / \sqrt{n}},$$

where  $\bar{x}$  is the sample mean,  $\mu$  is the population mean,  $\sigma$  is the population standard deviation, and  $n$  is the sample size. Under the null hypothesis, the test statistic has a standard normal distribution.

### Multidimensional Array

A multidimensional array has more than two dimensions. For example, if  $x$  is a 1-by-3-by-4 array, then  $x$  is a three-dimensional array.

### First Nonsingleton Dimension

The first nonsingleton dimension is the first dimension of an array whose size is not equal to 1. For example, if  $x$  is a 1-by-2-by-3-by-4 array, then the second dimension is the first nonsingleton dimension of  $x$ .

## Tips

- Use `sampsizepwr` to calculate:

- The sample size that corresponds to specified power and parameter values;
- The power achieved for a particular sample size, given the true parameter value;
- The parameter value detectable with the specified sample size and power.

## See Also

`sampsizepwr` | `ttest` | `ttest2`

**Introduced before R2006a**

## hmcSampler

Hamiltonian Monte Carlo (HMC) sampler

### Syntax

```
hmc = hmcSampler(logpdf, startpoint)
hmc = hmcSampler( ____, Name, Value)
```

### Description

`hmc = hmcSampler(logpdf, startpoint)` creates a Hamiltonian Monte Carlo (HMC) sampler, returned as a `HamiltonianSampler` object. `logpdf` is a function handle that evaluates the logarithm of the probability density of the equilibrium distribution and its gradient. The column vector `startpoint` is the initial point from which to start HMC sampling.

After you create the sampler, you can compute MAP (maximum-a-posteriori) point estimates, tune the sampler, draw samples, and check convergence diagnostics using the methods of the `HamiltonianSampler` class. For an example of this workflow, see [Bayesian Linear Regression Using Hamiltonian Monte Carlo](#) on page 7-35.

`hmc = hmcSampler( ____, Name, Value)` specifies additional options using one or more name-value pair arguments. Specify name-value pair arguments after all other input arguments.

### Examples

#### Create Hamiltonian Monte Carlo Sampler

Create a Hamiltonian Monte Carlo (HMC) sampler to sample from a normal distribution.

First, save a function `normalDistGrad` on the MATLAB® path that returns the multivariate normal log probability density and its gradient (`normalDistGrad` is

defined at the end of this example). Then, call the function with arguments to define the logpdf input argument to the hmcSampler function.

```
means = [1;-3];
standevs = [1;2];
logpdf = @(theta)normalDistGrad(theta,means,standevs);
```

Choose a starting point for the HMC sampler.

```
startpoint = randn(2,1);
```

Create the HMC sampler and display its properties.

```
smp = hmcSampler(logpdf,startpoint);
```

```
smp
```

```
smp =
```

```
HamiltonianSampler with properties:
    StepSize: 0.1000
    NumSteps: 50
    MassVector: [2x1 double]
    JitterMethod: 'jitter-both'
    StepSizeTuningMethod: 'dual-averaging'
    MassVectorTuningMethod: 'iterative-sampling'
    LogPDF: @(theta)normalDistGrad(theta,means,standevs)
    VariableNames: {2x1 cell}
    StartPoint: [2x1 double]
```

The normalDistGrad function returns the logarithm of the multivariate normal probability density with means in Mu and standard deviations in Sigma, specified as scalars or columns vectors the same length as startpoint. The second output argument is the corresponding gradient.

```
function [lpdf,glpdf] = normalDistGrad(X,Mu,Sigma)
Z = (X - Mu)./Sigma;
lpdf = sum(-log(Sigma) - .5*log(2*pi) - .5*(Z.^2));
```

```
glpdf = -Z./Sigma;  
end
```

- “Bayesian Linear Regression Using Hamiltonian Monte Carlo” on page 7-35

## Input Arguments

### **logpdf** — Logarithm of target density and its gradient

function handle

Logarithm of target density and its gradient, specified as a function handle.

`logpdf` must return two output arguments: `[lpdf, glpdf] = logpdf(X)`. Here, `lpdf` is the base-e log probability density (up to an additive constant), `glpdf` is the gradient of the log density, and the point `X` is a column vector with the same number of elements as `startpoint`.

The input argument `X` to `logpdf` must be unconstrained, meaning that every element of `X` can be any real number. Transform any constrained sampling parameters into unconstrained variables before using the HMC sampler.

If the `'UseNumericalGradient'` value is set to `true`, then `logpdf` does not need to return the gradient as the second output. Using a numerical gradient can be easier since `logpdf` does not need to compute the gradient, but it can make sampling slower.

Data Types: `function_handle`

### **startpoint** — Initial point to start sampling from

numeric column vector

Initial point to start sampling from, specified as a numeric column vector.

Data Types: `single` | `double`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.



Example: `'VariableNames', {'Intercept', 'Beta'}, 'MassVectorTuningMethod', 'hessian'` specifies sampling variable names and the mass vector tuning method to be 'hessian'.

### **StepSize** — Step size of Hamiltonian dynamics

0.1 (default) | positive scalar

Step size of Hamiltonian dynamics, specified as the comma-separated pair consisting of 'StepSize' and a positive scalar.

To propose a new state for the Markov chain, the HMC sampler integrates the Hamiltonian dynamics using leapfrog integration. This argument controls the step size of that leapfrog integration.

You can automatically tune the step size using `tuneSampler`.

Example: `'StepSize', 0.2`

### **NumSteps** — Number of steps of Hamiltonian dynamics

50 (default) | positive integer

Number of steps of Hamiltonian dynamics, specified as the comma-separated pair consisting of 'NumSteps' and a positive integer.

To propose a new state for the Markov chain, the HMC sampler integrates the Hamiltonian dynamics using leapfrog integration. This argument controls the number of steps of that leapfrog integration.

You can automatically tune the number of steps using `tuneSampler`.

Example: `'NumSteps', 20`

### **MassVector** — Mass vector of momentum variables

`ones(size(startpoint,1),1)` (default) | numeric column vector

Mass vector of momentum variables, specified as the comma-separated pair consisting of 'MassVector' and a numeric column vector with positive values and the same length as `startpoint`.

The “masses” of the momentum variables associated with the variables of interest control the Hamiltonian dynamics in each Markov chain proposal.

You can automatically tune the mass vector using `tuneSampler`.

Example: `'MassVector', rand(3,1)`

### **JitterMethod** — Method for jittering step size and number of steps

`'jitter-both'` (default) | `'jitter-numsteps'` | `'none'`

Method for jittering the step size and the number of steps, specified as the comma-separated pair consisting of `'JitterMethod'` and one of the following values.

Value	Description
<code>'jitter-both'</code>	Randomly jitter the step size and number of steps for each leapfrog trajectory.
<code>'jitter-numsteps'</code>	Jitter only the number of steps of each leapfrog trajectory.
<code>'none'</code>	Perform no jittering.

With jittering, the sampler randomly selects the step size or the number of steps of each leapfrog trajectory as values smaller than the `'StepSize'` and `'NumSteps'` values. Use jittering to improve the stability of the leapfrog integration of the Hamiltonian dynamics.

Example: `'JitterMethod', 'jitter-both'`

### **StepSizeTuningMethod** — Method for tuning sampler step size

`'dual-averaging'` (default) | `'none'`

Method for tuning the sampler step size, specified as the comma-separated pair consisting of `'StepSizeTuningMethod'` and `'dual-averaging'` or `'none'`.

If the `'StepSizeTuningMethod'` value is set to `'dual-averaging'`, then `tuneSampler` tunes the leapfrog step size of the HMC sampler to achieve a certain acceptance ratio for a fixed value of the simulation length. The simulation length equals the step size multiplied by the number of steps. To set the target acceptance ratio, use the `'TargetAcceptanceRatio'` name-value pair argument of the `tuneSampler` method.

Example: `'StepSizeTuningMethod', 'none'`

### **MassVectorTuningMethod** — Method for tuning sampler mass vector

`'iterative-sampling'` (default) | `'hessian'` | `'none'`

Method for tuning the sampler mass vector, specified as the comma-separated pair consisting of `'MassVectorTuningMethod'` and one of the following values.

Value	Description
'iterative-sampling'	Tune the <code>MassVector</code> via successive approximations by drawing samples using a sequence of mass vector estimates.
'hessian'	Set the <code>MassVector</code> equal to the negative diagonal Hessian of the <code>logpdf</code> at the startpoint.
'none'	Perform no tuning of the <code>MassVector</code> .

To perform the tuning, use the `tuneSampler` method.

Example: `'MassVectorTuningMethod', 'hessian'`

#### **CheckGradient** — Flag for checking analytical gradient

`true` (or 1) (default) | `false` (or 0)

Flag for checking the analytical gradient, specified as the comma-separated pair consisting of `'CheckGradient'` and either `true` (or 1) or `false` (or 0).

If `'CheckGradient'` is `true`, then the sampler calculates the numerical gradient at the startpoint and compares it to the analytical gradient returned by `logpdf`.

Example: `'CheckGradient', true`

#### **VariableNames** — Sampling variable names

`{'x1', 'x2', ...}` (default) | cell array of unique character vectors

Sampling variable names, specified as the comma-separated pair consisting of `'VariableNames'` and a cell array of unique character vectors. The length of the cell array must be the same as the length of `startpoint`.

Supply a `'VariableNames'` value to label the components of the vector you want to sample using the HMC sampler.

Example: `'VariableNames', {'Intercept', 'Beta'}`

#### **UseNumericalGradient** — Flag for using numerical gradient

`false` (or 0) (default) | `true` (or 1)

Flag for using numerical gradient, specified as the comma-separated pair consisting of `'UseNumericalGradient'` and either `true` (or 1) or `false` (or 0).

If you set the `'UseNumericalGradient'` value to `true`, then the HMC sampler numerically estimates the gradient from the log density returned by `logpdf`. In this case, the `logpdf` function does not need to return the gradient of the log density as the second output. Using a numerical gradient makes HMC sampling slower.

Example: `'UseNumericalGradient', true`

## Output Arguments

### **hmc** — Hamiltonian Monte Carlo sampler

HamiltonianSampler object

Hamiltonian Monte Carlo sampler, returned as a `HamiltonianSampler` object.

## See Also

### **Functions**

`mhsample` | `slicesample`

### **Classes**

`HamiltonianSampler`

## Topics

“Bayesian Linear Regression Using Hamiltonian Monte Carlo” on page 7-35

**Introduced in R2017a**

# HamiltonianSampler class

Hamiltonian Monte Carlo (HMC) sampler

## Description

A Hamiltonian Monte Carlo (HMC) sampler is a gradient-based Markov Chain Monte Carlo sampler that you can use to generate samples from a probability density  $P(x)$ . HMC sampling requires specification of  $\log P(x)$  and its gradient.

The parameter vector  $x$  must be unconstrained, meaning that every element of  $x$  can be any real number. To sample constrained parameters, transform these parameters into unconstrained variables before using the HMC sampler.

After creating a sampler, you can compute MAP (maximum-a-posteriori) point estimates, tune the sampler, draw samples, and check convergence diagnostics using the methods of this class. For an example of this workflow, see [Bayesian Linear Regression Using Hamiltonian Monte Carlo](#) on page 7-35.

## Construction

`hmc = hmcSampler(logpdf, startpoint)` creates a Hamiltonian Monte Carlo (HMC) sampler, returned as a `HamiltonianSampler` object. `logpdf` is a function handle that evaluates the logarithm of the probability density of the equilibrium distribution and its gradient. The column vector `startpoint` is the initial point from which to start HMC sampling.

`hmc = hmcSampler( ___, Name, Value)` specifies additional options using one or more name-value pair arguments. Specify name-value pair arguments after all other input arguments.

## Input Arguments

**logpdf** — **Logarithm of target density and its gradient**  
function handle

Logarithm of target density and its gradient, specified as a function handle.

`logpdf` must return two output arguments: `[lpdf, glpdf] = logpdf(X)`. Here, `lpdf` is the base-e log probability density (up to an additive constant), `glpdf` is the gradient of the log density, and the point `X` is a column vector with the same number of elements as `startpoint`.

The input argument `X` to `logpdf` must be unconstrained, meaning that every element of `X` can be any real number. Transform any constrained sampling parameters into unconstrained variables before using the HMC sampler.

If the `'UseNumericalGradient'` value is set to `true`, then `logpdf` does not need to return the gradient as the second output. Using a numerical gradient can be easier since `logpdf` does not need to compute the gradient, but it can make sampling slower.

Data Types: `function_handle`

#### **startpoint** — Initial point to start sampling from

numeric column vector

Initial point to start sampling from, specified as a numeric column vector.

Data Types: `single` | `double`

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'VariableNames', {'Intercept', 'Beta'}, 'MassVectorTuningMethod', 'hessian'` specifies sampling variable names and the mass vector tuning method to be `'hessian'`.

#### **StepSize** — Step size of Hamiltonian dynamics

0.1 (default) | positive scalar

Step size of Hamiltonian dynamics, specified as the comma-separated pair consisting of `'StepSize'` and a positive scalar.

To propose a new state for the Markov chain, the HMC sampler integrates the Hamiltonian dynamics using leapfrog integration. This argument controls the step size of that leapfrog integration.

You can automatically tune the step size using `tuneSampler`.

Example: `'StepSize', 0.2`

### **NumSteps** — Number of steps of Hamiltonian dynamics

50 (default) | positive integer

Number of steps of Hamiltonian dynamics, specified as the comma-separated pair consisting of `'NumSteps'` and a positive integer.

To propose a new state for the Markov chain, the HMC sampler integrates the Hamiltonian dynamics using leapfrog integration. This argument controls the number of steps of that leapfrog integration.

You can automatically tune the number of steps using `tuneSampler`.

Example: `'NumSteps', 20`

### **MassVector** — Mass vector of momentum variables

`ones(size(startpoint,1),1)` (default) | numeric column vector

Mass vector of momentum variables, specified as the comma-separated pair consisting of `'MassVector'` and a numeric column vector with positive values and the same length as `startpoint`.

The “masses” of the momentum variables associated with the variables of interest control the Hamiltonian dynamics in each Markov chain proposal.

You can automatically tune the mass vector using `tuneSampler`.

Example: `'MassVector', rand(3,1)`

### **JitterMethod** — Method for jittering step size and number of steps

`'jitter-both'` (default) | `'jitter-numsteps'` | `'none'`

Method for jittering the step size and number of steps, specified as the comma-separated pair consisting of `'JitterMethod'` and one of the following:

Value	Description
<code>'jitter-both'</code>	Randomly jitter the step size and number of steps for each leapfrog trajectory.

Value	Description
'jitter-numsteps'	Jitter only the number of steps of each leapfrog trajectory.
'none'	Perform no jittering.

With jittering, the sampler randomly selects the step size or the number of steps of each leapfrog trajectory as values smaller than the 'StepSize' and 'NumSteps' values. Use jittering to improve the stability of the leapfrog integration of the Hamiltonian dynamics.

Example: 'JitterMethod', 'jitter-both'

#### **StepSizeTuningMethod** — Method for tuning sampler step size

'dual-averaging' (default) | 'none'

Method for tuning the sampler step size, specified as the comma-separated pair consisting of 'StepSizeTuningMethod' and 'dual-averaging' or 'none'.

If the 'StepSizeTuningMethod' value is set to 'dual-averaging', then `tuneSampler` tunes the leapfrog step size of the HMC sampler to achieve a certain acceptance ratio for a fixed value of the simulation length. The simulation length equals the step size multiplied by the number of steps. To set the target acceptance ratio, use the 'TargetAcceptanceRatio' name-value pair argument of the `tuneSampler` method.

Example: 'StepSizeTuningMethod', 'none'

#### **MassVectorTuningMethod** — Method for tuning sampler mass vector

'iterative-sampling' (default) | 'hessian' | 'none'

Method for tuning the sampler mass vector, specified as the comma-separated pair consisting of 'MassVectorTuningMethod' and one of the following values

Value	Description
'iterative-sampling'	Tune the <code>MassVector</code> via successive approximations by drawing samples using a sequence of mass vector estimates.
'hessian'	Set the <code>MassVector</code> equal to the negative diagonal Hessian of the <code>logpdf</code> at the startpoint.



Value	Description
'none'	Perform no tuning of the MassVector.

To perform the tuning, use the `tuneSampler` method.

Example: `'MassVectorTuningMethod', 'hessian'`

#### **CheckGradient** — Flag for checking analytical gradient

`true` (or 1) (default) | `false` (or 0)

Flag for checking the analytical gradient, specified as the comma-separated pair consisting of `'CheckGradient'` and either `true` (or 1) or `false` (or 0).

If `'CheckGradient'` is `true`, then the sampler calculates the numerical gradient at the startpoint and compares it to the analytical gradient returned by `logpdf`.

Example: `'CheckGradient', true`

#### **VariableNames** — Sampling variable names

`{'x1', 'x2', ...}` (default) | cell array of unique character vectors

Sampling variable names, specified as the comma-separated pair consisting of `'VariableNames'` and a cell array of unique character vectors. The length of the cell array must be the same as the length of `startpoint`.

Supply a `'VariableNames'` value to label the components of the vector you want to sample using the HMC sampler.

Example: `'VariableNames', {'Intercept', 'Beta'}`

#### **UseNumericalGradient** — Flag for using numerical gradient

`false` (or 0) (default) | `true` (or 1)

Flag for using numerical gradient, specified as the comma-separated pair consisting of `'UseNumericalGradient'` and either `true` (or 1) or `false` (or 0).

If you set the `'UseNumericalGradient'` value to `true`, then the HMC sampler numerically estimates the gradient from the log density returned by `logpdf`. In this case, the `logpdf` function does not need to return the gradient of the log density as the second output. Using a numerical gradient makes HMC sampling slower.

Example: `'UseNumericalGradient', true`

## Properties

### **StepSize** — Step size of Hamiltonian dynamics

0.1 (default) | positive scalar

Step size of Hamiltonian dynamics, specified as a positive scalar.

To propose a new state for the Markov chain, the HMC sampler integrates the Hamiltonian dynamics using leapfrog integration. The value of this property controls the step size of that leapfrog integration.

### **NumSteps** — Number of steps of Hamiltonian dynamics

50 (default) | positive integer

Number of steps of Hamiltonian dynamics, specified as a positive integer.

To propose a new state for the Markov chain, the HMC sampler integrates the Hamiltonian dynamics using leapfrog integration. The value of this property controls the number of steps of that leapfrog integration.

### **MassVector** — Mass vector of momentum variables

`ones(size(startpoint,1),1)` (default) | numeric column vector

Mass vector of momentum variables, specified as a numeric column vector with positive values and the same length as `startpoint`.

The “masses” of the momentum variables associated with the variables of interest control the Hamiltonian dynamics in each Markov chain proposal.

### **JitterMethod** — Method for jittering step size and number of steps

'jitter-both' (default) | 'jitter-numsteps' | 'none'

Method for jittering the step size and the number of steps, specified as one of the following values.

Value	Description
'jitter-both'	Randomly jitter the step size and number of steps of each leapfrog trajectory.
'jitter-numsteps'	Jitter only the number of steps of each leapfrog trajectory.

Value	Description
'none'	Perform no jittering.

With jittering, the sampler randomly selects the step size or the number of steps of each leapfrog trajectory as values smaller than the 'StepSize' and 'NumSteps' values. Use jittering to improve the stability of the leapfrog integration of the Hamiltonian dynamics.

#### **StepSizeTuningMethod** — Method for tuning sampler step size

'dual-averaging' (default) | 'none'

Method for tuning the sampler step size, specified as 'dual-averaging' or 'none'.

If `StepSizeTuningMethod` equals 'dual-averaging', then `tuneSampler` tunes the leapfrog step size of the HMC sampler to achieve a certain acceptance ratio for a fixed value of the simulation length. The simulation length equals the step size multiplied by the number of steps. To set the target acceptance ratio, use the 'TargetAcceptanceRatio' name-value pair argument of the `tuneSampler` method.

#### **MassVectorTuningMethod** — Method for tuning sampler mass vector

'iterative-sampling' (default) | 'hessian' | 'none'

Method for tuning the sampler mass vector, specified as one of the following values.

Value	Description
'iterative-sampling'	Tune the <code>MassVector</code> via successive approximations by drawing samples using a sequence of mass vector estimates.
'hessian'	Set the <code>MassVector</code> equal to the negative diagonal Hessian of the <code>logpdf</code> at the startpoint.
'none'	Perform no tuning of the <code>MassVector</code> .

To perform the tuning, use the `tuneSampler` method.

#### **LogPDF** — Logarithm of target density and its gradient

function handle

Logarithm of target density and its gradient, specified as a function handle.

`LogPDF` returns two output arguments: `[lpdf, glpdf] = LogPDF(X)`. Here, `lpdf` is the base-e log probability density (up to an additive constant) and `glpdf` is the gradient

of the log density at the point  $X$ . The input argument  $X$  must be a column vector with the same number of elements as the `StartPoint` property.

If you set the `'UseNumericalGradient'` value to `true` when creating the sampler, then `LogPDF` returns the numerical gradient in `glpdf`.

**StartPoint** — Initial point to start sampling from

numeric column vector

Initial point to start sampling from, specified as a numeric column vector.

**VariableNames** — Sampling variable names

`{'x1', 'x2', ...}` (default) | cell array of unique character vectors

Sampling variable names, specified as a cell array of unique character vectors.

## Methods

<code>estimateMAP</code>	Estimate maximum of log probability density
<code>tuneSampler</code>	Tune Hamiltonian Monte Carlo (HMC) sampler
<code>drawSamples</code>	Generate Markov chain using Hamiltonian Monte Carlo (HMC)
<code>diagnostics</code>	Markov Chain Monte Carlo diagnostics

## Examples

### Create Hamiltonian Monte Carlo Sampler

Create a Hamiltonian Monte Carlo (HMC) sampler to sample from a normal distribution.

First, save a function `normalDistGrad` on the MATLAB® path that returns the multivariate normal log probability density and its gradient (`normalDistGrad` is defined at the end of this example). Then, call the function with arguments to define the `logpdf` input argument to the `hmcSampler` function.

```
means = [1;-3];  
standevs = [1;2];  
logpdf = @(theta)normalDistGrad(theta,means,standevs);
```

Choose a starting point for the HMC sampler.

```
startpoint = randn(2,1);
```

Create the HMC sampler and display its properties.

```
smp = hmcSampler(logpdf, startpoint);
```

```
smp
```

```
smp =
```

```
HamiltonianSampler with properties:
```

```

        StepSize: 0.1000
        NumSteps: 50
        MassVector: [2x1 double]
        JitterMethod: 'jitter-both'
        StepSizeTuningMethod: 'dual-averaging'
        MassVectorTuningMethod: 'iterative-sampling'
        LogPDF: @(theta) normalDistGrad(theta, means, standevs)
        VariableNames: {2x1 cell}
        StartPoint: [2x1 double]
```

The `normalDistGrad` function returns the logarithm of the multivariate normal probability density with means in `Mu` and standard deviations in `Sigma`, specified as scalars or column vectors the same length as `startpoint`. The second output argument is the corresponding gradient.

```
function [lpdf, glpdf] = normalDistGrad(X, Mu, Sigma)
Z = (X - Mu)./Sigma;
lpdf = sum(-log(Sigma) - .5*log(2*pi) - .5*(Z.^2));
glpdf = -Z./Sigma;
end
```

- “Bayesian Linear Regression Using Hamiltonian Monte Carlo” on page 7-35

## See Also

### Functions

`hmcSampler` | `mhsample` | `slicesample`

## **Topics**

“Bayesian Linear Regression Using Hamiltonian Monte Carlo” on page 7-35

**Introduced in R2017a**

# estimateMAP

**Class:** HamiltonianSampler

Estimate maximum of log probability density

## Syntax

```
xhat = estimateMAP(smp)
[xhat, fitinfo] = estimateMAP(smp)
[xhat, fitinfo] = estimateMAP(____, Name, Value)
```

## Description

`xhat = estimateMAP(smp)` returns the maximum-a-posteriori (MAP) estimate of the log probability density of the Monte Carlo sampler `smp`.

`[xhat, fitinfo] = estimateMAP(smp)` returns additional fitting information in `fitinfo`.

`[xhat, fitinfo] = estimateMAP(____, Name, Value)` specifies additional options using one or more name-value pair arguments. Specify name-value pair arguments after all other input arguments.

## Input Arguments

**smp** — Hamiltonian Monte Carlo sampler

HamiltonianSampler object

Hamiltonian Monte Carlo sampler, specified as a `HamiltonianSampler` object.

`estimateMAP` estimates the maximum of the log probability density specified in `smp.LogPDF`.

Use the `hmcSampler` function to create a sampler.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'IterationLimit', 100, 'StepTolerance', 1e-5` estimates the MAP point using an iteration limit of 100 and a step size convergence tolerance of  $1e-5$ .

### **StartPoint** — Initial point to start optimization from

`smp.StartPoint` (default) | numeric column vector

Initial point to start optimization from, specified as a numeric column vector with the same number of elements as the `StartPoint` property of the sampler `smp`.

Example: `'StartPoint', randn(size(smp.StartPoint))`

### **IterationLimit** — Maximum number of optimization iterations

1000 (default) | positive integer

Maximum number of optimization iterations, specified as a positive integer.

Example: `'IterationLimit', 100`

### **VerbosityLevel** — Verbosity level of Command Window output

0 (default) | positive integer

Verbosity level of Command Window output during function maximization, specified as 0 or a positive integer.

- With the value set to 0, `estimateMAP` displays no details on the optimization.
- With the value set to a positive integer, `estimateMAP` displays convergence information at each iteration.



## Convergence Information

Heading	Meaning
FUN VALUE	Objective function value.
NORM GRAD	Norm of the gradient of the objective function.
NORM STEP	Norm of the iterative step, meaning the distance between the previous point and the current point.
CURV	OK means the weak Wolfe condition is satisfied. This condition is a combination of sufficient decrease of the objective function and a curvature condition.
GAMMA	Inner product of the step times the gradient difference, divided by the inner product of the gradient difference with itself. The gradient difference is the gradient at the current point minus the gradient at the previous point. Gives diagnostic information on the objective function curvature.
ALPHA	Step direction multiplier, which differs from 1 when the algorithm performed a line search.
ACCEPT	YES means the algorithm found an acceptable step to take.

Example: 'VerbosityLevel', 1

### **GradientTolerance** — Relative gradient convergence tolerance

1e-6 (default) | positive scalar

Relative gradient convergence tolerance, specified as a positive scalar.

Let  $\tau = \max(1, \min(\text{abs}(f), \text{infnormg0}))$ , where  $f$  is the current objective function value and  $\text{infnormg0}$  is the initial gradient norm. If the norm of the objective function gradient is smaller than  $\tau$  times the 'GradientTolerance' value, then the maximization is considered to have converged to a local optimum.

Example: 'GradientTolerance', 1e-4

### **StepTolerance** — Step size convergence tolerance

1e-6 (default) | positive scalar

Step size convergence tolerance, specified as a positive scalar.

If the proposed step size is smaller than the 'StepTolerance' value, then the maximization is considered to have converged to a local optimum.

Example: 'StepTolerance', 1e-5

## Output Arguments

### **xhat** — MAP point estimate

numeric vector

MAP point estimate, returned as a numeric vector of the same size as `smp.StartPoint`.

### **fitinfo** — Fitting information

structure

Fitting information for the MAP computation, returned as a structure with these fields:

Field	Description
Iteration	Iteration indices from 0 through the final iteration.
Objective	Negative log probability density at each iteration. The MAP point is computed by minimizing the negative log probability density. You can check that the final values are all similar, indicating that the function optimization has converged to a local optimum.
Gradient	Gradient of the negative log probability density at the final iteration.

Data Types: `struct`

## Examples

### Estimate MAP Point of HMC Sampler

Create a Hamiltonian Monte Carlo sampler for a normal distribution and estimate the maximum-a-posteriori (MAP) point of the log probability density.

First, save a function `normalDistGrad` on the MATLAB® path that returns the multivariate normal log probability density and its gradient (`normalDistGrad` is defined at the end of this example). Then, call the function with arguments to define the `logpdf` input argument to the `hmcSampler` function.

```
means = [1;-1];
standevs = [1;0.3];
logpdf = @(theta)normalDistGrad(theta,means,standevs);
```

Choose a starting point and create the HMC sampler.

```
startpoint = zeros(2,1);
smp = hmcSampler(logpdf,startpoint);
```

Estimate the MAP point (the point where the probability density has its maximum). Show more information during optimization by setting the `'VerbosityLevel'` value to 1.

```
[xhat,fitinfo] = estimateMAP(smp,'VerbosityLevel',1);
```

```
o Solver = LBFGS, HessianHistorySize = 15, LineSearchMethod = weakwolfe
```

```
=====
| ITER | FUN VALUE | NORM GRAD | NORM STEP | CURV | GAMMA | ALPH
=====
| 0 | 6.689460e+00 | 1.111e+01 | 0.000e+00 | | 9.000e-03 | 0.000
| 1 | 4.671622e+00 | 8.889e+00 | 2.008e-01 | OK | 9.006e-02 | 2.000
| 2 | 9.759850e-01 | 8.268e-01 | 8.215e-01 | OK | 9.027e-02 | 1.000
| 3 | 9.158025e-01 | 7.496e-01 | 7.748e-02 | OK | 5.910e-01 | 1.000
| 4 | 6.339508e-01 | 3.104e-02 | 7.472e-01 | OK | 9.796e-01 | 1.000
| 5 | 6.339043e-01 | 3.668e-05 | 3.762e-03 | OK | 9.599e-02 | 1.000
| 6 | 6.339043e-01 | 2.488e-08 | 3.333e-06 | OK | 9.015e-02 | 1.000
```

```
Infinity norm of the final gradient = 2.488e-08
```

```
Two norm of the final step = 3.333e-06, TolX = 1.000e-06
```

```
Relative infinity norm of the final gradient = 2.488e-08, TolFun = 1.000e-06
```

```
EXIT: Local minimum found.
```

To further check that the optimization has converged to a local minimum, plot the `fitinfo.Objective` field. This field contains the values of the negative log density at each iteration of the function optimization. The final values are all very similar, so the optimization has converged.

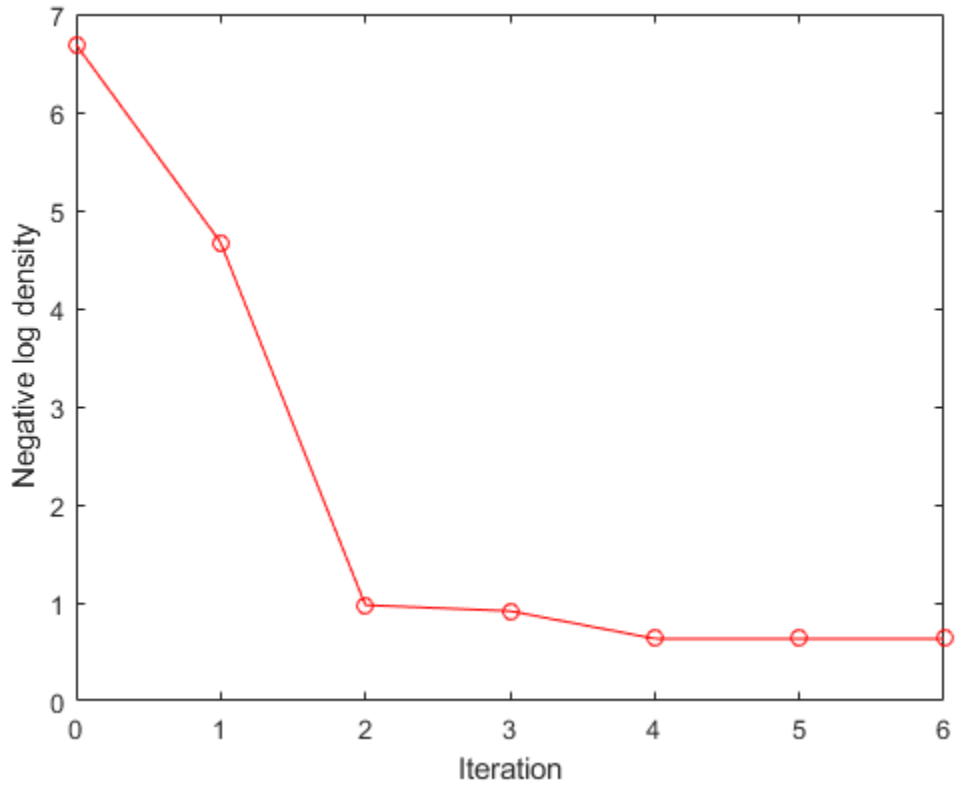
```
fitinfo

plot(fitinfo.Iteration, fitinfo.Objective, 'ro-');
xlabel('Iteration');
ylabel('Negative log density');

fitinfo =

    struct with fields:

        Iteration: [7x1 double]
        Objective: [7x1 double]
        Gradient: [2x1 double]
```



Display the MAP estimate. It is indeed equal to the means variable, which is the exact maximum.

```
xhat  
means
```

```
xhat =
```

```
  1.0000  
 -1.0000
```

```
means =
```

```
1  
-1
```

The `normalDistGrad` function returns the logarithm of the multivariate normal probability density with means in `Mu` and standard deviations in `Sigma`, specified as scalars or columns vectors the same length as `startpoint`. The second output argument is the corresponding gradient.

```
function [lpdf,glpdf] = normalDistGrad(X,Mu,Sigma)  
Z = (X - Mu)./Sigma;  
lpdf = sum(-log(Sigma) - .5*log(2*pi) - .5*(Z.^2));  
glpdf = -Z./Sigma;  
end
```

- “Bayesian Linear Regression Using Hamiltonian Monte Carlo” on page 7-35

## Tips

- First create a Hamiltonian Monte Carlo sampler using the `hmcSampler` function, and then use `estimateMAP` to estimate the MAP point.
- After creating an HMC sampler, you can tune the sampler, draw samples, and check convergence diagnostics using the other methods of the `HamiltonianSampler` class. Using the MAP estimate as a starting point in the `tuneSampler` and `drawSamples` methods can lead to more efficient tuning and sampling. For an example of this workflow, see Bayesian Linear Regression Using Hamiltonian Monte Carlo on page 7-35.

## Algorithms

- `estimateMAP` uses a limited memory Broyden-Fletcher-Goldfarb-Shanno (LBFGS) quasi-Newton optimizer to search for the maximum of the log probability density. See Nocedal and Wright [1].

## References

- [1] Nocedal, J. and S. J. Wright. *Numerical Optimization*, Second Edition. Springer Series in Operations Research, Springer Verlag, 2006.

## See Also

### Functions

`hmcSampler`

### Classes

`HamiltonianSampler`

## Topics

“Bayesian Linear Regression Using Hamiltonian Monte Carlo” on page 7-35

**Introduced in R2017a**

# tuneSampler

**Class:** HamiltonianSampler

Tune Hamiltonian Monte Carlo (HMC) sampler

## Syntax

```
tunedSmp = tuneSampler(smp)
[tunedSmp, tuningInfo] = tuneSampler(smp)
[tunedSmp, tuningInfo] = tuneSampler(____, Name, Value)
```

## Description

`tunedSmp = tuneSampler(smp)` returns a tuned Hamiltonian Monte Carlo (HMC) sampler.

First, `tuneSampler` tunes the mass vector of the HMC sampler `smp`. Then, it tunes the step size and number of steps of the leapfrog integrations to achieve a certain target acceptance ratio.

You can use the tuned sampler to create Markov chains using the `drawSamples` method.

`[tunedSmp, tuningInfo] = tuneSampler(smp)` returns additional tuning information in `tuningInfo`.

`[tunedSmp, tuningInfo] = tuneSampler(____, Name, Value)` specifies additional options using one or more name-value pair arguments. Specify name-value pair arguments after all other input arguments.

## Input Arguments

**smp** — Hamiltonian Monte Carlo sampler

HamiltonianSampler object

Hamiltonian Monte Carlo sampler to tune, specified as a `HamiltonianSampler` object.



Use the `hmcSampler` function to create a sampler.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'StepSizeTuningMethod', 'dual-averaging', 'MassVectorTuningMethod', 'hessian'` tunes an HMC sampler using the specified methods for tuning the step size and mass vector of the sampler.

### **StartPoint** — Initial point to start tuning from

`smp.StartPoint` (default) | numeric column vector

Initial point to start tuning from, specified as a numeric column vector with the same number of elements as the `StartPoint` property of the sampler `smp`.

Example: `'StartPoint', randn(size(smp.StartPoint))`

### **StepSizeTuningMethod** — Method for tuning sampler step size

`smp.StepSizeTuningMethod` (default) | 'dual-averaging' | 'none'

Method for tuning sampler step size, specified as the comma-separated pair consisting of 'StepSizeTuningMethod' and 'dual-averaging' or 'none'.

If 'StepSizeTuningMethod' is set to 'dual-averaging', then `tuneSampler` tunes the leapfrog step size of the HMC sampler to achieve a target acceptance ratio for a fixed value of the simulation length. The simulation length equals the step size multiplied by the number of steps. To specify the target acceptance ratio, set the 'TargetAcceptanceRatio' value.

To change the simulation length, set `smp.StepSize = a` and `smp.NumSteps = b`, for some values of `a` and `b`. This gives a simulation length of `a*b`.

Example: `'StepSizeTuningMethod', 'none'`

### **MassVectorTuningMethod** — Method for tuning sampler mass vector

`smp.MassVectorTuningMethod` (default) | 'iterative-sampling' | 'hessian' | 'none'

Method for tuning the sampler mass vector, specified as the comma-separated pair consisting of 'MassVectorTuningMethod' and one of the following values.

Value	Description
'iterative-sampling'	Tune the <code>MassVector</code> via successive approximations by drawing samples using a sequence of mass vector estimates.
'hessian'	Set the <code>MassVector</code> equal to the negative diagonal Hessian of the logpdf at the startpoint.
'none'	Perform no tuning of the <code>MassVector</code> .

Example: `'MassVectorTuningMethod', 'hessian'`

#### **NumStepSizeTuningIterations** — Number of step size tuning iterations

100 (default) | positive integer

Number of step size tuning iterations, specified as a positive integer.

If the 'StepSizeTuningMethod' value is 'none', then `tuneSampler` does not tune the step size.

Example: `'NumStepSizeTuningIterations', 50`

#### **TargetAcceptanceRatio** — Target acceptance ratio

0.65 (default) | scalar from 0 through 1

Target acceptance ratio of the Markov chain, specified as a scalar from 0 through 1.

`tuneSampler` tunes the step size and number of steps of the leapfrog integration to achieve the specified target acceptance ratio for a fixed value of the simulation length. The simulation length is the leapfrog integration step size multiplied by the number of integration steps.

If the 'StepSizeTuningMethod' value is 'none', then `tuneSampler` does not tune the step size.

To change the simulation length, set `smp.StepSize = a` and `smp.NumSteps = b`, for some values of `a` and `b`. This gives a simulation length of `a*b`.

Example: `'TargetAcceptanceRatio', 0.55`

**NumStepsLimit** — Maximum number of leapfrog steps

2000 (default) | positive integer

Maximum number of leapfrog steps allowed during step size tuning, specified as a positive integer.

If the 'StepSizeTuningMethod' value is 'none', then tuneSampler does not tune the step size.

Example: 'NumStepsLimit', 1000

**VerbosityLevel** — Verbosity level of Command Window output

0 (default) | nonnegative integer

Verbosity level of Command Window output during sampler tuning, specified as a nonnegative integer.

- With the value set to 0, tuneSampler displays no details of the tuning.
- With the value set to 1, tuneSampler displays details of the step size tuning.
- With the value set to 2 or larger, tuneSampler displays details of the step size and mass vector tuning.

Heading	Description
ITER	Iteration number.
LOG PDF	Log probability density at the current iteration.
STEP SIZE	Leapfrog integration step size at the current iteration.
NUM STEPS	Number of leapfrog integration steps at the current iteration.
ACC RATIO	Acceptance ratio, that is, the fraction of proposals which are accepted.

Heading	Description
DIVERGENT	Number of times the sampler failed to generate a valid proposal due to the leapfrog iterations generating NaNs or Infs. <code>tuneSampler</code> searches for a good value of the integration step size. While doing so, the algorithm can encounter regions of instability and report nonzero values in the <code>DIVERGENT</code> column. This behavior is normal and is not a problem in itself.

Example: `'VerbosityLevel', 1`

#### **NumPrint** — Verbose output frequency

100 (default) | positive integer

Verbose output frequency, specified as a positive integer.

If the `'VerbosityLevel'` value is a positive integer, `tuneSampler` outputs tuning details every `'NumPrint'` iterations.

Example: `'NumPrint', 50`

## Output Arguments

#### **tunedSmp** — Tuned Hamiltonian Monte Carlo sampler

`HamiltonianSampler` object

Tuned Hamiltonian Monte Carlo sampler, returned as a `HamiltonianSampler` object.

#### **tuningInfo** — Tuning information

structure

Tuning information, returned as a structure with these fields.

Field	Description
<code>MassVector</code>	Tuned mass vector
<code>StepSize</code>	Tuned leapfrog step size

Field	Description
NumSteps	Tuned value of the number of leapfrog integration steps
MassVectorTuningInfo	Structure with additional information on the mass vector tuning
StepSizeTuningInfo	Structure with additional information on the step size tuning

If you tune the mass vector using the 'iterative-sampling' method, then `MassVectorTuningInfo` has the following fields.

Field	Description
MassVector	Tuned mass vector
IterativeSamplingMassVectorProfile	$P$ -by- $K$ matrix of mass vectors used during the $K$ iterations, where $P$ is the number of sampling variables
IterativeSamplingNumSamples	$K$ -by-1 vector of the number of samples drawn for each of the $K$ iterations

If you tune the mass vector using the 'hessian' method, then `MassVectorTuningInfo` has the following fields.

Field	Description
MassVector	Tuned mass vector
NegativeDiagonalHessian	Negative diagonal Hessian of <code>logpdf</code> at the tuning start point. If some elements are negative, this field can be different from the <code>MassVector</code> field.
HessianPoint	Point at which the Hessian is evaluated

If the `MassVectorTuningMethod` value is 'none', then `MassVectorTuningInfo` is empty.

If you tune the step size using the 'dual-averaging' method, then `StepSizeTuningInfo` has the following fields.

Field	Description
StepSize	Tuned step size

Field	Description
NumSteps	Tuned value of the number of steps
StepSizeProfile	Column vector containing the step sizes at each tuning iteration
AcceptanceRatio	Final acceptance ratio achieved during tuning

If the step size is not tuned, then `StepSizeTuningInfo` is empty.

Data Types: `struct`

## Examples

### Tune Hamiltonian Monte Carlo Sampler

Tune the parameters of a Hamiltonian Monte Carlo (HMC) sampler.

Define the number of parameters to sample and their means.

```
NumParams = 9;
means = [1:NumParams]';
standevs = 1;
```

First, save a function `normalDistGrad` on the MATLAB® path that returns the multivariate normal log probability density and its gradient (`normalDistGrad` is defined at the end of this example). Then, call the function with arguments to define the `logpdf` input argument to the `hmcSampler` function.

```
logpdf = @(theta)normalDistGrad(theta,means,standevs);
```

Choose a starting point and create the HMC sampler.

```
startpoint = randn(NumParams,1);
smp = hmcSampler(logpdf,startpoint);
```

It is important to select good values for the sampler parameters to get efficient sampling. The best way to find good values is to automatically tune the `MassVector`, `StepSize`, and `NumSteps` parameters using `tuneSampler`. The method:

1. Tunes the `MassVector` of the sampler.

2. Tunes `StepSize` and `NumSteps` for a fixed simulation length to achieve a certain acceptance ratio. The default target acceptance ratio of 0.65 is good in most cases.

```
[smp,info] = tuneSampler(smp, 'NumStepSizeTuningIterations', 50, 'VerbosityLevel', 1, 'NumPr
```

```
o Tuning mass vector using method: iterative-sampling
Finished mass vector tuning iteration 1 of 5.
Finished mass vector tuning iteration 2 of 5.
Finished mass vector tuning iteration 3 of 5.
Finished mass vector tuning iteration 4 of 5.
Finished mass vector tuning iteration 5 of 5.
```

```
o Tuning step size using method: dual-averaging. Target acceptance ratio = 0.65
```

```
o Initial step size for dual-averaging = 2
```

```
=====
|  ITER  |  LOG PDF  |  STEP SIZE  |  NUM STEPS  |  ACC RATIO  |  DIVERGENT  |
|=====|
|    10  | -1.710457e+01 |  1.193e+00 |           4 |  5.000e-01 |           0 |
|    20  | -9.152514e+00 |  9.527e-01 |           5 |  5.500e-01 |           0 |
|    30  | -1.068923e+01 |  8.856e-01 |           6 |  5.333e-01 |           0 |
|    40  | -1.290816e+01 |  8.506e-01 |           6 |  5.750e-01 |           0 |
|    50  | -1.770386e+01 |  8.581e-01 |           6 |  6.000e-01 |           0 |
|=====|
```

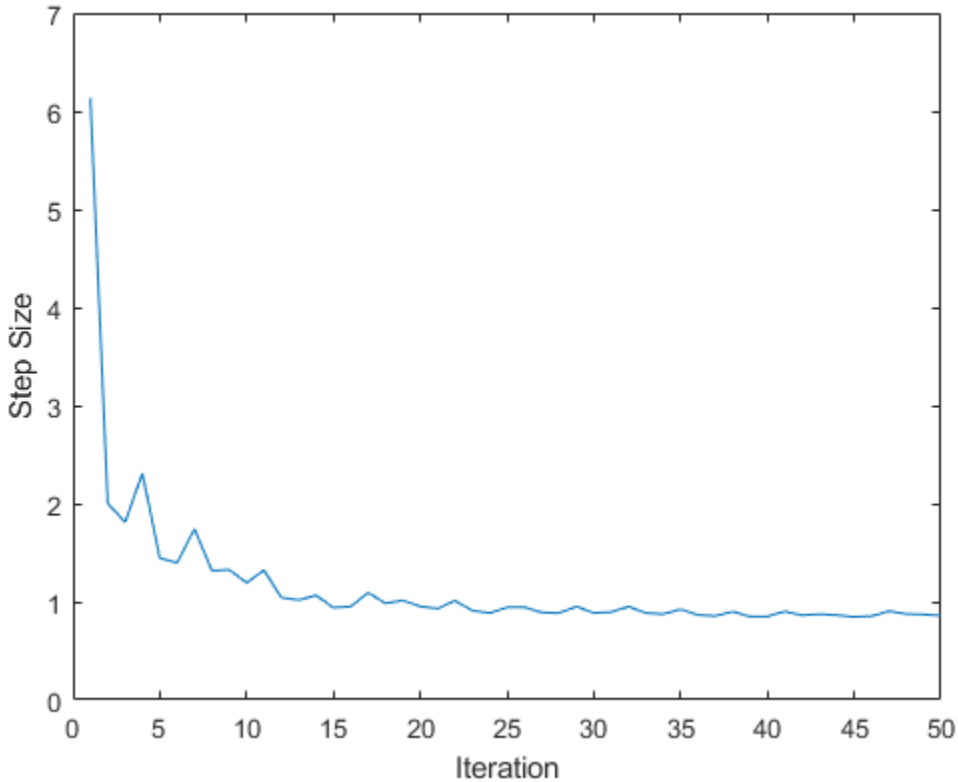
Plot the evolution of the step size during tuning to ensure that the step size tuning has converged. Display the achieved acceptance ratio.

```
figure;
plot(info.StepSizeTuningInfo.StepSizeProfile);
xlabel('Iteration');
ylabel('Step Size');

accratio = info.StepSizeTuningInfo.AcceptanceRatio

accratio =

    0.6000
```



The `normalDistGrad` function returns the logarithm of the multivariate normal probability density with means in `Mu` and standard deviations in `Sigma`, specified as scalars or columns vectors the same length as `startpoint`. The second output argument is the corresponding gradient.

```
function [lpdf,glpdf] = normalDistGrad(X,Mu,Sigma)
Z = (X - Mu)./Sigma;
lpdf = sum(-log(Sigma) - .5*log(2*pi) - .5*(Z.^2));
glpdf = -Z./Sigma;
end
```

- “Bayesian Linear Regression Using Hamiltonian Monte Carlo” on page 7-35



## Tips

- After creating an HMC sampler using the `hmcSampler` function, you can compute MAP (maximum-a-posteriori) point estimates, tune the sampler, draw samples, and check convergence diagnostics using the methods of the `HamiltonianSampler` class. For an example of this workflow, see [Bayesian Linear Regression Using Hamiltonian Monte Carlo](#) on page 7-35.

## See Also

### Functions

`hmcSampler`

### Classes

`HamiltonianSampler`

## Topics

“[Bayesian Linear Regression Using Hamiltonian Monte Carlo](#)” on page 7-35

**Introduced in R2017a**

## drawSamples

**Class:** HamiltonianSampler

Generate Markov chain using Hamiltonian Monte Carlo (HMC)

### Syntax

```
chain = drawSamples(smp)
[chain,endpoint,accratio] = drawSamples(smp)
[chain,endpoint,accratio] = drawSamples( ____,Name,Value)
```

### Description

`chain = drawSamples(smp)` generates a Markov chain by drawing samples using the Hamiltonian Monte Carlo sampler `smp`.

`[chain,endpoint,accratio] = drawSamples(smp)` also returns the final state of the Markov chain in `endpoint` and the fraction of accepted proposals in `accratio`.

`[chain,endpoint,accratio] = drawSamples( ____,Name,Value)` specifies additional options using one or more name-value pair arguments. Specify name-value pair arguments after all other input arguments.

### Input Arguments

**smp** — Hamiltonian Monte Carlo sampler

HamiltonianSampler object

Hamiltonian Monte Carlo sampler, specified as a HamiltonianSampler object.

`drawSamples` draws samples from the target log probability density in `smp.LogPDF`. Use the `hmcSampler` function to create a sampler.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'Burnin', 500, 'NumSamples', 2000` generates a Markov chain by discarding 500 burn-in samples and then drawing 2000 samples.

### **Burnin** — Number of burn-in samples to discard

1000 (default) | positive integer

Number of burn-in samples to discard from the beginning of the Markov chain, specified as a positive integer.

Example: `'Burnin', 500`

### **NumSamples** — Number of samples to draw

1000 (default) | positive integer

Number of samples to draw from the Markov chain using the HMC sampler, specified as a positive integer.

The `drawSamples` method generates this number of samples after the burn-in period.

Example: `'NumSamples', 2000`

### **ThinSize** — Markov chain thinning size

1 (default) | positive integer

Markov chain thinning size, specified as a positive integer.

Only one out of the `'ThinSize'` number of samples are kept. The rest of the samples are discarded.

Example: `'ThinSize', 5`

### **StartPoint** — Initial point to start sampling from

`smp.StartPoint` (default) | numeric column vector

Initial point to start sampling from, specified as a numeric column vector with the same number of elements as the `StartPoint` property of the sampler `smp`.

Example: `'StartPoint', randn(5,1)`

**verbosityLevel** — Verbosity level of Command Window output

0 (default) | positive integer

Verbosity level of Command Window output during sampling, specified as 0 or a positive integer.

With the value set to 0, `drawSamples` displays no details during sampling.

With the value set to a positive integer, `drawSamples` displays details of the sampling. To set the output frequency, use the `'NumPrint'` name-value pair argument.

`drawSamples` displays the output as a table with these columns.

Heading	Description
ITER	Iteration number
LOG PDF	Log probability density at the current iteration
STEP SIZE	Leapfrog integration step size at the current iteration. If the step size is jittered, it can vary between iterations.
NUM STEPS	Number of leapfrog integration steps at the current iteration. If the number of steps is jittered, it can vary between iterations
ACC RATIO	Acceptance ratio, that is, the fraction of proposals that are accepted. The acceptance ratio is calculated from the beginning of sampling, including the burn-in period.

Heading	Description
DIVERGENT	Number of times the sampler failed to generate a valid proposal due to the leapfrog iterations generating NaNs or Infs. When drawing samples, a nonzero value in the DIVERGENT column indicates that the chosen step size is above the stability threshold for some region of state space. To fix this issue, try to set the StepSize to a smaller value, draw new samples, and check that all values in the DIVERGENT column equal 0.

Example: `'VerbosityLevel', 1`

#### **NumPrint** — Verbose output frequency

100 (default) | positive integer

Verbose output frequency, specified as a positive integer.

If the `'VerbosityLevel'` value is a positive integer, then `drawSamples` outputs sampling details every `'NumPrint'` iterations.

Example: `'NumPrint', 200`

## Output Arguments

#### **chain** — Markov chain generated using Hamiltonian Monte Carlo

numeric matrix

Markov chain generated using Hamiltonian Monte Carlo, returned as a numeric matrix.

Each row of `chain` is a sample, and each column represents one sampling variable.

#### **endpoint** — Final state of Markov chain

numeric column vector

Final state of the Markov chain, returned as a numeric column vector of the same length as `smp.StartPoint`.

**accratio — Acceptance ratio**

numeric scalar

Acceptance ratio of the Markov chain proposals, returned as a numeric scalar. The acceptance ratio is calculated from the beginning of sampling, including the burn-in period.

## Examples

### Draw Samples Using HMC Sampler

Create MCMC chains for a multivariate normal distribution using a Hamiltonian Monte Carlo (HMC) sampler.

Define the number of parameters to sample and their means.

```
NumParams = 100;  
means = randn(NumParams,1);  
standevs = 0.1;
```

First, save a function `normalDistGrad` on the MATLAB® path that returns the multivariate normal log probability density and its gradient (`normalDistGrad` is defined at the end of this example). Then, call the function with arguments to define the `logpdf` input argument to the `hmcSampler` function.

```
logpdf = @(theta)normalDistGrad(theta,means,standevs);
```

Choose a starting point of the sampler. Create the HMC sampler and tune its parameters.

```
startpoint = randn(NumParams,1);  
smp = hmcSampler(logpdf,startpoint);  
smp = tuneSampler(smp);
```

Draw samples from the posterior density, using a few independent chains. Choose different, randomly distributed starting points for each chain. Specify the number of burn-in samples to discard from the beginning of the Markov chain and the number of samples to generate after the burn-in. Set the `'VerbosityLevel'` to print details during sampling for the first chain.

```

NumChains = 4;
chains     = cell(NumChains,1);
Burnin    = 500;
NumSamples = 2000;
for c = 1:NumChains
    if c == 1
        showOutput = 1;
    else
        showOutput = 0;
    end
    chains{c} = drawSamples(smp, 'Burnin', Burnin, 'NumSamples', NumSamples, ...
        'Start', randn(size(startpoint)), 'VerbosityLevel', showOutput, 'NumPrint', 500);
end

```

ITER	LOG PDF	STEP SIZE	NUM STEPS	ACC RATIO	DIVERGENT
500	8.017202e+01	5.303e-01	4	8.820e-01	0
1000	7.756258e+01	5.303e-01	8	8.500e-01	0
1500	9.227175e+01	5.303e-01	2	8.547e-01	0
2000	8.073805e+01	3.128e-02	6	8.610e-01	0
2500	9.672952e+01	5.161e-01	2	8.600e-01	0

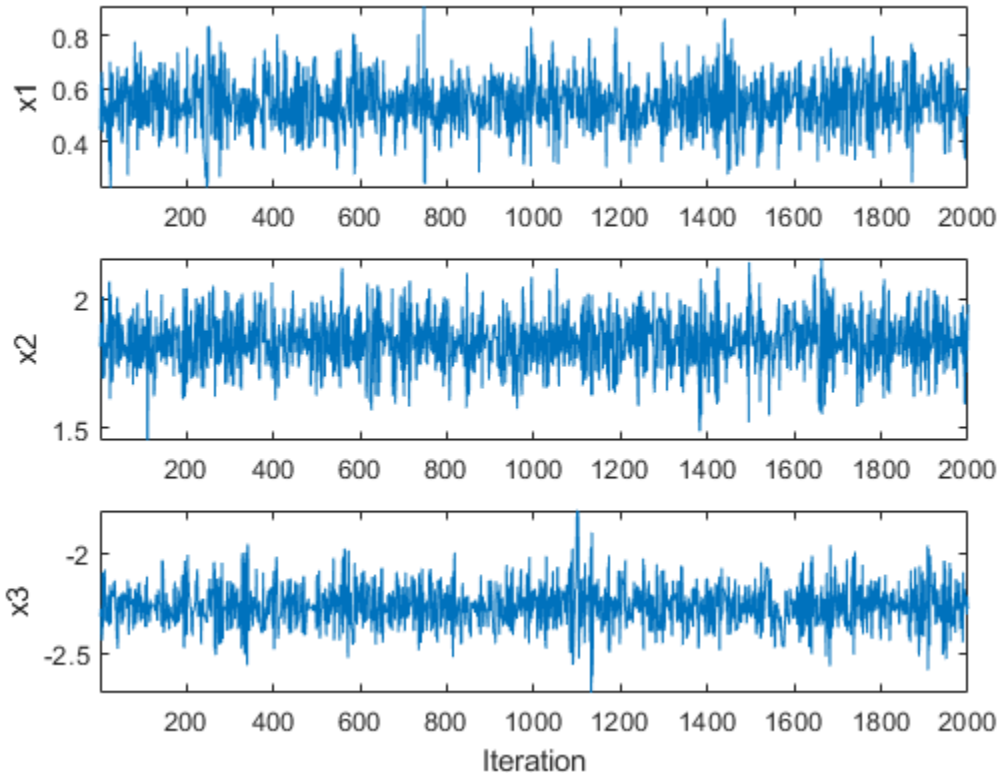
After obtaining a random sample, investigate issues such as convergence and mixing to determine whether the samples represent a reasonable set of random realizations from the target distribution. To examine the output, plot the trace plots of the samples for the first few variables using the first chain.

A number of burn-in samples have been removed to reduce the effect of the sampling starting point. Furthermore, the trace plots look like high-frequency noise, without any visible long-range correlation between the samples. This indicates that the chain is well mixed.

```

for p = 1:3
    subplot(3,1,p);
    plot(chains{1}(:,p));
    ylabel(smp.VariableNames(p))
    axis tight
end
xlabel('Iteration')

```



The `normalDistGrad` function returns the logarithm of the multivariate normal probability density with means in `Mu` and standard deviations in `Sigma`, specified as scalars or columns vectors the same length as the startpoint. The second output argument is the corresponding gradient.

```
function [lpdf,glpdf] = normalDistGrad(X,Mu,Sigma)
Z = (X - Mu)./Sigma;
lpdf = sum(-log(Sigma) - .5*log(2*pi) - .5*(Z.^2));
glpdf = -Z./Sigma;
end
```

- Bayesian Linear Regression Using Hamiltonian Monte Carlo on page 7-35



## Tips

- After creating an HMC sampler using the `hmcSampler` function, you can compute MAP (maximum-a-posteriori) point estimates, tune the sampler, draw samples, and check convergence diagnostics using the methods of the `HamiltonianSampler` class. For an example of this workflow, see [Bayesian Linear Regression Using Hamiltonian Monte Carlo](#) on page 7-35.

## See Also

### Functions

`hmcSampler`

### Classes

`HamiltonianSampler`

## Topics

[Bayesian Linear Regression Using Hamiltonian Monte Carlo](#) on page 7-35

**Introduced in R2017a**

## diagnostics

**Class:** HamiltonianSampler

Markov Chain Monte Carlo diagnostics

## Syntax

```
tbl = diagnostics(smp, chains)
tbl = diagnostics(smp, chains, 'MaxLag', maxlag)
```

## Description

`tbl = diagnostics(smp, chains)` returns Markov Chain Monte Carlo diagnostics for the chains in `chains`.

`tbl = diagnostics(smp, chains, 'MaxLag', maxlag)` specifies the maximum number of autocorrelation lags to use for computing effective sample sizes.

## Input Arguments

**smp** — Hamiltonian Monte Carlo sampler

HamiltonianSampler object

Hamiltonian Monte Carlo sampler, specified as a HamiltonianSampler object.

Use the `hmcSampler` function to create a sampler.

**chains** — MCMC chains

matrix | cell array

MCMC chains, specified as one of the following:

- A matrix, where each row is a sample and each column a parameter.
- A cell array of matrices, where the chain `chains{i}` is a matrix where each row is a sample and each column a parameter.

The number of parameters (that is, matrix columns) must equal the number of elements of the `StartPoint` property of the `smp` sampler.

### **maxlag** — Maximum number of autocorrelation lags

100 (default) | positive integer

Maximum number of autocorrelation lags for computing effective sample sizes, specified as a positive integer.

The effective sample size calculation uses lags of  $1, 2, \dots, \text{maxlag}$  for each chain in `chains` that has more than `maxlag` samples.

For chains with `maxlag` or fewer samples, the calculation uses  $N_i - 1$  lags, where  $N_i$  is the number of samples of chain  $i$ .

Example: `'MaxLag', 50`

## Output Arguments

### **tb1** — MCMC diagnostics

table

MCMC diagnostics, computed using all the chains in `chains` and returned as a table with these columns.

Column	Description
Name	Variable name
Mean	Posterior mean estimate
MCSE	Estimate of the Monte Carlo standard error (the standard deviation of the posterior mean estimate)
SD	Estimate of the posterior standard deviation
Q5	Estimate of the 5th quantile of the marginal posterior distribution
Q95	Estimate of the 95th quantile of the marginal posterior distribution

Column	Description
ESS	Effective sample size for the posterior mean estimate. Larger effective sample sizes lead to more accurate results. If the samples are independent, then the effective sample size is equal to the number of samples.
RHat	Gelman-Rubin convergence statistic. As a rule of thumb, values of RHat less than 1.1 are interpreted as a sign that the chains have converged to the target distribution. If RHat for any variable is larger than 1.1, try drawing more Monte Carlo samples.

## Examples

### Compute Markov Chain Monte Carlo Diagnostics

Create MCMC chains using a Hamiltonian Monte Carlo (HMC) sampler and compute MCMC diagnostics.

First, save a function on the MATLAB® path that returns the multivariate normal log probability density and its gradient. In this example, that function is called `normalDistGrad` and is defined at the end of the example. Then, call this function with arguments to define the `logpdf` input argument to the `hmcSampler` function.

```
means = [1;-2;2];
standevs = [1;2;0.5];
logpdf = @(theta)normalDistGrad(theta,means,standevs);
```

Choose a starting point. Create the HMC sampler and tune its parameters.

```
startpoint = randn(3,1);
smp = hmcSampler(logpdf,startpoint);
smp = tuneSampler(smp);
```

Draw samples from the posterior density, using a few independent chains. Choose different, randomly distributed starting points for each chain. Specify the number of

burn-in samples to discard from the beginning of the Markov chain and the number of samples to generate after the burn-in.

```

NumChains = 4;
chains = cell(NumChains,1);
Burnin = 500;
NumSamples = 1000;
for c = 1:NumChains
    chains{c} = drawSamples(smp, 'Burnin', Burnin, 'NumSamples', NumSamples, ...
        'Start', randn(size(startpoint)));
end

```

Compute MCMC diagnostics and display the results. Compare the true means in means with the column titled Mean in the MCMCdiagnostics table. The true posterior means are within a few Monte Carlo standard errors (MCSEs) of the estimated posterior means. The HMC sampler has accurately recovered the true means. Similarly, the estimated standard deviations in the column SD are very near the true standard deviations in standev.

```

MCMCdiagnostics = diagnostics(smp,chains)
means
standevs

```

```
MCMCdiagnostics =
```

```
3x8 table
```

Name	Mean	MCSE	SD	Q5	Q95	ESS	RHat
'x1'	1.0038	0.016474	0.96164	-0.58601	2.563	3407.4	1
'x2'	-2.0435	0.034933	1.999	-5.3476	1.1851	3274.5	1
'x3'	1.9957	0.008209	0.49693	1.2036	2.8249	3664.5	1

```
means =
```

```

1
-2
2

```

```
standevs =
```

```
1.0000
2.0000
0.5000
```

The `normalDistGrad` function returns the logarithm of the multivariate normal probability density with means in `Mu` and standard deviations in `Sigma`, specified as scalars or columns vectors the same length as the `startpoint`. The second output argument is the corresponding gradient.

```
function [lpdf,glpdf] = normalDistGrad(X,Mu,Sigma)
Z = (X - Mu)./Sigma;
lpdf = sum(-log(Sigma) - .5*log(2*pi) - .5*(Z.^2));
glpdf = -Z./Sigma;
end
```

- “Bayesian Linear Regression Using Hamiltonian Monte Carlo” on page 7-35

## Tips

- After creating an HMC sampler using the `hmcSampler` function, you can compute MAP (maximum-a-posteriori) point estimates, tune the sampler, draw samples, and check convergence diagnostics using the methods of the `HamiltonianSampler` class. For an example of this workflow, see Bayesian Linear Regression Using Hamiltonian Monte Carlo on page 7-35.

## See Also

### Functions

`hmcSampler`

### Classes

`HamiltonianSampler`

## Topics

- “Bayesian Linear Regression Using Hamiltonian Monte Carlo” on page 7-35

**Introduced in R2017a**

## Classification Learner

Train models to classify data using supervised machine learning

### Description

The **Classification Learner** app trains models to classify data. Using this app, you can explore supervised machine learning using various classifiers. You can explore your data, select features, specify validation schemes, train models, and assess results. You can perform automated training to search for the best classification model type, including decision trees, discriminant analysis, support vector machines, logistic regression, nearest neighbors, and ensemble classification.

You can perform supervised machine learning by supplying a known set of input data (observations or examples) and known responses to the data (e.g., labels or classes). You use the data to train a model that generates predictions for the response to new data. To use the model with new data, or to learn about programmatic classification, you can export the model to the workspace or generate MATLAB code to recreate the trained model.

---

**Tip** To get started, in the Classifier list, try **All Quick-To-Train** to train a selection of models. See “Automated Classifier Training” on page 23-12.

---

### Required Products

- MATLAB
- Statistics and Machine Learning Toolbox

### Open the Classification Learner App

- MATLAB Toolstrip: On the **Apps** tab, under **Math, Statistics and Optimization**, click the app icon.
- MATLAB command prompt: Enter `classificationLearner`.



## Examples

- “Train Classification Models in Classification Learner App” on page 23-11
- “Select Data and Validation for Classification Problem” on page 23-20
- “Automated Classifier Training” on page 23-12
- “Feature Selection and Feature Transformation Using Classification Learner App” on page 23-47
- “Choose Classifier Options” on page 23-26
- “Assess Classifier Performance in Classification Learner” on page 23-54
- “Export Classification Model to Predict New Data” on page 23-61

## See Also

### Apps

#### Regression Learner

### Functions

`fitcdiscr` | `fitcecoc` | `fitcensemble` | `fitcknn` | `fitcsvm` | `fitctree` | `fitglm`

## Topics

“Train Classification Models in Classification Learner App” on page 23-11

“Select Data and Validation for Classification Problem” on page 23-20

“Automated Classifier Training” on page 23-12

“Feature Selection and Feature Transformation Using Classification Learner App” on page 23-47

“Choose Classifier Options” on page 23-26

“Assess Classifier Performance in Classification Learner” on page 23-54

“Export Classification Model to Predict New Data” on page 23-61

### Introduced in R2015a

## Regression Learner

Train regression models to predict data using supervised machine learning

### Description

The **Regression Learner** app trains regression models to predict data. Using this app, you can explore your data, select features, specify validation schemes, train models, and assess results. You can perform automated training to search for the best regression model type, including linear regression models, regression trees, Gaussian process regression models, support vector machines, and ensembles of regression trees.

Perform supervised machine learning by supplying a known set of observations of input data (predictors) and known responses. Use the observations to train a model that generates predicted responses for new input data. To use the model with new data, or to learn about programmatic regression, you can export the model to the workspace or generate MATLAB code to recreate the trained model.

### Required Products

- MATLAB
- Statistics and Machine Learning Toolbox

### Open the Regression Learner App

- MATLAB Toolstrip: On the **Apps** tab, under **Math, Statistics and Optimization**, click the app icon.
- MATLAB command prompt: Enter `regressionLearner`.

### Examples

- Train Regression Models in Regression Learner App on page 24-2
- Select Data and Validation for Regression Problem on page 24-10

- [Automated Regression Model Training on page 24-3](#)
- [Choose Regression Model Options on page 24-15](#)
- [Feature Selection and Feature Transformation on page 24-32](#)
- [Assess Model Performance in Regression Learner on page 24-36](#)
- [Export Regression Model to Predict New Data on page 24-44](#)
- [Train Regression Trees Using Regression Learner App on page 24-48](#)

## See Also

### Apps

#### Classification Learner

### Functions

`fitlm` | `fitrensemble` | `fitrgp` | `fitrsvm` | `fitrtree` | `stepwiselm`

### Topics

[Train Regression Models in Regression Learner App on page 24-2](#)

[Select Data and Validation for Regression Problem on page 24-10](#)

[Automated Regression Model Training on page 24-3](#)

[Choose Regression Model Options on page 24-15](#)

[Feature Selection and Feature Transformation on page 24-32](#)

[Assess Model Performance in Regression Learner on page 24-36](#)

[Export Regression Model to Predict New Data on page 24-44](#)

[Train Regression Trees Using Regression Learner App on page 24-48](#)

### Introduced in R2017a

## Distribution Fitter

Fit probability distributions to data

### Description

The **Distribution Fitter** app interactively fits probability distributions to data imported from the MATLAB workspace. You can choose from 22 built-in probability distributions or create your own custom distribution. The app displays plots of the fitted distribution superimposed on a histogram of the data. Available plots include probability density function (pdf), cumulative distribution function (cdf), probability plots, and survivor functions. You can export the fitted parameter values to the workspace as a probability distribution object, and use object functions to perform further analyses. For more information on working with these objects, see “Working with Probability Distributions” on page 5-3. For the programmatic work flow of the Distribution Fitter app, see `distributionFitter`.

### Required Products

- MATLAB
- Statistics and Machine Learning Toolbox

### Open the Distribution Fitter App

- MATLAB Toolstrip: On the **Apps** tab, under **Math, Statistics and Optimization**, click the app icon.
- MATLAB command prompt: Enter `distributionFitter`.

### Examples

- “Fit a Distribution Using the Distribution Fitter App” on page 5-97

# Parameters

## Data

### Data — Data to import from the workspace

list of variables | valid MATLAB expression

Specify the data to import by selecting a variable from the drop-down list. If the variable is a matrix, the app imports the first column of the matrix by default. To select a different column or row of the matrix, click **Select Column or Row**. Alternatively, you can enter any valid MATLAB expression in the field.

### Censoring — Variable containing censoring data

list of variables

Specify the censoring data by selecting a variable from the drop-down list. If the variable is a matrix, the app imports the first column of the matrix by default. To select a different column or row of the matrix, click **Select Column or Row**. This parameter is optional.

### Frequency — Variable containing frequency data

list of variables

Specify the frequency data by selecting a variable from the drop-down list. If the variable is a matrix, the app imports the first column of the matrix by default. To select a different column or row of the matrix, click **Select Column or Row**. This parameter is optional.

### Data set name — Data set name

character vector

Specify a name for the data set as a character vector, or accept the default name.

### Manage data sets — Manage previously imported data sets

list of data sets

Manage previously imported data sets. Click the data set of interest, then click the buttons below this pane to view the data (**View**), set the bin rules (**Set Bin Rules**), rename the data set (**Rename**), or delete the data set (**Delete**).

### Data preview — Preview plot of data

histogram plot

Display a preview plot of the variable selected from the **Data** drop-down menu.

#### **New fit**

##### **Fit name — Name of fit**

character vector

Specify a name for the fit or accept the default name.

##### **Data — Data set to fit**

list of data sets

Specify the data to fit by selecting a data set from the drop-down list.

##### **Distribution — Distribution to fit**

Normal (default) | Exponential | Weibull | Non-parametric | ...

Specify the distribution to fit by selecting a distribution name from the drop-down list.

##### **Exclusion rule — Data exclusion rule**

list of exclusion rules

Specify a rule to exclude some data values by selecting an exclusion rule from the drop-down list. To populate this drop-down list, you must first define exclusion rules by clicking **Exclude** in the main window of the app. This parameter is optional

#### **Manage fits**

##### **Plot — Flag to plot fitted distribution**

checked (default) | unchecked

Specify which fit or fits to plot in the main window by selecting the **Plot** check box next to each fit. Clear the **Plot** check box to remove a fit from the plot.

##### **Conf bounds — Flag to plot confidence bounds**

unchecked (default) | checked

If you select **Plot** for a particular fit, you can select **Conf bounds** to display the confidence bounds for that fit on the plot in the main window. Clearing the **Conf bounds** check box removes the confidence intervals from the plot. The Distribution Fitter app displays confidence bounds only if the Display Type in the main window is set to **Cumulative probability (CDF)**, **Quantile (inverse CDF)**, **Survivor function**, or **Cumulative hazard**.

**Evaluate****Fit** — Fit to evaluate

list of fits

Select one or more fits from the list to evaluate.

**Function** — Available functions to fit

Density (PDF) (default) | Cumulative probability (CDF) | Quantile (inverse CDF) | Survivor function | Cumulative hazard | Hazard rate

Specify the type of probability function to evaluate from the drop-down list. Available probability functions include the probability density function (pdf), cumulative distribution function (cdf), quantile (inverse cdf), survival function, cumulative hazard, and hazard rate.

**At x =** — Values at which to evaluate function

numeric vector

Specify a numeric vector of values at which to evaluate the function. If you specify **Function** as **Quantile (inverse CDF)**, this field name changes to **At p =** and you enter a vector of probability values.

**Compute confidence bounds** — Flag to compute confidence bounds

unchecked (default) | checked

Select **Compute confidence bounds** to compute the confidence bounds for the selected fit. This check box is enabled only if you specify **Function** as **Cumulative probability (CDF)**, **Quantile (inverse CDF)**, **Survivor function**, or **Cumulative hazard**. This parameter is optional.

**Level** — Level for confidence bounds

95% (default) | numeric value

Specify the level at which to compute the confidence bounds. This check box is enabled only if you specify **Function** as **Cumulative probability (CDF)**, **Quantile (inverse CDF)**, **Survivor function**, or **Cumulative hazard**.

**Plot function** — Flag to plot function

unchecked (default) | checked

Select **Plot function** to display a plot of the distribution function, evaluated at the points that you enter in the **At  $x =$**  field, in a new window. This parameter is optional.

**Exclude****Exclusion rule name — Name of exclusion rule**

character vector

Specify a name for the exclusion rule as a character vector.

**Exclude sections — Define data exclusion rules numerically**

numeric value

Specify lower and upper limits for the data numerically.

**Exclude graphically — Define data exclusion rules graphically**

list of variables

Specify lower and upper limits for the data by selecting a variable from the **Select data** drop-down list and clicking **Exclude graphically**. An interactive plot opens in a new window, where you can add lower or upper limits by clicking and dragging a boundary on the plot.

**Existing exclusion rules — List of existing exclusion rules**

list of exclusion rules

Select an existing exclusion rule from the list. You can copy, view, rename, or delete exclusion rules by clicking the appropriate button.

## Programmatic Use

`distributionFitter` opens the Distribution Fitter app, or brings focus to the app if it is already open.

`distributionFitter(y)` opens the Distribution Fitter app populated with the data specified by the vector `y`.

`distributionFitter(y,cens)` uses the vector `cens` to specify whether the observation `y(j)` is censored, `(cens(j)==1)`, or observed exactly, `(cens(j)==0)`. If `cens` is omitted or empty, then no `y` values are censored.



`distributionFitter(y, cens, freq)` uses the vector `freq` to specify the frequency of each element contained in `y`. If `freq` is omitted or empty, then all values in `y` have a frequency of 1.

`distributionFitter(y, cens, freq, dsname)` creates a data set with the name `dsname` using the data vector, `y`, censoring indicator, `cens`, and frequency vector, `freq`. Specify `dsname` as a character vector, for example, `'mydata'`.

## See Also

### Functions

`distributionFitter` | `fitdist` | `makedist`

### Topics

“Fit a Distribution Using the Distribution Fitter App” on page 5-97

“Model Data Using the Distribution Fitter App” on page 5-70

“Working with Probability Distributions” on page 5-3

“Supported Distributions” on page 5-17

**Introduced before R2006a**

## plotPartialDependence

Create partial dependence plot (PDP) and individual conditional expectation (ICE) plots

### Syntax

```
plotPartialDependence(Mdl, Vars)
plotPartialDependence(Mdl, Vars, X)
plotPartialDependence( ____, Name, Value)
ax = plotPartialDependence( ____, Name, Value)
```

### Description

`plotPartialDependence(Mdl, Vars)` creates a PDP between features listed in `Vars` and responses predicted by using predictor data and a trained regression model in `Mdl`.

`plotPartialDependence(Mdl, Vars, X)` creates a PDP by using new predictor data in `X`.

`plotPartialDependence( ____, Name, Value)` uses additional options specified by one or more name-value pair arguments in addition to any of the arguments in the previous syntaxes. For example, if you specify `'Conditional'`, `'absolute'`, the `plotPartialDependence` function creates a figure including a PDP, a scatter plot of the selected feature and predicted responses, and an ICE plot for each observation.

`ax = plotPartialDependence( ____, Name, Value)` returns the axes of the plot using any of the input arguments combinations in the previous syntaxes.

### Examples

## Create Partial Dependence Plot

Train a regression tree using the `carsmall` data set, and create a PDP that shows the relationship between a feature and the predicted responses in the trained regression tree.

Load the `carsmall` data set.

```
load carsmall
```

Specify `Weight`, `Cylinders`, and `Horsepower` as the predictor variables ( $X$ ), and `MPG` as the response variable ( $Y$ ).

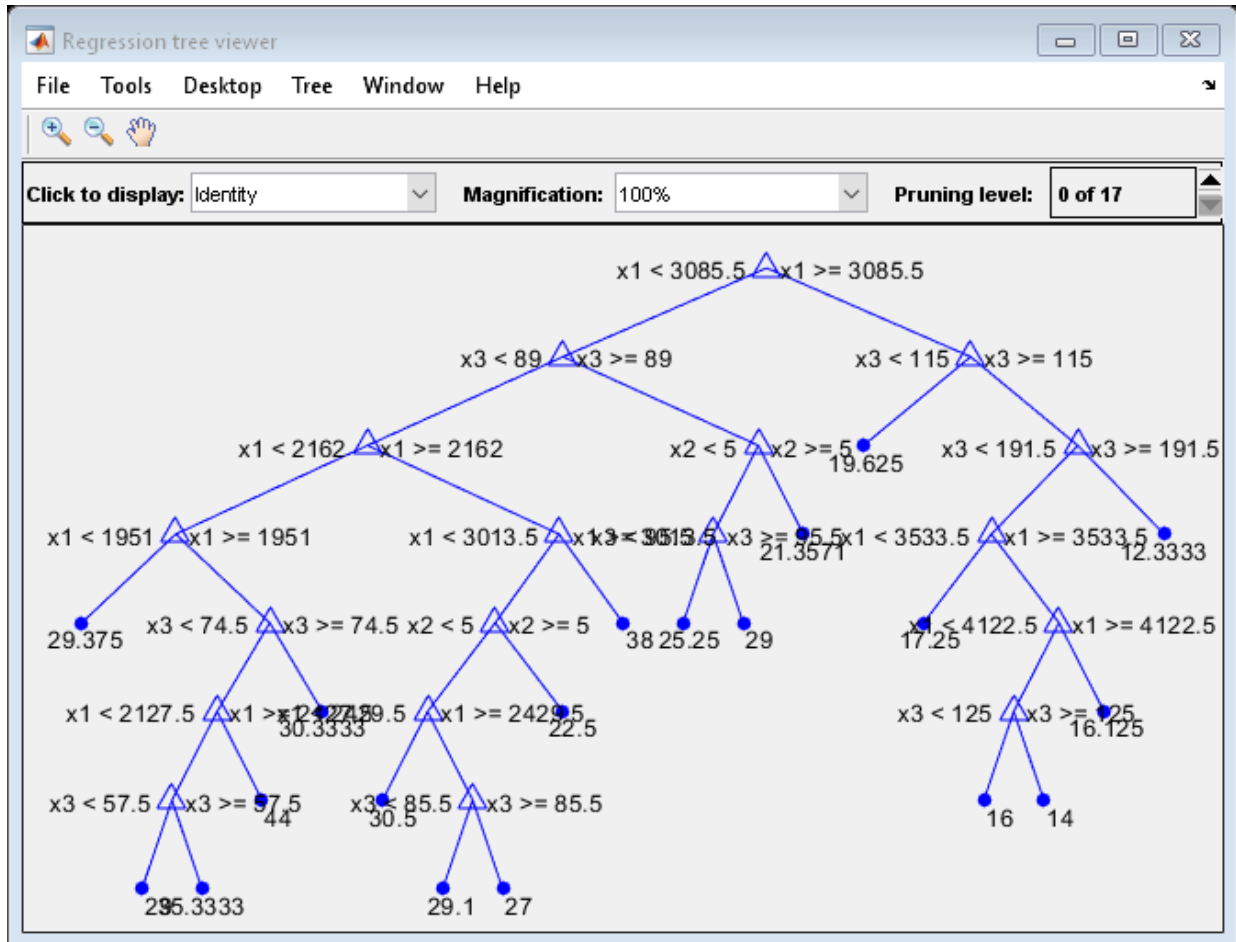
```
X = [Weight,Cylinders,Horsepower];  
Y = MPG;
```

Construct a regression tree using  $X$  and  $Y$ .

```
Mdl = fitrtree(X,Y);
```

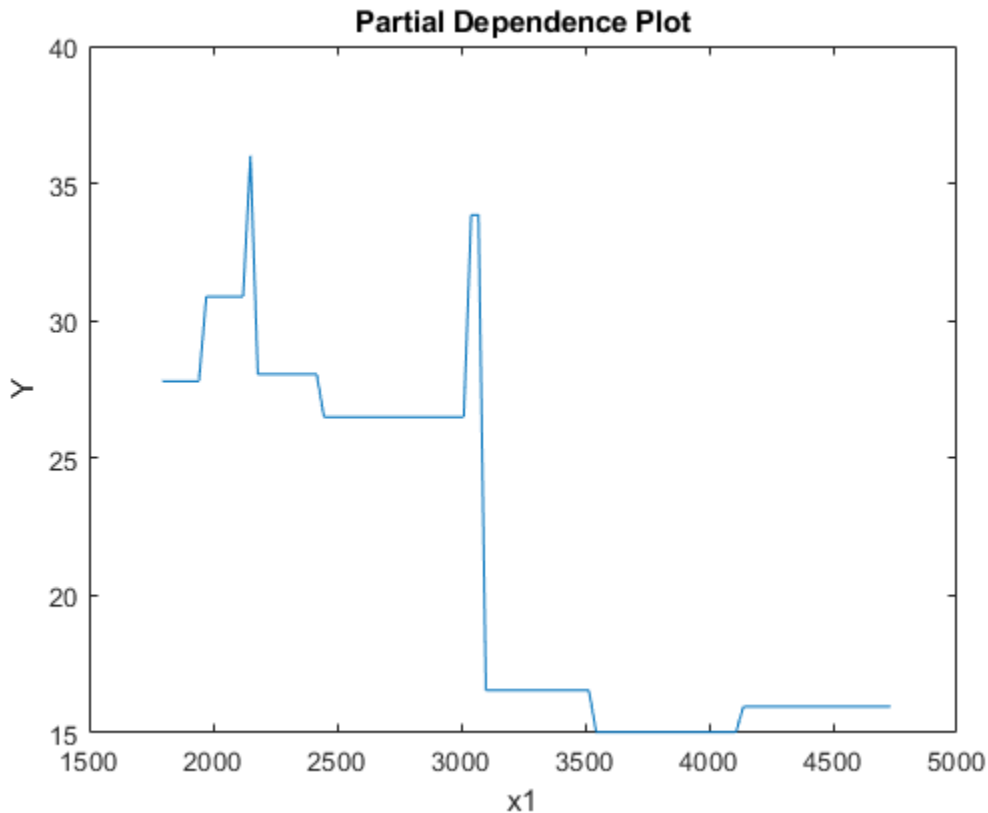
View a graphical display of the trained regression tree.

```
view(Mdl, 'Mode', 'graph')
```



Create a PDP of the first predictor variable, Weight.

`plotPartialDependence(Mdl, 1)`



The blue line represents averaged partial relationships between Weight (labeled as  $x_1$ ) and MPG (labeled as  $Y$ ) in the trained regression tree `Mdl`.

The regression tree viewer shows that the first decision is whether  $x_1$  is smaller than 3085.5. The PDP also shows a large change near  $x_1 = 3085.5$ . The tree viewer visualizes each decision at each node based on predictor variables. You can find several nodes split based on the values of  $x_1$ , but it is not easy to figure out the dependence of  $Y$  on  $x_1$ . However, the PDP plots averaged predicted responses against  $x_1$ , so you can clearly see the partial dependence of  $Y$  on  $x_1$ .

The labels  $x_1$  and  $Y$  are the default values of the predictor names and the response name. You can modify these names by specifying the name-value pair arguments

'PredictorNames' and 'ResponseName' when you train Mdl using `fitrtree`. You can also modify axis labels by using the `xlabel` and `ylabel` functions.

### Create Individual Conditional Expectation Plots

Train a Gaussian process regression model using generated sample data where a response variable includes interactions between predictor variables. Then, create ICE plots that show the relationship between a feature and the predicted responses for each observation.

Generate sample predictor data `x1` and `x2`.

```
rng('default') % For reproducibility
n = 200;
x1 = rand(n,1)*2-1;
x2 = rand(n,1)*2-1;
```

Generate response values that include interactions between `x1` and `x2`.

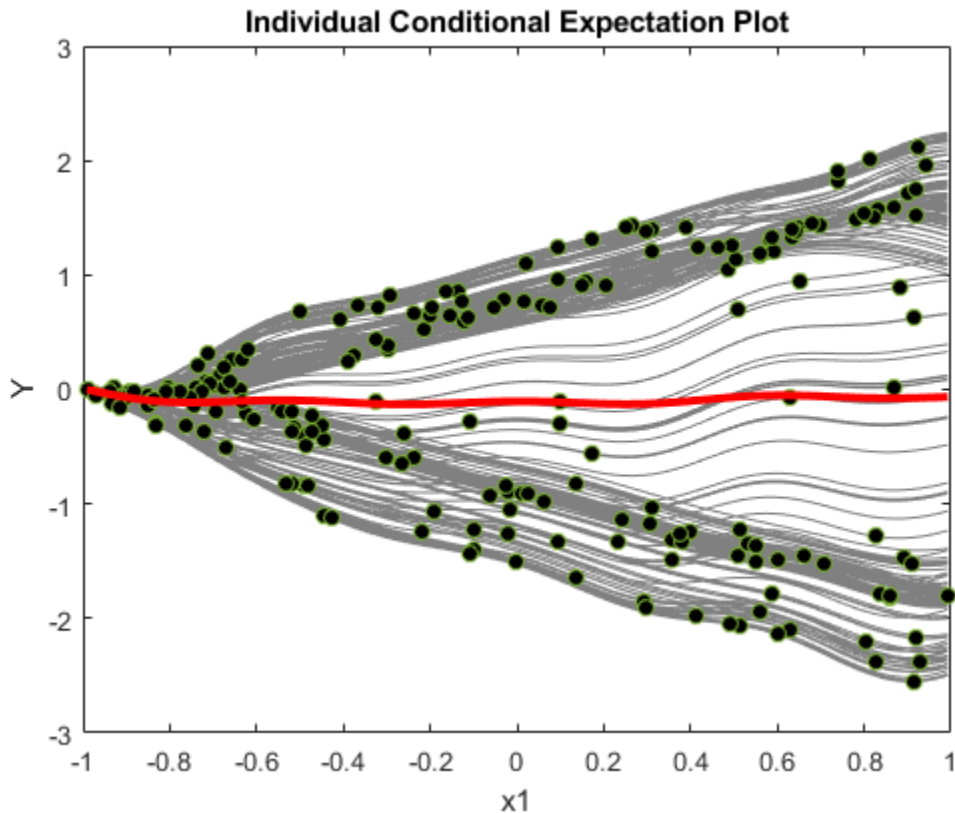
```
Y = x1-2*x1.*(x2>0)+0.1*rand(n,1);
```

Construct a Gaussian process regression model using `[x1 x2]` and `Y`.

```
Mdl = fitrgp([x1 x2],Y);
```

Create a figure including a PDP (red line) for the first predictor `x1`, a scatter plot (black circle markers) of `x1` and predicted responses, and a set of ICE plots (gray lines) by specifying 'Conditional' as 'centered'.

```
plotPartialDependence(Mdl,1,'Conditional','centered')
```



When 'Conditional' is 'centered', `plotPartialDependence` offsets plots so that all plots start from zero, which is helpful in examining the cumulative effect of the selected feature.

A PDP finds averaged relationships, so it does not reveal hidden dependencies especially when responses include interactions between features. However, the ICE plots clearly show two different dependencies of responses on `x1`.

### Compare Importance of Predictor Variables

Train a regression ensemble using the `carsmall` data set, and create a PDP plot and ICE plots for each predictor variable using a new data set, `carbig`. Then, compare the

figures to analyze the importance of predictor variables. Also, compare the results with the estimates of predictor importance returned by the `CompactRegressionEnsemble.predictorImportance` function.

Load the `carsmall` data set.

```
load carsmall
```

Specify `Weight`, `Cylinders`, `Horsepower`, and `Model_Year` as the predictor variables (X), and `MPG` as the response variable (Y).

```
X = [Weight,Cylinders,Horsepower,Model_Year];  
Y = MPG;
```

Train a regression ensemble using X and Y.

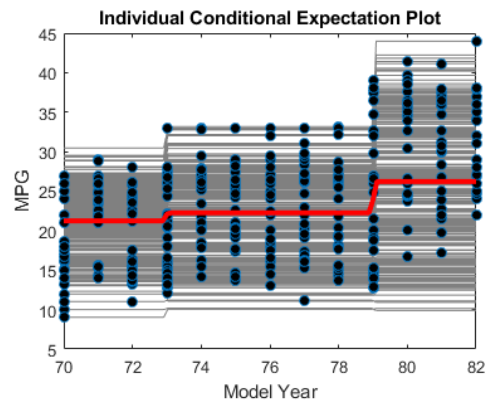
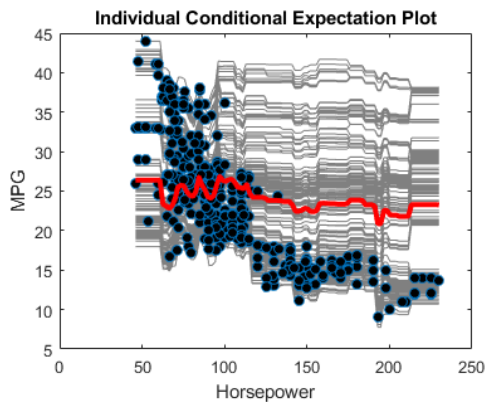
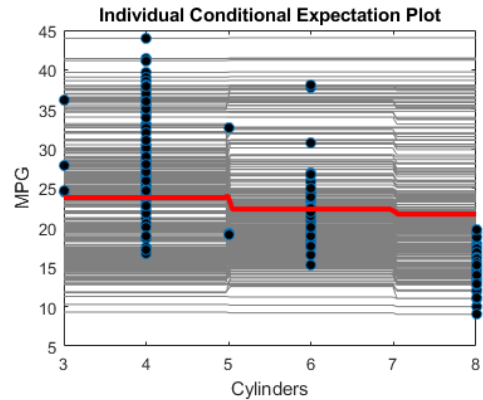
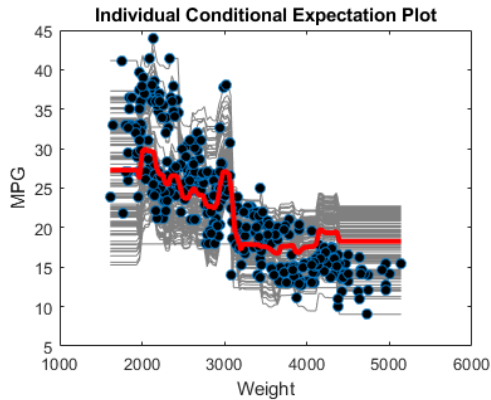
```
Mdl = fitensemble(X,Y, ...  
    'PredictorNames',{'Weight','Cylinders','Horsepower','Model_Year'}, ...  
    'ResponseName','MPG');
```

Compare the importance of predictor variables by using the `plotPartialDependence` and `CompactRegressionEnsemble.predictorImportance` functions. The `plotPartialDependence` function visualizes the relationships between a selected predictor and predicted responses. `predictorImportance` summarizes the importance of a predictor with a single value.

Create a figure including a PDP plot (red line) and ICE plots (gray lines) for each predictor by using `plotPartialDependence` and specifying `'Conditional','absolute'`. Each figure also includes a scatter plot (black circle markers) of the selected predictor and predicted responses. Also, load the `carbig` data set and use it as new predictor data, `Xnew`. When you provide `Xnew`, the `plotPartialDependence` function uses `Xnew` instead of the predictor data in `Mdl`.

```
load carbig  
Xnew = [Weight,Cylinders,Horsepower,Model_Year];  
  
f = figure;  
f.Position = [100 100 1.75*f.Position(3:4)]; % Enlarge figure for visibility.  
for i = 1 : 4  
    subplot(2,2,i)  
    plotPartialDependence(Mdl,i,Xnew,'Conditional','absolute')  
end
```

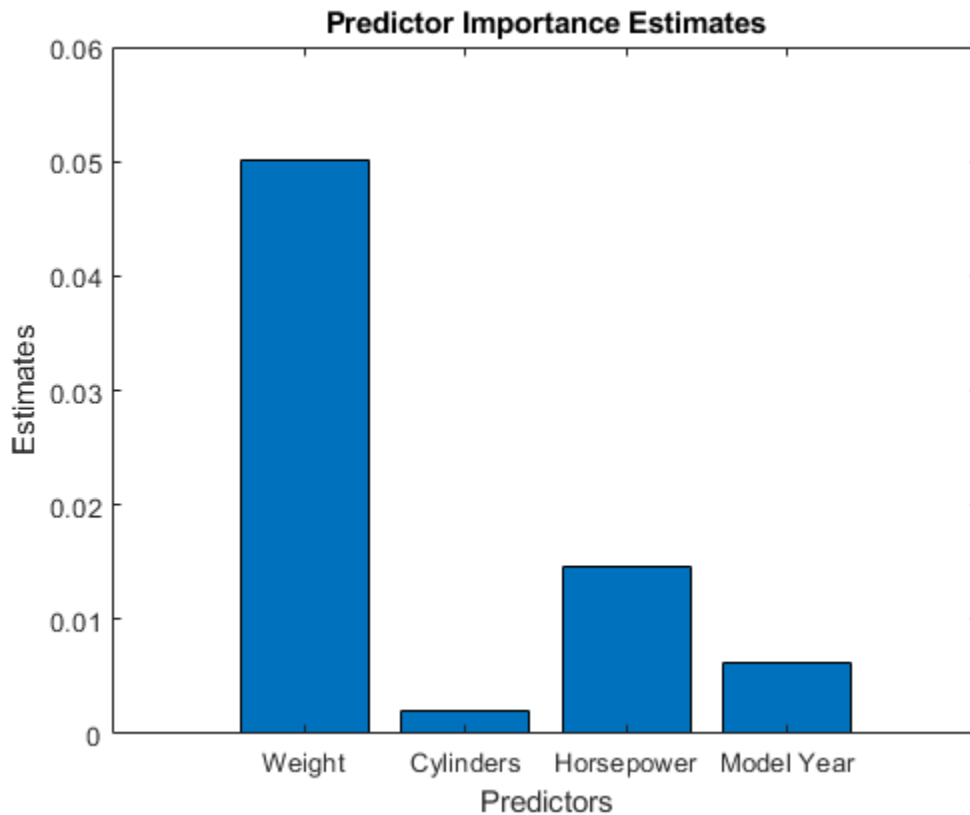




Compute estimates of predictor importance by using `predictorImportance`. This function sums changes in the mean-squared error (MSE) due to splits on every predictor, and then divides the sum by the number of branch nodes.

```
imp = predictorImportance(Mdl);
figure;
bar(imp);
title('Predictor Importance Estimates');
ylabel('Estimates');
xlabel('Predictors');
```

```
ax = gca;  
ax.XTickLabel = Mdl.PredictorNames;
```



The variable `Weight` has the most impact on MPG according to predictor importance. The PDP of `Weight` also shows that MPG has high partial dependence on `Weight`. The variable `Cylinders` has the least impact on MPG according to predictor importance. The PDP of `Cylinders` also shows that MPG does not change much depending on `Cylinders`.

## Extract Partial Dependence Estimates from Plots

Train a support vector machine (SVM) regression model using the `carsmall` data set, and create a PDP for two predictor variables. Then, extract partial dependence estimates from the output of `plotPartialDependence`.

Load the `carsmall` data set.

```
load carsmall
```

Specify `Weight`, `Cylinders`, and `Horsepower` as the predictor variables (`Tbl`).

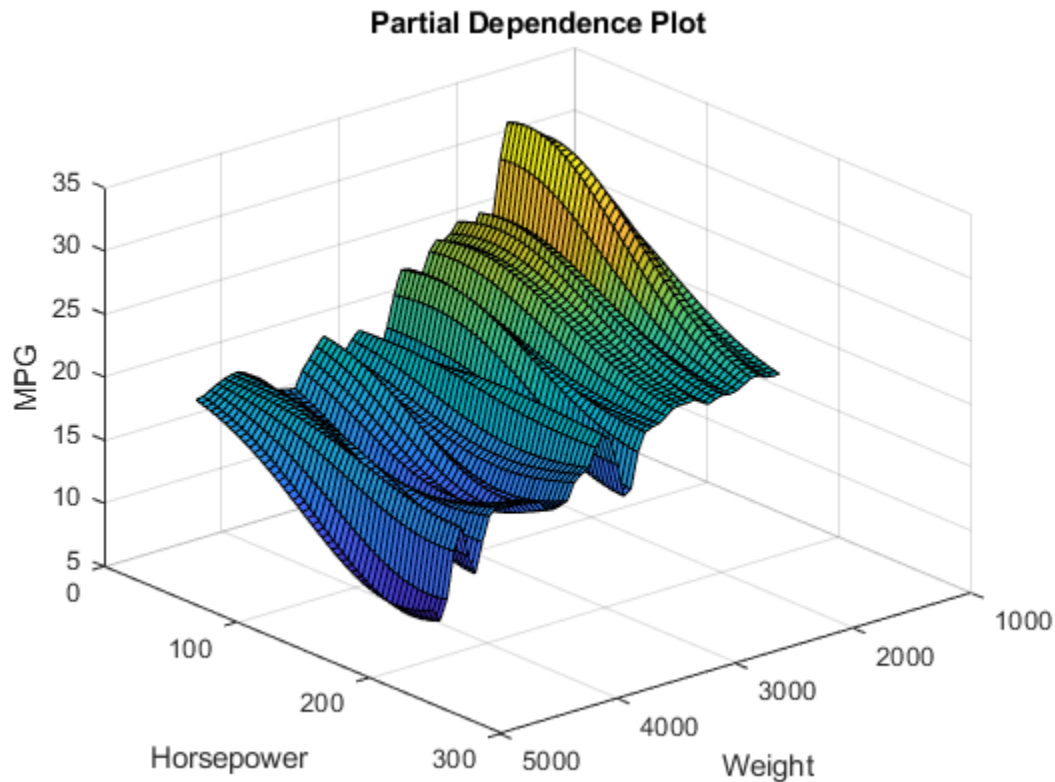
```
Tbl = table(Weight,Cylinders,Horsepower);
```

Construct a support vector machine (SVM) regression model using `Tbl` and the response variable `MPG`. Use a Gaussian kernel function with an automatic kernel scale.

```
Mdl = fitrsvm(Tbl,MPG,'ResponseName','MPG', ...  
             'KernelFunction','gaussian','KernelScale','auto');
```

Create a PDP that visualizes partial dependence of predicted responses (`MPG`) on the predictor variables `Weight` and `Horsepower`. Use the `QueryPoints` name-value pair argument to specify the points to compute partial dependence.

```
pt1 = linspace(min(Weight),max(Weight),50)';  
pt2 = linspace(min(Horsepower),max(Horsepower),50)';  
ax = plotPartialDependence(Mdl,{'Weight','Horsepower'},'QueryPoints',[pt1 pt2]);  
view(140,30) % Modify the viewing angle
```



The PDP shows an interaction effect between `Weight` and `Horsepower`.

Extract the estimated partial dependence of `MPG` on `Weight` and `Horsepower`. The `XData`, `YData`, and `ZData` values of `ax.Children` are x-axis values (the first selected predictor values), y-axis values (the second selected predictor values), and z-axis values (the corresponding partial dependence values), respectively.

```
xval = ax.Children.XData;
yval = ax.Children.YData;
zval = ax.Children.ZData;
```

If you specify `'Conditional'` as `'absolute'`, `plotPartialDependence` creates a figure including a PDP, a scatter plot, and a set of ICE plots. `ax.Children(1)` and

`ax.Children(2)` correspond to the PDP and scatter plot, respectively. The remaining elements of `ax.Children` correspond to the ICE plots. The `XData` and `YData` values of `ax.Children(i)` are x-axis values (the selected predictor values) and y-axis values (the corresponding partial dependence values), respectively.

## Input Arguments

### `mdl` — Trained regression model

full regression model object | compact regression model object

Trained regression model, specified as a full or compact regression model object described in the following table.

Trained Model Type	Regression Model Object	Returned By
Bootstrap aggregation for ensemble of decision trees	<code>TreeBagger</code> , <code>CompactTreeBagger</code>	<code>TreeBagger</code> , <code>compact</code>
Ensemble of regression models	<code>RegressionEnsemble</code> , <code>RegressionBaggedEnsemble</code> , <code>CompactRegressionEnsemble</code>	<code>fitrensemble</code> , <code>compact</code>
Gaussian process regression	<code>RegressionGP</code> , <code>CompactRegressionGP</code>	<code>fitrgp</code> , <code>compact</code>
Generalized linear mixed-effect model	<code>GeneralizedLinearMixedModel</code>	<code>fitglme</code>
Generalized linear model	<code>GeneralizedLinearModel</code> , <code>CompactGeneralizedLinearModel</code>	<code>fitglm</code> , <code>stepwiseglm</code> , <code>compact</code>
Linear mixed-effect model	<code>LinearMixedModel</code>	<code>fitlme</code> , <code>fitlmematrix</code>
Linear regression	<code>LinearModel</code> , <code>CompactLinearModel</code>	<code>fitlm</code> , <code>stepwiselm</code> , <code>compact</code>
Linear regression for high-dimensional data	<code>RegressionLinear</code>	<code>fitrlinear</code>

Trained Model Type	Regression Model Object	Returned By
Nonlinear regression	NonLinearModel	fitnlm
Regression tree	RegressionTree, CompactRegressionTree	fitrtree, compact
Support vector machine regression	RegressionSVM, CompactRegressionSVM	fitrsvm, compact

If `Mdl` is a compact regression model object, you must provide `X`.

`plotPartialDependence` does not support a trained model object with a sparse matrix. If you train `Mdl` by using `fitrlinear`, use a full numeric matrix for predictor data where rows correspond to individual observations.

### vars — Features to visualize

vector of positive integers | character vector | cell array of character vectors

Features to visualize, specified as a vector of positive integers, a character vector, or a cell array of character vectors. You can choose one or two features, as shown in the following tables.

#### One Feature

Value	Description
positive integer	Index value corresponding to the column of the predictor data to visualize.
character vector	Name of a predictor variable to visualize. The name must match the entry in <code>Mdl.PredictorNames</code> .

#### Two Features

Value	Description
vector of two positive integers	Index values corresponding to the columns of the predictor data to visualize.
cell array of character vectors	Names of the predictor variables to visualize. Each element in the array is the name of a predictor variable. The names must match the entries in <code>Mdl.PredictorNames</code> .

Example: `{'x1', 'x3'}`

Data Types: `single` | `double` | `char` | `cell`

**x — Predictor data**

numeric matrix | table

Predictor data, specified as a numeric matrix or table. Each row of `X` corresponds to one observation, and each column corresponds to one variable.

- If `Mdl` is a full regression model object, `plotPartialDependence` uses the predictor data in `Mdl`. If you provide `X`, then `plotPartialDependence` does not use the predictor data in `Mdl` and uses `X` only.
- If `Mdl` is a compact regression model object, you must provide `X`.

`X` must be consistent with the predictor data that trained `Mdl`, stored in either `Mdl.X` or `Mdl.Variables`.

- If you trained `Mdl` using a numeric matrix, then `X` must be a numeric matrix. The variables making up the columns of `X` must have the same number and order as the predictor variables that trained `Mdl`.
- If you trained `Mdl` using a table (for example, `Tbl`), then `X` must be a table. All predictor variables in `X` must have the same variable names and data types as the names and types in `Tbl`. However, the column order of `X` does not need to correspond to the column order of `Tbl`.
- `plotPartialDependence` does not support a sparse matrix.

Data Types: `single` | `double` | `table`

**Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `plotPartialDependence(Mdl, Vars, X, 'NumObservationsToSample', 100, 'UseParallel', true)` creates a PDP by using 100 sampled observations in `X` and executing for-loop iterations in parallel.

**Conditional — Plot type**

'none' (default) | 'absolute' | 'centered'

Plot type, specified as the comma-separated pair consisting of 'Conditional' and 'none', 'absolute', or 'centered'.

Value	Description
'none'	<p><code>plotPartialDependence</code> creates a PDP.</p> <ul style="list-style-type: none"> <li>• If you select one numeric feature in <code>Vars</code>, <code>plotPartialDependence</code> computes partial dependence at values between the minimum and maximum of the selected feature and creates a PDP with a solid blue line.</li> <li>• If you select one categorical feature in <code>Vars</code>, <code>plotPartialDependence</code> computes predicted responses at all categorical values and plots partial dependence with blue circle markers connected with a dotted line.</li> <li>• If you select two features in <code>Vars</code>, <code>plotPartialDependence</code> creates a surface plot.</li> </ul>
'absolute'	<p><code>plotPartialDependence</code> creates a figure including the following three types of plots:</p> <ul style="list-style-type: none"> <li>• PDP with a solid red line</li> <li>• Scatter plot of the selected feature and predicted responses with black circle markers</li> <li>• ICE plot for each observation with a solid gray line</li> </ul> <p>This value is valid when you select only one feature.</p>
'centered'	<p><code>plotPartialDependence</code> creates a figure including the same three types of plots as 'absolute'. <code>plotPartialDependence</code> offsets plots so that all plots start from zero.</p> <p>This value is valid when you select only one feature.</p>

For details, see “Partial Dependence Plot” on page 32-7068 and “Individual Conditional Expectation Plots” on page 32-7069.

Example: 'Conditional', 'absolute'

#### **NumObservationsToSample** — Number of observations to sample

number of total observations (default) | positive integer

Number of observations to sample, specified as the comma-separated pair consisting of 'NumObservationsToSample' and a positive integer. The default value is the number



of total observations in either `Mdl` or `X`. If you specify a value larger than the number of total observations, then `plotPartialDependence` uses all observations.

`plotPartialDependence` samples observations without replacement by using the function `datasample` and uses the sampled observations to compute partial dependence. If you specify `'Conditional'` as either `'absolute'` or `'centered'`, `plotPartialDependence` creates a figure including an ICE plot for each sampled observation.

Example: `'NumObservationsToSample', 100`

Data Types: `single` | `double`

#### **ParentAxisHandle — Axes in which to plot**

`gca` (default) | axes object

Axes in which to plot, specified as the comma-separated pair consisting of `'ParentAxisHandle'` and an axes object. If you do not specify the axes and if the current axes are Cartesian, then `plotPartialDependence` uses the current axes (`gca`). If axes do not exist, `plotPartialDependence` plots in a new figure.

Example: `'ParentAxisHandle', ax`

#### **QueryPoints — Points to compute partial dependence**

numeric column vector | numeric two column matrix | cell array of two numeric column vectors

Points to compute partial dependence, specified as the comma-separated pair consisting of `'QueryPoints'` and a numeric column vector, a numeric two-column matrix, or a cell array of two numeric column vectors.

The default value is a numeric column vector or a numeric two-column matrix, depending on the number of selected features, where a column contains 100 evenly spaced points between the minimum and maximum values of the selected feature.

- If you select one feature in `Vars`, use a numeric column vector.
- If you select two features in `Vars`:
  - Use a numeric two-column matrix to specify the same number of points for each feature.
  - Use a cell array of two numeric column vectors to specify a different number of points for each feature.

The default values for a categorical feature are all categorical values in the selected feature. You cannot modify 'QueryPoints' for a categorical feature. If you select one numeric feature and one categorical feature, you can specify 'QueryPoints' for a numeric feature by using a cell array consisting of a numeric column vector and an empty array.

Example: `'QueryPoints', {pt, []}`

Data Types: `single | double | cell`

#### **UseParallel** — Flag to run in parallel

`false` (default) | `true`

Flag to run in parallel, specified as the comma-separated pair consisting of 'UseParallel' and `true` or `false`. If you specify 'UseParallel' as `true`, `plotPartialDependence` executes for-loop iterations in parallel by using `parfor` when predicting responses for each observation and averaging them.

Example: `'UseParallel', true`

Data Types: `logical`

## Output Arguments

### **ax** — Axes of the plot

axes object

Axes of the plot, returned as an axes object. For details on how to modify the appearance of the axes and extract data from plots, see “Axes Appearance” (MATLAB) and “Extract Partial Dependence Estimates from Plots” on page 32-7060.

## Definitions

### Partial Dependence Plot

A partial dependence plot[1] (PDP) visualizes relationships between features and predicted responses in a trained regression model. `plotPartialDependence` creates either a line plot or a surface plot of predicted responses against a single feature or a pair of features, respectively, by marginalizing over the other variables.

Consider a PDP for a subset  $X^S$  of the whole feature set  $X = \{x_1, x_2, \dots, x_m\}$ . A subset  $X^S$  includes either one feature or two features:  $X^S = \{x_{S1}\}$  or  $X^S = \{x_{S1}, x_{S2}\}$ . Let  $X^C$  be the complementary set of  $X^S$  in  $X$ . A predicted response  $f(X)$  depends on all features in  $X$ :  $f(X) = f(X^S, X^C)$ .

The partial dependence of predicted responses on  $X^S$  is defined by the expectation of predicted responses with respect to  $X^C$ :

$$f^S(X^S) = E_C [f(X^S, X^C)] = \int f(X^S, X^C) p_C(X^C) dX^C,$$

where  $p_C(X^C)$  is the marginal probability of  $X^C$ , that is,  $p_C(X^C) = \int p(X^S, X^C) dX^S$ . Assuming that each observation is equally likely, and the dependence between  $X^S$  and  $X^C$  and the interactions of  $X^S$  and  $X^C$  in responses are not strong, `plotPartialDependence` estimates the partial dependence by using observed predictor data as follows:

$$f^S(X^S) \approx \frac{1}{N} \sum_{i=1}^N f(X^S, X_i^C),$$

where  $N$  is the number of observations and  $X_i = (X_i^S, X_i^C)$  is the  $i$ th observation.

`plotPartialDependence` creates a PDP by using “Equation 32-1” on page 32-7069. Input a trained model ( $f(\cdot)$ ) and select features ( $X^S$ ) to visualize by using the input arguments `Mdl` and `Vars`, respectively. `plotPartialDependence` computes partial dependence at 100 evenly spaced points of  $X^S$  or the points that you specify by using the `'QueryPoints'` name-value pair argument. You can specify the number ( $N$ ) of observations to sample from given predictor data by using the `'NumObservationsToSample'` name-value pair argument.

## Individual Conditional Expectation Plots

An individual conditional expectation (ICE) plot[2], as an extension of a PDP, visualizes the relationship between a feature and the predicted responses for each observation. While a PDP visualizes the averaged relationship between features and predicted responses, a set of ICE plots disaggregates the averaged information and visualizes an individual dependence for each observation.

`plotPartialDependence` creates an ICE plot for each observation. A set of ICE plots is useful to investigate heterogeneities of partial dependence originating from different

observations. `plotPartialDependence` can also create ICE plots with any predictor data provided through the input argument `X`. You can use this feature to explore predicted response space.

Consider an ICE plot for a selected feature  $x_S$  with a given observation  $X_i^C$ , where  $X^S = \{x_S\}$ ,  $X^C$  is the complementary set of  $X^S$  in the whole feature set  $X$ , and  $X_i = (X_i^S, X_i^C)$  is the  $i$ th observation. The ICE plot corresponds to the summand of the summation in “Equation 32-1” on page 32-7069:

$$f_i^S(X^S) = f(X^S, X_i^C).$$

`plotPartialDependence` plots  $f_i^S(X^S)$  for each observation  $i$  when you specify 'Conditional' as 'absolute'. If you specify 'Conditional' as 'centered', `plotPartialDependence` draws all plots after removing level effects due to different observations:

$$f_{i,\text{centered}}^S(X^S) = f(X^S, X_i^C) - f(\min(X^S), X_i^C).$$

This subtraction ensures that each plot starts from zero, so that you can examine the cumulative effect of  $X^S$  and the interactions between  $X^S$  and  $X^C$ .

## Weighted Traversal Algorithm

The weighted traversal algorithm[1] is a method to estimate partial dependence for a tree-based regression model. The estimated partial dependence is the weighted average of response values corresponding to the leaf nodes visited during the tree traversal.

Let  $X^S$  be a subset of the whole feature set  $X$  and  $X^C$  be the complementary set of  $X^S$  in  $X$ . For each  $X^S$  value to compute partial dependence, the algorithm traverses a tree from the root (beginning) node down to leaf (terminal) nodes and finds the weights of leaf nodes. The traversal starts by assigning a weight value of one at the root node. If a node splits by  $X^S$ , the algorithm traverses to the appropriate child node depending on the  $X^S$  value. The weight of the child node becomes the same value as its parent node. If a node splits by  $X^C$ , the algorithm traverses to both child nodes. The weight of each child node becomes a value of its parent node multiplied by the fraction of observations corresponding to each child node. After completing the tree traversal, the algorithm computes the weighted average by using the assigned weights.

For an ensemble of regression trees, the estimated partial dependence is an average of the weighted averages over the individual regression trees.

## Algorithms

plotPartialDependence uses a predict function to predict responses. plotPartialDependence chooses the proper predict function according to Mdl and runs predict with its default settings. For details about each predict function, see the predict function in the following table. If Mdl is a tree-based model and 'Conditional' is 'none', then plotPartialDependence uses the weighted traversal algorithm instead of the predict function. For details, see “Weighted Traversal Algorithm” on page 32-7070.

Trained Model Type	Regression Model Object	Function to Predict Responses
Bootstrap aggregation for ensemble of decision trees	CompactTreeBagger	predict
Bootstrap aggregation for ensemble of decision trees	TreeBagger	predict
Ensemble of regression models	RegressionEnsemble, RegressionBaggedEnsemble, CompactRegressionEnsemble	predict
Gaussian process regression	RegressionGP, CompactRegressionGP	predict
Generalized linear mixed-effect model	GeneralizedLinearMixedModel	predict
Generalized linear model	GeneralizedLinearModel, CompactGeneralizedLinearModel	predict
Linear mixed-effect model	LinearMixedModel	predict
Linear regression	LinearModel, CompactLinearModel	predict

Trained Model Type	Regression Model Object	Function to Predict Responses
Linear regression for high-dimensional data	<code>RegressionLinear</code>	<code>predict</code>
Nonlinear regression	<code>NonLinearModel</code>	<code>predict</code>
Regression tree	<code>RegressionTree</code> , <code>CompactRegressionTree</code>	<code>predict</code>
Support vector machine regression	<code>RegressionSVM</code> , <code>CompactRegressionSVM</code>	<code>predict</code>

## References

- [1] Friedman, J. H. “Greedy function approximation: a gradient boosting machine.” *The Annals of Statistics*. Vol. 29, No. 5, 2001, pp. 1189-1232.
- [2] Goldstein, A., A. Kapelner, J. Bleich, and E. Pitkin. “Peeking inside the black box: Visualizing statistical learning with plots of individual conditional expectation.” *Journal of Computational and Graphical Statistics*. Vol. 24, No. 1, 2015, pp. 44-65.
- [3] Hastie, T., R. Tibshirani, and J. H. Friedman. *The Elements of Statistical Learning*. New York: Springer, 2001.

## See Also

`oobPermutedPredictorImportance` | `predictorImportance` | `predictorImportance` | `relieff` | `sequentialfs`

Introduced in R2017b

## fitckernel

Fit Gaussian kernel classification model using feature expansion for big data

`fitckernel` trains a binary, Gaussian kernel classification for nonlinear classification with big data. `fitckernel` maps data in a low-dimensional space into a high-dimensional space, then fits a linear model in the high-dimensional space by minimizing the regularized objective function. The linear model in the high-dimensional space is equivalent to the model with Gaussian kernel in the low-dimensional space. Available linear classification models include regularized support vector machines (SVM) and logistic regression models.

## Syntax

```
Mdl = fitckernel(X,Y)
Mdl = fitckernel(X,Y,Name,Value)
[Mdl,FitInfo] = fitckernel(____)
```

## Description

`Mdl = fitckernel(X,Y)` returns a binary, Gaussian kernel classification model trained using the predictors `X` and class label `Y`. `fitckernel` maps the predictors `X` in a low-dimensional space into a high-dimensional space, then fits a binary support vector machine to the transformed predictors and class label `Y`. This linear model is equivalent to the Gaussian kernel classification model in the low-dimensional space.

`Mdl = fitckernel(X,Y,Name,Value)` returns a kernel classification model with additional options specified by one or more `Name, Value` pair arguments. For example, you can implement logistic regression or specify the dimension of the extended space.

`[Mdl,FitInfo] = fitckernel(____)` also returns the fit information in the structure array `FitInfo` using any of the previous syntaxes.

## Examples

### Train Kernel Classification Model

Train a binary, kernel classification model using support vector machines.

Load the `ionosphere` data. This data has 34 predictors and 351 binary responses for radar returns, either bad ('b') or good ('g').

```
load ionosphere
[n,p] = size(X)
resp = unique(Y)
```

```
n =
```

```
351
```

```
p =
```

```
34
```

```
resp =
```

```
2x1 cell array
```

```
 {'b'}
```

```
 {'g'}
```

Train a binary, kernel classification model that can identify whether the radar return is bad ('b') or good ('g'). Determine how well the optimization algorithm fit the model to the data by extracting a fit summary.

```
rng('default') % For reproducibility
[Mdl,FitInfo] = fitckernel(X,Y)
```

```
Mdl =
```

```
ClassificationKernel
    ResponseName: 'Y'
    ClassNames: {'b' 'g'}
    ScoreTransform: 'none'
    KernelScale: 1
```



```
BoxConstraint: 1
      Lambda: 0.0028
      Learner: 'svm'
Transformation: 'FastFood'
ExpansionDimension: 2048
```

```
FitInfo =
```

```
struct with fields:
```

```
      Solver: 'LBFGS-fast'
      LossFunction: 'hinge'
      Lambda: 0.0028
      BetaTolerance: 1.0000e-04
      GradientTolerance: 1.0000e-06
      ObjectiveValue: 0.2604
      GradientMagnitude: 0.0028
      RelativeChangeInBeta: 8.2512e-05
      FitTime: 0.2019
      History: []
```

Mdl is a ClassificationKernel model. You can pass Mdl and the training data or new data to the function `loss` to inspect the in-sample classification error. Or, you can pass Mdl and new predictor data to the function `predict` to predict class labels for new observations. You can also pass Mdl and the training data to the function `resume` to continue training.

FitInfo is a structure array containing, among other things, how long the solver took to fit the model to the data (`FitTime`) and history of optimization information for every iteration (`History`). It is good practice to use FitInfo to determine whether optimization-termination measurements are satisfactory.

For better accuracy, you can increase the maximum number of optimization iterations (`'IterationLimit'`) and decrease the tolerance values (`'BetaTolerance'` and `'GradientTolerance'`) by using the name-value pair arguments. This can improve measures like `ObjectiveValue` and `RelativeChangeInBeta` in FitInfo. You can also optimize model parameters such as the kernel scale parameter (`'KernelScale'`),

box constraint ('BoxConstraint'), and regularization term strength ('Lambda') by using the `bayseopt` function.

### Optimize Kernel Classifier Using Bayesian Optimization

Define a function handle that computes the classification loss for a kernel classifier at parameters you want to optimize. Then use `bayesopt` to optimize the parameters of the kernel classification model with respect to the classification loss.

Load the `ionosphere` data. This data has 34 predictors and 351 binary responses for radar returns, either bad ('b') or good ('g').

```
load ionosphere
```

Partition the data set into training and test sets. Specify a 30% holdout sample for testing.

```
rng('default') % For reproducibility
Partition = cvpartition(Y, 'Holdout', 0.30);
trainingInds = training(Partition); % Indices for the training set
testInds = test(Partition); % Indices for the test set
```

Prepare variables and the objective function for Bayesian optimization.

Set up a function that takes an input `z = [sigma, lambda]` and returns the cross-validation loss value of `z`. Take the components of `z` as positive, log-transformed variables between `1e-5` and `1e5`. Choose a wide range, because you don't know which values are likely to be good.

```
sigma = optimizableVariable('sigma', [1e-5, 1e5], 'Transform', 'log');
lambda = optimizableVariable('lambda', [1e-5, 1e5], 'Transform', 'log');
minfn = @(z) loss(fitckernel(X(trainingInds,:), Y(trainingInds), ...
    'KernelScale', z.sigma, 'Lambda', z.lambda), X(testInds,:), Y(testInds));
```

Search for the best parameters `[sigma, lambda]` using `bayesopt`. For reproducibility, choose the 'expected-improvement-plus' acquisition function. The default acquisition function depends on run time, and so can give varying results.

```
results = bayesopt(minfn, [sigma, lambda], ...
    'AcquisitionFunctionName', 'expected-improvement-plus')
```

Iter	Eval result	Objective	Objective runtime	BestSoFar (observed)	BestSoFar (estim.)	sigma
1	Best	0.36179	0.22974	0.36179	0.36179	0.00053467
2	Accept	0.36179	0.32748	0.36179	0.36179	7982.4
3	Best	0.35668	0.48843	0.35668	0.35904	9.8949e-05
4	Accept	0.36179	0.19132	0.35668	0.35669	1474
5	Accept	0.40518	0.75111	0.35668	0.36945	1.4899e-05
6	Accept	0.36179	0.15637	0.35668	0.36817	90193
7	Accept	0.36179	0.17335	0.35668	0.36726	2.5447
8	Accept	0.36179	0.33154	0.35668	0.36657	14.287
9	Best	0.11577	0.7593	0.11577	0.11579	8.3235
10	Best	0.077439	0.4415	0.077439	0.079849	3.1584
11	Accept	0.14314	0.3246	0.077439	0.077523	0.85548
12	Accept	0.09621	0.66004	0.077439	0.078361	3.0357
13	Accept	0.087217	0.5236	0.077439	0.084046	2.776
14	Accept	0.077439	0.58095	0.077439	0.080316	3.4657
15	Accept	0.077439	0.45411	0.077439	0.077129	4.3343
16	Accept	0.077439	0.75467	0.077439	0.076854	4.2753
17	Best	0.067269	1.2423	0.067269	0.072346	3.5395
18	Accept	0.087217	0.81662	0.067269	0.077752	3.3933
19	Accept	0.086825	0.76143	0.067269	0.07767	4.1701
20	Accept	0.086825	0.3918	0.067269	0.07859	9.5355
Iter	Eval result	Objective	Objective runtime	BestSoFar (observed)	BestSoFar (estim.)	sigma
21	Accept	0.077439	0.53655	0.067269	0.079831	12.342
22	Best	0.057491	0.3626	0.057491	0.060366	6.8552
23	Accept	0.067269	0.49405	0.057491	0.063186	6.2391
24	Accept	0.067269	0.36948	0.057491	0.06386	7.2309
25	Accept	0.057491	0.32256	0.057491	0.061893	6.9206
26	Accept	0.057491	0.44574	0.057491	0.060032	6.4806
27	Accept	0.067661	0.60574	0.057491	0.05997	4.2011
28	Accept	0.067269	0.34046	0.057491	0.061204	6.2727
29	Accept	0.36179	0.29576	0.057491	0.061171	0.021085
30	Accept	0.36179	0.1805	0.057491	0.061178	1.0349e-05

Optimization completed.

MaxObjectiveEvaluations of 30 reached.

Total function evaluations: 30

Total elapsed time: 62.9849 seconds.

Total objective function evaluation time: 14.3138

Best observed feasible point:

sigma	lambda
6.8552	0.00023984

Observed objective function value = 0.057491

Estimated objective function value = 0.061178

Function evaluation time = 0.3626

Best estimated feasible point (according to models):

sigma	lambda
6.9206	0.00026979

Estimated objective function value = 0.061178

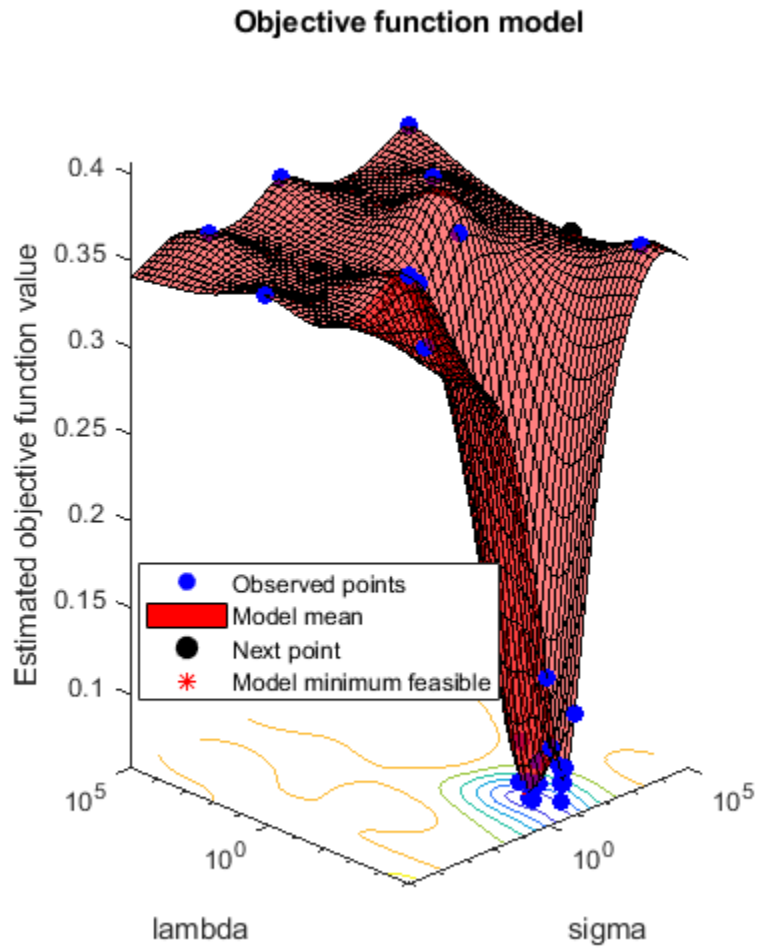
Estimated function evaluation time = 0.44663

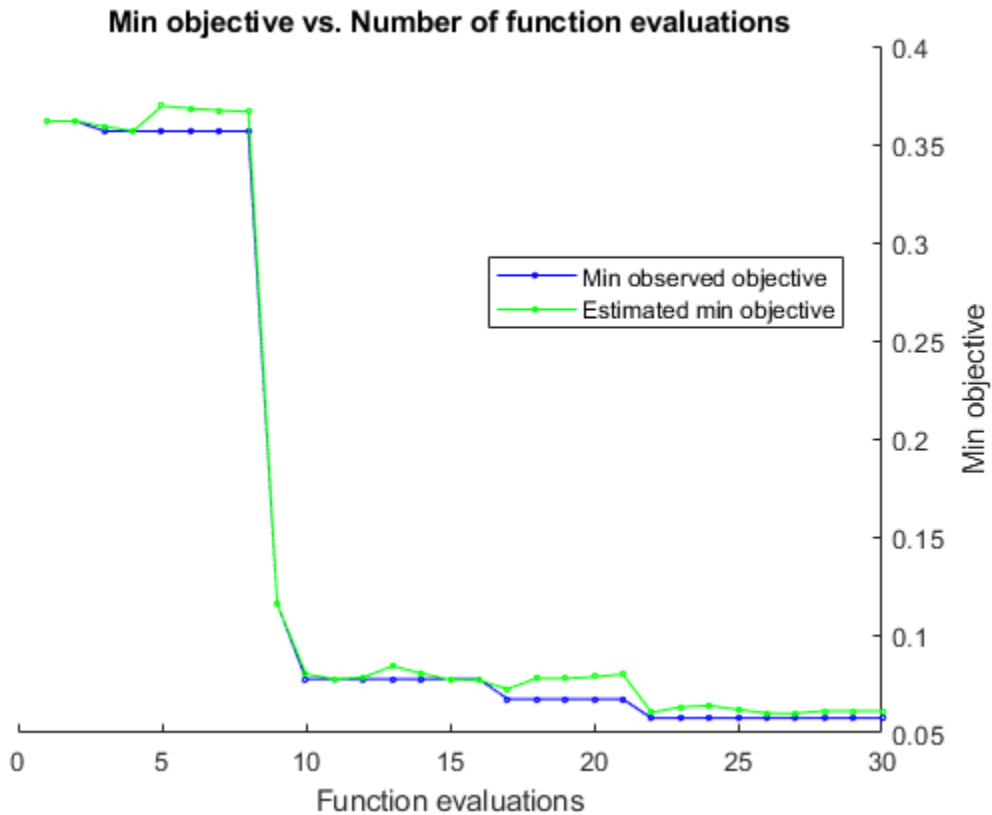
results =

BayesianOptimization with properties:

```
ObjectiveFcn: [function_handle]
VariableDescriptions: [1x2 optimizableVariable]
Options: [1x1 struct]
MinObjective: 0.0575
XAtMinObjective: [1x2 table]
MinEstimatedObjective: 0.0612
XAtMinEstimatedObjective: [1x2 table]
NumObjectiveEvaluations: 30
TotalElapsedTime: 62.9849
NextPoint: [1x2 table]
XTrace: [30x2 table]
ObjectiveTrace: [30x1 double]
ConstraintsTrace: []
UserDataTrace: {30x1 cell}
ObjectiveEvaluationTimeTrace: [30x1 double]
IterationTimeTrace: [30x1 double]
ErrorTrace: [30x1 double]
FeasibilityTrace: [30x1 logical]
```

```
FeasibilityProbabilityTrace: [30x1 double]
IndexOfMinimumTrace: [30x1 double]
ObjectiveMinimumTrace: [30x1 double]
EstimatedObjectiveMinimumTrace: [30x1 double]
```





Use the results to train a new, optimized kernel classifier.

```
z(1) = results.XAtMinObjective.sigma;
z(2) = results.XAtMinObjective.lambda;
Mdl = fitckernel(X(trainingInds,:),Y(trainingInds), ...
    'KernelScale',z(1),'Lambda',z(2));
```

Predict the test-sample labels.

```
labelTest = predict(Mdl,X(testInds,:));
```

Construct a confusion matrix for the test data.

```
ConfusionTest = confusionmat(Y(testInds),labelTest)
```

```
ConfusionTest =  
  
    34     3  
     3    65
```

The model misclassifies three bad radar returns as being good returns, and three good radar returns as being bad returns.

Estimate the classification error for the test data.

```
L = loss(Mdl,X(testInds,:),Y(testInds))
```

```
L =  
  
    0.0575
```

If the optimized model `Mdl` is not good enough, you can run the optimization process again with setting a higher dimension of the extended space by using the `'ExpansionDimension'` name-value pair argument.

For big data, the optimization procedure can take a long time. If the data set is too large to run optimization procedure, you can try to optimize the parameters using only partial data. Use the function `datasample` and specify `'Replace', 'false'` to sample data without replacement.

## Input Arguments

### **x** — Predictor data

numeric matrix

Predictor data, specified as an  $n$ -by- $p$  numeric matrix, where  $n$  is the number of observations and  $p$  is the number of predictors.

The length of `Y` and the number of observations in `X` must be equal.

Data Types: `single` | `double`

**Y — Class labels**

categorical array | character array | logical vector | vector of numeric values | cell array of character vectors

Class labels to which the classification model is trained, specified as a categorical or character array, logical or numeric vector, or cell array of character vectors.

- `fitckernel` only supports binary classification. Either `Y` must contain exactly two distinct classes, or you must specify two classes for training using the `ClassNames` name-value pair argument.
- If `Y` is a character array, then each element must correspond to one row of the array.
- The length of `Y` and the number of observations in `X` must be equal.
- It is good practice to specify the class order using the `ClassNames` name-value pair argument.

Data Types: `char` | `cell` | `categorical` | `logical` | `single` | `double`

---

**Note** `fitckernel` removes missing observations, that is, observations with any of these characteristics:

- NaN, empty character vector (' '), and `<undefined>` elements in the response (`Y`)
  - At least one NaN value in a predictor observation (row in `X`)
  - NaN value or 0 weight (`Weights`)
- 

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `Mdl = fitckernel(X,Y,'Learner','logistic','ExpansionDimension',2^15,'KernelScale','auto')` implements logistic regression after mapping the predictor data to the  $2^{15}$  dimensional space using feature expansion with a kernel scale parameter selected by a heuristic procedure.



## Kernel Classification Options

### Learner — Linear classification model type

'svm' (default) | 'logistic'

Linear classification model type, specified as the comma-separated pair consisting of 'Learner' and 'svm' or 'logistic'.

In this table,  $f(x) = T(x)\beta + b$ .

- $x$  is an observation (row vector) from  $p$  predictor variables.
- $T(\cdot)$  is a transformation of an observation (row vector) for feature expansion.  $T(x)$  maps  $x$  in  $\mathbb{R}^p$  to a high dimensional space ( $\mathbb{R}^m$ ).
- $\beta$  is a vector of  $m$  coefficients.
- $b$  is the scalar bias.

Value	Algorithm	Response Range	Loss Function
'svm'	Support vector machine	$y \in \{-1, 1\}$ ; 1 for the positive class, and -1 otherwise.	Hinge: $\ell[y, f(x)] = \max[0, 1 - yf(x)]$
'logistic'	Logistic regression	Same as 'svm'	Deviance (logistic): $\ell[y, f(x)] = \log\{1 + \exp[-yf(x)]\}$

Example: 'Learner', 'logistic'

Data Types: char

### ExpansionDimension — Dimension of extended space

'auto' (default) | positive integer

Dimension of the extended space, specified as the comma-separated pair 'ExpansionDimension' and 'auto' or a positive integer. For 'auto', fitckernel selects the dimension using  $2.^{\text{ceil}(\min(\log_2(p) + 5, 15))}$ , where  $p$  is the number of predictors.

Example: 'ExpansionDimension', 2^15

Data Types: char | single | double

**KernelScale — Kernel scale parameter**

1 (default) | 'auto' | positive scalar

Kernel scale parameter, specified as the comma-separated pair consisting of 'KernelScale' and 'auto' or a positive scalar. The software obtains random basis for random feature expansion using the kernel scale parameter. For details, see “Random Feature Expansion” on page 32-7092.

If you specify 'auto', then the software selects an appropriate kernel scale parameter using a heuristic procedure. This heuristic procedure uses subsampling, so estimates can vary from one call to another. Therefore, to reproduce results, set a random number seed using `rng` before training.

Example: 'KernelScale', 'auto'

Data Types: char | single | double

**BoxConstraint — Box constraint**

1 (default) | positive scalar

Box constraint on page 32-7093, specified as the comma-separated pair consisting of 'BoxConstraint' and a positive scalar.

This argument is valid only when 'Learner' is 'svm' (default) and you do not specify a value for 'Lambda'. You can specify either 'BoxConstraint' or 'Lambda' because the box constraint ( $C$ ) and the regularization term strength ( $\lambda$ ) are related by  $C = 1/(\lambda n)$ , where  $n$  is the number of observations (rows in  $X$ ).

Example: 'BoxConstraint', 100

Data Types: single | double

**Lambda — Regularization term strength**

'auto' (default) | nonnegative scalar

Regularization term strength, specified as the comma-separated pair consisting of 'Lambda' and 'auto' or a nonnegative scalar.

For 'auto', the value of 'Lambda' is  $1/n$ , where  $n$  is the number of observations (rows in  $X$ ).

You can specify either 'BoxConstraint' or 'Lambda' because the box constraint ( $C$ ) and the regularization term strength ( $\lambda$ ) are related by  $C = 1/(\lambda n)$ .

Example: 'Lambda', 0.01

Data Types: char | single | double

### Convergence Controls

#### **BetaTolerance** — Relative tolerance on linear coefficients and bias term

1e-5 (default) | nonnegative scalar

Relative tolerance on the linear coefficients and the bias term (intercept), specified as the comma-separated pair consisting of 'BetaTolerance' and a nonnegative scalar.

Let  $B_t = \begin{bmatrix} \beta_t' & b_t \end{bmatrix}$ , that is, the vector of the coefficients and the bias term at optimization iteration  $t$ . If  $\left\| \frac{B_t - B_{t-1}}{B_t} \right\|_2 < \text{BetaTolerance}$ , then optimization terminates.

If you also specify GradientTolerance, then optimization terminates when the software satisfies either stopping criterion.

Example: 'BetaTolerance', 1e-6

Data Types: single | double

#### **GradientTolerance** — Absolute gradient tolerance

1e-6 (default) | nonnegative scalar

Absolute gradient tolerance, specified as the comma-separated pair consisting of 'GradientTolerance' and a nonnegative scalar.

Let  $\nabla L_t$  be the gradient vector of the objective function with respect to the coefficients and bias term at optimization iteration  $t$ . If  $\|\nabla L_t\|_\infty = \max |\nabla L_t| < \text{GradientTolerance}$ , then optimization terminates.

If you also specify BetaTolerance, then optimization terminates when the software satisfies either stopping criterion.

Example: 'GradientTolerance', 1e-5

Data Types: single | double

**IterationLimit** — Maximal number of optimization iterations

positive integer

Maximal number of optimization iterations, specified as the comma-separated pair consisting of 'IterationLimit' and a positive integer.

The default value is 1000 if the transformed data fits in memory specified by using `BlockSize`. Otherwise, the default value is 100.

Example: `'IterationLimit', 500`

Data Types: `single` | `double`

**Other Kernel Classification Options****BlockSize** — Maximum amount of allocated memory $4e^3$  (4GB) (default) | positive scalar

Maximum amount of allocated memory (in megabytes), specified as the comma-separated pair consisting of 'BlockSize' and a positive scalar.

If `fitckernel` requires more memory than the value of 'BlockSize' to hold the transformed predictor data, the software uses a block-wise strategy. For details about the block-wise strategy, see “Algorithms” on page 32-7094.

Example: `'BlockSize', 1e4`

Data Types: `single` | `double`

**RandomStream** — Random number stream

global stream (default) | random stream object

Random number stream for reproducibility of data transformation, specified as the comma-separated pair consisting of 'RandomStream' and a random stream object. For details, see “Random Feature Expansion” on page 32-7092.

Use 'RandomStream' for reproducing the random basis functions that `fitckernel` uses to transform the data in `X` to a high dimensional space.

For details, see “Managing the Global Stream” (MATLAB) and “Creating and Controlling a Random Number Stream” (MATLAB).

Example: `'RandomStream', RandStream('mlfg6331_64')`

**HessianHistorySize — Size of history buffer for Hessian approximation**

15 (default) | positive integer

Size of history buffer for Hessian approximation, specified as the comma-separated pair consisting of 'HessianHistorySize' and a positive integer. That is, at each iteration, the software composes the Hessian using statistics from the latest HessianHistorySize iterations.

Example: 'HessianHistorySize',10

Data Types: single | double

**Verbose — Verbosity level**

0 (default) | 1

Verbosity level, specified as the comma-separated pair consisting of 'Verbose' and either 0 or 1. Verbose controls the amount of diagnostic information fitckernel displays at the command line.

Value	Description
0	fitckernel does not display diagnostic information.
1	fitckernel displays and stores the value of the objective function, gradient magnitude, and other diagnostic information. FitInfo.History contains the diagnostic information.

Example: 'Verbose',1

Data Types: single | double

**Other Classification Options****ClassNames — Names of classes to use for training**

categorical array | cell array of character vectors | character array | logical vector | numeric vector

Names of classes to use for training, specified as the comma-separated pair consisting of 'ClassNames' and a categorical or character array, logical or numeric vector, or cell array of character vectors. ClassNames must be the same data type as Y.

If `ClassNames` is a character array, then each element must correspond to one *row* of the array.

Use `ClassNames` to:

- Order the classes during training.
- Specify the order of any input or output argument dimension that corresponds to the class order. For example, use `ClassNames` to specify the order of the dimensions of `Cost` or the column order of classification scores returned by `predict`.
- Select a subset of classes for training. For example, suppose that the set of all distinct class names in `Y` is `{'a', 'b', 'c'}`. To train the model using observations from classes 'a' and 'c' only, specify `'ClassNames', {'a', 'c'}`.

The default is the set of all distinct class names in `Y`.

Example: `'ClassNames', {'b', 'g'}`

Data Types: categorical | char | logical | single | double | cell

### **Cost — Misclassification cost**

square matrix | structure array

Misclassification cost, specified as the comma-separated pair consisting of `'Cost'` and a square matrix or structure. If you specify:

- The square matrix `Cost`, then `Cost(i, j)` is the cost of classifying a point into class `j` if its true class is `i`. That is, the rows correspond to the true class and the columns correspond to the predicted class. To specify the class order for the corresponding rows and columns of `Cost`, also specify the `ClassNames` name-value pair argument.
- The structure `S`, then it must have two fields:
  - `S.ClassNames`, which contains the class names as a variable of the same data type as `Y`
  - `S.ClassificationCosts`, which contains the cost matrix with rows and columns ordered as in `S.ClassNames`

The default is `ones(K) - eye(K)`, where `K` is the number of distinct classes.

`fitckernel` uses `Cost` to adjust the prior class probabilities specified in `Prior`. Then, `fitckernel` uses the adjusted prior probabilities for training and resets the cost matrix to its default.

Example: 'Cost', [0 2; 1 0]

Data Types: single | double | struct

### **Prior — Prior probabilities**

'empirical' (default) | 'uniform' | numeric vector | structure array

Prior probabilities for each class, specified as the comma-separated pair consisting of 'Prior' and a character vector, numeric vector, or a structure.

This table summarizes the available options for setting prior probabilities.

Value	Description
'empirical'	The class prior probabilities are the class relative frequencies in $Y$ .
'uniform'	All class prior probabilities are equal to $1/K$ , where $K$ is the number of classes.
numeric vector	Each element is a class prior probability. Order the elements according to the order that appears in $Y$ . If you specify the order using the 'ClassNames' name-value pair argument, then order the elements accordingly.
structure	A structure $S$ with two fields: <ul style="list-style-type: none"> <li>• <math>S.ClassNames</math> contains the class names as a variable of the same type as <math>Y</math>.</li> <li>• <math>S.ClassProbs</math> contains a vector of corresponding prior probabilities.</li> </ul>

fitckernel normalizes the prior probabilities in Prior to sum to 1.

Example: struct('ClassNames', {'setosa', 'versicolor'}, 'ClassProbs', 1:2)

Data Types: char | double | single | struct

### **ScoreTransform — Score transformation**

'none' (default) | character vector | function handle

Score transformation, specified as the comma-separated pair consisting of 'ScoreTransform' and either a character vector or a function handle.

This table summarizes the available character vectors.

Value	Description
'doublelogit'	$1/(1 + e^{-2x})$
'invlogit'	$\log(x / (1-x))$
'ismax'	Set the score for the class with the largest score to 1, and set the scores for all other classes to 0.
'logit'	$1/(1 + e^{-x})$
'none' or 'identity'	$x$ (no transformation)
'sign'	-1 for $x < 0$ 0 for $x = 0$ 1 for $x > 0$
'symmetric'	$2x - 1$
'symmetricismax'	Set the score for the class with the largest score to 1, and set the scores for all other classes to -1.
'symmetriclogit'	$2/(1 + e^{-x}) - 1$

For a MATLAB function or a function you define, use its function handle. The function handle must accept a matrix (the original scores) and return a matrix of the same size (the transformed scores).

Example: 'ScoreTransform', 'logit'

Data Types: char | function\_handle

### Weights — Observation weights

positive numeric vector

Observation weights, specified as the comma-separated pair consisting of 'Weights' and a positive numeric vector of length  $n$ , where  $n$  is the number of observations in  $X$ . `fitckernel` weighs the observations in  $X$  with the corresponding values in `Weights`.

The default value is `ones(n, 1)`.

`fitckernel` normalizes `Weights` to sum up to the value of the prior probability in the respective class.



Example: 'Weights',w

Data Types: single | double

## Output Arguments

### **Mdl** — Trained kernel classification model

ClassificationKernel model object

Trained kernel classification model, returned as a ClassificationKernel model object.

To reference properties of Mdl, use dot notation. For example, enter `Mdl.ExpansionDimension` in the Command Window to display the dimension of the extended space.

---

**Note** Unlike many other classification models, and for economical memory usage, ClassificationKernel model object does not store the training data or training process details (for example, convergence history).

---

### **FitInfo** — Optimization details

structure array

Optimization details, returned as a structure array including fields described in this table. Fields specify final values or name-value pair argument specifications.

Field	Description
Solver	Objective function minimization technique: 'LBFGS-fast', 'LBFGS-blockwise', or 'LBFGS-tall'. For details, see “Algorithms” on page 32-7094.
LossFunction	Loss function. Either 'hinge' or 'logit' depending on linear classification model type. See Learner.
Lambda	Regularization term strength. See Lambda.
BetaTolerance	Relative tolerance on linear coefficients and bias term. See BetaTolerance.

Field	Description
GradientTolerance	Absolute gradient tolerance. See GradientTolerance.
ObjectiveValue	Value of the objective function when optimization terminates. The classification loss plus the regularization term compose the objective function.
GradientMagnitude	Infinite norm of the gradient vector of the objective function when optimization terminates. See BetaTolerance.
RelativeChangeInBeta	Relative changes in the linear coefficients and the bias term when optimization terminates. See GradientTolerance.
FitTime	Elapsed, wall-clock time took to fit the model to the data in seconds.
History	History of optimization information. This field is empty ([]) if you specify 'Verbose', 0. For details, see Verbose and “Algorithms” on page 32-7094.

To access fields, use dot notation. For example, to access the vector of objective function values for each iteration, enter `FitInfo.Objective` in the Command Window.

It is good practice to examine `FitInfo` to assess whether convergence is satisfactory.

## Definitions

### Random Feature Expansion

The random feature expansion such as Random Kitchen Sinks[1] and Fastfood[2] is a scheme to approximate Gaussian kernels of the kernel classification algorithm for big data in a computationally efficient way.

The kernel classification algorithm searches for an optimal hyperplane that separates the data into two classes after mapping features into a high dimensional space. Nonlinear features that are not linearly separable in a low dimensional space can be separable in the expanded high dimensional space. All the calculations for hyperplane classification use nothing more than dot products. You can obtain a nonlinear classification model by replacing the dot product  $x_1x_2'$  with a nonlinear kernel function

$G(x_1, x_2) = \langle \varphi(x_1), \varphi(x_2) \rangle$ , where  $x_i$  is the  $i$ th observation (row vector) and  $\varphi(x_i)$  is a transformation that maps  $x_i$  to a high-dimensional space (called the “kernel trick”). However, evaluating  $G(x_1, x_2)$  (Gram matrix) for each pair of observations is computationally expensive for large data set (large  $n$ ).

The random feature expansion scheme finds a random transformation such that its dot product approximates the Gaussian kernel. That is,

$$G(x_1, x_2) = \langle \varphi(x_1), \varphi(x_2) \rangle \approx T(x_1)T(x_2)',$$

where  $T(x)$  maps  $x$  in  $\mathbb{R}^p$  to a high dimensional space ( $\mathbb{R}^m$ ). The Random Kitchen Sink[1] scheme uses the following random transformation:

$$T(x) = m^{-1/2} \exp(iZx'),$$

where  $Z \in \mathbb{R}^{m \times p}$  is a sample drawn from  $N(0, \sigma^{-2})$  and  $\sigma^2$  is a kernel scale. This scheme requires  $O(mp)$  computation and storage. The Fastfood[2] scheme introduces another random basis  $V$  instead of  $Z$  using Hadamard matrices combined with Gaussian scaling matrices. This random basis reduces computation cost to  $O(m \log p)$  and reduces storage to  $O(m)$ .

The function `fitckernel` uses the Fastfood scheme for random feature expansion and uses linear classification to train a Gaussian kernel classification model. Unlike solvers in the function `fitcsvm` that require computation of the  $n$ -by- $n$  Gram matrix, the solver in `fitckernel` only needs to form a matrix of size  $n$ -by- $m$  with  $m$  typically much less than  $n$  for big data.

## Box Constraint

A parameter that controls the maximum penalty imposed on margin-violating observations, and aids in preventing overfitting (regularization). Increasing the box constraint can lead to longer training times.

The box constraint ( $C$ ) and the regularization term strength ( $\lambda$ ) are related by  $C = 1/(\lambda n)$ , where  $n$  is the number of observations.

## Algorithms

`fitckernel` minimizes the regularized objective function using Limited-memory Broyden-Fletcher-Goldfarb-Shanno (LBFGS) solver with ridge ( $L_2$ ) regularization. To find the type of LBFGS solver used for training, type `FitInfo.Solver` in the Command Window.

- `'LBFGS-fast'` — Limited-memory Broyden-Fletcher-Goldfarb-Shanno (LBFGS) solver
- `'LBFGS-blockwise'` — LBFGS solver with a block-wise strategy. If `fitckernel` requires more memory than the value of `BlockSize` to hold the transformed predictor data, the software uses a block-wise strategy.
- `'LBFGS-tall'` — LBFGS solver with a block-wise strategy for tall arrays.

When `fitckernel` uses a block-wise strategy, `fitckernel` implements LBFGS by distributing the calculation of the loss and gradient among different parts of the data at each iteration. Also, `fitckernel` refines the initial estimates of the linear coefficients and the bias term by fitting the model locally to parts of the data and combining the coefficients by averaging. If you specify `'Verbose', 1`, then `fitckernel` displays optimization information for each data pass and stores the information in the `History` field of `FitInfo`.

When `fitckernel` does not use the block-wise strategy, the initial estimates are zeros. If you specify `'Verbose', 1`, then `fitckernel` displays optimization information for each iteration and stores the information in the `History` field of `FitInfo`.

## References

- [1] Rahimi, A., and B. Recht. “Random Features for Large-Scale Kernel Machines.” *Advances in Neural Information Processing Systems*. Vol 20, 2008, pp. 1177–1184.
- [2] Le, Q., T. Sarlós, and A. Smola. “Fastfood — Approximating Kernel Expansions in Loglinear Time.” *Proceedings of the 30th International Conference on Machine Learning*. Vol. 28, No. 3, 2013, pp. 244–252.
- [3] Huang, P. S., H. Avron, T. N. Sainath, V. Sindhvani, and B. Ramabhadran. “Kernel methods match Deep Neural Networks on TIMIT.” *2014 IEEE International Conference on Acoustics, Speech and Signal Processing*. 2014, pp. 205–209.

## Extended Capabilities

### Tall Arrays

Calculate with arrays that have more rows than fit in memory.

Usage notes and limitations:

- Some name-value pairs have different defaults compared to the in-memory `fitckernel` function.
  - `'Verbose'` — Default value is 1.
  - `'BetaTolerance'` — Default value is relaxed to  $1e-3$ .
  - `'GradientTolerance'` — Default value is relaxed to  $1e-5$ .
  - `'IterationLimit'` — Default value is relaxed to 20.
- If `'KernelScale'` is `'auto'`, then `fitckernel` uses the random stream controlled by `ta1rng` for subsampling. For reproducibility, you must set a random number seed for both the global stream and the random stream controlled by `ta1rng`.
- If `'Lambda'` is `'auto'`, then `fitckernel` might take an extra pass through the data to calculate the number of observations in  $X$ .
- `fitckernel` uses a block-wise strategy. For details, see “Algorithms” on page 32-7094.

For more information, see “Tall Arrays” (MATLAB).

### See Also

`ClassificationKernel` | `bayesopt` | `fitclinear` | `fitcsvm` | `predict` | `resume`

### Topics

“Train SVM Classifiers Using a Gaussian Kernel” on page 18-195

Introduced in R2017b

# ClassificationKernel

Gaussian kernel classification model using feature expansion for big data

## Description

`ClassificationKernel` is a trained model object for binary, Gaussian kernel classification model using feature expansion for big data.

Unlike other classification models, and for economical memory usage, `ClassificationKernel` model objects do not store the training data. However, they do store, for example, the dimension of the extended space, kernel scale parameter, prior-class probabilities, and the regularization strength.

You can use trained `ClassificationKernel` models to continue training using the training data and to predict labels or classification scores for new data. For details, see `resume` and `predict`.

## Creation

Create a `ClassificationKernel` object using `fitkernel`. The function `fitkernel` maps data in a low-dimensional space into a high-dimensional space, then fits a linear model in the high-dimensional space by minimizing the regularized objective function. The linear model in the high-dimensional space is equivalent to the model with Gaussian kernel in the low-dimensional space. Available linear classification models include regularized support vector machines (SVM) and logistic regression models.

## Properties

### Kernel Classification Properties

**Learner** — Linear classification model type

`'logistic' | 'svm'`

Linear classification model type, specified as `'logistic'` or `'svm'`.

In this table,  $f(x) = T(x)\beta + b$ .

- $x$  is an observation (row vector) from  $p$  predictor variables.
- $T(\bullet)$  is a transformation of an observation (row vector) for feature expansion.  $T(x)$  maps  $x$  in  $\mathbb{R}^p$  to a high dimensional space ( $\mathbb{R}^m$ ).
- $\beta$  is a vector of  $m$  coefficients.
- $b$  is the scalar bias.

Value	Algorithm	Loss Function	FittedLoss Value
'logistic'	Logistic regression	Deviance (logistic): $\ell[y, f(x)] = \log\{1 + \exp[-yf(x)]\}$	'logit'
'svm'	Support vector machine	Hinge: $\ell[y, f(x)] = \max[0, 1 - yf(x)]$	'hinge'

#### **ExpansionDimension** — Dimension of extended space

positive integer

Dimension of the extended space, specified as a positive integer.

Data Types: single | double

#### **KernelScale** — Kernel scale parameter

positive scalar

Kernel scale parameter, specified as a positive scalar.

Data Types: char | single | double

#### **BoxConstraint** — Box constraint

positive scalar

Box constraint, specified as a positive scalar.

Data Types: double | single

#### **Lambda** — Regularization term strength

nonnegative scalar

Regularization term strength, specified as a nonnegative scalar.

Data Types: `single` | `double`

### **FittedLoss** — Loss function used to fit the linear model

'hinge' | 'logit'

Loss function used to fit the linear model, specified as 'hinge' or 'logit'.

Value	Algorithm	Loss Function	Learner Value
'hinge'	Support vector machine	Hinge: $\ell[y, f(x)] = \max[0, 1 - yf(x)]$	'svm'
'logit'	Logistic regression	Deviance (logistic): $\ell[y, f(x)] = \log\{1 + \exp[-yf(x)]\}$	'logistic'

### **Regularization** — Complexity penalty type

'ridge (L2)'

Complexity penalty type, which is always 'ridge (L2)'.

The software composes the objective function for minimization from the sum of the average loss function (see `FittedLoss`) and the regularization term, ridge ( $L_2$ ) penalty.

The ridge ( $L_2$ ) penalty is

$$\frac{\lambda}{2} \sum_{j=1}^p \beta_j^2$$

where  $\lambda$  specifies the regularization term strength (see `Lambda`). The software excludes the bias term ( $\beta_0$ ) from the regularization penalty.

## Other Classification Properties

### **CategoricalPredictors** — Indices of categorical predictors

[]

Indices of categorical predictors, which is always empty ([]) because a `ClassificationKernel` model does not support categorical predictors.



**ClassNames — Unique class labels**

categorical array | character array | logical vector | vector of numeric values | cell array of character vectors

Unique class labels used in training, specified as a categorical or character array, logical or numeric vector, or cell array of character vectors. `ClassNames` has the same data type as class labels ( $Y$ ). `ClassNames` also determines class order.

Data Types: `categorical` | `char` | `logical` | `single` | `double` | `cell`

**Cost — Misclassification costs**

square numeric matrix

This property is read-only.

Misclassification costs, specified as a square numeric matrix. `Cost` has  $K$  rows and columns, where  $K$  is the number of classes.

$Cost(i, j)$  is the cost of classifying a point into class  $j$  if its true class is  $i$ . The order of the rows and columns of `Cost` corresponds to the order of the classes in `ClassNames`.

Data Types: `double`

**ModelParameters — Parameters used for training model**

structure

Parameters used for training the `ClassificationKernel` model, specified as a structure.

Access fields of `ModelParameters` using dot notation. For example, access the relative tolerance on linear coefficients and bias term using

`Mdl.ModelParameters.BetaTolerance`.

Data Types: `struct`

**PredictorNames — Predictor names**

cell array of character vectors

Predictor names in their order of appearance in predictor data ( $X$ ), specified as a cell array of character vectors. `PredictorNames` has length equal to the number of columns in  $X$ .

Data Types: `cell`

**ExpandedPredictorNames** — Expanded predictor names

cell array of character vectors

Expanded predictor names, stored as a cell array of character vectors.

Because a `ClassificationKernel` model does not support categorical predictors, `ExpandedPredictorNames` and `PredictorNames` are equal.

Data Types: `cell`

**Prior** — Prior class probabilities

numeric vector

This property is read-only.

Prior class probabilities, specified as a numeric vector. `Prior` has as many elements as classes in `ClassNames`, and the order of the elements corresponds to the elements of `ClassNames`.

Data Types: `double`

**ScoreTransform** — Score transformation function to apply to predicted scores

function name | function handle

Score transformation function to apply to predicted scores, specified as a function name or function handle.

For kernel classification models and before the score transformation, the predicted

classification score for the observation  $x$  (row vector) is  $f(x) = T(x)\beta + b$ .  $T(\bullet)$  is a transformation of an observation for feature expansion.  $\beta$  and  $b$  correspond to `Mdl.Beta` and `Mdl.Bias`, respectively.

To change the score transformation function to, e.g., `function`, use dot notation.

- For a built-in function, enter this code and replace `function` with a character vector in the table.

```
Mdl.ScoreTransform = 'function';
```

Value	Description
'doublelogit'	$1/(1 + e^{-2x})$

Value	Description
'invlogit'	$\log(x / (1-x))$
'ismax'	Set the score for the class with the largest score to 1, and set the scores for all other classes to 0.
'logit'	$1/(1 + e^{-x})$
'none' or 'identity'	$x$ (no transformation)
'sign'	-1 for $x < 0$ 0 for $x = 0$ 1 for $x > 0$
'symmetric'	$2x - 1$
'symmetricismax'	Set the score for the class with the largest score to 1, and set the scores for all other classes to -1.
'symmetriclogit'	$2/(1 + e^{-x}) - 1$

- For a MATLAB function, or a function that you define, enter its function handle.

```
Mdl.ScoreTransform = @function;
```

*function* must accept a matrix of the original scores for each class, and then return a matrix of the same size representing the transformed scores for each class.

Data Types: char | function\_handle

### ResponseName — Response variable name

character array

Response variable name, specified as a character array.

Data Types: char

## Object Functions

edge Classification edge for Gaussian kernel classification model  
 loss Classification loss for Gaussian kernel classification model  
 margin Classification margins for Gaussian kernel classification model  
 predict Predict labels for Gaussian kernel classification model

`resume` Resume training of Gaussian kernel classification model

## Examples

### Train Kernel Classification Model

Train a binary, kernel classification model using support vector machines.

Load the `ionosphere` data. This data has 34 predictors and 351 binary responses for radar returns, either bad ('b') or good ('g').

```
load ionosphere
[n,p] = size(X)
resp = unique(Y)
```

```
n =
    351

p =
    34

resp =
    2x1 cell array

    {'b'}
    {'g'}
```

Train a binary, kernel classification model that can identify whether the radar return is bad ('b') or good ('g'). Determine how well the optimization algorithm fit the model to the data by extracting a fit summary.

```
rng('default') % For reproducibility
[Mdl,FitInfo] = fitckernel(X,Y)
```

```
Mdl =
```

```
ClassificationKernel
  ResponseName: 'Y'
  ClassNames: {'b' 'g'}
  ScoreTransform: 'none'
  KernelScale: 1
  BoxConstraint: 1
  Lambda: 0.0028
  Learner: 'svm'
  Transformation: 'FastFood'
  ExpansionDimension: 2048

FitInfo =

struct with fields:

    Solver: 'LBFGS-fast'
    LossFunction: 'hinge'
    Lambda: 0.0028
    BetaTolerance: 1.0000e-04
    GradientTolerance: 1.0000e-06
    ObjectiveValue: 0.2604
    GradientMagnitude: 0.0028
    RelativeChangeInBeta: 8.2512e-05
    FitTime: 0.2019
    History: []
```

Mdl is a ClassificationKernel model. You can pass Mdl and the training data or new data to the function `loss` to inspect the in-sample classification error. Or, you can pass Mdl and new predictor data to the function `predict` to predict class labels for new observations. You can also pass Mdl and the training data to the function `resume` to continue training.

FitInfo is a structure array containing, among other things, how long the solver took to fit the model to the data (`FitTime`) and history of optimization information for every iteration (`History`). It is good practice to use FitInfo to determine whether optimization-termination measurements are satisfactory.

For better accuracy, you can increase the maximum number of optimization iterations (`'IterationLimit'`) and decrease the tolerance values (`'BetaTolerance'` and

'GradientTolerance') by using the name-value pair arguments. This can improve measures like `ObjectiveValue` and `RelativeChangeInBeta` in `FitInfo`. You can also optimize model parameters such as the kernel scale parameter ('`KernelScale`'), box constraint ('`BoxConstraint`'), and regularization term strength ('`Lambda`') by using the `bayseopt` function.

### Predict Class Labels and Resume Training

Load the `ionosphere` data. This data has 34 predictors and 351 binary responses for radar returns, either bad ('b') or good ('g').

```
load ionosphere
```

Partition the data set into training and test sets. Specify a 20% holdout sample for testing.

```
rng('default') % For reproducibility
Partition = cvpartition(Y,'Holdout',0.20);
trainingInds = training(Partition); % Indices for the training set
XTrain = X(trainingInds,:);
YTrain = Y(trainingInds);
testInds = test(Partition); % Indices for the test set
XTest = X(testInds,:);
YTest = Y(testInds);
```

Train a binary, kernel classification model that can identify whether the radar return is bad ('b') or good ('g').

```
Mdl = fitckernel(XTrain,YTrain,'IterationLimit',5,'Verbose',1);
```

```
=====
| Solver | Pass | Iteration | Objective | Step | Gradient | F
|        |      |           |           |      | magnitude | char
|=====
| LBFGS | 1 | 0 | 1.000000e+00 | 0.000000e+00 | 2.811388e-01 |
| LBFGS | 1 | 1 | 7.585395e-01 | 4.000000e+00 | 3.594306e-01 | 1.
| LBFGS | 1 | 2 | 7.160994e-01 | 1.000000e+00 | 2.028470e-01 | 6.
| LBFGS | 1 | 3 | 6.825272e-01 | 1.000000e+00 | 2.846975e-02 | 2.
| LBFGS | 1 | 4 | 6.699435e-01 | 1.000000e+00 | 1.779359e-02 | 1.
| LBFGS | 1 | 5 | 6.535619e-01 | 1.000000e+00 | 2.669039e-01 | 4.
|=====
```

`Mdl` is a `ClassificationKernel` model.

Predict the test-sample labels, construct a confusion matrix for the test data, and estimate the classification error for the test-sample.

```
label = predict(Mdl,XTest);
ConfusionTest = confusionmat(YTest,label)
L = loss(Mdl,XTest,YTest)
```

```
ConfusionTest =
```

```
    0    25
    0    45
```

```
L =
```

```
    0.3594
```

Mdl misclassifies all bad radar returns as good returns.

Continue training by using `resume`. The function `resume` continues training using the same training options used for training Mdl.

```
UpdatedMdl = resume(Mdl,XTrain,YTrain);
```

```
=====
| Solver | Pass | Iteration | Objective | Step | Gradient | F
|        |      |           |           |      | magnitude | char
|=====|=====|=====|=====|=====|=====|=====
| LBFGS | 1 | 0 | 6.535619e-01 | 0.000000e+00 | 2.669039e-01 |
| LBFGS | 1 | 1 | 6.132547e-01 | 1.000000e+00 | 6.355537e-03 | 1.
| LBFGS | 1 | 2 | 5.938316e-01 | 4.000000e+00 | 3.202847e-02 | 1.
| LBFGS | 1 | 3 | 4.169274e-01 | 1.000000e+00 | 1.530249e-01 | 7.
| LBFGS | 1 | 4 | 3.679212e-01 | 5.000000e-01 | 2.740214e-01 | 2.
| LBFGS | 1 | 5 | 3.332261e-01 | 1.000000e+00 | 1.423488e-02 | 9.
| LBFGS | 1 | 6 | 3.235335e-01 | 1.000000e+00 | 7.117438e-03 | 7.
| LBFGS | 1 | 7 | 3.112331e-01 | 1.000000e+00 | 6.049822e-02 | 1.
| LBFGS | 1 | 8 | 2.972144e-01 | 1.000000e+00 | 7.117438e-03 | 5.
| LBFGS | 1 | 9 | 2.837450e-01 | 1.000000e+00 | 8.185053e-02 | 1.
| LBFGS | 1 | 10 | 2.797642e-01 | 1.000000e+00 | 3.558719e-02 | 5.
| LBFGS | 1 | 11 | 2.771280e-01 | 1.000000e+00 | 2.846975e-02 | 2.
| LBFGS | 1 | 12 | 2.741570e-01 | 1.000000e+00 | 3.914591e-02 | 3.
| LBFGS | 1 | 13 | 2.725701e-01 | 5.000000e-01 | 1.067616e-01 | 8.
| LBFGS | 1 | 14 | 2.667147e-01 | 1.000000e+00 | 3.914591e-02 | 3.
```

	LBFGS		1		15		2.621152e-01		1.000000e+00		7.117438e-03		5.
	LBFGS		1		16		2.601652e-01		1.000000e+00		3.558719e-02		3.
	LBFGS		1		17		2.589052e-01		1.000000e+00		3.202847e-02		3.
	LBFGS		1		18		2.583185e-01		1.000000e+00		7.117438e-03		6.
	LBFGS		1		19		2.556482e-01		1.000000e+00		9.252669e-02		4.
	LBFGS		1		20		2.542643e-01		1.000000e+00		7.117438e-02		4.
=====													
	Solver		Pass		Iteration		Objective		Step		Gradient		F
											magnitude		char
=====													
	LBFGS		1		21		2.532117e-01		1.000000e+00		1.067616e-02		1.
	LBFGS		1		22		2.529890e-01		1.000000e+00		2.135231e-02		1.
	LBFGS		1		23		2.523232e-01		1.000000e+00		3.202847e-02		1.
	LBFGS		1		24		2.506736e-01		1.000000e+00		1.779359e-02		2.
	LBFGS		1		25		2.501995e-01		1.000000e+00		1.779359e-02		2.
	LBFGS		1		26		2.488242e-01		1.000000e+00		3.558719e-03		1.
	LBFGS		1		27		2.485295e-01		5.000000e-01		3.202847e-02		1.
	LBFGS		1		28		2.482244e-01		1.000000e+00		4.270463e-02		8.
	LBFGS		1		29		2.479714e-01		1.000000e+00		3.558719e-03		7.
	LBFGS		1		30		2.477316e-01		1.000000e+00		3.202847e-02		3.
	LBFGS		1		31		2.476178e-01		2.500000e-01		3.202847e-02		5.
	LBFGS		1		32		2.474874e-01		1.000000e+00		1.779359e-02		3.
	LBFGS		1		33		2.473980e-01		1.000000e+00		7.117438e-03		2.
	LBFGS		1		34		2.472935e-01		1.000000e+00		3.558719e-03		2.
	LBFGS		1		35		2.471418e-01		1.000000e+00		3.558719e-03		1.
	LBFGS		1		36		2.469862e-01		1.000000e+00		2.846975e-02		7.
	LBFGS		1		37		2.469598e-01		1.000000e+00		2.135231e-02		6.
	LBFGS		1		38		2.466941e-01		1.000000e+00		3.558719e-02		4.
	LBFGS		1		39		2.466660e-01		5.000000e-01		1.423488e-02		2.
	LBFGS		1		40		2.465605e-01		1.000000e+00		3.558719e-03		4.
=====													
	Solver		Pass		Iteration		Objective		Step		Gradient		F
											magnitude		char
=====													
	LBFGS		1		41		2.465362e-01		1.000000e+00		1.423488e-02		5.
	LBFGS		1		42		2.463528e-01		1.000000e+00		3.558719e-03		2.
	LBFGS		1		43		2.463207e-01		1.000000e+00		1.511170e-03		3.
	LBFGS		1		44		2.462585e-01		5.000000e-01		7.117438e-02		2.
	LBFGS		1		45		2.461742e-01		1.000000e+00		7.117438e-03		2.
	LBFGS		1		46		2.461434e-01		1.000000e+00		3.202847e-02		3.
	LBFGS		1		47		2.461115e-01		1.000000e+00		7.117438e-03		1.
	LBFGS		1		48		2.460814e-01		1.000000e+00		1.067616e-02		1.
	LBFGS		1		49		2.460533e-01		5.000000e-01		1.423488e-02		1.
	LBFGS		1		50		2.460111e-01		1.000000e+00		1.423488e-02		4.



	LBFGS		1		51		2.459414e-01		1.000000e+00		1.067616e-02		3.
	LBFGS		1		52		2.458809e-01		1.000000e+00		1.423488e-02		1.
	LBFGS		1		53		2.458479e-01		1.000000e+00		1.067616e-02		1.
	LBFGS		1		54		2.458146e-01		1.000000e+00		1.455008e-03		1.
	LBFGS		1		55		2.457878e-01		1.000000e+00		7.117438e-03		1.
	LBFGS		1		56		2.457519e-01		1.000000e+00		2.491103e-02		1.
	LBFGS		1		57		2.457420e-01		1.000000e+00		7.473310e-02		9.
	LBFGS		1		58		2.457212e-01		1.000000e+00		3.558719e-03		3.
	LBFGS		1		59		2.457089e-01		1.000000e+00		4.270463e-02		6.
	LBFGS		1		60		2.457047e-01		5.000000e-01		1.423488e-02		3.

Solver	Pass	Iteration	Objective	Step	Gradient magnitude	F char							
	LBFGS		1		61		2.456991e-01		1.000000e+00		1.423488e-02		5.
	LBFGS		1		62		2.456898e-01		1.000000e+00		1.779359e-02		4.
	LBFGS		1		63		2.456792e-01		1.000000e+00		1.779359e-02		5.
	LBFGS		1		64		2.456603e-01		1.000000e+00		1.403782e-03		5.
	LBFGS		1		65		2.456482e-01		1.000000e+00		3.558719e-03		6.
	LBFGS		1		66		2.456358e-01		1.000000e+00		1.476262e-03		1.
	LBFGS		1		67		2.456124e-01		1.000000e+00		3.558719e-03		8.
	LBFGS		1		68		2.455980e-01		1.000000e+00		1.067616e-02		9.
	LBFGS		1		69		2.455780e-01		1.000000e+00		1.067616e-02		5.
	LBFGS		1		70		2.455633e-01		1.000000e+00		3.558719e-03		1.
	LBFGS		1		71		2.455449e-01		1.000000e+00		1.423488e-02		7.
	LBFGS		1		72		2.455261e-01		1.000000e+00		3.558719e-02		1.
	LBFGS		1		73		2.455142e-01		1.000000e+00		1.067616e-02		1.
	LBFGS		1		74		2.455075e-01		1.000000e+00		1.779359e-02		1.
	LBFGS		1		75		2.455008e-01		1.000000e+00		3.914591e-02		1.
	LBFGS		1		76		2.454943e-01		1.000000e+00		2.491103e-02		3.
	LBFGS		1		77		2.454918e-01		5.000000e-01		3.202847e-02		9.
	LBFGS		1		78		2.454870e-01		1.000000e+00		1.779359e-02		4.
	LBFGS		1		79		2.454865e-01		5.000000e-01		3.558719e-03		7.
	LBFGS		1		80		2.454775e-01		1.000000e+00		5.693950e-02		8.

Solver	Pass	Iteration	Objective	Step	Gradient magnitude	F char							
	LBFGS		1		81		2.454686e-01		1.000000e+00		1.183730e-03		1.
	LBFGS		1		82		2.454612e-01		1.000000e+00		2.135231e-02		1.
	LBFGS		1		83		2.454506e-01		1.000000e+00		3.558719e-03		6.
	LBFGS		1		84		2.454436e-01		1.000000e+00		1.423488e-02		1.
	LBFGS		1		85		2.454378e-01		1.000000e+00		1.423488e-02		3.
	LBFGS		1		86		2.454249e-01		1.000000e+00		1.423488e-02		8.

	LBFGS		1		87		2.454101e-01		1.000000e+00		1.067616e-02		3.
	LBFGS		1		88		2.453963e-01		1.000000e+00		1.779359e-02		5.
	LBFGS		1		89		2.453866e-01		1.000000e+00		1.067616e-02		1.
	LBFGS		1		90		2.453821e-01		1.000000e+00		7.117438e-03		2.
	LBFGS		1		91		2.453790e-01		5.000000e-01		6.761566e-02		8.
	LBFGS		1		92		2.453603e-01		1.000000e+00		2.135231e-02		1.
	LBFGS		1		93		2.453540e-01		1.000000e+00		2.135231e-02		2.
	LBFGS		1		94		2.453482e-01		1.000000e+00		1.779359e-02		1.
	LBFGS		1		95		2.453461e-01		1.000000e+00		1.779359e-02		1.
	LBFGS		1		96		2.453371e-01		1.000000e+00		3.558719e-02		2.
	LBFGS		1		97		2.453305e-01		1.000000e+00		4.270463e-02		7.
	LBFGS		1		98		2.453283e-01		2.500000e-01		2.135231e-02		3.
	LBFGS		1		99		2.453246e-01		1.000000e+00		3.558719e-03		3.
	LBFGS		1		100		2.453214e-01		1.000000e+00		3.202847e-02		1.
=====													
	Solver		Pass		Iteration		Objective		Step		Gradient		F
											magnitude		char
=====													
	LBFGS		1		101		2.453168e-01		1.000000e+00		1.067616e-02		3.
	LBFGS		1		102		2.453155e-01		5.000000e-01		4.626335e-02		3.
	LBFGS		1		103		2.453136e-01		1.000000e+00		1.779359e-02		2.
	LBFGS		1		104		2.453119e-01		1.000000e+00		3.202847e-02		4.
	LBFGS		1		105		2.453093e-01		1.000000e+00		1.423488e-02		2.
	LBFGS		1		106		2.453090e-01		1.000000e+00		2.846975e-02		1.
	LBFGS		1		107		2.453048e-01		1.000000e+00		1.423488e-02		3.
	LBFGS		1		108		2.453040e-01		1.000000e+00		3.558719e-02		1.
	LBFGS		1		109		2.452977e-01		1.000000e+00		1.423488e-02		8.
	LBFGS		1		110		2.452934e-01		1.000000e+00		2.135231e-02		5.
	LBFGS		1		111		2.452886e-01		1.000000e+00		1.779359e-02		3.
	LBFGS		1		112		2.452854e-01		1.000000e+00		1.067616e-02		2.
	LBFGS		1		113		2.452836e-01		1.000000e+00		1.067616e-02		1.
	LBFGS		1		114		2.452817e-01		1.000000e+00		7.117438e-03		4.
	LBFGS		1		115		2.452741e-01		1.000000e+00		1.779359e-02		9.
	LBFGS		1		116		2.452691e-01		1.000000e+00		2.135231e-02		9.
=====													

Predict the test-sample labels, construct a confusion matrix for the test data, and estimate the classification error for the test-sample.

```
UpdatedLabel = predict(UpdatedMdl,XTest);
UpdatedConfusionTest = confusionmat(YTest,UpdatedLabel)
UpdatedL = loss(UpdatedMdl,XTest,YTest)
```

```
UpdatedConfusionTest =
```

```
23     2
7      38
```

```
UpdatedL =
```

```
0.1284
```

The classification error decreased after updating the classification model with more iterations.

## See Also

`ClassificationLinear` | `fitckernel` | `fitclinear`

**Introduced in R2017b**

## edge

Classification edge for Gaussian kernel classification model

### Syntax

```
e = edge(Mdl,X,Y)
e = edge(Mdl,X,Y,'Weights',weights)
```

### Description

`e = edge(Mdl,X,Y)` returns the classification edge on page 32-7114 for the binary, Gaussian kernel classification model `Mdl` using predictor data in `X` and corresponding class labels in `Y`.

`e = edge(Mdl,X,Y,'Weights',weights)` returns the weighted classification edges for the observation weights supplied in `weights`.

### Examples

#### Estimate Test-Sample Edge

Load the `ionosphere` data. This data has 34 predictors and 351 binary responses for radar returns, either bad ('b') or good ('g').

```
load ionosphere
```

Partition the data set into training and test sets. Specify a 15% holdout sample for testing.

```
rng('default') % For reproducibility
Partition = cvpartition(Y,'Holdout',0.15);
trainingInds = training(Partition); % Indices for the training set
testInds = test(Partition); % Indices for the test set
```

Train a binary, kernel classification model using the training set.

```
Mdl = fitckernel(X(trainingInds,:),Y(trainingInds));
```

Estimate the training- and test-sample edge.

```
eTrain = edge(Mdl,X(trainingInds,:),Y(trainingInds))  
eTest = edge(Mdl,X(testInds,:),Y(testInds))
```

```
eTrain =  
  
    2.1703  
  
eTest =  
  
    1.5643
```

## Feature Selection Using Test-Sample Edges

One way to perform feature selection is to compare test-sample edges from multiple models. Based solely on this criterion, the classifier with the highest edge is the best classifier.

Load the `ionosphere` data. This data has 34 predictors and 351 binary responses for radar returns, either bad ('b') or good ('g').

```
load ionosphere
```

Partition the data set into training and test sets. Specify a 15% holdout sample for testing.

```
rng('default') % For reproducibility  
Partition = cvpartition(Y,'Holdout',0.15);  
trainingInds = training(Partition); % Indices for the training set  
XTrain = X(trainingInds,:);  
YTrain = Y(trainingInds);  
testInds = test(Partition); % Indices for the test set  
XTest = X(testInds,:);  
YTest = Y(testInds);
```

Randomly choose half of the predictor variables.

```
p = size(X,2); % Number of predictors
idxPart = randsample(p,ceil(0.5*p));
```

Train two binary, kernel classification models: one that uses the all of the predictors and one that uses half of the predictors.

```
Mdl = fitckernel(XTrain,YTrain);
PMdl = fitckernel(XTrain(:,idxPart),YTrain);
```

Mdl and PMdl are `ClassificationKernel` models.

Estimate the test sample edge for each classifier.

```
fullEdge = edge(Mdl,XTest,YTest)
partEdge = edge(PMdl,XTest(:,idxPart),YTest)
```

```
fullEdge =
```

```
1.6335
```

```
partEdge =
```

```
2.0205
```

Based on the test-sample edges, the classifier that uses half of the predictors is the better model.

## Input Arguments

### **Mdl** — Binary, kernel classification object

`ClassificationKernel` object

Binary, kernel classification model, specified as a `ClassificationKernel` model object. You can create a `ClassificationKernel` model object using `fitckernel`.

### **x** — Predictor data

$n$ -by- $p$  numeric matrix

Predictor data, specified as an  $n$ -by- $p$  numeric matrix, where  $n$  is the number of observations and  $p$  is the number of predictors used to train `Mdl`.

The length of `Y` and the number of observations in `X` must be equal.

Data Types: `single` | `double`

### **`Y` — Class labels**

categorical array | character array | logical vector | vector of numeric values | cell array of character vectors

Class labels, specified as a categorical or character array, logical or numeric vector, or cell array of character vectors.

- The data type of `Y` must be the same `Mdl.ClassNames`.
- The distinct classes in `Y` must be a subset of `Mdl.ClassNames`.
- If `Y` is a character array, then each element must correspond to one row of the array.
- The length of `Y` and the number of observations in `X` must be equal.

Data Types: `categorical` | `cell` | `char` | `double` | `logical` | `single`

### **`weights` — Observation weights**

positive numeric vector

Observation weights, specified as a positive numeric vector of length  $n$ , where  $n$  is the number of observations in `X`. If you supply `weights`, `edge` computes the weighted classification edge on page 32-7114.

The default value is `ones(n,1)`.

`edge` normalizes `weights` to sum up to the value of the prior probability in the respective class.

Data Types: `double` | `single`

## Output Arguments

### **`e` — Classification edge**

numeric scalar

Classification edge on page 32-7114, returned as a numeric scalar.

## Definitions

### Classification Edge

The classification edge is the weighted mean of the classification margins.

One way to choose among multiple classifiers, e.g., to perform feature selection, is to choose the classifier that yields the highest edge.

### Classification Margin

The classification margins for binary classification are, for each observation, the difference between the classification score for the true class and the classification score for the false class.

The software defines the classification margin for binary classification as

$$m = 2yf(x).$$

$x$  is an observation. If the true label of  $x$  is the positive class, then  $y$  is 1, and  $-1$  otherwise.  $f(x)$  is the positive-class classification score for the observation  $x$ . The literature commonly defines the margin as  $m = yf(x)$ .

If the margins are on the same scale, then they serve as a classification confidence measure, i.e., among multiple classifiers, those that yield larger margins are better.

### Classification Score

For linear classification models, the raw classification score for classifying the observation  $x$ , a row vector, into the positive class is defined by

$$f(x) = T(x)\beta + b.$$

$T(\cdot)$  is a transformation of an observation for feature expansion,  $\beta$  is the estimated column vector of coefficients (the model property `Beta`) and  $b$  is the estimated, scalar bias (the model property `Bias`).



The raw classification score for classifying  $x$  into the negative class is  $-f(x)$ . The software classifies observations into the class that yields the positive score.

If the linear classification model consists of logistic regression learners, then the software applies the 'logit' score transformation to the raw classification scores (see `ScoreTransform`).

## Extended Capabilities

### Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays” (MATLAB).

### See Also

`ClassificationKernel` | `fitckernel` | `margin` | `predict`

**Introduced in R2017b**

## loss

Classification loss for Gaussian kernel classification model

### Syntax

```
L = loss(Mdl,X,Y)
L = loss(Mdl,X,Y,Name,Value)
```

### Description

`L = loss(Mdl,X,Y)` returns the classification loss on page 32-7121 for the binary, Gaussian kernel classification model `Mdl` using predictor data in `X` and corresponding class labels in `Y`.

`L = loss(Mdl,X,Y,Name,Value)` specifies options using `Name,Value` pair arguments in addition to input arguments in previous syntax. For example, you can specify a classification loss function and observation weights. Then, `loss` returns the weighted classification loss using the specified loss function.

## Examples

### Estimate Test-Sample Classification Loss

Load the `ionosphere` data. This data has 34 predictors and 351 binary responses for radar returns, either bad ('b') or good ('g').

```
load ionosphere
```

Partition the data set into training and test sets. Specify a 15% holdout sample for testing.

```
rng('default') % For reproducibility
Partition = cvpartition(Y,'Holdout',0.15);
```

```
trainingInds = training(Partition); % Indices for the training set
testInds = test(Partition); % Indices for the test set
```

Train a binary, kernel classification model using the training set.

```
Mdl = fitckernel(X(trainingInds,:),Y(trainingInds));
```

Estimate the training- and test-sample classification error.

```
ceTrain = loss(Mdl,X(trainingInds,:),Y(trainingInds))
ceTest = loss(Mdl,X(testInds,:),Y(testInds))
```

```
ceTrain =
```

```
    0.0067
```

```
ceTest =
```

```
    0.1140
```

### Specify Custom Classification Loss

Load the `ionosphere` data. This data has 34 predictors and 351 binary responses for radar returns, either bad ('b') or good ('g').

```
load ionosphere
```

Partition the data set into training and test sets. Specify a 15% holdout sample for testing.

```
rng('default') % For reproducibility
Partition = cvpartition(Y,'Holdout',0.15);
trainingInds = training(Partition); % Indices for the training set
testInds = test(Partition); % Indices for the test set
```

Train a binary, kernel classification model using the training set.

```
Mdl = fitckernel(X(trainingInds,:),Y(trainingInds));
```

Create an anonymous function that measures linear loss, that is,

$$L = \frac{\sum_j -w_j y_j f_j}{\sum_j w_j}.$$

$w_j$  is the weight for observation  $j$ ,  $y_j$  is response  $j$  (-1 for the negative class, and 1 otherwise), and  $f_j$  is the raw classification score of observation  $j$ . Custom loss functions must be written in a particular form. For rules on writing a custom loss function, see the `LossFun` name-value pair argument.

```
linearloss = @(C,S,W,Cost) sum(-W.*sum(S.*C,2))/sum(W);
```

Estimate the training- and test-sample classification loss using the linear loss function.

```
ceTrain = loss(Mdl,X(trainingInds,:),Y(trainingInds),'LossFun',linearloss)
ceTest = loss(Mdl,X(testInds,:),Y(testInds),'LossFun',linearloss)
```

```
ceTrain =
    -1.0851
```

```
ceTest =
    -0.7821
```

## Input Arguments

### **mdl** — Binary, kernel classification object

`ClassificationKernel` object

Binary, kernel classification model, specified as a `ClassificationKernel` model object. You can create a `ClassificationKernel` model object using `fitckernel`.

### **x** — Predictor data

$n$ -by- $p$  numeric matrix

Predictor data, specified as an  $n$ -by- $p$  numeric matrix, where  $n$  is the number of observations and  $p$  is the number of predictors used to train `mdl`.

The length of `Y` and the number of observations in `X` must be equal.

Data Types: `single` | `double`

### **Y — Class labels**

categorical array | character array | logical vector | vector of numeric values | cell array of character vectors

Class labels, specified as a categorical or character array, logical or numeric vector, or cell array of character vectors.

- The data type of `Y` must be the same `Mdl.ClassNames`.
- The distinct classes in `Y` must be a subset of `Mdl.ClassNames`.
- If `Y` is a character array, then each element must correspond to one row of the array.
- The length of `Y` and the number of observations in `X` must be equal.

Data Types: `categorical` | `cell` | `char` | `double` | `logical` | `single`

## **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( `' '` ). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `L = loss(Mdl,X,Y,'LossFun','quadratic','Weights',weights)` returns the weighted classification loss using the quadratic loss function.

### **LossFun — Loss function**

`'classiferror'` (default) | `'binodeviance'` | `'exponential'` | `'hinge'` | `'logit'` | `'mincost'` | `'quadratic'` | function handle

Loss function, specified as the comma-separated pair consisting of `'LossFun'` and a built-in, loss-function name or function handle.

- The following lists available loss functions. Specify one using its corresponding character vector.

<b>Value</b>	<b>Description</b>
<code>'binodeviance'</code>	Binomial deviance
<code>'classiferror'</code>	Classification error

Value	Description
'exponential'	Exponential
'hinge'	Hinge
'logit'	Logistic
'mincost'	Minimal expected misclassification cost (for classification scores that are posterior probabilities)
'quadratic'	Quadratic

'mincost' is appropriate for classification scores that are posterior probabilities. For linear classification models, logistic regression learners return posterior probabilities as classification scores by default, but SVM learners do not (see `predict`).

- Specify your own function using function handle notation.

Let  $n$  be the number of observations in  $X$  and  $K$  be the number of distinct classes (`numel(Mdl.ClassNames)`,  $Mdl$  is the input model). Your function must have this signature

```
lossvalue = lossfun(C,S,W,Cost)
```

where:

- The output argument `lossvalue` is a scalar.
- You choose the function name (`lossfun`).
- $C$  is an  $n$ -by- $K$  logical matrix with rows indicating which class the corresponding observation belongs. The column order corresponds to the class order in `Mdl.ClassNames`.

Construct  $C$  by setting  $C(p, q) = 1$  if observation  $p$  is in class  $q$ , for each row. Set all other elements of row  $p$  to 0.

- $S$  is an  $n$ -by- $K$  numeric matrix of classification scores. The column order corresponds to the class order in `Mdl.ClassNames`.  $S$  is a matrix of classification scores, similar to the output of `predict`.
- $W$  is an  $n$ -by-1 numeric vector of observation weights. If you pass  $W$ , the software normalizes them to sum to 1.
- $Cost$  is a  $K$ -by- $K$  numeric matrix of misclassification costs. For example, `Cost = ones(K) - eye(K)` specifies a cost of 0 for correct classification, and 1 for misclassification.

Specify your function using `'LossFun', @lossfun`.

Data Types: `char` | `function_handle`

### **weights** — Observation weights

positive numeric vector

Observation weights, specified as the comma-separated pair consisting of `'Weights'` and a positive numeric vector of length  $n$ , where  $n$  is the number of observations in `X`. If you supply weights, `loss` computes the weighted classification loss on page 32-7121.

The default value is `ones(n,1)`.

`loss` normalizes `weights` to sum up to the value of the prior probability in the respective class.

Data Types: `double` | `single`

## Output Arguments

### **L** — Classification loss

numeric scalar

Classification loss on page 32-7121, returned as a numeric scalar. The interpretation of `L` depends on `Weights` and `LossFun`.

## Definitions

### Classification Loss

Classification loss functions measure the predictive inaccuracy of classification models. When comparing the same type of loss among many models, lower loss indicates a better predictive model.

Suppose that:

- $L$  is the weighted average classification loss.

- $n$  is the sample size.
- $y_j$  is the observed class label. The software codes it as  $-1$  or  $1$  indicating the negative or positive class, respectively.
- $f(X_j)$  is the raw classification score for the transformed observation (row)  $j$  of the predictor data  $X$  using feature expansion.
- $m_j = y_j f(X_j)$  is the classification score for classifying observation  $j$  into the class corresponding to  $y_j$ . Positive values of  $m_j$  indicate correct classification and do not contribute much to the average loss. Negative values of  $m_j$  indicate incorrect classification and contribute to the average loss.
- The weight for observation  $j$  is  $w_j$ . The software normalizes the observation weights so that they sum to the corresponding prior class probability. The software also normalizes the prior probabilities so they sum to 1. Therefore,

$$\sum_{j=1}^n w_j = 1.$$

The supported loss functions are:

- Binomial deviance, specified using `'LossFun', 'binodeviance'`. Its equation is

$$L = \sum_{j=1}^n w_j \log \{1 + \exp[-2m_j]\}.$$

- Exponential loss, specified using `'LossFun', 'exponential'`. Its equation is

$$L = \sum_{j=1}^n w_j \exp(-m_j).$$

- Classification error, specified using `'LossFun', 'classiferror'`. It is the weighted fraction of misclassified observations, with equation

$$L = \sum_{j=1}^n w_j I\{\hat{y}_j \neq y_j\}.$$

$\hat{y}_j$  is the class label corresponding to the class with the maximal posterior probability.  $I\{x\}$  is the indicator function.

- Hinge loss, specified using `'LossFun', 'hinge'`. Its equation is



$$L = \sum_{j=1}^n w_j \max\{0, 1 - m_j\}.$$

- Logit loss, specified using 'LossFun', 'logit'. Its equation is

$$L = \sum_{j=1}^n w_j \log(1 + \exp(-m_j)).$$

- Minimal cost, specified using 'LossFun', 'mincost'. The software computes the weighted minimal cost using this procedure for observations  $j = 1, \dots, n$ :

- 1 Estimate the 1-by- $K$  vector of expected classification costs for observation  $j$

$$\gamma_j = f(X_j)'C.$$

$f(X_j)$  is the column vector of class posterior probabilities.  $C$  is the cost matrix the input model stores in the property `Cost`.

- 2 For observation  $j$ , predict the class label corresponding to the minimum, expected classification cost:

$$\hat{y}_j = \min_{j=1, \dots, K} (\gamma_j).$$

- 3 Using  $C$ , identify the cost incurred ( $c_j$ ) for making the prediction.

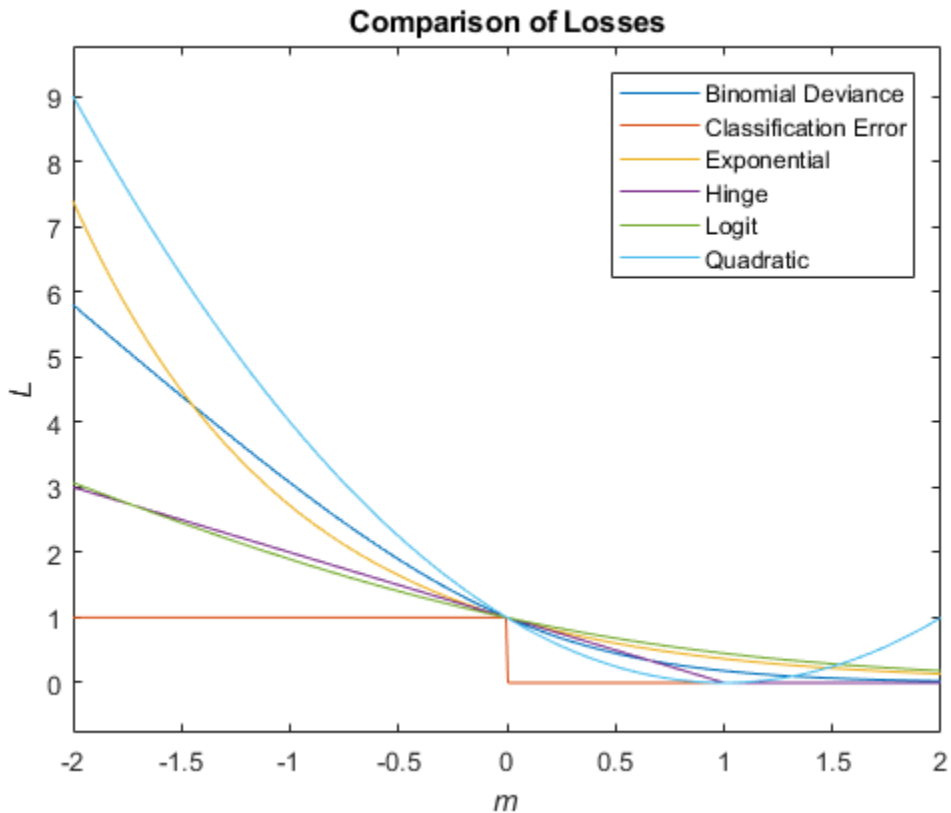
The weighted, average, minimum cost loss is

$$L = \sum_{j=1}^n w_j c_j.$$

- Quadratic loss, specified using 'LossFun', 'quadratic'. Its equation is

$$L = \sum_{j=1}^n w_j (1 - m_j)^2.$$

This figure compares some of the loss functions for one observation over  $m$  (some functions are normalized to pass through  $[0,1]$ ).



## Extended Capabilities

### Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays” (MATLAB).

## See Also

`ClassificationKernel` | `fitckernel` | `predict`

**Introduced in R2017b**

## margin

Classification margins for Gaussian kernel classification model

### Syntax

```
m = margin(Mdl,X,Y)
```

### Description

`m = margin(Mdl,X,Y)` returns the classification margins on page 32-7131 for the binary, Gaussian kernel classification model `Mdl` using predictor data in `X` and corresponding class labels in `Y`.

### Examples

#### Estimate Test-Sample Margins

Load the `ionosphere` data. This data has 34 predictors and 351 binary responses for radar returns, either bad ('b') or good ('g').

```
load ionosphere
```

Partition the data set into training and test sets. Specify a 30% holdout sample for testing.

```
rng('default') % For reproducibility
Partition = cvpartition(Y,'Holdout',0.30);
trainingInds = training(Partition); % Indices for the training set
testInds = test(Partition); % Indices for the test set
```

Train a binary, kernel classification model using the training set.

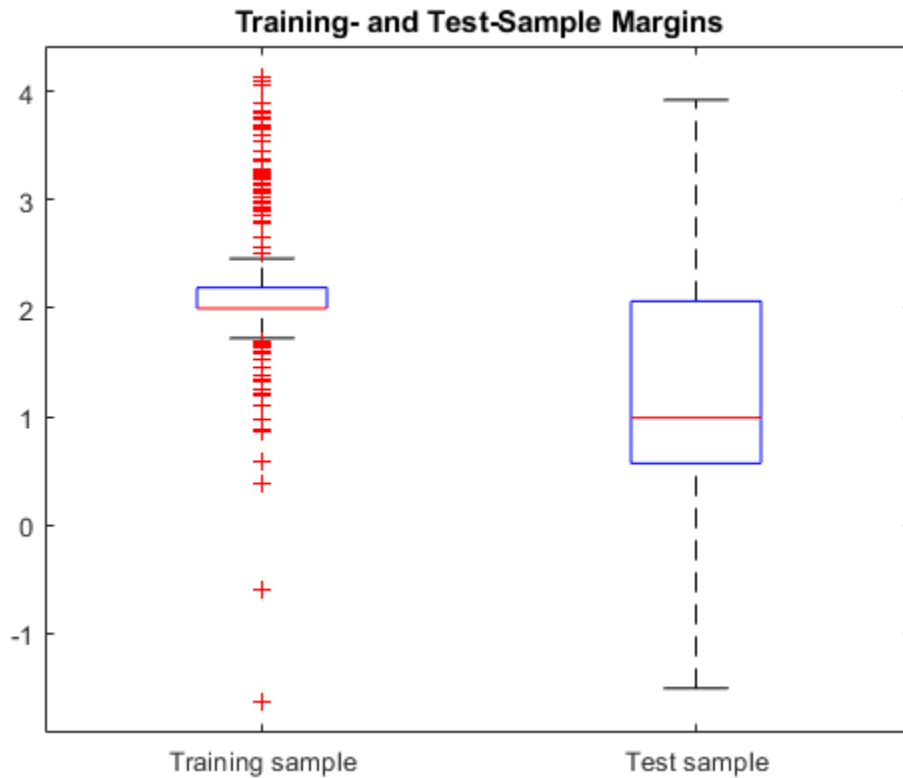
```
Mdl = fitckernel(X(trainingInds,:),Y(trainingInds));
```

Estimate the training- and test-sample margins.

```
mTrain = margin(Mdl,X(trainingInds,:),Y(trainingInds));  
mTest = margin(Mdl,X(testInds,:),Y(testInds));
```

Plot both sets of margins using box plots.

```
boxplot([mTrain; mTest],[zeros(size(mTrain,1),1); ones(size(mTest,1),1)], ...  
        'Labels',{'Training sample','Test sample'});  
title('Training- and Test-Sample Margins')
```



The margin distribution of the training-sample is situated higher than the margin distribution of the test-sample.

## Feature Selection Using Test-Sample Margins

One way to perform feature selection is to compare test-sample margins from multiple models. Based solely on this criterion, the classifier with the larger margins is the better classifier.

Load the `ionosphere` data. This data has 34 predictors and 351 binary responses for radar returns, either bad ('b') or good ('g').

```
load ionosphere
```

Partition the data set into training and test sets. Specify a 15% holdout sample for testing.

```
rng('default') % For reproducibility
Partition = cvpartition(Y, 'Holdout', 0.15);
trainingInds = training(Partition); % Indices for the training set
XTrain = X(trainingInds, :);
YTrain = Y(trainingInds);
testInds = test(Partition); % Indices for the test set
XTest = X(testInds, :);
YTest = Y(testInds);
```

Randomly choose 10% of the predictor variables.

```
p = size(X, 2); % Number of predictors
idxPart = randsample(p, ceil(0.1*p));
```

Train two binary, kernel classification models: one that uses the all of the predictors and one that uses the random 10%.

```
Mdl = fitckernel(XTrain, YTrain);
PMdl = fitckernel(XTrain(:, idxPart), YTrain);
```

`Mdl` and `PMdl` are `ClassificationKernel` models.

Estimate the test sample margins for each classifier.

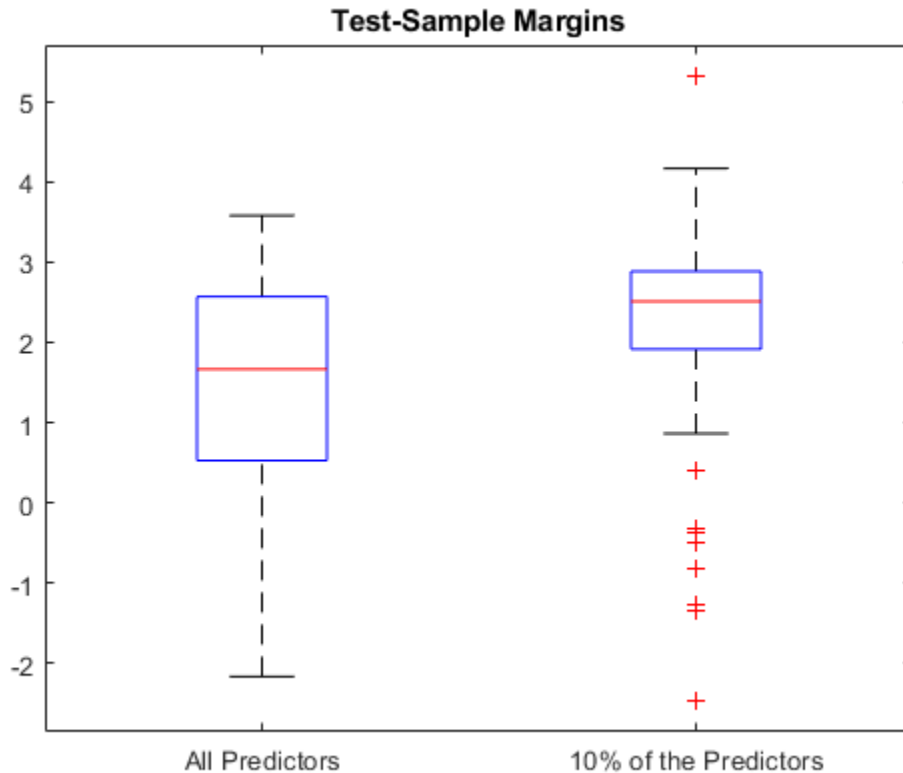
```
fullMargins = margin(Mdl, XTest, YTest);
partMargins = margin(PMdl, XTest(:, idxPart), YTest);
```

Plot the distribution of the margins sets using box plots.

```

boxplot([fullMargins partMargins], 'Labels', ...
        {'All Predictors', '10% of the Predictors'});
title('Test-Sample Margins')

```



The margin distribution of  $PM_{d1}$  is situated higher than the margin distribution of  $M_{d1}$ .

## Input Arguments

**`md1`** — Binary, kernel classification object  
 ClassificationKernel object

Binary, kernel classification model, specified as a `ClassificationKernel` model object. You can create a `ClassificationKernel` model object using `fitckernel`.

**x — Predictor data**

*n*-by-*p* numeric matrix

Predictor data, specified as an *n*-by-*p* numeric matrix, where *n* is the number of observations and *p* is the number of predictors used to train `Mdl`.

The length of `Y` and the number of observations in `X` must be equal.

Data Types: `single` | `double`

**y — Class labels**

categorical array | character array | logical vector | vector of numeric values | cell array of character vectors

Class labels, specified as a categorical or character array, logical or numeric vector, or cell array of character vectors.

- The data type of `Y` must be the same `Mdl.ClassNames`.
- The distinct classes in `Y` must be a subset of `Mdl.ClassNames`.
- If `Y` is a character array, then each element must correspond to one row of the array.
- The length of `Y` and the number of observations in `X` must be equal.

Data Types: `categorical` | `cell` | `char` | `double` | `logical` | `single`

## Output Arguments

**m — Classification margins**

numeric column vector

Classification margins on page 32-7131, returned as an *n*-by-1 numeric column vector, where *n* is the number of observations in `X`



## Definitions

### Classification Margin

The classification margins for binary classification are, for each observation, the difference between the classification score for the true class and the classification score for the false class.

The software defines the classification margin for binary classification as

$$m = 2yf(x).$$

$x$  is an observation. If the true label of  $x$  is the positive class, then  $y$  is 1, and  $-1$  otherwise.  $f(x)$  is the positive-class classification score for the observation  $x$ . The literature commonly defines the margin as  $m = yf(x)$ .

If the margins are on the same scale, then they serve as a classification confidence measure, i.e., among multiple classifiers, those that yield larger margins are better.

### Classification Score

For linear classification models, the raw classification score for classifying the observation  $x$ , a row vector, into the positive class is defined by

$$f(x) = T(x)\beta + b.$$

$T(\cdot)$  is a transformation of an observation for feature expansion,  $\beta$  is the estimated column vector of coefficients (the model property `Beta`) and  $b$  is the estimated, scalar bias (the model property `Bias`).

The raw classification score for classifying  $x$  into the negative class is  $-f(x)$ . The software classifies observations into the class that yields the positive score.

If the linear classification model consists of logistic regression learners, then the software applies the 'logit' score transformation to the raw classification scores (see `ScoreTransform`).

## Extended Capabilities

### Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays” (MATLAB).

### See Also

`ClassificationKernel` | `edge` | `fitckernel` | `predict`

**Introduced in R2017b**

# predict

Predict labels for Gaussian kernel classification model

## Syntax

```
Label = predict(Mdl,X)
[Label,Score] = predict(Mdl,X)
```

## Description

`Label = predict(Mdl,X)` returns predicted class labels for each observation in the predictor data `X` based on the trained, binary, Gaussian kernel classification model `Mdl`.

`[Label,Score] = predict(Mdl,X)` also returns classification scores on page 32-7138 for both classes.

## Examples

### Predict Training-Sample Labels

Predict the training-sample labels using a trained binary kernel classification model and display the confusion matrix for the resulting classification.

Load the `ionosphere` data. This data has 34 predictors and 351 binary responses for radar returns, either bad ('b') or good ('g').

```
load ionosphere
```

Train a binary, kernel classification model that can identify whether the radar return is bad ('b') or good ('g').

```
rng('default') % For reproducibility
Mdl = fitckernel(X,Y);
```

Mdl is a ClassificationKernel model.

Predict the training-sample, or resubstitution, labels.

```
label = predict(Mdl,X);
```

Construct a confusion matrix.

```
ConfusionTrain = confusionmat(Y,label)
```

```
ConfusionTrain =
```

```
    224     1
     1    125
```

The model misclassifies one radar return for each class.

### Predict Test-Sample Labels

Predict the test-sample labels using a trained binary kernel classification model and display the confusion matrix for the resulting classification.

Load the `ionosphere` data. This data has 34 predictors and 351 binary responses for radar returns, either bad ('b') or good ('g').

```
load ionosphere
```

Partition the data set into training and test sets. Specify a 15% holdout sample for testing.

```
rng('default') % For reproducibility
Partition = cvpartition(Y,'Holdout',0.15);
trainingInds = training(Partition); % Indices for the training set
testInds = test(Partition); % Indices for the test set
```

Train a binary, kernel classification model using the training set. It is good practice to define the class order.

```
Mdl = fitkernel(X(trainingInds,:),Y(trainingInds),'ClassNames',{'b','g'});
```

Predict the training- and test-sample labels.

```
labelTrain = predict(Mdl,X(trainingInds,:));
labelTest = predict(Mdl,X(testInds,:));
```

Construct a confusion matrix for the training data.

```
ConfusionTrain = confusionmat(Y(trainingInds),labelTrain)
```

```
ConfusionTrain =
```

```
    190     1
     1    107
```

The model misclassifies only one radar return for each class.

Construct a confusion matrix for the test data.

```
ConfusionTest = confusionmat(Y(testInds),labelTest)
```

```
ConfusionTest =
```

```
    17     1
     5    29
```

The model misclassifies one bad radar return as being a good return, and five good radar returns as being bad returns.

### Estimate Posterior Class Probabilities

Estimate test-sample, posterior class probabilities, and determine the quality of the model by plotting a ROC curve. Kernel classification models return posterior probabilities for logistic regression learners only.

Load the `ionosphere` data. This data has 34 predictors and 351 binary responses for radar returns, either bad ('b') or good ('g').

```
load ionosphere
```

Randomly partition the data into training and test sets by specifying a 30% holdout sample. Identify the training-set and test-set indices.

```
rng('default') % For reproducibility
Partition = cvpartition(Y, 'Holdout', 0.30);
trainingInds = training(Partition); % Indices for the training set
testInds = test(Partition); % Indices for the test set
```

Train a binary, kernel classification model. Fit logistic regression learners.

```
Mdl = fitckernel(X(trainingInds,:), Y(trainingInds), ...
    'ClassNames', {'b', 'g'}, 'Learner', 'logistic');
```

Predict the test-sample posterior class probabilities.

```
[~, posterior] = predict(Mdl, X(testInds,:));
```

Because there is one regularization strength in `Mdl`, the output `posterior` is a matrix with 2 columns and rows equal to the number of test-set observations. Column `i` contains posterior probabilities of `Mdl.ClassNames(i)` given a particular observation.

Obtain false and true positive rates, and estimate the area under the curve (AUC). Specify that the second class is the positive class.

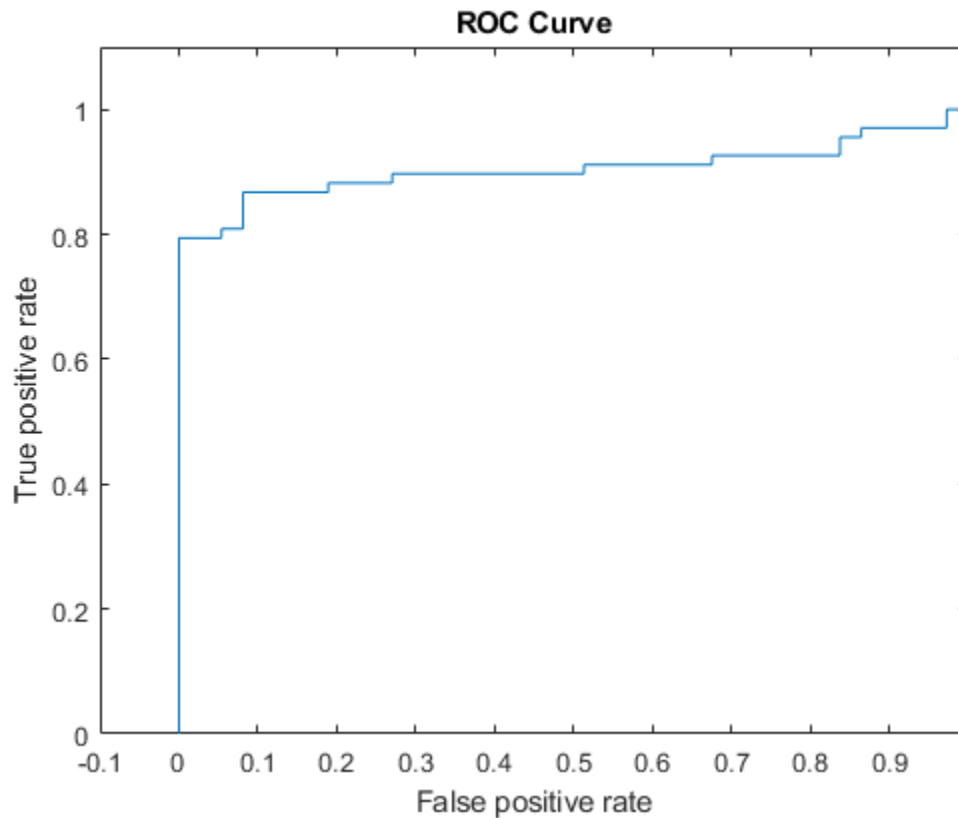
```
[fpr, tpr, ~, auc] = perfcurve(Y(testInds), posterior(:, 2), Mdl.ClassNames(2));
auc
```

```
auc =
    0.9042
```

The AUC is close to 1, which indicates that the model predicts labels well.

Plot an receiver operating characteristic (ROC) curve.

```
figure;
plot(fpr, tpr)
h = gca;
h.XLim(1) = -0.1;
h.YLim(2) = 1.1;
xlabel('False positive rate')
ylabel('True positive rate')
title('ROC Curve')
```



## Input Arguments

**mdl** — Binary, kernel classification object

`ClassificationKernel` object

Binary, kernel classification model, specified as a `ClassificationKernel` model object. You can create a `ClassificationKernel` model object using `fitckernel`.

**x** — Predictor data

$n$ -by- $p$  numeric matrix

Predictor data, specified as an  $n$ -by- $p$  numeric matrix, where  $n$  is the number of observations and  $p$  is the number of predictors used to train `Mdl`.

Data Types: `single` | `double`

## Output Arguments

### **Label** — Predicted class labels

categorical array | character array | logical matrix | numeric matrix | cell array of character vectors

Predicted class labels, returned as a categorical or character array, logical or numeric matrix, or cell array of character vectors.

`Label` is an  $n$ -by-1 array of the same data type as the observed class labels ( $Y$ ) used to train `Mdl`, where  $n$  is the number of observations in  $X$ .

`predict` classifies observations into the class yielding the largest score.

### **Score** — Classification scores

numeric array

Classification scores on page 32-7138, returned as a  $n$ -by-2 numeric array, where  $n$  is the number of observations in  $X$ . `Score(i, j)` is the score for classifying observation  $i$  into class  $j$ . `Mdl.ClassNames` stores the order of the classes.

If `Mdl.Learner` is `'logistic'`, then classification scores are posterior probabilities.

## Definitions

### Classification Score

For linear classification models, the raw classification score for classifying the observation  $x$ , a row vector, into the positive class is defined by

$$f(x) = T(x)\beta + b.$$



$T(\cdot)$  is a transformation of an observation for feature expansion,  $\beta$  is the estimated column vector of coefficients (the model property `Beta`) and  $b$  is the estimated, scalar bias (the model property `Bias`).

The raw classification score for classifying  $x$  into the negative class is  $-f(x)$ . The software classifies observations into the class that yields the positive score.

If the linear classification model consists of logistic regression learners, then the software applies the 'logit' score transformation to the raw classification scores (see `ScoreTransform`).

## Extended Capabilities

### Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function fully supports tall arrays. For more information, see “Tall Arrays” (MATLAB).

### See Also

`ClassificationKernel` | `confusionmat` | `fitckernel` | `perfcurve` | `resume`

**Introduced in R2017b**

## resume

Resume training of Gaussian kernel classification model

### Syntax

```
UpdatedMdl = resume(Mdl, X, Y)
UpdatedMdl = resume(Mdl, X, Y, Name, Value)
[UpdatedMdl, FitInfo] = resume(____)
```

### Description

`UpdatedMdl = resume(Mdl, X, Y)` continues training using the same training option, including the training data (predictor data `X` and class labels `Y`) and the same feature expansion, used for training `Mdl` and returns a new binary, Gaussian kernel classification model `UpdatedMdl`. The training starts at the current estimated parameters in `Mdl`.

`UpdatedMdl = resume(Mdl, X, Y, Name, Value)` returns a kernel classification model with additional options specified by one or more `Name, Value` pair arguments. For example, you can modify convergence control options such as convergence tolerances and the maximal number of extra optimization iterations.

`[UpdatedMdl, FitInfo] = resume(____)` also returns the fit information in the structure array `FitInfo` using any of the previous syntaxes.

### Examples

#### Predict Class Labels and Resume Training

Load the `ionosphere` data. This data has 34 predictors and 351 binary responses for radar returns, either bad ('b') or good ('g').

```
load ionosphere
```

Partition the data set into training and test sets. Specify a 20% holdout sample for testing.

```
rng('default') % For reproducibility
Partition = cvpartition(Y,'Holdout',0.20);
trainingInds = training(Partition); % Indices for the training set
XTrain = X(trainingInds,:);
YTrain = Y(trainingInds);
testInds = test(Partition); % Indices for the test set
XTest = X(testInds,:);
YTest = Y(testInds);
```

Train a binary, kernel classification model that can identify whether the radar return is bad ('b') or good ('g').

```
Mdl = fitckernel(XTrain,YTrain,'IterationLimit',5,'Verbose',1);
```

```
=====
| Solver | Pass | Iteration | Objective | Step | Gradient | F
|        |      |           |           |      | magnitude | char
|=====
| LBFGS | 1 | 0 | 1.000000e+00 | 0.000000e+00 | 2.811388e-01 | 1.
| LBFGS | 1 | 1 | 7.585395e-01 | 4.000000e+00 | 3.594306e-01 | 1.
| LBFGS | 1 | 2 | 7.160994e-01 | 1.000000e+00 | 2.028470e-01 | 6.
| LBFGS | 1 | 3 | 6.825272e-01 | 1.000000e+00 | 2.846975e-02 | 2.
| LBFGS | 1 | 4 | 6.699435e-01 | 1.000000e+00 | 1.779359e-02 | 1.
| LBFGS | 1 | 5 | 6.535619e-01 | 1.000000e+00 | 2.669039e-01 | 4.
|=====
```

Mdl is a ClassificationKernel model.

Predict the test-sample labels, construct a confusion matrix for the test data, and estimate the classification error for the test-sample.

```
label = predict(Mdl,XTest);
ConfusionTest = confusionmat(YTest,label)
L = loss(Mdl,XTest,YTest)
```

```
ConfusionTest =

     0     25
     0     45
```

L =

0.3594

Mdl misclassifies all bad radar returns as good returns.

Continue training by using `resume`. The function `resume` continues training using the same training options used for training Mdl.

```
UpdatedMdl = resume(Mdl,XTrain,YTrain);
```

Solver	Pass	Iteration	Objective	Step	Gradient magnitude	char
LBFGS	1	0	6.535619e-01	0.000000e+00	2.669039e-01	
LBFGS	1	1	6.132547e-01	1.000000e+00	6.355537e-03	1.
LBFGS	1	2	5.938316e-01	4.000000e+00	3.202847e-02	1.
LBFGS	1	3	4.169274e-01	1.000000e+00	1.530249e-01	7.
LBFGS	1	4	3.679212e-01	5.000000e-01	2.740214e-01	2.
LBFGS	1	5	3.332261e-01	1.000000e+00	1.423488e-02	9.
LBFGS	1	6	3.235335e-01	1.000000e+00	7.117438e-03	7.
LBFGS	1	7	3.112331e-01	1.000000e+00	6.049822e-02	1.
LBFGS	1	8	2.972144e-01	1.000000e+00	7.117438e-03	5.
LBFGS	1	9	2.837450e-01	1.000000e+00	8.185053e-02	1.
LBFGS	1	10	2.797642e-01	1.000000e+00	3.558719e-02	5.
LBFGS	1	11	2.771280e-01	1.000000e+00	2.846975e-02	2.
LBFGS	1	12	2.741570e-01	1.000000e+00	3.914591e-02	3.
LBFGS	1	13	2.725701e-01	5.000000e-01	1.067616e-01	8.
LBFGS	1	14	2.667147e-01	1.000000e+00	3.914591e-02	3.
LBFGS	1	15	2.621152e-01	1.000000e+00	7.117438e-03	5.
LBFGS	1	16	2.601652e-01	1.000000e+00	3.558719e-02	3.
LBFGS	1	17	2.589052e-01	1.000000e+00	3.202847e-02	3.
LBFGS	1	18	2.583185e-01	1.000000e+00	7.117438e-03	6.
LBFGS	1	19	2.556482e-01	1.000000e+00	9.252669e-02	4.
LBFGS	1	20	2.542643e-01	1.000000e+00	7.117438e-02	4.
Solver	Pass	Iteration	Objective	Step	Gradient magnitude	char
LBFGS	1	21	2.532117e-01	1.000000e+00	1.067616e-02	1.
LBFGS	1	22	2.529890e-01	1.000000e+00	2.135231e-02	1.
LBFGS	1	23	2.523232e-01	1.000000e+00	3.202847e-02	1.
LBFGS	1	24	2.506736e-01	1.000000e+00	1.779359e-02	2.

	LBFGS		1		25		2.501995e-01		1.000000e+00		1.779359e-02		2.
	LBFGS		1		26		2.488242e-01		1.000000e+00		3.558719e-03		1.
	LBFGS		1		27		2.485295e-01		5.000000e-01		3.202847e-02		1.
	LBFGS		1		28		2.482244e-01		1.000000e+00		4.270463e-02		8.
	LBFGS		1		29		2.479714e-01		1.000000e+00		3.558719e-03		7.
	LBFGS		1		30		2.477316e-01		1.000000e+00		3.202847e-02		3.
	LBFGS		1		31		2.476178e-01		2.500000e-01		3.202847e-02		5.
	LBFGS		1		32		2.474874e-01		1.000000e+00		1.779359e-02		3.
	LBFGS		1		33		2.473980e-01		1.000000e+00		7.117438e-03		2.
	LBFGS		1		34		2.472935e-01		1.000000e+00		3.558719e-03		2.
	LBFGS		1		35		2.471418e-01		1.000000e+00		3.558719e-03		1.
	LBFGS		1		36		2.469862e-01		1.000000e+00		2.846975e-02		7.
	LBFGS		1		37		2.469598e-01		1.000000e+00		2.135231e-02		6.
	LBFGS		1		38		2.466941e-01		1.000000e+00		3.558719e-02		4.
	LBFGS		1		39		2.466660e-01		5.000000e-01		1.423488e-02		2.
	LBFGS		1		40		2.465605e-01		1.000000e+00		3.558719e-03		4.

Solver	Pass	Iteration	Objective	Step	Gradient magnitude	F char							
	LBFGS		1		41		2.465362e-01		1.000000e+00		1.423488e-02		5.
	LBFGS		1		42		2.463528e-01		1.000000e+00		3.558719e-03		2.
	LBFGS		1		43		2.463207e-01		1.000000e+00		1.511170e-03		3.
	LBFGS		1		44		2.462585e-01		5.000000e-01		7.117438e-02		2.
	LBFGS		1		45		2.461742e-01		1.000000e+00		7.117438e-03		2.
	LBFGS		1		46		2.461434e-01		1.000000e+00		3.202847e-02		3.
	LBFGS		1		47		2.461115e-01		1.000000e+00		7.117438e-03		1.
	LBFGS		1		48		2.460814e-01		1.000000e+00		1.067616e-02		1.
	LBFGS		1		49		2.460533e-01		5.000000e-01		1.423488e-02		1.
	LBFGS		1		50		2.460111e-01		1.000000e+00		1.423488e-02		4.
	LBFGS		1		51		2.459414e-01		1.000000e+00		1.067616e-02		3.
	LBFGS		1		52		2.458809e-01		1.000000e+00		1.423488e-02		1.
	LBFGS		1		53		2.458479e-01		1.000000e+00		1.067616e-02		1.
	LBFGS		1		54		2.458146e-01		1.000000e+00		1.455008e-03		1.
	LBFGS		1		55		2.457878e-01		1.000000e+00		7.117438e-03		1.
	LBFGS		1		56		2.457519e-01		1.000000e+00		2.491103e-02		1.
	LBFGS		1		57		2.457420e-01		1.000000e+00		7.473310e-02		9.
	LBFGS		1		58		2.457212e-01		1.000000e+00		3.558719e-03		3.
	LBFGS		1		59		2.457089e-01		1.000000e+00		4.270463e-02		6.
	LBFGS		1		60		2.457047e-01		5.000000e-01		1.423488e-02		3.

Solver	Pass	Iteration	Objective	Step	Gradient magnitude	F char
--------	------	-----------	-----------	------	--------------------	--------

	LBFGS		1		61		2.456991e-01		1.000000e+00		1.423488e-02		5.
	LBFGS		1		62		2.456898e-01		1.000000e+00		1.779359e-02		4.
	LBFGS		1		63		2.456792e-01		1.000000e+00		1.779359e-02		5.
	LBFGS		1		64		2.456603e-01		1.000000e+00		1.403782e-03		5.
	LBFGS		1		65		2.456482e-01		1.000000e+00		3.558719e-03		6.
	LBFGS		1		66		2.456358e-01		1.000000e+00		1.476262e-03		1.
	LBFGS		1		67		2.456124e-01		1.000000e+00		3.558719e-03		8.
	LBFGS		1		68		2.455980e-01		1.000000e+00		1.067616e-02		9.
	LBFGS		1		69		2.455780e-01		1.000000e+00		1.067616e-02		5.
	LBFGS		1		70		2.455633e-01		1.000000e+00		3.558719e-03		1.
	LBFGS		1		71		2.455449e-01		1.000000e+00		1.423488e-02		7.
	LBFGS		1		72		2.455261e-01		1.000000e+00		3.558719e-02		1.
	LBFGS		1		73		2.455142e-01		1.000000e+00		1.067616e-02		1.
	LBFGS		1		74		2.455075e-01		1.000000e+00		1.779359e-02		1.
	LBFGS		1		75		2.455008e-01		1.000000e+00		3.914591e-02		1.
	LBFGS		1		76		2.454943e-01		1.000000e+00		2.491103e-02		3.
	LBFGS		1		77		2.454918e-01		5.000000e-01		3.202847e-02		9.
	LBFGS		1		78		2.454870e-01		1.000000e+00		1.779359e-02		4.
	LBFGS		1		79		2.454865e-01		5.000000e-01		3.558719e-03		7.
	LBFGS		1		80		2.454775e-01		1.000000e+00		5.693950e-02		8.
=====													
	Solver		Pass		Iteration		Objective		Step		Gradient		R
											magnitude		char
=====													
	LBFGS		1		81		2.454686e-01		1.000000e+00		1.183730e-03		1.
	LBFGS		1		82		2.454612e-01		1.000000e+00		2.135231e-02		1.
	LBFGS		1		83		2.454506e-01		1.000000e+00		3.558719e-03		6.
	LBFGS		1		84		2.454436e-01		1.000000e+00		1.423488e-02		1.
	LBFGS		1		85		2.454378e-01		1.000000e+00		1.423488e-02		3.
	LBFGS		1		86		2.454249e-01		1.000000e+00		1.423488e-02		8.
	LBFGS		1		87		2.454101e-01		1.000000e+00		1.067616e-02		3.
	LBFGS		1		88		2.453963e-01		1.000000e+00		1.779359e-02		5.
	LBFGS		1		89		2.453866e-01		1.000000e+00		1.067616e-02		1.
	LBFGS		1		90		2.453821e-01		1.000000e+00		7.117438e-03		2.
	LBFGS		1		91		2.453790e-01		5.000000e-01		6.761566e-02		8.
	LBFGS		1		92		2.453603e-01		1.000000e+00		2.135231e-02		1.
	LBFGS		1		93		2.453540e-01		1.000000e+00		2.135231e-02		2.
	LBFGS		1		94		2.453482e-01		1.000000e+00		1.779359e-02		1.
	LBFGS		1		95		2.453461e-01		1.000000e+00		1.779359e-02		1.
	LBFGS		1		96		2.453371e-01		1.000000e+00		3.558719e-02		2.
	LBFGS		1		97		2.453305e-01		1.000000e+00		4.270463e-02		7.
	LBFGS		1		98		2.453283e-01		2.500000e-01		2.135231e-02		3.
	LBFGS		1		99		2.453246e-01		1.000000e+00		3.558719e-03		3.
	LBFGS		1		100		2.453214e-01		1.000000e+00		3.202847e-02		1.

Solver	Pass	Iteration	Objective	Step	Gradient magnitude	char
LBFGS	1	101	2.453168e-01	1.000000e+00	1.067616e-02	3.
LBFGS	1	102	2.453155e-01	5.000000e-01	4.626335e-02	3.
LBFGS	1	103	2.453136e-01	1.000000e+00	1.779359e-02	2.
LBFGS	1	104	2.453119e-01	1.000000e+00	3.202847e-02	4.
LBFGS	1	105	2.453093e-01	1.000000e+00	1.423488e-02	2.
LBFGS	1	106	2.453090e-01	1.000000e+00	2.846975e-02	1.
LBFGS	1	107	2.453048e-01	1.000000e+00	1.423488e-02	3.
LBFGS	1	108	2.453040e-01	1.000000e+00	3.558719e-02	1.
LBFGS	1	109	2.452977e-01	1.000000e+00	1.423488e-02	8.
LBFGS	1	110	2.452934e-01	1.000000e+00	2.135231e-02	5.
LBFGS	1	111	2.452886e-01	1.000000e+00	1.779359e-02	3.
LBFGS	1	112	2.452854e-01	1.000000e+00	1.067616e-02	2.
LBFGS	1	113	2.452836e-01	1.000000e+00	1.067616e-02	1.
LBFGS	1	114	2.452817e-01	1.000000e+00	7.117438e-03	4.
LBFGS	1	115	2.452741e-01	1.000000e+00	1.779359e-02	9.
LBFGS	1	116	2.452691e-01	1.000000e+00	2.135231e-02	9.

Predict the test-sample labels, construct a confusion matrix for the test data, and estimate the classification error for the test-sample.

```
UpdatedLabel = predict(UpdatedMdl,XTest);
UpdatedConfusionTest = confusionmat(YTest,UpdatedLabel)
UpdatedL = loss(UpdatedMdl,XTest,YTest)
```

```
UpdatedConfusionTest =
```

```
    23     2
     7    38
```

```
UpdatedL =
```

```
    0.1284
```

The classification error decreased after updating the classification model with more iterations.

### Resume Training with Modified Convergence Control Training Options

Load the `ionosphere` data. This data has 34 predictors and 351 binary responses for radar returns, either bad ('b') or good ('g').

```
load ionosphere
```

Partition the data set into training and test sets. Specify a 20% holdout sample for testing.

```
rng('default') % For reproducibility
Partition = cvpartition(Y,'Holdout',0.20);
trainingInds = training(Partition); % Indices for the training set
XTrain = X(trainingInds,:);
YTrain = Y(trainingInds);
testInds = test(Partition); % Indices for the test set
XTest = X(testInds,:);
YTest = Y(testInds);
```

Train a binary, kernel classification model with relaxed convergence control training options by using the name-value pair arguments 'BetaTolerance' and 'GradientTolerance'.

```
[Mdl,FitInfo] = fitckernel(XTrain,YTrain,'Verbose',1, ...
    'BetaTolerance',1e-1,'GradientTolerance',1e-1);
```

```
=====
| Solver | Pass | Iteration | Objective | Step | Gradient |
|        |      |           |           |      | magnitude |
|        |      |           |           |      |           |
|=====|=====|=====|=====|=====|=====|
| LBFGS | 1 | 0 | 1.000000e+00 | 0.000000e+00 | 2.811388e-01 |
| LBFGS | 1 | 1 | 7.585395e-01 | 4.000000e+00 | 3.594306e-01 | 1.
| LBFGS | 1 | 2 | 7.160994e-01 | 1.000000e+00 | 2.028470e-01 | 6.
| LBFGS | 1 | 3 | 6.825272e-01 | 1.000000e+00 | 2.846975e-02 | 2.
|=====|=====|=====|=====|=====|=====|
```

`Mdl` is a `ClassificationKernel` model.

Predict the test-sample labels, construct a confusion matrix for the test data, and estimate the classification error for the test-sample.

```
label = predict(Mdl,XTest);
ConfusionTest = confusionmat(YTest,label)
L = loss(Mdl,XTest,YTest)
```



```
ConfusionTest =
```

```
    0    25
    0    45
```

```
L =
```

```
    0.3594
```

Mdl misclassifies all bad radar returns as good returns.

Continue training by using resume with new convergence control training options.

```
[UpdatedMdl,UpdatedFitInfo] = resume(Mdl,XTrain,YTrain, ...
    'BetaTolerance',1e-2,'GradientTolerance',1e-2);
```

Solver	Pass	Iteration	Objective	Step	Gradient magnitude	char
LBFGS	1	0	6.825272e-01	0.000000e+00	2.846975e-02	
LBFGS	1	1	6.692805e-01	2.000000e+00	2.846975e-02	1.
LBFGS	1	2	6.466824e-01	1.000000e+00	2.348754e-01	4.
LBFGS	1	3	5.441382e-01	2.000000e+00	1.743772e-01	5.
LBFGS	1	4	5.222333e-01	1.000000e+00	3.309609e-01	7.
LBFGS	1	5	3.776579e-01	1.000000e+00	1.103203e-01	6.
LBFGS	1	6	3.523520e-01	1.000000e+00	5.338078e-02	1.
LBFGS	1	7	3.422319e-01	5.000000e-01	3.202847e-02	9.
LBFGS	1	8	3.341895e-01	1.000000e+00	3.202847e-02	5.
LBFGS	1	9	3.199302e-01	1.000000e+00	4.982206e-02	8.
LBFGS	1	10	3.017904e-01	1.000000e+00	1.423488e-02	2.
LBFGS	1	11	2.853480e-01	1.000000e+00	3.558719e-02	9.
LBFGS	1	12	2.753979e-01	1.000000e+00	3.914591e-02	9.
LBFGS	1	13	2.647492e-01	1.000000e+00	3.914591e-02	9.
LBFGS	1	14	2.639242e-01	1.000000e+00	1.423488e-02	6.
LBFGS	1	15	2.617385e-01	1.000000e+00	1.779359e-02	2.
LBFGS	1	16	2.598600e-01	1.000000e+00	7.117438e-02	3.
LBFGS	1	17	2.594176e-01	1.000000e+00	1.067616e-02	2.
LBFGS	1	18	2.579350e-01	1.000000e+00	3.202847e-02	2.
LBFGS	1	19	2.570669e-01	1.000000e+00	1.779359e-02	4.
LBFGS	1	20	2.552954e-01	1.000000e+00	1.769940e-03	1.

Predict the test-sample labels, construct a confusion matrix for the test data, and estimate the classification error for the test-sample.

```
UpdatedLabel = predict(UpdatedMdl,XTest);  
UpdatedConfusionTest = confusionmat(YTest,UpdatedLabel)  
UpdatedL = loss(UpdatedMdl,XTest,YTest)
```

```
UpdatedConfusionTest =
```

```
    24     1  
     7    38
```

```
UpdatedL =
```

```
    0.1140
```

The classification error decreased after updating the classification model with smaller convergence tolerances.

## Input Arguments

### **mdl** — Binary, kernel classification object

`ClassificationKernel` object

Binary, kernel classification model, specified as a `ClassificationKernel` model object. You can create a `ClassificationKernel` model object using `fitckernel`.

### **x** — Predictor data used to train `mdl`

*n*-by-*p* numeric matrix

Predictor data used to train `mdl`, specified as an *n*-by-*p* numeric matrix, where *n* is the number of observations and *p* is the number of predictors used to train `mdl`.

Data Types: `single` | `double`

### **y** — Class labels used to train `mdl`

categorical array | character array | logical vector | vector of numeric values | cell array of character vectors

Class labels used to train `Mdl`, specified as a categorical or character array, logical or numeric vector, or cell array of character vectors.

Data Types: `categorical` | `cell` | `char` | `double` | `logical` | `single`

---

**Note** `resume` should only run on the same training data ( $X$  and  $Y$ ) and the same observation weights used for training `Mdl`. The function `resume` uses the same training options used for training `Mdl` including feature expansion.

---

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `UpdatedMdl = resume(Mdl, X, Y, 'GradientTolerance', 1e-5)` resumes training using the same training options used for training `Mdl`, except the absolute gradient tolerance.

### **Weights** — Observation weights used to train `Mdl`

positive numeric vector

Observation weights used to train `Mdl`, specified as the comma-separated pair consisting of `'Weights'` and a positive numeric vector of length  $n$ , where  $n$  is the number of observations in  $X$ . `resume` weighs the observations in  $X$  with the corresponding values in `Weights`.

The default value is `ones(n, 1)`.

`resume` normalizes `Weights` to sum up to the value of the prior probability in the respective class.

Example: `'Weights', w`

Data Types: `single` | `double`

### **BetaTolerance** — Relative tolerance on linear coefficients and bias term

`BetaTolerance` value used to train `Mdl` (default) | nonnegative scalar

Relative tolerance on the linear coefficients and the bias term (intercept), specified as the comma-separated pair consisting of 'BetaTolerance' and a nonnegative scalar.

Let  $B_t = [\beta_t' \ b_t]$ , that is, the vector of the coefficients and the bias term at optimization

iteration  $t$ . If  $\left\| \frac{B_t - B_{t-1}}{B_t} \right\|_2 < \text{BetaTolerance}$ , then optimization terminates.

If you also specify GradientTolerance, then optimization terminates when the software satisfies either stopping criterion.

Example: 'BetaTolerance', 1e-6

Data Types: single | double

### **GradientTolerance — Absolute gradient tolerance**

GradientTolerance value used to train Mdl (default) | nonnegative scalar

Absolute gradient tolerance, specified as the comma-separated pair consisting of 'GradientTolerance' and a nonnegative scalar.

Let  $\nabla_{L_t}$  be the gradient vector of the objective function with respect to the coefficients and bias term at optimization iteration  $t$ . If  $\|\nabla_{L_t}\|_\infty = \max |\nabla_{L_t}| < \text{GradientTolerance}$ , then optimization terminates.

If you also specify BetaTolerance, then optimization terminates when the software satisfies either stopping criterion.

Example: 'GradientTolerance', 1e-5

Data Types: single | double

### **IterationLimit — Maximal number of extra optimization iterations**

positive integer

Maximal number of extra optimization iterations, specified as the comma-separated pair consisting of 'IterationLimit' and a positive integer value.

The default value is 1000 if the transformed data fits in memory (Mdl.ModelParameters.BlockSize) that you specified by using the 'BlockSize' name-value pair argument when training Mdl. Otherwise, the default value is 100.

Note that the default value is not the value used to train `Mdl`.

Example: `'IterationLimit', 500`

Data Types: `single` | `double`

## Output Arguments

### **UpdatedMdl** — Updated kernel classification model

`ClassificationKernel` object

Updated kernel classification model, returned as a `ClassificationKernel` model object.

### **FitInfo** — Optimization details

structure array

Optimization details, returned as a structure array including fields described in this table. Fields specify final values or name-value pair argument specifications.

Field	Description
<code>Solver</code>	Objective function minimization technique: <code>'LBFGS-fast'</code> , <code>'LBFGS-blockwise'</code> , or <code>'LBFGS-tall'</code> . For details, see “Algorithms” on page 32-7094 of <code>fitckernel</code> .
<code>LossFunction</code>	Loss function. Either <code>'hinge'</code> or <code>'logit'</code> depending on linear classification model type. See <code>Learner</code> of <code>fitckernel</code> .
<code>Lambda</code>	Regularization term strength. See <code>Lambda</code> of <code>fitckernel</code> .
<code>BetaTolerance</code>	Relative tolerance on linear coefficients and bias term. See <code>BetaTolerance</code> .
<code>GradientTolerance</code>	Absolute gradient tolerance. See <code>GradientTolerance</code> .
<code>ObjectiveValue</code>	Value of the objective function when optimization terminates. The classification loss plus the regularization term compose the objective function.
<code>GradientMagnitude</code>	Infinite norm of the gradient vector of the objective function when optimization terminates. See <code>BetaTolerance</code> .

Field	Description
RelativeChangeInBeta	Relative changes in the linear coefficients and the bias term when optimization terminates. See <code>GradientTolerance</code> .
FitTime	Elapsed, wall-clock time took to fit the model to the data in seconds.
History	History of optimization information. This field includes the optimization information of training <code>Mdl</code> as well. This field is empty ( <code>[]</code> ) if you specified 'Verbose', 0 when training <code>Mdl</code> . For details, see <code>Verbose</code> and “Algorithms” on page 32-7094 of <code>fitckernel</code> .

To access fields, use dot notation. For example, to access the vector of objective function values for each iteration, enter `FitInfo.Objective` in the Command Window.

It is good practice to examine `FitInfo` to assess whether convergence is satisfactory.

## Definitions

### Random Feature Expansion

The random feature expansion such as `Random Kitchen Sinks`[1] and `Fastfood`[2] is a scheme to approximate Gaussian kernels of the kernel classification algorithm for big data in a computationally efficient way.

The kernel classification algorithm searches for an optimal hyperplane that separates the data into two classes after mapping features into a high dimensional space. Nonlinear features that are not linearly separable in a low dimensional space can be separable in the expanded high dimensional space. All the calculations for hyperplane classification use nothing more than dot products. You can obtain a nonlinear classification model by replacing the dot product  $x_1x_2'$  with a nonlinear kernel function

$G(x_1, x_2) = \langle \varphi(x_1), \varphi(x_2) \rangle$ , where  $x_i$  is the  $i$ th observation (row vector) and  $\varphi(x_i)$  is a transformation that maps  $x_i$  to a high-dimensional space (called the “kernel trick”). However, evaluating  $G(x_1, x_2)$  (Gram matrix) for each pair of observations is computationally expensive for large data set (large  $n$ ).

The random feature expansion scheme finds a random transformation such that its dot product approximates the Gaussian kernel. That is,

$$G(x_1, x_2) = \langle \varphi(x_1), \varphi(x_2) \rangle \approx T(x_1)T(x_2)',$$

where  $T(x)$  maps  $x$  in  $\mathbb{R}^p$  to a high dimensional space ( $\mathbb{R}^m$ ). The Random Kitchen Sink[1] scheme uses the following random transformation:

$$T(x) = m^{-1/2} \exp(iZx')$$

where  $Z \in \mathbb{R}^{m \times p}$  is a sample drawn from  $N(0, \sigma^{-2})$  and  $\sigma^2$  is a kernel scale. This scheme requires  $O(mp)$  computation and storage. The Fastfood[2] scheme introduces another random basis  $V$  instead of  $Z$  using Hadamard matrices combined with Gaussian scaling matrices. This random basis reduces computation cost to  $O(m \log p)$  and reduces storage to  $O(m)$ .

The function `fitckernel` uses the Fastfood scheme for random feature expansion and uses linear classification to train a Gaussian kernel classification model. Unlike solvers in the function `fitcsvm` that require computation of the  $n$ -by- $n$  Gram matrix, the solver in `fitckernel` only needs to form a matrix of size  $n$ -by- $m$  with  $m$  typically much less than  $n$  for big data.

## Extended Capabilities

### Tall Arrays

Calculate with arrays that have more rows than fit in memory.

Usage notes and limitations:

- The 'IterationLimit' name-value pair argument has a different default compared to the in-memory `resume` function. The default value is relaxed to 20.
- `resume` uses a block-wise strategy. For details, see “Algorithms” on page 32-7094 of `fitckernel`.

For more information, see “Tall Arrays” (MATLAB).

### See Also

`ClassificationKernel` | `fitckernel` | `predict`

**Introduced in R2017b**



# Sample Data Sets

## Sample Data Sets

Statistics and Machine Learning Toolbox software includes the sample data sets in the following table.

To load a data set into the MATLAB workspace, type:

```
load filename
```

where *filename* is one of the files listed in the table.

Data sets contain individual data variables, description variables with references, and dataset arrays encapsulating the data set and its description, as appropriate.

File	Description of Data Set
acetylene.mat	Chemical reaction data with correlated predictors
arrhythmia.mat	Cardiac arrhythmia data from the UCI machine learning repository
carbig.mat	Measurements of cars, 1970–1982
carsmall.mat	Subset of <i>carbig.mat</i> . Measurements of cars, 1970, 1976, 1982
census1994.mat	Adult data from the UCI machine learning repository
cereal.mat	Breakfast cereal ingredients
cities.mat	Quality of life ratings for U.S. metropolitan areas
discrim.mat	A version of <i>cities.mat</i> used for discriminant analysis
examgrades.mat	Exam grades on a scale of 0–100
fisheriris.mat	Fisher's 1936 iris data
flu.mat	Google Flu Trends estimated ILI (influenza-like illness) percentage for various regions of the US, and CDC weighted ILI percentage based on sentinel provider reports
gas.mat	Gasoline prices around the state of Massachusetts in 1993
hald.mat	Heat of cement vs. mix of ingredients
hogg.mat	Bacteria counts in different shipments of milk
hospital.mat	Simulated hospital data
imports-85.mat	1985 Auto Imports Database from the UCI repository

<b>File</b>	<b>Description of Data Set</b>
ionosphere.mat	Ionosphere dataset from the UCI machine learning repository
kmeansdata.mat	Four-dimensional clustered data
lawdata.mat	Grade point average and LSAT scores from 15 law schools
mileage.mat	Mileage data for three car models from two factories
moore.mat	Biochemical oxygen demand on five predictors
morse.mat	Recognition of Morse code distinctions by non-coders
nlpdata.mat	Natural language processing data extracted from the MathWorks® documentation.
ovariancancer.mat	Grouped observations on 4000 predictors
parts.mat	Dimensional run-out on 36 circular parts
polydata.mat	Sample data for polynomial fitting
popcorn.mat	Popcorn yield by popper type and brand
reaction.mat	Reaction kinetics for Hougen-Watson model
sat.dat	Scholastic Aptitude Test averages by gender and test (table)
sat2.dat	Scholastic Aptitude Test averages by gender and test (csv)
spectra.mat	NIR spectra and octane numbers of 60 gasoline samples
stockreturns.mat	Simulated stock returns



# Distribution Reference

## Bernoulli Distribution

**In this section...**

“Overview” on page B-2

“Parameters” on page B-2

“Probability Mass Function” on page B-2

“Mean and Variance” on page B-2

“Relationship to Other Distributions” on page B-3

### Overview

The Bernoulli distribution is a discrete probability distribution with the only two possible values for the random variable. Each instance of an event with a Bernoulli distribution is called a Bernoulli trial.

### Parameters

The Bernoulli distribution uses the following parameter.

Parameter	Description	Support
$p$	Probability of success	$0 \leq p \leq 1$

### Probability Mass Function

The probability mass function (pmf) is

$$f(x | p) = \begin{cases} 1 - p & , \quad x = 0, \\ p & , \quad x = 1. \end{cases}$$

### Mean and Variance

The mean is

$$\text{mean} = p.$$

The variance is

$$\text{var} = p(1 - p).$$

## Relationship to Other Distributions

The Bernoulli distribution is a special case of the binomial distribution on page B-9, with the number of trials  $n = 1$ . The geometric distribution on page B-64 models the number of Bernoulli trials before the first success (or first failure).

## See Also

`BinomialDistribution`

## More About

- “Binomial Distribution” on page B-9
- “Working with Probability Distributions” on page 5-3
- “Supported Distributions” on page 5-17

## Beta Distribution

In this section...
“Overview” on page B-4
“Parameters” on page B-4
“Probability Density Function” on page B-5
“Cumulative Distribution Function” on page B-7
“Example” on page B-7

### Overview

The beta distribution describes a family of curves that are unique in that they are nonzero only on the interval  $(0, 1)$ . A more general version of the function assigns parameters to the endpoints of the interval.

Statistics and Machine Learning Toolbox provides several ways to work with the beta distribution. You can use the following approaches to estimate parameters from sample data, compute the pdf, cdf, and icdf, generate random numbers, and more.

- Fit a probability distribution object to sample data, or create a probability distribution object with specified parameter values. See *Using BetaDistribution Objects* for more information.
- Work with data input from matrices, tables, and dataset arrays using probability distribution functions. See “Supported Distributions” on page 5-17 for a list of beta distribution functions.
- Interactively fit, explore, and generate random numbers from the distribution using an app or user interface.

For more information on each of these options, see “Working with Probability Distributions” on page 5-3.

### Parameters

The beta distribution uses the following parameters.



Parameter	Description	Support
a	First shape parameter	$a > 0$
b	Second shape parameter	$b > 0$

## Probability Density Function

### Definition

The probability density function (pdf) of the beta distribution is

$$y = f(x | a, b) = \frac{1}{B(a, b)} x^{a-1} (1-x)^{b-1} I_{[0,1]}(x)$$

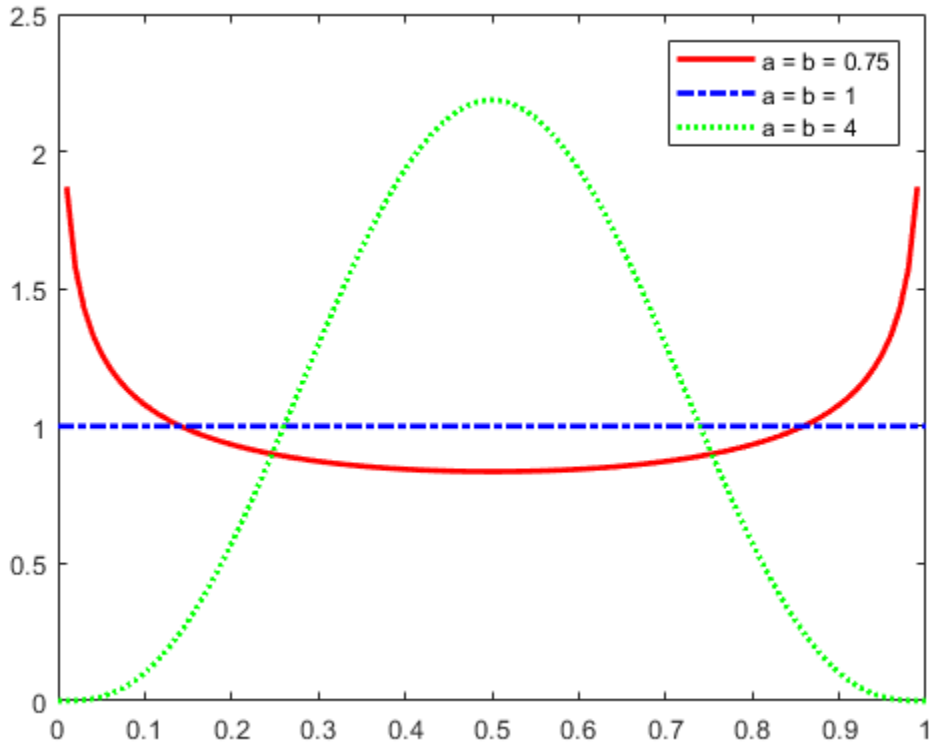
where  $B(\cdot)$  is the Beta function. The indicator function  $I_{(0,1)}(x)$  ensures that only values of  $x$  in the range  $(0, 1)$  have nonzero probability.

### Plot

This plot shows how changing the value of the parameters alters the shape of the pdf. The constant pdf (the flat line) shows that the standard uniform distribution is a special case of the beta distribution, which occurs when  $a = b = 1$ .

```
X = 0:.01:1;
y1 = betapdf(X,0.75,0.75);
y2 = betapdf(X,1,1);
y3 = betapdf(X,4,4);

figure
plot(X,y1,'Color','r','LineWidth',2)
hold on
plot(X,y2,'LineStyle','-','Color','b','LineWidth',2)
plot(X,y3,'LineStyle',':','Color','g','LineWidth',2)
legend({'a = b = 0.75','a = b = 1','a = b = 4'},'Location','NorthEast');
hold off
```



### Relationship to Other Distributions

The beta distribution has a functional relationship with the  $t$  distribution. If  $Y$  is an observation from Student's  $t$  distribution with  $\nu$  degrees of freedom, then the following transformation generates  $X$ , which is beta distributed.

$$X = \frac{1}{2} + \frac{1}{2} \frac{Y}{\sqrt{\nu + Y^2}}$$

$$\text{If } Y \sim t(\nu), \text{ then } X \sim \beta\left(\frac{\nu}{2}, \frac{\nu}{2}\right)$$

This relationship is used to compute values of the  $t$  cdf and inverse function as well as generating  $t$  distributed random numbers.

## Cumulative Distribution Function

The beta cdf is the same as the incomplete beta function.

## Example

Suppose you are collecting data that has hard lower and upper bounds of zero and one respectively. Parameter estimation is the process of determining the parameters of the beta distribution that fit this data best in some sense.

One popular criterion of goodness is to maximize the likelihood function. The likelihood has the same form as the beta pdf. But for the pdf, the parameters are known constants and the variable is  $x$ . The likelihood function reverses the roles of the variables. Here, the sample values (the  $x$ 's) are already observed. So they are the fixed constants. The variables are the unknown parameters. Maximum likelihood estimation (MLE) involves calculating the values of the parameters that give the highest likelihood given the particular set of data.

The function `betafit` returns the MLEs and confidence intervals for the parameters of the beta distribution. Here is an example using random numbers from the beta distribution with  $a = 5$  and  $b = 0.2$ .

```
rng default % For reproducibility
r = betarnd(5,0.2,100,1);
[phat, pci] = betafit(r)
```

```
phat =
```

```
    7.4911    0.2135
```

```
pci =
```

```
    5.0861    0.1744
   11.0334    0.2614
```

The MLE for parameter  $a$  is 7.4911, compared to the true value of 5. The 95% confidence interval for  $a$  goes from 5.0861 to 11.0334, which does not include the true value. While

this is an unlikely result, it does sometimes happen when estimating distribution parameters.

Similarly the MLE for parameter  $b$  is 0.2135, compared to the true value of 0.2. The 95% confidence interval for  $b$  goes from 0.1744 to 0.2614, which does include the true value. In this made-up example you know the “true value.” In experimentation you do not.

## See Also

BetaDistribution

## More About

- “Working with Probability Distributions” on page 5-3
- “Supported Distributions” on page 5-17

# Binomial Distribution

In this section...
“Overview” on page B-9
“Parameters” on page B-9
“Probability Density Function” on page B-9
“Mean and Variance” on page B-10
“Relationship to Other Distributions” on page B-10
“Example” on page B-10

## Overview

The binomial distribution models the total number of successes in repeated trials from an infinite population under the following conditions:

- Only two outcomes are possible on each of  $n$  trials.
- The probability of success for each trial is constant.
- All trials are independent of each other.

## Parameters

The binomial distribution uses the following parameters.

Parameter	Description	Support
$N$	Number of trials	positive integer
$p$	Probability of success	$0 \leq p \leq 1$

## Probability Density Function

The probability density function (pdf) is

$$f(x | N, p) = \binom{N}{x} p^x (1-p)^{N-x} \quad ; \quad x = 0, 1, 2, \dots, N,$$

where  $x$  is the number of successes in  $n$  trials of a Bernoulli process with probability of success  $p$ .

## Mean and Variance

The mean is  
 $\text{mean} = np$ .

The variance is  
 $\text{var} = np(1 - p)$ .

## Relationship to Other Distributions

The binomial distribution is a generalization of the Bernoulli distribution on page B-2, allowing for a number of trials  $n$  greater than 1. The binomial distribution generalizes to the multinomial distribution on page B-103 when there are more than two possible outcomes for each trial.

## Example

Suppose you are collecting data from a widget manufacturing process, and you record the number of widgets within specification in each batch of 100. You might be interested in the probability that an individual widget is within specification. Parameter estimation is the process of determining the parameter,  $p$ , of the binomial distribution that fits this data best in some sense.

One popular criterion of goodness is to maximize the likelihood function. The likelihood has the same form as the binomial pdf above. But for the pdf, the parameters ( $n$  and  $p$ ) are known constants and the variable is  $x$ . The likelihood function reverses the roles of the variables. Here, the sample values (the  $x$ 's) are already observed. So they are the fixed constants. The variables are the unknown parameters. MLE involves calculating the value of  $p$  that give the highest likelihood given the particular set of data.

The function `binofit` returns the MLEs and confidence intervals for the parameters of the binomial distribution. Here is an example using random numbers from the binomial distribution with  $n = 100$  and  $p = 0.9$ .

```
rng default; % for reproducibility
r = binornd(100,0.9)
[phat, pci] = binofit(r,100)
```

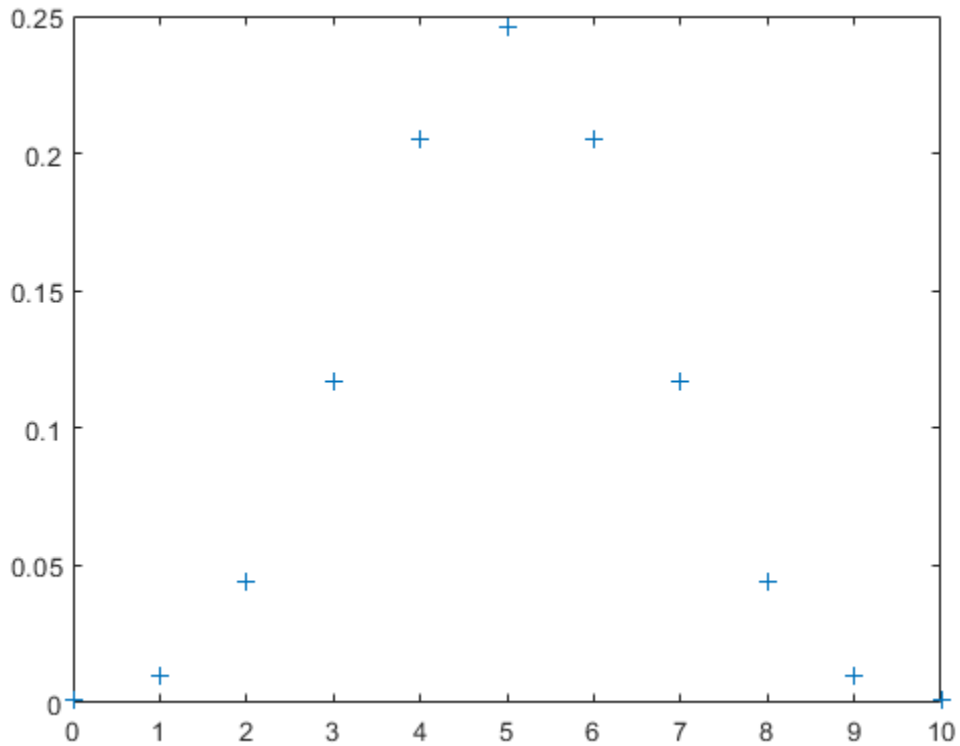
```
r =
```

```
85
phat =
    0.8500
pci =
    0.7647    0.9135
```

The MLE for parameter  $p$  is 0.8800, compared to the true value of 0.9. The 95% confidence interval for  $p$  goes from 0.7998 to 0.9364, which includes the true value. In this made-up example you know the “true value” of  $p$ . In experimentation you do not.

The following commands generate a plot of the binomial pdf for  $n = 10$  and  $p = 1/2$ .

```
x = 0:10;
y = binopdf(x,10,0.5);
plot(x,y, '+')
```



## See Also

`BinomialDistribution`

## More About

- “Bernoulli Distribution” on page B-2
- “Working with Probability Distributions” on page 5-3
- “Supported Distributions” on page 5-17



## Birnbauu-Saunderu Distribution

### In this section...

“Definition” on page B-13

“Background” on page B-13

“Parameters” on page B-14

### Definition

The Birnbauu-Saunderu distribution has the density function

$$\frac{1}{\sqrt{2\pi}} \exp \left\{ -\frac{\left( \sqrt{x/\beta} - \sqrt{\beta/x} \right)^2}{2\gamma^2} \right\} \left( \frac{\sqrt{x/\beta} + \sqrt{\beta/x}}{2\gamma x} \right)$$

with scale parameter  $\beta > 0$  and shape parameter  $\gamma > 0$ , for  $x > 0$ .

If  $x$  has a Birnbauu-Saunderu distribution with parameters  $\beta$  and  $\gamma$ , then

$$\frac{\left( \sqrt{x/\beta} - \sqrt{\beta/x} \right)}{\gamma}$$

has a standard normal distribution.

### Background

The Birnbauu-Saunderu distribution was originally proposed as a lifetime model for materials subject to cyclic patterns of stress and strain, where the ultimate failure of the material comes from the growth of a prominent flaw. In materials science, Miner's Rule suggests that the damage occurring after  $n$  cycles, at a stress level with an expected lifetime of  $N$  cycles, is proportional to  $n / N$ . Whenever Miner's Rule applies, the Birnbauu-Saunderu model is a reasonable choice for a lifetime distribution model.

## Parameters

To estimate distribution parameters, use `mle` or the Distribution Fitter app.

## See Also

`BirnbaumSaundersDistribution`

## More About

- “Working with Probability Distributions” on page 5-3
- “Supported Distributions” on page 5-17

## Burr Type XII Distribution

### In this section...

“Definition” on page B-15

“Background” on page B-15

“Parameters” on page B-16

“Fit a Burr Distribution and Draw the cdf” on page B-17

“Compare Lognormal and Burr pdfs” on page B-20

“Burr pdf for Various Parameters” on page B-21

“Survival and Hazard Functions of Burr Distribution” on page B-23

“Divergence of Parameter Estimates” on page B-25

### Definition

The Burr type XII distribution is a three-parameter family of distributions on the positive real line. The cumulative distribution function (cdf) of the Burr distribution is

$$F(x | \alpha, c, k) = 1 - \frac{1}{\left(1 + \left(\frac{x}{\alpha}\right)^c\right)^k}, \quad x > 0, \alpha > 0, c > 0, k > 0,$$

where  $c$  and  $k$  are the shape parameters and  $\alpha$  is the scale parameter. The probability density function (pdf) is

$$f(x | \alpha, c, k) = \frac{\frac{kc}{\alpha} \left(\frac{x}{\alpha}\right)^{c-1}}{\left(1 + \left(\frac{x}{\alpha}\right)^c\right)^{k+1}}, \quad x > 0, \alpha > 0, c > 0, k > 0.$$

The density of the Burr type XII distribution is L-shaped if  $c \leq 1$  and unimodal, otherwise.

### Background

Burr distribution was first discussed by Burr (1942) as a two-parameter family. An additional scale parameter was introduced by Tadikamalla (1980). It is a very flexible

distribution family that can express a wide range of distribution shapes. The Burr distribution includes, overlaps, or has as a limiting case, many commonly used distributions such as gamma, lognormal, loglogistic, bell-shaped, and J-shaped beta distributions (but not U-shaped). Some compound distributions also correspond to the Burr distribution. For example, compounding a Weibull distribution with a gamma distribution for its scale parameter results in a Burr distribution. Similarly, compounding an exponential distribution with a gamma distribution for its rate parameter,  $1/\mu$ , also yields a Burr distribution. The Burr distribution also has two asymptotic limiting cases: Weibull and Pareto Type I.

The Burr distribution can fit a wide range of empirical data. Different values of its parameters cover a broad set of skewness and kurtosis. Hence, it is used in various fields such as finance, hydrology, and reliability to model a variety of data types. Examples of data modeled by the Burr distribution are household income, crop prices, insurance risk, travel time, flood levels, and failure data.

The survival and hazard functions of Burr type XII distribution are, respectively,

$$S(x | \alpha, c, k) = \frac{1}{\left[1 + \left(\frac{x}{\alpha}\right)^c\right]^k}$$

and

$$h(x | \alpha, c, k) = \frac{\frac{kc}{\alpha} \left(\frac{x}{\alpha}\right)^{c-1}}{1 + \left(\frac{x}{\alpha}\right)^c}.$$

If  $c > 1$ , the hazard function  $h(x)$  is non-monotonic with a mode at  $x = \alpha(c - 1)^{1/c}$ .

## Parameters

The three-parameter Burr distribution is defined by its scale parameter  $\alpha$  and shape parameters  $c$  and  $k$ . You can estimate the parameters using `mle` or `fitdist`. Both functions support censored data for Burr distribution.

Generate sample data from a Burr distribution with scale parameter 0.5 and shape parameters 2 and 5.

```
rng('default')
R = random('burr', 0.5, 2, 5, 1000, 1);
```

Estimate the parameters and the confidence intervals.

```
[phat,pci] = mle(R, 'distribution', 'burr')

phat =

    0.4154    2.1217    4.0550

pci =

    0.2985    1.9560    2.4079
    0.5782    2.3014    6.8288
```

The default 95% confidence intervals for the parameters include the true parameter values.

The three-parameter Burr distribution converges asymptotically to one of the two limiting forms as its parameters diverge:

- If  $k \rightarrow 0$ ,  $c \rightarrow \infty$ ,  $ck = \lambda$ , then the Burr distribution reduces to a two-parameter Pareto distribution with the cdf

$$F_P = 1 - \left(\frac{x}{\alpha}\right)^{-\lambda}, \quad x \geq \alpha.$$

- If  $k \rightarrow \infty$ ,  $\alpha \rightarrow \infty$ ,  $\alpha/k^{1/c} = \theta$ , then the Burr distribution reduces to a two-parameter Weibull distribution with the cdf

$$F_W(x | c, \theta) = 1 - \exp\left[-\left(\frac{x}{\theta}\right)^c\right].$$

If `mle` or `fitdist` detects such divergence, it returns an error message, but informs you of the limiting distribution and corresponding parameter estimates for that distribution.

## Fit a Burr Distribution and Draw the cdf

This example shows how to fit a Burr distribution to data, draw the cdf, and construct a histogram with a Burr distribution fit.

1. Load the sample data.

```
load arrhythmia
```

The fifth column in `X` contains a measurement obtained from electrocardiograms, called QRS duration.

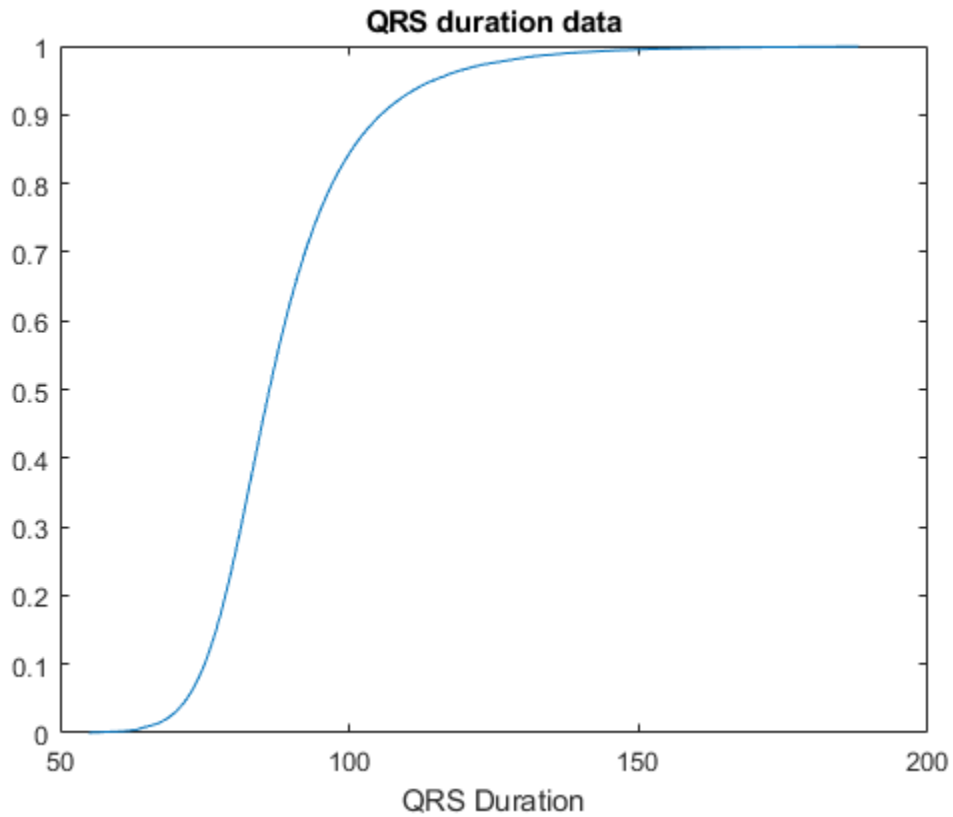
2. Fit a Burr distribution to the QRS duration data, and get the parameter estimates.

```
PD = fitdist(X(:,5), 'burr');
```

`PD` has the maximum likelihood estimates of the Burr distribution parameters in the property `Param`. The estimates are  $\alpha = 80.4515$ ,  $c = 18.9251$ ,  $k = 0.4492$ .

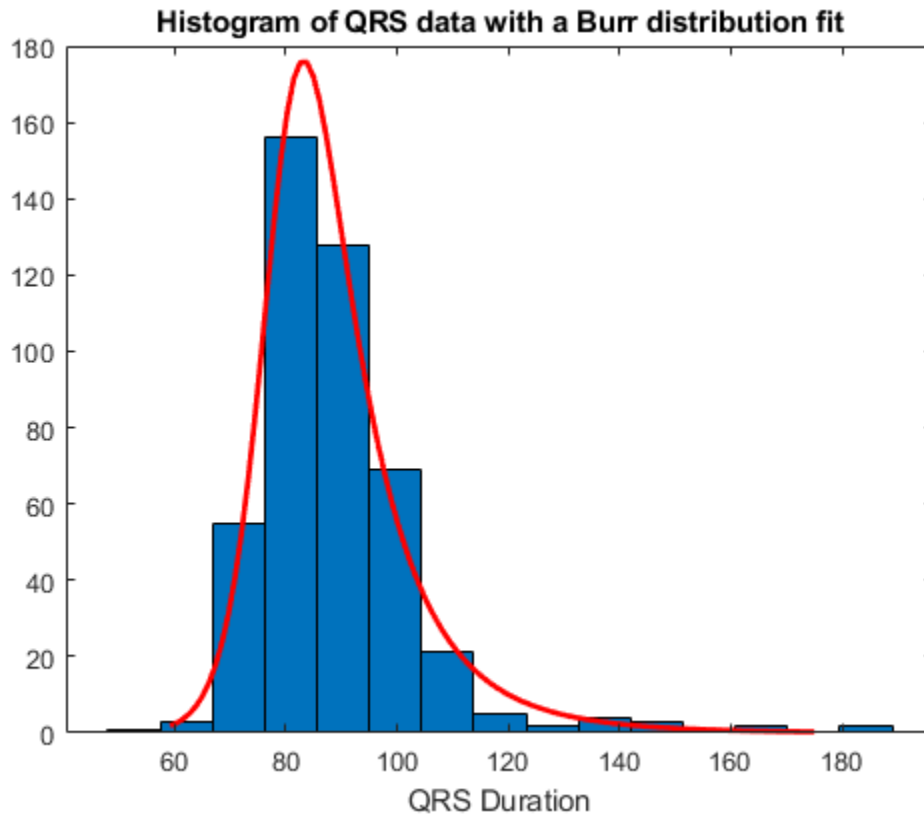
3. Plot the cdf of the QRS duration data.

```
QRScdf=cdf('burr', sortrows(X(:,5)), 80.4515, 18.9251, 0.4492);  
plot(sortrows(X(:,5)), QRScdf)  
title('QRS duration data')  
xlabel('QRS Duration')
```



4. Draw the histogram of QRS duration data with 15 bins and the pdf of the Burr distribution fit.

```
histfit(X(:,5),15,'burr')  
title('Histogram of QRS data with a Burr distribution fit')  
xlabel('QRS Duration')
```



## Compare Lognormal and Burr pdfs

This example shows how to compare the lognormal pdf to the Burr pdf using income data generated from a lognormal distribution.

Generate the income data.

```
rng default % for reproducibility
y = random('logn', log(25000), 0.65, 500, 1);
```

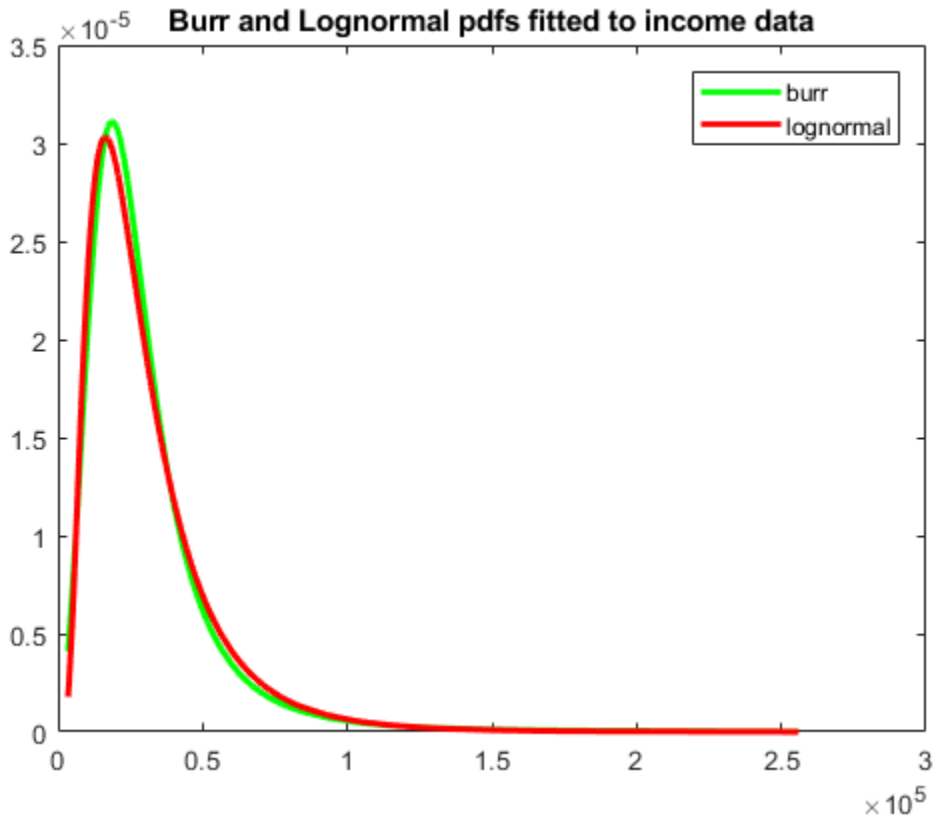
Fit a Burr distribution.

```
pd = fitdist(y, 'burr');
```



Plot both the Burr and lognormal pdfs of income data on the same axis.

```
p = pdf('burr', sortrows(y), 26007, 2.6374, 1.0966);
p2 = pdf('logn', sortrows(y), log(25000), 0.65);
plot(sortrows(y), p, 'g', sortrows(y), p2, 'r', 'LineWidth', 2)
title('Burr and Lognormal pdfs fitted to income data')
legend('burr', 'lognormal')
```



## Burr pdf for Various Parameters

This example shows how to create a variety of shapes for probability density functions of the Burr distribution.

```
X = 0:0.01:5;
c = [0.5 0.95 2 5];
k = [0.5 0.75 2 5];
alpha = [0.5 1 2 5];
colors = ['b';'g';'r';'k'];

figure
for i = 1:1:4
pdf1(i,:) = pdf('burr',X,1,c(i),0.5);
pdf2(i,:) = pdf('burr',X,1,2,k(i));
pdf3(i,:) = pdf('burr',X,alpha(i),2,0.5);

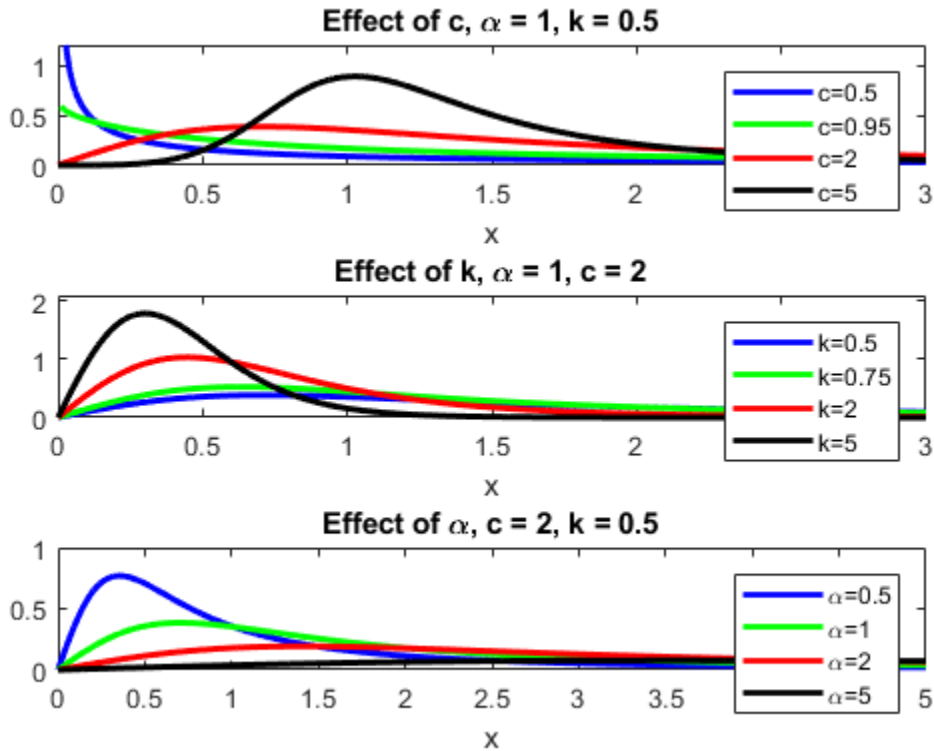
axC = subplot(3,1,1);
pC(i) = plot(X,pdf1(i,:),colors(i),'LineWidth',2);
title('Effect of c, \alpha = 1, k = 0.5'),xlabel('x')
hold on

axK = subplot(3,1,2);
pK(i) = plot(X,pdf2(i,:),colors(i),'LineWidth',2);
title('Effect of k, \alpha = 1, c = 2'),xlabel('x')
hold on

axAlpha = subplot(3,1,3);
pAlpha(i) = plot(X,pdf3(i,:),colors(i),'LineWidth',2);
title('Effect of \alpha, c = 2, k = 0.5'),xlabel('x')
hold on
end

set(axC,'XLim',[0 3],'YLim',[0 1.2]);
set(axK,'XLim',[0 3],'YLim',[0 2.1]);
set(axAlpha,'XLim',[0 5],'YLim',[0 1]);

legend(axC,'c=0.5','c=0.95','c=2','c=5');
legend(axK,'k=0.5','k=0.75','k=2','k=5');
legend(axAlpha,'\alpha=0.5','\alpha=1','\alpha=2','\alpha=5');
```



This figure illustrates how the shape and scale of the Burr distribution changes for different values of its parameters.

## Survival and Hazard Functions of Burr Distribution

This example shows how to find and plot the survival and hazard functions for a sample coming from a Burr distribution.

Generate the data.

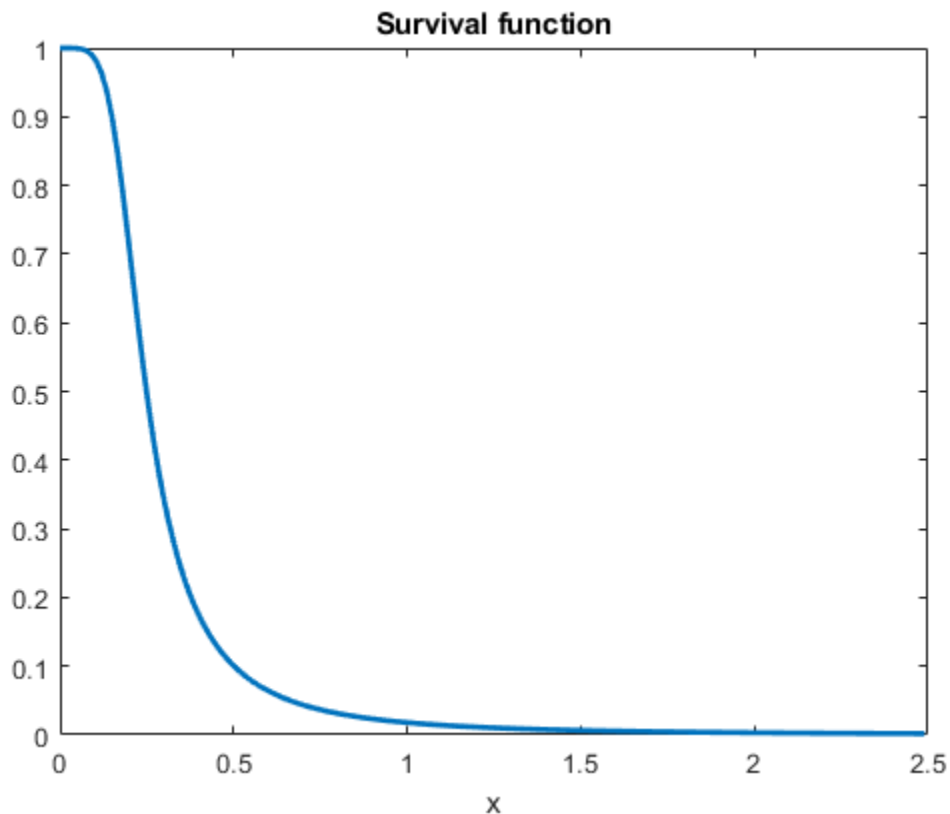
```
x = 0:0.015:2.5;
```

Evaluate the pdf and cdf of data in x.

```
Xpdf = pdf('burr',X,0.2,5,0.5);  
Xcdf = cdf('burr',X,0.2,5,0.5);
```

Evaluate and plot the survival function of data in X.

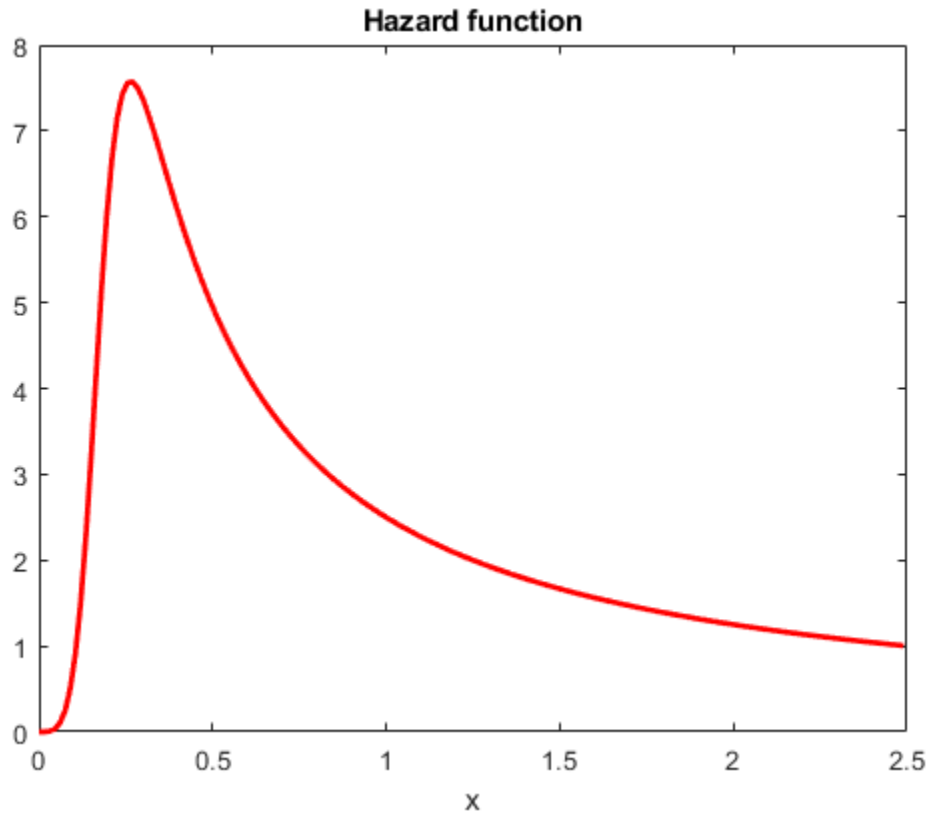
```
S = 1.-Xcdf; % survival function  
plot(X,S,'LineWidth',2)  
title('Survival function')  
xlabel('x')
```



Evaluate and plot the hazard function of data in X.

```
H = Xpdf./S; % hazard function  
plot(X,H,'r','LineWidth',2)
```

```
title('Hazard function')  
xlabel('x')
```



## Divergence of Parameter Estimates

This example shows how to interpret the display when the parameter estimates diverge when fitting a Burr distribution to input data.

1. Generate sample data from the Weibull distribution with parameters 0.5 and 2.

```
rng('default') % for reproducibility  
X = wblrnd(0.5,2,100,1);
```

2. Fit a Burr distribution.

```
PD = fitdist(X, 'burr');
```

```
Error using addburr>burrfit (line 566)
The data are not fit by a Burr distribution with finite parameters.
The maximum likelihood fit is provided by the k->Inf, alpha->Inf
limiting form of the Burr distribution: a Weibull distribution
with the parameters below.
    a (scale): 0.476817
    b (shape): 1.96219
```

```
Error in prob.BurrDistribution.fit (line 246)
    p = burrfitt(x,0.05,cens,freq,opt);
```

```
Error in fitdist>localfit (line 238)
pd = feval(fitter,x,'cens',c,'freq',f,varargin{:});
```

```
Error in fitdist (line 185)
    pd = localfit(dist,fitter,x,cens,freq,args{:});
```

The error message tells you that the Weibull family seems to fit the data better and gives you the parameter estimates from a Weibull fit. You can use those estimates directly. If you need covariance estimates for the parameters or other information about the fit, you can refit a Weibull distribution to the data.

3. Fit a Weibull distribution to the data and find the confidence intervals for the parameter estimates.

```
PD = fitdist(X, 'weibull');
paramci(PD)
```

```
ans =
    0.4291    1.6821
    0.5298    2.2890
```

These are the 95% confidence intervals of the parameter estimates for the Weibull distribution fit.

## References

- [1] Burr, Irving W. "Cumulative frequency functions." *The Annals of Mathematical Statistics*, Vol. 13, Number 2, 1942, pp. 215–232.

- [2] Tadikamalla, Pandu R. “A look at the Burr and related distributions.” *International Statistical Review*, Vol. 48, Number 3, 1980, pp. 337–344.
- [3] Rodriguez, Robert N. “A guide to the Burr type XII distributions.” *Biometrika*, Vol. 64, Number 1, 1977, pp. 129–134.
- [4] AL-Hussaini, Essam K. “A characterization of the Burr type XII distribution”. *Appl. Math. Lett.* Vol. 4, Number 1, 1991, pp. 59–61.
- [5] Grammig, Joachim and Kai-Oliver Maurer. “Non-monotonic hazard functions and the autoregressive conditional duration model.” *Econometrics Journal*, Vol. 3, 2000, pp. 16–38.

## See Also

BurrDistribution

## More About

- “Working with Probability Distributions” on page 5-3
- “Supported Distributions” on page 5-17

## Chi-Square Distribution

**In this section...**

“Overview” on page B-28

“Parameters” on page B-28

“Probability Density Function” on page B-28

“Cumulative Distribution Function” on page B-28

“Descriptive Statistics” on page B-29

“Relationship to Other Distributions” on page B-29

“Examples” on page B-29

### Overview

The chi-square distribution is commonly used in hypothesis testing, particularly the chi-squared test for goodness of fit.

### Parameters

The chi-square distribution uses the following parameter.

Parameter	Description	Support
$\nu$	Degrees of freedom	$\nu$ is a positive value

### Probability Density Function

The probability density function (pdf) is

$$y = f(x | \nu) = \frac{x^{(\nu-2)/2} e^{-x/2}}{2^{\nu/2} \Gamma(\nu/2)}$$

where  $\Gamma(\cdot)$  is the Gamma function,  $\nu$  is the degrees of freedom, and  $x \geq 0$ .

### Cumulative Distribution Function

The cumulative distribution function (cdf) is



$$p = F(x | v) = \int_0^x \frac{t^{(v-2)/2} e^{-t/2}}{2^{v/2} \Gamma(v/2)} dt$$

where  $\Gamma(\cdot)$  is the Gamma function,  $v$  is the degrees of freedom, and  $x \geq 0$ .

## Descriptive Statistics

The mean is  $v$ .

The variance is  $2v$ .

## Relationship to Other Distributions

The  $\chi^2$  distribution is a special case of the gamma distribution where  $b = 2$  in the equation for gamma distribution below.

$$y = f(x | a, b) = \frac{1}{b^a \Gamma(a)} x^{a-1} e^{-x/b}$$

The  $\chi^2$  distribution gets special attention because of its importance in normal sampling theory. If a set of  $n$  observations is normally distributed with variance  $\sigma^2$ , and  $s^2$  is the sample standard deviation, then

$$\frac{(n-1)s^2}{\sigma^2} \sim \chi^2(n-1)$$

This relationship is used to calculate confidence intervals for the estimate of the normal parameter  $\sigma^2$  in the function `normfit`.

## Examples

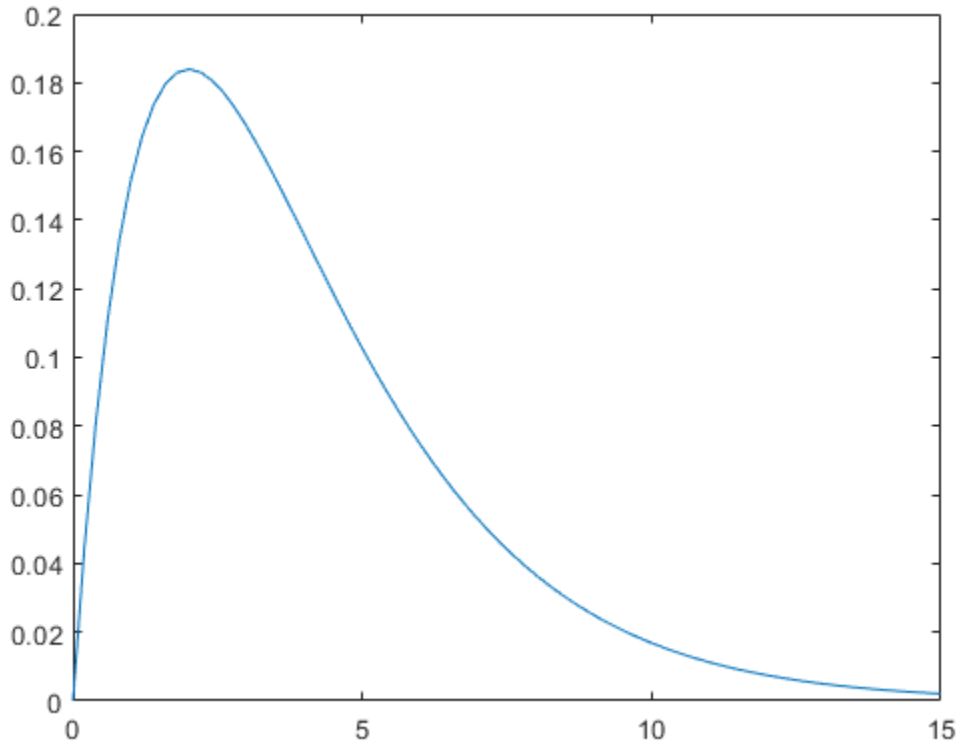
### Compute Chi-Square Distribution pdf

Compute the pdf of a chi-square distribution with 4 degrees of freedom.

```
x = 0:0.2:15;
y = chi2pdf(x, 4);
```

Plot the pdf.

```
figure;  
plot(x,y)
```



The chi-square distribution is skewed to the right, especially for few degrees of freedom.

## See Also

[chi2cdf](#) | [chi2gof](#) | [chi2inv](#) | [chi2pdf](#) | [chi2rnd](#) | [chi2stat](#) | [random](#)

## More About

- “Working with Probability Distributions” on page 5-3

- “Supported Distributions” on page 5-17

## Copulas

See “Copulas: Generate Correlated Samples” on page 5-164.

## Custom Distributions

User-defined custom distributions, created using files and function handles, are supported by the Statistics and Machine Learning Toolbox functions `pdf`, `cdf`, `icdf`, and `mle`, and the Statistics and Machine Learning Toolbox Distribution Fitter app.

## Exponential Distribution

In this section...
“Definition” on page B-34
“Background” on page B-34
“Parameters” on page B-34
“Examples” on page B-35

### Definition

The exponential pdf is

$$y = f(x | \mu) = \frac{1}{\mu} e^{\frac{-x}{\mu}}$$

### Background

Like the chi-square distribution, the exponential distribution is a special case of the gamma distribution (obtained by setting  $a = 1$ )

$$y = f(x | a, b) = \frac{1}{b^a \Gamma(a)} x^{a-1} e^{-\frac{x}{b}}$$

where  $\Gamma(\cdot)$  is the Gamma function.

The exponential distribution is special because of its utility in modeling events that occur randomly over time. The main application area is in studies of lifetimes.

### Parameters

Suppose you are stress testing light bulbs and collecting data on their lifetimes. You assume that these lifetimes follow an exponential distribution. You want to know how long you can expect the average light bulb to last. Parameter estimation is the process of determining the parameters of the exponential distribution that fit this data best in some sense.

One popular criterion of goodness is to maximize the likelihood function. The likelihood has the same form as the exponential pdf above. But for the pdf, the parameters are known constants and the variable is  $x$ . The likelihood function reverses the roles of the variables. Here, the sample values (the  $x$ 's) are already observed. So they are the fixed constants. The variables are the unknown parameters. MLE involves calculating the values of the parameters that give the highest likelihood given the particular set of data.

The function `expfit` returns the MLEs and confidence intervals for the parameters of the exponential distribution. Here is an example using random numbers from the exponential distribution with  $\mu = 700$ .

```
lifetimes = exprnd(700,100,1);
[muhat, muc1] = expfit(lifetimes)
```

```
muhat =
    672.8207
```

```
muc1 =
    547.4338
    810.9437
```

The MLE for parameter  $\mu$  is 672, compared to the true value of 700. The 95% confidence interval for  $\mu$  goes from 547 to 811, which includes the true value.

In the life tests you do not know the true value of  $\mu$  so it is nice to have a confidence interval on the parameter to give a range of likely values.

## Examples

### Exponentially Distributed Lifetime Data

For exponentially distributed lifetimes, the probability that an item will survive an extra unit of time is independent of the current age of the item. The example shows a specific case of this special property.

```
l = 10:10:60;
lpd = l+0.1;
deltap = (expcdf(lpd,50)-expcdf(l,50))./(1-expcdf(l,50))

deltap =
    0.0020    0.0020    0.0020    0.0020    0.0020    0.0020
```

**Compute the Exponential Distribution pdf**

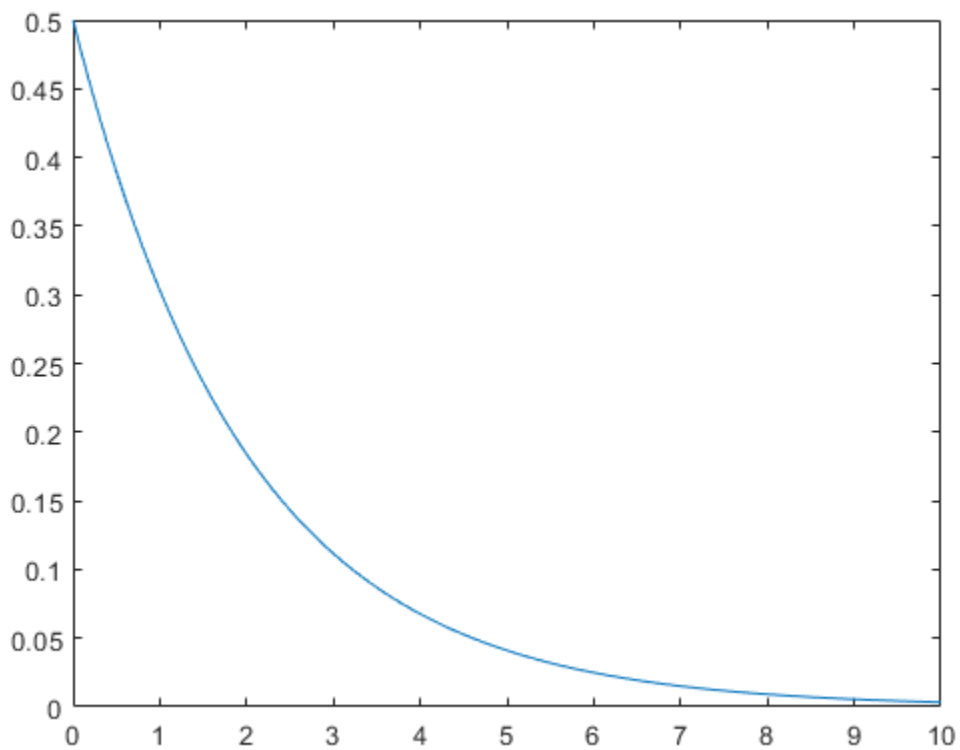
Compute the pdf of an exponential distribution with parameter  $\mu = 2$ .

```
x = 0:0.1:10;  
y = exppdf(x,2);
```

Plot the pdf.

```
figure;  
plot(x,y)
```





## See Also

`ExponentialDistribution`

## More About

- “Working with Probability Distributions” on page 5-3
- “Supported Distributions” on page 5-17

## Extreme Value Distribution

<b>In this section...</b>
“Definition” on page B-38
“Background” on page B-38
“Parameters” on page B-40
“Examples” on page B-41

### Definition

The probability density function for the extreme value distribution with location parameter  $\mu$  and scale parameter  $\sigma$  is

$$y = f(x | \mu, \sigma) = \sigma^{-1} \exp\left(\frac{x - \mu}{\sigma}\right) \exp\left(-\exp\left(\frac{x - \mu}{\sigma}\right)\right)$$

This form of the probability density function is suitable for modeling the minimum value. To model the maximum value, use the negative of the original values.

If  $T$  has a Weibull distribution on page B-193 with parameters  $a$  and  $b$ , then  $\log T$  has an extreme value distribution with parameters  $\mu = \log a$  and  $\sigma = 1/b$ .

### Background

Extreme value distributions are often used to model the smallest or largest value among a large set of independent, identically distributed random values representing measurements or observations. The extreme value distribution is appropriate for modeling the smallest value from a distribution whose tails decay exponentially fast, for example, the normal distribution. It can also model the largest value from a distribution, such as the normal or exponential distributions, by using the negative of the original values.

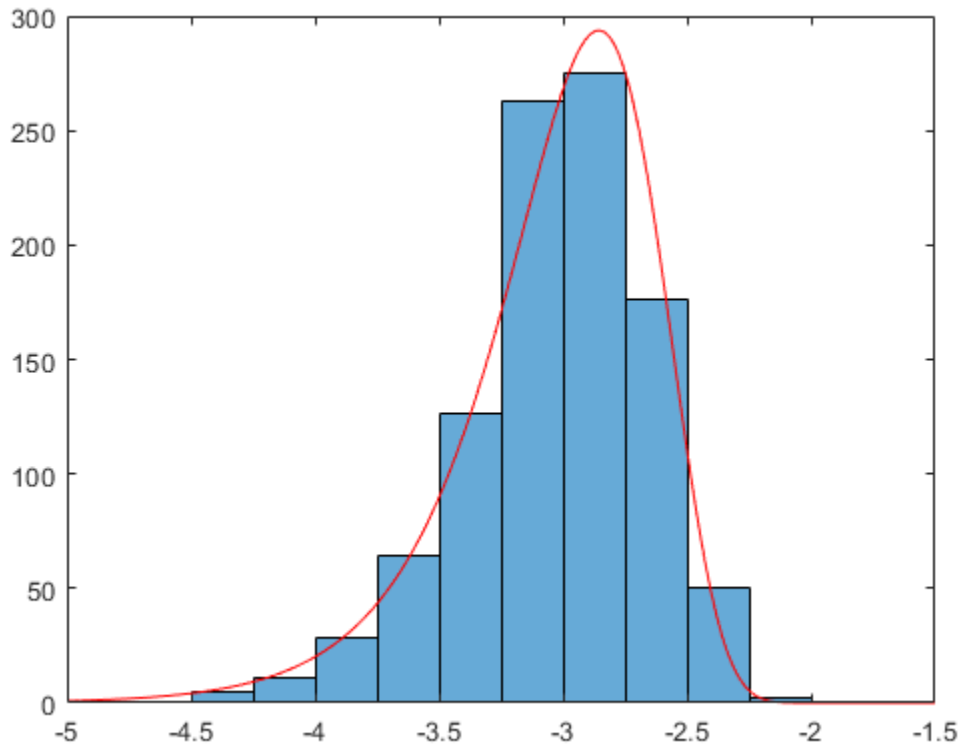
For example, the following fits an extreme value distribution to minimum values taken over 1000 sets of 500 observations from a normal distribution.

```
rng default; % For reproducibility
xMinima = min(randn(1000,500), [], 2);
paramEstsMinima = evfit(xMinima);
```

```

y = linspace(-5,-1.5,1001);
histogram(xMinima,-4.75:.25:-1.75);
p = evpdf(y,paramEstsMinima(1),paramEstsMinima(2));
line(y,.25*length(xMinima)*p,'color','r')

```



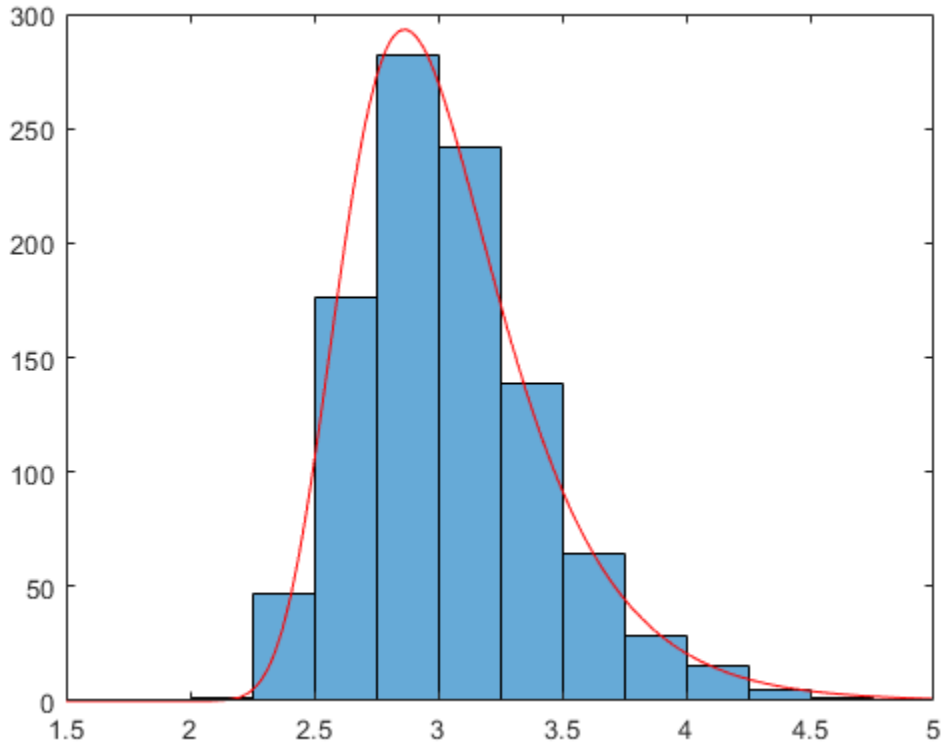
The following fits an extreme value distribution to the maximum values in each set of observations.

```

rng default; % For reproducibility
xMaxima = max(randn(1000,500), [], 2);
paramEstsMaxima = evfit(-xMaxima);
y = linspace(1.5,5,1001);
histogram(xMaxima,1.75:.25:4.75);

```

```
p = evpdf(-y,paramEstsMaxima(1),paramEstsMaxima(2));  
line(y,.25*length(xMaxima)*p,'color','r')
```



Although the extreme value distribution is most often used as a model for extreme values, you can also use it as a model for other types of continuous data. For example, extreme value distributions are closely related to the Weibull distribution. If  $T$  has a Weibull distribution, then  $\log(T)$  has a type 1 extreme value distribution.

## Parameters

The function `evfit` returns the maximum likelihood estimates (MLEs) and confidence intervals for the parameters of the extreme value distribution. The following example

shows how to fit some sample data using `evfit`, including estimates of the mean and variance from the fitted distribution.

Suppose you want to model the size of the smallest washer in each batch of 1000 from a manufacturing process. If you believe that the sizes are independent within and between each batch, you can fit an extreme value distribution to measurements of the minimum diameter from a series of eight experimental batches. The following code returns the MLEs of the distribution parameters as `parmhat` and the confidence intervals as the columns of `parmci`.

```
x = [19.774 20.141 19.44 20.511 21.377 19.003 19.66 18.83];
[parmhat, parmci] = evfit(x)

parmhat =
    20.2506    0.8223

parmci =
    19.644 0.49861
    20.857 1.3562
```

You can find mean and variance of the extreme value distribution with these parameters using the function `evstat`.

```
[meanfit, varfit] = evstat(parmhat(1),parmhat(2))

meanfit =
    19.776

varfit =
    1.1123
```

## Examples

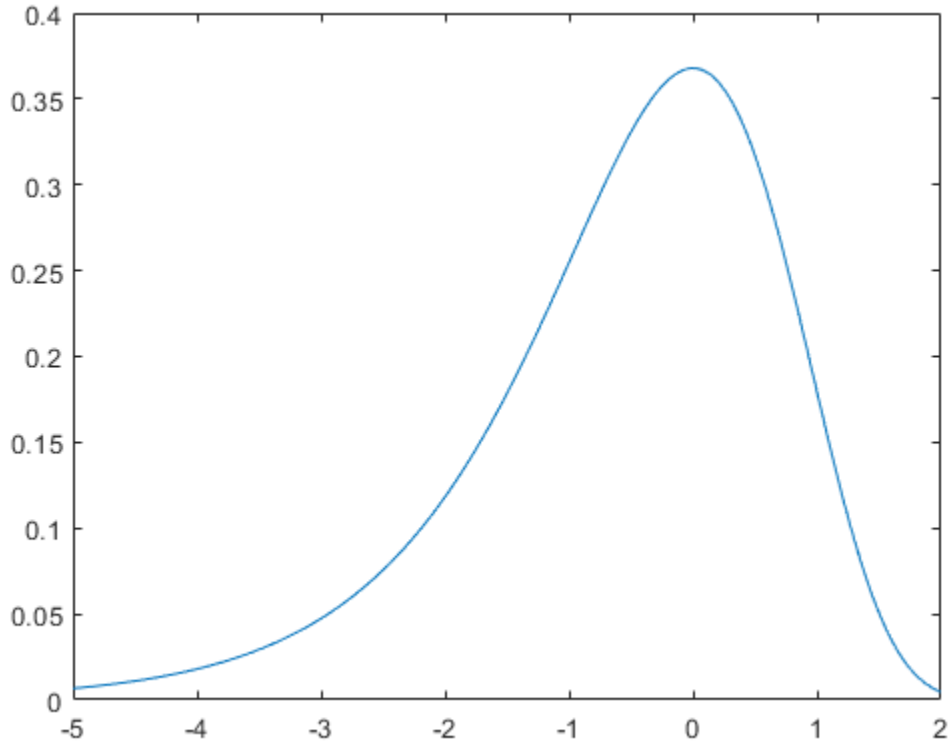
### Compute the Extreme Value Distribution pdf

Compute the pdf of an extreme value distribution.

```
t = [-5:.01:2];
y = evpdf(t);
```

Plot the pdf.

```
figure;
plot(t,y)
```

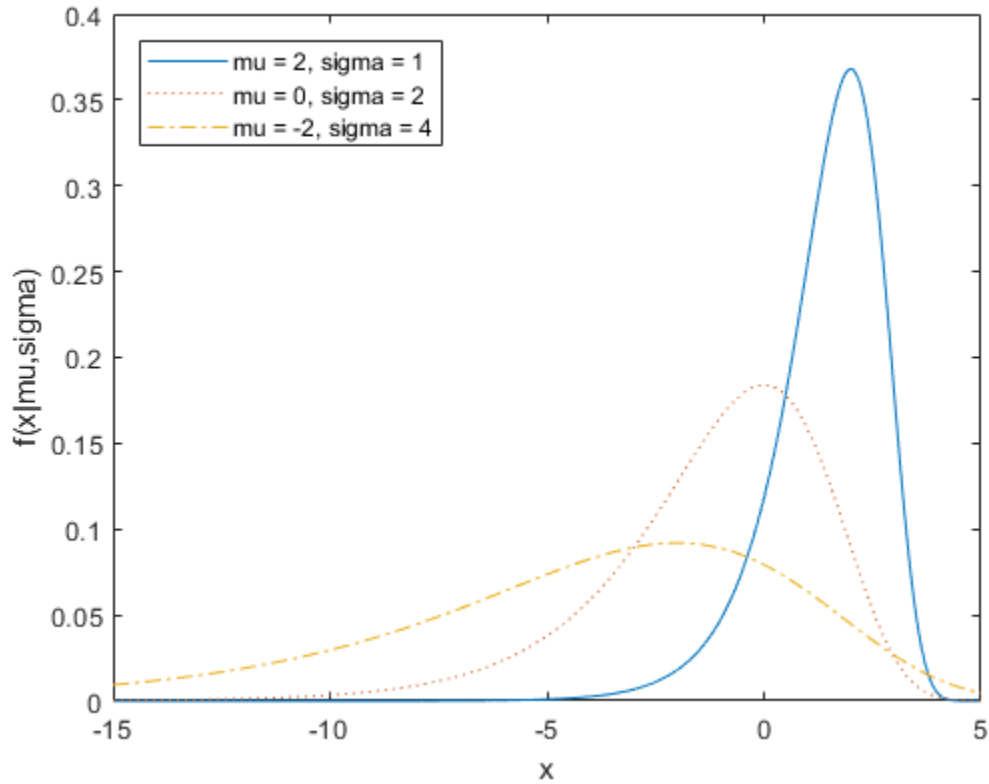


The extreme value distribution is skewed to the left, and its general shape remains the same for all parameter values. The location parameter,  $\mu$ , shifts the distribution along the real line, and the scale parameter,  $\sigma$ , expands or contracts the distribution.

The following plots the probability function for different combinations of  $\mu$  and  $\sigma$ .

```
x = -15:.01:5;
plot(x, evpdf(x, 2, 1), '- ', ...
      x, evpdf(x, 0, 2), ': ', ...
      x, evpdf(x, -2, 4), '-. ');
legend({'mu = 2, sigma = 1', ...
       'mu = 0, sigma = 2', ...
       'mu = -2, sigma = 4'}, ...
       'Location', 'NW')
```

```
xlabel('x')  
ylabel('f(x|mu,sigma)')
```



## See Also

`ExtremeValueDistribution`

## More About

- “Working with Probability Distributions” on page 5-3
- “Supported Distributions” on page 5-17

## F Distribution

In this section...
“Definition” on page B-44 “Background” on page B-44 “Examples” on page B-45

### Definition

The pdf for the  $F$  distribution is

$$y = f(x | \nu_1, \nu_2) = \frac{\Gamma\left[\frac{(\nu_1 + \nu_2)}{2}\right]}{\Gamma\left(\frac{\nu_1}{2}\right)\Gamma\left(\frac{\nu_2}{2}\right)} \left(\frac{\nu_1}{\nu_2}\right)^{\frac{\nu_1}{2}} \frac{x^{\frac{\nu_1-2}{2}}}{\left[1 + \left(\frac{\nu_1}{\nu_2}\right)x\right]^{\frac{\nu_1+\nu_2}{2}}}$$

where  $\Gamma(\cdot)$  is the Gamma function.

### Background

The  $F$  distribution has a natural relationship with the chi-square distribution. If  $\chi_1$  and  $\chi_2$  are both chi-square with  $\nu_1$  and  $\nu_2$  degrees of freedom respectively, then the statistic  $F$  below is  $F$ -distributed.

$$F(\nu_1, \nu_2) = \frac{\chi_1 / \nu_1}{\chi_2 / \nu_2}$$

The two parameters,  $\nu_1$  and  $\nu_2$ , are the numerator and denominator degrees of freedom. That is,  $\nu_1$  and  $\nu_2$  are the number of independent pieces of information used to calculate  $\chi_1$  and  $\chi_2$ , respectively.



## Examples

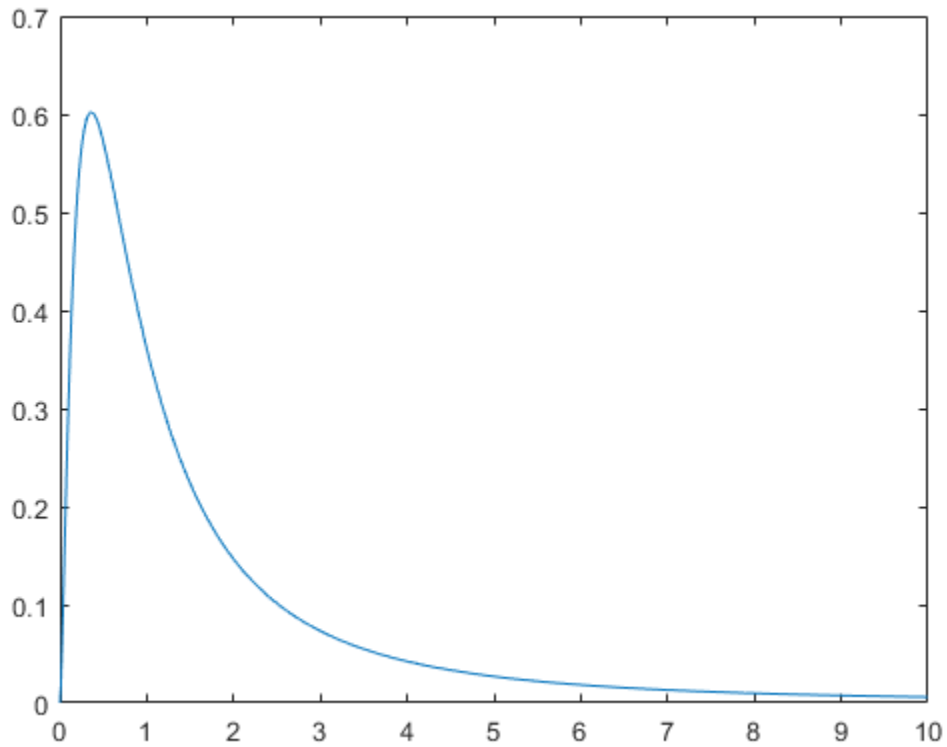
### Compute the F Distribution pdf

Compute the pdf of an  $F$  distribution with 5 numerator degrees of freedom and 3 denominator degrees of freedom.

```
x = 0:0.01:10;  
y = fpdf(x,5,3);
```

Plot the pdf.

```
figure;  
plot(x,y)
```



The plot shows that the  $F$  distribution exists on positive real numbers and is skewed to the right.

## See Also

`fcdf` | `finv` | `fpdf` | `frnd` | `fstat` | `random`

## More About

- “Working with Probability Distributions” on page 5-3
- “Supported Distributions” on page 5-17

# Gamma Distribution

## In this section...

“Definition” on page B-47

“Background” on page B-47

“Parameters” on page B-47

“Examples” on page B-49

## Definition

The gamma pdf is

$$y = f(x | a, b) = \frac{1}{b^a \Gamma(a)} x^{a-1} e^{-\frac{x}{b}}$$

where  $\Gamma(\cdot)$  is the Gamma function,  $a$  is a shape parameter,  $b$  is a scale parameter.

## Background

The gamma distribution models sums of exponentially distributed random variables.

The gamma distribution family is based on two parameters. The chi-square and exponential distributions, which are children of the gamma distribution, are one-parameter distributions that fix one of the two gamma parameters.

The gamma distribution has the following relationship with the incomplete Gamma function.

$$f(x | a, b) = \text{gammainc}\left(\frac{x}{b}, a\right)$$

When  $a$  is large, the gamma distribution closely approximates a normal distribution with the advantage that the gamma distribution has density only for positive real numbers.

## Parameters

Suppose you are stress testing computer memory chips and collecting data on their lifetimes. You assume that these lifetimes follow a gamma distribution. You want to

know how long you can expect the average computer memory chip to last. Parameter estimation is the process of determining the parameters of the gamma distribution that fit this data best in some sense.

One popular criterion of goodness is to maximize the likelihood function. The likelihood has the same form as the gamma pdf above. But for the pdf, the parameters are known constants and the variable is  $x$ . The likelihood function reverses the roles of the variables. Here, the sample values (the  $x$ 's) are already observed. So they are the fixed constants. The variables are the unknown parameters. MLE involves calculating the values of the parameters that give the highest likelihood given the particular set of data.

The function `gamfit` returns the MLEs and confidence intervals for the parameters of the gamma distribution. Here is an example using random numbers from the gamma distribution with  $\alpha = 10$  and  $b = 5$ .

```
lifetimes = gamrnd(10,5,100,1);  
[phat, pci] = gamfit(lifetimes)
```

```
phat =
```

```
    10.9821    4.7258
```

```
pci =
```

```
    7.4001    3.1543  
   14.5640    6.2974
```

Note `phat(1) =  $\hat{a}$`  and `phat(2) =  $\hat{b}$` . The MLE for parameter  $a$  is 10.98, compared to the true value of 10. The 95% confidence interval for  $a$  goes from 7.4 to 14.6, which includes the true value.

Similarly the MLE for parameter  $b$  is 4.7, compared to the true value of 5. The 95% confidence interval for  $b$  goes from 3.2 to 6.3, which also includes the true value.

In the life tests you do not know the true value of  $a$  and  $b$  so it is nice to have a confidence interval on the parameters to give a range of likely values.

## Examples

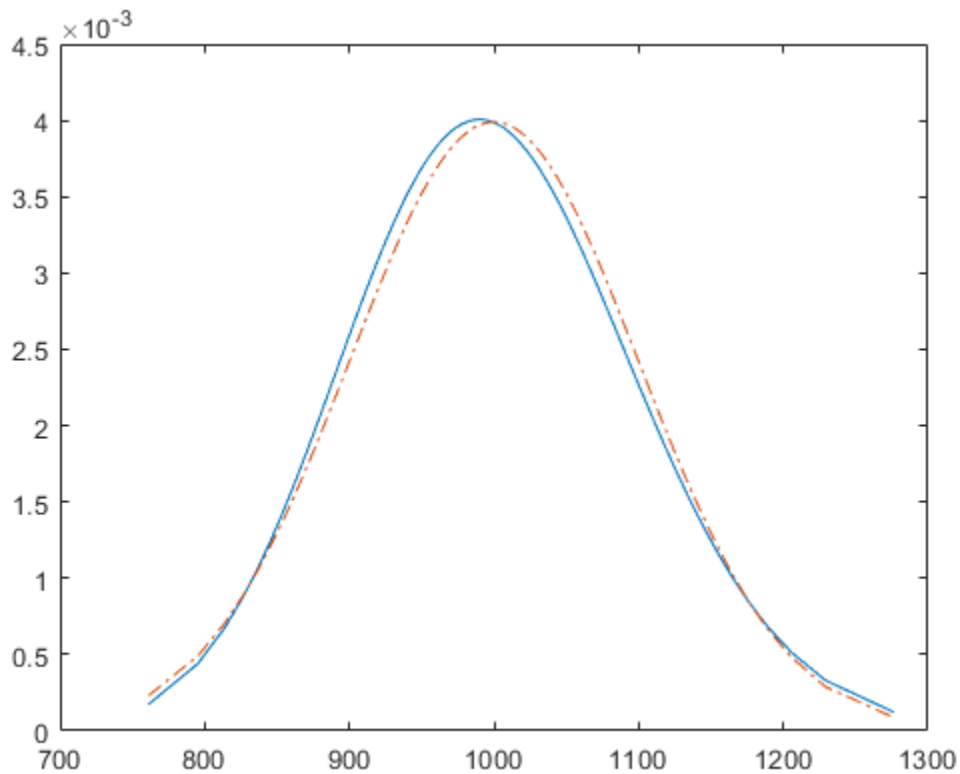
### Compute the Gamma Distribution pdf

Compute the pdf of a gamma distribution with parameters  $A = 100$  and  $B = 10$ . For comparison, also compute the pdf of a normal distribution with parameters  $\mu = 1000$  and  $\sigma = 100$ .

```
x = gaminv(0.005:0.01:0.995),100,10);  
y = gampdf(x,100,10);  
y1 = normpdf(x,1000,100);
```

Plot the pdfs of the gamma distribution and the normal distribution on the same figure.

```
figure;  
plot(x,y,'-',x,y1,'-.')
```



## See Also

`GammaDistribution`

## More About

- “Working with Probability Distributions” on page 5-3
- “Supported Distributions” on page 5-17

## Gaussian Distribution

See “Normal Distribution” on page B-134.

## Gaussian Mixture Distributions

See the discussion of the `gmdistribution` class in “Gaussian Mixture Models” on page 5-148.



## Generalized Extreme Value Distribution

### In this section...

“Definition” on page B-53

“Background” on page B-53

“Parameters” on page B-54

“Examples” on page B-55

### Definition

The probability density function for the generalized extreme value distribution with location parameter  $\mu$ , scale parameter  $\sigma$ , and shape parameter  $k \neq 0$  is

$$y = f(x | k, \mu, \sigma) = \left( \frac{1}{\sigma} \right) \exp \left( - \left( 1 + k \frac{(x - \mu)}{\sigma} \right)^{-\frac{1}{k}} \right) \left( 1 + k \frac{(x - \mu)}{\sigma} \right)^{-1 - \frac{1}{k}}$$

for

$$1 + k \frac{(x - \mu)}{\sigma} > 0$$

$k > 0$  corresponds to the Type II case, while  $k < 0$  corresponds to the Type III case. For  $k = 0$ , corresponding to the Type I case, the density is

$$y = f(x | 0, \mu, \sigma) = \left( \frac{1}{\sigma} \right) \exp \left( - \exp \left( - \frac{(x - \mu)}{\sigma} \right) - \frac{(x - \mu)}{\sigma} \right)$$

### Background

Like the extreme value distribution, the generalized extreme value distribution is often used to model the smallest or largest value among a large set of independent, identically distributed random values representing measurements or observations. For example, you might have batches of 1000 washers from a manufacturing process. If you record the size of the largest washer in each batch, the data are known as block maxima (or minima if you record the smallest). You can use the generalized extreme value distribution as a model for those block maxima.

The generalized extreme value combines three simpler distributions into a single form, allowing a continuous range of possible shapes that includes all three of the simpler distributions. You can use any one of those distributions to model a particular dataset of block maxima. The generalized extreme value distribution allows you to “let the data decide” which distribution is appropriate.

The three cases covered by the generalized extreme value distribution are often referred to as the Types I, II, and III. Each type corresponds to the limiting distribution of block maxima from a different class of underlying distributions. Distributions whose tails decrease exponentially, such as the normal, lead to the Type I. Distributions whose tails decrease as a polynomial, such as Student's  $t$ , lead to the Type II. Distributions whose tails are finite, such as the beta, lead to the Type III.

Types I, II, and III are sometimes also referred to as the Gumbel, Frechet, and Weibull types, though this terminology can be slightly confusing. The Type I (Gumbel) and Type III (Weibull) cases actually correspond to the mirror images of the usual Gumbel and Weibull distributions, for example, as computed by the functions `evcdf` and `evfit`, or `wblcdf` and `wblfit`, respectively. Finally, the Type II (Frechet) case is equivalent to taking the reciprocal of values from a standard Weibull distribution.

## Parameters

If you generate 250 blocks of 1000 random values drawn from Student's  $t$  distribution with 5 degrees of freedom, and take their maxima, you can fit a generalized extreme value distribution to those maxima.

```
blocksize = 1000;
nblocks = 250;
rng default % For reproducibility
t = trnd(5,blocksize,nblocks);
x = max(t); % 250 column maxima
paramEsts = gevfit(x)
```

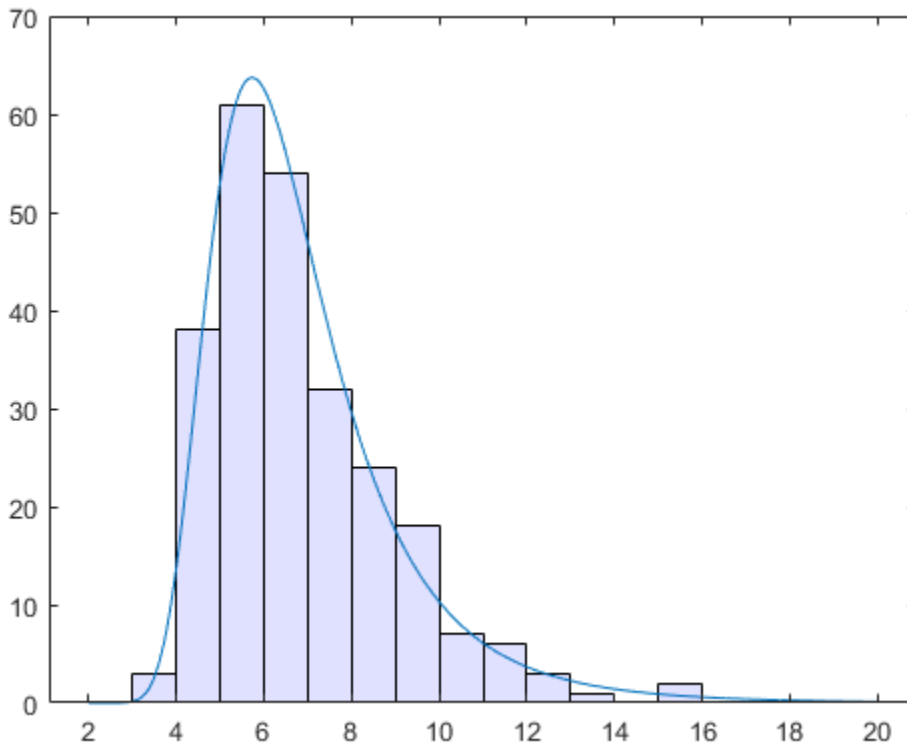
```
paramEsts =
    0.1185    1.4530    5.8929
```

Notice that the shape parameter estimate (the first element) is positive, which is what you would expect based on block maxima from a Student's  $t$  distribution.

```

histogram(x,2:20,'FaceColor',[.8 .8 1]);
xgrid = linspace(2,20,1000);
line(xgrid,nblocks*...
      gevpdf(xgrid,paramEsts(1),paramEsts(2),paramEsts(3)));

```



## Examples

### Compute the Generalized Extreme Value Distribution pdf

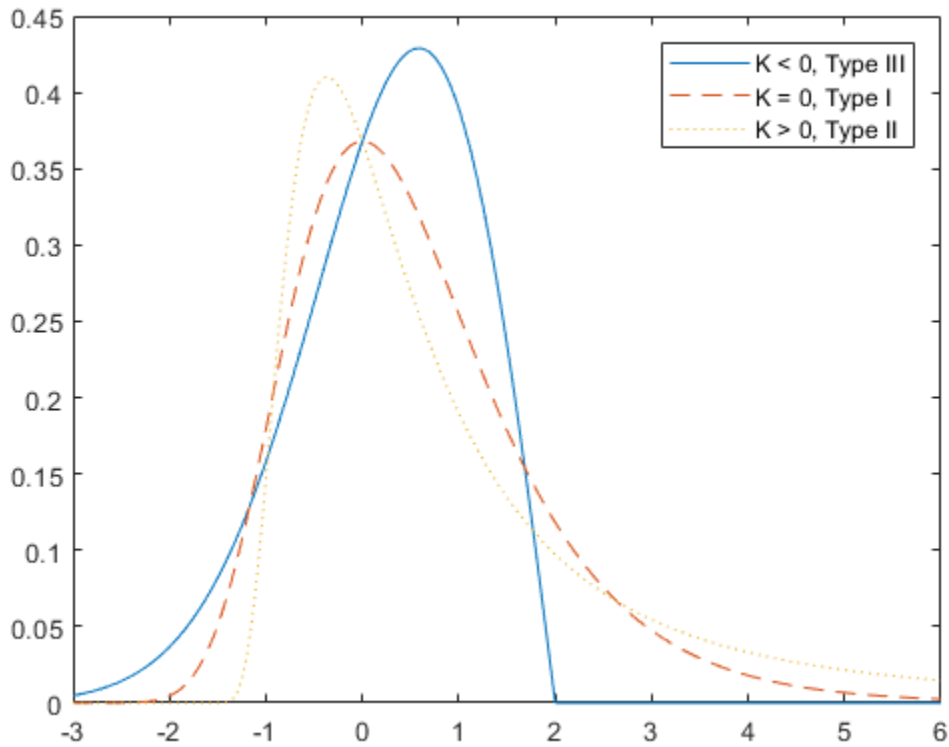
Generate examples of probability density functions for the three basic forms of the generalized extreme value distribution.

```

x = linspace(-3,6,1000);
y1 = gevpdf(x,-.5,1,0);

```

```
y2 = gevpdf(x,0,1,0);  
y3 = gevpdf(x,.5,1,0);  
plot(x,y1,'-', x,y2,'--', x,y3,':')  
legend({'K < 0, Type III' 'K = 0, Type I' 'K > 0, Type II'})
```



Notice that for  $k > 0$ , the distribution has zero probability density for  $x$  such that  $x < -\sigma/k + \mu$ .

For  $k < 0$ , the distribution has zero probability density for  $x > -\sigma/k + \mu$ .

For  $k = 0$ , there is no upper or lower bound.

## See Also

`GeneralizedExtremeValueDistribution`

## More About

- “Working with Probability Distributions” on page 5-3
- “Supported Distributions” on page 5-17

## Generalized Pareto Distribution

In this section...
“Definition” on page B-58
“Background” on page B-58
“Parameters” on page B-59
“Examples” on page B-61

### Definition

The probability density function for the generalized Pareto distribution with shape parameter  $k \neq 0$ , scale parameter  $\sigma$ , and threshold parameter  $\theta$ , is

$$y = f(x | k, \sigma, \theta) = \left( \frac{1}{\sigma} \right) \left( 1 + k \frac{(x - \theta)}{\sigma} \right)^{-1 - \frac{1}{k}}$$

for  $\theta < x$ , when  $k > 0$ , or for  $\theta < x < \theta - \sigma/k$  when  $k < 0$ .

For  $k = 0$ , the density is

$$y = f(x | 0, \sigma, \theta) = \left( \frac{1}{\sigma} \right) e^{-\frac{(x - \theta)}{\sigma}}$$

for  $\theta < x$ .

If  $k = 0$  and  $\theta = 0$ , the generalized Pareto distribution is equivalent to the exponential distribution. If  $k > 0$  and  $\theta = \sigma/k$ , the generalized Pareto distribution is equivalent to the Pareto distribution with a scale parameter equal to  $\sigma/k$  and a shape parameter equal to  $1/k$ .

### Background

Like the exponential distribution, the generalized Pareto distribution is often used to model the tails of another distribution. For example, you might have washers from a manufacturing process. If random influences in the process lead to differences in the sizes of the washers, a standard probability distribution, such as the normal, could be used to model those sizes. However, while the normal distribution might be a good model near its mode, it might not be a good fit to real data in the tails and a more complex

model might be needed to describe the full range of the data. On the other hand, only recording the sizes of washers larger (or smaller) than a certain threshold means you can fit a separate model to those tail data, which are known as *exceedences*. You can use the generalized Pareto distribution in this way, to provide a good fit to extremes of complicated data.

The generalized Pareto distribution allows a continuous range of possible shapes that includes both the exponential and Pareto distributions as special cases. You can use either of those distributions to model a particular dataset of exceedences. The generalized Pareto distribution allows you to “let the data decide” which distribution is appropriate.

The generalized Pareto distribution has three basic forms, each corresponding to a limiting distribution of exceedence data from a different class of underlying distributions.

- Distributions whose tails decrease exponentially, such as the normal, lead to a generalized Pareto shape parameter of zero.
- Distributions whose tails decrease as a polynomial, such as Student's *t*, lead to a positive shape parameter.
- Distributions whose tails are finite, such as the beta, lead to a negative shape parameter.

The generalized Pareto distribution is used in the tails of distribution fit objects of the `paretotails` on page 32-4570 class.

## Parameters

If you generate a large number of random values from a Student's *t* distribution with 5 degrees of freedom, and then discard everything less than 2, you can fit a generalized Pareto distribution to those exceedences.

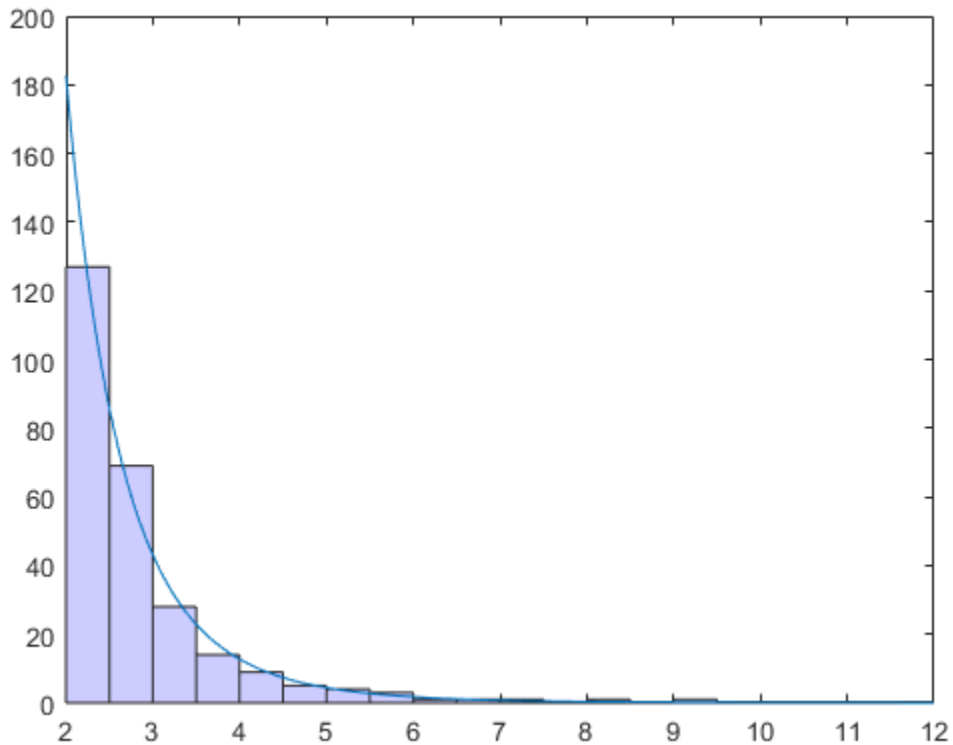
```
rng default % For reproducibility
t = trnd(5,5000,1);
y = t(t > 2) - 2;
paramEsts = gpfitt(y)
```

```
paramEsts =
```

```
    0.1445    0.7225
```

Notice that the shape parameter estimate (the first element) is positive, which is what you would expect based on exceedences from a Student's  $t$  distribution.

```
hist(y+2,2.25:.5:11.75);  
h = findobj(gca,'Type','patch');  
h.FaceColor = [.8 .8 1];  
xgrid = linspace(2,12,1000);  
line(xgrid,.5*length(y)*...  
      gppdf(xgrid,paramEsts(1),paramEsts(2),2));
```





## Examples

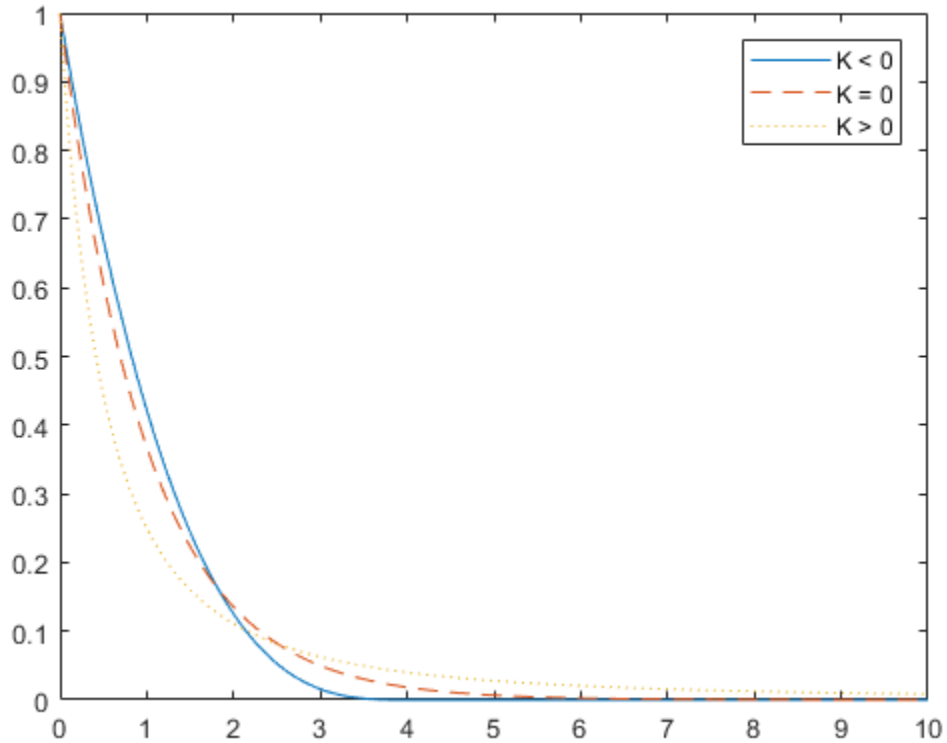
### Compute Generalized Pareto Distribution pdf

Compute the pdf of three generalized Pareto distributions. The first has shape parameter  $k = -0.25$ , the second has  $k = 0$ , and the third has  $k = 1$ .

```
x = linspace(0,10,1000);  
y1 = gppdf(x,-.25,1,0);  
y2 = gppdf(x,0,1,0);  
y3 = gppdf(x,1,1,0);
```

Plot the three pdfs on the same figure.

```
figure;  
plot(x,y1,'-', x,y2,'--', x,y3,':')  
legend({'K < 0' 'K = 0' 'K > 0'});
```



## See Also

`GeneralizedParetoDistribution`

## Related Examples

- “Fit a Nonparametric Distribution with Pareto Tails” on page 5-57

## More About

- “Nonparametric and Empirical Probability Distributions” on page 5-37

- “Working with Probability Distributions” on page 5-3
- “Supported Distributions” on page 5-17

## Geometric Distribution

In this section...
“Overview” on page B-64
“Parameters” on page B-64
“Probability Distribution Function” on page B-64
“Cumulative Distribution Function” on page B-67
“Mean and Variance” on page B-69
“Example” on page B-70

### Overview

The geometric distribution models the number of failures before one success in a series of independent trials, where each trial results in either success or failure, and the probability of success in any individual trial is constant. For example, if you toss a coin, the geometric distribution models the number of tails observed before getting a heads. The geometric distribution is discrete, existing only on the nonnegative integers.

### Parameters

The geometric distribution uses the following parameter.

Parameter	Description
$0 \leq p \leq 1$	Probability of success

### Probability Distribution Function

#### Definition

The probability distribution function (pdf) of the geometric distribution is

$$y = f(x | p) = p(1 - p)^x \quad ; \quad x = 0, 1, 2, \dots,$$

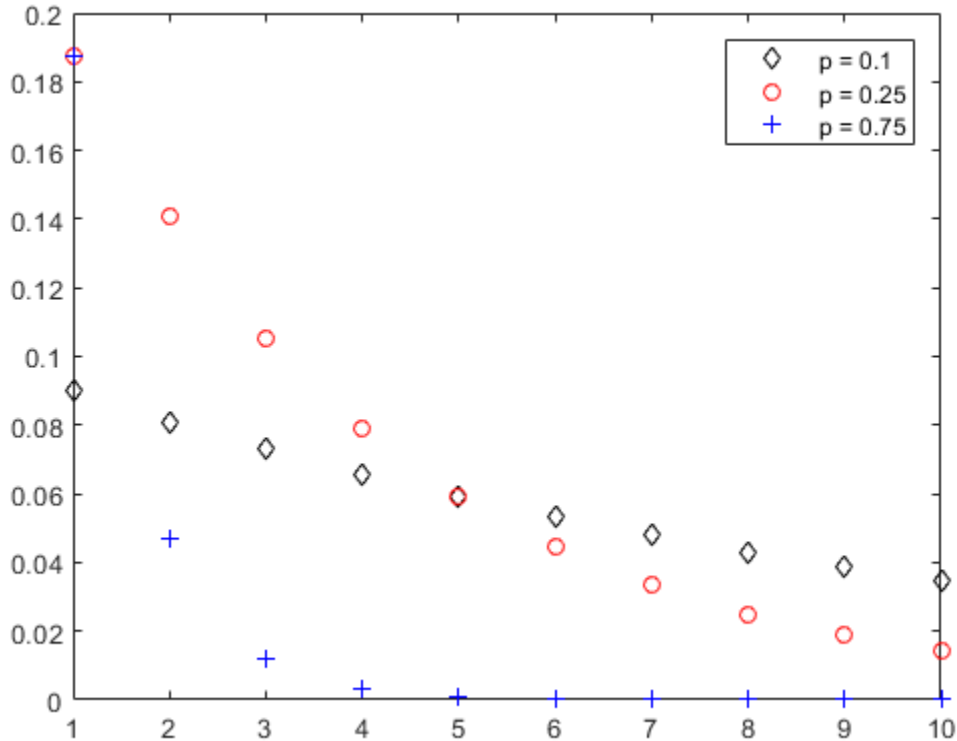
where  $p$  is the probability of success, and  $x$  is the number of failures before the first success. The result  $y$  is the probability of observing exactly  $x$  trials before a success, when the probability of success in any given trial is  $p$ . For discrete distributions, the probability distribution function is also known as the probability mass function (pmf).

## Plot

This plot shows how changing the value of the probability parameter  $p$  alters the shape of the pdf. Use `geopdf` to compute the pdf for values at  $x$  equals 1 through 10, for three different values of  $p$ . Then plot all three pdfs on the same figure for a visual comparison.

```
x = [1:10];
y1 = geopdf(x,0.1); % For p = 0.1
y2 = geopdf(x,0.25); % For p = 0.25
y3 = geopdf(x,0.75); % For p = 0.75

figure;
plot(x,y1,'kd')
hold on
plot(x,y2,'ro')
plot(x,y3,'b+')
legend({'p = 0.1','p = 0.25','p = 0.75'})
hold off
```



In this plot, the value of  $y$  is the probability of observing exactly  $x$  trials before a success. When the probability of success  $p$  is large,  $y$  decreases rapidly as  $x$  increases, and the probability of observing a large number of failures before a success quickly becomes small. But when the probability of success  $p$  is small,  $y$  decreases slowly as  $x$  increases. The probability of observing a large number of failures before a success still decreases as the number of trials increases, but at a much slower rate.

### Random Number Generation

A random number generated from a geometric distribution represents the number of failures observed before a success in a single experiment, given the probability of success  $p$  for each independent trial. Use `geornd` to generate random numbers from the

geometric distribution. For example, the following generates a random number from a geometric distribution with probability of success  $p$  equal to 0.1.

```
p = 0.1;
r = geornd(p)
```

```
r =
     1
```

The returned random number represents the number of failures observed before a success in a series of independent trials.

### Relationship to Other Distributions

The geometric distribution is a special case of the negative binomial distribution on page B-120, with the specified number of successes parameter  $r$  equal to 1.

## Cumulative Distribution Function

### Definition

The cumulative distribution function (cdf) of the geometric distribution is

$$y = F(x | p) = 1 - (1 - p)^{x+1}; \quad x = 0, 1, 2, \dots,$$

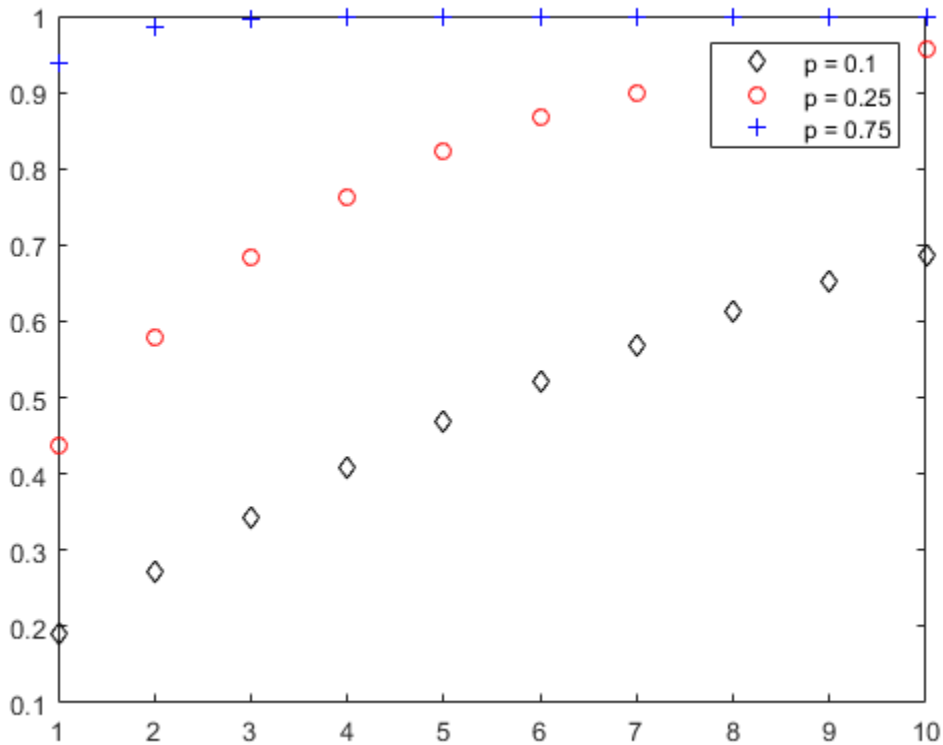
where  $p$  is the probability of success, and  $x$  is the number of failures before the first success. The result  $y$  is the probability of observing up to  $x$  trials before a success, when the probability of success in any given trial is  $p$ .

### Plot

This plot shows how changing the value of the parameter  $p$  alters the shape of the cdf. Use `geocdf` to compute the cdf values at  $x$  equals 1 through 10, for three different values of  $p$ . Then plot all three cdfs on the same figure for a visual comparison.

```
x = [1:10];
y1 = geocdf(x,0.1);    % For p = 0.1
y2 = geocdf(x,0.25);  % For p = 0.25
y3 = geocdf(x,0.75);  % For p = 0.75
```

```
figure;  
plot(x,y1,'kd')  
hold on  
plot(x,y2,'ro')  
plot(x,y3,'b+')  
legend({'p = 0.1', 'p = 0.25', 'p = 0.75'})  
hold off
```



In this plot, the value of  $y$  is the probability of observing up to  $x$  trials before a success. When the probability of success  $p$  is large,  $y$  increases rapidly as  $x$  increases. The probability of observing a success quickly becomes very high, even for a small number of trials. But when the probability of success  $p$  is small,  $y$  increases slowly as  $x$  increases. The probability of observing a success still increases as the number of trials increases, but at a much slower rate.



**Inverse cdf**

The inverse cdf of a geometric distribution determines the value of  $x$  that corresponds to a probability  $y$  of observing  $x$  successes in a row in independent trials. Use `geoinv` to compute the inverse cdf of the geometric distribution. For example, the following returns the smallest possible integer  $x$  such that the geometric cdf  $y$  evaluated at  $x$  is greater than or equal to 0.1, when the probability of success for each independent trial  $p$  is 0.03.

```
y = 0.1;
p = 0.03;
x = geoinv(y,p)
```

```
x =
```

```
3
```

**Mean and Variance**

The mean of the geometric distribution is

$$\text{mean} = \frac{1-p}{p},$$

and the variance of the geometric distribution is

$$\text{var} = \frac{1-p}{p^2},$$

where  $p$  is the probability of success.

Use `geostat` to compute the mean and variance of a geometric distribution. For example, the following computes the mean  $m$  and variance  $v$  of a geometric distribution with probability parameter  $p$  equal to 0.25.

```
p = 0.25;
[m,v] = geostat(p)
```

```
m =
```

```
3
```

```
y =  
12
```

## Example

### Compute Geometric Distribution Probabilities

Suppose the probability of a five-year-old car battery not starting in cold weather is 0.03. What is the probability of the car starting for 25 consecutive days during a long cold snap?

Model the scenario using a geometric distribution. In this case, the "failure" event is the car starting, and the "success" event is the car not starting. We want to determine the probability of observing 25 failures (the car starting) without observing a single success (the car not starting). The probability of success for each trial (the car not starting in any single attempt) is  $P = 0.03$ .

To solve, first compute the cumulative distribution function (cdf) for  $x = 25$  trials. This returns the probability of observing success (the car not starting) in up to 25 trials. Then subtract this result from 1 to determine the probability of *not* observing success in up to 25 trials - in other words, the probability that the car starts at every one of the 25 attempts.

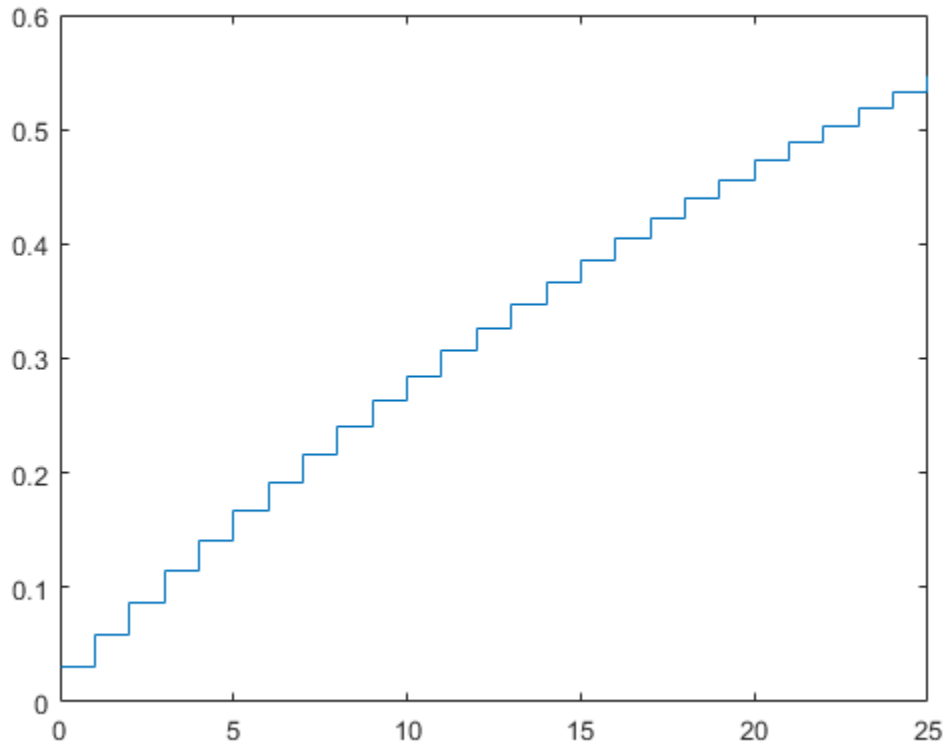
```
pstart = 1 - geocdf(25,0.03)
```

```
pstart =  
0.4530
```

The returned result `pstart = 0.4530` is the probability that the car will start every day for 25 days in a row during a cold snap.

This plot of the cdf for this scenario shows that, as the number of trials ( $x$ ) increases, the probability of success ( $y$ ) also increases. In the context of this example, it means that the more times you attempt to start the car, the greater the probability that it does not start on at least one of those occasions.

```
figure;  
x = 0:25;  
y = geocdf(x,0.03);  
stairs(x,y)
```



## See Also

[geocdf](#) | [geoinv](#) | [geopdf](#) | [geornd](#) | [geostat](#) | [random](#)

## More About

- “Working with Probability Distributions” on page 5-3

- “Supported Distributions” on page 5-17

## Half-Normal Distribution

In this section...
“Overview” on page B-73
“Parameters” on page B-73
“Probability Density Function” on page B-74
“Cumulative Distribution Function” on page B-75
“Descriptive Statistics” on page B-77
“Relationship to Other Distributions” on page B-78

### Overview

The half-normal distribution is a special case of the folded normal and truncated normal distributions. Some applications of the half-normal distribution include modeling measurement data and lifetime data.

### Parameters

The half-normal distribution uses the following parameters:

Parameter	Description
$-\infty < \mu < \infty$	Location parameter
$\sigma \geq 0$	Scale parameter

The support for the half-normal distribution is  $x \geq \mu$ .

Use `makedist` with specified parameter values to create a half-normal probability distribution object `HalfNormalDistribution`. Use `fitdist` to fit a half-normal probability distribution object to sample data. Use `mle` to estimate the half-normal distribution parameter values from sample data without creating a probability distribution object. For more information about working with probability distributions, see “Working with Probability Distributions” on page 5-3.

The Statistics and Machine Learning Toolbox implementation of the half-normal distribution assumes a fixed value for the location parameter  $\mu$ . Therefore, neither `fitdist` nor `mle` estimates the value of the parameter  $\mu$  when fitting a half-normal

distribution to sample data. You can specify a value for the  $\mu$  parameter by using the name-value pair argument 'mu'. The default value for the 'mu' argument is 0 in both `fitdist` and `mle`.

## Probability Density Function

The probability density function (pdf) of the half-normal distribution is

$$y = f(x | \mu, \sigma) = \sqrt{\frac{2}{\pi}} \frac{1}{\sigma} e^{-\frac{1}{2} \left( \frac{x-\mu}{\sigma} \right)^2}; x \geq \mu,$$

where  $\mu$  is the location parameter and  $\sigma$  is the scale parameter. If  $x \leq \mu$ , then the pdf is undefined.

To compute the pdf of the half-normal distribution, create a `HalfNormalDistribution` probability distribution object using `fitdist` or `makedist`, then use the `pdf` method to work with the object.

### PDF of Half-Normal Probability Distribution

This example shows how changing the values of the `mu` and `sigma` parameters alters the shape of the pdf.

Create four probability distribution objects with different parameters.

```
pd1 = makedist('HalfNormal');  
pd2 = makedist('HalfNormal','mu',0,'sigma',2);  
pd3 = makedist('HalfNormal','mu',0,'sigma',3);  
pd4 = makedist('HalfNormal','mu',0,'sigma',5);
```

Compute the probability density functions (pdfs) of each distribution.

```
x = 0:0.1:10;  
pdf1 = pdf(pd1,x);  
pdf2 = pdf(pd2,x);  
pdf3 = pdf(pd3,x);  
pdf4 = pdf(pd4,x);
```

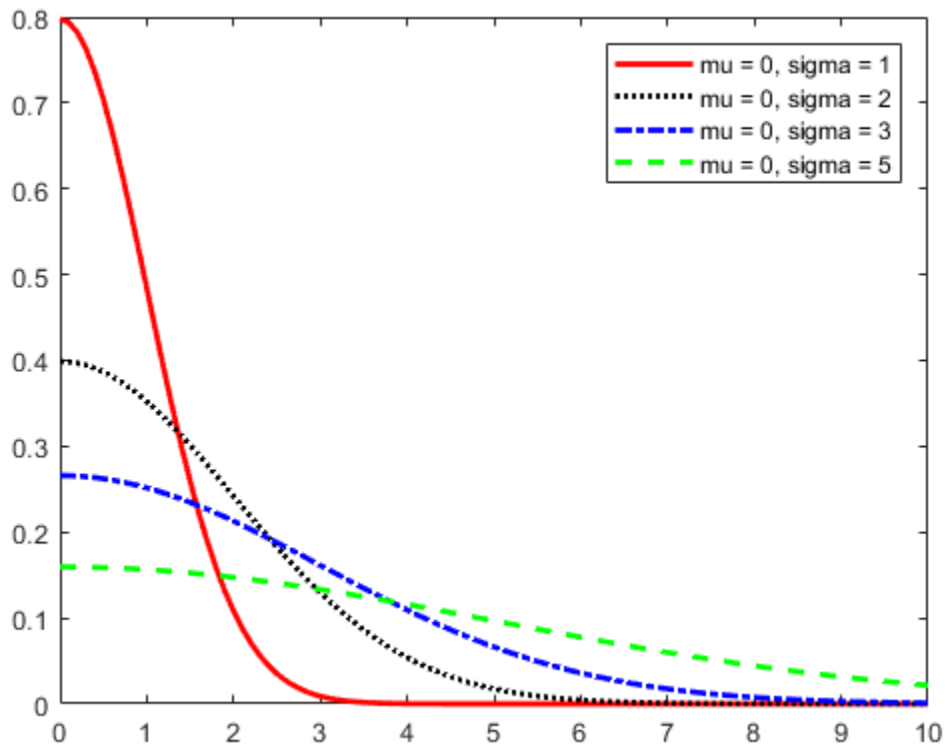
Plot the pdfs on the same figure.

```
figure;  
plot(x,pdf1,'r','LineWidth',2)
```

```

hold on;
plot(x,pdf2,'k:','LineWidth',2);
plot(x,pdf3,'b-.','LineWidth',2);
plot(x,pdf4,'g--','LineWidth',2);
legend({'mu = 0, sigma = 1','mu = 0, sigma = 2',...
      'mu = 0, sigma = 3','mu = 0, sigma = 5'},'Location','NE');
hold off;

```



As sigma increases, the curve flattens and the peak value becomes smaller.

## Cumulative Distribution Function

The cumulative distribution function (cdf) of the half-normal distribution is

$$y = F(x) = \operatorname{erf}\left(\frac{x - \mu}{\sqrt{2}\sigma}\right) = 2\Phi\left(\frac{x - \mu}{\sigma}\right) - 1; x \geq \mu,$$

where  $\mu$  is the location parameter,  $\sigma$  is the scale parameter,  $\operatorname{erf}(\cdot)$  is the error function, and  $\Phi(\cdot)$  is the cdf of the standard normal distribution. If  $x \leq \mu$ , then the cdf is undefined.

To compute the cdf of the half-normal distribution, create a `HalfNormalDistribution` probability distribution object using `fitdist` or `makedist`, then use the `cdf` method to work with the object.

### CDF of Half-Normal Probability Distribution

This example shows how changing the values of the `mu` and `sigma` parameters alters the shape of the cdf.

Create four probability distribution objects with different parameters.

```
pd1 = makedist('HalfNormal');  
pd2 = makedist('HalfNormal','mu',0,'sigma',2);  
pd3 = makedist('HalfNormal','mu',0,'sigma',3);  
pd4 = makedist('HalfNormal','mu',0,'sigma',5);
```

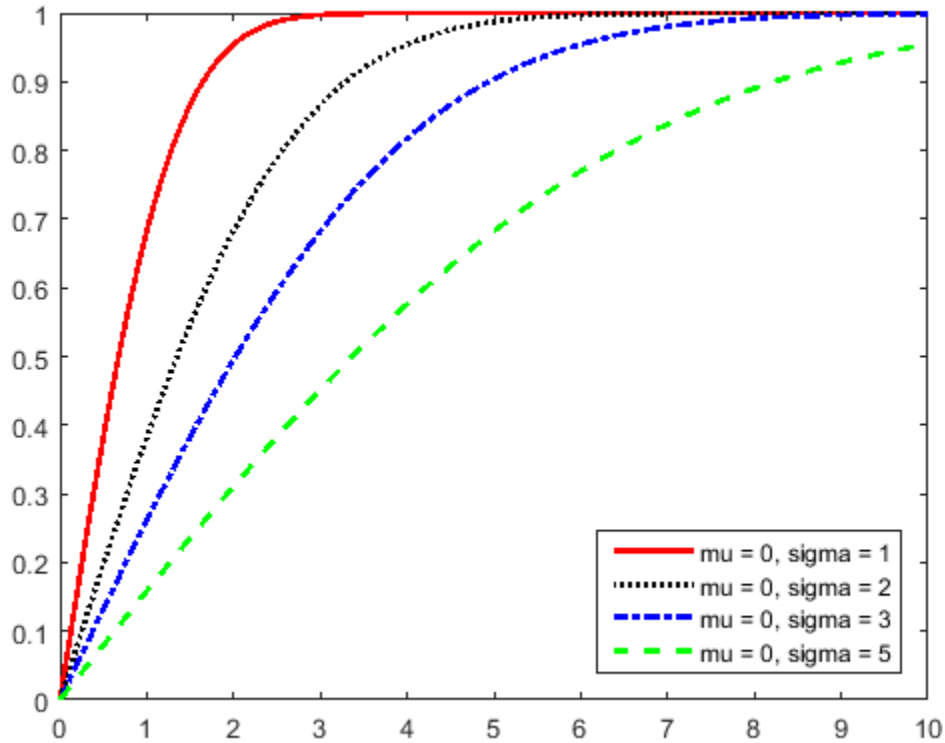
Compute the cumulative distribution functions (cdfs) for each probability distribution.

```
x = 0:0.1:10;  
cdf1 = cdf(pd1,x);  
cdf2 = cdf(pd2,x);  
cdf3 = cdf(pd3,x);  
cdf4 = cdf(pd4,x);
```

Plot all four cdfs on the same figure.

```
figure;  
plot(x,cdf1,'r','LineWidth',2)  
hold on;  
plot(x,cdf2,'k','LineWidth',2);  
plot(x,cdf3,'b-.','LineWidth',2);  
plot(x,cdf4,'g--','LineWidth',2);  
legend({'mu = 0, sigma = 1','mu = 0, sigma = 2',...  
       'mu = 0, sigma = 3','mu = 0, sigma = 5'},'Location','SE');  
hold off;
```





As sigma increases, the curve of the cdf flattens.

## Descriptive Statistics

The mean of the half-normal distribution is

$$\text{mean} = \mu + \sigma \sqrt{\frac{2}{\pi}},$$

where  $\mu$  is the location parameter and  $\sigma$  is the scale parameter.

The variance of the half-normal distribution is

$$\text{var} = \sigma^2 \left( 1 - \frac{2}{\pi} \right),$$

where  $\sigma$  is the scale parameter.

## Relationship to Other Distributions

If a random variable  $Z$  has a standard normal distribution with a mean  $\mu$  equal to zero and standard deviation  $\sigma$  equal to one, then  $X = \mu + \sigma|Z|$  has a half-normal distribution with parameters  $\mu$  and  $\sigma$ .

## References

- [1] Cooray, K. and M.M.A. Ananda. “A Generalization of the Half-Normal Distribution with Applications to Lifetime Data.” *Communications in Statistics – Theory and Methods*. Vol. 37, Number 9, 2008, pp. 1323–1337.
- [2] Pewsey, A. “Large-Sample Inference for the General Half-Normal Distribution.” *Communications in Statistics – Theory and Methods*. Vol. 31, Number 7, 2002, pp. 1045–1054.

## See Also

HalfNormalDistribution

## More About

- “Working with Probability Distributions” on page 5-3
- “Supported Distributions” on page 5-17

# Hypergeometric Distribution

**In this section...**

“Definition” on page B-79

“Background” on page B-79

“Examples” on page B-80

## Definition

The hypergeometric pdf is

$$y = f(x | M, K, n) = \frac{\binom{K}{x} \binom{M-K}{n-x}}{\binom{M}{n}}$$

## Background

The hypergeometric distribution models the total number of successes in a fixed-size sample drawn without replacement from a finite population.

The distribution is discrete, existing only for nonnegative integers less than the number of samples or the number of possible successes, whichever is greater. The hypergeometric distribution differs from the binomial only in that the population is finite and the sampling from the population is without replacement.

The hypergeometric distribution has three parameters that have direct physical interpretations.

- $M$  is the size of the population.
- $K$  is the number of items with the desired characteristic in the population.
- $n$  is the number of samples drawn.

Sampling “without replacement” means that once a particular sample is chosen, it is removed from the relevant population for all subsequent selections.

## Examples

### Compute and Plot Hypergeometric Distribution CDF

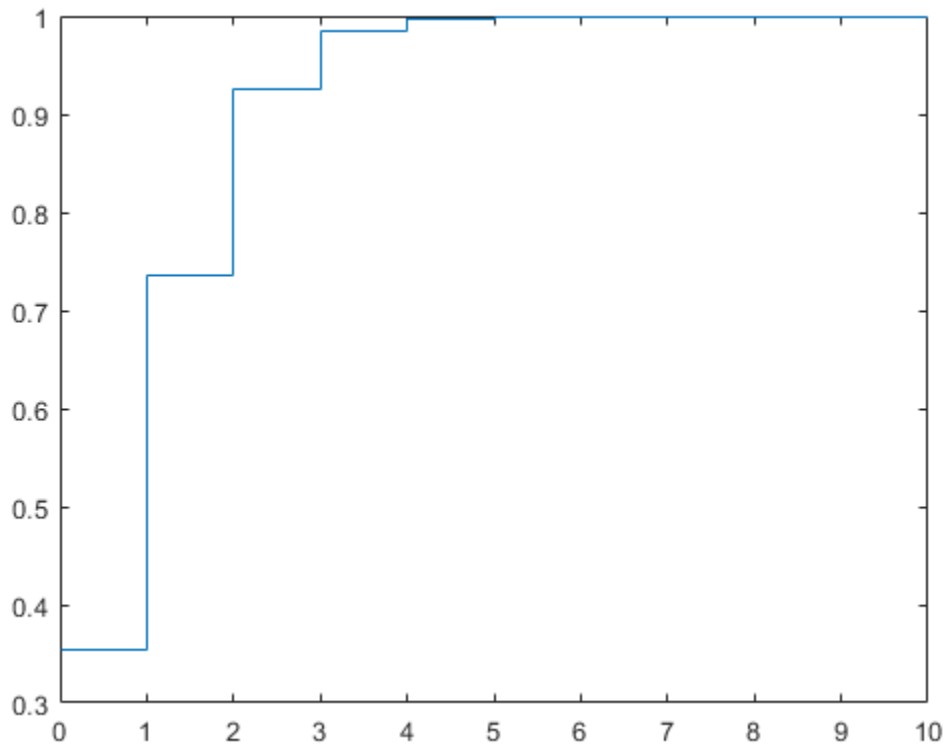
This example shows how to compute and plot the cdf of a hypergeometric distribution.

Compute the cdf of a hypergeometric distribution that draws 20 samples from a group of 1000 items, when the group contains 50 items of the desired type.

```
x = 0:10;  
y = hygecdf(x, 1000, 50, 20);
```

Plot the cdf.

```
stairs(x, y)
```



The x-axis of the plot shows the number of items drawn that are of the desired type. The y-axis shows the corresponding cdf values.

## See Also

`hygecdf` | `hygeinv` | `hygepdf` | `hygernd` | `hygestat` | `random`

## More About

- “Working with Probability Distributions” on page 5-3
- “Supported Distributions” on page 5-17

## Inverse Gaussian Distribution

In this section...
“Definition” on page B-82
“Background” on page B-82
“Parameters” on page B-82

### Definition

The inverse Gaussian distribution has the density function

$$\sqrt{\frac{\lambda}{2\pi x^3}} \exp\left\{-\frac{\lambda}{2\mu^2 x}(x - \mu)^2\right\}$$

### Background

Also known as the Wald distribution, the inverse Gaussian is used to model nonnegative positively skewed data. The distribution originated in the theory of Brownian motion, but has been used to model diverse phenomena. Inverse Gaussian distributions have many similarities to standard Gaussian (normal) distributions, which lead to applications in inferential statistics.

### Parameters

To estimate distribution parameters, use `mle` or the Distribution Fitter app.

### See Also

`InverseGaussianDistribution`

### More About

- “Working with Probability Distributions” on page 5-3
- “Supported Distributions” on page 5-17

## Inverse Wishart Distribution

### Definition

The probability density function of the  $d$ -dimensional Inverse Wishart distribution is given by

$$y = f(X, \Sigma, \nu) = \frac{|T|^{(\nu/2)} e^{\left(-\frac{1}{2}\text{trace}(TX^{-1})\right)}}{2^{(\nu d)/2} \pi^{(d(d-1))/4} |X|^{(\nu+d+1)/2} \Gamma(\nu/2) \dots \Gamma(\nu - (d-1)/2)},$$

where  $X$  and  $T$  are  $d$ -by- $d$  symmetric positive definite matrices, and  $\nu$  is a scalar greater than or equal to  $d$ . While it is possible to define the Inverse Wishart for singular  $T$ , the density cannot be written as above.

If a random matrix has a Wishart distribution with parameters  $T^{-1}$  and  $\nu$ , then the inverse of that random matrix has an inverse Wishart distribution with parameters  $T$  and  $\nu$ . The mean of the distribution is given by

$$\frac{1}{\nu - d - 1} T$$

where  $d$  is the number of rows and columns in  $T$ .

Only random matrix generation is supported for the inverse Wishart, including both singular and nonsingular  $T$ .

### Background

The inverse Wishart distribution is based on the Wishart distribution on page B-198. In Bayesian statistics it is used as the conjugate prior for the covariance matrix of a multivariate normal distribution.

### Example

Notice that the sampling variability is quite large when the degrees of freedom is small.

```
Tau = [1 .5; .5 2];
df = 10; S1 = iwishrnd(Tau, df) * (df-2-1)
```

```
S1 =
      1.7959      0.64107
      0.64107      1.5496

df = 1000; S2 = iwishrnd(Tau,df)*(df-2-1)

S2 =
      0.9842      0.50158
      0.50158      2.1682
```

## See Also

`iwishrnd`

## More About

- “Wishart Distribution” on page B-198
- “Working with Probability Distributions” on page 5-3
- “Supported Distributions” on page 5-17



## Johnson System

See “Pearson and Johnson Systems” on page 7-28.

## Kernel Distribution

In this section...
“Overview” on page B-86
“Kernel Density Estimator” on page B-86
“Kernel Smoothing Function” on page B-86
“Bandwidth” on page B-92

### Overview

A kernel distribution is a nonparametric representation of the probability density function (pdf) of a random variable. You can use a kernel distribution when a parametric distribution cannot properly describe the data, or when you want to avoid making assumptions about the distribution of the data. A kernel distribution is defined by a smoothing function and a bandwidth value, which control the smoothness of the resulting density curve.

### Kernel Density Estimator

The kernel density estimator is the estimated pdf of a random variable. For any real values of  $x$ , the kernel density estimator's formula is given by

$$\hat{f}_h(x) = \frac{1}{nh} \sum_{i=1}^n K\left(\frac{x - x_i}{h}\right),$$

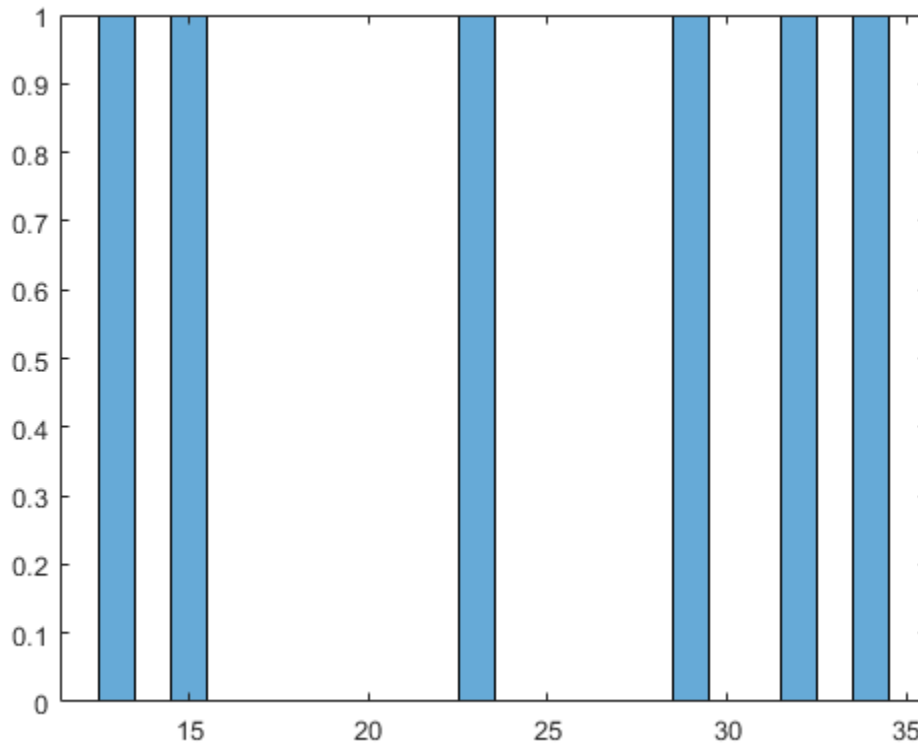
where  $x_1, x_2, \dots, x_n$  are random samples from an unknown distribution,  $n$  is the sample size,  $K(\cdot)$  is the kernel smoothing function, and  $h$  is the bandwidth.

### Kernel Smoothing Function

The kernel smoothing function defines the shape of the curve used to generate the pdf. Similar to a histogram, the kernel distribution builds a function to represent the probability distribution using the sample data. But unlike a histogram, which places the values into discrete bins, a kernel distribution sums the component smoothing functions for each data value to produce a smooth, continuous probability curve. The following plots show a visual comparison of a histogram and a kernel distribution generated from the same sample data.

A histogram represents the probability distribution by establishing bins and placing each data value in the appropriate bin.

```
SixMPG = [13;15;23;29;32;34];  
figure  
histogram(SixMPG)
```

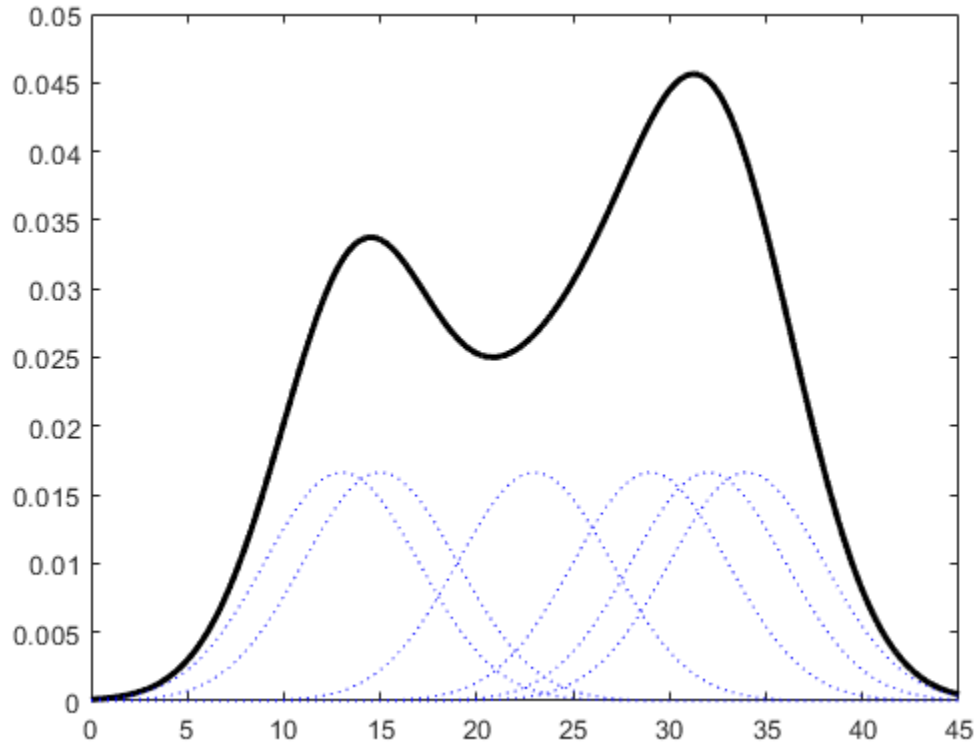


Because of this bin count approach, the histogram produces a discrete probability density function. This might be unsuitable for certain applications, such as generating random numbers from a fitted distribution.

Alternatively, the kernel distribution builds the pdf by creating an individual probability density curve for each data value, then summing the smooth curves. This approach creates one smooth, continuous probability density function for the data set.

```
figure
pdSix = fitdist(SixMPG, 'Kernel', 'BandWidth', 4);
x = 0:.1:45;
ySix = pdf(pdSix, x);
plot(x, ySix, 'k-', 'LineWidth', 2)

% Plot each individual pdf and scale its appearance on the plot
hold on
for i=1:6
    pd = makedist('Normal', 'mu', SixMPG(i), 'sigma', 4);
    y = pdf(pd, x);
    y = y/6;
    plot(x, y, 'b:')
end
hold off
```



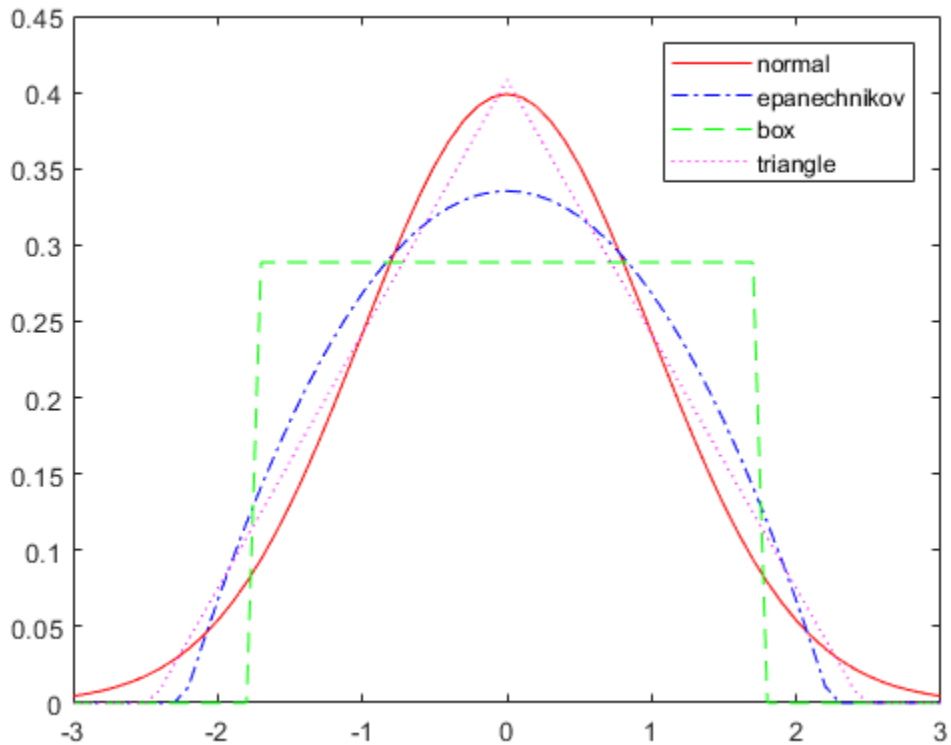
The smaller dashed curves are the probability distributions for each value in the sample data, scaled to fit the plot. The larger solid curve is the overall kernel distribution of the `SixMPG` data. The kernel smoothing function refers to the shape of those smaller component curves, which have a normal distribution in this example.

You can choose one of several options for the kernel smoothing function. This plot shows the shapes of the available smoothing functions.

Set plot specifications

```
hname = {'normal' 'epanechnikov' 'box' 'triangle'};
colors = {'r' 'b' 'g' 'm'};
lines = {'-', '-.', '--', ':'};
```

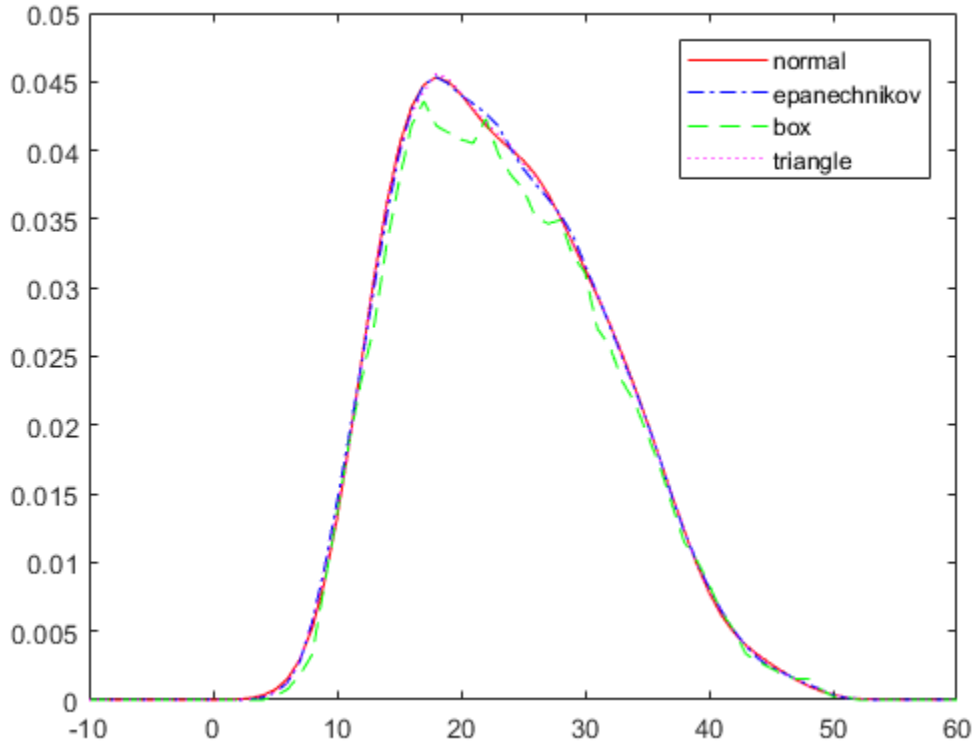
```
% Generate a sample of each kernel smoothing function and plot
data = [0];
figure
for j=1:4
    pd = fitdist(data,'kernel','Kernel',hname{j});
    x = -3:.1:3;
    y = pdf(pd,x);
    plot(x,y,'Color',colors{j},'LineStyle',lines{j})
    hold on;
end
legend(hname{:})
hold off
```



To understand the effect of different kernel smoothing functions on the shape of the resulting pdf estimate, compare plots of the mileage data (MPG) from `carbig.mat` using each available kernel function.

```
load carbig
% Set plot specifications
hname = {'normal' 'epanechnikov' 'box' 'triangle'};
colors = {'r' 'b' 'g' 'm'};
lines = {'-' '-.' '--' ':'};

% Generate kernel distribution objects and plot
figure
for j=1:4
    pd = fitdist(MPG, 'kernel', 'Kernel', hname{j});
    x = -10:1:60;
    y = pdf(pd, x);
    plot(x, y, 'Color', colors{j}, 'LineStyle', lines{j})
    hold on
end
legend(hname{:})
hold off
```



Each density curve uses the same input data, but applies a different kernel smoothing function to generate the pdf. The density estimates are roughly comparable, but the shape of each curve varies slightly. For example, the box kernel produces a density curve that is less smooth than the others.

## Bandwidth

The choice of bandwidth value controls the smoothness of the resulting probability density curve. This plot shows the density estimate for the same MPG data, using a normal kernel smoothing function with three different bandwidths.

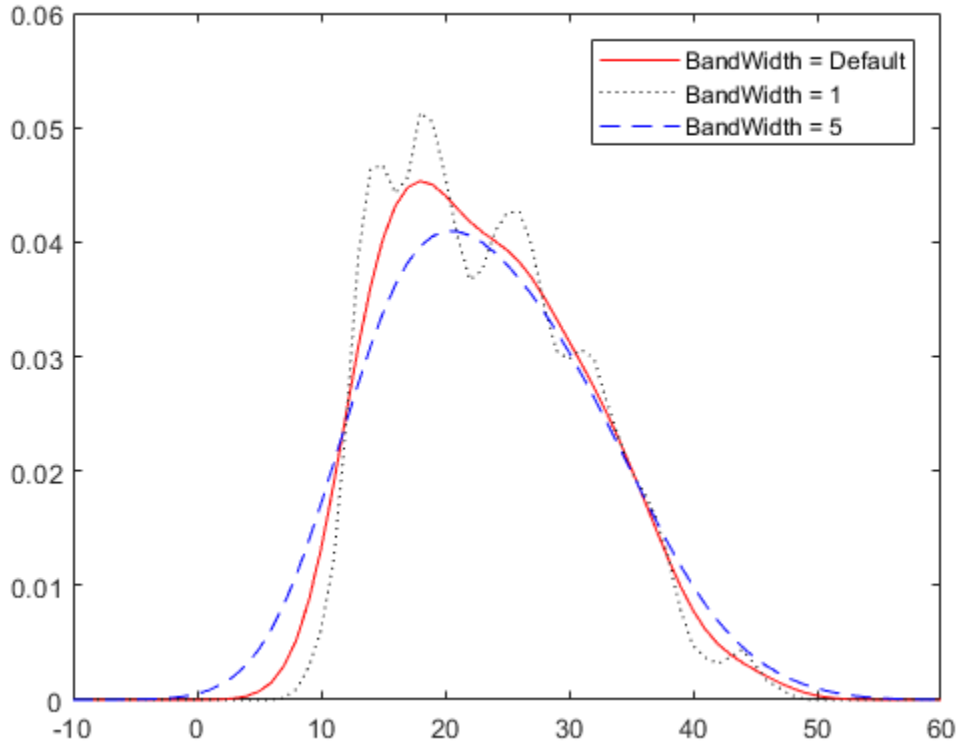
Create kernel distribution objects



```
pd1 = fitdist(MPG, 'kernel');
pd2 = fitdist(MPG, 'kernel', 'BandWidth', 1);
pd3 = fitdist(MPG, 'kernel', 'BandWidth', 5);

% Compute each pdf
x = -10:1:60;
y1 = pdf(pd1, x);
y2 = pdf(pd2, x);
y3 = pdf(pd3, x);

% Plot each pdf
plot(x, y1, 'Color', 'r', 'LineStyle', '-')
hold on
plot(x, y2, 'Color', 'k', 'LineStyle', ':')
plot(x, y3, 'Color', 'b', 'LineStyle', '--')
legend({'BandWidth = Default', 'BandWidth = 1', 'BandWidth = 5'})
hold off
```



The default bandwidth, which is theoretically optimal for estimating densities for the normal distribution, produces a reasonably smooth curve. Specifying a smaller bandwidth produces a very rough curve, but reveals that there might be two major peaks in the data. Specifying a larger bandwidth produces a curve nearly identical to the kernel function, and is so smooth that it obscures potentially important features of the data.

## See Also

`KernelDistribution` | `ksdensity`

## Related Examples

- “Fit Kernel Distribution Object to Data” on page 5-46
- “Fit Kernel Distribution Using ksdensity” on page 5-51
- “Fit Distributions to Grouped Data Using ksdensity” on page 5-54

## More About

- “Nonparametric and Empirical Probability Distributions” on page 5-37
- “Working with Probability Distributions” on page 5-3
- “Supported Distributions” on page 5-17

## Logistic Distribution

In this section...
“Overview” on page B-96
“Parameters” on page B-96
“Probability Density Function” on page B-96
“Relationship to Other Distributions” on page B-96

### Overview

The logistic distribution is used for growth models and in logistic regression. It has longer tails and a higher kurtosis than the normal distribution.

### Parameters

The logistic distribution uses the following parameters.

Parameter	Description	Support
mu	Mean	$-\infty < \mu < \infty$
sigma	Scale parameter	$\sigma \geq 0$

### Probability Density Function

The probability density function (pdf) is

$$f(x | \mu, \sigma) = \frac{\exp\left\{\frac{x - \mu}{\sigma}\right\}}{\sigma \left(1 + \exp\left\{\frac{x - \mu}{\sigma}\right\}\right)^2} ; \quad -\infty < x < \infty.$$

### Relationship to Other Distributions

The loglogistic distribution is closely related to the logistic distribution. If  $x$  is distributed loglogistically with parameters  $\mu$  and  $\sigma$ , then  $\log(x)$  is distributed logistically with mean and standard deviation.

## See Also

LogisticDistribution

## Related Examples

- “Compare Multiple Distribution Fits” on page 5-114

## More About

- “Working with Probability Distributions” on page 5-3
- “Supported Distributions” on page 5-17

## Loglogistic Distribution

In this section...
“Overview” on page B-98
“Parameters” on page B-98
“Probability Density Function” on page B-98
“Relationship to Other Distributions” on page B-99

### Overview

The loglogistic distribution is a probability distribution whose logarithm has a logistic distribution. This distribution is often used in survival analysis to model events that experience an initial rate increase, followed by a rate decrease. It is also known as the Fisk distribution in economics applications.

### Parameters

The loglogistic distribution uses the following parameters.

Parameter	Description	Support
mu	Log mean	$\mu > 0$
sigma	Log scale parameter	$\sigma > 0$

### Probability Density Function

The probability density function (pdf) is

$$f(x | \mu, \sigma) = \frac{1}{\sigma} \frac{1}{x} \frac{e^z}{(1 + e^z)^2} \quad ; \quad x \geq 0,$$

where  $z = \frac{\log(x) - \mu}{\sigma}$ .

## Relationship to Other Distributions

The loglogistic distribution is closely related to the logistic distribution. If  $x$  is distributed loglogistically with parameters  $\mu$  and  $\sigma$ , then  $\log(x)$  is distributed logistically with mean and standard deviation. The relationship is similar to that between the lognormal on page B-100 and normal on page B-134 distribution.

## See Also

`LoglogisticDistribution`

## More About

- “Working with Probability Distributions” on page 5-3
- “Supported Distributions” on page 5-17

## Lognormal Distribution

In this section...
“Overview” on page B-100
“Parameters” on page B-100
“Probability Density Function” on page B-100
“Descriptive Statistics” on page B-101
“Relationship to Other Distributions” on page B-101
“Examples” on page B-101

### Overview

The lognormal distribution is a probability distribution whose logarithm has a normal distribution. It is sometimes called the Galton distribution. The lognormal distribution is applicable when the quantity of interest must be positive, since  $\log(x)$  exists only when  $x$  is positive.

### Parameters

The lognormal distribution uses the following parameters.

Parameter	Description	Support
mu	Log mean	$-\infty < \mu < \infty$
sigma	Log standard deviation	$\sigma \geq 0$

### Probability Density Function

The probability density function (pdf) of the lognormal distribution is

$$f(x | \mu, \sigma) = \frac{1}{x\sigma\sqrt{2\pi}} \exp\left\{-\frac{(\ln x - \mu)^2}{2\sigma^2}\right\} ; \quad x > 0.$$



## Descriptive Statistics

The mean is

$$\text{mean} = \exp\left(\mu + \frac{\sigma^2}{2}\right).$$

The variance is

$$\text{var} = \exp(2\mu + \sigma^2)(\exp(\sigma^2) - 1).$$

You can compute these descriptive statistics using the `lognstat` function.

## Relationship to Other Distributions

The lognormal distribution is closely related to the normal distribution. If  $x$  is distributed lognormally with parameters  $\mu$  and  $\sigma$ , then  $\log(x)$  is distributed normally with mean  $\mu$  and standard deviation  $\sigma$ . The lognormal distribution is applicable when the quantity of interest must be positive, since  $\log(x)$  exists only when  $x$  is positive.

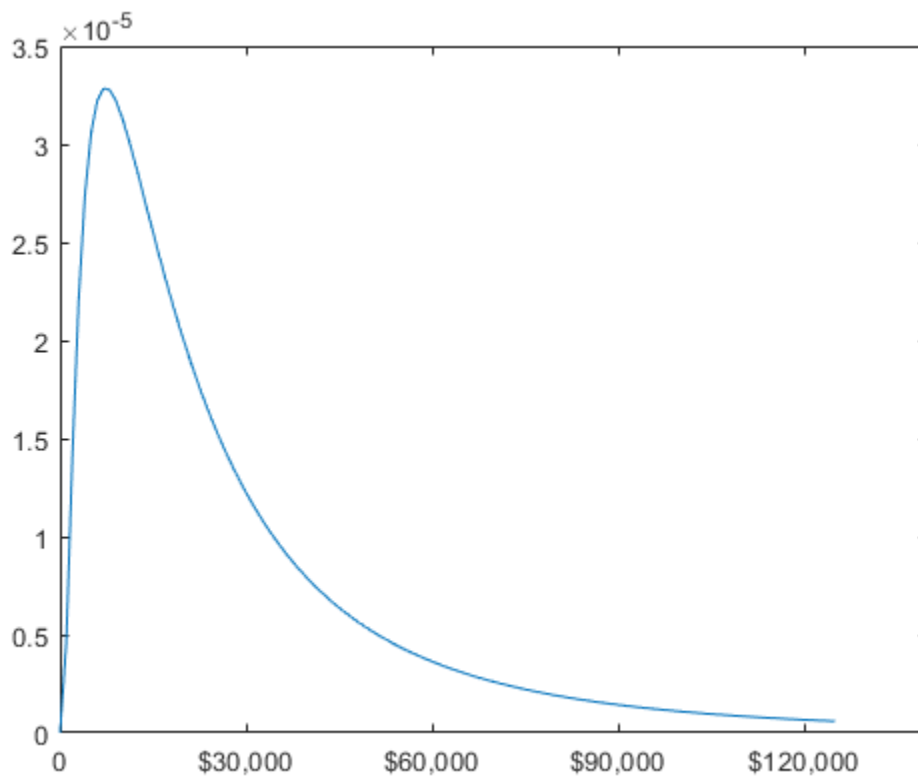
## Examples

### Compute the Lognormal Distribution pdf

Suppose the income of a family of four in the United States follows a lognormal distribution with `mu = log(20,000)` and `sigma = 1`. Compute and plot the income density.

```
x = (10:1000:125010)';
y = lognpdf(x, log(20000), 1.0);

figure;
plot(x, y)
h = gca;
h.XTick = [0 30000 60000 90000 120000];
h.XTickLabel = {'0', '$30,000', '$60,000', ...
               '$90,000', '$120,000'};
```



## See Also

`LognormalDistribution`

## More About

- “Working with Probability Distributions” on page 5-3
- “Supported Distributions” on page 5-17

## Multinomial Distribution

In this section...
“Overview” on page B-103
“Parameter” on page B-103
“Probability Density Function” on page B-103
“Descriptive Statistics” on page B-104
“Relationship to Other Distributions” on page B-104

### Overview

Multinomial distribution models the probability of each combination of successes in a series of independent trials. Use this distribution when there are more than two possible mutually exclusive outcomes for each trial, and each outcome has a fixed probability of success.

### Parameter

Multinomial distribution uses the following parameter.

Parameter	Description	Constraints
probabilities	Outcome probabilities	$0 \leq \text{probabilities}(i) \leq 1; \sum_{\text{all}(i)} \text{probabilities}(i) = 1$

### Probability Density Function

The multinomial pdf is

$$f(x | n, p) = \frac{n!}{x_1! \cdots x_k!} p_1^{x_1} \cdots p_k^{x_k},$$

where  $k$  is the number of possible mutually exclusive outcomes for each trial, and  $n$  is the total number of trials. The vector  $x = (x_1 \dots x_k)$  is the number of observations of each  $k$  outcome, and contains nonnegative integer components that sum to  $n$ . The vector  $p = (p_1 \dots p_k)$  is the fixed probability of each  $k$  outcome, and contains nonnegative scalar components that sum to 1.

## Descriptive Statistics

The expected number of observations of outcome  $i$  in  $n$  trials is

$$E\{x_i\} = np_i ,$$

where  $p_i$  is the fixed probability of outcome  $i$ .

The variance of outcome  $i$  is

$$\text{var}(x_i) = np_i(1 - p_i) .$$

The covariance of outcomes  $i$  and  $j$  is

$$\text{cov}(x_i, x_j) = -np_i p_j , i \neq j .$$

## Relationship to Other Distributions

The multinomial distribution is a generalization of the binomial distribution on page B-9. While the binomial distribution gives the probability of the number of “successes” in  $n$  independent trials of a two-outcome process, the multinomial distribution gives the probability of each combination of outcomes in  $n$  independent trials of a  $k$ -outcome process. The probability of each outcome in any one trial is given by the fixed probabilities  $p_1, \dots, p_k$ .

## See Also

MultinomialDistribution

## More About

- “Multinomial Probability Distribution Objects” on page 5-125
- “Multinomial Probability Distribution Functions” on page 5-129
- “Working with Probability Distributions” on page 5-3
- “Supported Distributions” on page 5-17

## Multivariate Gaussian Distribution

See “Multivariate Normal Distribution” on page B-106.

## Multivariate Normal Distribution

In this section...
“Definition” on page B-106
“Background” on page B-106
“Examples” on page B-107

### Definition

The probability density function of the  $d$ -dimensional multivariate normal distribution is given by

$$y = f(x, \mu, \Sigma) = \frac{1}{\sqrt{|\Sigma|} (2\pi)^d} \exp\left(-\frac{1}{2} (x-\mu) \Sigma^{-1} (x-\mu)'\right)$$

where  $x$  and  $\mu$  are 1-by- $d$  vectors and  $\Sigma$  is a  $d$ -by- $d$  symmetric positive definite matrix. While it is possible to define the multivariate normal for singular  $\Sigma$ , the density cannot be written as above. Only random vector generation is supported for the singular case. Note that while most textbooks define the multivariate normal with  $x$  and  $\mu$  oriented as column vectors, for the purposes of data analysis software, it is more convenient to orient them as row vectors, and Statistics and Machine Learning Toolbox software uses that orientation.

### Background

The multivariate normal distribution is a generalization of the univariate normal to two or more variables. It is a distribution for random vectors of correlated variables, each element of which has a univariate normal distribution. In the simplest case, there is no correlation among variables, and elements of the vectors are independent univariate normal random variables.

The multivariate normal distribution is parameterized with a mean vector,  $\mu$ , and a covariance matrix,  $\Sigma$ . These are analogous to the mean  $\mu$  and variance  $\sigma^2$  parameters of a univariate normal distribution. The diagonal elements of  $\Sigma$  contain the variances for each variable, while the off-diagonal elements of  $\Sigma$  contain the covariances between variables.

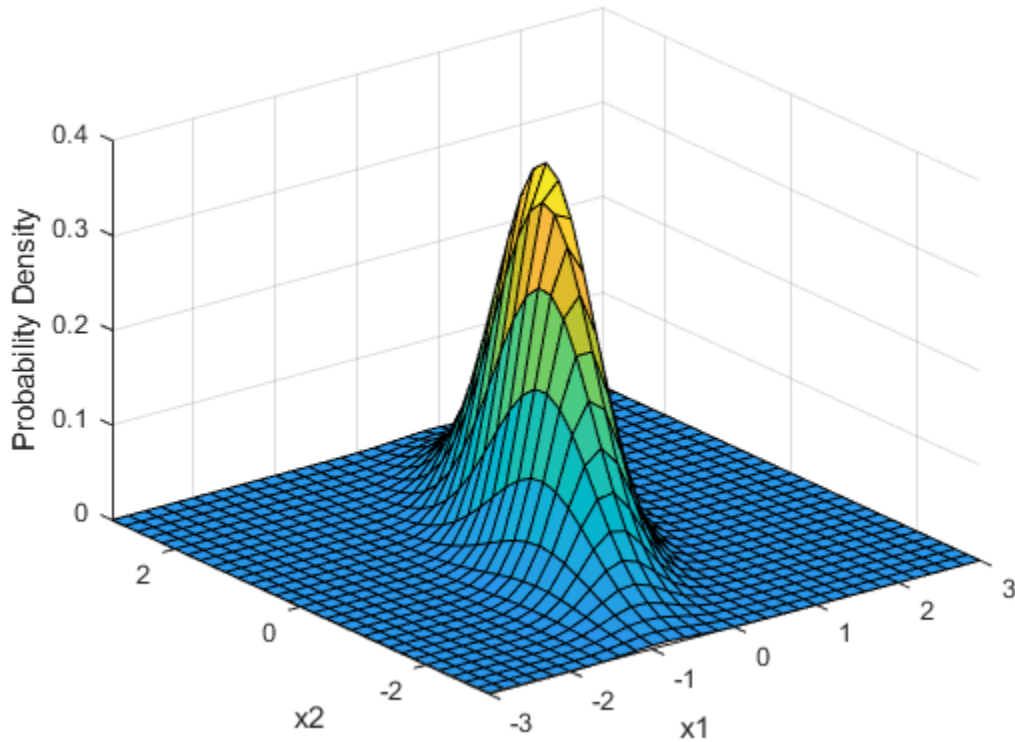
The multivariate normal distribution is often used as a model for multivariate data, primarily because it is one of the few multivariate distributions that is tractable to work with.

## Examples

### Compute the Multivariate Normal pdf

Compute and plot the pdf of a multivariate normal distribution.

```
mu = [0 0];
Sigma = [.25 .3; .3 1];
x1 = -3:.2:3; x2 = -3:.2:3;
[X1,X2] = meshgrid(x1,x2);
F = mvnpdf([X1(:) X2(:)],mu,Sigma);
F = reshape(F,length(x2),length(x1));
surf(x1,x2,F);
caxis([min(F(:))-.5*range(F(:)),max(F(:))]);
axis([-3 3 -3 3 0 .4])
xlabel('x1'); ylabel('x2'); zlabel('Probability Density');
```

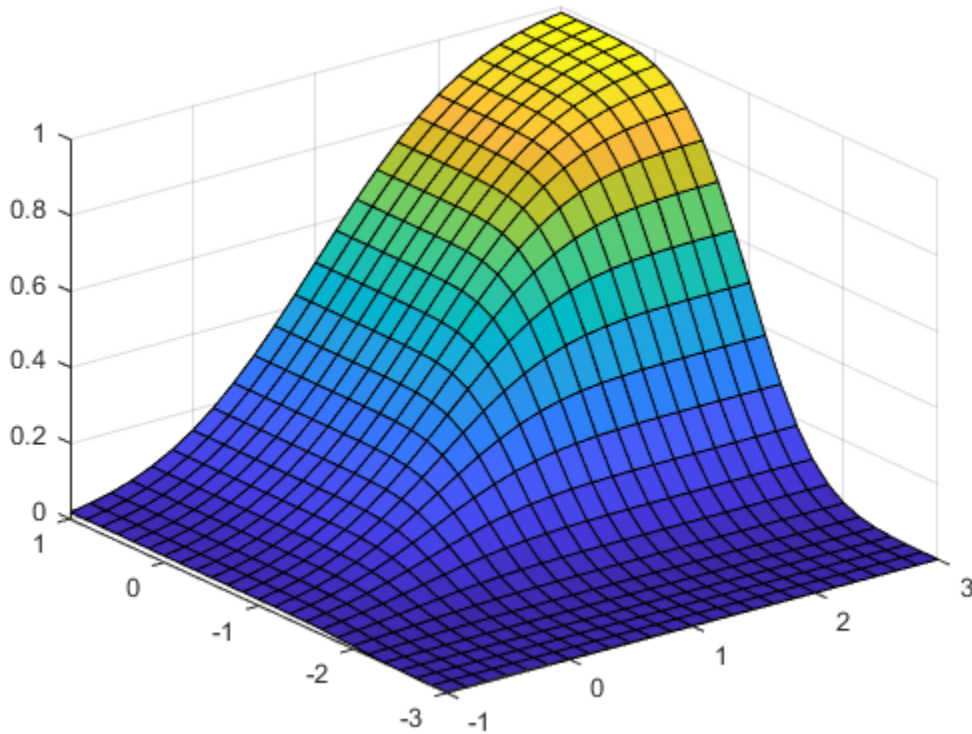


### Compute the Multivariate Normal cdf

Compute and plot the cdf of a multivariate normal distribution with parameters  $\mu = [1 \ -1]$  and  $\text{SIGMA} = \begin{bmatrix} .9 & .4 \\ .4 & .3 \end{bmatrix}$ .

```
mu = [1 -1];  
SIGMA = [.9 .4; .4 .3];  
figure;  
[X1,X2] = meshgrid(linspace(-1,3,25)',linspace(-3,1,25)');  
X = [X1(:) X2(:)];  
p = mvncdf(X,mu,SIGMA);  
surf(X1,X2,reshape(p,25,25));
```





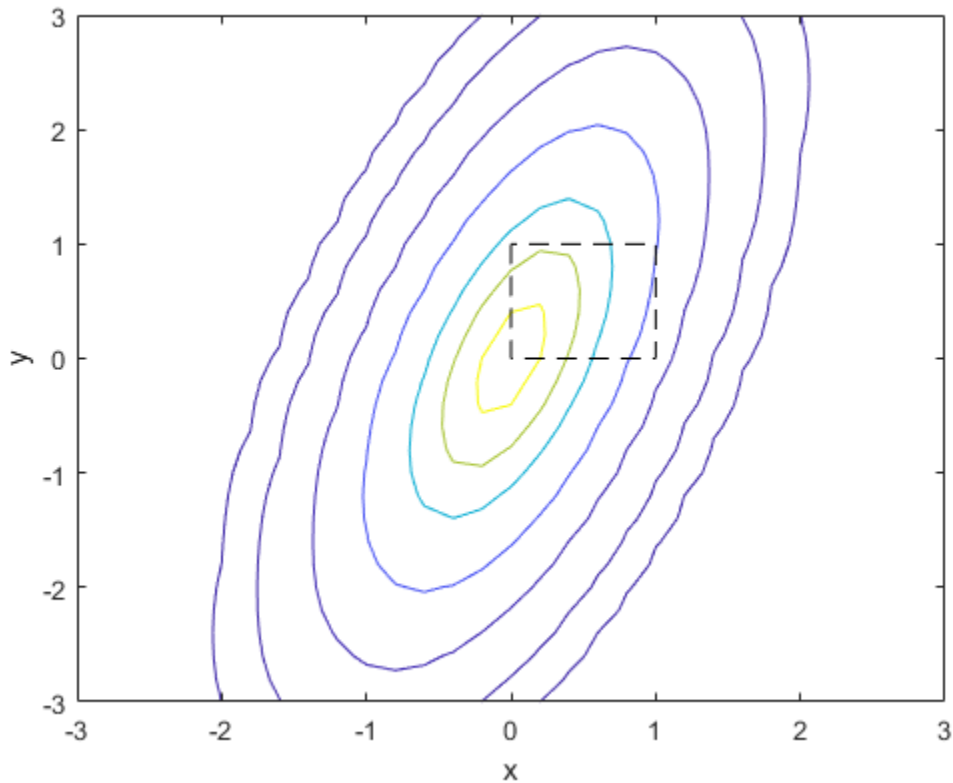
### Compute Cumulative Probabilities Over Regions

Since the bivariate normal distribution is defined on the plane, you can also compute cumulative probabilities over rectangular regions.

Compute the probability contained within the unit square, and create a contour plot of the results.

```
mu = [0 0];  
Sigma = [.25 .3; .3 1];  
x1 = -3:.2:3; x2 = -3:.2:3;  
[X1,X2] = meshgrid(x1,x2);  
F = mvnpdf([X1(:) X2(:)],mu,Sigma);  
F = reshape(F,length(x2),length(x1));
```

```
mvncdf([0 0],[1 1],mu,Sigma);  
contour(x1,x2,F,[.0001 .001 .01 .05:.1:.95 .99 .999 .9999]);  
xlabel('x'); ylabel('y');  
line([0 0 1 1 0],[1 0 0 1 1],'linestyle','--','color','k');
```



Computing a multivariate cumulative probability requires significantly more work than computing a univariate probability. By default, the `mvncdf` function computes values to less than full machine precision, and returns an estimate of the error as an optional second output.

```
[F,err] = mvncdf([0 0],[1 1],mu,Sigma)
```

F =

0.2097

err =

1.0000e-08

## See Also

[mvncdf](#) | [mvnpdf](#) | [mvnrnd](#)

## More About

- “Working with Probability Distributions” on page 5-3
- “Supported Distributions” on page 5-17

## Multivariate $t$ Distribution

In this section...
“Definition” on page B-112
“Background” on page B-112
“Example” on page B-113

### Definition

The probability density function of the  $d$ -dimensional multivariate Student's  $t$  distribution is given by

$$f(x, \Sigma, \nu) = \frac{1}{|\Sigma|^{1/2}} \frac{1}{\sqrt{(\nu\pi)^d}} \frac{\Gamma((\nu + d) / 2)}{\Gamma(\nu / 2)} \left( 1 + \frac{x' \Sigma^{-1} x}{\nu} \right)^{-(\nu+d)/2}.$$

where  $x$  is a 1-by- $d$  vector,  $\Sigma$  is a  $d$ -by- $d$  symmetric, positive definite matrix, and  $\nu$  is a positive scalar. While it is possible to define the multivariate Student's  $t$  for singular  $\Sigma$ , the density cannot be written as above. For the singular case, only random number generation is supported. Note that while most textbooks define the multivariate Student's  $t$  with  $x$  oriented as a column vector, for the purposes of data analysis software, it is more convenient to orient  $x$  as a row vector, and Statistics and Machine Learning Toolbox software uses that orientation.

### Background

The multivariate Student's  $t$  distribution is a generalization of the univariate Student's  $t$  to two or more variables. It is a distribution for random vectors of correlated variables, each element of which has a univariate Student's  $t$  distribution. In the same way as the univariate Student's  $t$  distribution can be constructed by dividing a standard univariate normal random variable by the square root of a univariate chi-square random variable, the multivariate Student's  $t$  distribution can be constructed by dividing a multivariate normal random vector having zero mean and unit variances by a univariate chi-square random variable.

The multivariate Student's  $t$  distribution is parameterized with a correlation matrix,  $\Sigma$ , and a positive scalar degrees of freedom parameter,  $\nu$ .  $\nu$  is analogous to the degrees of freedom parameter of a univariate Student's  $t$  distribution. The off-diagonal elements of

$\Sigma$  contain the correlations between variables. Note that when  $\Sigma$  is the identity matrix, variables are uncorrelated; however, they are not independent.

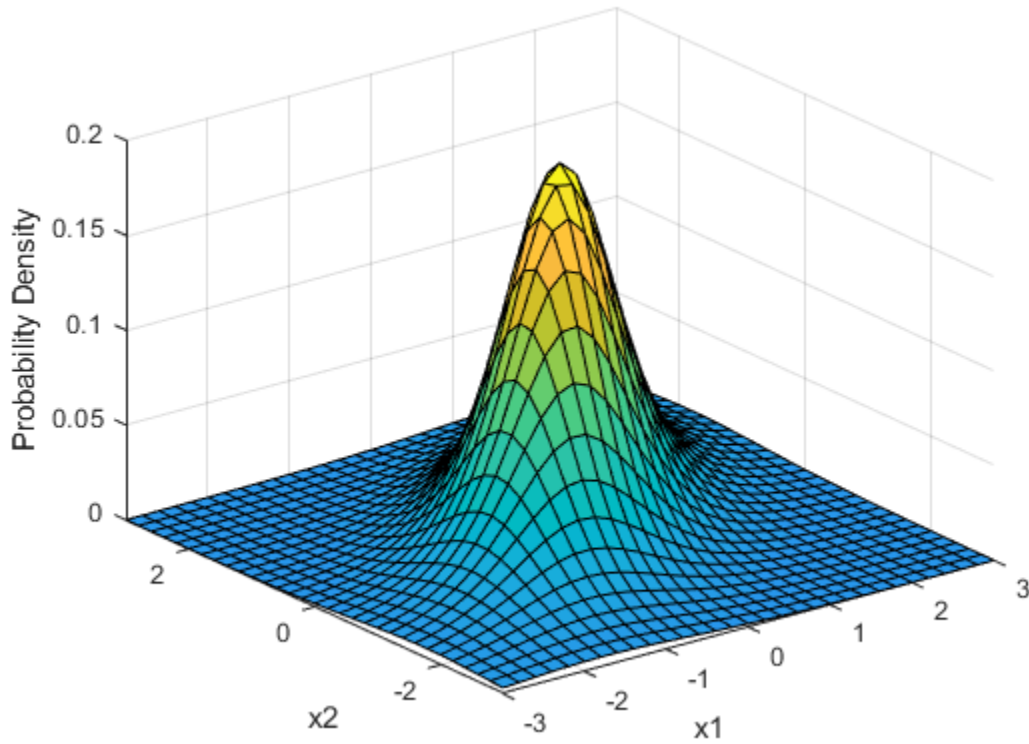
The multivariate Student's  $t$  distribution is often used as a substitute for the multivariate normal distribution in situations where it is known that the marginal distributions of the individual variables have fatter tails than the normal.

## Example

### Plot PDF and CDF of Multivariate $t$ -Distribution

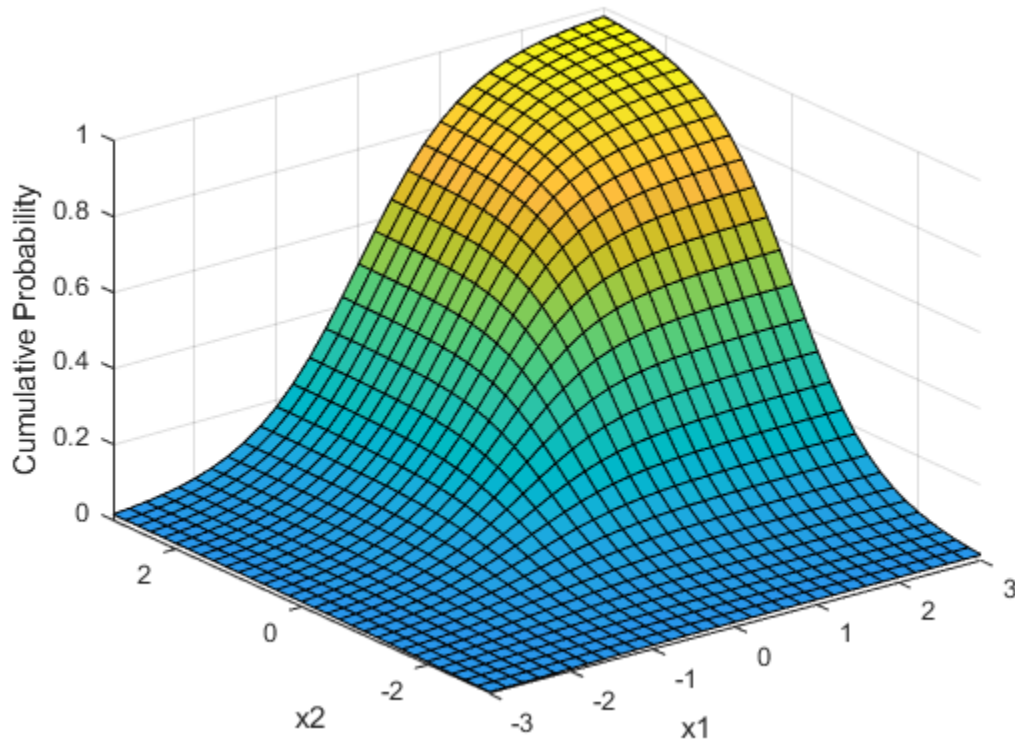
Plot the pdf of a bivariate Student's  $t$  distribution. You can use this distribution for a higher number of dimensions as well, although visualization is not easy.

```
Rho = [1 .6; .6 1];
nu = 5;
x1 = -3:.2:3; x2 = -3:.2:3;
[X1,X2] = meshgrid(x1,x2);
F = mvtpdf([X1(:) X2(:)],Rho,nu);
F = reshape(F,length(x2),length(x1));
surf(x1,x2,F);
caxis([min(F(:))-0.5*range(F(:)),max(F(:))]);
axis([-3 3 -3 3 0 .2])
xlabel('x1'); ylabel('x2'); zlabel('Probability Density');
```



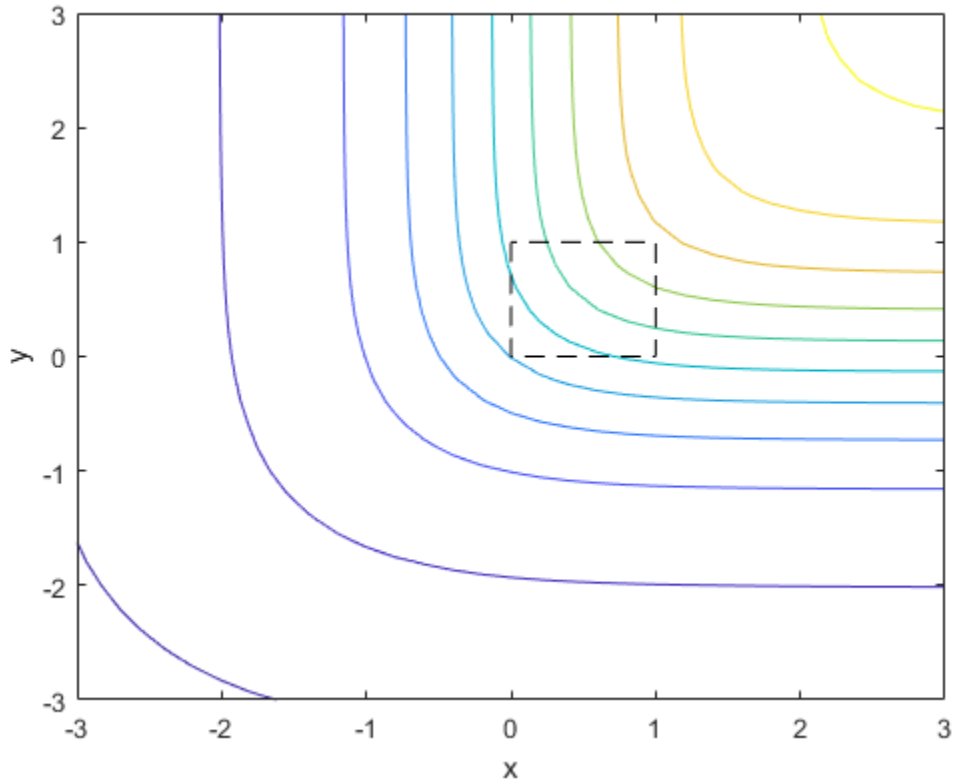
Plot the cdf of a bivariate Student's  $t$  distribution.

```
F = mvtcdf([X1(:) X2(:)],Rho,nu);  
F = reshape(F,length(x2),length(x1));  
surf(x1,x2,F);  
caxis([min(F(:))-0.5*range(F(:)),max(F(:))]);  
axis([-3 3 -3 3 0 1])  
xlabel('x1'); ylabel('x2'); zlabel('Cumulative Probability');
```



Since the bivariate Student's  $t$  distribution is defined on the plane, you can also compute cumulative probabilities over rectangular regions. For example, this contour plot illustrates the computation that follows, of the probability contained within the unit square shown in the figure.

```
contour(x1,x2,F,[.0001 .001 .01 .05:.1:.95 .99 .999 .9999]);
xlabel('x'); ylabel('y');
line([0 0 1 1 0],[1 0 0 1 1],'linestyle','--','color','k');
```



Compute the value of the probability contained within the unit square.

```
F = mvtcdf([0 0],[1 1],Rho,nu)
```

```
F =
```

```
0.1401
```

Computing a multivariate cumulative probability requires significantly more work than computing a univariate probability. By default, the `mvtcdf` function computes values to less than full machine precision and returns an estimate of the error, as an optional second output.



```
[F,err] = mvtcdf([0 0],[1 1],Rho,nu)
```

```
F =
```

```
    0.1401
```

```
err =
```

```
    1.0000e-08
```

## See Also

[mvtcdf](#) | [mvtpdf](#) | [mvtrnd](#)

## More About

- “Working with Probability Distributions” on page 5-3
- “Supported Distributions” on page 5-17

## Nakagami Distribution

In this section...
“Definition” on page B-118
“Background” on page B-118
“Parameters” on page B-118

### Definition

The Nakagami distribution has the density function

$$2\left(\frac{\mu}{\omega}\right)^{\mu} \frac{1}{\Gamma(\mu)} x^{(2\mu-1)} e^{-\frac{\mu}{\omega}x^2}$$

with shape parameter  $\mu$  and scale parameter  $\omega > 0$ , for  $x > 0$ . If  $x$  has a Nakagami distribution with parameters  $\mu$  and  $\omega$ , then  $x^2$  has a gamma distribution with shape parameter  $\mu$  and scale parameter  $\omega/\mu$ .

### Background

In communications theory, Nakagami distributions, Rician distributions on page B-149, and Rayleigh distributions on page B-146 are used to model scattered signals that reach a receiver by multiple paths. Depending on the density of the scatter, the signal will display different fading characteristics. Rayleigh and Nakagami distributions are used to model dense scatters, while Rician distributions model fading with a stronger line-of-sight. Nakagami distributions can be reduced to Rayleigh distributions, but give more control over the extent of the fading.

### Parameters

To estimate distribution parameters, use `mle` or the Distribution Fitter app.

### See Also

`NakagamiDistribution`

## More About

- “Working with Probability Distributions” on page 5-3
- “Supported Distributions” on page 5-17

## Negative Binomial Distribution

In this section...
“Definition” on page B-120
“Background” on page B-120
“Parameters” on page B-121
“Example” on page B-123

### Definition

When the  $r$  parameter is an integer, the negative binomial pdf is

$$y = f(x | r, p) = \binom{r+x-1}{x} p^r q^x I_{(0,1,\dots)}(x)$$

where  $q = 1 - p$ . When  $r$  is not an integer, the binomial coefficient in the definition of the pdf is replaced by the equivalent expression

$$\frac{\Gamma(r+x)}{\Gamma(r)\Gamma(x+1)}$$

### Background

In its simplest form (when  $r$  is an integer), the negative binomial distribution models the number of failures  $x$  before a specified number of successes is reached in a series of independent, identical trials. Its parameters are the probability of success in a single trial,  $p$ , and the number of successes,  $r$ . A special case of the negative binomial distribution, when  $r = 1$ , is the geometric distribution on page B-64, which models the number of failures before the first success.

More generally,  $r$  can take on non-integer values. This form of the negative binomial distribution has no interpretation in terms of repeated trials, but, like the Poisson distribution on page B-143, it is useful in modeling count data. The negative binomial distribution is more general than the Poisson distribution because it has a variance that is greater than its mean, making it suitable for count data that do not meet the assumptions of the Poisson distribution. In the limit, as  $r$  increases to infinity, the negative binomial distribution approaches the Poisson distribution.

## Parameters

Suppose you are collecting data on the number of auto accidents on a busy highway, and would like to be able to model the number of accidents per day. Because these are count data, and because there are a very large number of cars and a small probability of an accident for any specific car, you might think to use the Poisson distribution. However, the probability of having an accident is likely to vary from day to day as the weather and amount of traffic change, and so the assumptions needed for the Poisson distribution are not met. In particular, the variance of this type of count data sometimes exceeds the mean by a large amount. The data below exhibit this effect: most days have few or no accidents, and a few days have a large number.

```
accident = [2 3 4 2 3 1 12 8 14 31 23 1 10 7 0];  
m = mean(accident)  
v = var(accident)
```

```
m =  
  
8.0667
```

```
v =  
  
79.3524
```

The negative binomial distribution is more general than the Poisson, and is often suitable for count data when the Poisson is not. The function `nbinfit` returns the maximum likelihood estimates (MLEs) and confidence intervals for the parameters of the negative binomial distribution. Here are the results from fitting the `accident` data:

```
[phat,pci] = nbinfit(accident)
```

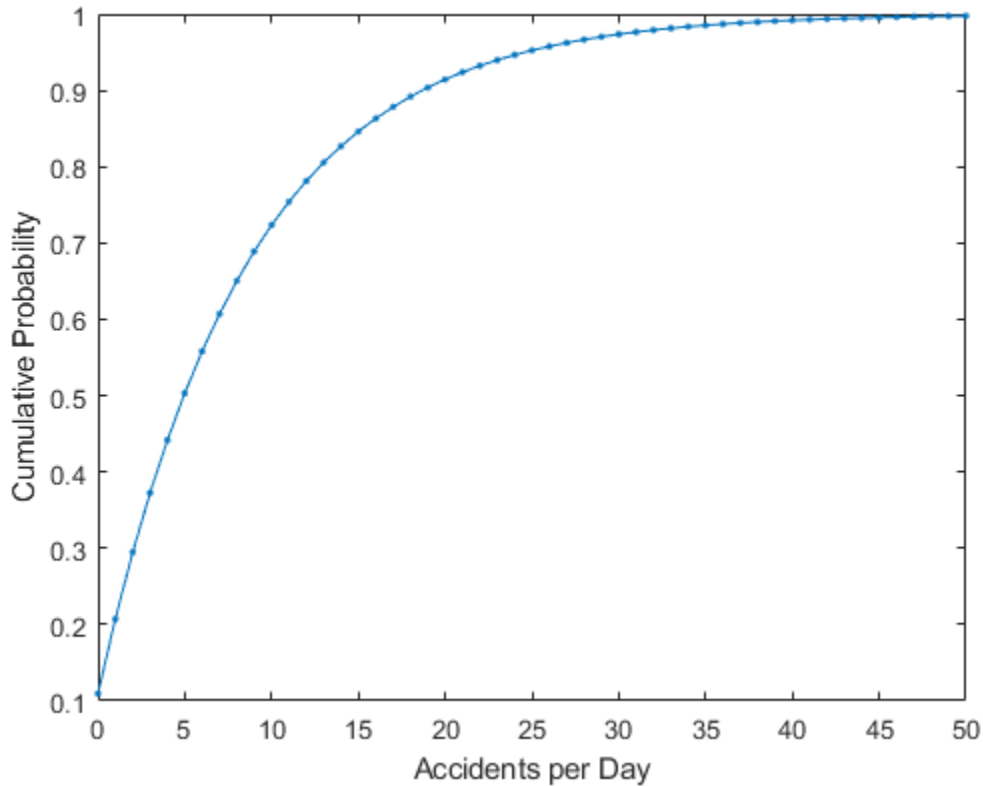
```
phat =  
  
1.0060    0.1109
```

```
pci =  
  
0.2152    0.0171
```

1.7968    0.2046

It is difficult to give a physical interpretation in this case to the individual parameters. However, the estimated parameters can be used in a model for the number of daily accidents. For example, a plot of the estimated cumulative probability function shows that while there is an estimated 10% chance of no accidents on a given day, there is also about a 10% chance that there will be 20 or more accidents.

```
plot(0:50,nbincdf(0:50,phat(1),phat(2)),'.-');  
xlabel('Accidents per Day')  
ylabel('Cumulative Probability')
```

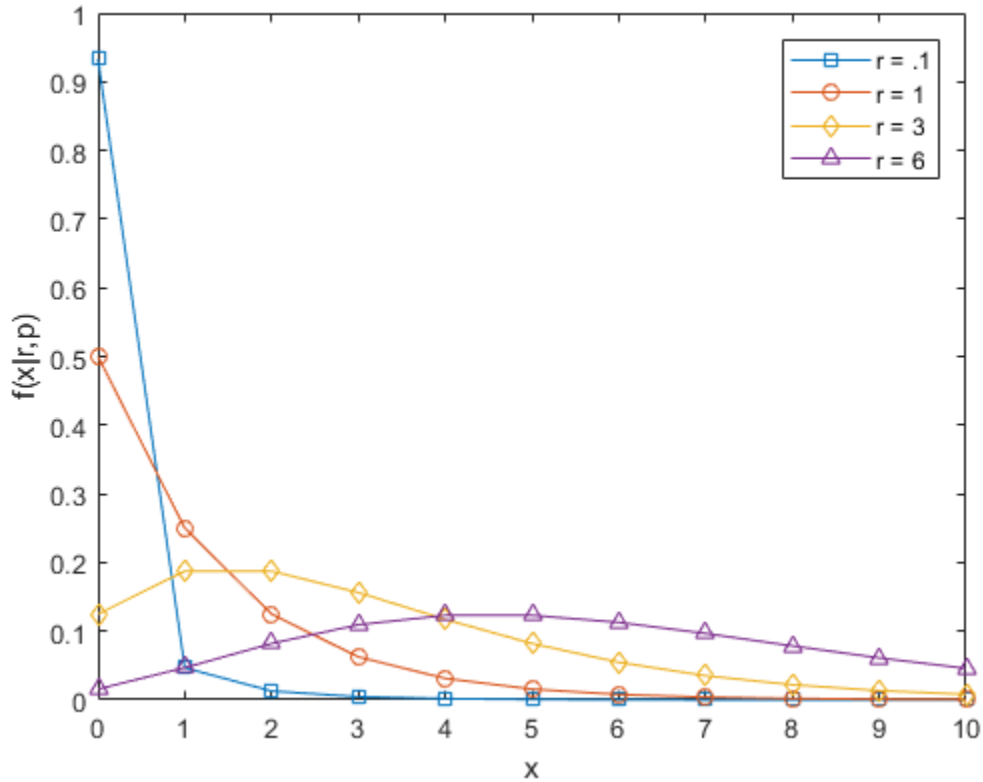


## Example

### Compute and Plot Negative Binomial Distribution PDF

Compute and plot the pdf using four different values for the parameter  $r$ , the desired number of successes: .1, 1, 3, and 6. In each case, the probability of success  $p$  is .5.

```
x = 0:10;
plot(x,nbinpdf(x,.1,.5),'s-', ...
      x,nbinpdf(x,1,.5),'o-', ...
      x,nbinpdf(x,3,.5),'d-', ...
      x,nbinpdf(x,6,.5),'^-',);
legend({'r = .1' 'r = 1' 'r = 3' 'r = 6'})
xlabel('x')
ylabel('f(x|r,p)')
```



The plot shows that the negative binomial distribution can take on a variety of shapes, ranging from very skewed to nearly symmetric, depending on the value of  $r$ .

## See Also

NegativeBinomialDistribution

## More About

- “Working with Probability Distributions” on page 5-3
- “Supported Distributions” on page 5-17



## Noncentral Chi-Square Distribution

### In this section...

“Definition” on page B-125

“Background” on page B-125

“Examples” on page B-126

### Definition

There are many equivalent formulas for the noncentral chi-square distribution function. One formulation uses a modified Bessel function of the first kind. Another uses the generalized Laguerre polynomials. The cumulative distribution function is computed using a weighted sum of  $\chi^2$  probabilities with the weights equal to the probabilities of a Poisson distribution. The Poisson parameter is one-half of the noncentrality parameter of the noncentral chi-square

$$F(x | \nu, \delta) = \sum_{j=0}^{\infty} \left( \frac{\left(\frac{1}{2}\delta\right)^j}{j!} e^{-\frac{\delta}{2}} \right) \Pr\left[\chi_{\nu+2j}^2 \leq x\right]$$

where  $\delta$  is the noncentrality parameter.

### Background

The  $\chi^2$  distribution is actually a simple special case of the noncentral chi-square distribution. One way to generate random numbers with a  $\chi^2$  distribution (with  $\nu$  degrees of freedom) is to sum the squares of  $\nu$  standard normal random numbers (mean equal to zero.)

What if the normally distributed quantities have a mean other than zero? The sum of squares of these numbers yields the noncentral chi-square distribution. The noncentral chi-square distribution requires two parameters: the degrees of freedom and the noncentrality parameter. The noncentrality parameter is the sum of the squared means of the normally distributed quantities.

The noncentral chi-square has scientific application in thermodynamics and signal processing. The literature in these areas may refer to it as the “Rician Distribution” on page B-149 or generalized “Rayleigh Distribution” on page B-146.

## Examples

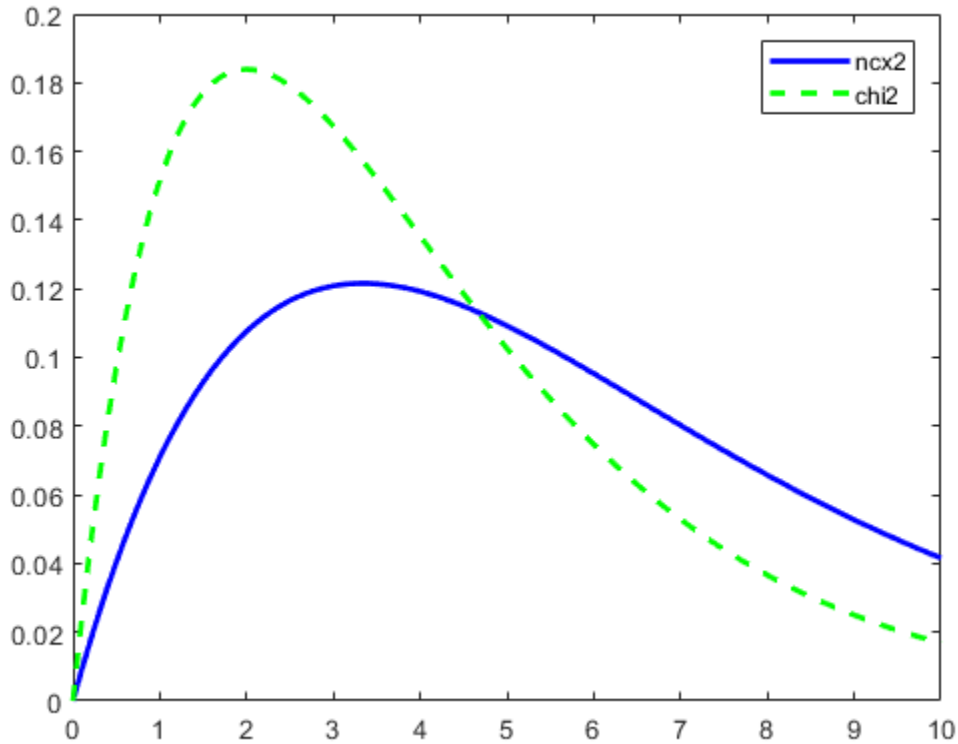
### Compute Noncentral Chi-Square Distribution pdf

Compute the pdf of a noncentral chi-square distribution with degrees of freedom  $\nu = 4$  and noncentrality parameter  $\text{DELTA} = 2$ . For comparison, also compute the pdf of a chi-square distribution with the same degrees of freedom.

```
x = (0:0.1:10)';  
ncx2 = ncx2pdf(x,4,2);  
chi2 = chi2pdf(x,4);
```

Plot the pdf of the noncentral chi-square distribution on the same figure as the pdf of the chi-square distribution.

```
figure;  
plot(x,ncx2,'b-','LineWidth',2)  
hold on  
plot(x,chi2,'g--','LineWidth',2)  
legend('ncx2','chi2')
```



## See Also

[ncx2cdf](#) | [ncx2inv](#) | [ncx2pdf](#) | [ncx2rnd](#) | [ncx2stat](#) | [random](#)

## More About

- “Working with Probability Distributions” on page 5-3
- “Supported Distributions” on page 5-17

## Noncentral F Distribution

In this section...
“Definition” on page B-128
“Background” on page B-128
“Examples” on page B-129

### Definition

Similar to the noncentral  $\chi^2$  distribution, the toolbox calculates noncentral  $F$  distribution probabilities as a weighted sum of incomplete beta functions using Poisson probabilities as the weights.

$$F(x | v_1, v_2, \delta) = \sum_{j=0}^{\infty} \left( \frac{\left(\frac{1}{2}\delta\right)^j}{j!} e^{-\frac{\delta}{2}} \right) I\left(\frac{v_1 \cdot x}{v_2 + v_1 \cdot x} \mid \frac{v_1}{2} + j, \frac{v_2}{2}\right)$$

$I(x | a, b)$  is the incomplete beta function with parameters  $a$  and  $b$ , and  $\delta$  is the noncentrality parameter.

### Background

As with the  $\chi^2$  distribution, the  $F$  distribution is a special case of the noncentral  $F$  distribution. The  $F$  distribution is the result of taking the ratio of  $\chi^2$  random variables each divided by its degrees of freedom.

If the numerator of the ratio is a noncentral chi-square random variable divided by its degrees of freedom, the resulting distribution is the noncentral  $F$  distribution.

The main application of the noncentral  $F$  distribution is to calculate the power of a hypothesis test relative to a particular alternative.

## Examples

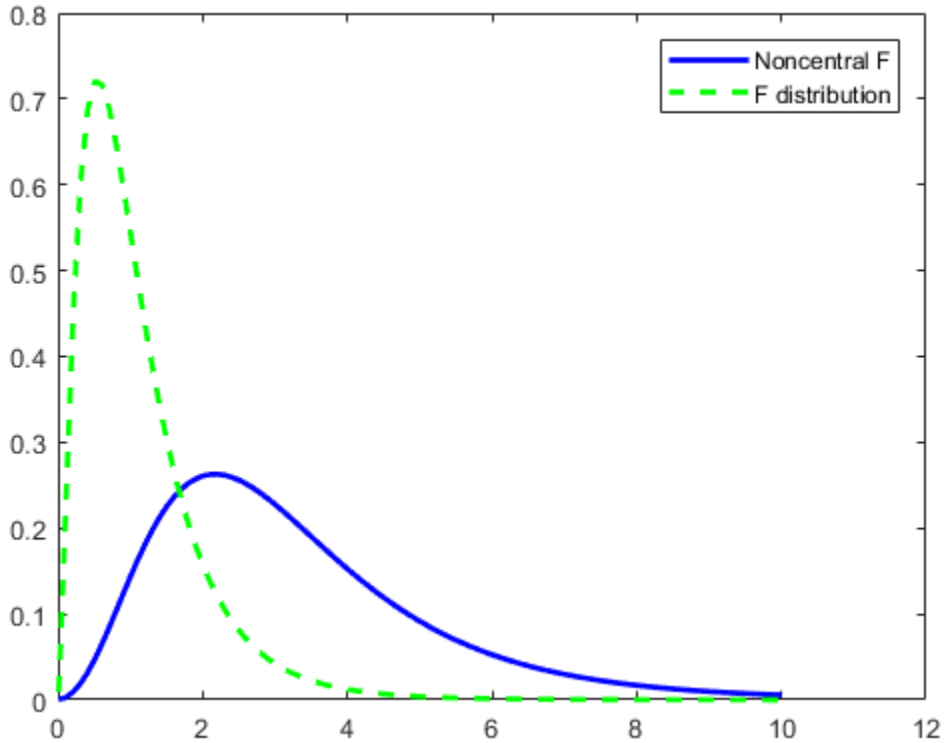
### Compute Noncentral F Distribution pdf

Compute the pdf of a noncentral  $F$  distribution with degrees of freedom  $NU1 = 5$  and  $NU2 = 20$ , and noncentrality parameter  $DELTA = 10$ . For comparison, also compute the pdf of an  $F$  distribution with the same degrees of freedom.

```
x = (0.01:0.1:10.01)';  
p1 = ncfpdf(x,5,20,10);  
p = fpdf(x,5,20);
```

Plot the pdf of the noncentral  $F$  distribution and the pdf of the  $F$  distribution on the same figure.

```
figure;  
plot(x,p1,'b-','LineWidth',2)  
hold on  
plot(x,p,'g--','LineWidth',2)  
legend('Noncentral F','F distribution')
```



## See Also

`ncfcdf` | `ncfinv` | `ncfpdf` | `ncfrnd` | `ncfstat` | `random`

## More About

- “Working with Probability Distributions” on page 5-3
- “Supported Distributions” on page 5-17

## Noncentral $t$ Distribution

### In this section...

“Definition” on page B-131

“Background” on page B-131

“Examples” on page B-132

### Definition

The most general representation of the noncentral  $t$  distribution is quite complicated. Johnson and Kotz [66] give a formula for the probability that a noncentral  $t$  variate falls in the range  $[-u, u]$ .

$$P(-u < x < u | v, \delta) = \sum_{j=0}^{\infty} \left( \frac{\left(\frac{1}{2}\delta^2\right)^j}{j!} e^{-\frac{\delta^2}{2}} \right) I\left(\frac{u^2}{v+u^2} \middle| \frac{1}{2} + j, \frac{v}{2}\right)$$

$I(x | v, \delta)$  is the incomplete beta function with parameters  $v$  and  $\delta$ .  $\delta$  is the noncentrality parameter, and  $v$  is the number of degrees of freedom.

### Background

The noncentral  $t$  distribution is a generalization of Student's  $t$  distribution.

Student's  $t$  distribution with  $n - 1$  degrees of freedom models the  $t$ -statistic

$$t = \frac{\bar{x} - \mu}{s / \sqrt{n}}$$

where  $\bar{x}$  is the sample mean and  $s$  is the sample standard deviation of a random sample of size  $n$  from a normal population with mean  $\mu$ . If the population mean is actually  $\mu_0$ , then the  $t$ -statistic has a noncentral  $t$  distribution with noncentrality parameter

$$\delta = \frac{\mu_0 - \mu}{\sigma / \sqrt{n}}$$

The noncentrality parameter is the normalized difference between  $\mu_0$  and  $\mu$ .

The noncentral  $t$  distribution gives the probability that a  $t$  test will correctly reject a false null hypothesis of mean  $\mu$  when the population mean is actually  $\mu_0$ ; that is, it gives the power of the  $t$  test. The power increases as the difference  $\mu_0 - \mu$  increases, and also as the sample size  $n$  increases.

## Examples

### Compute Noncentral $t$ Distribution pdf

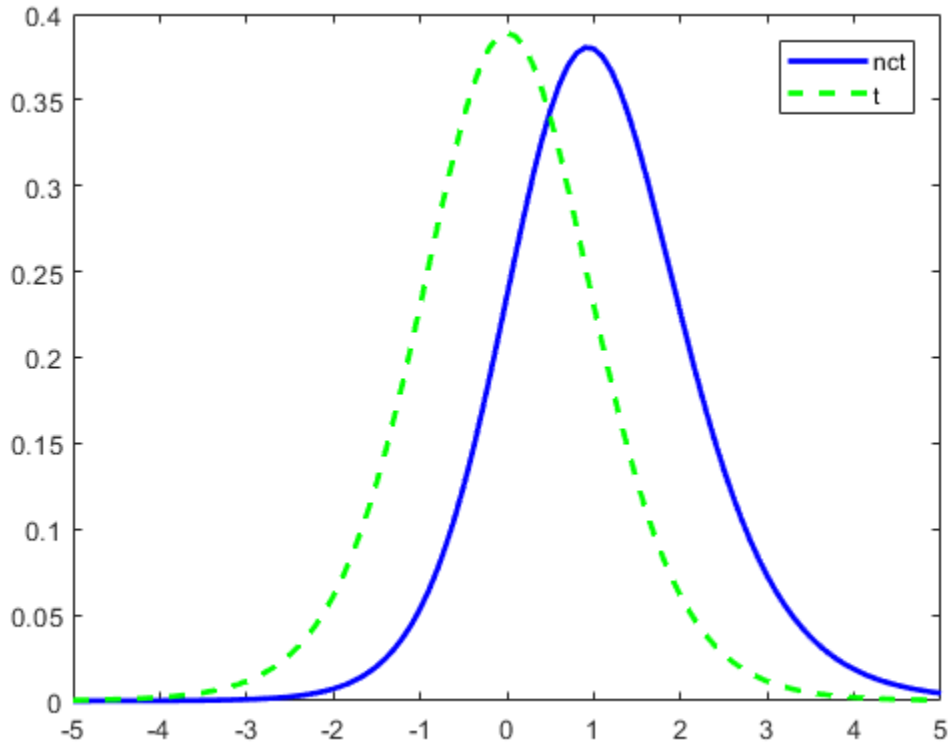
Compute the pdf of a noncentral  $t$  distribution with degrees of freedom  $\nu = 10$  and noncentrality parameter  $\text{DELTA} = 1$ . For comparison, also compute the pdf of a  $t$  distribution with the same degrees of freedom.

```
x = (-5:0.1:5)';  
nct = nctpdf(x,10,1);  
t = tpdf(x,10);
```

Plot the pdf of the noncentral  $t$  distribution and the pdf of the  $t$  distribution on the same figure.

```
plot(x,nct,'b-', 'LineWidth',2)  
hold on  
plot(x,t,'g--', 'LineWidth',2)  
legend('nct','t')
```





## See Also

`nctcdf` | `nctinv` | `nctpdf` | `nctrnd` | `nctstat` | `random`

## More About

- “Working with Probability Distributions” on page 5-3
- “Supported Distributions” on page 5-17

## Normal Distribution

In this section...
“Definition” on page B-134
“Background” on page B-134
“Parameters” on page B-134
“Examples” on page B-136

### Definition

The normal pdf is

$$y = f(x | \mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

### Background

The normal distribution is a two-parameter family of curves. The first parameter,  $\mu$ , is the mean. The second,  $\sigma$ , is the standard deviation. The standard normal distribution (written  $\Phi(x)$ ) sets  $\mu$  to 0 and  $\sigma$  to 1.

$\Phi(x)$  is functionally related to the error function, *erf*.

$$\text{erf}(x) = 2\Phi(x\sqrt{2}) - 1$$

The first use of the normal distribution was as a continuous approximation to the binomial.

The usual justification for using the normal distribution for modeling is the Central Limit Theorem, which states (roughly) that the sum of independent samples from any distribution with finite mean and variance converges to the normal distribution as the sample size goes to infinity.

### Parameters

To use statistical parameters such as mean and standard deviation reliably, you need to have a good estimator for them. The maximum likelihood estimates (MLEs) provide one

such estimator. However, an MLE might be biased, which means that its expected value of the parameter might not equal the parameter being estimated. For example, an MLE is biased for estimating the variance of a normal distribution. An unbiased estimator that is commonly used to estimate the parameters of the normal distribution is the *minimum variance unbiased estimator (MVUE)*. The MVUE has the minimum variance of all unbiased estimators of a parameter.

The MVUEs of parameters  $\mu$  and  $\sigma^2$  for the normal distribution are the sample mean and variance. The sample mean is also the MLE for  $\mu$ . The following are two common formulas for the variance.

$$s^2 = \frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})^2$$

$$s^2 = \frac{1}{n-1} \sum_{i=1}^n (x_i - \bar{x})^2$$

where

$$\bar{x} = \sum_{i=1}^n \frac{x_i}{n}$$

Equation 1 is the maximum likelihood estimator for  $\sigma^2$ , and equation 2 is the MVUE.

As an example, suppose you want to estimate the mean,  $\mu$ , and the variance,  $\sigma^2$ , of the heights of all fourth grade children in the United States. The function `normfit` returns the MVUE for  $\mu$ , the square root of the MVUE for  $\sigma^2$ , and confidence intervals for  $\mu$  and  $\sigma^2$ . Here is a playful example modeling the heights in inches of a randomly chosen fourth grade class.

```
rng default; % For reproducibility
height = normrnd(50,2,30,1); % Simulate heights
[mu,s,muci,sci] = normfit(height)
```

```
mu =
```

```
51.1038
```

```
s =  
    2.6001
```

```
muci =  
    50.1329  
    52.0747
```

```
sci =  
    2.0707  
    3.4954
```

Note that  $s^2$  is the MVUE of the variance.

```
s^2
```

```
ans =  
    6.7605
```

## Examples

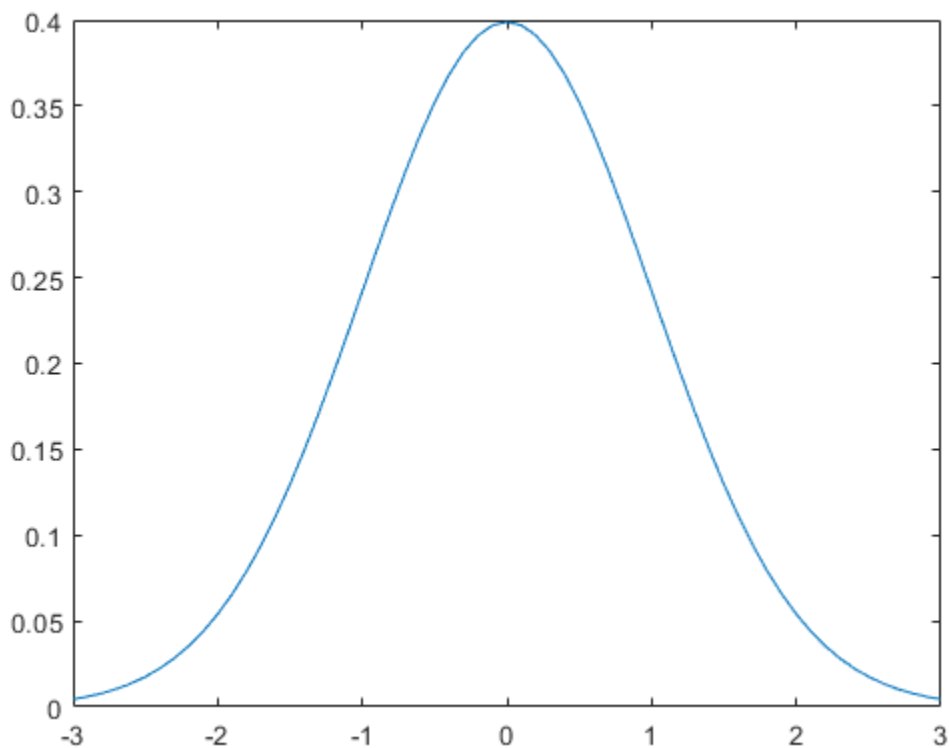
### Compute and Plot the Normal Distribution pdf

Compute the pdf of a standard normal distribution, with parameters  $\mu$  equal to 0 and  $\sigma$  equal to 1.

```
x = [-3:.1:3];  
norm = normpdf(x,0,1);
```

Plot the pdf.

```
figure;  
plot(x,norm)
```



## See Also

`NormalDistribution`

## Related Examples

- “Compare Multiple Distribution Fits” on page 5-114

## More About

- “Working with Probability Distributions” on page 5-3

- “Supported Distributions” on page 5-17

## Pareto Distribution

See “Generalized Pareto Distribution” on page B-58.

## Pearson System

See “Pearson and Johnson Systems” on page 7-28.



# Piecewise Linear Distribution

In this section...
“Overview” on page B-141
“Parameters” on page B-141
“Cumulative Distribution Function” on page B-141
“Relationship to Other Distributions” on page B-141

## Overview

The piecewise linear distribution creates a nonparametric representation of the cumulative distribution function (cdf) by linearly connecting the known cdf values from the sample data.

## Parameters

The piecewise linear distribution uses the following parameters.

Parameter	Description
$x$	Vector of $x$ values at which the cdf changes slope
$F_x$	Vector of cdf values that correspond to each value in $x$

## Cumulative Distribution Function

The piecewise linear distribution constructs a continuous cumulative distribution function (cdf) by connecting the empirical cdf values such that the cdf increases linearly between  $x(j)$  and  $x(j + 1)$ .

## Relationship to Other Distributions

The piecewise linear distribution is a continuous version of the discrete empirical cumulative distribution function (ecdf).

## See Also

PiecewiseLinearDistribution

## More About

- “Nonparametric and Empirical Probability Distributions” on page 5-37
- “Working with Probability Distributions” on page 5-3
- “Supported Distributions” on page 5-17

# Poisson Distribution

**In this section...**

“Definition” on page B-143

“Background” on page B-143

“Parameters” on page B-143

“Examples” on page B-144

## Definition

The Poisson pdf is

$$f(x | \lambda) = \frac{\lambda^x}{x!} e^{-\lambda}; x = 0, 1, 2, \dots, \infty.$$

## Background

The Poisson distribution is appropriate for applications that involve counting the number of times a random event occurs in a given amount of time, distance, area, etc. Sample applications that involve Poisson distributions include the number of Geiger counter clicks per second, the number of people walking into a store in an hour, and the number of flaws per 1000 feet of video tape.

The Poisson distribution is a one-parameter discrete distribution that takes nonnegative integer values. The parameter,  $\lambda$ , is both the mean and the variance of the distribution. Thus, as the size of the numbers in a particular sample of Poisson random numbers gets larger, so does the variability of the numbers.

The Poisson distribution is the limiting case of a binomial distribution where  $N$  approaches infinity and  $p$  goes to zero while  $Np = \lambda$ .

The Poisson and exponential distributions are related. If the number of counts follows the Poisson distribution, then the interval between individual counts follows the exponential distribution.

## Parameters

The MLE and the MVUE of the Poisson parameter,  $\lambda$ , is the sample mean. The sum of independent Poisson random variables is also Poisson distributed with the parameter

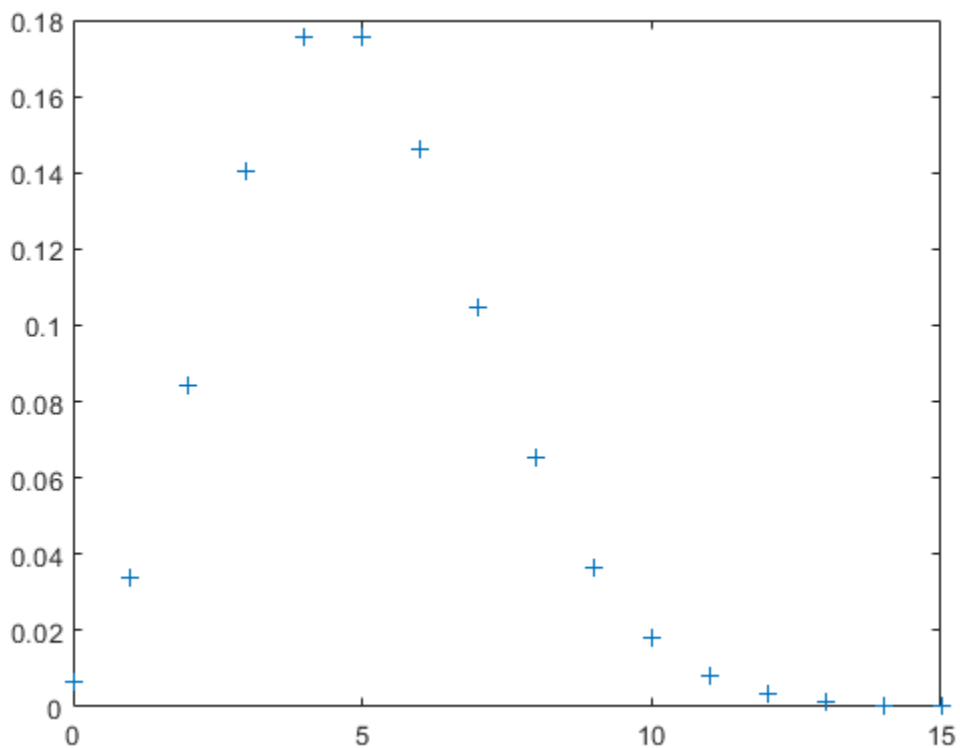
equal to the sum of the individual parameters. This is used to calculate confidence intervals  $\lambda$ . As  $\lambda$  gets large the Poisson distribution can be approximated by a normal distribution with  $\mu = \lambda$  and  $\sigma^2 = \lambda$ . This approximation is used to calculate confidence intervals for values of  $\lambda$  greater than 100.

## Examples

### Compute and Plot Poisson Distribution PDF

Compute and plot the pdf of a Poisson distribution with parameter `lambda = 5`.

```
x = 0:15;  
y = poisspdf(x, 5);  
plot(x, y, '+')
```



## See Also

`PoissonDistribution`

## More About

- “Working with Probability Distributions” on page 5-3
- “Supported Distributions” on page 5-17

## Rayleigh Distribution

In this section...
“Definition” on page B-146
“Background” on page B-146
“Parameters” on page B-147
“Examples” on page B-147

### Definition

The Rayleigh pdf is

$$y = f(x | b) = \frac{x}{b^2} e^{\left(\frac{-x^2}{2b^2}\right)}$$

### Background

The Rayleigh distribution is a special case of the Weibull distribution on page B-193. If  $A$  and  $B$  are the parameters of the Weibull distribution, then the Rayleigh distribution with parameter  $b$  is equivalent to the Weibull distribution with parameters  $A = \sqrt{2}b$  and  $B = 2$ .

If the component velocities of a particle in the  $x$  and  $y$  directions are two independent normal random variables with zero means and equal variances, then the distance the particle travels per unit time is distributed Rayleigh.

In communications theory, Nakagami distributions on page B-118, Rician distributions on page B-149, and Rayleigh distributions are used to model scattered signals that reach a receiver by multiple paths. Depending on the density of the scatter, the signal will display different fading characteristics. Rayleigh and Nakagami distributions are used to model dense scatters, while Rician distributions model fading with a stronger line-of-sight. Nakagami distributions can be reduced to Rayleigh distributions, but give more control over the extent of the fading.

## Parameters

The `raylfit` function returns the MLE of the Rayleigh parameter. This estimate is

$$b = \sqrt{\frac{1}{2n} \sum_{i=1}^n x_i^2}$$

## Examples

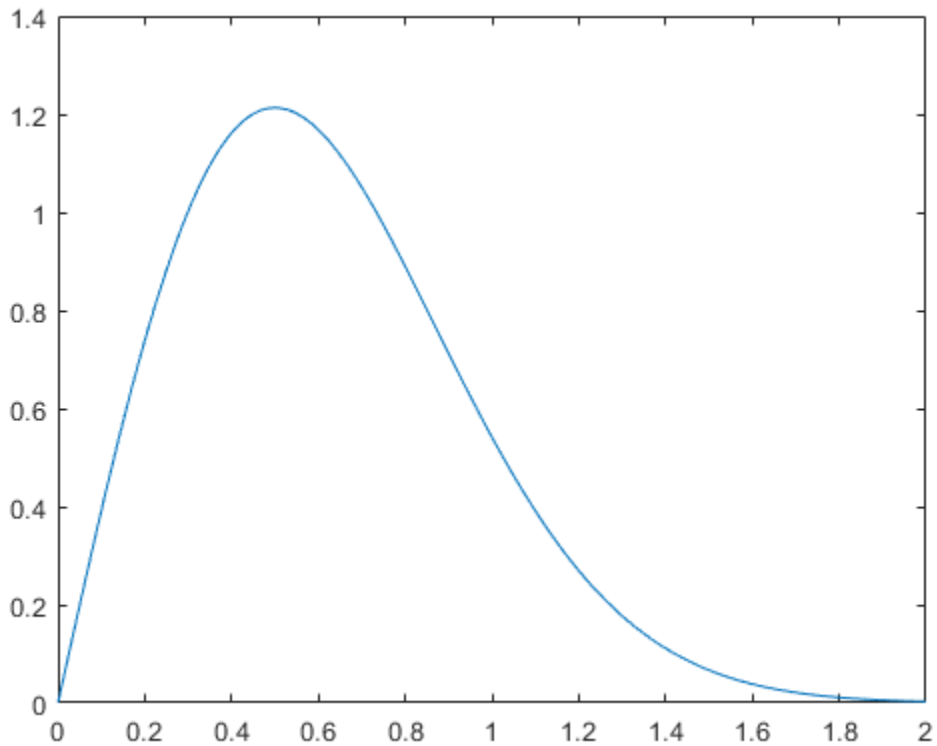
### Compute and Plot Rayleigh Distribution pdf

Compute the pdf of a Rayleigh distribution with parameter  $B = 0.5$ .

```
x = [0:0.01:2];  
p = raylpdf(x,0.5);
```

Plot the pdf.

```
figure;  
plot(x,p)
```



## See Also

`RayleighDistribution`

## More About

- “Working with Probability Distributions” on page 5-3
- “Supported Distributions” on page 5-17



## Rician Distribution

### In this section...

“Definition” on page B-149

“Background” on page B-149

“Parameters” on page B-149

### Definition

The Rician distribution has the density function

$$I_0\left(\frac{xs}{\sigma^2}\right) \frac{x}{\sigma^2} e^{-\left(\frac{x^2+s^2}{2\sigma^2}\right)}$$

with noncentrality parameter  $s \geq 0$  and scale parameter  $\sigma > 0$ , for  $x > 0$ .  $I_0$  is the zero-order modified Bessel function of the first kind. If  $x$  has a Rician distribution with parameters  $s$  and  $\sigma$ , then  $(x/\sigma)^2$  has a noncentral chi-square distribution with two degrees of freedom and noncentrality parameter  $(s/\sigma)^2$ .

### Background

In communications theory, Nakagami distributions on page B-118, Rician distributions, and Rayleigh distributions on page B-146 are used to model scattered signals that reach a receiver by multiple paths. Depending on the density of the scatter, the signal will display different fading characteristics. Rayleigh and Nakagami distributions are used to model dense scatters, while Rician distributions model fading with a stronger line-of-sight. Nakagami distributions can be reduced to Rayleigh distributions, but give more control over the extent of the fading.

### Parameters

To estimate distribution parameters, use `mle` or the Distribution Fitter app.

### See Also

RicianDistribution

## **More About**

- “Working with Probability Distributions” on page 5-3
- “Supported Distributions” on page 5-17

## Stable Distribution

### In this section...

“Overview” on page B-151

“Parameters” on page B-152

“Probability Density Function” on page B-152

“Cumulative Distribution Function” on page B-156

“Descriptive Statistics” on page B-160

“Relationship to Other Distributions” on page B-161

### Overview

Stable distributions are a class of probability distributions suitable for modeling heavy tails and skewness. A linear combination of two independent, identically-distributed stable-distributed random variables has the same distribution as the individual variables. In other words, if  $X_1, X_2, \dots, X_n$  are independent and identically distributed stable random variables, then for every  $n$

$$X_1 + X_2 + \dots + X_n \stackrel{d}{=} c_n X + d_n$$

where the constant  $c_n > 0$  and  $d_n \in \mathbb{R}$ .

The stable distribution is an application of the Generalized Central Limit Theorem, which states that the limit of normalized sums of independent identically distributed variables is stable.

Several different parameterizations exist for the stable distribution. The implementation in Statistics and Machine Learning Toolbox uses the parameterization described in [2].

In this case, a random variable  $X$  has the stable distribution  $S(\alpha, \beta, \gamma, \delta_0; 0)$  if its characteristic function is given by:

$$E\left(e^{itX}\right) = \begin{cases} \exp\left(-\gamma^\alpha |t|^\alpha \left[1 + i\beta \operatorname{sign}(t) \tan \frac{\pi\alpha}{2} \left((\gamma|t|)^{1-\alpha} - 1\right)\right] + i\delta_0 t\right) & \text{for } \alpha \neq 1, \\ \exp\left(-\gamma |t| \left[1 + i\beta \operatorname{sign}(t) \frac{2}{\pi} \ln(\gamma|t|)\right] + i\delta_0 t\right) & \text{for } \alpha = 1 \end{cases}$$

## Parameters

The stable distribution uses the following parameters.

Parameter	Description	Support
alpha	First shape parameter	$0 < \alpha \leq 2$
beta	Second shape parameter	$-1 \leq \beta \leq 1$
gam	Scale parameter	$0 < \gamma < \infty$
delta	Location parameter	$-\infty < \delta < \infty$

The first shape parameter,  $\alpha$ , describes the tails of the distribution. The software computes the densities of the stable distribution using the direct integration method. As explained in [1], numerical difficulties exist with accurately computing the pdf and cdf

when the  $\alpha$  parameter is close to 1 or 0. If  $\alpha$  is close to 1 (specifically,  $0 < |\alpha - 1| < 0.02$ ), then the software rounds  $\alpha$  to 1. If  $\alpha$  is close to 0, then the densities may not be accurate.

The second shape parameter,  $\beta$ , describes the skewness of the distribution. If  $\beta = 0$ , then the distribution is symmetric. If  $\beta > 0$ , then the distribution is right-skewed. If  $\beta < 0$ , then the distribution is left-skewed. When  $\alpha$  is small, the skewness of  $\beta$  is significant. As  $\alpha$  increases, the effect of  $\beta$  decreases.

## Probability Density Function

### Definition

Most members of the stable distribution family do not have an explicit probability density function (pdf). Instead, the pdf is described in terms of the characteristic function [2].

Some special cases of the stable distribution, such as the normal, Cauchy, and Lévy distributions, have closed-form density functions. See Relationship to Other Distributions on page B-161 for more information.

Use `pdf` to calculate the probability density function for the stable distribution. The software computes the pdf using the direct integration method. As explained in [1], numerical difficulties exist with accurately computing the pdf when the  $\alpha$  parameter is

close to 1 or 0. If  $\alpha$  is close to 1 (specifically,  $0 < |\alpha - 1| < 0.02$ ), then the software rounds  $\alpha$  to 1. If  $\alpha$  is close to 0, then the densities may not be accurate.

## Compare PDFs of Stable Distributions

The following plot compares the probability density functions for stable distributions with different alpha values. In each case, beta = 0, gam = 1, and delta = 0.

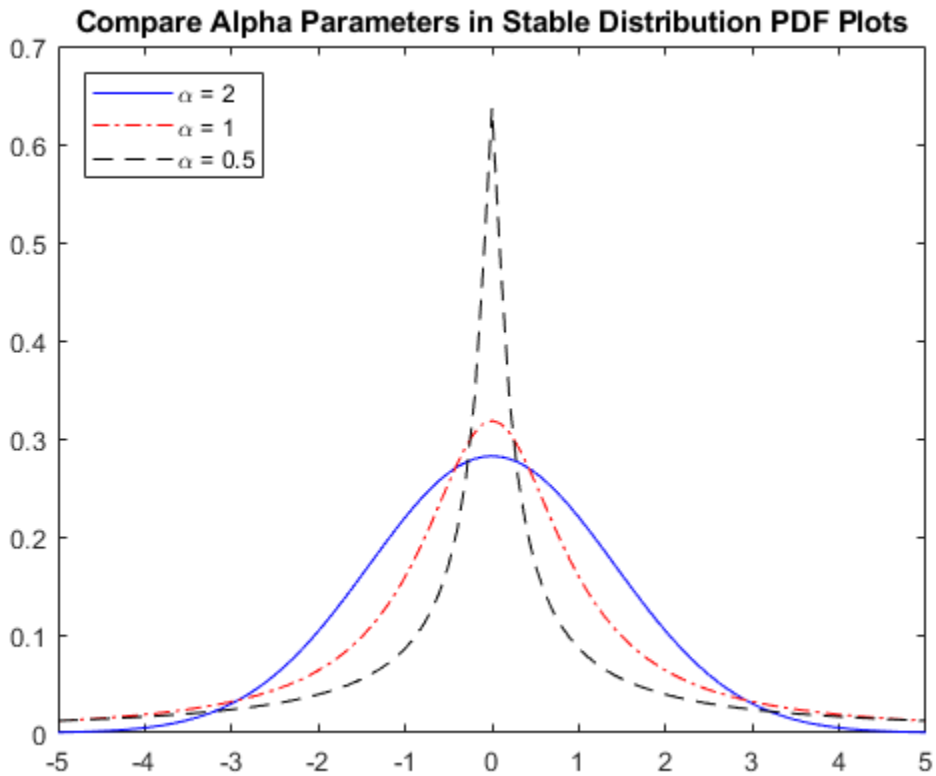
```
pd1 = makedist('Stable', 'alpha', 2, 'beta', 0, 'gam', 1, 'delta', 0);
pd2 = makedist('Stable', 'alpha', 1, 'beta', 0, 'gam', 1, 'delta', 0);
pd3 = makedist('Stable', 'alpha', 0.5, 'beta', 0, 'gam', 1, 'delta', 0);
```

Calculate the pdf for each distribution.

```
x = -5:.1:5;
pdf1 = pdf(pd1, x);
pdf2 = pdf(pd2, x);
pdf3 = pdf(pd3, x);
```

Plot all three pdf functions on the same figure for visual comparison.

```
figure
plot(x, pdf1, 'b-');
hold on
plot(x, pdf2, 'r-.');
plot(x, pdf3, 'k--');
title('Compare Alpha Parameters in Stable Distribution PDF Plots')
legend('\alpha = 2', '\alpha = 1', '\alpha = 0.5', 'Location', 'northwest')
hold off
```



The plot illustrates the effect of the alpha parameter on the tails of the distribution.

The next plot compares the probability density functions for stable distributions with different beta values. In each case, alpha = 0.5, gam = 1, and delta = 0.

```
pd1 = makedist('Stable','alpha',0.5,'beta',0,'gam',1,'delta',0);
pd2 = makedist('Stable','alpha',0.5,'beta',0.5,'gam',1,'delta',0);
pd3 = makedist('Stable','alpha',0.5,'beta',1,'gam',1,'delta',0);
```

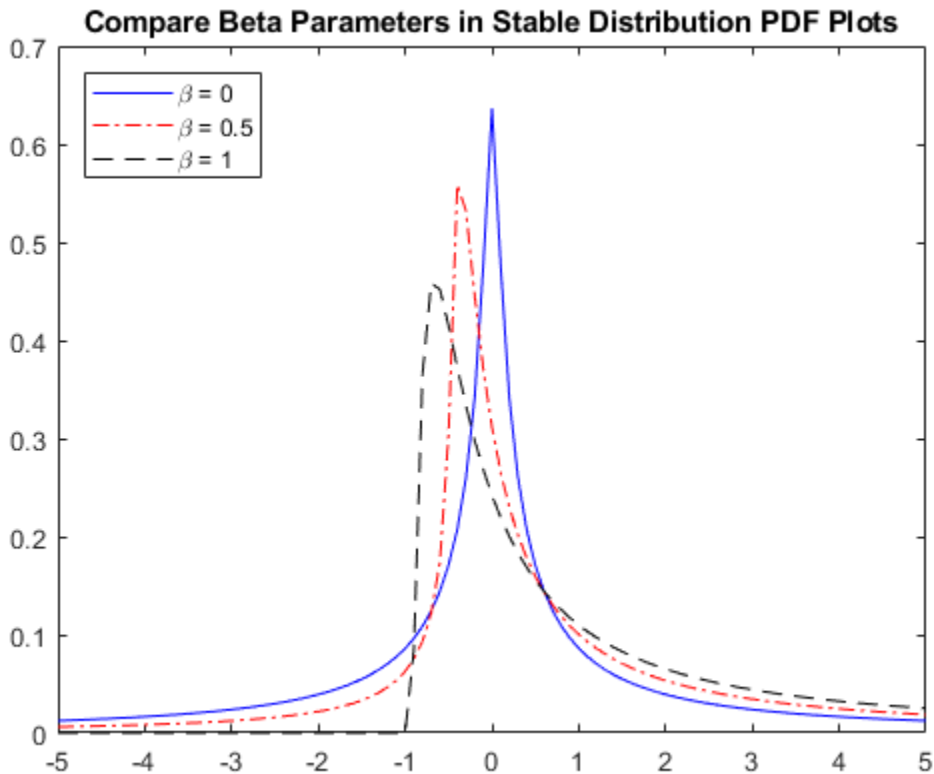
Calculate the pdf for each distribution.

```
x = -5:.1:5;
pdf1 = pdf(pd1,x);
```

```
pdf2 = pdf(pd2,x);  
pdf3 = pdf(pd3,x);
```

Plot all three pdf functions on the same figure for visual comparison.

```
figure  
plot(x,pdf1,'b-');  
hold on  
plot(x,pdf2,'r-.');  
plot(x,pdf3,'k--');  
title('Compare Beta Parameters in Stable Distribution PDF Plots')  
legend('\beta = 0', '\beta = 0.5', '\beta = 1', 'Location', 'northwest')  
hold off
```



### Random Number Generation

Use `random` to generate random numbers from the stable distribution. The software generates random numbers for the stable distribution using the method proposed in [3]

### Cumulative Distribution Function

#### Definition

Most members of the stable distribution family do not have an explicit cumulative distribution function (cdf). Instead, the cdf is described in terms of the characteristic function [2].



Use `cdf` to calculate the cumulative distribution function for the stable distribution. The software computes the cdf using the direct integration method. As explained in [1], numerical difficulties exist with accurately computing the cdf when the  $\alpha$  parameter is

close to 1 or 0. If  $\alpha$  is close to 1 (specifically,  $0 < |\alpha - 1| < 0.02$ ), then the software rounds  $\alpha$  to 1. If  $\alpha$  is close to 0, then the densities may not be accurate.

### Compare CDFs of Stable Distributions

The following plot compares the cumulative distribution functions for stable distributions with different alpha values. In each case,  $\beta = 0$ ,  $\gamma = 1$ , and  $\delta = 0$ .

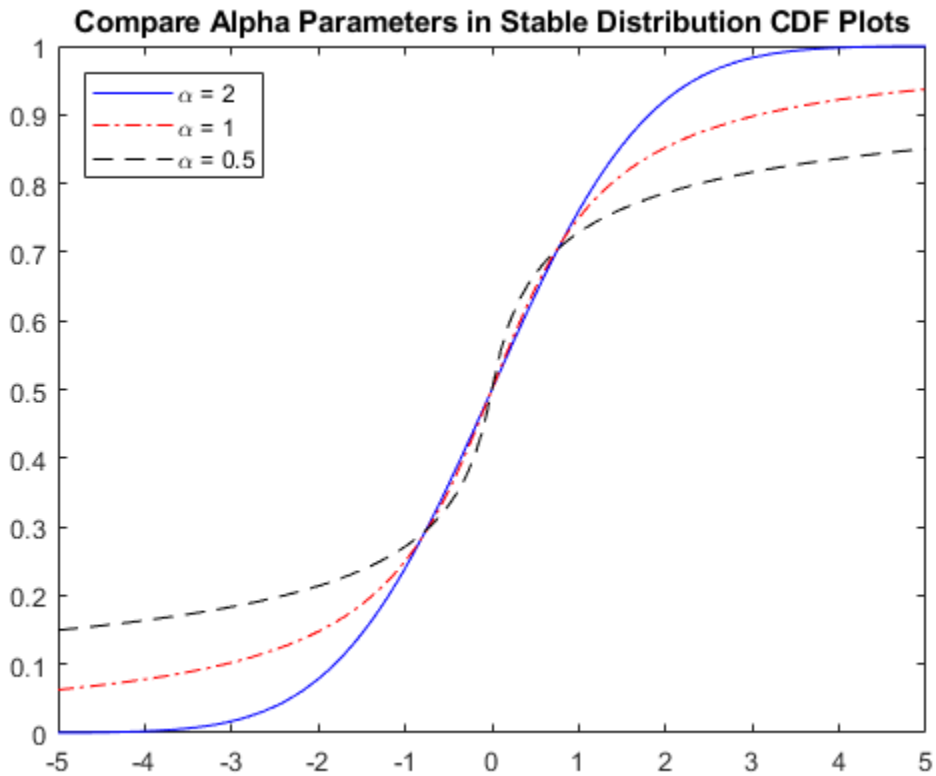
```
pd1 = makedist('Stable','alpha',2,'beta',0,'gam',1,'delta',0);
pd2 = makedist('Stable','alpha',1,'beta',0,'gam',1,'delta',0);
pd3 = makedist('Stable','alpha',0.5,'beta',0,'gam',1,'delta',0);
```

Calculate the cdf for each distribution.

```
x = -5:.1:5;
cdf1 = cdf(pd1,x);
cdf2 = cdf(pd2,x);
cdf3 = cdf(pd3,x);
```

Plot all three cdf functions on the same figure for visual comparison.

```
figure
plot(x,cdf1,'b-');
hold on
plot(x,cdf2,'r-.');
plot(x,cdf3,'k--');
title('Compare Alpha Parameters in Stable Distribution CDF Plots')
legend('\alpha = 2','\alpha = 1','\alpha = 0.5','Location','northwest')
hold off
```



The plot illustrates the effect of the alpha parameter on the shape of the cdf.

The next plot compares the cumulative distribution functions for stable distributions with different beta values. In all cases, alpha = 0.5, gam = 1, and delta = 0.

```
pd1 = makedist('Stable','alpha',0.5,'beta',0,'gam',1,'delta',0);
pd2 = makedist('Stable','alpha',0.5,'beta',0.5,'gam',1,'delta',0);
pd3 = makedist('Stable','alpha',0.5,'beta',1,'gam',1,'delta',0);
```

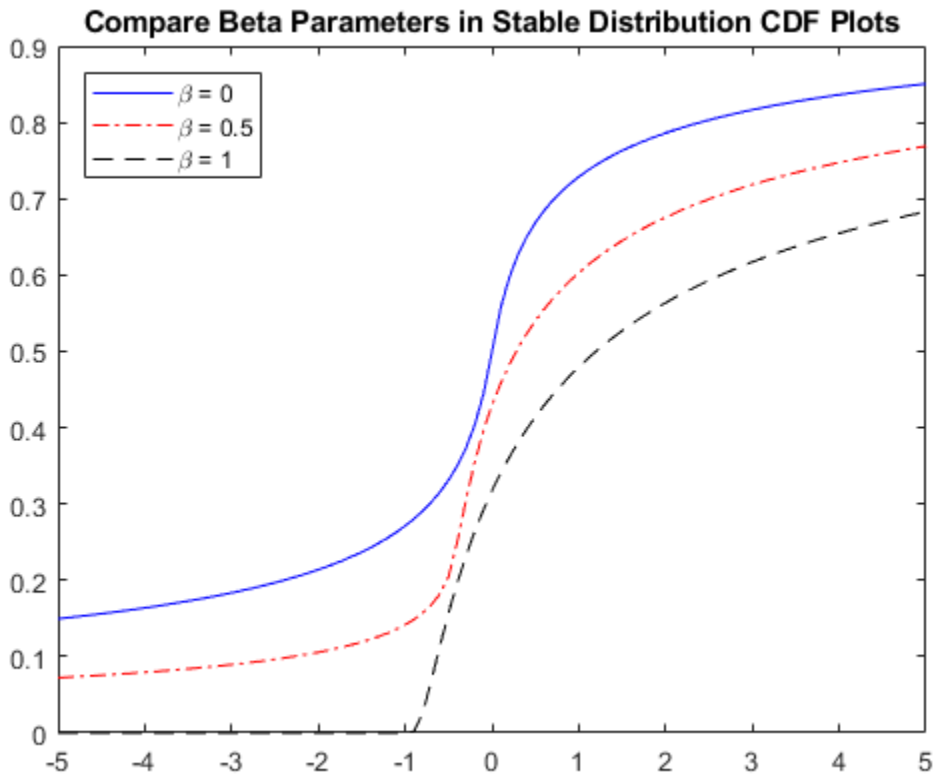
Calculate the cdf for each distribution.

```
x = -5:.1:5;
cdf1 = cdf(pd1,x);
```

```
cdf2 = cdf(pd2,x);  
cdf3 = cdf(pd3,x);
```

Plot all three pdf functions on the same figure for visual comparison.

```
figure  
plot(x,cdf1,'b-');  
hold on  
plot(x,cdf2,'r-.');  
plot(x,cdf3,'k--');  
title('Compare Beta Parameters in Stable Distribution CDF Plots')  
legend('\beta = 0', '\beta = 0.5', '\beta = 1', 'Location', 'northwest')  
hold off
```



### Descriptive Statistics

The mean of the stable distribution is undefined for values of  $\alpha \leq 1$ . For  $\alpha > 1$ , the mean of the stable distribution is

$$\text{mean} = \delta - \beta\gamma \tan\left(\frac{\pi\alpha}{2}\right).$$

Use `mean` to calculate the mean of the stable distribution.

The variance of the stable distribution is undefined for values of  $\alpha < 2$ . For  $\alpha = 2$ , the variance of the stable distribution is

$$\text{var} = 2\gamma^2.$$

Use `var` to calculate the variance of the stable distribution.

## Relationship to Other Distributions

The stable distribution has three special cases: The normal distribution, the Cauchy distribution, and the Lévy distribution. These distributions are notable because they have closed-form probability density functions.

### Normal Distribution

The normal, or Gaussian, distribution is a special case of the stable distribution with  $\alpha = 2$  corresponds to the normal distribution. In other words,

$$N(\mu, \sigma^2) = S\left(2, 0, \frac{\sigma}{\sqrt{2}}, \mu\right),$$

where  $\mu$  is the mean and  $\sigma$  is the standard deviation of the normal distribution.

Although the value of  $\beta$  has no effect when  $\alpha = 2$ , the normal distribution is usually associated with  $\beta = 0$ .

The probability density function for the normal distribution is

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right), -\infty < x < \infty.$$

A plot of the density for a normal distribution is symmetric and has a bell-shaped curve.

### Cauchy Distribution

The Cauchy distribution is a special case of the stable distribution with  $\alpha = 1$  and  $\beta = 0$ . In other words,

$$\text{Cauchy}(\delta, \gamma) = S(1, 0, \gamma, \delta),$$

where  $\gamma$  is the scale parameter and  $\delta$  is location parameter of the Cauchy distribution.

The probability density function for the Cauchy distribution is

$$f(x) = \frac{1}{\pi} \frac{\gamma}{\gamma^2 + (x - \delta)^2}, -\infty < x < \infty.$$

A plot of the density for a Cauchy distribution is symmetric and has a bell-shaped curve, but has heavier tails than the density of a normal distribution.

### Lévy Distribution

The Lévy distribution is a special case of the stable distribution where  $\alpha = 0.5$  and  $\beta = 1$ . In other words,

$$L\text{évy}(\delta, \gamma) = S(0.5, 1, \gamma, \gamma + \delta).$$

where  $\gamma$  is the scale parameter and  $\delta$  is location parameter of the Lévy distribution.

The probability density function for the Lévy distribution is

$$f(x) = \sqrt{\frac{\gamma}{2\pi}} \frac{1}{(x - \delta)^{3/2}} \exp\left(-\frac{\gamma}{2(x - \delta)}\right), \delta < x < \infty.$$

A plot of the density for a Lévy distribution is highly skewed and has heavy tails.

### Comparison Plot for Stable Distributions

The following plot compares the probability density functions for the standard normal, Cauchy, and Lévy distributions.

Create a probability distribution object for the standard normal, Cauchy, and Lévy distributions.

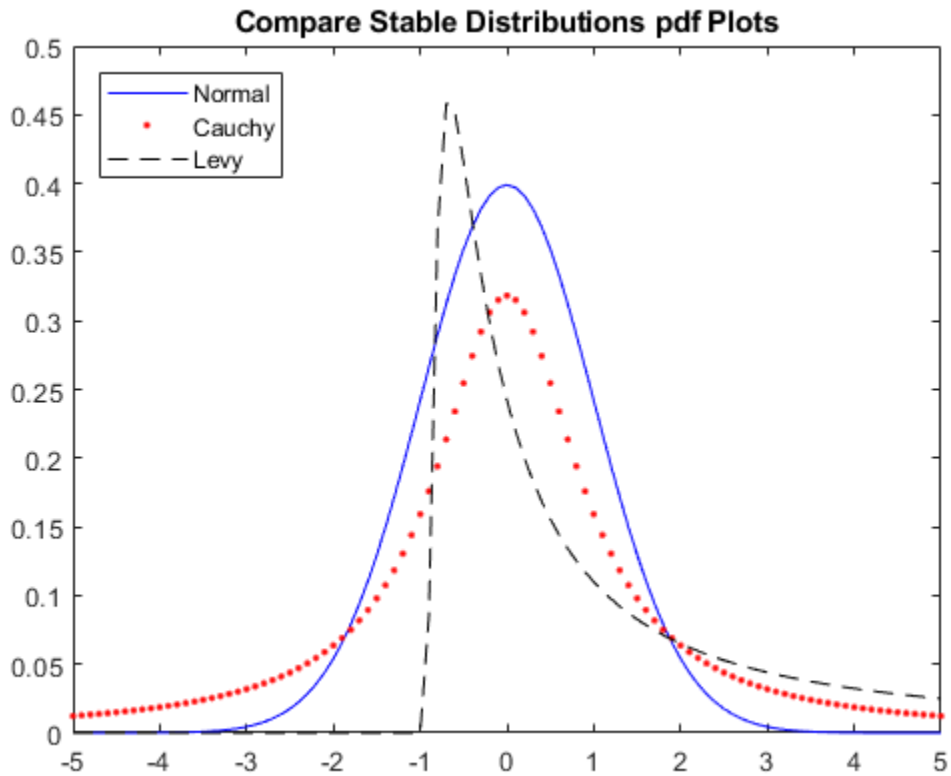
```
pd_norm = makedist('Stable', 'alpha', 2, 'beta', 0, 'gam', 1/sqrt(2), 'delta', 0);  
pd_cauchy = makedist('Stable', 'alpha', 1, 'beta', 0, 'gam', 1, 'delta', 0);  
pd_levy = makedist('Stable', 'alpha', 0.5, 'beta', 1, 'gam', 1, 'delta', 0);
```

Calculate the pdf for each distribution.

```
x = -5:.1:5;  
pdf_norm = pdf(pd_norm, x);  
pdf_cauchy = pdf(pd_cauchy, x);  
pdf_levy = pdf(pd_levy, x);
```

Plot all three pdf functions on the same figure for visual comparison.

```
figure
plot(x,pdf_norm,'b-');
hold on
plot(x,pdf_cauchy,'r. ');
plot(x,pdf_levy,'k--');
title('Compare Stable Distributions pdf Plots')
legend('Normal','Cauchy','Levy','Location','northwest')
hold off
```



## References

- [1] Nolan, J.P. “Numerical calculation of stable densities and distribution functions.” *Communications in Statistics: Stochastic Models*. Vol. 13, No. 4, 1997, pp. 759–774.
- [2] Nolan, J.P. *Stable Distributions Models for Heavy Tailed Data*. 2015. Note: In progress online.
- [3] Weron, A. and R. Weron. “Computer simulation of Lévy  $\alpha$ -stable variables and processes.” *Lecture Notes in Physics*. Vol. 457, 1995, pp. 379–392.



## See Also

StableDistribution

### More About

- “Working with Probability Distributions” on page 5-3
- “Supported Distributions” on page 5-17

## Student's *t* Distribution

In this section...
“Overview” on page B-166
“Parameters” on page B-166
“Probability Density Function” on page B-166
“Cumulative Distribution Function” on page B-169
“Mean and Variance” on page B-171
“Example” on page B-172

### Overview

The Student's *t* distribution is a family of curves depending on a single parameter  $\nu$  (the degrees of freedom).

### Parameters

The Student's *t* distribution uses the following parameter.

Parameter	Description
$\nu = 1, 2, 3, \dots$	Degrees of freedom

### Probability Density Function

#### Definition

The probability density function (pdf) of the Student's *t* distribution is

$$y = f(x | \nu) = \frac{\Gamma\left(\frac{\nu+1}{2}\right)}{\Gamma\left(\frac{\nu}{2}\right)} \frac{1}{\sqrt{\nu\pi}} \frac{1}{\left(1 + \frac{x^2}{\nu}\right)^{\frac{\nu+1}{2}}}$$

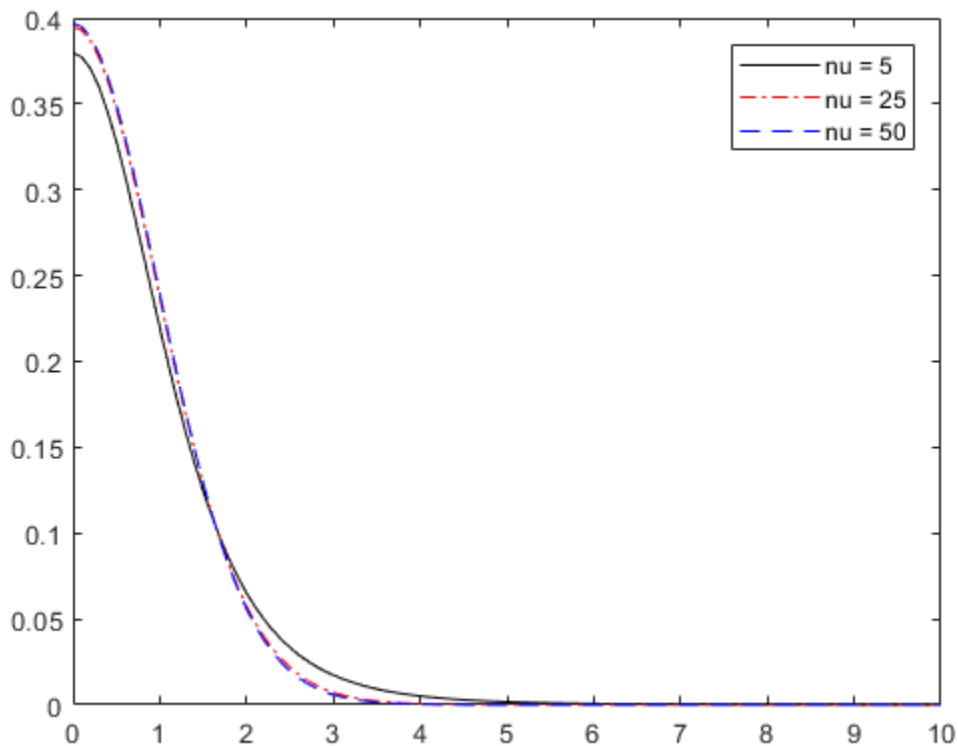
where  $\nu$  is the degrees of freedom and  $\Gamma(\cdot)$  is the Gamma function. The result  $y$  is the probability of observing a particular value of  $x$  from a Student's *t* distribution with  $\nu$  degrees of freedom.

## Plot

This plot shows how changing the value of the degrees of freedom parameter  $\nu$  alters the shape of the pdf. Use `tpdf` to compute the pdf for values  $x$  equals 0 through 10, for three different values of  $\nu$ . Then plot all three pdfs on the same figure for a visual comparison.

```
x = [0:.1:10];
y1 = tpdf(x,5);    % For nu = 5
y2 = tpdf(x,25);  % For nu = 25
y3 = tpdf(x,50);  % For nu = 50

figure;
plot(x,y1,'Color','black','LineStyle','-')
hold on
plot(x,y2,'Color','red','LineStyle','-')
plot(x,y3,'Color','blue','LineStyle','--')
legend({'nu = 5','nu = 25','nu = 50'})
hold off
```



### Random Number Generation

Use `trnd` to generate random numbers from the Student's  $t$  distribution. For example, the following generates a random number from a Student's  $t$  distribution with degrees of freedom  $\nu$  equal to 10.

```
nu = 10;  
r = trnd(nu)
```

```
r =
```

```
1.0585
```

## Relationship to Other Distributions

As the degrees of freedom  $\nu$  goes to infinity, the  $t$  distribution approaches the standard normal distribution.

If  $x$  is a random sample of size  $n$  from a normal distribution with mean  $\mu$ , then the statistic

$$t = \frac{\bar{x} - \mu}{s / \sqrt{n}}$$

where  $\bar{x}$  is the sample mean and  $s$  is the sample standard deviation, has Student's  $t$  distribution with  $n - 1$  degrees of freedom.

The Cauchy distribution is a Student's  $t$  distribution with degrees of freedom  $\nu$  equal to 1. The Cauchy distribution has an undefined mean and variance.

## Cumulative Distribution Function

### Definition

The cumulative distribution function (cdf) of Student's  $t$  distribution is

$$p = F(x | \nu) = \int_{-\infty}^x \frac{\Gamma\left(\frac{\nu+1}{2}\right)}{\Gamma\left(\frac{\nu}{2}\right)} \frac{1}{\sqrt{\nu\pi}} \frac{1}{\left(1 + \frac{t^2}{\nu}\right)^{\frac{\nu+1}{2}}} dt$$

where  $\nu$  is the degrees of freedom and  $\Gamma(\cdot)$  is the Gamma function. The result  $p$  is the probability that a single observation from the  $t$  distribution with  $\nu$  degrees of freedom will fall in the interval  $[-\infty, x]$ .

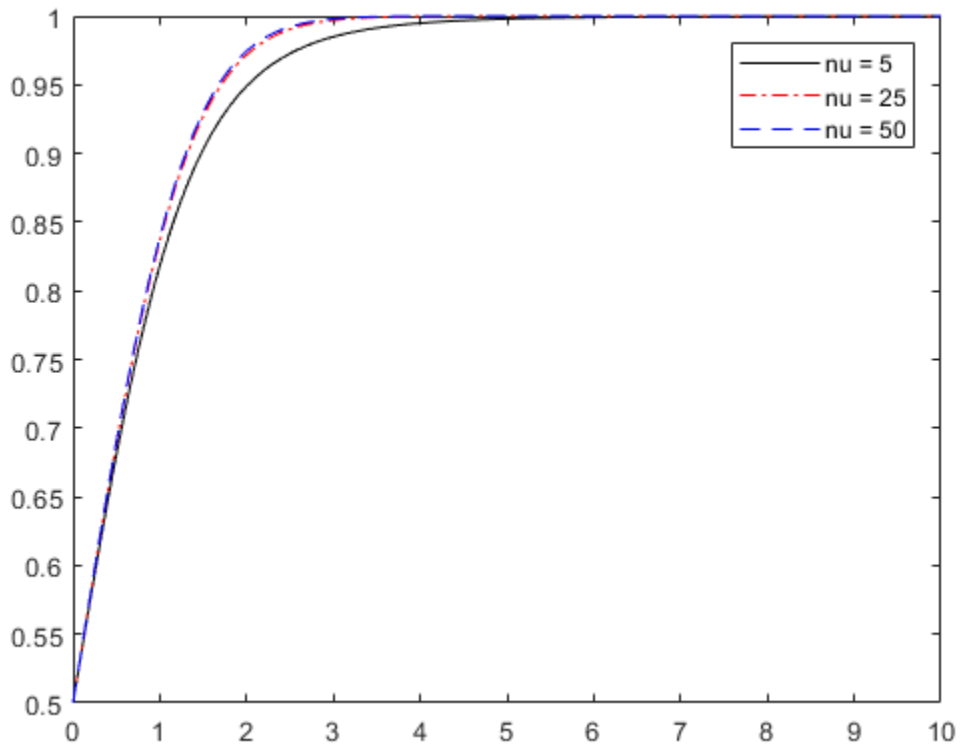
### Plot

This plot shows how changing the value of the parameter  $\nu$  alters the shape of the cdf. Use `tcdf` to compute the cdf for values  $x$  equals 0 through 10, for three different values of  $\nu$ . Then plot all three cdfs on the same figure for a visual comparison.

```
x = [0:.1:10];
y1 = tcdf(x,5); % For nu = 5
```

```
y2 = tcdf(x,25); % For nu = 25
y3 = tcdf(x,50); % For nu = 50

figure;
plot(x,y1,'Color','black','LineStyle','-')
hold on
plot(x,y2,'Color','red','LineStyle','-')
plot(x,y3,'Color','blue','LineStyle','-')
legend({'nu = 5','nu = 25','nu = 50'})
hold off
```



### Inverse cdf

Use `tinvcdf` to compute the inverse cdf of the Student's  $t$  distribution.

```
p = .95;  
nu = 50;  
x = tinv(p, nu)
```

```
x =
```

```
1.6759
```

## Mean and Variance

The mean of the Student's *t* distribution is

mean = 0

for degrees of freedom  $\nu$  greater than 1. If  $\nu$  equals 1, then the mean is undefined.

The variance of the Student's *t* distribution is

$$\text{var} = \frac{\nu}{\nu - 2}$$

for degrees of freedom  $\nu$  greater than 2. If  $\nu$  is less than or equal to 2, then the variance is undefined.

Use `tstat` to compute the mean and variance of a Student's *t* distribution. For example, the following computes the mean and variance of a Student's *t* distribution with degrees of freedom  $\nu$  equal to 10.

```
nu = 10;  
[m, v] = tstat(nu)
```

```
m =
```

```
0
```

```
v =
```

```
1.2500
```

## Example

### Compare Student's $t$ and Normal Distribution pdfs

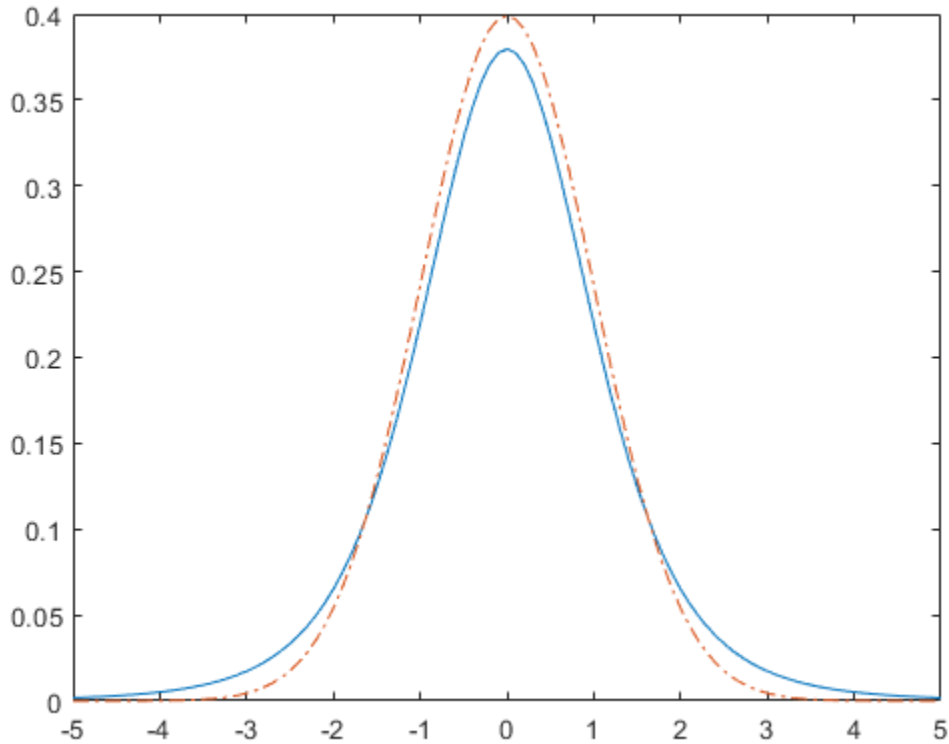
Compute the pdf for a Student's  $t$  distribution with parameter  $\nu = 5$ , and for a standard normal distribution.

```
x = -5:0.1:5;  
y = tpdf(x,5);  
z = normpdf(x,0,1);
```

Plot the Student's  $t$  and standard normal pdfs on the same figure. The standard normal pdf (dashed line) has shorter tails than the Student's  $t$  pdf (solid line).

```
figure;  
plot(x,y,'-',x,z,'-.')
```





## See Also

`random` | `tcdf` | `tinvs` | `tpdf` | `trnd` | `tstat`

## Related Examples

- “Generate Cauchy Random Numbers Using Student's  $t$ ” on page 5-140

## More About

- “Working with Probability Distributions” on page 5-3

- “Supported Distributions” on page 5-17

## t Location-Scale Distribution

In this section...
“Overview” on page B-175
“Parameters” on page B-175
“Probability Density Function” on page B-175
“Cumulative Distribution Function” on page B-176
“Descriptive Statistics” on page B-176
“Relationship to Other Distributions” on page B-177

### Overview

The  $t$  location-scale distribution is useful for modeling data distributions with heavier tails (more prone to outliers) than the normal distribution. It approaches the normal distribution as  $\nu$  approaches infinity, and smaller values of  $\nu$  yield heavier tails.

### Parameters

The  $t$  location-scale distribution uses the following parameters.

Parameter	Description	Support
$\mu$	Location parameter	$-\infty < \mu < \infty$
$\sigma$	Scale parameter	$\sigma > 0$
$\nu$	Shape parameter	$\nu > 0$

To estimate distribution parameters, use `mle`. Alternatively, fit a `tLocationScaleDistribution` object to data using `fitdist` or the **Distribution Fitter** app.

### Probability Density Function

The probability density function (pdf) of the  $t$  location-scale distribution is

$$\frac{\Gamma\left(\frac{v+1}{2}\right)}{\sigma\sqrt{v\pi}\Gamma\left(\frac{v}{2}\right)} \left[ \frac{v + \left(\frac{x-\mu}{\sigma}\right)^2}{v} \right]^{-\left(\frac{v+1}{2}\right)}$$

where  $\Gamma(\cdot)$  is the gamma function,  $\mu$  is the location parameter,  $\sigma$  is the scale parameter, and  $v$  is the shape parameter .

To compute the probability density function, use `pdf`. Alternatively, you can create a `tLocationScaleDistribution` object using `fitdist` or `makedist`, then use the `pdf` method to work with the object.

## Cumulative Distribution Function

To compute the probability density function, use `cdf`. Alternatively, you can create a `tLocationScaleDistribution` object using `fitdist` or `makedist`, then use the `cdf` method to work with the object.

## Descriptive Statistics

The mean of the *t* location-scale distribution is  
 $\text{mean} = \mu$  ,

where  $\mu$  is the location parameter. The mean is only defined for shape parameter values  $v > 1$ . For other values of  $v$ , the mean is undefined.

The variance of the *t* location-scale distribution is

$$\text{var} = \sigma^2 \frac{v}{v-2} ,$$

where  $\mu$  is the location parameter and  $v$  is the shape parameter. The variance is only defined for values of  $v > 2$ . For other values of  $v$ , the variance is undefined.

To compute the mean and variance, create a `tLocationScaleDistribution` object using `fitdist` or `makedist`. You can also use the **Distribution Fitter** app.

## Relationship to Other Distributions

If  $x$  has a  $t$  location-scale distribution, with parameters  $\mu$ ,  $\sigma$ , and  $\nu$ , then

$$\frac{x - \mu}{\sigma}$$

has a Student's  $t$  distribution with  $\nu$  degrees of freedom.

## See Also

`tLocationScaleDistribution`

## Related Examples

- “Represent Cauchy Distribution Using  $t$  Location-Scale” on page 5-136

## More About

- “Working with Probability Distributions” on page 5-3
- “Supported Distributions” on page 5-17

## Triangular Distribution

In this section...
“Overview” on page B-178
“Parameters” on page B-178
“Probability Density Function” on page B-179
“Cumulative Distribution Function” on page B-180
“Descriptive Statistics” on page B-182

### Overview

The triangular distribution provides a simplistic representation of the probability distribution when limited sample data is available. Its parameters are the minimum, maximum, and peak of the data. Common applications include business and economic simulations, project management planning, natural phenomena modeling, and audio dithering.

### Parameters

The triangular distribution uses the following parameters.

Parameter	Description	Constraints
$a$	Lower limit	$a \leq b$
$b$	Peak location	$a \leq b \leq c$
$c$	Upper limit	$c \geq b$

### Parameter Estimation

Typically, you estimate triangular distribution parameters using subjectively reasonable values based on the sample data. You can estimate the lower and upper limit parameters  $a$  and  $c$  using the minimum and maximum values of the sample data, respectively. You can estimate the peak location parameter  $b$  using the sample mean, median, mode, or any other subjectively reasonable estimate of the population mode.

## Probability Density Function

The probability density function (pdf) of the triangular distribution is

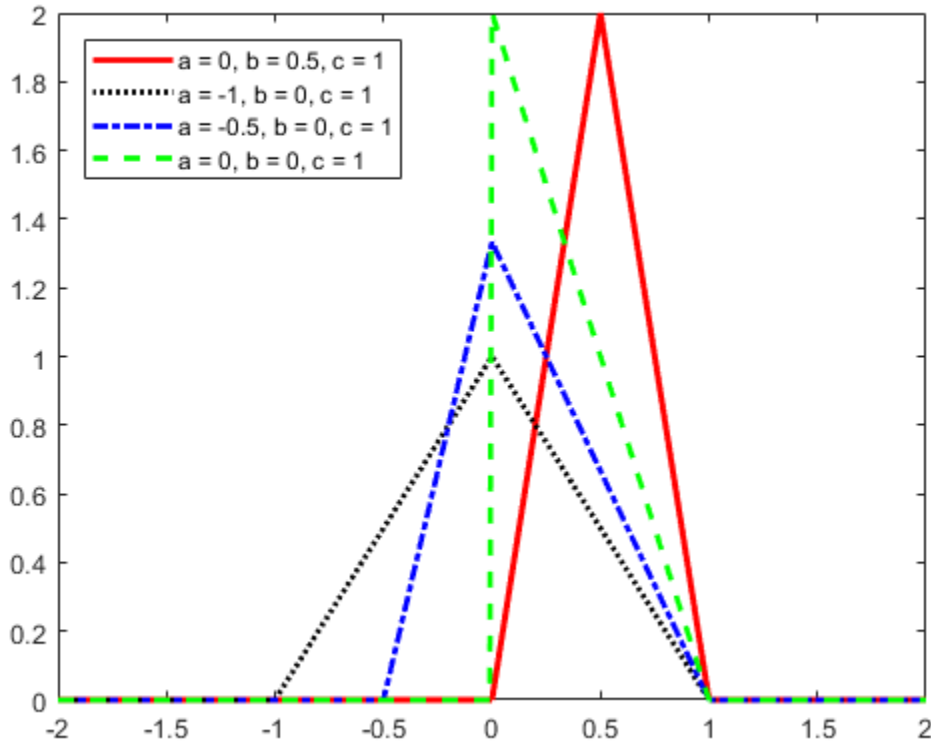
$$f(x | a, b, c) = \begin{cases} \frac{2(x-a)}{(c-a)(b-a)} & ; a \leq x \leq b \\ \frac{2(c-x)}{(c-a)(c-b)} & ; b < x \leq c \\ 0 & ; x < a, x > c \end{cases} .$$

This plot shows how changing the value of the parameters  $a$ ,  $b$ , and  $c$  alters the shape of the pdf.

```
% Create four distribution objects with different parameters
pd1 = makedist('Triangular');
pd2 = makedist('Triangular','a',-1,'b',0,'c',1);
pd3 = makedist('Triangular','a',-.5,'b',0,'c',1);
pd4 = makedist('Triangular','a',0,'b',0,'c',1);

% Compute the pdfs
x = -2:.01:2;
pdf1 = pdf(pd1,x);
pdf2 = pdf(pd2,x);
pdf3 = pdf(pd3,x);
pdf4 = pdf(pd4,x);

% Plot the pdfs
figure;
plot(x,pdf1,'r','LineWidth',2)
hold on;
plot(x,pdf2,'k:','LineWidth',2);
plot(x,pdf3,'b-.','LineWidth',2);
plot(x,pdf4,'g--','LineWidth',2);
legend({'a = 0, b = 0.5, c = 1','a = -1, b = 0, c = 1',...
'a = -0.5, b = 0, c = 1','a = 0, b = 0, c = 1'},'Location','NW');
hold off;
```



As the distance between  $a$  and  $c$  increases, the density at any particular value within the distribution boundaries decreases. Because the density function integrates to 1, the height of the pdf plot decreases as its width increases. The location of the peak parameter  $b$  determines whether the pdf skews right or left, or if it is symmetrical.

## Cumulative Distribution Function

The cumulative distribution function (cdf) of the triangular distribution is



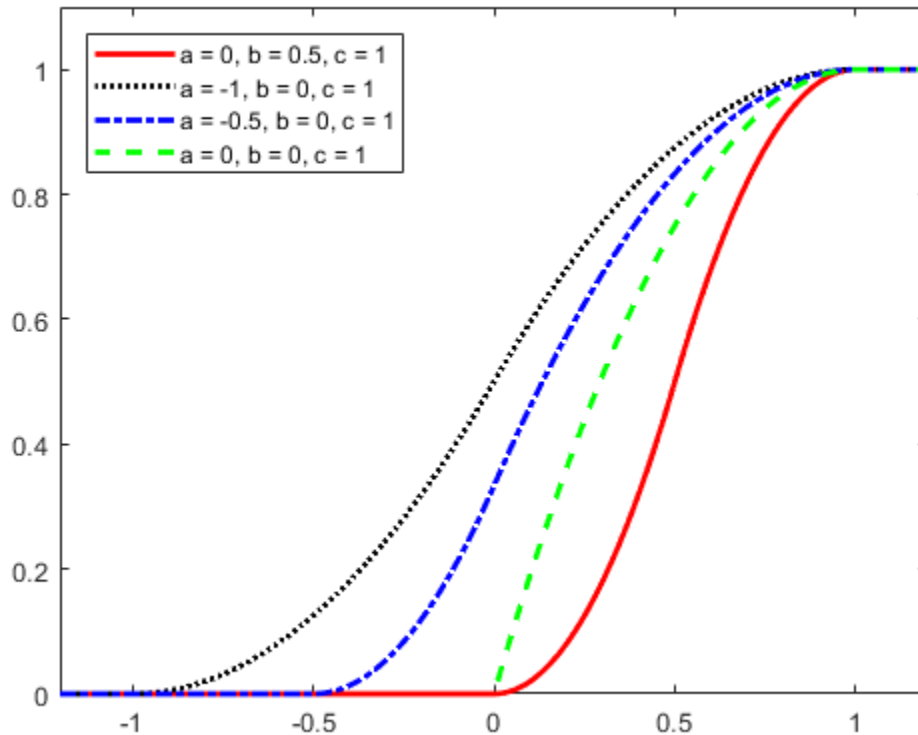
$$F(x | a, b, c) = \begin{cases} 0 & , x < a \\ \frac{(x-a)^2}{(c-a)(b-a)} & , a \leq x \leq b \\ 1 - \frac{(c-x)^2}{(c-a)(c-b)} & , b < x \leq c \\ 1 & , x > c \end{cases} .$$

This plot shows how changing the value of the parameters  $a$ ,  $b$ , and  $c$  alters the shape of the cdf.

```
% Create four distribution objects with different parameters
pd1 = makedist('Triangular');
pd2 = makedist('Triangular','a',-1,'b',0,'c',1);
pd3 = makedist('Triangular','a',-.5,'b',0,'c',1);
pd4 = makedist('Triangular','a',0,'b',0,'c',1);

% Compute the cdfs
x = -1.2:.01:1.2;
cdf1 = cdf(pd1,x);
cdf2 = cdf(pd2,x);
cdf3 = cdf(pd3,x);
cdf4 = cdf(pd4,x);

% Plot the cdfs
figure;
plot(x,cdf1,'r','LineWidth',2)
xlim([-1.2 1.2]);
ylim([0 1.1]);hold on;
plot(x,cdf2,'k:','LineWidth',2);
plot(x,cdf3,'b-.','LineWidth',2);
plot(x,cdf4,'g--','LineWidth',2);
legend({'a = 0, b = 0.5, c = 1','a = -1, b = 0, c = 1',...
'a = -0.5, b = 0, c = 1','a = 0, b = 0, c = 1'},'Location','NW');
hold off;
```



## Descriptive Statistics

The mean and variance of the triangular distribution are related to the parameters  $a$ ,  $b$ , and  $c$ .

The mean is

$$\text{mean} = \left( \frac{a + b + c}{3} \right).$$

The variance is

$$\text{var} = \left( \frac{a^2 + b^2 + c^2 - ab - ac - bc}{18} \right).$$

## See Also

TriangularDistribution

## Related Examples

- “Generate Random Numbers Using the Triangular Distribution” on page 5-62

## More About

- “Nonparametric and Empirical Probability Distributions” on page 5-37
- “Working with Probability Distributions” on page 5-3
- “Supported Distributions” on page 5-17

## Uniform Distribution (Continuous)

### In this section...

“Overview” on page B-184

“Parameters” on page B-184

“Probability Density Function” on page B-184

“Cumulative Distribution Function” on page B-186

“Descriptive Statistics” on page B-188

“Relationship to Other Distributions” on page B-189

### Overview

The uniform distribution (also called the rectangular distribution) is notable because it has a constant probability distribution function (pdf) between its two bounding parameters. It is appropriate for representing the distribution of round-off errors in values tabulated to a particular number of decimal places, and is used in random number generating techniques such as the inversion method.

### Parameters

The uniform distribution uses the following parameters.

Parameter	Description	Constraints
lower	Lower limit	$-\infty < \text{lower} < \text{upper}$
upper	Upper limit	$\text{lower} < \text{upper} < \infty$

### Parameter Estimation

The maximum likelihood estimator (MLE) for *lower* is the sample minimum. The MLE for *upper* is the sample maximum.

### Probability Density Function

The probability density function (pdf) of the continuous uniform distribution is

$$f(x | lower, upper) = \begin{cases} \left( \frac{1}{upper - lower} \right) & ; \quad lower \leq x \leq upper \\ 0 & ; \quad otherwise \end{cases} .$$

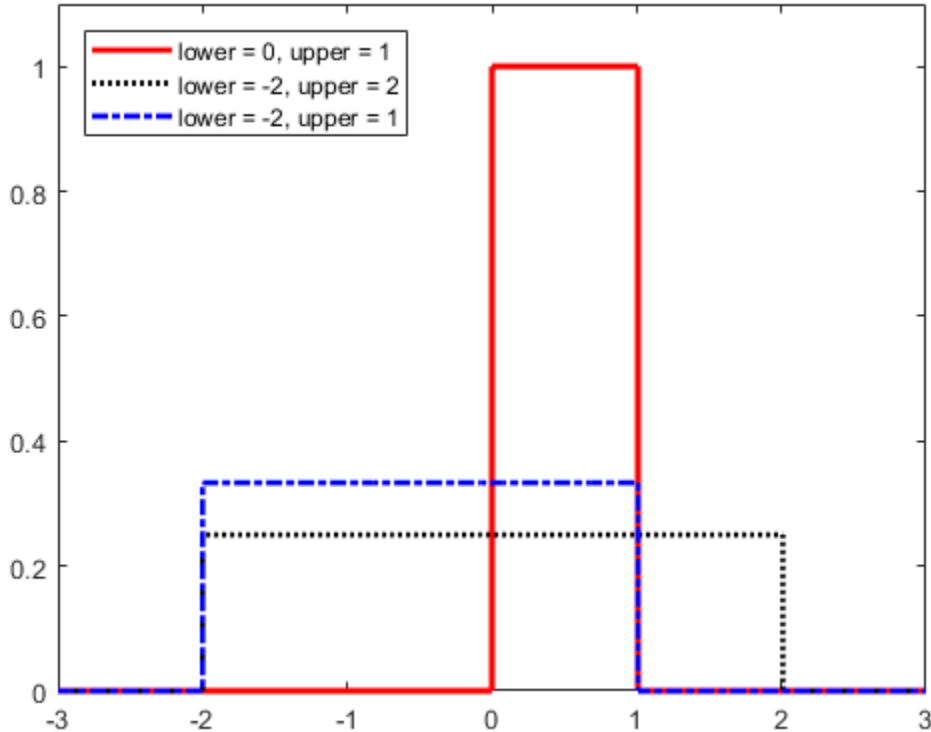
The pdf is constant between *lower* and *upper*.

This plot illustrates how changing the value of the parameters *lower* and *upper* affects the shape of the pdf.

```
% Create three distribution objects with different parameters
pd1 = makedist('Uniform');
pd2 = makedist('Uniform','lower',-2,'upper',2);
pd3 = makedist('Uniform','lower',-2,'upper',1);

% Compute the pdfs
x = -3:.01:3;
pdf1 = pdf(pd1,x);
pdf2 = pdf(pd2,x);
pdf3 = pdf(pd3,x);

% Plot the pdfs
figure;
stairs(x,pdf1,'r','LineWidth',2);
hold on;
stairs(x,pdf2,'k:','LineWidth',2);
stairs(x,pdf3,'b-.','LineWidth',2);
ylim([0 1.1]);
legend({'lower = 0, upper = 1','lower = -2, upper = 2',...
        'lower = -2, upper = 1'},'Location','NW');
hold off;
```



As the distance between *lower* and *upper* increases, the density at any particular value within the distribution boundaries decreases. Because the density function integrates to 1, the height of the pdf plot decreases as its width increases.

## Cumulative Distribution Function

The cumulative distribution function (cdf) of the continuous uniform distribution is

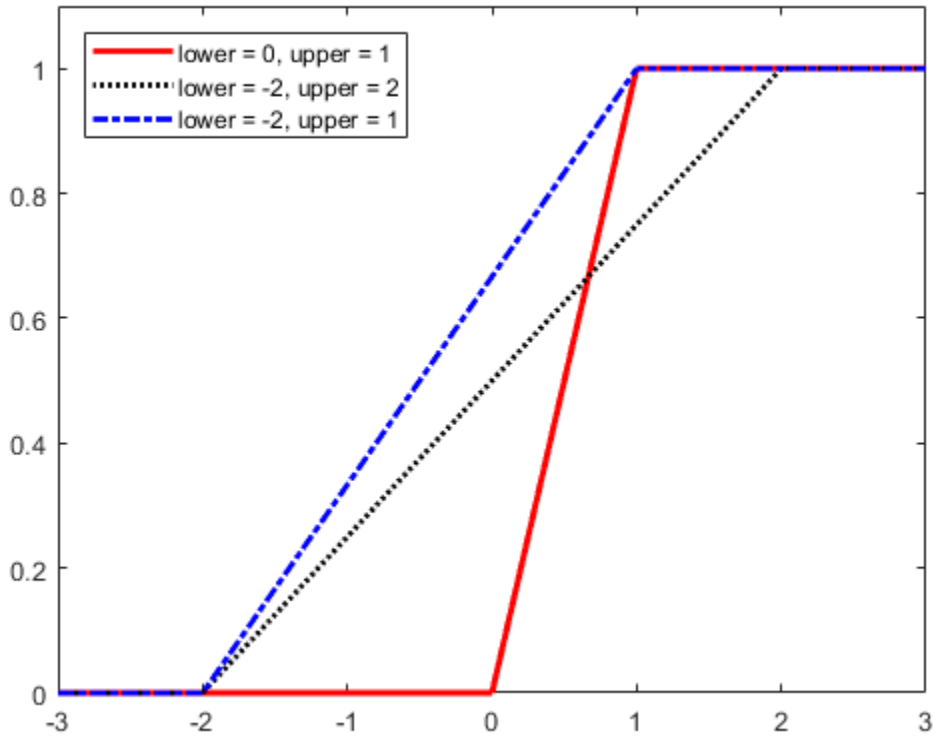
$$F(x \mid \text{lower}, \text{upper}) = \begin{cases} 0 & ; \quad x < \text{lower} \\ \frac{x - \text{lower}}{\text{upper} - \text{lower}} & ; \quad \text{lower} \leq x < \text{upper} . \\ 1 & ; \quad x \geq \text{upper} \end{cases}$$

This plot illustrates how changing the value of the parameters *lower* and *upper* affects the shape of the cdf.

```
% Create three distribution objects with different parameters
pd1 = makedist('Uniform');
pd2 = makedist('Uniform','lower',-2,'upper',2);
pd3 = makedist('Uniform','lower',-2,'upper',1);

% Compute the cdfs
x = -3:.01:3;
cdf1 = cdf(pd1,x);
cdf2 = cdf(pd2,x);
cdf3 = cdf(pd3,x);

% Plot the cdfs
figure;
plot(x,cdf1,'r','LineWidth',2);
hold on;
plot(x,cdf2,'k','LineWidth',2);
plot(x,cdf3,'b-.','LineWidth',2);
ylim([0 1.1]);
legend({'lower = 0, upper = 1','lower = -2, upper = 2',...
        'lower = -2, upper = 1'},'Location','NW');
hold off;
```



## Descriptive Statistics

The mean and variance of the continuous uniform distribution are related to the parameters *lower* and *upper*.

The mean is

$$\text{mean} = \frac{1}{2}(\text{lower} + \text{upper}).$$

The variance is

$$\text{var} = \frac{1}{12}(\text{upper} - \text{lower})^2.$$



## Relationship to Other Distributions

The standard uniform distribution (`lower = 0` and `upper = 1`) is a special case of the beta distribution on page B-4 obtained by setting the beta distribution parameters  $a = 1$  and  $b = 1$ .

The inversion method uses the continuous standard uniform distribution to generate random numbers for any other continuous distribution. The inversion method relies on the principle that continuous cumulative distribution functions (cdfs) range uniformly over the open interval  $(0,1)$ . If  $u$  is a uniform random number on  $(0,1)$ , then  $x = F^{-1}(u)$  generates a random number  $x$  from any continuous distribution with the specified cdf  $F$ .

## See Also

`UniformDistribution`

## Related Examples

- “Generate Random Numbers Using Uniform Distribution Inversion” on page 5-132

## More About

- “Working with Probability Distributions” on page 5-3
- “Supported Distributions” on page 5-17

## Uniform Distribution (Discrete)

In this section...
“Definition” on page B-190
“Background” on page B-190
“Examples” on page B-190

### Definition

The discrete uniform pdf is

$$y = f(x | N) = \frac{1}{N} I_{(1, \dots, N)}(x)$$

### Background

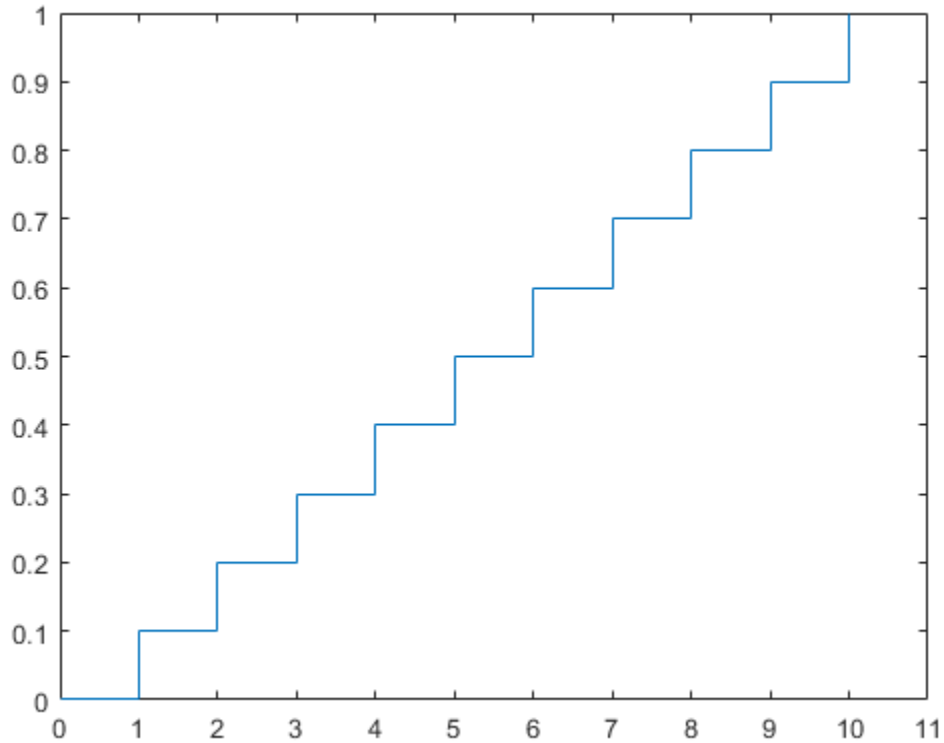
The discrete uniform distribution is a simple distribution that puts equal weight on the integers from one to  $N$ .

### Examples

#### Plot a Discrete Uniform Distribution cdf

As for all discrete distributions, the cdf is a step function. The plot shows the discrete uniform cdf for  $N = 10$ .

```
x = 0:10;  
y = unidcdf(x, 10);  
figure;  
stairs(x, y)  
h = gca;  
h.XLim = [0 11];
```



### Generate Discrete Uniform Random Numbers

Pick a random sample of 10 from a list of 553 items.

```
rng default; % for reproducibility  
numbers = unidrnd(553,1,10)
```

```
numbers =
```

451 501 71 506 350 54 155 303 530 534

## See Also

`random` | `unidcdf` | `unidinv` | `unidpdf` | `unidrnd` | `unidstat`

## More About

- “Working with Probability Distributions” on page 5-3
- “Supported Distributions” on page 5-17

## Weibull Distribution

### In this section...

“Definition” on page B-193

“Background” on page B-193

“Plot the Weibull Distribution Hazard Function” on page B-193

“Estimate Parameters of the Weibull Distribution” on page B-194

“Estimate Parameters of a Three-Parameter Weibull Distribution” on page B-195

### Definition

The Weibull pdf is positive only for positive values of  $x$ , and is zero otherwise. For strictly positive values of the shape parameter  $b$  and scale parameter  $a$ , the density is

$$f(x | a, b) = \frac{b}{a} \left( \frac{x}{a} \right)^{b-1} e^{-(x/a)^b}.$$

### Background

Waloddi Weibull offered the distribution that bears his name as an appropriate analytical tool for modeling the breaking strength of materials. Current usage also includes reliability and lifetime modeling. The Weibull distribution is more flexible than the exponential for these purposes.

To see why, consider the hazard rate function (instantaneous failure rate). If  $f(t)$  and  $F(t)$  are the pdf and cdf of a distribution, then the hazard rate is

$$h(t) = \frac{f(t)}{1 - F(t)}$$

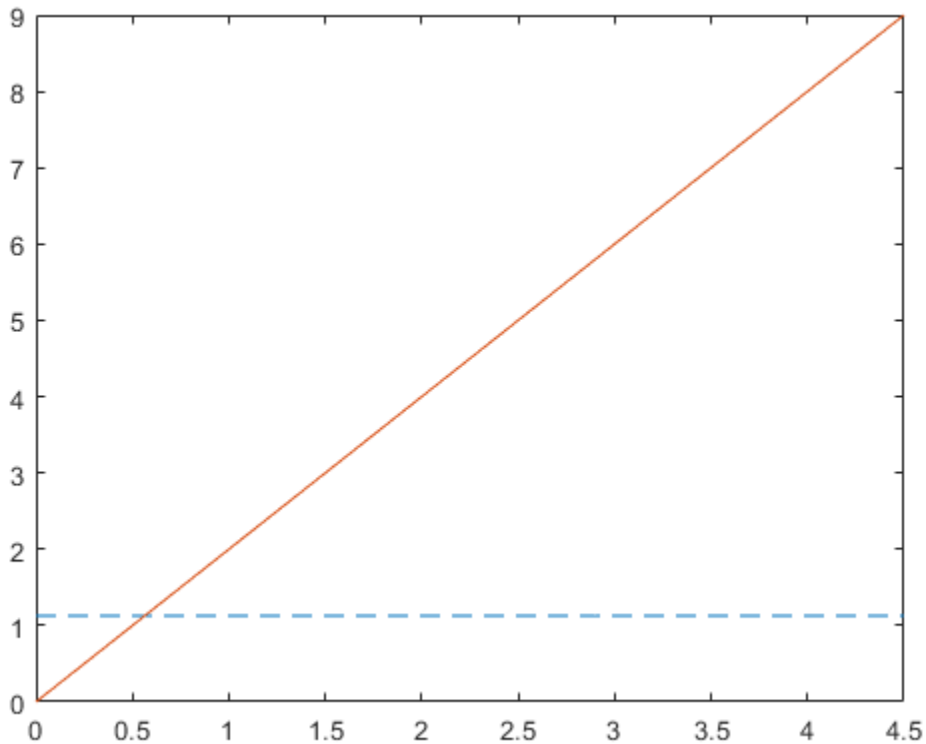
Substituting the pdf and cdf of the exponential distribution for  $f(t)$  and  $F(t)$  above yields a constant. The example below shows that the hazard rate for the Weibull distribution can vary.

### Plot the Weibull Distribution Hazard Function

The exponential distribution has a constant hazard function, which is not generally the case for the Weibull distribution. The plot shows the hazard function for exponential

(dashed line) and Weibull (solid line) distributions having the same mean life. The Weibull hazard rate here increases with age (a reasonable assumption).

```
t = 0:0.1:4.5;
h1 = exppdf(t,0.8862)/(1-expcdf(t,0.8862));
h2 = wblpdf(t,1,2)/(1-wblcdf(t,1,2));
plot(t,h1,'--',t,h2,'-')
```



## Estimate Parameters of the Weibull Distribution

Suppose you want to model the tensile strength of a thin filament using the Weibull distribution. The function `wblfit` gives maximum likelihood estimates and confidence intervals for the Weibull parameters.

```
rng('default'); % For reproducibility
strength = wblrnd(0.5,2,100,1); % Simulated strengths
[p,ci] = wblfit(strength)
```

p =

```
    0.4768    1.9622
```

ci =

```
    0.4291    1.6821
    0.5298    2.2890
```

The default 95% confidence interval for each parameter contains the true value.

## Estimate Parameters of a Three-Parameter Weibull Distribution

This example shows how to estimate parameters of a three-parameter Weibull distribution by using a custom probability density function.

Statistics and Machine Learning Toolbox™ uses a two-parameter Weibull distribution with a shape parameter  $a$  and a scale parameter  $b$ . The Weibull distribution can take one more parameter, a location parameter  $c$ . The probability density function becomes

$$f(x|a, b, c) = \begin{cases} \frac{b}{a} \left(\frac{x-c}{a}\right)^{b-1} \exp\left(-\left(\frac{x-c}{a}\right)^b\right) & \text{if } x > c, \\ 0 & \text{if } x \leq c, \end{cases}$$

where  $a$  and  $b$  are positive values, and  $c$  is a real value.

Generate sample data of size 1000 from a three-parameter Weibull distribution with the shape parameter 1, scale parameter 1, and location parameter 10.

```
rng('default') % For reproducibility
data = wblrnd(1,1,[1000,1])+10;
```

Define a probability density function for a three-parameter Weibull distribution.

```
custpdf = @(x,a,b,c) (x>c) .* (b/a) .* ((x-c)/a) .^(b-1) .* exp(-((x-c)/a) .^b);
```

The Weibull probability density function is positive only for  $x > c$ . This constraint also implies that a location parameter  $c$  is smaller than the minimum of the sample data.

Include the lower and upper bounds of parameters by using the name-value pair arguments 'LowerBound' and 'UpperBound', respectively.

```
opt = statset('MaxIter',1e5,'MaxFunEvals',1e5,'FunValCheck','off');  
phat = mle(data,'pdf',custpdf,'start',[5 5 5],'Options',opt,...  
    'LowerBound',[0 0 -Inf],'UpperBound',[Inf Inf min(data)])
```

```
phat =  
  
    1.0258    1.0618    10.0004
```

If `mle` does not converge with default statistics options, you can modify them by using the name-value pair argument 'Options'. Create a statistics options structure `opt` by using the function `statset`. Then use `opt` as the value of 'Options'.

The option `opt` includes the following:

- 'MaxIter',1e5 — Increase the maximum number of iterations to 1e5.
- 'MaxFunEvals',1e5 — Increase the maximum number of object function evaluations to 1e5.
- 'FunValCheck','off' — Turn off checking for invalid object function values.

For a distribution with a region that has zero probability density, `mle` might try some parameters that have zero density, and it will fail to estimate parameters. To avoid this problem, you can turn off the option that checks for invalid function values by using 'FunValCheck','off'.

If the scale parameter  $b$  is smaller than 1, the probability density of the Weibull distribution approaches infinity as  $x$  goes to  $c$ , where  $c$  is the location parameter. The maximum of the likelihood function is infinite. `mle` may find satisfactory estimates in some cases, but the global maximum is degenerate when  $b < 1$ .

## See Also

WeibullDistribution



## **Related Examples**

- “Compare Multiple Distribution Fits” on page 5-114

## **More About**

- “Working with Probability Distributions” on page 5-3
- “Supported Distributions” on page 5-17

# Wishart Distribution

**In this section...**

- “Overview” on page B-198
- “Parameters” on page B-198
- “Probability Density Function” on page B-198
- “Example” on page B-199

## Overview

The Wishart distribution is a generalization of the univariate chi-square distribution to two or more variables. It is a distribution for symmetric positive semidefinite matrices, typically covariance matrices, the diagonal elements of which are each chi-square random variables. In the same way as the chi-square distribution can be constructed by summing the squares of independent, identically distributed, zero-mean univariate normal random variables, the Wishart distribution can be constructed by summing the inner products of independent, identically distributed, zero-mean multivariate normal random vectors. The Wishart distribution is often used as a model for the distribution of the sample covariance matrix for multivariate normal random data, after scaling by the sample size.

Only random matrix generation is supported for the Wishart distribution, including both singular and nonsingular  $\Sigma$ .

## Parameters

The Wishart distribution is parameterized with a symmetric, positive semidefinite matrix,  $\Sigma$ , and a positive scalar degrees of freedom parameter,  $\nu$ .  $\nu$  is analogous to the degrees of freedom parameter of a univariate chi-square distribution, and  $\Sigma\nu$  is the mean of the distribution.

## Probability Density Function

The probability density function of the  $d$ -dimensional Wishart distribution is given by

$$y = f(X, \Sigma, \nu) = \frac{|X|^{((\nu-d-1)/2)} e^{\left(-\frac{1}{2}\text{trace}(\Sigma^{-1}X)\right)}}{2^{(\nu d)/2} \pi^{(d(d-1))/4} |\Sigma|^{\nu/2} \Gamma(\nu/2) \dots \Gamma(\nu-(d-1))/2}$$

where  $X$  and  $\Sigma$  are  $d$ -by- $d$  symmetric positive definite matrices, and  $v$  is a scalar greater than  $d - 1$ . While it is possible to define the Wishart for singular  $\Sigma$ , the density cannot be written as above.

## Example

If  $x$  is a bivariate normal random vector with mean zero and covariance matrix

$$\Sigma = \begin{pmatrix} 1 & .5 \\ .5 & 2 \end{pmatrix}$$

then you can use the Wishart distribution to generate a sample covariance matrix without explicitly generating  $x$  itself. Notice how the sampling variability is quite large when the degrees of freedom is small.

```
Sigma = [1 .5; .5 2];
df = 10; S1 = wishrnd(Sigma,df)/df
```

```
S1 =
    1.7959    0.64107
    0.64107    1.5496
```

```
df = 1000; S2 = wishrnd(Sigma,df)/df
```

```
S2 =
    0.9842    0.50158
    0.50158    2.1682
```

## See Also

wishrnd

## More About

- “Inverse Wishart Distribution” on page B-83
- “Working with Probability Distributions” on page 5-3
- “Supported Distributions” on page 5-17



# Bibliography

## Bibliography

- [1] Atkinson, A. C., and A. N. Donev. *Optimum Experimental Designs*. New York: Oxford University Press, 1992.
- [2] Bates, D. M., and D. G. Watts. *Nonlinear Regression Analysis and Its Applications*. Hoboken, NJ: John Wiley & Sons, Inc., 1988.
- [3] Belsley, D. A., E. Kuh, and R. E. Welsch. *Regression Diagnostics*. Hoboken, NJ: John Wiley & Sons, Inc., 1980.
- [4] Berry, M. W., et al. "Algorithms and Applications for Approximate Nonnegative Matrix Factorization." *Computational Statistics and Data Analysis*. Vol. 52, No. 1, 2007, pp. 155–173.
- [5] Bookstein, Fred L. *Morphometric Tools for Landmark Data*. Cambridge, UK: Cambridge University Press, 1991.
- [6] Bouye, E., V. Durrleman, A. Nikeghbali, G. Riboulet, and T. Roncalli. "Copulas for Finance: A Reading Guide and Some Applications." Working Paper. Groupe de Recherche Operationnelle, Credit Lyonnais, 2000.
- [7] Bowman, A. W., and A. Azzalini. *Applied Smoothing Techniques for Data Analysis*. New York: Oxford University Press, 1997.
- [8] Box, G. E. P., and N. R. Draper. *Empirical Model-Building and Response Surfaces*. Hoboken, NJ: John Wiley & Sons, Inc., 1987.
- [9] Box, G. E. P., W. G. Hunter, and J. S. Hunter. *Statistics for Experimenters*. Hoboken, NJ: Wiley-Interscience, 1978.
- [10] Bratley, P., and B. L. Fox. "ALGORITHM 659 Implementing Sobol's Quasirandom Sequence Generator." *ACM Transactions on Mathematical Software*. Vol. 14, No. 1, 1988, pp. 88–100.
- [11] Breiman, L. "Random Forests." *Machine Learning*. Vol. 4, 2001, pp. 5–32.
- [12] Breiman, L., J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Boca Raton, FL: CRC Press, 1984.
- [13] Bulmer, M. G. *Principles of Statistics*. Mineola, NY: Dover Publications, Inc., 1979.

- [14] Bury, K.. *Statistical Distributions in Engineering*. Cambridge, UK: Cambridge University Press, 1999.
- [15] Chatterjee, S., and A. S. Hadi. "Influential Observations, High Leverage Points, and Outliers in Linear Regression." *Statistical Science*. Vol. 1, 1986, pp. 379–416.
- [16] Collett, D. *Modeling Binary Data*. New York: Chapman & Hall, 2002.
- [17] Conover, W. J. *Practical Nonparametric Statistics*. Hoboken, NJ: John Wiley & Sons, Inc., 1980.
- [18] Cook, R. D., and S. Weisberg. *Residuals and Influence in Regression*. New York: Chapman & Hall/CRC Press, 1983.
- [19] Cox, D. R., and D. Oakes. *Analysis of Survival Data*. London: Chapman & Hall, 1984.
- [20] Davidian, M., and D. M. Giltinan. *Nonlinear Models for Repeated Measurements Data*. New York: Chapman & Hall, 1995.
- [21] Deb, P., and M. Sefton. "The Distribution of a Lagrange Multiplier Test of Normality." *Economics Letters*. Vol. 51, 1996, pp. 123–130.
- [22] de Jong, S. "SIMPLS: An Alternative Approach to Partial Least Squares Regression." *Chemometrics and Intelligent Laboratory Systems*. Vol. 18, 1993, pp. 251–263.
- [23] Demidenko, E. *Mixed Models: Theory and Applications*. Hoboken, NJ: John Wiley & Sons, Inc., 2004.
- [24] Delyon, B., M. Lavielle, and E. Moulines, *Convergence of a stochastic approximation version of the EM algorithm*, *Annals of Statistics*, 27, 94-128, 1999.
- [25] Dempster, A. P., N. M. Laird, and D. B. Rubin. "Maximum Likelihood from Incomplete Data via the EM Algorithm." *Journal of the Royal Statistical Society. Series B*, Vol. 39, No. 1, 1977, pp. 1–37.
- [26] Devroye, L. *Non-Uniform Random Variate Generation*. New York: Springer-Verlag, 1986.
- [27] Dobson, A. J. *An Introduction to Generalized Linear Models*. New York: Chapman & Hall, 1990.

- [28] Dunn, O.J., and V.A. Clark. *Applied Statistics: Analysis of Variance and Regression*. New York: Wiley, 1974.
- [29] Draper, N. R., and H. Smith. *Applied Regression Analysis*. Hoboken, NJ: Wiley-Interscience, 1998.
- [30] Drezner, Z. “Computation of the Trivariate Normal Integral.” *Mathematics of Computation*. Vol. 63, 1994, pp. 289–294.
- [31] Drezner, Z., and G. O. Wesolowsky. “On the Computation of the Bivariate Normal Integral.” *Journal of Statistical Computation and Simulation*. Vol. 35, 1989, pp. 101–107.
- [32] DuMouchel, W. H., and F. L. O’Brien. “Integrating a Robust Option into a Multiple Regression Computing Environment.” *Computer Science and Statistics: Proceedings of the 21st Symposium on the Interface*. Alexandria, VA: American Statistical Association, 1989.
- [33] Durbin, R., S. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis*. Cambridge, UK: Cambridge University Press, 1998.
- [34] Efron, B., and R. J. Tibshirani. *An Introduction to the Bootstrap*. New York: Chapman & Hall, 1993.
- [35] Embrechts, P., C. Klüppelberg, and T. Mikosch. *Modelling Extremal Events for Insurance and Finance*. New York: Springer, 1997.
- [36] Evans, M., N. Hastings, and B. Peacock. *Statistical Distributions*. 2nd ed., Hoboken, NJ: John Wiley & Sons, Inc., 1993, pp. 50–52, 73–74, 102–105, 147, 148.
- [37] Friedman, J. H. “Greedy function approximation: a gradient boosting machine.” *The Annals of Statistics*. Vol. 29, No. 5, 2001, pp. 1189–1232.
- [38] Genz, A. “Numerical Computation of Rectangular Bivariate and Trivariate Normal and t Probabilities.” *Statistics and Computing*. Vol. 14, No. 3, 2004, pp. 251–260.
- [39] Genz, A., and F. Bretz. “Comparison of Methods for the Computation of Multivariate t Probabilities.” *Journal of Computational and Graphical Statistics*. Vol. 11, No. 4, 2002, pp. 950–971.
- [40] Genz, A., and F. Bretz. “Numerical Computation of Multivariate t Probabilities with Application to Power Calculation of Multiple Contrasts.” *Journal of Statistical Computation and Simulation*. Vol. 63, 1999, pp. 361–378.



- [41] Gibbons, J. D. *Nonparametric Statistical Inference*. New York: Marcel Dekker, 1985.
- [42] Goldstein, A., A. Kapelner, J. Bleich, and E. Pitkin. “Peeking inside the black box: Visualizing statistical learning with plots of individual conditional expectation.” *Journal of Computational and Graphical Statistics*. Vol. 24, No. 1, 2015, pp. 44-65.
- [43] Goodall, C. R. “Computation Using the QR Decomposition.” *Handbook in Statistics*. Vol. 9, Amsterdam: Elsevier/North-Holland, 1993.
- [44] Goodnight, J.H., and F.M. Speed. *Computing Expected Mean Squares*. Cary, NC: SAS Institute, 1978.
- [45] Hahn, Gerald J., and S. S. Shapiro. *Statistical Models in Engineering*. Hoboken, NJ: John Wiley & Sons, Inc., 1994, p. 95.
- [46] Hald, A. *Statistical Theory with Engineering Applications*. Hoboken, NJ: John Wiley & Sons, Inc., 1960.
- [47] Harman, H. H. *Modern Factor Analysis*. 3rd Ed. Chicago: University of Chicago Press, 1976.
- [48] Hastie, T., R. Tibshirani, and J. H. Friedman. *The Elements of Statistical Learning*. New York: Springer, 2001.
- [49] Hill, P. D. “Kernel estimation of a distribution function.” *Communications in Statistics – Theory and Methods*. Vol. 14, Issue 3, 1985, pp. 605-620.
- [50] Hochberg, Y., and A. C. Tamhane. *Multiple Comparison Procedures*. Hoboken, NJ: John Wiley & Sons, 1987.
- [51] Hoerl, A. E., and R. W. Kennard. “Ridge Regression: Applications to Nonorthogonal Problems.” *Technometrics*. Vol. 12, No. 1, 1970, pp. 69–82.
- [52] Hoerl, A. E., and R. W. Kennard. “Ridge Regression: Biased Estimation for Nonorthogonal Problems.” *Technometrics*. Vol. 12, No. 1, 1970, pp. 55–67.
- [53] Hogg, R. V., and J. Ledolter. *Engineering Statistics*. New York: MacMillan, 1987.
- [54] Holland, P. W., and R. E. Welsch. “Robust Regression Using Iteratively Reweighted Least-Squares.” *Communications in Statistics: Theory and Methods*, A6, 1977, pp. 813–827.

- [55] Hollander, M., and D. A. Wolfe. *Nonparametric Statistical Methods*. Hoboken, NJ: John Wiley & Sons, Inc., 1999.
- [56] Hong, H. S., and F. J. Hickernell. "ALGORITHM 823: Implementing Scrambled Digital Sequences." *ACM Transactions on Mathematical Software*. Vol. 29, No. 2, 2003, pp. 95–109.
- [57] Huang, P. S., H. Avron, and T. N. Sainath, V. Sindhwani, and B. Ramabhadran. "Kernel methods match Deep Neural Networks on TIMIT." *2014 IEEE International Conference on Acoustics, Speech and Signal Processing*. 2014, pp. 205–209.
- [58] Huber, P. J. *Robust Statistics*. Hoboken, NJ: John Wiley & Sons, Inc., 1981.
- [59] Jackson, J. E. *A User's Guide to Principal Components*. Hoboken, NJ: John Wiley and Sons, 1991.
- [60] Jain, A., and R. Dubes. *Algorithms for Clustering Data*. Upper Saddle River, NJ: Prentice-Hall, 1988.
- [61] Jarque, C. M., and A. K. Bera. "A test for normality of observations and regression residuals." *International Statistical Review*. Vol. 55, No. 2, 1987, pp. 163–172.
- [62] Joe, S., and F. Y. Kuo. "Remark on Algorithm 659: Implementing Sobol's Quasirandom Sequence Generator." *ACM Transactions on Mathematical Software*. Vol. 29, No. 1, 2003, pp. 49–57.
- [63] Johnson, N., and S. Kotz. *Distributions in Statistics: Continuous Univariate Distributions-2*. Hoboken, NJ: John Wiley & Sons, Inc., 1970, pp. 130–148, 189–200, 201–219.
- [64] Johnson, N. L., N. Balakrishnan, and S. Kotz. *Continuous Multivariate Distributions*. Vol. 1. Hoboken, NJ: Wiley-Interscience, 2000.
- [65] Johnson, N. L., S. Kotz, and N. Balakrishnan. *Continuous Univariate Distributions*. Vol. 1, Hoboken, NJ: Wiley-Interscience, 1993.
- [66] Johnson, N. L., S. Kotz, and N. Balakrishnan. *Continuous Univariate Distributions*. Vol. 2, Hoboken, NJ: Wiley-Interscience, 1994.
- [67] Johnson, N. L., S. Kotz, and N. Balakrishnan. *Discrete Multivariate Distributions*. Hoboken, NJ: Wiley-Interscience, 1997.

- [68] Johnson, N. L., S. Kotz, and A. W. Kemp. *Univariate Discrete Distributions*. Hoboken, NJ: Wiley-Interscience, 1993.
- [69] Jolliffe, I. T. *Principal Component Analysis*. 2nd ed., New York: Springer-Verlag, 2002.
- [70] Jones, M.C. “Simple boundary correction for kernel density estimation.” *Statistics and Computing*. Vol. 3, Issue 3, 1993, pp. 135-146.
- [71] Jöreskog, K. G. “Some Contributions to Maximum Likelihood Factor Analysis.” *Psychometrika*. Vol. 32, 1967, pp. 443–482.
- [72] Kaufman L., and P. J. Rousseeuw. *Finding Groups in Data: An Introduction to Cluster Analysis*. Hoboken, NJ: John Wiley & Sons, Inc., 1990.
- [73] Kendall, David G. “A Survey of the Statistical Theory of Shape.” *Statistical Science*. Vol. 4, No. 2, 1989, pp. 87–99.
- [74] Klein, J. P., and M. L. Moeschberger. *Survival Analysis*. Statistics for Biology and Health. 2nd edition. Springer, 2003.
- [75] Kleinbaum, D. G., and M. Klein. *Survival Analysis*. Statistics for Biology and Health. 2nd edition. Springer, 2005.
- [76] Kocis, L., and W. J. Whiten. “Computational Investigations of Low-Discrepancy Sequences.” *ACM Transactions on Mathematical Software*. Vol. 23, No. 2, 1997, pp. 266–294.
- [77] Kotz, S., and S. Nadarajah. *Extreme Value Distributions: Theory and Applications*. London: Imperial College Press, 2000.
- [78] Krzanowski, W. J. *Principles of Multivariate Analysis: A User's Perspective*. New York: Oxford University Press, 1988.
- [79] Lawless, J. F. *Statistical Models and Methods for Lifetime Data*. Hoboken, NJ: Wiley-Interscience, 2002.
- [80] Lawley, D. N., and A. E. Maxwell. *Factor Analysis as a Statistical Method*. 2nd ed. New York: American Elsevier Publishing, 1971.
- [81] Le, Q., T. Sarlós, and A. Smola. “Fastfood — Approximating Kernel Expansions in Loglinear Time.” *Proceedings of the 30th International Conference on Machine Learning*. Vol. 28, No. 3, 2013, pp. 244–252.

- [82] Lilliefors, H. W. “On the Kolmogorov-Smirnov test for normality with mean and variance unknown.” *Journal of the American Statistical Association*. Vol. 62, 1967, pp. 399–402.
- [83] Lilliefors, H. W. “On the Kolmogorov-Smirnov test for the exponential distribution with mean unknown.” *Journal of the American Statistical Association*. Vol. 64, 1969, pp. 387–389.
- [84] Lindstrom, M. J., and D. M. Bates. “Nonlinear mixed-effects models for repeated measures data.” *Biometrics*. Vol. 46, 1990, pp. 673–687.
- [85] Little, Roderick J. A., and Donald B. Rubin. *Statistical Analysis with Missing Data*. 2nd ed., Hoboken, NJ: John Wiley & Sons, Inc., 2002.
- [86] Mardia, K. V., J. T. Kent, and J. M. Bibby. *Multivariate Analysis*. Burlington, MA: Academic Press, 1980.
- [87] Marquardt, D.W. “Generalized Inverses, Ridge Regression, Biased Linear Estimation, and Nonlinear Estimation.” *Technometrics*. Vol. 12, No. 3, 1970, pp. 591–612.
- [88] Marquardt, D. W., and R.D. Snee. “Ridge Regression in Practice.” *The American Statistician*. Vol. 29, No. 1, 1975, pp. 3–20.
- [89] Marsaglia, G., and W. W. Tsang. “A Simple Method for Generating Gamma Variables.” *ACM Transactions on Mathematical Software*. Vol. 26, 2000, pp. 363–372.
- [90] Marsaglia, G., W. Tsang, and J. Wang. “Evaluating Kolmogorov’s Distribution.” *Journal of Statistical Software*. Vol. 8, Issue 18, 2003.
- [91] Martinez, W. L., and A. R. Martinez. *Computational Statistics with MATLAB*. New York: Chapman & Hall/CRC Press, 2002.
- [92] Massey, F. J. “The Kolmogorov-Smirnov Test for Goodness of Fit.” *Journal of the American Statistical Association*. Vol. 46, No. 253, 1951, pp. 68–78.
- [93] Matousek, J. “On the L2-Discrepancy for Anchored Boxes.” *Journal of Complexity*. Vol. 14, No. 4, 1998, pp. 527–556.
- [94] McLachlan, G., and D. Peel. *Finite Mixture Models*. Hoboken, NJ: John Wiley & Sons, Inc., 2000.

- [95] McCullagh, P., and J. A. Nelder. *Generalized Linear Models*. New York: Chapman & Hall, 1990.
- [96] McGill, R., J. W. Tukey, and W. A. Larsen. "Variations of Boxplots." *The American Statistician*. Vol. 32, No. 1, 1978, pp. 12–16.
- [97] Meeker, W. Q., and L. A. Escobar. *Statistical Methods for Reliability Data*. Hoboken, NJ: John Wiley & Sons, Inc., 1998.
- [98] Meng, Xiao-Li, and Donald B. Rubin. "Maximum Likelihood Estimation via the ECM Algorithm." *Biometrika*. Vol. 80, No. 2, 1993, pp. 267–278.
- [99] Meyers, R. H., and D.C. Montgomery. *Response Surface Methodology: Process and Product Optimization Using Designed Experiments*. Hoboken, NJ: John Wiley & Sons, Inc., 1995.
- [100] Miller, L. H. "Table of Percentage Points of Kolmogorov Statistics." *Journal of the American Statistical Association*. Vol. 51, No. 273, 1956, pp. 111–121.
- [101] Milliken, G. A., and D. E. Johnson. *Analysis of Messy Data, Volume 1: Designed Experiments*. Boca Raton, FL: Chapman & Hall/CRC Press, 1992.
- [102] Montgomery, D. *Introduction to Statistical Quality Control*. Hoboken, NJ: John Wiley & Sons, 1991, pp. 369–374.
- [103] Montgomery, D. C. *Design and Analysis of Experiments*. Hoboken, NJ: John Wiley & Sons, Inc., 2001.
- [104] Mood, A. M., F. A. Graybill, and D. C. Boes. *Introduction to the Theory of Statistics*. 3rd ed., New York: McGraw-Hill, 1974. pp. 540–541.
- [105] Moore, J. *Total Biochemical Oxygen Demand of Dairy Manures*. Ph.D. thesis. University of Minnesota, Department of Agricultural Engineering, 1975.
- [106] Mosteller, F., and J. Tukey. *Data Analysis and Regression*. Upper Saddle River, NJ: Addison-Wesley, 1977.
- [107] Nelson, L. S. "Evaluating Overlapping Confidence Intervals." *Journal of Quality Technology*. Vol. 21, 1989, pp. 140–141.
- [108] Patel, J. K., C. H. Kapadia, and D. B. Owen. *Handbook of Statistical Distributions*. New York: Marcel Dekker, 1976.

- [109] Pinheiro, J. C., and D. M. Bates. “Approximations to the log-likelihood function in the nonlinear mixed-effects model.” *Journal of Computational and Graphical Statistics*. Vol. 4, 1995, pp. 12–35.
- [110] Rahimi, A., and B. Recht. “Random Features for Large-Scale Kernel Machines.” *Advances in Neural Information Processing Systems*. Vol 20, 2008, pp. 1177–1184.
- [111] Rice, J. A. *Mathematical Statistics and Data Analysis*. Pacific Grove, CA: Duxbury Press, 1994.
- [112] Rosipal, R., and N. Kramer. “Overview and Recent Advances in Partial Least Squares.” *Subspace, Latent Structure and Feature Selection: Statistical and Optimization Perspectives Workshop (SLSFS 2005), Revised Selected Papers (Lecture Notes in Computer Science 3940)*. Berlin, Germany: Springer-Verlag, 2006, pp. 34–51.
- [113] Sachs, L. *Applied Statistics: A Handbook of Techniques*. New York: Springer-Verlag, 1984, p. 253.
- [114] Scott, D. W. *Multivariate Density Estimation: Theory, Practice, and Visualization*. John Wiley & Sons, 2015.
- [115] Searle, S. R., F. M. Speed, and G. A. Milliken. “Population marginal means in the linear model: an alternative to least-squares means.” *American Statistician*. 1980, pp. 216–221.
- [116] Seber, G. A. F. and A. J. Lee. *Linear Regression Analysis*. 2nd ed. Hoboken, NJ: Wiley-Interscience, 2003.
- [117] Seber, G. A. F. *Multivariate Observations*. Hoboken, NJ: John Wiley & Sons, Inc., 1984.
- [118] Seber, G. A. F., and C. J. Wild. *Nonlinear Regression*. Hoboken, NJ: Wiley-Interscience, 2003.
- [119] Sexton, Joe, and A. R. Swensen. “ECM Algorithms that Converge at the Rate of EM.” *Biometrika*. Vol. 87, No. 3, 2000, pp. 651–662.
- [120] Silverman, B.W. *Density Estimation for Statistics and Data Analysis*. Chapman & Hall/CRC, 1986.

- [121] Snedecor, G. W., and W. G. Cochran. *Statistical Methods*. Ames, IA: Iowa State Press, 1989.
- [122] Spath, H. *Cluster Dissection and Analysis: Theory, FORTRAN Programs, Examples*. Translated by J. Goldschmidt. New York: Halsted Press, 1985.
- [123] Stein, M. “Large sample properties of simulations using latin hypercube sampling.” *Technometrics*. Vol. 29, No. 2, 1987, pp. 143–151. Correction, Vol. 32, p. 367.
- [124] Stephens, M. A. “Use of the Kolmogorov-Smirnov, Cramer-Von Mises and Related Statistics Without Extensive Tables.” *Journal of the Royal Statistical Society. Series B*, Vol. 32, No. 1, 1970, pp. 115–122.
- [125] Street, J. O., R. J. Carroll, and D. Ruppert. “A Note on Computing Robust Regression Estimates via Iteratively Reweighted Least Squares.” *The American Statistician*. Vol. 42, 1988, pp. 152–154.
- [126] Student. “On the Probable Error of the Mean.” *Biometrika*. Vol. 6, No. 1, 1908, pp. 1–25.
- [127] Velleman, P. F., and D. C. Hoaglin. *Application, Basics, and Computing of Exploratory Data Analysis*. Pacific Grove, CA: Duxbury Press, 1981.
- [128] Weibull, W. “A Statistical Theory of the Strength of Materials.” *Ingeniors Vetenskaps Akademiens Handlingar*. Stockholm: Royal Swedish Institute for Engineering Research, No. 151, 1939.
- [129] Zahn, C. T. “Graph-theoretical methods for detecting and describing Gestalt clusters.” *IEEE Transactions on Computers*. Vol. C-20, Issue 1, 1971, pp. 68–86.

